

# Appunti di Reti Logiche

## A.A. 2020/21

Questi appunti sono stati presi nell'A.A. 2020/2021, con il prof. Stea (teoria) e l'ing. Zippo (esercitazioni).

Essendo **materiale non ufficiale**, sono soggetti a sviste, errori, e soluzioni che potrebbero non essere le migliori.

Non troverete qui né le varie istruzioni Assembler, né la descrizione completa in Verilog del processore didattico sEP8, per le quali rimando ai testi ufficiali di riferimento.

Se possibile consiglio vivamente di studiare dal link  
[https://fbrogi.notion.site/Appunti-  
RL-8e5acb78a76046c68f7c8c6a49f4e5c1](https://fbrogi.notion.site/Appunti-RL-8e5acb78a76046c68f7c8c6a49f4e5c1)  
perché nella conversione da Notion a PDF sono state introdotte varie imperfezioni grafiche.

**Consiglio personale:** cercare di capire il più possibile e memorizzare il meno possibile.

Per superare l'esame c'è bisogno di padroneggiare appieno TUTTO il materiale, e ricordare non basta, perché le cose viste a lezione vanno applicate.

Buono studio ed in bocca al lupo!

Filippo Brogi

# Assembler

## Rappresentazione dei numeri nei calcolatori

Per tutta la trattazione si indicherà un **numero naturale** con una **lettera maiuscola** ( $X$ ), e un **numero intero** con una **lettera minuscola** ( $x$ ).

Su  $N$  bit sono rappresentabili  $2^N$  numeri, che essi siano **naturali** o **interi**.

Un numero NON è rappresentabile su un certo numero di bit se “sfora” **dall’intervallo di rappresentabilità**, che varia a seconda che sia un naturale o un intero.

### Numeri Naturali:

Su  $N$  bit si rappresentano i numeri compresi nell’intervallo  $[0, 2^N - 1]$ .

Vi è una corrispondenza fra **stringhe di bit** e **numeri naturali**, per cui:

- **Stringa di bit**  $(b_{N-1}, b_{N-2}, \dots, b_1, b_0) \rightarrow **Numero naturale**  $X$ :$

$$X = \sum_{i=0}^{N-1} b_i \cdot 2^i$$

- **Numero naturale**  $X \rightarrow **Stringa di bit**  $(b_{N-1}, b_{N-2}, \dots, b_1, b_0)$ :$

**Algoritmo DIV&MOD:** dividere  $X$  per 2 fintanto che non diventa 0, e una volta fatto si prendono i resti dal più recente al più lontano concatenandoli

### Numeri Zintei:

Su  $N$  bit si rappresentano i numeri compresi nell’intervallo  $[-2^{N-1}, 2^{N-1} - 1]$ .

Vi è una corrispondenza fra **stringhe di bit che corrispondono a numeri naturali** e **numeri interi**:

- **Numero intero**  $x \rightarrow **Numero naturale**  $X$  (stringa di bit):$

$$X = \begin{cases} x & \text{se } x \geq 0 \\ x + 2^N & \text{se } x < 0 \end{cases}$$

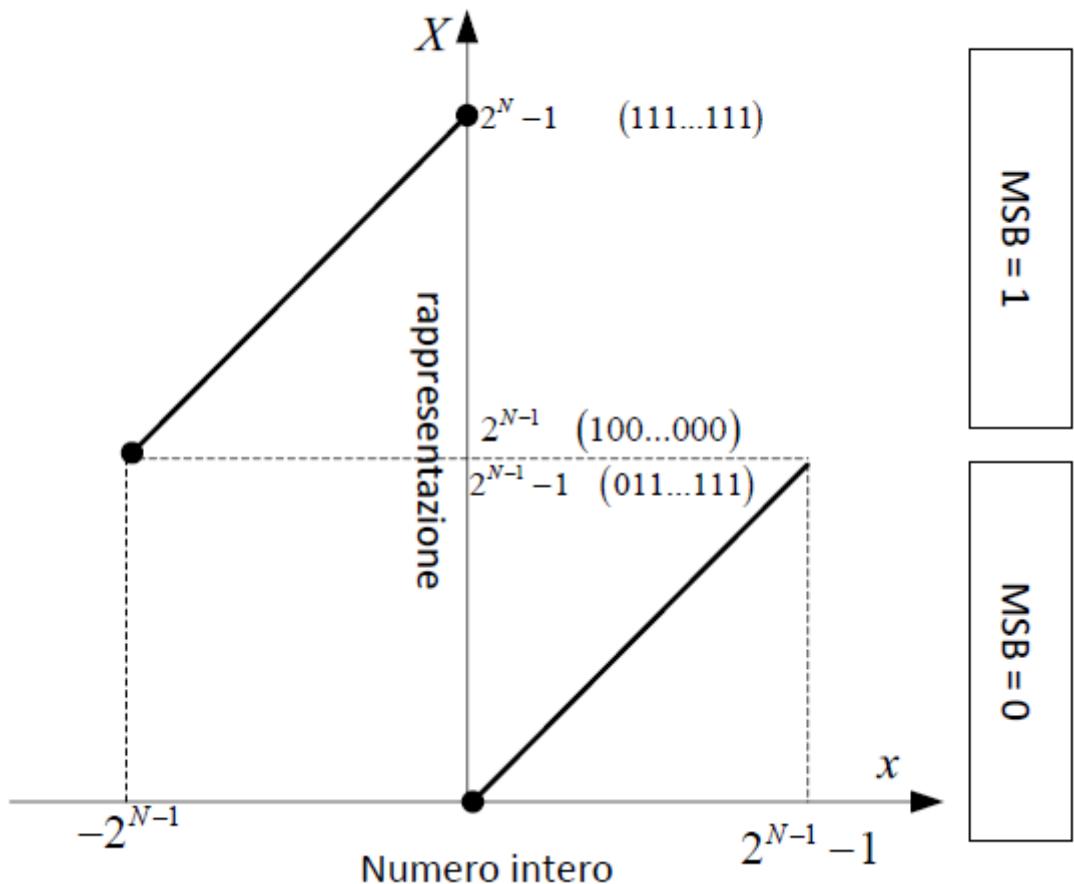
**ATTENZIONE!** Questa formula vale SOLO SE  $x$  è rappresentabile in C2 su  $N$  bit!

- **Numero naturale**  $X$  (stringa di bit) → **Numero intero**  $x$ :

$$x = \begin{cases} X & \text{se } X_{N-1} = 0 \\ -(X + 1) & \text{se } X_{N-1} = 1 \end{cases}$$

Dato un numero rappresentato dalla stringa di bit  $(b_{N-1}, b_{N-2}, \dots, b_1, b_0)$ , il bit  $b_{N-1}$  si dice **MOST SIGNIFICANT BIT (MSB)**, mentre il bit  $b_0$  si dice **LEAST SIGNIFICANT BIT (LSB)**.

$MSB = 0 \implies$  Numero intero positivo  
 $MSB = 1 \implies$  Numero intero negativo



# Codifiche delle istruzioni macchina nei calcolatori

**CODIFICA MACCHINA:** Serie di zeri ed uni che codificano le istruzioni che il processore esegue

**COFICICA MNEMONICA:** Codice simbolico human-friendly corrispondente ad una codifica macchina

**LINGUAGGIO ASSEMBLY:** Linguaggio di programmazione che usa una codifica mnemonica per le istruzioni, più una sovrastruttura per facilitare la vita al programmatore (e.g. riferire locazioni di memoria con nomi simbolici)

**ASSEMBLATORE:** Programma che traduce la codifica mnemonica in codifica macchina, per farla eseguire al processore

## Sintassi di assembler

Ogni operazione ha la seguente sintassi:

```
OPCODE sorgente, destinatario
```

Dove:

- `OPCODE` è il codice operativo dell'istruzione, ovvero la sua codifica mnemonica.
- `sorgente` è l'operando sorgente dell'istruzione. Alcune potrebbero non averlo.
- `destinatario` è l'operando destinatario dell'istruzione. Alcune potrebbero non averlo.

**Quando un'istruzione deve scrivere un risultato, e prende in input due operandi, sovrascrive l'operando `destinatario` col risultato!**

- **Per le costanti ci vuole il \$ davanti, sennò vengono interpretate come indirizzi!**

(e.g: `MOV $0x20, %EAX` muove il valore costante esadecimale '20' in EAX

`MOV 0x20, %EAX` muove il CONTENUTO della cella di memoria 00...20 in EAX)

- I numeri di default sono interpretati in base 10
- I registri vanno preceduti dal %

## Suffissi B, W, L

Vi sono 3 suffissi che consentono di specificare la lunghezza degli operandi di ciascuna istruzione:

- $B \rightarrow$  Byte, 8 bit
- $W \rightarrow$  Word, 16 bit
- $L \rightarrow$  Long, 32 bit



I suffissi sono OBBLIGATORI quando nessuno degli operandi (né sorgente né destinatario) è indirizzato con indirizzamento di registro (registro SENZA PARENTESI TONDE)

In assembler il destinatario di un'istruzione è quello che viene modificato, il sorgente no!

## Indirizzamenti

- **INDIRIZZAMENTO DI REGISTRO:** Uno o entrambi gli operandi sono registri.  
**Esempio:** `OPCODE %EAX` oppure `OPCODE %EAX, %EBX`
- **INDIRIZZAMENTO IMMEDIATO:** L'operando *sorgente* è una costante specificata direttamente nell'istruzione (attenzione al \$).  
**Esempio:** `OPCODE $0x20, %AL` (ha senso SOLO per il sorgente!)
- **INDIRIZZAMENTO DI MEMORIA:** Valido per uno solo degli operandi per volta, si divide in:
  - **DIRETTO:** L'indirizzo è specificato direttamente nell'istruzione.  
**Esempio:** `OPCODEW 0x00002010`

- **INDIRETTO (con displacement)**: L'indirizzo è calcolato tramite la seguente formula:

$$\text{indirizzo} = |BASE + (INDICE \cdot scala) \pm displacement|_{2^{32}}$$

Dove:

- $scala \in \{1, 2, 4, 8\}$ , e di default è 1
- $displacement \in \mathbb{N}$
- ***BASE e INDICE sono REGISTRI GENERALI a 32 bit!***  
(e.g. *EAX, EBX, ...*)

In Assembler questa formula viene specificata con questa sintassi:

```
OPCODE displacement(BASE, INDICE, scala)
```

**Esempio:** `OPCODEW (%EAX, %ECX, 2)`



**L'indirizzamento di memoria indiretto con displacement viene usato tipicamente per accedere a vettori di:**

- *byte (8 bit)* se  $scala = 1$
- *word (16 bit)* se  $scala = 2$
- *long (32 bit)* se  $scala = 4$



**Le istruzioni aritmetiche, logiche, e la CMP sono le uniche che modificano i flag!**

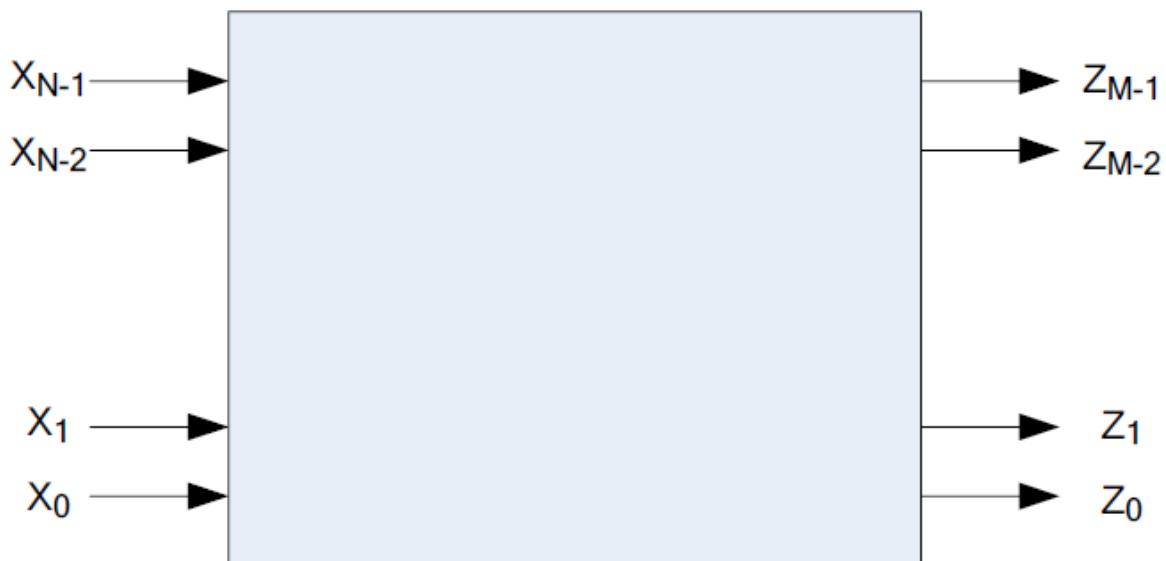
# Reti Combinatorie

## Generalità

**RETE LOGICA:** Modello astratto di un sistema fisico costituito da dispositivi interconnessi

Una **Rete Logica** è caratterizzata da 3 cose:

- $N$  **variabili di ingresso** ( $x_{N-1}, x_{N-2}, \dots, x_1, x_0$ ).  
Una specifica combinazione di valori assunta si chiama **stato di ingresso**.  
L'insieme ( $2^N$ ) di tutti i possibili stati di ingresso si indicherà con  $X$ .
- $M$  **variabili di uscita** ( $z_{M-1}, z_{M-2}, \dots, z_1, z_0$ ).  
Una specifica combinazione di valori assunta si chiama **stato di uscita**.  
L'insieme ( $2^M$ ) di tutti i possibili stati di uscita si indicherà con  $Z$ .
- Una **Legge di evoluzione temporale**  $Z = F(X)$



## Classificazione delle reti logiche

I criteri di classificazione sono 2, e sono:

- **Presenza o meno di memoria:**
  - **Reti Combinatorie:** Lo stato di uscita dipende **SOLO** dallo stato di ingresso.  
La legge  $F$  in questo caso è a tutti gli effetti una funzione.
  - **Reti Sequenziali:** Lo stato di uscita dipende dalla **STORIA DEGLI STATI DI INGRESSO**.
- **Temporizzazione di  $F$**  (la legge di evoluzione nel tempo):
  - **Reti Asincrone:** L'aggiornamento dell'uscita avviene  $\forall t$ .
  - **Reti Sincronizzate:** L'aggiornamento dell'uscita avviene in specifici istanti temporali.

	Asincrona	Sincrona
Combinatoria	<b>Reti Combinatorie</b>	
Sequenziale	<b>Reti Sequenziali Asincrone</b>	<b>Reti Sequenziali Sincronizzate</b>

## Descrizione vs Sintesi

**DESCRIZIONE:** E' un modo formale per dire cosa fa la rete (e.g. tabella di verità).

**SINTESI:** E' il progetto della realizzazione fisica della rete, in carne ed ossa, con fili e componenti elettronici, in modo tale che si comporti come specificato dalla descrizione.

## Problemi fisici nelle reti logiche

Anche se le variabili logiche transiscono istantaneamente, le variabili fisiche che esse rappresentano no, quindi ci sarà un transitorio in cui il valore della variabile logica è indeterminato.

Noi faremo l'ipotesi che il tempo del transitorio sia molto minore del tempo in cui la variabile logica sta a regime, quindi questo problema è risolto.

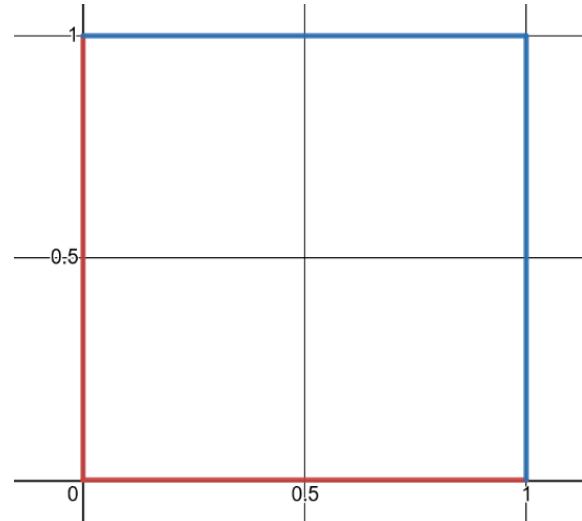
## Transizioni contemporanee

Il problema vero è invece cosa succede se faccio variare due (o più) variabili di ingresso contemporaneamente!

**Esempio:** voglio passare da  $(1, 0)$  a  $(0, 1)$ , e per farlo posso passare sia da  $(0, 0)$  che da  $(1, 1)$ , ma non so quale delle due strade prende la rete!

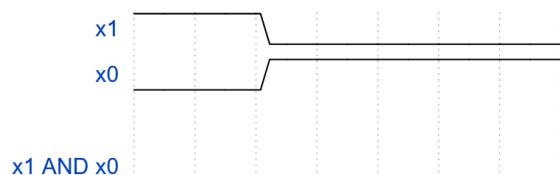
Questo problema può essere visto in due modi:

- **Matematico:** Voglio passare dal punto  $(1, 0)$  del grafico cartesiano  $(x, y)$  al punto  $(0, 1)$ . La via diretta è tagliare per la diagonale, ma potendomi spostare solo sugli assi (gli assi sono i miei gradi di libertà), inevitabilmente, finirò o col fare la strada  $(1, 0) - (0, 0) - (0, 1)$  oppure  $(1, 0) - (1, 1) - (0, 1)$

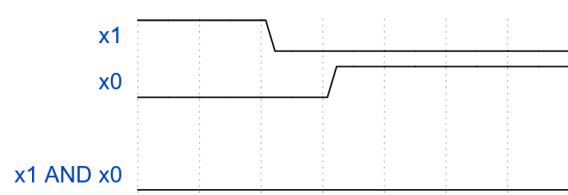


- **Elettronico:** Non potrò mai avere né la certezza né la precisione per far cambiare esattamente *nello stesso istante* due grandezze elettriche, una varierà prima, una dopo, ciascuna con transitori diversi magari. Allora quello che succede è che:

- Il segnale  $x_1$  può andare prima a 0 di quando  $x_0$  passa a 1, e quindi c'è un transitorio in cui la rete è in  $(0, 0)$ .
- Viceversa, se  $x_0$  si porta ad 1 prima che  $x_1$  abbia il tempo di

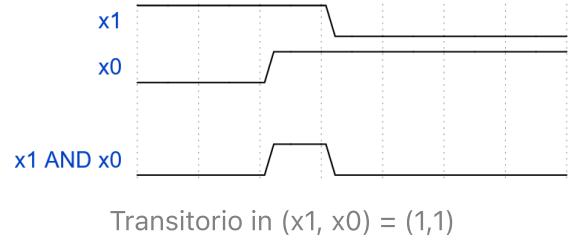


Situazione ideale, non fisicamente realizzabile



Transitorio in  $(x_1, x_0) = (0,0)$

andare a 0 la rete sta brevemente nello stato  $(1, 1)$ .



DIPENDENTEMENTE DAL TRANSITORIO LA RETE FA COSE DIVERSE!!!



Visione algebrica:

**STATI DI INGRESSO CONSECUTIVI DEVONO ESSERE ADIACENTI!**

(adiacenti significa che differiscono di un solo bit).

Visione elettronica:

**DEVE CAMBIARE UN SOLO SEGNALE PER VOLTA!**

altrimenti si crea un transitorio, ovvero una corsa fra variabili di stato, e il comportamento della rete diventa IMPREVEDIBILE per un piccolo lasso di tempo, peggio ancora se è sequenziale!

## Tempo di attraversamento

**TEMPO DI ATTRAVERSAMENTO:** E' il tempo che lo stato di uscita impiega per adeguarsi ad un cambiamento dello stato di ingresso.

Quando questo tempo è passato si dice che la rete va **A REGIME**, ed è quindi pronta per rispondere ad un'altra variazione dello stato di ingresso.



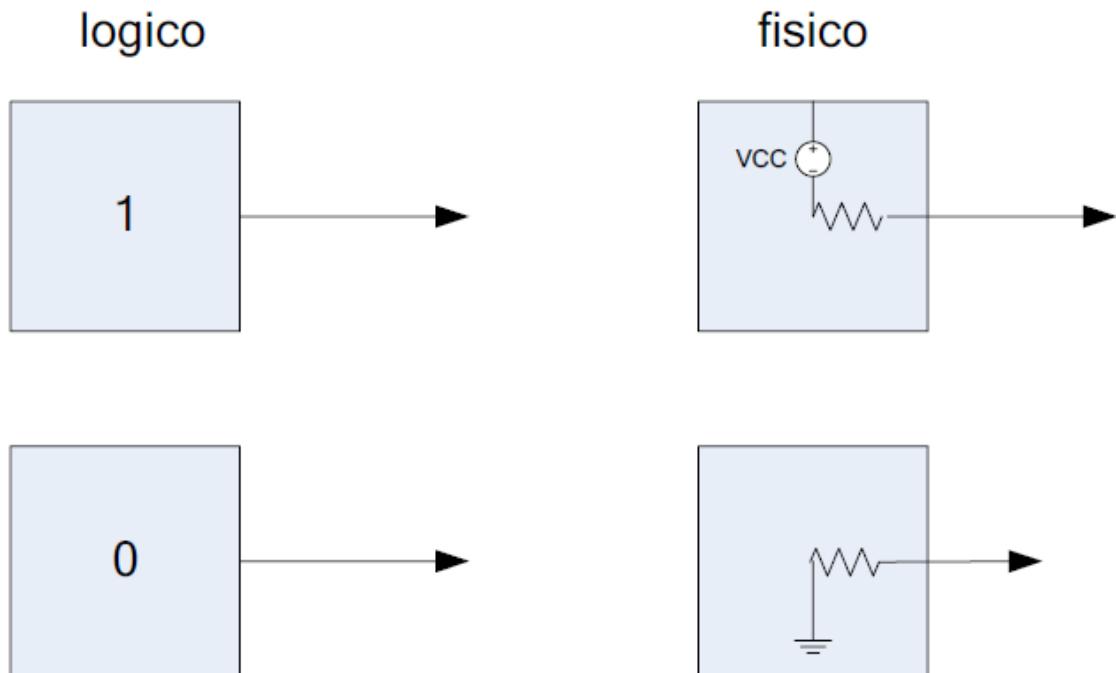
**Non puoi variare lo stato di ingresso prima che la rete sia andata a regime!**

**(RETE PILOTATA IN MODO FONDAMENTALE)**

# Reti Combinatorie Elementari

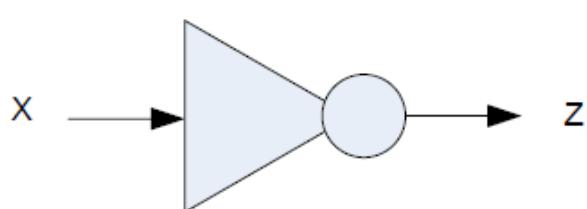
## Reti a 0 ingressi:

Si chiamano **GENERATORI DI COSTANTE** e non vi è presente alcuna logica, sono solo fili attaccati alla corrente (*vcc*) o a massa:



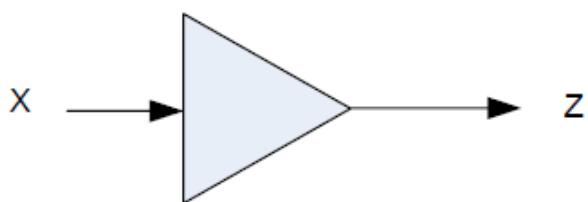
## Reti ad 1 ingresso:

### INVERTITORE:



X	Z
0	1
1	0

### ELEMENTO NEUTRO (BUFFER):



x	z
0	0
1	1

Il buffer serve a:

- **Generare ritardo** (utile per le temporizzazioni)
- **Rigenerare i segnali elettrici degradati**

Reti a 2 ingressi:

**PORTE AND:**

$x_1$	$x_0$	$z$
0	0	0
0	1	0
1	0	0
1	1	1



$$z = 1 \iff x_1 = x_0 = 1$$

**PORTE OR:**

$x_1$	$x_0$	$z$
0	0	0
0	1	1
1	0	1
1	1	1



$$z = 0 \iff x_1 = x_0 = 0$$

**PORTE XOR:**

$x_1$	$x_0$	$z$
0	0	0
0	1	1
1	0	1
1	1	0



**PORTE NAND:**

$x_1$	$x_0$	$z$
0	0	1
0	1	1
1	0	1
1	1	0

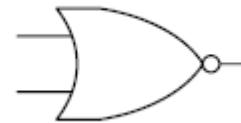
$$z = 1 \iff x_1 \neq x_0$$



**PORTE NOR:**

$x_1$	$x_0$	$z$
0	0	1
0	1	0
1	0	0
1	1	0

$$z = 0 \iff x_1 = x_0 = 1$$



**PORTE XNOR:**

$x_1$	$x_0$	$z$
0	0	1
0	1	0
1	0	0
1	1	1

$$z = 1 \iff x_1 = x_0 = 0$$

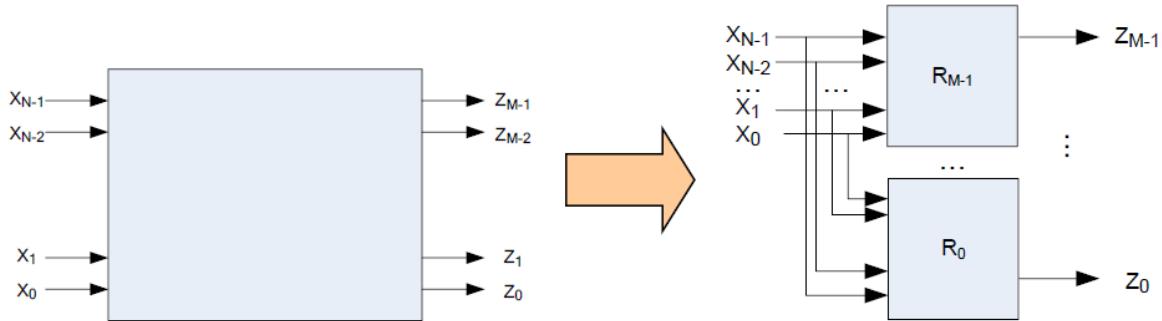


$$z = 1 \iff x_1 = x_0$$

## Reti Combinatorie a più uscite

Possiamo considerare solo reti ad un'uscita per via della seguente proprietà:

**Una Rete Combinatoria ad  $N$  ingressi ed  $M$  uscite può essere realizzata interconnettendo  $M$  Reti Combinatorie ad  $N$  ingressi ed un'uscita.**



## Algebra di Boole

L'**algebra di Boole** è costituita da:

- **Variabili logiche**, che assumono solo i valori  $\{0, 1\}$
- **Operatori logici**, che si applicano a variabili logiche

## Operatori Logici

**COMPLEMENTO:** Operatore Unario, si indica con  $\bar{x}$ .

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



**L'inverter implementa fisicamente il complemento!**

**PRODOTTO LOGICO:** Operatore Binario, si indica con  $x \cdot y$ .

$x$	$y$	$x \cdot y$
0	0	$0 \cdot 0 = 0$
0	1	$0 \cdot 1 = 0$
1	0	$1 \cdot 0 = 0$
1	1	$1 \cdot 1 = 1$



**La porta AND implementa fisicamente il prodotto logico!**

**SOMMA LOGICA:** Operatore Binario, si indica con  $x + y$ .

$x$	$y$	$x + y$
0	0	$0 + 0 = 0$
0	1	$0 + 1 = 1$
1	0	$1 + 0 = 1$
1	1	$1 + 1 = 1$



**La porta OR implementa fisicamente la somma logica!**

## Proprietà degli operatori booleani

<i>Proprietà</i>	
<i>Involutiva</i>	$\bar{\bar{x}} = x$
<i>Associativa...</i> della somma: del prodotto:	$x + y + z = (x + y) + z = x + (y + z)$ $x \cdot y \cdot z = (x \cdot y) \cdot z = x \cdot (y \cdot z)$
<i>Distributiva...</i> normale: <b>NUOVA:</b>	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$ $x + (y \cdot z) = (x + y) \cdot (x + z)$
<b>Complemento</b>	$x \cdot \bar{x} = 0$ $x + \bar{x} = 1$
<b>Idempotenza</b>	$x + x = x$ $x \cdot x = x$
<b>Elementi neutri</b>	$x + 0 = x$ $x \cdot 1 = x$
<b>Elementi assorbenti</b>	$x + 1 = 1$ $x \cdot 0 = 0$

## Teoremi di De Morgan

### **PRIMO TH. DE MORGAN:**

$$\overline{x_0 \cdot x_1 \cdot \dots \cdot x_{N-1}} = \overline{x_0} + \overline{x_1} + \dots + \overline{x_{N-1}}$$

Caso particolare  $N = 2 : \overline{x \cdot y} = \overline{x} + \overline{y}$

### **DIMOSTRAZIONE:**

**Si dimostra per induzione.**

**Passo base:** tabella di verità per  $N = 2$

$x_1$	$x_0$	$x_0 \cdot x_1$	$\overline{x_0 \cdot x_1}$	$\overline{x_0} + \overline{x_1}$
0	0	0	1	1
0	1	0	1	1
1	0	0	1	1
1	1	1	0	0

### **Passo induttivo:**

- Suppongo che  $\overline{x_0 \cdot x_1 \cdot \dots \cdot x_{N-1}} = \overline{x_0} + \overline{x_1} + \dots + \overline{x_{N-1}}$  sia vero
- Dimostro che ciò implica  $\overline{x_0 \cdot x_1 \cdot \dots \cdot x_{N-1} \cdot x_N} = \overline{x_0} + \overline{x_1} + \dots + \overline{x_{N-1}} + \overline{x_N}$

Per farlo basta semplicemente porre  $\alpha = x_0 \cdot x_1 \cdot \dots \cdot x_{N-1}$ , e quindi sostituendo nel passo base ottengo che  $\overline{x_0 \cdot x_1 \cdot \dots \cdot x_{N-1} \cdot x_N} = \overline{\alpha \cdot x_N} = \overline{\alpha} + \overline{x_N}$ .

A questo punto "scompongo  $\alpha$ " e ottengo  $\overline{x_0 \cdot x_1 \cdot \dots \cdot x_{N-1}} + \overline{x_N}$ , e applicando il passo base ottengo la tesi.

### **SECONDO TH. DE MORGAN:**

$$\overline{x_0 + x_1 + \dots + x_{N-1}} = \overline{x_0} \cdot \overline{x_1} \cdot \dots \cdot \overline{x_{N-1}}$$

Caso particolare  $N = 2 : \overline{x + y} = \overline{x} \cdot \overline{y}$

### **DIMOSTRAZIONE:**

**Si dimostra per induzione.**

**Passo base:** tabella di verità per  $N = 2$

$x_1$	$x_0$	$x_0 + x_1$	$\overline{x_0 + x_1}$	$\overline{x_0} \cdot \overline{x_1}$
0	0	0	1	1
0	1	1	0	0
1	0	1	0	0
1	1	1	0	0

**Passo induttivo:**

- Suppongo che  $\overline{x_0 + x_1 + \dots + x_{N-1}} = \overline{x_0} \cdot \overline{x_1} \cdot \dots \cdot \overline{x_{N-1}}$  sia vero
- Dimostro che ciò implica  $\overline{x_0 + x_1 + \dots + x_{N-1} + x_N} = \overline{x_0} \cdot \overline{x_1} \cdot \dots \cdot \overline{x_{N-1}} \cdot \overline{x_N}$

Per farlo basta semplicemente porre  $\alpha = x_0 + x_1 + \dots + x_{N-1}$ , e quindi ottengo sostituendo nel passo base che  $\overline{x_0 + x_1 + \dots + x_{N-1} + x_N} = \overline{\alpha + x_N} = \overline{\alpha} \cdot \overline{x_N}$ .

A questo punto "scomatto  $\alpha$ " e ottengo  $\overline{x_0 + x_1 + \dots + x_{N-1}} \cdot \overline{x_N}$ , e applicando il passo base ottengo la tesi.

## Algebra di Boole $\iff$ Reti Combinatorie

Esiste una corrispondenza fra Algebra di Boole e Reti Combinatorie, per cui

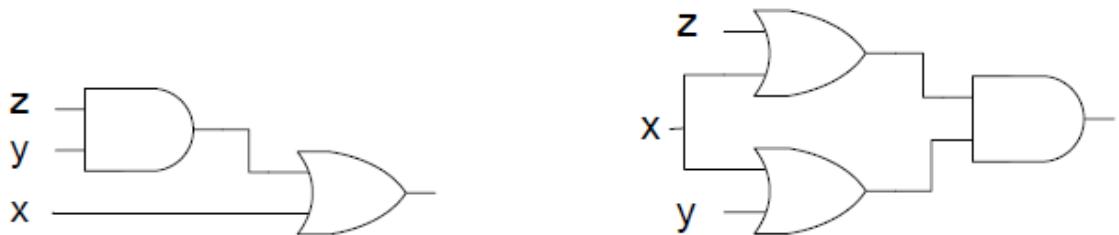


**Data una Rete Combinatoria è sempre possibile trovare un'espressione booleana che ne descrive le uscite.**  
**Data un'espressione in Algebra di Boole è sempre possibile sintetizzare una Rete Combinatoria che la calcoli.**



**Due Reti Combinatorie possono essere logicamente equivalenti ma non fisicamente!**

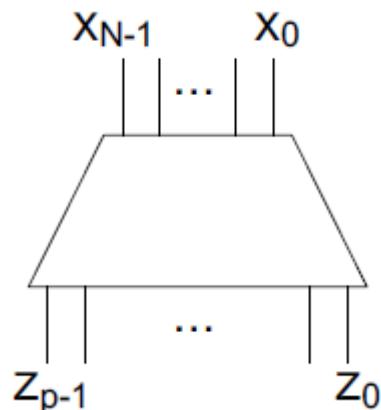
Esempio:



Le due reti sono **equivalenti logicamente**, nel senso che il valore dell'uscita è lo stesso, **ma la seconda costa più della prima** perché c'è una porta logica in più!

## Reti Combinatorie notevoli

### Decoder

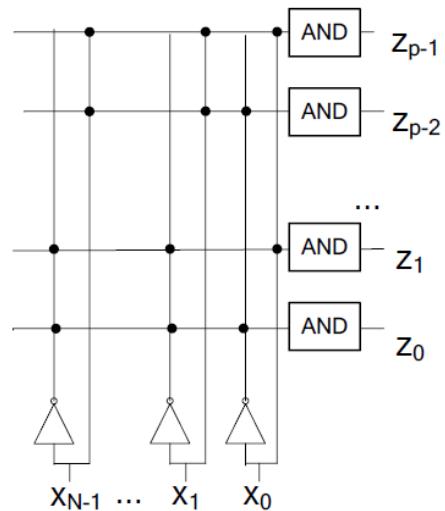


- **Ingressi:**  $N$
- **Uscite:**  $2^N$
- **Legge  $F$ :**  $z = z_j = 1 \iff (x_{N-1}x_{N-2} \dots x_1x_0)_{b2} \equiv j$   
L'uscita  $j$ -esima riconosce lo stato di ingresso i cui bit sono la codifica di  $j$  in base 2
- La sua tabella di verità è una matrice identità  $2^N \times 2^N$

- **Sintesi:** ( $p = 2^N$ )

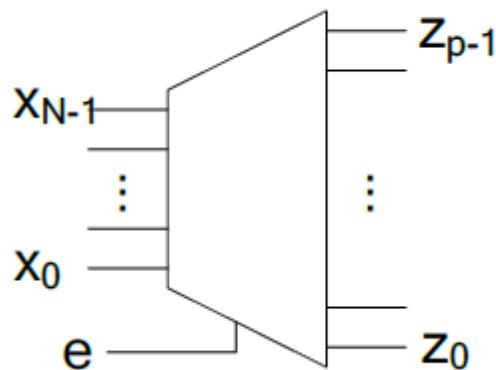
$$\begin{aligned} z_0 &= \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot \dots \cdot \overline{x_1} \cdot \overline{x_0} \\ z_1 &= \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot \dots \cdot \overline{x_1} \cdot x_0 \\ &\dots \\ z_{p-2} &= x_{N-1} \cdot x_{N-2} \cdot \dots \cdot x_1 \cdot \overline{x_0} \\ z_{p-1} &= x_{N-1} \cdot x_{N-2} \cdot \dots \cdot x_1 \cdot x_0 \end{aligned}$$

Espressione algebrica delle uscite



Circuito che implementa un decoder

## Decoder con enabler (decoder espandibile)



- **Ingressi:**  $N + 1$

- **Uscite:**  $2^N$

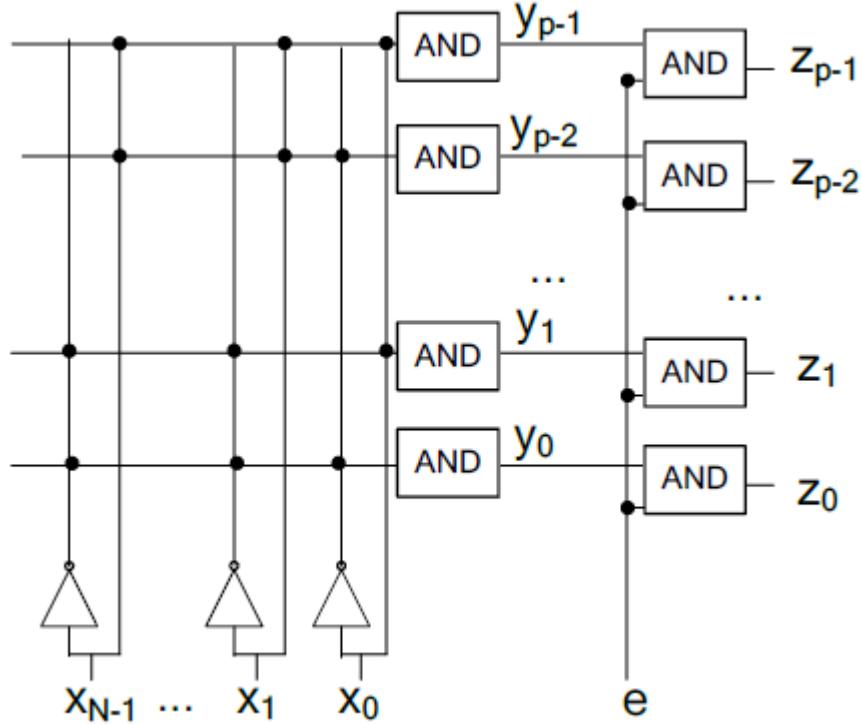
- **Legge F:**  $z_i = y_i \cdot e$

Se l'ingresso di enabler vale 1, la rete si comporta come un decoder  $N$  to  $2^N$ .

Se l'ingresso di enabler è 0 invece, tutte le uscite valgono 0.

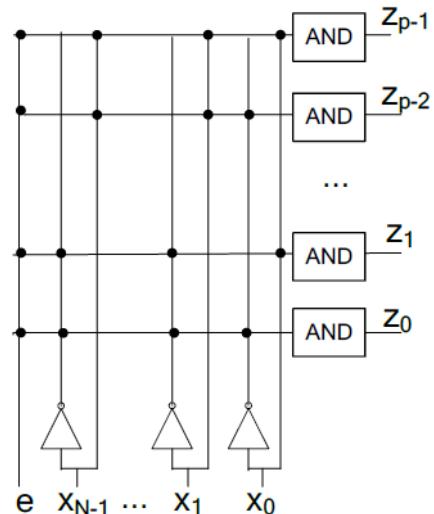
- **Sintesi:**

Una prima possibilità è la seguente



Dove  $y_i$  sarebbero le uscite del decoder se non avesse l'enabler. Ma visto che mettere due porte identiche in cascata è inutile, si giunge a questa sintesi:

$$\begin{aligned} z_0 &= e \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot \dots \cdot \overline{x_1} \cdot \overline{x_0} \\ z_1 &= e \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot \dots \cdot \overline{x_1} \cdot x_0 \\ &\dots \\ z_{p-2} &= e \cdot x_{N-1} \cdot x_{N-2} \cdot \dots \cdot x_1 \cdot \overline{x_0} \\ z_{p-1} &= e \cdot x_{N-1} \cdot x_{N-2} \cdot \dots \cdot x_1 \cdot x_0 \end{aligned}$$



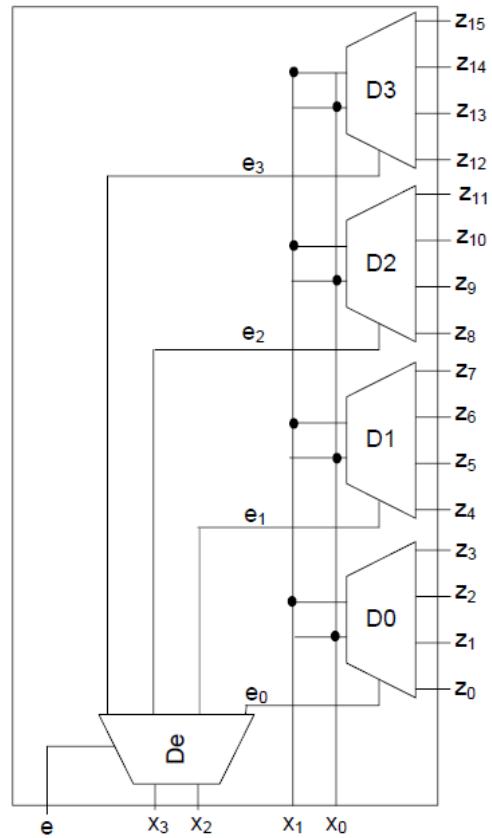
**Esempio: Costruzione di un decoder 4 to 16 a partire da decoder 2 to 4**

$x_3$	$x_2$	$x_1$	$x_0$	$z_0$	$z_1$	$z_2$	$z_3$	$z_4$	$z_5$	$z_6$	$z_7$	$z_8$	$z_9$	$z_{10}$	$z_{11}$	$z_{12}$	$z_{13}$	$z_{14}$	$z_{15}$
0	0	0	0	1	0	0	0												
0	0	0	1	0	1	0	0					0		0					
0	0	1	0	0	0	1	0												
0	0	1	1	0	0	0	1												
0	1	0	0					1	0	0	0								
0	1	0	1					0	1	0	0								
0	1	1	0					0	0	1	0								
0	1	1	1					0	0	0	1								
1	0	0	0						1	0	0	0							
1	0	0	1						0	1	0	0							
1	0	1	0						0	0	1	0							
1	0	1	1						0	0	0	1							
1	1	0	0							1	0	0	0						
1	1	0	1							0	1	0	0						
1	1	1	0							0	0	1	0						
1	1	1	1							0	0	0	1						

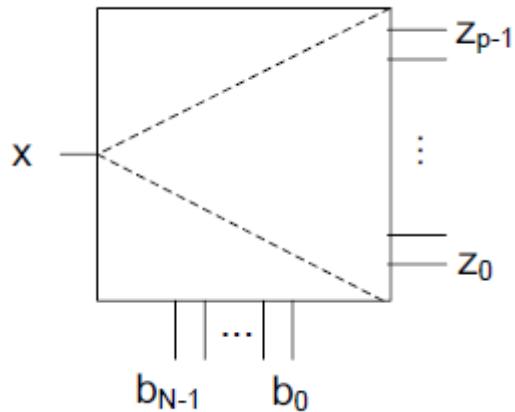
La tabella di verità è una matrice identità 16x16.

Il decoder  $D_i$  si attiva quando le variabili di comando  $(x_3x_2)_{b2} \equiv i$  e riceve in input le variabili  $(x_1x_0)$ . Quindi l'uscita  $|z_i|_4 = (x_1x_0)_{b2}$  considerando ogni singolo decoder.

L'enabler generale del decoder 4 to 16 va messo a monte di tutti, così che quando quello è 0 nessuno dei decoder  $D_i$  funziona.



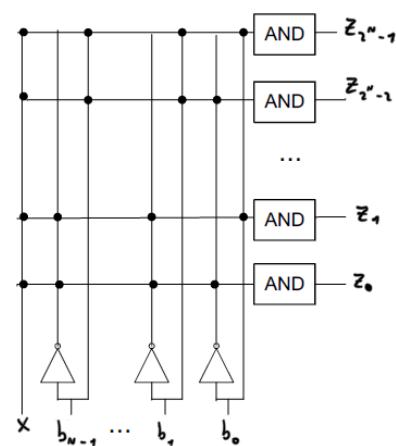
## Demultiplexer



- **Ingressi:**  $N + 1$  ( $x$  variabile da commutare e  $b_i$  variabili di comando)

- **Uscite:**  $2^N$
- **Legge F:**  $z = z_j = x \iff (b_{N-1} b_{N-2} \dots b_1 b_0)_{b2} \equiv j$   
L'uscita  $j$ -esima insegue la variabile da commutare se e solo se la codifica in base 2 delle variabili di comando è  $j$ , altrimenti vale 0
- **Sintesi:**

$$\begin{aligned} z_0 &= x \cdot \overline{b_{N-1}} \cdot \overline{b_{N-2}} \dots \overline{b_1} \cdot \overline{b_0}, \\ z_1 &= x \cdot \overline{b_{N-1}} \cdot \overline{b_{N-2}} \dots \overline{b_1} \cdot b_0, \\ &\dots \\ z_{2^N-2} &= x \cdot b_{N-1} \cdot b_{N-2} \dots b_1 \cdot \overline{b_0}, \\ z_{2^N-1} &= x \cdot b_{N-1} \cdot b_{N-2} \dots b_1 \cdot b_0 \end{aligned}$$

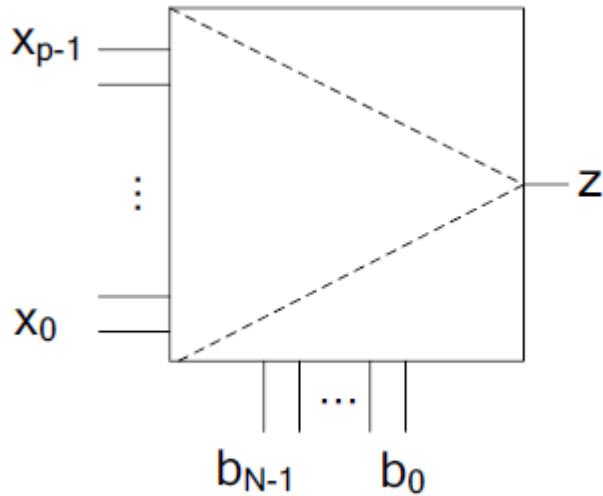


**Le variabili di comando indicano il numero della porta (uscita) che si deve attivare!**



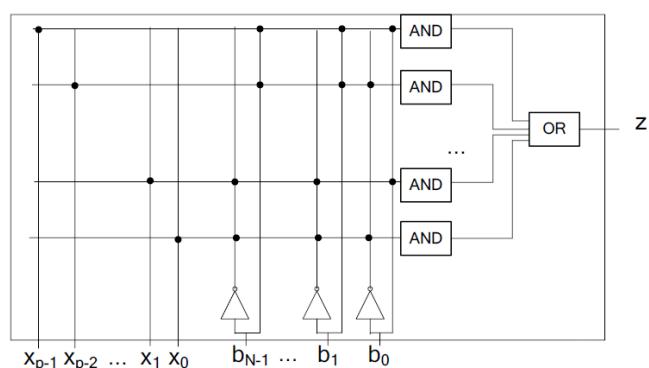
**Un Demultiplexer è UGUALE ad un decoder con enabler, solo che:**  
L'enabler del decoder *e ora si chiama  $x$*  (variabile da commutare/inseguire)  
**Gli ingressi  $x_i$  del decoder ora si chiamano  $b_i$**  (variabili di comando)

## Multiplexer



- **Ingressi:**  $N + 2^N$  ( $N$  variabili di comando  $b_i$  +  $2^N$  ingressi  $x_j$ )
- **Uscite:** 1
- **Legge F:**  $z = x_j \iff (b_{N-1}b_{N-2}\dots b_1b_0)_{b2} \equiv j$
- **Sintesi:**  
La sintesi è uguale a quella di un decoder con enabler, solo che  $e$  qui diventano i  $2^N$  ingressi  $x_j$  e che in fondo ci vuole una porta OR a  $N + 2^N$  ingressi, perché l'uscita è una ( $z$ ).

$$\begin{aligned} z = & x_0 \cdot \overline{b_{N-1}} \cdot \overline{b_{N-2}} \cdot \dots \cdot \overline{b_1} \cdot \overline{b_0} + \\ & x_1 \cdot \overline{b_{N-1}} \cdot \overline{b_{N-2}} \cdot \dots \cdot \overline{b_1} \cdot b_0 + \\ & \dots \\ & x_{p-2} \cdot b_{N-1} \cdot b_{N-2} \cdot \dots \cdot b_1 \cdot \overline{b_0} + \\ & x_{p-1} \cdot b_{N-1} \cdot b_{N-2} \cdot \dots \cdot b_1 \cdot b_0 \end{aligned}$$



## L'importanza del multiplexer

 Un Multiplexer con  $N$  variabili di comando è in grado di realizzare **QUALUNQUE** legge combinatoria di  $N$  ingressi ed un'uscita, basta attaccare ai  $2^N$  ingressi dei generatori di costante, in modo tale da rispecchiare la tabella di verità della rete desiderata.

Il Multiplexer è una rete a 2 **LIVELLI DI LOGICA (2LL)**, cioè la strada più lunga fra un ingresso e un'uscita passa attraverso 2 porte logiche.

Quindi:

 **OGNI Rete Combinatoria può essere costruita combinando porte AND, OR, NOT in MASSIMO 2 Livelli di Logica!!!**

Questo comporta che ci sia un limite superiore al ritardo di qualunque Rete Combinatoria!

## Sintesi di Reti Combinatorie in Forma SP

**FORMA SP:**  $\sum_{j=0}^{M-1} \left( \prod_{i=0}^{N-1} x_i \right)_j$

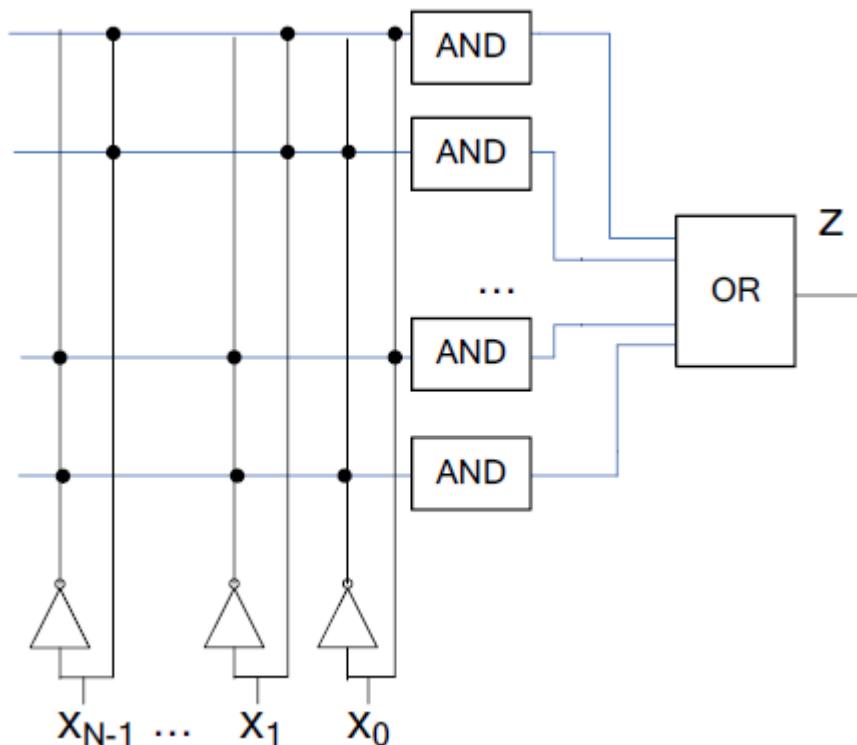
**Esempio:**  $\overline{x_2} \cdot \overline{x_1} \cdot x_0 + x_2 \cdot \overline{x_1} \cdot \overline{x_0} + x_2 \cdot \overline{x_1} \cdot x_0$

**Forma SP** → Ogni uscita è somma del prodotto di **qualche** variabile di ingresso, diretta o negata.

**Forma Canonica SP** → Ogni uscita è somma del prodotto di **TUTTE** le variabili di ingresso, dirette o negate



In una sintesi in Forma **CANONICA SP**, ogni variabile di uscita è la somma del prodotto di TUTTE le variabili di ingresso, dirette o negate.



## Sintesi in Forma SP a costo minimo

Esistono due **criteri di costo**:

- **A porte**: ogni **porta logica** conta come un'unità di costo
- **A diodi (a ingressi)**: ogni **ingresso** conta come un'unità di costo

**"LEMMA" DI SHANNON:** E' sempre possibile scrivere qualunque legge  $F$  di una Rete Combinatoria come somma di prodotti degli ingressi, diretti o negati.

Questo perché la forma del MUX è SP, e ogni legge  $F$  può essere realizzata tramite un MUX.

**ESPANSIONE DI SHANNON:** Data una legge  $z = f(x_{N-1}, x_{N-2} \dots, x_1, x_0)$ , l'Espansione di Shannon è la seguente:

$$\begin{aligned} z &= f(0, 0, \dots, 0, 0) \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdots \cdot \overline{x_1} \cdot \overline{x_0} + \\ &\quad f(0, 0, \dots, 0, 1) \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdots \cdot \overline{x_1} \cdot x_0 + \\ &\quad \cdots \\ &\quad f(1, 1, \dots, 1, 0) \cdot x_{N-1} \cdot x_{N-2} \cdots \cdot x_1 \cdot \overline{x_0} + \\ &\quad f(1, 1, \dots, 1, 1) \cdot x_{N-1} \cdot x_{N-2} \cdots \cdot x_1 \cdot x_0 \end{aligned}$$

Ciascuno dei termini si chiama **MINTERMINE**, e corrisponde ad uno **stato di ingresso riconosciuto dalla rete**

**Esempio:**

X <sub>3</sub>	X <sub>2</sub>	X <sub>1</sub>	X <sub>0</sub>	Z <sub>0</sub>	Espansione di Shannon	Forma canonica SP (lista dei mintermini)
0	0	0	0	0		
0	0	0	1	1		
0	0	1	0	1		
0	0	1	1	1	$z = 0 \cdot \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1} \cdot \overline{x_0}$	$\overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1} \cdot x_0$
0	1	0	0	0	$+ 1 \cdot \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1} \cdot x_0$	$+\overline{x_3} \cdot \overline{x_2} \cdot x_1 \cdot \overline{x_0}$
0	1	0	1	0	$\cdots$	$+\overline{x_3} \cdot \overline{x_2} \cdot x_1 \cdot x_0$
0	1	1	0	1	$+ 0 \cdot x_3 \cdot x_2 \cdot x_1 \cdot \overline{x_0}$	$+\overline{x_3} \cdot x_2 \cdot x_1 \cdot \overline{x_0}$
1	0	0	0	1	$+ 0 \cdot x_3 \cdot x_2 \cdot x_1 \cdot x_0$	$+\overline{x_3} \cdot x_2 \cdot x_1 \cdot x_0$
1	0	0	1	1		$+\overline{x_3} \cdot x_2 \cdot x_1 \cdot x_0$
1	0	1	0	1		$+\overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1} \cdot \overline{x_0}$
1	0	1	1	0		$+\overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1} \cdot x_0$
1	1	0	0	0		$+\overline{x_3} \cdot \overline{x_2} \cdot x_1 \cdot \overline{x_0}$
1	1	0	1	0		$+\overline{x_3} \cdot \overline{x_2} \cdot x_1 \cdot x_0$
1	1	1	0	0		$+\overline{x_3} \cdot x_2 \cdot \overline{x_1} \cdot \overline{x_0}$
1	1	1	1	0		$+\overline{x_3} \cdot x_2 \cdot \overline{x_1} \cdot x_0$

Applicando esaustivamente le due seguenti regole:  $\left\{ \begin{array}{l} \alpha x + \alpha \overline{x} = \alpha x + \alpha \overline{x} + \alpha \\ \alpha + \alpha = \alpha \end{array} \right.$

(la prima consente di fondere i mintermini, e la seconda ci ricorda di non inserire duplicati)

si ottiene una lista, in Forma SP, chiamata **LISTA DEGLI IMPLICANTI**.

**IMPLICANTE**: Prodotto di **alcune** variabili di ingresso, dirette o negate

**Un Mintermine è un caso particolare di implicante**

Ora, applicando la regola  $\alpha x + \alpha = \alpha$  si ottiene la **LISTA DEGLI IMPLICANTI PRINCIPALI**.

**IMPLICANTE PRINCIPALE**: Implicante che **non è in grado di fondere** con nessun altro implicante

**Recap: Dall'espansione di Shannon alla Lista di Copertura**

1. Dalla tabella di verità passo alla corrispettiva **Espansione di Shannon**
2. Applico le due regole  $\begin{cases} \alpha x + \alpha \bar{x} = \alpha x + \alpha \bar{x} + \alpha \\ \alpha + \alpha = \alpha \end{cases}$  e trovo la **Lista degli Implicanti**
3. Applico la regola  $\alpha x + \alpha = \alpha$  e trovo la **Lista degli Implicanti Principali**

**LISTA DI COPERTURA**: **Lista di Implicanti**, la cui somma è una Forma SP per la funzione  $f$ .

Alcuni **esempi**: la lista dei mintermini, la lista degli implicanti (e implicanti principali)

**LISTA DI COPERTURA NON RIDONDANTE**: Lista di copertura tale che, **se tolgo un elemento dalla lista, smette di essere una lista di copertura**

## Mappe di Karnaugh

Tecnica alternativa alle tabelle di verità per descrivere le Reti Combinatorie.

**MAPPA DI KARNAUGH:** Posta una rete con  $N$  ingressi, è una matrice di  $2^N$  celle.

Ogni cella è individuata da uno stato di ingresso (le coordinate della cella).

Ogni cella inoltre contiene il valore dell'uscita corrispondente allo stato di ingresso.



Le coordinate nelle Mappe di Karnaugh devono essere scritte in modo tale che stati di ingresso consecutivi (colonne e righe) siano adiacenti!

Le Mappe di Karnaugh vanno immaginate come se fossero disegnate su superfici sferiche, "effetto pacman". Inoltre **non sono usabili per reti che hanno più di 4 ingressi!**

Definizioni:

- **SOTTOCUBO DI ORDINE 1:** Casella contenente un '1'
  - **SOTTOCUBO DI ORDINE 2:** Sottocubo costituito da **sottocubi adiacenti di ordine 1**
- **ADIACENZA TRA SOTTOCUBI DI ORDINE 1:** Due sottocubi di ordine 1 si dicono adiacenti se **differiscono fra loro per una sola coordinata** (avendo le altre  $N - 1$  identiche)
  - **ADIACENZA TRA SOTTOCUBI DI ORDINE 2:** Due sottocubi di ordine 2 si dicono adiacenti se **differiscono fra loro per una sola coordinata** (avendo le altre  $N - 2$  identiche)
- **SOTTOCUBO PRINCIPALE:** Sottocubo per cui **non esiste nessun sottocubo più grande che lo copre completamente**
  - **SOTTOCUBO PRINCIPALE ESSENZIALE:** Sottocubo principale che è l'unico a coprire un dato '1' nella mappa
  - **SOTTOCUBO PRINCIPALE ASSOLUTAMENTE ELIMINABILE:** Sottocubo principale che **riconosce '1' già riconosciuti da Sottocubi Principali Essenziali**

- **SOTTOCUBO PRINCIPALE SEMPLICEMENTE ELIMINABILE:** Sottocubo principale che “**concorre con altri sottocubi**” per essere Essenziale
- **LISTA DI COPERTURA:** Insieme qualunque di sottocubi che coprono **TUTTI** gli ‘1’ della mappa
- **LISTA DI COPERTURA NON RIDONDANTE:** Lista di copertura tale che **se tolgo un sottocubo non è più una lista di copertura**

**Un ‘1’ corrisponde ad uno stato di ingresso riconosciuto dalla rete!**

Un sottocubo **COPRE** altri sottocubi quando li **contiene interamente**



**Esiste una corrispondenza biunivoca fra Implicanti Principali della legge  $F$  e Sottocubi Principali nella Mappa di Karnaugh:**

- Un sottocubo di ordine 1 è un mintermine
- Un sottocubo di ordine  $> 1$  è un implicante

### Algoritmo per la ricerca dei sottocubi principali / lista di copertura

1. Parto dai sottocubi più grandi che trovo e li segno
2. L’insieme corrente di **TUTTI** i sottocubi di ordine  $K$  copre tutti gli uni?  
(Cioè: formano una lista di copertura?)
  - a. Si: **FINE**
  - b. No:  $K \rightarrow K/2$  e torna al passo 2.

**Errore Tipico:** Fermarsi al passo 2 non appena si sono trovati alcuni sottocubi che coprono tutti gli uni. **Invece bisogna considerare TUTTI i sottocubi di ordine  $K$ , ce ne saranno alcuni superflui, ma va bene così, serve poi per costruire l’albero delle possibili liste di copertura!**

## Algoritmo per la lista di copertura di costo minimo

- Individuare i **Sottocubi Principali Essenziali**
- Si eliminano i **Sottocubi Principali Assolutamente Eliminabili**
- Generare l'**albero binario delle liste di copertura** andando avanti per **ipotesi alternative**, includendo o eliminando un determinato Sottocubo Principale Semplicemente Eliminabile per volta:
  - $SPSE \rightarrow SPE$ : lo includo nella lista. Qualche altro sottocubo diventa  $SPAЕ$
  - $SPSE \rightarrow SPAЕ$ : lo escludo dalla lista. Qualche altro sottocubo diventa  $SPE$
- Una volta finiti i Sottocubi Principali Semplicemente Eliminabili su cui fare ipotesi, **valutare il costo delle Liste di Copertura trovate (foglie dell'albero)** e scegliere la meno costosa

## Sintesi in Forma PS

**FORMA PS:** 
$$\prod_{j=0}^{M-1} \left( \sum_{i=0}^{N-1} x_i \right)_j$$

**Esempio:**  $(x_2 + \bar{x}_1 + x_0) \cdot (\bar{x}_2 + \bar{x}_1 + \bar{x}_0) \cdot (\bar{x}_2 + x_1 + x_0)$

La teoria della Forma PS è la duale della SP, con **implicati** al posto degli **implicanti**

## Sintesi in Forma PS a costo minimo

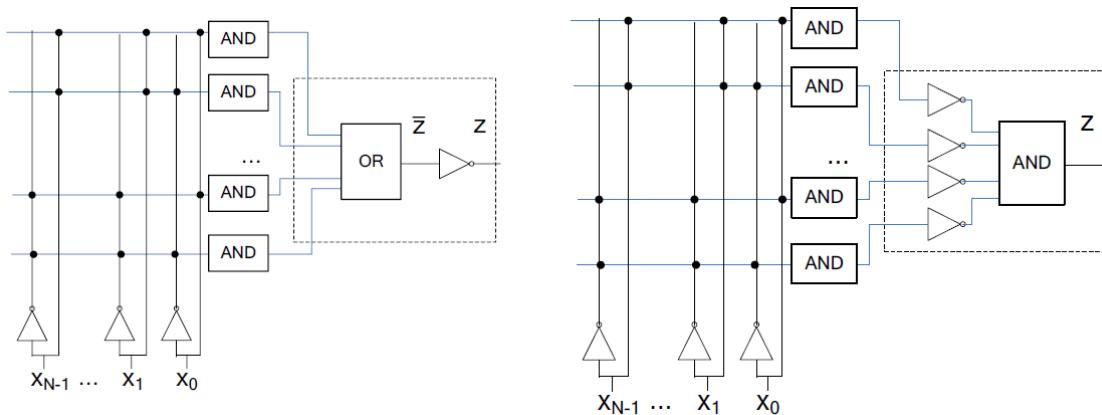
Al posto di rifare tutta la teoria, si utilizza questo algoritmo:

- Data una legge  $F$ , ricavo  $\bar{F}$
- **Faccio la sintesi in Forma SP di  $\bar{F}$**
- **Inserisco un inverter in cascata** alla Sintesi in Forma SP appena calcolata, così ottengo una sintesi in Forma SP di  $F$

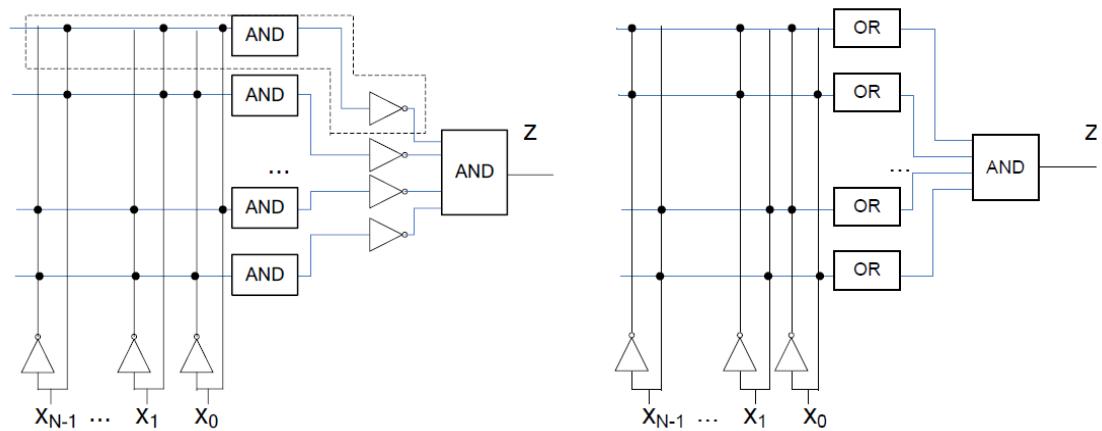
- Applico De Morgan 2 volte, dall'esterno all'interno:

$$\circ \sum_{j=0}^{M-1} \left( \prod_{i=0}^{N-1} x_i \right)_j \Rightarrow \prod_{j=0}^{M-1} \left( \prod_{i=0}^{N-1} x_i \right)_j \Rightarrow \prod_{j=0}^{M-1} \left( \sum_{i=0}^{N-1} \bar{x}_i \right)_j$$

Primo uso di De Morgan:



Secondo uso di De Morgan:



**Note:**

- Se  $\bar{F}$  è in Forma Canonica SP, allora  $F$  è in Forma Canonica PS
- **Se la sintesi in Forma SP di  $\bar{F}$  è di costo minimo, allora lo è anche la sintesi in Forma PS di  $F$**

# Porte Logiche Universali

Le porte NAND e NOR sono dette **PORTE LOGICHE UNIVERSALI**, perché ogni legge combinatoria può essere sintetizzata usando solo porte NAND o solo porte NOR:

	Porta	realizz. a NAND	realizz. a NOR
NOT $x = x \cdot x \Rightarrow \bar{x} = \overline{x \cdot x}$ $x = x + x \Rightarrow \bar{x} = \overline{x + x}$			
AND $x \cdot y = \overline{\overline{x \cdot y}}$ $x \cdot y = \overline{\overline{x}} \cdot \overline{\overline{y}}$			
OR $x + y = \overline{\overline{x + y}}$ $x + y = \overline{\overline{x}} + \overline{\overline{y}}$			

Visto che ogni legge combinatoria può essere realizzata con sole porte AND, OR e NOT, e queste si possono realizzare solo con porte NAND o NOR, ho dimostrato che le porte NAND e NOR sono Porte Logiche Universali.

Come ricordarsi da dove partire per il disegno:

- Porta NOT: scriverla come **Idempotenza e negare**
- Porte AND, OR: **negare due volte (e usare De Morgan)**

## Sintesi a Porte NAND



**PARTIRE DALLA SINTESI IN FORMA SPI!!!**

Dal punto di vista algebrico:

- Parto dalla sintesi in **Forma SP**:  $\sum_{j=0}^{M-1} \left( \prod_{i=0}^{N-1} x_i \right)_j$

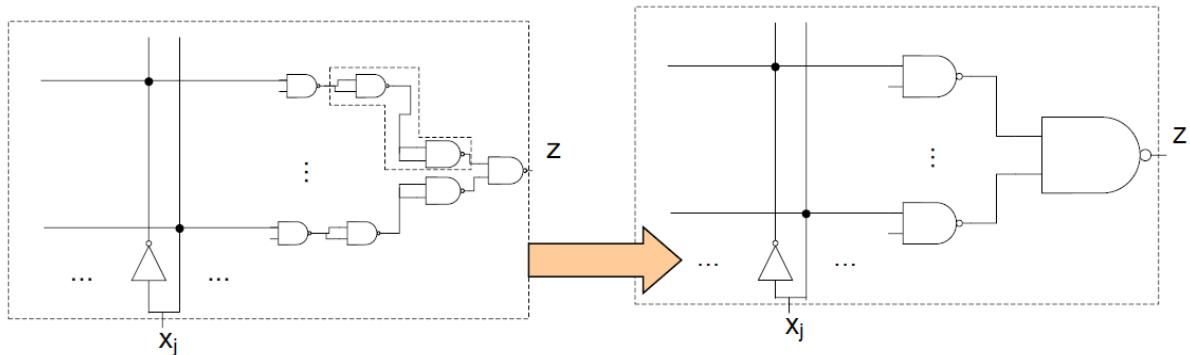
- Complemento due volte:**  $\overline{\sum_{j=0}^{M-1} \left( \prod_{i=0}^{N-1} x_i \right)_j}$

- Applico De Morgan** sul complemento interno:

$$\overline{\sum_{j=0}^{M-1} \left( \prod_{i=0}^{N-1} x_i \right)_j} \Rightarrow \overline{\prod_{j=0}^{M-1} \left( \overline{\prod_{i=0}^{N-1} x_i} \right)_j}$$

Dal punto di vista della sintesi:

- Data una sintesi in **Forma SP**
- Sostituisco le porte AND e OR con i loro equivalenti a porte NAND**
- Elimino le coppie di NAND in cascata createsi**



**Nota:** Gli inverter sugli ingressi puoi trascurarli

## Sintesi a Porte NOR



**PARTIRE DALLA SINTESI IN FORMA PS!!!**

Dal punto di vista algebrico:

- Parto dalla sintesi in **Forma PS**:  $\prod_{j=0}^{M-1} \left( \sum_{i=0}^{N-1} x_i \right)_j$

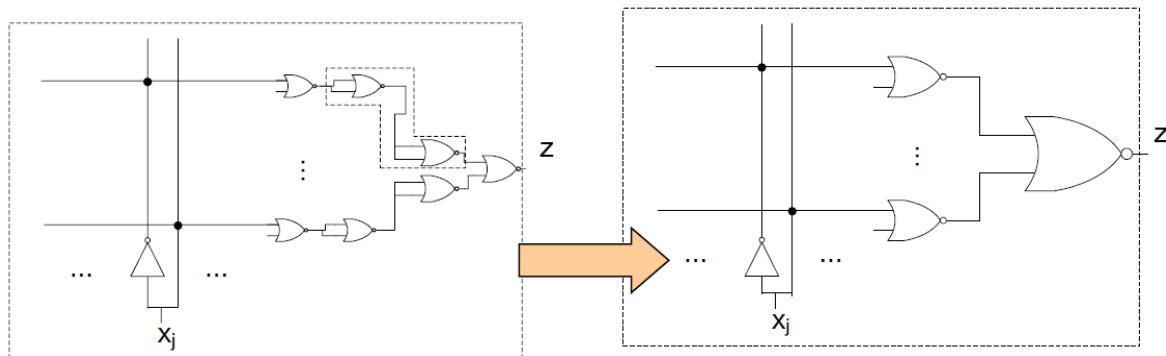
- Complemento due volte:**  $\overline{\overline{\prod_{j=0}^{M-1} \left( \sum_{i=0}^{N-1} x_i \right)_j}}$

- Applico De Morgan** sul complemento interno:

$$\overline{\prod_{j=0}^{M-1} \left( \sum_{i=0}^{N-1} x_i \right)_j} \Rightarrow \sum_{j=0}^{M-1} \overline{\left( \sum_{i=0}^{N-1} x_i \right)_j}$$

Dal punto di vista della sintesi:

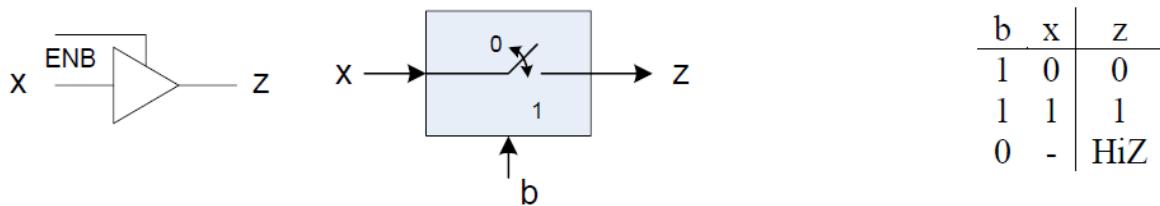
- Data una sintesi in **Forma PS**
- Sostituisco le porte AND e OR con i loro equivalenti a porte NOR**
- Elimino le coppie di NOR in cascata createsi**



**Nota:** Gli inverter sugli ingressi puoi trascurarli

## Porte tri-state

**PORTA TRI-STATE**: Rete logica capace di **disconnettere fisicamente un'uscita di una rete da una linea condivisa** (bus)



**Quando  $b$  vale 1 la tri-state si comporta come un buffer. Quando  $b$  vale 0, stacca fisicamente l'ingresso dall'uscita, che si dice si trova in **ALTA IMPEDENZA** ( $HiZ$ ).**



**Se su un bus ci sono  $N$  uscite collegate assieme, su OGNI uscita deve essere presente una tri-state, così che almeno  $N - 1$  porte in ogni istante siano in Alta Impedenza.**



**ATTENZIONE! Alta Impedenza NON E' UN VALORE LOGICO!**

Come non lo è "Non Specificato (-)"!

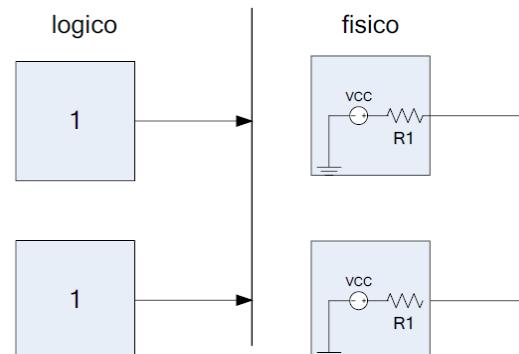
I valori logici sono solo 0 e 1!

## Perché servono le tri-state

Possono verificarsi 3 casi:

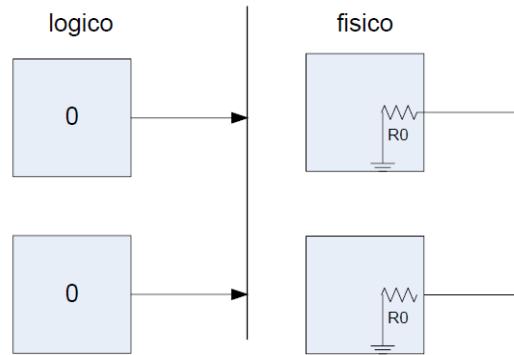
- **Entrambe le uscite valgono 1:**

Sulla linea condivisa ci finisce il valore logico 1 e tutto va bene, inoltre nel circuito non scorre corrente perché non c'è differenza di potenziale (sono entrambe a  $VCC$ )



- **Entrambe le uscite valgono 0:**

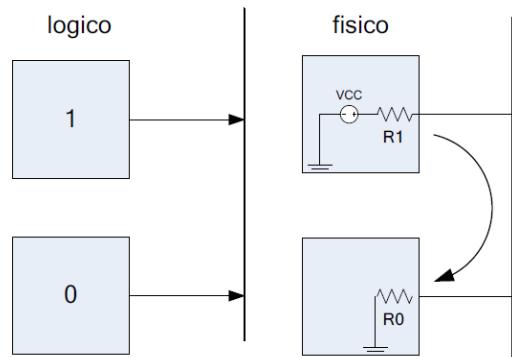
Sulla linea condivisa ci finisce il valore logico 0 e tutto va bene, inoltre nel circuito non scorre corrente perché non c'è differenza di potenziale (sono entrambe a *ground*)



- **Le uscite sono discordi:**

Qui si verificano due problemi:

1. **Elettrico:** c'è una maglia chiusa dove scorre corrente, perché la differenza di potenziale stavolta non è 0 ma è  $vcc$ . Visto che di solito in questi circuiti le resistenze  $R_i$  sono piccole, scorre tanta corrente e **il circuito si brucia!**
2. **Logico:** La tensione che c'è sulla linea condivisa (e quindi **il valore logico**) dipende dal **valore delle resistenze  $R_i$** , quindi non lo posso controllare!

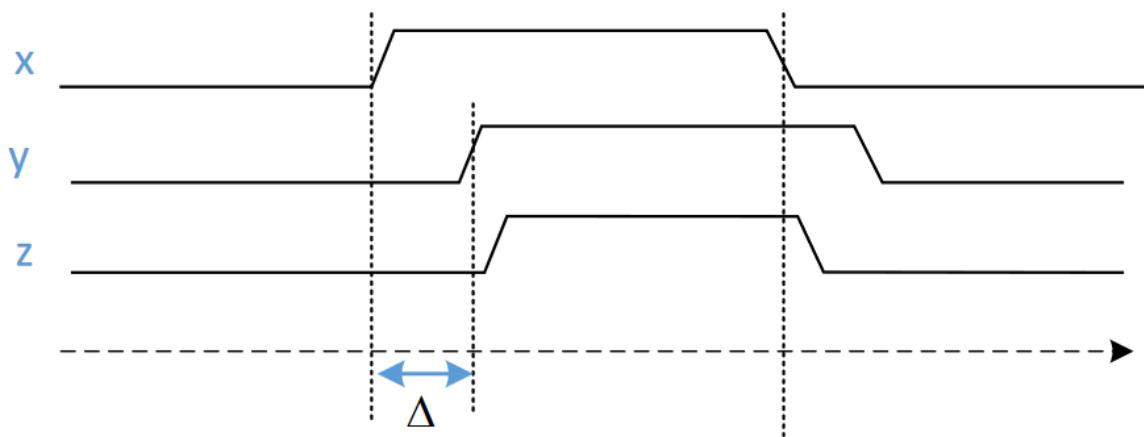
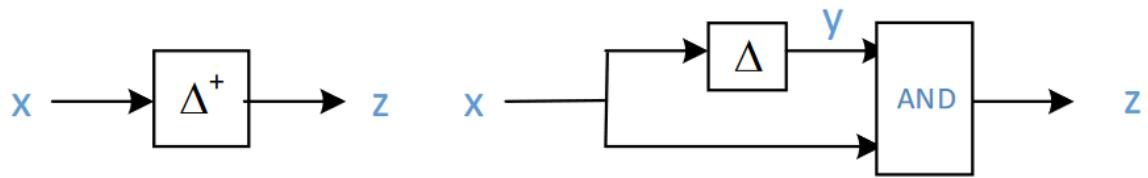


## Circuiti di ritardo e formatori di impulsi

Il buffer genera un ritardo "simmetrico", identico sulla trasizione 0-1 o 1-0.

Fa comodo invece avere circuiti che generano un ritardo asimmetrico controllabile:

### Circuito di ritardo sul fronte di salita

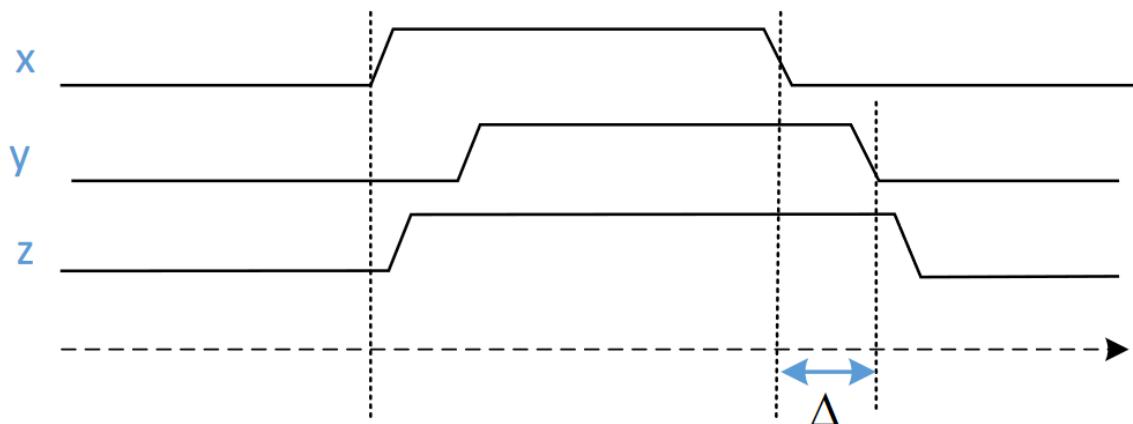
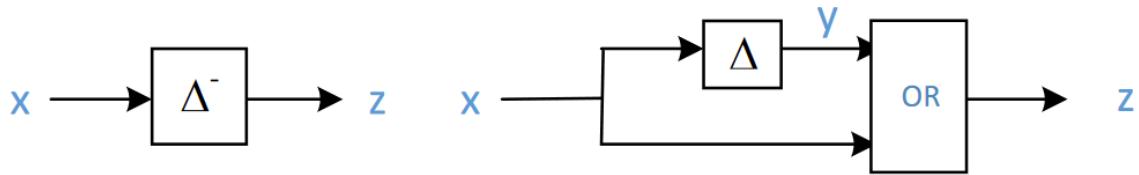


Visto che c'è una porta AND, sul fronte di salita,  $y$  (ingresso  $x$  ritardato con un buffer  $\Delta$ ) rimane a 0 per un ritardo pari a  $\Delta$ , quindi il ritardo sul fronte di salita è  $\Delta$

Sul fronte di discesa invece, non appena  $x$  va a 0 l'uscita viene forzata a 0.

**Devo tenere il più possibile l'uscita a 0  $\Rightarrow$  Porta AND**

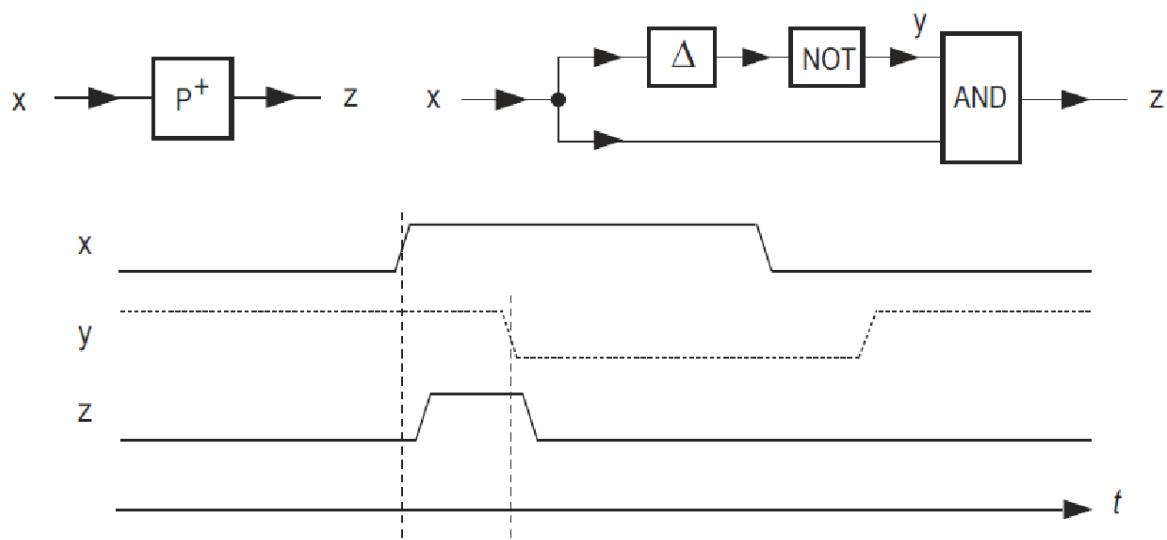
## Circuito di ritardo sul fronte di discesa



Visto che c'è una porta OR, sul fronte di salita non appena  $x$  passa ad 1 la rete transisce da 0 ad 1. Sul fronte di discesa, invece, visto che  $y$  è ritardato rispetto ad  $x$  esattamente di  $\Delta$ , la rete tornerà a 0 quando anche  $y$  (che è il più lento) tornerà a 0.

