PARTE SECONDA

Introduzione al linguaggio VHDL

Autori:

Alessandro Giorgetti

La presente parte è organizzata nei seguenti capitoli:

Capitolo 2. Introduzione al linguaggio VHDL

Capitolo 2. Introduzione al linguaggio VHDL

2.1. Come nasce

Con lo svilupparsi delle tecnologie che hanno permesso un sempre maggiore aumento della complessità e delle dimensioni (inteso come numero di componenti integrabili) dei sistemi digitali, si è fatta sempre più pressante l'esigenza di poter disporre di una serie di metodi per facilitare la progettazione e la descrizione di sistemi sempre più complessi.

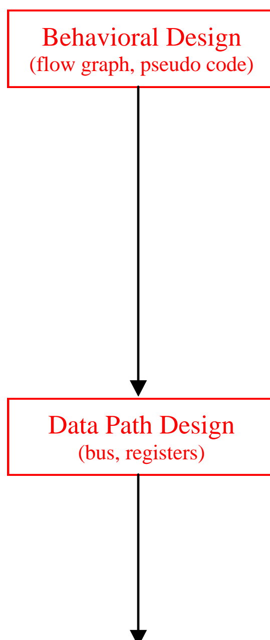
Si sono allora sviluppati nel tempo (soprattutto negli ultimi 20-30 anni) una serie di meta-linguaggi il cui scopo è quello di fornire una descrizione più o meno dettagliata del progetto che si intende realizzare.

All'inizio molti di questi linguaggi (raggruppati sotto il termine generico di HDL: Hardware Description Language) si svilupparono nelle università (a scopo didattico) o all'interno di note aziende (es. IBM) per lo sviluppo di nuovi progetti.

Il VHDL nasce dall'esigenza di sviluppare un tool standard per la progettazione, la sintesi, il testing e la documentazione dei sistemi digitali. In modo particolare il VHDL nasce dalla richiesta, da parte del ministero della difesa americano, di definire un linguaggio HDL standard per lo sviluppo del loro programma VHSIC (Very High Speed Integrated Circuits) e nel 1983 vengono fissati i requisiti per il VHSIC Hardware Description Language (ovvero VHDL).

2.2. A cosa serve: metodologia di progetto di un sistema digitale

La metodologia di progetto che si è sviluppata in questi anni si articola in una serie di passi ben definiti, riassunti dal seguente grafico.



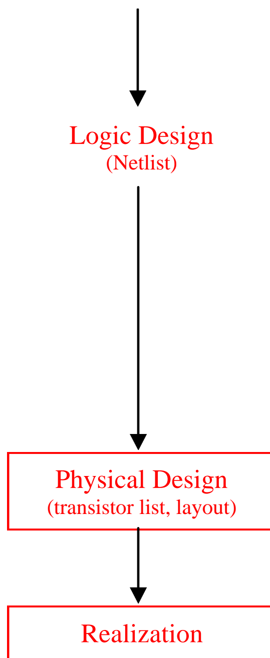
- La prima fase è la definizione ad alto livello del comportamento del sistema. Quello che si produce è un diagramma di flusso del funzionamento del sistema o dello pseudo-codice che ne descrive le funzionalità. Viene fornita una rappresentazione complessiva delle funzionalità e del mapping degli ingressi e delle uscite. Non si forniscono dettagli riguardo l'architettura o all'hardware da impiegare.

  Questo livello di descrizione del progetto si adatta ad una simulazione veloce di unità hardware molto complesse, alla modellazione di componenti standard e viene utilizzato anche come parte della documentazione dell'intero progetto.

- La fase successiva è la definizione del "data path", ovvero la specifica dei registri e della logica necessaria per l'implementazione. Si realizza un diagramma che illustra come i vari componenti dialoghino tra loro mediante una serie di percorsi dati (i bus di sistema); viene inoltre specificato l'hardware per la logica ed i segnali di controllo[9].

  La simulazione del progetto a questo stadio, essendo maggiormente

[9] Per questo si parla anche di Dataflow Design (riferendosi alla descrizione del flusso di dati) oppure di Structural Design (riferendosi alla descrizione di un componente in termini della interconnessione dei suoi elementi).



particolareggiata, richiede un tempo maggiore di quanto richiesto per la descrizione behavioral.

- La terza fase prevede l'implementazione a livello di porte logiche dei registri, dei bus e della logica di controllo derivata dal passo precedente. Il risultato di questa fase è una Netlist dei dispositivi impiegati e delle relative interconnessioni.

  Nella Netlist sono contenute dettagliatissime informazioni riguardanti l'hardware da realizzare, di conseguenza la simulazione del sistema a questo livello garantisce risultati molto accurati (soprattutto per il timing dei dispositivi) al costo del maggiore tempo impiegato per portare a termine la simulazione stessa.
- L'ultima fase di progetto prevede la trasformazione della Netlist nel layout finale del sistema. Questa fase prevede la sostituzione delle porte logiche e dei flip-flop risultanti dal passo precedente con le corrispondenti realizzazioni a transistor.
- Ovviamente il passo finale del progetto è la realizzazione "fisica" del dispositivo.

Questo sarà anche l'approccio che verrà seguito nel seguito del presente testo per lo sviluppo dei processori *Vertex Shader* e *Pixel Shader*.

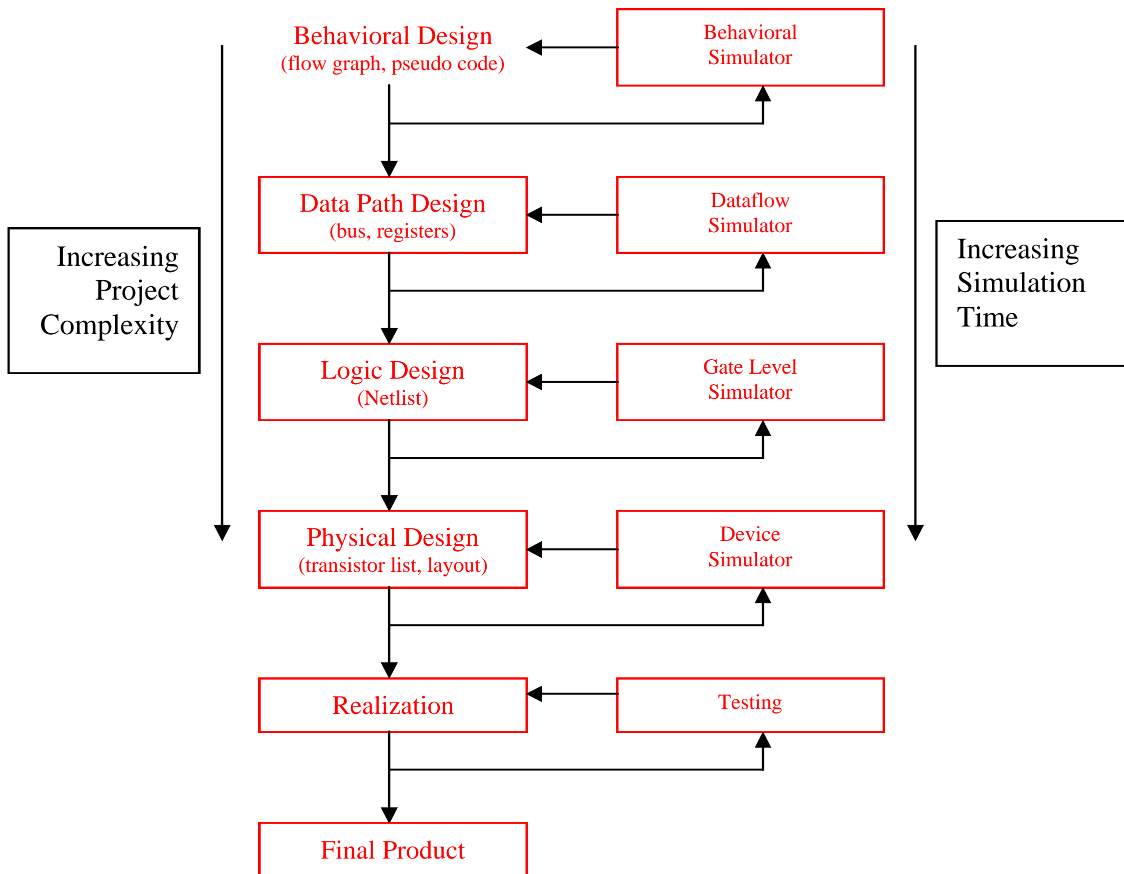
Dal punto di vista puramente progettuale il lavoro può dirsi concluso una volta realizzate le prime due fasi; il resto può essere svolto da strumenti software realizzati allo scopo di semplificare il lavoro del progettista (si parla di Design Automation). Così un programma di simulazione e di testing è utile per la verifica della correttezza del progetto, mentre un programma di sintesi si occupa della generazione automatica dell'hardware.

