Introduzione al linguaggio VHDL

Aspetti teorici ed esempi di progettazione

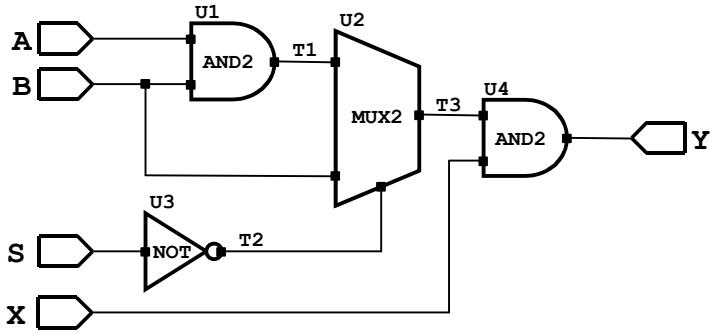
Carlo Brandolese
Politecnico di Milano

1. Introduzione

Il linguaggio VHDL (VLSI Hardware Description Language) è un linguaggio per la **descrizione** dell'hardware. Questa prima definizione sottolinea un aspetto molto importante: il VHDL non è un linguaggio eseguibile ovvero non descrive quali **operazioni** un esecutore deve svolgere per ricavare il risultato di una elaborazione, bensì **descrive gli elementi** che costituiscono il circuito digitale in grado di effettuare l'elaborazioe richiesta.
Una specifica VHDL non è quindi eseguibile e deve essere pensata come qualche cosa di completamente diverso da un programma o un algoritmo. Tuttavia, una specifica VHDL può essere **simulata** mediante opportuni strumenti. Simulare una specifica VHDL significa simulare il comportamento del circuito che la specifica descrive, quindi, ancora una volta, non si tratta di nulla di simile alla esecuzione di un programma.
Il linguaggio VHDL è estremamente ricco e flessibile e permette di fornire specifche di circuiti digitali a diversi **livelli di astrazione**. Nei paragrafi seguenti vedremo brevemente quali sono i livelli di astrazione del VHDL e le caratteristiche di ognuno di essi, in relazione alla struttura circuitale che essi sono in grado di rappresentare.

1.1 Il livello strutturale o gate-level

Al livello più basso di astrazione possiamo vedere un circuito come un grafo in cui i nodi rappresentano elementi logici (semplici o complessi) quali porte logiche, multiplexer, flip-flop oppure interi sottocircuiti, mentre gli archi rappresentano le connessioni tra tali elementi. Un esempio di una tale visone è riportato nella figura seguente.



A questo livello, il VHDL descrive quindi esplicitamente quali sono gli elementi che costituiscono un circuito e come questi sono connessi tra di loro mentre l'informazione relativa alle trasformazioni funzionali che subiscono i dati è implicita. In particolare, una tale rappresentazione descrive esplicitamente le seguenti informazioni:

1. I nomi ed il tipo degli ingresis e delle uscite primarie del circuito (**`A`**, **`B`**, **`S`**, **`X`**, **`Y`**)
2. Il tipo degli elementi logici (**`AND2`**, **`NOT`**, **`MUX2`**)
3. Il nome delle istanze degli elementi logici (**`U1`**, **`U2`**, **`U3`**, **`U4`**)
4. Il nome dei segnali interni (**`T1`**, **`T2`**, **`T3`**)
5. Le connessioni tra i segnali e le porte dei componenti

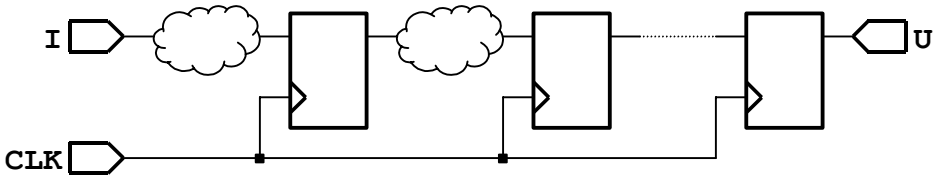
Tale rappresentazione è comunemente detta **netlist** ed è il modello che più si avvicina alla realizzazione finale del circuito in esame.

1.2 Il livello RTL o data-flow

Aumentando il livello di astrazione, si passa dal VHDL gate-level al VHDL RTL (Register Transfer Level) o data-flow. Secondo questo paradigma, la specifica descrive esplicitamente le trasformazioni che i dati subiscono durante la propagazione all'interno del circuito. In particolare, il circuito è visto come un insieme di due tipologie di elementi:

1. Reti combinatorie. Esprimono in forma esplicita le trasformazioni dei dati mediante espressioni algebriche, espressioni aritmetiche e condizioni.
2. Registri. Sono deputati a memorizzare i risultati intermedi di elaborazioni complesse.

Questa suddivisione spiega il nome Register Transfer: la specifica infatti esprime come avviene il trasferimento e l'elaborazione dei dati tra i registri della rete. In termini strutturali, si può vedere una specifica RTL come una sequenza di elementi di logica combinatoria interrotta dai registri.



In questo schema **`I`** rappresenta un generico insieme di ingressi, **`U`** un generico insime di uscite, le nuvolette rappresentano generiche reti puramente combinatorie con complessità arbitraria ed i flip-flop sono generici elementi di memoria che indicano le posizioni dei registri. Da questo scema appare evidente come, a livello RTL, una specifica VHDL esprima le trasformazioni dei segnali ed i punti di memorizzazione di tali segnali. Si noti che una tale organizzazione non vale solo per un circuito nel suo insieme ma anche per ogni sua sottoparte, qualunque sia la sua complessità o dimensione. Per sottocircuiti semplici è possibile che non vi sia la necessità di memorizzare alcun dato ovvero è possibile che il sottocircuito non contenga alcun registro. In questo caso la definizione di Register Transfer Level può rimanere invariaa a patto di considerare anche gli ingressi e le uscite primarie di un circuito o di un sottocircuito alla stregua di registri.
Infine è opportuno sottolineare una ulteriore caratteristica delle specifiche a livello RTL: ogni operazione (nel senso di elaborazione di dati) è assegnata esplicitamente ad un ben preciso stadio dell'elaborazione ovvero ad uno specifico ciclo di clock. L'operazione di assegnamento delle operazioni ai vari cicli di clock prende il nome di **scheduling** ed in questo caso è il progettista a farsene completamente carico.

1.3 Il livello algoritmico o behavioural

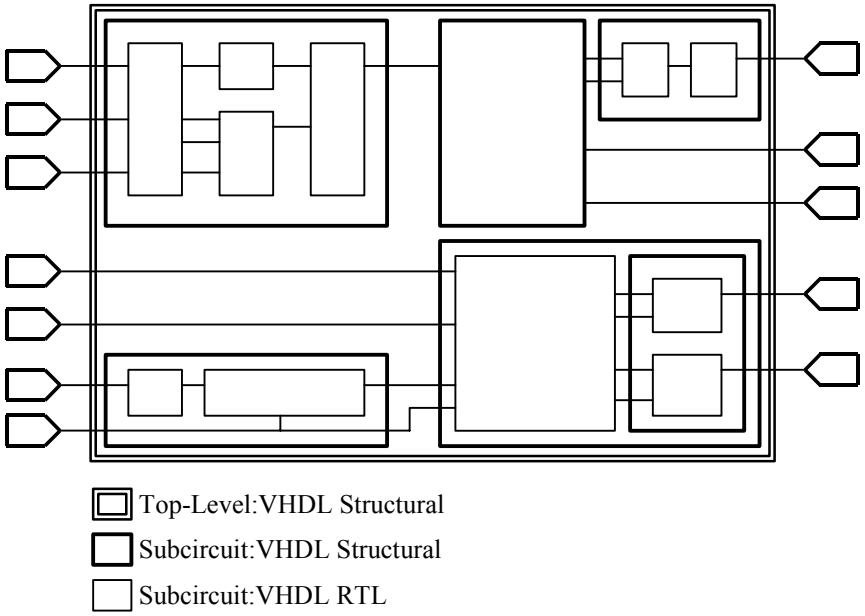
Il livello behavioural è il massimo livello di astrazione che il VHDL consente. Esso è in grado di descrivere la funzionalità di un circuito mediante uno o più algoritmi. In questo caso, né la struttura, né le singole trasformazioni che i dati subiscono sono esplicite. In particolare non è esplicito come le varie operazioni ed elaborazioni sono assegnate ai diversi cicli di clock. Non è quindi evidente la struttura mostrata per le descrizioni a livello RTL: sarà lo strumento di sintesi ad effettuare lo scheduling delle operazioni sui vari cicli di clock sulla base di vincoli imposti dal progettista quali ad esempio la frequenza di clock minima oppure l'area massima. Queste dispense non trattano le specifiche a livello behavioural.

1.4 Specifiche miste

Una specifica mista altro non è che una specifica in cui alcune porzioni sono descritte a livello strutturale mentre altre porzioni sono espresse a livello RTL. Benché non considerato in queste dispense, anche una specifica a livello behavioural può essere combinata con gli altri livelli di astrazione.
Una situazione tipica, derivata dalla pratica di progettazione nelle realtà industriali, suggerisce un uso ben preciso dei due stili di specifica all'interno di un circuito complesso. Per prima cosa notiamo che quando un circuito supera una certa complessità esso non è descritto come una entità unica bensì come una struttura gerarchica, composta da sottocircuiti. Ogni sottocircuito svolge una ben determinata funzione e dovrebbe essere quanto più possibile isolato ed indipendente dal resto del sistema. Sottocircuiti semplici sono poi opportunamente connessi a formare sottocircuiti di complessità superiore, fino ad arrivare alla funzionalità completa del circuito. È abbastanza chiaro che ci si trova di fronte alla necessità di specificare due tipologie ben distinte di sottocircuiti o moduli:

1. Moduli di elaborazione vera e propria, all'interno dei quali i dati subiscono trasformazioni esplicitamente descritte dal codice VHDL.
2. Moduli di integrazione, in cui una funzionalità complessa è costrutita mediante l'interconnessione di moduli di elaborazione.

Possiamo visualizzare graficamente questa situazione come segue.



Questa figura mostra sia la struttura gerarchica del circuito completo, sia i diversi stili adottati per la specifica dei moduli. Il circuito nell'insieme è un modulo, comunemente detto **top-level**, i cui ingressi e uscite sono gli ingressi e le uscite primarie e la sua struttura è data dalla connessione di un insieme di sottocircuiti. Ogni altro modulo è dotato di ingresi ed di uscite che possono essere gli ingressi o le uscite primarie del circuito oppure segnali interni. Lo stile di descrizione di tali sottocircuiti è, come regola generale, a livello strutturale per tutti i moduli tranne le foglie della gerarchia che sono invece descritte in stile register transfer.

2. Design entities

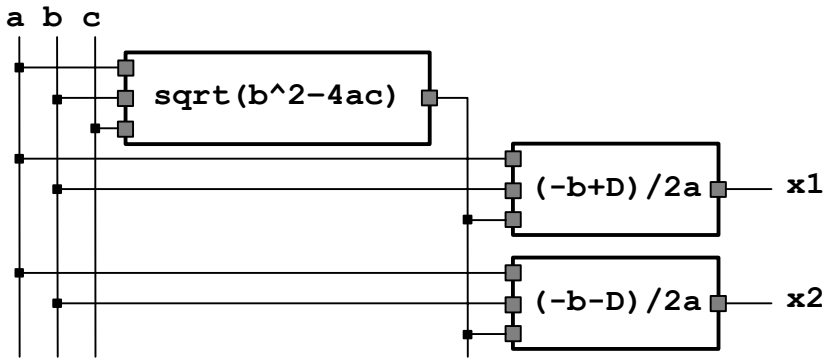
Ogni sistema, dal più semplice al più complesso, si compone di unità funzionali ben identificabili. Tali unità, a volte dette moduli o blocchi, hanno lo scopo di isolare una funzione ben precisa del sistema complessivo con il duplice scopo di fornire una visione strutturata del progetto e di scomporre un problema molto complesso – lo sviluppo di un intero sistema – in un insieme di sottoproblemi di complessità minore.
Si consideri, a titolo di esempio, un sistema per il calcolo delle radici di una equazione di secondo grado a partire dai tre coefficienti **a**, **b** e **c**. Le soluzioni, supponendo che il determinante si maggiore o ugugale a zero, sono le seguenti:

```
x1 = [-b + sqrt(b^2 – 4ac)] / 2a
x2 = [-b - sqrt(b^2 – 4ac)] / 2a
```

Una prima possibile scomposizione del problema in sottoproblemi più semplici potrebbe essere:

1. Calcolo della radice del detrminante **D=sqrt(b^2 – 4ac)**
2. Calcolo di **x1=(-b+D)/2a**
3. Calcolo di **x2=(-b-D)/2a**

Graficamente, si potrebbe rappresentare tale scomposizione nel modo seguente:

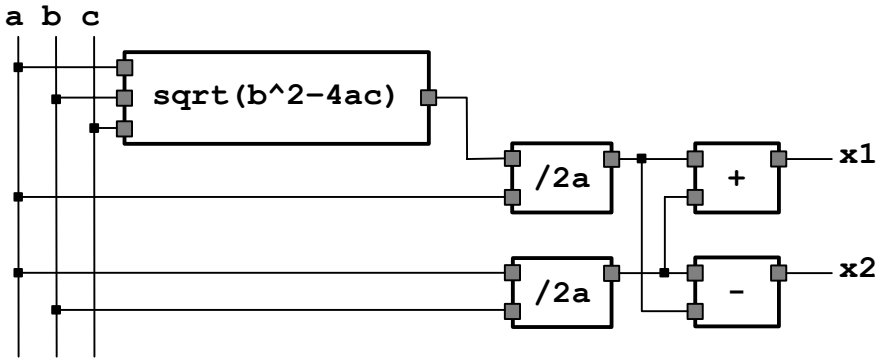


Questa decomposizione si basa su tre moduli di cui i due per il calcolo delle soluzioni a partire da **D**, **a** e **b**. I tre moduli sono abbastanza complessi e sufficientemente differenti da poter essere difficilmente condivisi o ottimizzati. Tuttavia tale scomposizione può essere ulteriormente raffinata notando che le soluzioni possono anche ricavate distribuendo il denominatore. In sintesi:

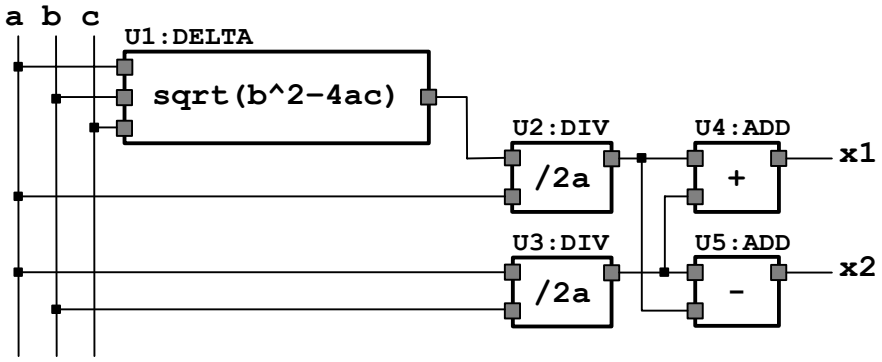
1. Calcolo della radice del detrminante **D=sqrt(b^2 – 4ac)**
2. Calcolo di **t1=-b/2a**
3. Calcolo di **t2=D/2a**
4. Calcolo di **x1=t1-t2**
5. Calcolo di **x2=t1+t2**

Questa nuova decomposizione evidenzia ben 5 blocchi. Inoltre rispetto alla soluzione precedente possiamo notare che l'operazione di divisione per il valore **2a** viene svolta due volte. Ciò significa che il blocco in questione può essere riutilizzato 2 volte (passi 2 e 3). Analogamente anche il modulo per la somma o la differenza di **t1** e **t2** può essere sfruttato sia per il passo 4 sia

per il passo 5. Di fatto, quindi, benché il progetto di componga effettivamente di 5 moduli, solo 3 di essi sono effettivamente differenti. La rappresentazione di tale decomposizione è la seguente:



Possiamo esprimere questa circostanza a proposito della possibilità di riutilizzare alcuni dei moduli del progetto introducendo la differenza tra **modulo** e **istanza**. Un modulo è una entità unica composta da una interfaccia ed un comportamento mentre una istanza rappresenta un utilizzo di tale modulo nella costruzione del circito. Lo schema a blocchi dell'ultima scomposizione mostrata si compone quindi di 3 moduli e di un totale di 5 istanze. Etichettando con un nome ognuna delle istanze possiamo rendere esplicita questa importante distinzione. La figura seguente mostra il diagramma a blocchi opportunamente annotato con nomi in cui la prima parte indica il nome dell'istanza mentre la seconda parte è un riferimento al modulo e ne indica quindi, benchè indirettamente, le caratteristiche.



Dopo avere brevemente considerato il problema della scomposizione di un sistema complesso in sottosistemi semplici, è possibile introdurre il concetto di **design entity**. Una design entity altro non è che un blocco (modulo). Una design entity, ed in particolare la sua specifica mediante il linguaggio VHDL, sottolinea e separa i due concetti complementari di **interfaccia** e **comportamento**. L'interfaccia è quella parte di blocco che specifica i segnali di ingresso e di uscita e che consente di connettere il blocco stesso ad altri blocchi. Il comportamento, invece, descrive come gli ingressi (i seganli di ingresso) devono essere trasformati o elaborati per produrre le uscite (i segnali di uscita). Vedremo nei paragrafi seguenti che l'interfaccia di un modulo prende il nome di **entity** ed è descritta mediante il construtto **`entity`** mentre il comportamento prende il nome di **architecture** ed è descritto dal costrutto **`architecture`**.

2.1 Entity

L'interfaccia di ogni design entity o modulo è descritta da una **entity declaration**. Tale costrutto specifica il **nome** del modulo, le **porte** del modulo ed, eventualmente, un insieme di **parametri**, detti generic, che ne possono modificare le proprietà. La sintassi generale di una entity declaration è la seguente:

```
entity entity_name is
    [generic( generic_list );]
    [port( port_list );]
end entity_name;
```

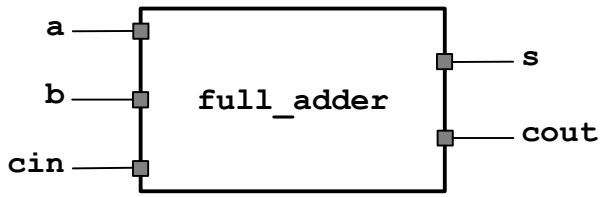
L'***entity_name*** definisce il nome della design entity e deve essere unico per ogni design. La ***port_list*** è una lista di dichiarazioni di porte e descrive i segnali che costituiscono gli ingressi e le uscite della design entity. Ogni dichairazione di porta è strutturata secondo la seguente sintassi:

```
port_name[,port_name,...]: {in|out|inout} port_type;
```

Il nome della porta è indicato da ***port_name***: mediante tale nome sarà possibile riferirsi, nella specifica dell'architettura, al segnale connesso a tale porta. La seguente parola chiave indica la direzione della porta e può essere **in** per le porte di ingresso, **out** per le porte di uscita e **inout** per le porte bidirezionale. È importante sottolineare che una porta con direzione **in** può solo essere letta (ovvero può apparire solo nella parte destra di un assegnamento o in una espressione condizionale) mentre una porta **out** può solo essere scritta (ovvero può apparire solo nella parte sinistra di un assegnamento). Le porte **inout**, invece, possono essere sia lette che scritte. È bene ricordare cha la sintassi del linguaggio prevede che l'ultima dichiarazione di porta che appare tra le parentesi tonde del costrutto **port** sia priva del punto e virgola conclusivo.
In maniera simile a quella delle porte, la definizione della ***generic_list*** segue la sintassi:

```
generic_name[,generic_name,...]: generic_type;
```

A differenza delle porte, i generic non hanno alcuna direzione e possono avere tipi complessi, normalmente non urlizzati per le porte, quali tipi in virgola mobile, stringhe e record.
A titolo di esempio, si consideri la dichiarazione dell'interfaccia di un full adder dotato di tre segnali di ingresso **a**, **b** e **cin** della dimensione di un bit e di due segnali di uscita **s** e **cout**, anch'essi di un bit. Una rappresentazione grafica del blocco in questione, seguendo uno stile e un insieme di convenzioni tipiche, è la seguente.



Si vuole inoltre specificare, ai fini della sola simulazione, un parametro **delay** che indica il ritardo del modulo. Sempre ai fini della simulazione, a tale parametro dovrà essere assegnato un valore: ciò viene fatto al momento della istanziazione del componente, ovvero all'atto dell'uso del componente in un progetto più complesso. La dichiarazione di entity completa è la seguente:

```
entity full_adder is
    generic( delay: real );
    port( a, b, cin: in  bit;
          s, cout:   out bit );
end full_adder;
```

Come nel caso di codice sorgente per la realizzazione di software, è buona norma arricchire la specifica con commenti che ne chiariscano gli aspetti essenziali. In VHDL i commenti sono introdotti dal simbolo `--` (due trattini) e si estendono fino alla fine della riga. Una versione commentata della dichiarazione precedente potrebbe essere la seguente:

```
entity full_adder is

    -- Generics
    generic(
        delay: real       -- The full adder delay in ns
    );

    -- Ports
    port(
       -- Inputs
       a:     in  bit;    -- First operand
       b:     in  bit;    -- Second operand
       cin:   in  bit;    -- Carry-in
       -- Outputs
       s:     out bit;    -- Sum
       cout:  out bit     -- Carry-out
    );

end full_adder;
```

Nei paragrafi seguenti, quando si introdurranno i tipi vettoriali, si presenterà una applicazione dei generic alla specifica di componenti configurabili orientata alla sintesi e non solo alla simulazione. La specifica di un componente parametrico, infatti, offre molti vantaggi sia dal punto di vista della compattezza e della chiarezza, sia dal punto di vista della riusabilità del codice VHDL. Quest'ultimo aspetto è di particolare rilevanza nella progettazione di sistemi molto complessi ed le metodologie di specifica di siffatti componenti sono attualmente argomento di attiva ricerca e studio.

2.2 Architecture

Abbiamo visto che una entity declaration definisce l'interfaccia di un modulo ma non dice né sulla funzionalità svolta dal modulo, né, tantomeno, sul modo in cui il modulo realizza tale funzionalità. La funzionalità di un modulo è descritta in VHDL mediante una architecture declaration, secondo la sintazzi seguente:

```
architecture architecture_name of entity_name is
    [declarations]
begin
    [implementation]
end architecture_name;
```

La prima cosa che risulta evidente è che ogni architecture è associata ad una ed una sola entity. Non è invece vero il vice versa: è infatti possibile specificare più architecture alternative per una stessa entity e selezionarne una specifica prima di procedere alla sintesi oppure alla simulazione. L'associazione di una architecture specifica ad una entity prende il nome di **configuration declaration**. Nei progetti di complessità media o bassa, l'utilità di avere diverse architecture alternative per un modulo risulta di scarsa utilità e quindi è una possibilità usata solo molto raramente.
La prima sezione di una architecture, opzionale ed indicata nella sintassi come declarations, è costituita da un elenco di dichiarazioni. Le dichiarazioni possono essere di quattro tipi:

1. Dichiarazioni di **costanti**: Nomi, tipi e valori delle costanti simboliche utilizzate nella specifica.
2. Dichiarazioni di **segnali**: Nomi e tipi dei segnali che saranno usati nella specifica della funzionalità del modulo.
3. Dichiarazioni di **tipi**: Nomi e definizioni di tipi definiti dall'utente ed utilizzati nella specifica.
4. Dichiarazioni di **componenti**: Nomi ed interfaccie dei moduli utilizzati nella specifica della architecture in esame.

Senza entrare ora nei dettagli delle varie tipologie di dichiarazioni, si tenga presente che le dichiarazioni presenti nella sezione di una data architecture hanno visibilità soltanto per quella architecture. Questa circostanza può comportare alcune complicazioni prevalentemente nella definizione di tipi aggiuntivi.
Tra le parole chiave **`begin`** ed **`end`** compare la sezione ***`implementation`***, destinata a raccogliere la descrizione della funzionalità che il modulo deve svolgere. La funzionalità di un intero circuito, quindi, si trova quindi distribuita nelle architecture declaration delle diverse entity che costituiscono i moduli del sistema.
A titolo di esempio si consideri il modulo full adder usato nell'esempio precedente e se ne dia la specifica della funzionalità. Una possibilità è la seguente:

```
architecture first of full_adder is
begin
    s <= a xor b xor cin;
    cout <= (a and b) or (b and c) or (a and c);
end first;
```

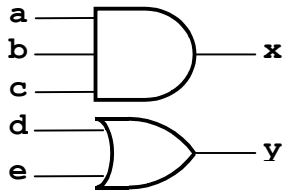
In questo semplice esempio tutti i segnali utilizzati sono o gli ingressi del modulo o le sue uscite. Per questo motivo non è necessaria alcuna parte dichiarativa. Per inciso, si noti che i segnali di ingresso **`a`**, **`b`** e **`cin`**, come richiesto ed eveidenziato in precedenza, sono soltanto letti mentre i segnali di uscita **`s`** e **`cout`** sono soltanto scritti. Consideriamo ora la descrisione seguente:

```
architecture second of full_adder is
    signal p1, p2, p3: bit
begin
    s <= a xor b xor cin;
    p1 <= a and b;
    p2 <= b and c;
    p3 <= a and c;
    cout <= p1 or p2 or p3;
end second;
```

I tre segnali temporanei, utilizzati per il calcolo di risultati intermedi, devono essere dichiarati esplicitamente nella sezione dichiarativa dell'architecture declaration e devono avere un tipo conforme al contesto in cui sono utilizzati. Si noti infine che tali segnali sono privi di direzionalità e possono quindi essere sia letti sia scritti.
Affrontiamo ora un aspetto essenziale delle architecture. La parte implementativa è in generale costituita da un insieme di statement quali assegnamenti, assegnamenti condizionali, costrutti di selezione, istanziazioni ecc. Tali statement sono da considerarsi paralleli ovvero il loro risultato è calcolato in parallelo. Nella specifica di componenti hardware, ciò è una necessità in quanto è chiaro che tutti gli elementi di un circuito elaborano i propri ingressi in modo inerentemente parallelo. Un primo esempio, molto semplice, chiarisce questo concetto:

```
architecture par_one of dummy is
begin
    x <= a and b and c;
    y <= d or e;
end par_one;
```

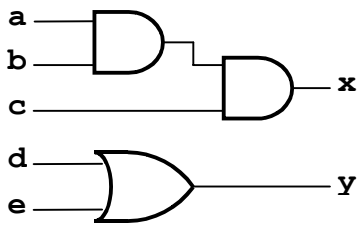
Questa definizione corrisponde alla realizzazione circuitale seguente:



Le due porte logiche calcolano ovviamente il valore delle uscite in modo parallelo. Risulta perciò evidente che invertire l'ordine degli assegnamenti nell'architecture non influisce in alcun modo sul risultato. Un caso leggermente più complsso è mostrato dalla architecture seguente:

```
architecture par_two of dummy is
    signal t: bit;
begin
    t <= a and b;
    x <= t and c;
    y <= d or e;
end par_two;
```

Il circuito che ne risulta è mostrato nella figura seguente:



Dallo schema circuitale appare chiaro che il valore del risultato **x** è disponibile solo dopo che la prima porta AND ha calcolato il prodotto logico **a and b** e dopo che la seconda ha utilizzato tale risultato per produrre il valore finale. Questo è esatto solo in parte, infatti la seconda porta AND calcola continuamente il prodotto logico dei suoi ingressi, a prescindere da come e quando tali ingressi sono calcolati. L'affermazion precedente riguardo il calcolo del risultato **x** deve quindi essere riformulata dicendo che il risultato **x** è sempre disponibile e viene ricalcolato in modo continuo. Tuttavia una variazione sugli ingressi **a**, **b** o **c** si ripercuoterà sull'uscita aggiornando il valore di **x** secondo lo schema di valutazione delle espressioni che implica una dipendenza tra i dati. In altre parole è vero che le due porte AND e la porta OR elaborano gli ingressi in parallelo, anche se la dipendenza tra i vari segnali impone un ordine di propagazione dei risultati intermedi.
Secondo tale principio, quindi, è possibile anche in questo caso scambiare l'ordine delle espressioni senza modificare il comportamento del circuito risultante. La seguente architecture è del tutto equivalente alla precedente:

```
architecture par_three of dummy is
    signal t: bit;
begin
    y <= d or e;
    x <= t and c;
    t <= a and b;
end par_three;
```

3. Tipi

Il VHDL dispone di un numero elevato di tipi. Ai fini della sintesi, tuttavia, solo pochi di essi risultano utlizzabili in quanto riconosciuti dagli strumenti di sintesi automatica. Nei seguenti paragrafi sono descritti i tipi base del linguaggio VHDL e sono introdotti alcuni concetti di base riguardo le slices ed i tipi definititi dall'utente.

