Università degli Studi di Milano

Laurea in Informatica
Laurea in Sicurezza dei sistemi e delle reti informatiche

Elementi di VHDL

Stefano Ferrari

Architetture e reti logiche

Indice

1. Introduzione

Il VHDL è un linguaggio formale per la descrizione di hardware.

Il suo nome è l'acronimo di VHSIC Hardware Description Language, dove VHSIC è a sua volta l'acronimo di Very High Speed Integrated Circuit.

Il VHDL è nato negli anni '80 come progetto del U.S. Department of Defense (DoD) per risolvere il problema dell'obsolescenza dell'hardware. Il problema della riprogettazione di dispositivi, magari con componenti non ben documentati, si ripresentava ad ogni cambiamento tecnologico e poteva dare problemi di incompatibilità con i componenti esistenti. Venne richiesto quindi un linguaggio con una ampia capacità descrittiva (dalla porta logica al sistema completo), che fornisse gli stessi effetti su ogni simulatore e che fosse indipendente dalla tecnologia e dalla metodologia di progettazione.

Le tappe dello sviluppo del VHDL sono:

1985 version 7.2, prima versione pubblica del linguaggio;

1986 tutti i diritti sul linguaggio vengono dati all'IEEE;

1987 IEEE Std 1076-1987, primo standard completo del linguaggio;

1987 US DoD richiede a tutti fornitori la specifica in VHDL della componentistica acquistata;

1993 IEEE Std 1076-1993, versione aggiornata del linguaggio;

1993 IEEE Std 1164, definisce un package standard per il tipo di dato MVL9 (Multi Valued Logic, 9 valori);

1999 IEEE Standard 1076.1-1999, definisce l'estensione VHDL per dispositivi analogici e a segnali misti.

1.1 Modellazione in VHDL

La specifica concettuale di un componente ne descrive:

- l'interfaccia, in che modo il componente può relazionarsi con l'esterno;
- le funzionalità, gli effetti esterni della manipolazione dei segnali che avviene all'interno.

Lo sviluppo di un modello VHDL a partire dalla specifica concettuale avviene attraverso un processo di raffinamento basato sulla ripetizione delle fasi:

- compilazione;
- analisi;
- simulazione.

La compilazione serve per verificare che la descrizione del modello rispetti le regole sintattiche del VHDL e per effettuare alcune operazioni progettuali che possono essere facilmente automatizzate.

La simulazione serve per mettere alla prova il modello virtuale che è oggetto della progettazione.

Ad ogni fase di compilazione o simulazione segue una fase di analisi nella quale si studiano i risultati dell'operazione precedente per poi decidere se proseguire con una ulteriore fase di compilazione o simulazione.

Un modello VHDL può essere creato a diversi livelli di astrazione, secondo un processo di raffinamento del modello iniziale.
La descrizione può essere di tipo:

- comportamentale o *behavioral*:
  - descrizione algoritmica;
- *dataflow*:
  - descrizione a livello di flusso dati tra registri;
- strutturale o *structural*:
  - descrizione di strutture composte dall'interconnessione di componenti di livello gerarchico inferiore.
- mista.

Il VHDL supporta:

- la descrizione a diversi livelli di astrazione della funzionalità del modello.
- la descrizione di componenti concorrenti;
- la descrizione gerarchia del sistema;
- la modellazione dell'andamento temporale dei segnali attraverso la descrizione di forme d'onda.

Per poter utilizzare il VHDL in tutta la sua potenzialità, è necessario disporre di opportuni strumenti di progettazione automatizzata (detti anche CAD, *Computer Aided Design*).
Un ambiente CAD per il VHDL generalmente comprende:

- un insieme di librerie:
  - fornisce le descrizioni di componenti già pronte per essere utilizzate;
- un compilatore:
  - controlla la correttezza della sintassi della descrizione;
  - converte il codice VHDL per le fasi di progetto seguenti (sintesi);
- un simulatore:
  - permette di sottoporre il componente progettato a prove di validazione;
  - tipicamente, permette di tracciare i segnali (virtuali) generati durante la simulazione.

2. Concetti di base

Il VHDL viene usato per descrivere sistemi hardware digitali. Essendo un linguaggio formale, condivide alcune caratteristiche con i linguaggi di programmazione, i quali vengono usati per descrivere sistemi software. Questa somiglianza può trarre in errore.

Un sistema hardware svolge, in generale, le seguenti operazioni:

- riceve dei segnali;
- opera delle trasformazioni sui segnali di ingresso;
- produce dei segnali in uscita.

Inoltre, esso può generalmente essere rappresentato in termini di sottosistemi, i quali posseggono analoghe caratteristiche ed operano in modo parallelo e potenzialmente indipendente gli uni dagli altri.

Il VHDL descrive una realtà di questo tipo.

2.1 Note tecniche generale

Il VHDL è un linguaggio *case insensitive*, cioè non fa differenza tra lettere maiscole e minuscole.

Gli *identificatori* sono nomi iniziano con una lettera e possono avere sia lettere che cifre numeriche nei simboli successivi. Il simbolo di *underscore* "_" è permesso, ma non come primo simbolo di un nome.

I commenti iniziano con un doppio meno "`--`" e occupano tutta la rimanente parte della riga. I commenti sono parti della descrizione che vengono ignorate dal compilatore e che possono quindi essere liberamente utilizzate per documentare il codice.

La formattazione del codice viene usata per migliorare la leggibilità, ma non modifica il significato dei costrutti coinvolti.

2.2 Design Entity

L'unità di base di un modello VHDL consiste nella *Design Entity*, che può rappresentare un intero sistema, un circuito stampato, un circuito integrato oppure una porta logica elementare.

Una Design Unit è costituita da una *Entity Declaration* e da almeno una *Architecture Body*, come schematizzato in fig. 1.

La *Entity Declaration* definisce l'interfaccia del modello.

L'*Architecture Body* definisce la funzionalità del modello. All'interno di un modello VHDL, ad una stessa Entity Declaration possono corrispondere diverse Architecture Body. Ogni diversa architettura rappresenta una diversa realizzazione della stessa funzionalità del modello, per mettere in luce un diverso aspetto progettuale.

Tipicamente, una Design Unit è descritta in un file denominato con l'identificatore della stessa unità. Tuttavia, in molti ambienti di sviluppo questo requisito è allentato, e l'utente può dare al file un nome differente dal nome dell'unità in esso descritta.

Il componente è inoltre corredato da un'eventuale *Configuration Declaration* dove viene specificato quale fra le varie Architecture Body debba essere utilizzata per implementare una Entity. Per gli scopi del corso, tuttavia, le Configuration Declaration sono da considerarsi un argomento avanzato, e non saranno quindi approfondite.

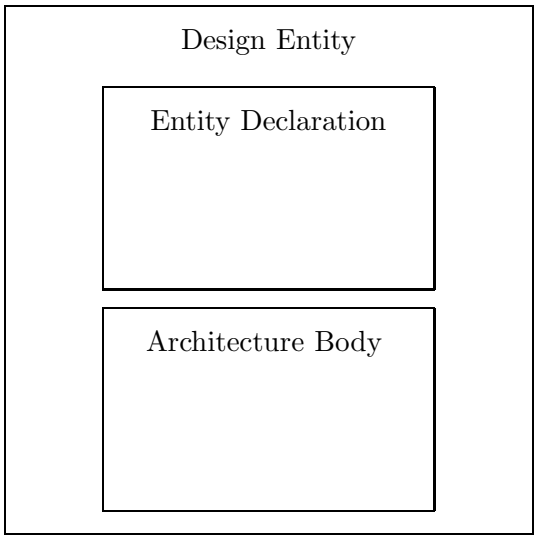


Figura 1: Struttura della Design Entity.

2.3 Entity Declaration

Tramite il costrutto di Entity Declaration viene definita l'interfaccia dell'unità con l'ambiente esterno.
La sintassi di una Entity Declaration può informalmente essere schematizzata come segue:

```
entity ENTITY_NAME is
   port (PORT_LIST); -- lista dei segnali
                     -- di interfaccia;
end [ENTITY_NAME];
```

Gli elementi della dichiarazione sono:

- il nome della entity (qui rappresentato da `ENTITY_NAME`;
- la lista dei segnali di interfaccia (rappresentato da `PORT_LIST`).

Una nota: come già detto, tutto ciò che segue "`--`" fino alla fine della riga viene interpretato dal compilatore come un commento, e, perciò, ignorato.

Costrutto `port`

Il costrutto **`port`** identifica l'insieme dei segnali di interfaccia della entity per comunicare con l'ambiente esterno.
La sintassi è:

```
port (NAME_LIST: MODE SIG_TYPE;
        ...
      NAME_LIST: MODE SIG_TYPE);
```

`MODE` identifica la direzione del segnale, cioè la direzione del flusso dati attraverso una porta. Può assumere i valori **`in`**, **`out`** e **`inout`**. Un segnale in modalità **`in`** può essere solo letto, ad uno in modalità **`out`** può essere solo assegnato un valore, e, infine, un segnale in modalità **`inout`** può essere sia letto che assegnato. Quest'ultima modalità è stata qui introdotta per completezza, ma

non verrà utilizzata nel seguito, in quanto viene usata in casi particolari che esulano dalla presente trattazione.

`SIG_TYPE` identifica il tipo del segnale, cioè un insieme di valori che il segnale può assumere (per esempio, 0 e 1) e un insieme di operazioni che possono essere eseguite sul segnale.

`NAME_LIST` è una lista di identificatori separati da una virgola. Tutti i segnali elencati nella lista saranno del tipo indicato immediatamente dopo la lista stessa.

2.4 Architecture Body

L'Architecture Body descrive la funzionalità del modello attraverso la definizione delle relazioni funzionali tra gli ingressi e le uscite del modello stesso.

La sua sintassi è:

```
architecture BODY_NAME of ENTITY_NAME is
   -- istruzioni dichiarative
begin
   -- istruzioni per descrivere la
   -- funzionalita' del modello
end [BODY_NAME];
```

dove `BODY_NAME` è il nome che viene assegnato alla particolare architettura descritta, mentre `ENTITY_NAME` è l'identificativo della dichiarazione che descrive l'interfaccia.

2.5 Esempio

La specifica ad alto livello di una unità consiste, generalmente, in una descrizione della funzionalità desiderata e dei vincoli che devono essere soddisfatti. Essa viene fornita dal committente e il progettista ne deve realizzare l'equivalente descrizione VHDL.

Per esempio, consideriamo la seguente specifica:

- L'unità riceve in ingresso due segnali digitali e genera in uscita un singolo segnale.
- Se entrambi i segnali di ingresso sono bassi, il segnale di uscita deve essere alto.
- Per ogni altra combinazione degli ingressi, l'uscita deve essere bassa.

A questa descrizione discorsiva può essere fatta seguire una descrizione più formale:

- L'unità ha due segnali di ingresso, `A` e `B`, e un segnale di uscita `C`.
- La relazione tra i segnali di ingresso e di uscita può essere descritto dalla tabella di verità in fig. 2a o dal diagramma dei segnali in fig. 2b.

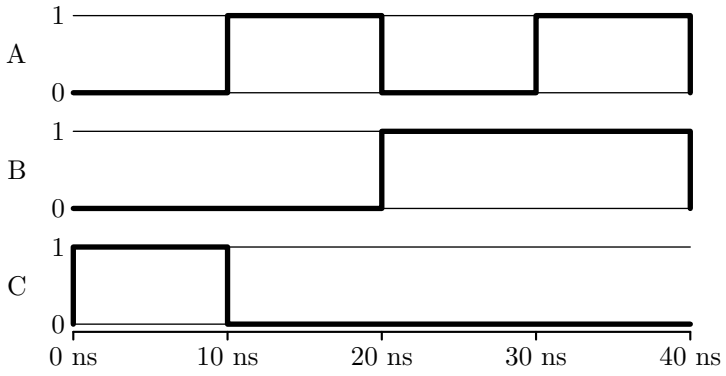
Infine, la descrizione VHDL può essere la seguente:

```
entity NOR_GATE is
   port (A, B : in bit;
             C : out bit);
end NOR_GATE;

architecture DATA_FLOW of NOR_GATE is
```

Ingressi		Uscita
A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

(a)



(b)

Figura 2: Descrizione formale di un componente: (a) tabella di verità; (b) diagramma dei segnali di I/O.

```
begin
   C <= A nor B;
end DATA_FLOW;
```

Il nome dato all'**entity** è NOR_GATE, in quanto la descrizione del comportamento dell'unità data dalle specifiche coincide con quella di una porta NOR (negazione dell'OR). Tuttavia, va sottolineato che il nome non costituisce alcun legame (né sintattico, né semantico) con il comportamento dell'unità: è semplicemente più significativo, per un lettore umano, comprendere il comportamento dell'unità se il nome dell'unità stessa richiama una entità già conosciuta.

Le porte A e B sono state definite di tipo binario (bit) accessibili in ingresso (**in**), mentre la porta C è stata definita binaria e attiva in uscita (**out**).

Il nome dato alla architettura è DATA_FLOW in quanto la descrizione del comportamento dell'unità ivi riportata è di tipo *dataflow*. Valgono anche in questo caso considerazioni analoghe a quelle già fatte per l'identificatore dell'**entity**.

L'istruzione C <= A **nor** B assegna al segnale C il valore risultante dalla combinazione dei segnali A e B mediante l'operatore **nor**. I dettagli di questo tipo di istruzione saranno trattati nel seguito. Per ora, è importante cogliere questa istruzione come esempio di descrizione del comportamento dell'unità.

2.6 Rappresentazione dei dati

Un sistema hardware può essenzialmente essere visto come un manipolatore di segnali.

In tal senso, i dati rappresentati all'interno di una descrizione VHDL sono segnali (digitali) e le caratteristiche ad essi associate.

Gli elementi utilizzati per rappresentare i dati sono le costanti, i segnali e le variabili, che nel loro insieme vengono chiamati *oggetti*. In questo contesto, ogni oggetto è definito da un identificatore, un tipo e da un valore. L'identificatore e il valore vengono definiti nella fase dichiarativa, mentre il valore può essere assegnato tramite un'opportuna operazione di assegnamento, eventualmente nella fase dichiarativa (nel qual caso ci si riferisce a tale situazione con il termine *inizializzazione*).

Costanti

Le costanti sono elementi ai cui viene assegnato un valore iniziale, senza possibilità di successiva modifica.

La sintassi per dichiarare una costante è:

```
constant CONST_LIST : CONST_TYPE [:= EXPR] ;
```

dove `CONST_LIST` è una lista di identificatori, separati dalla virgola, `CONST_TYPE` indica il tipo della costante, e `:= EXPR` indica l'eventuale costrutto di inizializzazione (in alcuni casi il valore della costante può essere specificato separatamente dalla sua dichiarazione).

Per esempio, il seguente frammento di codice:

```
constant REG_LEN: integer := 16;
```

descrive la dichiarazione di una costante intera, di nome `REG_LEN` e di valore 16.

Le costanti sono utili soprattuto per rendere più leggibile e facilmente portabile il codice: un certo valore viene assegnato ad una costante e nel codice si usa poi solo l'identificativo della costante, senza indicare ogni volta esplicitamente tale valore.

Segnali

I segnali sono oggetti che modellano i canali che trasportano i dati da un componente del sistema ad un altro. I loro valori sono quindi soggetti a cambiare nel tempo (eventualmente con un certo ritardo rispetto ad un evento di riferimento), e possono essere cambiati attraverso un'istruzione di assegnamento tra segnali:

La sintassi per dichiarare un segnale è:

```
signal SIG_LIST : SIG_TYPE [:= EXPR] ;
```

dove `SIG_LIST` è una lista di identificatori, separati dalla virgola, `SIG_TYPE` indica il tipo del segnale, e `:= EXPR` indica l'eventuale costrutto di inizializzazione.

Per esempio, il seguente frammento di codice:

```
signal A, Z : bit;
```

descrive la dichiarazione di due segnali di tipo binario, di nome `A` e `Z`.

Il seguente frammento di codice VHDL descrive invece un assegnamento di un segnale, con un ritardo di 1 ms:

```
SIG_A <= NEW_VALUE after 1 ms;
```

L'assegnamento di segnali necessita di qualche altra considerazione. L'argomento sarà ripreso nel seguito.

Variabili

Le variabili sono oggetti che ricevono un valore che può essere cambiato attraverso un istruzione di assegnamento tra variabili.

La sintassi per dichiarare una variabile è:

```
variable VAR_LIST : VAR_TYPE [:= EXPR] ;
```

dove `VAR_LIST` è una lista di identificatori, separati dalla virgola, `VAR_TYPE` indica il tipo della variabile, e `:= EXPR` indica l'eventuale costrutto di inizializzazione.

Per esempio, il seguente frammento di codice:

```
variable X: boolean;
```

descrive la dichiarazione di una variabile di tipo booleano, di nome `X`.

