Reti Sequenziali Asincrone

Giovanni Stea

a.a. 2016/17

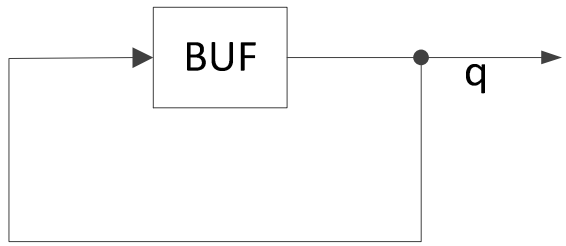
Ultima modifica: 17/11/2016

Sommario

1 La funzione di memoria

Le reti combinatorie sono **prive di memoria**: ad un dato stato di ingresso corrisponde (**a regime**) un dato stato di uscita. Per avere reti **sequenziali**, cioè reti la cui uscita dipende 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**. Prendiamo un esempio semplice:



In questo semplice anello esistono **due situazioni di stabilità**:

1) L'uscita vale 0 (e quindi va in ingresso al buffer, dove si rigenera)
2) L'uscita vale 1 (come sopra)

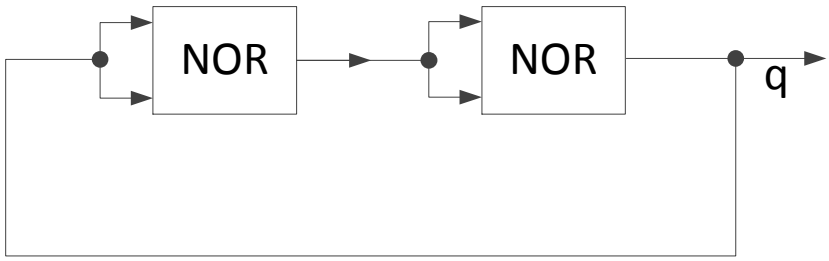
Possiamo dire che l'anello si può trovare **in due stati**, che possiamo chiamare S_0, S_1 (i nomi sono arbitrari), e che corrispondono allo stato in cui l'uscita vale 0 ed 1 rispettivamente.

Si noti che **la presenza del buffer è fondamentale**, in quanto garantisce che a q è associato un valore logico, impostato dal buffer medesimo. Se lo tolgo, q è connessa ad un filo staccato, quindi non ha un valore logico.

Una rete fatta così **non serve a niente in pratica**, perché non è possibile impostare né modificare il valore di q. Quando viene data tensione al sistema, questo si porterà in uno dei due stati S_0, S_1 in maniera **casuale**, e lì resterà finché non tolgo la tensione. Non è quindi possibile che questo anello **memorizzi bit diversi in tempi diversi** (a meno di non togliere e riattivare la tensione, e comunque sempre in modo casuale).

Vediamo di complicare un po' lo schema. Posso:

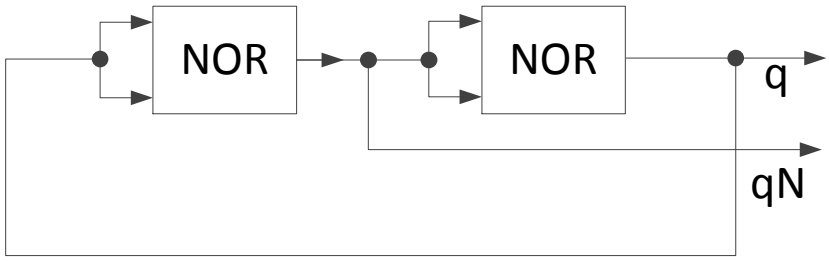
a) Sostituire il buffer con una coppia di **NOT**
b) Implementare ciascun NOT a porte **NOR** (per motivi che saranno chiari più avanti)



Nel circuito che ottengo sono presenti contemporaneamente **sia il bit 1 che il bit 0**. Infatti,

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

Già che ci siamo, possiamo sfruttare questa caratteristica per **dotare il circuito di un'altra uscita**, che chiamo **qN** (negata). Per convenzione, si dice che il circuito **memorizza il bit il cui valore è quello di *q*.**

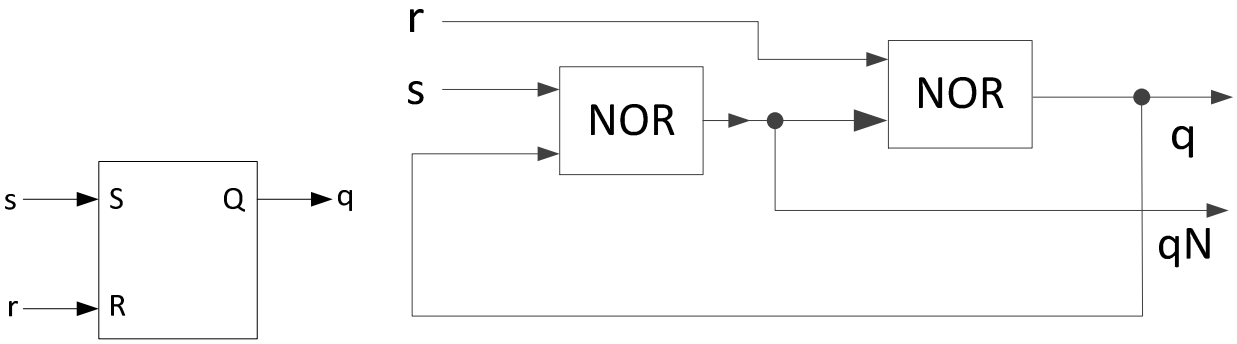


Vediamo cosa succede quando si **accende** questo circuito, connettendolo alla tensione. Se all'accensione ***q* e *qN* sono discordi**, la rete si trova già in uno dei due stati stabili, e lì resta. Se, invece, *q* e *qN* sono **concordi**, <u>in teoria</u> ciascuna delle due uscite oscilla all'infinito, con un tempo pari al tempo di accesso 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 *qN* sono discordi, che rimane stabile.

Anche in questo anello non è possibile memorizzare **bit diversi in tempi diversi**. Vediamo però come quest'ultima proprietà si possa facilmente introdurre, data la struttura che abbiamo impostato.

1.1 Il latch SR

È chiaro che, per poter impostare un valore in uscita, è necessario che un circuito abbia **degli ingressi che si possano pilotare**. Prendiamo **un ingresso ci ciascun NOR** e consideriamolo come filo di ingresso:



La rete che si ottiene è detta **latch SR** o (comunemente, ma impropriamente) **flip-flop SR**. "S" sta per **set**, mentre "R" sta per **reset**. Entrambe le variabili di ingresso si dicono **attive alte**, a indicare che la funzione che è indicata dal loro nome viene eseguita quando il valore dell'ingresso è pari a 1. Quando s=1 sto dando un comando di set. In caso contrario, si direbbero **attive basse**.

Vediamo che succede quando forniamo alcuni stati di ingresso:

- *s*=1, *r*=0: 0

- la **prima** porta NOR ha **un ingresso a 1**, quindi mette l'uscita a 0 (qualunque sia il valore di *q*). Pertanto, *qN*=0.
- La **seconda** porta NOR ha in ingresso **00**, quindi mette l'uscita *q*=1.

La rete si porta, quindi, nello stato **S1**, in cui **memorizza il bit 1**. In altre parole, si **setta**.

- *s*=0, *r*=1:
  - la **seconda** porta NOR ha **un ingresso a 1**, quindi mette l'uscita a 0 (qualunque sia il valore di *qN*). Pertanto, *q*=0.
  - La **prima** porta NOR ha in ingresso **00**, quindi mette l'uscita *qN*=1.

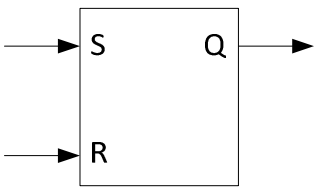
  La rete si porta, quindi, nello stato **S0**, in cui **memorizza il bit 0**. In altre parole, si **resetta.**
- *s*=0, *r*=0:
  - l'uscita della prima porta NOR **vale 0 se q=1, e vale 1 se q=0**. Pertanto, $qN = \overline{q}$.
  - La seconda porta NOR ha in ingresso **0 e qN**, quindi l'uscita *q* vale $\overline{qN}$.

  La rete, quindi, **conserva l'uscita al valore che aveva precedentemente**.

Quest'ultima cosa rende la rete **una rete sequenziale**: quando lo stato di ingresso è s=0, r=0, la rete **rimane nello stato stabile, S0 o S1**, nel quale si è portata in precedenza. In altre parole, **ricorda** l'ultimo comando (set o reset) ricevuto. Peraltro, il nome "latch" in Inglese ricorda **la chiusura a scatto**, il che è appropriato.

Manca da capire cosa succeda quando diamo in ingresso **lo stato *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. Pertanto, questo stato di ingresso **non è permesso** in un corretto pilotaggio.

Un modo per descrivere il comportamento del latch SR è dato dalla **tabella di applicazione** (attenzione a non confonderla con una tabella di **verità**). In questa si riporta – 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
0	0	0	-
0	1	1	0
1	0	0	1
1	1	-	0

- Se l'uscita è a zero, e voglio che ci rimanga, basta che **non dia un comando di set**. Posso o resettare (01), o conservare (00).
- Se l'uscita è a uno e voglio che ci rimanga, basta che **non dia un comando di reset**. Posso o settare (10) o conservare (00).
- Se voglio che l'uscita passi da 0 a 1, devo necessariamente **settare (10)**
- Se voglio che l'uscita passi da 1 a 0, devo necessariamente **resettare (01)**

Parliamo adesso delle **regole di pilotaggio** di un latch SR. Per le reti combinatorie ne conosciamo due:

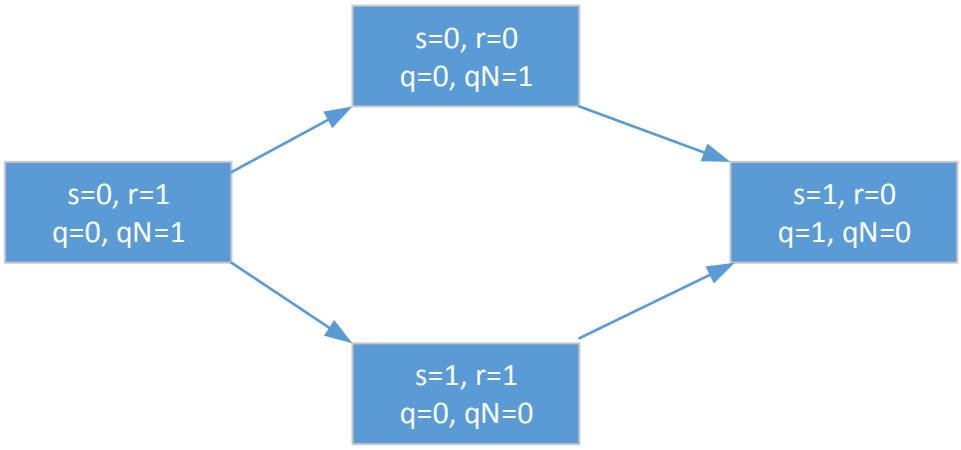
1) Pilotaggio in modo fondamentale: cambiare gli ingressi soltanto quando la rete è a regime
2) Stati di ingresso consecutivi devono essere adiacenti

Nel nostro caso, la regola 1) deve essere rispettata. Quando avremo le idee più chiare sulle reti sequenziali asincrone saremo in grado di **quantificare il tempo che ci vuole** perché una rete vada a regime. Per adesso, facciamo finta che per queste reti conosciamo una misura **analoga al tempo di attraversamento**, dalla quale possiamo desumere quando variare gli ingressi.
Per quanto riguarda la regola 2), in generale nelle RSA è di **importanza fondamentale**. Infatti, se non la rispetto, possono presentarsi in ingresso degli stati spuri, che mi fanno evolvere la rete in modo del tutto diverso. Però **nel solo caso del latch SR**, posso evitare di rispettarla.
Il latch SR è **robusto a pilotaggi scorretti**, ed è un bene, perché – come vedremo nel corso delle lezioni – è la rete che sta alla base dei registri e di tutti gli elementi di memoria.

s=1, r=0	q=1, qN=0
s=0, r=1	q=0 qN=1

Supponiamo che sia presente in ingresso lo stato **s=1, r=0**, e che si passi a **s=0, r=1**. Visto che non è possibile che entrambe le variabili cambino valore **contemporaneamente**, si passerà **o dallo stato intermedio 00 o da quello 11.**

- Se si passa dallo stato 00 non ci sono problemi. In quello stato, infatti, il latch SR **conserva** l'uscita al valore precedente.
- Se si passa dallo stato 11 **non ci sono problemi lo stesso**. Infatti, per un breve periodo entrambe le uscite saranno a 0, ma non potrebbe essere altrimenti, in quanto **anche due uscite non possono cambiare contemporaneamente**, e quindi delle due una varia prima in ogni caso.



Le stesse considerazioni si applicano anche alla **transizione opposta.**

Quello che **non deve mai succedere** è che si dia in ingresso s=1, r=1 (che peraltro è uno stato di ingresso che chi pilota la rete si deve ricordare di non impostare), e 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, di fatto, generare un bit a caso in uscita.

Il **tempo** che ci mette un **latch SR a stabilizzarsi** è di pochi ns.

1.2 Il problema dello stato iniziale

Abbiamo cominciato ad assorbire l'idea che il latch SR è l'elemento **alla base dei circuiti di memoria**. Abbiamo visto che, **all'accensione, il bit contenuto nell'SR** (o, se si preferisce, il suo **stato interno**) **è casuale**. All'accensione del calcolatore, alcuni elementi di memoria **possono** avere un **contenuto casuale** (esempio tipico: le celle della memoria RAM), ma altri no (ad esempio, **i registri del processore**, F e IP). Serve quindi un modo per **inizializzare un elemento di memoria** al valore voluto.

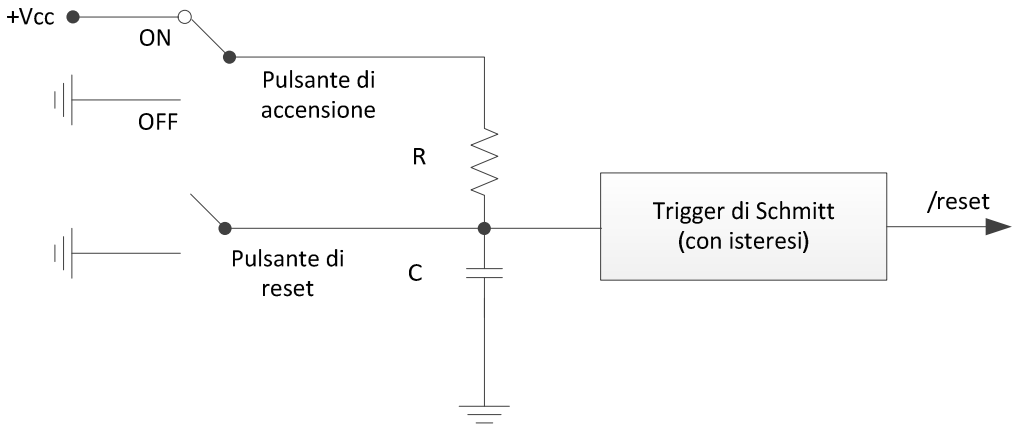
L'inizializzazione avviene **tutte le volte che si preme il pulsante di reset del calcolatore**. Il suo effetto è quello di inserire in tutti gli elementi di memoria (o meglio, in tutti quelli per cui mi interessa avere un valore iniziale) il contenuto iniziale desiderato.

In un calcolatore si definisce **fase di reset iniziale** una fase distinta da quella di **normale operatività**, nella quale si inizializzano gli elementi di memoria. Si tenga presente un problema di **nomenclatura**: con il nome **reset** si intendono due cose:

- Un **comando che mette a zero** l'uscita del latch SR;
- La **fase di ritorno ad una condizione iniziale** di un sistema di elaborazione.

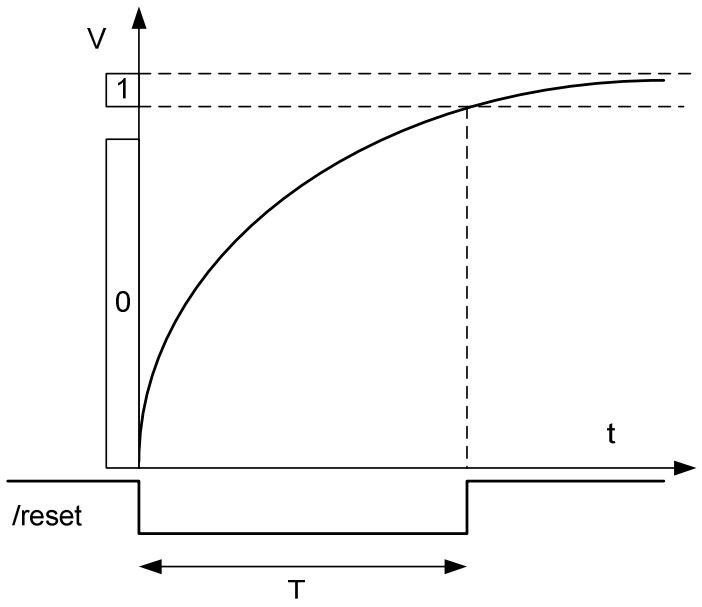
Dal che le persone sono portate a pensare che la condizione iniziale di un calcolatore è quella in cui in tutti gli elementi di memoria c'è scritto zero, che non è assolutamente vero.

Vediamo come è fatta la circuiteria per l'inizializzazione al reset.



Il circuito RC sulla sinistra si **carica in tempi dell'ordine del microsecondo**, e la tensione ai capi del condensatore diventa prossima a Vcc. Quando viene **premuto il pulsante di reset**, il condensatore **si scarica a massa**. La scarica non è istantanea, ma il contatto del pulsante di reset dura abbastanza perché essa avvenga. Non appena il pulsante di reset viene rilasciato, il condensatore ricomincia a caricarsi come fa all'accensione.

La tensione ai capi del condensatore viene fatta passare attraverso **uno squadratore di tensione**, detto trigger di Schmitt, che dà in uscita il valore 1 oppure 0, a seconda che la tensione sia sopra una soglia alta o sotto una soglia bassa. Pertanto, quello che succede quando si preme il pulsante di reset è che l'uscita di questo circuito resta bassa per un bel po' (nell'ordine dei microsecondi).



La variabile **/reset** è una variabile **attiva bassa**, e quindi si scrive con uno "/" davanti (che non è un operatore). In Verilog, visto che non si può usare "/" nei nomi di variabile, si conviene di indicare le variabili attive basse con un **simbolo di underescore** in fondo (nel caso, "**reset_**").

La variabile logica /reset, quindi, può essere usata per inizializzare gli elementi di memoria. La prassi a cui ci conformeremo è:

- quando /reset=0, l'elemento di memoria si porta nello stato interno iniziale desiderato, **indipendentemente dal valore dei suoi altri ingressi.**
- quando /reset=1, l'elemento di memoria funziona normalmente.

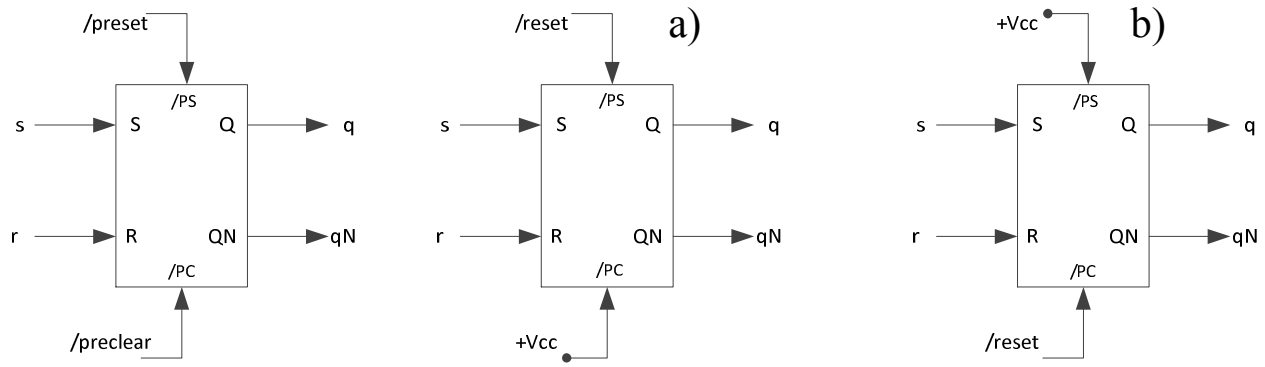
Per poterla applicare è necessario dotare un latch SR di **due ingressi aggiuntivi**, detti **/preset e /preclear**, entrambi attivi bassi, tali per cui:

- se /preset=/preclear=1, la rete si comporta come un latch SR;
- se /preset=0, la rete si porta nello stato S1 (indipendentemente dal valore degli ingressi s e r);
- se /preclear=0, la rete si porta nello stato S0 (indipendentemente dal valore degli ingressi s e r)
- /preset e /preclear non sono mai contemporaneamente a 0.

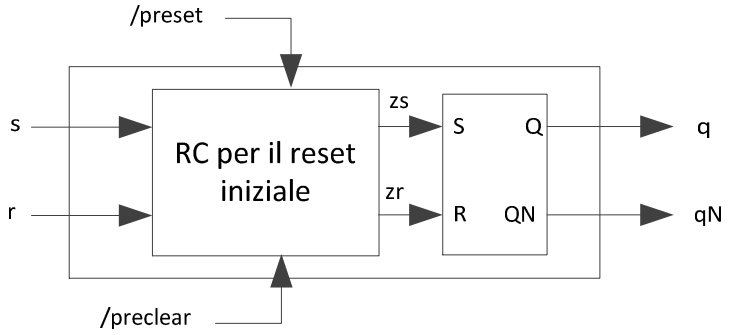
Con queste specifiche, è abbastanza chiaro cosa si debba fare:

a) Se si vuole **inizializzare a 1** l'elemento di memoria, si connette **/preset a /reset** e **/preclear a Vcc**.

b) Se si vuole **inizializzare a 0** l'elemento di memoria, si connette **/preclear a /reset** e **/preset a Vcc;**



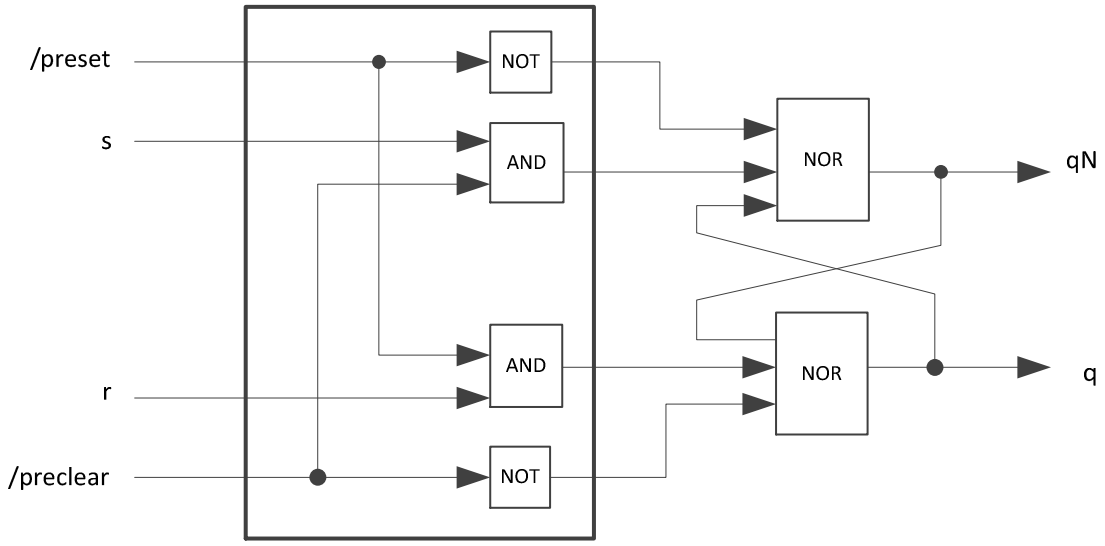
L'unica cosa che manca di fare è capire come **si deve modificare** il latch SR rispetto all'implementazione già vista. Conviene **mettergli davanti una rete combinatoria**, che ha come ingresso s,r, /preset e /preclear, ed in uscita due variabili zr, zs, che è facile da sintetizzare in forma SP.



