Reti Sequenziali Sincronizzate

Giovanni Stea

a.a. 2017/18

Ultima modifica: 15/11/2017

Sommario

1 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 <u>clock</u>**. 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**.

1.1 Registri

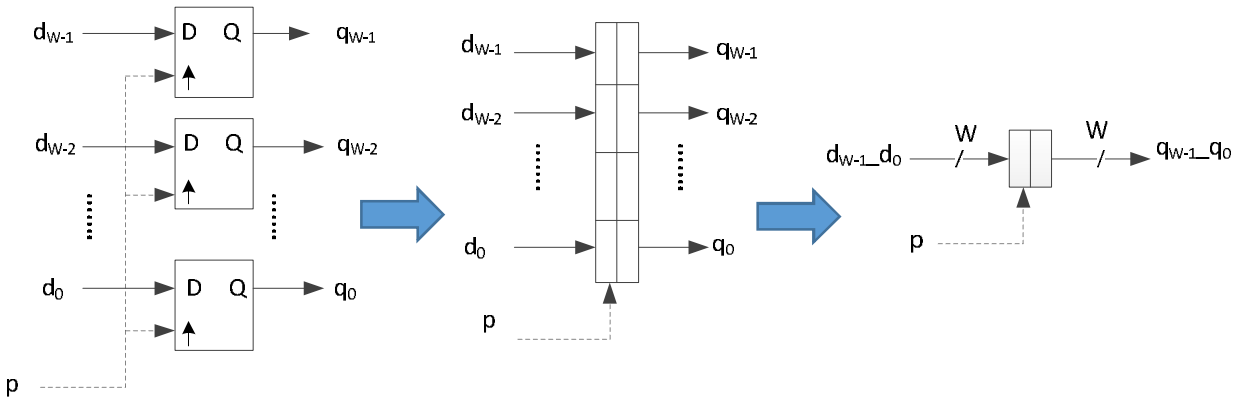
Definisco un **registro a *W* bit** come una **collezione di *W* D-Flip-Flop Positive-Edge-Triggered**, che hanno:

a) ingressi d_i ed uscite q_i separati (cioè indipendenti)

b) **ingresso *p* <u>a comune</u>**

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.

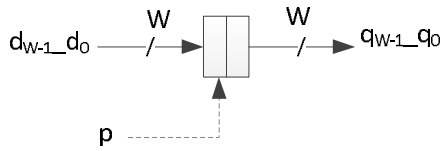
È fondamentale capire **bene** che **i registri memorizzano il proprio stato di ingresso al fronte di salita del clock**. Il fatto che **due stati di ingresso ai registri**, presentati su istanti di clock (fronti di salita) **consecutivi**, siano **identici, adiacenti o non adiacenti non riveste alcuna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

1.1.1 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:




```
// Dichiarazione di un registro da W bit di tipo reg
// delle variabili clock e reset_ da usarsi
// per l'impostazione dello stato interno iniziale.
// Dichiarazione di due variabili a W bit, dW-1_d0 e qW-1_q0 da usarsi,
// rispettivamente, come variabile di ingresso e come varia
// e impostazione di quest'ultima come effettiva variabile
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;

// Immissione nel registro del contenuto_iniziale al reset_
// della variabile dW-1_d0 all'arrivo di ogni segnale di sincr
always @(reset_==0) #1 REGISTRO<=contenuto_iniziale;
always @(posedge clock) if (reset_==1) #Tpropagation REGISTRO<=dW-1_d0;
```

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

Dichiarazione registri (**reg**) e fili (**wire**)

Blocco **assign** – **assegnamento continuo** (aggiornamento uscite)

Blocco **always** – **assegnamento procedurale** (scrittura registri)

@: Controllo degli eventi

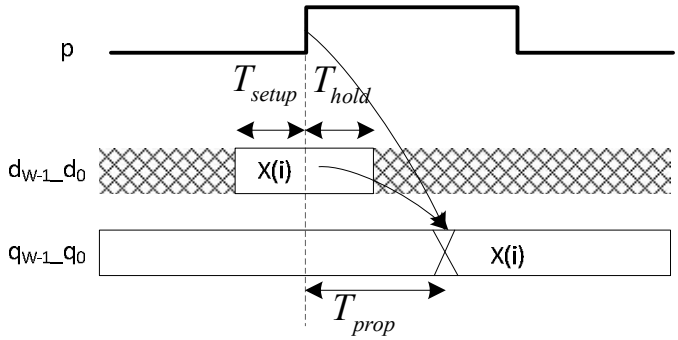
Tempo di propagazione. D'ora in avanti ci metteremo "3". Conviene che sia >0, perché sennò nelle simulazioni non si capisce cosa succede.

Assegnamento procedurale **non bloccante** "<="

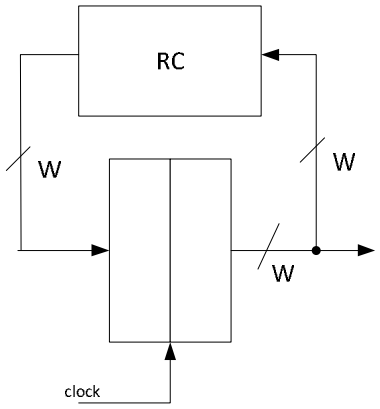
Si noti (è **importante**) la distinzione tra **assegnamento procedurale non bloccante "<=" e assegnamento continuo "=".** Il primo descrive la **scrittura in un registro**, che avviene in un **preciso momento** (vedasi condizione @...). Il secondo è una cosa diversa, e descrive qualcosa che è vero continuamente,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

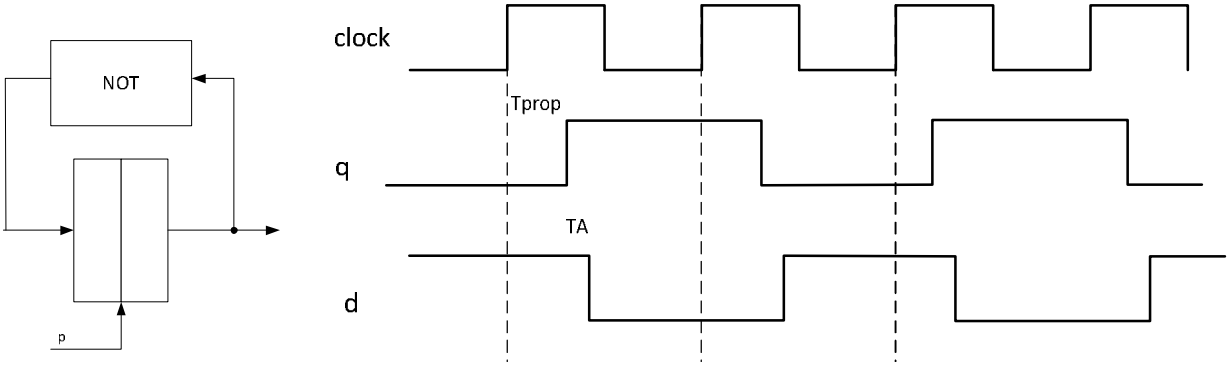
La **temporizzazione** del registro è scritta nella pagina successiva.



1.2 Prima definizione e temporizzazione di una RSS

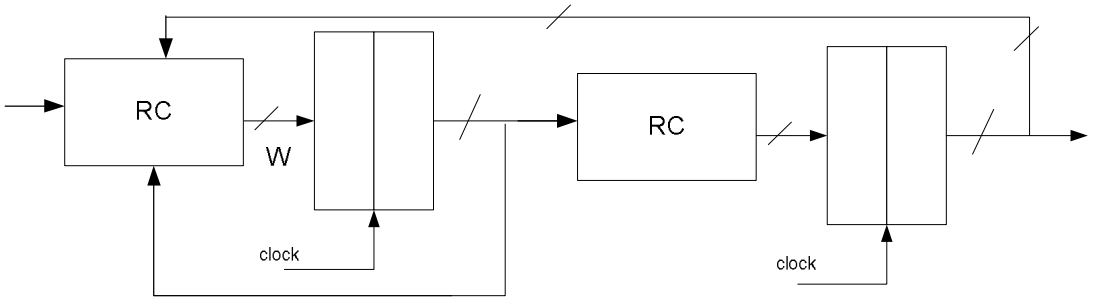


Una **rete sequenziale sincronizzata** è, in prima approssimazione (daremo in seguito definizioni più precise), una **collezione di registri e di reti combinatorie**, montati in qualunque modo si vuole, purché non ci siano **anelli di reti combinatorie** (che invece darebbero vita ad una **rete sequenziale 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. Ad esempio:

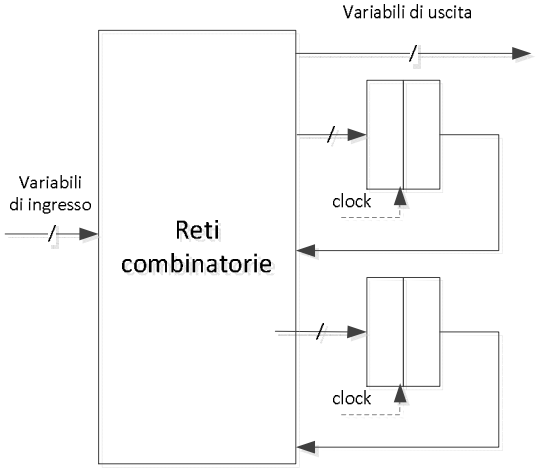


Nonostante l'uscita sia reazionata sull'ingresso, non ci sono oscillazioni incontrollate.

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 $\left[t_i - T_{setup}, t_i + T_{hold}\right]$, 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**:

- $T_{\text{in\_to\_reg}}$: 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**
- $T_{\text{reg\_to\_reg}}$: (… … …) **l'uscita di un registro** e **l'ingresso di un registro**
- $T_{\text{in\_to\_out}}$: (… … …) **un piedino di ingresso** e **un piedino di uscita**
- $T_{\text{reg\_to\_out}}$: (… … …) **l'uscita di un registro** e **un piedino di uscita**

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

a) ingressi costanti in $\left[t_i - T_{setup}, t_i + T_{hold}\right]$ (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.

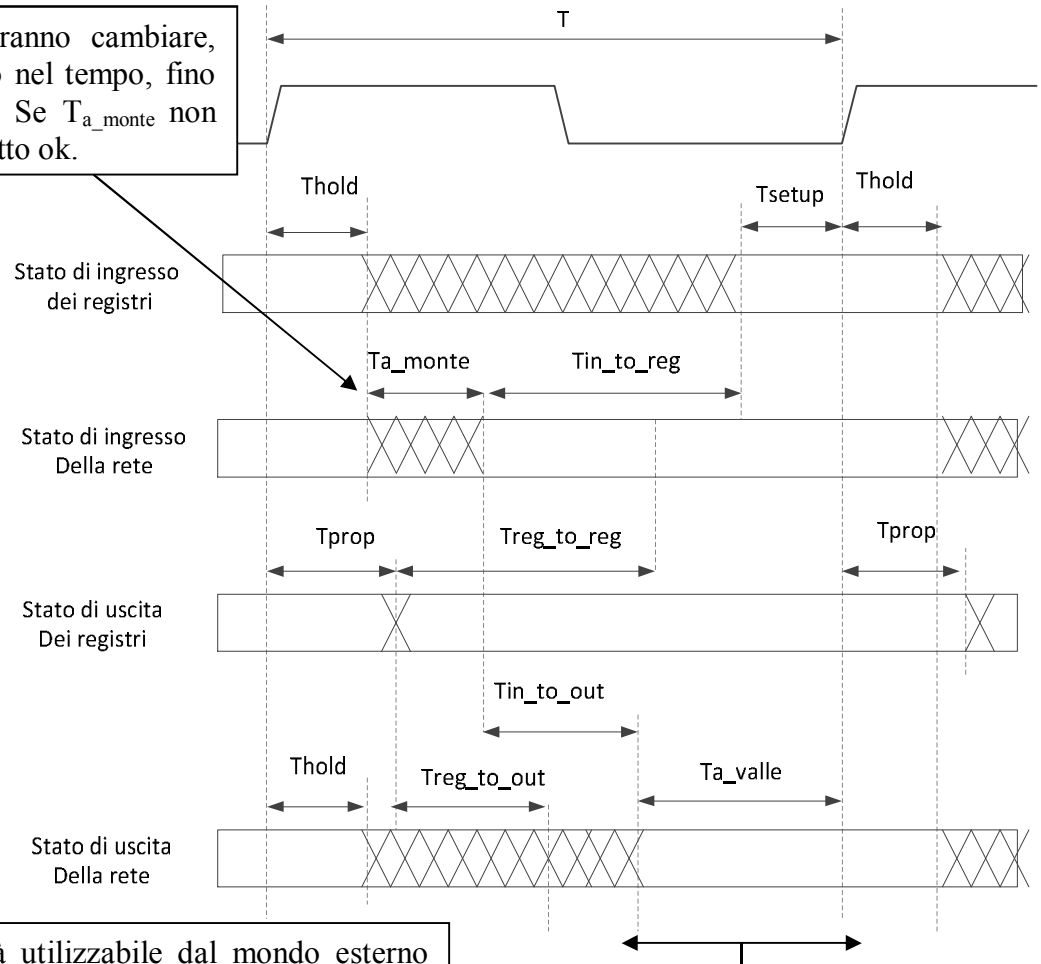
c) vincolo di pilotaggio in uscita: chi usa le uscite (chi sta "a valle" della RSS) deve averle stabili per un tempo T_{a_valle} per poterci fare qualcosa. Al netto di tutti i ritardi sopra scritti, dovrò lasciare una finestra larga almeno T_{a_valle} in ogni periodo di clock perché si possano usare 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)	$T \geq T_{hold} + T_{a\_monte} + T_{\text{in\_to\_reg}} + T_{setup}$	(percorso da ingresso a registro)
2)	$T \geq T_{prop} + T_{\text{reg\_to\_reg}} + T_{setup}$	(percorso da registro a registro)
3)	$T \geq T_{hold} + T_{a\_monte} + T_{\text{in\_to\_out}} + T_{a\_valle}$	(percorso da ingresso a uscita)
4)	$T \geq T_{prop} + T_{\text{reg\_to\_out}} + T_{a\_valle}$	(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 (Treg_to_out)
2) Lo stato di ingresso della rete avrà attraversato le RC fino all'uscita (Tin_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 a qualche registro arrivi prima e a qualche altro dopo, per via dei differenti ritardi sulle linee.
- 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 dimenticarcene.