Il seguente frammento di codice VHDL descrive invece un assegnamento di una variabile:

```
VAR_A := NEW_VALUE;
```

La funzione delle variabili è quella di conservare un determinato valore (numerico, booleano), cosa che un segnale, per sua natura non è in grado di fare. Esse sono, da questo punto di vista, l'equivalente delle variabili dei comuni linguaggi di programmazione. L'utilizzo di una variabile non sempre ha senso e viene permesso solo in alcuni ambiti, i costrutti sequenziali, i quali saranno meglio specificati nel seguito (vedi § 3.2).

Tipi di dato

Il tipo di un dato definisce i valori che un oggetto di un dato tipo può assumere e le operazioni che su esso possono essere effettuate.

I tipi possono classificarsi in due categorie: i tipi *scalari* e i tipi *composti*. Un tipo scalare contiene un valore che appartiene ad un determinato intervallo (detto anche *range*) o ad un insieme enumerativo. Un tipo composto è costituito da una aggregazione di elementi di tipo scalare. Se gli elementi sono dello stesso tipo, si ha un *array*, altrimenti il tipo viene detto *record*.

Fra i tipi scalari, i principali tipi standard sono: `boolean`, `bit`, `integer` e `time`.

Gli oggetti di tipo `boolean` possono assumere i valori `'false'` o `'true'` e vengono usati per il controllo del flusso di esecuzione.

Gli oggetti di tipo `bit` possono assumere i valori `'0'`, `'1'` e sono usati per modellare i segnali binari. Da notare che, sebbene matematicamente omeomorfo, il tipo `boolean` non può essere usato per modellare tali segnali.

Gli oggetti di tipo `integer` possono assumere valori in un intervallo che dipende dall'implementazione. Essi modellano i numeri interi e la loro rappresentazione deve essere ad almeno 32 bit.

Il tipo `time` è un tipo di dato speciale ed è costituito da un valore numerico e da un'unità di misura del tempo (per esempio, `ms`). Viene utilizzato per modellare i tempi, per esempio per i ritardi di propagazione. Un oggetto di tipo `time` può essere moltiplicato per un oggetto `integer`, ottenendo ancora un valore di tipo `time`.

Il VHDL è un linguaggio fortemente tipizzato. L'assegnamento tra segnali di diverso tipo richiede l'uso di appositi convertitori.

Tipi di dati definiti dall'utente

Il VHDL consente all'utente di definire tipi di dati, funzioni, operazioni, e componenti che possono poi essere utilizzati in modo trasparente, come se fossero elementi standard del linguaggio. È inoltre consentito di articolare la collezione di tali descrizioni in moduli definiti *package* e *library*. Un package è assimilabile ad una Design Unit, mentre una library è una raccolta precompilata di Design Unit.

L'utente può utilizzare un elemento di una library indicandone l'utilizzo mediante un'opportuna dichiarazione.

La dichiarazione **`library`** rende visibile ad un modello VHDL la libreria selezionata che contiene i package desiderati.

La dichiarazione **`use`** rende un package visibile ad un modello.

Per esempio, le dichiarazioni:

```
library IEEE;
use IEEE.Std_Logic_1164.all;
```

permettono di utilizzare tutti gli elementi descritti nel package `IEEE.Std_Logic_1164`. Essi sono molto importanti in quanto rappresentano un standard molto diffuso.

Tipi di dati in `IEEE.STD_LOGIC_1164`

I tipo di segnale standard è il `bit`, che descrive un segnale binario. Tuttavia, nella realtà è possibile che si verifichino situazioni in cui non è possibile assegnare con certezza un segnale ad uno dei due valori binari (per esempio, nel tri-state). Inoltre, usando il tipo `bit` non è possibile utilizzare alcune operazioni aritmetiche.

I tipi `STD_ULOGIC` e `STD_LOGIC` definiti in `IEEE.STD_LOGIC_1164` pongono rimedio a questa mancanza, definendo un tipo di segnale che può assumere nove valori:

`'U'` *undefined*;
`'X'` *forcing unknown*;
`'0'` *forcing 0*;
`'1'` *forcing 1*;
`'Z'` *high impedance*;
`'W'` *weak unknown*;
`'L'` *weak 0*;
`'H'` *weak 1*;
`'-'` *don't care*.

Nella pratica, è possibile ottenere situazioni in cui non sia chiaro che valore assegnare ad un segnale (per esempio, usando un segnale non opportunamente inizializzato). Questo tipo di difficoltà, inoltre, può non essere rilevabile al momento della compilazione, ma solo a *run-time* (per esempio, in presenza di assegnamenti multipli, come nel caso di un bus).

La differenza tra `STD_ULOGIC` e `STD_LOGIC` è che quest'ultimo risolve i conflitti (U sta per *unresolved*!), secondo lo schema seguente:

Lo schema va interpretato come segue: in caso di assegnamento multiplo, il valore che viene assegnato è quello che sta al livello più alto, o, nel caso in cui siano coinvolti segnali dello stesso livello, il valore del livello immediatamente superiore. Per esempio, sia data la seguente situazione, dove i segnali `A` e `B` vengono contemporaneamente assegnati a `C`:

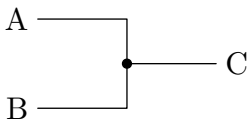


Figura 3: Assegnamento multiplo.

```
C <= A;
C <= B;
```

Questa descrizione VHDL è descritta dallo schematico in fig. 3. Nel caso in cui `A` valga `'1'` e `B` valga `'Z'`, al segnale `C` viene assegnato il valore `'1'`, mentre se `B` valesse `'0'`, a `C` verrebbe assegnato il valore `'X'`.

I segnali `A`, `B` e `C` devono essere di tipo `STD_LOGIC` per ottenere la risoluzione automatica dei conflitti di assegnamento. Va inoltre sottolineato che i segnali di tipo `bit` non ammettono l'assegnamento multiplo.

Vettori

Spesso viene comodo aggregare segnali dello stesso tipo. A tal proposito, il VHDL mette a disposizione gli **`array`**.

Un array è caratterizzato da un tipo base e da un intervallo di indicizzazione.

Un tipo **`array`** può essere definito usando la seguente sintassi:

```
type <type_name> is array <range> of <base_type>;
```

dove `<type_name>` e `<base_type>` sono rispettivamente gli identificatori dei tipi aggregato e base, mentre **`<range>`** descrive l'intervallo degli indici dell'array con una sintassi del tipo:

```
(0 to 3)

(5 downto 2)
```

Nel primo caso, gli elementi dell'array sono accessibili con gli indici che vanno da 0 a 3, mentre con la seconda dichiarazione gli stessi elementi sono accessibili tramite la numerazione a ritroso che va da 5 a 2.

Sono definiti i tipi `bit_vector`, `std_logic_vector` e `std_ulogic_vector`, che descrivono rispettivamente gli array di tipo `bit`, `std_logic` e `std_ulogic`.

Alcuni esempi d'uso:

```
port (a: in std_logic_vector(0 to 3);
      b: out bit_vector(3 downto 1));

type T_DATA is array (0 to 3) of
         std_logic_vector(7 downto 0);
constant DATA : T_DATA := ("00000000",
                           "00000001",
                           "00000010",
                           "00000100");
```

Nota: per costanti binarie si usano gli apici (`'1'`), mentre per i vettori di due o più bit si usano le doppie virgolette (`"001"`).

2.7 Operazioni

Assegnamento

I segnali sono caratteristici della descrizione hardware per modellare l'informazione che transita nel tempo su un singolo canale oppure su un bus.

L'istruzione di assegnamento tra segnali stabilisce un collegamento definitivo tra di essi. Esempio:

```
signal A, Z : bit;

Z <= A;
```

Come già sottolineato, in presenza di istruzioni di assegnamento multiple è necessaria una funzione di risoluzione. Il seguente frammento di codice darà problemi al momento della simulazione o della sintesi:

```
signal A, B, Z: bit;
Z <= A;
Z <= B;
```

Usando `std_logic` invece di `bit`, i problemi non ci dovrebbero essere.

Per i vettori, l'assegnamento:

```
signal Z_BUS: bit_vector(3 downto 0);
signal A_BUS: bit_vector(1 to 4);
-- ...
Z_BUS <= A_BUS;
```

equivale a:

```
Z_BUS(3) <= A_BUS(1); -- referenziazione per indici
Z_BUS(2) <= A_BUS(2);
Z_BUS(1) <= A_BUS(3);
Z_BUS(0) <= A_BUS(4);
```

ed equivale a:

```
Z_BUS(3 downto 1) <= A_BUS(1 to 3); -- bit-slicing
Z_BUS(0) <= A_BUS(4);
```

Composizione di segnali

I segnali si possono comporre per formare un bus tramite due operatori: concatenazione e aggregazione.

La concatenazione si opera solo sul lato destro dell'assegnamento:

```
architecture EXAMPLE of CONCATENATION is
signal BYTE : bit_vector (7 downto 0);
signal A_BUS, B_BUS : bit_vector (2 downto 0);
signal C, D : bit;
begin
  BYTE   <= A_BUS & C & B_BUS & D;
end EXAMPLE;
```

Il segnale concatenato si descrive mediante una lista di segnali, separata dall'operatore `&`.

L'aggregazione permette di costruire un array indicando il contenuto dei singoli segnali o gruppi di essi:

```
architecture EXAMPLE of AGGREGATES is
  signal BYTE :  bit_vector (7 downto 0);
  signal Z_BUS : bit_vector (2 downto 0);
  signal A_BIT, B_BIT, C_BIT, D_BIT : bit;
begin
  Z_BUS <= ( A_BIT, B_BIT, C_BIT);
  ( A_BIT, B_BIT, C_BIT, D_BIT ) <= bit_vector("1011");
  ( A_BIT, B_BIT, C_BIT, D_BIT ) <= BYTE(3 downto 0);
  BYTE <= ( 7 => '1', 5 downto 1 => '1', 6 => B_BIT, others => '0') ;
end EXAMPLE;
```

L'aggregazione viene descritta elencando i segnali (o i valori) in una lista, separata dalla virgola e racchiusa tra parentesi.

Nel primo caso dell'esempio, `Z_BUS <= (A_BIT, B_BIT, C_BIT, D_BIT)`, i segnali componenti vengono elencati in ordine: le informazioni riportate nelle sezioni dichiarative indicano a quali elementi del segnale composto verranno assegnati gli elementi dei segnali componenti.

Nel secondo caso, `(A_BIT, B_BIT, C_BIT, D_BIT) <= bit_vector("1011")`, nella parte destra viene indicato un segnale di `tipo bit_vector` e di valore `"1011"`. Tale valore viene assegnato al segnale ottenuto componendo opportunamente i segnali indicati nella parte sinistra dell'assegnamento.

Nel terzo caso, `(A_BIT, B_BIT, C_BIT, D_BIT) <= BYTE(3 downto 0)`, al segnale aggregato viene assegnato un sottoinsieme del segnale `BYTE`.

Ed infine, nel quarto caso, `BYTE <= (7=>'1', 5 downto 1=>'1', 6=>B_BIT, others=>'0')`, il segnale aggregato viene costruito indicando esplicitamente, elemento per elemento (o per intervalli), il valore assunto. **`others`** indica tutti gli elementi non ancora assegnati.

Operatori

Gli operatori si dividono in *logici*, *relazionali* e *aritmetici*.

Gli operatori *logici* sono definiti sui tipi `bit`, `bit_vector`, `std_logic`, `std_logic_vector` e `boolean`. Essi sono: **`AND`**, **`NAND`**, **`OR`**, **`NOR`**, **`XOR`**, **`NOT`**.

Gli operatori logici richiedono operandi confrontabili, dello stesso tipo e della stessa lunghezza. Le operazioni logiche vengono effettuate elemento per elemento. Le parentesi possono essere utilizzate per indicare la priorità delle operazioni. Il risultato dell'operazione è dello stesso tipo degli operandi.

Gli operatori *relazionali* danno un risultato di tipo `boolean` e sono definiti su tutti i tipi. Essi sono: `=`, `/=`, `<`, `>`, `<=`, `>=`.

Gli operatori relazionali sui vettori sono definiti bit a bit, allineati a destra. Ciò significa che l'espressione `"111"> "1011"` vale `true`, perché `"111"` si confronta con `"011"`.

Infine, gli operatori *aritmetici* sono definiti per i tipi numerici (per esempio, `integer`) e fisici (per esempio, `time`). Essi sono: `+`, `-`, `*`, `/`, `**`, **`abs`**, **`mod`**, **`rem`**.

Da notare che i tipi `bit` e `bit_vector` non supportano le operazioni aritmetiche. Il codice seguente:

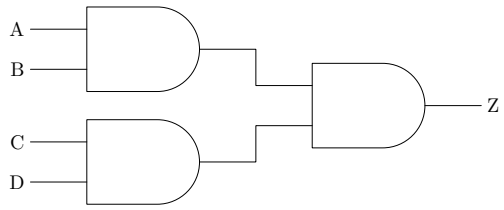


Figura 4: Rappresentazione schematica di una porta AND a quattro ingressi.

```
architecture myarch of esempio is
  signal t : bit_vector(0 to 7);
begin
   t <= t + "00000001";
end myarch;
```

genera un errore.

I tipi `std_logic`, `std_ulogic`, `std_logic_vector` e `std_ulogic_vector` invece le supportano. Il codice seguente:

```
architecture myarch of esempio is
  signal t : std_logic_vector(0 to 7);
begin
   t <= t + "00000001";
end myarch;
```

è perfettamente ammissibile.

2.8 Concorrenza

Le strutture hardware sono intrinsecamente concorrenti. Le attività sono infatti svolte in parallelo dai diversi componenti. Questa caratteristica si ritrova nel VHDL, dove l'ordine fra i costrutti che descrivono i componenti non ha alcun effetto. In altri termini, tutte le operazioni descritte in una **`architecture`** vengono cosiderate contemporaneamente.

Per esempio, si consideri il circuito schematizzato in fig. 4. A meno di ritardi di propagazione dei segnali, le tre porte logiche operano in parallelo.

Ciò puo essere descritto in VHDL come:

```
entity AND4 is
   port (A, B, C, D: in bit;
                  Z: out bit);
end AND4;

architecture DATA_FLOW of AND4 is
  signal T1, T2: bit;
begin
  T1 <= A and B;
  T2 <= C and D;
  Z <= T1 and T2;
end DATA_FLOW;
```

Da notare i segnali `T1` e `T2`, interni al componente, che descrivono i collegamenti fra le porte

logiche di primo livello e la porta logica di secondo livello dello schema in fig. 4. Essi sono definiti tramite il costrutto `signal T1, T2: bit;` e non possono essere acceduti dall'esterno.

Nella realtà i segnali non si propagano istantaneamente, e, in particolare, le porte logiche introducono dei ritardi di propagazione. Questo aspetto può essere modellato tramite gli appositi costrutti di temporizzazione (trattati al paragrafo 4.).

2.9 Descrizione strutturale

Le strutture hardware sono spesso descrivibili come interconnessione di componenti elementari.

Questo aspetto può essere catturato tramite lo stile di descrizione strutturale. L'uso di questo stile non solo permette di descrivere un sistema come concorrenza di componenti, ma permette anche di riusare parti di codice VHDL precedentemente scritte. Il riuso del codice è un aspetto comune di molti linguaggi formali e un punto di forza della metodologia top-down.

Dichiarazione `port map`

Per usare un componente bisogna dichiararlo nella sezione dichiarativa della **`architecture`**.

Una volta dichiarato, lo si può utilizzare (anche ripetuto in più occorrenze) all'interno della sezione descrittiva della **`architecture`**. Per far ciò, è necessario indicare i collegamenti ai segnali dichiarati nella sezione **`port`** del componente. Per questa operazione viene utilizzata la dichiarazione **`port map`**.

Ci sono due modi per indicare i segnali da collegare al componente:

- si indica un elenco di identificatori dei segnali e questi verranno assegnati nello stesso ordine ai segnali indicati nel **`port`** del componente;
- si elenca una lista di assegnamenti espliciti.

Entrambi i metodi sono illustrati nel seguente paragrafo, con un esempio.

Esempio

Il seguente esempio fornisce la descrizione strutturale di un latch:

```
entity latch is
  port ( s, r: in bit;
         q, nq: out bit);
end latch;

architecture structure of latch is

   component nor_gate
      port (a,b: in bit;
              c: out bit);
   end component;

   signal t1, t2: bit;

begin
```

```
   n1: nor_gate port map (r, t1, t2);          -- elenco identificatori
   n2: nor_gate port map (b=>t2, a=>s, c=>t1); -- assegnamenti espliciti
   nq <= t1;
   q <= t2;
end structure;
```

3. Costrutti

3.1 Costrutti concorrenti

I costrutti concorrenti sono utilizzabili solo nelle sezioni concorrenti della descrizione di una **architecture**.

Assegnamento selettivo

Il costrutto **with-select-when** è utilizzabile all'interno di una descrizione concorrente. Esso permette di effettuare un dato assegnamento in base al verificarsi di una data condizione.

La sua sintassi è la seguente:

```
with <expression> select
<signal_name> <= <signal_or_value> when <condition1>,
                 <signal_or_value> when <condition2>,
                  ...
                 <signal_or_value> when others;
```

Le condizioni devono essere mutuamente esclusive e tutti i valori possibili dell'espressione devono essere coperti.

