Attività integrativa di laboratorio/progetto di "Elettronica dei Sistemi Digitali 1" e "Calcolatori Elettronici"

INTRODUZIONE AL VHDL

PROGETTARE UTILIZZANDO IL VHDL:
DALL'ESEMPIO AL COSTRUTTO

Ringraziamenti

Questo dispensa è stata sviluppata nell'ambito del corso di Calcolatori Elettronici tenuti presso la Facoltà di Ingegneria del politecnico di Milano – Sede di Como ed è stata prevalentemente redatta da Alessandro Monti a cui porgo i miei più sentiti ringraziamenti.

Fabio Salice

Indice

Introduzione

Lo scopo di questo documento è quello di costruire un percorso formativo alla conoscenza del VHDL attraverso una serie di esempi che, di volta in volta, evidenziano alcuni aspetti fondamentali del linguaggio.

In primo luogo, il linguaggio prescelto (VHDL - Hardware Description Language) è un linguaggio che consente di descrivere i dispositivi che verranno realizzati in hardware sia delineando la relazione (connessione) tra le varie componenti che lo costituiscono, sia descrivendo l'attività che ogni componente svolge e la sua interfaccia.

Per quanto riguarda quest'ultimo punto, un aspetto preliminare che risulta significativo evidenziare riguarda il fatto che ogni dispositivo, qualunque sia la sua natura, è descritto separando gli elementi relativi l'interfaccia da quelli che riguardano la funzionalità attuata che, a sua volta, può essere descritta in modi differenti. A questo proposito, si prenda in analisi il seguente esempio: un *decoder* a 3 ingressi e 8 uscite in cui una configurazione di tre bit in ingresso al blocco funzionale in analisi indica quale delle otto linee di uscita dovrà assumere un valore pari ad 1 (ad esempio, per 000 si attiverà la linea 0 mentre, in relazione alla configurazione 111 si attiverà la linea 7).

Per quanto riguarda l'interfaccia, il dispositivo sarà costituito da 3 bit di ingresso e 8 bit di uscita come indicato con dalla descrizione fatta del dispositivo in analisi.

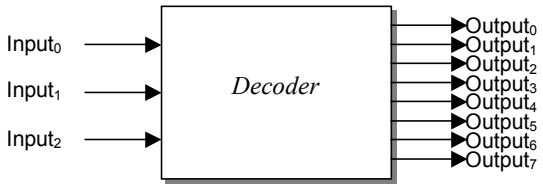


Figura 1. pin-out logico del decoder 3-8

Per quel che riguarda l'aspetto relativo alla funzionalità, è interessante evidenziare che, anche nel consueto modo di operare, una rete combinatoria (ma non solo) può essere rappresentata mediante una descrizione che mette in relazione le configurazioni di ingresso con le configurazioni di uscita (tabella della verità, tabella delle implicazioni, …) oppure attraverso un insieme di espressioni logiche oppure, infine, mediante una rappresentazione circuitale.

Per l'esempio preso come riferimento, le tre descrizioni sono nella forma di:

- espressioni logiche:

  y_0=!a!b!c; y_1=!a!bc; y_2= !ab!c; y_3= !abc; y_4= a!b!c; y_5= a!bc; y_6=ab!c; y_7=abc

- tabella della verità:

x_1 x_2 x_3	$y_0y_1y_2y_3y_4y_5y_6y_7$
0 0 0	1 0 0 0 0 0 0 0
0 0 1	0 1 0 0 0 0 0 0
0 1 0	0 0 1 0 0 0 0 0
0 1 1	0 0 0 1 0 0 0 0
1 0 0	0 0 0 0 1 0 0 0
1 0 1	0 0 0 0 0 1 0 0
1 1 0	0 0 0 0 0 0 1 0
1 1 1	0 0 0 0 0 0 0 1

- schema circuitale:

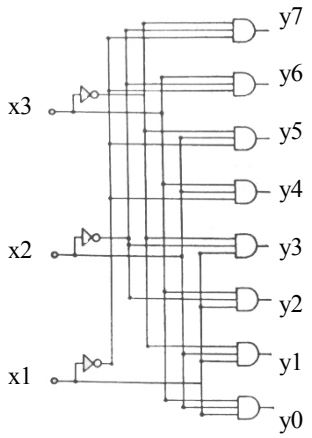


Figura 2 Una possibile rappresentazione circuitale del decoder a 3 ingressi.

Si osservi che le prime due modalità di rappresentazione identificano degli aspetti legati al *comportamento* poiché identificano le relazioni tra le configurazioni di ingresso e di uscita mentre l'ultimo schema di descrizione mette in risalto gli aspetti strutturali. Facendo specifico riferimento al linguaggio di descrizione dell'hardware in esame, esiste una corrispondenza con i modi per descrivere la funzionalità di un dispositivo indicati in precedenza e la loro rappresentazione in VHDL.

Le prossime sezioni hanno l'obiettivo di identificare come possa essere specificata, in VHDL, l'interfaccia del dispositivo e la funzionalità. Il problema del progetto di sistemi complessi verrà scomposto in alcune sezioni ognuna delle quali introdurrà, a partire da semplici questioni, sia dei principi di progettazione dei dispositivi digitali sia i costrutti tipici del linguaggio in esame.

Ogni sezione include un paragrafo di riassunto relativo ai costrutti introdotti ed una sezione di esempi sia proposti sia proposti e risolti.

Il testo termina con una serie di ulteriori esempi che sono caratterizzati da alcuni aspetti particolarmente significativi come, ad esempio, l'introduzione di parametri che ne generalizzano la funzionalità oppure la descrizione di una RAM che risulta di particolare interesse in molte applicazioni.

Reti combinatorie completamente specificate: descrizioni non gerarchiche

In questa sezione si analizzerà il problema relativo alla descrizione di una rete puramente combinatoria (il valore assunto dalla uscite dipende solo dagli ingressi e non dal tempo) completamente specificata priva di gerarchia. Quest'ultimo aspetto indica che le entità analizzate in questa sezione sono da considerarsi elementari cioè sono entità su cui non è necessario svolgere alcuna ulteriore scomposizione (livello terminale della scomposizione top-down di un progetto).

Descrizione dell'interfaccia del dispositivo (`entity`)

In VHDL ogni `entità` viene rappresentata descrivendone l'interfaccia cioè specificando gli ingressi della rete e le sue uscite senza preoccuparsi di come tale rete venga realizzata.

L'interfaccia viene dichiarata indicando ciò che si vuole descrivere (all'interno del costrutto **`entity`**) specificando per ogni ingresso ed uscita sia la direzione (`input`, `output`, `inout` [bidirezionale]) sia il tipo e la quantità di bit coinvolti (all'interno del costrutto **`port`**). Nel caso in esame (Figura 1) viene dichiarato un solo ingresso ed una sola uscita di tipo **`bit_vector`**; quest'ultimo è un tipo predefinito del linguaggio in esame ed è un array di bit di dimensione specificata.

```
ENTITY decoder IS
  PORT (input: IN bit_vector(0 TO 2);
    output: OUT bit_vector(0 TO 7));
END decoder;
```

Descrizione della relazione tra ingressi ed uscite (architecture)

Il passo successivo corrisponde alla descrizione delle relazioni tre ingresso ed uscita. Per quanto specificato in precedenza, è possibile descrivere la funzionalità di una specifica entità descrivendone o il comportamento o la struttura. Qui di seguito si analizzano questi aspetti.

Descrizione Comportamentale

Una funzionalità puramente combinatoria (si fa riferimento solo a questa tipologia rimandando a sezioni successive l'analisi di funzionalità in cui è presente il concetto di stato – Macchine a Stati Finiti -) può essere descritta specificando, per ogni *segnale* di uscita, il legame con i segnali di ingresso in termini di espressione logica (all'interno del costrutto **`architecture`**). Si anticipa fin d'ora che, in VHDL, esiste una differenza sostanziale tra **segnali** (**signal**) e **variabili** (**variable**) dovuta agli istanti di tempo in cui ne vengono aggiornati i valori.

Quando si dichiara un'entità **tutti i suoi ingressi e le sue uscite sono segnali**; l'operatore di assegnamento di un segnale è *signal* **<=** *expr* intendendo con *signal* un generico segnale (sia elementare sia strutturato – es. un array di bit -) e con *expr* una generica espressione logica; si osservi che il tipo generato da una *expr* deve essere compatibile con quello del signal cioè se *signal* è uno scalare (un bit, ad esempio) anche l'espressione deve generare uno scalare. Questa osservazione è particolarmente intuitiva se si pensa che non esiste alcun modo per collegare direttamente un *filo* con molti *fili* senza generare un potenziale conflitto (nota: anche se, dal punto di vista hardware, collegare un filo a più fili non costituisce un problema, si preferisce lasciare l'intero controllo al progettista che deve svolgere questa operazione in modo esplicito).

Nell'esempio qui di seguito riportato (decodBH.vhd), il decoder viene descritto utilizzando le espressioni logiche che lo rappresentano.

```
ARCHITECTURE EspressioniLogiche OF decoder IS
Begin
  output(0)<=NOT input(0)AND NOT input(1) AND NOT input(2);
  output(1)<=NOT input(0)AND NOT input(1) AND input(2);
  output(2)<=NOT input(0)AND input(1) AND NOT input(2);
  output(3)<=NOT input(0)AND input(1) AND input(2);
  output(4)<=input(0) AND NOT input(1) AND NOT input(2);
  output(5)<=input(0) AND NOT input(1) AND input(2);
  output(6)<=input(0) AND input(1) AND NOT input(2);
  output(7)<=input(0) AND input(1) AND input(2);
End EspressioniLogiche;
```

Una prima significativa osservazione è che le **istruzioni contenute nel corpo dell'architettura vengono eseguite in modo concorrente** (parallelamente). In pratica, il risultato prodotto della entità associata alla descrizione di una generica architettura non dipende dall'ordine con cui le singole righe di codice sono state scritte; in altre parole, le righe di codice contenute tra le parole chiave **`begin`** e **`end`** possono essere riordinate in qualunque modo ed il risultato ottenuto è esattamente lo stesso. Questo aspetto corrisponde esattamente a quanto ci si aspetta da un dispositivo hardware: la configurazione d'uscita (a regime) di una rete combinatoria dipendente solo dalla configurazione di ingresso (e, ad esempio, non dipende dal tempo o da come le espressioni logiche sono ordinate).

In VHDL è possibile descrivere anche dei comportamenti sequenziali. Per questo scopo si utilizza il costrutto **`process`**: all'interno di un **`process`** le istruzioni specificate sono eseguite una dopo l'altra secondo l'ordine indicato.

Quanto detto fin ora può essere riassunto nei seguenti punti:

- una *architecture* può contenere uno o più costrutti che vengono interpretati come elementi da eseguire in parallelo;
- un costrutto può essere anche realizzato da un processo (`process`);
- il contenuto di ogni `process` viene eseguito sequenzialmente.

Dal punto di vista della simulazione, i `process` vengono attivati nella fase di inizializzazione della simulazione ed eseguiti ripetutamente. Questo comportamento, che può essere indesiderato, può essere controllato imponendo che un dato `process` venga sospeso e riattivato solo allo scadere di uno specifico lasso temporale oppure in relazione ad un evento (variazione di un segnale). A questo fine è possibile adottare due soluzioni alternative: esplicitare un comando di attesa (istruzione *wait* che può fare riferimento o al tempo o ad uno o più eventi) oppure indicare in modo implicito i segnali che riattivano il processo utilizzando la sintassi `process(segnale`$_1$`, segnale`$_2$`, … segnale`$_n$`)`. La lista dei segnali di riattivazione di un processo viene denominata `sensitivity list`.

Nell'esempio che segue la riattivazione di un processo è implicitamente contenuta nella scrittura **`process`**`(`**`input`**`)`, dove il processo viene eseguito ogni volta che il segnale `input` cambia di valore.

```
ARCHITECTURE TabellaVerita OF decoder IS
BEGIN
  codifica: PROCESS(input)
  BEGIN
    CASE input IS
      WHEN B"000" => output <= B"10000000";
      WHEN B"001" => output <= B"01000000";  -- il costrutto case e'
      WHEN B"010" => output <= B"00100000";  -- sequenziale
      WHEN B"011" => output <= B"00010000";  -- e puo'essere utilizzato
      WHEN B"100" => output <= B"00001000";  -- solo in un process
      WHEN B"101" => output <= B"00000100";
      WHEN B"110" => output <= B"00000010";
      WHEN B"111" => output <= B"00000001";
    END CASE;
  END PROCESS;
End TabellaVerita;
```

I costrutti che per definizione sono sequenziali possono essere utilizzati solo nei *process*, mentre i costrutti concorrenti si possono usare anche nel corpo dell'architettura, come di seguito:

```
ARCHITECTURE TabellaVerita2 OF decoder IS
BEGIN
  output<= B"10000000" when input = B"000"   -- il costrutto A<= when B
    else B"01000000" when input = B"001" --                  else C;
    else B"00100000" when input = B"010" -- puo' essere utilizzato
    else B"00010000" when input = B"011" -- senza process
    else B"00001000" when input = B"100"
    else B"00000100" when input = B"101"
    else B"00000010" when input = B"110"
    else B"00000001";
End TabellaVerita2;
```

Nei due esempi riportati in precedenza, il decoder è stato descritto utilizzando la sua tabella della verità. È rilevante sottolineare la differenza tra l'ultima e la penultima descrizione; in particolare, la descrizione della architettura `TabellaVerita` contiene un `process` al cui interno è utilizzato il costrutto di controllo **`case`** che è **caratterizzato da una natura sequenziale e, di conseguenza, può essere utilizzato solo in un `process`**, mentre l'architettura `TabellaVerita2` utilizza il costrutto di controllo **`when-else`** che può essere utilizzato anche in assenza di `process`.

Le tre architetture appena descritte sono tutte contenute in un unico file in cui è dichiarata un'unica entità decoder; è interessante sottolineare che le tre architetture rappresentano tre differenti alternative dello stesso comportamento. In generale, è possibile dare differenti descrizioni dello stesso comportamento implementando, ad esempio, differenti algoritmi che risolvono lo stesso problema e selezionare, in momenti successivi, quello di maggior interesse (quella a minor area, quella con migliori prestazioni …); la scelta viene svolta in fase di simulazione e sintesi quando sarà indispensabile esplicitare quale entità utilizzare.

Descrizione strutturale

Una descrizione strutturale consiste nel realizzare una `architecture` che rappresenti esplicitamente uno schema circuitale; tale descrizione viene attuata indicando i componenti coinvolti e descrivendo la relazione che esiste tra ogni ingresso ed ogni uscita. Consideriamo il seguente esempio. In primo luogo, si supponga che il *componente* utilizzato sia una porta `and` a due ingressi la cui dichiarazione è:

```
COMPONENT And2
  PORT(i1,i2: IN std_logic; o: OUT std_logic);
END COMPONENT;
```

dove, come è avvenuto nel caso della `entity`, la parola chiave `port` consente di specificare l'interfaccia del componente stesso (due ingressi ed una uscita). Si ipotizzi, ora, di voler realizzare una descrizione strutturale che, per puro scopo esemplificativo, faccia uso (*istanza di*) due soli componenti (`primoAND` e `secondoAND`) in cui un ingresso è in comune (`x`) mentre l'uscita di una porta (`z`) è ingresso dell'altra; la seguente descrizione rappresenta da questa sezione di codice che, per semplicità, focalizza l'attenzione solo su questo aspetto trascurando ogni dichiarazione necessaria (per un esempio completo si veda più avanti).

```
primoAND: And2(i1=>x, i2=>y, o=>z);
secondoAND: And2(i1=>x, i2=>z, o=>w);
```

Si sottolinea che x,y,z e w sono segnali; in particolare, questi o sono dichiarati a livello di `entity` oppure devono essere dichiarate prima del corpo della `architecture` da implementare cioè, prima del `begin`.. In quest'ultimo caso si utilizza la seguente dichiarazione (dove si ipotizza che solo z sia dichiarato nella `architecture`).

```
library ieee;
use ieee.std_logic_1164.all;

ENTITY esempio_dueAND IS
  PORT (x,y: IN std_logic;
        w: OUT std_logic);
END decoder;

ARCHITECTURE esempio OF esempio_dueAND IS
  COMPONENT And2
    PORT(i1,i2: IN std_logic; o: OUT std_logic);
  END COMPONENT;

  SIGNAL z: std_logic;

  BEGIN
    primoAND: And2(i1=>x, i2=>y, o=>z);
    secondoAND: And2(i1=>x, i2=>z, o=>w);
  END esempio
```

Informalmente, il significato di quanto esposto è il seguente: nell'`entity` sono esplicitati tutti i *fili* di collegamento tra il contenuto del componente e l'ambiente esterno mentre, nella `architecture` sono dichiarati tutti i *fili* che collegano gli elementi interni. Si osservi che nulla viene dato per scontato: ogni *filo* presente deve essere dichiarato.

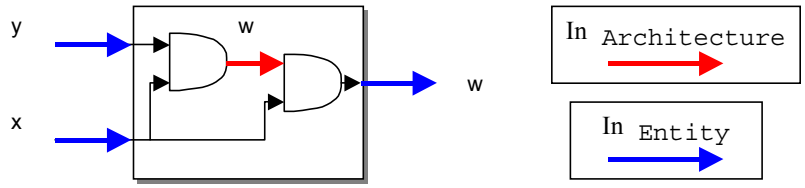


Figura 3. Rappresentazione schematica dell'esempio `esempio_dueAND`.

A livello di dell'architettura del sistema (`architecture`) si osservino le differenze rispetto agli esempi riportati nella sezione precedente:

1. nella parte dichiarativa si definiscono:
   a. i componenti (costrutto `component`) che devono essere utilizzati ed
   b. i segnali che li collegano (costrutto `signal`) non che non sono ne osservabili ne controllabili dall'esterno. Questi ultimi sono presenti nella descrizione della entità in analisi;
2. nel corpo dell'architettura si dovranno solo istanziare i componenti specificando per ognuno come e su quali segnali si mappano gli ingressi ed le uscite.