**Devo tenere il più possibile l'uscita a 1  $\Rightarrow$  Porta OR**

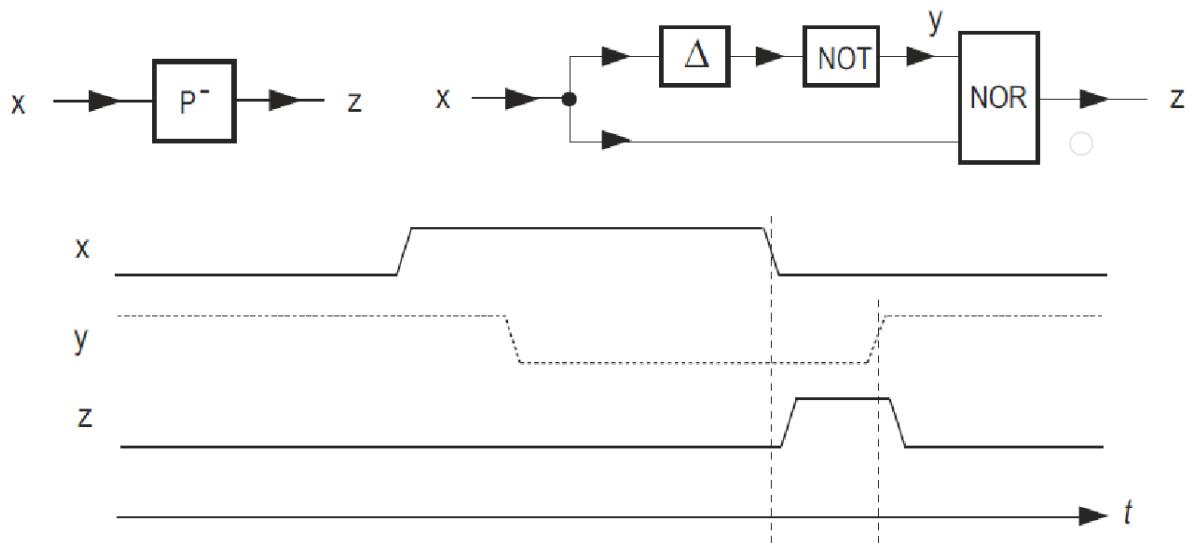
## Formatore di impulsi sul fronte di salita



Si prende il circuito del **Formatore di Ritardo sul fronte di salita** e si nega l'ingresso ritardato  $y$ , cioè gli si mette una NOT in cascata.

Si sfrutta il tempo  $\Delta$  in cui  $x$  passa ad 1 (istante controllabile in cui far partire l'impulso) ed  $y$  è ancora ad 1. Quando  $y$  passa a 0, dopo un tempo  $\Delta$  dal momento in cui  $x$  è passato ad 1, l'impulso finisce.

## Formatore di impulsi sul fronte di discesa



Si prende il circuito del **Formatore di Ritardo sul fronte di discesa** e si nega l'ingresso ritardato  $y$ , cioè gli si mette una NOT in cascata.

Inoltre **si deve mettere una porta NOR al posto dell'OR!**

Qui si sfrutta il tempo  $\Delta$  in cui  $x$  passa a 0 fino a quando, appunto  $\Delta$  dopo,  $y$  non torna ad 1. Serve il NOR perché sostanzialmente c'è sempre un'ingresso ad 1  $\forall t$ , tranne quando c'è il fronte di discesa.

Notare come lo schema  $(x, y)(t)$  sia  $(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)$  in entrambi i circuiti di ritardo, perché dipendenti solo da  $x$  e  $y$  e non dalla funzione del circuito!

Analogamente notare come lo schema  $(x, y)(t)$  in entrambi i circuiti formatori di impulsi sia  $(0, 1), (1, 1), (1, 0), (0, 0), (0, 1)$ !

# Aritmetica dei Naturali

## Rappresentazione dei numeri Naturali

Un **SISTEMA NUMERICO DI RAPPRESENTAZIONE** è costituito da:

- Un **numero  $\beta \geq 2$**  detto **BASE DI RAPPRESENTAZIONE**
- Un insieme di  $\beta$  **simboli**, detti **CIFRE**, a ciascuno dei quali è associato un numero naturale compreso fra 0 e  $\beta - 1$
- Una **LEGGE DI RAPPRESENTAZIONE** che fa corrispondere sequenze di cifre a numeri naturali

Quindi, **dato un numero  $A \in \mathbb{N}$ , lo posso rappresentare in due modi equivalenti:**

$$A = \sum_{i=0}^{n-1} a_i \cdot \beta^i = (a_{n-1}a_{n-2}\dots a_1a_0)_{\beta}$$

## Teorema della Divisione con Resto

Dati:

- $x \in \mathbb{Z}$  **DIVIDENDO**
- $\beta \in \mathbb{N}$ ,  $\beta > 0$  **DIVISORE**

$$\exists! (q, r), \text{ con } q \in \mathbb{Z}, r \in \mathbb{N}, \text{ e } 0 \leq r < \beta : x = q \cdot \beta + r$$

$q$  **QUOZIENTE**,  $r$  **RESTO** della divisione  $x/\beta$

$$q = \left\lfloor \frac{x}{\beta} \right\rfloor, \quad r = |x|_{\beta}$$

Quindi

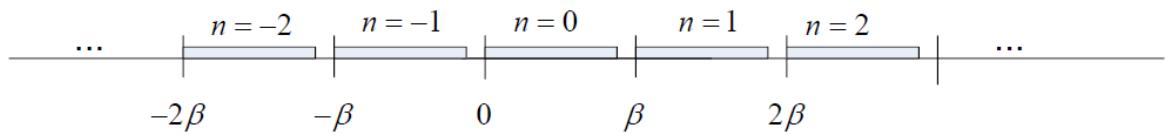
$$x = \left\lfloor \frac{x}{\beta} \right\rfloor \cdot \beta + |x|_{\beta}$$

### DIMOSTRAZIONE:

Si "spezza" quel  $\exists!$  in due, e si dimostra prima l' $\exists$ , poi l'unicità (!):

Dim. Esistenza della coppia  $(q, r)$ :

Si divide l'asse dei numeri interi in blocchi  $[n \cdot \beta, (n+1) \cdot \beta[, n \in \mathbb{Z}$ :



Per come l'abbiamo diviso, se ri-unisco tutti gli intervalli ottengo  $\mathbb{Z}$ :

$$\bigcup_{n \in \mathbb{Z}} [n \cdot \beta, (n+1) \cdot \beta[ \equiv \mathbb{Z}$$

Quindi,  $x$  fa per forza parte di uno di questi intervalli, sia esso il  $q$ -esimo.

Allora,  $q \cdot \beta \leq x < (q+1) \cdot \beta$ .

Se ora definisco  $r = x - (q \cdot \beta)$ , e sottraggo  $q \cdot \beta$  dalla relazione qui sopra, ho che:

$$\begin{aligned} q \cdot \beta - q \cdot \beta &\leq x - (q \cdot \beta) < (q+1) \cdot \beta - (q \cdot \beta) \\ \text{Quindi} \\ 0 &\leq r < \beta \end{aligned}$$

Ho dimostrato che la coppia  $(q, r)$  esiste sempre!

### Dim. Unicità della coppia $(q, r)$ :

**Questo si dimostra per assurdo!**

Supponiamo per assurdo che esistano DUE coppie  $(q_1, r_1), (q_2, r_2)$  DIVERSE e tali che  $x = q_1 \cdot \beta + r_1 = q_2 \cdot \beta + r_2$ , con:

- $q_i \in \mathbb{Z}$  (1)
- $0 \leq r_i < \beta$  (2)

Allora  $q_1 \cdot \beta + r_1 = q_2 \cdot \beta + r_2 \implies (q_1 - q_2) \cdot \beta = r_2 - r_1$ . (\*)

**Ora stimo la quantità  $r_2 - r_1$ :**

Dalla (2) si ha che  $0 - \beta < r_2 - r_1 < \beta - 0$ , quindi  $-\beta < r_2 - r_1 < \beta$

Ora, sostituendo questa stima in (\*) si ottiene che pure  $-\beta < (q_1 - q_2) \cdot \beta < \beta$ , e dividendo tutto per  $\beta$  si ha  $-1 < q_1 - q_2 < 1$ .

Ma dalla (1) segue subito che  $q_1 = q_2$  per forza.

Ma allora, se  $q_1 = q_2$ , dalla (\*) segue che pure  $r_1 = r_2$ , violando l'ipotesi fatta per assurdo in cui si diceva che le coppie  $(q_i, r_i)$  erano diverse.

Ho dimostrato che esiste un'**unica** coppia  $(q, r)$ !

## Proprietà dell'operatore modulo

Dato  $\alpha \in \mathbb{N}$ ,  $\alpha > 0$ ,

- $|x + k \cdot \alpha|_\alpha = |x|_\alpha$ ,  $k \in \mathbb{Z}$

### Dimostrazione:

Per il **Th. Div. con Resto**,  $x = \left\lfloor \frac{x}{\alpha} \right\rfloor \cdot \alpha + |x|_\alpha$  quindi, sommando ad entrambi i membri il termine  $k \cdot \alpha$ , si ottiene  $x + k \cdot \alpha = \left( \left\lfloor \frac{x}{\alpha} \right\rfloor + k \right) \cdot \alpha + |x|_\alpha$ .

Ora, visto che il termine fra parentesi tonda  $\in \mathbb{Z}$ , e  $|x|_\alpha \in [0, \alpha - 1]$ , per il **Th. Div. con Resto**, questi due termini sono quoziente e resto della divisione  $\frac{x + k \cdot \alpha}{\alpha}$ .

Quindi, visto che  $|a|_b$  è il resto della divisione  $a/b$ , ho dimostrato la proprietà!

**Spiegazione:** Visto che  $|\cdot|_\alpha \in [0, \alpha - 1]$ , posso sottrarre o aggiungere multipli a piacere, in modo da ricadere nell'intervallo  $[0, \alpha - 1]$

- $|x + y|_\alpha = ||x|_\alpha + |y|_\alpha|_\alpha$

### Dimostrazione:

Scrivo  $x$  ed  $y$  tramite il **Th. Div. con Resto**:

$$\begin{aligned} |x + y|_\alpha &= \left| \left\lfloor \frac{x}{\alpha} \right\rfloor \cdot \alpha + |x|_\alpha + \left\lfloor \frac{y}{\alpha} \right\rfloor \cdot \alpha + |y|_\alpha \right|_\alpha = \\ &= \left| |x|_\alpha + |y|_\alpha + \left( \left\lfloor \frac{x}{\alpha} \right\rfloor + \left\lfloor \frac{y}{\alpha} \right\rfloor \right) \cdot \alpha \right|_\alpha \end{aligned}$$

Ora, applicando la prima proprietà si ottiene la proprietà

**Spiegazione:** Visto che  $|\cdot|_\alpha \in [0, \alpha - 1]$ , mentre  $|x|_\alpha + |y|_\alpha \in [0, 2\alpha - 2]$ , può tranquillamente sorpassare  $\alpha - 1$ , ma visto che questo non è possibile perché c'è un'uguaglianza fra i due membri, pure i due intervalli di valori devono coincidere, quindi c'è bisogno di mettere il modulo finale per riportarlo in  $[0, \alpha - 1]$

- $|x \cdot y|_\alpha = ||x|_\alpha \cdot |y|_\alpha|_\alpha$

### Dimostrazione:

Scrivo  $x$  ed  $y$  tramite il **Th. Div. con Resto**:

$$|x \cdot y|_\alpha = \left| \left( \left\lfloor \frac{x}{\alpha} \right\rfloor \cdot \alpha + |x|_\alpha \right) \cdot \left( \left\lfloor \frac{y}{\alpha} \right\rfloor \cdot \alpha + |y|_\alpha \right) \right|_\alpha =$$

$$\begin{aligned}
&= \left| |x|_\alpha \cdot |y|_\alpha + \left( |x|_\alpha \cdot \left\lfloor \frac{y}{\alpha} \right\rfloor \cdot \alpha \right) + \left( |y|_\alpha \cdot \left\lfloor \frac{x}{\alpha} \right\rfloor \cdot \alpha \right) + \left\lfloor \frac{x}{\alpha} \right\rfloor \cdot \left\lfloor \frac{y}{\alpha} \right\rfloor \cdot \alpha^2 \right|_\alpha = \\
&= ||x|_\alpha \cdot |y|_\alpha + K \cdot \alpha|_\alpha
\end{aligned}$$

E pure qui applicando la prima proprietà si ottiene questa terza proprietà!

**Spiegazione:** Stessa spiegazione della 2 proprietà, la filosofia è la stessa, l'unico dettaglio che cambia è che prima si aveva  $|x|_\alpha + |y|_\alpha \in [0, 2\alpha - 2]$ , ora invece si ha  $|x|_\alpha \cdot |y|_\alpha \in [0, (\alpha - 1)^2]$

## Massimo numero Naturale rappresentabile

Su  $n$  cifre in base  $\beta$  posso rappresentare in totale  $\beta^n$  numeri.

Per i Naturali, visto che devo includere anche lo 0, il massimo numero rappresentabile su  $n$  cifre in base  $\beta$  è  $\beta^n - 1$ .



Il massimo numero in base  $\beta$  è quello che ottengo quando tutte le cifre hanno valore massimo, cioè ciascuna vale  $\beta - 1$

**Dimostrazione:**

$$A = \sum_{i=0}^{n-1} (\beta - 1) \cdot \beta^i = \sum_{i=0}^{n-1} \beta^{i+1} - \sum_{i=0}^{n-1} \beta^i = \sum_{i=1}^n \beta^i - \sum_{i=0}^{n-1} \beta^i = \beta^n - 1$$

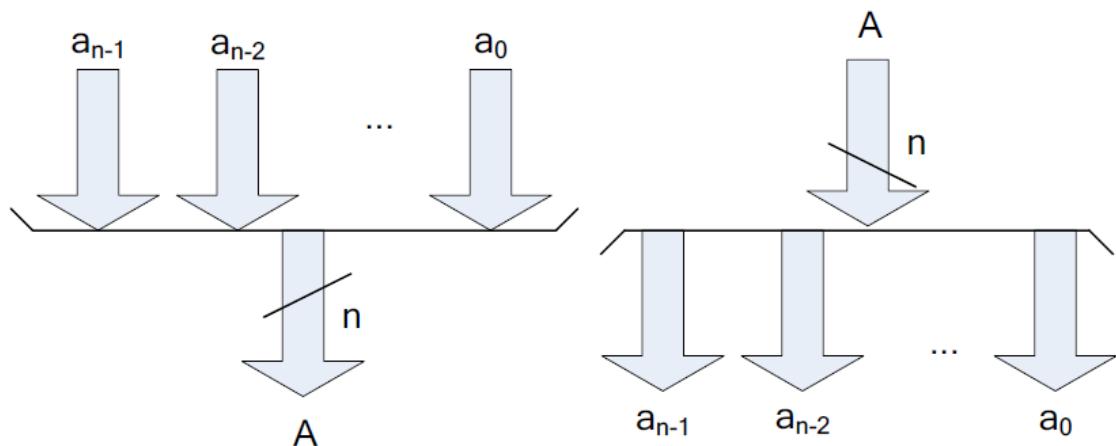
**Spiegazione:** Perché scrivi A in quella prima forma?

Perché, se  $A = \sum_{i=0}^{n-1} a_i \cdot \beta^i$ , allora sostituire  $a_i \rightarrow \beta - 1$  significa mettere  $\beta - 1$  ad ogni cifra,

che è appunto il valore massimo che quella cifra possa avere, in qualunque base!

## Circuiti logici per operazioni su Naturali

Per rappresentare il fatto che un numero è identificato da una sequenza di  $n$  cifre useremo questa notazione grafica dove ogni freccia rappresenta una cifra in base  $\beta$  generica:



Una variabile logica corrisponde ad una cifra in base  $\beta = 2$ .

**Quando si lavora in una base  $\beta \neq 2$  BISOGNA CODIFICARE LE SINGOLE CIFRE IN TERMINI DI VARIABILI LOGICHE!** (ad esempio usando la codifica BCD)

## Complemento

Dato  $A \equiv (a_{n-1}a_{n-2}\dots a_1a_0)_{\beta}$ , si definisce **COMPLEMENTO** di  $A$  in base  $\beta$  su  $n$  cifre:

$$\bar{A} = \beta^n - 1 - A$$

Ovvero è il massimo numero rappresentabile su  $n$  cifre in base  $\beta$ , meno il numero di cui voglio calcolare il complemento!

**ATTENZIONE! Il complemento richiede che si specifichi il numero di cifre!**



La prima cosa da chiedersi quando si definisce un'operazione è **SU QUANTE CIFRE STA IL RISULTATO!**

(Tip anche per gli es. d'esame di aritmetica!)

### Criterio pratico:

Ogni cifra di  $\bar{A}$  si ottiene complementando la corrispondente cifra di  $A$ :

$$\bar{A} \equiv (\overline{a_{n-1}} \ \overline{a_{n-2}} \ \dots \ \overline{a_1} \ \overline{a_0})_{\beta}$$

### Dimostrazione:

Si vede subito che anche  $0 \leq \bar{A} < \beta^n$ , quindi anche  $\bar{A}$  è rappresentabile su  $n$  cifre.  
 (perché  $A \in [0, \beta^n - 1] \implies 0 \leq \beta^n - 1 - A \leq \beta^n - 1$ )

Dalla definizione si ha

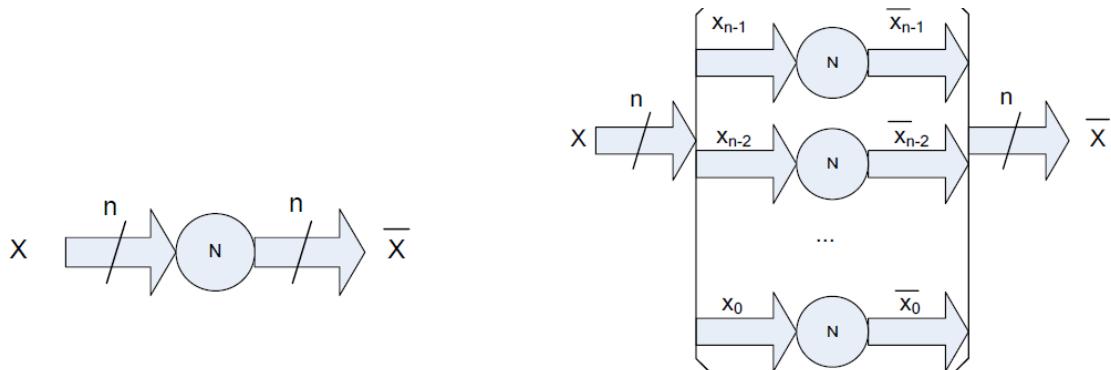
$$\bar{A} = \beta^n - 1 - A = \sum_{i=0}^{n-1} (\beta - 1) \cdot \beta^i - \sum_{i=0}^{n-1} a_i \cdot \beta^i = \sum_{i=0}^{n-1} (\beta - 1 - a_i) \cdot \beta^i$$

Dove il passaggio in **verde** è il questa osservazione sul massimo numero rappresentabile.

Ora, si vede che  $\beta - 1 - a_i$  è una cifra in base  $\beta$ , ed è il complemento della cifra  $a_i$ !

Da questo abbiamo dimostrato la frase sopra!

### Circuito logico per il complemento



Visione funzionale a sinistra e Dentro il circuito "N" a destra

Se la base è  $\beta = 2$  ogni circuito **N** sarà semplicemente una porta NOT.

Se la base è  $\beta \neq 2$  ogni circuito **N** sarà un **Sottrattore**, che avrà come ingressi:

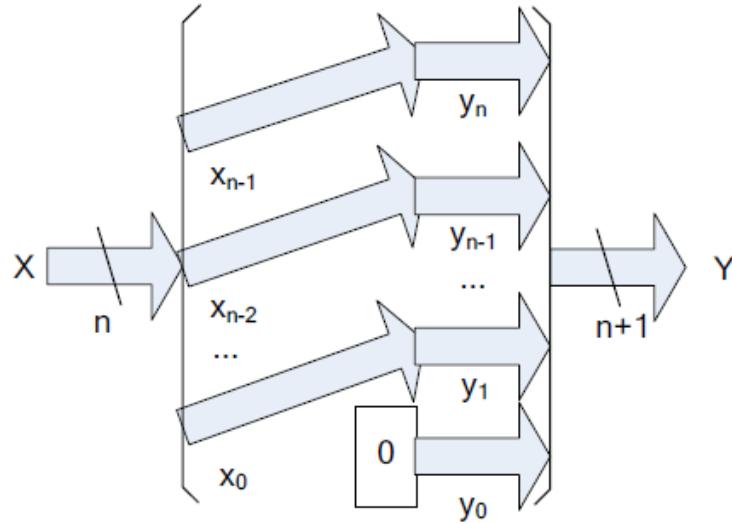
- Il massimo numero rappresentabile su 1 cifra in base  $\beta$ , cioè  $\beta - 1$
- La codifica della cifra  $i$ -esima del numero di cui voglio calcolare il complemento

E ciascuna rete  $N_i$  farà il complemento della corrispondente cifra, e poi riunendo e concatenando tutte le cifre in uscita avrò il numero complementato che volevo!

### Moltiplicazione per una potenza della base

#### Criterio pratico:

**Moltiplicare un numero su  $n$  cifre per  $\beta^k$  significa costruire un nuovo numero di  $n + k$  cifre, di cui le  $k$  cifre meno significative sono 0.**



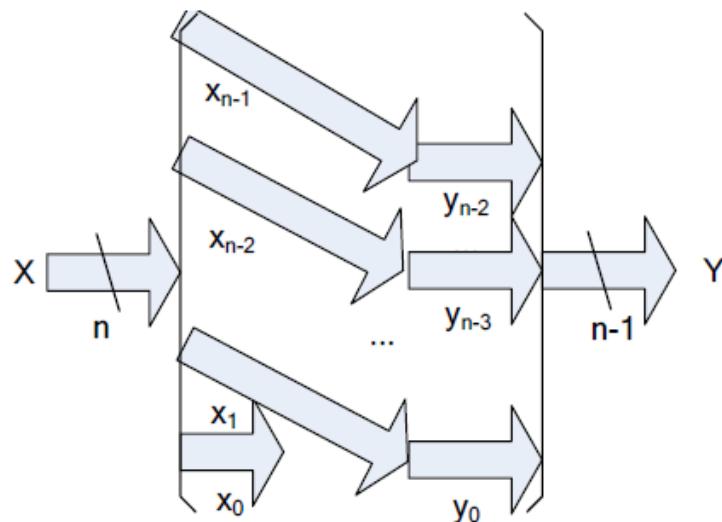
Circuito che calcola la moltiplicazione per  $\beta^1$

## Divisione per una potenza della base

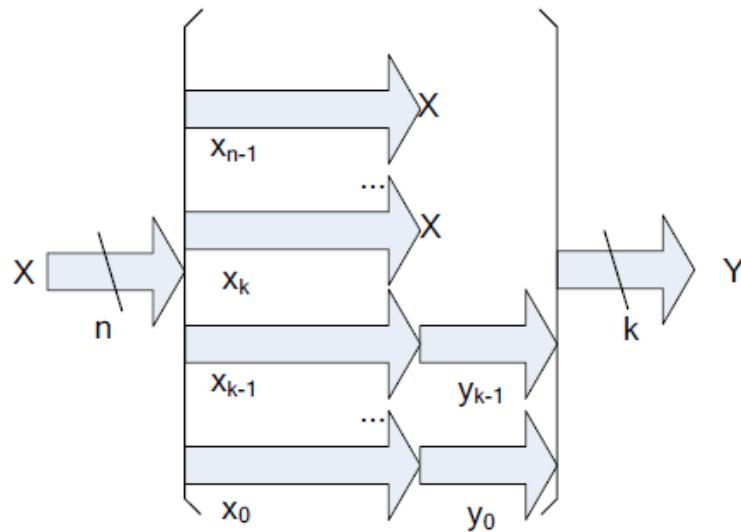
### Criterio pratico:

Dividere un numero su  $n$  cifre per  $\beta^k$  significa costruire DUE nuovi numeri!

- Il **quoziente** sono le  $n - k$  cifre più significative del numero di partenza
- Il **resto** sono le  $k$  cifre meno significative del numero di partenza



Circuito che calcola il QUOZIENTE della divisione per  $\beta^1$



Circuito che calcola il RESTO della divisione per  $\beta^k$

Queste reti sono dette **RETI DI SHIFT** e hanno complessità nulla.



Sarà considerato ERRORE moltiplicare e dividere per  $\beta^k$  non usando le Reti di Shift!

## Concatenamento

Dati due numeri:

- $Y$  su  $k$  cifre
- $Z$  su  $n - k$  cifre



L'operazione di **CONCATENAMENTO**  $X = Z \cdot \beta^k + Y$  E' DI COMPLESSITA' NULLA!

### Dimostrazione:

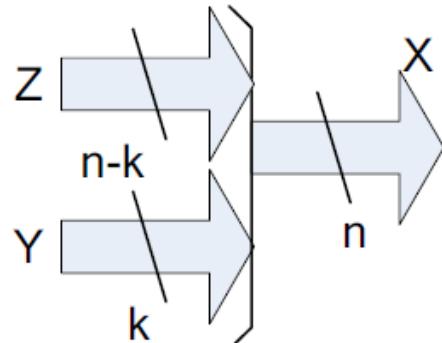
$$Z = (z_{n-k-1} z_{n-k-2} \dots z_1 z_0), Y = (y_{k-1} y_{k-2} \dots y_1 y_0)$$

Moltiplico  $Z$  per  $\beta^k$ :  $Z \cdot \beta^k = (z_{n-1} z_{n-2} \dots z_k | z_{k-1} \dots z_1 z_0)$  dove:

- Le cifre  $z_{n-1} \dots z_k$  sono le  $n - k$  cifre più significative di  $Z$

- Le nuove cifre  $z_{k-1} \dots z_0$  sono i  $k$  zeri aggiunti in coda a  $Z$

Quindi, sommare  $Y$  alle  $k$  cifre nulle  $z_{k-1} \dots z_0$  è di complessità nulla e il risultato che si ottiene è  $X = Z \cdot \beta^k + Y = (z_{n-1}z_{n-2} \dots z_k | y_{k-1}y_{k-2} \dots y_1 y_0)$



Circuito di concatenamento

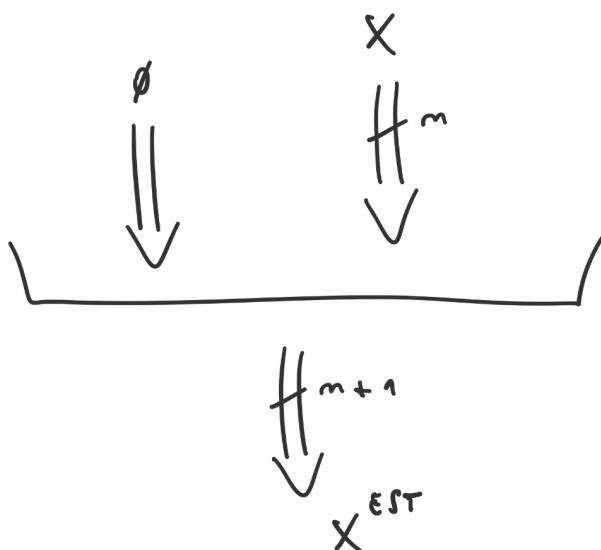
Dualmente, pure l'operazione di scomposizione di un numero di  $n$  cifre in due numeri di  $k$  ed  $n - k$  cifre ha complessità nulla!

## Estensione di campo ( $\mathbb{N}$ )

Dato  $X = (x_{n-1}x_{n-2} \dots x_1x_0)$ , lo si vuole rappresentare su un numero di cifre superiore.

### Criterio pratico per $\mathbb{N}$ :

**Si aggiunge uno zero in testa!**  $X^{EST} = (0 \mid x_{n-1}x_{n-2} \dots x_1x_0)$



Circuito per l'estensione di campo per  $\mathbb{N}$ aturali

## Addizione

### Algoritmo algebrico indipendente dalla base

1. Sommare le cifre di pari posto da destra verso sinistra
2. Se la somma di due cifre non è rappresentabile su una sola cifra, si usa il riporto, che vale sempre 0 (quando la somma è rappresentabile) o 1 (quando non è rappresentabile)

### Derivazione del criterio pratico

Dati:

- $X$  numero  $\mathbb{N}$ aturale in base  $\beta$  su  $n$  cifre
- $Y$  numero  $\mathbb{N}$ aturale in base  $\beta$  su  $n$  cifre
- $C_{in}$  numero  $\mathbb{N}$ aturale  $\in \{0, 1\}$  **RIPORTO ENTRANTE**

Voglio calcolare

$$Z = X + Y + C_{in}$$

dove  $C_{in}$  consente di rendere l'operazione di somma **modulare!**

**Su quante cifre sta il risultato?**

Visto che  $0 = 0 + 0 + 0 \leq X + Y + C_{in} \leq (\beta^n - 1) + (\beta^n - 1) + 1 = 2\beta^n - 1$  il risultato sta sempre su  $n + 1$  cifre in quanto  $2\beta^n - 1 \leq \beta^{n+1} - 1$  (dato che  $\beta \geq 2$ ).

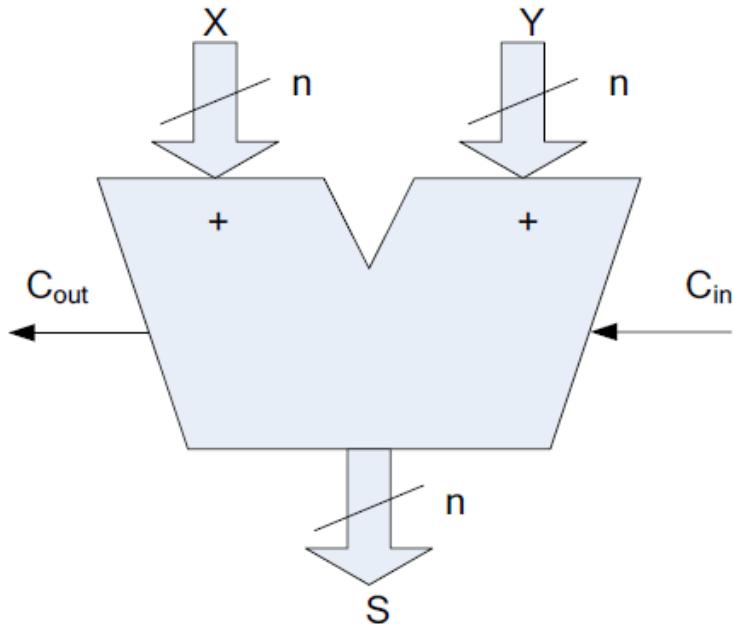
### Criterio pratico:

La somma di due numeri Naturali in base  $\beta$  su  $n$  cifre, più un eventuale riporto entrante  $C_{in} \in \{0, 1\}$ , produce un numero Naturale che è SEMPRE rappresentabile su  $n + 1$  cifre in base  $\beta$ .

La cifra di posto  $n + 1$  viene detta **RIPORTO USCENTE**  $C_{out} \in \{0, 1\}$ .

Se  $C_{out} = 0$  la somma è rappresentabile su  $n$  cifre.

Se  $C_{out} = 1$  la somma NON è rappresentabile su  $n$  cifre, ma su  $n + 1$ .



Visione funzionale di un Sommatore in base  $\beta$  ad  $n$  cifre per Naturali

**ATTENZIONE! Il numero di cifre di  $X$ ,  $Y$  e  $S$  DEVE ESSERE LO STESSO!**

### Note:

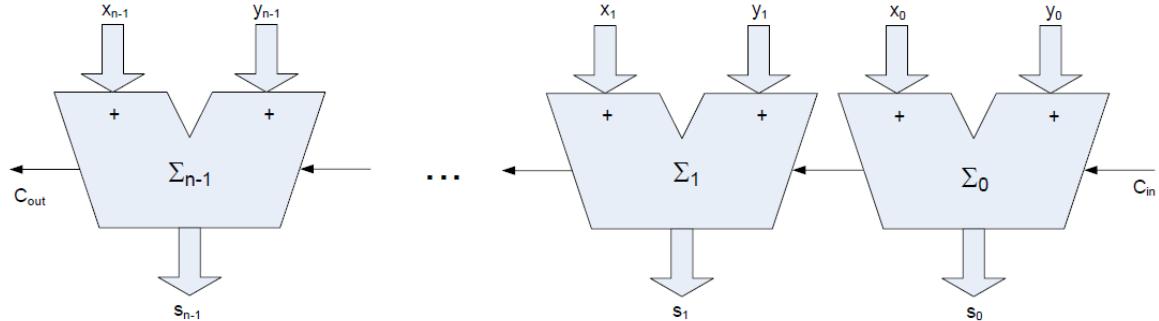
- Se gli addendi sono uno su  $n$  e l'altro su  $m$  cifre, bisogna estendere quello con meno cifre in modo da avere entrambi lo stesso numero di cifre! (Se  $n > m$  devi estendere quello ad  $m$  cifre inserendo  $n - m$  zeri in testa)
- Se gli addendi sono entrambi su  $n$  cifre, ma vuoi che la somma sia sempre rappresentabile, basta estendere entrambi gli addendi su  $n + 1$  cifre ed usare un sommatore ad  $n + 1$  cifre!

## Scomposizione della somma

La somma in base  $\beta$  su  $n$  cifre può essere scomposta in somme in base  $\beta$  su una cifra!

Basta sommare cifra per cifra e propagare il riporto!

Inoltre questo algoritmo è INDIPENDENTE DALLA BASE!



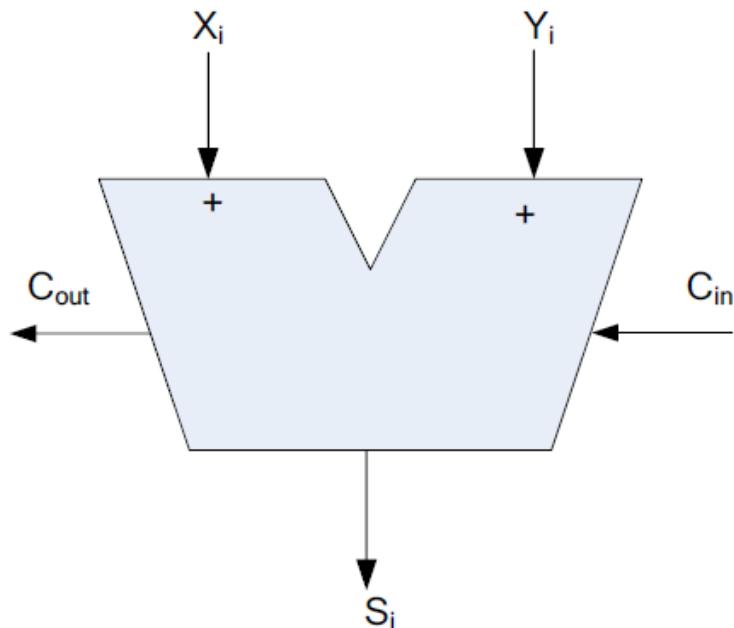
Visione funzionale del montaggio ripple carry con i full adder

Questo si chiama **MONTAGGIO RIPPLE CARRY** (propagazione del riporto).

Ogni  $\Sigma_i$  è un **sommatore ad una cifra**, e si chiama **FULL ADDER**.

## Implementazione e sintesi di un Full Adder in base $\beta = 2$

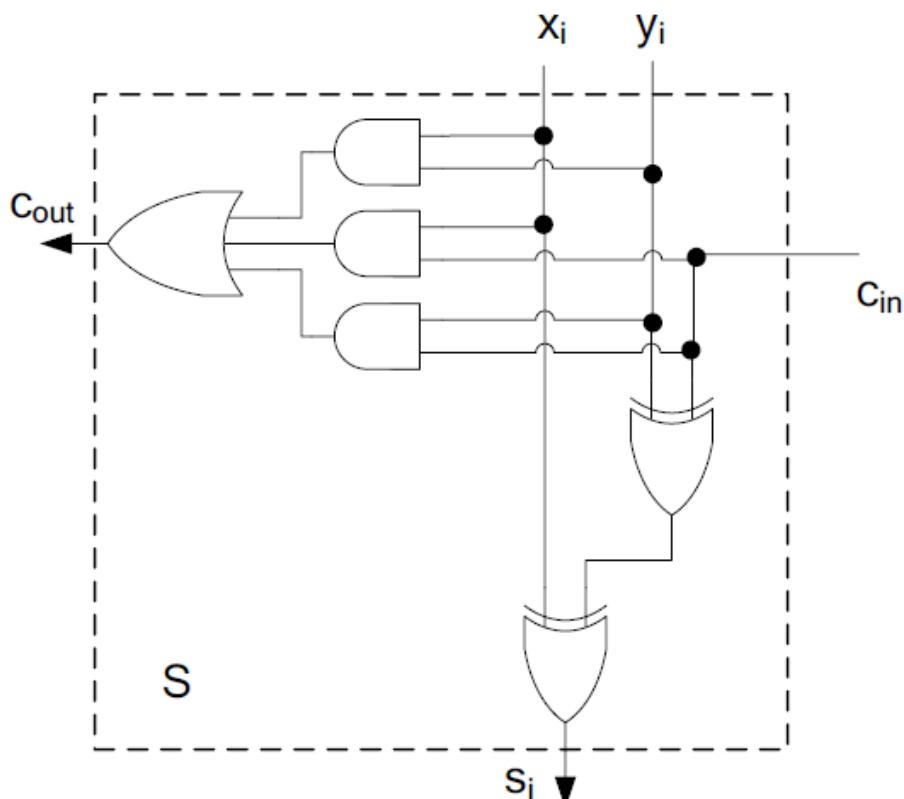
Sintetizziamo un Full Adder in base 2:



Visione funzionale di un full adder in base 2

$x_i$	$y_i$	$c_{in}$	$s_i$	$c_{out}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- Il riporto uscente  $C_{out}$  si sintetizza in Forma SP mediante mappa di Karnaugh
- La somma  $S_i$  si vede che è 1 se e solo se il numero di 1 in ingresso è dispari



Circuito Full Adder in base 2

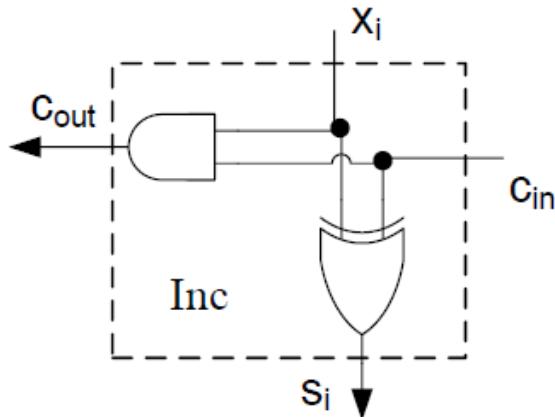
Questa è una possibile sintesi di un Full Adder in base 2, comunque il primo XOR fra  $y_i$  e  $C_{in}$  poteva essere messo pure fra  $x_i$  e  $y_i$ , e successivamente il secondo XOR farlo fra il risultato di questo e  $C_{in}$ .

In sostanza, basta fare  $x_i \oplus y_i \oplus C_{in}$  in una qualsiasi combinazione perché lo XOR è commutativo.

## Circuito incrementatore

Un incrementatore è un circuito che esegue  $S = X + C_{in}$ .

E' un caso particolare di un sommatore, quando  $Y = 0$ :



Circuito incrementatore in base 2

La sintesi è quella di un Full Adder, ma semplificata perché  $Y = 0$ .

In particolare:

- **Per la somma  $S_i$ :** Visto che  $Y = 0$  si ha  $0 \oplus C_{in} = 0$ , quindi il primo XOR sparisce
- **Per il riporto  $C_{out}$ :** Visto che  $Y = 0$  si ha  $x_i \wedge 0 = 0$  nella prima porta AND e  $0 \wedge C_{in} = 0$  nella terza porta AND, quindi si tolgono. A questo punto la porta OR in fondo avrebbe in input  $(0, x_i \wedge C_{in}, 0)$ , quindi si può togliere l'OR e lasciare solo la AND fra  $x_i$  e  $C_{in}$ .

## Sottrazione

### Algoritmo algebrico indipendente dalla base

1. Sottrarre le cifre di pari posto da destra verso sinistra
  2. Se la differenza di due cifre non è rappresentabile su una sola cifra, si usa il prestito per la coppia di cifre successive.
- Il prestito vale sempre 0 (quando la differenza è rappresentabile) o 1 (quando non è rappresentabile)

## Derivazione del criterio pratico

Dati:

- $X$  numero Naturale in base  $\beta$  su  $n$  cifre
- $Y$  numero Naturale in base  $\beta$  su  $n$  cifre
- $b_{in}$  numero Naturale  $\in \{0, 1\}$  **PRESTITO (BORROW)**

Voglio calcolare

$$Z = X - Y - b_{in}$$

dove  $b_{in}$  consente di rendere l'operazione di sottrazione **modulare!**

### Su quante cifre sta il risultato?

Visto che

$$-\beta^n = 0 - (\beta^n - 1) - 1 \leq X - Y - b_{in} \leq (\beta^n - 1) - 0 - 0 = \beta^n - 1$$

$Z$  può anche non essere un numero Naturale!

## Derivazione del criterio pratico per $D$

Ora scrivo  $Z$  come quoziente e resto di una divisione per  $\beta^n$ , ed osservando che il MINIMO quoziente della divisione di  $Z$  per  $\beta^n$  è  $-1$ , definisco  $-b_{out}$  come quoziente, e  $D$  come resto:

$$-b_{out} = \left\lfloor \frac{X - Y - b_{in}}{\beta^n} \right\rfloor, D = |X - Y - b_{in}|_{\beta^n}$$

Quindi, visto che  $b_{out} \in \{0, 1\}$ , ottengo:

$$Z = -b_{out} \cdot \beta^n + D = X - Y - b_{in} \quad (*)$$

Dunque  $D$  sta su  $n$  cifre!

### Criterio pratico per $D$ :

La differenza  $D$  tra due numeri Naturali in base  $\beta$  su  $n$  cifre, meno un eventuale prestito entrante  $b_{in}$ , produce un numero che, se Naturale, è SEMPRE rappresentabile su  $n$  cifre in base  $\beta$ .

Se invece produce un numero  $\mathbb{Z}$  intero, c'è un prestito uscente  $b_{out}$ .

In ogni caso,  $b_{out} \in \{0, 1\}$ .

## Derivazione del criterio pratico per $Z$

Ora devo trovare un criterio pratico per  $Z$ :

Visto che  $Y + \bar{Y} = \beta^n - 1$  ho che  $Y = -\bar{Y} + \beta^n - 1$ .

Ora sostituisco  $Y$  in (\*):

$$\begin{aligned} Z &= -b_{out} \cdot \beta^n + D = X - (-\bar{Y} + \beta^n - 1) - b_{in} = \\ &= X + \bar{Y} - \beta^n + 1 - b_{in} \end{aligned}$$

Ora, sistemando i due membri trovo:

$$\begin{aligned} \beta^n - b_{out} \cdot \beta^n + D &= X + \bar{Y} + 1 - b_{in} \\ \Rightarrow (1 - b_{out}) \cdot \beta^n + D &= X + \bar{Y} + (1 - b_{in}) \\ \Rightarrow \bar{b}_{out} \cdot \beta^n + D &= X + \bar{Y} + \bar{b}_{in} \end{aligned}$$

Dove, visto che  $b_{in}$  e  $b_{out} \in \{0, 1\}$  entrambi, posso usare la proprietà delle variabili logiche tale per cui  $1 - x = \bar{x}$  senza inficiare alla generalità di questa formula (che **RESTA VALIDA PER QUALUNQUE BASE  $\beta$** , infatti non è ancora stata fissata  $\beta$ !)

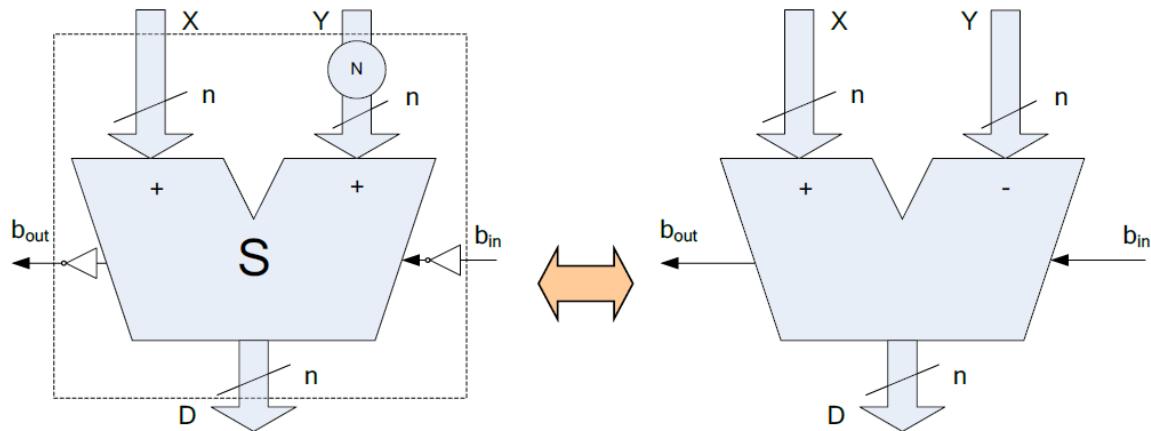
#### Criterio pratico per $Z$ :

La differenza fra  $X$  e  $Y$ , meno un eventuale **PRESTITO ENTRANTE**  $b_{in}$ , se è un numero  $\mathbb{N}$  naturale può essere ottenuta sommando  $X$ ,  $\bar{Y}$ , e un eventuale **RIPORTO ENTRANTE**  $\bar{b}_{in}$ .

- Se il **RIPORTO USCENTE**  $\bar{b}_{out}$  della somma  $X + \bar{Y} + \bar{b}_{in}$  è 1, la differenza è un numero  $\mathbb{N}$ aturale pari a  $D$ , e il **PRESTITO USCENTE**  $b_{out}$  è 0.
- Se il **RIPORTO USCENTE**  $\bar{b}_{out}$  della somma  $X + \bar{Y} + \bar{b}_{in}$  è 0, la differenza NON è un numero  $\mathbb{N}$ aturale, e il **PRESTITO USCENTE**  $b_{out}$  è 1.

#### **Spiegazione:**

- Se  $\bar{b}_{out} = 1 \Rightarrow b_{out} = 0$  e quindi  $D \in \mathbb{N}$  è il risultato corretto
- Se  $\bar{b}_{out} = 0 \Rightarrow b_{out} = 1$  e quindi  $D \in \mathbb{Z}$  e il risultato NON E' CORRETTO!



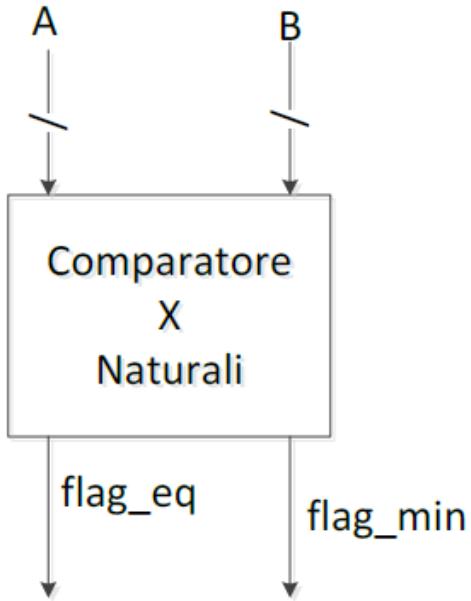
Visione funzionale di un Sottrattore in una base generica

Terminologia:

- $b_{in}$  PRESTITO (BORROW) ENTRANTE
- $b_{out}$  PRESTITO (BORROW) USCENTE
- $\overline{b_{in}}$  RIPORTO (CARRY) ENTRANTE
- $\overline{b_{out}}$  RIPORTO (CARRY) USCENTE

### Circuito comparatore in base $\beta = 2 (\mathbb{N})$

Se voglio sapere se un numero Naturale  $A$  è minore di un altro  $B$ , basta che li sottraggo e guardo il prestito uscente: se  $b_{out} = 1$  allora  $A < B$  perché si è generato un prestito uscente dall'ultima cifra.



Visione funzionale di un Comparatore per Naturali in base 2

- $flag_{eq} = 1 \iff A = B$

#### **Spiegazione:**

Questa uscita è il risultato dello XOR bit a bit di  $A$  e  $B$ , passato poi ad una NOR.

Se  $A = B$  le  $n$  XOR restituiscono tutti zeri, che entrano nella NOR e quindi il suo risultato è  $flag_{eq} = 1$ .

Se invece  $A \neq B$  le  $n$  XOR restituiscono almeno un uno, quindi, visto che una NOR è un OR con in cascata un inverter, la OR restituisce 1, di conseguenza la NOT restituisce  $flag_{eq} = 0$ .

- $flag_{min} = 1 \iff A < B$

#### **Spiegazione:**

Questa uscita è il prestito uscente  $b_{out}$  di un circuito sottrattore con ingressi  $A$  e  $B$ .

Se  $A < B$  allora  $b_{out}$  segnala che  $A - B < 0$ , quindi  $b_{out} = 1$ .

Se invece  $A - B > 0$  (quindi  $A - B$  è rappresentabile come Naturale),  $b_{out} = flag_{min} = 0$ .

## Moltiplicazione

### Algoritmo algebrico indipendente dalla base

1. Moltiplicare uno dei due fattori per TUTTE le cifre dell'altro
2. I risultati di ciascuno di questi prodotti parziali vengono scritti a partire dalla cifra corrente

3. Infine si sommano i prodotti parziali, con eventuale riporto, per ottenere il prodotto finale

## Derivazione del criterio pratico

Dati:

- $X$  numero Naturale in base  $\beta$  su  $n$  cifre
- $Y$  numero Naturale in base  $\beta$  su  $m$  cifre
- $C$  numero Naturale in base  $\beta$  su  $n$  cifre

Voglio calcolare

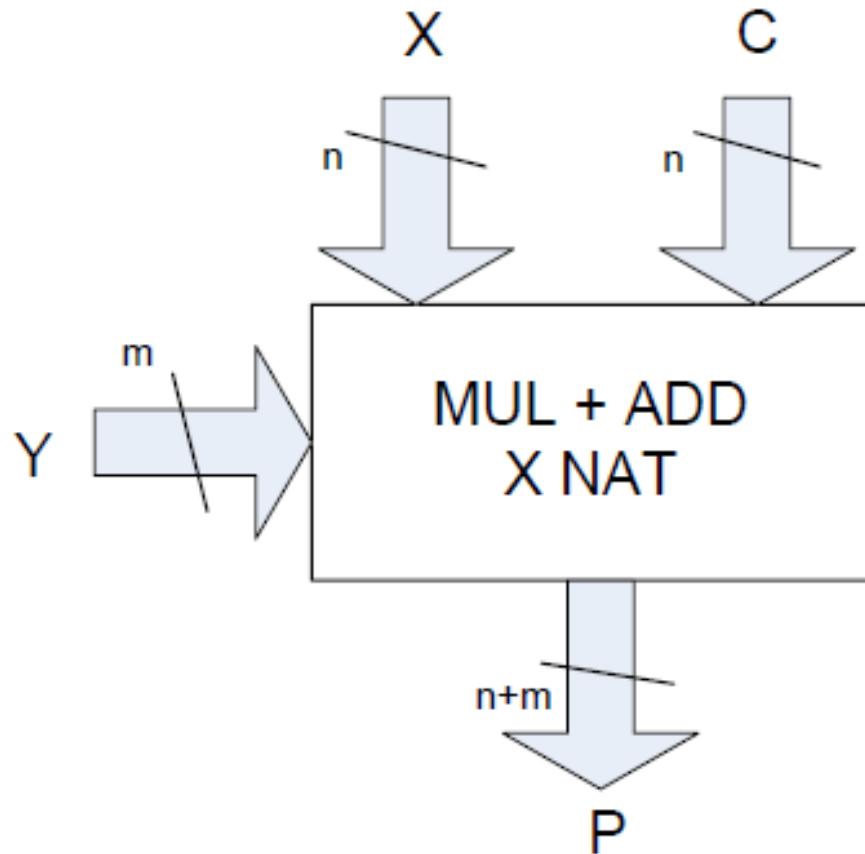
$$P = X \cdot Y + C$$

dove  $C$  consente di rendere l'operazione di prodotto **modulare!**

**Su quante cifre sta il risultato?**

$$\begin{aligned} P &= X \cdot Y + C \leq (\beta^n - 1) \cdot (\beta^m - 1) + (\beta^n - 1) = \\ &= (\beta^n - 1) \cdot ((\beta^m - 1) + 1) = (\beta^n - 1) \cdot \beta^m = \\ &= \beta^{n+m} - \beta^m < \beta^{n+m} - 1 \end{aligned}$$

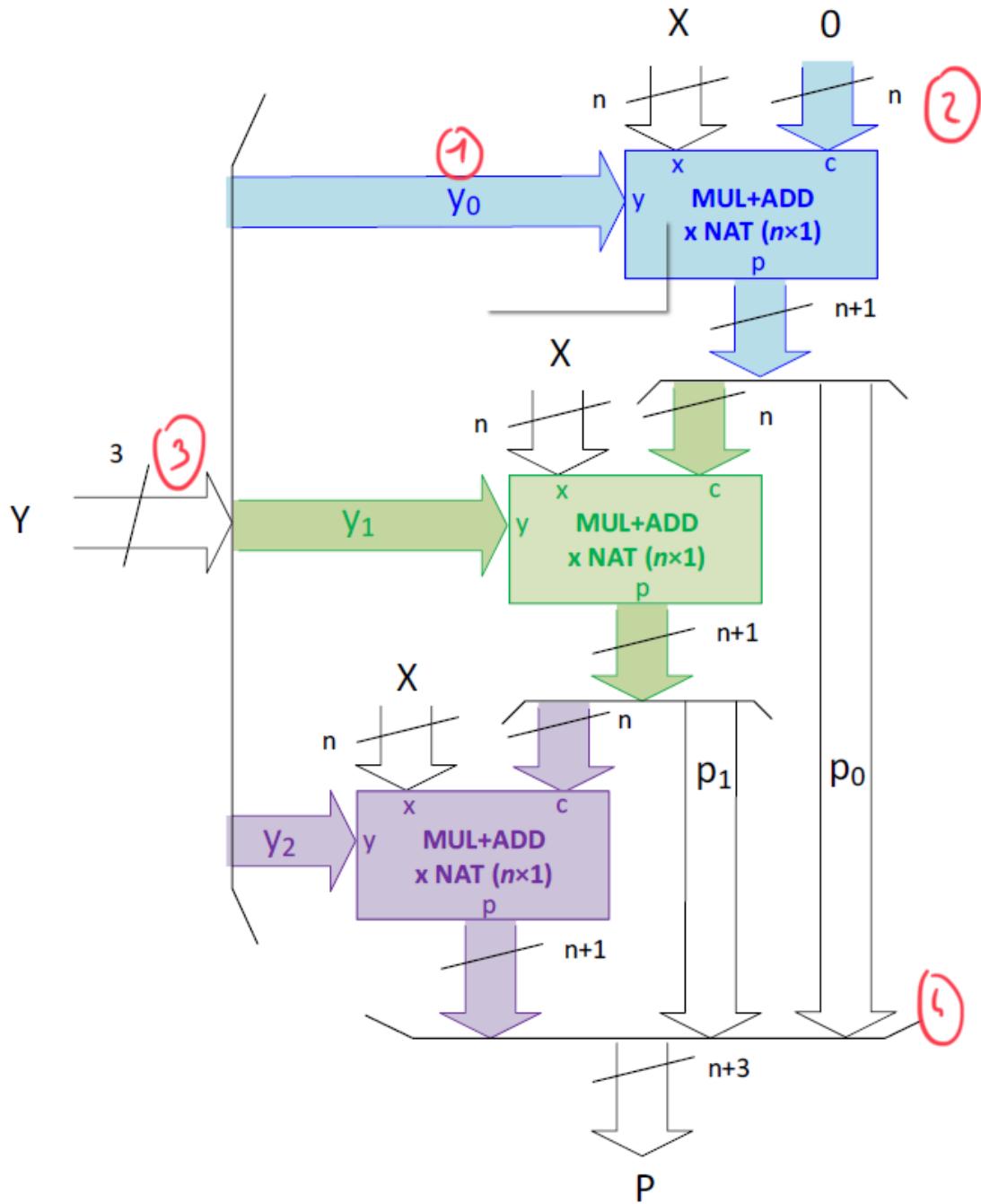
**Il prodotto  $P$  quindi è rappresentabile su  $n + m$  cifre!**



Visione funzionale di un Moltiplicatore in base generica

#### Criterio pratico:

- **Moltiplicare  $X$  per una cifra di  $Y$ , e successivamente sommare  $C$  (inizialmente è nullo) usando un sommatore su  $n + 1$  cifre, così è sempre rappresentabile**
- **La cifra meno significativa (LSD) diventa l' $i$ -esima cifra del prodotto  $P$**
- **Le altre cifre diventano il nuovo termine  $C$  da sommare al prossimo passo**
- **Infine si ottiene il prodotto  $P$  concatenando tutti i risultati**

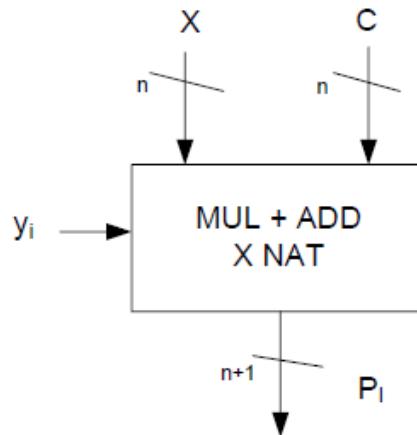


Circuito moltiplicatore in una base generica

1. Quella è la prima cifra di  $Y$
2. Per non avere problemi di rappresentabilità della somma fra  $X \cdot y_i$  e  $C$ , si usano, dentro i MUL+ADD, dei sommatori ad  $n$  cifre, concatenando poi  $C_{out}$  con  $S$ , così da avere il risultato sempre rappresentabile su  $n + 1$  cifre, anche all'inizio quando  $C = 0$
3. In questo esempio per semplicità viene posto  $m = 3$
4. Proprio per l'osservazione precedente, con  $m$  generico qui avremmo le  $m - 1$  cifre meno significative già a posto (infatti nell'esempio si hanno  $3 - 1 = 2$  cifre a posto) e il MUL+ADD

finale restituirebbe le  $n + 1$  cifre più significative, cosicché concatenando infine  $(n + 1|m - 1)$  si ottiene il risultato  $P$  su  $n + m$  cifre

### Circuito MUL+ADD $n \times 1$ in base $\beta = 2$

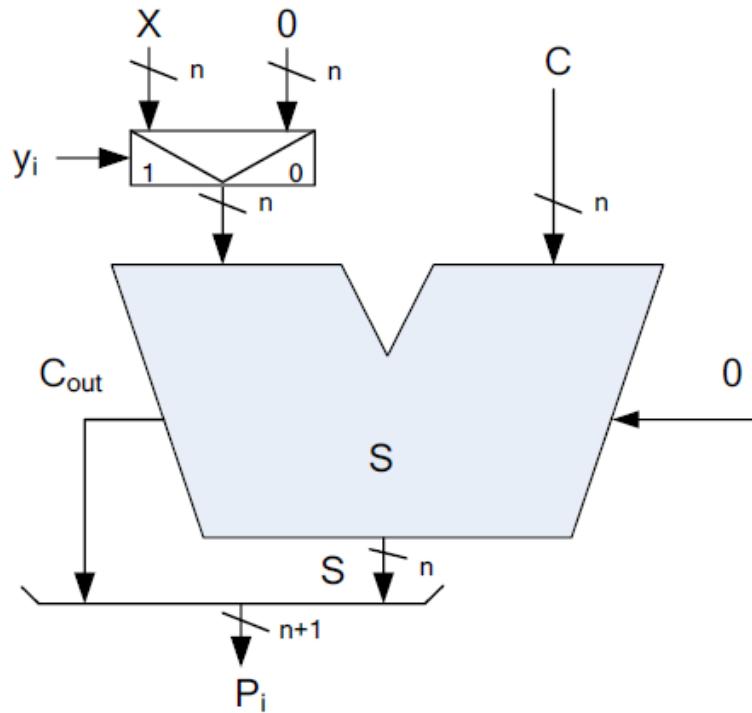


Visione funzionale di un MUL+ADD per Naturali in base 2

Il risultato di questo circuito deve essere:

$$P_i = y_i \cdot X + C = \begin{cases} 0 + C & \text{se } y_i = 0 \\ X + C & \text{se } y_i = 1 \end{cases}$$

Quindi, questo circuito sulla base del valore di  $y_i$  deve sommare  $X$  oppure 0 a  $C$ !



Circuito Moltiplicatore in base 2 (il sommatore lo ricavi da sopra)

Quindi, basta mettere un Multiplexer, con variabile di comando  $y_i$ :

- Se  $y_i = 0$  l'uscita del MUX insegue 0, quindi il sommatore esegue  $0 + C$
- Se  $y_i = 1$  l'uscita del MUX insegue  $X$ , quindi il sommatore esegue  $X + C$

Note:

**Attenzione che gli ingressi sono ad  $n$  bit ma il risultato, visto che lo concateno con  $C_{out}$ , è ad  $n + 1$  bit!**

**Lo posso fare SOLO PERCHE' SONO IN BASE  $\beta = 2$  e quindi  $C_{out}$  è anche una cifra perché è un bit!**

**In realtà, quello non è un unico MUX ma sono  $n$  MUX 2-to-1**, tutti aventi la STESSA FISSATA  $y_i$  come variabile di comando, ma ciascuno che sceglie se far passare il bit  $x_j$  (con  $j = 0 \dots n - 1$ ) oppure un bit 0!

Inoltre, volendo ottimizzare, quei  $n$  MUX 2-to-1 possono essere sostituiti da altrettante ( $n$ ) porte AND ( $x_j \wedge 1 = x_j$ ,  $x_j \wedge 0 = 0$ ), che hanno in ingresso  $x_j$  e  $y_i$  (attenzione ancora:  $j = 0 \dots n - 1$  ma  $i$  è FISSATO!!!).

## Divisione

### Algoritmo algebrico indipendente dalla base

1. Prendo il **MINIMO NUMERO NECESSARIO** ( $m + 1$ ) delle cifre più significative di  $X$  in modo da ottenere un numero compreso in  $[Y, \beta \cdot Y]$  (**cifra in base  $\beta$** ).  
Il numero che sto considerando lo chiamo  $X'$ .
2. Calcolo un quoziente e resto parziali  $q_i$  e  $R$  della divisione di  $X'$  per  $Y$ .  
Di sicuro  $q_i$  sta su **una sola cifra**, perché  $X' < \beta \cdot Y$ , quindi  $q_i < \beta$ .
3. Ottengo un nuovo dividendo  $X'$  concatenando il resto  $R$  con la cifra più significativa  $X_k$  non ancora utilizzata del dividendo  $X$  originale.  
Visto che  $R < Y \Rightarrow R_i < \beta$  ed  $X_k < \beta$ , il numero  $X' = (R_{m-1} \dots R_0 | X_k) < \beta \cdot Y = (Y_{m-1} \dots Y_1 Y_0 | 0)$ .
4. Finché non finiscono le cifre di  $X$ , continua dal passo 1.
5. Il quoziente  $Q$  è ottenuto concatenando i quozienti parziali  $q_i$  che stanno tutti su una sola cifra
6. Il resto  $R$  è il resto dell'ultima divisione parziale.

### Derivazione del criterio pratico

Dati:

- $X$  numero Naturale in base  $\beta$  su  $n + m$  cifre **DIVIDENDO**
- $Y$  numero Naturale in base  $\beta$  su  $m$  cifre **DIVISORE**

Voglio calcolare i due numeri  $Q$  (**QUOZIENTE**) ed  $R$  (**RESTO**) tali che

$$X = Q \cdot Y + R$$

( $Q$  ed  $R$  sono unici per il **Th. Div. con Resto**)

**Fattibilità dell'operazione:** questa rete dovrà avere anche un'uscita *no\_div* per tener conto del caso in cui  $Y = 0$  e quindi la divisione non si può fare.

Per ora mettiamoci nel caso  $Y > 0$ :

Su quante cifre stanno  $Q$  ed  $R$ ?

- Il Resto  $R$ , dato che  $R < Y$  per definizione, sta su  $m$  cifre
- Il Quoziente  $Q$  nel caso peggiore sta su  $n + m$  cifre (quando  $Y = 1$ )

**Io però VOGLIO CHE  $Q$  SIA RAPPRESENTABILE SU SOLE  $n$  cifre!!!**

Questo implica che

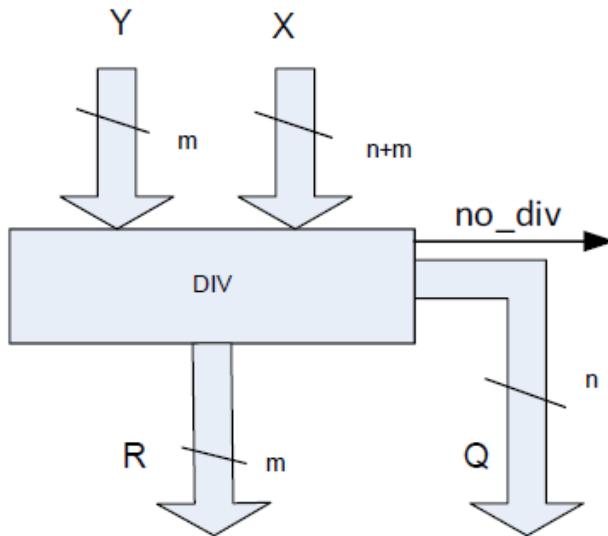
$$X = Q \cdot Y + R \leq (\beta^n - 1) \cdot Y + (Y - 1) = \beta^n \cdot Y - 1$$

Quindi  $X \leq \beta^n \cdot Y - 1 \implies X < \beta^n \cdot Y$  visto che  $X \in \mathbb{N}!!!$

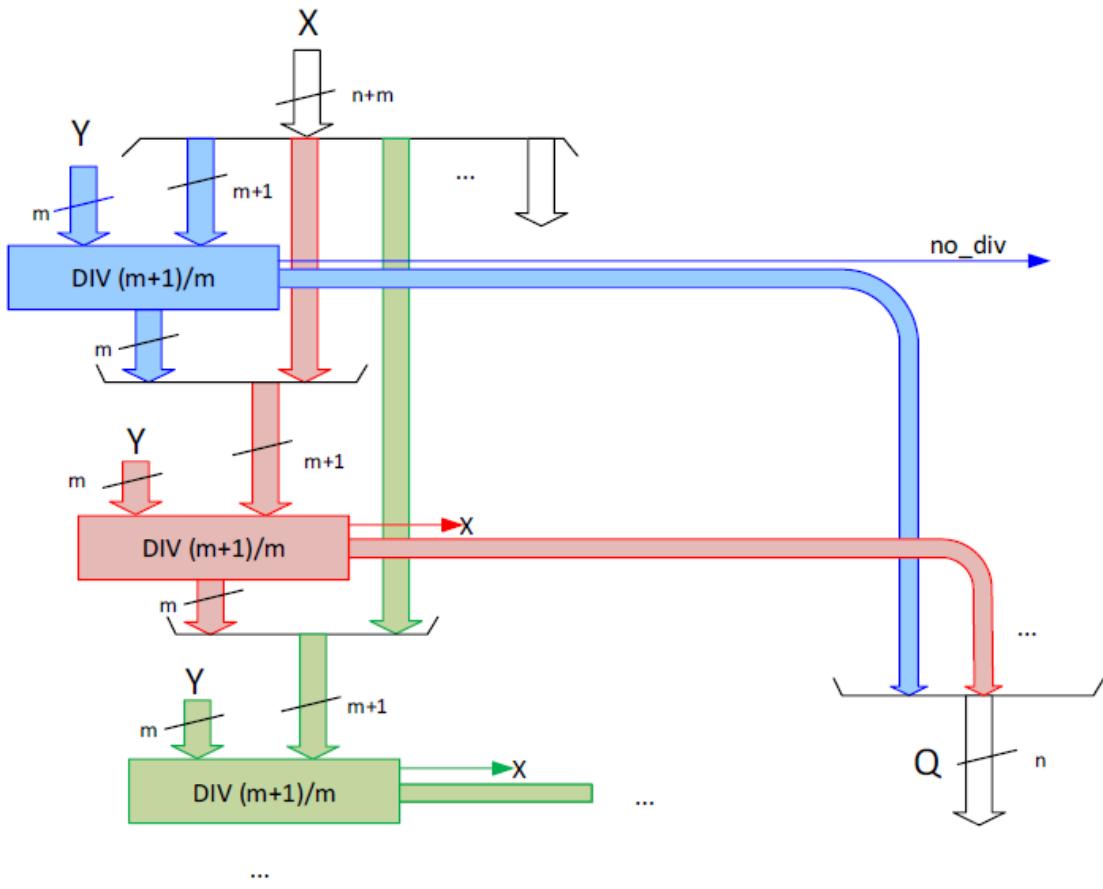
Questa ipotesi non è restrittiva se posso estendere a piacere la rappresentazione di  $X$  ed abbia un numero di cifre sufficienti per  $Q$ .

**Il problema (la restrittività dell'operazione presentata dall'aggiunta di questa ipotesi) si presenta quando si lavora su campi finiti, cioè all'interno di un calcolatore!!!**

**Quindi se  $Q$  non sta su  $n$  cifre, il circuito divisore deve settare la variabile logica  $no\_div$ , per indicare che la divisione non è fattibile!**



Visione funzionale di un Divisore in una base generica



Circuito Divisore in una base generica

Come si setta l'uscita *no\_div* a questo punto?

**Dire  $X < \beta^n \cdot Y$  equivale a dire che le  $m$  cifre più significative di  $X$  rappresentano un numero più piccolo di  $Y$ :**

$$\begin{array}{cccccc|ccccc} X: & x_{n+m-1} & x_{n+m-2} & \dots & x_{n+1} & x_n & x_{n-1} & \dots & x_0 \\ \beta^n \cdot Y: & y_{m-1} & y_{m-2} & \dots & y_1 & y_0 & 0 & \dots & 0 \end{array}$$

Quindi  $X < \beta^n \cdot Y \iff (x_{n+m-1} x_{n+m-2} \dots x_{n+1} x_n)_\beta < Y$ .

(stessa frase di sopra, ma detta in matematiche)

A sinistra del divisorio ci sono  $m$  cifre, a destra  $n$ .

Nel primo modulo DIV  $(m + 1)/m$  ci vanno le  $m + 1$  cifre più significative di  $X$ , che sono  $(x_{n+m-1} x_{n+m-2} \dots x_{n+1} x_n x_{n-1})$ , assieme alle cifre di  $Y = (y_{m-1} y_{m-2} \dots y_1 y_0)$ , quindi **l'uscita no\_div del circuito sottrattore finale è data dall'uscita no\_div del PRIMO MODULO DIV  $(m + 1)/m$ !**

**Perché già lì ho tutte le cifre necessarie per verificare la condizione  $X < \beta^n \cdot Y$ !**

## Circuito DIV $(m + 1)/m$ in base $\beta = 2$ ( $n = 1$ )

Ciascuna di queste unità produce:

- Un quoziente parziale  $q_i \in \{0, 1\}$  su una cifra
- Un Resto  $R$  su  $m$  cifre

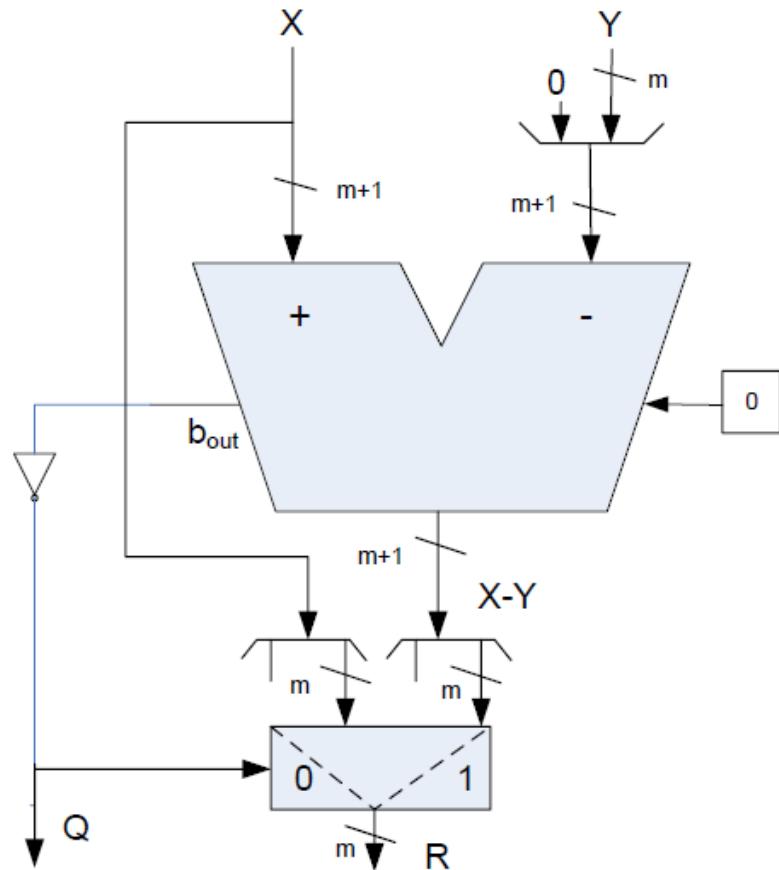
$$q_i = \begin{cases} 0 & \text{se } X < Y \\ 1 & \text{se } X \geq Y \end{cases}, \quad R = \begin{cases} X & \text{se } X < Y \\ X - Y & \text{se } X \geq Y \end{cases}$$

Vedendola in un altro modo:

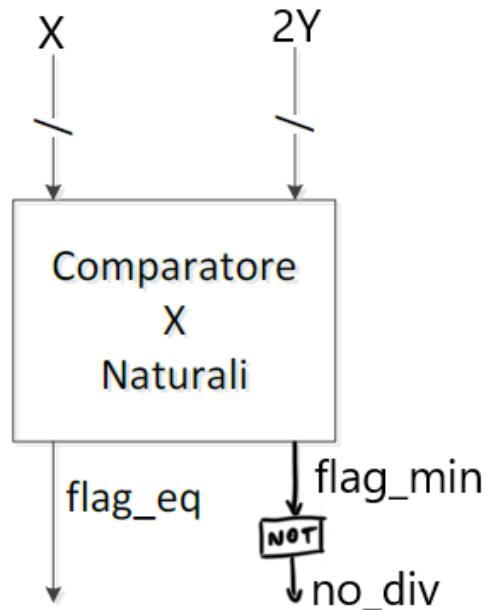
- Se  $X < Y \Rightarrow q_i = 0, R = X$   
Riprova:  $X = Q \cdot Y + R = 0 + X = X$

- Se  $X \geq Y \Rightarrow q_i = 1, R = X - Y$   
Riprova:  $X = Q \cdot Y + R = 1 \cdot Y + (X - Y) = Y + X - Y = X$

Quindi tutto ciò che deve fare questo circuito è stabilire se  $X < Y$  o viceversa, ed eventualmente sottrarli. Quindi basta usare un sottrattore per sintetizzare  $q_i$  ed  $R$ , e un circuito comparatore per Naturali per sintetizzare *no\_div*!



Parte di modulo DIV che sintetizza  $q_i$  ed  $R$



Rilevatore di fattibilità (parte di modulo che sintetizza  $no\_div$ )

Questo circuito sintetizza l'uscita  
 $no\_div = \overline{flag\_min}$  dell'intero circuito divisore totale.

Nel comparatore, c'è  $2Y$  perché voglio comparare  $X$  con  $\beta \cdot Y$  in base  $\beta = 2$ .

$\overline{flag\_min} = no\_div = 1 \iff X \geq 2Y$  (cioè  $q_i$  non sta su 1 cifra)

# Aritmetica degli $\mathbb{Z}$ interi

## Rappresentazione dei numeri $\mathbb{Z}$ interi

Preso un insieme di  $\beta^n$  numeri  $\mathbb{Z}$ interi, posso sempre trovare una legge **BIUNIVOCA** che gli fa corrispondere un insieme di  $\beta^n$  numeri  $\mathbb{N}$ aturali!

**Posso associare un insieme di  $n$  cifre in base  $\beta$  ad un numero  $\mathbb{Z}$ intero!**

Questa sarà la rappresentazione del numero  $\mathbb{N}$ aturale che gli faccio corrispondere.

Definisco questa corrispondenza:  $L(\cdot) : \mathbb{Z} \rightarrow \mathbb{N}$



D'ora in poi si scriveranno i numeri  $\mathbb{N}$ aturali con una MAIUSCOLA, e in numeri  $\mathbb{Z}$ interi con una MINUSCOLA.

Sbagliare una maiuscola con una minuscola sarà ERRORE!

Scegliendo opportunamente la legge  $L$ , è possibile sintetizzare dei circuiti che operano in maniera corretta sia che si interpretino le cifre in ingresso come rappresentazione di numeri  $\mathbb{N}$ aturali, sia come rappresentazione di  $\mathbb{Z}$ interi!

## Dominio e codominio della legge $L$

L'intervallo che si prende come dominio mi conviene sceglierlo il più simmetrico possibile rispetto all'origine, in modo che più operazioni possibili siano fattibili (e.g. calcolo dell'opposto)!

Siccome lavoro su  $n$  cifre in base  $\beta$ , posso rappresentare  $\beta^n$  numeri, quindi dividerò l'intervallo in due di  $\beta^n/2$ . C'è un problema però: devo rappresentare lo zero, e questo mi rende per forza di cose l'intervallo che scelgo asimmetrico. Si sceglie di rappresentare un numero negativo in più:

$$\left[ -\frac{\beta^n}{2}, \frac{\beta^n}{2} - 1 \right]$$

Per quanto riguarda il codominio è ovvio, operando con  $\mathbb{N}$ aturali su  $n$  cifre in base  $\beta$ !

Dunque la legge  $L$  sarà fatta così:

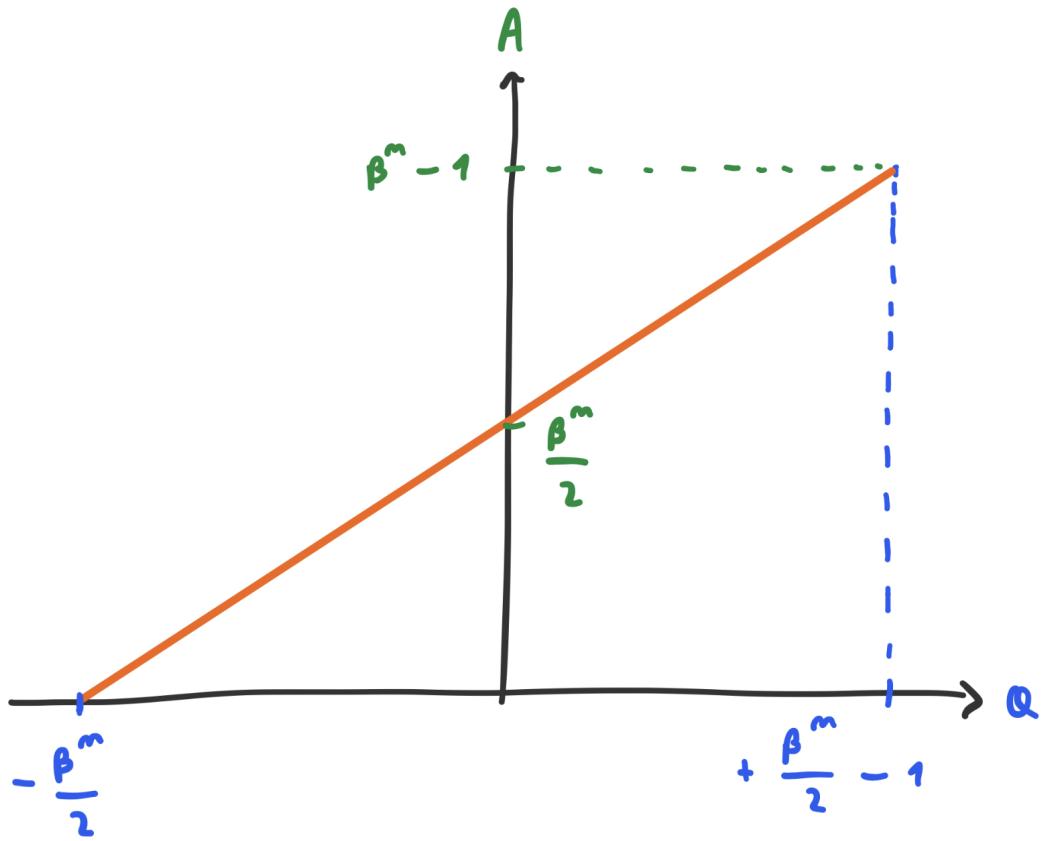
$$L : \left[ -\frac{\beta^n}{2}, \frac{\beta^n}{2} - 1 \right] \rightarrow [0, \beta^n - 1]$$

## Possibili scelte della legge $L$

### Traslazione

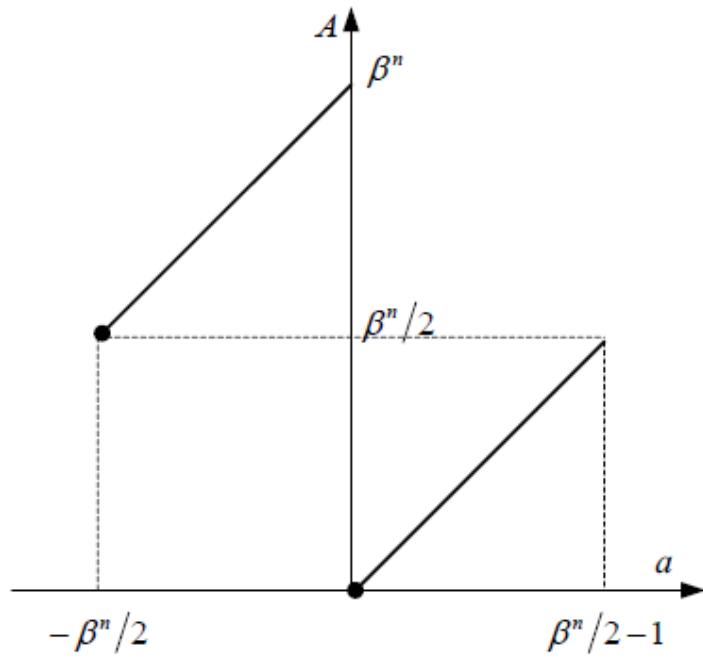
$$L : A = a + \frac{\beta^n}{2}$$

Questa legge  $L$  ha il vantaggio essere monotona, e viene usata nei convertitori A/D e D/A!



### Complemento alla radice

$$L : A = \begin{cases} a & \text{se } 0 \leq a < \frac{\beta^n}{2} \\ a + \beta^n & \text{se } -\frac{\beta^n}{2} \leq a < 0 \end{cases}$$



In altre parole:

- $a \geq 0 \Rightarrow A = a$
- $a < 0 \Rightarrow A = a + \beta^n$

Questa legge non è monotona su tutto l'intervallo di  $a$ , ma è monotona per  $a < 0$  e per  $a > 0$

## Modulo e segno

Questa legge non mettere in corrispondenza intervalli di  $\mathbb{Z}$  interi con intervalli di  $\mathbb{N}$  aturali!

Fa corrispondere ad un  $\mathbb{Z}$  intero una COPPIA di numeri  $(M, s)$ :

- $M \in \mathbb{N}$  **MODULO**
- $s \in \{0, 1\}$  **SEGNO** (variabile logica)

$$M = ABS(a), \quad s = \begin{cases} 0 & \text{se } a \geq 0 \\ 1 & \text{se } a < 0 \end{cases}$$

Fra queste 3 possibili leggi  $L$ , quella usata nei calcolatori è quella del **Complemento alla radice**.

Vediamone quindi le proprietà!

## Proprietà della Legge Complemento alla radice

### Determinazione del segno

#### Derivazione del criterio:

Sappiamo che:

$$\begin{aligned} a \geq 0 &\iff 0 \leq A < \frac{\beta^n}{2} \\ a < 0 &\iff \frac{\beta^n}{2} \leq A < \beta^n \end{aligned}$$

Quindi, per determinare il segno di  $a$  mi basta capire se  $A$  è  $\geq 0 <$  di  $\frac{\beta^n}{2}$ .

So che  $\frac{\beta^n}{2} = (\frac{\beta}{2} 0 0 \dots 0 0)_\beta$ , quindi

$$\frac{\beta^n}{2} - 1 = \left( (\frac{\beta}{2} - 1) (\beta - 1) (\beta - 1) \dots (\beta - 1) (\beta - 1) \right)_\beta$$

Quindi, per capire se  $A$  è  $\geq 0 <$   $\frac{\beta^n}{2}$  mi basta guardare la cifra più significativa!

E visto che per determinare il segno di  $a$  mi basta capire questo, ho finito!

