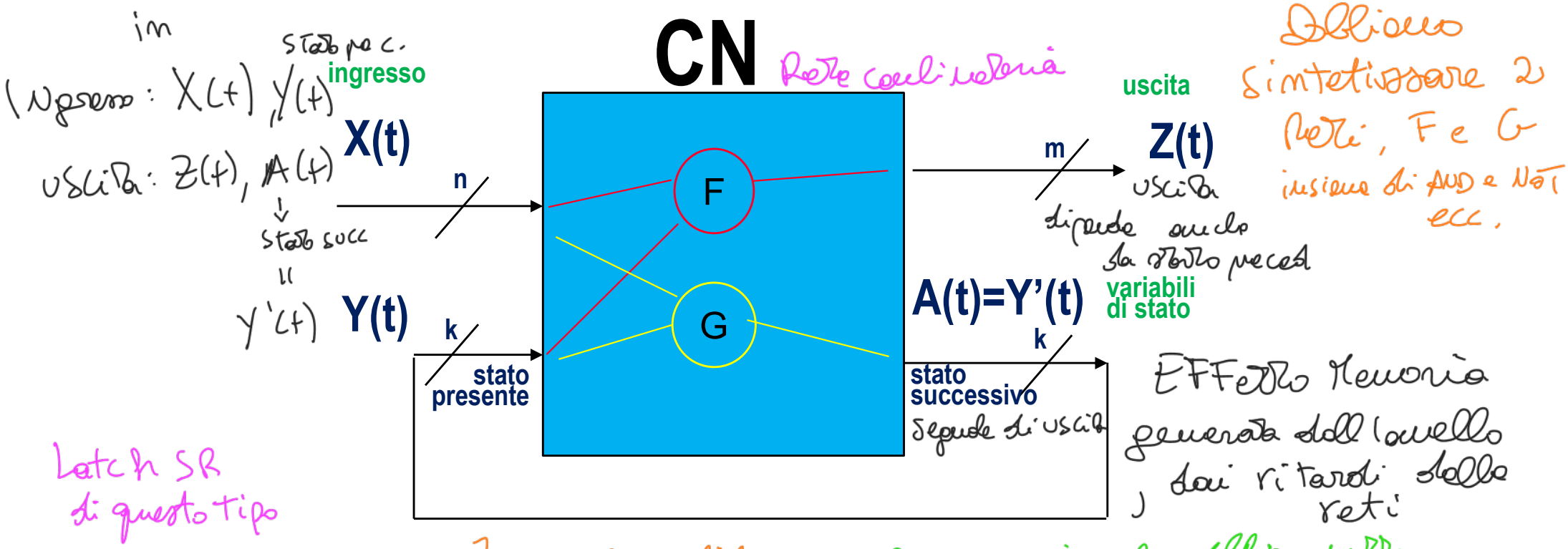
Lezione 6
Reti Logiche Sequenziali di Mealy e di Moore, Contatori e Addizionatori

Sincrone e Asincrone

Macchina di Mealy asincrona

- **Le variabili d'uscita, in un determinato istante, sono funzione del valore degli ingressi e delle variabili di stato**

$$Z(t) = F(X(t), Y(t))$$
$$Y'(t) = G(X(t), Y(t))$$



La rete CN è una rete combinatoria

Macchina di MEALY (sincronizzata)

Pongo un elemento di memoria esplicito

- Le uscite sono funzioni delle variabili di stato e degli ingressi

$$Z(t) = f_L(\ X(t),\ Y(t-1)\ )$$
$$Y(t) = g_L(\ X(t),\ Y(t-1)\ )$$

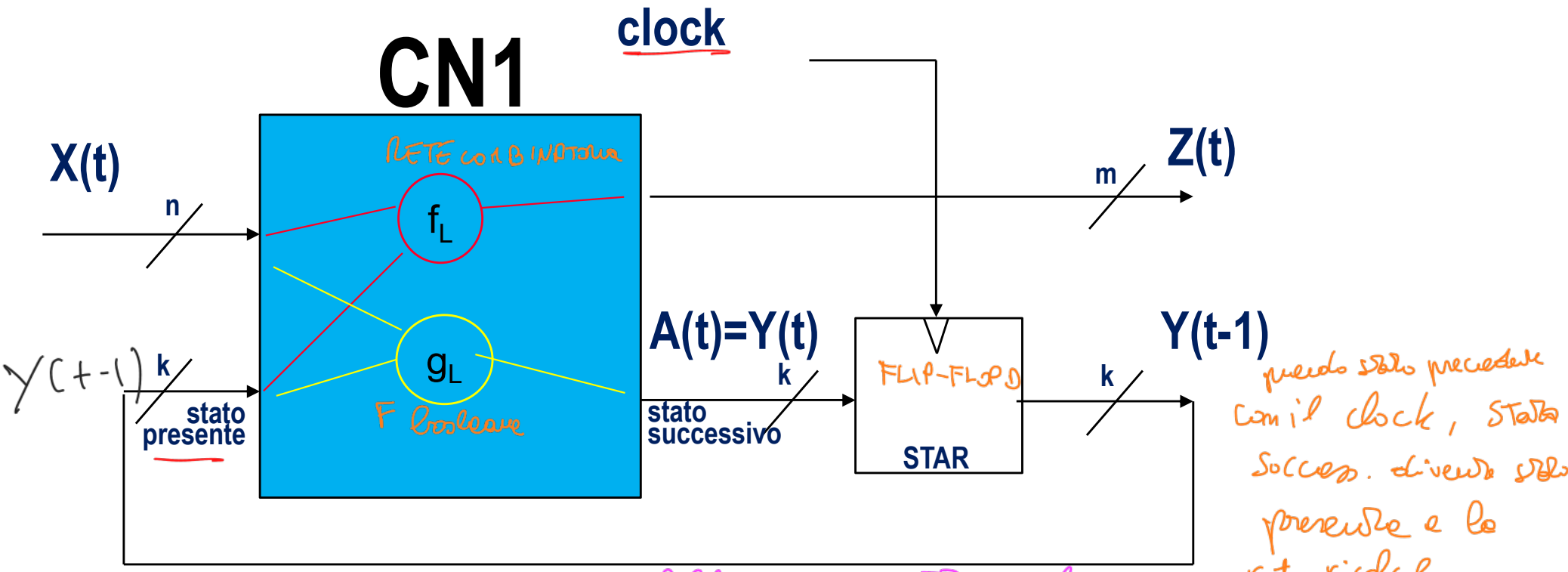
Stato presente

Salvo saldamente lo stato per evitare problemi di stabilità

Salvo stato in un FF

In MEALY: l'uscita (Z) dipende DIRETTAMENTE sia dallo stato (Y) che dagli ingressi (X)

Y(t-1) è lo stato presente
Y(t) è lo stato successivo



quando stato precedente con il clock, stato successivo diventa stato presente e la rete ricalcola

ad ogni colpo di clock, salvo lo stato

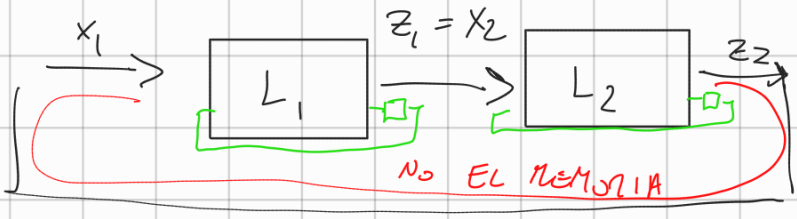
Dobbiamo aspettare che la rete si stabilizzi, per memorizzare. Calcolo ad ogni colpo clock

STAR (STATUS REGISTER) è costituito da Flip-Flop D

Aspetto il colpo di Clock per scrivere nel FF-D e calcolare uscita e stato succ lo stato successivo per minimizzare GLITCH

Difetto: se metto Macchine Mealy sincrone, in cascata e chiudo anello

Mealy sincr. in cascata



→ ora ho un percorso senza elemento di memoria

si crea un percorso interno, e se chiudo un anello, non avrei un elemento di memoria (1 per la uscita), creo un anello asincrono se voglio chiudere l'uscita dell'anello

USCITA DIPENDE solo da STATO prec.

Macchina di MOORE (sincronizzata)

l'elemento di memoria in mezzo fra ingresso e uscita

- Le variabili d'uscita, in un determinato istante, sono funzione delle sole variabili di stato

Separo f_M e g

note comb. separata

$$Z(t)=f_M(\ Y(t-1)\ )$$

solo funzione dello stato interno precedente

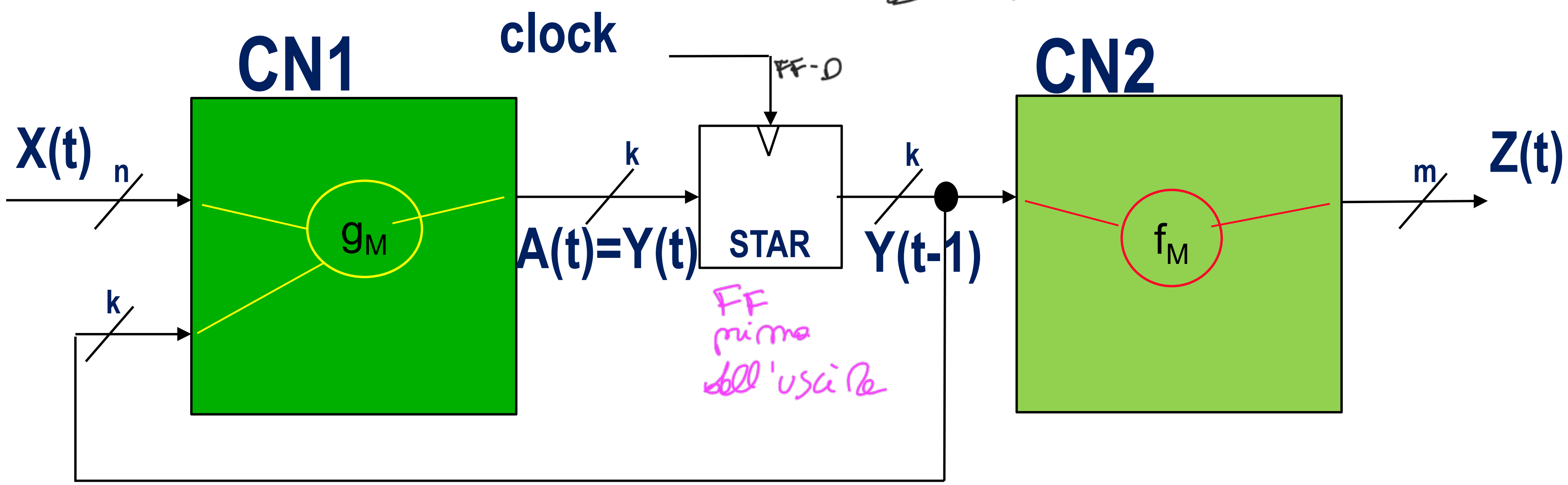
$$Y(t)=g_M\ (\ X(t),\ Y(t-1)\ )$$

X e stato presente

latch, no FF

In MOORE: l'uscita (Z) dipende DIRETTAMENTE solo dallo stato (Y)