3.1 Il tipo bit

Il tipo **bit** è il più semplice tipo disponibile in VHDL. Tale tipo rappresenta un valore binario e può unicamente assumere i valori logici 0 ed 1. Si noti che le costanti 0 ed uno devono essere racchiuse tra singoli apici (quindi **'0'** e **'1'**) per distinguerle dai valori numerici interi 0 ed 1.
Gli operatori definiti per tale tipo sono soltanto gli operatori di assegnamento, gli operatori confronto e gli operatori logici. I seguenti statement sono validi:

```
x <= a and b;
y <= '1';
z <= a xor (b and not c);
```

Quelli che seguono sono invece scorretti ed i commenti indicano il motivo:

```
x <= a + b;  -- L'operatore + non è definito per il tipo bit
y <= 1;      -- Le costanti di tipo bit richiedono gli apici
```

Spesso è utile raggruppare più segnali sotto un nome comune, ovvero utilizzare un array. Il linguaggio VHDL dispone a tale scopo del tipo **bit_vector**. Un bit vector è sostanzialmente un insieme di segnali contraddistinti da un nome comune e da un indice. È possibile accedere in lettura o in scrittura ai vari elementi del vettore mediante l'uso di indici. Analizziamo anzitutto la sintassi della dichiarazione di un segnale di tipo bit vector.

```
signal_name: bit_vector( index1 {to|downto} index2 );
```

La dichiarazione specifica un segnale composto formato da ***index2-index1+1*** segnali semplici di tipo bit. Inoltre il vettore ha un ordinamento che è determinato dalla paraola chiave **to** o **downto** nella parte dichiarativa degli indici.
L'informazione relativa all'ordinamento determina quale deve essere considerato il bit più significativo. Utilizzando la forma ***index1*** **to** ***index2*** il bit più significativo è quello con indice ***index1***, metre nella forma **downto** il bit più significativo è quello in posizione ***index2***. È sottinteso che tale distinzione ha senso solo quando il valore del vettore è interpretato nell'insieme piuttosto che come collezione di singoli bit. Questo accade ad esempio quando si utilizza un bit vector per rappresentare un valore numerico intero in codifica binaria naturale o in complemento a 2.
Dalla forma della dichiarazione appare evidente che gli indici iniziale e finale del vettore possono essere assolutamente arbitrari, purché interi, non negativi e consistenti con la specifica direzione.
Le seguenti dichiarazioni sono quindi valide:

```
a: bit_vector( 0 to 3 );       -- 4 elementi
b: bit_vector( 17 to 24 );     -- 8 elementi
c: bit_vector( 16 downto 1 );  -- 16 elementi
```

mentre le seguenti non lo sono:

```
a: bit_vector( 3 to 0 );          -- Errore: 3 > 0
b: bit_vector( -4 to +4 );        -- Errore: -2 negativo
c: bit_vector( 15 downto '0' );   -- Errore '0' non è un intero
```

Per riferirsi ad un elemento specifico del vettore si utilizza l'indice dell'elemento, racchiuso tra parentesi tonde. Così l'elemento 5 del vettore **a** si indica con **a(5)**. È sottinteso che l'accesso ad un elemento al di fuori dei limiti del vettore costituisce un errore.
Il tipo **bit** (e di conseguenza il tipo **bit_vector**) presenta alcune limitazioni. In particolare non consente di specificare condizioni di indifferenza e di alta impedenza. Si noti che quest'ultimo problema non dovrebbe essere contemplato a livello di sintesi logica bensì ad un livello più basso e precisamente a livello elettrico. Nonostante ciò, spesso accade di dover specificare componenti che in condizioni particolari presentano in uscita non tanto un valore logico bensì una alta impedenza, ad indicare un disaccoppiamento tra gli elementi che il componente connette. Vedremo nel seguito che questo problema è superato dai tipi risolti e che il tipo bit, nella pratica, è utilizzato molto raramente.

3.2 Il tipo integer

Il tipo **integer** rappresenta valori interi a 32 bit e può essere utilizzato per la sintesi. A tale proposito si presentano i due seguenti problemi. In primo luogo deve essere definito comen interpretare un valore binario su 32 bit dal punto di vista del segno. Per default, i valori rappresentati sono interi senza segno. Il secondo problema riguarda lo spreco di risorse che deriva dal fatto che ogni volta che un segnale intero viene utilizzato, lo strumento di sintesi lo considera come un bus a 32 bit e istanzia connessioni e componenti confacenti a tale dimensione.
Nella specifica di un contatore modulo 8, ad esempio, l'utilizzo di un segnale di tipo integer produrrebbe la sintesi di un sommatore a 32 bit e di un banco di 32 registri benché 3 bit sarebbero sufficienti.

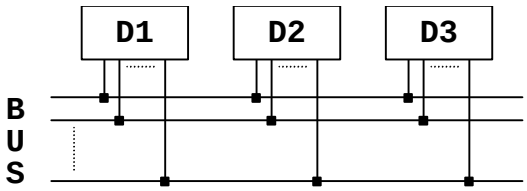
3.3 Tipi IEEE

Come accenntao in precedenza, a volte è necessario anche ai fini della sintesi, specificare come valore di un segnale un valore diverso da 0 e da 1. Il VHDL non dispone di un tale tipo, tuttavia esiste una libreria standard, la libreria **IEEE**, che definisce alcuni tipi aggiuntivi. I tipi in esame utilizzano un sistema logico a 9 valori, descritti di seguito:

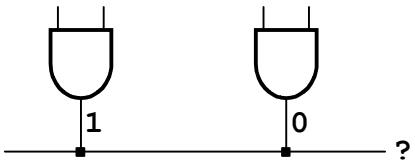
- **'0'** Valore logico 0.
- **'1'** Valore logico 1.
- **'Z'** Alta impedenza.
- **'X'** Indeterminato. Può essere 0 o 1.
- **'U'** Indefinito. Il valore non è mai stato assegnato.
- **'W'** Segnale debole. Non è possibile interpretarlo come 0 o 1.
- **'L'** Segnale debole. Interpretabile come 0.
- **'H'** Segnale debole. Interpretabile come 1.
- **'-'** Don't care.

Di questi, quelli di maggiore interesse e di uso più comune sono, oltre ai valori logici **0** ed **1**, il valore di alta impedenza **'Z'** ed i due valori di don't care **'X'** e **'-'**. Si noti che tra il valore indeterminato 'X' ed il valore indeterminato '-' esiste una differenza sostanziale: il primo viene generato da qualsiasi simulatore ogni volta che non è possibile determinare il valore di un segnale, mentre il secondo può essere solo assegnato e non generato in simulazione. Tale differenza non ha alcun effetto ai fini della sintesi.

La libreria **IEEE**, come ogni libreria VHDL, è suddivisa in **package** ognuno dei quali definisce alcuni oggetti che possono esssere utilizzati nella progettazione. In particolare il package **std_logic_1164** definisce i due tipi risolti **std_logic** e **std_ulogic** ed i correspondenti tipi vettoriali **std_logic_vector** ed **std_ulogic_vector**.
Di particolare interesse sono i tipi **std_logic** e **std_logic_vector** in quanto sono **tipi risolti** e sono supportati sia dagli strumenti di sintesi, sia dagli strumenti di simulazione. Per chiarire il concetto di tipo risolto faremo riferimento ad un caso tipico del suo utilizzo. Si consideri un bus sul quale sono connessi più dispositivi in wired-or, come nella figura seguente:



Ogni linea del bus è quindi connessa a più dispositivi open-collector, i quali possno forzare valori logici diversi in modo simultaneo, realizzando qello che si chiama wired-or. Benché questa situazione non abbia alcun senso dal punto di vista della progettazione logica, dal punto di vista elettrico è non solo sensata ma anche molto utilizzata e a volte persino necessaria. Supponiamo, ad esempio, che un modulo mandi in uscita su una certa linea il valore logico 1 metre, nello stesso istante, un altro modulo manda in uscita sulla stessa linea il valore logico 0.



Quale è il valore assunto dalla linea in questione? La risposta è che dipende dalle caratteristiche elettriche dei dispositivi, dai livelli di tensione utilizzati, dalla scelta di una logica positiva o negativa e da altri fattori che comunque esulano dal progetto logico. I tipi risolti permettono di definire il valore assunto dalla linea in questione mediante un insieme di regole associate alla logica a nove valori descritta in precedenza.
Bisogna sottolineare un altro aspetto importante: proprio per l'impossibilità di stabilire il valore risultatnte da una situazione di questo tipo, il linguaggio VHDL proibisce la connessione di più segnali in modalità wired-or, a meno che per il tipo dei segnali in questione non siano define una o più funzioni di risoluzione.
Dopo questa breve introduzione alla logica a 9 valori e ai concetti di tipo risolto e di funzione di risoluzione, vediamo come possono essere usati i tipi definiti dal package **std_logic_1164** della libreria **IEEE**. Tali tipi, sono dal punto di vista logico, esattamente equivalenti al tipo **bit** ed al tipo **bit_vector**. Anche per essi, infatti, sono definite le operazioni di assegnamento, di confronto e le operazioni logiche fondamentali.
La libreria **IEEE** dispone di tre package (**std_logic_unsigned**, **std_logic_signed** e **std_logic_arith**), di cui parleremo più dettagliatamente nel seguito, che ridefiniscono opportunamente alcuni operatori aritmetici e di confronto per questi tipi,rendendoli particolarmente adatti alla specifica di moduli ai fini della sintesi. Grazie a tali package, combinati con il package **std_logic_1164** è possibile definire, ad esempio, tre segnali **a**, **b** e **c** di tipo **std_logic_vector** e di dimensioni identiche e scrivere uno statement come:

```
c <= a + b;
```

intendendo con questo assegnare a **c** la rappresentazione binaria del risultato della somma dei valori numerici le cui rappresentazioni binarie sono i segnali **a** e **b**. Affronteremo meglio nel seguito questo tipo di problematiche.

3.4 Tipi definiti dall'utente

Un tipo definito dall'utente altro non è che un nuovo tipo le cui caratteristiche devono essere specificate dall'utente, il quale deve, eventualmente, farsi carico di ridefinire alcuni operatori in modo da poterli utilizzare su segnali del nuovo tipo. Il VHDL è un linguaggio molto flessibile e come tale dispone di diversi meccanismi di costruzione di tipi. Tuttavia, ai fini della specifica di sistemi di media e bassa complessità è sufficiente considerare due soli meccanismi: il **subtyping** e l'**enumeration**.
Il subtyping consiste nel definire un nuovo tipo come equivalente ad un tipo esistente limitando l'intervallo di variazione di quest'ultimo. Un tale meccanismo ha senso prevalentemente per i tipi interi ed in particolare per il tipo **integer**. La sintassi generale è la seguente:

```
subtype new_type_name is type_name range val1 to val2;
```

Tale definizione indica che il tipo ***new_type_name*** è equivalente al tipo ***type_name*** salvo per il fatto che può rappresentare solo i valori compresi tra ***val1*** e ***val2***. Nella pratica, il caso senza dubbio più utile riguarda il subtypeing per gli interi al fine di utilizzare per la loro rappresentazione binaria il minor numero possibile di bit. Volendo ad esempio, rappresentare un valore intero in codifica binaria naturale su 5 bit si può definire un subtype del tipo integer con estremi 0 e 2^5-1=31, ovvero:

```
subtype small_integer is integer range 0 to 31;
```

Benchè questa definizione introduca un tipo che può essere visto a tutti gli effetti come un vettore di 5 bit l'uso di segnali associati ad esso è sconsigliato rispetto all'alternativa di utilizzare segnali di tipo **std_logic_vector(0 to 4)** oppure come l'analogo **bit_vector(0 to 4)**.
Il secondo meccanismo di costruzione dei tipi prevede la definizione di un nuovo tipo mediante l'enumerazione di tutti i valori – simbolici – che esso può assumere. La sintassi per una tale definizione è la seguente:

```
type new_type_name is ( val0, val1, ..., valN );
```

In questa definizione ***val1***, ***val2*** ecc. rappresentano identificatori generici che possono essere interpretati come costanti simboliche. Si badi che a priori non è definita alcuna associazione tra tali valori simbolici e un qualsiasi equivalente numerico, né dal punto di vista del valore numerico e neppure dal punto di vista del numero di bit impiegati per rappresentarlo. Per questo motivo i tipi enumerativi devono essere usati con cognizione di causa altrimenti possono essere fonte di gravi errori.
Un caso tipico di utilizzo di un tipo enumerato si presenta nella descrizione delle macchine a stati finiti in cui spesso, la specifica è spesso fornita sotto forma di tabella degli stati in cui gli stati hanno un nome simbolico e tale lo si vuole mantenere fino alla scelta di un codice e di una codifica specifiche. Come esempio si consideri una macchina a stati costituita dai 5 stati RES, INIT, COMP, ERR e OK. La dichiarazione di un tipo adatto a rappresentare tali stati sarebbe:

```
type status is ( RES, INIT, COMP, ERR; OK );
```

Per un segnale pres di questo tipo un assegnamento come:

```
pres <= INIT;
```

costituisce uno statement corretto, così come un confronto come

```
if( pres = ERR ) ...
```

è una condizione logica valida. Infine è importante sottolinare un ulteriore aspetto dei tipi enumerativi. Benchè né il codice né la codifica siano assegnati a priori ai simboli che definiscono il tipo, tuttavia è fissato l'ordinamento. Questa circostanza permette di compiere due operazioni: la definizione di un sottotipo enumerativo ed il confronto tra segnali di tipo enumerativo. Una definizione valida di sottotipo potrebbe essere la seguente:

```
type states is ( INIT, S0, S1, S2, S3, WRITEOUT, FINISHED );
type kernel_states is states range S0 to WRITEOUT;
```

Per quanto concerne invece il confronto, va tenuto presente che lo strumento di sintesi assegna implicitamente la codifica binaria naturale ai simboli definiti basandosi sull'ordine in cui essi appaiono nella definizione. Quindi, nel caso del tipo states dell'esempio precedente, si assume la codifica seguente:

```
INIT       000
S0         001
S1         010
S2         011
S3         100
WRITEOUT   101
FINISHED   110
```

Come regola generale è quindi bene non utilizzare gli operatori di confronto tra valori simbolici di un tipo enumerativo. Per la stessa ragione è sconsigliato utlizzare operatori aritmetici tra segnali di tipo enumerativo.

4. Reti combinatorie

4.1 Slice e concatenamento

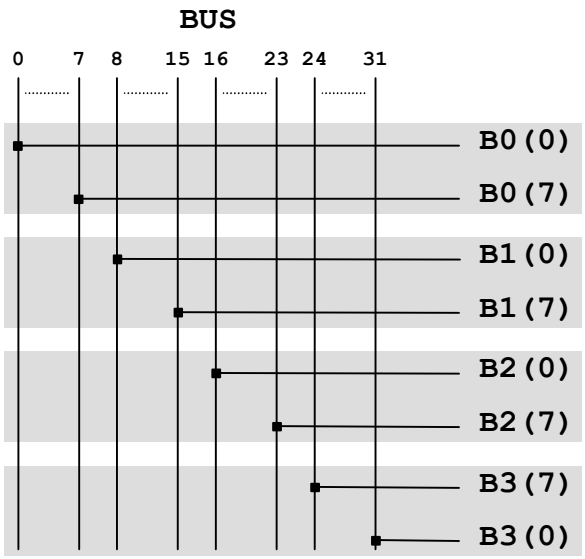
Prima di entrare nel dettaglio della specifica delle reti combinatorie è utile ntrodurre il concetto di **slice** e l'operazione di **concatenamento** di segnali. Una slice è una porzione di un vettore, ovvero, dal punto di vista circuitale, un sottoinsieme delle linee di un segnale composto. Per specificare una slice si usa la sintassi seguente:

```
signal_name( index1 {to|downto} index2 )
```

in cui ***index1*** ed ***index2*** devono essere indici validi per il vettore ***signal_name*** e devono rispettare l'oridnamento imposto dalla clausola **to** o **downto**. Si consideri, ad esempio, la seguente porzione di architecture:

```
architeture rtl of dummy is
    signal BUS: std_logic_vector(0 to 31);
    signal B0, B1, B2, B3: std_logic_vector(0 to 7);
begin
    …
    B0 <= BUS(0 to 7);
    B1 <= BUS(8 to 15);
    B2 <= BUS(16 to 23);
    B3 <= BUS(31 downto 24);
    …
end rtl;
```

Questo corrisponde, in termini circuitali alla situazione mostrata in figura. Si noti che gli elementi del vettore **B3** corrispondono agli elementi del vettore **BUS** tra 24 e 31, presi in ordine decrescente, come specificato dalla clausola **downto**.



Il meccanismo delle slice è spesso utilizzato per estrarre, sotto un nome comune e più significativo alcune liee di un bus. Un esempio tipico è quello mostrato, in un bus che porta una intera word (32 bit) è spezzato nei byte (8 bit) che la compongono.
Il meccanismo del concatenamento funziona in modo inverso, ovvero permette di raggruppare più segnali sotto uno stesso nome. L'operatore VHDL per il concatenamento è l'ampersand (**&**). Ad esempio se si volesse ricomporre un nuovo bus **BUS2** a partire dai quattro byte mostrati nell'architettura precedente, scambiandone l'ordine (cioè con **B3** come più significativo e **B0** come byte meno significativo si può ricorrere ad una espressione come la seguente:

```
BUS2 <= B3(7 downto 0) & B2 & B1 & B0;
```

L'esempio suppone che **BUS2** sia stato dichiarato come di tipo **std_logic_vector** ed avente 32 elementi. In altri casi, tale operatore è utilizzato per raccogliere sotto uno stesso nome segnali indipendenti al fine di poter effettuare confronti più rapidi. Si consideri, a tal proposito, il caso in cui si deve verificare se a vale 0, b vale 1 e c vale 0. Sono possibili due soluzioni. La prima è semplicemente la traduzione della condizione direttamente in VHDL, cioè:

```
if ( a = '1' and b = '0' and c = '1' ) then ...
```

tale soluzione è accettabile e comoda se il test sulle tre variabili è uno solo. Nel caso in cui si vogliano verificare diverse condizioni sui valori di a, b e c allora conviene adottare il meccanismo del concatenamento. Per prima cosa si dichiara un segnale temporaneo della dimensione adeguata, nel caso in esame un segnale di 3 bit:

```
signal temp: std_logic_vector(0 to 2);
```

quindi si costruisce il vettore temporaneo concatenando le variabili originali:

```
temp <= a & b & c;
```

ed infine si sccrivono tutte le condizioni nella forma compatta mostrata di seguito:

```
if ( temp = "101" ) then ...
```

Questo stile è più leggibile e più sintetico del precedente ed è utilizzato spesso.

4.2 Espressioni logiche

Le espressioni logiche possono essere tradotte in VHDL in modo molto semplice e diretto. Ricordiamo dapprima che per rappresentare correttamente variabili booleane si usano i tipi **bit** e **std_logic**. In rari casi può essere neccessario o convniente utilizzare anche il tipo non risolto **std_ulogic**. Gli operatori VHDL disponibili per la costruzione di espressioni logiche ed il loro significato sono riassunti nella tabella seguente:

Operatore logico	Operatore VHDL	Espressione logica	Espressione VHDL
and, •	**and**	ab, a•b	**a and b**
or, +	**or**	a+b	**a or b**
not, ',!	**not**	!a, a'	**not a**
xor, ⊕	**xor**	a ⊕ b	**a xor b**

L'espressione logica F = ac' + d(a' + b'c) ad esempio, è tradotta in VHDL dall'assegnamento:

```
F <= a and (not c) or d and ( (not a) or ((not b) and c) );
```

Alcune delle parentesi di tale espressione possono essere rimosse in quanto la valutazione dell'espressione segue le regole di precedenza e associatività definite per gli operatori dell'algebra booleana. Tuttavia una scirittura come quella mostrata non modifica l'espressione pur aumentandone la chiarezza e la leggibilità. Una espressione logica descrive come alcuni segnali devono essere combinati per produrre altri segnali e ciò è realizzato mediante l'operatore di assegnamento tra segnali **<=**. Le espressioni logiche quindi appartengono allo stile di specifica a livello RTL.

4.3 Tabelle della verità

Una tabella della verità è in sostanza una lista di condizioni sulle variabili di ingresso alle quali è subordinato il valore della funzione che si sta specificando. La seguente tabella della verità:

a	b	f
0	0	1
0	1	0
1	0	1
1	1	1

può quindi essere letta nel seguente modo: se **a** vale 0 e **b** vale 0 allora **f** vale 1, altrimenti se **a** vale 0 e **b** vale 1 allora **f** vale 0, altrimenti ecc. Questo modo di leggere la tabella della verità ammette una rappresetazione molto semplice in VHDL grazie al costrutto di **assegnamento condizionale** la cui è la seguente:

```
signal_name <= const_1 when cond_1 else
               const_2 when cond_2 else
               ...
               const_N when cond_N else
               const_default;
```

In tale costrutto il segnale **signal_name** assume i valori costanti **const_1**, ..., **const_N**, **const_default** a seconda del verificarsi o meno delle condizioni **cond_1**, ..., **cond_N**. In particolare, se nessuna delle consdizioni è vera, allora ilsegnale assume il valore di default **const_default**.
La tabella della verità dell'esesempio è quindi tradotta in VHDL dal seguente assegnamento:

```
f <= '1' when a='0' and b='0' else
     '0' when a='0' and b='1' else
     '1' when a='1' and b='0' else
     '1' when a='1' and b='1';
```

A questo punto è importante notare che il linguaggio VHDL prevede che nelle espressioni di assegnamento condizionato destinate a rappresentare reti puramente combinatorie, tutti i possibili casi siano esplicitamente enumerati. Torniamo ora all'esempio appena mostrato. Se le variabili a e b sono di tipo bit, l'assegnamento è corretto in quanto il tipo bit prevede solo i due valori 0 ed 1 e quindi tutti i possibili casi sono stati menzionati. Al conrario, se le variabili a e b fossero di tipo std_logic, l'assegnamento non sarebbe corretto in quanto iltipo in questione ammette nove

possibili valori. Esplicitare tutti i casi possibili vorrebbe dire in questo caso scrivere tutte le 81 possibili condizioni. Una possibile soluzione potrebbe essere quella di esprimere esplicitamente solo le prime 3 condizioni ed assegnare un valore di default alla quarta condizione e a tutte quelle non esplicitamente citate. Si avrebbe, secondo questo schema:

```
f <= '1' when a='0' and b='0' else
     '0' when a='0' and b='1' else
     '1' when a='1' and b='0' else
     '1';
```

Seguendo tale strategia è possibile anche semplificare la scrittura di una tabella della verità prendendo in considerazione solamente l'on-set oppure l'off-set. Nel caso in esame si potrebbe scrivere un assegnamento del tipo:

```
f <= '0' when a='0' and b='1' else
     '1';
```

Questo approccio, tuttavia, non è la mgiliore soluzione possibile in termini di chiarezza ed inoltre è applicabile solo alle funzioni completamente specificate, cioè quelle per cui il dc-set è vuoto.
Consideriamo ora una tabella della verità per una funzione non completamente specificata:

a	b	f
0	0	1
0	1	0
1	0	-
1	1	1

Al fine di permettere allo strumento di sintesi di individuare la soluzione ottima per tale funzione è necessario esprimere le condizioni di indifferenza presenti nella specifica. Come prima possibilità, ripercorrendo le varie soluzioni proposte per le funzioni completamente specificate, consiste nell'esprime esplicitamente tutti i valori assunti dalla funzione in corrispondenza delle possibili combinazioni dei valori in ingresso. Si noti, a tal proposito, che in qusto caso è necessario esprimere il concetto di condizione di indifferenza e ciò non è previsto dal tipo **`bit`** per cui è necessario ricorrere a segnali di tipo **`std_logic`**. Tale tipo infatti prevede due rappresentazioni per un valore indeterminato e cioè **`'X'`** e **`'-'`**. Nella situazione in esame entrambi possono essere usati indifferentemente. Una possibile soluzione è quindi la seguente:

```
f <= '1' when a='0' and b='0' else
     '0' when a='0' and b='1' else
     '-' when a='1' and b='0' else
     '1' when a='1' and b='1';
```

Questa soluzione, tuttavia, soffre del problema già visto e cioè non esprime tutti i possibili casi. Una scelta migliore è la seguente:

```
f <= '1' when a='0' and b='0' else
     '0' when a='0' and b='1' else
     '-' when a='1' and b='0' else
     '1';
```

Volendo, infine, specificare la tabella nella forma più sintetica possibile, si ricorre alla forma in cui solo due insiemi tra on-set, off-set e dc-set sono esplicitamente specificati. Scegliendo, ad esempio, off-set e dc-set si avrebbe:

```
f <= '0' when a='0' and b='1' else
     '-' when a='1' and b='0' else
     '1';
```

Scegliendo invece on-set ed off-set si avrebbe l'assegnamento:

```
f <= '1' when a='0' and b='0' else
     '1' when a='1' and b='1' else
     '0' when a='0' and b='1' else
     '-';
```

Il linguaggio VHDL dispone di un costrutto alternativo per l'assegnamento condizionato che offre la possibilità di sfruttare espressioni di confronto sintetiche mediante l'uso dell'operatore di concatenamento. La sua sintassi generale è la seguente:

```
with test_signal select
    signal_name <= const_1 when case_1,
                   const_2 when case_2,
                   ...
                   const_N when case_N,
                   const_default when others;
```

Il significato è il seguente: il valore di ***test_signal*** è confrontato con le costanti ***case_1***, ..., ***case_N*** ed al segnale ***signal_name*** è assegnato il corrispondente valore di una delle costanti ***const_1***, ..., ***const_N***. Se ***signal_test*** è deiverso da tutte le costanti ***case_1***, ..., ***case_N*** allora a ***signal_name*** è assegnato il valore di ***const_default***. È evidente che la semantica di questo costrutto è equivalente a quella dell'assegnamento condizionato appena visto, tuttavia, come modtra l'esempio seguente, questa forma è spesso più sintetica e leggibile.
La tabella della verità:

a	b	f
0	0	1
0	1	0
1	0	-
1	1	1

può essere codificata nel modo seguente. Innanzitutto è necessario definire un segnale vettoriale temporaneo destinato a contenere la concatenazione delle variabili su cui si deve effettuare il confronto, nel caso in esame le variabili **a** e **b**. Ciò e realizzabile come segue:

```
architecture rtl of dummy is
    signal temp: std_logic_vector(0 to 1);
begin
    …
    temp <= a & b;
    …
```

Quindi si procede alla verifica delle condizioni non più sulle variabili di ingresso originali, bensi sul nuovo vettore temporaneo, ovvero:

```
with temp select
    f <= '1' when "00",
         '0' when "01",
         '-' when "10",
         '1' when "11",
         '-' when others;
```

Si nota immediatamente che questa forma offre una rappresentazione immediate di una tabella della verità. Inoltre è bene sottolineare l'imoprtanza dell'ultima clausola del costrutto: in primo luogo essa è necessaria in quanto si sta utilizzando un segnale in logica a 9 valori e quindi molticasi non sono contemplati esplicitamente; in secondo luogo è bene notare che è buona norma non accorpare il dc-set con tale clausola, principalmente per ragioni di chiarezza. La parola chiave **others**, come sarà chiaro a questo punto, è una forma compatta per indicare tutti i casi non menzionati esplicitamente.

4.4 Tabelle di implicazione

Consideriamo ora il caso in un priority encoder da 6 a 3 bit. Il comportamento di tale elemento può essere descritto in modo molto compatto mediante una tablla delle implicazioni. La tabella della verità, per contro, avrebbe un numero molto elevato di righe (2^6=64). La tabella in questione è la seguente:

a	b	c	d	e	f	f0	f1	f2
0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	1
0	0	0	0	1	-	0	1	0
0	0	0	1	-	-	0	1	1
0	0	1	-	-	-	1	0	0
0	1	-	-	-	-	1	0	1
1	-	-	-	-	-	1	1	0

Nell'esempio assumiamo che la funzione da realizzare sia specificata unicamente mediante variabili scalari. In tal caso si dovrà ricorrere a due variabili temporanee che raccolgano in vettori gli ingressi e le uscite. Vediamo ora la specifica completa, partendo dalla entity declaration.

```
entity pri_enc is
    port( a:  in  std_logic;
          b:  in  std_logic;
          c:  in  std_logic;
          d:  in  std_logic;
          e:  in  std_logic;
          f:  in  std_logic;
          f0: out std_logic;
          f1: out std_logic;
          f2: out std_logic
    );
end pri_enc;
```

L'achitecture che specifica il comportamento traducendo di fatto la tabella delle implicazioni è quindi la seguente:

```
architecture rtl of pri_enc is
    signal enc_in:  std_logic_vector(0 to 5);
    signal enc_out: std_logic_vector(0 to 2);
begin

    -- Concatenates inputs into enc_in
    enc_in <= a & b & c & d & e & f;

    -- Implements the behaviour
    with enc_in select
        enc_out <= "000" when "000000",
                   "001" when "000001",
                   "010" when "00001-",
                   "011" when "0001--",
                   "100" when "001---",
                   "101" when "01----",
                   "110" when "1-----",
                   "---" when others;

    -- Splits the output vector enc_out
    f0 <= enc_out(0);
    f1 <= enc_out(1);
    f2 <= enc_out(2);

end rtl;
```

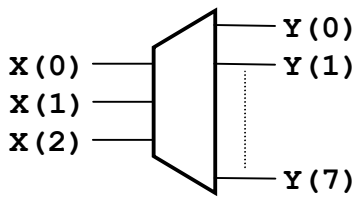
Questo esempio mostra chiaramente come il costrutto **`with`** … **`select`** sia sintetico e leggibile.

