

# Reti Sequenziali

Giovanni Stea

Corso di Laurea in Ingegneria Informatica, Università di Pisa

a.a. 2020/21

# Materiale didattico

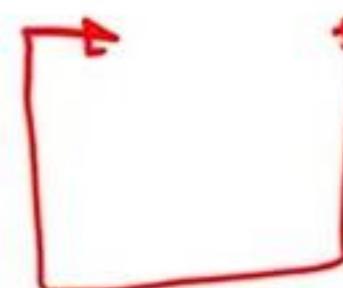
- Paolo Corsini, "Dalle porte AND, OR, NOT al Sistema calcolatore", edizioni ETS
- Appunti sulle Reti Sequenziali - versione 25/09/2020 (con esercizi svolti)

(Sistemi d. (Sistemi di Porte))

sistemi di

# La funzione di memoria

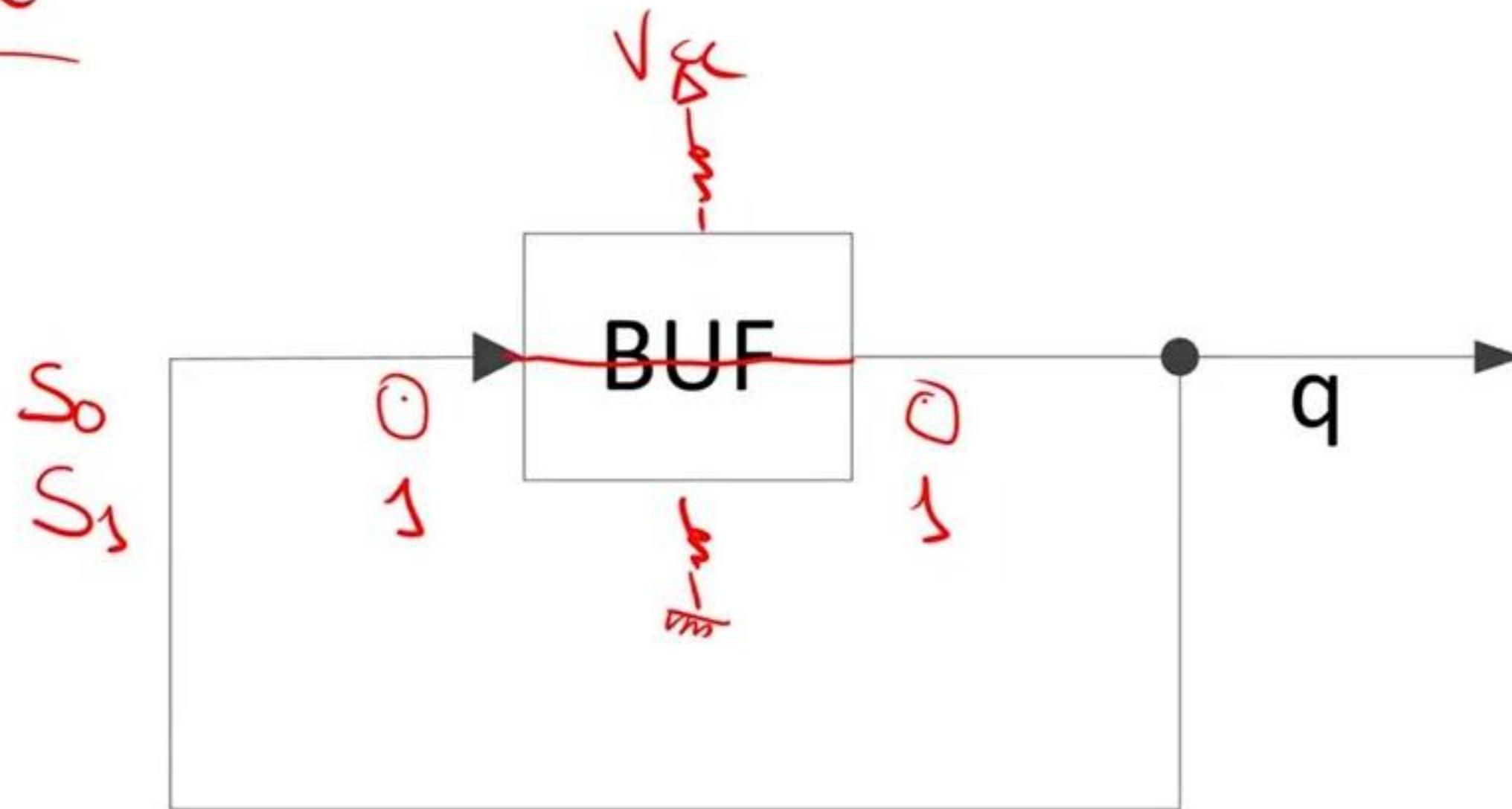
- Le reti combinatorie sono **prive di memoria**:
  - Lo stato di uscita dipende solo dallo stato di ingresso **presente**
- Per avere reti **sequenziali**, cioè reti la cui uscita dipenda dalla **sequenza degli stati di ingresso** visti dalla rete fino a quel momento, è necessario dotare le reti di **memoria**, cioè della capacità di **ricordare** quella sequenza.
- La memoria si implementa tramite **anelli di retroazione**.



# Esempio

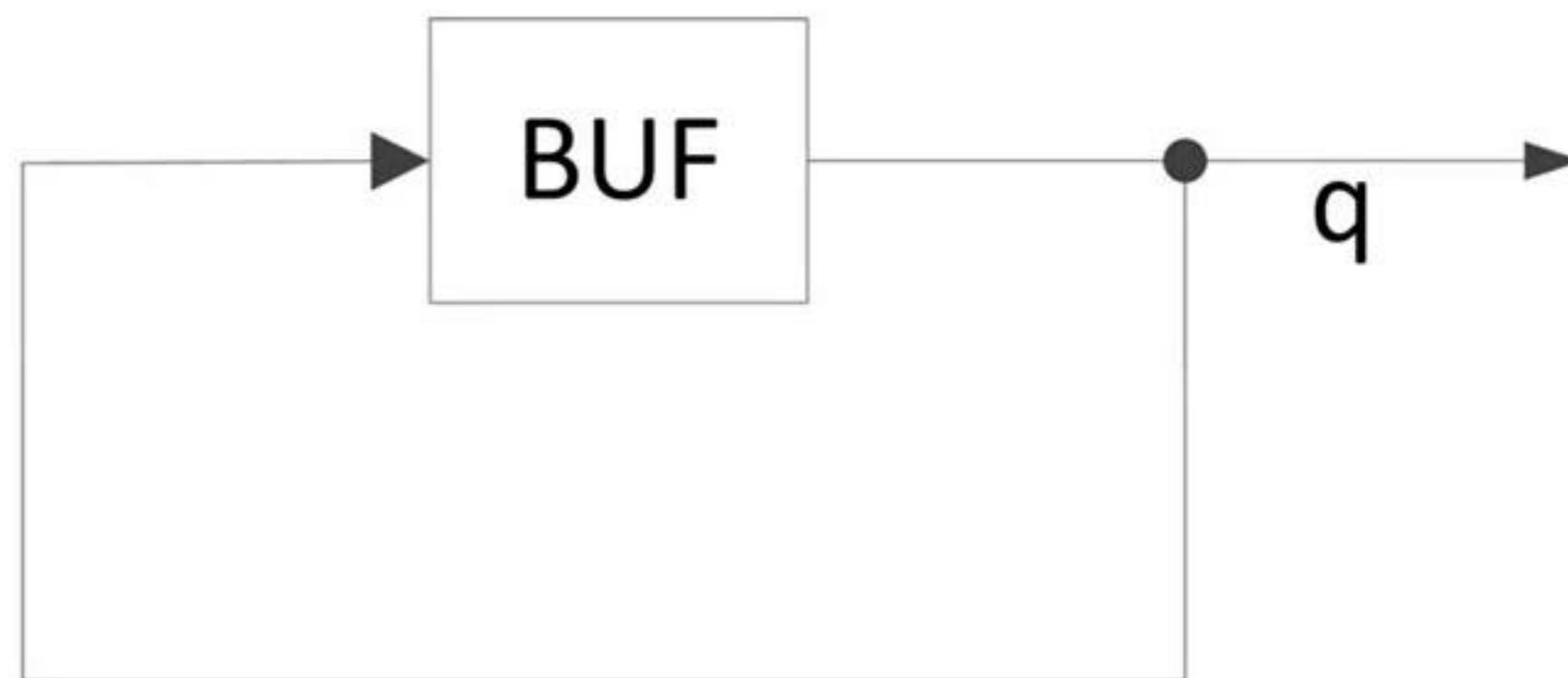
Casuale

- In questo anello esistono **due situazioni di stabilità**:
  1. L'uscita vale 0 (e va in ingresso al buffer, dove si rigenera)
  2. L'uscita vale 1 (idem)
- L'anello si può trovare **in due stati**  $S_0, S_1$  che corrispondono allo stato in cui l'uscita vale 0 ed 1 rispettivamente.
- **La presenza del buffer è fondamentale**: garantisce che a  $q$  sia associato un valore logico.
  - Se tolgo il buffer,  $q$  è un filo staccato.



## Esempio (cont.)

- Una rete fatta così **non serve a niente**
  - non si può impostare né modificare il valore di  $q$ .
- Quando accendo il sistema, questo si porta in uno dei due stati  $S_0, S_1$  in maniera **non controllabile**, e lì resta finché non spengo
- Non è possibile che questo anello memorizzi bit diversi in tempi diversi

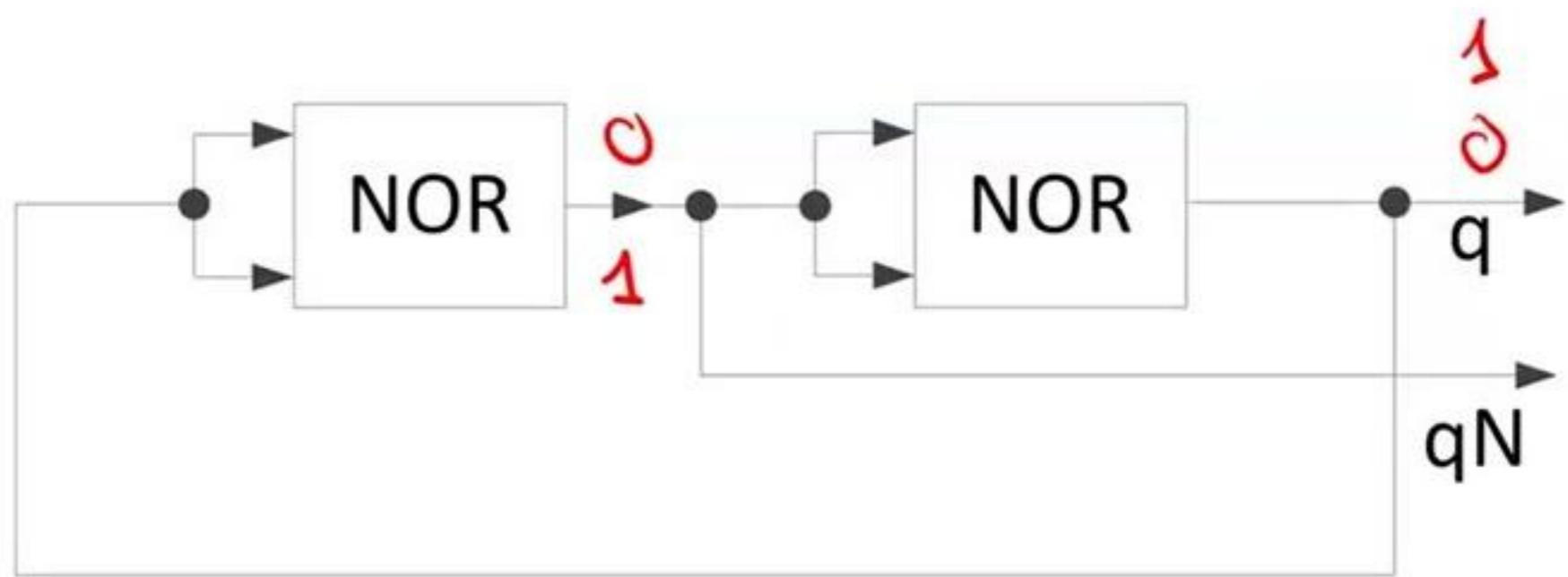


## Esempio (cont.)

- Posso:
  - Sostituire il buffer con una coppia di NOT
  - Implementare ciascun NOT a porte NOR

- Nel circuito che ottengo sono presenti contemporaneamente sia il bit 1 che il bit 0.

- se  $q = 1$ , allora tra le due NOR c'è 0
- Se  $q = 0$ , allora tra le due NOR c'è 1



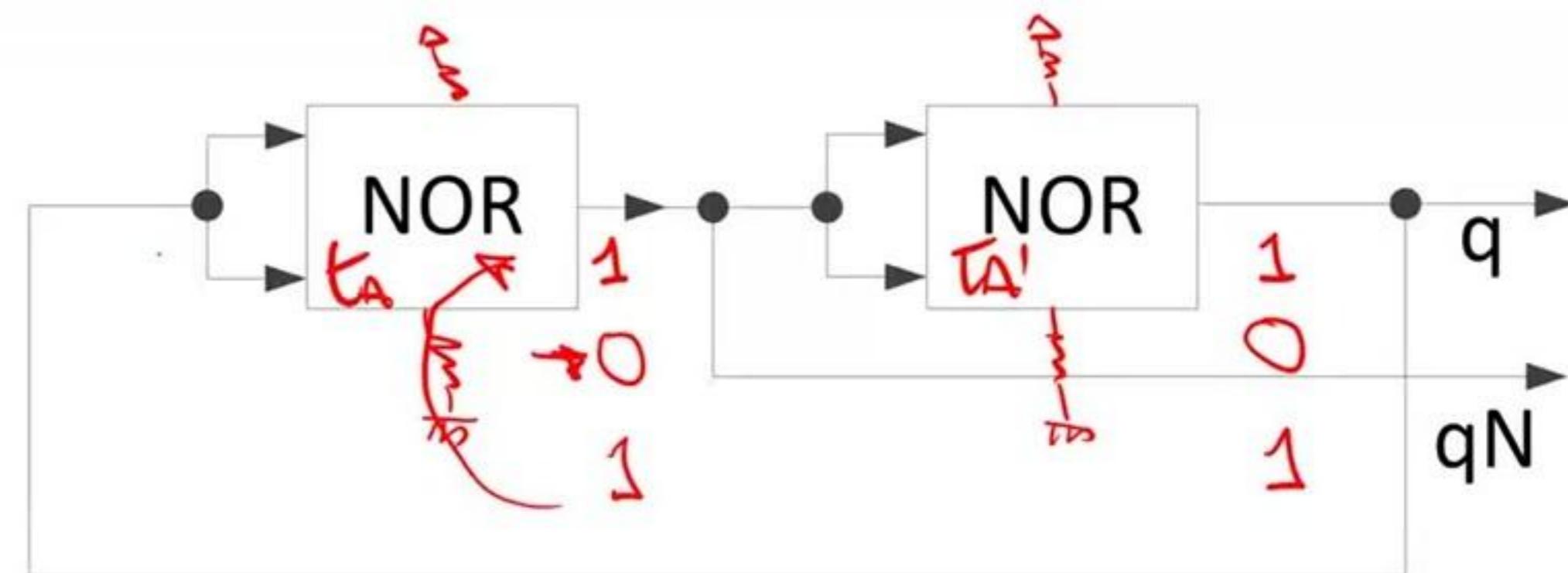
$$\begin{matrix} s_0 \\ s_1 \end{matrix}$$

$$q_N = \overline{q}$$

- Possiamo **dotare il circuito di un'altra uscita**, che chiamo  $q_N$  (negata). Per convenzione, si dice che il circuito **memorizza il bit il cui valore è quello di  $q$ .**

## Esempio (cont.)

$S_0$  |  
 $S_1$  |  
 $t_A$



- Se all'accensione  $q$  e  $q_N$  sono discordi, la rete si trova già in uno dei due stati stabili, e lì resta.
- Se, invece,  $q$  e  $q_N$  sono concordi, in teoria ciascuna delle due uscite oscilla all'infinito, con un tempo pari al tempo di risposta delle porte.
  - In pratica, invece, la **rete si stabilizza velocemente**, perché comunque il tempo di risposta delle due porte sarà **diverso**, e quindi si creerà immediatamente una situazione in cui  $q$  e  $q_N$  sono discordi, situazione che rimane stabile.

0	1
1	0

x	y	NOR
0	0	1
0	1	0
1	0	0
1	1	0

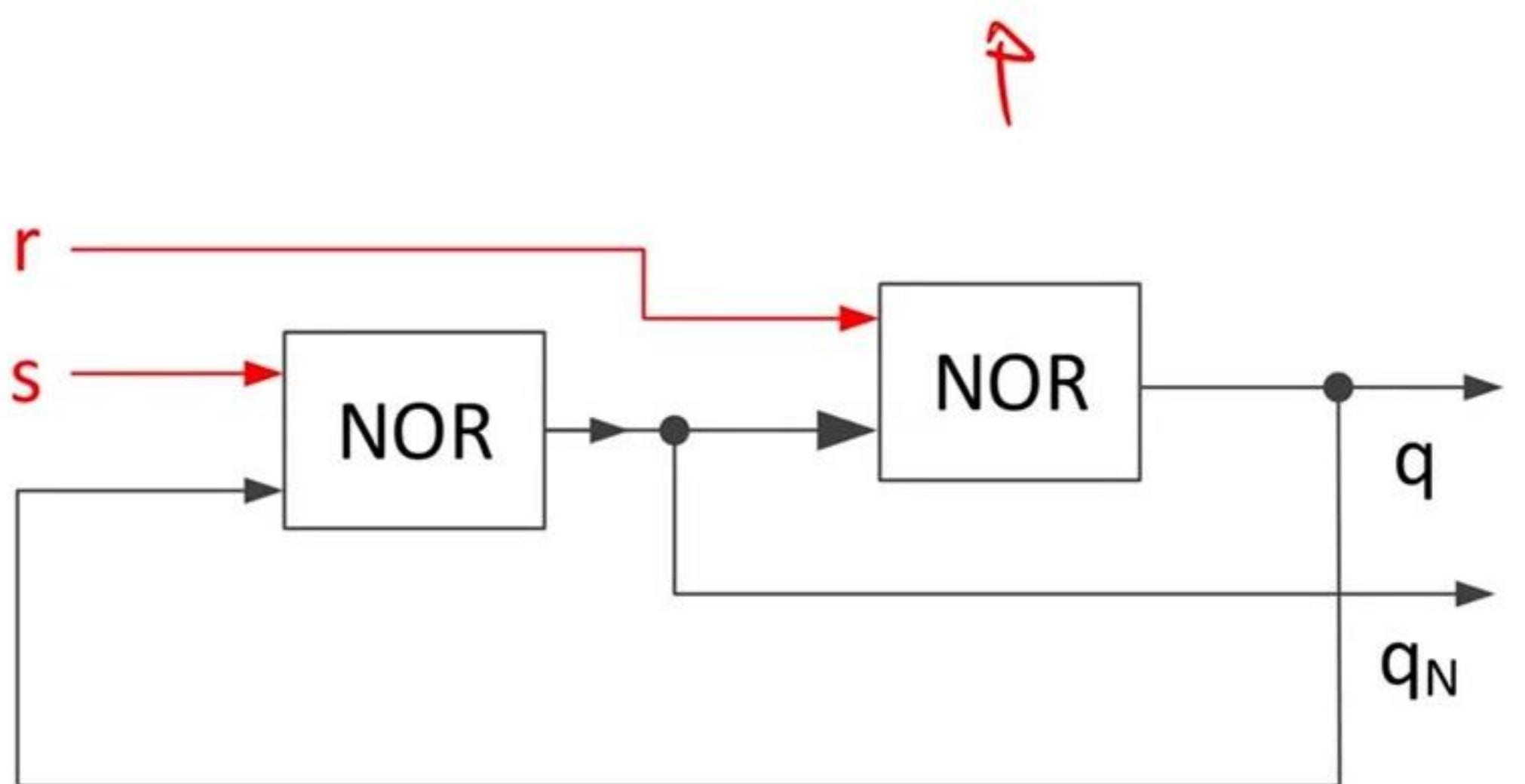
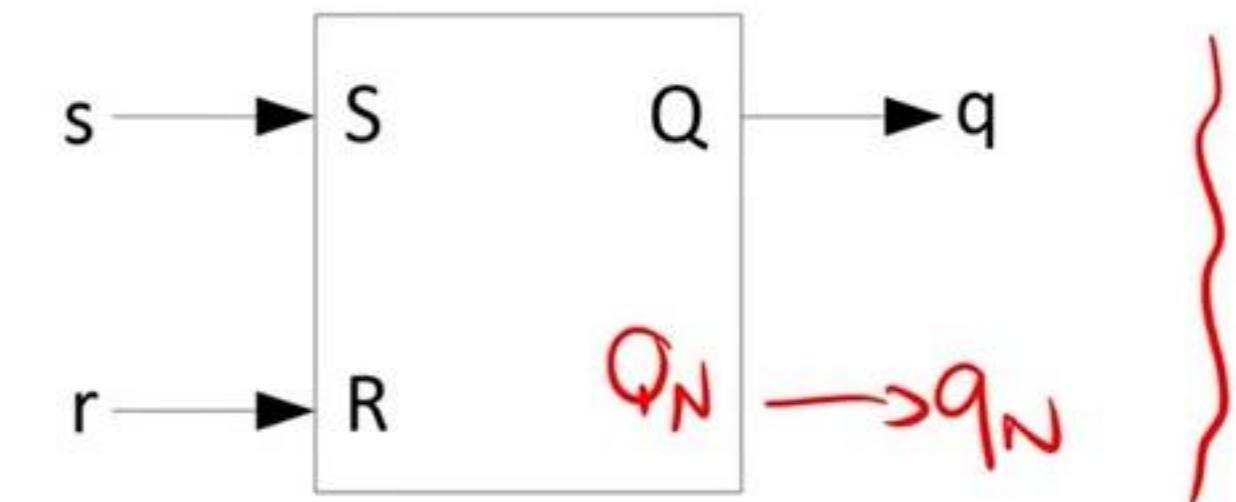
## Il latch SR

|P|P|P|  
attivo basso

- Trasformo uno degli ingressi di ciascuna NOR in un **ingresso pilotabile dall'esterno**

- Latch SR** (a volte detto, impropriamente, Flip-flop SR)

- S=Set 1 ○
- R=Reset 1 ○
- Ingressi «attivi alti» ○ 1



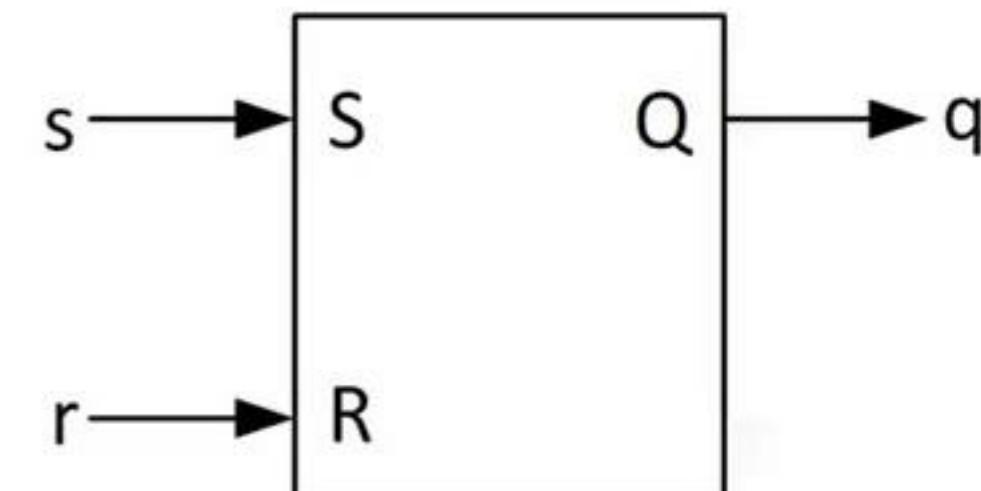
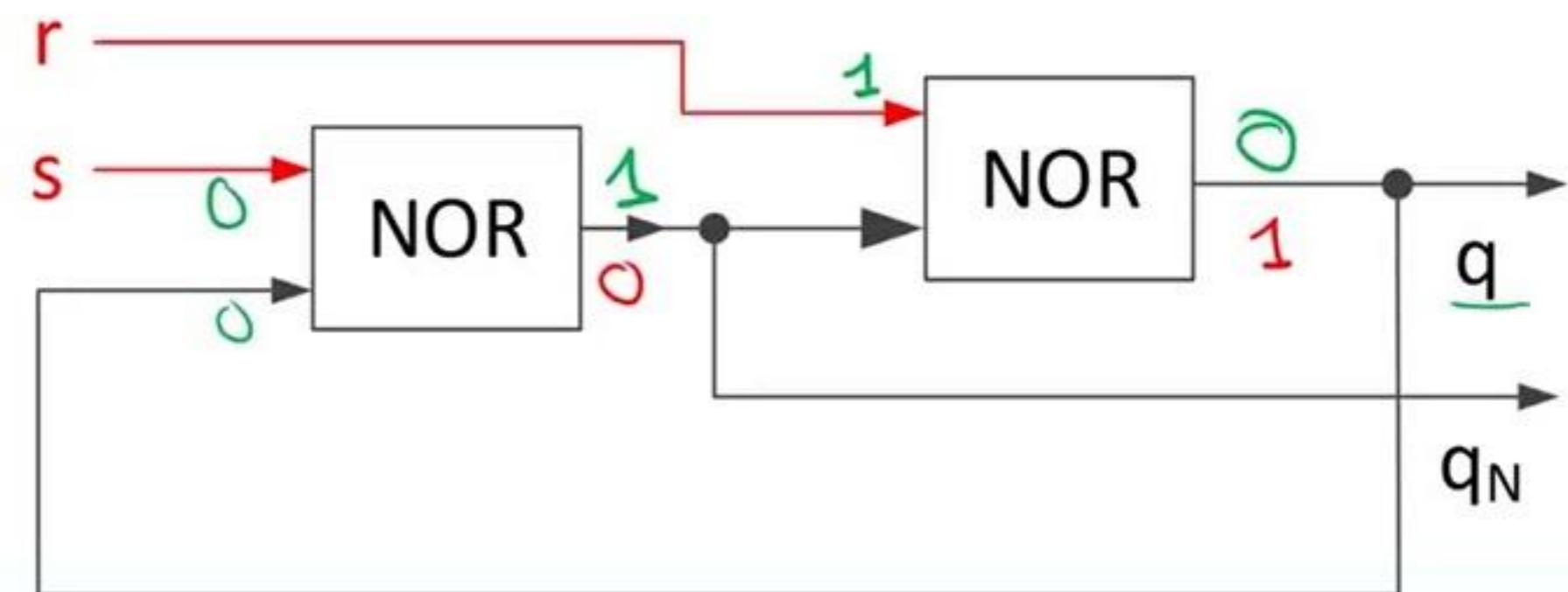
## Il latch SR (cont.)

- $s = 1, r = 0$ :

- la **prima NOR** ha **un ingresso a 1**, quindi mette l'uscita a 0 (qualunque sia il valore di  $q$ ). Pertanto,  $q_N = 0$ .
- La **seconda NOR** ha in ingresso **00**, quindi mette l'uscita  $q = 1$ .
- La rete si porta nello stato  $S_1$ , in cui memorizza il bit 1. In altre parole, l'uscita si setta.

- $s = 0, r = 1$ :

- la **seconda NOR** ha **un ingresso a 1**, quindi mette l'uscita a 0 (qualunque sia il valore di  $q_N$ ). Pertanto,  $q = 0$ .
- La **prima NOR** ha in ingresso **00**, quindi mette l'uscita  $q_N = 1$ .
- La rete si porta nello stato  $S_0$ , in cui memorizza il bit 0. In altre parole, l'uscita si resetta.



x	y	NOR
0	0	1
0	1	0
1	0	0
1	1	0

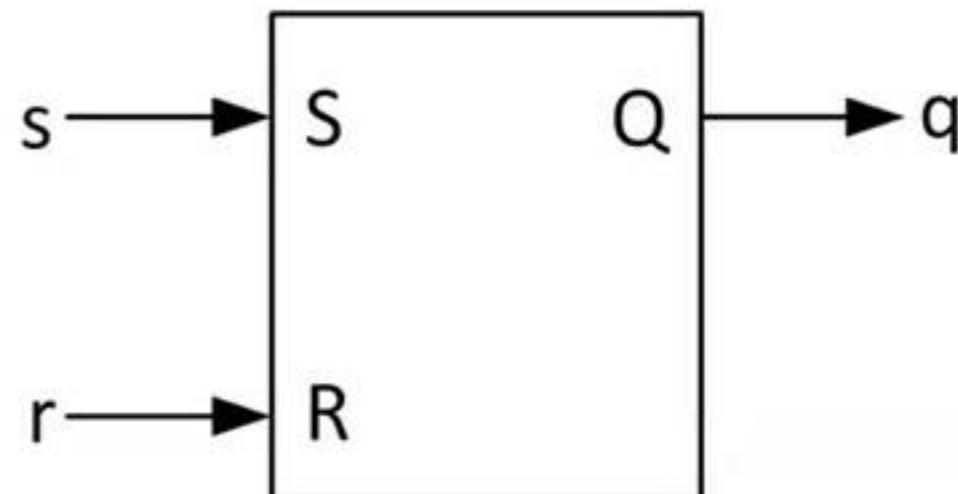
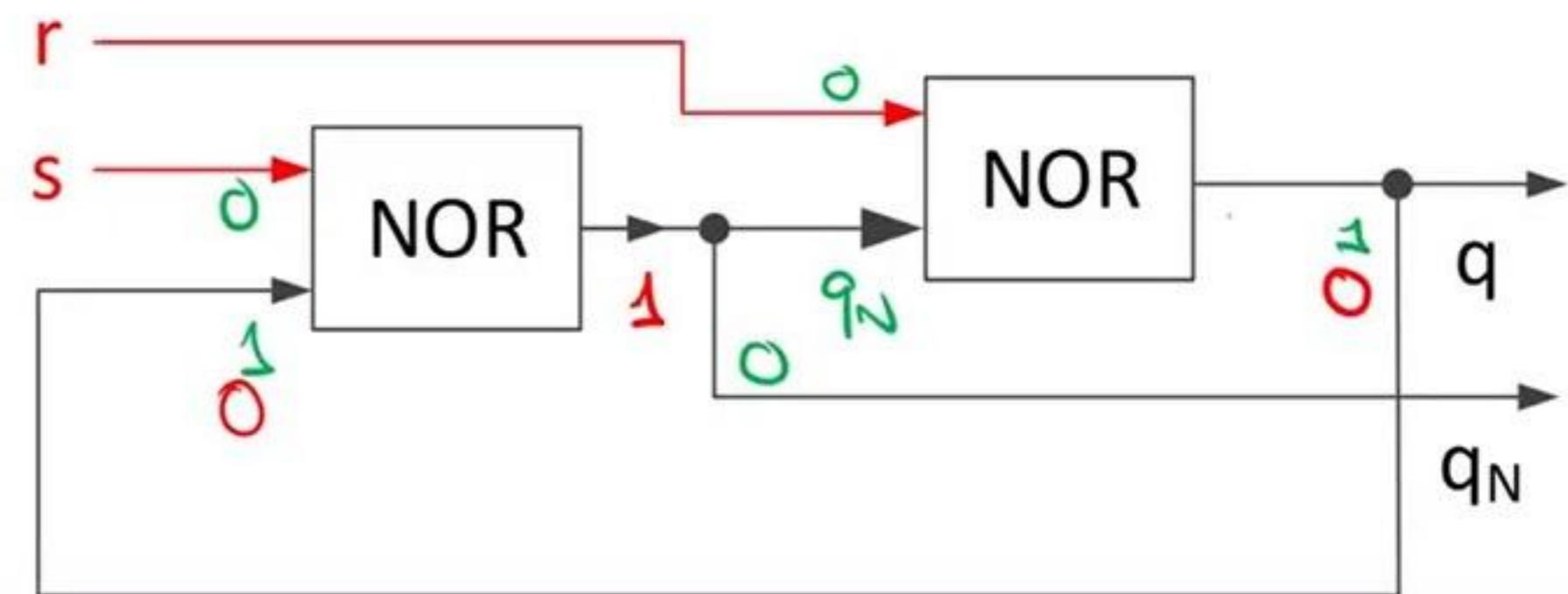
# Il latch SR (cont.)