**Per determinare il segno basta guardare la cifra più significativa di  $a$ !**

$$\begin{aligned} \text{se } a_{n-1} < \frac{\beta}{2} &\iff 0 \leq A < \frac{\beta^n}{2} \\ \text{se } a_{n-1} \geq \frac{\beta}{2} &\iff \frac{\beta^n}{2} \leq A < \beta^n \end{aligned}$$

#### Criterio pratico per determinare il segno:

- $a \geq 0 \iff a_{n-1} < \frac{\beta}{2}$
- $a < 0 \iff a_{n-1} \geq \frac{\beta}{2}$

### Criterio pratico per determinare il segno in base $\beta = 2$ :

- $a \geq 0 \iff a_{n-1} = 0$
- $a < 0 \iff a_{n-1} = 1$

### Legge inversa

#### Derivazione:

Si ottiene per **sostituzione**:

Abbiamo detto che la legge  $L$  è la seguente:

$$L : A = \begin{cases} a & \text{se } 0 \leq a < \frac{\beta^n}{2} \\ a + \beta^n & \text{se } -\frac{\beta^n}{2} \leq a < 0 \end{cases}$$

Per determinare  $L^{-1}$  basta:

- **Per determinare la relazione fra  $a$  ed  $A$ :**

- Nel ramo di sopra  $a = A$ , e così resta
- Nel ramo di sotto  $A = a + \beta^n \implies a = A - \beta^n$

- **Per determinare gli intervalli:**

- Nel ramo di sopra scambia  $a$  con  $A$  e basta
- Nel ramo di sotto, ancora,  $A = a + \beta^n \implies a = A - \beta^n$ .

Vuoi trovare l'intervallo in funzione di  $A$ , quindi sostituisci  $a = A - \beta^n$  nell'intervallo e trovi:  $-\frac{\beta^n}{2} \leq A - \beta^n < 0$ , quindi sommi  $\beta^n$  a tutto per trovarlo in funzione di  $A$  :

$$-\frac{\beta^n}{2} + \beta^n \leq A - \beta^n + \beta^n < 0 + \beta^n \implies \frac{\beta^n}{2} \leq A < \beta^n.$$

**Modo alternativo:** l'intervallo è in funzione di  $a$ , e io lo voglio in funzione di  $A$ .

So che  $A = a + \beta^n$ , quindi posso sommare  $\beta^n$  a tutti i membri e trovare subito l'espressione in funzione di  $A$  :  $\frac{\beta^n}{2} = -\frac{\beta^n}{2} + \beta^n \leq A = a + \beta^n < 0 + \beta^n = \beta^n$ .

Quindi la legge  $L^{-1}$  è:

$$L^{-1} : a = \begin{cases} A & \text{se } 0 \leq A < \frac{\beta^n}{2} \\ A - \beta^n & \text{se } \frac{\beta^n}{2} \leq A < \beta^n \end{cases}$$

**Esiste però una forma più semplice della legge inversa!**

#### Derivazione dalla precedente:

- Scrivo le condizioni degli intervalli come condizioni sulla cifra più significativa
- Sostituisco  $A - \beta^n$  aggiungendo con la definizione di complemento  $\bar{A} = \beta^n - 1 - A$ :  
Avrei  $a = A - \beta^n$ , quindi  $-a = \beta^n - A \implies -a - 1 = \beta^n - 1 - A = \bar{A}$ .  
Ho finito, perché  $-a - 1 = \bar{A} \Rightarrow -a = \bar{A} + 1 \Rightarrow a = -(\bar{A} + 1)$

$$L^{-1} : a = \begin{cases} A & \text{se } a_{n-1} < \frac{\beta}{2} \\ -(\bar{A} + 1) & \text{se } a_{n-1} \geq \frac{\beta}{2} \end{cases}$$

#### **Forma alternativa**

$$L : A = |a|_{\beta^n}$$

**ATTENZIONE! Questa formula vale SOLO SE**  $-\frac{\beta^n}{2} \leq a < \frac{\beta^n}{2}$  !!!

#### Derivazione:

- Se  $a$  è positivo allora  $0 \leq a < \frac{\beta^n}{2}$ , quindi  $A = a = |a|_{\beta^n}$ .
- Se  $a$  è negativo allora  $-\frac{\beta^n}{2} \leq a < 0$  quindi, per il Th. Div con Resto, se lo scrivo come quoziente e resto della divisione per  $\beta^n$  ottengo  $a = -\beta^n + |a|_{\beta^n}$ , quindi visto che  
 $A = a + \beta^n$ , si ha che  $A = (-\beta^n + |a|_{\beta^n}) + \beta^n = |a|_{\beta^n}$ .

## Circuiti logici per operazioni su $\mathbb{Z}$ interi

**ATTENZIONE! Abbiamo definito la legge  $L$  in questo modo perché VOGLIAMO LAVORARE SULLE RAPPRESENTAZIONI!**



**I circuiti vedono in ingresso CIFRE e in uscita CIFRE, non NUMERI!  
I NUMERI associati a quelle CIFRE CAMBIANO A SECONDA CHE LI INTEDA  
COME  
Naturali o  $\mathbb{Z}$ interi!**

## Valore Assoluto

**Su quante cifre sta il risultato?**

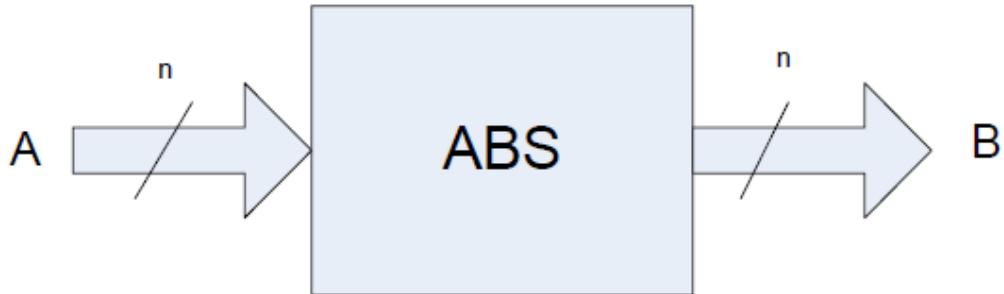
Visto che  $a \in \left[-\frac{\beta^n}{2}, \frac{\beta^n}{2} - 1\right]$ , allora  $B = ABS(a) \in \left[0, \frac{\beta^n}{2}\right]$ .

Quindi  $B$  è un numero Naturale rappresentabile su  $n$  cifre.

Perché  $n - 1$  non bastano?

- $\beta = 10, n = 3 \Rightarrow 10^3 / 2 = 500$  e servono  $n = 3$  cifre
- $\beta = 2, n = 3 \Rightarrow 2^3 / 2 = 2^{3-1} = 2^2 = 4 = (100)_{b2}$  e servono  $n = 3$  cifre!

**L'operazione è comunque sempre fattibile!**

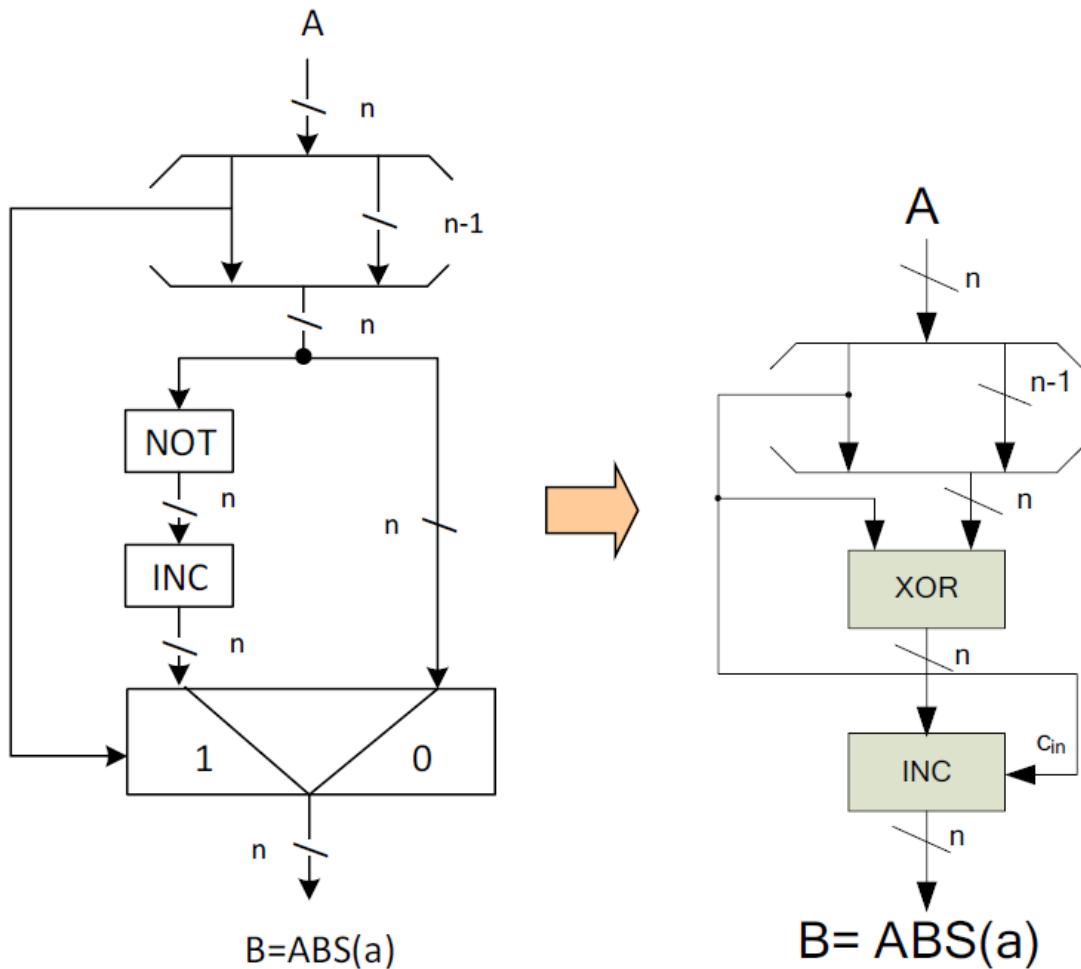


Visione funzionale di un circuito per il calcolo del valore assoluto

$$B = ABS(a) = \begin{cases} A & \text{se } a_{n-1} < \frac{\beta}{2} \\ A + 1 & \text{se } a_{n-1} \geq \frac{\beta}{2} \end{cases}$$

In base  $\beta = 2$  ovviamente le condizioni su  $a_{n-1}$  diventano  $a_{n-1} = 0$  oppure  $a_{n-1} = 1$

### Implementazione in base $\beta = 2$ :



Circuito per il calcolo del valore assoluto in base 2, non ottimizzato (sx), e ottimizzato (dx)

La variabile di comando del MUX è  $a_{n-1}$ .

Inoltre il circuito si può semplificare notando che:

- $A \oplus 0 = A$  perché  $0 \oplus 0 = 0$ ,  $1 \oplus 0 = 1$
- $A \oplus 1 = \overline{A}$  perché  $0 \oplus 1 = 1 = \overline{0}$ ,  $1 \oplus 1 = 0 = \overline{1}$

Quindi, visto che quelle  $n$  XOR eseguono  $a_i \oplus a_{n-1}$ ,

- Se  $a_{n-1} = 0$  allora  $a_i \oplus 0 = a_i$
- Se  $a_{n-1} = 1$  allora  $a_i \oplus 1 = \overline{a_i}$

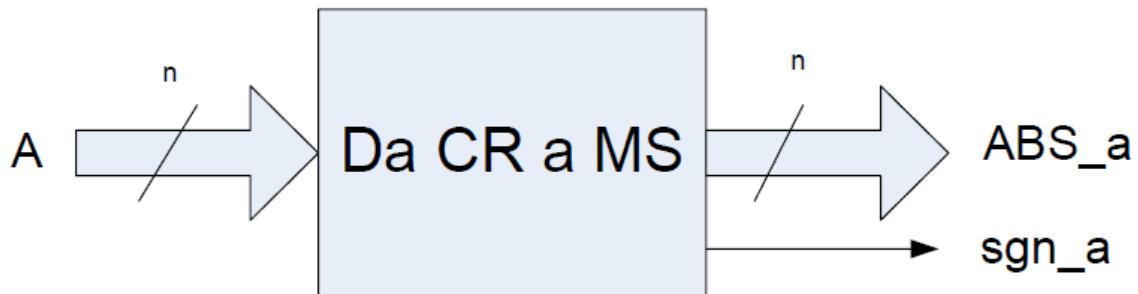
Infine, questo risultato viene dato ad un incrementatore che ha come  $C_{in}$  proprio  $a_{n-1}$ :

- Se  $a_{n-1} = 0$  allora l'incrementatore lascia invariata l'uscita dello XOR
- Se  $a_{n-1} = 1$  allora l'incrementatore ci somma 1

Ricapitolando:

- $a_{n-1} = 0 \implies a_i \oplus 0 = a_i \implies a_i + 0 = a_i \implies B = A$
- $a_{n-1} = 1 \implies a_i \oplus 1 = \overline{a_i} \implies \overline{a_i} + 1 = \overline{a_i} + 1 \implies B = \overline{A} + 1$

## Conversione $\sqrt{CR} \rightarrow (M, s)$



Visione funzionale del circuito per la conversione da Complemento alla Radice a Modulo e Segno in una base generica

**Questa operazione è sempre fattibile**, perché:

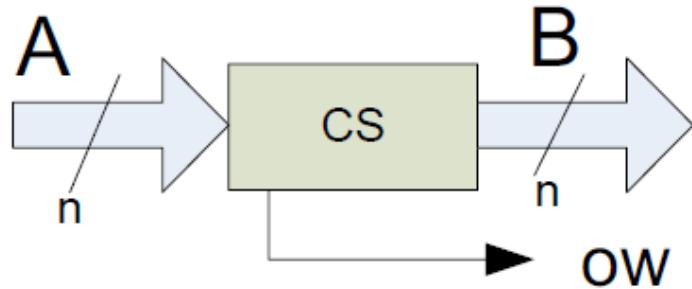
L'intervallo di rappresentabilità in CR è  $\left[-\frac{\beta^n}{2}, \frac{\beta^n}{2} - 1\right]$ , mentre quello in MS è  $\left[0, \frac{\beta^n}{2}\right]$  perché  $M = ABS(a)$ . Il secondo intervallo contiene il primo (in modulo).

## Opposto

**Questa operazione invece NON E' SEMPRE FATTIBILE (c'è solo un input per cui non è fattibile)!**

Questo perché, decidendo di rappresentare  $A$  e  $B$  sullo stesso numero  $n$  di cifre in  $\sqrt{CR}$ , si ha la rappresentazione di un negativo in più (intervallo di  $\sqrt{CR}$  asimmetrico), che non ha quindi il suo opposto!

Dunque dovrò dotare il circuito di un'uscita di *ow* (**OVERRLOW**), che deve essere settato se l'operazione non è fattibile, mentre a 0 se tutto va bene:



Visione funzionale del circuito per il calcolo dell'opposto in una base generica

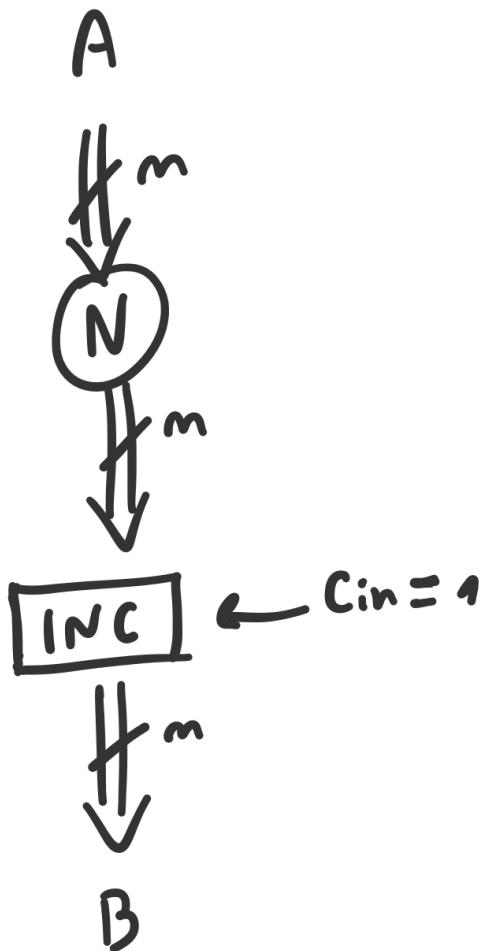
## Derivazione del criterio pratico

Intanto mettiamoci nell'ipotesi che tutto vada bene, quindi  $a \neq -\frac{\beta^n}{2}$ , allora:

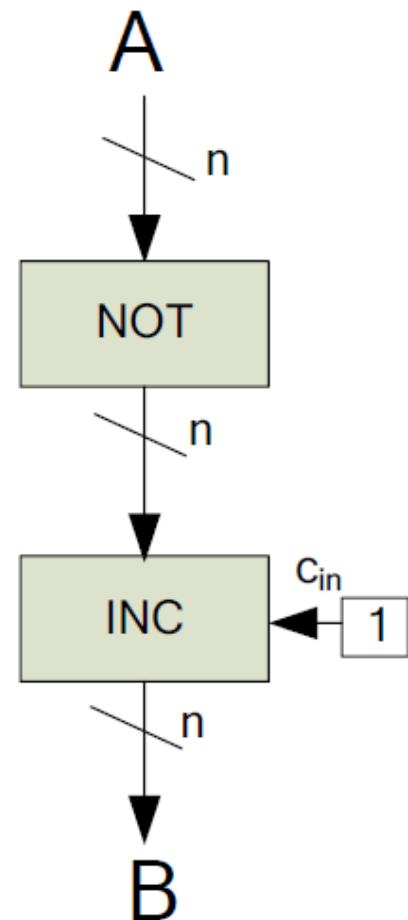
$$B = -A = |-a|_{\beta^n} = ||-1|_{\beta^n} \cdot |a|_{\beta^n}|_{\beta^n} = ||\beta^n - 1|_{\beta^n} \cdot |a|_{\beta^n}|_{\beta^n} = \\ |(\beta^n - 1) \cdot A|_{\beta^n} = |\beta^n \cdot A - A|_{\beta^n} = |-A|_{\beta^n}$$

Ora, usando la definizione di complemento  $\bar{A} = \beta^n - 1 - A \Rightarrow -A = -\beta^n + 1 + \bar{A}$  si ha:

$$|-A|_{\beta^n} = |-\beta^n + 1 + \bar{A}|_{\beta^n} = |\bar{A} + 1|_{\beta^n}$$



Circuito per il calcolo dell'opposto in base generica



Circuito per il calcolo dell'opposto in base 2

Il circuito  $N$  in realtà sono  $n$  circuiti  $N_i$ , ciascuno dei quali è un sottrattore che calcola il complemento  $\bar{a}_i = \beta - 1 - a_i$



**Si verifica overflow in base  $\beta = 2$  quando  $A = (1|0\ 0\dots 0\ 0)_{b2}$  con un qualsiasi numero di zeri!!**

La parte di circuito che sintetizza l'uscita  $ow$  è semplicemente una porta AND che esegue

$a_{n-1} \wedge b_{n-1}$ , perché se  $a_{n-1} = b_{n-1} = 1$  c'è un errore!

## Estensione di campo ( $\mathbb{Z}$ )

Quest'operazione è sempre fattibile!

Per i  $\mathbb{N}$ aturali si aveva che  $A^{EST} = (0|a_{n-1}a_{n-2}\dots a_1a_0) = 0 \cdot \beta^n + A$ .

Per gli  $\mathbb{Z}$ interi si ha che:

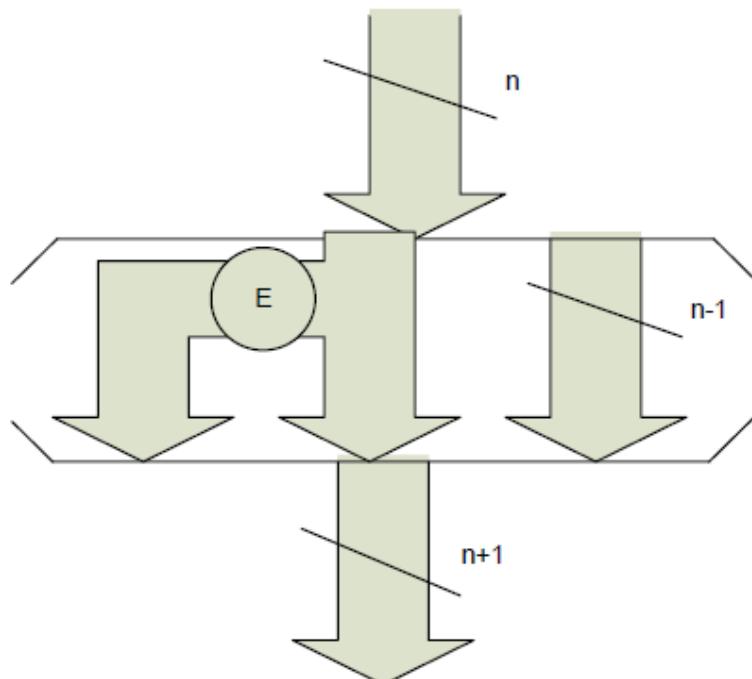
$$A^{EST} = \begin{cases} 0 & \cdot \beta^n + A \quad \text{se } a_{n-1} < \frac{\beta}{2} \\ (\beta - 1) & \cdot \beta^n + A \quad \text{se } a_{n-1} \geq \frac{\beta}{2} \end{cases}$$

Ovvero:

$$A^{EST} = \begin{cases} (0 \mid a_{n-1} a_{n-2} \dots a_1 a_0) & \text{se } a_{n-1} < \frac{\beta}{2} \\ ((\beta - 1) \mid a_{n-1} a_{n-2} \dots a_1 a_0) & \text{se } a_{n-1} \geq \frac{\beta}{2} \end{cases}$$

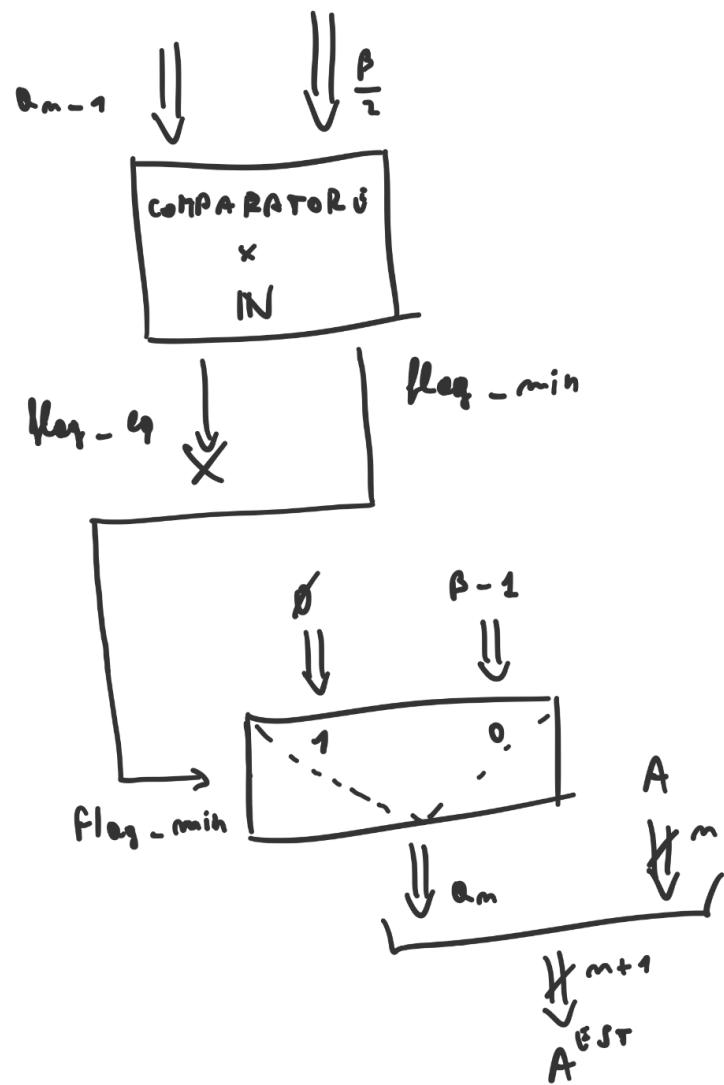
**Entrambe sono operazioni di concatenamento, quindi non richiedono logica!**

L'unica logica richiesta è quella per discriminare il segno del numero!



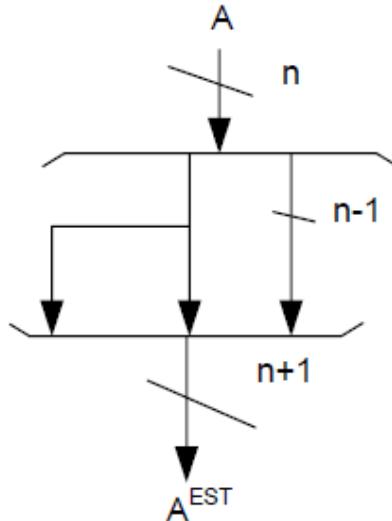
Visione funzionale del circuito per l'estensione di numeri interi in una base generica

Dove il circuito  $E$  (assieme alla concatenazione finale) sarà il seguente:



Circuito per l'estensione di campo di interi in una base generica (errato)

In base  $\beta = 2$  si semplifica tutto:  $A^{EST} = a_{n-1} \cdot \beta^n + A$ :



Circuito per l'Estensione di campo di interi in base 2

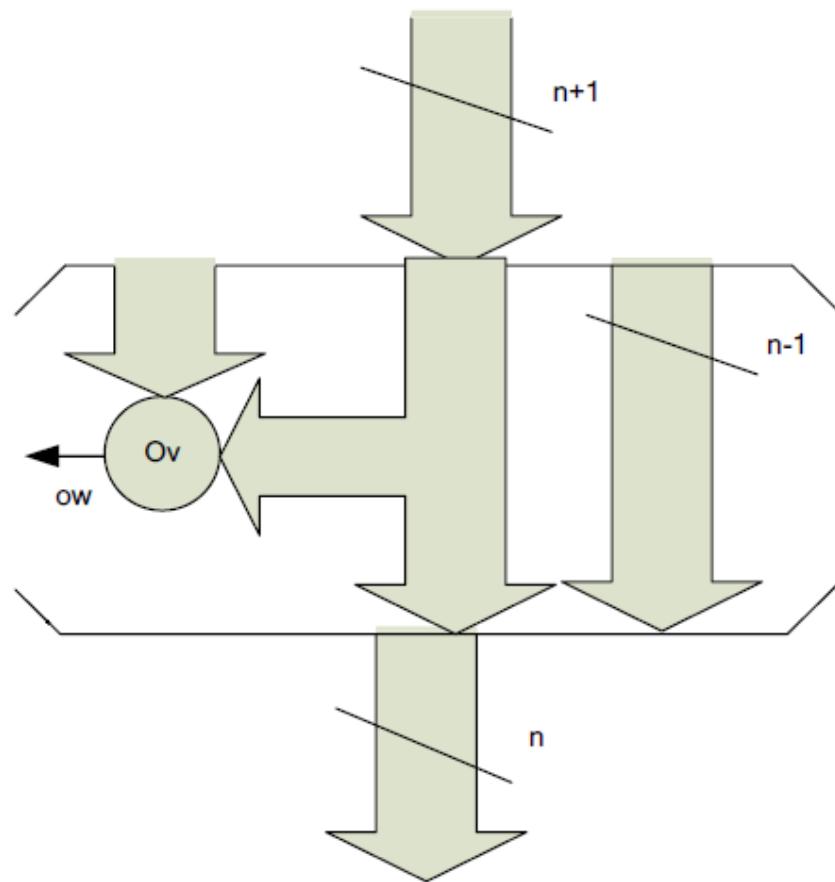
## Riduzione di campo

Ovviamente stavolta **l'operazione NON è sempre fattibile**. Infatti, perché sia fattibile c'è bisogno che  $a \in \left[ -\frac{\beta^n}{2}, \frac{\beta^n}{2} - 1 \right] \subset \left[ -\frac{\beta^{n+1}}{2}, \frac{\beta^{n+1}}{2} - 1 \right]$ .

Si deduce la **CONDIZIONE DI RIDUCIBILITÀ** dalle considerazioni fatte prima per l'estensione di campo:

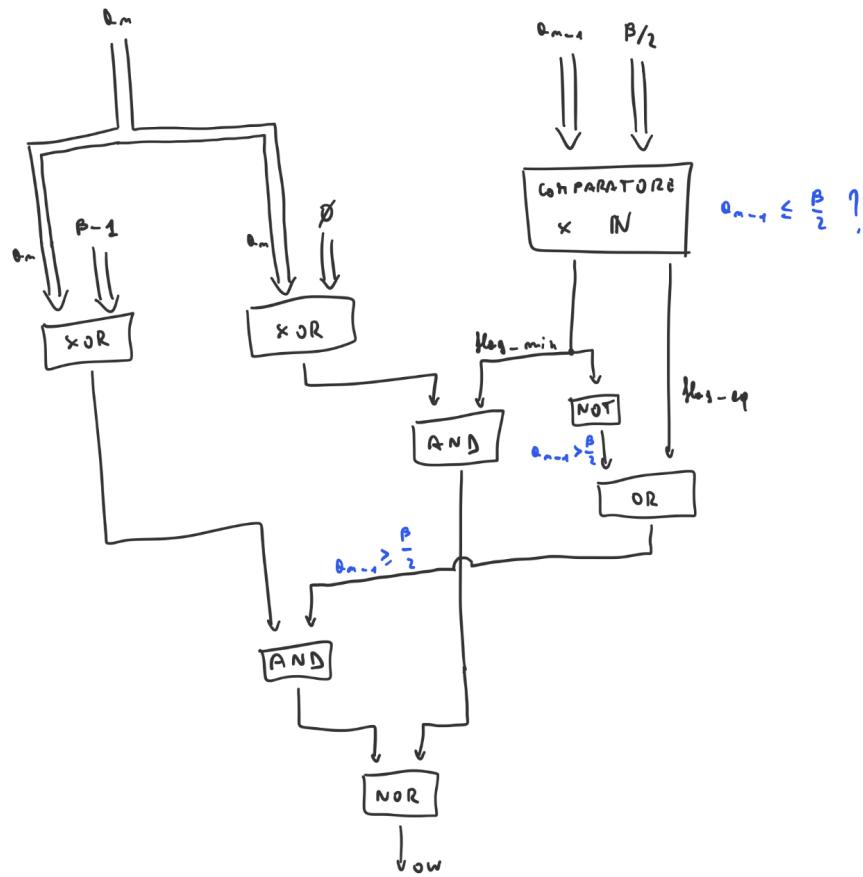
$$ow = 0 \iff \left( a_n = 0 \wedge a_{n-1} < \frac{\beta}{2} \right) \vee \left( a_n = (\beta - 1) \wedge a_{n-1} \geq \frac{\beta}{2} \right)$$

Nel caso in cui il numero sia riducibile,  $ow = 0$ , quindi la sua rappresentazione su  $n$  cifre è data dalle  $n$  cifre meno significative del numero di partenza da ridurre:



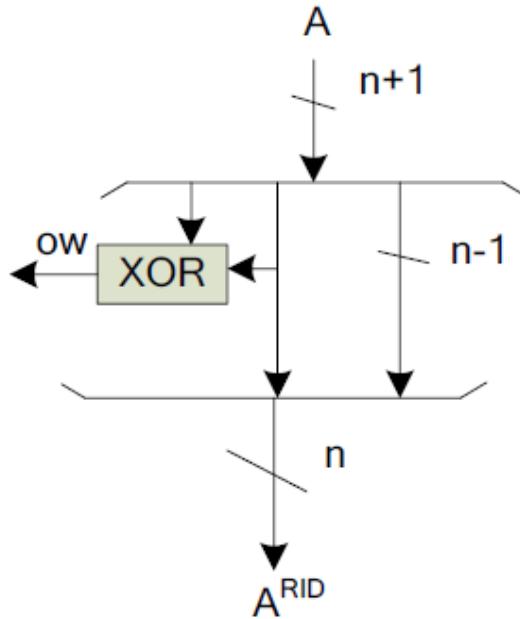
Visione funzionale del circuito per la riduzione di campo di interi in una base generica

Il circuito OV che ha in ingresso  $a_n$  ed  $a_{n-1}$ , e che sintetizza l'uscita di  $ow$  è il seguente:



Circuito OV per la sintesi dell'uscita ow in base generica

Come per l'estensione, anche per la riduzione le cose in base  $\beta = 2$  si semplificano:



Circuito per la riduzione di campo in base 2

In base  $\beta = 2$  il circuito OV per la sintesi dell'uscita *ow* "collassa" in uno XOR!



**CONDIZIONE DI RIDUCIBILITÀ IN BASE  $\beta = 2$ : Le due cifre più significative sono uguali!**

**Come si riduce di  $k$  cifre  $A = (a_{n+k} \dots a_n \dots a_0)_2$  un numero in base  $\beta = 2$ ?**

- Usando  $k$  XOR, fra le cifre  $a_{n+i}$  e  $a_{n+i+1}$ , con  $i = 0 \dots k - 1$ , e poi mandare tutto in ingresso ad una porta OR a  $k$  ingressi, generando così l'uscita *ow*
- Usando una AND ed una OR a  $k + 1$  ingressi, dandogli in ingresso le cifre  $a_{n+k} \dots a_n$  (che sono  $n + 1$ ), e mandando le loro uscite in ingresso ad una XOR.

Spiegazione:

$x$	$y$	$x \wedge y$	$x \vee y$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Dalla tabella di verità si vede che le due porte AND e OR danno un risultato uguale se e solo se entrambi i loro input sono uguali (i casi  $(0, 0)$  e  $(1, 1)$ ), quindi se tutti i

bit sono uguali le due porte restituiscono lo stesso valore, e lo XOR fa 0, infatti torna. Negli altri casi fa 1.

## Moltiplicazione e divisione per una potenza della base

### Moltiplicazione

**La moltiplicazione per potenza della base  $b = \beta \cdot a$  è un'operazione sempre fattibile:**

$$L : B = \begin{cases} b = \beta \cdot a & \text{se } 0 \leq a < \frac{\beta^n}{2} \\ \beta \cdot (a + \beta^n) = \beta \cdot a + \beta^{n+1} = b + \beta^{n+1} & \text{se } -\frac{\beta^n}{2} \leq a < 0 \end{cases}$$

### Divisione

Per la divisione  $b = \left\lfloor \frac{a}{\beta} \right\rfloor$  scrivo  $a$  come quoziente e resto della divisione per  $\beta^{n+1}$ :

$$\begin{aligned} B &= \left\lfloor \left\lfloor \frac{a}{\beta} \right\rfloor \right\rfloor_{\beta^n} = \left\lfloor \left\lfloor \frac{\left\lfloor \frac{a}{\beta^{n+1}} \right\rfloor \cdot \beta^{n+1} + |a|_{\beta^{n+1}}}{\beta} \right\rfloor \right\rfloor_{\beta^n} = \\ &\left\lfloor \left\lfloor \frac{a}{\beta^{n+1}} \right\rfloor \cdot \beta^n + \frac{|a|_{\beta^{n+1}}}{\beta} \right\rfloor_{\beta^n} = \left\lfloor \left\lfloor \frac{A}{\beta} \right\rfloor \right\rfloor_{\beta^n} = \left\lfloor \frac{A}{\beta} \right\rfloor \end{aligned}$$

**Quindi sia la moltiplicazione per  $\beta$  che la divisione si fanno nello stesso modo che per i Naturali!**

### Somma

**L'operazione di somma NON è sempre fattibile!** Vediamo perché:

Ho due numeri  $a$  e  $b$  rappresentati in  $\sqrt{CR}$  da  $A$  e  $B$ .

Visto che sono  $\mathbb{Z}$  interi,  $a, b \in \left[-\frac{\beta^n}{2}, \frac{\beta^n}{2} - 1\right]$ , quindi la loro somma  $s \in [-\beta^n, \beta^n - 2]$ .

Quindi, la somma  $s$  può non essere rappresentabile su  $n$  cifre, ma DI SICURO lo è su  $n + 1$ , perché  $\frac{\beta^{n+1}}{2} \geq \beta^n$ . Quindi il circuito sarà dotato di un'uscita di  $ow$ .

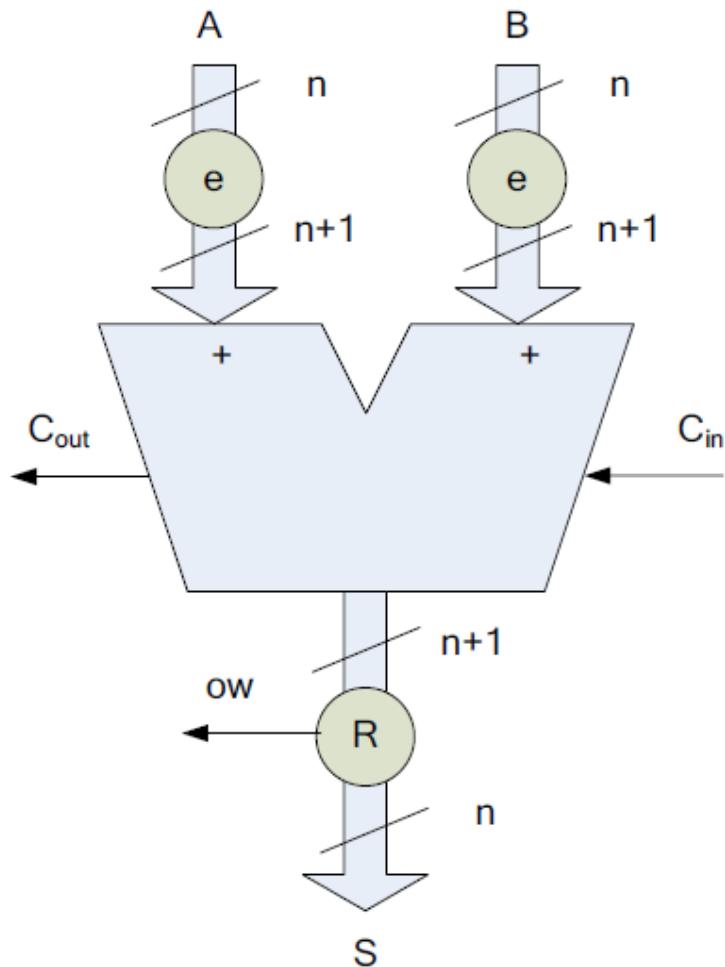
Intanto mettiamoci nell'ipotesi che  $s$  stia su  $n$  cifre, si ha:

$$S = |s|_{\beta^n} = |a + b|_{\beta^n} = ||a|_{\beta^n} + |b|_{\beta^n}|_{\beta^n} = |A + B|_{\beta^n}$$

**Cioè la rappresentazione  $S$  della somma  $a + b$  è la somma delle rappresentazioni  $|A + B|_{\beta^n}$ !**

Proprio per questa proprietà si è scelto di usare la rappresentazione in  $\sqrt{CR}$ !

Perché poi, visto che il sommatore è alla base anche delle altre operazioni, consente di riutilizzare molta circuiteria!



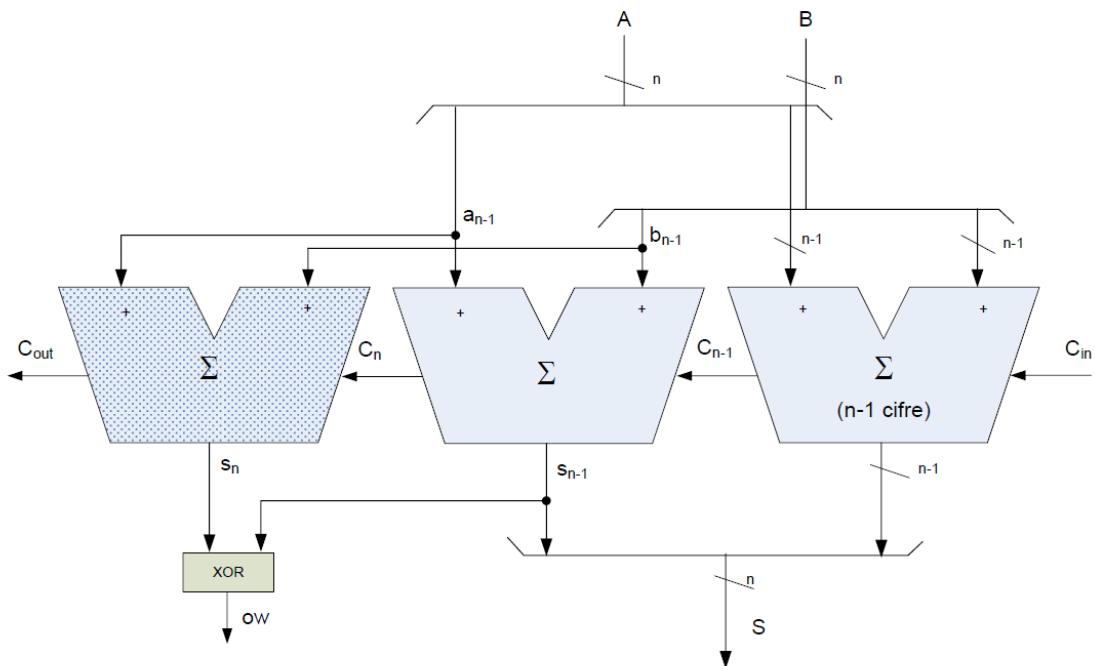
Visione funzionale del sommatore di interi in una base generica

I circuiti  $e$  in figura sono circuiti di Estensione di campo per numeri  $\mathbb{Z}$  interi, mentre  $R$  è il circuito di Riduzione di campo per  $\mathbb{Z}$  interi!

### Criterio pratico:

In sostanza, visto che come detto all'inizio,  $S$  potrebbe non essere rappresentabile su  $n$  cifre, ma su  $n + 1$  di sicuro si, si fa così:

- Si estendono gli operandi su  $n + 1$  cifre, così la somma è sempre rappresentabile
- Si usa un sommatore per Naturali su  $n + 1$  cifre
- Si riduce il risultato su  $n$  cifre, e se l'operazione non è fattibile si genera un *ow*

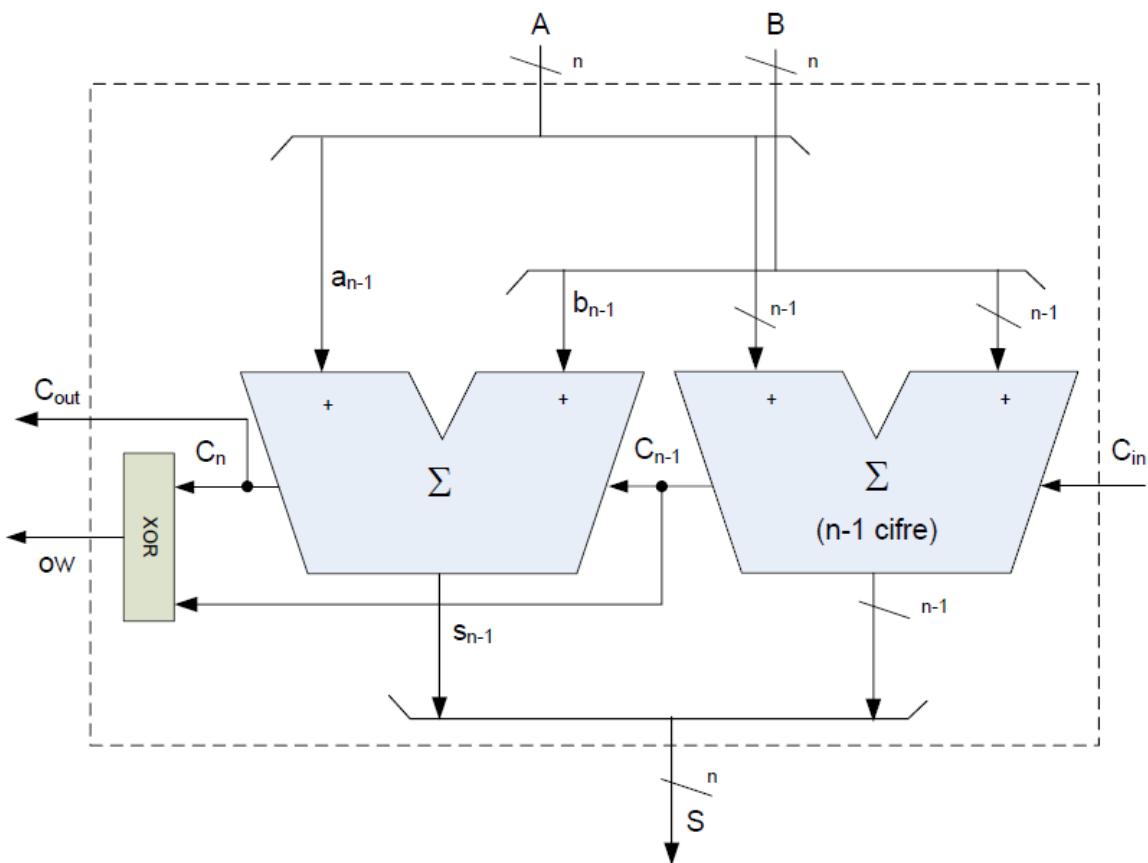


Montaggio a ripple carry di sommatori di interi in base 2

Tuttavia questo circuito si può semplificare togliendo l'ultimo Full Adder (appena introdotto perché si esegue una somma su  $n + 1$  cifre), giocando con le uscite degli ultimi due Full Adder:

$$\begin{aligned} \text{ow} &= s_n \oplus s_{n-1} = (a_{n-1} \oplus b_{n-1} \oplus C_n) \oplus (a_{n-1} \oplus b_{n-1} \oplus C_{n-1}) = \\ &= 0 \oplus C_n \oplus C_{n-1} = C_n \oplus C_{n-1} \end{aligned}$$

Perché sappiamo che  $a \oplus a = 0$ .



Montaggio a ripple carry di sommatori in base 2 ottimizzato

Quindi,



**I SOMMATORI SOMMANO SIA  $\mathbb{N}$  NATURALI CHE  $\mathbb{Z}$  INTERI!**

Per vedere se le uscite sono rappresentabili:

- $\mathbb{N} \rightarrow C_{out}$
- $\mathbb{Z} \rightarrow ow$

## Sottrazione

Molto molto simile alla somma!

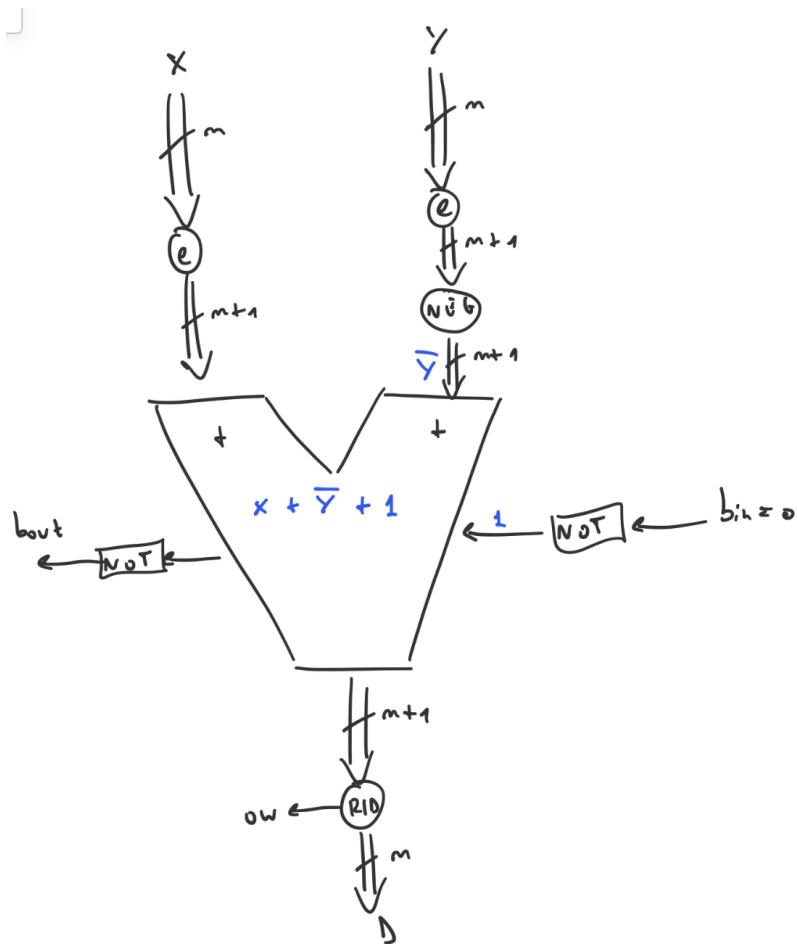
Anche per la sottrazione il risultato non è sempre rappresentabile su  $n$  cifre, però su  $n + 1$  di sicuro si.

Intanto mettiamoci nell'ipotesi in cui  $d = a - b$  sia rappresentabile su  $n$  cifre:

$$D = |\textcolor{red}{d}|_{\beta^n} = |a - b|_{\beta^n} = ||a|_{\beta^n} - |b|_{\beta^n}|_{\beta^n} = |A - B|_{\beta^n} = \\ = |\textcolor{red}{A + \overline{B} + 1}|_{\beta^n}$$

In rosso ho evidenziato le differenze con la formula per la somma, sono minime.  
Spunta fuori  $\overline{B}$  perché  $\overline{B} = \beta^n - 1 - B \implies -B = \overline{B} + 1 - \beta^n$ , e ovviamente il  $\beta^n$  lo posso cancellare essendo dentro un modulo.

Per quanto riguarda i circuiti, sono UGUALI a quelli appena sopra per la somma, solo che al posto dei sommatori ci sono sottrattori, quindi sommatori con l'ingresso  $B$  negato e  $b_{in/out}$  con un inverter al posto di  $C_{in/out}$ :





Come non esistono "sommatori per  $\mathbb{N}$ " e "sommatori per  $\mathbb{Z}$ ", non esistono nemmeno "sottrattori per  $\mathbb{N}$ " o "sottrattori per  $\mathbb{Z}$ "!

## Circuito comparatore ( $\mathbb{Z}$ )

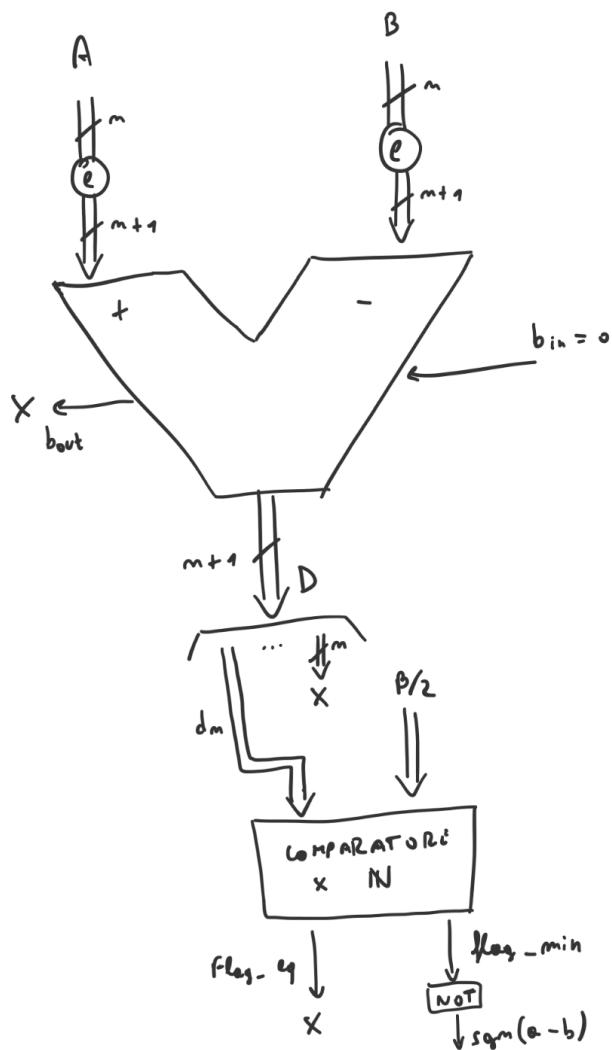
Per la parte riguardante l'uscita  $flag\_eq$  si fa nello stesso modo:  $n$  XOR, le quali vanno in ingresso ad una NOR ad  $n$  ingressi.

Per l'uscita  $flag\_min$  invece **NON VA GUARDATO IL PRESTITO USCENTE  $b_{out}$  MA IL SEGNO DELLA DIFFERENZA!**

**Spiegazione:**

La rappresentazione in  $\sqrt{CR}$  non è monotona, quindi  $A < B$  NON IMPLICA  $a < b$ !  
Quindi devo trovare un altro modo, che è sottrarre  $A$  e  $B$  con un sottrattore, e guardare il segno della differenza.

Attenzione che, perché il segno della differenza sia corretto, la differenza  $A - B$  deve essere corretta, quindi devo PER FORZA estendere  $A$  e  $B$  su  $n + 1$  cifre!  
(tra l'altro dovrei comunque estendere  $A$  e  $B$  perché in un sottrattore intero, che è un sommatore con qualche ritocchino di fatto, devo comunque estendere gli operandi)

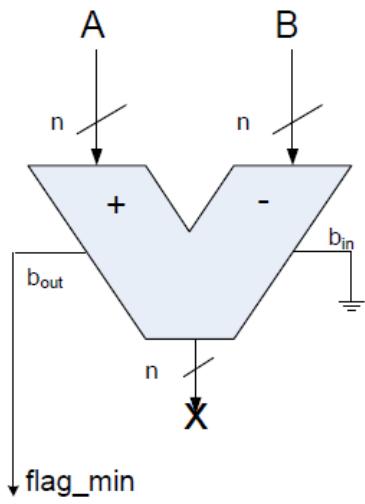


Parte del circuito comparatore per interi in una base generica che genera  $sgn(a - b) = \overline{flag\_min}$

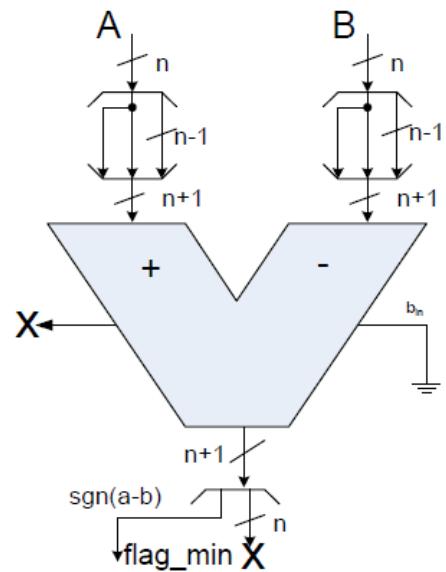
Nella figura i circuiti  $e$  sono circuiti di estensione di campo per  $\mathbb{Z}$  interi su  $n + 1$  cifre

In base  $\beta = 2$  ovviamente si semplifica:

Comparatore  
per numeri naturali



Comparatore  
per numeri interi

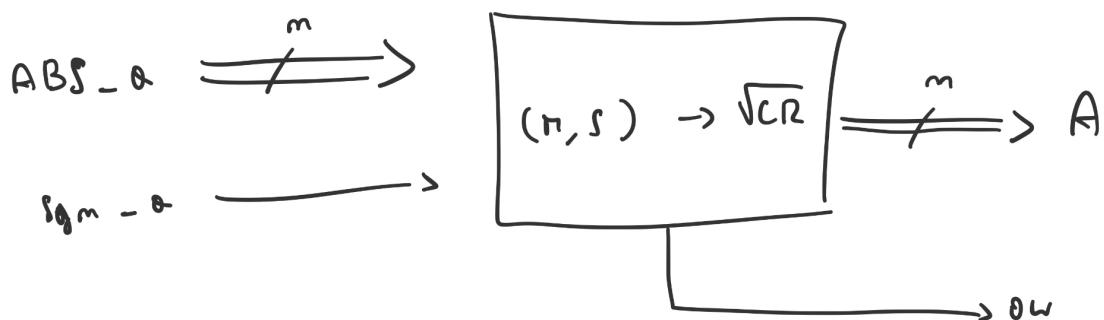


Non c'è bisogno di un circuito di riduzione di campo! Basta un sottrattore ad  $n + 1$  cifre!

Conversione  $(M, s) \rightarrow \sqrt{CR}$

Contrariamente alla sua controparte, questa operazione **NON** è sempre fattibile!

Questo perché l'intervallo di rappresentabilità in  $(M, s)$  è  $[-(\beta^n - 1), +(\beta^n - 1)]$ , mentre quello per  $\sqrt{CR}$  è  $\left[-\frac{\beta^n}{2}, \frac{\beta^n}{2} - 1\right]$ . Ci deve essere quindi un'uscita di *ow*:



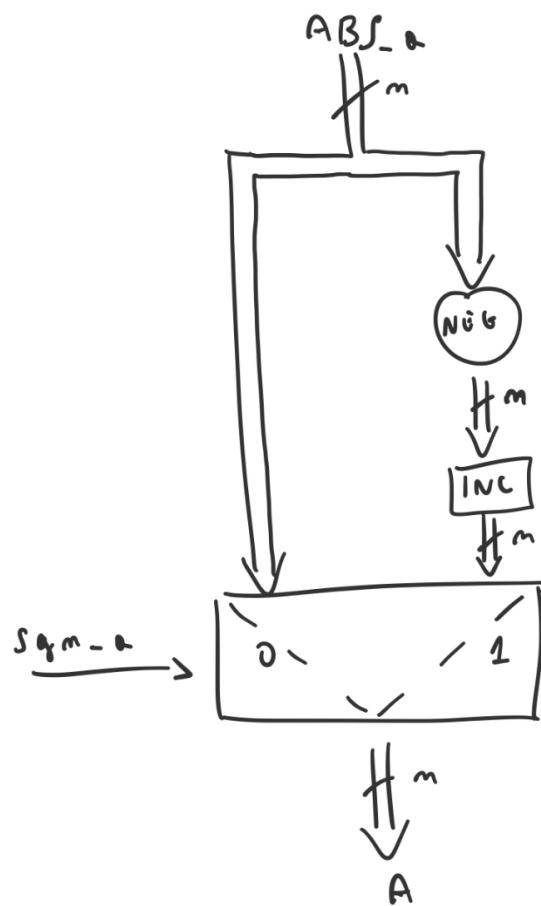
Visione funzionale del circuito di conversione da MS a CR

Intanto mettiamoci nell'ipotesi in cui sia fattibile:

$$A = |a|_{\beta^n} = \begin{cases} |ABS\_a|_{\beta^n} & \text{se } sgn\_a = 0 \\ |-ABS\_a|_{\beta^n} & \text{se } sgn\_a = 1 \end{cases} =$$

$$= \begin{cases} ABS\_a & \text{se } sgn\_a = 0 \\ |\overline{ABS\_a} + 1|_{\beta^n} & \text{se } sgn\_a = 1 \end{cases}$$

La parte di circuito che sintetizza  $A$  è molto simile ad un circuito per il calcolo del valore assoluto:



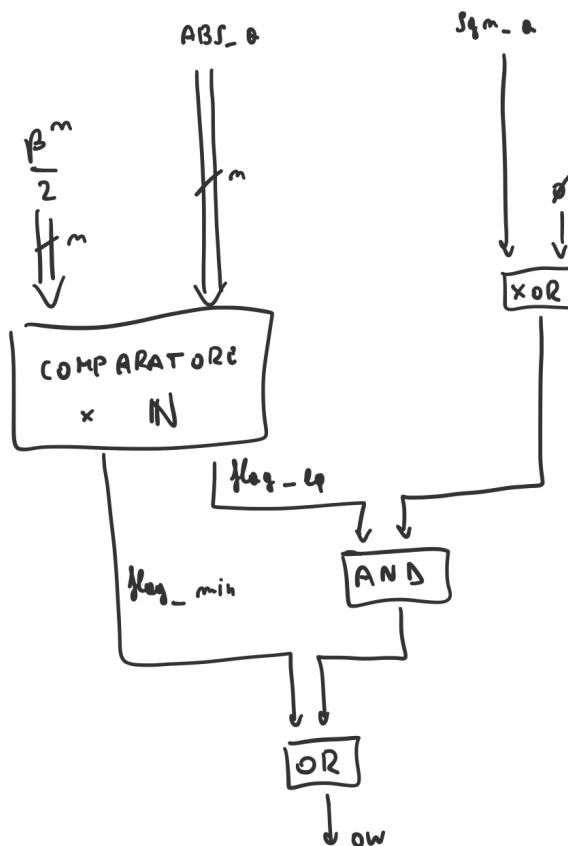
Poi, per l'uscita di *ow* si ha che:

$$ow = 1 \iff (ABS\_a > \frac{\beta^n}{2}) \vee (ABS\_a = \frac{\beta^n}{2} \wedge sgn\_a = 0)$$

### Spiegazione:

- Se  $ABS\_a > \frac{\beta^n}{2}$  allora sei fuori dall'intervallo di rappresentabilità in  $\sqrt{CR}$ , quindi qualsiasi numero fuori da quell'intervallo non può essere convertito perché ha modulo troppo grande.
- Se  $ABS\_a = \frac{\beta^n}{2} \wedge sgn\_a = 0$  significa che in ingresso c'è il valore che per l'asimmetria dell'intervallo in  $\sqrt{CR}$  viene tagliato fuori (se infatti fosse  $sgn\_a = 1$  andrebbe bene, perché l'intervallo include un negativo in più, per convenzione).

La parte di circuito che sintetizza  $ow$  sarà questa:



## Moltiplicazione

**Su quante cifre sta il risultato?**

Dati:

- $A$  RAPPRESENTAZIONE DI  $a \in \mathbb{Z}$  su  $n$  cifre
- $B$  RAPPRESENTAZIONE DI  $b \in \mathbb{Z}$  su  $m$  cifre

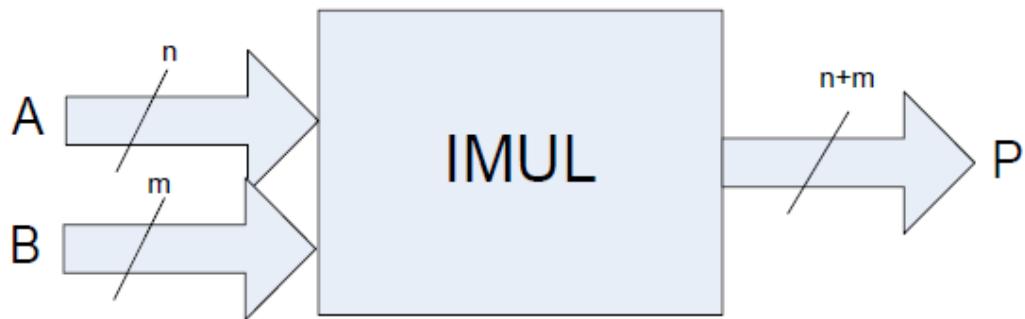
Voglio calcolare  $p = a \cdot b$  su  $n + m$  cifre, quindi, visto che:

- $a \in \left[ -\frac{\beta^n}{2}, \frac{\beta^n}{2} - 1 \right]$ ,
- $b \in \left[ -\frac{\beta^m}{2}, \frac{\beta^m}{2} - 1 \right]$ ,

allora

$$p \in \left[ -\frac{\beta^n}{2} \cdot \left( \frac{\beta^m}{2} - 1 \right), -\frac{\beta^n}{2} \cdot \left( -\frac{\beta^m}{2} \right) \right] \approx \left[ -\frac{\beta^{n+m}}{4}, \frac{\beta^{n+m}}{4} \right]$$

quindi il risultato sta su  $n + m$  cifre e **l'operazione è sempre fattibile!**



Visione funzionale di un circuito moltiplicatore per interi in una base generica

Sia per moltiplicazione che divisione intere, conviene spezzare il modulo dal segno, e trovare espressioni per ciascuno separatamente:

Posso sfruttare la rappresentazione "in modulo e segno" per rappresentare qualsiasi numero  $x$ :  $x = \text{sgn}(x) \cdot \text{ABS}(x)$

**Modulo:**

$$ABS(p) = ABS(a) \cdot ABS(b)$$

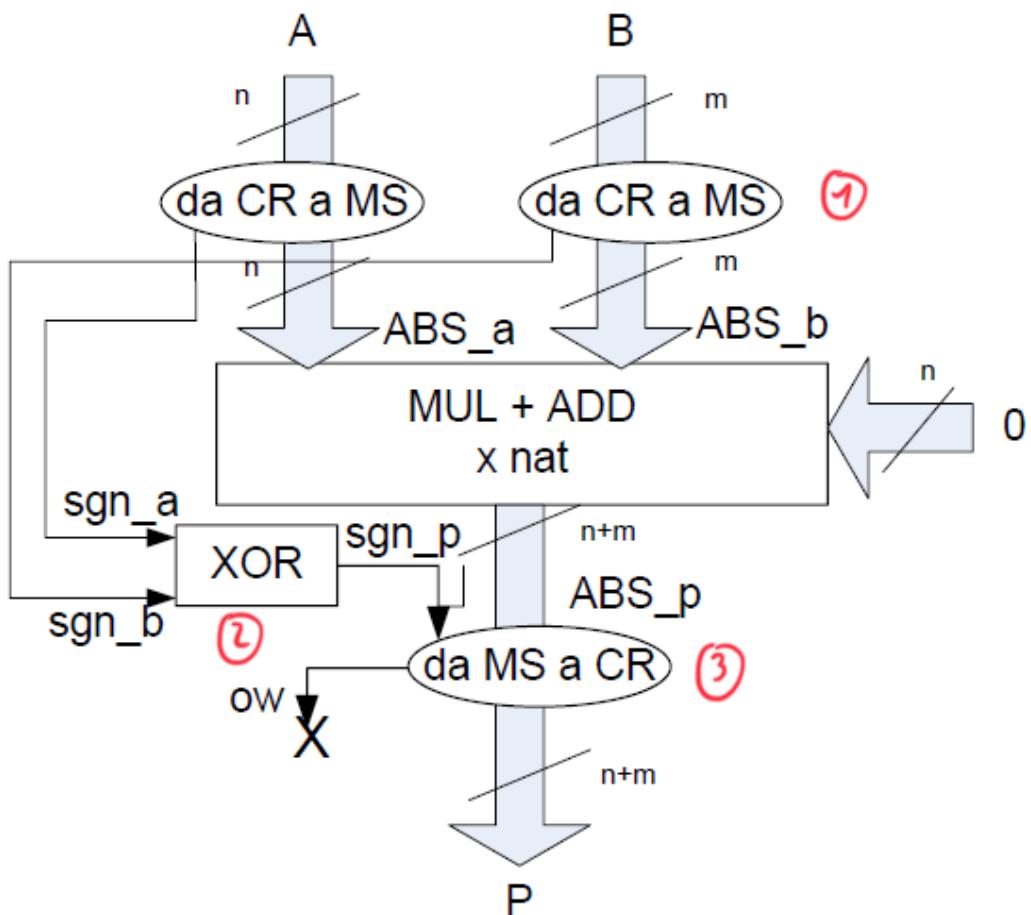
Segno:

$$sgn(p) = sgn(a) \cdot sgn(b)$$

Quindi:

$$\begin{cases} ABS(p) = ABS(a) \cdot ABS(b) \\ sgn(p) = sgn(a) \cdot sgn(b) \end{cases}$$

Dove  $ABS(a) \cdot ABS(b)$  è una moltiplicazione fra Naturali, che so già fare!



Circuito per la moltiplicazione di interi in una base generica

1. La conversione  $\sqrt{CR} \rightarrow (M, s)$  è sempre fattibile, quindi non ci vuole l'uscita di *ow*
2. Il segno del prodotto è:
  - a. 0 se i segni di  $a$  e  $b$  sono concordi ( $a \oplus a = 0$ )
  - b. 1 se i segni di  $a$  e  $b$  sono discordi ( $a \oplus \bar{a} = 1$ )
3. La conversione  $(M, s) \rightarrow \sqrt{CR}$  non è sempre fattibile, quindi ci vuole l'uscita di *ow!*

Però dal conto iniziale si vede che  $p \in \approx \left[ -\frac{\beta^{n+m}}{4}, \frac{\beta^{n+m}}{4} \right] \subset \left[ -\frac{\beta^{n+m}}{2}, \frac{\beta^{n+m}}{2} - 1 \right]$

quindi è rappresentabile su  $n + m$  cifre (sempre in  $\sqrt{CR}$ ), quindi qui l'uscita di *ow* è inutile!



#### Nella moltiplicazione per $\mathbb{Z}$ interi:

- **Non c'è bisogno del flag di *ow*!**
- **Non c'è l'ingresso  $C$  che invece c'era per quello per i  $\mathbb{N}$ aturali!**  
**(all'interno del circuito c'è  $C$ , semplicemente non viene contato come ingresso!)**

## Divisione

### Su quante cifre sta il risultato?

Dati:

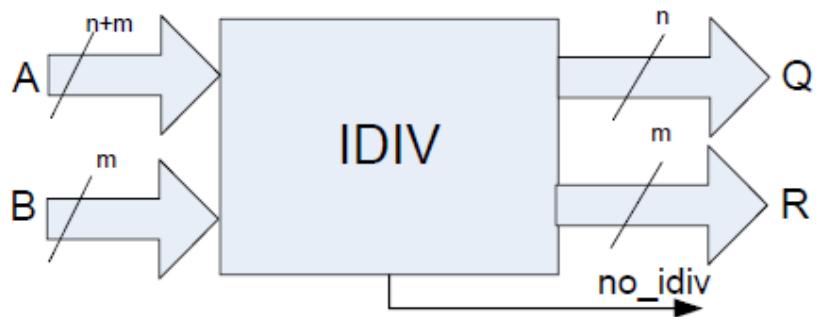
- **$A$  RAPPRESENTAZIONE DEL DIVIDENDO**  $a \in \mathbb{Z}$  su  $n + m$  cifre
- **$B$  RAPPRESENTAZIONE DEL DIVISORE**  $b \in \mathbb{Z}$  su  $m$  cifre

Voglio calcolare:

- **$Q$  RAPPRESENTAZIONE DEL QUOZIENTE**  $q \in \mathbb{Z}$  su  $n$  cifre
- **$R$  RAPPRESENTAZIONE DEL RESTO**  $r \in \mathbb{Z}$  su  $m$  cifre

tali che  $a = q \cdot b + r$

Ovviamente questa operazione non è sempre fattibile, perché  $q$  potrebbe non esistere oppure non essere rappresentabile su  $n$  cifre. Ci vuole quindi un'uscita di *ow!*



Visione funzionale del circuito di divisione per interi in una base generica

Finora ci siamo serviti del **Th. della Divisione con Resto**, che ci garantiva l'**UNICITA'** del risultato della divisione se  $r \in [0, b - 1]$ . Però questo andava bene finché  $b \in \mathbb{N}$ , ma quando diventa  $\mathbb{Z}$ , questo vincolo non va più bene perché  $b - 1$  sarebbe negativo.

### Criterio per garantire l'unicità del risultato

Allora bisogna trovare un vincolo che garantisca l'unicità del risultato e sia più generale del precedente, in modo da inglobarlo, ma considerando pure il caso  $\mathbb{Z}$  intero:

$$\begin{cases} ABS(r) < ABS(b) \\ sgn(r) = sgn(a) \end{cases}$$

La prima condizione è ovvia, perché generalizza banalmente quella  $r \in [0, b - 1]$ .

La seconda condizione invece è necessaria, perché, se il resto potesse essere sia negativo che positivo, potrei sempre trovare un risultato che abbia  $r > 0$  e un altro invece con  $r < 0$ , violando l'unicità.

**Allora si prende come CONVENZIONE che il resto abbia il segno del dividendo!**



### ATTENZIONE!

Mentre tutte le altre divisioni (SAR) APPROSSIMANO SEMPRE VERSO  $-\infty$ ,

**LA IDIV APPROXIMA SEMPRE PER VERSO LO 0 (PER TRONCAMENTO)**

## Derivazione del criterio pratico

Vediamo come ricavare una formula che ci permetta di usare la circuiteria per i  $\mathbb{N}$ aturali:

$$a = q \cdot b + r \implies \text{sgn}(a) \cdot ABS(a) = q \cdot (\text{sgn}(b) \cdot ABS(b)) + \text{sgn}(r) \cdot ABS(r)$$

Ora, moltiplicando per  $\text{sgn}(a)$ , e notando che:

- $\text{sgn}(a)^2 > 0 \rightarrow \text{sgn}(a)^2 = 1$
- $\text{sgn}(a) = \text{sgn}(r)$  per le condizioni scritte sopra, si ha:

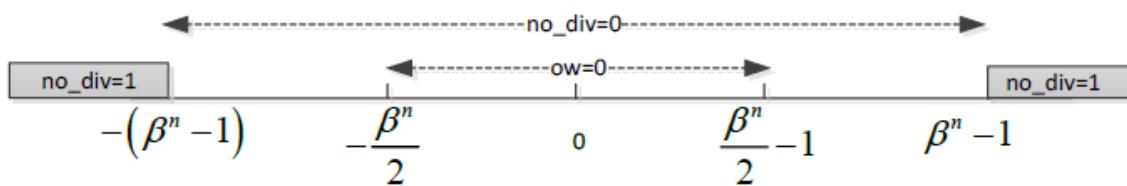
$$ABS(a) = q \cdot \text{sgn}(b) \cdot \text{sgn}(a) \cdot ABS(b) + ABS(r)$$

Ora, chiamo  $ABS(q) = q \cdot \text{sgn}(b) \cdot \text{sgn}(a)$  e ottengo:

$$ABS(a) = ABS(q) \cdot ABS(b) + ABS(r)$$

A questo punto mi sono ridotto ad un'espressione fra  $\mathbb{N}$ aturali, e posso svolgere questa divisione usando un divisore per  $\mathbb{N}$ aturali, purché però  $ABS(q)$  sia rappresentabile su  $n$  cifre, esattamente come ho imposto nella divisione fra  $\mathbb{N}$ aturali:

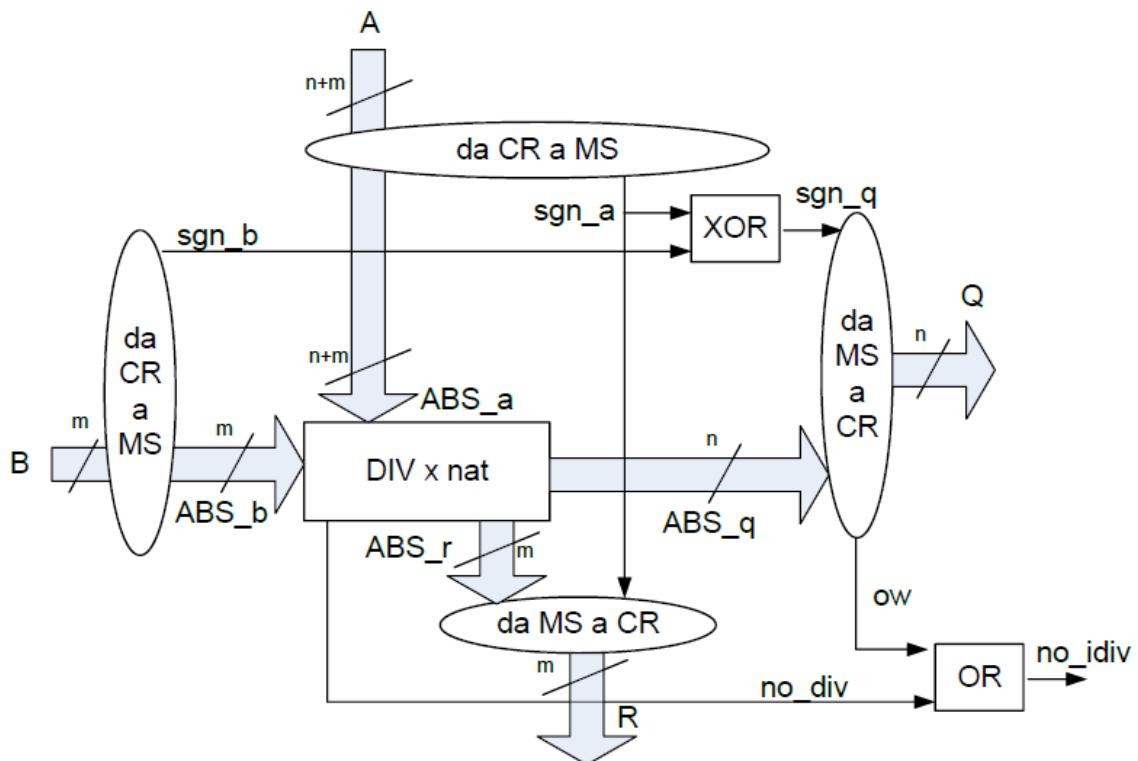
$$ABS(a) < \beta^n \cdot ABS(b) \quad (\iff X < \beta^n \cdot Y)$$



**La condizione  $no\_div : ABS(a) < \beta^n \cdot ABS(b)$  di qui sopra però è solo una CONDIZIONE NECESSARIA, ma non sufficiente per la fattibilità della divisione intera!**

Perché la divisione  $\mathbb{Z}$  intera sia fattibile ci vuole **PURE** che  $q$  sia un intero rappresentabile su  $n$  cifre!

$$q \in \left[ -\frac{\beta^n}{2}, \frac{\beta^n}{2} - 1 \right]$$



Circuito divisore per interi in una base generica

Quindi **prima viene effettuata la divisione fra Naturali  $ABS(a)/ABS(b)$** , si genera il flag  **$no\_div$**  che dice se il risultato di questa divisione è corretto, **e poi viene generato il flag  $ow$  che dice se il quoziente  $ABS(q)$  può essere o meno rappresentato su  $n$  cifre in base  $\beta$ .**

Se **ENTRAMBI** questi flag valgono 0, allora la divisione fra interi è corretta e il circuito restituisce i valori di  $Q$  ed  $R$  corretti.

Altrimenti, se anche UNO dei due flag vale 1, c'è stato un errore e quindi la divisione intera non era fattibile.

**Perché il circuito di conversione  $\sqrt{CR} \rightarrow (M, s)$  del resto  $R$  non ha il flag di  $ow$ ?**

Perché nella divisione  $R \leq B$  (resto  $\leq$  divisore), e visto che sia  $R$  che  $B$  stanno su  $m$

cifre in base  $\beta$ , se come input avevo  $B$ , rappresentato in  $\sqrt{CR}$  su  $m$  cifre in base  $\beta$ , allora pure  $R$  sarà per forza rappresentabile in  $\sqrt{CR}$  su  $m$  cifre in base  $\beta$ !

Recap sulla fattibilità della divisione intera:

- $no\_div = 0 \iff ABS(a) < \beta^n \cdot ABS(b)$
- $ow = 1 \iff \left(ABS(q) > \frac{\beta^n}{2}\right) \vee \left(ABS(q) = \frac{\beta^n}{2} \wedge sgn(q) = 0\right)$

Dove la prima condizione è l'analogo  $\mathbb{Z}$ intero della condizione sulla divisione Naturale, mentre la seconda condizione è quella dell'uscita  $ow$  di un circuito di conversione  $(M, s) \rightarrow \sqrt{CR}$ !

# Reti Sequenziali Asincrone

- Reti Combinatorie → senza memoria
- Reti Sequenziali → con memoria

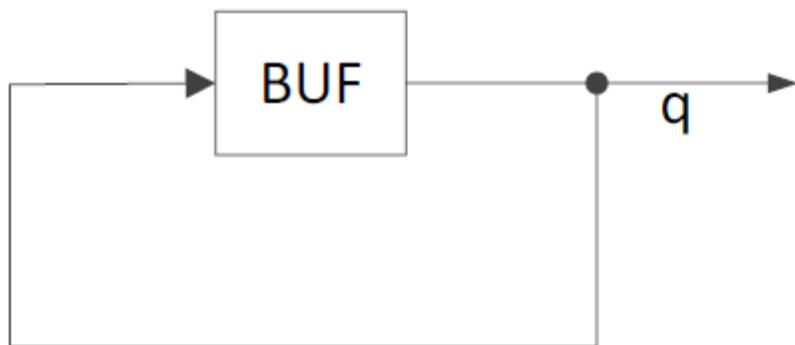


In generale, la memoria si implementa tramite **ANELLI DI RETROAZIONE** (fili di uscita che ritornano in ingresso)

## Reti Asincrone fondamentali

### Derivazione di come si arriva ad un Latch SR

Partiamo prendendo un circuito **BUFFER**, cioè quello che rigenera i segnali elettrici degradati, e collegiamo la sua uscita  $q$  al suo ingresso:



Ovviamente l'uscita  $q$  può valere 0 o 1, ma non è possibile controllare quale dei due valori assumerà quando si fornisce corrente a questo circuito.

**Se levassi il buffer,  $q$  non avrebbe un valore logico perché sarebbe un filo staccato.**

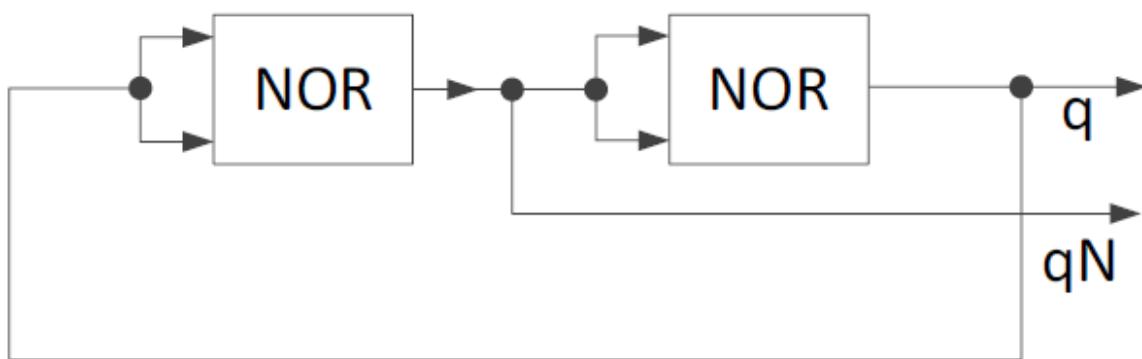


**Le possibili configurazioni in cui si può trovare l'uscita si chiamano STATI.**

In questa rete banale ci sono 2 stati (stabili):

- $S_0$ , in cui  $q = 0$
- $S_1$ , in cui  $q = 1$

Sostituiamo ora il buffer con due porte NOT, sintetizzate a porte NOR, fornendolo pure di una seconda uscita  $qN = \bar{q}$ :



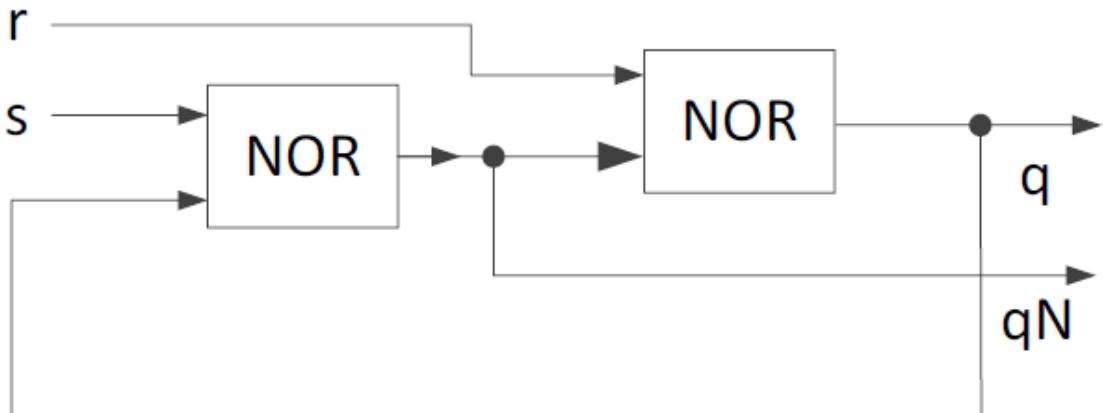
Buffer realizzato a porte NOR

Pure in questa rete però non posso controllare che valore assumono  $q$  e  $qN$ .