/preset /preclear (sr) \ zs zr	00	01	11	10
00	--	10	00	01
01	--	10	01	01
11	--	10	11	01
10	--	10	10	01

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

La struttura che si ottiene può essere ulteriormente **ottimizzata**: basta rendersi conto che la porta OR della rete che genera zr, zs **non è necessaria**. Infatti, il latch SR è fatto a NOR, che sono delle OR seguite da una negazione. Pertanto, si verrebbero a trovare due porte OR in cascata. Quindi, gli ingressi della porta OR si possono portare direttamente alla porta NOR, risparmiando un livello.



1.3 Tabelle di flusso e grafi di flusso

Il latch SR è stato descritto **a parole**, oppure usando la **tabella di applicazione**. In realtà le RSA si descrivono usando **tabelle di flusso** o **grafi di flusso**. I due formalismi sono equivalenti, e verranno descritti in modo più generale quando parleremo delle RSA in modo generale. Per adesso fa comodo introdurre informalmente queste due modalità di descrizione, esemplificandole sul latch SR.

Una **tabella di flusso** è una tabella che descrive come si evolvono **lo stato interno** e **l'uscita** al variare degli stati di ingresso. È una matrice che ha:

- In **riga**, gli **stati interni** della rete (nel nostro caso sono **due**, S0 S1)

- In **colonna**, gli **stati di ingresso** (sono quattro: {s=0, r=0}, 01, 10, 11). Nello stato di ingresso **non vengono mai contate** le variabili per l'inizializzazione al reset, in quanto il loro ruolo è ininfluente durante la normale operatività della rete (sono entrambe ad 1, e quindi la rete combinatoria che le sente si comporta da corto circuito).
- Nelle celle, dei nomi di **stati interni della rete**.

E si interpreta come segue: lo stato scritto in colonna è lo **stato interno presente (SIP)**, e quello scritto nelle celle è lo **stato interno successivo (SIS)**, nel quale la rete transisce quando sono presenti contemporaneamente:

- Lo stato interno presente della riga
- Lo stato di ingresso della colonna.

Ad esempio, relativamente al funzionamento del latch SR, so che **"se sono nello stato *S*0 e l'ingresso s è a 0, rimango nello stato *S*0 (qualunque cosa faccia r)".** Posso quindi riempire alcune caselle della tabella.

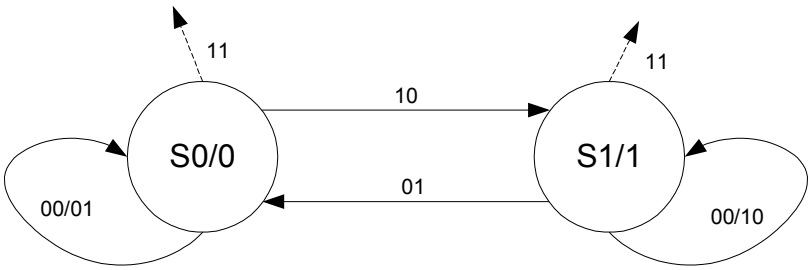
sr	00	01	11	10	q
S0	(S0)	(S0)	-	S1	0
S1	(S1)	S0	-	(S1)	1

In genere, si aggiunge a destra anche il **valore dell'uscita *q*** (o di entrambe), che dipende in questo caso soltanto dallo **stato interno presente.**

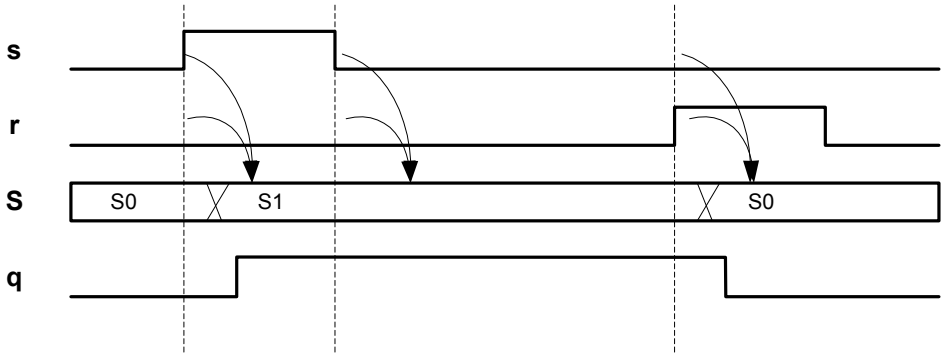
Una RSA **evolve (cioè modifica il proprio stato interno e/o la propria uscita) a seguito di cambiamenti dello stato di ingresso**. Non è difficile vedere che, se mi trovo nello stato (ad esempio) ***S*0** con ingresso 00, la rete **non evolve** (a meno che non cambi lo stato di ingresso). L'uscita rimane costante a 0, la variabile d'anello mantiene costante il proprio valore. Si dice, in questo caso, che **la rete ha raggiunto la stabilità (è a regime),** oppure che **lo stato interno S0 è stabile con stato di ingresso 00**. In tutti i casi in cui ciò succede, si mette **un cerchio** intorno allo stato interno stabile (è **errore** non metterlo).

Per lo stato di ingresso **11** il comportamento della rete **è non specificato**. Per questo motivo, si mette un trattino nella tabella di flusso.

Una maniera **del tutto equivalente** di rappresentare una rete sequenziale asincrona è quella dei **grafi di flusso**. Un grafo di flusso è un insieme di **nodi**, che rappresentano ciascuno uno **stato interno**, ed di **archi**, etichettati con uno **stato di ingresso**, diretti da uno stato ad un altro.



Gli archi che si perdono all'infinito sono relativi a stati di ingresso che non si possono (o non si debbono) verificare in corrispondenza di determinati stati interni. Gli archi che fanno "**orecchio**" indicano il fatto che uno stato interno è stabile per quegli stati di ingresso. Visto che l'uscita è funzione soltanto dello stato interno, posso scriverla direttamente **dentro il cerchio**.



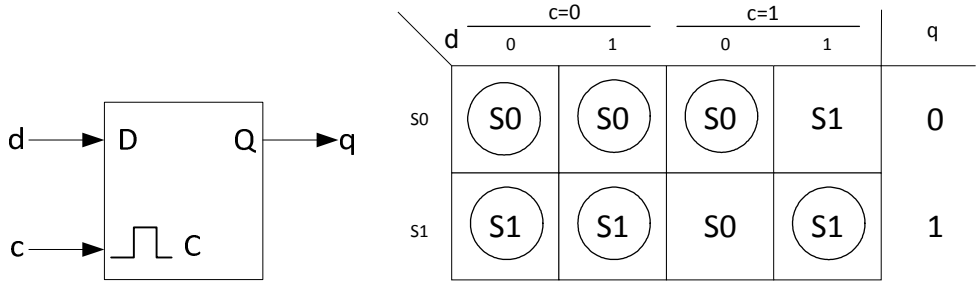
Questo è un **diagramma di temporizzazione**, il quale mi fa vedere che lo stato interno cambia (quando cambia) solo al variare dello stato di ingresso, e poi si stabilizza. L'uscita varia quando varia lo stato interno.

1.4 Il D-latch trasparente

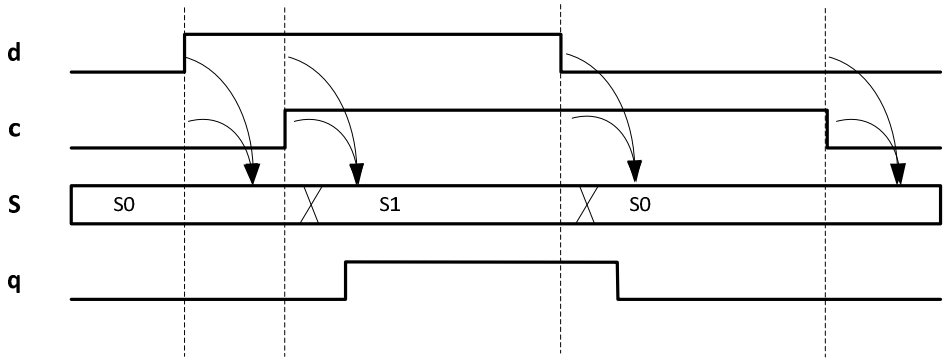
Abbiamo visto che il latch SR può memorizzare 1 o 0 a seconda del comando che gli viene dato (set, reset, conserva).

Il D-latch è una RSA con due variabili di ingresso, *d* (data) e *c* (control), ed una uscita *q* (vedremo poi che, come per il latch SR, in realtà è sempre disponibile anche l'uscita *qN*). 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. Quindi, sarà una rete che può trovarsi **in due stati**, uno nel quale ha memorizzato 0 ed uno nel quale ha memorizzato 1. Per questo, la tabella di flusso la posso disegnare come nella figura



Quando *c* vale 0, la variazione di ***d*** non è influente (non può cambiare lo stato della rete). Quando *c* vale 1, la variazione di **d** fa cambiare stato alla rete.



Quindi, per memorizzare un bit, basta

- Portare *c* ad 1
- Impostare *d* al valore da memorizzare
- Riportare *c* a 0.

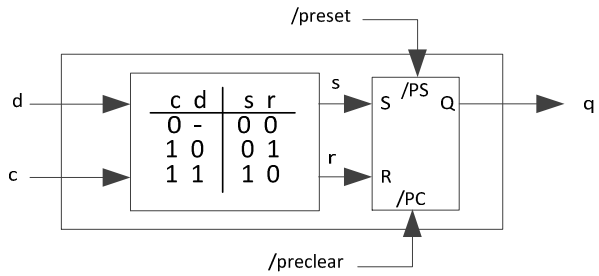
Una sintesi del D-latch si può ottenere facilmente a partire da quella di un latch SR. Supponiamo di avere un latch SR, a mettiamoci davanti una **rete combinatoria** che ha:

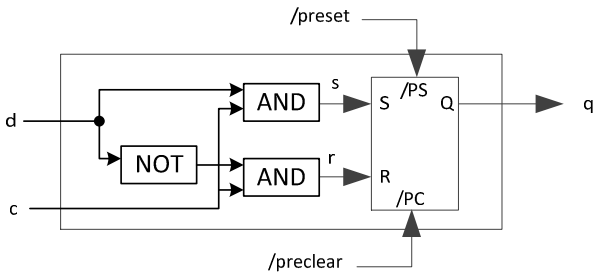
- Come ingressi, le due variabili *d* e *c*
- Come uscite, le due variabili *s, r*

E cerchiamo di capire come sintetizzare quest'ultima.

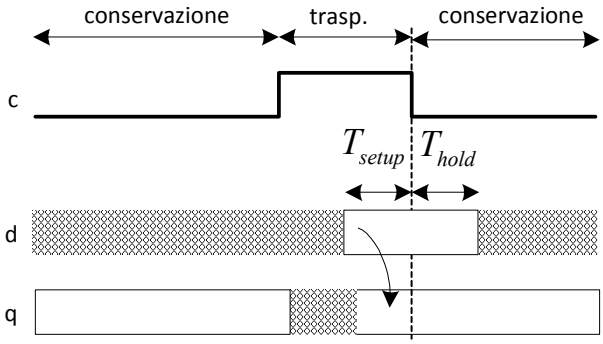
- Quando ***c*=0**, qualunque sia il valore di *d*, il D-latch dovrà mantenere l'uscita costante (è in **conservazione**). Pertanto, darò all'SR un comando di *conservazione* *s*=0, *r*=0.
- Quando ***c*=1**, il D-latch è in **trasparenza,** e quindi l'uscita deve adeguarsi a *d*. Pertanto,
    - Se *d*=0, darò un comando di **reset**, *s*=0, *r*=1
    - Se *d*=1, darò un comando di **set**, *s*=1, *r*=0

Scritta la tabella di verità, si vede abbastanza bene che $s = c \cdot d$, $r = c \cdot \overline{d}$. Si noti che, avendo usato un latch SR come stadio finale, viene gratis che anche il D-latch ha **la variabile di uscita diretta e negata.** Inoltre, posso sfruttare gli ingressi di /PS e/PC del latch SR per inizializzare il D-latch al reset.





Le regole di pilotaggio di questa rete stabiliscono che si debba tenere **d costante a cavallo della transizione di c da 1 a 0**. I tempi per cui deve essere costante (prima e dopo) sono chiamati T_{setup} e T_{hold}, rispettivamente, e sono dati di progetto della rete. Entrambi servono a garantire che la rete non veda **transizioni multiple di ingresso,** e che quindi si stabilizzi in modo prevedibile.



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 balla, e 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.**

In pratica, non si può memorizzare in un D-Latch (né in nessuna rete trasparente) **niente che sia funzione dell'uscita q**, altrimenti potrebbero verificarsi problemi di pilotaggio. Tutte le reti che abbiamo visto finora, ed in particolare:

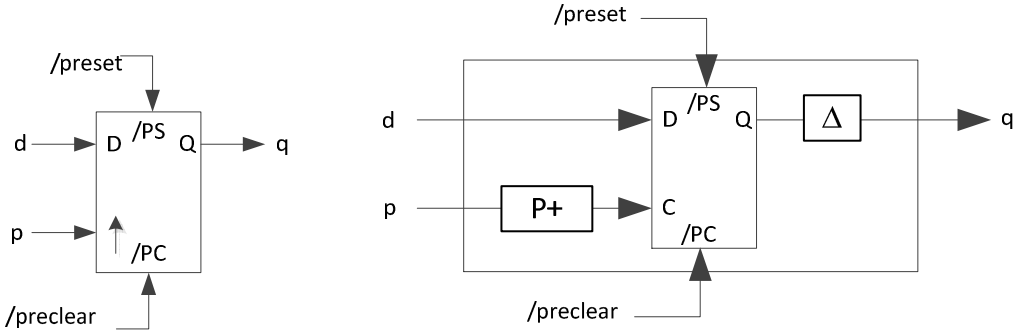
- Reti combinatorie
- latch SR
- D-latch

Sono reti trasparenti.

1.5 Il D flip-flop

Esistono anche reti (ed, in particolare, elementi di memoria) **non trasparenti**. In generale, elementi di memoria trasparenti si chiamano **latch**, mentre quelli non trasparenti si chiamano **flip-flop**. Uno

piuttosto comune si chiama **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 in salita, memorizza *d*, attendi un po' e adegua l'uscita**".



Uno schema **concettuale** per realizzare una rete che si comporti in questo modo è il seguente: si prende un D-latch, e si premette alla variabile *c* un **formatore di impulsi**, in modo tale che, al fronte di salita di *p*, il D-latch **vada brevemente in trasparenza** e memorizzi *d*. Inoltre (**fondamentale**) si **ritarda l'uscita** di un ritardo Δ **maggiore dell'intervallo del P+**. In questo modo, quando *q* cambia adeguandosi a *d*, **la rete non è più in trasparenza, ma in conservazione**.

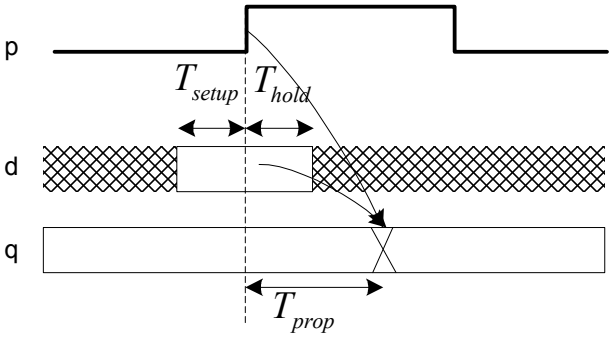
L'uscita *q* viene adeguata al valore campionato di *d* <u>dopo</u> che la rete ha smesso di essere sensibile al valore di *d*.

Il **pilotaggio** del D-FF deve avvenire nel rispetto di alcune regole:

- A cavallo del fronte di salita di *p*, la variabile *d* **deve rimanere costante**. I tempi per cui deve rimanere costante prima e dopo il fronte di salita si chiamano T_{setup}, T_{hold}. I nomi sono gli stessi del D-latch, ma i tempi **non sono gli stessi**, e dipendono da come è progettata la rete. Vedremo più avanti altre implementazioni (commerciali) del D-FF.
- Tra due transizioni in salita della variabile *p* deve passare abbastanza tempo perché l'uscita si possa adeguare.

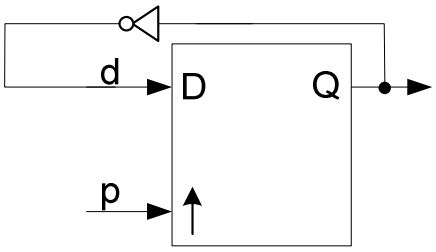
Il ritardo con cui si adegua l'uscita, misurato a partire dal fronte di salita di *p*, si chiama T_{prop}, ed è $\boxed{T_{prop} > T_{hold}}$. Quest'ultima disuguaglianza garantisce che la rete è **non trasparente**.

Il tutto si vede bene con un diagramma di temporizzazione:

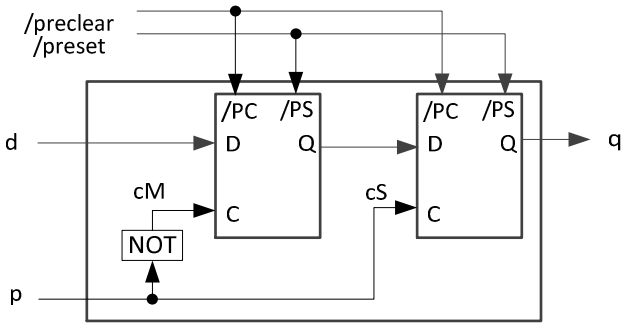


L'uscita di un D-FF **non oscilla mai** (a differenza di quella del D-latch), in quanto viene adeguata in modo **secco** ad un istante ben preciso, e non è mai "direttamente connessa" con l'ingresso *d*. Ciò comporta che si possono montare i D-FF in qualunque modo si voglia, tanto **non succede niente.**

Ad esempio, in questo caso non succede assolutamente niente. Ogni volta che arriva un fronte di salita di *p,* l'uscita cambia valore (con il debito ritardo). Posso mettere in ingresso a *d* **qualunque funzione dell'uscita q**, senza che ci siano problemi di sorta.



Esiste un'altra possibilità per la sintesi del D-FF. Si può usare una struttura **master/slave**, fatta da **due D-latch in cascata**.

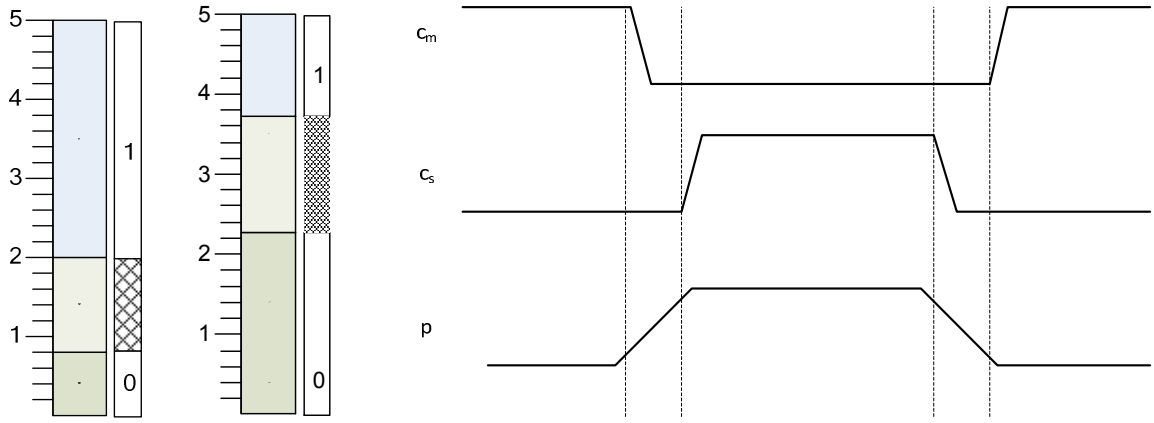


Quando:

- *p*=0, **il master campiona, e lo slave conserva** (quindi non ascolta il proprio ingresso *d*);
- *p*=1, **il master conserva, e lo slave campiona** (quindi insegue l'uscita del master).

Quindi, sul fronte di salita di *p*, il master memorizza l'ultimo valore di *d* che ha letto, e lo slave presenta (con un certo ritardo T_{prop}) quell'ultimo valore come uscita della rete globale. L'adeguamento dell'uscita avviene sul fronte di salita di *p*, ma, più o meno contemporaneamente, il master isola l'uscita dall'ingresso.

In teoria non fa una grinza. Ci possono essere problemi, però, per quanto riguarda il funzionamento **transitorio**. In particolare, può succedere che il master e lo slave siano **contemporaneamente in trasparenza**, anche se per un transitorio. Per evitare questo, normalmente, si agisce per via **elettronica**. Si fa in modo che l'ingresso *c* dello slave riconosca la tensione di ingresso come 1 soltanto quando questa è prossima al fondo scala, e riconosca come 0 un maggior range di tensioni.

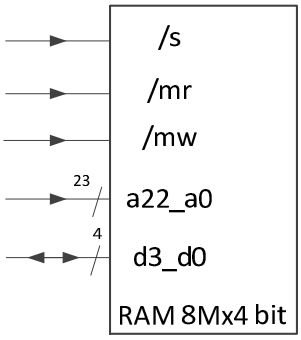


(master) (slave)

In questo modo si riesce a far sì che non siano mai entrambi in trasparenza.

1.6 Le memorie RAM statiche

Le RAM statiche, o S-RAM (esistono anche quelle **dinamiche**, 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**.