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

1)	$T \geq T_{sfas} + T_{hold} + T_{a\_monte} + T_{\text{in\_to\_reg}} + T_{setup}$	(percorso da ingresso a registro)
2)	$T \geq T_{sfas} + T_{prop}' + T_{\text{reg\_to\_reg}} + T_{setup}$	(percorso da registro a registro)
3)	$T \geq T_{sfas} + T_{hold} + T_{a\_monte} + T_{\text{in\_to\_out}} + T_{a\_valle}$	(percorso da ingresso a uscita)
4)	$T \geq T_{sfas} + T_{prop}' + T_{\text{reg\_to\_out}} + T_{a\_valle}$	(percorso da registro a uscita)

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

Se si rende 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 dai registri (senza reti combinatorie nel mezzo), nella disequazione 4) scompare il termine $T_{\text{reg\_to\_out}}$. 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.**

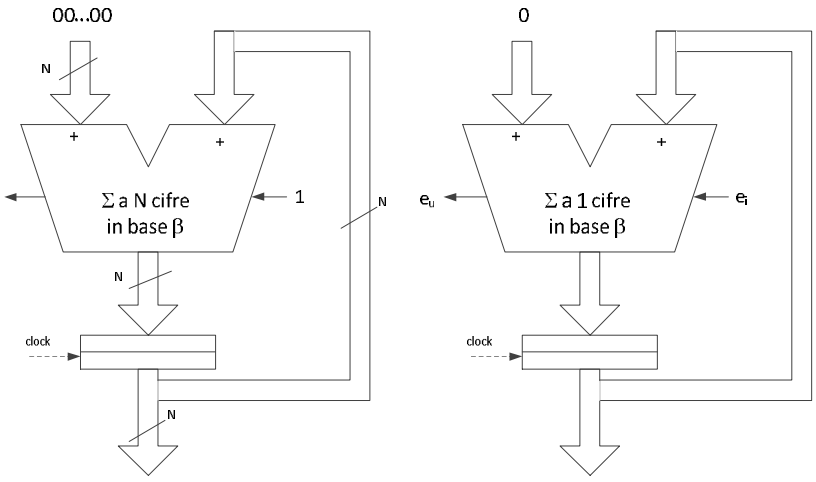
Questa caratteristica consente molta maggior libertà rispetto a quella che avevamo con le RSA, nelle quali eravamo vincolati a cambiare gli ingressi uno alla volta, e a far evolvere la rete soltanto a seguito del cambiamento degli ingressi.

1.3 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);
endmodule
```

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
```

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**:

a) scrivo "-" al posto di "+" nel caso di base 2

b) cambio la funzione "sommatore" scritta prima in qualcos'altro (nel caso di 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.

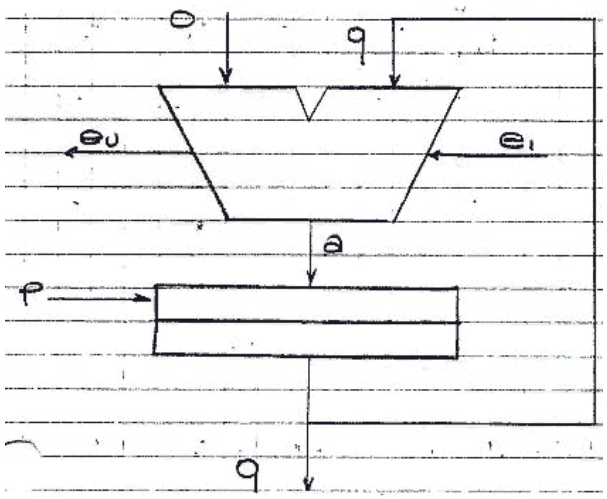
Come si fa a fare questa cosa? Basta che l'ingresso **e_i sia collegato al riporto entrante del sommatore**. In questo caso, la descrizione Verilog per la base 2 verrebbe in questo modo:

```
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
```

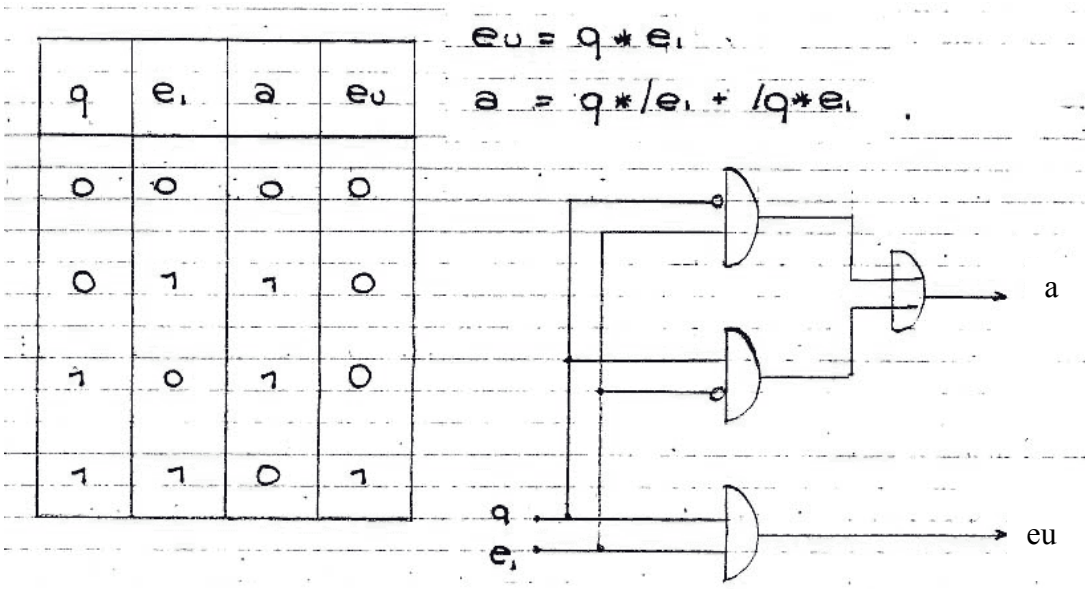
Mentre per la base generica *beta* avrei scritto:

```
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*, o, se ce lo mettiamo, con *carry lookahead*). 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:

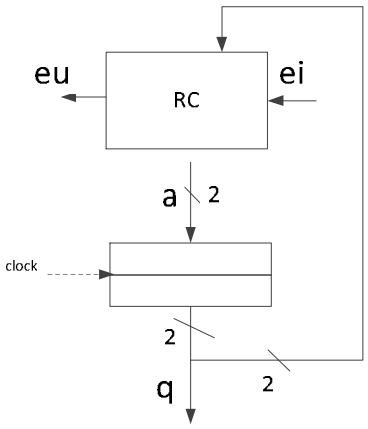
```
module Elemento_Contatore_Base_2(eu,q,ei,clock,reset_);
input clock,reset_;
input ei;
output eu,q;
reg OUTR; assign q=OUTR;
wire a; // variabile di uscita dell'incrementatore
assign {a,eu}=      ({q,ei}=='B00) ?'B00:
                    ({q,ei}=='B10) ?'B10:
                    ({q,ei}=='B01) ?'B10: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

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=B’10**. 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:

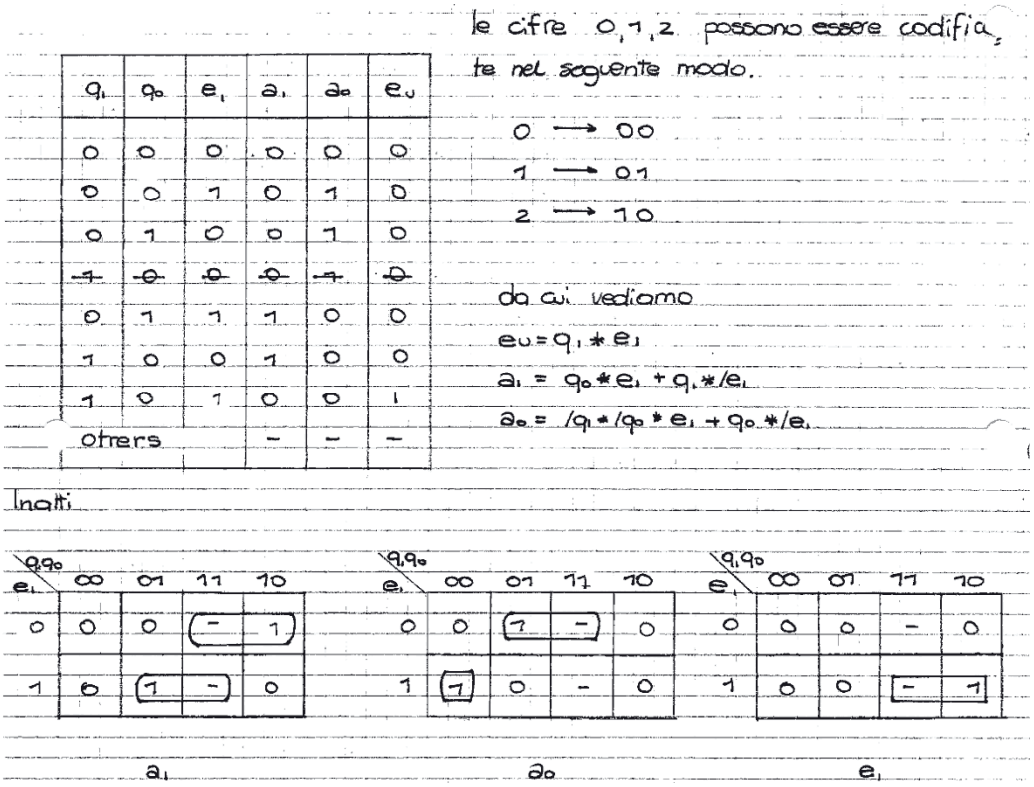
- un disegno come quello in figura, con tanto di tabella di verità
- una descrizione in Verilog



q_1	q_0	e_i	a_1	a_0	e_u
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	0	0	1
others			-	-	-

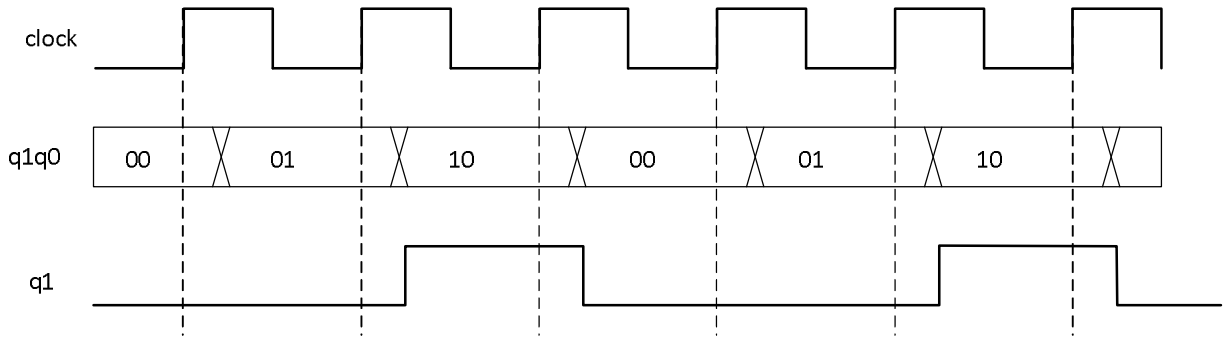
```
module Elemento_Contatore_Base_3(eu,q1_q0,ei,clock,reset_);
input clock,reset_;
input ei;
output eu;
output [1:0] q1_q0;
reg [1:0] OUTR; assign q1_q0=OUTR;
wire [1:0] a1_a0; // variabile di uscita dell'incrementatore
assign {a1_a0,eu}= ({q1_q0,ei}=='B000)?'B000:
                   ({q1_q0,ei}=='B010)?'B010:
                   ({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
```

Per quanto riguarda la sintesi, l'unica cosa che rimane da fare è scrivere espressioni algebriche ottimizzate per le tre uscite:



Osservazione: La sintesi di a_1 è, in teoria, soggetta ad alee, anche se in corrispondenza di una transizione di ingressi che non dovrebbe mai avvenire, in quanto siamo nella colonna q_1q_0=11, che non rappresenta un ingresso valido. La cosa **non ha alcuna rilevanza** (non ne avrebbe anche se l'uscita fosse specificata), in quanto la rete in ingresso al registro potrà presentare in uscita **tutti gli stati spuri che vuole**: basta che lo stato di uscita si sia stabilizzato entro l'istante giusto (ma questo dipende **dal tempo di attraversamento della rete**, e **non dall'eventuale presenza di alee**). Mettere un implicante (un AND) in più per eliminare l'alea, in questo caso, **non serve a niente.** Aumenta il costo, non riduce il tempo di attraversamento, e svolge un mestiere (eliminare l'alea) di cui non c'è alcun bisogno.

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.



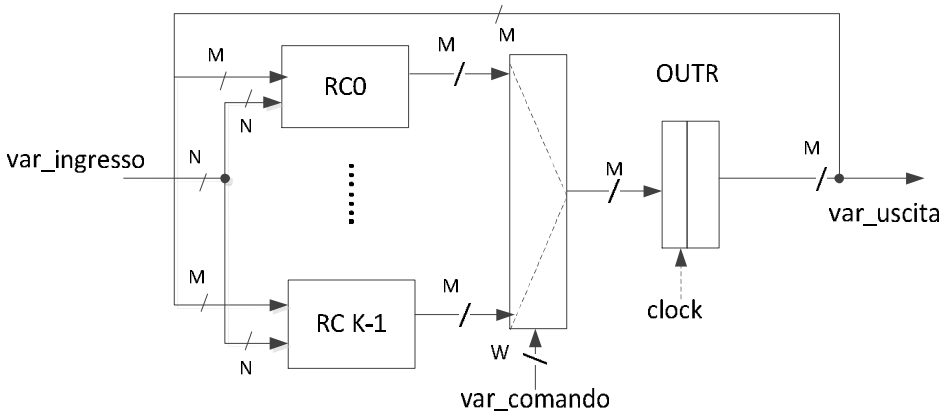
Analogamente, la **cifra più significativa** di un contatore ad *N* cifre b2 che riceve clock a periodo T può essere usata come clock a periodo $2^N \cdot T$. Si noti che, per generare un clock in questo modo, si possono usare solo **uscite di registri**, **mai quelle di reti combinatorie** (e.g., il riporto uscente e_u). Infatti, le uscite di combinatorie possono ballare, mentre un clock deve essere assolutamente stabile.

Alcune note sui contatori

- Esistono contatori **up/down**, in cui c'è un'ulteriore variabile di comando che dice se contare up o contare down. Si implementano con **sommatori/sottrattori**, la cui sintesi abbiamo visto a suo tempo.
- Nei contatori ad una cifra, il clock potrebbe essere dimensionato sulla base del solo ritardo della rete combinatoria che implementa il **semisommatore**. In realtà, bisogna osservare che, se molti di questi elementi vengono messi **in cascata** (per fare un contatore ad *N* cifre), **la catena dei riporti fa una rete combinatoria unica**, e quindi il clock va dimensionato tenendo conto anche di quella. Il "tempo a monte" e "tempo a valle" è in questo caso dato dal tempo che ci mette il resto del contatore ad andare a regime. I circuiti di *lookahead* servono appunto ad evitare che la catena dei riporti rallenti troppo il clock.