All'inizio vennero sviluppati tutta una serie di linguaggi HDL specifici per ognuno dei differenti stadi del processo di progettazione. In seguito la necessità di standardizzare e semplificare spinse il ministero della difesa a sviluppare un unico linguaggio (il VHDL appunto) che raggruppasse in sé le caratteristiche necessarie per descrivere un sistema sia da un punto di vista puramente astratto (ovvero definendone il comportamento) che da quello della realizzazione a porte logiche del progetto, consentendo che differenti blocchi dello schema, descritti ad un livello differente l'uno dall'altro, potessero dialogare tra loro ed essere simulati e testati assieme.

2.3. Differenti tipi di simulazioni

La possibilità di simulare un sistema mentre alcune sue parti sono ancora in via di sviluppo (ovvero sono ancora descritte a livello comportamentale e non a livello di layout) si paga ovviamente in termini di accuratezza del risultato ottenuto. Minori sono le informazioni sull'hardware che vengono fornite al simulatore (come ad esempio la mancanza dei tempi di risposta dei componenti) più grossolano ed affetto da errori sarà il risultato.

Ad ogni fase di progetto corrisponde quindi un diverso simulatore, come visibile dal grafo seguente.



Come abbiamo già detto i differenti tipi di simulatore possono "collaborare" tra loro e fornire una simulazione complessiva del progetto anche se alcune sue porzioni vengono descritte ad un livello più elevato di altre.

2.4. Sistemi di Hardware Synthesis

Oltre ai normali simulatori ed ai CAD di ausilio nella progettazione, un'altra categoria di strumenti completa il quadro: i tool di sintesi dell'hardware. Questi sistemi entrano in gioco nella terza fase della progettazione e dallo schema del data path producono la netlist del circuito, oppure, per quei tool specificamente realizzati per il mercato delle FPGA, un file con cui programmare la specifica FPGA per cui è stato compilato il progetto[10].

Questi programmi sono realizzati per effettuare una ottimizzazione della logica del progetto, tenendo conto della possibile condivisione delle risorse e dell'associazione di particolari operazioni con delle celle predefinite della FPGA di destinazione.

Altri compilatori (detti compilatori di silicio) operano direttamente sulla netlist per ottenere il layout del circuito che verrà poi inviato alle fonderie competenti. Questi strumenti non hanno ovviamente impiego nel campo delle FPGA.

---

[10] Molto spesso infatti risulta conveniente implementare un progetto complesso su di una FPGA, in modo da testarne le funzionalità e rilevare bug di progettazione, prima di realizzare il chip vero e proprio.

2.5. Prerequisiti e caratteristiche del linguaggio

Nei documenti del ministero della difesa americano, che stabiliscono le caratteristiche e gli obiettivi del VHDL, si enfatizza molto il concetto che tale linguaggio deve servire per la descrizione del sistema "dal progetto iniziale" fino "alla porta logica". Inoltre si pone particolare attenzione all'aspetto "concorrenziale" del sistema che si andrà a descrivere (inteso nel senso che molti blocchi del sistema saranno attivi e opereranno contemporaneamente interagendo l'uno con l'altro… non come in un normale linguaggio di programmazione, in cui le istruzioni di un singolo blocco vengono eseguite sequenzialmente ed in maniera isolata rispetto ad unità funzionali con cui questo potrebbe interagire[11]).

Eseguire una simulazione in maniera "concorrente" significa quindi che al termine del processo il risultato complessivo apparirà quello dell'azione contemporanea di tutti i blocchi, anche se da un punto di vista computazionale le simulazioni di ogni blocco saranno riorganizzate ed eseguite in modo indipendente l'una dalle altre.

Questo comporta la possibilità di specificare il "timing" ad ogni livello del design e la necessità per il progettista di definire liberamente un proprio schema di "clocking" per l'hardware che si sta realizzando. Il linguaggio non dovrà allora essere dotato di un proprio schema implicito per la gestione del clock, ma dovrà fornire una serie di costrutti per la specifica dei delay, i tempi di setup e di hold dei segnali, la rilevazione di picco, ecc…

In fase di progetto una delle caratteristiche principali richieste al linguaggio è la scalabilità e la possibilità di descrivere l'hardware in maniera gerarchica. Ogni singolo componente del progetto consisterà quindi in due sezioni distinte: una parte di dichiarazione dell'interfaccia ed una parte che specifichi le operazioni che vengono eseguite dal componente, se la descrizione è di tipo comportamentale, oppure che ne definisca la struttura. Tutto questo in perfetta simmetria con la classica metodologia di progetto di tipo Top-Down.

Da quanto detto si evince anche la necessità di poter gestire il "flusso di controllo" del progetto in maniera sequenziale (e non solo concorrente), utile soprattutto per le descrizioni a livello comportamentale degli elementi[12].

Una ulteriore conseguenza è la necessità dell'impiego di librerie di componenti precompilate e disponibili, solitamente fornite dai produttori dei componenti o delle fonderie di silicio che realizzano i chip sulla base delle tecnologie adottate e dei processi produttivi[13]. Le librerie risultano essenziali per lo sviluppo di progetti ad alto livello, in quanto consentono il riutilizzo di componenti già realizzati da altri e risparmiano al progettista il tempo che si impiegherebbe a "reinventare la ruota".

Per garantire inoltre un elevato livello di astrazione e facilitare il progetto, il linguaggio non deve essere limitato alla sola tipizzazione dei dati effettuata in termini di bit o di valori booleani. Il VHDL richiede l'utilizzo ti tipi di dato

---

[11] Anche se tutto questo discorso cade se si parla di programmazione multitasking.

[12] Anche il pensiero umano non è in grado di operare in maniera concorrente (almeno a livello conscio), ma segue schemi sequenziali per descrivere ad esempio una serie di operazioni correlate le une alle altre.

[13] Sicuramente la modellazione del componente sarà più accurata se effettuata dal produttore che, mediante testing del dispositivo realizzato, sarà in grado di determinare con esattezza i tempi di risposta ed i ritardi, piuttosto che basarsi su una stima di comportamento generico che potrebbe essere fatta dal progettista.

assai differenziati: integer, floating-point, tipi enumerati, tipi aggregati, così come tipi di dato definibili dall'utente. Il linguaggio dovrà allora essere fortemente tipizzato e caratterizzato da un "robusto" type-checking per garantire la coerenza dei dati (in modo particolare quelli interscambiati dai vari componenti). Il documento rilasciato dal ministero della difesa richiedeva inoltre la capacità del linguaggio di poter ridefinire gli operatori associati con i differenti tipi di dato[14].

2.6. Packages & Libraries

Consideriamo ora la seguente situazione: durante lo sviluppo del nostro progetto abbiamo avuto la necessità di sviluppare un componente che effettuasse l'addizione tra due numeri binari.

Per semplificare l'utilizzo di questo componente e per non complicare in maniera eccessiva il layout del progetto il VHDL ci fornisce la possibilità di poter includere in un singolo modulo (o package) lo schema appena elaborato, anche in vista di un possibile utilizzo futuro.

Se la descrizione del componente era completa (ovvero fino al livello delle porte logiche) l'intero package potrà venire trasposto in una netlist e quindi elaborato dai compilatori di silicio.

Inoltre si ha la possibilità di raggruppare differenti package in una libreria, (generalmente si raggruppano assieme elementi dalle funzionalità simili), in modo da rendere disponibile la descrizione del componente anche a progetti diversi da quello iniziale.

2.7. Progettazione Top-Down, Realizzazione Bottom-Up

Per la realizzazione di un progetto si procede nel seguente modo: durante la fase iniziale si stabiliscono le specifiche e le funzionalità che si desidera implementare; si procede quindi all'analisi del progetto ed alla sua ricorsiva ripartizione in unità funzionali che potranno a loro volta essere ulteriormente frazionate fino alla identificazione di semplici blocchi funzionali facilmente implementabili.

Questo metodo di procedere (lo scomporre il problema iniziale in sottoproblemi più semplici) è detto metodo Top-Down.

Il passo successivo consiste nella sintesi dei singoli blocchi funzionali appena identificati. La loro realizzazione a questo punto sarà senza dubbio più semplice che dover implementare direttamente l'intero sistema di partenza. In questo caso si segue una metodologia di tipo Bottom-Up.

Attenzione, il processo non è comunque così immediato come sembra: un corretto modo di procedere implica lo sviluppo di una descrizione iniziale del comportamento del sistema (behavioral description) che andrà simulata per verificare la correttezza del progetto. Ad ogni successiva ripartizione del dispositivo in sotto elementi dovranno essere sviluppate e simulate delle adeguate rappresentazioni per questi ultimi, inoltre andranno opportunamente "collegate" tra loro a formare un modello strutturale dell'hardware.

Una ulteriore simulazione di questo modello ed un raffronto con i dati

[14] Sconvolgente la somiglianza con il C++ ed i linguaggi orientati agli oggetti in generale.

delle simulazioni precedenti dimostreranno o meno la correttezza della scomposizione del modello.

Il procedimento prosegue in maniera iterativa fino a quando ogni singolo elemento non viene rimpiazzato con dei modelli dettagliati (a livello di porta logica) dell'hardware[15].