Dal punto di vista dell'utente, una memoria è dotata dei seguenti collegamenti:

- un certo numero di **fili di indirizzo**, che sono **ingressi**, in numero sufficiente ad indirizzare tutte le celle della memoria. In questo esempio, la memoria contiene 2^{23} celle di 4 bit, e ci vogliono 23 fili di indirizzo
- un certo numero di **fili di dati**, che sono fili di **ingresso/uscita** (come tali, andranno **forchettati con porte tri-state**, come visto a suo tempo). In questo esempio sono 4.
- Due segnali **attivi bassi** di **memory read e memory write**. Non dovranno mai essere attivi contemporaneamente. Servono a dare il comando di lettura o scrittura della cella il cui indirizzo è trasportato sui fili a22_a0.
- Un segnale (attivo basso) di **select**, che viene attivato quando la memoria è selezionata. Quando /s vale 1, la memoria è insensibile a tutti gli ingressi. Quando vale 0, la memoria è selezionata, e

reagisce agli ingressi (fare il parallelo con **ingresso di enabler in un decoder**). Ciò consente di realizzare una memoria "**grande**" (in termini di n. di locazioni) mettendo insieme più banchi di memoria "**piccoli**". Basta che ne selezioni **uno alla volta**, e poi posso mandare in parallelo tutto il resto (lo vediamo fra un minuto).

Il comportamento della memoria è quindi **deciso da */s*, */mw*, */mr***. Vediamo in che modo:

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

Vediamo adesso come è realizzata una RAM statica.

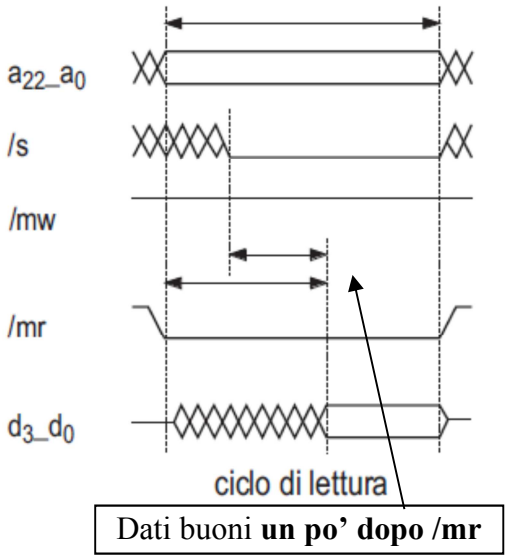
1) Disegnare la matrice di D-latch. Una riga è una **locazione**, bit 0 a destra, bit 3 a sinistra.
2) Le uscite dei D-latch dovranno essere selezionate **una riga alla volta**, per finire sui fili di dati in uscita. Ci vuole un **multiplexer per ogni bit**, in cui
    a. gli ingressi sono le uscite dei D-latch
    b. le variabili di comando sono **i fili di indirizzo**
3) Le uscite di ciascuno dei (4) multiplexer vanno **bloccate** con (4) tri-state. Queste dovranno essere abilitate **quando sto leggendo dalla memoria**. Ci vuole una RC che mi produca l'enable (chiamiamolo **b**) come funzione di */s*, */mw*, */mr*. (disegnare la tabella di verità).
4) Per quanto riguarda gli ingressi: posso portare a ciascuna **colonna** di D-latch i fili di dati sull'ingresso ***d***. Basta che faccia in modo che, quando voglio **scrivere**, **soltanto una riga di d-latch** sia abilitata, cioè abbia ***c* ad 1**. Quindi, ciascuna **riga** di D-latch avrà l'ingresso ***c*** prodotto da un **demultiplexer**, comandato dai fili di indirizzo. Questo demultiplexer **commuterà sulla riga giusta** il comando di lettura, attivando solo una riga di ***c*** alla volta. In questo modo, anche se i fili di dati vanno in ingresso contemporaneamente a tutti i D-latch, solo una riga li sentirà. Il comando di lettura (chiamiamolo **c**) è funzione di */s*, */mw*, */mr*. (disegnare la tabella di verità).

[vedere schema RAM sul libro]

Le RAM statiche sono **molto veloci** (pochi ns di tempo di risposta). Infatti, il loro tempo di attraversamento è quello di **pochi livelli di logica**. Questa tecnologia è usata per realizzare **memorie cache** (le memorie RAM montate nel calcolatore come memoria principale sono RAM **dinamiche**, e sono fatte diversamente).

Descriviamo adesso la **temporizzazione** del ciclo di **lettura** della memoria.

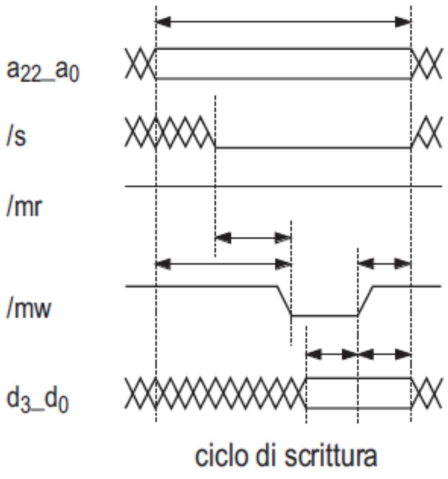
Ad un certo istante, gli indirizzi si stabilizzano al valore della cella che voglio leggere ed arriva il comando di **/mr**. Per motivi che saranno chiari fra un minuto, il comando di **/s** arriva con un po' di ritardo, e **balla** nel frattempo, in quanto è funzione combinatoria di altri bit di indirizzo.



Quando sia */s* che */mr* sono a 0, dopo un pochino le porte tri-state vanno in conduzione, così come i multiplexer sulle uscite vanno a regime. Da quel punto in poi i dati sono buoni, e chi li ha richiesti li può prelevare. Quando */mr* viene ritirato su (il che verrà fatto quando **chi voleva leggere i dati li ha già prelevati**), i dati tornano in alta impedenza. A quel punto gli indirizzi e */s* possono ballare a piacere, tanto non succede niente.

Descriviamo adesso la **temporizzazione** del ciclo di **scrittura** della memoria.

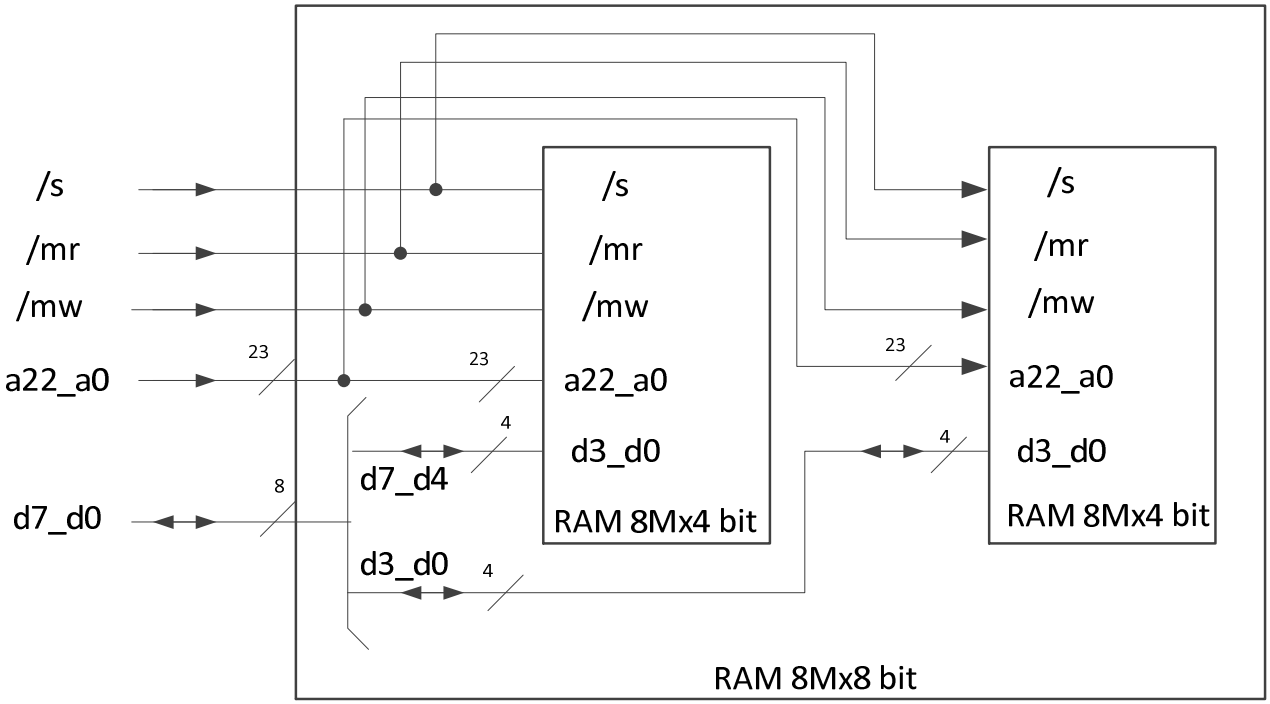
Qui le cose vanno diversamente. Visto che la **scrittura è distruttiva** (in quanto quando scrivo i D-latch sono in trasparenza), devo **attendere che /s e gli indirizzi siano stabili** prima di portare giù ***/mw*.** I dati, invece, possono ballare a piacimento (anche quando */mw* vale 0), ma devono **essere buoni a cavallo del fronte di salita di */mw***. Tale fronte, infatti, corrisponde (con un minimo di ritardo dovuto alla rete C ed al demultiplexer), al fronte di discesa di *c* sui D-latch. [Il tempo per cui devono essere tenuti buoni *dopo* il fronte di salita di ***/mw*** è *maggiore* di T_{hold}, in quanto c'è dell'altra logica davanti.]



1.6.1 Montaggio "in parallelo": raddoppio della capacità di ogni cella

Come si fa ad ottenere una memoria **8Mx8** usando banchi **8Mx4?** È facile. Basta connettere in parallelo tutti quanti i fili ed affastellare i dati.

NB: partire dal bus a sinistra, e disegnare due scatolette con gli ingressi. Connettere dopo.



1.6.2 Montaggio "in serie": raddoppio del n. di locazioni

Come si fa ad ottenere una memoria **16Mx8** usando banchi **8Mx8?** È facile, anche se richiede un po' di logica in più. 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" (a_{23}=0) nell'altro

Quindi, si **genera** il segnale di **select** per i due blocchi usando **il valore di a_{23}.** Se il modulo di memoria che si vuole creare deve essere inserito in uno spazio di indirizzamento più grande, fa comodo poter fornire all'esterno un segnale globale di *select*, che dovrà quindi essere messo in OR con il bit di indirizzo **a_{23}.** Tutto il resto viene portato **in parallelo** ai due blocchi.

[vedere schema sul libro]

NB: Quanto appena scritto giustifica il fatto che il filo di **select** normalmente si stabilizza **con un certo ritardo rispetto al filo di *~~/mr~~*,** perché è comunque funzione di (altri) fili di indirizzo.

Come ultima nota, osserviamo che questi sono **montaggi rigidi**. Sono cioè montaggi in cui si decide direttamente in **fase di progetto** quale deve essere il tipo di accesso (se a 4 o a 8 bit, ad esempio). Se si vuole mantenere **flessibilità**, cioè consentire in tempi diversi di fare accessi a 4 e ad 8 bit (si noti che, nel calcolatore, la memoria può essere letta a **byte, word, dword**) c'è bisogno di **altra logica** oltre quella (poca) che ci abbiamo messo noi. Montaggi del genere li vedrete (forse) nel corso di Calcolatori Elettronici.

1.7 Le memorie Read-only

Le memorie ROM (read-only memory) sono in realtà dei 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, e costituiscono la parte **non volatile** dello spazio di memoria (cioè quella che **mantiene l'informazione in assenza di tensione**).

Possono essere descritte per **semplificazione** delle memorie RAM, togliendo tutta la parte necessaria alla scrittura. Anche se sono reti combinatorie, le loro uscite devono essere supportate da **porte tri-state**, in quanto devono poter coesistere su bus condivisi con altri dispositivi (ad esempio, processore e memorie RAM).

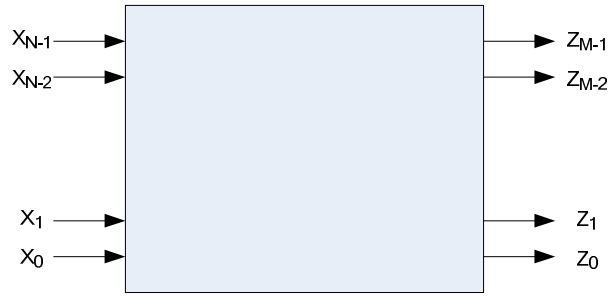
[vedere schema sul libro]

I D-latch sono sostituiti da (qualcosa di logicamente equivalente a) **generatori di costante**, la cui natura dipende dalla tecnologia. Si distinguono PROM, EPROM, EEPROM, a seconda che i generatori di costante possano essere "**programmati**" usando particolari apparecchiature. La programmazione **non può avvenire durante il funzionamento** (altrimenti sarebbero memorie RAM).

Il comando di lettura, unito al comando di select, mette in **conduzione le tri-state**, consentendo ai generatori di costante della riga selezionata dagli indirizzi di inserire sul bus il contenuto della cella.

2 Reti Sequenziali Asincrone

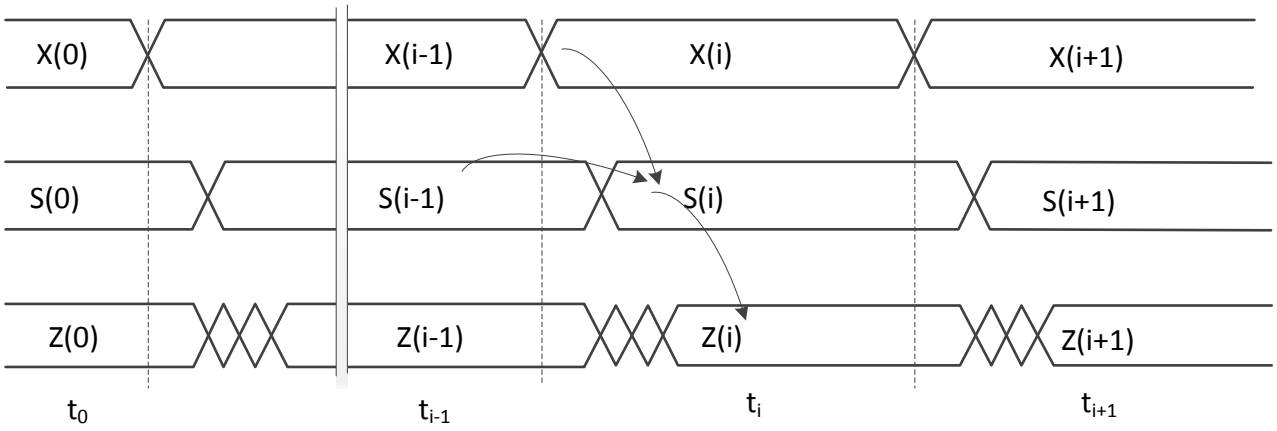
2.1 Modalità di descrizione delle reti sequenziali asincrone



Una rete sequenziale asincrona è rappresentata come segue:

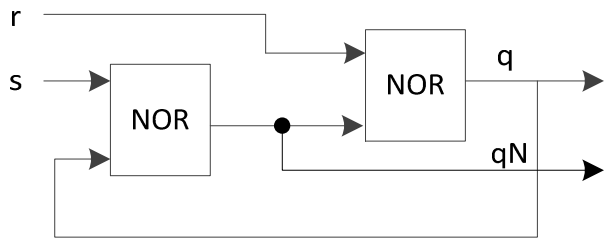
1) un insieme di *N* variabili logiche di ingresso.
2) un insieme di *M* variabili logiche di uscita.
3) Un **meccanismo di marcatura**, che ad ogni istante marca uno **stato interno presente**, scelto tra un insieme **finito** di K stati interni $\mathbf{S} \equiv \{S_0, ..., S_{K-1}\}$.
4) Una **legge di corrispondenza** del tipo $A : \mathbf{X} \times \mathbf{S} \to \mathbf{S}$, che mappa quindi una coppia (stato di ingresso, stato interno) in un nuovo stato interno.
5) Una **legge di corrispondenza** del tipo $B : \mathbf{S} \to \mathbf{Z}$, che individua lo stato di uscita a partire dallo stato interno. (Nota: tale legge avrebbe potuto essere più generale, del tipo $B : \mathbf{X} \times \mathbf{S} \to \mathbf{Z}$. Tale maggior generalità non è giustificata da nessun caso pratico di interesse).
6) Una rete sequenziale asincrona, essendo appunto **asincrona**, evolve in modo **continuo** nel tempo. Entrambe le leggi A e B sopra menzionate sono attive **continuamente**.

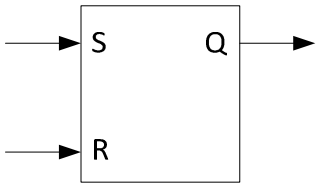
Le reti sequenziali asincrone, quindi, si evolvono quando **cambiano gli ingressi**: un cambiamento di ingresso, tramite la legge A, genera un nuovo stato interno (almeno, poi vedremo). Un nuovo stato interno genera un nuovo stato di uscita.



Visto che lo stato di uscita *i*-simo dipende dallo stato interno *i*-simo e questo dipende dallo stato di ingresso (*i*-1)-simo e dallo stato interno (*i*-1)-simo, e così via, è chiaro che lo stato di uscita *Z*(*i*) dipende da **tutti gli stati di ingresso** succedutisi dall'accensione fino a quel momento, e dallo **stato interno iniziale S(0)** impostato al reset. Ciò giustifica l'appellativo di rete **sequenziale**.

2.1.1 Esempio di rete sequenziale asincrona: il latch SR





$\mathbf{X} = \{00, 01, 10, \underline{11}\}$, $\mathbf{Z} = \{0,1\}$. Lo stato di ingresso 11 **non si deve presentare**, perché corrisponde ad un comando contraddittorio. Gli **stati interni** sono **due**, e li chiamiamo ***S*0, *S*1**. $\mathbf{S} = \{S0, S1\}$. Il fatto che ci siano <u>due</u> possibili stati interni è visibile dal fatto che c'è **una variabile di anello** (un solo filo che ritorna indietro). Come avete visto, la **funzione di memoria è associata alla presenza di anelli.** Siccome gli anelli sono variabili logiche, se una rete ha W variabili logiche di anello, il valore di queste W variabili definirà 2^W possibili **stati interni** che la rete può memorizzare. Il valore di queste variabili ad un certo istante definisce lo stato interno a quell'istante. Se volete, una **rete combinatoria** (che, come sappiamo non ha anelli), può essere vista come un caso degenere di rete sequenziale asincrona che ha **0 variabili d'anello, e quindi un solo stato interno possibile** (quindi, di fatto, non ha memoria). Infatti, le due leggi: $A: \mathbf{X} \times \mathbf{S} \to \mathbf{S}$, $B: \mathbf{X} \times \mathbf{S} \to \mathbf{Z}$, se ***S*** è un insieme con **un solo stato**, si riducono a:

- A: una legge **costante;**
- B: una legge da **ingresso a uscita.**

Il che, come sappiamo, descrive bene una rete combinatoria. Associamo:

- al nome ***S*0,** lo stato in cui il latch SR **memorizza il bit 0** (e lo presenta in uscita).
- al nome ***S*1,** lo stato in cui il latch SR **memorizza il bit 1** (e lo presenta in uscita).

q	q'	s	r
0	0	0	-
0	1	1	0
1	0	0	1
1	1	-	0

Abbiamo già visto che un latch SR si può descrivere tramite la **tabella di applicazione.**

Ora, posso osservare che, per quanto appena detto, la prima riga della tabella si legge come:

"se sono nello stato *S*0 e l'ingresso s è a 0, rimango nello stato *S*0 (qualunque cosa faccia r)". Iterando questi ragionamenti, posso costruire una **tabella di flusso,** che è il modo più usato per descrivere le possibili evoluzioni di una rete sequenziale asincrona. Questa altro non è che una **versione tabulare** delle leggi A e B descritte prima.

sr	00	01	11	10	q
S0	(S0)	(S0)	-	S1	0
S1	(S1)	S0	-	(S1)	1

La parte **sinistra della tabella** descrive la legge A, quella che produce il nuovo stato interno a partire dagli ingressi e dallo stato interno. La parte **destra della tabella** descrive la legge B, che produce le uscite sulla base dello stato interno presente.

Una RSA **evolve (cioè modifica il proprio stato interno e/o la propria uscita) a seguito di cambiamenti dello stato di ingresso**. Non è difficile vedere che, se mi trovo nello stato (ad esempio) ***S*0** con ingresso 00, la rete **non evolve** (a meno che non cambi lo stato di ingresso). L'uscita rimane costante a 0, la variabile d'anello mantiene costante il proprio valore. Si dice, in questo caso, che **la rete ha raggiunto la stabilità (è a regime),** oppure che **lo stato interno S0 è stabile con stato di ingresso 00**. In tutti i casi in cui ciò succede, si mette **un cerchio** intorno allo stato interno stabile (è **errore** non metterlo).

Noi siamo interessati alle RSA che, partendo da un qualunque stato interno, riescono **sempre** a raggiungere la stabilità, purché lo stato di ingresso sia mantenuto costante sufficientemente a lungo. Il latch SR ha questa proprietà. Se, ad esempio, avessi avuto una rete descritta dalla seguente tabella di flusso:

sr	00	01	11	10	q
S0	(S0)	(S0)	-	S1	0
S1	(S1)	S0	-	S0	1

(rete che per ora **non so realizzare** – vedremo più in là come si fa) avrei che, se capita l'ingresso 10, la rete oscilla indefinitamente tra i due stati S0 e S1, e l'uscita oscilla di conseguenza. Reti di questo genere **non ci interessano**.

D'ora in avanti, nel parlare delle RSA, supporremo che:

1) lo stato di ingresso venga mantenuto **per tutto il tempo che ci vuole a raggiungere la stabilità (pilotaggio in modo fondamentale)**.
2) Lo stato di ingresso venga cambiato **per un bit alla volta**, cioè stati di ingresso **consecutivi** siano anche **adiacenti**.

Si noti che una legge simile a quella 1) l'abbiamo enunciata per le reti combinatorie. Mentre per queste ultime, però, eravamo in grado di dare una caratterizzazione **quantitativa** semplice ("lo stato di ingresso deve essere mantenuto per un tempo maggiore del **tempo di attraversamento** della rete"), per reti sequenziali asincrone **non sono ancora in grado** di dare una caratterizzazione quantitativa. Vedremo più tardi come ci si arriva.