Un'altra novità introdotta in questa sezione riguarda l'uso della libreria **ieee.std_logic_1164**. Questa altro non è che una libreria standard del VHDL che definisce i dati di tipo std_logic e tutte le operazioni possibili su di essi. Poiché questa libreria è quella utilizzata normalmente, d'ora in avanti si farà uso solo ed esclusivamente di questa. Lo **std_logic** si differenzia dal tipo bit in quanto consente di rappresentare un più ampio insieme di valori oltre a 0 e 1. Questi valori aggiungono la possibilità di segnalare situazioni anomale che, diversamente, non sarebbero visibili in fase di simulazione. Un esempio è il valore 'U' (*Uninitialized*) che identifica una mancata inizializzazione; in altre parole, il comportamento del segnale in analisi non è noto e dipende strettamente da comportamenti non prevedibili del sistema. Questa indicazione, se critica, implica la necessità di forzare il sistema portandolo in uno stato noto come, ad esempio nello stato di inizializzazione o RESET.

TYPE std_ulogic IS ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');

Dove, i vari simboli indicano:

- 'U' *Uninitialized* (non inizializzato);
- 'X' *Forcing Unknown* (valore forzato ma non noto);
- '0' *Forcing 0* (valore forzato a 0);
- '1' *Forcing 1* (valore forzato a 1);
- 'Z' *High Impedance* (Alta impedenza);
- 'W' *Weak Unknown* (valore debolmente non noto);
- 'L' *Weak 0* (valore debole a 0);
- 'H' *Weak 1* (valore debole a 1);
- '-' *Don't care* (condizione di indifferenza);

Facendo ancora riferimento all'esempio del decoder a 3 ingressi e 8 uscite, ci si propone di realizzare la descrizione strutturale mostrata in Figura 2 facendo uso del linguaggio in oggetto (testo dell'esempio:: decodST.vhd).

```
-- Esempio 1; Decoder 3 ingressi 8 uscite: vista strutturale
library ieee;
use ieee.std_logic_1164.all;

ENTITY decoderST IS
  PORT (x: IN std_logic_vector(0 TO 2);
    y: OUT std_logic_vector(0 TO 7));
END decoderST;

ARCHITECTURE Structural OF decoderST IS
  --dichiarazione dei componenti utilizzati per realizzare il dispositivo
  COMPONENT And3
    PORT(a,b,c: IN std_logic; y: OUT std_logic);
  END COMPONENT;

  COMPONENT Not_g
    PORT (u: IN std_logic; y: OUT std_logic);
  END COMPONENT;

  --dichiarazione dei segnali utilizzati per collegare i componenti tra loro
  SIGNAL Nx1,Nx2,Nx3: std_logic;

BEGIN
  --istanziazione dei componenti
  notA: Not_g PORT MAP (u=>x(1), y=>Nx1);
  notB: Not_g PORT MAP (u=>x(2), y=>Nx2);
  notC: Not_g PORT MAP (u=>x(3), y=>Nx3);

  andA: And3 PORT MAP (a=>Nx1, b=>Nx2, c=>Nx3, y=>y(0));
  andB: And3 PORT MAP (a=>Nx1, b=>Nx2, c=>x(3), y=>y(1));
  andC: And3 PORT MAP (a=>Nx1, b=>x(2), c=>Nx3, y=>y(2));
  andD: And3 PORT MAP (a=>Nx1, b=>x(2), c=>x(3), y=>y(3));
  andE: And3 PORT MAP (a=>x(1), b=>Nx2, c=>Nx3, y=>y(4));
  andF: And3 PORT MAP (a=>x(1), b=>Nx2, c=>x(3), y=>y(5));
  andG: And3 PORT MAP (a=>x(1), b=>x(2), c=>Nx3, y=>y(6));
  andH: And3 PORT MAP (a=>x(1), b=>x(2), c=>x(3), y=>y(7));
End Structural;
```

In questo esempio, il decoder viene descritto seguendo le indicazioni riportate nella descrizione circuitale di Figura 1.

E' significativo osservare che la descrizione tutti i componenti a cui si fa riferimento nell'esempio deve essere specificata; in altre parole, deve esistere o deve essere creata una libreria in cui vengono dichiarate e descritte le porte logiche utilizzate. In particolare, le funzionalità di And3 (and a 3 ingressi), And2, Or2 e Not sono descritte in Porte.vhd.

Si osservi che la descrizione della funzionalità delle porte viene realizzata facendo uso del costrutto `process` con una costruzione che ricalca una verità. Si sottolinea che questa operazione è puramente esemplificativa e che potrebbe essere sostituita da una descrizione più semplice e compatta che si riferisce all'uso delle espressioni logiche. Ad esempio, la porta `And3` descritta come:

```
-- porta and a tre ingressi
library ieee;
use ieee.std_logic_1164.all;

ENTITY And3 IS
  PORT (a,b,c: IN std_logic; y: OUT std_logic);
END And3;

ARCHITECTURE tab OF And3 IS
BEGIN
  Elab: process(a,b,c)
  BEGIN
    If a='1' and b='1' and c='1'  then
      y <= '1';
    else
      y <= '0';
    end if;
  END process;
END tab;
```

potrebbe essere sostituita, in modo equivalente, da

```
-- porta and a tre ingressi
library ieee;
use ieee.std_logic_1164.all;

ENTITY And3 IS
  PORT (a,b,c: IN std_logic; y: OUT std_logic);
END And3;
ARCHITECTURE expr OF And3 IS
BEGIN
  y<= a and b and c;
END expr;
```

Una ulteriore osservazione si riferisce al fatto che il file Porte.vhd contiene tutte le porte logiche utilizzate e che per ogni entità dichiarata è necessario indicare le librerie che si vogliono utilizzare.

```
-- porta and a due ingressi
library ieee;
use ieee.std_logic_1164.all;

ENTITY And2 IS
  PORT (a,b: IN std_logic; y: OUT std_logic);
END And2;

ARCHITECTURE tab OF And2 IS
BEGIN
  Elab: Process(a,b)
  BEGIN
    If a='1' and b='1' then
      y <= '1';
    else
      y <= '0';
    end if;
  END process;
END tab;
```

Infine, si noti che ogni `process` presenta la lista delle variabili a cui è sensibile (*sensitivity list*). Si ricorda che questo aspetto è rilevante solo in fase di simulazione e non altera le caratteristiche funzionali descritte. In particolare, la simulazione è *guidata dagli eventi* cioè vengono valutate solamente le variazioni dello stato e, tra uno evento e l'altro, lo stato viene mantenuto inalterato. In assenza di *sensitivity list* (o di un comando di sospensione esplicito – es. wait-) il contenuto del `process` verrebbe valutato continuamente anche in assenza di variazioni di stato rallentando inutilmente la simulazione.

```
-- porta nand a tre ingressi
library ieee;
use ieee.std_logic_1164.all;

ENTITY nand3 IS
  PORT (a,b,c: IN std_logic; y: OUT std_logic);
END nand3;

ARCHITECTURE tab OF nand3 IS
BEGIN
  Elab: process(a,b,c)
  BEGIN
    If a='1' and b='1' and c='1' then
      y <= '0';
    else
      y <= '1';
    end if;
  END process;
END tab;
```

Con un fine puramente esemplificativo, si riporta anche la descrizione in termini di equazioni logiche della porta `nand3`. L'estensione alla altre porte è banale.

```
-- porta nand a tre ingressi
library ieee;
use ieee.std_logic_1164.all;

ENTITY nand3 IS
  PORT (a,b,c: IN std_logic; y: OUT std_logic);
END nand3;

ARCHITECTURE expr OF nand3 IS
BEGIN
  y<= not(a and b and c);
END expr;
```

```
-- porta nand a due ingressi
library ieee;
use ieee.std_logic_1164.all;

ENTITY nand2 IS
  PORT (a,b: IN std_logic; y: OUT std_logic);
END nand2;

ARCHITECTURE tab OF nand2 IS
BEGIN
  Elab: process(a,b)
  BEGIN
    If a='1' and b='1' then
      y <= '0';
    else
      y <= '1';
    end if;
  END process;
END tab;
```

```
-- negatore
library ieee;
use ieee.std_logic_1164.all;

ENTITY Not_g IS
  PORT (u: IN std_logic;
    y: OUT std_logic);
END Not_g;

ARCHITECTURE tab OF Not_g IS
BEGIN
  Elab: process(u)
  BEGIN
    If u='1' then
      y <= '0';
    else
      y <= '1';
    end if;
  END process;
END tab;
```

```
-- porta Or a due ingressi
library ieee;
use ieee.std_logic_1164.all;

ENTITY Or2 IS
  PORT (a,b: IN std_logic; y: OUT std_logic);
END Or2;

ARCHITECTURE tab OF Or2 IS
BEGIN
  Elab: process(a,b)
  BEGIN
    If a='0' and b='0' then
      y <= '0';
    else
      y <= '1';
    end if;
  END process;
END tab;
```

Simulazione

Testbench

Una volta realizzato il file vhd si deve passare alla fase di simulazione per verificare il corretto funzionamento della rete descritta. Per poter testare la rete è però necessario fornire dei valori ai segnali di ingresso, per fare ciò ci serviamo di un TestBench file, utilizziamo in questo caso il file per testare il decoder comportamentale TestB.vhd .

Tipicamente il TestBench lo possiamo vedere come una entità priva di ingressi e uscite in cui si dichiara come componente l'entità che si vuole testare e si mappano su di essa gli opportuni segnali che devono stimolarne gli ingressi. Attraverso un processo di generazione si forniscono quindi i valori a tali segnali.

In questo caso viene utilizzata la libreria standard dell'IEEE, textio, questa ci fornisce le funzioni di scrittura su file (*write, writeline*) necessarie per produrre il file decod1.txt che registrerà le variazioni dei segnali di ingresso ed uscita nei diversi istanti di tempo.

Una novità riguarda l'uso di una procedura, peraltro la sua scrittura è molto simile a quella utilizzata nei comuni linguaggi di programmazione, la cui dichiarazione deve avvenire prima del corpo dell'architettura.

Un'altra particolarità è l'uso del costrutto *wait for n* all'interno del processo. Il processo viene eseguito sequenzialmente, il *wait for* arresta il processo per un tempo n che deve essere espresso in unità temporali; il *wait* finale arresta il processo che altrimenti verrebbe ripetuto indefinitivamente.

```
library IEEE;
use STD.TEXTIO.all;

entity testbench is
end testbench;

architecture FUNCTIONAL_T of testbench is

  file RESULTS: TEXT open WRITE_MODE is "decod1.txt";

--procedura che scrive sul file decod1.txt i parametri input e output che le vengono
--passati al momento della sua chiamata, nel corpo dell'architettura
procedure WRITE_RESULTS(input: std_logic_vector(0 TO 2);
                        output: std_logic_vector(0 TO 7)) is
  variable L_OUT:LINE;
  begin
    write(l_out, now, right, 15, ns);
    write(l_out, input, right, 7);
    write(l_out, output, right, 14);
    writeline(results,l_out);
end;

component decoder
  port (input: IN std_logic_vector(0 TO 2);
        output: OUT std_logic_vector(0 TO 7));
END component;

-- per tutti i componenti decoder si utilizza l'entità decoder nella sua
-- architettura comportamentale. Questo è necessario in quanto nel file decodBH.vhd
-- differenti per la stessa entità
For all: decoder use entity decoder(TabellaVerita);

-- dichiarazione dei segnali su cui si vanno a mappare ingressi
-- e uscite del componente decoder
signal X: std_logic_vector(0 TO 2):="000";
signal Y: std_logic_vector(0 TO 7);

Begin
  --istanziazione del componente sotto test
  UUT: decoder port map(input=>X, output=>Y);

  --processo di generazione dei valori di ingresso
  GEN_S: process
    begin
      X <= B"000";
        wait for 10 ns;
      X <= B"001";
        wait for 10 ns;
      X <= B"010";
        wait for 10 ns;
      X <= B"011";
        wait for 10 ns;
      X <= B"100";
        wait for 10 ns;
      X <= B"101";
        wait for 10 ns;
      X <= B"110";
        wait for 10 ns;
      X <= B"111";
        wait for 10 ns;
      wait;
  end process;
  WRITE_TO_FILE: WRITE_RESULTS(X,Y);
End FUNCTIONAL_T;
```

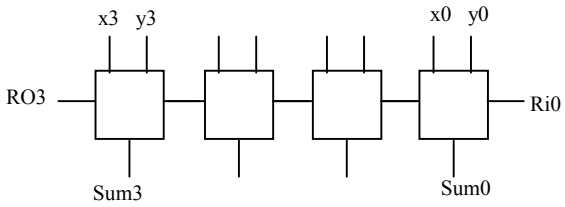
```
configuration FUNCTIONAL_CFG of testbench is
  for FUNCTIONAL_T
  end for;
end FUNCTIONAL_CFG;
```

L'analogo file TestbST.vhd effettua il test del decoder strutturale.

Ricapitolando ciò che si è visto fino ad ora, possiamo dire che:

- In VHDL per descrivere un'entità bisogna darne un modello
- Tale modello può a sua volta usare altre entità
- Ad una collezione di modelli di entità si riserva il nome di libreria
- L' entità è anche la porzione minima di codice VHDL eseguibile autonomamente
- Un modello VHDL che contiene più entità può essere suddiviso su più librerie
- Ogni libreria può contenere una o più entità
- La libreria VHDL è il minimo frammento di codice compilabile separatamente

Possiamo vedere un nuovo esempio che realizza un sommatore parallelo a 4 bit, in versione strutturale, attraverso l'utilizzo dell'entità elementare Full_adder realizzata sia in versione strutturale che comportamentale.



I file utilizzati sono: fadd.vhd, 4bitsum.vhd, testb.vhd

```
-- Esempio 2 Realizzazione di un Full Adder
library ieee;
use ieee.std_logic_1164.all;

ENTITY full_adder IS
  GENERIC (T: time:=1ns);
  PORT (x,y: IN std_logic:='0'; ri: IN std_logic:='0';
    Sum,RO: OUT std_logic);
END full_adder;


-- Versione comportamentale con ritardo
-- nella propagazione del bit di riporto
ARCHITECTURE behavioral OF full_adder IS
BEGIN
    Sum<= (x and y and ri) or (x and not y and not ri) or
   (not x and y and not ri) or (not x and not y and ri);
    RO<= (ri and y) or (x and y) or (ri and x) after T;
End behavioral;
```

Si può notare che nell'intestazione dell'entità Full_adder si dichiara una costante di tempo che viene poi utilizzata nella sua descrizione comportamentale per introdurre un ritardo di propagazione del bit di riporto in uscita. Il segnale R0 viene aggiornato dopo il tempo T utilizzando la sintassi **`S1 <= S0 after T`**.

In questo esempio ancora una volta si è data una descrizione del modello utilizzando le sue espressioni logiche.

```
-- Versione strutturale senza
-- ritardo di propagazione del riporto
ARCHITECTURE Structural OF full_adder IS
  COMPONENT And3
    PORT (a,b,c: IN std_logic; y: OUT std_logic);
  END COMPONENT;

  COMPONENT or2
    PORT (a,b: IN std_logic; y: OUT std_logic);
  END COMPONENT;

  COMPONENT Not_g
    PORT (u: IN std_logic; y: OUT std_logic);
  END COMPONENT;

  COMPONENT And2
    PORT(a,b: IN std_logic; y: OUT std_logic);
  END COMPONENT;

  SIGNAL Nx,Ny,Nri,U1,U2,U3,U4,U5,U6,U7,U8,U9,U10: std_logic:='0';

BEGIN
  notX: Not_g PORT MAP (u=>x, y=>Nx);
  notY: Not_g PORT MAP (u=>y, y=>Ny);
  notRI: Not_g PORT MAP (u=>ri, y=>Nri);

  andG0: And3 PORT MAP (a=>x, b=>y, c=>ri, y=>u1);
  andG1: And3 PORT MAP (a=>x, b=>Ny, c=>Nri, y=>u2);
  andG2: And3 PORT MAP (a=>Nx, b=>y, c=>Nri, y=>u3);
  andG3: And3 PORT MAP (a=>Nx, b=>Ny, c=>ri, y=>u4);

  orG0:  or2 PORT MAP (a=>U1, b=>U2, y=>U5);
  orG1:  or2 PORT MAP (a=>U3, b=>U4, y=>U6);
  orG2:  or2 PORT MAP (a=>U5, b=>U6, y=>Sum);

  and4: And2 PORT MAP (a=>ri, b=>y, y=>u7);
  and5: And2 PORT MAP (a=>x, b=>y, y=>u8);
  and6: And2 PORT MAP (a=>ri, b=>x, y=>u9);

  orG3:  or2 PORT MAP (a=>U7, b=>U8, y=>U10);
  orG4:  or2 PORT MAP (a=>U9, b=>U10, y=>RO);
End Structural;
```

Utilizzando il modello di full_adder realizziamo ora il sommatore a 4 bit:

```
-- Esempio 2
-- Sommatore a 4 bit parallelo strutturale
-- realizzato con Full Adder
-- file 4bitsum.vhd
-- ********************************************
library ieee;
use ieee.std_logic_1164.all;

ENTITY somma4 IS
  PORT (x1,y1: IN std_logic_vector(3 DOWNTO 0):="0000";
    sum4: OUT std_logic_vector(3 DOWNTO 0); Rip: OUT std_logic);
END somma4;

ARCHITECTURE Struct OF somma4 IS
COMPONENT full_adder
  PORT (x,y,ri: IN std_logic:='0';
  Sum,RO: OUT std_logic);
END COMPONENT;

for all: full_adder use entity full_adder(behavioral);

SIGNAL R1,R2,R3,z: std_logic:='0';
BEGIN
 add1: full_adder PORT MAP(x=>x1(0), y=>y1(0), ri=>Z, Sum=>Sum4(0), RO=>R1);
 add2: full_adder PORT MAP(x=>x1(1), y=>y1(1), ri=>R1, Sum=>Sum4(1), RO=>R2);
 add3: full_adder PORT MAP(x=>x1(2), y=>y1(2), ri=>R2, Sum=>Sum4(2), RO=>R3);
 add4: full_adder PORT MAP(x=>x1(3), y=>y1(3), ri=>R3, Sum=>Sum4(3), RO=>Rip);
END Struct;
```

E' possibile notare la dichiarazione di un array di tipo std_logic_vector in un modo leggermente diverso da quello visto precedentemente utilizzando il *DOWNTO* anziché il *TO*. Si deve prestare attenzione all'uso di questi due sistemi in quanto permettono di indicare quale sarà il bit più significativo.