*conservazione*

RSA

- ~~$S = 0, r = 0:$~~

- l'uscita della prima NOR vale **0** se  $q = 1$ , e vale **1** se  $q = 0$ .  
Pertanto,  $q_N = \bar{q}$ .
- La seconda NOR ha in ingresso **0 e  $q_N$** , quindi l'uscita  $q$  vale  $\underline{\bar{q}_N}$ .
- L'uscita **conserva il valore che aveva precedentemente.**



- Questo rende la rete **una rete sequenziale**:

- quando lo stato di ingresso è  $s=0, r=0$ , la rete **rimane nello stato stabile,  $S_0$  o  $S_1$** , nel quale si è portata in precedenza.
- In altre parole, **ricorda** l'ultimo comando (set o reset) ricevuto.
- **Asincrona**, perché l'uscita è costantemente adeguata all'ingresso

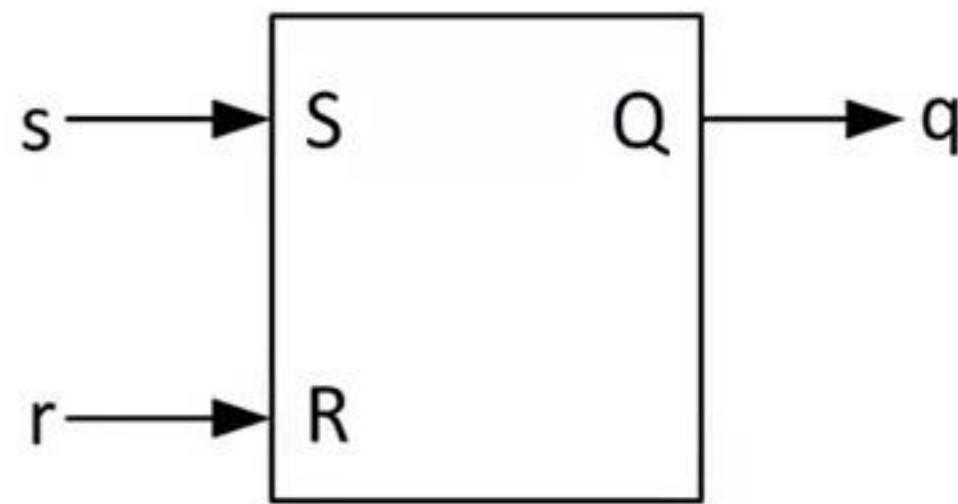
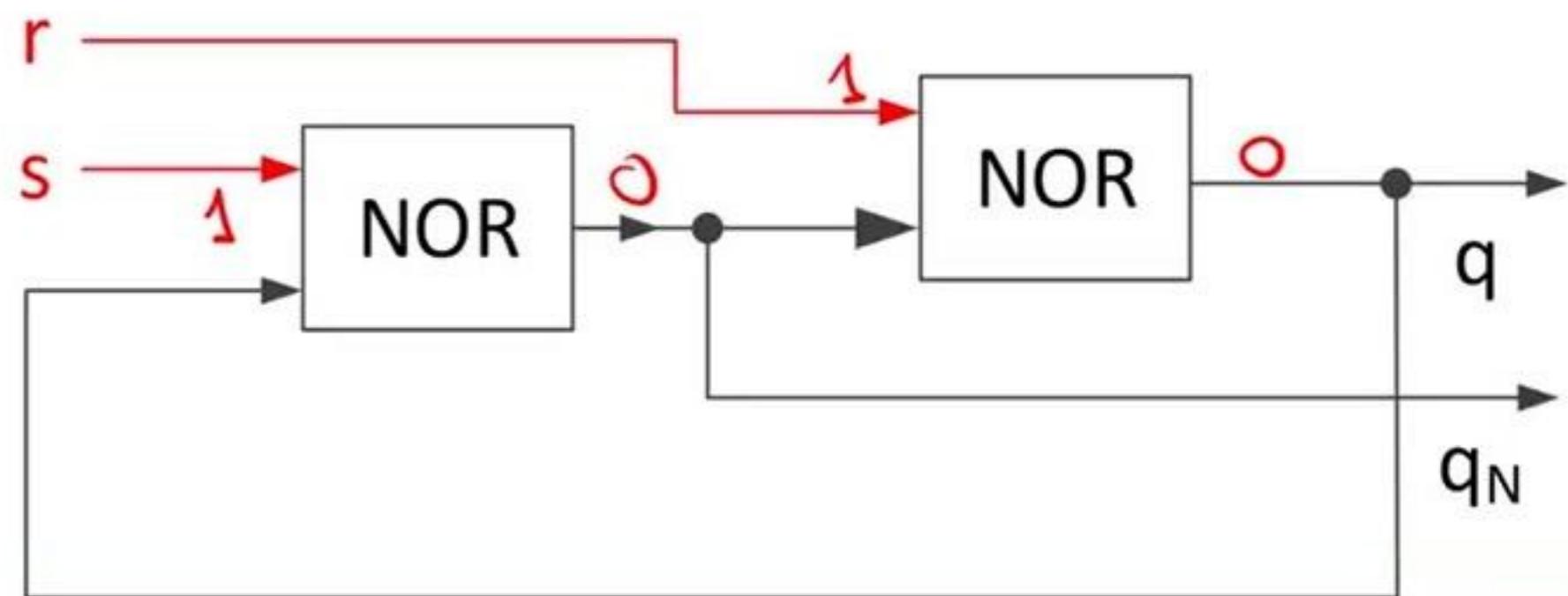
x	y	NOR
0	0	1
0	1	0
1	0	0
1	1	0

## Il latch SR (cont.)

- $s = 1, r = 1$ :

- In questo caso, **entrambe le uscite valgono 0**, e **contraddicono la regola** che vuole che siano l'una la versione negata dell'altra.
- Questo stato di ingresso non è permesso in un corretto pilotaggio.

$$q = q_N = \phi \quad q = \overline{q}_N$$



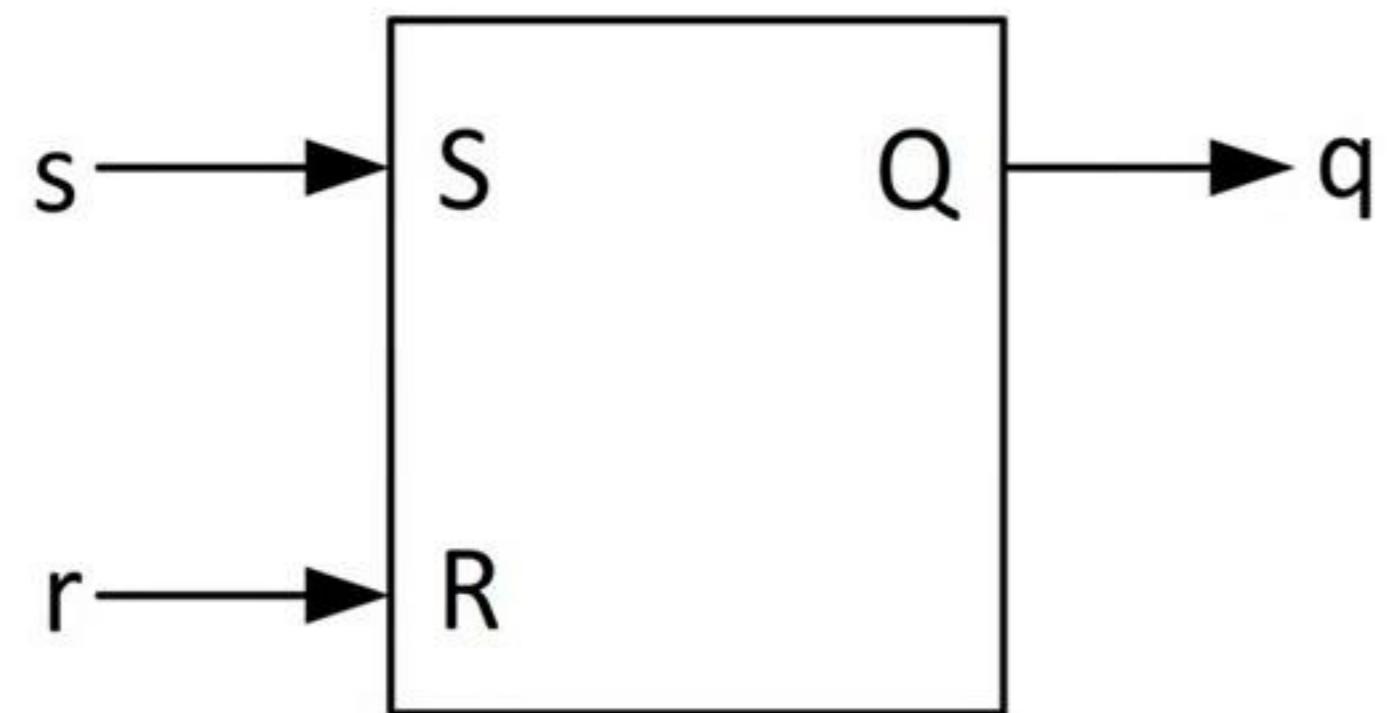
$$S=1 \quad r=0$$

$$S=0 \quad r=1$$

x	y	NOR
0	0	1
0	1	0
1	0	0
1	1	0

# Descrizione del latch SR

- **Tabella di applicazione** (non confondere con tabella di verità)
  - A sinistra il valore **attuale** della variabile (in questo caso, l'uscita  $q$ ) e il valore **successivo** che si vuole che questa assuma.
  - A destra viene specificato il **comando da dare alla rete** perché l'uscita passi dal valore attuale a quello successivo.



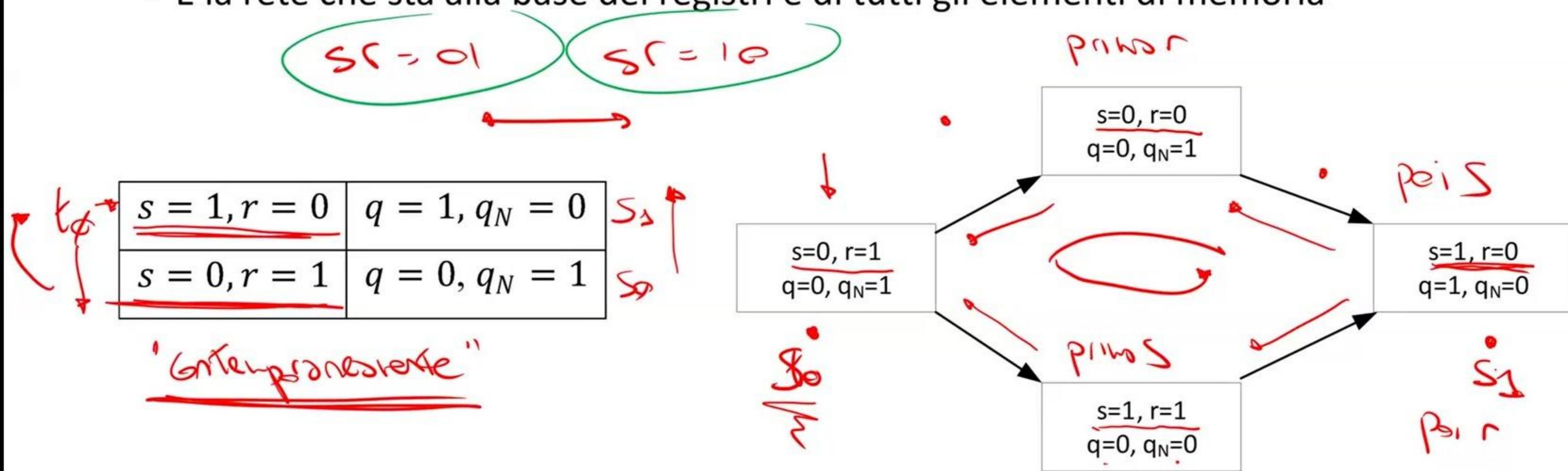
		q		q'		s r		comand	
		0	0	0	1	1	0	0	1
		0	0	1	0	1	0	00	01
		0	1	0	1	0	1	00	10
		1	0	0	1	0	1		
		1	1	-	0	-	0		

# Regole di pilotaggio del latch SR

- Per le reti combinatorie conosciamo due regole:
  - 1) Pilotaggio in modo fondamentale: cambiare gli ingressi soltanto quando la rete è a regime
  - 2) Stati di ingresso consecutivi devono essere adiacenti
- La regola 1) deve essere rispettata. Anche per le RSA esiste un **tempo di attraversamento**, dal quale possiamo desumere quando variare gli ingressi.
- In generale la regola 2) è di **importanza fondamentale** nelle RSA. Infatti, se non viene rispettata, possono presentarsi in ingresso degli stati transitori spuri, e l'evoluzione delle uscite diventa **non prevedibile**
  - Ma per il latch SR posso non rispettarla

# Regole di pilotaggio del latch SR (cont.)

- Il latch SR è **robusto nei confronti di pilotaggi scorretti**.
  - È un punto di forza
  - È la rete che sta alla base dei registri e di tutti gli elementi di memoria

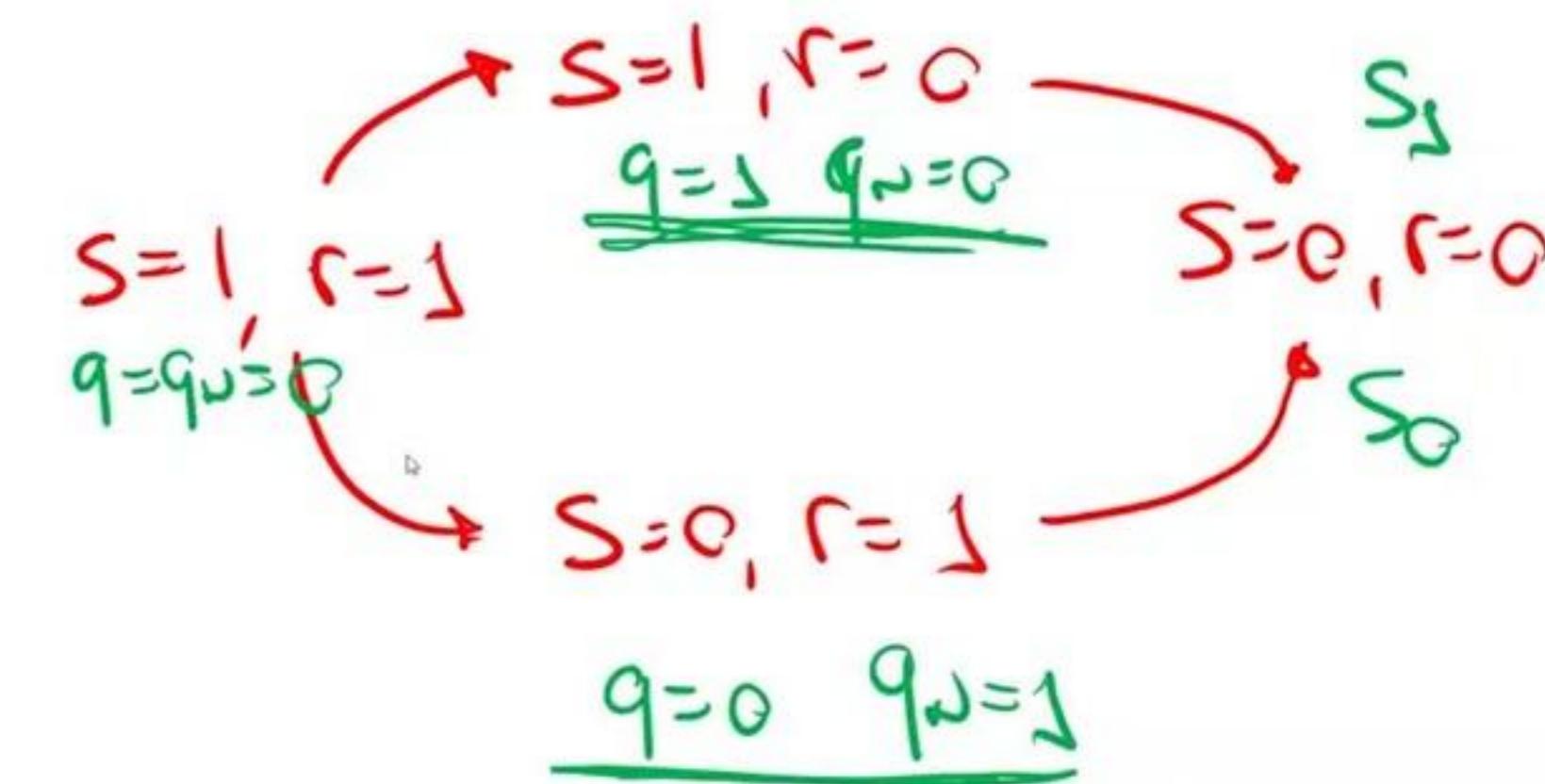


# Regole di pilotaggio del latch SR (cont.)

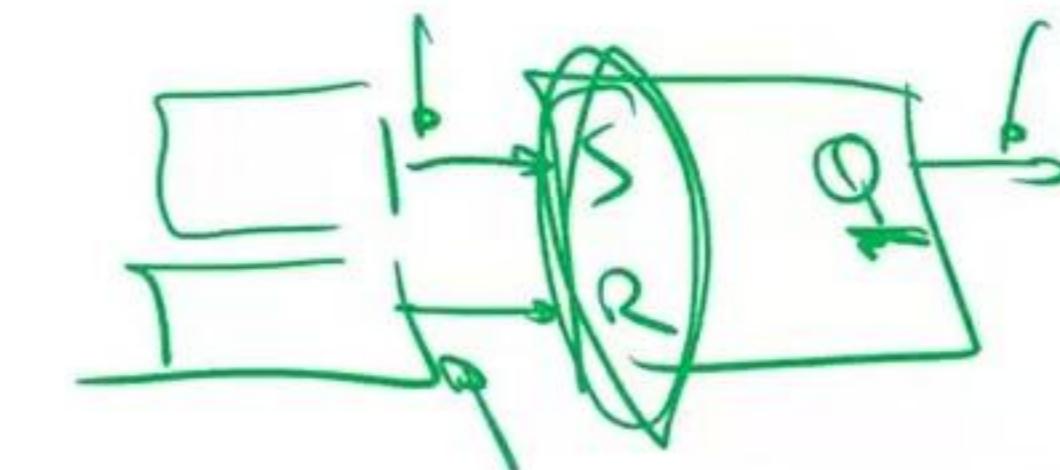
- Quello che **non deve mai succedere** è:
- a partire da uno stato di ingresso  $s = 1, r = 1$ 
  - che peraltro è un pilotaggio non consentito
- Si passi a  $s = 0, r = 0$ .
- In questo caso, il **primo dei due ingressi che transisce a zero determina lo stato in cui il latch SR si stabilizza**.
- Fare questo implica generare un'uscita casuale

$\sim 2-3$

- Il **tempo** che ci mette un **latch SR a stabilizzarsi** è di pochi nanosecondi

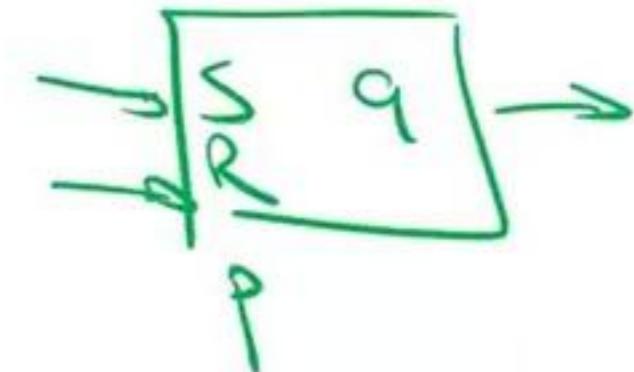
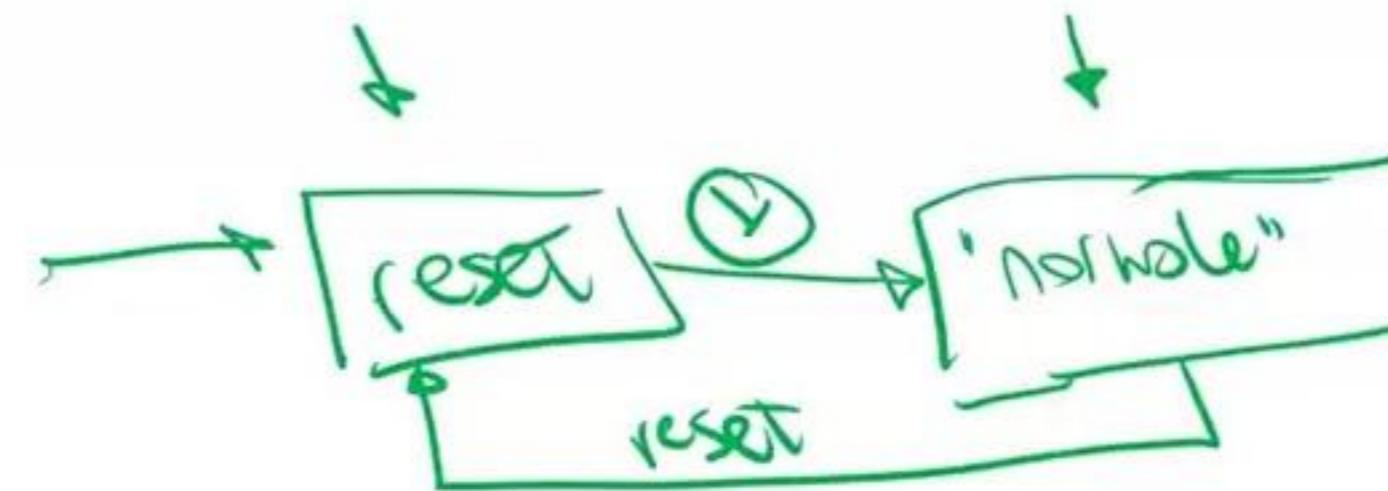


# Il problema dello stato iniziale



- Il latch SR è l'elemento **alla base dei circuiti di memoria**.
  - All'accensione, il bit contenuto nell'SR (il suo stato interno) è casuale.
- All'accensione del calcolatore, alcuni elementi di memoria **possono** avere un **contenuto casuale**
  - celle della memoria RAM
- Altri no
  - i registri del processore EF ed EIP
- Serve quindi un modo per **inizializzare un elemento di memoria** al valore voluto, cioè per impostarne l'uscita al valore desiderato all'accensione.
- L'inizializzazione avviene tutte le volte che si preme il pulsante di reset del calcolatore.

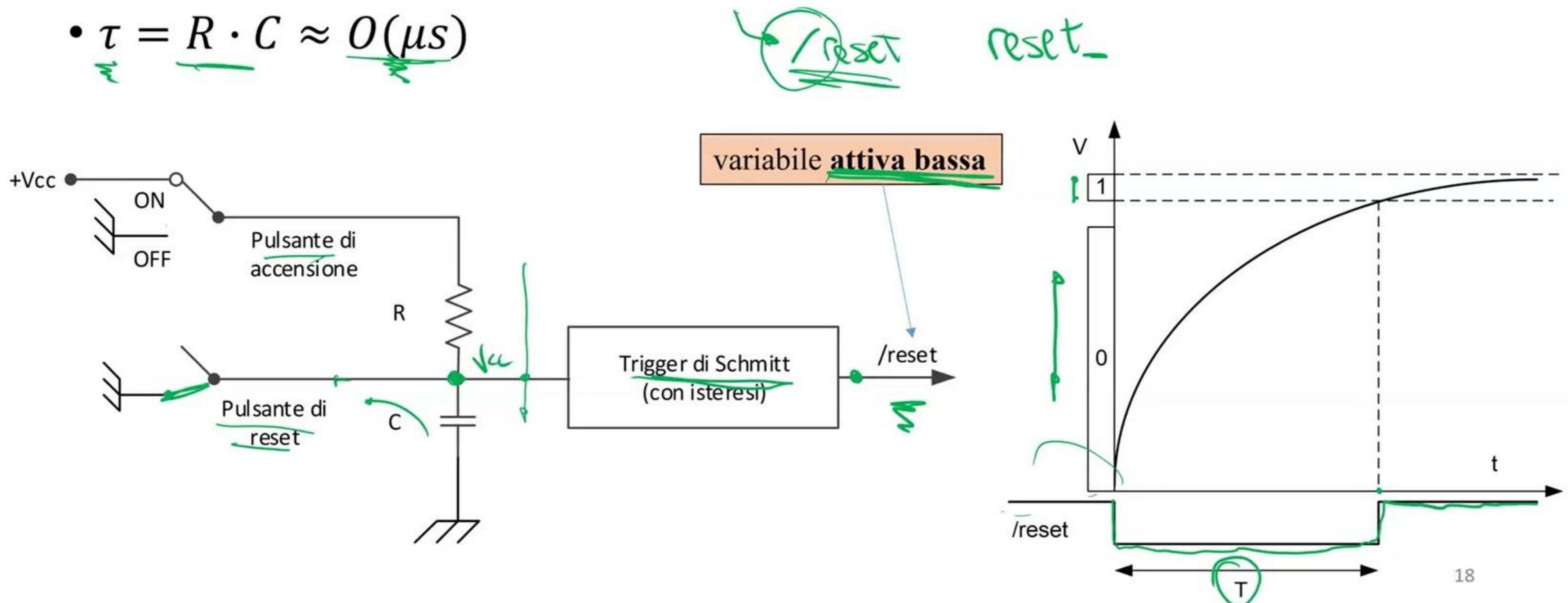
## Fase di reset



- In un sistema di elaborazione si definisce **fase di reset** una fase, distinta da quella di **normale operatività**, nella quale si inizializzano gli elementi di memoria.
- Problema di **nomenclatura**: con il nome reset si intendono due cose:
  - Un comando del latch SR, che ne mette a zero l'uscita;
  - La fase di inizializzazione degli elementi di memoria di un sistema di elaborazione, precedente alla sua operatività.
- Non è vero che gli elementi di memoria contengono tutti zeri quando si accende un calcolatore

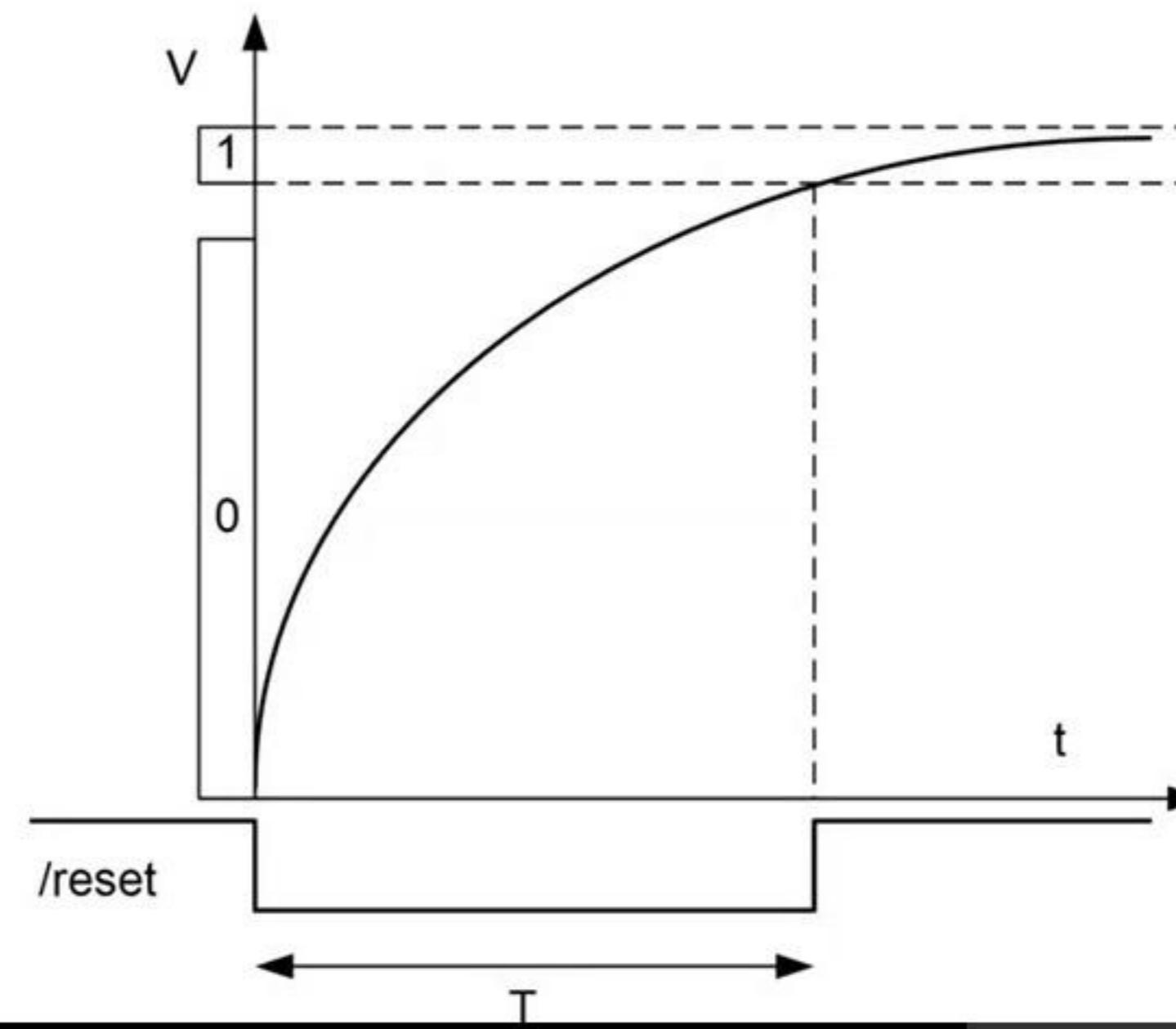
# Circuiteria per l'inizializzazione al reset

