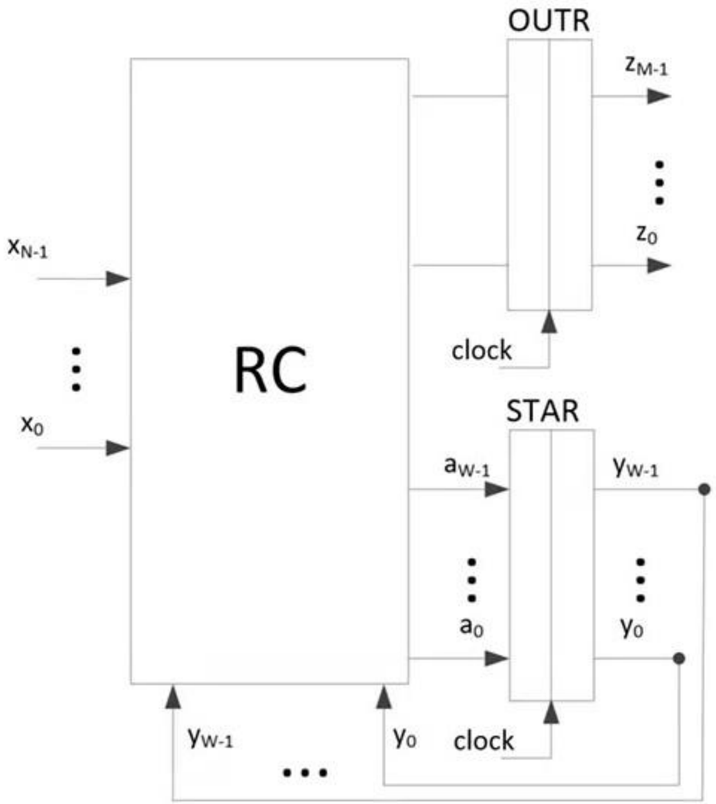
Materiale didattico

- Paolo Corsini, "Dalle porte AND, OR, NOT al Sistema calcolatore", edizioni ETS

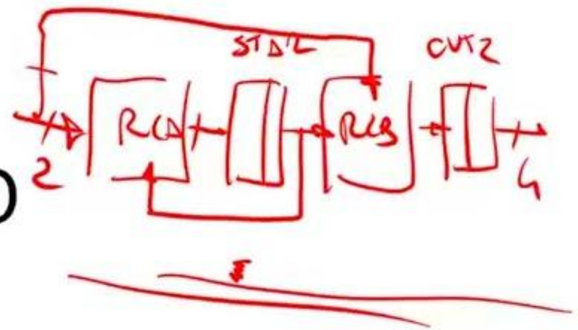
- Appunti sulle Reti Sequenziali - versione 25/09/2020 (con esercizi svolti), cap. 4-5

Descrizione e sintesi di RSS complesse

- I tre modelli di RSS visti finora 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.



Limiti del modello di Mealy Ritardato



- Supponiamo di voler descrivere, usando una rete di Mealy ritardato, una rete che **conta, modulo 16**, **il numero di sequenze corrette 00, 01, 10 ricevute** in ingresso. 5 ∞
- Ogni volta che vede una sequenza corretta incrementa di 1 l'uscita, rappresentata su 4 bit.
- Questa rete ha **due ingressi**, **quattro uscite**, e **quanti stati interni**?
  - 3 stati interni per riconoscere una sequenza di 3 passi
  - OUTR può solo memorizzare valori che sono funzioni di STAR (e dell'ingresso)
  - Quindi ci vogliono almeno $3 \cdot 16 = 48$ stati interni
  - STAR a 6 bit
  - Descrizione Verilog farraginosa (`casex`…`endcase` a 48 righe)

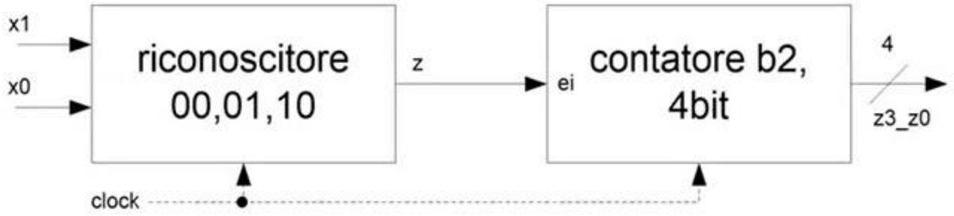
Linguaggio di trasferimento tra registri (cont.)

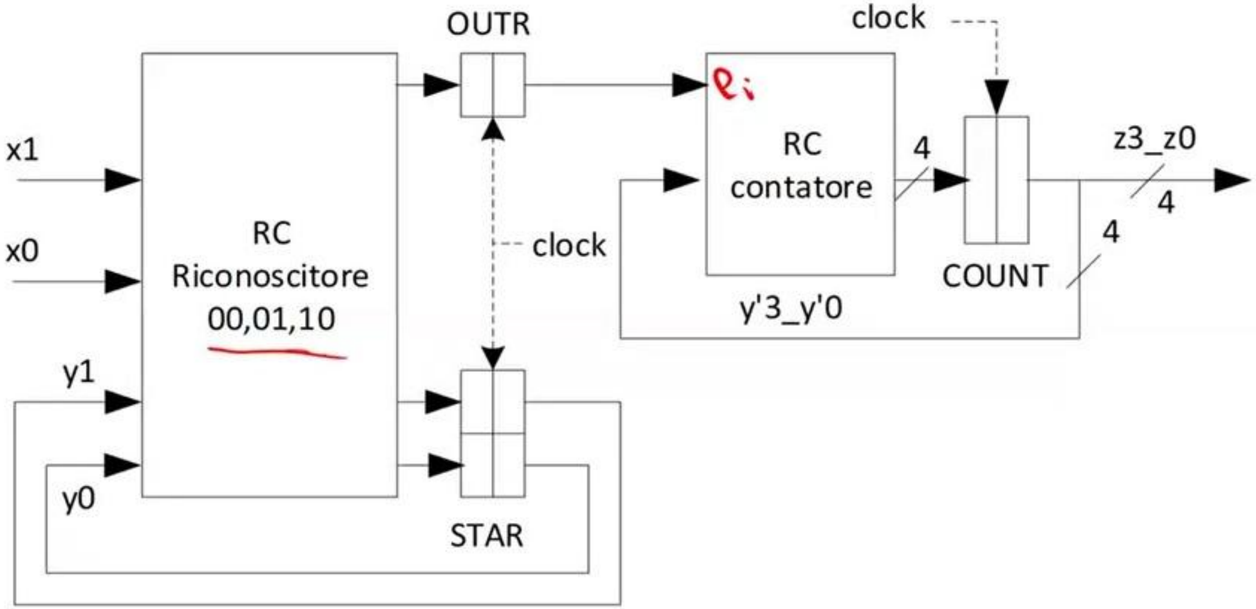
- Se, invece, ometto di specificare l'assegnamento **al registro di stato STAR**, è sottinteso che il **μ-salto è incondizionato, e porta allo statement successivo** nella descrizione.
- Non potrebbe essere altrimenti, perché se fosse `STAR<=STAR` si avrebbe un **deadlock**
- **Non ometteremo mai l'aggiornamento di STAR**
  - farlo diminuisce la leggibilità ed è fonte di errori

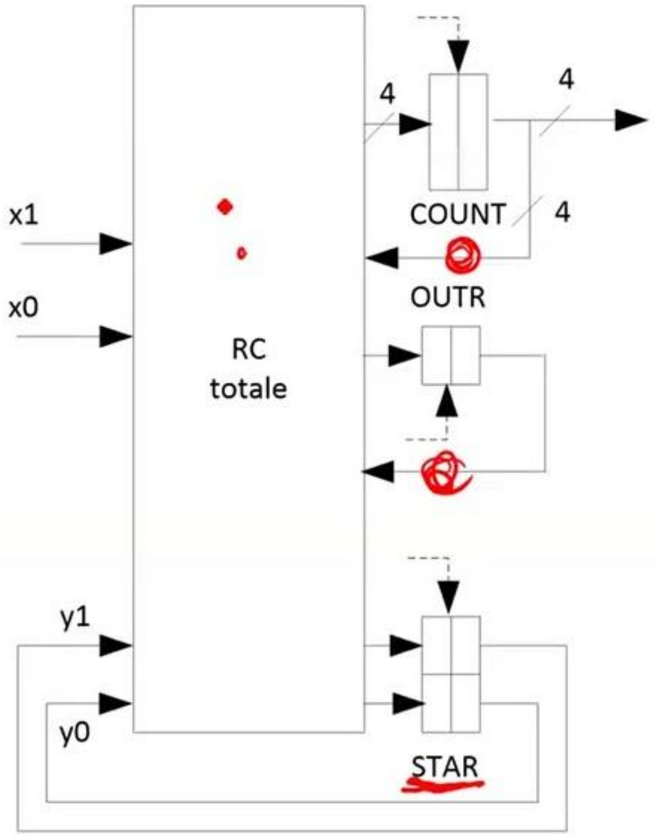
```
S0 : begin COUNT<=COUNT; STAR<=(x1_x0=='B00)?S1:S0; end

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
```

Sintesi euristica del contatore di sequenze

- Se non considero **il riporto uscente** (ininfluente per i nostri scopi), il contatore **è una rete di Mealy ritardato**
  - il numero in base 2 su 4 bit esce da un registro

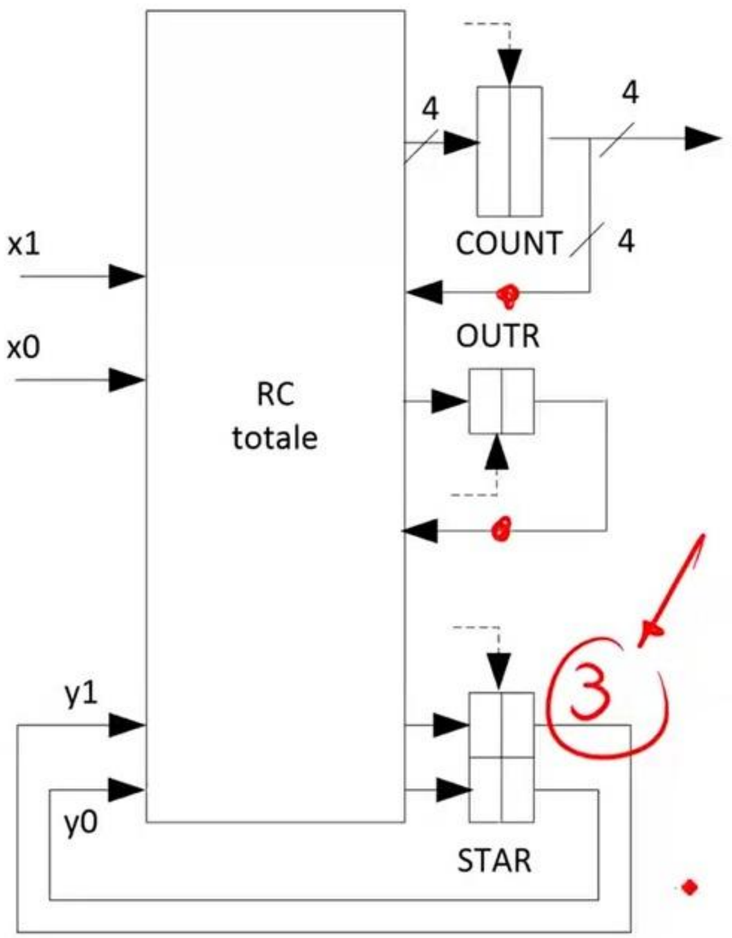






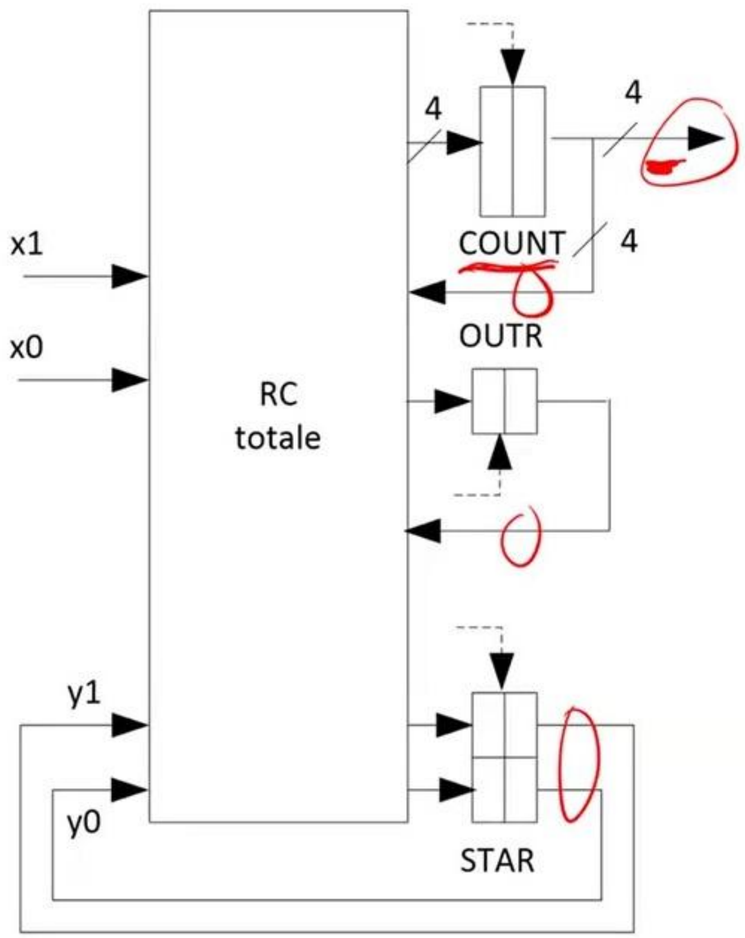
Un nuovo modello di sintesi

- Questa **non è una rete di Mealy ritardato**
- Non ha soltanto due registri
- Sto facendo **rientrare le uscite di altri registri**
  - Oltre STAR
  - E questo è fuori dal modello di Mealy ritardato
  - Ma fa **molto** comodo: registri **operativi** (COUNT, OUTR) supportano sia uscite che computazioni intermedie
  - Le descrizioni vengono molto più semplici



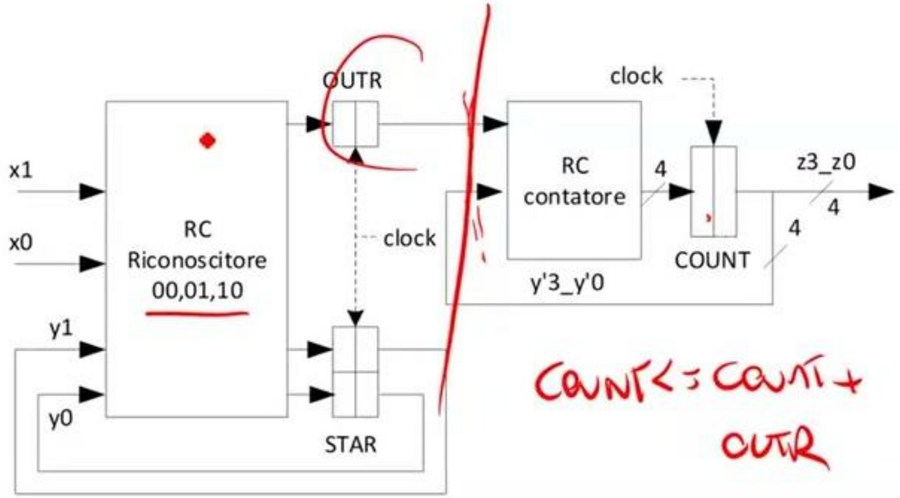
Modello più generale

- Registro di **stato STAR**, che svolge le funzioni che conosciamo
- **Registri operativi**
  - Tanti quanti me ne servono
  - 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
  - non necessariamente un registro operativo deve per forza sostenere un'uscita



Descrizione in Verilog

```
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
```

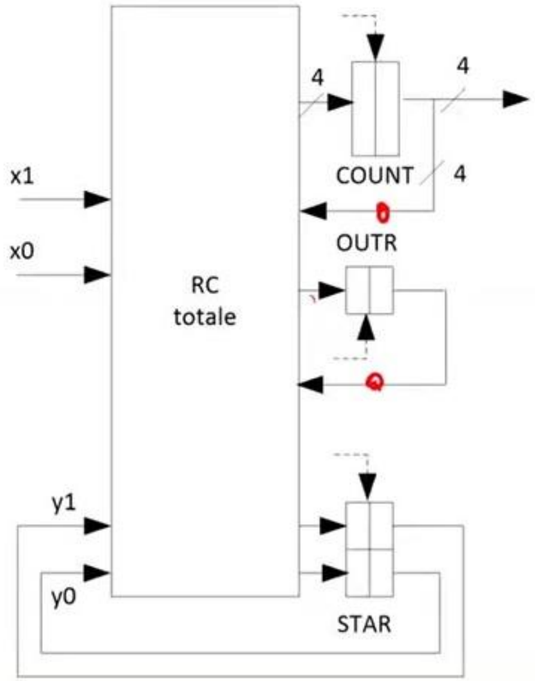


COUNT<=COUNT+OUTR

Descrizione in Verilog

```
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
```

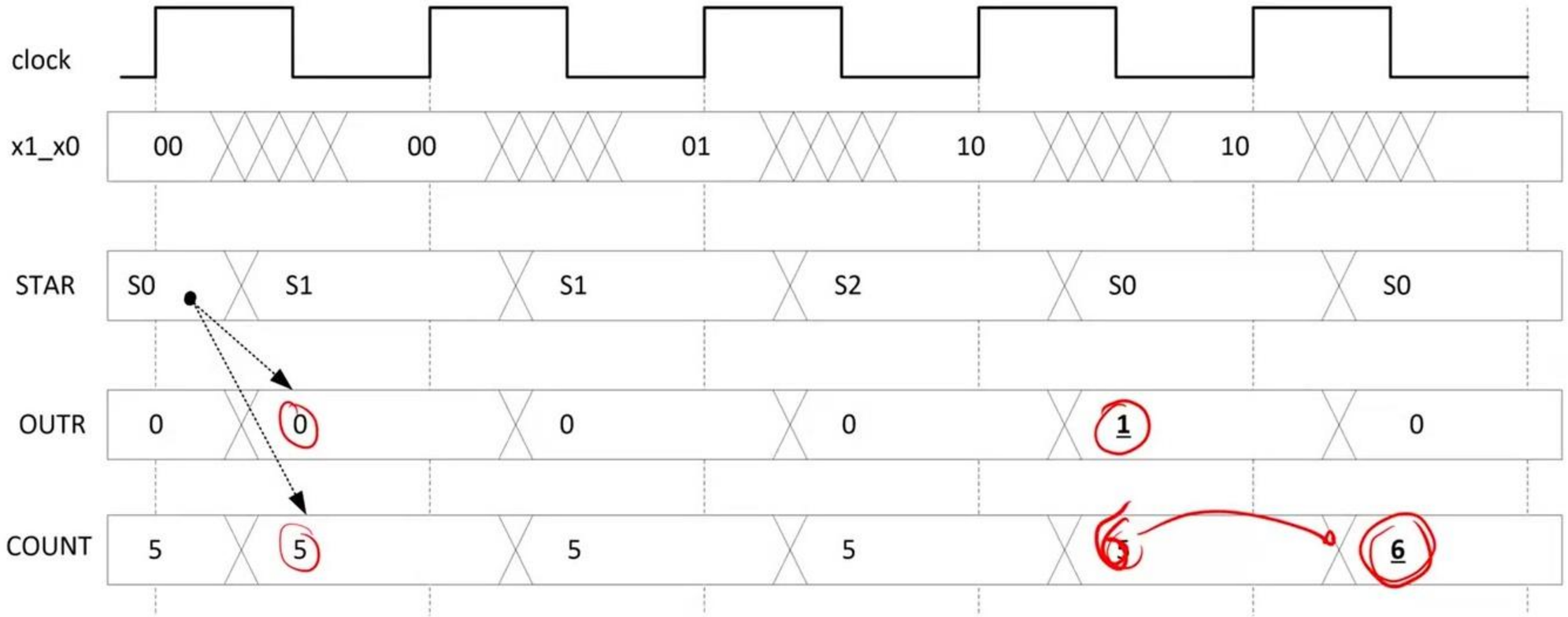
In questo nuovo modello, anche i registri operativi possono stare **a destra di un assegnamento procedurale**



