Alma Mater Studiorum
Università degli Studi di Bologna

Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

Tesi di Laurea Specialistica in Sistemi Digitali

Linguaggi e ambienti CAD per la sintesi logica di sistemi digitali

Candidato:
Stefano Suraci

Relatore:
Chiar.mo Prof. Eugenio Faldella

Correlatori:
Prof. Stefano Mattoccia
Prof. Marco Prandini

Anno Accademico 2008/2009 - Sessione I

A cosa serve il successo,
se poi non hai qualcuno
con cui poterlo condividere?

Indice

Capitolo 1

Introduzione

1 Modellazione circuitale

Affinché i prodotti elettronici possano stare al passo con cambiamenti così rapidi, quali quelli imposti dal mercato attuale, è indispensabile che siano progettati in tempi estremamente ridotti. In tale contesto, se da una parte la progettazione analogica è rimasta una professione estremamente specializzata, dall'altra quella digitale s'è sempre più affidata alla progettazione assistita al computer (*CAD*, computer-aided design), detta anche progettazione elettronica automatica (*EDA*, elettronic design automation).

Gli strumenti automatici devono assistere il progettista in due fasi fondamentali: la *sintesi* del circuito, cioè la traduzione di un insieme di specifiche in un'implementazione effettiva e la *simulazione* del circuito, attraverso cui è possibile verificare il corretto funzionamento del sistema a diversi livelli di astrazione.

Gli strumenti automatici di sintesi e simulazione richiedono naturalmente che le idee del progettista vengano opportunamente trasferite agli strumenti stessi. Uno dei modi possibili, noto come *schematic capture* ("cattura dello schema"), consiste nel disegnare uno schema circuitale utilizzando un pacchetto grafico specifico. In alternativa, il circuito può essere descritto in forma testuale, analogamente a quanto si fa con le istruzioni di un programma soft-

ware.

Un sistema digitale può essere rappresentato in questo modo usando un *HDL* (Hardware Description Language, "linguaggio di descrizione circuitale"). Negli ultimi 30 anni, sono stati proposti diversi HDL, ma i due più affermati sono il *Verilog* ed il *VHDL* (Very high speed integrated circuits HDL, "HDL per circuiti integrati a velocità molto elevata").

2 Cosè il VHDL?

Il VHDL è un acronimo per *VHSIC Hardware Description Language* (dove *VHSIC* è esso stesso un acronimo per Very High Speed Integrated Circuits). Si tratta di un linguaggio di descrizione hardware che può essere utilizzato per modellare un sistema digitale a vari livelli di astrazione, passando da un livello più alto (attraverso la descrizione algoritmica del comportamento) ad uno più basso (modellazione del circuito con gate elementari).

Questa prima definizione sottolinea un aspetto molto importante: il VHDL non è un linguaggio eseguibile ovvero non descrive quali operazioni un esecutore deve svolgere per ricavare il risultato di una elaborazione, bensì descrive gli elementi che costituiscono il circuito digitale in grado di effettuare l'elaborazione richiesta. Una specifica VHDL non è quindi eseguibile e deve essere pensata come qualche cosa di completamente diverso da un programma o un algoritmo. Tuttavia, una specifica VHDL può essere simulata mediante opportuni strumenti.

La complessità del sistema digitale modellato può variare dal semplice gate fino ad un sistema elettronico complesso costituito da molteplici dispositivi. Il sistema digitale, grazie al VHDL, può essere descritto gerarchicamente ed è possibile modellare anche il timing nella stessa descrizione.

Il VHDL racchiude in se le caratteristiche proprie dei linguaggi sequenziali e concorrenti, ma anche di quelli che producono *net-list* (ad esempio i lin-

guaggi CAD) fornendo la possibilità di definire specifiche temporali (*timing specifications*) e di generare forme d'onda (*waveform*). Per questo, il linguaggio ha costrutti che consentono di esprimere il comportamento sequenziale o concorrente di un componente, considerando o meno la risposta dello stesso alle varie temporizzazioni, e permette di realizzare interconnessioni di componenti. Tutte queste caratteristiche possono essere combinate per fornire una descrizione complessiva del sistema in un singolo modello.

Il linguaggio è fortemente tipato ed eredita molte delle sue caratteristiche, specialmente nella parte sequenziale, dal linguaggio *Ada*. Poichè il VHDL fornisce un range di possibilità di modellazione molto esteso è solitamente difficile da comprendere, fortunatamente è possibile assimilare rapidamente un set di costrutti chiave che permettano di modellare facilmente la maggior parte delle applicazioni, senza dover imparare subito i costrutti più complessi e avanzati.

3 Cenni storici

I requisiti del linguaggio sono stati redatti nel 1981 nell'ambito del programma VHSIC. In questo programma, una discreta quantità di industrie elettroniche americane sono state coinvolte nella progettazione di circuiti VHSIC per il *dipartimento della difesa* (DoD).
A quel tempo, la maggior parte delle società utilizzavano linguaggi proprietari di descrizione dell'hardware (che non necessariamente erano compatibili), con la conseguenza che venditori differenti non potevano scambiare tra loro i progetti in maniera efficiente ed erano costretti a rapportarsi con il dipartimento della difesa ognuno attraverso la propria descrizione. Ciò portava ad inefficienze e rendeva molto difficile riutilizzare componenti valide ma di precedenti fornitori.
Per questi motivi, si è deciso di sviluppare un linguaggio di descrizione dell'hardware che fosse standard per permettere una facile progettazione, documentazione e verifica di sistemi digitali prodotti da società diverse.

Un gruppo di tre società, IBM, Texas Instruments e Intermetrics, ha stretto il primo contratto per la nascita di una prima versione del linguaggio nel 1983. La versione 7.2 del VHDL è stata sviluppata e rilasciata nel 1985. C'è stata una forte partecipazione delle industrie elettroniche nel processo di sviluppo del linguaggio VHDL, specialmente dalle società che sviluppavano componenti VHSIC.

