



# **Introduzione al VHDL per logiche programmabili**





# Programma del corso

- 1) Introduzione al VHDL
  - Perché utilizzare il VHDL
- 2) PAL, CPLD e FPGA
  - Descrizione di una PAL
  - Descrizione di una CPLD
  - Descrizione di una FPAG
  - Principali differenze tra le FPGA disponibili in commercio
- 3) Il sistema di sviluppo per FPGA e CPLD Xilinx Foundation Express
- 4) Descrizione di un semplice progetto VHDL
  - Differenti modalità di descrizione di un circuito VHDL
  - Entity e architetture
  - Tipi di dati e operatori associati
- 5) Progetto di circuiti combinatori e circuiti sincroni
  - Descrizione di un semplice circuito combinatorio
  - Progetto di una FIFO
- 6) Progetto di macchine a stati finiti
  - Descrizione di una semplice macchina a stati finiti
  - Macchine di Moore e macchine di Mealy
- 7) Gestione di progetti complessi
  - Il concetto di gerarchia
  - Gestione e creazione di librerie
- 8) Funzioni e procedure
  - Definizione e creazione di funzioni
  - Definizione e creazione di procedure
- 9) Implementazione di un progetto
  - Esempio di implementazione di un progetto
  - Controllo dei parametri di implementazione



## Perché usare il VHDL

- **Potenza e flessibilità:** il VHDL contiene costrutti complessi con i quali è possibile scrivere in maniera succinta il codice di descrizione di circuiti complessi. Il VHDL consente diversi tipi di descrizione per controllare l'implementazione del progetto. Consente inoltre la progettazione tramite librerie e la creazione di librerie stesse definite dall'utente. Il VHDL è un linguaggio per la progettazione gerarchica. Inoltre è un linguaggio che oltre alla progettazione consente la simulazione del progetto.
- **Progettazione indipendente dal dispositivo:** il VHDL consente di creare un progetto senza dover prima scegliere un dispositivo sul quale implementarlo.
- **Portabilità:** il VHDL è un linguaggio standardizzato e quindi consente la portabilità del codice da un simulatore, un sintetizzatore ad un altro. Questo significa che un progetto VHDL o parte di esso possono essere riutilizzati in altri progetti su altri tipi di dispositivi.
- **Facilità di valutare le prestazioni del circuito:** la progettazione indipendente dal dispositivo e la portabilità consentono di valutare le prestazioni di un circuito su differenti dispositivi e con differenti programmi di sintesi. Questo permette di poter scegliere il dispositivo che meglio si adatta alle specifiche del progetto.
- **Migrazione verso gli ASIC:** quando si rende necessario, il VHDL facilita l'implementazione su ASIC.



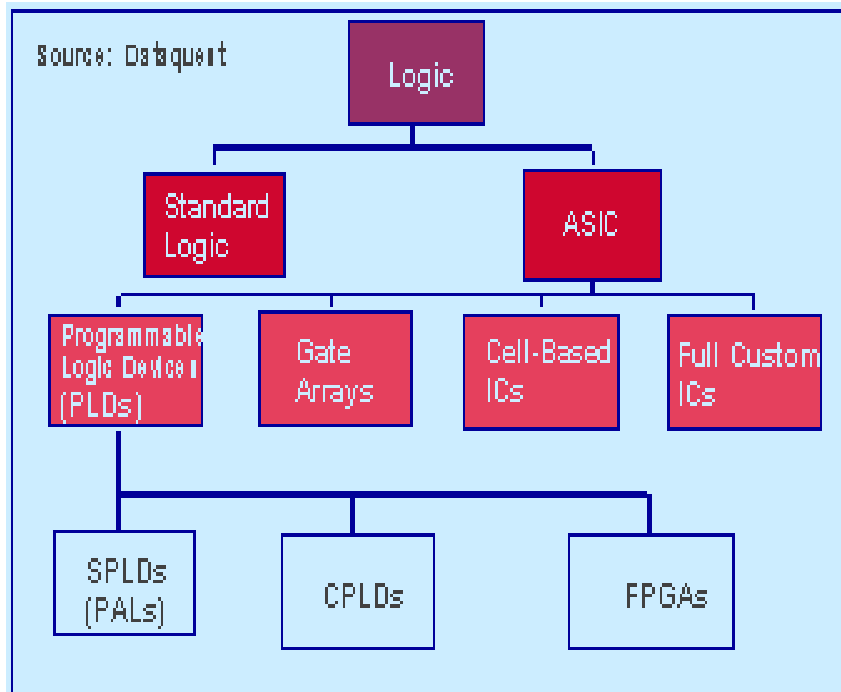
## Difetti

● **Difficoltà di controllo dell'implementazione a livello di gate dei circuiti che sono descritti con un linguaggio ad alto livello, che utilizza costrutti astratti:** non esiste alcun modo per aggirare questo difetto. Infatti, l'intento del VHDL come strumento di sintesi è quello di liberare il progettista da dover specificare l'implementazione del circuito a livello di gate.

● **Le implementazioni create dai programmi di sintesi sono a volte inefficienti:** il problema che la sintesi logica è inefficiente non è privo di giustificazione. I compilatori VHDL non sempre producono l'implementazione ottimale, specialmente perché la soluzione ottima dipende dagli obiettivi del progetto. I compilatori utilizzano algoritmi per decidere sull'implementazione logica, seguendo metodologie di progettazione standard. qualche volta non c'è un valido sostituto alla creatività umana.

● **La qualità della sintesi varia da programma a programma:** quest'ultimo difetto sta cominciando ad essere affrontato dai produttori di software per l'implementazione e la sintesi di progetti VHDL.

## CPLD e FPGA



### Evoluzione dei dispositivi programmabili

I dispositivi digitali giocano un ruolo chiave nel progetto dei dispositivi digitali. Essi sono chips general purpose che possono essere configurati per un'ampia gamma di applicazioni. Il primo tipo di dispositivo programmabile ad essere stato largamente utilizzato è stata la PROM (Programmable Read Only Memory). Un circuito logico può essere implementato utilizzando le linee di indirizzamento come ingressi del circuito, e le uscite sono definite dai bit immagazzinati. Con questa strategia può essere implementata qualunque tabella delle verità. Sono disponibili due versioni delle PROM, quelle che possono essere programmate solo dal produttore, e quelle che possono essere programmate dall'utente finale. Il primo tipo è chiamato *mask-programmable* mentre il secondo è chiamato *field-programmable*. Questi ultimi offrono diversi vantaggi:

- i dispositivi field-programmable sono meno costosi a bassi volumi di produzione rispetto a quelli mask-programmable perché sono componenti standard.
- i dispositivi field-programmable possono essere programmati immediatamente, in pochi secondi, mentre i dispositivi mask-programmable devono essere fabbricati dalla fonderia.



Sebbene le PROM sono una valida alternativa per realizzare semplici circuiti logici, è chiaro che la struttura di una PROM è più adatta all'implementazione di memorie. Un altro tipo di dispositivo programmabile progettato specificatamente per l'implementazione di circuiti logici è chiamato PLD (Programmable Logic Device). Una PLD è formata tipicamente da un array di porte AND connesse ad un array di porte OR. Per essere implementato in una PLD un circuito logico deve essere rappresentato nella forma di una somma di prodotti. La versione base di una PLD è chiamata PAL (Programmable Array Logic). Una PAL è formata da un piano di AND programmabile e da un piano di OR fisso. Le uscite delle porte OR possono essere in alcuni casi registrate mediante dei flip flop. Le PAL sono dispositivi di tipo field-programmable.

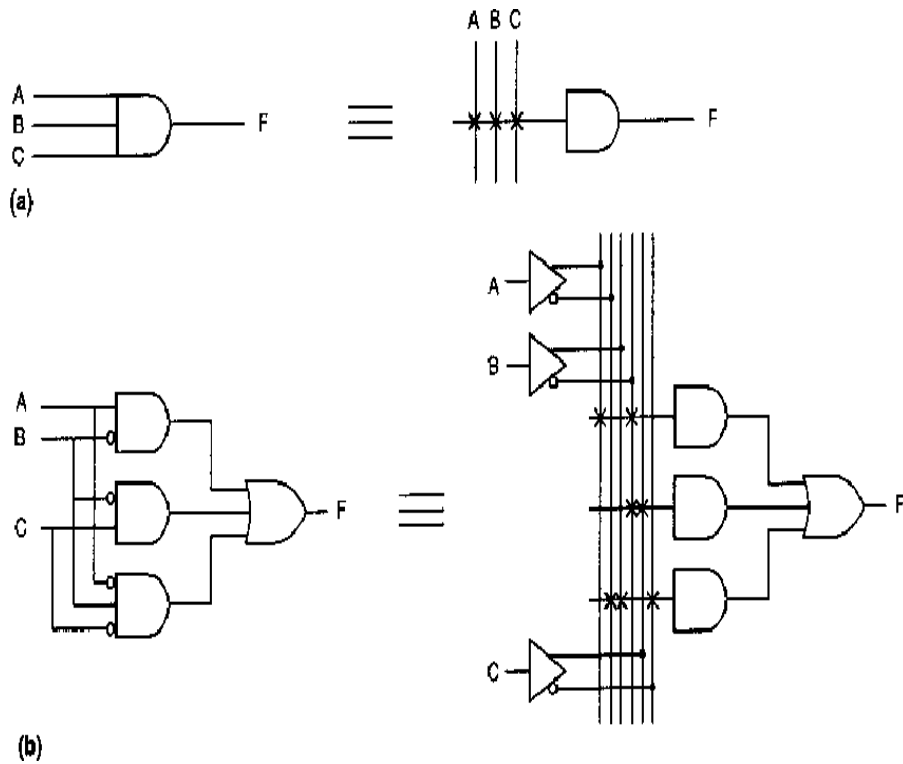
Una versione più flessibile delle PAL è la PLA (Programmable Logic Array). Anche la PLA è formata da una piano di AND e una di OR, ma in questo caso le connessioni su entrambi i piani sono programmabili.

Con la loro semplice struttura a due livelli, entrambi i tipi di PLD descritti sopra permettono l'implementazione di circuiti logici ad alta velocità. Tuttavia la loro semplice struttura è anche il loro principale difetto. Essi possono implementare solo piccoli circuiti logici che contengono un modesto numero di termini prodotto a causa del numero limitato di connessioni disponibili.

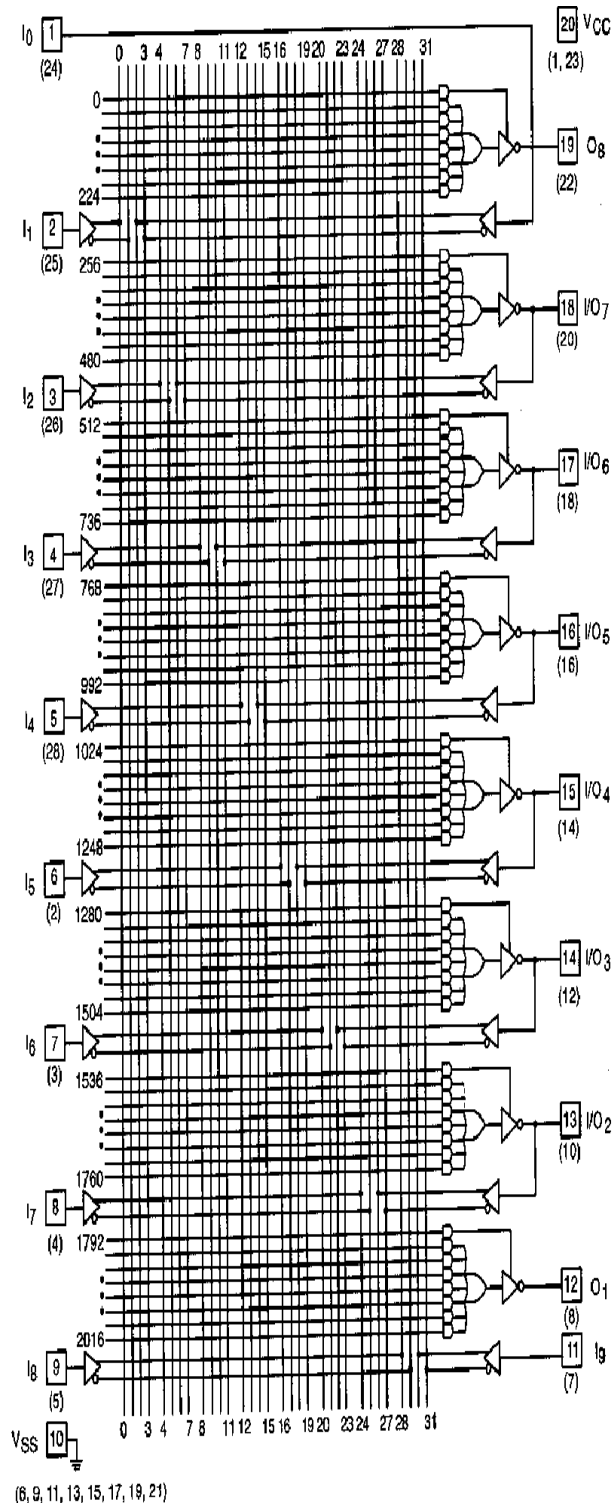
Le CPLD (Complex PLD) estendono il concetto di PLD ad un livello di integrazione più elevato per migliorare le prestazioni del sistema. Invece di costruire PLD più grandi, con più ingressi, termini prodotto e macrocelle, una CPLD contiene diversi blocchi logici, ognuno simile ad una piccola PLD. I blocchi logici comunicano l'uno con l'altro utilizzando segnali indirizzati mediante una rete di interconnessioni programmabile.

## PAL: Descrizione

Le PAL (Programmable Device Logic ) sono i più semplici dispositivi logici programmabili. Una PAL è formata da una matrice di porte AND e una matrice di porte OR. La matrice di AND è programmabile mentre quella di OR è fissa.



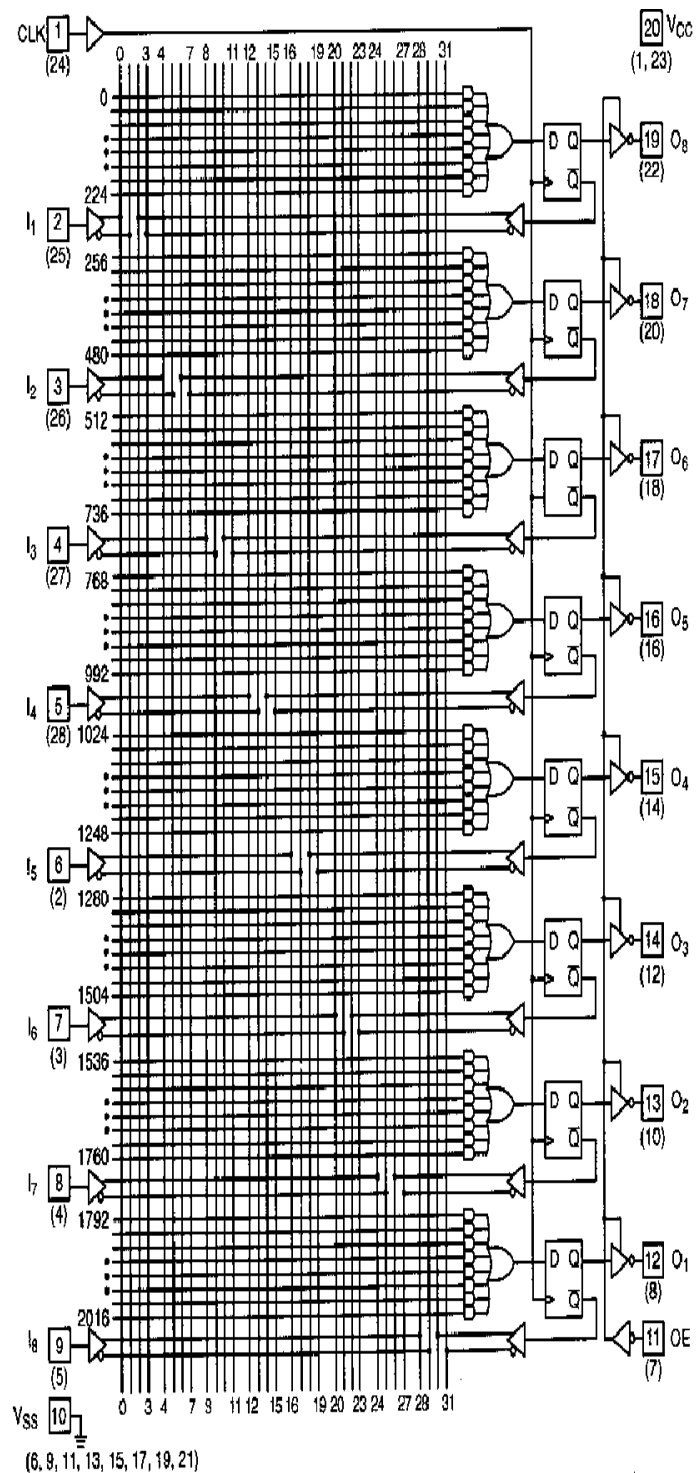
## PAL 16L8: architettura



La PAL 16L8 è chiamata così perché L sta per Logic, 16 sono gli ingressi nell'array di AND e 8 sono le uscite.

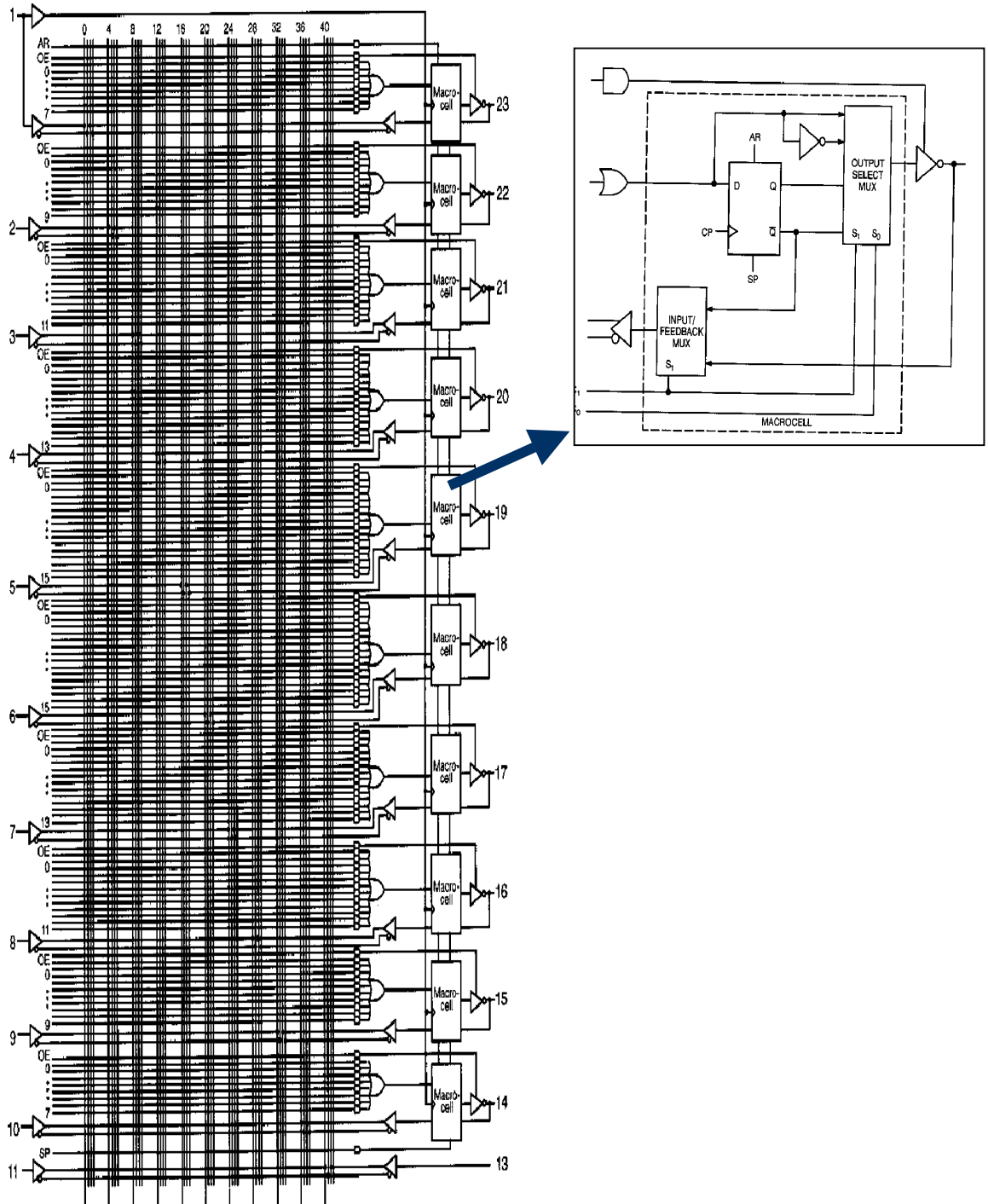


## PAL 16R8: architettura



Nella sigla della PAL 16R8 la R indica che sono presenti dei registri.

## PAL 22V10: architettura



La 22V10 rispetto alle altre PAL ha una macrocella programmabile e una distribuzione variabile di termini prodotto. Ogni macrocella può essere configurata individualmente configurando lo stato dei bit di configurazione.

Nella sigla 22V10 il 22 indica 22 ingressi, il 10 indica 10 uscite, la V sta per Variable e indica un numero variabile di termini prodotto per porte OR da 8 a 16.

## CPLD: descrizione

Invece di costruire PLD più grandi, con più ingressi, termini prodotto e macrocelle, una CPLD contiene diversi blocchi logici, ognuno simile ad una piccola PLD. I blocchi logici comunicano l'uno con l'altro utilizzando segnali indirizzati mediante una rete di interconnessioni programmabile.