4.5 Moduli di uso comune

In questo paragrafo sono raccolte le descrizioni, opportunamente commentate, di alcuni componenti combinatori di uso comune.

4.5.1 Decoder

Un decoder è un dispositivo con N linee di ingresso e 2^N linee di uscita, idealmente numerate da 0 a 2^N-1. Il funzionamento è il seguente: tutte le linee di uscita hanno valore 0 tranne quella il cui indice è fornito in ingresso in codifica binaria naturale. Il simbolo di un decoder 3-a-8, cioè con tre bit di ingresso e 8 di uscita, è il seguente:



Il comportamento di un decoder può essere descritto mediante una tabella della verità.

x	y
000	00000001
001	00000010
010	00000100
011	00001000
100	00010000
101	00100000
110	01000000
111	10000000

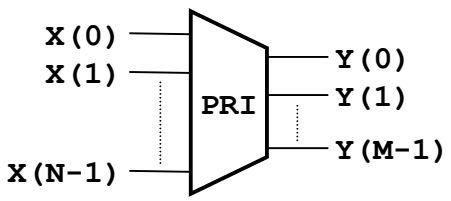
Si può anche vedere un decoder come un elemento che trasforma una codifica binaria naturale in una codifica one-hot. Basandosi sulla tabella della verità è immediato passare alla specifica in linguaggio VHDL:

```
entity decoder_3_8 is
    port( dec_in:  in  std_logic_vector(0 to 2);
          dec_out: out std_logic_vector(0 to 7)
    );
end decoder_3_8;

architecture rtl of decoder_3_8 is
begin
    with dec_in select
        dec_out <= "00000001" when "000",
                   "00000010" when "001",
                   "00000100" when "010",
                   "00001000" when "011",
                   "00010000" when "100",
                   "00100000" when "101",
                   "01000000" when "110",
                   "10000000" when "111",
                   "--------" when others;
end rtl;
```

Questo tipo di elemento è spesso utilizzato per decodificare un indirizzo di una memoria. Gli ingressi sono appunto l'indirizzo di memoria in codifica binaria naturale mentre l'uscita è tale da attivare una sola delle linee della memoria cui si vuole accedere.

4.5.2 Priority encoder

Un priority encoder è un componente con N linee di ingresso e $M=\lceil \log_2(N+1) \rceil$ linee di uscita. La funzione svolta da tale elemento è quella di fornire in uscita la posizione dell'uno più significativo presente sulle linee di ingresso. Il simbolo normalmente utilizzato è il seguente:



Riportiamo di seguito la specifica di un priority encoder con 6 bit di ingresso e 3 di uscita. Per quanto riguarda i signali di ingresso/uscita, la scelta migliore è quella di utlizzare direttamente vettori, come mostra l'entity declaration seguente:

```
entity pri_enc is
    port( enc_in:  in  std_logic_vector(0 to 5);
          enc_out: out std_logic_vector(0 to 2)
    );
end pri_enc;
```

Nei paragrafi cprecedenti si sono già presentate alcune possibili descrizioni delle architecture di un tale componente. Avendo scelto segnali vettoriali, la realizzazione della specifica del comportamento risulta molto semplificata in quanto non è necessario ricorrere a slicing e concatenamento. Il codice seguente mostra una possibile realizzazione.

```
architecture rtl of pri_enc is
begin
    with enc_in select
        enc_out <= "000" when "000000",
                   "001" when "000001",
                   "010" when "00001-",
                   "011" when "0001--",
                   "100" when "001---",
                   "101" when "01----",
                   "110" when "1-----",
                   "---" when others;
end rtl;
```

4.5.3 Parity encoder

Un parity encoder è un elemento che calcola la parità della parola in ingresso. La parità di parola, ovvero di una stringa di bit, è una indicazione del numero di uno che compaiono nella stringa. Esistono due tipi di parità: la parità pari e la parità dispari. Nel primo caso il bit di parità vale 0 se il numero di uno presenti nella parola d'ingresso è pari, mentre nel secondo caso vale zero se tale numero è dispari. Per tale elemento non esiste un simbolo standard. La tabella della verità che descrive il comportamento di un parirty encoder a parità pari è la seguente:

x	y
000	0
001	1
010	1
011	0
100	1
101	0
110	0
111	1

La specifica in VHDL segue lo schema generale adottato per le tabelle della verità, ovvero:

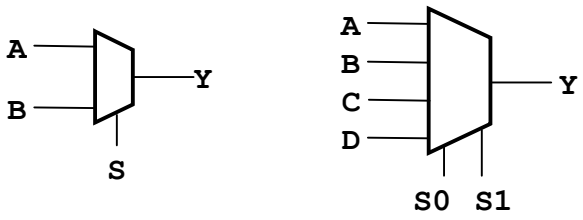
```
entity parity_enc is
    port( enc_in:  in  std_logic_vector(0 to 2);
          enc_out: out std_logic );
end parity_enc;
```

```
architecture rtl of parity_enc is
begin
    with enc_in select
        enc_out <= '0' when "000",
                   '1' when "001",
                   '1' when "010",
                   '0' when "011",
                   '1' when "100",
                   '0' when "101",
                   '0' when "110",
                   '1' when "111",
                   '-' when others;
end rtl;
```

Nella seconda parte di questa dispensa vedremo ome genralizzare un parity encoder a parole di lunghezza variabile mediante l'uso di elementi sequenziali.

4.5.4 Multiplexer

Negli esempi mostrati, al segnale di uscita sono sempre state assegnate delle costanti. Questa circostanza tuttavia è un caso particolare. Un multiplexer, infatti, altro non è che un caso più generale di quanto appena visto in cui un insieme di segnali di selezione permettono di portare sull'uscita uno dei segnali di ingresso. Il simbolo comunemente utilizzato mostrato nella figura seguente per due multiplexer differenti: 2-a-1 e 4-a-1.



Il comportamento di un multiplexer 2-a-1 è riassunto dalla seguente tabella:

s	y
0	a
1	b

La tabella ha il seguente significato. Il segnale **s** è il segnale di selezione, **a** e **b** sono gli ingressi ed **y** è l'uscita. Il seganle **s** è di un bit, mentre sulla dimensione degli ingressi e dell'uscita non è stato specificato nulla. Supponiamo dapprima che anch'essi siano di un bit. Il VHDL che descrive un simile comportamento è molto semplice. Vediamo nzitutto la entity declaration:

```
entity mux_2_1 is
    port( s:    in  std_logic;
          a, b: in  std_logic;
          y:    out std_logic );
end mux_2_1;
```

Una possibile architecture utilizza il costrutto di assegnamento condizionato:

```
architecture rtl of mux_2_1 is
begin
    y <= a when s='0' else
         b when s='1' else
         '-';
end rtl;
```

Una soluzione alternativa sfrutta il costrutto **with** ... **select**, come mostra il codice seguente:

```
architecture rtl of mux_2_1 is
begin
    with s select
        y <= a   when '0',
             b   when '1',
             '-' when others;
end rtl;
```

Passiamo ora alla descrizione di un multiplexer 4-a-1. Il suo comportamento è il seguente:

s0	s1	y
0	0	a
0	1	b
1	0	c
1	1	d

Vediamo ora la descrizione VHDL di tale multiplexer 4-a-1.

```
entity mux_4_1 is
    port( s0, s1:     in  std_logic;
          a, b, c, d: in  std_logic;
          y:          out std_logic
    );
end mux_4_1;

architecture rtl of mux_4_1 is
begin
    y <= a when s0='0' and s1='0' else
         b when s0='0' and s1='1' else
         c when s0='1' and s1='0' else
         d when s0='1' and s1='1' else
         '-';
end rtl;
```

Questa soluzione ricalca la rappresentazione vista per le tabelle della verità con la sola differenza chei i valori assegnati al segnale di uscita **y** non sno costanti bensì segnali. Anche in questo caso il costrutto **with** ... **select** offre la possibilità di rendere la specifica più sintetica e leggibile, come mostra la seguente architecture.

```
architecture rtl of mux_4_1 is
    signal sel: std_logic_vector(0 to 1);
begin
    with sel select
        y <= a   when ”00”,
             b   when ”01”,
             c   when ”10”,
             d   when ”11”,
             ’-’ when others;
end rtl;
```

Supponiamo ora che i segnali di ingresso, e di conseguenza il segnale di uscita, non siano di un bit bensì siano segnali a 8 bit. La nuova entity per il multiplexer in esame diventa la seguente:

```
entity mux_4_1_8bit is
    port( s0, s1:     in  std_logic;
          a, b, c, d: in  std_logic_vector(0 to 7);
          y:          out std_logic_vector(0 to 7)
    );
end mux_4_1_8bit;
```

L’architecture rimane pressochè invariata. L’unica modifica necessaria è per l’assegnamento della costante (condizione di indifferenza) per la clausa **`when others`**, cioè:

```
architecture rtl of mux_4_1_8bit is
    signal sel: std_logic_vector(0 to 1);
begin
    with sel select
        y <= a when ”00”,
             b when ”01”,
             c when ”10”,
             d when ”11”,
             ”--------” when others;
end rtl;
```

Vediamo ora, a titolo di esempio, come è possibile specificare un multiplexer 4-a-1 generico per segnali di dimensione variabile. Come accennato in precedenza, la parametrizzazione di un componente si realizza mediante il costrutto **`generic`**. Vediamo la dichiarazione dell’entity:

```
entity mux_4_1_Nbit is
    generic( N: integer );
    port( sel:        in  std_logic_vector(0 to 1);
          a, b, c, d: in  std_logic_vector(0 to N-1);
          y:          out std_logic_vector(0 to N-1)
    );
end mux_4_1_Nbit;
```

La dichiarazione di generic introduce nell’entity un parametro, **`N`**, che indica la dimensione dei segnali di ingresso e del segnale di uscita. Quando questo componente sarà utilizzato nel progetto, sarà necessario assegnare un valore specifico e costante al parametro **`N`**. Ciò fatto, lo strumento di sintesi sarà in grado di realizzare il componente in modo consistente con la dimensione assegnata

ai segnali. Consideriamo ora l'architecture descritta poco sopra: si nota che l'unica parte che mostra una dipendenza esplicita dalla dimensone dei segnali è l'assegnamento del valore costante costituito da tutti don't care. Non conoscendo a priori la dimensione del segnale di uscita **y**, non è possibile utilizzare una costante predefinita per tale assegnamento. A tal fine il VHDL dispone di un costrutto che permette di assegnare un valore costante ad ogni elemento di un vettore, sia quando la dimensione è nota a priori, sia quando non lo è. La sintassi di tale costrutto, nel caso di un assegnamento, è la seguente:

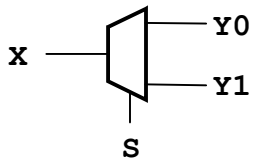
```
signal_name <= (others => const );
```

Questo sta a significare che ogni elemento del vettore ***signal_name*** prende il valore ***const***. Mediante tale costrutto è possibile riscrivere l'architecture del multiplexer generico in modo semplice e compatto, come mostrato di seguito.

```
architecture rtl of mux_4_1_Nbit is
begin
    with sel select
        y <= a when "00",
             b when "01",
             c when "10",
             d when "11",
             (others => '-') when others;
end rtl;
```

4.5.5 Demultiplexer

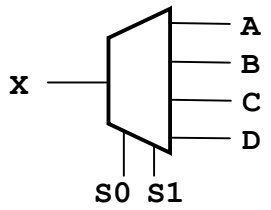
Un demultiplexer è un elemento che svolge la funzione duale del muliplexer. Nella sua versione più semplice, esso dispone di una linea di ingresso **x**, un ingresso di selezione **s** e due linee di uscita **y0** ed **y1**. Il simbolo utilizzato comunemente è il seguente:



Il demultiplexer pone in uscita il valore **x** su una delle due linee **y0** o **y1**, a seconda del valore del segnale di selezione mentre non è specificato il valore che assume l'altra linea. Il codice VHDL corrispondente, per un demultiplexer 1-a-2 a 1 bit è il seguente:

```
entity demux_1_2
    port( s:      in  std_logic;
          x:      in  std_logic;
          y0, y1: out std_logic );
end demux_1_2;

architecture rtl of demux_1_2 is
begin
    y0 <= x when s='0' else '-';
    y1 <= x when s='1' else '-';
end rtl;
```

L'estensione al demultiplexer 1-a-4 è semplice, così come lo è l'introduzuione di un generic che specifichi la dimensione delle linee di ingresso e uscita. Il simbolo è il seguente:

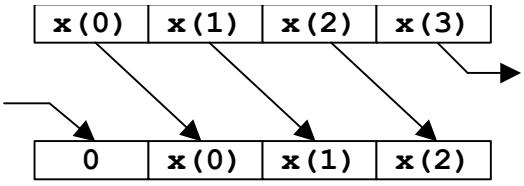


Il codice VHDL si può costruire estendendo il modello del demultiplexer e ricordando la tecnica dei generic per la parametrizzazione di un componente.

```
entity demux_1_4_Nbit
    generic( N: integer );
    port( sel:            in  std_logic_vector(;
          x:              in  std_logic_vector(0 to N-1);
          y0, y1, y2, y3: out std_logic_vector(0 to N-1)
    );
end demux_1_4_Nbit;

architecture rtl of demux_1_4_Nbit is
begin
    y0 <= x when sel="00" else (others => '-');
    y1 <= x when sel="01" else (others => '-');
    y2 <= x when sel="10" else (others => '-');
    y3 <= x when sel="11" else (others => '-');
end rtl;
```

4.5.6 Shifter

Uno shifter è un elemento che presenta all'uscita il valore dell'ingresso (vettoriale) spostato in una direzione prefissata di un certo numero di bit. Ad esempio un shifter a destra di una posizione per un segnale di 4 bit è un dispositivo che realizza la funzione descritta dalla seguente figura.



Dapprima vediamo l'entity declaration di tale dispositivo:

```
entity shift_right is
    port( x: in  std_logic_vector(0 to 3);
          y: out std_logic_vector(0 to 3) );
end shift_right;
```

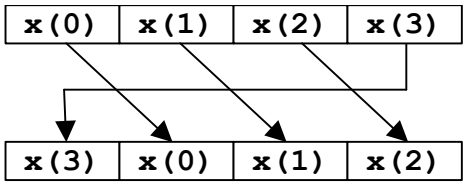
Una prima possible architettura è la seguente:

```
archietcture rtl of shift_right is
begin
    y(0) <= '0';
    y(1) <= x(0);
    y(2) <= x(1);
    y(3) <= x(2);
end rtl;
```

Ricordando il costrutto di slicing e l'operatore di concatenamento, possiamo fornire una specifica più compatta per la stessa funzionalità, ovvero:

```
archietcture rtl of shift_right is
begin
    y <= '0' & x(0 to 2);
end rtl;
```

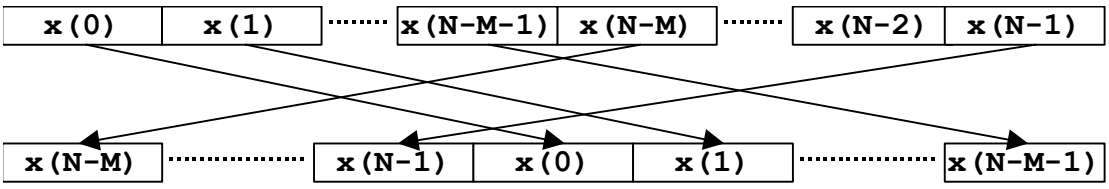
Un elemento simile è lo shifter circolare che realizza l'operazione mosrata dalla figura seguente:



La specifica VHDL, già data nella forma compatta facente uso di slice e concatenamento è:

```
entity shift_circular is
    port( x: in  std_logic_vector(0 to 3);
          y: out std_logic_vector(0 to 3)
    );
end shift_circular;

archietcture rtl of shift_circular is
begin
    y <= x(3) & x(0 to 2);
end rtl;
```

Mediante l'uso di generic è possibile realizzare, ad esempio, uno shifter circolare di un numero arbitrario di posizioni su dati di dimensione altrettanto arbitraria. Il parametro **N** indica la dimensione dei dati mentre il parametri **M** indica l'entità dello scorrimento.



L'entity declaration di unate componente è la seguente:

```
entity shift_circular_N_M is
    generic( N: integer;
             M: integer );
    port( x: in  std_logic_vector(0 to 3);
          y: out std_logic_vector(0 to 3) );
end shift_circular_N_M;
```

L'architettura ricalca lo schema mostrato dalla figura e, in kodo molto sintetico, si esprime in VHDL come:

```
archietcture rtl of shift_circular_N_M is
begin
    y <= x(N-M to N-1) & x(0 to N-M-1);
end rtl;
```

Si noti che questo componente non permette di effetture uno shift circolare programmabile mediante un segnale bensì una sorta di template di shifter circolare generico. Al momento dell'utilizzo di tale componente nella costruzione di un circuito più complesso, l'entità dello scorrimento deve essere fissata ad un valore costante.
Volendo realizzare uno shifter, ad esempio uno shifter a destra per vettori di 4 bit, controllabile tramite segnali si deve ricorrere ad una architettura più complessa. Il comportamento di un tale elemento potrebbe essere riassunto dalla tabella seguente, in cui s è un vettore di due elementi che indica l'entità dello scorrimento:

s	Scorrimento	y
00	0	x
01	1	'0' & x(0 to 2)
10	2	"00" & x(0 to 1)
11	3	"000" & x(0)

Ciò si traduce nella specifica seguente:

```
entity shift_right_prog is
    port( x: in  std_logic_vector(0 to 3);
          s: in  std_logic_vector(0 to 1);
          y: out std_logic(0 to 3) );
end shift_right_prog;

archiecture rtl of shif_right_prog is
begin
    with s select
        y <= x                when "00",
             '0' & x(0 to 2)  when "01",
             "00" & x(0 to 1) when "10",
             "000" & x(0)     when "11",
             "0000"           when others;
end rtl;
```

Questi semplici esempi mostrano come il VHDL offra la possibilità di specificare comportamenti complessi con estrema semplicità, sintesi e flessibilità.

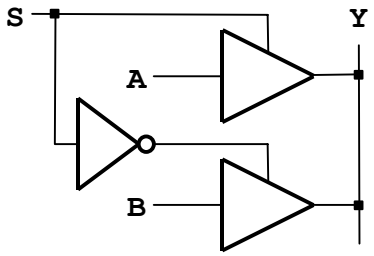
4.5.7 Buffer tri-state

Anche se di uso non molto frequente, i buffer tri-state possono essere utili in talune situazioni. Un buffer tristate è dotato di un ingresso dati **D**, di una uscita **Y** e di un segnale di enable **E**. Il suo comportamento è il seguente: se **E** vale 1 allora l'uscita **Y** è uguale a **D** altrimenti l'uscita è in alta impedenza. Alcuni buffer tri-state possono essere invertenti, in tal caso quando **E** vale 1 l'uscita **Y** è il complemento dell'ingresso **D**. I simboli con cui si indicano tali dispositivi sono i seguenti:

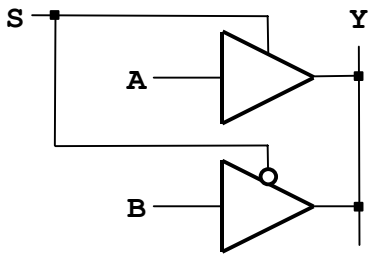
Ricordando che il valore **'Z'** del tipo **std_logic** indica la condizione di alta impedenza, la specifica di un buffer tri-state non invertente è la seguente:

```
entity tri_state is
    port( d: in  std_logic;
          e: in  std_logic;
          y: out std_logic );
end tri_state;

architecture rtl of tri_state is
begin
    y <= d when e='1' else 'Z';
end rtl;
```

Un esempio di applicazione del buffer tri-state è un multiplexer 2-a-1. Tale realizzazione sfrutta una connessione wired-or e due tri-state, come mostra la figura.



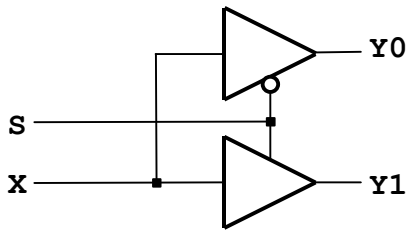
Si noti che a volte un buffer tri-state con segnale di enable attivo basso, come il buffer in basso della figura precedente si indica in forma compatta come mostrato nella figura seguente.



La specifica di un multiplexer realizzato in questo modo diviene quindi la seguente:

```
entity mux_2_1_tri_state is
    port( s:    in  std_logic;
          a, b: in  std_logic;
          y:    out std_logic
    );
end mux_2_1_tri_state;

architecture rtl of mux_2_1_tri_state is
begin
    y <= a when s='1' else 'Z';
    y <= b when s='0' else 'Z';
end rtl;
```

Come si può notare, il segnale di uscita **y** è assegnato due volte: ciò normalmente costituisce una violazione delle regole del VHDL secondo cui ogni segnale può avere uno ed un solo driver. Tuttavia, essendo i driver del segnale **y** dei buffer tri-state, la connessione in wired-or è accettata.
In maniera analoga, è molto intuitivo realizzare un demultiplexer mediante buffer tri-state. Mentre nella specifica vista in precedenza le linee di uscita non selezionate assumevano un valore di don't care, in questo caso esse assumono un valore di alta impedenza. La realizzazione circuitale che ne consegue è la seguente:



Ciò può essere tradotto in VHDL in modo chiaro e diretto:

```
entity demux_1_2_tri_state is
    port( s:      in  std_logic;
          x:      in  std_logic;
          y0, y1: out std_logic
    );
end demux_1_2_tri_state;

architecture rtl of demux_1_2_tri_state is
begin
    y0 <= x when s='0' else 'Z';
    y1 <= x when s='1' else 'Z';
end rtl;
```

4.6 Operatori aritmetici e relazionali

Come accennato nella sezione relativa alla libreria **IEEE**, i due tipi **std_logic_vector** e **std_ulogic_vector** possono essere utilizzati indifferentemente per rappresentare stringhe di bit e numeri interi in codifica binaria. Naturalmente una generica stringa di bit ammette almeno

due differenti interpretazioni del valore numerico corrispondente: quella naturale e quella in complemento a 2. Così, ad esempio, la stringa 1101 può essere interpretata come 13 (binario naturale) oppure come –3 (complmento a 2). Questa circostanza rende necessario esplicitare il tipo di codifica utilizzata ogni qualvolta si voglia usare un vettore di bit come numero intero.
La libreria **IEEE** definisce due package destinati a tale scopo: **std_logic_unsigned** e **std_logic_signed**. Si ricorda che tali package devono essere utilizzati in mutua esclusione e sempre accoppiati al package **std_logic_1164**. Così, volendo usare numeri senza segno si dovranno inserire nella specifica VHDL le seguenti dichiarazioni:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logc_unsigned.all;
```

mentre per l'uso di valori con segno si ricorrerà alla dichiarazione:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logc_signed.all;
```

Normalmente, l'interpretazione di stringhe di bit secondo una codifica numerica ha senso se e solo se su tali stringhe si vogliono eseguire operazioni aritmetiche. Così come i tipi **IEEE** non sono parte del linguaggio VHDL am ne costituiscono una estensione standardizzata, anche il supporto per la specifica e la sintesi di operatori aritmetici e relazionali è una estensione del linguaggio. Tutto ciò che è necessario a tal fine è raccolto nel package **std_logic_arith**, che, al'occorrenza deve essere esplicitamente richiesto mediante la dichiarazione:

```
use IEEE.std_logic_arith.all;
```

Questo package definisce l'interpretazione di alcuni operatori e stabilisce i corrispondenti template di sintesi in modo consistente. In particolare tutti gli strumenti di sintesi devono supportare i seguenti operatori:

Classe	Operatori
Aritmetici	**+**, **-**, *****
Relazionali	**>**, **>=**, **<**, **<=**, **=**, **/=**
Shift	**sll**, **srl**, **rol**, **ror**

Come primo esempio di utilizzo di tali operatori consideriamo la soma di due numeri naturali positivi **a** e **b** rappresentati su 8 bit. Per prima cosa si noti che la somma di due numeri di 8 bit deve essere rappresentasta su 9 bit, interpretabili tutti come risultato oppure come 8 bit bit di risultato più un bit (il più significativo) per la segnalazione dell'overflow.
Analizziamo dapprima il primo caso. L'entity declaration di un tale sommatore, completa di dichairazione delle librerie e dei package necessari, potrebbe essere quindi la seguente:

```
entity adder_8_8_9 is
    port( a, b: in  std_logic_vector(0 to 7);
          c:    out std_logic_vector(0 to 8)
    );
end adder_8_8_9;
```

Il linguaggio prevede che gli operandi ed il risultato di una somma abbiano tutti lo stesso numero di bit, mentre nell'entity declaration appena vista abbiamo indicato dimensioni differenti per motivi legati all'interpretazione del risultato. Nell'architecture, quindi, sarà necessario provvedere opportunamente al padding degli operandi, ottenuto aggiungendo uno zero nella posizione più significativa. Vediamo una possibile soluzione:

```
architecture rtl of adder_8_8_9 is
    signal tempa: std_logic_vector(0 to 8);
    signal tempb: std_logic_vector(0 to 8);
begin

    -- Explicit padding
    tempa <= '0' & a;
    tempb <= '0' & b;

    -- Addition
    c <= tempa + tempb;

end rtl;
```

Questa soluzione, benché corretta, richiede la definizione di due segnali temporanei. La soluzione seguente è altrettanto corretta ed ha il vantaggio di essere più sintetica:

```
architecture rtl of adder_8_8_9 is
begin
    c <= ('0' & a) + ('0' & b);
end rtl;
```

Vediamo ora come realizzare un sommatore più complesso, ovvero un sommatore dotato di due ingressi dati **a** e **b** di 8 bit, un ingresso di riporto **cin** ad 1 bit, un'uscita dati **s** a 8 bit ed un riporto in uscita **cout** ancora a 8 bit. L'operazine che si vuole realizzare è la seguente:

```
   0 a(0) a(1) a(2) a(3) a(4) a(5) a(6) a(7) +
   0 b(0) b(1) b(2) b(3) b(4) b(5) b(6) b(7) +
        0    0    0    0    0    0    0  cin =
-------------------------------------------------
cout s(0) s(1) s(2) s(3) s(4) s(5) s(6) s(7)
```

L'entity declaration è di facile scrittura:

```
entity adder_8_8_8_carry is
    port( a, b: in  std_logic_vector(0 to 7);
          cin:  in  std_logic;
          s:    out std_logic_vector(0 to 7);
          cout: out std_logic
    );
end adder_8_8_8_carry;
```

Per quanto concerne l'architecture, è necessario estendere su una lunghezza di 9 bit i due operandi ed il riporto, quindi sommarli e separare il riporto in uscita dal risultato della somma. Il codice seguente mostra una possibile soluzione.

```
architecture rtl of adder_8_8_8_carry is
    signal tempa:   std_logic_vector(0 to 8);
    signal tempb:   std_logic_vector(0 to 8);
    signal tempcin: std_logic_vector(0 to 8);
    signal temps:   std_logic_vector(0 to 8);
begin

    -- Extends operands
    tempa   <= '0' & a;
    tempb   <= '0' & a;
    tempcin <= "00000000" & cin;

    -- Addition
    temps <= tempa + tempb + tempcin;

    -- Splits the result
    cout <= temps(0);
    s    <= temps(1 to 8);

end rtl;
```

Anche in questo caso è possibile eliminare alcuni segnali ridondanti:

```
architecture rtl of adder_8_8_8_carry is
    signal temps:   std_logic_vector(0 to 8);
begin

    -- Addition
    temps <= ('0' & a) + ('0' & b) + ("00000000" & cin);

    -- Splits the result
    cout <= temps(0);
    s    <= temps(1 to 8);

end rtl;
```

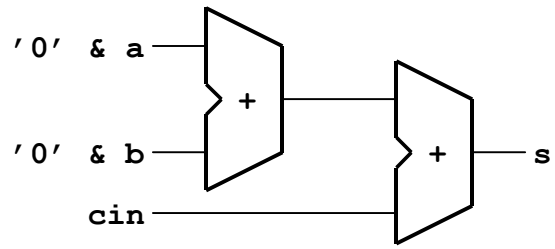
Questo schema è facilemente parametrizzabile sulla dimensione degli operandi. Ecco una soluzione possibile. Vediamo dapprima l'entity declaration, che, come appare evidente, segue lo schema già adottato per altri componenti parametrici.

```
entity adder_N_N_N_carry is
    generic( N: integer );
    port( a, b: in  std_logic_vector(0 to N-1);
          cin:  in  std_logic;
          s:    out std_logic_vector(0 to N-1);
          cout: out std_logic
    );
end adder_N_N_N_carry;
```

Consideriamo ora l'architecture. Si noti che l'estensione del riporto in ingresso sulla lunghezza di parola (**N**) non richiede l'uso di un segnale temporaneo in quanto il linguaggio – e lo strumento di sintesi – prevede l'estensone implicita alla dimansione massima tra i due operandi di una somma.

```
architecture rtl of adder_N_N_N_carry is
begin
    signal temps:   std_logic_vector(0 to N-1);
begin

    -- Addition
    temps <= ('0' & a) + ('0' & b) + cin;

    -- Splits the result
    cout <= temps(0);
    s    <= temps(1 to N-1);

end rtl;
```

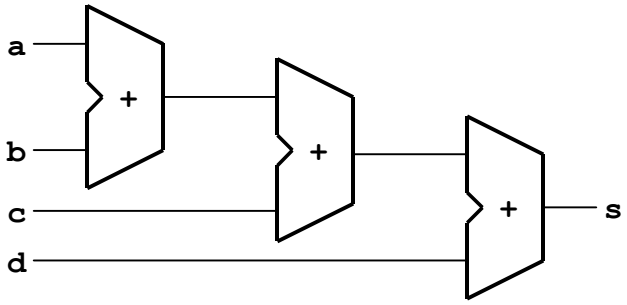
In quest'ultima architettura abbiamo visto che è possible sommare in un'unica espressione tre segnali. Questo risulta in una struttura come quella mostrata nella figura seguente:



Vediamo ora che cosa accade per una somma di quattro operandi, diaciamo a, b, c e d. Si consideri dapprima l'assegnamento:

```
s <= a + b + c + d;
```

L'architettura risultante è la seguente:



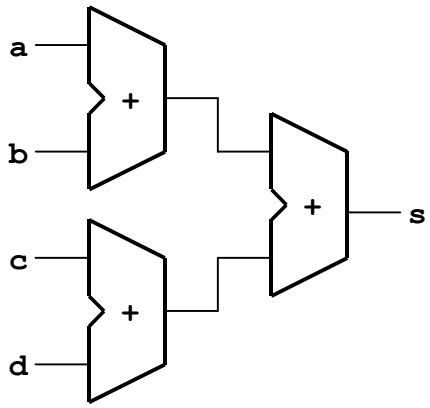
Questa struttura rispecchia l'associatività dell'operatore di somma, cioè equivale all'espressione:

```
s <= (((a + b) + c) + d);
```

Detto T il ritardo di propagazione del segnale attraverso un sommatore, la soluzione appena vista comporta un ritardo pari a 3T. Volendo ottimizzare le prestazioni si può forzare, mediante l'uso delle parentesi, un ordine differente di valutazione dell'espressione, cioè:

```
s <= (a + b) + (c + d);
```

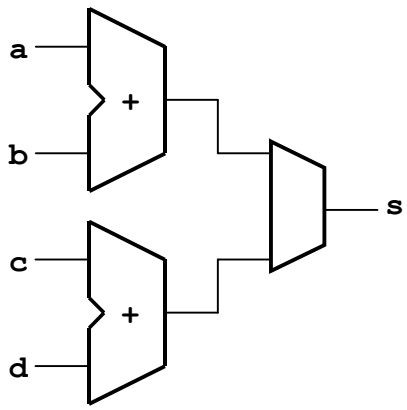
Da questa scrittura si perviene alla struttura ostrata di seguito, che, come appare chiaro, parallelizza due delle 3 somme e quindi presenta un ritardo complessivo pari a 2T.



