



Descrizione di Macchine a Stati finiti in VHDL

Descrizioni di Macchine Sequenziali in VHDL

- In questo capitolo vedremo come un sistema digitale sequenziale può essere descritto in VHDL.
- **Outline:**
 - Macchine sequenziali: rappresentazioni
 - Macchine di Mealy e di Moore
 - Macchine a Stati Finiti in VHDL
 - Assegnazioni con Guardia
 - Funzioni di risoluzione

Macchine Sequenziali

- Una **macchina sequenziale** è un sistema a tempo discreto in cui il valore dell'uscita dipende dai valori assunti dall'ingresso in tutti gli istanti di tempo (non da quelli successivi all'istante corrente se il sistema è fisicamente realizzabile);
- Il modello matematico di una macchina sequenziale è il seguente:

$M = \{I, U, S, F, G\}$ dove:

$I: \{i_1, i_2, \dots, i_n, \dots\}$ è l'*alfabeto di ingresso*,

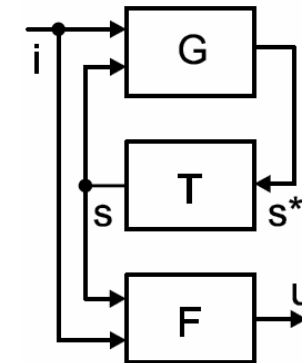
$U: \{u_1, u_2, \dots, u_n, \dots\}$ è l'*alfabeto di uscita*,

$S: \{s_1, s_2, \dots, s_n, \dots\}$ è l'*insieme degli stati*,

$F: S \times I \rightarrow U$ è la *funzione di uscita* e

$G: S \times I \rightarrow S$ è la *funzione di aggiornamento* dello stato interno;

- un opportuno blocco di ritardo (T) mantiene il “vecchio stato” s fino a quando non è necessario sostituirlo con il “nuovo stato” s^*

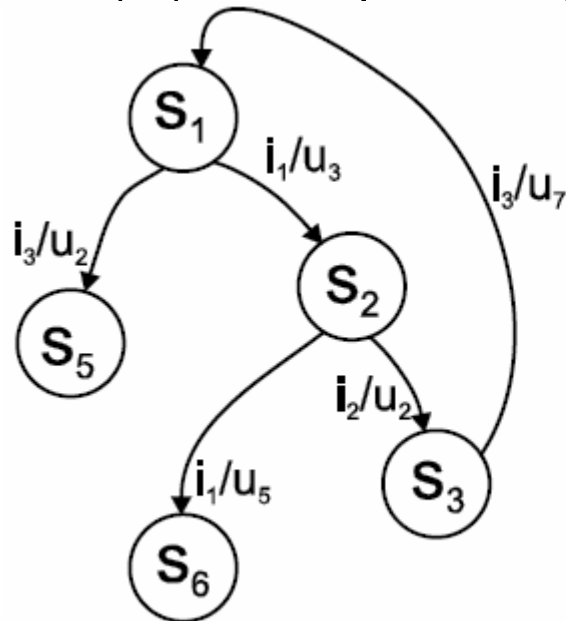


Macchine Sequenziali: FSM

- Nei calcolatori elettronici, ingresso uscita e stato sono codificati attraverso una rappresentazione digitale e possono assumere un numero finito di valori;
- ci interessiamo quindi di *FSM (Finite State Machine)*;
- esistono modelli matematici di macchine composte da una FSM che fa da unità di controllo e da una “memoria” virtualmente infinita. La macchina di Turing ne è un esempio.

Macchine Sequenziali: rappresentazioni

- Le funzioni di uscita (F) e di aggiornamento dello stato interno (G) sono spesso rappresentate mediante grafi oppure tabelle:



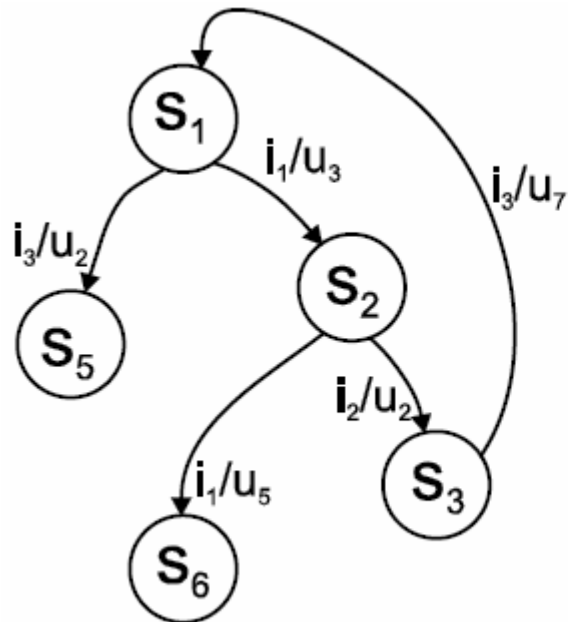
S	i	i		
		i ₁	i ₂	i ₃
S ₁
S ₂
S ₃	S ₅	U ₉
S ₄
S ₅
S ₆

Gli archi rappresentano le transizioni da uno stato all'altro in corrispondenza di uno specifico ingresso, con la conseguente uscita.

Tutte le possibili transizioni sono tabellate.