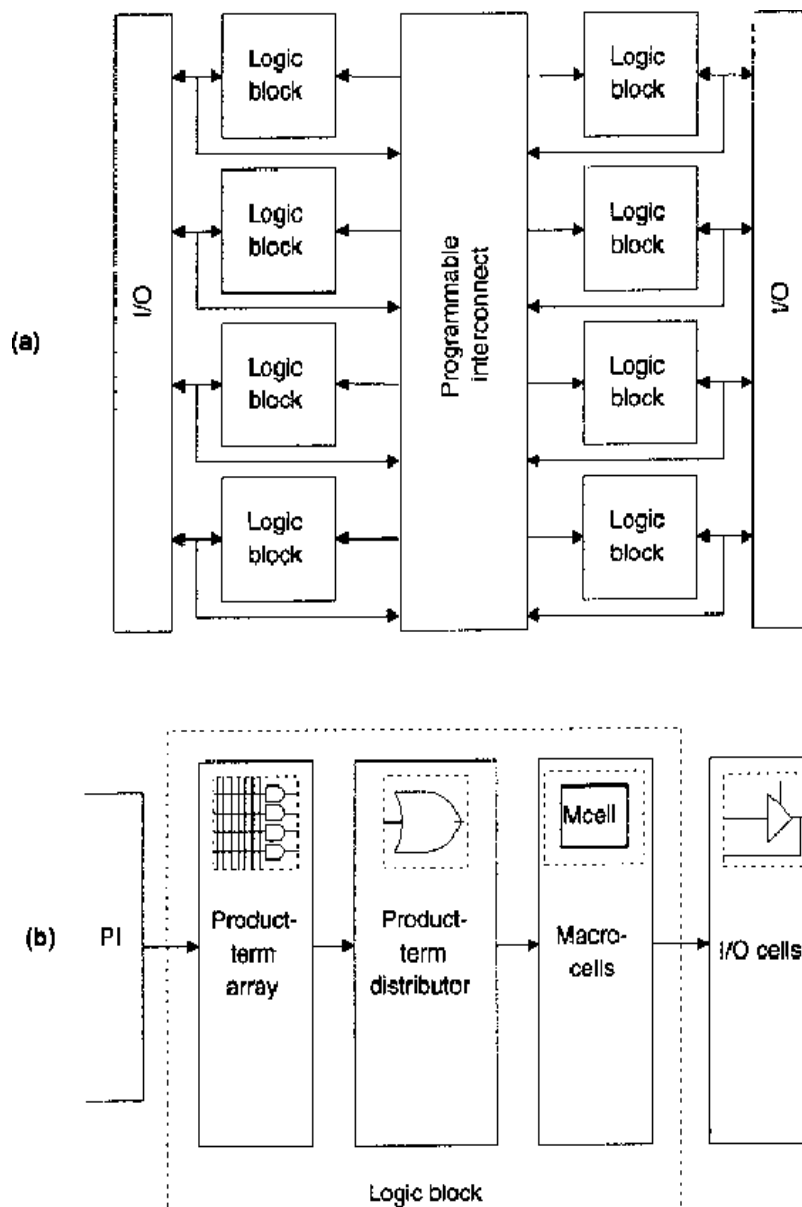
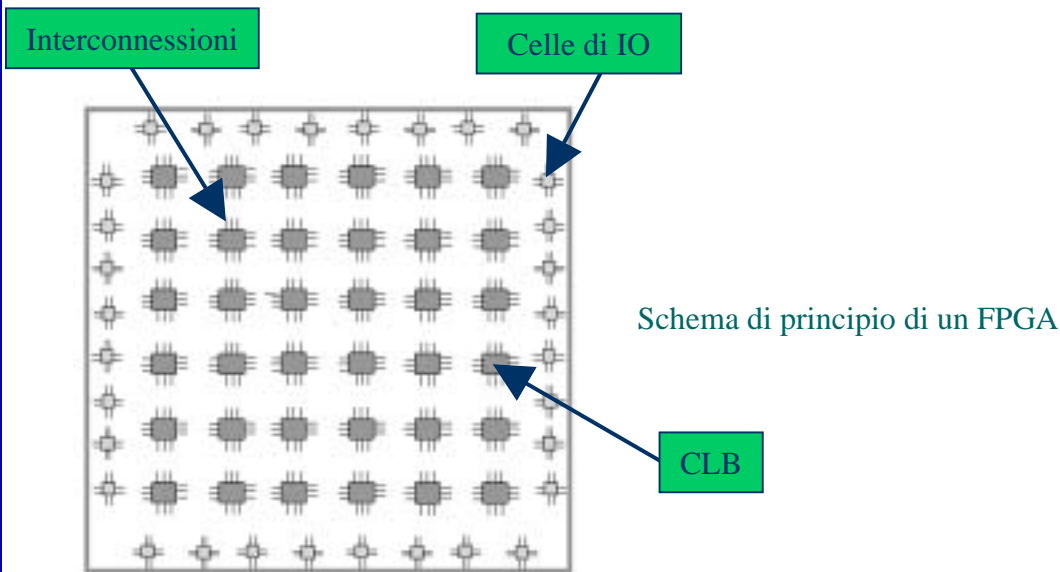


Figura (a): Architettura di una generica CPLD

(b): Generico blocco logico di una CPLD

## FPGA: descrizione



Un'FPGA (Field Programmable Gate Array) è un' array di celle logiche che possono essere interconnesse in un qualunque modo. Come nelle PAL le connessioni tra elementi sono programmabili dall'utente.

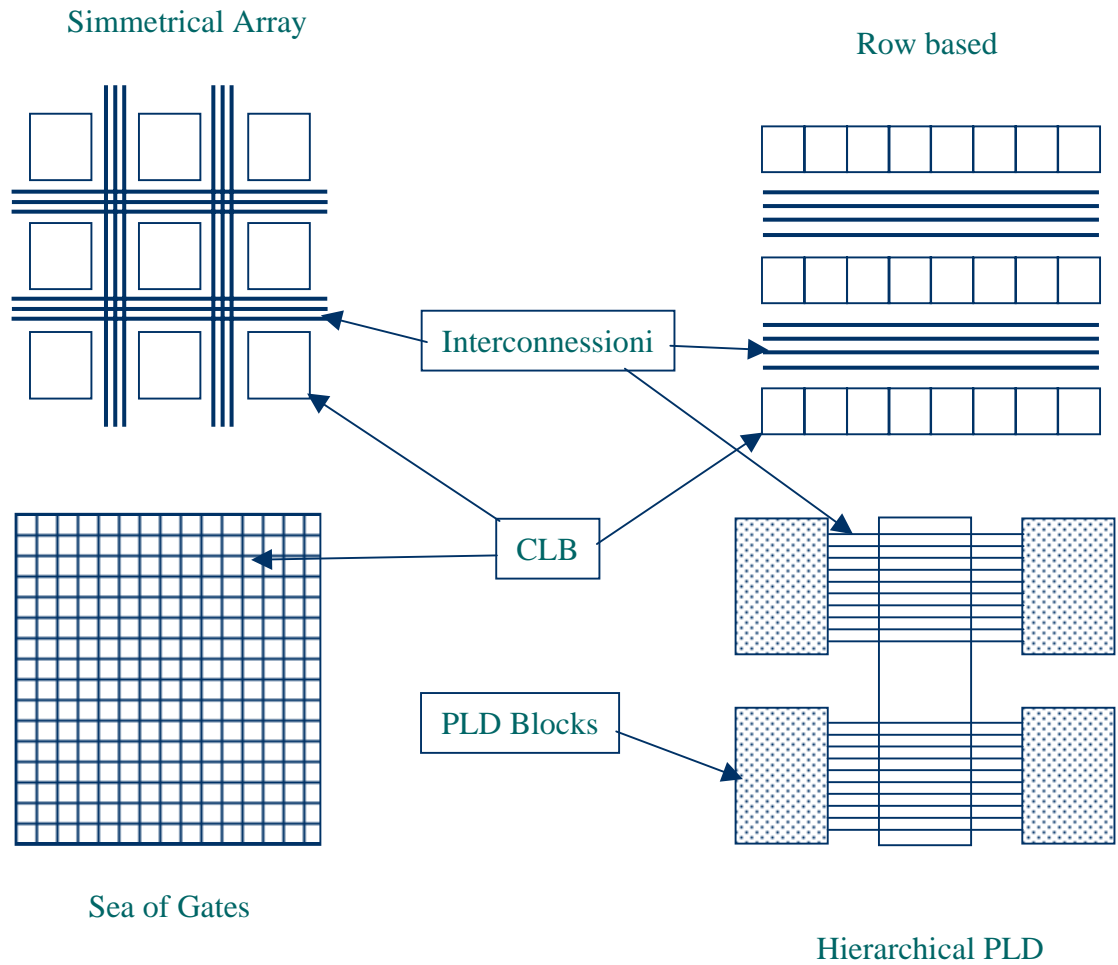
Le FPGA furono introdotte per la prima volta nel 1985 dalla Xilinx. Da allora molti differenti tipi di FPGA sono stati sviluppati da numerose altre compagnie: Actel, Altera, Plessey, Plus, Advanced Micro Devices (AMD), QuickLogic, Alotronic, Concurrent Logic, e molte altre.

La figura mostra un tipico diagramma concettuale di un'FPGA. Un'FPGA è composta da un'array bidimensionale di blocchi logici che possono essere connessi tramite una rete di interconnessioni. Nelle interconnessioni sono presenti degli switch programmabili che servono per connettere i blocchi logici con i fili della rete di interconnessioni, o segmenti di filo gli uni con gli altri.

Le FPGA sono state sviluppate per andare incontro alle esigenze del mercato, come ad esempio:

1. Le prestazioni, mettono in grado i sistemi real time di operare a frequenze sempre più elevate.
2. Densità e capacità, mettono in grado di aumentare l'integrazione, di integrare sempre più componenti su un chip (system on chip), e utilizzare tutte le porte disponibili sull'FPGA, fornendo così una soluzione efficiente anche dal punto di vista dei costi.
3. Facilità di utilizzo, mettono in grado i progettisti di portare sul mercato il loro prodotto velocemente.

## Differenze tra FPGA



Le quattro classi di FPGA disponibili commercialmente

## FPGA disponibili commercialmente

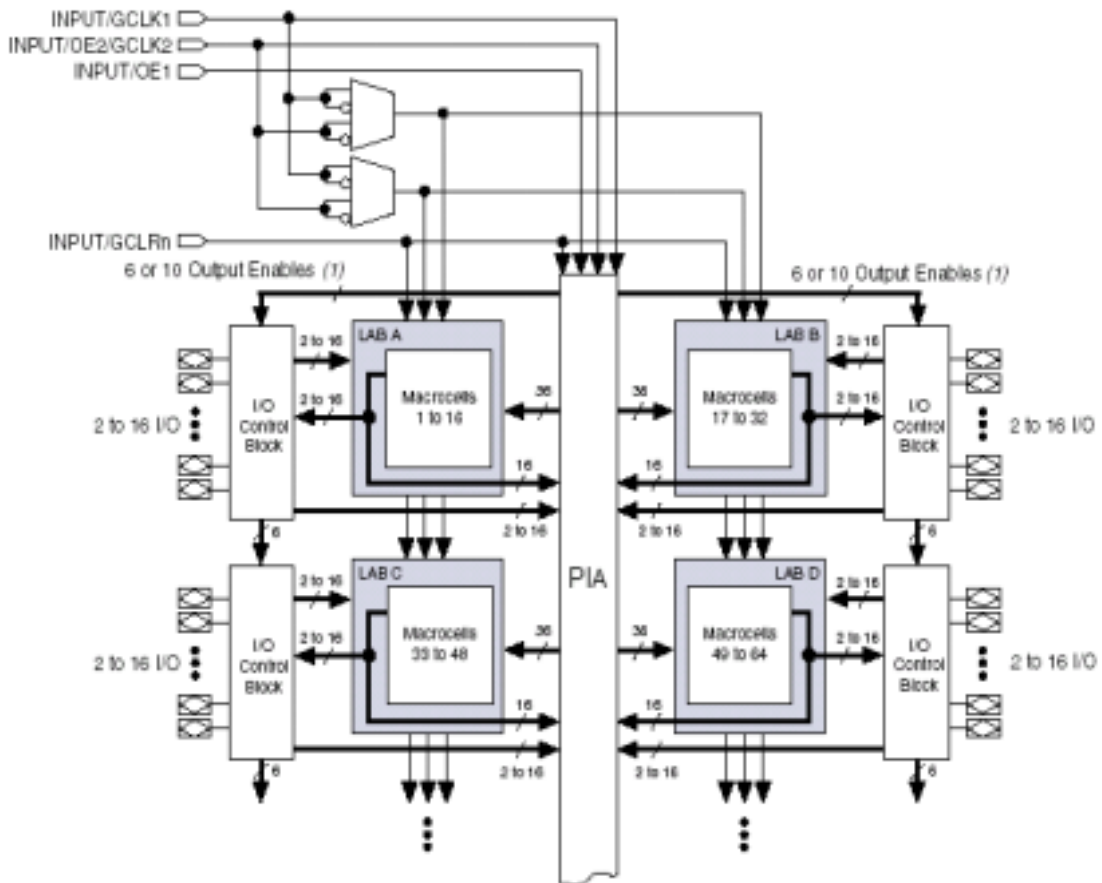
<b>Company</b>	<b>General Architecture</b>	<b>Logic Block Type</b>	<b>Programming Technology</b>
<b>Xilinx</b>	Symmetrical Array	Look-up Table	Static RAM
<b>Actel</b>	Row-based	Multiplexer-Based	anti-fuse
<b>Altera</b>	Hierarchical-PLD	PLD Block	EPROM
<b>Plessey</b>	Sea-of-gates	NAND-gate	Static RAM
<b>Plus</b>	Hierarchical-PLD	PLD Block	EPROM
<b>AMD</b>	Hierarchical-PLD	PLD Block	EEPROM
<b>QuickLogic</b>	Symmetrical Array	Multiplexer-Based	anti-fuse
<b>Algotronix</b>	Sea-of-gates	Multiplexers & Basic Gates	Static RAM
<b>Concurrent</b>	Sea-of-gates	Multiplexers & Basic Gates	Static RAM
<b>Crosspoint</b>	Row-based	Transistor Pairs & Multiplexers	anti-fuse

Sommario delle FPGA disponibili commercialmente della loro architettura e della tecnologia utilizzata per la programmazione



1. **FPGA della Xilinx:** l'architettura generale delle FPGA della Xilinx è formata da un array bidimensionale di blocchi programmabili chiamati Configurable Logic Block (CLB), con percorsi di routing orizzontali tra le righe di blocchi e percorsi di routing verticali tra le colonne di blocchi. La programmazione è controllata da celle di RAM statica.
2. **FPGA della Actel:** l'architettura base delle FPGA della ACTEL consiste di righe di blocchi programmabili, chiamati Logic Modules (LM), con percorsi di routing orizzontali tra le righe. Ogni switch di routing in queste FPGA è implementato per mezzo della tecnica PLICE anti-fuse.
3. **FPGA della Altera:** le FPGA della Altera sono considerevolmente differenti dalle altre incontrate finora perché esse rappresentano un raggruppamento gerarchico di PLD. Nondimeno, esse sono FPGA in quanto utilizzano un array bidimensionale di blocchi programmabili e una struttura di routing programmabile e infine sono programmabili dall'utente. L'architettura generale della Altera, che è basata su una tecnologia di programmazione a EPROM, è formata da un array di blocchi programmabili, chiamati Logic Array Block (LAB), interconnessi da una rete di connessioni chiamata Programmable Interconnect Array (PIA).
4. **FPGA della Plessey:** le FPGA della Plessey sono chiamate Electronically Reconfigurable Array. Queste utilizzano la tecnologia di programmazione a RAM e sono formate da un array bidimensionale regolare di blocchi logici con sovrapposta ad una rete di interconnessioni. Questo tipo di architettura assomiglia all'architettura di tipo sea of gate. Ogni blocco logico è relativamente semplice, contiene un multiplexer 8 a 2 le cui uscite vanno in una porta NAND, e un latch trasparente. Il multiplexer è controllato da un blocco di RAM statica ed è utilizzato per connettere i blocchi logici alle risorse di routing.

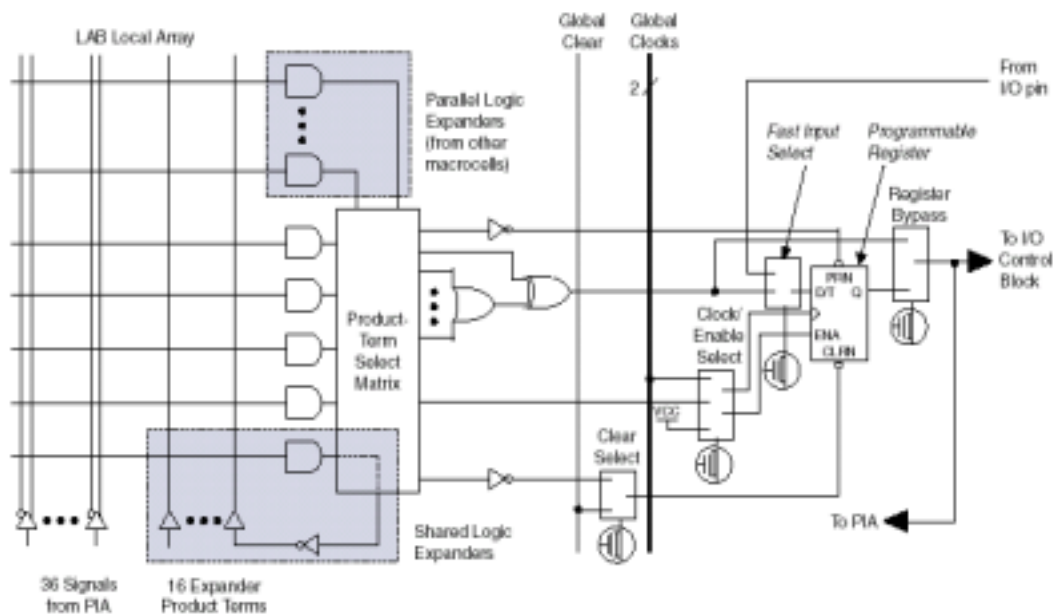
# Altera FPGA



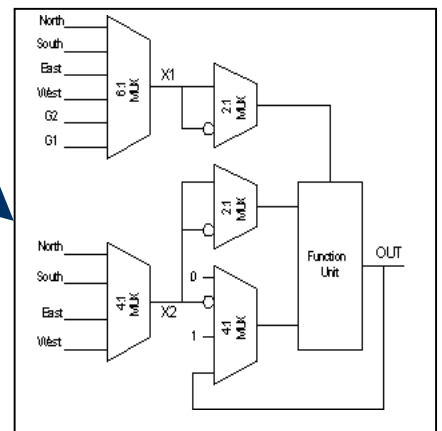
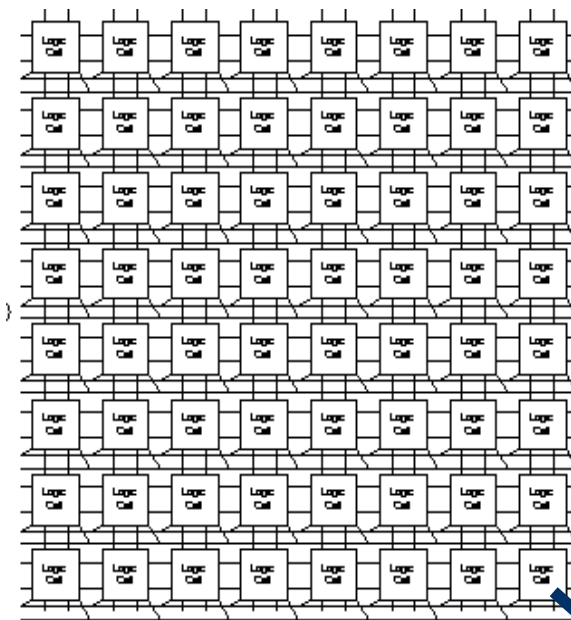


# Altera MAX 7000 LAB

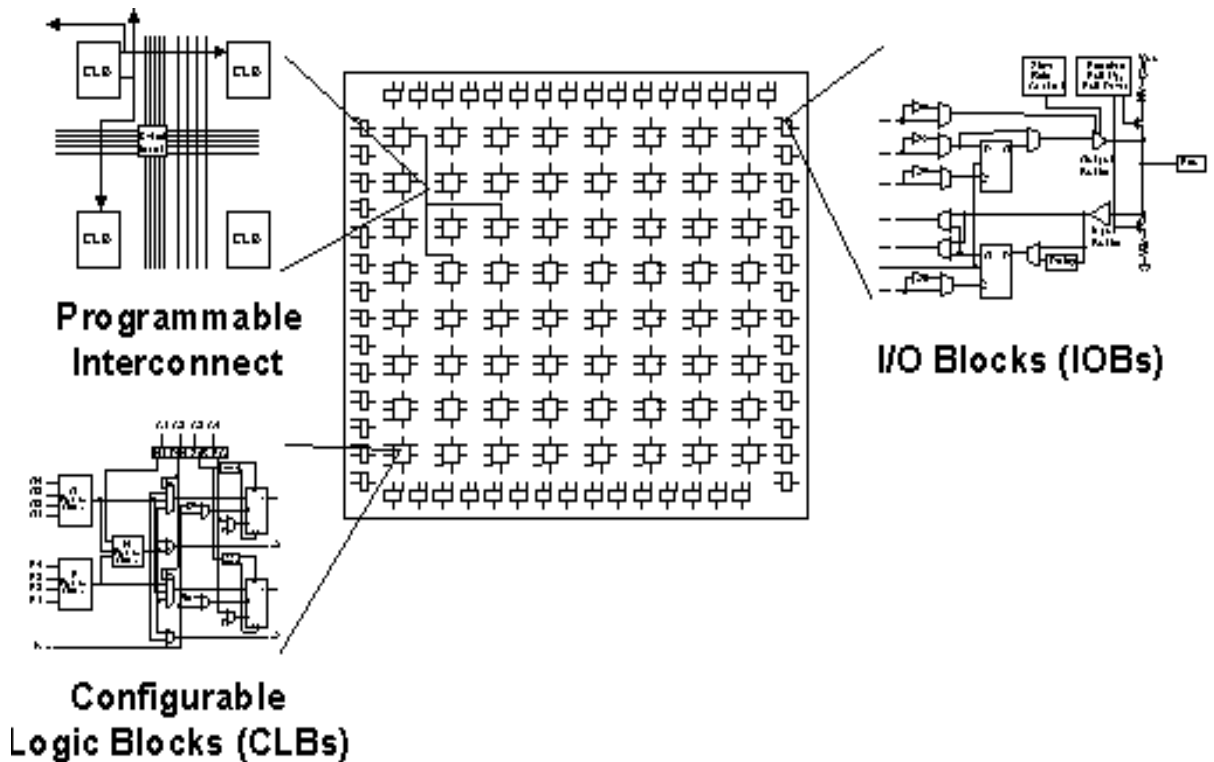
Figure 2. MAX 7000A Macrocell



# Algotronix FPGA: architettura

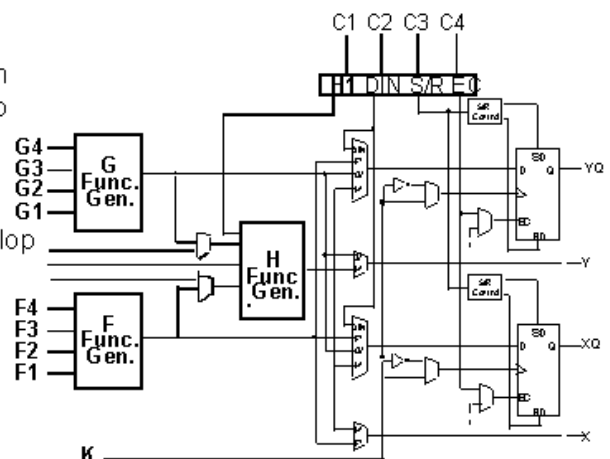


## Xilinx FPGA: architettura



### XC4000 Configurable Logic Blocks

- 2 Four-input function generators (Look Up Tables)
- 2 Registers
  - Each can be configured as Flip Flop or Latch
  - Independent clock polarity
  - Synchronous and asynchronous Set/Reset



# Entity e Architecture

L'esempio di codice riportato sotto è la descrizione VHDL di un comparatore di uguaglianza a quattro bit. Il codice è diviso in due sezioni: entity declaration e architecture body.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
--eqcomp4 è un comparatore di uguaglianza a 4 bit
ENTITY eqcomp4 IS
    PORT( a,b: IN bit_vector(3 DOWNT0 0);
          equals: OUT bit );
END eqcomp4;
ARCHITECTURE dataflow OF eqcomp4 IS
BEGIN
    equals<= '1' WHEN (a = b) ELSE '0';
END dataflow;
```

## Simboli:

-- è il simbolo che indica una riga di commento;

<= è il simbolo utilizzato per indicare l'assegnamento di un segnale.

Un entity è un'astrazione di un dispositivo che rappresenta un sistema completo, una scheda, un chip, una funzione o una porta logica. Una dichiarazione di entity descrive l'I/O di un progetto e può includere anche parametri utilizzati per customizzare l'entity stessa.

Un'architettura descrive le funzioni di un'entity. Un'architettura può contenere qualunque combinazione dei seguenti tipi di descrizione:

- Behavioural;
- Structural;
- Dataflow.

# Descrizione comportamentale (behavioral)

Il listato riportato sotto è un esempio di descrizione comportamentale.

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity eqcomp4 is port(
4     a, b:    in std_logic_vector(3 downto 0);
5     equals:  out std_logic);
6 end eqcomp4;
7
8 architecture behavioral of eqcomp4 is
9 begin
10 comp: process (a, b)
11     begin
12         if a = b then
13             equals <= '1';
14         else
15             equals <= '0';
16         end if;
17     end process comp;
18 end behavioral;
```

Viene chiamata descrizione comportamentale per il modo algoritmico in cui viene descritta l'architettura. La descrizione comportamentale viene indicata, a volte, come descrizione ad alto livello a causa dell'assomiglianza con i linguaggi di programmazione ad alto livello. Piuttosto che specificare la struttura o la netlist di un circuito, è possibile specificare un insieme di istruzioni che, quando eseguite in sequenza, descrivono il funzionamento, o il comportamento, dell'entità, o di parte di essa. Il vantaggio di una descrizione ad alto livello è che non è necessario focalizzarsi sull'implementazione a livello di gate.