- Un sistema di elaborazione è dotato della seguente circuiteria
- $\tau = R \cdot C \approx 0(\mu s)$



# Circuiteria per l'inizializzazione al reset (cont.)

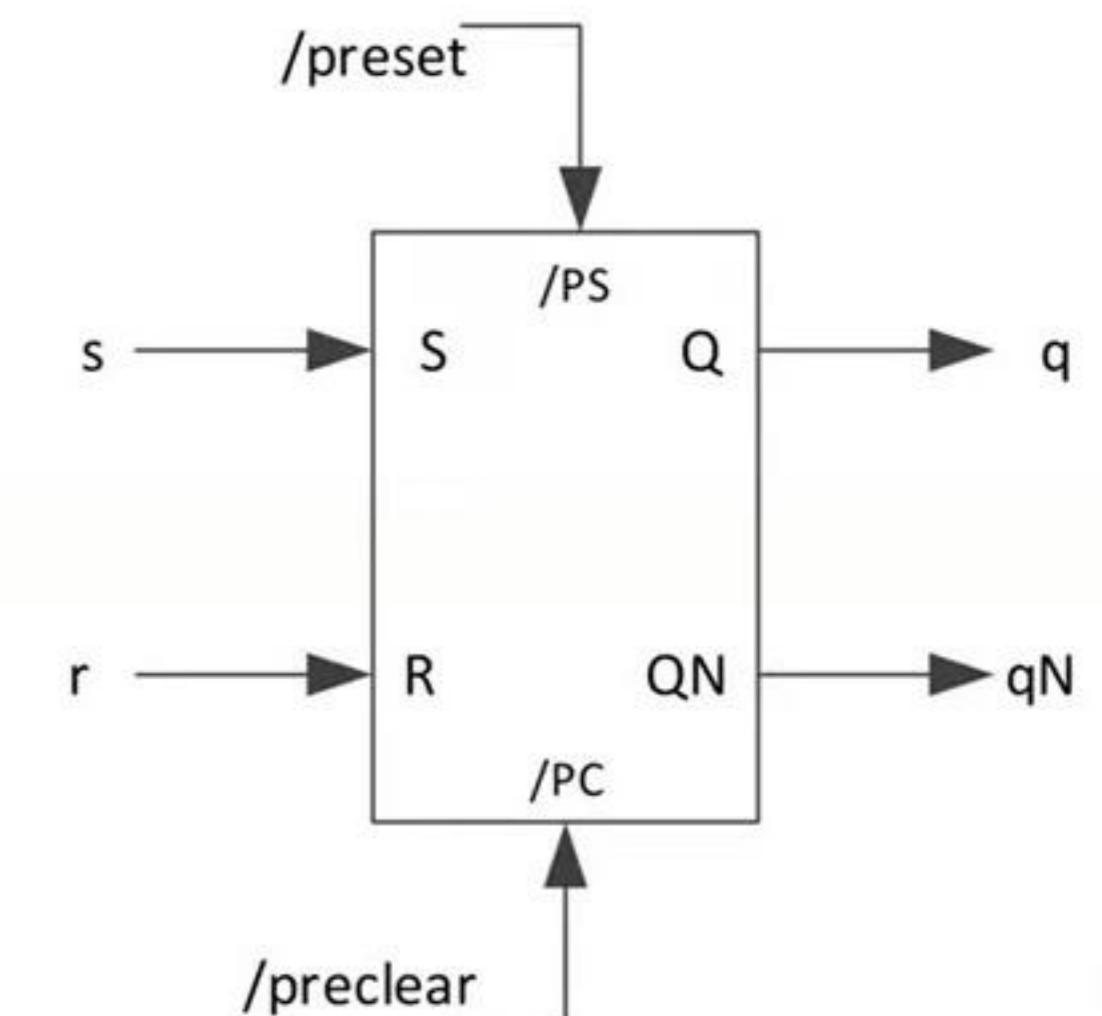
- La variabile logica  $/reset$  può essere usata per inizializzare gli elementi di memoria
  - •  $/reset = 0$ : l'elemento di memoria si porta nello stato interno iniziale desiderato, **indipendentemente dal valore dei suoi altri ingressi**
  - •  $/reset = 1$ : l'elemento di memoria funziona normalmente



# Ingressi di inizializzazione del latch SR

- Dotare un latch SR di due ingressi aggiuntivi, detti /preset e /preclear, entrambi attivi bassi:
    - $/preset = /preclear = 1$ , la rete si comporta come un latch SR "Normale operatività"
    - $/preset = 0$ : la rete si porta nello stato  $S_1$  (indipendentemente dal valore di s e r)
    - $/preclear = 0$ : la rete si porta nello stato  $S_0$  (indipendentemente dal valore di s e r)
    - $/preset$  e  $/preclear$  mai contemporaneamente a 0.

present



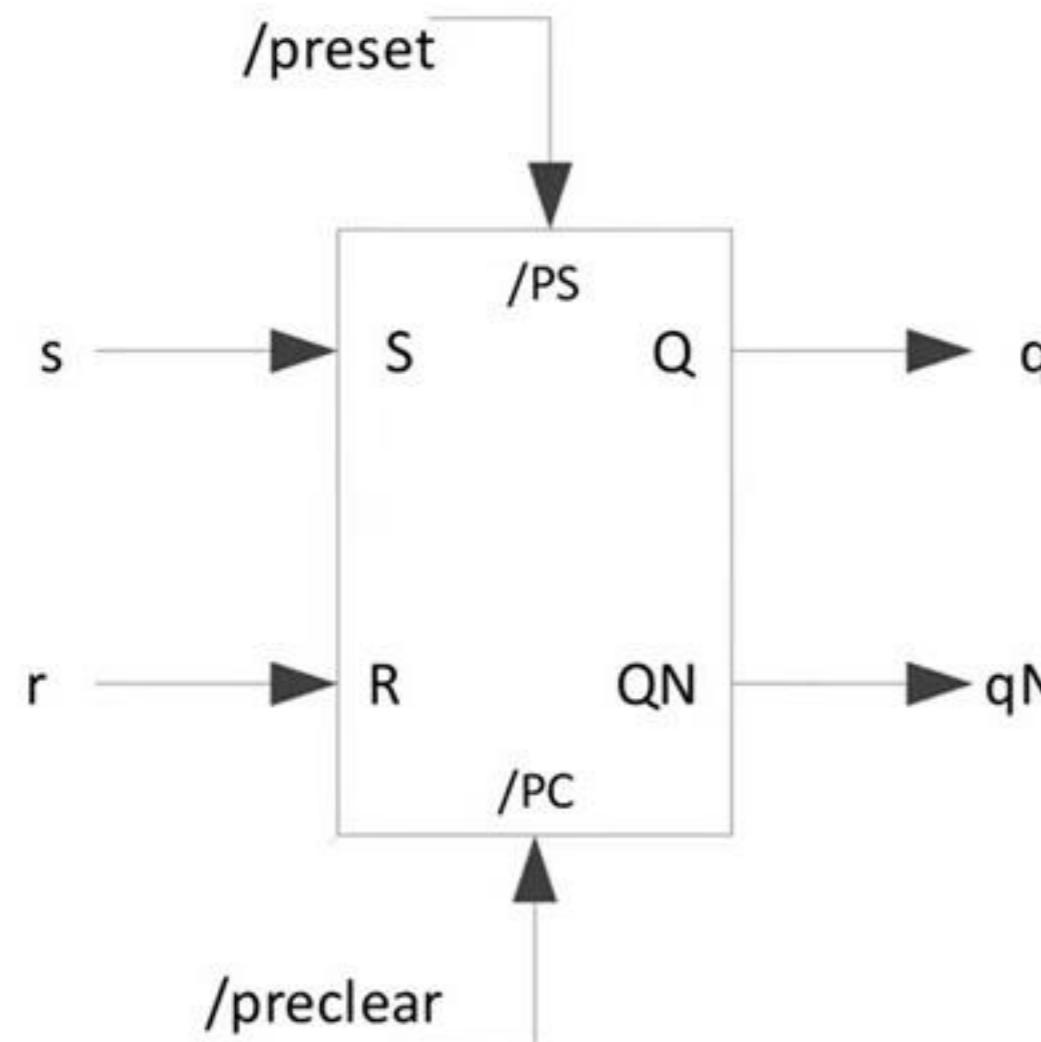
# Ingressi di inizializzazione del latch SR (cont.)

- Inizializzare il latch SR:

$S_1$

- Al valore 1:

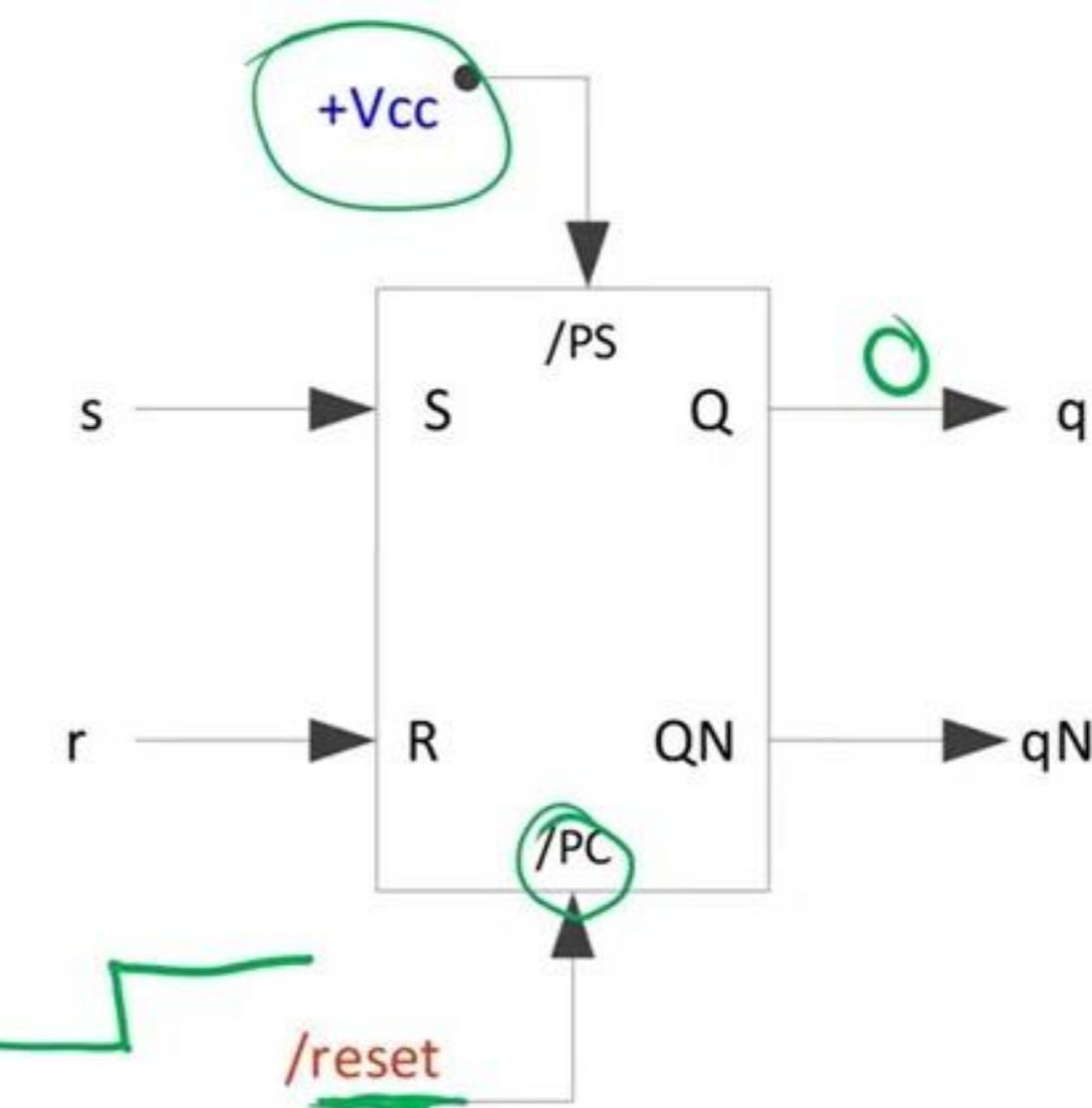
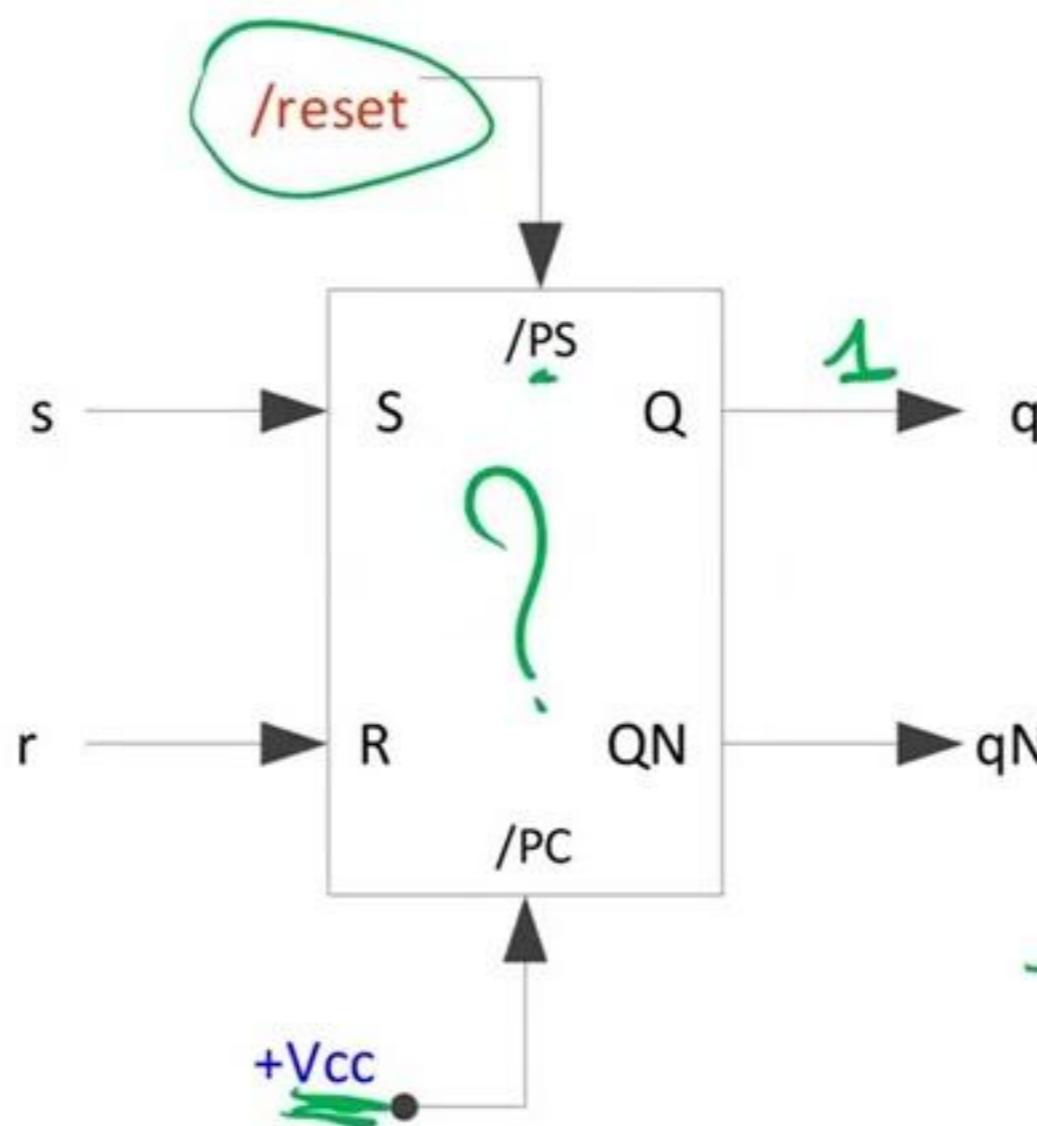
- $/preset \leftarrow /reset$
- $/preclear \leftarrow V_{cc}$



$S_0$

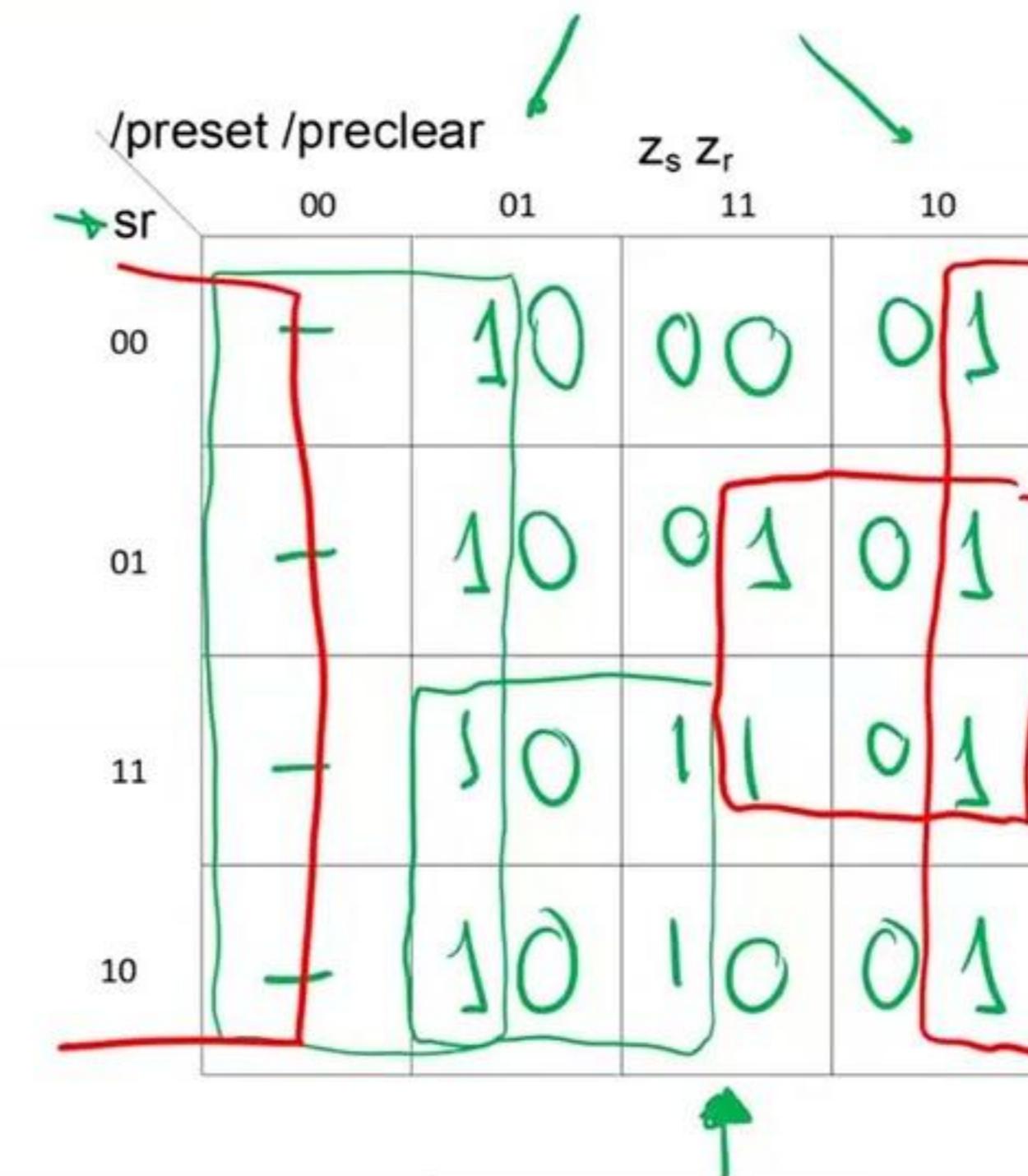
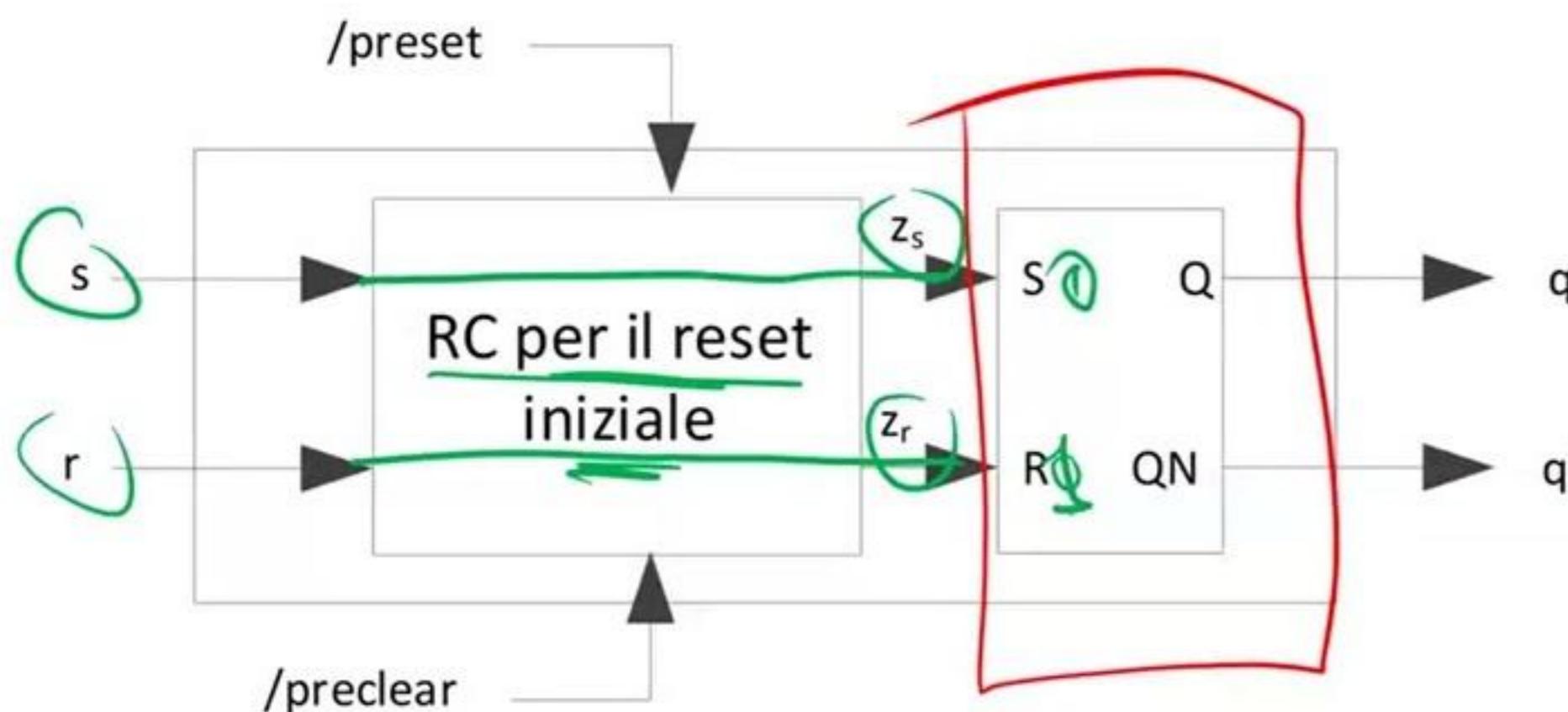
- Al valore 0:

- $/preset \leftarrow V_{cc}$
- $/preclear \leftarrow /reset$



# Ingressi di inizializzazione del latch SR (cont.)

- Devo modificare la sintesi del latch SR
- Conviene mettergli davanti una rete combinatoria, che ha come ingresso  $s$ ,  $r$ , /preset, /preclear, ed in uscita due variabili  $z_s$ ,  $z_r$



$$z_s = \overline{\text{/preset}} + \overline{\text{/preclear}} \cdot s$$
$$z_r = \overline{\text{/preclear}} + \overline{\text{/preset}} \cdot r$$

# Ingressi di inizializzazione del latch SR (cont.)

- Devo **modificare** la sintesi del latch SR
- Conviene **mettergli davanti una rete combinatoria**, che ha come ingresso  $s$ ,  $r$ ,  $/preset$ ,  $/preclear$ , ed in uscita due variabili  $z_r$ ,  $z_s$

		/preset /preclear		z <sub>s</sub> z <sub>r</sub>	
		00	01	11	10
sr		00	01	00	01
00	--	10	00	01	
01	--	10	01	01	
11	--	10	11	01	
10	--	10	10	01	

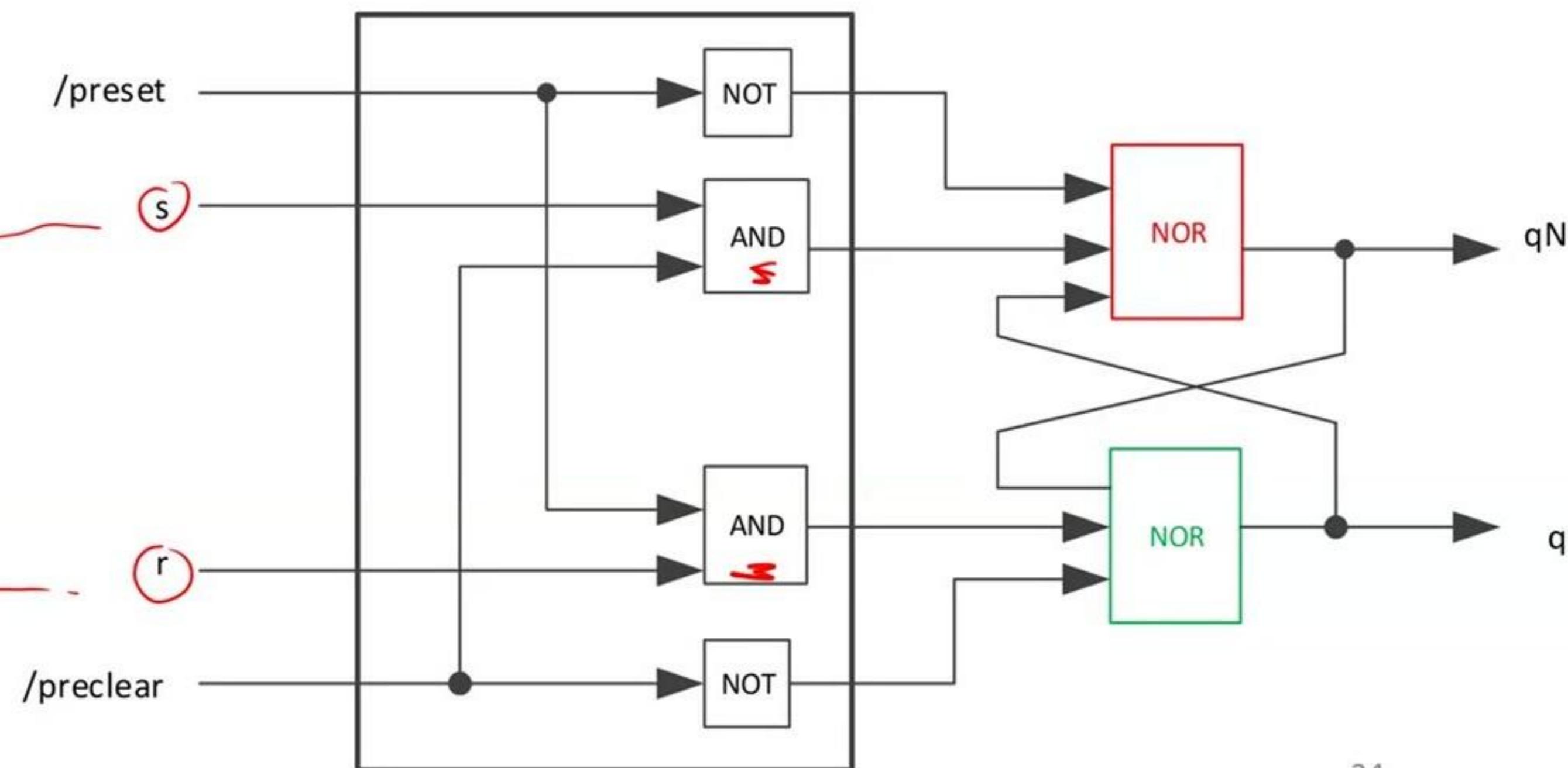
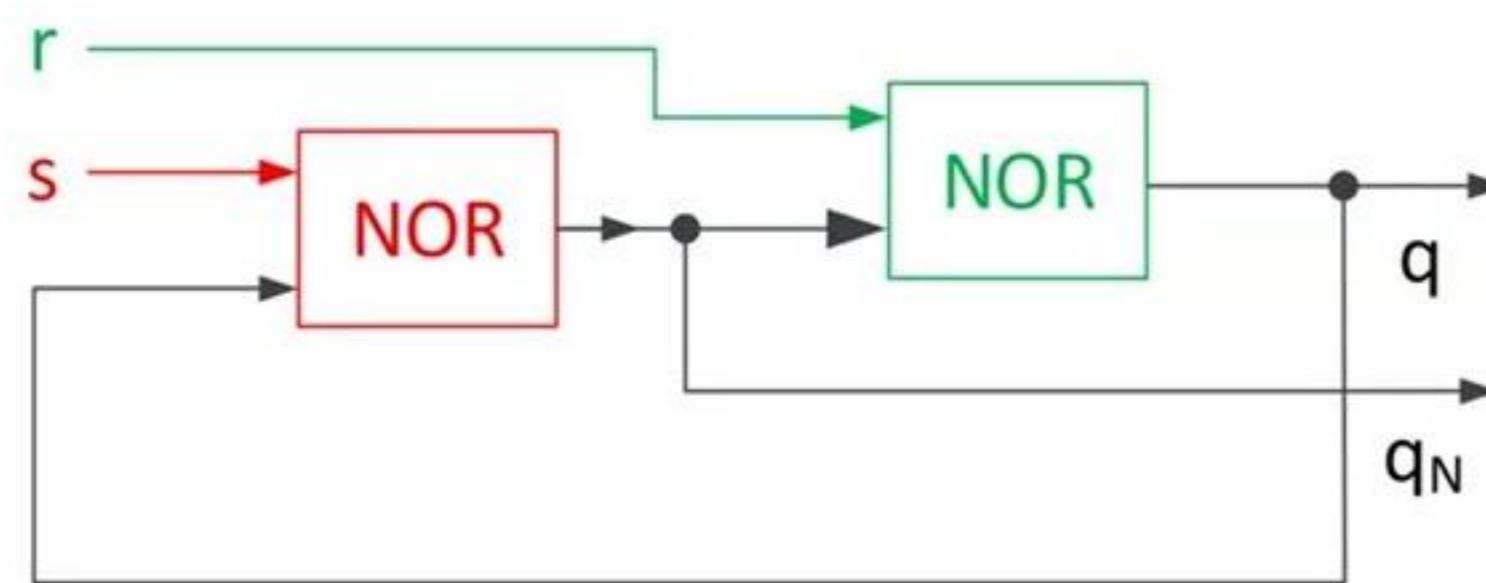
$$z_s = \overline{/preset} + (\overline{/preclear} \cdot s)$$
$$z_r = \overline{/preclear} + (\overline{/preset} \cdot r)$$

# Ingressi di inizializzazione del latch SR (cont.)

- Il Latch SR è realizzato a porte NOR
- $OR \rightarrow NOR$  in cascata si può semplificare
  - $OR \rightarrow (OR \rightarrow NOT) \Leftrightarrow (OR \rightarrow OR) \rightarrow NOT \Leftrightarrow OR \rightarrow NOT \Leftrightarrow NOR$

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

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



# Tabelle e grafi di flusso

- Le RSA si descrivono usando tabelle di flusso o grafi di flusso.
- Una **tabella di flusso** è una tabella che descrive come si evolvono lo **stato interno** e l'uscita al variare degli stati di ingresso.

dsndes

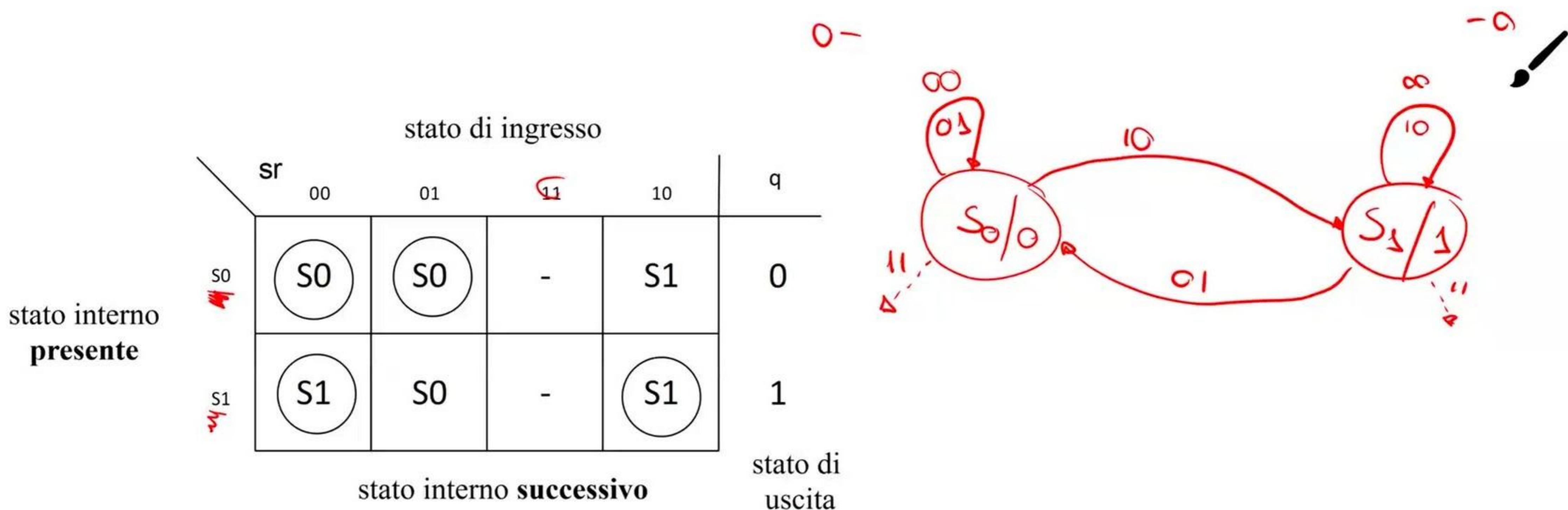
		stato di ingresso				q
		00	01	11	10	
sr	s0	$s_0$	$s_0$	-	$s_1$	0
	s1	$s_1$	$s_0$	-	$s_1$	1
stato interno successivo				stato di uscita		

stato interno presente



# Tabelle e grafi di flusso (cont.)

- Un grafo di flusso è un insieme di **nodi** che rappresentano **stati interni**, ed **archi** etichettati con gli stati di ingresso, per marcare le transizioni di stato.



