

Reti Sequenziali

nuovo programma (dall'a.a. 2018/19), versione per studenti

Giovanni Stea

Ultima modifica: 25/09/2020

Prerequisiti

Gli studenti conoscono

- le reti combinatorie e le relative procedure di sintesi;

Versioni:

12/09/2018: prima versione

13/11/2018: aggiunta di una parte sul Verilog (capitolo 2), di esempi di descrizione di RSS complesse (cap. 4) e di sintesi con scomposizione in parte operativa/parte controllo (cap. 4). Correzioni di errori detti a lezione, integrazione di cose dette a lezione. Miglioramento impaginazione e leggibilità.

16/11/2018: correzione di errori e modifiche cosmetiche sul capitolo 4.

12/11/2019: correzioni di errori detti a lezione, modifiche cosmetiche sul capitolo 3, aggiunta parte finale su uso del registro MJR e sottoliste.

22/11/2019: aggiunta di esercizi

Giugno 2020: aggiunta di esercizi, modifiche cosmetiche.

4/8/2020: modifiche cosmetiche

Sommario

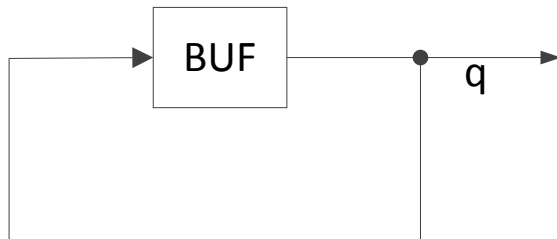
1	La funzione di memoria e le reti sequenziali asincrone.....	5
1.1	Il latch SR	6
1.2	Il problema dello stato iniziale.....	9
1.3	Tabelle di flusso e grafi di flusso	12
1.4	Il D-latch trasparente.....	14
1.5	Il D flip-flop.....	16
1.6	Le memorie RAM statiche.....	19
1.6.1	Montaggio “in parallelo”: raddoppio della capacità di ogni cella	22
1.6.2	Montaggio “in serie”: raddoppio del n. di locazioni.....	22
1.6.3	Collegamento al bus e maschere	23
1.7	Le memorie Read-only	24
1.7.1	ROM programmabili.....	27
2	Il linguaggio Verilog.....	29
2.1	Note sulla sintassi	30
2.2	Descrizione di reti combinatorie	31
2.3	Sintesi di reti combinatorie	33
3	Reti Sequenziali Sincronizzate	34
3.1	Registri.....	34
3.1.1	Descrizione in Verilog di registri.....	36
3.2	Prima definizione e temporizzazione di una RSS	38
3.3	Contatori	42
3.4	Registri multifunzionali	47
3.5	Modello di Moore	49
3.5.1	Esempio: il Flip-Flop JK.....	53
3.5.2	Esempio: riconoscitore di sequenze 11,01,10.....	55
3.5.3	Esercizio – Rete di Moore.....	60
3.5.4	Esercizio (per casa)	61
3.5.5	Esercizio (per casa)	62
3.6	Modello di Mealy.....	62
3.6.1	Esempio: sintesi del contatore espandibile in base 3	66
3.6.2	Esempio: riconoscitore di sequenza 11, 01, 10.....	67
3.6.3	Esercizio.....	69
3.6.4	Soluzione.....	69

3.7	Modello di Mealy ritardato	70
4	Descrizione e sintesi di reti sequenziali sincronizzate complesse	76
4.1	Linguaggio di trasferimento tra registri	76
4.1.1	Esempio: contatore di sequenze alternate 00,01,10 – 11,01,10	80
4.1.2	Esempio: formatore di impulsi con handshake /dav-rfd	81
4.1.3	Esempio: formatore di impulsi con handshake soc/eoc	87
4.2	Sintesi di RSS complesse – scomposizione in “parte operativa” e “parte controllo”	88
4.2.1	Esempio di sintesi: formatore di impulsi con handshake /dav-rfd	96
4.3	Tecniche euristiche di sintesi della parte controllo	100
4.4	Reintrodurre i μ -salti a più vie	102
4.4.1	Esempio di sintesi con uso di registro MJR	104
4.4.2	Sottoliste	105
4.5	Riflessione conclusiva su descrizione e sintesi delle reti logiche	106
5	Soluzioni di esercizi proposti	112
5.1	Soluzione esercizio 3.5.4	112
5.2	Soluzione esercizio 3.5.5	114
6	Altri esercizi svolti	118
6.1	Esercizio – RSS di Moore	118
6.1.1	Soluzione	118
6.2	Esercizio – descrizione e sintesi di RSS complessa	119
6.2.1	Descrizione	119
6.2.2	Sintesi	124
6.3	Esercizio – Calcolo del prodotto con algoritmo di somma e shift	126
6.3.1	Descrizione	126
6.3.2	Sintesi	131
6.4	Esercizio – tensioni analogiche	133
6.4.1	Descrizione	134

1 La funzione di memoria e le reti sequenziali asincrone

Le reti combinatorie sono **prive di memoria**: ad un dato stato di ingresso corrisponde 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)

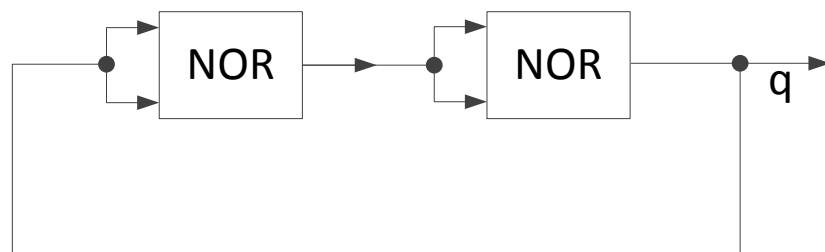
Possiamo dire che l'anello si può trovare **in due stati**, che possiamo chiamare S_0 , S_1 (i nomi sono arbitrari), e che corrispondono allo stato in cui l'uscita vale 0 ed 1 rispettivamente.

Si noti che **la presenza del buffer è fondamentale**, in quanto garantisce che a q sia 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 togliere e riattivare la tensione, e comunque sempre in modo casuale).

Vediamo di complicare un po' lo schema. Posso:

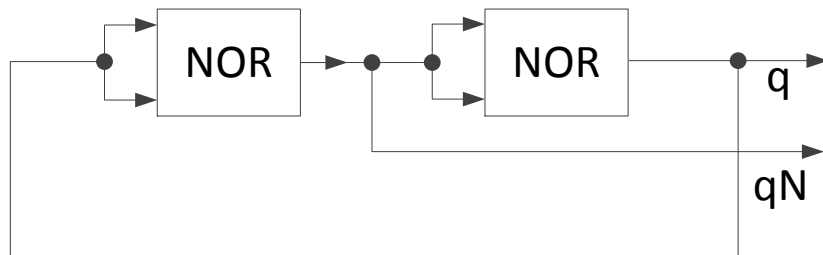
- a) Sostituire il buffer con una coppia di **NOT**
- b) Implementare ciascun NOT a porte **NOR** (per motivi che saranno chiari più avanti)



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

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

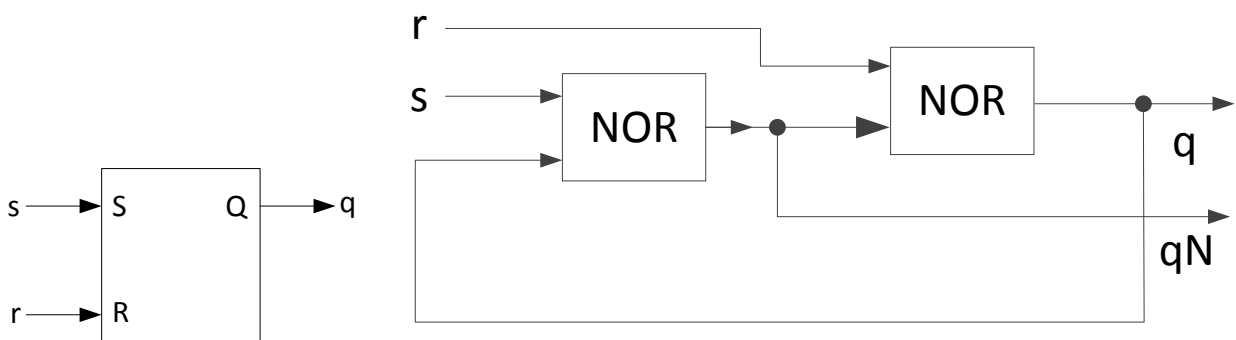


Vediamo cosa succede quando si **accende** questo circuito, connettendolo alla tensione. Se all'accensione **q e qN sono discordi**, la rete si trova già in uno dei due stati stabili, e lì resta. Se, invece, **q e qN sono concordi**, in teoria ciascuna delle due uscite oscilla all'infinito, con un tempo pari al tempo di risposta delle porte. In pratica, invece, la **rete si stabilizza velocemente**, perché comunque il tempo di risposta delle due porte sarà **diverso**, e quindi si creerà immediatamente una situazione in cui **q e qN sono discordi**, situazione 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 di 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:

- $s=1, r=0$:

- 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, l'uscita 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, l'uscita si **resetta**.

- $s=0, r=0$:

- l'uscita della prima porta NOR **vale 0 se $q=1$, e vale 1 se $q=0$** . Pertanto, $qN = \bar{q}$.

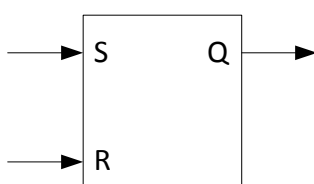
- La seconda porta NOR ha in ingresso **0 e qN** , quindi l'uscita q vale \overline{qN} .

L'uscita, quindi, **conserva il 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. La rete è inoltre **asincrona**, in quanto si aggiorna continuamente (e l'uscita, quindi, cambia in seguito ad una variazione dello stato di ingresso).

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. Esiste, anche per le reti sequenziali asincrone, una misura **analoga al tempo di attraversamento**, dalla quale possiamo desumere quando variare gli ingressi.

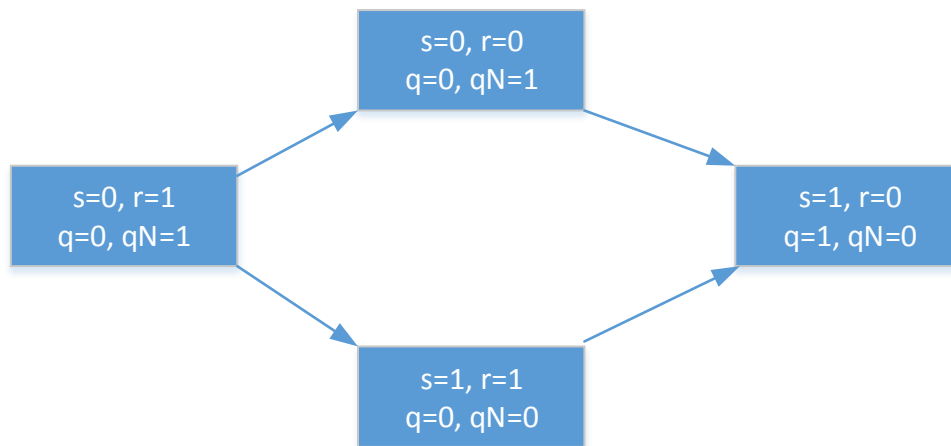
Per quanto riguarda la regola 2), in generale nelle RSA è di **importanza fondamentale**. Infatti, se non la rispetto, possono presentarsi in ingresso degli stati spuri, che mi fanno evolvere la rete in modo del tutto diverso. Però **nel solo caso del latch SR**, posso evitare di rispettarla.

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 EF ed EIP**). Serve quindi un modo per **inizializzare un elemento di memoria** al valore voluto.

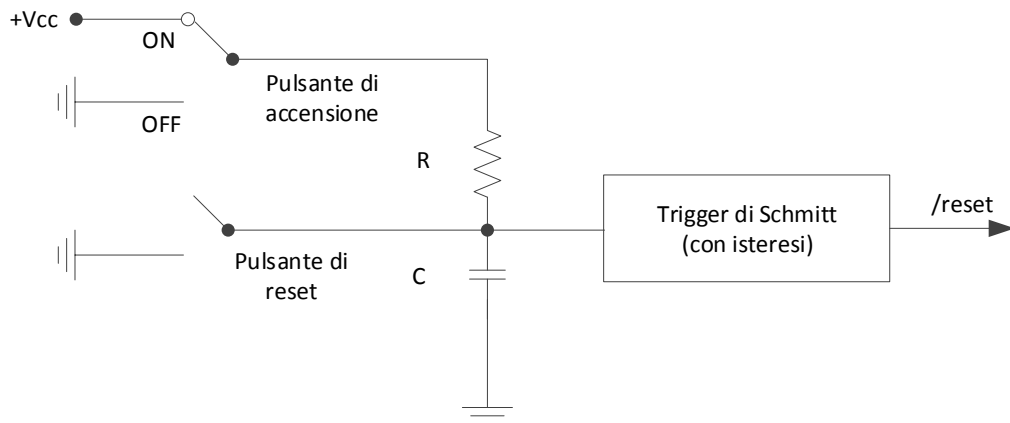
L'inizializzazione avviene **tutte le volte che si preme il pulsante di reset del calcolatore**. Il suo effetto è quello di inserire in tutti gli elementi di memoria (o meglio, in tutti quelli per cui mi interessa avere un valore iniziale) il contenuto iniziale desiderato.

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.

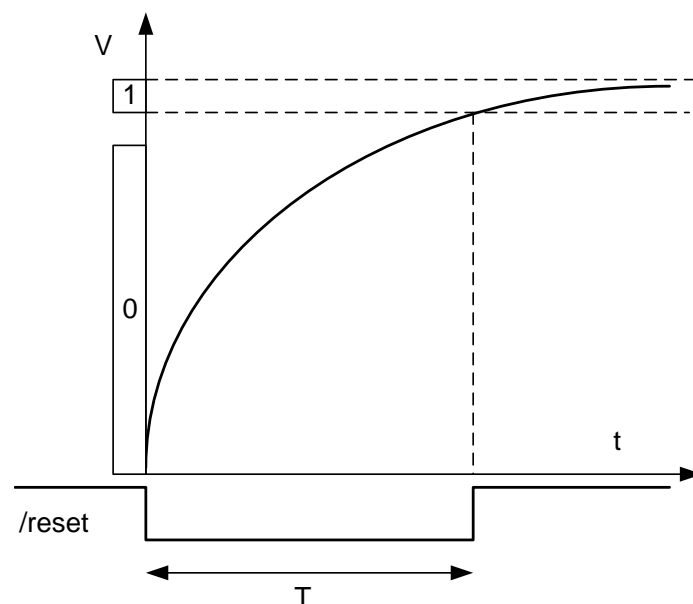
Dal che le persone sono portate a pensare che la condizione iniziale di un calcolatore è quella in cui in tutti gli elementi di memoria c'è scritto zero, che non è assolutamente vero.

Vediamo come è fatta la circuiteria per l'inizializzazione al reset dentro a un sistema di elaborazione.



Il circuito RC sulla sinistra si **carica in tempi dell'ordine del microsecondo**, e la tensione ai capi del condensatore diventa prossima a V_{cc} . 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 underscore** in fondo (nel caso, “**reset_**”).

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

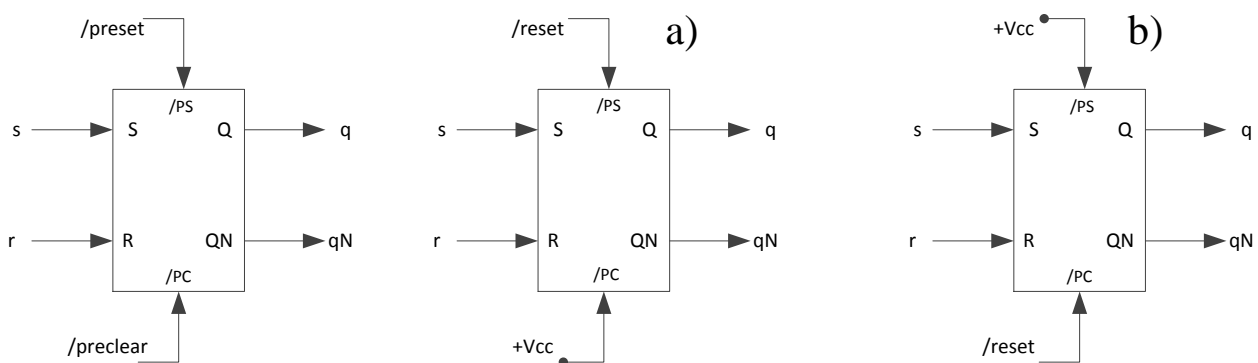
- quando $\text{/reset}=0$, l'elemento di memoria si porta nello stato interno iniziale desiderato, **indipendentemente dal valore dei suoi altri ingressi**.
- quando $\text{/reset}=1$, l'elemento di memoria funziona normalmente.

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

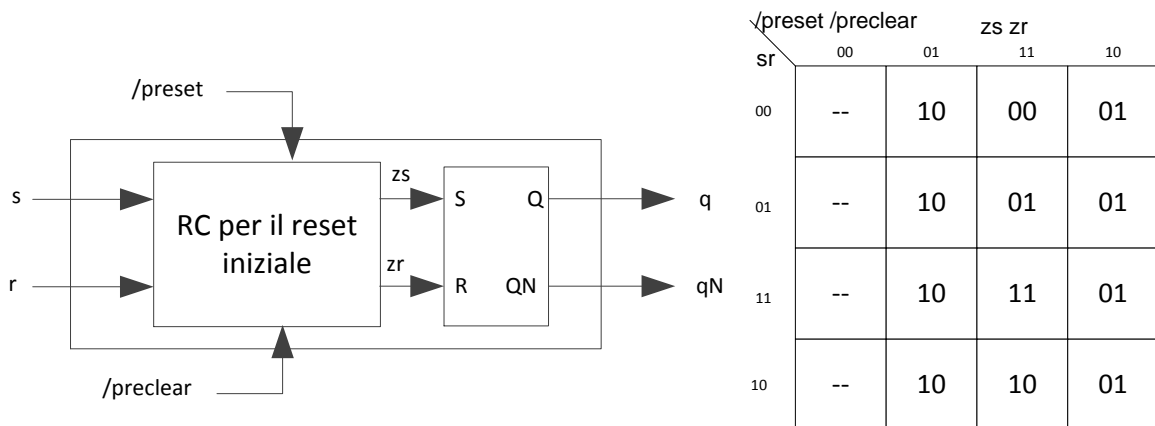
- se $\text{/preset}=\text{/preclear}=1$, la rete si comporta come un latch SR;
- se $\text{/preset}=0$, la rete si porta nello stato S1 (indipendentemente dal valore degli ingressi s e r);
- se $\text{/preclear}=0$, la rete si porta nello stato S0 (indipendentemente dal valore degli ingressi s e r)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

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

- Se si vuole **inizializzare a 1** l'elemento di memoria, si connette /preset a /reset e /preclear a V_{cc} .
- Se si vuole **inizializzare a 0** l'elemento di memoria, si connette /preclear a /reset e /preset a V_{cc} ;



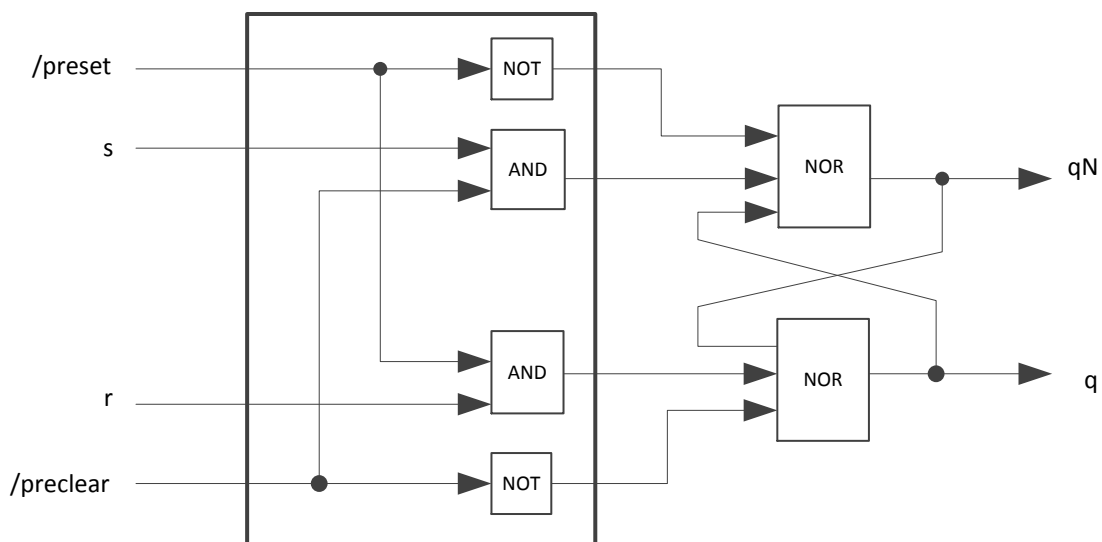
L'unica cosa che manca di fare è capire come **si deve modificare** il latch SR rispetto all'implementazione già vista. Conviene **mettergli davanti una rete combinatoria**, che ha come ingresso s,r, /preset e /preclear , ed in uscita due variabili z_r , z_s , che è facile da sintetizzare in forma SP.



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

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

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**. Possiamo istanziare entrambi i formalismi 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

lo è ininfluyente 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 S0 e l’ingresso s è a 0, rimango nello stato S0 (qualunque cosa faccia r)**”. Posso quindi riempire alcune caselle della tabella.

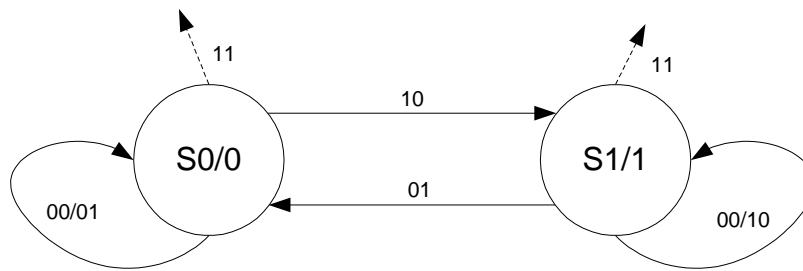
		sr				q
		00	01	11	10	
s0		S0	S0	-	S1	0
s1		S1	S0	-	S1	

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**.

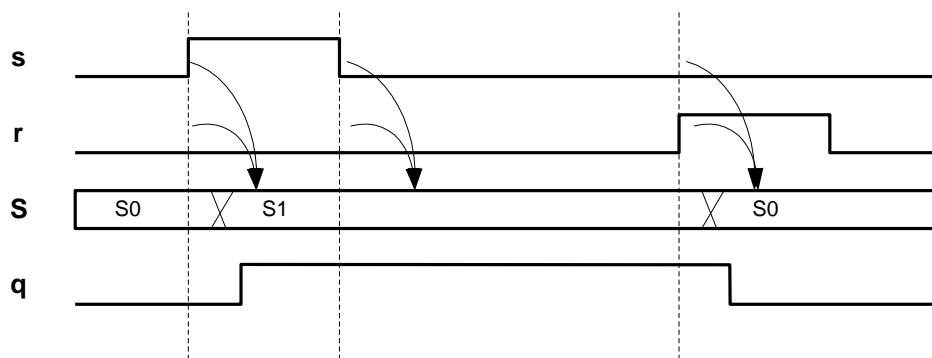
Una RSA **si evolve** (cioè **modifica il proprio stato interno e/o la propria uscita**) a seguito di **cambiamenti dello stato di ingresso**. Non è difficile vedere che, se mi trovo nello stato (ad esempio) **S0** con ingresso 00, la rete **non si evolve** (a meno che non cambi lo stato di ingresso). 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 queglii stati di ingresso. Visto che l'uscita è funzione soltanto dello stato interno, posso scriverla direttamente **dentro il cerchio**.



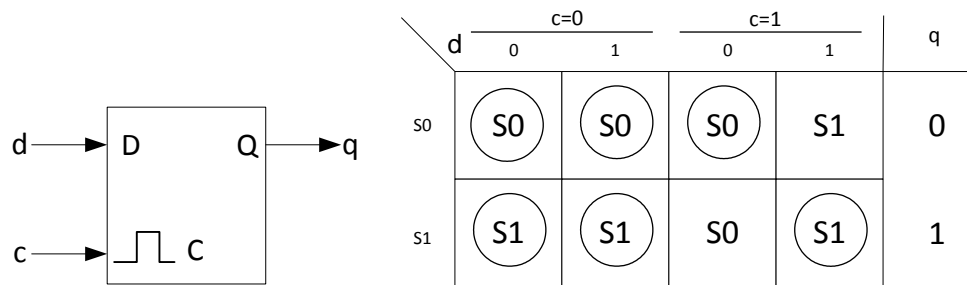
Questo è un **diagramma di temporizzazione**, il quale mi fa vedere che lo stato interno cambia (quando cambia) solo al variare dello stato di ingresso, e poi si stabilizza. L'uscita varia quando varia lo stato interno.

1.4 Il D-latch trasparente

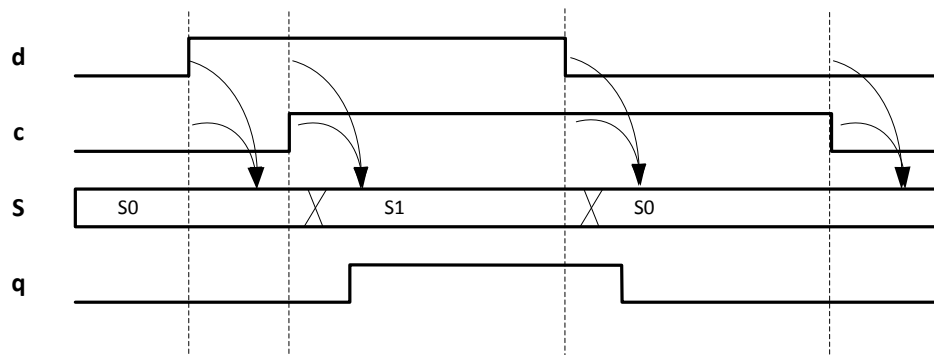
Abbiamo visto che il latch SR può memorizzare 1 o 0 a seconda del comando che gli viene dato (set, reset, conserva).

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.

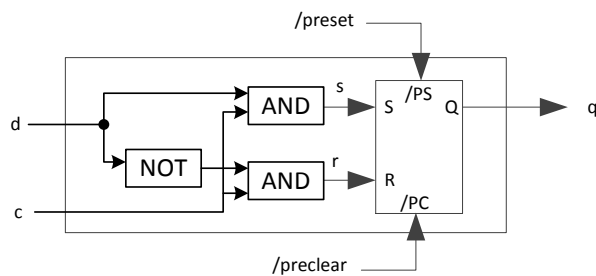
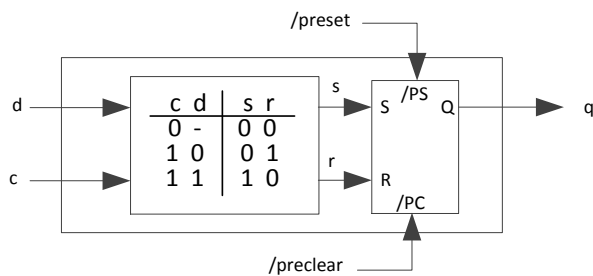
Una sintesi del D-latch si può ottenere facilmente a partire da quella di un latch SR. Supponiamo di avere un latch SR, e mettiamogli davanti una **rete combinatoria** che ha:

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

E cerchiamo di capire come sintetizzare quest'ultima.

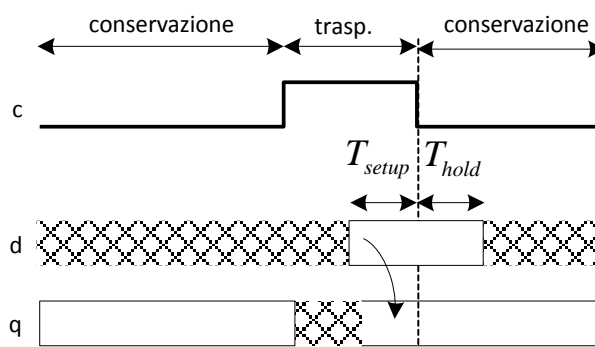
- Quando $c=0$, qualunque sia il valore di d , il D-latch dovrà mantenere l'uscita costante (è in **conservazione**). Pertanto, darò all'SR un comando di *conservazione* $s=0$, $r=0$.
- Quando $c=1$, il D-latch è in **trasparenza**, e quindi l'uscita deve adeguarsi a d . Pertanto,
 - o Se $d=0$, darò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

Scritta la tabella di verità, si vede abbastanza bene che $s = c \cdot d$, $r = c \cdot \bar{d}$. 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 /preset e /preclear 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:

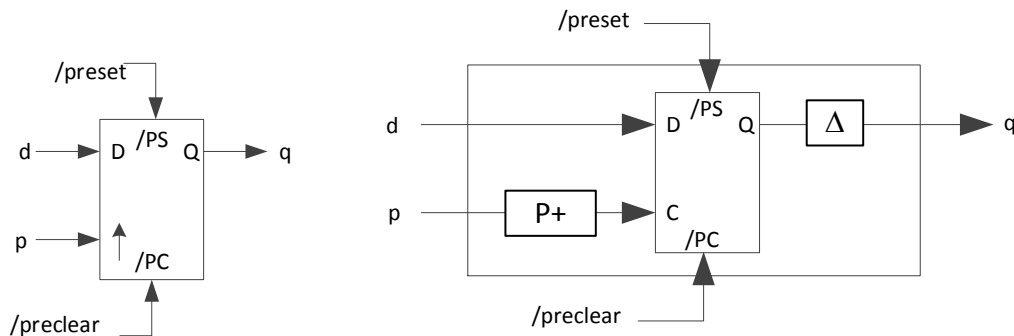
- Reti combinatorie
- latch SR
- D-latch

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 dopo che la rete ha smesso di essere sensibile al valore di d .

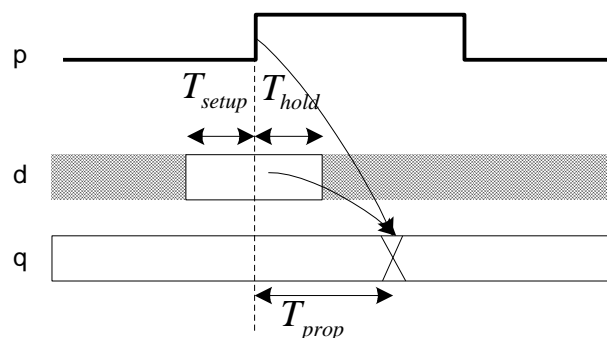
Il **pilotaggio** del D-FF deve avvenire nel rispetto di alcune regole:

- 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.
- 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

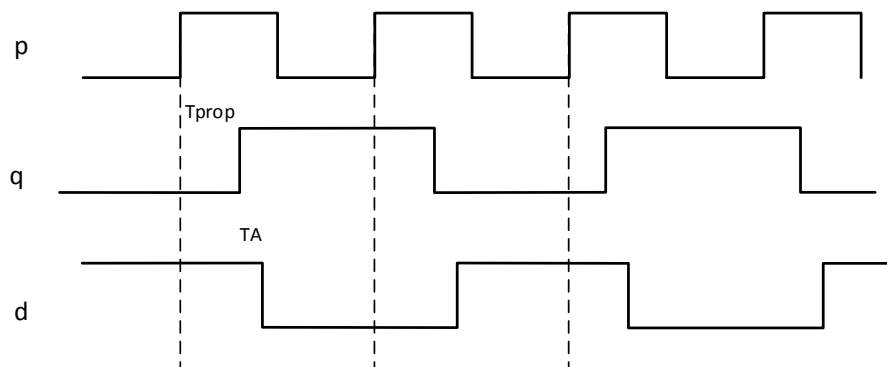
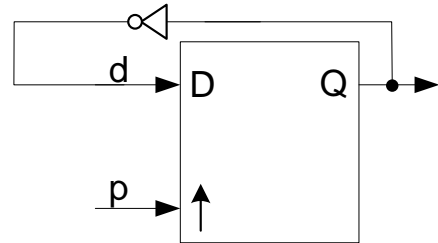
$T_{prop} > T_{hold}$. Quest’ultima disuguaglianza garantisce che la rete è **non trasparente**.

Il tutto si vede bene con un diagramma di temporizzazione:



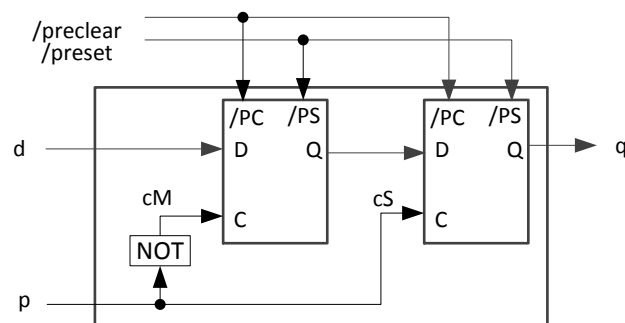
L'uscita di un D-FF **non oscilla mai** (a differenza di quella del D-latch), in quanto viene adeguata in modo **secco** ad un istante ben preciso, e non è mai “direttamente connessa” con l'ingresso d . Ciò comporta che si possono montare i D-FF in qualunque modo si voglia, tanto **non succede niente**.

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.



Nonostante l'uscita sia reazionata sull'ingresso, non ci sono oscillazioni incontrollate.

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



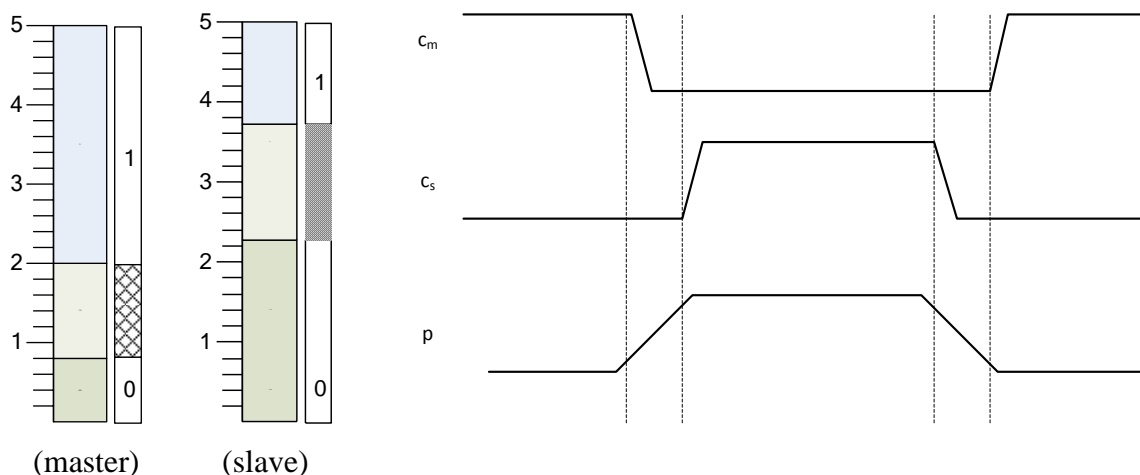
Quando:

- $p=0$, **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 dopo il fronte di salita di p , ma, più o meno contemporaneamente, il master isola l'uscita dall'ingresso.

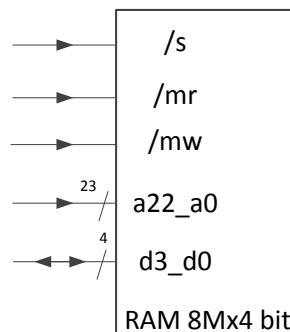
In teoria funziona tutto. Ci possono essere problemi, però, per quanto riguarda il funzionamento **transitorio**. In particolare, può succedere che il master e lo slave siano **contemporaneamente in trasparenza**, anche se per un transitorio. Per evitare questo, normalmente, si agisce per via **elettronica**. 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.



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

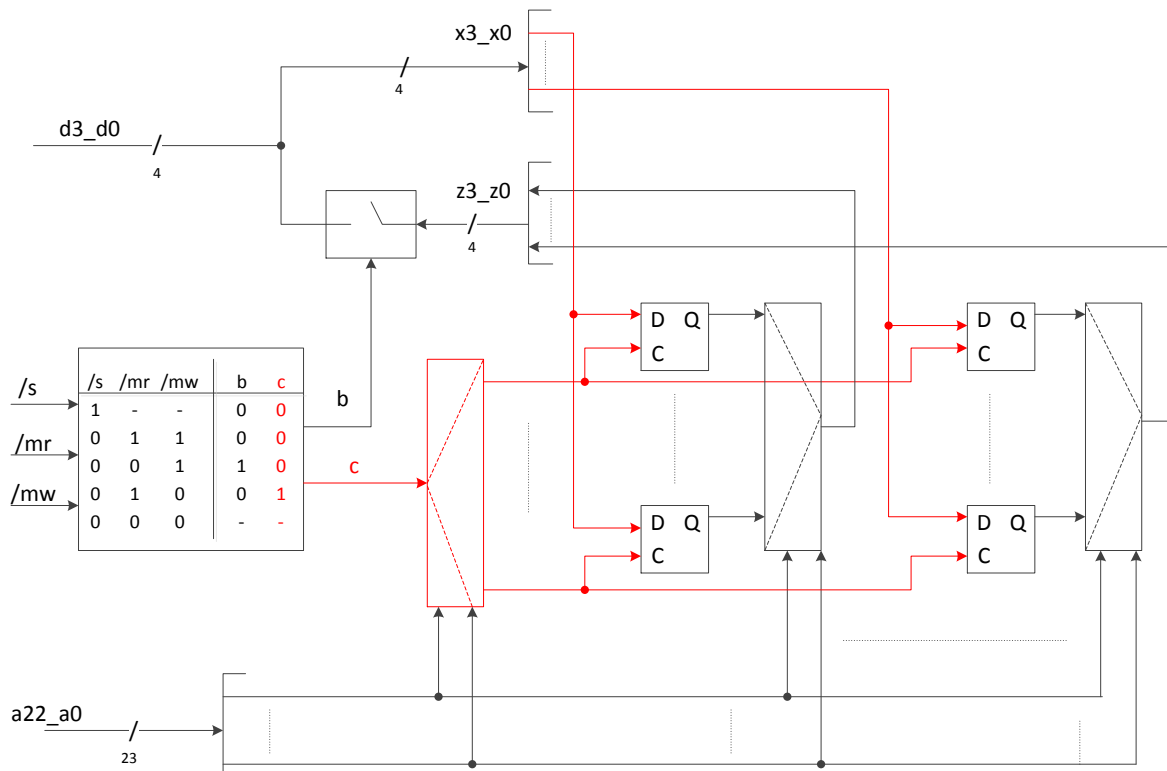
reagisce agli ingressi (fare il parallelo con **ingresso di enabler in un decoder**). Ciò consente di realizzare una memoria “**grande**” (in termini di n. di locazioni) mettendo insieme più banchi di memoria “**piccoli**”. Basta che ne selezioni **uno alla volta**, e poi posso mandare in parallelo tutto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

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 scrittura, 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 scrittura (chiamiamolo **c**) è funzione di $/s$, $/mw$, $/mr$. (disegnare la tabella di verità).

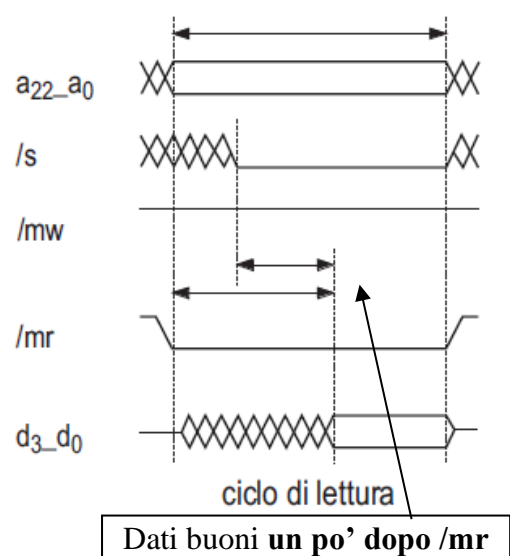


Le RAM statiche sono **molto veloci** (pochi ns di tempo di risposta). Infatti, il loro tempo di attraversamento è quello di **pochi livelli di logica**. Questa tecnologia è usata per realizzare **memorie cache** (le memorie RAM montate nel computer come memoria principale sono RAM **dinamiche**, e sono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

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

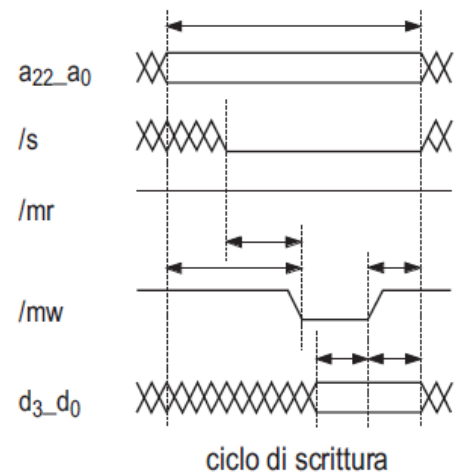
Ad un certo istante, gli indirizzi si stabilizzano al valore della cella che voglio leggere ed arriva il comando di **/mr**. Per motivi che saranno chiari fra un minuto, il comando di **/s** arriva con un po' di ritardo, e **balla** nel frattempo, in quanto è funzione combinatoria di altri bit di indirizzo.

Quando sia **/s** che **/mr** sono a 0, dopo un pochino le porte tri-state vanno in conduzione, e 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.



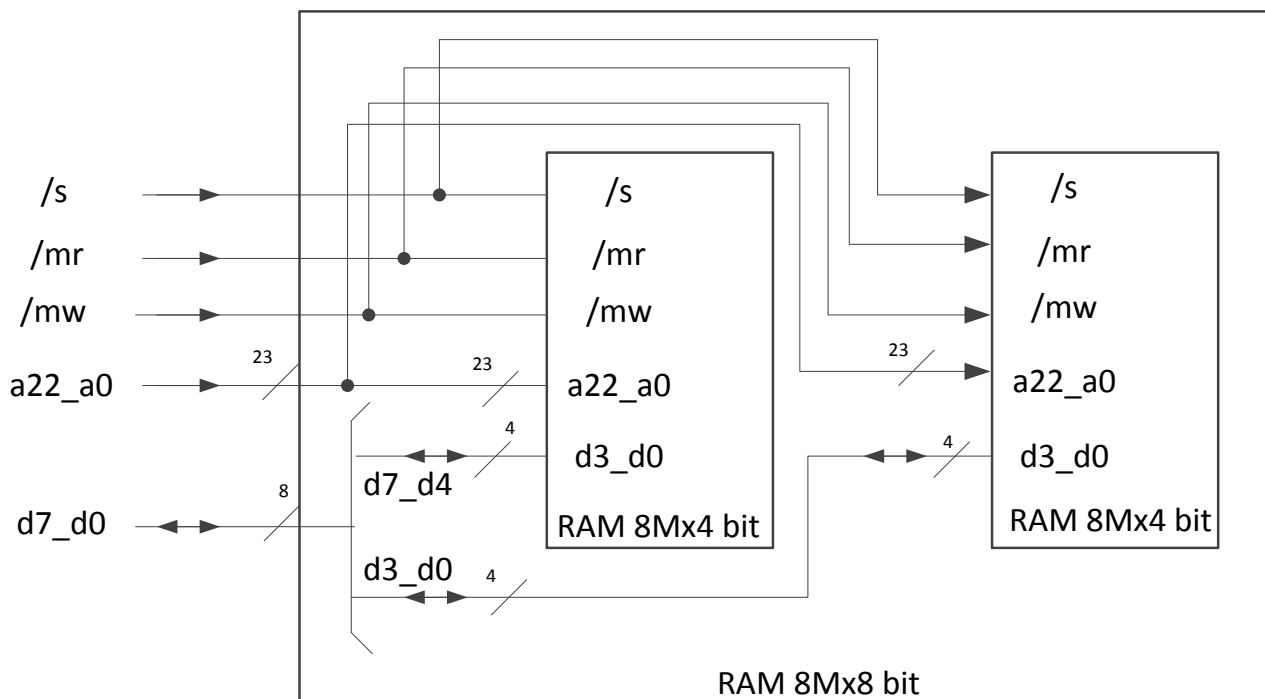
Descriviamo adesso la **temporizzazione** del ciclo di **scrittura** della memoria.

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.