Y(t-1) è lo stato presente
Y(t) è lo stato successivo



le f ora sono due blocchi separati

{ risolvo il problema, posso mettere reti in cascata evitando di creare un anello instabile

Macchina di Mealy Ritardata (sincronizzata)

- **Le uscite sono funzioni delle variabili di stato e degli ingressi, ma risultano sincronizzate**

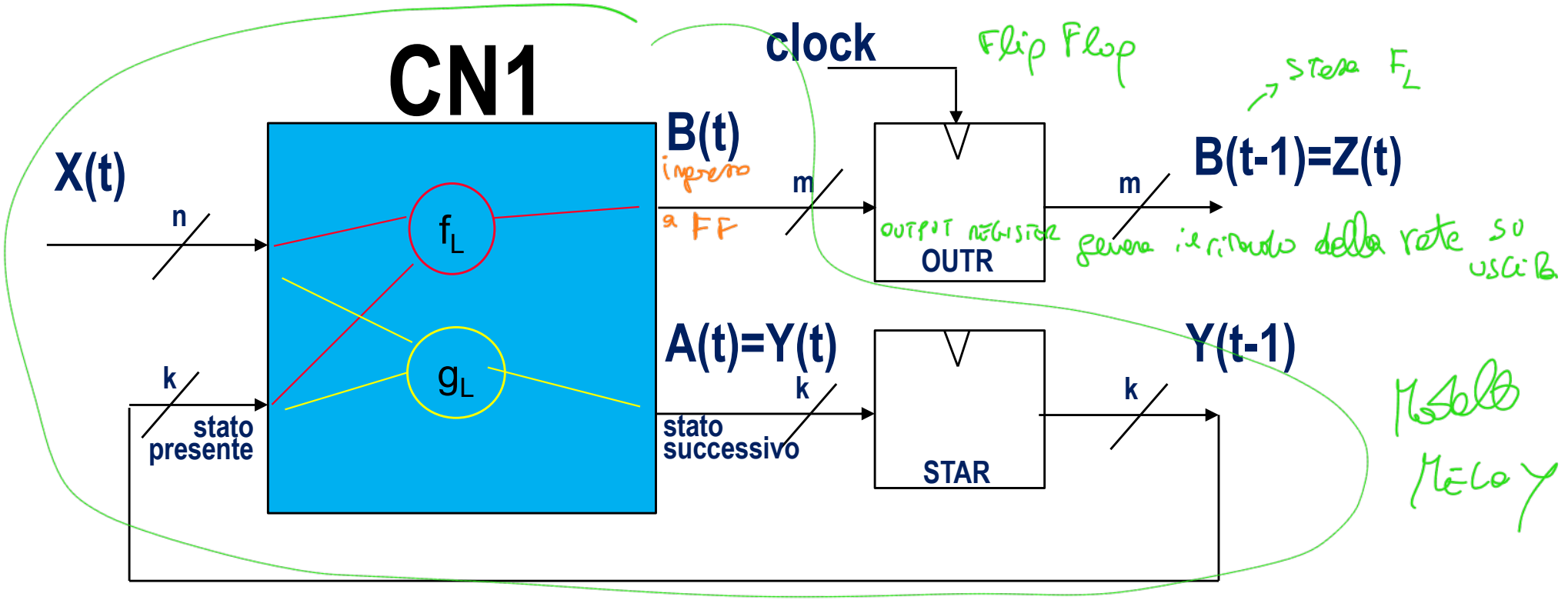
essendo $B(t)=f(X(t),Y(t-1))$...

$Z(t)=B(t-1)=f_L(\ X(t-1),\ Y(t-2)\ )$

$Y(t)=g_L(\ X(t),\ Y(t-1)\ )$

In MEALY: l'uscita (Z) dipende DIRETTAMENTE sia dallo stato (Y) che dagli ingressi (X)

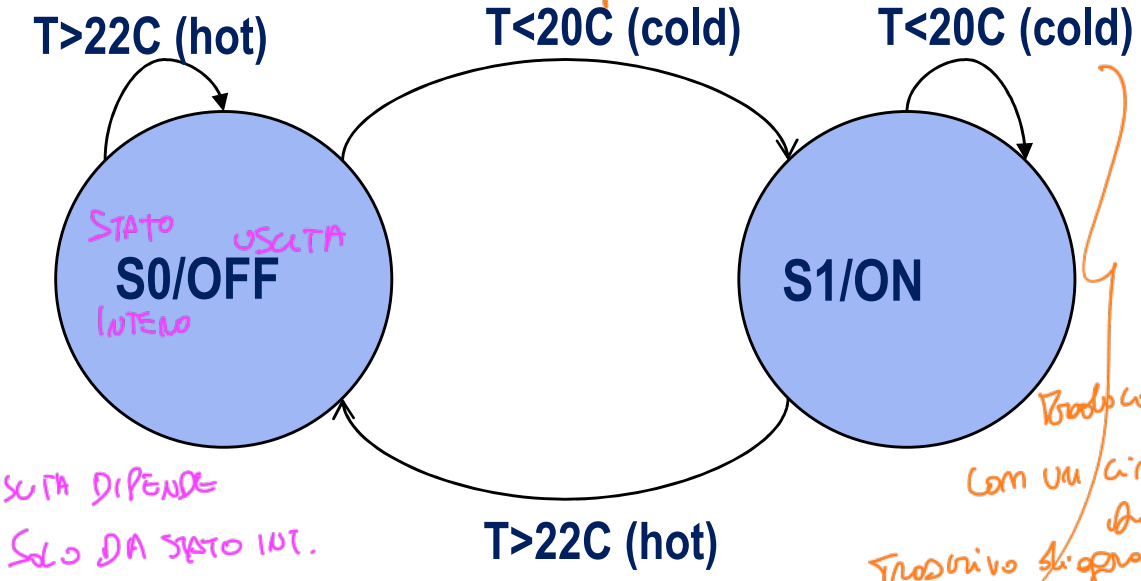
Y(t-1) è lo stato presente
Y(t) è lo stato successivo

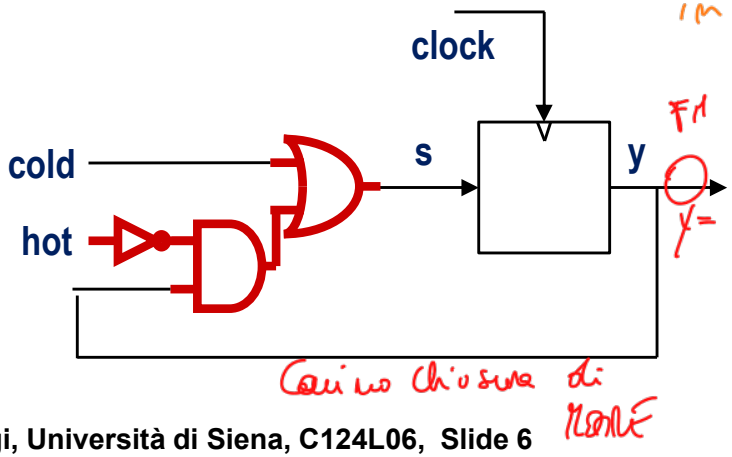


Tecnica generale per implementare una FSM con Moore

- S0 → Riscaldamento Spento (z=OFF)
- S1→ Riscaldamento Acceso (z=ON)

In genere MOORE produce più stati, ma si può implementare usando solo LUT





STEP1) TABELLA DELLE TRANSIZIONI

STATO PRESENTE (Y)	cold,hot (X) 00	01	11	10
S0	S0	S0	-	S1
S1	S1	S0	-	S1

STATO SUCCESSIVO (S)

STEP2) CODIFICA DEGLI STATI

Stato	S=Y
S0	0
S1	1

STEP3) SINTESI DELLA MAPPA PER S

y \ cold/hot	00	01	11	10
0	0	0	-	1
1	1	0	-	1

s

$s = cold + \overline{hot} * y$

STEP4) SINTESI DELLE USCITE Z

y (STATO)	S0	S1
Z	0	1

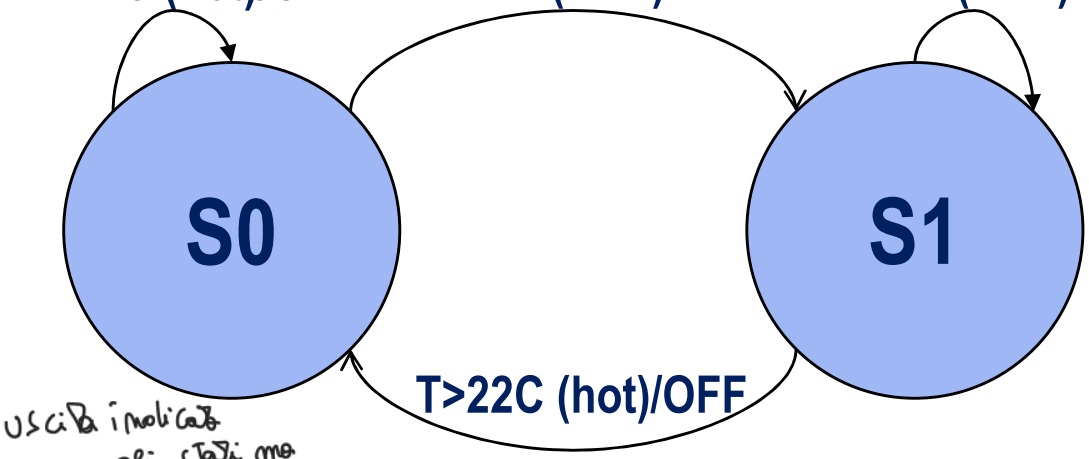
z

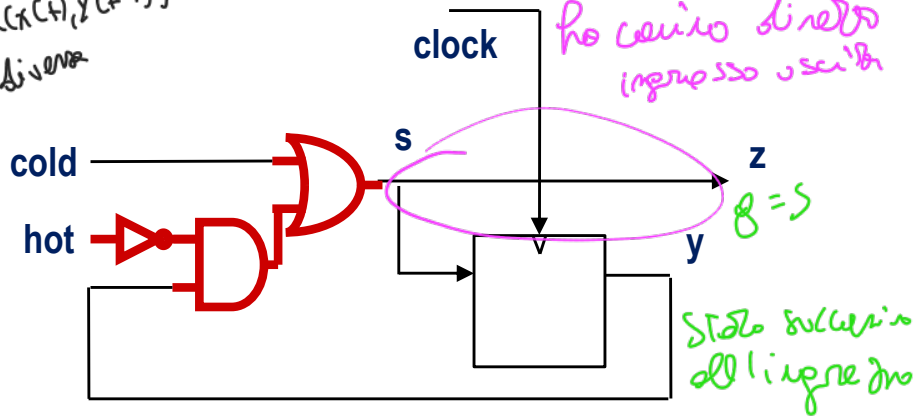
$z = y$

Tecnica generale per implementare una FSM con Mealy

- S0 → Riscaldamento Spento (z=OFF)
- S1→ Riscaldamento Acceso (z=ON)

