Università degli Studi di Messina
Dipartimento di Ingegneria

Introduzione al progetto di sistemi digitali mediante VHDL

Ultima modifica 21 ottobre 2021

Author
Carmine Ciofi

21 ottobre 2021

Indice

1 Linguaggi di descrizione dell'hardware (HDL) 1
1.1 Introduzione . . . 1
1.2 Fondamenti di VHDL : il simulatore VHDL . . . 2
1.3 Fondamenti di VHDL : entità e architettura . . . 4
1.4 Fondamenti di VHDL : elementi lessicali . . . 10
1.4.1 Commenti nei file sorgente VHDL . . . 10
1.4.2 Identificatori . . . 11
1.4.3 Numeri . . . 11
1.4.4 Caratteri . . . 12
1.4.5 Stringhe . . . 12
1.4.6 Costanti e variabili . . . 12
1.4.7 Tipi scalari . . . 13
1.4.8 Sottotipi . . . 18
1.4.9 Qualificatori di tipo . . . 18
1.4.10 Attributi applicabili ai tipi scalari . . . 19
1.4.11 Tipi composti . . . 19
1.5 Generic, component, configuration and generate . . . 21
1.5.1 Uso di costanti generiche ("generic constant") . . . 23
1.5.2 Component e configuration . . . 25
1.5.3 Generate . . . 29

Capitolo 1

Linguaggi di descrizione dell'hardware (HDL)

1.1 Introduzione

I linguaggi di descrizione dell'hardware o HDL (Hardware Description Language) nascono dall'esigenza di disporre di linguaggi formali per descrivere in modo completo e non ambiguo il comportamento di sistemi digitali. I linguaggi HDL più comunemente usati sono il Verilog e il VHDL. In queste note faremo esclusivo riferimento al VHDL. Ci sono diverse ragioni per le quali può essere utile, o necessario, ricorrere a un linguaggio formale per la descrizione dell'hardware.

Nel caso di un hardware digitale già realizzato e disponibile, il ricorso a un linguaggio formale di descrizione dell'hardware rappresenta un modo efficace e non ambiguo di documentare il comportamento dal circuito.

Nel caso in cui si stia progettando un nuovo sistema digitale, il ricorso a un linguaggio formale di descrizione dell'hardware rappresenta un modo efficace per definire in modo chiaro e non equivoco le specifiche del sistema, ovvero il comportamento che si vuole ottenere dal sistema che si intende realizzare.

Nel caso in cui il sistema da realizzare sia composto da più sottosistemi che devono operare in maniera coordinata, disporre di una descrizione formale dei sottosistemi e del modo in cui sono interconnessi consente di eseguire delle simulazioni per verificare che il comportamento del sistema, per come risulta dalla struttura impiegata, sia consistente con le specifiche desiderate.

Nel caso in cui si disponga di una descrizione formale sufficientemente dettagliata, è generalmente possibile arrivare a una sintesi automatica, mediante l'interconnessione di porte elementari standardizzate, del sistema digitale descritto.

Da un altro punto di vista è evidente che se si procede al progetto di un sistema digitale facendo uso di un linguaggio di descrizione dell'hardware, questo fatto si traduce, di per sé, nella disponibilità di una descrizione non equivoca del suo comportamento, nella possibilità di eseguire simulazioni per verificare che il comportamento ottenuto sia conforme a quanto desiderato, nella disponibilità di una base per procedere, ove possibile, alla

sintesi automatica del circuito per una sua implementazione mediante ASIC o mediante FPGA.

Come si può facilmente comprendere, non tutto ciò che può essere immaginato, ancorché corretto da un punto di vista logico, può trovare realizzazione mediante blocchi logici standard. Si supponga per esempio che i blocchi logici standardizzati di cui disponiamo siano costituito da blocchi logici combinatoriali con la possibilità della presenza di un flip flop non trasparente (un registro) che può operare esclusivamente o sul fronte di salita o sul fronte di discesa di un segnale di clock. In una situazione del genere è evidente che se è da un lato possibile immaginare un sistema logico sequenziale sincronizzato che operi sia sul fronte di salita sia su quello di discesa, in nessun caso questo sistema potrà essere sintetizzato (leggi "realizzazione mediante l'interconnessione dei blocchi logici disponibili") perché non si dispone di registri che possano operare sia sul frnte di salita sia sul fronte di discesa del clock.

Anche se il fine ultimo di questo corso è quello di arrivare ad essere in grado di progettare sistemi complessi per la sintesi automatica su dispositivi FPGA, nella prima parte di queste note non ci occuperemo dei problemi relativi alla possibilità di sintetizzare i sistemi che verranno descritti in VHDL: ci occuperemo principalmente di come il VHDL possa essere usato per descrivere un sistema logico e di come si possa procedere alla simulazione del comportamento del sistema stesso a partire dalla sua descrizione in VHDL.

1.2 Fondamenti di VHDL : il simulatore VHDL

Anche se da ora in poi faremo esplicito riferimento al VHDL, molti dei concetti fondamentali che verranno discussi in questa sezione si applicano a tutti i linguaggi di descrizione dell'hardware.

Per capire come sia possibile descrivere il comportamento di un circuito digitale mediante VHDL bisogna avere ben presente il modello di evoluzione dei circuiti logici che viene assunto dal linguaggio. Siccome il modello di evoluzione assunto dal linguaggio coincide con il modello usato per eseguire la simulazione del circuito stesso a partire dalla descrizione in VHDL, per capire i fondamenti del VHDL è utile partire dal comportamento del simulatore VHDL.

In una descrizione VHDL un sistema digitale è chiamato "entità" (parola chiave "entity"). Un sistema digitale può essere ottenuto dalla interconnessione di altri sistemi digitali, ovvero essere il risultato della interconnessione di altre entità. Una entità è caratterizzata da un certo numero di ingressi e un certo numero di uscite. Per semplicità in questa sede assumeremo che tutti gli ingressi e tutte le uscite corrispondano a variabili booleane (in linguaggio VHDL si direbbe che appartengono al tipo predefinito "bit").

Con riferimento alla Fig.1.1 l'entità E_3 rappresenta il circuito digitale descritto in VHDL che ha come ingressi x_1 ed x_2 e come uscite y_1 e y_2. Come indicato dalle aree tratteggiate che vogliono suggerire una vista della "struttura interna", l'entità E_3 si suppone costituita dall'interconnesione di moltre alte entità. Gli elementi (i "fili", semplificando al massimo) di collegamento fra ingressi e uscite di altre entità si chiamano "segnali" (parola chiave "signal") in VHDL. Sono i segnali a stabilire come ogni entità è collegata ad ogni altra. In un certo senso, per chi ha familiarità con la sintassi del simulatore circuitale SPICE, i

segnali in VHDL svolgono lo stesso ruolo dei nomi di nodo nella descrizione di un circuito in accordo con la sintassi SPICE.

Come si vede dalla Fig.1.1, il sistema che siamo interessati a simulare è parte di un'altra entità (E_0) all'interno della quale sono comprese due ulteriori entità (E_1 e E_2) che hanno il compito di generare i segnali di ingresso per l'entità E_3. Il motivo per il quale è necessario ricorrere all'entità "contenitore" E_0 che, come si nota dalla figura, non ha né ingressi né uscite, è che la simulazione VHDL non può tenere conto di variazioni su ingressi indotte da sistemi esterni al simulatore stesso. Le sollecitazioni sul sistema, che dal nostro punto di vista sono esterne al sistema E_3, con il ricorso al sistema "contenitore" E_0 sono, a tutti gli effetti, parte del sistema stesso.

L'elemento fondamentale alla base del meccanismo di simulazione del VHDL è il concetto di evento (parola chiave "event"). Si ha un evento all'istante di tempo t_i se un segnale cambia valore rispetto agli istanti di tempo precedenti. In generale, compito ci ciascuna entità è quella di programmare nuovi eventi (eventi in tempi futuri) ogni volta che si verifica un evento su un segnale collegato a un suo ingresso (vedremo più avanti il caso delle entità che non hanno ingressi). Gli eventi programmati per tempi futuri possono interessare segnali interni all'entità o porte di uscita dell'entità stessa (più propriamente gli eventi interesseranno i segnali collegati alle porte di uscita).

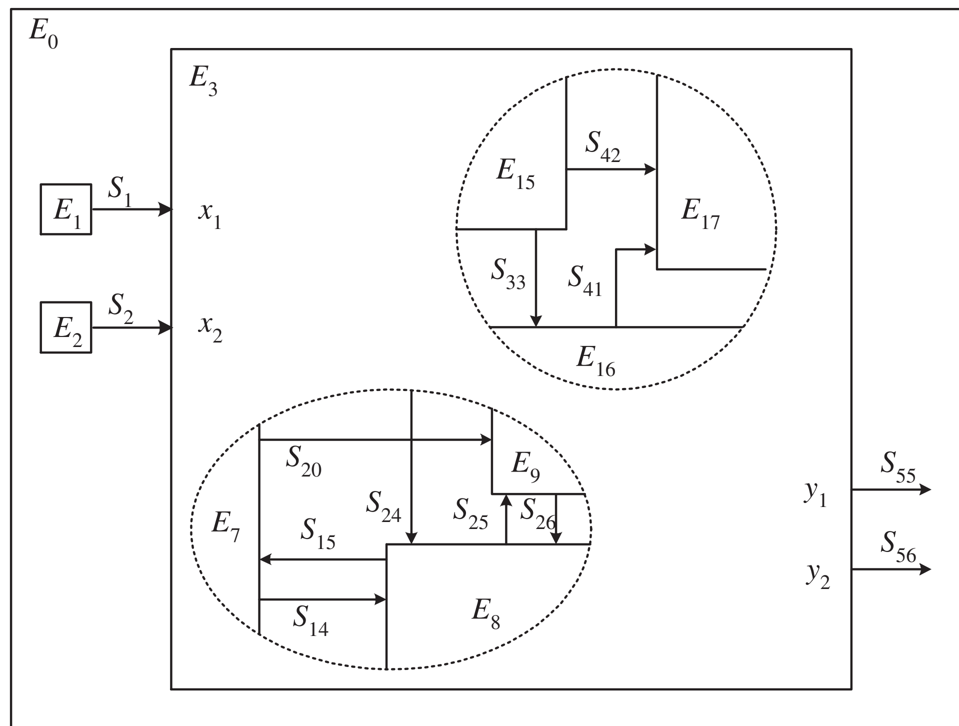

Figura 1.1: Sistema digitale generico.