Tuttavia, quando do corrente a questo circuito, possono succedere 2 cose:

- $q$  e  $qN$  sono discordi, quindi **la rete si trova in uno stato stabile, in cui resta**
- $q$  e  $qN$  sono concordi. In questo caso:
  - **teoricamente** la rete continua ad oscillare all'infinito
  - **in pratica invece**, visto che il tempo di risposta delle due porte NOR sarà (anche se infinitesimalmente) diverso, **il circuito si porta COMUNQUE in uno stato stabile in cui  $q$  e  $qN$  sono discordi!**

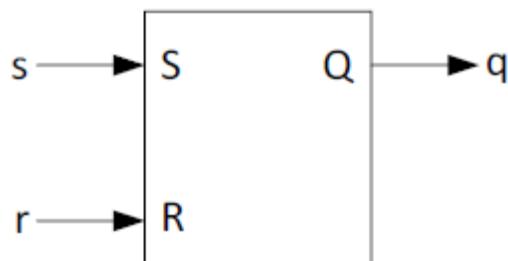
## Latch SR



Circuito che implementa un Latch SR

Ha due variabili di ingresso:

- $s$  SET
- $r$  RESET



Visione funzionale di un Latch SR

Le variabili di ingresso  $s$  ed  $r$  si dicono **ATTIVE ALTE**, cioè di base stanno a 0, e poi quando voglio eseguire l'azione/funzione associata a ciascuna di esse, la porto ad 1

$s$	$r$	$\bar{q} = qN$	$q$	Stato	Comando
1	0	0	1	$S_1$	<b>SET</b>
0	1	1	0	$S_0$	<b>RESET</b>
0	0	$\bar{q}$	$qN$	—	<b>CONSERVAZIONE</b>
1	1	0	0	—	<b>INGRESSO NON CONSENTITO</b>

**TABELLA DI APPLICAZIONE:** Specifica formale per descrivere il comportamento delle Reti Sequenziali (l'analogo della tabella di verità, ma per le reti sequenziali)

Tabella di applicazione del Latch SR:

$q$	$q'$	$s$	$r$
0	0	0	—
0	1	1	0
1	0	0	1
1	1	—	0

Dove:

- $q$  è il **VALORE ATTUALE DELL'USCITA**
- $q'$  è il **VALORE SUCCESSIVO CHE SI VUOLE CHE L'USCITA ASSUMA**
- $(s, r)$  **COMANDO DA DARE** affinché la rete passi da  $q$  a  $q'$



In generale le due **REGOLE DI PILOTAGGIO** valgono anche per le Reti Sequenziali:

- Cambiare gli ingressi solo dopo che la rete è andata a regime (**PILOTAGGIO IN MODO FONDAMENTALE**)
- Stati di ingresso **consecutivi** devono essere **adiacenti**

Fortunatamente però,



**IL LATCH SR E' ROBUSTO A PILOTAGGI SCORRETTI!**



**Lo stato (1, 1) non può essere dato MANUALMENTE come stato iniziale della rete, ma PUO' stare come stato intermedio!**

**Se dessi lo stato (1, 1) in ingresso, e poi volessi passare allo stato (0, 0), GENEREREI BIT CASUALI!**

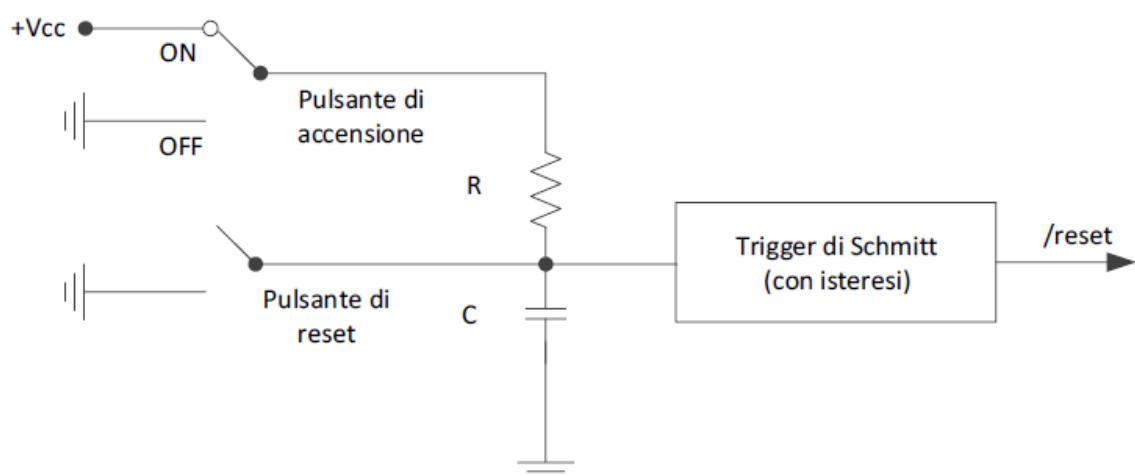
**Perché passerei CASUALMENTE 0 dal comando di set (1, 0) O da quello di reset (0, 1), casualmente, e infine conserverebbe questo stato casuale!!**



**Un Latch SR memorizza un BIT!**

## Inizializzazione degli elementi di memoria

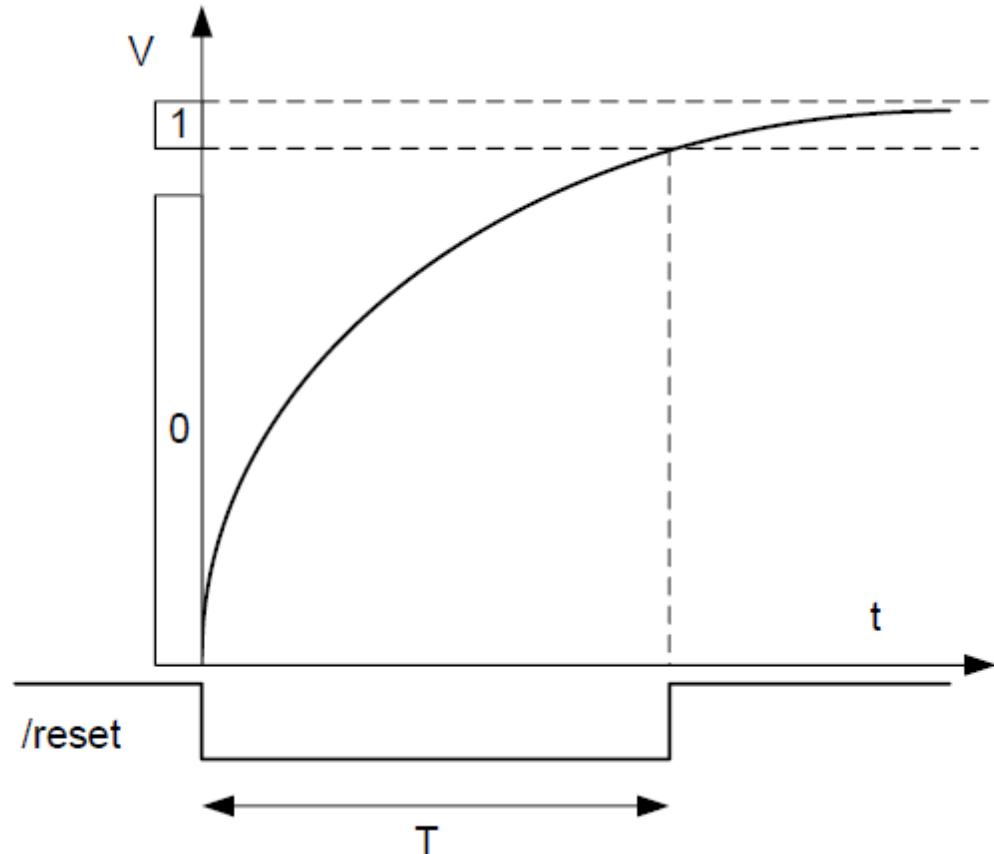
**FASE DI RESET INIZIALE:** Fase, distinta da quella di **Normale Operatività**, nella quale si inizializzano gli elementi di memoria del calcolatore



Quando il pulsante di reset del calcolatore viene premuto, il condensatore si scarica.

Quando il pulsante viene rilasciato, il condensatore comincia a caricarsi.

Il **Trigger di Schmitt** è uno **SQUADRATORE DI TENSIONE**, semplicemente un dispositivo elettrico che converte delle fasce di tensione in bit (0 o 1, a seconda della fascia).



La variabile */reset* è **attiva bassa**, e si usa per inizializzare gli elementi di memoria:

- $/reset = 0 \rightarrow$  inizializzazione in corso dell'elemento di memoria  
**(INDIPENDENTEMENTE DAL VALORE DEI SUOI ALTRI INGRESSI)**
- $/reset = 1 \rightarrow$  l'elemento di memoria funziona normalmente

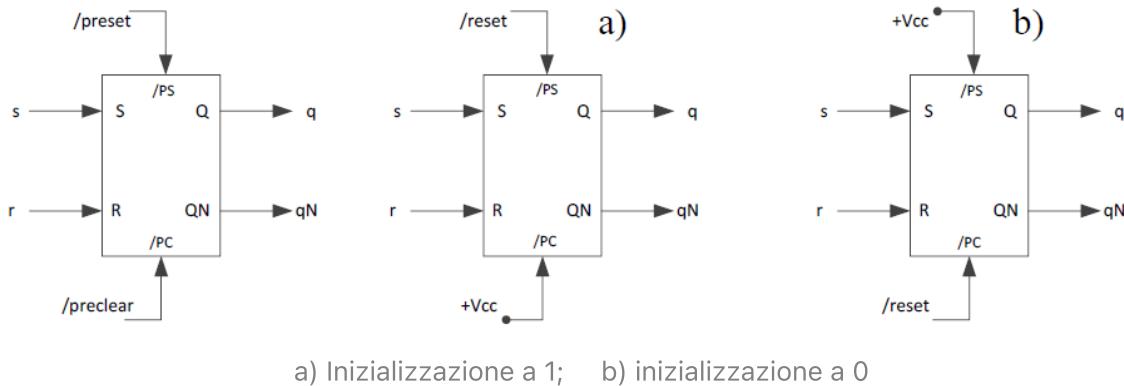
Per poterla usare per inizializzare il Latch SR, però, bisogna dotarlo di 2 ingressi aggiuntivi */preset* e */preclear*, entrambi **attivi bassi**, che si comportano così:

- $/preset = 0 \rightarrow$  la rete si porta nello stato  $S_1$ , **indipendentemente dai suoi altri ingressi**

- $/preclear = 0 \rightarrow$  la rete si porta nello stato  $S_0$ , **indipendentemente dai suoi altri ingressi**
- $/preset = /preclear = 1 \rightarrow$  condizione di **normale operatività**
- $/preset = /preclear = 0 \rightarrow$  **IMPOSSIBILE**

### Come usare $/preset$ e $/preclear$ per inizializzare un Latch SR:

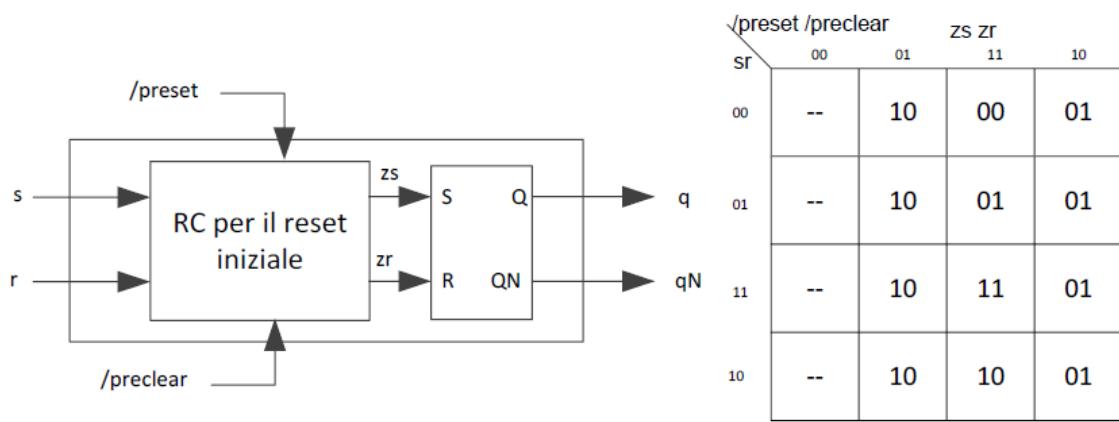
- Se connetti  $/reset$  a  $/preset$  la rete si porta nello stato  $S_1$ , quindi memorizza il bit 1.
- Se connetti  $/reset$  a  $/preclear$  la rete si porta nello stato  $S_0$ , quindi memorizza il bit 0.



In entrambi i casi, l'altra variabile deve essere messa ad 1 (in termini elettrici a  $V_{cc}$ ), così che, visto che sono attive basse, l'altra variabile stia ad 1.

### Latch SR modificato per il reset

Al posto di dotare effettivamente il Latch SR di due nuovi ingressi, si implementa il reset nel Latch SR mettendogli davanti una Rete Combinatoria:



$$z_s = \overline{/preset + (/preclear \cdot s)}$$

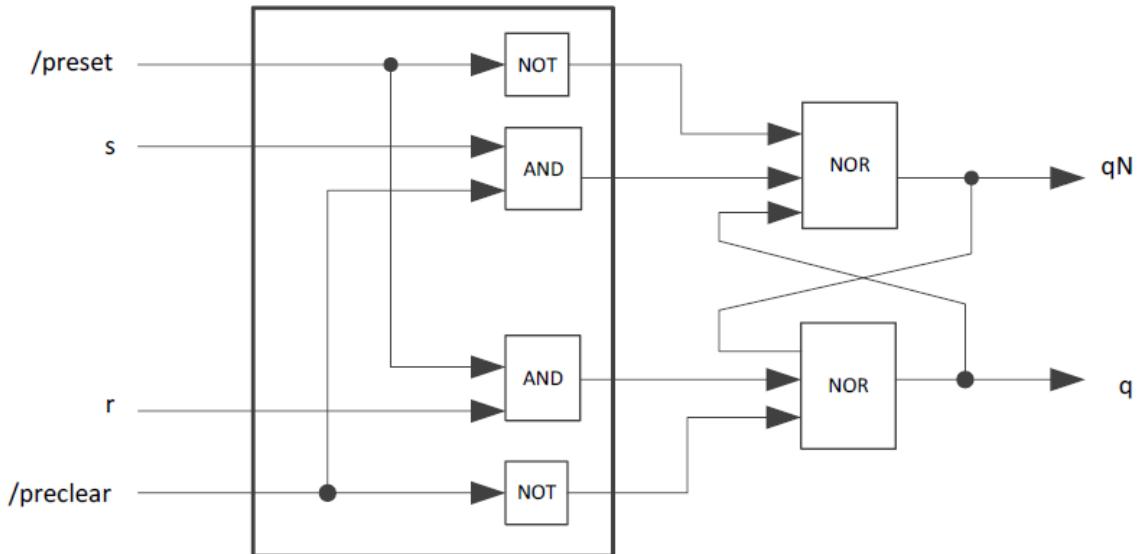
$$z_r = \overline{/preclear + (/preset \cdot r)}$$

Sintesi di un Latch SR con circuito combinatorio per l'inizializzazione

Descrizione della tabella di Karnaugh:

- Quando  $(/preset, /preclear) = (1, 0)$  vuol dire che agisce  $/preclear$ , che quindi porta la rete nello Stato  $S_0$  in cui memorizza il bit 0, ovvero si da il comando di *reset* ( $s = 0, r = 1$ ).
- Quando  $(/preset, /preclear) = (0, 1)$  vuol dire che agisce  $/preset$ , che quindi porta la rete nello Stato  $S_1$  in cui memorizza il bit 1, ovvero si da il comando di *set* ( $s = 1, r = 0$ ).
- Quando  $(/preset, /preclear) = (1, 1)$  c'è la **condizione di normale operatività**, ovvero il Latch sta a sentire i suoi ingressi ( $s, r$ ).
- Ovviamente la situazione  $(/preset, /preclear) = (0, 0)$  è impossibile che si presenti, quindi non mi interessa quanto valgano le uscite  $z_s, z_r$ .

Ma visto che il Latch SR è fatto a porte NOR, e nella sintesi c'è un OR (+), si può ottimizzare/semplicificare il circuito così:



Latch SR ottimizzato con Rete Combinatoria evidenziata con la riga nera, che gestisce il reset, sintetizzato a porte NOR

**Spiegazione:** si sfruttano le porte NOR dentro all'effettivo Latch SR per fungere anche da OR, perché un OR fatto a porte NOR è composto da  $xNORy$  e poi questo risultato viene negato, a porte NOR, cioè si sdoppia e si butta in una porta NOR, ma questo è già presente dentro il Latch SR, e allora tanto vale metterci un ingresso in più in quelle NOR e ottimizzare!

## Tabelle e grafi di flusso

Le RSA si descrivono tramite **TABELLE DI FLUSSO** o **GRAFI DI FLUSSO**.

**TABELLA DI FLUSSO:** Tabella che descrive come si evolvono lo **Stato Interno** e l'**uscita** della rete, al variare degli **Stati di Ingresso** (in pratica  $(S, Z) = f(X)$ ).

Una Tabella di Flusso è una **MATRICE** che ha:

- Come righe gli Stati Interni ( $S_0, S_1, etc \dots$ )
- Come colonne gli Stati di Ingresso ( $(s, r) = \{(0,0), (0,1), (1,0), (1,1)\}$ )
- Nelle celle, i **NOMI** degli Stati Interni della rete

Dove:

- Gli stati in riga sono lo **STATO INTERNO PRESENTE**
- Gli stati nelle celle sono lo **STATO INTERNO SUCCESSIVO**

		sr	00	01	11	10	q
		s0	$S_0$	$S_0$	-	$S_1$	0
		s1	$S_1$	$S_0$	-	$S_1$	1

Tabella di flusso di un Latch SR

La rete transisce allo Stato  $S_{ij}$  quando:

- Si trova nello stato  $S_i$
- C'è in ingresso lo stato di ingresso  $(s, r)_j$

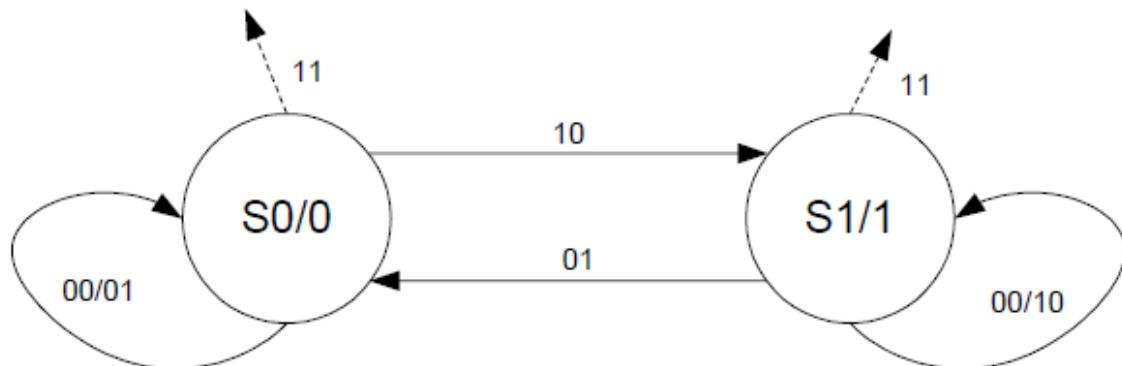
Inoltre, per le RSA, visto che l'uscita ( $q$  in questo caso) è direttamente funzione degli ingressi (cioè degli stati  $S_0$  e  $S_1$ ), la posso mettere a fianco.

Per ciascuno Stato Interno  $S_i$ , nella tabella di flusso si **CERCHIANO** gli **STATI STABILI**, ovvero, per ogni stato  $S_i$ , le **celle corrispondenti a stati di ingresso  $(s, r)$  (colonne)** che fanno mantenere la rete nello stato  $S_i$ .

Un **GRAFO DI FLUSSO** è costituito da:

- Un insieme di **nodi**, ciascuno rappresenta uno Stato Interno
- Un insieme di **archi** diretti da un nodo (Stato Interno) all'altro, etichettati con uno stato di ingresso. Gli archi possono:
  - Andare da uno stato  $S_i$  a uno stato  $S_j$  diversi
  - Rimanere nello stesso stato (**STATO STABILE** per quegli ingressi)

- Andare all'infinito, ovvero sono stati di ingresso che non si possono verificare

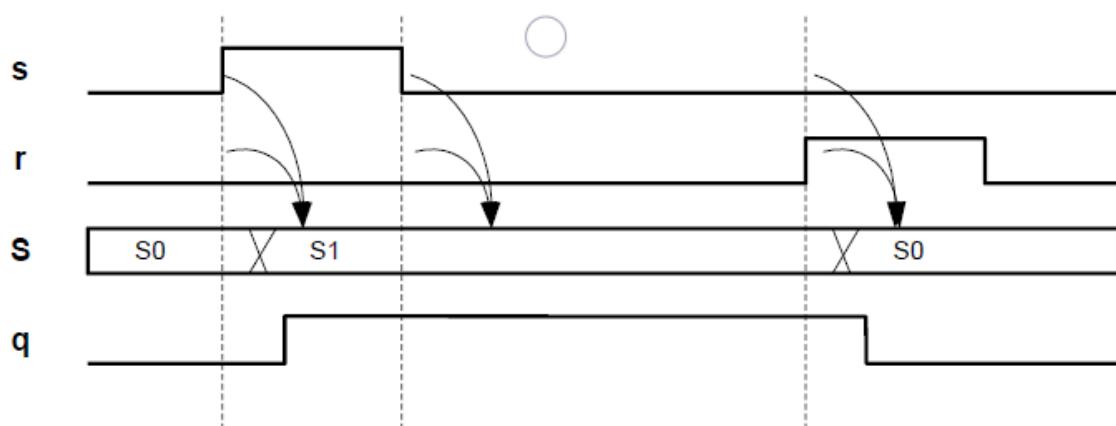


Grafo di flusso di un Latch SR



**Il modo per debuggere una Rete Sequenziale, sia asincrona che sincronizzata, è un **DIAGRAMMA DI TEMPORIZZAZIONE**, che mostra l'evoluzione dello Stato Interno in funzione:**

- Dello stato di ingresso nel caso di Reti Asincrone
- Del tempo nel caso di Reti Sincronizzate

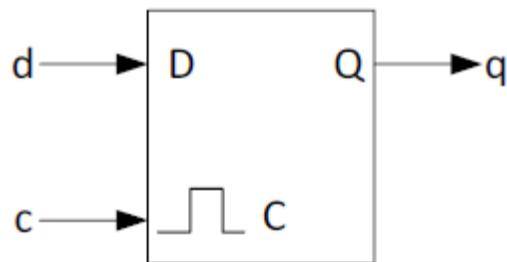


Esempio di un diagramma di temporizzazione per un Latch SR (RSA)

## D-Latch (trasparente)

Il D-Latch ha anch'esso, come il Latch SR, due variabili di ingresso:

- $d$  DATA
- $c$  CONTROL



Visione funzionale di un D-Latch

Descrizione a parole:

- Se  $c = 1$  (**TRASPARENZA**) il D-Latch **MEMORIZZA L'INGRESSO  $d$** .
- Se  $c = 0$  (**CONSERVAZIONE**) il D-Latch **MEMORIZZA L'ULTIMO VALORE CHE AVEVA  $d$  QUANDO  $c$  VALEVA ANCORA 1**.

		$c=0$		$c=1$		$q$
		0	1	0	1	
$d$	$s_0$	$S_0$	$S_0$	$S_0$	$S_1$	0
	$s_1$	$S_1$	$S_1$	$S_0$	$S_1$	1

Tabella di flusso di un D-Latch



**In una Tabella di Flusso non per forza stati di ingresso consecutivi devono essere adiacenti!**  
**Non è una Mappa di Karnaugh e non la devo sintetizzare!**

Per memorizzare un bit con un D-Latch basta:

1. Portare  $c$  ad 1
2. Impostare  $d$  al valore da memorizzare
3. Riportare  $c$  a 0

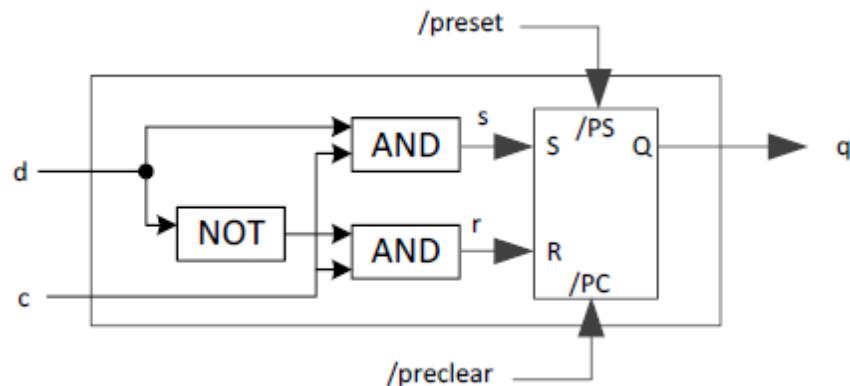
### **Sintesi di un D-Latch mediante Latch SR:**

Prendo un Latch SR e gli metto davanti una RC che ha come ingressi  $(c, d)$  e come uscite  $(s, r)$ :

$c$	$d$	$s$	$r$
0	—	0	0
1	0	0	1
1	1	1	0

### **Spiegazione:**

- Se  $c = 0$ , il D-Latch è in conservazione, quindi il valore di  $d$  corrente è ininfluente, devo dare al Latch SR il **Comando di CONSERVAZIONE**  $(0, 0)$
- Se  $c = 1$ , il D-Latch è in trasparenza, quindi devo memorizzare il valore corrente di  $d$ :
  - Se  $d = 0$  devo memorizzare il bit 0, quindi devo dare il **Comando di Reset**  $(0, 1)$
  - Se  $d = 1$  devo memorizzare il bit 1, quindi devo dare il **Comando di Set**  $(1, 0)$



Sintesi di un D-Latch a partire da un Latch SR, con anche i piedini per l'inizializzazione al reset

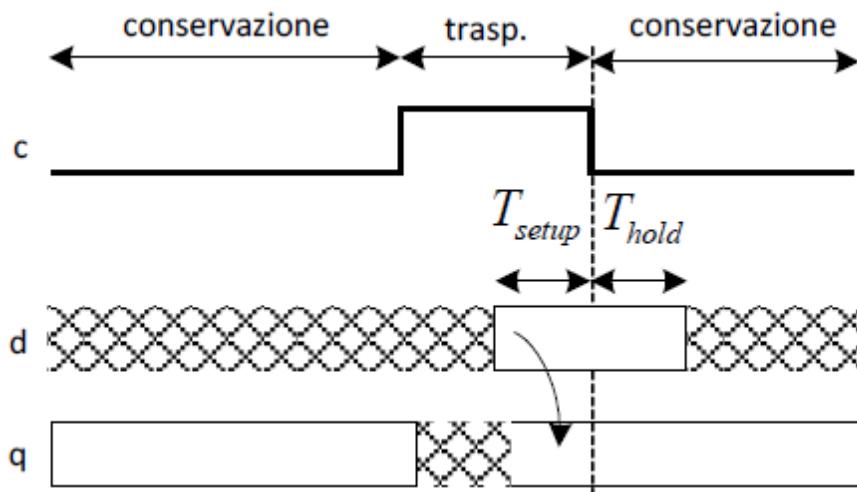
## Regole di pilotaggio di un D-Latch



d deve essere COSTANTE A CAVALLO della transizione di c da 1 a 0!!!

Per quanto?

- Da  $T_{setup}$  prima
- A  $T_{hold}$  dopo



### Spiegazione:

La transizione di  $0 \rightarrow 1$  di  $c$  non da nessun problema, perché passo da una situazione di **CONSERVAZIONE**, ad una di **TRASPARENZA**.

La transizione  $1 \rightarrow 0$  invece, da problemi per questo semplice fatto qui:

**Se  $d$  non stesse fermo durante questa transizione, finirei per conservare un valore sbagliato, perché c'è un tempo FISICO di transizione ( $\neq 0$ ) che ci mette una qualsiasi variabile (in questo caso  $c$ ) a cambiare valore (perché sono segnali elettrici alla fine)!**

Ovviamente io conservo il valore che aveva  $d$  quando ho iniziato il processo di transizione  $1 \rightarrow 0$ !

Ma se mi cambia  $d$  in questo tempo, io non lo posso sapere, e finisco per sbagliare!

### Esempio:

- INIZIO PROCEDURA TRANSIZIONE  $1 \rightarrow 0$ :  $[d = 1, c = 1]$
- TRANSIZIONE  $1 \rightarrow 0$  IN CORSO:  $[d = 0, c = 1]$
- TRANSIZIONE  $1 \rightarrow 0$  IN CORSO:  $[d = 1, c = 1]$
- TRANSIZIONE  $1 \rightarrow 0$  IN CORSO:  $[d = 0, c = 1]$
- TRANSIZIONE  $1 \rightarrow 0$  IN CORSO:  $[d = 0, c = 1]$
- TRANSIZIONE  $1 \rightarrow 0$  IN CORSO:  $[d = 0, c = 1]$
- FINE PROCEDURA TRANSIZIONE  $1 \rightarrow 0$ :  $[d = 1, c = 0]$

Qui, io memorizzo 1, ma nel mentre che  $c$  stava passando da 1 a 0,  $d$  ha avuto il tempo di cambiare valore "4" volte! Quindi non memorizzo il valore che  $d$  aveva quando EFFETTIVAMENTE  $c$  ha cambiato valore, ma solo quello  $\Delta t$  prima di questo cambio, che sarebbe il tempo fisico che ci mette  $c$  a cambiare!

Questo comporta che



**Il D-Latch è una rete TRASPARENTE, cioè la sua uscita CAMBIA MENTRE la rete è sensibile alle variazioni dell'ingresso ( $d$  in questo caso, discorso appena sopra)**

Per questo motivo si stabiliscono le regole di pilotaggio e si richiede che  $d$  stia fermo per  $T_{setup}$  prima fino a  $T_{hold}$  dopo il fronte di discesa di  $c$ !



**Il D-Latch memorizza sul FRONTE DI DISCESA di  $c$ !**



**Reti trasparenti:**

- Reti Combinatorie
- Latch SR
- D-Latch

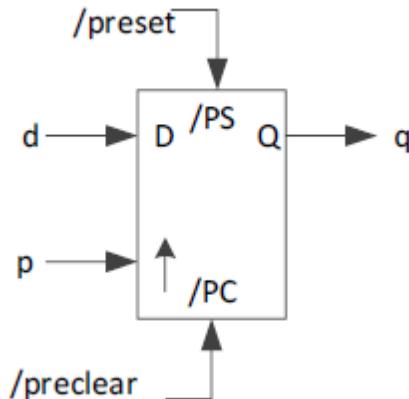
## Positive edge-triggered D-Flip-Flop



**Elementi di memoria/reti TRASPARENTI si chiamano LATCH.  
Elementi di memoria/reti NON TRASPARENTI si chiamano FLIP-FLOP.**

Ha due variabili di ingresso:

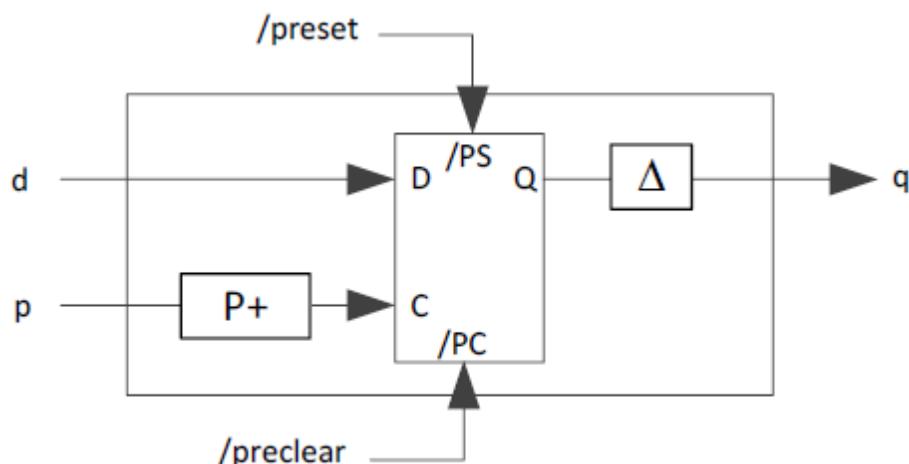
- $d$
- $p$



Visione funzionale di un D-Flip-Flop positive edge-triggered

Descrizione:

**Quando  $p$  ha un FRONTE DI SALITA, memorizza  $d$ , aspetta un pochino, e POI adegua l'uscita!**



**Realizzare (idealmente) un D-Flip-Flop tramite un D-Latch:**

- Si prende un D-Latch e si mette un circuito formatore di impulsi davanti a  $c$ . Così, al fronte di salita di  $p$ , il D-Latch va brevemente in trasparenza e sta a sentire  $d$ .

Poi torna subito in conservazione e a quel fronte di discesa memorizza  $d$ .

Quindi è lecito dire che, visto che l'impulso è di durata piccola, un D-FF memorizza sul fronte di SALITA di  $p$ , perché è molto vicino al fronte di discesa.

- Inoltre si deve ritardare l'uscita di un ritardo  $\Delta > P^+$ .  
Così, quando  $q$  cambia adeguandosi a  $d$ , **la rete non è più in trasparenza, ma in conservazione!**



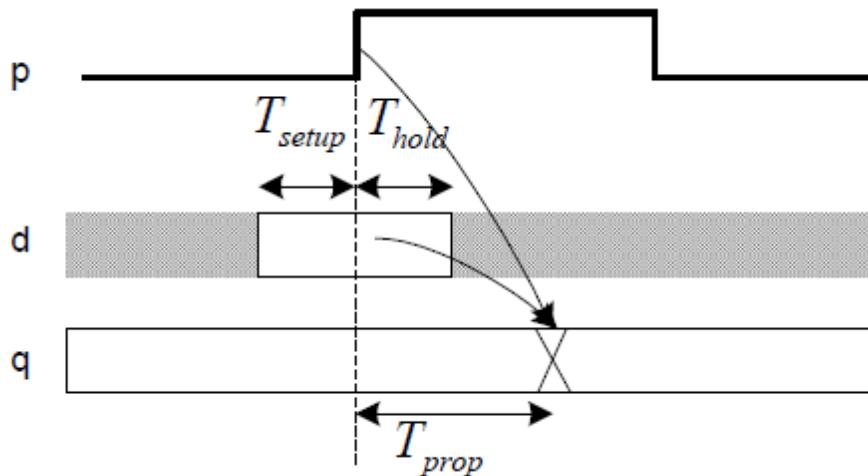
**L'uscita  $q$  viene adeguata al valore di  $d$  DOPO che la rete ha smesso di essere sensibile al valore di  $d$  (cioè dopo che è tornata in CONSERVAZIONE)**

Differenza con un D-Latch:

- Nel D-Latch si memorizza  $d$  quando  $c$  è su un **LIVELLO (1)**
- Nel D-Flip-Flop si memorizza  $d$  quando  $p$  **PASSA DA 0 A 1**, quindi **qui si memorizza SU UNA TRANSIZIONE (e SOLO sul fronte di salita)**, mentre nel D-Latch si memorizza su un **LIVELLO!**

### Regole di pilotaggio di un D-Flip-Flop

- **A cavallo del fronte di salita di  $p$ , l'ingresso  $d$  deve rimanere costante da  $T_{setup}$  prima fino a  $T_{hold}$  dopo!**  
(I nomi sono gli stessi di quelli dei tempi di un D-Latch, ma non sono gli stessi)
- **Tra due fronti di salita di  $p$  deve passare ALMENO  $T_{prop}$ , cioè il tempo che ci mette l'uscita ad adeguarsi agli ingressi, misurato dal fronte di salita di  $p$ .**  
**(Questo garantisce che concettualmente il D-Latch sia in conservazione quando arriverà il nuovo fronte di salita di  $p$ )**

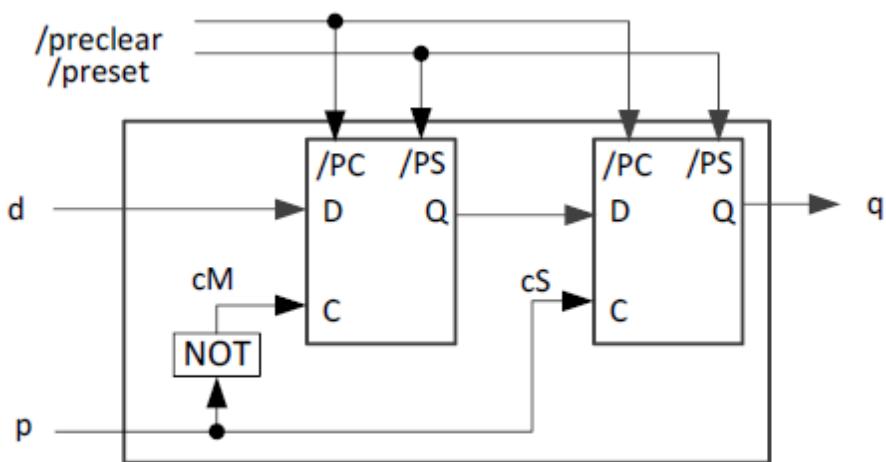


**CONDIZIONE DI NON TRASPARENZA:**  $T_{prop} > T_{hold}$



Visto che i D-Flip-Flop sono non trasparenti, posso montarli in qualsiasi modo!

## Sintesi di un D-Flip-Flop con montaggio Master/Slave



Qui uno dei due Flip-Flop (in realtà sono D-Latch) è sempre in conservazione!

- Quando  $p = 0$  il master memorizza  $d$ , mentre lo slave è in conservazione, quindi non sta a sentire il master ma conserva il valore che aveva  $d$  quando  $p$  valeva 1
- Quando  $p = 1$  il master è in conservazione, mentre lo slave memorizza  $Q$

Quindi, quando si presenta un fronte di salita di  $p$ :

1. Il master memorizza  $d$  e lo slave non lo sta a sentire
2. Il master va in conservazione, quindi stacca l'ingresso  $d$  dalla rete e presenta  $d$  in uscita
3. Lo slave va in trasparenza e memorizza, e presenta in uscita  $T_{prop}$  dopo il fronte di salita di  $p$ , il  $d$  precedentemente memorizzato dal master al punto 1

Per evitare transitori in cui entrambi sono in trasparenza si agisce per via elettronica.

## Memorie

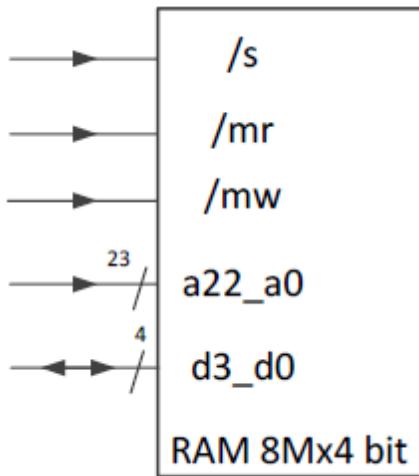
### Memorie RAM Statiche (S-RAM)

Le S-RAM sono matrici di D-Latch, dove:

- Ogni riga costituisce una **LOCAZIONE DI MEMORIA** (1 bit  $\iff$  1 D-Latch)
- Il numero di colonne costituisce il numero di bit di una locazione di memoria



$$2^{10} = 1KB, \quad 2^{20} = 1MB, \quad 2^{30} = 1GB$$



Visione funzionale di un modulo di RAM

Una **RAM** ha:

- Degli **ingressi** chiamati **FILI DI INDIRIZZO**, capaci di indirizzare tutte le celle di memoria.  
La memoria qui di esempio contiene  $8M = 2^3 \cdot 2^{20} = 2^{23}$  celle, quindi ci vogliono 23 fili di indirizzo  $a_{22} \dots a_0$  (sono variabili logiche)
- Degli **ingressi/uscite** (ci vuole una tri-state) chiamati **FILI DI DATI**, che **portano i dati** da leggere/scrivere dentro o fuori dal modulo di S-RAM.  
In questo esempio, visto che ogni locazione di memoria è da 4 bit, i fili di dato sono 4:  $d_3 \dots d_0$
- Due **variabili logiche di ingresso attive basse**, **Memory Read**  $/mr$  e **Memory Write**  $/mw$ .  
Servono a dare il comando o di lettura o di scrittura alla locazione il cui indirizzo è sui fili di indirizzo  $a_{22} \dots a_0$   
**Non devono MAI essere attivi contemporaneamente!**
- Una **variabile logica attiva bassa di ingresso** di **Select**  $/s$ , che viene **resettata quando questo specifico modulo di memoria è selezionato per eseguire un'operazione di read/write**.  
Quando  $/s = 1$  questo modulo di memoria **NON SENTE GLI INGRESSI, NESSUNO!**  
Quando  $/s = 0$  questo modulo di memoria reagisce agli ingressi.  
**Questo segnale di select è un pò come un “enabler della RAM”, che dice a ciascun modulo/pezzo di S-RAM se i dati e l’indirizzo presenti sui fili sono**

per lui o no.

**Il segnale di select serve quindi a rendere espandibile la S-RAM!**

<i>/s</i>	<i>/mr</i>	<i>/mw</i>	Azione
1	-	-	Nessuna azione (memoria non selezionata)
0	1	1	Nessuna azione (memoria selezionata, nessun ciclo in corso)
0	0	1	Ciclo di lettura in corso
0	1	0	Ciclo di scrittura in corso
0	0	0	Non definito



Un modulo di RAM/ROM  $2^X B \times Y$  bit ha  $X$  fili di indirizzo e  $Y$  fili di dato!

Esempi:

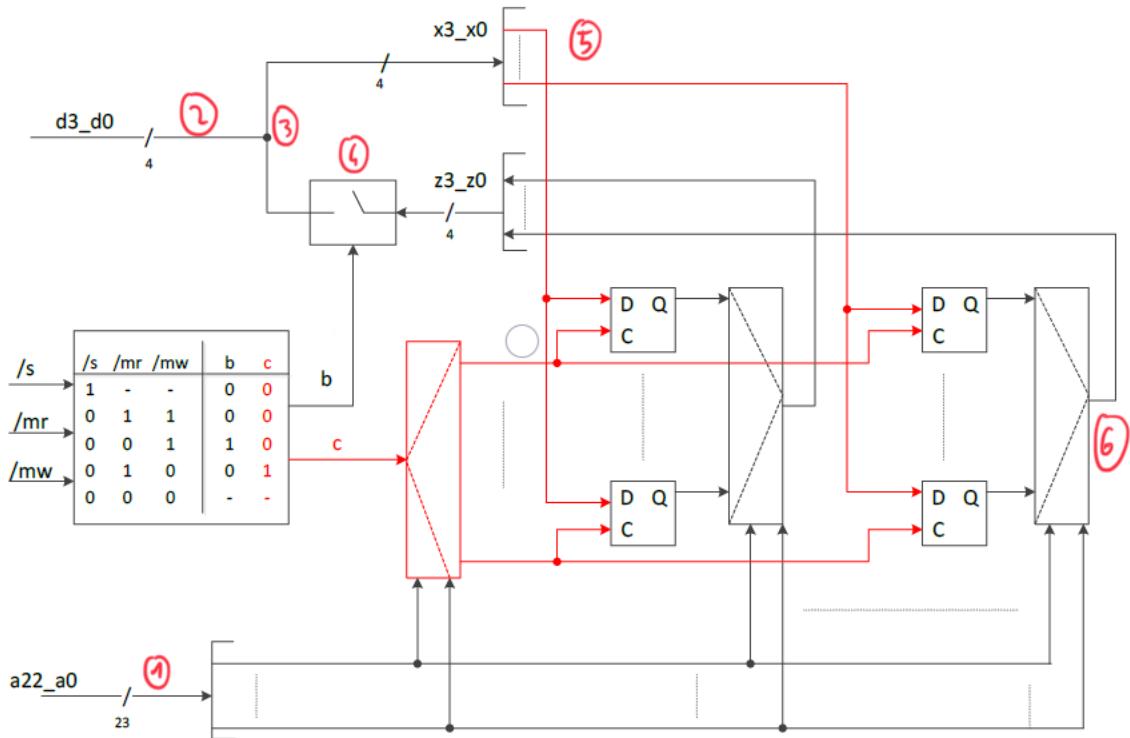
Un modulo da  $4KB \times 8$  bit ha 12 fili di indirizzo  $a_{11} \dots a_0$  e 8 fili di dato  $d_7 \dots d_0$ !

Un modulo da  $16MB \times 4$  bit ha 24 fili di indirizzo  $a_{23} \dots a_0$  e 4 fili di dato  $d_3 \dots d_0$ !

Un modulo da  $1GB \times 16$  bit ha 30 fili di indirizzo  $a_{29} \dots a_0$  e 16 fili di dato  $d_{15} \dots d_0$ !

## Implementazione delle operazioni di r/w

Abbrevio "operazione di scrittura" con *write*, e "operazione di lettura" con *read*



Implementazione di una memoria S-RAM di 8Mx4bit (matrice  $2^{23} \times 4$ ), con circuiteria per le operazioni di read e write

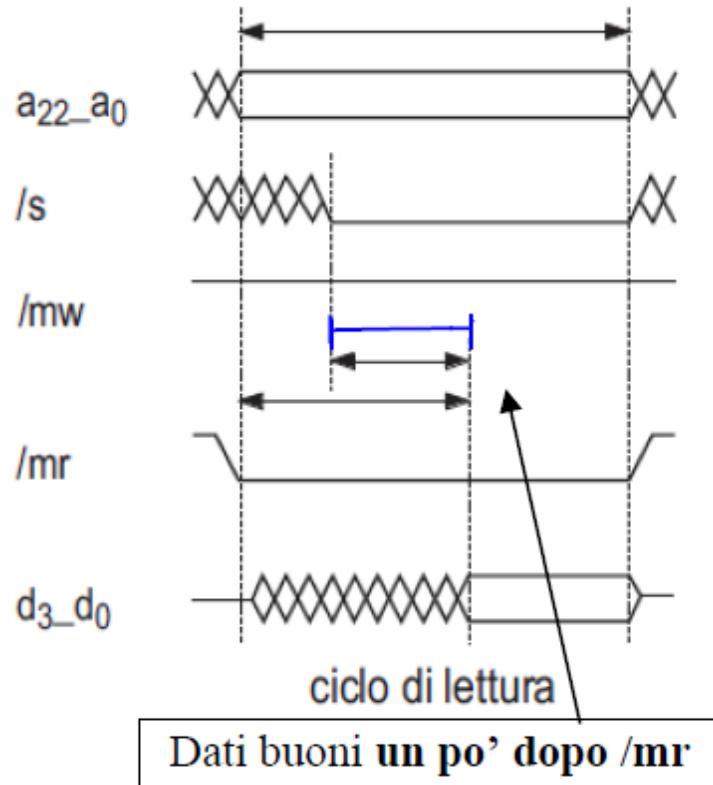
1. Quando voglio fare una *read*, questi fili contengono l'indirizzo in base 2 della cella di memoria nella quale voglio leggere. Idem per una *write*.  
Questi fili in sostanza **SCEGLONO** qual è la **RIGA DI D-LATCH SELEZIONATA** per un'operazione.
2. Quando eseguo una *write*, qui ci sono i dati che voglio scrivere.
3. Visto che  $d_3 \dots d_0$  sono fili sia di input che di output, ci vuole per forza il montaggio a forchetta, oltre alle tri-state!
4. Per quanto riguarda le tri-state appunto, questo è un BLOCCO di (in questo caso) 4 tri-state, una per ciascun filo di dato!  
Devono essere abilitate (cioè far passare i dati) quando sto facendo una *read*, perché i dati prelevati dalla memoria  $z_3 \dots z_0$  devono andare sul bus dati e andare a chi li ha richiesti!
5. **Cosa accade in una write:**
  - a. **Porto TUTTI e 4 i bit  $x_3 \dots x_0$  a TUTTI i D-Latch:** da  $x_0$  alla colonna più a destra fino a  $x_3$  a quella più a sinistra, così ogni riga di D-Latch ha ingresso il dato  $(d_3 d_2 d_1 d_0)_2$ .

- b. Ora, la riga opportuna nella quale si vuole scrivere viene selezionata dal DEMUX che manda l'ingresso  $C = c = 1$  (segnale di *write*) alla riga di D-Latch selezionata dai fili di indirizzo  $a_{22} \dots a_0$  che lo guidano.  
Le altre righe di D-Latch avranno  $C = 0$ .
- c. Così, solo i 4 D-Latch della riga selezionata staranno a sentire l'ingresso corrispondente  $d_i = z_i$ , mentre tutte le altre righe staranno in conservazione non sentendo l'ingresso.  
In termini pratici, nella riga selezionata, visto che sente l'ingresso, ci verrà scritto  $(d_3 d_2 d_1 d_0)_2$ , mentre nelle altre righe ci verrà ri-scritto il valore che c'era prima!

#### **6. Cosa accade in una *read*:**

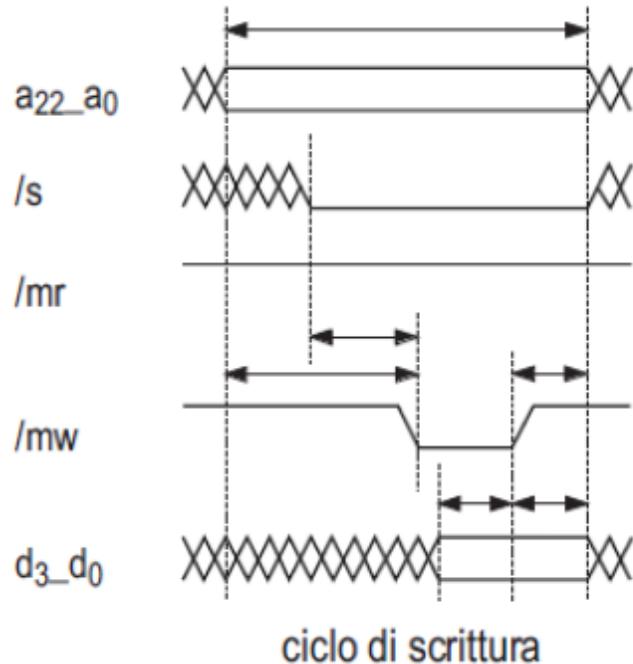
- a. Si seleziona la riga (locazione di memoria) in cui si vuole leggere tramite 4 MUX guidati dai fili di indirizzo  $a_{22} \dots a_0$ .
- b. Le uscite dei 4 MUX inseguono i corrispettivi 4 bit  $Q_i$ , per cui in uscita ci saranno i dati che voglio leggere.
- c. Prima di mandarli sul bus a chi li ha richiesti bisogna che le 4 tri-state ricevano il comando di *read*, così da assicurarsi di poter usare i 4 fili di dato  $d_3 \dots d_0$  come fili di output, ovvero quando  $b = 1$  (segnale di *read*).

### **Temporizzazione del ciclo di lettura in memoria**



1. I **fili di indirizzo**  $a_{22} \dots a_0$  si stabilizzano al valore della locazione in cui si vuole scrivere e arriva il comando di *read*:  $/mr = 0$ .
2. Il **comando di select**  $/s = 0$  si stabilizza un pò dopo, perché viene generato per ogni modulo/elemento di memoria da delle RC dette **MASCHERE**.
3. Quando  $/s = /mr = 0$  le **tri-state vanno in conduzione**, e i **MUX** restituiscono i dati buoni da poter prelevare.
4. Infine si termina l'operazione di lettura ritirando su  $/mr = 1$ , dunque le tri-state si disattivano, i fili di dato tornano in alta impedenza, e si è concluso il ciclo di lettura

### Temporizzazione del ciclo di scrittura in memoria



1. I **fili di indirizzo**  $a_{22} \dots a_0$  si stabilizzano per primi alla locazione in cui si vuole scrivere.
2. Il **comando di select**  $/s = 0$  si stabilizza un pò dopo, perché viene generato per ogni modulo/elemento di memoria da delle RC dette **MASCHERE**. (UGUALE ALLA LETTURA)
3. **Va dato il comando di write**  $/mw = 0$  **DOPPO che SIA gli indirizzi CHE IL SELECT (SOPRATTUTTO!) siano apposto**, perché i dati vengono portati ad ogni locazione di memoria, e poi viene scelta quella giusta, ma i problemi sono che:
  - a. Se  $/s$  è sbagliato perché non si è ancora assestato sul valore corretto, finisco per dare l'ordine di scrivere ad un elemento di memoria sbagliato, che magari sta facendo altro!
  - b. Se l'indirizzo  $a_{22} \dots a_0$  è sbagliato perché ancora non si è assestato sul valore corretto, finisco per scrivere in parti di memoria casuali involontariamente!
4. I fili di dato basta che siano buoni **a cavallo del FRONTE DI SALITA** di  $/mw$ , che corrisponde al **FRONTE DI DISCESA** di  $c$ , perché è il passaggio 1 – 0 dei D-Latch che porta problemi se il dato  $d_i$  non è stabile da  $T_{setup}$  prima a  $T_{hold}$  dopo! (qui in realtà è un pochino di più perché c'è pure il ritardo della RC che

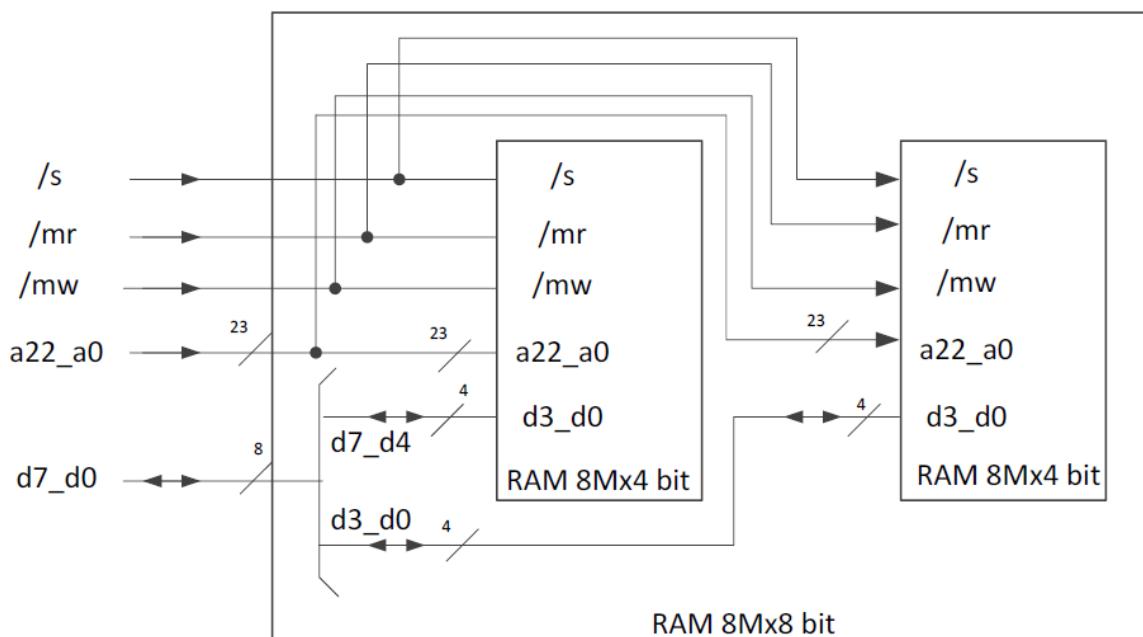
sintetizza  $b$  e  $c$ ).

Comunque:

- a.  $/mw = 0 \rightarrow c = 1 \rightarrow$  D-Latch in trasparenza, memorizzo il dato e lo scrivo nella cella
  - b.  $/mw = 1 \rightarrow c = 0 \rightarrow$  D-Latch in conservazione, finalizzo la scrittura del dato in memoria
5. Una volta scritti i dati in memoria posso riportare  $/mw = 1$  e sia i dati, che gli indirizzi, che il segnale di select possono cambiare a questo punto.

## Raddoppio della capacità delle celle di memoria

Voglio ottenere una memoria  $8M \times 8\text{ bit}$  usando banchi da  $8M \times 4\text{ bit}$ :



**Per raddoppiare la grandezza di ogni locazione di memoria basta mettere in parallelo un altro modulo/elemento di memoria**

Tutti i fili vanno in parallelo anche a questo nuovo modulo, con una sola differenza:

Visto che **ora i dati sono ad 8bit e non più a 4**, devo avere nella rete totale un **filo di dato  $d_7 \dots d_0$  ad 8 bit, che "smezzo" fra i due banchi di memoria**.

## Raddoppio del numero di locazioni di memoria

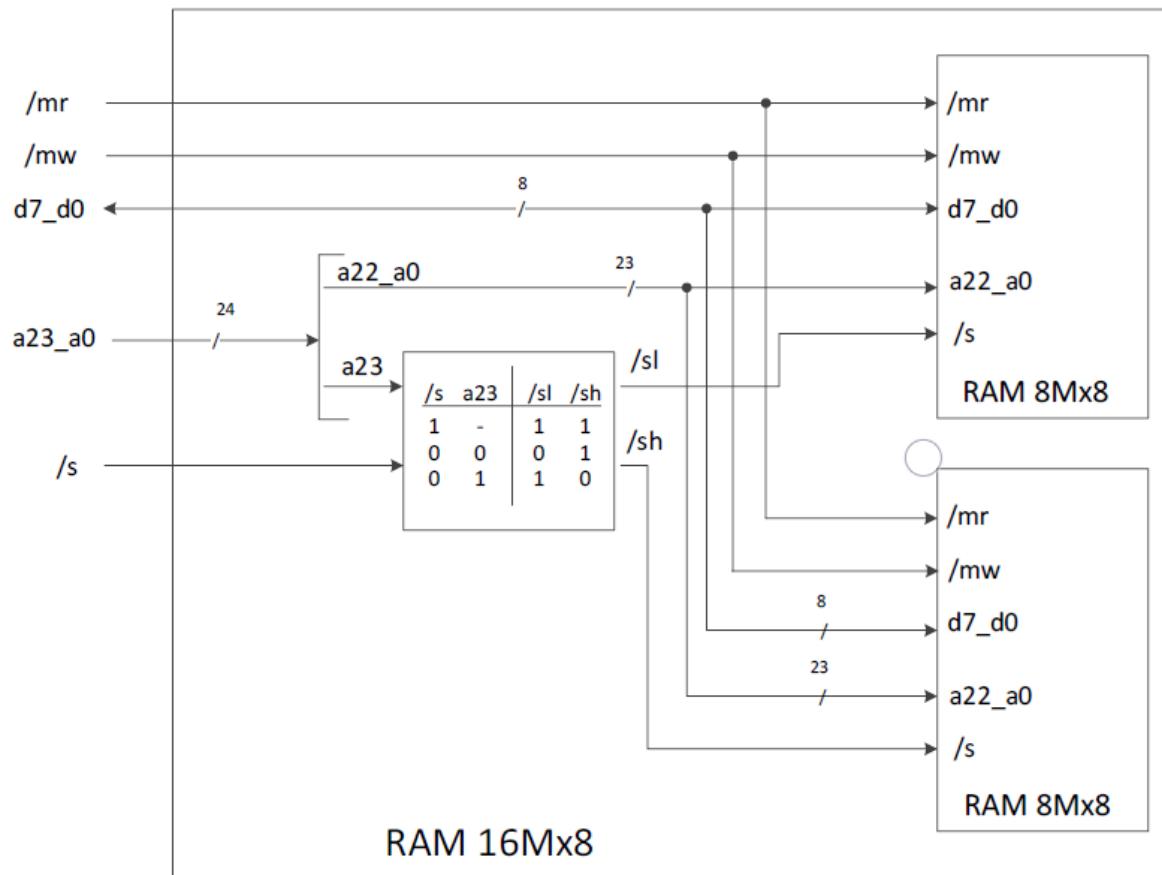
Voglio ottenere una memoria  $16M \times 8\text{ bit}$  usando banchi da  $8M \times 8\text{ bit}$ , e visto che  $16M = 2^4 \cdot 2^{20} = 2^{24}$ , ci vogliono ora non più 23 ma 24 fili di indirizzo!

Visto che voglio usare banchi da 8 bit, divido le locazioni in:

- Parte "alta" in un modulo di memoria, dove  $a_{23} = 1$
- Parte "bassa" nell'altro modulo, dove  $a_{23} = 0$

Infine, il segnale di **select**  $/s$  viene generato usando il valore di  $a_{23}$ .

Tutto il resto viene portato in parallelo ai due banchi di memoria come prima!



Qui si è deciso (arbitrariamente) che il segnale  $/sl$  (select low) abiliti a funzionare il chip di RAM sopra, mentre  $/sh$  (select high) quello sotto, ma è del tutto arbitrario!

**Ecco perché il segnale  $/s$  nelle operazioni di *read/write* si stabilizza dopo un pò!**

**Perché c'è una RC che si occupa di generarlo per ciascun modulo di memoria!**

Il fatto che i fili di dato  $d_7 \dots d_0$  siano collegati insieme ad entrambi i banchi di memoria non genera alcun problema, dato che in ogni istante ci sarà solo uno fra i due segnali di select  $/sl$  e  $/sh$  che sarà 0! (cioè attivo)

**Questo montaggio quindi NON permette anche di eseguire le operazioni in parallelo, permette solo di estendere il numero di locazioni di memoria totali, avendole su due banchi!**

## **Collegamento della memoria al bus dati**

**I fili di indirizzo della memoria provengono da un BUS INDIRIZZI**

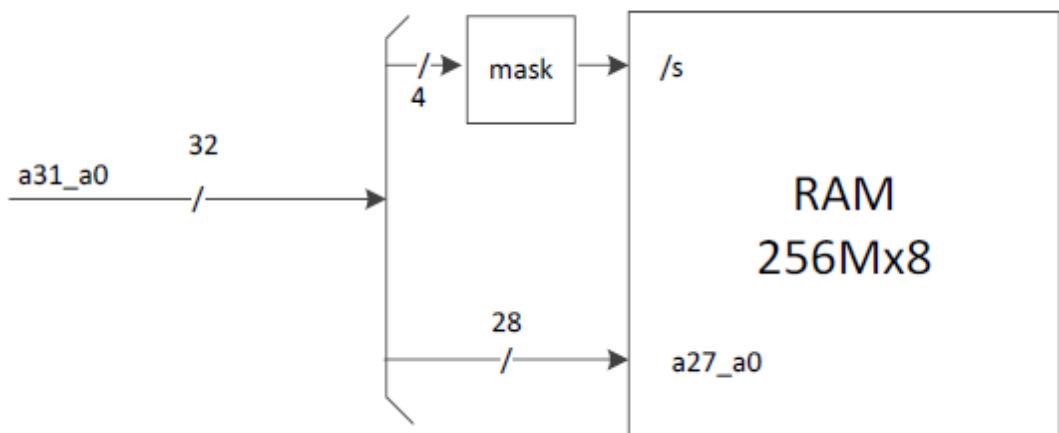
### **Esempio:**

Supponiamo di avere un bus indirizzi a 32 bit, e di voler montare un modulo di RAM  $256M \times 8$  bit a partire dall'indirizzo  $0xE0000000$ .

Ovviamente questo modulo di RAM avrà  $2^{28}$  fili di indirizzo (perché  $256 = 2^8$ ) e un filo di select  $/s$  che dovrà rispondere agli indirizzi nell'intervallo  $0xE0000000 - 0xFFFFFFFF$ .

Visto che il numero di zeri e di "F" è 7, e ogni carattere in quegli indirizzi corrisponde a 4 bit ( $4 \cdot 8 = 32$  bit, infatti il bus è a 32 bit), avrò che:

- I 28 fili di indirizzo meno significativi (28 perché  $4 \cdot 7 = 28$ ) del bus andranno in ingresso al modulo di RAM.
- I restanti 4 fili di indirizzo più significativi andranno in ingresso ad una RC chiamata **MASCHERA**, che **genera il segnale di select per il modulo di RAM**.



Quello che sostanzialmente la maschera deve fare, è **RICONOSCERE**  $0xE \dots$ , e la si può sintetizzare vedendo che  $(0xE)_{16} = (15)_{10} = (1110)_2$

La RC *mask* che sintetizza il segnale  $/s$  è banale:

a31	a30	a29	a28		$/s$
1	1	1	0		0
others					1

Quindi  $/s = \overline{a_{31}} + \overline{a_{30}} + \overline{a_{29}} + a_{28}$

## Memorie Read-Only (ROM)

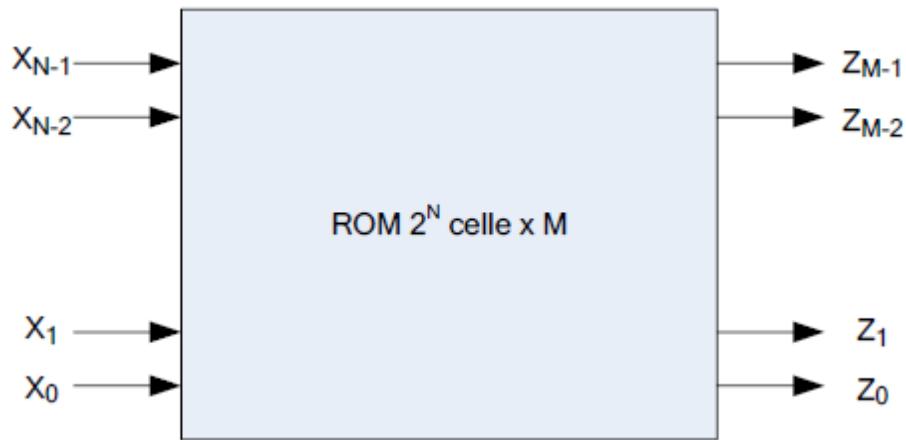


**Le memorie ROM sono circuiti combinatori!**  
**Infatti una ROM è un decoder  $N$ -to- $2^N$  seguito da una barriera di  $M$  porte OR!**

Una memoria ROM di  $2^N$  locazioni di memoria di  $M$  bit ciascuna ha:

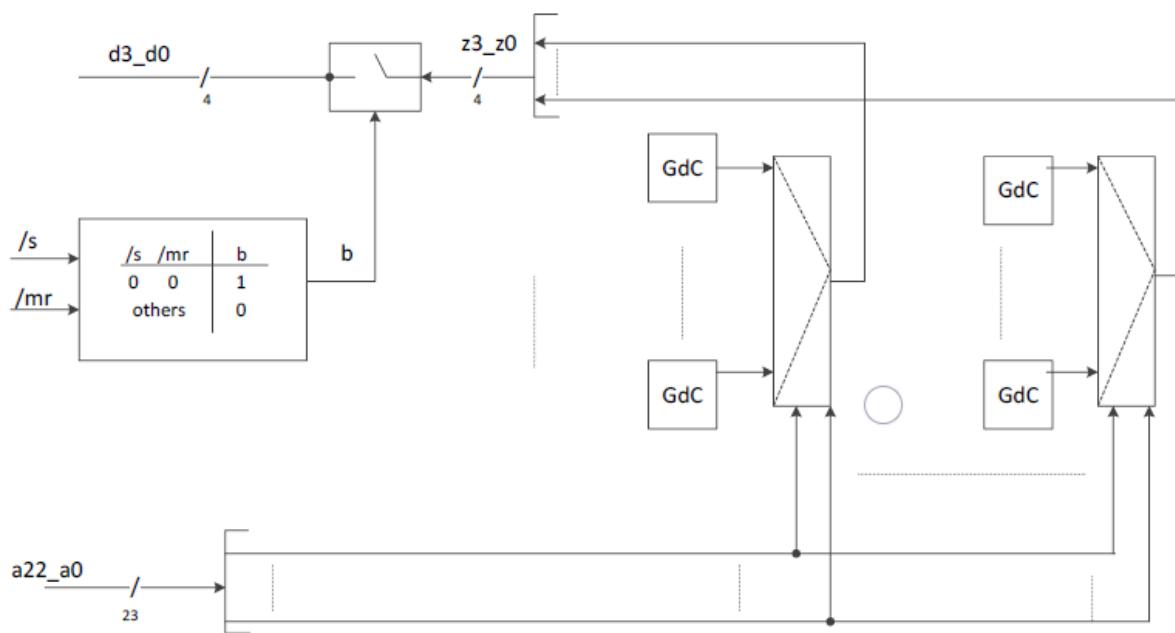
- Come ingresso l'indirizzo di una locazione di memoria

- Come uscita il contenuto della cella di memoria in ingresso



Ciascuna locazione di memoria delle ROM contiene valori **COSTANTI**, inseriti in modo INDELEBILE, infatti costituiscono la parte non volatile dello spazio di memoria.

Le ROM possono essere descritte per **semplificazione della RAM**:



Dove appunto, nelle locazioni di memoria sono presenti dei dispositivi che logicamente svolgono il ruolo di Generatori di Costante (*GdC*), al posto dei D-Latch.

Comunque il funzionamento è analogo a quello di un ciclo di lettura nella RAM.

Qui, come si è appena visto, i **fili di indirizzo**  $a_{22} \dots a_0$  **sono ingressi**, mentre i **fili di dato**  $d_3 \dots d_0$  **sono uscite!**

#### Note:

Due cose sono ovvie:

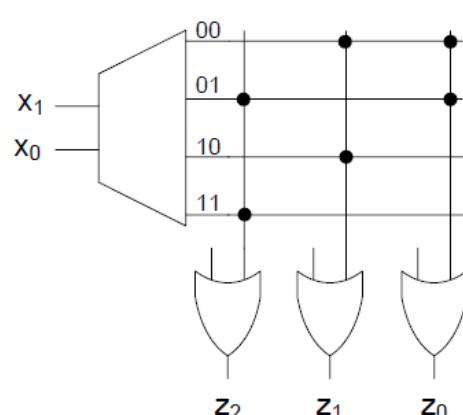
- Tutta la parte relativa ad operazioni di scrittura sparisce
- L'unica cosa che dovrà fare la RC che sintetizza  $b$  è vedere se sta avvenendo un ciclo di lettura.

### Sintesi di una memoria ROM

Una ROM può essere sintetizzata come un decoder  $N$ -to- $2^N$  seguita da una barriera di porte OR.

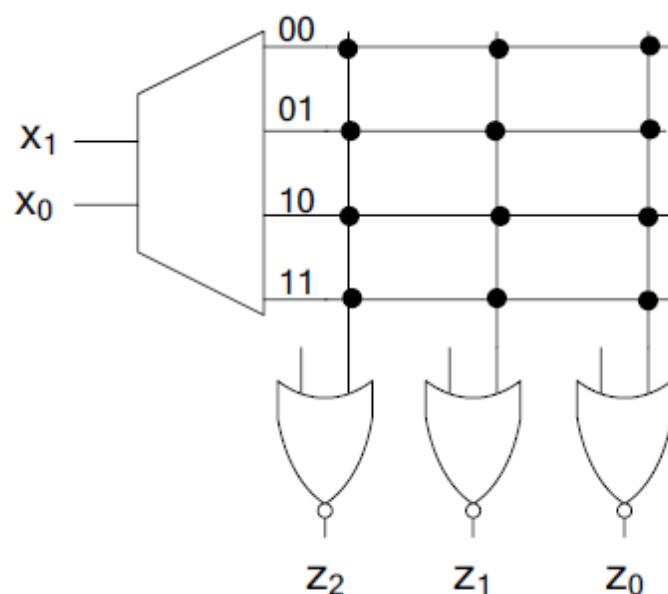
Il contenuto della ROM dipende da come si connettono le uscite del decoder alle porte OR, in particolare se c'è un collegamento il bit corrispondente vale 1, sennò 0 :

$X_1$	$X_0$	$Z_2$	$Z_1$	$Z_0$
0	0	0	1	1
0	1	1	0	1
1	0	0	1	0
1	1	1	0	0



## ROM Programmabili

Una ROM Programmabile è uno schema in cui vi sono tutti i contatti, e, VISTO CHE SI USANO TIPICAMENTE LE PORTE NOR, tutte le celle di memoria hanno al loro interno 0 (ovvero un pin attaccato visto che sono NOR):



Si possono poi bruciare i contatti che si vuole per mettere degli 1 a piacere.

**PROM (Programmable ROM)**: La matrice qui sopra è fatta da **fusibili**, che possono essere fatti saltare individualmente per inserire in ciascuna cella un 1, oppure lasciarli intatti per avere uno 0.

Questo chip viene venduto con tutti i fusibili intatti, e l'utente lo programma come vuole.

Il contro è che questa operazione è *one – time* perché i fusibili non si possono riparare.



**EPROM (Erasable Programmable ROM)**: Le connessioni sono fatte non più da fusibili, ma da **dispositivi elettronici (Field-Effect Transistors, FET)** che sono **programmabili per via elettrica e cancellabili tramite raggi UV** (infatti c'è un cerchietto di vetro per farle passare e colpire il chip).

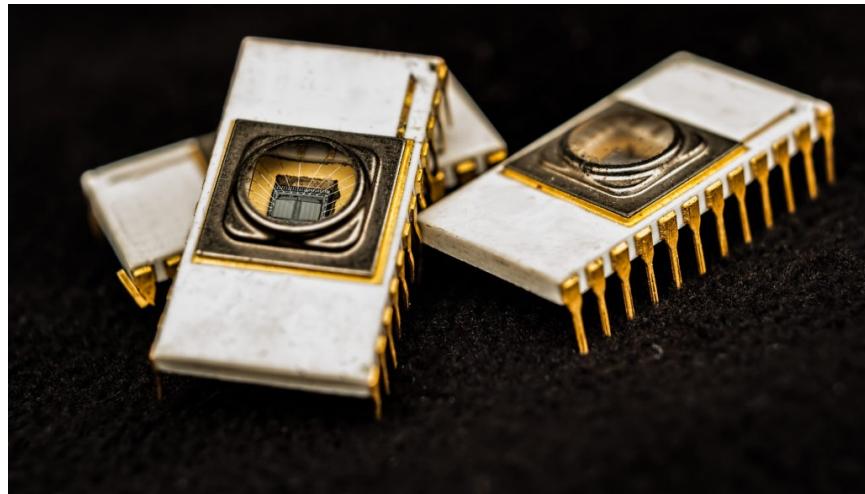
Quindi possono essere programmate, cancellate, riprogrammate, e così via.



**EEPROM (Electrically Erasable Programmable ROM)**: Possono essere **programmate e cancellate tramite segnali elettrici, quindi possono essere riprogrammate on – chip** (e.g. BIOS).

La EEPROM è un dispositivo programmabile, ma **la si chiama ancora ROM e non RAM perché:**

- Il tempo che ci vuole a riprogrammarla è molto maggiore del tempo che ci vuole a leggerla
- **Le tensioni non sono le stesse** della fase di normale operatività
- **La memoria è comunque non volatile**



# RSS Semplici

## Teoria

### Generalità sulle RSS

Le Reti Sequenziali Sincronizzate (RSS) si evolvono **SOLO** ad istanti di tempo ben precisi, detti **ISTANTI DI SINCRONIZZAZIONE**.



Quindi **NON** sono i cambiamenti dell'ingresso che fanno evolvere una RSS, ma l'arrivo del **SEGNALE DI SINCRONIZZAZIONE**, ovvero il **CLOCK**

Il Clock è un segnale di ingresso speciale, che ha una forma d'onda periodica squadrata di frequenza nota.



Si definisce **DUTY-CYCLE** il rapporto  $\frac{\tau}{T}$ , che deve essere sempre attorno al 50% (0.5).



L'evento che sincronizza qualsiasi rete che riceve il segnale di clock è il **FRONTE DI SALITA DEL CLOCK!**

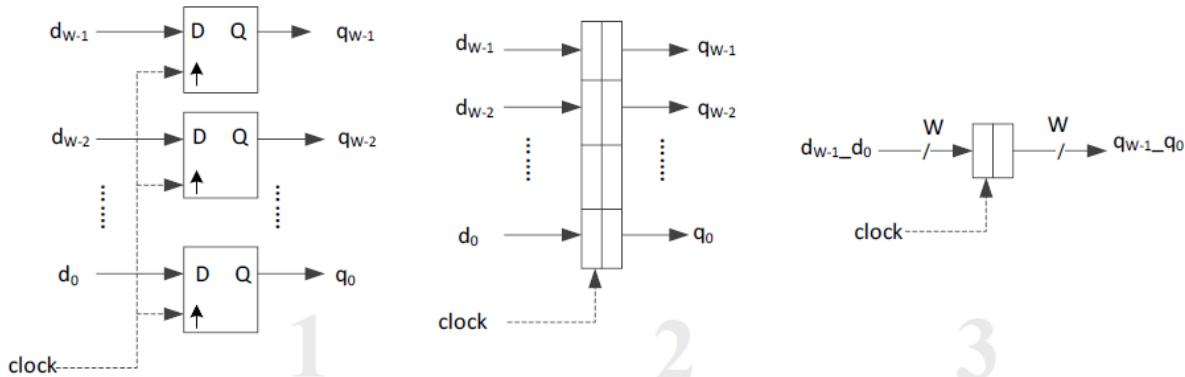


Le RSS si evolvono all'arrivo del **clock**, e non quando cambia un ingresso! Che il valore sia uguale o diverso dal precedente, la RSS lo ri-scrive! Per questo motivo **NON ESISTONO STATI STABILI NELLE RSS!**

## Registri, le reti fondamentali

Un **REGISTRO** a  $W$  bit è una **collezione (insieme)** di  $W$  D-Flip-Flop positive-edge triggered, i quali hanno:

- Gli **ingressi**  $d_i$  e le **uscite**  $q_i$  separati
- L'**ingresso**  $p$  a comune, a cui si collega il segnale di **clock**



Da sinistra a destra c'è un crescente livello di astrazione, quindi sempre meno dettagli.

La visione 1 è sostanzialmente la sintesi di un registro,

La 2 è la visione funzionale di una collezione di registri ad 1 bit,

La 3 è la visione funzionale di un registro a  $W$  bit.

L'ingresso del clock non si considera effettivamente come un ingresso, e lo si tralascia!

Per cui si dice che **un Registro a  $W$  bit ha  $W$  ingressi e  $W$  uscite!**

Nomenclatura:

- $W$  si chiama **CAPACITA' DEL REGISTRO**
- Lo **stato di uscita** del registro si chiama **CONTENUTO DEL REGISTRO**
- L'**utilizzo del contenuto del registro** si chiama **LETTURA DEL REGISTRO**
- La **memorizzazione dei  $W$  bit in ingresso** ad un certo istante di sincronizzazione si chiama **SCRITTURA DEL REGISTRO**

Per inizializzare un Registro ad un certo valore basta inizializzare ciascun D-FF al valore desiderato, collegando */reset* a */preset* o */preclear*, e l'altro a  $V_{cc}$

## Regole di pilotaggio di un Registro

Essendo costituiti da D-Flip-Flop, l'unico **REQUISITO DI PILOTAZIO DI UN REGISTRO** è che



**Gli ingressi  $d_i$  devono essere stabili da  $T_{setup}$  prima a  $T_{hold}$  dopo il fronte di salita del clock!**

### Similarità con i D-Flip-Flop:

- L'ingresso  $d$  deve comunque rimanere stabile da  $T_{setup}$  prima a  $T_{hold}$  dopo
- L'uscita  $q$  si adeguava comunque dopo  $T_{prop} > T_{hold}$

### Differenza con i D-Flip-Flop: (D-FF: RSA, Registri: RSS)

Nei **D-Flip-Flop** c'è la specifica condizione di non trasparenza  $T_{prop} > T_{hold}$ , visto che ogni volta che l'ingresso  $d$  cambiava, doveva cambiare pure  $p$ .

Nei **Registri** invece, visto che si memorizzano i cambiamenti degli ingressi solo al fronte di salita del clock, che invece è un segnale "fisso" e "prevedibile", basta solo che questo non sia troppo veloce (**disuguaglianze di temporizzazione**) e abbiamo già una condizione per la non trasparenza dei Registri!

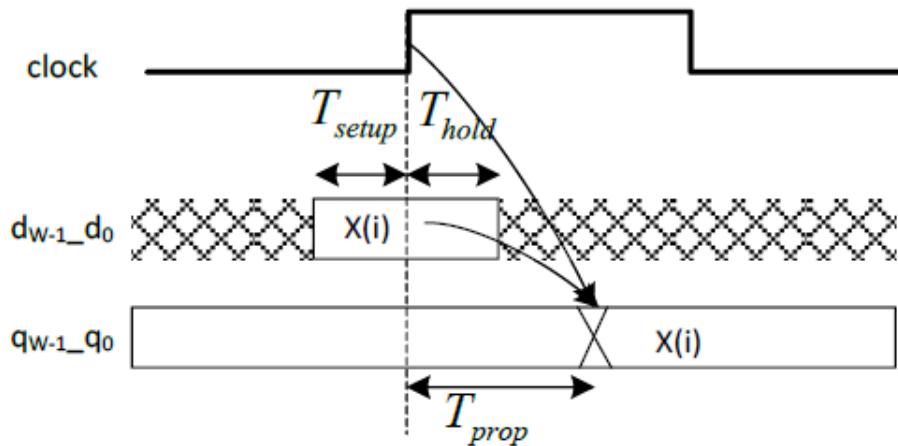


**C'è un cambio di paradigma dalle RSA alle RSS:**

- **Nelle RSA l'uscita CAMBIA quando CAMBIANO gli ingressi**
- **Nelle RSS l'uscita CAMBIA quando ARRIVA IL FRONTE DI SALITA DEL CLOCK!**

Mentre per le RSA, se io do in ingresso lo stesso valore 2 volte, la rete "non evolve", **le RSS invece ri-memorizzano COMUNQUE il nuovo valore presentato in ingresso, INDIPENDENTEMENTE dal fatto che sia uguale o diverso dal precedente!**  
Ciò che importa è che **arriva il fronte di salita del clock, quindi si memorizza!**  
**Se il valore è lo stesso, si riscrive, poco importa!**

Inoltre **le uscite restano costanti per il periodo  $T$  che va dal  $T_{prop}$  corrente al successivo  $T_{prop}$ !**



**Fra l'ingresso e l'uscita c'è un clock di differenza!**



**Nelle RSA l'uscita può cambiare ogni  $T_{setup} + T_{prop}$ , mentre nelle RSS sono vincolato dalle disuguaglianze di temporizzazione.**

#### RSS:

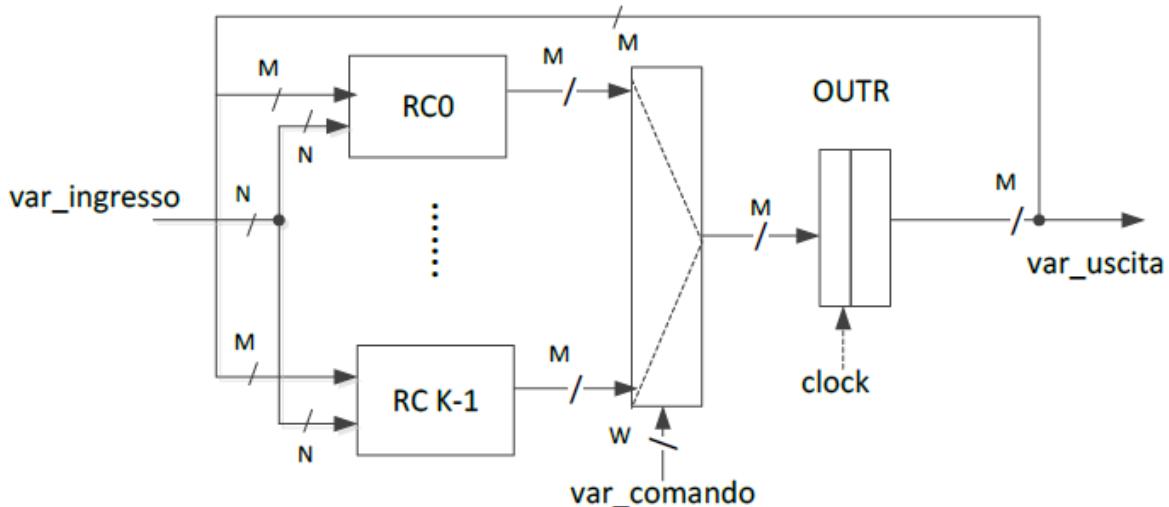
**Pro:** Si toglie la condizione di non trasparenza  $T_{prop} > T_{hold}$

**Con:** Si deve limitare la velocità del clock della rete per renderla non trasparente

## Registri Multifunzionali

**Un Registro Multifunzionale all'arrivo del clock memorizza, NEL REGISTRO STESSO, una fra  $K$  funzioni combinatorie possibili.**

Si realizza tramite un MUX a  $K$  ingressi, le RC che sintetizzano ciascuna una delle  $K$  funzioni combinatorie, ed un Registro per rendere il tutto una RSS:



Nella descrizione Verilog di questa rete ogni Rete Combinatoria  $RC_i$ ,  $i = 0 \dots K - 1$  sarà una *function*, e avrai una cosa del genere:

```

casex(var_comando)
  0: OUTR <= F_0(var_ingresso, OUTR)
  1: OUTR <= F_1(var_ingresso, OUTR)
  ...
  K-1: OUTR <= F_{K-1}(var_ingresso, OUTR)

  function [M-1:0] F_0;
    // DESCRIZIONE che implementa la Rete Combinatoria RC_0
  endfunction

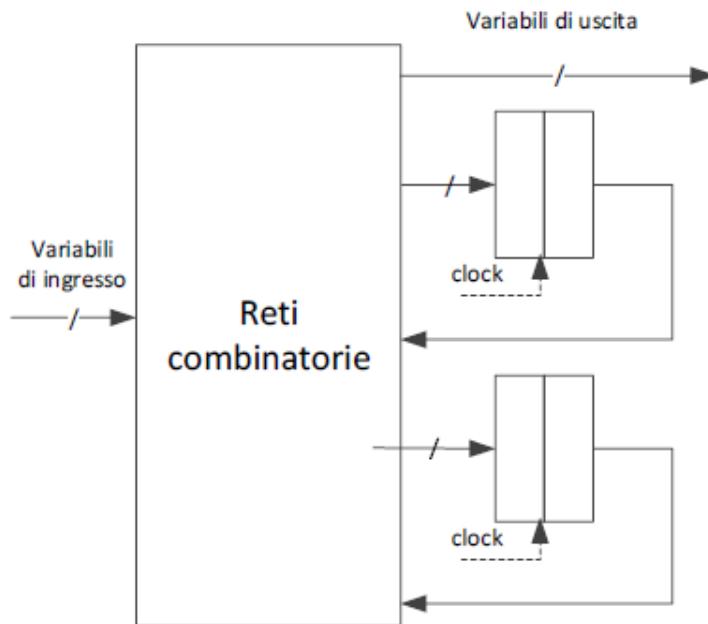
  function [M-1:0] F_1;
    // DESCRIZIONE che implementa la Rete Combinatoria RC_1
  endfunction

  ...
  function [M-1:0] F_{K-1};
    // DESCRIZIONE che implementa la Rete Combinatoria RC_{K-1}
  endfunction

```

## Definizione e temporizzazione delle RSS

Una RSS è un insieme di Registri e Reti Combinatorie, montati come si vuole, purché non ci siano anelli di Reti Combinatorie, e purché i Registri abbiano tutti lo stesso clock!