Esempio:

```
  entity and2 is
     port (a : in bit_vector (1 downto 0);
           y : out bit);
  end and2;

  architecture arch_and2 of and2 is
  begin
     with a select
        y <=  '1' when "11",
              '0' when "00",
              '0' when "01",
              '0' when "10";
  end arch_and2;
```

L'uso di **others** rende la descrizione più sintetica e leggibile, come si può apprezzare all'aumentare delle possibili combinazioni:

```
  entity and12 is
     port (a : in bit_vector (11 downto 0);
           y : out bit);
  end and12;
```

```
architecture arch_and12 of and12 is
begin
   with a select
      y <=  '1' when "111111111111",
            '0' when others;
end arch_and12;
```

Assegnamento condizionato

Il costrutto **when-else** è utilizzabile all'interno di una descrizione concorrente. Esso permette di assegnare un assegnamento al verificarsi di una condizione.

La sua sintassi è:

```
<signal_name> <= <signal_or_value> when <cond1> else
                 <signal_or_value> when <cond2> else
                 -- ...
                 <signal_or_value> when <condN> else
                 <signal_or_value>;
```

Le condizioni vengono valutate in sequenza e ci si ferma alla prima verificata. Inoltre, deve sempre esistere un assegnamento non condizionato.

Esempio:

```
entity and2 is
   port (a, b : in bit;
             y : out bit);
end and2;

architecture arch_and2 of and2 is
begin
   y<='1' when (a='1' and b='1') else
      '0';
end arch_and2;
```

C'è sempre più di un modo per descrivere la stessa cosa:

```
entity and2 is
       port (a: in bit_vector (1 downto 0);
             y : out bit);
end and2;

architecture arch_and2_bis of and2 is
begin
   y<='0' when a(0)='0' else
      '0' when a(1)='0' else
      '1';
end arch_and2_bis;
```

oppure:

```
entity and2 is
```

```
       port (a: in bit_vector (1 downto 0);
             y : out bit);
end and2;

architecture arch_and2_ter of and2 is
begin
   y<='1' when a="11" else
      '0';
end arch_and2_ter;
```

3.2 Costrutti sequenziali

Alcune operazioni sono troppo complesse per poter essere descritte da una singola istruzione o costrutto.

I *processi* sono costrutti che permettono di descrivere un'azione mediante una sequenza di istruzioni.

Un modello VHDL è quindi un insieme di processi che interagiscono tra loro in parallelo (concorrenza tra processi) costituiti da istruzioni sequenziali. I processi si scambiano informazioni tramite i segnali.

Il processo è il costrutto principale per la modellazione di componenti sequenziali.

La sintassi della descrizione di un processo è:

```
[<label>:] process [(<sensitivity_list>)]
begin
   <seq_instr1>
   ...
   <seq_instrN>
end process [<label>];
```

Ci sono due modi per controllare l'esecuzione di un processo: tramite il costrutto **wait**, che sarà trattato nel paragrafo 4.1, e tramite la *sensitivity list*.

Quando avviene un *evento* su un segnale indicato nella *sensitivity list* di un processo, quest'ultimo viene eseguito. Si definisce *evento* su un segnale quando il segnale stesso cambia valore.

L'esecuzione di un processo può causare uno o più eventi alla sua terminazione (e solo allora). Tali eventi possono attivare altri processi. Le eventuali variazioni dei segnali che avvengono nei processi prendono corpo solo al termine del processo. Questo perché il processo deve essere inteso come un'unica istruzione.

La sensitivity list viene utilizzata solo durante la simulazione, non durante la sintesi. Non è detto che il componente descritto tramite un processo possa essere direttamente sintetizzato. Tuttavia è comodo (e più veloce) esprimere il comportamento di un componente mediante un processo. Non è detto infatti che tale componente vada effettivamente realizzato: per esempio, potrebbe servire solo per descrivere un elemento esterno al progetto, ma con il quale è necessario interagire. Oppure, applicando una metodologia top-down, la descrizione comportamentale può essere realizzata in una fase iniziale del progetto, per poi essere raffinata. È quindi importante che durante la simulazione il modello abbia un comportamento il più fedele possibile al comportamento dopo la sintesi.

Affinché il componente sintetizzato abbia lo stesso comportamento del suo modello simulato, la sensitivity list deve contenere tutti i segnali che vengono letti all'interno del processo stesso.

Esempio

Qual'è la differenza di comportamento tra i due processi?

```
process (A, B, SEL)
begin
   if (SEL = '1') then
      OUT <= A;
   else
      OUT <= B;
   end if;
end process;
```

```
process (A, B)
begin
   if (SEL = '1') then
      OUT <= A;
   else
      OUT <= B;
   end if;
end process;
```

La sintesi produrrà lo stesso circuito, ma nella simulazione il primo processo sarà in grado di attivarsi anche su eventi del segnale `SEL`.

Costrutto `if-then-else`

`if-then-else` è un costrutto *sequenziale* (cioé può essere impiegato solo all'interno di un costrutto quale **`process`**).

La sua sintassi è la seguente:

```
if <condition>
  then <instruction1>;
  else <instruction2>;
end if;
```

La condizione `<condition>` deve essere un'espressione booleana.

Tale costrutto può essere esteso (**`if-then-elsif`**):

```
if <condition1>
  then <instruction1>;
  elsif <condition2>
  then <instruction2>;
  else <instruction3>;
end if;
```

Il seguente frammento di codice esemplifica l'uso del costrutto **`if-then-elsif`**:

```
architecture ARC1 of DEC24 is
begin
  P1: process (A, B)
  begin
    if (A='0' and B='0') then
      Z <= "1000";
    elsif (A='0' and B='1') then
      Z <= "0100";
    elsif (A='1' and B='0') then
      Z <= "0010";
    elsif (A='1' and B='1') then
      Z <= "0001"; end if;
  end process P1;
end ARC1;
```

Va sottolineato che l'ordine con cui le condizioni vengono valutate forza la priorità sulle istruzioni eseguite.

Costrutto `case-when`

Il costrutto **`case-when`** va utilizzato quando tutte le condizioni sono mutuamente esclusive e hanno la stessa priorità.

La sua sintassi è:

```
case <selection_signal> is
   when <value_1_selection_signal> => <instr_1>;
   when <value_2_selection_signal> => <instr_2>;
   when <value_3_selection_signal> => <instr_3>;
   ...
   when <value_n_selection_signal> => <instr_n>;
   when others => <instruction_others>;
end case;
```

Tutti i possibili valori del segnale di selezione devono essere coperti, eventualmente ricorrendo alla clausola **`others`**.

Esempio:

```
architecture ARC2 of DEC24 is
begin
  P2: process (A, B)
  begin
    case (A & B) is
      when "00" => Z <= "1000";
      when "01" => Z <= "0100";
      when "10" => Z <= "0010";
      when "11" => Z <= "0001";
    end case;
  end process P2;
end ARC2;
```

Costrutto `loop`

I costrutti ciclici sono comuni nei linguaggi di programmazione. Essi, in quel contesto, sono utilizzati per ripetere sequenza di istruzioni.

Va sottolineata la più evidente differenza con i cicli in VHDL: poiché il VDHL descrive hardware, un ciclo descrive una ripetizione di uno schema circuitale.

L'uso di questi costrutti, per gli scopi di questo corso, va quindi limitato e ponderato.

Esistono diversi costrutti ciclici, il cui significato è autoesplicativo:

- **`loop`**

```
loop
   <instruction>;
   exit when <condition>;
end loop;
```

- **while-loop**

```
while (<condition>) loop
   <instruction>;
end loop;
```

- **for-loop**

```
for <var> in <min_value> to <max_value> loop
   <instruction>;
end loop;

for <var> in <max_value> downto <min_value> loop
   <instruction>;
end loop;
```

Le variabili usate per controllare il numero di iterazioni, devono essere di tipo `integer`.

4. Temporizzazione

Per modellare i ritardi di propagazione dei segnali, il VHDL mette a disposizione dei costrutti temporali.

Il principale è il costrutto **`after`** che consente di introdurre un ritardo nell'assegnamento di un segnale.

La sintassi è esemplificata nel seguente frammento di codice:

```
NEW_SIGNAL <= SIGNAL_EXPR after TIME_PERIOD;
```

dove `TIME_PERIOD` deve essere di tipo `time`.

Il ritardo può inoltre essere seguito da un altro assegnamento, disegnando così una forma d'onda.

Per esempio:

```
C <= A and B after 10 ns;
D_OUT <=  1  after 10 ns,
          0  after 20 ns,
          1  after 30 ns,
          0  after 40 ns;
Q <=  1  after 10 ns when B =  1  else
      0  after 5 ns;
```

4.1 Costrutti `wait`

Il costrutto **`wait`** è molto utile per descrivere specifiche temporali. Esso può essere usato per controllare un processo in alternativa alla sensitivity list.

Esistono diverse forme per questo costrutto:

- **`wait on`**, attende il cambiamento di uno o più segnali;

- **`wait until`**, attende il verificarsi di una espressione booleana;
- **`wait for`**, attende un determinato intervallo di tempo.

Un processo senza sensitivity list (quindi controllato da **`wait`**) viene mandato in esecuzione subito, e, una volta eseguita l'ultima istruzione, riprende ad essere eseguito, finché non incontra una istruzione **`wait`**.

Questa caratteristica può essere sfruttata, per esempio, per generare segnali periodici. Il seguente frammento di codice descrive un'onda quadra con periodo 20 ns:

```
clock_p : process
begin
  clock <= '0';
  wait for 10 ns;
  clock <= '1';
  t <= X;
  wait for 10 ns;
end process;
```

4.2 Attributo `event`

Ogni segnale ha una serie di attributi ad esso associati, denotati dal costrutto:

- `<nome_segnale>'<nome_attributo>`

Uno dei più usati è l'attributo booleano `event`. Esso assume il valore `true` solo nell'istante in cui il segnale associato cambia di valore (si verifica un evento sul segnale).

Per esempio, `event` può essere utilizzato per individuare un fronte di salita:

- l'espressione booleana `clock'event and clock='1'` assume il valore `true` solo in occasione del fronte di salita (`clock` cambia valore ed assume il valore `'1'`).

5. Componenti combinatori

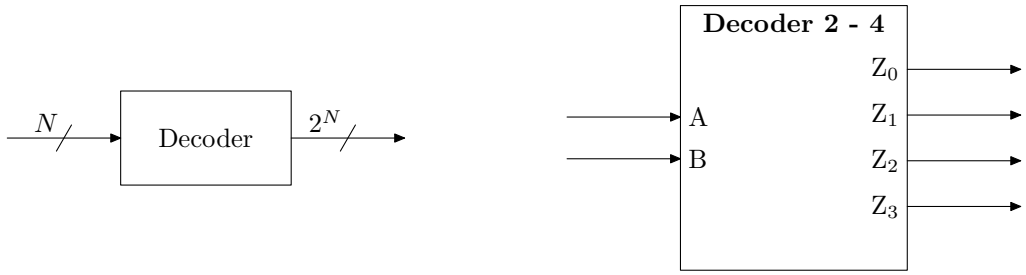
5.1 Decoder

Il decoder è un componente dotato di N ingressi e 2^N uscite. Le uscite sono poste tutte a "0" tranne quella corrispondente al numero binario in ingresso, la quale viene posta ad "1".

Ad esempio, per $N = 2$, il funzionamento del decoder è rappresentato dalla seguente tabella di verità:

A	**B**	$\mathbf{Z}_0$	$\mathbf{Z}_1$	$\mathbf{Z}_2$	$\mathbf{Z}_3$
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

La descrizione VHDL di tale componente è la seguente:



(a) (b)

Figura 5: Rappresentazione schematica di un decoder: (a) a N ingressi; (b) a due ingressi e quattro uscite.

```
entity DEC24 is
  port (A, B: in bit;
        Z: out bit_vector (0 to 3));
end DEC24;
```

La descrizione del funzionamento del decoder può essere espressa dal seguente codice:

```
-- modello comportamentale --
architecture ARC1 of DEC24 is
begin
  P1: process (A, B)
  begin
    if (A='0' and B='0') then
      Z <= "1000";
    elsif (A='0' and B='1') then
      Z <= "0100";
    elsif (A='1' and B='0') then
      Z <= "0010";
    elsif (A='1' and B='1') then
      Z <= "0001";
    end if;
  end process P1;
end ARC1;
```

Alternativamente, per evidenziare la mutua esclusività delle configurazioni dei segnali di ingresso, è possibile utilizzare il seguente codice:

```
-- modello comportamentale --
architecture ARC2 of DEC24 is
  signal t : bit_vector(0 to 1);
begin
  t <= A & B;
  P2: process (t)
  begin
    case t is
      when "00" => Z <= "1000";
      when "01" => Z <= "0100";
```

```
      when "10" => Z <= "0010";
      when "11" => Z <= "0001";
    end case;
  end process P2;
end ARC2;
```

Il corrispondente modello "dataflow" è il seguente:

```
-- modello data-flow --
architecture DATA_FLOW1 of DEC24
is
begin
  Z <= "1000" when (A='0' and B='0')
       else "0100" when (A='0' and B='1')
       else "0010" when (A='1' and B='0')
       else "0001";
end DATA_FLOW1;
```

Oppure:

```
-- modello data-flow --
architecture DATA_FLOW2 of DEC24 is
  signal t : bit_vector(0 to 1);
begin
  t <= A & B;
  with t select
    Z <= "1000" when "00",  -- 0
         "0100" when "01",  -- 1
         "0010" when "10",  -- 2
         "0001" when "11";  -- 3
end DATA_FLOW2;
```

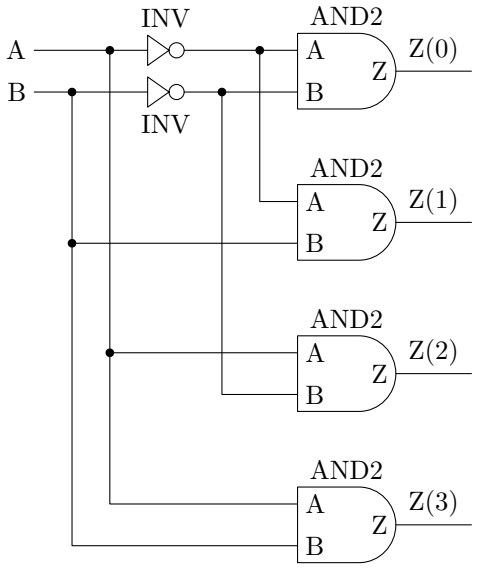


Figura 6: Rappresentazione circuitale del decoder a due vie

In alternativa, considerando lo schema in fig. 6 (derivato tramite le note tecniche di sintesi dei circuiti combinatori) il decoder può essere descritto tramite la seguente architettura dataflow:

```
-- modello data-flow --
architecture DATA_FLOW3 of DEC24 is
begin
  Z(0) <= (not A and not B);
  Z(1) <= (not A and B);
  Z(2) <= (A and not B);
  Z(3) <= (A and B);
end DATA_FLOW3;
```

Inoltre, considerando le porte logiche come componenti, è possibile dare la seguente descrizione strutturale del decoder in esame:

```
-- modello strutturale --
architecture STRUCT of DEC24 is
  component INV
    port (A: in bit;
          Z: out bit);
  end component;
  component AND2
    port (A, B: in bit;
          Z: out bit);
  end component;
  signal NOT_A, NOT_B: bit;
begin
  u1: INV port map (A, NOT_A);
  u2: INV port map (B, NOT_B);
  u3: AND2 port map (NOT_A, NOT_B, Z(0));
  u4: AND2 port map (NOT_A, B, Z(1));
  u5: AND2 port map (A, NOT_B, Z(2));
  u6: AND2 port map (A, B, Z(3));
end STRUCT;
```

Va sottolineato il fatto che il precedente modello è da intendersi come un *esempio* di utilizzo dei costrutti sintattici per la descrizione strutturale dei componenti. Le porte logiche sono infatti descrivibili (più efficentemente e con maggiore leggibilità) mediante gli operatori logici, come mostrato nell'**architecture** DATA_FLOW2 precedentemente riportata.

Decoder BCD 7 segmenti

Il concetto di decoder può essere generalizzato, ricordando che la codifica è sempre una scelta arbitraria. La codifica BCD (*Binary Coded Decimal*) viene utilizzata per codificare i simboli "0", "1", ..., "9", cioè le cifre utilizzate nella notazione posizionale decimale per la rappresentazione dei numeri. Dato che vi sono dieci simboli da codificare, la codifica BCD consta di dieci diverse configurazioni di una stringa di 4 bit:

numero	*codifica BCD*
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

I display a sette segmenti sono comunemente utilizzati per la visualizzazione di valori numerici (vedi fig. 7).