1.4 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.



La descrizione Verilog, è:

```
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) #3
casex(var_comando)
   0 : OUTR<=F0(var_ingresso,OUTR);
   ...
   ...
   K-1: OUTR<=FK-1(var_ingresso,OUTR);
endcase
endmodule
```

F_0(… …)
…
F_{K-1}(… …)
Saranno **reti combinatorie**, che eventualmente descriveremo come **funzioni**, come sempre.

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

- **carica** un nuovo valore, cioè memorizza *N* 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:

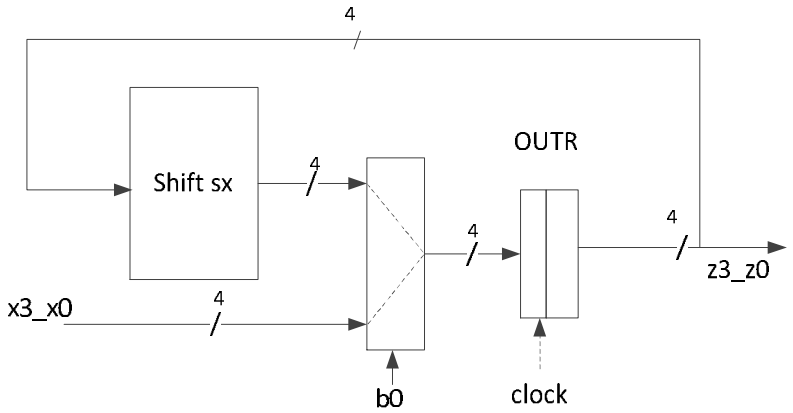
```
module Registro_CaricaParallelo_TraslaSinistro(z3_z0,x3_x0,b0,
clock,reset_);
input clock,reset_;
input [3:0] x3_x0;
input b0;
output [3:0] z3_z0;

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

always @(reset_==0) #1 OUTR<='B0000;
always @(posedge clock) if (reset_==1) #3
casex(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 <=** ci può stare qualunque espressione (che può coinvolgere anche bit del registro), **a sinistra** ci deve stare **un registro intero**. Non ha senso scrivere

`OUTR[2]<='B1`

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



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.

1.5 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 $\mathbf{S} \equiv \{S_0, ..., S_{K-1}\}$
4. Una **legge di evoluzione nel tempo** del tipo $A : \mathbf{X} \times \mathbf{S} \to \mathbf{S}$, che mappa quindi una coppia (stato di ingresso, stato interno) in un nuovo stato interno.
5. Una **legge di evoluzione nel tempo** del tipo $B : \mathbf{S} \to \mathbf{Z}$, che decide lo stato di uscita basandosi sullo stato interno. (Nota: tale legge **non è** più generale, del tipo $B : \mathbf{X} \times \mathbf{S} \to \mathbf{Z}$. Se fosse più generale saremmo **fuori dal modello**).

Fin qui, ho detto esattamente le stesse cose che dissi a suo tempo per le reti **sequenziali asincrone**. La vera differenza arriva adesso:

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**,

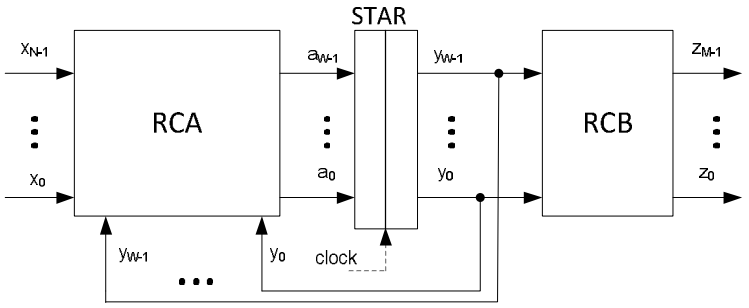
a) individuare il **nuovo stato interno da marcare S'=A(S,X)**

b) **attendere l'arrivo del segnale di sincronizzazione**

c) **promuovere S'** al rango di **stato interno marcato, quando il registro non è più sensibile all'ingresso**

d) **individuare <u>continuamente</u> 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, ovviamente, una **batteria di D-FF**, che sono **<u>non trasparenti</u>.** 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.

Mentre nel caso di RSA c'eravamo fatti un sacco di problemi riguardo al fatto che RCA potesse manifestare **stati di uscita spuri**, per una o più delle seguenti cause:

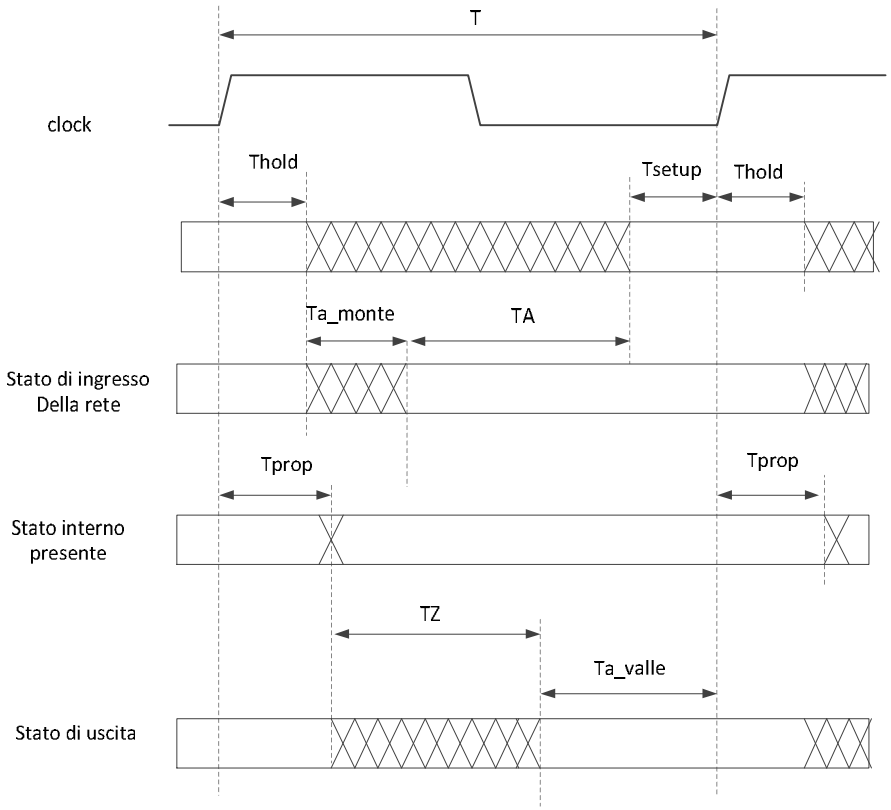
- **alee** (combinatorie) di RCA
- **alee essenziali**, cioè reazione troppo veloce delle variabili di stato che induce stati di uscita non voluti su RCA
- **codifiche non adiacenti** di stati interni consecutivi, che instaurano **corse delle var. di stato.**
- **errori di pilotaggio**

In questo caso posso osservare che

a) lo stato interno marcato rientra quando il registro non è più sensibile: **<u>niente alee essenziali, né problemi di corse</u>**. Stati <u>interni</u> consecutivi possono essere arbitrariamente distanti.

b) Purché sia in grado di tenere l'uscita di RCA stabile in $\left[t_i - T_{setup}, t_i + T_{hold}\right]$, eventuali alee su RCA non danno problemi. Stati di <u>ingresso</u> consecutivi possono essere arbitrariamente distanti

L'unica cosa alla quale devo stare attento, allora, è **la temporizzazione (cioè il pilotaggio).** Una temporizzazione che garantisce il rispetto delle regole già viste per una rete di Moore è la seguente:

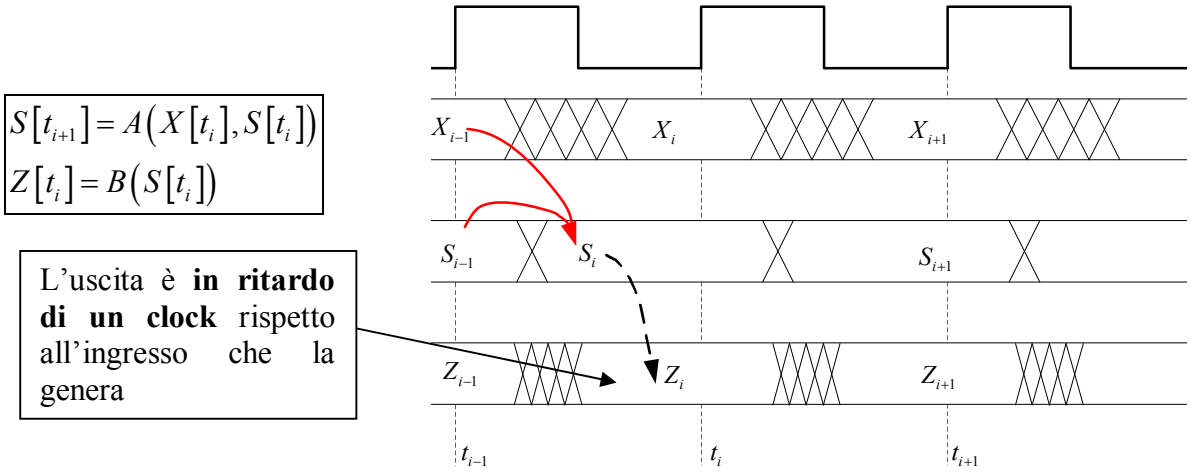


Le equazioni che ne derivano sono le seguenti:

$T \geq T_{hold} + T_{a\_monte} + T_A + T_{setup}$	(percorso da ingresso a STAR)
$T \geq T_{prop} + T_A + T_{setup}$	(percorso da STAR a STAR)
$T \geq T_{prop} + T_Z + T_{a\_valle}$	(percorso da STAR a uscita)

Di queste, la seconda è certamente meno restrittiva della prima, in quanto T_{prop} e T_{hold} sono dello stesso ordine di grandezza, mentre $T_{a\_monte} \gg T_{prop}$. Quindi può essere **ignorata**, perché implicata dalla prima.

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).

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:

Dichiarazione di costante