Temporizzazione della rete

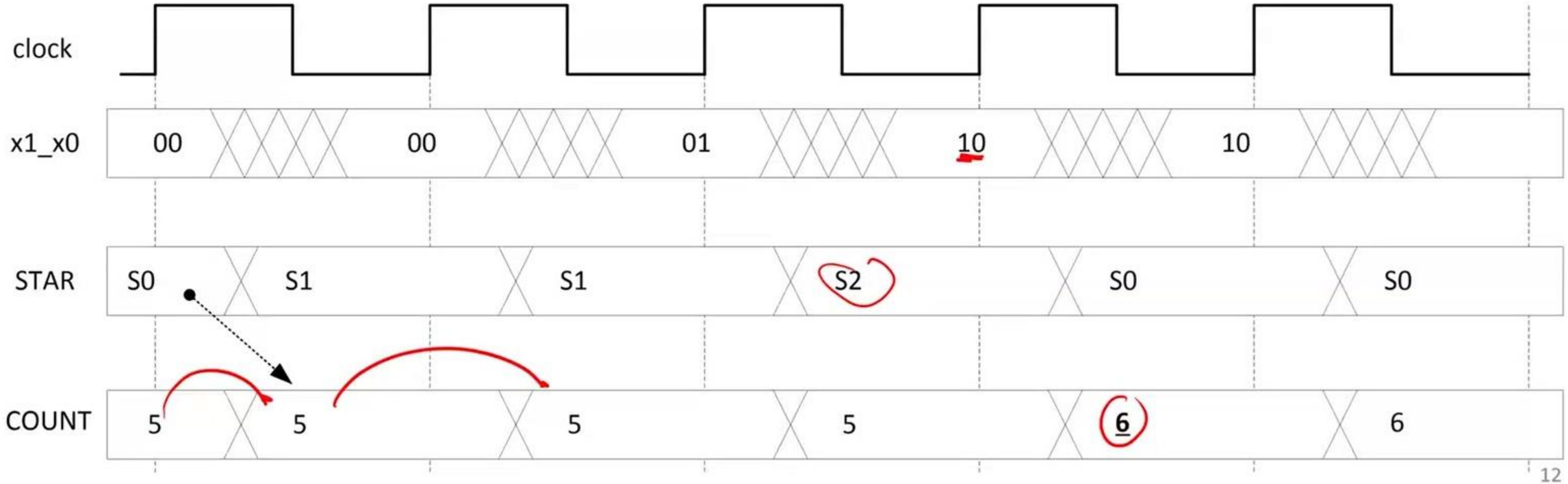
```
S0 : begin OUTR<=0; COUNT<=COUNT+OUTR; STAR<=(x1_x0=='B00)?S1:S0; end

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
```

COUNT <= (x1_x0 == 'B10)? COUNT+1 : COUNT ;



Temporizzazione della rete - ottimizzata

```
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
```



Linguaggio di trasferimento tra registri

software

- Il **Verilog comprende**, tra mille altre cose, un linguaggio di trasferimento tra registri
- Ogni ramo del `casex` si chiama **statement**, e comprende:
  - Zero o più **μ-istruzioni**, cioè assegnamenti a registri operativi
  - Un **μ-salto**, cioè un assegnamento al registro **`STAR`.**
    - a **due vie**, come in S_0, S_2, a **più vie**, come in S_1,
    - **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
```

μ-istruzioni:
μ- sta per
«hardware»,
non per «piccolo»

Linguaggio di trasferimento tra registri (cont.)

- Posso omettere di specificare il comportamento di un **registro operativo** (attenzione: **operativo**). In questo caso, è come se scrivessi:

```
REGISTRO<=REGISTRO;
```

- Ad esempio, potrei omettere di scrivere l'aggiornamento di COUNT in S_0, S_1
  - è quello che faremo d'ora in poi

```
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
```

Linguaggio di trasferimento tra registri (cont.)

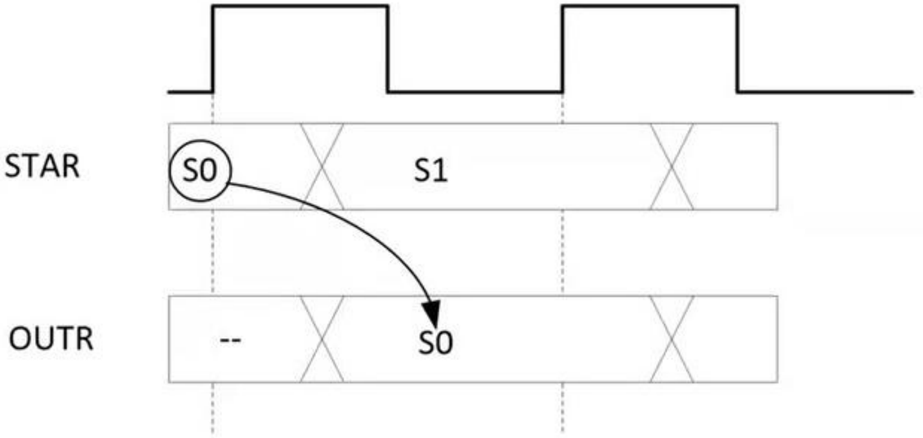
- Se, invece, ometto di specificare l'assegnamento **al registro di stato STAR**, è sottinteso che il **μ-salto è incondizionato, e porta allo statement successivo** nella descrizione.
- Non potrebbe essere altrimenti, perché se fosse `STAR<=STAR` si avrebbe un **deadlock**
- **Non ometteremo mai l'aggiornamento di STAR**
  - farlo diminuisce la leggibilità ed è fonte di errori

```
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
```

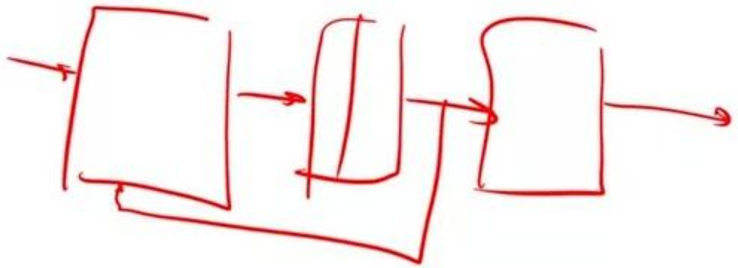
Vincoli di temporizzazione

- Le stesse disequazioni di una rete di Mealy Ritardato
  - I percorsi sono gli stessi: in->reg, reg->reg, e le uscite sono prese dai registri
- 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**.

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



…e la sintesi?



- Stiamo parlando di **modalità di <u>descrizione</u>** di una RSS complessa

- Il punto di arrivo del nostro lavoro dovrà essere la **sintesi** della medesima
  - 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.
  - La **descrizione** è la parte complicata. La sintesi è automatica

Esercizio: contatore di sequenze alternate

- Consideriamo due sequenze di tre passi: 00,01,10 e 11,01,10
- Descrivere una rete che **incrementa** un contatore a 4 bit
  - quando riconosce la **prima** delle due sequenze
  - poi quando vede la **seconda**,
  - poi di nuovo la prima, e così via in modo alternato

- Variazione sul tema del precedente
  - Le due sequenze differiscono soltanto per il primo passo
  - (ma l'esercizio non è significativamente più complicato se le sequenze sono arbitrarie, provateci per casa)

Esercizio (cont.)

- Due sequenze di tre passi: 00,01,10 e 11,01,10
- Il numero di sequenze contate sta nel registro COUNT
  - Come primo passo devo riconoscere
    - $x_1 x_0 = 00$ se COUNT è pari,
    - $x_1 x_0 = 11$ se COUNT è dispari

- Posso descrivere facilmente una **rete combinatoria** che lo fa

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

Descrizione della precedente rete

00, 01, 10

S1

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

  always @(reset_ ==0) #1 begin COUNT<=0; STAR<=S0; end
  always @(posedge clock) if (reset_==1) #3
   casex(STAR)
     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
   endcase
endmodule
```

Esercizio (cont.)

- Posso modificare la descrizione precedente come segue

```
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
```

```
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
```

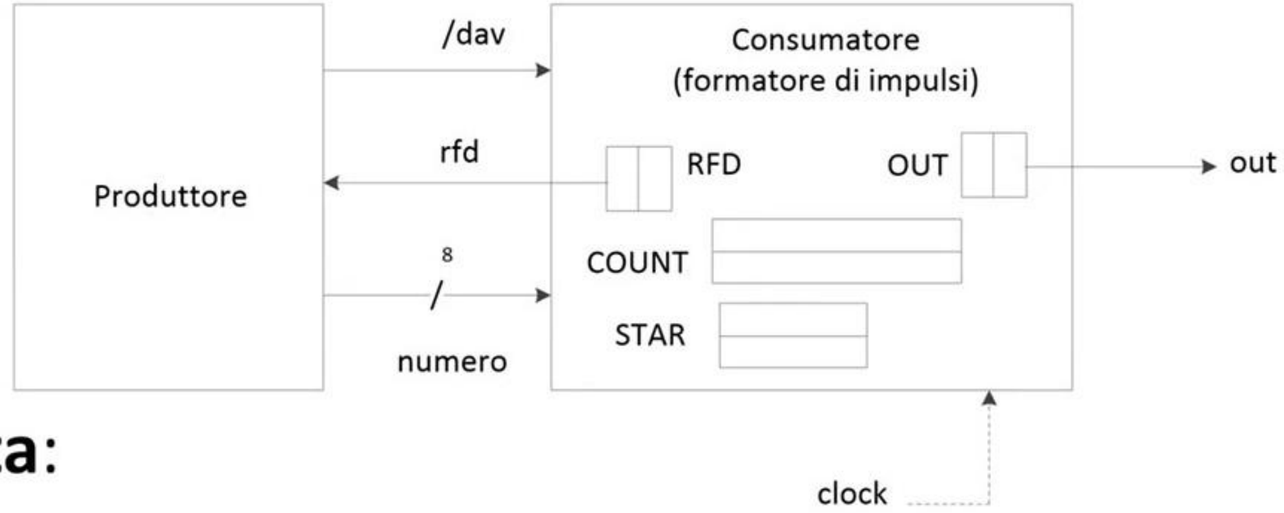
COUNT[0], x_1, x_0	match
000	1
111	1
Others	0

Esercizio (cont.)