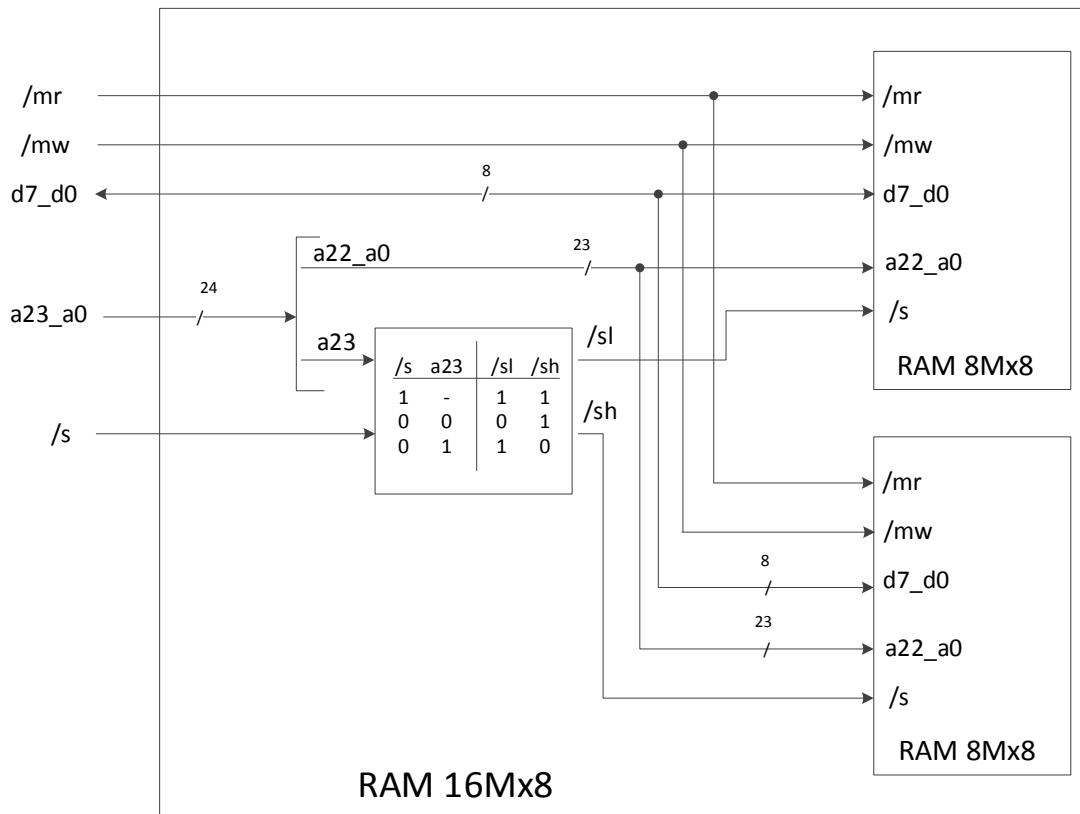
1.6.2 Montaggio “in serie”: raddoppio del n. di locazioni

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

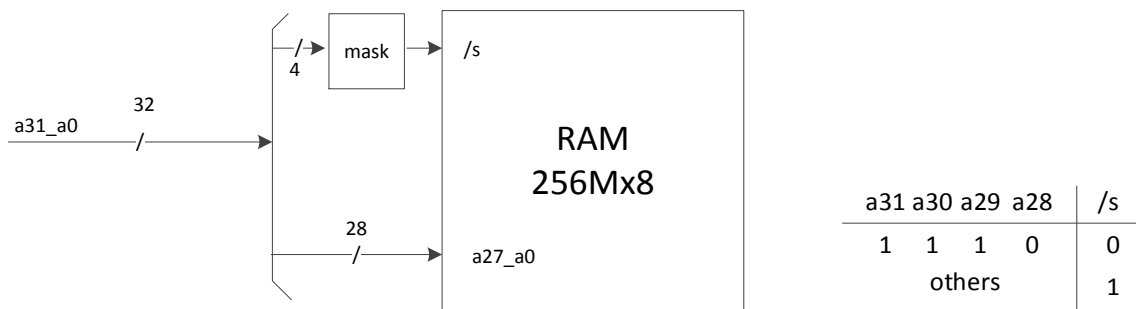
Quindi, si **genera** il segnale di **select** per i due blocchi usando il **valore di a_{23}** . Se il modulo di memoria che si vuole creare deve essere inserito in uno spazio di indirizzamento più grande, fa comodo poter fornire all’esterno un segnale globale di *select*, che dovrà quindi essere messo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



1.6.3 Collegamento al bus e maschere

I fili di indirizzo della memoria provengono da un **bus indirizzi**, dove il processore (e, talvolta, altri moduli) ne impostano il valore. Il piedino */s* di un modulo di RAM serve appunto a poter **realizzare uno spazio di memoria grande usando moduli di memoria più piccoli**. Ad esempio, supponiamo di avere un bus indirizzi a **32 bit** (capace, quindi, di indirizzare 2^{32} celle di memoria), e di voler montare un modulo di RAM 256Mx8 bit **a partire dall’indirizzo 0xE0000000**. Il modulo avrà 28 fili di indirizzo ($2^8=256$, $2^{20}=1M$), ed un filo di select */s*, e dovrà rispondere agli indirizzi nell’intervallo **0xE0000000--0xFFFFFFFF**. Affinché ciò succeda:

- I 28 fili di indirizzo meno significativi del bus andranno in ingresso al modulo di RAM.
- I restanti 4 fili di indirizzo più significativi andranno in ingresso ad una **maschera**, che genera il select per il modulo di RAM.



La maschera deve riconoscere la configurazione di bit richiesta (0xE=B1110). Pertanto, deve essere

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

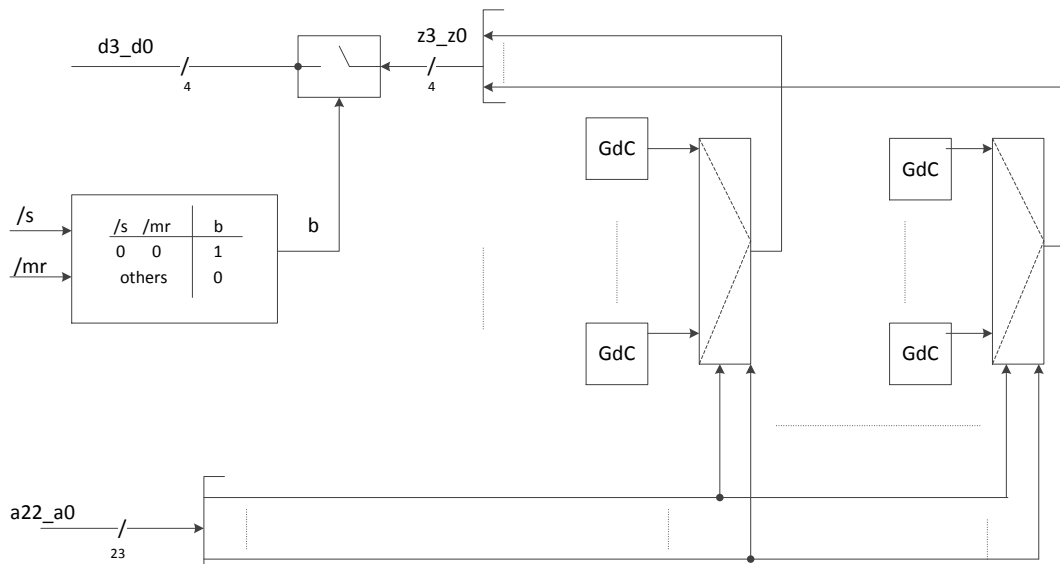
Chi progetta la maschera? Colui che assembla il sistema. Progettare la maschera significa decidere a che intervallo di indirizzi saranno associate le celle di un modulo di RAM. Quanto appena scritto giustifica il fatto che il filo di **select** normalmente si stabilizza **con un certo ritardo rispetto ai fili di indirizzo**, perché è comunque funzione di (altri) fili di indirizzo.

Come ultima nota, osserviamo che quelli visti finora 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).



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).

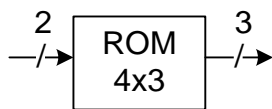
Il comando di lettura, unito al comando di select, mette in **conduzione le tri-state**, consentendo ai generatori di costante della riga selezionata dagli indirizzi di inserire sul bus il contenuto della cella.

Se si eccettua la presenza delle porte tri-state, una memoria ROM di 2^N celle di M bit ciascuna è una **rete combinatoria, con N ingressi ed M uscite**. Infatti, ad ogni possibile stato di ingresso (2^N possibili) deve corrispondere sempre lo stesso stato di uscita (contenuto della cella di memoria, M bit). Vediamo un esempio.



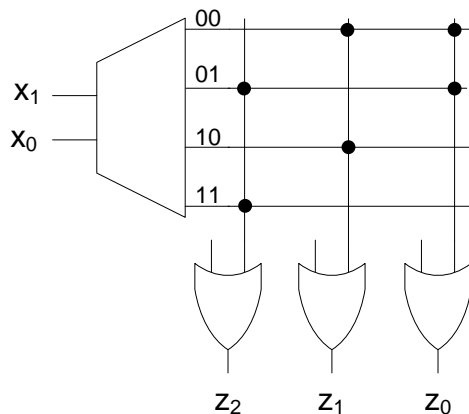
Dal punto di vista logico, una ROM di 2^N celle di M bit ciascuna è una **rete combinatoria, con N ingressi ed M uscite**. Infatti, ad ogni possibile stato di ingresso (2^N possibili) deve corrispondere sempre lo stesso stato di uscita (contenuto della cella di memoria, M bit). Vediamo un modo diverso di sintetizzarla.

Esempio: memoria 4x3



x_1	x_0	z_2	z_1	z_0
0	0	0	1	1
0	1	1	0	1
1	0	0	1	0
1	1	1	0	0

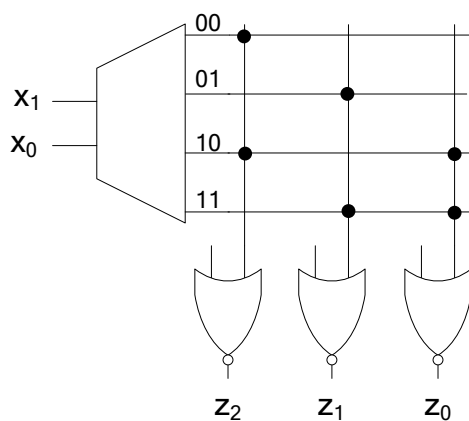
Come realizzo una simile rete? Facendo riferimento al **modello strutturale universale** descritto in precedenza. Bisogna che l'uscita z_0 riconosca gli stati 00 e 01, etc... Quindi:



Si connettono le uscite del decoder alla porta OR quando la cella che corrisponde a quella porta ha il corrispondente bit a 1, e non si connettono se il bit è a zero.

In generale, quindi, una ROM è un decoder N to 2^N , ed una batteria di M porte OR. Il contenuto della ROM è dato da come connetto le porte OR alle uscite del decoder.

Posso pensare di usare come stadio finale anche delle **porte NOR**.

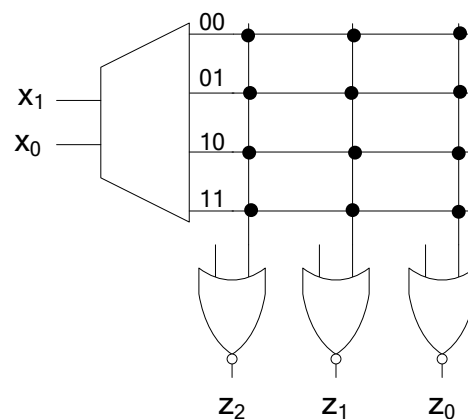


Si connettono le uscite del decoder alla porta NOR quando la cella che corrisponde a quella porta ha il corrispondente bit a 0, e non si connettono se il bit è a 1.

Una ROM è realizzata su un singolo chip di silicio, e deve uscire dalla fabbrica **già programmata** (cioè con i contatti già stabiliti), in quanto il processo di programmazione è parte integrante del processo di fabbricazione del chip. Visto che preparare lo “stampo” per una ROM ha un costo fisso molto elevato, la realizzazione di una ROM si giustifica soltanto con scale molto larghe, nell’ordine delle centinaia di migliaia di pezzi. È chiaro che dovrò trovare qualcosa di alternativo per le basse tirature, perché potrei non di meno aver bisogno di memorie non volatili. Vediamo le possibili alternative.

1.7.1 ROM programmabili

Pensiamo alla seguente possibilità: fornisco uno schema del genere, in cui i contatti tra le uscite degli AND e gli ingressi dei NOR ci sono tutti. Ciò vuol dire che il contenuto di ogni cella è **zero**. Se però posso “**bruciare**” qualcuno di questi contatti, posso **programmare la ROM** perché ciascuna cella contenga un contenuto arbitrario.



Bruciare un contatto significa mettere un bit ad 1.

- **PROM** (Programmable ROM). La matrice di connessione è fatta da **fusibili**, che possono essere fatti saltare in modo selettivo in modo da inserire in ciascuna cella il valore desiderato. Il chip viene venduto con tutti i fusibili a posto, e viene **successivamente** programmato dall’utente. È chiaro che la programmazione è **distruttiva**, non può cioè essere ripetuta.
- **EPROM** (Erasable Programmable ROM) Le connessioni sono fatte non con fusibili, ma con dispositivi elettronici (**field-effect transistors**), che sono programmabili per via elettrica e cancellabili tramite esposizione a raggi ultravioletti. Possono pertanto essere **cancellate** e riprogrammate più volte.
 - Probabilmente qualcuno ha già visto un chip EPROM su una scheda del PC. Avrete fatto caso che hanno un “buco” sul dorso, tappato da un adesivo. È infatti attraverso quel foro che si cancellano, sottoponendole ai raggi ultravioletti. Ovviamente, per cancellarle bisogna toglierle da dove sono. Le EPROM si programmano con un apposito **programmatore di EPROM**.

La scarica di una EPROM prende qualche minuto (una decina, se sottoposta a lampada ad ultravioletti), ed è non selettiva. Non può essere fatta un numero infinito di volte, ed i dati che vengono memorizzati si degradano nel tempo, anche se molto lentamente:

- **endurance**: capacità di sopportare riprogrammazioni (nell'ordine delle 10K-100K volte)
- **data retention**: periodo per il quale si può far affidamento sul contenuto di una EPROM (nell'ordine dei 10-100 anni)
- **EEPROM (E²PROM)**: (Electrically Erasable Programmable ROM). Possono essere programmate e cancellate tramite **segnali elettrici** appositi (**diversi** da quelli del normale funzionamento a regime, ovviamente). Quindi possono essere riprogrammate direttamente **on chip**. Esempio: quando cambiate le impostazioni del BIOS, le salvate su un dispositivo dove:
 - In assenza di tensione vengono mantenute
 - Le potete modificare successivamente.

Tale dispositivo è appunto una EEPROM.

Anche per questi esistono parametri simili a quelli delle EPROM (data retention, endurance). A ben guardare, una EEPROM è un dispositivo programmabile. Lo si continua a chiamare ROM (invece che RAM) in quanto la programmazione è un modo operativo differente da quello della normale attività della memoria. Infatti:

- il numero di volte in cui si può riprogrammare una EEPROM è comunque limitato
- il tempo che ci vuole a riprogrammare una EEPROM è molto maggiore (ms) del tempo che ci vuole a leggerla
- le **tensioni** che si usano non sono le stesse (12-18V, contro 5)
- le memorie continuano ad essere non volatili.

2 Il linguaggio Verilog

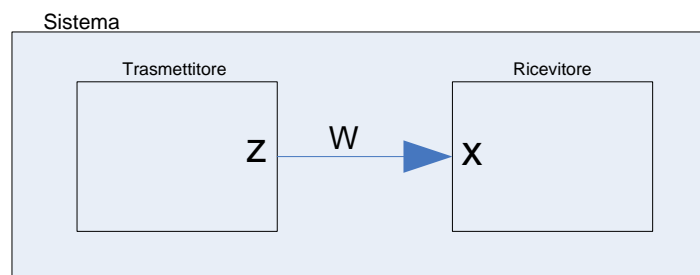
Per **descrivere** le reti logiche, soprattutto quelle **complesse**, fa comodo poter adottare una notazione **testuale**. Quest'ultima ha il pregio di essere

- **Compatta** (quindi facile da trasportare)
- **Interpretabile in modo automatico da una macchina**, che quindi può **simulare** il comportamento della rete logica descritta in modo testuale, dati certi ingressi alla medesima.

Esistono numerosi **linguaggi di descrizione** delle reti logiche. Uno di questi è il **Verilog**, che useremo d'ora in avanti.

Il Verilog consente di fare **molte più cose** di quelle che vediamo, e consente di farle in **molti modi differenti**. Noi adotteremo un particolare stile, funzionale ai compiti che dobbiamo svolgere, e quindi introdurremo il Verilog in modo **informale**, attraverso **esempi** (invece che specificandone sintassi e semantica formalmente).

Prendiamo il primo esempio visto all'inizio del corso, e scriviamolo in Verilog:



```
module Sistema;
```

```
  wire w;
```

```
  Trasmettitore T(w);
```

```
  Ricevitore R(w);
```

```
endmodule
```

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

Wire = "filo", cioè variabile logica.

```
module Trasmettitore(z);
```

```
  output z;
```

```
  //descrizione della struttura interna
```

```
endmodule
```

Rete di tipo "Trasmettitore", che ha una variabile di I/O che riferisco come "z" all'interno di questa descrizione.

```
module Ricevitore(x);
```

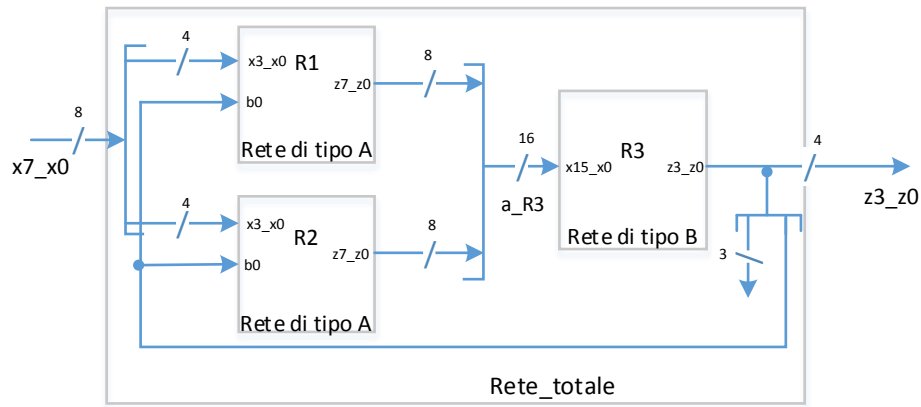
```
  input x;
```

```
  //descrizione della struttura interna
```

```
endmodule
```

La variabile z è un'uscita per il modulo "Trasmettitore"

Vediamo adesso un esempio più complicato:



In questo esempio ci sono **tre sottoreti**, di due tipi diversi, collegate in varia maniera, all'interno di una rete "top level" che ha ingressi ed uscite con il mondo esterno. Gli ingressi e le uscite sono in genere **a più di un bit**.

Cominciamo dalle sottoreti (si capisce meglio):

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

Costrutto che indica il **numero di bit** associato ad un identificatore (se omesso vale uno).

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

Un modo semplice di descrivere il modulo "top level" è il seguente (ne esistono altri, del tutto equivalenti):

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

Posso riferire **un intervallo di bit** (contigui) in una variabile logica a più componenti, o un singolo bit

2.1 Note sulla sintassi

Il Verilog è **case sensitive**. In particolare, le parole chiave (module, endmodule, etc.) sono tutte minuscole, e come tali vanno scritte.

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

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

$$n' \{B, D, H\} \text{valore}$$

dove:

- n è il numero di **bit** su cui va intesa la costante (qualunque sia il formato in cui viene scritta). Può essere omesso, nel caso viene ricavato dalla dimensione delle variabili in gioco.
- B, D, H indica che il resto della costante va interpretato in base 2, 10 16
- Il resto contiene il valore.

Ad esempio: **8'D32** indica la costante 32 (espressa in decimale), rappresentata su 8 bit (00010000)

Le costanti binarie si possono scrivere anche raggruppando i bit a gruppi di quattro ed inserendo underscore. Ad esempio: **16'B0010_1000_0101_1100** (per leggibilità).

Attenzione: **10 è la base di default**. Scrivere **h=0101** significa scrivere che h vale centouno (eventualmente troncato sul numero di bit su cui la variabile h è definita). Se h è una variabile a 4 bit, bisogna scrivere $h='B0101$.

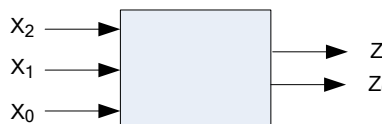
In Verilog è possibile assegnare ad una variabile quattro valori:

- **Uno:** 1'B1, o semplicemente 1 (che è decimale, ma vale uno lo stesso)
- **Zero:** 1'B0, o semplicemente 0 (che è decimale, ma vale zero lo stesso)
- **Alta impedenza:** 1'BZ
- **Non specificato:** 1'BX (da usare soltanto per valori di uscita)

I **commenti** si racchiudono (come in C++) tra `/*...*/`, oppure tra `//` e fine riga. Ciascuna riga va terminata con un punto e virgola.

2.2 Descrizione di reti combinatorie

Vediamo come si **descrive** una rete combinatoria in Verilog. Lo si fa trasferendo la **tabella di verità** nella maniera più pedissequa possibile. Ad esempio:



x2	x1	x0	z1	z0
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	0	0
1	1	1	0	0

```

module RC(z1, z0, x2, x1, x0);
input x2, x1, x0;
output z1, z0;
assign #T {z1, z0} = ({x2, x1, x0} == 'B000') ? 'B00 :
                    ({x2, x1, x0} == 'B001') ? 'B01 :
                    ({x2, x1, x0} == 'B010') ? 'B10 :
                    ({x2, x1, x0} == 'B011') ? 'B10 :
                    ({x2, x1, x0} == 'B100') ? 'B11 :
                    ({x2, x1, x0} == 'B101') ? 'B11 :
                    ({x2, x1, x0} == 'B110') ? 'B00 :
                    ({x2, x1, x0} == 'B111') ? 'B00 ;

endmodule

```

Posso raggruppare **più variabili logiche** in un unico costrutto, mettendole tra parentesi graffe.

Commenti

- **assign**: assegnamento **continuo**. Significa: **in ogni** istante temporale, ciò che sta a sinistra viene adeguato a ciò che sta a destra. È ciò che succede in una **rete asincrona**.
- **#T**: **ritardo di assegnamento**. Serve a modellare il **tempo di attraversamento** (sostituendo a T una costante). Fa comodo quando si simula una descrizione, perché consente di vedere l'uscita che cambia con un po' di ritardo rispetto all'ingresso.
- Costrutto condizionale (come in C, dove si chiama "if aritmetico"): **(...)?...:...**

Ci sono **mille altri modi** di scrivere la stessa cosa:

- Uso di un valore di **default**. "default" non è una parola chiave che si può usare in un costrutto condizionale, ma possiamo comunque omettere (ad esempio) i casi in cui l'uscita è 'B00 e metterli tutti in fondo.

```

assign #T {z1, z0} = ({x2, x1, x0} == 'B001') ? 'B01 :
                    ({x2, x1, x0} == 'B010') ? 'B10 :
                    ({x2, x1, x0} == 'B011') ? 'B10 :
                    ({x2, x1, x0} == 'B100') ? 'B11 :
                    ({x2, x1, x0} == 'B101') ? 'B11 :
                    /*default*/           'B00 ;

```

- Uso di **functions**. Una function in Verilog è una **rete combinatoria** (non è un **linguaggio di programmazione**).

In particolare, posso scrivere:


```

module RC(z1, z0, x2, x1, x0);
  input x2, x1, x0;
  output z1, z0;
  assign #T {z1,z0}=F(x2,x1,x0);

  function [1:0] F;
    input x2,x1,x0;
    casex({x2,x1,x0})
      'B000 : F='B00;
      'B001 : F='B01;
      'B010 : F='B10;
      'B011 : F='B10;
      'B100 : F='B11;
      'B101 : F='B11;
      'B110 : F='B00;
      'B111 : F='B00;
    endcase
  endfunction
endmodule

```

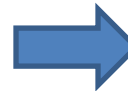
Posso usare “?” per dire che il valore del bit che non specifico è irrilevante



```

      'B001 : F='B01;
      'B011 : F='B10;
      'B010 : F='B10;
      'B100 : F='B11;
      'B101 : F='B11;
      default : F='B00;

```



```

      'B001 : F='B01;
      'B01? : F='B10;
      'B10? : F='B11;
      default : F='B00;

```

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

2.3 Sintesi di reti combinatorie

In Verilog si possono anche scrivere **sintesi** di reti combinatorie. In particolare, nei costrutti **assign** a destra dell’uguale si possono inserire **espressioni logiche**, contenenti gli operatori AND, OR, NOT, XOR, che si scrivono **&**, **|**, **~**, **^**. Questo consente di scrivere, ad esempio, uscite RC sintetizzate in forma SP. Per esempio, per la rete combinatoria che abbiamo scritto all’inizio, abbiamo che la sintesi SP a costo minimo è $z_1 = \overline{x_2} \cdot x_1 + x_2 \cdot \overline{x_1} = x_2 \oplus x_1$, $z_0 = x_2 \cdot \overline{x_1} + x_0 \cdot \overline{x_1}$. Quindi, potrò scrivere in Verilog:

```

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

```

Si noti la differenza (fondamentale) tra **descrizione** e **sintesi**. Una descrizione dice **che cosa fa** la rete. Una tabella di verità è una descrizione, e la posso riportare in Verilog in uno dei modi scritti sopra. Lo scopo di una descrizione è di essere **verificabile** (da un utente umano o da una macchina). Una sintesi dice **quali porte ci metto** affinché succeda quello che c’è scritto nella descrizione. Quello che c’è scritto sopra è una **sintesi**, e non una **descrizione**. Per una rete combinatoria è abbastanza semplice ricavare la **descrizione dalla sintesi** (cioè la tabella di verità dal circuito), ma per le reti sequenziali complesse è praticamente impossibile. Il fatto che in Verilog si possano scrivere **sia descrizioni che sintesi** confonde le idee agli studenti. Le due cose vanno tenute distinte in testa.

3 Reti Sequenziali Sincronizzate

Le RSS sono quelle che si evolvono **soltanto in corrispondenza di istanti temporali ben precisi**, detti appunto **istanti di sincronizzazione**. Tali istanti devono essere opportunamente distanziati (non possono essere troppo ravvicinati). Non sono i **cambiamenti di ingresso** che fanno evolvere una RSS, come era invece per le RSA: le RSS si evolvono **all'arrivo del segnale di sincronizzazione**.

Come si **realizza fisicamente** la sincronizzazione? Portando alle reti un **segnale di ingresso particolare, detto clock**. Tale segnale scandisce, con le sue transizioni, la sincronizzazione della rete.



Il clock ha, normalmente, una forma d'onda **periodica**, di **frequenza nota** $1/T$. Non necessariamente il **duty-cycle** τ/T è del 50%, ma non può essere troppo piccolo (né, ovviamente, troppo grande). L'evento che **sincronizza** la rete che riceve questo segnale è, normalmente, il **fronte di salita del clock**.

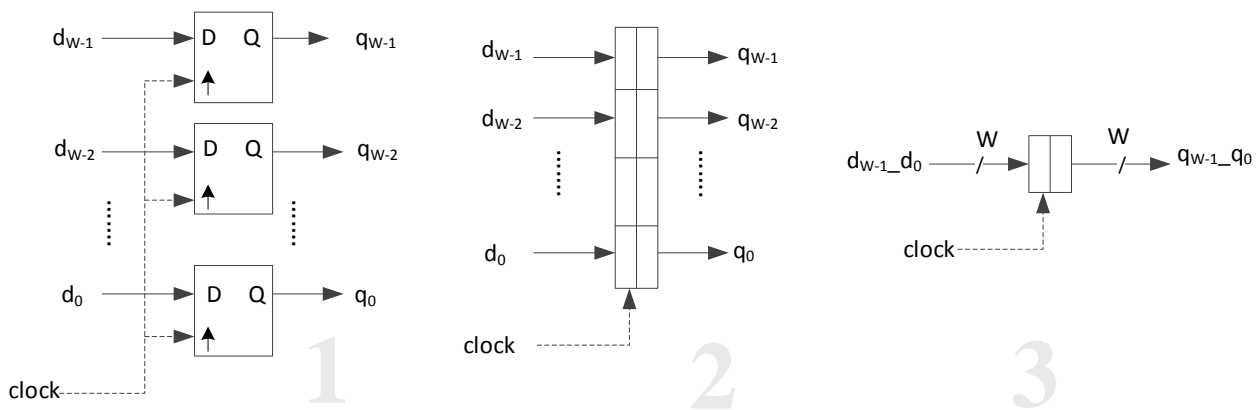
3.1 Registri

Definisco un **registro a W bit** come una **collezione di W D-Flip-Flop Positive-Edge-Triggered**, che hanno:

- a) ingressi d_i ed uscite q_i separati (cioè indipendenti)
- b) **ingresso p a comune**

Un **registro può essere visto come una rete sequenziale sincronizzata**, in cui l'ingresso p funge da **segnale di sincronizzazione**.

Quindi, d'ora in poi, useremo i registri (basati su D-FF) come **elemento base** per la sintesi di RSS. Pur essendo il D-FF una rete sequenziale **asincrona**, se considero i due ingressi d e p nella loro generalità, niente mi vieta di attribuire all'ingresso p un **valore speciale**, appunto quello di **segnale di sincronizzazione**, e vedere il D-FF come una rete sequenziale **sincronizzata**.



Visto che p non specifica più, in quest'ottica, **un valore di ingresso, posso smettere di annoverarlo tra gli ingressi**: non mi interessa, infatti, il suo valore, ma soltanto l'istante in cui transisce da 0 ad 1. Dirò, d'ora in avanti, che il registro a W bit **ha W ingressi e W uscite**, sottintendendo che ha anche un ulteriore ingresso di clock, dedicato però a portare il segnale di sincronizzazione. Lo stato di uscita del registro (W bit, detti **capacità** del registro) ad un certo istante verrà anche chiamato **contenuto** del registro stesso in quell'istante. L'utilizzo di tale contenuto (ad esempio per fornire ingresso ad una rete combinatoria) verrà detto **lettura del registro**. La memorizzazione dei W bit in ingresso ad un certo istante di sincronizzazione verrà detta **scrittura** del registro.

Infine, se mi interessa impostare un **valore iniziale** per il registro, collegherò i piedini **/preset** e **/preclear** di ciascun D-FF alla variabile di /reset o ad 1 in modo da impostare lo stato desiderato.

L'unico requisito di **pilotaggio** per un registro è che gli ingressi d si mantengano **stabili** intorno al fronte di salita del clock, per un tempo T_{setup} prima e T_{hold} dopo. L'uscita, come sappiamo, cambia dopo $T_{prop} > T_{hold}$. **Tutto ciò che accade ai suoi ingressi al di fuori di questo intervallo è irrilevante, e non verrà memorizzato**. Posso montare un registro nei modi più barbari senza che si perda la prevedibilità dell'evoluzione del suo stato.

È fondamentale capire **bene** che **i registri memorizzano il proprio stato di ingresso al fronte di salita del clock**. Il fatto che **due stati di ingresso ai registri**, presentati su istanti di clock (fronti di salita) **consecutivi**, siano **identici, adiacenti o non adiacenti non riveste alcuna importanza**.

Quindi:

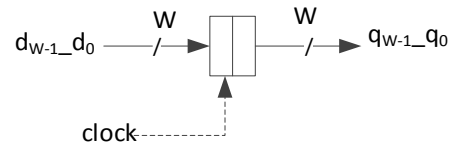
Tra due fronti di salita del clock, **lo stato di ingresso ai registri può cambiare in qualunque modo (o non cambiare affatto)**. Al nuovo fronte di salita del clock, lo stato di ingresso presente verrà memorizzato (come se fosse un nuovo stato, anche se identico al precedente).

Inoltre, le **uscite cambiano T_{prop} dopo il fronte di salita del clock, e restano costanti per tutto un periodo**.

3.1.1 Descrizione in Verilog di registri

È il caso di iniziare ad utilizzare il linguaggio **Verilog** per descrivere reti sequenziali sincronizzate. Lo facciamo in parallelo ad **altri** formalismi (tipo tabelle di flusso, etc.) perché finiremo a descrivere RSS di **notevole complessità**, per le quali gli altri formalismi sono assolutamente inefficienti (pensate ad **una rete con 50 stati e 20 ingressi**, e vedete se con le tabelle ve la cavate). Non che tale linguaggio fosse inadatto a descrivere, ad esempio, le reti combinatorie o le RSA. Però finora ce l'abbiamo fatta senza, e tanto bastava.

Descriviamo in Verilog questa semplice rete:



Var **attiva bassa** (non posso mettere "/" nel nome

e di un registro da W bit di tipo reg delle variabili clock e reset_ da usarsi per l'impostazione dello stato interno iniziale.

Dichiarazione registri (reg) e fili (wire)

// Dichiarazione di due variabili a W bit, dw-1_d0 e qw-1_q0 da usarsi, // rispettivamente, come variabile di ingresso e come variabile di uscita // e impostazione di quest'ultima come effettiva variabile di uscita

reg [W-1:0] REGISTRO;

wire clock, reset_;

wire [W-1:0] dw-1_d0;

wire [W-1:0] qw-1_q0; **assign** qw-1_q0=REGISTRO;

Blocco assign - assegnamento continuo (aggiornamento uscite)

// Immissione nel registro del contenuto_iniziale al reset_ // della variabile dw-1_d0 all'arrivo di ogni segnale di sincronizzazione

always @(reset_==0) #1 REGISTRO<=contenuto_iniziale;

always @(posedge clock) **if** (reset_==1) #Tpropagation REGISTRO<=dw-1_d0;

Blocco always - assegnamento procedurale (scrittura registri)

@: Controllo degli eventi

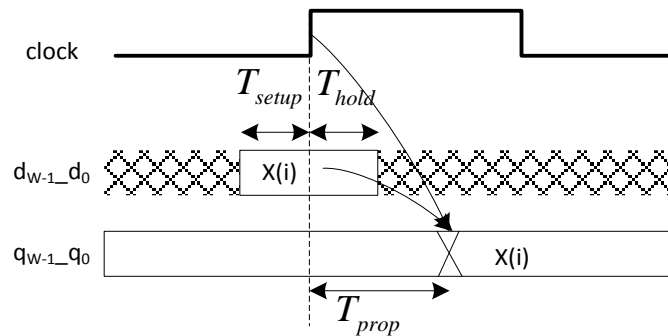
Tempo di propagazione. D'ora in avanti ci metteremo "3". Conviene che sia >0, perché sennò nelle simulazioni non si capisce cosa succede.

Assegnamento procedurale non bloccante "<="

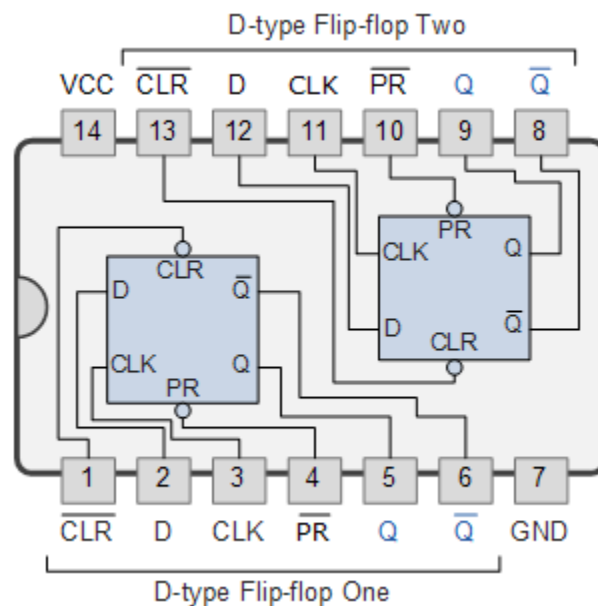
Si noti (è **importante**) la distinzione tra **assegnamento procedurale non bloccante** "<=" e **assegnamento continuo** "=". Il primo descrive la **scrittura in un registro**, che avviene in un **preciso momento** (vedasi condizione @...). Il secondo è una cosa diversa, e descrive qualcosa che è vero continuamente, ad ogni istante *t*. È necessario ricordare che:

- gli assegnamenti **ai fili di uscita** vanno messi in statement **assign** da scrivere in cima (assegnamenti continui).
- Le scritture **dei registri** vanno messi nel blocco **always** da scrivere in fondo (assegnamenti procedurali).

La **temporizzazione** del registro è scritta di seguito.



Ad esempio, in commercio potete trovare il **74LS74 Dual D-type Flip Flop**, il cui schema è il seguente:



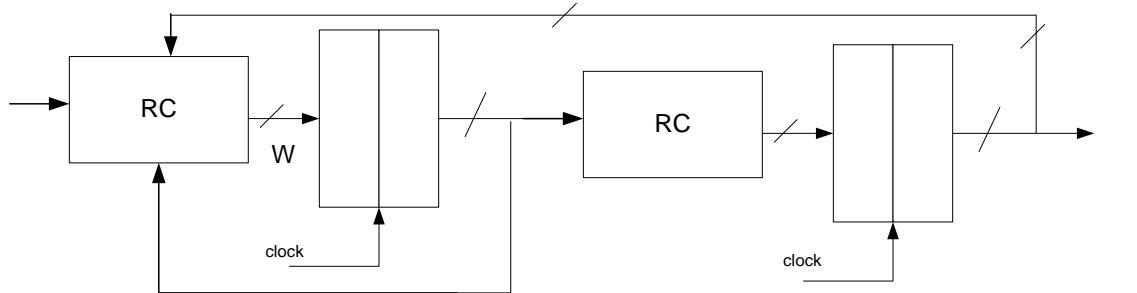
In realtà si tratta di due D-FF autonomi ad un bit, ma se collego i piedini di clock (3, 11) allo stesso generatore di clock, ottengo un registro a due bit. Osservate:

- i piedini per l'inizializzazione al reset (13, 10 e 1, 4, con nomenclatura leggermente diversa rispetto a quella che adottiamo noi)
- Il fatto che ciascun elemento di memoria ha l'uscita diretta (5, 9) e quella negata (6, 8)
- Il fatto che ci sono anche i piedini di collegamento alla tensione e a massa (14, 7)

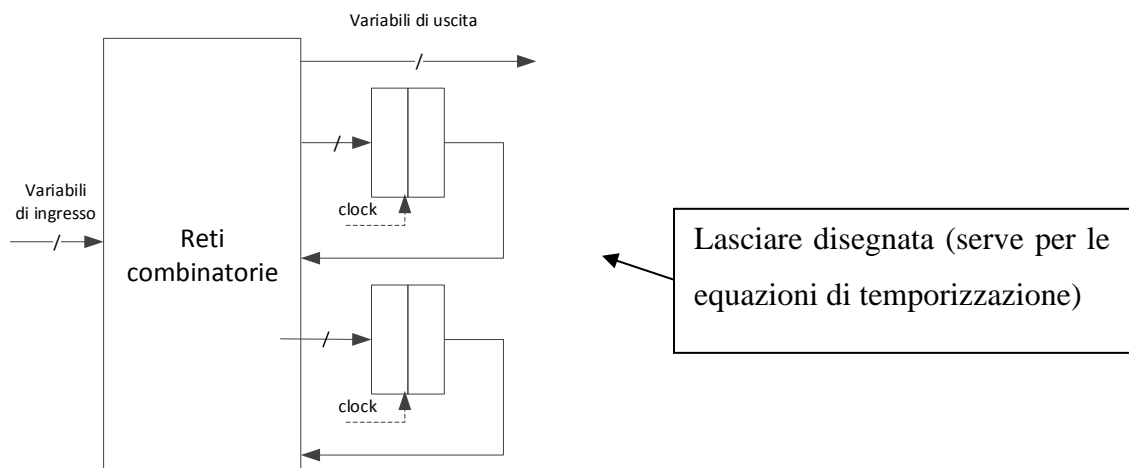
3.2 Prima definizione e temporizzazione di una RSS

Una **rete sequenziale sincronizzata** è, in prima approssimazione (daremo in seguito definizioni più precise), una **collezione di registri e di reti combinatorie**, montati in qualunque modo si vuole, purché non ci siano **anelli di reti combinatorie** (che invece darebbero vita ad una **rete sequenziale asincrona**), e purché i **registri abbiano tutti lo stesso clock**. Ci possono essere, invece, anelli che **abbiano registri al loro interno**, in quanto questo non crea alcun problema.

Posso montare registri e reti combinatorie anche così:



Lo stesso disegno lo posso fare, più in generale, come scritto sotto:



L'unica regola di pilotaggio che dobbiamo garantire (e dalla quale discende tutto il resto) è che

Detto t_i l' i -esimo fronte di salita del clock, lo stato di ingresso ai registri deve essere stabile in $[t_i - T_{setup}, t_i + T_{hold}]$, per ogni i .

Vediamo dove ci porta questa regola. Non posso fare il clock **veloce quanto voglio**. In particolare, se voglio che uno stato di ingresso, attraverso le reti combinatorie, concorra a formare gli ingressi ai registri, dovrò **dare il tempo a chi pilota la rete**: a) di produrre un nuovo stato di ingresso, b) di farlo arrivare, attraverso le reti combinatorie, fino in ingresso ai registri. Definiamo i seguenti **ritardi**:

- $T_{in_to_reg}$: il tempo di attraversamento della più lunga catena fatta di **sole** reti combinatorie che si trovi tra **un piedino di ingresso fino all'ingresso di un registro**
- $T_{reg_to_reg}$: (... ..) **l'uscita di un registro e l'ingresso di un registro**
- $T_{in_to_out}$: (... ..) **un piedino di ingresso e un piedino di uscita**
- $T_{reg_to_out}$: (... ..) **l'uscita di un registro e un piedino di uscita**

Ho i ritardi sopra scritti, e ho **tre vincoli temporali**

- a) ingressi costanti in $[t_i - T_{setup}, t_i + T_{hold}]$ (vincolo costruttivo dei registri)
- b) vincolo di pilotaggio in ingresso: chi pilota gli ingressi (chi sta “a monte” della RSS) deve avere almeno un tempo T_{a_monte} per poterli cambiare. Al netto di tutti i ritardi sopra scritti, dovrò lasciare una finestra larga almeno T_{a_monte} in ogni periodo di clock per il pilotaggio della rete.
- c) vincolo di pilotaggio in uscita: chi usa le uscite (chi sta “a valle” della RSS) deve averle stabili per un tempo T_{a_valle} per poterci fare qualcosa. Al netto di tutti i ritardi sopra scritti, dovrò lasciare una finestra larga almeno T_{a_valle} in ogni periodo di clock perché si possano usare le uscite.

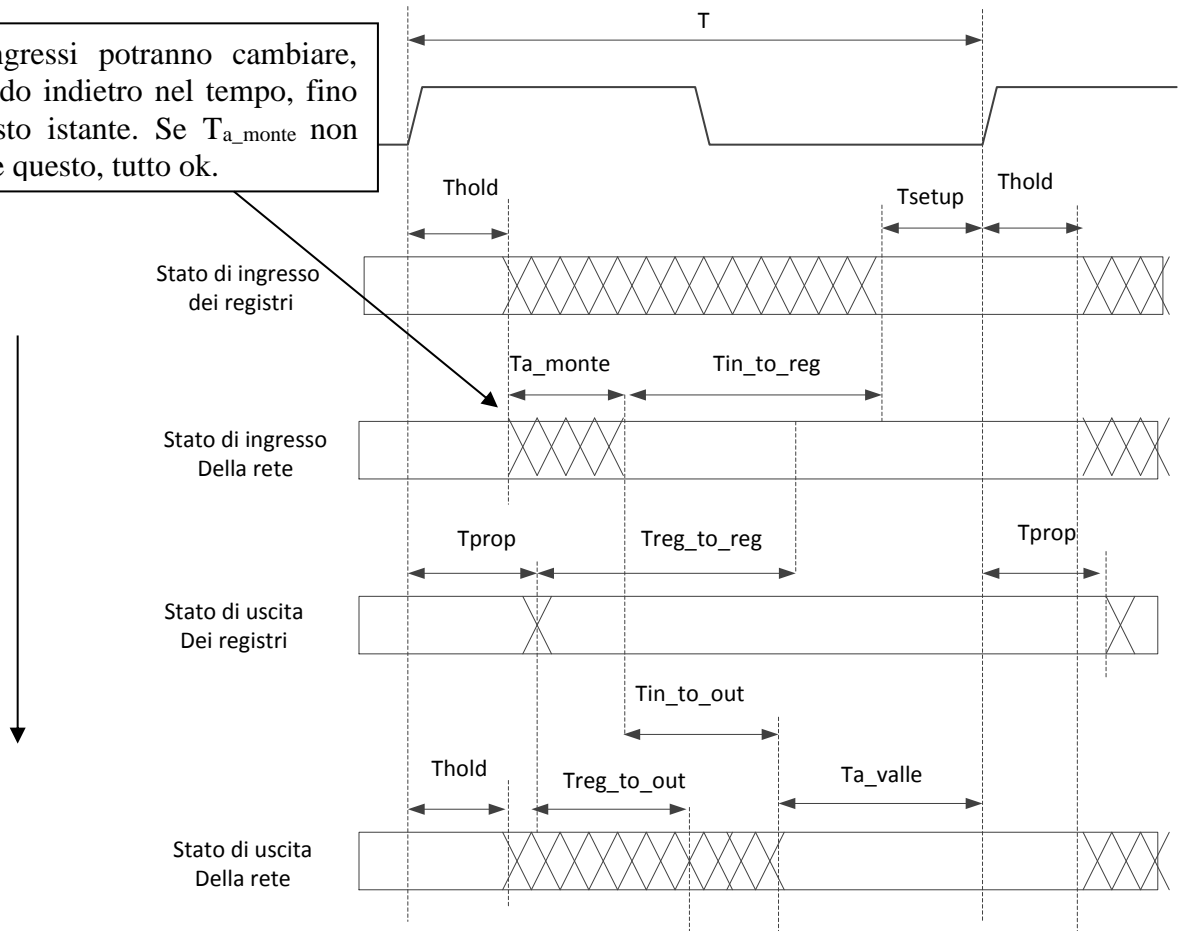
Ciò detto, posso **dimensionare il periodo di clock** in modo da tener conto dei tre vincoli sopra scritti, noti i ritardi che abbiamo definito.

(**chiave di lettura**: all'istante 0 il clock ha il fronte. Da lì elenco tutti i tempi che mi ci vogliono.

Disegnare con riferimento alla figura di temporizzazione di sotto)

- | | |
|--|-----------------------------------|
| 1) $T \geq T_{hold} + T_{a_monte} + T_{in_to_reg} + T_{setup}$ | (percorso da ingresso a registro) |
| 2) $T \geq T_{prop} + T_{reg_to_reg} + T_{setup}$ | (percorso da registro a registro) |
| 3) $T \geq T_{hold} + T_{a_monte} + T_{in_to_out} + T_{a_valle}$ | (percorso da ingresso a uscita) |
| 4) $T \geq T_{prop} + T_{reg_to_out} + T_{a_valle}$ | (percorso da registro a uscita) |

Gli ingressi potranno cambiare, tornando indietro nel tempo, fino a questo istante. Se T_{a_monte} non eccede questo, tutto ok.



Lo stato di uscita sarà utilizzabile dal mondo esterno dopo che:

- 1) Lo stato dei registri avrà attraversato le RC per arrivare in uscita ($T_{reg_to_out}$)
- 2) Lo stato di ingresso della rete avrà attraversato le RC fino all'uscita ($T_{in_to_out}$)

Le uscite saranno utilizzabili per tutto questo tempo. Se T_{a_valle} non eccede questo intervallo, tutto ok.