Mano a mano che si procede con lo sviluppo si noterà che i modelli più sofisticati, pur mantenendo le stesse funzionalità del behavioural, presenteranno delle differenze per quanto riguarda le temporizzazioni dei segnali (timing). Poiché la simulazione di modelli molto dettagliati è parecchio dispendiosa in termini di tempo (si necessita comunque della maggior precisione possibile), se occorre simulare dei progetti molto grandi (formati da più componenti complicati ormai descritti a livello logico) si possono "aggiustare" i modelli behavioral tenendo conto dei risultati ottenuti dai livelli più dettagliati. Ci si riferisce a questo modo di procedere come ad una *Back Annotation.*

2.8. Elementi del linguaggio[16]

Commenti: in un sorgente scritto in VHDL i commenti al codice vengono identificati dal simbolo " -- " (doppio trattino); tutto quanto segue questo simbolo fino al termine della riga viene ignorato dal parser del compilatore.

Tipi di Dato: alcuni tipi che il linguaggio mette a disposizione sono: BIT, BOOLEAN[17], INTEGER, REAL, tipi aggregati, tipi enumerati, tipi di dato che rappresentano quantità fisiche (ad esempio le unità di tempo), UDT (User Defined Types, tipi definiti dall'utente), ecc...

Operatori (presentati in ordine crescente di precedenza):

	Operators	Operand type	Result type
Logical	AND, OR, NAND, NOR, XOR, XNOR	BIT or BOOLEAN	BIT or BOOLEAN
Relational	=, /=, <, <=, >=, >	All	BOOLEAN
Shift	SLL, SRL, SLA, SRA, ROL, ROR	Left: BIT or BOOLEAN vector Right: INTEGER	BOOLEAN
Adding	+, -, &	Numeric Array or Array Element	Same type
Sign	+, -	Numeric	Same type
Multiplying	*, /	INTEGER, REAL	Same type
	MOD, REM	INTEGER	Same type
Miscellaneous	ABS	Numeric	Same type
	**	Left: Numeric Right: INTEGER	Same as Left

[15] A questo punto vengono in aiuto le librerie fornite dai produttori di componenti: l'elemento che ci interessa sintetizzare potrebbe già essere disponibile!

[16] Per una descrizione completa del linguaggio si rimanda a testi specialistici, come ad esempio "VHDL Cookbook" (scaricabile gratuitamente dalla rete) e "VHDL: analysis and modeling of digital systems" edito da McGraw-Hill; entrambi fanno riferimento allo standard del 1993.

[17] Equivalente a BIT, per ovvi motivi.

Oggetto: un oggetto in VHDL è una entità (generalmente contrassegnata da un nome identificativo) specificato da un tipo di dato a cui viene inoltre assegnato un valore. Gli oggetti vengono raggruppati in tre categorie distinte: le **costanti**, le **variabili** ed i **segnali**.

Le dichiarazioni sono le seguenti:

- `CONSTANT nome : tipo [:= valore]`[18]
- `VARIABLE nome : tipo [:=valore]`
- `SIGNAL nome : tipo [REGISTER|BUS`[19]`] [:=valore]`

Le **variabili** vengono utilizzate localmente alle procedure ed alle funzioni ed esclusivamente all'interno di blocchi ad elaborazione sequenziale. I **segnali** possono essere utilizzati sia nei blocchi sequenziali che in quelli concorrenti, inoltre sono il solo modo con cui è possibile scambiare informazioni tra processi ed unità di elaborazione separate tra loro.

Attributo: i tipi di dato e gli oggetti possono inoltre essere caratterizzati da delle informazioni aggiuntive, dette attributi. Tutta una serie di attributi standard fa già parte del linguaggio e caratterizza i differenti tipi di dato, altri possono essere aggiunti dall'utente. La sintassi per accedere ad un attributo fa uso del carattere " ' " (apice) nel seguente modo:

`oggetto'attributo`[(n)]

In caso di tipi di dato aggregati o enumerati è possibile anche accedere all'attributo dell'elemento specificato dall'indice.

Per un elenco completo degli attributi e del relativo significato si consulti un manuale di VHDL.

Come in ogni linguaggio di programmazione si hanno inoltre espressioni che consentono di modificare lo stato degli oggetti: si parla di **assegnazioni** (indicate con il simbolo **:=** per le variabili, con il simbolo **<=** per i segnali[20]).

Si hanno inoltre istruzioni per il controllo del flusso dell'esecuzione, come ad esempio le istruzioni: **if…then…else…end if**, **case…end case**, **while…loop..end loop**, **for…loop…end loop**, ecc… Per la sintassi esatta ed il corretto utilizzo di queste istruzioni si rimanda a testi specializzati (od al codice presentato nelle sezioni seguenti).

Ulteriori elementi verranno presentati di volta in volta a seconda delle necessità.

2.9. I tipi di dato nel dettaglio

Si intende qui fornire una panoramica sui principali tipi di dato presenti nel VHDL e illustrare le istruzioni che consentono di definirne di nuovi.

E' possibile definire un nuovo tipo di dato ricorrendo all'istruzione TYPE … IS, il cui funzionamento verrà illustrato mediante esempi nei paragrafi seguenti.

---

[18] Le espressioni tra […] indicano elementi facoltativi.

[19] Questi due ulteriori specificatori di tipo assumono particolare importanza, come vedremo in seguito, nell'assegnamento di valori NULL ai segnali in esame.

[20] Ad eccezione del valore di default dato nell'inizializzazione, quando si ricorre a := anche per i segnali.

2.9.1 Integer & Real

Sono analoghi a quelli presenti in qualsiasi linguaggio di programmazione; ovviamente le quantità rappresentate dipendono dall'implementazione hardware della macchina su cui si opera.

2.9.2 Tipi enumerati

Il tipo enumerato è un esempio di UDT e deve essere dotato, se necessario, di apposite funzioni di risoluzione, in quanto le funzioni e gli operatori standard del VHDL non si applicano al nuovo tipo definito.

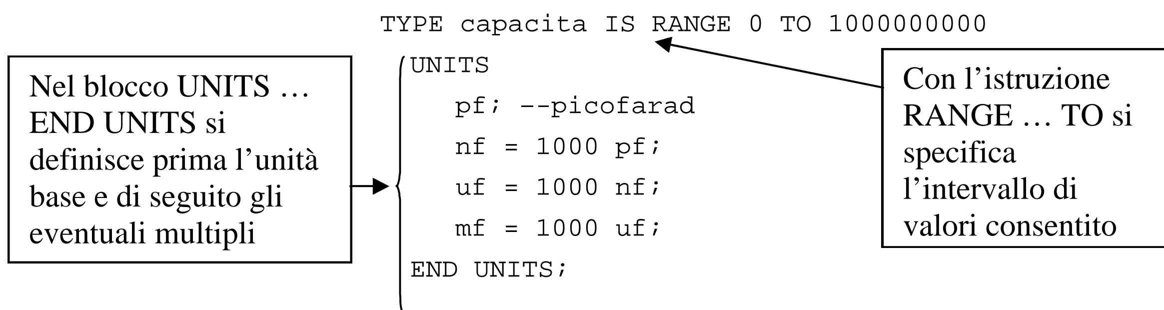
Un esempio di tipo enumerato:

```
TYPE enum IS (stato1, stato2, stato3);
```

2.9.3 Tipi che rappresentano entità fisiche

Esempi di questo tipo di dato sono il tempo, la massa, ecc… Occorre dichiarare l'unità base e calcolare tutti i multipli a partire da questa.

Ad esempio definiamo un tipo di dato utile ad indicare la capacità di un condensatore:



2.9.4 Tipo aggregato – ARRAY

Un array è un raggruppamento di elementi tutti dello stesso tipo; l'accesso ad ogni singolo elemento avviene mediante indicizzazione. Gli array sono particolarmente utili per la modellazione di RAM e ROM. Per definire un array si ricorre alla parola chiave ARRAY, indicando poi tra parentesi tonde il range di valori dell'indice.

Come esempio definiamo un nuovo tipo costituito da un array di bit:

- `TYPE bitArray IS ARRAY (7 DOWNTO 0) OF BIT;`

  bitArray è un array di 8 bit i cui indici vanno da 7 a 0.
- `TYPE bitArray IS ARRAY (0 TO 7) OF BIT;`

  bitArray è un array di 8 bit i cui indici vanno da 0 a 7.
- `TYPE bitArray IS ARRAY (INTEGER RANGE <>) OF BIT;`

  In questo caso abbiamo definito un tipo di array di bit senza alcuna

costrizione; il range dovrà essere deciso in fase di istanziazione dell'array.