Macchine Sequenziali di Mealy

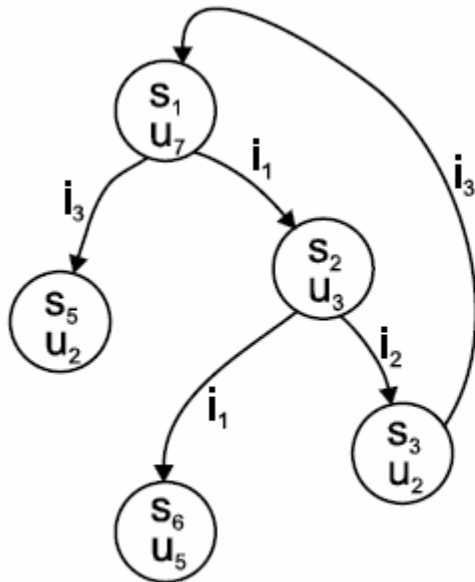
- L'uscita è una funzione tanto dello stato corrente quanto dell'ingresso:
$$u = F(s, i)$$
- Il grafo e la tabella precedentemente mostrati hanno una forma adatta a descrivere il comportamento di una macchina di Mealy



S	i	i ₁	i ₂	i ₃
S ₁	
S ₂	
S ₃		...	S ₅	U ₉
S ₄	
S ₅	
S ₆	

Macchine Sequenziali di Moore

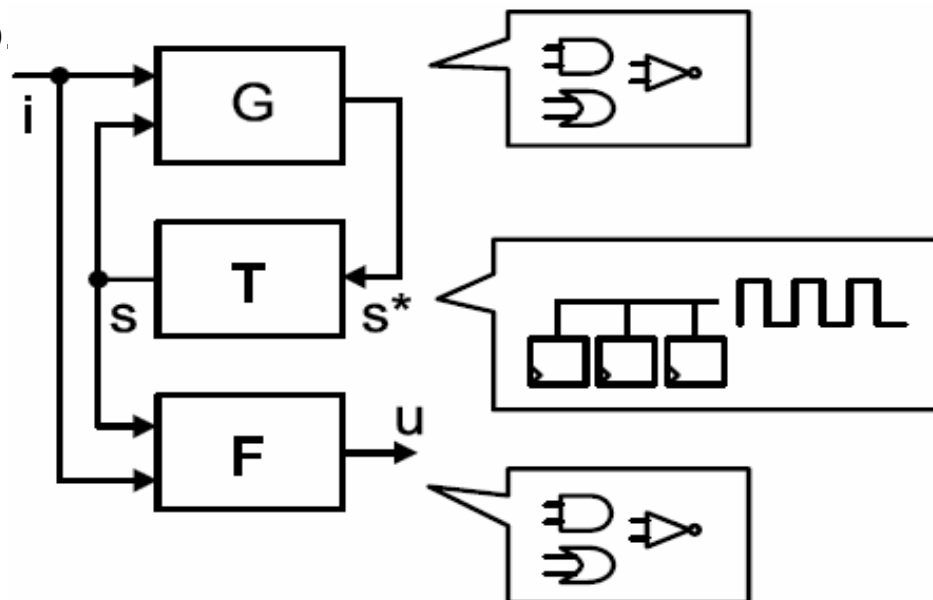
- L'uscita è una funzione solo dello stato corrente, mentre non c'è dipendenza dell'uscita rispetto all'ingresso corrente: $u = F(s)$
- Il grafo e la tabella precedentemente mostrati possono essere semplificati per descrivere il comportamento di una macchina di Moore:



S	i	i₁	i₂	i₃	u
S₁	
S₂	
S₃		...	S₅	...	u₉
S₄	
S₅	
S₆	

Macchine Sequenziali

- Nella loro implementazione elettronica, le macchine sequenziali richiedono degli elementi di memoria.
- Precisamente, le funzioni F e G sono realizzate mediante blocchi combinatori, mentre l'elemento di ritardo è costituito da flip-flop.
- Nelle macchine sequenziali sincrone un unico segnale (il segnale di clock) controlla gli elementi di memoria determinando l'istante di cambio dello stato.



Progetto di Macchine Sequenziali

1. Costruzione della tabella di stato/uscita a partire dalla descrizione verbale delle specifiche. I valori di ingresso, stato ed uscita possono avere nomi mnemonici qualsiasi.
2. Minimizzazione degli stati.
3. Codifica binaria dei valori di ingresso, stato ed uscita.
4. Scelta del tipo di flip-flop con cui realizzare l'elemento di ritardo.
5. Costruzione delle "equazioni di eccitazione" dei flip-flop corrispondenti alla tabella di transizione di stato.
6. Costruzione delle reti logiche che implementino le equazioni di eccitazione e di uscita.
7. Ottimizzazione delle reti logiche.
 - Nella pratica, soltanto il primo passo è gestito dal progettista, mentre i successivi possono essere demandati agli appositi tool di sviluppo.

FSM in VHDL

- Il VHDL consente di descrivere macchine a stati finiti;
- gli stati della macchina possono essere elencati definendo un tipo enumerato e quindi indicando con un nome mnemonico ogni valore dello stato.

- Un esempio:

```
--definizione dello stato
```

```
TYPE state IS (idle, read, post_read);
```

```
SIGNAL current_state, next_state : state;
```

- La descrizione della FSM può essere effettuata con diverse tecniche.
- In particolare è possibile destinare due *process* distinti alla descrizione della parte combinatoria e quella con memoria della FSM.
- In alternativa, è possibile descrivere in un unico *process* il comportamento complessivo della macchina.

Descrizione della FSM con due process

- Un *process* è utilizzato per descrivere la parte combinatoria della FSM, ovvero la funzione di aggiornamento dello stato G e la funzione di uscita F . Pertanto, nella *sensitivity list* del *process* saranno certamente contenuti i segnali relativi agli ingressi della FSM ed allo stato corrente, mentre tra i segnali aggiornati ci sarà certamente quello che rappresenta lo stato prossimo.
- Un costrutto CASE...WHEN sul segnale che rappresenta lo stato corrente consentirà di decidere, anche in funzione dell'ingresso, qual è l'uscita attuale e quale sarà il prossimo stato.
- L'altro *process* descrive l'elemento di ritardo della FSM. Dunque, avrà nella *sensitivity list* il segnale di clock, mentre aggiornerà il segnale che rappresenta lo stato corrente.