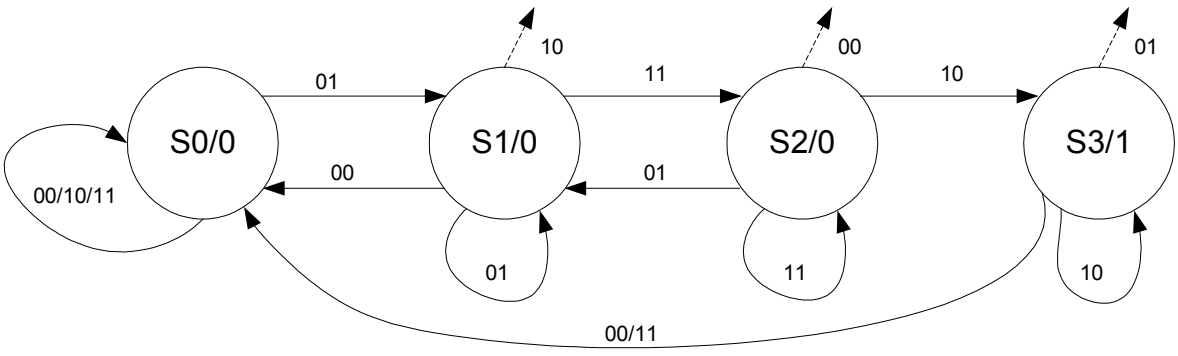
2.1.2 Esempio: riconoscitore della sequenza 01, 11, 10

Un riconoscitore di sequenza è una rete che ha N ingressi (in questo caso N=2, sono sequenze di 2 bit), ed un'uscita. A parole, l'evoluzione della rete è la seguente:

"l'uscita è ad 1 soltanto quando si è presentata la sequenza degli stati di ingresso voluta (01,11,10), ed è a zero altrimenti"

È, se vogliamo, la rete che sblocca la serratura di una cassaforte in cui la combinazione è data da una sequenza di parole di N bit. È ovvio che è una rete **con memoria**, in quanto deve ricordare una data sequenza di stati di ingresso. In particolare, ciò che va memorizzato è **il numero di passi di sequenza corretti consecutivi** visti finora, quindi sono richiesti ***K*+1 stati interni per una sequenza di *K* stati di ingresso da riconoscere** (ciascuno stato interno corrispondente a 0, 1, …, K passi riconosciuti). L'ultimo di questi stati avrà un'uscita ad 1, gli altri a 0.

S0: Supponiamo che al reset iniziale la rete si trovi in uno **stato interno iniziale S0** (vedremo più avanti come si fa a imporre uno stato interno iniziale), che corrisponde alla nozione che non è ancora iniziata nessuna sequenza corretta. Ovviamente, in questo stato l'uscita dovrà valere 0, in quanto non devo sbloccare la cassaforte finché non ho ricevuto tutta la sequenza.



Dallo stato iniziale non esco finché non ho visto in ingresso il primo passo **corretto** della sequenza che voglio riconoscere, cioè 01. Quando arriva 01, **devo cambiare stato**, perché devo memorizzare

che la sequenza è cominciata. Se in S0 memorizzavo il fatto che la sequenza non era ancora iniziata, adesso mi ci vuole un altro stato. Chiamiamolo S1.

S1: In S1 devo restare **finché l'ingresso non cambia nuovamente**. Per tutto il tempo che permane 01 come stato di ingresso, la rete non deve fare niente. S1 è stabile con ingresso 01. Quando cambia lo stato di ingresso, devo **per forza** prendere una decisione.

- Se il nuovo stato di ingresso è 11, vuol dire che ho ricevuto **due passi di sequenza corretta**, e devo memorizzare questo nuovo avvenimento. Mi serve un **nuovo stato interno**, perché gli altri due che ho usato memorizzavano "0 passi corretti" (S0) e "1 passo corretto" (S1).
- Lo stato di ingresso potrebbe anche diventare 00, nel qual caso si deve ripartire da capo, tornando in S0.
- Lo stato di ingresso **non può diventare 10**. Sono arrivato in S1 con lo stato di ingresso 01. Ci vuole un arco verso "infinito".

S2: memorizza il fatto che ho ricevuto **due passi corretti** di sequenza. Ci sono arrivato con ingresso 11, e finché resta 11 in ingresso resto in S2 (orecchio). Per lo stesso motivo di prima, ci vuole un arco verso infinito per l'ingresso 00. Gli stati di ingresso 10 e 01 mi porteranno **fuori** da S2 necessariamente. Se arriva 01, la sequenza corrente **è errata**, ma **potrebbe esserne cominciata una nuova,** e quindi devo andare in S1. Se invece arriva 10, **ho terminato la sequenza corretta**, e quindi devo andare in uno stato **diverso dai precedenti**.

S3: in questo stato **dovrò porre l'uscita ad 1**, perché devo sbloccare la cassaforte. Resto in questo stato fintanto che si mantiene l'ingresso 10. L'ingresso 01 non è possibile (arco all'infinito). Qualunque altra cosa arrivi **devo ripartire da S0**, in quanto non può essere l'inizio di una sequenza corretta.

In maniera automatica, posso costruire la *tabella* di flusso dal *grafo* di flusso. Le due descrizioni sono **equivalenti**, ma alcune proprietà si vedono meglio su una, altre sull'altra.

x_1x_0	00	01	11	10	z
S0	(S0)	S1	(S0)	(S0)	0
S1	S0	(S1)	S2	--	0
S2	--	S1	(S2)	S3	0
S3	S0	--	S0	(S3)	1

Gli archi verso "infinito" corrispondono a **stati interni successivi non specificati** nella tabella di flusso

Si noti che:

- ogni **colonna** contiene uno stato cerchiato (almeno uno). Ciò è condizione **necessaria** (ma non sufficiente) per la stabilità. La condizione non è sufficiente perché posso sempre mettere, oltre ad uno stato stabile, anche una coppia di stati che oscillano (e.g. S1 al posto di S2 in 11/S2).
- La legge A ha una caratteristica particolare. Partendo da uno stato stabile e variando lo stato di ingresso, si raggiunge **sempre uno stato stabile in al più un passo**. Una legge di questo tipo si dice **normale** (ed è un concetto importante). Più in dettaglio, una legge è normale se e solo se:

$$\boxed{\forall S, X, \quad \left[A(S,X)=S\right] \vee \left[A(S,X)=A\left(A(S,X),X\right)\right]}$$

  Cioè, una legge è normale se, dato uno stato di ingresso X ed uno stato interno S, S è stabile con X, oppure se S non è stabile con X, ma lo stato in cui si va a finire è stabile con X. C'è al massimo **una transizione di stato interno**.

Osservazione (importante): Non è possibile, con una RSA, descrivere un riconoscitore di sequenze **qualunque**. Ad esempio, questi due non possono essere sintetizzati come RSA:
- **01, 10, 11** (due variabili di ingresso non possono cambiare contemporaneamente)
- **01, 01, 11** (la rete si **evolve solo** in seguito alle variazioni dello stato di ingresso).

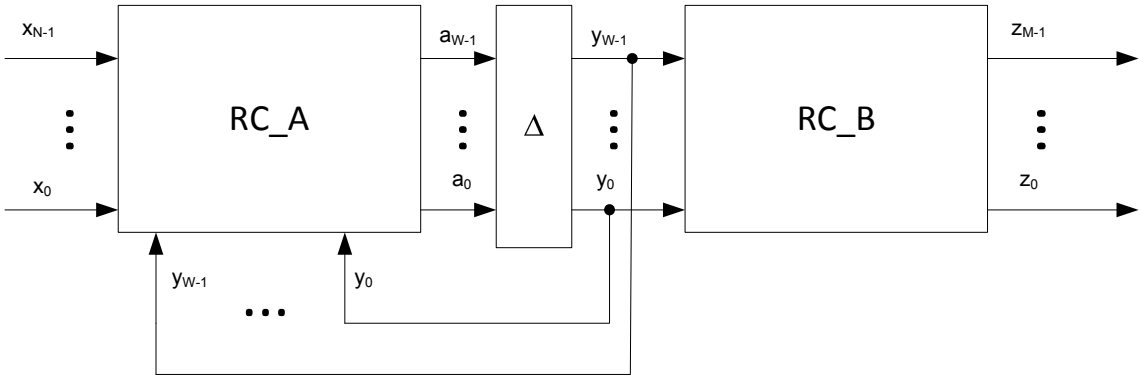
2.2 Modelli strutturali per la sintesi di reti sequenziali asincrone

Abbiamo visto come si **descrivono** le reti sequenziali, vediamo come si **sintetizzano**, cioè come si ottiene una loro realizzazione in termini di porte logiche. Introduciamo un modello generale, e poi vediamo la sintesi delle reti che abbiamo descritto.

Il punto di partenza sono le leggi A e B descritte in precedenza. Andiamo a guardare come sono fatte:

1) Una **legge di evoluzione nel tempo** del tipo $A: \mathbf{X} \times \mathbf{S} \to \mathbf{S}$, che mappa quindi una coppia (stato di ingresso, stato interno) in un nuovo stato interno.
2) Una **legge di evoluzione nel tempo** del tipo $B: \mathbf{S} \to \mathbf{Z}$, che decide lo stato di uscita basandosi sullo stato interno.

Per poter sintetizzare una RSA è necessario **codificare gli stati interni come insiemi di variabili logiche**, che chiameremo $y_0, ..., y_{W-1}$ (variabili di stato).



La legge B, quindi, può essere implementata con una **rete combinatoria** con W ingressi ed M uscite, e la legge A richiede una rete combinatoria con N+W ingressi e W uscite. Nel mezzo, c'è un **meccanismo di marcatura**, che per adesso è semplicemente un **ritardo**. Vedremo poi **se e quando** questo ritardo possa essere eliminato. Per questo motivo (e per chiarezza sintattica), chiamo in maniera diversa le variabili che codificano

- **lo stato interno presente** $y_0,...,y_{W-1}$
- **lo stato interno successivo** $a_0,...,a_{W-1}$

Lo stato interno successivo prodotto dalla rete RC_A, in funzione di quello presente e degli ingressi, **diventa stato interno presente** dopo un ritardo pari a delta.

È necessario osservare quanto segue:

Una RSA si evolve in modo prevedibile **soltanto** quando la rete RC_A (quella che implementa la legge A) **non produce <u>mai</u> stati di uscita spuri**.

Affinché non si producano stati di uscita spuri in una rete combinatoria (RC_A) sono necessarie tre condizioni:

1) RC_A deve essere priva di alee.
2) RC_A deve essere **pilotata in modo fondamentale**: i suoi ingressi devono cambiare soltanto quando essa è a regime.
3) Gli stati di ingresso di RC_A devono essere variati **un bit alla volta** (stati di ingresso a RC_A consecutivi devono essere adiacenti).

Il problema 1) è di facile soluzione: basta implementare RC_A priva di alee, eventualmente inserendo sottocubi ridondanti nella sintesi. I problemi 2) e 3) sono invece più complessi, perché RC_A ha **sia ingressi che vengono pilotati dall'esterno** (le variabili di ingresso x_0, x_{n-1}) che **ingressi sui quali non posso dare vincoli di pilotaggio** (le variabili di stato $y_0,...,y_{W-1}$). Pertanto, dovremo

esercitare parecchia cura per far sì che gli ingressi della rete RC_A varino un bit alla volta, e soltanto quando la rete RC_A è a regime.

2.2.1 Alee in uscita da RC_A

Vediamo cosa succede **quando si verificano alee sulle uscite** di RC_A. Prendiamo ad esempio il caso **del D-latch**, e scegliamo come codifica degli stati interni S0=0, S1=1. Dalla tabella di flusso mi ricavo banalmente una mappa di Karnaugh:

d	c=0		c=1		q
	0	1	0	1	
S0	S0	S0	S0	S1	0
S1	S1	S1	S0	S1	1

y	cd 00	01	11	10	q
0	0	0	1	0	0
1	1	1	1	0	1

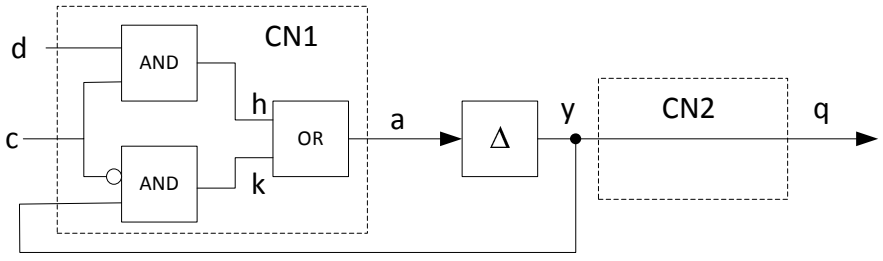
Ciò significa che posso implementare le leggi A e B in termini di **reti combinatorie**. In particolare:

- la rete RC_B che implementa la legge B avrà come ingresso le **variabili di stato** e come uscita le **variabili di uscita** della RSA. In questo caso sarà un cortocircuito, visto come abbiamo scelto la codifica.
- la rete RC_A che implementa legge A avrà come ingresso le **variabili di ingresso e le variabili di stato** e come uscita le **variabili di stato** della RSA. Ciò implica necessariamente che ci siano degli **anelli nella realizzazione della rete**, perché delle variabili di uscita per una rete combinatoria devono anche rientrare in ingresso.

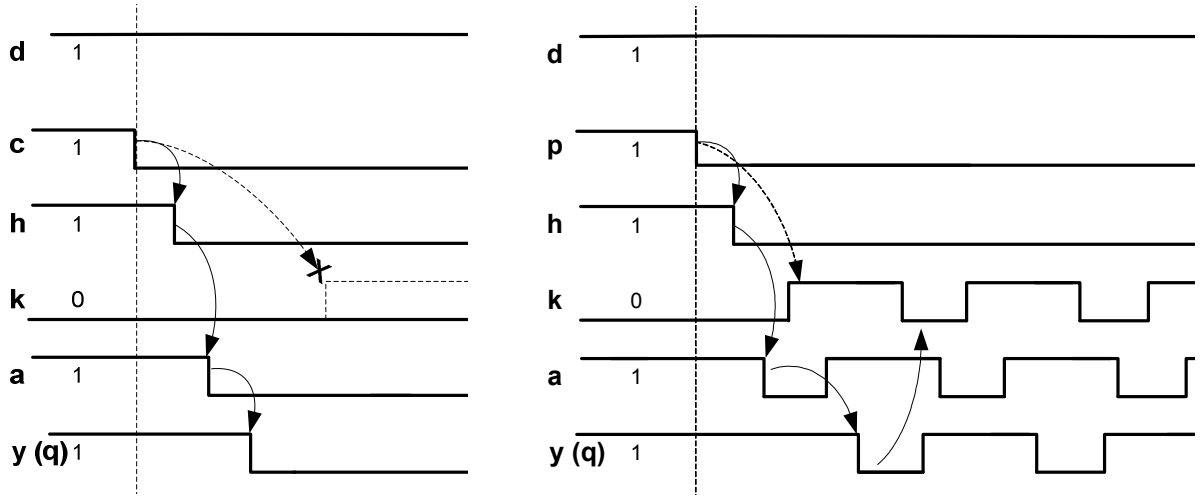
Per il D-latch, abbiamo che RC_B è un cortocircuito, e RC_A si ottiene dalla sintesi (ad esempio) SP a costo minimo della mappa di Karnaugh scritta sopra:

$$a = c \cdot d + y \cdot \bar{c}$$

La sintesi che abbiamo appena scritto **è affetta da alee del prim'ordine** (statiche sul livello 1, essendo SP a 2 LL). Quest'alea si può verificare quando ***d*=1, *y*=1 e *c* passa da 1 a 0** (o viceversa). Un'alea è sempre una cosa da evitare in generale. In questo caso vediamo che succede alla rete se si verifica l'alea.



Ho evidenziato con ***h*** **e** ***k*** i due ingressi all'OR. Facciamo un diagramma temporale:



Supponiamo che il ritardo Δ sia rispettivamente "abbastanza piccolo" (a sinistra), o "piuttosto grande" (a destra). La transizione di *c* dà luogo ad un'alea sull'uscita ***a*** di RC_A, che produce uno **stato interno da marcare spurio** pari a 0. Se questo stato diventa **stato interno marcato**, cioè se *y* registra la variazione di ***a*** prima che ***k* sia tornato ad 1**, allora in ingresso all'AND in basso si ha lo stato 10, che mantiene ***k* fisso a 0**. Allora anche *a* non si può più muovere, e lo stato marcato resta 0, così come l'uscita. Si dice in tal caso che lo stato spurio **si <u>autosostiene</u>**, cioè diventa **permanente**. Nel caso a destra, addirittura, **la rete oscilla** all'infinito (finché, almeno, non si cambiano gli ingressi).

Abbiamo appena dimostrato come la **presenza di alee su RC_A può indurre una RSA ad evolversi in maniera del tutto casuale.**

<u>REGOLA DI PROGETTO N.1:</u> non devono verificarsi alee in uscita da RC_A. Una condizione sufficiente perché ciò succeda è che RC_A sia **priva di alee del prim'ordine.**

Non è un problema, perché siamo sempre in grado di sintetizzare una rete combinatoria a 2 LL in modo che sia priva di alee del primo ordine. Quindi, nel caso del D-Latch, deve essere:

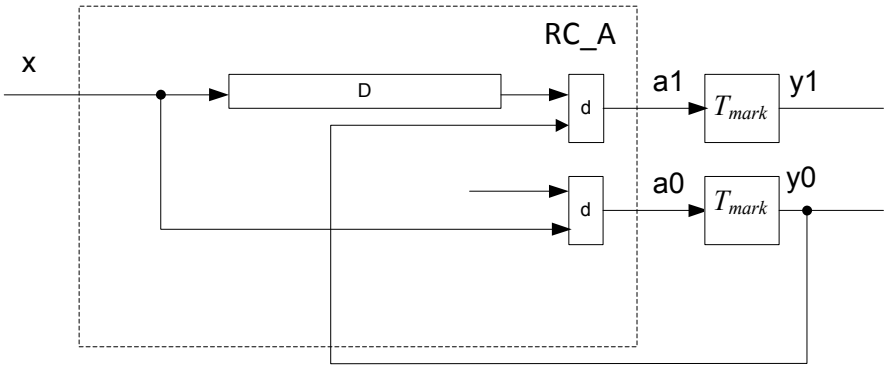
$$a = c \cdot d + y \cdot \bar{c} + y \cdot d$$

2.2.2 Ritardi di marcatura ed alee essenziali

Abbiamo detto, in maniera piuttosto vaga, che il meccanismo di marcatura **è un ritardo.** La presenza del ritardo deriva dal fatto che l**a rete combinatoria RC_A deve essere pilotata in modo fondamentale.** Il problema è che gli ingressi di RC_A **non** sono solo le variabili di ingresso della RSA, ma anche le variabili di stato che rientrano. Così come, per un corretto pilotaggio, evito di variare gli ingressi finché la rete combinatoria non è a regime, dovrò fare anche in modo che le **variazioni delle variabili di stato (che sono ingressi per RC_A) avvengano soltanto quando la rete è a regime**. Questo si fa, in generale, **ritardandole opportunamente.**

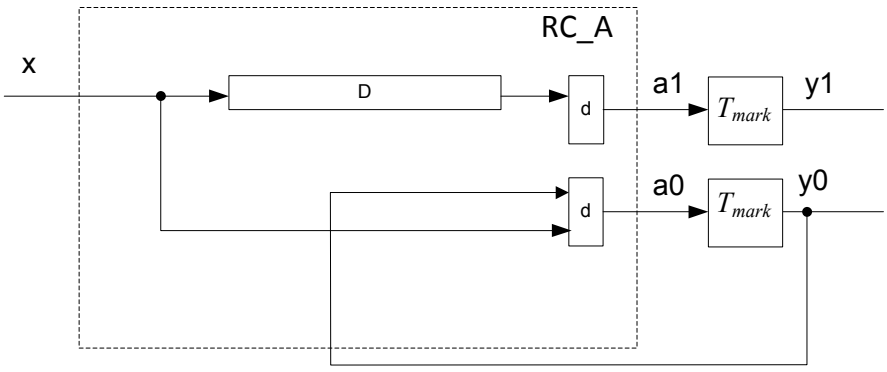
Vediamo se e quando il ritardo sugli elementi di marcatura è necessario.

Prendiamo un esempio semplice. In questo esempio la **dimensione orizzontale delle porte** (non specificate, in quanto non mi interessa sapere cosa fa questa rete) è **proporzionale al loro ritardo**.



Per questa rete, abbiamo $T_A \geq D + d$. Supponiamo $D > 2d + T_{mark}$, cioè che Δ sia un ritardo **piccolo.** In questo caso, y_0 viene prodotta "presto", e torna in ingresso alla parte di rete che produce a_1 **quando questa non è a regime,** cioè quando questa **non ha ancora assorbito la variazione di *x***. Ciò comporta che **non sono in grado di prevedere come evolverà** la RSA. Condizione **sufficiente** perché ciò non avvenga è che $T_{mark} \geq T_A$. In questo caso il ritardo è necessario. Si dice che, in questo caso, la rete è **affetta da alee essenziali**. Il termine "alea essenziale" non ha **niente a che vedere** con le alee delle reti combinatorie a cui siamo abituati.

In quest'altro esempio, invece, quando y_0 rientra, la parte di rete su cui è reazionata **è necessariamente già a regime**. Quindi il ritardo non mi serve a niente, anzi mi dà fastidio perché rallenta la rete.



Si deriva quindi la seguente regola di progetto:

REGOLA DI PROGETTO N.2: sia T_A il tempo di attraversamento di RC_A. Se la rete non è affetta da alee essenziali, il ritardo del meccanismo di marcatura può essere reso nullo. Altrimenti il meccanismo di marcatura deve introdurre un ritardo T_{mark}**, maggiore o uguale a** T_A.

Come si fa a discriminare i due casi? C'è un teorema che mi garantisce che posso vedere quale delle due situazioni si verifichi **direttamente guardando la tabella di flusso.** Torniamo per un attimo alla tabella di flusso del riconoscitore di sequenza visto prima.

x_1x_0	00	01	11	10	z
S0	(S0)	S1	(S0)	(S0)	0
S1	S0	(S1)	S2	--	0
S2	--	S1	(S2)	S3	0
S3	S0	--	S0	(S3)	1

Si **guarda la tabella di flusso, e, se questa è <u>normale</u>,** (perché il teorema si applica <u>solo</u> alle reti normali) **si verifica se:**

"partendo da uno stato stabile (cerchiato) e variando di un bit lo stato di ingresso, si finisce in uno stato stabile S, e tale stato stabile S è lo stesso nel quale si finisce variando altre due volte lo <u>stesso</u> bit di ingresso."
Se ciò è vero:

a) a partire da tutti gli stati cerchiati

b) per ogni possibile variazione (di un bit) dello stato ingresso

allora la rete è **priva di alee essenziali,** e posso eliminare il ritardo. Altrimenti, la rete è **affetta da alee essenziali, e non posso eliminare il ritardo**.