Essendo stato descritto il full_adder in due architetture diverse si deve ancora specificare quale di queste deve essere utilizzata all'interno del sommatore. In particolare in questo esempio abbiamo la possibilità di vedere il comportamento diverso del sommatore se si considerano o meno i ritardi di propagazione del bit di riporto. Se si vuole effettuare il test senza ritardi si deve utilizzare l'architettura *structural*, in caso contrario (come qui sopra) si utilizza l'architettura *behavioral*

Ancora una volta per poter simulare il comportamento della rete abbiamo bisogno di un testbench file testb.vhd.

```
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.std_logic_TEXTIO.all;
use STD.TEXTIO.all;

entity testbench is
end testbench;

architecture FUNCTIONAL_T of testbench is

file RESULTS: TEXT open WRITE_MODE is "somma4.txt";

procedure WRITE_RESULTS(input1,input2,output: bit_vector(3 DOWNTO 0);Rip:bit) is
variable L_OUT : LINE;
begin
  write(l_out, now, right, 15, ns);
  write(l_out, input1, right, 7);
  write(l_out, input2, right, 7);
  write(l_out, output, right, 7);
  write(l_out, Rip, right, 4);
  writeline(results,l_out);
end;

component somma4
  PORT (x1,y1: IN bit_vector(3 DOWNTO 0);
 sum4: OUT bit_vector(3 DOWNTO 0); Rip: OUT bit);
END component;

signal A,B: bit_vector(3 DOWNTO 0):="0000";
signal Y:bit_vector(3 DOWNTO 0);
signal Carry: bit;

Begin
 UUT: somma4 port map(x1=>A, y1=>B, sum4=>Y, Rip=>Carry);
 GEN_S0: process
    begin
      A(0)<='0';
      B(1)<='0';
      wait for 5ns;
      A(0)<='1';
      B(1)<='1';
      wait for 5ns;
    end process;
 GEN_S1: process
    begin
      A(1)<='0';
      B(0)<='0';
      wait for 10ns;
      A(1)<='1';
      B(0)<='1';
      wait for 10ns;
  end process;
 GEN_S2: process
    begin
      A(2)<='0';
      B(3)<='0';
      wait for 20ns;
      A(2)<='1';
      B(3)<='1';
      wait for 20ns;
    end process;
 GEN_S3: process
    begin
      A(3)<='0';
```

```
      B(2)<='0';
      wait for 40ns;
      A(3)<='1';
      B(2)<='1';
      wait for 40ns;
   end process;
   WRITE_TO_FILE: WRITE_RESULTS(A,B,Y,Carry);
End FUNCTIONAL_T;

configuration FUNCTIONAL_CFG of testbench is
for FUNCTIONAL_T
end for;
end FUNCTIONAL_CFG;
```

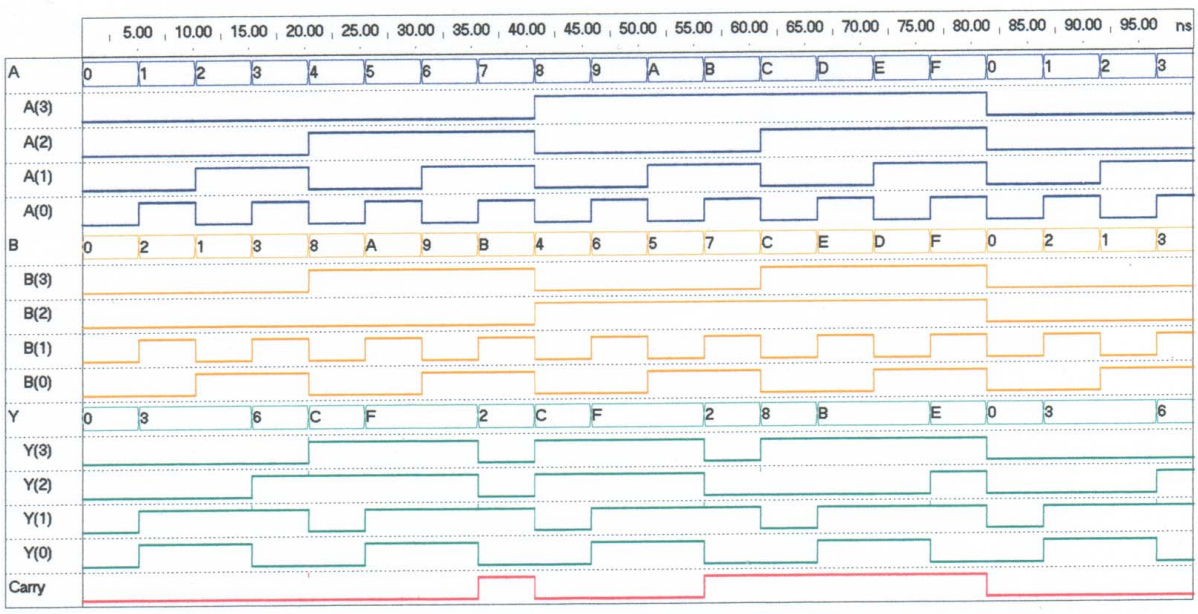
Si può notare che in questo caso per fornire in ingresso un ampio numero di valori ai due addendi si è pensato di far variare ciclicamente (proprio come un segnale di clock) i segnali di ingresso con temporizzazioni diverse da bit a bit, questo è stato fatto realizzando quattro processi concorrenti. A differenza del precedente esempio manca l'istruzione *wait* alla fine dei processi, in fase di simulazione quindi si dovrà procedere per passi di tempo finiti altrimenti questa continuerà indefinitivamente.

Simulazione

Nel file di test è stata prevista la scrittura di un file di testo, somma4.txt, che ci fornisce le variazioni di tutti i segnali in passi discreti di tempo.

Un altro sistema più leggibile per analizzare la simulazione, presente spesso negli ambienti di lavoro, è quello di poter visualizzare l'andamento grafico, tramite forme d'onda, dei segnali.

Si da di seguito un esempio dei due sistemi, applicati al sommatore a 4 ingressi senza ritardo di propagazione



Time	A	B	Y	Crry
0 ns	0000	0000	0000	0
5 ns	0001	0010	0000	0
5 ns	0001	0010	0011	0
10 ns	0010	0001	0011	0
15 ns	0011	0011	0011	0
15 ns	0011	0011	0000	0
15 ns	0011	0011	0110	0
20 ns	0100	1000	0110	0
20 ns	0100	1000	1010	0
20 ns	0100	1000	0100	1
20 ns	0100	1000	1100	0
25 ns	0101	1010	1100	0
25 ns	0101	1010	1111	0
30 ns	0110	1001	1111	0
35 ns	0111	1011	1111	0
35 ns	0111	1011	1100	0
35 ns	0111	1011	1010	0
35 ns	0111	1011	0010	1
40 ns	1000	0100	0010	1

```
40 ns   1000   0100   0100   1
40 ns   1000   0100   1100   0
45 ns   1001   0110   1100   0
45 ns   1001   0110   1111   0
50 ns   1010   0101   1111   0
55 ns   1011   0111   1111   0
55 ns   1011   0111   1100   0
55 ns   1011   0111   1010   0
55 ns   1011   0111   0010   1
60 ns   1100   1100   0010   1
60 ns   1100   1100   1110   1
60 ns   1100   1100   1000   1
65 ns   1101   1110   1000   1
65 ns   1101   1110   1011   1
70 ns   1110   1101   1011   1
75 ns   1111   1111   1011   1
75 ns   1111   1111   1000   1
75 ns   1111   1111   1110   1
80 ns   0000   0000   1110   1
80 ns   0000   0000   1110   0
80 ns   0000   0000   0000   0
85 ns   0001   0010   0000   0
85 ns   0001   0010   0011   0
90 ns   0010   0001   0011   0
95 ns   0011   0011   0011   0
95 ns   0011   0011   0000   0
95 ns   0011   0011   0110   0
100 ns  0100   1000   0110   0
100 ns  0100   1000   1010   0
100 ns  0100   1000   0100   1
100 ns  0100   1000   1100   0
```

Riassunto

Nella sezione appena conclusa abbiamo visto l'uso di alcuni costrutti fondamentali del VHDL, utilizzati per descrivere reti combinatorie completamente specificate.

Abbiamo scoperto che con questo linguaggio, proprio come siamo abituati a fare manualmente su carta, è possibile rappresentare un dispositivo in diversi modi, ad esempio utilizzando la sua tavola della verità, le sue espressioni logiche oppure il suo schema logico.

In particolare si è posta l'attenzione sulla differenza tra descrizioni di modelli di tipo strutturale, comportamentale e misto.

Qualche parola è stata spesa a spiegare come, una volta terminato il modello di un dispositivo, sia buona regola realizzare un file di test per sottoporre a simulazione il lavoro svolto.

Quick Reference dei costrutti utilizzati fin qui:

Descrizione delle Entità

```
Entity entity_identifier is
   [generic (generic_list) ;]
   [port (generic_list) ;]
   [entity_decl ]
Begin
[entity_stmt ]
End entity_identifier;

Architecture arch_identifier of entity_name is
[declaration]
Begin
[cuncurrent_statement]
End [architecture] arch_identifier;
```

Istruzioni Sequenziali

If condition **then**
{sequential_statement}
[{**elsif** condition **then**
{sequential_statement}}]
[**else**
{sequential_statement}]
end If ;

Case expr **is**
{**when** choice [{**|** choice}] **=>** {sequential_statement}}
end Case;

Wait [**On** signal] [**Until** condition] [**For** simple_expr];

Istruzioni Concorrenti

[LABEL**:**] **Process** [**(** {signal_list**,**} **)**]
[{declaration}]
Begin
[{sequential_statement}]
End process;

Signal_name <= {value [**after** time] [**when** condition **else**]}
value [**after** time];

Dichiarazioni

Component identifier [***is***]
[**generic (** {ID **:** TYPEID [:= expr];} **);**]
[**port (**{ID **: in** | **out** | **inout** TYPEID [**:=** expr];**});**]
end Component;

Signal name_list: subtype [:= expr]

For comp_label | **All** :comp_name **Use Entity** ent_name(arch_name);

Sottoprogrammi

Procedure proc_name [(**Constant** | **Signal** | **Variable** name_list: [**In** | **Out** | **InOut**] subtype)] **Is** sub-prog_decl
Begin
{sequential statement}
End [proc_name];

Esercizi proposti

Si propongono una serie di esercizi, alcuni dei quali con relativa soluzione, altri da risolvere:

Esercizio 1

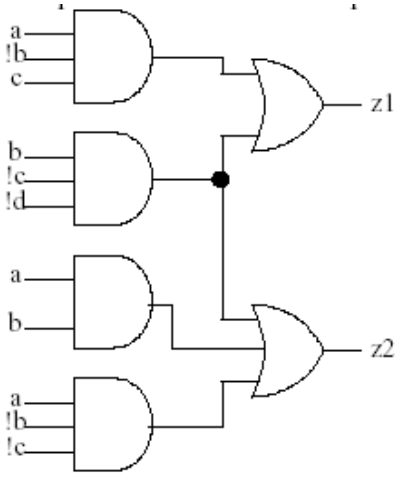
Data la seguente funzione combinatoria completamente specificata,
F(a,b,c,d): ONset(a b c d; a !c; !a !b !c !d; !a b !c; !a·!b·!c d) DCset();
Scrivere il codice vhdl di due architetture comportamentali: la prima sulla base della tavola della verità e la seconda utilizzando le espressioni logiche della funzione.

Esercizio 2

Data la seguente espressione logica: Z=(*not* a *and not* b) *or* (a *and* b *and not* c *and* d)

Darne una descrizione comportamentale ed una strutturale (in tal caso realizzare un file contenente la descrizione delle porte logiche utilizzate)
[Soluzione proposta Esercizio2.vhd, porte_logiche.vhd]

Esercizio 3
Data la seguente funzione combinatoria con 4 ingressi descritta dalla seguente rappresentazione strutturale, in cui il ! indica le variabile complementate (!a = NOT a), realizzarne in VHDL la descrizione strutturale, utilizzando il file di libreria delle porte logiche dell'esercizio precedente aggiungendo la descrizione delle porte mancanti.



Esercizio 4
Data la seguente tabella della verità completamente specificata:

abc	f
000	0
001	1
010	1
011	0
100	1
101	0
110	0
111	1

realizzare due architetture in VHDL; la prima che utilizzi istruzioni sequenziali e la seconda che utilizzi solo istruzioni concorrenti.

Esercizio 5
Realizzare il modello in VHDL della seguente rete combinatoria gerarchica:

Φ= (not a and c) or (a and not c) or (a and not b)

Φde	Z
000	0
001	1
010	1
011	0
100	1
101	0
110	1
111	1

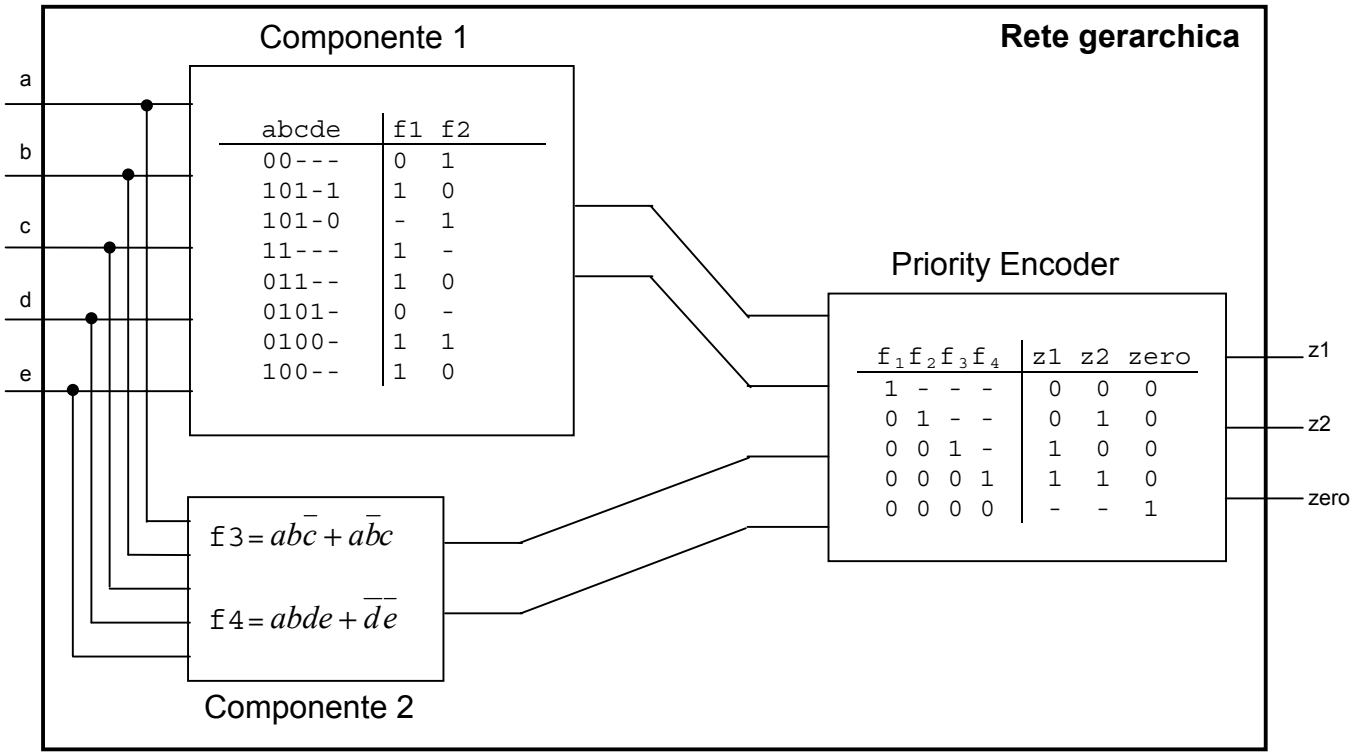
Si richiede di realizzare il modello della funzione Φ utilizzando le sue espressioni logiche, quello della funzione Z utilizzando la sua tavola della verità ed infine quello della rete complessiva in modo strutturale.

[Soluzione proposta Rete1_Es5.vhd, Rete2_Es5.vhd, ReteG_Es5.vhd]

Reti Combinatorie non completamente specificate

Quello che vedremo ora è come sia possibile, grazie alla libreria `std_logic` realizzare un modello per una rete non completamente specificata.

Supponiamo di voler modellare la seguente rete combinatoria, scomposta gerarchicamente in tre componenti e così definita:



Per descriverne il comportamento attraverso un'architettura strutturale utilizziamo tre file, uno per ogni entità `C1.vhd`, `C2.vhd`, `P_encoder.vhd` più un ultimo file `reteg.vhd` che ci dirà come questi sono collegati per realizzare la rete gerarchica.

Il componente 1 deve essere descritto attraverso una tabella non completamente specificata, per fare questo utilizziamo il costrutto *If ... Then ... Else* per valutare tutte le possibili configurazioni d'ingresso e quindi per assegnare ai segnali d'uscita gli opportuni valori.