# Descrizione di tipo dataflow

Il listato riportato sotto è un esempio di descrizione di tipo dataflow di un comparatore a quattro bit.

```
-- eqcomp4 is a four bit equality comparator
library ieee;
use ieee.std_logic_1164.all;
entity eqcomp4 is
port (a, b:    in std_logic_vector(3 downto 0);
      equals:  out std_logic);
end eqcomp4;

architecture dataflow of eqcomp4 is
begin
    equals <= '1' when (a = b) else '0';    -- equals is active high
end dataflow;
```

Questa è un'architettura di tipo dataflow perché specifica come i dati vengono trasferiti da segnale a segnale e da ingresso a uscita senza ricorrere all'utilizzo di istruzioni sequenziali. La differenza principale con la descrizione di tipo behavioral è che una utilizza i processi e l'altra no. Però entrambe le descrizioni non sono di tipo strutturale.

Conviene utilizzare le descrizioni di tipo dataflow nel caso in cui è possibile scrivere semplici equazioni, istruzioni di assegnamento condizionato (WHEN-ELSE), o istruzioni di assegnamento selettivo (WITH-SELECT-WHEN), piuttosto che algoritmi completi.

D'altra parte, quando è necessario utilizzare strutture annidate, sono preferibili le istruzioni sequenziali.

# Descrizione strutturale

Il listato sotto riporta la descrizione strutturale di un comparatore a quattro bit.

```

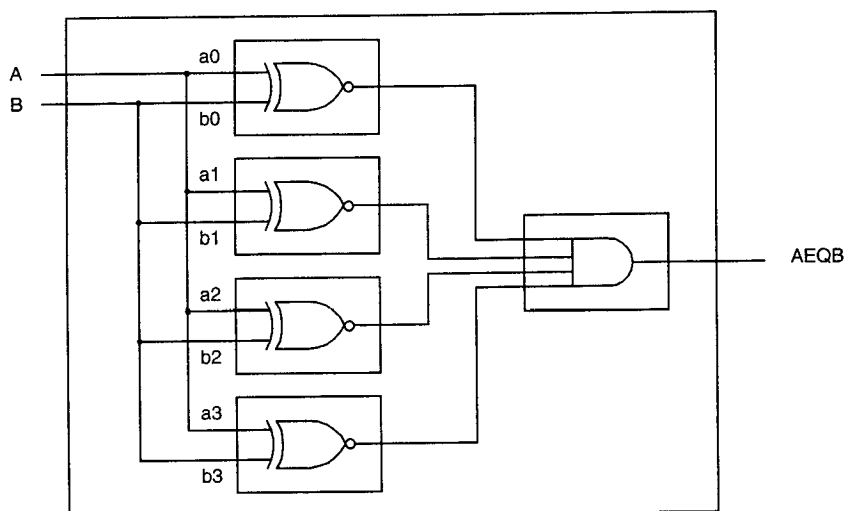
library ieee;
use ieee.std_logic_1164.all;
entity eqcomp4 is port(
    a, b:   in std_logic_vector(3 downto 0);
    aeqb:   out std_logic);
end eqcomp4;

use work.gatespkg.all;
architecture struct of eqcomp4 is
    signal x : std_logic_vector(0 to 3);
begin
    u0: xnor2 port map (a(0),b(0),x(0));
    u1: xnor2 port map (a(1),b(1),x(1));
    u2: xnor2 port map (a(2),b(2),x(2));
    u3: xnor2 port map (a(3),b(3),x(3));
    u4: and4 port map (x(0),x(1),x(2),x(3),equals);
end struct;

```

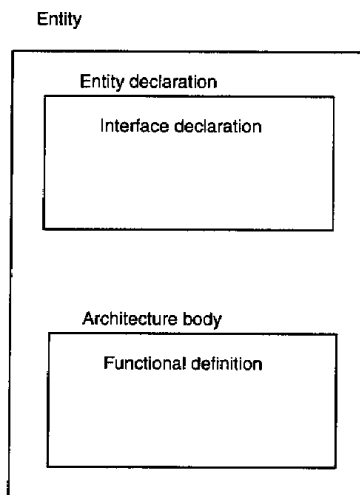
La descrizione strutturale consiste di una netlist VHDL. Questa netlist è molto simile alla netlist di uno schematico: I componenti sono elencati e connessi insieme mediante segnali. I progetti strutturali sono di tipo gerarchico.

Nella figura (1) è riportato lo schematico del comparatore a quattro bit descritto nel listato precedente.



# Dichiarazione di Entity

Una dichiarazione di entity descrive gli ingressi e le uscite del progetto. Può anche descrivere parametri costanti. La dichiarazione di entity è analoga al simbolo di uno schematico che descrive le connessioni di un componente con il resto del progetto.



Figura(2): relazione tra un progetto, la dichiarazione di entità e l'architecture body.

```

entity add4 is port(
    a, b: in std_logic_vector(3 downto 0);
    ci:   in std_logic;
    sum:  out std_logic_vector(3 downto 0);
    co:   out std_logic);
end add4;
  
```

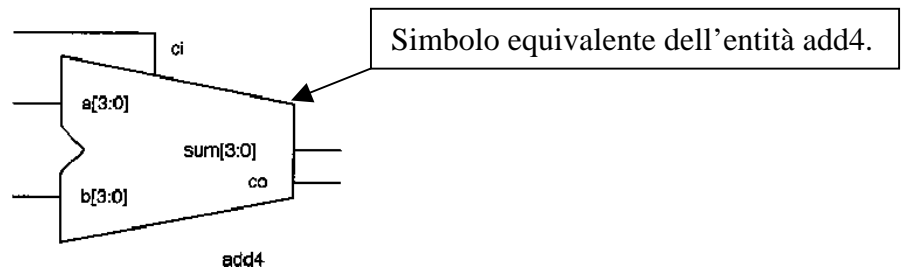


Figura (3): dichiarazione di entity e simbolo del sommatore a quattro bit.



# Porte

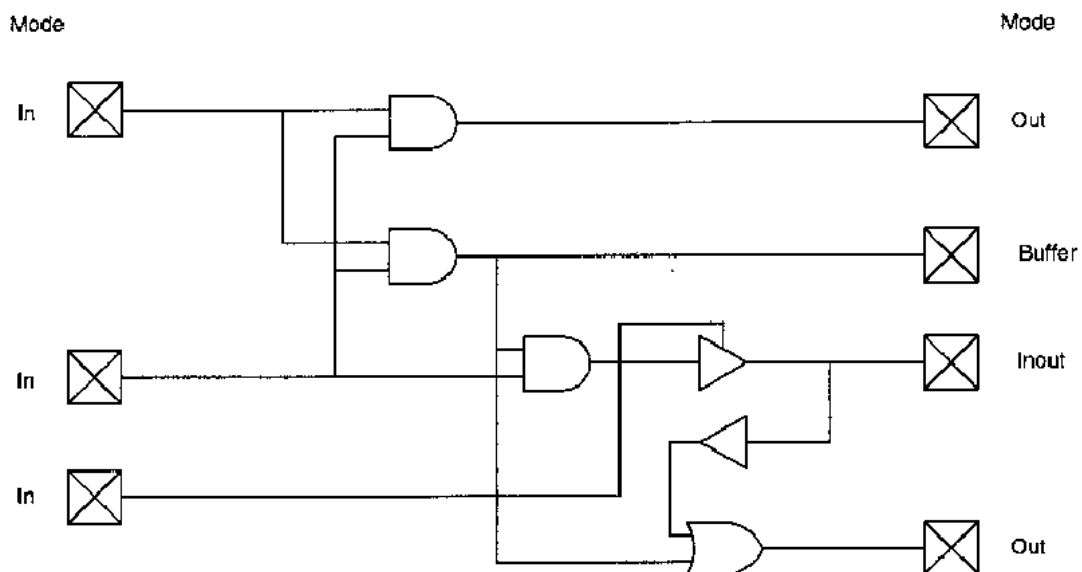
Ogni segnale di I/O in una dichiarazione di entity è chiamato **porta**, ed è analogo ad un pin nel simbolo di uno schematico. Ad una porta possono essere assegnati valori o essere usata all'interno delle espressioni.

L'insieme delle porte definite in una entity è chiamato dichiarazione delle porte. Ogni porta che viene dichiarata deve avere un nome, una direzione (**modo**) e un data type.

**MODI:** i modi descrivono la direzione in cui un dato è trasferito attraverso la porta. Il modo può assumere uno dei quattro valori: IN, OUT, INOUT e BUFFER. Se il modo non viene specificato allora la porta di default è in modo IN.

L'utilizzo dei modi è il seguente:

- **IN.** I dati fluiscono solo dentro l'entità. Il driver per una porta di modo IN è al di fuori dell'entità. Il modo IN viene utilizzato principalmente per gli ingressi di clock, gli ingressi di controllo (come load, reset e enable) e per gli ingressi unidirezionali dei dati.
- **OUT.** I dati fluiscono solo verso le porte di uscita dell'entità. Il driver per una porta di modo OUT è contenuto all'interno dell'entità. Il modo OUT non permette la retroazione perché tale porta non è considerata leggibile dall'interno dell'entità. Il modo OUT è utilizzato per le uscite.
- **BUFFER.** Per utilizzare internamente la retroazione (cioè per utilizzare una porta come driver all'interno dell'architettura), è necessario dichiarare la porta in modalità BUFFER, o dichiarare un segnale separato da utilizzare all'interno dell'architettura come segnale interno. Una porta che viene dichiarata in modalità BUFFER è simile ad una porta che è dichiarata come OUT, eccetto che permette di effettuare internamente una retroazione. La modalità BUFFER non consente ad una porta di essere bidirezionale perché non consente alla porta di essere pilotata dall'esterno dell'entità. Vanno fatte due considerazioni aggiuntive sull'utilizzo della modalità BUFFER: (1) una porta in modalità BUFFER non consente di essere pilotata da più driver; (2) una porta in modalità BUFFER può essere connessa solo ad un segnale interno o ad un'altra porta di modalità BUFFER di un'altra entità. Non può essere connessa a nessun'altra porta in modalità OUT o INOUT di un'altra entità, eccetto che passando attraverso un segnale interno. La modalità BUFFER è utilizzata per porte che devono essere leggibili all'interno dell'entità.
- **INOUT.** Per segnali bidirezionali è necessario dichiarare le porte di modo INOUT, questa modalità permette ai dati di fluire dentro o fuori dall'entità. In altre parole, il driver del segnale può essere all'interno o all'esterno dell'entità. Il modo INOUT permette anche le retroazioni interne. Il modo INOUT è in grado di sostituire tutti gli altri modi. Cioè, IN, OUT e BUFFER possono essere sostituiti dal modo INOUT (questa opzione è sconsigliabile in quanto riduce molto la leggibilità del codice del progetto).



Figura(3): modalità delle porte e le loro possibilità di collegamento con l'interno e l'esterno dell'entità.

## Tipi associati alle porte

Oltre a specificare i nomi e i modi delle porte è necessario anche dichiarare il tipo delle porte. I tipi più utili e meglio supportati per la sintesi forniti dal package IEEE std\_logic\_1164 sono i tipi STD\_LOGIC e gli array derivati da questi. Come implica il nome, *standard logic* vuole essere un tipo standard utilizzato per descrivere i circuiti. La dichiarazione dei tipi deve essere resa visibile all'entità per mezzo delle istruzioni LIBRARY e USE, come riportato del listato sotto:

```
library ieee;
use ieee.std_logic_1164.all;
entity add4 is port(
    a, b: in std_logic_vector(3 downto 0);
    ci:   in std_logic;
    sum:  out std_logic_vector(3 downto 0);
    co:   out std_logic);
end add4;
```

Il VHDL è un linguaggio fortemente tipizzato, cioè i dati di differenti tipi base non possono essere assegnati gli uni agli altri senza utilizzare una funzione di conversione di tipo. Di seguito è riportata una descrizione di alcuni dei tipi che più comunemente si possono incontrare.

**Tipi scalari.** I tipi scalari possiedono un ordine che permette l'applicazione di operatori relazionali tra loro. Ci sono tre principali categorie di operatori scalari: enumerativi, interi, floating.

- **Tipi enumerativi.** Un tipo enumerativo è una lista di valori che un oggetto di un dato tipo può assumere. E' possibile definire la lista dei valori. I tipi enumerativi sono utilizzati spesso per definire lo stato delle macchine sequenziali:

**TYPE states IS (idle, preamble, data, jam, nofsd, error);**

Un segnale può essere definito del tipo enumerativo appena dichiarato:

**SIGNAL current\_state: states;**

Come tipo scalare, il tipo enumerativo è ordinato. L'ordine in cui i valori sono elencati nella dichiarazione del tipo, definiscono la loro relazione. Il valore a sinistra è il più piccolo di tutti gli altri valori. Ogni valore è più grande di quello a sinistra e più piccolo di quello a destra.

Ci sono altri due tipi enumerativi, predefiniti dallo standard IEEE 1076, che sono particolarmente utili per la sintesi: BIT e BOOLEAN. Essi sono definiti come segue:

**TYPE boolean IS (FALSE, TRUE);**

**TYPE bit IS ('0','1');**

Lo standard IEEE 1164 definisce un tipo addizionale e diversi sottotipi che sono utilizzati consistentemente come standard sia per la simulazione che per la sintesi. Il tipo STD\_LOGIC è definito come segue:

**TYPE std\_logic IS ( 'U', -- Uninitialized**

**'X', -- Forcing Unknown**

**'0', -- Forcing 0**

**'1', -- Forcing 1**

**'Z', -- High impedance**

**'W', -- Weak Unknown**

**'L', -- Weak 0**

**'H', -- Weak 1**

**'-' , -- Don't care );**

I valori '0', '1', 'L' e 'H' sono valori logici che sono supportati dalla sintesi. I valori 'Z' e '-' sono anch'essi supportati dalla sintesi per i driver three state e per i valori don't care. I valori 'U', 'X' e 'W' non sono supportati dalla sintesi.

Nella maggior parte dei progetti viene utilizzato il tipo STD\_LOGIC. E' più versatile del tipo BIT perché fornisce il valore di alta impedenza 'Z' e il valore don't care '-'. Lo standard IEEE 1164 definisce array di STD\_LOGIC come STD\_LOGIC\_VECTOR.

Una nota finale sul tipo STD\_LOGIC: sebbene gli identificatori non sono case-sensitive, l'interpretazione dei valori letterali come 'Z' e 'L' è case-sensitive. Questo significa che 'z' **non** è equivalente a 'Z'.

• **Tipi interi.** Nel VHDL sono predefiniti gli interi e gli operatori relazionali ed aritmetici sugli interi. Gli strumenti software che processano il VHDL devono supportare gli interi nel range da  $-2.147.483.647$ ,  $-(2^{31}-1)$  a  $2.147.483.647$ ,  $(2^{31}-1)$ . Un segnale o una variabile che è di tipo intero e che deve essere sintetizzata nel circuito logico deve essere limitata in valore cioè:

**VARIABLE a: INTEGER RANGE -255 TO 255;**

Non tutti i sistemi di sintesi per logiche programmabili sono in grado di trattare valori con segno.

•**Tipo floating.** I valori del tipo floating sono utilizzati per approssimare i numeri reali. Il solo tipo floating point predefinito è il REAL, che include al minimo il range tra  $-1.0E38$  e  $+1.0E38$ . I tipi floating spesso non sono supportati dai programmi di sintesi (in particolare quelli per logiche programmabili) a causa della grande quantità di risorse richieste per l'implementazione delle operazioni aritmetiche tra loro.

**Tipi composti.** Ai tipi scalari può essere assegnato solo un valore. I tipi composti possono invece assumere più valori contemporaneamente. I tipi composti sono i tipi ARRAY e i tipi RECORD.

•**Tipi array.** Un oggetto di tipo array consiste di elementi multipli dello stesso tipo. I tipi di array più comunemente usati sono quelli predefiniti dagli standard IEEE 1076 e 1164:

```
type bit_vector is array (natural range <>) of bit;
type std_ulogic_vector is array (natural range <>) of std_ulogic;
type std_logic_vector is array (natural range <>) of std_logic;
```

Questi tipi sono comunemente utilizzati per indicare i bus.

Quando si utilizzano le stringhe di bit, è possibile utilizzare una notazione opportuna per indicare se i bit delle stringhe sono specificati in binario, in ottale o in esadecimale. Se tuttavia la stringa è specificata in binario può essere assegnata solo ad un oggetto di tipo BIT\_VECTOR e non ad un oggetto di tipo STD\_LOGIC\_VECTOR. Per i formati ottale ed esadecimale occorre trasformare la stringa binaria nel formato desiderato ad esempio:

```
a <= X"7A";
```

Richiede che **a** sia di 8 bit, dove **a** può essere un BIT\_VECTOR o uno STD\_LOGIC\_VECTOR il cui valore diventa "01111010". La notazione per indicare le cifre esadecimali è **X** seguita dalla cifra esadecimale tra virgolette, per le ottali l'indicatore è **O** mentre per quelle binarie è **B**.

•**Tipi record.** Un oggetto di tipo record ha elementi multipli di tipi differenti. I singoli campi possono essere referenziati con il nome dell'elemento, come nell'esempio riportato sotto.

```
type iocell is record
    buffer_inp: bit_vector(7 downto 0);
    enable: bit;
    buffer_out: bit_vector(7 downto 0);
end record;

signal busa, busb, busc: iocell;
signal vec: bit_vector(7 downto 0);

busa.buffer_inp <= vec;           -- one bit_vector assigned
                                -- to another
busb.buffer_inp <= busa.buffer_inp; -- assigning one field;
busb.enable <= '1';
busc <= busb;                   -- assigning entire object
```

# Operatori

logical_operator	::=	and		or		nand		nor		xor		xnor
relational_operator	::=	=		/=		<		<=		>		>=
shift_operator	::=	sll		srl		sla		sra		rol		ror
adding_operator	::=	+		-		&						
sign	::=	+		-								
multiplying_operator	::=	*		/		mod		rem				
miscellaneous_operator	::=	**		abs		not						

Figura (5): tabella degli operatori

## Operatori Logici

AND

OR

NAND

XOR

XNOR

NOT

Sono operatori predefiniti per i tipi BIT e BOOLEAN o per i vettori di BIT o BOOLEAN. Lo standard IEEE 1164 estende questi operatori per il tipo STD\_LOGIC e il suo vettore STD\_LOGIC\_VECTOR.

Questi operatori non hanno un ordine di precedenza: **sono richieste le parentesi**.

~~X <= a OR b AND c~~

**ERRATO**

X <= (a OR b) AND c

X <= a OR (b AND c)

**OK**

# Operatori Relazionali

Gli operatori relazionali sono utilizzati per testare l'uguaglianza, la disuguaglianza e l'ordinamento.

=	<b>Operatore di uguaglianza</b>
/=	<b>Operatore di disuguaglianza</b>
<	<b>Minore</b>
>, >=	<b>Maggiore e maggiore uguale</b>
<, <=	<b>Minore e minore uguale</b>

Gli operatori = e /= sono definiti per tutti i tipi di dati finora incontrati,

Gli operatori >, >=, < e <= sono definiti per i tipi scalari o gli array con un range di valori discreto.

Gli array sono uguali solo se le loro lunghezze sono equivalenti e tutti gli elementi corrispondenti sono uguali.

Il risultato degli operatori relazionali è booleano (cioè, è vero o falso).

I tipi di operandi in un'operazione relazionale devono essere uguali.

Operator	Operation	Operand type	Result type
=	Equality	Any type, other than a file type or a protected type	BOOLEAN
/=	Inequality	Any type, other than a file type or a protected type	BOOLEAN
< <= > >=	Ordering	Any scalar type or discrete array type	BOOLEAN

Figura(6): tabella degli operatori relazionali

# WITH-SELECT-WHEN

Assegnamento selettivo dei segnali:

WITH selection\_signal SELECT

```

    signal_name <= value_a WHEN value_1_of_selection_signal,
                        value_b WHEN value_2_of_selection_signal,
                        value_c WHEN value_3_of_selection_signal,
                        .....
                        value_x WHEN last_value_of_selection_signal;
  
```

Al segnale *signal\_name* viene assegnato un valore in base al valore corrente del segnale *selection\_signal*. Tutti i valori di *selection\_signal* devono essere elencati nell'istruzione WHEN e tali valori sono mutuamente esclusivi.

Esempio:

```

library ieee;
use ieee.std_logic_1164.all;
entity mux is port(
    a, b, c, d:    in std_logic_vector(3 downto 0);
    s:             in std_logic_vector(1 downto 0);
    x:             out std_logic_vector(3 downto 0));
end mux;

architecture archmux of mux is
begin
  with s select
    x <= a when "00",
         b when "01",
         c when "10",
         d when others;
end archmux;
  
```

La parola riservata OTHERS viene utilizzata per indicare tutti gli altri possibili valori della variabile *s*.



# WHEN-ELSE

Assegnamento condizionato dei segnali, cioè ad un segnale viene assegnato un valore in base al verificarsi di una determinata condizione.

```

signal_name <= value_a WHEN condition1 ELSE
                value_b WHEN condition2 ELSE
                value_c WHEN condition3 ELSE
                .....
                value_x

```

Al segnale *signal\_name* viene assegnato il valore in base alla verifica di una delle condizioni elencate.

Esempio:

```

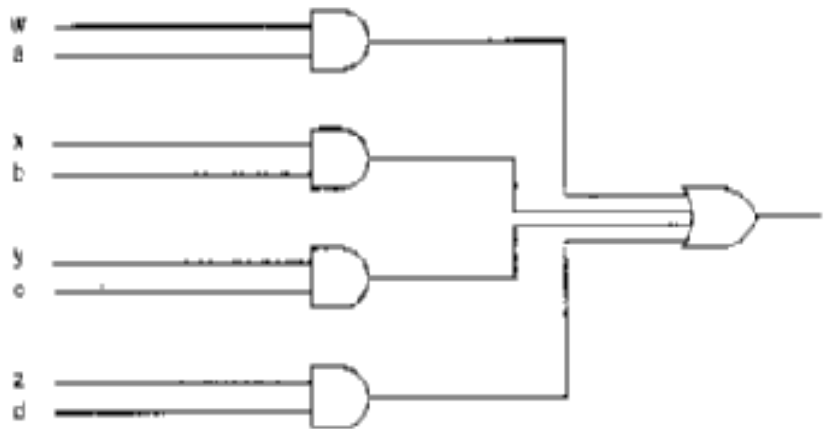
library ieee;
use ieee.std_logic_1164.all;
entity mux is port(
    a, b, c, d:      in std_logic_vector(3 downto 0);
    s:               in std_logic_vector(1 downto 0);
    x:               out std_logic_vector(3 downto 0));
end mux;

architecture archmux of mux is
begin
    x <= a when (s = "00") else
        b when (s = "01") else
        c when (s = "10") else
        d;
end archmux;

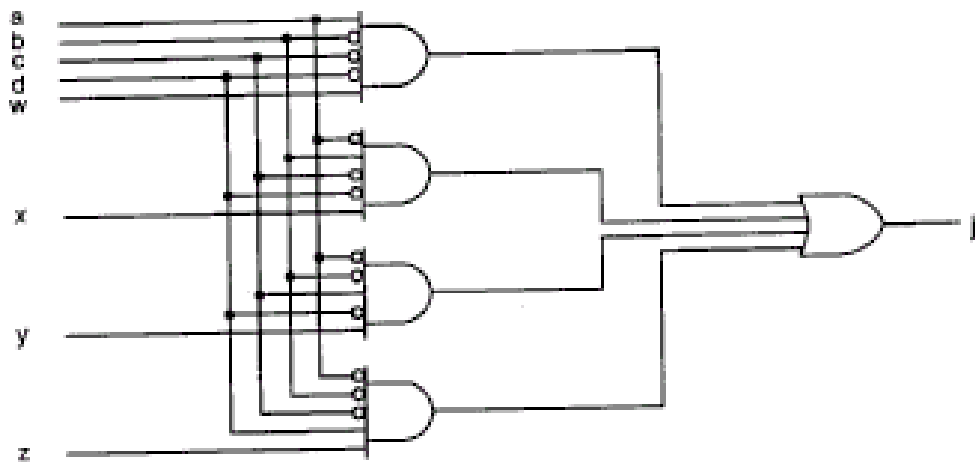
```

Mentre la condizione WHEN nell'istruzione WITH-SELECT-WHEN deve specificare valori mutuamente esclusivi del segnale di selezione, la condizione WHEN in un'istruzione WHEN-ELSE può specificare qualunque semplice espressione.