# Tabelle e grafi di flusso (cont.)

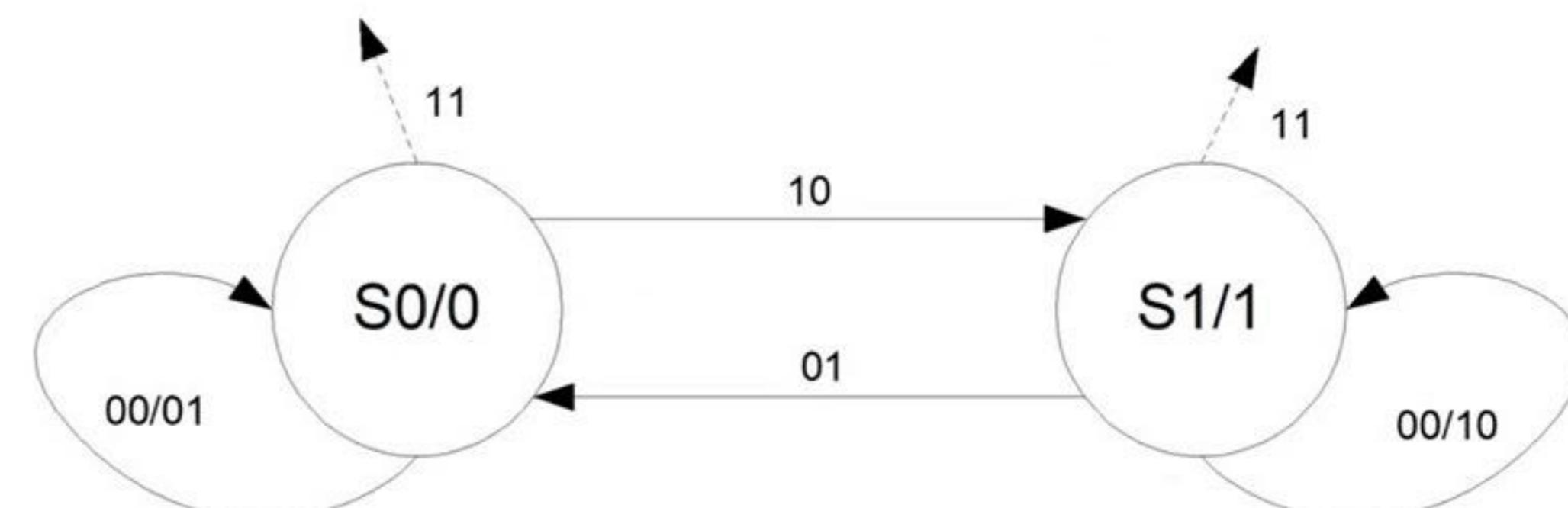
- Un grafo di flusso è un insieme di **nodi** che rappresentano **stati interni**, ed **archi** etichettati con gli stati di ingresso, per marcare le transizioni di stato.

stato di ingresso					q	0	1
sr	00	01	11	10			
s0	<b>S0</b>	<b>S0</b>	-	<b>S1</b>			
s1	<b>S1</b>	<b>S0</b>	-	<b>S1</b>			

stato interno **presente**

stato interno **successivo**

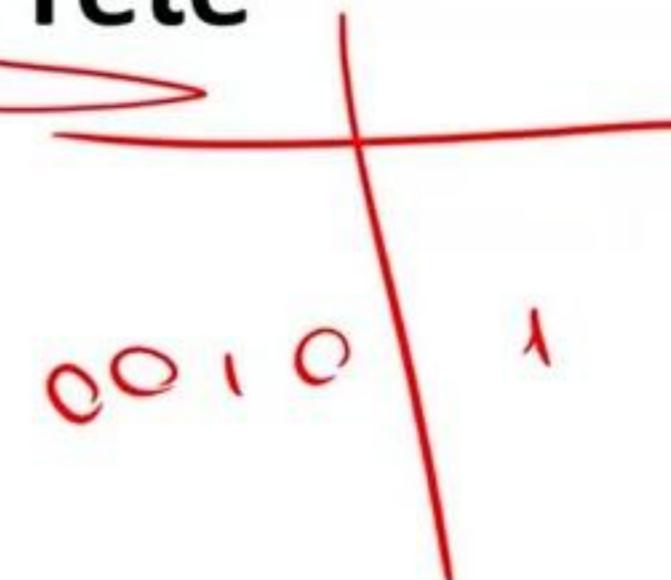
stato di uscita



# Dalla descrizione al diagramma di temporizzazione

Cosa fa

- Descrizione: consente di verificare il comportamento della rete
  - RC: tabella di verità
  - RSA: tabella/grafico di flusso
- Verifica della descrizione
  - RC: ispezione della tabella di verità (verifica **statica**)
  - RSA: simulazione nel tempo dell'evoluzione della rete (verifica **dinamica**)



# Dalla descrizione al diagramma di temporizzazione

Verikore

- **Diagramma di temporizzazione:** serve a capire se la rete si comporta come previsto

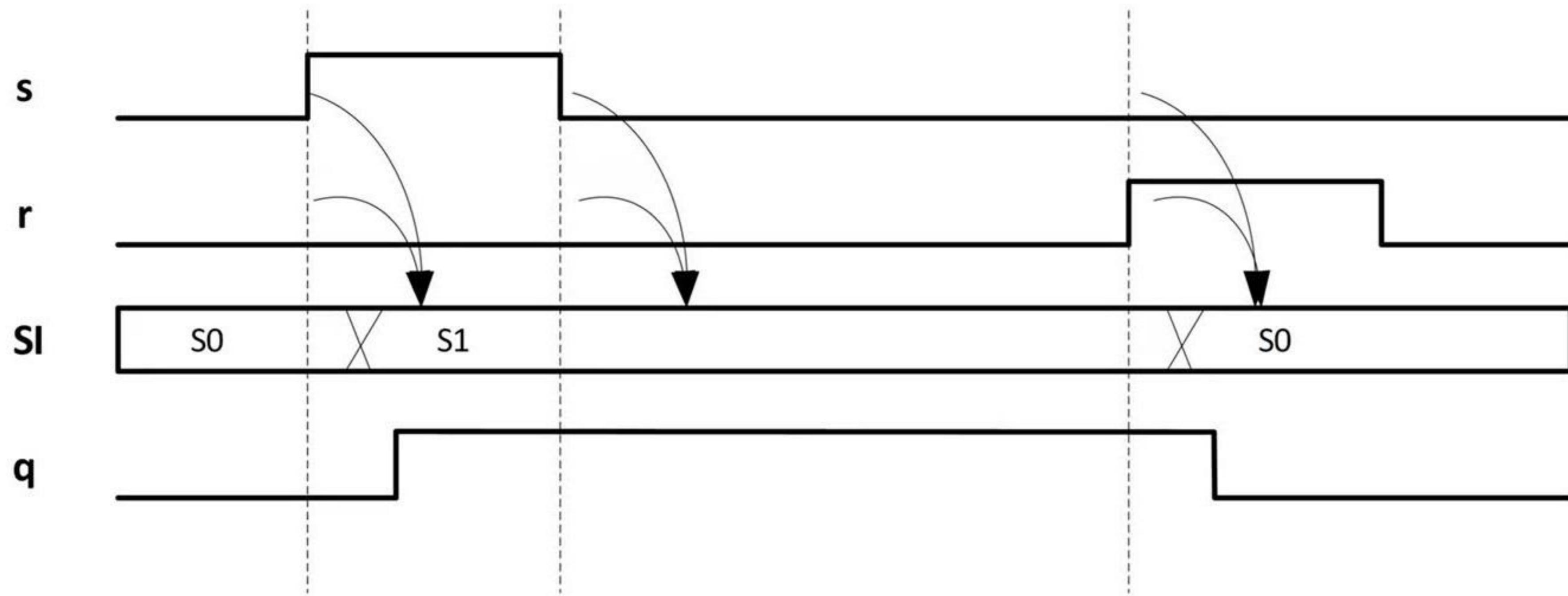
- Decido uno stato iniziale
- Attribuisco valori agli ingressi nel tempo
- Osservo come si evolve la rete

- **Equivalente hardware del testing di un programma**

- Necessario saperlo fare
- Necessario farlo

# Diagramma di temporizzazione

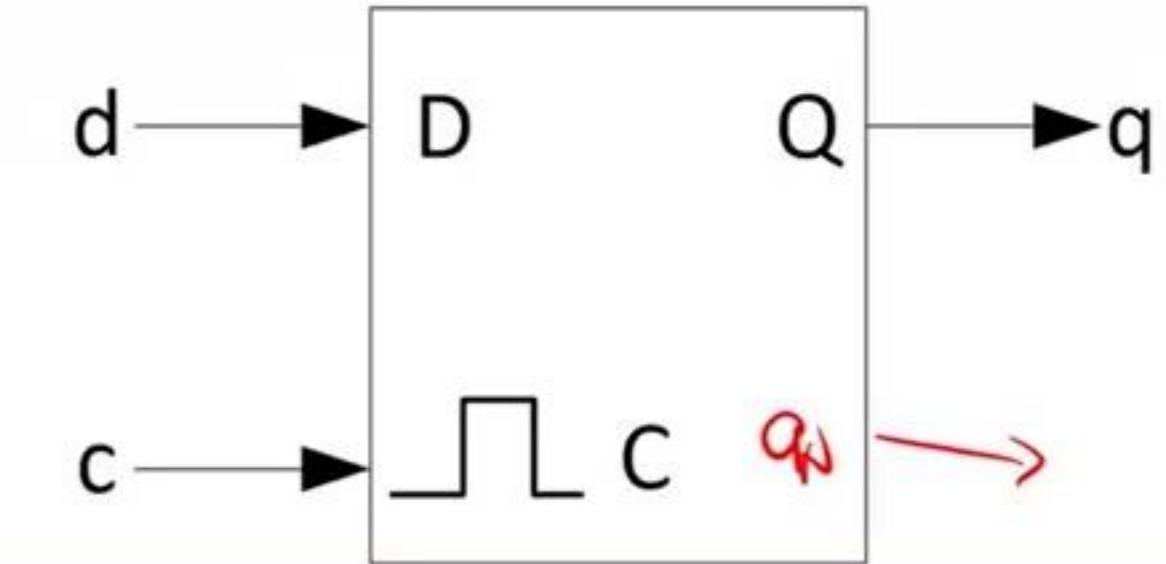
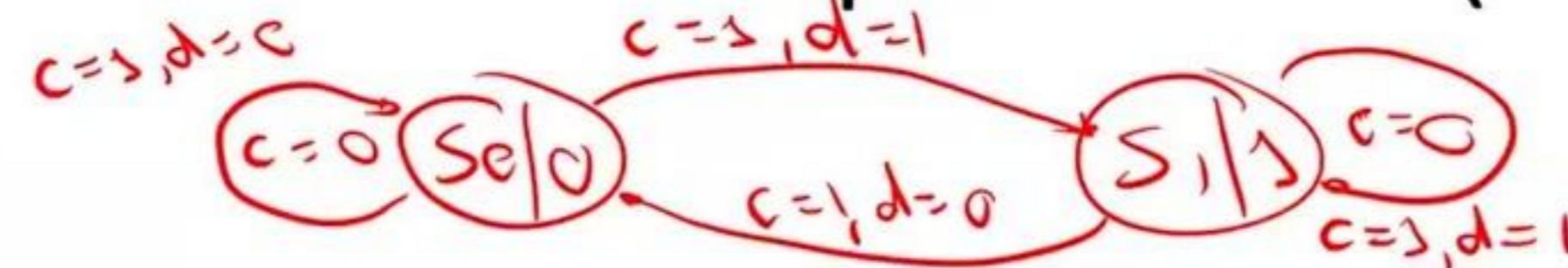
sr	00	01	11	10	q
s0	S0	S0	-	S1	0
s1	S1	S0	-	S1	1



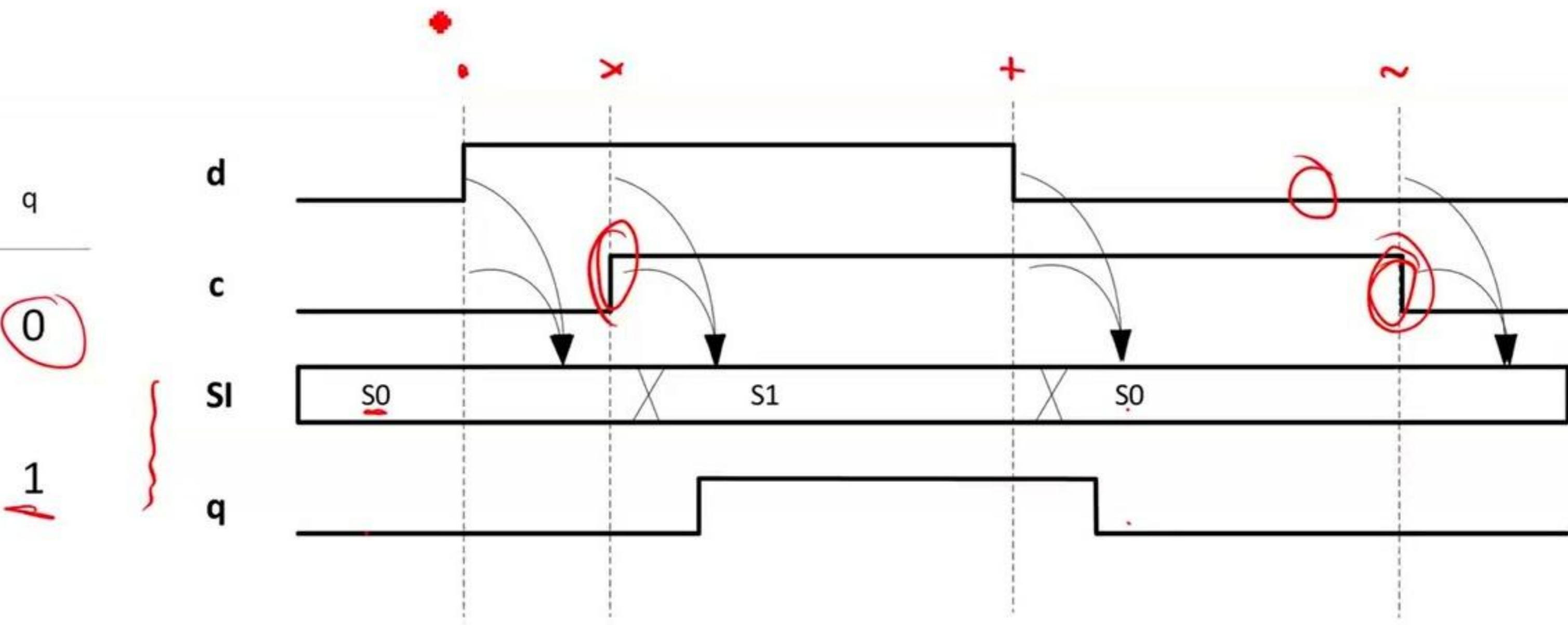
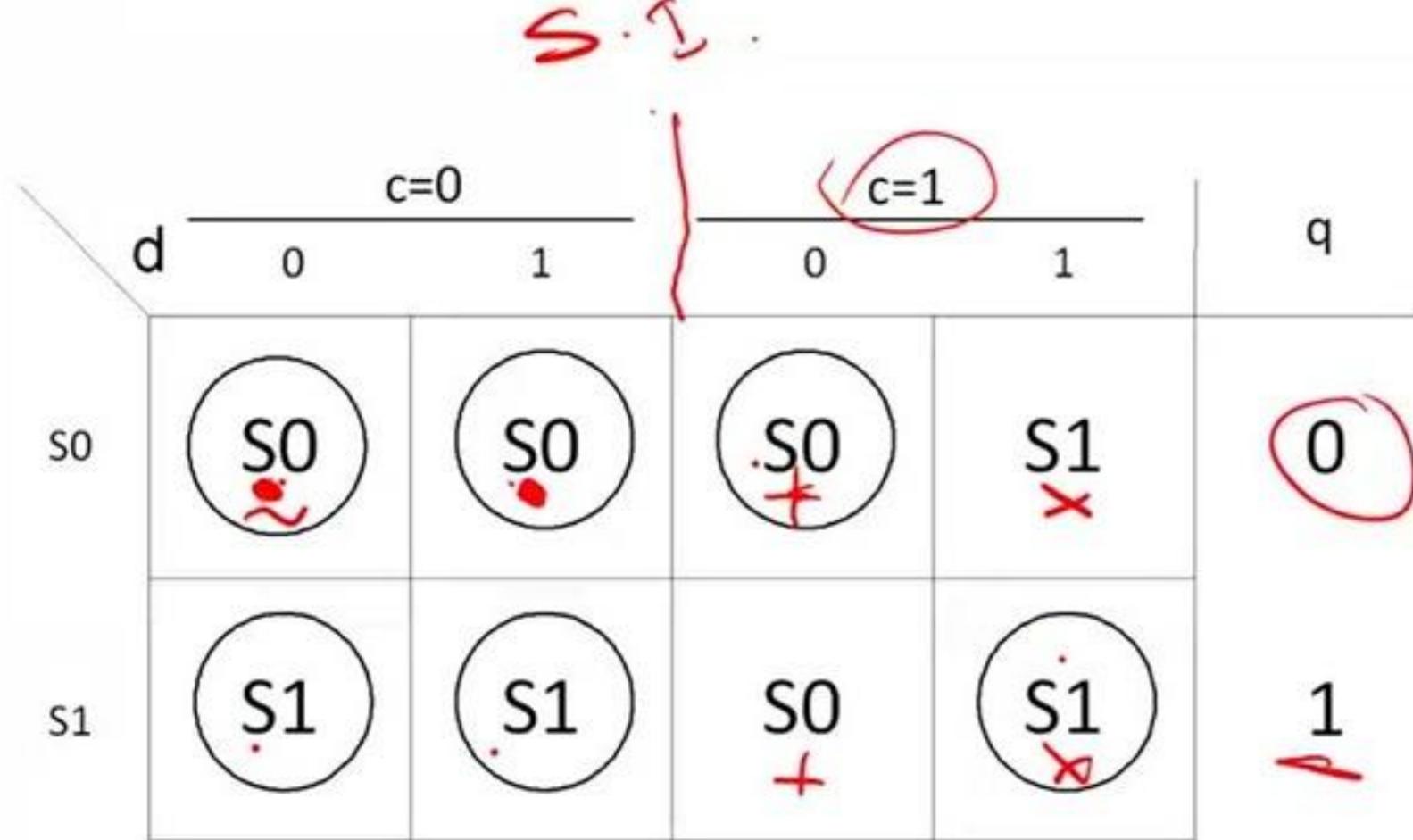
## Il D-latch trasparente

- RSA con due ingressi
    - d (data)
    - c (control)
  - Ed una uscita  $q$  (in realtà è sempre disponibile anche l'uscita  $q_N$ ).
  - La sua descrizione (a parole) è la seguente:
  - Il D-latch **memorizza l'ingresso  $d$**  (quindi, memorizza **un bit**) quando  **$c$  vale 1 (trasparenza)**
  - Quando  **$c$  vale 0**, invece, è **in conservazione**, cioè mantiene in uscita (**memorizza**) l'ultimo valore che  $d$  ha assunto quando  $c$  valeva 1
- $\rightarrow \text{cons. } c=0 \quad \cancel{\star} \quad q = GSI.$   
 $\rightarrow \text{trasp. } c=1 \quad \cancel{q=d}$

## Il D-latch trasparente (cont.)

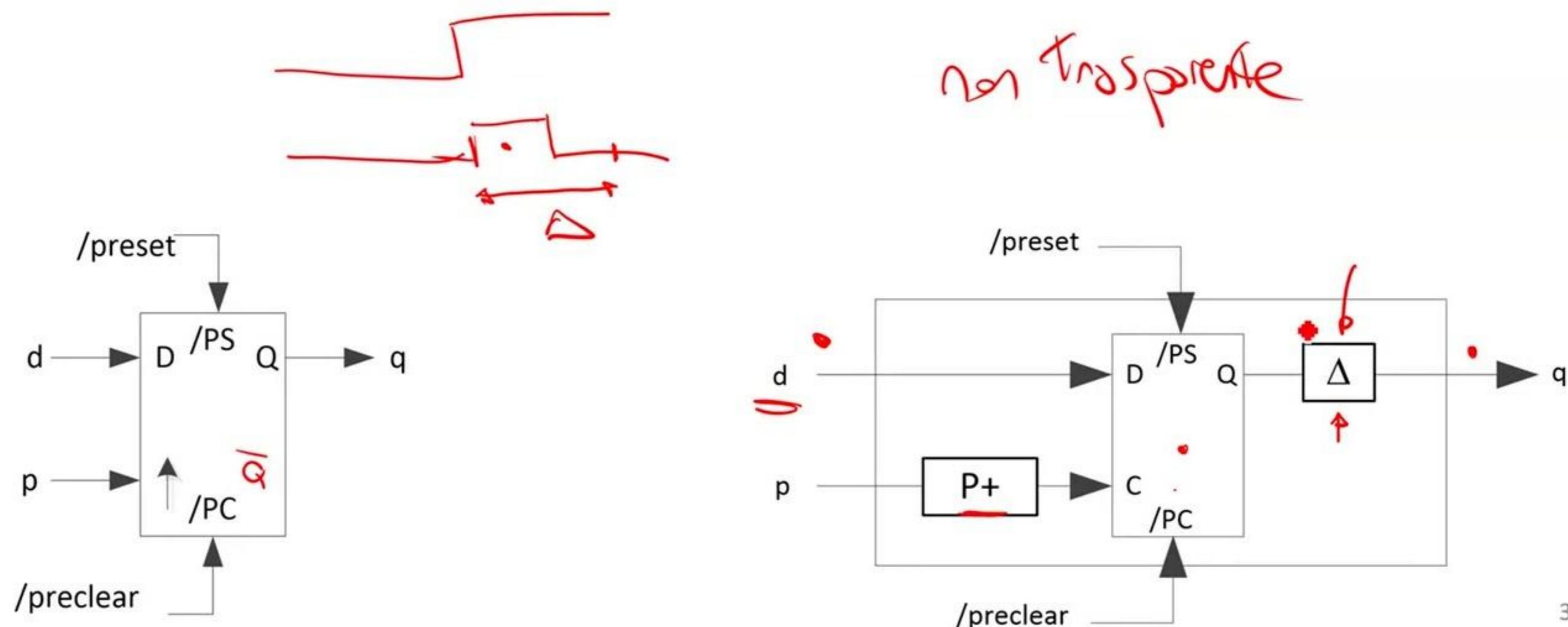


- È una rete che può trovarsi **in due stati**, uno nel quale ha memorizzato 0 ed uno nel quale ha memorizzato 1



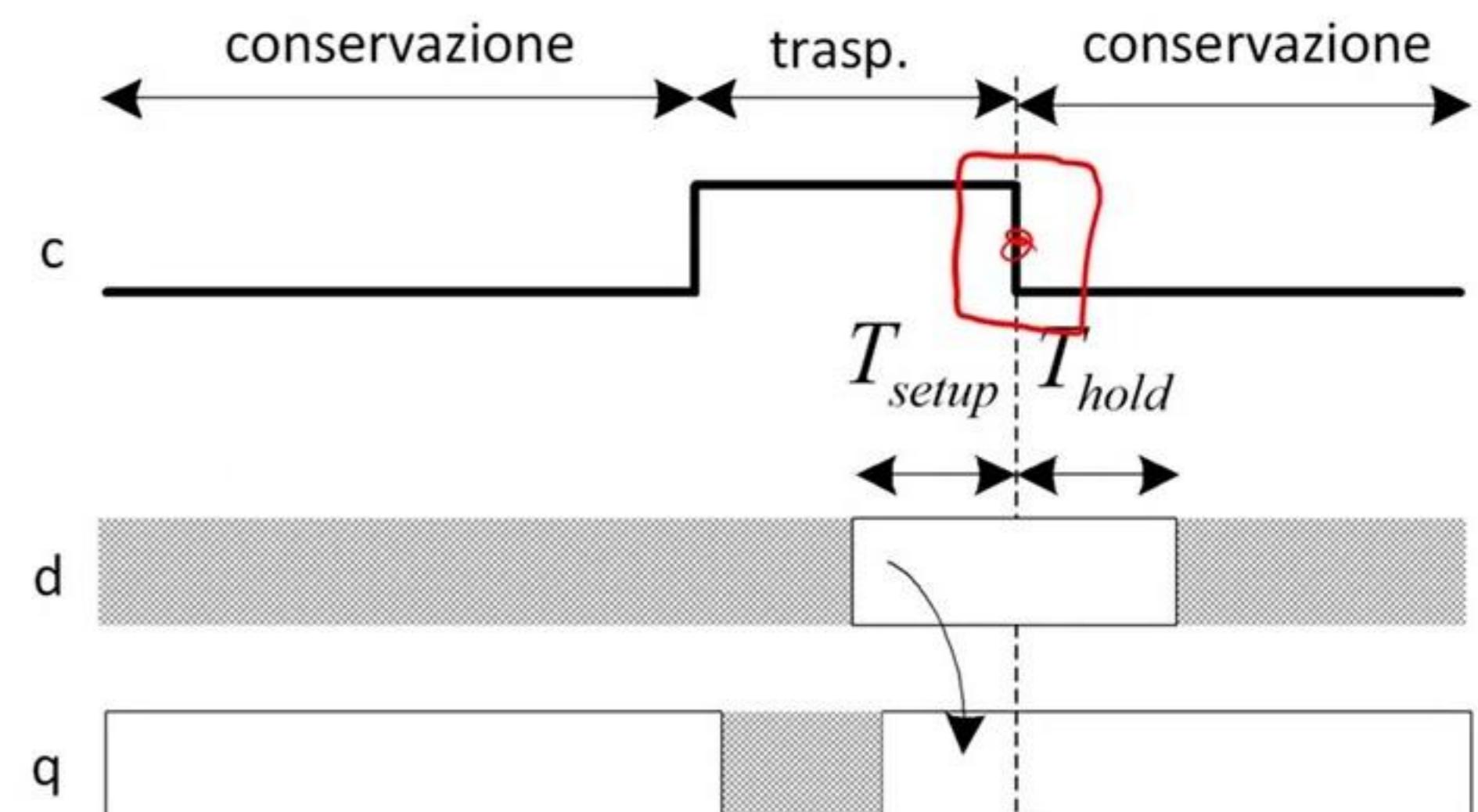
# Il D flip-flop

- Primo schema concettuale per il positive edge-triggered D flip-flop
- Ritardo  $\Delta$  maggiore dell'intervallo del  $P^+$ 
  - quando  $q$  cambia adeguandosi a  $d$ , la rete non è più in trasparenza, ma in conservazione.

