LA BIBBIA DI VERILOG DA EDOARDO

Introduzione

--> Un blocco hardware con input e output si chiama modulo, ad esempio AND gate, mux, sono tutti moduli. Ci sono 2 tipi di moduli

- Comportamentale: descrive cosa fa un modulo
- Strutturale: descrive com'è costruito un modulo, partendo da moduli più semplici
  -> Esempio SystemVerilog comportamentale:

```
module nomemodulo(input logic a, b, c,
                  output logic y);
        assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;
endmodule
```

module/endmodule servono per iniziare/finire il modulo
nomemodulo è il nome del modulo
input logic sono i nostri input mentre output logic sono gli output
assign y assegna all'output l'espressione booleana
operatori:
~ not
| or
& and
-> Altro esempio:

```
module nuovomodulo(input logic a, b, c
                                output logic y);
        assign y = ~b & ~c | a & ~b
endmodule
```

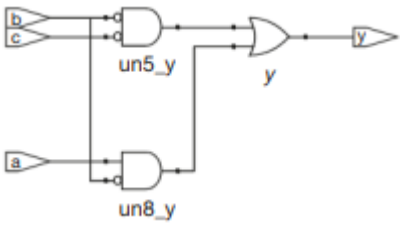
--> I due obiettivi principali degli HDL sono la simulazione e la sintesi.

- Nella simulazione si forniscono valori di ingresso al modulo e si controlla alle uscite se il modulo funziona correttamente.
- Nella sintesi, la descrizione testuale del modulo viene tradotta in rete di porte logiche.
  -> Esempio sintesi:
  Modulo originale:

```
module funzionequalunque (input logic a, b, c,
                                                output logic y);
```

```
        assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;
endmodule
```

Circuito dopo il processo di sintesi:



--> Idiomi: vari classi di componenti logiche

Logica combinatoria

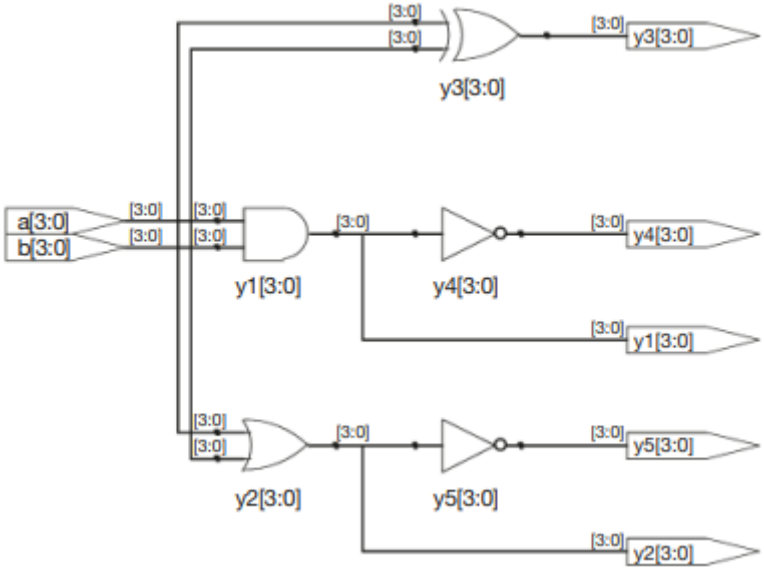
--> Gli operatori a singolo bit (bitwise) agiscono su segnali costituiti da bit singoli o su bus multibit.
Esempio descrizione quattro negatori (not) collegati a bus a 4 bit:

```
module neg (input logic [3:0] a,
                    output logic [3:0] y);
        assign y = ~a
endmodule
```

in questo caso [3:0] rappresenta un bus a 4 bit, dal più significativo al meno significativo, poteva anche essere [4:1] o [0:3]. L'output che produce è il complemento di ognuno dei bit del bus.
-> Altro esempio:

```
module porte(input logic [3:0] a, b,
                        output logic [3:0] y1, y2, y3, y4, y5); /* cinque
diverse porte logiche a 2 ingressi collegate a bus a 4 bit */
        assign y1 = a & b; // AND
        assign y2 = a | b; // OR
        assign y3 = a ^ b; // XOR
        assign y4 = ~(a & b); // NAND
        assign y5 = ~(a | b); // NOR
endmodule
```

qua mandiamo il bus con 4 bit e ogni output produce un risultato a 4 bit, manipolandoli in