**REGOLA DI PILOTTAGGIO:** (la stessa dei Registri, qui detta in matematichese)



Sia  $t_i$  l' $i$ -esimo fronte di salita del clock, allora **LO STATO DI INGRESSO DEI REGISTRI DEVE ESSERE STABILE IN**  $[t_i - T_{setup}, t_i + T_{hold}] \quad \forall i$

Si dà un nome a vari **tempi di attraversamento delle più lunghe catene fatte di SOLE Reti Combinatorie**, che stanno fra:

- $T_{in\_to\_reg}$ : L'ingresso della Rete Totale e l'ingresso di un Registro
- $T_{reg\_to\_reg}$ : L'uscita di un Registro e l'ingresso di un Registro
- $T_{in\_to\_out}$ : L'ingresso della Rete Totale e l'uscita della Rete Totale
- $T_{reg\_to\_out}$ : L'uscita di un Registro e l'uscita della Rete Totale
- $T_{a\_monte}$ : Il tempo che ci mette la rete prima ("a monte") della mia a produrre un ingresso per la mia Rete Totale
- $T_{a\_valle}$ : Il tempo che ci mette la rete dopo ("a valle") la mia ad usare l'uscita che io gli ho fornito

## Disuguaglianze di Temporizzazione

1)	$T \geq T_{hold} + T_{a\_monte} + T_{in\_to\_reg} + T_{setup}$	IN → REG
2)	$T \geq T_{prop} + T_{reg\_to\_reg} + T_{setup}$	REG → REG
3)	$T \geq T_{hold} + T_{a\_monte} + T_{in\_to\_out} + T_{a\_valle}$	IN → OUT
4)	$T \geq T_{prop} + T_{reg\_to\_out} + T_{a\_valle}$	REG → OUT

### Spiegazione:

- Nei percorsi  $IN \rightarrow \dots$ 
  - Il primo è  $T_{hold}$  perché, considerando come istante 0 l'arrivo del fronte di salita del clock, io ricevo dalla rete a monte un ingresso.  
Allora se la rete a monte fosse molto veloce a fornirmi un altro ingresso, e questo quindi mi CAMBIASSE LO STATO D'INGRESSO DENTRO l'intervallo fra il fronte di salita e  $T_{hold}$ , starei violando la condizione di pilotaggio dei registri!
  - Il secondo termine è  $T_{a\_monte}$ , perché la rete a monte ci mette  $T_{a\_monte}$  per darmi un nuovo input
  - Il terzo è  $T_{IN\_to\_<\dots>}$ , e si vede banalmente
- Nei percorsi  $REG \rightarrow \dots$ 
  - Il primo termine è sempre  $T_{prop}$ , perché l'uscita di un Registro è buona dopo  $T_{prop}$  dal fronte di salita del clock
  - Il secondo è  $T_{REG\_to\_<\dots>}$ , e si vede banalmente
- Nei percorsi  $\dots \rightarrow REG$  l'ultimo termine è sempre  $T_{setup}$ , perché per via della condizione di pilotaggio di un Registro, l'ingresso deve essere stabile almeno  $T_{setup}$  prima del fronte di salita del clock
- Nei percorsi  $\dots \rightarrow OUT$  l'ultimo termine è sempre  $T_{a\_valle}$ , perché bisogna aspettare che la rete a valle abbia usato l'input che gli ho fornito, prima di mandargliene un altro!

### SOLO nei percorsi $IN \rightarrow \dots$ ci sono sempre 4 termini!

Restringendo la generalità di alcune di queste disuguaglianze si ottengono dei Modelli importanti:

- Se **non esiste una via combinatoria fra ingresso e uscita**, la 3 disuguaglianza scompare (si lascia la prima perché è più stringente, perché  $T_{setup} < T_{a\_valle}$ ), e si ha il **MODELLO DI MOORE**

- Se le uscite sono prese direttamente dai registri, senza reti combinatorie nel mezzo, nella disuguaglianza 4 scompare il termine  $T_{reg\_to\_out}$  e si ha il **MODELLO DI MEALY**

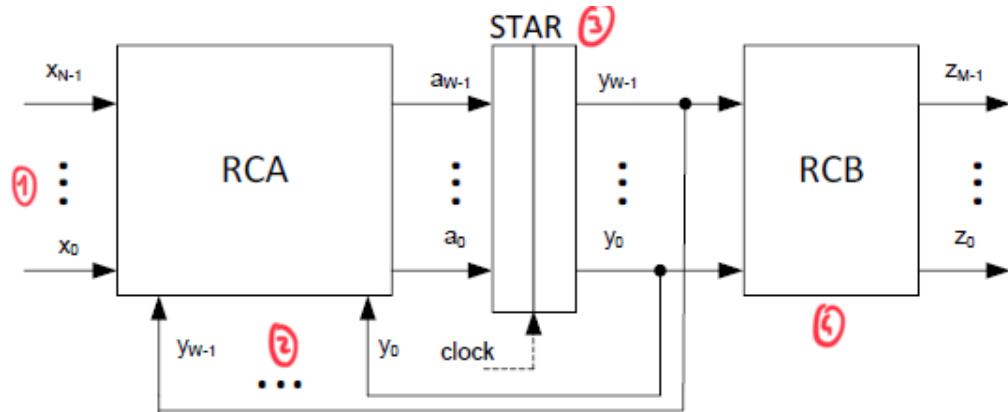
Vediamo questi modelli più nel dettaglio:

## Modello di Moore

Una RSS di Moore ha:

- **$N$  variabili logiche di ingresso**
- **$M$  variabili logiche di uscita**
- Un **MECCANISMO DI MARCATURA**, che **ad ogni istante ( $\forall t$ ) MARCA LO STATO INTERNO PRESENTE/MARCATO** scelto fra l'insieme degli Stati Interni possibili per quella rete
- **DUE LEGGI DI EVOLUZIONE NEL TEMPO:**
  - $A : X \times S \rightarrow S$ ,  $(StatoIngresso, StatoInterno) \rightarrow NuovoStatoInterno$
  - $B : S \rightarrow Z$ ,  $StatoInterno \rightarrow StatoUscita$

**Nota: NEL MODELLO DI MOORE è  $S \rightarrow Z$ , in ALTRI MODELLI sarà altro**
- Il **segnale di clock** come segnale di sincronizzazione
- Una **LEGGE DI TEMPORIZZAZIONE:**
  - Individua, tramite la legge  $A$ , il **NUOVO STATO INTERNO  $S' = A(X, S)$**
  - Attendi  $T_{prop}$  dal fronte di salita del clock
  - Tramite il Meccanismo di Marcatura, segna  $S'$  come **STATO INTERNO MARCATO  $\forall t$**
  - Individua, tramite la legge  $B$ , lo **STATO DI USCITA  $Z = B(S)$   $\forall t$**



Modello standard di sintesi di una rete di Moore

$$1. X = (x_{N-1} x_{N-2} \dots x_1 x_0)$$

$$2. S = (y_{W-1} y_{W-2} \dots y_1 y_0)$$

3. Il **Registro STAR** è il Registro che **contiene lo Stato Interno Marcato  $S$** .

Ovviamente va dimensionato in modo che possa contenere tutti i possibili Stati Interni.

**Le variabili  $a_{W-1} \dots a_0$  codificano il Nuovo Stato Interno  $S'$ , che verrà promosso a Stato Interno Presente  $S$  dopo  $T_{prop}$  dal fronte di salita del clock.**

4. Se la legge  $B$  fosse  $B : X \times S \rightarrow Z$  (come nel **Modello di Mealy**), avremmo che le variabili  $x_{N-1} \dots x_0$  sarebbero variabili di ingresso non solo per  $RCA$ , ma anche per  $RCB$ !

Questo significherebbe però che esiste una via combinatoria fra ingresso ed uscita, che è proprio l'ipotesi che abbiamo tolto quando abbiamo definito il Modello di Moore.

Quindi **nel Modello di Moore la legge  $B$  DEVE ESSERE  $B : S \rightarrow Z$ , cioè  $Z = X(S)$ .**

Una Rete di Moore si descrive specificando le due leggi,  $A$  e  $B$

### Disuguaglianze di Temporizzazione nel Modello di Moore

- |  |  |
|--|--|
| 1) $T \geq T_{hold} + T_{a\_monte} + T_A + T_{setup}$<br>2) $T \geq T_{prop} + T_A + T_{setup}$<br>3) $T \geq T_{prop} + T_B + T_{a\_valle}$ | IN $\rightarrow$ STAR<br>STAR $\rightarrow$ STAR<br>STAR $\rightarrow$ OUT |
|--|--|

Dove:

- $T_A$  è il Tempo di Attraversamento di  $RCA$
- $T_B$  è il Tempo di Attraversamento di  $RCB$

Queste 3 diseguaglianze sono le STESSE (ovviamente) di una RSS generale, con le seguenti sostituzioni:

1.  $T_{in\_to\_reg} \rightarrow T_A$
2.  $T_{reg\_to\_reg} \rightarrow T_A$
3. Questa non esiste più
4.  $T_{reg\_to\_out} \rightarrow T_B$

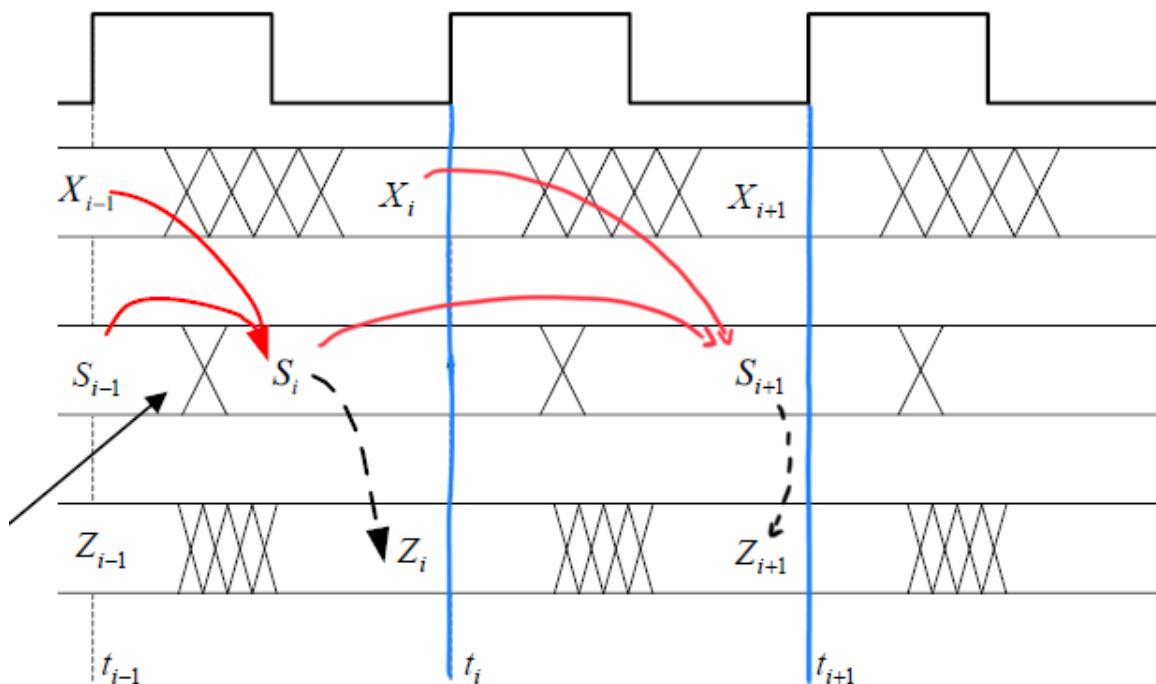


Diagramma di temporizzazione per una Rete di Moore

$$\begin{cases} S[t_{i+1}] = A(X[t_i], S[t_i]) & \text{Valida SOLO negli istanti di sincronizzazione } t_i \\ Z[t_i] = B(S[t_i]) & \text{Relazione combinatoria, valida } \forall t \end{cases}$$

Le frecce in **rosso** rappresentano la prima legge,  $A$ , infatti partono da due valori, PASSA UN CLOCK, e POI influenza il valore successivo dello stato interno marcato!

Le frecce in **nero** rappresentano la seconda legge,  $B$ , infatti parte da un valore, e influenza SUBITO (perché è combinatoria) il valore dell'uscita!



**L'uscita è in ritardo di un clock rispetto all'ingresso che la genera!**

La prima relazione coinvolge un Registro (STAR), perché  $S[t_{i+1}] \propto S[t_i]$ , e quando c'è un Registro di mezzo si perde un clock!

Attenzione che pure la seconda perde un clock! Perché  $Z[t_i] \propto S[t_i] \propto S[t_{i-1}]$ , quindi pure l'uscita è un clock in ritardo rispetto all'ingresso!

Possiamo dire che l'uscita  $Z[t_i]$  è **proporzionale alla STORIA DEGLI STATI DI INGRESSO della rete, fino a quello al tempo  $t_{i-1}$ .**

**Sostanzialmente si perde un clock per via della non trasparenza, ma questo permette di montare come si vuole le reti di Moore, senza problemi!**

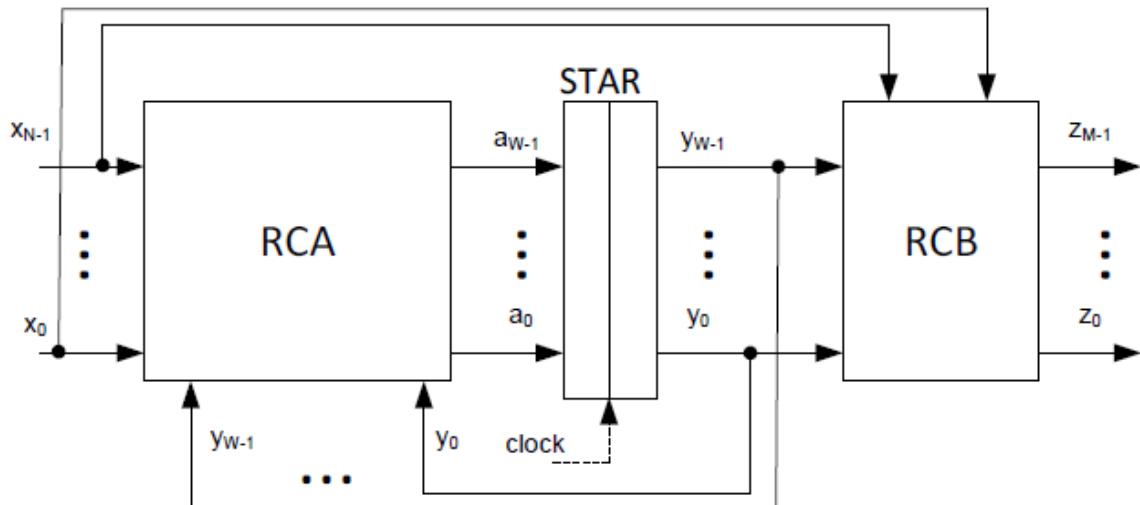
L'ingresso  $X[t_i]$  non può influire sull'uscita  $Z[t_i]$  se non c'è una via combinatoria fra ingresso ed uscita!

## Modello di Mealy

L'unica differenza con Modello di Moore è la seguente:

**Il dominio della legge  $B$  diventa uguale a quello della legge  $A$ :**

- $A : X \times S \rightarrow S, (StatoIngresso, StatoInterno) \rightarrow NuovoStatoInterno$
- $B : X \times S \rightarrow Z, (StatoIngresso, StatoInterno) \rightarrow StatoUscita$



Modello standard di sintesi di una Rete di Mealy

Dove appunto, visto che entrambe le leggi hanno come ingressi  $X \times S$ , entrambe avranno in input sia le variabili  $x_{N-1} \dots x_0$  sia  $y_{W-1} \dots y_0$

### Disuguaglianze di Temporizzazione nel Modello di Mealy

- |  |                         |
|--|-------------------------|
| 1) $T \geq T_{hold} + T_{a\_monte} + T_A + T_{setup}$    | IN $\rightarrow$ STAR   |
| 2) $T \geq T_{prop} + T_A + T_{setup}$                   | STAR $\rightarrow$ STAR |
| 3) $T \geq T_{hold} + T_{a\_monte} + T_B + T_{a\_valle}$ | IN $\rightarrow$ OUT    |
| 4) $T \geq T_{prop} + T_B + T_{a\_valle}$                | STAR $\rightarrow$ OUT  |

(Io ho fatto il preciso, ma puoi tranquillamente rimpiazzare sia  $T_A$  che  $T_B$  con  $T_{RC}$ )

Dove, la prima, la seconda e la quarta sono **IDENTICHE** a quelle del **Modello di Moore**!  
La terza è l'unica "nuova", che ovviamente "sfrutta il nuovo percorso combinatorio sopra".

Sostanzialmente **una volta ricavate le prime due, basta cambiare  $T_A$  con  $T_B$  e  $T_{setup}$  con  $T_{a\_valle}$  e hai gratis le seconde due!**

Le puoi anche ricavare da quelle generali sostituendo:

- $T_{in\_to\_reg}, T_{reg\_to\_reg} \rightarrow T_A$   
Cioè nelle prime due c'è  $T_A$  al posto di ogni  $T_{<\dots>_to_{<\dots>}}$
- $T_{in\_to\_out}, T_{reg\_to\_out} \rightarrow T_B$   
Cioè nelle ultime due c'è  $T_B$  al posto di ogni  $T_{<\dots>_to_{<\dots>}}$

Visto che nella terza equazione si sommano  $T_{a\_monte}$  e  $T_{a\_valle}$ , che sono i tempi più lunghi assieme a  $T_A$  e  $T_B$ , **il clock nelle reti di Mealy deve andare più lentamente di quello nelle Reti di Moore!**



**Il vantaggio delle Reti di Mealy è che non devo aspettare per forza il fronte di salita del clock per produrre un nuovo stato di uscita, ma visto che c'è una via combinatoria, basta che l'ingresso cambi per produrre una nuova uscita!**

**Di conseguenza il clock dovrà essere più lento di quello di una Rete di Moore!**

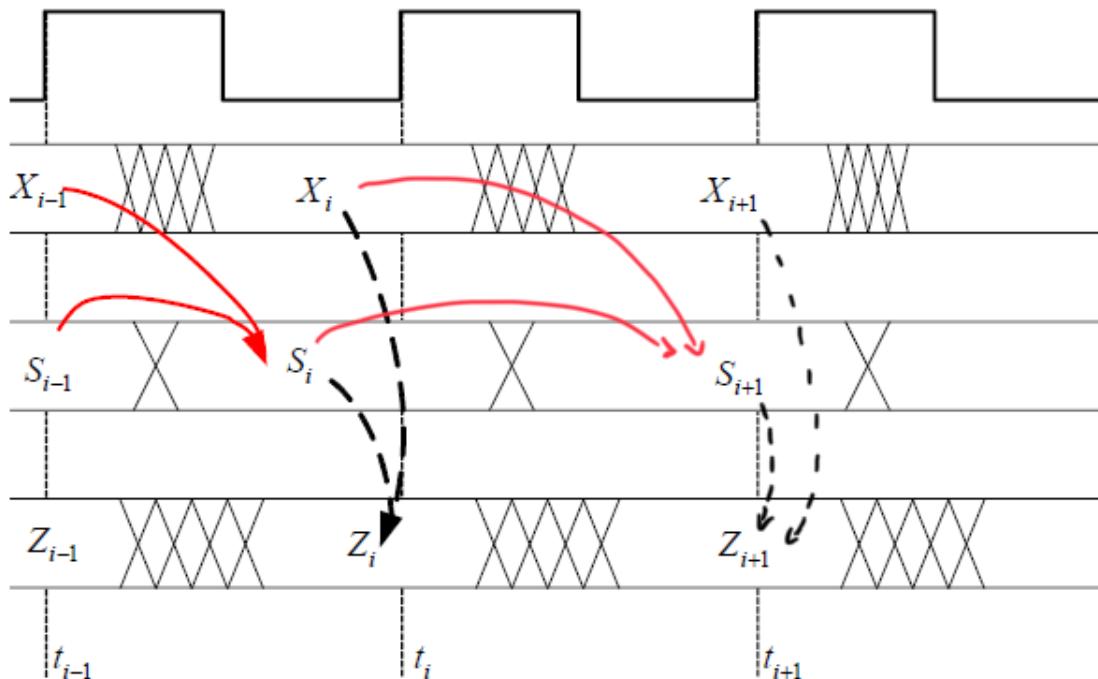


Diagramma di temporizzazione di una Rete di Mealy

$$\begin{cases} S[t_{i+1}] = A(X[t_i], S[t_i]) & \text{Valida SOLO negli istanti di sincronizzazione } t_i \\ Z[t_i] = B(\textcolor{blue}{X[t_i]}, S[t_i]) & \text{Relazione combinatoria, valida } \forall t \end{cases}$$

Ho segnato in **blu** l'UNICA differenza che c'è con le Leggi di Temporizzazione del Modello di Moore.

Nel diagramma di temporizzazione questo si tradurrebbe nell'eliminare le frecce  $X_i \Rightarrow Z_i$ ,  $X_{i+1} \Rightarrow Z_{i+1}$ !

Le frecce **rosse** rappresentano la prima legge, infatti ci mettono un clock per agire

Le frecce **nere** rappresentano la seconda legge, che qui, l'unica differenza col Modello di Moore è che ha semplicemente un ingresso in più: da  $B : S \rightarrow Z$  è diventata  $B : X \times S \rightarrow Z$ !

Si vede bene una cosa dal confronto dei diagrammi di temporizzazione:

- **Nelle Reti di Moore l'uscita è sempre un clock in ritardo rispetto agli ingressi**, perché non esiste via combinatoria fra ingresso e uscita, e quando c'è un Registro di mezzo si perde un clock per via della non trasparenza!

Infatti **l'uscita corrente dipende dal penultimo stato di ingresso ricevuto**:

$$Z[t_i] \propto S[t_i] \propto (X[t_{i-1}], S[t_{i-1}]) \propto (X[t_{i-1}], (X[t_{i-2}], S[t_{i-2}])) \propto \dots \\ \Rightarrow Z[t_i] \propto X[t_{i-1}], X[t_{i-2}], \dots X[t_1], X[t_0], S[t_0]$$

- **Nelle Reti di Mealy invece l'uscita è sempre in pari (in fase) con gli ingressi**, perché una via combinatoria fra ingresso e uscita stavolta c'è!

Infatti **l'uscita corrente dipende anche dallo stato di ingresso corrente**:

$$Z[t_i] \propto (\textcolor{blue}{X[t_i]}, S[t_i]) \propto (\textcolor{blue}{X[t_i]}, (X[t_{i-1}], S[t_{i-1}])) \propto (\textcolor{blue}{X[t_i]}, (X[t_{i-1}], (X[t_{i-2}], \dots \\ \Rightarrow Z[t_i] \propto \textcolor{blue}{X[t_i]}, X[t_{i-1}], X[t_{i-2}], \dots X[t_1], X[t_0], S[t_0]$$

Dove ancora, in **blu** ho segnato le differenze con l'espressione sopra per Moore



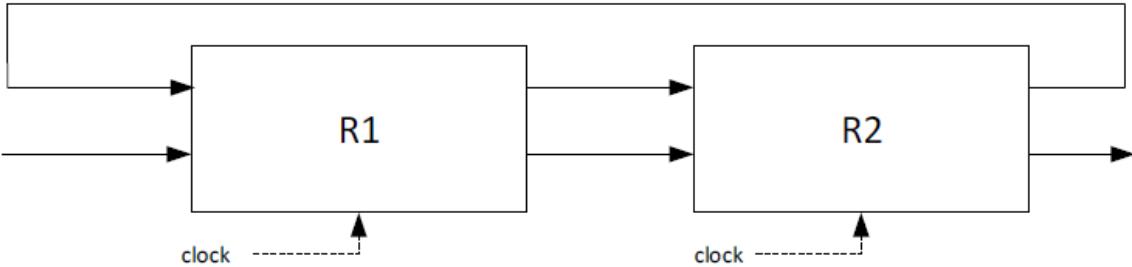
**Nelle Reti di Mealy si riesce a risolvere gli STESSI PROBLEMI che si risolvono con le Reti di Moore, ma usando meno Stati Interni perché la legge  $B$  è più flessibile!**

### Ma quindi è meglio Moore o è meglio Mealy?

L'uscita di una Rete di Mealy è "un clock in anticipo" rispetto a quella di una Rete di Moore.

D'altro canto il clock in una Rete di Mealy deve essere più lento rispetto ad una Rete di Moore.

Siamo alla pari, ma anche se non lo fossimo, la differenza fondamentale la fa la **TRASPARENZA**:



Per poter montare due RSS generiche in questo modo, ALMENO una deve essere una Rete di Moore! Sennò c'è un anello combinatorio e questa diventa una RSA (per via della relazione combinatoria  $B$ )!



**Le Reti di Moore sono NON TRASPARENTI**

**Le Reti di Mealy sono TRASPARENTI**

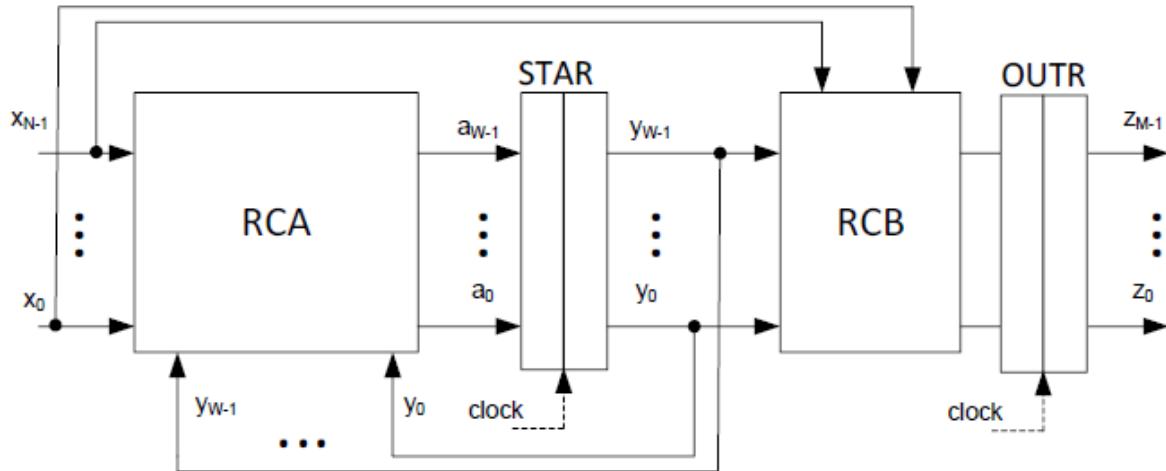
(Moore > Mealy)

## Modello di Mealy ritardato

Questo modello si chiama "ritardato" perché è una variazione del Modello di Mealy, in cui si aggiunge un registro in fondo, prima dell'uscita della rete.

Vista l'aggiunta di questo registro, il Modello di Mealy cambia così:

- Le uscite ora variano in maniera netta, all'arrivo del clock dopo un tempo  $T_{prop}$ , e rimangono stabili per tutto un periodo.  
Tutte caratteristiche ovvie, visto che c'è un Registro!
- Inoltre, importante, a differenza del Modello di Mealy, visto che si è aggiunto un registro, **LE USCITE DIVENTANO NON TRASPARENTI!**



Modello standard di sintesi di una Rete di Mealy ritardato

Sostanzialmente è tutto uguale al Modello di Mealy, l'unica cosa che cambia è la **LEGGE DI TEMPORIZZAZIONE**, dove ora al **clock** si eseguono le seguenti cose:

- Individuare SIA il Nuovo Stato Interno  $S' = A(X, S)$  SIA il Nuovo Stato di Uscita  $Z = B(X, S)$
- Attendere  $T_{prop}$  dal fronte di salita del clock
- Tramite il Meccanismo di Marcatura, segna SIA  $S'$  come **STATO INTERNO MARCATO** SIA  $Z$  come **NUOVO STATO DI USCITA**



**ATTENZIONE!** Lo Stato di uscita cambia DOPPO IL CLOCK, e il suo valore dipende dallo stato di ingresso e dallo stato interno marcato PRIMA DEL CLOCK!

### Disuguaglianze di Temporizzazione nel Modello di Mealy ritardato

- 1)  $T \geq T_{hold} + T_{a\_monte} + T_{RC} + T_{setup}$       IN  $\rightarrow$  REG
- 2)  $T \geq T_{prop} + T_{RC} + T_{setup}$                           REG  $\rightarrow$  REG
- 3)  $T \geq T_{prop} + T_{a\_valle}$                                   REG  $\rightarrow$  OUT

Dove, come nel Modello di Mealy, sono tutte uguali al Modello di Moore, con la differenza che la disequazione numero 3 del Modello di Mealy qui è scomparsa, e nella attuale 3 è scomparso pure il termine  $T_B(T_{RC})$ !

Le puoi ricavare da quelle del Modello di Moore, da quelle generali, in ogni caso sostituendo sia  $T_A$  che  $T_B$  con  $T_{RC}$  e togliendo il termine  $T_B$  nella terza!

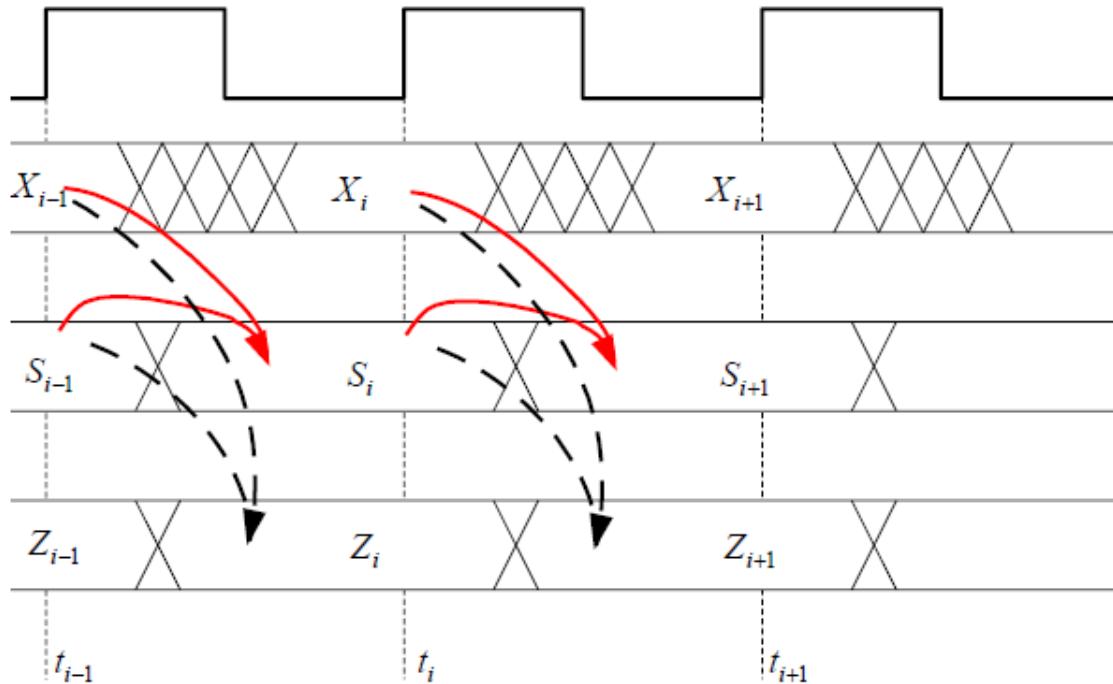


Diagramma di Temporizzazione di una Rete di Mealy ritardato

$$\begin{cases} S[t_{i+1}] = A(X[t_i], S[t_i]) & \text{Valida SOLO negli istanti di sincronizzazione } t_i \\ Z[t_{i+1}] = B(X[t_i], S[t_i]) & \text{Valida SOLO negli istanti di sincronizzazione } t_i \end{cases}$$

In **blu** ho segnato l'unica differenza (che però è importante!) col Modello di Mealy

**Dunque si possono montare Reti di Mealy ritardato in qualunque modo si vuole, visto che sono NON TRASPARENTI!**

Inoltre non sono rallentate da percorsi combinatori lunghi!

**Recap:**



RETI DI MOORE

⇒ NON TRASPARENTI ✓ ✓ ✓

RETI DI MEALY

⇒ TRASPARENTI ✗ ✗ ✗

RETI DI MEALY RITARDATO ⇒ NON TRASPARENTI ✓ ✓ ✓

## RSS Semplici fondamentali

### Contatori (Rete di Moore)

Vi sono 3 tipi di contatori:

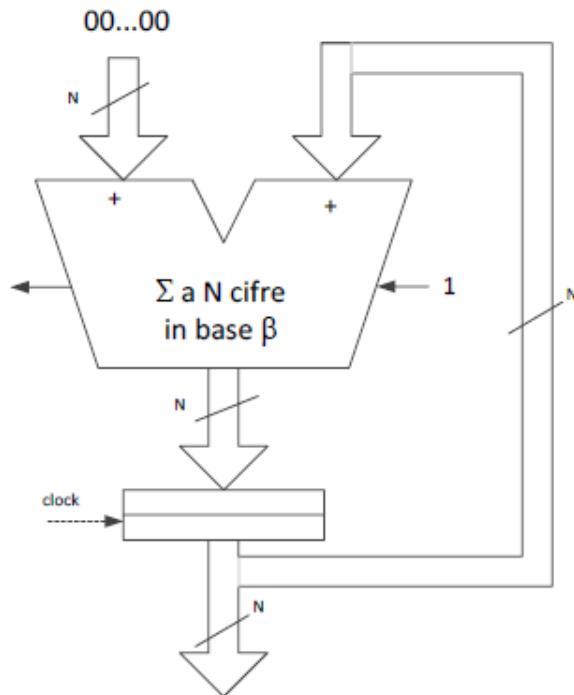
- **CONTATORE UP**: Ad ogni clock **incrementa di uno** (modulo  $\beta^n$ ) **il valore in uscita**
- **CONTATORE DOWN**: Ad ogni clock **decrementa di uno** (modulo  $\beta^n$ ) **il valore in uscita**
- **CONTATORE UP/DOWN**: Ad ogni clock **incrementa o decrementa di uno** (modulo  $\beta^n$ ) **il valore in uscita**, a seconda del valore della **variabile di comando**



Cosa conta un contatore? Conta il **NUMERO DI CLOCK TRASCORSI!**

### Contatore Up

Un **CONTATORE UP** si realizza con un **Sommatore** ed un **Registro**:



Circuito contatore in base  $\beta$  su  $n$  cifre

```

module ContatoreUp(numero, clock, reset_);
    input clock, reset_;
    output [W-1:0] numero;
    reg [W-1:0] OUTR; assign numero=OUTR;

    always @(reset_ == 0) #1 OUTR <= 0;
    always @(posedge clock) if(reset_ == 1) #3 OUTR <= Inc(OUTR);
    // se si lavora in base 2 non c'è bisogno di scrivere la funzione
    // qui sotto, ma basta scrivere "OUTR <= OUTR + 1;" oppure "OUTR <= numero + 1;"

    //funzione blueprint generica nel caso in cui si lavori in base beta
    function [W-1:0] Inc_Beta;
        input [W-1:0] numero;
        casex(numero)
            codifica_di_0: Inc_Beta = codifica_di_1;
            codifica_di_1: Inc_Beta = codifica_di_2;
            ...
            default: Inc_Beta = 'BXXXX...XXX;
        endcase
    endfunction

endmodule

```

## Contatore Down

Cambia molto poco:

- Nel disegno al posto di un sommatore ci va un sottrattore

- Nella descrizione verilog cambia un '+' in un '-', e ovviamente nel caso si lavori in base  $\beta \neq 2$  ci vuole una *function* che ad ogni codifica associ la codifica "più bassa"

```

module ContatoreDown(numero, clock, reset_);
    input clock, reset_;
    output [W-1:0] numero;
    reg [W-1:0] OUTR; assign numero=OUTR;

    always @(reset_ == 0) #1 OUTR <= 0;
    always @(posedge clock) if(reset_ == 1) #3 OUTR <= Dec(OUTR);
    // se si lavora in base 2 non c'è bisogno di scrivere la funzione
    // qui sotto, ma basta scrivere "OUTR <= OUTR - 1;" oppure "OUTR <= numero - 1;"

    //funzione blueprint generica nel caso in cui si lavori in base beta
    function [W-1:0] Dec_Beta;
        input [W-1:0] numero;
        casex(numero)
            codifica_di_9999: Dec_Beta = codifica_di_9998;
            codifica_di_9998: Dec_Beta = codifica_di_9997;
            ...
            default: Dec_Beta = 'BXXXX...XXX;
        endcase
        // qui ovviamente non ci vanno 9999, sarebbe istanziato in base 10 su 4 cifre,
        // ma ci va ovviamente il massimo numero in base beta su n cifre, cioè beta^n -1
        // quindi sarebbe:
        /*
            codifica_di_(b^n -1): Dec_Beta = codifica_di_(b^n-2);
            codifica_di_(b^n -2): Dec_Beta = codifica_di_(b^n-3);
            ...
            default: Dec_Beta = 'BXXXX...XXX;
        */
    endfunction

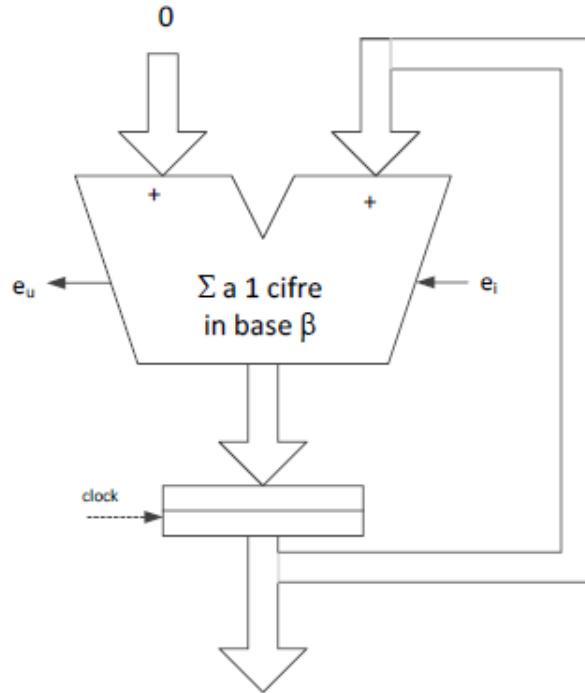
endmodule

```

## Contatore Up/Down con enabler

Un Contatore Up/Down dotato di un ingresso di abilitazione (enabler)  $e_i$ :

- $e_i = 1$ : all'arrivo del clock **CONTA UP** o **CONTA DOWN**
- $e_i = 0$ : all'arrivo del clock **CONSERVA L'ULTIMO VALORE**



Circuito contatore UP con enabler

(Se dovesse contare down dovrei sostituire il sommatore con un sottrattore)

```

module ContatoreUpEnabler(numero, clock, reset_, ei);
    input clock, reset_, ei;
    output [W-1:0] numero;
    reg [W-1:0] OUTR; assign numero=OUTR;

    always @(reset_ == 0) #1 OUTR <= 0;
    always @(posedge clock) if(reset_ == 1) #3
        casex(ei)
            0: OUTR <= OUTR;      // se ei = 0 conserva l'ultimo valore
            1: OUTR <= F(OUTR);   // se ei = 1 CONTA,
                                // se conti UP allora F = Inc_Beta,
                                // se invece conti DOWN allora F = Dec_Beta
        endcase

    // se si lavora in base 2 non c'è bisogno di scrivere la funzione
    // qui sotto, ma basta scrivere:
    // "OUTR <= OUTR + {W-1'B0, ei};" se si conta UP
    // "OUTR <= OUTR - {W-1'B0, ei};" se si conta DOWN
    // dove {W-1'B0, ei} è la concatenazione di W-1 zeri, e un 1 alla fine come LSB

    // funzione blueprint generica nel caso in cui si lavori in base beta
    // e si conti UP
    function [W-1:0] Inc_Beta;
        input [W-1:0] numero;
        casex(numero)
            codifica_di_0: Inc_Beta = codifica_di_1;
            codifica_di_1: Inc_Beta = codifica_di_2;

```

```

...
    default: Inc_Beta = 'BXXXX...XXX;
endcase
endfunction

// funzione blueprint generica nel caso in cui si lavori in base beta
// e si conti DOWN
function [W-1:0] Dec_Beta;
    input [W-1:0] numero;
    casex(numero)
        codifica_di_9999: Dec_Beta = codifica_di_9998;
        codifica_di_9998: Dec_Beta = codifica_di_9997;
        ...
        default: Dec_Beta = 'BXXXX...XXX;
    endcase
    // qui ovviamente non ci vanno 9999, sarebbe istanziato in base 10 su 4 cifre,
    // ma ci va ovviamente il massimo numero in base beta su n cifre, cioè beta^n -1
    // quindi sarebbe:
    /*
        codifica_di_(b^n -1): Dec_Beta = codifica_di_(b^n-2);
        codifica_di_(b^n -2): Dec_Beta = codifica_di_(b^n-3);
        ...
        default: Dec_Beta = 'BXXXX...XXX;
    */
endfunction

endmodule

```

Visto però che ogni sommatore lo posso scomporre secondo il montaggio a ripple carry, se fisso la base  $\beta = 2$ , la sintesi diventa quella di un incrementatore ad una cifra in base  $\beta = 2$ , dove però  $C_{in} \rightarrow e_i$ ,  $C_{out} \rightarrow e_u$ ,  $X_i \rightarrow q$ ,  $S_i \rightarrow a$ :

```

module Contatore1CifraB2(eu, ei, q, clock, reset_);
    input clock, reset_, ei;
    output eu, q;
    reg OUTR; assign q = OUTR;

    wire a; // a è la variabile di uscita dall'incrementatore che poi entra in OUTR

    assign {a, eu} = ( {q, ei} == 'B00 ) ? 'B00 :
                    ( {q, ei} == 'B01 ) ? 'B10 : // a = q XOR ei
                    ( {q, ei} == 'B10 ) ? 'B10 : // eu = q AND ei
                    ( {q, ei} == 'B11 ) ? 'B01;

    // è la tabella di verità di un contatore

    // al posto di così lo potevi pure scrivere in quest'altro modo:
    // assign a = (q != ei) ? 1 : 0;
    // assign eu = ( q == ei == 1 ) ? 1 : 0;

    // ATTENZIONE CHE QUESTA E' UNA DESCRIZIONE, NON UNA SINTESI!
    // QUI NON POTEVI SCRIVERE "assign a = q ^ ei; assign eu = q & ei;",
    // QUELLO LO SCRIVERAI POI NEL FILE DELLA SINTESI!

```

```

always @(reset_ == 0) #1 OUTR <= 0;
always @(posedge clock) if(reset_ == 1) #3 OUTR <= a;
// in questo caso c'è solo una legge combinatoria da dare alla rete!

endmodule

```

Attenzione alla descrizione!

Se volessi realizzare un contatore in base  $\beta = 3$ , avrei questo:

$q_1$	$q_0$	$e_i$	$a_1$	$a_0$	$e_u$
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	0	0	1
others			-	-	-

Dove, nell'ultima riga  $(q_1 q_0)_3 = (2)_3$ , e visto che  $e_i = 1$  lo dovrei incrementare, ma visto che i numeri in base 3 vanno fino a 2, la colonna di destra va incrementata

MODULO  $\beta^n$ !

Quindi verrebbe  $(a_1 a_0) = |3|_{3^2} = 0$ , con riporto uscente 1.

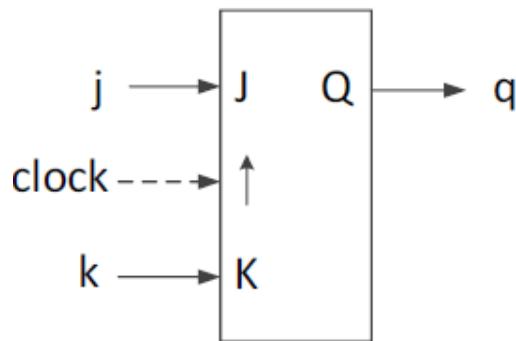
Notare che non puoi dire  $a_1 = q_1 \oplus e_i$ ,  $a_0 = q_0 \oplus e_i$ ,  $e_u = ???$

**Tutta la difficoltà di sintetizzare questa rete sta nel fare a modo la tabella di verità!**

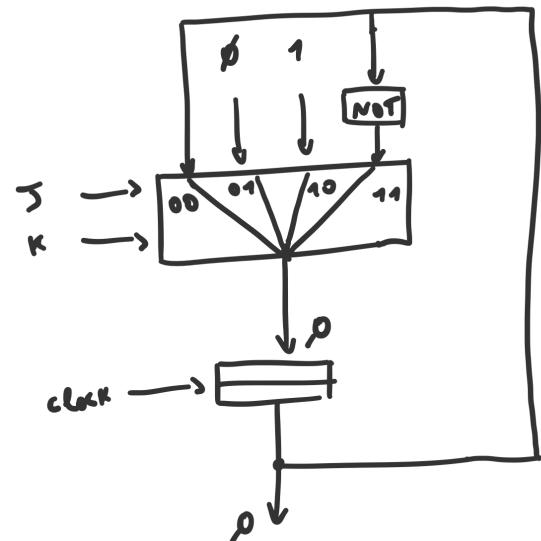
I contatori sono usati in pratica per dividere in frequenza un clock, e generarne un altro che va più lento: il MSD di un contatore ad  $N$  cifre in base  $\beta$  che riceve clock a periodo  $T$  corrisponde ad un clock di periodo  $\beta^N \cdot T$ !

## Flip-Flop JK (Rete di Moore)

E' possibile vedere il Flip-Flop JK (FF JK) come Registro Multifunzionale ad 1 bit:



Visione funzionale di un FF JK



Circuito che implementa un FF JK come Registro Multifunzionale ad 1 bit

Il FF JK è una RSS a due ingressi ed un'uscita, che all'arrivo del clock, si comporta come segue, seguendo gli ingressi  $j$  e  $k$ :

$j$	$k$	Azione
0	0	Conserva
0	1	Resetta
1	0	Setta
1	1	Commuta

Quindi sostanzialmente è un Latch SR, ma sincronizzato, in cui si può dare l'ingresso  $(1, 1)$ , il quale esegue l'azione di negare l'ingresso e darlo in uscita.

Diamone la **tabella di applicazione**:

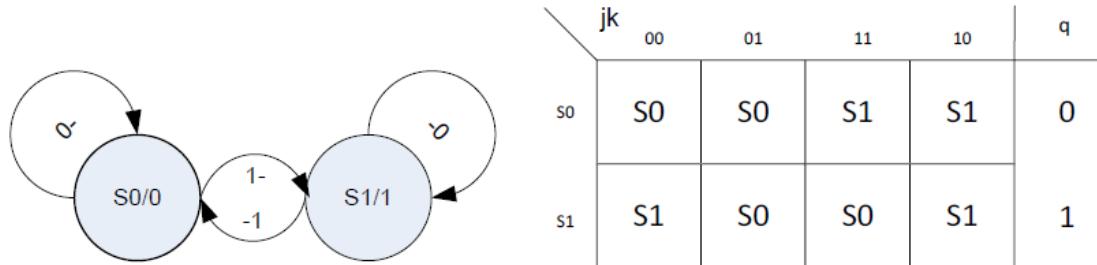
$q$	$q'$	$j$	$k$
0	0	0	—
0	1	1	—
1	0	—	1
1	1	—	0

Ora, visto che il Latch SR memorizza un bit, anche il FF JK memorizza un bit.

Quindi questa rete avrà 2 Stati Interni:  $S_0$ , nel quale memorizza 0, e  $S_1$ , nel quale memorizza 1.

**Se CODIFICO lo stato  $S_0$  con il bit 0, e lo stato  $S_1$  con il bit 1, la RC RCB diventa un cortocircuito! Perché il valore dell'uscita  $Z$  corrisponde alla codifica scelta per  $S$ ! ( $B : S \rightarrow Z$ )**

Diamo quindi il grafo/tabella di flusso di questa rete:



Grafo di flusso e tabella di flusso di un FF JK

```
module FFJK(q, j, k, clock, reset_);
  input j, k, clock, reset_;
  output q;
  reg STAR; parameter S0='B0, S1='B1;
  // quando si descrive una rete con degli stati interni, bisogna scegliere
  // una codifica degli stati interni in termini di bit, anche se è roba da sintesi!
  assign q = (STAR == S0) ? 0 : 1;

  always @(reset_ == 0) #1 STAR <= S0;
  always @(posedge clock) if(reset_ == 1) #3
    casex(STAR)
      S0: STAR <= (j == 0) ? S0 : S1;
      S1: STAR <= (k == 0) ? S1 : S0;
    endcase
endmodule
```

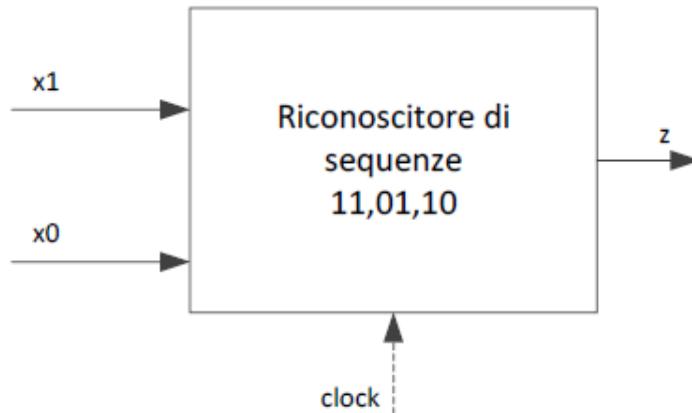


**Per scegliere la codifica degli Stati Interni in Verilog, guarda cosa deve fare RCB, e scegli la codifica in modo da semplificare la descrizione in più possibile!**

**Il massimo che puoi raggiungere è che RCB diventa un cortocircuito!**

Per la sintesi basta sinterizzare le Reti Combinatorie  $RCA$  e  $RCB$

## Riconoscitore di sequenze [11, 01, 10]



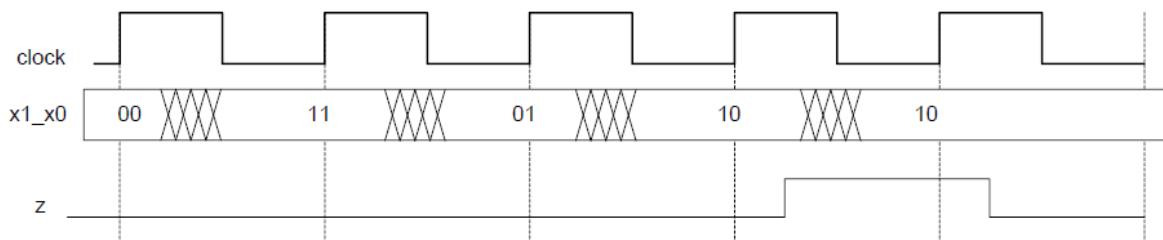
Visione funzionale del riconoscitore di sequenze per sequenze a 2 cifre

Un Riconoscitore di sequenza (**ognuno riconosce una specifica sequenza, hardcoded nella descrizione, non tutte le possibili sequenze esistenti!**) ha  $N$  ingressi, tanti quanti il numero di bit da cui è composto ciascun passo della sequenza, e un'uscita.

### Riconoscitore di sequenze come Rete di Moore

La descrizione del Riconoscitore di sequenze è la seguente:

L'uscita è 1 solo quando si è presentata, in clock consecutivi, la sequenza degli stati di ingresso voluta, ed è 0 altrimenti:



Notare che l'uscita sta ad 1 solo quando si riceve l'ultimo passo della sequenza, e in ogni caso sta ad 1 per un singolo clock, sia che nel prossimo clock ci sia ancora lo stesso "passo finale", sia che ci sia il "passo iniziale" della sequenza!

Questa rete deve memorizzare il **NUMERO DI PASSI DELLA SEQUENZA CORRETTI (CONSECUTIVI)** visti finora. Dunque la rete avrà bisogno di  $K + 1$  Stati Interni, cioè da 0 a  $K$  passi riconosciuti. Di questi Stati Interni, solo l'ultimo avrà in uscita 1 , mentre gli altri 0.

Analizziamo gli Stati Interni, in questo esempio abbiamo bisogno di 4 Stati Interni:

- $S_0$ : **STATO INIZIALE, USCITA=0**

Lo Stato in cui **non è ancora iniziata nessuna sequenza corretta**.

Non vi esco finché non arriva in input il **Primo Passo della Sequenza (11)** in questo caso)

- $S_1$ : **STATO INTERMIO 1, USCITA=0**

◦ Se lo stato di ingresso è il **Passo Successivo della Sequenza (01)**, passo a  $S_2$

◦ Se lo stato di ingresso è (di nuovo) il **Primo Passo della Sequenza (11)**, sto in  $S_1$

◦ Se lo stato di ingresso non è nessuno dei precedenti qui sopra, torno in  $S_0$

- $S_2$ : **STATO INTERMIO 2, USCITA=0**

◦ Se lo stato di ingresso è il **Passo Successivo della Sequenza (10)**, la sequenza è terminata, e quindi devo andare nello "**Stato Finale**" ( $S_3$ ) che ha uscita 1

◦ Se lo stato di ingresso è il **Primo Passo della Sequenza (11)**, torno in  $S_1$

◦ Se lo stato di ingresso non è nessuno dei precedenti qui sopra, torno in  $S_0$

- $S_3$ : **STATO FINALE, USCITA=1**

◦ Se lo stato di ingresso è il **Primo Passo della Sequenza (11)**, torno in  $S_1$

◦ Se lo stato di ingresso non è il **Primo Passo della Sequenza**, torno in  $S_0$

Chiaramente una sequenza di  $N$  stati si deve presentare in  $N$  clock, perché questa rete funziona correttamente. Se si presentano troppo velocemente me ne perdo qualcuno, se si presentano troppo lentamente qualcuno viene contato "doppio".

**Si può fare una versione alternativa di questa rete, in cui ciascun passo della sequenza può essere tenuto in ingresso un numero arbitrario di clock!**

L'unica modifica richiesta è che, per ciascun stato (che già non l'abbia!), se si riceve lo stato di ingresso con cui siamo saltati ad esso dallo stato precedente, si deve restare dove siamo!

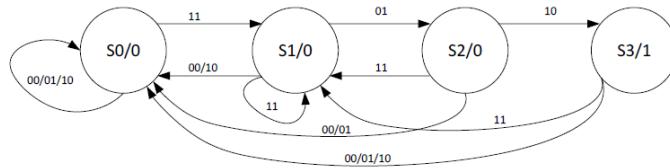
**Esempio:**

$S_0$ : non importa fare nulla

$S_1$ : ce l'ha già, riconosce 11 e sta in  $S_1$  se lo riceve

$S_2$ : Bisogna riconoscere che se arriva 01, mentre stiamo in  $S_2$ , bisogna continuare a stare in  $S_2$ !

$S_3$ : Se arriva 10, bisogna continuare a stare in  $S_3$  (con uscita=1), quindi si tiene l'uscita ad 1 per il numero di clock pari al numero di '10' arrivati consecutivamente!



Grafo di flusso per il Riconoscitore della sequenza '11-01-10'

$x_1x_0$	00	01	11	10	$z$
$S_0$	$S_0$	$S_0$	$S_1$	$S_0$	0
$S_1$	$S_0$	$S_2$	$S_1$	$S_0$	0
$S_2$	$S_0$	$S_0$	$S_1$	$S_3$	0
$S_3$	$S_0$	$S_0$	$S_1$	$S_0$	1

Tabella di flusso per il  
Riconoscitore della sequenza '11-  
01-10'

```

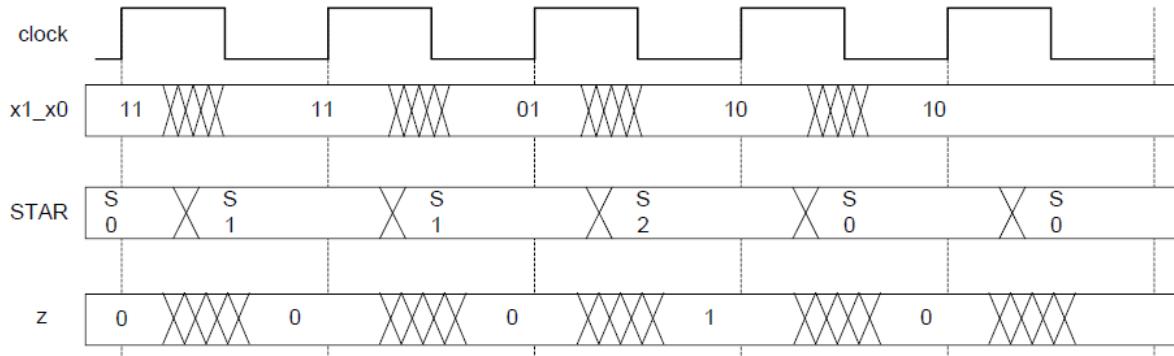
module RicSeqMoore(x1_x0, z, clock, reset_);
  input clock, reset_;
  input [1:0] x1_x0;
  output z;

  reg [1:0] STAR; parameter S0='B00, S1='B01, S2='B10, S3='B11;
  assign z = (STAR == S3) ? 1 : 0;

  always @(reset_ == 0) #1 STAR <= S0;
  always @(posedge clock) if(reset_ == 1) #3
    casex(STAR)
      S0: STAR <= (x1_x0 == 'B11) ? S1 : S0;
      S1: STAR <= (x1_x0 == 'B01) ? S2 : (x1_x0 == 'B11) ? S1 : S0;
      S2: STAR <= (x1_x0 == 'B10) ? S3 : (x1_x0 == 'B11) ? S1 : S0;
      S3: STAR <= (x1_x0 == 'B11) ? S1 : S0;
    endcase
  
```

## Riconoscitore di sequenze come Rete di Mealy

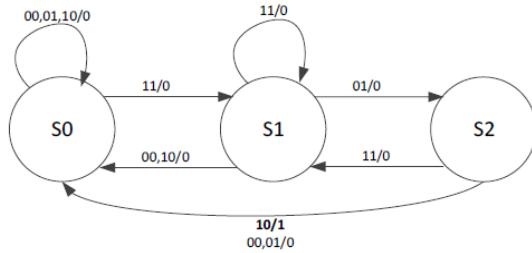
Questa è il diagramma di temporizzazione se si decide di usare il Modello di Mealy per descrivere questa rete:



Notare come stavolta non ci sia bisogno di passare ad un quarto stato  $S_3$  (e quindi usare un altro clock) con l'unico scopo di mettere l'uscita ad 1 (ne avevamo bisogno perché  $B : S \rightarrow Z$ ), ma si può direttamente settare l'uscita quando si verificano entrambe queste condizioni:

- Siamo nello **STATO FINALE** ( $S_2$ )
- Arriva in ingresso il passo finale della sequenza (10)

Questo è possibile perché ora la legge  $B$  è  $B : X \times S \rightarrow Z$ , quindi posso fare questo Grafo di Flusso, dove ad ogni freccia che collega Stato Interni, si ha pure l'uscita corrispondente:



Sintassi: <Passi per i quali si passa di stato>/<Valore dell'uscita quando si fa>

	X <sub>1</sub> X <sub>0</sub>			
	00	01	11	10
S0	S0/0	S0/0	S1/0	S0/0
S1	S0/0	S2/0	S1/0	S0/0
S2	S0/0	S0/0	S1/0	S0/1

Notare come:

```
module RicSeqMealy(x1_x0, z, clock, reset_);
  input clock, reset_;
  input [1:0] x1_x0;
  output z;

  reg [1:0] STAR; parameter S0='B00, S1='B01, S2='B10;
  assign z = ((STAR == S2) && (x1_x0 == 'B10)) ? 1 : 0;

  always @ (reset_ == 0) #1 STAR <= S0;
```

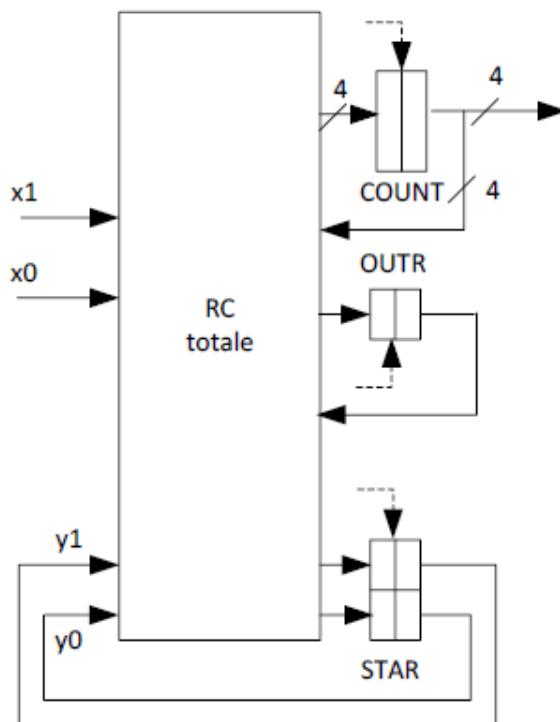
```
always @(posedge clock) if(reset_ == 1) #3
casex(STAR)
S0: STAR <= (x1_x0 == 'B11) ? S1 : S0;
S1: STAR <= (x1_x0 == 'B01) ? S2 : (x1_x0 == 'B11) ? S1 : S0;
S2: STAR <= (x1_x0 == 'B11) ? S1 : S0;
endcase
endmodule
```

# RSS Complesse

## Teoria

### Generalità

Per sintetizzare più facilmente RSS complesse, si adotta un nuovo Modello, prendendo come base il Modello di Mealy Ritardato:



Immaginare che in generale i fili siano di più

Questo modello ha le seguenti caratteristiche:

- C'è un **REGISTRO DI STATO STAR**
- Gli altri Registri vengono chiamati **REGISTRI OPERATIVI**, e possono essere quanti voglio
- **Posso mandare l'uscita di TUTTI i Registri, sia STAR che quelli Operativi, in ingresso a Reti Combinatorie, la cui uscita andrà in ingresso a dei Registri, e così via...**
- Le uscite sono tutte sostenute da **Registri Operativi**



**TUTTI i Registri, sia STAR che quelli Operativi, vengono aggiornati CONTEMPORANEAMENTE all'arrivo del fronte di salita del clock!**

Le disuguaglianze di temporizzazione di questo Modello Generalizzato sono LE STESSE del Modello di Mealy ritardato!



**Quando un Registro compare a destra di un assegnamento procedurale ( $<=$ ), ci si riferisce al valore che aveva PRIMA DEL FRONTE DI SALITA DEL CLOCK!**

**Il fatto che un Registro può stare a destra di un assegnamento procedurale ( $<=$ ) è perché ha un filo di retroazione (come OUTR e COUNT in figura) che rientra nella rete totale! Se non ci fosse questo filo non potrei nemmeno scrivere  $OUTR <= OUTR$ !**

## Linguaggio di trasferimento tra registri

Una descrizione fatta secondo il modello qui sopra si dice **A LIVELLO DI LINGUAGGIO DI TRASFERIMENTO TRA REGISTRI.**

Ogni ramo del *casex* si chiama **STATEMENT**, e comprende:

- 0+  **$\mu$ -ISTRUZIONI**, cioè **assegnamenti a Registri Operativi**
- 1  **$\mu$ -SALTO**, cioè un **assegnamento al Registro STAR**.

Un  $\mu$ -salto può essere:

- Ad una via:  $S_0 : STAR <= S_1 \iff (STAR == S_0) ? S_1 : S_1$
- A due vie:  $(STAR == S_0) ? S_1 : S_0$
- A più vie:  $(STAR == S_0) ? S_1 : (STAR == S_1) ? S_2 : S_0$

Se ho  $N$  Registri Operativi  $RO_i$ , uno **Statement** è questo:

```
S_k: // un qualsiasi Stato Interno, non importa quale
begin
    RO1 <= /* espressione */;
    RO2 <= /* espressione */;
    ...
    RO_N <= /* espressione */;
```

```
STAR <= /* espressione */;  
end
```



**ATTENZIONE!** Omettere una  $\mu$ -istruzione corrisponde a scrivere  
 $RO_j <= RO_j!$

## Handshake /dav-rfd

Gli handshake servono a far comunicare due RSS che hanno clock diversi! (non in fase)



**IN QUASI TUTTI GLI ESAMI C'E' UN HANDSHAKE!**



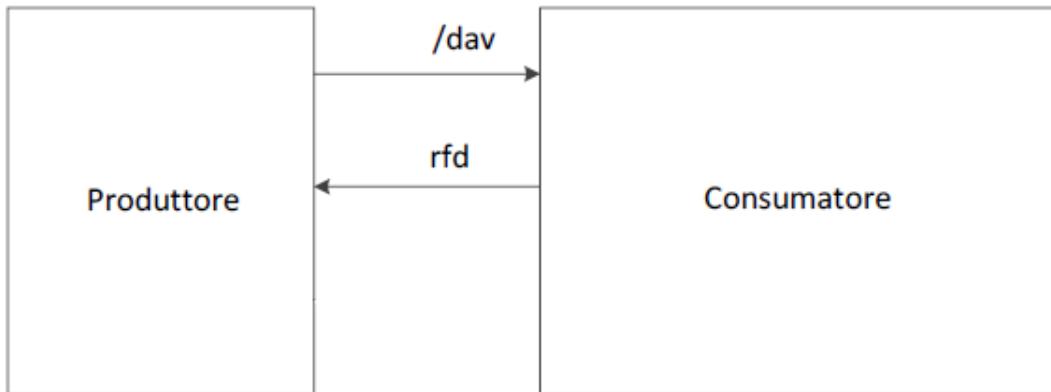
**/dav e rfd hanno TRANSIZIONI ALTERNATE in modo da convalidarsi a vicenda!**

(Perché non posso fare ipotesi sulla velocità dell'altra rete!)

L'handshake si fa dotando le due reti che devono comunicare, chiamate **PRODUTTORE** e **CONSUMATORE**, di due fili (variabili logiche) aggiuntivi:

- **/dav: DATA VALID (attivo basso)**
  - $/dav = 0 \rightarrow$  **CI SONO DATI NUOVI DA PRELEVARE**
  - $/dav = 1 \rightarrow$  **NON CI SONO DATI NUOVI, E SE LI PRELEVI SONO SBAGLIATI!**
- **rfd: READY FOR DATA (attivo alto)**
  - $rfd = 0 \rightarrow$  **IL CONSUMATORE NON E' ANCORA PRONTO AD ACCETTARE NUOVI DATI**

- $rfd = 1 \rightarrow$  IL CONSUMATORE E' PRONTO AD ACCETTARE UN NUOVO DATO



Visione funzionale di un handshake /dav-rfd

Vediamo ora in dettaglio come funziona la temporizzazione dell'handshake:



Si parte da una situazione iniziale in cui tutte le reti sono connesse allo stesso circuito di *reset*, quindi  $(/dav, rfd) = (1, 1)$

### 1. STEP 1:

- Il produttore prepara un nuovo dato** e lo mette sui fili di dato
- SOLO DOPO**, mette  $/dav \rightarrow 0$  (convalida del dato per il consumatore)  
**(In Verilog questo "solo dopo" richiederà lo Stato Interno successivo)**

### 2. STEP 2:

- Il consumatore preleva il dato** e lo memorizza
- SOLO DOPO**, mette  $rfd \rightarrow 0$  (conferma di avvenuta lettura per il produttore)  
(Questo è importante perché il fronte di discesa di  $rfd$  segnala al produttore che può "ritirare" i dati dai fili di dato, quindi se poi li leggo dopo che sono stati "ritirati", ci trovo roba a caso!)  
**(In Verilog questo "solo dopo" richiederà lo Stato Interno successivo)**

### 3. STEP 3:

- a. Quando vuole, il produttore rimette  $/dav \rightarrow 1$  (conferma di poter ritirare su  $rfd$  al consumatore e finire l'handshake)

#### 4. STEP 4:

- a. Il consumatore rimette  $rfd = 1$  (ritorno alla situazione di *reset*)

Le transizioni devono essere alternate perché sennò si potrebbe generare un *deadlock*, infatti il produttore inizialmente potrebbe:

- Portare  $/dav \rightarrow 0$
- Vedere ancora  $rfd = 1$

Ora il consumatore porta  $rfd \rightarrow 0$ .

A questo punto se il consumatore non si assicura che il produttore non abbia GIA' riportato  $/dav \rightarrow 1$ , e tenta di riportare  $rfd \rightarrow 1$ , si crea un *deadlock* perché il consumatore aveva visto  $rfd = 1$  quando aveva controllato, e anche ora lo vede ad 1, quindi aspetta la prossima mossa dal consumatore, cioè comunicare quando sarà pronto per il dato mettendo  $rfd \rightarrow 0$ , ma il consumatore aspetta che riparta l'handshake ( $/dav \rightarrow 0$ )!



#### ERRORI TIPICI:

- **Nello STEP 2: Supporre che il dato sia buono dopo che  $rfd$  è andato a 0!**  
**(Ovvero prendere il dato dai fili di dato dopo che  $rfd \rightarrow 0$ )**
- **Nello STEP 4: Non finire l'handshake ammodo scordandosi di aspettare che  $/dav \rightarrow 1$  prima di rimettere  $rfd = 1$**

Un handshake */dav-rfd* lato consumatore si può fare in 3 stati:

```
// Al reset: /dav = 1, rfd = 1
S0: begin
    RFD <= 1;
    STAR <= (dav_ == 0) ? S1 : S0;
    // qui puoi già memorizzare il dato sui fili di dato,
    // perché per quando dav_ va a 0 il dato era già buono
    // da un pò di tempo!
end
```

```

S1: begin
    // Ci vuole un clock di distanza da quando prelevo il dato e quando metto RFD <= 0
    RFD <= 0;
    STAR <= S2;
    // qui puoi già lavorare col dato che hai memorizzato in S0
    // Inoltre:
    // Se salvi il dato in un registro e ci lavori, e non hai più bisogno
    // del dato che sta sui fili di dato, puoi buttare RFD <= 0
    //
    // Se invece devi usare ancora il dato sui fili, DEVI PER FORZA
    // TENERE RFD <= 1 IN QUESTO STATO, E SOLO DOPO CHE HAI FINITO DI
    // LAVORARCI (NELLO STATO DOPO, IN QUESTO CASO IN S2) BUTTARE GIU RFD!
end

S2: begin
    RFD <= 0;
    STAR <= (dav_ == 1) ? S0 : S2;
end

```

## Ma anche in 2:

```

S0: begin
    RFD <= 1
    STAR <= (dav_ == 0) ? S1 : S0;
    BUFFER <= d7_d0;
end

S1: begin
    // Ci vuole un clock di distanza da quando prelevo il dato e quando metto RFD <= 0
    RFD <= 0;
    STAR <= (dav_ == 1) : S0 : S1;
end

```

## Un handshake /dav-rfd lato produttore si fa tipicamente in 3 stati:

```

// Al reset: /dav = 1, rfd = 1
S0: begin
    D7_D0 <= d7_d0;
    STAR <= S1;
end

S1: begin
    // Quando metto DAV_ <= 0 il dato deve essere già pronto!
    DAV_ <= 0;
    STAR <= (rfd == 0) ? S2 : S1;
    // Lato produttore non devo preoccuparmi del dato, ci pensa il consumatore!
end

S2: begin
    DAV_ <= 1;

```

```

STAR <= (rfd == 1) ? S0 : S2;
end

```

**Come ricordare le transizioni:**  $daV - rfd$

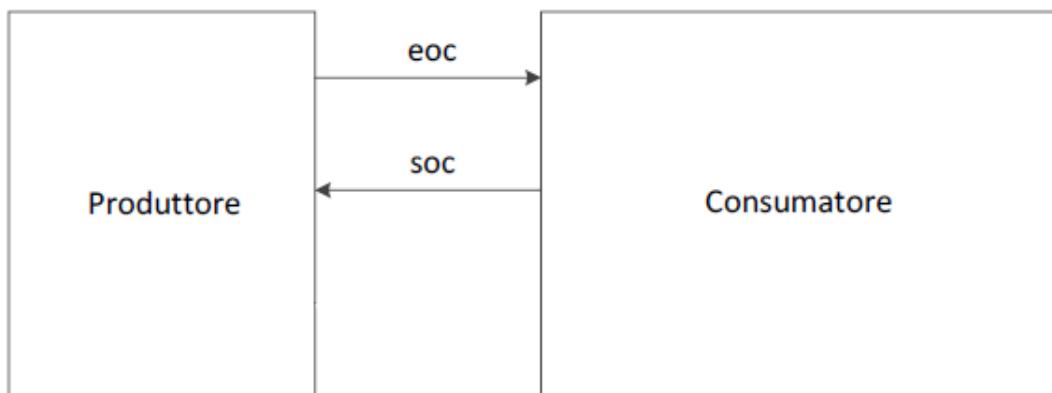
- $/dav$  fa la transizione  $1 - 0 - 1$ , la quale sembra una  $\mathcal{V}$
- $rfd$  è  $/dav$  ritardato (senza offesa)

## Handshake soc-eoc

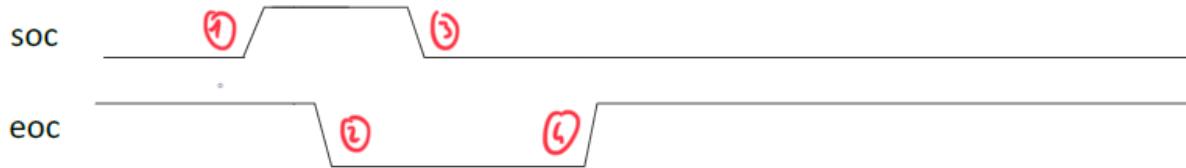
L'handshake soc-eoc è un handshake che viene usato nei Convertitore D/A e A/D!

Anche qui si dotano le due reti di due fili (variabili logiche):

- $soc$ : **START OF COMPUTATION** (attiva alta)
  - $soc = 0 \rightarrow$  AVVENUTA RICEZIONE DELL'ACCETTAZIONE DELLA RICHIESTA
  - $soc = 1 \rightarrow$  RICHIESTA DI UN NUOVO DATO
- $eoc$ : **END OF COMPUTATION** (attiva alta)
  - $eoc = 0 \rightarrow$  ACCETTAZIONE RICHIESTA DI NUOVO DATO
  - $eoc = 1 \rightarrow$  DATO PRONTO



Vediamo come funziona in dettaglio la temporizzazione di questo handshake:



Si parte da una situazione al *reset* in cui  $(soc, eoc) = (0, 1)$ .

1. **STEP 1:**

- a. Il consumatore porta  $soc \rightarrow 1$  per richiedere l'invio di un nuovo dato

2. **STEP 2:**

- a. Il produttore porta  $eoc \rightarrow 0$ , segnalando l'accettazione della richiesta.

**Ancora non si può prelevare il dato!**

3. **STEP 3:**

- a. Il consumatore riporta  $soc \rightarrow 0$  e aspetta il dato

4. **STEP 4:**

- a. Il produttore prepara il dato e lo mette sui fili di dato

- b. **SOLO DOPO**, riporta  $eoc \rightarrow 1$

**ATTENZIONE!** Non si può subito prelevare il dato che c'è sul bus al *reset*, perché ce l'avrà messo qualche altra rete del calcolatore, e non è il dato che devo richiedere io!

Un handshake *soc-eoc* lato consumatore si fa tipicamente in 2 stati:

```
// Al reset: soc = 0, eoc = 1
S0: begin
    SOC <= 1;
    STAR <= (eoc == 0) ? S1 : S0;
end

S1: begin
    SOC <= 0;
    STAR <= (eoc == 1) ? S2 : S1;
    // puoi memorizzare il dato che c'è sui fili di dato già qui!
    // così puoi cominciare a lavorarci dal prossimo Stato S2!
end
```

**Un handshake soc-eoc lato produttore si fa tipicamente in 3 stati:**

```
// Al reset: soc = 0, eoc = 1
S0: begin
    STAR <= (soc == 1) ? S1 : S0;
end

S1: begin
    EOC <= 0;
    STAR <= (soc == 0) ? S2 : S1;
end

S2: begin
    EOC <= 1;
    BUFFER <= d7_d0;
end

// Qui ci vogliono PER FORZA 3 stati, perché da S2 quando soc -> 0
// non posso tornare in S0 e metterci EOC <= 1!
// Perché i dati che c'erano sui fili di dato al reset
// NON ERANO PER ME, NON SONO BUONI, quindi devo PER FORZA
// andare in un terzo stato e qui riportare EOC ad 1 e memorizzare il dato!
```

**E' ovvio che il consumatore ci mette 1 stato in meno del produttore, comincia lui l'handshake!**

**Come ricordare le transizioni:**

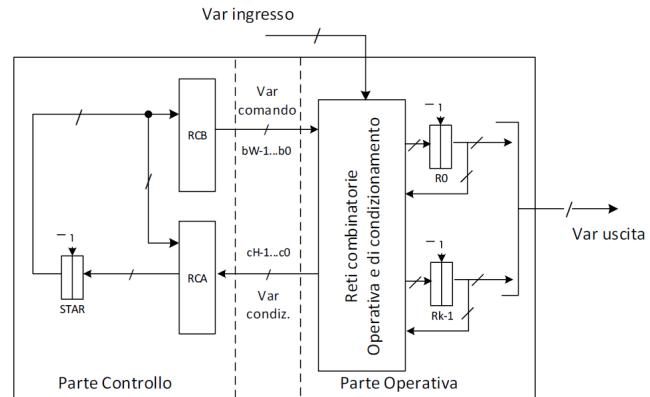
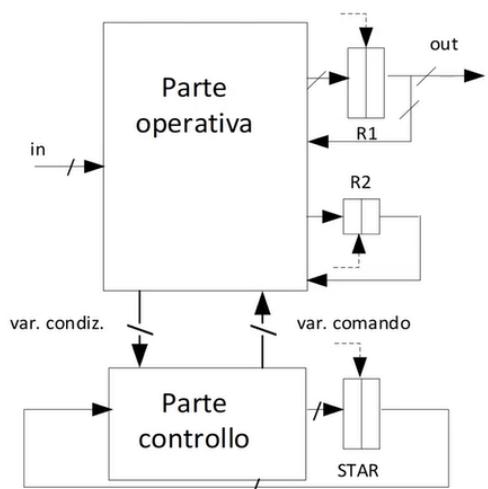
- *soc* è simile graficamente ad un impulso
- *eoc* ha la stessa forma di */dav* ed *rfd*, inoltre è uguale a *soc* ritardato e negato!

## Sintesi di RSS mediante scomposizione in PO/PC

Si divide la rete in due sottoreti:

- Parte Operativa
- Parte Controllo

Dove queste sottoreti comunicano fra loro con delle nuove variabili che bisogna sintetizzare, ed hanno entrambe lo stesso clock



Scomposizione in Parte Operativa e Parte Controllo di una RSS

## Parte Operativa:

**Logica necessaria ad interfacciarsi col resto del calcolatore e produrre gli ingressi per i Registri Operativi.**

### INGRESSI:

- Variabili di comando
- Variabili di ingresso alla rete totale

### USCITE:

- Variabili di condizionamento
- Variabili di uscita della rete totale

**RETE COMBINATORIA OPERATIVA:** Rete Combinatoria che sta davanti al proprio Registro Operativo

Ciascuna RCO ha come ingressi:

- Lo **Stato di Uscita** del corrispondente Registro Operativo
- Le **Variabili di Comando**
- [ Le **variabili di ingresso della rete** ]

E come uscita lo **Stato di Ingresso** del proprio Registro Operativo



**VA SINTETIZZATA FINO ALLE RETI STANDARD (MUX, SOMMATORE, CONTATORE, ETC...)**

**RETE COMBINATORIA DI CONDIZIONAMENTO:** Rete Combinatoria che sintetizza le **VARIABILI DI CONDIZIONAMENTO**  $c_i$ .

La RCC ha come ingressi:

- Le **variabili di ingresso della rete**
- Lo **Stato dei Registri Operativi**

E come uscita le **VARIABILI DI CONDIZIONAMENTO**



**VA SINTETIZZATA FINO ALLE ESPRESSIONI IN ALGEBRA DI BOOLE!**

**Visto che la RCC è una RSS di Mealy,** (perché le variabili di condizionamento sono funzione combinatoria degli ingressi), **tutta la RETE OPERATIVA (RCC + RCO) è di Mealy!**

## Algoritmo di sintesi della PO

- **SINTESI DELLE RCO:**

**Per ciascun Registro Operativo:**

- Lo si isola
- Si individuano le  $\mu$ -operazioni **DIVERSE** per esso  
(cioè le operazioni fatte sul Registro che appaiono dopo il  $<=$ )
- Si sintetizza il Registro Operativo come se fosse un Registro Multifunzionale
- Si generano le **VARIABILI DI COMANDO** che guidano il MUX tramite un'altra RC che ha come ingresso lo Stato Interno

- **SINTESI DELLA RCC:**

- Si individuano le condizioni **INDIPENDENTI** del Registro di Stato STAR  
(cioè le scelte che guidano i  $\mu$ -salti)  
**(ATTENZIONE!  $(x_0 == 0) ? A : B \iff (x_0 == 1) ? B : A$ )**
- Assegno a ciascuna di queste una **VARIABILE DI CONDIZIONAMENTO**, che vale 1 se la condizione è vera, 0 se è falsa

## Parte Controllo:

Logica dedicata all'aggiornamento del Registro di Stato STAR.

Notare come la PC sia ISOLATA sia dagli ingressi che dalle uscite della rete!

La PC è una Rete di Moore, al contrario della PO che è di Mealy!



**PARTE OPERATIVA**  $PO \rightarrow Mealy$  (**operAtiva** → **meAly**)

**PARTE CONTROLLO**  $PC \rightarrow Moore$  (**contrOllo** → **mOOre**)

Inoltre la PC non controlla le uscite, ma solo l'evoluzione dello Stato Interno (STAR)

## Sintesi standard della PC

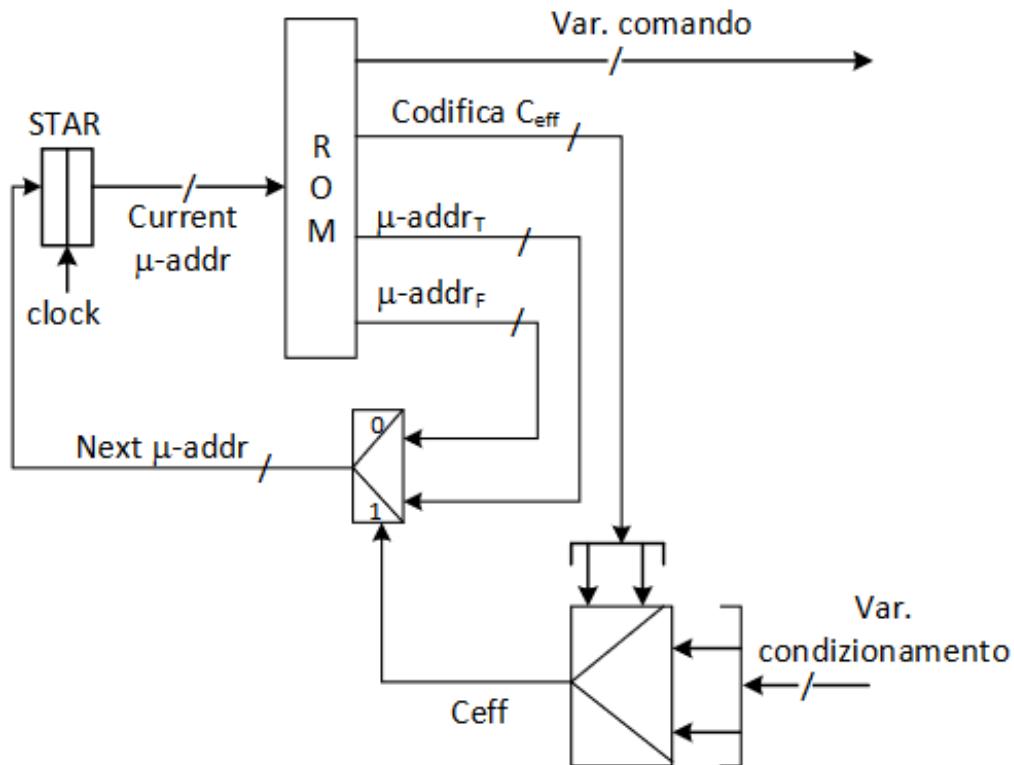
### TABELLINA DELLA ROM:

- **$\mu$ -INDIRIZZO:** la codifica degli Stati Interni
- Per ciascun  $\mu$ -indirizzo:
  - Individuo il  **$\mu$ -CODICE**, ovvero lo stato delle variabili di comando corrispondente
  - Associo una codifica in base  $\beta = 2$  ai PEDICI delle variabili di condizionamento
  - **$c_{eff}$** : **VARIABILE DI CONDIZIONAMENTO EFFICACE**, ovvero la variabile che guida il  $\mu$ -salto  
(Nel caso di  $\mu$ -salto incondizionato, la codifica è non specificata -)
- $\mu - addr_T$ :  $\mu$ -indirizzo a cui si salta se la condizione è vera
- $\mu - addr_F$ :  $\mu$ -indirizzo a cui si salta se la condizione è falsa  
(Nel caso di  $\mu$ -salto incondizionato  $\mu - addr_T = \mu - addr_F$ )

$\mu - addr$	$b_i$	$c_{eff}$	$\mu - addr_T$	$\mu - addr_F$
...	...	...	...	...