Questo esempio mostra come sia possibile modificare, mediante un attento uso delle parentesi, il risultato della sintesi di espressioni algebricamente equivalenti.
Accenniamo ora al problema della condivisione degli operatori, noto anche come **resource sharing**. A tale scopo utilizziamo un semplice esempio. Si consideri un modulo **`add_sel`** dotato di 4 ingressi dati **`a`**, **`b`**, **`c`** e **`d`**, una uscita dati **`y`**, tutti ad 8 bit, ed un segnale di selezione **`s`**; se s vale 0 allora **`y=a+b`** altrimenti **`y=c+d`**. Vediamo una possibile architecture:

```
architecture rtl of add_sel is
begin
    y <= a + b when s='0' else c + d;
end rtl;
```

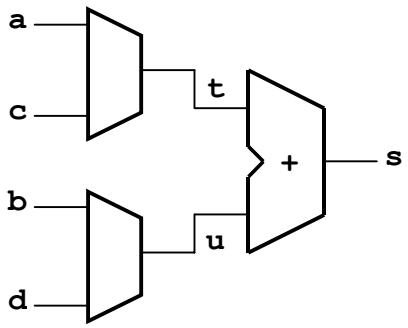
Il risultato della sintesi, benché comunque aderente alla specifica, dipende dale caratteristiche dello strumento di sintesi utilizzato. Una prima implementazione potrebbe essere la seguente:



Questa sluzione è corretta ma poco efficiente in termini di area occupata, ovvero in termini di numero di porte logiche necessarie. Una riformulazione del problema, e conseguentemente della specifica VHDL, porta ad una architettura più compatta ed ugualmente efficiente.

```
architecture rtl of add_sel is
    signal t: std_logic_vector(0 to 7);
    signal u: std_logic_vector(0 to 7);
begin
    t <= a when s='0' else c;
    u <= b when s='0' else d;
    y <= t + u;
end rtl;
```

In questo caso l'operazione di somma è una sola ed il problema si sposta sulla selezione degli operandi mediante multiplexer. Il vantaggio in termini di area deriva dalla complessità inferiore di un multiplexer rispetto ad un sommatore. Dal punto di vista circuitale questa specifica corrisponde alla struttura mostrata di seguito:



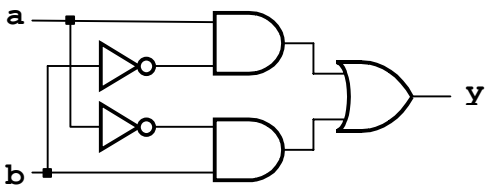
Come nota conclusiva, è bene notare che alcuni strumenti di sintesi particolarmente efficienti sono in grado di riconoscere situazioni di questo tipo anche se implicita, come nella scrittura originale, e pervenire a soluzioni ottimizzate come quella appena mostrata. Questo processo di ottimizzazione prende il nome di operator sharing.

4.7 Descrizione strutturale

Quanto visto fino a questo punto è sufficiente per la specifica di moduli di bassa o media complessità ma poco si adatta a circuiti di elevata complessità. Come in molti altri campi della scienza, un buon approccio alla soluzione di un problema complesso consiste nella scomposizione di questo in sottoproblemi più semplice e quindi nella ricomposizione delle singole soluzioni. In termini di progettazione questo significa che la progettazione di un circuito molto complesso deve essere affrontata individuando le funzionalità di base del sistema, progettando componenti in grado di realizzarle e quindi comporre i vari moduli, mediante opportune interconnessioni, fino a formare il sistema completo.
Per chiarire questo concetto, iniziamo da un semplice esempio. Si voglia realizzare una porta XOR utilizzando le porte fondamentali OR, AND e NOT. Sappiamo che:

```
a xor b = (a and (not b)) or ((not a) and b)
```

In termini circuitali, questo corrisponde alla struttura mostrata nella seguente figura:



Per realizzare la porta XOR parteno dai componenti di base è necessario per prima cosa realizzare tali componenti. Vediamo le entity e le architecture necessarie:

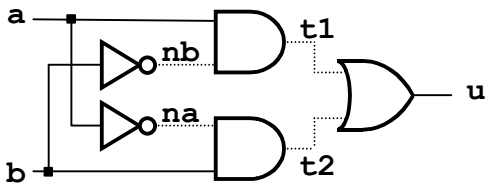
```
-- The NOT gate
entity NOT_GATE is
    port( x: in  std_logic;
          z: out std_logic );
end NOT_GATE;

architecture rtl of NOT_GATE is
begin
    z <= not x;
end rtl;

-- The 2-input AND gate
entity AND2_GATE is
    port( x: in  std_logic;
          y: in  std_logic;
          z: out std_logic );
end AND2_GATE;

architecture rtl of AND2_GATE is
begin
    z <= x and y;
end rtl;

-- The 2-input OR gate
entity OR2_GATE is
    port( x: in  std_logic;
          y: in  std_logic;
          z: out std_logic );
end OR2_GATE;

architecture rtl of OR2_GATE is
begin
    z <= x or y;
end rtl;
```

Ora che disponiamo dei tre elementi di base possiamo procedere alla soluzione del problema connettendo tali elementi in maniera opportune, cioè secondo lo schema circuitale mostrato precedentemente. È bene notare, a scanso di equivoci, che esiste una differenza sostanziale tra, ad esempio, l'operatore VHDL **and** e il componente **AND2_GATE** appena specificato: il primo è predefinito come appartenete al linguaggio e può unicamente essere utilizzato nelle espressioni, menre il secondo è un componente generico che non può essere utilizzato in alcuna espressione,

ma piuttosto deve essere **istanziato**. Istanziare un componente significa inserirlo nella specifica, e quindi nel circuito che si sta realizzando, e connetterlo opportunamente ad altri componenti mediante l'uso di segnali.
Iniziamo ora la specifica della porta XOR partendo dalla entity declaration.

```
entity XOR2_GATE is
    port( a: in  std_logic;
          b: in  std_logic;
          u: out std_logic );
end XOR2_GATE;
```

Ricordando che i segnali di ingresso e di uscita dichiarati in una entity declaration sono utilizzabili nella corrispondente architecture, analizzaiamo il circuito che realizza la porta XOR per individuare quali e quanti segnali sono necessari per le connessioni:



La figura mostra, tratteggiati, tali segnali: **`na`** ed **`nb`** sono le uscite delle porte NOT e servono per connettere queste agli ingressi delle due porte AND mentre i segnali **`t1`** e **`t2`** sono le uscite delle porte AND e servono per connettere queste agli ingressi della porta OR. Dato che tali segnali non sono né ingressi né uscite della porta XOR, si dice che si tratta di **segnali interni**. La parte dichiarativa della architecture dovrà specificarne nome e tipo. Prima di procedere alla scrittura della architecture ricordiamo un'altro aspetto del linguaggio VHDL: una architecture che utlizza altri componenti deve dichiararne il nome e l'interfaccia nella sua parte dichiarativa. La dichiarazione di un componente segue lo schema generale seguente:

```
component component_name is
    [generic( generic_list );]
    port( port_list );
end component;
```

Sostanzialmente è identico alla corrispondente entity declaration, salvo l'uso della parola chiave **`component`** al posto di **`entity`**. A questo punto possiamo iniziare a vedere il codice VHDL dell'architecture, iniziando dalla parte dichiarativa:

```
architeture structural of XOR2_GATE is

    -- The NOT gate
    component NOT_GATE is
        port( x: in  std_logic;
              z: out std_logic );
    end component;
```

```
    -- The 2-input AND gate
    component AND2_GATE is
        port( x: in  std_logic;
              y: in  std_logic;
              z: out std_logic );
    end component;

    -- The 2-input OR gate
    component OR2_GATE is
        port( x: in  std_logic;
              y: in  std_logic;
              z: out std_logic );
    end component;

    -- Internal signals
    signal na: std_logic;
    signal nb: std_logic;
    signal t1: std_logic;
    signal t2: std_logic;
begin
    ...
```

A questo punto abbiamo dichiarato tutto ciò che è necessario per procedere alla costruzione del circuito. Per fare ciò, come già accennato, è necessario istanziare i componenti. Una istanziazione ha la forma generale seguente:

```
instance_name: component_name
    [generic map( generic_assignment_list );]
    port map( port_assignment_list );
```

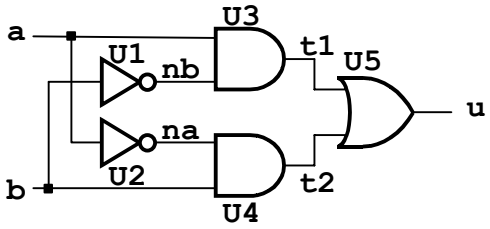
Ogni volta che si utilizza un componente è necessario assegnargli un nome: tale nome, detto **instance name**, deve essere unico all'interno dell'architecture e non ha alcun legame con il nome del componente che si sta istanziando. Il nome del componente di cui ***instance_name*** è un'istanza è ***component_name*** e deve essere stato preventivamente dichiarato nella parte diachiarativa dell'architecture. Trascuriamo per il momento la parte opzionale del costrutto di istanziazione che riguarda i generic e passiamo ad analizzare il costrutto **port map**. Tale costrutto ha lo scopo di indicare come le porte dell'istanza ***instance_name*** del componente ***component_name*** sono connesse ai segnali dell'architecture, siano essi ingressi/uscite o segnali interni. La port_assignment_list è una lista di tali connessioni e può assumere due forme: **positional** (posizionale) oppure **named** (nominale). La forma posizionale ha la seguente sintassi:

```
port map( signal_1, ..., signal_N );
```

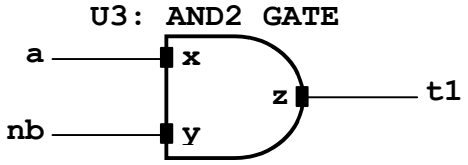
In tal caso si assume che l'ordine dei segnali sia significativo in quanto il primo segnale della lista sarà connesso alla prima porta che appare nella dichiarazione del component, il secondo segnale alla seconda porta e così via. Questa forma, sicuramente molto sintetica, può risultare a volte di difficile lettura. La seconda possibilità consiste nell'uso della forma nominale che segue la sintassi seguente:

```
port map( port_1 => signal_1, ..., port_N => signal_N );
```

in cui I nomi delle porte del componente che si sta istanziando appaiono nominati esplicitamente. Grazie a questo fatto l'associazione (ovvero la connessione dei segnali) avviene per nome e quindi la posizione all'interno della lista perde di significato e può pertanto essere arbitraria.
Riprendiamo ora lo schema circuitale della porta XOR aggiungendo i nomi che intendiamo assegnare alle istanzedei vari componenti.



Per chiarire il concetto di istanziazione e mostrare un esempio di entrambi gli approcci, consideriamo dettagliatamente l'istanziazione di una delle porte AND, ad esempio l'istanza **U3**. La figura che segue mostra tale istanza indicando il nome del component cui fa riferimento, i nomi delle porte dichiarate nel component ed il nome dei segnali che si intendono connettere.



Il codice VHDL per effettuare tale istanziazione secondo il secondo schema (nominale), è il seguente:

```
U3: AND2_GATE
    port map( x => a, y => nb, z => t1 );
```

Questo significa che la porta **x** del component **AND2_GATE** sarà connessa al segnale **a**, la porta **y** sarà connessa al segnale **nb** e la porta **z** sarà connessa al segnale **t1**. Come si nota, la direzione delle porte non ha alcuna influenza sulla sintassi dell'istanziazione. Per scrivere la stessa istanziazione nella prima forma (posizionale) è necessario ricordare che le porte del componente **AND2_GATE** sono state dichiarate nel seguente ordine: **x**, **y** e **z**. Avremo pertanto:

```
U3: AND2_GATE
    port map( a, nb, t1 );
```

Si noti infine che la direzionalità dei segnali di ingresso e uscita dell'architecture deve essere rispettata. Questo significa che un segnale di ingresso dell'architecture (ad esempio **a**) non può essere connesso ad una porta di uscita di uno dei componenti (poichè ciò equivarrebbe ad una scrittura di un segnale di ingresso, circostanza che il VHDL proibisce) e così un segnale di uscita dell'architecture (ad esempio **u**)non può essere connesso ad un ingresso di uno dei componenti.
A questo punto possiamo completare il corpo dell'architecture della porta XOR semplicemente elencando tutte le istanze che la compongono. Si noti che è possibile utilizzare all'interno della stessa architecture sia istanziazioni con port map posizionale sia istanziazioni con port map nominale. Vediamo ora il corpo completo dell'architecture.

```
...
begin
    U1: NOT_GATE
        port map( x => b, z => nb );
    U2: NOT_GATE
        port map( x => a, z => na );
    U3: AND2_GATE
        port map( x => a, y => nb, z => t1 );
    U4: AND2_GATE
        port map( x => na, y => b, z => t2 );
    U3: OR2_GATE
        port map( x => t1, y => t2, z => u );
end structural;
```

Questo stile di scrittura del VHDL prende il nome di **VHDL strutturale** in quanto pone l'accento sulla struttura della rete piuttosto che sulle trasformazioni che i segnali subiscono.
Nella pratica di progettazione, l'uso di VHDL strutturale si limita alla connessione di componenti complessi e solo molto raramente utilizza elementi quali porte logiche o flip-flop. È altresì comune utilizzare nella stessa architecture sia lo stile strutturale sia lo stile RTL, in modo da sfruttare i vantaggi di entrambi. Il VHDL così ottenuto non è né strutturale né RTL ma una miscela dei due.
A titolo di esempio suponiamo di voler realizzare la stessa porta XOR sfruttando come componenti solo le porte AND (**`AND2_GATE`**) ed OR (**`OR2_GATE`**) ricorrendo allo stile RTL per costruire i segnali **`na`** ed **`nb`** mediante loperatore VHDL **`not`**. La parte dichiarativa dell'archietcture sarà in questo caso:

```
architeture mixed of XOR2_GATE is

    -- The 2-input AND gate
    component AND2_GATE is
        port( x: in  std_logic;
              y: in  std_logic;
              z: out std_logic );
    end component;

    -- The 2-input OR gate
    component OR2_GATE is
        port( x: in  std_logic;
              y: in  std_logic;
              z: out std_logic );
    end component;

    -- Internal signals
    signal na: std_logic;
    signal nb: std_logic;
    signal t1: std_logic;
    signal t2: std_logic;

begin
    ...
```

Mantenedo invariati i nomi delle istanze delle porte AND e OR, il corpo dell'architecture diviene:

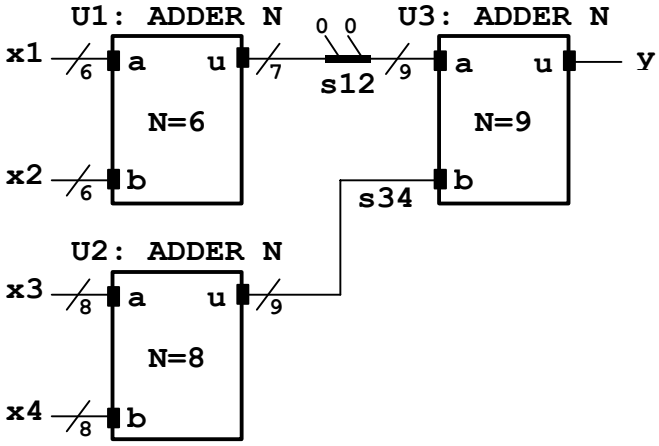
```
...
begin
    -- RTL style
    na <= not a;
    nb <= not b;

    -- Structural style
    U3: AND2_GATE
        port map( x => a, y => nb, z => t1 );
    U4: AND2_GATE
        port map( x => na, y => b, z => t2 );
    U3: OR2_GATE
        port map( x => t1, y => t2, z => u );
end mixed;
```

Affrontiamo ora il problema dei generic. Anche in questo caso usiamo un semplice esempio per chiarire i diversi aspetti. Si voglia realizzare un sommatore di 4 valori **x1**, **x2**, **x3** e **x4** interi e senza segno. Di questi **x1** ed **x2** sono rappresentati su 6 bit mentre **x3** ed **x4** sono rappresentati su 8 bit. Il risultato **z** deve essere rappresentato su 10 bit per garantire che non si verifichino overflow. Per fare ciò si vuole procedere specificando dapprima una sommatore generico a due ingressi e quindi utilizzare tale componente per realizzare il sommatore completo dei quattro valori. Iniziamo dalla specifica del sommatore generico:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity ADDER_N is
    -- Input signal size
    generic( N: integer );
    -- Ports
    port( a: in  std_logic_vector(0 to N-1); -- First input
          b: in  std_logic_vector(0 to N-1); -- Second input
          u: out std_logic_vector(0 to N) ); -- Output
end ADDER_N

architecture rtl of ADDER_N is
begin

    -- Extends input operands and adds
    u <= ('0' & a) + ('0' & b);

end rtl;
```

Analizziamo ora il problema iniziale per determinare quail caratteristiche dovranno avere I sommatori necessari e quali segnali interni saranno richiesti. Per la somma di **x1** ed **x2** è necessario un sommatore a 6 bit che produrrà il risultato **s12** su 7 bit. Per la somma di **x3** ed **x4** è necessario un seommatore su 8 bit che produrrà il risultato **s34** su 9 bit. Infine per la somma

dei due risultati parziali **s12** ed **s34** è necessario per prima cosa estendere **s12** su 9 bit quindi utilizzare un sommatore a 9 bit che produrrà il risultato voluto su 10 bit. Questa soluzione è schematizzata nella figura seguente in cui sono evidenziati i segnali interni, le loro dimensioni ed i nomi delle istanze.



5. Reti sequenziali

In questo capitolo affronteremo il problema della descrizione delle reti sequenziali. Per poter introdurre i processi, che stanno alla base della specifica di comportamenti seuqnziali, è necessario chiarire il concetto su cui si fonda il modello computazionale del linguaggio VHDL: il concetto di evento. Consideriamo, ad esempio una porta AND ai cui ingressi sono presenti un uno ed uno 0 e, di conseguenza, alla cui uscita si ha uno zero. Se gli ingressi non mutano allora nemmeno l'uscita cambierà valore. Tuttavia appena si ha una transizione su uno dei segnali di ingresso, il valore dell'uscita potrebbe cambiare. Se l'ingresso che si trova a zero passasse al valore uno, allora si avrebbe una transizione sull'uscita della porta da zero a uno. Ciò, nella realtà fisica, accade perchè la transizione in ingresso corrisponde ad un cambiamento della tensione sul gate di uno dei MOSFET della porta AND che ha come conseguenza una variazione dello stato di polarizzazione del MOSFET il quale, entrando in conduzione, porta l'uscita a massa, ovvero al valore logico uno. Il modello di calcolo del VHDL deve poter rappresentare una simile situazione in cui in assenza di variazione di segnale agli ingressi di una porta il sistema permane in uno stato stabile mentre in presenza di una transizione il sistema deve poter reagire modificando il valore di opportuni segnali. Usando una terminologia mutuata dall'ambito della simulazione di specifiche VHDL si dice che il comportamento che una descrizione VHDL coglie è conttollato dagli eventi o **event-driven**.
Consideriamo, ad esempio, una ipotetica porzione di architecture composta dai seguenti statement concorrenti:

```
x <= a or z;
y <= c and (not x);
z <= b and d;
```

Benché l'ordine in cui sono scritti questi statement no rispecchi le dipendenze tra i segnali, la corretta valutazione è garantita dal meccanismo degli eventi. Vediamo per quale motivo. Supponiamo che il valore del segnale **b** passi da 0 ad 1: ciò significa che sul segnale **b** si è verificato un evento. Nella realtà fisica tale transizione corrisponderebbe ad una variazione del valore di tensione in un punto della rete e comportrebbe una propagazione di tale variazione all'interno della rete. Il modo in cui ciò è reso dal VHDL segue una logica analoga. In corrispondenza dell'evento sul segnale **b** è infatti necessario ricalcolare il valore di tutte le espressioni che utilizzano **b** nella parte destra. Nel caso in esame deve essere ricalcolato il valore di **z**. Se il valore di **z** dovesse cambiare (supponendo ad esempio che **d** valga 1) ciò produrrebbe un nuovo evento, questa volta sul segnale **z**. Come conseguenza, sarebbe necessario ricalcolare tutte le espressioni in cui **z** compare nella parte destra, ovvero, continuando con l'esempio, l'espressione che calcola il valore del segnale **x**. Se anche **x** cambiasse di valore, sarebbe necessario analizzare, in modo simile a quelllo appena visto, tutte le espressioni al fine di propagare l'evento. Si noti che nel modello di calcolo del VHDL tutto ciò avviene in un tempo nullo, spesso indicato con il termine di **delta-time**. Se il primo evento su **b** si verifica ad un certo tempo **t**, allora la corrispondente variazione del segnale **z** avviene al tempo $t+\Delta t$. Tale variazione si propaga al segnale **x** al tempo $t+2\Delta t$ ed infine su **y** al tempo $t+3\Delta t$. Dal momento che l'intervallo di tempo fittizio Δt ha durata nulla, si ha una propagazione istantanea della variazione del segnale b sul segnale **y**.
Ritornando al concetto di evento possiamo dire che uno statement VHDL è attivato da un evento su uno dei segnali che lo statement legge. L'attivazione di uno statement potrebbe a sua volta causare un evento ed attivare nuovi statement. Tutto ciò rispecchia in modo corretto il comportamento delle reti combinatorie ideali. Una volta chiarite queste idee, possiamo passare ad introdurre i processi.

5.1 Processi

Un processo è uno statement concorrente, al pari di un assegnamento, un assegnamento condizionato o una istanziazione. Un processo, tuttavia, è anche uno statement composto, costituito, tra le altre code da un corpo, a sua volta composto da statement. Una prima importantissima questione a tale riguardo è che gli statement all'interno di un processo non sono concorrenti besì sequenziali. Un primo utlizzo di un processo potrebbe quindi essere quello di forzare una inerpretazione sequenziale del codice VHDL del suo corpo.
Prima di procedere oltre, vediamo la sintassi generale per la dichiarazione di un processo:

```
[process_name]: process ( sensitivity_list )
    [declarations]
begin
    [body]
end process;
```

Un processo ha un nome opzionale ***process_name***, utile prevalentemente per motivi di leggibilità, una parte dichiarativa opzionale ***declaration***, simile a quella dell'architecture declaration ed un corpo ***body***, anch'esso teoricamente opzionale (un processo privo di corpo è accettabile in VHDL ma è privo di qualsiasi scopo). Infine un processo ha una **sensitivity list** ovvero una lista di segnali in grado di attivarlo: tutte le volte che si verifica un evento, e soltanto in queso caso, su uno dei segnali della ***sensitivity_list*** il processo è attivato altrimenti esso rimane inattivo. Per chiarire questo concetto consideriamo uno statement concorrente come:

```
z <= (x + y) or t;
```

Questo statement sarà attivato, come abbiamo visto, da un evento su almeno uno dei segnali **x**, **y** oppure **t**. Analogamente, se volessimo rendere un processo attivabile da tali segnali, ovvero dalgi eventi che si dovessero verificare su tali segnali dovremmo inserire **x**, **y** e **t** nella sensitivity list, come mostra la seguente porzione di codice:

```
sample: process ( x, y, t )
begin
    …
```

Consideriamo ora il seguente processo e studiamone il significato ed il comportamento:

```
sample: process ( a, b )
begin
    x <= a or b;
    y <= x or c;
end;
```

Dal momento che la sensitivity list contempla solo i segnali **a** e **b**, un evento sul segnale **c** non produrrebbe l'attivazione del processo con la conseguenza che il valore del segnale **y** non verrebbe aggiornato se non in corrispondenza di un evento su **a** o **b**. Una tale specifica è pertanto scorretta in quanto non rispecchia lo schema di calcolo previsto per il segnale **y**.
Chiarito il principio che sta alla base del meccanismo di attivazione dei processi, vediamo quali sono gli statement sequenziali di cui il VHDL dispone.

5.2 Statement sequenziali

In costrutti sequenziali che possono essere utilizzati nel corpo di un processo sono quelli tipici di ogni linguaggio di programmazione: espressioni ed assegnamenti, costrutti condizionali e cicli. Questi ultimi, tuttavia, pongono una serie di problemi rispetto alla sintesi ed appartengono allo stile di specifica behavioural, di cui non parliamo in questa breve introduzione.

5.2.1 Statement `if-then-else`

Il costrutto condizionale di base del linguaggio VHDL è lo statement if. Nella forma più generale esso ha la seguente sintassi:

```
if condition_1 then
    statement_1
[elsif condition_2 then
    statement_2]
...
[else
    statement_N]
end if;
```

Come si nota, sia il ramo **eslsif** sia il ramo **else** sono opzionali. Inoltre il numero di rami **elsif** può essere arbitrario. A tale proposito è importante sottolineare e ricordare un concetto già introdotto parlando degli assegnamenti condizionali. Anche per il costrutto **if** vale infatti la regola per cui tutti i possibili casi devono essere esplicitamente considerati. Questo, molto spesso si riduce all'obbligo di avere il ramo **else**. Quindi, escluse le eccezioni che vedremo nel seguito, la forma minima corretta di tale costrutto è la seguente:

```
if condition then
    statement_then
else
    statement_else
end if;
```

Il significato è quello desumibile dai più comuni linguaggi di programmazione: se la condizione **condition** è vera l'insieme degli statement **statement_then** sarà abilitato, mentre se la condizione è falsa saranno gli statement **statement_else** ad essere abilitati. Consideriamo ora un primo, semplice esempio. Supponiamo di voler realizzare un multiplexer a due ingressi **a** e **b** ed indichiamo con **u** l'uscita e con **s** il segnale di selezione di tale elemento. Abbiamo visto che lo statement concorrente di assegnamento condizionato è adatto a tale scopo, precisamente:

```
architecture rtl_concurrent of mux is
begin
    u <= a when s='0' else b;
end rtl_concurrent;
```

Questa specifica non richiede ulteriori commenti in quanto ampiamente studiata in precedenza. Volendo realizzare lo stesso multiplexer mediante l'uso di processi e statement sequenziali, si procederà come segue. Pe prima cosa è necessario individuare i segnali ai quali vogliamo che il processo sia sensibile: nel nostro caso una variazione su uno qualsiasi dei segnali **a**, **b** ed **s** produrrà un cambiamento dell'uscita, quindi i tre segnali dovranno apparire nella sesitivity list del processo. La condizione in questo caso è molto semplice e consiste nel confronto di **s** con il

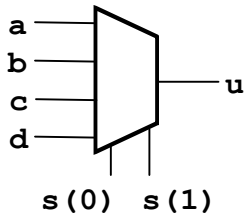
valore costante 0 (avremmo anche potuto effettuare il confronto con il valore, ricordando di scambiare il ramo then con il ramo else). In conclusione possiamo scrivere quanto segue:

```
architecture rtl_sequential of mux is
begin
    select: process( a, b, s )
    begin
        if( s = '0' ) then
            u <= a;
        else
            u <= b;
    end process;
end rtl_sequential;
```

Risulta chiaro che una tale specifica è poco conveniente in termini di compattezza, benché conduca comunque alla sintesi di un multiplexer. Consideriamo ora un esempio leggermente più complesso e cioè un multiplexer a 4 ingressi in cui il segnale di selezione s è dunque un vettore di due bit. Secondo lo schema di specifica concorrente si scriverebbe (tralasciando alcuni dettagli ovvi):

```
with s select
    u <= a when "00",
         b when "01",
         c when "10",
         d when "11",
         'X' when others;
```

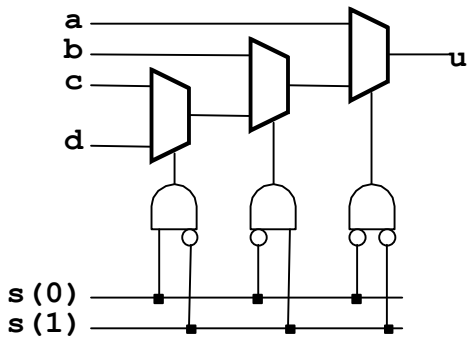
Questa scrittura porta alla sintesi di una rete del tipo:



In termini di statement sequenziali, una soluzione potrebbe essere la seguente:

```
if( s = "00" ) then
    u <= a;
elsif( s = "01" ) then
    u <= b;
elsif( s = "10" ) then
    u <= c;
elsif( s = "11" ) then
    u <= d;
else
    u <='X';
end if;
```

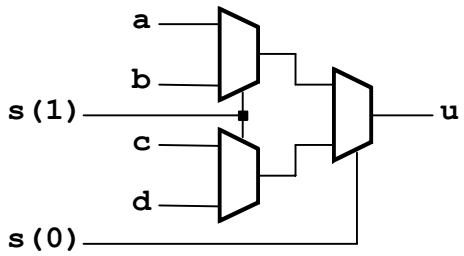
Questa scrittura, tuttavia, produce un risultato diverso da quello che ci si potrebbe aspettare per il fatto che lo statement sequenziale if impone un ordine di valutazione delle condizioni. La struttura che ne deriva è quindi la seguente:



Stabilito che la funzione svolta da tale rete è corretta ed equivalente a quella del circuito ottenuto dalla sintesi della specifica comportamentale, si ha un differenza in termini di temporizzazione. In particolare notiamo che se **s="00"** il segnale di ingresso **a** attraversa un solo multiplexer per propagarsi fino all'uscita (quindi 2 livelli di logica) mentre se **s="10"** oppure **s="11"** i segnali **c** o **d** devono attraversare tre multiplexer ovvero 6 livelli di logica. Questo problema di temporizzazione a volte deve essere esplicitamente considerato e deve essere il criterio che guida il progettista verso uno stile di specifica parallelo oppure sequenziale.
Una soluzione ancora differente potrebbe essere la seguente:

```
if( s(0) ='0' ) then
    if( s(1) = '0' ) then
        u <= a;
    else
        u <= b;
    end if;
else
    if( s(1) = '0' ) then
        u <= c;
    else
        u <= d;
    end if;
end if;
```

Questo stile di specifica conduce ad una rete ancora diversa, e cioè:



L'utilizzo del costrutto if, quindi, offre la possibilità di un notevole controllo sull'architettura che si intende realizzare.

5.2.2 Statement `case`

Molto spesso accade di avere la necessità di confrontare un segnale con una serie di costanti e eseguire operazioni subordinatamente all'esito di tale confronto. Come risulta chiaro, è possibile realizzare una tale rete mediante l'uso del costrutto **`if-then-elsif-else`** esprimendo le diverse condizioni nei diversi rami. Secondo quanto appena discusso, tuttavia, una tale struttura è intrinsecamente sequenziale e comporta sia l'introduzione di una priorità per i vari costrutti, sia lo sbilanciamento della rete dal punto di vista dei ritardi di propagazione delle diverse linee. Infine, una struttura molto complessa di if, eventualmente annidati, risulta poco leggibile. Per ovviare a tutti questi problemi si ricorre allo statement **`case`**. Tale costrutto è l'equivalente sequenziale del costrutto concorrente **`with-select`**, dotato di una maggiore flessibilità. La sintassi generale del costrutto **`case`** è la seguente:

```
case signal_name is
    when range_1 =>
        statement_1
    when range_2 =>
        statement_2
    ....
    when range_N =>
        statement_N
    when others =>
        statement_default
end case;
```

Il significato è deducibile facendo riferimento o al costrutto **`case`** di molti linguaggi di programmazione, oppure ricordando la sintassi ed il significato del costrutto **`with-select`**.
Ad esempio, il multiplexer 4-a-1 di un esempio precedente potrebbe essere specificato così:

```
case sel is
    when "00" =>
        u <= a;
    when "01" =>
        u <= b;
    when "10" =>
        u <= c;
    when "11" =>
        u <= d;
    when others =>
        u <= 'X';
end case;
```

In questo semplice esempio abbiamo usato come condizioni delle costanti. Le clausole **`when`** del costrutto **`case`**, tuttavia, permettono di verificare condizioni più complesse in forma compatta. La sintassi generale di una condizione è la seguente:

```
values [ | values] ...
```

in cui ***values*** può essere o una costante oppure un range sia per variabili di tipo **integer**, sia per tipi enumerativi definiti dall'utente. Ad esempio, il seguente case statement è corretto:

```
signal x: integer range 0 to 15;

case x is
    when 0 =>
        z <= "00";
    when 1 | 3 =>
        z <= "01";
    when 4 to 8 | 2 =>
        z <= "10";
    when 9 to 15 =>
        z <= "1";
    when others =>
        z <= "XX";
end case;
```

Quando le condizioni dei diversi rami sono espresse mediante range potrebbe accadere che alcuni dei range siano parzialmente sovrapposti. Questo fatto costituisce una violazione delle regole di sintesi e pertanto è un errore che deve essere evitato.

5.2.3 Statement `for`

Lo statement for è utilizzato per realizzare cicli. Nella specifica a livello RTL il costrutto for deve sottostare ad alcuni vincoli che vdremo tra breve. La sua sintassi generale è la seguente:

```
for identifier in range loop
    statements
end loop;
```

L'identificatore identifier è la variabile di controllo del ciclo che ad ogni iterazione assume i valori appartenenti all'intervallo specificato da range. Si noti che identifier non è un segnale e pertanto non può essere usato in nessun costrutto che richiede come operando un segnale. L'intervallo di variazione di tale indice, cioè range, deve essere un intervallo costante ovvero tale per cui gli estremi siano noti al momento della sintesi. L'intervallo può assumere una delle seguenti tre forme:

```
first to last
last downto first
array_name'range
```

Le prime due forme hanno lo stesso significato già visto per la specifica delle dimensioni di un array e per la descrizione delle slice. La terza forma sfrutta l'attributo **range** dei segnali VHDL. Se ***array_name*** è il nome di un segnale di tipo vettoriale allora l'attributo **range** applicato a tale segnale ritorna l'intervallo di variazione degli indici del vettore stesso. Così, ad esempio, avendo dichiarato il segnale **datain** come:

```
signal datain: std_logic_vector(1 to 8);
```

l'espressione **datain'range** ritorna il range espresso come **1 to 8**. In questo modo è possibile accedere a tutti gli elementi di un array senza necessariamente conoscerne a priori le dimensioni. Infine, dato che il costrutto **for** è consentito solo all'interno di un process, il suo corpo, indicato con statements nella sintassi mostrata, è necessariamente composto da una lista di statement sequenziali.
Vediamo un semplice esempio di utilizzo del costrutto for per l'assegnamento di un segnale vettoriale **b** di 16 bit ad un segnale vettoriale **a** della stessa dimensione, svolto bit a bit.

```
signal a, b: std_logic_vector(0 to 15);
...
for I in 0 to 15 loop
    a(I) <= b(I);
end loop;
```

Se la dimensione dei segnali in esame fosse, ad esempio dipendente da un generic N, potremmo effetture lo stesso assegnamento mediante la scrittura:

```
signal a, b: std_logic_vector(0 to N-1);
...
for I in 0 to N-1 loop
    a(I) <= b(I);
end loop;
```

Un'altrenativa per la specifica del range di variazione dell'indice potrebbe anche essere:

```
signal a, b: std_logic_vector(0 to N-1);
...
for I in b'range loop
    a(I) <= b(I);
end loop;
```

Sull'indice del ciclo è consentito svolgere operazioni aritmetiche, purchè sia rispettata la condizione di calcolabilità al momento della sintesi. Ad esempio, per copiare il vettore b di 16 bit nel vettore a invertendo l'ordinamento dei bit possiamo scrivere:

```
signal a, b: std_logic_vector(0 to 15);
...
for I in 0 to 15 loop
    a(I) <= b(15-I);
end loop;
```

Vediamo ora come realizzare uno shifter puramente combinatorio in grado di effettuare lo scorrimento di M bit verso destra di una parola di lunghezza pari ad N bit. L'entity declaration è:

```
entiry shift_right_N_M is
    generic( N: integer;
             M: integer );
    port( data_in:  in  std_logic_vector(0 to N-1);
          data_out: out std_logic_vector(0 to N-1) );
end shift_right_N_M;
```

L'architecture può essere realizzata in modo semplice mediante l'uso di due cicli for: uno per l'assegnamento degli M bit più significativi, tutti costantemente uguali a zero ed uno per l'assegnamento dei restanti bit, opportunamente traslati. L'architecture è quindi:

```
architecture RTL of shift_right_N_M is
begin

    process( data_in )
    begin
        -- The most significant M bits
        for I in 0 to M-1 loop
            data_out(I) <= '0';
        end loop;

        -- The least significant N-M bits
        for I in M to N-1 loop
            data_out(I) <= data_in(I-M);
        end loop;
    end process;

end RTL;
```

Come mostrano tutti gli esempi presentati, i valori delle espressioni in cui compare l'indice del ciclo I sono determinabili in fase di sintesi, o perché si tratta di costanti oppure perché si tratta di generic il cui valore deve essere necessariamente assegnato epr procedere alla sintesi. Si noti infine che il VHDL impone che gli statement del corpo del ciclo non modifichino il valore dell'indice. In altre parole, la variabile di iterazione può soltanto essere letta.

5.3 Latch

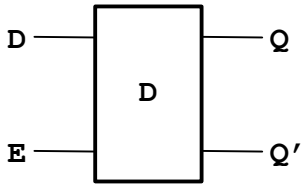
Un latch è il più semplice elemento di memoria che si possa utilizzare nella realizzazione di circuiti digitali. Un latch è dotato di un segnale di controllo, generalmente indicato con il nome **`enable`** o **`E`** e di uno o due ingressi di controllo. Quando il seganle di enable è attivo (alto o basso, a seconda della convenzione scelta dal progettista, una qualsiasi variazione degli ingressi produce una variazione sull'uscita: in questo caso si dice che il latch è trasparente. Qunado enable non è attivo, il latch non risponde alle variazioni sugli ingressi e pertanto mantiene memorizzato l'ultimo valore assegnato. I vari latch differiscono per la tipologia dei segnali di controllo. Vediamo di seguito i diversi dispositivi, le loro caratteristiche e le loro specifiche in VHDL.

5.3.1 Latch D

Il comportamento di un latch D è specificato dalla seguente tabella:

E	D	Q*
0	-	Q
1	0	0
1	1	1

Ciò può essere espresso a parole dicendo che quando **`E`** vale 0 il latch è opaco e mantiene l'ultimo valore memorizzato mentre quando **`E`** vale 1 il latch è trasparente e l'uscita segue l'ingresso. Questo comportamento può essere reso in modo chiaro mediante il costrutto process con una opportuna sensitivity list. Il simbolo utilizzato comunemente per tale dispositivo è il seguente:



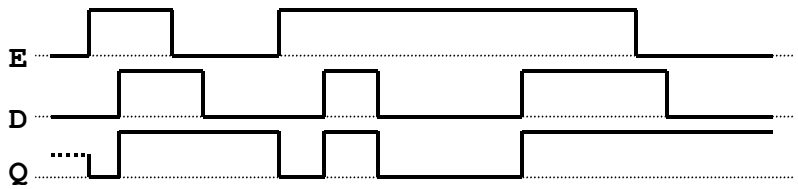
Descriviamo ora l'entity del latch.

```
entity D_LATCH is
    port( D: in  std_logic;
          E: in  std_logic;
          Q: out std_logic
    );
end D_LATCH;
```

Passiamo ora ad analizzare il comportamento. Per prima cosa notiamo che il valore dell'uscita del latch può cambiare in corrispondenza di un evento sia sul segnale **D** sia sul segnale **E**. Pertanto entrambi i segnali devono essere menzionati nella sensitivity list. Inoltre notiamo che l'uscita varia solo quando il segnale di enable è alto, mentre in caso contario il segnale di uscita non deve essere aggiornato. L'architecture che ne consegue è la seguente:

```
architetcture rtl of D_LATCH is
begin
    latch: process( D, E )
    begin
        if( E = '1' ) then
            Q <= D;
        end if;
    end process;
end rtl;
```

La prima cosa che si può notare è l'assenza del ramo else del costrutto if: questo, come abbiamo ampiamente discusso, è un errore quando si vuole specificare un comportamento puramente combinatorio. Il motivo di tutto ciò ammette la seguente spiegazione intuitiva: non essendo esplicitamente specificato cosa deve succedere quando il valore di **E** è diverso da 1, lo strumento di sintesi assume che i segnali che il processo scrive (cioè quelli che compaiono a sinistra di un simbolo di assegnamento) non varino il loro valore il che implica che deve essere presente un elemento di memoria in grado di conservare appunto tale valore.
Si noti inoltre che in tutti gli intervalli di tempo in cui il segnale di enable è alto, il valore dell'uscita **Q** segue il valore dell'ingresso **D**: ciò è garantito dal fatto che **D** è presente nella sensitivity list e quindi ogni evento su tale segnale è in grado di attivare il processo. Il seguente diagramma temporale mostra il comportamento del latch D.

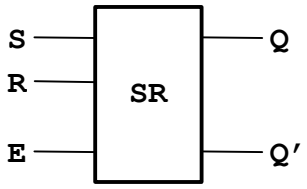


5.3.2 Latch SR

In maniera analoga si procede per la descrizione del latch SR. Il comportamento che vogliamo ottenere è il seguente:

E	S	R	Q*
0	-	-	Q
1	0	0	Q
1	0	1	0
1	1	0	1
1	1	1	-

Ricordiamo che il comportamento del latch SR non è definito quando entrambi i segnali di controllo valgono 1. Il simbolo usato per rappresentare un latch SR è il seguente:



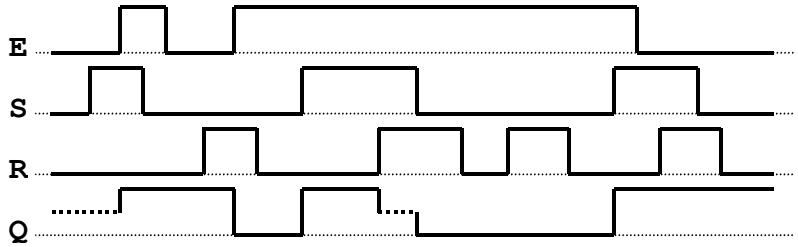
L'entity declaration per il latch SR è la seguente:

```
entity SR_LATCH is
    port( S: in  std_logic;
          R: in  std_logic;
          E: in  std_logic;
          Q: out std_logic
    );
end SR_LATCH;
```

Il comportamento può essere descritto in diversi modi: una possibilità è la seguente.

```
architetcture rtl of SR_LATCH is
begin
    latch: process( S, R, E )
    begin
        if( E = '1' ) then
            if( S = '1' and R = '0' ) then
                Q <= '1';
            elsif( S = '0' and R = '1' ) then
                Q <= '0';
            elsif( S = '0' and R = '0' ) then
                null;
            elsif( S = '1' and R = '1' ) then
                Q <= 'X';
            end if;
        end if;
    end process;
end rtl;
```

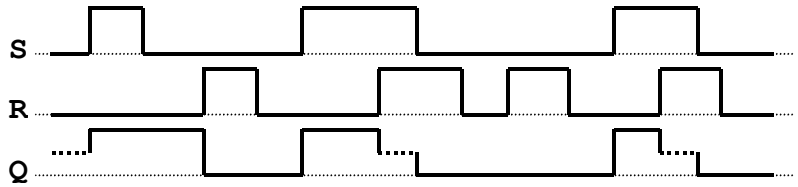
Un primo commento riguarda la sensitivity list: essa contiene tutti i segnali di ingresso quindi il processo è attivato da un evento su ognuno di essi. Il cordpo del processo, tuttavia, contiene un **`if`** principale che verifica il valore del segnale di enable. Se tale segnale è al livello logico 1, allora il comportamento del latch dipende dal valore dei segnali di controllo **`S`** ed **`R`** ed è immediatatmente sensibile ad ogni evento su di essi, altrimenti il latch mantiene il suo stato.
È importante sottolineare un'altra caratteristica importante della specifica. Quando entrambi i segnali di controllo hanno valore zero, il latch, pur essendo abilitato, deve mantenere il proprio stato: per indicare questo comportamento si utilizza lo statement VHDL **`null`**, che indica appunto un'azione nulla. Questo comportamento è esemplificato dal seguente diagramma temporale.



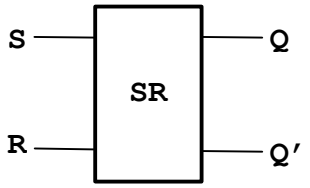
A differenza del latch D, un latch SR può essere privo del segnale di enable. In questo caso il suo comportamento è determinato unicamente dai segnali di controllo. La tabella della verità che descrive questo tipo di elemento è la seguente:

S	R	Q*
0	0	Q
0	1	0
1	0	1
1	1	-

Un esempio di diagramma temporale può aiutare a chiarire il funzionamento del latch in esame.



Il simbolo utilizzato nelle rappresentazioni circuitali è simile a quello visto, ma privo del segnale di enable in ingresso.



Conseguentemente l'entity declaration e l'architecture corrispondente sono:

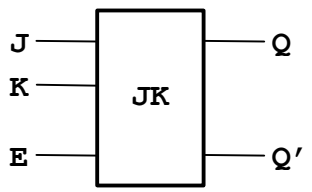
```
entity SR_LATCH_NO_ENABLE is
    port( S: in  std_logic;
          R: in  std_logic;
          Q: out std_logic
    );
end SR_LATCH_NO_ENABLE;

architetcture rtl of SR_LATCH_NO_ENABLE is
begin

    latch: process( S, R )
    begin
        if( S = '1' and R = '0' ) then
            Q <= '1';
        elsif( S = '0' and R = '1' ) then
            Q <= '0';
        elsif( S = '0' and R = '0' ) then
            null;
        elsif( S = '1' and R = '1' ) then
            Q <= 'X';
        end if;
    end process;

end rtl;
```

5.3.3 Latch JK

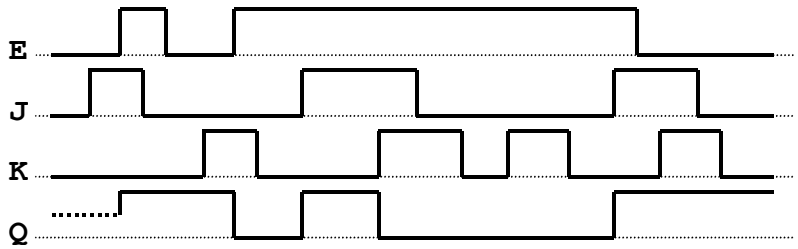
Un latch JK ha un comportamento simile ad un latch SR salvo il fatto che risolve la condizione di indeterminazione data dagli ingressi **S=1** e **R=1**. In tale latch, infatti quando entrambi i segnali di ccontrollo **J** e **K** valgono 1, l'uscita assume il valore dello stato corrente complementato. Riassumendo, il comportamento di un latch JK con segnale di enable è il seguente:

E	J	K	Q*
0	-	-	Q
1	0	0	Q
1	0	1	0
1	1	0	1
1	1	1	Q'

Il simbolo utilizzato nelle rappresentazioni circuitali è:



Un esempio di funzionamento è rappresentato dal seguente diagramma temporale.



La specifica del lach JK ricalca quella già vista per il latch SR rimuovendo il caso indeterminato in cui entrambi gli ingressi di controllo valgono 1. L'entity declaration non pone alcun problema.

```
entity JK_LATCH is
    port( J: in  std_logic;
          K: in  std_logic;
          E: in  std_logic;
          Q: out std_logic
    );
end JK_LATCH;
```

Per quanto riguarda l'architecture, ricordando la specifica data dalla tabella della verità, si nota che sarà necessario un assegnamento del tipo:

```
Q <= not Q;
```

Tuttavia **Q** è una porta di tipo **out** e pertanto non può essere letta. Questa circostanza impone l'utilizzo di un segnale temporaneo, ad esempio **QINTERNAL**, per la gestione dello stato. Tale segnale può quindi essere assegnato alla porta di uscita **Q** che in tal modo viene solo scritta, come richiede il VHDL. L'architecture seguente mostra una tale realizzazione.

```
architetcture rtl of JK_LATCH is
    signal QINTERNAL: std_logic;
begin
    latch: process( J, K, E )
    begin
        if( E = '1' ) then
            if( J = '1' and K = '0' ) then
                QINTERNAL <= '1';
            elsif( J = '0' and K = '1' ) then
                QINTERNAL <= '0';
            elsif( J = '0' and K = '0' ) then
                null;
            elsif( J = '1' and K = '1' ) then
                QINTERNAL <= not QINTERNAL;
            end if;
        end if;
    end process;

    Q <= QINTERNAL;
end rtl;
```

Il problema della lettura di un segnale di uscita si ripresenterà, come vedremo, nella specifica alcuni registri e contatori.

5.4 Flip-flop

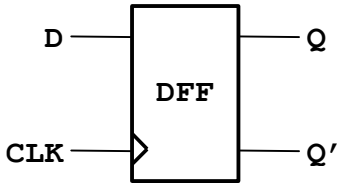
Un flip-flop è un dispositivo sequenziale in grado di mantenere memorizzato un bit. A differenza di un latch, un flip-flop è sensibile ad un fronte del segnale di sincronizzazione piuttosto che al suo livello. Il segnale di sicronizzazione di un flip-flop prende solitamente il nome di **clock** ed è un segnale periodico.

5.4.1 Flip-flop D

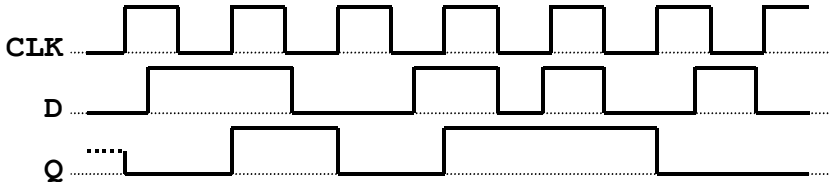
Un flip-flop D è descritto dalla tabella della verità che segue. Si noti che il egnale di clock non è esplicitamente considerato in quanto si assume che le transizioni sul valore dello stato **Q*** possano avvenire solo in corrispondenza di un fronte del clock.

D	Q*
0	0
1	1

Il simbolo utilizzato nelle rappresentazioni circuitali è:



Un esempio di funzionamento è riportato nel seguente diagramma temporale.



Iniziamo ora con la descrizione dell'entity del flip-flop D.

```
entity D_FF is
    port( CLK: in  std_logic;
          D:   in  std_logic;
          Q:   out std_logic
    );
end D_FF;
```

Passiamo ora alla specifica dell'architecture. Nel caso dei latch la condizione cui l'aggiornamento dell'uscita era subordinata riguardava il livello di un segnale (il segnale di enable) e come tale poteva essere espressa semplicemente confrontando il valore dl segnale con il livello di interesse (0 oppure 1). Nel caso dei flip-flop la condizione riulta più complessa in quanto è necessario

esprimere il verificarsi una transizione. Per fissare le idee, supponiamo che il dipositivo che vogliamo realizzare sia sensibile al fronte di salita. In tal caso dobbiamo verificare le due seguenti condizioni:

1. Si è verificata una transizione sul segnale di clock (evento)
2. Il valore del segnale dopo la transizione è 1

Per quanto riguarda la seconda condizione non si hanno problemi. Per esprimere il verificarsi di un evento sul segnale è necessario ricorrere ad un costrutto particolare bassato sul concetto di attributo VHDL. Nel caso specifico dovremo verificare che l'attributo **`event`** del segnale di clock sia vero. La condizione complessiva, quindi, si esprime come segue

```
if( CLK'event and CLK = '1' ) then
    ...
end if;
```

Dato che in assenza di un fronte di salita sul segnale di clock il dispositivo deve mantenere il proprio stato, anche in questo caso il ramo **`else`** dell'**`if`** dovrà essere omesso. È chiaro infine che il segnale dati **`D`** non dovrà apparire nella sensitivity list del processo in quanto lo stato del flip-flop non è influenzato da un evento su tale segnale. Basandoci su quanto appena visto possiamo procedere alla specifica dell'architecture.