Il teorema di sopra si chiama Teorema di Unger, dal suo scopritore. Fare alcuni esempi. In particolare, mostrare che partendo da S2/11 e variando x0, si finisce in S3 o in S0. Questa rete è affetta da alee essenziali, e quindi **non posso togliere i ritardi**. Se lo facessi, la rete **potrebbe evolvere in modo del tutto casuale** in seguito a certe variazioni dello stato di ingresso.

Riassumendo: questo riconoscitore di sequenza ha una tabella di flusso **normale**, ed è **affetto da alee essenziali**. Detto T_A il tempo di attraversamento della sua rete RC_A (che sintetizzerò a 2LL, priva di alee del primo ordine), devo:

a) prevedere un meccanismo di marcatura (un elemento neutro) con un ritardo $T_{mark} \geq T_A$;

b) pilotare la rete con stati di ingresso adiacenti, distanziati tra loro di $T \geq T_A + \left(T_{mark} + T_A\right)$.

Cerchiamo di dare **un senso intuitivo** al teorema che ci consente di capire se la rete è soggetta ad alee essenziali.

1) Una RSA evolve **solo in seguito a transizioni dello stato di ingresso.**

2) Una transizione dello stato di ingresso **può provocare transizioni dello stato interno**

Quando mi muovo su una tabella di flusso **normale**, per descrivere l'evoluzione della rete a seguito di una variazione dello stato di ingresso a partire da uno stato interno stabile, faccio in genere **due movimenti:** uno **orizzontale**, il primo, corrispondente alla transizione dello stato di ingresso, ed (eventualmente) uno **verticale**, il **secondo**, corrispondente alla transizione dello stato interno che ne consegue. Se le variabili di stato **rientrano troppo velocemente** (il che può succedere se non metto ritardi), può darsi che rientrino **prima che la variazione dello stato di ingresso** sia stata "digerita" da tutta la rete. In questo caso, è **come se mi muovessi <u>prima in verticale e poi in orizzontale</u>** sulla tabella, perché – ai fini della produzione del nuovo stato interno da marcare – varia **prima** la variabile di stato e **poi** quella di ingresso (fare riferimento all'esempio scritto precedentemente).

Con un po' di pazienza si capisce che il Teorema di Unger controlla cosa succeda invertendo l'ordine di transizione tra la variabile di ingresso (che dovrebbe variare per prima) e quella di stato (che dovrebbe variare per seconda). Possono darsi due casi:

1) caso fortunato: quale che sia l'ordine di variazione: (►▼, oppure ▼►), **la rete si stabilizza sempre nello stesso stato interno.**
2) caso sfortunato: a seconda di come mi muovo sulla tabella mi stabilizzo in due stati interni diversi.

Nel **primo caso**, anche se non metto ritardi sulle variabili di stato non ci sono problemi. Nel secondo invece **devo introdurre ritardi**, altrimenti la rete si evolve in modo non predicibile.

Dalla regola di progetto appena enunciata si deduce subito la seguente:

<u>REGOLA DI PILOTAGGIO N.1:</u> lo stato di ingresso di una RSA deve restare costante per un tempo non inferiore a:

$$T = T_A + i \cdot \left(T_{mark} + T_A \right)$$

Dove *i* è il **massimo numero di stati interni che si devono succedere prima che la rete vada a finire in uno stato stabile**.

(regola del **pilotaggio in modo fondamentale**).

Il numero *i* può essere facilmente desunto dall'ispezione della tabella di flusso. Si guarda **colonna per colonna**, e si vede quanti passi ci vogliono, al massimo, per arrivare ad uno stato cerchiato. Se la tabella di flusso è **normale**, allora ***i* vale 1,** in quanto partendo da ogni stato interno stabile, al variare dell'ingresso, o rimango in tale stato, oppure vado a finire in un nuovo stato stabile.

Per completezza, ricordiamo anche:

REGOLA DI PILOTAGGIO N.2: lo **stato di ingresso** di una RSA deve variare di un bit alla volta (stati di ingresso consecutivi devono essere adiacenti).

Svolgiamo adesso la **sintesi** del riconoscitore di sequenza descritto precedentemente.

x_1x_0	00	01	11	10	z
S0	(S0)	S1	(S0)	(S0)	0
S1	S0	(S1)	S2	--	0
S2	--	S1	(S2)	S3	0
S3	S0	--	S0	(S3)	1

Per sintetizzare una RSA è necessario **scegliere una codifica degli stati interni**. Scegliamo quella che ci viene più immediata (vedremo fra un po' che è necessaria qualche **cautela** nel farlo):

S0=00, S1=01, S2=11, S3=10.

Ciò mi consente di **sintetizzare le leggi combinatorie A e B**, tramite le due reti RC_A e RC_B. Partiamo da RC_B, che è più semplice:

y1	y0	z
0	0	0
0	1	0
1	0	1
1	1	0

$$z = y_1 \cdot \overline{y_0}$$

Dalla tabella di flusso posso costruire, in maniera pedissequa, la **tabella delle transizioni**, che dà il valore di a_1a_0 in funzione di x_1x_0, y_1y_0.

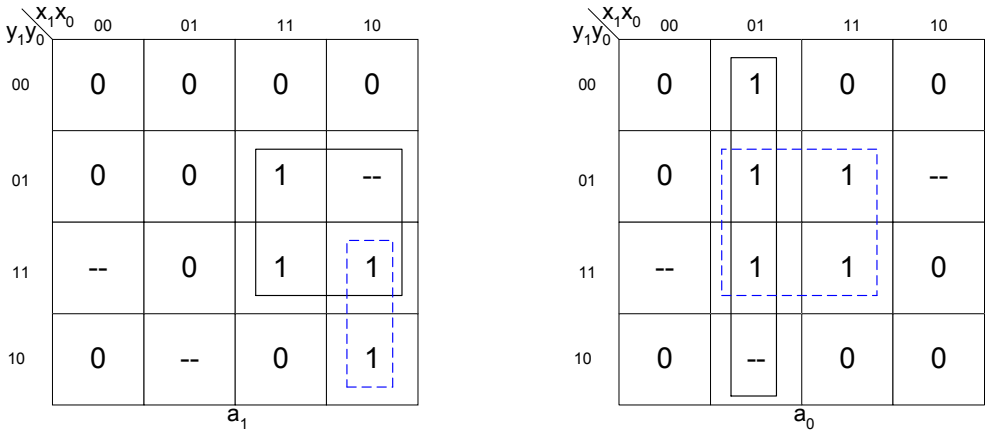
NB: in una tabella di flusso non esiste l'obbligo di avere **stati di ingresso adiacenti in colonne consecutive**. Idem per una tabella delle transizioni. Però posso sempre permutare le righe e le colonne di quest'ultima in modo da ottenere una **mappa di Karnaugh**, che so sintetizzare.

y_1y_0 \ x_1x_0	00	01	11	10
00	00	01	00	00
01	00	01	11	--
11	--	01	11	10
10	00	--	00	10

a_1a_0

Tabella delle transizioni

Dividendo quest'ultima, faccio la sintesi (ad esempio SP) delle due reti combinatorie ad 1 uscita (che è quello che so fare) che producono, rispettivamente, a_1 ed a_0.



$$a_1 = y_0 \cdot x_1 + y_1 \cdot x_1 \cdot \overline{x_0}$$
$$a_0 = \overline{x_1} \cdot x_0 + y_0 \cdot x_0$$

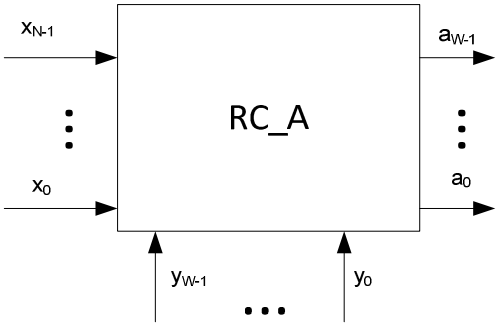
Entrambe le sintesi sono prive di alee, come deve essere.

2.2.3 Codifica degli stati interni e corse delle variabili di stato

Nel precedente esempio abbiamo dato per scontata la **codifica degli stati interni**. Il problema di come codificare gli stati interni richiede però una certa attenzione. Infatti, le variabili di stato sono ingressi per RC_A, e dobbiamo verificare che **gli stati di ingresso di RC_A cambino di un bit alla volta**. Questo si fa:

1) Cambiando un bit alla volta delle variabili di ingresso, e solo quando la RSA è a regime

2) Facendo sì che **stati interni che possono essere consecutivi abbiano codifiche adiacenti**



Se questo non è vero, ci sono due problemi:

1) si possono avere in ingresso alla rete RC_A dei fenomeni di transizione multipla dei suoi ingressi (in particolare, delle variabili $y_0,...,y_{W-1}$).

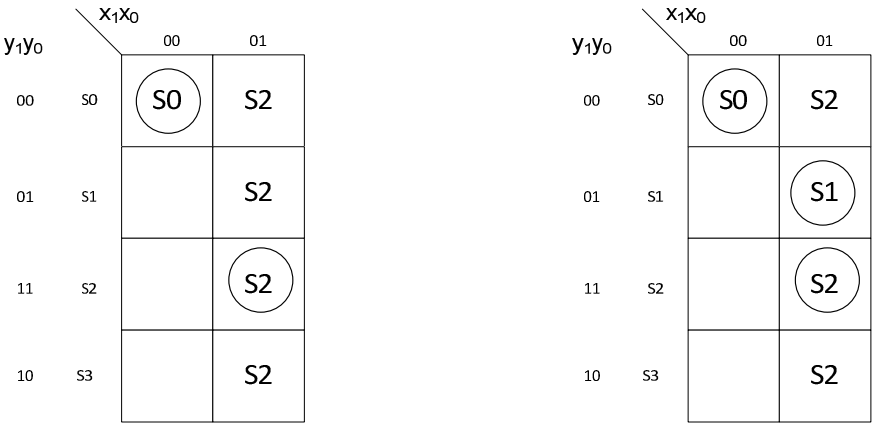
2) In conseguenza di questo, si possono presentare **in uscita dalla rete RC_A degli stati spuri**. Questa cosa va evitata, perché, a causa della retroazione, gli stati interni spuri potrebbero far evolvere la rete in modo del tutto differente da quello previsto. Tale fenomeno si chiama **corsa delle variabili di stato**, e va evitato assolutamente.

Supponiamo infatti di avere due stati interni, S0, S2, che possono essere consecutivi (ad esempio, perché c'è un arco dall'uno all'altro nel grafo di flusso), codificati come 00 e 11. Quando si presenta lo stato di ingresso che provoca la transizione tra S0 e S2, per un tempo non nullo gli ingressi y1y0 di RC_A assumeranno il valore 01 oppure 10. In forza di questo possono succedere due cose:

- La rete si **stabilizza più tardi**, attraverso due o più transizioni di stato, nello stato dove **volevo che si stabilizzasse**. È un caso fortunato, di **corsa non critica**.
- La rete si **evolve in maniera del tutto differente**, raggiungendo un punto di stabilità diverso (**corse critiche**).

Esempio:

Prendiamo un pezzo di tabella di flusso di una rete qualunque, in cui ho segnato accanto ad ogni stato interno la codifica scelta.



Il caso a sinistra non crea eccessivi problemi, se non per il fatto che la rete dovrà quasi sicuramente attraversare più stati interni prima di trovare la stabilità (e quindi i vincoli di pilotaggio dovranno essere aggiustati per tener conto di questo). La corsa **è non critica.**

Nel caso a destra, la corsa è **critica**, perché se la variabile di stato y_0 "vince la corsa", cioè transisce **prima** di y_1, la rete evolve in modo **diverso** da quello che prevedevo (si raggiunge, infatti, un diverso stato stabile S1).

Le corse di variabili di stato **vanno evitate**, sia che siano critiche sia che siano non critiche. Come si fa a scegliere la codifica degli stati interni in modo da garantire l'assenza di corse? Esistono metodi formali, che non vedremo. Nei casi che vediamo in pratica la tabella è fatta di 3-4 stati interni, e quindi **si fa a mano**. Si guarda la tabella riga per riga, o meglio ancora il **grafo di flusso (dove si vede meglio)**, e per ogni stato interno S si guarda la **lista degli stati interni direttamente connessi** ad S. Tali stati dovranno avere codifiche adiacenti a quella di S.

Si parte da uno stato (qualunque, per esempio S0) e si sceglie **una codifica qualunque**. Poi si va a quelli a lui connessi, e si scelgono codifiche adiacenti a quella di S0, e si va avanti così, stando

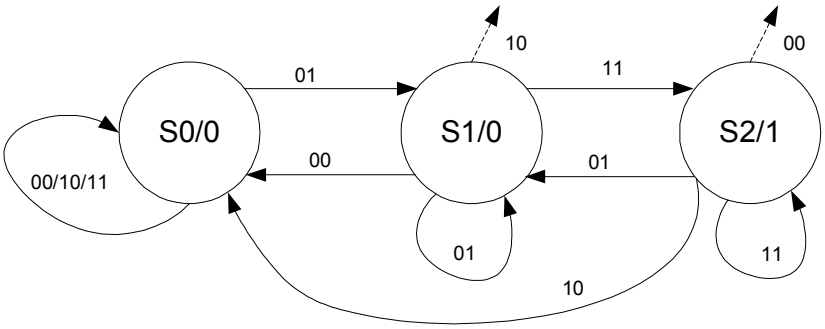
attenti a non creare inconsistenze. In tutti i casi che ci interessano, cosa fare si vede a occhio e non c'è bisogno di preoccuparsi.

REGOLA DI PROGETTO N.3: Gli stati interni devono essere codificati in modo da evitare corse. Condizione sufficiente perché ciò non accada è che stati interni che **possono essere consecutivi abbiano codifiche adiacenti.**

Si noti che, in ogni caso, ho dei **gradi di libertà** per la scelta della codifica degli stati interni. In particolare, per il primo stato interno posso scegliere la codifica che mi pare, senza che ciò limiti la mia libertà di aderire alla regola di progetto sopra scritta. In alcuni casi posso sfruttare questa libertà per ottenere delle reti **RC_B** più semplici (**corto circuiti**, ad esempio), visto che la sintesi di RC_B dipende dalla codifica degli stati interni.
Ci sono casi in cui **non è possibile** scegliere codifiche adiacenti per stati consecutivi. In particolare, in tutti i casi in cui esistono **percorsi ciclici di ordine tre** sul grafo di flusso.

2.2.4 Esempio

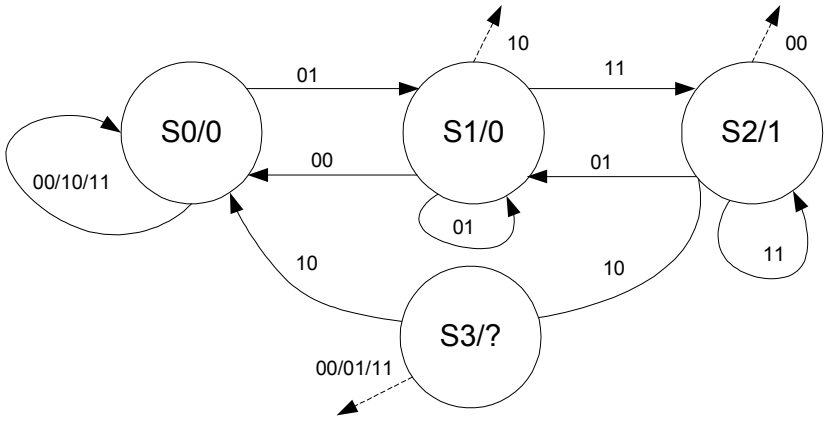
Costruiamo un **riconoscitore di sequenza 01, 11**. E' una rete simile a quella già vista, solo che la sequenza è di due passi invece che di tre.



In questo caso i tre stati S0, S1, S2 devono essere adiacenti, perché sono raggiungibili l'uno dall'altro. C'è un **percorso ciclico di ordine 3**. Qualunque codifica (su 2 bit) scelga per S0, non riuscirò mai a fare 3 stati adiacenti. Scegliamo, ad esempio, **S0=00, S1=01, S2=11.** Con questa scelta ho una transizione multipla di variabili di stato (corsa) quando si presenta l'ingresso 10 che fa transire la rete da S2 a S0. [Se avessi scelto un'altra codifica, avrei avuto corse per **un'altra transizione**, ma ne avrei avute in ogni caso]. Casi di questo genere sono possibili in pratica, e vanno saputi trattare.

Si può osservare che, con 2 bit a disposizione per la codifica degli stati interni, posso codificare 4 stati, e ne sto usando soltanto 3. Posso usare la rimanente codifica per creare uno **stato ponte**, cioè

un 4° stato "di appoggio" il cui solo scopo è quello di **guidare la transizione multipla (corsa)** delle variabili di stato in modo opportuno.



Tale stato ponte S3:

- **Dovrà avere una codifica intermedia** tra quelle dei due stati di partenza e arrivo, quindi **10**
- **Non dovrà essere uno stato stabile per 10**, e quindi non avrà l'orecchio.
- Non essendo S3 uno stato stabile, dovrò **evitare di variare lo stato di ingresso** quando S3 è marcato (**regola di pilotaggio n.1**), e quindi **non ha senso** specificare nel grafo di flusso il comportamento con gli altri ingressi (metto archi che si perdono)
- Ho libertà di assegnare il valore della variabile di uscita nel caso di stato interno S3. Visto che tale stato interno esiste in **via transitoria**, l'uscita può essere o quella dello stato di **partenza** (1) o quella dello stato di **arrivo** (0), indifferentemente. Quindi la lascio non specificata.

E' chiaro che, in presenza di stati ponte, **la rete non è più normale**. Gli stati ponte mi servono, però, a **guidare le corse** delle variabili di stato, ed in questo caso non ne posso fare a meno.

Facciamo la sintesi come esercizio:

x_1x_0	00	01	11	10	z
S0	(S0)	S1	(S0)	(S0)	0
S1	S0	(S1)	S2	--	0
S2	--	S1	(S2)	S0	1

1) **Descrizione della rete tramite tabella (o grafo) di flusso**
   a. Verifica dell'assenza di **oscillazioni**
   b. Verifica del **numero massimo di transizioni** di stato tra due stati stabili
   c. Verifica della presenza di **alee essenziali**
      -> non ce ne sono

x_1x_0	00	01	11	10	z
S0	(S0)	S1	(S0)	(S0)	0
S1	S0	(S1)	S2	--	0
S2	--	S1	(S2)	S3	1
S3	--	--	--	S0	-

2) Codifica degli stati interni

a. Verificare la presenza di corse delle variabili di stato **-> ci sono corse**

b. Se ci sono corse, introdurre **stati ponte** (eventualmente, ritoccare il numero massimo di transizioni di stato calcolato al punto 1).

3) Sintesi di RC_A

a. Si passa alla tabella delle transizioni

b. La sintesi deve essere priva di alee

y_1y_0 \ x_1x_0	00	01	11	10	z
00	00	01	00	00	0
01	00	01	11	--	0
11	--	01	11	10	1
10	--	--	--	00	-

a_1

y_1y_0 \ x_1x_0	00	01	11	10
00	0	0	0	0
01	0	0	1	--
11	--	0	1	1
10	--	--	--	0

a_0

y_1y_0 \ x_1x_0	00	01	11	10
00	0	1	0	0
01	0	1	1	--
11	--	1	1	0
10	--	--	--	0

$$a_1 = x_1 \cdot y_0 \quad a_0 = \overline{x_1} \cdot x_0 + x_0 \cdot y_0$$

La sintesi è priva di alee.

4) Sintesi di RC_B

a. Dipende dalla codifica degli stati

b. Non fa male che sia priva di alee

y_1	y_0	z
0	0	0
0	1	0
1	0	-
1	1	1

Avendo libertà sulla scelta dell'uscita da associare allo stato ponte S3, posso scegliere che valga 1, e in questo modo ho $\boxed{z = y_1}$, cioè un corto circuito.

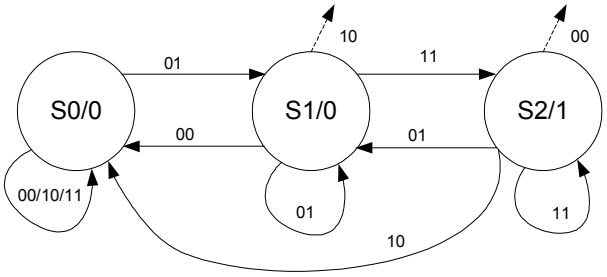
5) Dimensionamento dei ritardi e dei tempi di pilotaggio

$T_{mark} \geq T_A$ (la tabella non è normale, quindi non posso applicare il Teorema di Unger per capire se posso rendere il ritardo di marcatura nullo);

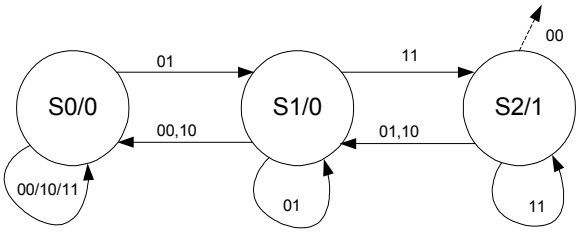
$T \geq T_A + 2\left(T_{mark} + T_A\right)$ (la rete deve fare due transizioni di stato interno, nel caso peggiore, prima di tornare stabile).

Attenzione:

Riprendiamo in mano il grafo di flusso originale (quello senza stato ponte)

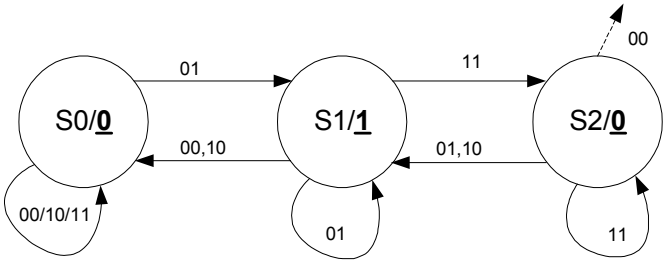


Il comportamento in S1 per ingresso 10 non è specificato, in quanto in S1 arrivo soltanto con ingresso 01. Posso **sfruttare questo fatto per usare <u>S1 stesso</u> come stato ponte**. Il grafo viene in questo modo:



Quando posso farlo, è bene **non introdurre stati aggiuntivi, ma riciclare quelli che già ci sono**: infatti, si ottengono sintesi in genere più semplici (la tabella delle transizioni conterrà una riga di non specificati).

Un caso classico in cui **non è possibile usare uno stato già presente come stato ponte** è questo [la rete è simile al caso precedente, ma ho cambiato i valori delle uscite nei vari stati interni]



In questo caso, se **usassi S1 come stato ponte** avrei un'alea in uscita (perché nel transito da S2 a S0 mi aspetto che l'uscita resti a 0, e invece passa per un certo tempo a 1). Quindi in questo caso è obbligatorio usare un altro stato, **in cui l'uscita dovrà essere 0**, come ponte tra S2 e S0. **Per esercizio** fare la sintesi della rete con S1 come stato ponte, e confrontarla con l'altra.

