Università degli Studi di Messina
Dipartimento di Ingegneria

Introduzione al progetto di sistemi digitali mediante VHDL

Ultima modifica 13 ottobre 2021

Author
Carmine Ciofi

13 ottobre 2021

Indice

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).

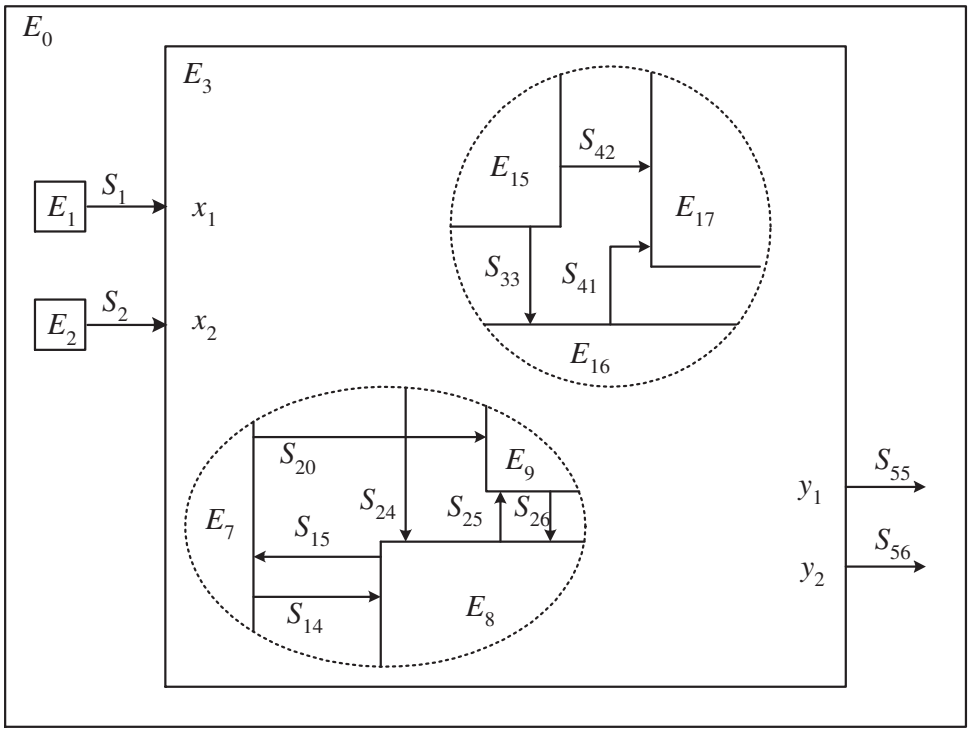

Figura 1.1: Sistema digitale generico.

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

```vhdl
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.