2.9.5 Tipo aggregato – RECORD

Un record è un raggruppamento di differenti tipi di oggetti; chi è esperto nella programmazione in C/C++ può pensare al record come ad una **struct** priva di costruttore e di funzioni membro, ma dotata solamente di membri dato[21]. Un record viene definito all'interno del blocco RECORD … END RECORD, come nel seguente esempio:

```
TYPE instruction IS
  RECORD
     OpCode : BIT_VECTOR (7 DOWNTO 0);
     Data : BIT_VECTOR (15 DOWNTO 0);
  END RECORD;
```

2.9.6 Subtype

Un sottotipo (subtype) è un sottoinsieme del tipo base; sono consentiti assegnamenti di valori tra il tipo base ed il sottotipo definito. Per definire un sottotipo utilizziamo la parola chiave SUBTYPE al posto di TYPE:

```
SUBTYPE alcuniNumeri IS INTEGER RANGE 0 to 100;
```

2.9.7 Alias

Un Alias è una denominazione alternativa per un segnale, una variabile od una costante. Gli alias si rivelano molto utili per accedere a determinate porzioni di un vettore, come visibile dall'esempio seguente:

```
SIGNAL instruction : BIT_VECTOR (15 DOWNTO 0);
ALIAS opCode : BIT_VECTOR (7 DOWNTO 0) IS instruction
(15 DOWNTO 8);
ALIAS data : BIT_VECTOR(7 DOWNTO 0) IS instruction (7
DOWNTO 0);
```

2.9.8 Puntatori – ACCESS TYPE

In VHDL possiamo definire tipi di dato che non hanno una diretta corrispondenza con l'hardware, ma che (come i puntatori in C/C++) vengono utilizzati per manipolare dati. La sintassi per definire un tipo di dato che punti ad un tipo base è la seguente:

```
TYPE pointerToInteger IS ACCESS INTEGER;
```

---

[21] Membri public per altro!

In questo caso pointerToInteger è un puntatore ad intero.

2.9.9 File types

Esistono inoltre dei tipi di dato per la manipolazione di file sia in lettura che in scrittura, ma verranno introdotti in seguito quando sarà richiesto il loro utilizzo.

2.10. Descrizione dei componenti

Per descrivere completamente un componente si deve:

- Fornire una dichiarazione che specifichi l'interfaccia del componente stesso.
- Fornirne una descrizione architetturale.
- Collegare l'interfaccia ad una determinata architettura.

2.10.1 ENTITY

La prima fase della dichiarazione avviene ricorrendo alla parola chiave ENTITY[22] (entità), seguita dalla dichiarazione delle porte di input/output (che specificano inoltre i tipi di dato coinvolti) e da eventuali parametri che ne descrivono le dipendenze da fattori di natura fisica (la temperatura, le temporizzazioni, ecc…). Facciamo un esempio:

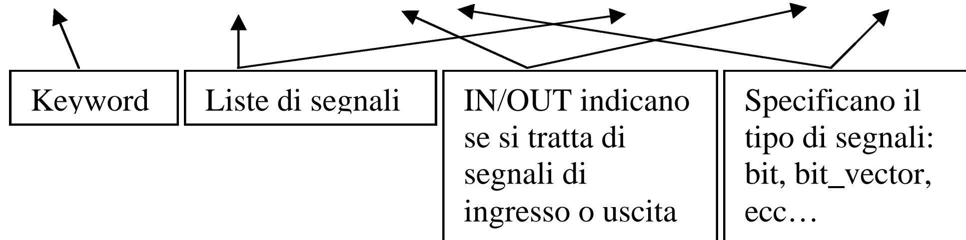
```
ENTITY nome_componente IS
  Dichiarazione delle porte di input/output
      Dichiarazione di altri parametri
END nome_componente;
```

Attenzione: il " ; " va inserito al termine di ogni dichiarazione e di ogni istruzione.

Nel dettaglio:

porte di input/output: la dichiarazione è definita dalla parola chiave PORT, con la seguente sintassi.

```
PORT(input1,…, inputN :IN type; out1, ... , outN :OUT type);
```



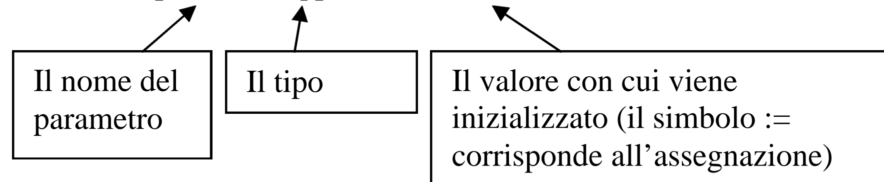
[22] Continuando con il parallelo con il C++ potremmo dire che un'entità corrisponde alla dichiarazione di una classe.

Per quanto riguarda la distinzione tra segnali di ingresso (IN) ed uscita (OUT) notiamo che ad un segnale dichiarato come IN non può venire assegnato un valore dall'intero del componente, mentre un segnale contrassegnato come OUT può essere utilizzato solamente come destinatario dell'assegnazione (ovvero non può essere impiegato nella parte destra di una qualsiasi operazione di assegnazione).

Esistono inoltre due ulteriori specificatori di segnale: INOUT, che identifica un segnale bidirezionale (ovvero che può essere utilizzato sia come ingresso che come uscita) e BUFFER, che indica la presenza di un registro collegato all'output (in sostanza consente di leggere l'output anche all'interno dell'entità, cosa impossibile altrimenti).

Parametri: in questa categoria ricadono tutti quelle "variabili"[23] che sono caratteristiche del dispositivo; essi vengono dichiarati ricorrendo alla parola chiave GENERIC, ad esempio:

```
GENERIC( param : type := inizializzazione);
```



Un esempio di dichiarazione completa è quindi il seguente:

```
ENTITY prova IS
      PORT( in1, clk : IN BIT ; out1 : OUT := '0' );
      GENERIC(ritardo : TIME := 6 ns );
END prova;
```

2.10.2 ARCHITECTURE

La descrizione di come opera il componente, ovvero la sua architettura, viene definita mediante la parola chiave ARCHITECTURE.

```
ARCHITECTURE identificativo OF nome_componente IS
      Dichiarazioni
BEGIN
      Descrizione del comportamento del
  Componente (behavioural, structural,ecc…)
END identificativo;
```

Analizziamo questa definizione:

identificativo: è un nome che si assegna alla particolare descrizione del componente che verrà realizzata (come già detto, per uno stesso componente possono esistere più descrizioni).

[23] In pratica vengono trattate come costanti entro l'entità e l'architettura a cui sono associate; il loro valore può essere mappato dall'esterno, ma non internamente all'architettura.

Nome_componente: è il componente per cui si vuole fornire la descrizione, definito in precedenza mediante la dichiarazione di ENTITY.

Dichiarazioni: possiamo dichiarare in questa sezione altri componenti che possono venire utilizzati nella descrizione della funzionalità del componente dato, oppure possiamo definire altri parametri di interesse (non accessibili dall'esterno), oppure altri segnali utili per esplicitare le connessioni tra diversi moduli che compongono la rappresentazione.

BEGIN – END *identificativo:* racchiudono il corpo principale della specifica. Possono comprendere:

- la specifica di differenti moduli che compongono il progetto (direttive BLOCK e COMPONENT, descritte in seguito)
- il mapping[24] dei segnali che collegano i differenti moduli
- una rapprentazione "sequenziale" (PROCESS), quindi software-like, del comportamento del componente nel suo complesso.

Occorre tenere presente questo fatto: ogni istruzione presente nella sezione BEGIN-END di un'architettura deve NECESSARIAMENTE essere preceduta da una etichetta (label), seguita dai " : ", che la identifichi in modo univoco all'interno dell'architettura (per tutta una serie di motivi che saranno chiari nel seguito).

2.10.3 CONFIGURATION

Come abbiamo già detto, possiamo definire differenti architetture per una stessa entità; quello che ci occorre ora è quindi un modo per poter indicare al compilatore quale architettura vogliamo utilizzare nella simulazione. La sintassi è la seguente:

```
CONFIGURATION configuration_name OF entity_name IS
  FOR architecture_name
    FOR label : component_name
     USE ENTITY library.component_name(architecture_name);
    END FOR
  END FOR
END configuration_name;
```

entity_name: è l'entità per la quale vogliamo specificare le architetture utilizzate dai componenti.

architecture_name: è l'architettura dell'entità indicata sopra per cui vogliamo specificare le architetture utilizzate dai componenti.

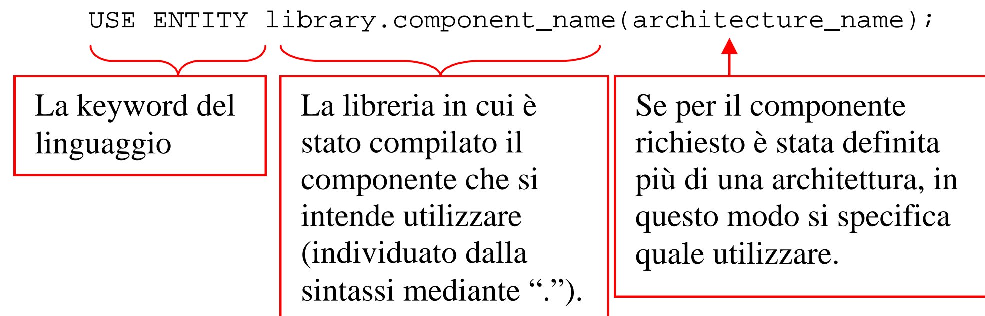
label : component_name: lo specifico componente a cui si fa riferimento nell'architecture body dell'entità.