In genere MEALY produce meno stati, ma può richiedere sia più FF che logica aggiuntiva





STEP1) TABELLA DELLE TRANSIZIONI

STATO PRESENTE (Y) \ cold,hot (X)	00	01	11	10
S0	S0/0	S0/0	- / -	S1/1
S1	S1/1	S0/0	- / -	S1/1

STATO SUCCESSIVO/USCITA (S/Z)

STEP2) CODIFICA DEGLI STATI

Stato	S=Y
S0	0
S1	1

STEP3) SINTESI DELLA MAPPA PER S

y \ cold/hot (CN1)	00	01	11	10
0	0	0	-	1
1	1	0	-	1

s

$s = cold + \overline{hot} * y$

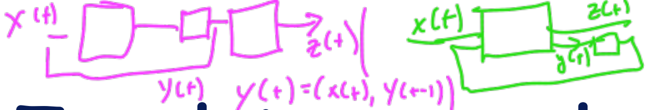
STEP4) SINTESI DELLE USCITE Z (in generale diverso da S)

y \ cold/hot (CN2)	00	01	11	10
0	0	0	-	1
1	1	0	-	1

z

$z = cold + \overline{hot} * y$

=s (in questo caso)



Template generale per Moore in Verilog

```
module Moore(z,x,clock,reset_);
  input          clock, reset_;
  input  [N-1:0]  x;
  output [M-1:0]  z;
  reg    [K-1:0]  STAR;
  parameter S0=codifica0, …, SKmeno1=codificaKmeno1;
  always @(reset_==0) #1 STAR<=stato_iniziale;

  assign z=(STAR==S0)?codificaf0:
           (STAR==S1)?codificaf1:
            …
           /*(STAR==SKmeno1)*/ codificafKmeno1;
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0:  STAR<=g0(x);
      S1:  STAR<=g1(x);
       …
      SKmeno1:  STAR<=gKmeno1(x);
    endcase
endmodule
```

NOTA: Y non è altro che STAR

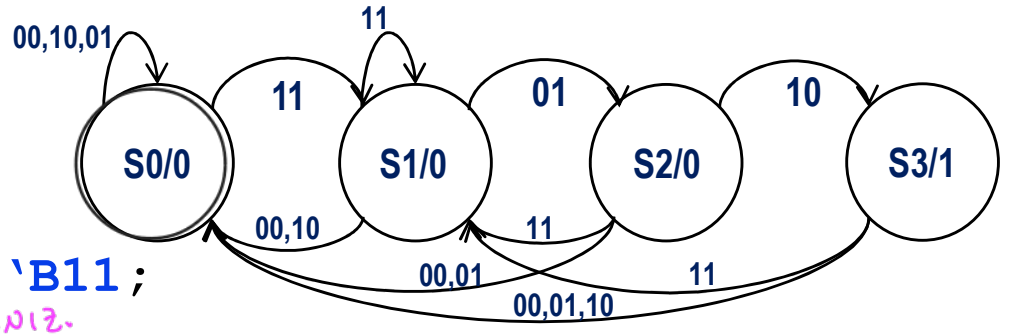
$f_M(Y)$

In gM la dipendenza dalla Y è attraverso il «casex(STAR)»

$g_M(X,Y)$

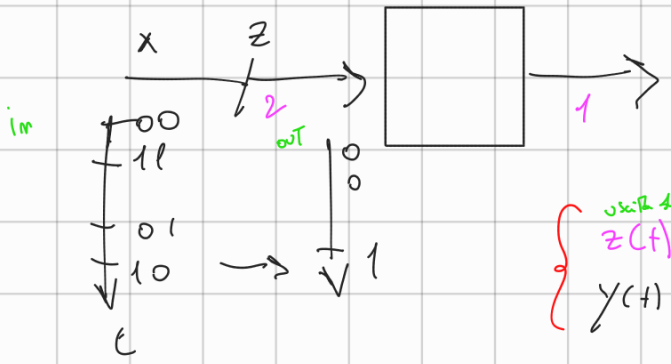
Esempio - riconoscitore di sequenza 11,01,10 con Moore

```
module ricseq110110moore(z,x,clock,reset_);
  input            clock, reset_;
  input  [1:0]      x;
  output            z;
  reg [1:0]         STAR;
  parameter S0='B00,S1='B01,S2='B10,S3='B11;
  always @(reset_==0) #1 STAR<=S0;
  assign z=(STAR==S3)?1:0;
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: STAR<=(x=='B11)?S1:S0;
      S1: STAR<=(x=='B01)?S2:(x=='B11)?S1:S0;
      S2: STAR<=(x=='B10)?S3:(x=='B11)?S1:S0;
      S3: STAR<=(x=='B11)?S1:S0;
  endcase
endmodule
```

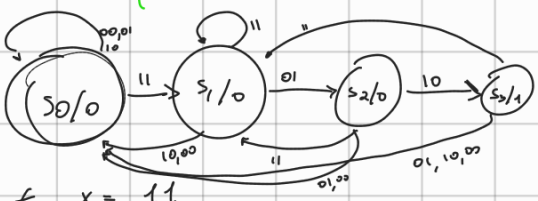


Moore

Voglio un sistema digitale



uscita va a 1 se riconosco un certo pattern



t, x = 11

$$\begin{cases} z(t) = F(y(t-1)) \\ y(t) = g(x(t), y(t-1)) \end{cases}$$

uscita dipende da stato

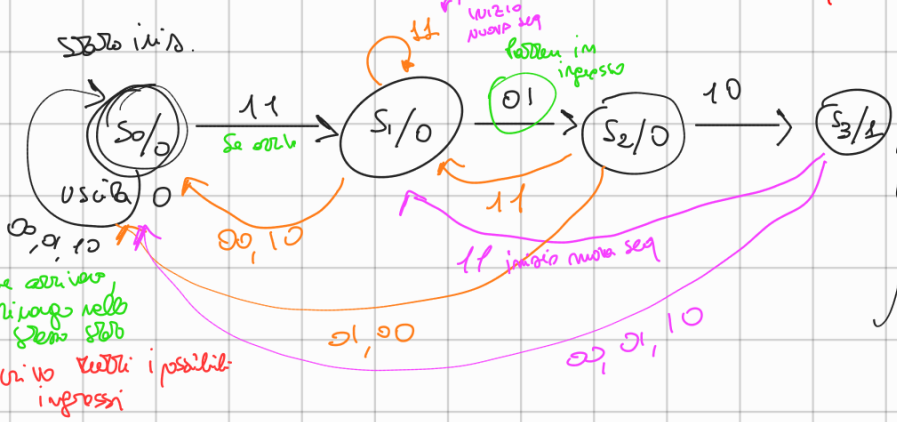
mi ricordo di dove eromino

Moore: uscita è solo funzione dello stato

RICONOSCITORE SEQUENZA del Tipo 11, 01, 10 (Moore)

dell'uscita

11 prodotto come inizio di sequenza



altri casi.

- Descrivo il diagramma, codifico

Stato iniziale lo indico con 2 cerchi

descrivo il mio diagramma, codifico

Descrivo Moore uscita. => Rete va a 1 quando sono in S_3

quando STAR = S_3

Template generale per Mealy in Verilog

```
module Mealy(z,x,clock,reset_);
  input           clock, reset_;
  input  [N-1:0]   x;
  output [M-1:0]   z;
  reg    [K-1:0]   STAR;
  parameter  S0=codifica0, …, SKmeno1=codificaKmeno1;
  always @(reset_==0) #1 STAR<=stato_iniziale;
            sia STAR che x
  assign z=(STAR==S0)?f0(x):
           (STAR==S1)?f1(x):
            …
           /*(STAR==SKmeno1)*/ fKmeno1(x);
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: STAR<=g0(x);
      S1: STAR<=g1(x);
       …
      SKmeno1: STAR<=gKmeno1(x);
    endcase
endmodule
```

codifica stato

def. stato in.

NOTA: Y non è altro che STAR

$z = f(x(t), y(t-1))$

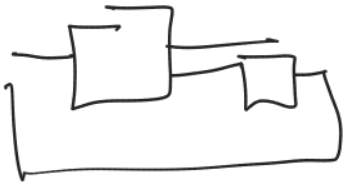
$f_L(X,Y)$ $z = f_L(x(t), y(t-1))$

come reg. stato succ. a partire da un ingresso e stato prec.