Ci sono alcune sottigliezze da tenere in conto:

- T_{sfas} il **massimo sfasamento tra due clock**. Se voglio portare un clock comune a elementi diversi, non posso che aspettarmi che, per via dei ritardi sulle linee, a qualche registro arrivi prima e a qualche altro dopo.
- T_{reg} : sappiamo che lo stato di un D-FF cambia dopo T_{prop} dal fronte di salita. Se un registro è formato da $W > 1$ bit, è impensabile che cambino **tutti contemporaneamente**. Ci sarà, quindi, un tempo in più da attendere dopo T_{prop} per essere certi che lo stato di uscita di un registro sia cambiato **per intero**. Possiamo quindi scrivere $T_{prop}' = T_{prop} + T_{reg}$ e dimenticarcelo

Quindi, ad essere precisi, le disequazioni dovrebbero essere riscritte in questa maniera

1) $T \geq T_{sfas} + T_{hold} + T_{a_monte} + T_{in_to_reg} + T_{setup}$	(percorso da ingresso a registro)
2) $T \geq T_{sfas} + T_{prop}' + T_{reg_to_reg} + T_{setup}$	(percorso da registro a registro)
3) $T \geq T_{sfas} + T_{hold} + T_{a_monte} + T_{in_to_out} + T_{a_valle}$	(percorso da ingresso a uscita)
4) $T \geq T_{sfas} + T_{prop}' + T_{reg_to_out} + T_{a_valle}$	(percorso da registro a uscita)

In generale, però, T_{sfas} è **molto piccolo**, e quindi lo supporremo nullo d'ora in avanti.

Se rendiamo il modello disegnato in figura **un po' meno generale**, magari vietando qualche cammino, è probabile che le cose si semplifichino. In particolare, la condizione 3) rischia di essere la più vincolante, perché costringe a tenere conto contemporaneamente delle esigenze di chi sta “a monte” e di chi sta “a valle”. Se, ad esempio, impongo che **le uscite siano soltanto funzione combinatoria del contenuto dei registri**, e che quindi **non ci sia mai connessione diretta tra ingresso e uscita** (cioè, non esista mai una via **combinatoria** tra ingresso e uscita), la terza condizione scompare. Reti così fatte si chiamano **reti (su modello) di Moore**, e le vedremo in dettaglio più in là. Se, invece, impongo che le uscite siano prese direttamente dalle uscite dei registri (senza reti combinatorie nel mezzo), nella disequazione 4) scompare il termine $T_{reg_to_out}$. Reti così fatte si chiamano **reti (su modello) di Mealy ritardato**.

È difficile, se non impossibile, che **chi interagisce con una RSS** possa rispettare vincoli di temporizzazione (e.g., cambiare gli ingressi a monte soltanto durante la finestra consentita) se non è a conoscenza del **clock della rete a valle**. Se vogliamo far interagire due reti, delle due l'una:

- Le due reti devono avere un clock a comune
- Le due reti devono implementare meccanismi di sincronizzazione, detti *handshake*, che vedremo più avanti nel corso.

Esistono tecniche formali e diversi modelli per la sintesi di RSS. Come al solito, prima vediamo qualche esempio semplice, sintetizzato in maniera **euristica**, per acquisire dimestichezza.

Considerazione importante

Nelle RSS, **lo stato di ingresso** (opportunamente modificato dalle reti combinatorie) viene campionato **all'arrivo del clock**. Cosa faccia lo stato di ingresso **tra due clock non ha alcuna importanza**, purché si stabilizzi in tempo. Non ci interessa:

- se cambia di n bit, con $n > 1$
- se non cambia affatto, e rimane identico per due fronti di salita del clock.

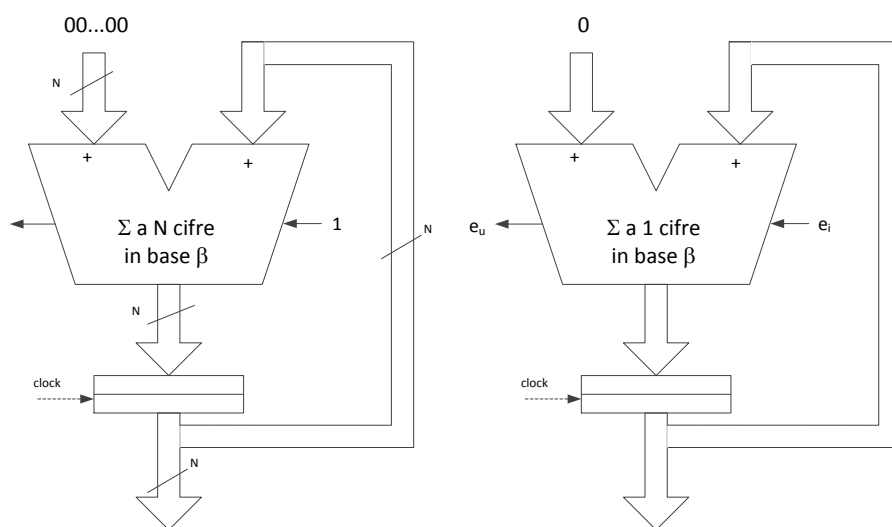
In quest'ultimo caso sarà **comunque** visto come due stati di ingresso **differenti** (perché presentati ad istanti differenti). Le RSS **evolvono all'arrivo del clock** (per essere più precisi: **ad ogni fronte di salita**, ma non lo diremo più per semplicità), **non quando cambiano gli ingressi**.

3.3 Contatori

Un **contatore** è una RSS il cui stato di uscita può essere visto come un **numero naturale ad n cifre in base β** , secondo una qualche codifica. Ad esempio, potremo parlare di **contatori a 2 cifre in base 10 BCD**, o di **contatori a n cifre in base 2**.

Ad ogni clock, il contatore fa la seguente cosa:

- **incrementa di uno** (modulo β^n , ovviamente), il valore in uscita (**contatore up**);
- **decrementa di uno** (modulo β^n , ovviamente), il valore in uscita (**contatore down**);
- **incrementa o decrementa** a seconda del valore di una **variabile di comando** (contatore up/down).



Posso realizzare un **contatore up** con un **modulo sommatore** ed un **registro**. Il sommatore sarà una **rete combinatoria** che lavora in base β , capace di sommare N cifre. Visto che devo incrementare sempre di uno, tanto vale che uno dei due ingressi sia 0, ed il riporto entrante sia uguale ad 1. Dal punto di vista Verilog, la descrizione sarà di questo tipo:

```
module ContatoreUp_Ncifre_BaseBeta(numero, clock, reset_);
  input clock, reset_;
  output [W-1:0] numero;
  reg [W-1:0] OUTR; assign numero=OUTR;
  always @(reset_==0) #1 OUTR<=0;
  always @(posedge clock) if (reset_==1) #3
    OUTR<= Inc_N_cifre_beta(OUTR);
```

Assumendo che **W bit uguali a 0** siano una codifica buona per un numero ad N cifre in base beta

Supponiamo che N cifre in base beta possano essere rappresentate su W bit

Ed il **sommatore** lo descrivo come una rete combinatoria:

```
function [W-1:0] Inc_N_cifre_beta;
  input [W-1:0] numero;
  casex(numero)
    <cod_0> : Inc_N_cifre_beta = <cod_1>;
    <cod_1> : Inc_N_cifre_beta = <cod_2>;
    ...
    default : Inc_N_cifre_beta = 'BXXX...XXX;
  endcase
endfunction
endmodule
```

Se poi il contatore è **in base 2**, e soltanto in quel caso, in Verilog posso scrivere tutto in modo più semplice, in quanto in Verilog è definito l'operatore di somma +, che descrive una rete combinatoria che fa da **sommatore ad N cifre in base 2**. Ovviamente la cosa funziona soltanto in base 2.

```
module ContatoreUp_Ncifre_Base2(numero, clock, reset_);
  input clock, reset_;
  output [N-1:0] numero;
  reg [N-1:0] OUTR; assign numero=OUTR;
  always @(reset_==0) #1 OUTR<=0;
  always @(posedge clock) if (reset_==1) #3 OUTR<=numero+1;
endmodule
```

N bit uguali a 0 sono una codifica buona per un numero naturale ad N cifre in base due.

N cifre in base due sono N bit

Anche:
OUTR<=OUTR+1

Fin qui ho parlato di contatori **up**. Se voglio fare contatori **down**,

- scrivo “-” al posto di “+” nel caso di base 2
- cambio la funzione “sommatore” scritta prima in qualcos'altro (in base β generica)

Un contatore può essere dotato di un **ingresso di abilitazione e_i** , in modo che:

- se l'ingresso e_i vale 1, all'arrivo del clock **conta** (up o down, a seconda di come lo faccio)
- se l'ingresso e_i vale 0, all'arrivo del clock **conserva** l'ultimo valore.

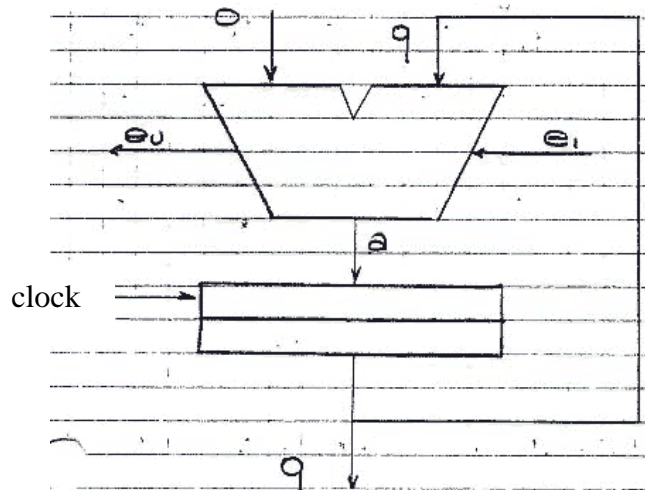
Come si fa a fare questa cosa? Basta che l'ingresso **ei** sia **collegato al riporto entrante del sommatore**. In questo caso, la descrizione Verilog per la base 2 verrebbe in questo modo:

```
module ContatoreUp_Ncifre_Base2 (numero, clock, reset_, ei)
  input      clock, reset_, ei;
  [...]
  always @(posedge clock) if (reset==1) #3 OUTR<=OUTR+{'B00...00,ei};
endmodule
```

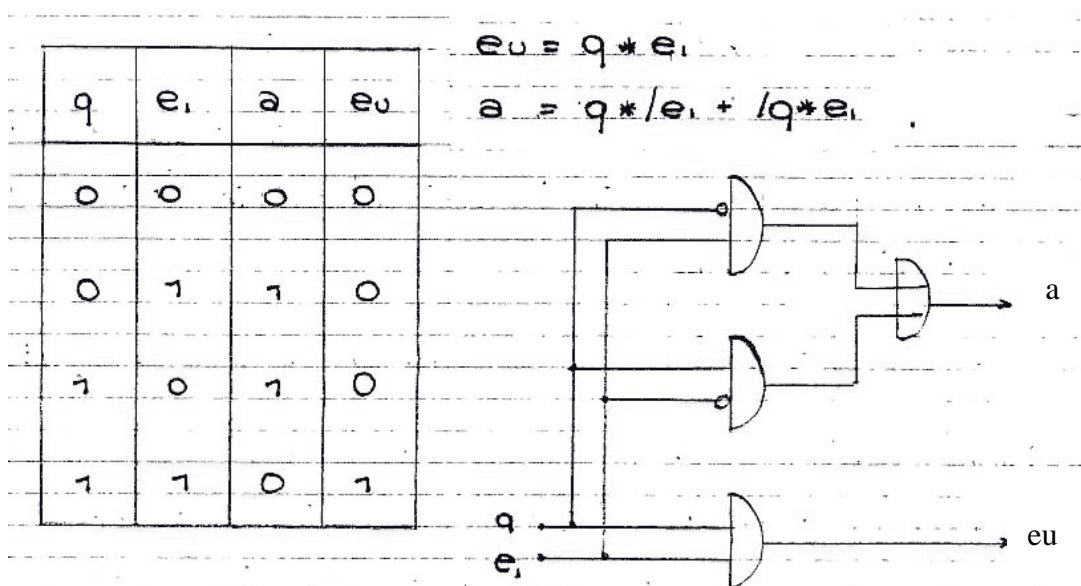
Mentre per la base generica *beta* avrei scritto:

```
always @(posedge clock) if (reset==1) #3
  casex(ei)
    0: OUTR<=OUTR;
    1: OUTR<=Inc_N_cifre_beta(OUTR)
  endcase
```

Un contatore ad *N* cifre, qualunque sia la sua base, può sempre essere scomposto come una serie di contatori **ad una cifra collegati mediante catena dei riporti** (*ripple carry*). In questo caso il registro è costituito dalla giustapposizione di tutti i D-FF che reggono una cifra, D-FF che hanno tutti lo stesso clock.



Nel caso di base 2, la sintesi che viene fuori è quella che conoscete dell'**incrementatore ad 1 cifra in base 2** (o **semisommatore, half-adder**), la cui tabella di verità è la seguente:



Una corrispondente descrizione in Verilog è la seguente:

```
module Elemento_Contatore_Base_2(eu,q,ei,clock,reset_);
  input clock,reset_;
  input ei;
  output eu,q;
  reg OUTR; assign q=OUTR;
  wire a; // variabile di uscita dell'incrementatore
  assign {a,eu}= ({q,ei}=='B00) ?'B00:
                ({q,ei}=='B10) ?'B10:
                ({q,ei}=='B01) ?'B10:
                /* ({q,ei}=='B11) */'B01;
  always @(reset_==0) #1 OUTR<=0;
  always @(posedge clock) if (reset_==1) #3 OUTR<=a;
endmodule
```

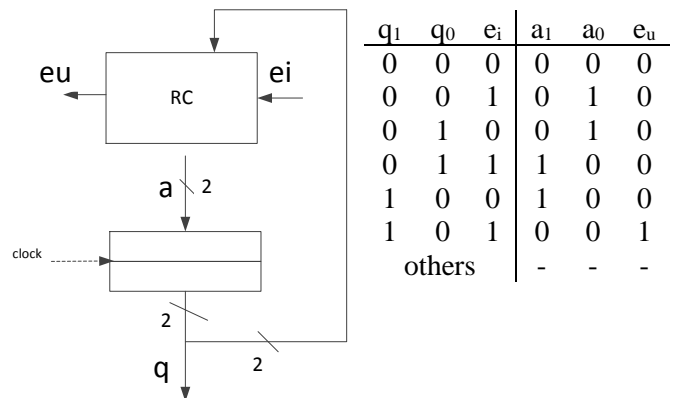
Ci sono 1000 modi diversi per scrivere l'**assign**, tra cui:
 assign
 a=(q!=ei)?1:0;
 assign eu=(q&ei);
 Oppure, usando lo XOR (^)
 assign a=q^ei;
 Questa, però, è una **descrizione**, non una **sintesi**.

Descriviamo e sintetizziamo il contatore **ad una cifra in base 3**. In base 3 ci vogliono 2 bit per codificare una cifra, e possiamo assumere la seguente codifica delle cifre: **0='B00**, **1='B01**, **2='B10**. Quindi ci vorrà un registro **a due bit**, ed una rete che:

- ha in ingresso i 2 bit che escono dal registro ed un riporto entrante
- ha in uscita i due bit che vanno in ingresso al registro ed il riporto uscente

Per la **descrizione**, dovrò fare una delle due seguenti cose:

- un disegno come quello in figura, con tanto di tabella di verità
- una descrizione in Verilog



```
module Elemento_Contatore_Base_3(eu,q1_q0,ei,clock,reset_);
  input clock,reset_;
  input ei;
  output eu;
  output [1:0] q1_q0;
  reg [1:0] OUTR; assign q1_q0=OUTR;
  wire [1:0] a1_a0; // variabile di uscita dell'incrementatore
  assign {a1_a0,eu}= ({q1_q0,ei}=='B000)?'B000:
                    ({q1_q0,ei}=='B010)?'B010:
                    ({q1_q0,ei}=='B100)?'B100:
                    ({q1_q0,ei}=='B001)?'B010:
                    ({q1_q0,ei}=='B011)?'B100:
                    ({q1_q0,ei}=='B101)?'B001:
                    /* default */ 'BXXX;
  always @(reset_==0) #1 OUTR<='B00;
  always @(posedge clock) if (reset_==1) #3 OUTR<=a1_a0;
endmodule
```

Se volessi farne la sintesi, dovrei scrivere espressioni algebriche ottimizzate per le tre uscite:

$q_1 q_0$					$q_1 q_0$					$q_1 q_0$				
e_i	00	01	11	10	e_i	00	01	11	10	e_i	00	01	11	10
0	0	0	-	1	0	0	1	-	0	0	0	0	-	0
1	0	1	-	0	1	1	0	-	0	1	0	0	-	1
a_1					a_0					e_u				

$$a_1 = q_1 \cdot \bar{e}_i + q_0 \cdot e_i$$

$$a_0 = q_0 \cdot \bar{e}_i + \bar{q}_1 \cdot \bar{q}_0 \cdot e_i$$

$$e_u = q_1 \cdot e_i$$

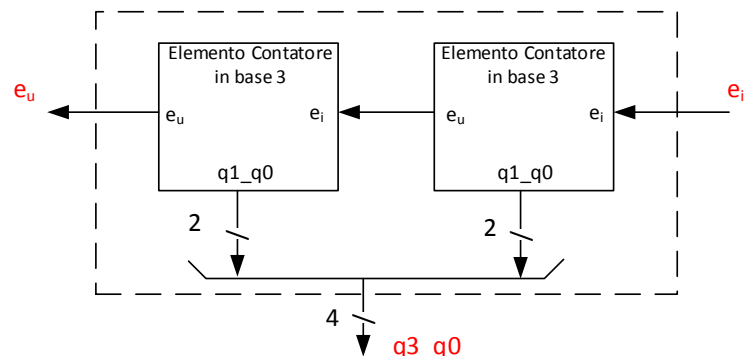
In Verilog posso scrivere anche la sintesi, purché traduca le espressioni algebriche scritte sopra in statement **assign** da sostituire nella precedente descrizione.

```

wire [1:0] a1_a0; // variabile di uscita dell'incrementatore
wire a1, a0; // variabili a un bit che compongono a1_a0
assign a1_a0={a1,a0};
assign eu=q1_q0[1]&ei;
assign a1=(q1_q0[0]&ei) | (q1_q0[1]&~ei);
assign a0=(~q1_q0[1]&~q1_q0[0]&ei) | (q1_q0[0]&~ei);

```

Se, infine, voglio descrivere un contatore **a due cifre** in base 3, posso ottenerlo connettendo i moduli ad una cifra che ho già descritto, scrivendo in Verilog il disegno accanto:



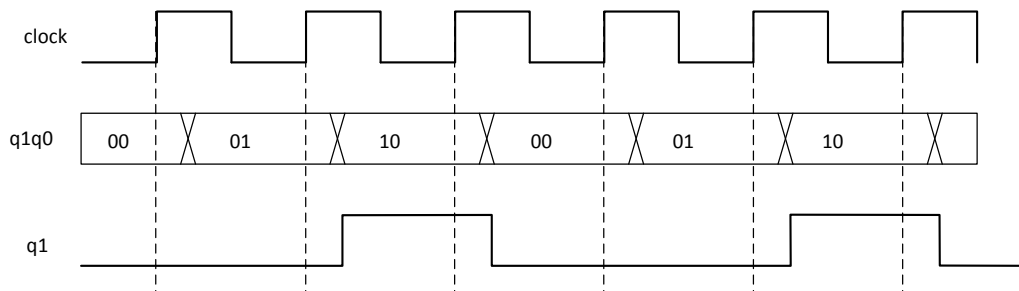
```

module Contatore_Base_3_2Cifre(eu,q3_q0,ei,clock,reset_);
  input clock,reset_;
  input ei;
  output eu;
  output [3:0] q3_q0;
  wire riporto;

  Elemento_Contatore_Base_3 LSD(riporto,q3_q0[1:0],ei,clock,reset_),
    MSD(eu,q3_q0[3:2],riporto,clock,reset_);
endmodule

```

I contatori “**dividono in frequenza**”. Possono essere usati per **dividere la frequenza del clock** per un certo valore. Ad esempio, posso usare il **bit più significativo** dell’uscita del contatore in base 3 per ottenere un clock che va 3 volte più lento del clock del contatore.



Analogamente, la **cifra più significativa** di un contatore ad N cifre in base due che riceve clock a periodo T è un clock a periodo $2^N \cdot T$. Si noti che, per generare un clock in questo modo, si possono usare solo **uscite di registri**, mai quelle di reti combinatorie (e.g., il riporto uscente e_u). Infatti, le uscite di combinatorie potrebbero ballare, mentre un clock deve essere assolutamente stabile.

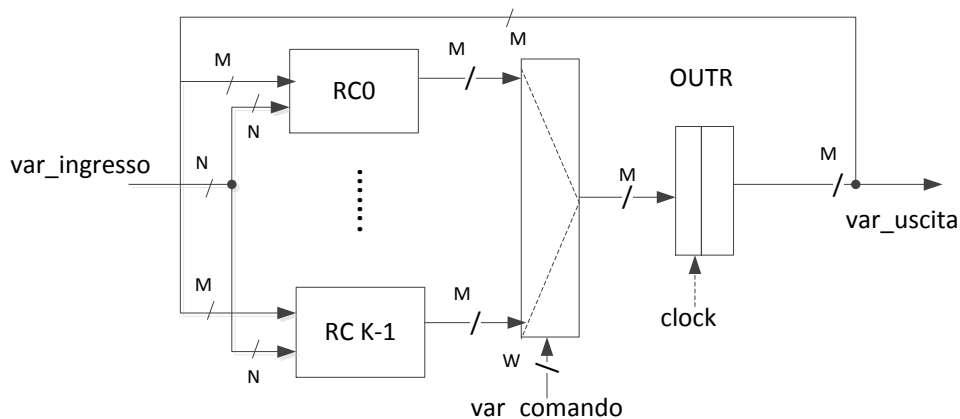
Descrizione del contatore espandibile ad 1 cifra in base 10

```
module Elemento_Contatore_Base_10(eu,q3_q0,ei,clock,reset_);
  input clock,reset_;
  input ei;
  output eu;
  output [3:0] q3_q0;
  reg [3:0] OUTR; assign q3_q0=OUTR;
  wire [3:0] a3_a0;
  assign {a3_a0,eu}={({q3_q0,ei}=='B00000')?'B00000':
    ({q3_q0,ei}=='B00010')?'B00010':
    ({q3_q0,ei}=='B00100')?'B00100':
    ({q3_q0,ei}=='B00110')?'B00110':
    [...]
    ({q3_q0,ei}=='B10010')?'B10010':
    ({q3_q0,ei}=='B00001')?'B00010':
    [...]
    ({q3_q0,ei}=='B10011')?'B00001':
    /* default */ 'BXXXXX;

  always @(reset_==0) #1 OUTR<='H0;
  always @(posedge clock) if (reset_==1) #3 OUTR<=a3_a0;
endmodule
```

3.4 Registri multifunzionali

Un **registro multifunzionale** è una rete che, all'arrivo del clock, memorizza nel registro stesso **una tra K funzioni combinatorie possibili**, scelte impostando un certo numero di **variabili di comando** ($W = \lceil \log_2 K \rceil$). Tali funzioni combinatorie potranno essere fatte in un modo qualunque, ad esempio potranno avere in ingresso l'uscita del registro stesso (ed altre variabili logiche). Si realizza con un **multiplexer a K ingressi**, alcune reti combinatorie ed un registro.



Dal punto di vista della descrizione Verilog, abbiamo:

```

module Registro_Multifunzionale(var_uscita,var_ingresso,
                                var_comando,clock,reset_);

    input clock,reset_;
    input [N-1:0] var_ingresso;
    input [W-1:0] var_comando;
    output [M-1:0] var_uscita;

    reg [M-1:0] OUTR; assign var_uscita=OUTR;

    always @(reset_==0) #1 OUTR<=contenuto_iniziale;
    always @(posedge clock) if (reset_==1)
        case(var_comando)
            0 : OUTR<=F0(var_ingresso,OUTR);
            ...
            ...
            K-1: OUTR<=FK-1(var_ingresso,OUTR);
        endcase
    endmodule

```

$F_0(\dots)$

...

$F_{K-1}(\dots)$

Saranno **reti combinatorie**, che eventualmente descriveremo come **funzioni**, come sempre.

Un esempio semplice di registro multifunzionale è il caso (**bifunzionale**) di caricamento/traslazione. Un registro che, in base ad una variabile di comando **b0**,

- **carica** un nuovo valore, cioè memorizza M bit ex novo, oppure
- **trasla** a sinistra il proprio contenuto, cioè **butta via** il bit più significativo, fa scorrere gli altri di una posizione a sinistra, ed inserisce **zero** come bit meno significativo. Nel caso $M=4$ abbiamo:

```

module Registro_CaricaParallelo_TraslaSinistro(z3_z0,x3_x0,b0,
clock,reset_);
    input clock,reset_;
    input [3:0] x3_x0;
    input b0;
    output [3:0] z3_z0;

    reg [3:0] OUTR; assign z3_z0=OUTR;

    always @(reset_==0) #1 OUTR<='B0000;
    always @(posedge clock) if (reset_==1)
        case(b0)
            'B0: OUTR<=x3_x0;
            'B1: OUTR<={OUTR[2:0],1'b0};
        endcase
    endmodule

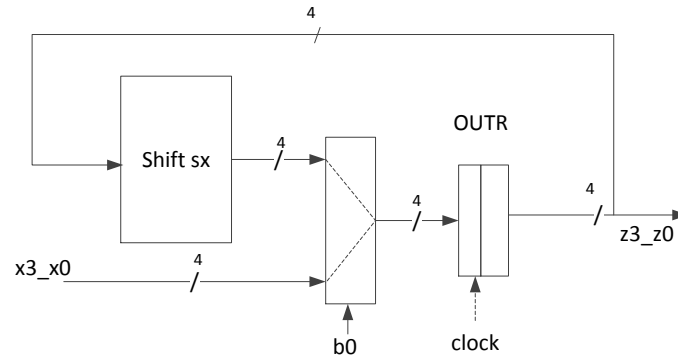
```

Spiegare giustapposizione: 1'B0 vuol dire "la costante 0, espressa in binario, su 1 bit"

NB: Mentre **a destra di** \leq ci può stare qualunque espressione (che può coinvolgere anche bit del registro), **a sinistra** ci deve stare **un registro intero**. Non ha senso scrivere
 $\text{OUTR}[2] \leq 'B1$

Il clock, infatti, arriva **contemporaneamente** a tutti i bit del registro.

endmodule

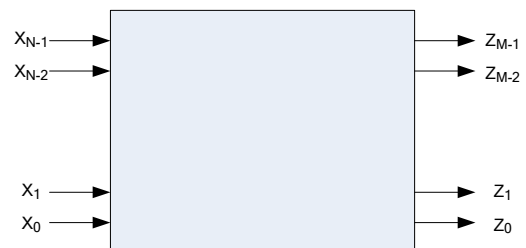


Dopo aver visto alcuni esempi semplici di RSS, passiamo a descrivere i modelli **formali** per la loro sintesi. Vedremo che ce ne sono **tre**, e sono: il modello di **Moore**, il modello di **Mealy**, quello di **Mealy ritardato**. Partiamo dal più semplice dei tre.

3.5 Modello di Moore

Una **RSS di Moore** è rappresentata come segue:

1. un insieme di N variabili logiche di ingresso.
2. un insieme di M variabili logiche di uscita.



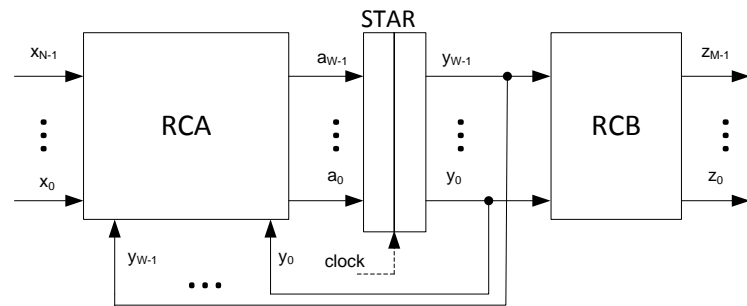
3. Un **meccanismo di marcatura**, che ad ogni istante marca uno **stato interno presente**, scelto tra un insieme **finito** di K stati interni $\mathbf{S} \equiv \{S_0, \dots, S_{K-1}\}$
4. Una **legge di evoluzione nel tempo** del tipo $A: \mathbf{X} \times \mathbf{S} \rightarrow \mathbf{S}$, che mappa quindi una coppia (stato di ingresso, stato interno) in un nuovo stato interno.
5. Una **legge di evoluzione nel tempo** del tipo $B: \mathbf{S} \rightarrow \mathbf{Z}$, che decide lo stato di uscita basandosi sullo stato interno. (Nota: tale legge **non** è più generale, del tipo $B: \mathbf{X} \times \mathbf{S} \rightarrow \mathbf{Z}$. Se fosse più generale saremmo **fuori dal modello**).
6. La rete riceve **segnali di sincronizzazione**, come transizioni da 0 a 1 del segnale di clock
7. Si adegua alla seguente **legge di temporizzazione**:

“Dato S , stato interno marcato ad un certo istante, e dato X ingresso ad un certo istante immediatamente **precedente l’arrivo di un segnale di sincronizzazione**,

- a) individuare il **nuovo stato interno da marcare** $S' = A(S, X)$
- b) **attendere T_{prop} dopo l’arrivo del segnale di sincronizzazione**
- c) **promuovere S' al rango di stato interno marcato**

E, inoltre, **individuare continuamente** $Z=B(S)$ e presentarlo in uscita

Una rete di Moore può sempre essere sintetizzata secondo il modello di figura. STAR è lo **status register**, cioè il registro che memorizza lo **stato interno presente (marcato)**.



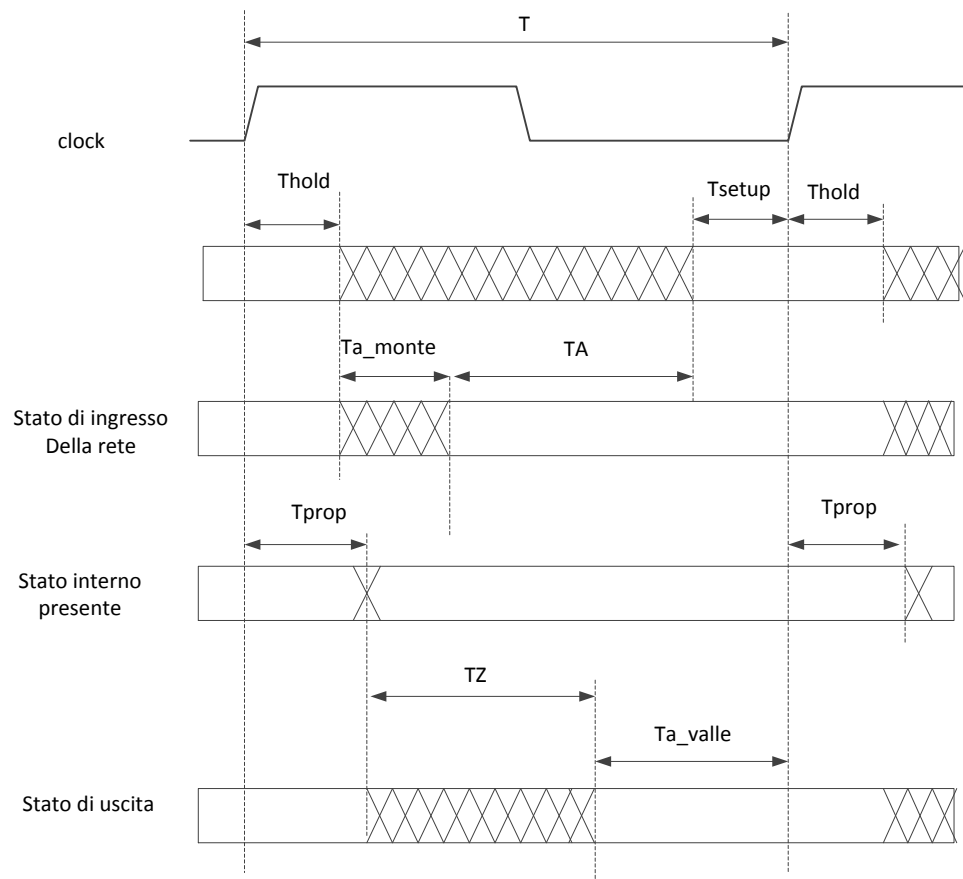
Si noti che:

- Lo **status register** è una **batteria di D-FF**, che sono **non trasparenti**. Pertanto, il nuovo stato interno verrà presentato alla rete RCB dopo T_{prop} dal fronte del clock. A questo punto, la rete **non sarà più sensibile all'ingresso**, e quindi non ci sono problemi di nessun tipo.
- Il nuovo stato interno delle RSS è lo stato di uscita della rete RCA, che ha in ingresso sia gli ingressi della RSS che le variabili di stato.
- Tutto questo sottende una **codifica degli stati interni** in termini di variabili logiche.

Come già visto, il clock non può andare arbitrariamente veloce, perché c'è da garantire

- che chi sta a monte ed a valle della rete ne possa pilotare gli ingressi ed usare le uscite,
- che gli ingressi ai registri non varino nella finestra $[T_{setup}; T_{hold}]$;
- che i nuovi stati di ingresso riescano ad arrivare in tempo agli ingressi dei registri
- che i nuovi stati interni riescano ad arrivare in tempo in uscita.

Devo quindi dimensionare il clock in accordo alle disuguaglianze scritte sotto, dove T_A e T_Z sono i tempi di attraversamento delle reti combinatorie RCA e RCB, rispettivamente.



$$T \geq T_{hold} + T_{a_monte} + T_A + T_{setup} \quad (\text{percorso da ingresso a STAR})$$

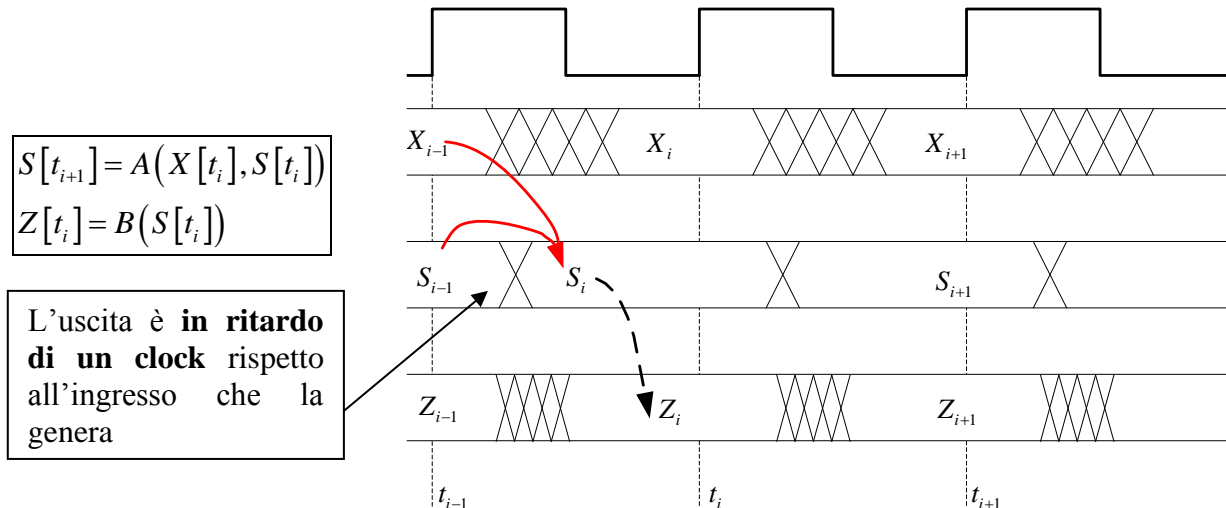
$$T \geq T_{prop} + T_A + T_{setup} \quad (\text{percorso da STAR a STAR})$$

$$T \geq T_{prop} + T_Z + T_{a_valle} \quad (\text{percorso da STAR a uscita})$$

Di queste, la seconda è certamente meno restrittiva della prima, in quanto $T_{prop} \approx T_{hold}$, e $T_{a_monte} \leq T_{prop}$. Quindi può essere **ignorata**, in quanto implicata dalla prima.

NB: nelle domande per l'orale **ce le mettete tutte e tre**, e **poi** scrivete che la 2a si può trascurare.

Se le condizioni di temporizzazione sono rispettate, una rete di Moore si evolve in modo **deterministico** (cioè prevedibile). Detti $t_i, i \geq 0$, gli **istanti di sincronizzazione**, e detti $X[t_i], S[t_i], Z[t_i]$ gli stati di **ingresso, interno e di uscita** all'istante t_i , sarà:



Si dice che la rete di Moore approssima **un automa ideale sincrono a stati finiti**. Si può osservare che lo stato di uscita all'istante t_i è funzione, attraverso lo stato interno, della **storia degli stati di ingresso e dello stato interno iniziale** (quello impostato al reset), **fino allo stato di ingresso precedente all'ultimo clock**. Infatti, tra ingresso ed uscita c'è un registro, ed in un registro **si perde un clock** per via della non trasparenza.

Una rete di Moore si **descrive** dando la specifica delle leggi combinatorie A e B, in uno qualunque dei modi consueti: **tabella di flusso, grafo di flusso**, descrizione in **Verilog**.

Una rete di Moore si può descrivere in Verilog come segue:

```

module Rete_di_Moore(zM-1, ..., z0, xN-1, ..., x0, clock, reset_);
  input clock, reset_;
  input xN-1, ..., x0;
  output zM-1, ..., z0;
  reg [W-1:0] STAR; parameter S0=codifica0, ..., SK-1=codificaK-1;
  assign {zM-1, ..., z0} = (STAR==S0) ? ZS0 :
                                (STAR==S1) ? ZS1 :
                                ...
                                /* (STAR==SK-1) */ ZSK-1;

  always @(reset_==0) #1 STAR<=stato_interno_iniziale;
  always @(posedge clock) if (reset_==1) #3
    case(STAR)
      S0 : STAR<=AS0(xN-1, ..., x0);
      S1 : STAR<=AS1(xN-1, ..., x0);
      ...
      SK-1: STAR<=ASK-1(xN-1, ..., x0);
    endcase
endmodule

```

Dichiarazione di costante

Legge B

Legge A

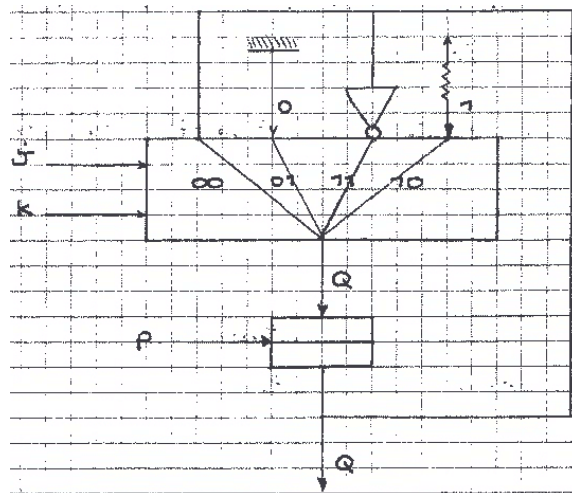
Avrei potuto anche scrivere **STAR<=legge_A(xN-1, ..., x0, STAR)**;
Così scrivo K espressioni combinatorie diverse, corrispondenti al caso in cui il contenuto di STAR vale S0, S1, ... Sk-1.

Questa descrizione è **consistente** con il modello strutturale che abbiamo visto prima.

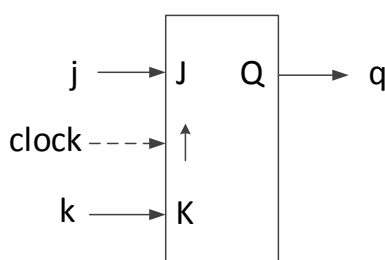
3.5.1 Esempio: il Flip-Flop JK

Il FF JK è una **rete sequenziale sincronizzata** con due ingressi ed un'uscita che, all'arrivo del clock, valuta i suoi due ingressi **j** e **k**, e si comporta come segue:

<i>jk</i>	Azione in uscita
00	Conserva
10	Setta
01	Resetta
11	Commuta



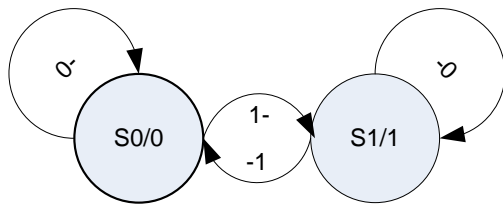
Un modo (non ottimizzato) di vedere questa rete è come **registro multifunzionale ad un bit**. Il multiplexer a 4 vie prende in ingresso *j* e *k*, e commuta l'uscita, la sua negata, e due costanti. Posso dare, per questa rete, una **tabella di applicazione** simile a quella del Latch SR. Visto che ci si può avvalere dell'ingresso 11, stavolta lecito, finirà che **uno dei due ingressi è sempre non specificato**.



Attenzione: questa tabella vuol dire: *se quando arriva il clock voglio che la variabile di uscita vada a...*

q	q'	j	k
0	0	0	-
0	1	1	-
1	0	-	1
1	1	-	0

Il FF JK è una rete di Moore. Infatti, non c'è una via combinatoria dagli ingressi all'uscita, che è invece funzione soltanto del contenuto del registro. Possiamo quindi sintetizzarlo come **rete di Moore**. Il FF JK serve a **memorizzare un bit**, e quindi posso associargli **due stati**, S0 e S1, nei quali memorizzo rispettivamente 0 e 1. Codificherò questi stati interni con **una variabile di stato** che varrà 0 e 1 nei due casi, così **RCB diventa un cortocircuito**. Posso disegnare la tabella di flusso (o il grafo di flusso), che è uno dei modi con cui si descrivono gli automi a stati finiti.



jk		00	01	11	10	q
s	s0	S0	S0	S1	S1	0
	s1	S1	S0	S0	S1	1

Attenzione a cosa vuol dire la tabella di flusso (o il grafo di flusso) in questo caso: vuol dire che la rete si evolve, cambiando il proprio stato interno marcato, **quando arriva il clock**. Nelle RSS **non ha senso cerchiare gli stati e parlare di stati stabili**: il concetto di stabilità è legato alla presenza di **anelli combinatori**, che nelle RSA sono gli elementi che memorizzano lo stato interno. In una rete di Moore non ci sono anelli combinatori, perché gli stati interni sono memorizzati dai **registri**. Tutti gli stati sono stabili per un periodo di clock, se la rete è pilotata correttamente, e ad ogni clock ho una **nuova transizione**, che in alcuni casi può concretizzarsi nella marcatura del **medesimo stato** interno in cui la rete già si trova, come nel caso di S_0 con ingresso 0-.

A livello di Verilog, possiamo dare una descrizione di questa rete **semplificando** quella generale vista prima per le reti di Moore.

```

module FlipFlop_JK(q,j,k,clock,reset_);
input clock,reset_;
input j,k;
output q;
reg STAR; parameter S0='B0,S1='B1;
assign q=(STAR==S0)?0:1;

always @(reset_==0) #1 STAR<=S0;
always @(posedge clock) if (reset_==1) #3
  casex(STAR)
    S0: STAR<=(j==0)?S0:S1;
    S1: STAR<=(k==0)?S1:S0;
  endcase
endmodule

```

Si può ovviamente procedere alla **sintesi** di RCA ed RCB. Basta scegliere una **codifica** per gli stati interni ($S_0='B0$, $S_1='B1$), e si ottiene quanto segue:

jk		00	01	11	10	q
s	s0	S0	S0	S1	S1	0
	s1	S1	S0	S0	S1	1

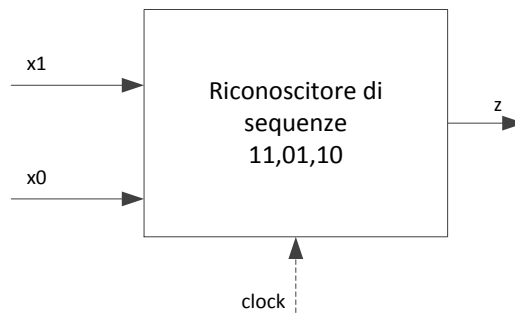
jk		00	01	11	10
y0	0	0	0	1	1
	1	1	0	0	1

$$a_0 = j \cdot \overline{y_0} + \overline{k} \cdot y_0, \quad q = y_0.$$

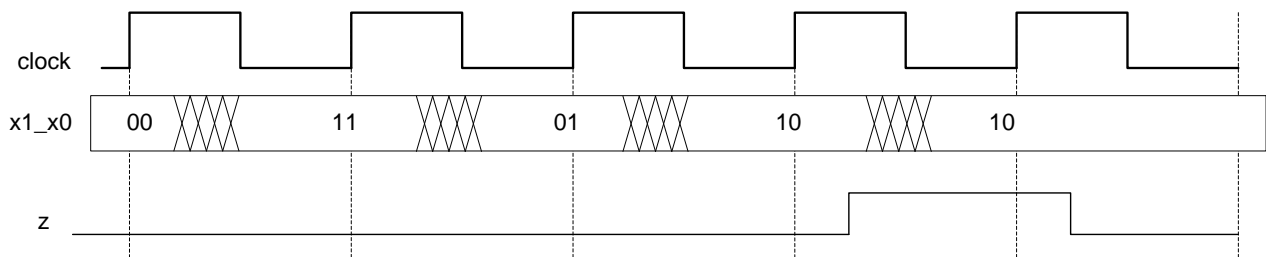
3.5.2 Esempio: riconoscitore di sequenze 11,01,10

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 (11,10,01), 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, che si devono presentare **in tre clock consecutivi**. Se un valore permane per più di un ciclo di clock, la sequenza è **diversa**. Un esempio di quello che deve succedere è scritto nella temporizzazione sottostante:



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 interni avrà un'uscita ad 1, gli altri a 0.