```
architecture rtl_rising_edge of D_FF is
begin

    ff: process( CLK )
    begin
        if( CLK'event and CLK = '1' ) then
            Q <= D;
        end if;
    end process;

end rtl_rising_edge;
```

Volendo descrivere un flip-flop D sensibile al fronte di discesa, è sufficiente verificare che il valore del segnale di clock dopo l'evento sia 0 anziché 1, ovvero:

```
architecture rtl_falling_edge of D_FF is
begin

    ff: process( CLK )
    begin
        if( CLK'event and CLK = '0' ) then
            Q <= D;
        end if;
    end process;

end rtl_falling_edge;
```

In molti casi (ad esempio nella realizzazione di machine a stati) sono utili flip-flop dotati di un segnale di reset, sincrono o asincrono. Il segnale di reset sincrono forza il valore 0 sul flip-flop

quando vale 1 (se attivo alto) solo in corrispondenza di un evento sul clock, mentre è ignorato in caso contrario. Questo significa che per realizzare tale comportamento in VHDL il valore del segnale di reset deve essere testato solo quando si è verificato un evento sul clock e che quindi un evento sul segnale di reset non deve attivare il processo. Una possibile specifica è la seguente:

```
entity D_FF_SYNC_RESET is
    port( CLK:   in  std_logic;
          RESET: in  std_logic;
          D:     in  std_logic;
          Q:     out std_logic
    );
end D_FF_SYNC_RESET;

architecture rtl of D_FF_SYNC_RESET is
begin

    ff: process( CLK )
    begin
        if( CLK'event and CLK = '0' ) then
            if( RESET = '1' ) then
                Q <= '0';
            else
                Q <= D;
            end if;
        end if;
    end process;

end rtl;
```

A volte può essere utile anche un segnale di preset, cioè un segnale di controllo indipendente che ha lo scopo di forzare lo stato del flip-flop ad 1. Dato che utilizzeremo un costrutto if-elsif, dobbiamo stabilire la priorità che devono avere i due segnali di reset e preset. Scegliamo, ad esempio che il segnale di reset abbia una priorità maggiore del segnale di preset. In tal caso il comportamento del flip-flop sarà il seguente:

RESET	PRESET	D	Q*
1	-	-	0
0	1	-	1
0	0	0	0
0	0	1	1

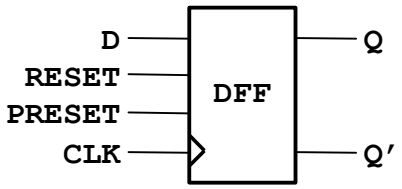
L'entity declaration è una estensione della precedente:

```
entity D_FF_SYNC_RESET_PRESET is
    port( CLK:    in  std_logic;
          RESET:  in  std_logic;
          PRESET: in  std_logic;
          D:      in  std_logic;
          Q:      out std_logic
    );
end D_FF_SYNC_RESET_PRESET;
```

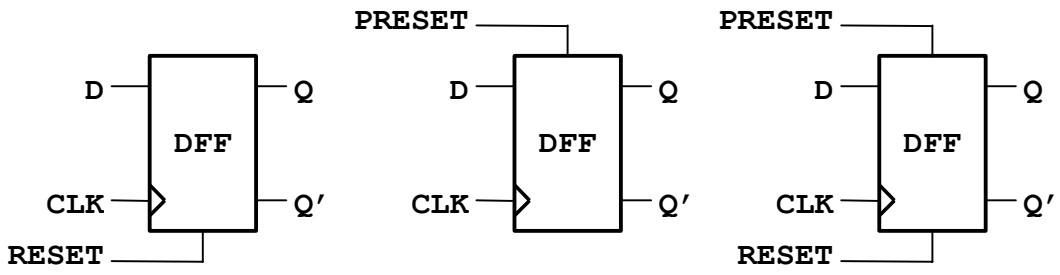
Per realizzare la priorità stabilta è necessario testare dapprima il valore del segnale di reset, quindi se questo è 0, il segnale di reset. Tutto ciò, trattandiso di segnali sincroni, deve essere fatto solo in corrispondenza di un fronte del clock. Ecco l'architecture che specifica un tale comportamento:

```
architecture rtl of D_FF_SYNC_RESET_PRESET is
begin

    ff: process( CLK )
    begin
        if( CLK'event and CLK = '0' ) then
            if( RESET = '1' ) then
                Q <= '0';
            elsif( PRESET = '1' ) then
                Q <= '1';
            else
                Q <= D;
            end if;
        end if;
    end process;

end rtl;
```

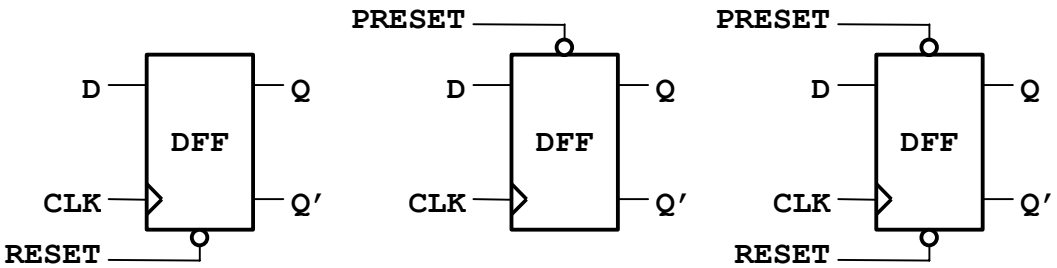
Il simbolo comunemente utilizzato per un tale flip-flop è il seguente:



Vediamo ora come realizzare un flip-flop D dotato di un segnale di reset asincrono. Un reset asincrono forza il valore 0 in uscita, indipendentemente dal verificarsi o meno di un evento sul clock. La risposta del flip-flop ad un evento sul reset asincrono è quindi immediata: ciò significa che il processo VHDL che implementa tale dispositivo deve essere sensibile anche al segnale di reset. Inoltre, dato che il reset asincrono produce un cambiamento nello stato indipendentemente dal verificarsi di un evento sul clock, il suo valore andrà verificato prima di analizzare il clock. I simboli comunemente usati per i dispositivi con segnali di reset/preset asincroni ed attivi alti sono i seguenti:



Quando gli stessi segnali sono attivi bassi si utilizzano invece i simboli seguenti:



Vediamo ora la specifica completa, iniziando, come di consueto, dall'entity declaration:

```
entity D_FF_ASYNC_RESET is
    port( CLK:   in  std_logic;
          RESET: in  std_logic;
          D:     in  std_logic;
          Q:     out std_logic
    );
end D_FF_SYNC_RESET;
```

La corrispondente architecture, codificata secondo le considerazioni appena svolte è la seguente:

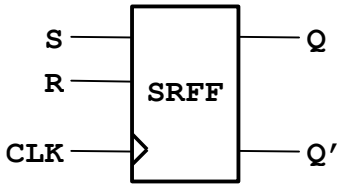
```
architecture rtl of D_FF_ASYNC_RESET is
begin

    ff: process( CLK )
    begin
        if( RESET = '1' ) then
            Q <= '0';
        else
            if( CLK'event and CLK = '0' ) then
                Q <= D;
            end if;
        end if;
    end process;

end rtl;
```

5.4.2 Flip-flop SR

Il flip-flop SR dispone di due segnali di controllo ed ha un comportamento molto simile al latch SR già visto. L'unica differenza consiste nel modo in cui lo stato viene aggiornato: nel latch SR ciò avviene durante tutto il periodo in cui il segnale di enable è attivo mentre nel flip-flop l'aggiornamento avviene solo in corrispondenza di un fronte del clock. Il comportamento è quindi descritto dalla seguente tabella della verità:

S	R	Q*
0	0	Q
0	1	0
1	0	1
1	1	-

Il simbolo utilizzato nelle rappresentazioni circuitali è mostrato di seguito.



La specifica segue lo schema di quella già vista per il flip-flop D per quanto riguarda la sensibilità al fronte di clock e quella del latch SR per quanto concerne l'effetto dei segnali di controllo.

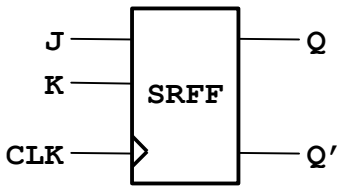
```
entity SR_FF is
    port( CLK: in  std_logic;
          S:   in  std_logic;
          R:   in  std_logic;
          Q:   out std_logic
    );
end SR_FF;

architecture rtl of SR_FF is
begin

    ff: process( CLK )
    begin
        if( CLK'event and CLK = '0' ) then
            if( S = '1' and R = '0' ) then
                Q <= '1';
            elsif( S = '0' and R = '1' ) then
                Q <= '0';
            elsif( S = '0' and R = '0' ) then
                null;
            elsif( S = '1' and R = '1' ) then
                Q <= 'X';
            end if;
        end if;
    end process;

end rtl;
```

Anche a questo elemento è possibile aggiungere segnali di reset e/o preset sia sincroni che asincroni seguendo i principi già visti per il flip-flop D e ricalcando gli schemi di implementazione già presentati.

5.4.3 Flip-flop JK

Per la realizzazione del flip-flop JK adottiamo lo stesso schema visto per il flip-flop SR. La tabella della verità ed il simbolo circuitale sono riportati di seguito.

J	K	Q*
0	0	Q
0	1	0
1	0	1
1	1	Q'



La specifica VHDL non pone particolari problemi. È solo opportuno ricordare che è necessario introdurre un segnale locale temporaneo per mantere lo stato del flip-flop in quanto il segnale Q, essendo una porta di uscita (quindi di tipo out) può soltanto essere scritta. Per maggiori dettagli si veda la descrizione del latch JK data in precedenza. Vediamo ora la specifica VHDL.

```
entity JK_FF is
    port( CLK: in  std_logic;
          J:   in  std_logic;
          K:   in  std_logic;
          Q:   out std_logic
    );
end JK_FF;

architetcture rtl of JK_FF is
    signal QINTERNAL: std_logic;
begin

    ff: process( CLK )
    begin
        if( CLK'event and CLK = '1' ) then
            if( J = '1' and K = '0' ) then
                QINTERNAL <= '1';
            elsif( J = '0' and K = '1' ) then
                QINTERNAL <= '0';
            elsif( J = '0' and K = '0' ) then
                null;
            elsif( J = '1' and K = '1' ) then
                QINTERNAL <= not QINTERNAL;
            end if;
        end if;
    end process;

    Q <= QINTERNAL;

end rtl;
```

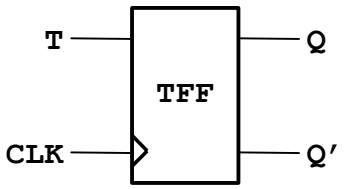
5.4.4 Flip-flop T

L'ultimo tipo di flip-flop di cui ci occuperemo è il flip-flop T, detto comunemente toggle. Esso dispone di un ingrsso di controllo, **T** appunto, che, se alto, provoca la commutazione dello stato, altrimenti mantiene il valore precedente. È bene notare che tale elemento non dispone di alcun ingresso di controllo in grado di forzare un valore specifico dello stato e ciò significa che, nella sua forma più semplice, tale flip-flop non può essere portato in uno stato iniziale noto. Questa circostanza rende il flip-flop toggle spesso inutilizzabile se privo di un segnale di reset o di preset

sincrono o asincrono. Approfondiremo questo problema tra poco.Vediamo dapprima il comportamento del flip-flop mediante la sua tabella della verità:

T	Q*
0	Q
1	Q'

Il simbolo circuitale è analogo ai precedenti:

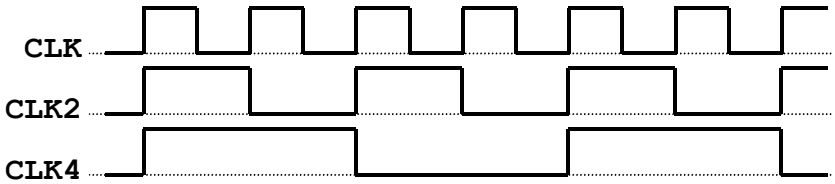


La specifica di questo elemento, analogamente a quella del flip-flop JK, richiede l'uso di un segnale locale per la memorizzazione dello stato. Il codice VHDL è quindi il seguente.

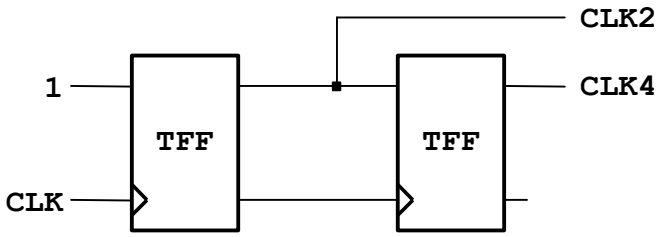
```
entity T_FF is
    port( CLK: in  std_logic;
          T:   in  std_logic;
          Q:   out std_logic
    );
end T_FF;

architetcture rtl of T_FF is
    signal QINTERNAL: std_logic;
begin

    ff: process( CLK )
    begin
        if( CLK'event and CLK = '1' ) then
            if( T = '1' ) then
                QINTERNAL <= not QINTERNAL;
            end if;
        end if;
    end process;

    Q <= QINTERNAL;

end rtl;
```

Il dispositivo specificato in questo modo non dispone di un segnale per forzare lo stato ad un valore preciso ma può sltanto commutare oppure mantenere il valore corrente. In un dato istante, perciò, non è possibile forzare un valore specifico. Questo è spesso un problema, tuttavia, in alcuni casi il valore iniziale è ininfluente. Vediamo una applicazione tipica del flip-flop T: il divisore di clock. Un divisore di clock è un dispositivo che riceve un clock con frequenza f in ingresso e fornisce in uscita un segnale di clock con frequenza f/k. Se k è una potenza del due, il divisore assume una forma particolarmente semplice. Supponiamo di voler realizzare un circuito che riceve in ingresso un clock **CKL** ad una frquenza f (arbitraria) e fornisce in uscita due nuovi

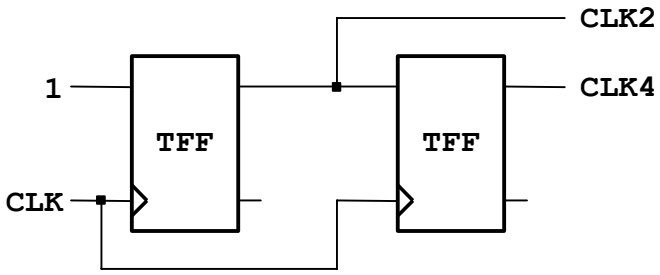
segnali di clock **CLK2** e **CLK4** rispettivamente a frequenza f/2 ed f/4. Il diagramma temporale seguente mostra l'andamento dei segnali **CLK**, **CLK2** e **CLK4**.



Per realizzare questo comportamento si può ricorrere ad esempio alla seguente soluzione:



Tale scelta, tuttavia, non è corretta in quanto i due flip-flop non sono sincroni poichè alimentati da due segnali di clock diversi. Per realizzare un divisore sincrono è necessario che i due flip-flop siano alimentati dallo stesso clock. Una soluzione potrebbe essere quindi la seguente:



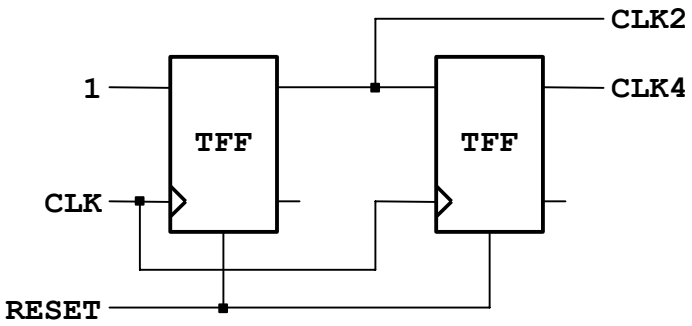
Passimo quindi alla stesura del codice VHDL del dispositivo. L'entity declaration è la seguente:

```
entity CLK_DIV_2_4 is
    port( CLK:  in  std_logic;
          CLK2: out std_logic;
          CLK4: out std_logic
    );
end CLK_DIV_2_4;
```

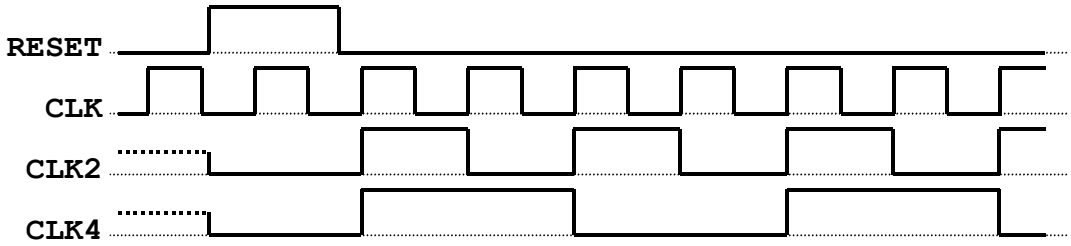
Per descrivere l'architecture procediamo come già visto. Il comportamento di un flip-flop T richiede l'uso di un segnale temporaneo per mantenere lo stato, quindi il divisore nell'insieme utilizzerà due segnali temporanei **CLK2T** e **CLK4T**, uno per ogni flip-flop.

```
architetcture rtl of CLK_DIV_2_4 is
    signal CLK2T: std_logic;
    signal CLK4T: std_logic;
begin
    -- Divides CLK by two
    div2_1: process( CLK )
    begin
        if( CLK'event and CLK = '1' ) then
            -- Don't test T because is always 1
            CLK2T <= not CLK2T;
        end if;
    end process;

    -- Divides CLK2 by two, i.e. CLK by four
    div2_2: process( CLK )
    begin
        if( CLK'event and CLK = '1' ) then
            -- The toggle signal is CLK2T
            if( CLK2T = '1' ) then
                CLK4T <= not CLK4T;
            end if;
        end if;
    end process;

    -- Assigns output signals
    CLK2 <= CLK2T;
    CLK4 <= CLK4T;
end rtl;
```

Dal codice e dalla definizione del comportamento del flip-flop T risulta chiaro che il valore dello stato non può che essere indeterminato in quanto non esiste alcun segnale di controllo forzante. Nella realtà fisica, il valore dello stato nel momento in cui il circuito viene alimentato sarà o 1 o 0 benché non è possibile determinare quale dei due valori. Indipendentemente da ciò, il circuito funzionerà correttamente. Questo però pone un problema nel momento in cui volessimo simulare il comportamento di tale rete: il simulatore, non potendo stabilire il valore iniziale dello stato dei flip-flop assegnerà ad essi il valore simbolico X che, come conseguenza si propagherà attraverso i flip-flop e produrrà per ogni segnale un valore indeterminato. Questo fatto impedisce di verificare mediante simulazione il comportamento del circuito. Inoltre, e soprattutto, una regola di buona progettazione impone che ogni flip-flop di un circuito sia dotato almeno del segnale di reset. Per questo motivo la rete migliore per la realizzazione del divisore è la seguente:



L’andamento dei segnali per un tale dispositivo è mostrato dal seguente diagramma temporale.



L’introduzione del segnale di reset porta quindi alla seguente specifica:

```
entity CLK_DIV_2_4_RESET is
    port( CLK:   in  std_logic;
          RESET: in  std_logic;
          CLK2:  out std_logic;
          CLK4:  out std_logic
    );
end CLK_DIV_2_4_RESET;

architetcture rtl of CLK_DIV_2_4_RESET is
    signal CLK2T: std_logic;
    signal CLK4T: std_logic;
begin
    -- Divides CLK by two
    div2_1: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            CLK2T <= '0';
        elsif( CLK'event and CLK = '1' ) then
            -- Don't test T because is always 1
            CLK2T <= not CLK2T;
        end if;
    end process;
    -- Divides CLK2 by two, i.e. CLK by four
    div2_2: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            CLK2T <= '0';
        elsif( CLK'event and CLK = '1' ) then
            -- The toggle signal is CLK2T
            if( CLK2T = '1' ) then
                CLK4T <= not CLK4T;
            end if;
        end if;
    end process;
    -- Assigns output signals
    CLK2 <= CLK2T;
    CLK4 <= CLK4T;
end rtl;
```

Ora vediamo un modo per realizzare la stessa specifica in maniera molto più compatta. Questo è un caso particolare di pipeline, come vedremo nell'ultima parte di questo capitolo. L'idea sta nel raggruppare in un unico processo la descrizione del comportamento dei due flip-flop, come mostra la seguente architecture:

```
architetcture rtl of CLK_DIV_2_4_RESET is
    signal CLK2T: std_logic;
    signal CLK4T: std_logic;
begin

    -- Divides CLK by two and by four
    div24: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            -- Resets both signals (flip-flops)
            CLK2T <= '0';
            CLK4T <= '0';
        elsif( CLK'event and CLK = '1' ) then

            -- First flip-flop: Toggle is always '1'
            CLK2T <= not CLK2T;

            -- Second: flop-flop: Toggle is CLK2T
            if( CLK2T = '1' ) then
                CLK4T <= not CLK4T;
            end if;

        end if;
    end process;

    -- Assigns output signals
    CLK2 <= CLK2T;
    CLK4 <= CLK4T;

end rtl;
```

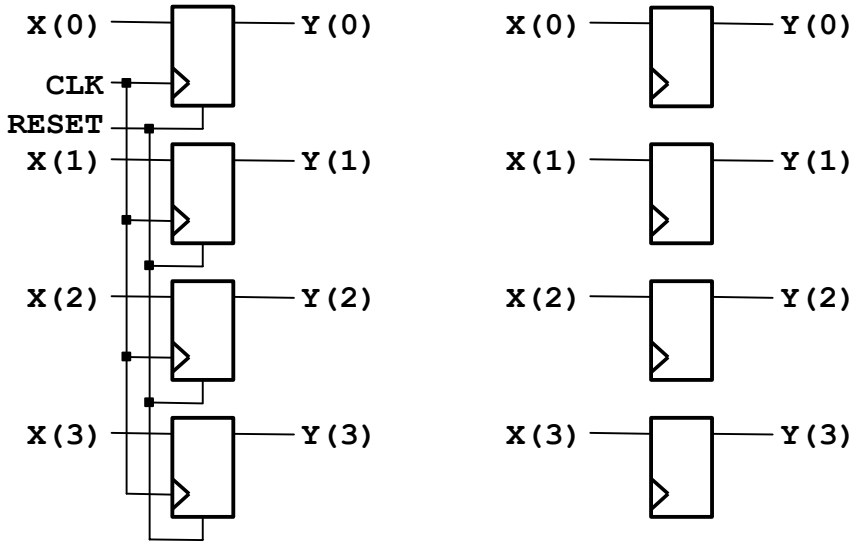
Con questo semplice esempio concludiamo la presentazione degli elementi sequenziali di base per passare ad un nuovo argomento: i registri.

6. Registri

Nell'accezione più comune un registro è un insieme di flip-flop, molto spesso di tipo D, destinato a memorizzare non un singolo bit, bensì parole di dimensione maggiore. Esistono poi molti registri che svolgono anche operazioni di manipolazione dei dati quali scorrimento, serializzazione, parallelizzazione ecc. Nel seguito vediamo come realizzare in VHDL alcuni dei registri più utilizzati, commentando gli apsetti di progettazione più rilevanti.

6.1 Registro parallelo-parallelo

Un registro parallelo-parallelo è quello cui ci si riferisce normalemente quando si parla semplicemente di registro. Esso è costituito da un insieme (banco) di flip-flop D, tutti sincronizzati dallo stesso segnale di clock, in cui le linee dati di ingresso sono connesse in parallelo agli ingressi dati dei flip-flop e le uscite dati sono le uscite degli stessi flip-flop. Un registro parallelo-parallelo a 4 bit ha pertanto una struttura come quella mostrata in figura. Nella parte sinistra della figura sono mostrati esplicitamente tutti i segnali e le connessioni compresi il segnale di clock **CLK** ed il segnale di reset **RESET**. La figura di destra mostra invece una rappresentazione semplificata in cui la presenza di unico segnale di clock ed un unico segnale di reset comune a tutti i flip flop è sottintesa.



Questa struttura ammette una semplice rappresentazione VHDL, riportata di seguito.

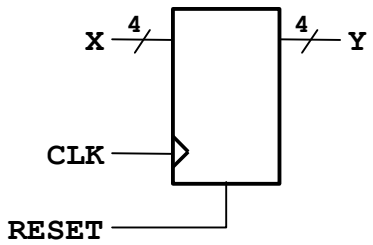
```
entity REG_PP_4_BIT is
    port( CLK:   in  std_logic;
          RESET: in  std_logic;
          X:     in  std_logic_vector(0 to 3);
          Y:     out std_logic_vector(0 to 3)
    );
end REG_PP_4_BIT;
```

```
architetcture rtl of REG_PP_4_BIT is
begin
    reg: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            Y(0) <= '0';
            Y(1) <= '0';
            Y(2) <= '0';
            Y(3) <= '0';
        elsif( CLK'event and CLK = '1' ) then
            Y(0) <= X(0);
            Y(1) <= X(1);
            Y(2) <= X(2);
            Y(3) <= X(3);
        end if;
    end process;
end rtl;
```

Si noti che le uscite di tutti i flip-flop sono aggiornate nello stesso istante poichè tutti i lip-flop sono sensibili allo stesso clock. Per questo motivo è possibile scrivere un unico process in cui tutti i segnali di ingresso sono assegnati ai corrispondenti segnali di uscita. Inoltre, trattandosi di segnali vettoriali, possiamo descrivere il registro in forma molto più compatta e leggibile, ovvero:

```
architetcture rtl of REG_PP_4_BIT is
begin

    reg: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            Y <= "0000";
        elsif( CLK'event and CLK = '1' ) then
            Y <= X;
        end if;
    end process;

end rtl;
```

Questa scrittura, nonchè il concetto su cui si basa, giustifica l'uso del simbolo seguente per la rappresentazione circuitale di registri parallelo-parallelo:

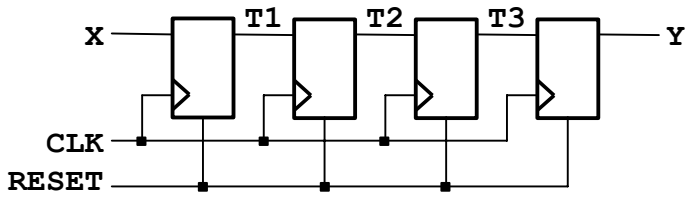


Seguendo questo schema, ricordando il significato dei generic ed il costrutto **`others => ...`**, possiamo facilmente descrivere un registro generico ad N bit. Ecco il codice.

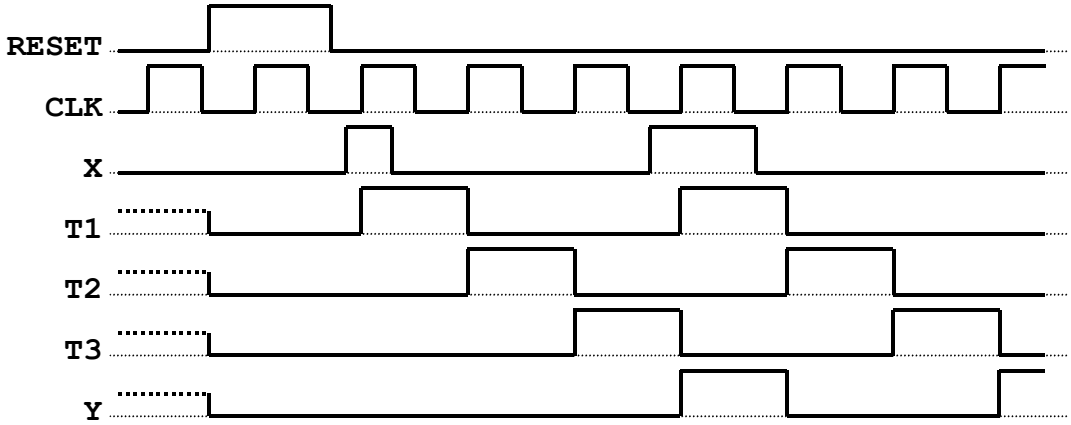
```
entity REG_PP_N_BIT is
    generic( N: integer );
    port( CLK:   in  std_logic;
          RESET: in  std_logic;
          X:     in  std_logic_vector(0 to N-1);
          Y:     out std_logic_vector(0 to N-1) );
end REG_PP_N_BIT;

architetcture rtl of REG_PP_N_BIT is
begin
    reg: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            Y <= (others => '0');
        elsif( CLK'event and CLK = '1' ) then
            Y <= X;
        end if;
    end process;
end rtl;
```

6.2 Registro serie-serie: shift register

Un registro serie-serie è un componente dotato di unico ingresso dati di un bit ed un unica uscita dati anch'essa di un bit. Il registro è costituito da un banco di flip-flop connessi in cascata, cioè connessi in modo tale che l'uscita di uno sia l'ingresso del successivo. Ad esempio, un registro a scorrimento o shift register a 4 bit ha la seguente struttura:



A differenza del registro parallelo-parallelo, questo registro non ha alcun simbolo standard. Il suo comportamento è quello di propagare un segnale di ingresso attraverso i vari flip-flop in corrispondenza di ogni fronte del clock, come mostra il seguente diagramma temporale.



Per procedere all'implementazione VHDL possiamo descrivere a parole il comportamento del registro a scorrimento in questo modo: ad ogni colpo di clock **T1** assume il valore di **X**, **T2** assume il valore di **T1**, **T3** assume il valore di **T2** e **Y** assume il valore di **T3**.
Si noti che questa descrizione, se mal interpretata, potrebbe essere letta come: ad ogni colpo di clock **Y** assume il valore di **X**. Ciò non è vero poichè all'atto dell'attivazione di un process il valore di tutti i segnali viene letto contemporaneamente ed i nuovi valori sono assegnati, anch'essi contemporaneamente, in un istante successivo di tempo. Quindi al primo fronte di clock, il valore di **X**, **T1**, **T2** e **T3** viene letto, quindi vengono eseguiti gli assegnamenti ai segnali **T1**, **T2**, **T3** ed **Y**.
Come regola mnemonica è utile ricordare che in un process tutti i segnali sono assegnati (cioè il loro valore è modificato) alla fine del process stesso.
Al solito, iniziamo dalla descrizione dell'entity:

```
entity REG_SS_4_BIT is
    port( CLK:   in  std_logic;
          RESET: in  std_logic;
          X:     in  std_logic;
          Y:     out std_logic
    );
end REG_SS_4_BIT;

architetcture rtl of REG_SS_4_BIT is
    signal T1, T2, T3: std_logic;
begin

    reg: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            T1 <= '0';
            T2 <= '0';
            T3 <= '0';
            Y  <= '0';
        elsif( CLK'event and CLK = '1' ) then
            T1 <= X;
            T2 <= T1;
            T3 <= T2;
            Y  <= T3;
        end if;
    end process;

end rtl;
```

Un modo equivalente ma più elegante, per descrivere lo stesso comportamento ricorre all'uso di un vettore temporaneo anziché più segnali scalari:

```
architetcture rtl of REG_SS_4_BIT is
    signal T: std_logic_vector(0 to 3);
begin
```

```
    reg: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            T <= "0000";
        elsif( CLK'event and CLK = '1' ) then
            T(0) <= X;
            T(1) <= T(0);
            T(2) <= T(1);
            T(3) <= T(2);
        end if;
    end process;

    -- Assigns the output signal
    Y <= T(3);

end rtl;
```

Ricordando l'uso delle slice, la stessa architettura assume una forma ancora più compatta, regolare e leggibile.