E' importante notare che in questo caso siamo obbligati a dichiarare i segnali di tipo std_logic per poter assegnare in uscita i valori '–' don't care e 'Z' alta impedenza. Si può notare come oltre al singolo dato sia possibile dichiarare anche i vettori di tipo std_logic.

```
--Esempio3
--Descrizione di una tabella delle implicazioni non
--completamente specificata
--file C1.vhd
Library ieee;
Use ieee.std_logic_1164.all;

Entity componente_A is
  port(X:in std_logic_vector(4 DOWNTO 0):="00000";
     F1,F2: out std_logic);
End componente_A;
Architecture dataflow of componente_A is
Begin
cod: process(X)
  begin
  if X(4)='0' and X(3)='0' then
    F1<='0';
    F2<='1';
  elsif X(4)='1' and X(3)='0' and X(2)='1' and X(0)='1' then
    F1<='1';
    F2<='0';
  elsif X(4)='1' and x(3)='0' and X(2)='1' and X(0)='0' then
    F1<='-';
    F2<='1';
  elsif X(4)='1' and X(3)='1' then
    F1<='1';
    F2<='-';
  elsif X(4)='0' and X(3)='1' and X(2)='1' then
    F1<='1';
    F2<='0';
  elsif X(4)='0' and X(3)='1' and X(2)='0' and X(1)='1' then
    F1<='0';
    F2<='-';
  elsif X(4)='0' and X(3)='1' and X(2)='0' and X(1)='0' then
    F1<='1';
    F2<='1';
  elsif X(4)='1' and X(3)='0' and X(2)='0' then
    F1<='1';
    F2<='0';
  else F1<='Z';
       F2<='Z';
  End if;
End process;
End dataflow;
```

Per la descrizione del componente 2 molto semplicemente si realizza un'architettura comportamentale in cui si specificano le espressioni logiche che ne identificano il funzionamento. Invece di dichiarare un segnale per ogni ingresso si è preferito utilizzare un vettore di 5 celle; le espressioni logiche quindi effettuano il confronto tra i valori delle singole celle del vettore.

```
--Descrizione di un componente tramite
--espressioni logiche
--file C2.vhd
Library ieee;
Use ieee.std_logic_1164.all;

Entity componente_B is
  port(X:in std_logic_vector(4 DOWNTO 0):="00000";
     F3,F4: out std_logic);
End componente_B;

Architecture behavior of componente_B is
Begin
  F3<= (X(4) and X(3) and not X(2)) or (X(4) and not X(3) and X(2));
  F4<= (X(4) and X(3) and X(1) and X(0)) or (not X(1) and not X(0));
End behavior;
```

La descrizione del priority encoder avviene nello stesso modo utilizzato per il componente1.

```
--Esempio 3
--Descrizione di un priority encoder
--tramite la sua tabella delle implicazioni
--file P_encoder.vhd
Library ieee;
Use ieee.std_logic_1164.all;

Entity P_encoder is
  port(F:in std_logic_vector(3 DOWNTO 0):="0000";
       Z: out std_logic_vector(2 DOWNTO 0));
End P_encoder;

Architecture behavior of P_encoder is
Begin
codifica: process(F)
     begin
     If F(3)='1' then Z<= "000";
     elsif (F(3)='0' and F(2)='1') then Z<= "010";
     elsif (F(3)='0' and F(2)='0' and F(1)='1')then Z<= "100";
     elsif (F(3)='0' and F(2)='0' and F(1)='0' and F(0)='1') then Z<= "110";
     elsif (F(3)='0' and F(2)='0' and F(1)='0' and F(0)='0') then Z<= "--1";
     end if;
  end process;
End behavior;
```

L'ultimo file `reteg.vhd` descrive esattamente la gerarchia della rete complessiva:

- si definisce l'entità `Rete_gerarchica` in cui si specificano i segnali di interfaccia con l'esterno
- si procede poi con l'architettura strutturale della rete definendo i componenti che la realizzano e i segnali che servono come canali di comunicazione tra questi
- infine nel corpo dell'architettura si creano le istanze di questi componenti mappando per ognuno ingressi ed uscite

```
-- Esempio 3
-- Scomposizione gerarchica di una rete combinatoria
-- in 3 componenti: C1
--        C2
--      Priority Encoder
Library ieee;
Use ieee.std_logic_1164.all;
Entity Rete_gerarchica is
  port (a, b, c, d, e: in std_logic;
    Z1, Z2, Zero: out std_logic);
End Rete_gerarchica;

Architecture structural of Rete_gerarchica is
  Component componente_A
  port(X:in std_logic_vector(4 DOWNTO 0):="00000";
     F1,F2: out std_logic);
  End component;

  Component componente_B
  port(X:in std_logic_vector(4 DOWNTO 0):="00000";
     F3,F4: out std_logic);
  End component;

  Component P_encoder
  port(F:in std_logic_vector(3 DOWNTO 0);
     Z: out std_logic_vector(2 DOWNTO 0));
  END component;

  Signal S0, S1, S2, S3: std_logic;

Begin
    comp1:componente_A port map(X(4)=>a, X(3)=>b, X(2)=>c, X(1)=>d,
                  X(0)=>e, F1=>S0, F2=>S1);
    comp2:componente_B port map(X(4)=>a, X(3)=>b, X(2)=>c, X(1)=>d,
          X(0)=>e, F3=>S2, F4=>S3);
  comp3:P_encoder port map(F(3)=>S0, F(2)=>S1, F(1)=>S2, F(0)=>S3,
             Z(2)=>Z1, Z(1)=>Z2, Z(0)=>Zero);
End structural;
```

Alla fine si crea il file di test testrg.vhd attraverso il quale si forniscono i valori agli ingressi e si procede alla simulazione del modello realizzato.

```
--esempio 3
--File testrg.vhd
--TestBench file per lo stimolo della rete
--gerarchica e la produzione di un file di testo rete.txt
--che contiene le variazioni dei segnali di ingresso ed uscita
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_TEXTIO.all;
use STD.TEXTIO.all;

entity testbench is
end testbench;

architecture FUNCTIONAL_T of testbench is

file RESULTS: TEXT open WRITE_MODE is "rete.txt";

procedure WRITE_RESULTS(x0,x1,x2,x3,x4,y0,y1,y2: std_logic) is
  variable L_OUT : LINE;
  begin
     write(l_out, now, right, 15, ns);
      write(l_out, x0, right, 3);
      write(l_out, x1, right, 1);
      write(l_out, x2, right, 1);
      write(l_out, x3, right, 1);
      write(l_out, x4, right, 1);
      write(l_out, y0, right, 4);
      write(l_out, y1, right, 1);
      write(l_out, y2, right, 1);
      writeline(results,l_out);
end;

Component Rete_gerarchica
  port (a, b, c, d, e: in std_logic;
      Z1, Z2, Zero: out std_logic);
End component;

signal U: std_logic_vector(4 DOWNTO 0):="00000";
signal Y: std_logic_vector(2 DOWNTO 0);



Begin
  UUT: Rete_gerarchica port map(a=>U(4), b=>U(3), c=>U(2), d=>U(1),
          e=>U(0), Z1=>Y(2), Z2=>Y(1), Zero=>Y(0));
  GEN_S: process
      begin
      U <= "10000";
      wait for 10 ns;

      U <= "00111";
      wait for 10 ns;

      U <= "01011";
      wait for 10 ns;

      U <= "11011";
      wait for 10 ns;

      U <= "11111";
      wait for 10 ns;
```

```
      U <= "10110";
      wait for 10 ns;

      U <= "10111";
      wait for 10 ns;

      U <= "00000";
      wait for 10 ns;
    wait;
    end process;
    WRITE_TO_FILE: WRITE_RESULTS(U(4),U(3),U(2),U(1),U(0),Y(2),Y(1),Y(0));
End FUNCTIONAL_T;

configuration FUNCTIONAL_CFG of testbench is
  for FUNCTIONAL_T
  end for;
end FUNCTIONAL_CFG;
```

Della stessa rete è sempre possibile dare una rappresentazione "per processi", anche se questa fa perdere l'idea della gerarchia.

Se si legge il listato del modello, nel file ReteBH.vhd, ci si accorge che questo rimane nei contenuti sostanzialmente lo stesso del precedente se solo si uniscono i processi dei singoli componenti utilizzati precedentemente in un unico file. In questo caso invece di creare un istanza di un componente abbiamo tre processi che vengono eseguiti in modo concorrente.

```
-- Esempio 3
-- Scomposizione gerarchica di una rete combinatoria
-- Realizzazione per processi
Library ieee;
Use ieee.std_logic_1164.all;

Entity Rete_gerarchicaP is
  port (a, b, c, d, e: in std_logic;
      Z1, Z2, Zero: out std_logic);
End Rete_gerarchicaP;

Architecture behavioral of Rete_gerarchicaP is
  Signal F1, F2, F3, F4: std_logic;
  Begin

  --processo che descrive il comportamento del componente 1
  C1: process(a,b,c,d,e)
      Begin
       if a='0' and b='0' then
        F1<='0';
        F2<='1';
      elsif a='1' and b='0' and c='1' and e='1' then
              F1<='1';
            F2<='0';
      elsif a='1' and b='0' and c='1' and e='0' then
        F1<='-';
        F2<='1';
      elsif a='1' and b='1' then
        F1<='1';
        F2<='-';
      elsif a='0' and b='1' and c='1' then
        F1<='1';
        F2<='0';
      elsif a='0' and b='1' and c='0' and d='1' then
          F1<='0';
        F2<='-';
      elsif a='0' and b='1' and c='0' and d='0' then
        F1<='1';
        F2<='1';
      elsif a='1' and a='0' and c='0' then
        F1<='1';
        F2<='0';
      else F1<='Z';
           F2<='Z';
      End if;
    End process;

  --processo che descrive il comportamento del componente 2
  C2: process(a,b,c,d,e)
      begin
        F3<= (a and b and not c) or (a and not b and c);
        F4<= (a and b and d and e) or (not d and not e);
       end process;

--processo che descrive il comportamento del Priority Encoder
  P_encod: process(F1,F2,F3,F4)
    begin
       If F1='1' then
         Z1<= '0';
         Z2<= '0';
         zero<='0';
       elsif (F1='0' and F2='1') then
```

```
        Z1<= '0';
        Z2<= '1';
        zero<='0';
      elsif (F1='0' and F2='0' and F3='1')then
        Z1<= '1';
        Z2<= '0';
        zero<='0';
      elsif (F1='0' and F2='0' and F3='0' and F4='1') then
        Z1<= '1';
        Z2<= '1';
        zero<='0';
      elsif (F1='0' and F2='0' and F3='0' and F4='0') then
        Z1<= '-';
        Z2<= '-';
        zero<='1';
      end if;
   End process;
End behavioral;
```

Essendo che l'entità rete_gerarchicaP ha la stessa interfaccia del modello strutturale e cambia solo nella parte finale del nome, per effettuarne il test basta correggere il nome dell'entità da simulare nel TestBench file testrg.vhd.

Riassunto

Con l'uso di un esempio di rete scomposta gerarchicamente abbiamo visto come in VHDL sia possibile realizzare modelli di reti non completamente specificate utilizzando un tipo di dato opportuno, lo standard logic. Questo tipo permette di gestire un numero di valori (U- Uninitialized, '-' don't care, Z High Impedante ecc.) più ampio rispetto ai classici '0' e '1' del tipo bit. Le specifiche di tutte le operazioni possibili su di esso e i risultati che si ottengono sono contenute nella libreria standard dell'IEEE STDLOGIC.VHD contenuta nei pacchetti di sviluppo per VHDL.

Nell'esempio realizzato si è visto come sia possibile realizzare una rete strutturata in modo gerarchico; ciò è stato fatto utilizzando metodi descrittivi differenti per le sottoreti.

Esercizi proposti

Esercizio 6

Data la seguente funzione combinatoria non completamente specificata, produrre il codice VHDL che la descrive in modo comportamentale sulla base della tavola della verità ottenuta.

F(a,b,c,d)=|f1;f2; f3|=
F(a,b,c,d)=|ONset1(5;7;14)DCset1(15); ONset2(1;5;12;15)DCset2(7;13); ONSet3(6;11)DCset2(5;7;15)|;

Esercizio 7

Data la seguente funzione combinatoria non completamente specificata, produrre il codice VHDL che la descrive in modo comportamentale sulla base della tavola della verità ottenuta.

F(a,b,c,d)=|f1;f2|=|ONset1(3;7;12) DCset1(0;14,15);ONset2(12;14;15) DCset2(0;3;7)|;

Esercizio 8

Data la seguente tavole della verità non completamente specificata tradurla in codice VHDL utilizzando i costrutti noti.

a	b	c	d	F_1	F_2	F_3
0	0	-	1	1	-	1
0	1	-	-	0	0	1
1	0	0	0	0	-	-

1	0	0	1	-	-	1
1	0	1	1	-	1	-
1	1	0	0	1	1	1
1	1	0	1	0	0	0
1	1	1	0	-	1	0
1	1	1	1	0	1	-

[Soluzione proposta *Esercizio8.vhd*]

Macchine Sequenziali

Vediamo ora come sia possibile descrivere delle macchine sequenziali in VHDL. In modo del tutto simile a quanto visto in precedenza non c'è un'unica via predefinita per fare ciò, ma è possibile descrivere il modello evidenziando gli aspetti strutturali o comportamentali.

Iniziamo con un semplice esempio:

Si vuole progettare una rete sequenziale sincrona con due ingressi *x* e *R*, e un'uscita *Z*, che deve comportarsi come un ritardo programmabile. Se R=0, deve essere Z(t)=x(t-1); se R=1, deve essere Z(t)=x(t-2). Si completi il progetto utilizzando bistabili JKT.

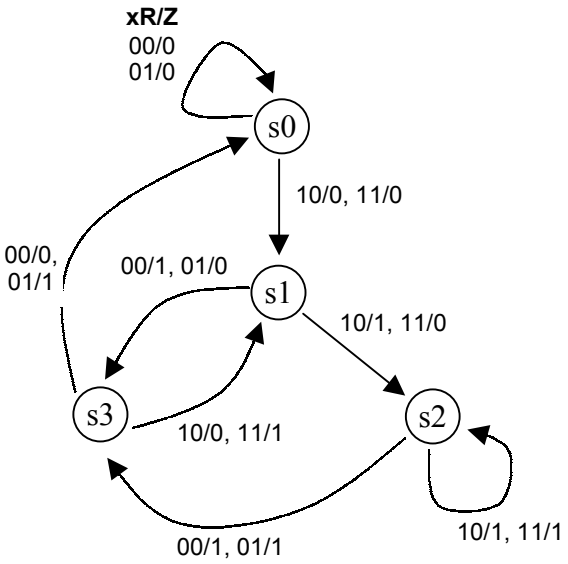
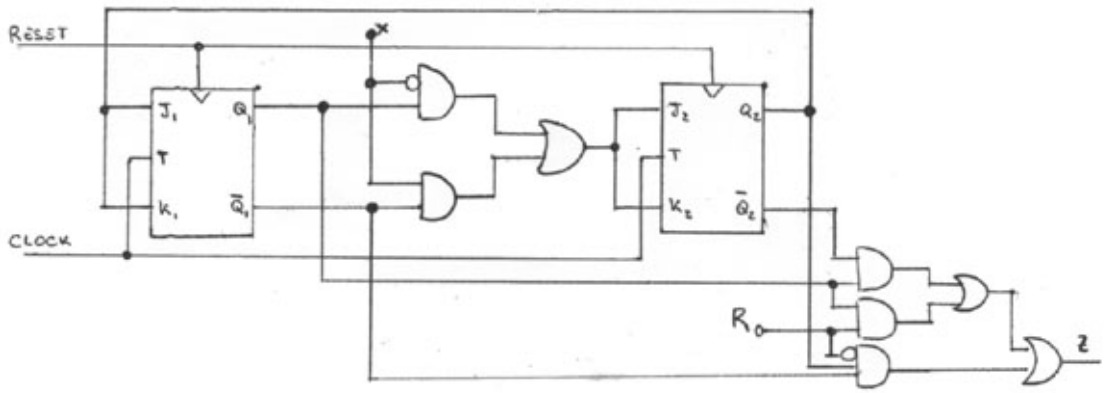


Tabella degli stati

S \ xR	00	01	11	10
s0	s0, 0	s0, 0	s1, 0	s1, 0
s1	s3, 1	s3, 0	s2, 0	s2, 1
s2	s3, 1	s3, 1	s2, 1	s2, 1
s3	s0, 0	s0, 1	s1, 1	s1, 0



Per questo esempio decidiamo di effettuare due descrizioni diverse: la prima si di tipo comportamentale che descrive il funzionamento della macchina utilizzando il diagramma degli stati; la seconda di tipo strutturale che indica i collegamenti tra i componenti utilizzati nella rete logica. Entrambe sono contenute nel file Mealy.vhd.

```
-- Realizzazione di una rete sequenziale sincrona con due ingressi
-- e un'uscita, che si comporti come un ritardo programmabile
-- Realizzata come macchina di Mealy
-- File mealy.vhd
-----------------------------------
Library ieee;
Use ieee.std_logic_1164.all;

entity Ritardo is
  port(X,R, clock, zero: in std_logic;
         Z: out std_logic);
end Ritardo;

-- descrizione comportamentale
architecture behavioral of Ritardo is
type stato is (s0, s1, s2, s3);
  signal stato_corrente, stato_prox: stato;
begin
  elabora: process(X,R,stato_corrente)
      begin
       case stato_corrente is
         when s0 =>
            if ((X='0'and R='0') or (X='0' and R='1')) then
                        Z <= '0';
                        stato_prox <= s0;
              elsif ((X='1' and R='0') or (X='1' and R='1')) then
                        Z <= '0';
                        stato_prox <= s1;
               end if;

            when s1 =>
           if (X='0'and R='0') then
                        Z <= '1';
                        stato_prox <= s3;
                  elsif (X='0' and R='1') then
                        Z<= '0';
                        stato_prox<= s3;
             elsif (X='1' and R='0') then
                  Z<= '1';
                  stato_prox <= s2;
             elsif (X='1' and R='1') then
                        Z <= '0';
                        stato_prox <= s2;
               end if;

            when s2 =>
           if ((X='0'and R='0') or (X='0' and R='1')) then
                        Z <= '1';
                        stato_prox <= s3;
                  elsif ((X='1' and R='0') or (X='1' and R='1')) then
                  Z<= '1';
                  stato_prox <= s2;
       end if;

       when s3 =>
          if (X='0'and R='0') then
                        Z <= '0';
                        stato_prox <= s0;
                  elsif (X='0' and R='1') then
                        Z<= '1';
                        stato_prox<= s0;
             elsif (X='1' and R='0') then
```

```
                Z<= '0';
                stato_prox <= s1;
           elsif (X='1' and R='1') then
                      Z <= '1';
                      stato_prox <= s1;
            end if;
       end case;
     end process;

controlla: process
   begin
     wait until clock'event and clock= '1';
     stato_corrente<= stato_prox;
   end process;
end behavioral;
```

Per l'identificazione degli stati si è impostato con il costrutto **type** il tipo *stato* a cui appartengono i segnali *stato_corrente* e *stato_prox*.

Per la descrizione della macchina si sono utilizzati due processi concorrenti; nel primo (*elabora*) si analizzano gli ingressi e lo stato corrente per decidere le transizioni di stato e di uscita, nel secondo (*controlla*) si effettuano i passaggi di stato sul fronte di salita del clock.

```
-- descrizione strutturale
architecture structural of ritardo is

component JK_FF
port (J, K, T, Reset: in std_logic;
    Q, QN: out std_logic);
end component;

component Or2
  PORT (a,b: IN std_logic; y: OUT std_logic);
end component;

component Not_g
  PORT (u: IN std_logic;
    y: OUT std_logic);
end component;

component And2
  PORT (a,b: IN std_logic; y: OUT std_logic);
end component;

component And3
  PORT (a,b,c: IN std_logic;
    y: OUT std_logic);
end component;

signal s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13: std_logic;
signal Res: std_logic :='0';

Begin
  notA: not_g port map(X, s11);
  notB: not_g port map(R, s12);
  andA: and2      port map(s11, s9, s1);
  andB: and2      port map(X, s10, s2);
  andC: and2      port map(s5, s9, s6);
  andD: and2      port map(R, s9, s7);
  andE: and3      port map(s12, s4, s10, s8);
  orA:  Or2port map(s1, s2, s3);
  orB:  Or2port map(s7, s8, s13);
  orC:  Or2port map(s6, s13, Z);

  FFlop1: JK_FF port map(s4, s4, Clock, zero, s9, s10);
  FFlop2: JK_FF port map(s3, s3, Clock, zero, s4, s5);
End structural;
```

La descrizione strutturale segue lo stesso tipo di implementazione usata nei precedenti esempi: si dichiarano i componenti e vengono instanziati nel corpo dell'architettura.