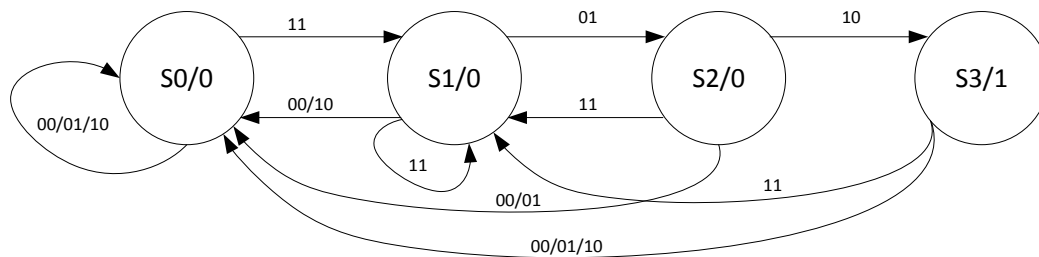
Dobbiamo quindi pensare ad un **grafo di flusso con 4 stati**, nel quale:

- Nei primi tre stati (chiamiamoli S_0 , S_1 , S_2), l'uscita varrà 0
- Nel 4° stato (chiamiamolo S_3) l'uscita varrà 1.

Quello che resta da fare è capire con quali stati di ingresso si transisce da uno stato all'altro. Analizziamoli uno alla volta.

S0: Supponiamo che al reset iniziale la rete si trovi in uno **stato interno iniziale S_0** (sappiamo 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 11. Quando arriva 11, **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 ci vuole un altro stato. Chiamiamolo S1.

S1: In S1 valuto il nuovo stato di ingresso, e prendo la seguente decisione.

- Se il nuovo stato di ingresso è 01, 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).
- Il nuovo stato di ingresso potrebbe anche essere 11, nel qual caso devo restare in S1, perché la sequenza 11, 11 non è corretta, ma il secondo 11 potrebbe essere l’inizio di una sequenza corretta. In tutti gli altri casi si deve ripartire da capo, tornando in S0

S2: memorizza il fatto che ho ricevuto **due passi corretti** di sequenza. Ci sono arrivato con ingresso 01. Al successivo clock, devo comunque uscire da S2. Se arriva 11, 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**. In tutti gli altri casi torno in S0.

S3: in questo stato **dovrò porre l'uscita ad 1**, perché devo sbloccare la cassaforte. Al prossimo clock, o vado in S1 (se vedo in ingresso 11) o, per qualunque altro stato di ingresso, **devo ripartire da S0**, in quanto non vedo 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
S_0	S_1	S_2	S_3	z	
S0	S0	S0	S1	S0	0
S1	S0	S2	S1	S0	0
S2	S0	S0	S1	S3	0
S3	S0	S0	S1	S0	1

Nota importante: siamo d'accordo che lo stato di ingresso può cambiare come vuole tra due clock successivi, fatto salvo che rimanga stabile a cavallo del fronte di salita. Va però tenuto conto del fatto che, se devo riconoscere una sequenza di stati di ingresso con una RSS, il vincolo è che la sequenza di **N stati dovrà presentarsi in N clock**. Non potrà presentarsi **più velocemente**, altrimenti ne perdo qualcuno per strada. Se ho un clock al secondo, e gli ingressi mi variano ogni 10mo di secondo, di sicuro ne perdo 9 tra un clock e l'altro. Non potrà presentarsi **più lentamente**, perché altrimenti devo **cambiare la descrizione del riconoscitore**. Si può descrivere e sintetizzare (fare per casa) una rete che riconosce la sequenza di stati di ingresso 11, 01, 10, ciascuno tenuto in ingresso per un numero arbitrario (ma non inferiore ad 1) di clock.

```

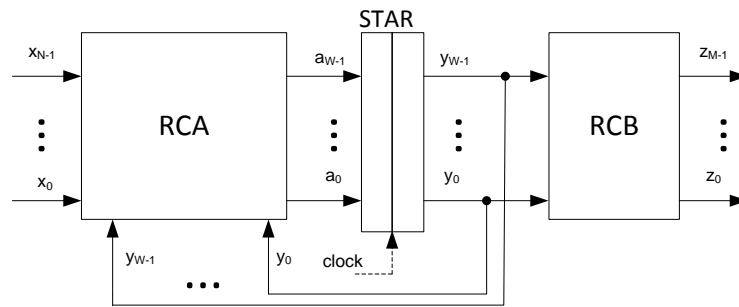
module Riconoscitore_di_Sequenza(z,x1_x0,clock,reset_);
  input clock,reset_;
  input [1:0] x1_x0;
  output z;
  reg [1:0] STAR; parameter S0='B00, S1='B01, S2='B10, S3='B11;
  assign z=(STAR==S3)?1:0;
  always @(reset_==0) #1 STAR<=S0;
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: STAR<=(x1_x0=='B11)?S1:S0;
      S1: STAR<=(x1_x0=='B01)?S2:(x1_x0=='B11)?S1:S0;
      S2: STAR<=(x1_x0=='B10)?S3:(x1_x0=='B11)?S1:S0;
      S3: STAR<=(x1_x0=='B11)?S1:S0;
    endcase
endmodule

```

Per le codifiche degli stati interni un buon criterio di scelta è: **guardare cosa deve fare RCB**, e sceglierle in modo da semplificarla il più possibile.

NB: abituatevi a scrivere descrizioni Verilog:
 assign z0=STAR[1]&STAR[0]
 è una sintesi.

Posso sintetizzare il riconoscitore usando il modello strutturale di Moore visto prima. La sintesi corrisponde alla sintesi delle due reti combinatorie RCA e RCB del modello:



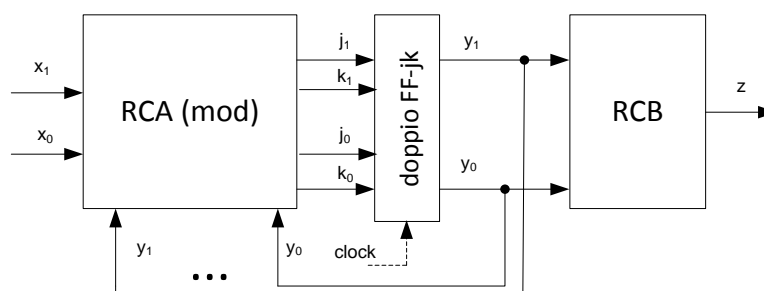
Per effettuare la sintesi di RCA, RCB, devo scegliere una codifica per gli stati interni. Siccome sono 4, servono due bit. La codifica può essere quella che ho scritto nella descrizione Verilog sopra, riportata nella tabella sottostante (si può scegliere una codifica qualunque, non è un problema).

x_1x_0						S_{int}		y_1y_0		x_1x_0						z	
		00	01	11	10							00	01	11	10		
S_0	S0	S0	S1	S0	0	S_0	00	y_1y_0	00	00	00	01	00	00	0		
S_1	S0	S2	S1	S0	0	S_1	01		01	01	00	10	01	00	0		
S_2	S0	S0	S1	S3	0	S_2	10		10	11	00	00	01	00	1		
S_3	S0	S0	S1	S0	1	S_3	11		11	10	00	00	01	11	0		

a_1a_0

La sintesi sarà: $a_1 = \overline{x_1} \cdot \overline{x_0} \cdot \overline{y_1} \cdot y_0 + \overline{x_1} \cdot \overline{x_0} \cdot y_1 \cdot \overline{y_0}$, $a_0 = x_1 \cdot x_0 + x_1 \cdot y_1 \cdot \overline{y_0}$, $z = y_1 \cdot y_0$.

Posso anche fare la sintesi in un modo diverso. Visto che i **FF-JK** servono, appunto, a **memorizzare dei bit**, posso usare questi come elementi di memorizzazione. Ne viene fuori un altro modello:



Sempre di una rete di Moore si tratta. 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 FF-JK 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.

		x_1x_0			
		y_1y_0	00	01	11
a_1a_0	00	00	00	01	00
	01	00	10	01	00
	11	00	00	01	00
	10	00	00	01	11

		x_1x_0			
		y_1y_0	00	01	11
a_1	00	0	0	0	0
	01	0	1	0	0
	11	0	0	0	0
	10	0	0	0	1

		x_1x_0			
		y_1y_0	00	01	11
a_0	00	0	0	1	0
	01	0	0	1	0
	11	0	0	1	0
	10	0	0	1	1

Ripartiamo dalla tabella in cui si individua la codifica del **nuovo stato interno** sulla base della codifica dello **stato interno presente** e dello **stato di ingresso**. Isoliamo la variabile a_1 . Guardo la **tabella di applicazione** del FF-JK, e considero il **nuovo stato interno** (a_1) come la nuova uscita q' , e lo stato interno presente (y_1) come l'uscita corrente (q), e posso facilmente dedurre che:

- se y_1 valeva 0 e deve continuare a valere 0 (cella 1,1), devo pilotare il FF-JK con $j=0, k=-$ (**o resetto o conservo**, tanto è lo stesso)
- se y_1 valeva 0 e deve valere 1 (cella 2,2), devo pilotare il FF-JK con $j=1, k=-$ (**setto o commuto**)
- se y_1 valeva 1 e deve continuare a valere 1 (cella 3,1), devo pilotare il FF-JK con $j=-, k=0$ (**setto oppure conservo**)
- se y_1 valeva 1 e deve valere 0 (cella 4,4), devo pilotarlo il FF-JK con $j=-, k=1$ (**resetto o commuto**).

Lo stesso faccio per la sintesi di a_0 . Le mappe che vengono fuori sono piene di non specificati.

q q'		jk	x ₁ x ₀				
			y ₁ y ₀	00	01	11	10
0 0		0-	00	0-	0-	0-	0-
0 1		1-	01	0-	1-	0-	0-
1 0		-1	11	-1	-1	-1	-1
1 1		-0	10	-1	-1	-1	-0
			j ₁ k ₁				

		x ₁ x ₀				
		y ₁ y ₀	00	01	11	10
		00	0-	0-	1-	0-
		01	-1	-1	-0	-1
		11	-1	-1	-0	-1
		10	0-	0-	1-	1-
		j ₀ k ₀				

		y ₁ y ₀	z
		00	0
	01	01	0
	10	10	0
	11	11	1

$$j_1 = \overline{x_1} \cdot x_0 \cdot y_0, \quad k_1 = \overline{x_1} + x_0 + y_0, \quad j_0 = x_1 \cdot y_1 + x_1 \cdot x_0, \quad k_1 = \overline{x_1} + \overline{x_0}, \quad z = y_1 \cdot y_0.$$

Si noti che:

- la sintesi di RCA è piuttosto semplice;
- la sintesi di RCB non cambia, in quanto dipende soltanto dalla codifica degli stati interni (e non dal modello di sintesi adottato).

3.5.3 Esercizio – Rete di Moore

- 1 Descrivere una rete sequenziale sincronizzata di Moore ad 1 ingresso che riconosce la sequenza **0,0,1,0,1,1,0**. Si presti particolare attenzione a non perdere nessuna sequenza, e non si considerino valide sequenze interallacciate.
- 2 Sintetizzare la rete descritta al punto precedente. La sintesi delle reti RCA e RCB deve essere a costo minimo in forma SP.

Soluzione

1) La sequenza consta di 7 stati di ingresso consecutivi. Devo pertanto prevedere $7+1=8$ stati interni, l'ultimo dei quali avrà un'uscita pari ad 1. La rete può essere descritta come in figura (gli stati in neretto nella tabella corrispondono all'evoluzione degli stati conseguente al riconoscimento di una sequenza).

x \	0	1	z
S0	S1	S0	0
S1	S2	S0	0
S2	S2	S3	0
S3	S4	S0	0
S4	S2	S5	0
S5	S1	S6	0
S6	S7	S0	0
S7	S1	S0	1

2) Per quanto riguarda la sintesi, si può osservare quanto segue:

- adottando la codifica degli stati $S_i = (i)_{b_2}$, la rete RCB è
 $z = y_2 \cdot y_1 \cdot y_0$, a costo minimo.
- la rete RCA ha 4 ingressi (3 variabili di stato y_2, y_1, y_0 , 1 variabile di ingresso).

bile di stato più significativa come variabile in colonna insieme agli ingressi:

Decido di adottare un modello di sintesi con D-FF come elementi di marcatura. Per svolgere la sintesi metto la varia-

x y ₂ y ₁ y ₀		z	
		0	1
000	001	000	0
001	010	000	0
010	010	011	0
011	100	000	0
100	010	101	0
101	001	110	0
110	111	000	0
111	001	000	1

x	y ₂ =0		y ₂ =1	
	0	1	0	1
00	001	000	010	101
01	010	000	001	110
10	010	011	111	000
11	100	000	001	000



y ₂ x y ₁ y ₀	a ₂ a ₁ a ₀			
	00	01	11	10
00	001	000	101	010
01	010	000	110	001
11	100	000	000	001
10	010	011	000	111

Per la sintesi di tutte e tre le variabili di uscita tutti gli implicantti sono essenziali.
Si ottiene quanto segue:

$$a_2 = y_2 \cdot \overline{y_1} \cdot \overline{x} + \overline{y_2} \cdot y_1 \cdot y_0 \cdot \overline{x} + y_2 \cdot y_1 \cdot \overline{y_0} \cdot \overline{x}$$

$$a_1 = \overline{y_2} \cdot y_1 \cdot \overline{y_0} + y_2 \cdot \overline{y_0} \cdot \overline{x} + y_2 \cdot \overline{y_1} \cdot y_0 \cdot x + \overline{y_2} \cdot \overline{y_1} \cdot y_0 \cdot \overline{x}$$

$$a_0 = \overline{y_2} \cdot \overline{y_1} \cdot \overline{y_0} \cdot \overline{x} + y_2 \cdot \overline{y_1} \cdot \overline{y_0} \cdot x + \overline{y_2} \cdot y_1 \cdot \overline{y_0} \cdot x + y_2 \cdot y_1 \cdot \overline{x} + y_2 \cdot y_0 \cdot \overline{x}$$

3.5.4 Esercizio (per casa)

Descrivere una rete sequenziale sincronizzata di Moore che ha due variabili di ingresso j e k , ed una variabile di uscita q e si comporta come il flip-flop JK, differenziandosene per la diversa evoluzione nel solo caso $j = k = 1$.

In tal caso infatti porta q ad 1 se la volta precedente in cui lo stato d'ingresso $j = k = 1$ si era presentato, l'uscita era stata resettata; porta q a 0 se la volta precedente in cui lo stato d'ingresso $j = k = 1$ si era presentato, l'uscita era stata settata.

NOTA: la prima volta che si presenta lo stato d'ingresso $j = k = 1$, allora porta q ad 1 .

2) Sintetizzare la rete a porte NOR.

[Soluzione](#)

3.5.5 Esercizio (per casa)

Si consideri una rete sequenziale sincronizzata di Moore con due variabili di ingresso e due variabili di uscita. Interpretando le due variabili di uscita come un numero naturale a due cifre in base due, il comportamento della rete è il seguente:

- quando gli ingressi sono *diversi*, la rete conta in avanti (modulo 4)
- quando gli ingressi sono *uguali*, la rete conta all'indietro (modulo 4)

Descrivere e sintetizzare la rete. Calcolare il costo (a porte e a diodi) della rete combinatoria RCA.

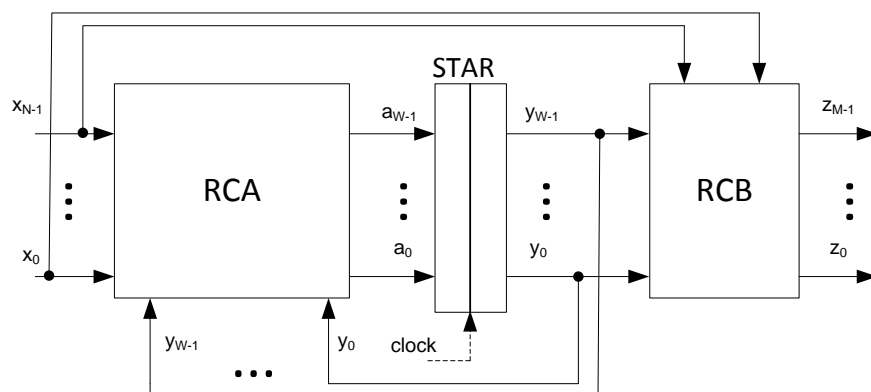
Parte facoltativa: sintetizzare la rete RCA utilizzando *esclusivamente* porte XOR e porte NOT.

Calcolare il costo (a porte e a diodi) della rete combinatoria RCA così realizzata, assumendo che il costo di una porta XOR sia pari ad uno.

[Soluzione](#)

3.6 Modello di Mealy

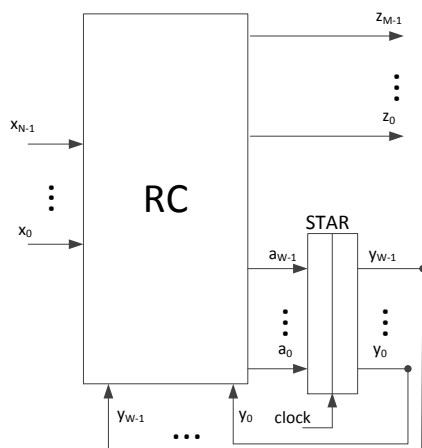
Nel modello di Moore, l'uscita è funzione **soltanto dello stato interno presente**, tramite la legge $B: S \rightarrow Z$. Se si consente a tale legge di essere più generale, scrivendo $B: X \times S \rightarrow Z$, si ottengono reti realizzate secondo il **modello di Mealy**.



Le reti RCA e RCB hanno, secondo questo modello, **gli stessi ingressi**.

Pertanto posso disegnare una rete di Mealy anche in questo modo.

Riprendiamo le leggi di temporizzazione viste a suo tempo:



$$T \geq T_{hold} + T_{a_monte} + T_{RC} + T_{setup}$$

(percorso da ingresso a registro)

$$T \geq T_{prop} + T_{RC} + T_{setup}$$

(percorso da registro a registro)

$$T \geq T_{hold} + T_{a_monte} + T_{RC} + T_{a_valle}$$

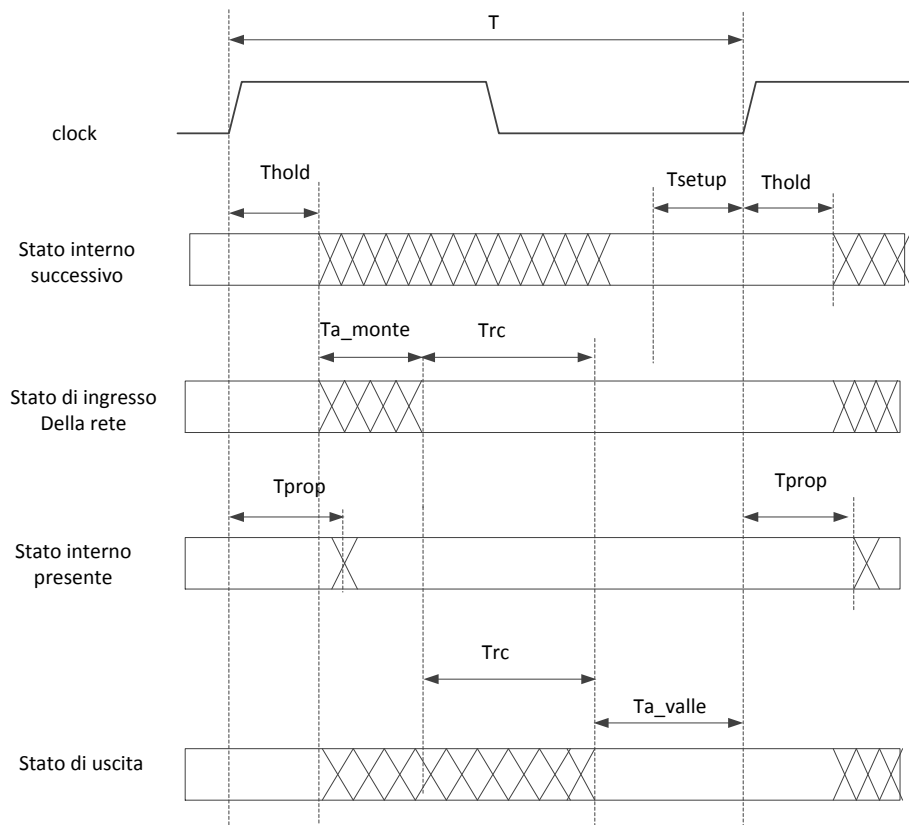
(percorso da ingresso a uscita)

$$T \geq T_{prop} + T_{RC} + T_{a_valle}$$

(percorso da registro a uscita)

Come in precedenza, la seconda disuguaglianza è praticamente implicata dalla prima, e possiamo trascurarla. Allo stesso modo, la **quarta è (praticamente) implicata dalla terza**. Delle due che rimangono, di sicuro la **terza** è la **più vincolante**, in quanto somma i **tre tempi più lunghi**: quelli della rete combinatoria **RC** e quelli del mondo esterno “**a monte**” e “**a valle**”. In una rete di Moore, se ricordate, questi tempi si trovavano nelle equazioni, ma al massimo sommati a due a due (mai tutti e tre insieme). Ciò comporta che, in genere, una rete di Mealy, **il clock debba andare più lentamente che in una rete di Moore** (a parità di condizioni sulla temporizzazione imposte dal mondo esterno).

Nelle disequazioni di temporizzazione si è indicato con T_{RC} il tempo di attraversamento della rete combinatoria, senza distinguere tra i diversi percorsi. Se la RC è sintetizzata in modo ottimizzato, infatti, i tempi di attraversamento dovrebbero essere più o meno uguali tra ogni coppia di morsetti.



```

module Rete_di_Mealy(ZM-1,...,Z0,XN-1,...,X0,clock,reset_);
  input clock,reset_;
  input XN-1,...,X0;
  output ZM-1,...,Z0;
  reg [W-1:0] STAR; parameter S0=codifica0,...,SK-1=codificaK-1;
  assign {ZM-1,...,Z0} = (STAR==S0)? ZS0(XN-1,...,X0) :
    ...
    ...
    ...
    /* (STAR==SK-1) */ ZSK-1(XN-1,...,X0);
  always @(reset_==0) #1 STAR<=stato_interno_iniziale;
  always @(posedge clock) if (reset_==1) #3
    casex (STAR)
      S0 : STAR<=AS0(XN-1,...,X0);
      ...
      ...
      SK-1 : STAR<=ASK-1(XN-1,...,X0);
    endcase
endmodule

```

La legge B ha una struttura diversa da quella di una rete di Moore

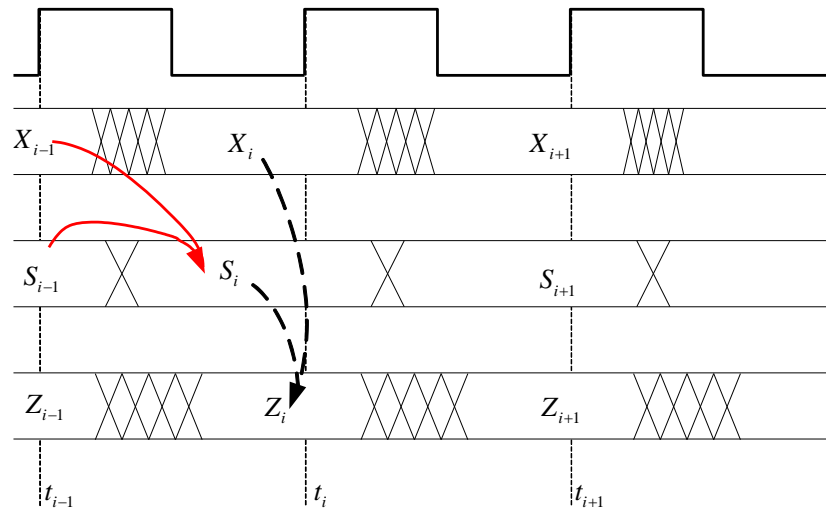
Il vantaggio di questo modello, rispetto al precedente modello di Moore, è che **al variare dell'ingresso** posso produrre un **nuovo stato di uscita** senza dover aspettare il successivo fronte del clock. Nelle reti di Moore l'uscita varia quando arriva il clock (in realtà un po' dopo, per via del

tempo di propagazione), perché dipende **solo dallo stato interno**, nelle reti di Mealy varia **anche quando varia lo stato di ingresso**.

$$\begin{aligned} S[t_{i+1}] &= A(X[t_i], S[t_i]) \\ Z[t_i] &= B(X[t_i], S[t_i]) \end{aligned}$$

Confrontare con la temporizzazione di una rete di Moore, dove è:

$$Z[t_i] = B(S[t_i]).$$



Si dice che

- nelle reti di Moore, **l'uscita è un clock in ritardo rispetto all'ingresso che l'ha generata**. Dipende, infatti, soltanto dal *penultimo* stato di ingresso, quello al clock precedente. Infatti, è:

$$Z[t_i] \propto X[t_{i-1}], X[t_{i-2}], \dots, X[t_0], S[t_0]$$

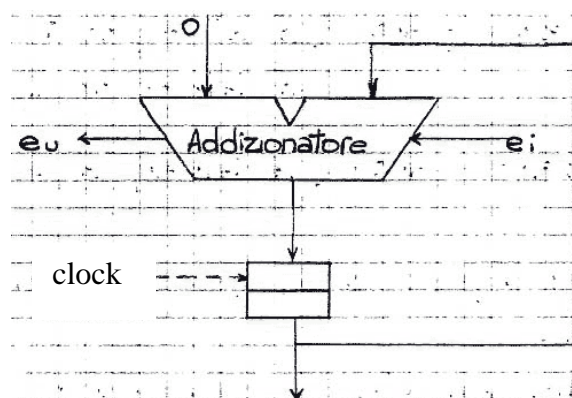
- nelle reti di Mealy, **l'uscita dipende anche dall'ultimo stato di ingresso, quello presente al clock attuale**.

$$Z[t_i] \propto \mathbf{X[t_i]}, X[t_{i-1}], X[t_{i-2}], \dots, X[t_0], S[t_0]$$

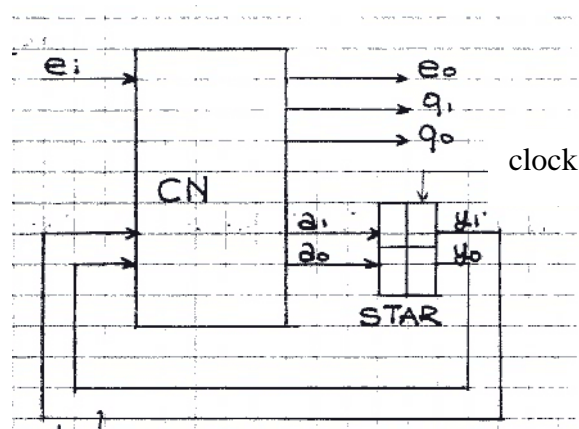
Essendo la legge **B** più flessibile che nel caso precedente, in genere si riescono a risolvere gli stessi problemi con un numero **minore di stati interni**.

3.6.1 Esempio: sintesi del contatore espandibile in base 3

Un esempio di rete di Mealy è il **contatore espandibile**, qualunque sia il suo numero di cifre, base e codifica. Infatti, il **riporto uscente** è funzione **combinatoria dello stato interno e del riporto entrante**, e quest'ultimo è un **ingresso** alla rete.



Prendiamo, ad esempio, il contatore espandibile **ad una cifra in base 3**. Ne abbiamo dato una sintesi in termini euristici. Possiamo darne adesso una descrizione in termini di **tabella di flusso** per una rete di Mealy. Stavolta, però, le uscite non sono una **colonna**, **ma una tabella a parte**, in quanto dipendono anche dallo stato di ingresso.



Il contatore avrà **tre stati interni**, corrispondenti ai tre possibili contenuti del registro. In funzione dello stato interno e dello stato di ingresso marcato, calcolerà un **nuovo stato interno**, che verrà marcato al prossimo fronte del clock, **ed uno stato di uscita**, che sarà **presentato immediatamente**, senza aspettare il clock successivo (come invece farebbe una rete di Moore).

	e_i	$q_1 q_0$		e_u		e_i	e_u		$q_1 q_0$	
		0	1				0	1		
(00)	S0	S0 00 0	S1 00 0			S0	S0 0	S1 0		00
(01)	S1	S1 01 0	S2 01 0			S1	S1 0	S2 0		01
(10)	S2	S2 10 0	S0 10 1			S2	S2 0	S0 1		10

Mentre le uscite $q_1 q_0$ sono **uscite del registro**, e **dipendono solo dallo stato interno** (uscite di Moore), l'uscita **eu** è combinatoria, e dipende **sia dallo stato che dall'ingresso** (uscita **di Mealy**).

Posso scriverla tutta nella tabella, o con le uscite $q_1 q_0$ in una colonna a parte.

Adottando le codifiche (ovvie) $S0='B00$, $S1='B01$, $S2='B10$, e scrivendo la tabella delle transizioni dalla tabella di flusso si ottiene molto velocemente **la stessa sintesi** già vista a suo tempo:

$y_1 y_0 \backslash e_i$	0	1
00	0	0
01	0	0
11	-	-
10	0	1

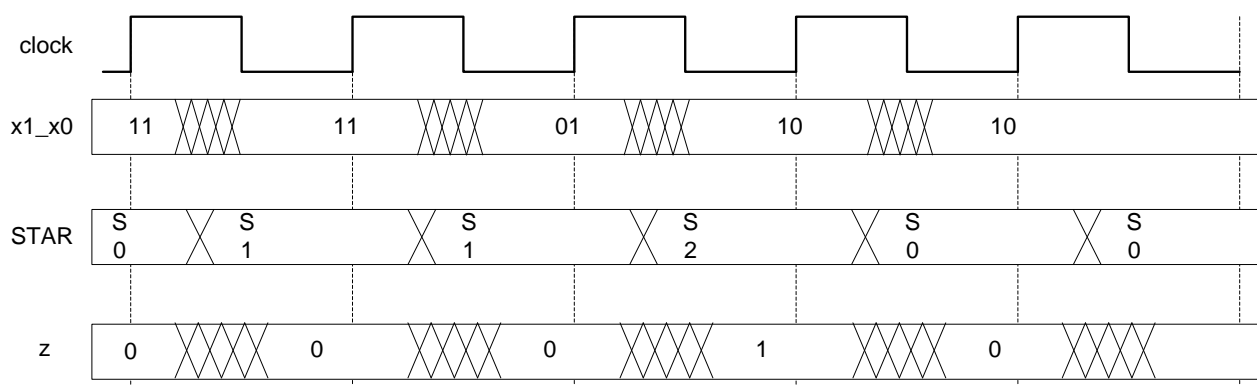
da cui ricaviamo l'implicante
 $e_0 = y_1 \cdot e_1$

Per il resto della sintesi (di q_1, q_0, a_1, a_0) si procede nello stesso modo.

Ovviamente esistono modelli di sintesi alternativi anche per le reti di Mealy. Ad esempio posso sempre utilizzare dei **FF JK** come elementi di marcatura.

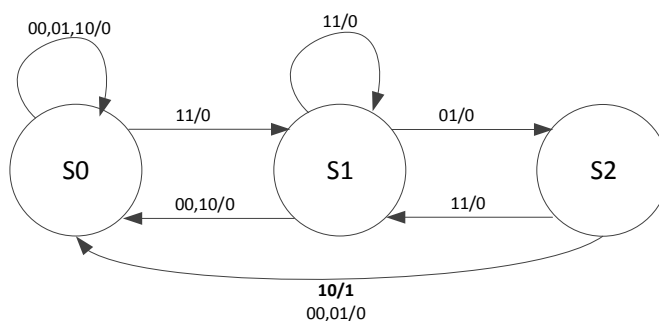
3.6.2 Esempio: riconoscitore di sequenza 11, 01, 10

Abbiamo già sintetizzato questa rete come rete di Moore. Vediamo di realizzarlo come rete di Mealy. Potremmo avere la seguente temporizzazione per il riconoscimento della sequenza corretta:



Dove, se sono in S2 (cioè ho già riconosciuto due passi corretti 11,01) e vedo ingresso 10, **posso direttamente mettere l'uscita ad 1 senza aspettare il clock successivo**, e poi tornare in S0 per prepararmi a riconoscere una nuova sequenza.

Volendo descrivere questa rete con un **grafo di flusso**, dove si vede meglio cosa fare, i valori delle uscite vanno messi **non negli stati**, **ma sugli archi**.



Analogamente, nella tabella di flusso la legge B va scritta in forma tabellare come la A.

X_1X_0		00	01	11	10
	S0	S0/0	S0/0	S1/0	S0/0
	S1	S0/0	S2/0	S1/0	S0/0
	S2	S0/0	S0/0	S1/0	S0/1

Attenzione a cosa si scrive in questo caso: nella tabella di flusso, l'evoluzione dello **stato** avverrà **all'arrivo del clock**, come sempre, mentre quella delle uscite avverrà **ad ogni t**, visto che c'è una legge combinatoria nel mezzo. È chiaro che almeno una riga di tutta la tabella **dovrà contenere uscite differenti**, altrimenti sto facendo una rete di Moore senza accorgermene.

Posso descrivere questa rete anche in Verilog, come segue:

```

module Riconoscitore_di_Sequenze(z,x1_x0,clock,reset_);
  input clock,reset_;
  input [1:0] x1_x0;
  output z;
  reg [1:0] STAR; parameter S0='B00,S1='B01,S2='B10;
  assign z=((STAR==S2) & (x1_x0=='B10'))?1:0;

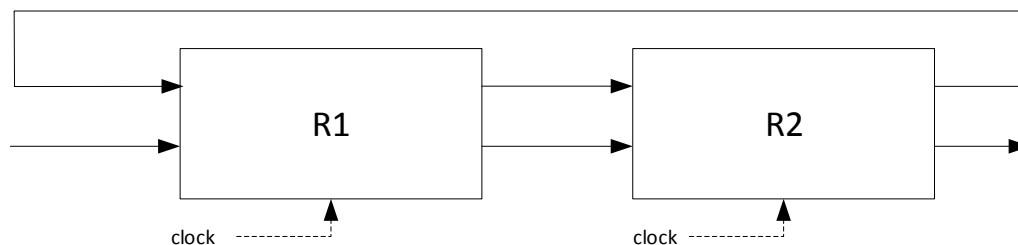
  always @(reset_==0) #1 STAR<=S0;
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: STAR<=(x1_x0=='B11)?S1:S0;
      S1: STAR<=(x1_x0=='B01)?S2:(x1_x0=='B11)?S1:S0;
      S2: STAR<=(x1_x0=='B11)?S1:S0;
    endcase
endmodule

```

Nota finale: Ci si può chiedere se sia **più opportuno**, data una specifica, realizzarla tramite una **rete di Mealy o di Moore**. A un primo sguardo potrebbe sembrare che, visto che la legge B per reti di Mealy è più generale (in quanto dipende anche dagli ingressi), la **potenza descrittiva del modello di Mealy sia maggiore del modello di Moore**, cioè che esistano problemi che **non si possono risolvere con reti di Moore**, ma soltanto con reti di Mealy. In realtà **non è così**. Qualunque problema sia risolubile con un modello è risolubile anche con l'altro. Da Moore a Mealy è abbastanza ovvio (basta osservare che, di fatto, Moore è un caso particolare di Mealy). Da Mealy a Moore è **meno ovvio**: ci sono tecniche meccaniche di trasformazione (che non vedremo), il cui trucco è che **può essere necessario aumentare il numero degli stati interni nel passaggio da Mealy a Moore**. Ad esempio, il riconoscitore di sequenza può essere realizzato come rete di Mealy con **tre stati** interni (come rete di Moore ce ne volevano quattro).

Dal punto di vista della **velocità di risposta**, l'uscita di una rete di Mealy è sempre “un clock in anticipo”, e quindi una rete di Mealy risulta più veloce. Però, se vado a guardare le temporizzazioni, vedo che una rete di Mealy avrà in genere il **clock più lento**.

Ciò che fa la differenza sostanziale, quindi, non è né la potenza descrittiva né la velocità. È la **trasparenza delle uscite**. Date due RSS generiche, posso montarle in questo modo?



Lo posso fare **soltanto se il ramo di sopra non è un anello combinatorio**, altrimenti sto realizzando una **RSA** senza accorgermene. Affinché non ci sia un anello di reti combinatorie, è **necessario che almeno una delle due reti sia di Moore**, in quanto questo garantisce che ci sia **almeno un registro dentro l'anello** (che quindi non è più un anello combinatorio). Se sono entrambe di Mealy, si possono creare dei problemi. Le reti di Mealy sono **trasparenti**, cioè adeguano le proprie uscite mentre sono sensibili agli ingressi. Quelle di Moore sono **non trasparenti**.

3.6.3 Esercizio

Descrivere e sintetizzare una rete sequenziale sincronizzata di Mealy che ha due variabili di ingresso x_1 e x_0 , ed una variabile di uscita z . La rete evolve nel seguente modo: se lo stato d'ingresso corrente è **uguale al precedente**, $z = x_1 \text{ AND } x_0$, altrimenti, se lo stato d'ingresso corrente **non è uguale al precedente**, $z = x_1 \text{ XOR } x_0$.

Nota: il primo stato di uscita della rete è non significativo.

3.6.4 Soluzione

Sarà certamente necessario **memorizzare l'ultimo stato di ingresso** visto dalla rete. Pertanto, essendo 4 gli stati di ingresso possibili, non posso fare a meno di avere **quattro stati interni**. Chiamiamoli S_0 , S_1 , S_2 , S_3 , e disegniamo la tabella di flusso. Posso associare lo stato interno al precedente stato di ingresso nel seguente modo:

S. interno	S. ingresso precedente
S_0	00
S_1	01
S_2	11
S_3	10

Ciò significa che la tabella di flusso, relativamente alla parte che sintetizza la legge A, è la seguente:

La legge A non dipende dallo stato interno (infatti è identica su ogni riga). La parte di rete RC che produce il nuovo stato interno non ha in ingresso le uscite del registro.

x_1x_0	00	01	11	10
S_0	$S_0/0$	$S_1/1$	$S_2/0$	$S_3/1$
S_1	$S_0/0$	$S_1/0$	$S_2/0$	$S_3/1$
S_2	$S_0/0$	$S_1/1$	$S_2/1$	$S_3/1$
S_3	$S_0/0$	$S_1/1$	$S_2/0$	$S_3/0$

Se, inoltre, adotto un modello di **sintesi con D-FF come elementi di marcatura** e scelgo come codifica per gli stati interni la più ovvia (cioè **S0=00**, **S1=01**, etc.), ottengo anche che $a_1 = x_1$, $a_0 = x_0$: la rete che calcola lo stato interno successivo è costituita da due cortocircuiti.

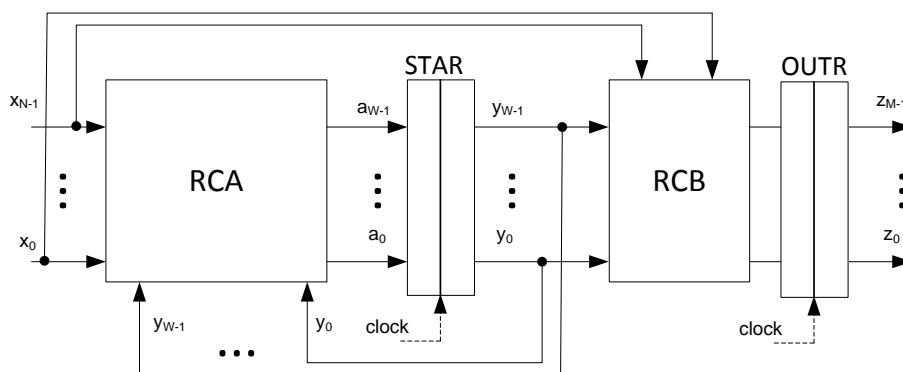
Per quanto riguarda le uscite, basta leggere le specifiche ed aver presente che lo stato interno codifica lo stato di ingresso precedente. Sulla **diagonale** dovrò eseguire l'operazione $z = x_1 \cdot x_0$, fuori dalla diagonale avrò $z = x_1 \otimes x_0$.

Posso quindi fare la sintesi della parte di RC che gestisce le uscite come segue (ad esempio in forma PS): $z = (x_1 + x_0) \cdot (\bar{x}_0 + y_1 + \bar{y}_0) \cdot (\bar{x}_1 + \bar{y}_1 + y_0) \cdot (\bar{x}_1 + \bar{x}_0 + y_1)$

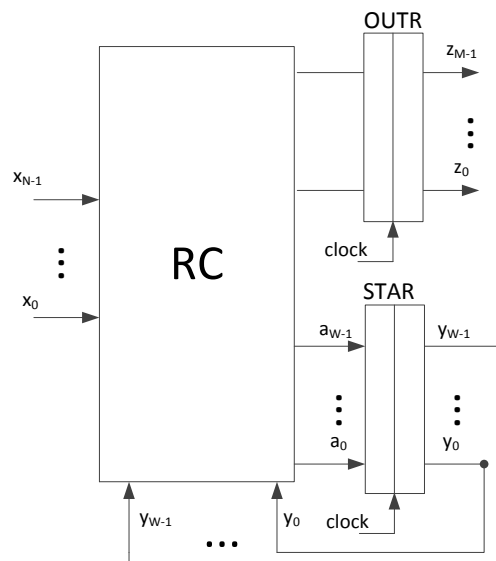
3.7 Modello di Mealy ritardato

Si parte da una rete di Mealy, e si mette in uscita un **registro OUTR**. In questo modo, le uscite:

- variano sempre **all'arrivo del clock**, dopo un tempo T_{prop} (dove l'aggettivo "ritardato");
- variano in maniera **netta, senza oscillazioni** (come invece può succedere in una rete di Mealy se gli ingressi ballano un po' prima di stabilizzarsi);
- rimangono stabili per **l'intero ciclo di clock**.
- sono **non trasparenti**.



Come al solito, RCA e RCB hanno gli stessi ingressi, e quindi posso disegnare il tutto così:



Vediamo intanto di definire formalmente le proprietà di una rete di Mealy ritardato.

- 1) Ha una **legge di evoluzione nel tempo** del tipo $A: \mathbf{X} \times \mathbf{S} \rightarrow \mathbf{S}$, che mappa quindi una coppia (stato di ingresso, stato interno) in un nuovo stato interno.
- 2) Una **legge di evoluzione nel tempo** del tipo $B: \mathbf{X} \times \mathbf{S} \rightarrow \mathbf{Z}$, che mappa quindi una coppia (stato di ingresso, stato interno) in un nuovo stato di uscita.
- 3) Si adegua alla seguente **legge di temporizzazione**:

“Dato S, stato interno presente (marcato) ad un certo istante, e dato X stato di ingresso ad un certo istante **precedente l’arrivo di un segnale di sincronizzazione**,

- 1) individuare SIA il nuovo stato interno da marcare $S' = A(S, X)$, SIA il nuovo stato di uscita $Z = B(S, X)$
- 2) attendere T_{prop} dopo l’arrivo del segnale di sincronizzazione
- 3) promuovere S' al rango di stato interno marcato, e promuovere Z al rango di nuovo stato di uscita

Attenzione a capire **bene** una cosa (non averla capita **ora** rende la soluzione dei compiti d’esame impossibile **dopo**):

Lo stato di uscita cambia **dopo il clock**, ed il suo valore dipende dallo stato di ingresso e dallo stato interno marcato **precedenti all’arrivo del clock**.

Prendiamo in esame le condizioni di temporizzazione, derivandole dalle equazioni generali che avevamo scritto a suo tempo:

$$T \geq T_{hold} + T_{a_monte} + T_{RC} + T_{setup}$$

(percorso da ingresso a registro)

$$T \geq T_{prop} + T_{RC} + T_{setup}$$

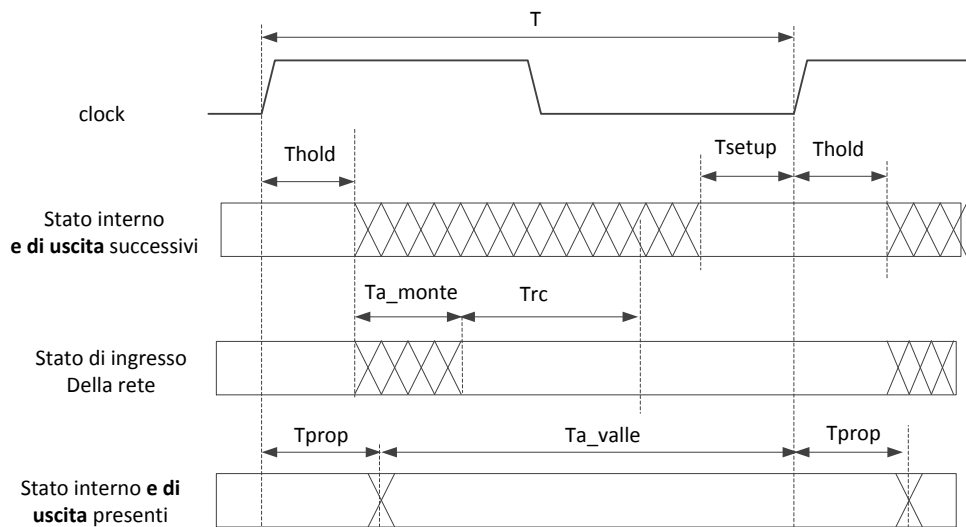
(percorso da registro a registro)

$$T \geq T_{prop} + T_{a_valle}$$

(percorso da registro a uscita)

Di queste, al solito, la seconda sarà più o meno implicata dalla prima, e potremo trascurarla. La più vincolante è quindi la **prima**.

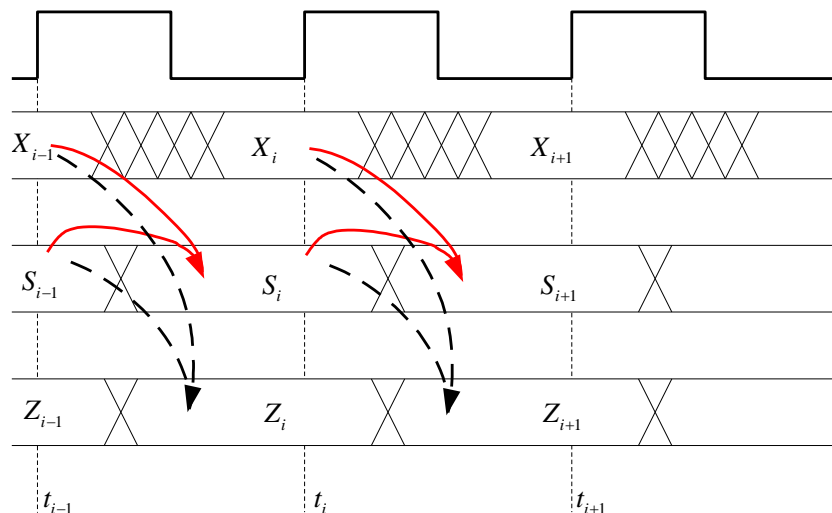
Se le condizioni di temporizzazione sono rispettate (vediamo tra un attimo), una rete di Mealy ritardato si evolve in modo **deterministico**. Detti t_i , $i \geq 0$, gli **istanti di sincronizzazione**, e detti $X[t_i], S[t_i], Z[t_i]$ gli stati di **ingresso, interno e di uscita** all'istante t_i , sarà:



$$\begin{aligned} S[t_{i+1}] &= A(X[t_i], S[t_i]) \\ Z[t_{i+1}] &= B(X[t_i], S[t_i]) \end{aligned}$$

e tali nuovi stati interni e di uscita saranno resi disponibili **dopo** T_{prop} .