La riga di codice seguente effettua l'associazione desiderata e specifica

---

[24] Con la definizione mapping di un segnale si intende stabilire una relazione (mediante opportuna sintassi) tra gli ingressi e le uscite di ogni singolo modulo ed i segnali che fisicamente collegano i vari componenti. Se ad esempio si hanno due circuiti integrati l'uno collegato all'altro da un opportuno conduttore, si può assegnare un identificativo al conduttore ed esprimere il collegamento fisico tra i due circuiti mappando l'uscita dell'uno e l'ingresso dell'altro sullo stesso conduttore.

quale architettura di componente utilizzare per l'elemento in questione:



Esistono inoltre dei modi più compatti per dichiarare il binding tra entità ed architetture; ciò può essere fatto utilizzando la USE ENTITY direttamente nella porzione di codice che si riferisce alla parte dichiarativa nella definizione di entità, ma in questo modo si renderebbe il codice meno leggibile.

2.11. Tipi di dato e funzioni definiti dall'utente

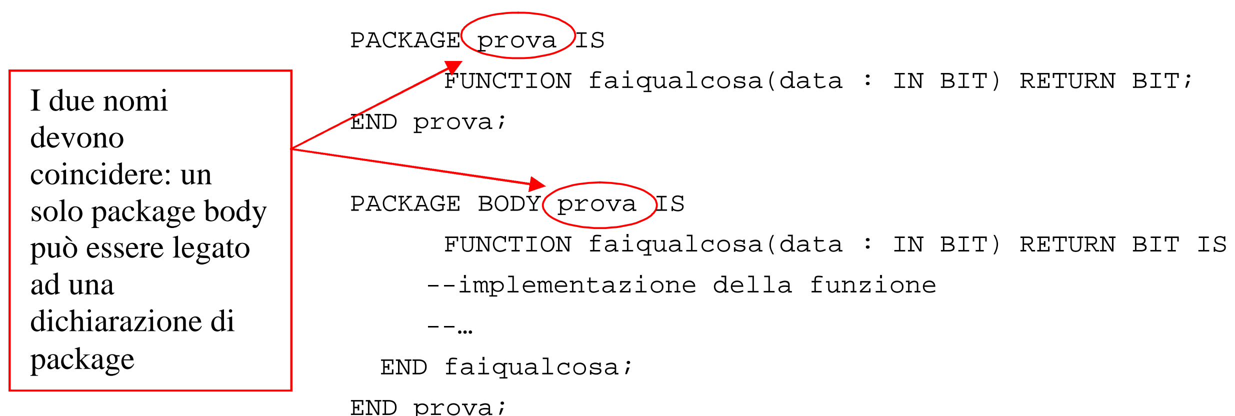
In VHDL spesso si richiede l'esigenza di definire dei propri tipi di dato semplicemente perché quelli messi a disposizione dallo standard non si rivelano sufficienti alle necessità del momento. Un esempio potrebbe essere la necessità di definire una rappresentazione compatta per un bus di 9 bit.

Risulta inoltre molto comoda la possibilità di poter raggruppare una serie di definizioni (tipi di dato, costanti, funzioni, ecc…) all'interno di un contenitore, in modo da poter organizzare anche logicamente le varie definizioni e funzioni ad esse collegate.

2.11.1 PACKAGE & PACKAGE BODY

La direttiva PACKAGE opera come ENTITY, nel senso che dichiara l'interfaccia per le funzioni e le procedure che si desidera includere nel package. La definizione delle suddette funzioni e procedure viene realizzata nel PACKAGE BODY, che opera analogamente alla implementazione dell'architettura di una entità. La differenza fondamentale sta nel fatto che un solo PACKAGE BODY può essere associato al corrispondete PACKAGE.

Come esempio si consideri il codice seguente:



Facciamo inoltre notare che il VHDL consente (entro un package) l'overloading delle funzioni, ovvero possiamo definire due o più funzioni dotate dello stesso nome, ma con liste di argomenti differenti. Il compilatore al momento della chiamata deciderà quale corpo di funzione utilizzare in base agli argomenti che gli vengono passati; rifacendosi all'esempio precedente possiamo scrivere:

Funzioni in overloading

```
PACKAGE prova IS
    FUNCTION faiqualcosa(data : IN BIT) RETURN BIT;
    FUNCTION faiqualcosa(data : IN BIT_VECTOR) RETURN
     BIT;
END prova;

PACKAGE BODY prova IS
     FUNCTION faiqualcosa(data : IN BIT) RETURN BIT IS
  BEGIN
        --implementazione della funzione
  END faiqualcosa;
  FUNCTION faiqualcosa(data : IN BIT_VECTOR) RETURN
            BIT IS
  BEGIN
     --implementazione della funzione
  END faiqualcosa;
END prova;
```

2.12. Block & Component

Una direttiva BLOCK indica un modulo dichiarato e definito all'interno dell'architettura corrente, di cui in sostanza si fornisce sia la parte dichiarativa (quindi la descrizione e la mappatura delle porte) che la descrizione comportamentale: in sostanza è un po' come nidificare una architettura dentro l'altra.

Il componente si definisce quindi:

Queste dichiarazioni hanno significato analogo a quanto visto per le entità.

```
block_label : BLOCK [ ( guarding_expression ) ]
  Dichiarazione e mapping delle porte di input/output
  Dichiarazione di altri parametri
BEGIN
  Descrizione comportamentale del componente
END BLOCK block_label
```

L'utilizzo dell'etichetta all'inizio della dichiarazione è essenziale perché il blocco viene definito in fase di implementazione dell'architettura.

Il mapping delle porte di ingresso e uscita ai segnali ed alle porte di I/O dell'entità viene eseguito ricorrendo ad istruzioni differenti nel caso si stiano

considerando parametri o segnali del componente; le due istruzioni in questione sono le seguenti:

- `PORT MAP ( block_signal => architecture_signal , ecc… );`
- `GENERIC MAP ( block_param => new_value , ecc… );`

Nel primo caso si mappano i segnali, nel secondo si sovrascrivono i valori di dafault specificati in fase di dichiarazione del componente, anche se quest'ultima opportunità risulta molto più utile quando si lavora con i componenti.

Da notare inoltre la presenza dell'opzionale espressione di guardia (*guarding_expression*) che implica un uso completamente differente della keyword BLOCK: possiamo inserire in questa sezione una espressione (che deve essere valutata come booleana) e che deve basarsi su segnali ed attributi definiti nell'entità di cui l'architettura specifica l'implementazione.

Una ulteriore parola chiave (GUARDED) potrà quindi essere utilizzata all'interno di BEGIN-END BLOCK per permettere l'esecuzione di singole istruzioni solo quando l'espressione di guardia viene valutata come vera.

La direttiva COMPONENT viene invece utilizzata per riferirsi ad entità descritte separatamente e situate nelle librerie di progetto[25]. Per dichiarare un componente si utilizza la forma:

```
COMPONENT identificativo_componente
  Dichiarazione delle porte di input/output
  Dichiarazione di altri parametri
END COMPONENT;
```

Queste dichiarazioni hanno significato analogo a quanto visto per le entità.

Essendo una dichiarazione, questa va riportata nella parte dichiarativa dell'implementazione dell'architettura (prima del BEGIN); l'utilizzo del componente avviene mediante la sintassi seguente:

```
label : identificativo_componente
  [GENERIC MAP];
  [PORT MAP];
```

Il mapping avviene con le modalità sopra elencate.

2.13. Subprograms

Con il termine subprograms si intendono funzioni (functions) e procedure (procedures) che possono essere richiamate da un modulo VHDL. Sostanzialmente si tratta di blocchi di codice eseguito in maniera sequenziale (come i PROCESS descritti nella sezione seguente). Come i blocchi descritti sopra, i subprograms possono dichiarare variabili, costanti ed altri subprograms locali. Occorre porre attenzione al fatto che, sebbene i dati vengano passati mediante liste di parametri, è comunque possibile accedere, sotto determinate condizioni, anche a segnali dichiarati esternamente e non facenti parte della

[25] Sarà inoltre importante specificare in seguito da quale libreria si "preleva" l'entità che si intende utilizzare

lista dei parametri; questo comporta che all'interno di una funzione o procedura possiamo modificare dei segnali facenti parte di altri moduli VHDL: si parla in questo caso di *side-effect*.

2.13.1 Function

Una funzione è una porzione di codice che a partire da dei valori, forniti mediante passaggio di argomenti, restituire un risultato del tipo specificato. Le funzioni non ammettono side-effect, e non è quindi possibile intervenire per modificare il valore dei segnali o delle variabili che costituiscono gli argomenti di una funzione.