Successivamente al rilascio della versione 7.2 c'è stata una necessità sempre maggiore di rendere il linguaggio uno standard industriale, conseguentemente il linguaggio è stato proposto all'IEEE per la standardizzazione nel 1986, che lo ha reso effettivamente standard nel 1987 (versione nota come *IEEE Std 1076-1987*). La descrizione ufficiale del linguaggio appare in *"IEEE Standard VHDL Language Reference Manual"*, ma il linguaggio è stato anche riconosciuto come standard dall'*ANSI* (American National Standard Institute).

In accordo alle regole IEEE, uno standard deve essere votato ogni cinque anni per poter ancora essere considerato tale: conseguentemente, il linguaggio è stato aggiornato con nuove caratteristiche, la sintassi di molti costrutti è stata resa più uniforme e molte ambiguità presenti nella prima versione sono state superate. La nuova versione del linguaggio è nota come *IEEE Std 1076-1993* e i moderni ambienti IDE utilizzano questa versione del linguaggio.

Il dipartimento della difesa, a partire da Settembre 1988, richiede a tutti i fornitori di circuiti ASIC di fornire la descrizione in VHDL dei dispositivi forniti e delle componenti che li costituiscono, sia a livello comportamentale sia a livello strutturale. Anche i *testbench* che sono stati realizzati per verificare il funzionamento di detti circuiti devono essere forniti in VHDL.

A partire dal 1987, si è reso sempre più necessario realizzare un pacchetto standard per agevolare l'interoperabilità, questo perché molti venditori di software *CAE* (computer-aided engineering) supportavano packages differenti sui loro sistemi. Erano infatti presenti valori logici con rappresentazioni che passavano da 46 possibili stati logici, a 7, a 4 e così via, tutto ciò rendeva estremamente poco portatile i progetti fatti con il linguaggio VHDL (benché

lo stesso fosse standard). Gli sforzi di uniformare i packages portarono alla nascita di una logica a 9 valori: questo package, chiamato *STD_LOGIC_1164*, è stato votato e approvato per divenire uno standard IEEE nel 1993 (*IEEE Std 1164-1993*).

4 Elementi caratteristici del linguaggio

In questa sezione, si riportano le caratteristiche principali del linguaggio VHDL che lo differenziano dagli altri linguaggi di descrizione hardware, tali caratteristiche saranno successivamente riprese e commentate dettagliatamente nel proseguo della tesi.

- Il linguaggio può essere utilizzato come mezzo di comunicazione tra differenti strumenti *CAD* e *CAE*: per esempio un programma di disegno assistito può essere utilizzato per generare una descrizione in VHDL del circuito, quindi tale descrizione può essere fornito ad un simulatore (anche di un'altra casa costruttrice) che lo eseguirà.
- Il linguaggio consente di modellare una gerarchia di componenti: un sistema digitale può essere modellato come un set di componenti interconnessi; ciascun componente può essere modellato come un insieme di sotto-componenti interconnessi.
- Il linguaggio non è "technology-specific", ma è capace di supportare elementi specifici: un utente può definire nuovi tipi logici e nuove componenti. Essendo indipendente dalla tecnologia, lo stesso modello può essere sintetizzato in librerie implementate da società differenti.
- Supporta la modellazione di macchine sincrone e asincrone, possono essere utilizzate inoltre diverse tecniche di modellazione quali descrizione di macchine a stati finiti ed equazioni Booleane.
- Il linguaggio è pubblico, comprensibile sia dalle macchine sai dall'uomo e non è proprietario.
- Il linguaggio ha elementi sintattici che rendono semplice la progettazione di circuiti su larga scala: ad esempio componenti, funzioni, procedure

e packages. Inoltre non ci sono limitazioni imposte dal linguaggio sulla dimensione del progetto.

- I ritardi nominali di propagazione dei componenti, ma più in generale, i ritardi, i tempi di setup e di hold, i vincoli di temporizzazione e il rilevamento dei glitch possono essere descritti molto facilmente in questo linguaggio.
- Un modello può non solo descrivere le funzionalità di un circuito, ma può anche contenere informazioni sul design stesso: ad esempio attributi definiti dall'utente, quali l'area totale occupata o la velocità.
- I modelli comportamentali che aderiscono a precisi vincoli di sintesi possono essere sintetizzati automaticamente al livello di gate.
- La possibilità di definire nuovi tipi di dato fornisce il potere di descrivere e simulare una nuova tecnica di progettazione ad alto livello di astrazione senza preoccuparsi dei dettagli implementativi.

5 Dalla modellazione alla sintesi

Il VHDL è un linguaggio che permette al progettista di sfruttare il calcolatore ed i tool di sviluppo, lungo tutta la filiera produttiva di un circuito digitale.

Partendo da un insieme di specifiche, è infatti possibile suddividere il sistema in moduli, secondo il principio del *divide et impera*, e modellare ciascuno di essi nella forma più opportuna. Il flusso di progettazione classico è riportato in fig. 1.1.
L'estrazione del comportamento della rete che si vuol progettare, così come la descrizione *RTL*, dalle specifiche è affidata all'esperienza del progettista. Mentre il VHDL agisce nei tre livelli evidenziati in arancione, ossia prendendo in ingresso il risultato della prima fase.

Il linguaggio permette, una volta codificato opportunamente il comportamento del circuito o il suo design RTL, di simularne il comportamento in tutte le condizioni (mediante l'uso di *testbench*), garantendo così la correttezza di quanto elaborato. Per far ciò, come si vedrà nel seguito di questa tesi, sono