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

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

  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
```

Esercizio (cont.)



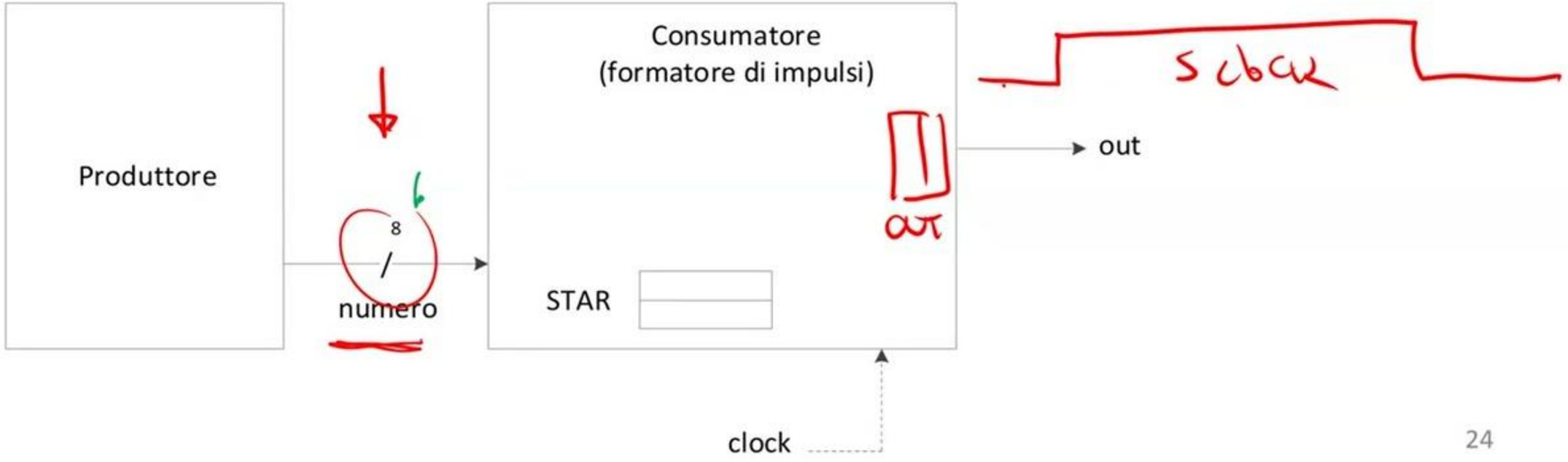
- Facciamo il punto di cosa ci serve
- **un registro** per ogni **variabile di uscita**:
  - **OUT a 1 bit, RFD a 1 bit**.
  - **COUNT**, per contare quanti cicli di clock mancano a riportare a 0 *out*.

bit?

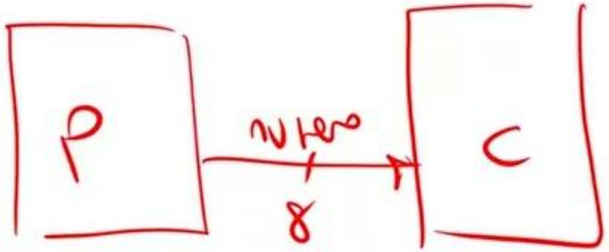
Esercizio (cont.)

- Il **produttore** produce **numeri naturali in base 2 su 8 bit**.
- Il **consumatore** prende in ingresso questi numeri, e tiene una linea di uscita `out` a 1 per il numero di clock specificato dal numero.
  - Se il numero vale 0, allora terrà l'uscita ad 1 per **256 cicli di clock**.
- Dobbiamo descrivere il **consumatore**

HW



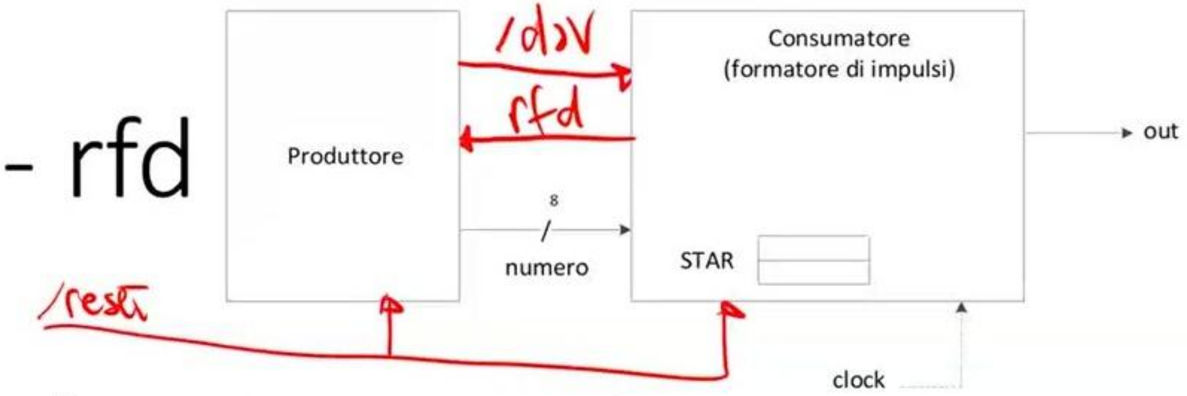
Esercizio - Sincronizzazione



- Come fanno a mettersi d'accordo?
- **Consumatore**: come fa a sapere che c'è un dato nuovo in ingresso?
  - Potrebbe essere che il produttore mette **due volte lo stesso numero N** in uscita
  - Non ho modo di distinguere questo caso dal permanere del valore **N**
- **Produttore**: come fa a sapere che il consumatore è pronto a ricevere un nuovo dato?
  - Potrebbe essere sempre occupato a terminare la gestione del dato precedente

- Questo problema si presenta **sempre** quando due reti devono comunicare
- Si risolve aggiungendo **linee di handshake** per la sincronizzazione

Esercizio - Handshake /dav - rfd

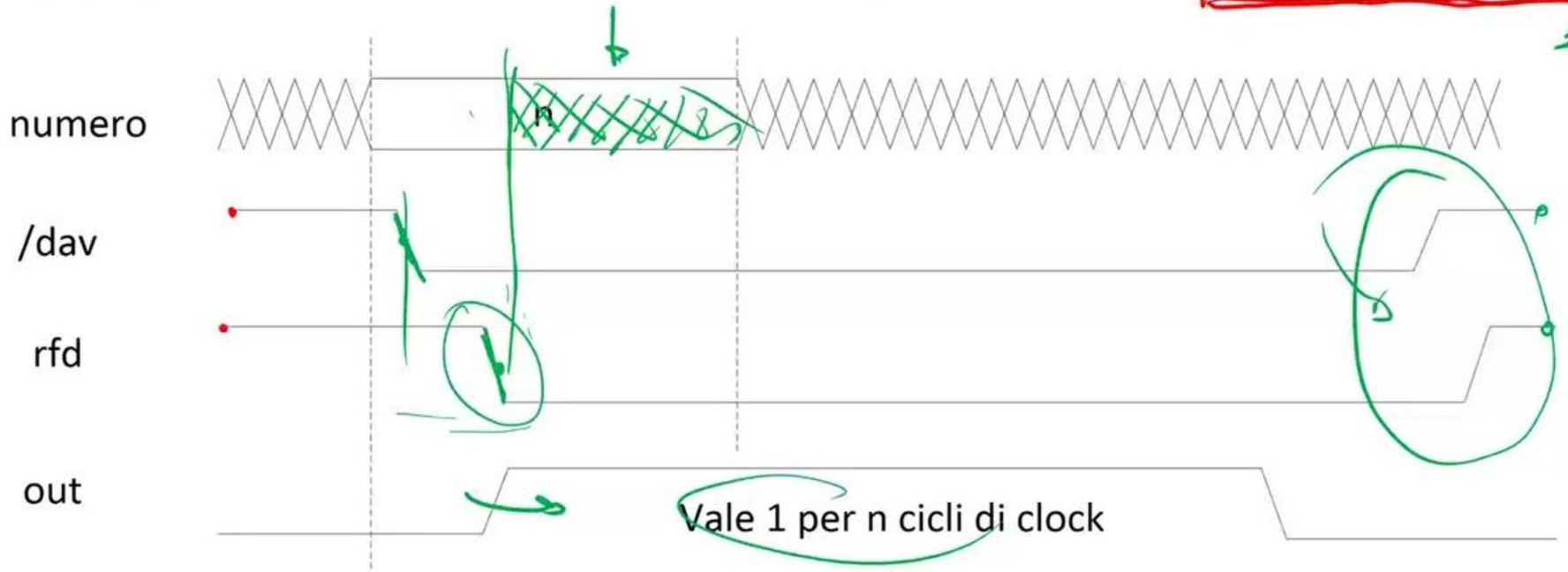


- **Due fili in più** che consentono:
  - Al <u>produttore</u> di segnalare al consumatore che è in arrivo un nuovo dato
  - Al <u>consumatore</u> di segnalare al produttore la capacità di accettare nuovi dati

deadlock

- */dav* e *rfd* (**data valid** e **ready for data**), che hanno **transizioni alternate**

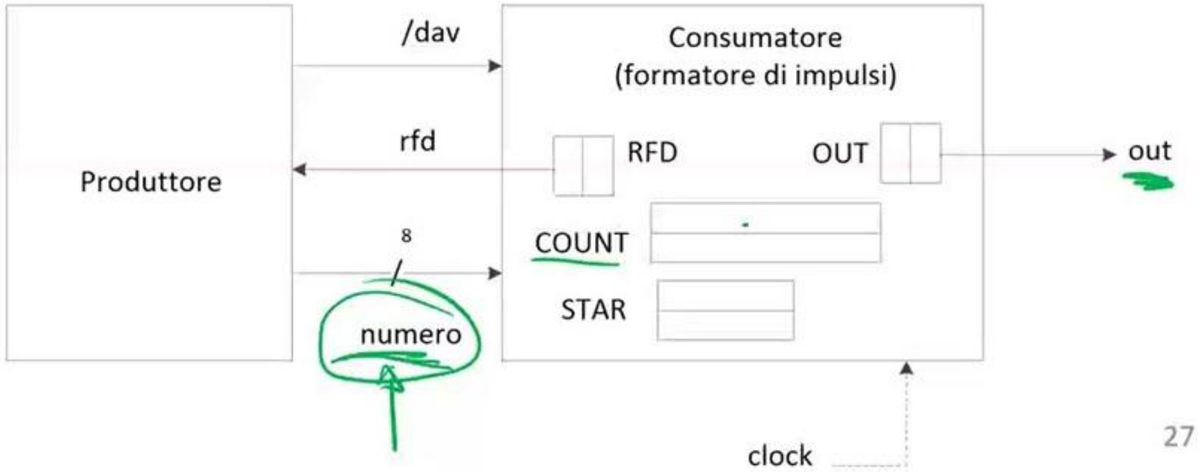
reset



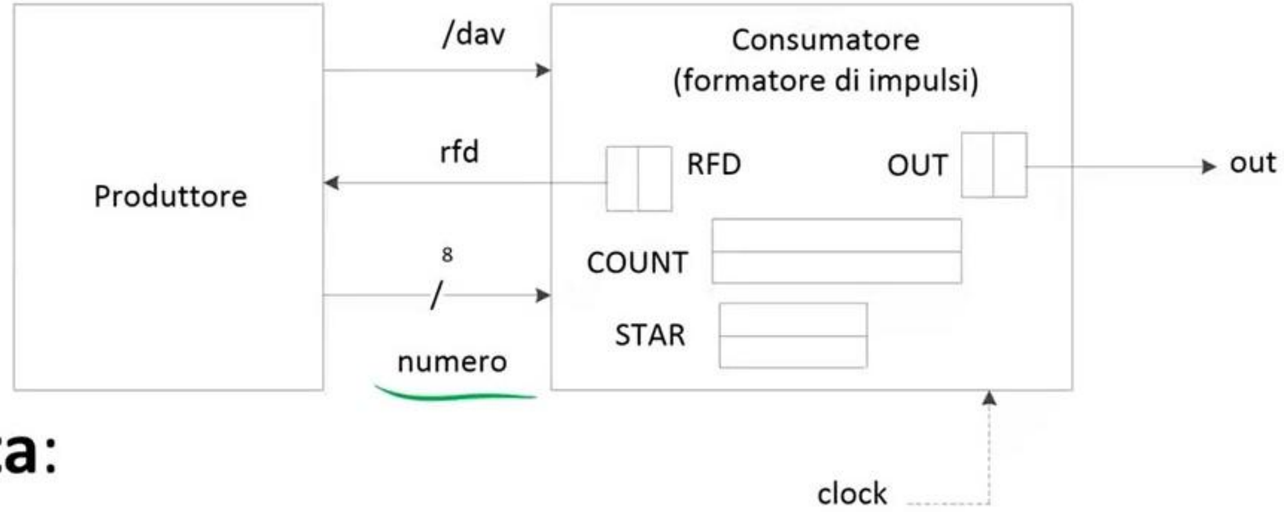
Esercizio - Temporizzazione dell'uscita

- Come si fa a fare in modo che l'uscita stia ad 1 per un dato numero di clock?
- Si prende
  - un registro a 1 bit (OUT) che deve sostenere l'uscita, e gli assegniamo il valore 1 dopo aver letto *numero*.
  - un **registro contatore**, COUNT che inizializziamo ad un certo valore (**parente di *numero***), e lo **decrementiamo ad ogni clock**. Quando arriva a 0 (o ad 1, vediamo), saranno trascorsi un certo numero di cicli di clock. Allora metteremo nuovamente OUT a 0.

approx successive

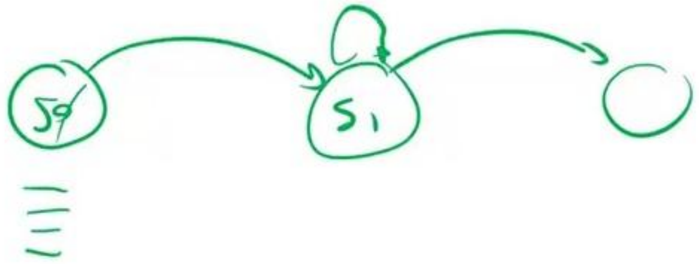


Esercizio (cont.)



- Facciamo il punto di cosa ci serve
- **un registro** per ogni **variabile di uscita**:
  - **OUT a 1 bit, RFD a 1 bit**.
  - **COUNT**, per contare quanti cicli di clock mancano a riportare a 0 *out*. bit?
    - **8 bit?** 9 bit?
  - un registro di stato **STAR**, che tiene traccia dell'evoluzione della rete. A quanti bit?
    - Dipende da come scrivo la descrizione.

M...

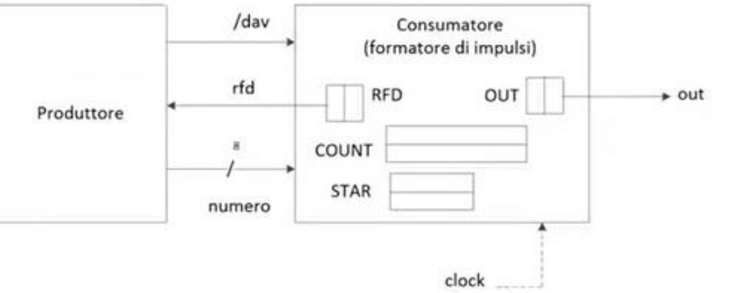


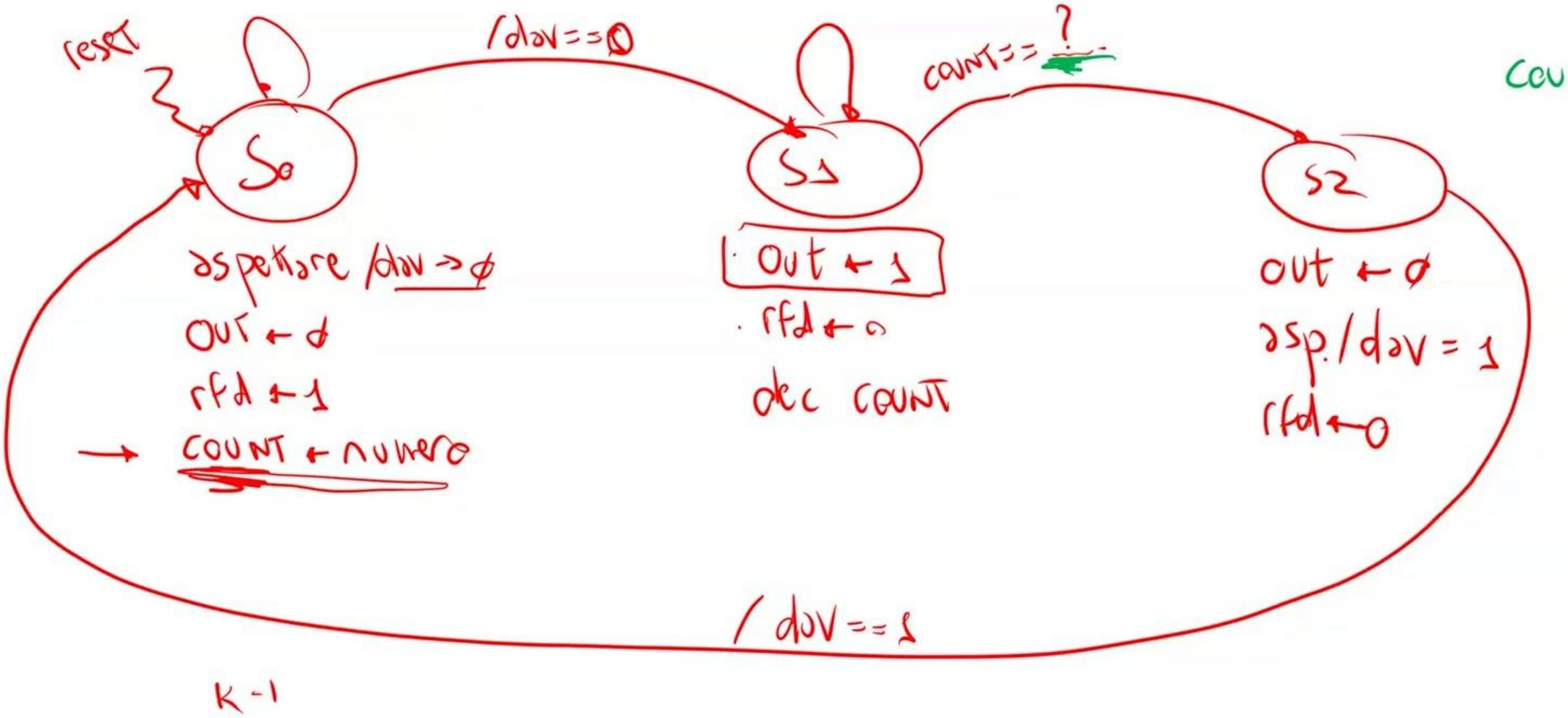
- È necessario pensare ad un **diagramma a stati**
  - In ogni stato posso assegnare valori ai registri **operativi**
  - Salto da uno stato all'altro a seconda di **condizioni di salto**

Esercizio (cont.)

reset - hs a riposo
- out = 0

/dav = 1 ; rfd = 1 out = 0
STAR = S0





COUNT ?

STAR 2 bit

numero = 0 ?

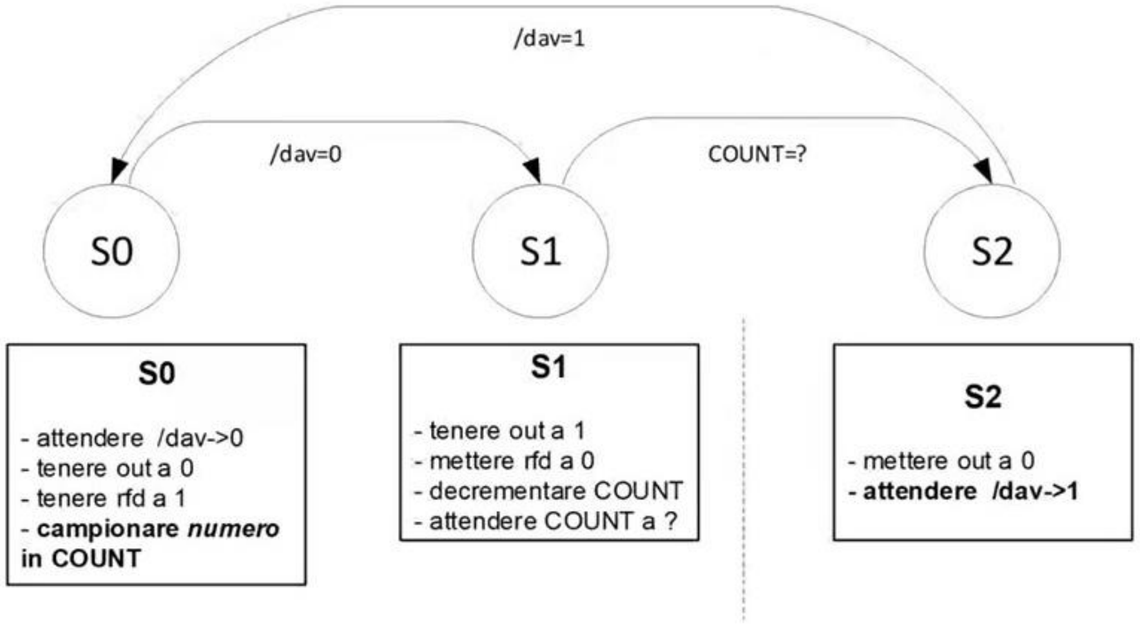
K-1

Esercizio (cont.)

Ipotesi al reset:

produttore
- /dav=1

consumatore:
- out=0
- rfd=1




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

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

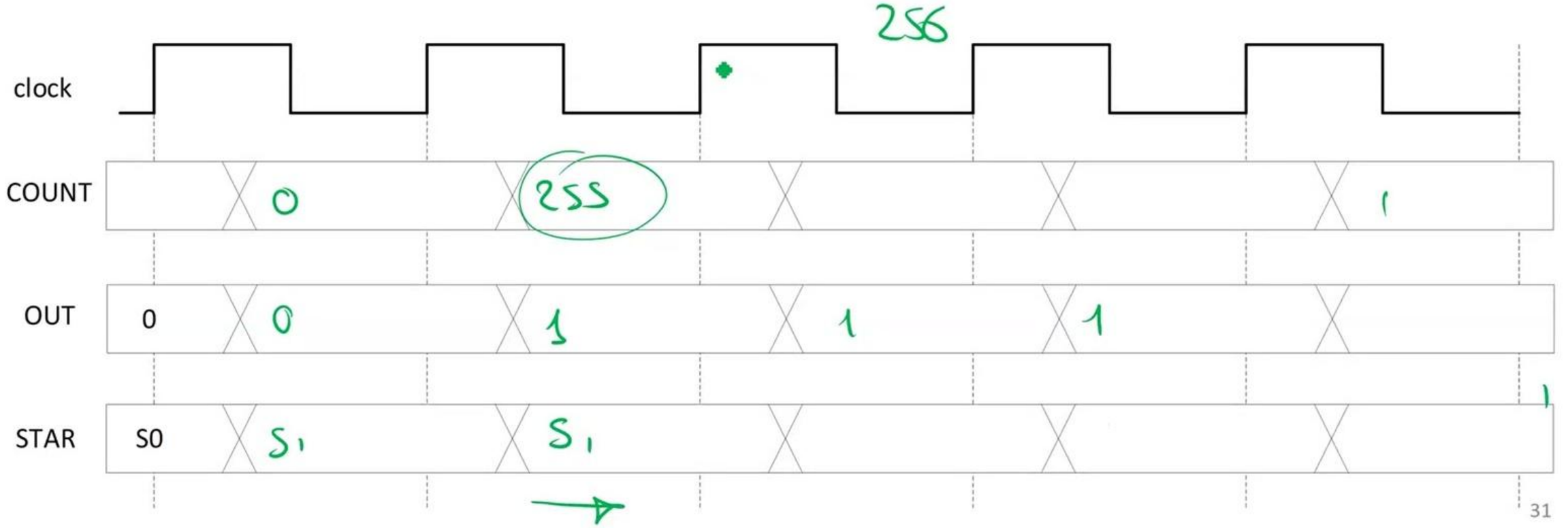
Esercizio (cont.) - uscita da S1

```
numero = 2
/dav va a 0 subito
```

```
S0: begin RFD<=1; OUT<=0; COUNT<=numero; STAR<=(dav_==1)?S0:S1; end
S1: begin RFD<=0; OUT<=1; COUNT<=COUNT-1; STAR<=(COUNT==1)?S2:S1; end
S2: begin RFD<=0; OUT<=0; STAR<=(dav_==1)?S0:S2; end
```

- Serve un diagramma di temporizzazione

3 clock numero ≠1 ≠1



Esercizio (cont.) - uscita da S1

```
numero = 2
/dav va a 0 subito
```

```
S0: begin RFD<=1; OUT<=0; COUNT<=numero; STAR<=(dav_==1)?S0:S1; end
S1: begin RFD<=0; OUT<=1; COUNT<=COUNT-1; STAR<=(COUNT==1)?S2:S1; end
S2: begin RFD<=0; OUT<=0; STAR<=(dav_==1)?S0:S2; end
```

- Serve un diagramma di temporizzazione

3 clock numero ↑ +1

Se:

- in uno stato S_{init} inizializzo un registro a $\boldsymbol{k}$
- in uno stato S_{test} lo decremento e per uscire testo se il suo valore è pari a $\boldsymbol{j}$ $(\leq \boldsymbol{k})$,

il numero di **cicli** nello stato S_{test} è $\boldsymbol{k - j + 1}$.

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

k-j+1

Esercizio (cont.)

```
S0: begin RFD<=1; OUT<=0; COUNT<=numero; STAR<=(dav_==1)?S0:S1; end
S1: begin RFD<=0; OUT<=1; COUNT<=COUNT-1; STAR<=(COUNT==_1_)?S2:S1; end
S2: begin RFD<=0; OUT<=0; STAR<=(dav_==1)?S0:S2; end
```

- Che succede quando numero=0?
  - Da S1 si passa comunque (quindi OUT va ad 1)
  - Al primo decremento COUNT vale 255 (è un registro a 8 bit)
  - Quindi farò **altri 255** cicli

Note sulla descrizione

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

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

NESSUNO VI HA ORDINATO DI FARE LE GARE

Note sulla descrizione (cont.)

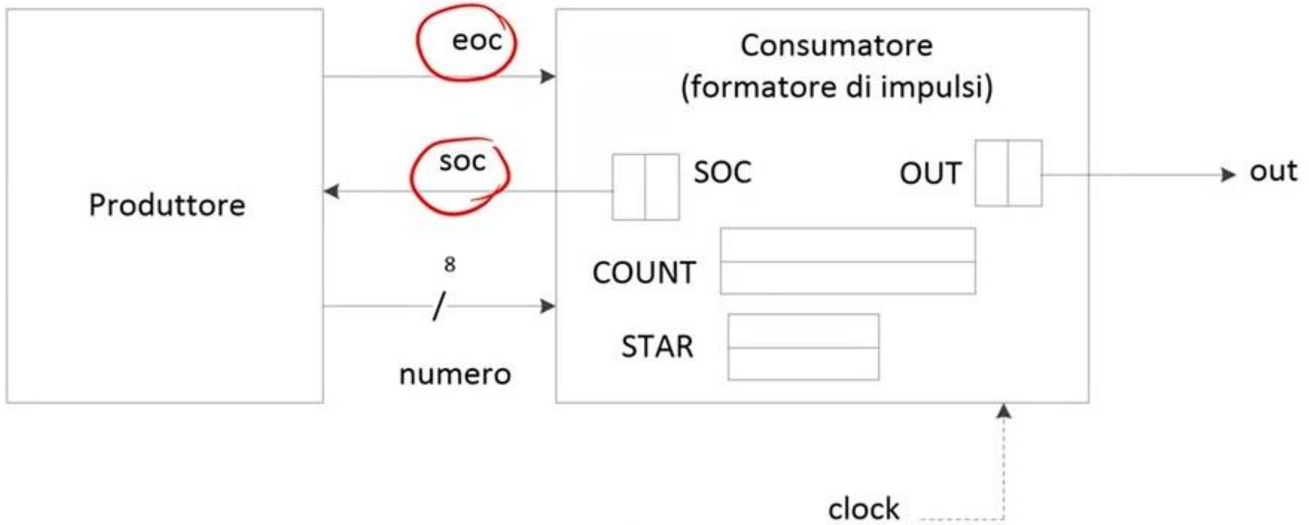
- Se la specifica fosse stata: "**se *numero*=0, OUT non deve andare a 1**", cosa avrei dovuto scrivere?