```
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
casex(STAR)
    S0 : STAR<=AS0(xN-1,...,x0);
    S1 : STAR<=AS1(xN-1,...,x0);
    ...
    SK-1: STAR<=ASK-1(xN-1,...,x0);
endcase
endmodule
```

Legge B

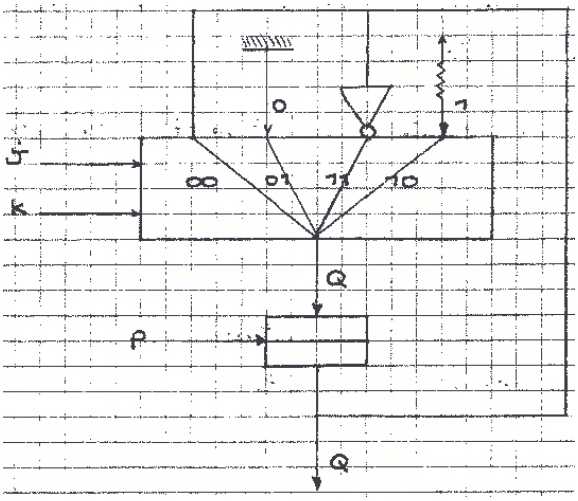
Legge A

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

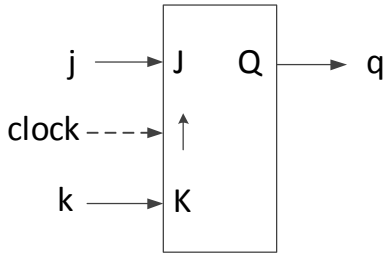
1.5.1 Esempio: il Flip-Flop JK

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:

jk	**Azione in uscita**
00	Conserva
10	Setta
01	Resetta
11	Commuta



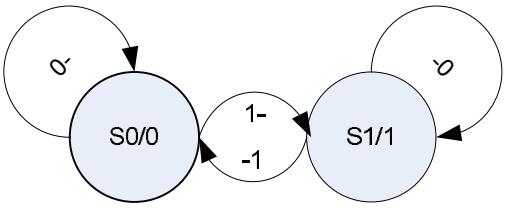
Un modo (non ottimizzato) di vedere questa rete è come **registro multifunzionale ad un bit**. Il multiplexer a 4 vie prende in ingresso *j* e *k*, e commuta l'uscita, la sua negata, e due costanti. Posso dare, per questa rete, una **tabella di applicazione** simile a quella del Latch SR. Visto che ci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



questa tabella vuol dire: *se **quando arriva il clock** voglio che la variabile di uscita da q diventi q'…*

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
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**.

Domanda: la rete oscilla (o, detto in altro modo, è **instabile)?**

Risposta: **certo che <u>NO</u>**. Anche se nella colonna 11 ho due stati che rimandano l'uno all'altro, le transizioni avvengono **quando arriva il clock**, e non quando cambiano gli ingressi. È chiaro che, se mantengo entrambi gli ingressi a 11, la rete ad ogni clock cambia uscita (di fatto, così pilotata, darebbe in uscita un clock con periodo $2T$).

Per questo motivo nelle RSS **non ha senso cerchiare gli stati e parlare di stati stabili**: il concetto di stabilità è legato alla presenza di **anelli combinatori**. In una rete di Moore non ci sono anelli combinatori. 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 dello **stesso stato** dove mi trovavo, come nel caso di S0 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 procedere alla **sintesi** di RCA.

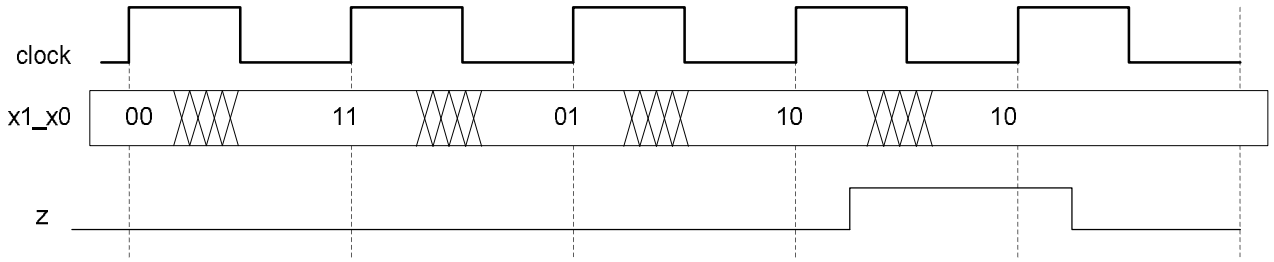
	jk 00	01	11	10	q
S0	S0	S0	S1	S1	0
S1	S1	S0	S0	S1	1

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

$a_0 = j \cdot \overline{y_0} + \overline{k} \cdot y_0$, $q = y_0$. Così facendo RCA sarà soggetta ad alee, ma **non è più un problema.**

1.5.2 Esempio: riconoscitore di sequenze 11,01,10

Voglio costruire un riconoscitore di sequenze di 2 bit come **RSS di Moore**. Facciamo sequenze lunghe 3, sennò ci vogliono troppi stati interni. Attenzione a cosa si intende per **riconoscere una sequenza**, ad esempio 11, 01, 10: significa che **in tre clock consecutivi si devono presentare i tre valori dello stato di ingresso.** Se un valore permane per più di un ciclo di clock, la sequenza è **diversa**.



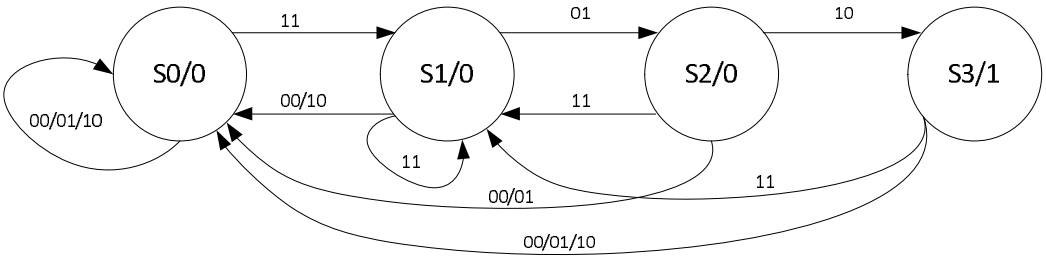
Domanda: posso riconoscere sequenze del tipo **11,01,10? 11,11,11?**

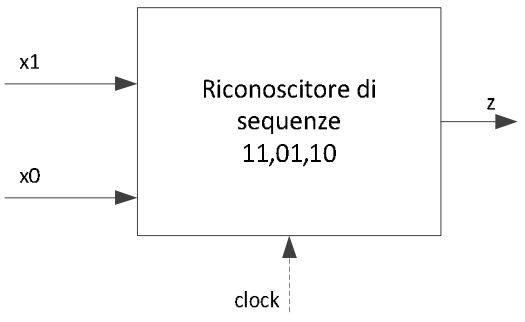
Certo che sì, in quanto l'ingresso:

- viene valutato **sul fronte del clock**, e tra due fronti può cambiare come si vuole
- stati di ingresso identici **valutati in tempi diversi** sono comunque distinti. Il secondo caso è quello di una rete che sblocca la serratura se l'ingresso vale 11 per 3 clock.

Descriviamo e sintetizziamo allora la rete che riconosce la sequenza **11, 01, 10**. Una tale rete non potrebbe **mai essere implementata come RSA**. La descriviamo dandone il **grafo di flusso**.

- Descrivere il grafo di flusso, partendo dalla **catena di ingressi corretta** che sblocca la serratura
- Osservare che qualunque stato di ingresso non permesso riporta in S0, a meno che non sia l'ingresso 11, che può essere l'inizio di una nuova sequenza corretta e quindi deve portare in S1.





x_1x_0	00	01	11	10	z
S_0	S0	S0	S1	S0	0
S_1	S_0	S2	S1	S0	0
S_2	S0	S0	S1	S3	0
S_3	S0	S0	S1	S0	1

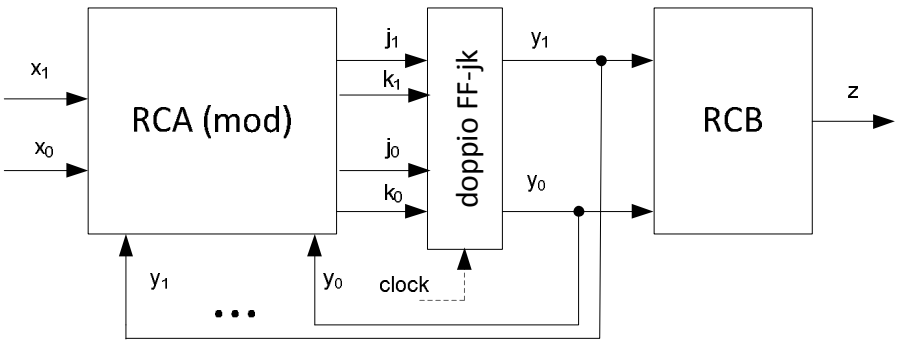
Nota importante: siamo d'accordo che lo stato di ingresso può cambiare come vuole tra due clock successivi, fatto salvo che rimanga stabile a cavallo del fronte di salita. Va però tenuto conto del fatto che, se devo riconoscere una sequenza di stati di ingresso con una RSS, il vincolo è che la sequenza 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. In una RSS, il concetto di **tempo** gioca un ruolo chiave. **In una RSA, invece, il tempo non esiste** (esistono solo le transizioni di ingresso).

```
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
```

Attenzione: le codifiche degli stati interni consecutivi non hanno nessun bisogno di essere adiacenti. Pertanto posso sceglierle come voglio. Un buon criterio di scelta è **guardare cosa deve fare RCB**, e scegliere le codifiche in modo da dover usare meno logica possibile.

Posso sintetizzare il riconoscitore usando il modello strutturale visto prima ma posso anche usare una **variante** del precedente, che fa uso di **FF JK come elementi di marcatura**. Sempre di rete di Moore si tratta, ovviamente. Sarà diversa la sintesi di RCA, ed in generale più semplice, perché ogni pilotaggio del FF JK ha **almeno un ingresso non specificato.** C'è un'**ovvia dualità** tra modelli per reti asincrone (che noi abbiamo infatti supposto essere di Moore, anche se non l'abbiamo detto esplicitamente) e modelli per reti sincronizzate di Moore. Nelle prime usavo, alternativamente, **elementi neutri e latch SR**. In queste, userò alternativamente **D-FF** (che sono, di fatto, elementi neutri di memorizzazione per reti sincronizzate) e **FF JK**, che sono l'omologo sincronizzato dei latch SR.



La sintesi si fa in maniera identica a come visto nel caso di RSA che usino il Latch SR come elemento di marcatura. Si noti che in questo caso RCA viene **ancora più semplice**, visto che la tabella di applicazione del FF JK ha sempre almeno un ingresso non specificato.

x_1x_0	00	01	11	10	z
S_0	S0	S0	S1	S0	0
S_1	S_0	S2	S1	S0	0
S_2	S0	S0	S1	S3	0
S_3	S0	S0	S1	S0	1

Sint	y_1y_0
S_0	00
S_1	01
S_2	10
S_3	11

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

a_1a_0

q q′	jk
0 0	0-
0 1	1-
1 0	-1
1 1	-0

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

j_1k_1

y_1y_0 \ x_1x_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_0k_0

y_1y_0	z
00	0
01	0
10	0
11	1

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

1.5.3 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.

1.5.4 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

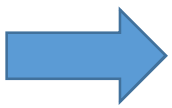
2) Per quanto riguarda la sintesi, si può osservare quanto segue:

- adottando la codifica degli stati $S_i = (i)_{b2}$, 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).

y2y1y0	x=0	x=1	z
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

Decido di adottare un modello di sintesi con D-FF come elementi di marcatura. Per svolgere la sintesi metto la variabile di stato più significativa come variabile in colonna insieme agli ingressi:

	y2=0		y2=1	
x	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_1y_0 \ y_2x	$a_2a_1a_0$ 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 implicanti sono essenziali. Si ottiene quanto segue:

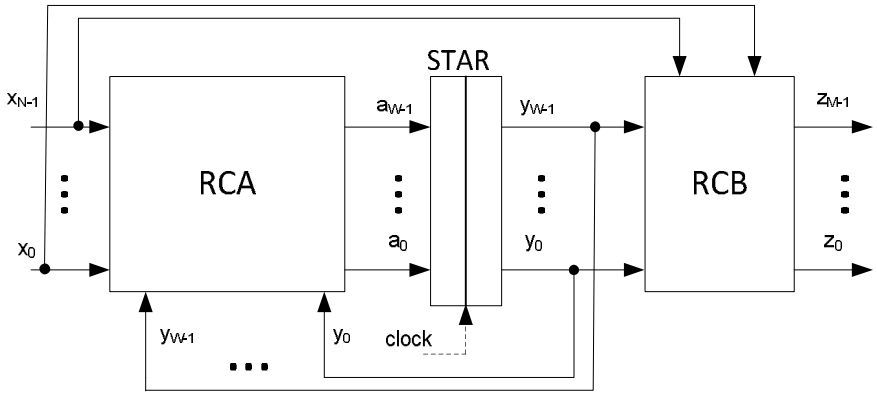
$$a_2 = y_2 \cdot \overline{y_1} \cdot x + \overline{y_2} \cdot y_1 \cdot y_0 \cdot \overline{x} + y_2 \cdot y_1 \cdot \overline{y_0} \cdot \overline{x}$$

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

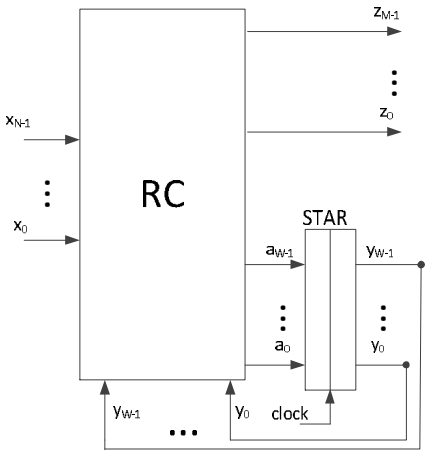
$$a_0 = \overline{y_2} \cdot \overline{y_1} \cdot \overline{y_0} \cdot \overline{x} + y_2 \cdot \overline{y_1} \cdot \overline{y_0} \cdot x + \overline{y_2} \cdot y_1 \cdot \overline{y_0} \cdot x + y_2 \cdot y_1 \cdot \overline{x} + y_2 \cdot y_0 \cdot \overline{x}$$

1.6 Modello di Mealy

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



Nel disegno di sopra le reti RCA e RCB hanno **gli stessi ingressi.** Pertanto poso disegnare una rete di Mealy anche in questo modo, evitando di distinguerle.



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)

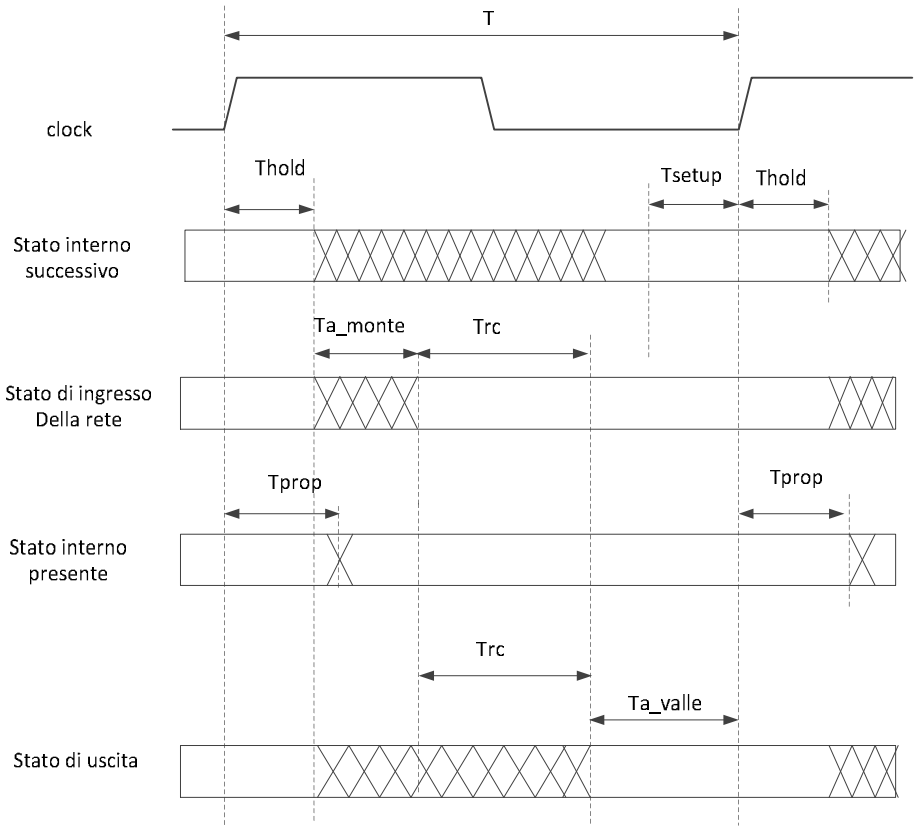
$\boxed{T \geq T_{hold} + T_{a\_monte} + T_{RC} + T_{a\_valle}}$ (percorso da ingresso a uscita)

$T \geq T_{prop} + T_{RC} + T_{a\_valle}$ (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.



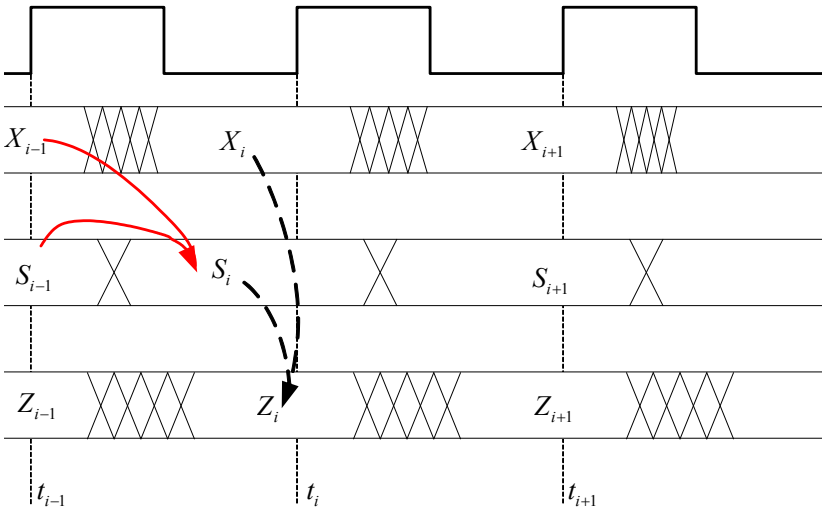
```
module Rete_di_Mealy(zM-1,...,z0,xN-1,...,x0,clock,reset_);
input clock,reset_;
input xN-1,...,x0;
output zM-1,...,z0;
reg [W-1:0] STAR; parameter S0=codifica0,...,SK-1=codificaK-1;
assign {zM-1,...,z0} = (STAR==S0)? zS0(xN-1,...,x0) :
                                ...
                                ...
                                ...
                                /*(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.**

$$S[t_{i+1}] = A(X[t_i], S[t_i])$$
$$Z[t_i] = B(X[t_i], S[t_i])$$

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:

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

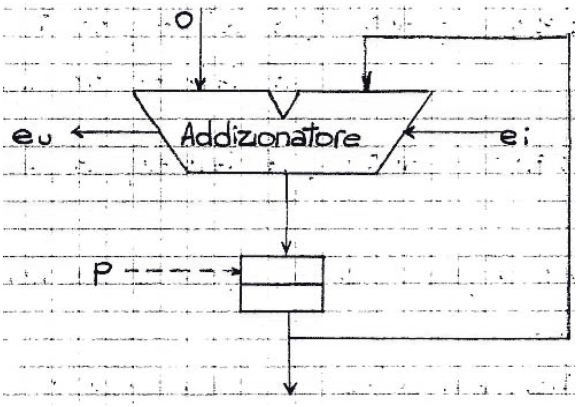
- nelle reti di Mealy, **l'uscita dipende anche dall'ultimo stato di ingresso, quello presente al clock attuale.**

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

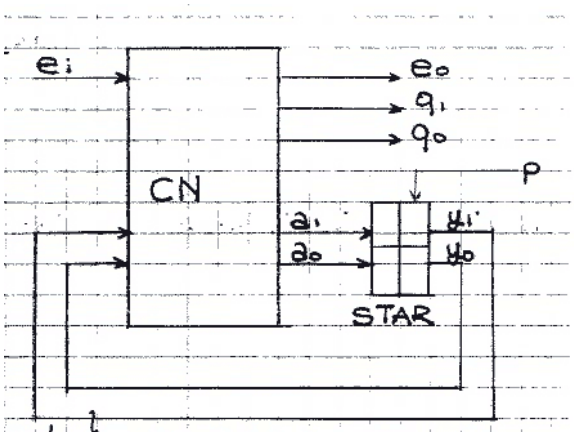
Essendo la legge **B** più flessibile che nel caso precedente, in genere si riescono a risolvere gli stessi problemi con un numero **minore di stati interni**.