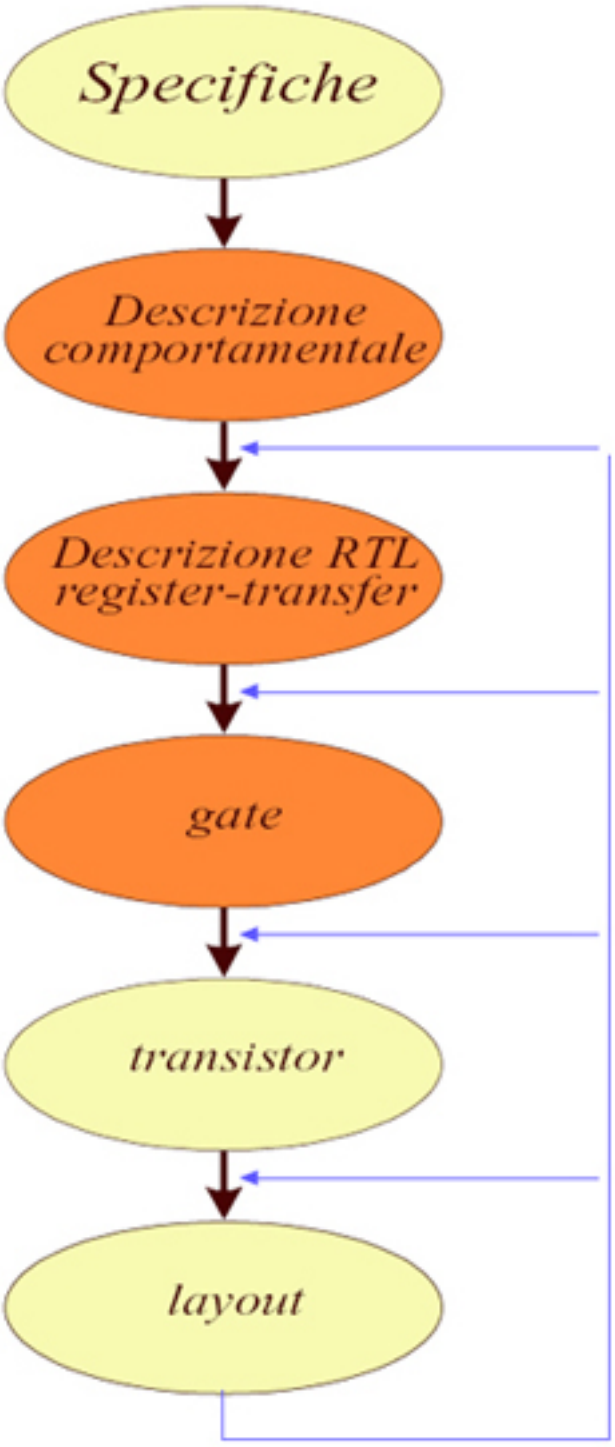


Figura 1.1: Fasi costruttive di un circuito digitale

disponibili molti ausili alla modellazione e al testing, sia propri del linguaggio, sia forniti dall'IDE di programmazione.
Il testing ad ogni fase è importante perché evita la propagazione in cascata degli errori e ne rende più semplice la rilevazione e la correzione (fig. 1.2).

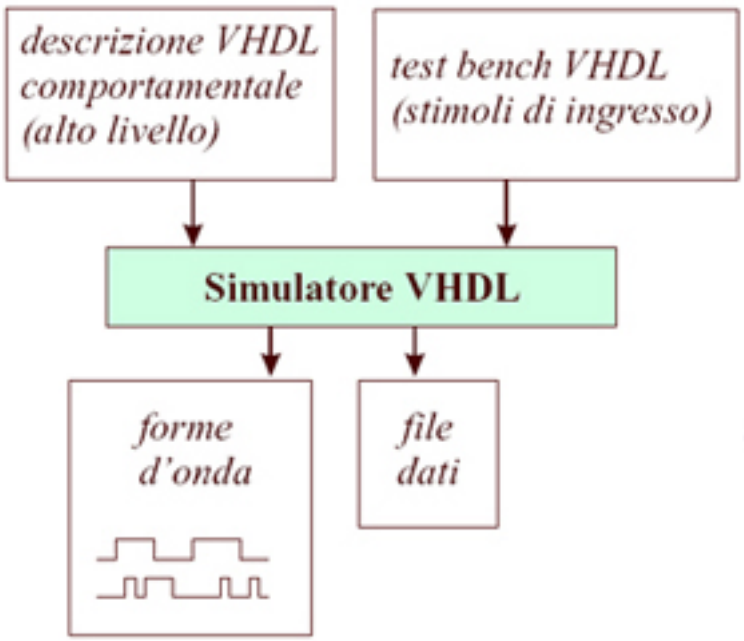


Figura 1.2: Verifica della simulazione

La fase successiva è la *sintesi*: la traduzione in logica combinatoria elementare, ossia l'arrivo al livello gate. Questa attività potrebbe essere fatta dall'operatore, ma - soprattutto per circuiti complessi - sarebbe troppo complessa e prona agli errori. Il vantaggio nell'utilizzo del VHDL è la sintetizzabilità. In altre parole, collegando un *sintetizzatore* - anche di un'altra azienda - a valle della fase di modellazione, è possibile far eseguire automaticamente la conversione al livello gate. L'algoritmo di sintesi è molto sofisticato e permette di introdurre vincoli (ad esempio sull'*ampiezza* del circuito e sullo *sbroglio* dello stesso, inoltre garantisce l'ottimizzazione dell'elaborato (fig. 1.3).

L'elaborato della sintesi prende il nome di *net-list*, questa - attraverso un programma di *place e routing* - è in grado di generare il codice macchina da inserire nella FPGA (ossia in un'unità programmabile) o di istruire i robot

Figura 1.3: Processo di sintesi

a realizzare i circuiti fisici sulla scheda.

L'aspetto interessante è la possibilità di testare il codice ottenuto dal processo di sintesi, nello stesso ambiente di sviluppo, senza dover riscrivere i testbench in un altro linguaggio. Essendo un flusso unico, supportato dal linguaggio, i test effettuati dopo la modellazione sono ripetibili anche dopo la sintesi, con un risparmio netto di tempo e di costi (fig. 1.4).