Confrontare con quelli per reti di Moore e Mealy



La descrizione in Verilog di una rete di Mealy ritardato è la seguente:

```
module Rete_di_Mealy_Ritardato(zM-1,...,z0,xN-1,...,x0,clock,reset_);
  input clock,reset_;
  input xN-1,...,x0;
  output zM-1,...,z0;

  reg [W-1:0] STAR; parameter S0=codifica0,...,SK-1=codificaK-1;
  reg [M-1:0] OTR; assign {zM-1,...,z0}=OTR;

  always @(reset_==0) #1 begin OTR<=...; STAR<=...; end
  always @(posedge clock) if (reset_==1) #3
    case (STAR)
      S0 : begin OTR<=ZS0(xN-1,...,x0); STAR<=AS0(xN-1,...,x0); end
      ...
      SK-1 : begin OTR<=ZSK-1(xN-1,...,x0); STAR<=ASK-1(xN-1,...,x0); end
    endcase
endmodule
```

Due o più assegnamenti procedurali **racchiusi tra begin...end** verranno resi operativi **contemporaneamente**, all'arrivo del clock. Pertanto, scriverli in un ordine o in un altro **non cambia niente (non è il C++!)**. In particolare, è ovvio che se uso OTR a destra di STAR<=, sto usando il **vecchio valore** (quello **prima** del fronte del clock), **non il nuovo**.

Posso anche descrivere una rete di Mealy ritardato con **tabelle e grafi di flusso** (ammesso che sia semplice abbastanza). In questo caso, la descrizione sarà **visivamente identica a quella di una rete di Mealy standard**, in quanto il fatto che sia **Mealy o Mealy ritardato** sta nella maniera di **rendere operativa la legge B**, non nella formulazione della legge stessa. Posso descrivere il **riconoscitore di sequenza** come rete di Mealy ritardato: la tabella di flusso sarà identica, sarà diversa la **temporizzazione delle uscite**. Infatti, in questo caso, **entrambe le parti della tabella** vengono rese vere all'arrivo del clock.

		X ₁ X ₀			
		00	01	11	10
S	S0	S0/0	S0/0	S1/0	S0/0
	S1	S0/0	S2/0	S1/0	S0/0
	S2	S0/0	S0/0	S1/0	S0/1

A livello di Verilog, invece, le due descrizioni saranno differenti. In particolare, la parte che gestisce l'**evoluzione di STAR** sarà **identica**, quella che gestisce l'evoluzione delle uscite sarà radical-

mente diversa, e consisterà in **assegnamenti procedurali al registro OUTR** (invece che assegnamenti continui, come avevamo nel caso di Mealy).

```

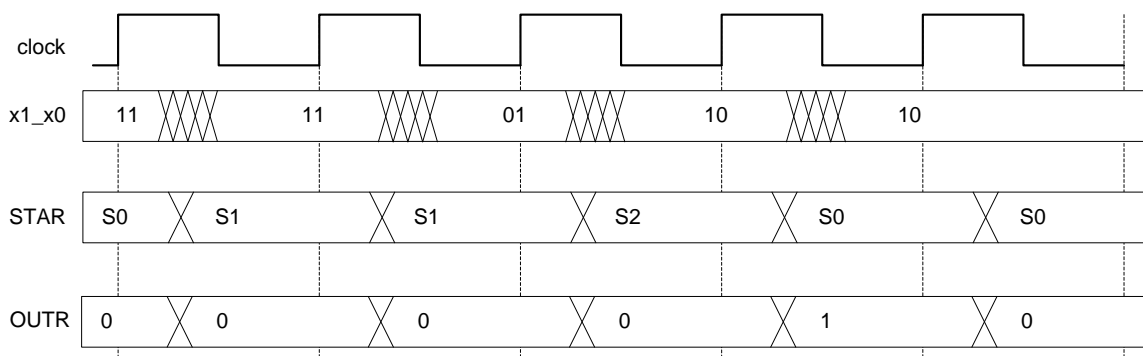
module Riconoscitore_di_Sequenze(z,x1_x0,clock,reset_);
  input clock,reset_;
  input [1:0] x1_x0;
  output z;

  reg [1:0] STAR; parameter S0='B00,S1='B01,S2='B10;
  reg OUTR; assign z=OUTR;

  always @(reset_==0) #1 begin OUTR<=0; STAR<=S0; end
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: begin OUTR<=0; STAR<=(x1_x0=='B11)?S1:S0; end
      S1: begin OUTR<=0;
              STAR<=(x1_x0=='B01)?S2:(x1_x0=='B11)?S1:S0; end
      S2: begin OUTR<=(x1_x0=='B10)?1:0; STAR<=(x1_x0=='B11)?S1:S0; end
    endcase
endmodule

```

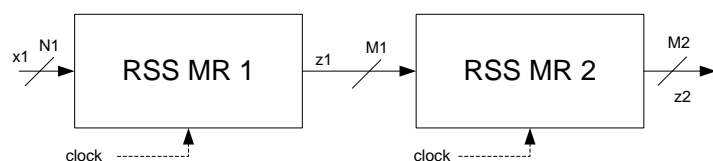
Il diagramma di temporizzazione relativo ad una possibile evoluzione di questa rete sarà quindi:



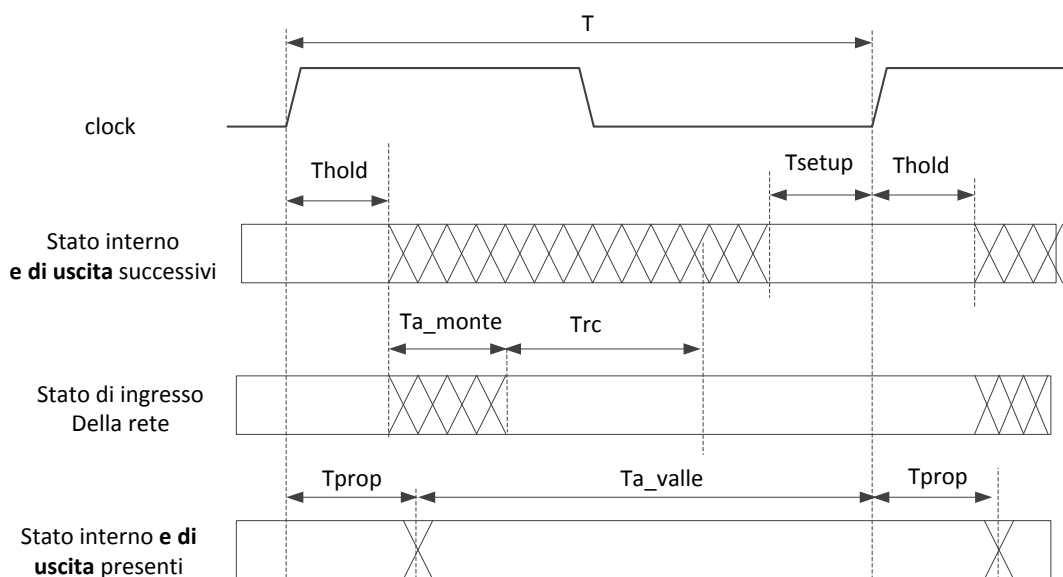
Il che giustifica ancora una volta l'aggettivo **ritardato**. L'uscita è in ritardo di un clock rispetto a quanto succedeva nel modello di Mealy.

Osservazione: posso certamente montare reti di Mealy ritardato in qualunque configurazione. Sono **non trasparenti**, al contrario delle reti di Mealy (in cui ho una connessione diretta ingresso-uscita). Inoltre, il fatto che **le uscite siano costanti per un intero periodo di clock** fa sì che possa mettere **catene di reti di Mealy ritardato arbitrariamente lunghe**, essendo sicuro che, se piloti gli ingressi di una rete a valle con le uscite di una rete a monte non avrò **mai problemi di temporizzazione**, in quanto le uscite sono certamente stabili a cavallo dei fronti di clock, e cambiano soltanto dopo.

Prendiamo due reti messe in questo modo (ed aventi lo stesso clock):



Nelle ipotesi di pilotaggio, lo stato di ingresso della **seconda** (che è anche lo stato di uscita della prima) deve essere pronto $T_{RC_2} + T_{setup}$ prima del fronte del clock. Nel nostro caso, lo stato di uscita della prima rete è pronto **già** T_{prop} **dopo il fronte del clock**. Allora basta che $T \geq T_{prop} + T_{RC_2} + T_{setup}$ perché la prima rete possa pilotare la seconda mantenendo i vincoli di temporizzazione. Visto che **questa disuguaglianza è già vera** (è infatti quella che regola il percorso da registro a registro dentro la RSS n. 2), allora non ci sono problemi a mettere reti di MR con lo stesso clock in cascata.

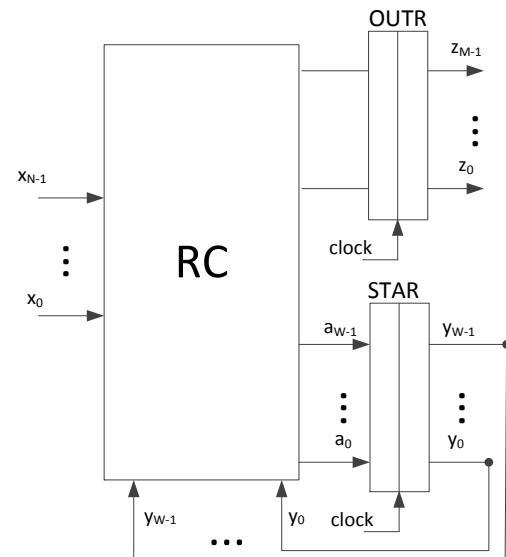


Riepilogando, le reti di Mealy ritardato:

- **sono non trasparenti;**
- hanno una legge B flessibile, che mi porta in genere a risolvere problemi usando un **numero minore di stati interni;**
- hanno **uscite stabili**, che cambiano in tempi certi;
- **non sono rallentate** da percorsi combinatori troppo lunghi (ricordare le disuguaglianze);
- possono essere **montate in cascata** senza problemi di pilotaggio;
- possono essere **montate in reazione** senza problemi di stabilità.

4 Descrizione e sintesi di reti sequenziali sincronizzate complesse

Il problema dei tre modelli di RSS visti finora è che vanno bene soltanto per reti molto semplici. Se si devono sintetizzare reti complesse, la stessa “pulizia concettuale” dei modelli diventa un limite. Prendiamo come punto di partenza il modello di Mealy ritardato, che abbiamo visto avere diverse caratteristiche interessanti.



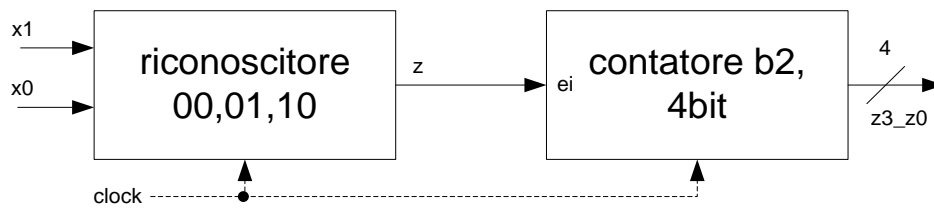
4.1 Linguaggio di trasferimento tra registri

Supponiamo di voler descrivere, usando questo modello, una rete che **conta, modulo 16, il numero di sequenze corrette 00, 01, 10 ricevute** in ingresso. In pratica, ogni volta che vede una sequenza corretta, incrementa di 1 il valore in uscita, rappresentato su 4 bit. Tale rete ha **due ingressi, quattro uscite, e quanti stati interni?**

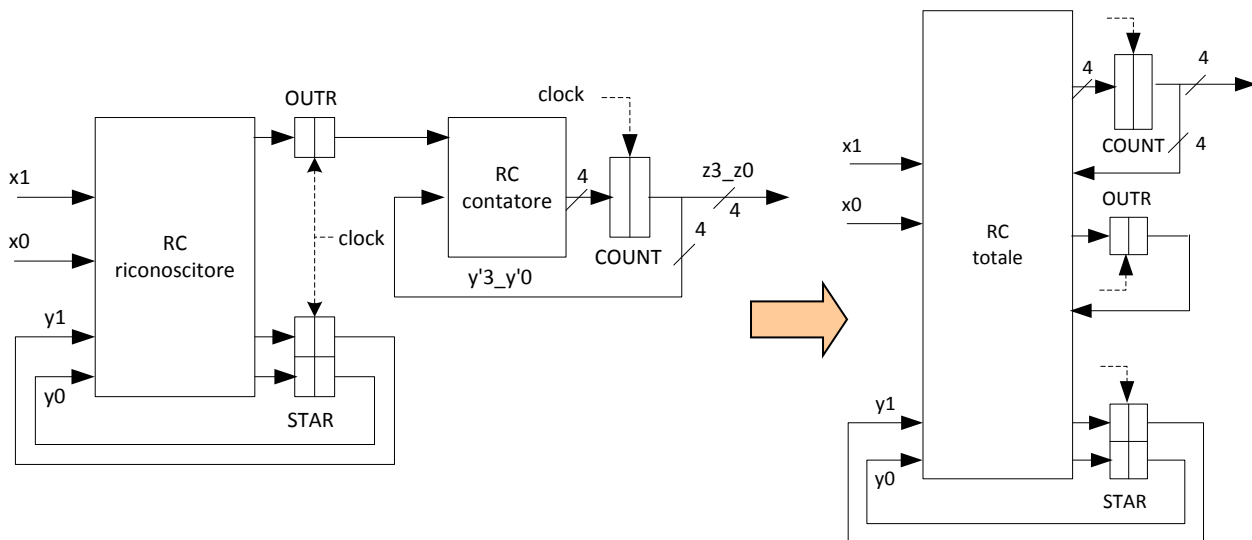
Facendo un conto a spanne, se ci vogliono 3 stati interni per riconoscere una sequenza di tre passi corretti (visto che adopero una rete di Mealy ritardato, perché se fosse stata di Moore ce ne sarebbero voluti 4), allora i 4 bit di uscita dovrebbero cambiare di uno ogni tre stati interni, e mi ci vogliono (a occhio) $3 \cdot 16 = 48$ stati interni. Ci vuole un registro **STAR a 6 bit**, la descrizione e la sintesi diventano ingestibili (in Verilog, il blocco `always` avrebbe un `case` a 48 etichette...).

La maggior parte di voi avrà osservato che, per realizzare una rete così fatta, la soluzione che sto proponendo non è certamente ottimale. Si farebbe molto prima a:

- Sintetizzare un riconoscitore di una sequenza come rete di MR, con **un bit di uscita**;
- Sintetizzare un contatore a 4 bit in base 2, che prende come ingresso **ei** (riporto entrante) l'uscita del riconoscitore, e produce esso stesso un'uscita su 4 bit.



Il contatore, se non considero **il riporto uscente** (del quale infatti nulla mi interessa, ai fini della risoluzione del mio problema) è una rete di Moore, oppure, se vogliamo, di Mealy ritardato (in quanto l'uscita che rappresenta il numero in base 2 su 4 bit è supportata direttamente da un registro). Vediamo come è fatta questa rete con maggior dettaglio:



Si ricava immediatamente la struttura a destra, che **non è una struttura di Mealy ritardato**, in quanto non distinguo più **soltanto due registri**, uno dei quali ha variabili che rientrano (STAR) e l'altro no (OUTR). Ho **tanti registri**, che possono supportare o meno variabili di uscita (ad esempio, OUTR non supporta variabili di uscita), il cui contenuto può comunque essere dato in ingresso alle reti combinatorie. In particolare, l'ingresso di COUNT sarà funzione dell'uscita di OUTR, e soprattutto sarà funzione **dell'uscita di COUNT stesso**.

Sono arrivato ad un modello **più generale**, in cui:

- Ho un registro di **stato STAR**, che svolge le stesse funzioni che in una RSS qualunque;
- posso usare quanti altri registri voglio, della capacità che voglio. Tali registri prendono il nome di **registri operativi** (ma questo non è un grosso miglioramento, tanto valeva fare un registro solo OUTR "molto grande");
- posso usare **il contenuto dei registri operativi (oltre che di STAR) per fornire ingresso a reti combinatorie**, che prepareranno l'ingresso ad altri registri, e così via. **Questo** è un grosso miglioramento rispetto al modello di Mealy ritardato;

- le uscite sono **tutte sostenute da registri operativi**, come nel modello di Mealy ritardato, anche se non necessariamente un registro operativo deve per forza sostenere un'uscita.

Con un simile modello posso risolvere problemi **complessi**, mantenendo descrizioni e sintesi molto **compatte**.

Esempio: contatore di sequenze 00,01,10

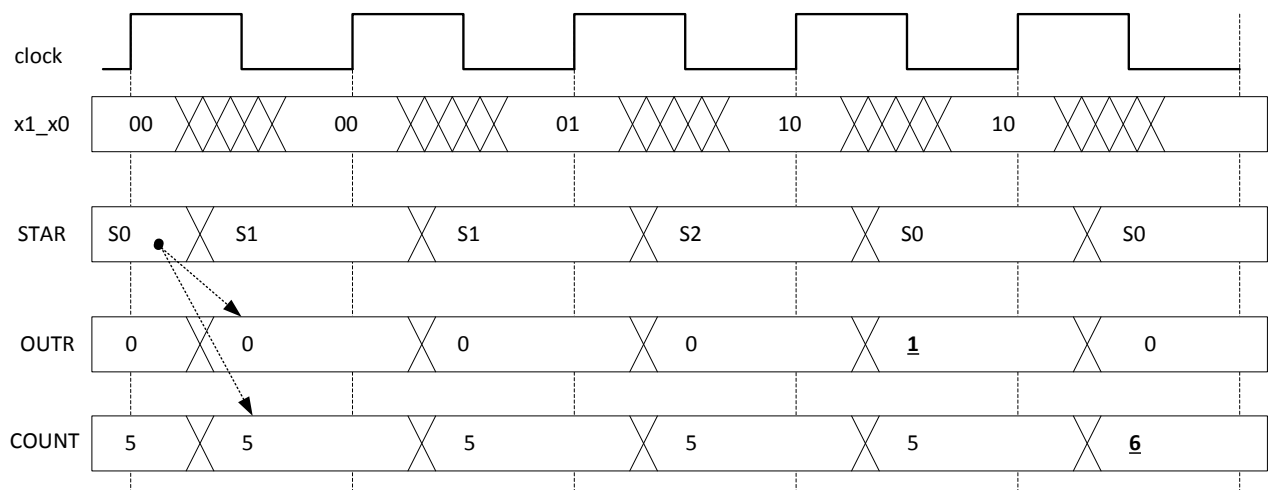
```
// Contatore di sequenze secondo il modello generalizzato
module Contatore_Sequenze(z3_z0, x1_x0, clock, reset_)
    input clock, reset_;
    input [1:0] x1_x0;
    output [3:0] z3_z0;

    reg [3:0] COUNT;
    reg      OTR;
    reg [1:0] STAR;

    parameter S0='B00, S1='B01, S2='B10;
    assign z3_z0=COUNT;

    always @(reset_ ==0) #1 begin OTR<=0; COUNT<=0; STAR<=S0; end
    always @(posedge clock) if (reset_==1) #3
        casex (STAR)
            S0 : begin OTR<=0; COUNT<=COUNT+OTR; STAR<=(x1_x0=='B00)?S1:S0; end
                end
            S1 : begin OTR<=0; COUNT<=COUNT+OTR;
                    STAR<=(x1_x0=='B01)?S2:(x1_x0=='B00)?S1:S0; end
            S2 : begin OTR<=(x1_x0=='B10)?1:0; COUNT<=COUNT+OTR;
                    STAR<=(x1_x0=='B00)?S1:S0; end
        endcase
    endmodule
```

Diamo uno sguardo all'evoluzione temporale di questa rete:

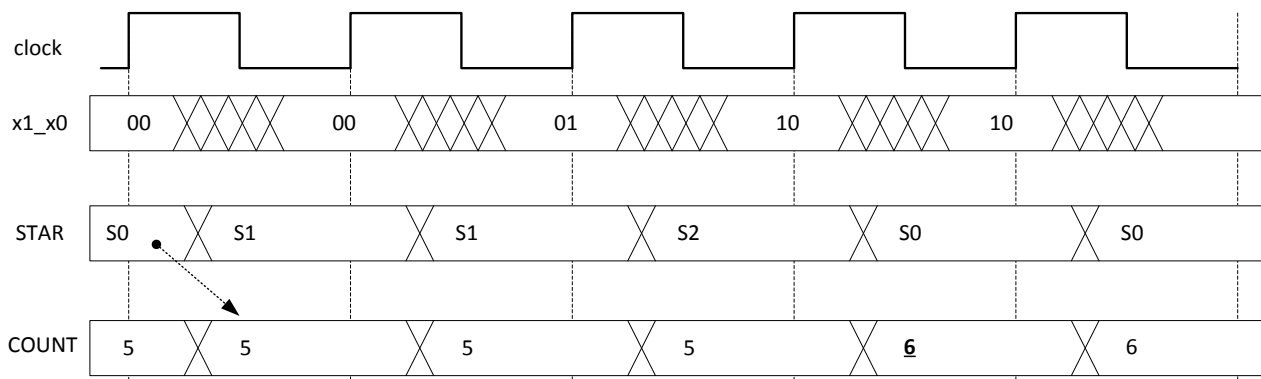


Si vede che COUNT incrementa di un clock in ritardo rispetto alla sequenza degli ingressi riconosciuta. Ciò è dovuto al fatto che il valore di incremento viene **prima** memorizzato in OTR e **poi** sommato a COUNT. Visto che non abbiamo bisogno di questo, possiamo ottimizzare la descrizione **eliminando OTR**, con il che si risparmia un registro e l'uscita si aggiorna un clock prima.

```

S0 : begin COUNT<=COUNT; STAR<=(x1_x0=='B00')?S1:S0; end
S1 : begin COUNT<=COUNT;
      STAR<=(x1_x0=='B01')?S2:(x1_x0=='B00')?S1:S0; end
S2 : begin COUNT<=(x1_x0=='B10')?COUNT+1:COUNT;
      STAR<=(x1_x0=='B00')?S1:S0; end

```



Si noti che questa rete ha soltanto **tre stati interni**. Tale compattezza è intrinsecamente legata al fatto che ho potuto avvalermi **dello stato dei registri** per **generare gli ingressi alle reti combinatorie**. Come si vede questa cosa? Dal fatto che registri **operativi** si trovano a **destra dell'operatore di assegnamento procedurale**. Nel modello di Mealy ritardato **non ci potevano stare**.

Un po' di **nomenclatura**:

- Una descrizione così fatta si dice a **livello di linguaggio di trasferimento tra registri**.
- Il **Verilog comprende**, tra mille altre cose, un linguaggio di trasferimento tra registri
- Ogni ramo del `case` si chiama **statement**, e comprende:
 - a) Zero o più **μ-istruzioni**, cioè assegnamenti a registri operativi;
 - b) Un μ-salto, cioè un assegnamento al registro **STAR**. Tale μ-salto può essere a **due vie**, come in S0, S2, a **più vie**, come in S1, o a **una via (incondizionato)**, se scrivo, e.g., `STAR<=S2`.

In generale, supponendo di avere Q registri operativi, uno statement avrà la seguente forma:

```

Sj : begin
      R0<=espressione(j,0) (var_ingresso, R0, ... RQ-1);
      [...]
      RQ-1<=espressione(j,Q-1) (var_ingresso, R0, ... RQ-1);
      STAR<=espressionej (var_ingresso, R0, ... RQ-1);
    end

```

Posso omettere di specificare il comportamento di un **registro operativo** (attenzione: **operativo**) in uno statement della descrizione. In questo caso, è come se scrivessi:

REGISTRO<=REGISTRO;

Ad esempio, potrei omettere di scrivere l'aggiornamento di COUNT in S0, S1, ed è quello che faremo normalmente nel seguito. Se, invece, ometto di specificare l'assegnamento **al registro di stato STAR**, è sottinteso che il **μ-salto è incondizionato, e porta allo statement successivo** nella descrizione. Non potrebbe essere altrimenti, perché se fosse `STAR<=STAR` si avrebbe un **deadlock**,

cioè una condizione di stallo dalla quale non si esce finché qualcuno non decide di dare un colpo di reset. **Noi non ometteremo mai l'aggiornamento di STAR**, perché farlo diminuirebbe la leggibilità ed è fonte di errori.

Per quanto riguarda i **vincoli di temporizzazione**, questa rete è soggetta alle stesse disequazioni di una rete di Mealy Ritardato. A livello di **diagrammi di temporizzazione**, lo stato di **tutti i registri** (operativi e di stato) cambia in modo **sincronizzato** all'arrivo del clock. Pertanto, quando un registro compare **a destra** di un assegnamento, ci si riferisce al **valore che aveva prima del fronte del clock**.

Attenzione: stiamo parlando di **modalità di descrizione** di una RSS complessa. È chiaro che il punto di arrivo del nostro lavoro dovrà essere la **sintesi** della medesima, cioè decidere quali “scatole” vanno messe e come vanno collegate affinché la rete abbia il comportamento specificato nella descrizione. Le modalità di **sintesi** verranno affrontate più avanti, quando avremo fatto pratica con il formalismo di descrizione.

4.1.1 Esempio: contatore di sequenze alternate 00,01,10 – 11,01,10

Variante sul tema, che complica leggermente quanto visto nell'esempio precedente. Voglio descrivere una rete che **incrementi** un contatore a 4 bit quando riconosce la **prima** delle due sequenze, poi incrementa quando vede la **seconda**, poi di nuovo la prima, e così via in modo alternato.

La rete avrà due var. di ingresso (x_1x_0) e quattro di uscita (il contenuto del registro COUNT). Per descriverne il comportamento mi serve almeno **un altro registro oltre STAR**, che chiamo COUNT e dimensiono a 4 bit, come da specifica. Analizzando le specifiche si vede subito che le due sequenze da riconoscere **differiscono soltanto per il primo passo**, e poi sono identiche. Le sequenze **dispari** devono cominciare per 00, quelle **pari** per 11. Posso quindi sfruttare questo aspetto per realizzare una rete semplice.

Visto che ogni volta che incremento COUNT cambia il tipo di sequenza da riconoscere, posso pensare di avere una **rete combinatoria** che, basandosi sul **bit meno significativo di COUNT** (che mi dice appunto se ho contato un numero pari o dispari di sequenze), e **sullo stato di ingresso alla rete**, dà in uscita 1 se quel passo è il primo passo corretto e 0 altrimenti.

COUNT[0], x_1, x_0	match
000	1
111	1
Others	0

Se ho a disposizione una rete così fatta, la descrizione in Verilog del contatore di sequenze alternate è assai semplice. Basta sostituire quello che ho scritto prima con:


```

S0: begin COUNT<=COUNT; STAR<=(match(COUNT[0],x1_x0)==1)?S1:S0; end
S1: begin COUNT<=COUNT;
      STAR<=(x1_x0=='B01')?S2:(match(COUNT[0],x1_x0)==1)?S1:S0;
      end
S2: begin COUNT<=(x1_x0=='B10')?COUNT+1:COUNT;
      STAR<=<=(match(COUNT[0],x1_x0)==1)?S1:S0; end

```

E definire da qualche parte nella descrizione la funzione **match** che abbiamo usato.

Si noti ancora che la semplicità di questa descrizione è dovuta alla possibilità di usare il valore di COUNT come ingresso alle reti combinatorie.

```

module Riconoscitore_e_Contatore(z3_z0,x1_x0,clock,reset_);
  input clock,reset_;
  input [1:0] x1_x0;
  output [3:0] z3_z0;

  reg [1:0] STAR; parameter S0='B00',S1='B01',S2='B10';
  reg [3:0] COUNT; assign z3_z0=COUNT; // Registro operativo

  always @(reset_==0) #1 begin COUNT<='B0000; STAR<=S0; end
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: begin COUNT<=COUNT;
            STAR<=(match(COUNT[0],x1_x0)==1)?S1:S0; end
      S1: begin COUNT<=COUNT;
            STAR<=(x1_x0=='B01')?S2:(match(COUNT[0],x1_x0)==1)?S1:S0; end
      S2: begin COUNT<=(x1_x0=='B10')?COUNT+1:COUNT;
            STAR<=(match(COUNT[0],x1_x0)==1)?S1:S0; end
    endcase

  function match;
    input tipo_sequenza;
    input [1:0] x1_x0;
    casex({tipo_sequenza,x1_x0})
      'B000: match=1;
      'B111: match=1;
      default: match=0;
    endcase
  endfunction
endmodule

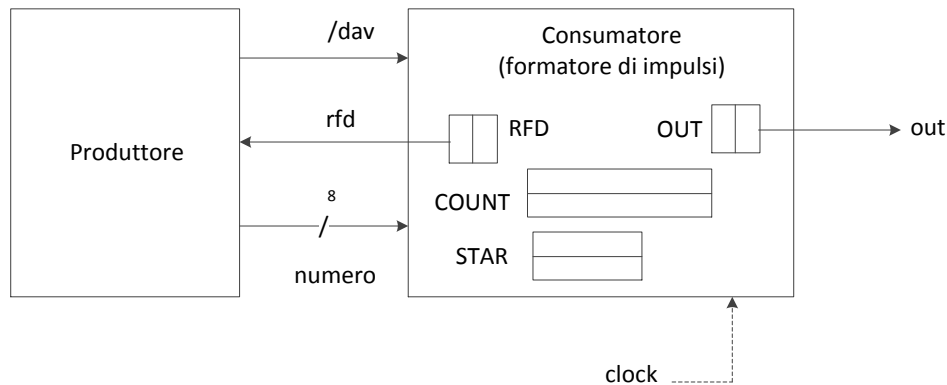
```

4.1.2 Esempio: formatore di impulsi con handshake /dav-rfd

Vediamo subito un altro esempio, un po' più complesso, che introduce i concetti fondamentali di **handshake e temporizzazione**. Supponiamo di avere due RSS **mutuamente asincrone**, che vuol dire che:

- Ognuna si evolve in modo indipendente dall'altra;
- non c'è nessuna forma di sincronizzazione comune. Ad esempio i clock delle due reti potrebbero essere di **frequenza differente**, e comunque **non saranno sincronizzati**.

Una delle due reti, che chiamiamo **produttore**, produce ogni tanto dei **numeri naturali in base 2 su 8 bit**. L'altra rete, quella che vogliamo sintetizzare, **prende in ingresso questi numeri**, e tiene una linea di uscita *out* a 1 per il numero di clock specificato nel numero. Nel caso particolare in cui il numero vale 0, allora terrà l'uscita ad 1 per **256 cicli di clock**. Pertanto la chiamiamo **consumatore**, o **formatore di impulsi**.

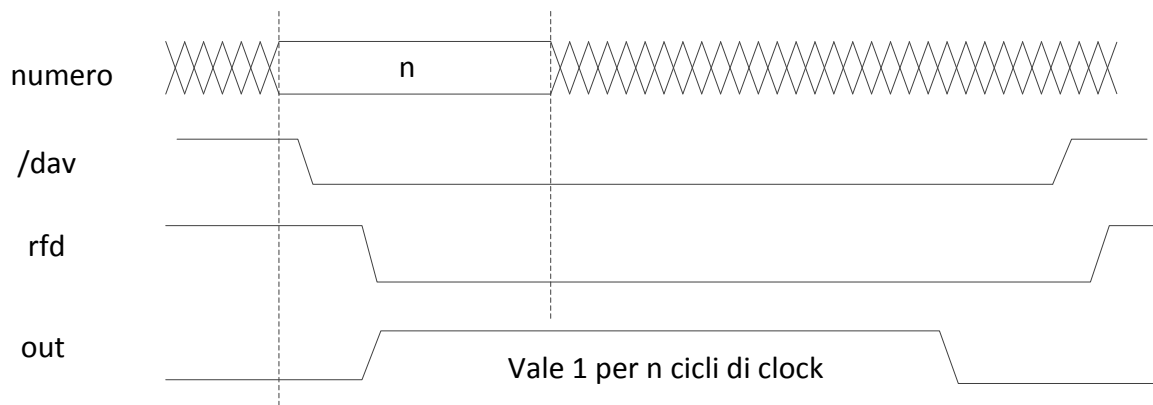


Primo problema: **come fanno le due reti a sincronizzarsi tra di loro?** Se il produttore volesse dare in ingresso due volte lo stesso numero, come fa il consumatore a distinguere che sono due numeri diversi? E, inoltre, come fa il produttore a essere sicuro che il consumatore ha “memorizzato” il numero, visto che potrebbe in teoria anche essere molto più lento? Infine, il consumatore – dopo che ha accettato un numero in ingresso – sarà occupato per un po’ di tempo a tenere alta la linea di uscita. In quel frattempo, o riusciamo a bloccare il produttore, o questo potrebbe produrre degli stati di uscita che nessuno ascolterà, ed il sistema si evolve in maniera non predicibile.

Questo problema **si presenta sempre** quando ci sono unità **mutuamente asincrone** che si devono scambiare dei dati. Quindi va trovata una maniera per risolverlo una volta per tutte. Se voglio aggiungere **sincronizzazione**, è necessario che doti entrambe le reti di **fili in più** che consentano:

- Al produttore di segnalare al consumatore che è in arrivo un nuovo dato;
- Al consumatore di segnalare al produttore la capacità o incapacità di accettare nuovi dati.

Quest’ultima interazione, giocata sui due fili appena menzionati, è appunto detta **handshake**. L’handshake si fa con due fili, detti **/dav** e **rfd** (**data valid** e **ready for data**), il primo **attivo basso**, il secondo **attivo alto**, che hanno transizioni **alternate**.



Si parte da una situazione – che possiamo supporre vera al reset: d’ora in avanti supporremo che tutti i dispositivi che fanno parte del nostro problema, sia quelli che dobbiamo sintetizzare, sia quelli con i quali ci interfacciamo, **siano connessi allo stesso circuito di reset** – in cui entrambe le linee di handshake sono a 1.

- */dav* a 1 segnala che **non ci sono dati nuovi** (è una variabile attiva-bassa), cioè che lo stato dei fili *numero* non è da considerarsi un dato valido.

- *rfd* a 1 segnala che **siamo disposti ad accettare un nuovo dato**

La prima mossa la deve fare, necessariamente, il **produttore**, il quale, nell’ordine

- a) prepara un nuovo dato, e lo mette sull’uscita *numero*
- b) abbassa */dav*

quindi, il fronte di discesa di */dav* **convalida**, per il consumatore, il dato in ingresso.

Il consumatore deve, nell’ordine:

- c) prelevare il nuovo dato (cioè memorizzarlo da qualche parte)
- d) abbassare *rfd*, a segnalare che non è più disponibile a prelevare nuovi dati.

Il fronte di discesa di *rfd* è, **per il produttore, il segnale che non è più necessario mantenere il dato in uscita**. Quindi (errore tipico nei compiti) è **sbagliato memorizzare un dato di ingresso in un registro dopo che si è tirato giù *rfd***, perché si deve fare conto che il produttore lo può togliere **subito dopo** aver visto scendere *rfd*.

A questo punto, **quando pare a lui**, il produttore riporta su */dav*. Può farlo in un istante qualunque successivo alla transizione 1-0 di *rfd*.

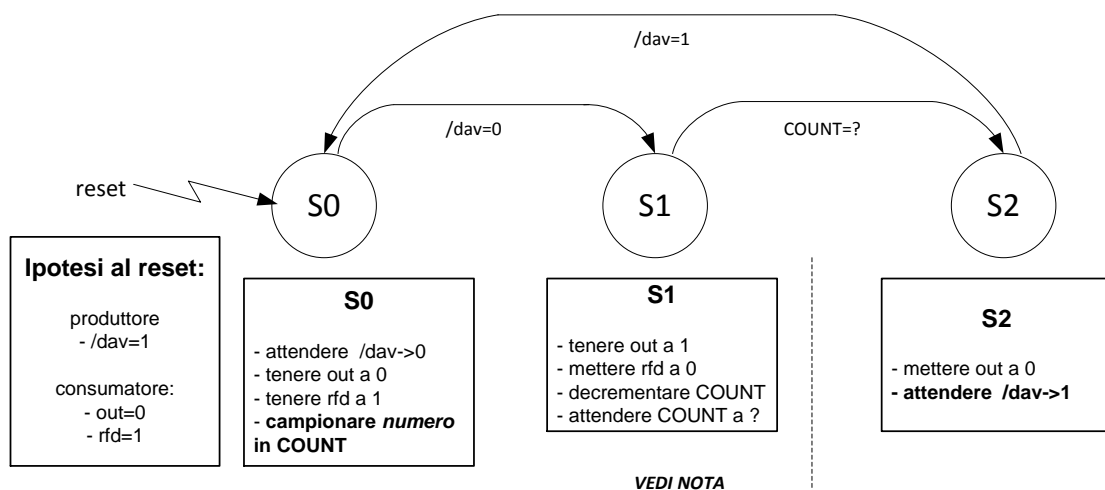
Quando il consumatore è nuovamente pronto ad accettare un dato in ingresso, **tira su *rfd***. Non può però farlo **prima di essersi assicurato che */dav* sia tornato a 1**. Se infatti lo facesse, il produttore **potrebbe non accorgersi neanche che *rfd* è andato a 0**, con il che il sistema andrebbe **in dead-lock (stallo)**, e dovrebbe essere resettato.

Come si fa a fare in modo che l’uscita stia ad 1 per un dato numero di cicli di clock? Si prende

- 1) un registro a 1 bit (OUT) che deve sostenere l'uscita, e gli assegniamo il valore 1 dopo aver letto *numero*.
- 2) un **registro contatore**, COUNT che inizializziamo ad un certo valore (**parente di *numero***), e facciamo contare **down ad ogni clock**. Quando arriva a 0 (o ad 1, vediamo), vorrà dire che sono passati un certo numero di cicli di clock. Allora metteremo nuovamente a 0 il valore di OUT.

Ciò detto, facciamo il punto di cosa ci serve nella nostra rete:

- Ci vuole **un registro per ogni variabile di uscita**: quindi, **OUT a 1 bit, RFD a 1 bit**.
- Ci vuole **un registro COUNT**, per tenere in mente quanti cicli di clock mancano a rimettere a 0 la variabile di uscita *out*. **In prima battuta posso supporre che COUNT sia a 8 bit**, perché memorizzerò *numero* e conterò all'indietro. Il dimensionamento dei registri si fa **prima a occhio**, poi ci si ritorna sopra mentre si scrive la descrizione.
- Ci vuole **un registro di stato STAR**, che tiene traccia dell'evoluzione della rete. A quanti bit? **Dipende da come scrivo la descrizione.**



Nota: quando sono in S1, e ci sono stato per tutto il tempo necessario, **posso tornare in S0?** Certo che **NO**, perché in S0 metto RFD a 1, e per fare questo devo essere **sicuro che /dav=1**. Ci vuole **per forza** un altro stato.

Allora, dopo aver visto questo diagramma, posso concludere che **STAR è un registro a 2 bit**.

```
module Formatore_di_Impulsi(dav_, rfd, numero, out, clock, reset_);
    input clock, reset_;
    input dav_;
    output rfd;
    input [7:0] numero;
    output out;
    reg RFD; assign rfd=RFD;
    reg OUT; assign out=OUT;
    reg [7:0] COUNT;
    reg [1:0] STAR; parameter S0='B00, S1='B01, S2='B10;
```

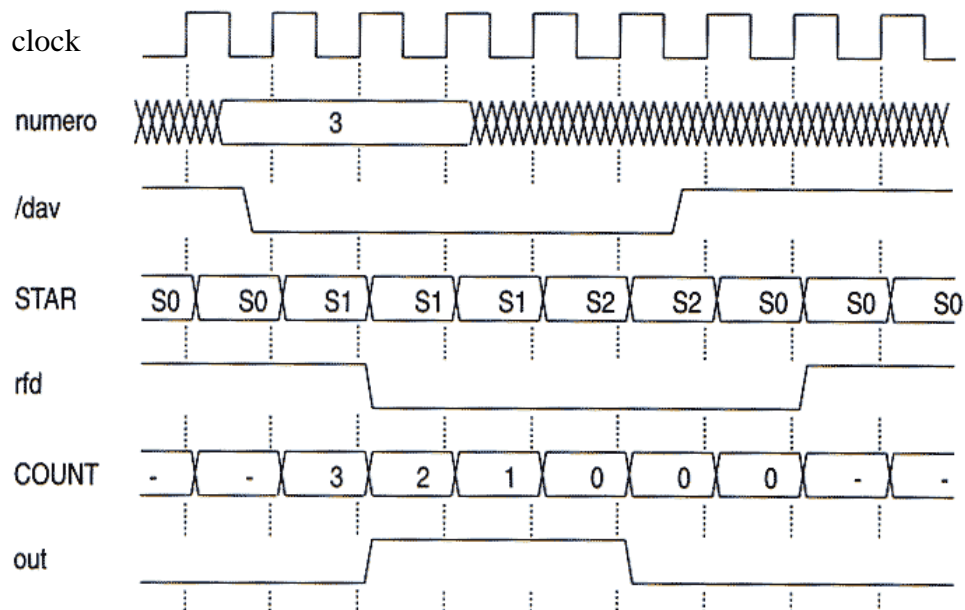
```

always @(reset_==0) #1 begin RFD<=1; OUT<=0; STAR<=S0; end
always @(posedge clock) if (reset_==1) #3
  case(STAR)
    S0: begin RFD<=1; OUT<=0; COUNT<=numero;
        STAR<=(dav_==1)?S0:S1; end
    S1: begin RFD<=0; OUT<=1; COUNT<=COUNT-1;
        STAR<=(COUNT==1)?S2:S1; end
    S2: begin RFD<=0; OUT<=0; STAR<=(dav_==1)?S0:S2; end
  endcase
endmodule

```

Come si fa a vedere **cosa devo scrivere come condizione di transizione da S1 a S2?** Si fa con un diagramma di temporizzazione, ed in nessun altro modo.

- Disegnare il diagramma di temporizzazione con **numero=1** e **COUNT==0** nel test, ed osservare che OUT sta su per 2 clock invece che per 1.
- Disegnare il diagramma di temporizzazione con **numero=1** e **COUNT==1** nel test, ed osservare che OUT sta su per 1 clock, come deve essere.



Quando si hanno **cicli di decremento e test** (come in questo caso) la regola è semplice:

Se in uno stato S_{init} inizializzo un registro a k ed in S_{test} lo decremento e per uscire testo se il valore è pari a j ($\leq k$), il numero di cicli nello stato S_{test} è $k - j + 1$.

```

Sinit: begin ... COUNT<=k; STAR<=Stest; end
Stest: begin ... COUNT<=COUNT-1; STAR<=(COUNT==j)?Spoi:Stest; end
Spoi : begin ... end

```

Nel fare il diagramma di temporizzazione, si prende atto che

- a) **il valore dei registri cambia dopo il fronte del clock**
- b) **tutti i registri memorizzano il loro (nuovo) ingresso contemporaneamente.**

E quindi il fatto che io abbia scritto il decremento di COUNT nello stesso statement S1 in cui testo COUNT per il μ -salto non ci deve indurre in errore. Il decremento avverrà **dopo il clock**, e quindi il test per il μ -salto verrà effettuato **sul vecchio valore di COUNT**, non sul nuovo.

A consuntivo, si vede subito che, visto che da S1 si deve sempre passare, ed in S1 abbiamo $OUT \leq 1$, **in ogni caso OUT starà ad 1 almeno per un clock**. Pertanto, se in COUNT metto 0, il test in S1 fallisce, quindi si **decrementa COUNT** (modulo 2^8), e quindi si fanno altri 255 cicli in S1, coerentemente con le specifiche ricevute.

Alcune note sulla descrizione

Sappiamo che si può omettere di specificare il comportamento di un **registro operativo** (attenzione: **operativo**) in uno stato della descrizione. In questo caso, è come se scrivessi

```
REGISTRO<=REGISTRO;
```

ogni volta che non scrivo niente. Ad esempio:

- In S2, è come se avessi scritto $COUNT \leq COUNT$; in realtà, in S2 non ci interessa cosa faccia COUNT, in quanto nessuno lo sta usando, e prima che qualcuno lo riutilizzi (in S1) ci verrà scritto qualcosa (in S0).
- A ben guardare, $RFD \leq 0$ in S2 non è necessario, in quanto in S2 arrivo soltanto da S1, ed in S1 sono sicuro che RFD è a 0.
- Lo stesso $OUT \leq 0$ in S0, in quanto in S0 arrivo soltanto da S2 e dal reset, ed in entrambi i casi sono sicuro che OUT è a 0.

In ogni caso, **mantenere l'assegnamento ai registri operativi aumenta la leggibilità.**

Siccome nessuno mi obbliga a mettere $RFD \leq 0$ subito dopo che $/dav$ è andato a 0, potrei tenere $RFD \leq 1$ in S1, e metterlo a 0 direttamente in S2. In questo modo pospongo la gestione dell'handshake a quando ho terminato la temporizzazione. Ciò significa che, probabilmente, si passerà più tempo ad attendere in S2, perché il produttore dovrà reagire alla transizione di *rfd* e rimettere $/dav$ a 1, cosa che avrebbe potuto fare direttamente mentre il consumatore ciclava in S1. Visto che **nessuno mi obbliga a fare le corse** (non c'è nessuna specifica al riguardo) posso fare anche così.