Descrizione della FSM con due process

```
TYPE state IS (S1, S2,    );
SIGNAL current_state, next_state : state;
BEGIN
  Reg: PROCESS (clock,    )
    BEGIN
      IF rising_edge(clock) THEN
        current_state <=next_state; END IF;
    END PROCESS reg;

  Comb: PROCESS (i1, i2,    , current_state)
    BEGIN
      CASE current_state IS
        WHEN S1 => IF (i1=1) THEN u1<=1;
                   next_state <=s3;
        WHEN S2 =>
      END CASE;
    END PROCESS Comb;
```

Descrizione della FSM con un process

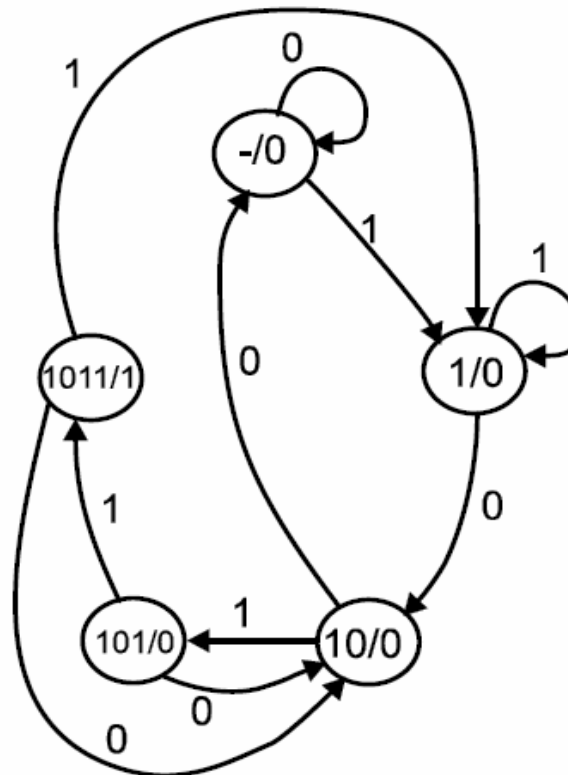
- E' comunque possibile descrivere l'intera FSM entro un solo processo, peraltro utilizzando un unico segnale di tipo *stato*. Ne risulta una descrizione più compatta
- L'uso del costrutto CASE...WHEN è sostanzialmente lo stesso rispetto al caso precedente.

Descrizione della FSM con un process

```
TYPE state IS (S1,S2,   );  
SIGNAL current_state: state;  
BEGIN  
  
fsm: PROCESS (clock,   )  
BEGIN  
    IF rising_edge(clock) THEN  
        CASE current_state IS  
            WHEN S1 => IF (i1=1) THEN u1<=1;  
                        current_state <=s3;  
            WHEN S2 =>  
        END CASE;  
    END IF;  
END PROCESS fsm;
```

Esempio: riconoscitore di stringa (Moore)

- Il grafo in basso rappresenta una macchina di Moore che riconosce la stringa di ingresso 1011.
- La descrizione VHDL di tale macchina discende immediatamente dal grafo rappresentato.





Riconoscitore di stringa: due processi

```
ENTITY moore_detector IS
PORT (
    x, clk : IN BIT;
    z : OUT BIT);
END moore_detector;
--
ARCHITECTURE two_processes_state_machine OF moore_detector IS
TYPE state IS (reset, got1, got10, got101, got1011);
SIGNAL next, current : state := reset;
BEGIN
PROCESS (clk)
    BEGIN
    IF rising_edge(clk) THEN
        current<= next; END IF;
    END PROCESS;
```

Riconoscitore di stringa: due processi

```
PROCESS(current, x)
BEGIN
Z<='0';    --assegnazione di default
CASE current IS
WHEN reset =>
    IF x = '1' THEN next <= got1; ELSE next <= reset;
    END IF;
WHEN got1 =>
    IF x = '0' THEN next <= got10; ELSE next <= got1;
    END IF;
WHEN got10 =>
    IF x = '1' THEN next <= got101; ELSE next <= reset;
    END IF;
WHEN got101 =>
    IF x = '1' THEN next <= got1011; ELSE next <= got10;
    END IF;
WHEN got1011 =>
    z <= '1';
    IF x = '1' THEN next <= got1; ELSE next <= got10;
    END IF;
END CASE;

END PROCESS;
END two_processes_state_machine;
```



Riconoscitore di stringa: un processo

```
ENTITY moore_detector IS
PORT (
    x, clk : IN BIT;
    z : OUT BIT);
END moore_detector;
--
ARCHITECTURE one_processes_state_machine OF moore_detector IS
TYPE state IS (reset, got1, got10, got101, got1011);
SIGNAL next, current : state := reset;
BEGIN
```

Riconoscitore di stringa: un processo

```
PROCESS (clk)
BEGIN
  Z <= '0';    --assegnazione di default
  IF rising_edge(clk) THEN
    CASE current IS
      WHEN reset =>
        IF x = '1' THEN current <= got1; ELSE current <= reset;
        END IF;
      WHEN got1 =>
        IF x = '0' THEN current <= got10; ELSE current <= got1;
        END IF;
      WHEN got10 =>
        IF x = '1' THEN current <= got101; ELSE current <= reset;
        END IF;
      WHEN got101 =>
        IF x = '1' THEN current <= got1011; ELSE current <= got10;
        END IF;
      WHEN got1011 =>
        z <= '1';
        IF x = '1' THEN current <= got1; ELSE current <= got10;
        END IF;
    END CASE;
  END IF;
END PROCESS;
END two_processes_state_machine;
```

Descrizione comportamentale di una FSM

- Per descrivere una FSM elaborata in VHDL può spesso essere comodo ricorrere ad una descrizione comportamentale della macchina.
- Particolarmente utile a tal fine è lo statement WAIT, che permette di introdurre gli elementi sequenziali del circuito descritto.
- Il riconoscitore di stringa introdotto in precedenza può essere descritto ricorrendo allo statement wait, come segue:

```
ENTITY moore_detector IS
```

```
PORT (
```

```
    x, clk : IN BIT;
```

```
    z : OUT BIT);
```

```
END moore_detector;
```

```
ARCHITECTURE behavioral_state_machine OF moore_detector IS
```

```
TYPE state IS (reset, got1, got10, got101, got1011);
```

```
SIGNAL current : state := reset;
```

```
BEGIN
```