Riassumendo, data una stessa descrizione VHDL, si succedono due processi molto differenti tra loro:

- *Simulazione*: verifica comportamento Ingresso - Uscita;

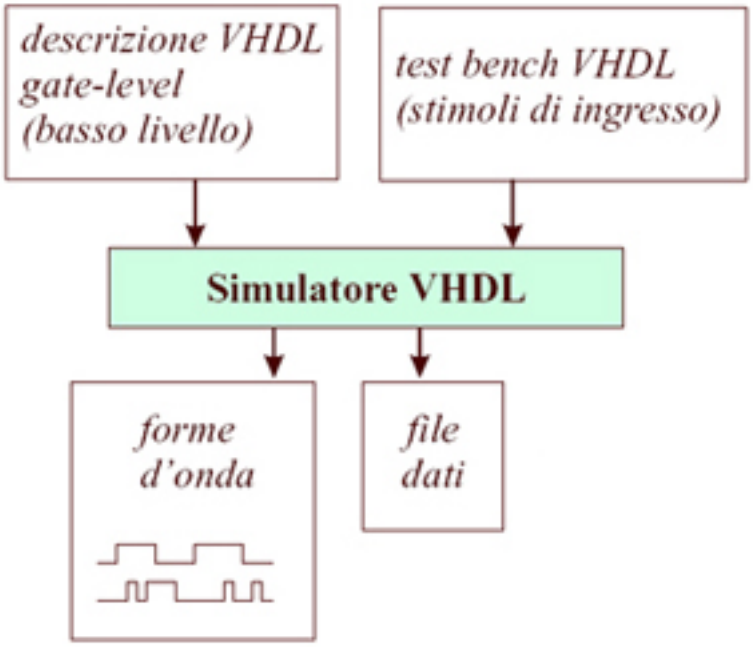


Figura 1.4: Verifica della sintesi

- *Sintesi*: passaggio automatico da una descrizione ad alto livello (comportamentale o RTL) ad una a basso livello (net-list).

In entrambe si utilizzano programmi CAD, anche completamente differenti, per le fasi di sintesi e di simulazione (sia la fase di sintesi che quella di simulazione prevedono un passo intermedio di "compilazione" del listato VHDL).

Per contro, la sintesi introduce anche degli svantaggi:

- *importanza del sintetizzatore*: si ha poco controllo nel definire l'implementazione gate-level di un sistema descritto ad alto livello (direttive di sintesi), se il sintetizzatore non prevede una buona configurabilità;
- *scarsa efficienza*: il circuito sintetizzato può non essere efficiente, molto spesso ciò è dovuto ad una descrizione VHDL inefficace (come un programma C scritto male può essere molto lento o richiedere eccessiva memoria, un codice VHDL scritto male può dar luogo ad una logica inutilmente complessa);
- *qualità del sintetizzatore*: la qualità del circuito sintetizzato varia da tool a tool. E' il problema sempre meno sentito, grazie al continuo miglioramento dei sistemi di sviluppo;

- *non sintetizzabilità* di parte del codice: alcune istruzioni del linguaggio VHDL, in particolare quelle sulla modellazione dei ritardi e sulla gestione dei file non hanno equivalente circuitale. In altre parole, il VHDL sintetizzabile è un sottoinsieme del linguaggio.

Nell'elaborato di tesi, ci si concentrerà sulla modellazione e la simulazione, rimandando le considerazioni sulla sintesi a sviluppi futuri.

Capitolo 2

Il VHDL

1 Astrazione dall'hardware

IL VHDL è utilizzato per descrivere un modello per un dispositivo hardware digitale. Un modello permette di definire la vista esterna del dispositivo ed una o più viste interne. La vista esterna costituisce l'*interfaccia* del dispositivo attraverso cui esso dialoga con il mondo esterno o con gli altri dispositivi del suo ambiente, mentre le viste interne specificano la funzionalità o la struttura.

Un dispositivo hardware può avere vari modelli che lo descrivono, questa è una conseguenza del diverso livello di astrazione con cui è possibile delineare il dispositivo in questione. Per esempio, un dispositivo modellato ad alto livello di astrazione potrebbe non avere un pin per il segnale clock, poiché il clock potrebbe non essere utilizzato in questa descrizione, mentre modellandolo a livello più basso è necessario sviluppare un modello che lo preveda; oppure il trasferimento di dati all'interfaccia potrebbe essere trattato in termini di valori interi, piuttosto che valori logici.

In VHDL, ciascun modello è considerato come una rappresentazione distinta di un unico dispositivo, chiamato *entity*. La figura 2.1 mostra la visione VHDL di un dispositivo hardware che ha diversi modelli, ciascuno dei quali caratterizzato da una entity. Sebbene le entity numerate dalla 1 alla N siano differenti dal punto di vista del VHDL, in realtà esse rappresentano lo stesso

dispositivo hardware.

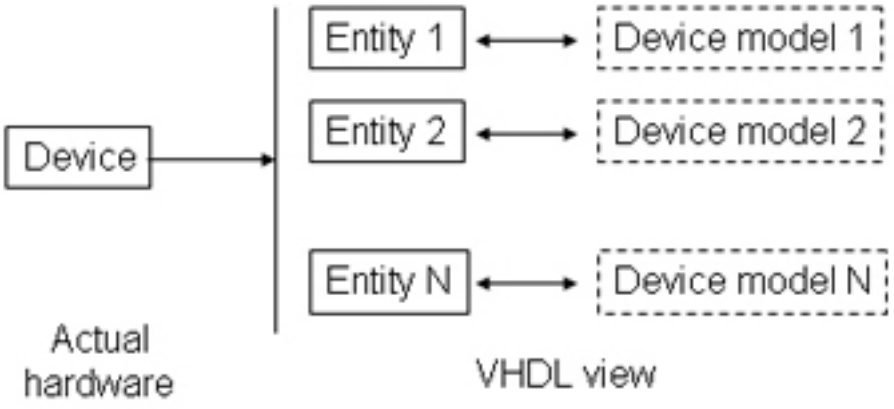


Figura 2.1: Visione equivalente di un dispositivo in VHDL

L'entity è quindi un astrazione dell'hardware, ciascuna entity è descritta utilizzando un modello, che (come detto) contiene una vista esterna e una o più viste interne. Allo stesso tempo, un dispositivo hardware può essere rappresentato da una o più entity.

Per descrivere una entity, il VHDL fornisce cinque differenti costrutti primitivi, detti *design units*. Essi sono:

- la dichiarazione di entità (*entity declaration*);
- il corpo dell'architettura (*architecture body*);
- la dichiarazione della configurazione (*configuration declaration*);
- la dichiarazione di package (*package declaration*);
- il corpo di un package (*package body*).