Il simulatore VHDL mantiene costantemente aggiornata una lista degli eventi, con gli eventi ordinati temporalmente. Il simulatore VHDL tiene nota del tempo trascorso (il tempo "simulato" di evoluzione del sistema). Si rifletta però sul fatto che qualunque cambiamento dello stato del sistema digitale non può che manifestarsi nella forma di un evento, ovvero di cambiamento di valore di un segnale. Pertanto, gli unici istanti di tempo che sono rilevanti sono quelli in cui avvengono eventi. Il simulatore VHDL non segue pertanto l'evoluzione temporale del sistema "istante per istante" che, oltre

che impossibile (gli istanti di tempo in un qualunque intervallo finito di tempo sono infiniti), sarebbe inutile, ma scorre la lista degli eventi, passando da un evento a quello immediatamente successivo. E' bene tenere presente che la lista degli eventi si modifica continuamente, giacché in corrispondenza di ogni evento le diverse entità che compongono il sistema possono programmare nuovi eventi (ovviamente solo nel futuro) e ciascuno dei nuovi eventi programmati entra a far parte della lista degli eventi. La simulazione, in VHDL, procede nel modo seguente: si cerca nella lista il primo evento (primo in ordine temporale); si fa avvenire l'evento (per esempio si fa passare un certo segnale da '0' a '1' o viceversa); si "congela" lo stato del sistema e si esaminano tutte le entità per verificare se, come parte della loro definizione, è previsto che esse programmino nuovi eventi in risposta all'evento appena avvenuto. Tutti i nuovi eventi (programmati per il futuro) vengono inseriti nella lista degli eventi nella posizione specificata da ciascuna entità; completato l'inserimento dei nuovi eventi nella lista degli eventi, il simulatore esamina nuovamente la lista degli eventi e passa all'evento futuro più vicino e si ripete quanto fatto per l'evento precedente. La simulazione si completa quando si esaurisce la lista degli eventi oppure quando viene raggiunto il tempo massimo di simulazione predeterminato.

Sulla base di quanto sopra esposto, nuovi eventi nascono come conseguenza di altri eventi e i segnali assumono specifici valori in corrispondenza di eventi. Come è evidente rimane da specificare come, all'inizio dei tempi, sono stabiliti i valori iniziali di tutti i segnali e come possano essere inseriti i primi eventi nella lista degli eventi, in assenza dei quali il meccanismo di simulazione, per come descritto, non potrebbe procedere.

Proprio perché il VHDL nasce dall'esigenza di descrivere in maniera univoca e non ambigua il comportamento di un sistema digitale, questi due problemi sono affrontati e risolti in modo assolutamente univoco e definito. Per quanto riguarda il valore iniziale assunto dai segnali, visto che per il momento ci siamo limitati a segnali di tipo "bit" che possono assumere solo i valori '0' e '1', all'inizio dei tempi tutti i segnali assumono il valore '0'[1]. Inoltre, l'inizio dei tempi viene trattato come un evento "attiva" tutte le entità, le quali, sulla base di questa attivazione, anche in assenza di specifici eventi sui segnali (siamo all'inizio dei tempi) possono, secondo meccanismi che impareremo a conoscere, programmare eventi.

1.3 Fondamenti di VHDL : entità e architettura

L'entità, per qunato abbiamo visto fino a questo punto, è il blocco funzionale fondamentale attraverso il quale si può descrivere un sistema digitale complesso in VHDL. Alla definizione di una entità si arriva mediante due blocchi di codice VHDL distinti. Il primo blocco è relativo alla dichiarazione dell'entità (costrutto *entity*) il secondo blocco è relativo alla specifiche di funzionamento dell'entità stessa, ovvero al modo in cui l'entità risponde a eventi ai suoi ingressi (costrutto *architecture*).

Nel caso più semplice, la dichiarazione di entità contiene soltanto il nome dell'entità stessa e la dichiarazione delle porte di ingresso e di uscita. Nel dichiarare una nuova entità non

[1] il tipo bit è un tipo definito per enumerazione con due soli elementi che sono '0' e '1'. I segnali possono in realtà appartenere a molteplici tipi. Tutte le volte che i segnali appartengono a un tipo definito per enumerazione (più avanti avremo a che fare con il tipo definito per enumerazione "std_logic" che contiene 9 elementi), all'inizio dei tempi il valore iniziale dei segnali è quello "più a sinistra" nell'elenco che definisce i valori che possono essere assunti.

si vincola in alcun modo il suo comportamento, se non per il fatto di definire il numero e il tipo dei suoi ingressi e delle sue uscite.

Una volta definita l'entità, si può usare il costrutto "architecture" per specificare in dettaglio il suo comportamento. Per semplicità nel seguito faremo riferimento al costrutto "architecture" con il termine italiano di "architettura". E' bene chiarire sin da ora che ad una entità dichiarata possono corrispondere più architetture. Il VHDL mette a disposizioni opportuni meccanismi per associare, volta per volta,l'architettura desiderate fra quelle disponibili per una stessa entità. Può essere utile, a questo punto, procedere alla descrizione e alla simulazione di una entità estremamente semplice, ovvero di un inverter, cosa che ci consentitrà da un lato di familiarizzare con la sintassi VHDL, dall'altro di puntualizzare e chiarire alcune delle conseguenze insite nel processo di simulazione descritto nel paragrafo precedente. Iniziamo quindi a dichiarare l'entita che svolgerà la funzione di inverster e decidiamo di darle il nome di "invertitore". Questa entità avrà un solo ingresso, che chiameremo "a" e una sola uscita che chiameremo "y". Sia all'ingresso sia all'uscita supporremo di collegare segnali di tipo bit e pertanto le porte di ingresso e di uscita saranno anch'esse di tipo bit. Per la dichiarazione dell'entità è sufficiente predisporre un file di testo con il contenuto seguente:

```
entity invertitore is
        port(a:in bit; y:out bit);
end entity invertitore;
```

Listing 1: Dichiarazione dell'entità invertitore.

Le parole evidenziate in verde sono parole chiave del linguaggio e non possono essere usate come nomi per oggetti definiti dall'utente. Le parola "bit" identifica il tipo delle porte di ingresso e di uscita. Si presti attenzione al fatto che il VHDL non distingue fra lettere maiuscole e minuscole, sia nel caso di identificatori, sia nel caso di parole chiave del linguaggio (più avanti vederemo alcune eccezioni).

Un file che contenga la definizione di una entità costituisce un blocco di codice che può essere esaminato dal compilatore VHDL per verificarne le correttezza formale. Se la definizione di entità risulta corretta, la sua esistenza, il suo nome e le sue caratteristiche vengono memorizzate un un data-base locale. Questa operazione, da parte del compilatore VHDL è necessaria perché quando verrà eseminato, in un momento successivo, il file che contiene la definizione del comportamento (blocco "architecture") il nome dell'entità e le sue caratteristiche devono essere già note.

Supponiamo ora di volere fornire una descrizione del comportamento dell'entità appena dichiarata. Per fare ciò dobbiamo predisporre una "architettura". Possiamo predisporre l'architettura in un file separato da quello della definizione dell'entità. La descrizione del comportamento si ottiene avendo ben presente il modo di operare del VHDL. Una possibie architettura per l'inverter è riportata nel listing 2.

Nel seguito faremo riferimento al simulatori ghdl. Parttiamo da una directory vuota nella quale creeremo tutti i file necessari a descrivere completamente il sistema al quale siamo interessati. Supponiamo allora di aver preparato un file di testo dal nome `first_entity_inv.vhd` che contiene il codice nel listing 2.

```
 architecture mia_arc of invertitore is
 -- sezione dichiarativa dell'architettura
 -- che per il momento è vuota
 begin
    process (a) is
    begin
        if (a='1') then
            y<='0' after 10 ns;
        else
            y<='1' after 29 ns;
        end if;
    end process;
end architecture mia_arc;
```

Listing 2: Architettura per l'entità invertitore.

Nella prima riga di codice dichiariamo di voler definire una architettura (parola chiave `architecture`) dal nome `mia_arc` per l'entità `invertitore`. Come già detto, possono essere definite più architetture per una stessa entità. Il VHDL prevede meccanismi per selezionare quale architettura deve essere usata in ogni specifica simulazione o nella fase di sintesi automatica del sistema. Fra la prima riga e la riga in cui compare la parola chiave `begin` per la prima volta è presente la sezione dichiarativa dell'architettura, dove, fra le altre cose, si possono dichiarare segnali che possono essere usati all'interno dell'architettura. In questo esempio non facciamo uso della parte dichiarativa. Si noti che in VHDL tutto quanto segue un "doppio meno" (`-`) viene considerato un commento e viene ignorato. Più precisamente, nell'esaminare una riga, se viene incontrato un "doppio meno" vengono ignorati i caratteri "`-`" e tutti i caratteri che seguono fino a fine riga. La descrizione del comportamento dell'architettura inizia dopo la parola chiave `begin` e può essere ottenuta mediante diversi costrutti. In questo primo esempio facciamo uso del costrutto `process`. Prima di procedere oltre, si noti che non sono presenti nell'architettura dichiarazioni che fanno riferimento alla natura di "a" e di "y": la natura (ovvero il ruolo di porte di ingresso o di uscita e il tipo) di "a" e "y" sono noti perché dichiarati nell'entità per la quale stiamo scrivendo l'architettura. Per comprendere quale sia il senso di quanto contenuto all'interno di un blocco process dobbiamo ricordare come opera il simulatore VHDL: tutte le volte che si verifca un evento, vengono prese in esame tutte le entità e si controlla se l'evento appena avenuto produce altri eventi da inserire nella lista degli eventi. Con questo in mente, lo statement "`process(a)`" può essere interpretato come segue: se, a un certo passo di simulazione VHDL (ovvero quando stiamo prendendo in considerazione un evento nella lista degli eventi) capita che questo evento sia avvenuto su "a", allora il comportamento dell'entità deve essere quello che viene descritto di seguito (fra le parole chiave `begin` e `end process;`). Si ricordi sempre che per "comportamento dell'entità si intende sempre e solo l'eventuale inserimento di eventi nella lista degli eventi generale. Nel nostro sempice esempio c'e' il solo "`a`" in ingresso all'entità e non ci sono altri segnali interni all'architettura che possono subire eventi. In generale, però, possono verificarsi eventi su più ingressi e segnali e la descrizione del comportamento dell'entità può avvenire mediante più processi, che operano in parallelo, ciascuno dei quali è sensibile a un diverso gruppo di eventi. Il gruppo di eventi al quale un process è sensibile si specifica elencando fra parentesi le porte di ingresso o i segnali i cui eventi devono attivare il