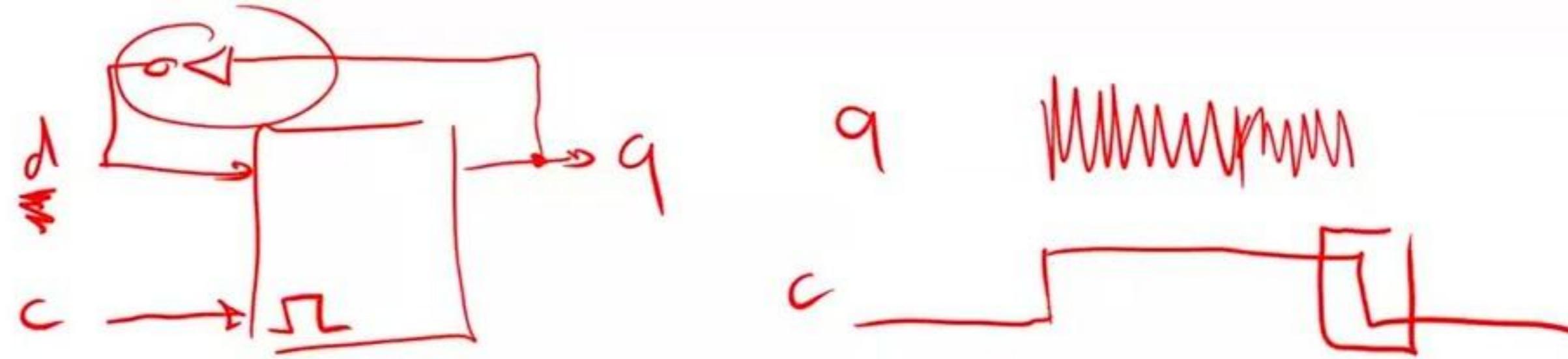


# Pilotaggio del D-latch

- d deve essere **costante** a cavallo della transizione di c da 1 a 0.
- I tempi per cui deve essere costante (prima e dopo) sono chiamati  $T_{\text{setup}}$  e  $T_{\text{hold}}$ , e sono **dati di progetto** della rete.
- Servono a garantire che la rete non veda transizioni multiple di ingresso, e che quindi si stabilizzi in modo prevedibile



# Trasparenza

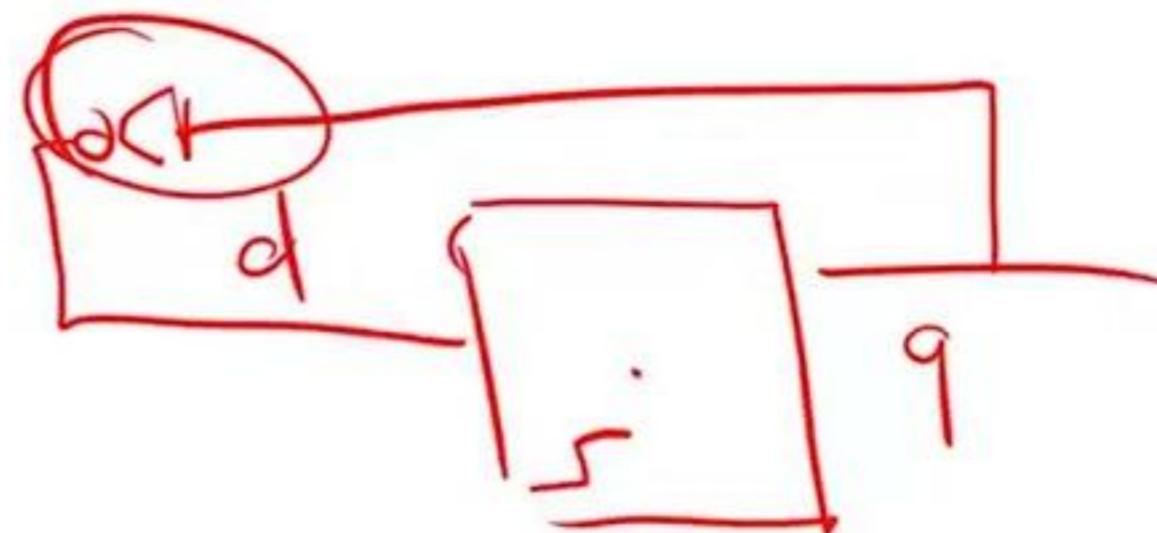


- Quando il D-latch è in **trasparenza**, l'ingresso è “direttamente connesso” all’uscita
  - in senso **logico**: dal punto di vista **fisico** ci sono comunque delle porte logiche in mezzo
- Pertanto, se  $q$  e  $d$  sono collegati in **retroazione negativa**,
  - quando  $c$  è ad 1 l’uscita **oscilla in maniera incontrollata**
  - quando  $c$  va a 0 si stabilizza ad un **valore casuale**

Il D-Latch è una rete **trasparente**, cioè  
**la sua uscita cambia mentre la rete è sensibile alle variazioni di ingresso.**

- non si può memorizzare **niente che sia funzione dell’uscita  $q$** , altrimenti l’evoluzione della rete non è prevedibile

## Trasparenza (cont.)



- [...] se  $q$  e  $d$  sono collegati in **retroazione negativa** [...]
- ???
- INC ~~%AX~~
- Il contenuto di AX è **l'uscita** di un certo numero di elementi di memoria
- Il nuovo valore che deve finire in AX è **l'ingresso**  $d$  di questi elementi
- La relazione tra il vecchio valore di AX ed il nuovo è combinatoria
  - può certamente succedere che singoli bit siano invertiti
- Reazionare le uscite sugli ingressi è una cosa che capita di dover fare **piuttosto spesso**, e non poterlo fare è una limitazione pesante



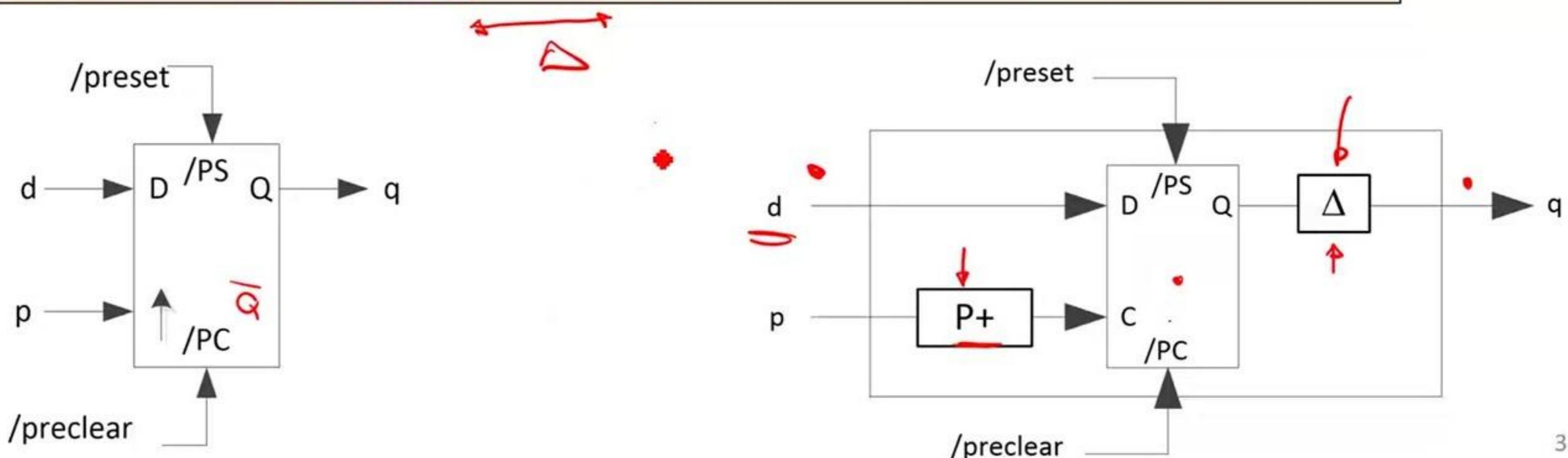
# Il D flip-flop

- Rete non trasparente.
  - trasparenti => latch
  - non trasparenti => **flip-flop**
- Positive edge-triggered D flip-flop, ed è una rete con due variabili di ingresso,  $d$  e  $p$ , che si comporta come segue:
  - “Quando  $p$  ha un fronte di salita, memorizza  $d$ , **attendi un po’** e adegua l’uscita”

# Il D flip-flop

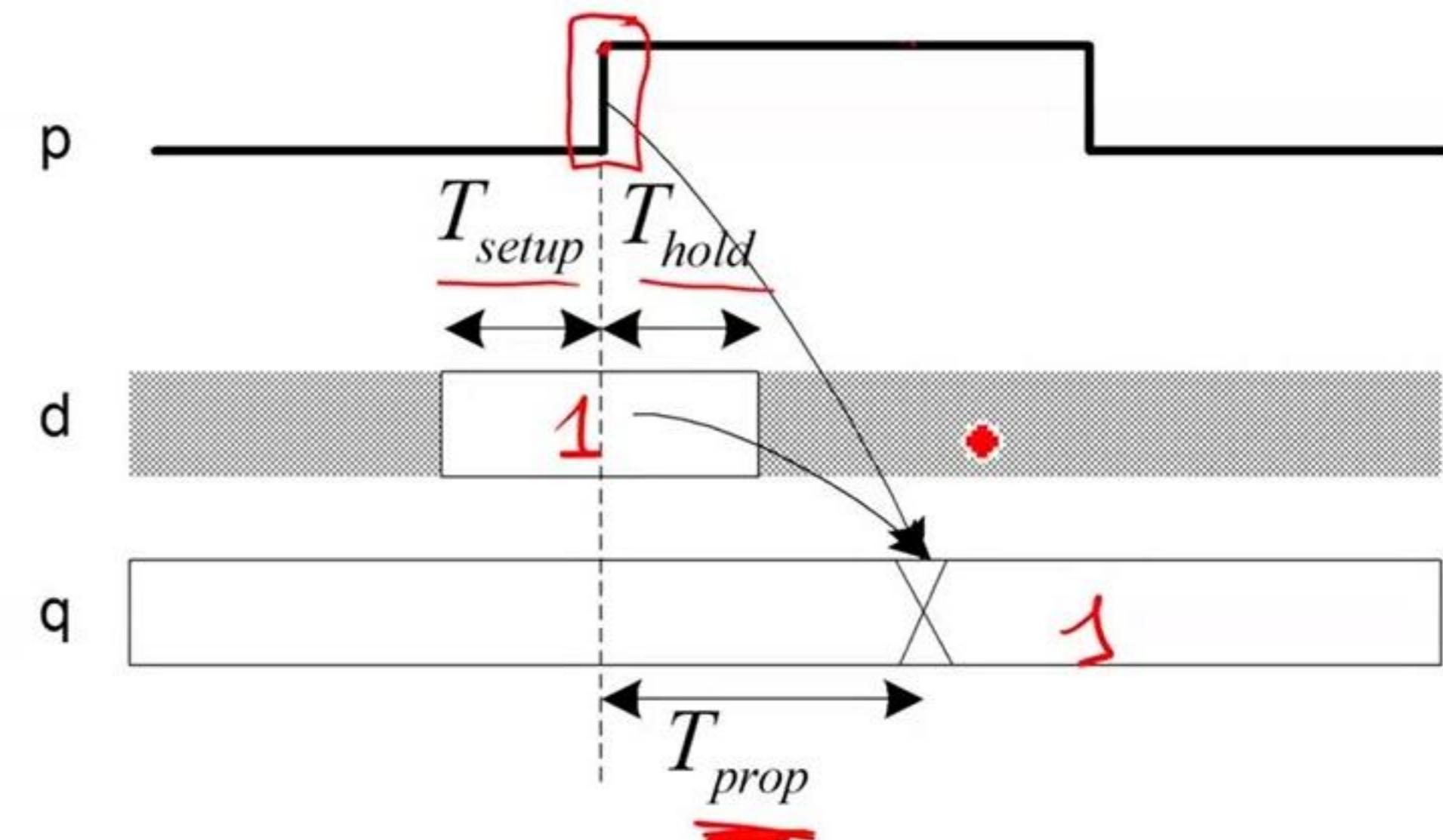
- Primo schema concettuale per il positive edge-triggered D flip-flop
- Ritardo  $\Delta$  maggiore dell'intervallo del  $P^+$ 
  - quando  $q$  cambia adeguandosi a  $d$ , la rete non è più in trasparenza, ma in conservazione.

L'uscita  $q$  viene adeguata al valore campionato di  $d$   
dopo che la rete ha smesso di essere sensibile al valore di  $d$ .



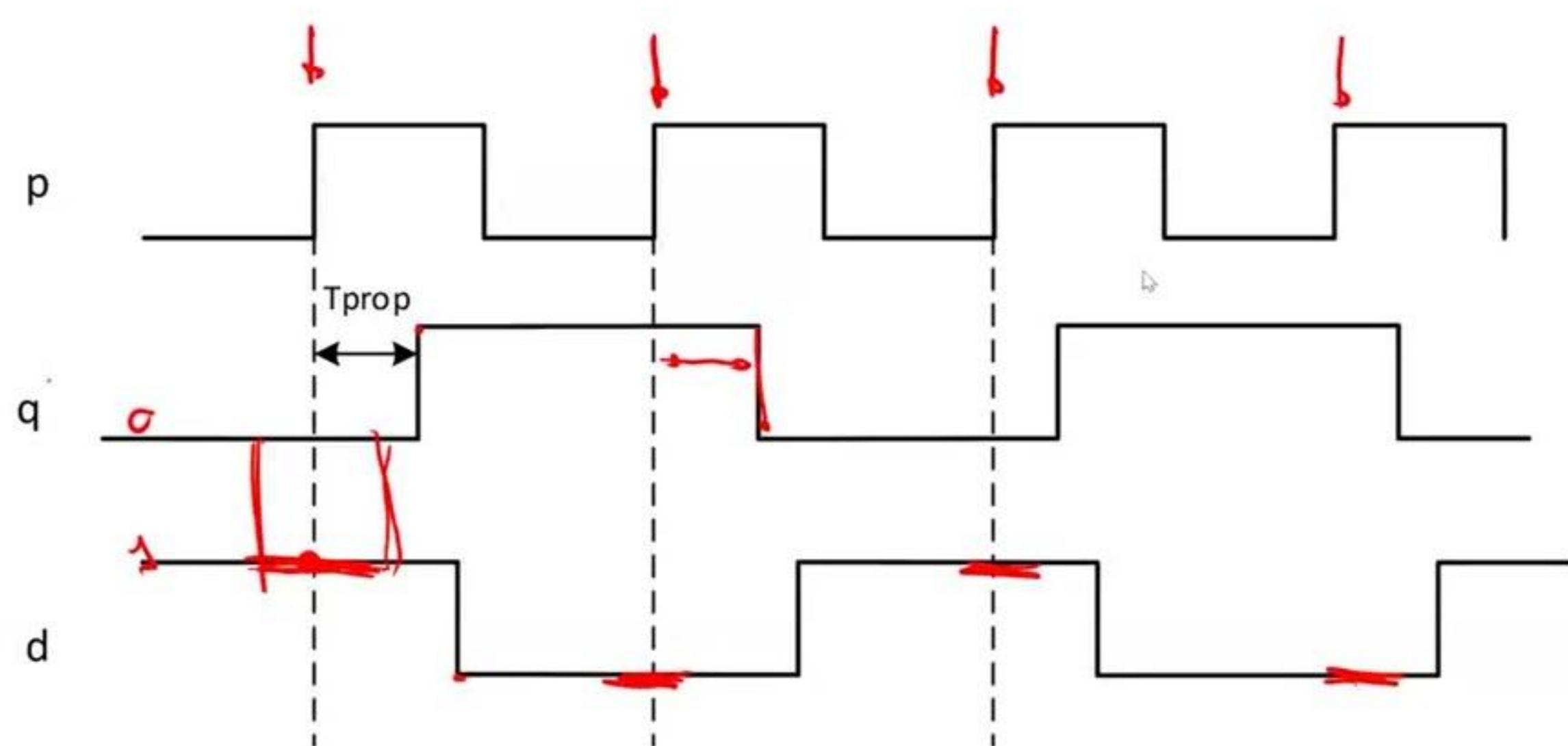
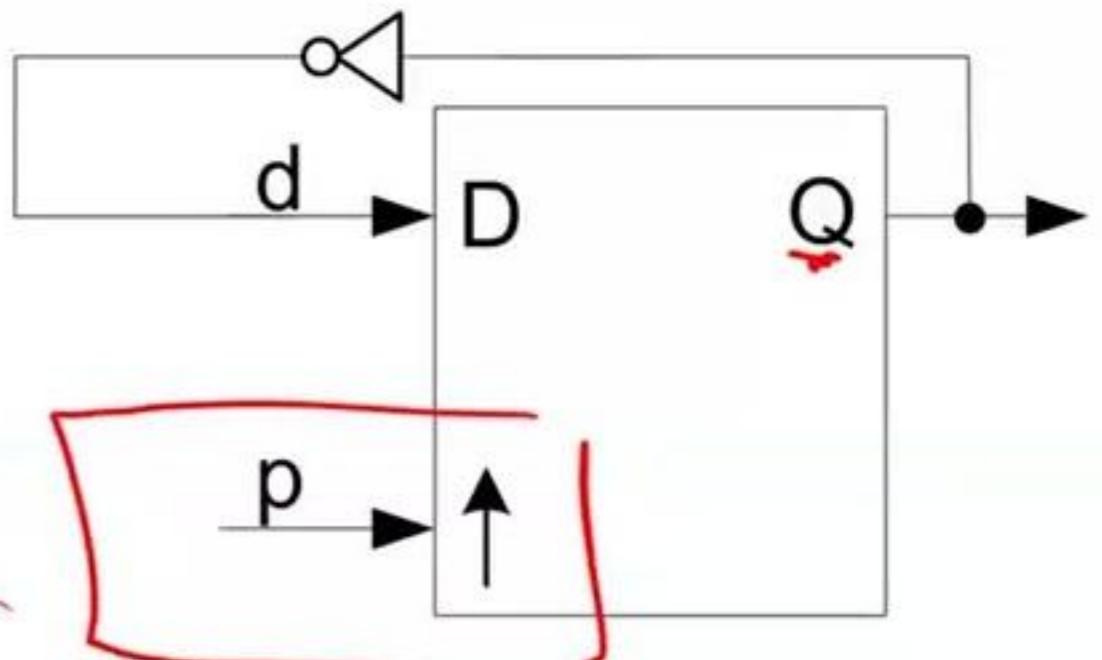
# Pilotaggio del D flip-flop

- A cavallo del fronte di salita di  $p$  l'ingresso  **$d$  deve rimanere costante.**
  - I tempi per cui deve rimanere costante si chiamano  $T_{setup}$ ,  $T_{hold}$
  - Il loro valore dipende da come è progettata la rete.
- Il ritardo con cui si adegua l'uscita si chiama  $T_{prop}$ , ed è
$$T_{prop} > T_{hold}$$
- Quest'ultima diseguaglianza garantisce che questa rete sia **non trasparente**.



## Pilotaggio del D flip-flop (cont.)

- L'uscita di un D-FF non oscilla mai
  - a differenza di quella del D-latch
- viene adeguata in modo secco ad un istante ben preciso
- non è mai "direttamente connessa" (in senso logico) con l'ingresso  $d$
- Si possono montare i D-FF in qualunque modo, **senza che la rete cessi di evolversi in modo prevedibile.**



# Sintesi Master-Slave di un D flip-flop

- Montaggio fatto da due D-latch in cascata.

- $p = 0$ : master campiona, slave conserva

- quindi non ascolta il proprio ingresso  $d$

- $p = 1$ : master conserva, slave campiona

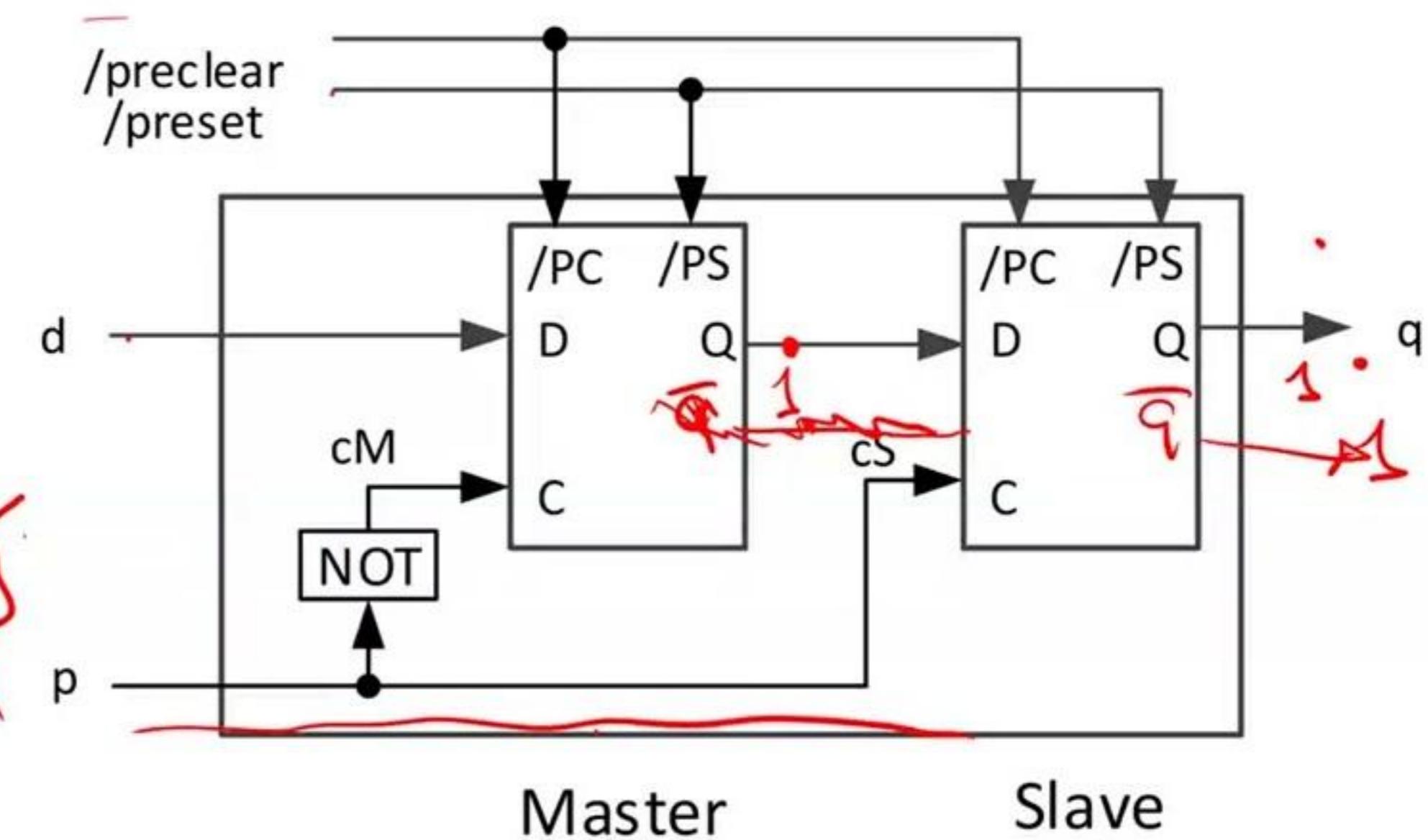
- quindi insegue l'uscita del master



- Possibili problemi nel funzionamento transitorio

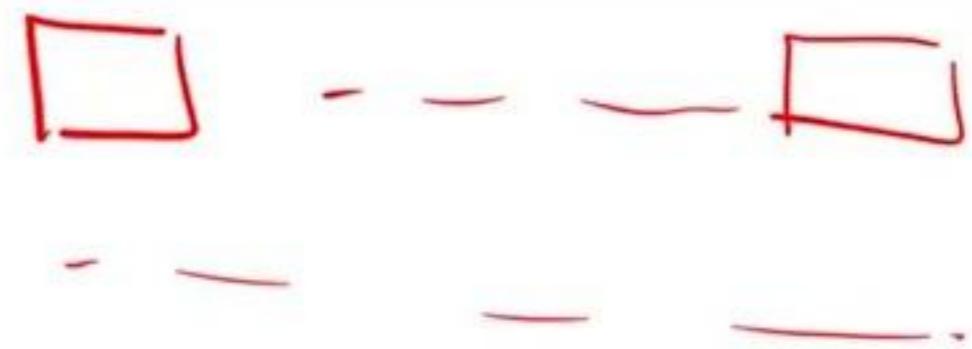
- Master e lo slave possono essere contemporaneamente in trasparenza, anche se per poco tempo

- Per evitarlo si agisce per via elettronica: i due ingressi  $c$  commutano con valori di tensione diversi



# Memorie RAM statiche

$2^{32}$



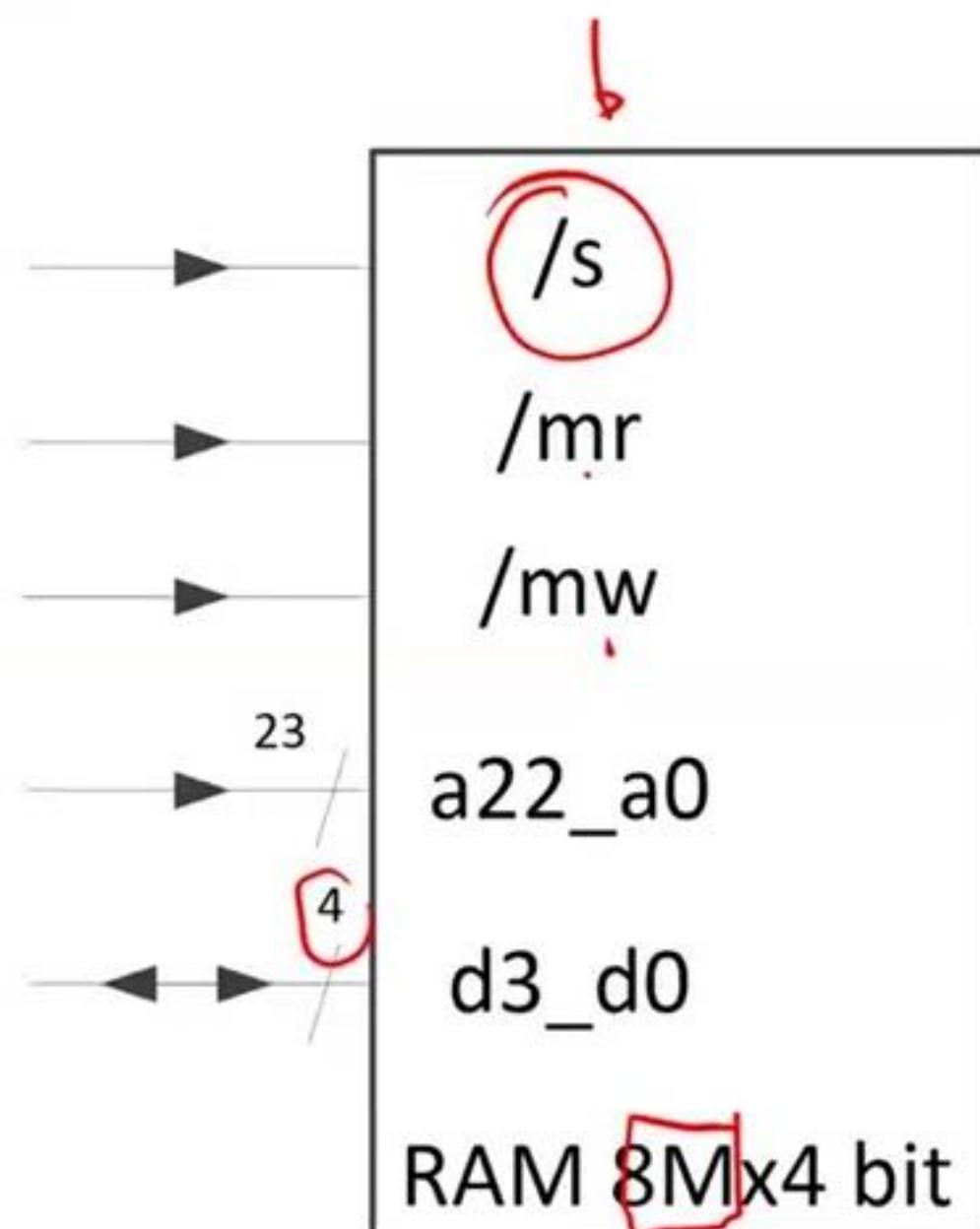
- S-RAM
- Esistono anche quelle **dinamiche** (D-RAM), ma sono fatte in modo del tutto differente
- Sono batterie di D-Latch montati a **matrice**.
- Una riga di D-Latch costituisce una **locazione di memoria**, che può essere **letta o scritta** con un'operazione di lettura o scrittura.
- Le operazioni di lettura e scrittura **non possono essere simultanee**

# Memorie RAM statiche (cont.)

$$\begin{array}{ll} \parallel & \\ & 2^{10} \quad 2^{20} \quad 2^{30} \quad 2^{40} \\ K & M \quad G \quad T \end{array}$$

$$8K \quad 2^10 \cdot 8 = 2^{13}$$

- Fili di indirizzo (ingressi)
  - In numero sufficiente ad indirizzare tutte le celle della memoria.
  - Nell'esempio,  $2^{23}$  celle di 4 bit => 23 fili di indirizzo
- Fili di dati (ingresso/uscita)
  - andranno forchettati con porte tri-state, come visto a suo tempo.
- Due segnali attivi bassi di memory read e memory write.
  - Comandi di lettura e scrittura della cella il cui indirizzo è trasportato sui fili  $a_{22\_a_0}$
  - Mai attivi contemporaneamente.
- Un segnale (attivo basso) di select
  - viene attivato quando la memoria è selezionata.
  - /s = 1, la memoria è insensibile a tutti gli ingressi.
  - /s = 0, la memoria reagisce agli ingressi (simile a enabler in un decoder)