Una entity è modellata utilizzando una *entity declaration* ed almeno un'*architecture body*. L'entity declaration descrive la vista esterna dell'entità; per esempio, i nomi dei segnali di input e di output. La architecture body contiene invece la descrizione interna dell'entità; ad esempio, come set di componenti inter-connessi che rappresentano la struttura dell'entità, o un set di istruzioni concorrenti (o sequenziali) che rappresentano il comportamento dell'entità.

Ciascuno stile di rappresentazione, come spiegato meglio nel seguito, può essere specificato in un differente architecture body o mescolato all'interno del singolo architecture body.

2 Entità ed architetture

2.1 Entity Declaration

L'entity declaration specifica il nome dell'entità modellata e ne elenca le porte che costituiscono la sua interfaccia. Le *porte* sono i segnali tramite cui l'entità comunica con le altre entità cui è collegata o con il mondo esterno.

Un primo semplice esempio di entità, un multiplexer, è dato dal seguente codice:

```
entity MUX is
    port(
        I3 : in std_logic_vector(7 downto 0);
        I2 : in std_logic_vector(7 downto 0);
        I1 : in std_logic_vector(7 downto 0);
        I0 : in std_logic_vector(7 downto 0);
        S1 : in std_logic;
        S0 : in std_logic;
        O : out std_logic_vector(7 downto 0)
        );
end MUX;
```

che modella il seguente componente in figura 2.2.

Analizzando con maggior dettaglio il codice, si nota che sono complessivamente presenti 6 piedini di ingresso e uno di uscita. Ogni piedino corrisponde ad un segnale e deve avere una precisa direzione, assegnata mediante una di queste parole chiave:

- *in*: input port. Una variabile o un segnale può leggere un valore da questo tipo di porta, ma non è possibile assegnare ad essa un valore.

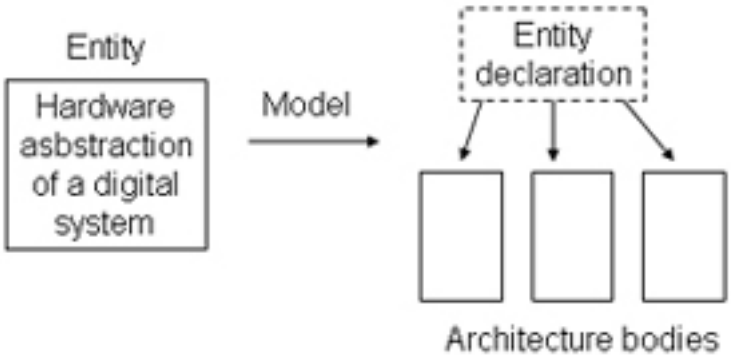


Figura 2.2: Un'entity e il suo modello

- *out*: output port. E' consentito assegnare uno stato logico in uscita a questa porta, ma non è possibile leggere un valore da essa.
- *inout*: è una porta bidirezionale, è possibile assegnarle un valore o leggere un valore da essa.
- *buffer*: è una porta di output, ma con possibilità di lettura. Differisce da una porta inout perchè il suo valore può essere aggiornato solo da una sorgente, mentre una porta inout può essere aggiornata da zero o più sorgenti.

Non tutti i piedini che compongono questa entità sono uguali: i piedini `[I0..I3]` e O sono modellati con segnali di 8 bit (ossia con un array di segnali), ciò equivale ad avere 8 dispositivi in parallelo, ciascuno con canali da un bit, mentre S0 e S1 sono normali pin singoli e servono per l'indirizzamento.

Un'ultima nota è sul tipo di segnali utilizzati in questo esempio: si sono utilizzati segnali std_logic, appartenenti alla libreria IEEE.std_logic_1164; essi sono caratterizzati da 9 livelli per definire il valore del segnale: "1, 0, H, L, W, Z, U, X, - "(per una maggiore descrizione dei tipi di dato, si veda il capitolo 3).

2.2 Architecture body

I dettagli interni di un'entity sono specificati da un'architecture body utilizzando uno dei seguenti stili di modellazione:

- *Stile stutturale*: come set di componenti inter-connessi;
- *Stile dataflow*: come set di istruzioni di assegnamento concorrenti;
- *Stile comportamentale*: come set di istruzioni di assegnamento sequenziali;
- Come combinazioni delle tre precedenti voci.

Stile di modellazione strutturale

Nello stile di modellazione strutturale un'entity è descritta come set di componenti inter-connessi. Definendo la seguente entity:

```
entity HALF_ADDER is
     port(
        A, B: in bit;
        SUM, CARRY: out bit
        );
end;
```

si può citare come esempio la struttura interna di un Half Adder:

```
architecture HA_STRUCTURE of HALF_ADDER is
   component XOR2
     port(
        X, Y: in bit;
        Z: out bit
        );
   end component;

   component AND2
     port(
        L, M: in bit;
        N: out bit
        );
   end component;

begin
```

```
   X1: XOR2 port map (A, B, SUM);
   A1: AND2 port map (A, B, CARRY);
end;
```

Il nome della architecture body (in seguito, chiamata più brevemente architettura) è HA_STRUCTURE. La entity declaration specifica le porte di interfaccia per questa architettura. L'architettura è composta da due sezioni: la parte *dichiarativa* (prima della keyword `begin`) e la parte *assegnativa* (dopo la keyword `begin`).
Due *component declaration* sono presenti nella parte dichiarativa, esse specificano l'interfaccia dei componenti utilizzati nell'architettura. I componeti XOR2 e AND2 potrebbero essere componenti predefiniti presenti in una libreria o, se non esistono, potrebbero essere collegati in seguito ad altri componenti presenti in una libreria.

I componenti sono istanziati nella seconda sezione, usando istruzioni di istanziazione. *X1* e *A1* sono le etichette associate alle istanziazioni dei componenti. La prima di queste mostra che i segnali A e B (che costituiscono le porte di ingresso dell'HALF_ADDER) sono connessi alle porte X e Y in ingresso del componente XOR2, mentre la porta in output Z di tale componente è connessa alla porta di uscita SUM dell'HALF_ADDER.
Similmente, nella seconda istruzione di istanziazione, i segnali A e B sono connessi alle porte L e M del componente And2, mentre la porta N è connessa alla porta CARRY dell'HALF_ADDER.