Descrizione comportamentale di una FSM

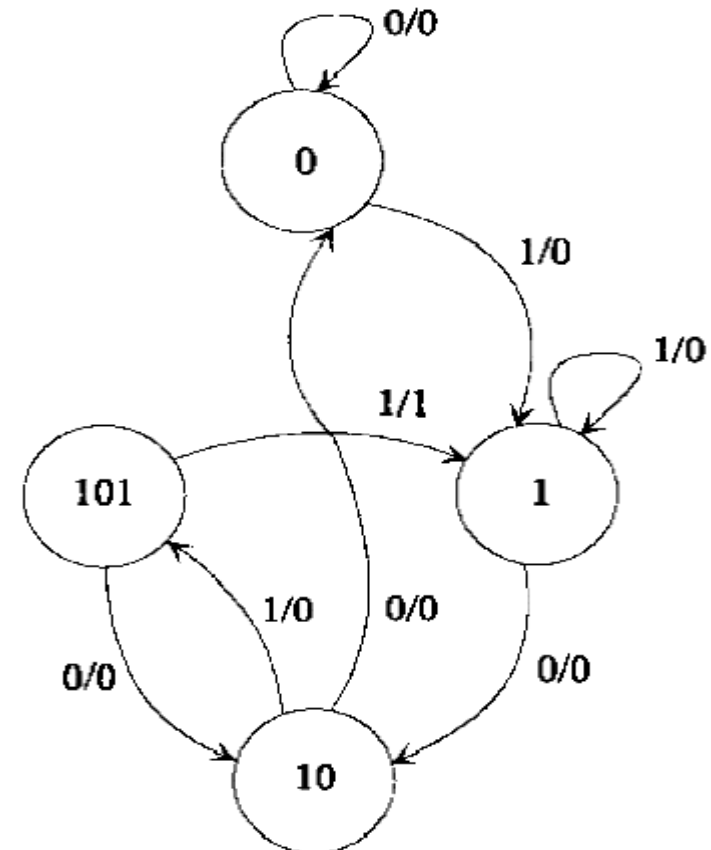
```
PROCESS
BEGIN
CASE current IS
  WHEN reset =>
    WAIT UNTIL clk = '1';
    IF x = '1' THEN current <= got1; ELSE current <= reset;
    END IF;
  WHEN got1 =>
    WAIT UNTIL clk = '1';
    IF x = '0' THEN current <= got10; ELSE current <= got1;
    END IF;
  WHEN got10 =>
    WAIT UNTIL clk = '1';
    IF x = '1' THEN current <= got101; ELSE current <= reset;
    END IF;
  WHEN got101 =>
    WAIT UNTIL clk = '1';
    IF x = '1' THEN current <= got1011; ELSE current <= got10;
    END IF;
  WHEN got1011 => z <= '1';
    WAIT UNTIL clk = '1';
    IF x = '1' THEN current <= got1; ELSE current <= got10;
    END IF;
END CASE;
```

Descrizione comportamentale di una FSM

```
WAIT FOR 1 NS;  
z <= '0';  
END PROCESS;  
END behavioral_state_machine;
```

Riconoscitore di stringa (Mealy)

- Il riconoscitore della stringa di ingresso 1011 può essere modellato attraverso una macchina di Mealy schematizzata nel grafo seguente:
- Come si è visto VHDL permette di rappresentare diverse versioni di una macchina a stati finiti, a diversi livelli di astrazione.
- Nella descrizione VHDL, per modellare il riconoscitore di stringhe attraverso una macchina di Mealy, si ricorre ai costrutti di assegnazione con guardia e funzione di risoluzione.



Assegnazione con guardia sui segnali

- Un'assegnazione con guardia è un'istruzione di assegnamento che è eseguita solo quando l'espressione booleana ad essa associata è valutata come vera.
- Il segnale che compare alla sinistra di un'assegnazione con guardia è detto quindi “segnale con guardia” (guarded signal).
- La condizione di guardia può essere definita implicitamente utilizzando uno statement **Block** con espressione di guardia.
- Tale blocco conterrà nella prima riga della sua definizione la condizione che costituisce la guardia.

```
label_b: BLOCK (Condizion_Guard)  
BEGIN  
w_signal <= GUARDED x_signal AFTER delay_a;  
z_signal <= y_signal AFTER delay_b;  
END
```

w_signal è un segnale
con guardia

Assegnazione con guardia sui segnali

```
label_b: BLOCK (Condizion_Guard)  
BEGIN  
w_signal <= GUARDED x_signal AFTER delay_a;  
z_signal <= y_signal AFTER delay_b;  
END
```

- All'interno del Block, le istruzioni di assegnazione con guardia sono ottenute ricorrendo alla parola chiave GUARDED
- Se è presente "GUARDED", la singola istruzione di assegnazione è eseguita **solo** quando l'espressione di guardia viene valutata come **vera**.
- Quando la condizione di guardia è falsa, l'assegnamento non viene eseguito, qualunque sia la forma d'onda dei segnali sul lato destro dell'assegnamento
- Quando la guardia è falsa i segnali sul lato sinistro dell'assegnamento sono detti disconnessi dai loro rispettivi driver

Assegnazione con guardia: Esempio

Un blocco deve cominciare con un'etichetta

```
label_block: BLOCK (Condizion_Guard)
```

```
BEGIN
```

```
output_signal_1 <= GUARDED input_signal AFTER delay_a;
```

```
output_signal_2 <= input_signal AFTER delay_b;
```

```
END
```

L'assegnamento su **output_signal_1** ha effetto solo se l'espressione booleana **Condizion_Guard** risulta vera

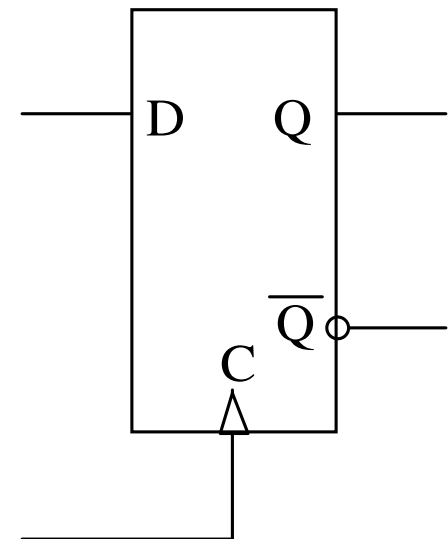
Se **Condizion_Guard** è falsa, l'assegnamento che coinvolge **output_signal_1** non sarà eseguito, anche se occorre l'evento che aggiorna il valore di **input_signal**