È possibile individuare una serie **di regole generali per l'individuazione di stati ponte**.

Data una transizione $S_i \xrightarrow{X} S_j$, con **Si, Sj non adiacenti**, posso usare **Sp** come stato ponte se:

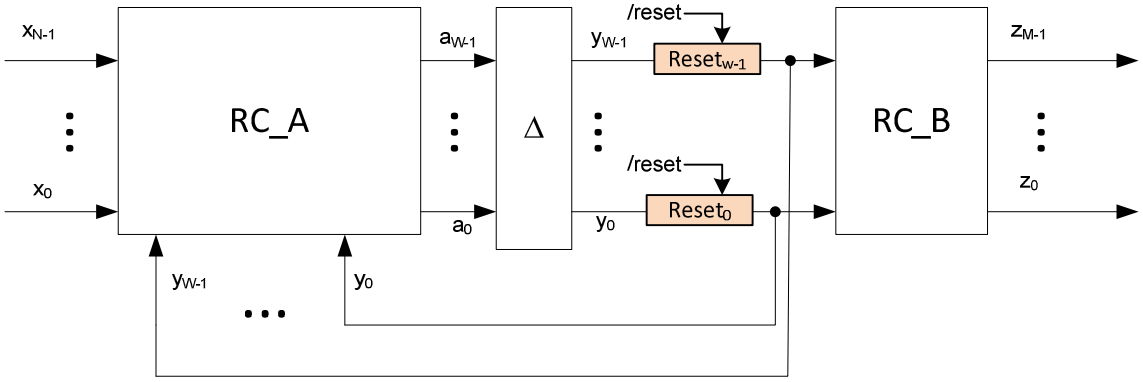
1) $S_i \xleftarrow{adj} S_p \xleftarrow{adj} S_j$, cioè lo stato ponte è adiacente a quello di partenza e di arrivo

2) $A(S_p, X) = S_j \vee A(S_p, X) = -$, cioè lo stato ponte porta direttamente in Sj con ingresso *X*, oppure posso specificare io dove porta (e quindi specificherò che porta in Sj)

3) La sequenza di transizioni di stato $S_i \xrightarrow{X} S_p \xrightarrow{X} S_j$ **non genera alee** in uscita a RC_B.

2.3 Inizializzazione al reset

Una RSA si evolve **a partire da uno stato interno iniziale**, che costituisce uno degli argomenti della prima applicazione della legge A. Per questo motivo, è necessario un metodo per impostare lo stato interno iniziale desiderato al reset. Un modo semplice per farlo è quello di far seguire gli elementi di ritardo da una **rete combinatoria**, che prende in ingresso la variabile **/reset**, ed imposta il valore richiesto per la codifica di ciascun bit dello stato interno quando **/reset=0**. Quando /reset=1, la rete combinatoria deve comportarsi da corto circuito.



Pertanto:

- se il bit j-simo dello stato interno deve avere valore iniziale 0, la rete combinatoria sarà una porta **AND**.
- se il bit j-simo deve avere valore iniziale 1, la rete combinatoria sarà dovrà avere la seguente tabella di verità:

/reset	y_j	y_j'
0	0	1
0	1	1
1	0	0
1	1	1

E sarà quindi $y_j' = y_j + \overline{/reset}$, oppure $y_j' = \overline{\overline{y_j} \cdot /reset}$.

Nel caso, il ritardo (piccolo) delle porte per l'inizializzazione al reset può essere contato **come parte del ritardo di marcatura.**

D'ora in avanti non considereremo la parte del circuito relativa all'inizializzazione, perché appesantisce il disegno. Negli esercizi si può direttamente scrivere che si assume che la rete parta dallo stato interno iniziale Sx, tanto sappiamo come fare.

2.4 Latch SR – una diversa implementazione

In precedenza abbiamo trattato il latch SR in modo intuitivo. Vediamo adesso una diversa implementazione, più diffusa.

sr	00	01	11	10	q	qN
S0	(S0)	(S0)	-	S1	0	1
S1	(S1)	S0	-	(S1)	1	0

Il latch SR è realizzato in accordo al modello di sintesi con **elementi neutri di ritardo** come meccanismo di marcatura (l'unica scelta possibile, ovviamente) nel modo che segue:

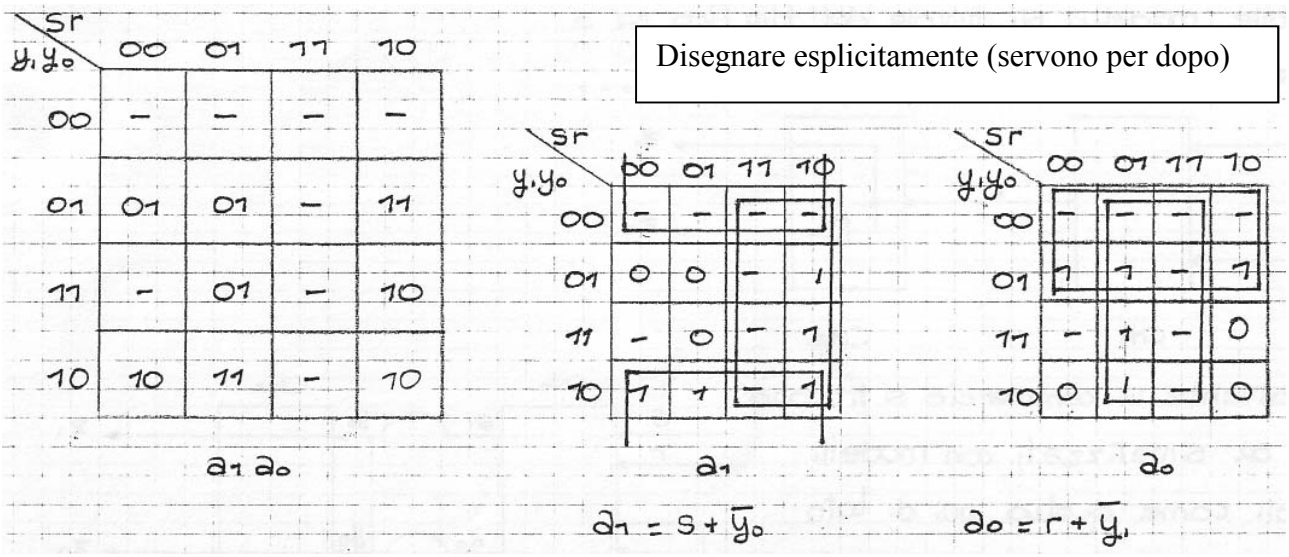
- **si codificano gli stati interni con 2 bit**, pari al valore delle uscite: **S0=01, S1=10**
- la rete RC_B è un **corto circuito**

La codifica su 2 bit degli stati interni (2 in tutto) è **ridondante**. Non solo, crea anche problemi di **corse** tra le variabili di stato. Perché adottarla, allora? Perché alla fine si arriva ad una sintesi **semplice** di rete **molto veloce** e **molto robusta** (nel senso che si comporta correttamente anche se le ipotesi di pilotaggio non sono rispettate).

Se ho due stati consecutivi **non adiacenti**, devo **guidare le corse inserendo uno stato ponte**. Ho due scelte possibili: SA=11, SB=00. Scegliamo **SA.** Le transizioni di stato S0-S1 ed S1-S0 dovranno tutte passare per SA.

sr	00	01	11	10	*q*	*qN*
S_0	(S_0)	(S_0)	–	S_A	0	1
S_1	(S_1)	S_A	–	(S_1)	1	0
S_A	–	S_0	–	S_1	1	1
S_B	–	–	–	–	0	0

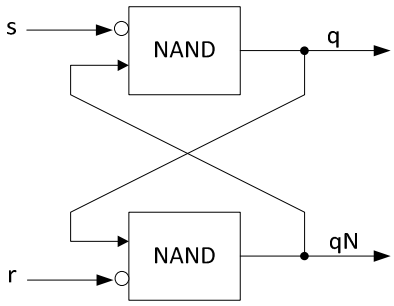
Sostituendo la codifica degli stati nella tabella di flusso, si ottiene la seguente sintesi:



che posso scrivere nella forma **a porte NAND** (nella quale viene normalmente realizzato il latch SR):

$$a_1 = \overline{\overline{s \cdot y_0}} \quad a_o = \overline{\overline{r \cdot y_1}}$$

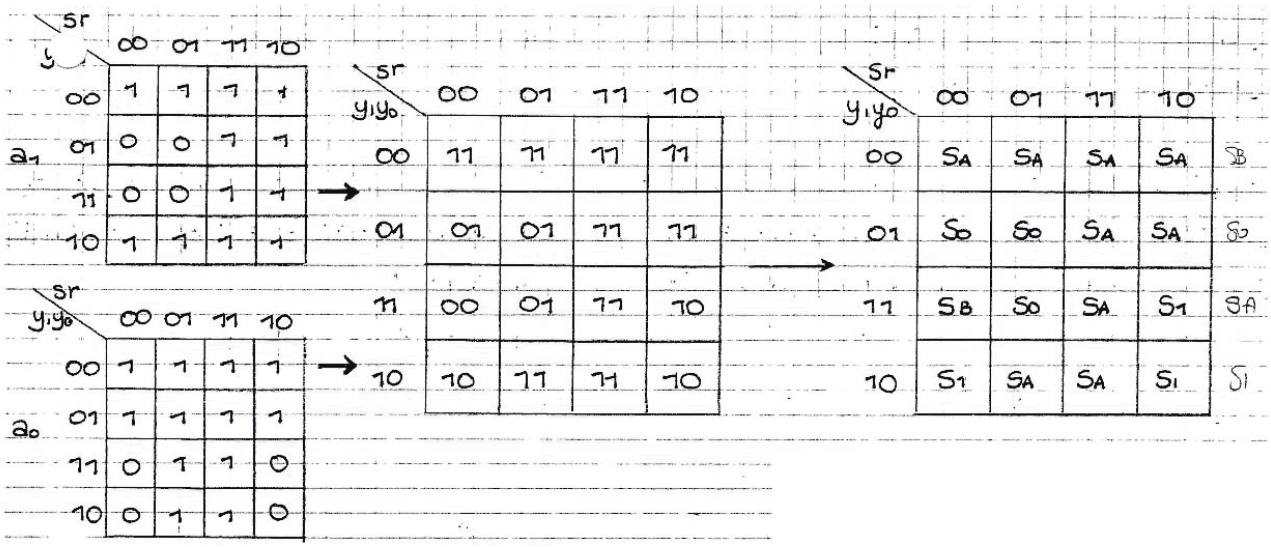
Per produrre il nuovo stato interno da marcare **basta un livello di logica**. La rete è veloce, e la sua implementazione è molto semplice.



Si osserva che i ritardi di marcatura possono **essere resi nulli**. Non lo si può vedere con il **teorema delle alee essenziali**, perché la tabella di flusso **non è normale**. Però, si vede a occhio che le variabili di stato **rientrano in zone della rete che non sono interessate da transizioni di ingresso**. Infatti, se cambio ***s, q*** rientra sull'**altra porta NAND, che è a regime.** Dato che non c'è bisogno di ritardi, la rete è a maggior ragione **veloce**.

Vediamo adesso perché questa rete è **robusta** nei confronti di pilotaggi scorretti.

La sintesi di sopra sfrutta molto la presenza di valori **non specificati**. Questi valori non erano stati specificati perché corrispondenti a situazioni in cui la rete, se pilotata correttamente, **non si dovrebbe mai trovare**. A sintesi terminata, però, alcuni di quei non-specificati sono finiti **coperti da (almeno) un implicante** (e quindi varranno 1), ed altri **non coperti** (e quindi varranno 0). Forte di questa conoscenza, posso percorrere **il percorso inverso**: dalla sintesi posso ricavare la tabella di flusso "realmente implementata", quella in cui le caselle sono state tutte specificate.



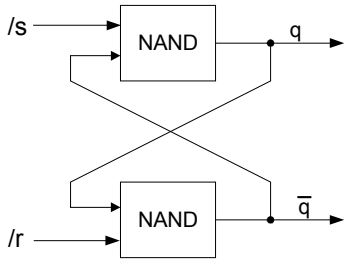
	sr 00	sr 01	sr 11	sr 10
SB	SA	SA	SA	SA
S0	(S0)	(S0)	SA	SA
SA	SB	S0	(SA)	S1
S1	(S1)	SA	SA	(S1)

Dimostriamo adesso che una tale sintesi è **robusta**. Supponiamo di avere, in fase di pilotaggio, violazioni della regola n.2: supponiamo che vengano forniti alla rete stati di ingresso consecutivi **non adiacenti** (il che non è fisicamente possibile: vorrà dire che nel mezzo ci sarà anche qualche altro stato di ingresso), e cerchiamo di capire cosa succede guardando la tabella di flusso.

X(i)	X(i+1)	Sequenza possibile X	Sequenza S corrispondente	Risultato
01	10	01-00-10 01-11-10	S0, S0, SA-**S1** S0, SA, **S1**	La rete si stabilizza in S1
10	01	01-00-10 01-11-10	S1, S1, SA-**S0** S1, SA, **S0**	La rete si stabilizza in S0
00	11	00-01-11 00-10-11	S0, S0, **SA** S1, SA-S0, **SA** S0, SA-S1, **SA** S1, S1, **SA**	La rete si stabilizza in SA (uscita 11)
11	00	11-10-00 11-01-00	SA, S1, **S1** SA, S0, **S0**	La rete si stabilizza in S1 La rete si stabilizza in S0

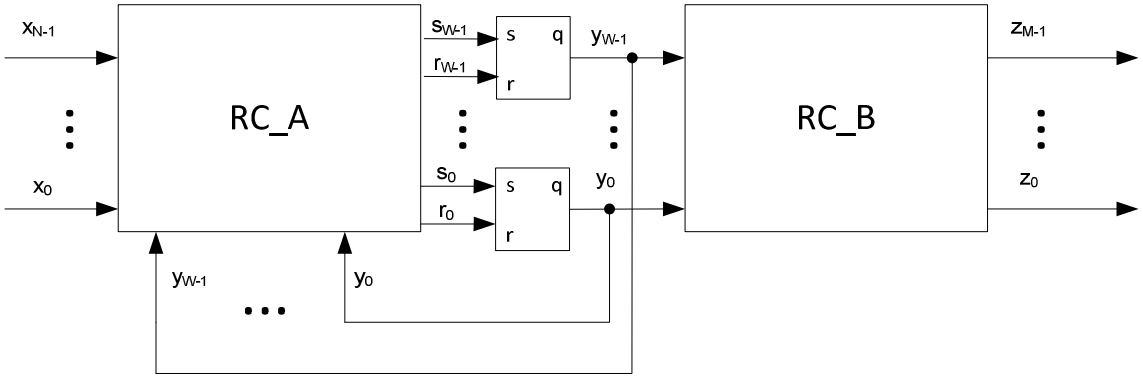
Quindi, in quasi tutti i casi il latch SR **tollera bene** transizioni multiple, nel senso che si stabilizza dove mi aspetto che si stabilizzi. Inoltre, la presenza dello stato di ingresso 11 non crea alcun problema (né transitoriamente, né come stato di ingresso stabile). L'unica cosa che non devo fare è passare da 11 a 00, perché in quel caso il latch SR si stabilizza in uno stato **casuale, a seconda dell'ordine di transizione delle variabili di ingresso.**

Talvolta, il latch SR in genere è realizzato con le variabili di ingresso **attive basse**.



2.5 Modello strutturale con latch SR come elementi di marcatura

Abbiamo visto un modello strutturale per la sintesi di RSA che utilizza **elementi neutri di ritardo** come elementi di marcatura. Visto che i **latch SR** servono, appunto**, a memorizzare dei bit**, posso usare questi come elementi di marcatura. Ne viene fuori un altro modello:



In questo caso, la sintesi della rete combinatoria RC_A richiede di produrre **un numero doppio** di variabili di uscita. Tali variabili non codificano più **il nuovo stato interno**, ma **l'ingresso da dare ai latch SR affinché marchino il nuovo stato interno.**

Sembra che mi stia complicando la vita. Visto che sintetizzo le uscite una per volta, adesso **dovrei fare il doppio del lavoro**, usando all'incirca **il doppio delle porte**. In realtà no, perché - con questo modello - le mappe che sintetizzano ciascuna delle uscite saranno **piene di non specificati**, e quindi le sintesi che ne risultano sono spesso addirittura **più semplici** che nel caso del modello ad elementi neutri di ritardo.

Vediamo la sintesi del **riconoscitore di sequenza** secondo questo modello strutturale. Manteniamo la stessa codifica degli stati interni già adottata, e scriviamo la tabella delle transizioni.

x_1x_0	00	01	11	10	z
S0	(S0)	S1	(S0)	(S0)	0
S1	S0	(S1)	S2	--	0
S2	--	S1	(S2)	S3	0
S3	S0	--	S0	(S3)	1

y_1y_0 \ x_1x_0	00	01	11	10
00	00	01	00	00
01	00	01	11	--
11	--	01	11	10
10	00	--	00	10

tabella di applicazione del latch SR

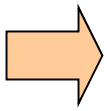
q	q'	s	r
0	0	0	-
0	1	1	0
1	0	0	1
1	1	-	0

La tabella delle transizioni, stavolta, non indica più le **uscite di RC_A**. Indica invece se il **nuovo valore** delle variabili di stato y_1 o y_0 (quello che leggo dentro una cella) è uguale o meno al vecchio (che leggo in cima alla riga corrispondente).

Prendiamo la tabella per la variabile di stato y_1:

y_1y_0 \ x_1x_0	00	01	11	10
00	0	0	0	0
01	0	0	1	--
11	--	0	1	1
10	0	--	0	1

a_1



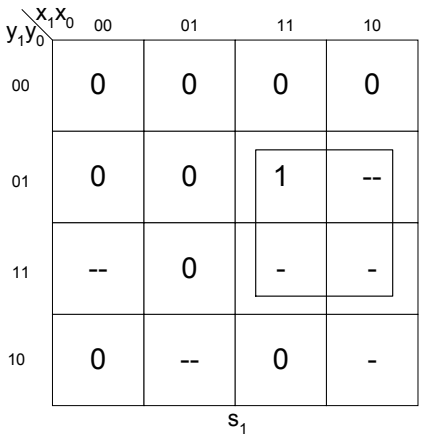
y_1y_0 \ x_1x_0	00	01	11	10
00	0-	0-	0-	0-
01	0-	0-	10	--
11	--	01	-0	-0
10	01	--	01	-0

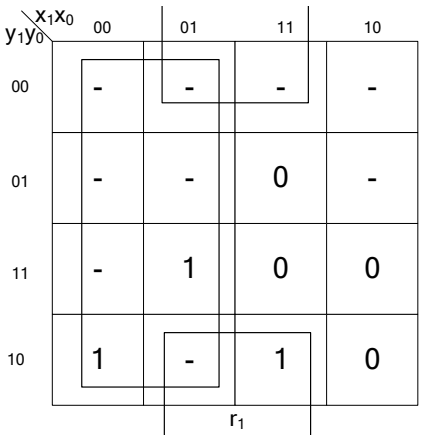
s_1 r_1

Guardando **la tabella di applicazione** del latch SR, posso facilmente dedurre che:

- se y_1 valeva 0 e deve continuare a valere 0 (cella 1,1), devo pilotare il latch SR con ***s*=0, *r*=-** (**o resetto o conservo**, tanto è lo stesso)
- se y_1 valeva 0 e deve valere 1 (cella 2,3), devo **settare** il latch SR, cioè pilotarlo con ***s*=1, *r*=0**
- se y_1 valeva 1 e deve continuare a valere 1 (cella 3,3), devo pilotare il latch SR con ***s*=-, *r*=0** (**setto oppure conservo**)
- se y_1 valeva 1 e deve valere 0 (cella 4,1), devo **resettare** il latch SR, cioè pilotarlo con ***s*=0, *r*=1**

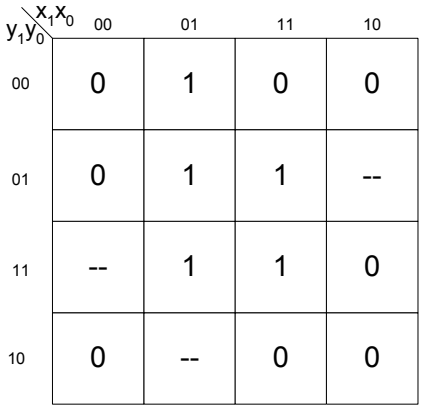
La mappa che viene fuori è piena di non specificati. Vediamo la sintesi:

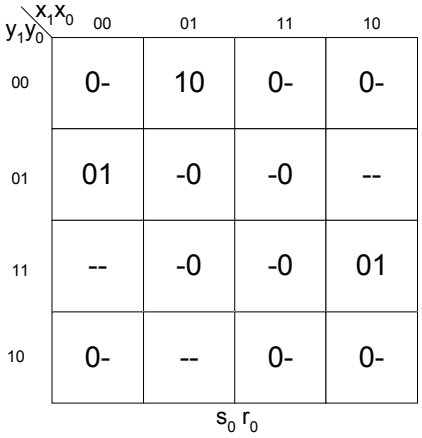




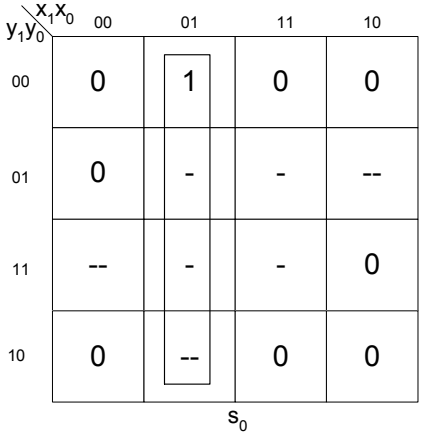
$$s_1 = y_0 \cdot x_1 \quad r_1 = \overline{x_1} + \overline{y_0} \cdot x_0$$

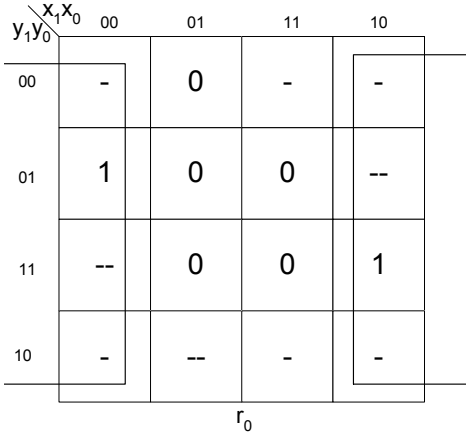
La stessa cosa posso fare con l'altra variabile di stato y0





e quindi, per la sintesi





$$s_0 = \overline{x_1} \cdot x_0 \quad r_0 = \overline{x_0}$$

Il costo a porte è pari a 4, mentre il costo della precedente sintesi era pari a 6. Analogamente, quello a diodi è **8** contro **12.**

Osservazione 1:

La batteria di latch SR **introduce del ritardo**. Tale ritardo potrebbe essere già sufficiente a garantire il buon funzionamento della rete qualora questa sia affetta da alee essenziali (questa è l'assunzione che si fa in genere). Se non fosse sufficiente, basta aggiungerne altro in cascata.