Ancora una volta vengono utilizzate le porte logiche descritte precedentemente e viene introdotto un nuovo elemento che è il Flip Flop JKT la cui descrizione avviene di seguito ed è contenuta nel file jkflip_flop.vhd.

Un'ultima cosa da notare è che aumentando il numero di componenti, con la complessità della rete, si devono utilizzare un elevato numero di segnali per collegarli, di conseguenza per una rete più estesa può essere più semplice la descrizione comportamentale.

```
-- modello di flip flop JKT comandato da clock
-- e dotato di ingresso di reset
-- file jkflip_flop.vhd
Library ieee;
Use ieee.std_logic_1164.all;

entity JK_FF is
port (J, K, T, Reset: in std_logic;
    Q, QN: out std_logic);
end JK_FF;

architecture behv of JK_FF is
    signal state: std_logic;
    signal input: std_logic_vector(1 downto 0);
begin
  input <= J & K;
    p: process(T, Reset)
    begin
  if (reset='1') then
      state <= '0';
  elsif (T'event) and (T='1') then
        -- costruzione sulla base della tavola della verita'
      case (input) is
   when "11" =>
       state <= not state;
   when "10" =>
       state <= '1';
   when "01" =>
       state <= '0';
   when others =>
       null;
   end case;
  end if;
    end process;
    -- istruzioni concorrenti
    Q <= state;
    QN <= not state;
end behv;
```

Come si può vedere il flip flop è stato descritto in modo simile alla sua tabella della verità, lo stato del componente varia sul fronte di salita del clock e la sua uscita viene posta a zero se l'ingresso *reset* viene posto a 1. Questo aspetto è molto importante in fase di simulazione; se si osserva il file di test, testb1.vhd, utilizzato per provare la versione strutturale della macchina di Mealy, prima di far variare gli ingressi si pone alto l'ingresso *zero* per qualche nanosecondo proprio per resettare i flip flop. Si può provare a non fare questa operazione e ci si accorge subito che la macchina fornirà ai segnali un elevato numero di valori *'U' – Uninitialized.*

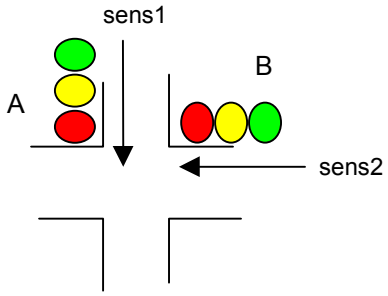
Le macchine sequenziali sono comandate da un segnale di clock, dobbiamo quindi modellizzare anche questo elemento (file clock.vhd):

```
---------------------
--
--Generatore di clock
--
---------------------
Library ieee;
Use ieee.std_logic_1164.all;

entity clk is
  generic(tempo: time:= 10 ns);
  port( y : out std_logic);
end clk;

architecture behave of clk is
begin
genera:process
   begin
    y<='0';
    wait for tempo/2;
    y<='1';
    wait for tempo/2;
   end process;
end behave;
```

In questo caso si è impostato un clock con frequenza di 100 MHz, per variare questo valore basta instanziare il componente con un valore del segnale *tempo* diverso.

Infine per simulare il comportamento complessivo della macchina si utilizzano i file testb.vhd (versione comportamentale) e testb1.vhd (versione strutturale).

Nell'esempio successivo vogliamo realizzare la simulazione un incrocio stradale con due semafori in cui la durata del segnale verde è comandata da due sensori che rilevano il traffico nelle due strade. Se il traffico aumenta in una strada raddoppia il tempo del verde sull'opportuno semaforo, se aumenta in entrambe le strade raddoppia il verde su entrambi i semafori. Il tutto viene realizzato con una macchina di Moore.



I possibili valori per i sensori saranno:

sens1	sens2	
0	0	traf. normale
0	1	più traf. su B
1	0	più traf. su A
1	1	traf. intenso

I possibili valori per i semafori:

A/B	
00	rosso
01	verde
10	giallo

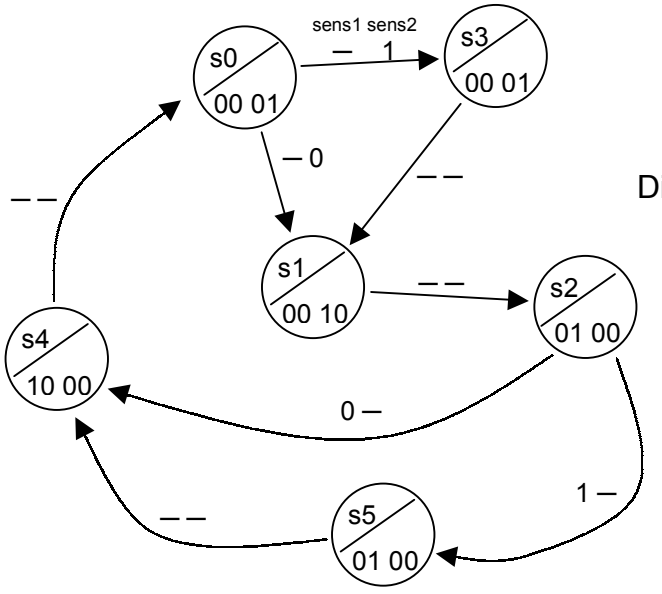
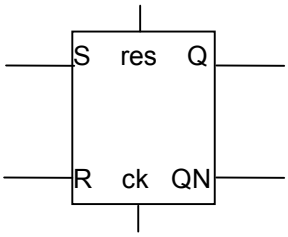


Diagramma degli stati

Tabella degli stati

sens1 / sens2 S	00	01	11	10	Z
s0	s1	s3	s3	s1	00 01
s1	s2	s2	s2	s2	00 10
s2	s4	s4	s5	s5	10 00
s3	s1	s1	s1	s1	00 01
s4	s0	s0	s0	s0	10 00
s5	s4	s4	s4	s4	01 00

Sviluppata l'analisi del problema si giunge alla sintesi di una rete logica con due ingressi e quattro uscite in cui si utilizzano tre elementi di memoria di tipo SR comandati da un segnale di clock.



Si forniscono di seguito le espressioni logiche di funzionamento della rete ottenuta:

- Sa= (x_2 *and* QNa *and* QNb *and* QNc) *or* (x_1 *and* QNa *and* Qb *and* Qc)
- Ra= (Qb *and* QNc) *or* (QNb *and* Qc)
- Sb= QNa *and* Qc
- Rb= QNa *and* QNc
- Sc= QNa *and* QNb
- Rc= QNa *and* Qb
- Z0= QNa *and* Qb *and* QNc
- Z1= (QNa *and* Qb *and* Qc) *or* (Qa *and* Qb *and* QNc)
- Z2= QNa *and* QNb *and* Qc
- Z3= (QNa *and* QNb *and* QNc) *or* (Qa *and* QNb *and* Qc)

Si è deciso di descrivere la rete in tre modi diversi:

1. modellizzazione comportamentale sulla base del diagramma degli stati
2. modello misto che utilizza una parte strutturale costituita da istanze dei bistabili e una comportamentale che utilizza le equazioni logiche fornite sopra
3. modello realizzato a partire dalla tabella degli stati

Le tre architetture dell'entità semaforo sono contenute nel file semaforo_moore.vhd

```
-- Realizzazione di una rete sequenziale sincrona con due ingressi
-- e quattro uscite, che simula un incrocio semaforico
-- Realizzata come macchina di Moore
-- File semaforo_moore.vhd
-----------------------------------
Library ieee;
Use ieee.std_logic_1164.all;

-- dichiarazione dell'entita' da simulare
entity semaforo is
  port(sens1, sens2, clock, zero: in std_logic;
      Z: out std_logic_vector(3 downto 0));
end semaforo;

-- descrizione comportamentale basata sul diagramma
-- degli stati
architecture behavioral of semaforo is
type stato is (s0, s1, s2, s3, s4, s5);
  signal stato_corrente, stato_prox: stato;
begin
 elabora: process(sens1,sens2,stato_corrente)
    begin
      case stato_corrente is
      when s0 =>
         Z<= "0001";            -- ogni when puo' essere visto
       if sens2='1' then        -- come la rappresentazione di uno
            stato_prox <= s3;   -- stato e dei suoi archi uscenti
             elsif sens2='0' then  -- sul diagramma
            stato_prox <= s1;
       end if;

      when s1 =>
         Z<= "0010";
         stato_prox <= s2;

      when s2 =>
         Z<= "0100";
       if sens1='1' then
            stato_prox <= s5;
               elsif sens1='0' then
           stato_prox <= s4;
       end if;

      when s3 =>
    Z <= "0001";
            stato_prox <= s1;

      when s4 =>
    Z <= "1000";
            stato_prox <= s0;

              when s5 =>
    Z <= "0100";
            stato_prox <= s4;
      end case;
    end process;

-- processo che controlla la transizione degli stati
-- in corrispondenza del fronte di salita del clock
controlla: process
   begin
     wait until clock'event and clock= '1';
```

```
     stato_corrente<= stato_prox;
      end process;
end behavioral;
```

Con il primo blocco di istruzioni si dichiara l'entità semaforo con i suoi ingressi e le sue uscite; al posto di utilizzare 4 uscite separate si utilizza un vettore di 4 celle. Oltre ai due segnali *sens1* e *sens2* che sono gli ingressi della rete ci serve un ingresso che riceva il clock che sincronizza il modello ed un ingresso *zero* che verrà utilizzato solo nell'architettura in cui sono presenti i flip-flop per inizializzarli correttamente.

Come si può vedere la prima architettura è del tutto simile, nella sua struttura, a quella dell'esempio precedente; si utilizza un process per l'elaborazione degli stati ed uno per il controllo delle transizioni. La differenza significativa sta nel punto in cui si aggiornano le uscite. Questo esempio infatti descrive una macchina di Moore, di conseguenza le uscite sono funzione solo dello stato corrente e non degli ingressi. Si può notare infatti che mentre nell'esempio precedente le uscite venivano modificate all'interno dei costrutti *if... then ... else* in cui si valutavano gli ingressi, ora l'uscita cambia solo in corrispondenza della valutazione dello stato corrente nel costrutto *case ... when*.

```
-- descrizione mista che utilizza come componenti, flip-flop SR
-- (parte strutturale)e descrive la parte combinatoria attraverso le sue
-- equazioni (parte comportamentale)
architecture dataflow of semaforo is

component SR_FF is
port (S, R, T, Reset: in std_logic;
    Q, QN: out std_logic);
end component;

signal SA, SB, SC, RA, RB, RC, QA, QB, QC, QNA, QNB, QNC: std_logic;

begin
  FFA: SR_FF port map (SA, RA, clock, zero, QA, QNA);
  FFB: SR_FF port map (SB, RB, clock, zero, QB, QNB);
  FFC: SR_FF port map (SC, RC, clock, zero, QC, QNC);

  SA<= (sens2 and QNA and QNB and QNC) or (sens1 and QNA and QB and QC);
  RA<= (QB and QNC) or (QNB and QC);
  SB<= QNA and QC;
  RB<= QNA and QNC;
  SC<= QNA and QNB;
  RC<= QNA and QB;

  Z(3)<= QNA and QB and QNC;
  Z(2)<= (QA and QB and QNC) or (QNA and QB and QC);
  Z(1)<= QNA and QNB and QC;
  Z(0)<= (QNA  and QNB and QNC) or (QA and QNB and QC);
end dataflow;
```

In questa architettura si è utilizzato un sistema misto di descrizione. Si può notare infatti che si dichiara il componente flip-flop SR e si creano nel corpo dell'architettura tre istanze dell'elemento, questo rappresenta un modo tipicamente strutturale per rappresentare il modello. Successivamente però invece di creare le istanze di tutte le porte logiche, come è stato fatto negli esempi precedenti, si è pensato (per semplificare la leggibilità del modello) di fornire per la parte combinatoria le espressioni logiche dei segnali di uscita e degli ingressi degli elementi di memoria.

```
-- descrizione basata sulla tabella degli stati
architecture state_table of semaforo is

function ingresso (x1,x2: std_logic)return integer is
variable temp: std_logic_vector(0 to 1);
begin
  temp:=x1 & x2;              -- funzione che allinea i valori dei
  case temp is                -- sensori con le colonne della tabella
    when "00" => return 0;    -- degli stati
    when "01" => return 1;
    when "11" => return 2;
    when "10" => return 3;
    when others => return 0;
  end case;
end function ingresso;

type table is array (integer range <>, integer range <>) of integer;
type OutTable is array (integer range <>) of std_logic_vector(3 downto 0);
signal state, NextState: integer := 0;
constant ST: table (0 to 5, 0 to 3) := ((1,3,3,1), (2,2,2,2), (4,4,5,5),
(1,1,1,1), (0,0,0,0), (4,4,4,4));
constant OT: OutTable (0 to 5) := ("0001", "0010", "0100", "0001", "1000",
"0100");

begin       -- inizio elaborazione concorrente
process(sens1, sens2, clock)  -- se il processo non fosse sensibile al clock non
variable X: integer;          -- verrebbe eseguito quando non variano
begin       -- i due sensori
  X:= ingresso(sens1, sens2);
  NextState <= ST(state,X);   -- legge lo stato prossimo dalla bella
end process;                  -- delle transizioni

elabora: process(clock)       -- processo che controlla la transizione
begin -- degli stati
  if clock = '1' then         -- fronte di salita del clock
  State <= NextState;
  end if;
end process;
Z <= OT(state);               -- legge l'uscita dall'array

end state_table;
```

L'ultima architettura presenta alcune novità rispetto a quanto visto fino ad ora, cerchiamo di analizzarne gli aspetti salienti.

Innanzitutto ciò che si è voluto fare è dare una rappresentazione della macchina di Moore basandosi sulla sua tabella degli stati; questa ha un aspetto tipico a matrice quindi si realizza il tipo *table* utilizzando un array di interi di n righe ed n colonne. La sintassi **range <>** indica che non si è stabilita a priori la dimensione dell'array. La stessa operazione viene fatta per le uscite; in questo caso però il contenuto di ogni singola cella è il vettore delle uscite Z.

Successivamente si dichiarano le costanti che contengono tutte le combinazioni degli stati prossimi e delle uscite proprio come nella tabella fornita nel testo dell'esercizio. Si può vedere che in questo momento si dimensiona la matrice degli stati *ST* (6 righe e 4 colonne) e il vettore delle uscite *OT* (6 elementi).

A questo punto nel corpo dell'architettura non resta che, in base allo stato attuale, andare a leggere nella matrice degli stati e nel vettore delle uscite le evoluzioni che deve seguire il modello. C'è però un piccolo problema dovuto a un'incongruenza tra gli ingressi dell'entità semaforo e la corrispondente colonna della matrice *ST*, dobbiamo infatti dire al modello che in corrispondenza del valore 11 deve andare a prelevare il valore dello stato prossimo sulla colonna 2 della matrice. Per risolvere questo

problema si è utilizzata una variabile *X* in cui si memorizza il valore della colonna da selezionare fornito dalla funzione *ingresso*.

A questo punto è bene precisare la differenza tra variabili e segnali. Le variabili non possono essere utilizzate nel corpo principale dell'architettura ma solo nei process, nelle funzioni e nelle procedure, mentre i segnali si utilizzano ovunque. Come si è anticipato precedentemente segnali e variabili sono aggiornati in istanti di tempo differenti, lo possiamo vedere con un esempio:

Si utilizzano variabili

```
begin
wait on trigger;
var1:= var2+var3;          var1 = 2 + 3 = 5
var2:= var1;               var2 = 5
var3:= var2;               var3 = 5
sum<= var1+var2+var3;      sum = 5 + 5 + 5 = 15 (dopo D)
end process;
```

Si utilizzano segnali

```
process
begin
wait on trigger;
sig1 <= sig2 + sig3;       sig1 = 2 + 3 = 5 (dopo D)
sig2 <= sig1;              sig2 = 1 (dopo D)
sig3 <= sig2;              sig3 = 2 (dopo D)
sum <= sig1 + sig2 + sig3; sum = 1 + 2 + 3 = 6 (dopo D)
end process;
```

In pratica si può osservare che le variabili vengono ag
mentre i segnali solo dopo un tempo delta.

In questo caso era necessario avere subito il valore degli ingressi e si è quindi fatto uso della variabile *X*. La funzione *ingresso* riceve i valori binari dei due sensori e restituisce come risultato la corretta colonna da leggere nella matrice degli stati.