Al cambiare del valore di **input_signal** si avrà un cambio di valore di **output_signal_2** (dopo un tempo **delay_b**), mentre il valore di **output_signal_1** (dopo un tempo **delay_a**) sarà aggiornato solo se la condizione **Condizion_Guard** risulta vera

Flip Flop D

```
ENTITY D_flipFlop IS  
Generic (delay1:TIME:=4 ns; delay2:TIME:=5NS);  
PORT (d,c:IN BIT; q,qb:OUT BIT);  
END D_flipFlop;
```

```
ARCHITECTURE Guarding OF D_flipFlop IS  
BEGIN  
ff:BLOCK (c='1' AND NOT c'STABLE)  
BEGIN  
q <= GUARDED d AFTER delay1;  
qb <= GUARDED NOT d AFTER delay2;  
END BLOCK ff;  
END Guarding;
```



Blocchi Innestati

- Un blocco può contenere al suo interno la definizione di un altro blocco innestato,
 - che a sua volta, può contenere un altro blocco...

Se i blocchi innestati sono con guardia allora:

- Dato un assegnamento con guardia, la guardia fa riferimento a tutti i blocchi in cui l'assegnamento è contenuto

```
Outer_b: BLOCK (Outer_Guard)
BEGIN
  w <= GUARDED x AFTER delay_a;
  Inner_b: BLOCK (Inner_Guard)
  BEGIN
    z <= GUARDED y AFTER delay_b;
  END
END
```

L'assegnamento su **w** ha effetto solo se **Outer_Guard** risulta vera

L'assegnamento su **z** ha effetto solo se la condizione booleana **Outer_Guard AND Inner_Guard** risulta vera

- L'espressione **guard** è ottenuta ponendo in AND le condizioni di guardia di tutti i blocchi nello scope

Funzione di risoluzione per segnali con più Driver

- Quando si hanno diverse sorgenti per lo stesso segnale, il valore che tale segnale assumerà non è determinato.
- Ad esempio, date le seguenti assegnazioni concorrenti, in cui si hanno più sorgenti per lo stesso segnale **USignal**

```
USignal <= a;
```

```
USignal <= b;
```

```
USignal <= c;
```

```
USignal <= d;
```

- Si otterrebbe un messaggio di errore, infatti supponiamo il caso in cui, ad un determinato istante, **a** valga '0' e **b**'1', quale valore dovrebbe assumere **USignal**?
- Per risolvere questo problema si ricorre alla **Funzione di Risoluzione**

Funzione di risoluzione

- La funzione di Risoluzione è applicata ad un segnale, ed ha il compito di definire il valore che esso dovrà assumere; essa è definita nella parte dichiarativa del segnale a cui è applicata.

```
FUNCTION Resolv_fun_Name (Drivers: Signal_Type_Vector)  
                                RETURN Signal_Type;
```

- Tale funzione, è invocata ogni volta che occorre un evento su uno qualsiasi degli input per il segnale a cui è applicata
- Quando invocata, restituisce un valore dello stesso tipo del segnale, tale valore sarà assegnato al segnale.
- I parametri di input della funzione devono essere di tipo vettore, i cui elementi sono dello stesso tipo del segnale a cui è associata.

Funzione di risoluzione

- Consideriamo, la funzione di risoluzione, che al singolo segnale, associa la concatenazione in AND di tutte le sue sorgenti multiple

```
--USE qit, qit_vector, AND from basic_utilities
```

```
FUNCTION Anding (Drivers: qit_vector) RETURN qit IS
```

```
    VARIABLE accumulate: qit := '1';
```

variabile locale in cui si memorizza
il risultato dell'AND dei qit in input

```
BEGIN
```

```
    FOR I IN driver' RANGE LOOP
```

```
        accumulate := accumulate AND drivers(i);
```

Per tutti gli elementi
presenti nel vettore

```
END LOOP;
```

```
RETURN accumulate;
```

```
END Anding;
```

- Utilizziamo tale funzione per la risoluzione dell'assegnamento concorrente sul segnale Usignal

Funzione di risoluzione: Esempio

```
USE WORK.basic.utilities.ALL
ARCHITECTURE wired_and OF circuit_component
FUNCTION Anding (Drivers: qit_vector) RETURN qit IS
    VARIABLE accumulate: qit := '1';
BEGIN
    FOR I IN driver'RANGE LOOP
        accumulate := accumulate AND drivers(i);
    END LOOP;
    RETURN accumulate;
END Anding;
SIGNAL USignal : Anding qit;
    USignal <= a;
    USignal <= b;
    USignal <= c;
    USignal <= d;
    ...
END wired_and
```

Mediante questo statement, la funzione di risoluzione è applicata al segnale USignal, di tipo qit (in logica a 4 valori)

I 4 assegnamenti concorrenti, applicati allo stesso segnale USignal, sono **risolti** tramite la funzione Anding: un evento su **ALMENO uno** dei segnali in input provoca l'invocazione della funzione Anding, che assegna ad USignal il valore determinato dall'AND di a,b,c,d. VHDL non specifica l'ordine con cui le diverse sorgenti sono concatenate.



Funzione di risoluzione

- La funzione di risoluzione è invocata ad ogni ciclo di simulazione in cui il segnale risolto è attivo.
- Ogni volta che viene invocata, riceve in input un array di valori, contenente un elemento per ogni sorgente del segnale.
- Gli elementi di tale vettore in input sono tutte le sorgenti del segnale per cui la guardia è **vera**.
- Quando, in un assegnamento, la guardia è **falsa**, il valore della relativa sorgente non viene inserito tra gli elementi del vettore
- In questo caso si dice
 - che il driver relativo a quella sorgente è disconnesso oppure,
 - che la transazione relativa è nulla
- Può capitare quindi, che tutti gli assegnamenti con guardia siano disconnessi, e che alla funzione di risoluzione sia passato in input un vettore vuoto.