La rappresentazione strutturale del componente non dice nulla del suo effettivo funzionamento; infatti è necessario descrivere anche i componenti XOR2 e AND2, ciascuno dei quali deve avere la propria entità e la propria architettura.

Stile di modellazione dataflow

In questo stile di modellazione, il flusso di dati attraverso l'entità è espresso principalmente attraverso assegnamenti concorrenti di segnali. La struttura dell'entità non è esplicitamente specificata in questo stile di modellazione,

ma può essere dedotta implicitamente.

Si consideri la seguente architettura alternativa per l'entità HALF_ADDER introdotta precedentemente:

```
architecture HA_CONCURRENT of HALF_ADDER is
begin
   SUM <= A xor B after 8 ns;
   CARRY <= A and B after 4 ns;
end;
```

Si può vedere che, in questo caso, il modello dataflow utilizza due assegnazioni concorrenti: il simbolo <= indica che si sta effettuando l'assegnazione di un valore ad un segnale (si veda anche 3.3). Il valore dell'espressione sul lato destro dell'istruzione è calcolato ed assegnato al segnale presente sul lato sinistro dell'istruzione stessa, tale segnale è detto anche *target signal*.
Un'assegnazione concorrente ad un segnale è eseguita solamente quando si verifica un evento su uno qualsiasi dei segnali presenti sul lato destro dell'istruzione stessa.

Le informazioni sul ritardo di propagazione del segnale sono incluse nell'istruzione di assegnamento, grazie all'uso della clausola *after*. Se si registra un evento sui segnali A o B, al tempo T, entrambe le espressioni sono ricalcolate (perchè entrambe contengono almeno uno dei due segnali sul loro lato destro). Il segnale SUM è schedulato per avere in nuovo valore dopo 8 ns, mentre il segnale CARRY otterà il nuovo valore dopo 4 ns; quando il tempo di simulazione avavnzerà a (T + 4 ns), il segnale CARRY varierà e la simulazione avanzerà a (T + 8 ns) e anche SUM sarà aggiornato. Quindi, entrambe le istruzioni di assegnamento saranno eseguite in maniera concorrente e, per questa ragione, il loro ordine non è importante. Riguardo le temporizzazioni e il funzionamento della simulazione, se ne darà un visione più approfondita in 3.3.

Stile di modellazione comportamentale

In contrasto con gli stili affrontati precedentemente, quello comportamentale specifica il comportamento di un'entity come un set di istruzioni eseguite sequenzialmente in un ordine ben determinato. Questo set di istruzioni sequenziali, che sono specificate all'interno di un processo, non specificano esplicitamente la struttura dell'entità, ma principalmente il suo funzionamento.
Un *processo* può essere considerata una macro-istruzione concorrente, presente nell'architettura associata ad un'entity.

Si consideri ad esempio la seguente entità (modellante un decoder 2 x 4):

```
entity DECODER2x4 is
   port(
      A, B, EN: in bit;
      Z: out std_logic(3 downto 0)
      );
end;
```

con al seguente architettura in stile comportamentale:

```
architecture DEC_SEQ of DECODER2x4 is
begin
   process(A, B, EN)
      variable An, Bn: std_logic;
      begin
        An := not A;
        Bn := not B;
        if En = '1' then
            Z(3) <= not (A and B);
            Z(0) <= not (An and Bn);
            Z(2) <= not (A and Bn);
            Z(1) <= not(An and B);
        else
            Z <= "1111";
```

```
        end if;
    end process;
end;
```

Un processo ha (similmente all'architettura) una *parte dichiarativa* (prima della keyword begin) e una *parte di assegnamento* (tra le keyword begin e end process): le istruzioni presenti in questa sezione sono sequenziali e pertanto sono eseguite nell'ordine esatto con cui sono scritte. La lista dei segnali, specificata tra parentesi dopo la keyword process, costituisce la *sensitivity list* e il processo stesso è eseguito al verificarsi di un evento su uno dei segnali in essa presenti.

3 Strutture del linguaggio

In questa sezione si approfondiranno con maggior dettaglio i due pattern per modellare componenti in VHDL, partendo da quanto esaminato fino ad ora. L'uso di processi si richiama allo stile di modellazione comportamentale e si manifesta con una programmazione procedurale simile (per sintassi e semantica) a quella di linguaggi di programmazione di alto livello come il C o il Pascal.

Viceversa, l'uso di istruzioni concorrenti si richiama allo stile di modellazione dataflow ed è una delle caratteristiche distintive del linguaggio VHDL.

3.1 VHDL sequenziale: i processi

Il processo (*process*) è l'istruzione fondamentale delle descrizioni comportamentali in VHDL. Un costrutto process, all'interno di un'architettura, rappresenta una porzione di un progetto descritta dal punto di vista algoritmico: in questo senso, si pone l'accento sul suo comportamento (*cosa fa*), non sulla sua effettiva implementazione (*come lo fa*).

Un processo è uno statement concorrente (come una assegnazione o una istanziazione di un componente) che può essere usato solamente all'interno di una architettura ed è in grado di reagire "contemporaneamente" agli altri statement concorrenti. Un processo al suo interno contiene solo istruzioni sequenziali (assegnazioni sequenziali, if-then-else, case,...).

Un processo può essere identificato durante una simulazione attraverso una label (opzionale) prima della keyword *process*, ma è comunque costituito da 3 parti:

```
label: PROCESS (segnale_1,    , segnale_n);
   dichiarazioni di tipi;
   dichiarazioni di costanti;
   dichiarazioni di variabili;
   dichiarazioni di procedure e functions;
BEGIN
   istruzione sequenziale_1;

   istruzione sequenziale_N;
END PROCESS label;
```

Figura 2.3: Struttura di un processo

- una *sensitivity list* (opzionale) che contiene i trigger dei segnali in grado di attivare il processo stesso (in verde, in figura 2.3);
- una *parte dichiarativa*, che contiene le dichiarazioni dei tipi, di sottotipi, delle costanti, delle variabili, delle procedure e delle function, che potranno essere usate nel suo body (hanno visibilità locale) - (in blu, in figura 2.3);
- un *process body*, che rappresenta il comportamento del processo, specificato tramite un insieme di istruzioni eseguite in maniera sequenziale fra gli statement *BEGIN* ed *END PROCESS* (in viola, in figura 2.3);

Un *process body* consiste in un insieme di istruzioni sequenziali, il cui ordine di esecuzione è definito dall'utente e rappresentato dall'ordine con cui compaiono nel process body. Solo statement sequenziali sono leciti nel corpo di un processo: le assegnazioni `<=` sono lecite, in quanto l'assegnazione di un

segnale è considerata sequenziale se effettuata all'interno di un processo.