1.6.1 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).

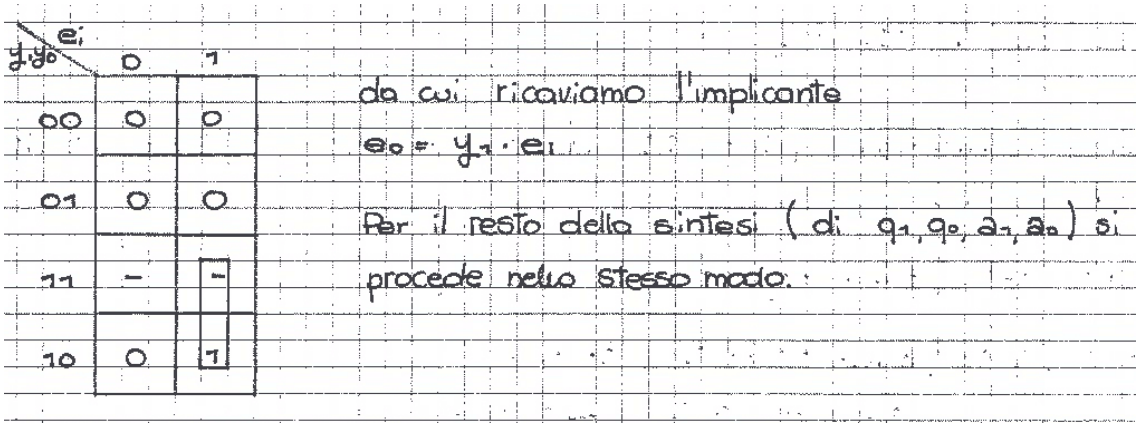
		e_i	q1q0 e_u 0	1
(00)	S0		S0 00 0	S1 00 0
(01)	S1		S1 01 0	S2 01 0
(10)	S2		S2 10 0	S0 10 1

	e_i	e_u 0	1	q1q0
S0		S0 0	S1 0	00
S1		S1 0	S2 0	01
S2		S2 0	S0 1	10

Mentre le uscite q1q0 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_1q_0 in una colonna a parte.

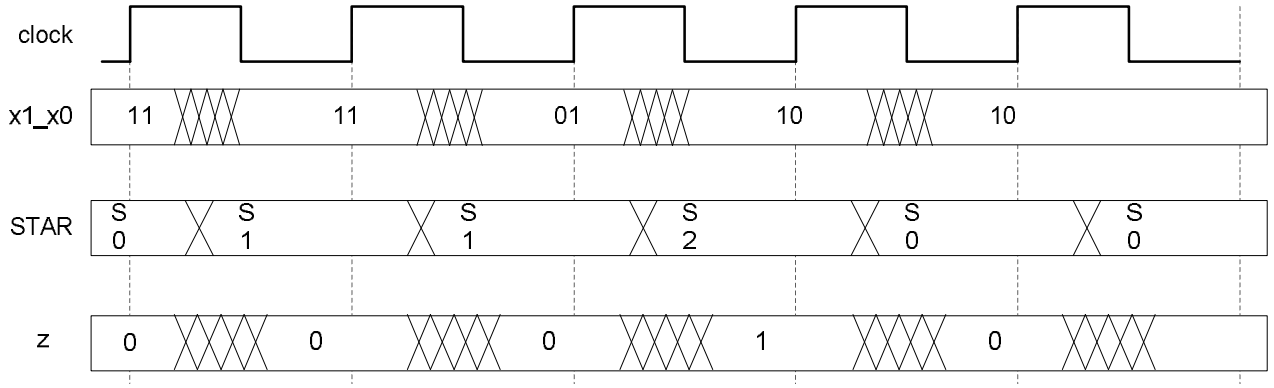
Adottando le codifiche (ovvie) **S0='B00, S1='B01, S2='B10**, e scrivendo la tabella delle transizioni dalla tabella di flusso si ottiene molto velocemente **la stessa sintesi** già vista a suo tempo:



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

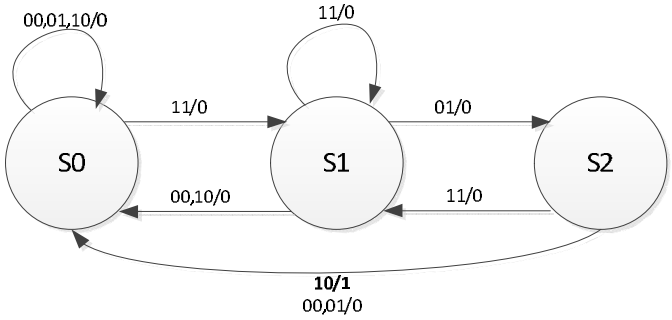
1.6.2 Esempio: riconoscitore di sequenza 11, 01, 10

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.

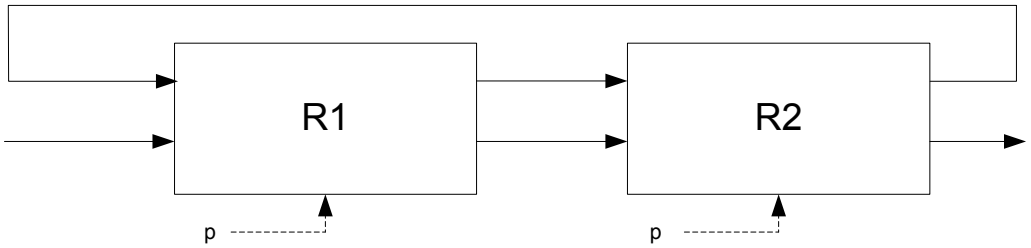
Posso descrivere questa rete anche in Verilog, come segue:

```
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.**

1.6.3 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**, $z = x_1$ **AND** x_0**,** altrimenti, se lo stato d'ingresso corrente **non è uguale al precedente**, $z = x_1$ **XOR** x_0.

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

1.6.4 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 S0, S1, S2, S3, e disegniamo la tabella di flusso. Posso associare lo stato interno al precedente stato di ingresso nel seguente modo:

S. interno	S. ingresso precedente
S0	00
S1	01
S2	11
S3	10

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

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	$S_0/0$	$S_1/1$	$S_2/0$	$S_3/0$

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.

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): $z = (x_1 + x_0) \cdot (\overline{x}_0 + y_1 + \overline{y}_0) \cdot (\overline{x}_1 + \overline{y}_1 + y_0) \cdot (\overline{x}_1 + \overline{x}_0 + y_1)$

1.7 Modello di Mealy ritardato

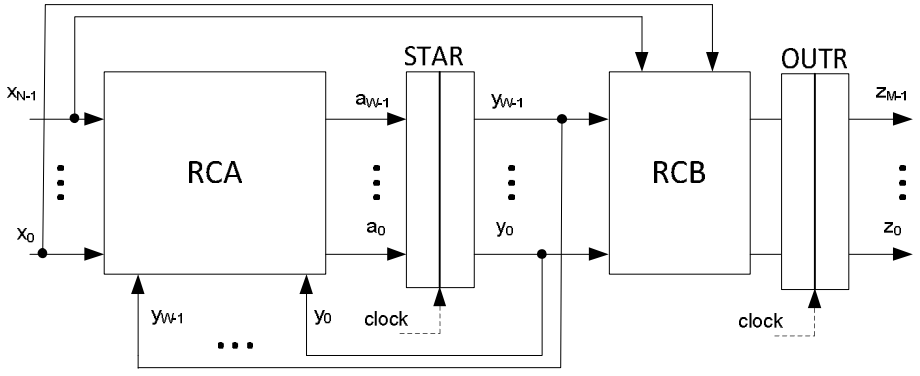
Si parte da una rete di Mealy, e si mette in uscita un **registro OUTR**. In questo modo, le uscite:

a) variano sempre **all'arrivo del clock**, dopo un tempo T_{prop} (donde l'aggettivo "ritardato");

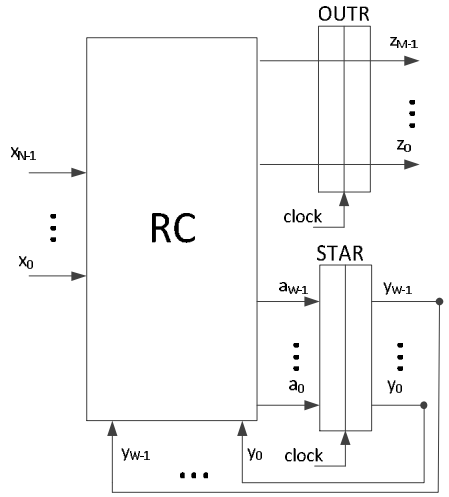
b) variano in maniera **netta, senza oscillazioni** (come invece può succedere in una rete di Mealy se gli ingressi ballano un po' prima di stabilizzarsi);

c) rimangono stabili per **l'intero ciclo di clock.**

d) 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 $A: \mathbf{X} \times \mathbf{S} \to \mathbf{S}$, che mappa quindi una coppia (stato di ingresso, stato interno) in un nuovo stato interno.
2) Una **legge di evoluzione nel tempo** del tipo $B: \mathbf{X} \times \mathbf{S} \to \mathbf{Z}$, 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 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, quando la rete non è più sensibile agli ingressi.**

Attenzione a capire **bene** una cosa (non averla capita ora rende impossibile risolvere i compiti d'esame **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:

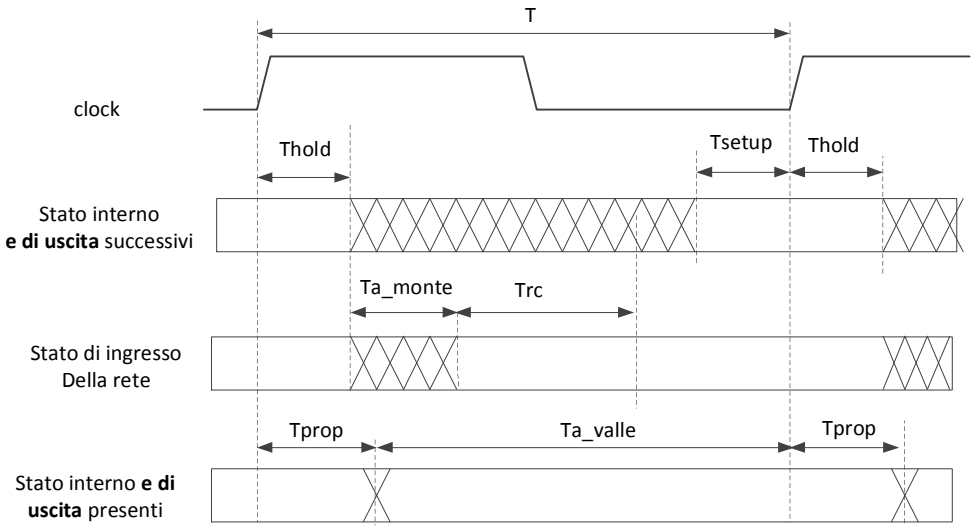
$T \geq T_{hold} + T_{a\_monte} + T_{RC} + T_{setup}$ (percorso da ingresso a registro)

$T \geq T_{prop} + T_{RC} + T_{setup}$ (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**.

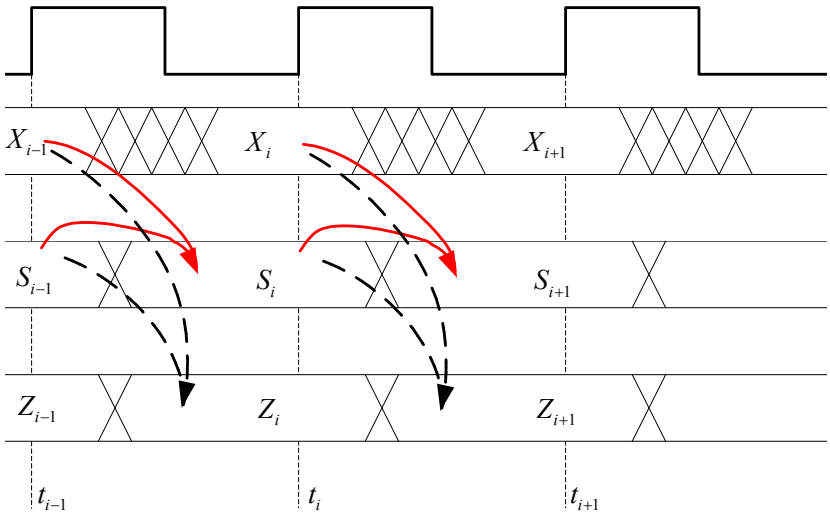
Se le condizioni di temporizzazione sono rispettate (vediamo tra un attimo), una rete di Mealy ritardato si evolve in modo **deterministico**. Detti t_i, $i \geq 0$, gli **istanti di sincronizzazione**, e detti $X[t_i], S[t_i], Z[t_i]$ gli stati di **ingresso, interno e di uscita** all'istante t_i, sarà:



$$S[t_{i+1}] = A(X[t_i], S[t_i])$$
$$Z[t_{i+1}] = B(X[t_i], S[t_i])$$

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] OUTR; assign {zM-1,...,z0}=OUTR;

always @(reset_==0) #1 begin OUTR<=...; STAR<=...; end
always @(posedge clock) if (reset_==1) #3
```

```
casex(STAR)
   S0 : begin
        OUTR<=ZS0(xN-1,...,x0);
        STAR<=AS0(xN-1,...,x0);
        end
   ...
   SK-1 : begin
        OUTR<=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 `OUTR` a destra di `STAR<=`**, sto usando il **vecchio valore** (quello **prima** del fronte del clock)**, non il nuovo.**

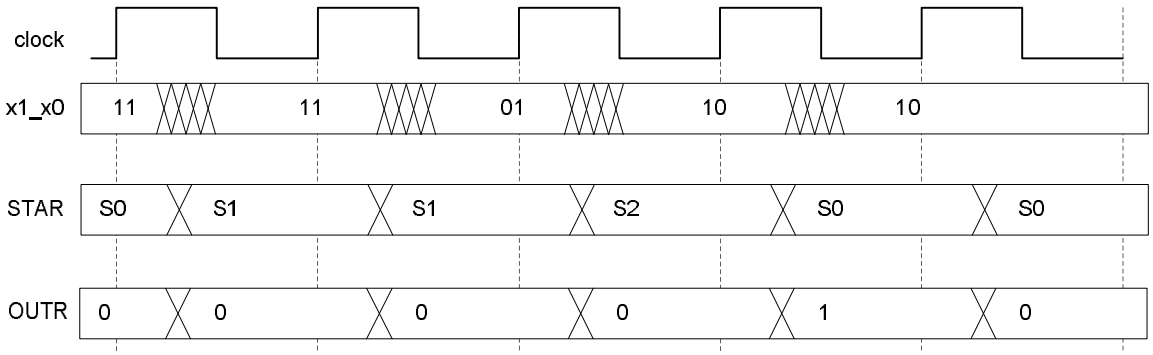
Posso anche descrivere una rete di Mealy ritardato con **tabelle e grafi di flusso** (ammesso che sia semplice abbastanza). In questo caso, la descrizione sarà **visivamente identica a quella di una rete di Mealy standard**, in quanto il fatto che sia **Mealy o Mealy ritardato** sta nella maniera di **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_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

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 radicalmente 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
```