Funzione di risoluzione: Tipo di segnale

- In questo caso, la funzione avrà un comportamento definito dal tipo di segnale a cui è associata al momento della dichiarazione:

```
signal signal_name : resol_function_name [ signal_kind ]  
                                     [ := espressione_defaul ] ;
```

```
signal_kind ::= bus | register
```

- Se il segnale è di tipo **bus**, ovvero se la connessione multipla ad esso viene modellata come un bus, la funzione di risoluzione, se invocata con un vettore vuoto, restituisce il valore che rappresenta l'output di default del bus quando non c'e' nessun segnale che lo pilota
- Se il segnale è di tipo **register**, allora la funzione di risoluzione lascia inalterato il valore che il segnale aveva prima che essa fosse invocata.

Riconoscitore di stringa (Mealy)

- Di seguito riportiamo il codice sorgente che descrive il riconoscitore di stringa modellato mediante una macchina di Mealy

```
LIBRARY ieee;  
USE ieee.std_logic_1164.ALL;
```

```
entity detector is  
  port(x, clk: in bit; z: out bit);  
end detector;
```

```
architecture singular_state_machine of detector is  
  type state is (reset, got1, got10, got101);  
  type state_vector is array(natural range <>) of state;  
  function one_of (sources: state_vector) return state is  
begin  
  return sources(sources'left);  
end one_of;
```

```
signal current: one_of state register := reset;
```

Mediante la funzione di risoluzione il segnale Current ha in ogni istante un unico valore di pilotaggio

La funzione di risoluzione accetta in input un vettore di stati di dimensione non prefissata

L'operatore LEFT restituisce il primo indice del vettore:
sources(sources'left) è quindi il primo elemento del vettore

Riconoscitore di stringa (Mealy)

```
begin
  clocking: block (clk='1' and not clk'stable)
  begin
    s1: block (current=reset and guard)
    begin
      current<=guarded got1 when x='1' else reset;
    end block s1;

    s2: block (current=got1 and guard)
    begin
      current<=guarded got10 when x='0' else got1;
    end block s2;

    s3: block (current=got10 and guard)
    begin
      current<=guarded got101 when x='1' else reset;
    end block s3;
```

Vi sono due blocchi innestati, la guardia è costituita dall'espressione **current=reset AND (clk='1' and not clk'stable)**

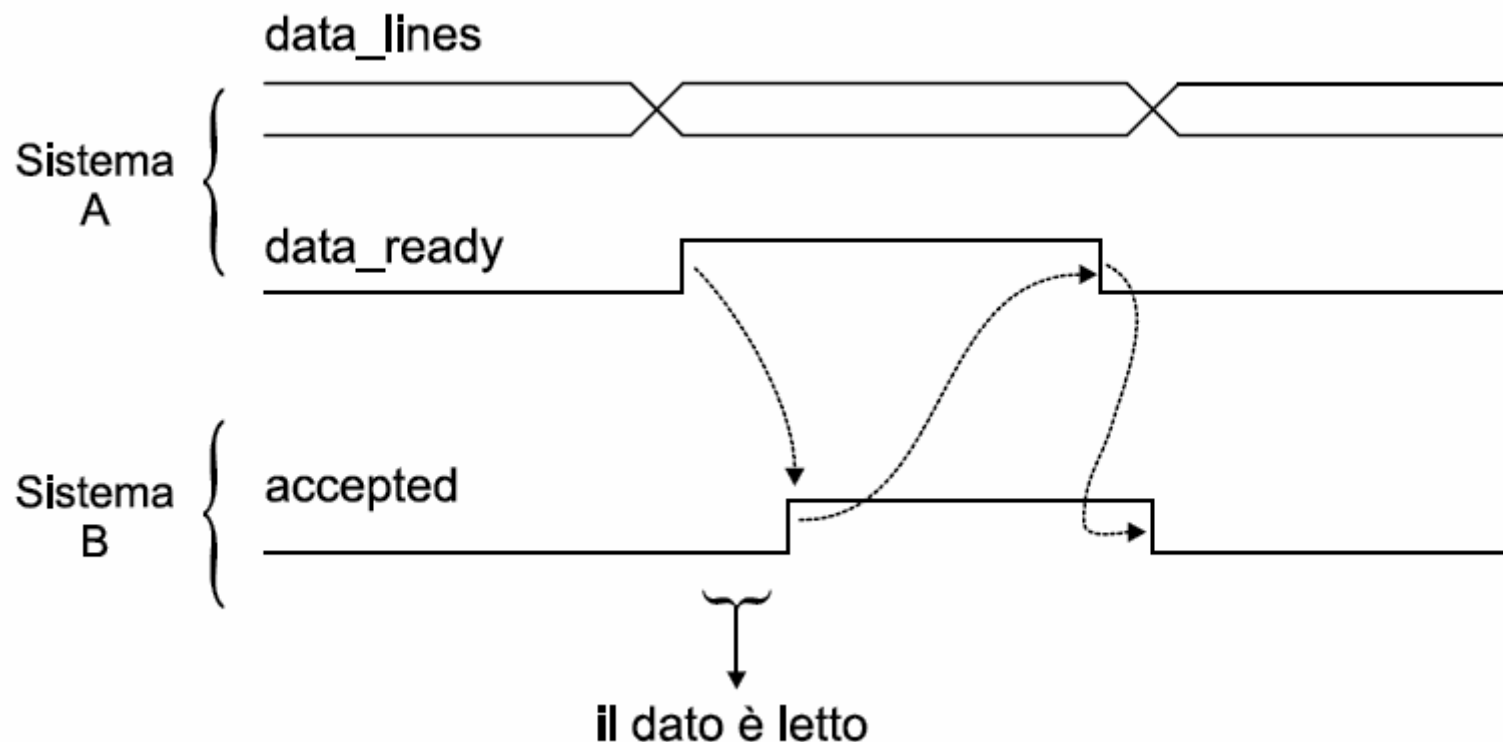
Riconoscitore di stringa (Mealy)

```
s4: block (current=got101 and guard)
begin
    current<=guarded got1 when x='1' else got10;
    z<='1' when (current=got101 and x='1') else '0';
end block s4;
end block clocking;
end singular_state_machine;
```

Statements
concorrenti

Un altro esempio: handshaking

- Immaginiamo di avere due sistemi, A e B, in grado di dialogare tra loro mediante un semplice protocollo di handshaking.
- La figura sottostante mostra i dettagli del protocollo.