La dichiarazione di funzione è la seguente:

Gli argomenti vanno espressi nella forma: NAME : TYPE. Lo specificatore IN in questo caso non è necessario in quanto gli argomenti non possono essere modificati all'interno della funzione (in sostanza OUT non è ammesso).

```
FUNCTION name ( argument_list ) RETURN return_type;
```

A questa deve seguire un "function body" che specifica il comportamento della funzione:

```
FUNCTION name ( argument_list ) RETURN return_type IS
  [Variable_declaration]
BEGIN
  Function body
  RETURN value_returned
END name;
```

Le funzioni possono venire inserite nei packages per raggrupparle secondo la logica delle operazioni svolte.

2.13.2 Procedures

Una procedura agisce come un processo, ma può essere integrato in un package per essere richiamata da qualsiasi modulo ne abbia bisogno (senza la necessità di duplicare codice, come avverrebbe nel caso si volessero utilizzare dei processi locali). Analogamente alle funzioni, una procedure è dotata di una lista di argomenti, che questa volta possono essere sia di ingresso che di uscita (si devono indicare gli specificatori IN/OUT sui segnali coinvolti), ma a differenza delle funzioni non consente di ritornare direttamente un valore. Le procedure consentono il side-effect.

La dichiarazione è la seguente:

```
PROCEDURE name ( argument_list );
```

La dichiarazione di argument_list è analoga a quella delle porte di ingresso-uscita di una entity, vedi il paragrafo 2.10.1.

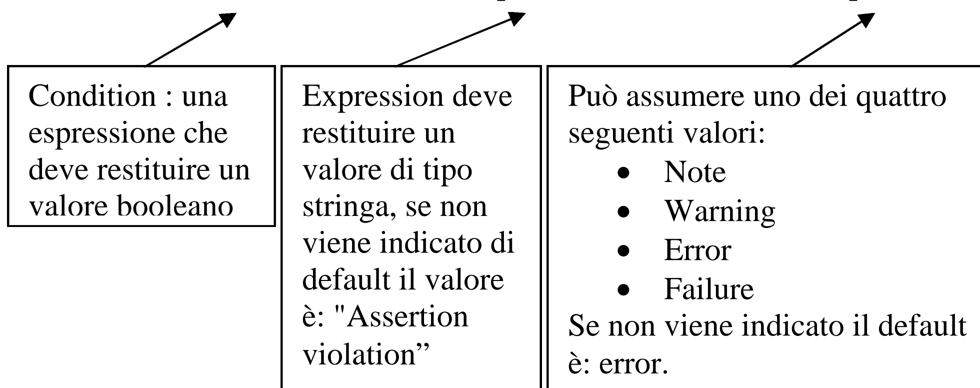
Alla dichiarazione deve seguire il "procedure body", nella forma:

```
PROCEDURE name ( argument_list ) IS
BEGIN
     Descrizione del comportamento della procedura
END name;
```

2.14. Tenere sotto controllo l'esecuzione – ASSERT

Molto spesso le simulazioni sono eccessivamente lunghe in termini di tempo impiegato, soprattutto con il crescere della complessità del progetto in esame. È necessario quindi un metodo per riportare dei messaggi di stato all'utente ed avvertirlo in questo modo del punto a cui si è giunti durante la simulazione, oppure riportare un errore nel caso di comportamento anomalo del sistema. Per raggiungere facilmente questo obiettivo, il VHDL mette a disposizione l'istruzione ASSERT.

ASSERT serve a verificare una condizione ed a riportare un messaggio se questa condizione viene violata. La sintassi è la seguente:



L'esecuzione della simulazione potrebbe terminare se il livello di severity è superiore ad una determinata soglia che di solito viene stabilita dall'utente.

2.15. L'importanza del Tempo

Il trasferimento di segnali tra componenti o all'interno dello stesso componente viene effettuato mediante dei collegamenti o dei bus. In VHDL con il termine di segnale ci si riferisce tanto all'informazione vera e propria (ovvero alla forma d'onda), quanto al supporto fisico su cui viaggia. Quindi, a causa dei ritardi associati alla propagazione, l'assegnamento di un valore ad un segnale deve necessariamente coinvolgere anche il fattore tempo.

Con il termine *scheduling* (dal verbo *to schedule*, letteralmente: mettere in lista) si indica il fatto che una componente temporale viene associata ad un valore che sta per essere assegnato ad un dato segnale; il VHDL inserisce quindi il dato in una lista interna di azioni da eseguire in determinati istanti temporali.

Questa associazione tra valori e intervalli temporali, in aggiunta a

costrutti specifici del linguaggio, permette la creazione di un ambiente in grado di simulare le elaborazioni "concorrenti" dei segnali[26] nei blocchi del circuito in esame.

2.16. Operazioni di Assegnazione

Distinguiamo le assegnazioni in due differenti tipologie, a seconda del tipo di oggetto coinvolto:

- Assegnazione di variabili e costanti.
- Assegnazione di segnali.

2.17. Assegnazione di Variabili e Costanti

Le variabili possono venire dichiarate e possono ricevere un valore solamente entro le porzioni "sequenziali" di codice[27]; il simbolo di assegnazione utilizzato è il seguente:

: =

Le costanti possono invece essere dichiarate e definite anche all'interno di blocchi ad elaborazione concorrente.

> Le operazioni di assegnamento a variabile e costante non coinvolgono in nessun modo la variabile tempo.

2.17.1 Assegnazione di Segnali

L'assegnamento di un valore ad un segnale coinvolge il tempo: si dice quindi che il valore viene inserito in un *signal driver* (possiamo considerarlo come una struttura interna al linguaggio) e compare sul segnale dopo un intervallo di tempo che viene specificato.

> Assegnamenti multipli in ogni blocco concorrenziale sono considerati attivi simultaneamente e l'ordine in cui appaiono è assolutamente ininfluente; quando si assegna un valore ad una serie di segnali, pensare che gli assegnamenti avvengano in maniera sequenziale è un ERRORE assai grave. Un approccio mentale di tipo procedurale è errato!

> E' inoltre possibile che ad uno stesso segnale vengano assegnati valori provenienti da fonti differenti; in questo caso viene creato un signal driver per ogni fonte ed il progettista deve prevedere una adeguata **funzione di risoluzione** che indichi come gestire l'eventuale confluenza di dati. Un segnale dotato di driver multipli è detto **resolved signal**.

[26] In realtà le simulazioni che vengono eseguite "localmente" sono sequenziali (per la natura stessa degli elaboratori su cui vengono eseguite) mentre, mediante particolari accorgimenti, i risultati complessivi vengono presentati al progettista come se l'elaborazione fosse il risultato dell'azione simultanea di tutti i blocchi coinvolti.

[27] Vedremo i blocchi sequenziali in seguito, per ora basti sapere che sono porzioni di codice che vengono eseguite in sequenza, disattivando quindi la modalità di simulazione concorrente.

assegnamento

Il simbolo impiegato per l'assegnamento di segnali è il seguente:

```
<=
```

Lo scheduling (l'associazione con un intervallo temporale) si ottiene specificando la parola chiave AFTER seguita dalla quantità di tempo necessaria. Quest'ultima può comunque essere omessa, in questo caso si assume che l'assegnazione avvenga con un ritardo fisico NULLO[28].

Un esempio è il seguente. Si supponga signal1 di tipo BIT:

```
signal1 <= '1' AFTER 5 ns;
```

signal1 viene portato al valore di un bit dopo 5 ns.

Funzione di risoluzione

Una funzione di risoluzione accetta in ingresso un array di valori e restituisce in uscita un valore del tipo desiderato; il VHDL non fornisce nessuna funzione standard, tutte le funzioni necessarie devono venire implementate dal progettista.

Di seguito viene presentata una funzione di risoluzione basata sull'OR dei segnali: possiamo visualizzare questa funzione come una unica porta OR che effettua la suddetta operazione su tutti i segnali dell'array di ingresso e che, nel caso in esame, restituisce in uscita il singolo bit risultante dall'operazione logica.

```
function oring(driver : in bit_vector) return BIT is
  --la funzione di risoluzione è utilizzata quando
     --più dati debbono confluire nello stesso bus
  variable result : BIT := '0';
begin
  for Index in driver'range loop
        result := result or driver(Index);
  end loop;
  return result;
end oring;
```

Ovviamente nel caso di più segnali dotati di funzione di risoluzione che confluiscono in uno stesso BUS, il contenuto del BUS sarà il risultato dato dall'applicazione della funzione di risoluzione a tutti i segnali interessati.

Un assegnamento di segnale modella il passaggio fisico di una forma d'onda entro un conduttore, quindi non può tenere in considerazione solamente il ritardo di propagazione: deve essere considerata anche la risposta in frequenza del conduttore.