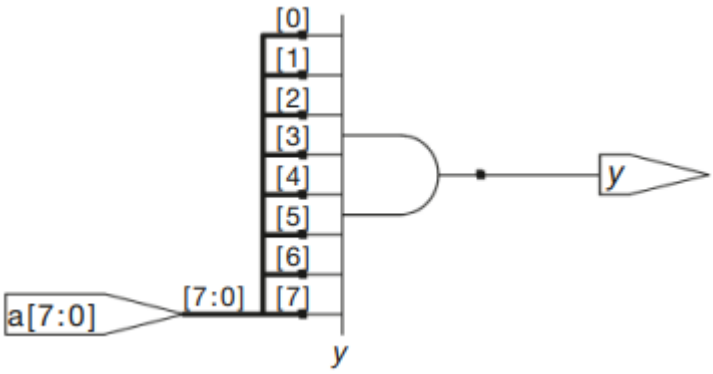


modo diverso. Sintesi del circuito:

--> Gli operatori di riduzione sono costituiti da porte logiche a tanti ingressi che producono un'unica uscita.
Esempio porta AND a 8 ingressi:

```
module and8(input logic [7:0] a,
                        output logic y); /* cinque diverse porte logiche a 2
ingressi collegate a bus a 4 bit */
        assign y = ~a
        // assign y = &a è molto più facile da scrivere di
        // assign y = a[7] & a[6] & a[5] & a[4] & // a[3] & a[2]
        //                       &a[1] & a[0];
endmodule
```

sarebbe un AND tra a7 a6 a5..... Sintesi del circuito:

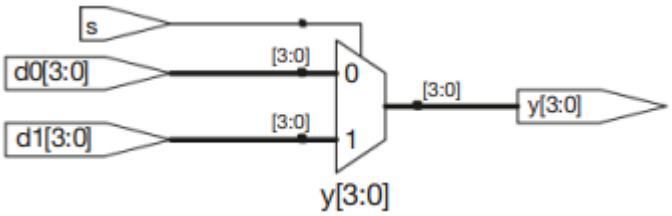


--> Un assegnamento condizionale seleziona l'uscita da generare tra varie alternative sulla base di un ingresso chiamato condizione.
Esempio Mux 2:1 a 4 bit di ingresso che fa uso di assegnamento condizionale:

```
module mux2(input logic [3:0] d0, d1, // entrambi bus a 4 bit
                        input logic s, // singolo bit, sarà 1 o 0
                        output logic [3:0] y); // output a 4 bit
```

```
        assign y = s ? d1 : d0;
endmodule
```

L'operatore condizionale `?` seleziona, sulla base della prima espressione, la seconda o la terza espressione. La prima espressione è denominata condizione, se s vale 1 sceglie d1, se s vale 0 sceglie d0.
? E' molto utile per descrivere un multiplexer, che sulla base del valore del primo ingresso ne seleziona uno degli altri due. Circuito:



-> Altro esempio Mux 4:1

```
module mux4(input logic [3:0] d0, d1, d2, d3
                        input logic [1:0] s,
                        output logic [3:0] y);
        assign y = s[1] ? (s[0] ? d3 : d2;)
                                    : (s[0] ? d1 : d0)
endmodule
```

in questo caso la nostra condizione s è formata da 2 bit, quindi 4 combinazioni. In base al valore di s[1], cioé il primo bit dei due, il programma sceglie la combinazione s[0] da prendere. Esempio
s[1] può valere 1 o 0, se vale 1 il programma prende 10 o 11, quindi quando s[0] vale 1 o 0, in base poi a questa scelta vengono presi i valori d3/d2/d1/d0

--> Le variabili interne non sono ne ingressi ne uscite ma sono utilizzate solo all'interno del modulo, come le variabili locali. Esempio:

```
module sommatore(input logic a, b, rin
                        output logic s, rout);

                logic p, g;

        assign p =a ^ b;
        assign g =a & b;
        assign s = p ^ rin;
        assign rout = g | (p & rin);
endmodule
```

p e g non sono valori accessibili al di fuori del modulo sommatore, però vengono usati per dei calcoli all'interno del modulo.

--> Le parentesi sono importanti o utilizzerà un sistema di precedenza delle operazioni tutto suo
--> I numeri lunghi possono essere spezzati così 111_232342_22313 lui li leggerà comunque senza trattini bassi
--> Nel linguaggio Verilog, `4'bz` è una notazione per rappresentare un bus di 4 bit con tutti i bit impostati a "Z" (alta impedenza o valore indefinito).