$g_L(X,Y)$

In gL la dipendenza dalla Y è attraverso il «casex(STAR)»

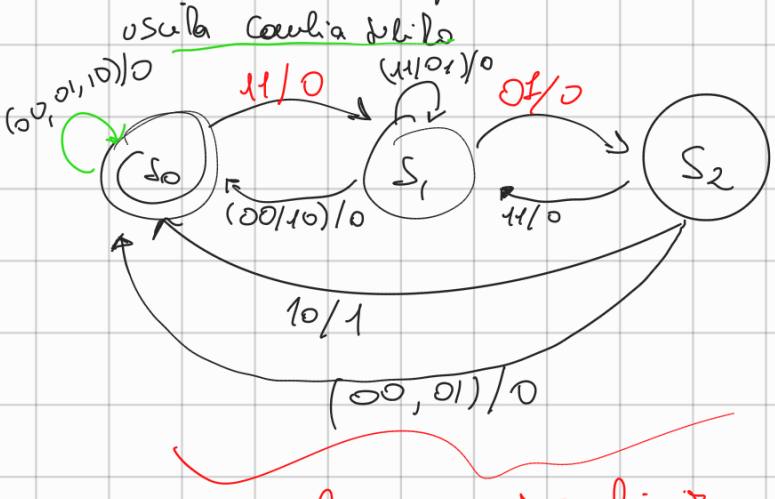
$Y = g_L(x(t), y(t-1))$



Stesso riconoscimento con Mealy

Riconoscere sequenza 11, 01, 10 (MEALY)

- Varia subito uscita in base all'ingresso e stato prec.
- No 3 stati
- Cambio stato è sull'arco



- negli stati non c'è la specifica dell'uscita ma c'è il calcio stato è sull'arco

Mealy, cambio stato sull'arco, dalla transizione

Mio con Moore avevo bisogno di uno stato in più perché l'uscita dipende solo dallo stato precedente, dunque posso leggere solo da esso, non posso dire

" Se stato è S2 e ingresso è 11, l'uscita è 1 ",

ma posso solo dire " se sono in S3, allora verifica che si è verificata la condizione, allora uscita è 1 "

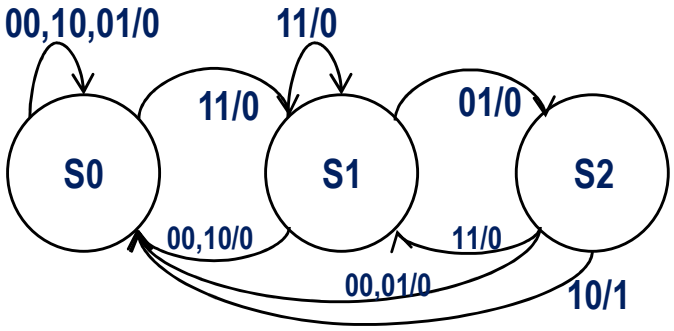
Moore infatti $z(t) = f(y(t-1))$

Con Mealy invece posso farlo, in quanto posso confrontare input e stato insieme

$$z(t) = f(x(t), y(t-1))$$

Esempio - riconoscitore di sequenza 11,01,10 con Mealy

```
module ricseq110110mealy(z,x,clock,reset_);
  input           clock, reset_;
  input  [1:0]     x;
  output           z;
  reg [1:0]        STAR;
  parameter  S0='B00, S1=`B01, S2=`B10;
  always @(reset_==0) #1 STAR<=S0;
  assign z=((STAR==S2)&(x==`B10))?1:0;
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: STAR<=(x=='B11)?S1:S0;
      S1: STAR<=(x=='B01)?S2:(x=='B11)?S1:S0;
      S2: STAR<=(x=='B11)?S1:S0;
  endcase
endmodule
```



Template generale per Mealy Ritardato in Verilog

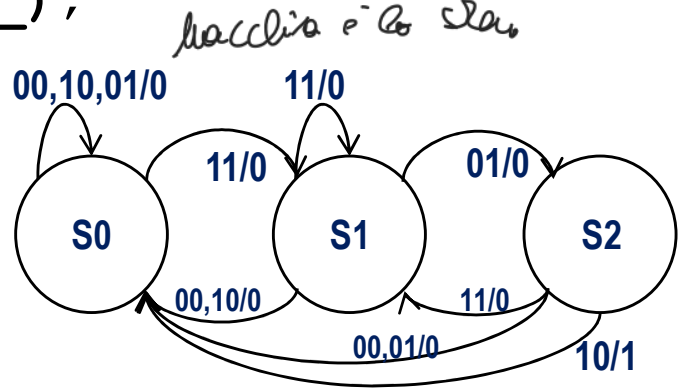
```
module MealyRitardato(z,x,clock,reset_);
  input            clock, reset_;
  input  [N-1:0]   x;
  output [M-1:0]   z;
  reg    [K-1:0]   STAR;
  reg    [M-1:0]   OUTR;
  parameter  S0=codifica0, …, SKmeno1=codificaKmeno1;
  always @(reset_==0) #1 begin STAR<=…; OUTR<=…; end;
assign z=OUTR;
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: begin OUTR<=f0(x); STAR<=g0(x); end
      S1: begin OUTR<=f1(x); STAR<=g1(x); end
       …
      SKmeno1: begin OUTR<=fKmeno1(x); STAR<=gKmeno1(x); end
    endcase
endmodule
```

NOTA: Y non è altro che STAR

$f_L(X,Y)$ $g_L(X,Y)$

Esempio - riconoscitore di sequenza 11,01,10 con Mealy Rit

```
module ricseq110110mealyritardato(z,x,clock,reset_);
  input            clock, reset_;
  input  [1:0]      x;
  output            z;
  reg [1:0]         STAR;
  reg               OUTR;
  parameter  S0='B00, S1=`B01, S2=`B10;
  always @(reset_==0) #1 begin OUTR<=0; STAR<=S0; end
  assign      z=OUTR;
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: begin OUTR<=0; STAR<=(x=='B11)?S1:S0; end
      S1: begin OUTR<=0; STAR<=(x=='B01)?S2:(x=='B11)?S1:S0; end
      S2: begin OUTR<=(x=='B10)?1:0; STAR<=(x=='B11)?S1:S0; end
  endcase
endmodule
```



MSF è la stessa con Mealy RIT

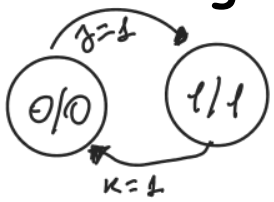
Diag. temporale come Moore, ma ha usato meno stati

Come Moore, però qui ho meno stati; potrebbe essere un vantaggio

Flip-Flop JK

- Il FF-JK è tale che in corrispondenza del fronte in salita del clock:
  - per JK=10 l'uscita va a 1,
    per JK=01 l'uscita va a 0,
    per JK=00 il FF conserva (l'uscita resta inalterata)
    per JK=11 il FF commuta (l'uscita diventa il suo negato)

CON MOORE

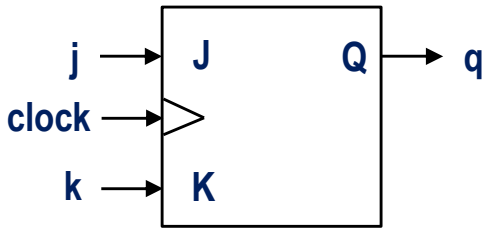



```
module FFJK(q,j,k,clock,reset_);
   input            clock, reset_;
   input             j,k;
   output            q;
   reg               STAR;
   parameter  S0=0, S1=1;
   assign q=(STAR==S0)?0:1;
   always @(reset_==0) #1 STAR<=0;
   always @(posedge clock) if (reset_==1) #3
     casex(STAR)
       S0: STAR<=(j==1)?S1:S0;
       S1: STAR<=(k==1)?S0:S1;
     endcase
endmodule
```

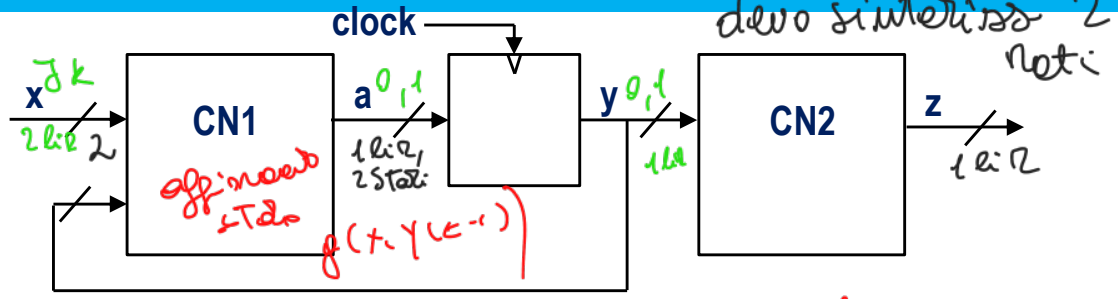
JK	Ck	Q
XX	1	Q
XX	0	Q
XX	⊓	Q
00	⊓	Q
01	⊓	0
10	⊓	1
11	⊓	/Q

Tabella di Verità

Simbolo del JK



Sintesi del FF-JK con Moore



- 2 stati → STAR ha 1 bit
- → a ha 1 bit, y ha 1 bit
  z ha 1 bit, (x ha 2 bit)

per JK=10 l'uscita va a 1,
per JK=01 l'uscita va a 0,
per JK=00 il FF conserva
per JK=11 il FF commuta

Step 1

Tabella delle transizioni e delle uscite

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

CONSERVO COMMUTO

Stato	y0
S0	0
S1	1

codifica

CN1

y \ jk	00	01	11	10
0	0	0	1	1
1	1	0	0	1

a

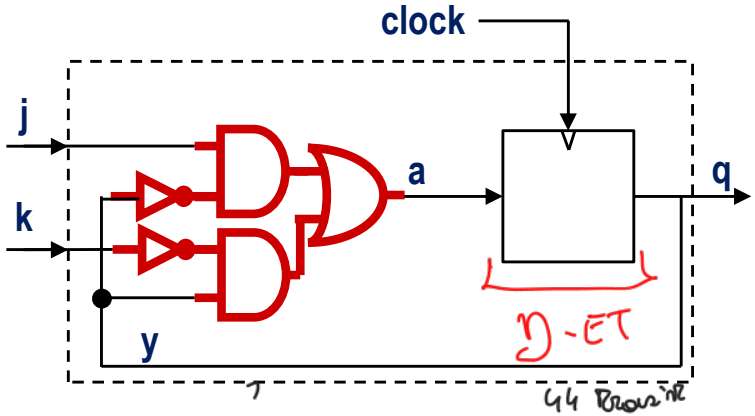
$a = j\bar{y} + \bar{k}y$

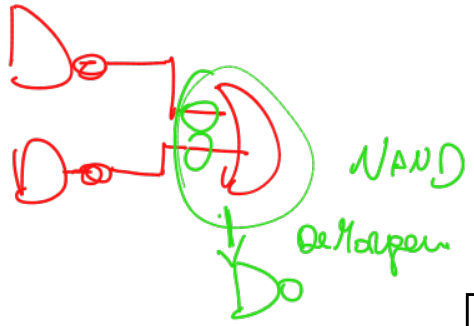
CN2 è un C.C.