process. Questo elenco prende il nome di "sensitivity list". Nel nostro esempio semplice facciamo uso di un solo process nella cui sensitivity list compare esclusivamente la porta di ingresso "a". Ricordiamo che se è avvenuto un evento su "a", nel momento in cui prendiamo in esame il corpo del process, "a" ha il valore assunto per effetto dell'evento: se l'evento è consistito nel passaggio dal valore '0' al valore '1' di un segnale collegato alla porta a dell'entità, all'interno del process "a" ha il valore '1'. Come si può facilmente intuire, anche se non conosciamo ancora i dettagli del linguaggio VHDL, all'interno del process si compiono azioni distinte a seconda l'unico comportamento possibile dell'entità quando l'evento su "a" è consistito in un suo passaggio al valore '1' o meno. Si noti che visto che "a" appartiene al tipo bit, l'unica altra possibilità di avere un evento su "a" senza che questo assuma il valore '1' è che "a" abbia subito un evento che lo ha fatto passare al valore '0' a partire dal valore '1'. La coppia di caratteri "<=" viene utilizzata in VHDL come un unico simbolo che indica che si sta programmando un evento sulla porta (o, se si vuole, sul segnale che sarà collegato alla porta) alla sinistra del simbolo stesso. A destra del simbolo viene indicato il valore che dovrà assumere la porta ("y" in questo caso) e dopo quanto tempo questo valore verrà assunto rispetto al tempo in cui è evento l'evento che sta portando alla programmazione del nuovo evento. Supponiamo per esempio che al tempo $t = 500$ ns "a" sia passato da '0' a '1'. Quando il simulatore VHDL arriva a esaminare gli eventi che avvengono a $t = 500$ ns, siccome fra questi c'e' un evento su "a", e a seguito dell'evento "a" ha assunto il valore '1', il process dell'architettura porterà alla programmazione dell'evento "y diventa '0' al tempo $t = 500 + 10 = 510$ ns. A questo punto il compito del process è concluso e il process, per così dire, resterà in attesa del prossimo evento su "a" per procedere, eventualmente, all'inserimento di nuovi eventi nella lista degli eventi. Resta da esaminare cosa accade all'inizio dei tempi. Nella sezione precedente avevamo detto che tutti i segnali di tipo bit assumono il valore '0'. Pertanto anche "a" al tempo $t = 0$ vale '0' perché varrà zero il segnale ad esso collegato nel sistema complessivo. Varrà inoltre '0' anche il segnale collegato alla porta di uscita "y". Si noti che non possiamo propriamente dire che "y" vale '0' perché all'interno di una architettura (e quindi di un process) no c'e' modo di accedere ("di leggere") il valore di una porta di uscita (ovvero dichiarata out al momento della creazione dell'entità). Inoltre, avevamo affermato genericamente che all'inizio dei tempi tutte le entità erano chiamate a intervenire. Ora che abbiamo visto che a determinare la programmazione di eventi sono i process, possiamo più propriamente dire che all'inizio dei tempi tutti i process di tutte le entità vengono attivati, indipendentemente dalla sensitivity list di ciascuno di essi. Questo significa, nel nostro caso, che all'inizio dei tempi, e indipendentemente dal comportamento futuro dell'ingresso "a", verà programmato l'evento "y diventa '1' a $t = 0 + 29$ ns.

Ora che abbiano creato una entità e abbiamo definito un suo possibile comportamento, supponiamo di volere arrivare a verificarne il funzionamento mediante il simulatore VHDL. Come abbiamo detto nella sezione precedente, dobbiamo riuscire a create una entità che sia in grado di generare un segnale di test da mandare in ingresso all'inverter, dopo di che dobbiamo costruire una entità sneza ingressi e senza uscite, che ontiene al suo interno il generatore e l'inverter per poter procedere alla simulazione.

Cominciamo a vedere come si può creare una entità con 0 ingressi e una sola uscita capace di generare un segnale di test. Supponiamo di chiamare questa entità "sorgente". Un esempio di sorgente (dichiarazione di entità e relativa architettura è riportata nel listing 3

```
entity sorgente is
	port (y:out bit);
end entity sorgente;

architecture prova of sorgente is
begin
	process is
	begin
		wait for 100 ns;
		y<='1';
		wait for 50 ns;
		y<='0';
		wait for 100 ns;
		y<='1';
		wait for 50 ns;
		y<='0';
		wait for 100 ns;
		wait;
	end process;
end architecture prova;
```

Listing 3: Dichiarazione di entità e architettura per il generatore di stimolo.

Nell'architettura dell'entità `sorgente` si sfrutta il fatto che tutti i process, inclusi quelli senza alcuna sensitivity list, vengono attivati all'inizio dei tempi. Senza per ora entrare nei dettagli della sintassi usata, tutto quanto scritto nel corpo del process nel listing 3 corrisponde, al tempo zero, a inserire l'evento `y<='1'` al tempo $t = 0 + 100$ ns, l'evento `y<='0'` al tempo $t = 0 + 100 + 50$ ns e così via. L'ultimo statemant `wait;` senza alcuna specifica di tempo corrisponde a escludere ongli ulteriore possibile attivazione del process. In sostanza, il process svolge il compito, al tempo $t = 0$, di riempire la lista degli eventi secondo le specifiche date.

A questo punto dobbiamo realizzare un sistema (una entità) senza ingressi e senza uscite che consiste nell'opportuno collegamento delle due entità sviluppate fino a questo punto. Chiameremo questa entità `testbench` e per la sua architettura sfrutteremo la possibilità, offerta dal VHDL, di definire il comportamento di una entità mediali il collegamento di altre entità.

La dichiarazione di entità e l'architettura (che chiameremo `struct`) dell'entità `testbench` è riportata nel listing 4.

La dichiarazione dell'entità `tetsbench`, in assenza di ingressi e uscite, si riduce alle linee 1 e 2 nel listing 4. Nella parte dichiarativa dell'architettura sono introdotti due segnali di tipo `bit`: `sin` e `sout` (linee 6 e 7). Il primo segnale svolge la funzione di collegamento fra l'uscita dell'entità che genera il seganle di ingresso e l'ingresso dell'invertitore; il secondo segnale è semplicemente collegato all'uscita dell'invertore, in modo che sia possibile conoscere lo stato dell'uscita in fase di verifica della simulazione. Il comportamento dell'entità (corpo dell'architettura) è in questo caso il risultato del collegamento di due "copie" (in inglese si direbbe "instances") delle entità definite in precedenza. Per poter usare altre entità come parte di una architettura, il compilatore/simulatore VHDL deve essere a

```
 1 entity testbench is
 2 end entity testbench;
 3
 4 architecture struct of testbench is
 5
 6 signal sin:bit;
 7 signal sout:bit;
 8 begin
 9
10 the_inverter:entity work.invertitore(mia_arc)
11                 port map(a=>sin, y=>sout);
12
13 the_source:entity work.sorgente(prova)
14         port map (y=>sin);
15 end architecture struct;
```

Listing 4: Testbench per la simulazione dell'invertitore. I numeri di riga non fanno parte del file. Sono stati aggiunti per semplificare la discussione

conoscenza della loro esistenza e della loro architettura. Ogni volta che si definisce una entità o una architettura, questa può essere farra analizzare dal compilatore VHDL. Una volta analizzate, se sintatticamentye corrette, queste entrano a far parte della libreria predefinita `work`, accessibile dalla directory in cui si sta lavorando. Per usare una copia di una entità occorre darle un nome (nella forma di una label, vedi per esempio `the_inverter` alla linea 10 e `the_source` alla linea 13) e specificare di quale entità si tratti (nella forma `work.<nome_entità>` come nelle linee 10 e 13), e anche quale delle possibili architetture, fra quelle disponibili, deve essere usata[2].

Il collegamento fra le entità si ottiene facendo ricorso ai segnali definiti nel corpo dell'architettura, indicando quale segnale è collegato a ciascuna porta. Per esempio, nella linea 11 si specifica che la porta "`a`" dell'entità `the_inverter` è collagata ("`=>`") al segnale `sin`, mentre la porta "`y`" della stessa entità viene collegata al segnale `sout`. ALlo stesso modo, nella linea 14 si specifica che la porta "`y`" dell'entità `the_source` deve essere collegata al segnale `sin`. Come è intuibile, porte collegate allo stesso segnale sono a tutti gli effetti collegate fra di loro.

Ora che abbiamo a disposizione tutti gli elementi e file necessari, possiamo procedere alla simulazione del sistema. A questo scopo faremo uso del compilatore/simulatore VHDL chiamato GHDL. Il software GHDL è un software di pubblico dominio ed è incluso nella macchina virtuale linux a disposizione degli studenti.

I passaggi necessari per la simulazione sono illustrati nel video prima_simulazione_vhdl.mp4 reperibile dalla pagina web `celec.org`[3].

[2] Il fatto che si possano usare nel corpo di architetture altre entità solo se queste sono già state completamente definite consente una agevole progettazione di tipo bottom-to (dal dettaglio al generale), ma limiterebbe fortemente un approccio di progettazione di tipo top-bottom. Per risolvere questo problema si ricorre ad altri modi di descrivere la struttura di un sistema che vedremo in seguito.

[3] Per un approfondimento degli argomenti trattati in questa sezione, si faccia riferimento al libro di testo (paragrafi da 1.1 a 1.4). Gli stessi argomenti sono riassunti nel capitolo 1 del "VHDL cookbook" (reperibile al link presente sul sito celec.org nella sezione dedicata al corso di sistemi elettronici

1.4 Fondamenti di VHDL : elementi lessicali

Il VHDL è un linguaggio ricco e articolato e non è semplice riassumere sinteticamente tutte le regole sintattiche che devono essere rispettate e tutte le possibilità offerte dal linguaggio ai fini della modellazione e simulazione di sistemo digitali. Piuttosto che elencare pedissequamente quanto può essere facilmente reperito su qualunque libro di testo dedicato al VHDL o sui tanti siti online dedicati alla descrizione del linguaggio, cercheremo di ridurre al minimo l'elencazione sistematica di specifici elementi lessicali e, dove possibile, cercheremo invece di introdurli sempre accompagnati da esempi che ne chiariscono le modalità di utilizzo.

1.4.1 Commenti nei file sorgente VHDL

Abbiamo già visto come si introducono commenti in VHDL, ma lo ripetiamo brevemente in questa sezione. Un commento in VHDL è una sezione di testo che viene completamente ignorata dal compilatore e serve solo per "documentare" il codice. Il VHDL considera commento tutto quanto segue il doppio carattere "-" (meno meno) fino alla fine della riga. Un commento può iniziare dopo una parte di codice VHDL valido, ma si conclude in ogni caso alla fine della riga. Non è previsto un modo sintetico per indicare che più linee consecutive devono essere considerate un commento: ciascuna linea di commento deve cominciare con "-" come nell'esempio riportato nel listing 5.

```
----------------------------------
-- Esempio sull'uso dei commenti--
----------------------------------
entity testbench is
-- questo e' un commento....
end entity testbench;

--quella che segue e' una  architettura
architecture struct of testbench is

signal sin:bit;  --sin collega il generatore all'ingresso
                 --dell'inverter;

signal sout:bit; --sout e' il segnale sul quale si preleva l'uscita del sistema
begin

the_inverter:entity work.invertitore(mia_arc)
                port map(a=>sin, y=>sout);--e uno
the_source:entity work.sorgente(prova)
            port map (y=>sin); --e due
--tutto fatto. possimao chiudere l'architettura
end architecture struct;
```

Listing 5: Esempio di uso di commenti. I caratteri "-" e tutto quanto segue fino alla fine della riga vengono ignorati

programmabili)

1.4.2 Identificatori

Gli identificatori sono i nomi degli "oggetti" usati in VHDL. Sono identificatori i nomi delle entità, delle architetture, dei segnali ecc. In linea di principio sono identificatori anche le parole chiave del linguaggio (per esempio `entity, port, end, if, then architecture, ...`). Le parole chiave del linguaggio non possono essere usate come identificatori di oggetto definiti dall'utente. Un identificatore è una stringa di caratteri, ma non tutte le stringhe di caratteri possono essere identificatori. Un identifcatore, per essere valido deve soddisfare le seguenti caratteristiche:

- deve essere formato solo da lettere dell'alfabeto, cifre decimali e dal carattere "`_`";
- un identificatore deve comunque cominciare con una lettera dell'alfabeto (non può cominciare con un numero o con il carattere "`_`");
- un identificatore non può terminare con il carattere "`_`";
- se sono presenti più caratteri "`_`", questi non possono essere consecutivi.