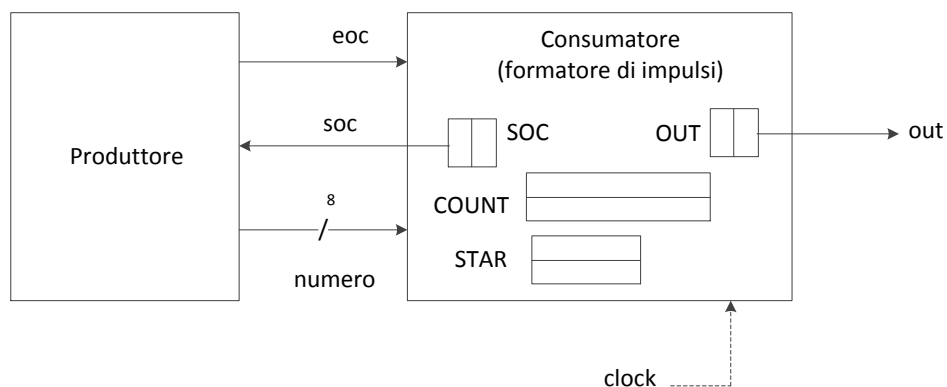
Domanda: se la specifica fosse stata: “**se numero=0, OUT non deve andare a 1**”, cosa avrei dovuto scrivere?

```
S1: begin RFD<=1; OUT<=(numero==0)?0:1; COUNT<=COUNT-1;  
      STAR<=((COUNT==1)|(numero==0))?S2:S1; end  
S2: begin RFD<=0; OUT=0;...
```

In questo esempio è **essenziale** che RFD venga lasciato ad 1 in S1, altrimenti non potrei usare il valore di *numero* in S1: dopo il primo passaggio in S1 il produttore, vedendo rfd a zero, potrebbe averlo tolto, ma in S1 verrebbe usato nuovamente.

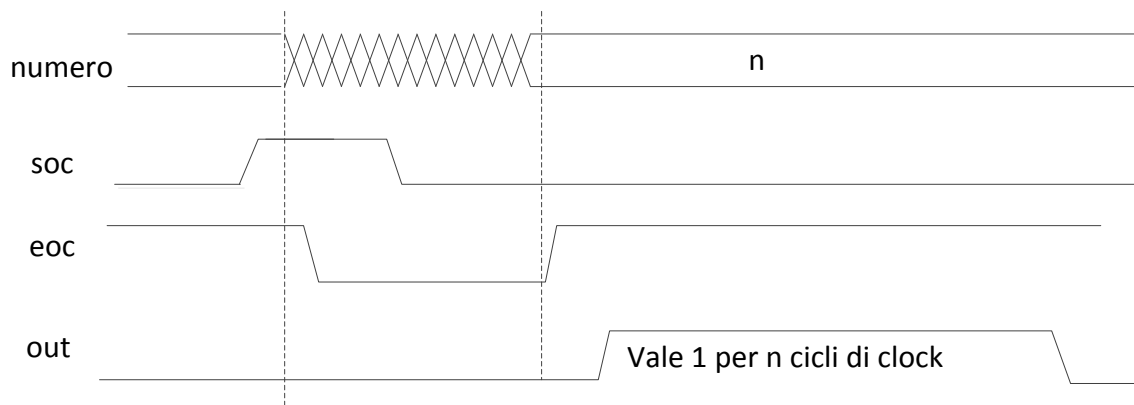
4.1.3 Esempio: formatore di impulsi con handshake soc/eoc

Possiamo modificare le specifiche dell’handshake da produttore a consumatore. Un altro modo per sincronizzare due reti per il passaggio di dati è l’**handshake soc-eoc**. SOC sta per Start Of Computation, EOC sta per End of Computation. Il **consumatore** imposta **soc** per chiedere un nuovo dato, e il **produttore** risponde con il dato quando ha finito la computazione. Questo handshake è tipico dei **convertitori** analogico/digitali.



In questo caso:

- Il valore di riposo è **soc=0, eoc=1**, ed il dato che questo handshake protegge è **valido**.
- La prima mossa è fatta dal consumatore, che porta **soc ad 1**, per segnalare la richiesta di un nuovo dato
- Il produttore risponde portando **eoc a zero**, che segnala l'accettazione del comando di start. Quando **eoc =0, il dato protetto dall'handshake può non essere significativo**, e quindi non va guardato.
- Il consumatore riporta **soc a zero**, segnalando che ha capito che il produttore sta preparando il dato.
- Il produttore prepara il nuovo dato, lo mette sull'uscita, e **dopo (mai prima) riporta eoc a 1**.



La descrizione va cambiata di conseguenza:

```

module Formatore_di_Impulsi_Soc_Eoc(soc,eoc,numero,out,clock,reset_);
  input clock,reset_;
  output soc;
  input eoc;
  input [7:0] numero;
  output out;
  reg SOC; assign soc=SOC;
  reg OUT; assign out=OUT;
  reg [7:0] COUNT;
  reg [1:0] STAR; parameter S0='B00,S1='B01,S2='B10;

  always @(reset_==0) #1 begin SOC<=0; OUT<=0; STAR<=S0; end
  always @(posedge clock) if (reset_==1) #3
    case(STAR)
      S0: begin OUT<=0; SOC<=1; STAR<=(eoc==1)?S0:S1; end
      S1: begin SOC<=0; COUNT<=numero; STAR<=(eoc==0)?S1:S2; end
      S2: begin OUT<=1; COUNT<=COUNT-1; STAR<=(COUNT==1)?S0:S2; end
    endcase
endmodule

```

4.2 Sintesi di RSS complesse – scomposizione in “parte operativa” e “parte controllo”

Abbiamo visto esempi di **descrizione** scritta usando il **linguaggio di trasferimento tra registri del Verilog** (ne vedremo molti altri nel corso delle lezioni). La descrizione può essere **simulata** (con un diagramma di temporizzazione) per verificarne il comportamento. Devo adesso preoccuparmi di come si fa a **realizzare** una rete così descritta, in termini di **porte AND, OR, NOT** (o comunque di circuiti logici più complessi, fatti in termini delle suddette porte, la cui composizione interna abbiamo già visto a lezione). Ciò corrisponde a fare la **sintesi** di queste reti.

Esiste un procedimento **semiautomatico** che genera una **sintesi a partire dalla descrizione**, chiamato **scomposizione in “parte operativa” e “parte controllo”**. Tale procedimento **non ha come scopo arrivare ad una sintesi ottima**, quale che sia il criterio per l’ottimalità (e.g., costo minimo).

La sintesi ottima **non ci interessa**, ci interessa **arrivare ad una sintesi in modo semi-automatico**

partendo dalla descrizione. Riprendiamo l'esempio del **riconoscitore e contatore di due sequenze**, il cui blocco *always* è richiamato di seguito:

```
always @(reset_==0) #1 begin COUNT<='B0000; STAR<=S0; end
always @(posedge clock) if (reset_==1) #3
case(STAR)
  S0: begin COUNT<=COUNT;
      STAR<=(match(COUNT[0],x1_x0)==1)?S1:S0; end
  S1: begin COUNT<=COUNT;
      STAR<=(x1_x0=='B01)?S2:(match(COUNT[0],x1_x0)==1)?S1:S0; end
  S2: begin COUNT<=(x1_x0=='B10)?COUNT+1:COUNT;
      STAR<=(match(COUNT[0],x1_x0)==1)?S1:S0; end
endcase
```

Andiamo per passi.

Punto 1

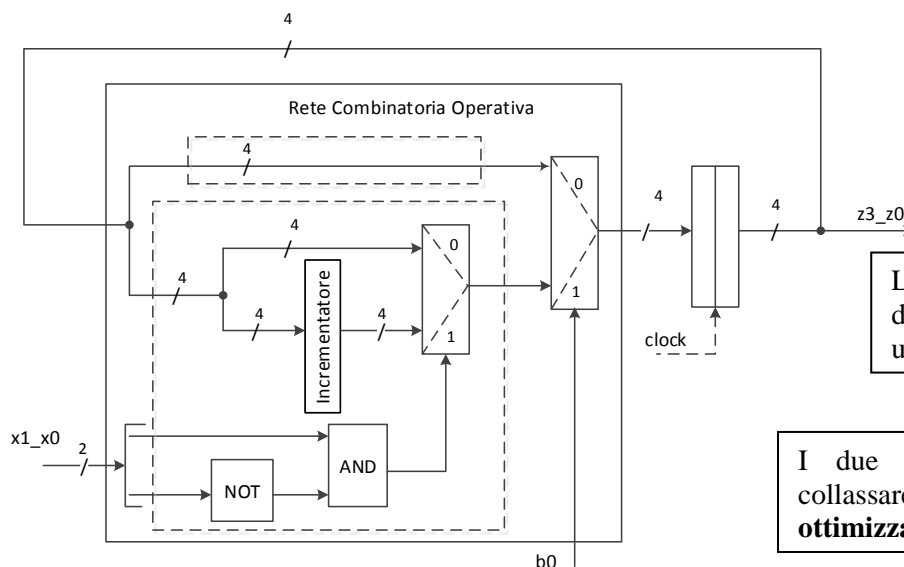
Il trucco da cui discende tutto è

Guardare ai registri operativi come a registri multifunzionali

Si isola ciascun registro operativo (in questo caso ne abbiamo uno solo, **COUNT**), e si individuano, nel blocco *always*, le **μ-operazioni diverse** per il registro considerato:

```
S0, S1: COUNT<=COUNT;
S2:      COUNT<=(x1_x0=='B10)?COUNT+1:COUNT;
```

Data questa situazione, posso vedere **COUNT** come **registro multifunzionale a due vie**, ciascuna corrispondente ad una μ-operazione elencata. La prima è una **conservazione**, la seconda è una μ-operazione più complessa, che a sua volta può essere implementata con un **multiplexer a due vie** comandato da una variabile che è funzione degli ingressi.



Lo stesso procedimento che sto descrivendo qui si può scrivere usando la sintassi del Verilog.

I due multiplexer si possono collassare, ma **non mi interessa ottimizzare**.

Resta da capire come si genera la variabile che guida il multiplexer a due vie del registro multifunzionale COUNT. Tale variabile prende il nome di **variabile di comando**, e può essere prodotta da una rete combinatoria che abbia come ingresso lo stato interno marcato come segue:

```
assign b0=(STAR==S2)?1:0;
```

La rete combinatoria che sta davanti al registro operativo viene detta **rete combinatoria operativa**. Nel caso generale, ce ne saranno tante quante sono i registri operativi. La rete combinatoria operativa ha in ingresso, in questo caso:

- La (le) **variabili di comando** (vediamo fra un attimo chi le produce).
- **Lo stato di uscita dei registri operativi**

E produce in uscita **lo stato di ingresso dei registri operativi**. In un caso più generale, potrà avere in ingresso anche *le variabili di ingresso della rete*. Dipende ovviamente dalle μ -operazioni specificate in ogni statement.

Punto 2

Dopo aver fatto quanto appena detto **per tutti i registri operativi**, passiamo a considerare il registro di stato **STAR**. Per questo registro si considerano le **condizioni indipendenti**, cioè le scelte (indipendenti) che guidano i μ -salti. Al solito, si guarda il blocco *always*, e si isolano. Sono **due**:

1) (match(COUNT[0],x1_x0)==1), e 2) (x1_x0=='B01)

Assegno a ciascuna di queste una **variabile di condizionamento**, cioè una variabile che vale **uno se la condizione è vera, e zero se la condizione è falsa**. Nel nostro caso:

```
assign c1=(match(COUNT[0],x1_x0)==1)?1:0,
      c0=(x1_x0=='B01)?1:0
```

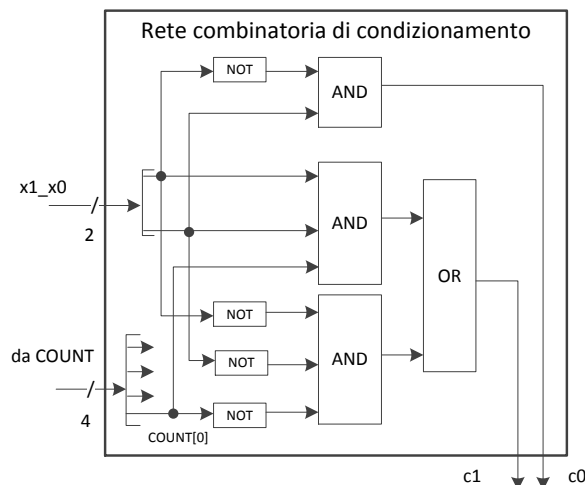
La rete che produce queste due variabili si chiama **rete combinatoria di condizionamento**. Tale rete ha in ingresso **le variabili di ingresso**, e in questo caso anche **lo stato dei registri operativi**.

Dalla tabellina vista prima, si

ricava immediatamente

$$c_1 = COUNT[0] \cdot x_1 \cdot x_0 + \overline{COUNT[0]} \cdot \overline{x_1} \cdot \overline{x_0}$$

$$c_0 = x_1 \cdot x_0$$



Riscriviamo la descrizione della rete avendo introdotto queste nuove variabili.

```
module Riconoscitore_e_Contatore(z3_z0,x1_x0,clock,reset_);
  input clock,reset_;
  input [1:0] x1_x0;
  output [3:0] z3_z0;
  reg [1:0] STAR; // Registro di stato
  parameter S0='B00,S1='B01,S2='B10;
  reg [3:0] COUNT; // Registro operativo
  assign z3_z0=COUNT;

  // Introduzione delle variabili di comando e di condizionamento
  wire b0,c1,c0;

  // Generazione delle variabili di comando
  assign b0=(STAR==S2)?1:0;

  // Generazione delle variabili di condizionamento
  wire x1,x0; assign x1=x1_x0[1]; assign x0=x1_x0[0];
  assign c1 = (~COUNT[0] & ~x1 & ~x0) | (COUNT[0] & x1 & x0);
  assign c0 = ~x1 & x0;

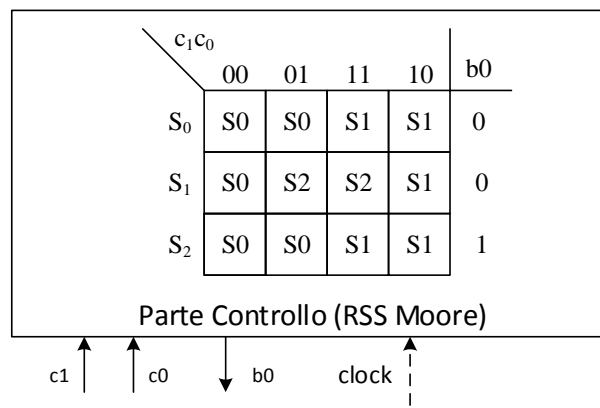
  // Blocco always modificato per il registro operativo COUNT
  always @(reset_==0) #1 COUNT<='B0000;
  always @(posedge clock) if (reset_==1) #3
    casex(b0)
      0: COUNT<=COUNT;
      1: COUNT<=(x1_x0=='B10)?COUNT+1:COUNT;
    endcase

  // Blocco always modificato per il registro di stato STAR
  always @(reset_==0) #1 STAR<=S0;
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: STAR<=(c1==1)?S1:S0;
      S1: STAR<=(c0==1)?S2:(c1==1)?S1:S0;
      S2: STAR<=(c1==1)?S1:S0;
    endcase
endmodule
```

La rete descritta in questo modo è, sostanzialmente, **identica alla precedente**. È però facile osservare che in questa descrizione posso **isolare un blocco** (quello riquadrato) che ha:

- il registro STAR
- le variabili di condizionamento come **ingressi**
- le variabili di comando (una, in questo caso) come **uscite**

e **nessun contatto con il mondo esterno** (inteso come ingressi e uscite della rete). Tale rete è **una RSS di Moore**, e prende il nome di **parte controllo**. La parte controllo non ha direttamente a che fare con la produzione delle uscite, ma solo con l'evoluzione dello stato interno.



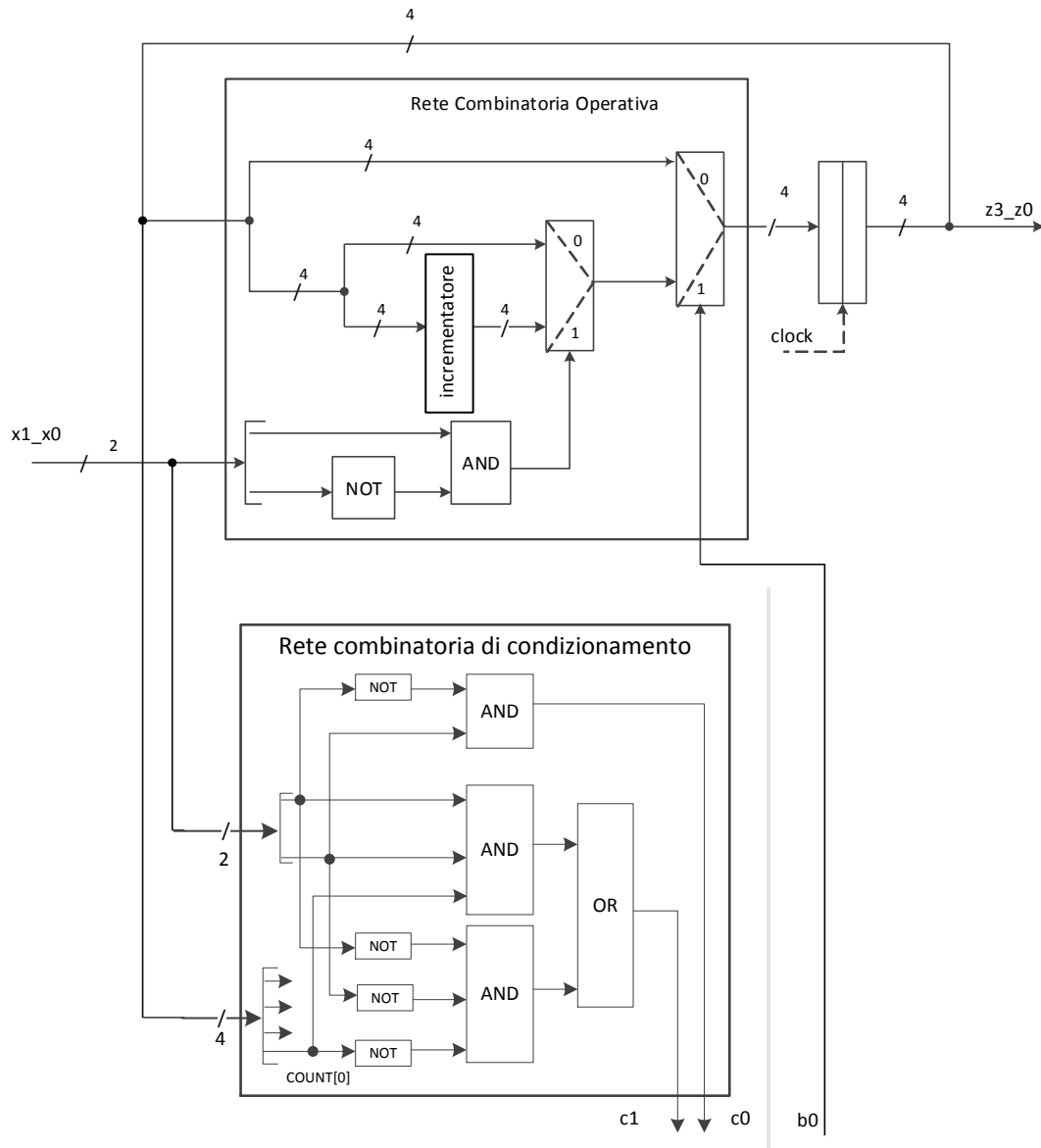
Tutto quello che rimane, comprensivo di **reti combinatorie operative**, **reti combinatorie di condizionamento**, **registri operativi**, può essere visto, a sua volta, come una RSS a sé stante, detta **parte operativa**. La parte operativa è quella che produce le uscite. Tale rete ha, come **ingressi**:

- 1) le variabili di comando (una, in questo caso)
- 2) **le variabili di ingresso alla rete originale**

e come **uscite**:

- 1) le variabili di condizionamento
- 2) **le variabili di uscita della rete originale**

ed è, in questo caso, **una RSS di Mealy**, in quanto **alcune uscite** (in particolare, le variabili di condizionamento) sono funzione combinatoria degli ingressi.



Quindi abbiamo **due RSS interconnesse**, però non abbiamo **mai un anello di reti combinatorie**, perché una delle due (la parte controllo) è di Moore.

```

module Riconoscitore_e_Contatore(z3_z0,x1_x0,clock,reset_);
    input clock,reset_;
    input [1:0] x1_x0;
    output [3:0] z3_z0;
    wire c1,c0,b0;
    Parte_Operativa PO(z3_z0,x1_x0,c1,c0,b0,clock,reset_);
    Parte_Controllo PC(b0,c1,c0,clock,reset_);
endmodule

```

```
//-----
module Parte_Operativa(z3_z0,x1_x0,c1,c0,b0,clock,reset_);
  input clock,reset_;
  input [1:0] x1_x0;
  output [3:0] z3_z0;
  input b0;
  output c1,c0;
  reg [3:0] COUNT; assign z3_z0=COUNT;
  wire x1,x0; assign x1=x1_x0[1]; assign x0=x1_x0[0];
  assign c1 = (~COUNT[0] & ~x1 & ~x0) | (COUNT[0] & x1 & x0);
  assign c0= ~x1 & x0;
  always @(reset_==0) #1 COUNT<='B0000;
  always @(posedge clock) if (reset_==1) #3
    casex(b0)
      0: COUNT<=COUNT;
      1: COUNT<=(x1_x0=='B10)?COUNT+1:COUNT;
    endcase
endmodule
//-----
module Parte_Controllo(b0,c1,c0,clock,reset_);
  input clock,reset_;
  input c1,c0;
  output b0;
  reg [1:0] STAR; parameter S0='B00,S1='B01,S2='B10;
  assign b0=(STAR==S2)?1:0;
  always @(reset_==0) #1 STAR<=S0;
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: STAR<=(c1==1)?S1:S0;
      S1: STAR<=(c0==1)?S2:(c1==1)?S1:S0;
      S2: STAR<=(c1==1)?S1:S0;
    endcase
endmodule
```

La scomposizione della rete descritta in **parte operativa e parte controllo**, con:

- 1) specifica (al livello di **descrizione di ciascun registro multifunzionale**) di ciascuna rete combinatoria operativa. Qui basta fermarsi quando si vedono **reti standard**.
- 2) specifica (al livello di **espressioni di algebra di Boole**) della rete combinatoria di condizionamento. Qui è necessario essere **precisi fino al bit**.
- 3) sintesi **della parte controllo** (scegliendo uno tra i possibili formalismi che conoscete o che vedrete più avanti)

costituisce il punto di arrivo della sintesi richiesta.

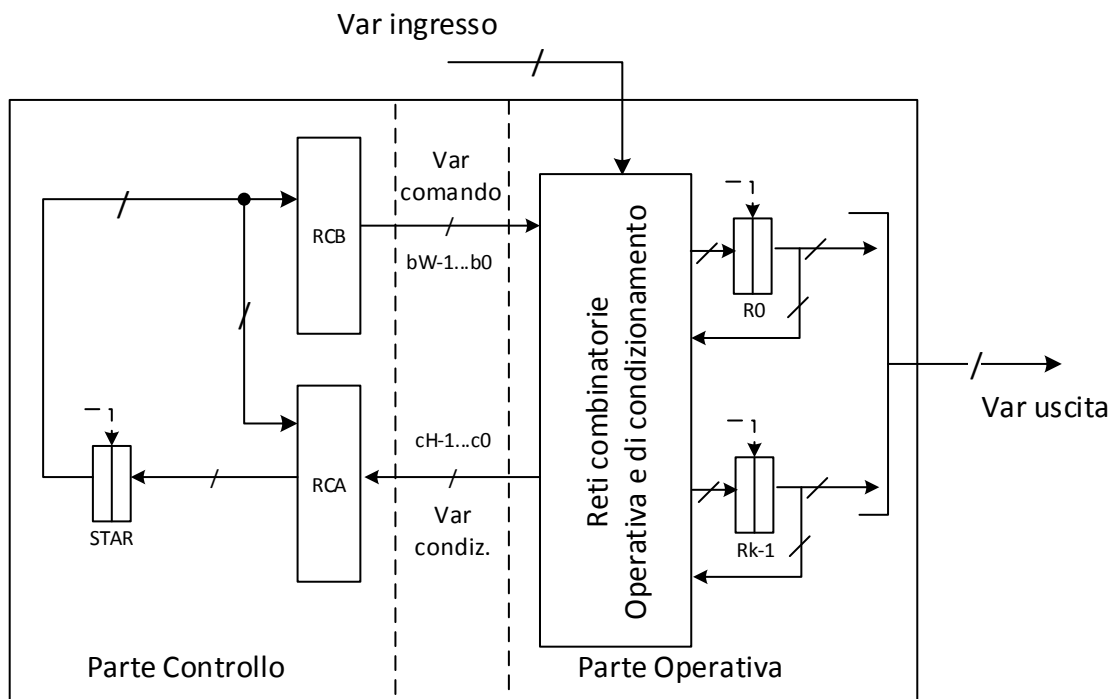
Il motivo per cui abbiamo fatto questa scomposizione è che:

- la si può fare **in modo automatico** a partire dalla descrizione in Verilog
- la sintesi della parte operativa è euristica, ma **semplice**, in quanto consta di
 - a) registri multifunzionali
 - b) rete combinatoria di condizionamento, normalmente semplice

- la sintesi della parte controllo è affrontabile tramite tecniche **standard**, e risulta sempre piuttosto semplice (**farla per esercizio**).

Per l'esempio considerato, la parte controllo è *particolarmente* semplice (perché ha tre stati interni), e può essere **sintetizzata** partendo dalla tabella di flusso, come sappiamo fare. Vedremo più avanti **tecniche standard di sintesi per la parte controllo** valide anche per casi più complessi.

In linea generale, una rete con Q registri operativi si può **sempre scomporre come visto finora**. Lo schema generale è questo, e la procedura per arrivare in fondo è soltanto più lunga (e noiosa).



4.2.1 Esempio di sintesi: formatore di impulsi con handshake /dav-rfd

Richiamiamo la descrizione per completezza:

```
module Formatore_di_Impulsi(dav_, rfd, numero, out, clock, reset_);
  input clock, reset_;
  input dav_;
  output rfd;
  input [7:0] numero;
  output out;
  reg RFD; assign rfd=RFD;
  reg OUT; assign out=OUT;
  reg [7:0] COUNT;
  reg [1:0] STAR; parameter S0='B00, S1='B01, S2='B10;

  always @(reset_==0) #1 begin RFD<=1; OUT<=0; STAR<=S0; end
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: begin RFD<=1; OUT<=0; COUNT<=numero;
               STAR<=(dav_==1)?S0:S1; end
      S1: begin RFD<=0; OUT<=1; COUNT<=COUNT-1;
               STAR<=(COUNT==1)?S2:S1; end
      S2: begin RFD<=0; OUT<=0; STAR<=(dav_==1)?S0:S2; end
    endcase
endmodule
```

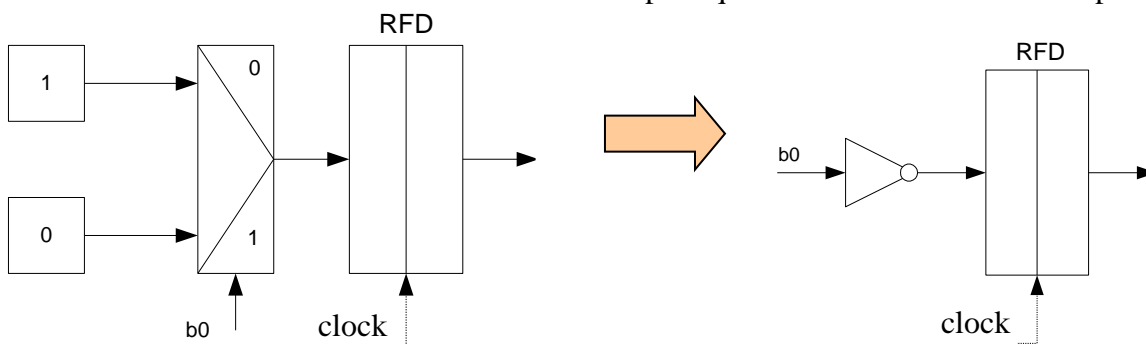
Vediamo adesso la sintesi con scomposizione in PO/PC. In questo caso abbiamo **tre registri operativi**, invece che uno: RFD, COUNT, OUT. Li guardiamo come **registri multifunzionali**, e consideriamo il numero di diverse *funzioni*, cioè μ -operazioni indipendenti che questi devono supportare. I passaggi, invece di farli in **Verilog** (troppo lungo), si fanno in modo informale (poi si riscrive tutto in Verilog in fondo).

RFD:

S0: RFD<=1
S1, S2: RFD<=0

In teoria, due μ -operazioni: registro a 2 vie, con **una** var. di comando b_0 . Tale variabile di comando dovrà avere **due valori diversi**, uno in S0 (0), ed uno in S1/S2 (1) (riempire la tabella).

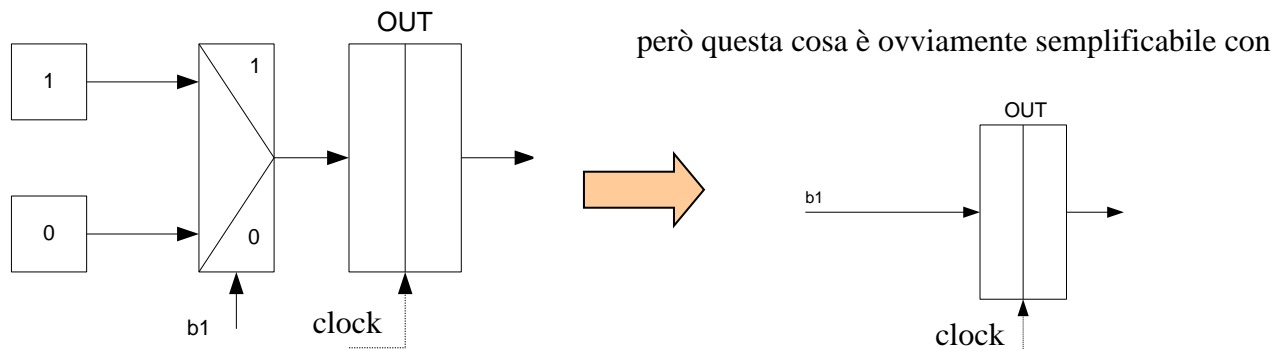
però questa cosa è ovviamente semplificabile con



OUT:

S0, S2: OUT<=0
S1: OUT<=1

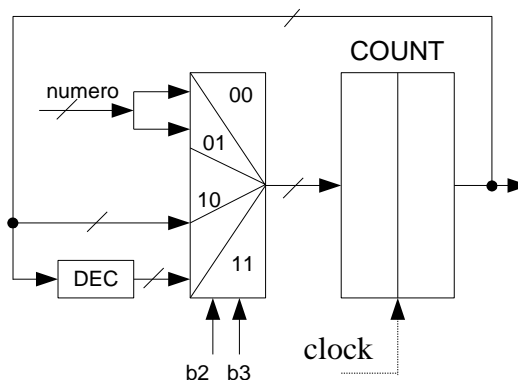
In teoria, due μ -operazioni: registro a 2 vie, con **una** var. di comando b_1 . Tale variabile di comando dovrà avere **due valori diversi**, uno in S0/S2 (0), ed uno in S1 (1) (riempire la tabella).



COUNT:

S0: COUNT<=numero
S1: COUNT<=COUNT-1
S2: COUNT<=COUNT

Tre μ -operazioni: registro a 3 vie, con **due** var. di comando b_2, b_3 . Le variabili di comando dovranno avere **tre valori diversi**, uno in S0, uno in S1, uno in S2. (riempire la tabella).



In questo caso, posso permettermi di lasciare qualcosa **non specificato**, perché sto usando 3 combinazioni di 4 possibili. Se scelgo come in tabella, vedo subito che $b_3=b_0$, e che b_2 può essere resa uguale a b_1 specificando opportunamente il valore.

STAR	b_0	b_1	b_2	b_3
S0	0	0	-	0
S1	1	1	1	1
S2	1	0	0	1

Nota: Quello di osservare che due variabili di comando sono (o possono essere rese) **uguali** è **l'unico tipo di ottimizzazione che ci concederemo nella sintesi**. Ad esempio, qualcuno potrebbe osservare che, visto che la μ -operazione in S2 per COUNT non è significativa, tanto valeva usare un registro a 2 vie, in cui ho solo **decremento e caricamento**: in S2 COUNT farà una delle due cose, non importa quale. Scrivere questa cosa nella sintesi è **errore (grave)**. La sintesi **deve essere coerente 1:1 con la descrizione**. Se non coincide con la descrizione, è sbagliata.

Né ha minimamente senso **rimettere le mani nella descrizione col proposito di "migliorare" la sintesi**. Sarebbe come riscrivere un programma in C++, dopo averlo visto assemblato, per rispar-

miare qualche istruzione in linguaggio macchina. La descrizione deve essere **leggibile**, la sintesi deve essere **identica alla descrizione**.

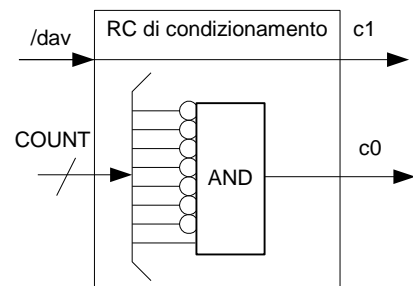
Passiamo adesso al registro **STAR**, e ricaviamo le **variabili di condizionamento**.

STAR:

```
S0: STAR<=(dav_==1)?S0:S1;  
S1: STAR<=(COUNT==1)?S2:S1;  
S2: STAR<=(dav_==1)?S0:S2;
```

Due condizioni indipendenti: sono necessarie **due** var. di condizionamento c_1, c_0 .

- La prima, c_1 , sarà 1 quando $/dav$ vale 1, e 0 altrimenti. Quindi, la rete che la produce ha in ingresso $/dav$ ed è un **corto circuito**.
- La seconda, c_0 , sarà 1 quando COUNT vale 1, e 0 altrimenti. Quindi, la rete che la produce ha in ingresso l'uscita di COUNT (su 8 bit) ed è un AND a 8 ingressi con 7 invertitori sui bit più alti.



E' richiesto sempre che disegniate la RC di condizionamento. Non disegnarla è considerato **errore**, perché vi dà la falsa sicurezza di poter scrivere, nella descrizione, delle condizioni di salto che non siete in grado di realizzare.

Detto questo, si può scrivere senza problemi tutta la sintesi in Verilog.

```
module Formatore_di_Impulsi(dav_, rfd, numero, out, clock, reset_);  
    input clock, reset_;  
    input dav_;  
    output rfd;  
    input [7:0] numero;  
    output out;  
    wire c1, c0, b1, b0;  
    Parte_Operativa PO(out, rfd, dav_, numero, c1, c0, b1, b0, clock, reset_);  
    Parte_Controllo PC(b1, b0, c1, c0, clock, reset_);  
endmodule
```

```

module Parte_Operativa(dav_,rfd,numero,out,c1,c0,b1,b0,clock,reset_);
  input clock,reset_;
  input dav_;
  output rfd;
  input [7:0] numero;
  output out;
  input b1,b0;
  output c1,c0;

  reg RFD; assign rfd=RFD;
  reg OUT; assign out=OUT;
  reg [7:0] COUNT;

  assign c1=(dav_==1)?1:0;
  assign c0=(COUNT==1)?1:0;

  //Registro RFD
  always @(reset_==0) #1 RFD<=1;
  always @(posedge clock) if (reset_==1) #3 RFD<=~b0;

  //Registro OUT
  always @(reset_==0) #1 OUT<=0;
  always @(posedge clock) if (reset_==1) #3 OUT<=b1;

  //Registro COUNT
  always @(posedge clock) #3
    case ({b1,b0})
      'B?0: COUNT<=numero;
      'B11: COUNT<=COUNT-1;
      'B01: COUNT<=COUNT;
    endcase
endmodule

module Parte_Controllo(b1,b0,c1,c0,clock,reset_);
  input clock,reset_;
  input c1,c0;
  output b1,b0;
  reg [1:0] STAR; parameter S0='B00,S1='B01,S2='B10;
  assign {b1,b0}= (STAR==S0)?'B00:
                (STAR==S1)?'B11:
                (STAR==S2)?'B01:
                /*default*/'BXX;
  always @(reset_==0) #1 STAR<=S0;
  always @(posedge clock) if (reset_==1) #3
    case (STAR)
      S0: STAR<=(c1==1)?S0:S1;
      S1: STAR<=(c0==1)?S2:S1;
      S2: STAR<=(c1==1)?S0:S2;
    endcase
endmodule

```

Come nel caso precedente, la parte controllo è molto semplice, ed essendo una rete di Moore si può sintetizzare in maniera ottimizzata con i metodi che conosciamo (scelta di un meccanismo di marcatura e sintesi di RCA, RCB).

4.3 Tecniche euristiche di sintesi della parte controllo

Per rendere possibile trattare casi di parti controllo complesse (cioè, con più di 4 stati interni e 2 variabili di condizionamento) è necessario introdurre una tecnica di sintesi più generale che ci porti alla **sintesi della parte controllo in maniera automatizzata**. Tale tecnica **euristica** non porterà, in generale, a sintesi ottimizzate. Supponiamo di aver ottenuto una descrizione in cui **tutti i μ -salti sono:**

- **incondizionati**, del tipo **STAR<=S2 ; oppure**
- **a due vie**, del tipo **STAR<= (c1==1) ? S0 : S1 ;**

Quale, ad esempio, quella risultante dal secondo dei due esempi presentati (nell'esempio del contatore di sequenze c'era un **salto a tre alternative**). Il salto incondizionato può essere visto come **caso limite** di salto a due vie. Infatti, lo posso sempre scrivere come:

STAR<=(c1==1)?S2:S2 // con c1 o qualunque altra var di condizionamento

Dato questo vincolo (che è un po' **pesante**, e che più tardi rimuoveremo), faccio come segue:

- a) prendo la codifica degli stati interni, che chiamo **μ -indirizzo**.
- b) In corrispondenza di un μ -indirizzo, posso individuare, nella parte controllo (disegnare tabella):
 1. lo stato delle variabili di comando, detto **μ -codice**.
 2. Associa una **codifica in base 2** ai pedici delle variabili di condizionamento. Nel caso d'esempio ho **due var. di condizionamento**, e quindi associa loro una codifica su **un bit**: $c_1 = 1$, $c_0 = 0$. Se ne avessi avute 4, c_3, \dots, c_0 , avrei codificato c_3 come 11 e c_0 come 00. Nella tabella scriverò la **codifica della variabile di condizionamento efficace** per quel μ -indirizzo, cioè quella che guida il μ -salto, che chiameremo c_{eff} . Nello stato S0, di μ -indirizzo 00, la var. di condizionamento efficace è c_1 . In S1 è c_0 , in S2 è nuovamente c_1 .
 3. Il **μ -indirizzo** cui si salta nel caso di **condizione vera**.
 4. Il **μ -indirizzo** cui si salta nel caso di **condizione falsa**.

μ -addr	μ -instruction			
	$b_1 b_0$	c_{eff}	μ -addr _T	μ -addr _F
00	00	1	00	01
01	11	0	10	01
10	01	1	00	10
11	--	-	--	--

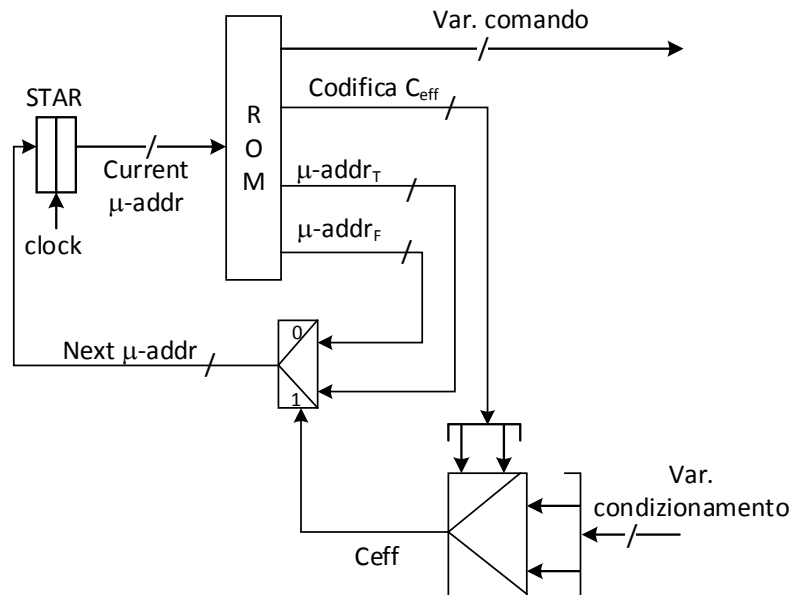
Nel caso (non presente in questo esempio) di μ -salto **incondizionato**, la codifica della var. di condizionamento efficace è **non specificata**, e la codifica dei due μ -indirizzi True e False è identica.

La tabella scritta sopra è, a tutti gli effetti, una **ROM**, con 3 celle da 7 bit ciascuna. Il contenuto di una cella è detto **μ -istruzione**, in quanto descrive compiutamente tutto quello che la parte controllo deve fare nello stato codificato con il μ -indirizzo corrispondente. Abbiamo, infatti,

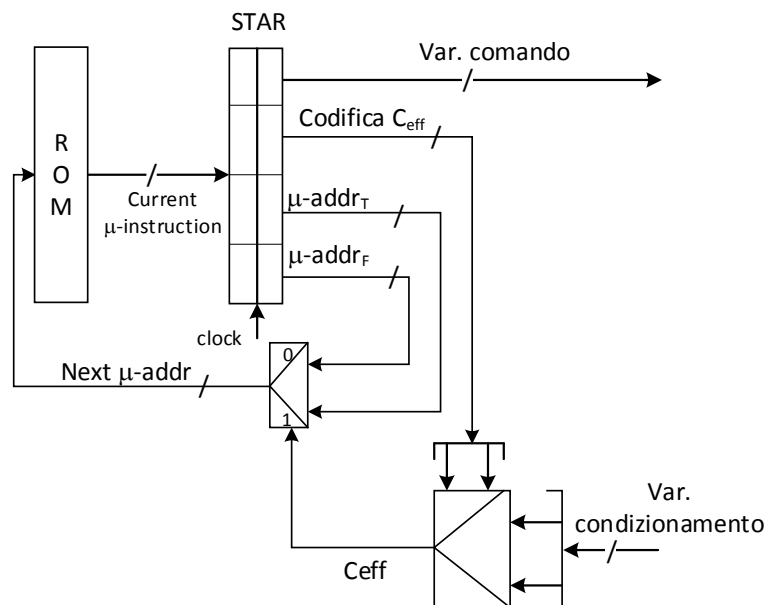
- lo stato di uscita della parte controllo (**μ -codice**);
- entrambe le alternative di salto (questo è il motivo per cui devono essere al massimo due), e la codifica della variabile di condizionamento che guida il salto.

Questa ROM **specifica completamente** il comportamento della parte controllo.

Una sintesi della parte controllo può quindi essere affrontata, in modo euristico, partendo dalla ROM appena descritta, nel seguente modo. Questo tipo di sintesi della parte controllo si chiama **μ -address based**, in quanto il registro STAR contiene, appunto, i μ -indirizzi.



Niente mi vieta, peraltro, di **scambiare di posto la ROM ed il registro STAR**. Utilizzo un registro STAR “molto grande”, che metto **a valle della ROM**. Questo tipo di sintesi della parte controllo si chiama **μ -instruction based**, in quanto il registro STAR contiene, appunto, le μ -istruzioni.



Vediamo di confrontare i due modelli. Nel primo caso (**μ -address based**), la ROM e la RC operativa sono in cascata. Visto che queste sono, normalmente, le reti più pesanti dal punto di vista del tempo di attraversamento, il periodo di clock va tenuto largo. Per contro, abbiamo un registro di

stato piccolo. Nel modello **μ -instruction based**, invece, il registro è grande (la codifica degli stati è ridondante). Per reti neanche troppo complesse si arriva facilmente a registri di 100 bit ed oltre, mentre nell'altro caso difficilmente si va sopra la decina. Però la ROM e la RC operativa non sono più in cascata, il che consente normalmente di far andare il clock più velocemente. In quest'ultimo caso, però, **sono in cascata la RC di condizionamento e la ROM**. La RC di condizionamento è di norma meno pesante di quella operativa, e quindi il problema si pone meno.

4.4 Reintrodurre i μ -salti a più vie

Abbiamo dato come vincolo il fatto che i **μ -salti siano a due alternative**. Se così non fosse, la ROM diventerebbe ingestibile. In realtà ci sono molti casi in cui mi fa comodo poter usare più alternative per un μ -salto. Anzi, è frequente il caso in cui i μ -salti sono **quasi sempre a due alternative, tranne qualche caso (pochi) in cui ce ne sono N , con N piuttosto grande**. Possiamo anticipare che, quando vedremo la descrizione del **processore** come RSS, avremo quasi sempre salti a due alternative, tranne in due casi:

- all'**inizio** della fase di fetch, quando devo **decodificare il formato** dell'istruzione, e quindi saltare ad un certo numero di blocchi di μ -istruzioni differenti a seconda del formato. I formati delle istruzioni sono più di due (nell'ordine di diverse unità).
- alla **fine** di ogni fase di fetch relativa a ciascun formato di istruzione, quando devo decidere quale istruzione, di quelle consentite in quel formato, devo andare ad eseguire.