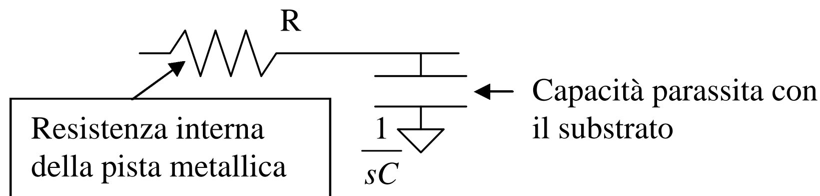
L'operazione di assegnazione viene quindi specificata da due ulteriori parole chiave: INERTIAL e TRANSPORT.

---

[28] Attenzione, come vedremo tra poco parlando delle transazioni, questa non implica affatto che anche lo scheduling avvenga con ritardo nullo.

2.17.2 INERTIAL delay – Risposta in frequenza limitata

Un conduttore metallico, ricordando un po' di elettromagnetismo, può essere schematizzato nel seguente modo:



La funzione di trasferimento di questo semplice circuito è $\frac{1}{sRC+1}$, si tratta quindi di un passa-basso (ovvero il circuito è limitato in frequenza).

Poiché $\omega = 2\pi f = \frac{2\pi}{T}$ ne consegue che segnali che restano alti[29] per un periodo di tempo inferiore a quello che caratterizza la risposta in frequenza del conduttore verranno "tagliati" dal circuito, ovvero non verranno trasmessi all'uscita.

Per modellare questo comportamento e si fa ricorso alla coppia di keywords REJECT – INERTIAL. Il funzionamento è il seguente:

REJECT specifica l'intervallo di tempo minimo in cui il segnale deve assumere un valore significativo, altrimenti il valore viene rigettato e lo stato del segnale rimane inalterato.

INERTIAL specifica che stiamo considerando una trasmissione di segnale lungo una linea con perdite.

Un esempio:

```
signal1 <= REJECT 3 ns waveform AFTER 2 ns;
```

La precedente assegnazione significa che un impulso (appartenente alla forma d'onda definita dal segnale `waveform`) di lunghezza inferiore ai 3 ns viene rigettato, mentre tutti i segnali che soddisfano il vincolo imposto, compaiono su signal1 con un ritardo di 2 ns.

Se non viene precisata nessuna parola chiave per il tipo di assegnamento, si assume che esso sia di tipo INERTIAL; questo è il tipo di assegnazione di default. Ad esempio:

```
signal1 <= waveform AFTER 5 ns;
```

Se un impulso di durata inferiore ai 5 ns compare sulla forma d'onda, esso viene rigettato, altrimenti viene piazzato sul driver del segnale e compare in uscita dopo 5 ns.

---

[29] Parlando di circuiti digitali è ovvio che in ogni caso ci si riferisce a segnali di tipo binario.

2.17.3 TRANSPORT delay – Risposta in frequenza illimitata

Se la linea ha risposta virtualmente illimitata in frequenza, un assegnamento con un ritardo di tipo TRANSPORT fa sì che sul driver del segnale venga riportata esattamente la sequenza di impulsi che compare sulla parte destra dell'espressione, opportunamente ritardata del tempo indicato:

```
signal1 <= TRANSPORT waveform AFTER 7 ns;
```

In questo caso waveform viene replicata su signal1 ritardata di 7 ns.

2.17.4 Particolarità degli assegnamenti

Definiamo *waveform element* una espressione dotata di un opportuno meccanismo di ritardo, seguita dalla clausola AFTER.

Quando sul driver di un segnale vengono inseriti degli assegnamenti multipli[30] di differenti waveform element, il meccanismo di ritardo specificato viene applicato solamente al primo di essi, tutti gli altri elementi vengono "convertiti" in TRANSPORT.

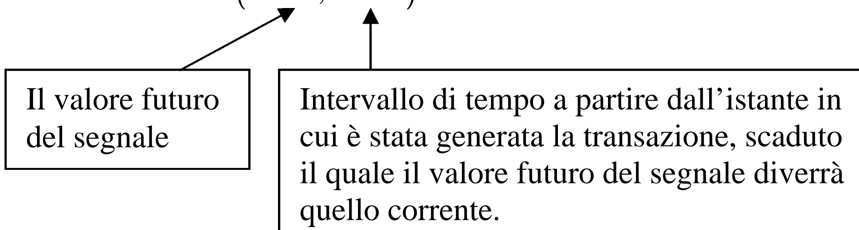
2.18. Eventi e transazioni

Quando una forma d'onda causa una variazione nel valore di un segnale, si dice che su quel segnale si è verificato un *evento*; quando si verifica lo scheduling di un valore (il valore viene messo in una lista di attesa per essere assegnato ad un segnale) si dice che una *transazione* è stata inserita sul driver del segnale di destinazione.

Un **evento** è quindi un assegnamento ad un segnale (anche se il valore assunto è lo stesso che il segnale conteneva in precedenza). Il verificarsi di un evento su un qualsiasi elemento che si trovi nella parte destra di una espressione di assegnazione implica una nuova completa valutazione dell'espressione (e, se necessario, nuove transazioni che vengono inserite sul driver dei segnali coinvolti).

Una **transazione** è una coppia valore-tempo in cui il valore è lo stato corrente del segnale se il tempo è 0, altrimenti il valore è lo stato futuro del segnale dopo il periodo di tempo indicato; una transazione implica il cambiamento di stato di un segnale.

Transazione: $tr = (value, \Delta time)$



[30] Gli assegnamenti vengono concatenati per mezzo dell'operatore "," (virgola).

Occorre prestare particolare attenzione a non inserire delle nuove transazioni sul driver di un segnale in cui ci siano già transazioni pendenti. Se questo accade occorre prendere una decisione e stabilire se una sola od entrambe debbano rimanere sul driver del segnale; questa situazione è oggetto del prossimo paragrafo.

2.18.1 Transazioni multiple sullo stesso driver

Transazioni multiple possono presentarsi sia in porzioni concorrenziali che sequenziali del codice; il comportamento del VHDL in questo caso dipende da tutta una serie di fattori e dai parametri che caratterizzano la transazione (ovvero dal meccanismo di trasporto, dal valore dei segnali, dai ritardi di assegnazione, ecc…).

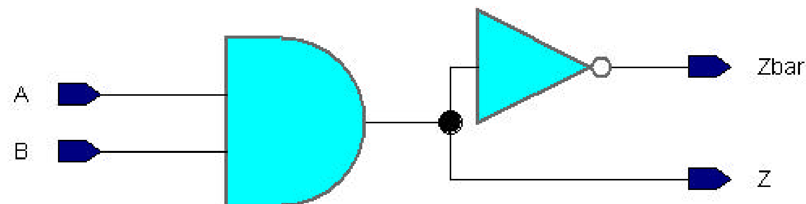
Il tutto viene riassunto nella tabella seguente.

<table>
<tr><td rowspan="2"></td><td colspan="2">Meccanismo di trasporto</td></tr>
<tr><td>TRANSPORT</td><td>INERTIAL</td></tr>
<tr><td>La nuova transazione va inserita
PRIMA
delle transazioni precedenti</td><td>Le transazioni PRECEDENTI vengono SOVRASCRITTE</td><td>Le transazioni PRECEDENTI vengono SOVRASCRITTE</td></tr>
<tr><td rowspan="6">La nuove transazione va inserita
DOPO
le transazioni precedenti</td><td rowspan="6">La nuova transazione viene ACCODATA alle transazioni precedenti</td><td>$Value_{new}=Value_{old}$</td></tr>
<tr><td>La nuova transazione viene ACCODATA alle transazioni precedenti</td></tr>
<tr><td>$Value_{new}/=Value_{old}$ e $(time_{new}-time_{old}) > reject$ time</td></tr>
<tr><td>La nuova transazione viene ACCODATA alle transazioni precedenti</td></tr>
<tr><td>$Value_{new}/=Value_{old}$ e $(time_{new}-time_{old}) < reject$ time</td></tr>
<tr><td>Le transazioni PRECEDENTI vengono SOVRASCRITTE</td></tr>
</table>

2.19. Delta Delay (δ-delay)

Oltre ai normali ritardi, esplicitamente indicati nelle espressioni di assegnazione, il VHDL introduce un ulteriore elemento di ritardo, detto δ-delay, utilizzato internamente per eseguire la simulazione di operazioni concorrenti. L'utente finale non può in alcun modo intervenire su questo ritardo, dato che viene introdotto internamente solamente al fine di eseguire la simulazione.