Si noti inoltre che il VHDL non fa alcuna differenza fra lettere maiuscole e minuscole. Pertanto "`cane`","`Cane`" e "`CANE`" indicano lo stesso oggetto. Tutte le limitazioni sopra elencate possono essere superate ricorrendo ai così detti identificatori estesi. Un identificatore esteso è una qualunque sequenza di caratteri (di qualunque tipo) contenuta fra due caratteri "`\`". Nel caso degli identificatori estesi, lettere maiuscole e minuscole sono interpretate come caratteri diversi. Pertanto "`\cane\`","`\Cane\`" e "`\CANE\`" indicano oggetti diversi.

1.4.3 Numeri

Ci sono due tipi di numeri che possono comparire esplicitamente in un file VHDL: numeri interi e numeri reali (dove il significato di numeri interi e numeri reali deve essere inteso nello stesso senso di qualunque linguaggio di programmazione). Il compilatore VHDL capisce che si ha a che fare con un numero intero se non sono presenti punti decimali. Se sono presenti numeri decimali, il VHDL interpreta il numero come numero reale. Per esempio "`12`" indica un intero di valore 12, mentre `12.0` indica un reale di valore 12. Si noti che in VHDL si può usare la notazione esponenziale sia per gli interi sia per i reali.

Vale la pena di notare il fatto che in VHDL sia i numeri reali sia i numeri interi possono essere espressi rispetto a una base diversa da 10. Le basi possibili vanno da 2 a 16. Per specificare la base di rappresentazione si fa precedere la scringa che rappresenta il numero dal valore della base in decimale e si racchiudono le cifre dal numero fra "`#`". Per esempio `2#0100#` è il numero "quattro" espresso in base 2. Allo stesso modo `2#0.100#` è il numero "zero virgola cinque" espresso in base due. Si noti che tutte le volte che vogliamo riferirci a un numero e non alla sua rappresentazione useremo l'espressione estesa a parole della rappresentazione decimale del numero stesso [4]

Una caratteristica interessante del VHDL è il fatto che nella scrittura di quantità numeriche eventuali caratteri "`_`" sono ignorati. Questo rende più agevole scrivere numeri in basi

[4] Sapere apprezzare la differenza fra un numero e la sua rappresentazione è di grande importanza quando si progettano sistemi di calcolo. Alcuni dei concetti accennati in queste note varranno ripresi quando si affronteranno elementi di aritmetica.

basse (per esempio in base 2). Un numero in base 2 con 16 cifre può essere posto nella forma `2#0100_0001_0101_0000#` in modo da raggruppare le cifre a quattro a quattro per una più chiara lettura/scrittura.

1.4.4 Caratteri

Il singolo carattere (una costante di tipo carattere) viene indicato racchiudendo il carattere fra apici: 'A', 'b', '?'.

1.4.5 Stringhe

Una costante di tipo stringa si indica racchiudendo i caratteri che la compongono fra virgolette. Esempio: "Questa e' una stringa" Il carattere `'&'` viene usato per concatenare più costanti di tipo stringa. Stringhe di bit In VHDL si ha spesso a che fare con sequenze di bit. Si può rappresentare una sequenza di bit con riferimento alla notazione binaria, ottale o esadecimale. Esempi di stringhe di bit secondo le tre possibili notazioni sono:

`"10001101"` oppure `B"1000_1101"` stringa di bit in notazione binaria

`O"342"` stringa di bit in notazione ottale (equivalente a `B"011_100_010"`)

`X"AF"` stringa di bit in notazione esadecimale (equivalente a `B"1010_1111"`)

Nel seguito si farà riferimento, in alcuni casi, alla notazione EBNF per la descrizione degli elementi del linguaggio. Le parole chiave del linguaggio sono indicate in grassetto.

1.4.6 Costanti e variabili

Costanti e variabili devono essere dichiarate prima di poter essere usate in un modello. Una dichiarazione introduce il nome di un oggetto (costante o variabile), ne definisce il tipo e può fissare un valore iniziale.

Dichiarazione di costante

EBNF: dichiarazione_di_costante<=**constant** identificatore {,...}: indicatore_di_sottotipo [:=espressione];

La parte di assegnazione (:=espressione) è ovviamente obbligatoria nel caso della definizione di costanti, ma è indicata come opzionale perché ci sono alcuni casi particolari in cui si può definire una costante senza assegnare il valore. Esempi di dichiarazione di costanti:

```
constant numero_di_byte:integer:=4;

constant numero_di_bit:integer:=8*numero_di_bytes;

constant e:real:=2.718281;

constant ritardo_di_propagazione: time:=3 ns;
```

Dichiarazione di variabile

EBNF: dichiarazione_di_variabile<= **variable** identificatore {,...}: identificatore_di_sottotipo [:=espressione];

Il valore iniziale, nel caso in cui non sia esplicitamente specificato, dipende dal tipo. Per i tipi scalari, il valore iniziale è il valore "più a sinistra" del tipo. L'espressione "più a sinistra" sarà più chiara in seguito.

N. B. Le variabili possono essere usate esclusivamente all'interno di process.

Assegnazione di variabile

EBNF: assegnazione_di_variabile<=[label:] nome:= espressione;

Per esempio:

se `var_a` è una variabile di tipo bit, allora per assegnare il valore `'1'` si scrive `var_a:='1';`

se `var_b` è una variabile di tipo intero si può scrivere `var_b:=3+7;`

1.4.7 Tipi scalari

In VHDL i tipi scalari sono i tipi interi ("integer"), i tipi "floating point", i tipi fisici ("physical") e i tipi per enumerazioni ("enumeration");

Dichiarazione di tipo

EBNF: dichirazione_di_tipo<=**type** identificatore **is** definizione_del_tipo;

esempio di definizione di tipo:

```
type mele is range 0 to 100;
type arance is range 0 to 100;
```

Si noti che, nonostante mele e arance prevedano gli stessi insiemi di valori, si tratta di due tipi diversi! Il VHDL è un linguaggio fortemente tipizzato e non si possono "mischiare mele con arance".

Pertanto, se definissimo le seguenti variabili:

```
variable mela:mele:=0;
variable arancia:arance:=1;
variable frutto:arance;
```

l'assegnazione seguente sarebbe scorretta:

```
frutto:=mela+arancia; --QUESTO NON VA BENE!
```

Si ponga attenzione al fatto che se si vogliono usare tipi definiti dall'utente nella dichiarazione delle porte di una entità, i tipi devono essere in qualche modo già definiti prima della dichiarazione dell'entità stessa. Questo si ottiene definendo i tipi che si intendono sfruttare all'interno di un "package".

Per il momento un package può essere visto come in insieme di definizioni di tipo. Un package può essere contenuto in un file di testo separato.

Esempio di definizione di package:

```
package tipi_interi is
type small_int is range 0 to 255;
type med_int is range 0 to 65535;
end package tipi_interi;
```

Una volta che il file che contiene il "package" viene elaborato, i tipi definiti sono "visibili" se immediatamente prima della definizione dell'entità si usa la parola chiave "use" seguita dalla specifica del package da usare:

```
use work.tipi_interi.all;
entity prova is
    port (a,b: in small_int; c: out med_int);
end entity prova;
```

Nota bene: quando si elabora un package nella stessa directory di lavoro in cui si elaborano le entità, tutte le definizioni (tipi, entità ecc.) entrano a far parte della libreria di default "work".

Tipi interi

I tipi interi in VHDL hanno valori nell'insieme dei numeri interi. Lo standard del linguaggio prevede che l'intervallo dei valori possibili si estenda da un minimo di $-2^{31}+1$ a un massimo di $2^{31}-1$. Il tipo `integer` è il tipo predefinito con valori sul massimo intervallo dell'implementazione di VHDL in uso. Si possono definire nuovi tipi "intero" nel modo seguente:

EBNF: definizione_di_tipo_intero <= **range** espressione_semplice {**to** | **downto**} espressione_semplice

Si noti che le parole chiave **to** e **downto** servono a definire un ordinamento interno di tipo crescente o decrescente all'interno del tipo. Il valore "più a sinistra" del tipo è il valore più piccolo nel caso di ordinamento crescente mentre corrisponde al valore più grande nel caso di ordinamento decrescente.

Esempi di definizione di tipi interi:

```
type giorno_del_mese is range 0 to 31;
type anno is range 0 to 2100;
```

A questo punto si possono definire delle variabili che appartengono ai tipi definiti:

```
variable oggi:giorno_del_mese:=9;
variable anno_di_nascita:anno:=1965;
```

Ricordiamo che non è permesso scrivere espressioni come:

```
oggi:=anno_di_nascita:=oggi;
```

perché `oggi` e `anno_di_nascita` appartengono a due tipi distinti.

Sono possibili definizioni del tipo:

```
constant numero_di_bit:integer:=32;
type bit_index is range 0 to numero_di_bit-1;
```

Sui tipi interi sono definite alcune operazioni:

- `+` : addizione o identità;
- `-` : sottrazione o negazione
- `*` : moltiplicazione
- `/` : divisione
- `mod` : modulo
- `rem`: resto
- `**`: elevamento a potenza (esponenti interi non negativi)

Attenzione: la divisione `(A/B)` e il resto sono tali da soddisfare la relazione seguente:

`A=(A/B)*B+(A rem B)`

con `(A rem B)` che ha lo stesso segno di `A` e un valore assoluto inferiore al valore assoluto di B.

L'operazione modulo (`mod`) coincide invece con la omonima funzione matematica.

Tipo Floating Point

Lo standard prevede che l'intervallo dei valori rappresentabili deve essere almeno pari a (-1.8E+308 +1.8E308)

In VHDL è esiste il tipo predefinito `real` che assume tutti i valori possibili nell'implementazione in uso.

La definizione di altri tipi floating point si ottiene nel modo seguente:

EBNF: Floating_point_definition <= **range** simple_expression {**to** | **downto**} simple_expression

Come si può notare la defizione di tipi floating point appare identica a quella dei tipi interi. La differenza sta nel fatto che gli estremi del range sono espressi come numueri "reali".