Se le condizioni in un'istruzione WHEN-ELSE non sono mutuamente esclusive, la priorità più alta viene assegnata alla prima condizione WHEN elencata. Le priorità delle successive condizioni WHEN sono assegnate in base all'ordine di apparizione.



Figura(7): selezione dei segnali w,x,y e z in base ai segnali mutuamente esclusivi a,b,c e d.



Figura(8): selezione dei segnali w,x,y e z in base ai segnali non mutuamente esclusivi a,b,c e d.

# IF-THEN-ELSE

L'istruzione IF-THEN-ELSE è utilizzata per selezionare un insieme di istruzioni da eseguire in base alla valutazione di una condizione o un'insieme di condizioni.

```
IF (condition) THEN
    do something;
ELSE
    do something different;
END IF;
```

Se la condizione specificata viene trovata vera, vengono eseguite le istruzioni che seguono la parola chiave THEN. Se la condizione viene trovata falsa, vengono eseguite le istruzioni dopo ELSE. L'insieme di istruzioni viene chiuso da END IF.

Dato che le istruzioni sequenziali vengono eseguite nell'ordine di apparizione, i seguenti processi sono funzionalmente equivalenti

```
signal step: std_logic;
signal addr: std_logic_vector(7 downto 0);
.
.
.

similar1: process (addr)
begin
    step <= '0';
    if addr > x"0F" then
        step <= '1';
    end if;
end process;

similar2: process (addr)
begin
    if addr > x"0F" then
        step <= '1';
    else
        step <= '0';
    end if;
end process similar2; X
```

Con entrambi i processi, *step* assume il valore '1' se *addr* è maggiore di 0F hex, e '0' se è minore o uguale al valore indicato.

Il processo:

```
not_similar: process (addr)
begin
  if addr > x"0F" then
    step <= '1';
  end if;
end process;
```

non descrive la stessa logica perché non è assegnato né un valore di default né un valore di ELSE al segnale *step*. Il processo *not\_similar* implica che *step* deve mantenere il suo valore se *addr* è minore o uguale a 0F hex. Questa viene chiamata **memoria implicita**. Così, una volta che è stato asserito, *step* rimarrà sempre al valore asserito come mostrato nella figura (9), e definito dalla seguente equazione:

$$step = addr(3)*addr(2)*addr(1)*addr(0) + step$$



Figura(9): memoria implicita.

Se non si vuole che il valore di *step* venga memorizzato è necessario includere un valore di default o completare l'istruzione IF-THEN con un ELSE.

L'istruzione IF-THEN\_ELSE può essere ulteriormente espansa includendo ELSIF per specificare ulteriori condizioni.

La sintassi di questa operazione è

```
if (condition1) then
  do something;
elsif (condition2) then
  do something different;
else
  do something completely different;
end if;
```

# CASE-WHEN

L'istruzione CASE-WHEN viene utilizzata per specificare un insieme di istruzioni da eseguire in base al valore di un segnale selezione. L'istruzione CASE-WHEN può essere utilizzata, per esempio, in modo equivalente all'istruzione WITH-SELECT-WHEN.

La sintassi è

```

case selection_signal is
  when value_1_of_selection_signal =>
    (do something)      --set of statements 1
  when value_2_of_selection_signal =>
    (do something)      --set of statements 2
  when value_3_of_selection_signal =>
    (do something)      --set of statements 3
  ...
  when last_value_of_selection_signal =>
    (do something)      --set of statements x
end case;

```

Di seguito viene riportato un esempio di applicazione dell'istruzione CASE-WHEN alla creazione di un decodificatore di indirizzi.

```

library ieee;
use ieee.std_logic_1164.all;
entity test_case is
  port (address : in std_logic_vector(2 downto 0);
        decode:   out std_logic_vector(7 downto 0));
end test_case;

architecture design of test_case is
begin
  process (address)
  begin
    case address is
      when "001" => decode <= X"11";
      when "111" => decode <= X"42";
      when "010" => decode <= X"44";
      when "101" => decode <= X"88";
      when others => decode <= X"00";
    end case;
  end process;
end design;

```

# Logica Sincrona

## Filp flop di tipo D:

```

library ieee;
use ieee.std_logic_1164.all;
entity dff_logic is port (
    d, clk : in std_logic;
    q       : out std_logic );
end dff_logic;

architecture example of dff_logic is
begin
    process (clk) begin
        if (clk'event and clk = '1') then
            q <= d;
        end if;
    end process;
end example;

```

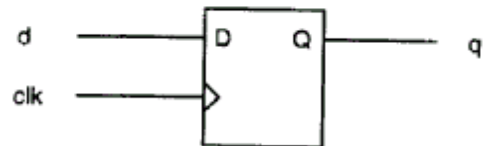


Figura (10): descrizione e simbolo di un flip flop di tipo D che memorizza sul fronte positivo del clock

## Filp flop di tipo T:

```

library ieee;
use ieee.std_logic_1164.all;
entity tff_logic is port (
    t, clk : in std_logic;
    q       : buffer std_logic
);
end tff_logic;

architecture t_example of tff_logic is
begin
    process (clk) begin
        if (clk'event and clk = '1') then
            if (t = '1') then
                q <= not(q);
            else
                q <= q;
            end if;
        end if;
    end process;
end t_example;

```

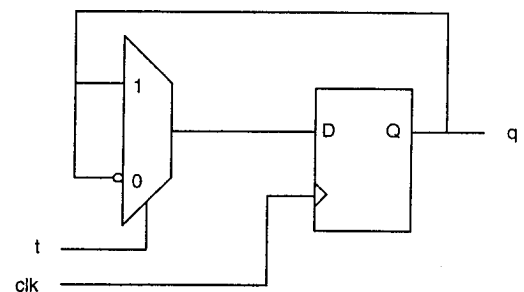


Figura (11): descrizione e simbolo di un flip flop di tipo T (toggle) che memorizza sul fronte positivo del clock

Descrizione VHDL di un registro di 8 bit con memorizzazione del dato sul fronte positivo del clock.

```
library ieee;
use ieee.std_logic_1164.all;
entity reg_logic is port (
    d    : in std_logic_vector(0 to 7);
    clk  : in std_logic;
    q    : out std_logic_vector(0 to 7)
);
end reg_logic;

architecture r_example of reg_logic is
begin
    process (clk) begin
        if (clk'event and clk = '1') then
            q <= d;
        end if;
    end process;
end r_example;
```

# WAIT-UNTIL

E' possibile descrivere il comportamento di un flip flop di tipo D anche mediante l'istruzione WAIT-UNTIL invece di IF (clk' EVENT AND clk=1):

```
architecture example2 of dff_logic is
begin
    process begin
        wait until (clk = '1');
        q <= d;
    end process;
end example2;
```

Questo processo non utilizza una sensitivity list, ma inizia con l'istruzione WAIT. Un processo che utilizza un'istruzione WAIT non può avere una sensitivity list (l'istruzione WAIT definisce implicitamente la sensitivity list). Per i circuiti che devono essere sintetizzati, l'istruzione WAIT-UNTIL deve essere la prima del processo. A causa di questo, la logica sincrona descritta utilizzando l'istruzione WAIT non può essere resettata in modo asincrono.



## Funzioni rising\_edge e falling\_edge

Il package STD\_LOGIC\_1164 definisce le funzioni RISING\_EDGE e FALLING\_EDGE per rilevare i fronti di salita e di discesa dei segnali.

Una di queste funzioni può essere utilizzata per sostituire l'espressione (clk' EVENT AND clk='1') se il segnale clk è di tipo STD\_LOGIC.

Queste funzioni a volte sono preferite dai progettisti perché nelle simulazioni la funzione RISING\_EDGE assicura che la transizione avviene tra '0' e '1' e non qualche altra transizione come ad esempio tra 'U' e '1'.

```
library ieee;
use ieee.std_logic_1164.all;
entity dff_logic is port (
    d, clk : in std_logic;
    q       : out std_logic );
end dff_logic;

architecture example of dff_logic is
begin
    process (clk) begin
        if rising_edge(clk) then
            q <= d;
        end if;
    end process;
end example;
```

Descrizione di un flip flop di tipo D utilizzando la funzione RISING\_EDGE

# Reset e preset nella logica sincrona

Lo standard VHDL non richiede che un circuito venga inizializzato o resettato. Lo standard specifica che per la simulazione, a meno che un segnale non sia esplicitamente inizializzato, un segnale viene inizializzato al valore *'left'* del suo tipo. Così, il tipo `STD_LOGIC` verrà inizializzato a *'U'*, e un bit verrà inizializzato a *'0'*.

Nell'hardware, questo non è sempre vero, non tutti i dispositivi si inizializzano nello stato di reset, ma in uno stato privo di significato.

E' possibile descrivere il reset e il preset di un dispositivo mediante una semplice modifica del codice VHDL come segue:

```
architecture rexample of dff_logic is
begin
  process (clk, reset) begin
    if reset = '1' then
      q <= '0';
    elsif rising_edge(clk) then
      q <= d;
    end if;
  end process;
end rexample;
```

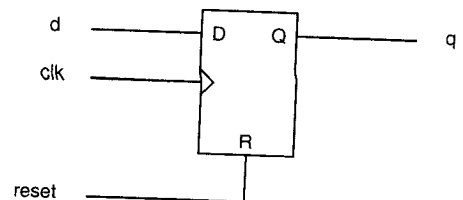


Figura (12): descrizione simbolo di un flip flop di tipo D con reset asincrono

La sensitivity list indica che questo processo è sensibile ai cambiamenti di *clk* e *reset*. Se il segnale di *reset* viene messo a *'1'*, al segnale *q* viene assegnato il valore *'0'* qualunque sia il valore di *clk*.

Per descrivere un preset al posto di un reset, è possibile modificare la sensitivity list e scrivere

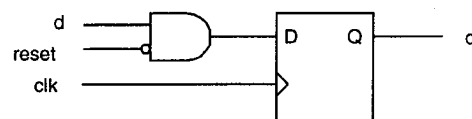
```
if (preset = '1') then
  q <= '1';
elsif rising_edge(clk) then ...
```

invece di

```
if (reset = '1')
  then q <= '0'
elsif rising_edge(clk) then ...
```

E' possibile anche resettare (o presetare) un flip flop in modo sincrono ponendo la condizione di reset (o preset) all'interno della porzione del processo che descrive la logica che è sincrona con il clock, come segue

```
architecture sync_rexample of dff_logic is
begin
  process (clk) begin
    if rising_edge(clk) then
      if (reset = '1') then
        q <= '0';
      else
        q <= d;
      end if;
    end if;
  end process;
end sync_rexample;
```



Descrizione e simbolo di un flip flop di tipo D con reset sincrono.

Generalmente la logica sincrona richiede risorse hardware aggiuntive rispetto a reset e preset asincrono.

In VHDL è possibile anche descrivere una combinazione di reset e/o preset sincroni/asincroni. Per esempio, un registro a 8 bit può essere resettato a 0 ogni volta che il segnale *reset* va a '1', e può essere inizializzato con tutti 1 e caricato dal fronte di salita del clock, come riportato nel listato che segue:

```
library ieee;
use ieee.std_logic_1164.all;
entity reg_logic is port (
  d          : in std_logic_vector(0 to 7);
  reset, init, clk : in std_logic;
  q          : out std_logic_vector(0 to 7) );
end reg_logic;

architecture fancy_example of reg_logic is
begin
  process (clk, reset) begin
    if (reset = '1') then
      q <= b"00000000";
    elsif (clk'event and clk = '1') then
      if (init = '1') then
        q <= b"11111111";
      else
        q <= d;
      end if;
    end if;
  end process;
end fancy_example;
```

Registro a 8 bit con reset asincrono e inizializzazione sincrona

# Buffer three-state e segnali bidirezionali

## Descrizione comportamentale di un buffer tree-state.

I valori che un segnale three-state può assumere sono '0', '1' e 'Z', e sono supportati tutti dal tipo STD\_LOGIC.

L'utilizzo dei buffer three-state viene illustrato mediante l'esempio di un contatore a 8 bit caricabile:

```

library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity cnt8 is port(
    txclk, grst: in std_logic;
    enable, load: in std_logic;
    oe:          in std_logic;          --output enable
    data:        in std_logic_vector(7 downto 0);
    cnt_out:     buffer std_logic_vector(7 downto 0)); -- cnt output
end cnt8;

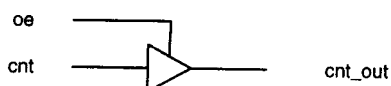
architecture archcnt8 of cnt8 is
    signal cnt: std_logic_vector(7 downto 0); -- cnt signal for counting
begin
    count: process (grst, txclk)
    begin
        if grst = '1' then
            cnt <= "00111010";
        elsif rising_edge(txclk) then
            if load = '1' then
                cnt <= data;
            elsif enable = '1' then
                cnt <= cnt + 1;
            end if;
        end if;
    end process count;

    oes: process (oe, cnt)
    begin
        if oe = '0' then
            cnt_out <= (others => 'Z');
        else
            cnt_out <= cnt;
        end if;
    end process oes;
end archcnt8;

```

Il processo chiamato *oes* è utilizzato per descrivere l'uscita three-state del contatore. Questo processo indica semplicemente che se *oe* viene messo a '1', il valore di *cnt* viene assegnato a *cnt-out* e se *oe* viene messo a '0', l'uscita di questo dispositivi viene posta in alta impedenza. Il processo è sensibile al cambiamento di entrambe le variabili *oe* e *cnt*, perché un cambiamento in entrambi i segnali provoca un cambiamento nell'uscita *cnt\_out*.

Il processo *oes* descrive il comportamento di un buffer three-state.



Il controllo dei buffer three-state può essere descritto anche con l'istruzione WHEN-ELSE. Nel listato che segue, che è una versione modificata del contatore a 8 bit, è stata aggiunta un'uscita aggiuntiva, *collision*, che viene messa a '1' quando i segnali *load* e *enable* sono contemporaneamente posti a '1'.

```

library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity cnt8 is port(
    txclk, grst:    in std_logic;
    enable, load:   in std_logic;
    oe:             in std_logic;
    data:           in std_logic_vector(7 downto 0);
    collision:       out std_logic;           -- 3-state out
    cnt_out:        buffer std_logic_vector(7 downto 0));
end cnt8;

architecture archcnt8 of cnt8 is
    signal cnt: std_logic_vector;
begin
    count: process (grst, txclk)
    begin
        if grst = '1' then
            cnt <= "00111010";
        elsif rising_edge(txclk) then
            if load = '1' then
                cnt <= data;
            elsif enable = '1' then
                cnt <= cnt + 1;
            end if;
        end if;
    end process count;
    -- three-state outputs described here:
    cnt_out <= (others => 'Z') when oe = '0' else cnt;
    collision <= (enable and load) when oe = '1' else 'Z';
end archcnt8;

```

## Segnali bidirezionali.

I segnali bidirezionali possono essere descritti con una piccola modifica dei listati del contatore visti in precedenza.

```

library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity cnt8 is port(
    txclk, grst:    in std_logic;
    enable, load:   in std_logic;
    oe:            in std_logic;
    cnt_out:       inout std_logic_vector(7 downto 0));    -- inout req'd
end cnt8;

architecture archcnt8 of cnt8 is
    signal cnt: std_logic_vector(7 downto 0);
begin
    count: process (grst, txclk)
    begin
        if grst = '1' then
            cnt <= "00111010";
        elsif (txclk'event and txclk='1') then
            if load = '1' then
                cnt <= cnt_out;    -- cnt now loaded from the cnt_out port
            elsif enable = '1' then
                cnt <= cnt + 1;
            end if;
        end if;
    end process count;

    oes: process (oe, cnt)
    begin
        if oe = '0' then
            cnt_out <= (others => 'Z');
        else
            cnt_out <= cnt;
        end if;
    end process oes;
end archcnt8;

```

In questo caso il contatore viene caricato con il valore corrente sui pin associati con l'uscita del contatore, questo significa che il valore caricato quando *load* viene posto a '1' potrebbe essere il valore precedente del contatore oppure un valore posto sui pin da un altro dispositivo, a seconda dello stato di *oe*.

# FOR-GENERATE

Se si volesse utilizzare un componente chiamato *threestate*, precedentemente creato, per implementare un buffer three-state per un bus di 32 bit, al posto di istanziare 32 buffer è possibile utilizzare l'istruzione FOR-GENERATE come segue:

```
gen_label:
  for i in 0 to 31 generate
    inst_label:threestate port map (value(i), read, value_out(i));
  end generate;
```

Questo tipo di istruzione viene implementato nella parte delle istruzioni concorrenti di un'architettura, non all'interno di un *process*. L'istruzione FOR-GENERATE richiede una label; in questo caso *gen\_label*.

```
g1:  for i in 0 to 7 generate
      u0t7:  threestate port map (val(i), byte_rd(0), val_out(i));
      u8t15: threestate port map (val(i+8), byte_rd(1), val_out(i+8));
      u16t23: threestate port map (val(i+16), byte_rd(2), val_out(i+16));
      u24t31: threestate port map (val(i+24), byte_rd(3), val_out(i+24));
    end generate;
```

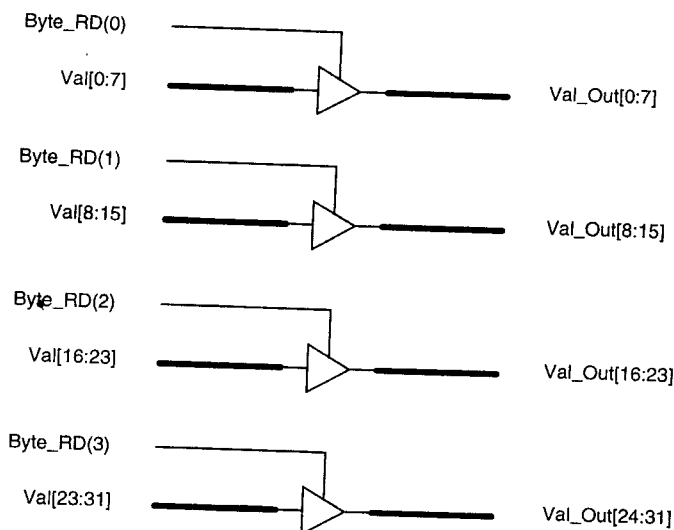


Figura (14): esempio buffer three state con output enable per ogni byte.

E' possibile utilizzare schemi più complicati dell'istruzione FOR-GENERATE in cui sono presenti assegnamenti di tipo condizionato, come ad esempio:

```

g1:  for i in 0 to 3 generate
      g2:  for j in 0 to 7 generate
            if i < 1 then generate
              ua: thrst port map(val(j), byte_rd(0), val_out(j));
            end generate;
            if i = 1 then generate
              ub: thrst port map (val(j+8), byte_rd(1), val_out(j+8));
            end generate;
            if i = 2 then generate
              uc: thrst port map (val(j+16), byte_rd(2), val_out(j+16));
            end generate;
            if i > 2 then generate
              ud: thrst port map (val(j+24), byte_rd(3), val_out(j+24));
            end generate;
          end generate;
      end generate;

```

Lo il listato riportato sopra include l'istruzione IF-THEN. Quando viene utilizzata insieme all'istruzione FOR-GENERATE, l'istruzione IF-THEN non può includere un ELSE o un ELSIF.



# Convertire i buffer three-state in multiplexer

Alcune FPGA contengono al loro interno dei buffer three-state, altre no. Tuttavia, i progetti che utilizzano bus three-state possono essere convertiti in progetti che utilizzano i multiplexer. Alcuni strumenti di sintesi possono fare questo automaticamente.

La conversione dei buffer three-state in multiplexer è illustrata nella figura che segue:

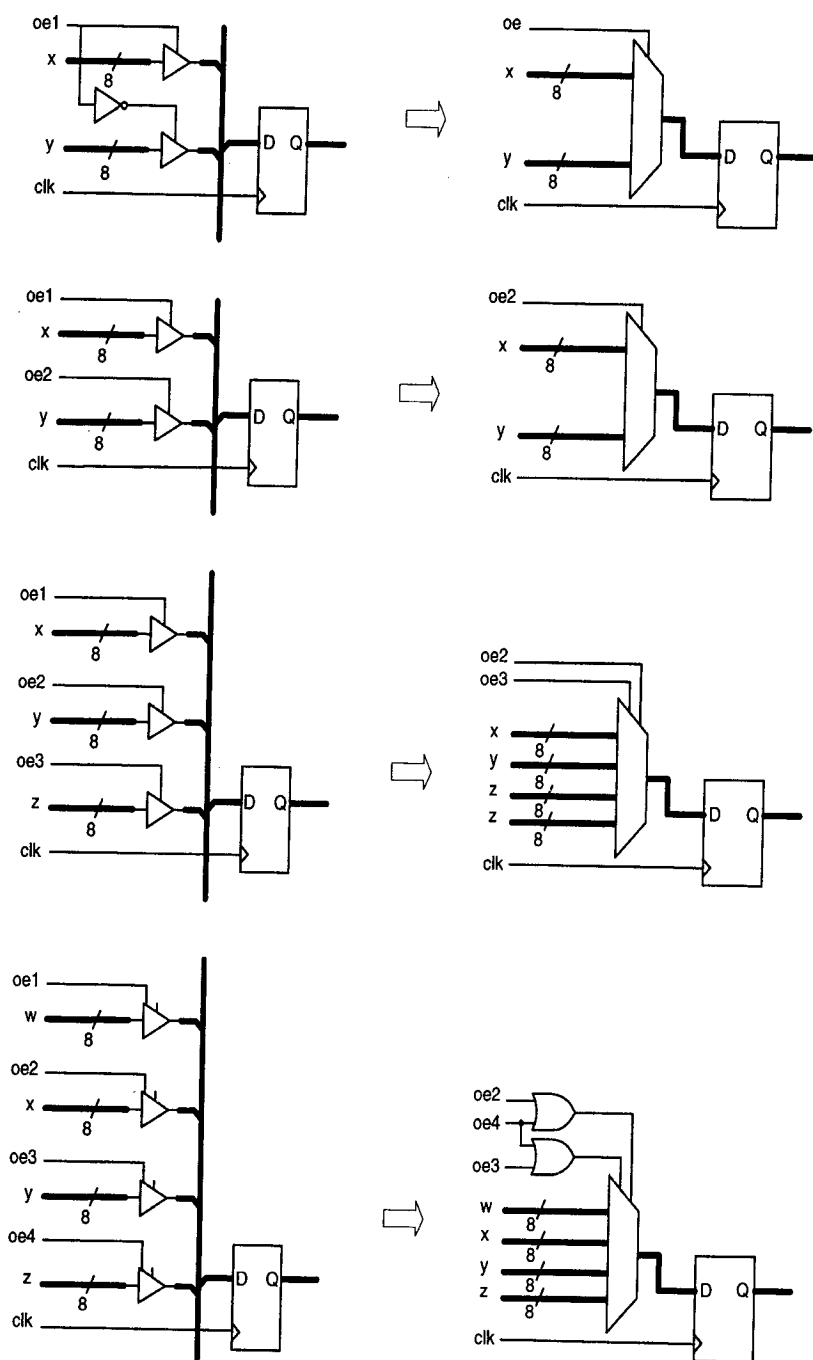


Figura (15): conversione dei buffer three state in multiplexer.