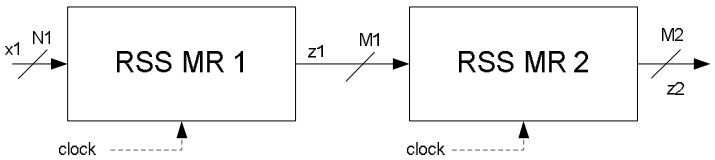
Il diagramma di temporizzazione relativo ad una possibile evoluzione di questa rete sarà quindi:



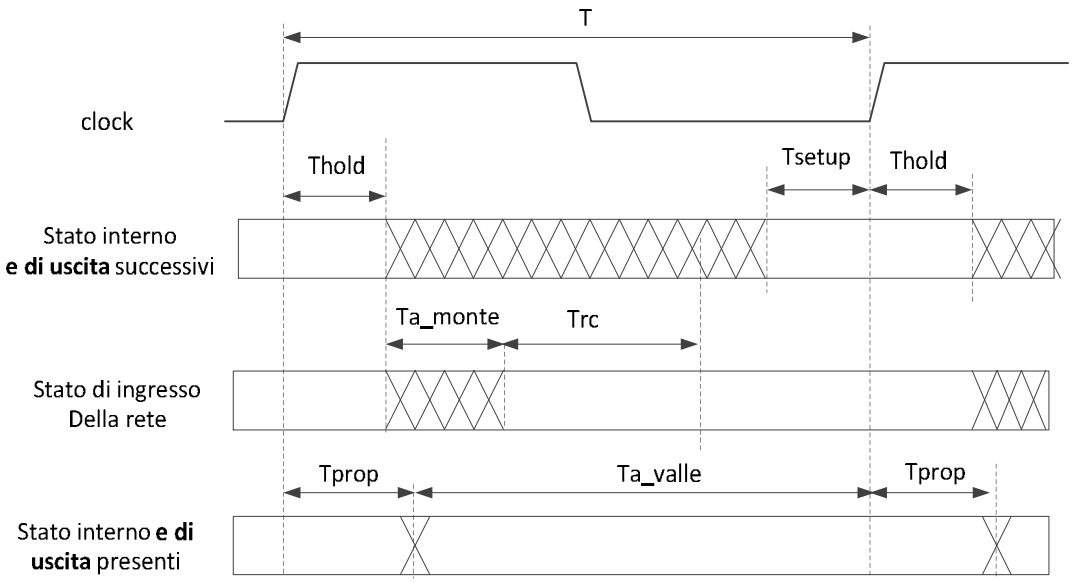
Il che giustifica ancora una volta l'aggettivo **ritardato**. L'uscita è in ritardo di un clock rispetto a quanto succedeva nel modello di Mealy.

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 piloto 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.

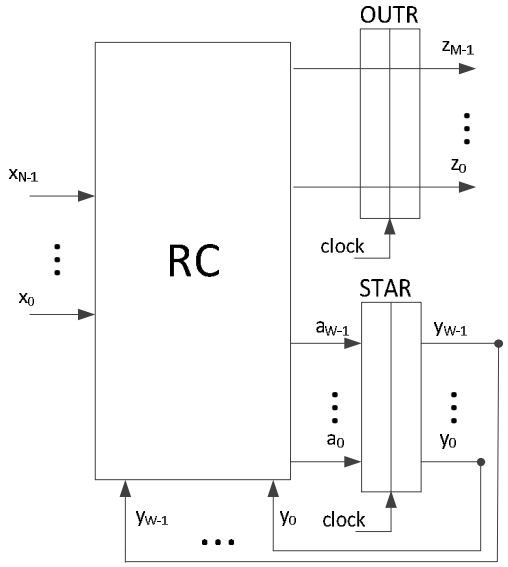


Riepilogando, le reti di Mealy ritardato:

- **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.

2 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.



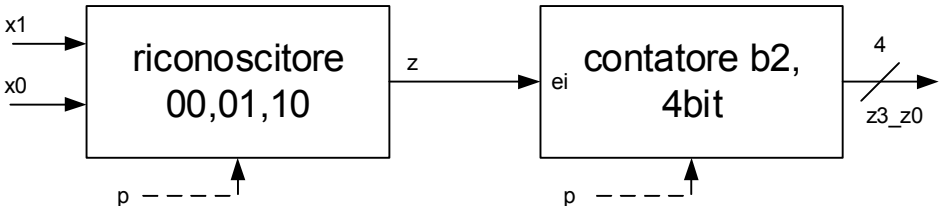
2.1 Linguaggio di trasferimento tra registri

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**?

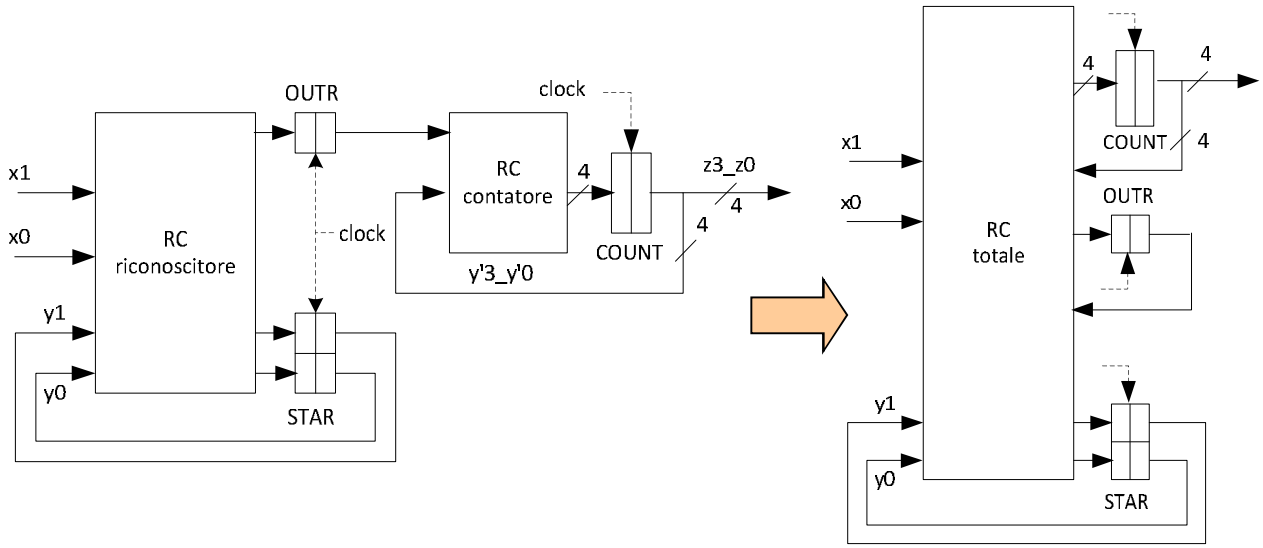
Facendo un conto a spanne, se ci vogliono 3 stati interni per riconoscere una sequenza di tre passi corretti (visto che adopero una rete di Mealy ritardato, perché se fosse stata di Moore ce ne sarebbero voluti 4), allora i 4 bit di uscita dovrebbero cambiare di uno ogni tre stati interni, e mi dovrebbero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

Si può però osservare che, per realizzare una rete così fatta, la soluzione proposta 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 dal registro di stato). 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:

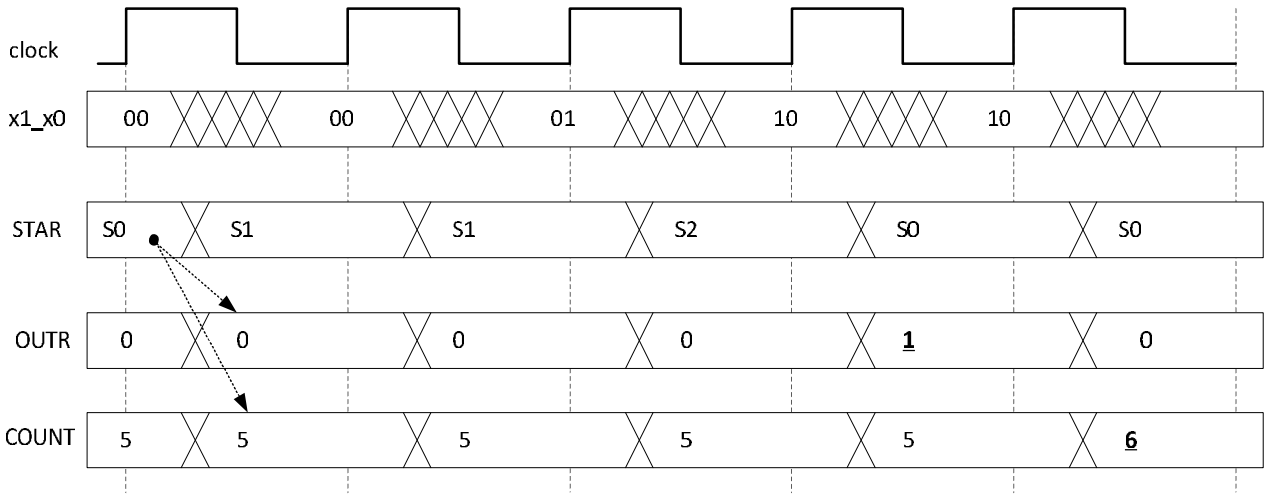
- Ho un registro di **stato STAR**, che svolge le stesse funzioni che in una RSS qualunque;
- posso usare quanti altri registri voglio, della capacità che voglio. Tali registri prendono il nome di **registri operativi** (ma questo non è un grosso miglioramento, tanto varrebbe avere un registro solo OUTR "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 le descrizioni molto **compatte**. Tali descrizioni, poi, possono essere **sintetizzate** in maniera automatica, con poco sforzo.

2.1.1 Esempio: contatore di sequenze 00,01,10

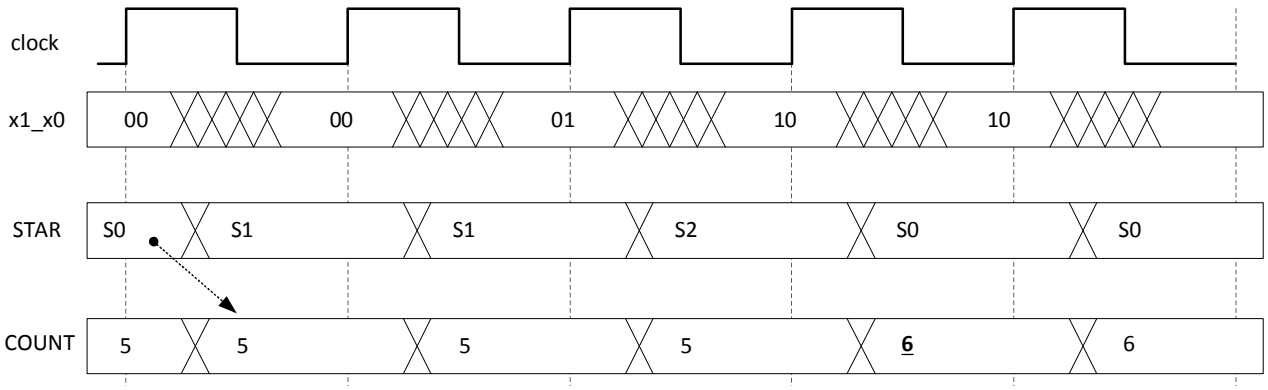
```
// Contatore di sequenze secondo il modello generalizzato
module Contatore_Sequenze(z3_z0, x1_x0, clock, reset_)

input clock, reset_;
input [1:0] x1_x0;
output [3:0] z3_z0;

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

parameter     S0='B00, S1='B01, S2='B10;

assign z3_z0=COUNT;

always @(reset_ ==0) #1 begin OUTR<=0; COUNT<=0; STAR<=S0; end
always @(posedge clock) if (reset_==1) #3
   casex(STAR)
   S0 : begin OUTR<=0; COUNT<=COUNT+OUTR; STAR<=(x1_x0=='B00)?S1:S0;
              end
   S1 : begin OUTR<=0; COUNT<=COUNT+OUTR;
              STAR<=(x1_x0=='B01)?S2:(x1_x0=='B00)?S1:S0; end
   S2 : begin OUTR<=(x1_x0=='B10)?1:0; COUNT<=COUNT+OUTR;
              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 OUTR e **poi** sommato a COUNT. Visto che non abbiamo bisogno di questo, possiamo ottimizzare la descrizione **eliminando OUTR,** 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 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 `casex` 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 "scatolette" 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.

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

Variazione 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 incrementi quando vede la **seconda**, poi di nuovo la prima, e così via in modo alternato.
La rete avrà due var. di ingresso (x1x0) 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
```

3 Esercizi

3.1 Esercizio – Rete di Moore

1) 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 $j = k = 1$.