--> Concatenazione di bit. Esempio dove viene assegnato al bus y il valore a 9 bit c2c1d0d0d0c0101:

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

Notazioni importanti: per dire 3 volte d0 scrivo {3{d[0]}.
per dire 3 bit impostati a 101 scrivo 3'b101

--> Porte logiche con ritardi: è possibile assegnare ritardi alle istruzioni, utili per capire quanto velocemente funzionerà un circuito o per il debugging, nella fase di sintesi i ritardi non contano:

```
'timescale 1ns/1ps
module esempio(input logic a, b, c,
                                output logic y);
        logic ab, bb, cb, n1, n2, n3;

        assign #1 {ab, bb, cb} = ~{a, b, c};
        assign #2 n1 = ab & bb & cb;
        assign #2 n2 = a & bb & cb;
        assign #2 n3 = a & bb & c;
        assign #4 y = n1 | n2 | n3;
endmodule
```

'timescale 1ns/1ps indica il ritardo di 1ns con 1ps di precisione, mentre il ritardo vero e proprio si indica con assign # ..., esempio: assign #2 n1 = ab & bb & cb;
Notazioni importanti: assign {ab, bb, cb} = ~{a, b, c}; significa che ab = ~a, bb = ~b, cb = ~c

Modellazione strutturale

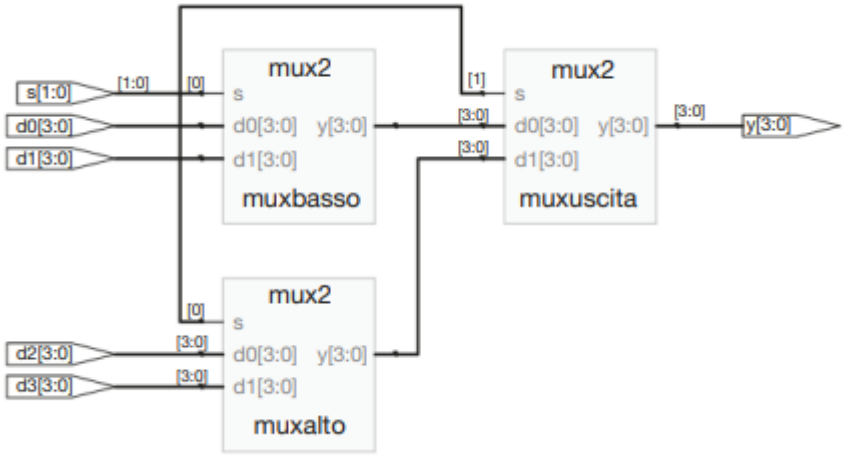
--> Mux 4:1 usando 2 mux 2:1

```
module mux2(input logic [3:0] d0, d1, // entrambi bus a 4 bit
                        input logic s, // singolo bit, sarà 1 o 0
                        output logic [3:0] y); // output a 4 bit
        assign y = s ? d1 : d0;
endmodule

module mux4(input logic [3:0] d0, d1, d2, d3,
                    input logic [1:0] s,
```

```
                    output logic[3:0] y);
        logic [3:0] basso, alto;
        mux2 muxbasso(d0, d1, s[0], basso)
        mux2 muxalto(d2, d3, s[0], alto)
        mux2 muxuscita(basso, alto, s[1], y)
endmodule
```

quando scrivo mux2 sto utilizzando il modulo prima con gli argomenti (d0, d1, s[0], basso). basso e alto sono le uscite di muxbasso e muxalto



Logica sequenziale

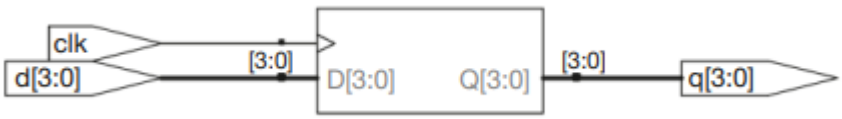
--> Flip Flop D:

```
module flop(input logic clk,
                        input logic [3:0] d,
                        output logic [3:0] q);
        always_ff @(posedge clk)
                q <= d;
endmodule
```

always_ff indica il flip flop, posedge clk sarebbe che l'istruzione q <= d si attiva solo quando il fronte d'onda del clock è positivo. Al momento consideriamo <= come =
In generale, un'istruzione always in SystemVerilog ha la forma:

```
always @(sensitivity list)
        istruzione
```

inoltre abbiamo varie istruzioni always a seconda di cosa usiamo: always_ff, always_latch e always_comb
-> Sintesi del circuito di prima:



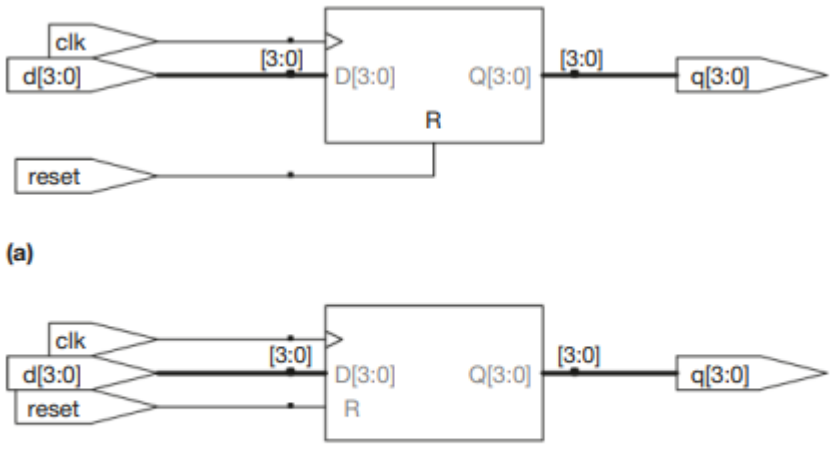
-> Le istruzioni always possono essere usate per scopi di memorizzazione quindi per reti sequenziali, mentre istruzioni come assign sono valutate ogni volta da capo

--> Registro resettabile in modo asincrono/sincrono

```
module flopr(input logic clk,
                         input logic reset,
                         input logic [3:0] d,
                         output logic [3:0] q);
                         // reset asincrono
                 always_ff @(posedge clk or posedge reset)
                         if (reset) q <= 4'b0;
                         else q <= d;
endmodule

module flopr(input logic clk,
                         input logic reset,
                         input logic [3:0] d,
                         output logic [3:0] q);
                         // reset sincrono
                 always_ff @(posedge clk)
                         if (reset) q <= 4'b0;
                         else q <= d;
endmodule
```

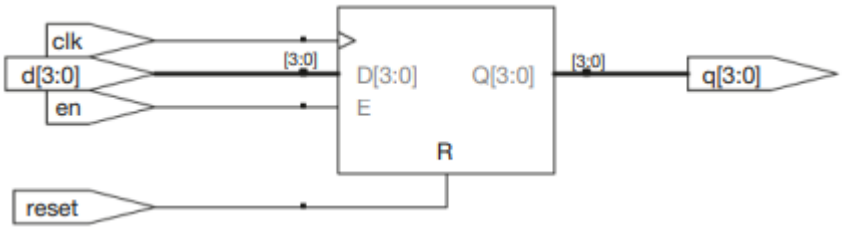
Asincrono: quando il segnale di reset è attivo (alto), il flip-flop imposta il suo stato q a zero, indipendentemente dal fronte di salita successivo del segnale di clock.
Sincrono: quando il segnale di reset è attivo (alto) al fronte di salita successivo del segnale di clock, il flip-flop imposta il suo stato q a zero
-> Più segnali nella sensitivity list sono separati da una virgola o dalla parola or.



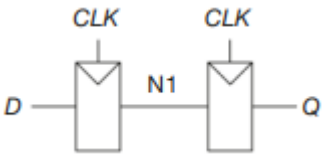
ra 4.15 Sintesi del circuito `flopr` **(a) con reset asincrono, (b) con reset sincrono.**

--> I registri con l'abilitazione (enable) reagiscono al clock solo se il segnale enable è attivo.
Esempio:

```
module flopenr(input logic clk,
                           input logic reset,
                           input logic en,
                           input logic [3:0] d,
                           output logic [3:0] q);
                           // reset asincrono
                always_ff @(posedge clk, posedge reset)
                 if (reset) q <= 4'b0;
                 else if (en) q <= d;
endmodule
```



--> Registri multipli: sincronizzatore:




```
module sinc(input logic clk,
                        input logic d,
                         output logic q);
        logic n1;
        always_ff @(posedge clk)
         begin
                n1 <= d; // non bloccante
                q <= n1; // non bloccante
         end
endmodule
```

Il costrutto begin/end è necessario perché ci sono più istru□zioni all'interno dell'istruzione always. begin/ end non era necessario nell'esempio flopr perché il costrutto if/else corrisponde a una singola istruzione.