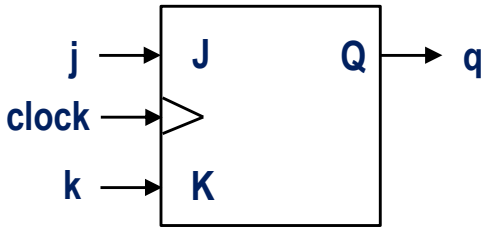
y	0	1
q	0	1

z

$z = y$







Simbolo del JK

Layout simbolico: 60 transistors
(44 per D-ET + 3*NAND + 2*NOT)

FF D 44 + 16 = 60

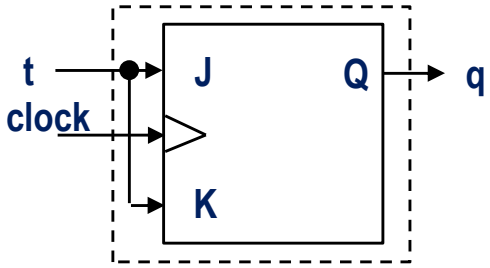
Con FF-JK posso fare un contatore, Contocircuito JK

Flip-Flop T (detto FF contatore)

- Il FF-T è tale che <u>in corrispondenza del fronte in salita del clock:</u>
  - per T=0 il FF conserva (l'uscita rimane inalterata)
  - per T=1 il FF commuta (l'uscita diventa il suo negato)

JK = 00

JK = 11, avviene ad ogni colpo di clock



invece di JK, ho t

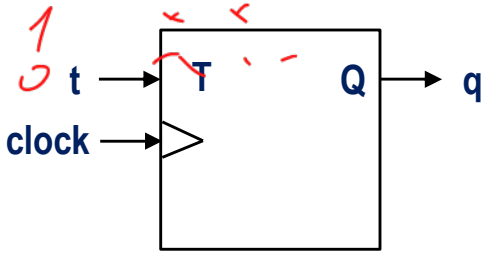
```
module FFT(q,t,clock,reset_);
  input            clock, reset_;
  input             t;
  output            q;
  reg               STAR;
  parameter  S0=0, S1=1;
  assign q=(STAR==S0)?0:1;
  always @(reset_==0) #1 STAR<=stato_iniziale;
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: STAR<=(t==1)?S1:S0;
      S1: STAR<=(t==1)?S0:S1;
    endcase
endmodule
```

Moore

t = 1, commuto, altrimenti t = 0, conservo

al posto di J e K, t

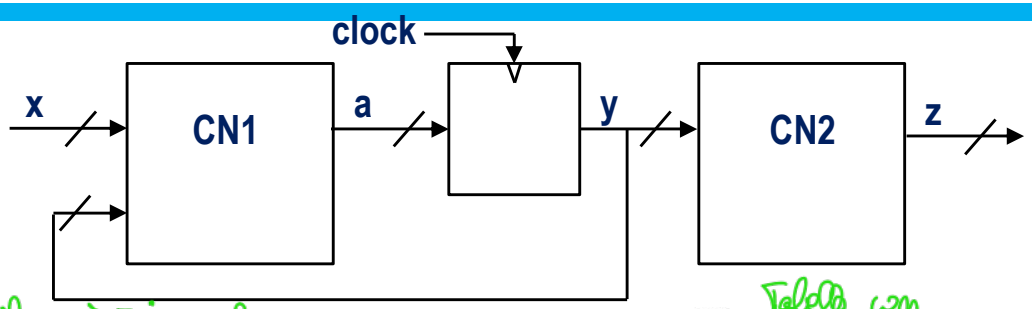
clock	J	K	Q	$\bar{Q}$
↓	0	0	Q	$\bar{Q}$
↓	0	1	0	1
↓	1	0	1	0
↓	1	1	$\bar{Q}$	Q
⌊	X	X	Q	$\bar{Q}$



Abbiamo messo t al posto sia di j (prima riga – S0) che di k (seconda riga – S1)

FSM a mano

Sintesi del riconoscitore di sequenza 11,01,10 con Moore A MANO



codifica stati diversa

- 4 stati → STAR ha 2 bit
- → a ha 2 bit, y ha 2 bit
  z ha 1 bit, (x ha 2 bit)

Codifica stati più furba

Stato	y1 y0
S0	00
S1	01
S2	11
S3	10

Codifica stati in modo intelligente

2 bit per gli stati, sono 4

Tabella con codifica

$y_1 y_0$ \ $x_1 x_0$	00	01	11	10
S^0 00	00	00	01	00
S^1 01	00	11	01	00
S^2 11	00	00	01	10
S^3 10	00	00	01	00

CN1 — ingressi

a_1 a_0

- in corrispondenza degli stati pres. e ingressi, definisco stato succ.

$$a_1 = \bar{x}_1 x_0 \bar{y}_1 y_0 + x_1 \bar{x}_0 y_1 y_0$$

$$a_0 = x_1 x_0 + x_0 \bar{y}_1 y_0$$

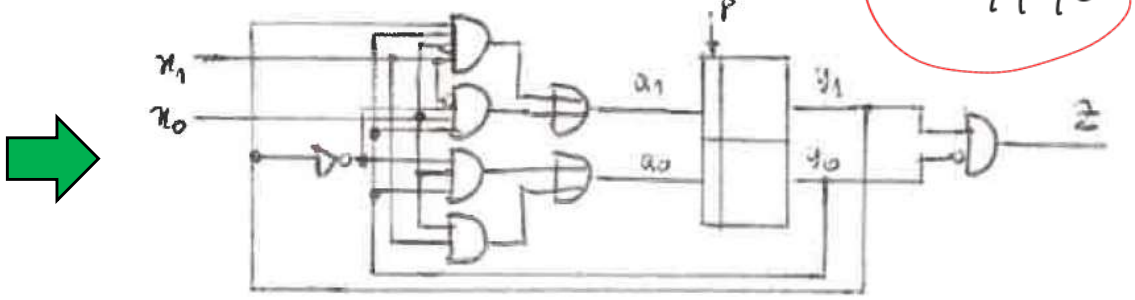
CN2

Codifica stati

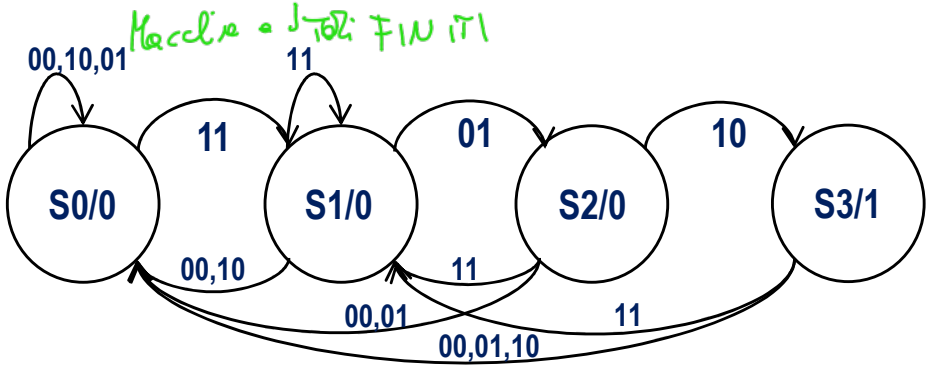
$y_1 y_0$	z
00	0
01	0
11	0
10	1

$z = y_1 \bar{y}_0$

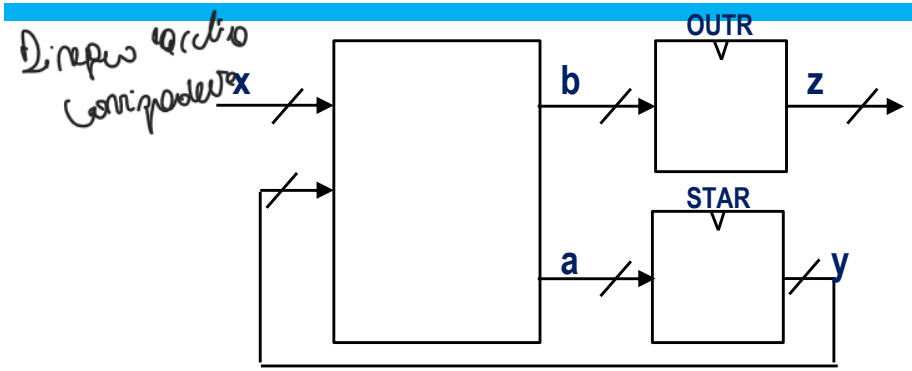
$z = y_1 \bar{y}_0$ Con Mintermi



Macchine a stati FINITI



Sintesi del riconoscitore di sequenza 11,01,10 con Mealy Rit



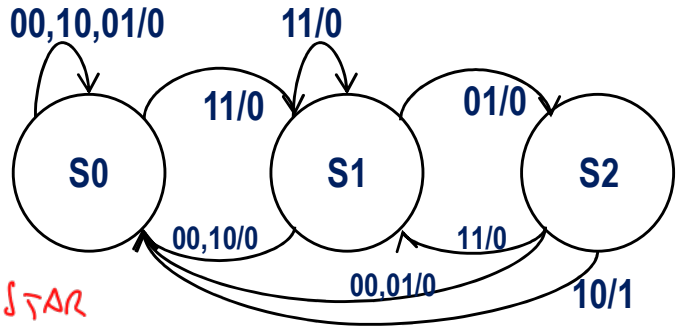


Tabella delle transizioni e delle uscite

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
S3	x	x	x	x

Stato	y1y0
S0	00
S1	11
S2	01

y_1y_0 / x_1x_0	00	01	11	10
00	00	00	11	00
01	00	00	11	00
11	00	01	11	00
10	--	--	--	--

a_1a_0

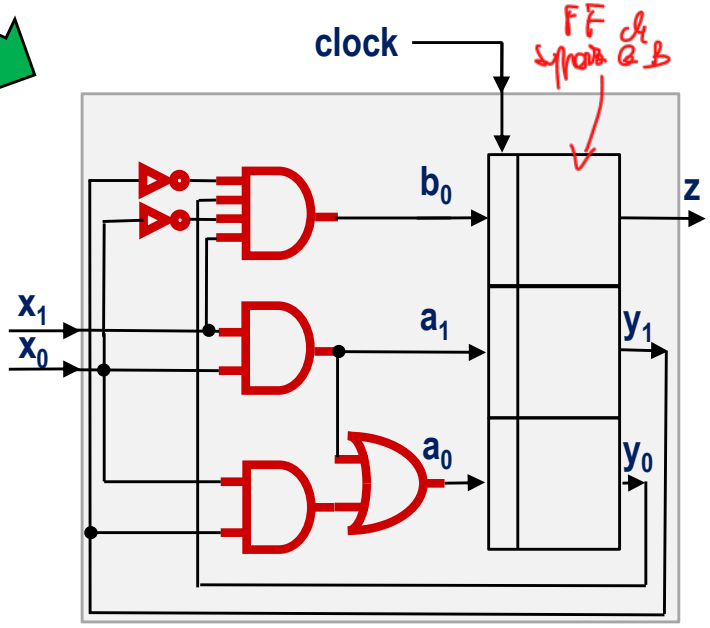
y_1y_0 / x_1x_0	00	01	11	10
00	0	0	0	0
01	0	0	0	1
11	0	0	0	0
10	-	-	-	-