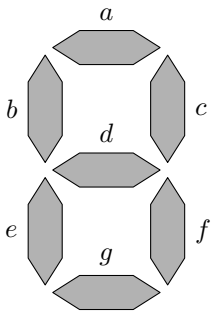


Figura 7: Display a sette segmenti

Un decoder BCD - 7 segmenti è un componente che riceve in ingresso un segnale codificato in BCD e invia in uscita un segnale in grado di attivare la corrispondente configurazione di led. Indicando ognuno dei sette led con un segnale differente, i led possono essere accesi e spenti portando sulla linea corrispondente i valori "1" e "0", rispettivamente. Per esempio, al segnale in ingresso "0010" ("2" codificato in BCD) viene fatto corrispondere il segnale in uscita "1011101" (accesi i led $\{a$, c, d, e, $g\}$, spenti gli altri).

La tabella di verità del decoder BCD - 7 segmenti è quindi la seguente:

numero	codice BCD				led accesi						
	$\mathbf{X}_0$	$\mathbf{X}_1$	$\mathbf{X}_2$	$\mathbf{X}_3$	***a***	***b***	***c***	***d***	***e***	***f***	***g***
0	0	0	0	0	1	1	1	0	1	1	1
1	0	0	0	1	0	0	1	0	0	1	0
2	0	0	1	0	1	0	1	1	1	0	1
3	0	0	1	1	1	0	1	1	0	1	1
4	0	1	0	0	0	1	1	1	0	1	0
5	0	1	0	1	1	1	0	1	0	1	1
6	0	1	1	0	1	1	0	1	1	1	1
7	0	1	1	1	1	0	1	0	0	1	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	1	1

La descrizione VHDL di tale componente (dove i segnali a, ..., g sono stati rinominati Z(0), ..., Z(6)) è la seguente:

```
entity DEC_BCD_7 is
```

```
  port (X: in bit_vector(0 to 3);
        Z: out bit_vector (0 to 6));
end DEC_BCD_7;
```

La descrizione del funzionamento del decoder può essere espressa dal seguente codice:

```
architecture DATAFLOW of DEC_BCD_7 is
begin
  with X select
    --    abcdefg        BCD
    Z <= "1110111" when "0000",  -- 0
         "0010010" when "0001",  -- 1
         "1011001" when "0010",  -- 2
         "1011011" when "0011",  -- 3
         "0111010" when "0100",  -- 4
         "1101011" when "0101",  -- 5
         "1101111" when "0110",  -- 6
         "1010010" when "0111",  -- 7
         "1111111" when "1000",  -- 8
         "1111011" when "1001",  -- 9
         "1101101" when others;  -- Errore
end DATAFLOW;
```

Da notare che è stata inserita la clausola **`when others`** per visualizzare un messaggio di errore in caso si presenti in ingresso una configurazione che non corrisponde ad alcun elemento della codifica BCD.

5.2 Multiplexer

Un multiplexer è un componente dotato di due tipi di linee di ingresso (dati e selezione) e da una linea di uscita. Esso pone sulla linea di uscita il valore di una delle sue linee dati in ingresso. Quest'ultima viene scelta tramite una opportuna combinazione dei valori delle linee di selezione.

Se il multiplexer ha N linee dati, necessita quindi di almeno $\lceil \log_2 N \rceil$ linee di selezione.

Il multiplexer più semplice è il multiplexer a due vie, dotato di due linee dati e una linea di selezione (fig. 8).

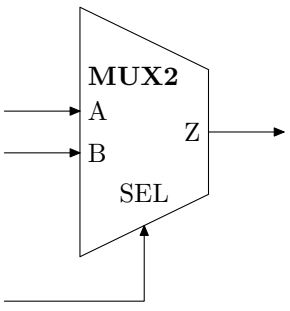


Figura 8: Multiplexer a due vie

La sua tabella di verità è la seguente:

SEL	A	B	Z
0	0	—	0
0	1	—	1
1	—	0	0
1	—	1	1

La descrizione dell'entità VHDL corrispondente è:

```
entity MUX2 is
  port (A, B, SEL: in bit;
                Z: out bit);
end MUX2;
```

La descrizione del comportamento di questo componente può essere espressa tramite il seguente frammento di codice:

```
-- modello comportamentale --
architecture ARC1 of MUX2 is
begin
  P1: process (A, B, SEL)
  begin
    if (SEL = '0') then
      Z <= A;
    else
      Z <= B;
    end if;
  end process P1;
end ARC1;
```

Il corrispondente modello dataflow è il seguente:

```
-- modello data-flow --
architecture DATA_FLOW1 of MUX2 is
begin
  Z <= A when (SEL = '0') else
       B ;
end DATA_FLOW1;
```

Tramite le note tecniche di sintesi di circuiti combinatori, si può verificare che il multiplexer a due vie è equivalente allo schema rappresentato in fig. 9.

```
-- modello data-flow --
architecture DATA_FLOW2 of MUX2 is
begin
  Z <= (A and not SEL) or (B and SEL);
end DATA_FLOW2;
```

Con le stesse considerazioni esposte nel caso del decoder, lo schema circuitale in fig. 9 può essere descritto dalla seguente architettura VHDL:

```
-- modello strutturale --
architecture STRUCT of MUX2 is
  component INV
    port (A: in bit;
```

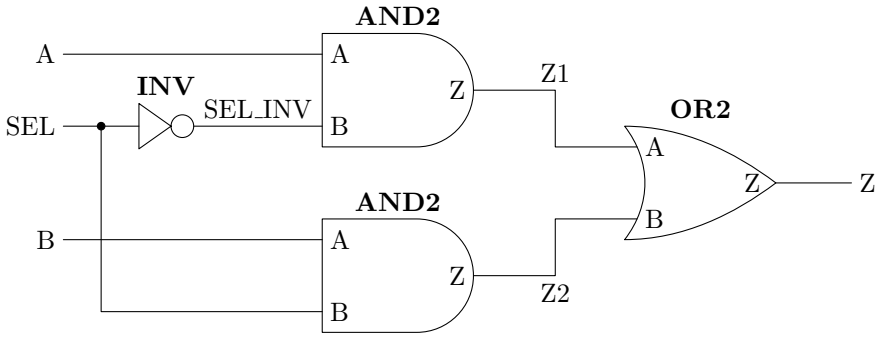


Figura 9: Schema circuitale del multiplexer a due vie.

```
        Z: out bit);
  end component;
  component AND2
    port (A, B: in bit;
          Z: out bit);
  end component;
  component OR2
    port (A, B: in bit;
          Z: out bit);
  end component;

  signal NOT_SEL, Z1, Z2: bit;

begin
  u1: INV port map (SEL, NOT_SEL);
  u2: AND2 port map (A, NOT_SEL, Z1);
  u3: AND2 port map (B, SEL, Z2);
  u4: OR2 port map (Z1, Z2, Z);
end STRUCT;
```

Multiplexer a 4 vie

Il funzionamento del multiplexer a due vie può essere facilmente esteso ad un numero maggiore di linee dati. In fig. 10 è riportato un multiplexer a quattro vie. Il funzionamento di tale componente è riportato nella seguente tabella:

SEL0	SEL1	A	B	C	D	Z
0	0	0	—	—	—	0
0	0	1	—	—	—	1
0	1	—	0	—	—	0
0	1	—	1	—	—	1
1	0	—	—	0	—	0
1	0	—	—	1	—	1
1	1	—	—	—	0	0
1	1	—	—	—	1	1

La descrizione dell'entità VHDL corrispondente è:

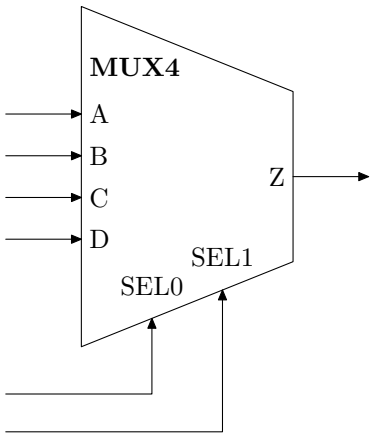


Figura 10: Multiplexer a quattro vie

```
entity MUX4 is
  port (A, B, C, D, SEL0, SEL1: in bit;
                            Z: out bit);
end MUX4;
```

La descrizione del comportamento di questo componente può essere espressa tramite il seguente frammento di codice VHDL:

```
  -- modello comportamentale --
  architecture ARC1 of MUX4 is
  begin
    P1: process (A, B, C, D, SEL0, SEL1)
    begin
    if (SEL0 = '0' and SEL1 = '0') then
      Z <= A;
    elsif (SEL0 = '0' and SEL1 = '1') then
      Z <= B;
    elsif (SEL0 = '1' and SEL1 = '0') then
      Z <= C;
    elsif (SEL0 = '1' and SEL1 = '1') then
      Z <= D;
    end if;
  end process P1;
end ARC1;
```

Lo stesso comportamento può essere descritto in maniera più appropriata usando il costrutto **case-when** e l'operatore di aggregazione &:

```
-- modello comportamentale --
architecture ARC2 of MUX4 is
  signal t : bit_vector(0 to 1);
begin
  t <= SEL0 & SEL1;
  P2: process (A, B, C, D, SEL0, SEL1)
  begin
    case t is
```

```
      when "00" =>
        Z <= A;
      when "01" =>
        Z <= B;
      when "10" =>
        Z <= C;
      when "11" =>
        Z <= D;
    end case;
  end process P2;
end ARC2;
```

Il seguente codice descrive il modello dataflow di un multiplexer a quattro vie:

```
-- modello data-flow --
architecture DATA_FLOW1 of MUX4 is
begin
  Z <= A when (SEL0 = '0' and SEL1 = '0') else
       B when (SEL0 = '0' and SEL1 = '1') else
       C when (SEL0 = '1' and SEL1 = '0') else
       D;
end DATA_FLOW1;
```

Sfruttando le tecniche di sintesi di circuiti combinatori, si può ottenere il circuito in fig. 11, descritto dal seguente modello dataflow:

```
-- modello data-flow --
architecture DATA_FLOW2 of MUX4 is
begin
  Z <= (A and not SEL0 and not SEL1)or (B and not SEL0 and SEL1)
       or (C and SEL0 and not SEL1) or (D and SEL0 and SEL1) ;
end DATA_FLOW2;
```

Per puro esercizio, lo schema in fig. 11 può essere descritto utilizzando un modello VHDL strutturale:

```
-- modello strutturale --
architecture STRUCT1 of MUX4 is
  component INV
    port (A: in bit;
          Z: out bit);
  end component;
  component AND3
    port (A, B, C: in bit;
          Z: out bit);
  end component;
  component OR4
    port (A, B, C, D: in bit;
          Z: out bit);
  end component;
  signal NOT_SEL0, NOT_SEL1, Z1, Z2, Z3, Z4: bit;
begin
```

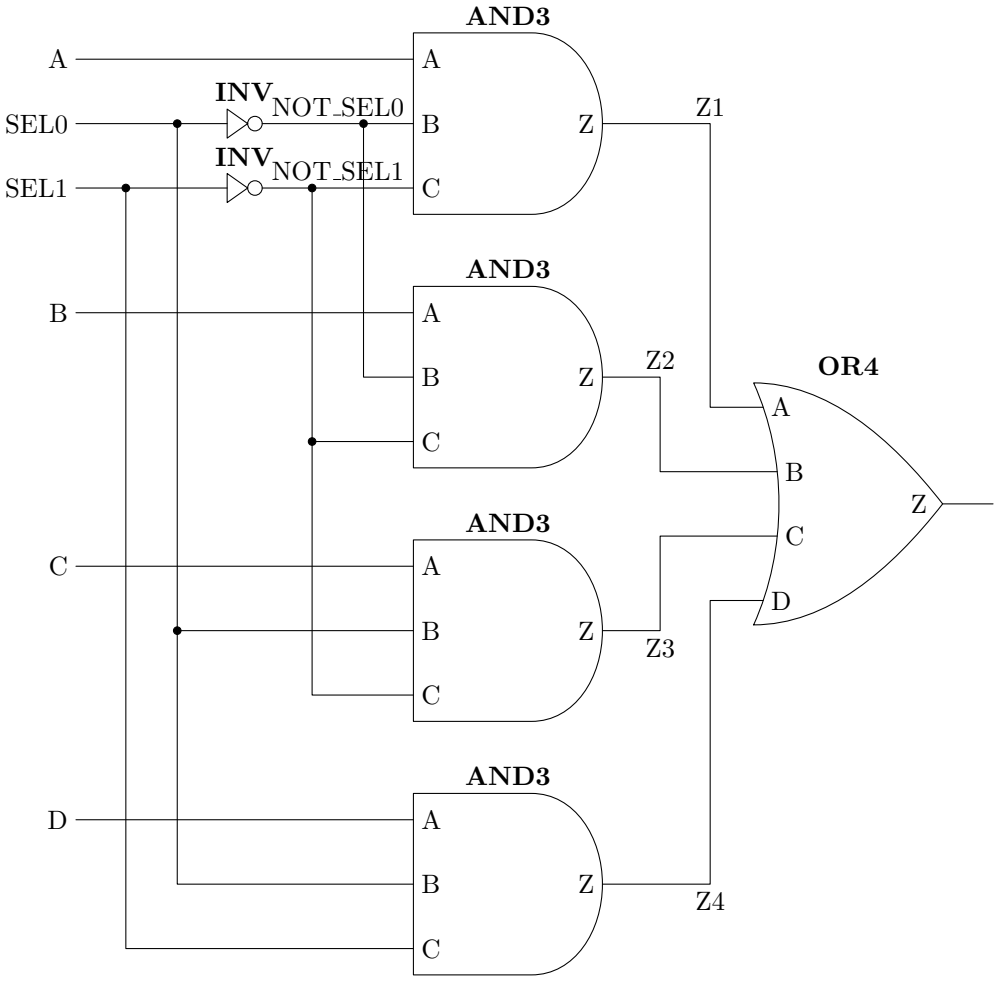


Figura 11: Schema circuitale del multiplexer a quattro vie.

```
  u1: INV port map (SEL0, NOT_SEL0);
  u2: INV port map (SEL1, NOT_SEL1);
  u3: AND3 port map (A, NOT_SEL0, NOT_SEL1, Z1);
  u4: AND3 port map (B, NOT_SEL0, SEL1, Z2);
  u5: AND3 port map (C, SEL0, NOT_SEL1, Z3);
  u6: AND3 port map (D, SEL0, SEL1, Z4);
  u7: OR4 port map (Z1, Z2, Z3, Z4, Z);
end STRUCT1;
```

La fig. 12 illustra come si possano comporre tre multiplexer a due vie per ottenere un multiplexer a quattro vie. Due multiplexer, entrambi pilotati dalla prima linea di selezione, lasciano passare solo due dei segnali dati, i quali vanno in ingresso al terzo multiplexer. Quest'ultimo è pilotato dalla seconda linea di selezione, la quale stabilisce quale dei due segnali dovrà essere portato in uscita.

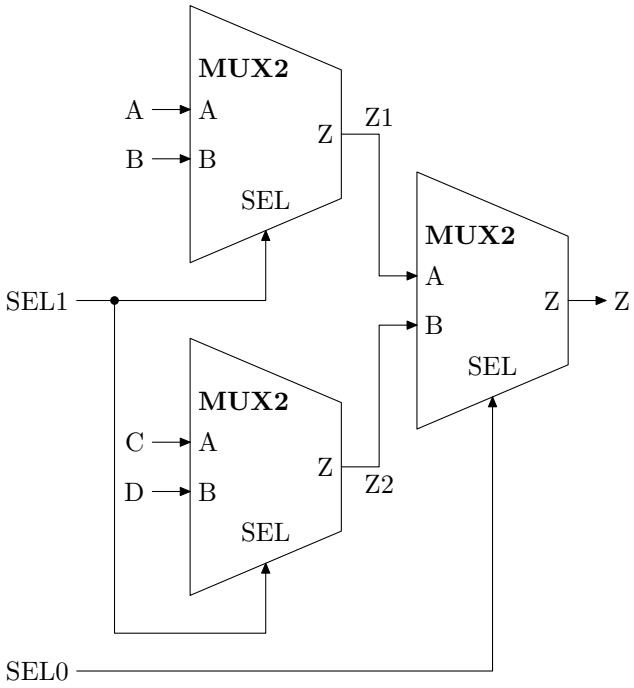


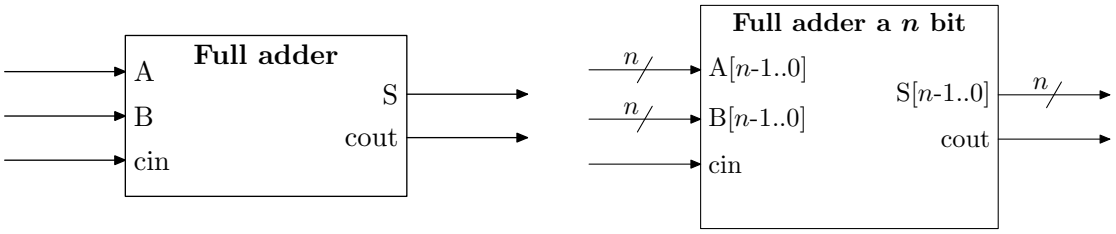
Figura 12: MUX4 costruito assemblando tre MUX2

Il codice VHDL che descrive questo circuito può giovarsi della descrizione strutturale:

```
-- modello strutturale --
architecture STRUCT2 of MUX4 is
  component MUX2
    port (A, B, SEL: in bit;
          Z: out bit);
  end component;
  signal Z1, Z2: bit ;
begin
  u1: MUX2 port map (A, B, SEL1, Z1);
  u2: MUX2 port map (C, D, SEL1, Z2);
  u3: MUX2 port map (Z1, Z2, SEL0, Z);
end STRUCT2;
```

5.3 Full Adder

Il full adder è un componente che realizza la somma di due segnali di ingresso, tenendo in considerazione anche l'eventuale riporto (sia in ingresso che in uscita). Esso è quindi dotato di tre segnali di ingresso (due addendi e il riporto in ingresso) e due segnali di uscita (la somma e il riporto in uscita), come rappresentato in fig. 13a.