Esempio di definizione di un tipo floating point

```
type real_value is range 0.0 to 1e6;
```

Tipi “fisici”

Il VHDL prevede un tipo "fisico" predefinito che serve ad esprimere il tempo ("time").

Piuttosto che fornire la descrizione EBNF preferiamo in questo caso procedere per esempi cominciando proprio dal modo in cui è (internamente) definito il tipo "time":

```
type time is range XXX to YYY ---range dipendente dall'implementazione
    units
    fs;
```

```
    ps= 1000 fs;
    ns= 1000 ps;
    us= 1000 ns;
    ms=1000 us;
    sec=1000 ms;
    min= 60 sec;
    hr= 60 min;
    end units;
```

Si noti che è obbligatoria la presenza di uno spazio fra il valore numerico e l'unità di misura tutte le volte che si assegna un valore a una variabile di tipo fisico.

Usando lo stesso metodo, possiamo definire altri tipi fisici. Per Esempio:

```
type resistance is range 0 to 1e9
    units
    ohm;
end units;
```

Con la definizione data, possiamo assegnare un valore a una variabile di tipo "`resistance` nel modo seguente:

```
r23:= 330 ohm;
```

Se lo riteniamo utile, come nel caso del tipo "time", possiamo definire anche unità principali e secondarie. Per esempio:

```
type resistance is range 0 to 1e9
    units
    ohm;
    kohm= 1000 ohm;
    Mohm=1000 kohm;
    end units;

type lunghezze is range 0 to 1e9
    units
    um;
    mm=1000 um;
    m=1000 mm;
    pollice= 25.4 mm;
    piede= 12 pollici;
    km= 1000 m;
    end units;
```

Tipi definiti per enumerazione

Nei tipi definiti per enumerazione vengono esplicitamente specificati gli elementi del tipo mediante l'elencazione di identificatori o di caratteri alfanumerici.

EBNF: enumeration_type<=((identificatore | carattere), {,...})

Esempi:

```
type colori_semaforo is (rosso, giallo, verde);
type colori is ('b','w','g','y','r');
```

Il tipo `character` i VHDL è un tipo predefinito definito per enumerazione:

```
type character is (nul, soh, ....'a','b',.......);
```

Un altro importante tipo predefinito definito per enumerazione è il tipo `boolean`

```
type boolean is (false,true);
```

Questo tipo è usato per rappresentare il valore delle "condizioni" che controllano l'esecuzione di modelli comportamentali. Ci sono un certo numero di operatori logici e relazsionali che producono risultati del tipo boolean.

Le espressioni 123=123, 'A'='A', 7 ns=7 ns producono il valore true;

Le espressioni 123=443, 'A'='z', 7 ns=17 us producono il valore false

Gli operatori logici `and, or, nand, nor, xor, xnor e not` si applicano a espressioni o variabili di tipo boolean e producono risultati di tipo boolean.

Gli operatori `and, or, nand, nor` sono chiamati operatori di tipo "short circuit". Questa denominazione deriva dal fatto che nella valutazione dell'espressione logica viene valutata prima l'espressione a sinistra dell'operatore. Se il risultato è tale che il valore dell'espressione logica è determinato (per esempio nel caso dell' and, se l'espressione a sinistra vale false, il risultato dell'operazione `and` è comunque false) l'espressione a destra non viene valutata. Questo può essere utile in un caso come quello seguente:

```
if ((b/=0) and (a/b>1)) then
....
....
```

L'espressione che compare come argomento dell' "if" non produce errore se `b=0`. Infatti in questo caso l'espressione a destra non viene valutata. Al contrario, l'espressione

```
if ((a/b>1) and (b/=0) ) then
....
....
```

produrrebbe errore se venisse valutata con `b=0`.

Un altro importante tipo definito per enumerazione è il tipo bit:

```
type bit is ('0','1');
```

Sul tipo bit sono definite operazioni di tipo logico corrispondenti alle operazioni dell'algebra booleana. Si faccia attenzione al fatto che l'operatore `and`, quando applicato al tipo bit produce come risultato un valore del tipo bit. `('1' and '1')` produce come risultato il valore '1' del tipo bit. Come dovrebbe essere ormai ovvio, non hanno significato espressioni del tipo `('1' and true)`.

Un altro tipo per enumerazione che, pur non essendo predefinito, useremo spesso in quanto presente in una libreria standardizzata è il tipo `std_ulogic`, così definito:

```
type std_ulogic is (
    'U', --Unitialized
```

```
    'X', --Forcing Unknown
    '0', --Forcing zero
    '1', --Forcing one
    'Z', --High impedence
    'W', --Weak Unknown
    'L', --Weak zero
    'H', --Weak one
    '-'  --Don't care
    );
```

Come vedremo in seguito, questo tipo è alla base del tipo std_logic usato per descrivere in modo più realistico il comportamento dei sistemi logici.

1.4.8 Sottotipi

Come suggerisce il nome, un sottotipo è un tipo ottenuto a partire da un tipo già definito. L'insieme dei valori di un sottotipo è generalmente un sottoinsieme dei valori del tipo. Variabili di un sottotipo e del tipo di partenza possono essere combinati in operazioni di assegnamento o di calcolo.

La definizione di un sottotipo si ottiene nel modo seguente:

EBNF: subtype_declaration<=**subtype** identifier **is** subtype_indication;

subtype_indication<= type_mark [**range** simple_expression { **to** | **downto**} simple expession]

Per esempio:

```
subtype small_int is integer range 0 to 255;
subtype med_int is integer range 0 to 65535;
```

Supponiamo ora che sia:

```
variable var1:small_int:=37;
variable var2:med_int:=528;
```

L'espressione

```
var2:=var2+var1;
```

ha perfettamente senso.

Si noti che l'espressione

```
var1:=var1+var2;
```

pur essendo lecita, produce un errore perché il risultato dell'operazione non appartiene al sottotipo della variabile a sinistra dell'operazione di assegnazione.

1.4.9 Qualificatori di tipo

Supponiamo di aver definito i seguenti tipi:

```
type colore is (rosso, verde,giallo);
type semaforo is (rosso,verde,giallo);
```

I tipi sono distinti e pertanto il valore “rosso” del tipo `colore` è cosa diversa dal valore “rosso” del tipo `semaforo`. Per distinguere chiaramente i due valori si può fare esplicito riferimento al tipo di appartenenza mediante le espressioni:

```
colore'(rosso)
semaforo'(rosso)
```

1.4.10 Attributi applicabili ai tipi scalari

Sia TIPO l'identificatore di un tipo o sottotipo scalare. Allora

TIPO'left indica l'elemento più a sinistra del tipo

TIPO'right indica l'elemento più a destra del tipo

TIPO'low indica il valore più piccolo del tipo

TIPO'high indica il valore più grande del tipo

TIPO'ascending è un valore boolean che vale true se il tipo è ordinato in modo crescente, altrimenti vale false

TIPO'image(x) produce una stringa che rappresenta il valore di x

TIPO'value(s) produce il valore x del tipo “TIPO” che è rappresentato dalla stringa s

TIPO'pos(x) produce un intero che indica la posizione di x all'interno dell'insieme dei valori del tipo (il primo valore del tipo ha indice di posizione 0)

TIPO'val(n) produce il valore x che ha indice di posizione n

TIPO'succ(x) produce il valore del tipo di indice di posizione immediantamente superiore a quello di x

TIPO'pred(x) produce il valore del tipo di indice di posizione immediantamente inferiore a quello dix

TIPO'leftof(x) produce il valore del tipo con posizione immediantamente a sinistra di x

TIPO'rightof(x) produce il valore del tipo con posizione immediantamente a destra di x

1.4.11 Tipi composti

Array

Un array è una collezione ordinata di valori tutti apparteneti allo stesso tipo. La definisione di un tipo array segue la sintassi:

EBNF: array_type_definition<=**array** (discrete_range ,.... **of** element_subtype_indication)

discrete_range<=discrete_subtype_indication | simple_expression (**to** | **downto**) simple_expression

subtype_indication<=type_mark [**range** simple_expression (**to**|**downto**) simple_expression]

Esempi:

```
type word is array (0 to 31) of bit;
type word is array (31 downto 0) of bit;
```

Supponiamo sia

```
type colori is (nero,rosso,verde,blu,giallo,bianco);
```

possiamo ora definire un array di numeri naturali ("natural" è un sottotipo predefinito del tipo intero) come:

```
type quantita is array (rosso to giallo) of natural;
```

In questa definizione ci si affida al fatto che il tipo del range è chiaro dal contesto. Se non fosse cosi' (per esempio se rosso e giallo sono elementi di più tipi definiti per enumerazione) allora si può scrivere:

```
type quantita is array (colori range rosso to giallo) of natural;
```

Altro modo:

```
subtype alcuni_colori is colori range rosso to giallo;
type quantita is array (alcuni_colori) of natural;
```

Se si definisce una variabile di tipo array

```
variable parola:word;
```

allora `parola(3)` indica il bit di indice 3 all'interno della parola.

Se si ha

```
variable parola1,parola2:word;
```

allora l'espressione

```
parola1:=parola2;
```

copia tutti gli elementi di `parola2` in `parola1`. E' possibile definire array multidimensionali.

array "unconstrained"

E' possibile definire un tipo di array in cui si specifica la natura (il tipo) dell'indice, ma non il numero di elementi dell'array. L'intervallo di valori da usare come indice, purché compatibile con la definizione del tipo dell'array, potrà essere fatta nel momento in cui si definisce un elemento (costante, variabile o segnale) che appartiene al tipo.

La definizione di un array unconstrained e la definizione di variabili e/o costanti appartenenti al tipo può avvenire in analogica con l'esempio seguente:

```
type multidim is array (natural range <>) of integer;
.
.
.

variable ar1:multidim(0 to 15);
constant zeri:mutidim(63 downto 0):=(others=>0);
```

In VHDL esiste un tipo di array predifenito e unconstrained di bit che si chiama bit_vector. Il tipo bit_vector può essere pensato come definito nel modo seguente:

```
type bit_vetor is array  (natural range <>) of bit;
```

Con questo tipo predefinito a disposizione è facile definire "array di bit" che possano rappresentare collezioni ordinate di bit. Per esempio:

```
variable un_nibble:bit_vector(3 downto 0);
variable un_byte:bit_vector(7 downto 0);
variable una_word:bit_vector (31 donto 0);
```

Assegnazione di valori ad un array

L'inizializzazione di costanti o variabili di tipo array può essere ottenuta come segue

```
type point is array (1 to 3) of real;
constant origin:point:=(0.0,0.0,0.0);
variable view_point:point:=(10.0,20.0,0.0);
```

In alternativa alla "associazione posizionale", si può specificare il valore da assegnare alle variabili corrispondenti a specifici indici. Per esempio:

```
variable view_point:point:=(1=>10.0, 2=>20.0,3=>0.0);
```

Altra interessante possibilità:

```
type coeff_array is array (coeff_ram_address) of real;
variable coeff:coeff_array:=(0=>1.6, 1 =>2.3, 2 to 63 =>0.0);
```

o ancora:

```
variable coeff:coeff_array:=(0|3|5|9 =>1.2, 1=>2.3, others=>0.0);
```

Il simbolo "|" serve per separare indici a cui corrisponde lo stesso valore. "others" significa "per tutti gli indici non esplicitamente indicati" e deve essere sempre messo in fondo).