b_0

$a_0 = x_1 x_0 + x_0 y_1$

$a_1 = x_1 x_0$

$b_0 = x_1 \bar{x}_0 \bar{y}_1 y_0$



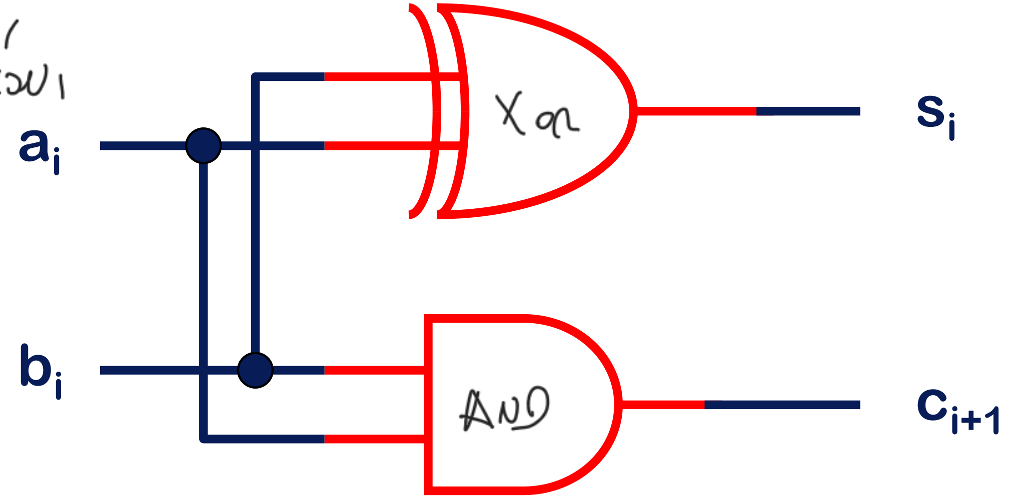
Half Adder in VERILOG

ADDER CON CARRY SOLO USCITA

- Somma di due bit (senza riporto in ingresso)

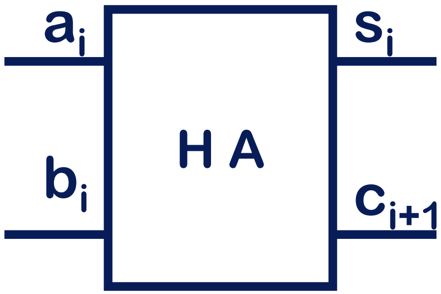
2 uscite, 2 FUNZIONI

a_i	b_i	s_i	c_{i+1}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



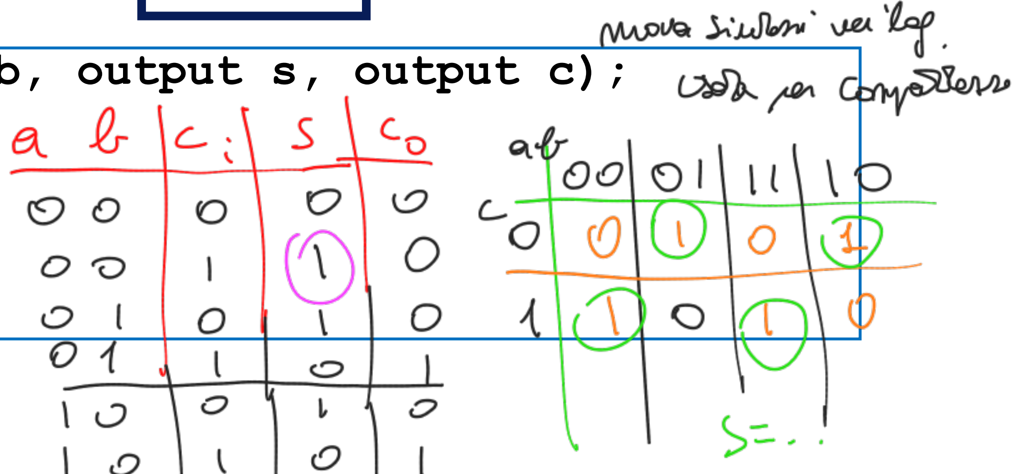
$$s_i = a_i \oplus b_i$$

$$c_{i+1} = a_i \cdot b_i$$




```
module half_adder(input a, input b, output s, output c);
    assign s = a ^ b;
    assign c = a & b;
endmodule
```

2 reti combinatorie



Full Adder in VERILOG

anche carry in ingresso, così posso fare un sommatore di n bit

s_i = generico bit di somma

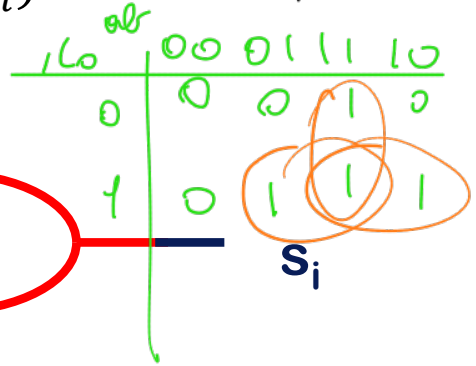
$$s_i = a_i \oplus b_i \oplus c_i = a_i \oplus (b_i \oplus c_i)$$

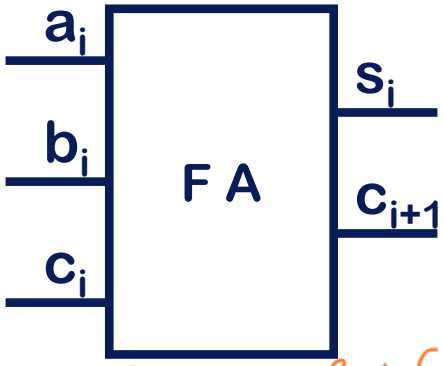
$$c_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i$$

riporto

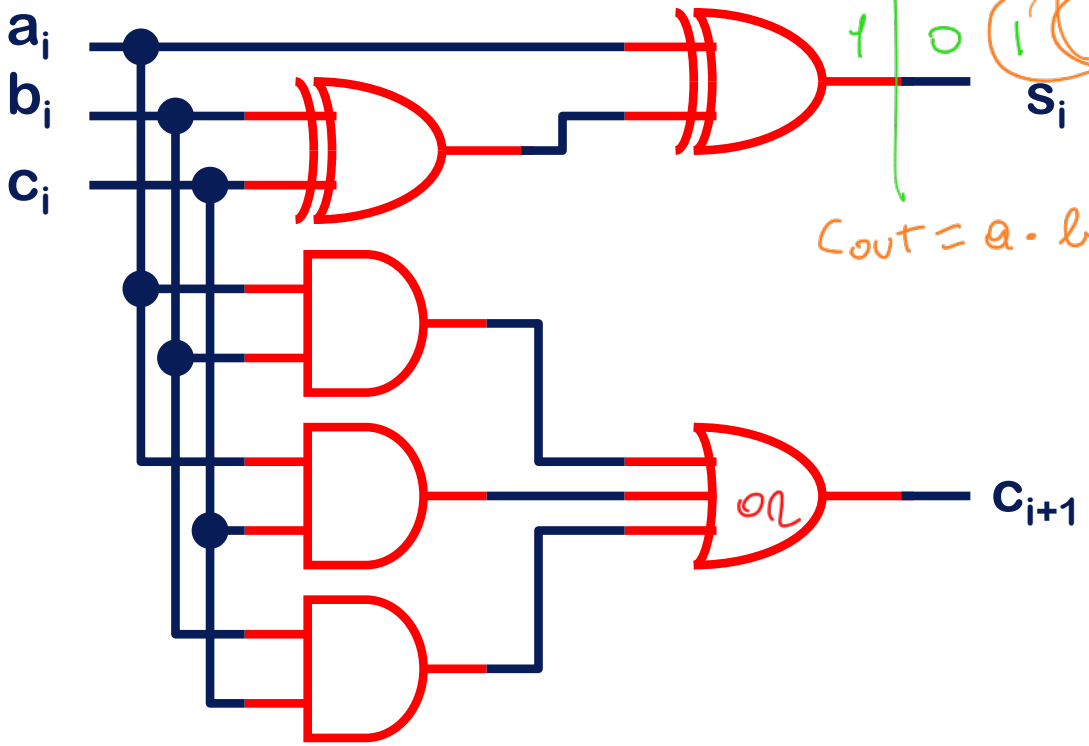
faccio ripple carry, vi parlo con ALU

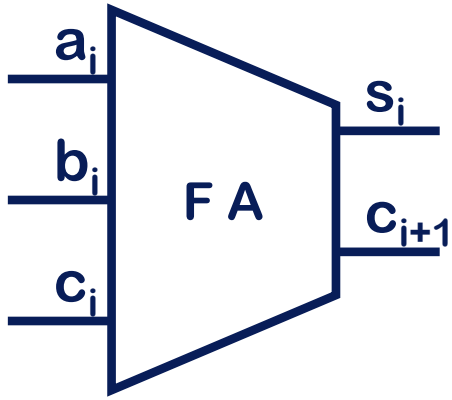
s_i = XOR tra a, b, c





$C_{out} = a \cdot b + c \cdot a + c \cdot b = a \cdot b + c(a + b)$






```
module full_adder(input a,input b,input c_in, output s,output c);
        assign s = a ^ (b ^ c_in);
        assign c = (a & b) | (c_in & (a | b));
endmodule
```