Di seguito si fornisce il codice che descrive il Flip-Flop SR utilizzato (srflip_flop.vhd)

```
-- modello di flip flop SRT comandato da clock
-- e dotato di ingresso di reset
-- file srflip_flop.vhd
Library ieee;
Use ieee.std_logic_1164.all;

entity SR_FF is
port (S, R, T, Reset: in std_logic;
    Q, QN: out std_logic);
end SR_FF;

architecture behv of SR_FF is
  signal state: std_logic;
  signal input: std_logic_vector(1 downto 0);
begin
  input <= S & R;
    p: process(T, Reset)
    begin
  if (reset='1') then
      state <= '0';
  elsif (T'event) and (T='1') then

        -- costruzione sulla base della tavola della verita'
     case (input) is
   when "11" =>
       state <= 'X';
   when "10" =>
       state <= '1';
   when "01" =>
       state <= '0';
   when others =>
       null;
   end case;
  end if;
    end process;
    -- istruzioni concorrenti
    Q <= state;
    QN <= not state;
End behv;
```

La sua descrizione è del tutto simile a quella del flip-flop JK; si può notare che in corrispondenza di ingresso 11 si fornisce in uscita il valore *'X'—Unknown*

Ora non resta che procedere alla simulazione utilizzando il file di test testB.vhd

```
--------------------------------
-- TestBench per la macchina a stati
-- di Moore semaforo
-- file testB.vhd
--------------------------------
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.std_logic_TEXTIO.all;
use STD.TEXTIO.all;

entity testbench is
end testbench;

architecture FUNCTIONAL_T of testbench is

file RESULTS: TEXT open WRITE_MODE is "moore.txt";

procedure WRITE_RESULTS(a,b,c: std_logic; u:std_logic_vector) is
  variable L_OUT : LINE;
  begin
     write(l_out, now, right, 15, ns);
      write(l_out, a, right, 3);
      write(l_out, b, right, 3);
      write(l_out, c, right, 3);
      write(l_out, u, right, 5);
      writeline(results,l_out);
end;

Component semaforo
  port(sens1,sens2, clock, zero: in std_logic;
         Z: out std_logic_vector(3 downto 0));
end component;

component clk
generic(tempo: time:= 10 ns);
  port( y : out std_logic);
end component;

-- modificare qui sotto structural o behavioral o state_table a seconda del
-- tipo di implementazione che si vuole testare

For all: semaforo use entity semaforo(state_table);

signal ss1,ss2, ck, reset: std_logic;
signal yy: std_logic_vector(3 downto 0);
Begin
  UUT: semaforo port map(sens1=>ss1, sens2=>ss2, clock=>ck, zero=>reset, Z=>yy);

  GENERA: clk generic map(20 ns) port map(y=>ck);

  GEN: process
    begin
      ss2<='0';
      wait until ck'event and ck= '0';
      ss2<='1';
      wait until ck'event and ck= '0';
      ss1<='0';
      wait until ck'event and ck= '0';
      ss1<='0';
      wait until ck'event and ck= '0'; --si torna in S0

      ss2<='1';
```

```
    wait until ck'event and ck= '0'; --si va in S3
    ss2<='1';
    wait until ck'event and ck= '0'; --si va in S2
    ss2<='0';
    wait until ck'event and ck= '0'; --si va in S5
    ss1<='1';
    wait until ck'event and ck= '0';
    ss1<='1';
    wait until ck'event and ck= '0';
    ss1<='1';                --si torna in s0 dopo il giro piu'lungo
   end process;
    WRITE_TO_FILE: WRITE_RESULTS(ck, ss1, ss2, yy);
End FUNCTIONAL_T;

configuration FUNCTIONAL_CFG of testbench is
  for FUNCTIONAL_T
  end for;
end FUNCTIONAL_CFG;
```

Il file di test va bene per provare tutte le architetture sviluppate in precedenza; abbiamo detto che queste sono contenute in un unico file semaforo_moore.vhd e di conseguenza con il costrutto **For all: semaforo use ...** si deve indicare quale architettura simulare. Si deve fare attenzione solo ad una cosa, in questo testbench non si fa il reset automatico della rete; questa operazione serve solo se si fa la simulazione dell'architettura *dataflow* per inizializzare la rete. Basta che all'inizio della simulazione si forzi a 1 il valore dell'ingresso *zero* della macchina a stati per un piccolo istante di tempo e tutto funziona a dovere.

Riassunto

In questa sezione abbiamo applicato le conoscenze apprese in precedenza alla macchine sequenziali. Abbiamo realizzato, con l'aiuto di un paio di esempi, la modellizzazione sia di macchine di Moore che di Mealy. Ancora una volta si può osservare che, a fronte di una complessità che dipende dal progetto che si vuole realizzare, è possibile per una stessa macchina descrivere modelli diversi. In particolare abbiamo visto modelli basati sul diagramma degli stati, sullo schema logico e sulla tabella degli stati; la loro realizzazione, soprattutto per progetti più articolati, risulta più o meno complicata quindi sta a noi scegliere quale sia la più opportuna.

Con le reti sequenziali si introducono gli elementi di memoria e il segnale di sincronizzazione. Negli esempi trattati si sono utilizzati flip flop JK ed SR oltre che un dispositivo generatore di clock; si vuole far osservare che anche in questo caso è possibile realizzare, o trovare in altri progetti, descrizioni differenti da quelle proposte per questi componenti.

Quick Reference dei nuovi costrutti introdotti:

```
SOTTOPROGRAMMI
Function fcn_name [(Constant | Signal | Variable name_list: subtype)]
  Return type Is
Begin
{Sequential statement}
End {fcn_name};
```

```
DEFINIZIONI DI TIPO
type ID is ( {ID,} );
type ID is range number downto | to number;
type ID is array ( {range | TYPEID ,}) of TYPEID;
```

```
DICHIARAZIONI
Constant name_list: subtype [:= expr];
Variable name_list: subtype [:= expr];
```

```
ATTRIBUTI PER SEGNALI
Signal_name'Event
```

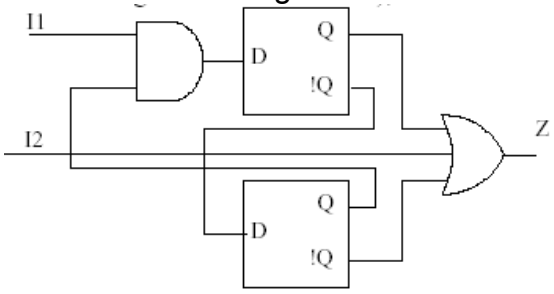
Esercizi Proposti

Esercizio 9

Data la seguente descrizione strutturale (non è riportato il segnale di clock per chiarezza descrittiva) realizzarne il modello in VHDL seguendo lo schema logico riportato.
[Soluzione proposta Dflip_flop.vhd, Esercizio9.vhd, Porte_logiche.vhd]

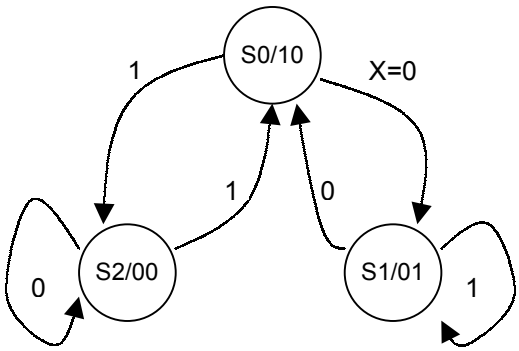
Esercizio 10

Data la seguente descrizione strutturale (non è riportato il segnale di clock per chiarezza descrittiva) realizzarne il modello in VHDL seguendo lo schema logico riportato.



Esercizio 11

Descrivere utilizzando, i costrutti VHDL noti, la seguente macchina sequenziale sincrona



[Soluzione proposta Esercizio11.vhd]

Esercizio 12

Descrivere utilizzando, i costrutti VHDL noti, la seguente macchina sequenziale sincrona

	reset	Active
start	start	LOW
LOW	start	HIGH/0
HIGH	start	LOW/1

Esercizio 13

Data la seguente descrizione relativa ad una macchina di Mealy sequenziale sincrona completamente specificata, realizzare il modello VHDL sulla base della sua tabella degli stati

	0	1
start	4/01	2/10
2	8/00	4/11
3	8/00	6/11
4	6/01	3/10
5	5/01	2/10
6	start/01	7/10
7	8/01	start/11
8	9/01	2/11
9	8/01	2/10
10	8/00	start/00

Esercizio 14

Data la seguente descrizione relativa ad una macchina sequenziale sincrona completamente specificata, realizzare il modello VHDL utilizzando i costrutti noti, produrre lo schema circuitale e realizzare il relativo modello facendo uso di Flip Flop JK.

	0	1
a	b/11	a/10
b	b/01	start/00
c	b/11	c/10
d	d/01	start/00
start	e/11	c/10
e	e/01	f/00
f	e/11	f/10

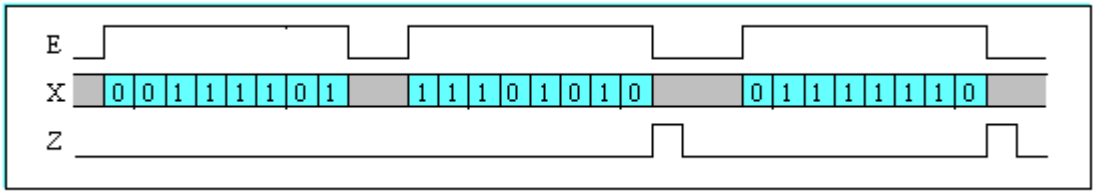
Esercizio 15

Si sintetizzi in VHDL un FFJK facendo uso di FF di tipo D.
La soluzione proposta si basa su un'architettura di tipo strutturale.
[Soluzione proposta Esercizio15.vhd, porte_logiche.vhd, Dflip_flop.vhd]

Macchine sequenziali interagenti

Quando abbiamo parlato di reti combinatorie abbiamo visto come sia possibile realizzare un progetto gerarchico in VHDL; ci proponiamo di fare la stessa operazione per le macchine sequenziali. Per fare ciò partiamo da un problema iniziale che richiede quanto segue:

Una rete sequenziale sincrona è caratterizzata da due segnali d'ingresso (E, X) e da un segnale di uscita Z. Attraverso l'ingresso X la rete riceve serialmente parole di k bit. Il segnale E, attivo per k intervalli di clock, segnala la fase di ricezione di ciascuna parola. L'uscita Z può assumere il valore logico 1 soltanto in corrispondenza dell'intervallo di clock immediatamente successivo a quello di ricezione dell'ultimo bit di ciascuna parola, e ciò se la parola comprende almeno tre 1 consecutivi, ma non due 0 consecutivi.



Per il problema dato si identifica il seguente diagramma degli stati di una rete sequenziale di Mealy:

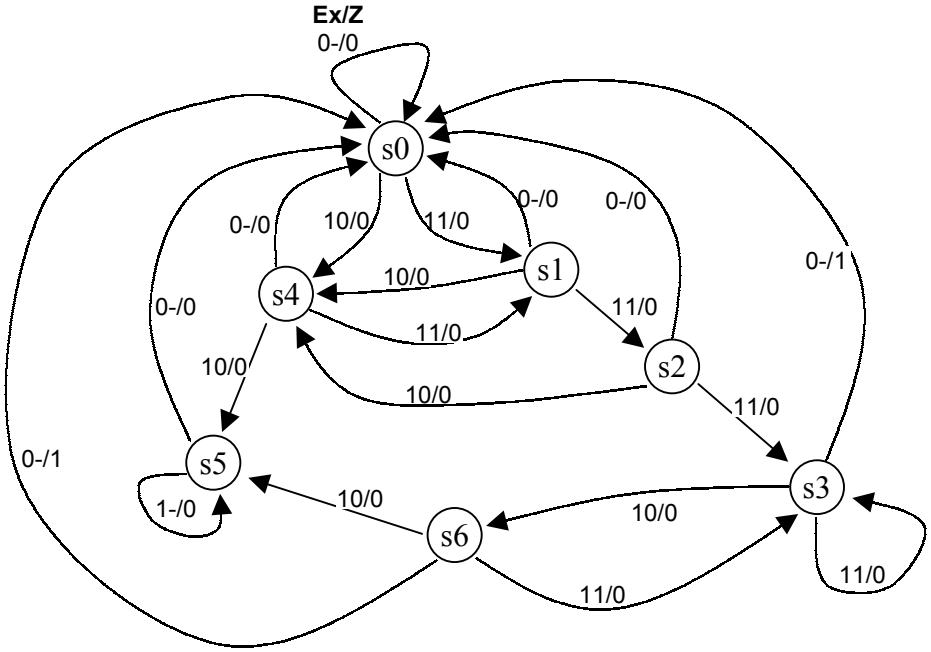


Tabella degli stati

S \ Ex	00	01	11	10
s0	s0, 0	s0, 0	s1, 0	s4, 0
s1	s0, 0	s0, 0	s2, 0	s4, 0
s2	s0, 0	s0, 0	s3, 0	s4, 0
s3	s0, 1	s0, 1	s3, 0	s6, 0
s4	s0, 0	s0, 0	s1, 0	s5, 0
s5	s0, 0	s0, 0	s5, 0	s5, 0
s6	s0, 1	s0, 1	s3, 0	s5, 0

Descriviamo ora la rete in VHDL in modo comportamentale seguendo un approccio simile al diagramma degli stati, il codice è contenuto nel file sequenze.vhd

```
-----------------------------------
-- Realizzazione di un riconoscitore di sequenze
-- con macchina a stati di Mealy
-- file sequenze.vhd
-----------------------------------
Library ieee;
Use ieee.std_logic_1164.all;

entity Sequenze is
      port(E,X, clock: in std_logic;
             Z: out std_logic);
end Sequenze;

architecture behavioral of Sequenze is
type stato is (s0, s1, s2, s3, s4 , s5, s6);
signal stato_corrente, stato_prox: stato;
begin
  elabora: process(E,X,stato_corrente)
             begin
              case stato_corrente is
                  when s0 =>
                      if E='0' then
                        Z <= '0';
                        stato_prox <= s0;
                      elsif (E='1' and X='1') then
                        Z <= '0';
                        stato_prox <= s1;
                      elsif (E='1' and X='0') then
                        Z<= '0';
                        stato_prox<= s4;
                      end if;

                  when s1 =>
                      if E='0' then
                        Z <= '0';
                        stato_prox <= s0;
                      elsif (E='1' and X='1') then
                        Z <= '0';
                        stato_prox <= s2;
                      elsif (E='1' and X='0') then
                        Z<= '0';
                        stato_prox<= s4;
                      end if;

                  when s2 =>
                      if E='0' then
                        Z <= '0';
                        stato_prox <= s0;
                      elsif (E='1' and X='1') then
                        Z <= '0';
                        stato_prox <= s3;
                      elsif (E='1' and X='0') then
                        Z<= '0';
                        stato_prox<= s4;
                      end if;
```

```
                when s3 =>
                    if E='0' then
                      Z <= '1';
                      stato_prox <= s0;
                    elsif (E='1' and X='1') then
                      Z <= '0';
                      stato_prox <= s3;
                    elsif (E='1' and X='0') then
                      Z<= '0';
                      stato_prox<= s6;
                    end if;

                when s4 =>
                   if E='0' then
                      Z <= '0';
                      stato_prox <= s0;
                   elsif (E='1' and X='1') then
                      Z <= '0';
                      stato_prox <= s1;
                   elsif (E='1' and X='0') then
                      Z<= '0';
                      stato_prox<= s5;
                   end if;

                 when s5 =>
                    if E='0' then
                      Z <= '0';
                      stato_prox <= s0;
                    elsif E='1' then
                      Z <= '0';
                      stato_prox <= s5;
                    end if;

                when s6 =>
                    if E='0' then
                      Z <= '1';
                      stato_prox <= s0;
                    elsif (E='1' and X='1') then
                      Z <= '0';
                      stato_prox <= s3;
                    elsif (E='1' and X='0') then
                      Z<= '0';
                      stato_prox<= s5;
                    end if;
                end case;
            end process;

controlla: process
            begin
              wait until clock'event and clock= '1';
              stato_corrente<= stato_prox;
           end process;
end behavioral;
-- ************************************
```

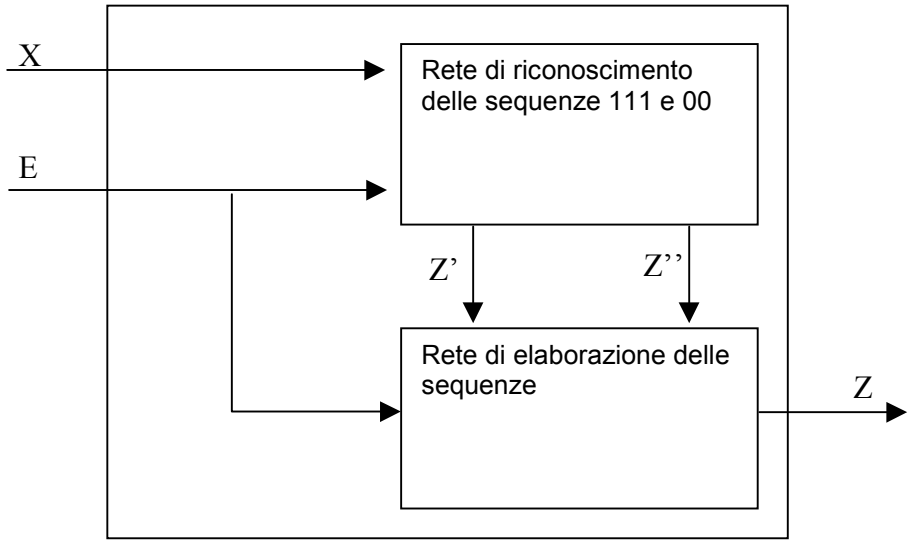
La struttura descrittiva è identica a quella utilizzata negli esempi precedenti; ci sono due process concorrenti, il primo (*elabora*) che analizza gli ingressi per ogni stato del diagramma e di conseguenza prevede le transizioni che devono avvenire, mentre il secondo (*controlla*) sincronizza la rete e fa avvenire le transizioni di stato solo sul fronte di salita del clock, assegnando allo stato corrente il valore dello stato prossimo.

Supponiamo ora di cambiare le specifiche del problema pensando che l'uscita Z si debba attivare quando riconosce parole in cui:

- c'è la sequenza 111 ma non 00 (come nel problema iniziale)
- ci sono entrambe le sequenze 111 e 00
- c'è la sequenza 00 ma non 111

E' evidente che l'approccio usato fin qui ci obbliga a realizzare ex novo una rete sequenziale senza poter riutilizzare il lavoro precedentemente fatto.
E' possibile però, applicando il principio di decomposizione, scomporre una rete sequenziale in 2 reti più semplici. Questo principio consente non solo di semplificare e strutturare meglio il progetto, ma anche di riutilizzarne alcune parti a fronte di variazione delle specifiche.