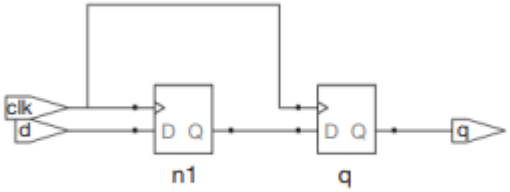


Figura 4.18 Sintesi del circuito `sinc`.

--> Latch D: è trasparente quando il clock è ALTO, permettendo ai dati di passare dall'ingresso all'uscita. Il latch diventa "opaco" quando il clock è BASSO, e mantiene lo stato

raggiunto in precedenza. Esempio:

```
module latch(input logic clk,
                        input logic [3:0] d,
                         output logic [3:0] q);
        always_latch
                if (clk) q <= d;
endmodule
```

- always*latch equivale ad always*@(clk,d) ed è l'idioma da usare di preferenza in SystemVerilog per descrivere un latch.
- Viene valutato ogni volta che clk o d cambiano. Se clk è ALTO, d attraversa i latch e arriva a q, quindi questo codice descrive un latch sensibile al livello positivo. Altrimenti q mantiene il valore precedente

--> Negatore di always:

```
module neg(input logic [3:0] a, output logic [3:0] y); always_comb y = ~a;
endmodule
```

- always*comb rivaluta le istruzioni all'interno dell'istruzione always ogni volta che i segnali a destra di <= oppure = nell'istruzione always cambiano. In questo caso, equivale a always*@(a)
- Il simbolo = nell'istruzione always è chiamato assegnamento bloccante, rispetto all'assegnamento non bloccante <=. In SystemVerilog è buona norma usare assegnamenti bloccanti per la logica combinatoria e non bloccanti per quella sequenziale.

--> Sommatore completo facendo uso di always:

```
module sommatore(input logic a, b, rin,
                                output logic s, rout);
        logic p, g;

        always_comb
                begin
                        p = a ^ b; // bloccante
                        g = a & b; // bloccante
                        s = p ^ rin; // bloccante
                        rout = g | (p & rin); // bloccante
                end
endmodule
```

--> Istruzione case:

```
module sette_segmenti(input logic [3:0] dati,
                                        output logic [6:0] segmenti);
        always_comb
                case(dati) // abc_defg
                        0: segmenti = 7'b111_1110;
                        1: segmenti = 7'b011_0000;
                        2: segmenti = 7'b110_1101;
                        3: segmenti = 7'b111_1001;
                        4: segmenti = 7'b011_0011;
                        5: segmenti = 7'b101_1011;
                        6: segmenti = 7'b101_1111;
                        7: segmenti = 7'b111_0000;
                        8: segmenti = 7'b111_1111;
                        9: segmenti = 7'b111_0011;
                        default: segmenti = 7'b000_0000;
                endcase
endmodule
```

- a seconda del valore di (dati) esegue l'azione dopo i :, ad esempio il caso dati = 0 impostan segmenti = 7'b111_1110; se dati = 5 imposta segmenti = 7'b101_1011;
- La clausola default è un modo conveniente per definire le uscite in tutti i casi non elencati esplicitamente, garantendo in questo modo una logica combinatoria.

--> Decoder 3:8

```
module decoder3_8(input logic [2:0] a,
                                output logic [7:0] y);
        always_comb
                case(a)
                        3'b000: y = 8'b00000001;
                        3'b001: y = 8'b00000010;
                        3'b010: y = 8'b00000100;
                        3'b011: y = 8'b00001000;
                        3'b100: y = 8'b00010000;
                        3'b101: y = 8'b00100000;
                        3'b110: y = 8'b01000000;
                        3'b111: y = 8'b10000000;
                        default: y = 8'bxxxxxxxx;
                endcase
endmodule
```

- in questo caso se a vale 3 bit impostati a 000 allora y vale 8 bit impostati a 00000001 e così via per tutti i casi
  -> *Posso usare casez per impostare don't care con il ?*

```
module circprio_indiff(input logic [3:0] a,
                                          output logic [3:0] y);
        always_comb
                casez(a)
                        4'b1???: y = 4'b1000;
                        4'b01??: y = 4'b0100;
                        4'b001?: y = 4'b0010;
                        4'b0001: y = 4'b0001;
                        default: y = 4'b0000;
                endcase
endmodule
```