c_in, con ingresso della XOR

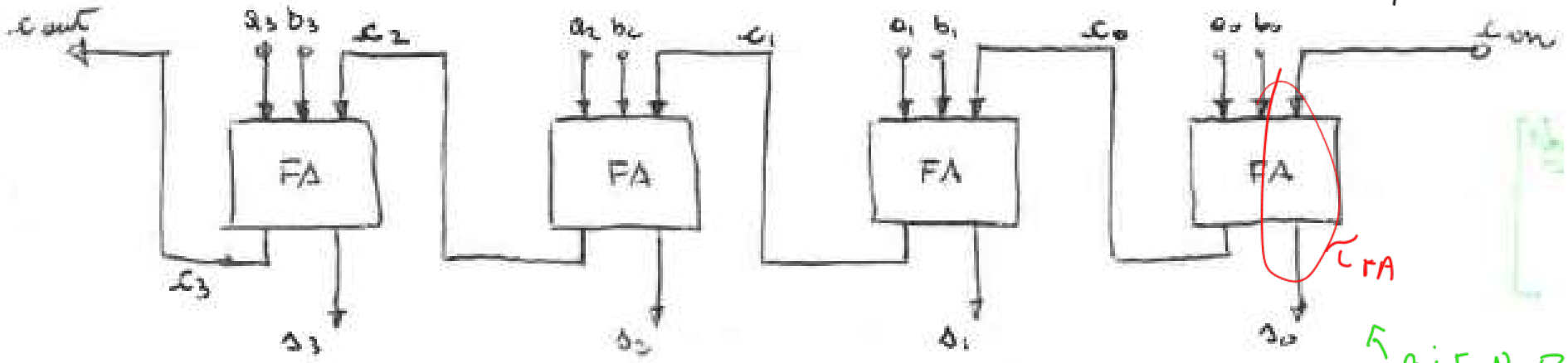
^ XOR

Sommatore Parallelo con riporto seriale

Sommatore a n bit, bit messi in parallelo

Problema => risultato sull'ultimo sommatore arriva dopo che tutti gli altri hanno fatto la somma

Sommatore ad n bit



Problema: il risultato finale, c_{out} arriva solo quando sono disponibili i precedenti

$$\tau_{cout} = N \cdot \tau_C$$

$$\tau_{tot} = (N-1) \cdot \tau_C + \tau_{FA}$$

ritardo progressivo con gli FA

τ_{tot} sommatore = ritardo Carry + rit. somma

ritardo dell'ultimo carry + ritardo dell'ultima somma

N=numero di bit del sommatore
τ_C = ritardo per generare il carry sul sommatore
τ_{FA} = ritardo del Full-Adder singolo
τ_{tot} = ritardo totale del sommatore

τ_{tot} dipende da N, gli sommatori, troppo

Sommatore Parallelo con riporto seriale in VERILOG

```
module rc_adder(input wire[N-1:0]a,input wire[N-1:0]b,input wire cin,
                output wire [N-1:0]s,output wire cout);

    parameter N = 32;  // bits
    wire [N-1:0] _c;

    assign cout = _c[N-1];
    full_adder add0(a[0],b[0],cin,  s[0],_c[0]);

    genvar i;
    generate
        for (i = 1; i < N; i = i + 1) begin : gen_adder

            full_adder addN(a[i],b[i],_c[i-1],  s[i],_c[i]);

        end
    endgenerate
endmodule
```

"add0" rappresenta un'istanza di full_adder

Le variabili 'genvar' spariscono dopo l'elaborazione del codice (non diventano circuiti)

I nomi generati avranno il prefisso "gen_adder[i]."

"addN" rappresenta un'istanza di full_adder

Voglio tirare fuori il riporto subito

(V. PHRV1, APPENDICE A.6)

Carry Look-Ahead Adder (Riporto Parallelo)

più veloce

Parto dalle equaz. Booleane

Somma intera

$$s_i = a_i \oplus_{xor} b_i \oplus_{xor} c_{i-1}$$

Riporto il carry out

Xor cambia solo con a_i e $b_i = 1$

Calcolo c_{i-1}

oss

$$c_i = a_i b_i + a_i c_{i-1} + b_i \cdot c_{i-1} = a_i b_i + c_{i-1}(a_i + b_i) = a_i b_i + c_{i-1}(a_i \oplus b_i)$$

$c_{i-1}(a_i + b_i)$

ai FINI del calcolo NON CAMBIA se SOSTITUISCO

ai+bi si può sostituire col suo xor perché nell'espressione quando ai=1 e bi=1 anche ai*bi=1

Chiamo $G_i = a_i b_i$ Generate

$$\left.\begin{aligned} G_i &\triangleq a_i b_i \\ P_i &\triangleq a_i \oplus b_i \end{aligned}\right] \Rightarrow$$

$P_i = a_i \oplus b_i$ Propagate

$$s_i = P_i \oplus c_{i-1}$$

$$s_i = P_i \oplus c_{i-1}$$

$$c_i = G_i + P_i \cdot c_{i-1}$$

$$c_i = G_i + P_i \circ c_{i-1}$$

Gi=generate
Pi=propagate

Cinque sommare

$$c_0 = G_0 + P_0 \cdot c_{in}$$

$$c_1 = G_1 + P_1 c_0 = G_1 + G_0 P_1 + P_1 P_0 c_{in}$$

$$c_2 = G_2 + P_2 c_1 = G_2 + G_1 P_2 + G_0 P_1 P_2 + P_2 P_1 P_0 \cdot c_{in}$$

$$c_3 = G_3 + P_3 \cdot c_2 = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + P_3 P_2 P_1 P_0 c_{in}$$

Posso esplicitare i carry con i carry di ingresso

posso esplicitare qui carry in funzione di G_i e C_i

qui carry è una di uscita, G_i e P_i sono disponibili subito, all'ingr. disponibili all'ingr.

a b	a+b	a ⊕ b
0 0	0	0
0 1	1	1
1 0	1	1
1 1	1	0

osservo.

diretta funzione di a e b

=> per il carry finale ho dall'ultimo rimasto disponibile

differenzi solo per ciò, voglio riusare lo xor che ho già fatto

voglio usare a ⊕ b perché lo ho già nel circuito per la somma

$a[i] + b[i] \neq a \oplus b$

ma dato che ci sono a[i] b[i], e' uno quando a e b sono 1, e se fosse... anche se a ⊕ b è 0, il risultato non cambia

vado a sommarci a · b poi, il risultato non cambia

in questo caso posso pensare

Carry Look-Ahead Adder (2)

G e P sono funzioni di a e b

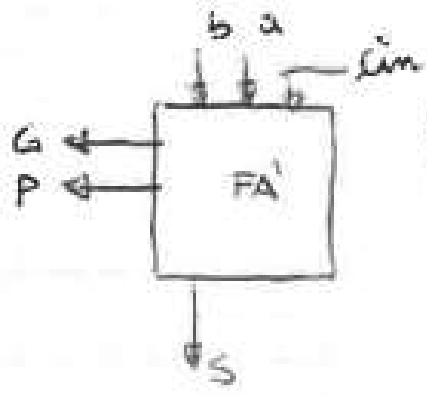
C_{i+1}	a_i	b_i	S_i	C_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

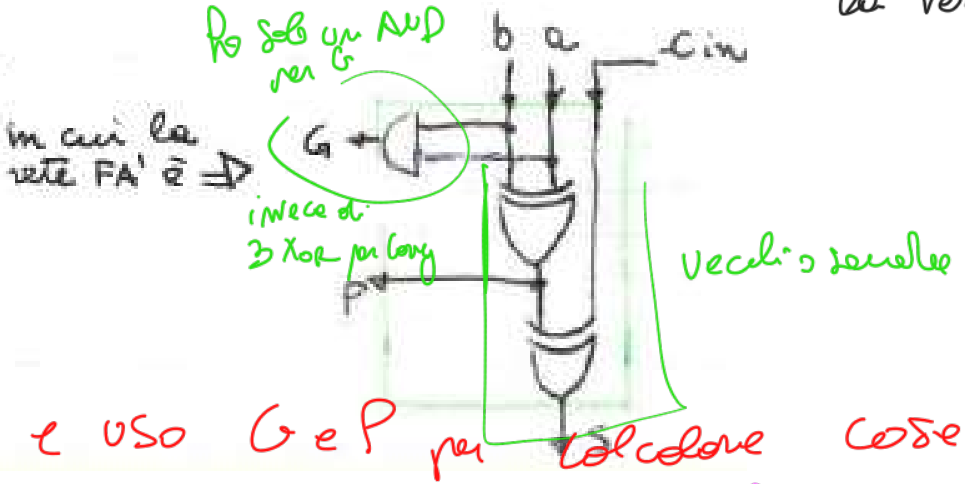
Un altro modo di vedere la funzione di Generate e Propagate:

In questi due casi il carry di uscita è presente perché "si genera" a causa del fatto che nella somma ai e bi sono entrambi "a uno"

In questi altri due casi il carry di uscita è presente perché "si propaga" dal carry precedente (ovvero dal Full Adder precedente)

La rete è semplificata





Prendo G e P di ritardare, e uso G e P per calcolare cose intermedie

Rete semplificata

Ho solo un and per la G e ritardare

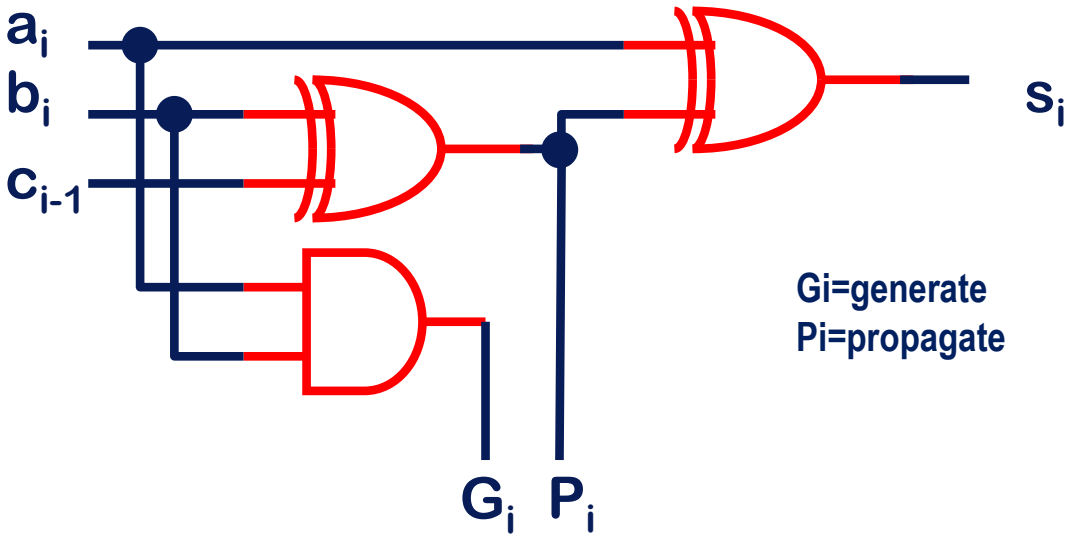
Full Adder per Carry-Look-Ahead in VERILOG