Possiamo quindi pensare di risolvere il problema con una soluzione di questo tipo



Rete di riconoscimento delle sequenze:
La rete deve attivare le uscite quando riconosce in ingresso parole contenenti le sequenze 111 e 00; le parole vengono lette quando è attivo l'ingresso E

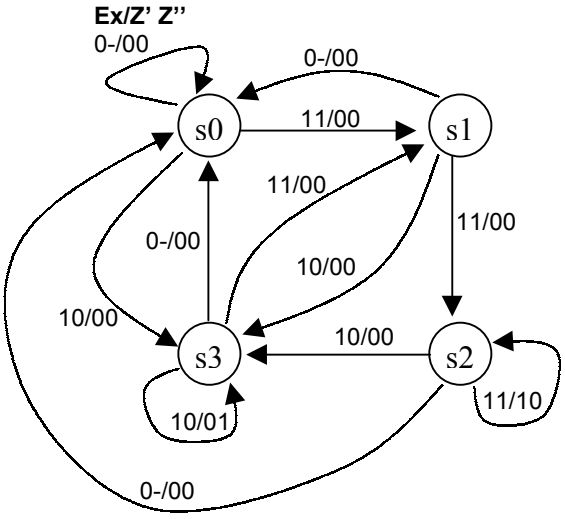


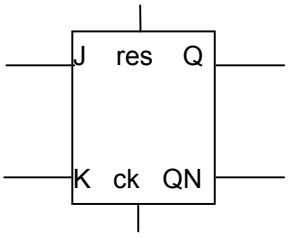
Tabella degli stati

S \ Ex	00	01	11	10
s0	s0, 00	s0, 00	s1, 00	s3, 00
s1	s0, 00	s0, 00	s2, 00	s3, 00
s2	s0, 00	s0, 00	s2, 10	s3, 00
s3	s0, 00	s0, 00	s1, 00	s3, 01

Z'= 1 riconosciuta la sequenza 111
Z''= 1 riconosciuta la sequenza 00

La rete ottenuta è realizzata con flip flop JK e si forniscono di seguito le espressioni logiche:

- J1= (*not* X *and* E) *or* (E *and* Qb)
- K1= *not* E *or* (X *and* QNb)
- J2= E *and* X
- K2= *not* E *or not* X
- Z'= E *and* X *and* Qa *and* Qb
- Z''= E *and* Qa *and* QNb *and not* X



Ancora una volta possiamo descrivere la rete analizzata in diversi modi; decidiamo di utilizzare un modello comportamentale sulla base del diagramma degli stati ed un modello che faccia uso delle espressioni logiche della rete (file ric_mealy.vhd)

```
-------------------------------------
--
-- Realizzazione di una rete sequenziale sincrona con due ingressi
-- e due uscite, che si comporti come un riconoscitore di sequenze
-- 111 e 00 realizzata come macchina di Mealy
-- File ric_mealy.vhd
-------------------------------------
Library ieee;
Use ieee.std_logic_1164.all;

entity Riconoscitore is
      port(E,X, clock, zero: in std_logic;
             Z: out std_logic_vector(0 to 1));
end Riconoscitore;

architecture behavioral of Riconoscitore is
type stato is (s0, s1, s2, s3);
  signal stato_corrente, stato_prox: stato;
begin
      elabora: process(E,X,stato_corrente)
                  begin
                       case stato_corrente is
                             when s0 =>
                           if E='0' then
                             Z <= "00";
                             stato_prox <= s0;
                       elsif (E='1' and X='1') then
                             Z <= "00";
                             stato_prox <= s1;
                       elsif (E='1' and X='0') then
                             Z<= "00";
                             stato_prox<= s3;
                         end if;

                         when s1 =>
                           if E='0' then
                             Z <= "00";
                             stato_prox <= s0;
                       elsif (E='1' and X='1') then
```

```
                        Z <= "00";
                        stato_prox <= s2;
                    elsif (E='1' and X='0') then
                        Z<= "00";
                        stato_prox<= s3;
                      end if;

                      when s2 =>
                        if E='0' then
                          Z <= "00";
                          stato_prox <= s0;
                    elsif (E='1' and X='1') then
                          Z <= "10";
                          stato_prox <= s2;
                    elsif (E='1' and X='0') then
                          Z<= "00";
                          stato_prox<= s3;
                      end if;

                          when s3 =>
                        if E='0' then
                          Z <= "00";
                          stato_prox <= s0;
                    elsif (E='1' and X='1') then
                          Z <= "00";
                          stato_prox <= s1;
                    elsif (E='1' and X='0') then
                          Z<= "01";
                          stato_prox<= s3;
                      end if;
                  end case;
                end process;

    controlla: process
                begin
                      wait until clock'event and clock= '1';
                      stato_corrente<= stato_prox;
                end process;
end behavioral;
```

La descrizione comportamentale non necessita di particolari commenti visto che la sua struttura e del tutto simile a quella dell'esempio precedente.

```
-- descrizione mista che utilizza come componenti, flip-flop JK
-- (parte strutturale)e descrive la parte combinatoria attraverso le sue
-- equazioni (parte comportamentale)
architecture dataflow of riconoscitore is

component JK_FF is
port (J, K, T, Reset: in std_logic;
       Q, QN: out std_logic);
end component;

signal JA, JB, KA, KB, QA, QB, QNA, QNB: std_logic;

begin
      FFA: JK_FF port map (JA, KA, clock, zero, QA, QNA);
      FFB: JK_FF port map (JB, KB, clock, zero, QB, QNB);

      JA<= (E and (not X)) or (E and QB);
      KA<= (not E) and (X and QNB);
```

```
      JB<= E and X;
      KB<= not E or not X;

      Z(0)<= E and X and QA and QB;
      Z(1)<= E and not X and QA and QNB;
end dataflow;
-- ***************************************************
```

Nell'architettura *dataflow* vengono create le istanze dei due flip flop presenti nella rete, i quali si interfacciano con i segnali che trasportano i risultati delle espressioni logiche della parte combinatoria della rete. La dichiarazione degli otto segnali di tipo std_logic è necessaria proprio per collegare i flip flop al resto della rete.

Rete di elaborazione delle sequenze:
La rete riceve in ingresso le uscite del riconoscitore di sequenze 111 e 00. Quando riceve il segnale di fine parola E ci dovrà quali di esse erano presenti secondo le richieste del problema.

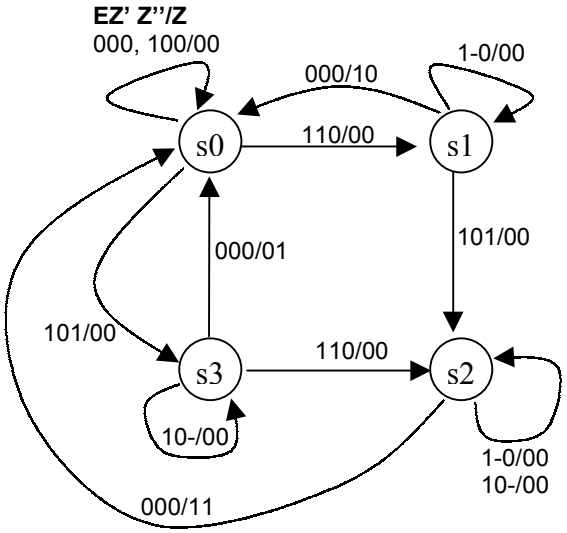
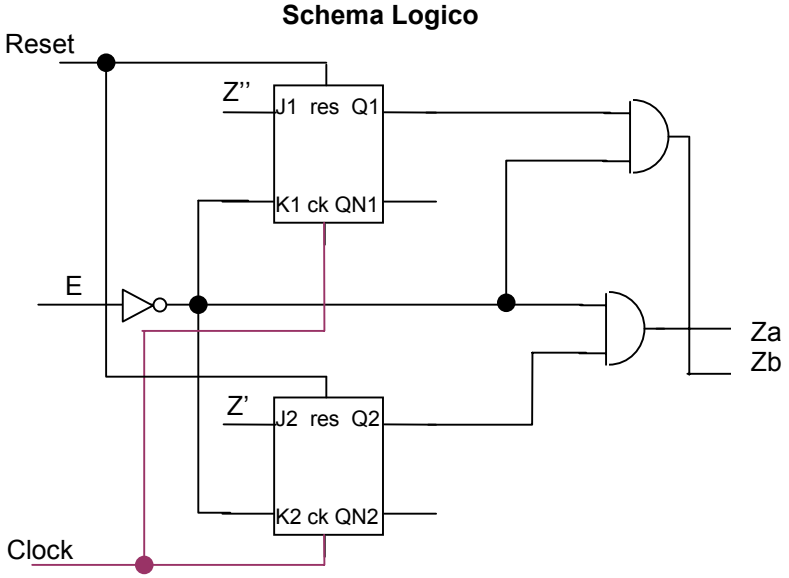


Tabella degli stati

S \ EZ'Z''	000	100	101	110
s0	s0, 00	s0, 00	s3, 00	s1, 00
s1	s0, 10	s1, 00	s2, 00	s1, 00
s2	s0, 11	s2, 00	s2, 00	s2, 00
s3	s0, 01	s3, 00	s3, 00	s2, 00

Z= 00 nessuna specifica rispettata
Z= 01 presente solo seq. 00 e non 111
Z= 10 presente solo seq. 111 e non 00
Z= 11 presenti 111 e 00

Schema Logico



Questi schemi ci bastano per realizzare differenti modelli della rete in oggetto. Per differenziarci dai sistemi usati nella rete di riconoscimento questa volta descriviamo un modello completamente strutturale sulla base dello schema logico ed uno ancora comportamentale sulla base della tabella degli stati. Non dovrebbero essere introdotte grandi novità in quanto tali sistemi sono già stati usati per i primi esempi di reti sequenziali. Tutto il codice è contenuto nel file elab_seq.vhd.

```
-----------------------------------
--
-- Realizzazione di una rete sequenziale sincrona con tre ingressi
-- e due uscite, che elabora le uscite prodotte dal riconoscitore
-- di sequenze, realizzata come macchina di Mealy
-- File elab_seq.vhd
-----------------------------------
Library ieee;
Use ieee.std_logic_1164.all;

entity elabora_seq is
      port(xA, xB, E, clock, zero: in std_logic;
             Z: out std_logic_vector(0 to 1));
end elabora_seq;

architecture structural of elabora_seq is
component JK_FF
port (J, K, T, Reset: in std_logic;
        Q, QN: out std_logic);
end component;

component Not_g
      PORT (u: IN std_logic;
            y: OUT std_logic);
end component;

component And2
      PORT (a,b: IN std_logic; y: OUT std_logic);
end component;

signal s1, s2, s3: std_logic;

Begin
      notA: not_g port map(E, s1);
      andA: and2  port map(s1, s2, Z(1));
      andB: and2  port map(s1, s3, Z(0));

      FFlop1:     JK_FF port map(xB, s1, Clock, zero, s2);
      FFlop2:     JK_FF port map(xA, s1, Clock, zero, s3);
End structural;
```

Nell'architettura appena descritta si può riconoscere in modo evidente l'aspetto strutturale, con le istanze dei diversi componenti che vengono prodotte nel corpo dell'architettura e che vengono mappate sui segnali che hanno la funzione di collegare i componenti.
Si ricorda che per poter utilizzare i componenti questi devono essere descritti da qualche parte ed i particolare nel file porte.vhd si trovano gli elementi logici **and** a due ingressi e **not**, mentre nel file jkflip_flop.vhd il corrispondente elemento di memoria.
In questo caso risulta particolarmente poco oneroso descrivere un'architettura strutturale grazie alla semplicità dello schema logico della rete, mentre avrebbe richiesto un po' più di tempo la descrizione sulla base del diagramma degli stati.

```
-- descrizione basata sulla tabella degli stati
architecture state_table of elabora_seq is

function ingresso (a,b,c: std_logic)return integer is
variable temp: std_logic_vector(0 to 2);
begin
  temp:=a & b & c;                  -- funzione che allinea i valori degli
  if temp="000" then                -- ingressi con le colonne della tabella
      return 0;                     -- degli stati
  elsif temp="100" then
      return 1;
  elsif temp="101" then
      return 2;
  elsif temp="110" then
      return 3;
  else return 4;
      end if;
end function ingresso;

type table is array (integer range <>, integer range <>) of integer;
type OutTable is array (integer range <>, integer range <>) of
std_logic_vector(0 to 1);
signal state, NextState: integer := 0;
constant ST: table (0 to 3, 0 to 3) := ((0,0,3,1), (0,1,2,1), (0,2,2,2),
(0,3,3,2));
constant OT: OutTable (0 to 3, 0 to 3) := (("00","00","00","00"),
("10","00","00","00"), ("11","00","00","00"), ("01","00","00","00"));

begin             -- inizio elaborazione concorrente
process(xA, xB, E, clock) -- se il processo non fosse sensibile al clock non
variable X: integer;      -- verrebbe eseguito quando non variano gli ingressi
begin
  X:= ingresso(E, xA, xB);
  if X=4 then
      Z<= "ZZ";
  else
   NextState <= ST(state,X);  -- legge lo stato prossimo
   Z <= OT(state,X);          -- legge l'uscita
  end if;
end process;

elabora: process(clock) -- processo che controlla la transizione degli stati
            begin
             if clock = '1' then    -- fronte di salita del clock
                  State <= NextState;
             end if;
          end process;
end state_table;
-- ********************************
```

Questo tipo di descrizione è già stata utilizzata in precedenza quindi dovrebbe essere chiaro l'uso degli array a matrice, delle variabili e delle funzioni.

In particolare in questo caso si può vedere l'utilità della funzione ingresso; la macchina a stati infatti ha tre bit di ingresso che però assumono solo particolari configurazioni delle 2^n possibili, in quanto queste dipenderanno dalle uscite fornite dal riconoscitore di sequenze con cui dovrà interagire. Di conseguenza abbiamo la necessità di specificare, in base ad un dato ingresso, quale colonna della tabella

degli stati deve essere presa in considerazione. Si può notare inoltre che la funzione restituisce un valore uguale a 4 nel caso in ingresso non si abbia nessuna delle configurazioni previste; in tal caso il processo che riceve il risultato nel corpo dell'architettura pone in uscita un valore di alta impedenza **ZZ**. Nel normale regime di funzionamento della macchina, collegata al riconoscitore, questo non accadrà mai; questa possibilità è stata aggiunta solo al fine di testare il funzionamento della macchina non collegata a nessun altro dispositivo. Se infatti si provasse, in fase di simulazione, a fornire un ingresso non previsto, senza questo espediente la funzione *ingresso* restituirebbe un segnale d'errore bloccando il simulatore.

Un'altro aspetto differente rispetto all'esempio precedente, in cui è stato utilizzato lo stesso tipo di architettura, riguarda la funzione di uscita. In questo caso infatti, essendo il modello di una macchina di Mealy, anche per l'uscita si deve utilizzare un array a matrice essendo questa dipendente dallo stato corrente e dall'ingresso.

Una volta descritte le due reti che dovranno interagire non ci resta che descrivere la rete complessiva che risponde alle specifiche del progetto; il codice è contenuto nel file interagenti.vhd.

```
-----------------------------------
--
-- Realizzazione di una rete sequenziale sincrona con due ingressi
-- e due uscite. La rete e' costituita da due macchine
-- interagenti e si comporta come da specifiche
-- File interagenti.vhd
-----------------------------------
Library ieee;
Use ieee.std_logic_1164.all;

entity macchine_interagenti is
      port(read, word, ck, rst: in std_logic;
             Y: out std_logic_vector(0 to 1));
end macchine_interagenti;


architecture structural of macchine_interagenti is

component riconoscitore is
      port(E,X, clock, zero: in std_logic;
             Z: out std_logic_vector(0 to 1));
end component;

component elabora_seq is
      port(xA, xB, E, clock, zero: in std_logic;
             Z: out std_logic_vector(0 to 1));
end component;

-- entrambe le macchine sono state descritte in due architetture
-- diverse, si deve scegliere quindi quale architettura utilizzare
for all: riconoscitore use entity riconoscitore(behavioral);
for all: elabora_seq use entity elabora_seq(state_table);

signal collega: std_logic_vector(0 to 1);

begin
      rete1: riconoscitore port map(read, word, ck, rst, collega);
      rete2: elabora_seq port map(collega(0), collega(1), read, ck, rst, Y);
end structural;
-- ***********************************
```

L'architettura con cui si realizza la macchina complessiva non può che avere un aspetto di tipo strutturale; si dichiarano le due macchine descritte in precedenza come component e all'interno del corpo dell'architettura si creano le relative istanze collegando tra loro le uscite del riconoscitore con gli ingressi dell'elaboratore.

Unica nota riguarda il costrutto *For all:* ; abbiamo visto che per ogni macchina sono state descritte due architetture diverse, **behavioral** e **dataflow** per il riconoscitore, **structural** e **state_table** per l'elaboratore, con questo costrutto possiamo decidere quale architettura dovrà essere utilizzata all'interno della macchina complessiva.

Ultimo passo da compiere è l'effettiva prova della macchina così realizzata, ancora una volta si crea un testbench file ad hoc e si forniscono in ingresso alcune sequenze di prova (file testb.vhd).

```
--------------------------------
-- TestBench per la macchina a stati
-- gerarchica
-- file testB.vhd
--------------------------------
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.std_logic_TEXTIO.all;
use STD.TEXTIO.all;

entity testbench is
end testbench;

architecture FUNCTIONAL_T of testbench is

file RESULTS: TEXT open WRITE_MODE is "int.txt";

procedure WRITE_RESULTS(a,b,c: std_logic; u:std_logic_vector) is
      variable L_OUT : LINE;
      begin
      write(l_out, now, right, 15, ns);
          write(l_out, a, right, 3);
          write(l_out, b, right, 3);
          write(l_out, c, right, 3);
          write(l_out, u, right, 5);
          writeline(results,l_out);
end;

component macchine_interagenti is
      port(read, word, ck, rst: in std_logic;
             Y: out std_logic_vector(0 to 1));
end component;

component clk
generic(tempo: time:= 10 ns);
      port( y : out std_logic);
end component;

For all: macchine_interagenti use entity macchine_interagenti(structural);

signal X,E, clock, reset: std_logic;
signal yy: std_logic_vector(0 to 1);
Begin
UUT: macchine_interagenti port map(read=>E, word=>X, ck=>clock, rst=>reset,
Y=>yy);

GENERA: clk generic map(20 ns) port map(y=>clock);

GEN: process
      begin
            E<='0';
            wait until clock'event and clock= '0';
            E<='1';
```

```
            X<='1';
            wait until clock'event and clock= '0';
            X<='1';
            wait until clock'event and clock= '0';
            X<='1';
            wait until clock'event and clock= '0';
            X<='0';
            E<='0';
            wait until clock'event and clock= '0';
            -- fine della prima parola di test


            E<='1';
            X<='0';
            wait until clock'event and clock= '0';
            X<='0';
            wait until clock'event and clock= '0';
            X<='1';
            wait until clock'event and clock= '0';
            X<='0';
            E<='0';
            wait until clock'event and clock= '0';
            -- fine della seconda parola di test

            E<='1';
            wait until clock'event and clock= '0';
            X<='1';
            wait until clock'event and clock= '0';
            X<='0';
            wait until clock'event and clock= '0';
            X<='0';
            wait until clock'event and clock= '0';
            X<='1';
            wait until clock'event and clock= '0';
            X<='1';
            wait until clock'event and clock= '0';
            X<='1';
            wait until clock'event and clock= '0';
            X<='0';
            E<='0';
            wait until clock'event and clock= '0';
            -- fine della terza parola di test

      end process;
      WRITE_TO_FILE: WRITE_RESULTS(clock, E, X, yy);
End FUNCTIONAL_T;

configuration FUNCTIONAL_CFG of testbench is
  for FUNCTIONAL_T
  end for;
end FUNCTIONAL_CFG;
-- ************************************************
```

Per effettuare il test della macchina gerarchica si dichiara questa come component, si utilizza anche il generatore di clock come component e si creano le loro istanze all'interno del corpo dell'architettura dell'entità di test. Successivamente si crea un process che genera gli ingressi per la macchina sequenziale.