Tutto ciò che sta a destra del  $\mu - addr$  si chiama  **$\mu$ -ISTRUZIONE**, perché descrive tutto quello che fa la PC nello stato codificato con il  $\mu$ -indirizzo corrispondente!

### SINTESI BASATA SUI $\mu$ -INDIRIZZI ( $\mu$ -ADDRESS BASED):



Sintesi della PC basata sui  $\mu$ -indirizzi

**La parte di ROM che sintetizza  $b_i$  (variabili di comando, ovvero il  $\mu$ -codice) è  $RCB$ .**

**La parte di ROM che sintetizza le altre 3 colonne  $c_{eff}$ ,  $\mu - addr_T$ ,  $\mu - addr_F$  è  $RCA$ .**

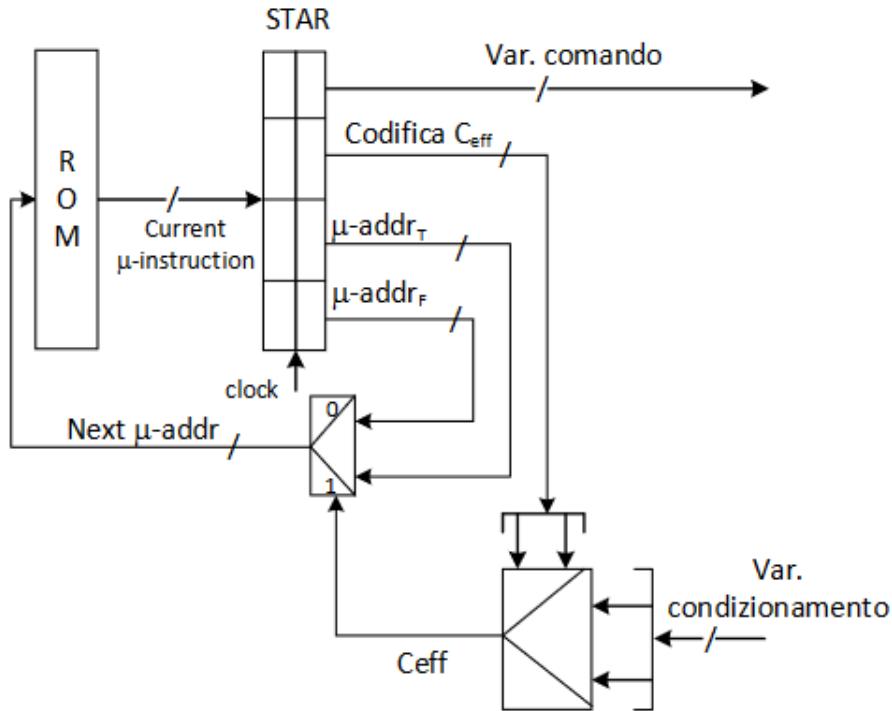
Se la PC ha più variabili di condizionamento, le uniche cose che si "complicano", cioè aumentano di dimensione, sono il MUX che ha in uscita  $c_{eff}$  e la ROM

Anche se questa sintesi presenta più di 2LL, non è un problema, perché c'è solo roba molto semplice. Il vero problema che ritarda la rete è cosa c'è nella PO, perché lì c'è il grosso (sommatori, moltiplicatori, etc..) che esegue le operazioni!

**Qui bisogna tenere largo il clock, perché ho la ROM in cascata alla RCO!**

### **SINTESI BASATA SULLE $\mu$ -ISTRUZIONI ( $\mu$ -INSTRUCTION BASED>):**

Basta scambiare ROM e Registro STAR:



Sintesi della PC basata sulle  $\mu$ -istruzioni

**Qui invece la dimensione del Registro STAR aumenta notevolmente, ma visto che la ROM e la RCO non sono più in cascata posso far andare il clock più velocemente!**

**La ROM passa da essere in cascata alla RCO (prima) ad essere in cascata alla RCC (ora), e visto che la RCC tipicamente è più semplice della RCO, non c'è più il problema di dover tenere largo il clock!**

### Sintesi per casi particolari della PC

Come si gestiscono i  $\mu$ -salti a più di 2 vie?

Usando il **REGISTRO OPERATIVO MJR (MULTIWAY JUMP REGISTER)**.



**A meno che non sia assolutamente necessario, estremamente comodo, o esplicitamente richiesto dal testo dell'esame, EVITARE DI USARE IL REGISTRO MJR!**

**La maggior parte degli esercizi si fa usando salti a due vie!!!**

**Edit: svolgendo tutte le prove d'esame dal 2019 al 2022 non mi è mai capitato di dover usare *MJR* per fare una sintesi!**

Per la descrizione è semplice:

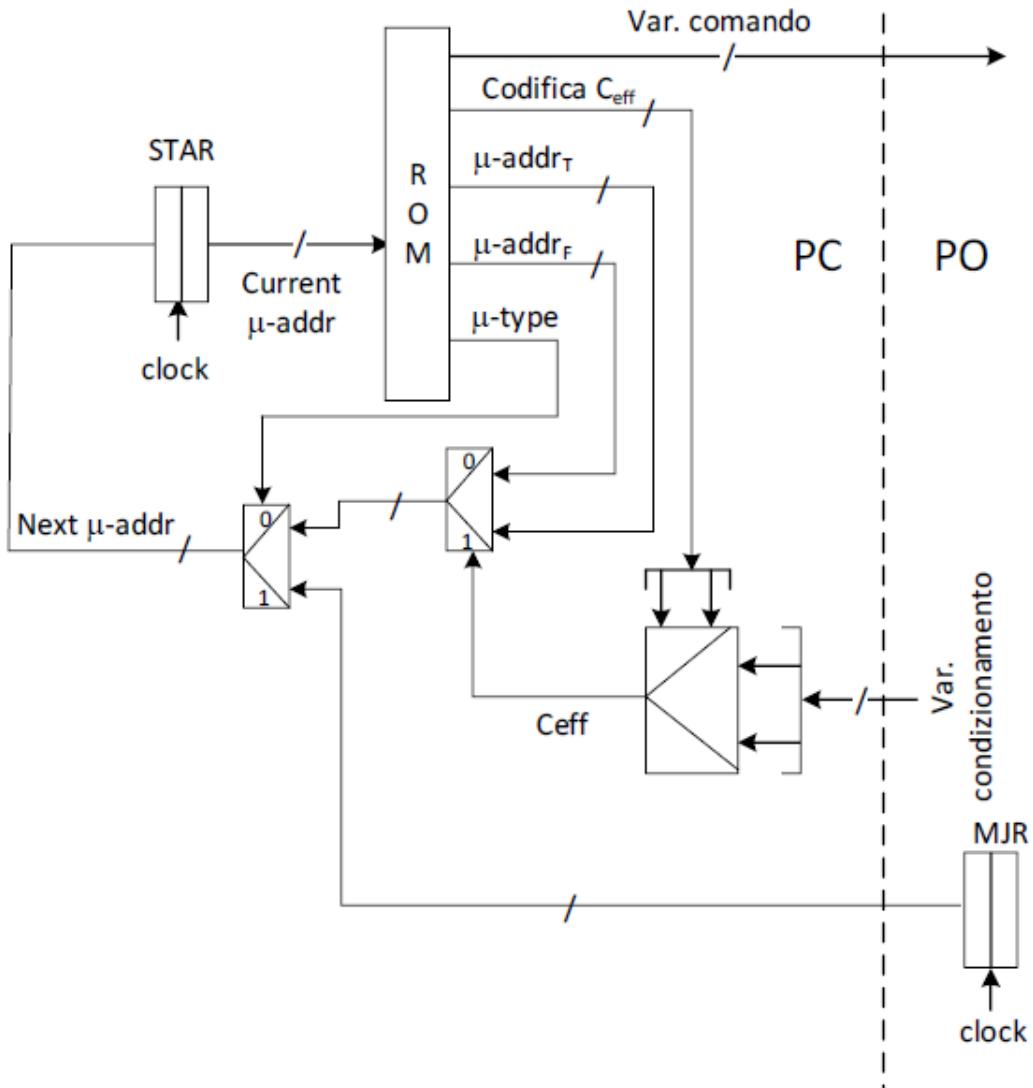
```
// Ho questo salto a k vie
S0: begin
    STAR <= (condizione 1) ? S1 :
        (condizione 2) ? S2 :
        ...
        (condizione k-1) ? S{k-1} : S{k};
end

// Che traduco così:
S0: begin
    MJR <= (condizione 1) ? S1 :
        (condizione 2) ? S2 :
        ...
        (condizione k-1) ? S{k-1} : S{k};
    STAR <= S0_1;
end

S0_1: begin STAR <= MJR; end
// le cose segnate in rosso sono le modifiche/aggiunte
```

Cioè sostanzialmente si fa valutare la condizione assegnandola ad *MJR*, e poi una volta valutata si assegna a *STAR*

Vediamo ora come si sintetizza:



Viene aggiunta nella ROM una colonna in più, il  **$\mu$ -TIPO** del  $\mu$ -salto, che dice per ogni  $\mu$ -salto se questo è guidato dalla sua variabile di condizionamento o dal Registro Operativo MJR

Praticamente tutto quello che basta fare è mettere un altro MUX a monte del registro STAR, che ha come variabili di comando il  $\mu$ -tipo, e come ingressi:

- L'uscita del MUX che insegue  $\mu - addr_T$  o  $\mu - addr_F$
- L'uscita del Registro operativo MJR

**ATTENZIONE! Il Registro MJR lo posso usare per fare sintesi basate sia su  $\mu$ -indirizzi che su  $\mu$ -istruzioni, ma IN QUALESiasi CASO IL REGISTRO MJR CONTIENE  $\mu$ -INDIRIZZI!**

## Sottoliste

Il Registro MJR può essere usato per implementare le **SOTTOLISTE**, cioè pezzi di  $\mu$ -programma che possono essere raggiunti a partire da stati di partenza diversi.

In piena analogia con Assembler, il controllo passa all  $\mu$ -sottoprogramma, e poi una volta terminato, il controllo ritorna al  $\mu$ -programma chiamante e continua dallo **Stato Successivo!**

**ATTENZIONE! Il parallelo con l'Assembler si limita ad un livello di annidamento!**

```
S0: begin
    MJR <= S1;      // salvo lo Stato di Ritorno (il PROSSIMO Stato)
    STAR <= Sub1; // faccio il "CALL sottoprogramma"
end

S1: /* conti */

...
Sx: begin
    MJR <= S{x+1}; // salvo lo Stato di Ritorno (il PROSSIMO Stato)
    STAR <= Sub1; // faccio il "CALL sottoprogramma"
end

S{x+1}: /* conti */

...
Sub1: begin
    /* conti */
    // si continua ad andare avanti ed eseguire Sub2, Sub3, etc...
end

...
Sub{k}: begin
    /* conti */
    STAR <= MJR; // RET
end
```

In sostanza  $MJR$  prende lo stato successivo, salvandolo, poi si manda  $STAR$  ai sotto-stati che ci interessa eseguire, e poi si dice  $STAR \leq MJR$  così  $STAR$  "ritorna" al flusso principale

# RSS Complesse fondamentali

## Contatore di sequenze alternate [00, 01, 10] - [11, 01, 10]

Questa rete deve incrementare un contatore a 4 bit quando riconosce la prima sequenza, poi incrementi quando riconosce la seconda, poi di nuovo la prima, etc...

La rete avrà:

- 2 ingressi:  $x_1, x_0$
- 4 uscite: il contenuto del Registro COUNT, che è a 4 bit

### Descrizione

Si nota subito una cosa: le due sequenze sono uguali, la sola cosa che cambia è il primo passo di ciascuna sequenza!

Quindi, decido io arbitrariamente che:

- Se  $COUNT[0] = 0$  la rete deve riconoscere la sequenza che inizia per 00
- Se  $COUNT[0] = 1$  la rete deve riconoscere la sequenza che inizia per 11

Una volta capito questo, fa comodo fare una RC (che chiamo *match*) che abbia:

- Come input  $COUNT[0], x_1, x_0$
- Come output un bit che vale 1 se  $(COUNT[0] \ x_1 \ x_0) = (111)$  oppure  $(000)$

$COUNT[0]$	$x_1$	$x_0$	$match$
0	0	0	1
1	1	1	1
...	...	...	0

```
// File: Descrizione.v
module ContSeqAlternato(x1_x0, z3_z0, clock, reset_);
    input clock, reset_
    input [1:0] x1_x0;
    input [3:0] z3_z0;

    reg [1:0] STAR; parameter S0 = 'B00, S1 = 'B01, S2 = 'B10;
    reg [3:0] COUNT; assign z3_z0 = COUNT;

    always @(reset_ == 0) #1
        begin
```

```

STAR <= S0;
COUNT <= 'B0000;
end

always @(posedge clock) if(reset_ == 1) #3
  casex(STAR)
    S0: begin
      COUNT <= COUNT;
      STAR <= ( match(COUNT[0], x1_x0) == 1 ) ? S1 : S0;
    end
    S1: begin
      COUNT <= COUNT;
      STAR <= (x1_x0 == 'B01) ? S2 : ( match(COUNT[0], x1_x0) == 1 ) ? S1 : S0;
    end
    S2: begin
      COUNT <= (x1_x0 == 'B10) ? COUNT +1 : COUNT;
      STAR <= ( match(COUNT[0], x1_x0) == 1 ) ? S1 : S0;
    end
  endcase
endmodule

```

Commenti:

- $COUNT$  viene inizializzato a 0 al reset, quindi la prima sequenza che si riconosce è [00, 01, 10].
- $COUNT$  viene incrementato ad ogni sequenza corretta riconosciuta, quindi il suo bit meno significativo,  $COUNT[0]$ , alterna effettivamente fra 0 e 1

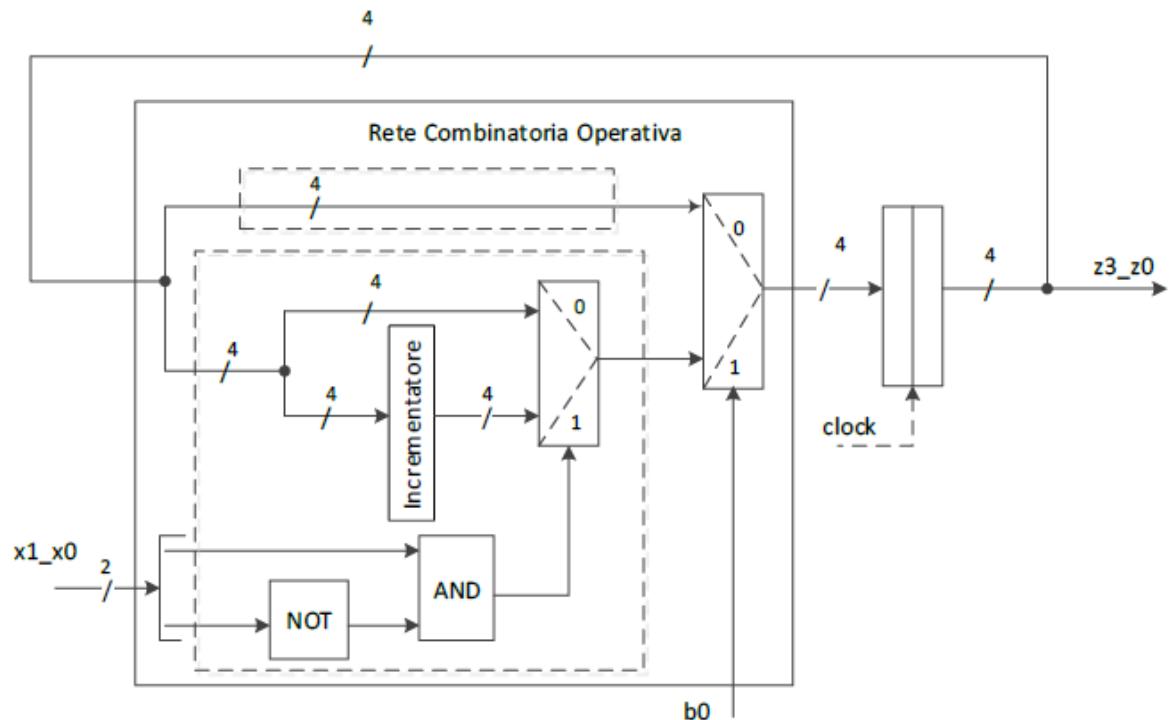
## Sintesi - Parte Operativa

### RETE COMBINATORIA OPERATIVA:

```

S0, S1: COUNT <= COUNT;
S2:     COUNT <= (x1_x0 == 'B10) ? COUNT +1 : COUNT;

```



```
assign b0 = (STAR == S2) ? 1 : 0; // Generazione della variabile di comando
```

### RETE COMBINATORIA DI CONDIZIONAMENTO:

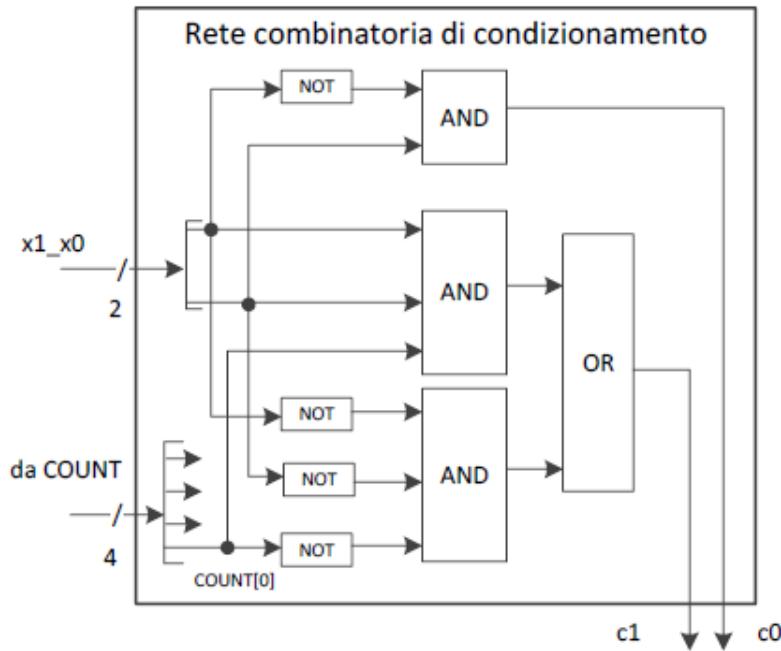
```

Condizione 1: ( match(COUNT[0], x1_x0) == 1 )
Condizione 2: ( x1_x0 == 'B01 )

// Generazione delle variabili di condizionamento (descrizione)
assign c1 = ( match(COUNT[0], x1_x0) == 1 ) ? 1 : 0;
assign c0 = ( x1_x0 == 'B01 ) ? 1 : 0;

// Generazione delle variabili di condizionamento (sintesi)
assign c1 = ( COUNT[0] & x1 & x0 ) | ( (~COUNT[0]) & (~x1) & (~x0) );
assign c0 = (~x1) & x0;

```



Ora, mettendo insieme i pezzi di descrizione prodotti:

```

module RicSeqAlternate(x1_x0, z3_z0, clock, reset_);
  input clock, reset_;
  input [1:0] x1_x0;
  output [3:0] z3_z0;

  reg [1:0] STAR; parameter S0 = 'B00, S1 = 'B01, S2 = 'B10;
  reg [3:0] COUNT; assign z3_z0 = COUNT;

  // Variabili di comando e condizionamento:
  wire b0, c1, c0;

  // Generazione delle variabili di comando
  assign b0 = (STAR == S2) ? 1 : 0;

  // Generazione delle variabili di condizionamento
  assign c1 = ( COUNT[0] & x1_x0 ) | ( (~COUNT[0]) & (~x1) & (~x0) );
  assign c0 = (~x1) & x0;

  // Blocco always per il Registro Operativo COUNT
  always @(reset_ == 0) #1 COUNT <= 'B0000;
  always @(posedge clock) if(reset_ == 1) #3
    casex(b0)
      0: COUNT <= COUNT;
      1: COUNT <= (x1_x0 == 'B10) ? COUNT +1 : COUNT;
    endcase

  // Blocco always per il Registro di Stato STAR
  always @(reset_ == 0) #1 STAR <= S0;
  always @(posedge clock) if(reset_ == 1) #3
    casex(b0)
      S0: STAR <= (c1 == 1) ? S1 : S0;
      S1: STAR <= (c0 == 1) ? S2 : (c1 == 1) ? S1 : S0;

```

```

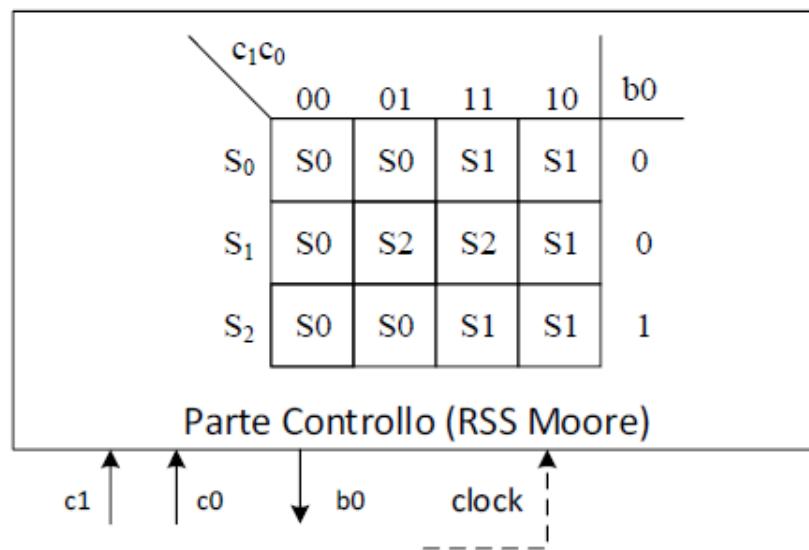
S2: STAR <= (c1 == 1) ? S1 : S0;
endcase

endmodule

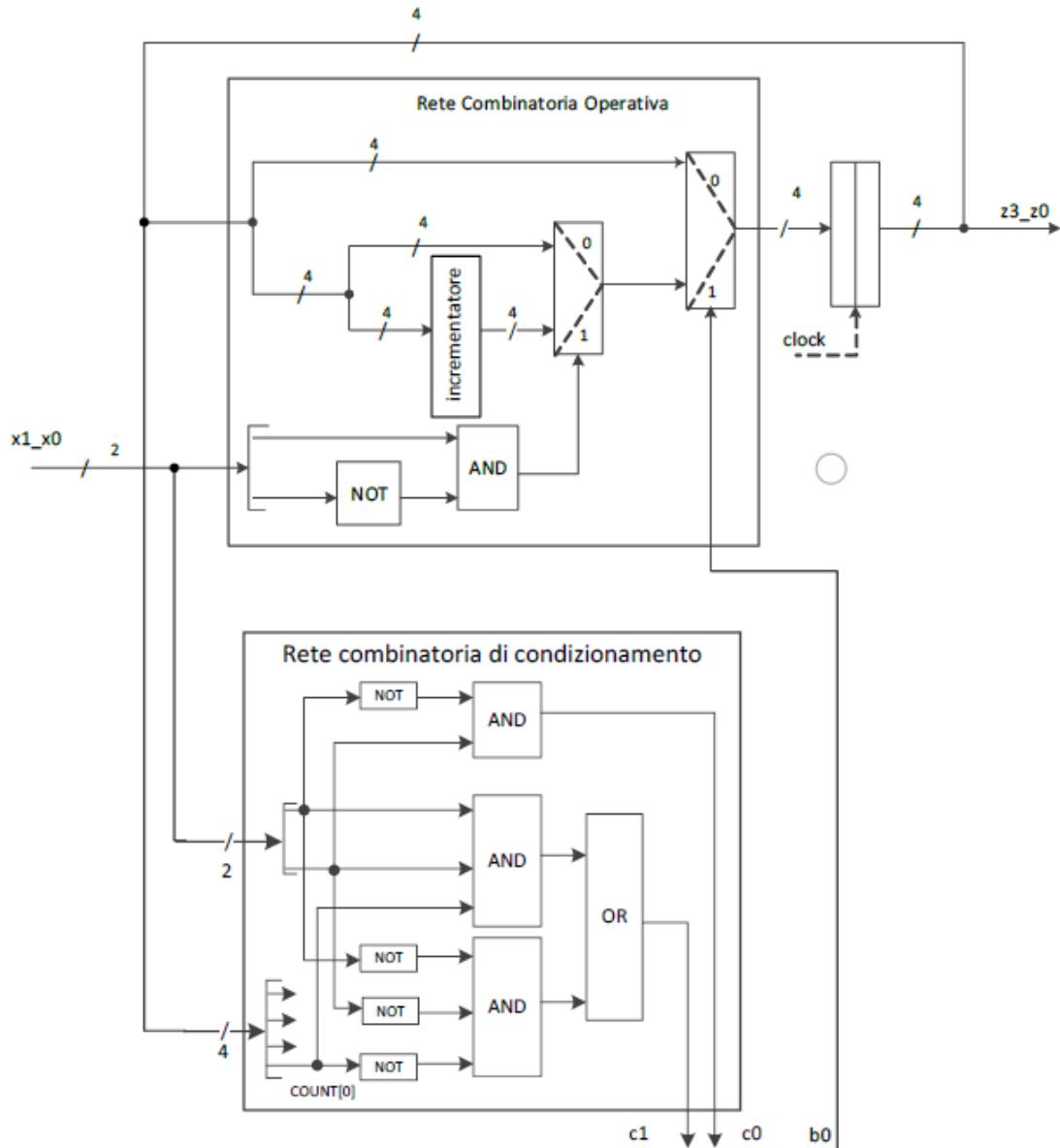
```

## Sintesi - Parte Controllo

La PC implementa la tabella di flusso:



## Sintesi - Parte Operativa



```
// File: Sintesi.v
module RicSeqAlternate(x1_x0, z3_z0, clock, reset_);
  input clock, reset_
  input [1:0] x1_x0;
  output [3:0] z3_z0;
  wire b0, c1, c0;

  ParteOperativa PO(x1_x0, z3_z0, b0, c1, c0, clock, reset_);
  ParteControllo PC(b0, c1, c0, clock, reset_);

endmodule

module ParteOperativa(x1_x0, z3_z0, b0, c1, c0, clock, reset_);
  input clock, reset_
  input [1:0] x1_x0;
  output [3:0] z3_z0;
```

```

input b0;
output c1, c0;

reg [3:0] COUNT; assign z3_z0 = COUNT;
wire x1, x0; assign x1 = x1_x0[1]; assign x0 = x1_x0[0];

// Generazione delle variabili di condizionamento
assign c1 = ( COUNT[0] & x1 & x0 ) | ( (~COUNT[0]) & (~x1) & (~x0) );
assign c0 = (~x1) & x0;

// Blocco always per il Registro Operativo COUNT
always @(reset_ == 0) #1 COUNT <= 'B0000;
always @(posedge clock) if(reset_ == 1) #3
  casex(b0)
    0: COUNT <= COUNT;
    1: COUNT <= (x1_x0 == 'B10) ? COUNT +1 : COUNT;
  endcase
endmodule

module ParteControllo(b0, c1, c0, clock, reset_);
  input clock, reset_;
  input c1, c0;
  output b0;

  reg [1:0] STAR; parameter S0 = 'B00, S1 = 'B01, S2 = 'B10;

  // Generazione delle variabili di comando
  assign b0 = (STAR == S2) ? 1 : 0;

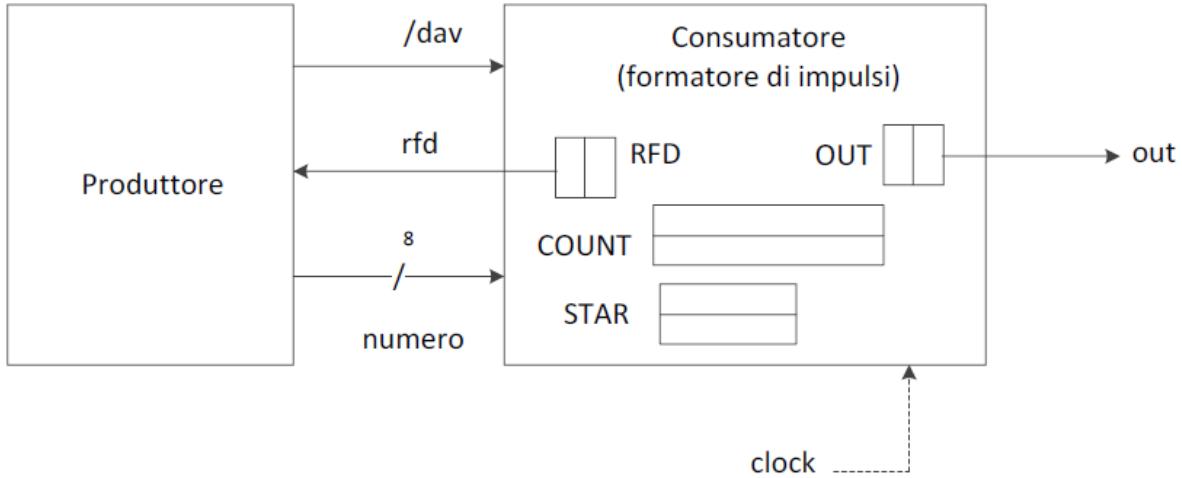
  // Blocco always per il Registro di Stato STAR
  always @(reset_ == 0) #1 STAR <= S0;
  always @(posedge clock) if(reset_ == 1) #3
    casex(b0)
      S0: STAR <= (c1 == 1) ? S1 : S0;
      S1: STAR <= (c0 == 1) ? S2 : (c1 == 1) ? S1 : S0;
      S2: STAR <= (c1 == 1) ? S1 : S0;
    endcase
  endmodule

/* tabellina della ROM */

```

## Formatore di impulsi con handshake /dav-rfd

Ho due reti che hanno clock diversi, quindi hanno bisogno di sincronizzarsi.



Una rete (produttore) produce numeri Naturali in base  $\beta = 2$  su 8 bit.

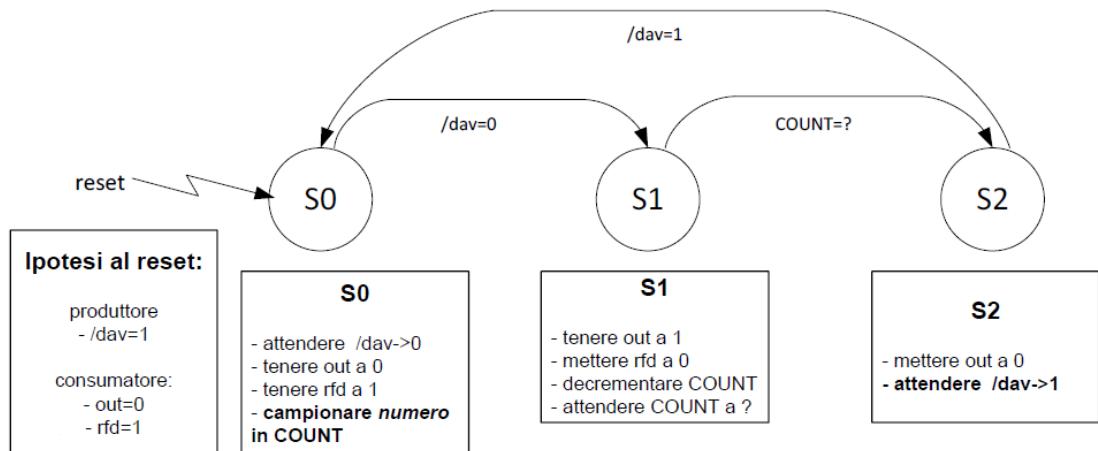
L'altra (consumatore) li preleva, e tiene la sua uscita *out* ad 1 per il numero di clock specificato dal dato prelevato, che si chiama *numero*.

Nel caso particolare in cui *numero* = 0, il consumatore deve tenere l'uscita ad 1 per 256 clock.

Innanzitutto, come si fa a far stare ad 1 l'uscita per un certo *numero* di clock? Servono:

- Il Registro *STAR*, a quanti bit? Lo vedrò dopo che avrò fatto la descrizione!
- Un registro *OUT* a 1 bit che sostiene l'uscita
- Un registro contatore *COUNT*, e facciamo "contare down" (decrementare) ad ogni clock.

A quanti bit sarà? Ad occhio per ora lo metto ad 8 bit, poi vedrò meglio una volta finita la descrizione!



In Verilog si descrive così:

```

module FormImpHS(numero, out, dav_, rfd, clock, reset_);
    input clock, reset_;
    input dav_;
    input [7:0] numero;

    output rfd, out;

    reg RFD; assign rfd = RFD;
    reg OUT; assign out = OUT;
    reg [7:0] COUNT;
    reg [1:0] STAR; parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;

    always @(reset_ == 0) #1
        begin
            RFD <= 1;
            OUT <= 0;
            STAR <= S0;
        end

    // Mia descrizione
    always @(posedge clock) if(reset_ == 1) #3
        casex(STAR)
            S0: begin
                OUT <= 0;
                COUNT <= numero;

                STAR <= (dav_ == 0) ? S1 : S0; //HS
            end

            S1: begin
                RFD <= 0; //HS

                STAR <= S2;
            end

            S2: begin

```

```

        STAR <= (dav_ == 1) ? S3 : S2; //HS
    end

    S3: begin
        RFD <= 1; //HS

        COUNT <= COUNT - 1;
        OUT <= 1;
        STAR <= (COUNT == 1) ? S0 : S4;
    end
endcase

// Descrizione di Stea
always @(posedge clock) if(reset_ == 1) #3
    casex(STAR)
        S0: begin
            RFD <= 1;
            OUT <= 0;
            COUNT <= numero;
            STAR <= (dav_ == 0) ? S1 : S0;
        end

        S1: begin
            RFD <= 0;
            OUT <= 1;
            COUNT <= COUNT - 1;
            STAR <= (COUNT == 1) ? S2 : S1;
        end

        S2: begin
            RFD <= 0;
            OUT <= 0;
            STAR <= (dav_ == 1) ? S0 : S2;
        end
    endcase
endmodule

```

E questa è la sintesi in PO/PC:

```

module FormImpHS(numero, out, dav_, rfd,
                 clock, reset_)

    input clock, reset_;
    input dav_;
    input [7:0] numero;

    output rfd, out;

    wire b0, b1, b2, b3;
    wire c0, c1;

    FormImpHS_PO PO(numero, out, dav_, rfd,
                     c0, c1,

```

```

        b0, b1, b2, b3,
        clock, reset_);

FormImpHS_PC PC(c0, c1,
                b0, b1, b2, b3,
                clock, reset_);

endmodule

module FormImpHS_PO(numero, out, dav_, rfd,
                    c0, c1,
                    b0, b1, b2, b3,
                    clock, reset_);

    input clock, reset_;
    input dav_;
    input [7:0] numero;

    output rfd, out;

    reg RFD; assign rfd = RFD;
    reg OUT; assign out = OUT;
    reg [7:0] COUNT;

    assign c0 = ~dav_; // dav_ == 0
    assign c1 = (~COUNT[7]) & (~COUNT[6]) & (~COUNT[5]) & (~COUNT[4]) & (~COUNT[3]) & (~COUNT[2]

always @(reset_ == 0) #1 RFD <= 1;
always @(posedge clock) if(reset_ == 1) #3
    casex(b0)
        'B0 S0: RFD <= 1;
        'B1 S1: RFD <= 0;
    endcase
    // Ottimizzazione: always .... if(reset_ == 1) #3 RFD <= ~b0;

always @(reset_ == 0) #1 OUT <= 0;
always @(posedge clock) if(reset_ == 1) #3
    casex(b1)
        'B0 S0: OUT <= 0;
        'B1 S1: OUT <= 1;
    endcase
    // Ottimizzazione: always .... if(reset_ == 1) #3 OUT <= b1;

always @(posedge clock) if(reset_ == 1) #3
    casex({b3, b2})
        'B00 S0: COUNT <= numero;
        'B01 S1: COUNT <= COUNT - 1;
        'B1? S2: COUNT <= COUNT;
    endcase

endmodule

module FormImpHS_PC(c0, c1,
                    b0, b1, b2, b3,

```

```

        clock, reset_);

input clock, reset_;
input c0, c1;
output b0, b1, b2, b3;

reg [1:0] STAR; parameter S0 = 0, S1 = 1, S2 = 2;

assign {b3, b2, b1, b0} =
    (STAR == S0) ? 'B0000 :
    (STAR == S1) ? 'B0111 :
    (STAR == S2) ? 'B1X01 :
/*default*/      'BXXXX;

always @(reset_ == 0) #1 STAR <= S0;
always @(posedge clock) if(reset_ == 1) #3
    casex(STAR)
        S0: STAR <= c0 ? S1 : S0;
        S1: STAR <= c1 ? S2 : S1;
        S2: STAR <= c0 ? S2 : S0;
    endcase

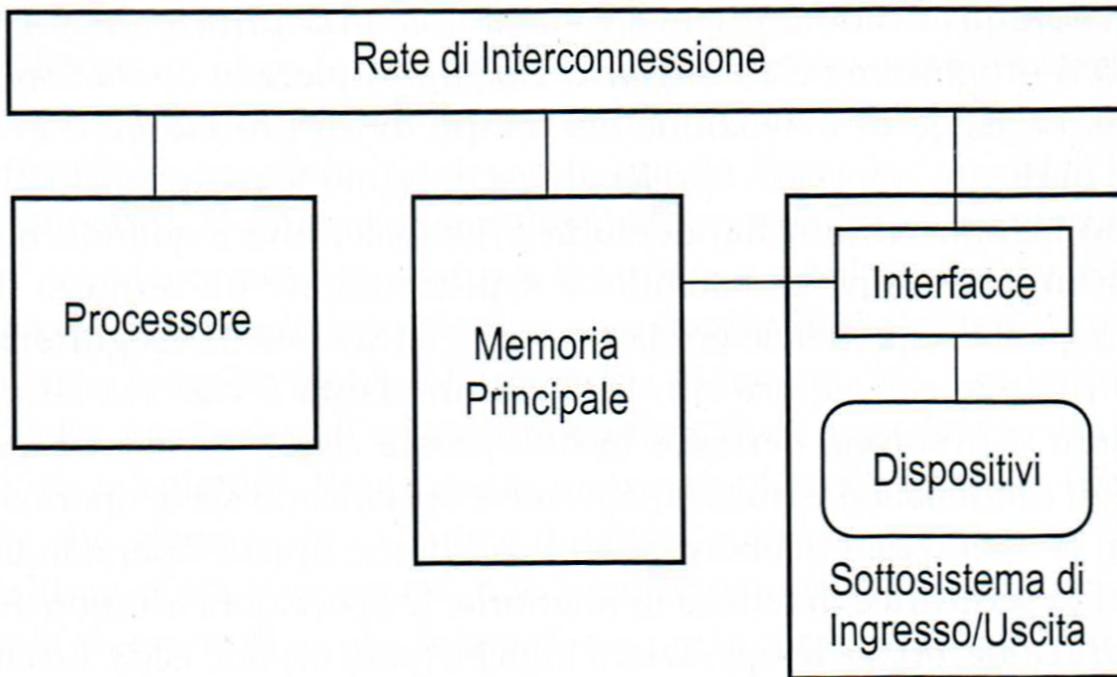
/*
ROM:
S0 = 00, S1 = 01, S2 = 10
c0 = 0, c1 = 1

u-addr      b3, b2, b1, b0    Ceff    u-addrT      u-addrF
00          0000                0       01           00
01          0111                1       10           01
10          1X01                0       10           00
*/
endmodule

```

# Struttura del calcolatore

## Generalità



Schema a blocchi di un calcolatore

Il calcolatore si compone di varie parti:

- **SOTTOSISTEMA DI INGRESSO/USCITA (I/O)**: Gestisce la codifica delle informazioni e il loro scambio col mondo esterno.

All'interno di questo sottosistema si distinguono **due componenti**:

- **INTERFACCE**: Gestiscono i dispositivi, cioè **standardizzano il colloquio fra i dispositivi (esterni) e il processore**. Le Interfacce inoltre possiedono dei **REGISTRI DI INTERFACCIA**, su cui il processore può fare *read* o *write*
  - **DISPOSITIVI**: Sono i **dispositivi esterni al calcolatore**
- **MEMORIA PRINCIPALE**: Contiene le istruzioni e i dati che il processore **elabora**. Visto che una parte di questa è adibita a **MEMORIA VIDEO**, vi è un collegamento diretto fra Memoria e Interfaccia (Video)

- **PROCESSORE:** Ciclicamente preleva una istruzione dalla memoria (**fase di fetch**) e la esegue. L'istruzione speciale *HLT* blocca questo ciclo e interrompe il processore.

Le istruzioni sono di due tipi:

- **ISTRUZIONI OPERATIVE:** Si trovano in sequenza in memoria (cioè in celle contigue), e specificano sia gli operandi che l'operazione (e.g. **istruzioni aritmetiche e logiche**).  
`ADD, SUB, MUL, IMUL, DIV, IDIV, AND, OR, NOT, XOR, ...`
- **ISTRUZIONI DI CONTROLLO:** Permettono di alterare il flusso sequenziale di esecuzione, alterando quindi l'ordine sequenziale dell'esecuzione del programma (e.g. **salti, condizionati e non, chiamate a sottoprogrammi, ritorno da sottoprogrammi**).

Le sole istruzioni di controllo sono queste: `JMP, Jcond, CALL, RET`

- **BUS (RETE DI INTERCONNESSIONE):** E' l'insieme di fili che mette in comunicazione tutti questi moduli presenti in un calcolatore.

Ma come si fa a capire quale istruzione eseguire dopo aver fatto il *reset*?

Il processore deve:

- Iniziare a **leggere la memoria da una locazione ben precisa**
- In quella locazione ci deve essere del **codice, scritto in maniera indelebile!**

Questo si realizza facendo in modo che al *reset* il processore abbia inizializzato l'**INSTRUCTION POINTER IP** alla prima cella della porzione di memoria implementata in tecnologia EPROM, la quale contiene il programma **BOOTSTRAP** che viene eseguito all'avvio del calcolatore.

Inoltre il Registro dei Flag *F* deve essere inizializzato a 0.

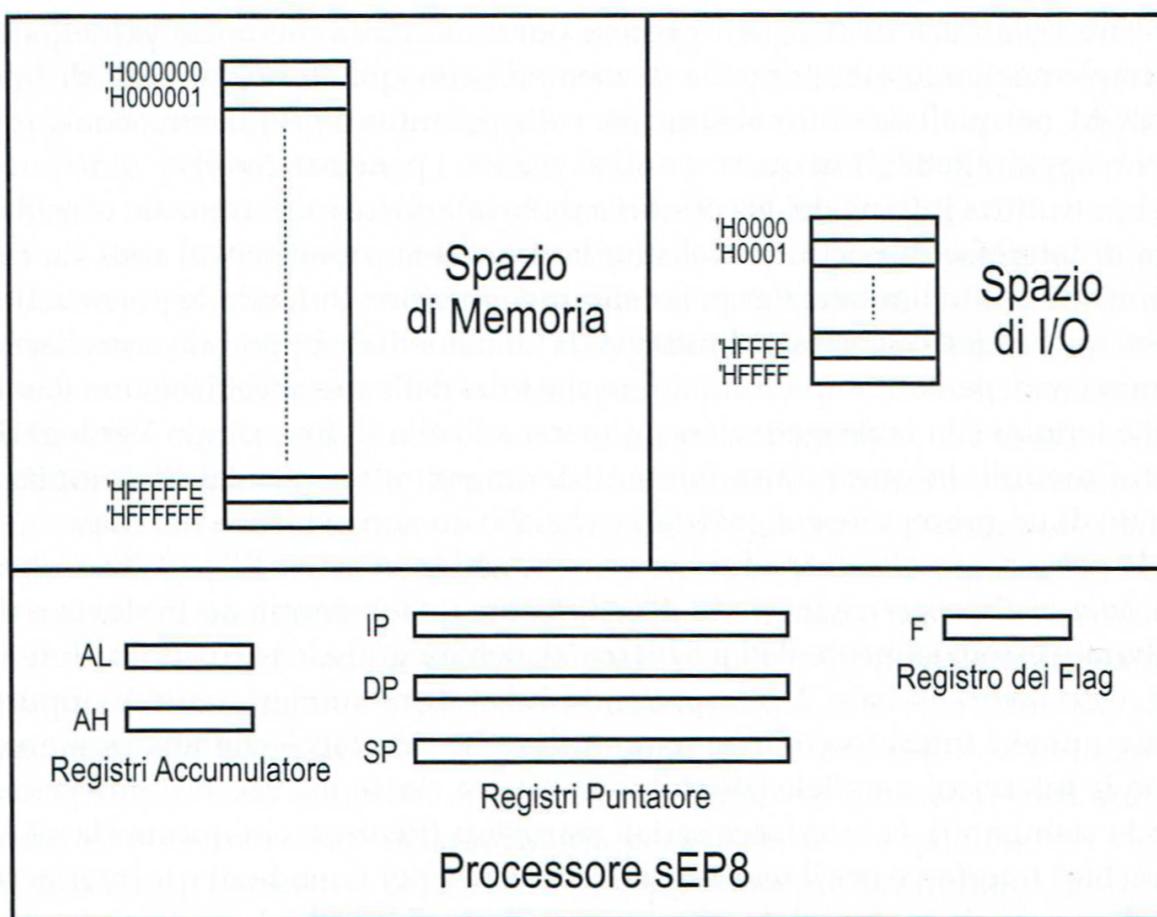


## **IL CALCOLATORE E' UNA SERIE DI RSS!**

Perché:

- **IL PROCESSORE E' UNA RSS**
- **LE INTERFACCE SONO DELLE RSS**
- **LA RAM E' UNA RSA**
- **LA ROM E' UNA RC**

## **Struttura logica del calcolatore visto a lezione**



Visione logica di un calcolatore dal punto di vista di un programmatore

La memoria è uno spazio lineare di  $2^{24}$  locazioni, da 8 bit ciascuna, per un totale di **16MB**.

**Spiegazione:**  $2^{24} = 2^4 \cdot 2^{20} = 2^4 MB = 16 MB$

**Dunque gli indirizzi dello spazio di memoria saranno a 24 bit!**  
**Mentre le locazioni sono da 8 bit ciascuna!**

Lo spazio di I/O appare come uno spazio lineare da  $2^{16} = 64K$  locazioni o **PORTE**.

Non necessariamente a ciascuna porta nello spazio di I/O corrisponderà un Registro di Interfaccia dal quale poter leggere o scrivere!

Quindi:



**Le porte di I/O implementate sono quelle in cui ci sono i Registri di Interfaccia!**

**Spiegazione:**  $2^{16} = 2^6 \cdot 2^{10} = 2^6 KB = 64 KB$

**Dunque gli indirizzi dello spazio di I/O saranno a 16 bit!**

**Mentre le locazioni sono da 8 bit ciascuna!**

**Come capire dimensione delle celle e dimensione di uno spazio di memoria:**

Supponiamo di avere lo spazio ' $H0000\_0000$  – ' $HFFFF\_FFFF$ '.

**DIMENSIONE DELLE CELLE → CONTA I CARATTERI ⇒ 8**

**DIMENSIONE DELLO SPAZIO → OGNI CARATTORE VALE 4 ⇒  $2^{32} = 2^2 \cdot 1 GB = 4 GB$**

**Reminder:**  $2^{10} = 1 KB$ ,       $2^{20} = 1 MB$ ,       $2^{30} = 1 GB$

Infine, il processore *sEP8* ha **3 tipi di registri**:

- **REGISTRI ACCUMULATORE:**  $AH$ ,  $AL$ , ad 8 bit, **contengono operandi di elaborazioni**
- **REGISTRO DEI FLAG:**  $F$ , ad 8 bit, **contiene tutti e 8 i flag** (che sono bit).

Gli unici flag interessanti per noi sono:

- $CF$ : **CARRY FLAG**
- $ZF$ : **ZERO FLAG**

- *SF*: **SIGN FLAG**
- *OF*: **OVERFLOW FLAG**
- **REGISTRI PUNTATORE**: Sono a 24 bit, perché devono contenere indirizzi di memoria, e sono:
  - *IP* : **INSTRUCTION POINTER**, contiene l'indirizzo della PROSSIMA istruzione da eseguire
  - *SP*: **STACK POINTER**, contiene l'indirizzo della cima dello stack
  - *DP*: **DATA POINTER**, contiene l'indirizzo di operandi a seconda della modalità di indirizzamento

## Linguaggio macchina del processore *sEP8*

Il formato delle istruzioni è `OPCODE source, destination`

### Modalità di indirizzamento di istruzioni operative:

- **INDIRIZZAMENTO DI REGISTRO**: Uno o entrambi gli operandi sono Registri.  
**Esempio:** `OPCODE %AL, %AH` oppure `OPCODE %DP`
- **INDIRIZZAMENTO IMMEDIATO**: L'operando *sorgente* è specificato direttamente nell'istruzione come costante.  
**Esempio:** `OPCODE $0x10, %AL`
- **INDIRIZZAMENTO DI MEMORIA**: Valido per il *sorgente* o per il *destinatario*, ma **MAI PER ENTRAMBI!** Ve ne sono due tipi:
  - **DIRETTO**: L'indirizzo è specificato direttamente nell'istruzione.  
**Esempio:** `OPCODE 0xF035A, %AL`
  - **INDIRETTO**: La locazione di memoria ha indirizzo contenuto nel registro *DP*.  
**Esempio:** `OPCODE (%DP), %AL`
- **INDIRIZZAMENTO DELLE PORTE DI I/O**: Le porte di I/O si indirizzano in modo DIRETTO, specificando l'offset della porta nell'istruzione.  
**Esempio:** `IN 0x1010, %AL` oppure `OUT %AL, 0x9F10`

## Fase di fetch vs. Fase di exec

Per noi umani è importante il tipo di operazione, l'OPCODE, mentre per il processore è importante la posizione degli operandi:

```
MOV %AH, %AL  
MOV $0x10, %AL  
MOV (%DP), %AL
```

Queste 3 operazioni hanno fasi di fetch differenti, perché nella prima la cpu ha già specificato il *sorgente* nel registro *AH*; nella seconda deve leggere in memoria; mentre nella terza deve leggere in memoria, all'indirizzo specificato nel registro *DP*.

Quindi sono 3 cose differenti che deve fare!

Mentre la fase di exec di queste 3 è la stessa! Perché a quel punto, una volta che ha l'operando *sorgente*, deve solo metterlo in *AL*!

In sintesi, **hanno fasi di fetch differenti perché gli operandi sono indirizzati in maniere differenti, mentre hanno fasi di exec uguali perché l'opcode è lo stesso.**

**Altro esempi:**

- `MOV %AH, %AL` e `CMP %AH, %AL` hanno fetch uguali ed exec diverse
- `MOV %AH, %AL` e `MOV %AL, indirizzo` hanno fetch diverse ma exec uguali



**Due istruzioni hanno fasi di fetch uguali (diverse) se gli operandi sono indirizzati con modalità di indirizzamento uguali (diverse).**

**Due istruzioni hanno fasi di exec uguali (diverse) se l'OPCODE è lo stesso (diverso).**

**Ciasuna istruzione macchina è lunga almeno un byte!**

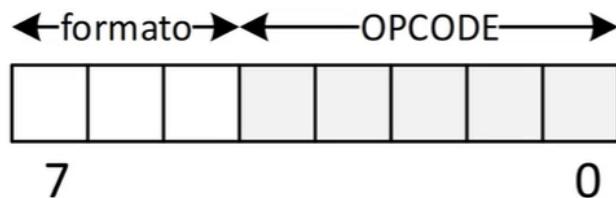
**Il primo byte di ogni istruzione codifica:**

- Il **FORMATO DELL'ISTRUZIONE**, cioè il **modo con cui si devono recuperare gli operandi**, che è rilevante nella fase di fetch
- Il **TIPO DI OPERAZIONE** (opcode), che è rilevante nella fase di exec

Dunque, le istruzioni conviene dividerle in **FORMATI**, e non in **TIPI**!

I **formati nel processore sEP8 sono 8**, come il numero, quindi:

- I primi 3 bit codificano il **FORMATO**
- Gli altri 5 bit codificano il **CODICE OPERATIVO** (32 possibili opcode)



## Formati

- **FORMATO F0 (000): Indirizzamento di Registro <1 byte>**  
Formato per tutte le istruzioni per le quali il processore non deve fare nulla per procurarsi gli operandi, perché:
  - Gli operandi sono Registri
  - L'istruzione non ha operandi (le uniche sono `HLT`, `NOP`, `RET`)

Dunque le istruzioni di questo formato sono un solo *byte*.

La fase di fetch corrisponde con la lettura di questo *byte*, all'indirizzo puntato da *IP*.

HLT	00000000
NOP	00000001
MOV AL, AH	00000010
MOV AH, AL	00000011
INC DP	00000100
SHL AL	00000101
SHR AL	00000110
NOT AL	00000111
SHL AH	00001000
SHR AH	00001001
NOT AH	00001010
PUSH AL	00001011
POP AL	00001100
PUSH AH	00001101
POP AH	00001110
PUSH DP	00001111
POP DP	00010000
RET	00010001

- **FORMATO F1 (001): FORMATO “ANOMALO”! <1 byte>**

Formato per:

- Le **istruzioni dello spazio di I/O per le quali si deve prelevare in memoria l’indirizzo (a 16 bit!) della porta di I/O sorgente/destinatario.**
- Le **istruzioni `MOV` che hanno come operando uno dei registri a 24 bit *DP o SP*.**

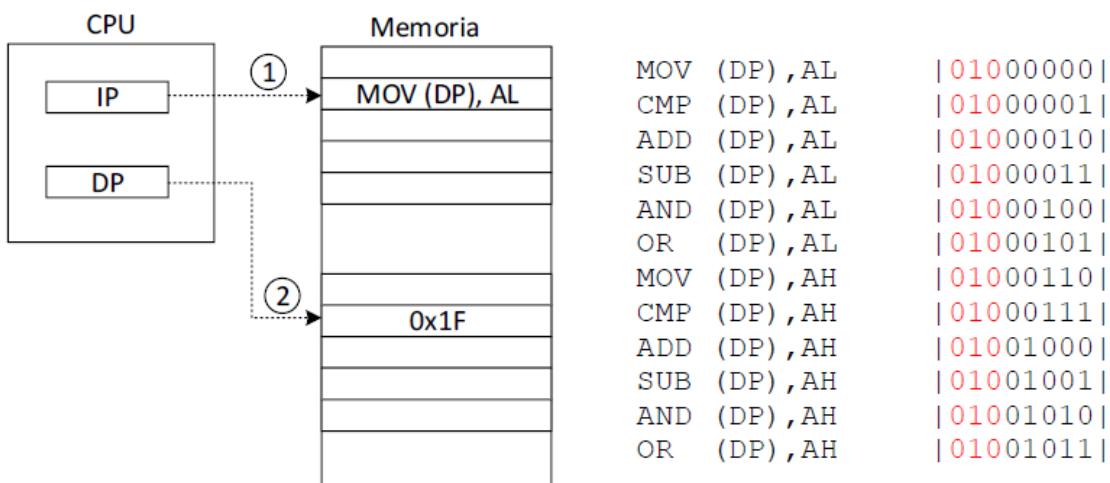
**Contrariamente a quanto si fa per tutti gli altri formati**, visto che questo è il formato di scarto dove ci finisce tutta la peggio roba, **nella fase di fetch l’unica cosa che si fa è prelevare l’opcode, cioè leggere 1 byte!**

Il “problema” di questo formato è che al suo interno ci sono operandi che hanno dimensione diversa da quelle che sono descritte negli altri formati!

IN offset,AL	00100000	offset	
OUT AL,offset	00100001	offset	
MOV \$operando,DP	00100010	operando	
MOV \$operando,SP	00100011	operando	
MOV indirizzo,DP	00100100	indirizzo	
MOV DP,indirizzo	00100101	indirizzo	

### Descrizione di ciascuna:

- `IN offset, %AL` per questa si devono leggere 3 byte e andare nello spazio di I/O a leggere all'indirizzo *offset* per procurarsi il *sorgente*
- `OUT %AL, offset` qui si deve leggere 3 byte, e mi tengo i 2 del *destinatario* da parte perché poi in fase di exec dovrò fare una *write*
- `MOV $operando, DP/SP` qui si deve leggere 4 byte, perché *operando* sta su 3 byte, e non 1 come facevo in *F4*!
- **FORMATO F2 (010): Indirizzamento di Memoria Indiretto (sorgente) <1 byte>**  
**Formato per le istruzioni in cui il sorgente si trova in memoria, indirizzato in modo indiretto tramite il registro puntatore DP.**  
Tutta l'istruzione sta su un solo byte, come nel Formato *F0*, ma la fase di fetch è molto diversa da quella del Formato *F0*!  
La fase di fetch prevede un accesso in memoria all'indirizzo puntato da *DP*, per leggere l'operando *sorgente*, le questa lettura deve essere di 1 byte.



- **FORMATO F3 (011): Indirizzamento di Memoria Indiretto (destinatario) <1 byte>**

**Formato per le istruzioni in cui l'operando *destinatario* si trova in memoria, indirizzato in modo indiretto tramite il registro puntatore *DP*.**

Anche qui, come in *F2*, tutta l'istruzione sta su 1 *byte*.

Ci sono solo due istruzioni in questo formato ⊕:

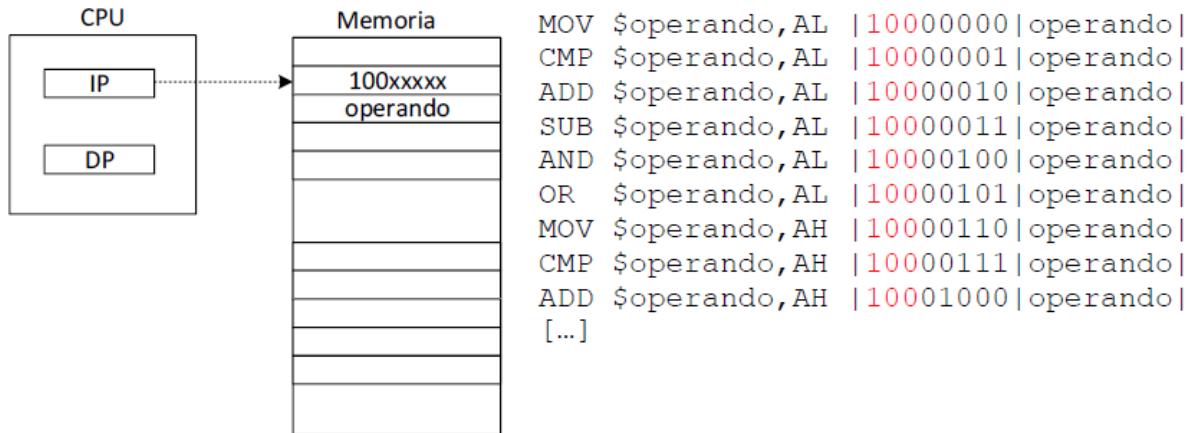
```
MOV AL, (DP) | 01100000 |
MOV AH, (DP) | 01100001 |
```

- **FORMATO F4 (100): Indirizzamento Immediato [SOLO sorgente!] <2 byte>**

**Formato per le istruzioni in cui l'operando *sorgente* è indirizzato in modo immediato, e sta su 8 bit.**

Quindi, l'istruzione sta su 2 *byte*, uno per formato+opcode, l'altro per il *sorgente*.

Quindi, la fase di fetch dovrà leggere 2 *byte* in memoria ad indirizzi consecutivi puntati da *IP*.



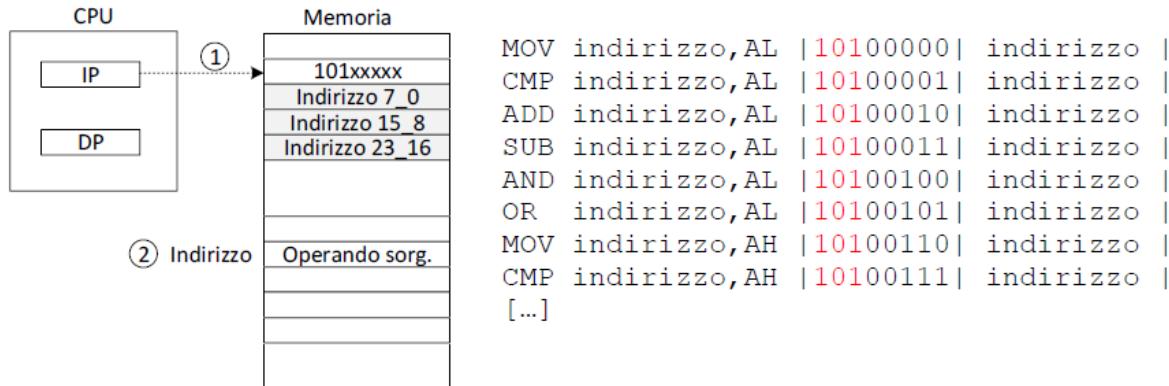
- **FORMATO F5 (101): Indirizzamento di Memoria Diretto (sorgente) <4 byte>**

**Formato per le istruzioni in cui l'operando *sorgente* è indirizzato in modo diretto.**

Quindi, l'istruzione sta su 4 *byte*, uno di formato+opcode, e tre di indirizzo di memoria!

Quindi, la fase di fetch dovrà leggere in memoria 4 *byte* (uno di

formato+opcode, altri 3 di *indirizzo*), a indirizzi consecutivi puntati da *IP*, similmente al formato *F4*, e, ora che ha l'indirizzo a cui è locato *sorgente*, dovrà andare a leggerlo in memoria!

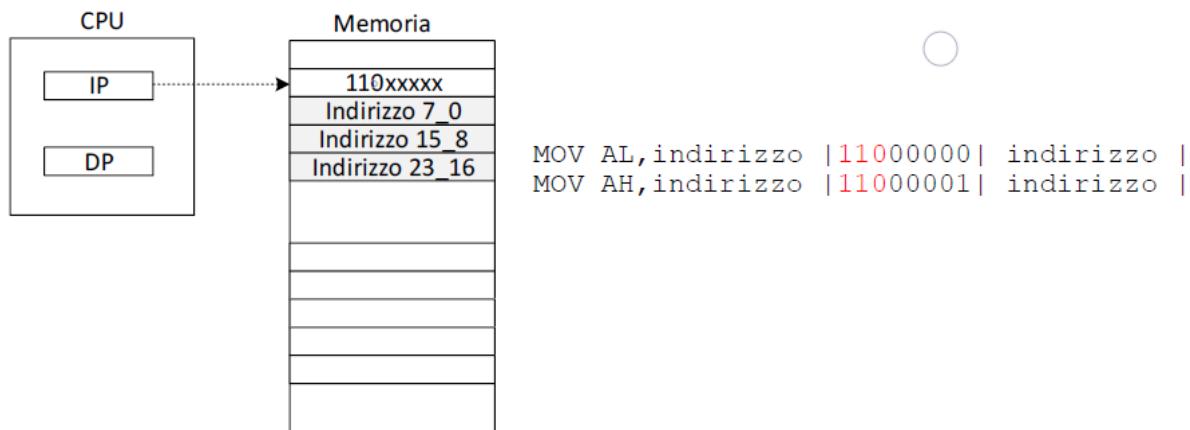


- **FORMATO F6 (110): Indirizzamento di Memoria Diretto (destinatario) <4 byte>**

**Formato per le istruzioni in cui l'operando *destinatario* è in memoria, indirizzato in modo diretto.**

Quindi, anche qui, come in *F5*, l'istruzione sta su 4 *byte*.

Quindi, la fase di fetch legge 4 *byte* in memoria, per procurarsi l'indirizzo del *destinatario*, a locazioni consecutive puntate da *IP*, proprio come nel formato *F5*.

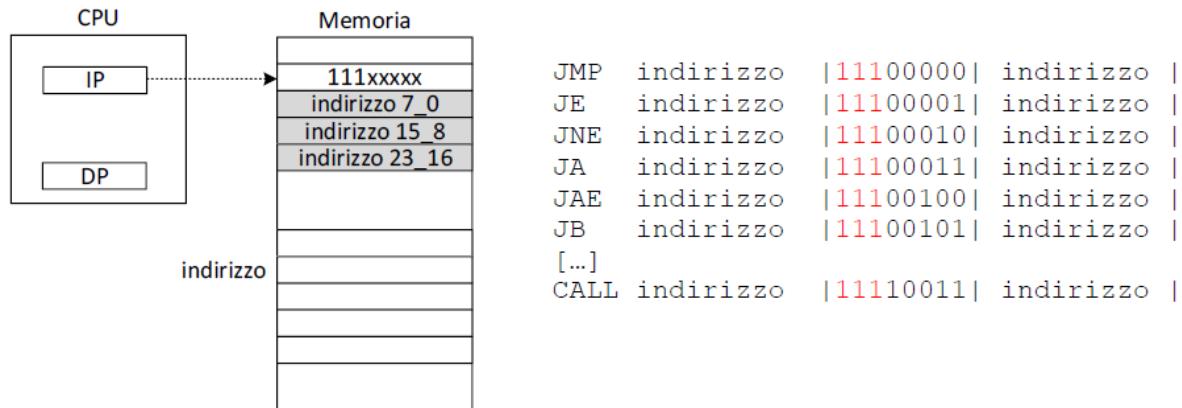


- **FORMATO F7 (111): Indirizzamento di Memoria Diretto (istr. Controllo) <4 byte>**

**Formato per le istruzioni di controllo in cui ho un indirizzo di salto ( **CALL**, **JMP**, **Jcond** ), specificato in modo diretto nell'istruzione stessa (quindi su 3**

*byte*).

Quindi, in fase di fetch vanno letti 4 *byte* consecutivi, puntati dall'indirizzo *IP*.



Questi formati passano da essere difficili da ricordare a facili notando che possono essere divisi in 3 "gruppi" a seconda della codifica in base 2 del numero del formato stesso.

Avendo il numero del formato e ricordandosi il numero di *byte* occupati da questi 3 gruppi di formati si riesce benissimo a ricavare tutto il resto, ricordandosi ben poco.

- $F0 \dots F3 \rightarrow 000 \dots 011 \rightarrow 1 \text{ byte}$
- $F4 \rightarrow 100 \rightarrow 2 \text{ byte}$
- $F5 \dots F7 \rightarrow 101 \dots 111 \rightarrow 4 \text{ byte}$

**Esempio (informale):** mi viene chiesto il formato  $F5$ , ragiono così:

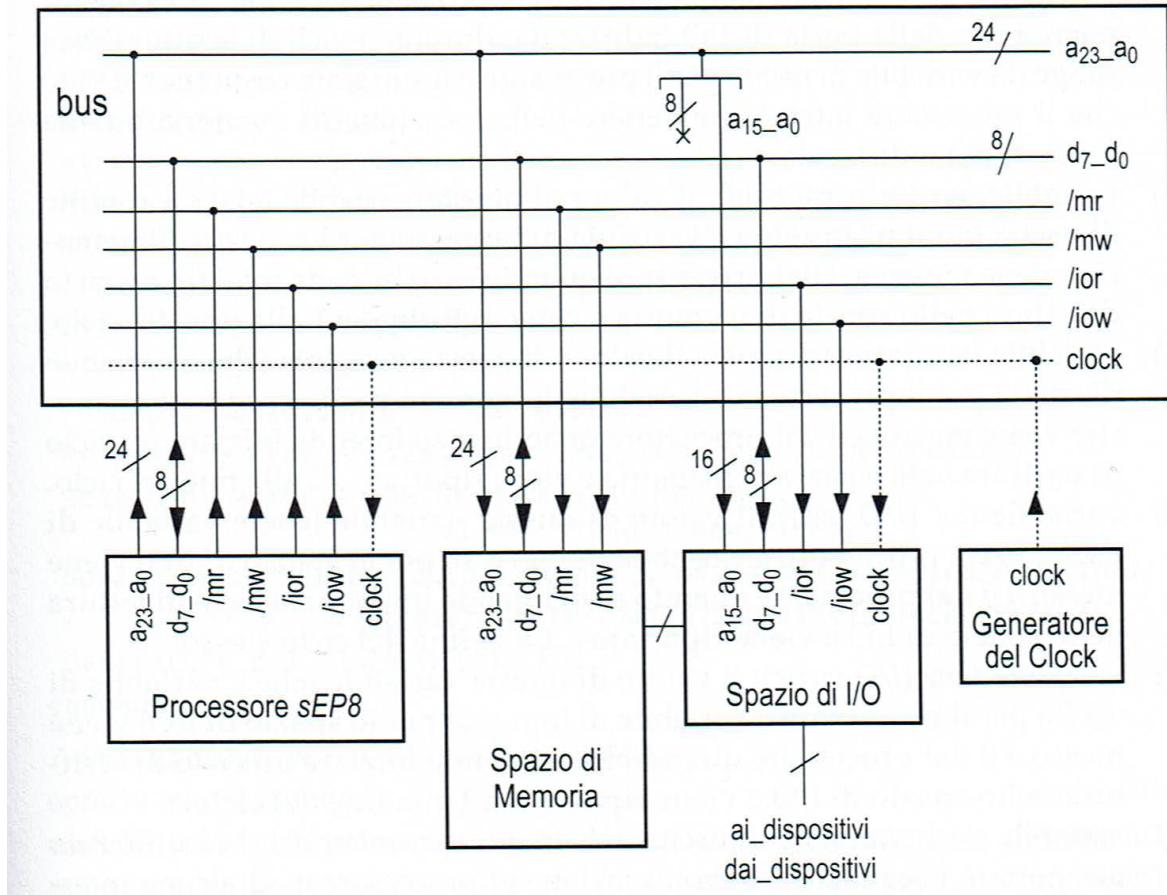
- Mi ricordo che le uniche dimensioni in *byte* che può assumere qualsiasi formato sono 1, 2 oppure 4 e che il formato  $F4$  funge da "spartiacque" fra 1 e 4 *byte*.
- Mi ricordo tutti gli indirizzamenti possibili (importante!).
- So che  $(5)_{10} = (101)_2$ , quindi ricavo intanto che le istruzioni nel formato  $F5$  sono grandi 4 *byte*.
- Se le istruzioni sono grandi 4 *byte*, e nel calcolatore preso in esempio (ma credo più o meno in tutti) gli opcode sono grandi solo 1 *byte*, per forza questo formato dovrà avere operandi!
- Mi ricordo facilmente che:

- I primi formati  $F0 \dots F3$  o non hanno operandi ( $F0$ ), o sono balordi e coinvolgono operandi nello spazio di I/O ( $F1$ ), o hanno operandi indirizzati tramite indirizzamento di memoria indiretto, perché è l'unico modo per far stare queste istruzioni su 1 byte ( $F2, F3$ )
- Il formato  $F4$  è l'unico che ha istruzioni di 2 byte, perché queste sono indirizzate tramite indirizzamento immediato, con cui si può indirizzare SOLO l'operando sorgente, ovviamente! Infatti questo formato è "da solo" e funge da spartiacque.
- I formati  $F5 \dots F7$  sono messi in ordine facile:
  - $F5 \rightarrow$  indirizzamento SORGENTE
  - $F6 \rightarrow$  indirizzamento DESTINATARIO
  - $F7 \rightarrow$  indirizzamento CONTROLLO (che alla fine sono destinatari un pò speciali)

Mettendo tutto questo assieme si riesce a ricordare i formati con facilità, memorizzando il meno possibile e cercando di capire il più possibile che nessi ci sono fra essi!

## **Struttura hardware del calcolatore visto a lezione**

### **Bus dati**



- **FILI DI INDIRIZZO:** Visto che lo spazio di memoria è a 24 bit, servono 24 fili di indirizzo  $a_{23} \dots a_0$ .  
Questi sono uscite per il processore, il quale imposta gli indirizzi delle locazioni di memoria, o delle porte di I/O, dove vuole leggere o scrivere. Sono ingressi quindi per tutte le altre reti.  
Ovviamente, visto che lo spazio di I/O è a 16 bit, alcuni di questi fili dovranno essere buttati via quando il processore vuole fare una *read* o una *write* nello spazio di I/O.
- **FILI DI DATI:** Il processore legge e scrive *byte*, quindi ha bisogno di 8 fili di dato  $d_7 \dots d_0$ .  
Ovviamente, questi devono portare dati da/per tutte le reti, e chiunque può usarli per ricevere o trasmettere dati, quindi dovranno essere forchettati in maniera opportuna!
- **FILI DI CONTROLLO:** Sono tutti attivi bassi, sono uscite per il processore e ingressi per il resto delle reti, come i fili di indirizzo, e sono:
  - $/mr$ : Memory Read, che serve per leggere in memoria

- $/mw$ : **Memory Write**, che serve per **scrivere in memoria**
- $/ior$ : **I/O Read**, che serve per **leggere nello spazio di I/O**
- $/iow$ : **I/O Write**, che serve per **scrivere nello spazio di I/O**
- **SEGNALE DI CLOCK**: Tutte le reti hanno lo stesso clock
- **FILI DI INTERCONNESSIONE TRA INTERFACCE E DISPOSITIVI**
- **(FILI DI COMUNICAZIONE TRA LA MEMORIA VIDEO E L'ADATTATORE GRAFICO)**

Inoltre, **tutte le reti del calcolatore saranno collegate allo stesso circuito di *reset*, e avranno quindi gli stessi piedini di  $/reset$ , così che il tutto si inizializzi coerentemente.**



**ATTENZIONE! Sul bus NON CI SONO I FILI DI SELECT  $/s$ !**

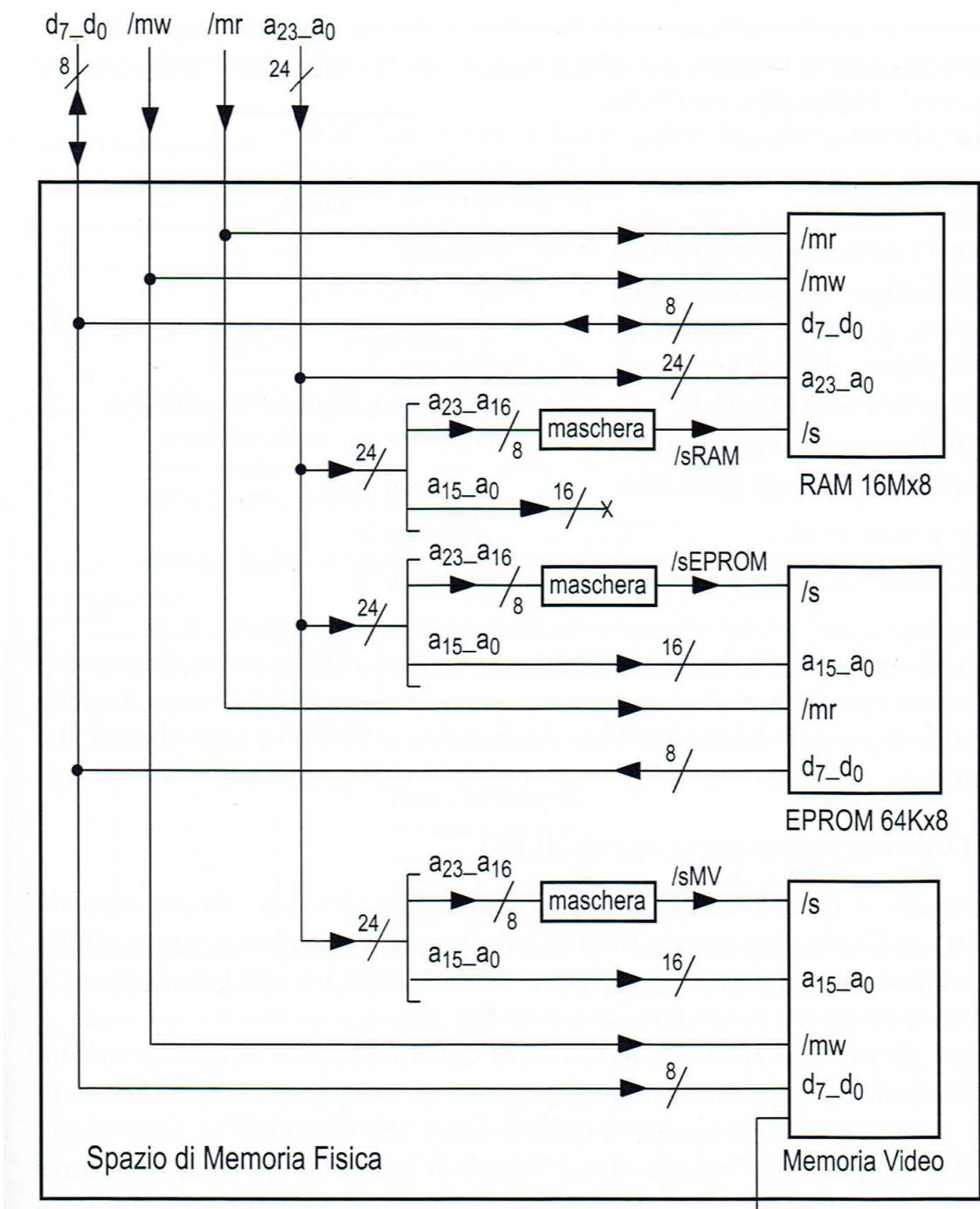
Il segnale di select VIENE CREATO PER OGNI CHIP DI MEMORIA da delle RC dette maschere (già viste nella RAM nelle RSA, appunto) che lo sintetizzano avendo in ingresso i fili di indirizzo  $a_{23} \dots a_0$ !

Altrimenti, se fosse sul bus, a chi dovrebbe andare?



**ATTENZIONE ANCHE CHE IL SEGNALE DI SELECT E' ATTIVO BASSO!**

## Memoria



$a_{23} \ a_{22} \ a_{21} \ a_{20} \ a_{19} \ a_{18} \ a_{17} \ a_{16}$	$/sRAM$	$/sMV$	$/sEPROM$
0 0 0 0 1 0 1 0	1	0	1
1 1 1 1 1 1 1 1 altro	1	1	0
	0	1	1

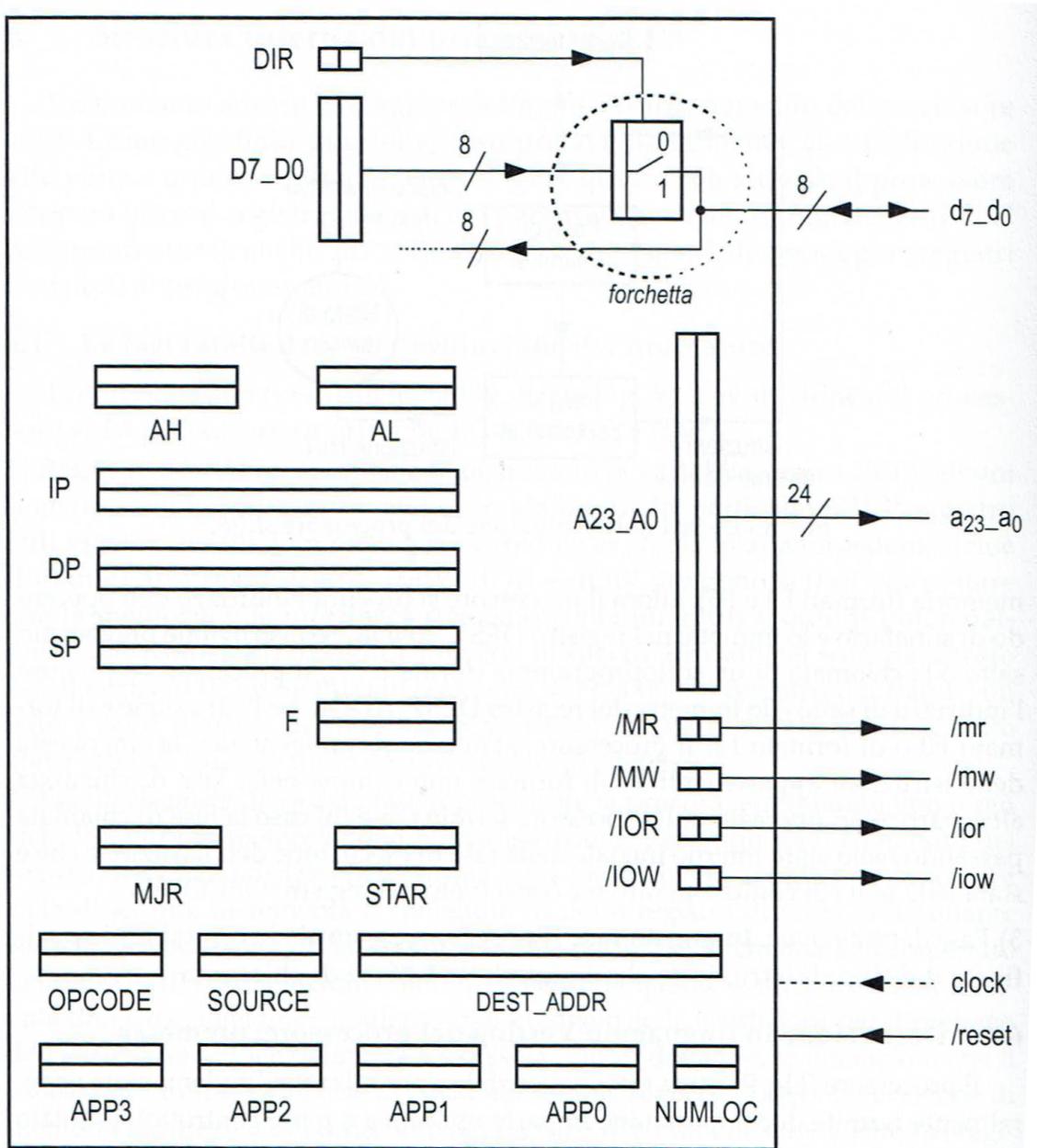
(collegamenti con l'adattatore grafico)

Tabella di Verità delle Maschere

Si monta la EPROM in modo che copra le locazioni ' $HFF0000 - HFFFFFFFFFF$ ',  
la memoria video in modo che copra ' $H0A0000 - H0AFFFFF$ ',

la memoria RAM in modo che copra ' $H000000$  – ' $H09FFFF$  e ' $H0B0000$  – ' $HFEFFFF$

## Processore



Nel processore ci sono vari Registri:

- **STAR**, Registro di Stato, visto che il processore è una RSS!
- **MJR**, per gestire i  $\mu$ -salti a più vie:

- Nella fase di fetch bisogna fare un salto a 8 vie, scegliendo fra gli 8 formati
- Nella fase di exec bisogna fare un salto a  $N$  vie ( $N \approx 50$ ), per decidere quale opcode dell'instruction set del processore *sEP8* eseguire!
- **INSTRUCTION REGISTERS:**  
**Registri in cui viene messo in fase di fetch un operando o un opcode:**
  - *OPCODE*: Contiene l'opcode dell'istruzione da eseguire.
  - *SOURCE*: Contiene l'operando *sorgente*, se sta in memoria.
  - *DEST\_ADDR*: Contiene l'indirizzo dell'operando *destinatario*, se sta in memoria.
- Registri che sostengono le uscite sul bus dati
- *DIR*: Serve per **abilitare la tri-state quando il processore deve effettuare scritture** sia nello spazio di memoria che nello spazio di I/O.  
**Di base sta sempre a 0, tranne quando devo scrivere qualcosa: lo metto ad 1, e quando ho finito lo rimetto a 0!**
- *APPx, NUMLOC*: Registri di appoggio che servono per i cicli di *read/write*

In particolare, al *reset*, le inizializzazioni sono queste:

- *IP* := 'FF0000 (primo indirizzo del blocco di memoria EPROM)
- *F* := 0
- */MR, /MW, /IOR, /IOW* := 1
- *d7\_d0* := 8'BZ (i fili di dato vanno messi in alta impedenza!)
- *DIR* := 0
- *STAR* := *fetch0* (stato definito nella descrizione Verilog del processore sul libro)

Il processore esegue ciclicamente queste fasi:

- **FASE DI FETCH:**

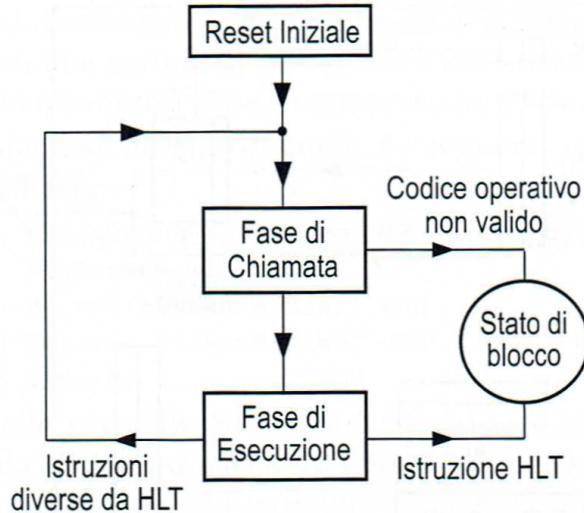
- Preleva un *byte* dalla memoria (formato+opcode), all'indirizzo puntato da *IP*
- Incrementa *IP* (modulo  $2^{24}$ )
- Controlla che quel *byte* che ha prelevato corrisponda all'opcode di una delle istruzioni valide per il suo instruction set. Se non è valida si blocca
- Inserisce il *byte* letto nel Registro *OPCODE*, e
- **Valuta il formato dell'istruzione:** (salto a 8 vie con *MJR*)
  - *F0*: Il processore non deve fare niente, gli operandi sono già nei registri
  - *F1*: Il processore non deve fare niente, il modo (anomalo/speciale) di prelevare gli operandi sarà gestito nella fase di exec
  - *F2, F4, F5*: Procurarsi l'operando *sorgente* a 8 bit e metterlo in *SOURCE*
  - *F3, F6, F7*: Procurarsi l'indirizzo dell'operando *destinatario* e metterlo in *DEST\_ADDR*

Nei formati *F4* . . . *F7* inoltre bisogna anche incrementare *IP* del numero corretto di *byte*, caso per caso, perché gli operandi si trovano subito dopo l'opcode in memoria!