Il modello d'esecuzione di un processo è relativamente semplice. Ogni successiva esecuzione del processo, dopo quella di inzializzazione all'avvio della simulazione, è innescata da eventi che vanno esplicitamente indicati (nella sensitivity list, ad esempio, o in un'istruzione di wait); inoltre il processo esegue tutte le istruzioni sequenziali e poi le ripete ripartendo dall'inizio, il tutto ripetuto come in un loop infinito.

La stesura del codice di un processo, generalmente segue due template ben precisi:

- con *sensitivity list* (in figura 2.4);

```
PROCESS (segnale_1,    , segnale_n);
   dichiarazioni
BEGIN
   process body
END PROCESS;
```

Figura 2.4: Processo con sensitivity list

- con *istruzioni di tipo WAIT* (in figura 2.5);

```
PROCESS
   dichiarazioni
BEGIN

   WAIT   ;

END PROCESS;
```

Figura 2.5: Processo con istruzioni di WAIT

Non è lecito usare una sensitivity list assieme ad una istruzione WAIT nel medesimo processo, quindi i due templates sono alternativi. La differenza fra i due template è nell'attivazione e nella sospensione:

- nel caso di sensitivity list, il processo viene attivato da un evento su un segnale che appartiene alla sensitivity list e sospeso quando raggiunge la fine del processo stesso;
- nel caso di uso di istruzioni WAIT, quando il flusso di esecuzione incontra un WAIT, il processo viene sospeso e la sua esecuzione è ripresa quando la condizione richiesta dall'istruzione WAIT è verificata;

L'istruzione *wait* è strettamente sequenziale e quindi può comparire solo internamente ad un process body.
I possibili tipi di uno statement WAIT sono:

- *wait for waiting_time*: sospende il processo finché la condizione logica sulla durata temporale impostata non è soddisfatta. Il processo, sarà di fatto risvegliato dopo "waiting_time" unità di tempo;
- *wait on waiting_sensitivity_list*: sospende il processo fintanto chè non si verifica un evento su uno dei segnali presenti nella sensitivity list. Quando un evento coinvolge uno di questi, il processo è pronto per essere schedulato e riavviato;
- *wait until waiting_condition*: sospende il processo finchè il valore dell'espressione "waiting_condition" è false, viceversa il processo è risvegliato quando questa assume il valore true.
- *wait*: sospende indefinitamente il processo che la esegue. L'utilizzo dell'istruzione con questa semantica è particolarmente utile quando si vuol terminare il processo e assicurandosi che non sia più eseguito; ad esempio, un processo che si occupi di settare i valori dei segnali nel tempo va arrestato all'ultimo valore previsto, perchè altrimenti (riavviandosi ciclicamente) continuerebbe ad assegnare valori non corretti.

Un processo ha visibilità di tutti gli "oggetti" definiti nella sua architettura (tipi, sottotipi, costanti, segnali, procedure, functions,...): in altre parole lo *scope* di un processo è lo stesso della architettura che lo contiene.

L'unico modo che ha un processo per comunicare con l'esterno (e con altri process) è tramite i segnali di cui ha visibilità e che assegna e legge; in particolare non è possibile condividere variabili fra processi (perchè queste sono squisitamente locali al processo stesso, a meno delle *shared variable*

di cui si parlerà nella successiva sottosezione) e non è possibile dichiarare segnali all'interno di un processo (questi possono essere dichiarati solo nella definizione dell'architettura).

Variabili

Le variabili possono essere dichiarate ed utilizzate solamente all'interno di un processo. L'assegnamento ad una variabile è un'istruzione della forma: `value-object := expression;`. L'espressione è valutata quando l'istruzione è eseguita e il valore calcolato nell'espressione è assegnato immediatamente alla variabile stessa, quindi nell'istante temporale corrente. Una cosa importante da notare è che le operazioni di assegnamento a variabili non incrementano mai il tempo di simulazione, neanche se all'interno di loop.

Le variabili sono create al tempo dell'elaborazione e mantengono il loro valore durante l'intera simulazione (similmente alle variabili *static* presenti nel linguaggio C). Ciò è giustificato dal fatto che il processo non termina mai realmente, sia esso in esecuzione o sospeso in attesa di un evento.

Si consideri il seguente esempio:

```
process (A)
variable EVENTS_ON_A : integer := -1;
begin
   EVENTS_ON_A := EVENTS_ON_A + 1;
end process;
```

All'avvio della simulazione, il processo è eseguito una prima volta, questo per la politica di inizializzazione del simulatore. La variabile EVENTS_ON_A è inizializzata a -1 e quindi incrementata di uno. Completata la fase di inizializzazione, ogni volta che un evento avviene sul segnale A, il processo si attiva e viene eseguita l'istruzione di incremento della variabile. Al termine della simulazione, la variabile conterrà il numero totale di eventi occorsi sul segnale A.

Shared variable

Una variabile dichiarata all'esterno di un processo o di un sottoprogramma si definisce *shared variable* (variabile condivisa). Una variabile condivisa può essere letta e aggiornata da più di un processo. Tuttavia, l'interpretazione delle variabili condivise non è fornita direttamente dal linguaggio, pertanto non dovrebbero essere utilizzate.