Si può ovviamente fare riferimento al singolo elemento di un array indicando l'indice dell'elemento al quale si vuole fare riferimento fra parentesi tonde subito dopo il nome dell'array.

1.5 Generic, component, configuration and generate

In VHDL è possibile descrivere il comportamento di una entità in termini della interconnessione fra altre entità. Nell'esempio seguente, descriviamo l'architettura di un flip flop SR mediante l'interconnessione di due porte nor.

Le modalità di descrizione strutturale dell'esempio precedente prevedono che le entità che intendiamo usare come parte di una architettura siano state precedentemente definite e immagazzinate come parte di una libreria (la libreria predefinita work nel caso precedente). Questo significa che per poter descrivere l'architettura del flip flop in termini della interconnesisone di entità, queste ultime devono essere completamente definite prima che possano essere impiegate nella descrizione stessa. Questi vincoli costituiscono

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

architecture behav of nor_port is
begin
        y<=a nor b after 5 ns;
end architecture behav;


entity ff_sr is
        port (s,r: in bit; q, not_q: out bit);
end entity ff_sr;

architecture struct of ff_sr is
signal q_int, not_q_int: bit;
begin
        nor1:entity work.nor_port port map (a=>s, b=>q_int, y=>not_q_int);
        nor2:entity work.nor_port port map (a=>r, b=>not_q_int, y=>q_int);

        q<=q_int;
        not_q<=not_q_int;

end architecture struct;
```

Listing 6: Esempio di descrizione strutturale di un flip-flop SR.

una limitazione nella possibilità di affrontare la progettazione di un sistema digitale in un approccio top-down, ovvero in una situazione in cui si voglia in primo luogo definire l'architettura generale del sistema in termini della interconnessione di blocchi funzionali per poi passare in un secondo momento a definire l'architettura dettagliata di ogni sotto-blocco. Nell'esempio che abbiamo riportato più sopra questa limitazione può apparire tollerabile, ma non è necessariamente così nel caso di sistemi più complessi. Si immagini di dover progettare un microcontrollore, ovvero un sistema che sia composto da un microprocessore, una memoria e numerose periferiche interconnesse fra loro a formare un unico circuito integrato. In una situazione del genere è praticamente indispensabile affrontare il progetto a partire dal livello di astrazione più alto per poi scendere via via a definire le specifiche a più basso livello. Fortunatamente in VHDL è prevista la possibilità di poter descrivere una architettura in termini della interconnessione fra entità di cui si conosca (o si preveda) in dettaglio la configurazione delle porte ma non la funzione o la struttura interna dettagliata. Ciò avviene mediante il ricorso all'uso del costrutto "component" che varrà descritto più avanti. Evidentemente, a un certo punto dello sviluppo del progetto sarà necessario definire in dettaglio l'architettura di ogni "component". Questo avviene mediante una procedura di configurazione ("configuration") che consente di associare una specifica entità (e relativa architettura) a ciascun "component", una volta che siano disponibili le descrizioni funzionali o strutturali complete delle entità che costituiscono l'effettiva implementazione di ciascun componente.

In questa stessa sede affronteremo il problema della "programmabilità" delle entità, inten-

dendo con questo la possibilità di specificare alcuni parametri relativi al comportamento di una unità che, in generale, possono essere conosciuti solo all'atto del suo impiego in una descrizione strutturale nei termini dell'esempio proposto o, come vedremo più avanti, all'atto della operazione di configurazione. Per comprendere l'opportunità di disporre di un certo grado di programmabilità per una entità, faremo riferimento a due situazioni tipiche.

1.5.1 Uso di costanti generiche ("generic constant")

In una prima situazione, supponiamo che si debba creare una entità che rappresenti il comportamento di una porta NOR. Facciamo riferimento alla coppia entità-architettura usate nell'esempio precedente, riprendiamo la definizione della porta NOR.

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

architecture behav of nor_port is
begin
        y<=a nor b after 5 ns;
end architecture behav;
```

Listing 7: Descrizione di una porta NOR.

Nell'architettura della porta NOR riportata nel listing 7 abbiamo specificato un ritardo di propagazione di 5 ns. Ora è ben noto che il ritardo di propagazione non è una proprietà intrinseca di una porta, ma dipende dal numero di ingressi che la sua uscita deve pilotare. Senza pretesa di eccessiva accuratezza, nel caso di una tecnologia CMOS possiamo sostenere che il ritardo di propagazione è proporzionale al numero di ingressi pilotati. In effetti, nel caso del CMOS, il ritardo di propagazione di una porta NOR può essere diverso anche in ragione del modo in cui sono pilotati gli ingressi, ma per non complicare troppo la discussione, trascureremo questo aspetto. Se volessimo tener conto della dipendenza del tempo di propagazione del numero di ingressi pilotati, attenendoci alla sintassi riportata più sopra, non avremmo altra scelta che definire più architetture per la stessa porta NOR in modo da tener conto del numero di ingressi che si pilotano. Così, per esempio, con riferimento al listing 8, potremmo preparare più architetture con tempi di propagazione diversi e volta per volta, quando utilizziamo una porta nor come parte di una descrizione strutturale, potremmo scegliere l'architettura che corrisponde al ritardo corretto in base al numero di porte che si prevede di pilotare.

Supponiamo ora che si preveda di dover usare porte NOR a 2, a 3, a 4 o a 5 e più ingressi. In linea di principio dovremmo definire una entità distinta per ogni numero possibile di ingressi e, per ciascuna di esse, una architettura distinta per ogni possibile ritardo. Si osservi che il fatto di dover definire entità distinte per ogni diverso numero di ingressi è, per la verità, un fatto che appare assolutamente naturale: una nor a 2 ingressi è cosa diversa da una nor a 4 ingressi, e pertanto non si vede perché si debba voler vedere questo fatto come una limitazione o un problema. Tuttavia il fatto è che, soprattutto in una descrizione funzionale ad alto livello, è piuttosto scomodo dover fare riferimento a entità distinte che svolgono la medesima funzione su un numero diverso di ingressi. Questo non vale solo per le funzioni logiche: sarebbe per esempio estremamente comodo avere la possibilità di

```
architecture behav_1 of nor_port is
begin
        y<=a nor b after 5 ns;
end architecture behav_1;

architecture behav_2 of nor_port is
begin
        y<=a nor b after 10 ns;
end architecture behav_2;
.
.
.
architecture behav_13 of nor_port is
begin
        y<=a nor b after 65 ns;
end architecture behav_13;
.
.
.
```

Listing 8: Architetture per una porta nor. Si prevede una diversa architetture (diverso tempo di propagazione) a secondo del numero (supposto) di ingressi pilotati.

disporre di una unica entità che svolge la funzione di "sommatore su n bit" o "contatore a n bit" con la possibilità di specificare il valore "n" solo all'atto dell'utilizzo dell'entità all'interno di una specifica architettura. La possibilità di ottenere entità "programmabili" o, se si vuole "configurabili" in VHDL è demandata alla possibilità di definire l'entità e la sua relativa architettura in termini di parametri così detti "generic". La sintassi per la dichiarazione di entità nel caso in cui si vogliano usare parametri "generic" è la seguente:

```
entity_declaration<=
entity identifier is
        [generic (generic_list)];
        [ port (port_interface_list);]
        {entity_declarative_item}
end [entity][identifier];

generic_list <=(identifier{, ... }:subtype_indication[:=expression]) {;.....}
```

Listing 9: Sistassi per l'uso di generic

Per chiarire l'uso dei parametri "generic" facciamo riferimento a una situazione in cui vogliamo poter definire, all'atto dell'impiego di una porta nor all'interno di una architettura, sia il numero di ingressi, sia il ritardo di propagazione. Si noti che per ottenere ciò, nell'esempio che segue, gli ingressi delle porta NOR sono gli elementi di un vettore di tipo bit_vector.

Nella procedura che implementa la funzione NOR nel listing 10 abbiamo sfruttato il fatto che l'uscita di una NOR è '0' quando almeno uno degli ingressi è a '1'. Si noti l'uso della

```vhdl
entity nor_gen is
        generic (num_ingr:integer:=2; prop_delay:time:=5 ns);
        port (ingressi:in bit_vector(1 to num_ingr); y:out bit);
end entity nor_gen;

architecture behav of nor_gen is

begin

        process (ingressi)
        variable uscita:bit;

        begin
                uscita:='1';
                for i in 1 to num_ingr loop
                        if (ingressi(i)='1') then
                                uscita:='0';
                                exit;
                        end if;
                end loop;
        y<=uscita after prop_delay;
        end process;
end architecture behav;
```

Listing 10: Esempio di uso di costanti generiche

parola chiave `exit` per concludere il for—loop non appena si verifica che un ingresso è a 1. Si rifletta sul fatto che l'uso di `exit` è utile ai soli fini di ridurre i tempi di simulazione, ma che la sua assenza non modifica in alcun modo il comportamento descritto per la porta. Supponiamo ora di voler descrivere una entità a quattro ingressi e una uscita che esegue la funzione logica:

$$y = \overline{(\overline{x_1 + x_2}) + (\overline{x_2 + x_3 + x_4})} \tag{1.1}$$

Per implementare questa funzione sono necessarie 2 porte NOR a due ingressi e una porta NOR a tre ingressi. Supporremo, al solo fine di esemplificare l'uso dei parametri "generic", che le porte a due ingressi siano caratterizzate da un ritardo di propagazione di 5 ns mentre la porta a tre ingressi sia caratterizzata da un tempo di propagazione di 10 ns (questa ipotesi non ha nessuna base "fisica": è solo una scusa per fare un esempio con tempi di propagazione diversi).

Un esempio di definizione di entità e di architetura che consentono l'implementazione della funzione logica in VHDL con l'utilizzo di costanti generiche è riportato nel listing 11.

Si noti come non sia stato necessario usare esplicitamente alcun process.