```
architetcture rtl of REG_SS_4_BIT is
    signal T: std_logic_vector(0 to 3);
begin

    reg: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            T <= "0000";
        elsif( CLK'event and CLK = '1' ) then
            T(0) <= X;
            T(1 to 3) <= T(0 to 2);
        end if;
    end process;

    -- Assigns the output signal
    Y <= T(3);

end rtl;
```

Quest'utlima forma ci permette di generalizzare la struttura del registro rendendo la profordità funzione di un generic. Vediamo il codice VHDL corrispondente:

```
entity REG_SS_N_BIT is
    generic( N: integer );
    port( CLK:   in  std_logic;
          RESET: in  std_logic;
          X:     in  std_logic;
          Y:     out std_logic
    );
end REG_SS_N_BIT;
```

```
architetcture rtl of REG_SS_N_BIT is
    signal T: std_logic_vector(0 to N-1);
begin

    reg: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            T <= (others => '0');
        elsif( CLK'event and CLK = '1' ) then
            T(0) <= X;
            T(1 to N-1) <= T(0 to N-2);
        end if;
    end process;

    -- Assigns the output signal
    Y <= T(N-1);

end rtl;
```

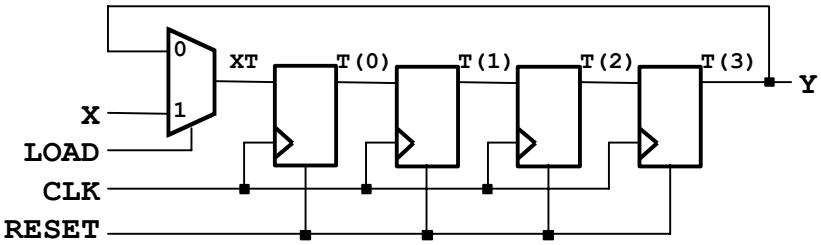
Solo a titolo di esempio prendiamo in considerazione un'altra possibilità implementativa basata sull'uso del costrutto sequenziale for. Lo scorrimento, infatti, ricorrendo ancora all'uso di un vettore temporaneo, può essere espresso dicendo che l'elemento i-esimo del vettore assume il valore dell'elemento (i-1)-esimo, e ciò per tutti gli elementi ad eccezione del primo, cui deve essere assegnato esplicitamente il valore del segnale X, e dell'ultimo che deve essere asseganto esplicitamente al segnale Y. Vediamo quindi come questo approccio si traduce in una specifica VHDL corretta.

```
architetcture rtl of REG_SS_N_BIT is
    signal T: std_logic_vector(0 to N-1);
begin

    reg: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            T <= (others => '0');
        elsif( CLK'event and CLK = '1' ) then
            -- First element
            T(0) <= X;
            -- Other elements
            for I in 1 to N-1 loop
                T(I) <= T(I-1);
            end loop;
        end if;
    end process;

    -- Assigns the output signal
    Y <= T(N-1);

end rtl;
```

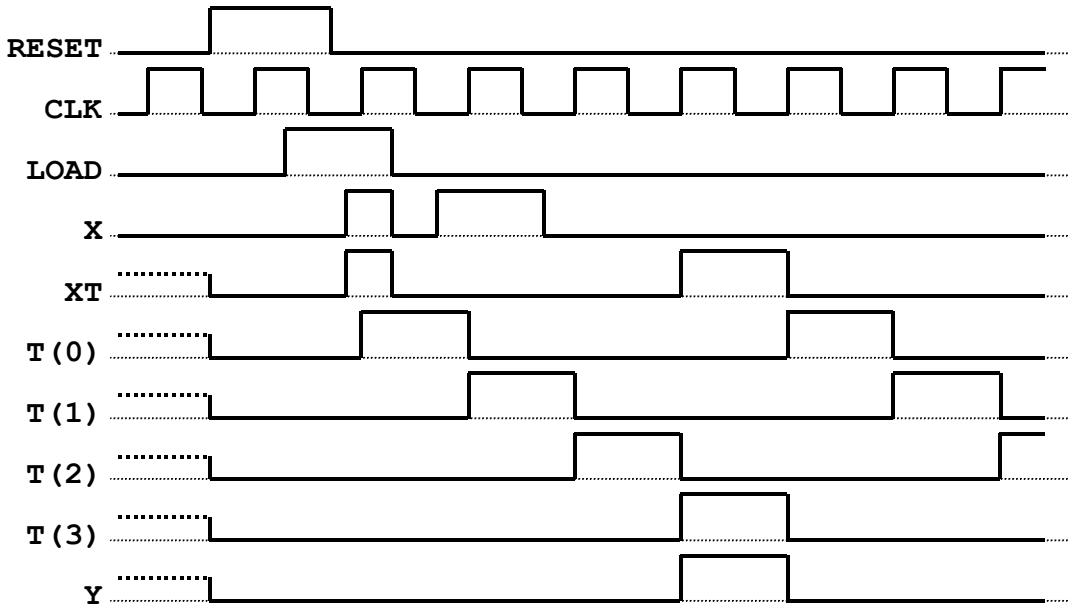
Si noti che l'assegnamento del segnale **Y** effettuato all'interno del process, dopo la fine del ciclo **for** avrebbe costituito un grave errore. In tal caso, infatti, il valore di **Y** sarebbe stato assegnato dopo aver letto il valore di **T(N-1)** ovvero nel successivo ciclo di clock. Il risultato sarebbe quindi stato quello di avere un flip-flop aggiuntivo tra **T(N-1)** ed **Y**; inoltre, tale flip-flop sarebbe privo di segnale di reset, come si può chiaramente dedurre dall'assenza di uno statement di inizializzazione di **Y** al valore zero.

6.3 Registro serie-serie circolare: circular shift register

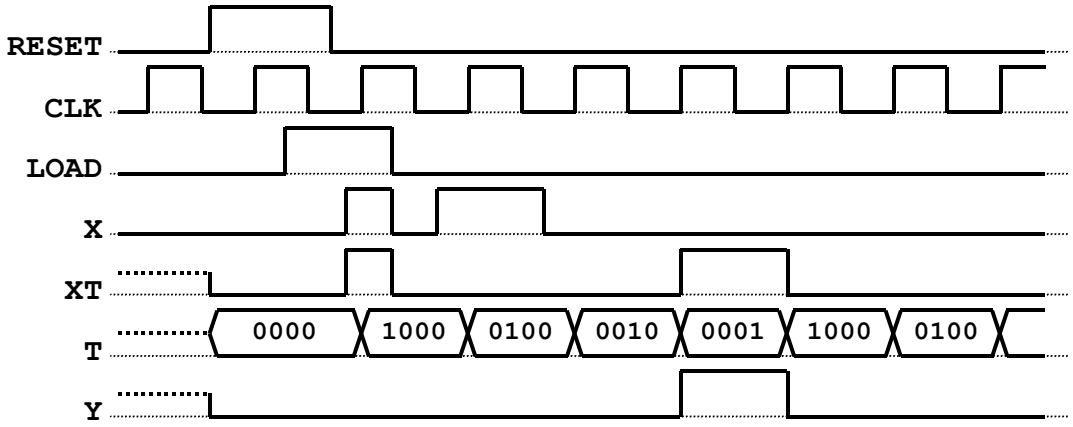
Un registro serie-serie circolare dispone un ingresso **X** di un bit, un'uscita **Y** di un bit e di due modalità di funzionamento: una di caricamento ed una di scorrimento circolare, selezionabili mediante un segnale di controllo **LOAD**. Nella modalità di caricamento, che supponiamo essere attiva quando il segnale **LOAD** vale 1, il registro si comporta come un normale shift register, propagando il segnale di ingresso **X** attraverso i vari flip-flop. Quando il segnale **LOAD** vale invece zero, l'ingresso **X** è sconnesso dal registro ed il primo flip-flop (quello più a sinistra) riceve in ingresso il bit memorizzato nell'ultimo flip-flop (quello più a destra), ottenedo quindi un ricircolo dei dati nella catena di flip-flop. Lo schema seguente mostra l'architettura del registro.



Nella figura sono evidenziati i nomi dei segnali temporanei, in particolare, trattandosi di un unico segnale vettoriale, sono riportati gli indici. Vediamo un esempio di diagramma temporale.



Questo modo di rappresentare il valore dei segnali in un circuito è molto poco compatto. Spesso si ricorre ad una rappresetazione in cui è evidenziato il valore binario di un intero segnale vettoriale piuttosto che il valore delle singole linee. Il diagramma appena visto diviene quindi:



Quest'ultima rappresentazione evidenzia chiaramente lo scorrimento del bit ad uno nella catena di flip-flop rappresentata dal segnale vettoriale **T**. Dal confronto con lo shift register visto in precedenza si nota che il registro circolare altro non è che un normale shift register con ingresso **XT** ed uscita **Y**, combinato con un multiplexer a monte che permette di selezionare il valore da assegnare ad **XT** mediante il segnale di controllo **LOAD**. Quando **LOAD** vale 0 il multiplexer seleziona **Y** (ovvero **T(3)**) altrimenti **X**. Procediamo quindi alla specifica in VHDL seguendo questo schema che mantiene chiaramente separato il multiplexer dal registro a scorrimento.

```
entity REG_SS_CIRCULAR_4_BIT is
    port( CLK:   in  std_logic;
          RESET: in  std_logic;
          LOAD:  in  std_logic;
          X:     in  std_logic;
          Y:     out std_logic
    );
end REG_SS_CIRCULAR_4_BIT;

architetcture rtl of REG_SS_CIRCULAR_4_BIT is
    signal T:  std_logic_vector(0 to 3);
    signal XT: std_logic;
begin

    -- Input multiplexer
    XT <= T(3) when LOAD = '0' else
          X    when LOAD = '1' else
          'X';

    -- Register
    reg: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            T <= "0000";
```

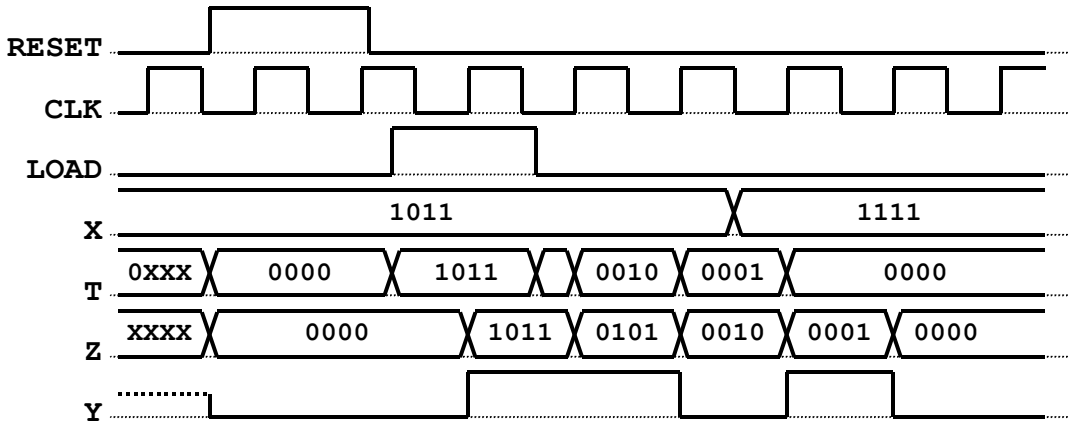
```
        elsif( CLK'event and CLK = '1' ) then
            T(0) <= XT;
            T(1 to 3) <= T(0 to 2);
        end if;
    end process;

    -- Assigns the output signal
    Y <= T(3);

end rtl;
```

Una soluzione differente consiste nel codificare il multiplexer direttamente all'interno del process, prima dell'assegnamento dell'ingresso del primo flip-flop. Ecco la nuova architecture:

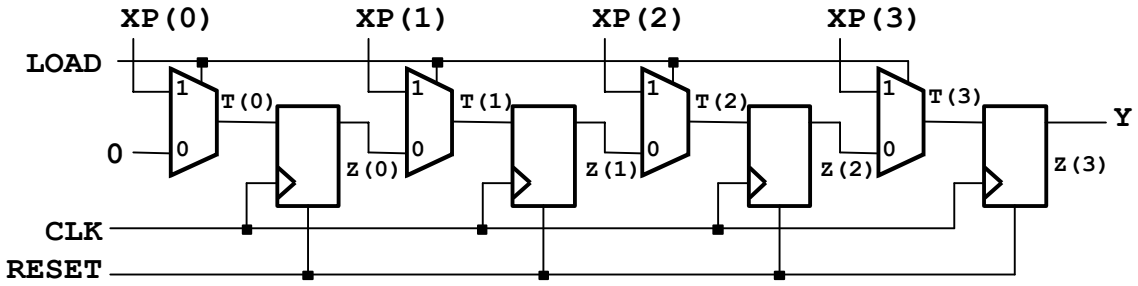
```
architetcture rtl of REG_SS_CIRCULAR_4_BIT is
    signal T:  std_logic_vector(0 to 3);
begin

    reg: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            T <= "0000";
        elsif( CLK'event and CLK = '1' ) then
            -- Input Multiplexer
            if( LOAD = '0' ) then
                T(0) <= T(3);
            else
                T(0) <= Y;
            end if;
            -- Shifter
            T(1 to 3) <= T(0 to 2);
        end if;
    end process;

    -- Assigns the output signal
    Y <= T(3);

end rtl;
```

6.4 Registro parallelo-serie

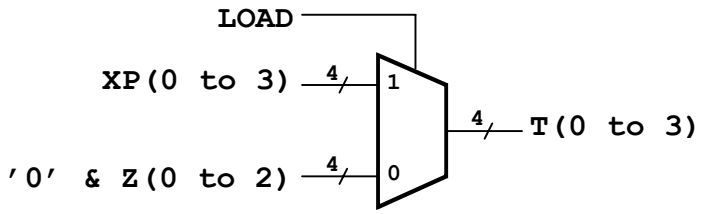
Un registro parallelo-serie dispone di un ingresso parallelo ad N bit **XP**, di un'uscita seriale **Y** ad un bit e di un segnale di controllo **LOAD**. Quando il segnale **LOAD** vale 1 il registro memorizza in parallelo tutta la parola di N bit **XP**, come se si trattasse di un normale registro parallelo-parallelo. Quando invece **LOAD** vale 0, l'ingresso **XP** viene sconnesso dai flip-flop ed il registro si comporta come uno shift register, portando sull'uscita **Y** i bit della parola caricata, uno ad ogni fronte di salita del clock. Per chiarire il comportamento del dispositivo, vediamo il diagramma temporale di un registro a 4 bit in cui, sul primo fronte utile dopo il reset, si carica in parallelo la parola 1011 e nei quattro cicli di clock successivi i bit della parola sono portati in serie sull'uscita. Indichiamo ancora con T il vettore temporaneo utilizzato per la memorizzazione della parola.



Nel diagramma non sono riportati tutti i segnali necessari per l'implementazione ma solamente quelli necessari a chiarirne il funzionamento.



Consideriamo una prima implementazione. Secondo lo schema mostrato, il multiplexer per la selezione degli ingressi dei flip-flop può essere sintetizzato nella seguente rappresentazione:

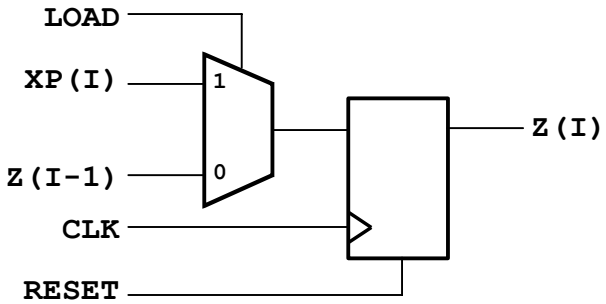


Dallo stesso schema si deduce anche che il multiplexer è asincrono e quindi la sua funzionalità deve essere codificata al di fuori del costrutto per il rilevamento dei fronti di clock. Scegliamo dapprima una implementazione mediante uno statement concorrente per il multiplexer ed un process per la parte sequenziale.

```
entity REG_PS_4_BIT is
    port( CLK:   in  std_logic;
          RESET: in  std_logic;
          LOAD:  in  std_logic;
          XP:    in  std_logic_vector(0 to 3);
          Y:     out std_logic_vector(0 to 3) );
end REG_PS_4_BIT;
```

```
architetcture rtl of REG_PS_4_BIT is
    signal T:  std_logic_vector(0 to 3);
    signal Z:  std_logic_vector(0 to 3);
begin

    -- Input multiplexer
    T <= XP              when LOAD = '1' else
         '0' & Z(0 to 2) when LOAD = '0' else
         "XXXX";

    reg: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            T <= "0000";
        elsif( CLK'event and CLK = '1' ) then
            Z <= T;
        end if;
    end process;

    -- Assigns the output signal
    Y <= Z(3);
end rtl;
```

Questo tipo di specifica è fortemente aderente alla rappresentazione circuitale appena vista. Una soluzione alternativa si basa su una differente visione dello stesso circuito. Non è difficile infatti notare che la struttura del registro è perfettamente regolare ed in particolare è costituita da quattro stadi identici composti da un multiplexer ed un flip-flop. Tutti i multiplexer sono pilotati dal segnale **LOAD** e tutti i flip-flop ricevono lo stesso clock e lo stesso segnale di reset. Ad ogni stadio cambiano gli ingressi dati del multiplexer e l'uscita del flip-flop. Possiamo rappresentare il generico stadio secondo il seguente schema:



Se confrontiamo lo schema di questo stadio con lo schema d'insieme visto in precedenza notiamo che per lo stadio più a sinistra l'ingresso seriale del multiplexer deve essere sempre 0. Notiamo inoltre che per lo stadio più a destra l'uscita del flip-flop è l'uscita seriale del registro cioè **Y**. Per tutti gli stadi intermedi, gli indici seguono lo schema generale mostrato nella figura. Vediamo ora una prima stesura della specifica secondo questo tipo di scomposizione.

```
architetcture rtl of REG_PS_4_BIT is
    signal Z:  std_logic_vector(0 to 3);
begin

    reg: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            Z <= "0000";
        elsif( CLK'event and CLK = '1' ) then
            if( LOAD = '0' ) then
                Z(0) <= '0';
                Z(1) <= Z(0);
                Z(2) <= Z(1);
                Z(3) <= Z(2);
            else
                Z(0) <= XP(0);
                Z(1) <= XP(1);
                Z(2) <= XP(2);
                Z(3) <= XP(3);
            end if;
        end if;
    end process;

    -- Assigns the output signal
    Y <= Z(3);

end rtl;
```

Si noti che il costrutto **if** più interno, destinato al controllo del segnale di caricamento **LOAD**, può essere scritto ricorrendo alle slice in modo più compatto, ovvero:

```
if( LOAD = '0' ) then
    Z <= '0' & Z(0 to 2);
else
    Z <= XP;
end if;
```

Oppure, meglio ancora:

```
if( LOAD = '0' ) then
    Z <= '0' & Z(0 to 2);
elsif( LOAD = '1' ) then
    Z <= XP;
else
    Z <= "XXXX"
end if;
```

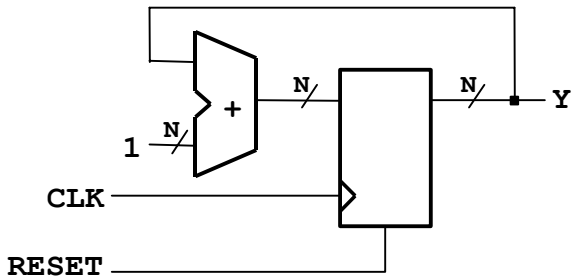
7. Contatori

7.1 Contatore modulo 2^N

Un contatore modulo 2^N è una semplice macchina a stati con 2^N stati che possono essere associati direttamente ai valori di uscita da 0 a 2^N-1. Un tale contatore può essere descritto in modo molto semplice non tanto ricorrento alla rappresetazione mediante tabella della verità, bensì mediante la seguente equazione:

$$Q_{t+1} = Q_t + 1$$

Questa espressione, da intendersi in forma algebrica e non logica, esprime sinteticamente la relazione che lega lo stato prossimo Q_{t+1} allo stato presente Q_t. Nel caso in esame l'uscita coincide con lo stato presente. È noto che per rappresentare 2^N stati secondo la codifica binaria naturale sono necessari N flip-flop. Il funzionamento del contatiore è il seguente: un registro mantiene meorizzato il valore corrente mentre un sommatore calcola il valore dello stato prossimo sommando uno al valore corrente; ad ogni fronte di clock il valore dello stato corrente è aggiornato. Questo corrispone alla seguente semplice architettura:



Questa semplice architettura si traduce in VHDL in maniera quasi immediata. La ragione è che utilizzando una istruzione di assegnamento come:

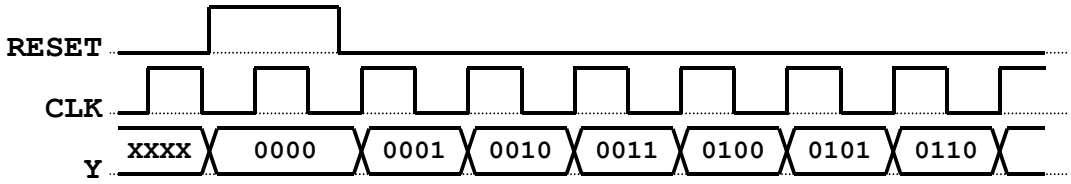
```
TY <= TY + "1";
```

lo strumento di sintesi per prima cosa estende il valore "1" su un vettore della stessa dimensione di TY, quindi calcola la somma e tronca il risultato ancora su una dimnsione pari a quella degli operandi. Vediamo dapprima un semplice contatore modulo 16, ovvero su 4 bit (2^4=16). In tal caso quanto il conteggio raggiunge il valore 15, ovvero 1111, la successiva somma produce il risultato 10000 che, troncato su 4 bit diviene, come ci si aspetta, 0000.

```
entity COUNTER_4_BIT is
    port( CLK:   in  std_logic;
          RESET: in  std_logic;
          Y:     out std_logic_vector(0 to 3)
    );
end COUNTER_4_BIT;
```

```
architetcture rtl of COUNTER_4_BIT is
    signal TY: std_logic_vector(0 to 3);
begin

    -- Counts
    count: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            TY <= "0000";
        elsif( CLK'event and CLK = '1' ) then
            TY <= TY + "0001";
        end if;
    end process;

    -- Assigns the output signal
    Y <= TY;

end rtl;
```

Un esempio di andamento dei segnali per un tale dispositivo è dato dal diagramma seguente.



È molto semplice generalizzare questa struttura per ottenere un countatore a N bit mediante l'uso di un generic.

```
entity COUNTER_N_BIT is
    generic( N: integer );
    port( CLK:   in  std_logic;
          RESET: in  std_logic;
          Y:     out std_logic_vector(0 to N-1) );
end COUNTER_N_BIT;

architetcture rtl of COUNTER_N_BIT is
    signal TY: std_logic_vector(0 to N-1);
begin

    -- Counts
    count: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            TY <= (others => '0');
        elsif( CLK'event and CLK = '1' ) then
            TY <= TY + "1";
        end if;
    end process;
```

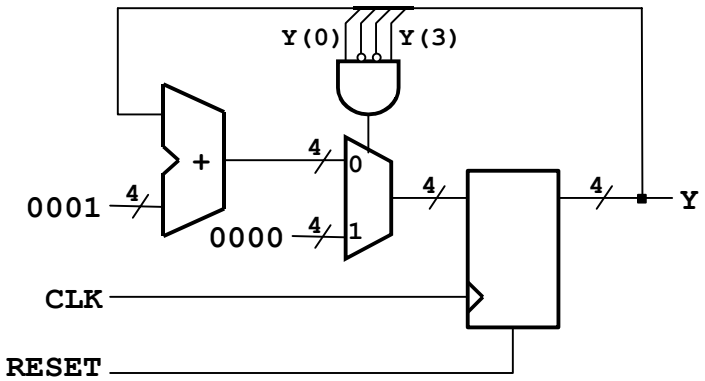
```
    -- Assigns the output signal
    Y <= TY;

end rtl;
```

7.2 Contatore modulo M

Un contatore modulo M altro non è che un contatore, come quello appena visto, con l'unica differenza che il modulo di conteggio è diverso da una potenza del 2. Per questo motivo non è più possibile contare sul fenomeno del troncamento per avere il corretto funzionamento. In questo caso infatti il contatore dovrà iniziare da 0 ed arrivare fino a M-1, quindi, al fronte di clock successivo ritornare a 0. Questo comportamento deve essere codificato esplicitamente in VHDL. Si noti infine che per un contatore modulo M sono necessari $\lceil \log_2 M \rceil$ bit, ovvero un registro costituito da $\lceil \log_2 M \rceil$ flip-flop. Consideriamo, ad esempio, un contatore modulo 10: il numero di registri necessari è $\lceil \log_2 10 \rceil = 4$ ed il codice che ne risulta è il seguente:

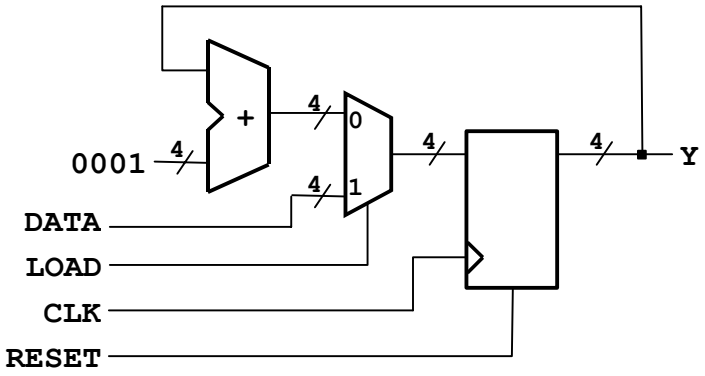
```
entity COUNTER_MOD_10 is
    port( CLK:   in  std_logic;
          RESET: in  std_logic;
          Y:     out std_logic_vector(0 to 3)
    );
end COUNTER_MOD_10;

architetcture rtl of COUNTER_MOD_10 is
    signal TY: std_logic_vector(0 to 3);
begin

    -- Counts
    count: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            TY <= "0000";
        elsif( CLK'event and CLK = '1' ) then
            if( TY = "1001" ) then
                TY <= "0000";
            else
                TY <= TY + "0001";
            end if;
        end if;
    end process;

    -- Assigns the output signal
    Y <= TY;

end rtl;
```

Questo codice produrrà una architettura come quella mostrata nella figura seguente.

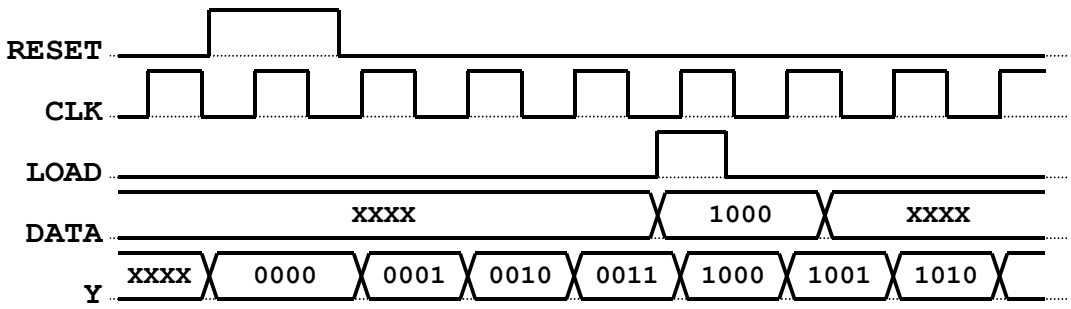


7.3 Contatore con caricamento

Nei contatori appena visti è possibile solamente svolgere l'operazione di reset, ovvero riportare il contatore al valore 0. In molti casi è utile anche poter preimpostare un valore qualsiasi. In questo caso è necessario aggiungere due ingressi: un ingresso di controllo LOAD da un bit, ed un ingresso DATA su N bit che specifica il valore da caricare. Il comportamento che si vuole ottenere è che quando LOAD è attivo (attivo alto, ad esempio) il contatore carica nel registro il valore DATA sul primo fronte di clock utile, mentre quando LOAD è basso il contatore si comporta normalmente. L'architettura che ne deriva è mostrata di seguito.



Vediamo ora il diagramma temporale di un esempio di funzionamento di un contatore modulo 16.



Il codice VHDL che realizza le funzionalità specificate è il seguente.

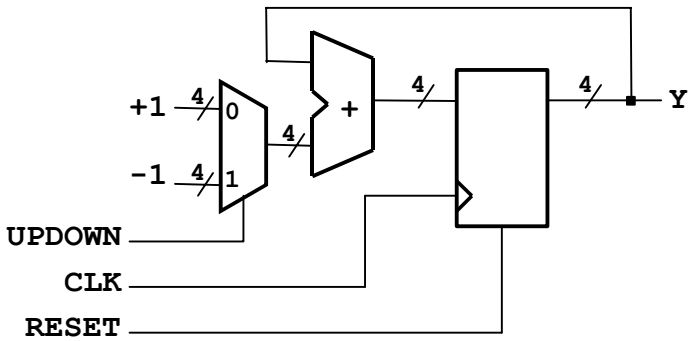
```
entity COUNTER_LOAD is
    port( CLK:   in  std_logic;
          RESET: in  std_logic;
          LOAD:  in  std_logic;
          DATA:  in  std_logic_vector(0 to 3)
          Y:     out std_logic_vector(0 to 3)
    );
end COUNTER_LOAD;

architetcture rtl of COUNTER_LOAD is
    signal TY: std_logic_vector(0 to 3);
begin

    -- Counts
    count: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            TY <= "0000";
        elsif( CLK'event and CLK = '1' ) then
            if( LOAD = '1' ) then
                TY <= DATA;
            else
                TY <= TY + "0001";
            end if;
        end if;
    end process;

    -- Assigns the output signal
    Y <= TY;

end rtl;
```

Naturalmente è possible anche realizzare un contatore con caricamento con modulo diverso da una potenza del 2: a tale scopo è sufficiente comporre in modo adeguato le specifiche del contatore modulo M e del contatore con caricamento.

7.4 Contatore up-down

Un'ultima interessante estensione di un contatore prevede la possibilità di effettuare il conteggio sia in avanti sia all'indietro, sotto il controllo di un opportuno segnale **`UPDOWN`**. L'architettura che permette di realizzare tale dispositivo si basa sulla seguente idea: normalmente il sommatore che permette l'incremento del valore corrente riceve in ingresso il valore costante 1, opportunamente codificato; per poter effettuare il conteggio sia avanti sia indietro è sufficiente poter controllare l'ingresso di tale sommatore. Nel caso il contatore debba procedere in avanti il valore in ingresso al sommatore dovrà essere +1, nel cso in cui debba procedere all'indietro il valore dovrà essere –1. Supponendo di utilizzare un normale sommatore per tale scopo, rimane da verificare che il comportamento sia corretto.
Vediamo dapprima una situazione generica. Supponiamo che il valore corrente per un contatore a 4 bit (modulo 16) sia 8 ovvero 1000. La codifica di –1 in complemento a 2 è 1111. Sommando tali valori si ottiene: 1000 + 1111 = 10111 che, troncato a 4 bit, è 0111 cioè 7. In questo caso non ci sono problemi, ma è opportuno considerare il caso limite che si ha quando il valore corrente è 0

e la direzione di conteggio è all'indietro. In questo caso si ha 0000 + 1111 = 1111 che è appunto il valore corretto, cioè 15.
In conclusione, quindi, l'architettura proposta, e mostrata di seguito, si rivela adatta allo scopo.



Il codice VHDL che specifica un tale componente è il seguente.