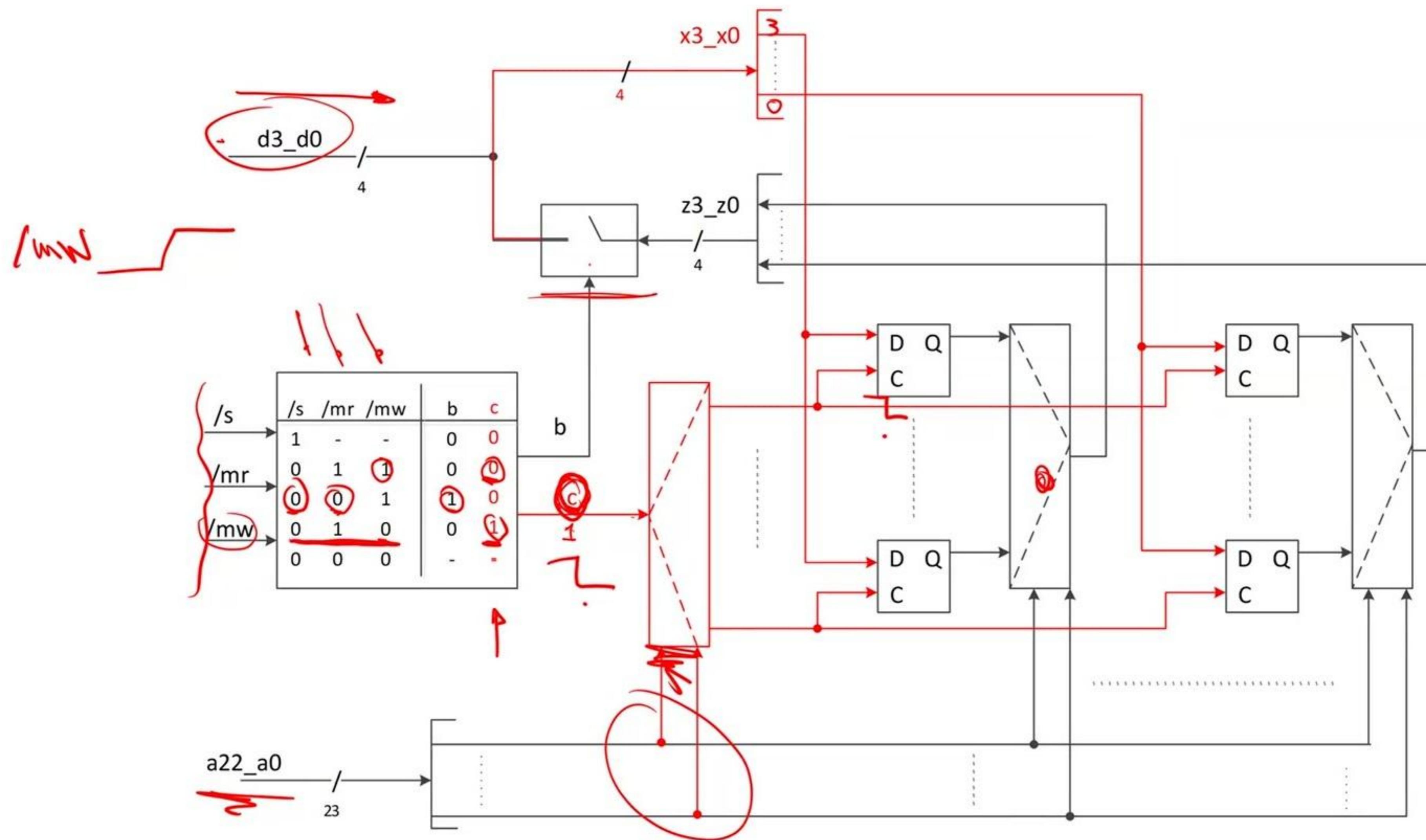


# Memorie RAM statiche (cont.)

- Il comportamento della memoria è deciso da **/s,/mw,/mr.**

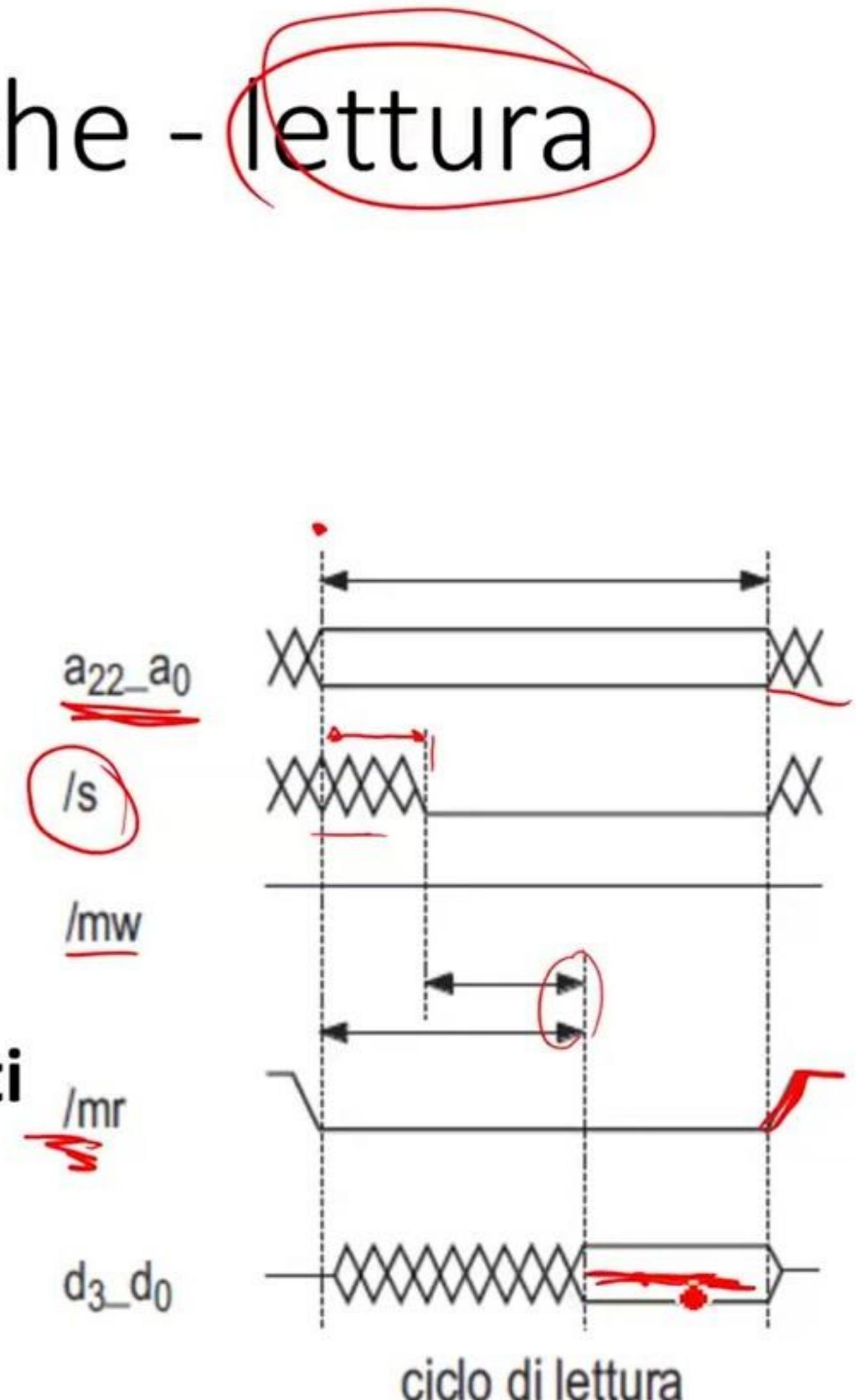
<b>/s</b>	<b>/mr</b>	<b>/mw</b>	<b>Azione</b>
→ 1	-	-	Nessuna azione (memoria non selezionata)
→ 0	1	1	Nessuna azione (memoria selezionata, nessun ciclo in corso)
0	0	1.	Ciclo di lettura in corso
0	1	0	Ciclo di scrittura in corso
0	0	0	Non definito

# Struttura della RAM statica



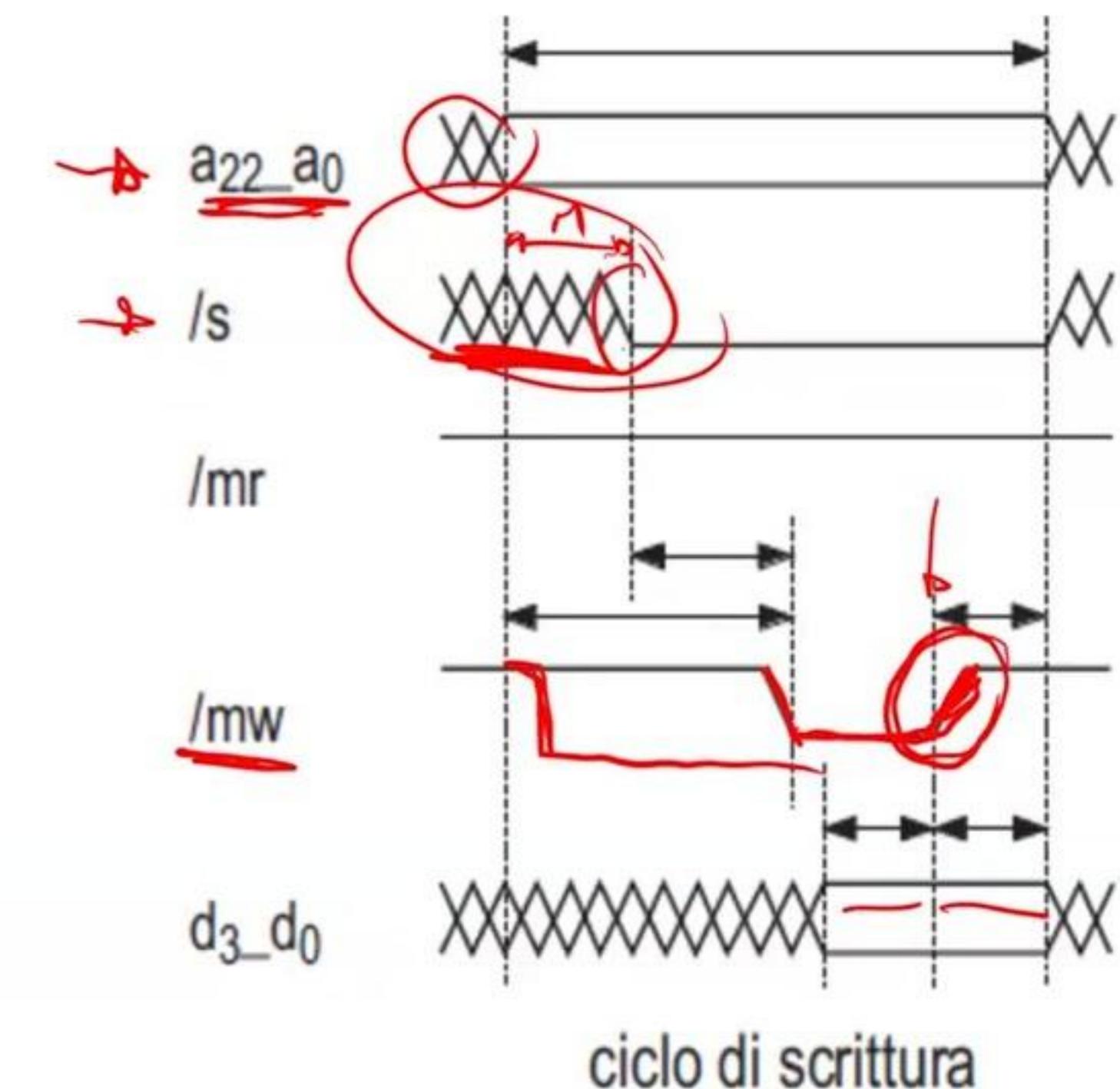
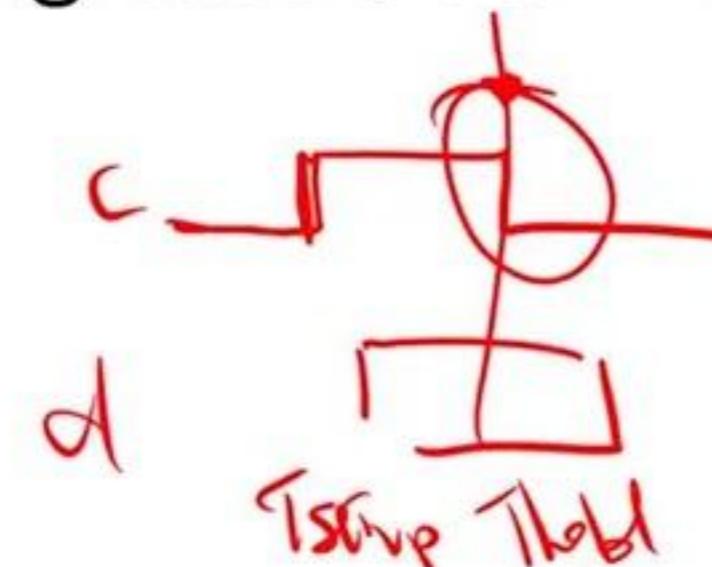
# Temporizzazione delle RAM statiche - lettura

- Gli indirizzi si stabilizzano ed arriva il comando di **/mr**.
- Il comando di **/s** arriva con un po' di ritardo, e **balla** nel frattempo, (funzione combinatoria di altri bit di indirizzo)
- Quando **sia /s che /mr sono a 0**,
  - le tri-state vanno in conduzione
- I multiplexer sulle uscite vanno a regime dopo gli indirizzi
  - Da lì in poi i dati sono buoni, e chi li ha richiesti li può prelevare
- Quando **/mr torna ad 1** (dopo che **chi voleva leggere i dati li ha prelevati**), i dati tornano in alta impedenza.
- A quel punto gli indirizzi e **/s** possono tornare a ballare



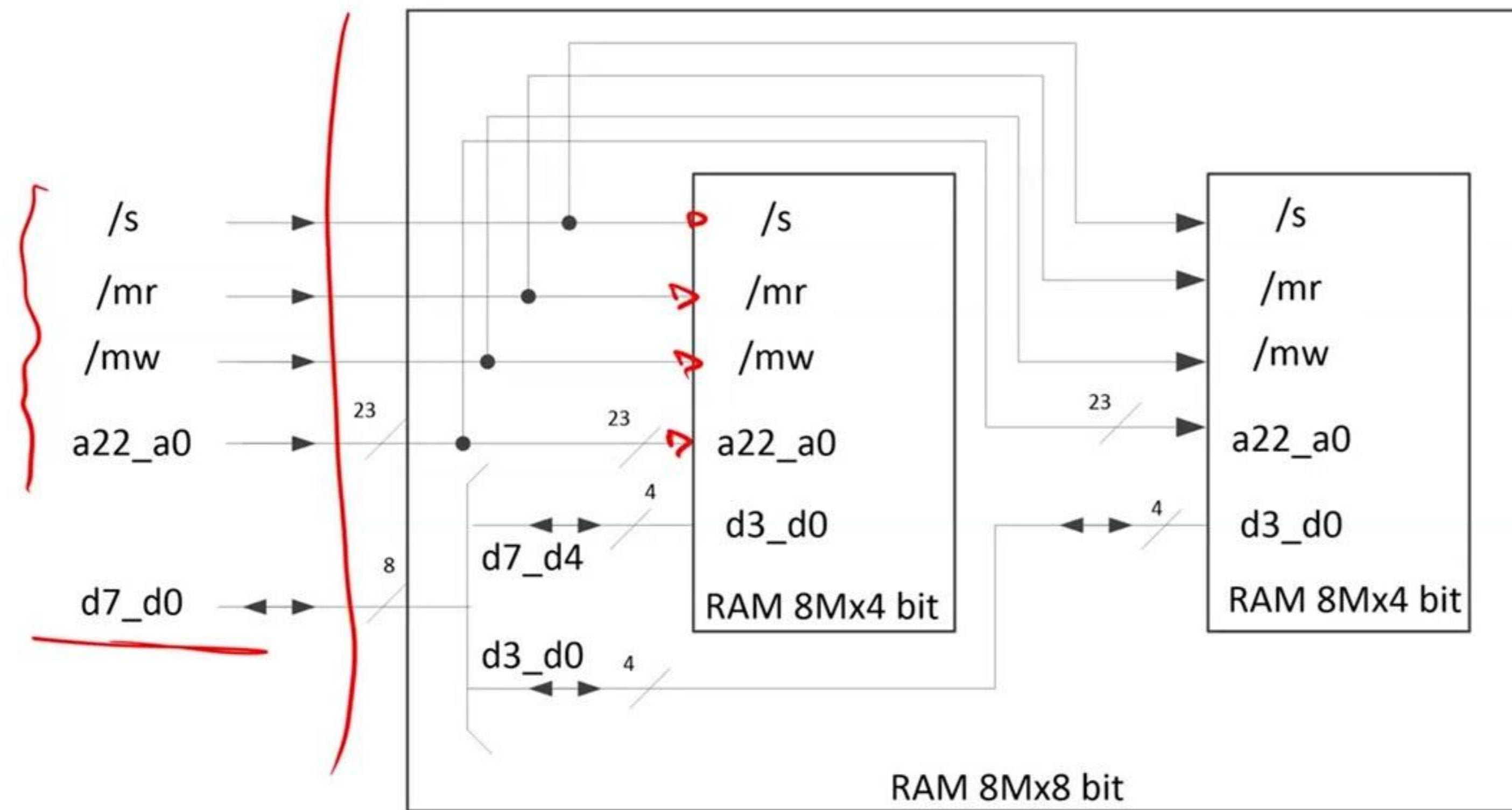
# Temporizzazione delle RAM statiche - scrittura

- La **scrittura è distruttiva** (i D-latch sono in trasparenza)
- Devo **attendere che /s e gli indirizzi siano stabili** prima di portare giù **/mw**
- I dati devono **essere corretti a cavallo del fronte di salita di /mw**
  - Corrisponde (con un minimo di ritardo dovuto alla rete combinatoria ed al demultiplexer), al fronte di discesa dell'ingresso *c* dei D-latch



# Montaggio «in parallelo»: raddoppio capacità celle

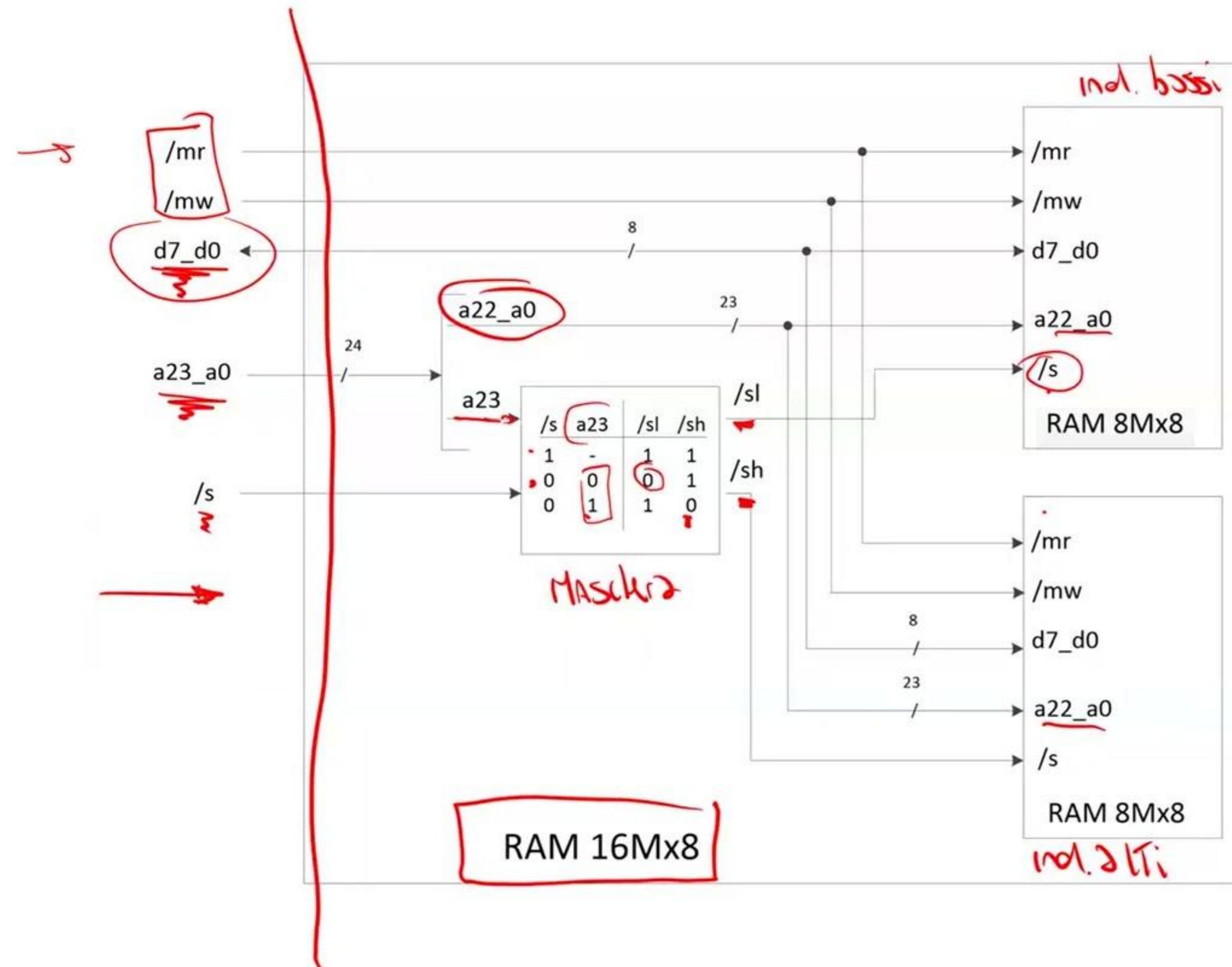
- Ottenerne una memoria **8Mx8** usando banchi **8Mx4**
- Basta connettere in parallelo tutti quanti i fili ed affastellare i dati.



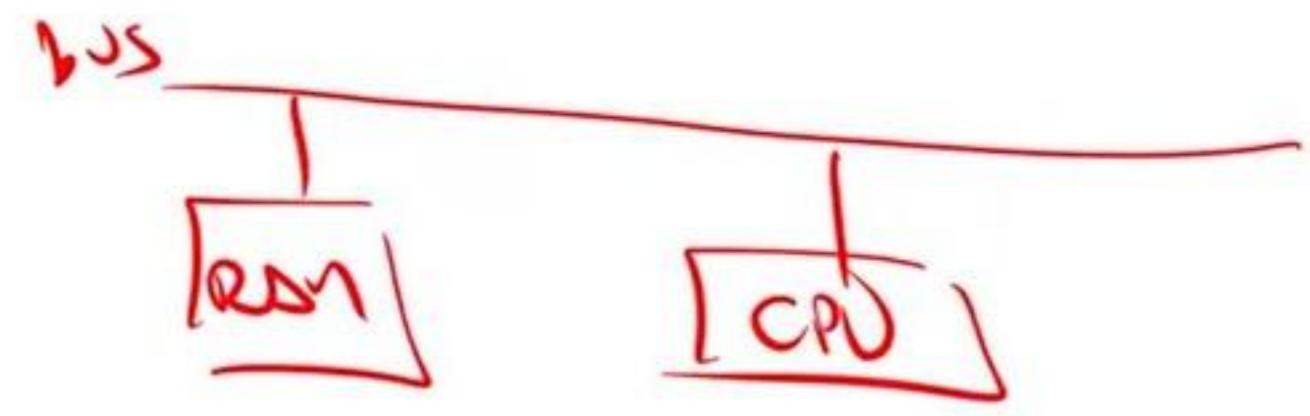
# Montaggio «in serie»: raddoppio numero locazioni

- Ottenerne una memoria  $16M \times 8$  usando banchi  $8M \times 8$
- Per indirizzare  $16M$  ci vogliono 24 fili di indirizzo, uno in più. Si dividono le locazioni in questo modo:
  - parte “alta” ( $a_{23} = 1$ ) in un blocco
  - parte “bassa” ( $\underline{\underline{a_{23}}}$ ) nell’altro
- Si genera il segnale di select per i due blocchi usando **il valore di  $a_{23}$**

# Montaggio «in serie»: raddoppio numero locazioni



# Collegamento al bus e maschere

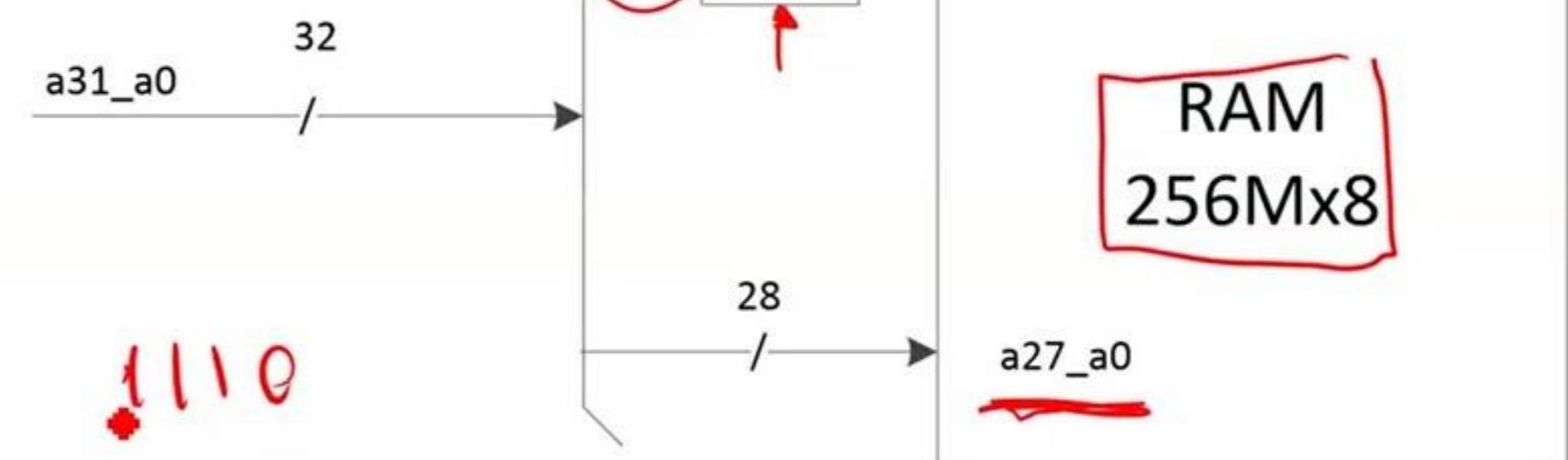


- I fili di indirizzo della memoria provengono da un **bus indirizzi**
- il processore ne imposta il valore

2.2<sup>30</sup>      not ox - - 1<sup>10x</sup>  
4Gb

## Esempio

- Bus indirizzi a **32 bit**
- Modulo di RAM **256Mx8** bit (28 fili di indirizzo)
- Montare a partire dall'indirizzo **0xE0000000**
- Dovrà rispondere agli indirizzi nell'intervallo **0xE0000000-0xFFFFFFFF**



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

# Collegamento al bus e maschere (cont.)

- Modulo di RAM 256Mx8 bit (28 fili di indirizzo, a partire dall'indirizzo 0xE0000000)

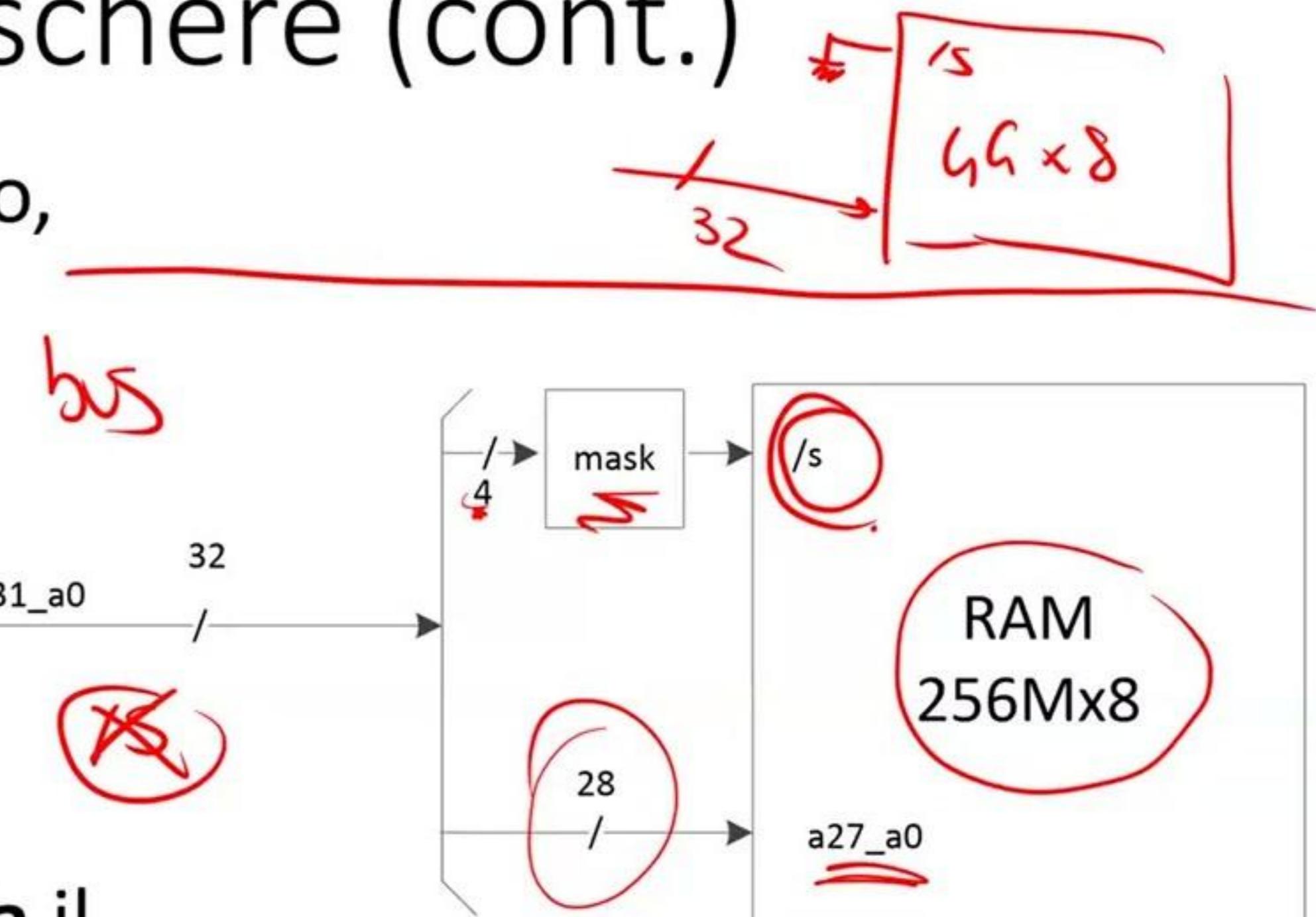
- Dovrà rispondere agli indirizzi nell'intervallo **0xE0000000-0xFFFFFFFF**

- La maschera deve riconoscere  $0xE = B1110$
- $/s = \overline{a_{31}} + \overline{a_{30}} + \overline{a_{29}} + a_{28}$

- Chi progetta la maschera? Colui che **assembلا** il sistema

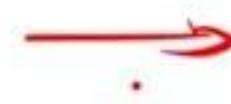
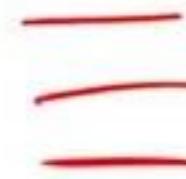
- Progettare la maschera  $\Leftrightarrow$  decidere a quale intervallo di indirizzi risponde un modulo di RAM
- Il filo /s si stabilizza un po' dopo gli indirizzi