$$G_i = a_i \cdot b_i \qquad P_i = a_i \oplus b_i$$

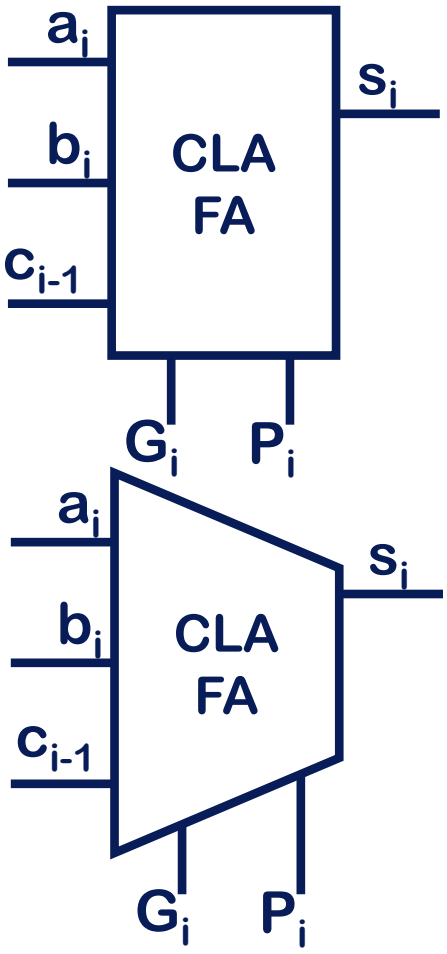
$$s_i = a_i \oplus b_i \oplus c_{i-1} = P_i \oplus c_{i-1}$$

$$\begin{aligned} c_i &= a_i \cdot b_i + a_i \cdot c_{i-1} + b_i \cdot c_{i-1} \\ &= a_i \cdot b_i + c_{i-1} \cdot (a_i + b_i) \\ &= a_i \cdot b_i + c_{i-1} \cdot (a_i \oplus b_i) \\ &= G_i + c_{i-1} \cdot P_i \end{aligned}$$

ai+bi si può sostituire col suo xor perché nell'espressione quando ai=1 e bi=1 anche ai*bi=1



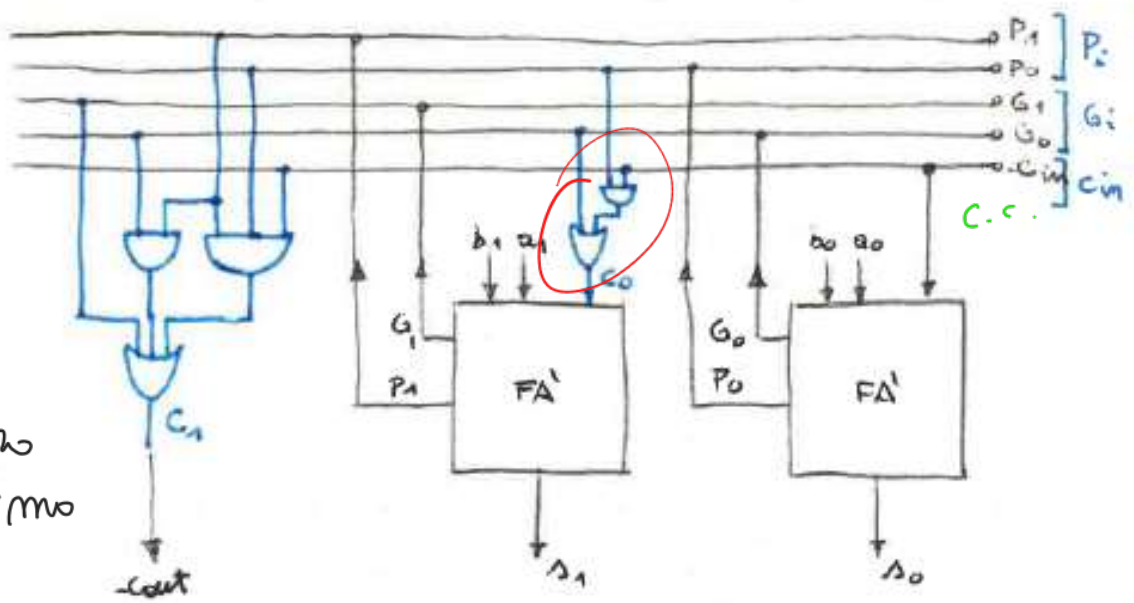
Gi=generate
Pi=propagate




```
module cla_full_adder(input a,input b,input c,output g,output p,output s);
        assign g = a & b;
        assign p = a ^ b;
        assign s = a ^ (b ^ c);
endmodule
```

Carry Look-Ahead Adder (3)

Es. Nel caso pari a 2

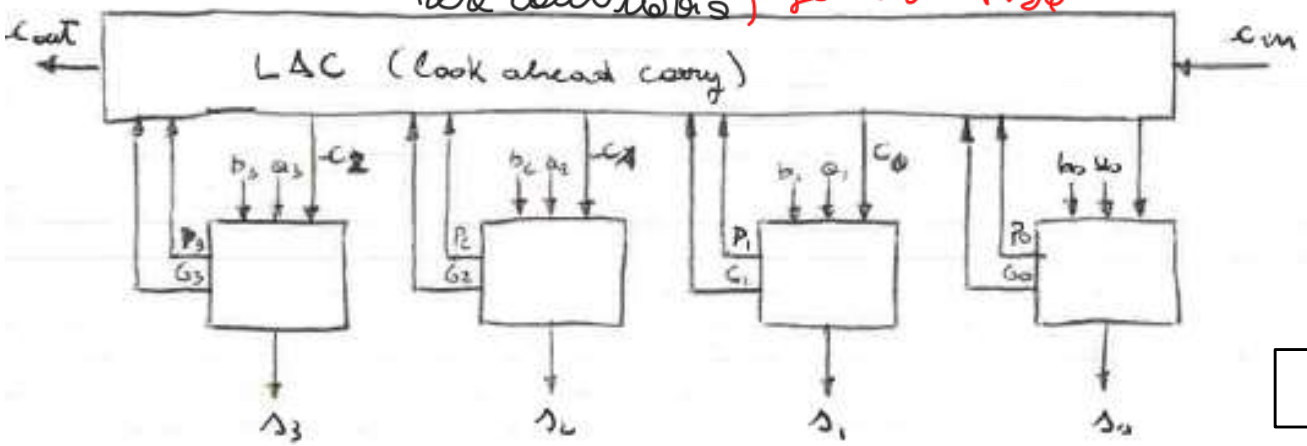


Rete combinatoria in cui faccio solo di prodotti.

Tutti i C_i vengono generati in un tempo fisso, ritardo della LAC

Segue più complesso ritardare l'iesimo carry, ma segue 2 LIVELLI LOGICA

Rete combinatoria, serie di PISO con Somma di prodotti



Ritardo per generare il carry di ogni somma è costante, FISSO ritardo di 2 porte logiche NON DIPENDE DA N! RITARDO FISSO

$$\tau_{cout} = \tau_{LAC}$$

$\tau_{tot} = \tau_{LAC} + \tau_{FA'}$ → ritardo generico sommatore (i 2 XOR)

Ritardo rete carry + ritardo di un sommatore

Ritardo rete del carry

Carry Look-Ahead Adder (4 bit) in VERILOG

```
module cla_adder_4bit(input wire[3:0]a,input wire[3:0]b,input wire cin,
                      output wire[3:0]s,output wire cout);
    wire [4:0] c;
    wire [3:0] g, p;

    assign c[0] = cin;
    assign cout = c[4];

    cla_full_adder add0(a[0],b[0],c[0],g[0],p[0],s[0]);
    assign c[1] = g[0] | (p[0] & c[0]);

    cla_full_adder add1(a[1],b[1],c[1],g[1],p[1],s[1]);
    // assign c[2] = g[1] | (p[1] & c[1]);
    assign c[2] = g[1] | (p[1] & g[0]) | (p[1] & p[0] & c[0]);

    cla_full_adder add2(a[2],b[2],c[2],g[2],p[2],s[2]);
    // assign c[3] = g[2] | (p[2] & c[2]);
    assign c[3] = g[2] | (p[2] & g[1]) | (p[2] & p[1] & g[0])
                | (p[2] & p[1] & p[0] & c[0]);

    cla_full_adder add3(a[3],b[3],c[3],g[3],p[3],s[3]);
    // assign c[4] = g[3] | (p[3] & c[3]);
    assign c[4] = g[3]|(p[3]&g[2])|(p[3]& p[2]&g[1])|(p[3]&p[2]&p[1]&g[0])
                  |(p[3]&p[2]&p[1]&p[0]&c[0]);

endmodule
```

Tre diverse versioni

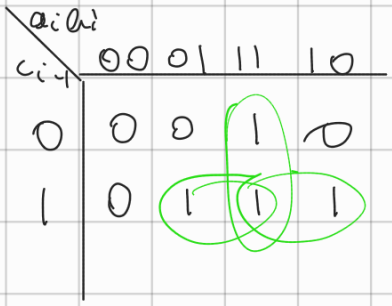
$C[2] = g_1 | (p_1 \& c_1)$

$= g_1 + p_1 (g_0 + p_0 c_0)$

$$S_i = a_i \oplus b_i \oplus c_{i-1}$$

$$C_{i+1} = a_i b_i + c_{i-1}(a_i + b_i)$$

a_i	b_i	c_{i-1}	S_i	C_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



$$C_i = a_i b_i + C_i b_i + C_i a_i$$
$$a_i b_i + C_i(a_i + b_i)$$

a_i	b_i	$a_i + b_i$	$a_i \oplus b_i$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0

$$S_i = a_i \oplus b_i \oplus C_{i-1} \Rightarrow S_i = P_i \oplus C_{i-1}$$
$$C_i = \underbrace{a_i b_i}_{g} + C_{i-1}\underbrace{(a_i \oplus b_i)}_{P} \qquad C_i = g_i + C_{i-1} P_i$$

$$C_1 = g_0 + C_{in} P_0$$
$$C_2 = g_1 + C_1 P_1 = g_1 + (g_0 + C_{in} P_0) P_1$$
$$= g_1 + P_1 g_0 + P_1 P_0 C_{in}$$