(a) (b)

Figura 13: Rappresentazione schematica di un full adder: (a) con addendi a 1 bit; (b) con addendi a n bit.

La descrizione VHDL dell'interfaccia del full adder è la seguente:

```
entity fulladd is
  port (A, B : in bit;       -- addendi
        cin  : in bit;       -- riporto in input
        S    : out bit;      -- somma (output)
        cout : out bit);     -- riporto in output
end fulladd;
```

Il comportamento del full adder può essere descritto tramite la seguente tabella di verità:

A	B	cin	S	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Usando le tecniche di sintesi delle reti logiche, si ottiene la seguente descrizione RTL del componente:

```
architecture rtl of fulladd is
begin
  -- calcolo della somma
  S <= A xor B xor cin;
  -- calcolo del riporto
  cout <= (A and B) or (A and cin) or (B and cin);
end rtl;
```

Alternativamente, usando il costrutto **`case-when`** si può dare la seguente descrizione comportamentale:

```
-- modello comportamentale --
architecture behav of fulladd is
signal t: bit_vector(0 to 2);
begin
  t <= A & B & cin;
  process(t)
  begin
     case t is
      when "000" => (S,cout) <="00";
      when "011" | "101" | "110" => (S,cout)<="01";
      when "001" | "010" | "100" => (S,cout)<="10";
      when "111" => (S,cout)<="11";
     end case;
  end process;
end behav;
```

Full adder a n bit

Il comportamento del full adder può essere generalizzato per consentire l'utilizzo di addendi con un numero di cifre arbitrario, n. Il full adder a n bit è un componente che realizza la somma di due segnali di ingresso da n bit ciascuno, tenendo in considerazione anche l'eventuale riporto (sia in ingresso che in uscita). Esso è dotato in ingresso di due segnali da n bit ciascuno (gli addendi) e da un segnale di un bit per il riporto, e in uscita da un segnale da n bit per la somma e da un segnale di un bit per il riporto (fig. 13b). La descrizione VHDL dell'interfaccia del full adder a 3 bit, per esempio, è la seguente:

```
entity fulladd is
  port (A, B : in bit_vector(2 downto 0);    -- addendi
        cin  : in bit;                       -- riporto in input
        S    : out bit_vector(2 downto 0);   -- somma (output)
        cout : out bit);                     -- riporto in output
end fulladd;
```

Il full adder a n bit può essere realizzato mettendo in cascata n full adder, come descritto in fig. 14.

Il codice VHDL che descrive il circuito in fig. 14b è il seguente:

```
architecture struct of fulladd3 is
  component fulladd
    port (A, B  : in std_logic;
          cin   : in std_logic;
          S     : out std_logic;
          cout  : out std_logic);
  end component;

  signal t   : std_logic_vector(0 to 3);

begin
  t(0) <= cin;
  fa0: fulladd port map (A(0), B(0), t(0), S(0), t(1));
```

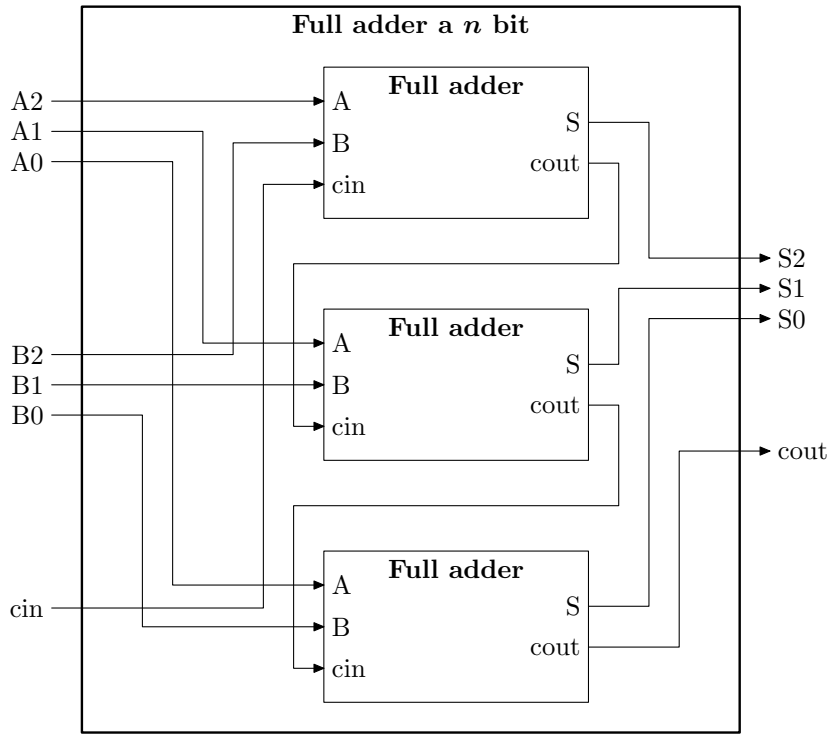


Figura 14: Descrizione strutturale di un full adder a 3 bit composto da 3 full adder.

```
  fa1: fulladd port map (A(1), B(1), t(1), S(1), t(2));
  fa2: fulladd port map (A(2), B(2), t(2), S(2), t(3));
  cout <= t(3);
end struct;
```

Esso può essere realizzato anche mediante l'uso del costrutto **for**, riproducendo esplicitamente il funzionamento di 3 full adder in cascata:

```
architecture behav of fulladd3 is
begin

  process(A,B,cin)
    variable t : bit_vector(0 to 3);
  begin
    t(0) := cin;
    for i in 0 to 2 loop
      S(i) <= A(i) xor B(i) xor t(i);
      t(i+1) := (A(i) and B(i)) or (A(i) and t(i)) or (B(i) and t(i));
    end loop;
    cout <= t(3);
 end process;

end behav;
```

Il comportamento del full adder a n bit può essere efficacemente descritto sfruttando la definizione dell'operatore `+` del tipo `std_logic_vector` (dove viene descritto un full adder a 8 bit):

```
library ieee;
```

```
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

entity fulladd8 is
  port (A, B : in std_logic_vector(7 downto 0);  -- input
        cin  : in std_logic;                     -- riporto in input
        S    : out std_logic_vector(7 downto 0); -- somma (output)
        cout : out std_logic);                   -- riporto in output
end fulladd8;

architecture behav of fulladd8 is
  signal t: std_logic_vector(8 downto 0);
begin

  process(A,B,cin)
  begin
    t <= ('0'&A)+('0'&B)+("00000000"&cin);
  end process;

  cout<=t(8);
  S <= t(7 downto 0);

end behav;
```

Nel precedente esempio, viene usato un segnale ausiliario `t` che dispone di una cifra in più rispetto agli addendi. Essa serve per tener traccia dell'eventuale riporto. Agli addendi, `A` e `B`, e al riporto in ingresso, `cin`, viene anteposta un sequenza di `'0'` in modo da renderli della stessa lunghezza del segnale ausiliario. Dopo di ciò, i tre segnali vengono sommati tra loro. Se si verifica un overflow (riporto in uscita pari a 1), la cifra più significativa del segnale ausiliario, `t`, varrà `'1'`, altrimenti varrà `'0'`. Tale valore verrà quindi assegnato al segnale di riporto in uscita, `cout`. Il valore del segnale somma, `S`, verrà a trovarsi nelle rimanenti cifre del segnale ausiliario.

Per l'esempio sopra riportato, l'uso del tipo `bit_vector` avrebbe richiesto la definizione del comportamento dell'operatore `+`. Ciò non è necessario per il tipo `std_logic`, in quanto tale definizione è già presente nel package `ieee.std_logic_1164`.

6. Componenti sequenziali

6.1 Registri

I registri sono una famiglia di componenti utilizzati per la memorizzazione. Il loro funzionamento dipende quindi, oltre che dai segnali di ingresso, anche dalla loro storia passata. Questa proprietà è caratteristica dei circuiti sequenziali.

L'elemento di memoria minimale è quello che consente la memorizzazione di un bit. Due sono le famiglie principali che realizzano questo comportamento: i *flip-flop* ed i *latch*. Essi saranno utilizzati per costruire elementi di memoria più capaci e con funzionalità più complesse.

Flip-flop di tipo D

Il flip-flop di tipo D è caratterizzato da due linee di ingresso (nella fig. 15 indicati come d e clk) e da una linea di uscita (q). Il segnale di ingresso clk viene detto *clock* e governa la modifica dello stato del segnale di uscita: l'uscita q assume il valore dell'ingresso d solo in presenza di un fronte di salita del clock clk (cioè in corrispondenza di un cambiamento del segnale clk da 0 a 1). Ciò significa che se il clock non cambia di valore (e precisamente passando da 0 a 1), il segnale di ingresso d non avrà alcun effetto sull'uscita q. Per questo motivo questo flip-flop viene anche chiamato *edge triggered*.

Va notato che la forma del segnale di clock non viene in alcun modo specificata. Dal punto di vista puramente teorico, il clock potrebbe assumere qualsiasi forma d'onda. Tuttavia, tipicamente il clock è un segnale periodico che assume in un semiperiodo il valore 0 e nell'altro semiperiodo il valore 1. Esso serve a sincronizzare i componenti di uno stesso circuito ("detta i tempi"). Un comportamento come quello dell'aggiornamento dell'uscita del flip-flop viene detta *sincrono*, in quanto avviene in sincronia con il segnale di clock.

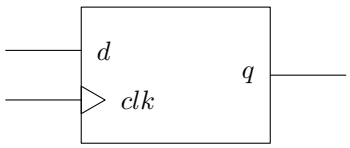


Figura 15: Rappresentazione schematica di un flip-flop.

La descrizione dell'entity VHDL corrispondente al componente in fig. 15 è:

```
entity FF_D is
  port (d, clk: in bit;
              q: out bit);
end FF_D;
```

Ci sono due modi per implementare in VHDL il comportamento di un flip-flop D:

```
architecture behav1 of FF_D is
begin
  process (clk)
  begin
    if (clk'event and clk='1') then
      q <= d;
    end if;
  end process;
end behav1;
```

```
architecture behav2 of FF_D is
begin
  process -- no sensitivity list
  begin
    wait until clk'event and clk='1';
    q <= d;
  end process;
end behav2;
```

Entrambe le **architecture** descrivono il flip-flop in stile comportamentale e utilizzano il costrutto **process** per gestire la dipendenza da segnale di clock. L'**architecture** behav1 fa

uso della sensitivity list del processo per esplicitare la sincronizzazione del segnale di uscita, `q`, dal segnale di clock, `clk`, mentre L'**`architecture`** `behav2` non riposta alcun segnale in sensitivity list, ma lascia che il processo rimanga in attesa di un fronte di salita sul segnale `clk`.

L'espressione usata per descrivere il fronte di salita del clock, `clk'event` **`and`** `clk='1'`, è composta da due sottoespressioni. `clk'event` indica l'attributo `event` del segnale `clk`. Esso assume il valore booleano *vero* solo nell'istante in cui il segnale cambia di valore. L'espressione `clk'event` **`and`** `clk='1'` va quindi letta come: il valore del segnale `clk` è appena cambiato e il suo valore è 1. Tale situazione si verifica solo in presenza di un fronte di salita del segnale.

Registri a n bit

Componendo n flip-flop, si possono costruire registri a n bit: basta mettere i flip-flop in parallelo sullo stesso clock, come rappresentato in fig. 16.

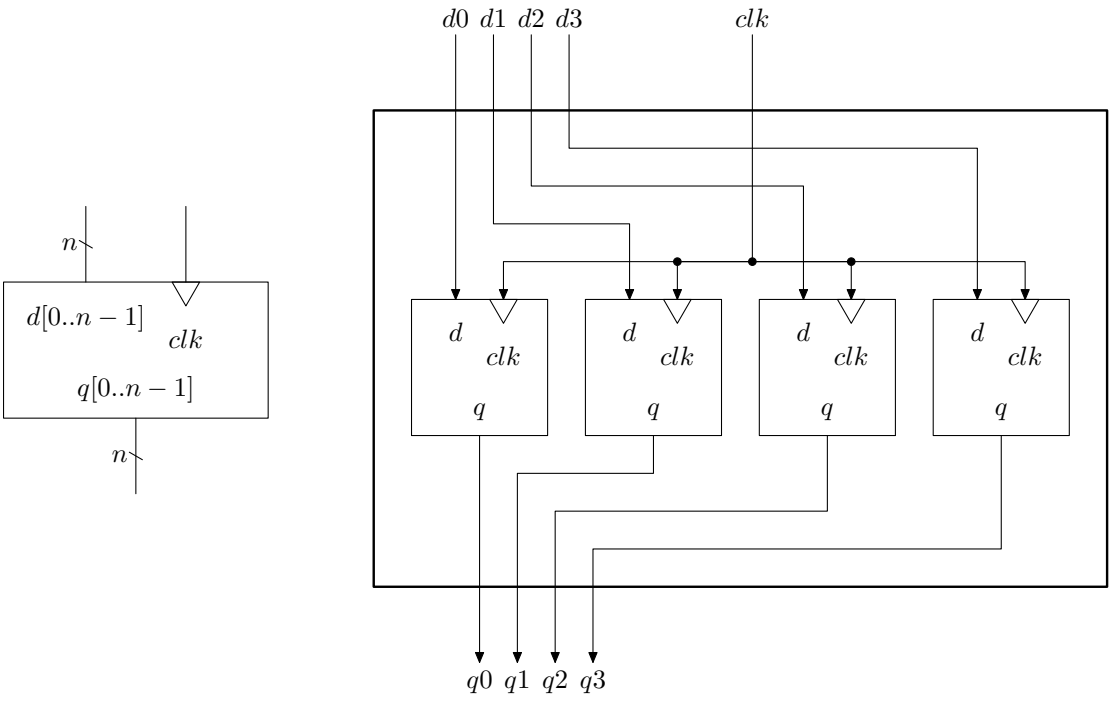


Figura 16: (a) Un registro a n bit. (b) Registro a 4 bit costruito utilizzando 4 flip-flop di tipo D.

Il circuito in fig. 16b può essere descritto in VHDL strutturale come segue:

```
entity REG4 is
  port (  d: in bit_vector(0 to 3);
          clk: in bit;
          q: out bit_vector(0 to 3));
end REG4;

architecture STRUCT of REG4 is
  component FF_D
    port (d, clk: in bit;
```

```
          q: out bit);
  end component;
begin
  u1: FF_D port map (d(0),clk,q(0));
  u2: FF_D port map (d(1),clk,q(1));
  u3: FF_D port map (d(2),clk,q(2));
  u4: FF_D port map (d(3),clk,q(3));
end STRUCT;
```

Flip-flop di tipo SR

Un altro tipo di flip-flop è il *flip-flop di tipo SR* (fig. 17), che deve il suo nome alla presenza di due segnali di ingresso, s (set) e r (reset), la cui funzione è quella di impostare il segnale di uscita a 1 o a 0, rispettivamente.

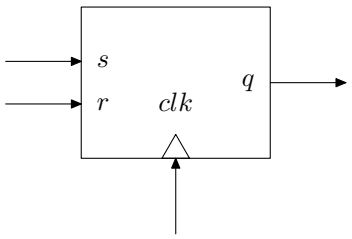


Figura 17: Rappresentazione schematica del flip-flop di tipo SR.

La descrizione dell'entity VHDL che descrive il componente in fig. 17 è la seguente:

```
entity FF_set_reset is
  port (s, r, clk: in std_logic;
                q: out std_logic);
end FF_set_reset;
```

In un flip-flop di tipo SR le operazioni di set e reset sono sincrone. La seguente tabella delle verità descrive questo comportamento:

s	r	clk	q
0	0	↑	q
0	1	↑	0
1	0	↑	1
1	1	↑	**X**
-	-	0	q
-	-	1	q

Il simbolo ↑ indica il fronte di salita del segnale, mentre il simbolo "**X**" significa che il segnale non è definito (ciò accade quando i segnali di set e reset vengono attivati contemporaneamente).

Il comportamento descritto nella tabella precedente viene descritto dal seguente VHDL comportamentale:

```
architecture BEHAV of FF_set_reset is
begin
  process (clk)
  begin
```

```
    if (clk'event and clk='1') then
      if (s='1' and r='1') then
        q <= 'X';
      elsif (r='1') then
        q <= '0';
      elsif (s='1') then
        q <= '1';
      end if;
    end if;
  end process;
end BEHAV;
```

L'utilizzo del tipo `std_logic` anziché `bit` consente di specificare il comportamento indefinito in occasione della contemporaneità dell'attivazione del segnale di set e di quello di reset (nonché del fronte di salita).