- Per via della maschera



$a_{31}$	$a_{30}$	$a_{29}$	$a_{28}$	$/s$
1	1	1	0	0
others				1

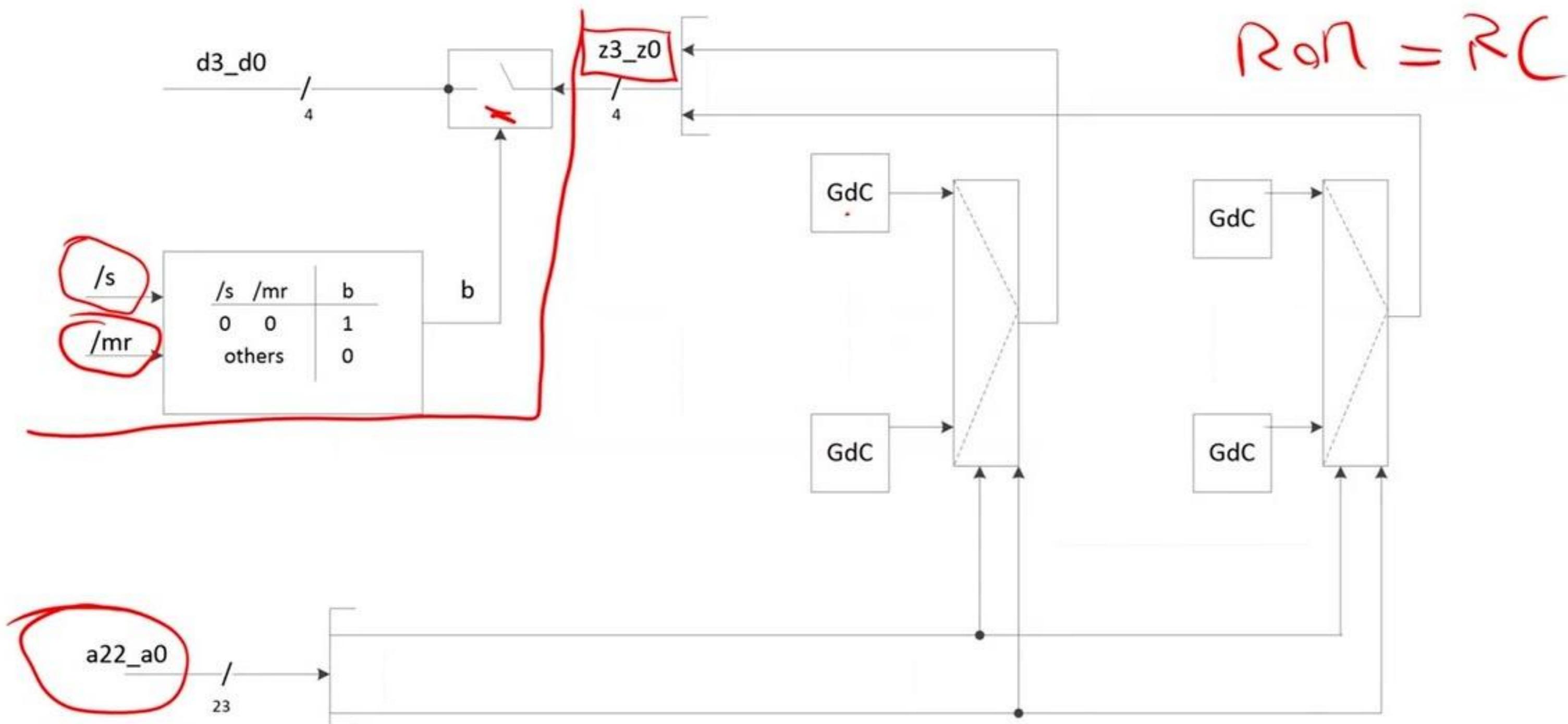
# Memorie read-only (ROM)



- Sono circuiti combinatori.
  - Infatti, ciascuna locazione contiene dei valori **costanti**, inseriti in modo **indelebile e dipendente dalla tecnologia**.
- Sono montate insieme alle memorie RAM nello spazio di memoria
  - costituiscono la parte non volatile dello spazio di memoria
  - mantengono l'informazione in assenza di tensione
- La loro struttura si ottiene per semplificazione delle memorie RAM
  - togliere la parte necessaria alla scrittura

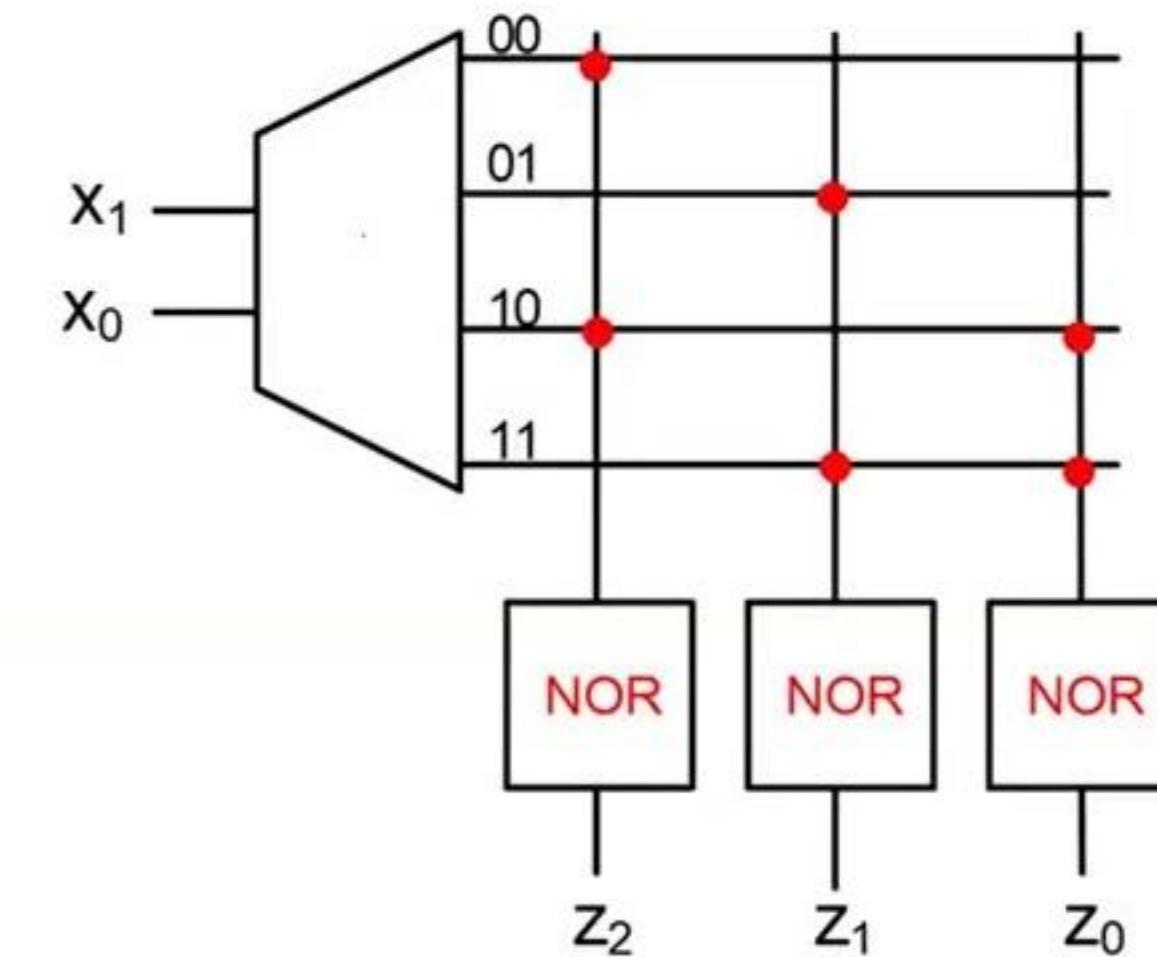
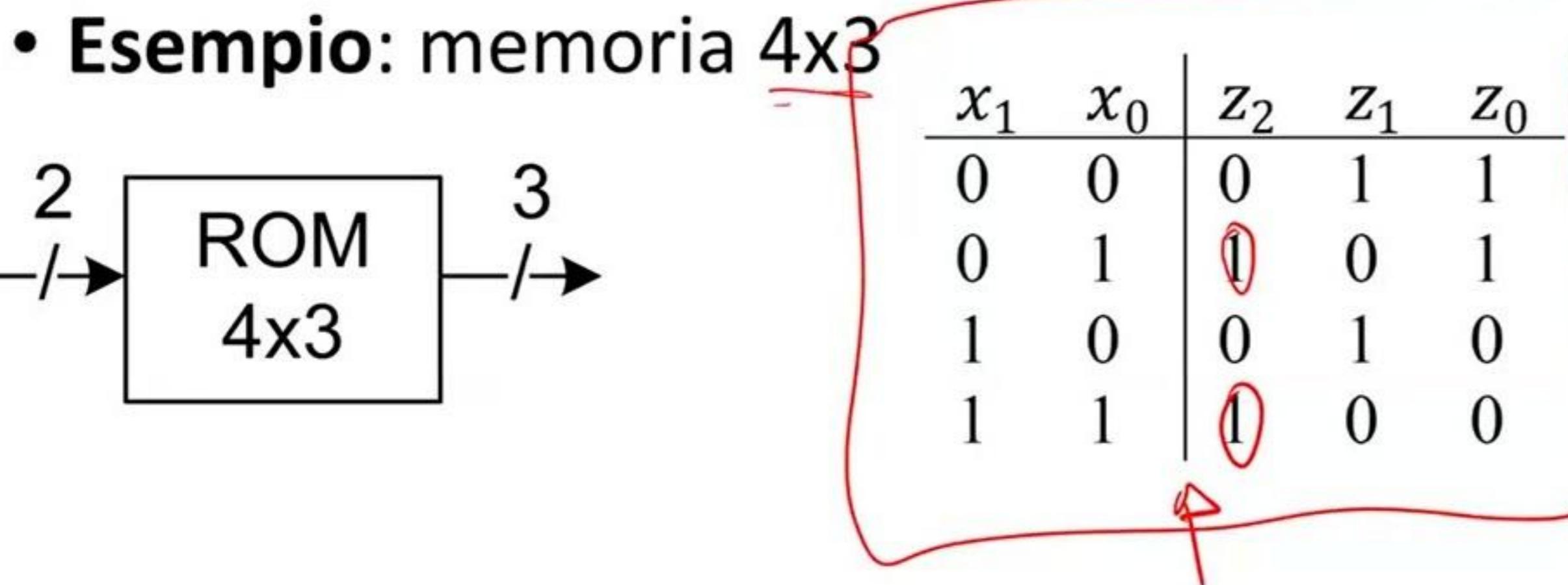
# Memorie read-only (ROM) (cont.)

- Schema concettuale, per semplificazione dalle RAM statiche
  - Uscite supportate da **porte tri-state**
  - devono poter coesistere su bus condivisi con altri dispositivi



# Memorie ROM (cont.)

- Una memoria ROM di  $2^N$  celle di  $M$  bit ciascuna è una rete combinatoria, con  $N$  ingressi ed  $M$  uscite.
  - Più le porte tri-state e la logica associata (non disegnate)



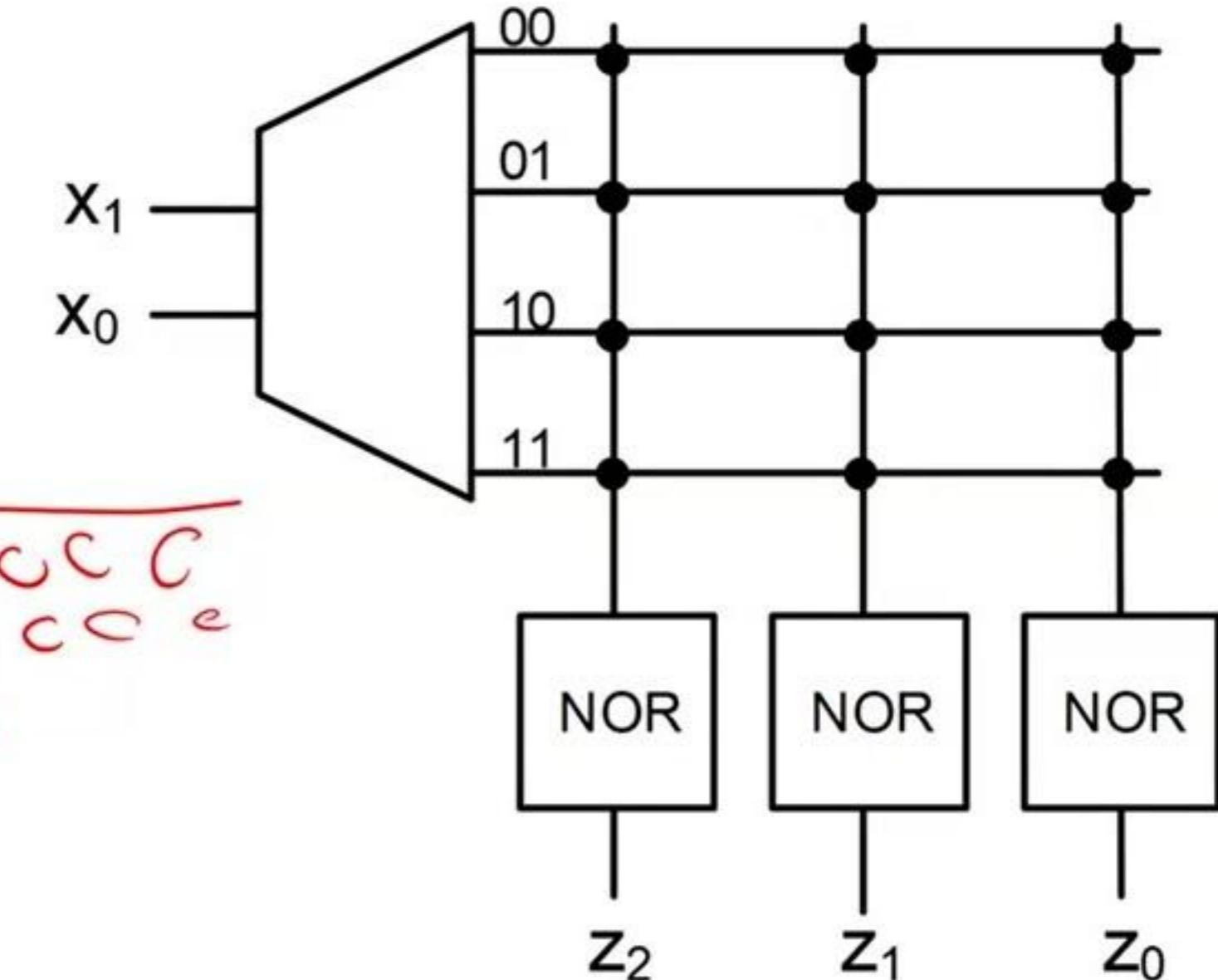
# ROM programmabili

- Una ROM è realizzata su un singolo chip di silicio,
  - deve uscire dalla fabbrica **già programmata** (cioè con i contatti già stabiliti)
  - il processo di programmazione è parte integrante del processo di fabbricazione del chip
- Preparare lo “stampo” per una ROM ha un costo fisso molto elevato
  - la realizzazione di una ROM si giustifica soltanto con scale molto larghe
- Esistono vari tipi di **ROM programmabili**

# ROM programmabili (cont.)

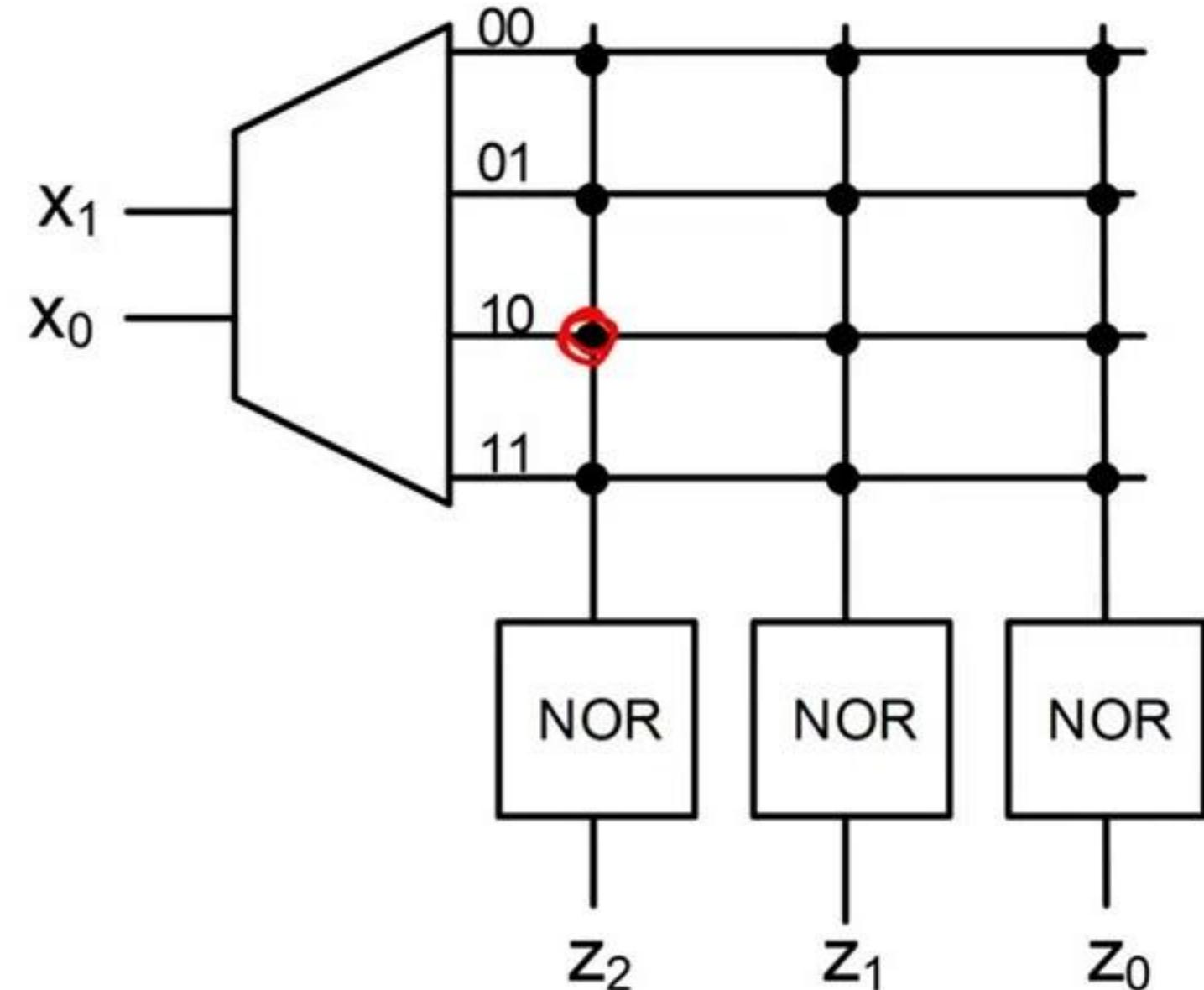
- I contatti tra le uscite degli AND e gli ingressi dei NOR ci sono tutti.
  - il contenuto di ogni cella è **zero**.
- Posso “**disabilitare**” qualcuno di questi contatti
  - Programmare la ROM in modo che ogni cella contenga un valore arbitrario

-- | Pig  
+----+  
| CCC  
| CCC  
+----+



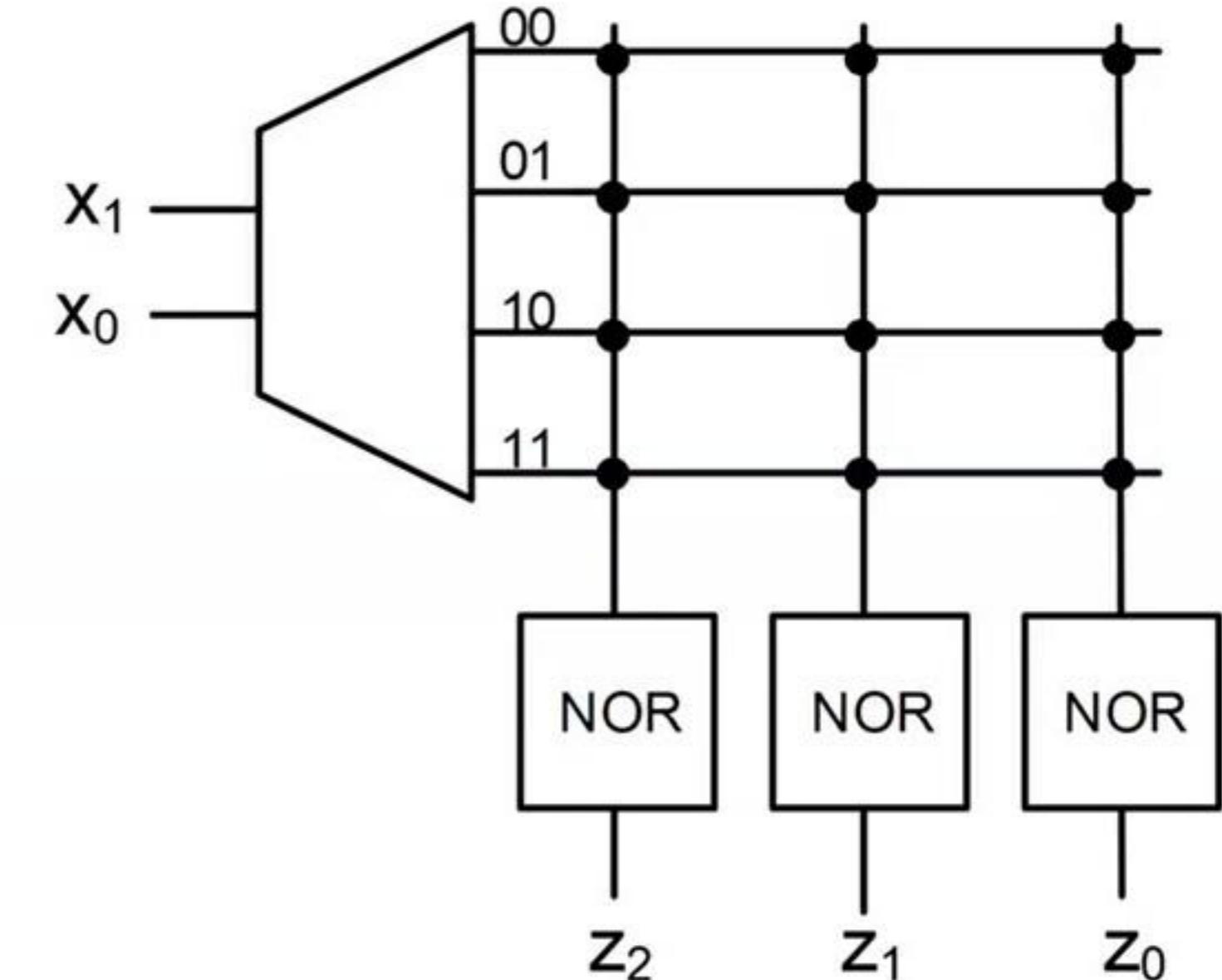
# ROM programmabili (cont.)

- **EPROM** (Erasable Programmable ROM) Le connessioni sono fatte con dispositivi elettronici **(field-effect transistors)**
  - programmabili per via elettrica
  - cancellabili tramite esposizione a raggi ultravioletti.
  - Possono pertanto essere cancellate e riprogrammate



# ROM programmabili (cont.)

- Le EPROM si programmano con un apposito **programmatore di EPROM**.  
~~on chip~~
- La **scarica** di una EPROM prende qualche minuto
  - **endurance**: capacità di sopportare riprogrammazioni - O(10K-100K volte)
  - **data retention**: periodo per il quale si può far affidamento sul contenuto di una EPROM – O(10-100 anni)



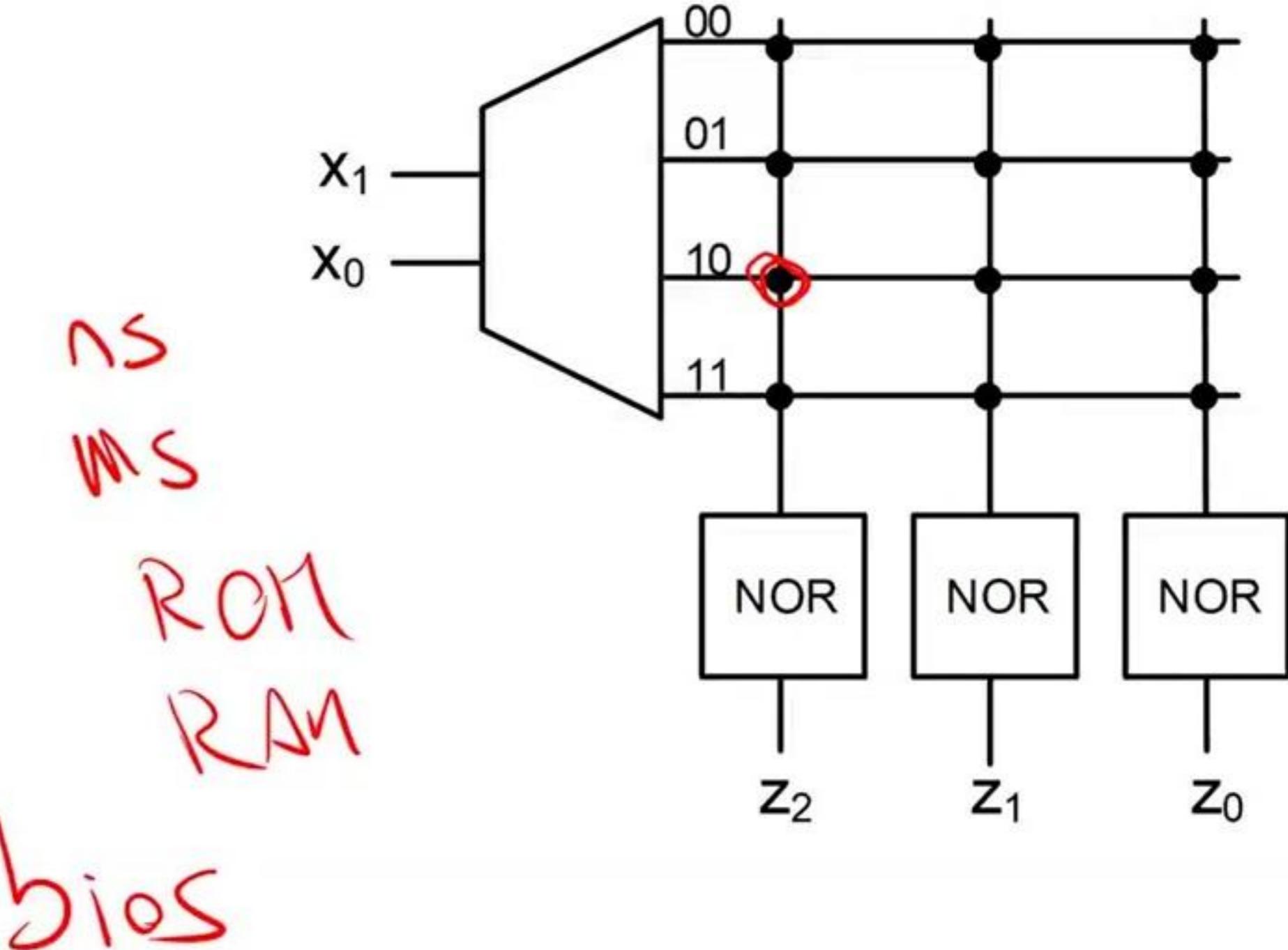
# ROM programmabili (cont.)

- **EEPROM (E<sup>2</sup>PROM)**: (Electrically Erasable Programmable ROM). Possono essere programmate e cancellate tramite segnali elettrici appositi

- diversi da quelli del funzionamento a regime
- riprogrammate direttamente on chip
- Endurance, data retention

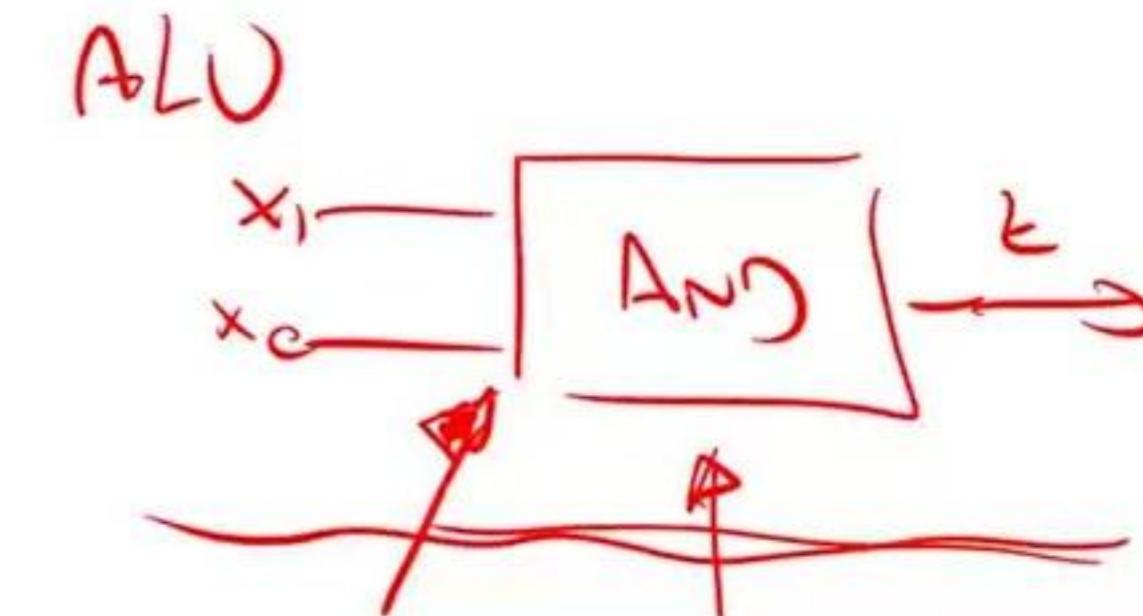
- ROM (invece che RAM): la programmazione è un modo operativo differente da quello della normale attività della memoria.