Osservazione 2:

Non ci siamo preoccupati del fatto che RC_A ha **due uscite che vanno in ingresso ad un singolo latch SR,** e che in pratica possono cambiare **in un ordine qualunque**, in modo tale da **violare le regole di pilotaggio** del latch SR (che è esso stesso una RSA). **Ma sappiamo che il latch SR tollera bene pilotaggi scorretti**. L'unica cosa che non devo fare è **passare da 11 a 00**, poi evolve sempre correttamente. Nel caso della sintesi di sopra, **non succede mai** (in quanto seguendo questo procedimento non viene mai dato 11 in ingresso ad uno dei latch SR).

Osservazione 3:

Se voglio inizializzare al reset una rete così sintetizzata, posso agire sugli ingressi di /preset e /preclear dei latch SR (per semplicità non li disegniamo, ma sappiamo che ci sono).

Per gli **esercizi d'esame** ci sono quindi due alternative, entrambe lecite, riguardo a quale modello usare. In alcuni esercizi può essere chiesto esplicitamente di usare uno dei due modelli. In mancanza di una indicazione del genere, si può usare l'uno o l'altro indifferentemente.

2.6 Riepilogo sulla sintesi di reti sequenziali asincrone

1) **Descrizione** della rete tramite tabella di flusso o grafo di flusso.

- Verificare l'assenza di **oscillazioni**
- Calcolare il **numero massimo di transizioni** di stato tra due stati stabili
- Verificare l'assenza di **alee essenziali (se tabella normale)**
- Verificare se esistono percorsi ciclici di ordine 3 (che rendono necessari stati ponte)

2) **Sintesi:** scelta di un modello strutturale (con elementi neutri di ritardo o con latch SR)

2b) Scelta della codifica degli stati interni

- Verificare l'assenza di corse delle variabili di stato
- Se ci sono corse, introdurre **stati ponte** (eventualmente, ritoccare il numero massimo di transizioni di stato calcolato al punto 1)
- Se sono necessari stati ponte, discutere quali stati si possono usare.

2c) Sintesi di RC_A

- Dipende dal modello strutturale scelto
- Deve essere priva di alee

2d) Sintesi di RC_B

- Non dipende dal modello strutturale scelto
- Dipende dalla codifica degli stati
- Non fa male che sia priva di alee

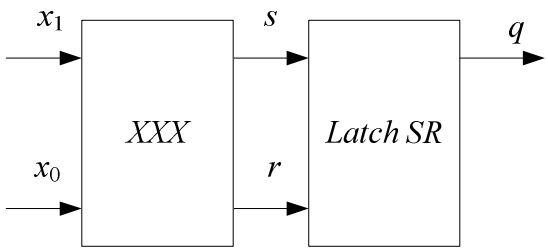
3) Dimensionamento del ritardo di marcatura

- Se non si hanno alee essenziali, può essere reso nullo
- Altrimenti, deve essere maggiore del tempo di attraversamento di RC_A (che conosco dopo che ne ho fatta la sintesi)
- Se uso dei latch SR come meccanismo di marcatura, questi introducono un ritardo. Tale ritardo è di norma sufficiente per evitare problemi di pilotaggio alla rete RC_A.

3b) Scrittura delle regole di pilotaggio in modo fondamentale, in dipendenza da:

- Numero massimo di transizioni di stato tra 2 stati stabili
- Ritardo di marcatura calcolato al punto precedente.

2.6.1 Esercizio Gennaio 2014 (descrizione e sintesi)



Descrivere la rete sequenziale asincrona XXX in modo tale che la variabile q *commuti* (una volta) ogni qual volta si presenta in ingresso ad XXX lo stato $x_1 x_0 = 11$, e conservi il suo valore altrimenti.

Sintetizzare XXX secondo il modello con elementi neutri di ritardo, sintetizzandone le reti combinatorie in forma SP. Calcolare il tempo *minimo* per cui l'ingresso di XXX deve rimanere costante.

NB: non preoccuparsi del valore che q assume *la prima volta* che XXX riceve in ingresso 11 dopo il reset.

Soluzione

La rete XXX deve essere fatta in modo tale che le sue uscite siano

- alternativamente 10, 01 quando gli ingressi sono 11
- 00 in tutti gli altri casi.

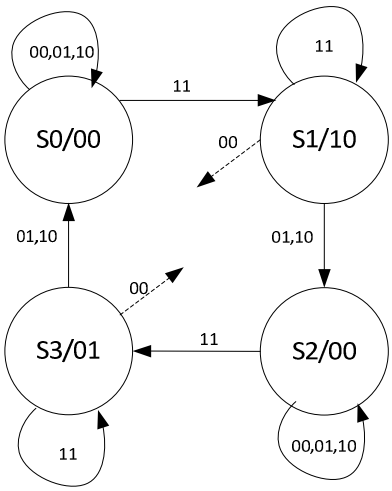
Per la descrizione si può ragionare in questo modo:

Dovendo la rete esibire tre diversi stati di uscita, dovrò mettere a preventivo **almeno** tre stati interni (magari ce ne vorranno di più). Supponiamo che **al reset** si abbia q=0, e ragioniamo di conseguenza.

Parto da uno stato interno S0, nel quale do al latch SR a valle il comando di **conservazione** (00). Da questo stato mi muovo soltanto quando vedo l'ingresso 11. Con ingresso 11, do un comando di **set** al latch SR: mi muovo in uno **stato S1**, dove mi stabilizzo finché non cambia l'ingresso.

Quando cambia l'ingresso, non posso tornare in S0. Se lo facessi, la **prossima volta** che ho in ingresso 11 sarei condannato a ripercorrere lo stesso arco, e quindi non potrei che settare il latch SR, mentre lo devo **resettare.** Mi serve quindi un **nuovo stato interno**, chiamiamolo S2, in cui il comando che do al latch SR a valle è sempre di **conservazione (00)**. In tale stato S2 resto fintanto che non ho nuovamente ingresso 11.

Con ingresso 11, a questo punto, devo dare un comando di **reset**: mi serve un nuovo stato interno, chiamiamolo S3, dove questo comando viene presentato in uscita. Da S3 devo tornare in S0 appena possibile, per ricominciare il giro.



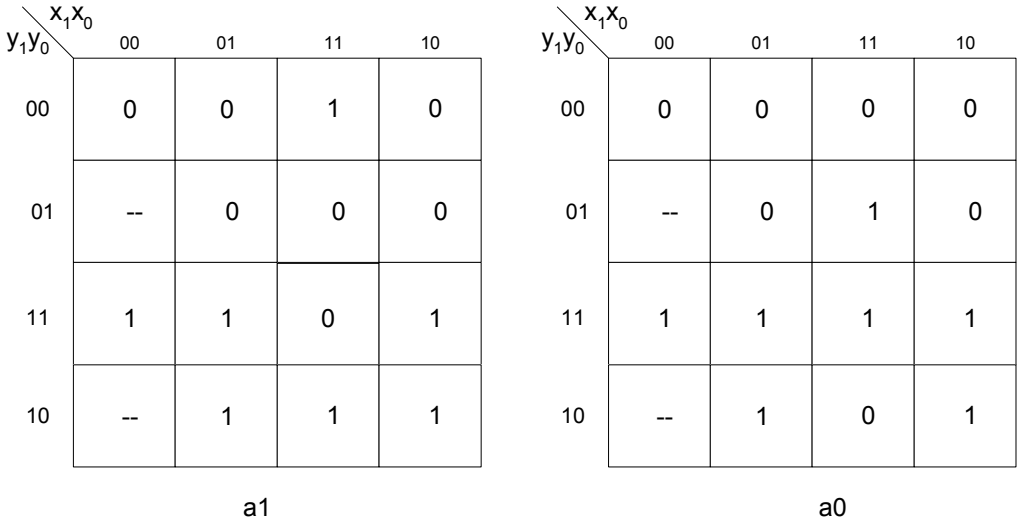
La tabella di flusso che traduce il grafo di cui sopra è quindi la seguente. Si osserva che, visto che gli stati S0 ed S2 seguono, rispettivamente, uno stato S3 in cui il comando è **reset** ed uno stato S1 in cui il comando è di **set**, sono sicuro che il latch sta conservando il valore 0 in S0, ed 1 in S2. Posso sfruttare questa conoscenza per **evitare di specificare una delle uscite** in S0 ed S2.

x_1x_0	00	01	11	10	sr
S0	(S0)	(S0)	S1	(S0)	0-
S1	--	S2	(S1)	S2	10
S2	(S2)	(S2)	S3	(S2)	-0
S3	--	S0	(S3)	S0	01

La rete è normale e soggetta ad alee essenziali (e.g., da S0, 01->11). Quindi il ritardo minimo degli elementi di marcatura deve essere uguale al tempo di attraversamento di RC_A.

Adottando la codifica S0=00, S1=10, S2=11, S3=01, che è priva di corse, si ottiene per la rete RC_B l'espressione $s = y_1$, $r = \overline{y_1}$.

Utilizzando come meccanismo di marcatura degli elementi neutri di ritardo, si ottengono le seguenti mappe per la rete combinatoria RC_A:



y_1y_0 \ x_1x_0	00	01	11	10
00	0	0	1	0
01	--	0	0	0
11	1	1	0	1
10	--	1	1	1

a1

y_1y_0 \ x_1x_0	00	01	11	10
00	0	0	0	0
01	--	0	1	0
11	1	1	1	1
10	--	1	0	1

a0

Dalle quali si ottiene:

$$a_1 = y_1 \cdot \overline{y_0} + y_1 \cdot \overline{x_1} + y_1 \cdot \overline{x_0} + x_1 \cdot x_0 \cdot \overline{y_0},$$

$$a_0 = y_1 \cdot y_0 + y_1 \cdot \overline{x_1} + y_1 \cdot \overline{x_0} + x_1 \cdot x_0 \cdot y_0.$$

Il costo a porte della rete RC_A è 8 (e non 10), in quanto le stesse due porte AND possono essere utilizzate contemporaneamente nella sintesi di a_1 ed a_0. Analogamente, il costo a diodi è 22 e non 26.

Il tempo minimo di permanenza di uno stato di ingresso è $T = 3 \cdot T_{CN1}$.

2.7 D-Flip-Flop 7474

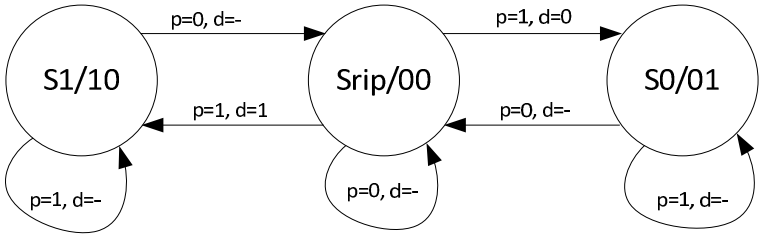
Il D-FF 7474 è un FF di tipo positive-edge-triggered, ed è uno dei più diffusi FF non trasparenti. Si basa su:

- una rete che campiona *d* sul fronte in salita di p (**campionatore**)
- una rete che trattiene l'uscita del campionatore fino a quando quest'ultimo non è più sensibile agli ingressi (**ritardatore**).

Quest'ultima rete è **un latch SR**. Questo mi consente di sintetizzare il campionatore in modo **semplice**. Infatti, la rete campionatore dovrà

- nella maggior parte del tempo, **conservare** il valore campionato (quale che sia)
- sul fronte di salita di p, **memorizzare 1 o memorizzare 0**, e poi conservare.

Devo poter pilotare il latch SR ritardatore con comandi di **conservazione, set e reset**. Quindi, visto che il campionatore è una RSA e lo stato di uscita di una RSA è funzione solo dello stato interno, la rete campionatore ha ***almeno* 3 stati interni**.



- **Srip:** Quando p=0, la rete è in **conservazione**. L'uscita del campionatore vale 00, la rete non è sensibile a d. Quando ***p* va ad 1**, posso dover **resettare** il ritardatore (se d=0) o **settarlo** (se d=1). Visto che le uscite del campionatore saranno diverse nei due casi, devo prevedere **due stati distinti**, S0 ed S1 nei quali, rispettivamente, resetto e setto il ritardatore.
- **S0**: arrivo in questo stato se d=0 quando p va ad 1. Quindi devo dare ordine al ritardatore di resettare, e **devo restare in S0**. Non posso tornare in *Srip* subito, altrimenti potrebbe darsi che l'ingresso al ritardatore non venga mantenuto per abbastanza tempo. S0 deve essere **stabile**. Inoltre, in questo stato, **non devo ascoltare *d*,** perché ho già dato il comando al ritardatore. L'unica transizione di stato che mi può far muovere da S0 è quella per cui ***p* va a zero**. In questo caso torno in *Srip*, preparandomi ad un nuovo fronte in salita di p.
- **S1**: vale esattamente tutto ciò che ho detto per S0, salvo che in questo caso il ritardatore deve ricevere in ingresso 10.

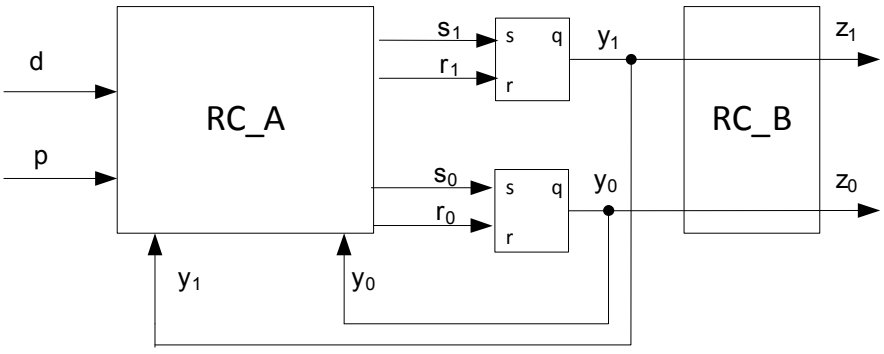
Dal grafo di flusso si ricava la tabella di flusso:

d	p=0		p=1		$z_s z_r$
	0	1	0	1	
Srip	(Srip)	(Srip)	S0	S1	00
S0	Srip	Srip	(S0)	(S0)	01
S1	Srip	Srip	(S1)	(S1)	10

Vediamo di sintetizzare il campionatore appena descritto.

Osservo che ci sono **tre stati**. Però non c'è **nessun percorso ciclico di ordine 3**, in quanto S0 e S1 **non possono mai essere consecutivi**. Pertanto non si pone il problema di dover utilizzare stati ponte per recuperare l'adiacenza tra stati consecutivi. Posso anzi codificare gli stati interni direttamente con il valore delle uscite (da dare in ingresso al ritardatore), in modo da ottenere una rete RC_B di complessità nulla (cortocircuito).

Scelgo come modello strutturale quello con **latch SR come elemento di ritardo**.



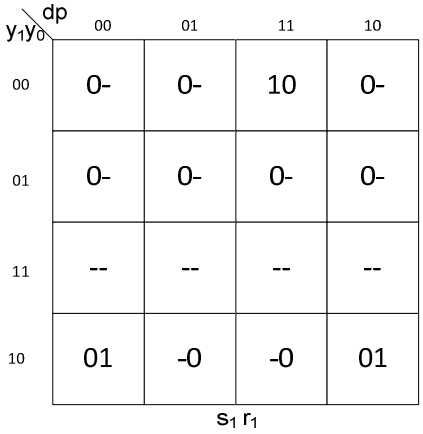
La sintesi si ottiene mettendo a confronto la tabella delle transizioni e la tabella di applicazione del latch SR:

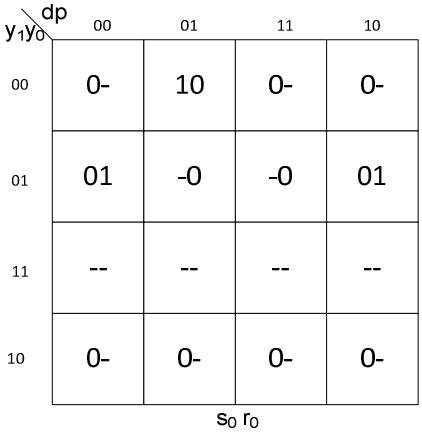
y_1y_0 \ dp	00	01	11	10
00	00	01	10	00
01	00	01	01	00
11	--	--	--	--
10	00	10	10	00

$a_1\ a_0$

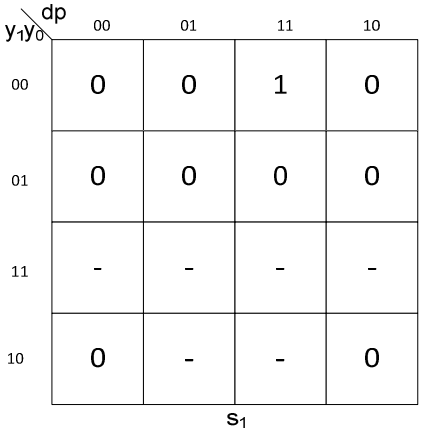
Tabella di applicazione del latch SR

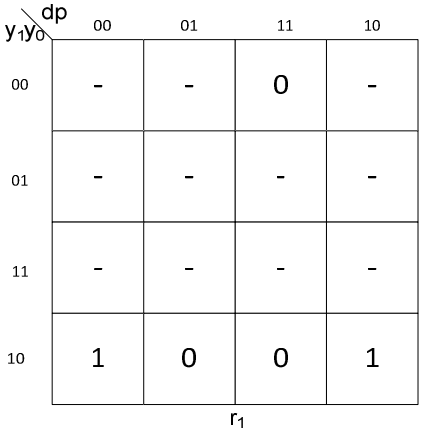
q	q'	s	r
0	0	0	-
0	1	1	0
1	0	0	1
1	1	-	0





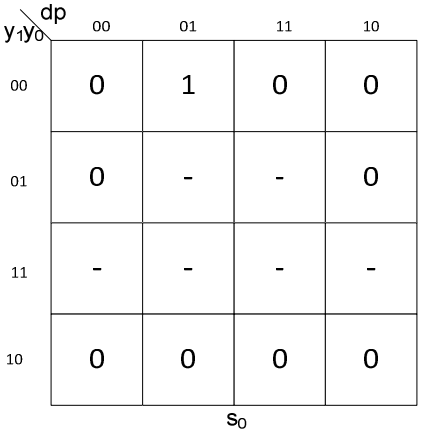
Per la parte di uscite di RC_A che va al latch SR che memorizza y_1 abbiamo:

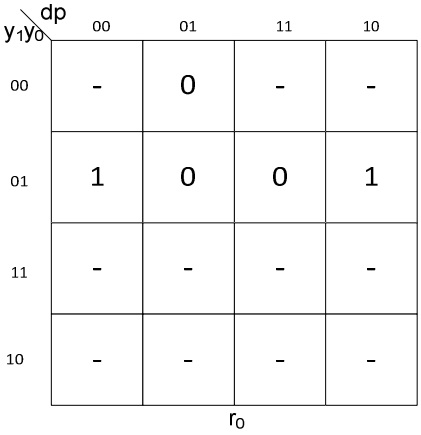




$$s_1 = p \cdot d \cdot \overline{y_0} = \overline{\overline{p \cdot d \cdot \overline{y_0}}} = \overline{\overline{p} + \overline{d} + y_0}\,, \qquad r_1 = \overline{p}$$

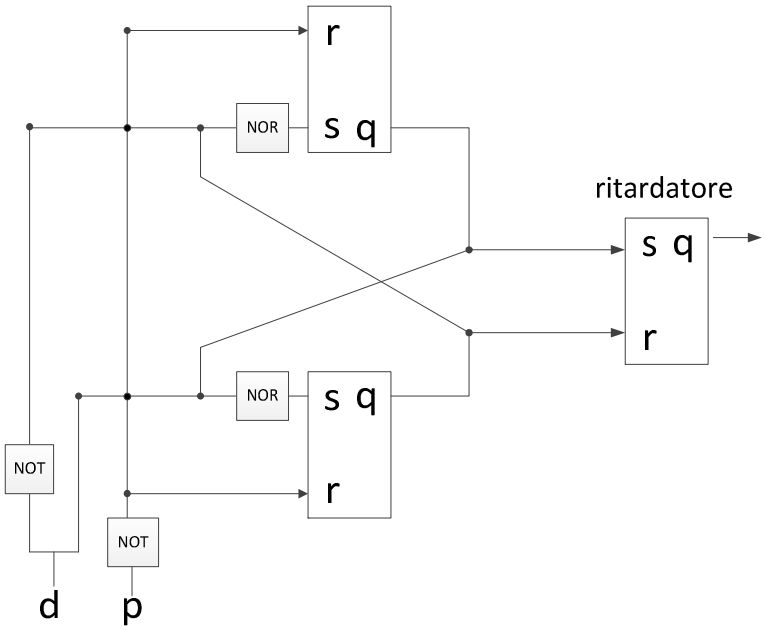
Per la parte di uscite di RC_A che va al latch SR che memorizza y_0 abbiamo:



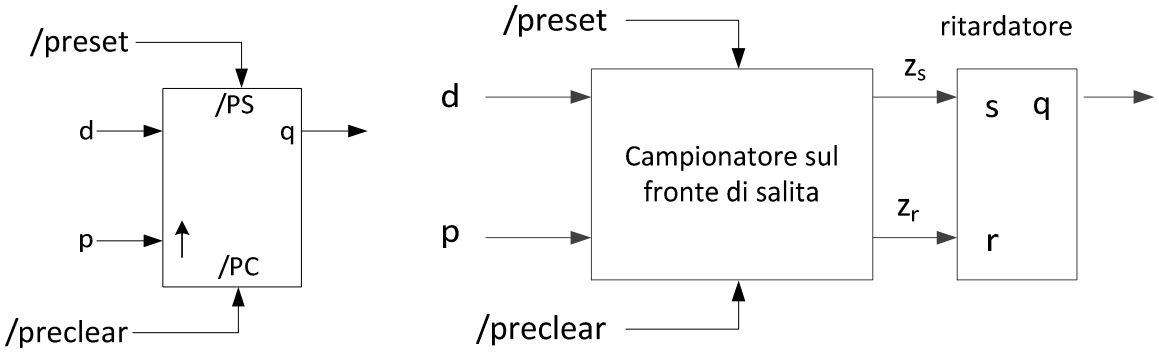


$$s_0 = p \cdot \overline{d} \cdot \overline{y_1} = \overline{\overline{p \cdot \overline{d} \cdot \overline{y_1}}} = \overline{\overline{p} + d + y_1}\,, \qquad r_0 = \overline{p}$$

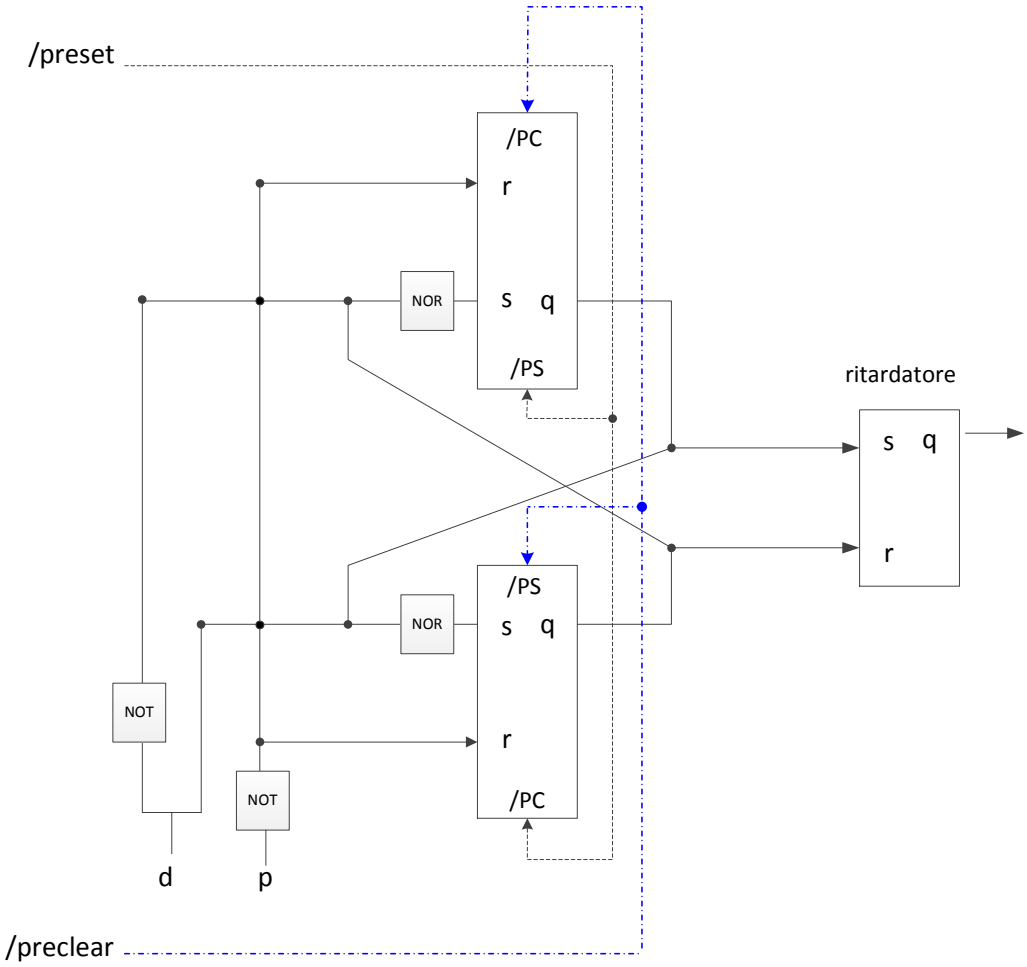
RC_A richiede **un solo livello di logica** (in quanto si è usato il modello con latch SR come elementi di marcatura, che in genere rende la rete RC_A più semplice).