Latch

La differenza tra un FF e un latch è che l'uscita del latch riproduce il valore del segnale d'ingresso per tutto il tempo in cui il segnale di clock assume il valore alto. Il segnale di clock assume pertanto il significato di un segnale abilitatore o inibitore e sarà per questo indicato con il nome di *enable* nel seguito.

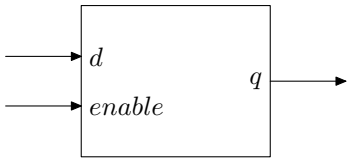


Figura 18: Latch di tipo D.

Il latch riportato in fig. 18 può essere descritto dalla seguente entity VHDL:

```
entity latch_D is
  port ( d, enable: in std_logic;
                 q: out std_logic);
end latch_D;
```

Il latch viene detto *trasparente* quando il segnale *enable* è alto e viene detto *opaco* quando il segnale *enable* è basso. Questo comportamento è descritto dal seguente codice VHDL:

```
architecture behaviour of latch_D is
begin
  process (enable)
  begin
    if (enable = '1') then
      q <= d;
    end if;
  end process;
end behaviour;
```

SR latch

Analogamente alla famiglia dei flip-flop, esistono diverse varianti per i latch. Fra queste, il latch di tipo SR, il quale viene controllato tramite un segnale di set, s, e uno di reset, r, (fig. 19). Data l'assenza di una segnale di clock, si tratta quindi di una rete asincrona.

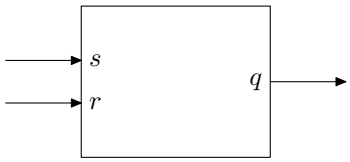


Figura 19: Latch di tipo SR

Il funzionamento è il seguente:

- quando s vale 1, q viene posto ad 1 (a tale valore rimane anche quando s torna a 0);
- quando r vale 1, q viene posto ad 0 (a tale valore rimane anche quando r torna a 0);
- quando sia s sia r valgono 0, il valore di q non cambia;
- quando sia s sia r valgono 1, il valore di q è indefinito.

Ciò viene descritto dalla seguente tabella di verità:

s	r	q
0	0	q
0	1	0
1	0	1
1	1	**X**

Il latch di tipo SR può essere descritto mediante il seguente codice VHDL:

```
entity latch_SR is
  port ( s, r: in std_logic;
            q: out std_logic);
end latch_SR;

architecture behaviour of latch_SR is
begin
  process (s, r)
  begin
    if (s = '1' and r = '0') then
      q <= '1';
    elsif (s = '0' and r = '1') then
      q <= '0';
    elsif (s = '1' and r = '1') then
      q <= 'X';
    end if;
  end process;
end behaviour;
```

Registro con reset asincrono

È possibile realizzare varianti dei componenti fin qui presentati limitando la sincronizzazione solo ad alcune operazioni. Per esempio, un flip-flop con reset asincrono è descritto dal seguente codice VHDL:

```
-- in questo caso il reset e' prioritario!
architecture behaviour of FF_asincr_reset is
begin
  process (clk, r)
  begin
    if (r='1') then
      q <= '0';
    elsif (clk'event and clk='1') then
      if (s='1') then
        q <= '1';
      end if;
    end if;
  end process;
end behaviour;
```

dove l'entity `FF_asincr_reset` è identica all'entity `FF_set_reset`

Da notare che il segnale `r` è stato aggiunto nella sensitivity list e che la sequenza dei costrutti **`if`** rispecchia la priorità data agli eventi.

6.2 Contatore

I registri sono elementi essenziali per costruire i contatori. Questi ultimi formano una famiglia di componenti sequenziali molto variegata. Nella versione più semplice (fig. 20), il contatore ha la caratteristica di incrementare periodicamente l'uscita di 1. L'uscita è costituita da una sequenza di bit e può pertanto essere utilizzata per denotare un numero in notazione binaria. Gli incrementi avvengono solo ad ogni fronte di clock.

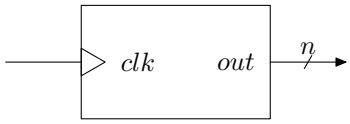


Figura 20: Contatore a n bit.

Ad esempio, nel caso di un contatore a 4 bit di uscita, ipotizzando che il segnale di *out* valga inizialmente "0000", avremmo questo andamento:

# fronti	*out*
0	"0000"
1	"0001"
2	"0010"
3	"0011"

e così via. Un contatore come quello visto nell'esempio, cioè con 4 bit di uscita, viene chiamato contatore modulo 16. Tale componente "conta" fino al numero binario 1111 (15 in base 10) e poi riparte da 0000 (0 in base 10). Un contatore modulo n ha perciò n stati.

Normalmente il contatore ha alcune ulteriori funzionalità, date dalla presenza di segnali addizionali. In fig. 21 ne sono riportati alcuni:

reset riporta l'uscita al valore minimo indicabile, e cioè 0 (che in base binaria si rappresenta con una stringa di n zeri);

set riporta l'uscita al valore massimo indicabile, e cioè $2^n - 1$ (che in base binaria si rappresenta con una stringa di n uno);

up/down indica al contatore in quale direzione contare: se ascendente oppure discendente;

enable indica al counter se contare oppure no nel corrente ciclo di clock;

data in/load se il segnale *load* è attivato, porta in uscita la sequenza di bit immessa dal segnale *data in*. v

Le operazioni di set, reset e caricamento possono essere (in modo indipendente l'una dall'altra) in sincronia con il clock oppure in modo asicrono. Questo fatto dà luogo a molteplici varianti del componente contatore.

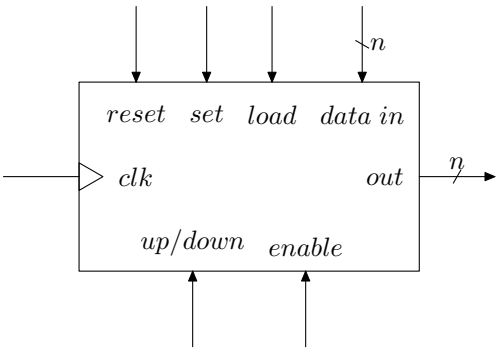


Figura 21: Contatore a n bit con segnali addizionali

Il seguente codice VHDL descrive un contatore a 8 bit con reset asincrono.

```
entity cont8 is
  port(clk, reset: in std_logic;
       outa: out std_logic_vector(0 to 7));
end cont8;

architecture rtl of cont8 is
  signal t: std_logic_vector(0 to 7);
begin

  process(clk, reset)
  begin
    if (reset = '1' ) then
      t <= "00000000";
    elsif (clk'event and clk = '1' ) then
        t <= t + "00000001";
    end if;
  end process;
```

```
  outa <= t;

end rtl;
```

Il segnale `t` viene utilizzato per gestire l'incremento. Esso appare sia a destra che a sinistra dell'operatore di assegnamento. Al suo posto non può perciò essere usato un segnale in modalità **`in`** perché non potrebbe essere a sinistra, né un segnale in modalità **`out`** perché non potrebbe essere a destra dell'operatore di assegnamento. Il segnale `t` è di tipo `std_logic` per poter effettuare le operazioni di somma su sequenze di bit. Da notare, infine, che l'assegnamento `outa <= t` viene effettuato all'esterno del processo. Questo perché i valori dei segnali modificati all'interno di un processo vengono aggiornati solo al termine del processo stesso.

La versione sincrona dell'operazione di reset può essere descritta in VHDL portando il test sul segnale `reset` all'interno del codice eseguito dopo il test sul fronte di salita del segnale `clk`. Inoltre, il segnale `reset` può essere tolto dalla sensitivity list.

Di seguito è riportato il codice VHDL di un contatore che include le modifiche sopra descritte, con l'aggiunta del segnale `updown` per governare la direzione del conteggio.

```
entity counter is
  port(clk, reset, updown: in std_logic;
              outa: out std_logic_vector(0 to 7));
end counter;

architecture sincr_reset of counter is
  signal t: std_logic_vector(0 to 7);
begin
  process(clk)
  begin
    if (clk'event and clk = '1') then
      if (reset = '1') then
        t <= "00000000";
      elsif (updown = '1') then
        t <= t + "00000001";
      else
        t <= t - "00000001";
      end if;
    end if;
  end process;

  q <= t;
end sincr_reset;
```

Contatore completo

Il contatore può essere descritto come composizione di componenti fin qui visti. La fig. 22 illustra questo concetto. Alla fine della catena di componenti, si trova un registro. La sua funzione è mantenere in memoria il valore in del segnale in uscita, *out* e di aggiornarlo ad ogni fronte di salita del segnale di clock, *clk*. Inoltre, vengono fornite le funzionalità di *set* e *reset* tramite gli omonimi segnali. Il segnale di ingresso del registro viene fornito da un multiplexer che, regolato dal segnale *load*, seleziona il valore dell'uscita del registro al battito di clock precedente opportunamente

incrementato o il valore del segnale *data in*. L'incremento del valore di *out* viene effettuato da un addizionatore, che, oltre al segnale di *out*, riceve in ingresso un segnale con il valore da impostato dai segnali *enable* e *up/down* tramite due multiplexer in cascata. Il segnale *enable* posto a zero, ha l'effetto di porre il segnale di incremento a zero, in modo da inibire la modifica del valore dell'uscita. Il segnale *up/down* alto pone il valore di incremento pari a "+1", mentre lo pone a "-1" se basso. Da notare che i valori "0", "+1" e "-1" devono essere rappresentati come una opportuna sequenza di n bit.

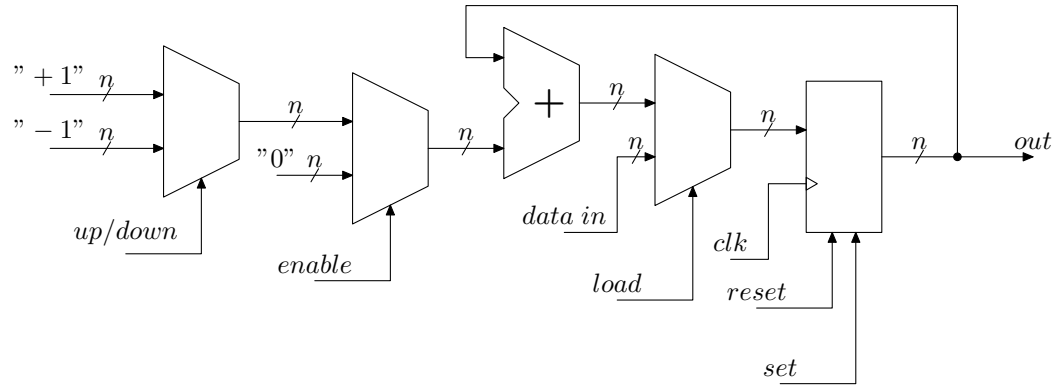


Figura 22: Visione sistemistica di un contatore.

Più in dettaglio, le funzionalità di *reset* e *set* possono essere realizzate ponendo il segnale di ingresso del registro ripettivamente in *and* con il segnale di *reset* invertito e in *or* con il segnale di *set*.

Questa descrizione del contatore può essere riportata quasi pedissequamente in VHDL, usando lo stile strutturale:

```
entity cont_full_sist is
  port(clk, set, reset, load, enable, updown: in std_logic;
       datain: in std_logic_vector(0 to 7);
       outa: out std_logic_vector(0 to 7));
end cont_full_sist;

architecture struct of cont_full_sist is

  component REG8                  -- registro da 8 bit
    port (
      clk, set, reset : in std_logic;
      d               : in std_logic_vector (0 to 7);
      q               : out std_logic_vector (0 to 7));
  end component;

  component MUX2_8                -- multiplexer a 2 ingressi da 8 bit
    port (
      a, b : in  std_logic_vector (0 to 7);
      sel  : in std_logic;
      q    : out std_logic_vector (0 to 7));
      -- q vale a se sel vale '1', b altrimenti
  end component;
```

```
  component ADDER_8                -- sommatore da 8 bit
    port (
      a, b : in std_logic_vector (0 to 7);
      r    : out std_logic_vector (0 to 7));
  end component;

  signal t1,t2,t3,t4,t5 : std_logic_vector (0 to 7);
  -- i seguenti segnali svolgono la funzione di costanti
  signal zero, piu_uno, meno_uno: std_logic_vector (0 to 7);

begin
-- outa non puo' essere usato in modalita' "in" come richiesto dalla
-- port map dell'adder: viene usato t3 al suo posto

  registro: REG8 port map (clk,set,reset,t1,t3);
  load_mux: MUX2_8 port map (datain,t2,load,t1);
  adder: ADDER_8 port map (t3,t4,t2);
  enable_mux: MUX2_8 port map (t5,zero,enable,t4);
  updown_mux: MUX2_8 port map (piu_uno,meno_uno,updown,t5);
  outa <= t3;
  -- NB: esiste un modo piu' corretto di indicare le costanti in VHDL,
  -- ma non e' stato visto a lezione
  zero <= "00000000";
  piu_uno <= "00000001";
  meno_uno <= "11111111";    -- "-1" in complemento a 2
end struct;
```

Da notare che i segnali interni `t1`–`t5` vengono usati per connettere i componenti, mentre i segnali `zero`, `piu_uno`, `meno_uno` sono usati impropriamente per definire dei segnali costanti (esistono costrutti più appropriati per descrivere valori costanti, ma non rientrano nel sottoinsieme di VHDL visto a lezione).

6.3 Registro a scorrimento

Il registro a scorrimento (*shift register*) è costituito da una catena di flip-flop in cascata. Nella sua versione più semplice (fig. 23), il registro a scorrimento ha il seguente funzionamento: ad ogni colpo di clock, ogni flip-flop passa il proprio valore al successivo ed il primo flip-flop della catena assume il valore del segnale di ingresso d. Il valore precedentemente memorizzato nell'ultimo flip-flop viene assegnato al segnale di uscita del registro a scorrimento, q.

Figura 23: Registro a scorrimento

Il componente in fig. 23 può essere descritto in VHDL dalla seguente entità:

```
entity ShiftRegister is
  port (
```

```
    d, clk : in std_logic;
    q : out std_logic);
end ShiftRegister;
```

Da notare che la lunghezza della catena di flip-flop che compongono il registro a scorrimento non compare in alcun modo nella dichiarazione di entity (e nemmeno nello schema in fig. 23). Il numero di flip-flop impiegati costituisce la capacità del registro a scorrimento e, secondo la descrizione sopra data, rappresenta il numero di colpi di clock che bisogna attendere perchè un segnale di ingresso venga presentato in uscita.

Il funzionamento di un registro a scorrimento a 4 bit può essere descritto come segue:

```
architecture behav of ShiftRegister is
  signal t : std_logic_vector(0 to 3);
begin
  process (clk)
  begin
    if clk'event and clk = '1' then -- fronte di salita del clock
      t <= d & t(0 to 2);
      q <= t(3);
    end if;
  end process;
end behav;
```

I registri (interni) vengono rappresentati tramite un vettore, `t`, di opportuna lunghezza. Da notare come anche dopo l'assegnamento `t <= d & t(0 to 2)`, il valore di `t(3)` sia lo stesso che era all'ingresso del processo: l'aggiornamento dei valori dei segnali modificati in un **`process`**, infatti, avviene solo al termine del **`process`** stesso.

Il modello semplice del registro a scorrimento fin qui illustrato può essere arricchito di funzionalità. Ciò richiede, analogamente al caso del contatore, l'aggiunta di alcuni segnali addizionali:

reset imposta i valori dei flip-flop vengono impostati a `'0'`;

set imposta i valori dei flip-flop vengono impostati a `'1'`;

right/left indica in che direzione effettuate lo scorrimento;

enable abilita oppure disabilita lo scorrimento nel corrente ciclo di clock;

data in/load imposta nel registro la sequenza di bit immessa dal segnale *data in* una volta attivato il segnale di *load*;

data out porta all'esterno i valori di uscita di tutti i flip-flop, consentendo una lettura parallela del contenuto dello shift register.

Con l'aggiunta dei segnali *data in/load* e *data out*, il registro a scorrimento può essere utilizzato per la conversione seriale/parallela dell'elaborazione.

Come per il contatore, le operazioni sullo shift register possono essere sincrone o asincrone, a seconda della modalità con cui possono essere effettuate: se vengono permesse solo in coincidenza di un fronte di salita, vengono dette *sincrone*; se possono essere effettuate in ogni istante, vengono dette *asincrone*.