```
S0: begin RFD<=1; OUT<=0; COUNT<=numero; STAR<=(dav_==1)?S0:S1; end
S1: begin RFD<=0; OUT<=1; COUNT<=COUNT-1; STAR<=(COUNT==1)?S2:S1; end
S2: begin RFD<=0; OUT<=0; STAR<=(dav_==1)?S0:S2; end
```

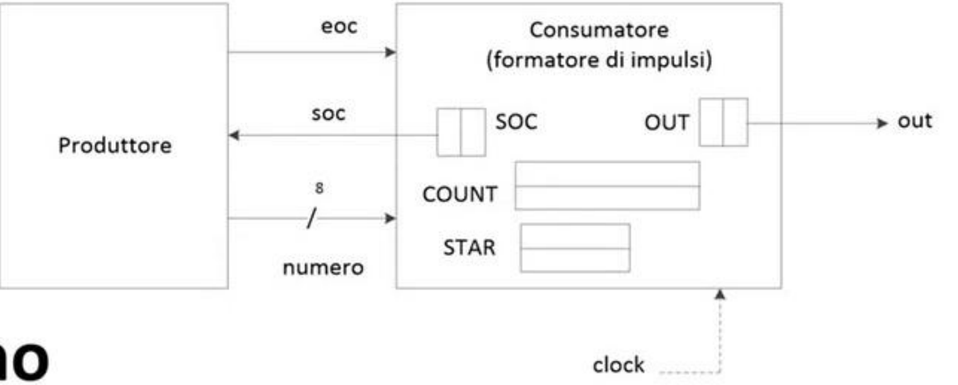
```
S1: begin  RFD<=1; OUT<=(numero==0)?0:1; COUNT<=COUNT-1;
                   STAR<=((COUNT==1)|(numero==0))?S2:S1; end
S2: begin  RFD<=0; OUT=0;…
```

Esempio – handshake soc-eoc

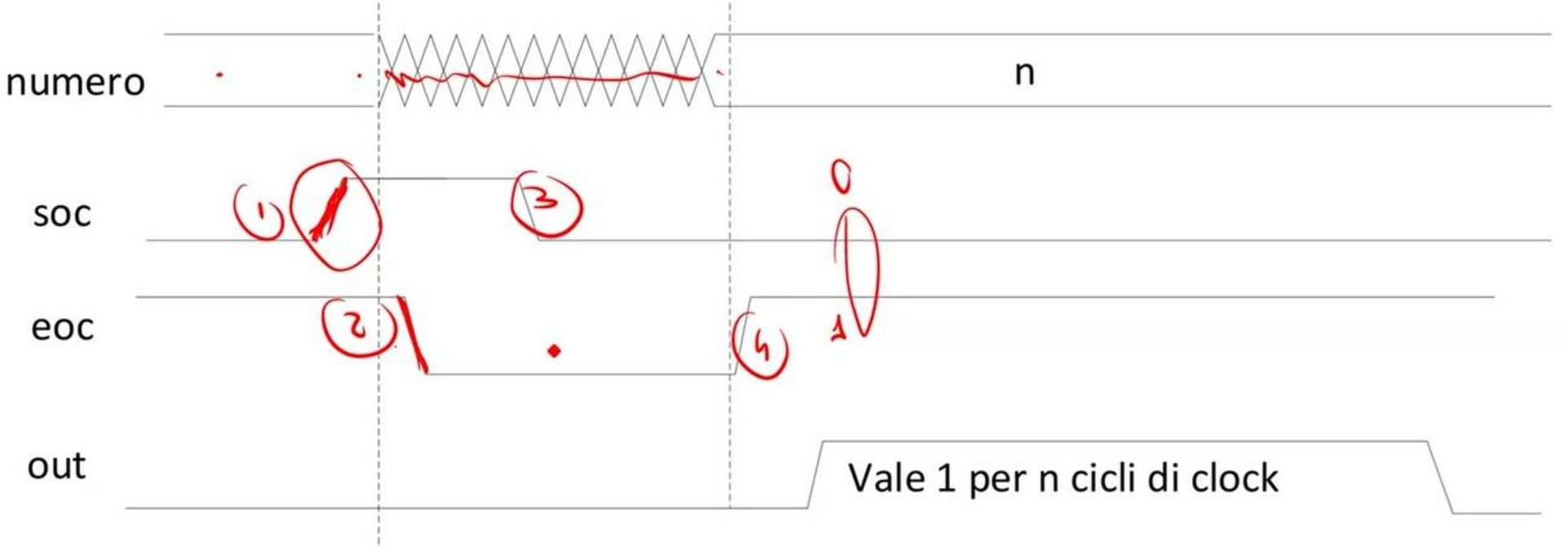
- Nell'esempio appena visto l'handshake parte dal **produttore**
- Esiste un altro handshake, in cui il **consumatore** fa la prima mossa
  - Richiede un dato al produttore mettendo ad uno **«start of computation», soc**
  - Attende che il produttore metta ad uno **«end of computation», eoc**
  - Gli intervalli di validità dei dati sono **diversi** dall'handshake /dav-rfd



Handshake soc-eoc (cont.)



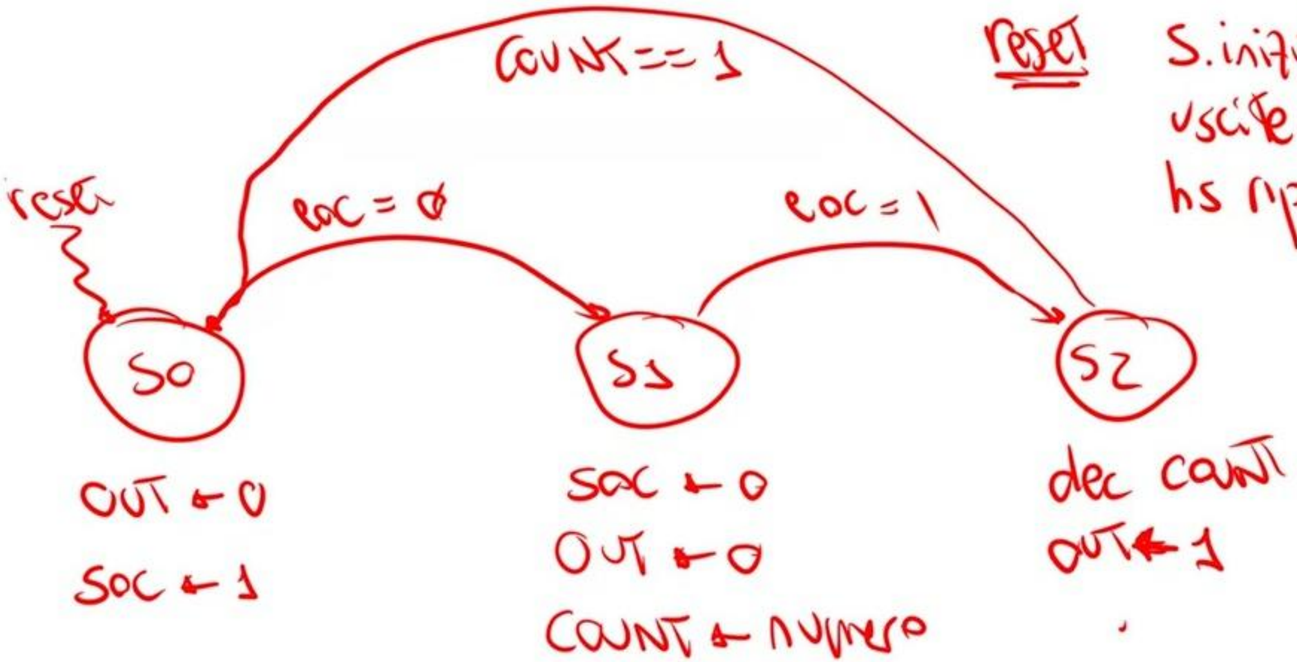
- La situazione di riposo è **soc=0, eoc=1, dato buono**
- Il dato è buono **quando eoc vale 1**
- Le transizioni di soc ed eoc sono comunque alternate

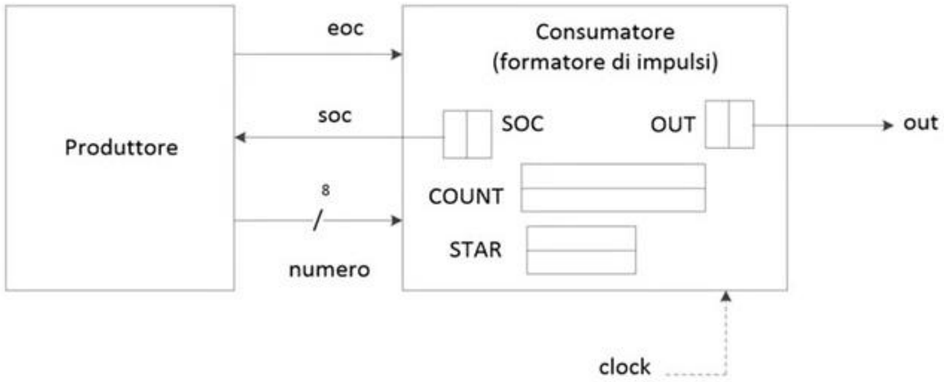


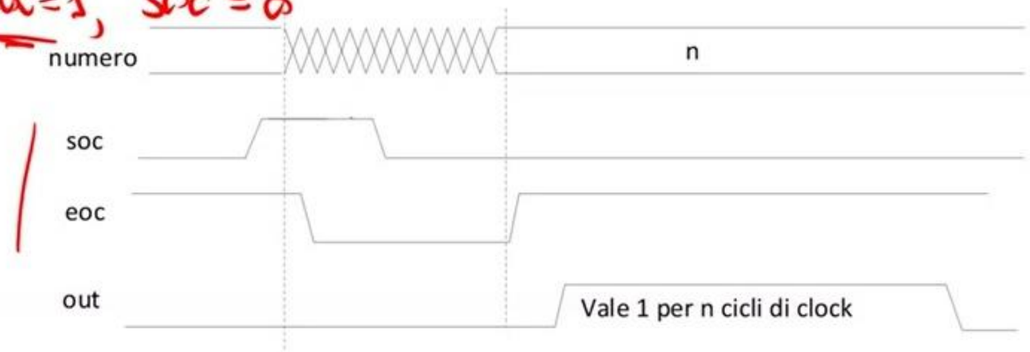
Esercizio (cont.)

STAR
soc, OUT 1 bit
COUNT 8 bit

reset S. iniziale S0
uscite OUT = ∅
hs riposo eoc = 1, soc = 0

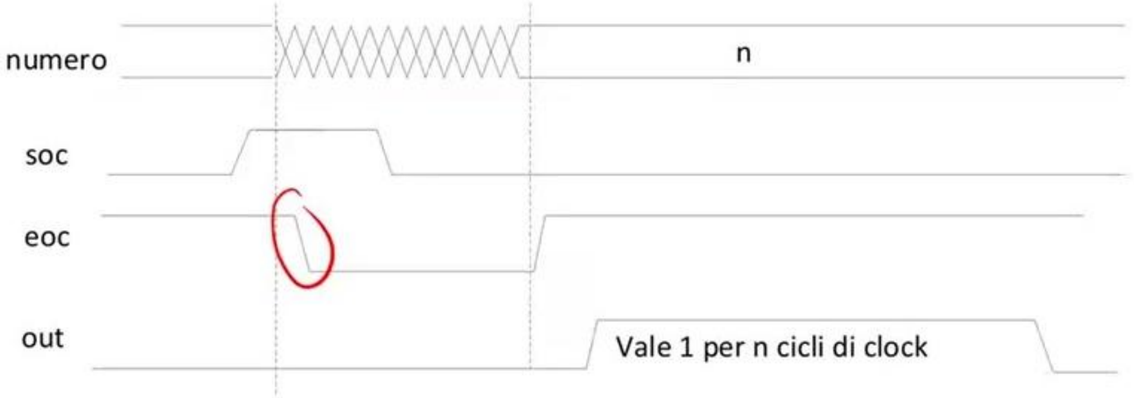






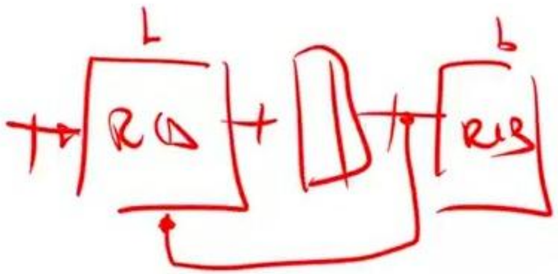
Esempio – handshake soc-eoc (cont.)

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

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



Sintesi di RSS complesse

RC

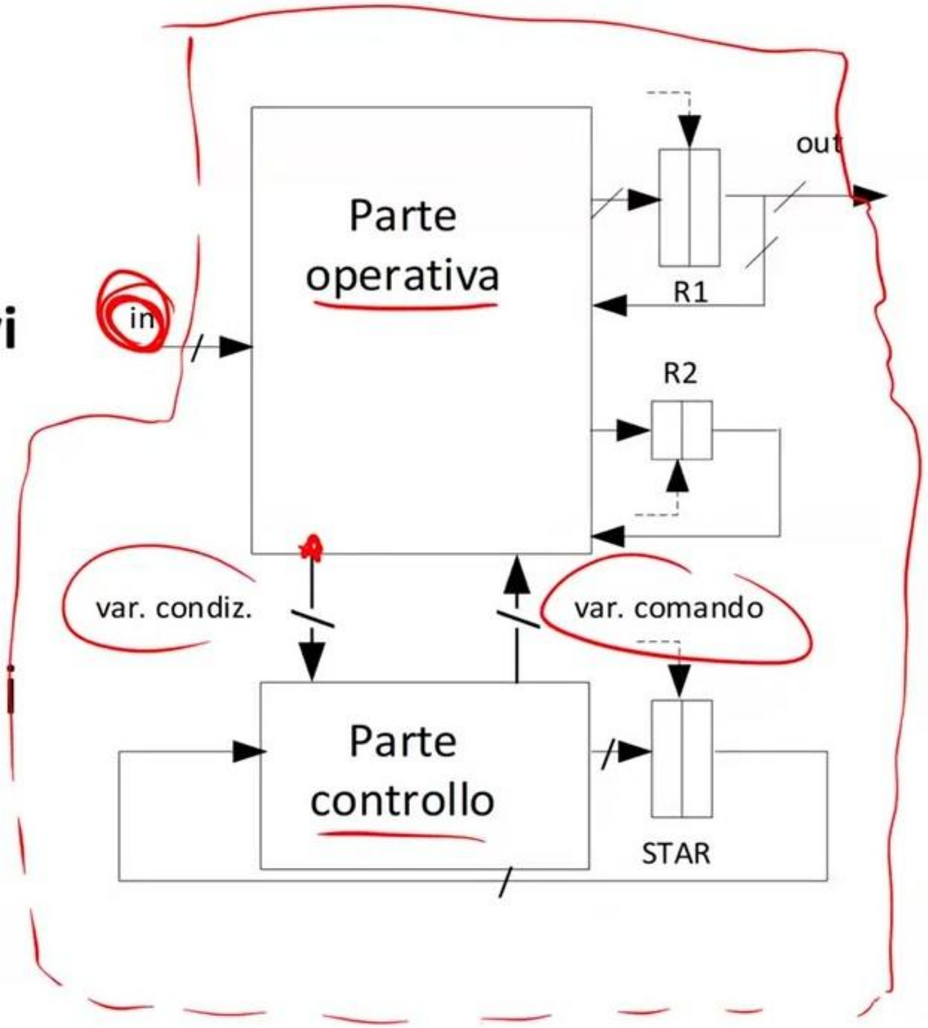


- Abbiamo visto esempi di **descrizione** scritta usando il **linguaggio di trasferimento tra registri del Verilog**.
- La descrizione può essere **simulata** per verificarne il comportamento tramite diagrammi di temporizzazione.
- Dobbiamo adesso preoccuparci di come si fa a **sintesi** una rete così descritta, in termini **circuiti elementari noti**
- Sintesi tramite **scomposizione in "parte operativa" e "parte controllo"**
  - procedimento **semiautomatico**
  - **non ha come scopo arrivare ad una sintesi ottima**

SCN SP

Scomposizione PO/PC

- **Dividere la RSS** in due sottoreti comunicanti:
  - Parte **operativa:** tutta la logica necessaria **all'interfacciamento con il mondo esterno** e alla produzione degli stati di ingresso per i **registri operativi**
  - Parte **controllo**: la logica necessaria all'aggiornamento dello **stato interno**
- Le due reti hanno lo **stesso clock**
- Comunicano tra loro tramite due gruppi di variabili
  - Var. di **comando**: PC -> PO
  - Var. di **condizionamento**: PO -> PC



Esempio: riconoscitore e contatore di seq. alternate

```
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
```

- Partiamo dalla **parte operativa**.
  1. Aggiornare lo stato dei registri **operativi** ad ogni clock
  2. Produrre le variabili di condizionamento

1. Guardare ai registri operativi come a registri multifunzionali

Esempio: riconoscitore e contatore di seq. alternate