```
entity COUNTER_UPDOWN is
    port( CLK:    in  std_logic;
          RESET:  in  std_logic;
          UPDOWN: in  std_logic;
          Y:     out std_logic_vector(0 to 3)
    );
end COUNTER_UPDOWN;

architetcture rtl of COUNTER_UPDOWN is
    signal TY: std_logic_vector(0 to 3);
    signal DY: std_logic_vector(0 to 3);
begin

    -- Selects the increment/decrement
    DY <= "0001" when UPDOWN = '0' else
          "1111" when UPDOWN = '1' else
          "XXXX";

    -- Counts
    count: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            TY <= "0000";
        elsif( CLK'event and CLK = '1' ) then
            TY <= TY + DY;
        end if;
    end process;

    -- Assigns the output signal
    Y <= TY;

end rtl;
```

Si noti che quest aarchitettura inferenzia esplicitamente il multiplexer per la selezione del valore di incremento/decremento al di fuori del process. Una soluzione come la seguente:

```
...
count: process( CLK, RESET )
begin
    if( RESET = '1' ) then
        TY <= "0000";
    elsif( CLK'event and CLK = '1' ) then
        if( UPDOWN = '0' ) then
            TY <= TY + "0001";
        else
            TY <= TY + "1111";
        end if;
    end if;
end process;
```

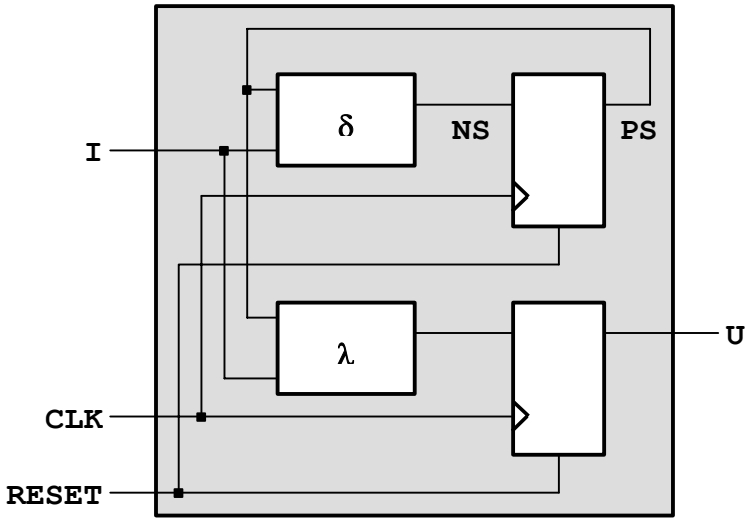
avrebbe probabilmente portato ad un risultato peggiore, costituita da un sommatore, un sottrattore ed un multiplexer (si veda la sezione relativa all'operator sharing).

8. Macchine a stati finiti

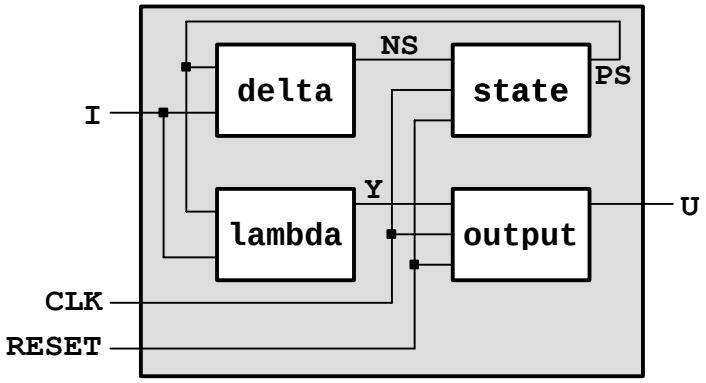
Una macchina a stati finiti o FSM (Finite State Machine) è, come noto, costituita da un banco di flip-flop per la memorizzazione dello stato e da un blocco di logica combinatoria per il calcolo delle funzioni λ e δ. Spesso, nella pratica di progetto, anche le uscite sono registrate mediante un banco addizionale di flip-flop. La sua interfaccia è costituita da un insieme di segnali di ingresso (l'alfabeto di ingresso I), un insieme di segnali di uscita (l'alfabeto di uscita U) ed i segnali di clock e reset. Nel seguito analizzeremo alcune delle soluzioni tipicamente adottate.

8.1 Modello generale

Lo schema generale è quindi il seguente:



Questo modello generale è solitamente adottato anche per la specifica VHDL delle FSM. In particolare, ognuno dei quattro blocchi identificati è realizzato mediante un process. In alcuni casi particolari, alcuni dei process possono anche essere realizzati mediante statement concorrenti. In termini di processi il diagramma precedente diviene.



Analizziamo separatemente i quattro processi necessari alla realizzazione della FSM secondo questo schema. Si noti il significato di alcuni simboli: **`I`** ed **`U`** indicano due insiemi di segnali di ingresso/uscita (non necessariamente vettori), **`NS`** indica lo stato prossimo (Next State) e **`PS`** lo

stato presente (Present State) e **RESET** e **CLK** hanno il significato usuale. Nel codice VHDL che appare nei seguenti paragrafi i simboli ***I***, ***U***, ***NS*** e ***PS*** sono indicati in corsivo per evidenziare questa circostanza.

8.1.1 Il process `delta`

Questo processo ha lo scopo di determinare lo stato prossimo a partire dallo stato presente e dal vettore degli ingressi. Per questo motivo la sua sensitivity list conterrà appunto sia ***I*** che ***PS***. Il processo è puramente combinatorio ed è sostanzialmente costituito da un costrutto **case** che in base allo stato presente seleziona i diversi comportamenti in funzione degli ingressi. Il processo sarà quindi strutturato secondo questo schema:

```
delta: process( I, PS )
begin
    case PS is
        when state0 =>
            -- NS <= ...
        ....
        when stateN =>
            -- NS <= ...
        when others =>
            -- Exception code
            -- NS <= ...
    end case;
end process;
```

Difficilmente è possibile esprimere la complessità della funzione di stato prossimo mediante statement concorrenti.

8.1.2 Il process `lambda`

Questo processo ha lo scopo di determinare l'uscita a partire dallo stato presente e dal vettore degli ingressi. Per questo motivo la sua sensitivity list conterrà appunto sia ***I*** che ***PS***. Il processo è puramente combinatorio è sostanzialmente simile al processo **delta** e produce come uscita un insieme di segnali temporanei ***Y*** che in genere sono poi regitrati prima di essere propagati all'uscita (vedi schema generale). La struttura più generale è quindi la seguente:

```
lambda: process( I, PS )
begin
    case PS is
        when state0 =>
            -- Y <= ...
        ....
        when stateN =>
            -- Y <= ...
        when others =>
            -- Exception code
            -- Y <= ...
    end case;
end process;
```

A volte, tuttavia, la funzione di uscita può assumere forme particolarmente semplici: in tal caso non è detto che la scelta migliore sia quella di utilizzare un process ed un costrutto **case**.

8.1.3 Il process state

Il process **state** è semplicemente un registro parallelo-parallelo che, in corrispondenza di un fronte di clock memorizza in PS il valore dello stato prossimo, ovvero NS. Inoltre, il processo si occupa di riportare la macchina nello stato di reset in corrispondenza del valore attivo del segnale **RESET**. Si noti che lo stato di reset non è necessariamente costituito da tutti zeri ma può assumere una codifica assolutamente arbitraria. Nelle macchine a stati finiti di utilità pratica il segnale di resete può essere sia sincrono che asincrono (più raramente). Vediamo ora la struttura generale di tale process, considerando una FSM con reset sincrono.

```
status: process( CLK )
begin
    if( CLK'event and CLK = '1' ) then
        if( RESET = '1' ) then
            PS <= reset_state;
        else
            PS <= NS;
        end if;
    end if;
end process;
```

Questa struttura è estremeamente tipica e difficilmente si necessita di comportamenti più complessi.

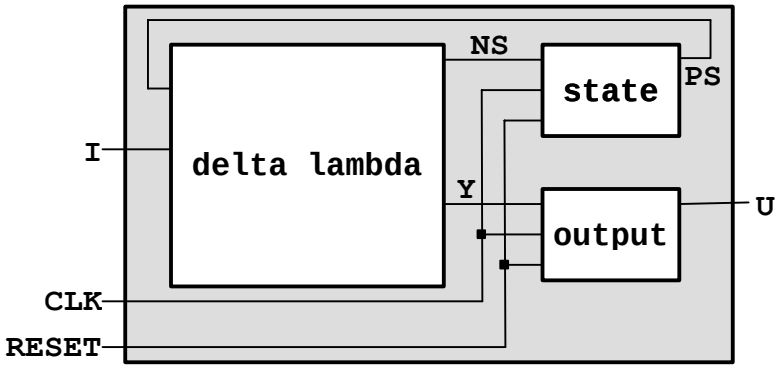
8.1.4 Il process output

Il processo in esame è semplicemente un registro parallelo-parallelo, dotato anch'esso di reset. Per le ragioni esposte precedentemente, il segnale di reset non assegna necessariamente tutti zeri all'uscita ***U***. La forma tipica di tale processo è pertanto la seguente.

```
output: process( CLK )
begin
    if( CLK'event and CLK = '1' ) then
        if( RESET = '1' ) then
            U <= reset_state_output;
        else
            U <= Y;
        end if;
    end if;
end process;
```

8.2 Macchine di Mealy

Nelle macchine di Mealy l'uscita non dipense solamente dallo stato ma anche dagli ingressi. Questo significa che l'uscita è associata alla transizione e non allo stato. In questo caso è pratica comune, salvo nel caso di macchine molto complesse, unire i processi **delta** e **lambda** in unico processo **delta_lambda**, avente lo scopo di calcolare sia lo stato prossimo, sia le uscite. Lo schema seguente mostra i processi necessari e le loro relazioni.



I processi output e state rimangono invariati mentre il processo **delta_lambda** assume la forma mostrata di seguito, ottenuta combinando i processi **delta** e **lambda**.

```
delta_lambda: process( I, PS )
begin
    case PS is
        when state0 =>
            -- NS <= ...
            -- Y  <= ...
        ....
        when stateN =>
            -- NS <= ...
            -- Y  <= ...
        when others =>
            -- Exception code
            -- NS <= ...
            -- Y  <= ...
    end case;
end process;
```

Vediamo ora un semplice esempio di macchina di Mealy, discutendo le scelte effettuate per la codifica. A tale scopo partiamo dalla tabella delle transizioni riportata nel seguito.

	0	1
RST	S0/00	S0/11
S0	S0/00	S1/01
S1	S3/10	S2/00
S2	S3/11	S2/10
S3	S0/01	S0/00

Iniziamo, al solito dalla specifica della entity.

```
entity FSM_MEALY is
    port( I:     in  std_logic;
          CLK:   in  std_logic;
          RESET: in  std_logic;
          U:     out std_logic_vector(0 to 1)
    );
end FSM_MEALY;
```

Supponiamo inoltre di non avere effettuato alcuna scelta, come spesso accade, per la codifica dello stato. In tal caso vogliamo lasciare questo ulteriore grado di libertà allo strumento di sintesi e quindi è necessario mantenere l'informazine relativa allo stato in forma simbolica. A tale scopo, come già visto, si definisce un tipo enumerativo, che nel caso in esame è il seguente:

```
type STATUS is ( RST, S0, S1, S2, S3 );
```

Sono inoltre necessari tre segnali interni: **PS** per lo stato presente, **NS** per lo stato prossimo e **Y** per l'uscita, prima di essere registrata in uscita. Possiamo ora vedere l'architecture.

```
architecture rtl of FSM_MEALY is
    type STATUS is ( RST, S0, S1, S2, S3 );
    signal PS, NS: STATUS;
    signal Y:      std_logic_vector(0 to 1);
begin
    -- Next state and output
    delta_lambda: process( PS, I )
    begin
        case PS is
            when RST =>
                if( I = '0' ) then
                    NS <= S0;
                    Y  <= "00";
                else
                    NS <= S0;
                    Y  <= "11";
                end if;
            when S0 =>
                if( I = '0' ) then
                    NS <= S0;
                    Y  <= "00";
                else
                    NS <= S1;
                    Y  <= "01";
                end if;
            when S1 =>
                if( I = '0' ) then
                    NS <= S3;
                    Y  <= "10";
                else
                    NS <= S2;
                    Y  <= "00";
                end if;
            when S2 =>
                if( I = '0' ) then
                    NS <= S3;
                    Y  <= "11";
                else
                    NS <= S2;
                    Y  <= "10";
                end if;
```

```
            when S3 =>
                if( I = '0' ) then
                    NS <= S0;
                    Y  <= "01";
                else
                    NS <= S0;
                    Y  <= "00";
                end if;
            when others =>
                NS <= RST;
                Y  <= "00";
        end case;
    end process;

    -- State register
    state: process( CLK )
    begin
        if( CLK'event and CLK = '1' ) then
            if( RESET = '1' ) then
                PS <= RST;
            else
                PS <= NS;
            end if;
        end if;
    end process;

    -- Output register
    output: process( CLK )
    begin
        if( CLK'event and CLK = '1' ) then
            if( RESET = '1' ) then
                U <= "00";
            else
                U <= Y;
            end if;
        end if;
    end process;

end rtl;
```

Questa specifica segue lo schema generale presentato e rappresenta la macchina a stati descritta dalla tabella delle transizioni data più sopra. È tuttavia possibile codificare alcune delle transizioni in modo più compatto osservando che o lo stato prossimo o l'uscita per un certo stato presente non dipendono dall'ingresso. Così, ad esempio, le transizioni relative allo stato presente **RST**, riportate qui di seguito per comodità:

```
...
when RST =>
    if( I = '0' ) then
        NS <= S0;
        Y  <= "00";
    else
        NS <= S0;
        Y  <= "11";
    end if;
...
```

possono essere scritte più sinteticamente come:

```
...
when RST =>
    NS <= S0;
    if( I = '0' ) then
        Y  <= "00";
    else
        Y  <= "11";
    end if;
...
```

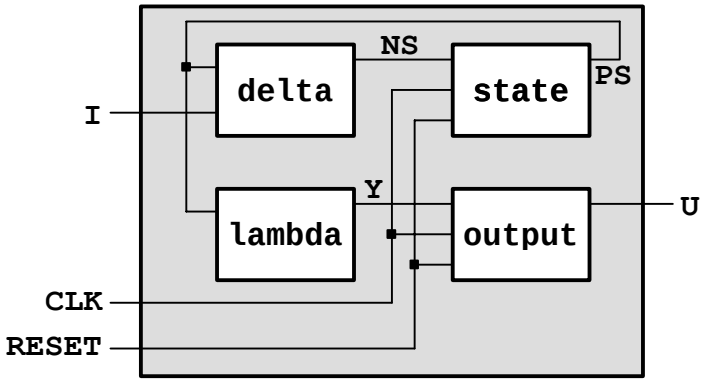
Benché la struttura costituita dai tre processi **delta_lambda**, **state** e **output** sia molto generale e goda di una ottima leggibilità, a volte può essere comunque conveniente raccogliere anche i due processi **state** e **output** in un unico processo **state_output**, strutturato come segue:

```
state_output: process( CLK )
begin
    if( CLK'event and CLK = '1' ) then
        if( RESET = '1' ) then
            PS <= RST;
            U  <= "00"
        else
            PS <= NS;
            U  <= Y;
        end if;
    end if;
end process;
```

Questa nuova architettura costituita da due soli processi è molto sintetica ma non presenta alcun vantaggio in termini di realizzazione hardware.

8.3 Macchine di Moore

In una macchina di Moore le uscite non dipendono dagli ingressi bensì solamente dallo stato presente. Questo fatto porta ad una evidente semplificazione in quanto il processo per il calcolo delle uscite, come mostrato dalla figura seguente, è una rete combinatoria con un numero minore di ingressi. Per le macchine di Moore si utilizza quindi uno schema a tre processi, dedotto dallo schema generale rimuovendo la dipendenza del processo **lambda** dagli ingressi.



Per chiarire come procedere alla specifica di una tale macchina a stati utilizziamo un semplice esempio descritto dalla seguente tabella delle transizioni.

	0	1	U
RST	S0	S0	00
S0	S0	S1	01
S1	S3	S2	11
S2	S3	S2	10
S3	S0	S0	01

Per quanto riguarda la codifica degli stati e la definizione dei segnali intreni alla architecure valgono le considerazioni già viste per le macchine di Mealy. L'entity declaration è la seguente:

```
entity FSM_MOORE is
    port( I:     in  std_logic;
          CLK:   in  std_logic;
          RESET: in  std_logic;
          U:     out std_logic_vector(0 to 1)
    );
end FSM_MOORE;
```

Vediamo ora l'architecture, strutturata nei quattro processi **delta**, **labda**, **state** e **output**. Si noti che il codice di alcune delle transizioni ed alcuni degli assegnamenti delle uscite sono stati già ottimizzati (ovvero espressi in forma sintetica).

```
architecture rtl of FSM_MOORE is
    type STATUS is ( RST, S0, S1, S2, S3 );
    signal PS, NS: STATUS;
    signal Y:      std_logic_vector(0 to 1);
begin

    -- Next state
    delta: process( PS, I )
    begin
        case PS is
            when RST | S3 =>
                NS <= S0;
```

```
        when S0 =>
            if( I = '0' ) then
                NS <= S0;
            else
                NS <= S1;
            end if;
        when S1 | S2 =>
            if( I = '0' ) then
                NS <= S3;
            else
                NS <= S2;
            end if;
        when others =>
            NS <= RST; -- Error
    end case;
end process;

-- Output
lambda: process( PS )
begin
    case PS is
        when RST =>
            Y <= "00";
        when S0 | S3 =>
            Y <= "01";
        when S1 =>
            Y <= "11";
        when S2 =>
            Y <= "10";
        when others =>
            Y <= "00"; -- Error
    end case;
end process;

-- State register
state: process( CLK )
begin
    if( CLK'event and CLK = '1' ) then
        if( RESET = '1' ) then
            PS <= RST;
        else
            PS <= NS;
        end if;
    end if;
end process;

-- Output register
output: process( CLK )
begin
    if( CLK'event and CLK = '1' ) then
        if( RESET = '1' ) then
            U <= "00";
```

```
            else
                U <= Y;
            end if;
        end if;
    end process;

end rtl;
```

Si nota chiaramente che l'uso di un process per la funzione lambda rende la specifica alquanto prolissa. È molto più conveniente infatti sostituire il process con uno statement di assegnamento concorrente condizionato come il seguente:

```
with PS select
    Y <= "00" when RST,
    Y <= "01" when S0,
    Y <= "11" when S1,
    Y <= "10" when S2,
    Y <= "01" when S3,
    Y <= "00" when others;
```

Tale forma è del tutto equivalente al process ma risulta più compatta e leggibile. Inoltre, come nel caso delle macchine di Mealy è possibile raccogliere i processi **`state`** e **`output`** in un unico processo. L'architecture, nella sua forma definitiva diviene quindi la seguente.

```
architecture rtl of FSM_MOORE is
    type STATUS is ( RST, S0, S1, S2, S3 );
    signal PS, NS: STATUS;
    signal Y:      std_logic_vector(0 to 1);
begin
    -- Next state
    delta: process( PS, I )
    begin
        case PS is
            when RST | S3 =>
                NS <= S0;
            when S0 =>
                if( I = '0' ) then
                    NS <= S0;
                else
                    NS <= S1;
                end if;
            when S1 | S2 =>
                if( I = '0' ) then
                    NS <= S3;
                else
                    NS <= S2;
                end if;
            when others =>
                NS <= RST; -- Error
        end case;
    end process;
```

```
    -- Output
    with PS select
        Y <= "00" when RST,
        Y <= "01" when S0,
        Y <= "11" when S1,
        Y <= "10" when S2,
        Y <= "01" when S3,
        Y <= "00" when others;

    -- State and output registers
    state_output: process( CLK )
    begin
        if( CLK'event and CLK = '1' ) then
            if( RESET = '1' ) then
                PS <= RST;
                U  <= "00"
            else
                PS <= NS;
                U  <= Y;
            end if;
        end if;
    end process;

end rtl;
```

8.4 Pipelines

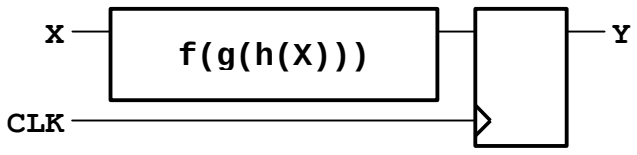
Una pipeline è una struttura in cui fasi diverse dell'elaborazione di un dato avvengono in cicli di clock distinti. Dal punto di vista funzionale possiamo identificare una pipeline come una scomposizione che porta da una generica elaborazione come:

```
Y = f( g( h( X ) ) );
```

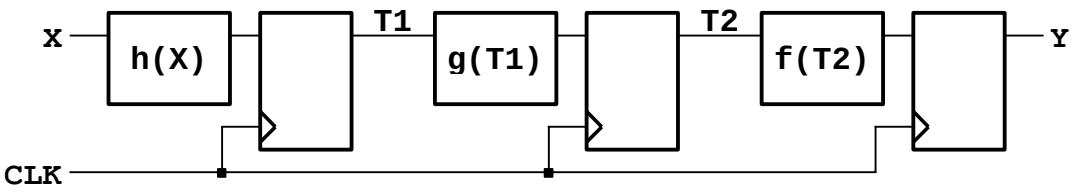
in elaborazioni parziali successive, ognuna delle quali sia effettuata in un ciclo di clock distinto:

```
T1 = h( X );
T2 = g( T1 );
Y  = f( T2 )
```

In termini strutturali, la trasformazione appena vista porta da una struttura come la seguente:



ad una architettura partizionata come segue:



Supponiamo che ognuna delle tre funzioni puramente combinatorie **f**, **g** ed **h** richieda un tempo T: in tal caso il periodo del clock per la prima soluzione implementativa dovrebbe essere $T_{CK, NOPIPE} = 3T$. Seguendo invece l'implementazione come pipeline, ogni stadio, ovvero ogni funzione combinatoria compresa tra due registri, richiede un tempo T quindi il nuovo periodo di clock diviene $T_{CK, PIPE} = T$, ovvero $T_{CK, PIPE} = 1/3\ T_{CK, NOPIPE}$. Il nuovo sistema può quindi funzionare ad una frequenza tre volte superiore.
Tralasciando la forma delle tre funzioni **f**, **g** ed **h**, la struttura iniziale senza pipeline potrebbe essere codificata in VHDL come segue.

```
architeture nopipe of SAMPLE is
begin

    Yreg: process( CLK )
    begin
        if( CLK'event and CLK = '1' ) then
            Y <= f( g( h( X ) ) );
        end if;
    end process;

end nopipe;
```

Per realizzare la pipeline possiamo seguire lo schema di specifica seguente in cui sono esplicitamente visibili i tre processi necessari alla realizzazione dei tre registri:

```
architeture pipe of SAMPLE is
    signal T1, T2: std_logic;
begin

    T1reg: process( CLK )
    begin
        if( CLK'event and CLK = '1' ) then
            T1 <= h( X );
        end if;
    end process;

    T2reg: process( CLK )
    begin
        if( CLK'event and CLK = '1' ) then
            T2 <= g( T1 );
        end if;
    end process;
```

```
    Yreg: process( CLK )
    begin
        if( CLK'event and CLK = '1' ) then
            Y <= f( T2 );
        end if;
    end process;

end pipe;
```

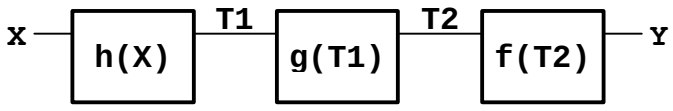
Tuttavia, ricordando che la valutazione dell'assegnamento dei segnali all'interno di un processo avviene solo una volta alla fine del processo stesso, possiamo scrivere in forma più compatta:

```
architeture pipe of SAMPLE is
    signal T1, T2: std_logic;
begin

    regs: process( CLK )
    begin
        if( CLK'event and CLK = '1' ) then
            T1 <= h( X );
            T2 <= g( T1 );
            Y  <= f( T2 );
        end if;
    end process;

end pipe;
```

Si noti che questa scrittura significa che i nuovi valori dei segnali temporanei T1 e T2 e del segnale di uscita Y sono asseganti contemporaneamente alla fine del process stesso, ovvero solamente in corrispondenza di un fronte di salita del clock. Una architecture come la seguente:

```
architeture wrongpipe of SAMPLE is
    signal T1, T2: std_logic;
begin

    T1 <= h( X );
    T2 <= g( T1 );
    Y  <= f( T2 );

end wrongpipe;
```

produce un risultato molto diverso e cioè una struttura del tipo:



Quest'ultima struttura è del tutto identica alla struttura mostrata all'inizio del paragrafo, salvo il fatto che l'uscita Y non è sincronizzata da alcun registro.