Il seguente codice VHDL descrive uno shift register a 4 bit con reset asincrono e load sincrono:

```
entity shift_reg is
  port (
    d, clk, load, reset : in  bit;
    datain              : in  bit_vector (0 to 3);
    q                   : out bit);
end shift_reg;

architecture reset_asincr of shift_reg is
  signal t : std_logic_vector(0 to 3);
begin
  process (clk,reset)
  begin
    if reset = '1' then
      t <= "0000";
    elsif clk'event and clk = '1' then -- fronte di salita del clock
      if load = '1' then
        t <= datain;
      else
        t <= d & t(0 to 2);
        q <= t(3);
      end if;
    end if;
  end process;
end reset_asincr;
```

L'operazione di scorrimento dei valori dei flip-flop può essere descritta anche tramite un costrutto di ciclo **for-loop**:

```
architecture reset_asincr of shift_reg is
  signal t : std_logic_vector(0 to 3);
begin
  process (clk,reset)
  begin
    if reset = '1' then
      t <= "0000";
    elsif clk'event and clk = '1' then -- fronte di salita del clock
      if load = '1' then
        t <= datain;
      else
        for i in 0 to 2 loop
          t(i+1) <= t(i);
        end loop;
        t(0) <= d;
        q <= t(3);
      end if;
    end if;
  end process;
end reset_asincr;
```

Il funzionamento del registro a scorrimento può essere descritto in termini di composizione di componenti standard elementari. Per esempio, in fig. 24 è illustrato uno schema a blocchi che

descrive un registro a scorrimento a 4 bit con segnali di set e reset utilizzando quattro flip-flop di tipo D (dotati anch'essi di segnali *set* e *reset*). Esso può essere descritto in VHDL strutturale come segue:

```
entity sh_reg_sr is
  port (
    d, clk, set, reset : in  bit;
    q                  : out bit);
end sh_reg_sr;

architecture struct of sh_reg_sr is
  component ffd_sr
    port (
      d, clk, set, reset : in  bit;
      q                  : out bit);
  end component;

  signal t1, t2, t3 : bit;

begin  -- struct
  reg0 : ffd_sr port map (d, clk, set, reset, t1);
  reg1 : ffd_sr port map (t1, clk, set, reset, t2);
  reg2 : ffd_sr port map (t2, clk, set, reset, t3);
  reg3 : ffd_sr port map (t3, clk, set, reset, q);
end struct;
```

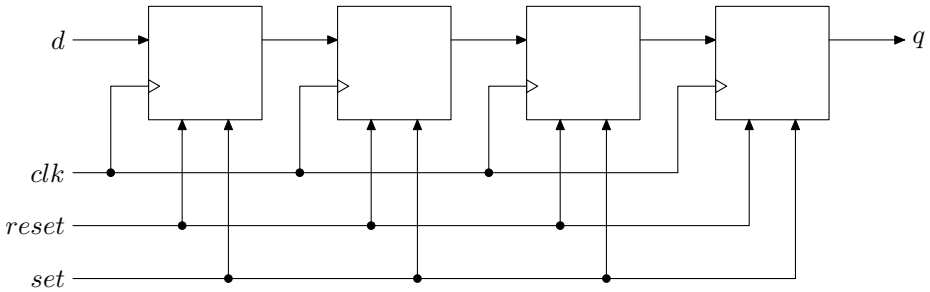


Figura 24: Schema a blocchi di un registro a scorrimento a 4 bit con segnali di set e reset.

Da notare che la modalità con cui operano i segnali `set` e `reset` dipendono dall'implementazione del componente `ffd_sr`: ad esempio, se `ffd_sr` ha il reset asincrono, anche `sh_reg_sr` lo avrà.

La fig. 25 illustra, mediante uno schema a blocchi, una realizzazione di un registro a scorrimento a 4 bit con caricamento di dati in parallelo. Questa azione è controllata dal segnale `load`, il quale, agendo su una serie di multplexer, seleziona l'ingresso per i flip-flop: se `load` vale `'0'`, il registro a scorrimento ha il funzionamento caratteristico, mentre se `load` vale `'1'`, agli ingressi dei flip-flop vengono presentati i valori portati dal bus `datain`.

Questo circuito può essere descritto in VHDL strutturale come segue:

```
entity sh_reg_load is
  port (
    d, clk, load : in  bit;
    datain : in  bit_vector (0 to 3);
```

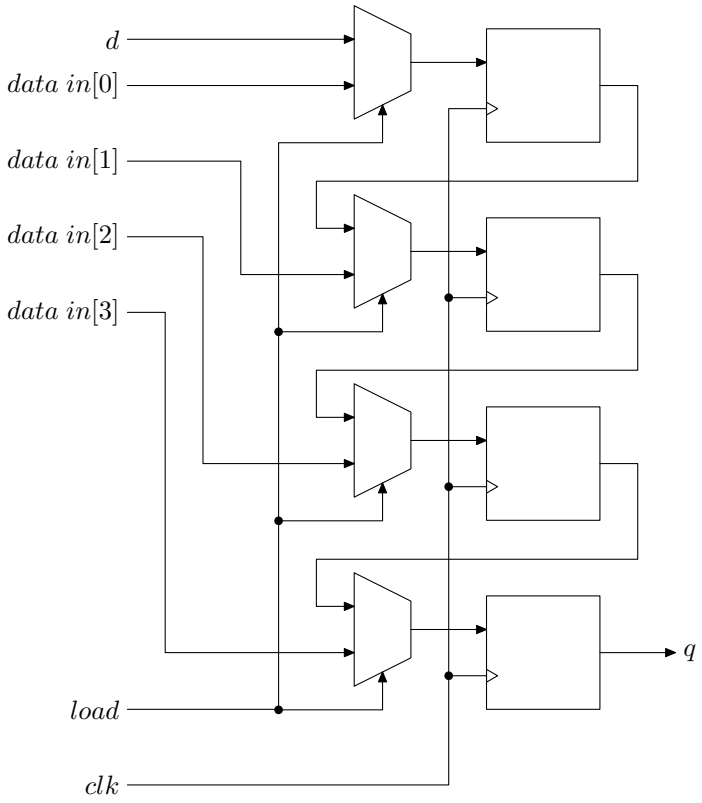


Figura 25: Schema a blocchi di un registro a scorrimento a 4 bit con caricamento in parallelo.

```
    q                   : out bit);
end sh_reg_load;

architecture struct of sh_reg_load is
  component ffd
    port (
      d, clk : in bit;
      q                 : out bit);
  end component;

  component mux
    port (
      a, b, sel: in bit;
      z: out bit);
  end component;

  signal t1, t2, t3, t4, t5, t6, t7: bit;

begin  -- struct
  reg0 : ffd port map (t4,clk,t1);
  reg1 : ffd port map (t5,clk,t2);
  reg2 : ffd port map (t6,clk,t3);
  reg3 : ffd port map (t7,clk,q);
  mux0 : mux port map (d,datain(0),load,t4);
```

```
  mux1 : mux port map (t1,datain(1),load,t5);
  mux2 : mux port map (t2,datain(2),load,t6);
  mux3 : mux port map (t3,datain(3),load,t7);
end struct;
```

7. Verifica

Ogni progetto prevede, come ultima fase, la verifica dell'aderenza dell'opera alle specifiche di progetto. La scrittura di codice, ed in particolare la stesura di una descrizione VHDL, non deroga a questa prassi.

Sebbene vi siano diversi metodi per verificare la correttezza di una descrizione VHDL, il metodo più diffuso è senz'altro la costruzione di un *test bench* (banco di prova). Per verificare la correttezza di un componente elettronico, lo si inserisce in un apposito circuito costruito per verificarne le funzionalità. Tale circuito (un banco di prova, appunto) stimola gli ingressi del circuito sotto test con appropriati segnali elettrici, analizzando i segnali elettrici che vengono prodotti alle uscite del componente sotto test. Confrontando i segnali di ingresso (sui quali si ha un controllo molto preciso) con i segnali di uscita, si può verificare se il componente risponde correttamente (cioè se il suo comportamento soddisfa i requisiti di progettazione).

Analogamente, il VHDL può essere utilizzato per costruire una descrizione del circuito di prova. La verifica del funzionamento del componente sotto test richiederà la disponibilità di un opportuno programma in grado di simulare il comportamento del circuito descritto dal codice VHDL (un simulatore di VHDL comportamentale). Un tool di questo tipo è generalmente incluso in ogni suite di programmi per la progettazione in VHDL.

Il VHDL dispone di diversi costrutti per la descrizione dei ritardi nella propagazione dei segnali all'interno di un circuito. Queste descrizioni vengono utilizzate per descrivere realisticamente il comportamento di una realizzazione del circuito. Tali ritardi nella propagazione del segnale sono dipendenti dalla tecnolgia utilizzata per la realizzazione del circuito ed ha quindi senso introdurli in fase di progettazione avanzata oppure per provare gli effetti di eventuali alee.

Oltre a verificare la correttezza della descrizione di un componente, il test bench è utile anche come strumento di documentazione: esso fornisce la descrizione di uno o più casi d'uso del componente progettato.

7.1 Entity di testbench

L'entità VHDL che descrive il banco di prova deve includere l'entità sotto test come componente e deve descrivere gli opportuni segnali in grado di fornire gli stimoli per la verifica del circuito sotto test.

In fig. 26 viene illustrato questo schema. L'unità sotto test, UUT, ha due porte di input, a e b, ed una porta di uscita, c. L'entità di testbench, indicata in figura come Testbench, include UUT ed è fornita dei segnali a_s, b_s e c_s, collegati rispettivamente alle porte a, b e c. Al fine di ottenere una verifica estensiva, i segnali dell'unità di testbench devono assumere valori tali da testare tutte le possibili combinazioni dei segnali di ingresso dell'unità sotto test. La comparazione dei segnali delle porte di uscita dell'unità sotto test con i valori attesi costituisce la verifica della correttezza dell'unità sotto test. Poiché talvolta le combinazioni dei valori dei segnali di ingresso sono troppo numerose per poter essere valutate, generalmente il test viene limitato alle configurazioni più significative.

I segnali di ingresso possono mutare nel tempo. Ciò consente di coprire più combinazioni di

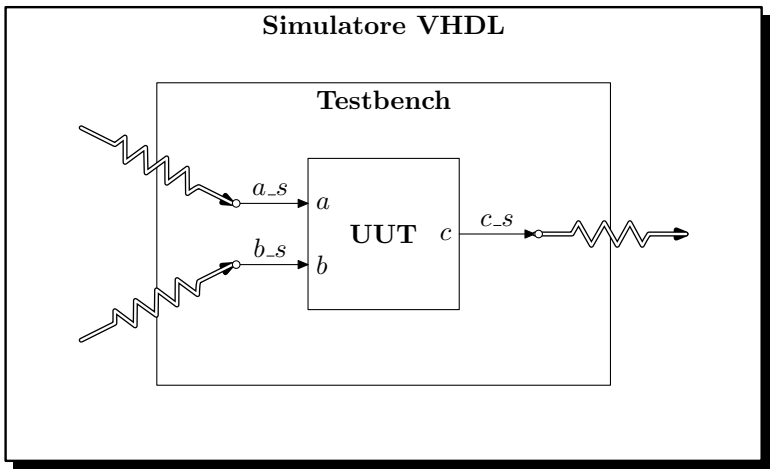


Figura 26: Verifica di componenti VHDL tramite testbench. L'entità UUT viene incluso nell'entity Testbench come componente. Un opportuno programma (simulatore VHDL) simula il funzionamento del circuito così descritto, permettendo di analizzare i segnali generati.

ingresso usando la stessa entità di testbench e anche di valutare il funzionamento di circuiti sequenziali (il cui comportamento dipende, oltre che dai valori di ingresso, anche dallo stato del componente).

I segnali possono essere descritti utilizzando istruzioni concorrenti (assegnamento con clausola **after**) o sequenziali (costrutti **wait** e **wait for** in istruzioni **process**).

L'entità di testbench non necessita di porte di input e di output con l'esterno. Per esempio, la descrizione in VHDL del circuito di prova della fig. 26 è il seguente:

```
entity Testbench is
end Testbench;

architecture behav of Testbench is
component UUT
  port (
     a, b: in  bit;
     c   : out bit);
end component;

  signal a_s, b_s, c_s: bit;

begin

  test_unit: UUT port map (a_s, b_s, c_s);

  a_p: process -- andamento del segnale a_s
    begin
      a_s <= '0';
      wait for 10 ns;
      a_s <= '1';
      wait for 10 ns;
    end process;
```

```
  b_p: process   -- andamento del segnale b_s
    begin
      b_s <= '0';
      wait for 20 ns;
      b_s <= '1';
      wait for 20 ns;
    end process;

end behav;
```

L'andamento nel tempo dei segnali `a_s` e `b_s` sono riportati in fig. 27. Si può notare che essi assumono tutte le possibili combinazioni dei valori di ingresso.

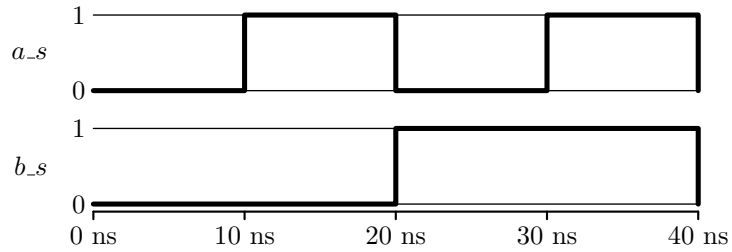


Figura 27: Andamento nel tempo dei segnali descritti nella entità di testbench.

Il simulatore VHDL elabora le descrizioni delle entità Testbench e UUT e fornisce l'evoluzione nel tempo dei segnali delle entità simulate.

8. Testbench di circuiti combinatori

In questo paragrafo verranno illustrati alcuni esempi di unità di testbench per i principali componenti combinatori: il decoder ed il multiplexer.

8.1 Decoder

Un decoder a due ingressi e quattro uscite può essere descritto dal seguente codice:

```
entity DEC24 is
  port (A, B: in bit;
          Z: out bit_vector (0 to 3));
end DEC24;

architecture BEHAV of DEC24 is
begin
  P1: process (A, B)
  begin
    if (A='0' and B='0') then
      Z <= "1000";
    elsif (A='0' and B='1') then
      Z <= "0100";
    elsif (A='1' and B='0') then
```

```
      Z <= "0010";
    elsif (A='1' and B='1') then
      Z <= "0001"; end if;
  end process P1;
end BEHAV;
```

Poiché ha due ingressi (e si tratta di un componente combinatorio), il suo comportamento può essere verificato estensivamente generando un segnale a due bit che assuma tutte e quattro le combinazioni possibili, riportate in fig. 28a.

Essi sono modellati all'interno della seguente unità di testbench:

```
entity  DEC24_TB is
end DEC24_TB;

architecture BEHAV of DEC24_TB is
component DEC24
  port (
     A, B: in  bit;
     Z: out bit_vector (0 to 3));
end component;

  signal A_S, B_S: bit;
  signal Z_S: bit_vector (0 to 3);

begin

  uut:  DEC_24 port map (A_S, B_S, Z_S);

  A_p: process -- andamento del segnale A_S
    begin
      A_S <= '0';
      wait for 10 ns;
      A_S <= '1';
      wait for 10 ns;
    end process;

  B_p: process   -- andamento del segnale B_S
    begin
      B_S <= '0';
      wait for 20 ns;
      B_S <= '1';
      wait for 20 ns;
    end process;

end BEHAV;
```

La simulazione di queste entità in un simulatore di VHDL comportamentale dovrebbe generare i segnali riportati in fig. 28b.

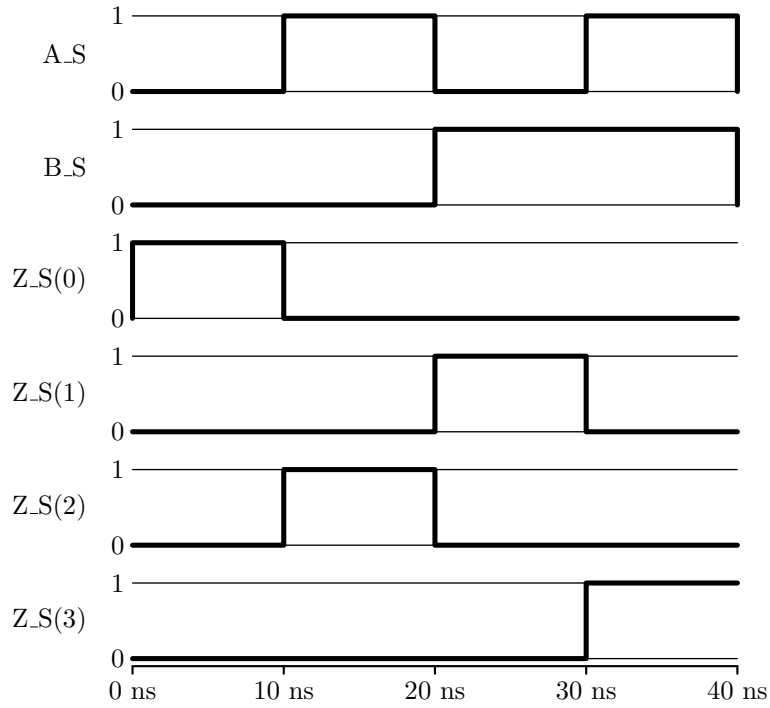


Figura 28: I segnali per il test di un decoder 2-4. I segnali di ingresso coprono tutte le possibili combinazioni.