1.5.2 Component e configuration

Come accennavamo all'inizio di queste note, possiamo essere interessati a scrivere la struttura di una entità in termini del collegamento di altre entità prima che queste siano state

```
entity comb_nor_gen is
        port (x_1,x_2,x_3,x_4:in bit; y:out bit);
end entity comb_nor_gen;

architecture struct of comb_nor_gen is
signal vect_input_3:bit_vector (1 to 3);
signal vect_input_2: bit_vector (1 to 2);
signal vect_internal_2:bit_vector (1 to 2);
begin
vect_input_3(1)<=x_1;
vect_input_3(2)<=x_2;
vect_input_3(3)<=x_3;
vect_input_2(1)<=x_1;
vect_input_2(1)<=x_2;
nor_3in:entity work.nor_gen
        generic map (num_ingr=>3, prop_delay=>10 ns)
        port map (ingressi=>vect_input_3, y=>vect_internal_2(1));
nor_2in_1:entity work.nor_gen
                generic map (num_ingr=>2, prop_delay=>5 ns)
                port map (ingressi=>vect_input_2, y=>vect_internal_2(2));
nor_2in_2:entity work.nor_gen
                generic map (num_ingr=>2, prop_delay=>5 ns)
                port map (ingressi=>vect_internal_2, y=>y);

end architecture struct;
```

Listing 11: Esempio di uso di costanti generiche nella descrizione di una funzione logica

effettivamente definite. In questo caso, all'interno della parte dichiarativa dell' architettura che vogliamo descrivere, potremo dichiarare la presunta esistenza di tali entità, definendo le porte di connessione e gli eventuali parametri generic che saranno poi effettivamente presenti allorché stabiliremo, mediante una successiva fase di configurazione, l'effettiva corrispondenza fra le entità presunte e quelle realmente presenti nel nostro progetto completo. La dichirazione dell'esistenza presunta di entità si ottiene mediante la parola chiave component con una sintassi del tutto simile alla dichiarazione di entità come riassunto nel listing 12.

```
component_declaration<=
component identifier is
        [generic (generic_list)];
        [ port (port_interface_list);]
end component [identifier];
```

Listing 12: Sintassi per la dichiarazione di un componente

Nell'esempio che segue supporremo di iniziare a lavorare in una directory vuota e di usare più file per la descrizione dell'intero progetto e per la simulazione del comportamento dell'entità principale. Per semplicità, faremo riferimento al progetto di generatore di clock a due fasi. Immaginiamo di voler descrivere la struttura tipo flip-flop che compare

nel generatore di clock a due fase con una architettura strutturale in termini di due componenti presunti: nor_port e inv_port.

Nel file two_phase.vhd possiamo scrivere quanto riportato nel listing 13

```
entity two_phase is
        port (ck:in bit; f1, f2:out bit);
end entity two_phase;

architecture struct of two_phase is

component nor_port is
        generic(prop_delay: time);
        port (a,b:in bit; y:out bit);
end component nor_port;

component inv_port is
        generic(prop_delay: time);
        port (a:in bit; y:out bit);
end component inv_port;

signal f1_int:bit;
signal f2_int:bit;
signal aa:bit;
begin

inv_1:component inv_port
        generic map (prop_delay=>5 ns)
        port map (a=>ck, y=>aa);

nor_1:component nor_port
        generic map(prop_delay=> 5 ns)
        port map (a=>ck,b=>f1_int,y=>f2_int);

nor_2:component nor_port
        generic map(prop_delay=> 5 ns)
        port map (a=>aa, b=>f2_int,y=>f1_int);
f1<=f1_int;
f2<=f2_int;
end architecture struct;
```

Listing 13: Descrizione di un generatore di clock a due fasi mediante componenti.

Il comando "`ghdl -a two_phase.vhd`" non produrrà alcun errore. Tuttavia, seproviamo a eseguire il comando "`ghdl -e two_phase`" nel tentativo di comporre tutto quanto serve a descrivere l'entità "two_phase" ai fini della successiva simulazione, riceviamo messaggi di warning che ci segnalano che ai componenti "nor_port" e "inv_port" non corrisponde nessuna architettura.

Procediamo allora inserendo nel file "comp.vhd" le definizioni delle entità di una porta NOR e di una porta "inverter" che poi, in un successivo momento, indicheremo come effettive implementazioni dei componenti utilizzati.

Nel file comp.vhd potremo scrivere quanto riportato nel listing 14

```
entity porta_nor is
        generic(prop_delay: time);
        port (a,b:in bit; y:out bit);
end entity porta_nor;

entity porta_inv is
        generic(prop_delay: time);
        port (a:in bit; y:out bit);
end entity porta_inv;

architecture behav of porta_inv is
begin
        y<=not a after prop_delay;
end architecture behav;

architecture behav of porta_nor is
begin
        y<=(a nor b) after prop_delay;
end architecture behav;
```

Listing 14: Descrizione delle entità da far cossrispondere ai componenti nel generatore di clock a due fasi.

Si presti attenzione al fatto che è necessario usare gli stessi nomi “formali” per gli identificatori delle porte e delle "generic" nella dichiarazione di una entità e del componente a cui essa si riferisce. Una volta eseguito il comando `ghdl -a comp.vhd`, la descrizione e l'architettura delle entità è contenuta nella libreria di default "work".

A questo punto è necessario ricorrere alla "configurazione" dell'entità "two_phase", ovvero è necessario specificare, per ogni componente l'entità che ne cosciuisce l'effettiva implementazione. Ciò può essere ottenuto grazie al costrutto "configuration". Per i dettagli circa questo costrutto si rimanda al manuale VHDL. In questa sede ci limiteremo a riporatre un esempio di file di configurazione per come può essere applicati al sistema che stiamo costruendo. Supponiamo di create il file "configurazion.vhd" come riportato nel listing 15.

In ossequio alle regole sintattiche, "test_configurazione" è il nome della configurazione dell'entità "two_phase" per la quale intendiamo fare riferimento all'architettura "struct".

Quando si esegue il comando "`ghdl -a`", le informazioni relative alla configurazione sono memorizzate nella libreria di default work. Si presti attenzione al fatto che, se si vuole usare ora l'entità "two_phase" all'interno di un test bench, la sua effettiva implementazione è indicata dall'identificatore usato per la configurazione, ovvero da "test_configurazione" nel nostro caso.

```
configuration test_configurazione of two_phase is
        for struct
                for nor_1,nor_2:nor_port
                        use entity work.porta_nor(behav);
                end for;
                for inv_1:inv_port
                        use entity work.porta_inv(behav);
                end for;
        end for;
end configuration test_configurazione;
```

Listing 15: Esempio di configurazione per la "costruzione" effettiva dell'entità che implementa il generatore di clock a due fasi.

Un possibile "test_bench" per la simulazione del generatore di clock a due fasi è riportato nel listing 16.

E' possibile avere diverse configurazioni per la stessa entità e per la stessa architettura, facendo corrispondere diverse entità effettive (o diverse architetture di queste) a ciascun componente.

Quando all'interno di una architettura sono impiegati componenti a cui corrispondono entità che impiegano a loro volta componenti e così via, è evidentemente necessario specificare la configurazione di tutti i componenti. I modi per ottenere questo risultato sono molteplici e il loro esame dettagliato esula dagli obbiettivi di questo corso.

Vale la pena di fare tuttavia almeno un esempio. Supponiamo che l'architettura "struct" dell'entità top_entity impieghi i componenti porta_a e porta_b. Supponiamo di voler far corrispondere al primo componente l'architettura "behav" della entità med_entity_a che al suo interno non impiega componenti. Supponiamo invece di voler far corrispondere al secondo componente l'architettura "struct_med" dell'entità med_entity_b che a sua volta impiega i componenti low_a e low_b che devono corrispondere alle architettura "behav" delle entità "low_entity_a" e "low_entity_b". Supponiamo che tutte le definizioni di componenti e architetture siano contenute nella libreria di default work.

Sia "esempio" l'identificatore della configurazione. Per ottenere la configurazione della entità top_entity potremo ricorrere a un file di configurazione come quello riportato nel listing 17.

1.5.3 Generate

Lo statement generate è usato in VHDL per inserire all'interno di una architettura un insieme di statement concorrenti in tutti quei casi in cui la struttura da implementare sia sufficientemente regolare da consentire di descrive la stessa in forma algoritmica. Gli statement concorrenti che possono essere inseriti grazie allo statement generate sono tipicamente process o componenti.

La sintassi di uno statement generate è riportato nel listing 18

```
entity test_bench is
end entity test_bench;
architecture behav of test_bench is
signal clock, fase1,fase2:bit;
begin
generatore: configuration work.test_configurazione
                        port map (ck=>clock, f1=>fase1, f2=>fase2);
simula:process
        begin
        clock<='0';
        wait for 20 ns;
        clock<='1';
        wait for 20 ns;
        clock<='0';
        wait for 20 ns;
        clock<='1';
        wait for 20 ns;
        clock<='0';
        wait for 20 ns;
        clock<='1';
        wait for 20 ns;
        clock<='0';
        wait;
end process simula ;
end architecture behav;
```

Listing 16: Esempio di test bench per la simulazione del generatore di clock a due fasi ottenuto mediante configurazione.

E' inoltre possibile che la generazione (ovvero l'inserimento all'interno dell'architettura) di uno statement concorrente sia condizionato al valore di una espressione di tipo boolean. In questo caso la sintassi diventa quella riportata nel listing 19.

Facciamo subito un esempio per chiarire l'uso dello statement generate. Supponiamo di voler descrivere l'entità dichiarata nel listing 20.

L'entità deve svolgere la funzione di addizionatore su enne bit. Supponiamo di volerne descrivere l'architettura in termini dell'interconnessione fra la "enne" component di nome "cfa" ciascuno dei quali si suppone implementi la funzione di un full adder. Supponiamo che il componente "cfa" sia definito come nel listing 21.

In questa situazione si può descrivere la struttura del sistema algoritmicamente come segue. Sia `index` un indice che rappresenta l'ordine dei bit degli ingressi `x_in` e `y_in`. Con lo stesso indice `index` possimao fare riferimento al `full_adder` (cfa) al quale sono collegati, come addendi, i bit di ordine `index` degli ingressi `x_in` e `y_in`. In una descrizione a "parole" potremmo descrivere la struttura del sommatore come segue:

per `index` che assume tutti i valori fra 0 e `(enne-1)`:

- se `index` è 0 allora agli ingressi del relativo full_adder vanno collegati `x_in(0)`,`y_in(0)` e `c_in`; alle uscite va collegato `s_out(0)` per quanto riguarda la somma, mentre all'u-

```
configuration esempio of top_entity is
        for struct
                for all : port_a
                        use entity work.med_entity_a(behav);
                end for;
                for all: port_b
                        use entity work.med_entity_b(med_struct);
                        for med_struct
                                for all:low_a
                                        use entity work.low_entity_a(behav);
                                end for;
                                for all: low_b
                                        use entity work.low_entity_b(behav);
                                end for;
                        end for;
                end for;
        end for;
end configuration esempio;
```

Listing 17: Esempio di ipotetica configurazione con più livelli di "for".

```
generate_label: for identifier in discrete_range generate
[{block_declarative_item} begin]
        {concurrent_statement}
end generate [generate_label];
```

Listing 18: Sintassi generate.

scita di carry `co` dobbiamo collegare un segnale che servirà a poter sfruttare questa uscita come ingresso di riporto per il full_adder di ordine 1;

- se `index` è `(enne-1)` allora agli ingressi del relativo full_adder vanno collegati `x_in(enne-1)`,`y_in(enne-1)` e il segnale al quale è collegato il riporto in uscita del full-adder precedente; alle uscite va collegato `s_out(enne-1)` per quanto riguarda la somma, mentre all'uscita di carry `co` dobbiamo collegare `c_out`;
- se `index` è diverso da `0` e da `(enne-1)` allora agli ingressi del relativo full_adder vanno collegati `x_in(i)`,`y_in(i)` e il segnale al quale è collegato il riporto in uscita del full-adder precedente; alle uscite va collegato `s_out(enne-1)` per quanto riguarda la somma, mentre all'uscita di carry `co` dobbiamo collegare un segnale che servirà a poter sfruttare questa uscita come ingresso di riporto per il full_adder di ordine `i+1`;

Lo statement "generate" consente di tradurre in codice VHDL la descrizione "a parole" sopra riportata.