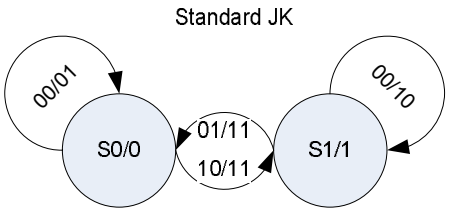
   In tal caso infatti porta q ad 1 se la volta precedente in cui lo stato d'ingresso $j = k = 1$ si era presentato, l'uscita era stata resettata; porta q a 0 se la volta precedente in cui lo stato d'ingresso $j = k = 1$ si era presentato, l'uscita era stata settata.

   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

3.1.1 Descrizione della rete

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

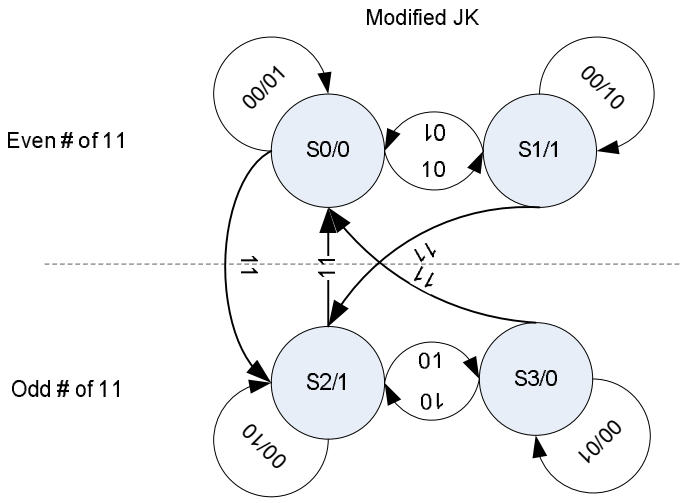


Il JK modificato si differenzia dal precedente per il fatto che il comportamento quando JK=11 varia

- 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:

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

3.1.2 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
  - 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 filp-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 *a1a0*

y_1y_0 \ jk	00	01	11	10	q
	a_1a_0				
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.

y_1y_0 \ jk	00	01	11	10	q
	$\overline{a_1}$				
00	1	1	0	1	0
01	1	1	0	1	1
11	0	0	1	0	1
10	0	0	1	0	0

y_1y_0 \ jk	00	01	11	10	q
	$\overline{a_0}$				
00	1	1	0	0	0
01	0	1	0	0	1
11	0	1	1	0	1
10	1	1	1	0	0

$$\overline{a_1} = \overline{j} \cdot \overline{y_1} + \overline{k} \cdot \overline{y_1} + j \cdot k \cdot y_1$$
$$\overline{a_0} = \overline{j} \cdot k + k \cdot y_1 + \overline{j} \cdot \overline{y_0}$$

Applicando DeMorgan si ottiene:

$$\overline{a_1} = \overline{(j + y_1)} + \overline{(k + y_1)} + \overline{(\overline{j} + \overline{k} + \overline{y_1})}$$
$$\overline{a_0} = \overline{(j + \overline{k})} + \overline{(j + y_0)} + \overline{(\overline{k} + \overline{y_1})}$$

Da cui, complementando, si ottiene la soluzione:

$$a_1 = \overline{\overline{(j + y_1)} + \overline{(k + y_1)} + \overline{(\bar{j} + \bar{k} + \bar{y}_1)}}$$
$$a_0 = \overline{\overline{(j + \bar{k})} + \overline{(j + y_0)} + \overline{(\bar{k} + \bar{y}_1)}}$$
$$q = y_0$$

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.

y_1y_0 \ $j\,k$	00	01	11	10	q
	Nuovo stato interno				
00	00	00	11	01	0
01	01	00	11	01	1
11	11	10	00	11	1
10	10	10	00	11	0

y_1y_0 \ $j\,k$	j_1k_1 00	01	11	10	q
00	0-	0-	1-	0-	0
01	0-	0-	1-	0-	1
11	-0	-0	-1	-0	1
10	-0	-0	-1	-0	0

y_1y_0 \ $j\,k$	j_0k_0 00	01	11	10	q
00	0-	0-	1-	1-	0
01	-0	-1	-0	-0	1
11	-0	-1	-1	-0	1
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.

y_1y_0 \ $j\,k$	00	$\overline{j1}$ 01	11	10	q
00	1	1	0	1	0
01	1	1	0	1	1
11	-	-	-	-	1
10	-	-	-	-	0

y_1y_0 \ $j\,k$	$\overline{k_1}$ 00	01	11	10	q
00	-	-	-	-	0
01	-	-	-	-	1
11	1	1	0	1	1
10	1	1	0	1	0

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

3.2 Esercizio – rete di Moore

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:

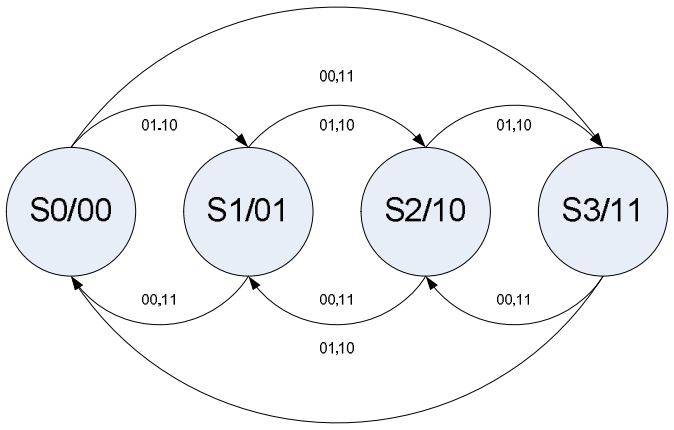
- quando gli ingressi sono *diversi*, la rete conta in avanti (modulo 4)
- quando gli ingressi sono *uguali*, la rete conta all'indietro (modulo 4)

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.

3.2.1 Soluzione

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



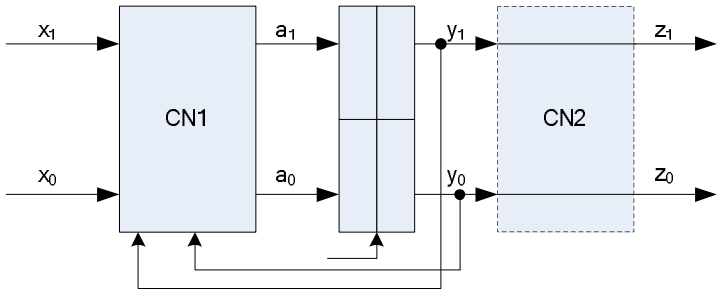
x_1x_0	00	01	11	10	z_1z_0
S0	S3	S1	S3	S1	00
S1	S0	S2	S0	S2	01
S2	S1	S3	S1	S3	10
S3	S2	S0	S2	S0	11

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:

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

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

a) dipende da tutte e quattro le variabili
b) 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.

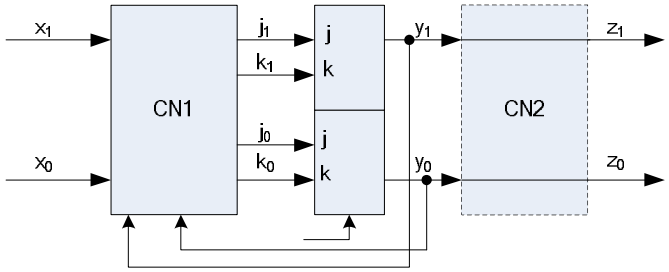
La variabile a_1 può essere riscritta come segue: $a_1 = \overline{y_1 \otimes y_0 \otimes x_1 \otimes x_0}$, dal che si ricava che, utilizzando soltanto **porte XOR e NOT**, il costo a porte è 3 e quello a diodi è 6.

Ci si arriva, volendo, per via algebrica:

$$a_1 = \overline{y_1}\cdot\overline{y_0}\cdot\left(\overline{x_1}\cdot\overline{x_0} + x_1\cdot x_0\right) + \overline{y_1}\cdot y_0\cdot\left(\overline{x_1}\cdot x_0 + x_1\cdot\overline{x_0}\right)$$
$$+ y_1\cdot\overline{y_0}\cdot\left(\overline{x_1}\cdot x_0 + x_1\cdot\overline{x_0}\right) + y_1\cdot y_0\cdot\left(\overline{x_1}\cdot\overline{x_0} + x_1\cdot x_0\right)$$
$$= \left(\overline{y_1}\cdot\overline{y_0} + y_1\cdot y_0\right)\cdot\left(\overline{x_1}\cdot\overline{x_0} + x_1\cdot x_0\right) + \left(\overline{y_1}\cdot y_0 + y_1\cdot\overline{y_0}\right)\cdot\left(\overline{x_1}\cdot x_0 + x_1\cdot\overline{x_0}\right)$$
$$= \overline{\left(y_1\otimes y_0\right)}\cdot\overline{\left(x_1\otimes x_0\right)} + \left(y_1\otimes y_0\right)\cdot\left(x_1\otimes x_0\right)$$
$$= \overline{\left(y_1\otimes y_0\right)\otimes\left(x_1\otimes x_0\right)}$$
$$= \overline{y_1\otimes y_0\otimes x_1\otimes x_0}$$

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.

Vecchio valore	Nuovo valore	jk
0	0	0-
0	1	1-
1	0	-1
1	1	-0



y_1y_0 \ x_1x_0	00	01	11	10
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_1k_1

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

j_0k_0

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

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

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

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

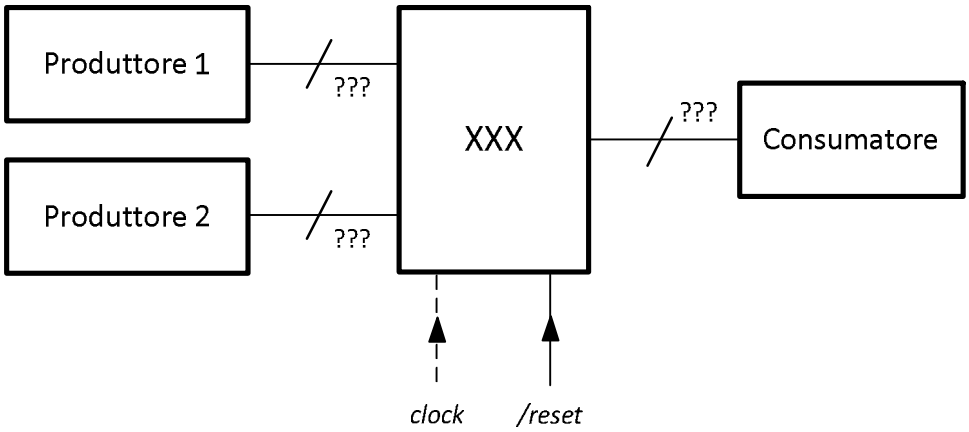
j_1

y_1y_0 \ x_1x_0	00	01	11	10
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.

3.3 Esercizio – descrizione e sintesi di RSS complessa

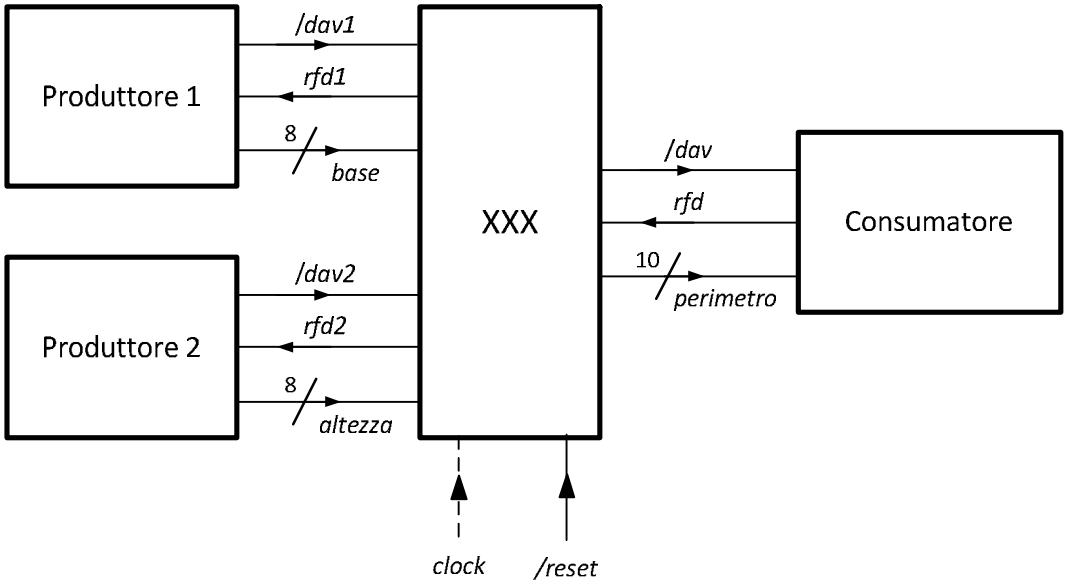


Descrivere la rete **XXX** che si evolve ciclicamente come segue: "preleva un byte dal Produttore 1 e un byte dal Produttore 2, elabora i byte ed invia il risultato della elaborazione al Consumatore." L'elaborazione viene fatta tramite una funzione *mia_rete*(*base*,*altezza*), che interpreta i byte 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

3.3.1 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.

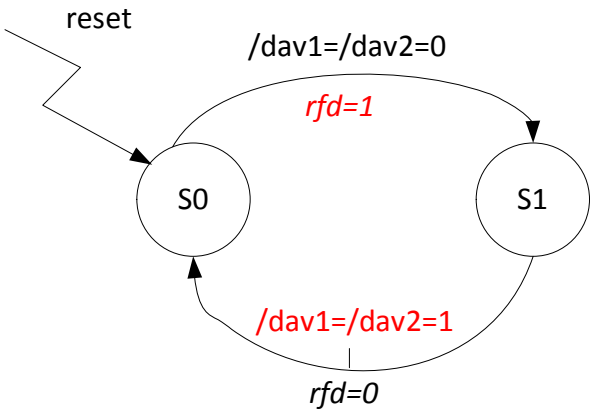
- Ne servono alcuni per sostenere le uscite: RFD1, RFD2, DAV_, a 1 bit ciascuno.
- PERIMETRO a 10 bit.
- STAR a un po' di bit (quanti saranno lo vediamo alla fine della descrizione)
- I valori *base* e *altezza* si prendono tramite handshake. Potrei volerli memorizzare da qualche parte (nel qual caso 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: */dav*1=1, */dav*2=1, *rfd*=1. **Devo settare gli output** di conseguenza: */dav*=1, *rfd*1=1, *rfd*2=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 iniziarne un'altra.