```
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
```

- Per ogni registro **operativo** si isolano le **μ-operazioni diverse**
- COUNT

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

- Registro multifunzionale a **due vie**

Registri multifunzionali

- COUNT

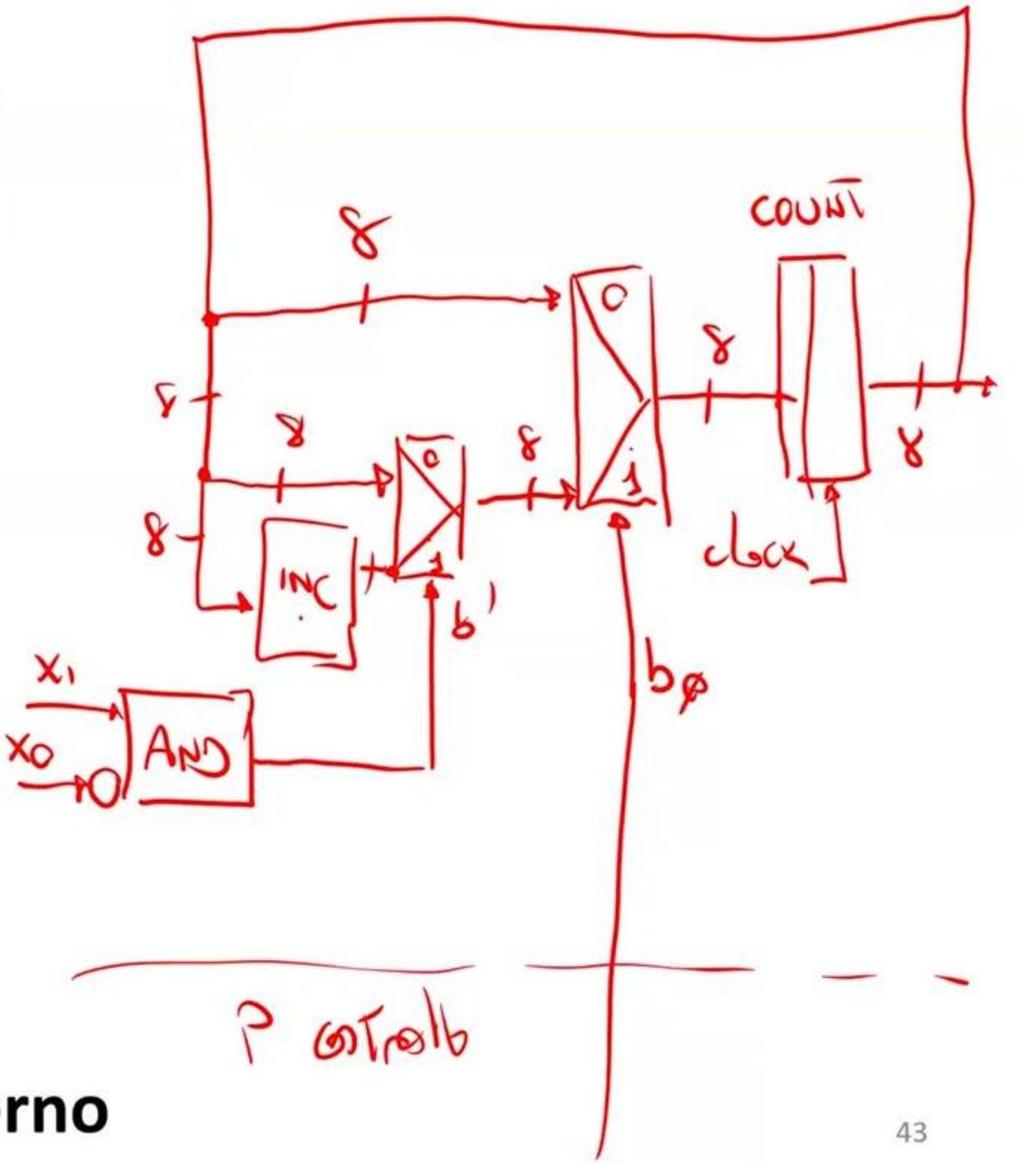
```
S0, S1 : COUNT<=COUNT;
S2     : COUNT<=(x1_x0=='B10)?COUNT+1:COUNT;
```

x_1 x_0	b'
1 0	1
others	0

	$b'=1$	$b'=0$	b_0
S0, S1			0
S2			1

- Registro multifunzionale a **due vie**

STATO	b_0
S_0	0
S_1	0
S_2	1



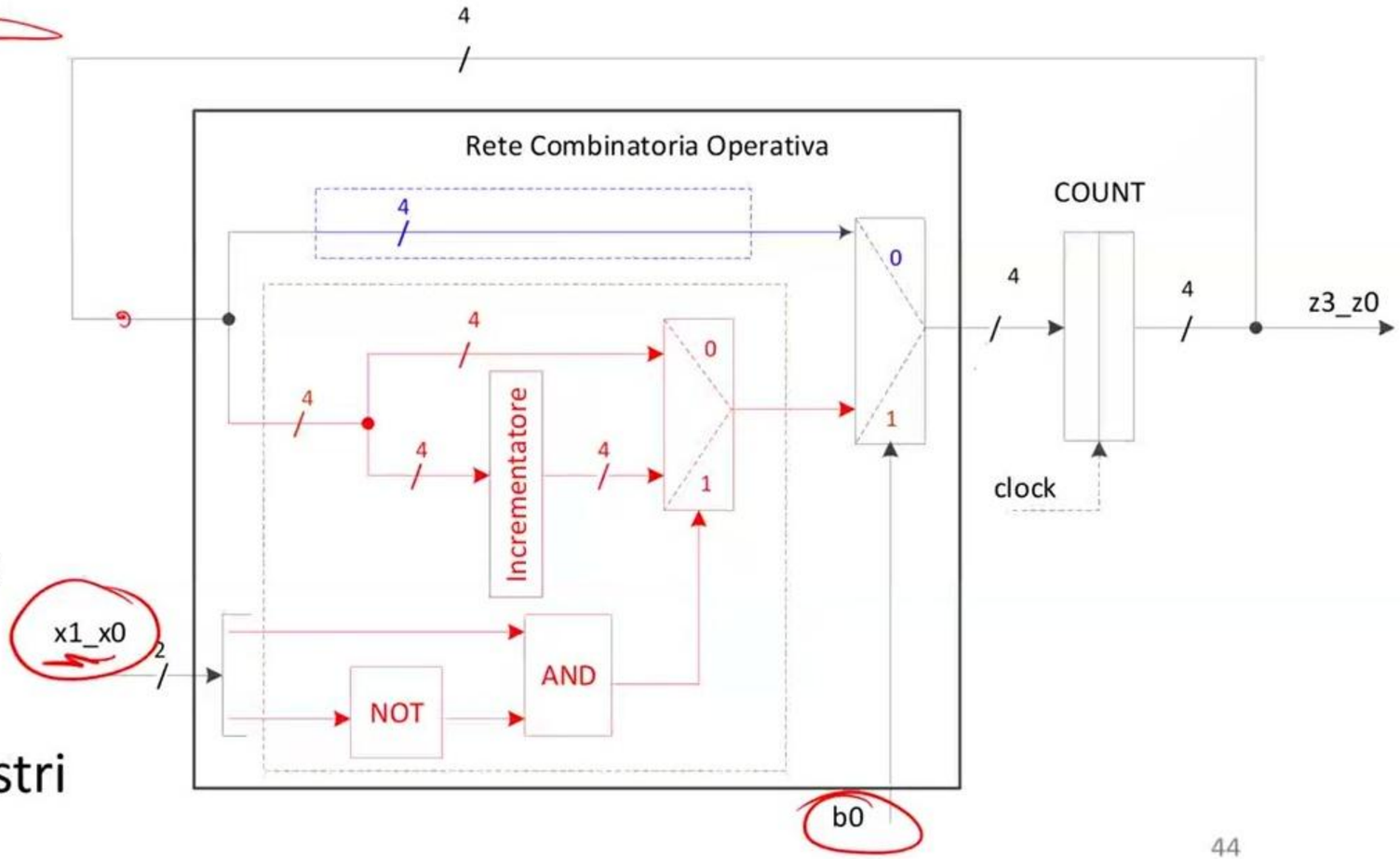
- La variabile b0 dipende **solo dallo stato interno**

Registri multifunzionali (cont.)

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

- Verrà prodotta dalla **parte controllo**
  - Variabili di comando

- Rete combinatoria operativa
  - Ingressi: ingressi della RSS, stato dei registri **operativi**, variabili di **comando**
  - Uscite: stato di ingresso dei registri operativi



Esempio: riconoscitore e contatore di seq. alternate

```
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
```

- Partiamo dalla **parte operativa**.
  1. Aggiornare lo stato dei registri operativi ad ogni clock
  2. Produrre le variabili di condizionamento

2. Cercare le **condizioni indipendenti** su **STAR**

Esempio: riconoscitore e contatore di seq. alternate

```
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
```

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

- Assegno a ciascuna di queste una **variabile di condizionamento**,
  - vale **uno se la condizione è vera, e zero se la condizione è falsa.**

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

Esempio: riconoscitore e contatore di seq. alternate

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

descrittivo

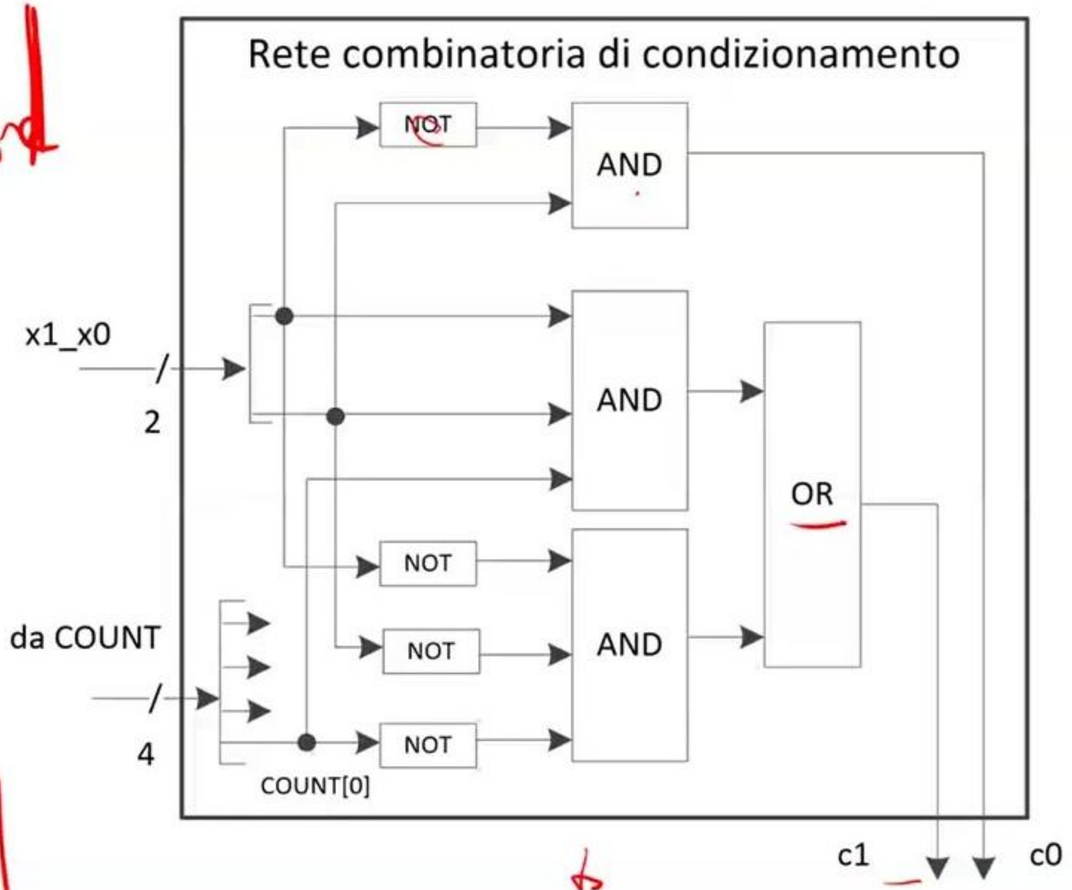
- **Rete combinatoria di condizionamento**
- Ingresso
  - le variabili di ingresso della RSS,
  - lo stato dei registri operativi.
- Uscita
  - le variabili di condizionamento

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

$$c_0 = \overline{x_1} \cdot x_0$$

```
assign c1=(COUNT[0] & x1 & x0)|
          (~COUNT[0] & ~x1 & ~x0);
assign c0= ~x1 & x0
```

sintesi

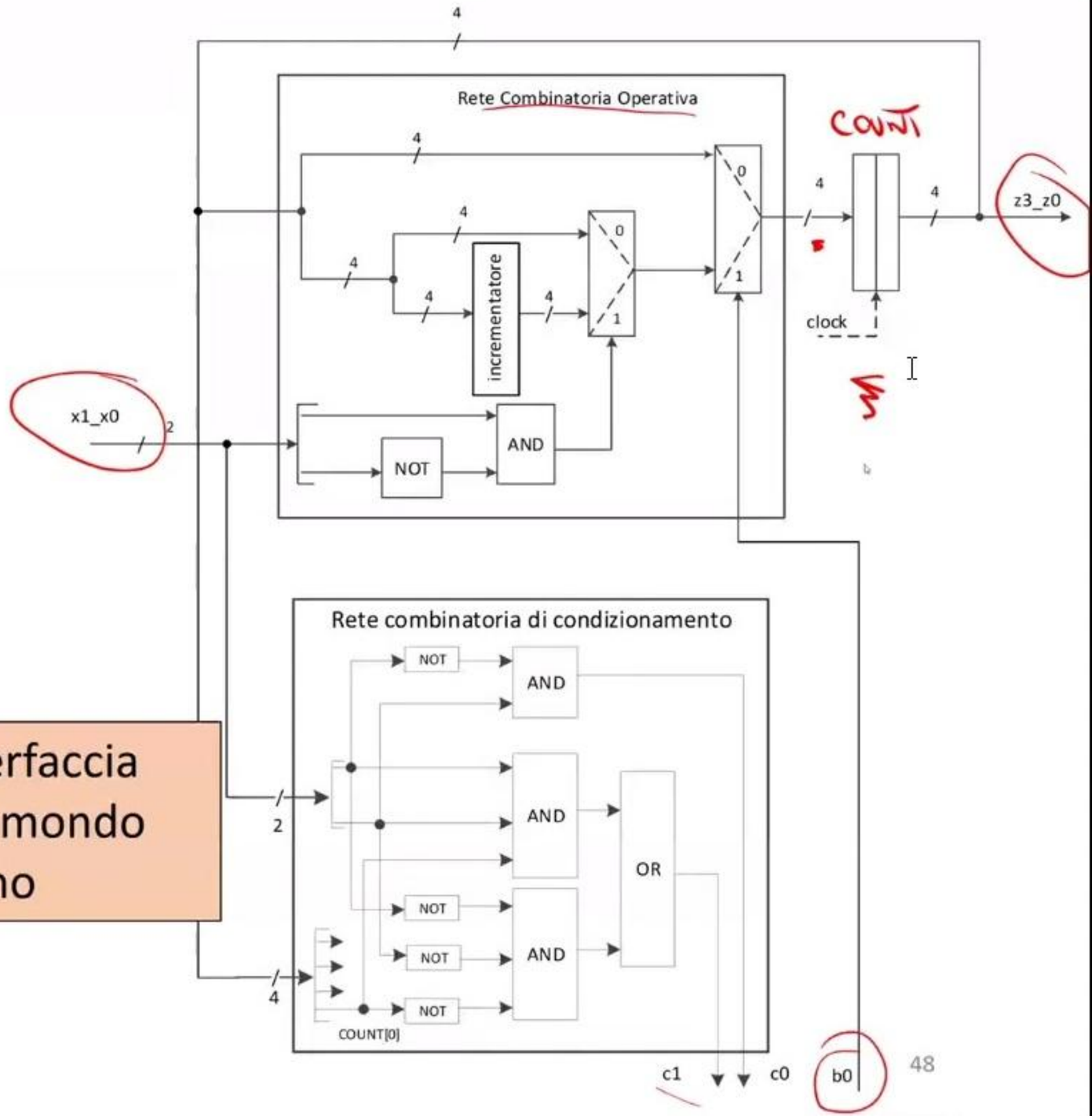


COUNT[0], x_1, x_0	match
000	1
111	1
Others	0

Parte operativa

- Contiene
  - i **registri operativi** e le relative **reti combinatorie operative**
  - La rete combinatoria di condizionamento
- Ha in ingresso
  - Gli ingressi della RSS
  - Le **variabili di comando**
- Ha in uscita
  - Le uscite della RSS
  - Le **variabili di condizionamento**

Si interfaccia con il mondo esterno



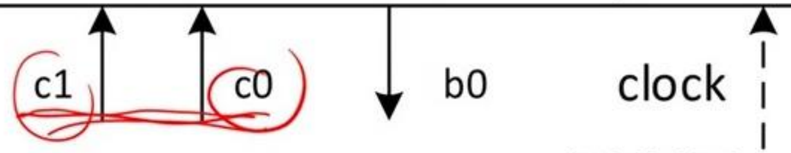
Parte controllo

- Contiene
  - il **registro STAR** ed alcune **reti combinatorie**
- Ha in ingresso
  - Le **variabili di condizionamento**
- Ha in uscita
  - Le **variabili di comando**

- Non si interfaccia con l'esterno
  - Normalmente piuttosto semplice

	c_1c_0 00	01	11	10	b0
S_0	S0	S0	S1	S1	0
S_1	S0	S2	S2	S1	0
S_2	S0	S0	S1	S1	1

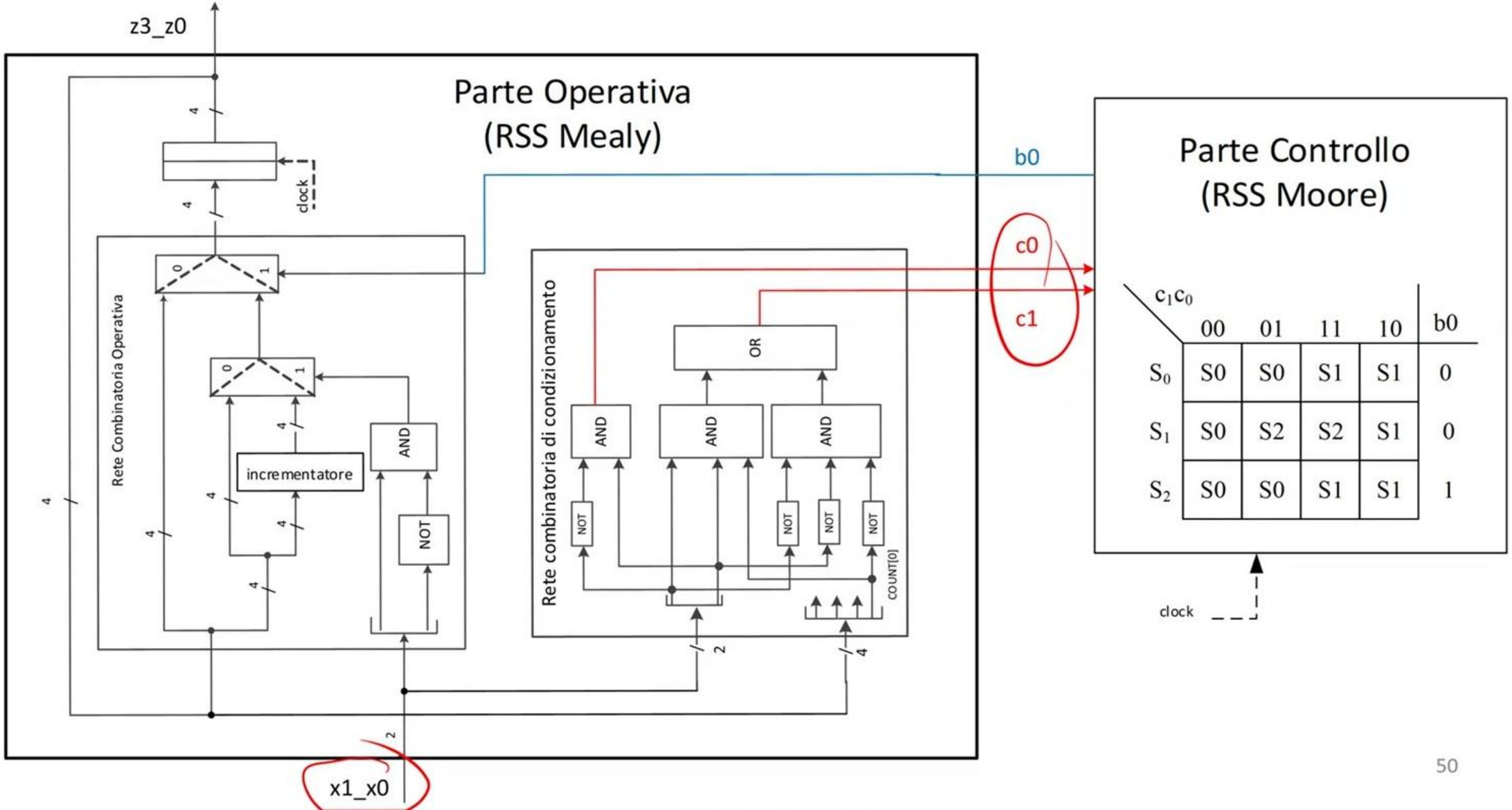
Parte Controllo (RSS Moore)




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

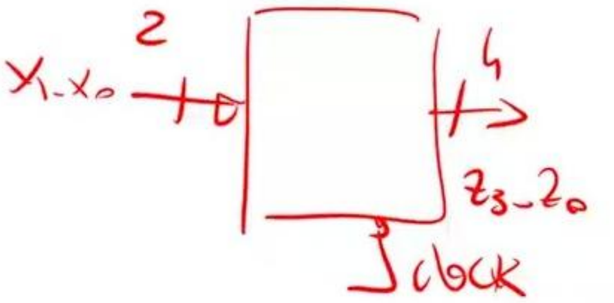
STAR: si evolve a seconda delle var. di condizionamento

Scomposizione PO/PC



	c_1c_0 = 00	01	11	10	b0
S_0	S0	S0	S1	S1	0
S_1	S0	S2	S2	S1	0
S_2	S0	S0	S1	S1	1

In Verilog




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

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

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

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

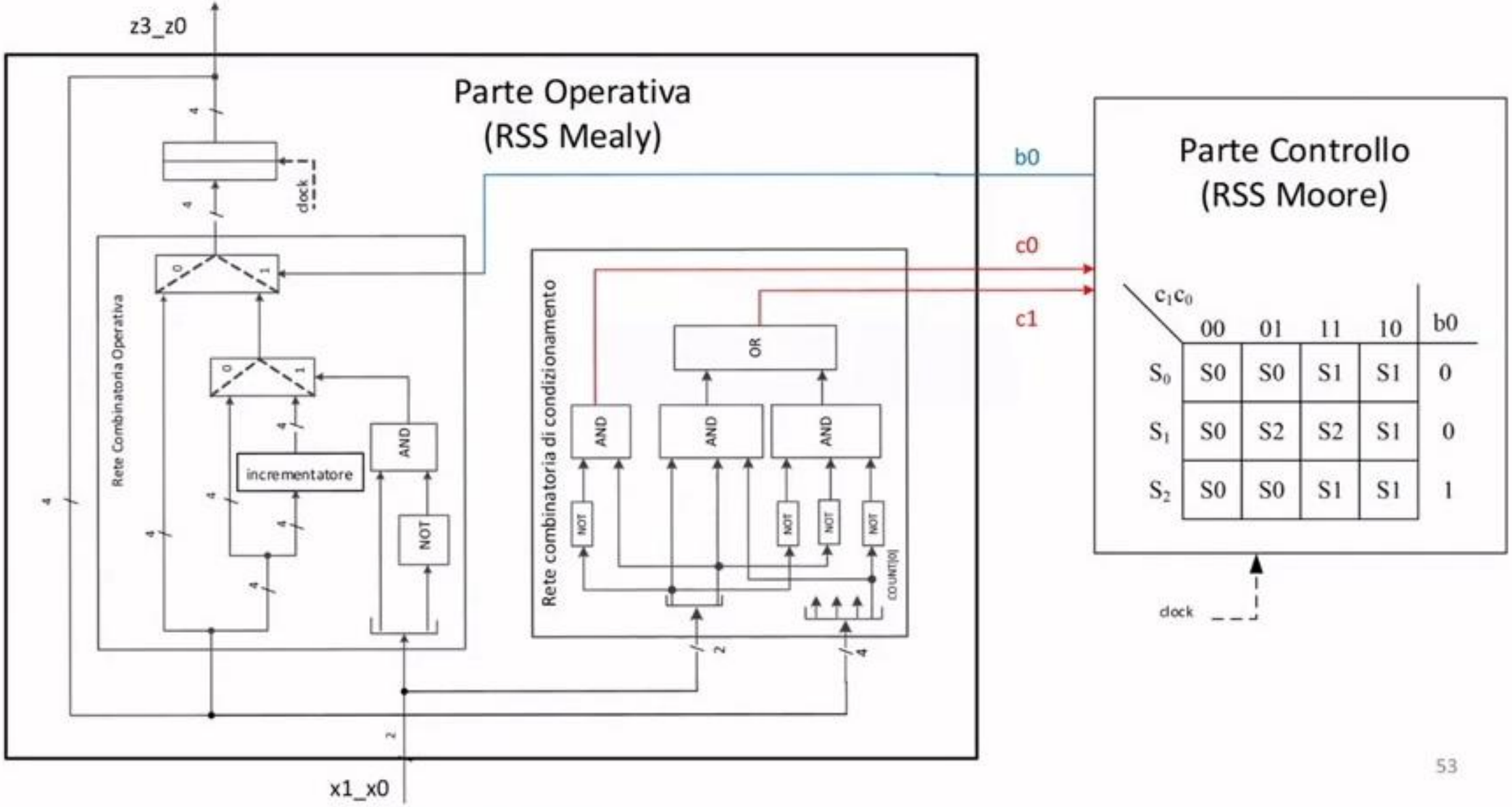
In Verilog (cont.)

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

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

Il blocco segnato in verde è **la parte controllo**, ed è **una RSS di Moore.** Uscite (var di comando) dipendono **solo dallo stato interno (STAR)**

Tutto il resto è la **parte operativa**, che è una **RSS di Mealy**

Scomposizione PO/PC



c_1c_0	00	01	11	10	b0
S_0	S0	S0	S1	S1	0
S_1	S0	S2	S2	S1	0
S_2	S0	S0	S1	S1	1

In Verilog (cont.)

out in

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

B: X×S → Z

RC condizionato

Moly

1:1

In Verilog (cont.)