# I cicli

I cicli sono utilizzati per implementare operazioni ripetitive, e possono essere sia cicli FOR che cicli WHILE. Il ciclo FOR esegue un numero predeterminato di iterazioni in base ad un valore di controllo. Il ciclo WHILE continua l'esecuzione di un'operazione finché non si verifica una condizione logica di controllo.

Si consideri per esempio il ciclo che esegue il reset asincrono della FIFO

```
for i in 7 downto 0 loop
    fifo(i) <= (others => '0');
end loop;
```

Questo ciclo che scandisce tutti gli 8 STD\_LOGIC\_VECTOR che compongono la FIFO, inizializza ogni elemento del vettore a '0'. In un ciclo FOR, la variabile del ciclo viene dichiarata automaticamente.

Al posto del ciclo FOR potrebbe essere utilizzato anche un ciclo WHILE, ma questo richiede delle dichiarazioni, delle inizializzazioni e un incremento della variabile di ciclo addizionali, come mostrato nell'esempio sotto:

```
reg_array: process (rst, clk)
    variable i: integer := 0;
begin
    if rst = '1' then
        while i < 7 loop
            fifo(i) <= (others => '0');
            i := i + 1;
        end loop;
    ...
```

## Iterazioni condizionate.

L'istruzione NEXT viene utilizzata per saltare un'operazione in base al verificarsi di una determinata condizione. Si supponga per esempio che quando il segnale *rst* viene posto ad '1', tutti i registri della FIFO siano resettati eccetto che per il registro FIFO(4):

```
reg_array: process (rst, clk)
begin
    if rst = '1' then
        for i in 7 downto 0 loop
            if i = 4 then
                next;
            else
                fifo(i) <= (others => '0');
            end loop;
    ...
```

Oppure può essere scritto utilizzando un ciclo WHILE:

```
reg_array: process (rst, clk)
    variable i: integer;
begin
    i := 0;
    if rst = '1' then
        while i < 8 loop
            if i = 4 then
                next;
            else
                fifo(i) <= (others => '0');
                i := i + 1;
            end loop;
        ...
    end loop;
end process;
```

### Uscita dai cicli.

L'istruzione EXIT viene utilizzata per uscire dai cicli, e può essere utilizzata per testare una condizione illegale. La condizione deve essere verificata all'istante della compilazione. Si supponga, per esempio, che la FIFO sia un componente che viene istanziato in un progetto gerarchico. Si supponga inoltre che la profondità della FIFO venga definita per mezzo di un parametro di tipo GENERIC. Si vuole uscire dal ciclo quando la profondità della FIFO diventa più grande di un valore predeterminato. Ad esempio:

```
reg_array: process (rst, clk)
begin
    if rst = '1' then
        loop1: for i in deep downto 0 loop
            if i > 20 then
                exit loop1;
            else
                fifo(i) <= (others => '0');
            end loop;
        end loop;
    end if;
end process;
```

Il frammento di codice riportato sopra può essere riscritto come:

```
reg_array: process (rst, clk)
begin
    if rst = '1' then
        loop1: for i in deep downto 0 loop
            exit loop1 when i > 20;
            fifo(i) <= (others => '0');
        end loop;
    end if;
end process;
```

# Progetto di una FIFO

Si vuole progettare una FIFO con profondità di 8 parole di 9 bit.

Descrizione dei segnali:

Quando il segnale *rd* viene posto a '1', l'uscita della FIFO, *data\_out*, deve essere abilitata. Quando *rd* è a '0', l'uscita deve essere posta in alta impedenza.

Quando il segnale di write *wr* viene posto a '1' si vuole scrivere in uno dei registri di 9 bit.

I segnali *rdinc* e *wrinc* sono utilizzati per incrementare i puntatori di lettura e scrittura che indicano quale registro leggere e scrivere. *Rdptrclr* e *wrptrclr* resettano i puntatori di lettura e scrittura e li indirizzano al primo registro della FIFO.

*Data\_in* è il dato che deve essere caricato in uno dei registri.

La figura (16) mostra uno schema a blocchi della FIFO.

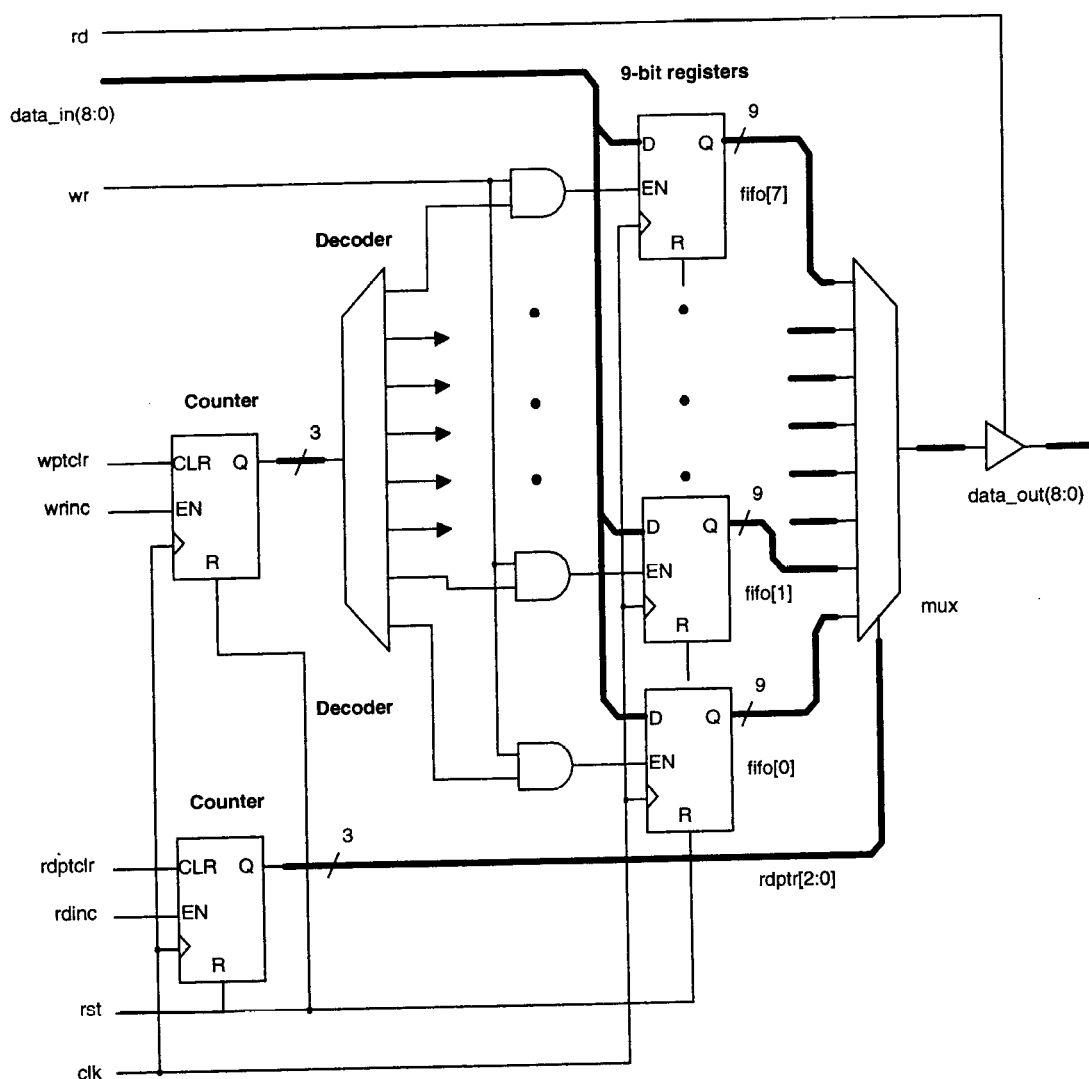


Figura (16): schema a blocchi della FIFO di 8 parole di 9 bit.

Il listato riportato di seguito illustra un modo di implementare la FIFO che utilizza la progettazione gerarchica e strutturale.

```

library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity fifoxbyy is generic (wide : integer := 32);    --width is 31 + 1
  port(
    clk, rst, oe:           in std_logic;
    rd, wr, rdinc, wrinc:   in std_logic;
    rdptrclr, wrptrclr:     in std_logic;
    data_in:                in std_logic_vector(wide downto 0);
    data_out:               out std_logic_vector(wide downto 0));
end fifoxbyy;

architecture archfifoxbyy of fifoxbyy is
  constant deep: integer := 20;--depth is 20 + 1
  type fifo_array is array(deep downto 0) of std_logic_vector(wide
downto 0);

  signal fifo: fifo_array;
  signal wrptr, rdptr: integer range 0 to deep;
  signal en: std_logic_vector(deep downto 0);
  signal dmuxout: std_logic_vector(wide downto 0);

begin

-- fifo register array:
reg_array: process (rst, clk)
begin
  if rst = '1' then
    for i in fifo'range loop
      fifo(i) <= (others => '0');
    end loop;
  elsif rising_edge(clk) then
    if wr = '1' then
      fifo(wrptr) <= data_in;
    end if;
  end if;
end process;

```

```

-- read pointer
read_count: process (rst, clk)
begin
    if rst = '1' then
        rdptr <= 0;
    elsif rising_edge(clk) then
        if rdptrclr = '1' then
            rdptr <= 0;
        elsif rdinc = '1' then
            rdptr <= rdptr + 1;
        end if;
    end if;
end process;

-- write pointer
write_count: process (rst, clk)
begin
    if rst = '1' then
        wrptr <= 0;
    elsif rising_edge(clk) then
        if wrptrclr = '1' then
            wrptr <= 0;
        elsif wrinc = '1' then
            wrptr <= wrptr + 1;
        end if;
    end if;
end process;

-- data output multiplexer
dmuxout <= fifo(wrptr);

-- three-state control of outputs
three_state: process (oe, dmuxout)
begin
    if oe = '1' then
        data_out <= dmuxout;
    else
        data_out <= (others => 'Z');
    end if;
end process;

end archfifoxbyy;

```

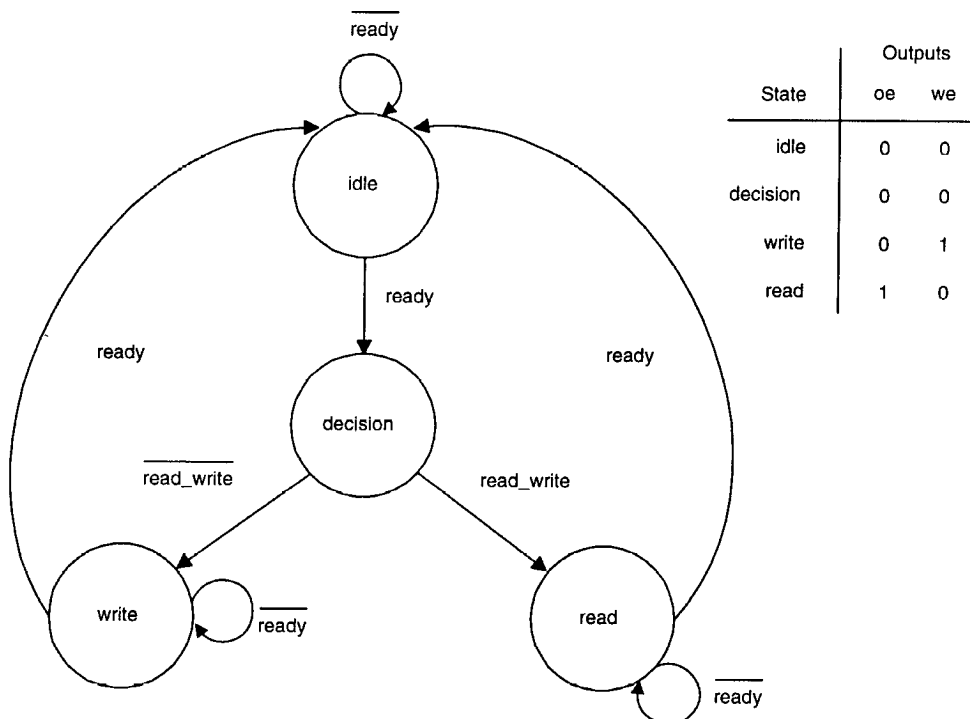
# Progetto di macchine a stati finiti

In VHDL descrivere il comportamento di una macchina a stati finiti è solo un problema di tradurre il diagramma a stati in una serie di istruzioni CASE-WHEN e IF-THEN-ELSE, come verrà mostrato nell'esempio illustrato di seguito.

Si vuole descrivere un controller che viene utilizzato per abilitare e disabilitare i segnali di write enable (*we*) e output enable (*oe*) di un buffer di memoria durante la fase di lettura e scrittura. I segnali *ready* e *read\_write* sono le uscite di un microprocessore e gli ingressi del controller.

Una nuova transizione inizia quando il segnale *ready* viene posto a '1'. Un ciclo di clock dopo l'inizio della transizione il valore del segnale *read\_write* determina se effettuare una transizione di lettura o scrittura. Se *read\_write* viene posto uguale a '1', allora si esegue un ciclo di lettura, altrimenti si esegue un ciclo di scrittura. Un ciclo viene completato quando *ready* viene posto a '1', dopo di che può iniziare una nuova transizione. Write enable viene posto a '1' durante il ciclo di scrittura e output enable viene posto a '1' durante un ciclo di lettura.

Nella figura che segue viene riportato il diagramma a bolle del controller.



# Metodologia di progetto tradizionale

Con il metodo di progetto tradizionale, il primo passo è quello di disegnare il diagramma dal quale ricavare la tabella degli stati. E' possibile quindi fare l'assegnamento degli stati e creare una tabella di transizione degli stati da cui si possono determinare le equazioni che determinano lo stato successivo e le uscite in base ai tipi di flip flop utilizzati per l'implementazione. La figura del diagramma a bolle (o diagramma degli stati) è riportata nella pagina precedente.

In questa macchina non ci sono degli stati equivalenti. E' possibile combinare la tabella di assegnamento degli stati con la tabella di transizione degli stati, come è riportato nella figura sotto.

PS		read_write ready				NS $Q_0 Q_1$		Outputs	
State	$q_0 q_1$	00	01	11	10	oe	we		
idle	00	00	01	01	00	0	0		
decision	01	11	11	10	10	0	0		
write	11	11	00	00	11	0	1		
read	10	10	00	00	10	1	0		

read_write, ready		$Q_0$			
$q_0 q_1$		00	01	11	10
00		0	0	0	0
01		1	1	1	1
11		1	0	0	1
10		1	0	0	1

$$Q_0 = \overline{q_0} q_1 + q_0 \overline{\text{ready}}$$

read_write, ready		$Q_1$			
$q_0 q_1$		00	01	11	10
00		0	1	1	0
01		1	1	0	0
11		1	0	0	1
10		0	0	0	0

$$Q_1 = \overline{q_0} \overline{q_1} \text{ready} + \overline{q_0} q_1 \overline{\text{read\_write}} + q_0 q_1 \overline{\text{ready}}$$

$$\text{oe} = q_0 \overline{q_1}$$

$$\text{we} = q_0 q_1$$



L'assegnamento degli stati è elencato nella colonna present state (PS). Si vuole utilizzare il minor numero possibile di registri per gli stati, cioè due. La colonna next state (NS) mostra la transizione dallo stato presente verso lo stato successivo in base al valore dei due ingressi *read\_write* e *ready*. Le uscite sono riportate nella colonna più a destra.

E' possibile determinare ora l'equazione dello stato successivo per ognuno dei due bit di stato. Nelle equazioni mostrate,  $Q1$  e  $Q0$  rappresentano i valori dello stato successivo, e  $q1$  e  $q0$  rappresentano i valori dello stato presente. Le mappe di Karnaugh possono essere generate facilmente dalla tabella delle transizioni: ogni riga corrisponde ad uno stato, ogni colonna corrisponde ad una combinazione degli ingressi, e gli ingressi nella mappa di Karnaugh corrispondono ai valori di  $Q1$  e  $Q0$  trovati nella tabella delle transizioni.

Le mappe di Karnaugh vengono poi utilizzate per trovar le equazioni minime assumendo che si utilizzino flip flop di tipo D. Le uscite sono funzioni solo dello stato presente (macchian di Moore).

# Le macchine a stati finiti in VHDL

Il diagramma degli stati mostrato precedentemente può essere facilmente tradotto in VHDL senza dover effettuare l'assegnamento degli stati, generare la tabella di transizione degli stati, o determinare le equazioni che individuano lo stato successivo in base al tipo di flip flop disponibile.

In VHDL ogni stato può essere tradotto in un caso mediante una istruzione CASE-WHEN. La transizione degli stati può essere specificata mediante una serie di istruzioni IF-THEN-ELSE.

Per esempio, per tradurre in VHDL un diagramma degli stati si inizia definendo un tipo enumerativo, formato dal nome degli stati, e dichiarando due segnali del tipo:

```
type StateType is (idle, decision, read, write);
signal present_state, next_state : StateType;
```

L'operazione successiva da fare è quella di creare un processo. *Next\_state* viene determinato tramite una funzione di *present\_state* e degli ingressi (*ready* e *read\_write*). Così il processo deve essere sensibile a questi segnali:

```
state_comb: process (present_state, read_write, ready)
begin
    ...
end process state_comb;
```

All'interno del processo vengono descritte le transizioni della macchina a stati finiti. Viene aperta una istruzione CASE-WHEN e specificato il primo caso (condizione WHEN), cioè lo stato di *idle*. Per questo caso, si specificano le uscite definite nello stato di *idle* e le transizioni che partono da esso.

```
state_comb: process (present_state, read_write, ready)
begin
    case present_state is
        when idle =>
            oe <= '0'; we <= '0';
            if ready = '1' then
                next_state <= decision;
            else
                next_state <= idle; --else not necessary
            end if; --included for readability
```

Ci sono due opzioni in questo caso (cioè, quando *present\_state* è *idle*): (1) la transizione allo stato *decision* se *ready* è uguale a '1' oppure (2) rimanere nello stato di *idle*. In questo caso la condizione di ELSE non è richiesta perché viene utilizzata la memoria implicita e lo stato *next\_state* rimane lo stesso.

La codifica degli altri stati viene effettuata nello stesso modo: per ogni stato viene creato un ramo nell'istruzione CASE (WHEN *state\_name* =>), si specifica quali sono le uscite dello stato, e si definisce la transizione agli altri stati con un'istruzione IF-THEN-ELSE.

```
state_comb:process(present_state, read_write, ready) begin
  case present_state is
    when idle =>      oe <= '0'; we <= '0';
      if ready = '1' then
        next_state <= decision;
      else
        next_state <= idle;
      end if;
    when decision => oe <= '0'; we <= '0';
      if (read_write = '1') then
        next_state <= read;
      else
        --read_write='0'
        next_state <= write;
      end if;
    when read =>      oe <= '1'; we <= '0';
      if (ready = '1') then
        next_state <= idle;
      else
        next_state <= read;
      end if;
    when write =>     oe <= '0'; we <= '1';
      if (ready = '1') then
        next_state <= idle;
      else
        next_state <= write;
      end if;
  end case;
end process state_comb;
```

Il processo mostrato sopra indica che l'assegnamento di *next\_state* è basato su *present\_state* e sugli ingressi correnti, ma non indica quando *next\_state* diventa *present\_state*. Questo avviene in modo sincrono, sul fronte di salita del clock, come indicato in un secondo processo:

```
state_clocked:process(clk) begin
  if (clk'event and clk='1') then
    present_state <= next_state;
  end if;
end process state_clocked;
```

Il codice completo della macchina a stati finiti è il seguente:

```

entity example is port (
    read_write, ready, clk : in bit;
    oe, we                  : out bit);
end example;

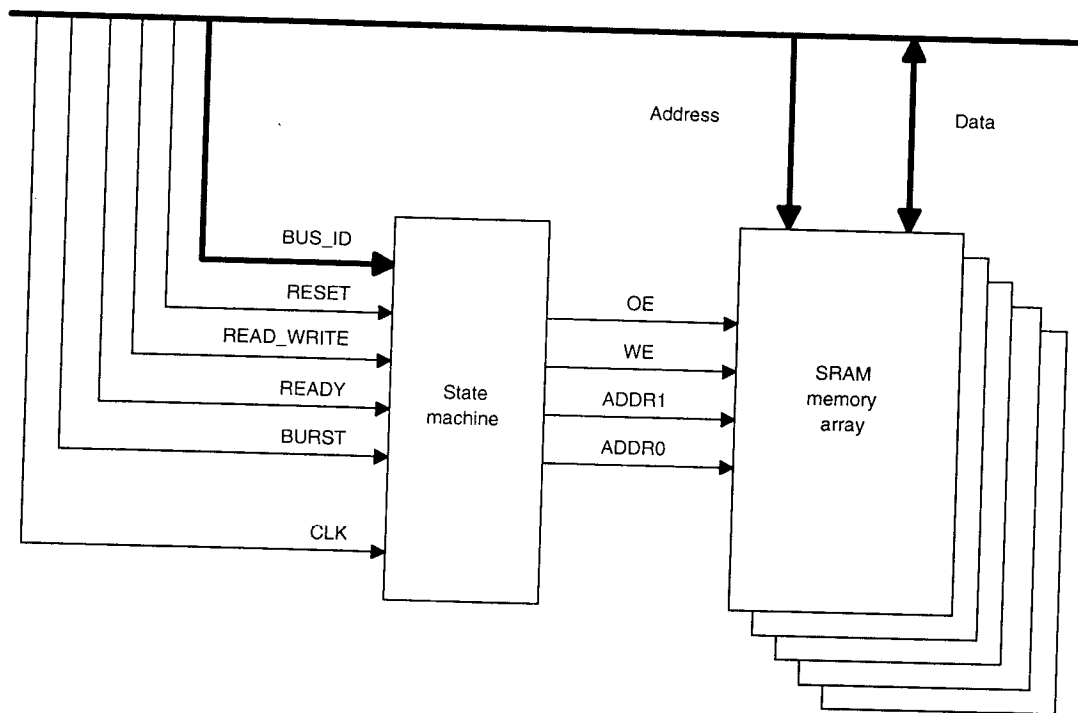
architecture state_machine of example is
    type StateType is (idle, decision, read, write);
    signal present_state, next_state : StateType;
begin
    state_comb:process(present_state, read_write, ready) begin
        case present_state is
            when idle =>      oe <= '0'; we <= '0';
                if ready = '1' then
                    next_state <= decision;
                else
                    next_state <= idle;
                end if;
            when decision => oe <= '0'; we <= '0';
                if (read_write = '1') then
                    next_state <= read;
                else          --read_write='0'
                    next_state <= write;
                end if;
            when read =>      oe <= '1'; we <= '0';
                if (ready = '1') then
                    next_state <= idle;
                else
                    next_state <= read;
                end if;
            when write =>      oe <= '0'; we <= '1';
                if (ready = '1') then
                    next_state <= idle;
                else
                    next_state <= write;
                end if;
        end case;
    end process state_comb;

    state_clocked:process(clk) begin
        if (clk'event and clk='1') then
            present_state <= next_state;
        end if;
    end process state_clocked;

end architecture state_machine;
    --"architecture" is optional; for clarity