In altre parole, per continuare il flusso di esecuzione bisogna "saltarli" e procedere da dopo!

- **Valuta l'OPCODE:** (salto a  $\approx 50$  vie con *MJR*)
 

Ora che si sa il formato dell'istruzione, si guarda l'opcode e si capisce quale istruzione si deve realmente eseguire
- **FASE DI EXEC:** Avendo già fatto tutto il grosso del lavoro nella fase di fetch decodificando l'istruzione, qui bisogna solo eseguire l'istruzione.  
Al limite se si esegue la *HLT* il processore si blocca.



L'unica cosa che fa fermare questo ciclo è il prelevare ed eseguire l'istruzione *HLT* che lo inchioda, o non riconoscere l'opcode dell'istruzione prelevata.



Il Registro *IP* contiene la locazione della **PROSSIMA istruzione da eseguire**, sennò si eseguirebbe sempre la stessa.

## Spazio di I/O

### Implementazione di due porte nello spazio di I/O:

Supponiamo di avere nello spazio di I/O 2 interfacce, entrambe hanno 2 porte.

Le due porte della prima si trovano agli offset contigui '*H03C8*, '*H03C9*.

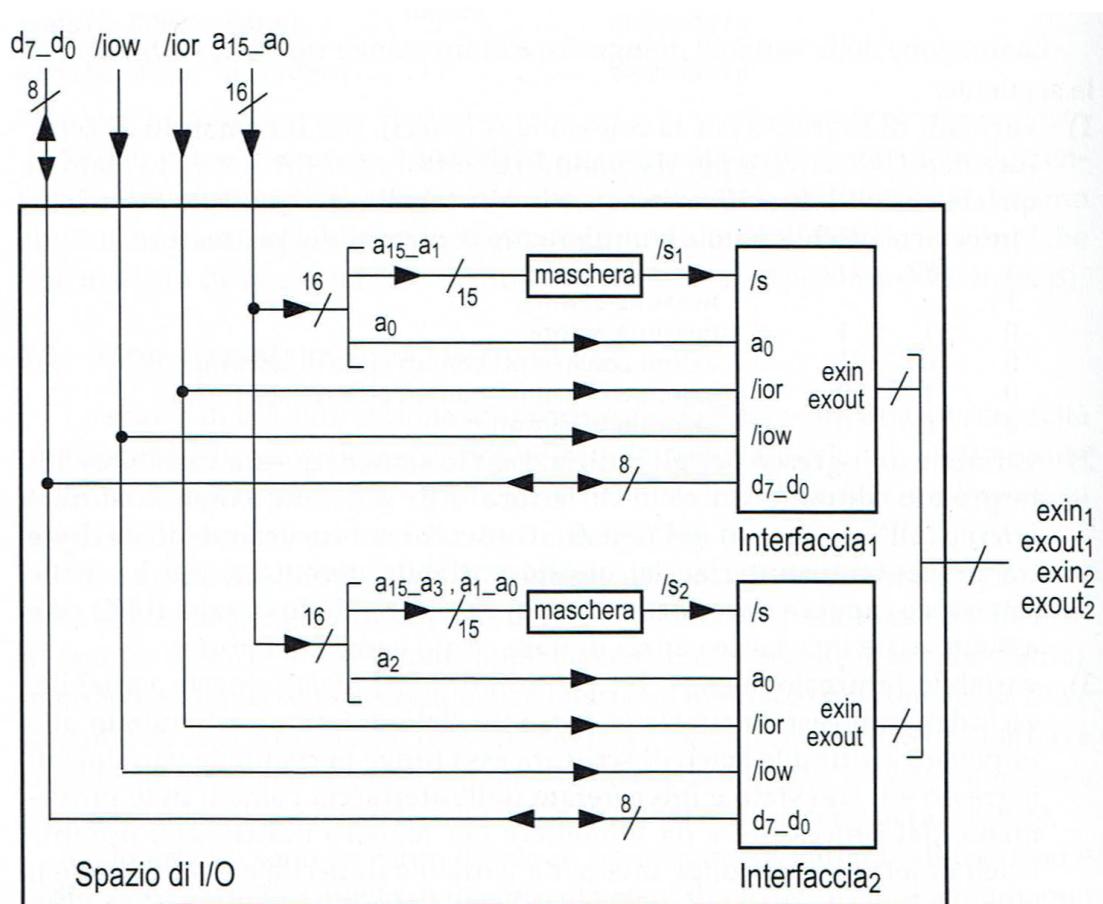
Le due porte della seconda si trovano agli offset (non contigui) '*H0060*, '*H0064*.

Come genero il segnale di select per entrambe?

- Visto che le prime due sono contigue saranno agli indirizzi  $(a_{15} \dots a_1 0)$ ,  $(a_{15} \dots a_1 1)$
  - Le seconde due invece, saranno agli indirizzi  $(a_{15} \dots a_3 0 a_1 a_0)$ ,  $(a_{15} \dots a_3 1 a_1 a_0)$
- Spiegazione:**  $(a_2 \ a_1 \ a_0) = (100)_2 = 4$



**La parte STATICÀ/UGUALE degli indirizzi VA NELLA MASCHERA  
La parte che CAMBIA degli indirizzi va nel modulo come FILI DI  
INDIRIZZO**

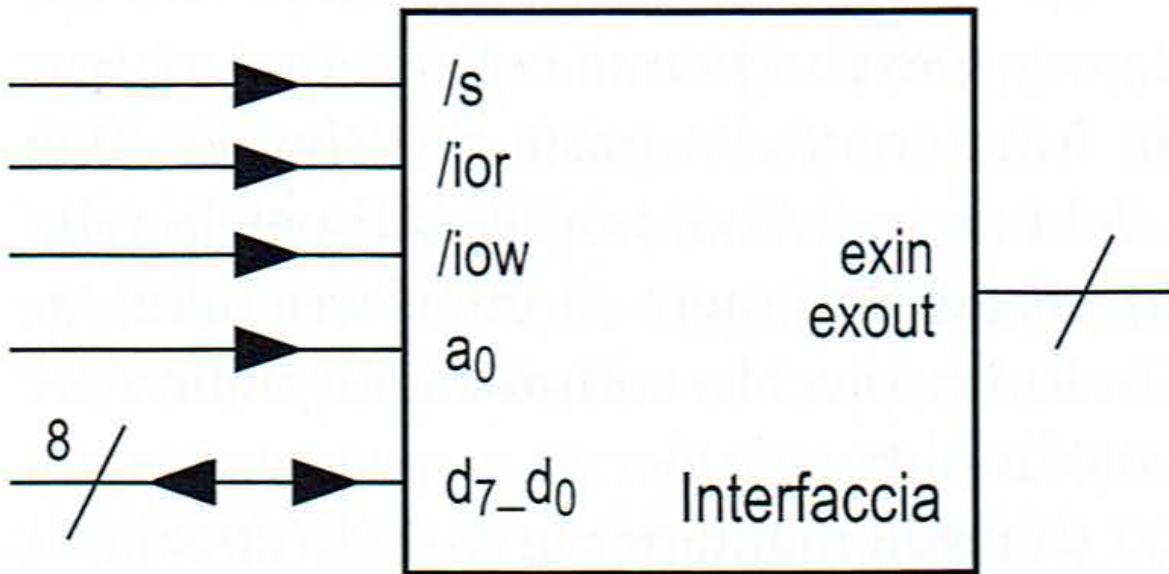


Visione logica dello spazio di I/O

**Lo Spazio di I/O è realizzato fisicamente tramite interfacce, che fungono da raccordo tra il bus dati e i dispositivi di I/O.**

Quindi un'interfaccia fa da tramite fra i due, e ha collegamenti da entrambe le parti: lato bus, sono uguali a quelli di una memoria RAM.

Le **locazioni** che si trovano nelle interfacce prendono il nome di **PORTE DI I/O**, o **REGISTRI DI INTERFACCIA**.



Visione funzionale di un'interfaccia generica

**Il vantaggio delle interfacce** è che:

- Lato device/dispositivo saranno dipendenti dal device specifico
- Lato bus saranno **TUTTE IDENTICHE**, e questo permette al processore di trattarle tutte alla stessa maniera, colloquendo con qualsiasi dispositivo esterno di I/O (e.g. mouse, tastiera, HDD, schermo, ...) **nello stesso modo**, delegando la gestione della "conversione" e del dispositivo in sé **all'interfaccia!**

**Perché non si attaccano direttamente i dispositivi al bus?**

- I dispositivi hanno **velocità diverse fra loro**, e molto spesso sono più lenti del processore. Avendo le interfacce di mezzo, la temporizzazione è quella di una scrittura/lettura di interfacce, la stessa per tutte, e più veloce!
- I dispositivi hanno **modalità di trasferimento dati molto diverse fra loro**. Alcuni trasmettono un bit alla volta (seriali), altri gruppi di bit insieme. Visto che il processore dovrebbe "accomodarsi" ad ogni esigenza di ogni dispositivo, **si mette di mezzo un'interfaccia che renda comodo scrivere e prelevare dati, NELLO STESSO MODO PER TUTTI, da ogni dispositivo, senza fargli perdere tempo o farlo smattare!** 😊

## Differenze nelle temporizzazioni RAM vs. I/O:

- Nella RAM si può leggere o scrivere in qualsiasi locazione.  
Nello spazio di I/O solo nelle locazioni che supportano un Registro di Interfaccia!  
Inoltre alcuni Registri si possono solo leggere, ed altri solo scrivere!
- **Se un'interfaccia implementa una sola porta (quindi un solo Registro di Interfaccia) non sono necessari i fili di indirizzo, ma basta solo il select /s!**



### Differenze nei cicli di scrittura RAM vs I/O:

- RAM: Indirizzi a 24 bit, si usano /mr e /mw
- I/O: Indirizzi a 16 bit, si usano /ior e /iow

Ciclo Lettura I/O → PRIMA metto a posto gli indirizzi, POI do il comando di lettura

## Ciclo di lettura (di un byte) nello spazio di memoria

```
// Ciclo di lettura in memoria RAM dal punto di vista del processore
mem_r0: begin
    A23_A0 <= un_indirizzo;
    DIR <= 0;
    MR_ <= 0;
    STAR <= mem_r1;
end

// Si fa l'ipotesi che non ci voglia uno stato di wait, cioè
// si fa l'ipotesi che la memoria sia veloce!

// Non si creano problemi a mettere gli indirizzi a posto e MR_ a 0
// nello stesso stato perché il dato poi lo memorizzo un clock dopo!
// Inoltre in memoria RAM posso leggere dove mi pare e non succede nulla
mem_r1: begin
    QUALENTE_REGISTRO <= d7_d0;
    MR_ <= 1;
end
```

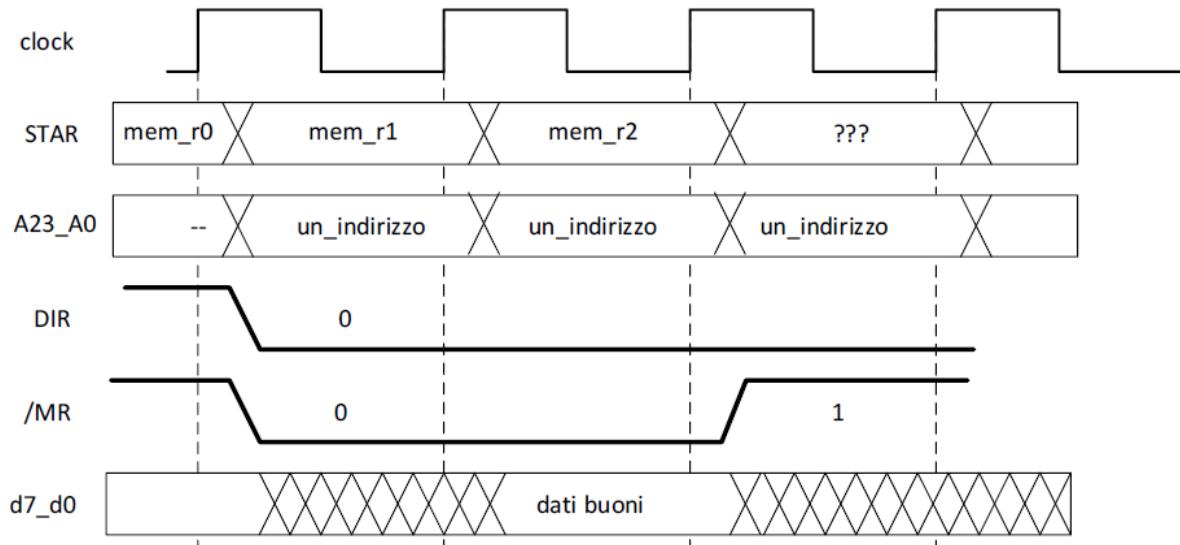
```
// Pure qui non si creano problemi perché campiono il valore di d7_d0
// del clock precedente!
```

**Posso campionare  $d_7 \dots d_0$  in mem\_r1?** Si, perché metto in *QUALCHE\_REGISTRO* il valore che  $d_7 \dots d_0$  aveva PRIMA DEL CLOCK, e poi DOPO IL CLOCK metto *MR\_* a 1!

**Posso portare *DIR* a 1 in mem\_r1?** No, perché c'è il rischio che in un piccolo transitorio siano *DIR* = 1 (cioè uso i fili di dato mettendo in conduzione le tri-state perché sto scrivendo) e *MR\_* = 0 (cioè sto facendo un ciclo di lettura, quindi le tri-state non sono in conduzione perché sto leggendo)

**Se volessi fare letture multiple:**

```
mem_r1: begin
    QUALCHE_REGISTRO <= d7_d0;
    MR_ <= MR_; // TENGO MR_ a 0 perché continua il ciclo di lettura!
    A23_A0 <= nuovo_indirizzo;
    STAR <= (quando finire le letture) ? Spoi : mem_r1;
end
```



Temporizzazione del ciclo di lettura nello spazio di memoria (qui però mem\_r1 è lo stato di wait)

## Ciclo di scrittura (di un byte) nello spazio di memoria

```

// Ciclo di scrittura in memoria RAM dal punto di vista del processore
mem_w0: begin
    A23_A0 <= un_indirizzo;
    D7_D0 <= un_byte;
    DIR <= 1;
    STAR <= mem_w1;
end

mem_w1: begin
    MW_ <= 0;
    STAR <= mem_w2;
end

mem_w2: begin
    MW_ <= 1; // Comando di memorizzazione
    STAR <= mem_w3;
end
// Indirizzi, dati e DIR devono stare fermi in mem_w2!

mem_w3: begin
    DIR <= 0;
end

```

**Posso riportare  $DIR$  a 0 in  $mem\_w2$ ? No, perché sennò i dati  $d_7 \dots d_0$  non sarebbero stabili sul fronte di salita di  $/mw$ !**

**Perchè il fronte di salita di  $/mw$  è il comando di memorizzazione!**

**Posso mettere subito  $MW_$  a 0 in  $mem\_w0$ ?** No, perché la scrittura è distruttiva, e devo aspettare prima che sia i fili di indirizzo che il segnale di select  $/s$  siano andati a posto!

Poi i dati possono ballare fino al fronte di salita di  $/mw$ , non importa che siano pronti subito, perché il comando di scrittura in memoria è il fronte di salita di  $/mw$ !

(Il quale corrisponde al riportare in conservazione la riga di D-Latch selezionata nella RAM)

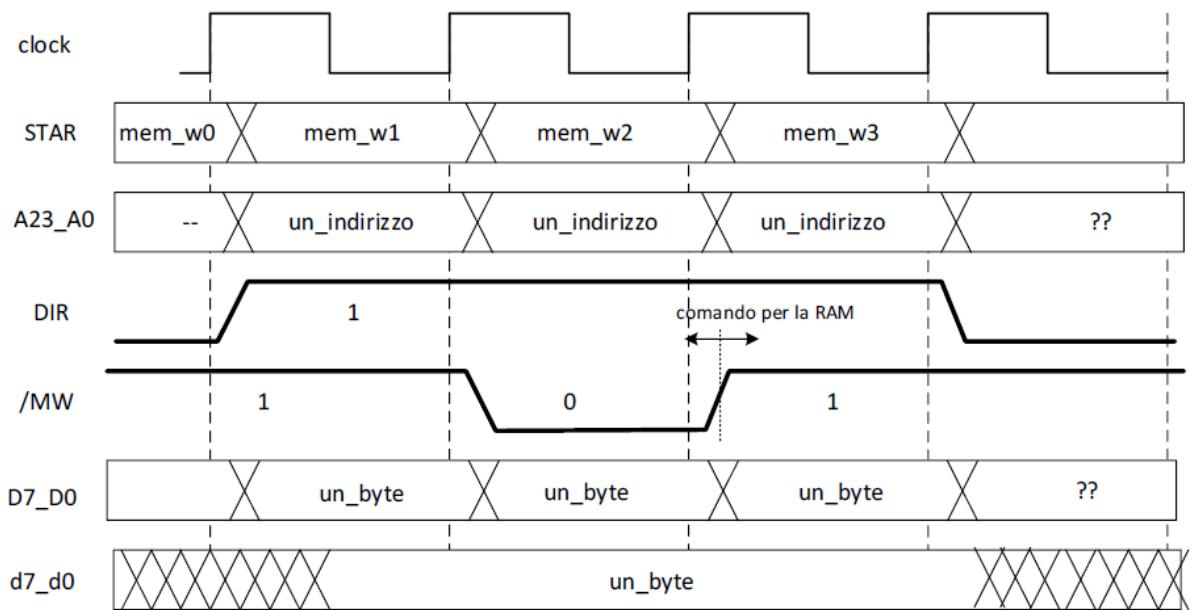
**Posso riassegnare  $A23\_A0$  in  $mem\_w2$ ?** No, per lo stesso motivo di sopra in rosso

**Posso portare  $D7\_D0 <= un\_byte;$  in  $mem\_w1$ ?** Si, per il motivo di sopra in rosso

**Se volessi fare scritture multiple:**

Non si può fare come per la lettura, perché se sovrascrivo i fili di indirizzo, sto mandando in trasparenza dei D-Latch, senza accorgermene!

Allora le scritture multiple vanno fatte una alla volta, senza scorciatoie!



Temporizzazione del ciclo di scrittura nello spazio di memoria

### Note:



#### Fare attenzione a *DIR*:

- Va messo a 1 UN CLOCK PRIMA di mettere  $MW\_ \leq 0$
- Va rimesso a 0 UN CLOCK DOPO aver rimesso  $MW\_ \leq 1$



#### Fare attenzione alla temporizzazione:

Inoltre gli indirizzi `A23_A0` e i dati `d7_d0` devono stare FERMI (non oscillare) quando rimetti  $MW\_ \leq 1$ , cioè **devi assegnare `A23_A0` e `d7_d0` in uno Stato Interno PRECEDENTE a quando rimetti  $MW\_ \leq 1$ !**

## Ciclo di lettura (di un byte) nello spazio di I/O

Il ciclo di lettura in RAM si fa in 2 clock, quello nello spazio di I/O in 3, perché qui gli indirizzi devono essere pronti un clock prima del comando di lettura (cioè `IOR_ <= 0`).

Questo perché leggere dati da un interfaccia (handshake FI/FO) è il segnale che l'interfaccia è pronta per essere riscritta!

Quindi gli indirizzi devono essere pronti PRIMA di dare il comando di lettura, sennò potrei leggere qualche interfaccia a caso facendogli riscrivere il dato!

```
// Ciclo di lettura nello spazio di I/O dal punto di vista del processore
io_r0: begin
    A23_A0 <= {'H00, un_offset}; // *
    DIR <= 0;
    STAR <= io_r1;
end

// PRIMA sistemo gli indirizzi, POI al clock successivo metto IOR_ <= 0

io_r1: begin
    IOR_ <= 0;
    STAR <= io_r2;
end

// Si fa l'ipotesi anche qui che non ci voglia uno stato di wait!

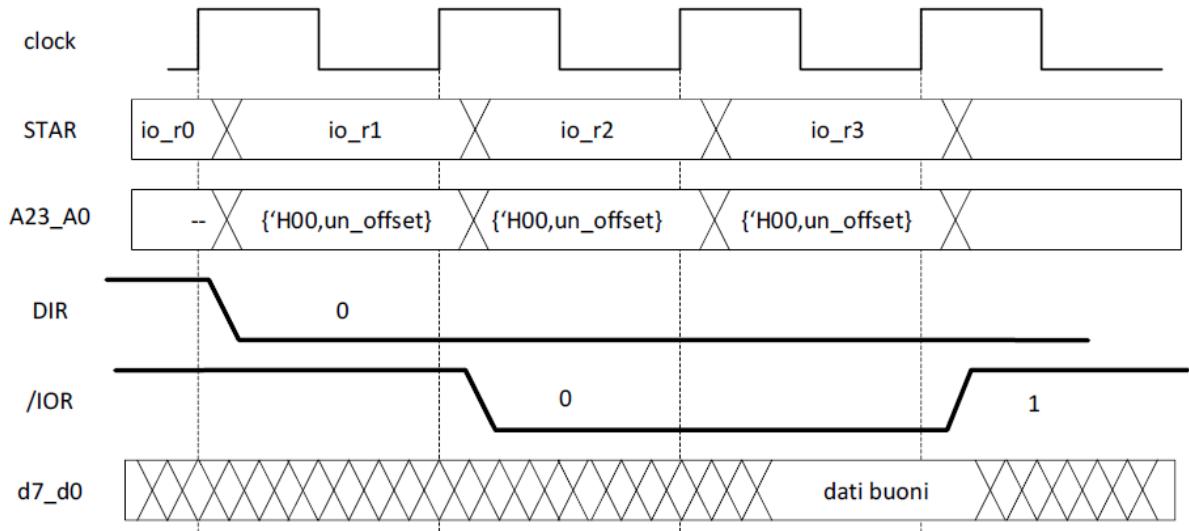
io_r2: begin
    QUALENTE_REGISTRO <= d7_d0;
    IOR_ <= 1;
end
```

\* Perché i fili di indirizzo devono essere estesi? Gli indirizzi della memoria sono a 24 bit, quelli dello spazio di I/O sono a 16. Ma visto che i fili di indirizzo devono essere assegnati tutti in Verilog, e poi si buttano via gli 8 più significativi, bisogna estendere `un_offset`, che è su 16 bit, con qualcosa di casuale (potresti pure mettere `8'BX` o `8'BZ` teoricamente, poi non so se VL te lo fa fare)!

**Posso portare `A23_A0 <= {'H00, un_offset}` in `io_r1`?** No, per il motivo di qui sopra in verde!

**Se volessi fare letture multiple:**

Non puoi farle come nello spazio di memoria! Perché non puoi riassegnare `A23_A0` in `io_r2`, perché ci sarebbe un transitorio in cui potresti leggere un'interfaccia a caso e cambiarne lo stato!



Temporizzazione del ciclo di lettura nello spazio di I/O

## Ciclo di scrittura (di un byte) nello spazio di I/O

Il codice è uguale identico a quello per il ciclo di scrittura nello spazio di memoria, MA:

- La memoria RAM memorizza sul fronte di SALITA di `/mw`
- Lo spazio di I/O memorizza sul fronte di DISCESA di `/iow`

```
// Ciclo di scrittura nello spazio di I/O dal punto di vista del processore
io_w0: begin
    A23_A0 <= {'H00, un_offset};
    D7_D0 <= un_byte;
    DIR <= 1;
    STAR <= io_w1;
end

io_w1: begin
    IOW_ <= 0;
    STAR <= io_w2;
end

io_w2: begin
    IOW_ <= 1;
    STAR <= io_w3;
end

io_w3: begin
```

```

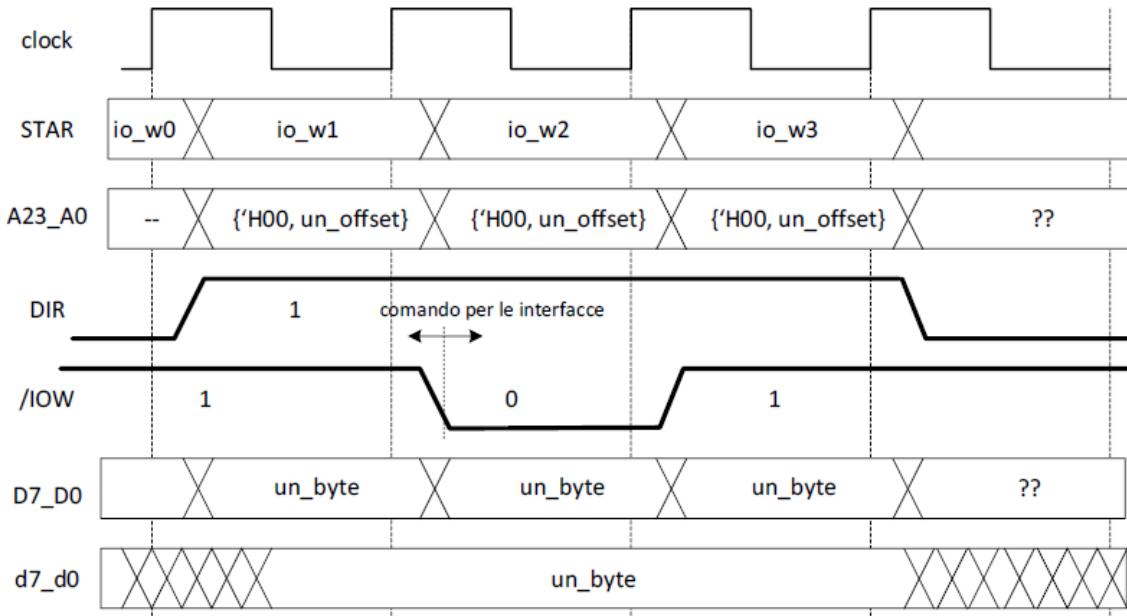
DIR <= 0;
end

```

**Posso spostare `D7_D0 <= un_byte` in `io_w1`?** No, per il motivo detto qui sopra in rosso!

### **Se volessi fare scritture multiple:**

Come nella scrittura normale, no, devi farne una per volta



Temporizzazione del ciclo di scrittura nello spazio di I/O

### **Differenze fra cicli di lettura/scrittura in memoria RAM vs. Spazio di I/O:**

- **LETTURA:**

- **RAM:** Puoi assegnare gli indirizzi (`A23_A0 <= indirizzo`) e dare il comando di lettura (`MR_ <= 0`) **nello stesso stato interno**, tanto leggere cose a caso non è un problema, poi verranno sovrascritte al fronte di salita di `MR_` al quale si finalizza la lettura!
- **I/O:** Devi assegnare gli indirizzi (`A23_A0 <= indirizzo`) **in uno stato interno PRECEDENTE** a quello in cui dai il comando di lettura (`MR_ <= 0`), sennò

VS.

potresti leggere un'interfaccia casuale e cambiarne lo stato!

- **SCRITTURA:**

- **RAM:** Il comando di memorizzazione è `MW_ <= 1`, quindi dati, indirizzi e *DIR* devono stare fermi a cavallo del FRONTE DI SALITA DI *MW\_*!  
VS.  
◦ **I/O:** Il comando di memorizzazione è `IOW_ <= 0`, quindi dati, indirizzi e *DIR* devono stare fermi a cavallo del FRONTE DI DISCESA DI *IOW\_*!

La descrizione Verilog del processore non è riportata in questi appunti!

# Interfacce

## Generalità

Le interfacce sono di vari tipi:

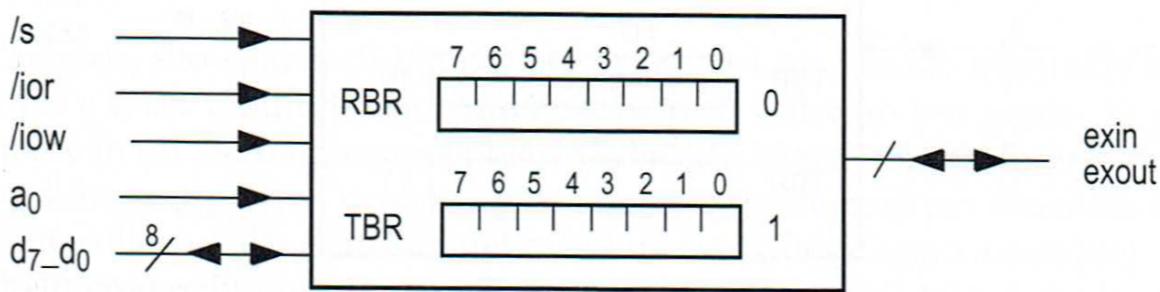
- **PARALLELE**: Inviano e ricevono 1 *byte* per volta dai dispositivi di I/O
- **SERIALI**: Inviano e ricevono 1 *bit* per volta dai dispositivi di I/O
- **CONVERTITORI**: Convertono tensioni in gruppi di *bit* e viceversa

Ovviamente la “caratterizzazione” di un’interfaccia (cioè se è seriale o parallela) si vede lato-dispositivo, perché sono tutte identiche lato bus, è quello il loro scopo!

Tipicamente le interfacce possono avere i seguenti **REGISTRI DI INTERFACCIA**:

- **RECEIVE BUFFER REGISTER (RBR)**: Registro che serve al processore per leggere il dato che il dispositivo ha prodotto
- **TRANSMIT BUFFER REGISTER (TBR)**: Registro che serve al processore per scriverci il dato che l’interfaccia deve mandare al dispositivo

Questi due Registri di Interfaccia sono distinti dal valore del filo di indirizzo  $a_0$ , mentre gli altri fili di indirizzo  $a_{15} \dots a_1$  sono usati per generare il segnale di select  $/s$  per l’interfaccia



Visione funzionale di un’interfaccia

Per leggere il dato memorizzato nell’interfaccia basta fare `IN offset_RBR, %AL`, mentre per scrivere nell’interfaccia un dato, basta fare `OUT %AL, offset_TBR`.

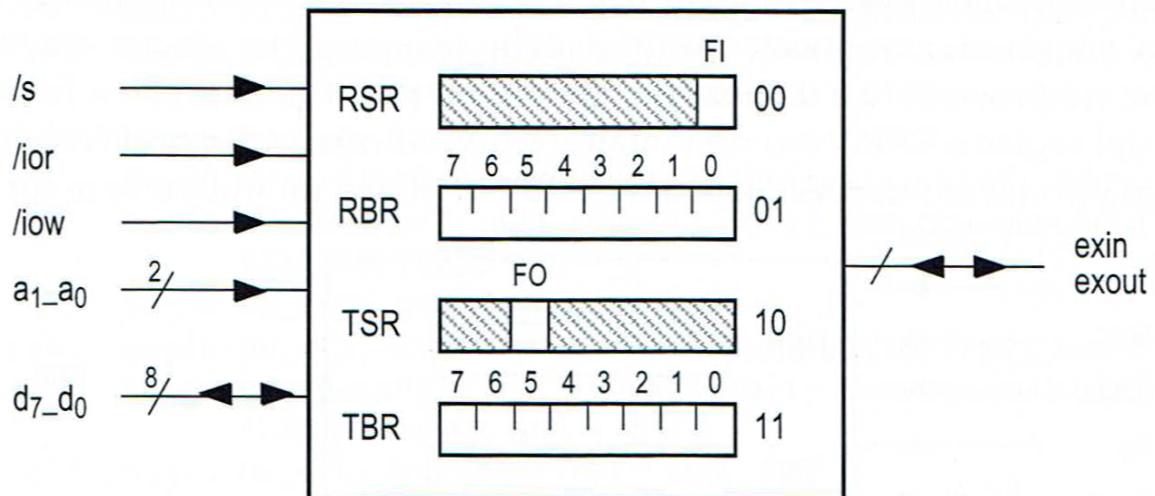
Ma chi mi garantisce che l'interfaccia sia pronta per ricevere un nuovo dato o che il dispositivo sia pronto per ricevere un nuovo dato dall'interfaccia? Nessuno!

Ecco perché si dotano le interfacce di ulteriori registri, chiamati **REGISTRI DI STATO**, che servono a svolgere un **handshake fra processore e dispositivo**, e sono *read – only*:

- **RECEIVE STATUS REGISTER (RSR)**: Registro nel quale il processore può leggere il *flag FI*.
- **TRANSMIT STATUS REGISTER (TSR)**: Registro nel quale il processore può leggere il *flag FO*.

Questi due *flag* sono gli unici *bit* significativi dei due Registri di Stato, e li gestisce l'interfaccia!

Tra l'altro, visto che sono in posizioni diverse, si possono collassare i due Registri di Stato *RSR* e *TBR* in un unico Registro di Stato *RTSR*.

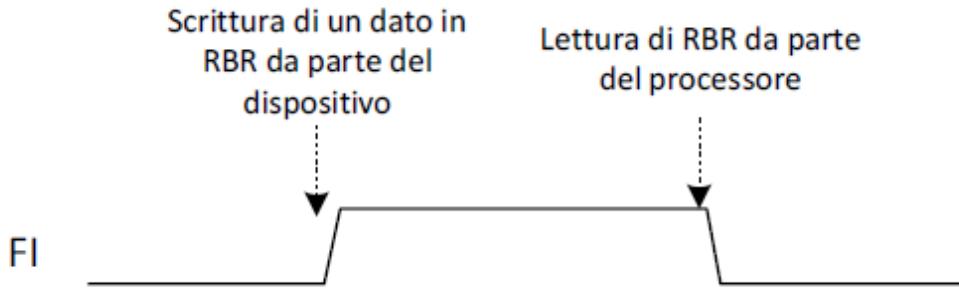


Visto che quest'interfaccia ha 4 Registri ora, e non più solo 2, per distinguerne gli indirizzi interni ora mi servono 2 *bit*:  $a_1$  e  $a_0$ !

## Ingresso dati a controllo di programma



**FI è il FLAG DI INGRESSO PIENO!**



*FI* all'inizio e alla fine è a 0.

L'interfaccia lo mette a 1 quando il dispositivo scrive un dato in *RBR*.

L'interfaccia lo rimette a 0 quando il processore legge il dato in *RBR*.

**Come ricavare questo meccanismo:**

- **Come capire chi scrive il dato:**

*RBR* è di SOLA LETTURA PER IL PROCESSORE!

Quindi **PER FORZA** è il dispositivo che scrive il dato in *RBR*!

- **Come capire chi legge il dato:**

Visto che il dispositivo ha scritto il dato, perché lo può scrivere solo lui, per forza sarà il processore a leggerlo!

- **Come ricordarsi l'andamento (0, 1, 0):**

Ricorda molto uno "spike", tipo una forza impulsiva, la quale ricorda molto una  $\mathcal{I}$ , la stessa che c'è nel nome *FI*

- **Come ricordarsi che il registro è *RBR*:**

*FI* → *input* → *RBR*

- **Come ricordarsi che prima si scrive e poi si legge:**

*FI* → Flag di INGRESSO PIENO → inizialmente è a 0, quindi il processore deve aspettare.

Aspettare cosa? Che il dispositivo **SCRIVA** in *RBR* il dato da poter poi leggere!

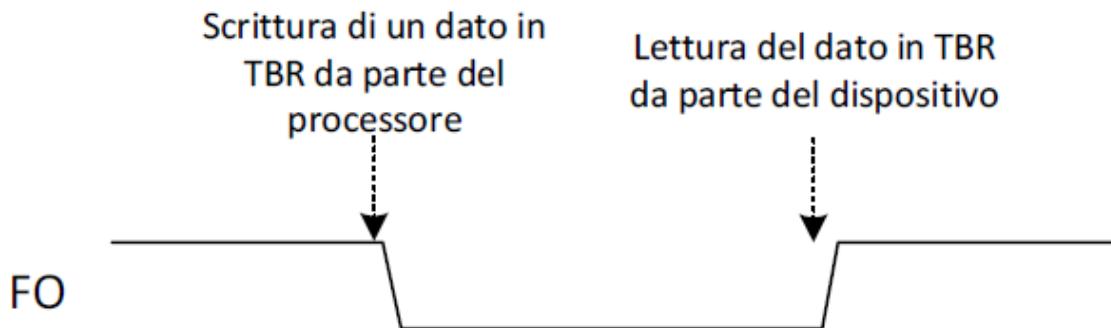
```
// Sottoprogramma per la lettura con ingresso dati a controllo di programma
readFI: IN RSR_offset, %AL
        AND $0x01, %AL      // Azzero tutto AL tranne il flag FI
        JZ readFI           // Ciclo finche FI non passa ad 1
```

```
IN RBR_offset, %AL // Quando FI=1 leggo il nuovo dato  
RET
```

## Accesso a controllo di programma



**FO è il FLAG DI INGRESSO VUOTO!**



*FO* all'inizio e alla fine è a 1.

L'interfaccia lo mette a 1 quando il processore scrive un dato in *TBR*.

L'interfaccia lo rimette a 0 quando il dispositivo legge il dato in *TBR*.

**Come ricavare questo meccanismo:**

- **Come capire chi scrive il dato:**  
*TBR* è di SOLA SCRITTURA PER IL PROCESSORE!  
Quindi PER FORZA è il processore che scrive il dato in *TBR*!
- **Come capire chi legge il dato:**  
Visto che il processore ha scritto il dato, perché lo può scrivere solo lui, per forza sarà il dispositivo a leggerlo!
- **Come ricordarsi l'andamento (1, 0, 1):**  
Ricorda molto una cOnca, la cui forma ricorda molto una *O*, la stessa che c'è nel nome *FO*
- **Come ricordarsi che il registro è *TBR*:**  
*FO* → *output* → *TBR*

- **Come ricordarsi che prima si scrive e poi si legge:**

$FO \rightarrow$  Flag di INGRESSO VUOTO  $\rightarrow$  inizialmente è a 1, quindi il processore può far qualcosa!

Può cioè SCRIVERE in  $TBR$  un dato, che il dispositivo poi leggerà!

```
// Sottoprogramma per la scrittura con accesso a controllo di programma
PUSH %AL                                // Salvo in pila il dato che voglio scrivere
writeFO: IN TSR_offset, %AL
    AND $0x20, %AL                      // Azzero tutto AL tranne il flag FO
    JZ writeFO                           // Ciclo finche FI non passa ad 1

    POP %AL
    OUT RBR_offset, %AL      // Quando FO=1 scrivo il nuovo dato
    RET
```



Sia  $FI$  che  $FO$  partono da un valore che è **I'OPPOSTO** della seconda lettera:

- $FI$  parte da 0 (e finisce a 0)
- $FO$  parte da 1 (e finisce a 1)

Inoltre, in entrambe avviene prima una *write* e poi una *read*!

**Quando uno dei due flag è 1 il processore può far qualcosa:**

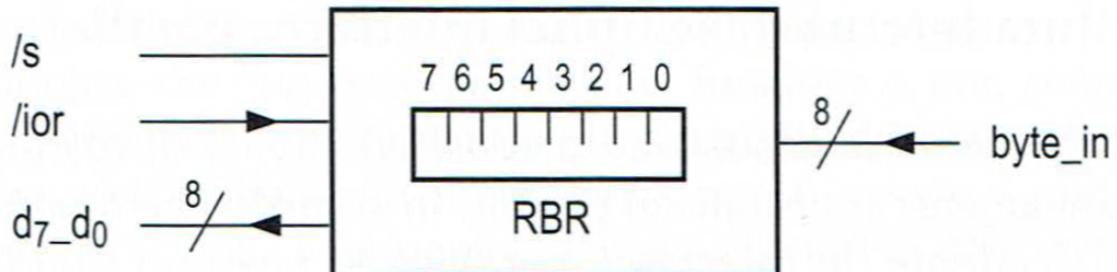
- **Se  $FI = 1$  (flag di INGRESSO PIENO) il processore può LEGGERE (in  $RBR$ )!**
- **Se  $FO = 1$  (flag di INGRESSO VUOTO) il processore può SCRIVERE (in  $TBR$ )!**

## Interfacce parallele senza handshake

## Interfaccia parallela di ingresso

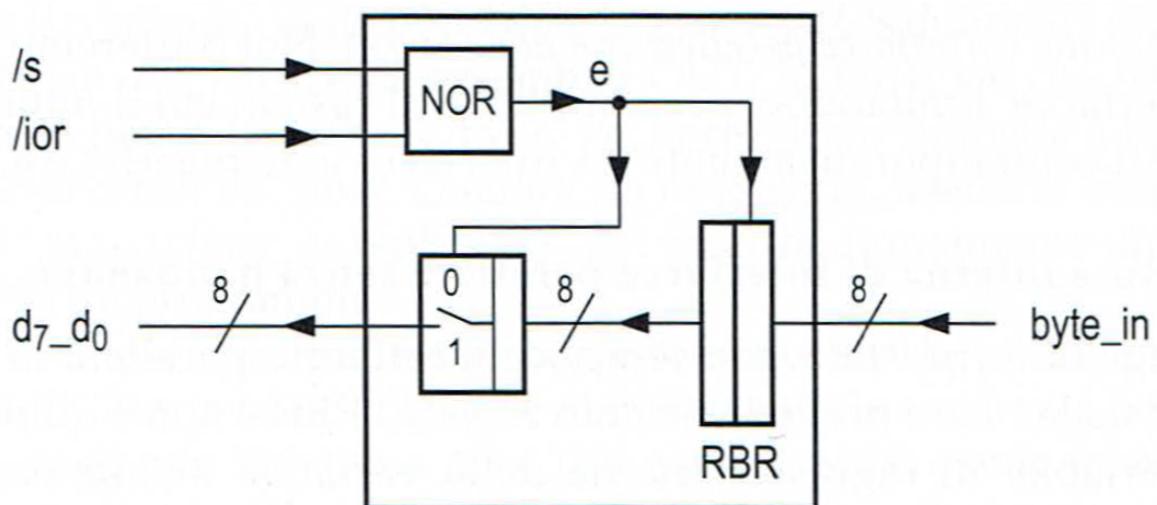
E' un'interfaccia SENZA HANDSHAKE dalla quale il processore può SOLO LEGGERE!

Il dispositivo invia un dato *byte\_in* all'interfaccia, e il processore lo preleva!



Visione funzionale di un'interfaccia parallela di (solo) ingresso

Vediamo come è realizzata fisicamente:



Circuito che implementa un'interfaccia parallela di ingresso

<i>/s</i>	<i>/ior</i>	<i>e</i>
0	0	1
0	1	0
1	0	0
1	1	0

Il segnale *e* deve essere 1 quando:

- L'interfaccia è selezionata: */s* = 0

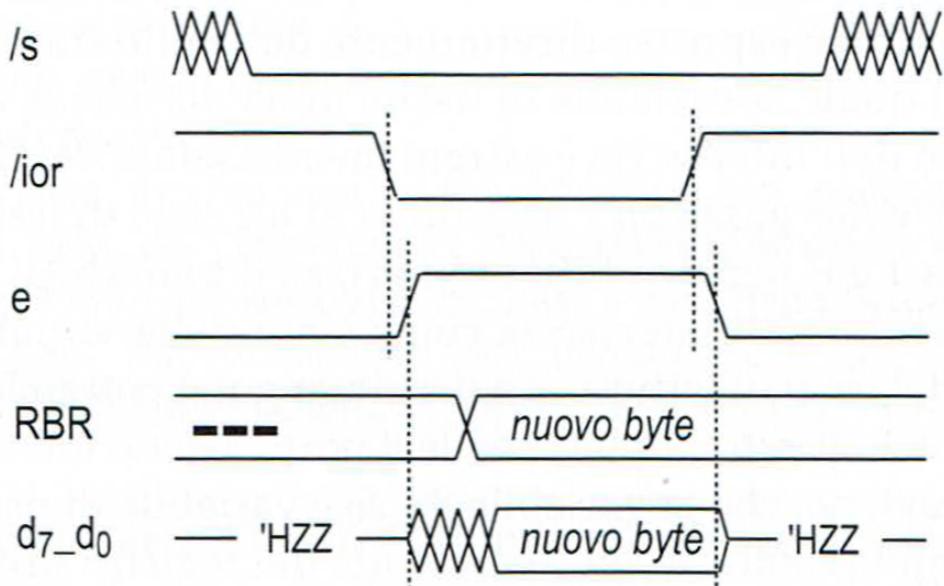
- Viene dato il comando di *read*:  $/ior = 0$

Ovvero, quando questo succede *e* ha un fronte di salita, al quale il registro *RBR* campiona il dato (*byte\_in*) in ingresso dal dispositivo!

Inoltre, il fronte di salita di *e* mette in conduzione le 8 tri-state per collegare l'uscita del registro *RBR* all'uscita dell'interfaccia, così da poterlo mandare al processore!

**Ci vogliono le tri-state perché  $d_7 \dots d_0$  sono i fili di dato che vanno sul bus dati!**

Ora, vediamone la temporizzazione:



Visto che siamo nello spazio di I/O, */s* deve essere pronto (a 0) PRIMA di dare il comando di lettura!

Un'interfaccia parallela di ingresso si descrive così in Verilog:

```
// Descrizione di un'interfaccia parallela di ingresso
module IntParIngresso(s_, ior_, d7_d0, byte_in);
    input s_, ior_;
    input [7:0] byte_in;
    output [7:0] d7_d0;

    reg [7:0] RBR;
    wire e; assign e = ( {s_, ior_} == 'B00 ) ? 1 : 0; // la porta NOR
    assign d7_d0 = (e == 1) ? RBR : 'HZZ;
```

```

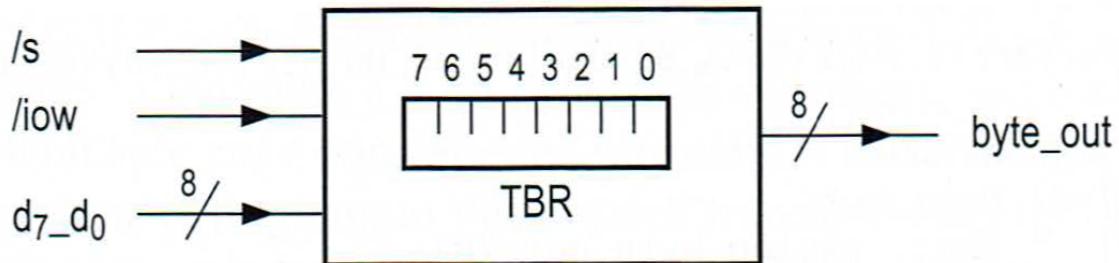
    always @(posedge e) #3 RBR <= byte_in;
endmodule

```

## Interfaccia parallela di uscita

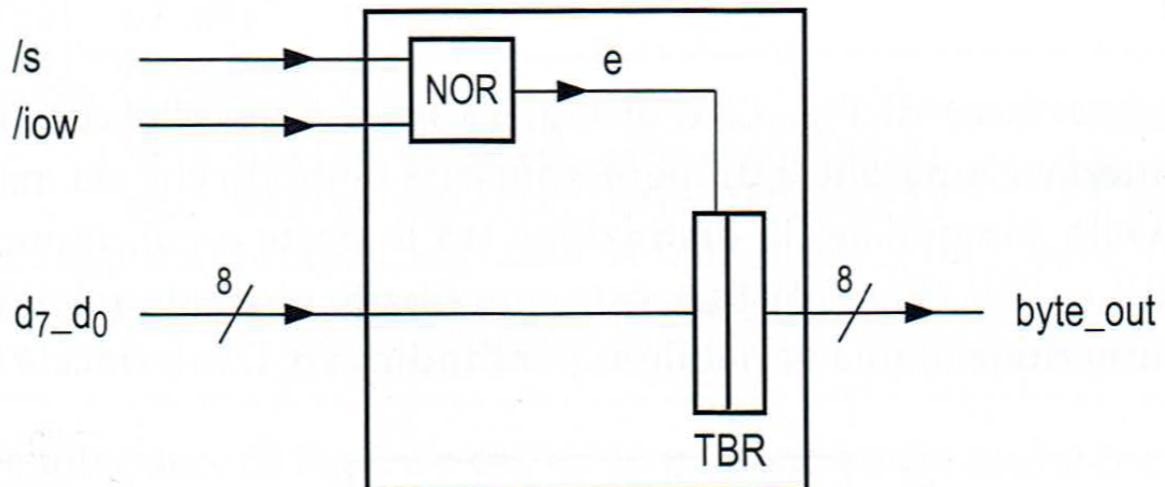
E' un'interfaccia **SENZA HANDSHAKE** nella quale il processore può **SOLO SCRIVERE!**

**Il processore invia un dato  $d_7 \dots d_0$  all'interfaccia, e il dispositivo lo preleva!**



Visione funzionale di un'interfaccia parallela di (sola) uscita

Vediamo come è realizzata fisicamente:



Circuito che implementa un'interfaccia parallela di uscita

$/s$	$/iow$	$e$
0	0	1
0	1	0
1	0	0
1	1	0

Il segnale  $e$  deve essere 1 quando:

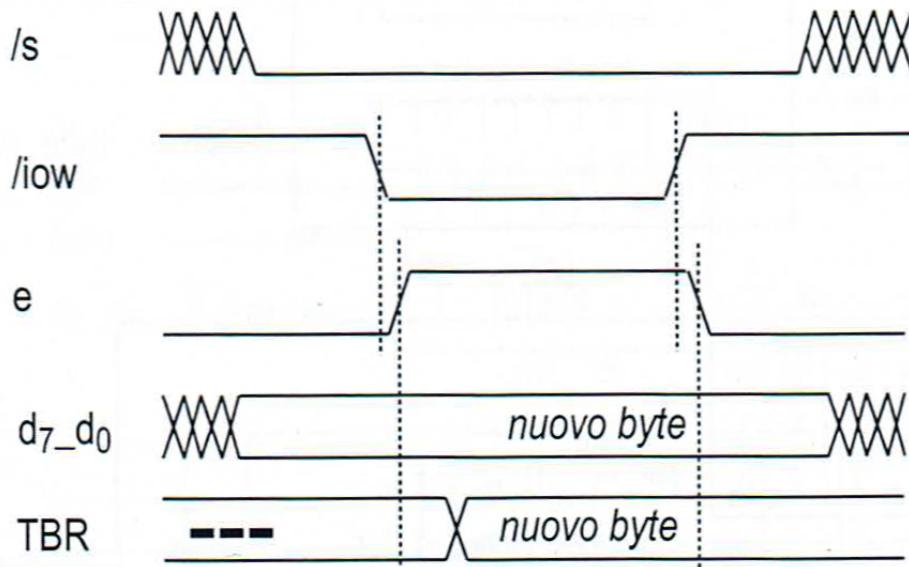
- L'interfaccia è selezionata:  $/s = 0$
- Viene dato il comando di *write*:  $/iow = 0$

Ovvero, quando questo succede  $e$  ha un fronte di salita, al quale il registro *TBR* campiona il dato

$(d_7 \dots d_0)$  in ingresso dal processore!

Qui la tri-state non serve perché i dati  $d_7 \dots d_0$  ora sono solo ingressi per questa interfaccia, quindi non mi devo preoccupare del bus dati! Ho solo ingressi dal bus, non ho linee condivise!

Ora, vediamone la temporizzazione:



**Da questa temporizzazione si vede bene perché nello spazio di I/O, contrariamente alla memoria, il comando di scrittura è il fronte di DISCESA di  $/iow$ !**

Perché è quel fronte di discesa che corrisponde al fronte di salita di  $e$ , che quindi,

mandato in ingresso al registro *TBR* funge da clock, e gli fa campionare il dato  $d_7 \dots d_0$ !

Inoltre, qui, proprio perché il fronte di discesa di */iow* è il comando di memorizzazione, i dati  $d_7 \dots d_0$  devono essere pronti PRIMA! (Differenza con la temporizzazione dell'interfaccia parallela di ingresso)

Un'interfaccia parallela di uscita si descrive così in Verilog:

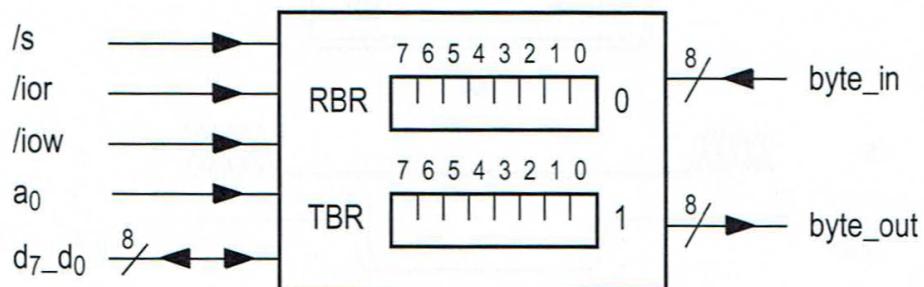
```
// Descrizione di un'interfaccia parallela di uscita
module IntParUscita(s_, ior_, d7_d0, byte_out);
    input s_;
    input [7:0] d7_d0;
    output [7:0] byte_out;

    reg [7:0] TBR; assign byte_out = TBR;
    wire e; assign e = ( {s_, ior_} == 'B00 ) ? 1 : 0; // la porta NOR
    /* assign d7_d0 = (e == 1) ? RBR : 'HZZ; */

    always @(posedge e) #3 TBR <= d7_d0;
endmodule
```

## Interfaccia parallela di I/O

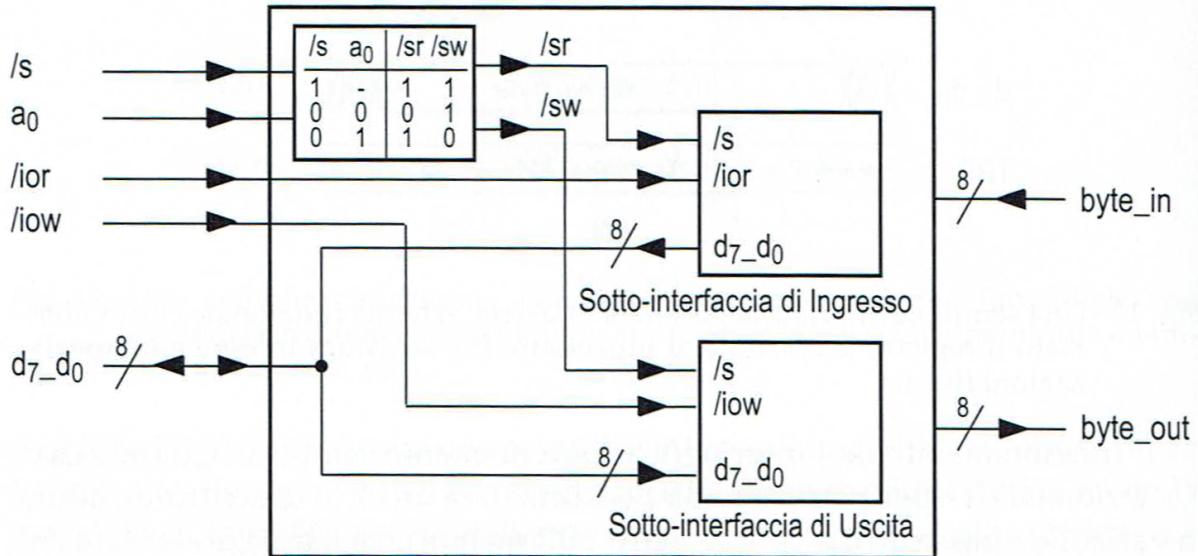
E' un'interfaccia SENZA HANDSHAKE dalla quale il processore può SIA LEGGERE (in *RBR*) CHE SCRIVERE (in *TBR*)



Visione funzionale di un'interfaccia parallela di ingresso/uscita

Visto che ci sono due registri, c'è bisogno stavolta pure di un fil di indirizzo  $a_0$  per indirizzare i due registri!

Vediamo adesso come è realizzata fisicamente:



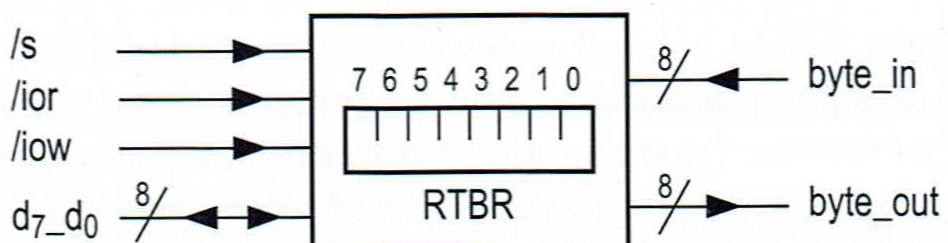
Circuito che implementa un'interfaccia parallela di I/O

C'è bisogno di una RC molto semplice che realizza (come per i banchi di RAM) due segnali di select sulla base di  $a_0$ .

Quindi effettivamente, tutto quello che è un'interfaccia di I/O sono due interfacce in parallelo: una di ingresso, una di uscita, con i rispettivi registri *RBR* e *TBR*.

## Interfaccia parallela di I/O compattata

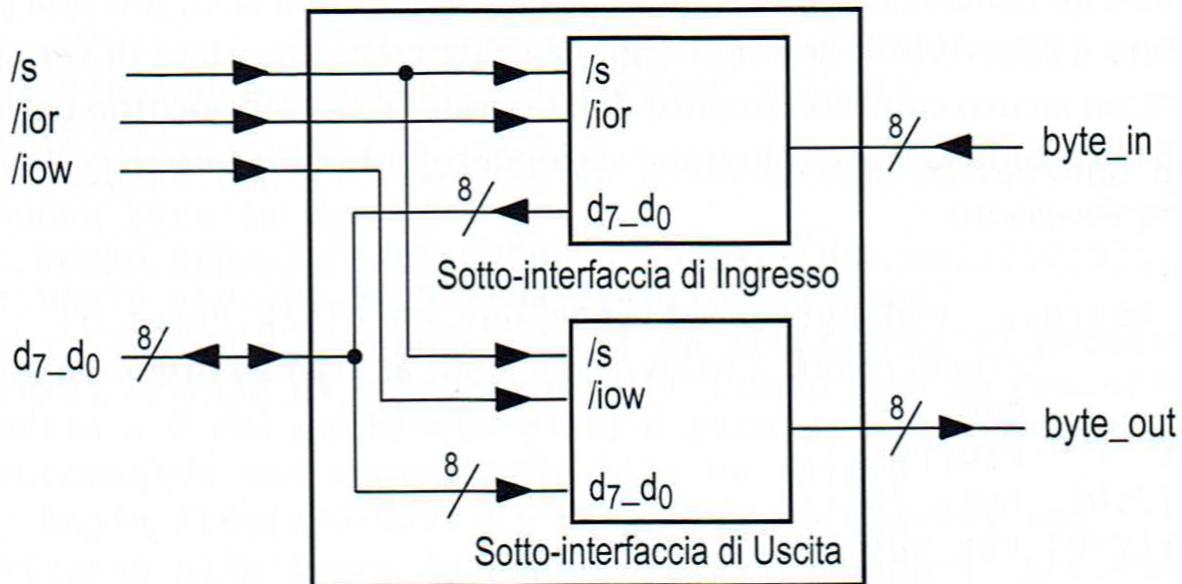
**Un'interfaccia di I/O può essere montata anche compattando i due registri *RBR* e *TBR* allo stesso offset:**



Visione funzionale di un'interfaccia parallela di I/O compattata

Dunque questa non avrà bisogno di  $a_0$  per distinguere i due registri!

Vediamo adesso come è realizzata fisicamente:



Circuito che implementa un'interfaccia di I/O compattata

**Non c'è più bisogno nè di  $a_0$  né della RC che sintetizza due segnali di select distinti!**

Ho un solo segnale di select perché dal punto di vista logico (del processore) c'è un solo registro!

Inoltre, posso sempre fare solo un'operazione per volta, quindi posso selezionare entrambe le Sotto-Interfacce, e poi sarà il comando da fare ( $/ior = 0, /iow = 0$ ) a decidere a quale registro accederò, sfruttando le porte NOR delle Sotto-Interfacce!

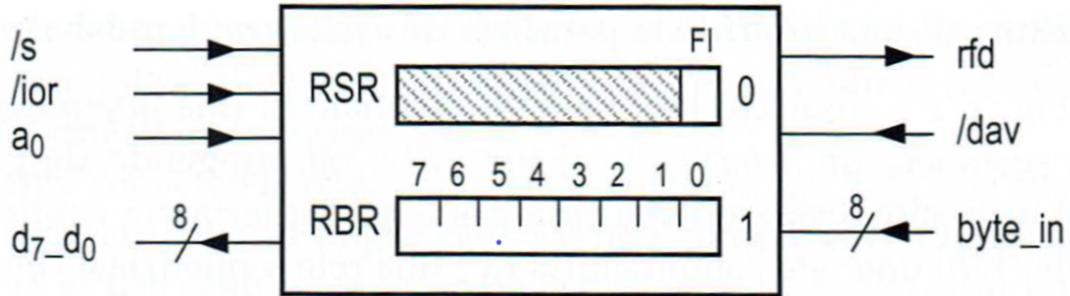
Quando il processore vorrà leggere accederà in lettura alla Sotto-Interfaccia di Ingresso (cioè a  $RBR$ ), quando vorrà scrivere accederà in scrittura alla Sotto-Interfaccia di Uscita (cioè a  $TBR$ )!

Fisicamente sono due registri distinti, ma il processore li vede come un unico registro su cui può fare sia *read* che *write*, ma fisicamente, se fa una *read* accede a  $RBR$ , se fa una *write* accede a  $TBR$ !

## Interfacce parallele con handshake

### Interfaccia parallela di ingresso con handshake

E' un'interfaccia CON HANDSHAKE nella quale il processore può SOLO LEGGERE!

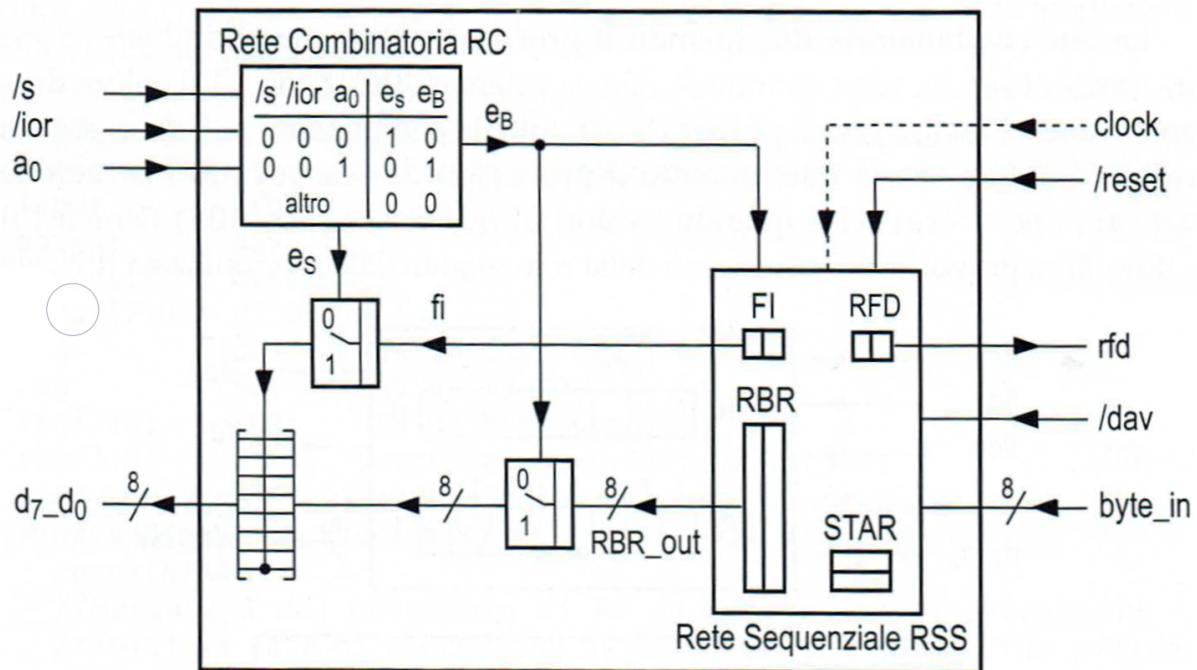


Visione funzionale di un'interfaccia parallela di ingresso con handshake

L'interfaccia gestisce 2 handshake:

- L'handshake software *FI* col processore
- L'handshake hardware */dav – rfd* col dispositivo a valle, dove l'interfaccia è *consumatore* e il dispositivo è *produttore*

Vediamo adesso come è realizzata fisicamente:



Circuito che implementa un'interfaccia parallela di ingresso con handshake

Quando sto selezionando quest'interfaccia ( $/s = 0$ ) e sto facendo una lettura ( $/ior = 0$ ) allora sulla base del valore  $a_0$ , che distingue se accedo al Registro *RSR* o *RBR* decido di:

- Abilitare la singola tri-state per far passare il flag *FI* in uscita ( $e_s = 1$ )

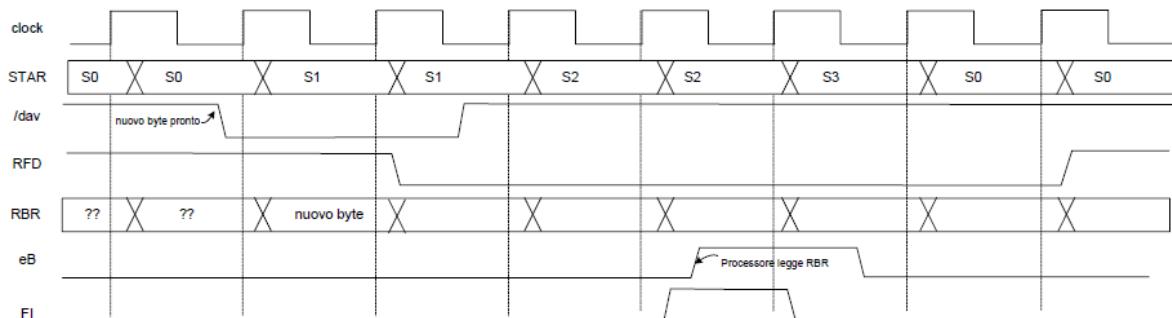
- Abilitare le 8 tri-state per far passare in uscita il dato campionato in *RBR* ( $e_b = 1$ ), fornendo  $e_b$  come clock alla RSS per far campionare *byte\_in* ad RBR al fronte di salita di  $e_b$

Il filo  $e_b$  ha anche un altro scopo: lo mando in ingresso alla RSS perché, nell'handshake *FI*, quando il processore legge, devo riportare *FI* a 0, che coincide con l'istante in cui metto in conduzione le 8 tri-state! Quindi,  $e_b$  serve a:

- Mettere in conduzione le tri-state per far passare il dato campionato in *RBR* sul bus dati
- In contemporanea, aggiornare il registro *FI* rimettendolo a 0 per far concludere l'handshake col processore (perché il dato è stato appena letto)

Entrambe le tri-state non possono essere attive contemporaneamente!

Ora, vediamone la temporizzazione:



Attenzione!  $e_B$  e  $FI$  sono sbagliati!

Un'interfaccia parallela di ingresso con handshake si descrive così in Verilog:

```
// Descrizione di un'interfaccia parallela di ingresso con handshake
// Si gestisce UN HANDSHAKE PER VOLTA, prima /dav-rfd, poi FI!

module RSS(byte_in, RBR_out, eb, fi, dav_, rfd, clock, reset_);
    input clock, reset_;
    input dav_, eb;
    input [7:0] byte_in;
```

```

output fi, rfd;
output [7:0] RBR_out;

wire clock_RSS; assign #5 clock_RSS = clock; // Si ritarda il clock di questa RSS

reg RFD; assign rfd = RFD;
reg FI; assign fi = FI;
reg [7:0] RBR; assign RBR_out = RBR;
reg [1:0] STAR; parameter S0 = 'B00, S1 = 'B01, S2 = 'B10, S3 = 'B11;

always @(reset_ == 0) #1
begin
    RFD <= 1;
    FI <= 0;
    STAR <= S0;
end

always @ (posedge clock_RSS) if(reset_ == 1) #3
casex(STAR)
    S0: begin
        RFD <= 1;
        RBR <= byte_in;
        STAR <= (dav_ == 0) ? S1 : S0;
    end

    S1: begin
        RFD <= 0;
        STAR <= (dav_ == 1) ? S2 : S1;
    end

    S2: begin
        FI <= (eb == 0) ? 1 : 0;
        STAR <= (eb == 0) ? S2 : S3;
    end

    S3: begin
        STAR <= (eb == 1) ? S3 : S0;
    end

// Modo alternativo di scrivere S2 e S3
S2: begin
    FI <= 1;
    RFD <= 1;
    STAR <= (eb == 1) ? S3 : S2;
end

S3: begin
    FI <= 0;
    STAR <= (eb == 0) ? S0 : S3;
end

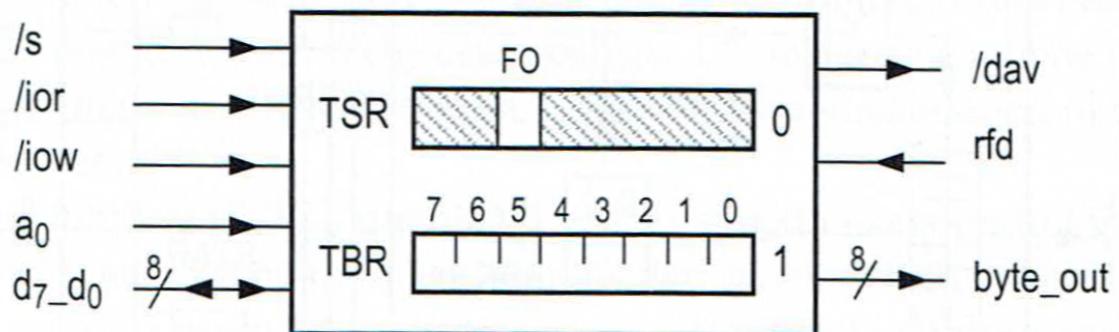
endcase

endmodule

```

## Interfaccia parallela di uscita con handshake

E' un'interfaccia CON HANDSHAKE nella quale il processore può SOLO SCRIVERE!



Visione funzionale di un'interfaccia parallela di uscita con handshake

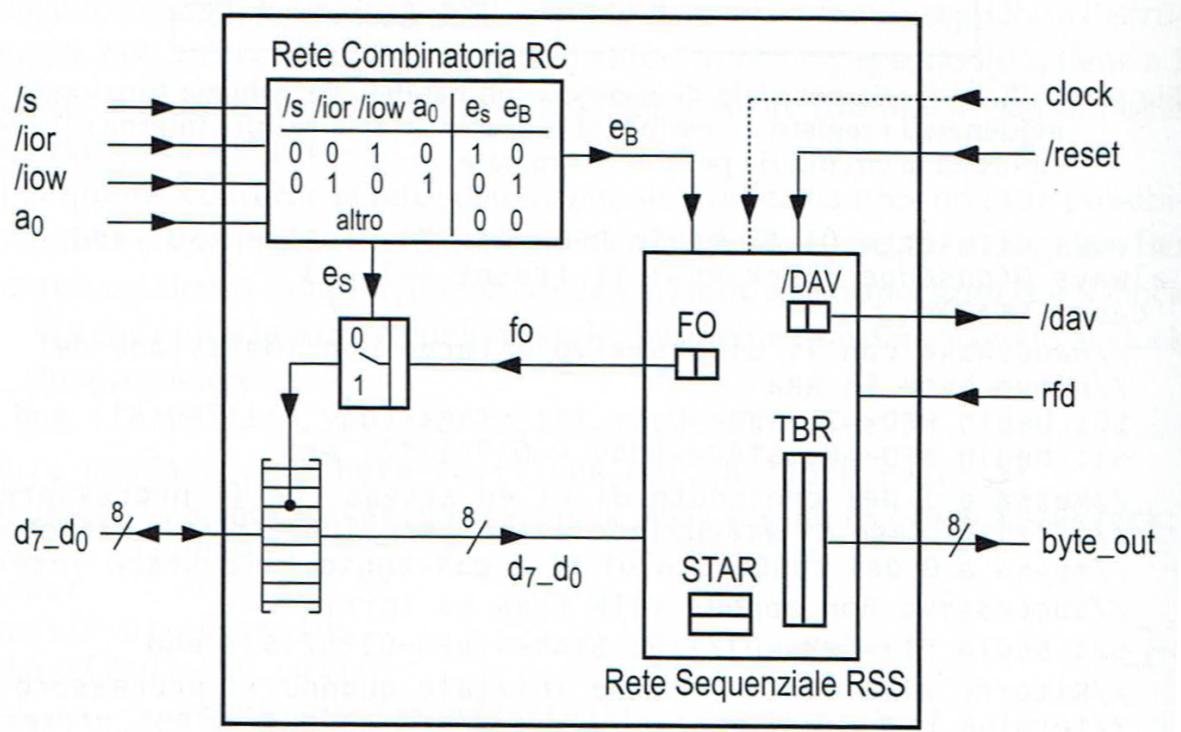
L'interfaccia gestisce 2 handshake:

- L'handshake software *FO* col processore
- L'handshake hardware */dav – rfd* col dispositivo a valle, dove l'interfaccia è *produttore* e il dispositivo è *consumatore*

### Attenzione alle differenze:

- Interfaccia par. ingresso con handshake: Handshake *FI*, l'interfaccia è *consumatore*!
- Interfaccia par. uscita con handshake: Handshake *FO*, l'interfaccia è *produttore*!

Vediamo adesso come è realizzata fisicamente:



Circuito che implementa un'interfaccia parallela di uscita con handshake

Il flag  $e_s$  serve ad abilitare la tri-state per far passare il flag  $FO$  in uscita sul bus (leggo in *TSR*).

Il flag  $e_b$  serve a far avanzare l'handshake  $FO$  col processore, perché il fronte di discesa di  $FO$  corrisponde al fronte di salita di  $e_b$ !

Il fronte di discesa di  $rfd$  mi dice invece che il dispositivo ha letto  $TBR$ , e quindi  $FO$  torna ad 1!

Non mi hanno dato la temporizzazione  $\otimes$ .

Un'interfaccia parallela di ingresso con handshake si descrive così in Verilog:

```
// Descrizione di un'interfaccia parallela di uscita con handshake
// Si gestisce UN HANDSHAKE PER VOLTA, prima FO, poi /dav-rfd!

module RSS(byte_out, d7_d0, eb, fi, dav_, rfd, clock, reset_);
    input clock, reset_;
    input rfd, eb;
    input [7:0] d7_d0;
    output fo, dav_;
    output [7:0] byte_out;

    wire clock_RSS; assign #5 clock_RSS = clock; // si ritarda il clock di questa RSS
```

```

reg DAV_; assign dav_ = DAV_;
reg FO; assign fo = FO;
reg [7:0] TBR; assign byte_out = TBR;
reg [1:0] STAR; parameter S0 = 'B00, S1 = 'B01, S2 = 'B10, S3 = 'B11;

always @(reset_ == 0) #1
begin
    DAV_ <= 1;
    FO <= 1;
    STAR <= S0;
end

always @(posedge clock_RSS) if(reset_ == 1) #3
casex(STAR)
    S0: begin
        FO <= (eb == 0) ? 1 : 0;
        TBR <= d7_d0;
        STAR <= (eb == 0) ? S0 : S1;
    end

    S1: begin
        STAR <= (eb == 0) ? S2 : S1;
    end

    S2: begin
        DAV_ <= 0;
        STAR <= (rfd == 0) ? S3 : S2;
    end

    S3: begin
        DAV_ <= 1;
        STAR <= (rfd == 1) ? S0 : S3;
    end

// Modo alternativo di scrivere S0 e S1
S0: begin
    FO <= 1;
    TBR <= d7_d0;
    STAR <= (eb == 0) ? S0 : S1;
end

S1: begin
    FO <= 0;
    STAR <= (eb == 0) ? S2 : S1;
end

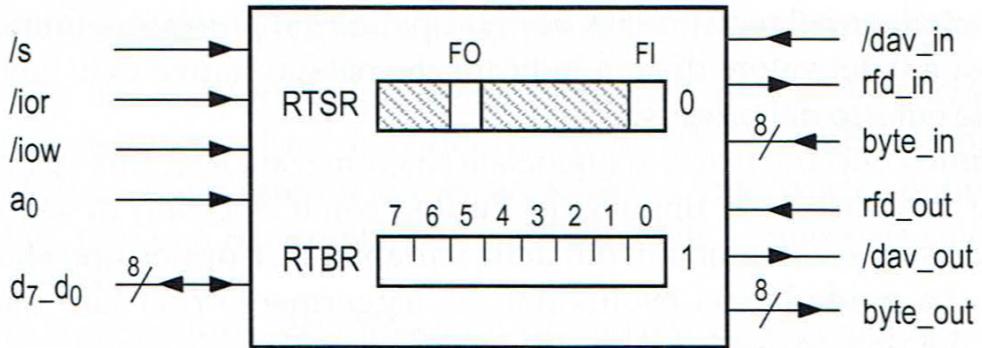
endcase

endmodule

```

## Interfaccia parallela di I/O compattata con handshake

E' semplicemente un'interfaccia parallela di I/O compattata, ma con anche l'handshake /dav-rfd:



Visione funzionale di un'interfaccia parallela di I/O compattata con handshake

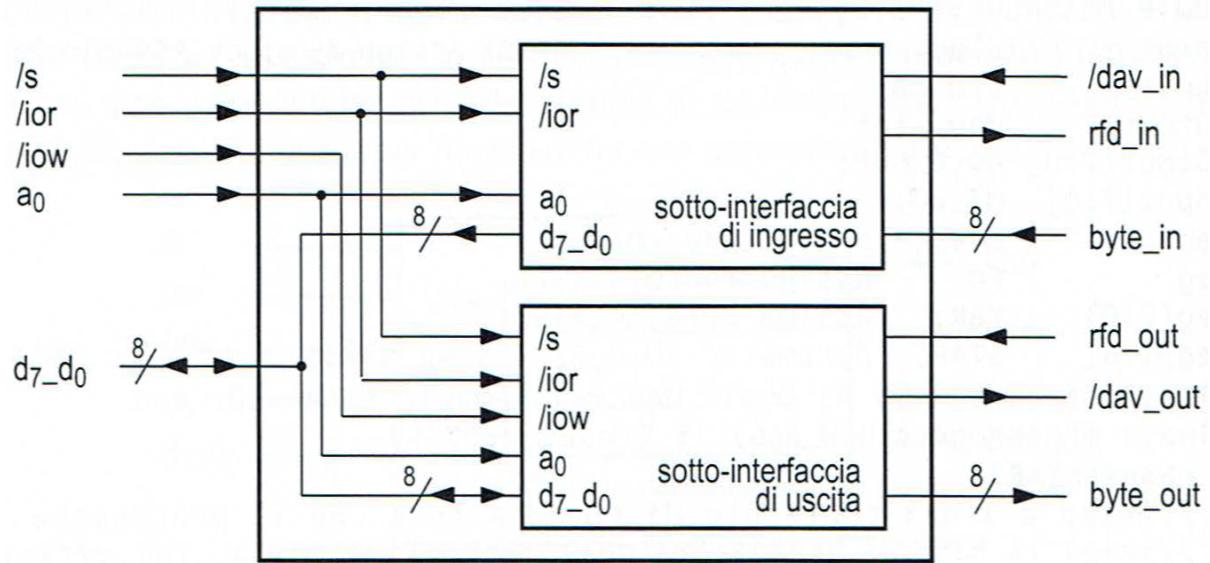
Faccio le stesse cose che facevo anche senza handshake:

- Monto *RBR* e *TBR* allo stesso offset, tanto in *RBR* il processore può solo leggere, e in *TBR* può solo scrivere
- Monto *RSR* e *TSR* allo stesso offset, tanto solo due bit sono significativi (*FO* e *FI*)

Inoltre, ora avrò da gestire due handshake /dav-rfd:

- Uno, */dav\_in*, *rfd\_in* con un dispositivo che è *produttore*, e per il quale l'interfaccia è *consumatore*
- L'altro, */dav\_out*, *rfd\_out* con un dispositivo che è *consumatore*, e per il quale l'interfaccia è *produttore*

Vediamo adesso come è realizzata fisicamente:



Circuito che implementa un'interfaccia parallela di I/O compattata con handshake

Notare come si porti tutto in parallelo a entrambe le sotto-interfacce, tranne */iow*! Questo perché il processore può (solo) leggere in entrambi *RSR* e *TSR*, quindi */ior* ci vuole in entrambe, mentre può scrivere SOLO in *TSR*, che è presente nella sotto-interfaccia di uscita!

Quindi */iow* andrà solo a questa interfaccia!

## Interfacce seriali start/stop

### Comunicazione seriale

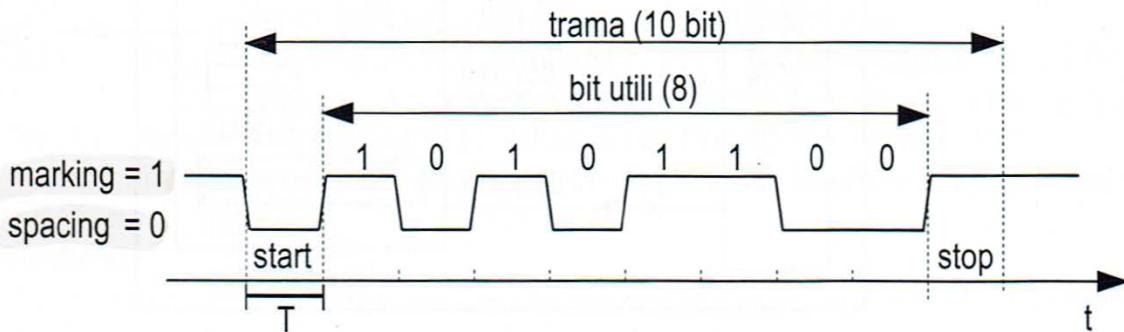
La comunicazione di un'interfaccia seriale è la seguente:

- Lavora in byte lato processore
- Lavora in bit lato dispositivo

Vediamo un pò di definizioni iniziali:

- **TEMPO DI BIT *T***: Tempo che intercorre fra due *bit* consecutivi
- **TRAMA (FRAME)**: Gruppo di *bit* che viene inviato insieme
- **BIT UTILI**: Sono i *bit* in una trama che sono l'effettiva informazione trasmessa

- **BIT DI START:** Specifico *bit* che segna l'inizio di una trama
- **BIT DI STOP:** Specifico *bit* che segna la fine di una trama
- **MARKING:** 1 logico
- **SPACING:** 0 logico



**La trasmissione di un *bit* consiste nel tenere la linea in uno stato di marking o spacing per il tempo di bit  $T$ !**

## Sincronizzazione

**Come si fa a sincronizzare trasmettitore e ricevitore, se solo uno conosce il tempo di bit?**

Ricevitore e Trasmettitore devono:

- Concordare sul tempo di bit  $T$
- Concordare sul numero di bit che compone una trama
- Concordare sui bit di start e stop

### Noi faremo così:

1. **Di base la linea sta nello stato di marking.**
2. Quando voglio iniziare una trama, la porto nello stato di spacing.  
**Quindi il bit di start è 0.**

- Quando voglio finire la trama, porto la linea nello stato di marking per ALMENO un tempo di bit.