```
module Parte_Controllo(b0,c1,c0,clock,reset_);
  input clock,reset_;
  input c1,c0;
  output b0;
  reg [1:0] STAR; parameter S0='B00,S1='B01,S2='B10;
  assign b0=(STAR==S2)?1:0;
  always @(reset_==0) #1 STAR<=S0;
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: STAR<=(c1==1)?S1:S0;
      S1: STAR<=(c0==1)?S2:(c1==1)?S1:S0;
      S2: STAR<=(c1==1)?S1:S0;
    endcase
endmodule
```

In Verilog (cont.)

- Modulo «top-level»

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

La scomposizione PO/PC è (quasi) una sintesi

- La scomposizione PO/PC di una rete descritta, con:
  - specifica (al livello di **descrizione di ciascun registro multifunzionale**) di ciascuna rete combinatoria operativa. Qui basta fermarsi quando si vedono **reti standard**.
  - specifica (al livello di **espressioni di algebra di Boole**) della rete combinatoria di condizionamento. Qui è necessario essere **precisi fino al bit**.
  - sintesi **della parte controllo** (scegliendo uno tra i possibili formalismi che conoscete o che vedrete più avanti)
- costituisce il **punto di arrivo** della sintesi richiesta

ni Stea

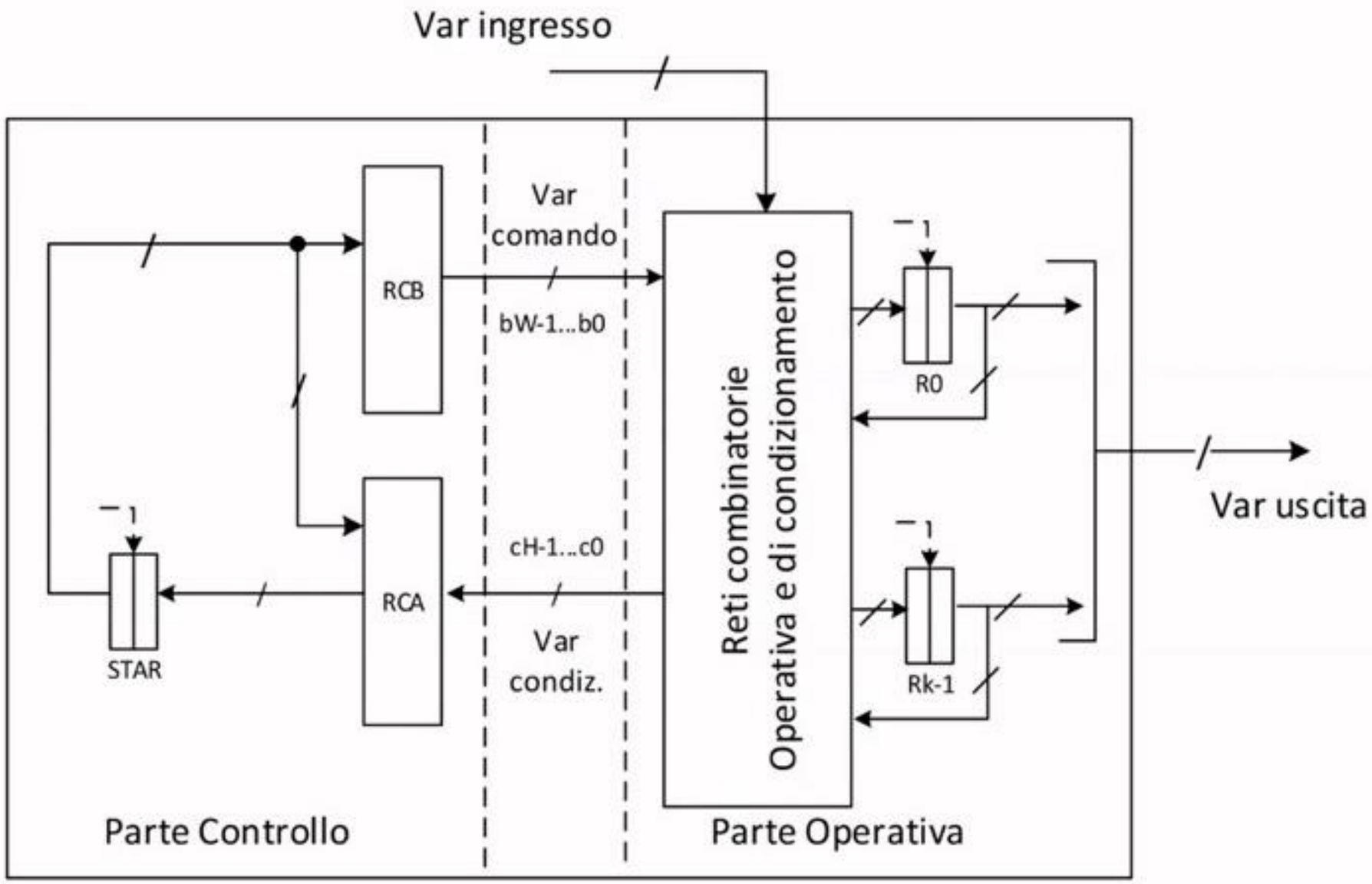
La scomposizione PO/PC è (quasi) una sintesi

- Si può fare **in modo automatico** a partire dalla descrizione in Verilog
- la sintesi della PO è euristica, ma **semplice**, in quanto consta di
  - registri multifunzionali
  - rete combinatoria di condizionamento, normalmente semplice
- la sintesi della PC è affrontabile tramite tecniche **standard**, e risulta sempre piuttosto semplice
  - Nell'esempio considerato (rete di Moore a 3 stati) si può fare sintetizzando le reti RCA ed RCB, come visto a suo tempo
  - In casi più complessi esiste comunque un metodo **euristico** per la sintesi della PC

disegnate

ni Stea

Scomposizione PO/PC di una rete generica



Esercizio: formatore di impulsi h.s. /dav-rfd

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

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

d. s.

tre registri operativi: RFD, COUNT, OUT.

Esercizio: formatore di impulsi h.s. /dav-rfd

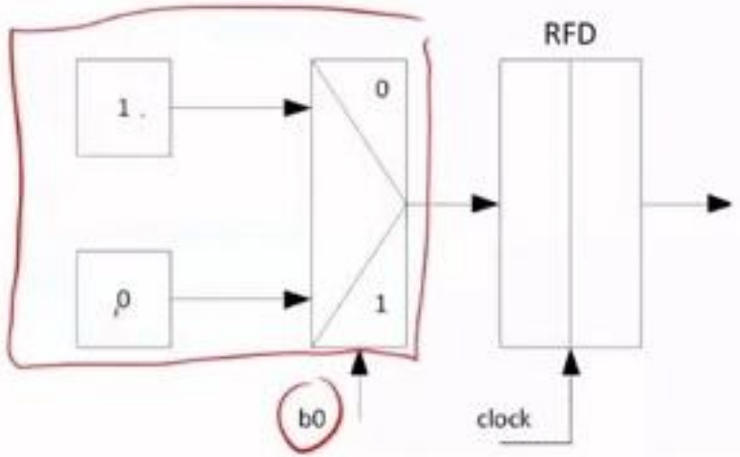
```
S0: begin RFD<=1; OUT<=0; COUNT<=numero; STAR<=(dav_==1)?S0:S1; end
S1: begin RFD<=0; OUT<=1; COUNT<=COUNT-1; STAR<=(COUNT==1)?S2:S1; end
S2: begin RFD<=0; OUT<=0; STAR<=(dav_==1)?S0:S2; end
```

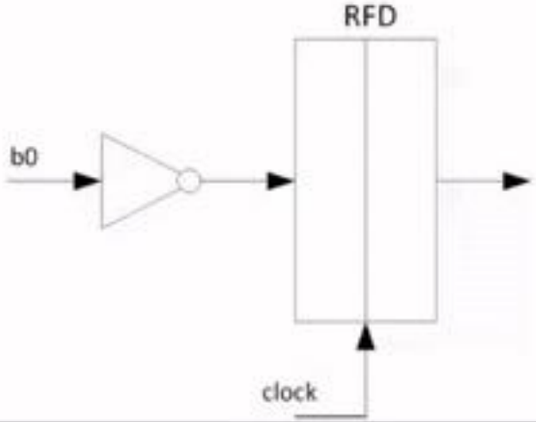
- **RFD:** be

S0: RFD<=1 1

S1,S2: RFD<=0 0

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





Esercizio: formatore di impulsi h.s. /dav-rfd

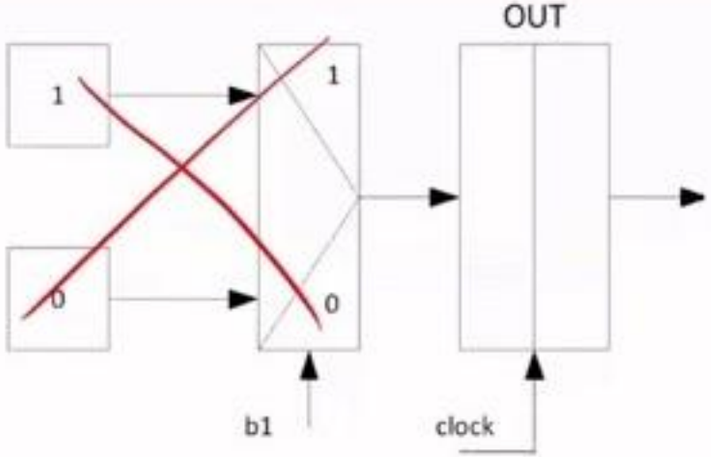
```
S0: begin RFD<=1; OUT<=0; COUNT<=numero; STAR<=(dav_==1)?S0:S1; end
S1: begin RFD<=0; OUT<=1; COUNT<=COUNT-1; STAR<=(COUNT==1)?S2:S1; end
S2: begin RFD<=0; OUT<=0; STAR<=(dav_==1)?S0:S2; end
```

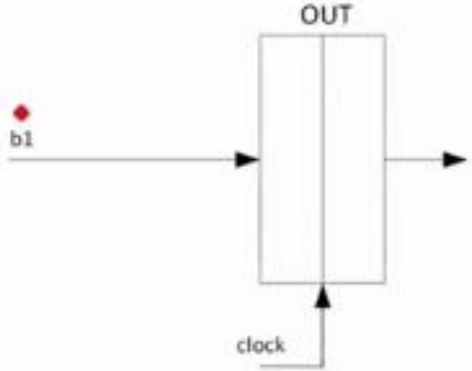
- **OUT:** b_1

S0, S2: OUT<=0 0

S1: OUT<=1 1

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





Esercizio: formatore di impulsi h.s. /dav-rfd

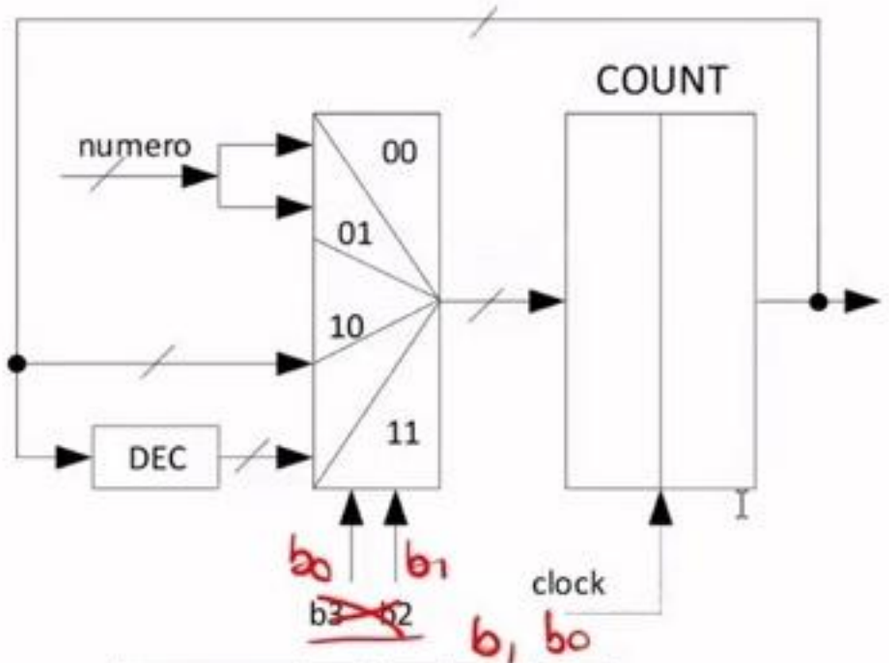
```
S0: begin RFD<=1; OUT<=0; COUNT<=numero; STAR<=(dav_==1)?S0:S1; end
S1: begin RFD<=0; OUT<=1; COUNT<=COUNT-1; STAR<=(COUNT==1)?S2:S1; end
S2: begin RFD<=0; OUT<=0; STAR<=(dav_==1)?S0:S2; end
```

- **COUNT:**

b_3 b_2

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

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



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

Ottimizzazioni della sintesi

- Far coincidere variabili di comando
  - eventualmente sfruttando non specificati
- **E basta.**

- Cose da **non** fare
  - Fare la sintesi diversa dalla descrizione (grave)
  - Fare la sintesi, ottimizzarla rendendola diversa dalla descrizione, **e poi tornare a modificare la descrizione** per renderla coerente alla sintesi ottimizzata

ni Stea

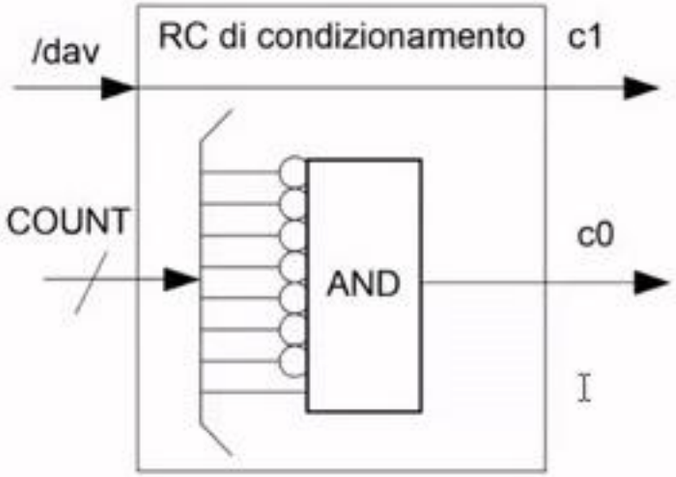
Esercizio: formatore di impulsi h.s. /dav-rfd