Esempio: handshaking

- Il comportamento dei due sistemi, per quanto riguarda il loro modo di comunicare con l'esterno, può essere descritto molto semplicemente in VHDL, ricorrendo allo statement wait.

- Sistema A:

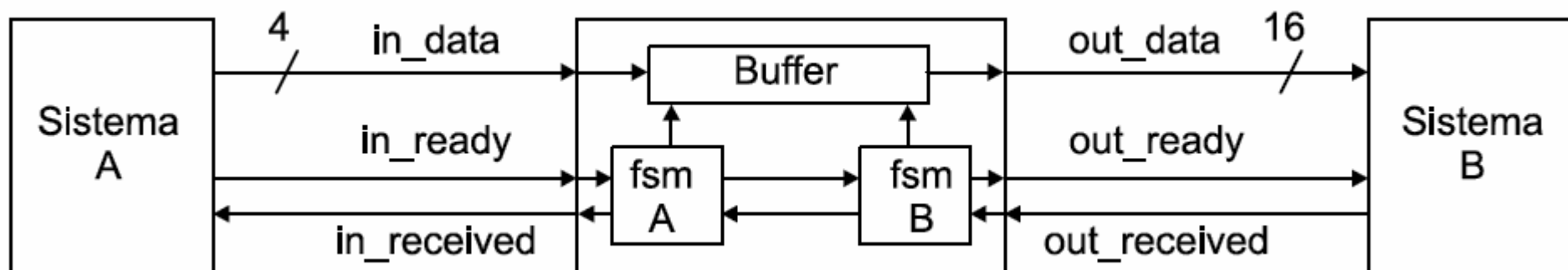
```
--quando pronto, il sistema A pone il dato sul bus ed alza data_ready  
data_lines <= newly_prepared_data;  
data_ready <= 1;  
--ricevuto l'acknowledgement, la situazione iniziale è ripristinata  
WAIT UNTIL accepted = '1';  
data_ready <= 0;
```

- Sistema B:

```
--attende il segnale di data_ready, alza accepted quando il dato è ricevuto  
WAIT UNTIL data_ready = '1';  
accepted <= '1';  
--ricevuto l'acknowledgement, la situazione iniziale è ripristinata  
WAIT UNTIL data_ready = '0';  
accepted <= '0';
```

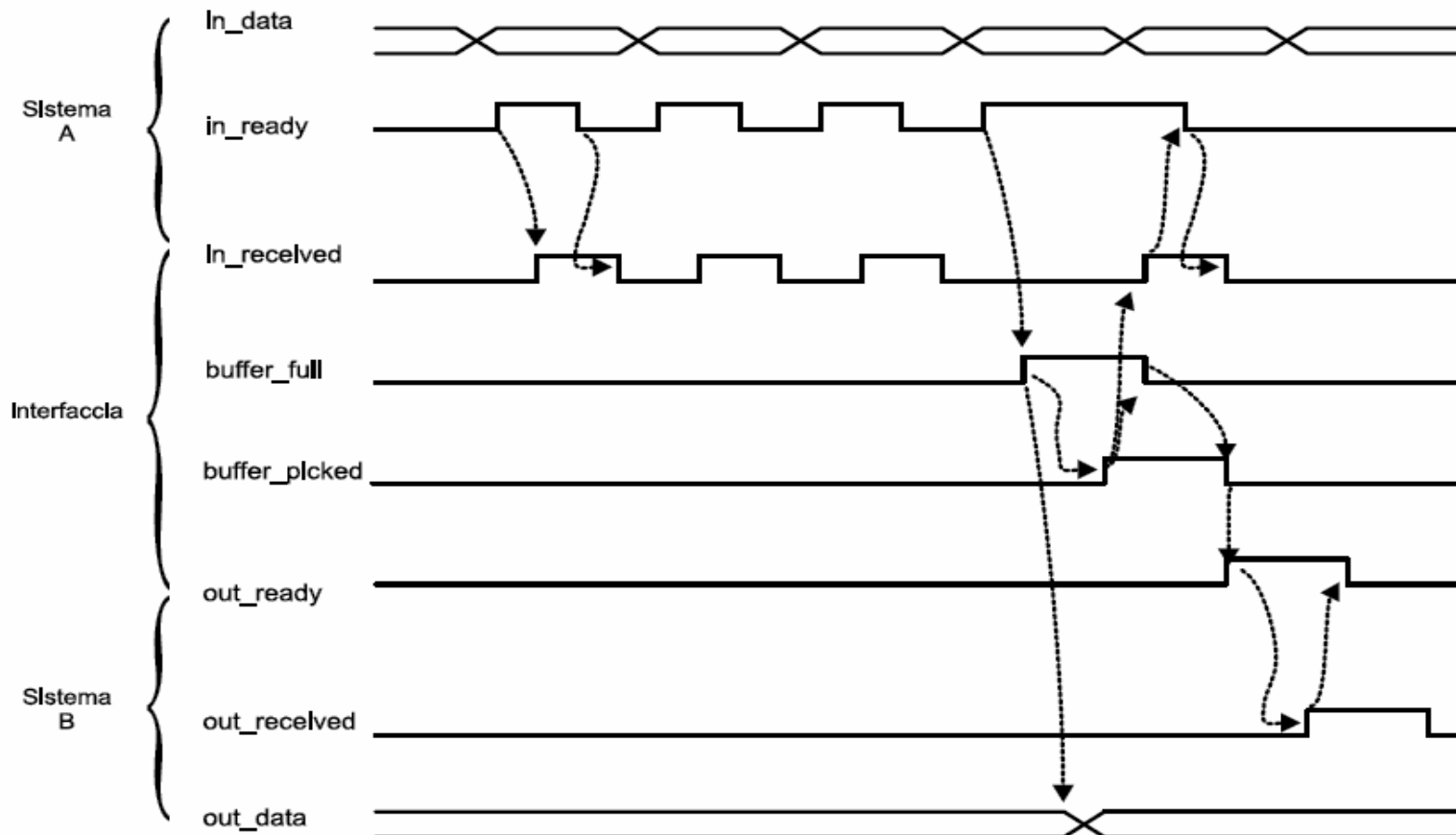
Esempio: handshaking

- Adesso immaginiamo che i due sistemi non abbiano lo stesso parallelismo nel trasferimento di dati.
- Il sistema A comunica tramite un bus di 4 bit in uscita.
- Il sistema B legge da un bus di 16 bit in ingresso.
- Vogliamo realizzare un blocco di interfaccia che legga sequenzialmente quattro parole di 4 bit dal sistema A, memorizzandole in un buffer di 16 bit, e che trasferisca poi l'intera parola al sistema B.
- Questo blocco deve essere composto dal buffer e da due fsm, collegate rispettivamente al sistema A e B, in grado di dialogare tra loro.