**Quindi il bit di stop è ALMENO un 1.**

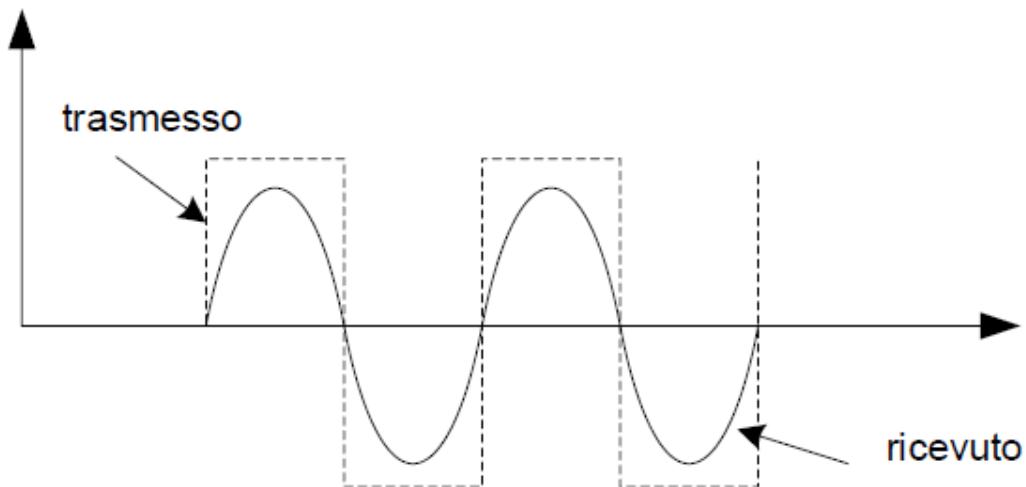
Questo si fa perché non si vogliono aggiungere altri fili oltre ai 2 necessari per eseguire la comunicazione (*ground* e il filo dei dati), ma questo si paga col prezzo della velocità di trasmissione, che sarà  $\frac{n}{n+2}$ , dove  $n$  è il numero di bit utili!

Quindi converrebbe aumentare  $n$  in modo da ottimizzare la trasmissione!

### Ci sono 2 problemi però:

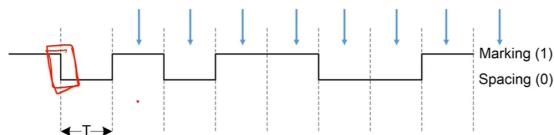
- Il trasmettitore trasmette un'onda quadra, ma il ricevitore, a causa delle reattanze dei conduttori che filtrano le frequenze più alte del segnale, riceve una versione "continua" e "scalata" del segnale originale:  
Le trasmissioni sono fatte mettendo 1 sui segnali con  $y \geq 0$ , mentre 0 su quelli  $y < 0$ .

**Se campiono il segnale nei punti dove la sinusoide si avvicina a 0**, visto che nella realtà ci sarà pure del rumore che si sommerà al segnale puro, **rischio di scambiare un 1 per uno 0 e viceversa!**

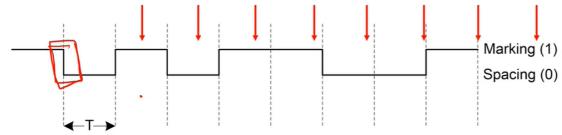


- Il clock del ricevitore e del trasmettitore sono diversi, anche se di poco.**  
Allora, bisogna fare in modo che non succeda che questo errore si accumuli di *bit* in *bit*, fino a far campionare due volte lo stesso *bit* o (al contrario) saltarne uno!

O meglio, l'errore si accumula, inevitabilmente, ciò che bisogna fare è capire quanti bit massimi si possono trasmettere senza avere un errore!



Versione ideale: i due clock sono sincronizzati



Versione reale: clock del ricevitore è un pochino più lento di quello del trasmettitore



**Per questi motivi conviene campionare i bit sulla linea il più vicino possibile alla metà!**

Con riferimento alla figura sopra, sapendo che il bit di start è 0 e quello di stop è 1, il ricevitore deve:

- Aspettare  $\frac{3}{2}T$  da quando la linea passa da marking a spacing (inizio della trama), così da
  - Saltare il bit di start ( $T$ )
  - Campionare il primo bit utile nel mezzo ( $\frac{1}{2}T$ )
- Aspettare  $T$  per campionare il prossimo bit utile

Se il clock del ricevitore misura un tempo di bit  $T \pm \Delta T$ , per poter campionare correttamente  $n$  bit ci vuole che  $n \cdot \Delta T < \frac{T}{2}$ , quindi che

$$\frac{\Delta T}{T} < \frac{1}{2n}$$

Dove  $\Delta T/T$  è la **PRECISIONE RELATIVA DEL CLOCK**.

**Quindi  $n$  non può essere grande arbitrariamente!**

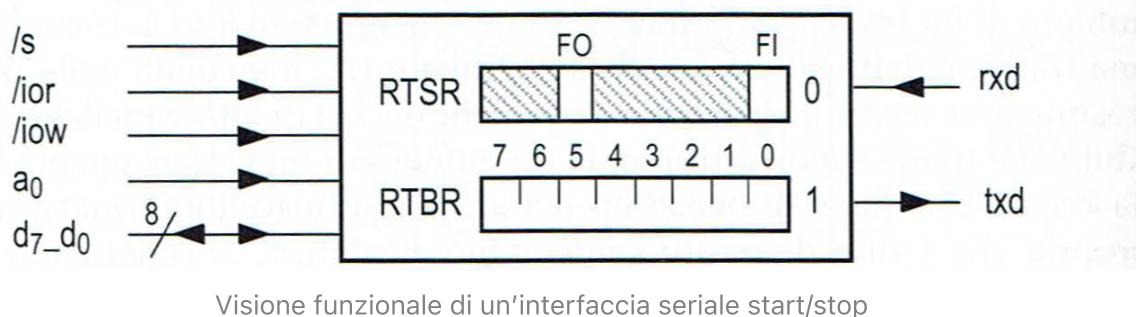


### Il clock del Trasmettitore è più lento di quello del Ricevitore!

Il clock del Ricevitore DEVE essere più veloce di quello del Trasmettitore, così può campionare a  $\frac{1}{2}T$  e  $\frac{3}{2}T$  con la massima precisione possibile, minimizzando l'errore che commette!  
Più veloce è il clock del Ricevitore più grandi sono le trame trasmissibili (efficienza)!

## Interfaccia seriale start/stop

Un'interfaccia seriale start/stop è un'interfaccia che comunica lato-processore in *byte*, mentre lato-dispositivo in singoli *bit*!

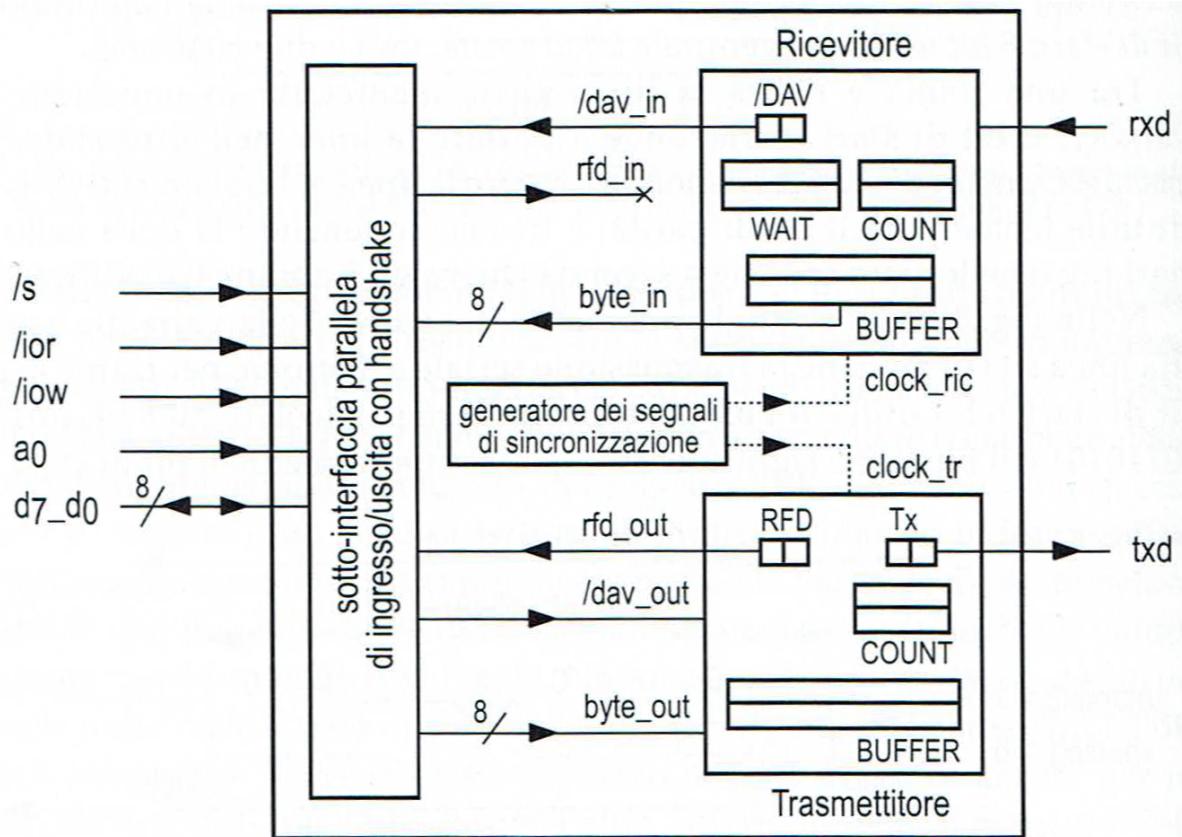


Sono sostanzialmente interfacce di I/O compattate con handshake, ma solo l'handshake "software" lato processore, perché lato dispositivo c'è la comunicazione seriale!

## Cose nuove:

- **Linea di trasmissione *txd*** che comunica con una rete Trasmettitore (non disegnata)
- **Linea di ricezione *rxd*** che comunica con una rete Ricevitore (non disegnata)
- Un filo di massa (*ground*) che non si disegna

Vediamo adesso come è realizzata fisicamente:



Circuito che implementa un'interfaccia seriale start/stop

Oltre alla sotto-interfaccia parallela di I/O compattata con handshake che abbiamo già visto, ci sono due Reti, che **si occupano della serializzazione ( $byte \iff bit$ )**:

- **RICEVITORE**, la quale **riceve dalla linea seriale la trama un  $bit$  alla volta, la raggruppa in un  $byte$ , e la manda alla sotto-interfaccia di I/O**
- **TRASMETTITORE**, la quale **riceve dall'interfaccia parallela di I/O un  $byte$ , e lo manda, un  $bit$  alla volta, sulla linea di trasmissione**

Si genera appositamente un segnale di *clock* per entrambe queste nuove reti, perché il loro *clock* deve essere molto simile al tempo di bit  $T$ !

Perché il filo  $rfd\_in$  viene scartato?

Perché non posso controllare la velocità con la quale il Trasmettitore che sta a destra (non disegnato) mi trasmette i dati! Comanda lui!

Se il processore ce la fa in tempo a star dietro al Ricevitore disegnato in figura bene, sennò si sovrascriverà qualcosa! (nelle reti vere in realtà è più complicato il funzionamento, questo sopra è una versione didattica semplificata!).



**Per il Trasmettitore si fa l'ipotesi semplificativa  $T_{trs} = T$ , ovvero che esso abbia periodo di clock proprio pari al tempo di bit  $T$ !**

Per il Ricevitore, invece, questo non si può fare, per i 2 problemi visti sopra:

- Il primo ci costringe a campionare i *bit* nel mezzo
- Il secondo ci costringe a fare quanto più preciso possibile il clock

Quindi, per campionare un *bit* nel mezzo, il Ricevitore dovrà avere periodo di clock  $T_{ric} \leq T/2$ , perché sennò non riuscirei a campionare nel mezzo ogni bit, e non riuscirei ad aspettare  $3/2 T$ .

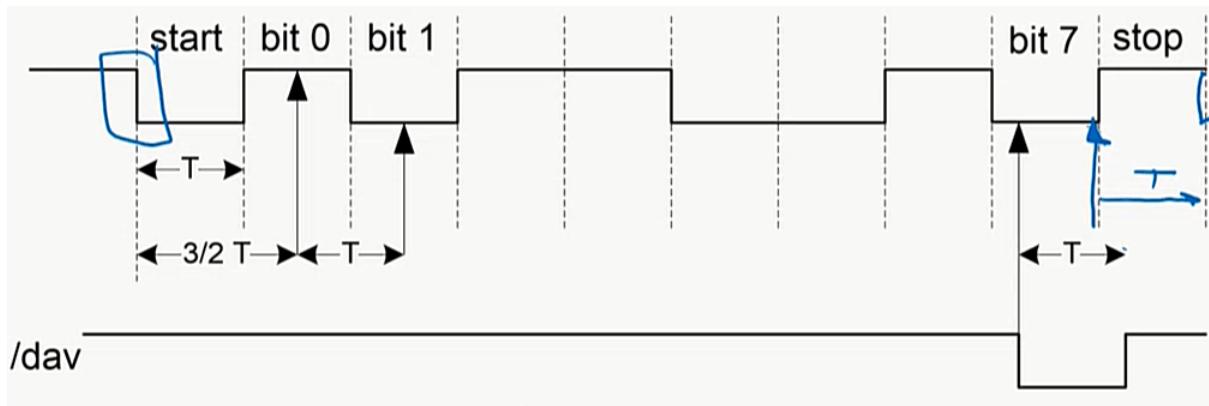
Inoltre, visto che c'è il secondo problema, bisogna farlo il più preciso possibile, che significa farlo il più veloce possibile!



**Per il Ricevitore si fa l'ipotesi  $T_{ric} = T/16$ , ovvero che il suo periodo di clock sia 16 volte più veloce del tempo di bit  $T$ !**

Infine, devo tenere */dav* a 0 per "esattamente"  $T$ , né più né meno:

- Se lo tengo a 0 meno di  $T$ , potrebbe riiniziare una nuova trama mentre sono ancora al *bit* 7 e campionarlo come primo *bit* della nuova trama
- Se lo tengo a 0 più di  $T$ , potrebbe riiniziare una nuova trama con il *bit* di start (*spacing*) e io ho saltato il *bit* di stop (*marking*)



Descriviamo ora in Verilog le nuove reti **Trasmettitore** e **Ricevitore**:

```
// Facciamo l'ipotesi che il periodo di clock del trasmettitore sia
// lungo proprio T = tempo di bit (ipotesi semplificativa)
module Trasmettitore(byte_out, txd, dav_out_, rfd_out, clock, reset_);
    input clock, reset_;
    input dav_out_;
    output rfd_out, txd;

    reg [3:0] COUNT;
    reg [9:0] BUFFER;
    reg RFD, TXD; assign rfd_out = RFD; assign txd = TXD;

    reg [1:0] STAR; parameter S0 = 'B00, S1 = 'B01, S2 = 'B10;

    parameter mark = 1'B1, start_bit = 1'B0, stop_bit = 1'B1;

    always @(reset_ == 0) #1
        begin
            RFD <= 1;
            TXD <= mark;
            STAR <= S0;
        end

    always @ (posedge clock) if(reset_ == 1) #3
        casex(STAR)
            S0: begin
                RFD <= 1;
                COUNT <= 10;
                TXD <= mark;
                BUFFER <= {stop_bit, byte_out, start_bit};
                STAR <= (dav_out_ == 0) ? S1 : S0;
            end

            S1: begin
                RFD <= 0;
                TXD <= BUFFER[0];
                BUFFER <= {mark, BUFFER[9:1]};
                COUNT <= COUNT - 1;
            end
        endcase
    end
endmodule
```

```

        STAR <= (COUNT == 1) ? S2 : S1;
    end

    S2: begin
        STAR <= (dav_out_ == 1) ? S0 : S2;
    end
endcase

endmodule

// Facciamo l'ipotesi che il clock del ricevitore abbia
// frequenza 16x di quella della trasmissione (tempo di bit)
// Inoltre, DAV_ deve stare a 0 per "ESATTAMENTE" T, né più né meno!
module Ricevitore(byte_in, rxd, dav_in, clock, reset_);
    input clock, reset_;
    input rxd;
    output dav_in;
    output [7:0] byte_in;

    reg DAV_;
    reg [3:0] COUNT;
    reg [4:0] WAIT;
    reg [7:0] BUFFER; assign byte_in = BUFFER;
    reg [1:0] STAR; parameter S0 = 'B00, S1 = 'B01, Wbit = 'B10, Wstop = 'B11;

    parameter start_bit = 1'B0;

    always @(reset_ == 0) #1
    begin
        DAV_ <= 1;
        STAR <= S0;
    end

    always @ (posedge clock) if(reset_ == 1) #3
    casex(STAR)
        // Inizializzo WAIT a 23 per aspettare 3/2 T inizialmente
        S0: begin
            DAV_ <= 1;
            COUNT <= 8;
            WAIT <= 23;
            STAR <= (rxd == start_bit) ? Wbit : S0;
        end

        // In questo stato Wait bit si aspetta WAIT, quindi
        // si aspetta 1 tempo di bit (T) se WAIT = 16
        // e si aspetta 1.5 tempo di bit (3/2 T) se WAIT = 23
        Wbit: begin
            WAIT <= WAIT - 1;
            STAR <= (WAIT == 1) ? S1 : Wbit;
        end

        // Ora, una volta aspettato 3/2 T sono nel mezzo del primo bit,
        // quindi campiono (qui in S1) e aspetto T (in Wbit), per COUNT volte.
        // COUNT serve a contare quanti bit ancora mi mancano per finire la trama
        // WAIT invece serve a matchare il mio clock (16x T) col tempo di bit T

```

```

S1: begin
    COUNT <= COUNT - 1;
    WAIT <= 15;
    BUFFER <= {rxd, BUFFER[7:1]}; // shift a destra
    STAR <= (COUNT == 1) ? Wstop : Wbit;
end

// Si aspetta T da quando DAV_ va a 0 e poi si torna in S0 perché è finita la trama
Wstop: begin
    DAV_ <= 0;
    WAIT <= WAIT - 1;
    STAR <= (WAIT == 1) ? S0 : Wstop;
end
endcase
endmodule

```

# Convertitori A/D e D/A

## Generalità

I computer lavorano coi *bit*, cioè **GRANDEZZE DIGITALI**, ovvero **grandezze che variano in modo discreto**.

Il mondo fisico invece lavora con **GRANDEZZE ANALOGICHE**, ovvero **grandezze che variano in modo continuo**.

Dunque, per far entrare e uscire informazioni da un calcolatore, bisogna interfacciarsi con le grandezze analogiche:

- Per far entrare informazioni nel calcolatore c'è bisogno di una conversione A/D
- Per far uscire informazioni nel calcolatore c'è bisogno di una conversione D/A

L'unica grandezza analogica che consideriamo è la **TENSIONE**  $v$  per semplicità, ed ovviamente la tensione non è fissa, ma varia.

Definiamo **FULL-SCALE RANGE (FSR)** l'intervallo in cui varia il **VALORE MASSIMO** della tensione.

La convertiamo in un numero  $x$  ( $\mathbb{N}$ aturale o  $\mathbb{Z}$ intero) in base  $\beta = 2$  su  $N$  bit, grandezza digitale.

Vi sono due tipi di conversioni:

- **CONVERSIONE UNIPOLARE:** ( $x \in \mathbb{N}$ )

- $v \in [0, FSR]$

- $x \in [0, 2^N - 1]$

- **CONVERSIONE BIPOLARE:** ( $x \in \mathbb{Z}$ )

- $v \in \left[ -\frac{FSR}{2}, \frac{FSR}{2} \right]$

- $x \in [-2^{N-1}, 2^{N-1} - 1]$



Si può passare dalla rappresentazione bipolare a quella in  $\sqrt{CR}$  e viceversa invertendo il MSB della rappresentazione!

Definiamo ora la **COSTANTE DI PROPORZIONALITA'** fra i due intervalli:

$$K = \frac{FSR}{2^N}$$

Ora, una conversione ideale sarebbe  $v = K \cdot x$ .

Ma visto che vi sono vari errori nella conversione, che ora analizzeremo, in realtà le conversioni saranno sempre  $|v - K \cdot x| \leq err$ , con  $err$  **ERRORE DI CONVERSIONE.**

Questo errore di conversione è composto da due componenti:

- **ERRORE DI NON LINEARITA':** E' presente in entrambe le conversioni A/D e D/A.

E' dovuto al fatto che i convertitori sono circuiti con resistenze, fili, reattanze, che ovviamente presenteranno imprecisioni.

- **ERRORE DI QUANTIZZAZIONE:** E' presente SOLO nella conversione A/D.

E' dovuto al fatto che devo convertire una grandezza continua (Analogica) in una Digitale, e perdo quindi informazioni a causa dell'arrotondamento.

Stimiamo la massima tolleranza a questi errori:

### **ERRORE DI NON LINEARITA':**

Riscrivo la formula sopra:

$$|v - K \cdot x| \leq err \Rightarrow \begin{cases} v - K \cdot x \leq err \\ -v + K \cdot x \leq err \end{cases} \Rightarrow \begin{cases} v \leq K \cdot x + err \\ v \geq K \cdot x - err \end{cases}$$

Quindi  $v \in [K \cdot x - err, K \cdot x + err]$ .



**L'errore di non linearità DEVE ESSERE PIU' PICCOLO DI  $\frac{K}{2}$ .**

### **Spiegazione:**

Perché se non lo fosse, avrei (esempio): ( $x = 1$  per il primo,  $x = 2$  per il secondo), ( $err = K$ )

$$x = 1 : \quad v \in [K - K, K + K] = [0, 2K]$$

$$x = 2 : \quad v \in [2K - K, 2K + K] = [K, 3K]$$

**Quindi si ha il problema che la conversione non è più monotona!**

Invece se si prende proprio  $err = K/2$ , si ha conflitto solo per  $v = \frac{3}{2}K$ :

$$x = 1 : \quad v \in \left[ K - \frac{K}{2}, K + \frac{K}{2} \right] = \left[ \frac{K}{2}, \frac{3}{2}K \right]$$

$$x = 2 : \quad v \in \left[ 2K - \frac{K}{2}, 2K + \frac{K}{2} \right] = \left[ \frac{3}{2}K, \frac{5}{2}K \right]$$

Da questo si ricava la stima superiore dell'errore di non linearità.

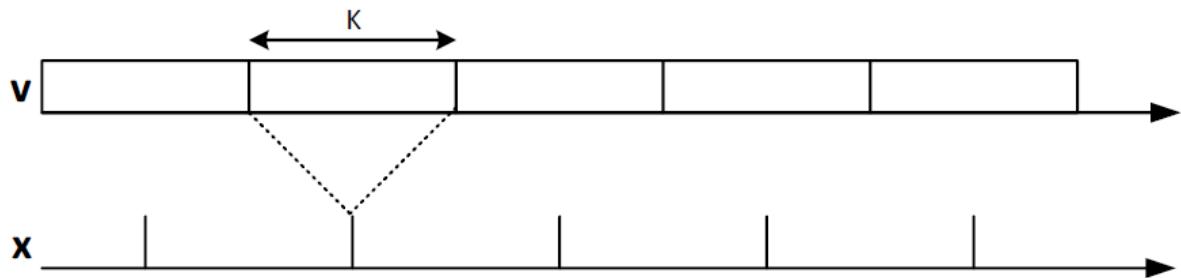
### **ERRORE DI QUANTIZZAZIONE:**



**L'errore di quantizzazione E' SEMPRE PARI A  $\frac{K}{2}$ .**

### Spiegazione:

Se divido il  $FSR$  in  $2^N$  intervalli, ciascuno largo  $K$ :



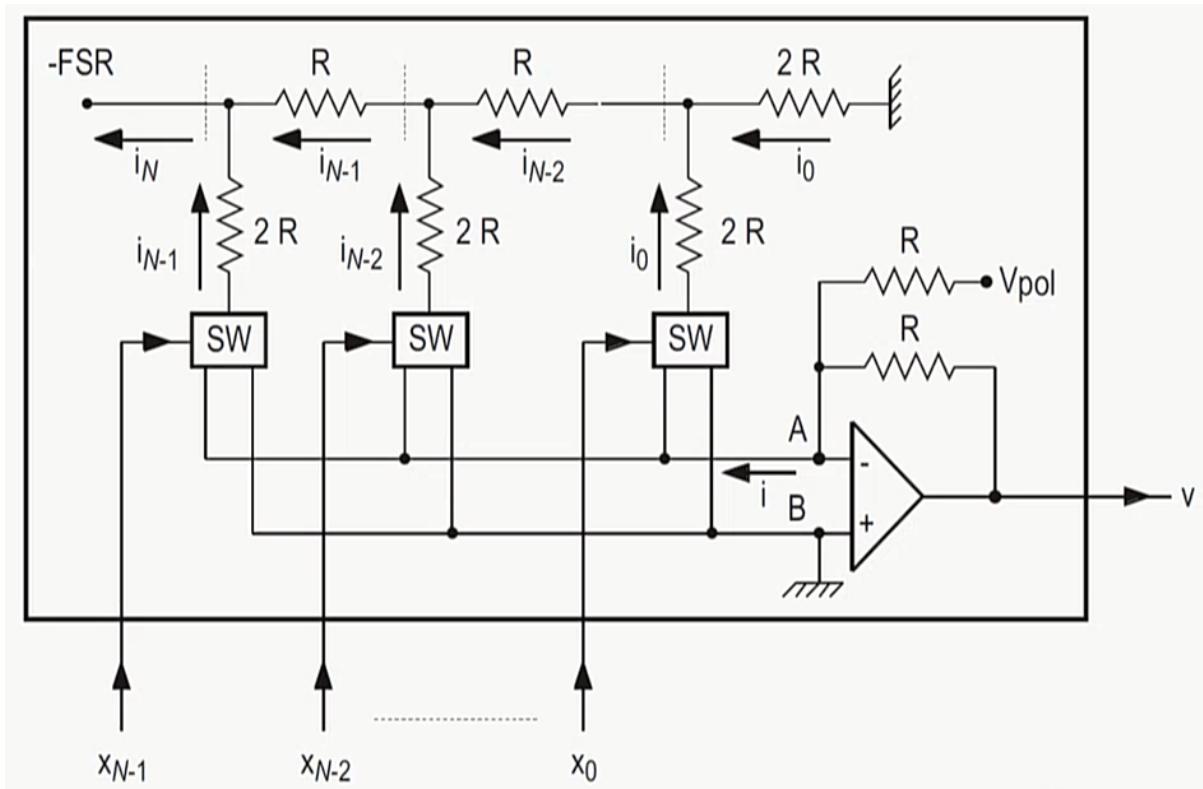
Ottengo che il massimo errore che commetto (che ho quando converto sul "bordo" di un intervallo  $v_i$  nel corrispondente valore  $x$ ) è proprio  $K/2$ , come quando arrotondo 3.5 in 3 o 4, l'errore è 0.5

Quindi:

- **Errore nella Conversione D/A:**  $err < \frac{K}{2}$  (**errore di non linearità**)
- **Errore nella Conversione A/D:**  $err < \frac{K}{2} + \frac{K}{2} = K$  (**errore di non linearità e di quantizzazione**)

## Convertitore D/A

Un Convertitore D/A è una Rete Combinatoria fatta così:



Circuito che implementa un Convertitore D/A

Il numero  $x = (x_{N-1} \ x_{N-2} \ \dots \ x_1 \ x_0)$  viene convertito nella tensione  $v$  in uscita.

I dispositivi chiamati ***SW*** sono **SWITCH**, ovvero **interruttori**, guidati dal bit  $x_j$  che entra, e attaccano l'uscita  $i_j$  alla linea di destra se  $x_j = 0$ , o a quella di sinistra se  $x_j = 1$ .

(Questo perché la linea di destra è attaccata a massa)

Il triangolino è un dispositivo chiamato **AMPLIFICATORE OPERAZIONALE**, il quale **NON FA PASSARE CORRENTE** al suo interno, ma esegue la seguente funzione (**AMPLIFICA LA TENSIONE**):

$$V_{out} = \alpha \cdot (V^+ - V^-), \text{ con } \alpha \gg 1$$

Guardiamo ora cosa succede analizzando il circuito nei due casi separati che possono assumere le switch:

**QUANDO**  $x_j = 0 \ \forall j = 0 \dots N - 1$ :

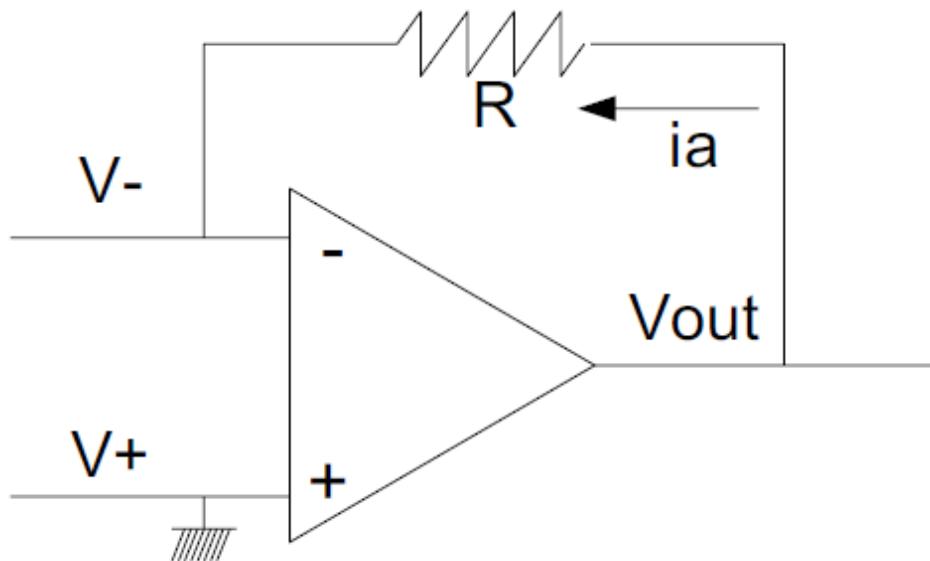
**La corrente che va verso sinistra  $i_j$  raddoppia ad ogni switch**, cioè  $i_N = 2^N \cdot i_0$ .

Inoltre, per la Legge di Ohm  $i_N = \frac{FSR}{R}$  analizzando il circuito elettrico.

$$\text{Quindi } i_0 = \frac{FSR}{R} \cdot \frac{1}{2^N} = \frac{FSR}{2^N} \cdot \frac{1}{R} = \frac{K}{R}.$$

$$\text{Quindi } i_j = 2^j \cdot \frac{K}{R}.$$

**QUANDO**  $x_j = 1 \forall j = 0 \dots N - 1$ :



Si ha che  $V^+ = 0$ , perché è attaccata a massa, quindi

$$V_{out} = \alpha \cdot (0 - V^-) = -\alpha \cdot V^-$$

Per la Legge di Ohm, ancora,  $V^- = V_{out} - R \cdot i_a$  perché l'amplificatore operazionale non fa passare corrente!

Quindi, sostituendo si ottiene

$$\begin{aligned} V_{out} &= -\alpha \cdot (V_{out} - R \cdot i_a) \Rightarrow V_{out} + \alpha \cdot V_{out} = \alpha \cdot R \cdot i_a \\ &\Rightarrow V_{out} = \frac{\alpha}{1 + \alpha} \cdot R \cdot i_a \approx R \cdot i_a \text{ perché } \alpha \gg 1 \end{aligned}$$

Ma allora  $V^- = V_{out} - R \cdot i_a \approx 0$ !

**Quindi per qualunque cosa facciano gli switch le linee in uscita possiamo considerarle come attaccate a massa!**

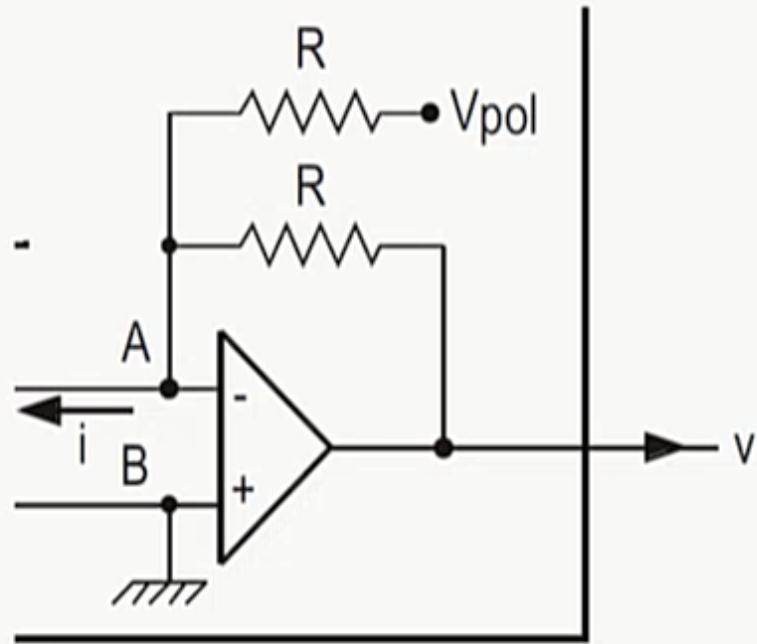
Ora, la corrente  $i$  che esce dal punto  $A$  e va verso sinistra vale:

$$i = x_0 \cdot i_0 + x_1 \cdot i_1 + \cdots + x_{N-1} \cdot i_{N-1} = \\ = i_0 \cdot x_0 + (2 \cdot i_0) \cdot x_1 + \cdots + (2^{N-1} \cdot i_0) \cdot x_{N-1} = i_0 \cdot \sum_{i=0}^{N-1} x_i \cdot 2^i$$

Che è proprio la legge di rappresentazione posizionale del numero Naturale  $X$  in base  $\beta = 2!!!$

Quindi, visto che  $i_0 = \frac{K}{R}$ , si ha che  $i = \frac{K}{R} \cdot X$  (cioè  $i \propto X$ )

Per concludere, a cosa serve la tensione  $V_{pol}$  con in serie una resistenza  $R$ ?



Applicando il 1 principio di Kirkhoff al nodo  $A$  si ha:

$$i = \frac{K}{R} \cdot X = \frac{V_{pol}}{R} + \frac{v}{R} = \frac{V_{pol} + v}{R}$$

Quindi  $K \cdot X = V_{pol} + v \Rightarrow$

$$\Rightarrow v = K \cdot X - V_{pol} = K \cdot \left( X - V_{pol} \cdot \frac{2^N}{FSR} \right)$$

perché  $K = \frac{FSR}{2^N}$ .

$V_{pol}$  serve per poter decidere il tipo di conversione semplicemente settandolo ad un valore o a un altro:

**CONVERSIONE UNIPOLARE:** Metto  $V_{pol} := 0 \Rightarrow v = K \cdot X$

**CONVERSIONE BIPOLARE:** Metto  $V_{pol} := \frac{FSR}{2} \Rightarrow v = K \cdot (X - 2^{N-1}) = K \cdot x$

(rappresentazione in traslazione:  $X = x + 2^{N-1}$ )

**Questo idealmente, ma cosa succede nella realtà? (segno le differenze in blu)**

Intanto abbiamo che le resistenze  $R_1 = \frac{R}{\gamma_1}$  e  $R_2 = \frac{R}{\gamma_2}$ , con  $\gamma_i$  coefficienti, sono diverse fra loro.

Ora, riapplicando come prima il 1 principio di Kirhoff al nodo  $A$  si ha:

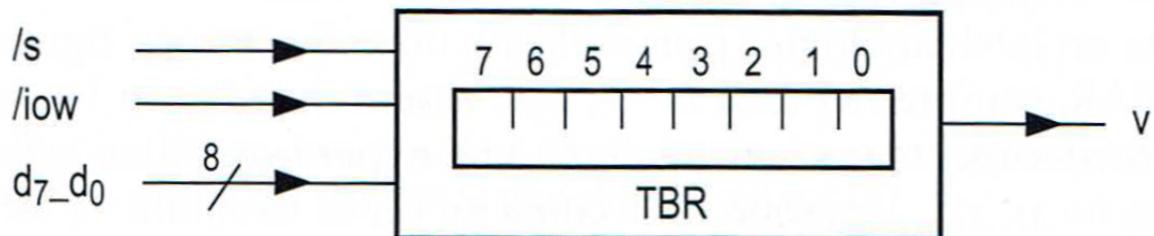
$$i = \frac{K}{R} \cdot X = \frac{V_{pol}}{\frac{R}{\gamma_1}} + \frac{v}{\frac{R}{\gamma_2}} = \frac{V_{pol} \cdot \gamma_1 + v \cdot \gamma_2}{R}$$

Quindi  $K \cdot X = V_{pol} \cdot \gamma_1 + v \cdot \gamma_2 \Rightarrow$

$$\Rightarrow v = \frac{1}{\gamma_2} \cdot (K \cdot X - V_{pol} \cdot \gamma_1) = \frac{K}{\gamma_2} \cdot \left( X - V_{pol} \cdot \gamma_1 \cdot \frac{2^N}{FSR} \right)$$

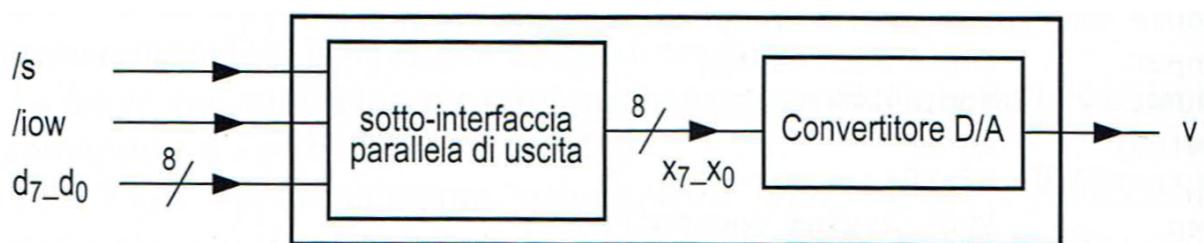
## Interfaccia di conversione D/A

E' come un'interfaccia parallela di uscita senza handshake! E' semplicissima!



Visione funzionale di un'interfaccia per la conversione D/A

Vediamo ora come è realizzata fisicamente:



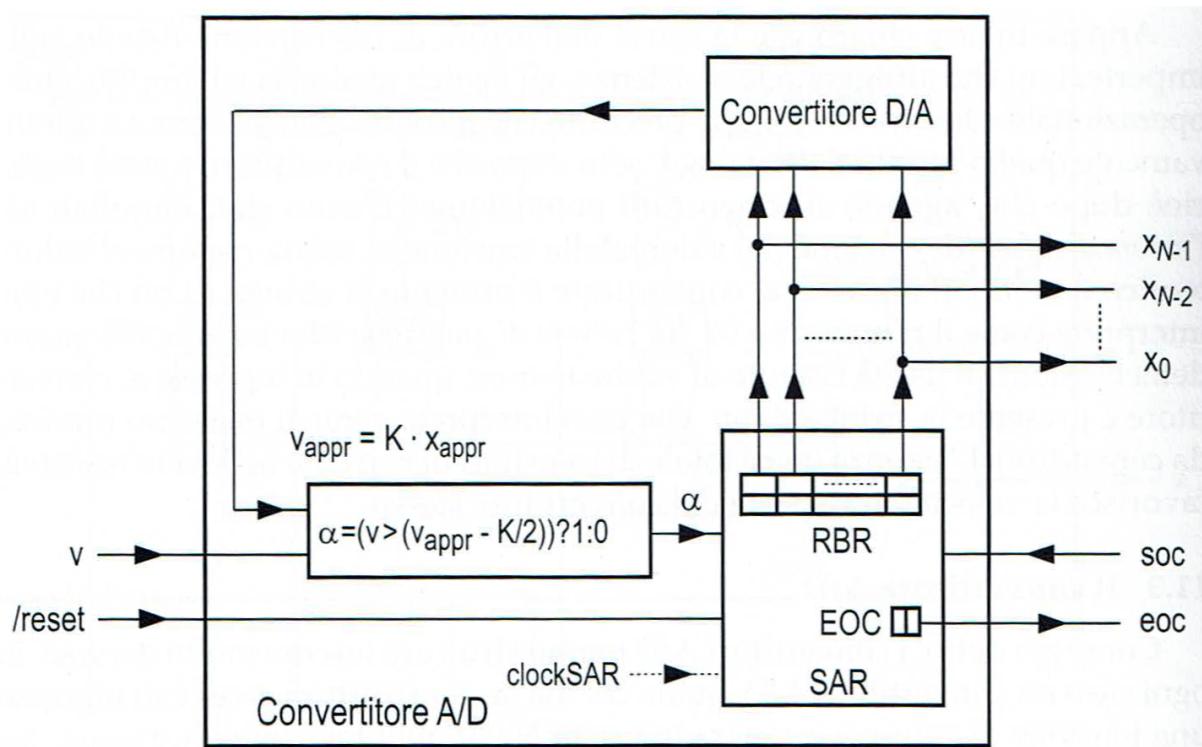
Circuito che implementa un'interfaccia per la conversione D/A

## Convertitore A/D

Ci sono vari modi di fare questa conversione, noi useremo al tecnica detta **SUCCESSIVE APPROXIMATION REGISTER (SAR)**. Inoltre, la conversione sarà ad 8 bit, cioè un *byte*.

La tecnica del SAR consiste nell'eseguire una **RICERCA LOGARITMICA**:

- Prende una tensione analogica  $v$  dall'esterno
- Scommette sul suo valore, e la mette dentro *RBR*
- Converte la tensione scommessa in analogico, grazie al Convertitore D/A
- La confronta con la tensione vera che viene da fuori, e a seconda che sia maggiore o minore tarerà la prossima scommessa più in alto o più in basso, come se fosse un array di numeri



Circuito che implementa un Convertitore A/D

**La ricerca logaritmica in base  $\beta = 2$  significa mettere a posto un bit ad ogni passo.**

**Quindi il numero di passi totali sarà uguale al numero di bit da mettere a posto (nel nostro caso 8)!**

$\alpha$  è il risultato della comparazione, e vale:

- 1 se devo andare nella metà più alta del byte, cioè che la tensione vera  $v$  è più piccola della tensione approssimata  $v_{appr}$
- 0 se devo andare nella metà più bassa del byte, cioè che la tensione vera  $v$  è più grande della tensione approssimata  $v_{appr}$

Si parte mettendo nel Registro *RBR* il numero 1000\_0000, perché è il punto di mezzo dell'intervallo sia che la conversione sia unipolare, sia che sia bipolare! (traslazione)

Spiegazione:

$$\text{Unipolare: } X = 1000\_0000 = 1 \cdot 2^7 = 2^7$$

$$\text{Bipolare: } X = 1000\_0000 + 2^7 = 0$$

 =provvisorio	
1   0   0   0   0   0   0   0	RBR(1)
0   1   0   0   0   0   0   0	$\alpha=0 (=b7)$ RBR(2)
0   1   1   0   0   0   0   0	$\alpha=1 (=b6)$ RBR(3)
0   1   1   0   0   0   0   0	$\alpha=1 (=b5)$ RBR(4)
...	

Ed il valore di  $\alpha$  al passo  $j$ -esimo è proprio il bit  $(8 - j)$ -esimo (con  $j = 1 \dots 8$ )!

Cioè: alla prima iterazione trovo  $b_7$ , alla seconda  $b_6$ , ... all'ottava trovo  $b_0$ !



Nel mentre che eseguo la ricerca logaritmica la tensione esterna  $v$  deve rimanere costante!

Quindi ci vuole un Latch Analogico, che campiona  $v$  e la tiene costante per tutto il tempo.

### Che ci sta a fare il termine $K/2$ nella formula per $\alpha$ ?

Se quel termine non ci fosse, l'errore di quantizzazione sarebbe  $K$ , e non  $K/2$ !

Sostanzialmente il  $K/2$  serve a confrontare  $v$  (analogica) non con la metà dell'intervallo (discreto), ma con il MINIMO valore dell'intervallo!

Se la formula è  $v > v_{appr} - \frac{K}{2}$  sarebbe come dire  $v_{appr} - v < \frac{K}{2}$ , cioè che la distanza fra la tensione che io approssimo e quella reale deve essere minore di  $K/2$ !

Un convertitore A/D si descrive così in Verilog:

```

module SAR(x7_x0, alpha, soc, eoc, clockSAR, reset_);
    input clockSAR, reset_;
    input soc, alpha;

    output eoc;
    output [7:0] x7_x0;

    reg EOC; assign eoc = EOC;
    reg [7:0] RBR; assign x7_x0 = RBR;
    reg [2:0] COUNT;
    reg [3:0] STAR;

    parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;

    always @(reset_ == 0) #1
    begin
        EOC <= 1;
        COUNT <= 7;
        STAR <= S0;
    end

    always @(posedge clockSAR) if(reset_ == 1) #3
    casex(STAR)
        S0: begin
            EOC <= 1;
            STAR <= (soc == 1) ? S1 : S0;
        end

        S1: begin
            RBR <= 'B1000_0000;
            EOC <= 0;
            STAR <= S2;
        end

        S2: begin
            RBR <= nuovobyte(RBR, alpha, COUNT);
            COUNT <= COUNT -1;
            STAR <= (COUNT == 0) ? S3 : S2;
        end
        // Testo COUNT a 0, quindi quando esco da S2 vale di nuovo 7, quindi
        // non ho bisogno di ri-inizializzarlo da altre parti se non al reset_!

        S3: begin
            // EOC <= (soc == 1) ? 0 : 1; ???
            STAR <= (soc == 0) ? S0 : S3;
        end
    endcase

    function [7:0] nuovobyte;
        input [7:0] vecchiobyte;
        input alpha;
        input [2:0] posizione;

        casex(posizione)

```

```

7: nuovobyte = {alpha, 'B100_0000};
6: nuovobyte = {vecchiobyte[7], alpha, 'B10_0000};
5: nuovobyte = {vecchiobyte[7:6], alpha, 'B1_0000};
4: nuovobyte = {vecchiobyte[7:5], alpha, 'B1000};
3: nuovobyte = {vecchiobyte[7:4], alpha, 'B100};
2: nuovobyte = {vecchiobyte[7:3], alpha, 'B10};
1: nuovobyte = {vecchiobyte[7:2], alpha, 'B1};
0: nuovobyte = {vecchiobyte[7:1], alpha};

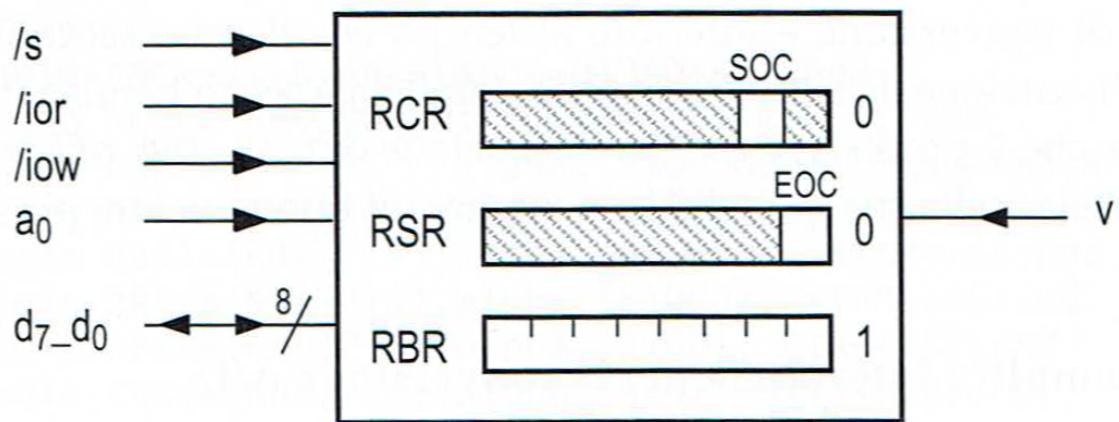
endcase
endfunction

endmodule

```

## Interfaccia di conversione A/D

E' un'interfaccia parallela di I/O CON handshake! (al contrario di quella D/A)



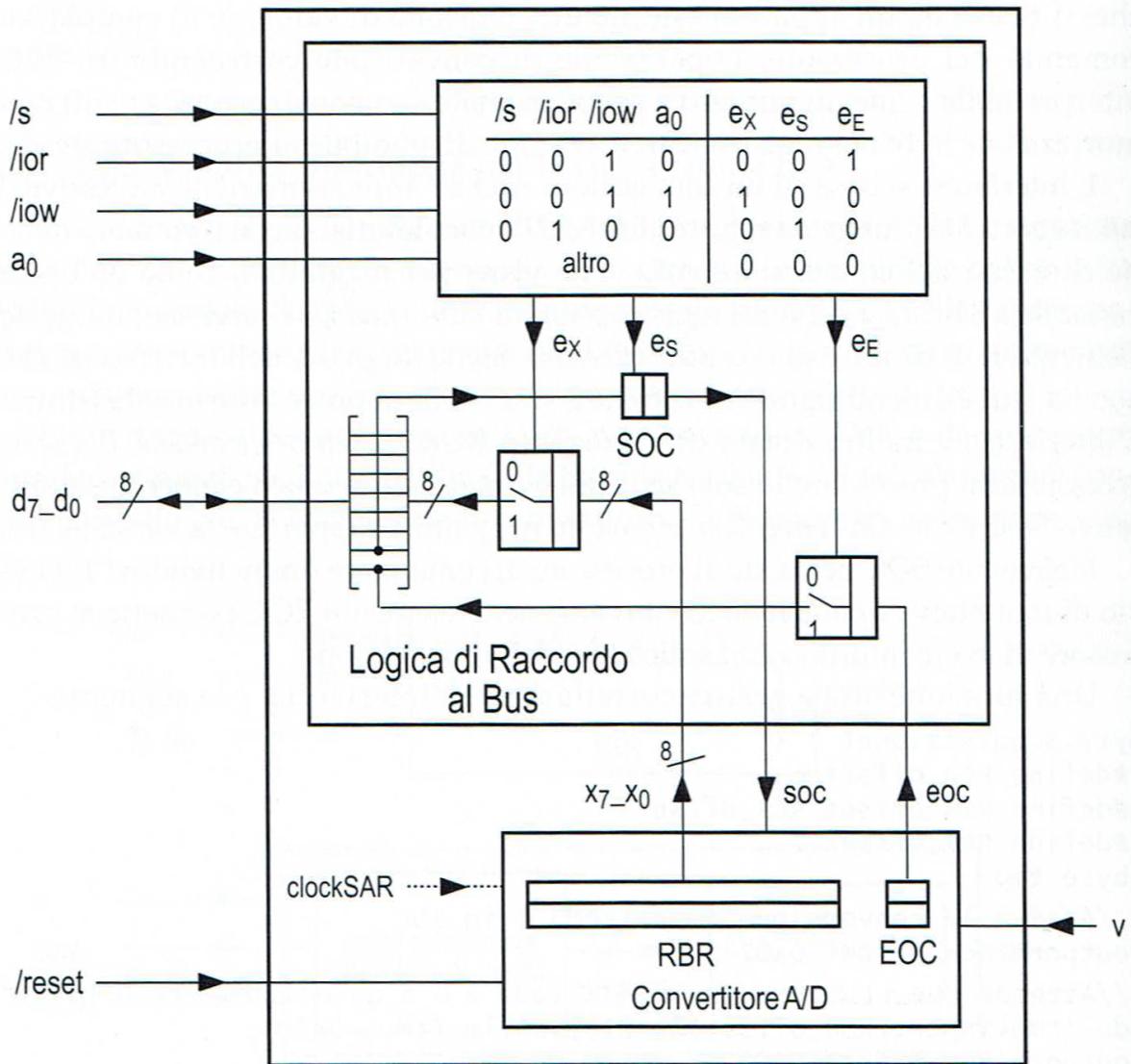
Visione funzionale di un'interfaccia per la conversione A/D

E' di ingresso perché il processore deve leggere dati da *RBR* e leggere il flag *EOC*, ma deve anche poter impostare il flag *SOC*, quindi deve essere anche di uscita!

I Registri *RCR* e *RSR* possono essere collassati in un Registro solo *RSCR* (**RECEIVE STATUS AND CONTROL REGISTER**).

Infatti sono mappati allo stesso indirizzo interno perché si distinguono dal tipo di accesso: lettura o scrittura.

Vediamo ora come realizzarla fisicamente:



Circuito che implementa un convertitore A/D

Sia per *RBR* che *EOC*, visto che supportano delle uscite, ci dovranno essere delle opportune tri-state.

Il resto è semplicemente una RC che sintetizza i segnali di *enable* per le due tri-state e per il registro *SOC*, che ha come clock *e<sub>s</sub>*, al cui fronte di salita memorizza

## Sottoprogramma per la conversione A/D

```

MOV $0x02, %AL
OUT %AL, RCR_offset // soc = 1

test1: IN RSR_offset, %AL // ciclo per attendere che EOC -> 0
    
```

```
AND $0x01, %AL
JNZ test1

XOR %AL, %AL
OUT %AL, RCR_offset // SOC = 0

test2: IN RSR_offset, %AL // ciclo per attendere che EOC -> 1
AND $0x01, %AL
JZ test2

IN RBR_offset, %AL // prelevo il dato convertito
RET
```



**Interfacce → Handshake** /*dav – rfd*  
**Convertitori → Handshake** *soc – eoc*

# Note sul Verilog

Il Verilog è un HDL, Hardware Description Language, quindi



**Il Verilog è un linguaggio di DESCRIZIONE, non di PROGRAMMAZIONE!**

**Ogni rete in VL è un modulo.**

## Sintassi

### Costanti

Si possono scrivere in base  $\beta = 2, 16, 10$

#### **FORMATO:**

$n' \{B, D, H\} \text{ valore}$

- $n$  è il **NUMERO DI BIT** su cui è la costante
- $B, D, H$  sono le tre possibili **BASI** (Binary, Decimal, Hexadecimal)
- *valore* contiene il **valore** effettivo, ovviamente espresso nella base appena scritta!

#### **Esempi:**

- $8'D32$  è la costante 32, in formato decimale, su 8 bit
- $3'B110$  è la costante 110, in base 2, su 3 bit

Di 4 bit in 4 puoi mettere un underscore:  $16'B0010_1000_0101_1100$



**ATTENZIONE! La base  $\beta = 10$  è quella di default!**

**Puoi omettere SOLO  $n$ ! Sia la base che il valore vanno sempre specificati!**

Una variabile logica può assumere in VL 4 valori:

- $1'B1$  che sarebbe 1
- $1'B0$  che sarebbe 0
- $1'BZ$  che sarebbe **Alta Impedenza (USARE SOLO PER VALORI DI ENTRATA)**
- $1'BX$  che sarebbe **Non Specificato (USARE SOLO PER VALORI DI USCITA)**

## “Funzioni”



**LE “FUNZIONI” IN VERILOG SERVONO SOLO PER SCRIVERE TABELLE DI VERITÀ!!!**

Infatti le reti combinatorie sono funzioni matematiche, mentre per Reti Sequenziali c’è pure di mezzo lo Stato Interno Presente!

```
function[1:0] F; // va messo il costrutto [X-1:0] per indicare che l'output della
                // funzione è su X bit. Puoi ometterlo se è su 1 bit
input x2, x1, x0;

casex( {x2, x1, x0} ) // costrutto simile allo switch nel C++
    'B001: F = 'B01;
    'B01?: F = 'B10;
    'B10?: F = 'B11;
default: F = 'B00; // la keyword default è usabile SOLO ALL'INTERNO DI UN CASEX
endcase
endfunction
```

## Forchettare gli ingressi/uscite

```
inout d7_d0;  
assign d7_d0 = (DIR == 1) ? D7_D0 : 'HZZ;
```

## Commenti

Stessa cosa del C++:

- // per singola riga
- /\* \*/ per multi riga

## Tips & Tricks



**Se inizializzo un Registro a  $K$  e lo testo a  $J \leq K$ , il numero di iterazioni è  $K - J + 1$**

Esempi:

**Se inizializzo a 8, per avere 8 iterazioni a quanto devo testare?**

$$8 - x + 1 = 8 \Rightarrow x = 1$$

**A quanto devo inizializzare per avere 8 iterazioni testando a 1?**

$$x - 1 + 1 = 8 \Rightarrow x = 8$$



**Se devi accedere ad un *bit* diverso di un Registro ad ogni iterazione di un ciclo, DEVI SEMPRE shiftare a destra o a sinistra, e prendere il LSB o il MSB!**

**Esempi:**

**Ciclo shiftando a destra:**

```
REG <= BUFFER[0];
BUFFER <= {marking, BUFFER[7:1]}; // Buffer è a 8 bit
[7 6 5 4 3 2 1 0] -> [marking 7 6 5 4 3 2 1]
```

**Ciclo shiftando a sinistra:**

```
REG <= BUFFER[7];
BUFFER <= {BUFFER[6:0], marking};
[7 6 5 4 3 2 1 0] -> [6 5 4 3 2 1 0 marking]
```

Inoltre non importa cosa ci metti, se *marking* o altro, perché quel valore non verrà mai usato!

## Cose a cui fare attenzione!

```
OUTR <= {OUTR[1], 0}; // va bene
OUTR[1] <= 0; // NON HA SENSO SCRIVERLO ED E' SBAGLIATO!!! ERRORE GRAVE!
```

**Scrivere il secondo statement equivarrebbe a dire che c'è un circuito a cui arriva il clock, ma solo ad un bit di questo, mentre all'altro bit no, che non ha senso!**



**Devi sempre pensare a cosa corrisponde in circuiti quello che stai scrivendo il Verilog!  
In particolare, il Verilog serve solo per trascrivere in notazione testuale un disegno circuitale!**



**A SINISTRA DI  $<=$  CI VA SEMPRE SEMPRE SEMPRE IL NOME DI UN REGISTRO INTERO!  
NON ~~OUTR[1], STAR[5], ETC...~~!**

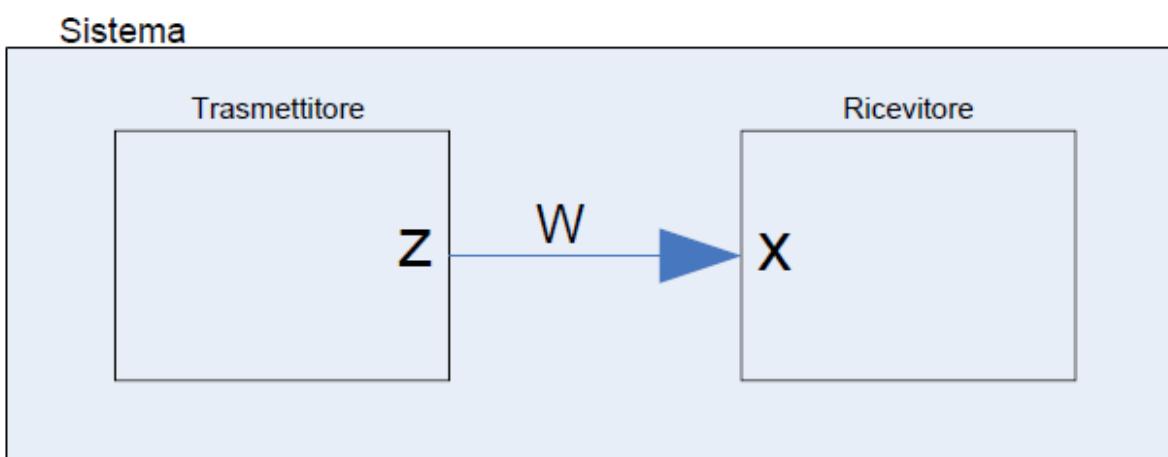
Se vuoi dire che il 2 bit ( OUTR[1] ) di OUTR (a 8 bit) prende il valore 0, si scrive così:

$\text{OUTR} <= \{\text{OUTR}[7:2], 1'B0, \text{OUTR}[0]\}$   
~~E NON  $\text{OUTR}[1] <= 0$ , !!!!!~~

## Descrizioni di Reti in VL:

### Esempio introduttivo

Si vuole descrivere le seguenti reti: Trasmettitore, Ricevitore e Sistema



```
module Sistema;
    wire w; // un wire è un filo, cioè una variabile logica

    // ora INSTANZIO due istanze T ed R di due tipi di rete
    // Trasmettitore e Ricevitore, sono tipo gli oggetti
    // che istanzi da una classe

    Trasmettitore T(w);
```

```

Ricevitore R(w);
endmodule

module Trasmettitore(z);
// nelle parentesi dopo la definizione di un modulo, ci vanno
// sostanzialmente tutte le variabili con cui quel modulo
// si interfaccia con l'esterno, infatti si chiamano Variabili di Interfaccia

output z; // la variabile z è una variabile di uscita per la rete Trasmettitore

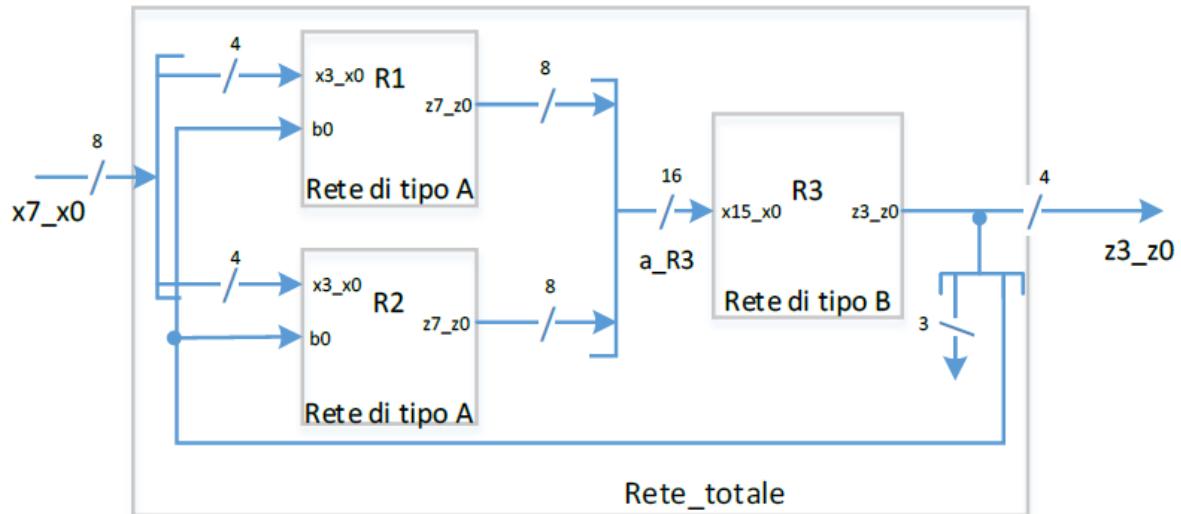
// *descrizione della funzione di questa rete* //
endmodule

module Ricevitore(x);
input x; // la variabile x è una variabile di ingresso per la rete Ricevitore

// *descrizione della funzione di questa rete* //
endmodule

```

## Esempio con interconnessioni



```

module ReteTipoA(z7_z0, x3_x0, b0);
input[3:0] x3_x0;
// questo costrutto con le parentesi quadre indica che
// la variabile x3_x0 è a 4 bit
input b0;

output[7:0] z7_z0;

// *descrizione della funzione di questa rete* //
endmodule

```

```

module ReteTipoB(z3_z0, x15_x0);
    input[15:0] x15_x0;

    output[3:0] z3_z0;

    // *descrizione della funzione di questa rete* //
endmodule

module ReteTotale(z3_z0, x7_x0);
    input[7:0] x7_x0;

    output[3:0] z3_z0;

    wire[15:0] a_R3;

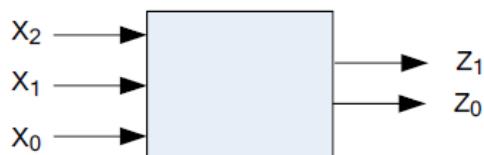
    ReteTipoA R1(a_R3[15:8], x7_x0[7:4], z3_z0[0]),
               R2(a_R3[7:0], x7_x0[3:0], z3_z0[0]);
    // qui con le parentesi si riferiscono degli specifici
    // intervalli di bit delle variabili

    ReteTipoB R3(z3_z0, a_R3);
endmodule

```

## Rete Combinatoria

**Per le RC si deve semplicemente trascrivere la tabella di verità!**



x2	x1	x0	z1	z0
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	0	0
1	1	1	0	0

```

module RC(z1, z0, x2, x1, x0);
    input x2, x1, x0;
    output z1, z0;

    assign #T {z1, z0} = ( {x2, x1, x0}=='B000 ) ? 'B00 :
                           ( {x2, x1, x0}=='B001 ) ? 'B01 :
                           ( {x2, x1, x0}=='B010 ) ? 'B10 :
                           ( {x2, x1, x0}=='B011 ) ? 'B10 :
                           ( {x2, x1, x0}=='B100 ) ? 'B11 :
                           ( {x2, x1, x0}=='B101 ) ? 'B11 :
                           ( {x2, x1, x0}=='B110 ) ? 'B11 :
                           ( {x2, x1, x0}=='B111 ) ? 'B11 :
                           'B00;

```

```

        ( {x2, x1, x0}=='B101 ) ? 'B11 :
        ( {x2, x1, x0}=='B110 ) ? 'B00 :
        ( {x2, x1, x0}=='B111 ) ? 'B00;

// semplificazione: uso del valore di "default" se tutte le alternative falliscono
// tipicamente si usa se c'è un singolo valore di uscita molto ricorrente,
// e quindi lo si mette come "default" come alternativa a tutti i casi "particolari"
assign #T {z1, z0} =  ( {x2, x1, x0}=='B001 ) ? 'B01 :
                      ( {x2, x1, x0}=='B010 ) ? 'B10 :
                      ( {x2, x1, x0}=='B011 ) ? 'B10 :
                      ( {x2, x1, x0}=='B100 ) ? 'B11 :
                      ( {x2, x1, x0}=='B101 ) ? 'B11 :
                                         'B00;

// ulteriore semplificazione: uso del valore "?"
assign #T {z1, z0} =  ( {x2, x1, x0}=='B001 ) ? 'B01 :
                      ( {x2, x1, x0}=='B01? ) ? 'B10 :
                      ( {x2, x1, x0}=='B10? ) ? 'B11 :
                                         'B00;

endmodule

```

*assign* è il costrutto di **ASSEGNAZIONE CONTINUO**, cioè  $\forall t$  rende vera quella uguaglianza!

### Si usa nelle RC per specificare la tabella di verità

#*T* è il **RITARDO DI ASSEGNAZIONE**, che serve a modellare il Tempo di Attraversamento.

Fa comodo averlo, sia perché ovviamente è fisicamente richiesto, sia perché **nelle simulazioni ti aiuta a vedere cosa è causa di cosa!**

```
cond1 ? istr_vera : istr_falsa; // shorthand if statement
```

### Sintesi della RC qui sopra



Nelle **DESCRIZIONI** compaiono **VALORI ESPLICITI** tipo 110, 010, etc...

Nelle **SINTESI** compaiono **ESPRESSIONI ALGEBRICHE** tipo  $x_1 \& x_2$

```

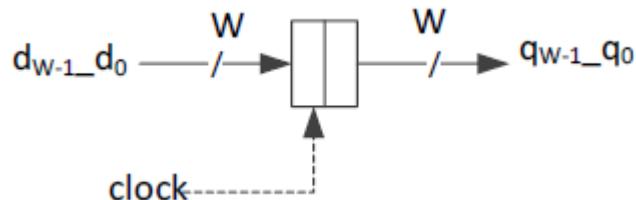
module RC(z1, z0, x2, x1, x0);
    input x2, x1, x0;
    output z1, z0;

    assign #T z1 = (~x2 & x1) | (x2 & ~x1);
    assign #T z2 = (x2 & ~x1) | (x0 & ~x1);

```

<i>Operatore</i>	<i>Verilog</i>
<i>AND</i>	&
<i>OR</i>	
<i>NOT</i>	~
<i>XOR</i>	$\wedge$

## Descrizione di un Registro



```

reg[W-1 : 0] REGISTRO; // convenzione: scrivere i nomi dei registri in maiuscolo
wire clock, reset_; // quando si dichiara un registro, DEVI dichiarare due fili
// di clock e reset_ !!!
wire[W-1 :0] d;
wire[W-1 :0] q; assign q=REGISTRO;

always @(reset_ == 0) #1 REGISTRO <= contenuto_iniziale;
always @ (posedge clock) if(reset_ == 1) #Tprop REGISTRO <= d;

```



Gli **ASSEGNAZIMENTI CONTINUI** (*assign*) sono "asincroni" ( $\forall t$ )

Gli **ASSEGNAZIMENTI PROCEDURALI** ( $<=$ ) sono **SOLO PER I REGISTRI**  
e **SI USANO SOLO DENTRO GLI STATEMENT ALWAYS!**

Inoltre **SCRIVERE UN ASSEGNAZIMENTO PROCEDURALE SIGNIFICA**  
**FAR PASSARE DEL TEMPO! SI "PERDE" UN CLOCK ESEGUENDOLO!**



La stragrande maggioranza degli errori che si fanno all'esame è assumere che quando si scrive una cosa in Verilog quella diventi vera subito.

Non è così, diventa vera al clock dopo:

**NUOVO VALORE DOPO IL CLOCK  $<=$  VECCHIO VALORE PRIMA DEL CLOCK**

**TUTTI GLI STATEMENT RAGGRUPPATI DENTRO UN *begin ... end*  
SONO ESEGUITI NELLO STESSO ISTANTE,  
CONTEMPORANEAMENTE!  
NON CONTA L'ORDINE CON CUI LI SCRIVO!**

Esempio:

```
begin
    STAR <= S1;
    OUTR <= STAR;
end
```

Quanto vale *OUTR*? **NON VALE S1!**

**VALE IL VECCHIO VALORE, QUELLO PRIMA DEL CLOCK, CIOE' S0!**



**Quando ho a che fare con dei registri, PER CIASCUNO vanno scritti due statement *always*, uno per l'inizializzazione al reset, e l'altro per l'aggiornamento dell'uscita al fronte di salita del clock!**



**Gli assegnamenti dei registri ai FILI DI USCITA vanno messi in uno statement *assign*!**

**Gli assegnamenti dei registri al reset e al posedge del clock vanno messi negli statement *always*!**



### MOORE:

- $B(S)$  : *assign*
- $A(X, S)$  : *casex*

### MEALY:

- $B(X, S)$  : *assign*
- $A(X, S)$  : *casex*

### MEALY RITARDATO:

$(A, B)(X, S)$  : *casex begin...end*

# Tips & Tricks

## Assembler

### Estensione di campo

- $\mathbb{N}$ :
  - $8 \rightarrow 16$ : `AND $0x00FF, %AL`
  - $16 \rightarrow 32$ : `AND $0x0000FFFF, %AX`
  - $8 \rightarrow 32$ : `AND $0x000000FF, %AL`
- $\mathbb{Z}$ : Usare l'istruzione `CWDE`

### Manipolare singoli bit in registri

- **Settare singoli bit**: `OR` con una maschera che ha 1 solo ai bit che si vogliono settare
  - Settare il bit 4: `OR $0x10, %AL`  $(10)_{16} = (0001\ 0000)_2$
- **Resettere singoli bit**: `AND` con una maschera che ha 0 solo ai bit che si vogliono resettare
  - Resettere il bit 4: `AND $0xEF, %AL`  $(EF)_{16} = (1110\ 1111)_2$
- **Invertire singoli bit**: `XOR` con una maschera che ha 1 solo ai bit che si vogliono invertire
  - Invertire il bit 4: `XOR $0x10, %AL`  $(10)_{16} = (0001\ 0000)_2$
- **Testare singoli bit**: Tentare di azzerare il registro con una `AND`, e poi eseguire un salto condizionale `JZ`.
  - Testare il bit 4:

```
AND $0x10, %AL
JZ vale_zero

// Se il bit numero 4 era 0 allora si salta all'etichetta vale_zero
// Se invece era 1 allora non si salta
```

## Mettere in un registro un dato scritto in due registri (parte alta e bassa)

Un esempio tipico di questo è dopo aver fatto una [\(I\)MUL](#), si vuole portare, ad esempio, il risultato che è stato scritto in [DX\\_AX](#) (moltiplicazione a 16 bit) dentro [EAX](#).

```
PUSH %DX // Pusho la parte alta  
PUSH %AX // Pusho la parte bassa  
POP %EAX
```

## Convertire maiuscole ⇌ minuscole

Voglio convertire un carattere ASCII che è stato messo nel registro AL:

```
// Da maiuscola a minuscola  
OR $0x20, %AL  
  
// Da minuscola a maiuscola  
AND $0x20, %AL
```

## Pulire il Carry Flag

In qualche raro caso è comodo negli esercizi di esame pulire il carry flag, e questo si può fare in due modi:

- Usando l'istruzione [CLC](#), che è stata creata apposta
  - Esiste pure la controparte per settarlo, ed è la [STC](#)
- Eseguire un'istruzione logica [AND](#), [OR](#), oppure [XOR](#), le quali resettano sia il CF che l'OF

## Invertire l'ordine dei bit o delle cifre

Gli algoritmi seguenti sono stati scritti interamente da me, quindi è molto probabile che ci siano errori o ottimizzazioni possibili da fare.

## Invertire l'ordine dei bit in un registro

- Bit da invertire in `%AL`
- Registro di appoggio dove finiranno i bit invertiti (output): `%BL`

```
XOR %BL, %BL
MOV $8, %CL // Numero di iterazioni da fare = numero di bit del registro

ciclo:
    CMP $0, %CL
    JE fine
    RCR %AL
    RCL %BL
    DEC %CL
    JMP ciclo

fine:
```

## Invertire l'ordine delle cifre di un numero

- Il numero è in un vettore `vett` di byte (`.BYTE vett`)
- Il numero di cifre è `NumeroCifre`

```
LEA vett, %ESI
XOR %ECX, %ECX

ciclo_push:
    CMP $NumeroCifre, %CL
    JE poi
    { CLD + LODSB } oppure { MOVB %ESI(%ECX), %AL + INC %ECX }
    AND $0xFF, %AL // Estendo su 16 bit per pushare
    PUSH %AX
    JMP ciclo_push

poi:
    /* CICLI DI POP: */
    LEA vett, %EDI
    XOR %ECX, %ECX

ciclo_pop:
    CMP $NumeroCifre, %CL
    JE fine
    POP %AX
```

```

{ CLD + STOSB } oppure { MOVB %AL, %ESI(%ECX) + INC %ECX }
JMP ciclo_pop

fine:

// Soluzione più semplice:
// si dichiara un altro vettore uguale al primo, si inizializza a 0,
// e si fa una copia memoria-memoria con DF=1
LEA vett, %ESI
LEA vett_inverso, %EDI
MOV $NumeroCifre, %CL
STD
REP MOVSB

```

## Verilog

### Compilazione

```

>> iverilog -Wall -Winfloop -o output desc.v testbench.v
>> vvp output
>> gtkwave waveform.vcd &
oppure puoi fare "File > Write Save File As", salvare il file come "config.gtkw"
e poi eseguire il comando gtkwave waveform.vcd config.gtkw

```

## Visualizzazione (MOLTO utile)

- **FORMATO DEI SEGNALI (HEX, BIN, DEC):** `Ctrl+a`, poi `right-click` e seleziona **Data Format**
- **COLORE DEI SEGNALI:** `Ctrl+a`, poi `right-click` e seleziona **Color Format**

## Logica

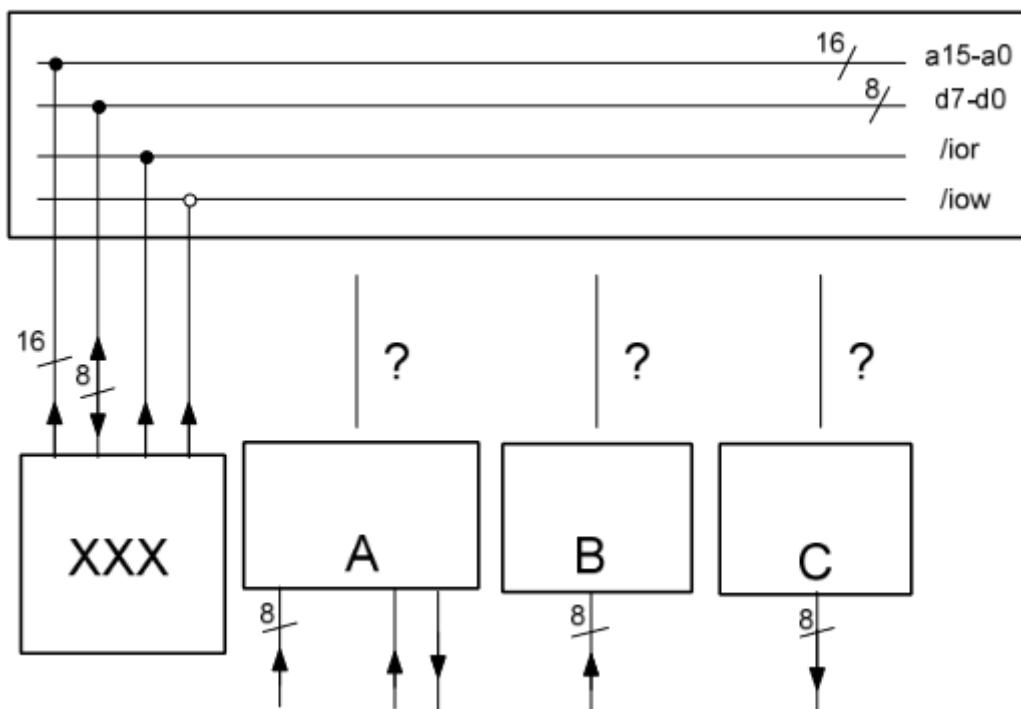


Blocchi di logica combinatoria → Assegnamento bloccante ‘ = ‘  
 Blocchi di logica sequenziale → Assegnamento non bloccante ‘ <= ‘

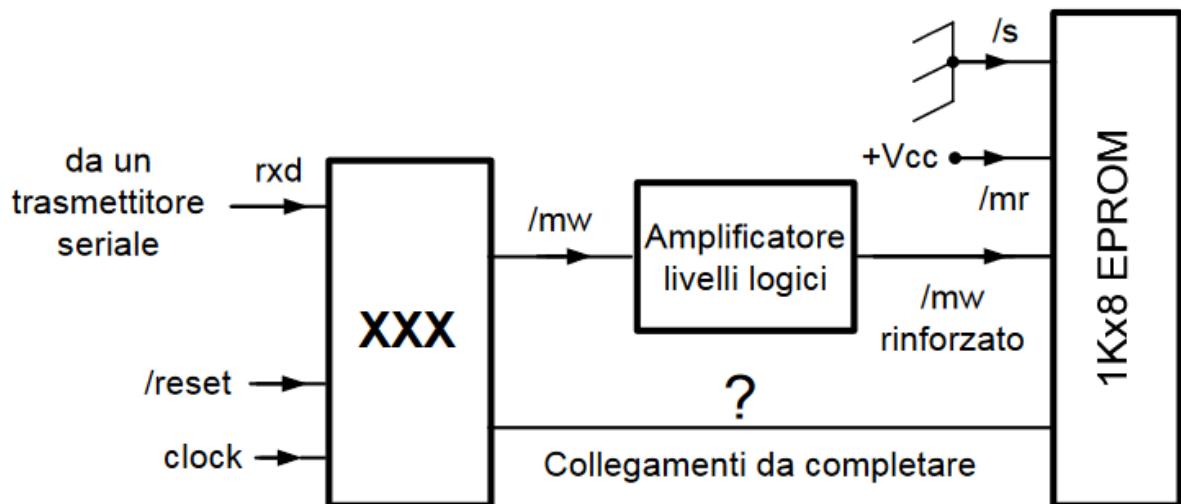


***DIR SERVE SOLO SE LE TUE USCITE LE DEVI MANDARE SU UNA LINEA CONDIVISA DOVE CI SONO PURE ALTRE RETI!***

**Esempio:**



Esempio in cui SERVE DIR!



Esempio in cui NON SERVE DIR!

## Pretest

Quelle che seguono sono delle risposte ad alcune domande dei pretest, spero possano essere utili per un veloce ripasso.



**UNA TABELLA DI VERTIA' SINTETIZZATA IN FORMA *SP* A 2LL PUO' ANCHE ESSERE SINTETIZZATA IN FORMA *PS* A 2LL!**



**I flag *FI* ed *FO* non vanno controllati in nessun modo!  
Non ha senso cercare di testarli!**



Dato il naturale  $A$ , rappresentazione dell'intero  $a$ , il segno di  $a$  si può desumere dalla cifra più significativa di  $A$  SIA SE LA RAPPRESENTAZIONE E' IN  $\sqrt{CR}$  SIA SE E' IN TRASLAZIONE!



In una RSA, cambiare un ingresso alla volta, e solo quando  $RCA$  è a regime è SOLO C.N. PERCHE' L'EVOLUZIONE DELLA RETE SIA PREVEDIBILE!

Sarebbe C.S. se l'ingresso fosse cambiato quando L'INTERA RETE E' A REGIME!



Alla fine della fase di chiamata (fetch), il registro  $IP$  contiene l'indirizzo della PROSSIMA ISTRUZIONE DA ESEGUIRE!

NON dell'istruzione la cui fase di fetch sta per iniziare, perché alla fine della fase di fetch ho già incrementato IP!



La lista degli implicanti principali di una legge combinatoria NON PUO' AVERE COSTO SUPERIORE A QUELLO DELLA FORMA CANONICA SP!



Un comparatore per interi rappresentati in traslazione e' identico ad uno per naturali!



**Data una tabella di flusso che descrive una RSA, la sintesi della RC che produce l'uscita dipende SOLTANTO DALLA CODIFICA DEGLI STATI INTERNI!**



**Il clock in una Rete di Moore è più veloce che in una di Rete di Mealy!**

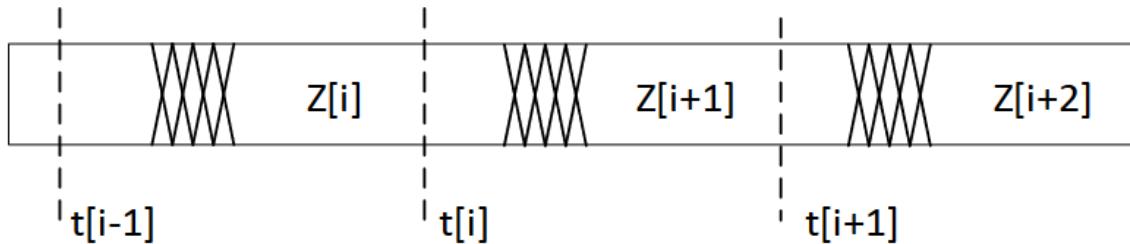


**Le RSS in cui lo stato di uscita ad un istante  $t$  NON dipende dallo stato di ingresso presente all'istante  $t$  sono quelle di MOORE e di MEALY RITARDATO!**



**La rete che calcola il complemento di un numero è una barriera di NOT SOLO se il numero è rappresentato in base 2!  
NON vale se è rappresentato BCD! Fai una prova per convincerti!**

---



La temporizzazione sopra scritta riguarda lo stato di uscita di una rete sincronizzata. Tale rete è:

- Di Moore o di Mealy**
  - Di Moore o di Mealy ritardato
  - Di Mealy o di Mealy ritardato
  - Nessuna delle precedenti
- 



#### Criterio per la determinazione dell'OF della somma fra interi:

- OPERANDI **DISCORDI** → **RISULTATO SEMPRE CORRETTO**
- OPERANDI **CONCORDI** → **RISULTATO CORRETTO SOLO SE E' CONCORDE CON GLI OPERANDI!**



**Date due sintesi A COSTO MINIMO della stessa legge combinatoria,**  
ma

- Una in Forma SP
- Una a Porte NOR

**I LORO COSTI NON SONO CORRELATI!**



**Se la divisione intera fra due numeri è fattibile, allora pure la divisione naturale fra i moduli è fattibile!**



**Convertitori D/A → Errore di Non Linearità**

**Convertitori A/D → Errore di Non Linearità + Errore di Quantizzazione**



**La ROM non contiene i valori iniziali da dare ai registri al reset del calcolatore, ma contiene il CODICE che li inizializza!**