```

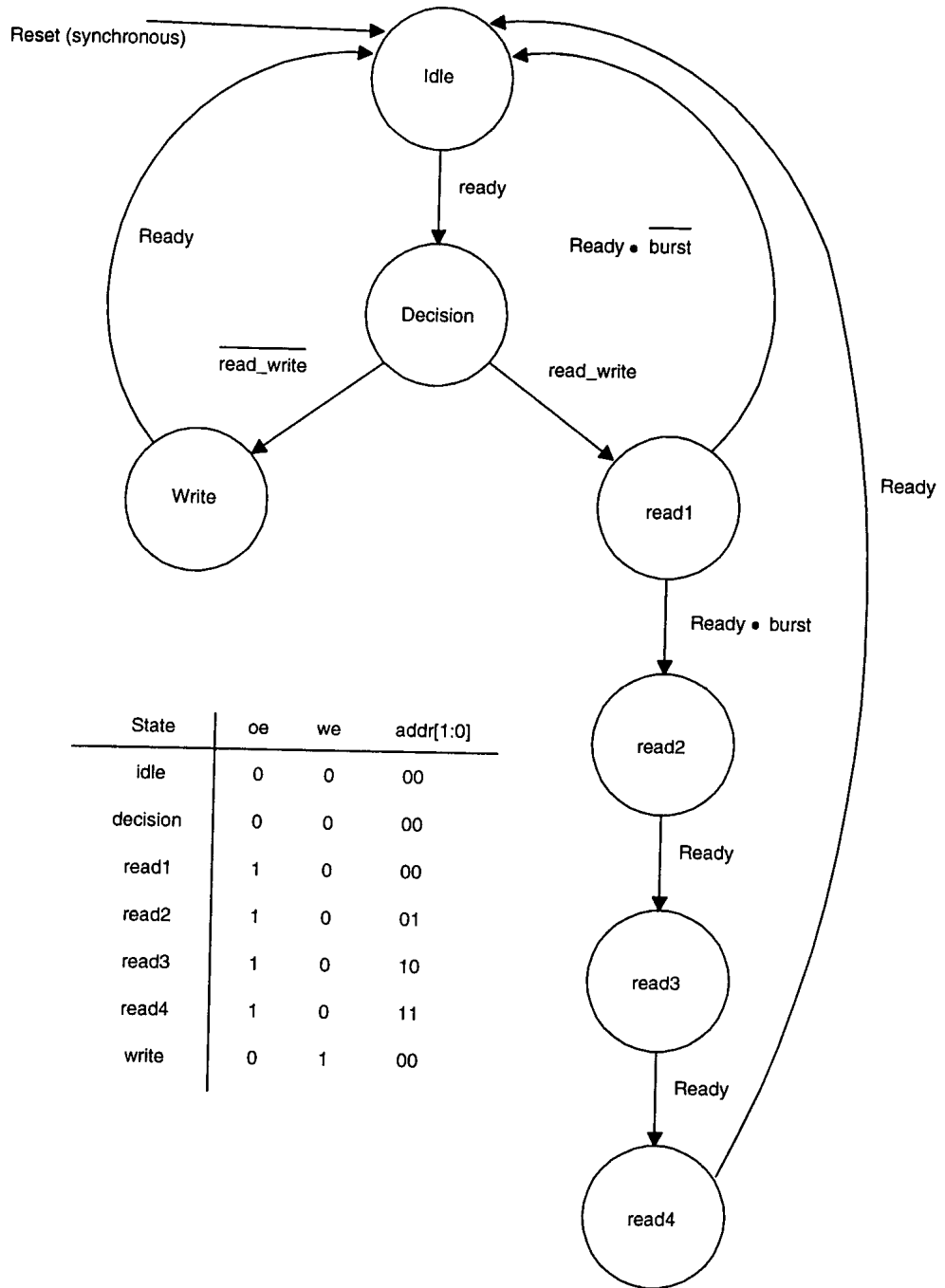
# Memory controller



Il memory controller mostrato nella figura sopra, ha il seguente funzionamento: altri dispositivi sul bus iniziano un accesso al buffer di memoria identificandolo sul bus con il suo indirizzo, F3 in esadecimale. Un ciclo di clock dopo, il segnale di *read\_write* viene messo a '1' per indicare una lettura dal buffer di memoria; oppure viene messo a '0' per indicare una scrittura sul buffer di memoria. Se l'accesso alla memoria è una lettura, la lettura potrebbe essere di una sola parola o di un burst di quattro parole. La lettura di un burst è indicata mettendo a '1' il segnale *burst* durante il primo ciclo di lettura, dopo di che il controller accede a quattro locazioni di memoria dal buffer. Le locazioni consecutive sono accedute mettendo il segnale *ready* a '1'. Il controller mette *oe* (output enable) a '1' al buffer di memoria durante una lettura, e incrementa i due bit più bassi dell'indirizzo durante una lettura a burst.

La scrittura nel buffer di memoria è sempre una scrittura di una singola parola, mai di un burst. Durante una scrittura, *we* viene messo a '1', permettendo a *data* di essere scritto nella locazione di memoria specificata da *address*. Gli accessi in lettura e scrittura sono completati quando il segnale *ready* viene messo a '1'.

La figura successiva mostra il diagramma degli stati del memory controller.



Il diagramma degli stati può essere facilmente tradotto in una serie di istruzioni CASE-WHEN come segue (per ora non viene considerato un reset sincrono):

```

case present_state is
  when idle    =>  oe <= '0'; we <= '0'; addr <= "00";
    if (bus_id = "11110011") then
      next_state <= decision;
    else
      next_state <= idle;
    end if;
  when decision=>  oe <= '0'; we <= '0'; addr <= "00";
    if (read_write = '1') then
      next_state <= read1;
    else
      --read_write='0'
      next_state <= write;
    end if;
  when read1   =>  oe <= '1'; we <= '0'; addr <= "00";
    if (ready = '0') then
      next_state <= read1;
    elsif (burst = '0') then
      next_state <= idle;
    else
      next_state <= read2;
    end if;
  when read2   =>  oe <= '1'; we <= '0'; addr <= "01";
    if (ready = '1') then
      next_state <= read3;
    else
      next_state <= read2;
    end if;
  when read3   =>  oe <= '1'; we <= '0'; addr <= "10";
    if (ready = '1') then
      next_state <= read4;
    else
      next_state <= read3;
    end if;
  when read4   =>  oe <= '1'; we <= '0'; addr <= "11";
    if (ready = '1') then
      next_state <= idle;
    else
      next_state <= read4;
    end if;
  when write   =>  oe <= '0'; we <= '1'; addr <= "00";
    if (ready = '1') then
      next_state <= idle;
    else
      next_state <= write;
    end if;
end case;

```

## Reset sincrono un una macchina a stati finiti con due processi.

Questa macchina a stati finiti richiede un reset sincrono. Al posto di specificare la transizione di reset nello stato di *idle* in ogni ramo dell'istruzione CASE, è possibile includere un'istruzione IF-THEN-ELSE all'inizio del process per fare in modo che la macchina entri nello stato di *idle* se il segnale di *reset* viene messo a '1'. Se il segnale di *reset* è a '0', allora avvengono le transizioni di stato normali della macchina, come specificato nell'istruzione CASE. Nell'istruzione IF-THEN-ELSE è necessario specificare anche come si vuole che siano le uscite (*oe*, *we* e *addr*) se il segnale di *reset* viene messo a '1'. Se non vengono specificate le uscite in questa condizione, allora si ottiene l'effetto di memoria implicita: verranno creati dei latch per assicurare che quando il *reset* viene messo a '1', le uscite mantengano il loro valore. Dato che, in questo caso, non si vuole che siano creati dei latch, viene specificato il valore *don't care* quando il segnale di reset viene messo a '1'.

Il codice richiesto per il reset è il seguente:

```
state_comb:process(reset, present_state, burst, read_write, ready)
begin
  if (reset = '1') then
    oe <= '-'; we <= '-'; addr <= "--";
    next_state <= idle;
  else
    case present_state is
      ...
    end case;
  end if;
end process state_comb;
```

Diversamente, la condizione di reset potrebbe essere messa dopo l'istruzione CASE, come ultima istruzione nel process *state\_comb*. In questo modo, potrebbe essere una semplice istruzione IF-THEN e sarebbe necessaria solo la transizione dello stato. Non è più richiesta la definizione del valore delle uscite. Il codice richiesto in questo caso è:

```
state_comb:process(reset, present_state, burst, read_write, ready)
begin
  case present_state is
    ...
  end case;
  if (reset = '1') then
    next_state <= idle;
  end if;
end process state_comb;
```



Il codice completo del memory controller è mostrato nel listato sotto:

```

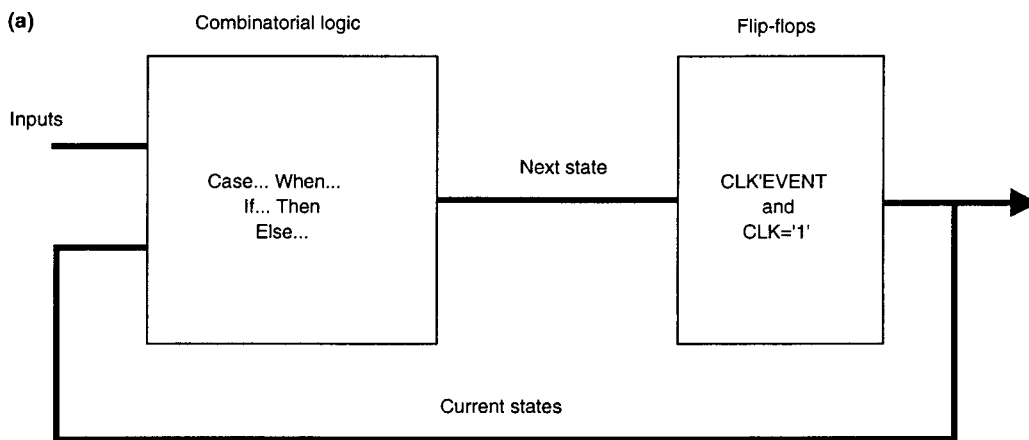
library ieee;
use ieee.std_logic_1164.all;
entity memory_controller is port (
    reset, read_write, ready,
    burst, clk                : in std_logic;
    bus_id                    : in std_logic_vector(7 downto 0);
    oe, we                    : out std_logic;
    addr                      : out std_logic_vector(1 downto 0));
end memory_controller;

architecture state_machine of memory_controller is
    type StateType is (idle, decision, read1, read2, read3, read4, write);
    signal present_state, next_state : StateType;
begin
    state_comb:process(reset, bus_id, present_state, burst, read_write, ready)
    begin
        if (reset = '1') then
            oe <= '-'; we <= '-'; addr <= "--";
            next_state <= idle;
        else
            case present_state is
                when idle => oe <= '0'; we <= '0'; addr <= "00";
                    if (bus_id = "11110011") then
                        next_state <= decision;
                    else
                        next_state <= idle;
                    end if;
                when decision=> oe <= '0'; we <= '0'; addr <= "00";
                    if (read_write = '1') then
                        next_state <= read1;
                    else
                        next_state <= write;
                    end if;
                when read1 => oe <= '1'; we <= '0'; addr <= "00";
                    if (ready = '0') then
                        next_state <= read1;
                    elsif (burst = '0') then
                        next_state <= idle;
                    else
                        next_state <= read2;
                    end if;
                when read2 => oe <= '1'; we <= '0'; addr <= "01";
                    if (ready = '1') then
                        next_state <= read3;
                    else
                        next_state <= read2;
                    end if;
                when read3 => oe <= '1'; we <= '0'; addr <= "10";
                    if (ready = '1') then
                        next_state <= read4;
                    else
                        next_state <= read3;
                    end if;
                when read4 => oe <= '1'; we <= '0'; addr <= "11";
                    if (ready = '1') then
                        next_state <= idle;
                    else
                        next_state <= read4;
                    end if;
                when write => oe <= '0'; we <= '1'; addr <= "00";
                    if (ready = '1') then
                        next_state <= idle;
                    else
                        next_state <= write;
                    end if;
            end case;
        end if;
    end process state_comb;

    state_clocked:process(clk) begin
        if rising_edge(clk) then
            present_state <= next_state;
        end if;
    end process state_clocked;
end state_machine;

```

Il listato riportato nella pagina precedente descrive una macchina a stati finiti con due processi. Un processo che descrive la logica combinatoria, e un altro che descrive la sincronizzazione delle transizioni di stato con il clock, come illustrato nella figura seguente.



### Reset asincrono.

Se è necessario utilizzare un reset asincrono invece che uno sincrono, allora è possibile utilizzare uno schema di reset come quello riportato nel seguente listato:

```

state_clocked:process(clk,reset) begin
  if reset= '1' then
    present_state <= idle;
  elsif rising_edge(clk) then
    present_state <= next_state;
  end if;
end process;

```

Se il segnale di reset è utilizzato solo durante l'inizializzazione o i malfunzionamenti del sistema, allora è preferibile utilizzare un reset di tipo asincrono. Questo principalmente perché un reset sincrono richiede risorse del dispositivo addizionali, inoltre elimina la possibilità di introdurre accidentalmente la memoria implicita.

Il codice nel listato riportato sotto è funzionalmente equivalente a quello riportato nel listato precedente, però utilizza solamente un processo per descrivere la transizione tra gli stati e la sincronizzazione delle transizioni con il clock. Una serie di istruzioni concorrenti aggiuntive sono utilizzate per descrivere il comportamento delle uscite.

```

architecture state_machine of memory_controller is
    type StateType is (idle, decision, read1, read2, read3, read4, write);
    signal state : StateType;
begin
    state_tr:process(reset, clk) begin
        -- one process fsm
        if reset = '1' then
            -- asynchronous reset
            state <= idle;
        elsif rising_edge(clk) then
            -- synchronization to clk
            -- state transitions defined
            case state is
                when idle =>
                    if (bus_id = "11110011") then
                        state <= decision;
                    else
                        -- not req'd; for clarity
                        state <= idle;
                    end if;
                when decision=>
                    if (read_write = '1') then
                        state <= read1;
                    else
                        --read_write='0'
                        state <= write;
                    end if;
                when read1 =>
                    if (ready = '0') then
                        state <= read1;
                    elsif (burst = '0') then
                        state <= idle;
                    else
                        state <= read2;
                    end if;
                when read2 =>
                    if (ready = '1') then
                        state <= read3;
                    end if;
                    -- no else state <= read2; implicit
                when read3 =>
                    if (ready = '1') then
                        state <= read4;
                    else
                        -- else not req'd; could use
                        state <= read3;
                        -- implicit memory, as in read2 above
                    end if;
                when read4 =>
                    if (ready = '1') then
                        state <= idle;
                    else
                        state <= read4;
                    end if;
                when write =>
                    if (ready = '1') then
                        state <= idle;
                    else
                        state <= write;
                    end if;
            end case;
        end if;
    end process state_tr;

    -- combinatorially decoded outputs
    with state select
        oe <= '1' when read1 | read2 | read3 | read4,
            '0' when others;

    we <= '1' when state = write else '0';

    with state select
        addr <= "01" when read2,
            "10" when read3,
            "11" when read4,
            "00" when others;
    end state_machine;

```

# Area, velocità e utilizzazione delle risorse del dispositivo.

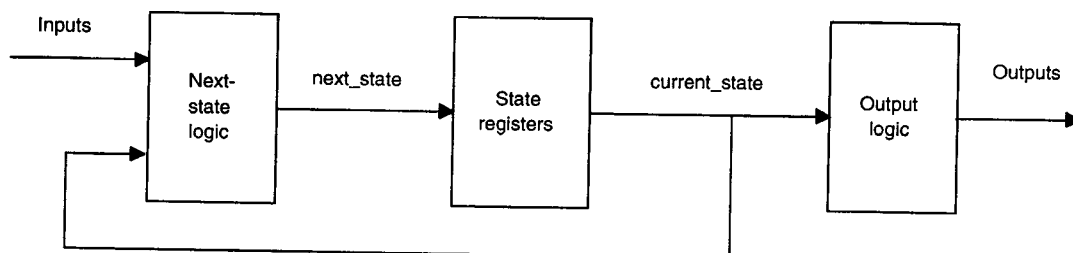
Di seguito vengono illustrate quattro tecniche per generare l'uscita delle macchine a stati finiti per le macchine di Moore:

- decodifica dell'uscita dai bit di stato in modo combinatorio
- decodifica dell'uscita in registri di uscita paralleli
- codifica dell'uscita all'interno dei bit di stato
- codifica *one hot*

Ognuna di queste tecniche produce un'implementazione logica con differenti caratteristiche di temporizzazione.

## Decodifica dell'uscita dai bit di stato in modo combinatorio

I listati precedenti descrivono le uscite che devono essere decodificate dallo stato presente dei bit di stato come riportato nella figura sotto. Questa decodifica combinatoria può richiedere livelli di logica addizionali, rendendo così più lenta la propagazione dall'uscita dei bit di stato ai segnali di uscita del dispositivo.

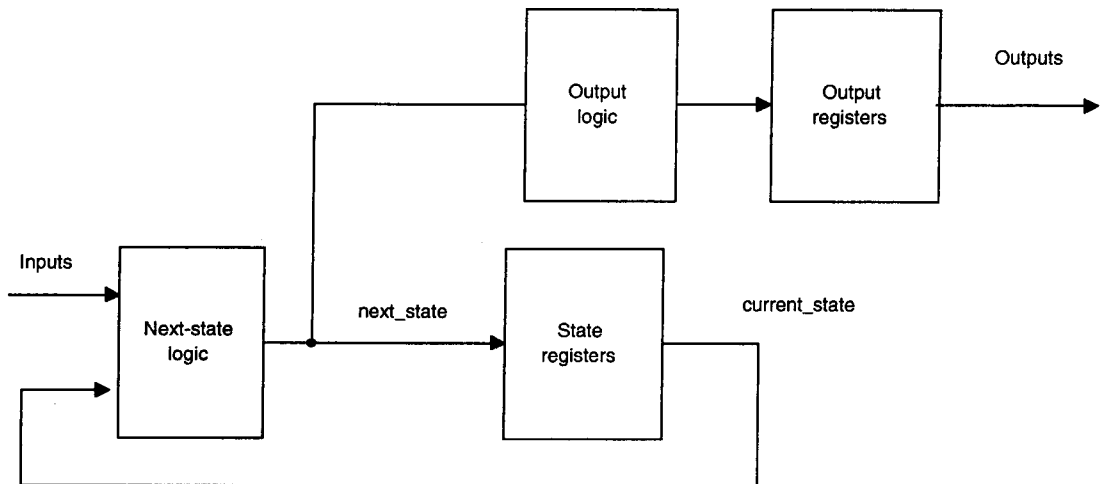


## Decodifica dell'uscita in registri di uscita paralleli

Un modo per assicurare che le uscite della macchina a stati finiti arrivino ai pin del dispositivo più rapidamente è decodificare le uscite dai bit di stato prima che i bit di stato vengano memorizzati, e quindi immagazzinare l'informazione decodificata in alcuni registri. Questo può essere fatto sia con la descrizione della macchina a stati finiti con un processo che con due processi.

In entrambi i casi, l'assegnamento a *addr* deve essere descritto fuori dal processo in cui sono definite le transizioni di stato. Invece di utilizzare le informazioni del *present\_state* per determinare il valore di *addr*, viene utilizzata l'informazione di *next\_state* per determinare come dovrebbe essere il valore di *addr* durante il successivo ciclo di clock.

Il concetto di immagazzinare il valore delle uscite in base al valore di *next\_state* è illustrato nella figura sotto.



Per modificare il progetto del memory controller per includere la decodifica delle uscite in registri di uscita paralleli, il listato precedente viene modificato come riportato di seguito. I commenti sono utilizzati per indicare dove ci sono differenze nel codice.

```

library ieee;
use ieee.std_logic_1164.all;
entity memory_controller is port (
    reset, read_write, ready,
    burst, clk
    bus_id
    oe, we
    addr
    : in std_logic;
    : in std_logic_vector(7 downto 0);
    : out std_logic;
    : out std_logic_vector(1 downto 0));
end memory_controller;

architecture state_machine of memory_controller is
    type StateType is (idle, decision, read1, read2, read3,
                      read4, write);
    signal present_state, next_state : StateType;
    signal addr_d: std_logic_vector(1 downto 0) -- D-input to
                                                -- addr f-flops
begin
    state_comb: process(bus_id, present_state, burst, read_write, ready)
    begin
        case present_state is
            when idle => oe <= '0'; we <= '0'; -- addr outputs not
            if (bus_id = "11110011") then -- defined here.
                next_state <= decision;
        end case;
    end process;
end state_machine;
  
```

```

        else
            next_state <= idle;
        end if;
    when decision=>    oe <= '0'; we <= '0';
        if (read_write = '1') then
            next_state <= read1;
        else
            next_state <= write;
        end if;
    when read1 =>    oe <= '1'; we <= '0';
        if (ready = '0') then
            next_state <= read1;
        elsif (burst = '0') then
            next_state <= idle;
        else
            next_state <= read2;
        end if;
    when read2 =>    oe <= '1'; we <= '0';
        if (ready = '1') then
            next_state <= read3;
        else
            next_state <= read2;
        end if;
    when read3 =>    oe <= '1'; we <= '0';
        if (ready = '1') then
            next_state <= read4;
        else
            next_state <= read3;
        end if;
    when read4 =>    oe <= '1'; we <= '0';
        if (ready = '1') then
            next_state <= idle;
        else
            next_state <= read4;
        end if;
    when write =>    oe <= '0'; we <= '1';
        if (ready = '1') then
            next_state <= idle;
        else
            next_state <= write;
        end if;
    end case;
end process state_comb;

with next_state select
    addr_d <= "01" when read2,
             "10" when read3,
             "11" when read4,
             "00" when others;

state_clocked:process(clk, reset) begin
    if reset = '1' then
        present_state <= idle;
        addr <= "00";
    elsif rising_edge(clk) then
        present_state <= next_state;
        addr <= addr_d;
    end if;
end process state_clocked;

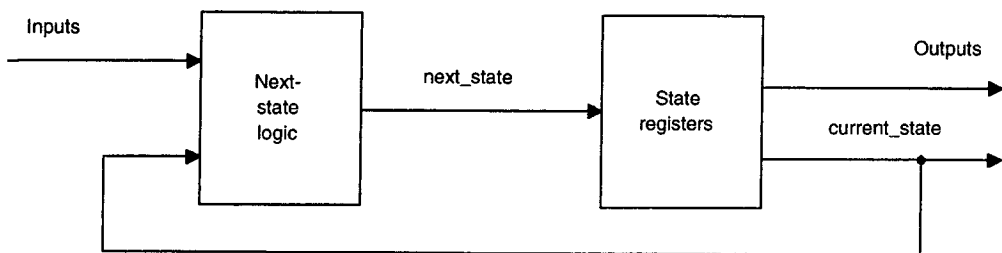
end state_machine;

```

In questo progetto è stato dichiarato un segnale aggiuntivo: *addr\_d*. Questo viene utilizzato per decodificare in modo combinatorio come dovrebbe essere il valore di *addr* il ciclo di clock successivo. Questo viene fatto rimuovendo l'istruzione di assegnamento del segnale *addr* dall'istruzione CASE, e inserendo un'istruzione di assegnamento dei segnali WITH-SELECT-WHEN, in cui il segnale di selezione è *next\_state*. Poi, il segnale *addr\_d* viene memorizzato nel segnale *addr* sul fronte di salita del clock. Tutto questo viene descritto nel processo *state\_clocked*. L'istruzione WITH-SELECT-WHEN non è necessario che sia posta tra i due processi potrebbe essere messa prima o dopo di loro.