È chiaro che, in teoria, **potrei sempre scrivere un μ -salto a N vie utilizzando soltanto μ -salti a due vie**. Vediamo come si fa:

```
S0: begin /*elaborazioni;*/
      STAR<=(condizione1)?S1:
        (condizione2)?S2:
        [...]
        (condizionek-1)?Sk-1:Sk;
end
```

può sempre essere tradotto in:

```
S0:      begin /*elaborazioni;*/ STAR<=(condizione1)?S1:S0_1; end
S0_1:    begin STAR<=(condizione2)?S2:S0_2; end
S0_2:    begin STAR<=(condizione3)?S3:S0_3; end
[...]
S0_k-1:  begin STAR<=(condizionek-1)?Sk-1:Sk; end
```

Il problema è che mentre scorro questi stati **il tempo passa**. Se realizzo un processore come RSS, non posso certo pensare di perdere decine di cicli di clock per prelevare e decodificare ogni istruzione del linguaggio macchina. Né si può prevedere spazio per 50 indirizzi alternativi nella ROM,

soltanto perché in **due casi** su chissà quante μ -istruzioni mi servono salti con 50 alternative. Il problema si risolve usando un **registro operativo** apposta, detto **Multiway Jump Register, MJR**. A questo punto, la parte di μ -programma scritto sopra la posso tradurre in:

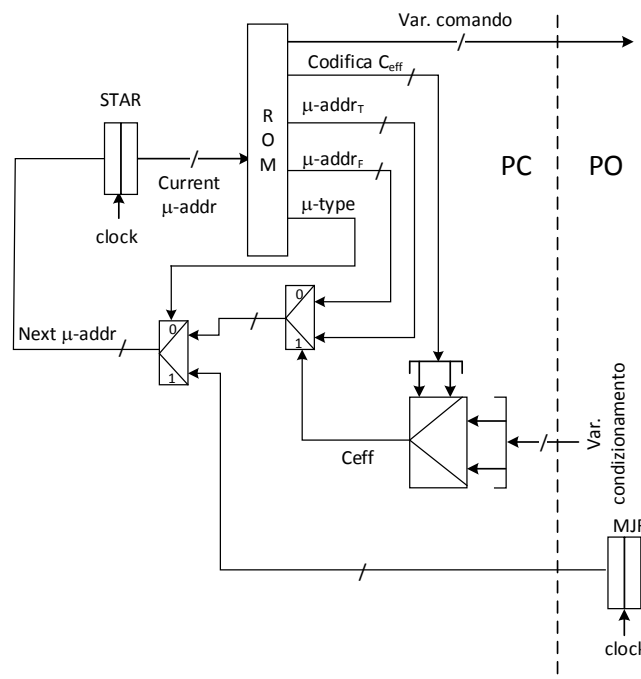
```

S0:      begin /*elaborazioni;*/
          MJR<=(condizione1) ?S1:
            (condizione2) ?S2:
            [...]
            (condizionek-1) ?Sk-1:Sk;
          STAR<=S0_1;
        end
S0_1:    begin STAR<=MJR; end

```

Ed in **due stati interni posso gestire qualunque μ -salto a N vie**, qualunque sia il valore di N .

Questo per quanto riguarda la **descrizione**. Vediamo come questa prassi si traduce nella **sintesi**. MJR è un registro operativo come un altro, e quindi sintetizzo la parte di RC operativa che lo riguarda usando la tecnica nota di registro multifunzionale. La sintesi, di norma, si presta a qualche ottimizzazione (vediamo un esempio fra un attimo). Per quanto riguarda la **parte controllo della rete**, mi fa comodo mantenere la consueta struttura con μ -salti a due vie (perché la maggior parte dei μ -salti sarà a due vie), aggiungendo in più la possibilità di guidare un salto con il contenuto di MJR.



Nella ROM serve una **variabile in più**, che discrimina se il μ -salto che sto per fare è guidato dalle variabili di condizionamento o da MJR. Tale variabile si chiama **μ -tipo** del μ -salto. Nell'esempio di sopra, in S0 il μ -tipo sarà pari a 0 (μ -salto guidato dalle variabili di condizionamento – peraltro c_{eff}

sarà non specificata perché il salto è incondizionato), ed in S0_1 sarà pari ad 1 (μ -salto guidato da MJR).

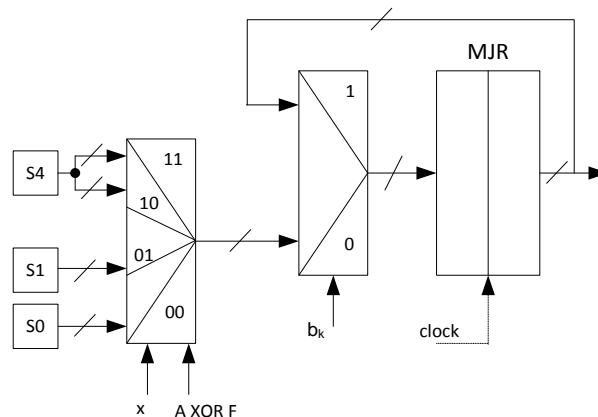
4.4.1 Esempio di sintesi con uso di registro MJR

Prendiamo una rete descritta (a spanne) in questo modo, ed affrontiamo la sintesi di parte della ROM e della parte di RC operativa relativa a MJR.

```
[...]
input x;
reg [K-1:0] STAR, MJR;
reg A,B,F;
[...]
```

```
S0: begin /*elaborazioni*/ STAR<=(x==0)?S0:S1; end
S1: begin /*elaborazioni*/ STAR<=S2; end
S2: begin /*elaborazioni*/ STAR<=(A!=B)?S5:S3; end
S3: begin /*elaborazioni*/ STAR<=(gamma(A)==0)?S1:S4; end
S4: begin /*elaborazioni*/ STAR<=S5; end
S5: begin MJR<=(x==1)?S4:(A!=F)?S1:S0; STAR<=S6; end
S6: begin /*elaborazioni*/ STAR<=MJR; end
```

In questo caso, la RC operativa dietro MJR sarà un registro multifunzionale **a due vie**, caricamento di qualcosa (S5) e conservazione (tutti gli altri stati). Questo qualcosa dipende da una rete combinatoria che andiamo a sintetizzare:



E quanto ho disegnato a sinistra si presta a ovvie ottimizzazioni una volta che ho deciso la codifica degli stati interni (gli ingressi al multiplexer sono infatti **costanti**, cioè gruppi di fili attaccati a massa o all'alimentazione). Per quanto riguarda la ROM, almeno la parte che possiamo sintetizzare con le informazioni che abbiamo, osserviamo che:

- abbiamo 3 condizioni indipendenti, corrispondenti ai μ -salti a due vie in S0, S2, S3.
- Abbiamo alcuni μ -salti incondizionati S1, S4, S5
- Abbiamo un salto guidato da MJR in S6.

- Abbiamo necessità di una variabile di comando per controllare MJR (più altre che non possiamo decidere non avendo dettagliato le elaborazioni compiute dalla parte operativa).
- Non abbiamo deciso alcuna codifica per gli stati interni, e quindi li riporteremo con il nome simbolico per semplicità. Ci vorranno, comunque, 3 bit.

μ -addr	μ -code			c_{eff}	μ -addr T	μ -addr F	μ -type
	...	b_k	...				
S0	...	1	...	00	S0	S1	0
S1	...	1	...	--	S2	S2	0
S2	...	1	...	01	S5	S3	0
S3	...	1	...	10	S1	S4	0
S4	...	1	...	--	S5	S5	0
S5	...	0	...	--	S6	S6	0
S6	...	1	...	--	--	--	1

4.4.2 Sottoliste

MJR può essere utilizzato anche per implementare le **sottoliste**. In alcuni casi fa comodo strutturare una descrizione di RSS con sottoliste equivalenti a **sottoprogrammi**, cioè a **pezzi di μ -programma che possono essere raggiunti a partire da stati di partenza diversi**, di modo che il controllo possa:

- a) passare temporaneamente al sottoprogramma
- b) **riprendere dallo stato successivo a quello in cui si era interrotta.**

Questa cosa si realizza salvando, all'atto della "chiamata di sottoprogramma", la codifica dello stato di ritorno dentro MJR. Alla fine del sottoprogramma inserirò un salto guidato da MJR, che avrà il ruolo equivalente a quello di una istruzione *return* di un linguaggio di programmazione.

```

S0:  begin  /*elaborazioni*/ MJR<=S1; STAR<=Ssub1; end
S1:  begin  /*elaborazioni*/; end

[...]

Sx:   begin  /*elaborazioni*/ MJR<=Sx+1; STAR<=Ssub1; end
Sx+1: begin  /*elaborazioni*/; end

[...]

Ssub1: begin  /*elaborazioni*/ end
[...]
SsubK: begin  /*elaborazioni*/ STAR<=MJR; end
```

Lista princ.

Sottolista

Attenzione: il parallelo con i sottoprogrammi si limita **ad un livello di annidamento**. Non posso annidare più sottoprogrammi, se ho un solo registro MJR. Se lo volessi fare, mi servirebbe una **μ -pila di registri MJR** (volendo è possibile, ma non lo vediamo).

Nota finale: a meno che non sia assolutamente necessario o estremamente comodo (o esplicitamente richiesto nel testo di un esercizio), **evitare di usare il registro MJR**. In genere la maggior parte degli esercizi che dovrete svolgere può essere svolta usando salti a due vie.

4.5 Riflessione conclusiva su descrizione e sintesi delle reti logiche

Abbiamo visto vari tipi di reti logiche: quelle combinatorie, sia semplici (pochi ingressi ed uscite) sia complesse (e.g., quelle per l'aritmetica, caratterizzate da molti ingressi ed uscite); quelle sequenziali asincrone; quelle sequenziali sincronizzate, sia semplici (cioè con pochi ingressi, stati interni ed uscite) che complesse (le ultime che abbiamo visto). Siamo al punto giusto per riprendere in mano il progetto di una rete logica con maggior cognizione di causa.

Una rete logica deve essere **prima descritta**, e **poi sintetizzata**.

La **descrizione** altro non è che un **modo formale** di fornire le **specifiche del comportamento** della rete medesima. Infatti:

- Nel caso di una **rete combinatoria**, è un'associazione tra stati di ingresso e stati di uscita, per esempio scritta sotto forma di tabella di verità.
- Nel caso di una **rete sequenziale** (asincrona o sincronizzata), cioè di una rete con memoria, è un **diagramma a stati**, che può essere rappresentato:
 - a) tramite tabella o grafo se la rete è abbastanza semplice, oppure
 - b) tramite un formalismo più complesso, in cui ad ogni stato interno vengono associate delle azioni che la rete esegue (nel nostro caso, assegnamenti a registri).

Per descrivere una rete esistono **formalismi differenti**, che si adattano meglio o peggio ad un particolare tipo di rete. Anche per uno stesso tipo di rete si possono usare più formalismi diversi. Ad esempio, per una rete combinatoria possiamo usare indifferentemente una **tabella di verità** o **poche righe di Verilog**.

x2	x1	x0	z
0	0	0	0
0	0	1	1
0	1	0	0
...

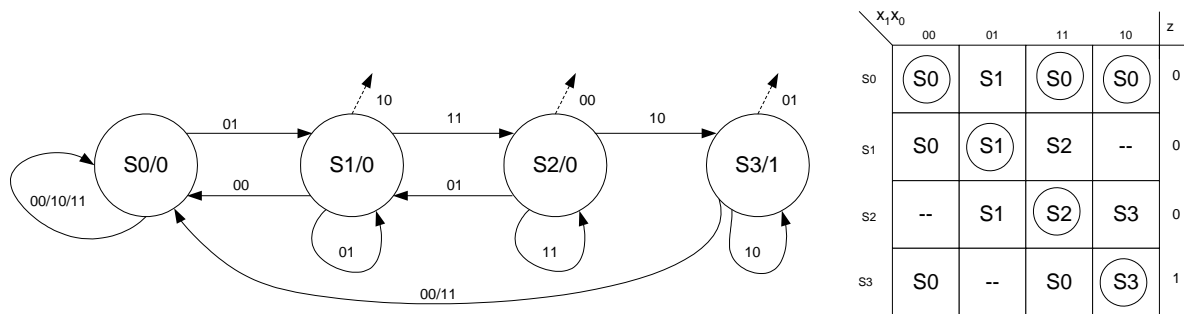
```

module Rete(x2, x1, x0, z);
  input x2, x1, x0;
  output z;
  assign z = ({x2,x1,x0} == 'B000) ?      'B0 :
              ({x2,x1,x0} == 'B001) ?      'B1 :
              ({x2,x1,x0} == 'B010) ?      'B0 :
              ...
endmodule

```

È chiaro che la tabella di verità e la descrizione in Verilog scritta accanto hanno la medesima semantica. È altresì chiaro che entrambe dicono **cosa fa** la rete, **ma non come è realizzata**, cioè quali sono le porte logiche che la compongono.

Allo stesso modo, la descrizione di una RSA si può dare sotto forma di grafo di flusso, di tabella di flusso, o di linguaggio Verilog. Quest'ultimo non l'abbiamo mai usato, ma è chiaro che avremmo potuto farlo.



È **indispensabile** avere a disposizione un modo formale per descrivere una rete, perché una descrizione formale può essere **verificata**:

- verificare la descrizione di una RC significa controllare che gli stati di uscita siano quelli desiderati per ogni possibile stato di ingresso. È una verifica **statica**, che si fa per ispezione diretta.
- Verificare la descrizione di una RS (asincrona o sincronizzata) significa **simulare l'evoluzione della rete** a partire da una condizione iniziale di reset, e controllare che questa sia coerente con le specifiche (normalmente date a parole). Questa è una verifica **dinamica** (richiede un'evoluzione temporale della rete), concettualmente simile al processo di testing di un software. Così come è difficile, se non impossibile, testare il software in maniera esaustiva (i.e., per tutti i possibili input), è difficile verificare in maniera esaustiva il comportamento di reti sequenziali che non siano estremamente semplici. Nondimeno, è **sbagliato** non verificare la descrizione di una RS, tanto quanto è sbagliato non testare del software. Un **diagramma di temporizzazione** che mostra la simulazione di una RSS complessa con un certo input, come quelli che abbiamo fatto svariate volte finora, è un modo per verificare la descrizione di quella rete.

Ciò che rende un **formalismo di descrizione** preferibile rispetto ad un altro è **quanto è comodo da usare**. Per una RC semplice una tabella di verità o una mappa di Karnaugh sono (leggermente) più comodi di una descrizione in Verilog. La verifica statica di una tabella di verità è (leggermente) più agevole rispetto a quella di una descrizione in Verilog.

Allo stesso modo, per una RSA (semplice) una tabella di flusso è più comoda da scrivere di una descrizione in Verilog, e soprattutto è più facile – per un utente umano – simulare l’evoluzione della rete sulla tabella di flusso che sulla descrizione in Verilog. Se invece volessi avvalermi di un simulatore Verilog per simulare il comportamento di una RSA, mi converrebbe ovviamente scriverne la descrizione in Verilog.

Viceversa, per una RSS complessa, una descrizione in Verilog risulta ben leggibile, e può essere agevolmente fornita ad un simulatore Verilog per verificare il comportamento della rete.

Una descrizione non può fornire informazioni su **come è realizzata la rete**, cioè quali sono i suoi componenti elementari e come sono interconnessi. Per arrivare a questo livello è necessario procedere alla **sintesi**, che è **sempre** il punto di arrivo degli esercizi che svolgiamo. Fermarsi alla descrizione, infatti, comunica l’impressione di **non essere in grado di realizzare** ciò che si è pensato, impressione che un ingegnere non dovrebbe mai dare.

Esistono **due approcci** per la sintesi, e li abbiamo usati entrambi: quello **euristico**, che consiste nel “fare le cose ad occhio”, e quello **formale**, che consiste nel seguire un procedimento algoritmico. L’approccio euristico viene usato tipicamente nel caso di **reti combinatorie per l’aritmetica**. La sintesi richiesta negli esercizi di aritmetica consiste, di fatto, nel prendere dei blocchi “atomici”, e.g., sommatore, moltiplicatore, etc. – la cui struttura interna è data per assodata – ed assemblarli in modo che soddisfino delle specifiche date a parole (il testo dell’esercizio). Approcci euristici per la risoluzione di problemi sono tipici del know-how di un ingegnere (il processo di scrittura del software a partire dalle specifiche è, infatti, un procedimento euristico), e richiedono l’esperienza che si acquisisce solo con la pratica.

Abbiamo anche usato **approcci formali** per la sintesi delle reti logiche. Ad esempio:

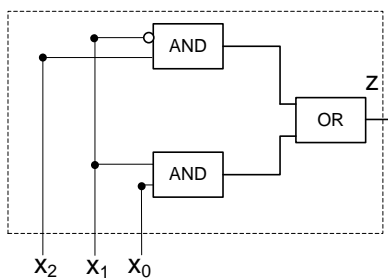
- La sintesi a costo minimo in forma SP (PS, NAND, NOR) di una RC;
- la sintesi di una RSS di Moore, Mealy, Mealy Ritardato, secondo il modello con registri FF-D o FF-JK;
- la sintesi di una RSS complessa secondo il modello con scomposizione in Parte Operativa e Parte Controllo.

In tutti questi casi, il processo seguito è di tipo algoritmico. Si parte dalla **descrizione** della rete, si adotta un **modello di sintesi**, e si procede secondo i passi dell’algoritmo. La scelta del modello di sintesi determina l’algoritmo. Ad esempio, l’algoritmo di sintesi di una RSS di Moore è diverso se

scelgo di usare il modello con registro di stato FF-D o quello con registro di stato FF-JK. L'algoritmo di sintesi di una RC è diverso a seconda che scelga un modello di sintesi SP o PS.

Sia come sia, alla fine una **sintesi** deve essere comunicata a qualcuno (e.g., il tecnico che realizzerà l'hardware) in forma intelligibile, e quindi deve essere **scritta secondo un qualche formalismo** pure lei. Come per le descrizioni, abbiamo usato diversi formalismi, scegliendo di volta in volta quello **più comodo perché più facile da leggere**.

Ad esempio, per le RC semplici abbiamo usato indifferentemente **diagrammi, espressioni di algebra di Boole, o il linguaggio Verilog**.



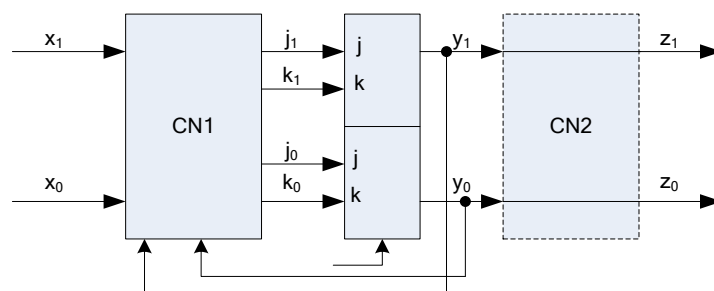
```
module Rete (x2, x1, x0, z);
    input x2, x1, x0;
    output z;
    assign z = (x1 & x0) |
               (x2 & ~x1);
endmodule
```

$$z = (x_1 \cdot x_0) + (x_2 \cdot \overline{x_1})$$

Tutti e tre sono equivalenti, ma noi preferiamo usare le espressioni algebriche perché sono più compatte. Si noti che il linguaggio **Verilog** può essere usato sia per rappresentare la **descrizione che la sintesi**. Ciò è spesso fonte di confusione, perché si confonde il **formalismo** (cioè il linguaggio Verilog) con il **contenuto** (cioè la descrizione o la sintesi, a seconda dei casi). È però chiaro che il pezzo di Verilog scritto sopra è una **sintesi**, in quanto è mappabile direttamente su porte logiche interconnesse. Non è una descrizione perché non dice come stati di uscita corrispondono a stati di ingresso.

Per la sintesi di RC complesse (quali quelle per l'aritmetica) usiamo normalmente **diagrammi**. Più raramente, scriviamo sintesi in Verilog (sono più difficili da leggere).

Per le RSS di Moore, Mealy, e Mealy Ritardato usiamo normalmente un **diagramma** che specifica quale **modello di sintesi** abbiamo scelto, ed **espressioni algebriche** che descrivono le relazioni ingresso-uscita delle reti combinatorie facenti parte del modello.



$$j_1 = k_1 = \overline{y_0} \cdot \overline{x_1} \cdot \overline{x_0} + \overline{y_0} \cdot x_1 \cdot x_0 + y_0 \cdot \overline{x_1} \cdot \overline{x_0} + y_0 \cdot x_1 \cdot x_0$$

$$j_0 = k_0 = 1$$

Per le RSS complesse, è di gran lunga più conveniente usare il **Verilog** per scrivere la sintesi, sempre stando attenti a non confondere il formalismo con il contenuto, visto che usiamo il Verilog **anche** per scrivere la descrizione. Scrivere la sintesi sotto forma di diagramma in questo caso richiederebbe fogli enormi e grossi intrighi di fili, e ne risulterebbe qualcosa di difficile da leggere. **Resta inteso**, in ogni caso, che ciò che si scrive quando si fa una sintesi in Verilog è **in corrispondenza biunivoca con un diagramma**, in cui:

- la parte operativa è composta da registri multifunzionali, che hanno le variabili di comando b_j come ingressi di comando dei propri multiplexer;
- ci sono reti combinatorie che generano le variabili di condizionamento;
- la parte di controllo consiste in un registro di stato ed una ROM (e poco altro), ed ha in ingresso le variabili di condizionamento e in uscita quelle di comando.

Non aver capito questo significa non aver capito cosa si sta facendo, ed è **grave**. Per rendere chiaro che si è capito, negli esercizi di esame è **sempre** richiesto di disegnare almeno i diagrammi delle reti combinatorie di condizionamento e di specificare il contenuto della ROM sotto forma di tabella di verità. Talvolta, può essere richiesto di disegnare anche alcune porzioni della parte operativa.

Fare una **sintesi** di una rete sequenziale senza averne fatto la descrizione (errore che capita di vedere talvolta durante la correzione dei compiti) è cosa completamente **assurda: è praticamente impossibile inferire** il comportamento della rete a partire da una sintesi, e quindi non si riesce a verificare la correttezza della medesima rispetto a delle specifiche date.

Come nota a margine, si osserva che per una **rete combinatoria** è invece relativamente semplice risalire alla descrizione (e.g., alla tabella di verità) a partire dalla sintesi – il che non è comunque un buon motivo per saltare la descrizione. Ciò è dovuto al fatto che le reti combinatorie non hanno memoria.

Una sintesi **deve quindi essere coerente con la descrizione che l'ha prodotta**, in modo tale che la correttezza del comportamento della rete (che si verifica sulla descrizione) sia mantenuta nella implementazione della medesima. I procedimenti formali per sintetizzare le reti (quelli elencati sopra) partono infatti da una descrizione e la realizzano secondo un modello. Tali procedimenti **garantiscono** che la rete così sintetizzata si comporti nel modo specificato dalla sua descrizione.

Ad esempio, una RSS di Moore sintetizzata a partire dalla tabella di flusso secondo il modello con FF-D come elementi di memorizzazione, con clock dimensionato opportunamente, si comporta come specificato nella tabella di flusso, se pilotata correttamente. Analogamente, una RSS complessa sintetizzata secondo il modello PO/PC a partire dalla descrizione, si comporterà come la descrizione, purché il clock sia dimensionato in modo corretto e gli input non vengano modificati a cavallo dei fronti di salita del clock.

Alcune ottimizzazioni in fase di sintesi sono talvolta possibili. Si deve tener presente, però, che le sintesi non si giudicano dal livello di ottimizzazione: se fosse necessario ottenere una sintesi “di costo minimo”, qualunque cosa questo voglia dire, lo si farebbe fare ad un programma.

Ricapitolando: il progetto di una rete logica (qualunque) si affronta nel seguente modo:

1. **descrizione**, per stabilire in modo formale (e quindi verificabile) qual è il comportamento della rete;
2. **sintesi**, a partire dalla descrizione e seguendo un apposito modello, per realizzare una rete che si comporta come specificato nella descrizione.

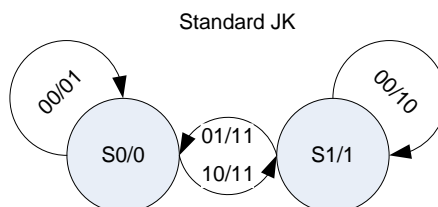
Per rappresentare i risultati di entrambi i passi si usano dei **formalismi**, quelli che meglio si adattano al tipo di rete. Il fatto che lo stesso formalismo (in particolare, il Verilog) si possa usare per entrambi i passi non può essere fonte di confusione, se si è capito cosa si sta facendo. La scelta del formalismo da usare nella descrizione e nella sintesi risponde a criteri di facilità di utilizzo ed economicità di spazio. In particolare:

- per le RC (semplici), una descrizione come tabella di verità è più leggibile. Una descrizione come mappa di Karnaugh è equivalente, e facilita il procedimento di sintesi secondo uno qualunque dei modelli noti. La sintesi si dà sotto forma di espressioni algebriche.
- per le RSA e le RSS di Moore, Mealy e Mealy ritardato con pochi ingressi e pochi stati interni, una descrizione come tabella di flusso è facile da verificare, ed inoltre facilita il procedimento di sintesi secondo uno dei modelli noti. La sintesi si dà sotto forma di indicazione del modello da utilizzare, più le espressioni algebriche delle uscite delle reti combinatorie facenti parte del modello.
- Per le RSS complesse, una descrizione in Verilog è facile da verificare, ed inoltre facilita il procedimento di sintesi secondo il modello con scomposizione in PO/PC. La sintesi si dà parimenti in Verilog, con alcuni diagrammi e tabelle di verità a completamento.

5 Soluzioni di esercizi proposti

5.1 Soluzione esercizio 3.5.4

Il JK standard può essere descritto con il seguente diagramma a stati.

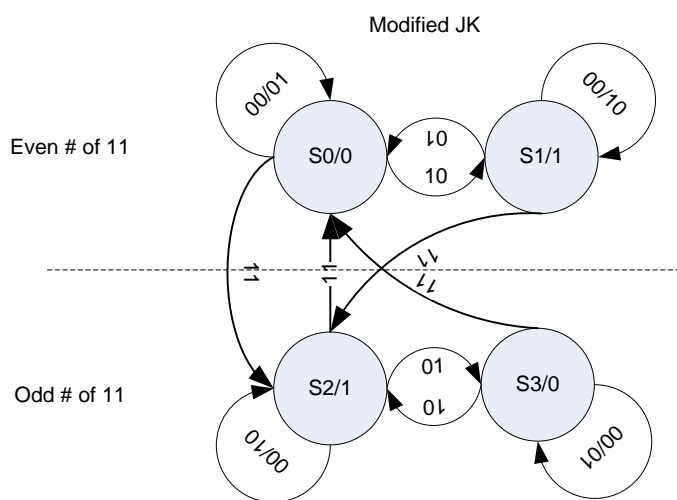


Il JK modificato si differenzia dal precedente per il fatto che il comportamento quando $JK=11$ varia

- se il n. di volte che è stato dato in ingresso $JK=11$ è **pari**, allora l'uscita è **0**
- se il n. di volte che è stato dato in ingresso $JK=11$ è **dispari**, allora l'uscita è **1**

Quindi, **il numero di stati** va differenziato a seconda che:

- l'uscita valga 1 o 0
- il n. di volte che è stato dato 11 sia pari o dispari.



La tabella di flusso è la seguente:

$j \ k$						q
		00	01	11	10	
S_0	S_0	S_0	S_0	S_2	S_1	0
S_1	S_1	S_1	S_0	S_2	S_1	1
S_2	S_2	S_2	S_3	S_0	S_2	1
S_3	S_3	S_3	S_3	S_0	S_2	0

Sintesi della rete a porte NOR

- scelta di una **codifica degli stati**: $S_0 = 00$, $S_1 = 01$, $S_2 = 11$, e $S_3 = 10$

La scelta può essere guidata dalla semplicità di implementazione della rete combinatoria RCB che produce l'uscita dallo stato interno. Con questa scelta, la rete combinatoria è un corto circuito perché q è uguale ad una delle variabili logiche con cui si codifica lo stato.

- scelta di un **modello strutturale**: quale meccanismo uso per la marcatura? Ho 2 alternative
 - o flip-flop D-positive-edge-triggered

- flip-flop JK

Così come nel caso di reti asincrone ho come alternative un corto circuito (o un ritardo, se la rete ha alee essenziali) oppure un flip-flop SR.

Supponiamo di adottare il **modello strutturale con D-positive-edge-triggered**. Visto che nelle celle della tabella ci devo mettere gli ingressi da dare al meccanismo di marcatura, dovrò mettere direttamente la codifica del nuovo stato interno a_1a_0

$\begin{matrix} jk \\ y_1y_0 \end{matrix}$	a_1a_0				q
	00	01	11	10	
00	00	00	11	01	0
01	01	00	11	01	1
11	11	10	00	11	1
10	10	10	00	11	0

Sintesi a porte **NOR** Devo fare una **sintesi PS**, che posso poi trasformare a porte NOR. Sintetizziamo le due variabili separatamente.

$\begin{matrix} jk \\ y_1y_0 \end{matrix}$	\bar{a}_1				q
	00	01	11	10	
00	1	1	0	1	0
01	1	1	0	1	1
11	0	0	1	0	1
10	0	0	1	0	0

$\begin{matrix} jk \\ y_1y_0 \end{matrix}$	\bar{a}_0				q
	00	01	11	10	
00	1	1	0	0	0
01	0	1	0	0	1
11	0	1	1	0	1
10	1	1	1	0	0

$$\bar{a}_1 = \bar{j} \cdot \bar{y}_1 + \bar{k} \cdot \bar{y}_1 + j \cdot k \cdot y_1$$

$$\bar{a}_0 = \bar{j} \cdot k + k \cdot y_1 + \bar{j} \cdot \bar{y}_0$$

Applicando DeMorgan si ottiene:

$$\bar{a}_1 = \overline{(j + y_1) + (k + y_1) + (\bar{j} + \bar{k} + \bar{y}_1)}$$

$$\bar{a}_0 = \overline{(j + \bar{k}) + (j + y_0) + (\bar{k} + \bar{y}_1)}$$

Da cui, complementando, si ottiene la soluzione:

$$\boxed{\begin{aligned} a_1 &= \overline{(j + y_1) + (k + y_1) + (\bar{j} + \bar{k} + \bar{y}_1)} \\ a_0 &= \overline{(j + \bar{k}) + (j + y_0) + (\bar{k} + \bar{y}_1)} \\ q &= y_0 \end{aligned}}$$

Supponiamo di adottare il **modello strutturale con flip-flop JK**. Visto che nelle celle della tabella ci devo mettere gli ingressi da dare al meccanismo di marcatura, **non** dovrò stavolta mettere direttamente la codifica del nuovo stato interno a_1a_0 , ma gli ingressi da dare ai 2 flip-flop JK affinché portino le loro uscite a coincidere con il nuovo stato interno.

$j k$ $y_1 y_0$		Nuovo stato interno				q
		00	01	11	10	
00	00	00	00	11	01	0
01	01	01	00	11	01	1
11	11	11	10	00	11	1
10	10	10	10	00	11	0

$j k$ $y_1 y_0$		$j_1 k_1$				q
		00	01	11	10	
00	00	0-	0-	1-	0-	0
01	01	0-	0-	1-	0-	1
11	11	-0	-0	-1	-0	1
10	10	-0	-0	-1	-0	0

$j k$ $y_1 y_0$		$j_0 k_0$				q
		00	01	11	10	
00	00	0-	0-	1-	1-	0
01	01	-0	-1	-0	-0	1
11	11	-0	-1	-1	-0	1
10	10	0-	0-	0-	-0	0

Con il che fare una sintesi, qualunque ne sia il tipo, risulta estremamente semplice per l'alto numero di valori non specificati.

Facciamo per esempio la sintesi della parte di rete che produce le variabili j_1, k_1 a porte NOR.

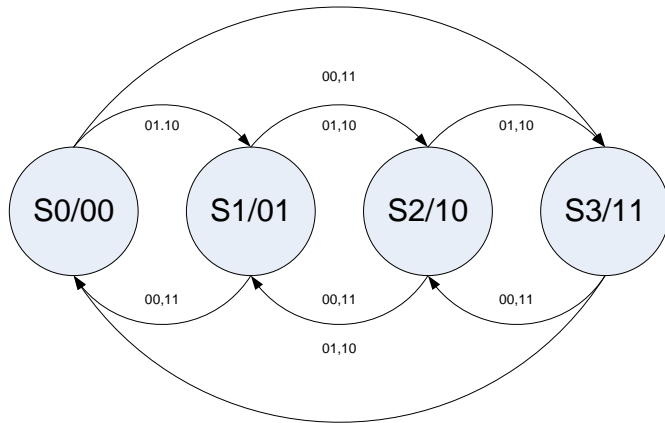
$j k$ $y_1 y_0$		$\overline{j_1}$				q
		00	01	11	10	
00	00	1	1	0	1	0
01	01	1	1	0	1	1
11	11	-	-	-	-	1
10	10	-	-	-	-	0

$j k$ $y_1 y_0$		$\overline{k_1}$				q
		00	01	11	10	
00	00	-	-	-	-	0
01	01	-	-	-	-	1
11	11	1	1	0	1	1
10	10	1	1	0	1	0

Da cui: $\overline{j_1} = \overline{k_1} = \overline{j + k}$, cioè $j_1 = k_1 = \overline{j + k}$ (lasciare l'altra per esercizio).

5.2 Soluzione esercizio 3.5.5

Il diagramma e la tabella di flusso della rete in questione sono riportati in figura



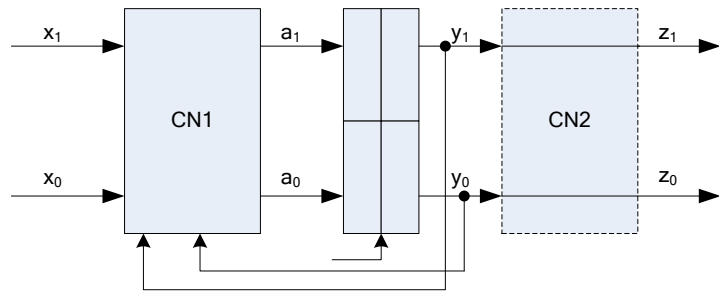
x_1x_0	00	01	11	10	z_1z_0
S0	S3	S1	S3	S1	00
S1	S0	S2	S0	S2	01
S2	S1	S3	S1	S3	10
S3	S2	S0	S2	S0	11

Scegliendo per ciascuno stato una codifica su due bit equivalente al valore che le variabili di uscita assumono in quello stato, si ottiene una rete RCB di complessità nulla, quale che sia il modello strutturale usato.

Utilizzando un modello strutturale che prevede flip-flop D-positive-edge-triggered come meccanismi di marcatura, la mappa di Karnaugh per la rete RCA è la seguente:

y_1y_0	x_1x_0	00	01	11	10
S0	00	11	01	11	01
S1	01	00	10	00	10
S3	11	10	00	10	00
S2	10	01	11	01	11

a_1a_0



Dalle mappe di Karnaugh sopra riportate si ricava la seguente sintesi SP:

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

$$a_0 = \overline{y_0}$$

Il cui **costo a porte** è **9** ed il cui **costo a diodi** è **40**.

Si può osservare che la copertura di x_1 è **a scacchi**. La funzione f che riconosce gli stati di ingresso

- dipende da tutte e quattro le variabili
- deve essere fatta in modo tale che, se $f(X) = t$, $f(X') = \bar{t}$, per ogni stato di ingresso X , se X' è adiacente a X .

Due stati adiacenti differiscono sempre per il **numero di variabili ad 1**, si conclude che la funzione richiesta deve avere un valore di uscita che dipende **soltanto** dal **numero di bit a 1** dello stato di

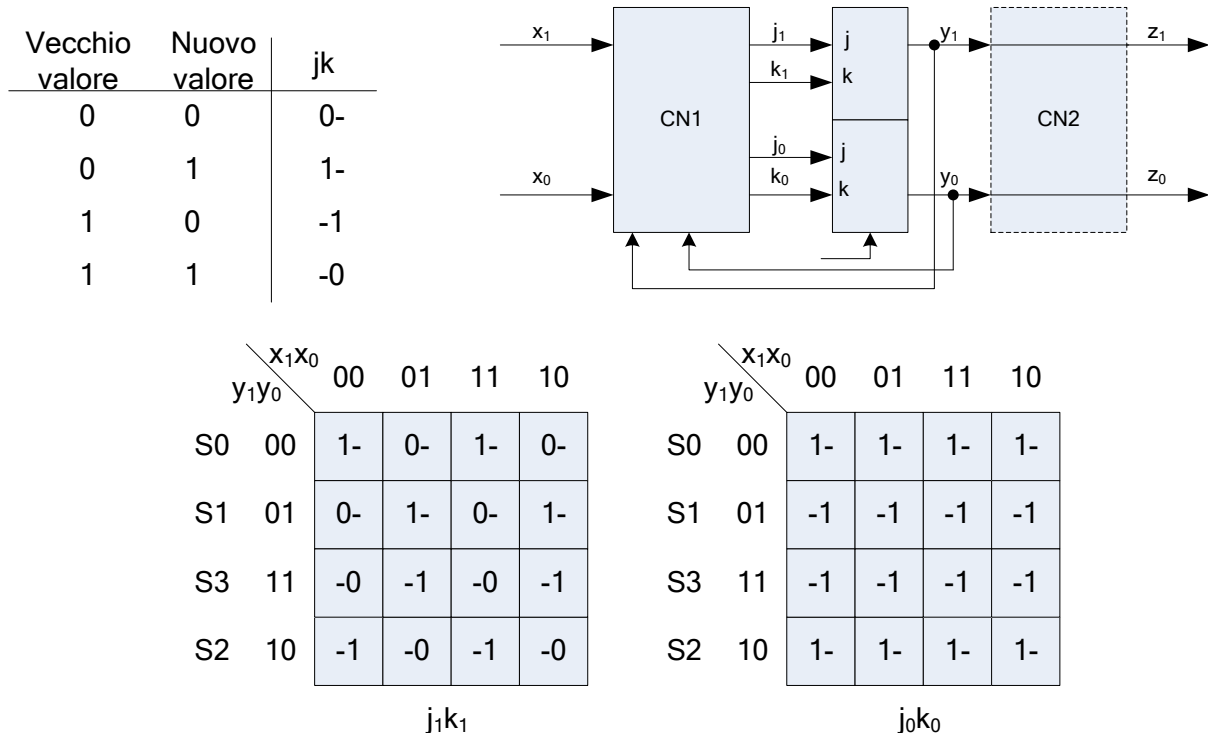
ingresso. Deve essere, appunto, uno XOR a quattro ingressi. Visto che $f(0000)=1$, allora devo negare l'uscita.

La variabile a_1 può essere riscritta come segue: $a_1 = \overline{y_1 \otimes y_0 \otimes x_1 \otimes x_0}$, dal che si ricava che, utilizzando soltanto **porte XOR e NOT**, il costo a porte è 3 e quello a diodi è 6.

Ci si arriva, volendo, per via algebrica:

$$\begin{aligned}
 a_1 &= \overline{y_1 \cdot y_0} \cdot (\overline{x_1 \cdot x_0} + x_1 \cdot x_0) + \overline{y_1 \cdot y_0} \cdot (\overline{x_1 \cdot x_0} + x_1 \cdot x_0) \\
 &\quad + y_1 \cdot \overline{y_0} \cdot (\overline{x_1 \cdot x_0} + x_1 \cdot x_0) + y_1 \cdot y_0 \cdot (\overline{x_1 \cdot x_0} + x_1 \cdot x_0) \\
 &= (\overline{y_1 \cdot y_0} + y_1 \cdot y_0) \cdot (\overline{x_1 \cdot x_0} + x_1 \cdot x_0) + (\overline{y_1 \cdot y_0} + y_1 \cdot y_0) \cdot (\overline{x_1 \cdot x_0} + x_1 \cdot x_0) \\
 &= (\overline{y_1 \otimes y_0}) \cdot (\overline{x_1 \otimes x_0}) + (y_1 \otimes y_0) \cdot (x_1 \otimes x_0) \\
 &= \overline{(y_1 \otimes y_0) \otimes (x_1 \otimes x_0)} \\
 &= \overline{y_1 \otimes y_0 \otimes x_1 \otimes x_0}
 \end{aligned}$$

Volendo, si possono utilizzare **flip-flop JK** come meccanismo di marcatura. In questo caso, ricaviamo la tabella per la rete RCA a partire dalla tabella di flusso e dalla tabella di applicazione del flip-flop JK.



Dalle mappe di Karnaugh sopra riportate si ricava la seguente sintesi SP:

$$\begin{aligned}
 j_1 &= k_1 = \overline{y_0 \cdot x_1 \cdot x_0} + \overline{y_0 \cdot x_1 \cdot x_0} + \overline{y_0 \cdot x_1 \cdot x_0} + \overline{y_0 \cdot x_1 \cdot x_0} \\
 j_0 &= k_0 = 1
 \end{aligned}$$

Il cui costo **a porte** è **5** ed il cui costo **a diodi** è **16**.

Nella sintesi SP di j_1 e k_1 i valori non specificati sono stati assunti come 1 o 0 in accordo alla procedura di sintesi a costo minimo in forma SP di reti parzialmente specificate. Assumendo invece pari ad 1 i valori non specificati corrispondenti alle caselle evidenziate nelle mappe sottostanti, è immediato ottenere che

$$k_1 = \overline{j_1} = y_1 \otimes y_0 \otimes x_1 \otimes x_0.$$

		x_1x_0			
		00	01	11	10
y_1y_0	S0 00	1-	0-	1-	0-
	S1 01	0-	1-	0-	1-
	S3 11	-0	-1	-0	-1
	S2 10	-1	-0	-1	-0

j_1

		x_1x_0			
		00	01	11	10
y_1y_0	S0 00	1-	0-	1-	0-
	S1 01	0-	1-	0-	1-
	S3 11	-0	-1	-0	-1
	S2 10	-1	-0	-1	-0

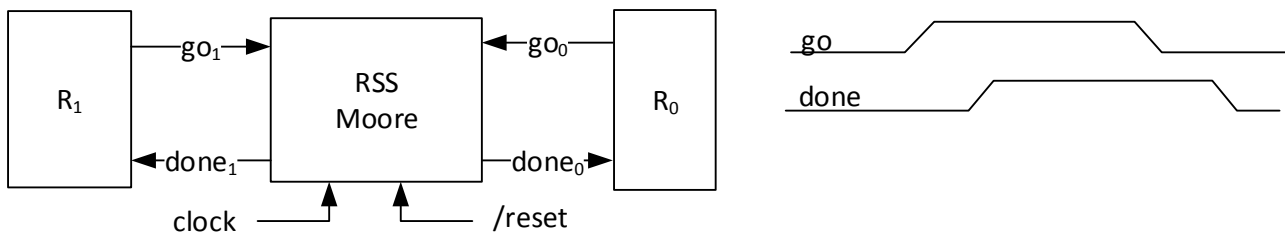
k_1

Quindi, utilizzando soltanto porte XOR e NOT il costo a porte è 3 e quello a diodi 6.

6 Altri esercizi svolti

6.1 Esercizio – RSS di Moore

Descrivere e sintetizzare una rete sequenziale sincronizzata di Moore che ha due variabili di ingresso go_1 , go_0 , e due variabili di uscita $done_1$, $done_0$. Tali variabili supportano due handshake con altrettante reti R_1 ed R_0 , e gli handshake sono disegnati in figura. Le due reti R_1 ed R_0 si attengono al protocollo di handshake.



Si supponga che al reset tutti gli handshake siano a riposo. La RSS gestisce **un handshake alla volta**, e le due uscite $done_1$, $done_0$ non sono mai settate contemporaneamente. Se R_1 inizia un handshake, la RSS risponde portando $done_1$ ad 1, ma se dopo R_0 inizia a sua volta un handshake, la RSS non setta $done_0$ finché non ha resettato $done_1$. Viceversa, se R_0 inizia per prima. Se R_1 ed R_0 iniziano l'handshake contemporaneamente, la rete gestisce prima l'handshake con R_0 , poi quello con R_1 .

6.1.1 Soluzione

Si chiami S_R lo stato interno iniziale, ed S_i lo stato interno in cui si gestisce l'handshake con la rete R_i . La tabella di flusso della rete è la seguente:

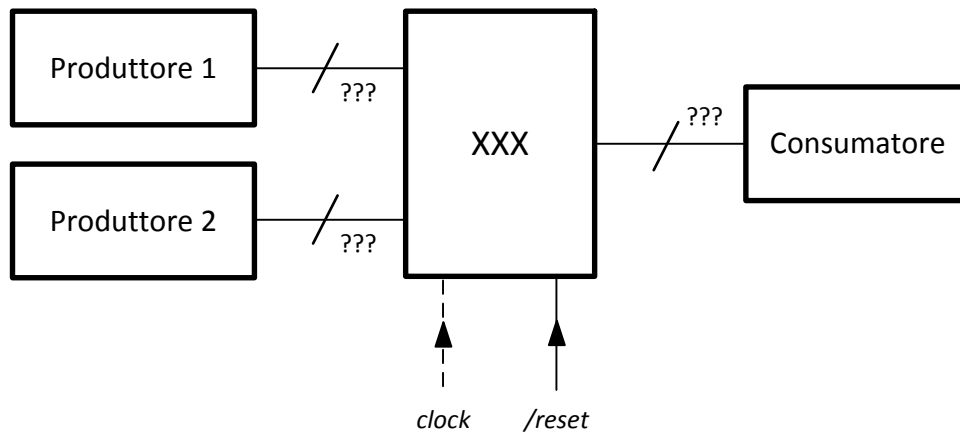
	go_1, go_0				$done_1, done_0$
	00	01	11	10	
SR	SR	S0	S0	S1	00
S0	SR	S0	S0	S1	01
S1	SR	S0	S1	S1	10

Scegliendo la codifica (ovvia) $S_R = 00$, $S_0 = 01$, $S_1 = 10$, la rete combinatoria che genera le uscite è un cortocircuito, $done_i = y_i$, e la rete combinatoria a porte NOR che genera gli ingressi ai DFF è la seguente:

$$a_1 = \overline{\overline{go_1} + (\overline{go_0} + y_1)}$$

$$a_0 = \overline{\overline{go_0} + (\overline{go_1} + y_1)}$$

6.2 Esercizio – descrizione e sintesi di RSS complessa

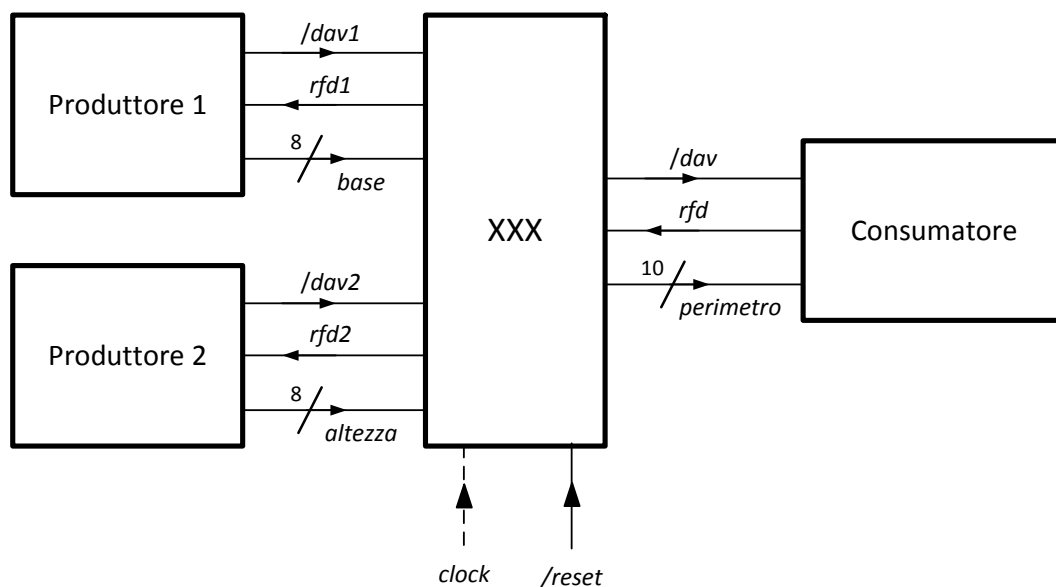


Descrivere la rete **XXX** che si evolve ciclicamente come segue: “preleva un byte dal Produttore 1 e un byte dal Produttore 2, elabora i byte ed invia il risultato della elaborazione al Consumatore.” L’elaborazione viene fatta tramite una funzione $mia_rete(base, altezza)$, che interpreta i byte ricevuti da XXX come numeri naturali in base 2 costituenti la base e l’altezza di un rettangolo e restituisce il perimetro del rettangolo. **Specificare in dettaglio** la struttura della rete combinatoria che implementa la funzione mia_rete di cui sopra.