Esempio: handshaking

- Le due fsm dialogano tra loro mediante i segnali *buffer_full* e *buffer_picked*.



Esempio: handshaking

- Il blocco di interfaccia è descritto tramite due process corrispondenti ciascuno ad una fsm; al loro interno è usato lo statement wait.

```
ENTITY system_i IS
PORT ( in_data : IN BIT_VECTOR (3 DOWNT0 0);
      out_data : OUT BIT_VECTOR (15 DOWNT0 0);
      in_ready, out_received : IN BIT;
      in_received, out_ready : OUT BIT);
END system_i;
```

```
ARCHITECTURE waiting_arch OF system_i IS
SIGNAL buffer_full, buffer_picked : BIT := '0';
SIGNAL word_buffer : BIT_VECTOR (15 DOWNT0 0);
BEGIN
Fsm_A: PROCESS
BEGIN
. . .
-- Dialoga con A, raccoglie 4 parole da 4 bit, mantiene il conto delle
-- parole lette.
-- Terminata la lettura, passa il dato a 16 bit al processo fsm_B
. . .
END PROCESS fsm_A;
```

Esempio: handshaking

```
Fsm_B: PROCESS
```

```
BEGIN
```

```
. . .
```

```
-- Aspetta un dato a 16-bit dal processo fsm_A
```

```
-- Una volta ricevuto il dato, dialoga con il sistema B
```

```
-- per passarglielo
```

```
. . .
```

```
END PROCESS Fsm_B;
```

```
END waiting_arch;
```

Esempio: handshaking

Fsm_A: PROCESS

```
VARIABLE count : INTEGER RANGE 0 TO 4 := 0;
BEGIN
  WAIT UNTIL in_ready = '1';
  count := count + 1;
  CASE count IS
    WHEN 0 => NULL;
    WHEN 1 => word_buffer (03 DOWNT0 00) <= in_data;
    WHEN 2 => word_buffer (07 DOWNT0 04) <= in_data;
    WHEN 3 => word_buffer (11 DOWNT0 08) <= in_data;
    WHEN 4 => word_buffer (15 DOWNT0 12) <= in_data;
    buffer_full <= '1'; WAIT UNTIL buffer_picked = '1';
    buffer_full <= '0'; count := 0;
  END CASE;
  in_received <= '1';
  WAIT UNTIL in_ready = '0';
  in_received <= '0';
END PROCESS Fsm_A;
```

Esempio: handshaking

```
Fsm_B: PROCESS
```

```
BEGIN
```

```
IF buffer_full = '0'
```

```
    THEN WAIT UNTIL buffer_full = '1'; END IF;
```

```
    out_data <= word_buffer;
```

```
    buffer_picked <= '1';
```

```
    WAIT UNTIL buffer_full = '0';
```

```
    buffer_picked <= '0';
```

```
    out_ready <= '1';
```

```
    WAIT UNTIL out_received = '1';
```

```
    out_ready <= '0';
```

```
END PROCESS Fsm_B;
```

Codifica dello stato

- Negli esempi presentati si è sempre fatto ricorso ad un tipo enumerato per introdurre i possibili stati del sistema descritto. Qualora il codice VHDL sia sottoposto ad un programma di sintesi per derivarne una netlist, lo stato sarà automaticamente codificato attraverso opportuni valori binari.

```
TYPE state IS (reset, got1, got10, got101, got1011);
```

- In generale, ci si può aspettare che la codifica risponda a criteri di ottimizzazione.
- In qualche caso, comunque, può essere necessario controllare esplicitamente la codifica dello stato.
- Per esempio, in qualche caso può essere desiderabile avere una codifica *one hot* dello stato.

Codifica dello stato

- In generale, per definire esplicitamente come, allo stato “astratto” definito nel codice VHDL, verrà associato un insieme di bit durante la fase di sintesi, occorre intervenire sul programma di sintesi, con modalità che non sono generalizzabili.
- Un modo molto diffuso per “comunicare” col programma di sintesi consiste nel fornire indicazioni tramite attributi VHDL. Un esempio può essere il seguente:

```
TYPE state IS (reset, got1, got10, got101, got1011);
```

```
ATTRIBUTE enum_encoding OF state IS : TYPE IS 000 001 011 101 111;
```

- Nell'esempio si usa un attributo su un tipo (il tipo *state*) consistente in una stringa che elenca la codifica richiesta per gli elementi del tipo. Il sintetizzatore che metta a disposizione questa tecnica riconoscerà l'attributo e si comporterà di conseguenza.
- Sottolineiamo ancora che una tale modalità e il riconoscimento stesso dell'attributo dipende dal particolare programma di sintesi usato.

Codifica dello stato

- Un modo più generale che è possibile usare consiste nel definire esplicitamente i valori dello stato come vettori di bit. E' poi possibile definire dei vettori costanti cui è possibile dare il nome degli stati, per un immediato riferimento.

```
SUBTYPE state IS std_logic_vector(2 DOWNTO 0);  
CONSTANT      reset   : state := 000;  
CONSTANT      got1    : state := 001;  
CONSTANT      got10   : state := 011;  
CONSTANT      got101  : state := 101;  
CONSTANT      got1011: state := 111;  
SIGNAL current : state;
```

```
Current<= got10;
```