--> Linee guide per assegnamenti bloccanti e non bloccanti:

1. Usare `always_ff @(posedge clk)` e assegnamenti non bloccanti per modellizzare logica sequenziale sincrona.

```
always_ff @(posedge clk)
  begin
    n1 <= d; // non bloccante
    q <= n1; // non bloccante
  end
```

2. Usare assegnamenti continui per modellizzare logica combinatoria semplice.

```
assign y = s ? d1 : d0;
```

3. Usare `always_comb` e assegnamenti bloccanti per modellizzare logica combinatoria più complessa, dove l'istruzione `always` è molto utile.

```
always_comb
  begin
    p = a ^ b; // bloccante
    g = a & b; // bloccante
    s = p ^ rin;
    rout = g | (p & rin);
  end
```

4. Non fare più assegnamenti allo stesso segnale in più di un'istruzione `always` o più di un'istruzione di assegnamento continuo.

--> Sommatore completo con uso di assegnamenti non bloccanti

```
// assegnamenti non bloccanti (sconsigliati)
module sommatore(input logic a, b, rin,
                                 output logic s, rout);
                logic p, g;

                always_comb
                        begin
                                p <= a ^ b; // non bloccante
                                g <= a & b; // non bloccante
                                s <= p ^ rin;
```

```
                        rout <= g | (p & rin);
                end
endmodule
```

--> Contatore modulo 11 significa che conterà fino a 10, quindi mi servono 4 bit per rappresentarlo esempio di implementazione:

```
module contatore(input logic [3:0] count,
                        input logic clk,
                        input logic reset,
                        output logic next_count);
        always_ff @(posedge clk or posedge reset) // asincrono
                if (reset)
                        count <= 4'b0000;
                else if (count == 4'b1010)
                        count <= 4'b0000;
                else
                        next_count <= count + 1
endmodule
```

Macchine a stati finiti

--> Implementazione macchina a stati finiti "riconosce sequenza", è previsto un segnale di reset asincrono: MOORE

```
module sequenzeMoore(input logic clk,
                                input logic reset,
                                input logic a,
                                output logic y);
        typedef enum logic [1:0] {S0, S1, S2} tipostato;
        tipostato stato, statopross;

        // registro di stato
        always_ff @(posedge clk, posedge reset)
                if (reset) stato <= S0;
                else stato <= statopross;

                // logica di stato prossimo
        always_comb
                 case (state)
                        S0: if (a) statopross = S0;
                                else statopross = S1;
                        S1: if (a) statopross = S2;
                                else statopross = S1;
                        S2: if (a) statopross = S0;
                                else statopross = S1;
                        default: statopross = S0;
```

```
                endcase

                // logica di uscita
        assign y = (stato==S2);
endmodule
```

- L'istruzione typedef definisce tipostato come valore di tipo logic a due bit con tre possibilità: S0, S1 o S2. stato e statopross sono segnali di tipo tipostato. Quindi stato è di tipo tipostato che è di tipo logic enum, è formato da 2 bit e può valere S0 o S1 o S2
- Le codifiche enumerative (typeddef enum) per default seguono l'ordine numerico, quindi S0 = 00, S1 = 01 e S2 = 10
- Dal momento che la logica di stato prossimo deve essere combinatoria, è necessaria la clausola default.
- Manda 1 come output quando riconosce la sequenza S2
- **MOORE: uscita dipende solo dallo stato presente. MEALY: uscita dipende da ingresso e stato presente**

--> Esempio MEALY:

```
module sequenzeMoore(input logic clk,
                                        input logic reset,
                                        input logic a,
                                        output logic y);
        typedef enum logic [1:0] {S0, S1, S2} tipostato;
        tipostato stato, statopross;

        // registro di stato
        always_ff @(posedge clk, posedge reset)
                if (reset) stato <= S0;
                else stato <= statopross;

                // logica di stato prossimo
        always_comb
                 case (state)
                        S0: if (a) statopross = S0;
                                else statopross = S1;
                        S1: if (a) statopross = S2;
                                else statopross = S1;
                        S2: if (a) statopross = S0;
                                else statopross = S1;
                       default: statopross = S0;
                endcase

                // logica di uscita
        assign y = (a & stato==S1);
endmodule
```

- *Uguale a Moore con uscita diversa*
- *S1 input 1 va in S2, come Moore*