Una architettura possibile per l'entità adder_enne_bit che fa uso dello statement generate è quella riportata nel listing 22.

Nell'architettura gen1, il segnale `carry_chain` è usato per collegare il riporto in uscita a ciascun full-adder con il riporto in ingresso al full-adder successivo.

```
generate_label: if boolean_expression generate
[{block_declarative_item} begin]
        {concurrent_statement}
end generate [generate_label];
```

Listing 19: Sintassi generate con inserimento condizionato.

```
entity adder_enne_bit is
        generic (enne:integer:=8);
        port (
                c_in:in bit;
                x_in:in bit_vector (enne-1 downto 0);
                y_in:in bit_vector (enne-1 downto 0);
                c_out:out bit;
                s_out:out bit_vector (enne-1 downto 0)
                );
end entity adder_enne_bit;
```

Listing 20: Dichiarazione di una entità sommatore.

Si noti che le label usate prima di ogni generate sono necessarie affinché sia possibile eseguire la corretta configurazione delle celle. Si noti anche che abbiamo usato la label `"cella"` al momento di inserire ciascun full-adder. A questo proposito si tenga presente che la prima "`cella`" è identificata internamente al VHDL in quando facente parte dall'entità adder_enne_bit, dell'architettura `gen1`, del blocco generate `gen_stat` e del blocco generate `prima_cella` (oltre che dal fatto che viene "generata" in corrispondenza dell'indice 0). Potremmo quindi, usando una sintassi arbitraria che non ha nulla a che vedere con il VHDL ma che può rappresentare un utile ausilio mnemonico, con la stringa:

adder_enne_bit::gen1::gen_stat(0)::prima_cella::cella

L'ultima cella sarebbe identificata da (supponiamo enne=8):

adder_enne_bit::gen1::gen_stat(7)::ultima_cella::cella

Le altre 6 celle sarebbero identificate da:

adder_enne_bit::gen1::gen_stat(1)::altre_celle::cella

adder_enne_bit::gen1::gen_stat(2)::altre_celle::cella

.

adder_enne_bit::gen1::gen_stat(6)::altre_celle::cella

In sostanza, grazie all'uso delle label e al fatto che in uno statement generate eseguito con un ciclo for si tiene conto automaticamente del valore dell'indice nel momento in cui viene generata una cella, tutte i componenti "cella" sono univocamente determinati.

Alle label usate nei blocchi generate si fa infatti riferimento al momento della cofigurazione. Supponiamo infatti che sia definita l' entità `full_adder` (alla quale si faranno corrispondere i componenti cfa) come nel listing 23.

```
component cfa is
        port        (
                a:in bit;
                b:in bit;
                ci:in bit;
                s:out bit;
                co:out bit
                );
end component cfa;
```

Listing 21: Dichiarazione del componente "cfa". Se si vuole usare il componente "cfa", questa dichiarazione deve essere inserita nella parte dichiarativa di una architettura.

A questo punto una possibile configurazione della entità `adder_enne_bit` è riporatta nel listing 24.

Supponiamo ora di voler verificare il comportamento dell'entità appena definita (e configurata) mediante un testbench. Supponiamo di voler fissare enne a 4. Un possibile testbench sritto a questo scopo è riportato nel listing 25.

All'interno di blocchi generate possono essere definiti dei segnali che possono essere utilizzati per collegare fra loro i componenti che vengono generati. Supponiamo per esempio di voler riscrivere l'architettura del sommatore a enne bit usando degli half adder per eseguire la somma dei signoli bit.

L'architettura `gen2` di `adder_enne_bit` potrebbe essere quella riportata nel listing 26.

Si noti che, grazie al fatto di aver definito `carry_chain` di dimensione 8, è stato possibile semplificare lo statement generate in quanto sono ora distinguibili due soli casi (vedi ultimo statement concorrente nell'architettura). Sarebbe possibile, definendo `carry_chain` di dimensione 9 e usando carry_chain(i) come ingresso di carry per la cella i e `carry_chain(i+1)` come uscita di carry per la cella i, eliminare completamente la distinzione fra le diverse celle (basterebbe aggiungere lo statement `carry_chain(0)<=c_in` all'interno dell'architettura.

Si noti in particolare che sono generate enne triplette distinte di segnali `sig1`,`sig2` e `sig3`. Infatti ciascuna tripletta di segnali è relativa a ciascuna singola iterazione di generazione e distinta dalle triplette della altre generazioni. Usando una sintassi arbitraria, come già fatto precendentemente, è come se si fossero definiti enne distinti segnali sig1, ovvero:

adder_enne_bit::gen1::gen_stat(0)::sig1

adder_enne_bit::gen1::gen_stat(1)::sig1

.

adder_enne_bit::gen1::gen_stat(enne-1)::sig1

Supponendo che siano state definite delle opportune entità per l'implementazione degli `half_adder` e della porta OR, il file di configurazione potrebbe essere quello riportato nel listing 27.

```
architecture gen1 of adder_enne_bit is
signal carry_chain: bit_vector (enne-2 downto 0);
component cfa is
        port         (
                a:in bit;
                b:in bit;
                ci:in bit;
                s:out bit;
                co:out bit
                );
end component cfa;
begin
    gen_stat:for index in 0 to (enne-1) generate
                begin
                prima_cella:if (index=0)  generate
                        begin
                        cella: component cfa
                               port map (a=>x_in(0),b=>y_in(0),ci=>c_in,s=>s_out(0),co=
                        end generate prima_cella;
                ultima_cella:if (index=(enne-1)) generate
                        cella: component cfa
                               port map (a=>x_in(enne-1),b=>y_in(enne-1),ci=>carry_chai
                        end generate ultima_cella;
                altre_celle:if ((index/=0) and (index/=(enne-1))) generate
                        cella: component cfa
                               port map (a=>x_in(index),b=>y_in(index),ci=>carry_chain(
                        end generate altre_celle;
                end generate gen_stat;
end architecture gen1;
```

Listing 22: Architettura del full addere a "enne bit" ottenuta mediante descrizione strutturale con statement "generate".

```vhdl
entity full_adder is
	port(
		a:in bit;
		b:in bit;
		ci:in bit;
		s:out bit;
		co:out bit
		);
end entity full_adder;

architecture behav of full_adder is
begin
	s<=a xor b xor ci after 10 ns;
	co<=(a and b) or (a and ci) or (b and ci) after 5 ns;
end architecture behav;
```

Listing 23: Entità e architettura per il full_adder per l'implementazione del componente cfa.

```vhdl
configuration sommatore of adder_enne_bit is
	for gen1
		for gen_stat
			for prima_cella
				for cella:cfa
					use entity work.full_adder(behav);
				end for;
			end for;

			for ultima_cella
				for cella:cfa
					use entity work.full_adder(behav);
				end for;
			end for;

			for altre_celle
				for cella:cfa
					use entity work.full_adder(behav);
				end for;
			end for;
		end for;
	end for;
end configuration sommatore;
```

Listing 24: Possibile configurazione dell'entità adder_enne_bit.

```
entity testbench is
end entity testbench;

architecture behav of testbench is
signal add1,add2,somma: bit_vector (3 downto 0);
signal riporto: bit;
begin
sommatore4: configuration work.sommatore
                generic map (enne=>4)
                port map (x_in=>add1,y_in=>add2,c_in=>'0', s_out=>somma, c_out=>riporto)
process is
begin
add1<="0000";
add2<="0000";
wait for 100 ns;
add1<="0001";
add2<="0001";
wait for 100 ns;
add1<="0010";
add2<="0001";
wait for 100 ns;
add1<="0010";
add2<="0010";
wait for 100 ns;
add1<="0011";
add2<="0011";
wait for 100 ns;
add1<="0100";
add2<="0100";
wait for 100 ns;
add1<="0101";
add2<="0101";
wait;
end process;
end architecture behav;
```

Listing 25: Possibile testbench per la verifica dell'entità `adder_enne_bit`.

```
architecture gen2 of adder_enne_bit is
signal carry_chain: bit_vector (enne-1 downto 0);
component ha is
        port        (
                a:in bit;
                b:in bit;
                s:out bit;
                c:out bit
                );
end component ha;
component cor is
        port        (
                a:in bit;
                b:in bit;
                c:out bit
                );
end component cor;
begin
        gen_stat:for index in 0 to (enne-1) generate
                signal sig1, sig2,sig3:bit;
                begin
                prima_cella:if (index=0)  generate
                        begin
                        cella1: component ha
                                port map (a=>x_in(0),b=>y_in(0),s=>sig1,c=>sig2);
                        cella2:  component ha
                                port map (a=>c_in,b=>sig1,s=>s_out(0), c=>sig3) ;
                        cella3:  component cor
                                port map (a=>sig3,b=>sig2,c=>carry_chain(0));
                        end generate prima_cella;
                altre_celle:if (index/=0)  generate
                        cella1: component ha
                                port map (a=>x_in(index),b=>y_in(index),s=>sig1,c=>sig2)
                        cella2:  component ha
                                port map (a=>carry_chain(index-1),b=>sig1,s=>s_out(index
                        cella3:  component cor
                                port map (a=>sig3,b=>sig2,c=>carry_chain(index));
                        end generate altre_celle;
                end generate gen_stat;
        c_out<=carry_chain(enne-1);
end architecture gen2;
```

Listing 26: Architettura alternativa per `adder_enne_bit` che dimostra come si possano utilizzare segnali definiti nei blocchi generate.

```
configuration sommatore of adder_enne_bit is
        for gen2
                for gen_stat
                        for prima_cella
                                for all:ha
                                        use entity work.half_adder (behav);
                                end for;
                                for all:cor
                                        use entity work.orport(behav);
                                end for;

                        end for;


                        for altre_celle
                                for all:ha
                                        use entity work.half_adder (behav);
                                end for;
                                for all:cor
                                        use entity work.orport(behav);
                                end for;
                        end for;
                end for;
        end for;
end configuration sommatore;
```

Listing 27: Architettura per la realizzazione di un adder a enne bit che utilizza half-adder e segnali definiti all'interno di blocchi generate.