### Codifica dell'uscita con i bit di stato

Un contatore è un esempio di una macchina a stati finiti nella quale le uscite sono i bit di stato. Questo metodo potrebbe funzionare meglio dei precedenti in alcuni casi particolari, ma richiede una scelta accurata della codifica degli stati: l'uscita deve corrispondere al valore memorizzato nel registro degli stati, come mostrato nella figura sotto.



Questo approccio rende la progettazione più complicata da comprendere e cambiare, così è raccomandata solo per i casi che richiedono ottimizzazioni di area e prestazioni non fornite dagli strumenti di sintesi, o automaticamente dalle direttive di implementazione.

## Codifica one-hot

Questa è una tecnica che utilizza  $n$  flip flop per rappresentare una macchina a stati finiti con  $n$  stati. Ogni stato ha un suo proprio flip flop, e solo un flip flop è “hot” (cioè ha memorizzato un ‘1’) ogni intervallo di tempo.

Uno dei vantaggi della codifica “one hot” è che il numero di porte richieste per decodificare l’informazione degli stati per le uscite e per la transizione allo stato successivo è molto minore del numero di porte richiesto a questo scopo dagli altri metodi di codifica. Questa differenza nella complessità diventa più evidente al crescere del numero degli stati.

A seconda dell’architettura del dispositivo utilizzato, una macchina a stati finiti con codifica *one hot* potrebbe richiedere meno risorse del dispositivo per l’implementazione del progetto. Una logica per determinare lo stato successivo più semplice richiede meno livelli di logica tra i registri di stato, permettendo così frequenze di funzionamento più alte.

La codifica *one hot* non è sempre la miglior soluzione, principalmente perché richiede più flip flop che una macchina a stati finiti codificata in modo sequenziale. In generale, la codifica *one hot* è più utile quando l’architettura del dispositivo programmabile che si vuole utilizzare contiene un numero relativamente elevato di registri e poca logica combinatoria tra ogni registro. Per esempio, la codifica *one hot* risulta più utile per le macchine a stati finiti implementate dentro alle FPGA, che generalmente hanno una densità di flip flop più elevata che le CPLD ma un minor numero di porte logiche per flip flop.

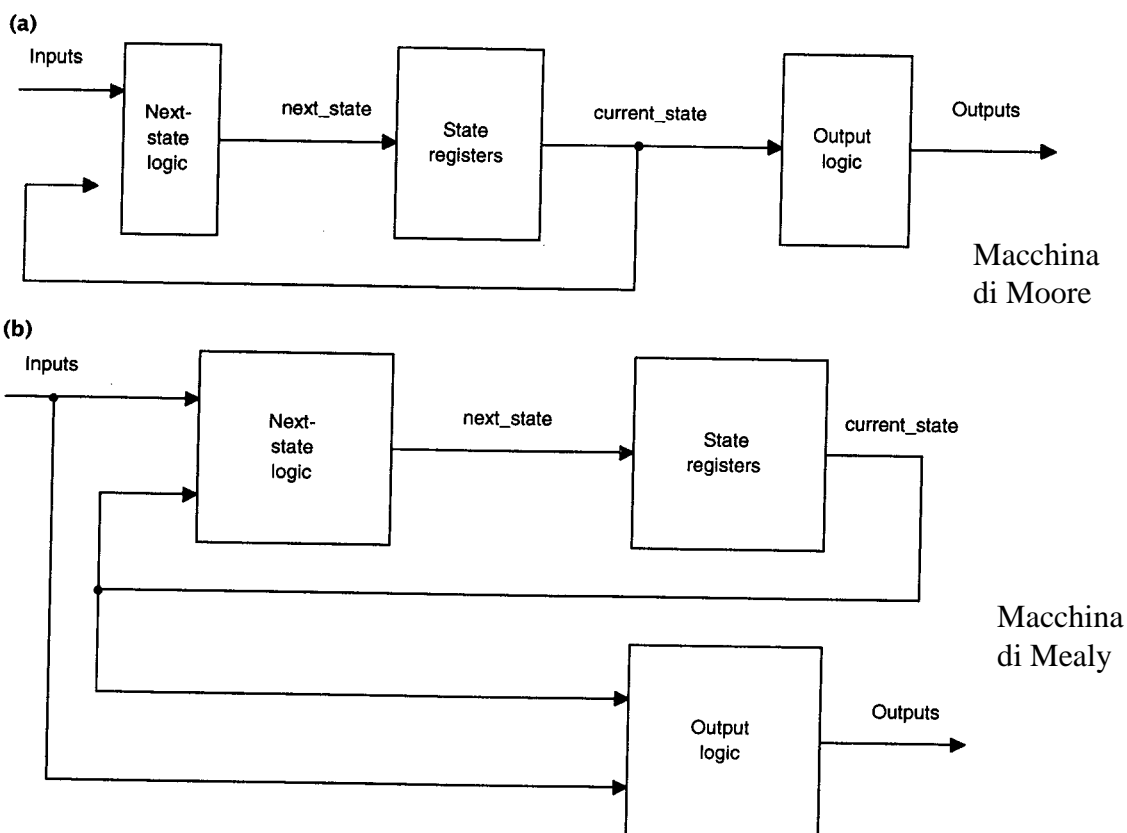
Finora si è discusso di che cosa è la codifica *one hot* e quando conviene utilizzarla, vediamo ora come descrivere una macchina a stati finiti con la codifica *one hot*. Fortunatamente, la codifica *one hot* richiede pochi o nessun cambiamento al codice sorgente, ma dipende dal software di sintesi che viene utilizzato. Molti software per la sintesi permettono di utilizzare alcune direttive di sintesi nella forma di istruzioni VHDL, o di opzioni GUI (come avviene nel software Foundation Express).

Generare le uscite per una macchina a stati finiti con codifica *one hot* è analogo a generare le uscite per macchine in cui le uscite sono decodificate dai registri di stato. La decodifica è abbastanza semplice perché gli stati sono solo dei bit singoli e non un intero vettore. La logica di uscita consiste di una porta OR perché le macchine di Moore hanno le uscite che sono funzioni solo dello stato, e tutti gli stati in una macchina a stati finiti con codifica *one hot* sono rappresentati da un bit. In una FPGA, il ritardo associato ad una porta OR è tipicamente accettabile ed è un miglioramento rispetto alle decodifiche dell’uscita che utilizzano un intero vettore di stato.



# Macchine di Mealy

Finora sono state trattate solo le macchine di Moore (figura sotto a) in cui le uscite sono funzioni solamente dello stato corrente. Le macchine di Mealy (figura sotto b) possono avere uscite che sono funzioni dello stato presente e dei segnali presenti in ingresso, come illustrato nella figura sotto.



Il lavoro aggiuntivo che è necessario per descrivere le macchine di Mealy rispetto alle macchine di Moore è minimo. Per implementare una macchina di Mealy occorre semplicemente descrivere un'uscita come funzione sia dei bit di stato che degli ingressi. Per esempio, se vi è un ingresso aggiuntivo al memory controller chiamato *write\_mask* che quando vale '1' non consente a *we* di essere abilitato, è possibile descrivere la logica per *we* come

```
if (current_state = s6) and (write_mask = '0') then
    we <= '1';
else
    we <= '0';
end if;
```

Questo rende *we* un'uscita di Mealy.

# Ulteriori Considerazioni di progetto

## Codifica degli stati utilizzando i tipi enumerativi.

I tipi enumerativi forniscono un modo facile per codificare le macchine a stati. Quando vengono sintetizzati, i segnali dello stato sono convertiti in vettori dal software di sintesi. A ogni valore dello stato è assegnata una codifica. Per la codifica *one hot*, a ogni valore dello stato corrisponde un flip flop. Per le altre codifiche, come quelle sequenziali, il numero minimo di stati richiesto è il valore intero più grande di  $\log_2 n$ , dove  $n$  è il numero degli stati. Con questa codifica, se  $\log_2 n$  non è una potenza di 2 ci saranno degli stati indefiniti.

Nel memory controller, sono stati definiti sette stati:

```
type StateType is (idle, decision, read1, read2, read3,
                  read4, write);
signal state : StateType;
```

Utilizzando una codifica sequenziale, *state* richiederà 3 flip flop. Tre flip flop tuttavia consentono di definire otto stati, così la codifica sequenziale per *state* è mostrata nella tabella riportata sotto.

**Table 5-8** Sequential encoding for an enumeration type

State	$q_0$	$q_1$	$q_2$
idle	0	0	0
decision	0	0	1
read1	0	1	0
read2	0	1	1
read3	1	0	0
read4	1	0	1
write	1	1	0
undefined	1	1	1

## Don't care impliciti.

Nell'esempio del memory controller che fa uso di un tipo enumerativo, la sintesi assume che il valore 111 sia una condizione di *don't care*. Non sono state specificate transizioni dentro e fuori da questo stato. Se la macchina a stati finisse per caso in questo stato indefinito, o illegale, non funzionerebbe più in maniera prevedibile. Il comportamento di una macchina a stati finiti, se posta in uno stato illegale dipenderà dalle equazioni che definiscono le transizioni di stato. Il vantaggio di avere la transizione fuori dallo stato illegale come condizione *don't care* è che non è richiesta alcuna logica addizionale per assicurare che la macchina avrà una transizione fuori da questo stato. Lo svantaggio di utilizzare questo *don't care* implicito è che la macchina a stati è più sensibile ai guasti.

## Tolleranza ai guasti: uscita dagli stati illegali.

Lo stato 111 è uno stato illegale per il memory controller, ma nell'hardware, glitches, spostamenti della massa, rumore, l'accensione o configurazioni illegali degli ingressi potrebbero causare il cambiamento dello stato di un flip flop, facendo in modo che la macchina entri in uno stato illegale. Se questo succede, la macchina a stati non si comporterà più in modo prevedibile, e questo potrebbe causare problemi all'intero sistema. La macchina potrebbe entrare in uno stato illegale e rimanervi indefinitamente, o, tra le altre possibilità, potrebbe generare uscite che sono illegali.

La macchina a stati può essere resa più tollerante ai guasti aggiungendo una parte di codice che genera transizioni che escono dagli stati illegali:

1. è necessario determinare quanti stati illegali sono possibili: il numero di stati illegali è il numero di stati della macchina meno 2 elevato al numero di flip flop utilizzati per la codifica della macchina. Per il memory controller c'è un solo stato indefinito.

2. Nel tipo enumerativo è necessario introdurre il nome di uno stato per ogni stato indefinito. Per esempio:

3. Infine, per la macchina a stati, deve essere specificata una transizione fuori dallo stato indefinito

```
type StateType is (idle, decision, read1, read2, read3, read4,
                  write, undefined);
```

```
case present_state is
    ...
    when undefined => next_state <= idle;
end case;
```

4. Se c'è più di uno stato indefinito, è possibile seguire lo stesso procedimento aggiungendo diversi stati indefiniti al tipo enumerativo:

```

type states is (s0, s1, s2, s3, s4, u1, u2, u3);
signal next_state, present_state: states;
...
case present_state is
    ...
    when others => next_state <= s0;
end case;

```

### Codifica esplicita degli stati: don't care e tolleranza ai guasti.

Il progetto delle macchine a stati finiti con codifica esplicita degli stati, in cui le costanti sono utilizzate per definire gli stati, deve dichiarare esplicitamente i don't care. La seguente codifica degli stati definisce sette stati (tuttavia, *state* può avere 32 valori binari e numerosi altri stati metalogici):

```

signal present_state, next_state : std_logic_vector(4 downto 0)
constant idle      : std_logic_vector(4 downto 0) := "00000";
constant decision : std_logic_vector(4 downto 0) := "00001";
constant read1    : std_logic_vector(4 downto 0) := "00100";
constant read2    : std_logic_vector(4 downto 0) := "01100";
constant read3    : std_logic_vector(4 downto 0) := "10100";
constant read4    : std_logic_vector(4 downto 0) := "11100";
constant write    : std_logic_vector(4 downto 0) := "00010";

```

Per creare una macchina a stati più insensibile ai guasti è necessario specificare una transizione di stato per gli altri 25 stati:

```

when others => next_state <= idle;

```

### Don't care espliciti.

E' necessaria della logica addizionale per specificare le transizioni dagli stati illegali ad altri stati. Tuttavia, questa soluzione potrebbe risultare troppo costosa rispetto alla necessità di avere un'elevata tolleranza ai guasti. In questo caso, piuttosto che specificare tutte le transizioni dagli stati illegali agli stati validi è possibile dichiarare che la transizione di stato è una condizione di *don't care*. E' possibile definire la condizione di *don't care* per la codifica esplicita come:

```

when others => next_state <= "-----";

```

Se le condizioni di *don't care* sono implicite nel progetto di macchine a stati che utilizzano i tipi enumerativi che non specificano tutti i possibili valori delle combinazioni, i *don't care* devono essere definiti esplicitamente per le macchine a stati progettate con una codifica degli stati esplicita. L'utilizzo delle costanti permette di definire esplicitamente sia le condizioni di *don't care* che le transizioni dagli stati illegali.

## Tolleranza ai guasti per le macchine con codifica one-hot.

La probabilità di entrare in uno stato illegale aumenta molto quando vengono utilizzate uscite codificate con i bit di stato o con la codifica *one-hot*. Entrambe queste tecniche possono dare luogo a molti più stati illegali o indefiniti. Con la codifica *one-hot*, per esempio, ci sono  $2^n$  possibili valori per il vettore di stato di  $n$  bit. La macchina a stati ha solo  $n$  stati. Qui si ha un problema: la codifica *one-hot* viene scelta in genere per raggiungere un'implementazione efficiente della macchina a stati, ma includendo la logica che controlla le transizioni da tutti gli stati illegali a uno stato di reset o ad un altro stato noto, si ottiene un'implementazione inefficiente della macchina a stati. Per specificare completamente le transizioni degli stati per una macchina con codifica degli stati *one-hot* che ha 18 stati, devono essere decodificate altre 262126 ( $2^{18}-18$ ) transizioni. Questo equivale ad introdurre una enorme quantità di logica aggiuntiva. In modo diverso, piuttosto che aggiungere le transizioni fuori da tutti i possibili stati, è possibile includere della logica che rileva quando più di un flip flop sta a '1' nello stesso istante di tempo.

Può essere generato un segnale di collisione per rilevare che più di un flip flop sta a '1' allo stesso tempo. Per otto stati, è generato come indicato nel listato sotto. La stessa tecnica può essere estesa ad un numero qualsiasi di stati.

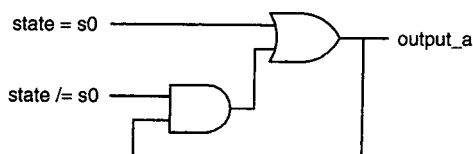
```
colin      <= (state1 and (state2 or state3 or state4 or
                state5 or state6 or state7 or state8)) or
            (state2 and (state1 or state3 or state4 or
                state5 or state6 or state7 or state8)) or
            (state3 and (state1 or state2 or state4 or
                state5 or state6 or state7 or state8)) or
            (state4 and (state1 or state2 or state3 or
                state5 or state6 or state7 or state8)) or
            (state5 and (state1 or state2 or state3 or
                state4 or state6 or state7 or state8)) or
            (state6 and (state1 or state2 or state3 or
                state4 or state5 or state7 or state8)) or
            (state7 and (state1 or state2 or state3 or
                state4 or state5 or state6 or state8)) or
            (state8 and (state1 or state2 or state3 or
                state4 or state5 or state6 or state7)) ;
```

## Istruzioni IF-THEN-ELSE non completamente specificate.

Le istruzioni IF-THEN-ELSE possono essere utilizzate per decodificare le uscite di una macchina a stati o in generale creare della logica combinatoria. Quando si utilizzano le istruzioni IF-THEN-ELSE, occorre fare attenzione ad assegnare esplicitamente il valore dei segnali per tutte le condizioni. Lasciare delle ambiguità implica utilizzare della memoria: se non viene specificato l'assegnamento di un segnale, questo implica che il segnale mantiene il suo valore. Per esempio, si consideri il seguente codice sintatticamente corretto:

```
if (present_state = s0) then
    output_a <= '1';
elsif (present_state = s1) then
    output_b <= '1';
else    -- current_state = s3
    output_c <= '1';
end if;
```

Questo è come potrebbe essere scritto scorrettamente il codice se si vuole che l'uscita *output\_a* sia posta a '1' solo nello stato *s0*, l'uscita *output\_b* sia posta a '1' solo nello stato *s1* e l'uscita *output\_c* sia posta a '1' solo nello stato *s2*. Quello che il codice descrive realmente è che all'uscita *output\_a* viene assegnato '1' nello stato *s0* e mantiene qualunque valore attualmente abbia in ogni altro stato. La figura sotto mostra l'implementazione logica di questo codice. Il circuito mostra che dopo che *present\_state* diventa *s0* una volta, il valore dell'uscita *output\_a* è sempre '1'.



Per evitare di avere sintetizzati dei latch per le uscite *output\_a*, *output\_b* e *output\_c* il codice riportato sopra potrebbe essere riscritto, inizializzando i segnali all'inizio del processo, come segue:

```
output_a <= '0';
output_b <= '0';
output_c <= '0';
if (current_state = s0) then
    output_a <= '1';
elsif (current_state = s1) then
    output_b <= '1';
else    -- current_state = s3
    output_c <= '1';
end if;
```

Questo garantisce che la logica che implementa queste uscite sia strettamente una decodifica combinatoria dei bit di stato.

# Il concetto di gerarchia

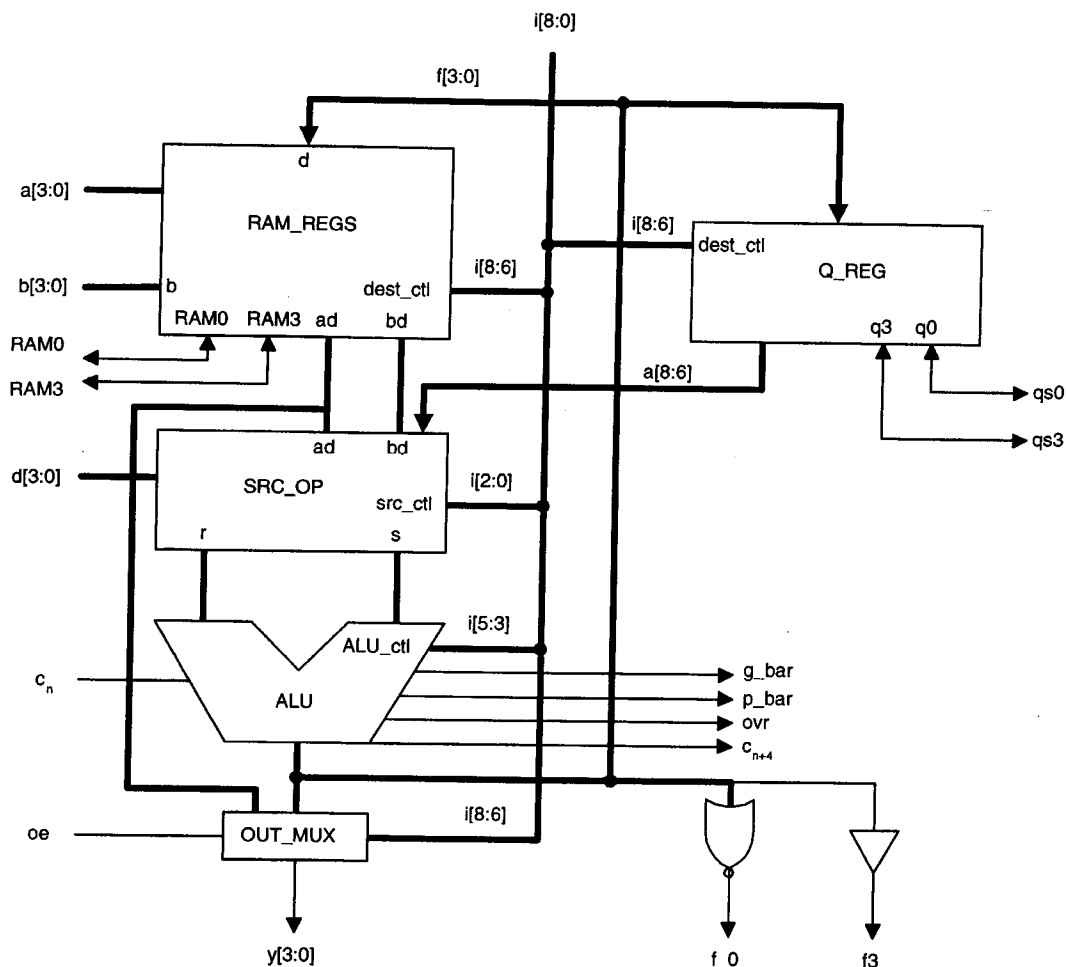
I vantaggi della progettazione di tipo gerarchico sono dovuti al fatto che permette:

- di definire i dettagli solo di una delle parti del progetto alla volta;
- di focalizzare l'attenzione solo su una piccola parte del sistema consentendo tempi di debugging più rapidi;
- di verificare ogni componente individualmente;
- di costruire il progetto a stadi, interfacciando i vari componenti uno alla volta.

## Esempio

Per illustrare la progettazione di tipo gerarchico viene utilizzato il progetto del microprocessore a 4 bit AM2901 della AMD.

L'architettura del microprocessore è stata divisa in vari blocchi come mostrato nella figura sotto.



Le unità funzionali sono:

•**RAM-REG**: questa è la RAM a due indirizzi utilizzata per leggere e scrivere i dati. Questa RAM è stata implementata utilizzando un registro fatto di flip flop. Le FPGA che contengono al loro interno della RAM sono in grado di implementare il registro utilizzando un'area inferiore. Tuttavia, utilizzare la RAM interna all'FPGA riduce la portabilità del codice VHDL su altri tipi di componenti. Viene utilizzato uno shifter per memorizzare nella RAM  $f, f/2$  o  $2*f$ ;  $ram0$  e  $ram3$  sono segnali bidirezionali utilizzati per lo shift-in e lo shift-out dei valori. Anche lo shifter è costruito nel componente RAM\_REG.

•**Q\_REG**: è un registro di una parola utilizzato principalmente nelle operazioni di moltiplicazione e divisione, o come un accumulatore. Uno shifter viene utilizzato per caricare  $f, f/2$  o  $2*f$  nel registro Q;  $qs0$  e  $qs3$  sono segnali bidirezionali utilizzati per inserire e estrarre i valori. Lo shifter è costruito all'interno del componente Q\_REG.

•**SRC\_OP**: questo componente definisce il multiplexer per gli operandi.

•**ALU**: è l'unità logico aritmetica. Questa effettua tre operazioni aritmetiche e cinque funzioni logiche su due operandi.

•**OUT\_MUX**: il multiplexer di uscita seleziona tra le uscite della ALU e i dati  $ad$  del registro.

Gli indirizzi  $a$  e  $b$  di 4 bit sono utilizzati per il registro di 16 parole RAM\_REG. Questi indirizzi determinano quali delle parole  $ad$  e  $bd$  sono lette dal registro. Quando il registro è abilitato, gli indirizzi  $b$  indicano anche quale locazione viene scritta.

I segnali  $r$  ed  $s$  sono parole di quattro bit che, insieme con il carry-in  $c_{in}$ , formano gli ingressi della ALU. Il multiplexer per gli operandi della ALU seleziona fra i segnali  $ad$ ,  $bd$ ,  $d$ ,  $q$  e 0.