Il D-FF 7474 deve poter essere **inizializzato al reset** (**i registri dei processori** sono fatti con tecnologia simile). Per poterlo fare, è necessario dotarlo di due ingressi cui connettere le variabili /preset e /precelar, che devono essere connesse opportunamente.



In particolare, per essere sicuri che in uscita al ritardatore appaia il valore desiderato al reset, basta **assicurarsi che il campionatore parta nello stato S1 o S0**, cioè passi al ritardatore un comando di *set* (10) o di *reset* (01). Quindi è il campionatore a dover avere /preset e /preclear. Gli ingressi vanno connessi agli ingressi /PS e /PC dei latch SR che fungono da elementi di memorizzazione:


/preset
/PC
r
NOR
s q
/PS
ritardatore
s q
r
/PS
NOR
s q
r
/PC
NOT
NOT
d
p
/preclear

3 Esercizi

3.1 Esercizio – Sintesi

Sintetizzare, secondo il modello strutturale con elementi neutri di ritardo, la rete sequenziale asincrona descritta dalla seguente tabella di flusso, introducendo stati ponte ove necessario. Per la sintesi delle reti combinatorie utilizzare solo porte NOR.

Detto inoltre T_L il tempo di attraversamento di un livello di logica, calcolare il tempo per cui uno stato di ingresso deve rimanere costante.

x_1x_0	00	01	11	10	z
S_0	(S0)	S1	—	S2	0
S_1	S0	(S1)	(S1)	S2	0
S_2	S0	—	S1	(S2)	1

3.1.1 Soluzione

[**Osservazione preliminare**: la sintesi di RSA è cavillosa, ma non difficile. Se uno ha capito e studiato tutto non ha troppi problemi. La vera difficoltà sta nella **descrizione**, cioè nel **progettare una rete.** Vedremo più avanti alcuni esercizi che richiedono di fare anche questo].

Intanto osserviamo che la tabella è **normale**, e **priva di alee essenziali**. Ciononostante si vede facilmente che c'è un percorso ciclico di ordine 3, e quindi **saranno necessari stati ponte**.

Adottando le codifiche **S_0 = 00, S_1 = 01, S_2 = 11**, si rende necessario l'uso di due stati ponte, rispettivamente fra gli stati **S_0 e S_2**, nel passaggio dello stato d'ingresso da 'B00 a 'B10, e fra gli stati **S_2 e S_0**, nel passaggio dello stato d'ingresso da 'B10 a 'B00. In entrambi i casi si può utilizzare come stato ponte lo stato S_1. La tabella diventa quindi:

x_1x_0	00	01	11	10	z
S_0	(S0)	S1	—	S1	0
S_1	S0	(S1)	(S1)	S2	0
S_2	S1	—	S1	(S2)	1
S_3	—	—	—	—	-

La tabella non è più normale, ma non ho più problemi di corse.

Con riferimento al modello strutturale con elementi neutri di ritardo, le mappe di Karnaugh relative alle uscite della rete RC_A sono:

y_1y_0 \ x_1x_0	00	01	11	10	z
00	00	01	- -	01	0
01	00	01	01	11	0
11	01	- -	01	11	1
10	- -	- -	- -	- -	-

a_1a_0

Una possibile implementazione della rete RC_A (esente da alee statiche) è la seguente:

$$a_1 = \overline{\overline{y_0} + \overline{x_1} + x_0}\,, \qquad a_0 = x_1 + x_0 + y_1$$

Si noti che a0 non è sintetizzata a porte NOR, in quanto consta di un solo livello di logica.

Per la rete RC_B, è immediato verificare che $z = y_1$.

3.2 Esercizio – Descrizione e sintesi

Si consideri un D-flip-flop di tipo positive-edge-triggered che ha due variabili di ingresso *d* e *p* ed una variabile di uscita *q* e che si evolve in accordo alle seguenti specifiche: "Quando la variabile *p* transisce da 0 a 1 campiona il valore della variabile *d*; al termine della fase di campionamento, se il valore di *d* campionato è 0, il D-flip-flop si *resetta* (cioè mette a 0 il valore della variabile di uscita *q*), altrimenti, se il valore di *d* campionato è 1, il flip-flop *commuta* (cioè modifica il valore della variabile *q*)".

Con riferimento ad una implementazione di tale D-flip-flop come coppia di reti campionatore-ritardatore, **descrivere** e **sintetizzare** la rete campionatore come rete sequenziale asincrona.

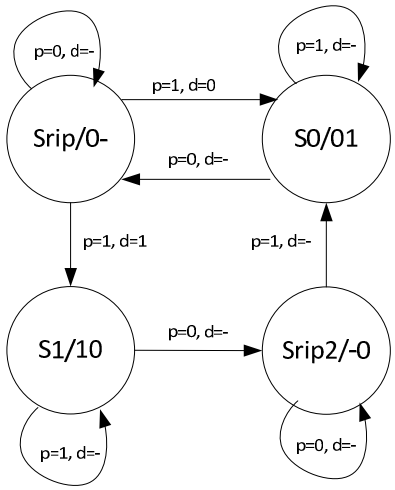
3.2.1 Soluzione

Abbiamo una rete campionatore-ritardatore. Il ritardatore sarà, al solito un latch SR, che va pilotato con comandi di **conservazione, set e reset**. Il problema è che **non ho, per il latch SR, un comando di commutazione**. Posso surrogare questa mancanza se **ricordo l'ultimo comando dato al ritardatore**, e per commutare ne fornisco uno di conseguenza:

- se l'ultima volta gli avevo detto di **resettare**, stavolta gli dirò di **settare**
- se l'ultima volta gli avevo detto di **settare**, stavolta gli dirò di **resettare**

E **la prima volta** che comando devo inviare? **Non mi interessa**, una cosa o l'altra va bene, tanto non è specificato.

- Partiamo quindi da uno stato iniziale **Srip**, in cui la rete campionatore **non fa niente**. Tale stato permane (**è stabile**) fintanto che *p* è a 0, qualunque cosa faccia *d*. Supponiamo che in questo stato l'uscita del D-FF valga **0**. Affinché l'uscita sia 0, posso porre le variabili di uscita del campionatore o a 00 (do al ritardatore il comando di **conservare**), o a 01 (do al ritardatore il comando di **reset**). Da qui mi muovo se *p* vale 1. In questo caso:



  - Se *d* vale 0, devo **resettare** il ritardatore. Devo muovermi in uno stato **diverso** in cui le uscite del campionatore saranno 01. Chiamiamolo **S0.**
  - Se *d* vale 1, devo **commutare** il ritardatore, cioè devo **settarlo**, perché ho supposto che l'uscita valesse 0 quando ero in Srip. Devo muovermi in uno stato **diverso** in cui le uscite del campionatore saranno 10. Chiamiamolo **S1.**
- **S0**, **S1** saranno **stabili** finché *p* vale 1, qualunque cosa faccia *d*. Il problema, a questo punto, è stabilire **dove si va quando p ritorna a 0**. Se fosse un D-FF standard, ritornerei in Srip. Ma se torno in Srip, la prossima volta che arriva un comando di **commutazione** (p=1, d=1) farei la stessa cosa che ho fatto prima, e non va bene. Non va bene perché **perdo memoria dell'ultima cosa che ho fatto**, in quanto ritorno nello stesso stato.

La soluzione è che **mi serve un altro stato.**

- Supponiamo di essere in S1, e supponiamo che *p* vada a 0. Devo andare in uno stato **Srip2,** in cui resterò fintanto che *p* vale 0, ma quando arriva *p*=1 farò **cose diverse da quelle che facevo in Srip**. In particolare, se d=0, andrò in uno stato in cui resetto (S0). Se, invece **d=1**, andrò in uno stato in cui, ancora, devo **resettare il ritardatore**, visto che l'ultima volta lo avevo settato. Devo andare ancora in S0. Per quanto riguarda **l'uscita** in Srip2, questa dovrà essere di **conservazione**. Quindi, in prima approssimazione, mettiamo che è **00**.
- Manca da definire cosa faccio in S0 quando *p* va a 0. Devo ritornare in Srip, perché se arriva un nuovo comando di commutazione (p=1, d=1), stavolta devo **settare il ritardatore.**

Quindi, posso osservare che in Srip2 **conservo il valore 1**, e quindi lo stato di uscita da dare in ingresso al ritardatore può essere –0.

Traducendo in tabella di flusso:

d	$p=0$		$p=1$		z_1z_0
	0	1	0	1	
S_{rip}	(S_{rip})	(S_{rip})	S_0	S_1	0-
S_0	S_{rip}	S_{rip}	(S_0)	(S_0)	01
S_1	S_{rip2}	S_{rip2}	(S_1)	(S_1)	10
S_{rip2}	(S_{rip2})	(S_{rip2})	S_0	S_0	-0

La tabella è normale, ed affetta da alee essenziali (partendo da Srip con ingresso d=1,p=0, al fronte di salita di p si va in S1, e se si varia nuovamente p due volte si finisce invece in S0).

Domanda: ora che abbiamo dato la tabella di flusso, cosa fa la rete quando campiona il **primo d=1**?

Dipende dallo stato in cui è partita. Se mi interessa che faccia una cosa o un'altra, impongo lo stato di partenza agendo sulle variabili di inizializzazione (/preset e /preclear), che non sono state disegnate ma ci sono.

Adottando le codifiche $S_{rip} = 00$, $S_0 = 01$, $S_1 = 10$, $S_{rip2} = 11$, la tabella delle transizioni corrispondente risulta essere

y_1y_0 \ pd	00	01	11	10
00	00	00	10	01
01	00	00	01	01
11	11	11	01	01
10	11	11	10	10

a_1a_0

Con riferimento al modello strutturale con latch SR come elementi di marcatura (in genere sufficienti a generare il ritardo che ci vuole) , le mappe di Karnaugh relative alle uscite della rete RC_A sono:

y_1y_0 \ pd	00	01	11	10
00	0-	0-	10	0-
01	0-	0-	0-	0-
11	-0	-0	01	01
10	-0	-0	-0	-0

s_1r_1

y_1y_0 \ pd	00	01	11	10
00	0-	0-	0-	10
01	01	01	-0	-0
11	-0	-0	-0	-0
10	10	10	0-	0-

s_0r_0

Ottimizzando si ottiene:

$s_1 = d \cdot p \cdot \overline{y}_0$, $r_1 = p \cdot y_0$

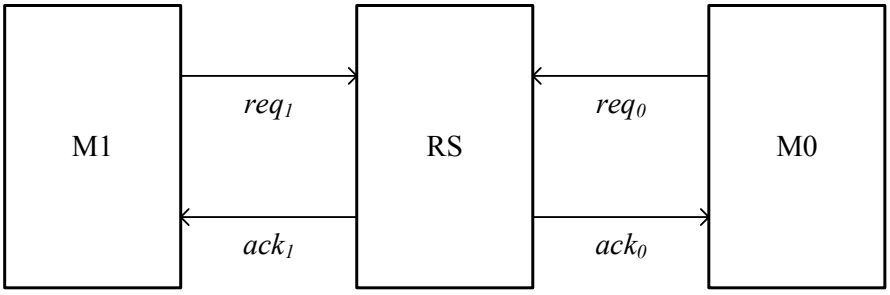
$s_0 = \overline{p} \cdot y_1 + \overline{d} \cdot p \cdot \overline{y}_1$, $r_0 = \overline{p} \cdot \overline{y}_1$

Per la rete RC_B, è immediato verificare che $z_1 = y_1$, $z_0 = \overline{y}_1$.

3.3 Esercizio – Descrizione e sintesi

Con riferimento alla figura, la rete sequenziale asincrona RS effettua un handshake asincrono – senza scambio di dati – con il modulo M_1 (M_0) mediante la coppia di collegamenti req_1, ack_1 (req_0, ack_0).

L'iniziativa degli handshake è presa dai moduli M_1 ed M_0 e non da RS, che si limita a rispondere. Inoltre, l'iniziativa non viene mai presa contemporaneamente, nel rispetto delle regole di pilotaggio delle reti sequenziali asincrone, ma può accadere che mentre RS gestisce l'handshake con un modulo, l'altro inizi a sua volta l'handshake. *Gli handshake sono diversi da quelli visti a lezione* e tutti i segnali sono attivi alti.



In dettaglio:

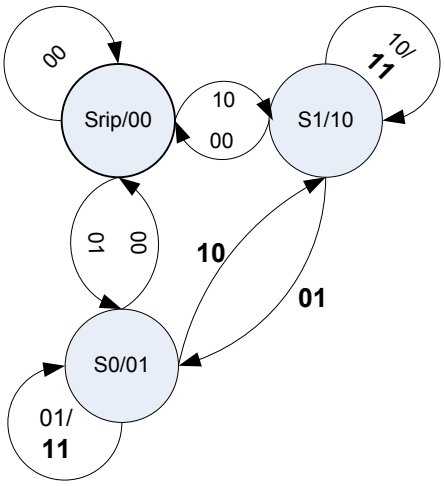
- Partendo da una condizione di stabilità in cui nessun handshake è in corso ($req_1 = ack_1 = 0$, $req_0 = ack_0 = 0$), la rete RS attende che uno dei due moduli indichi l'inizio di un handshake. Supponendo sia M_1 (ovvero req_1 passi a 1), la rete setta ack_1, attende che req_1 venga resettato, quindi chiude l'handshake resettando immediatamente ack_1, così tornando alla condizione di stabilità iniziale. Il comportamento è del tutto analogo nel caso di handshake iniziato da M_0.
- Se, durante l'effettuazione di un handshake con uno dei due moduli - sia M_1 per esempio - l'altro modulo - M_0 nell'esempio - indica l'inizio di un handshake settando req_0, la rete RS pospone l'effettuazione dell'handshake con M_0, mantenendo ack_0 a 0. Quindi, al termine dell'handshake con M_1, pone ack_0 a 1 iniziando l'handshake con M_0. Analogamente scambiando M_1 con M_0, se l'handshake in corso è con il modulo M_0.

Descrivere mediante tabella di flusso e **sintetizzare** la rete RS.

3.3.1 Descrizione

La rete ha due ingressi e due uscite. Disegniamone il comportamento in accordo alle specifiche:

- prima consideriamo gli archi ed i nodi relativi alla prima specifica (gestione di handshake non interallacciati)
- vediamo successivamente (vedi archi in **grassetto**) come aggiustare il tutto per tenere in conto la seconda specifica.



Attenzione: nel passare da S2 a S1 e viceversa dovrò necessariamente cambiare **due uscite "contemporaneamente"**, il che significa che si potrà presentare in uscita uno stato qualunque tra 11 e 00. Non ho nessun motivo per preoccuparmene, visto che le due uscite sono mandate a reti diverse. L'importante è che transiscano entrambe, non mi importa in quale ordine.
La tabella di flusso che emerge da questa descrizione è la seguente:

$req_1\ req_0$	00	01	11	10	$ack_1\ ack_0$
S_{rip}	(S_{rip})	S_0	—	S_1	00
S_1	S_{rip}	S_0	(S_1)	(S_1)	10
S_0	S_{rip}	(S_0)	(S_0)	S_1	01

La tabella di flusso che si ottiene è **normale**, nel senso che ogni transizione di ingresso si risolve in un solo cambiamento di stato.

3.3.2 Sintesi della rete

Devo scegliere le **codifiche** per gli stati, ed il meccanismo di marcatura (latch SR oppure elementi neutri di ritardo).
Visto che ho **tre stati adiacenti**, qualunque sia la codifica che scelgo (su 2 bit) sarà necessario **inserire uno stato ponte tra due stati**.

Adottando le codifiche $\mathbf{S_0 = 00, S_1 = 10, S_2 = 01}$, si rende necessario uno stato ponte fra S_1 e S_2, nel passaggio dello stato d'ingresso da 11 a 01, e fra S_2 e S_1, nel passaggio dello stato d'ingresso da 11

a 10. È immediato verificare che in entrambi i casi lo stato S_0 può fungere da stato ponte. La tabella diventa quindi:

La tabella di flusso **non è normale** in questo caso, in quanto esistono transizioni di ingresso che comportano più di una transizione di stato.

	$req_1\ req_0$ 00	01	11	10	$ack_1\ ack_0$
S_0	(S0)	S_2	–	S_1	00
S_1	S_0	S_0	(S1)	(S1)	10
S_2	S_0	(S2)	(S2)	S_0	01

Con riferimento al modello strutturale con elementi neutri di ritardo, le tabelle di transizione sono:

y_1y_0 \ $req_1\ req_0$	00	01	11	10
00	00	01	--	10
01	00	01	01	00
11	--	--	--	--
10	00	00	10	10

a_1a_0

Una possibile soluzione è $a_1 = req_1 \cdot \overline{y}_0$, $a_0 = req_0 \cdot \overline{y}_1$.

Per la rete RC_B, è immediato verificare che $ack_1 = y_1$, $ack_0 = y_0$.

3.4 Esercizio Gennaio 2015 (sintesi)

Si consideri la rete sequenziale asincrona descritta dalla seguente tabella riportata in Figura:
Tale rete deve essere inserita in un contesto nel quale è indispensabile che non produca mai l'uscita 11, neanche nei transitori.

1. Sintetizzare la rete usando elementi neutri di ritardo come elementi di marcatura. Per RC_A si effettui una sintesi SP.
2. Esprimere la formula del tempo minimo per il quale uno stato di ingresso deve rimanere costante affinché il pilotaggio avvenga in maniera corretta.

x_1x_0	00	01	11	10	z_1z_0
S0	(S0)	S1	(S0)	(S0)	00
S1	S0	(S1)	S2	--	01
S2	--	S1	(S2)	S0	10

3.4.1 Soluzione

La rete in questione ha un ciclo di ordine tre. E' pertanto impossibile scegliere le codifiche in modo tale da evitare corse delle variabili di stato, a meno di non introdurre stati ponte (rendendo in tal modo *non normale* la rete). Visto che la tabella di flusso consente il passaggio da S1 a S2 (e viceversa), con uscite che varierebbero in maniera incontrollata tra 01 e 10, è necessario inserire uno stato ponte S3 tra gli stati S1 ed S2, con uscita 00, in modo da soddisfare il vincolo di progetto.
La tabella di flusso modificata, pertanto, risulta essere quella riportata a destra:

x_1x_0	00	01	11	10	z_1z_0
S0	(S0)	S1	(S0)	(S0)	00
S1	S0	(S1)	S3	--	01
S3	--	S1	S2	--	00
S2	--	S3	(S2)	S0	10

La codifica S0=00, S1=01, S2=10, S3=11, è esente da corse. La rete RC_B risulta essere: $z_1 = y_1 \cdot \overline{y}_0$, $z_0 = \overline{y}_1 \cdot y_0$. Per la rete RC_A, usando elementi neutri di ritardo, abbiamo le seguenti due mappe combinatorie in cui sono evidenziati i sottocubi sufficienti per una copertura senza alee:

y_1y_0 \ x_1x_0	00	01	11	10
00	0	0	0	0
01	0	0	1	-
11	-	0	1	-
10	-	1	1	0

a_1

y_1y_0 \ x_1x_0	00	01	11	10
00	0	1	0	0
01	0	1	1	-
11	-	1	0	-
10	-	1	0	0

a_0

E quindi $a_1 = x_1 \cdot y_0 + x_0 \cdot y_1 \cdot \overline{y}_0 + x_1 \cdot x_0 \cdot y_1$, $a_0 = \overline{x}_1 \cdot x_0 + x_0 \cdot \overline{y}_1 \cdot y_0$

La sintesi di a_1 comprende anche un implicante assolutamente eliminabile, che è necessario ad evitare alee.

2) Visto che il massimo numero di transizioni di stato è pari a 2, il tempo di permanenza di uno stato di ingresso è pari a $T_A + 2 \cdot \left(T_{mark} + T_A \right)$ con $T_{mark} \geq T_{CN1}$.