- il **# di volte** in cui si può riprogrammare una EEPROM è comunque limitato
  - il **tempo** che ci vuole a riprogrammare una EEPROM è >> (ms) del tempo di lettura
- • le **tensioni** che si usano non sono le stesse (12-18V, contro 5)
- Comunque non volatile



# Il linguaggio Verilog

- Hardware Description Language



- Per descrivere le reti logiche, soprattutto quelle **complesse**, fa comodo poter adottare una notazione **testuale**.
  - Compatta (quindi facile da trasportare)
  - Interpretabile in modo automatico da una macchina, che può simulare il comportamento della rete logica, dati certi ingressi alla medesima
- Il Verilog consente di fare
  - molte più cose di quelle che vediamo
  - in molti modi differenti.
- Noi adotteremo un particolare stile, funzionale ai compiti che dobbiamo svolgere, e quindi introdurremo il Verilog in modo **informale**, attraverso esempi

# Esempio

Module

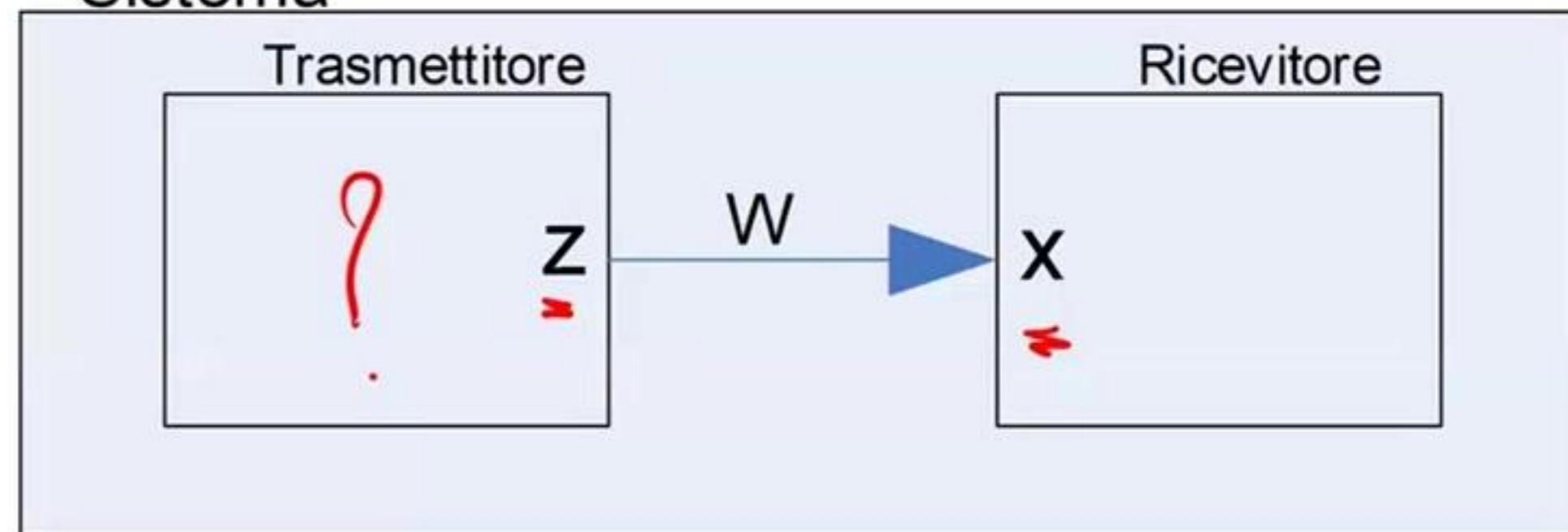
- Prendiamo l'esempio visto all'inizio del corso, e scriviamolo in Verilog

```
module Trasmettitore(z);
    output z;
    //descrizione della struttura interna // 
endmodule

module Ricevitore(x);
    input x;
    //descrizione della struttura interna
endmodule

module Sistema;
    wire w;
    Trasmettitore T(w);
    Ricevitore R(w);
endmodule
```

Sistema



Rete di tipo "Trasmettitore", che ha una variabile di I/O che riferisco come "z" all'interno di questa descrizione. La variabile z è un'uscita per il modulo "Trasmettitore"

Alternativamente:

Trasmettitore T(.z(w));  
Ricevitore R(.x(w));

Nome del modulo ed elenco dei fili di ingresso ed uscita (nessuno, in questo caso)

Wire = "filo", cioè variabile logica.

# Esempio

$x_7-x_0[7:4]$

```

module Rete_di_Tipo_A(z7_z0, x3_x0, b0);
  input [3:0] x3_x0;
  input b0;
  output [7:0] z7_z0;
  //descrizione della struttura interna delle reti di tale tipo
endmodule

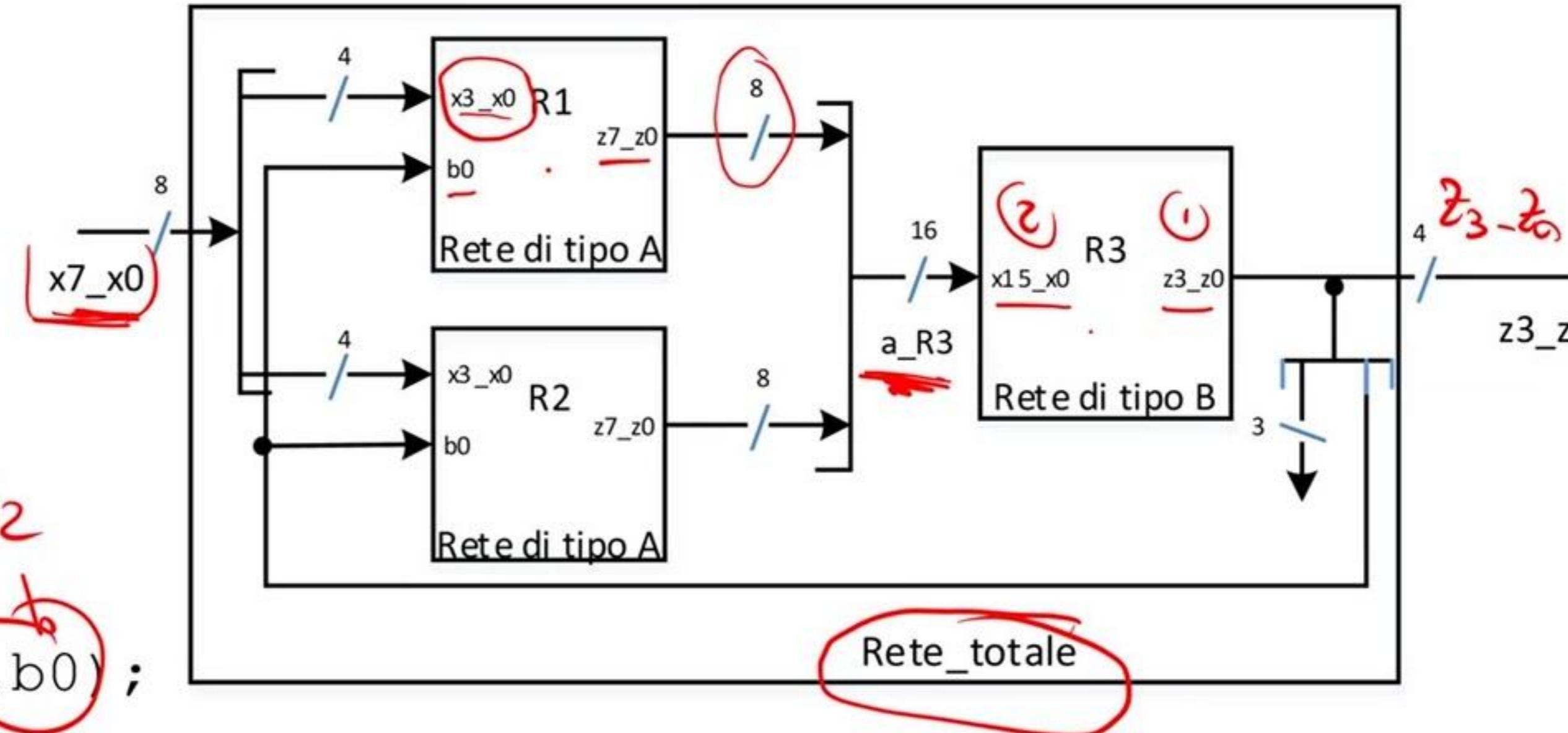
```

(1) (2)

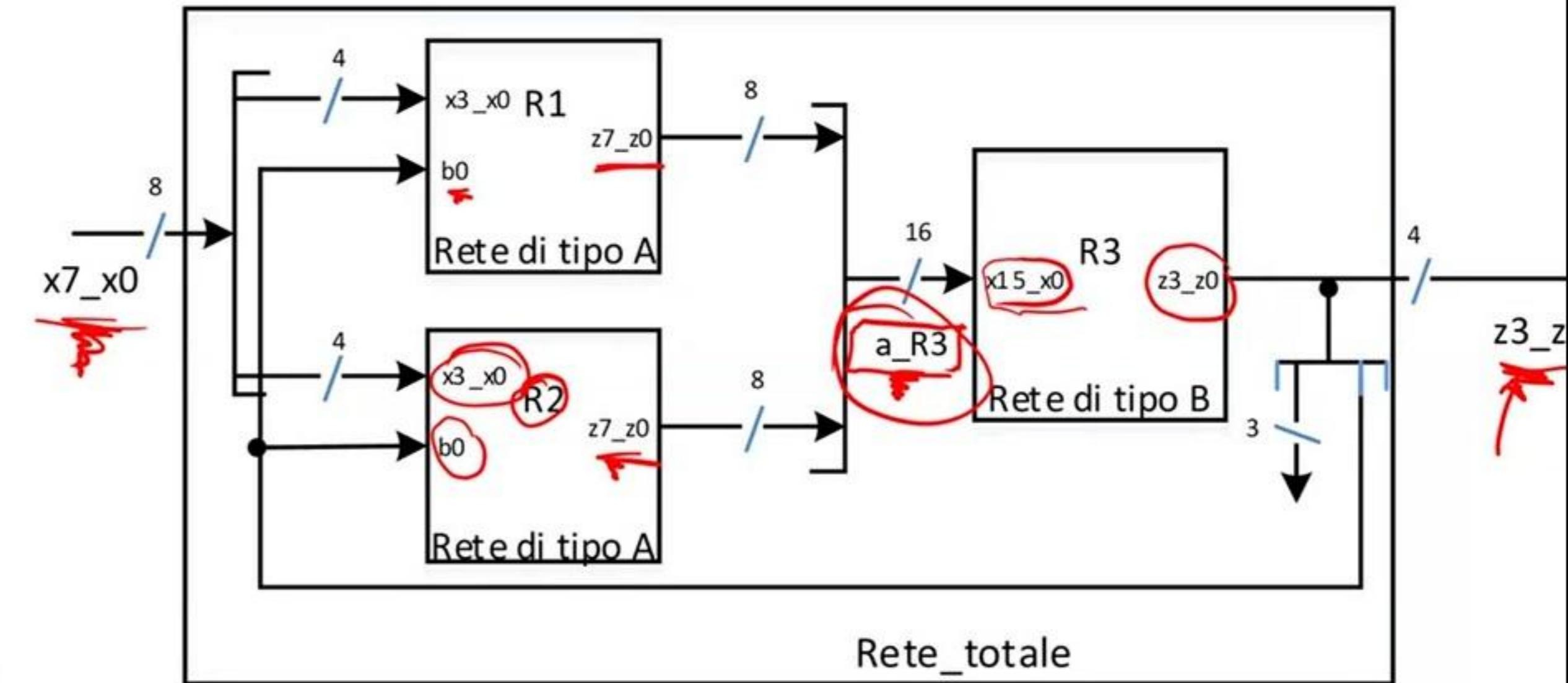
```

module Rete_di_Tipo_B(z3_z0, x15_x0);
  input [15:0] x15_x0;
  output [3:0] z3_z0;
  //descrizione della struttura interna delle reti di tale tipo
endmodule

```



## Esempio (cont.)



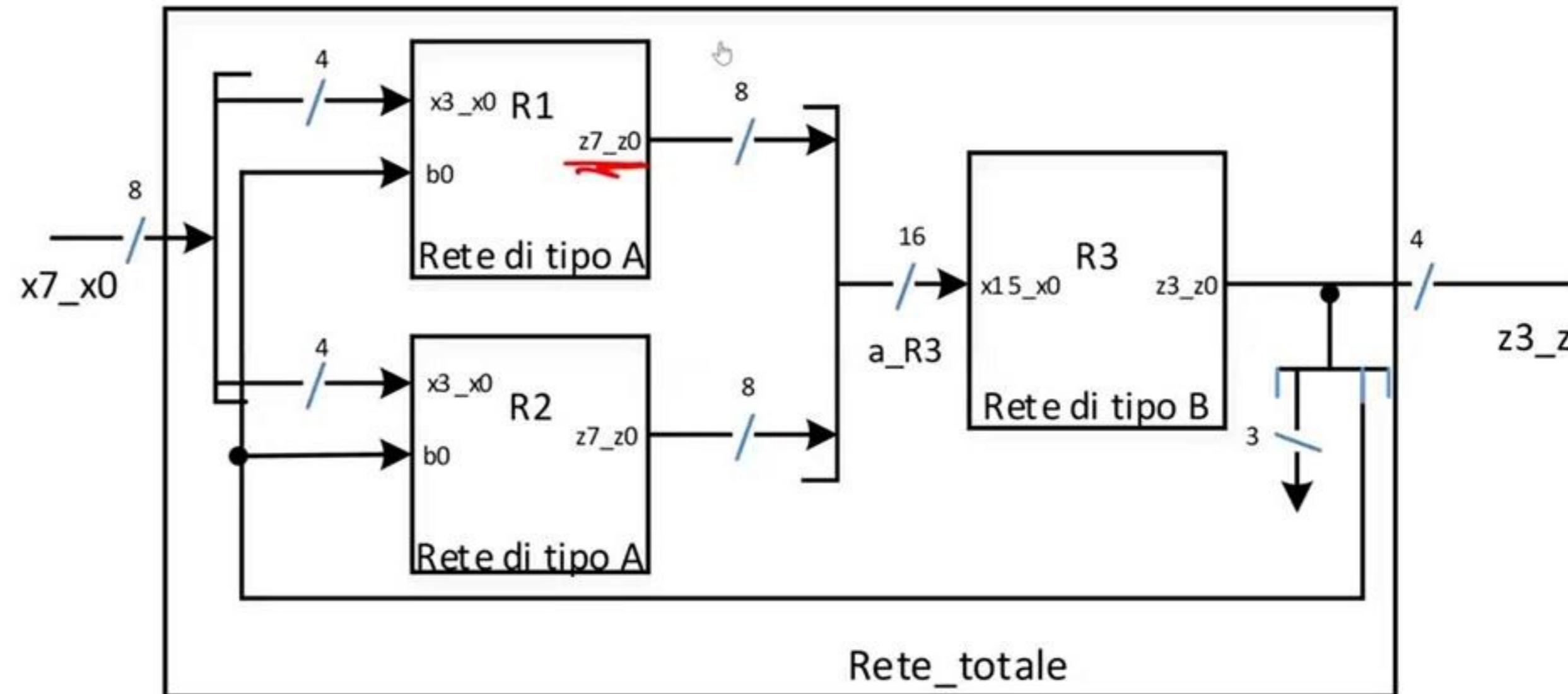
```
module Rete_Totale(z3_z0, x7_x0);
    input [7:0] x7_x0;
    output [3:0] z3_z0;
    wire [15:0] a_R3;
    Rete_di_Tipo_A R1(a_R3[15:8], x7_x0[7:4], z3_z0[0]),
    → R2(a_R3[7:0], x7_x0[3:0], z3_z0[0]);
    Rete_di_Tipo_B R3(z3_z0, a_R3);
endmodule
```

is  
↳

0 ↓

1 ↗ z3\_z0[3:0]

# Esempio (cont.)



```

module Rete_Totale(z3_z0,x7_x0);
  input [7:0] x7_x0;
  output [3:0] z3_z0;
  wire [15:0] a_R3;
  Rete_di_Tipo_A
    R1 (.z7_z0(a_R3[15:8]), .x3_x0(x7_x0[7:4]), .b0(z3_z0[0])),
    R2 (.z7_z0(a_R3[7:0]), .x3_x0(x7_x0[3:0]), .b0(z3_z0[0]));
  Rete_di_Tipo_B R3 (.z3_z0(z3_z0), .x15_x0(a_R3));
endmodule

```

Annotations in red highlight specific connections:

- Rete\_di\_Tipo\_A:** The connection between the  $.z7_z0$  output of R1 and the  $a_R3[15:8]$  input of R2 is highlighted.
- Rete\_di\_Tipo\_B:** The connection between the  $.z3_z0$  output of R3 and the  $.z3_z0$  output of the Rete\_totale module is highlighted.

# Note sulla sintassi del Verilog

- Il Verilog è case sensitive. b
- Le parole chiave (module, endmodule, etc.) sono **tutte minuscole**, e come tali vanno scritte

Pippo PIPPO

- Gli **identificatori** possono contenere lettere, numeri e underscore, e non possono cominciare con un numero

# Note sulla sintassi Verilog (cont.)

- Le **costanti** si possono scrivere in base 2, 16, 10. Il formato di una costante è il seguente:

'D32       $n'$   $\{B,D,H\}$  valore      10

- dove:
  - $n$  è il numero di **bit** su cui va intesa la costante. Se omesso viene ricavato dalla dimensione delle variabili in gioco
  - $B, D, H$  indica che il resto della costante va interpretato in base 2, 10, 16
- 8'D32** indica la costante 32 (espressa in decimale), rappresentata su 8 bit (00010000)
- Le costanti binarie si possono scrivere anche raggruppando i bit a gruppi di quattro ed inserendo underscore. Ad esempio: **16'B0010\_1000\_0101\_1100** (per leggibilità).
- 10 è la base di default**. Scrivere  **$h = 0101$**  significa scrivere che  $h$  vale centouno. Se  $h$  è una variabile a 4 bit, bisogna scrivere  **$h = \underline{\underline{'B0101}}$**

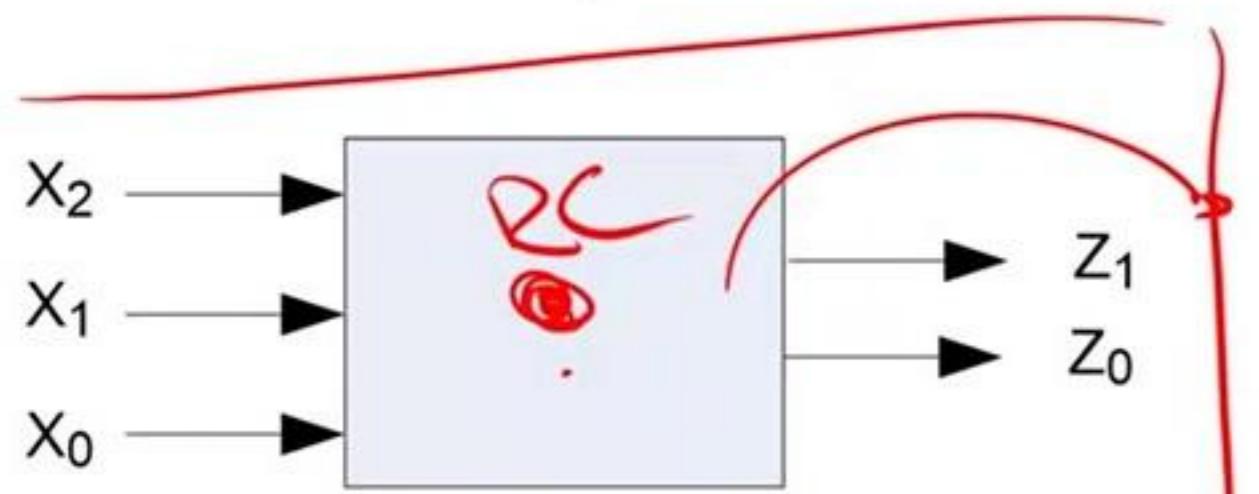
# Note sulla sintassi Verilog (cont.)

- In Verilog è possibile assegnare ad una variabile logica quattro valori:
  - **Uno:** 1'B1, o semplicemente 1 (che è decimale, ma vale uno lo stesso)
  - **Zero:** 1'B0, o semplicemente 0 (che è decimale, ma vale zero lo stesso)
  - **Alta impedenza:** 1'BZ (da usare soltanto per valori di uscita)
  - **Non specificato:** 1'BX (da usare soltanto per valori di uscita)
- **Commenti:** tra /\* ... \*/, oppure tra // e fine riga (come in C++)
- Ciascuna riga va terminata con un punto e virgola

# Descrizione di reti combinatorie in Verilog

-? - ; - ; if distretto

- Si copia la **tabella di verità** in maniera pedissequa



x <sub>2</sub>	x <sub>1</sub>	x <sub>0</sub>	z <sub>1</sub>	z <sub>0</sub>
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 #1 {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) ? 'B00:
        ({x2, x1, x0} == 'B111) ? 'B00;
endmodule
```

ds. continuo #3  
At

# Descrizione di reti combinatorie in Verilog

- Si copia la tabella di verità in maniera pedissequa

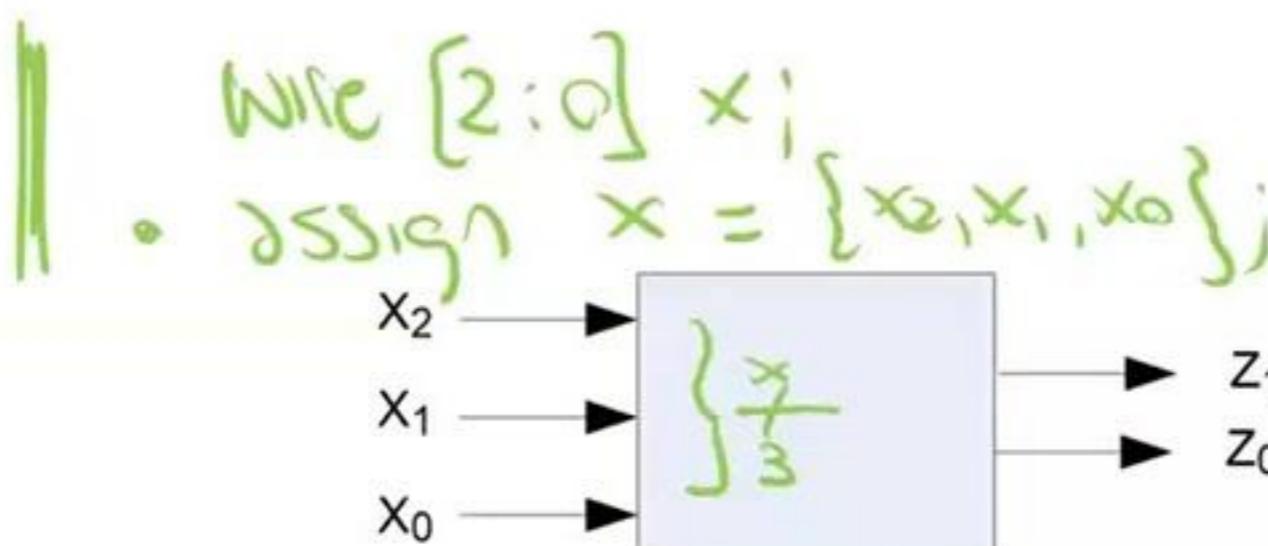


$x_2$	$x_1$	$x_0$	$z_1$	$z_0$
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	0	0
1	1	1	0	0

```
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:  
/*default*/'B00;
```

# Descrizione di reti combinatorie in Verilog

```
module RC(z1, z0, x2, x1, x0);
    input x2, x1, x0;
    output z1, z0;
    assign #T {z1, z0}=F(x2, x1, x0);
    function [1:0] F;
        • input x2, x1, x0;      input [2:0] x;
        casex( {x2, x1, x0} )
            'B000 : F='B00;
            'B001 : F='B01;
            'B010 : F='B10;
            'B011 : F='B10;
            'B100 : F='B11;
            'B101 : F='B11;
            'B110 : F='B00;
            'B111 : F='B00;
        endcase
    endfunction
endmodule
```



x <sub>2</sub>	x <sub>1</sub>	x <sub>0</sub>	z <sub>1</sub>	z <sub>0</sub>
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

In un costrutto  
“casex...endcase” posso  
usare la parola chiave  
**“default”**.

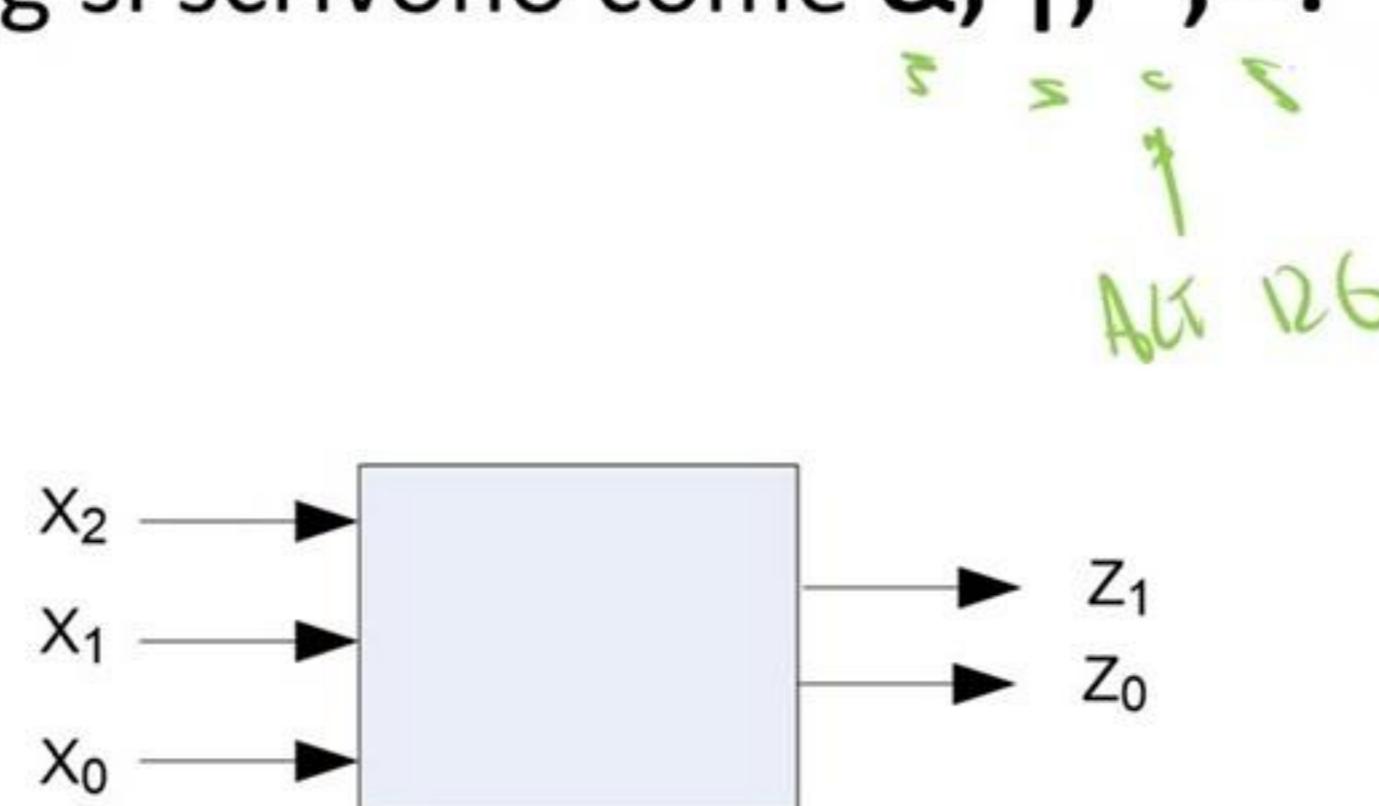
'B001 : F='B01;  
'B01? : F='B10;  
'B10? : F='B1X;  
**default** : F='B00;

Posso usare “?” per dire che  
il valore del bit che non  
specifico è irrilevante

# Sintesi di Reti Combinatorie

- Si può scrivere ugualmente in Verilog *di Algebra di Boole*
- Nei costrutti assign si possono inserire **espressioni logiche**
  - Contenenti operatori AND, OR, NOT, XOR
  - Che in Verilog si scrivono come **&**, **|**, **~**, **^**.

$x_2$	$x_1$	$x_0$	$z_1$	$z_0$
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

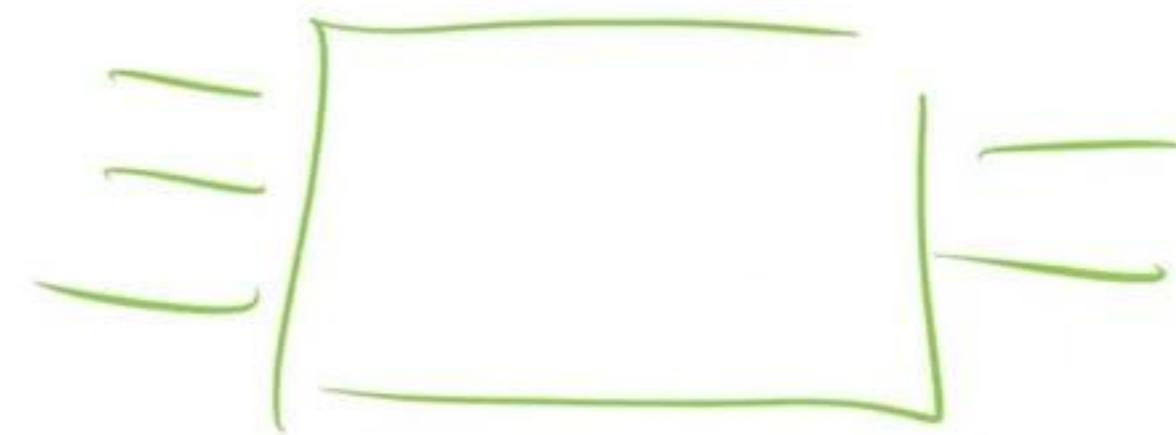


- $$z_1 = \overline{x_2} \cdot x_1 + x_2 \cdot \overline{x_1} = x_2 \oplus x_1$$
- $$z_0 = x_2 \cdot \overline{x_1} + x_0 \cdot \overline{x_1}$$

# Sintesi di Reti Combinatorie (cont.)

$$z_1 = \overline{x_2} \cdot x_1 + x_2 \cdot \overline{x_1} = x_2 \oplus x_1$$

$$z_0 = x_2 \cdot \overline{x_1} + x_0 \cdot \overline{x_1}$$



```
- module RC(z1, z0, x2, x1, x0);
  input x2, x1, x0;
  output z1, z0;
  assign #3 z1=(~x2 & x1) | (x2 & ~x1); // assign #T z1=x2 ^ x1;
  assign #3 z0=(x2 & ~x1) | (x0 & ~x1);
endmodule
```

# Descrizione e sintesi in Verilog

- Una descrizione dice **che cosa fa** la rete.
  - Tabella di verità
  - Lo scopo di una descrizione è di essere **verificabile** (da un utente umano o da una macchina).
- Una sintesi dice **quali porte ci metto** affinché succeda quello che c'è scritto nella descrizione.
- Quella appena scritta è una **sintesi**, e non una **descrizione**.
- Il fatto che entrambe si possano scrivere in sintassi Verilog è spesso fonte di confusione.
  - Dobbiamo imparare a tenere le due cose distinte.