In questo caso particolare abbiamo provato il comportamento della macchina su tre diverse parole di bit: 1110 (presenza della sequenza 111 e non 00), 0010 (presenza della sequenza 00 e non 111), 1001110 (presenza di entrambe le sequenze). I risultati forniti dalla macchina sono stati quelli attesi e possono

essere verificati, o lanciando il testbench stesso e visualizzando le forme d'onda prodotte, oppure leggendo il file int.txt prodotto a fine simulazione.

Da quanto esposto in precedenza, quando abbiamo parlato dei file di test, dovrebbe essere chiaro che la lettura di un diagramma che presenti graficamente le evoluzioni dei segnali risulterà più immediata e semplice rispetto al file di testo che produce i valori di tutti i segnali per tutti gli istanti di tempo discreto analizzati dal simulatore.

E' ovvio che fornendo diverse sequenze di ingresso possono essere effettuate altre verifiche.

Si può infine notare che se ora volessimo cambiare ancora le specifiche del progetto, richiedendo altre elaborazioni sulle sequenze precedenti, sarà possibile ad esempio riutilizzare il riconoscitore e si dovrà riprogettare solo la parte di elaborazione. Non solo, confrontando il diagramma degli stati della macchina di inizio esercizio con i diagrammi delle due reti interagenti si evidenzia, come anticipato, una notevole semplificazione di questi.

Riassunto

Esattamente come abbiamo fatto per l'analisi di strutture combinatorie, siamo riusciti a realizzare un modello gerarchico per una rete sequenziale, utilizzando in questo caso il principio di decomposizione.

Siamo partiti da un problema iniziale che non dava altre possibilità di minimizzazione degli stati e lo abbiamo implementato seguendo il suo diagramma degli stati. Ci siamo poi chiesti cosa avremmo potuto fare del nostro lavoro se le specifiche del problema fossero cambiate anche di poco. A questo punto abbiamo capito che era il caso di scomporre il progetto iniziale per renderlo flessibile a possibili variazioni delle specifiche senza dover per forza ripartire da capo ogni volta. Per fare ciò non è servito introdurre nessuna novità, rispetto a quanto visto in precedenza; con i semplici costrutti a nostra disposizione siamo riusciti a realizzare un modello modulare proprio come avevamo già fatto per le reti combinatorie.

Esercizi proposti

Esercizio 16

Data la seguente specifica, la si traduca in VHDL.
Il dispositivo ha due ingressi scalari, A e B, un ingresso vettoriale DATO da 8 bit e una uscita vettoriale OUT da 8 bit. Le configurazioni di ingresso (AB) 01 e 10 memorizzano DATO sul registro R1 e R2, rispettivamente. Le configurazioni di ingresso 00 e 11 presentano su OUT il contenuto di R1 e R2, rispettivamente. Durante il campionamento (scrittura nei registri) il valore di OUT è l'ultimo valore portato in uscita (inizialmente 0).

Esercizio 17

Data la seguente descrizione relativa ad una FSM non completamente specificata, realizzarne il modello VHDL utilizzando i costrutti noti. Dopo aver fatto l'appropriato assegnamento, sintetizzare la macchina con flip flop di tipo D.

	00	01	11	10
A	B/0	B/1	-/-	-
B	A/-	E/-	-/-	-
C	-/1	E/-	D/1	-
D	A/1	-/1	C/0	-
E	B/1	E/1	C/0	-

Esercizio 18

Data la seguente specifica, la si traduca in VHDL.
Il dispositivo ha come ingressi il segnale A e il vettore DATO da 8 bit e come uscita il segnale OUT. Un fronte di salita sull'ingresso A impone la memorizzazione di DATO su un registro R da 8 bit. Quando il valore di A torna a 0, il registro R viene decrementato ad

ogni colpo di clock. Al raggiungimento del valore 0 nel registro R, sulla uscita OUT (normalmente a 0) viene posto il valore 1 per un ciclo di clock.
Durante tutta la fase di conteggio l'ingresso A è disabilitato.

Esercizio 19
Data la seguente descrizione relativa ad una FSM non completamente specificata, realizzarne il modello VHDL utilizzando i costrutti noti.

	0	1
S0	-	S1/1
S1	S0/1	S2/1
S2	S5/1	S3/1
S3	S4/-	S3/-
S4	S0/0	S2/0
S5	S4/0	-

Altri Esempi

Si propongono di seguito alcuni esempi di modelli trovati sul web che possono risultare di interesse per una maggiore comprensione delle metodologie di programmazione VHDL.

Esempio 1: registro a N bit
Realizzazione di un registro comportamentale a N bit, dove N può essere specificato in fase di test oppure ha valore predefinito uguale a 4. registro1.vhd

```
-------------------------------------------------------------------
-- Design unit: reg(behavioural) (Entity and Architecture)
--            :
-- File name  : reg.vhd
--            :
-- Description: RTL model of N bit synchronous register
--            :
-- Limitations: None
--            :
-- System     : VHDL'93, STD_LOGIC_1164
--            :
-- Author     : Mark Zwolinski
--            : Department of Electronics and Computer Science
--            : University of Southampton
--            : Southampton SO17 1BJ, UK
--            : mz@ecs.soton.ac.uk
--
-- Revision   : Version 1.1 27/11/01
-------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
entity reg is
  generic (n : natural := 4);
  port (D : in std_ulogic_vector(n-1 downto 0);
        Clock, Reset : in std_ulogic;
        Q : out std_ulogic_vector(n-1 downto 0));
end entity reg;

architecture behavioural of reg is
begin
p0: process (Clock, Reset) is
    begin
      if (Reset = '0') then
        Q <= (others => '0');
      elsif rising_edge(Clock) then
        Q <= D;
      end if;
    end process p0;
end architecture behavioural;
```

Esempio 2: shift register a 8 bit
Modello comportamentale di uno shift register a 8 bit. Il modello è dotato di un ingresso di clock che sincronizza le operazioni di shift e di lettura del dato sul fronte di salita, di un bit di reset che azzera il contenuto del registro e di un bit di Load che abilita il registro alla lettura della parola Data in ingresso. Si fornisce di seguito anche il relativo testbench file per poter agevolmente effettuare la simulazione. ShiftReg.vhd, testshift.vhd

```
-------------------------------------------------------
-- 8-bit barrel shifter
--
-- This example circuit demonstrates the behavior level
-- of abstraction. The operation of the barrel shifter
-- is described as the circuit's response to stimulus
-- (such as clock and reset events) over time.
--
-- This circuit is synthesizable. Test the circuit with
-- the supplied test bench (testshif.vhd)
--
-- Copyright 1995, Accolade Design Automation, Inc.
--

library ieee;
use ieee.std_logic_1164.all;

entity shifter is
    port( Clk, Rst, Load: in std_ulogic;
          Data: std_ulogic_vector(0 to 7);
          Q: out std_ulogic_vector(0 to 7));
end shifter;

architecture behavior of shifter is
begin
    -- We use a process to describe the operation of
    -- the shifter over time, in response to its inputs...
    reg: process(Rst,Clk)
        -- Using a variable simplifies register feedback...
        variable Qreg: std_ulogic_vector(0 to 7);
    begin
        if Rst = '1' then   -- Async reset
            Qreg := "00000000";
        elsif rising_edge(Clk) then
            if Load = '1' then
                Qreg := Data;
            else
                Qreg := Qreg(1 to 7) & Qreg(0);
            end if;
        end if;
        Q <= Qreg;
    end process;
end behavior;
-- *********************************************************


-------------------------------------------------------
-- Test bench for 8-bit barrel shifter
--
-- Copyright 1995, Accolade Design Automation, Inc.
--
library ieee;
use ieee.std_logic_1164.all;

entity testrot is
end testrot;

architecture stimulus of testrot is
    component shifter
        port(Clk, Rst, Load: in std_ulogic;
             Data: std_ulogic_vector(0 to 7);
```

```
            Q: out std_ulogic_vector(0 to 7));
    end component;
--
    constant PERIOD: time := 40 ns;
--
    signal Clk,Rst,Load: std_ulogic;
    signal Data: std_ulogic_vector(0 to 7);
    signal Q: std_ulogic_vector(0 to 7);
--
begin
    DUT: shifter port map(Clk,Rst,Load,Data,Q);

    -- This test bench uses two processes to describe
    -- the stimulus.  This first process describes a
    -- constantly running clock of 40 ns cycle time...
    CLOCK: process
    begin
        Clk <= '1';
        wait for PERIOD / 2;
        Clk <= '0';
        wait for PERIOD / 2;
    end process;
    -- This process applies a sequence of inputs to the
    -- circuit to exercise this shift and load features...
    INPUTS: process
    begin
        wait for PERIOD / 2;
        Rst <= '1';
        Data <= "00000000";
        Load <= '0';
        wait for PERIOD;
        Rst <= '0';
        wait for PERIOD;
        Data <= "00001111";
        Load <= '1';
        wait for PERIOD;
        Load <= '0';
        wait for PERIOD * 4;
        Rst <= '1';
        Data <= "00000000";
        Load <= '0';
        wait for PERIOD;
        Rst <= '0';
        wait for PERIOD;
        Data <= "10101010";
        Load <= '1';
        wait for PERIOD;
        Load <= '0';
        wait for PERIOD * 4;
         Rst <= '1';
        Data <= "00000000";
        Load <= '0';
        wait for PERIOD;
        Rst <= '0';
        wait for PERIOD;
        Data <= "10000001";
        Load <= '1';
        wait for PERIOD;
        Load <= '0';
        wait for PERIOD * 4;
        wait;
     end process;
end stimulus;
```

```
-- ******************************
```

Esempio 3: resistro Serial-Input Parallel-Output

Modello di registro Serial Input Parallel Output a N bit, questo numero, se non modificato, è uguale a 8. In ingresso si legge un bit per volta, sul fronte di salita del clock, e viene fornito in uscita il risultato in parallelo. Attenzione che il primo bit che si legge sarà il più significativo della parola in uscita quando si sarà riempito il registro.

Si può notare come, all'interno del processo p0, venga utilizzata una variabile reg di tipo vettore per fornire in uscita subito il valore aggiornato del registro. Per shiftare il contenuto a sinistra di volta in volta si usa l'operatore di concatenamento &. SiPoReg.vhd

```
-------------------------------------------------------------------
-- Design unit: sipo(rtl) (Entity and Architecture)
--            :
-- File name  : sipo.vhd
--            :
-- Description: RTL model of serial in, parallel out register
--            :
-- Limitations: None
--            :
-- System     : VHDL'93, STD_LOGIC_1164
--            :
-- Author     : Mark Zwolinski
--            : Department of Electronics and Computer Science
--            : University of Southampton
--            : Southampton SO17 1BJ, UK
--            : mz@ecs.soton.ac.uk
--
-- Revision   : Version 1.0 08/03/00
-------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity sipo is
  generic(n : natural := 8);
  port(a : in std_ulogic;
       q : out std_ulogic_vector(n-1 downto 0);
       clk : in std_ulogic);
end entity sipo;

architecture rtl of sipo is
begin
p0: process (clk) is
      variable reg : std_ulogic_vector(n-1 downto 0);
    begin
      if rising_edge(clk) then
        reg := reg(n-2 downto 0) & a;
        q <= reg;
      end if;
    end process p0;
end architecture rtl;
-- ***********************************************
```

Esempio 4: ROM (16 parole da 7 bit)

Modello di una memoria ROM di 16 parole a 7 bit che contiene i valori delle uscite di un display a sette segmenti. La realizzazione del modello è molto semplice; in ingresso si ha un *address* di tipo intero con un range che va da 0 a 15; in uscita il vettore *data* a 7 bit che conterrà la parola cercata. Si definisce poi il tipo *rom_array* che altro non è che un array di 16 celle di vettori a 7 bit, successivamente con l'uso della costante *rom* si riempie la memoria dei valori. Il lavoro è finito in quanto nel corpo dell'architettura ci basta un'istruzione per leggere il contenuto di una cella di memoria e porlo in uscita.

```
-- Description: RTL model of ROM containing decoding patterns for
--            : 7 segment display
--            :
-- Limitations: None
--            :
-- System     : VHDL'93, STD_LOGIC_1164
--            :
-- Author     : Mark Zwolinski
--            : Department of Electronics and Computer Science
--            : University of Southampton
--            : Southampton SO17 1BJ, UK
--            : mz@ecs.soton.ac.uk
--
-- Revision   : Version 1.0 08/03/00
--------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
entity rom16x7 is
  port (address : in integer range 0 to 15;
        data : out std_ulogic_vector (6 downto 0));
end entity rom16x7;

architecture sevenseg of rom16x7 is
  type rom_array is array (0 to 15) of std_ulogic_vector(6 downto 0);
  constant rom : rom_array := (
                               "1110111",
                               "0010010",
                               "1011101",
                               "1011011",
                               "0111010",
                               "1101011",
                               "1101111",
                               "1010010",
                               "1111111",
                               "1111011",
                               "1101101",
                               "1101101",
                               "1101101",
                               "1101101",
                               "1101101",
                               "1101101");
begin
  data <= rom(address);
end architecture sevenseg;
-- ****************************************
```

Esempio 5: RAM parametrica (default: 256 parole da 8 bit)

Modello per una RAM la cui dimensione può essere impostata in fase di test. Se questo non viene fatto, di default la memoria sarà di 256 parole da 8 bit.

Il tipo Unsigned utilizzato in questo modello gestisce una serie di funzioni matematiche di conversione per i vettori std_logic. Si può osservare infatti che, mentre dall'esterno noi forniamo indirizzi binari delle celle di memoria, il processo interno opera su numeri interi, utilizzando la funzione di conversione *conv_integer* contenuta proprio nella libreria std_logic_unsigned. Ram.vhd

```
--------------------------------------------------------------------------------
--
--      The following information has been generated by Exemplar Logic
--      and may be freely distributed and modified.
--
--      Design name : ram
--
--      Purpose : This design is a generic ram. Both the address and data
--                have programmable widths. This ram can be used in place
--                of technology specific rams.
--
--------------------------------------------------------------------------------
Library IEEE ;
use IEEE.std_logic_1164.all ;
use IEEE.std_logic_arith.all ;
use IEEE.std_logic_unsigned.all ;

entity ram is
   generic (data_width     : natural := 8 ;
            address_width  : natural := 8);
   port (
         data_in  : in  UNSIGNED(data_width - 1 downto 0) ;
         address  : in  UNSIGNED(address_width - 1 downto 0) ;
         we       : in  std_logic ;
         data_out : out UNSIGNED(data_width - 1 downto 0)
        );
end ram ;

architecture rtl of ram is
  type mem_type is array (2**address_width downto 0) of
                      UNSIGNED(data_width - 1 downto 0) ;
  signal mem : mem_type ;
  begin
    I0 : process (we,address,mem,data_in)
      begin
        if (we = '1') then
          mem(conv_integer(address)) <= data_in ;
        end if ;
        data_out <= mem(conv_integer(address)) ;
    end process ;
end RTL ;
-- ***************************************************
```

Esempio 6: Priority Encoder a N bit

Modello di un priority encoder a N bit. Il dispositivo restituisce l'indirizzo del primo 1 che trova sul bus dati in ingresso; è in grado di riconoscere anche la presenza di tutti zeri.

Il modello proposto può risultare di non semplicissima comprensione dato che utilizza alcuni costrutti che non sono stati presentati precedentemente. Lo si presenta comunque per mostrare l'utilizzo dei cicli in VHDL. In questo caso vengono utilizzati due cicli *For*; il primo, contenuto nella funzione *to_unsigned*, serve per calcolare l'indirizzo del bit a 1 trovato; il secondo, contenuto nel corpo dell'architettura, serve per effettuare la scansione del bus alla ricerca del primo 1. Priority_Encoder.vhd

```
--------------------------------------------------------------------------------
--
--      The following information has been generated by Exemplar Logic and
--      may be freely distributed and modified.
--
--      Entity name : priority_encoder
--
--      Purpose : This design is an n-bit priority encoder. It looks at the
--        input data bus and returns the address of the 1st "1" found in the
--        word. If also has a output the detects if the entire data bus is all
--        zeros.
--
--------------------------------------------------------------------------------
Library IEEE ;
use IEEE.std_logic_1164.all ;
use IEEE.std_logic_arith.all ;

entity priority_encoder is
   generic (data_width    : natural := 25 ;
            address_width : natural := 5 ) ;
   port (
         data    : in  UNSIGNED(data_width - 1 downto 0) ;
         address : out UNSIGNED(address_width - 1 downto 0) ;
         none    : out STD_LOGIC
        );
end priority_encoder ;

architecture rtl of priority_encoder is
  attribute SYNTHESIS_RETURN : STRING ;

  FUNCTION to_stdlogic (arg1:BOOLEAN)  RETURN STD_LOGIC IS
      BEGIN
      IF(arg1) THEN
        RETURN('1') ;
      ELSE
        RETURN('0') ;
      END IF ;
  END ;

    function to_UNSIGNED(ARG: INTEGER; SIZE: INTEGER) return UNSIGNED is
      variable result: UNSIGNED(SIZE-1 downto 0);
      variable temp: integer;
        attribute SYNTHESIS_RETURN of result:variable is "FEED_THROUGH" ;
    begin
      temp := ARG;
      for i in 0 to SIZE-1 loop
          if (temp mod 2) = 1 then
            result(i) := '1';
          else
            result(i) := '0';
          end if;
          if temp > 0 then
            temp := temp / 2;
          else
            temp := (temp - 1) / 2;
          end if;
      end loop;
      return result;
    end;

  constant zero : UNSIGNED(data_width downto 1) := (others => '0') ;
```

```
  begin
PRIO :  process(data)
         variable temp_address : UNSIGNED(address_width - 1 downto 0) ;
         begin
          temp_address := (others => '0') ;
          for i in data_width - 1 downto 0 loop
            if (data(i) = '1') then
              temp_address := to_unsigned(i,address_width) ;
              exit ;
            end if ;
          end loop ;
          address <= temp_address ;
          none <= to_stdlogic(data = zero) ;
        end process ;
end RTL ;
-- ***********************************************
```