```
S0: begin RFD<=1; OUT<=0; COUNT<=numero; STAR<=(dav_==1)?S0:S1; end
S1: begin RFD<=0; OUT<=1; COUNT<=COUNT-1; STAR<=(COUNT==1)?S2:S1; end
S2: begin RFD<=0; OUT<=0; STAR<=(dav_==1)?S0:S2; end
```

- **STAR:**

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

0000 0001



- **Due** variabili di condizionamento
  - c_1=1 quando */dav* vale 1, e 0 altrimenti
  - c_0=1 quando COUNT vale 1, e 0 altrimenti

In Verilog

- Partiamo dal modulo top-level

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

In Verilog (cont.)

assign c1 = dav_;
assign c0 = (~COUNT[7] & ~COUNT[6] ... & COUNT[0]),

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

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

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

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

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

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

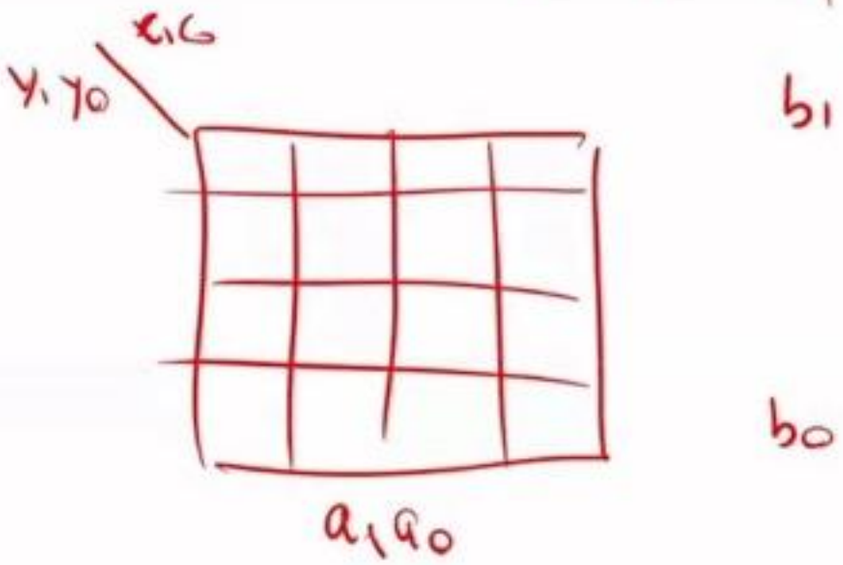
In Verilog (cont.)

2 in
3 S.i.

out in

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

S_{int} \ c_1c_0	00	01	11	10	b_1b_0
00	S_0	S_1	S_1	S_0	S_0	00
01	S_1	S_1	S_2	S_2	S_1	11
10	S_2	S_2	S_2	S_0	S_0	01



Sintesi della parte controllo

- Se la rete è complessa
  - molti stati interni
  - molte variabili di condizionamento
- Non è agevole usare la tecnica di sintesi nota usata per le RSS di Moore
  - Scelta modello + sintesi RCA, RCB

- È necessaria una tecnica **più generale**
  - **Euristica**
  - **Non ottimizzata**
  - Comporta dei **vincoli sul modo in cui si scrive la descrizione**

Sintesi della parte controllo (cont.)

- Vincolo: i μ-salti **devono essere**
  - **Incondizionati:** `STAR<=S2;`
  - Oppure **a due vie:** `STAR<=(c1==1)?S0:S1;`

- Un μ-salto incondizionato è un caso limite di μ-salto a due vie
  - `STAR<=S2` ⇔ `STAR<=(c1==1)?S2:S2;`

- Con questa tecnica **non sono sintetizzabili** μ-salti a più di due vie
  - `STAR<=(x1_x0=='B01)?S2:(x1_x0=='B11)?S1:S0;`

- Vedremo in seguito un modo per recuperare **alcuni tipi di** μ-salti a molte vie
  - Quelli con *n* vie (*n* molto grande)
  - Quindi, in ogni caso: **una via, due vie, molte vie** (e non: tre vie, come quello sopra)

Sintesi della parte controllo (cont.)

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

- **μ-indirizzo:** codifica degli stati interni
  - Quella riportata nel **parameter**
- ∀ μ-indirizzo, individuo lo stato delle **var. di comando (μ-codice)**

μ-addr	$b_1 b_0$
00	00
01	11
10	01
11	--

Sintesi della parte controllo (cont.)

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

- Associo una **codifica in base 2** alle var. di condizionamento.
  - Due var.: -> **un bit**
- Aggiungo la var. di condizionamento **efficace**
  - Quella che **guida il μ-salto**

	μ-addr	$b_1 b_0$	c_{eff}
S0	00	00	1
S1	01	11	0
S2	10	01	1
—	11	--	-

Reintrodurre i μ-salti a più vie (cont.)

- Ammesso che l'ingresso resti fermo per due clock
  - Perdo un clock per ogni via dopo la seconda
  - Molto inefficiente se il numero di vie è molto elevato

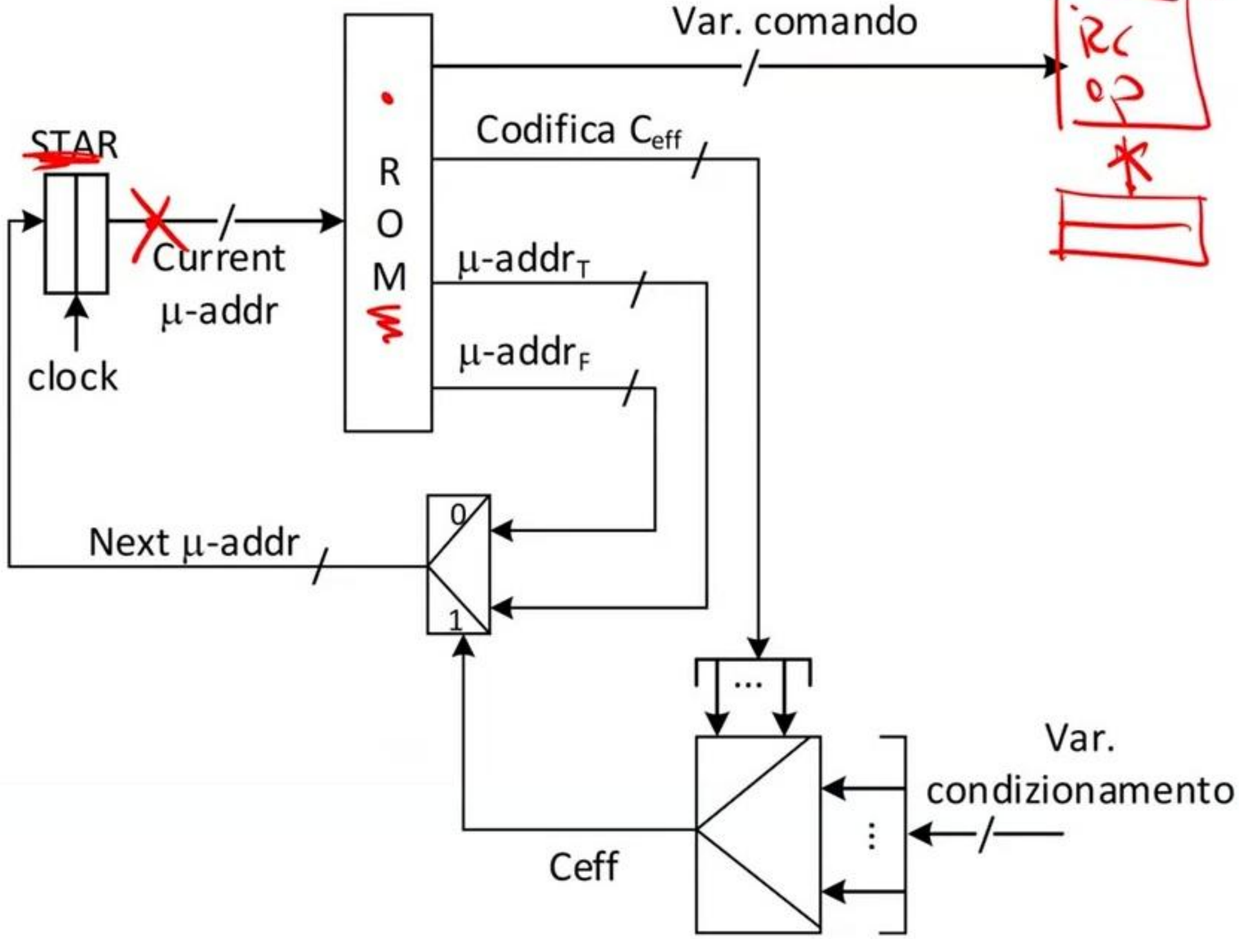
```
S0: begin /*elaborazioni;*/
      STAR<=(condizione_1)?S1:
            (condizione_2)?S2:
            […]
            (condizione_k-1)?Sk-1:Sk;
    end
```

```
S0:      begin /*elaborazioni;*/
                STAR<=(condizione_1)?S1:S0_1; end
S0_1:    begin STAR<=(condizione_2)?S2:S0_2; end
S0_2:    begin STAR<=(condizione_3)?S3:S0_3; end
[…]
S0_k-1: begin STAR<=(condizione_2)?Sk-1:Sk; end
```

Sintesi della PC - modello μ-address based

- Modello di sintesi **μ-address based**
  - STAR contiene **μ-indirizzi**
  - ROM in cascata a RC operative
  - Probabile rallentamento del clock

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

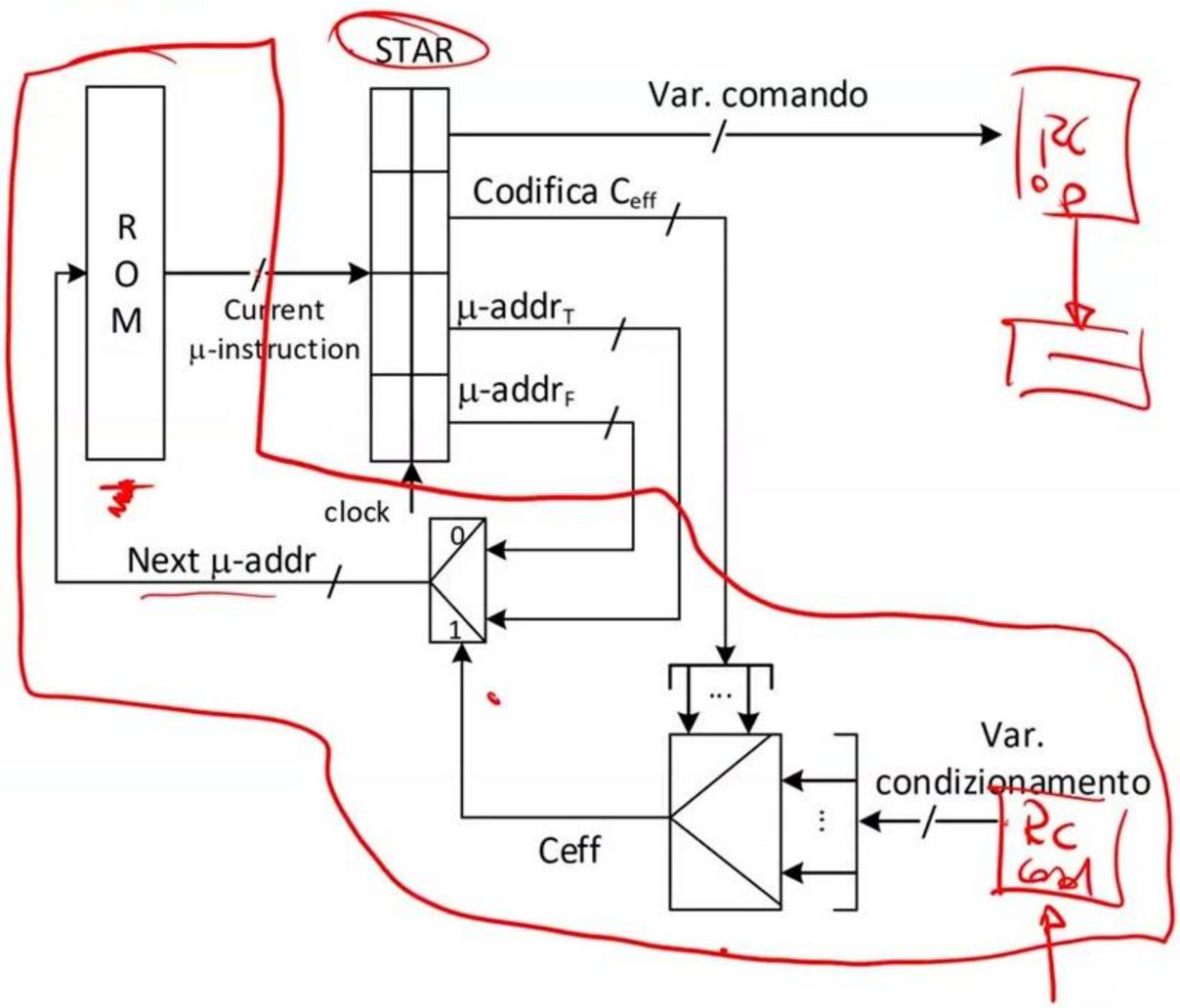


Sintesi della PC - modello μ-instruction based

T ≥ . . . . . 7 bit

- Modello di sintesi **μ-instruction based**
  - STAR contiene **μ-istruzioni**
  - ROM in cascata a RC di condizionamento
  - Minori problemi sul clock
  - Ma STAR più grande (spesso di parecchio)

	100			
	μ-instruction			
μ-addr	$b_1 b_0$	c_{eff}	**μ-addr$_T$**	**μ-addr$_F$**
00	00	1	00	01
01	11	0	10	01
10	01	1	00	10
11	--	-	--	--



Reintrodurre i μ-salti a più vie

- Se in una descrizione ho salti a 3 vie, devo **considerare le condizioni** in clock differenti

  - `S0: STAR<=(x1_x0=='B01)?S2:(x1_x0=='B11)?S1:S0;`

  - `S0:  STAR<=(x1_x0=='B01)?S2:S0b:`
  - `S0b: STAR<=(x1_x0=='B11)?S1:S0;`

- Ammesso che l'ingresso resti fermo per due clock
  - Perdo un clock per ogni via dopo la seconda
  - Molto inefficiente se il numero di vie è molto elevato

MOV %AX,%BX MOV $0x1001,%AL

Reintrodurre i μ-salti a più vie (cont.)

fetch → exe

alcune cert.

- In pratica, i salti sono spesso a due vie.
- In un processore è così, **tranne che in due casi**:

- All'**inizio** della fase di fecth
  - Leggo il **formato** dell'istruzione (che mi dice dove sono **gli operandi**), e dovrò saltare a un blocco di statement diverso a seconda di dove si trovano

- Alla **fine** della fase di fecth
  - Dopo che ho letto l'istruzione, devo saltare al blocco di statement che gestisce la **fase di esecuzione** di quell'istruzione.

- In entrambi i casi, servono salti a **molte vie** (# formati, # istruzioni)
  - Impensabile perdere clock ogni volta

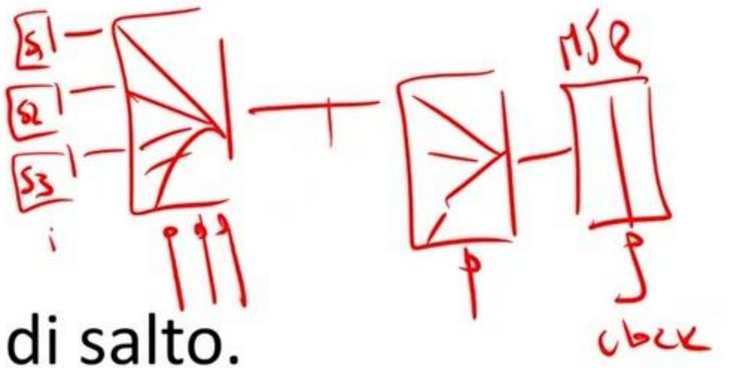
Reintrodurre i μ-salti a più vie (cont.)

- Ammesso che l'ingresso resti fermo per due clock
  - Perdo un clock per ogni via dopo la seconda
  - Molto inefficiente se il numero di vie è molto elevato

```
S0: begin /*elaborazioni;*/
      STAR<=(condizione_1)?S1:
            (condizione_2)?S2:
            […]
            (condizione_k-1)?Sk-1:Sk;
    end
```

```
S0:      begin /*elaborazioni;*/
               STAR<=(condizione_1)?S1:S0_1; end
S0_1:    begin STAR<=(condizione_2)?S2:S0_2; end
S0_2:    begin STAR<=(condizione_3)?S3:S0_3; end
[…]
S0_k-1:  begin STAR<=(condizione_2)?Sk-1:Sk; end
```

Multiway Jump Register (MJR)



- Registro **operativo** destinato a contenere indirizzi di salto.
  - Lo sintetizzo nella parte operativa
  - Aggiusterò la parte controllo per consentire di usare salti guidati da MJR

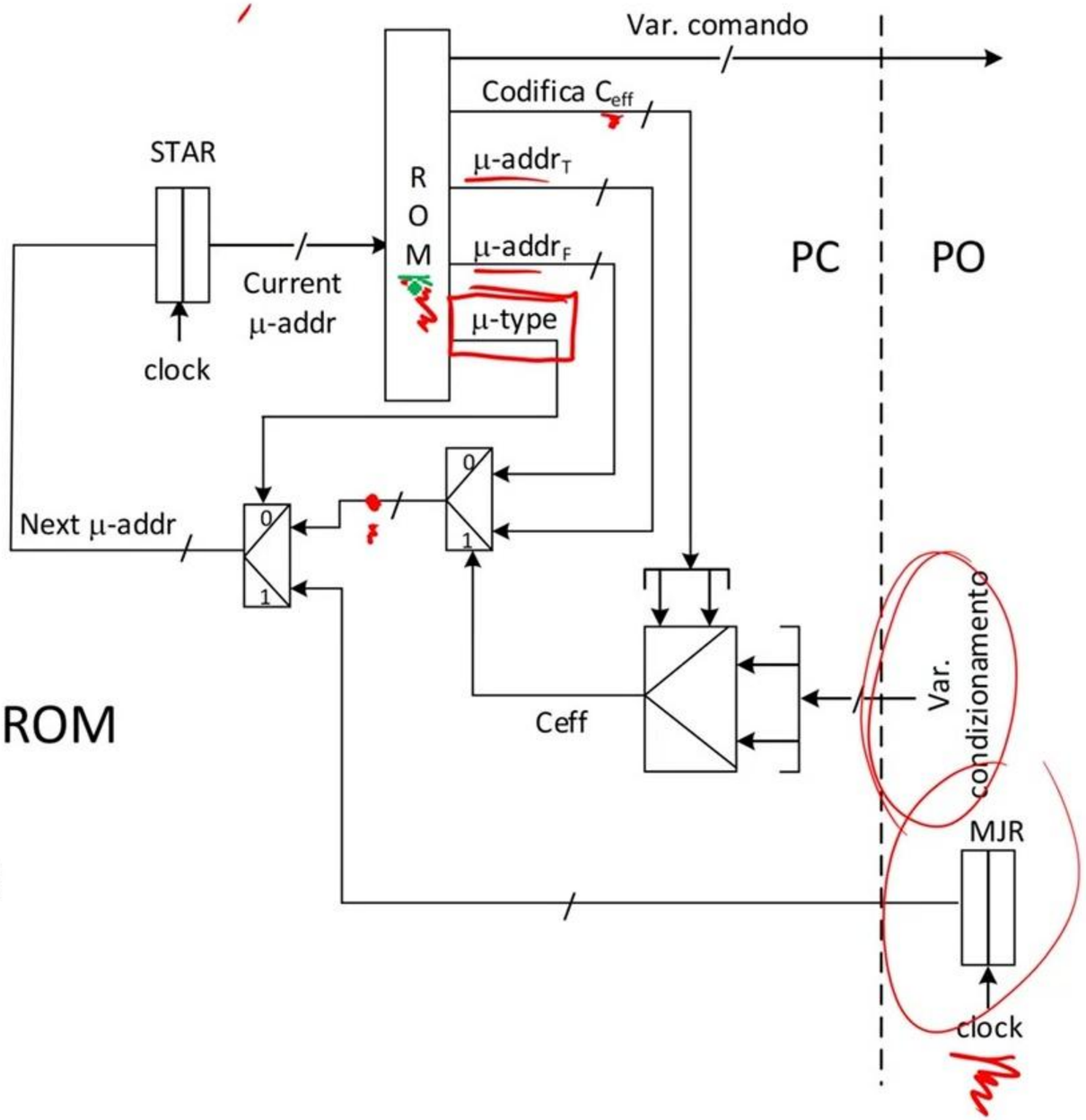
```
S0: begin /*elaborazioni;*/
      STAR<=(condizione1)?S1:
            (condizione2)?S2:
            […]
            (condizionek-1)?Sk-1:Sk;
    end
```

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

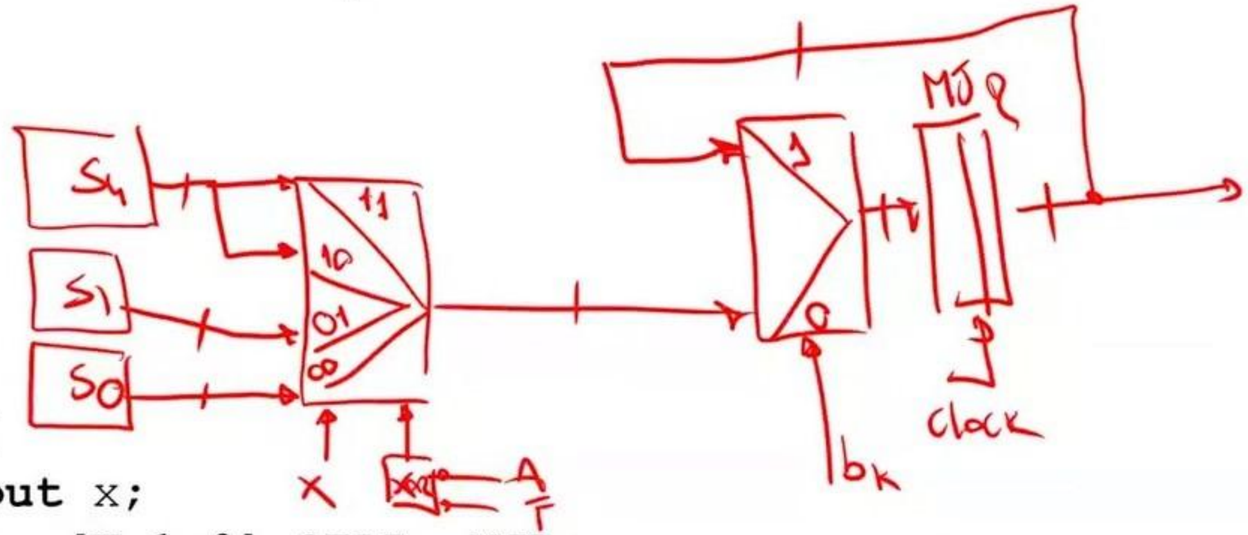
Sintesi con registro MJR

- Nella PO
  - MJR è un registro operativo
  - come qualunque altro

- Nella PC: due **tipi di μ-salto**
  - «standard», a 2 vie – uso c_{eff} presa dalla ROM
  - MJR-based: uso MJR
  - Serve un bit in più per codificare il **μ-tipo**:
    - 0: μ-salto **«standard»**
    - 1: μ-salto **basato su MJR**
  - Va aggiunto alla ROM anche il **μ-tipo**
  - Serve un altro MUX a 2 vie per distinguere il tipo di salto



Esempio di sintesi con MJR