Consideriamo il seguente esempio, che schematizza il circuito combinatorio qui sotto riportato:




```
ENTITY prova IS
      PORT(a,b : IN BIT ; z,zbar : BUFFER BIT);
END ENTITY;

ARCHITECTURE delta OF prova IS
BEGIN
      zbar <= NOT z;
  z <= a AND b AFTER 7 NS;
END delta;
```

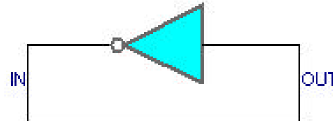
Dalla simulazione di questo blocco concorrente ci aspettiamo che il risultato di (a AND b) e la negazione di z compaiano contemporaneamente (dopo 7 ns) rispettivamente su z e zbar. Ma per avere un risultato corretto di NOT z occorre aspettare che z abbia assunto il nuovo valore. Ecco allora come opera il δ-delay: all'inizio della simulazione entrambe le espressioni vengono valutate e sul driver di z viene posta la transazione (a AND b, 7ns); ma l'evento su z provoca una nuova valutazione dell'espressione in zbar, di conseguenza il VHDL inserisce su zbar una transazione (NOT z, 7 ns + δ), in modo da riportare correttamente anche il valore negato. Il linguaggio dunque aspetta implicitamente che su z ci sia il nuovo valore per poi rivalutare tutte le espressioni che dipendono dall'evento verificatosi.

Il δ-delay viene solamente utilizzato per i cicli di simulazione interna e non contribuisce al ritardo "reale" con cui avvengono gli assegnamenti, quindi al termine della simulazione al progettista sembra che z e zbar abbiano assunto contemporaneamente i nuovi valori.

2.19.1 Pericoli del δ-delay

Se il circuito non viene adeguatamente caratterizzato, il δ-delay può portare a risultati assurdi ed imprevedibili; consideriamo il seguente esempio, perfettamente coerente dal punto di vista del linguaggio, ma concettualmente errato da un punto di vista progettuale.

Schematizziamo un buffer con un NOT in retroazione:



che possiamo descrivere come segue (nota: assegniamo dei valori iniziali ai segnali):

```
ENTITY prova2 IS
END prova2;

ARCHITECTURE oscillazione OF prova2 IS
  SIGNAL in : BIT := '0';
  SIGNAL out : BIT := '1';
BEGIN
  out <= in; --niente ritardo
  in <= NOT out; --niente ritardo
END oscillazione;
```

Analizziamo le transazioni che vengono generate: all'istante iniziale vengono generate le transazioni:

- sul driver di out ('1',0)
- sul driver di in ('0',0)

Ma entrambe generano un evento sull'altro segnale (per via della ricorsione), che provoca quindi le transazioni:

- sul driver di out ('0',0+δ)
- sul driver di in ('1',0+δ)

Che generano nuovi eventi, e quindi nuove transazioni facendo riprendere il ciclo dall'inizio… in sostanza la simulazione non si muove mai dall'istante 0 ed oscilla all'infinito (od almeno fino all'esaurimento dello stack di chiamate che può generare il simulatore).

In questo caso il simulatore genera un errore.

2.20. Concurrent processing

Ora che abbiamo introdotto elementi sufficienti del linguaggio, possiamo ritornare sulla differenza che esiste tra elaborazione concorrente ed elaborazione sequenziale delle differenti unità logiche che compongono un progetto in VHDL. Come detto in precedenza è possibile "mescolare" e far interagire tra loro blocchi concorrenti (le cui istruzioni vengono eseguite tutte in parallelo, a seconda del timing associato alle transazioni) e blocchi sequenziali (le cui istruzioni vengono eseguite in sequenza, ma in parallelo a quelle degli altri blocchi concorrenti).

2.20.1 Descrivere un blocco sequenziale – PROCESS

Un blocco ad elaborazione sequenziale si definisce mediante la keyword PROCESS e può essere dotato o meno di una sensitivity list: una lista di segnali che al cambiamento di stato attivano l'esecuzione del processo; se tale lista è assente il processo viene innescato direttamente all'inizio della simulazione (parallelamente a tutte le altre istruzioni concorrenti).

Un processo si dichiara nel modo seguente:

```
[ process_label : ] process [ ( sensitivity_list ) ]
      Dichiarazioni
begin
      Descrizione comportamentale del processo
      (istruzioni sequenziali)
end process [ process_label ] ;
```

nel dettaglio:

le parti tra parentesi quadre ([…]) sono facoltative.

process_label: il nome dell'istanza del processo entro l'architecture body.

sensitivity_list: una lista di segnali (caratterizzati o meno da specifici attributi) che attivano il processo al cambiamento di stato. Mettiamo in evidenza che il processo viene COMUNQUE ESEGUITO in fase di inizializzazione della simulazione, a prescindere da quanto specificato nella sensitivity list.

Dichiarazioni: possiamo dichiarare in questa sezione altri componenti che possono venire utilizzati nella descrizione della funzionalità del processo, oppure possiamo definire altri parametri di interesse (non accessibili dall'esterno), oppure altri segnali utili internamente, ecc… Tutto quanto definito all'interno di un processo è locale al processo stesso.

Facciamo notare che un blocco sequenziale viene utilizzato esclusivamente in descrizioni di tipo behavioural, in quanto non direttamente implementabile in hardware.

2.20.2 Un esempio di elaborazione concorrente

Consideriamo il seguente codice:

Ecco il segmento di codice sequenziale!

```
ARCHITECTURE arc OF prova;
BEGIN
      1: c <= a AND b AFTER delay;
      2: d <= b AND c AFTER delay;
  PROCESS
  BEGIN
      3: e <= a AND b AFTER delay;
      WAIT FOR 0 ns;
      4: f <= b AND c;
  END PROCESS;
END arch;
```

Le istruzioni contrassegnate con 1 e 2 fanno parte di un blocco ad elaborazione concorrente, ma anche il processo nel suo insieme è un elemento ad elaborazione concorrente. Se ne deduce quindi che le istruzioni 1, 2 e 3 (la prima del blocco) vengono inserite sui driver dei segnali[31] simultaneamente (le transazioni hanno tutte la stessa componente di tempo). L'istruzione WAIT FOR 0 ns (maggiori dettagli nel paragrafo seguente) viene qui utilizzata per passare al δ-delay di elaborazione successivo per l'istruzione 4. All'istante di elaborazione successivo saranno eseguite[32] le istruzioni 1, 2 e 4.

Se non ci fosse stato il WAIT FOR l'intero blocco di istruzioni sarebbe stato eseguito simultaneamente alle altre:

Durante una simulazione un processo viene eseguito fino a che non si incontra una istruzione WAIT o la fine del processo stesso. TUTTE LE ISTRUZIONI VALUTATE VENGONO CONSIDERATE COME UNA SINGOLA ISTRUZIONE CONCORRENTE!

Attenzione: se si utilizza una sensitivity list nella creazione di un processo, non saranno consentite istruzioni WAIT all'interno del corpo del processo stesso.

E' inoltre possibile creare dei processi che ciclino all'infinito; la simulazione resta allora confinata entro il processo ed il tempo non avanzerebbe mai! Il simulatore in questo caso non genera alcun errore, al massimo avvisa l'utente del comportamento anomalo.

Da ultimo si ponga attenzione agli assegnamenti ai segnali:

Se, in un singolo processo, vengono eseguiti più assegnamenti ad uno stesso segnale, solamente l'ultimo di essi verrà preso in considerazione proprio per la caratteristica di sequenzialità con cui vengono valutate le istruzioni.

[31] Sempre che uno dei segnali nella parte destra delle istruzioni abbia cambiato stato causando una nuova valutazione dell'espressione.
[32] Vedi nota precedente.

2.20.3 Processes & WAIT statement

Possiamo interrompere temporaneamente l'esecuzione di un processo mediante una istruzione WAIT; tale istruzione assume quattro differenti forme:

- WAIT;

  In questo caso il processo viene eseguito fino alla clausola WAIT, dopodiché si arresta per sempre.

- WAIT ON sensitivity_list;

  La sensitivity_list specifica una serie di segnali a cui il processo è sensibile: una variazione di segnali causa la ripresa del processo. Il funzionamento è analogo a quello della sensitivity list associata direttamente al processo (in questo caso il compilatore VHDL introduce in maniera trasparente al programmatore un WAIT ON prima di END PROCESS).

- WAIT UNTIL condition;

  Quando l'espressione booleana data dalla condizione viene valutata come vera, il processo riprende, altrimenti rimane in stallo. Il processo in questo caso è sensibile a tutti i segnali menzionati nell'espressione ed una loro variazione causa una nuova valutazione dell'espressione stessa.

- WAIT FOR time_expression;

  In questo caso il processo rimane sospeso per il periodo di tempo indicato.

Sfruttando le istruzioni WAIT si possono generare dei processi dotati di sensitivity list senza dichiararla direttamente in fase di creazione del processo. Questa opportunità risulta particolarmente utile per aggirare alcuni dei limiti dei sintetizzatori di hardware in commercio: un processo dotato di sensitivity list generalmente non viene sintetizzato correttamente, si preferisce allora utilizzare una istruzione WAIT (si consiglia inoltre di inserire un solo WAIT in ogni PROCESS e di posizionarlo all'inizio).

Utilizzando sapientemente le istruzioni WAIT ON / UNTIL è assai facile costruire processi sincroni od asincroni.

> Attenzione: un processo senza sensitivity list e senza WAIT verrà sicuramente indicato come errore dal compilatore, dato che sotto queste condizioni continuerebbe ad essere sospeso per un tempo infinito!