In **S0** devo:

- **tenere *rfd*1, *rfd*2, */dav* a 1;**
- aspettare che **entrambi */dav*1 e */dav*2** 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);
  ```

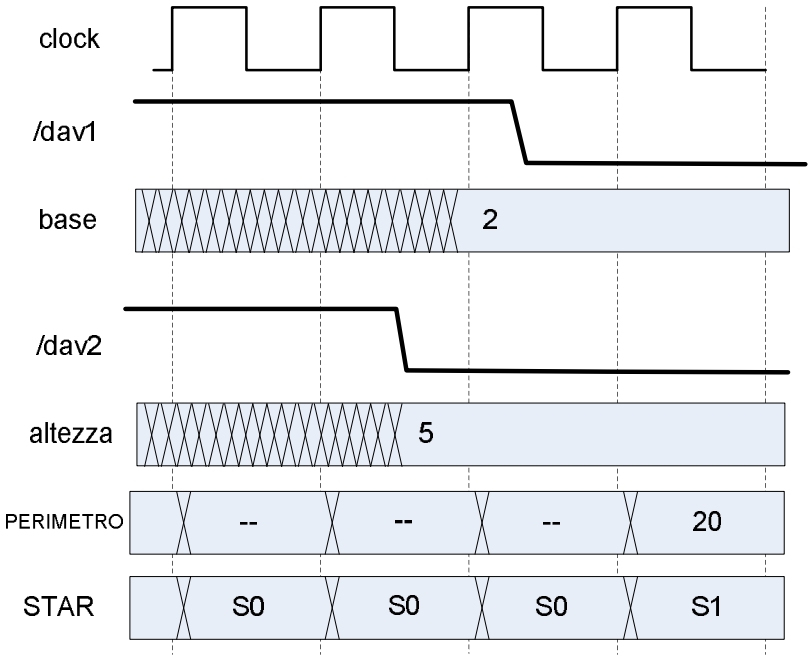
  Posso contestualmente **portare DAV_ a 0**, perché il dato di uscita è pronto, in modo da iniziare l'handshake con il consumatore.

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:**

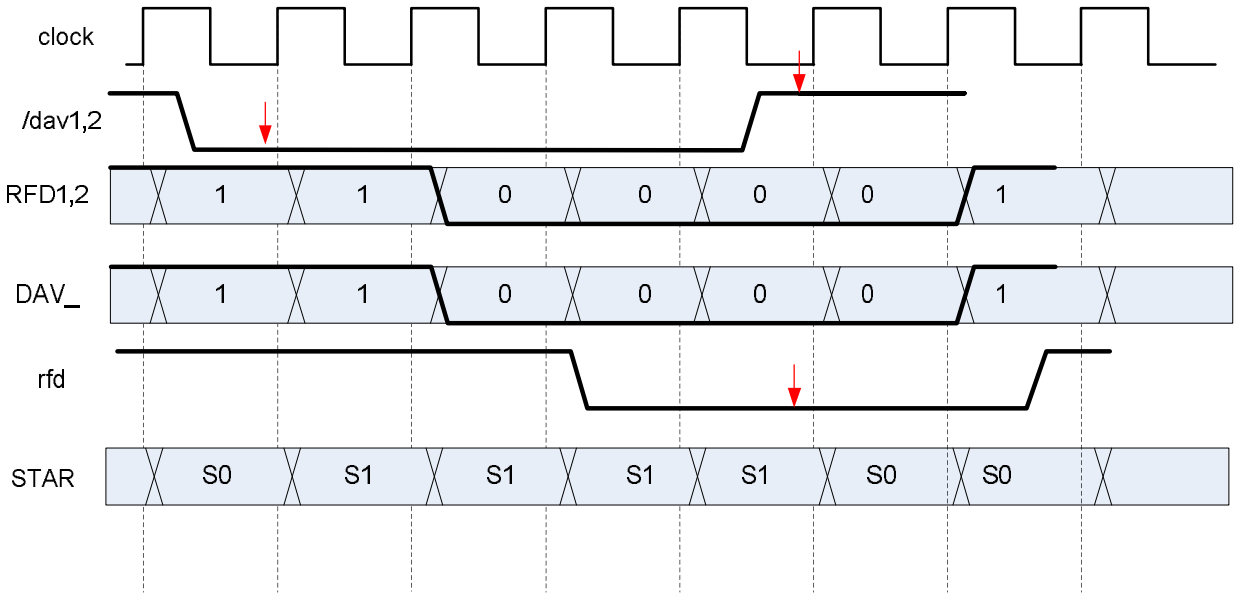
```
PERIMETRO<=mia_rete(base, altezza);
```

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
casex(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
```

3.3.2 Sintesi

La sintesi è banale: andiamo per ordine:

Registro operativo HS

```
S0: HS<=1;
S1: HS<=0;
```

Una variabile di comando b0, che vale 1 in S0 e 0 in S1.

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

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** c0 e c1:

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

Abbiamo tutto per poter scrivere la sintesi secondo il paradigma 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;
```

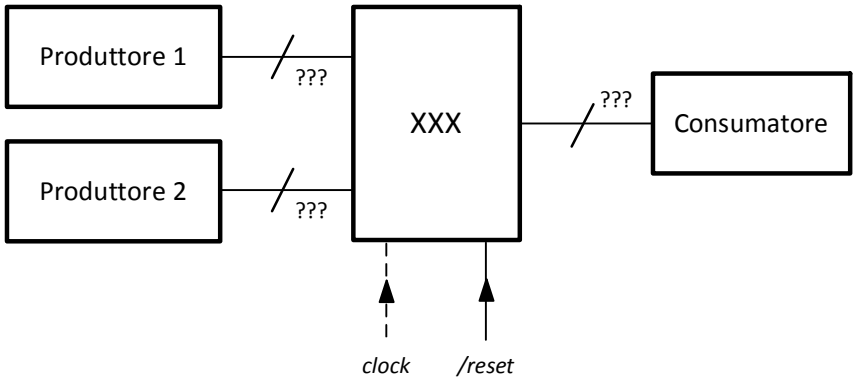
```
 wire c1,c0,b0;
 Parte_Operativa PO(base,dav1_,rfd1,altezza,dav2_,rfd2,
                    perimetro,dav_,rfd, c1,c0,b0,clock,reset_);
 Parte_Controllo PC(b0,c1,c0,clock,reset_);
endmodule

//-----------------------------------------------------------------
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:

μ-addr	**μ-code**	C_{eff}	**μ-addr T**	**μ-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 c0=j, c1=k, b0=~q.

3.4 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:*

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

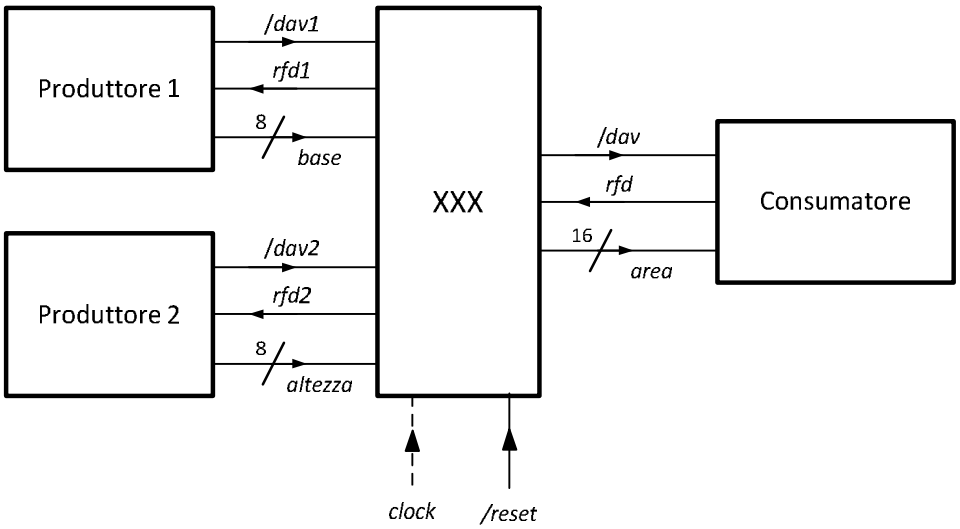
$$P_{i+1} = \left\lfloor \frac{y_i \cdot \beta^m \cdot X + P_i}{\beta} \right\rfloor$$

$P_m = P$

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

3.4.1 Descrizione

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



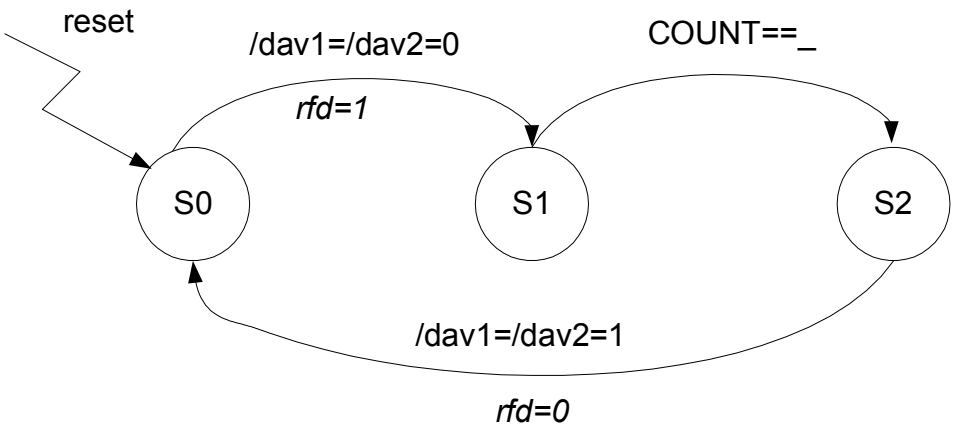
Per far girare l'algoritmo, ho $C = 0$, quindi $P_0 = 0$. Definirò una rete combinatoria `mia_rete` che sintetizza il passo iterativo dell'algoritmo.

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 */dav*1=1, */dav*2=1, *rfd*=1 (input), e **devo fare in modo che** gli output siano consistenti: */dav*=1, *rfd*1=1, *rfd*2=1.

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



In **S0** devo:

- campionare *base, altezza*
- tenere *rfd*1, *rfd*2, */dav* a 1
- aspettare che */dav*1 **e** */dav*2 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 semplice:

se inizializzo a *k* e testo a *j* (<=*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 inizializzarlo 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
 casex(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
```

3.4.2 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; end
S1: STAR<=(COUNT==0)?S2:S1; end
S2: STAR<=({dav1_,dav2_,rfd}=='B110)?S0:S2; end
```

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
  casex({b1,b0})
   'b00: HS<=1;
   'b01: HS<=HS;
   'b10: HS<=0;
  endcase

[…]

//Registro COUNT
 always @(reset_==0) #1 COUNT<=7;
 always @(posedge clock) if (reset_==1) #3
  casex(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
  casex(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:

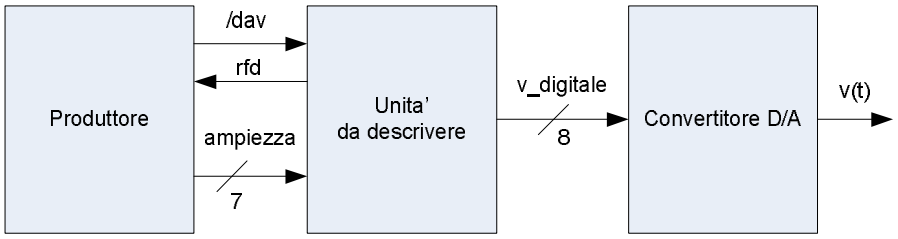
μ-addr	**μ-code** b_1b_0	C_{eff}	**μ-addr T**	**μ-addr F**
00 (S0)	00	00	01 (S1)	00 (S0)
01 (S1)	01	01	10 (S2)	01 (S1)
10 (S2)	10	10	00 (S0)	10 (S2)

3.5 Esercizio – tensioni analogiche

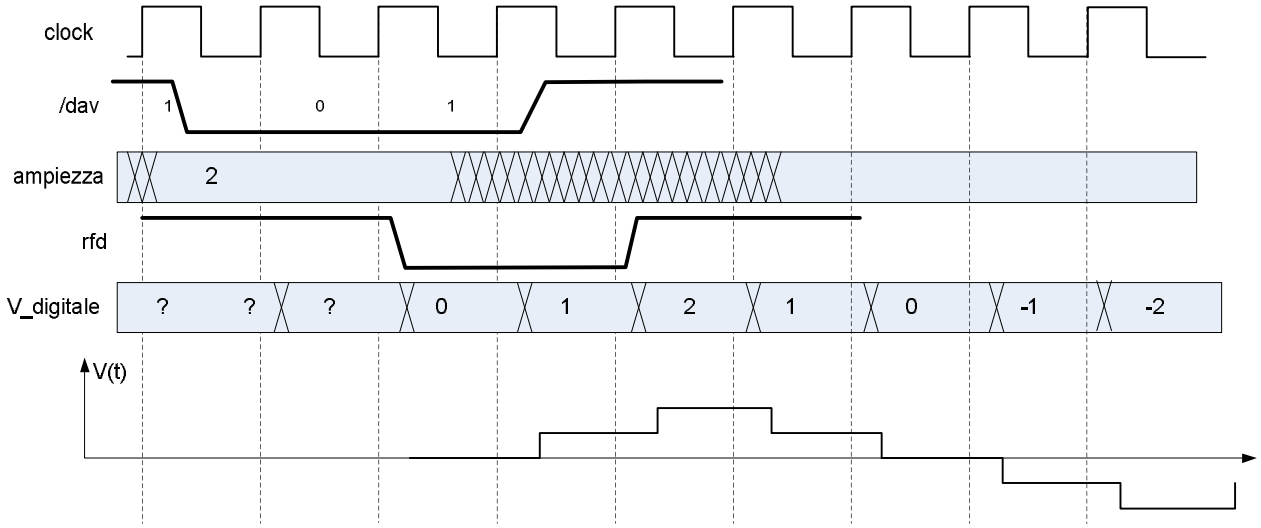
Descrivere l'unità di figura che opera ciclicamente nelle seguenti ipotesi:

- 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:



3.5.1 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 $k = -\left(2^7 - 1\right), ..., -1, 0, 1, ..., +\left(2^7 - 1\right)$, cioè $K = 2^7 + \left\{-\left(2^7 - 1\right), ..., -1, 0, 1, ..., +\left(2^7 - 1\right)\right\}$.

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 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)?...
```