```
[…]
input x;
reg  [K-1:0] STAR, MJR;
reg  A,B,F;
[…]
S0:  begin […] STAR<=(x==0)?S0:S1; end
S1:  begin […] STAR<=S2; end
S2:  begin […] STAR<=(A!=B)?S5:S3; end
S3:  begin […] STAR<=(gamma(A)==0)?S1:S4; end
S4:  begin […] STAR<=S5; end
S5:  begin […] MJR<=(x==1)?S4:(A!=F)?S1:S0; STAR<=S6; end
S6:  begin […] STAR<=MJR; end
```

c_0 0

0

c_1 0

c_2 0

0

0

bk MJR

0 S5 : (x==1)? S4 : ---

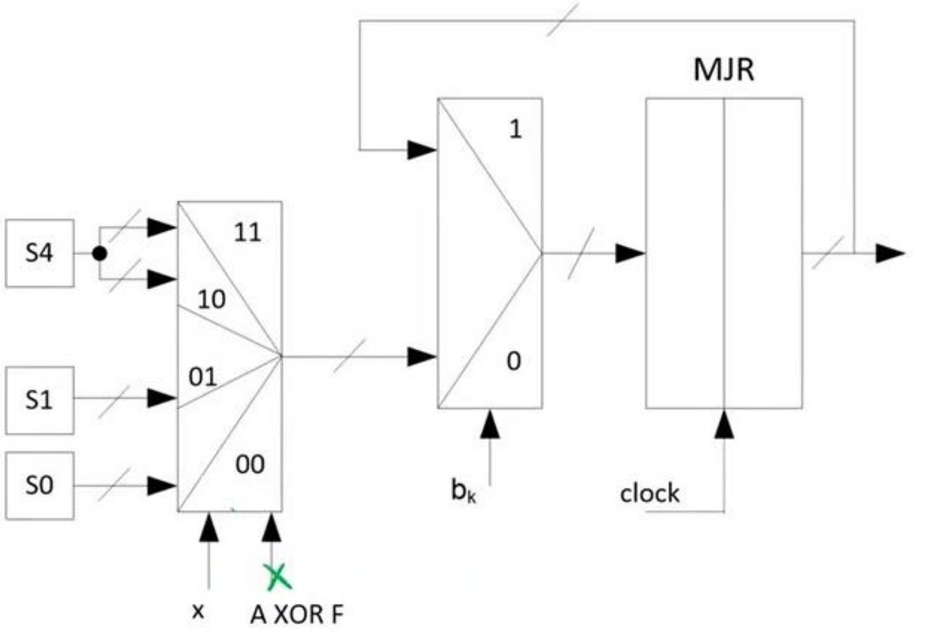
1 others MJR

1 μ-type

Esempio di sintesi con MJR

assign
$c_0 = (x==0)?1:0;$
$c_1 = (A!=B)?1:0;$
$c_2 = (gamma(A)==0)?1:0;$

- MJR: registro multifunzionale a **due vie**




```
[…]
input x;
reg  [K-1:0] STAR, MJR;
reg  A,B,F;
[…]
S0:  begin […] STAR<=(x==0)?S0:S1; end
S1:  begin […] STAR<=S2; end
S2:  begin […] STAR<=(A!=B)?S5:S3; end
S3:  begin […] STAR<=(gamma(A)==0)?S1:S4; end
S4:  begin […] STAR<=S5; end
S5:  begin […] MJR<=(x==1)?S4:(A!=F)?S1:S0; STAR<=S6; end
S6:  begin […] STAR<=MJR; end
```

c_0 0
0
c_1 0
c_2 0
0
0
1

M-type

Esempio di sintesi con MJR

- Struttura calcolatore
- Mer 1.15 12

10 GS — 12

μ-addr	μ-code			c_{eff}	μ-addr T	μ-addr F	μ-type
	…	b_k	…				
S0	…	1	…	00	S_0	S_1	0
S1	…	1	…	--	S_2	S_2	0
S2	…	1	…	01	S_5	S_3	0
S3	…	1	…	10	S_1	S_4	0
S4	…	1	…	--	S_5	S_5	0
S5	…	0	…	--	S_0	S_6	0
S6	…	1	…	--	—	—	1

```
[…]
input x;
reg  [K-1:0] STAR, MJR;
reg  A,B,F;
[…]
S0:  begin […] STAR<=(x==0)?S0:S1; end
S1:  begin […] STAR<=S2; end
S2:  begin […] STAR<=(A!=B)?S5:S3; end
S3:  begin […] STAR<=(gamma(A)==0)?S1:S4; end
S4:  begin […] STAR<=S5; end
S5:  begin […] MJR<=(x==1)?S4:(A!=F)?S1:S0; STAR<=S6; end
S6:  begin […] STAR<=MJR; end
```

c_0 00

c_1 01

c_2 10

Parte operativa in Verilog

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

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

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

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

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

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

Sottoliste (cont.)

```
S0:   begin  /*elaborazioni*/ MJR<=S1; STAR<=Ssub1; end
S1:   begin  /*elaborazioni*/; end
[…]
Sx:     begin  /*elaborazioni*/ MJR<=Sx+1; STAR<=Ssub1; end
Sx+1:   begin  /*elaborazioni*/; end
[…]
Ssub1:  begin  /*elaborazioni*/ end
[…]
SsubK:  begin  /*elaborazioni*/ STAR<=MJR; end
```

RET

Lista principale

Sottolista

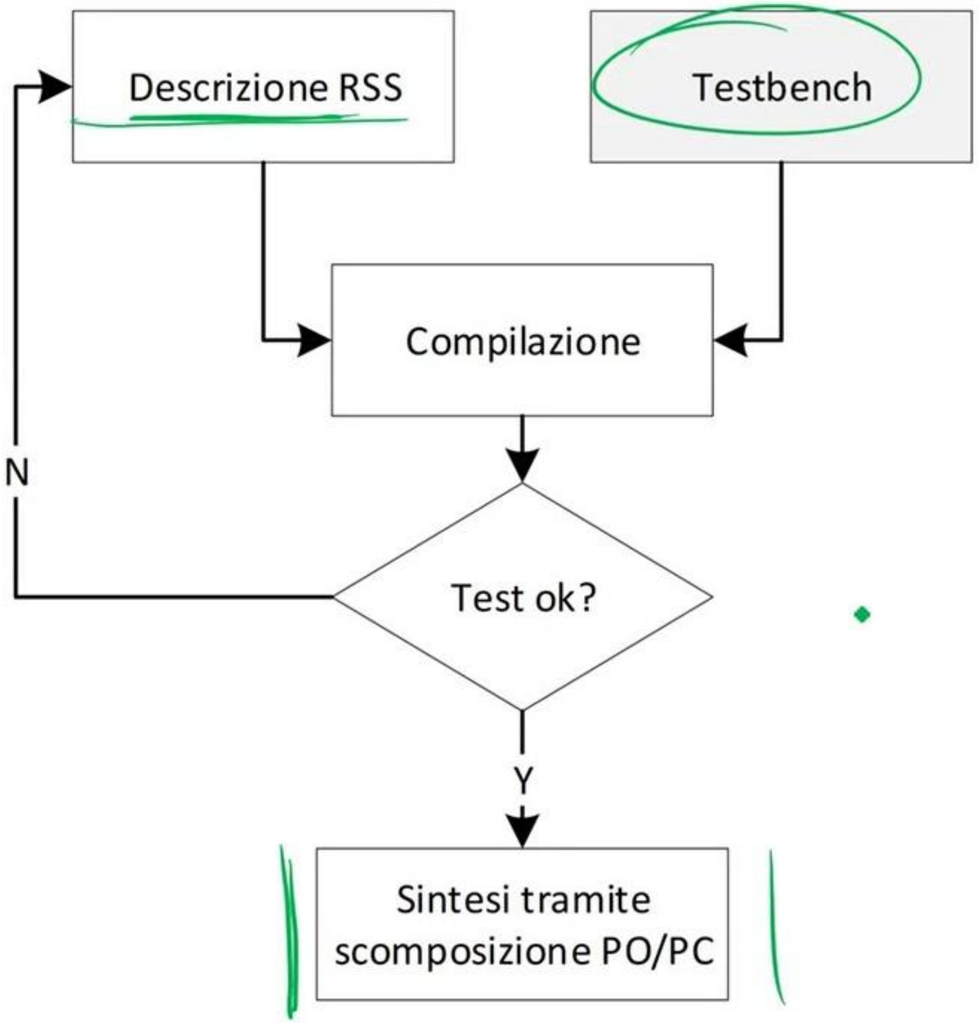
- Massimo **un livello di annidamento** di sottoliste
  - Altrimenti ci vorrebbe una **pila di registri MJR**
  - Tecnica nota: si chiama ***μ*-pila**
  - Ma va oltre gli scopi di un corso come questo

μ-salti ed utilizzo di MJR

- Quando usare MJR:
  - Quando è esplicitamente richiesto in un esercizio
  - Quando non se ne può fare a meno

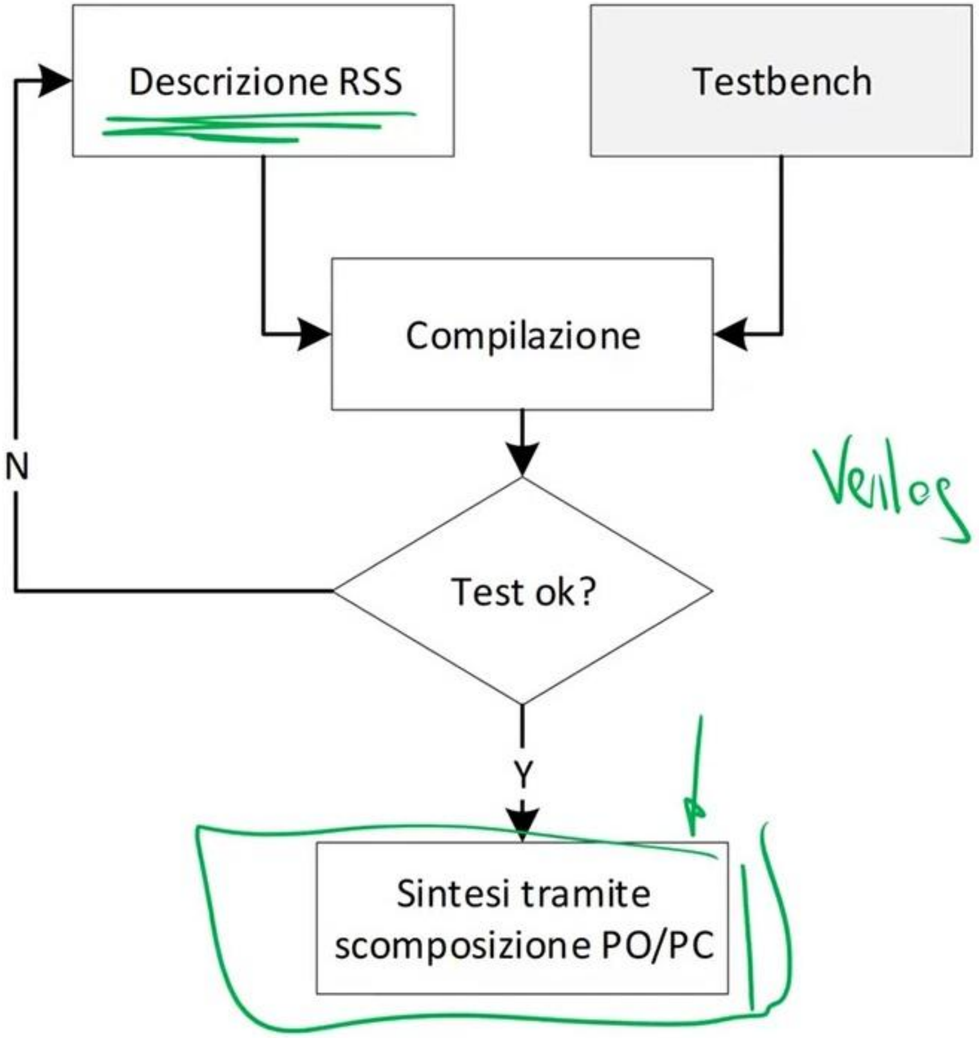
- In tutti gli altri casi, limitarsi a scrivere descrizioni con μ-salti a due vie
  - Altrimenti è impossibile fare la sintesi PO/PC

Note su descrizione e sintesi di RSS complesse



- Sintesi tramite PO/PC è il **punto di arrivo**
  - Agli esami può essere richiesta o meno
  - Ma se **non** richiesta, è comunque il punto di arrivo «mentale»
- **Non sono accettate** descrizioni che non si possono sintetizzare secondo i modelli visti nelle slide precedenti

Note su descrizione e sintesi di RSS complesse



Verilog

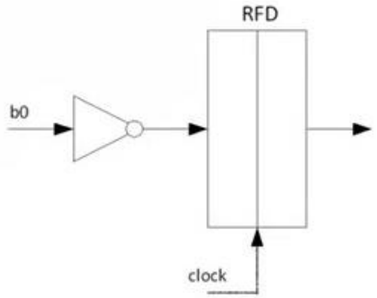
- Sintesi PO/PC
  - Parte operativa
  - Collezione di **registri multifunzionali**
  - Rete combinatoria di condizionamento

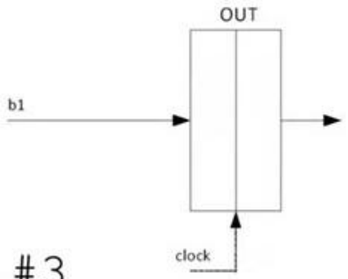
  - Parte controllo
  - RSS di Moore

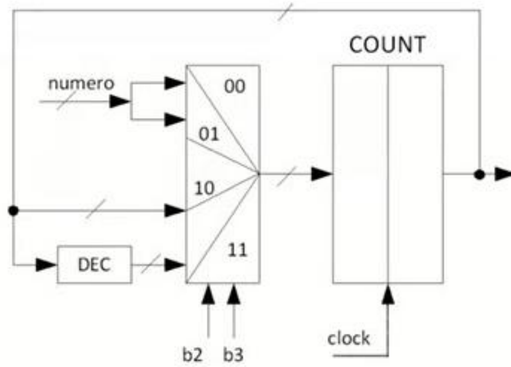
- Per entrambe, **descrizione in Verilog**
  - Perché 1:1 con circuiti noti

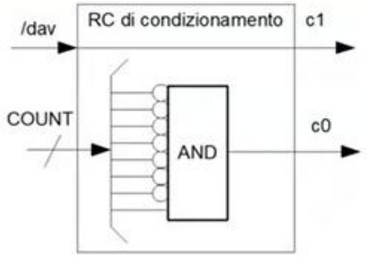
Parte operativa in Verilog

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

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

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

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

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

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



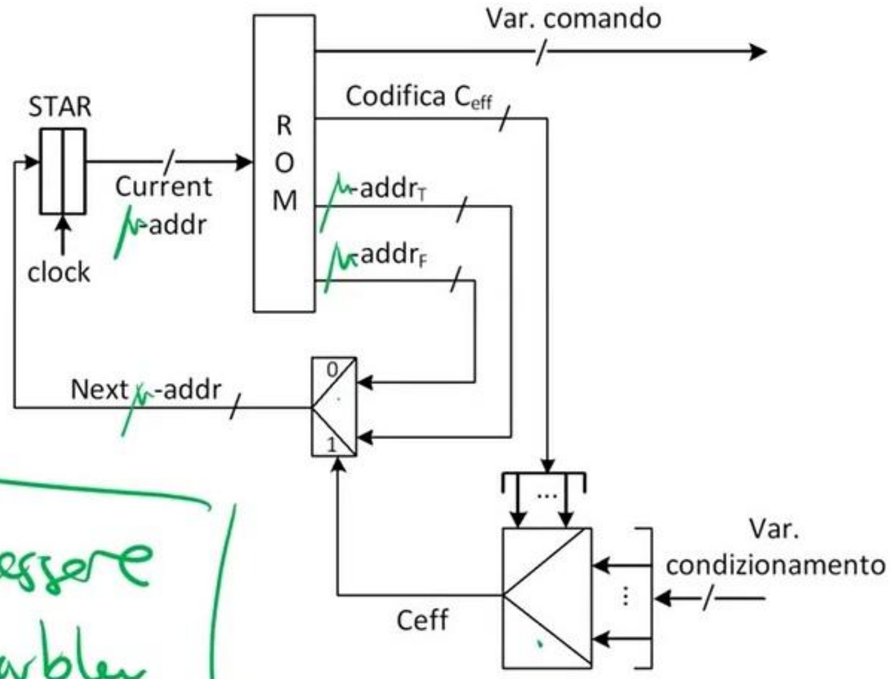




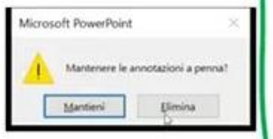


Parte controllo in Verilog

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



AND



Processore Assembler

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