NOTE: non è possibile fare nessuna ipotesi sulla velocità dei Produttori e del Consumatore

6.2.1 Descrizione

Le reti Produttore i e Consumatore sono asincrone rispetto alla rete XXX, quindi il colloquio deve essere protetto da **handshake** /dav-rfd. La somma dei due lati sta su 9 bit, quindi il perimetro sta su 10 bit. Detto questo, possiamo disegnare i collegamenti nel dettaglio.



La descrizione può essere affrontata **per approssimazioni successive**, mettendo a posto un po' di dettagli alla volta.

1) guardiamo quali **registri** servono, ed ipotizziamone un dimensionamento di massima.

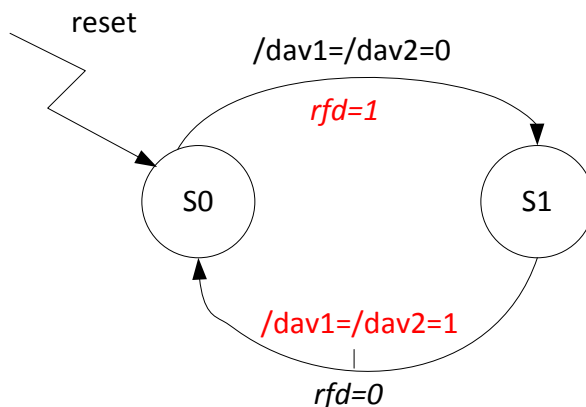
- Ne servono alcuni per sostenere le uscite: RFD1, RFD2, DAV_, a 1 bit ciascuno.
- PERIMETRO a 10 bit.
- STAR a un po' di bit (quanti saranno lo vediamo alla fine della descrizione)
- I valori *base* e *altezza* si prendono tramite handshake. Potrei volerli memorizzare da qualche parte (nel qual caso userei due registri BASE e ALTEZZA, a 8 bit ciascuno), ma probabilmente non sarà necessario (posso scrivere il risultato direttamente in PERIMETRO usando una rete combinatoria *mia_rete* che prende in input i valori *base* e *altezza*)

2) condizioni al **reset**:

Posso dare per scontato che tutti gli handshake siano a riposo, quindi che gli input siano inizialmente: $\text{/dav1}=1$, $\text{/dav2}=1$, $\text{rfd}=1$. **Devo settare gli output** di conseguenza: $\text{/dav}=1$, $\text{rfd1}=1$, $\text{rfd2}=1$.

Il valore iniziale di PERIMETRO non è significativo (tanto la sua validità è determinata dal fronte di discesa di /dav). Assumo che lo stato iniziale sia S0.

3) **diagramma a stati** (di massima) della rete:



La rete avrà un comportamento ciclico: quando avrà finito un'elaborazione tornerà in S0 per iniziare un'altra.

In **S0** devo:

- **tenere rfd1 , rfd2 , /dav a 1;**
- aspettare che **entrambi /dav1 e /dav2** siano andati a zero.

Infatti, non ha senso attendere **prima uno e poi l'altro**, passando da uno stato intermedio. Non ho nessun motivo per credere che uno sia più veloce dell'altro, né uno dei due dati che prelevo con l'handshake dipende dall'altro. Devo comunque aspettare entrambi.

In **S1** gli ingressi *base* e *altezza* sono corretti. Posso quindi calcolare $P = 2(B + A)$ (il conto lo faccio fare a una rete combinatoria), e:

- devo portare a zero RFD1 e RFD2 per far progredire l'handshake con i produttori.
- Devo assegnare PERIMETRO:

```
PERIMETRO<=mia_rete(base, altezza);
```

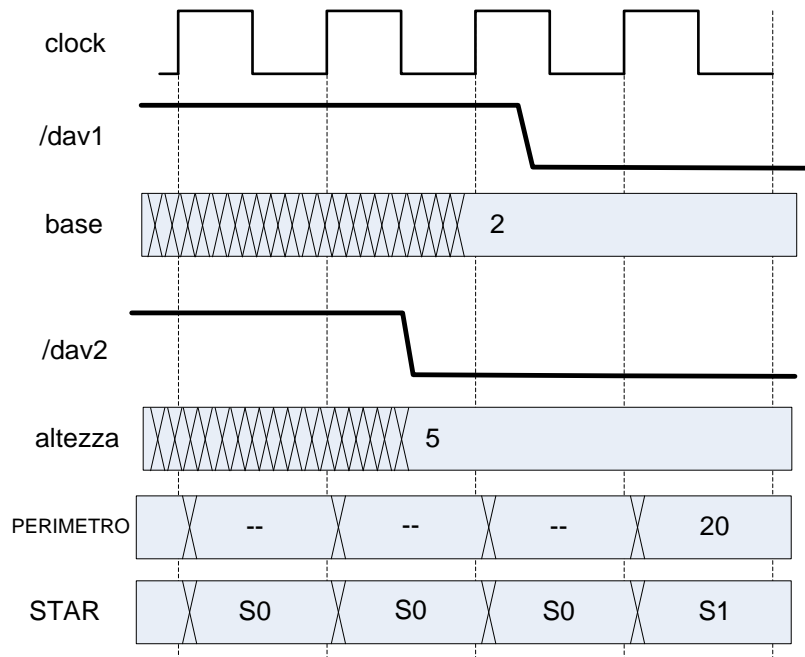
Posso contestualmente **portare DAV_ a 0**, perché il dato di uscita è pronto, in modo da iniziare l'handshake con il consumatore.

Potrei pensare di **ciclare** in S1 finché il consumatore non porta *rfd* a 0. Se le cose stanno come le ho scritte sopra **non lo posso fare**. Infatti, in S1 ho un assegnamento a PERIMETRO che dipende da **dei fili di input**. Ma in S1 ho portato a 0 RFD1 e RFD2, segnalando ai due produttori che quei dati sono **già stati prelevati**. Quando un produttore vede la transizione 1/0 del proprio *rfd*, **non ha più l'obbligo di mantenere il dato in uscita**. Quindi, se ciclo in S1, ad ogni iterazione verrà rinnovato l'assegnamento, ma il valore degli ingressi è garantito essere corretto **soltanto alla prima iterazione**. Dalla seconda iterazione in poi gli input su cui calcolo il perimetro possono essere **non significativi**.

Visto che comunque ho bisogno di attendere che *rfd* del consumatore vada a 0, come risolvo la situazione? O uso un altro stato, oppure (meglio) **porto in S0 l'assegnamento**:

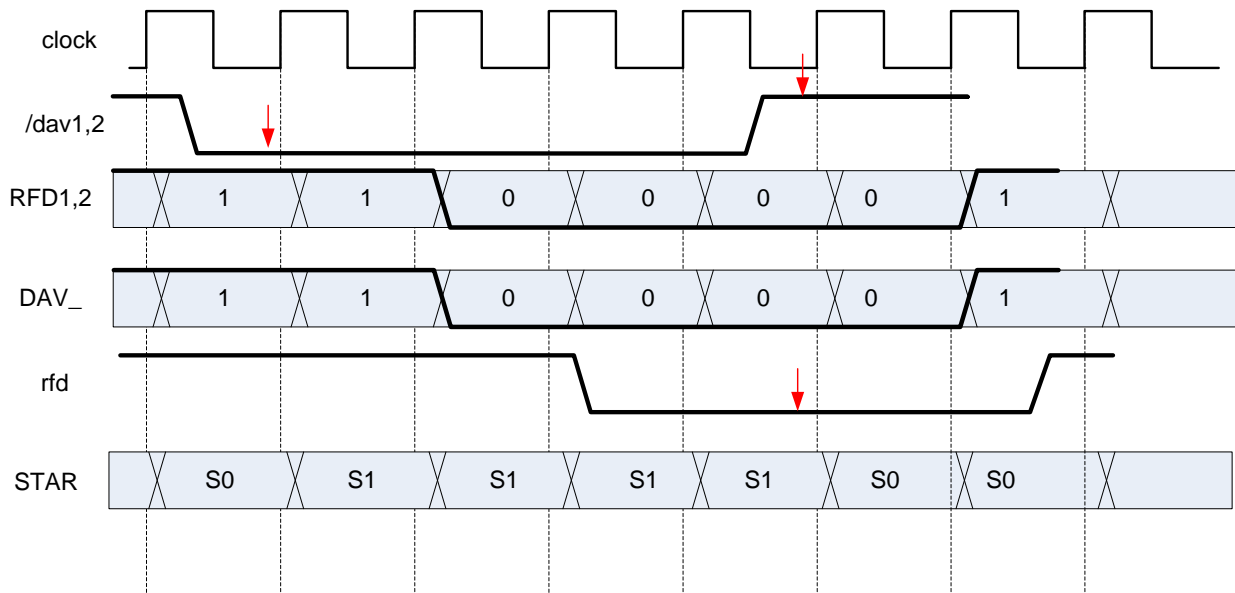
```
PERIMETRO<=mia_rete(base, altezza);
```

In questo modo in S0 ciclo assegnando a PERIMETRO valori a caso (finché almeno uno dei due */dav* vale 1). Però esco da S0 quando **entrambi i /dav** sono a 0, e quindi l'ultimo assegnamento è quello corretto. Il contenuto del registro PERIMETRO balla, ma non è un problema, perché l'output *perimetro* è soggetto all'handshake, e non verrà letto dal consumatore prima che *dav_* vada a 0.



A questo punto in S1 posso gestire l'handshake con il consumatore. Tiro giù DAV_, attendo che *rfd* sia andato a 0 e vado dove? Non posso saltare indietro in S0, a meno che non mi sia assicurato **anche** che */dav1* */dav2* siano tornati a 1 (cioè che si sia chiuso l'handshake con i due produttori. Se tornassi in S0 senza testare, finirei per rischiare di violare l'handshake.

Guardiamo meglio i tre handshake:



Quindi la condizione per poter tornare in S0 è che:

- *rfd*=0
- */dav1* e */dav2* sono **entrambi a 1** (infatti in S0 metto *rfd1* e *rfd2* ad 1, chiudendo l'handshake)

Però si vede bene che, in questo modo, **non si controlla mai** che *rfd* sia tornato a 1. Pertanto, ad un nuovo ciclo di esecuzione, non potrei essere sicuro che quando metto *dav* a 0 in S1 l'handshake con il consumatore si è svolto correttamente.

Si rimedia testando che ***rfd* sia tornato a 1 in S0**. La condizione per uscire da S0 va modificata, aggiungendo che *rfd* deve essere uguale a 1.

La descrizione è quindi la seguente:

```
module XXX (base,dav1_,rfd1,altezza,dav2_,rfd2,perimetro,dav_,rfd,
            clock,reset_);
  input      clock,reset_;
  input      dav1_, dav2_, rfd;
  output     rfd1, rfd2, dav_;
  input [7:0] base, altezza;
  output [9:0] perimetro;

  reg        RFD1, RFD2, DAV_;
  reg [9:0] PERIMETRO;
  reg STAR;
  parameter S0=0,S1=1;

  assign rfd1=RFD1;
  assign rfd2=RFD2;
  assign dav_=DAV_;
  assign perimetro=PERIMETRO;

  always @(reset_==0) #1 begin STAR<=S0; RFD1<=1; RFD2<=1; DAV_<=1; end
  always @(posedge clock) if (reset_==1) #3
    casex (STAR)
      S0: begin RFD1<=1; RFD2<=1; DAV_<=1; PERIMETRO<=mia_rete(base,altezza);
              STAR<=({dav1_,dav2_,rfd}=='B001)?S1:S0; end

      S1: begin RFD1<=0; RFD2<=0; DAV_<=0;
              STAR<=({dav1_,dav2_,rfd}=='B110)?S0:S1; end
    endcase

  function [9:0] mia_rete;
    input [7:0] base, altezza;
    mia_rete = {(1'b0,base)+(1'b0,altezza)},1'b0;
  endfunction

endmodule
```

Guardando la descrizione ci si rende subito conto che RFD1, RFD2 e DAV_ sono lo stesso registro, e quindi ne basta uno solo, chiamiamolo HS (handshake). La descrizione ottimizzata è:

```
[...]
reg        HS;
[...]
```

```
assign rfd1=HS;
assign rfd2=HS;
```

```

assign    dav_ = HS;

[...]

always @(reset_ == 0) #1 begin STAR = S0; HS <= 1; end

case (STAR)
S0: begin HS <= 1; PERIMETRO <= mia_rete(base, altezza);
        STAR <= ({dav1_, dav2_, rfd} == 'B001') ? S1 : S0; end

S1: begin HS <= 0; STAR <= ({dav1_, dav2_, rfd} == 'B110') ? S0 : S1; end

endcase

```

6.2.2 Sintesi

La sintesi è banale: andiamo per ordine:

Registro operativo HS

<pre>S0: HS<=1; S1: HS<=0;</pre>	<p>Una variabile di comando b0, che vale 1 in S0 e 0 in S1.</p> <pre>always @(posedge clock) if (reset_==1) #3 HS<=b0;</pre>
--	---

Registro operativo PERIMETRO

```
S0: PERIMETRO<=mia_rete(base,altezza);
S1: PERIMETRO<=PERIMETRO;
```

Questo è un registro multifunzionale a due vie. Basta una variabile di comando b0, che vale 1 in S0 e 0 in S1.

```
always @(posedge clock) if (reset_==1) #3
  casex(b0)
    `B1: PERIMETRO<=mia_rete(base,altezza);
    `B0: PERIMETRO<=PERIMETRO;
  endcase
```

Registro di stato STAR

```
S0: STAR<=({dav1_,dav2_,rfd}=='B001)?S1:S0;
S1: STAR<=({dav1_,dav2_,rfd}=='B110)?S0:S1;
```

Ci sono due condizioni indipendenti, quindi servono **due variabili di condizionamento** c_0 e c_1 :

```
c0=({dav1_,dav2_,rfd}=='B001')?1:0;
c1=({dav1_,dav2_,rfd}=='B110')?1:0;
```

Abbiamo tutto per poter scrivere la sintesi secondo il modello parte operativa / parte controllo.

```

module XXX (base,dav1_,rfd1,altezza,dav2_,rfd2,perimetro,dav_,rfd.
            clock, reset_);

    input      clock,reset_;
    input      dav1_, dav2_, rfd;
    output     rfd1, rfd2, dav ;

```

```

input    [7:0] base, altezza;
output   [9:0] perimetro;

wire c1,c0,b0;
Parte_Operativa PO(base,dav1_,rfd1,altezza,dav2_,rfd2,
                    perimetro,dav_,rfd, c1,c0,b0,clock,reset_);
Parte_Controllo PC(b0,c1,c0,clock,reset_);
endmodule

//-----
module Parte_Operativa(base,dav1_,rfd1,altezza,dav2_,rfd2,
                    perimetro,dav_,rfd, c1,c0,b0,clock,reset_);
    input          clock,reset_;
    input          dav1_, dav2_, rfd;
    output         rfd1, rfd2, dav_;
    input   [7:0] base, altezza;
    output   [9:0] perimetro;

    input b0;
    output c1,c0;

    reg          HS;
    reg   [9:0] PERIMETRO;

    assign c0=({dav1_,dav2_,rfd}=='B001)?1:0;
    assign c1=({dav1_,dav2_,rfd}=='B110)?1:0;

//Registro HS
    always @(reset_==0) #1 HS<=1;
    always @(posedge clock) if (reset_==1) #3 HS<=b0;

//Registro PERIMETRO
    always @(posedge clock) if (reset_==1) #3
        casex(b0)
            'B1: PERIMETRO<=mia_rete(base,altezza);
            'B0: PERIMETRO<=PERIMETRO;
        endcase
endmodule

module Parte_Controllo(b0,c1,c0,clock,reset_);
    input clock,reset_;
    input c1,c0;
    output b0;
    reg STAR; parameter S0='B0,S1='B1;
    assign b0=(STAR==S0)?'B1:'B0;

    always @(reset_==0) #1 STAR<=S0;
    always @(posedge clock) if (reset_==1) #3
        casex(STAR)
            S0: STAR<=(c0==1)?S1:S0;
            S1: STAR<=(c1==1)?S0:S1;
        endcase
endmodule

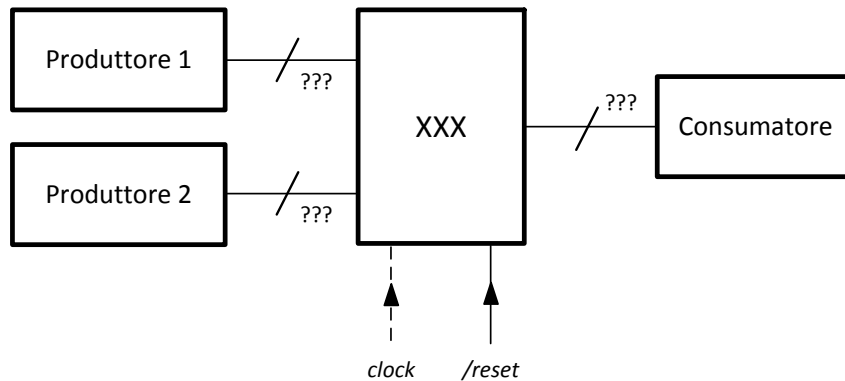
```

Qualora si voglia vedere la parte controllo come ROM-based (micro-address-based o micro-instruction based), abbiamo:

$\mu\text{-addr}$	$\mu\text{-code}$	C_{eff}	$\mu\text{-addr T}$	$\mu\text{-addr F}$
	b_0			
0 (S0)	1	0	1 (S1)	0 (S0)
1 (S1)	0	1	0 (S0)	1 (S1)

Si osservi che la parte controllo è di fatto un FF-JK, con $c_0=j$, $c_1=k$, $b_0=\sim q$.

6.3 Esercizio – Calcolo del prodotto con algoritmo di somma e shift



Descrivere il circuito **XXX** che si evolve ciclicamente come segue: “preleva un byte dal Produttore 1 e un byte dal Produttore 2, elabora i byte ed invia il risultato della elaborazione al Consumatore.” L’elaborazione consiste nell’interpretare i byte ricevuti da XXX come numeri naturali in base 2 costituenti la base e l’altezza di un rettangolo e calcolare l’**area** del rettangolo, *usando l’algoritmo di somma e shift*:

Dati X, C numeri naturali in base β su n cifre, Y numero naturale in base β su m cifre, l’algoritmo calcola $P = X \cdot Y + C$ come segue:

$$P_0 = C \cdot \beta^m,$$

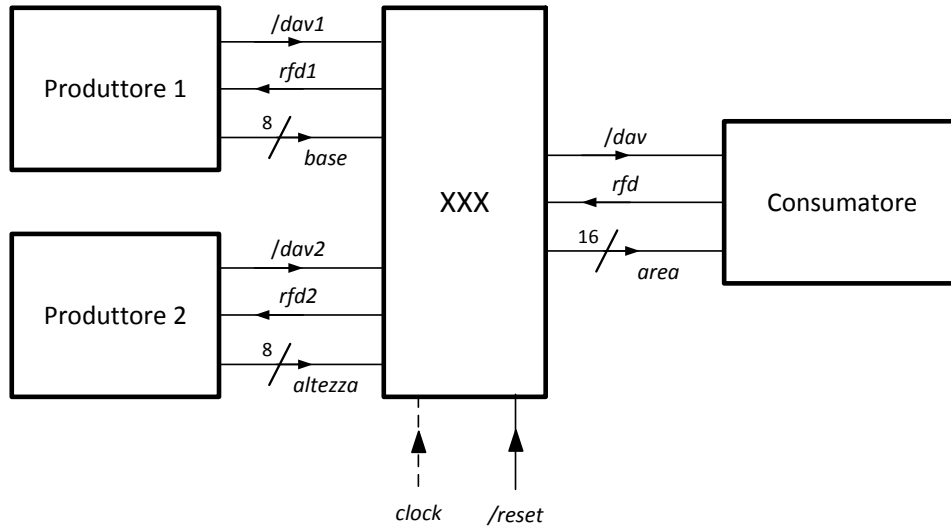
$$P_{i+1} = \left\lfloor \frac{y_i \cdot \beta^m \cdot X + P_i}{\beta} \right\rfloor$$

$$P_m = P$$

NOTE: non è possibile fare nessuna ipotesi sulla velocità dei Produttori e del Consumatore

6.3.1 Descrizione

I collegamenti della rete sono identici al caso precedente, salvo che adesso l’uscita si chiama *area* ed è su 16 bit.



Per far girare l'algoritmo, ho $C = 0$, quindi $P_0 = 0$. Definirò una rete combinatoria `mia_rete` che sintetizza il passo iterativo dell'algoritmo.

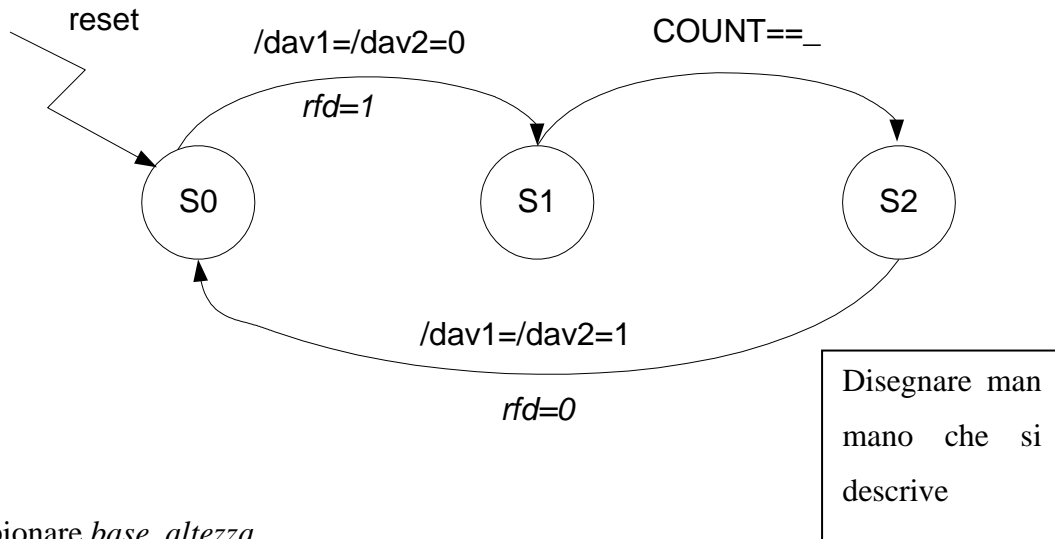
1) guardiamo i **registri**, con un dimensionamento di massima.

- Ne servono alcuni per sostenere le uscite: RFD1, RFD2, DAV_, a 1 bit ciascuno. Non è improbabile che possa compattarli come nel caso precedente.
- AREA a 16 bit
- STAR a un po' di bit (quanti saranno lo vediamo alla fine della descrizione)
- Ne servirà qualcuno **per fare i conti**. Devo calcolare qualcosa attraverso un algoritmo **iterativo**, e quindi userò un registro COUNT come "variabile di conteggio" per tener traccia del numero di iterazioni. A quanti bit? Devo fare 8 iterazioni, quindi ci vorranno 3 o 4 bit.
- I valori *base* e *altezza* li prendo con un handshake. Converrà memorizzarli da qualche parte. Uso due registri BASE e ALTEZZA, a 8 bit ciascuno.

2) condizioni al **reset**:

Tutti gli handshake sono a riposo, quindi posso dare per scontato che $/dav1=1$, $/dav2=1$, $rfd=1$ (input), e **devo fare in modo che** gli output siano consistenti: $/dav=1$, $rfd1=1$, $rfd2=1$.

3) **diagramma a stati** (di massima) della rete



In **S0** devo:

- campionare *base*, *altezza*
- tenere *rfd1*, *rfd2*, */dav* a 1
- aspettare che */dav1* e */dav2* siano andati a zero. Con l'esperienza dell'esercizio precedente, possiamo dire fin d'ora che ci vorrà una terza condizione, cioè *rfd=1*, per gestire la chiusura dell'handshake con il consumatore.

In **S1** comincio il **calcolo iterativo del prodotto**. Facciamo che AREA contiene, ad ogni clock, P_i .

Devo quindi:

- inizializzare COUNT e AREA, e lo devo fare in S0, perché in S1 li sto usando. AREA lo inizializzo a zero, e COUNT, **in prima battuta**, lo inizializzo a 8.
- devo portare a zero RFD1 e RFD2 (**e non DAV_**, perché il dato non è ancora stato calcolato)
- devo implementare la formula scritta sopra. Mi serve una **rete combinatoria**, che abbia in ingresso: a) BASE, b) il vecchio valore di AREA, e c) **un bit di altezza**. Il bit che mi serve cambia da un ciclo all'altro. Tutte le volte che questo succede, la tecnica **standard** da usare è la seguente: metto il valore da usare in un registro, utilizzo il bit 0 del registro, e **shifto a destra il contenuto del registro ad ogni clock** (visto che mi servono, nell'ordine, i bit dal meno al più significativo. Se fosse stato il contrario avrei preso il bit 7 e shiftato a sinistra).

Quindi, in S1, devo decrementare COUNT, shiftare ALTEZZA, e assegnare AREA:

```

AREA<=mia_rete(BASE, ALTEZZA[0], AREA);
dec COUNT;
shr ALTEZZA;
    
```

Devo infine **saltare altrove, quando** ho finito le iterazioni del ciclo che dovevo fare. Scrivo la condizione in modo generico, come

```
STAR<=(COUNT==_) ? S2 : S1;
```

poi la specifico meglio in fondo.

Vado in uno stato **S2**: il dato è pronto, e devo gestire l'handshake con il consumatore. Tiro giù DAV_, attendo che *rfd* sia andato a 0 e vado dove? Posso saltare indietro in S0, se mi assicuro **anche** che */dav1 /dav2* siano tornati a 1.

Ma a questo punto bisogna che, tra S0 e S2, testi anche che **rfd sia tornato a 1**. Lo posso fare soltanto in S0, e quindi la condizione per uscire da S0 va modificata, aggiungendo che *rfd* deve essere uguale a 1.

Mancano da chiarire la condizione per uscire da S1 e la rete combinatoria. Quando si hanno **cicli di decremento e test** (come in questo caso) la regola è la seguente:

se inizializzo a k e testo a j ($\leq k$), il numero di iterazioni è $k - j + 1$.

Quindi, se inizializzo COUNT a 8, lo devo testare ad 1 per avere 8 iterazioni. Allora conviene inizializzarlo a 7 e testarlo a 0, così tra l'altro posso dimensionare il registro su 3 bit invece che 4.

Per quanto riguarda la **rete combinatoria mia_rete**, basta seguire la formula. Al numeratore:

- se $y_i = 0$, ho direttamente P_i
- se $y_i = 1$, ho una somma di due addendi, entrambi a 16 bit. La somma sta quindi su 17 bit.

In uscita, devo buttare il bit meno significativo (divisione per beta). Quindi serve un sommatore a 17 bit, un multiplexer e devo strigare un po' di fili.

In realtà la rete che fa la somma può essere semplificata, perché si vede subito che gli 8 bit più bassi sono $P_i[7:0]$, in quanto sono sommati a zeri. Quindi non c'è bisogno di un sommatore a 17 bit, ma ne basta uno a 9.

La descrizione è la seguente:

```

module XXX (base,dav1_,rfd1,altezza,dav2_,rfd2,area,dav_,rfd,
              clock,reset_);

  input      clock,reset_;
  input      dav1_, dav2_, rfd;
  output     rfd1, rfd2, dav_;
  input      [7:0] base, altezza;
  output     [15:0] area;

  reg        RFD1, RFD2, DAV_;
  reg        [15:0] AREA;
  reg        [7:0] BASE, ALTEZZA;
  reg        [3:0] COUNT;           // non posso sapere subito quanti bit
  reg        [1:0] STAR;            // non posso sapere subito quanti bit
  parameter S0='B00,S1='B01,S2='B10;

  assign     rfd1=RFD1;
  assign     rfd2=RFD2;
  assign     dav_=DAV_;
  assign     area=AREA;

```

```

always @(reset_==0) #1 begin STAR<=S0; RFD1<=1; RFD2<=1; DAV_<=1; end
always @(posedge clock) if (reset_==1) #3
  casex (STAR)
    S0: begin RFD1<=1; RFD2<=1; DAV_<=1; BASE<=base; ALTEZZA<=altezza;
              AREA<=0; COUNT<=7; STAR<=({dav1_,dav2_,rfd}=='B001)?S1:S0;
            end
    S1: begin RFD1<=0; RFD2<=0; COUNT<=COUNT-1;
              ALTEZZA<={1'B0, ALTEZZA[7:1]};
              AREA<=mia_rete(BASE, ALTEZZA[0], AREA);
              STAR<=(COUNT==0)?S2:S1; end
    S2: begin DAV_<=0; STAR<=({dav1_,dav2_,rfd}=='B110)?S0:S2; end
  endcase
function [15:0] mia_rete;
  input [7:0] x;
  input y_i;
  input [15:0] p_i;
  casex (y_i)
    'B0: mia_rete={1'B0, p_i[15:1]};
    'B1: mia_rete={1'B0, p_i[15:1]}+{1'B0, x, 7'B0000000};
  endcase
endfunction
endmodule

```

Un registro non è un vettore.
Non lo posso indicizzare con
una *variabile* in Verilog. Non
posso scrivere
ALTEZZA[COUNT]

Possibili migliorie rispetto alla descrizione sopra riportata:

- RFD1, RFD2 possono andare a 0 direttamente in S2. In tal caso, RFD1, RFD2 e DAV_ contengono sempre lo stesso bit in ogni stato, quindi posso usare un solo registro HS come nel precedente esercizio. Peraltro, se faccio così, **non ho più bisogno del registro BASE**, e posso mandare in input a mia_rete direttamente gli ingressi *base*, che in S1 sono tenuti stabili dal produttore1 (dato che */dav1=0, rfd1=1*). L'ingresso *altezza* va invece salvato in un registro in ogni caso, perché devo usarne un bit alla volta.
- Volendo risparmiare qualcosa, posso inserire l'inizializzazione di COUNT direttamente nella fase di reset. Infatti, quando si esce dal ciclo in S1 COUNT vale 7, cioè il valore che gli sarebbe stato assegnato al successivo passaggio in S0. Pertanto non c'è bisogno di iniziarlo in S0 esplicitamente.

La descrizione con le modifiche di cui sopra è la seguente:

```

[...]  
reg          HS;  
reg [15:0] AREA;  
reg [7:0] ALTEZZA;  
reg [2:0] COUNT;  
reg [1:0] STAR;  
  
assign rfd1=HS;

```

```

assign    rfd2=HS;
assign    dav_=HS;
[...]
```

```

always @(reset_==0) #1 begin STAR<=S0; HS<=1; COUNT<=7; end
always @(posedge clock) if (reset_==1) #3
  case (STAR)
    S0: begin HS<=1; ALTEZZA<=altezza;
        AREA<=0; STAR<=({dav1_,dav2_,rfd}=='B001)?S1:S0; end

    S1: begin COUNT<=COUNT-1;
        ALTEZZA<={1'B0, ALTEZZA[7:1]};
        AREA<=mia_rete(base, ALTEZZA[0], AREA);
        STAR<=(COUNT==0)?S2:S1; end

    S2: begin HS<=0;
        STAR<=({dav1_,dav2_,rfd}=='B110)?S0:S2; end
  endcase
endmodule
```

6.3.2 Sintesi

Partiamo dai registri operativi:

Registro operativo HS

```

S0: HS<=1;
S1: HS<=HS;
S2: HS<=0;
```

Registro operativo ALTEZZA

```

S0: ALTEZZA<=altezza;
S1: ALTEZZA<={1'B0, ALTEZZA[7:1]};
S2: ALTEZZA<=ALTEZZA;
```

Registro operativo AREA

```

S0: AREA<=0;
S1: AREA<=mia_rete(base, ALTEZZA[0], AREA);
S2: AREA<=AREA;
```

Registro operativo COUNT

```

S0, S2: COUNT<=COUNT;
S1: COUNT<=COUNT-1;
```

Tre dei registri operativi sono registri multifunzionali a tre funzioni con un multiplexer a 3 vie, comandato da due variabili di comando. Pertanto, sono necessarie due variabili di comando, b0, b1, che posso assegnare in questo modo:

```

S0: b1b0=00;
S1: b1b0=01;
S2: b1b0=10;
```

In tal modo, posso usare solo b0 per comandare il multiplexer di COUNT, che è a due vie.

Registro di stato STAR

```

S0: STAR<=({dav1_,dav2_,rfd}=='B001)?S1:S0;
S1: STAR<=(COUNT==0)?S2:S1;
S2: STAR<=({dav1_,dav2_,rfd}=='B110)?S0:S2;
```

Ci sono tre condizioni indipendenti, quindi servono tre variabili di condizionamento:

```

c0=({dav1_,dav2_,rfd}=='B001)?1:0
```

```
c1=(COUNT==0)?1:0
c2=({dav1_,dav2_,rfd}=='B110)?1:0
```

Quindi avremo per la parte operativa (saltando un po' di sintassi):

```
[...]
input b1,b0;
output c2,c1,c0;

assign c0=({dav1_,dav2_,rfd}=='B001)?1:0;
assign c1=(COUNT==0)?1:0;
assign c2=({dav1_,dav2_,rfd}=='B110)?1:0;

//Registro HS
always @(reset_==0) #1 HS<=1;
always @(posedge clock) if (reset_==1) #3
  case ({b1,b0})
    2'B00: HS<=1;
    2'B01: HS<=HS;
    2'B10: HS<=0;
  endcase

[...]
```

```
//Registro COUNT
always @(reset_==0) #1 COUNT<=7;
always @(posedge clock) if (reset_==1) #3
  case (b0)
    'B0: COUNT<=COUNT;
    'B1: COUNT<=COUNT-1;
  endcase
```

Per la parte controllo abbiamo (sempre saltando un po' di sintassi):

```
module Parte_Controllo(b0,c1,c0,clock,reset_);

  [...]
  input c2,c1,c0;
  output b1,b0;

  reg STAR; parameter S0='B00, S1='B01, S2='B10;

  assign {b1,b0}= (STAR==S0)?'B00:
                  (STAR==S1)?'B01:
                  /*(STAR==S2)?'B10;

  always @(reset_==0) #1 STAR<=S0;
  always @(posedge clock) if (reset_==1) #3
    case (STAR)
      S0: STAR<=(c0==1)?S1:S0;
      S1: STAR<=(c1==1)?S2:S1;
      S2: STAR<=(c2==1)?S0:S2;
    endcase
endmodule
```

Qualora si voglia vedere la parte controllo come ROM-based (micro-address-based o micro-instruction based), abbiamo:

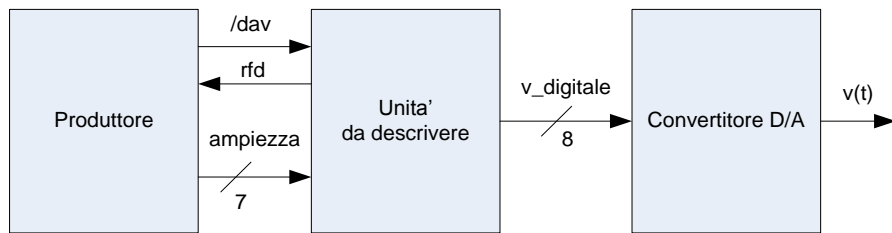
$\mu\text{-addr}$	$\mu\text{-code}$ b_1b_0	C_{eff}	$\mu\text{-addr T}$	$\mu\text{-addr F}$
00 (S0)	00	00	01 (S1)	00 (S0)
01 (S1)	01	01	10 (S2)	01 (S1)
10 (S2)	10	10	00 (S0)	10 (S2)

6.4 Esercizio – tensioni analogiche

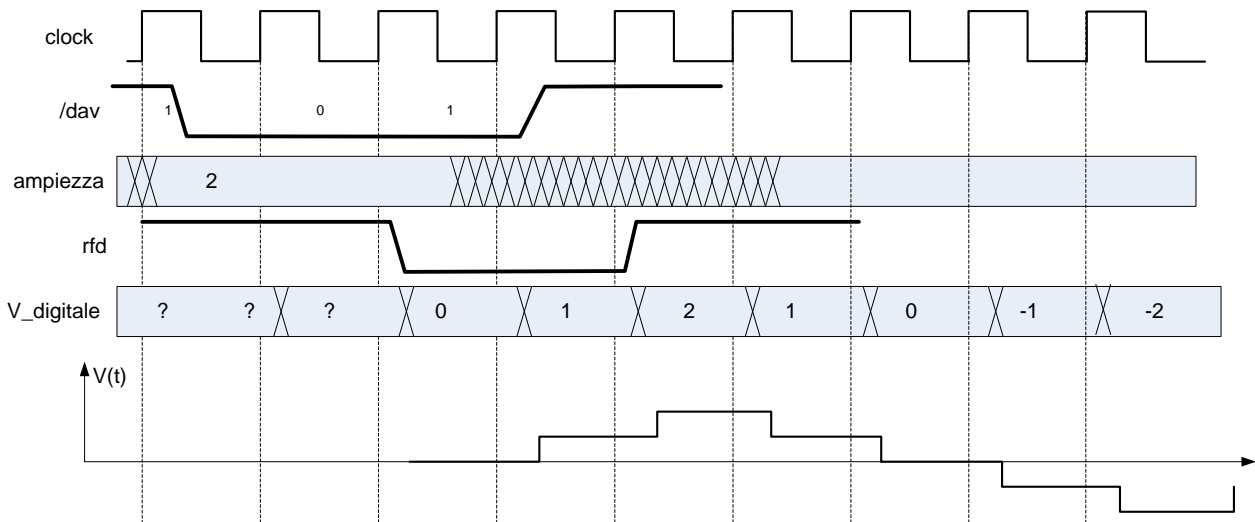
Descrivere l'unità di figura che opera ciclicamente nelle seguenti ipotesi:

- l'unità gestisce con il *produttore* un handshake classico, con passaggio di dati a **7 bit** (e non 8)
- l'unità interpreta il dato fornito dal produttore tramite la variabile *ampiezza* come un numero naturale N in base 2, e presenta la variabile di uscita $v_{digitale}$ in ingresso ad un convertitore D/A. Tale variabile costituirà una sequenza di byte (al ritmo di uno per clock), che il convertitore interpreta in modo tale da produrre in uscita una tensione con forma d'onda triangolare di ampiezza pari ad N

Si faccia riferimento ad un convertitore che opera secondo la legge *binaria bipolare*. Si assuma che N sia diverso da zero.



Ad esempio:



6.4.1 Descrizione

L'esercizio parla di **binario bipolare**: i numeri da mandare al convertitore D/A sono **interi rappresentati in traslazione**. Devo poter mandare tutti i numeri compresi in $k \in [-N; +N]$, dove $0 < N \leq 2^7 - 1$. La rappresentazione in traslazione di un numero intero k su 8 bit è $K = k + 2^{8-1}$. Pertanto:

- $N = 1$: devo poter inviare i numeri $k = -1, 0, 1$, cioè $K = 2^7 + \{-1, 0, 1\}$;
- $N = 2$: devo poter inviare i numeri $k = +2, -1, 0, 1, +2$, cioè $K = 2^7 + \{-2, -1, 0, 1, +2\}$;
- $N = 2^7 - 1$: devo poter inviare i numeri $k = -(2^7 - 1), \dots, -1, 0, 1, \dots, +(2^7 - 1)$, cioè $K = 2^7 + \{-(2^7 - 1), \dots, -1, 0, 1, \dots, +(2^7 - 1)\}$.

Quindi, nel caso peggiore devo inviare numeri naturali K compresi tra 1 e $2^8 - 1$, il che va bene perché ho otto bit a disposizione per l'uscita.

Per poter generare le tensioni digitali, dovrò scrivere **tre cicli**:

- Il primo ciclo, facendo uscire i numeri da 2^{8-1} a $2^{8-1} + N$ (a salire);
- Il secondo ciclo sarà a scendere da $2^{8-1} + N$ a $2^{8-1} - N$;
- Il terzo ciclo sarà a salire da $2^{8-1} - N$ a 2^{8-1} .

Il tutto facendo caso a tenere l'uscita ai valori estremi per un solo clock.

Oltre a questo devo gestire l'handshake con il produttore. Dovrò campionare l'ampiezza sul fronte di discesa di $/dav$, mettere rfd a zero, **e** dovrò attendere che $/dav$ sia tornato a uno prima di ricominciare. Si faccia caso al fatto che **non è scritto da nessuna parte** che una nuova iterazione del lavoro della rete deve iniziare **immediatamente** quando è finito il precedente. Peraltro, potrebbe non essere possibile se il produttore è lento a riportare $/dav$ ad 1.

Detto questo, cerchiamo di capire di quali **registri** posso aver bisogno:

- Un registro OUT per sostenere l'uscita $v_{digitale}$, che dovrà essere ad 8 bit.
- Un registro RFD per sostenere l'uscita omonima (ad 1 bit)
- Visto che devo confrontare il valore di OUT con delle costanti che dipendono dall'ampiezza campionata, potrebbe farmi comodo memorizzare queste costanti in due registri MAX e MIN, entrambi a 8 bit.
- Un registro STAR, da dimensionare opportunamente alla fine della descrizione.

Per quanto riguarda le **condizioni al reset**:

- Potrò dare per scontato che $/dav=1$
- Dovrò fare in modo che $RFD=1$, e che $v_{digitale}=2^{8-1}$, quindi $OUT=2^{8-1}$.

Conviene definire una *costante*:

```
parameter zero='H80;
```

In modo da rendere la descrizione più leggibile.

Descriviamo ora cosa dovrebbe succedere nei vari stati:

S0: si arriva qui dal reset, e si dovrà campionare *ampiezza* finché $/dav$ non va a zero. Teniamo $RFD=1$, $OUT=zero$, e possiamo assegnare $MAX=zero + ampiezza$, $MIN=zero - ampiezza$. Si va in S1 quando $/dav$ va a zero.

S1: si deve portare RFD a zero per far proseguire l'handshake, e poi bisogna eseguire il primo ciclo: si incrementa OUT , e si testa se OUT è arrivato a MAX . Quando questo succede si va in S2.

S2: eseguire il secondo ciclo: si decrementa OUT , e si testa se OUT è arrivato a MIN . Quando questo succede si va in S3.

S3: eseguire il terzo ciclo: si incrementa OUT , e si testa se OUT è arrivato a $zero$.

S4: si finisce di gestire l'handshake (si testa se $/dav=1$) e si torna in S0.

Quindi, a posteriori, posso dire che servono cinque stati interni, quindi $STAR$ deve essere di 3 bit.

Come si fa a testare se OUT ha raggiunto il valore necessario (e.g., MAX)?

- Se lo sto **incrementando**, dovrò scrivere

```
Sx: begin ... OUT<=OUT+1; STAR<=(OUT==MAX-1)?Sy:Sx; ... end
```

- Se lo sto **decrementando**, dovrò scrivere

```
Sx: begin ... OUT<=OUT-1; STAR<=(OUT==MIN+1)?Sy:Sx; ... end
```

In quanto la condizione viene testata sul **vecchio** valore di OUT , quello **prima del** clock. All'arrivo del clock, OUT verrà incrementato o decrementato ancora una volta.

```
module XXX (out, dav_, rfd, ampiezza, clock, reset_);
  input      clock, reset_;
  input  [6:0] ampiezza;
  input      dav_;
  output      rfd;
  output  [7:0] out;

  reg      RFD;
  reg  [7:0] OUT, MAX, MIN;
  reg  [2:0] STAR;
  parameter S0=0, S1=1, S2=2, S3=3, S4=4, zero='H80; // zero=128
```

```

assign    rfd=RFD;
assign    out=OUT;

always @(reset_==0) #1 begin RFD<=1; OUT<=zero; STAR<=S0; end
always @(posedge clock) if (reset_==1) #3
  casex(STAR)
    S0: begin RFD<=1; OUT<=zero; MAX<=zero+{'B0,ampiezza};
           MIN<=zero-{'B0,ampiezza}; STAR<=(dav_==0)?S1:S0; end
    S1: begin RFD<=0; OUT<=OUT+1; STAR<=(OUT==MAX-1)?S2:S1; end
    S2: begin OUT<=OUT-1; STAR<=(OUT==MIN+1)?S3:S2; end
    S3: begin OUT<=OUT+1; STAR<=(OUT==zero-1)?S4:S3; end
    S4: begin STAR<=(dav_==0)?S4:S0; end
  endcase
endmodule

```

Possibili modifiche:

se si evita di porre RFD a 0 in S1, e lo si lascia quindi a 1 fino a S3 (o S4), si può usare l'ingresso *ampiezza* per fare i conti. In questo caso, non c'è più bisogno dei registri MAX e MIN, in quanto si può usare direttamente il valore *ampiezza* in S1 ed S2. Posso riscrivere quindi le condizioni in S1 e S2 come:

```
STAR<=(OUT==zero+ampiezza-1)?...
```

```
STAR<=(OUT==zero-ampiezza+1)?...
```