8.2 Multiplexer

Il multiplexer più semplice, il multiplexer a due vie, può essere descritto in VHDL come:

```
entity MUX2 is port (
   A, B, SEL: in bit;
   Z: out bit);
end MUX2;

architecture DATA_FLOW of MUX2 is
begin
   Z <= (A and not SEL)or (B and SEL);
end DATA_FLOW;
```

Il suo comportamento può essere testato in modo estensivo utilizzando tutte le combinazioni dei segnali di ingresso. Una entità di testbench per il multiplexer sopra descritto è la seguente:

```
entity  MUX2_TB is
end MUX2_TB;

architecture BEHAV of MUX2_TB is
component MUX2
  port (
     A, B, SEL: in  bit;
     Z: out bit);
end component;

  signal A_S, B_S, SEL_S, Z_S: bit;
```

```
begin

  uut:  MUX2 port map (A_S, B_S, SEL_S, Z_S);

  SEL_p: process -- andamento del segnale SEL_S
    begin
      SEL_S <= '0';
      wait for 10 ns;
      SEL_S <= '1';
      wait for 10 ns;
    end process;

  A_p: process -- andamento del segnale A_S
    begin
      A_S <= '0';
      wait for 20 ns;
      A_S <= '1';
      wait for 20 ns;
    end process;

  B_p: process   -- andamento del segnale B_S
    begin
      B_S <= '0';
      wait for 40 ns;
      B_S <= '1';
      wait for 40 ns;
    end process;

end BEHAV;
```

In fig. 29 sono riportati i segnali risultanti da una simulazione dell'entità `MUX2_TB` sopra riportata.

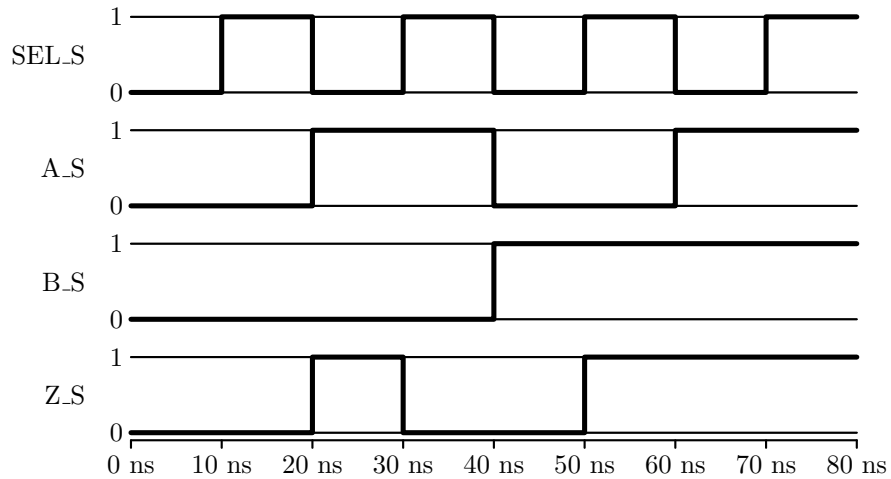


Figura 29: I segnali per il test di un multiplexer a due vie. I segnali di ingresso coprono tutte le possibili combinazioni.

9. Testbench di circuiti sequenziali

I circuiti sequenziali hanno un comportamento che dipende, oltre che dallo valore dei segnali di ingresso, anche dallo stato del componente stesso, il quale, a sua volta, dipende dalla sequenza passata dei segnali di ingresso. Per effettuare un test esaustivo, quindi, non sarebbe sufficiente presentare ai morsetti di ingresso tutte le combinazioni possibili dei valori dei corrispondenti segnali, ma bisognerebbe provare tutte le possibili sequenze di ingressi. Questo modo di procedere, in generale, non è fattibile. Tuttavia, spesso è sufficiente verificare che il componente si comporti bene in alcuni casi particolari, i quali, sebbene non coprano la totalità dei casi possibili, assicurano la copertura delle configurazioni più significative. In questo paragrafo verranno illustrati alcuni esempi di testbench per circuiti sequenziali: il latch di tipo SR, il flip-flop di tipo D e il contatore a 3 bit.

9.1 Latch SR

Il latch di tipo SR può essere descritto dal seguente codice VHDL:

```
library ieee;
USE ieee.std_logic_1164.all;

entity latch_SR is
  port ( s, r: in std_logic;
            q: out std_logic);
end latch_SR;

architecture behaviour of latch_SR is
begin
  process (s, r)
  begin
    if (s = '1' and r = '0') then
      q <= '1';
    elsif (s = '0' and r = '1') then
      q <= '0';
    elsif (s = '1' and r = '1') then
      q <= 'X';
    end if;
  end process;
end behaviour;
```

Per verificare il comportamento di un latch SR è importante controllare che:

(a) quando solo il segnale di set, `s`, assume il valore alto, anche l'uscita, `q`, assuma il valore alto;

(b) quando solo il segnale di reset, `s`, assume il valore alto, l'uscita, `q`, assuma il valore basso;

(c) quando entrambi i segnali `s` e `r` assumono valore basso, l'uscita non cambi valore.

L'unità di testbench descritta dal seguente codice VHDL permette di verificare le precedenti condizioni in alcuni casi salienti:

```
library ieee;
USE ieee.std_logic_1164.all;

entity latch_SR_tb is
end latch_SR_tb;

architecture testbench of latch_SR_tb is
  component latch_SR is
    port ( s, r: in std_logic;
              q: out std_logic);
  end component;

  signal s_s, r_s, q_s : std_logic;

begin
  uut: latch_SR port map (s_s, r_s, q_s);

  s_p: process -- andamento del segnale s_s
  begin
    s_s <= '0';
    wait for 10 ns;
    s_s <= '1';
    wait for 10 ns;
    s_s <= '0';
    wait for 50 ns;
    s_s <= '1';
    wait for 10 ns;
    s_s <= '0';
    wait for 10 ns;
  end process;

  r_p: process -- andamento del segnale r_s
  begin
    r_s <= '0';
    wait for 30 ns;
    r_s <= '1';
    wait for 10 ns;
    r_s <= '0';
    wait for 10 ns;
    r_s <= '1';
    wait for 10 ns;
    r_s <= '0';
    wait for 30 ns;
  end process;

end testbench;
```

La simulazione di questo testbench dovrebbe generare i segnali riportati in fig. 30. Essi permettono di dire che le condizioni (a)–(c) sopra elencate sono rispettate. La condizione (a) viene testata agli istanti α e η. La condizione (b) viene testata agli istanti γ e ϵ. Infine, le configurazioni dei

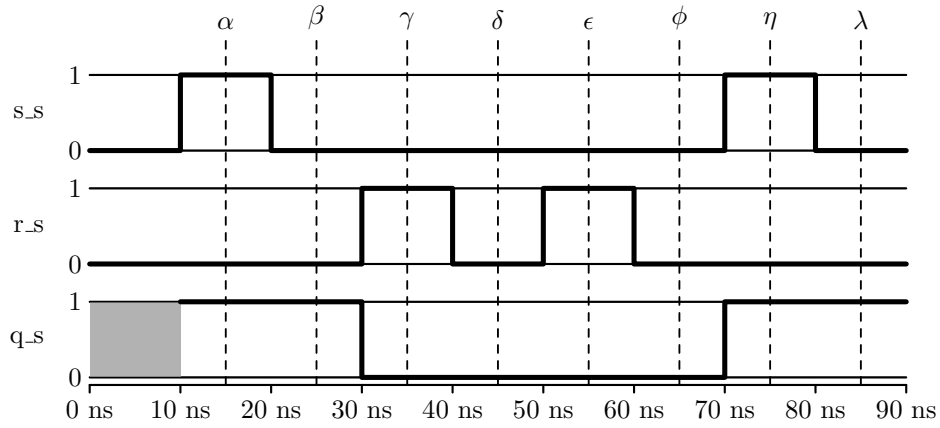


Figura 30: I segnali per il test di un latch SR. I segnali di ingresso coprono le configurazioni necessarie per verificare le condizioni di funzionamento più significative.

segnali `s` e `r` agli istanti β, δ, ϕ e λ mettono alla prova la condizione (c). Da notare che, fino al primo verificarsi di un set o reset, l'uscita è indefinita. Questa situazione si verifica nei primi 10 ns della simulazione qui considerata, ed è stata resa nel grafico rappresentando il segnale `q` in tale intervallo di tempo mediante un'area grigia.

9.2 Flip-flop di tipo D

Il flip-flop di tipo D può essere descritto dal seguente codice VHDL:

```
library ieee;
USE ieee.std_logic_1164.all;

entity ff_D is
  port (d, clk: in std_logic;
              q: out std_logic);
end ff_D;

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

Per verificare il comportamento di un flip-flop di questo tipo è importante controllare che:

(a) in presenza di un fronte di salita del clock, `clk`, l'uscita, `q`, assuma il valore del segnale di ingresso, `d`;

(b) in assenza di un fronte di salita di `clk`, l'uscita mantenga lo stesso valore, indipendentemente da eventuali variazioni del segnale di ingresso.

L'unità di testbench descritta dal seguente codice VHDL permette di verificare le precedenti condizioni in alcuni casi salienti:

```
library ieee;
USE ieee.std_logic_1164.all;

entity ff_D_tb is
end ff_D_tb;

architecture testbench of ff_D_tb is

  component ff_D is
  port (d, clk: in std_logic;
            q: out std_logic);
  end component;

  signal d_s, clk_s, q_s : std_logic;


begin

  uut: ff_D port map (d_s, clk_s, q_s);

  clk_p: process -- andamento del segnale clk_s
  begin
    clk_s <= '0';
    wait for 10 ns;
    clk_s <= '1';
    wait for 10 ns;
  end process;

  d_p: process -- andamento del segnale d_s
  begin
    d_s <= '1';
    wait for 25 ns;
    d_s <= '0';
    wait for 20 ns;
    d_s <= '1';
    wait for 20 ns;
  end process;

end testbench;
```

La simulazione di questo testbench dovrebbe generare i segnali riportati in fig. 31. Essi permettono di dire che le condizioni (a) e (b) sopra elencate sono rispettate. L'analisi dei valori in uscita agli istanti α, β e γ conferma la condizione (b), in quanto nell'intervallo di tempo coperto da questi istanti non vi sono fronti di salita e, nonostante sia il clock che il segnale di ingresso varino il loro valore, il segnale di uscita rimane costante. Il confronto tra lo stato dei segnali agli istanti γ e δ (e anche agli istanti δ e ϵ) verifica la condizione (a). Da notare che, fino al primo verificarsi di un fronte di salita, l'uscita è indefinita. Questa situazione si verifica nei primi 10 ns

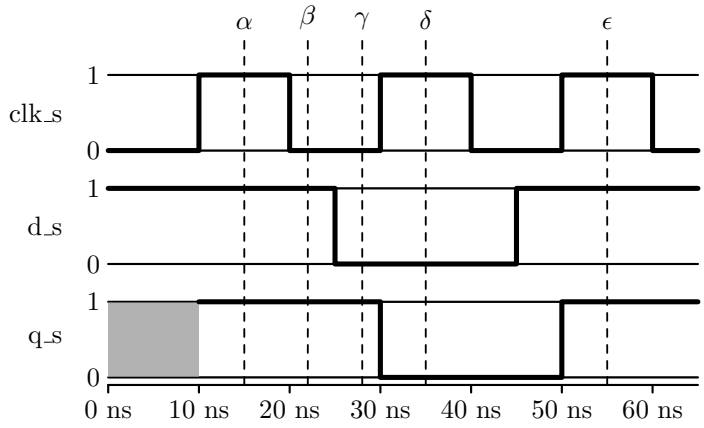


Figura 31: I segnali per il test di un flip-flop di tipo D. I segnali di ingresso coprono le configurazioni necessarie per verificare le condizioni di funzionamento più significative.

della simulazione qui considerata, ed è stata resa nel grafico rappresentando il segnale `q` in tale intervallo di tempo mediante un'area grigia.

9.3 Contatore

Il contatore a 3 bit (con reset asincrono) può essere descritto dal seguente codice VHDL:

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

entity cont3 is
port(clk, r: in std_logic;
          q: out std_logic_vector(0 to 2));
end cont3;

architecture behav of cont3 is
  signal t: std_logic_vector(0 to 2);
begin
  process(clk, r)
  begin
    if (r = '1' ) then
      t <= "000";
    elsif(clk'event and clk = '1' ) then
      t <= t + "001";
    end if;
  end process;

  q <= t;
end behav;
```

Per verificare il comportamento di un contatore di questo tipo è importante controllare che:

(a) l'incremento unitario del segnale di uscita, `q`, avvenga solo in presenza di un fronte di salita del clock, `clk`;

(b) dopo 8 (2^n per un contatore a n bit) incrementi, l'uscita torni a 0;

(c) finché il segnale di reset, `r` è alto, il segnale di uscita venga azzerato (indipendentemente dal clock).

L'unità di testbench descritta dal seguente codice VHDL permette di verificare le precedenti condizioni in alcuni casi salienti:

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

entity cont3_tb is
end cont3_tb;

architecture testbench of cont3_tb is
  component cont3
    port(clk, r: in std_logic;
                 q: out std_logic_vector(0 to 2));
  end component;

  signal clk_s,r_s: std_logic;
  signal q_s: std_logic_vector(0 to 2);

begin
  uut: cont3 port map (clk_s,r_s,q_s);

  clk_p: process
  begin
    clk_s <= '0';
    wait for 10 ns;
    clk_s <= '1';
    wait for 10 ns;
  end process;

  r_p: process
  begin
    r_s <= '1';
    wait for 5 ns;
    r_s <= '0';
    wait for 202 ns;
    r_s <= '1';
    wait for 6 ns;
    r_s <= '0';
    wait for 7 ns;
  end process;

end testbench;
```

La simulazione di questo testbench dovrebbe generare i segnali riportati in fig. 32. Essi permettono di dire che le condizioni (a)–(c) sopra elencate sono rispettate. Il segnale in uscita, `q`,

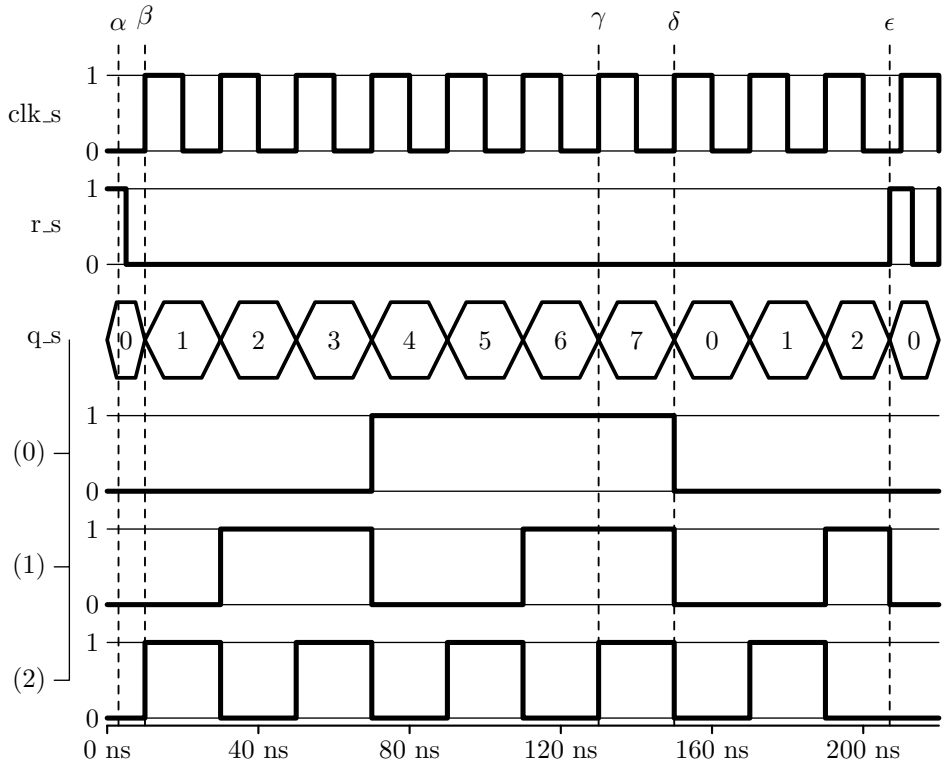


Figura 32: I segnali per il test di un contatore a 3 bit. I segnali di ingresso coprono le configurazioni necessarie per verificare le condizioni di funzionamento più significative. Il segnale q_s è un segnale di tipo `std_logic_vector(0 to 2)` e viene rappresentato sia riportando il valore decimale (positivo) che esso rappresenta, sia mediante le sue componenti binarie.

viene incrementato solo in corrispondenza dei fronti di salita del clock, `clk`, come mostrato tra gli istanti β e γ, dove il segnale di uscita passa da 0 a 7, confermando la condizione (a). All'ottavo fronte di salita del clock (istante δ) l'uscita torna al valore 0, confermando la condizione (b). Infine, agli istanti α e ϵ, dove il segnale di reset, `r`, assume il valore alto, il valore dell'uscita viene posto a 0, confermando così la condizione (c).