Il segnale  $d$  è un ingresso diretto a  $r$  nel multiplexer per gli operandi.

Il segnale  $q$  rappresenta il contenuto di Q\_REG e viene inviato a  $s$  nel multiplexer per gli operandi.

I segnali  $ad$  ed  $f$  alimentano il multiplexer di uscita.

Il segnale  $f$  è inviato anche agli ingressi sia di Q\_REG che di RAM\_REG.

La ALU ha quattro uscite aggiuntive: carry-out  $c_{n4}$ , carry-generate (invertito)  $g_{bar}$ , carry-propagate  $p_{bar}$  (invertito) e overflow  $ovr$ .

I due segnali  $f_0$  e  $f_3$  indicano, rispettivamente, se l'uscita della ALU è zero e se il bit più significativo è 1. Queste uscite permettono di determinare se un'operazione aritmetica ha un risultato nullo o negativo senza abilitare l'uscita three-state  $y$ .

Le funzioni del microprocessore sono definite da microistruzioni di 9 bit. I primi tre bit definiscono gli operandi (vedi tabella 1), i successivi tre bit definiscono la funzione della ALU (vedi tabella 2) e gli ultimi tre bit definiscono dove devono essere inviate le uscite della ALU (vedi tabella 3).



Mnemonic	Microcode				ALU source operands	
	I <sub>2</sub>	I <sub>1</sub>	I <sub>0</sub>	Octal code	R	S
AQ	0	0	0	0	A	Q
AB	0	0	1	1	A	B
ZQ	0	1	0	2	O	Q
ZB	0	1	1	3	O	B
ZA	1	0	0	4	O	A
DA	1	0	1	5	D	A
DQ	1	1	0	6	D	Q
DZ	1	1	1	7	D	O

Tabella (1).

Mnemonic	Microcode				ALU function
	I <sub>5</sub>	I <sub>4</sub>	I <sub>3</sub>	Octal code	
ADD	0	0	0	0	R + S
SUBR	0	0	1	1	S - R
SUBS	0	1	0	2	R - S
OR	0	1	1	3	R OR S
AND	1	0	0	4	R AND S
NOTRS	1	0	1	5	(NOT R) AND S
EXOR	1	1	0	6	R XOR S
EXNOR	1	1	1	7	R XNOR S

Tabella (2).

Mnemonic	Microcode			RAM function		Q-REG function		Y Output	RAM shifter		Q shifter	
	I <sub>8</sub>	I <sub>7</sub>	I <sub>6</sub>	Shift	Load	Shift	Load		RAM <sub>0</sub>	RAM <sub>3</sub>	Q <sub>0</sub>	Q <sub>3</sub>
QREG	0	0	0	X	NONE	NONE	F->Q	F	X	X	X	X
NOP	0	0	1	X	NONE	X	NONE	F	X	X	X	X
RAMA	0	1	0	NONE	F->B	X	NONE	A	X	X	X	X
RAMF	0	1	1	NONE	F->B	X	NONE	F	X	X	X	X
RAMQD	1	0	0	DOWN	F/2->B	DOWN	Q/2->Q	F	F <sub>0</sub>	IN <sub>3</sub>	Q <sub>0</sub>	IN <sub>3</sub>
RAMD	1	0	1	DOWN	F/2->B	X	NONE	F	F <sub>0</sub>	IN <sub>3</sub>	Q <sub>0</sub>	X
RAMQU	1	1	0	UP	2F->B	UP	2Q->Q	F	IN <sub>0</sub>	F <sub>3</sub>	IN <sub>0</sub>	Q <sub>3</sub>
RAMU	1	1	1	UP	2F->B	X	NONE	F	IN <sub>0</sub>	F <sub>3</sub>	X	Q <sub>3</sub>

X = Don't care.

B = Register addressed by B inputs.

UP is toward MSB, DOWN is toward LSB.

Tabella (3).

# Librerie e Packages

Le librerie e i packages sono utilizzati per dichiarare e immagazzinare i componenti, le funzioni, le procedure in modo tale che questi possano essere riutilizzati in altri progetti.

## Librerie

Una libreria è un posto in cui possono essere compilate le singole unità del progetto. Il formato di compilazione può essere un formato specifico del programma che si utilizza. Le unità presenti all'interno di una libreria possono essere riutilizzate all'interno di altre unità a patto che la libreria sia resa visibile. Più unità possono essere definite all'interno di un solo file, anche se tipicamente se ne trova solo una per file.

Finora è stato incontrato un tipo di libreria: la libreria IEEE. Nella libreria IEEE sono immagazzinate tutte le unità di progetto standard IEEE , come il packages STD\_LOGIC\_1164 e altri.

Per utilizzare l'unità di progetto definita in un package, è necessario rendere la libreria visibile utilizzando l'istruzione LIBRARY come segue:

```
library ieee;
```

Questa istruzione rende accessibile la libreria. Tuttavia non permette di utilizzare immediatamente le unità di progetto presenti nella libreria. È necessario renderli prima visibili utilizzando l'istruzione USE.

## Packages

Un package è un'unità di progetto che viene utilizzata per rendere visibili ad altre unità di progetto o a se stessa componenti, funzioni ecc...

Un package consiste di una dichiarazione di package, e opionalmente un corpo del package. Una dichiarazione di package viene utilizzata per dichiarare componenti, funzioni, procedure ecc...

Un package viene reso visibile utilizzando l'istruzione USE nella forma:

```
use library_name.package_name.item;
```

Se si vuole che tutti i componenti del package siano visibili, è possibile utilizzare la parola riservata ALL, come si è visto negli esempi precedenti. A meno che non ci sia qualche componente specifico, in un package, che entra in conflitto con la definizione di un componente in un altro package, è preferibile utilizzare la parola riservata ALL.

## Componenti

I componenti sono entity del progetto che sono utilizzate da altre entity del progetto. E' quindi necessario che la dichiarazione del componente sia resa visibile, dal progetto in cui deve essere utilizzata. Una dichiarazione di componente definisce una interfaccia per istanziare il componente. La dichiarazione di componente permette una progettazione di tipo top-down.

Per esempio, ogni componente nel progetto dell'AM2901 potrebbe essere dichiarato in un package, e una unità di progetto separata potrebbe utilizzare queste dichiarazioni per definire una netlist che connette i componenti insieme.

Un esempio di dichiarazione di componente per una entity chiamata *dflop* è riportata nel listato che segue.

```

library ieee;
use ieee.std_logic_1164.all;
package my_package is
    component dflop port (
        d, clk: in std_logic;
        q:      out std_logic);
    end component;
end my_package;

library ieee;
use ieee.std_logic_1164.all;
entity dflop is port(
    d, clk: in std_logic;
    q:      out std_logic);
end dflop;
architecture archdflop of dflop is
begin
    process
        wait until clk = '1';
        q <= d;
    end;
end;
```

Per utilizzare questo componente in un'altra entity, il progetto dell'entity e la sua dichiarazione di componente devono essere resi visibili. Il listato riportato sotto è un progetto che istanzia semplicemente due *dflop* in serie, assumendo che *my\_package* e *dflop* sono stati compilati nella libreria *work*.

```

library ieee,work;
use ieee.std_logic_1164.all;
use work.my_package.all;
entity my_design is port( async, clock: in std_logic;
                           filt: out std_logic);

end my_design;
architecture arch_my_design of my_design is
    signal tmp: std_logic;
begin
u1: dflop port map (clk => clock, d => async, q =>
tmp);
u2: dflop port map(tmp, clock, filt);
end arch_my_design;

```

Nella prima istanziazione, il simbolo => viene utilizzato in associazione con il nome per associare i segnali correnti (i segnali *async*, *clock*, e *tmp* che portano i dati dentro la entity) con i segnali locali (le porte *d*, *clk* e *q* che definiscono l'interfaccia del componente).

Nella seconda istanziazione, per mappare *tmp* con *d*, *clock* con *clk* e *filt* con *q* è stata utilizzata l'associazione in base alla posizione. In questo caso ad ogni segnale corrente è associato un segnale locale mediante la sua posizione all'interno della **port map**.

Le uscite non connesse, cioè le uscite del componente che non sono necessarie, possono essere associate con la parola chiave OPEN. Per esempio, la porta *c\_out* che rappresenta il carry-out di un sommatore *add* potrebbe essere non necessaria e quindi lasciata aperta istanziando il componente come segue:

```
U3: port map(a => r, b => s, sum => f, c_out => open);
```

Per concludere l'esempio iniziale del microprocessore a 4 bit AM2901 di seguito è riportato il listato VHDL che definisce l'architettura del microprocessore:

```
library ieee;
use ieee.std_logic_1164.all;
use work.numeric_std.all;
use work.am2901_comps.all;
entity am2901 is port(
    clk, rst:    in std_logic;
    a, b:        in unsigned(3 downto 0);    -- address inputs
    d:           in unsigned(3 downto 0);    -- direct data
    i:           in std_logic_vector(8 downto 0);    -- micro instruction
    c_n:         in std_logic;                -- carry in
    oe:          in std_logic;                -- output enable
    ram0, ram3:  inout std_logic;            -- shift lines to ram
    qs0, qs3:    inout std_logic;            -- shift lines to q
    y:           buffer unsigned(3 downto 0); -- data outputs (3-state)
    g_bar, p_bar: buffer std_logic;          -- carry generate, propagate
    ovr:         buffer std_logic;           -- overflow
    c_n4:        buffer std_logic;           -- carry out
    f_0:         buffer std_logic;           -- f = 0
    f3:          buffer std_logic;           -- f(3) w/o 3-state
end am2901;
architecture am2901 of am2901 is
    alias dest_ctl: std_logic_vector(2 downto 0) is i(8 downto 6);
    alias alu_ctl:  std_logic_vector(2 downto 0) is i(5 downto 3);
    alias src_ctl:  std_logic_vector(2 downto 0) is i(2 downto 0);
    signal ad, bd: unsigned(3 downto 0);
    signal q: unsigned(3 downto 0);
    signal r, s: unsigned(3 downto 0);
    signal f: unsigned(3 downto 0);
begin
    -- instantiate and connect components
    u1: ram_regs port map(clk => clk, rst => rst, a => a, b => b, f => f,
        dest_ctl => dest_ctl, ram0 => ram0, ram3 => ram3,
        ad => ad, bd => bd);
    u2: q_reg port map(clk => clk, rst => rst, f => f, dest_ctl => dest_ctl,
        qs0 => qs0, qs3 => qs3, q => q);
    u3: src_op port map(d => d, ad => ad, bd => bd, q => q,
        src_ctl => src_ctl, r => r, s => s);
    u4: alu port map(r => r, s => s, c_n => c_n, alu_ctl => alu_ctl,
        f => f, g_bar => g_bar, p_bar => p_bar,
        c_n4 => c_n4, ovr => ovr);
    u5: out_mux port map(ad => ad, f => f, dest_ctl => dest_ctl,
        oe => oe, y => y);
    -- define f_0 and f3 outputs
    f_0 <= '0' when f = "0000" else 'Z';
    f3 <= f(3);
end am2901;
```

Come si può vedere dal listato VHDL nel progetto, oltre alle librerie standard, è stata richiamata la libreria AM2901\_COMPS con tutti i suoi componenti. Questa libreria contiene tutti i macroblocchi del microprocessore illustrati precedentemente e tutti i componenti necessari per creare i macroblocchi.

## Generic e componenti parametrizzati

Il progetto di un componente parametrizzato è simile a quello di ogni altro componente, eccetto che potrebbe essere utilizzato un parametro, o *generic*, per definire le dimensioni del componente.

Nel listato riportato sotto viene definito un registro con reset sincrono di  $n$  bit (dove  $n$  è definito da dal parametro generic *size*)

```
--      Register Set
--      sizes: (size) a generic
--
--      clk      -- posedge clock input
--      rst      -- asynchronous reset
--      pst      -- asynchronous preset
--      load     -- active high input loads register
--      d        -- register input
--      q        -- register output
-----
library ieee;
use ieee.std_logic_1164.all;
entity reg is generic ( size: integer := 2);
    port( clk, load:    in std_logic;
          rst, pst:    in std_logic;
          d:           in std_logic_vector(size-1 downto 0);
          q:           buffer std_logic_vector(size-1 downto 0));
end reg;
architecture archreg of reg is
begin
    p1: process (clk) begin
        if rst = '1' then
            q <= (others => '0');
        elsif pst = '1' then
            q <= (others => '1');
        elsif (clk'event and clk='1') then
            if load = '1' then
                q <= d;
            else
                q <= q;
            end if;
        end if;
    end process;
end archreg;
```

Quando viene istanziato il componente, *size* è utilizzato per indicare la larghezza del registro. L'istanziamento per questo componente è la seguente:

```
U1: reg generic map(4)

    port map(myclk,vdd,reset,vss,data,mysig);
```

Se le parole chiave GENERIC MAP vengono omesse il parametro *size* assume il valore di default.

# Funzioni e procedure

Utilizzate frequentemente per la simulazione, le funzioni e le procedure sono costrutti di alto livello che calcolano valori o definiscono processi per la conversione di tipo tra gli operatori. Le funzioni di maggiore utilizzo sono già predefinite negli standard IEEE 1076, 1164 e 1076.3.

## Funzioni

Si consideri ad esempio il seguente listato:

```

1  function bl2bit(a:BOOLEAN) return BIT is
2      begin
3          if a then
4              return '1';
5          else
6              return '0';
7          end if;
8      end bl2bit;
```

Viene descritta una funzione che viene utilizzata per convertire un segnale dal tipo BOOLEAN al tipo BIT. Entrambi i tipi BOOLEAN e BIT sono tipi predefiniti nello standard IEEE 1076. La linea 1 dichiara la funzione *bl2bit* e definisce i parametri di ingresso come tipi BOOLEAN e ritorna un valore che è di tipo BIT. Le linee da 2 a 8 iniziano e concludono la definizione della funzione. Le linee da 3 a 7 sono istruzioni di tipo sequenziale che definiscono il valore ritornato in base alla condizione booleana *a*.

## Parametri delle funzioni

I parametri delle funzioni possono essere solo ingressi, il parametro *a* riportato nel listato sopra e solo un ingresso. Per default tutti i parametri sono di modo IN, così non è necessario dichiararli esplicitamente. Le funzioni possono ritornare un solo argomento (le procedure, come verrà mostrato in seguito, possono ritornare più di un argomento).

Tutte le istruzioni all'interno di una funzione sono sequenziali. In aggiunta, non possono essere dichiarati nuovi segnali; tuttavia, in una funzione possono essere dichiarate delle variabili, nella parte dichiarativa della funzione, e possono essergli assegnati dei valori all'interno della funzione, come viene mostrato nel listato che segue.

```

1  -- bv2i
2  -- Bit_vector to Integer.
3  -- In:   bit_vector.
4  -- Return: integer.
5  --
6  function bv2I (bv : bit_vector) return integer is
7      variable result, abit : integer := 0;
8      variable count       : integer := 0;
9      begin -- bv2i
10         bits : for I in bv'low to bv'high loop
11             abit := 0;
12             if ((bv(I) = '1')) then
13                 abit := 2**(I - bv'low);
14             end if;
15             result := result + abit;      -- Add in bit if '1'.
16             count := count + 1;
17             exit bits when count = 32;    -- 32 bits max.
18         end loop bits;
19         return (result);
20     end bv2I;
```



## Utilizzo delle funzioni

Una funzione può essere definita nella parte dichiarativa di una architettura, nel qual caso la definizione della funzione serve anche come dichiarazione della funzione. Diversamente, potrebbe essere utilizzato un **package** per dichiarare la funzione con la definizione della funzione che avviene nel **package** body. E' possibile creare una collezione di funzioni di conversione di tipo metterle in un package, compilare il package in una libreria in modo tale da poter essere facilmente utilizzato in un qualunque progetto.

Una funzione dichiarata nella parte dichiarativa di una architettura è visibile solo da quella architettura. Una funzione dichiarata in un **package** può essere resa visibile, utilizzando l'istruzione USE, da altri progetti. Inoltre, se una funzione richiede l'utilizzo di altre funzioni, è più comodo avere la dichiarazione e la definizione delle funzioni che servono in un package piuttosto che in una architettura.

La definizione di funzione è localizzata nella parte dichiarativa di una architettura, e serve anche come dichiarazione della funzione, come riportato nel listato che segue.

```
entity full_add is port(
    a, b, carry_in: in bit;
    sum, carry_out: out bit);
end full_add;
architecture full_add of full_add is
    function majority (a, b, c: bit) return bit is
    begin
        return ((a and b) or (a and c) or (b and c));
    end majority;
begin
    sum <= a xor b xor carry_in;
    carry_out <= majority(a, b, carry_in);
end;
```

In altro modo, la funzione può essere dichiarata nella dichiarazione di package e definita nel corpo del package. Anche altre dichiarazioni possono essere parte del package, come mostrato nel listato che segue.

```

package my_package is
  function majority (a, b, c: bit) return bit;
  function inc_bv (a: bit_vector) return bit_vector;
  function i2bv (val, width : integer) return bit_vector;
end my_package;

package body my_package is
  function majority (a, b, c: bit) return bit is
  begin
    return ((a and b) or (a and c) or (b and c));
  end majority;

  -- inc_bv
  -- Increment bit_vector.
  -- In:    bit_vector.
  -- Return: bit_vector.
  --
  function inc_bv (a: bit_vector) return bit_vector is
    variable s      : bit_vector (a'range);
    variable carry  : bit;
  begin
    carry := '1';
    for i in a'low to a'high loop
      s(i) := a(i) xor carry;
      carry := a(i) and carry;
    end loop;

    return (s);
  end inc_bv;

  -- i2bv
  -- Integer to Bit_vector.
  -- In:    integer, value and width.
  -- Return: bit_vector, with right bit the most significant.
  --
  function i2bv (val, width : integer) return bit_vector is
    variable result : bit_vector (0 to width-1) := (others=>'0');
    variable bits   : integer := width;
  begin
    if (bits > 32) then
      bits := 32;
    else
      ASSERT 2**bits > VAL REPORT
        "Value too big for BIT_VECTOR width"
        SEVERITY WARNING;
    end if;

    for i in 0 to bits - 1 loop
      if ((val/(2**i)) MOD 2 = 1) then
        result(i) := '1';
      end if;
    end loop;

    return (result);
  end i2bv;

end my_package;

```

La dichiarazione del package dichiara solo le funzioni. Questa dichiarazione delle funzioni fornisce un'interfaccia per il progetto che chiama queste funzioni; la definizione delle funzioni è contenuta nel corpo del package.

Il listato che segue rende visibili la dichiarazione e la definizione delle funzioni. Si assume che il package è compilato nella libreria *work*.

```

entity full_add is port(
    a, b, carry_in: in bit;
    sum, carry_out: out bit);
end full_add;

use work.my_package.majority -- could specify .all, but not needed
architecture full_add of full_add is
begin
    sum <= a xor b xor carry_in;
    carry_out <= majority(a, b, carry_in);
end;

```

La posizione dell'istruzione USE prima dell'architettura rende visibile il package all'architettura, ma non alla dichiarazione di entity.

## Procedure

Come le funzioni, le procedure sono costrutti per la progettazione ad alto livello per calcolare o definire processi che possono essere utilizzati per la conversione di tipo o in alternativa all'istanziamento di componenti.

Le procedure differiscono dalle funzioni in pochi particolari. Per iniziare, una procedura può ritornare più di un valore. Questo viene fatto con i parametri: se un parametro è dichiarato di modo OUT o INOUT, allora il parametro viene ritornato al parametro attuale della procedura chiamante. Un parametro in una funzione, tuttavia, può essere solo di modo IN. Un'altra differenza tra una procedura e una funzione è che una procedura può contenere una istruzione di WAIT, mentre la funzione no.

Come per le funzioni, tutte le istruzioni all'interno di una procedura devono essere istruzioni sequenziali, e anche nelle procedure non si possono dichiarare segnali. Ma le variabili possono essere dichiarate nella parte dichiarativa della procedura.

Le procedure e le funzioni sono dichiarate e definite nello stesso modo: entrambe nella parte dichiarativa di una architettura o in un package con associata la loro definizione nel corpo del package.

Di seguito è riportato un esempio di dichiarazione di procedura.

```

library ieee;
use ieee.std_logic_1164.all;
package myflops is
    procedure dff ( signal d: bit_vector;
                    signal clk: bit;
                    signal q: out bit_vector);
    procedure dff ( signal d: bit_vector;
                    signal clk, rst: bit;
                    signal q: out bit_vector);
    procedure dff ( signal d: bit_vector;
                    signal clk, rst, pst: bit;
                    signal q: out bit_vector);
    procedure dff ( signal d: bit_vector;
                    signal clk, rst: bit;
                    signal q, q_bar: out bit_vector);
    procedure dff ( signal d: std_logic_vector;
                    signal clk: std_logic;
                    signal q: out std_logic_vector);
    procedure dff ( signal d: std_logic_vector;
                    signal clk, rst: std_logic;
                    signal q: out std_logic_vector);
    procedure dff ( signal d: std_logic_vector;
                    signal clk, rst, pst: std_logic;
                    signal q: out std_logic_vector);
    procedure dff ( signal d: std_logic_vector;
                    signal clk, rst: std_logic;
                    signal q, q_bar: out std_logic_vector);
end myflops;

package body myflops is
    procedure dff ( signal d: bit_vector;
                    signal clk: bit;
                    signal q: out bit_vector) is
    begin
        if clk'event and clk = '1' then
            q <= d;
        end if;
    end procedure;

    procedure dff ( signal d: bit_vector;
                    signal clk, rst: bit;
                    signal q: out bit_vector) is
    begin
        if rst = '1' then q <= (others => '0');
        elsif clk'event and clk = '1' then
            q <= d;
        end if;
    end procedure;

    procedure dff ( signal d: bit_vector;
                    signal clk, rst, pst: bit;
                    signal q: out bit_vector) is
    begin
        if rst = '1' then q <= (others => '0');
        elsif pst = '1' then q <= (others => '1');
        elsif clk'event and clk = '1' then
            q <= d;
        end if;
    end procedure;
end myflops;

```

```

procedure dff ( signal d: bit_vector;
                signal clk, rst: bit;
                signal q, q_bar: out bit_vector) is
begin
    if rst = '1' then q <= (others => '0');
    elsif clk'event and clk = '1' then
        q <= d; q_bar <= not d;
    end if;
end procedure;

procedure dff ( signal d: std_logic_vector;
                signal clk: std_logic;
                signal q: out std_logic_vector) is
begin
    if clk'event and clk = '1' then
        q <= d;
    end if;
end procedure;

procedure dff ( signal d: std_logic_vector;
                signal clk, rst: std_logic;
                signal q: out std_logic_vector) is
begin
    if rst = '1' then q <= (others => '0');
    elsif clk'event and clk = '1' then
        q <= d;
    end if;
end procedure;

procedure dff ( signal d: std_logic_vector;
                signal clk, rst, pst: std_logic;
                signal q: out std_logic_vector) is
begin
    if rst = '1' then q <= (others => '0');
    elsif pst = '1' then q <= (others => '1');
    elsif clk'event and clk = '1' then
        q <= d;
    end if;
end procedure;

procedure dff ( signal d: std_logic_vector;
                signal clk, rst: std_logic;
                signal q, q_bar: out std_logic_vector) is
begin
    if rst = '1' then q <= (others => '0');
    elsif clk'event and clk = '1' then
        q <= d; q_bar <= not d;
    end if;
end procedure;

end myflops;

```

Nel listato riportato in questa pagina e nella pagina precedente sono dichiarate otto procedure chiamate *dff* per vari tipi e numeri di parametri. Quando il numero di parametri della procedura è lo stesso, esse sono differenziate per il tipo di ogni parametro.