Segnali

I segnali sono valori assegnati tramite un'istruzione di assegnamento. La forma più semplice per assegnare un valore ad un segnale è:
`signal-object <= expression [after delay-value];`

Un'istruzione di assegnazione ad un segnale può apparire sia all'interno, che all'esterno di un processo. Se è presente all'esterno di un processo, deve essere considerata concorrente, viceversa sarà strettamente sequenziale e quindi sarà eseguita in sequenza rispetto all'ordine con cui le istruzioni appaiono nel processo stesso.

Quando un'istruzione di assegnazione di un segnale è eseguita, il valore dell'espressione è calcolato e schedulato per essere assegnato al segnale dopo uno specificato ritardo. *È* importante notare che l'espressione è valutata al tempo in cui l'istruzione è raggiunta (ossia, al tempo di simulazione corrente), non dopo il ritardo specificato: sarà l'effettivo aggiornamento del valore del segnale ad essere ritardato.

Istruzioni aggiuntive per modellare i ritardi

Modellare un ritardo nella propagazione di un segnale, è un'operazione relativamente semplice in VHDL. Sono però presenti più istruzioni, che rendono possibile ciò, ciascuna delle quali presenta della sfumature nel suo comportamento. Se ne darà una visione più approfondita anche nella sez. 3.6, relativa ai driver associati ai segnali.

- *inertial delay*: è espresso con l'assegnamento:
  `signal-object <= value after X ns;`

questo ritardo impone l'aggiornamento del segnale dopo X ns (espressi in modo assoluto, ossia rispetto all'istante zero della simulazione) dalla sua valutazione; al tempo stesso, filtra variazioni del segnale in ingresso che abbiano durata minore di X ns. Il problema che si manifesta è che il ritardo inerziale modella il componente fisico come se avesse una sua inerzia e una capacità di filtrare i glitch, mentre il modello proposto dalla istruzione after del VHDL assume che il ritardo di propagazione e il ritardo inerziale siano la stessa cosa.

- *transport delay*: è espresso con l'assegnamento:
  `signal-object <= transport value after X ns;`
  ha comportamento simile al caso precedente, ma non filtra i glitch di durata minore al ritardo.
- *reject inertial*: è espresso con l'assegnamento:
  `signal-object <= reject Y ns inertial value after X ns;`
  in questo caso, il segnale è ritardato di X ns, solo se la sua durata è stata superiore a Y ns, viceversa non viene propagato.

Delta delay

Alcuni eventi devono poter essere schedulati con un ritardo nullo, questo perchè - soprattutto ai livelli più alti - si astrae dai ritardi reali dei singoli componenti. Questo fatto può generare dei problemi: il VHDL non permette di variare il valore attuale di un segnale istantaneamente, in quanto tale aggiornamento istantaneo di un valore letto da un altro processo concorrente, renderebbe l'esecuzione non deterministica. Si potrebbe inoltre verificare il mancato rispetto dei vincoli temporali di causa-effetto.

Nell'assegnazione di un segnale, se non è specificato un ritardo, o se il ritardo è di 0 ns, si assume la presenza di un ritardo di durata delta. Un delta delay è una quantità di tempo infinitesimale che non ha significato fisico e non causa il cambiamento del tempo di simulazione. Il meccanismo del delta delay garantisce l'ordinamento degli eventi sui segnali che si manifesterebbero nello stesso istante temporale.

Ciascuna unità del tempo di simulazione può essere considerata come composta di un numero infinito di delta delay. Quindi un evento avverrà sempre in corrispondenza di un istante di simulazione più una certa quantità di delta. Ad ogni modo, il segnale è considerato stabile e può essere campionato da un altro processo, solo in corrispondenza all'ultimo delta delay del suo istante d'aggiornamento.

Consideriamo il seguente processo, in cui TEMP1 e TEMP2 sono variabili, mentre A, B, C, D e Z sono segnali:

```
process (A, B, C, D)
variable TEMP1, TEMP2 : std_logic;
begin
   TEMP1 := A and B;
   TEMP2 := C and D;
   TEMP1 := TEMP1 or TEMP2;
   Z <= not TEMP1;
end process;
```

Assumiamo che avvenga all'istante T un evento sul segnale D: essendo un processo, le istruzioni sono eseguite in sequenza. A TEMP1 è assegnato immediatamente un valore essendo una variabile. Quindi è eseguita la seconda istruzione e anche a TEMP2 è assegnato subito un valore (essendo anche essa una variabile). Quindi è eseguita la terza istruzione, che determina il nuovo valore di TEMP1. Infine è eseguita la quarta istruzione, questa viene calcolata e il risultato è schedulato per l'istante T + 1 delta. Solamente quando il tempo di simulazione si incrementerà a T + 1 delta il nuovo valore sarà assegnato a Z, viceversa se ci fossero state altre istruzioni a seguire che usavano il valore di Z, esse avrebbero utilizzato ancora il valore vecchio associato al segnale. Come descritto nel seguito, l'aggiornamento del valore di un segnale all'interno di un processo si verifica al raggiungimento di una istruzione di wait (si veda la figura 2.6).

Time	Delta	A	B	C	D	Z
0.000	0	0	0	1	0	U
0.000	1	0	0	1	0	1
10.000 ns	0	0	0	1	1	1
10.000 ns	1	0	0	1	1	0

Figura 2.6: List del valore dei segnali considerati nell'esempio

Un altro esempio, aiuterà a comprendere meglio le tempistiche di variazione dei segnali:

```
PZ: process (A)   -- A e Z sono segnali
variable V1, V2 : integer;
begin
   V1 :=  A - V2;
   Z <= -V1;
   V2 := Z + V1 * 2;
end process;
```

Se un evento si verifica sul segnale A al tempo T, l'esecuzione della prima istruzione causa l'aggiornamento immediato del valore della variabile V1, quindi il segnale Z viene schedulato per assumere un nuovo valore all'istante T + 1 delta. Quindi è eseguita l'istruzione 3 con il vecchio valore di Z, la ragione di ciò è che il tempo di simulazione è ancora all'istante T e non è ancora avanzato all'istante T + 1 delta, in questo caso l'incremento del tempo di simulazione all'istante T + 1 delta (e quindi l'aggiornamento del segnale Z) avverrà al termine del loop corrente: si rammenti che la presenza della sensitivity list ha lo stesso comportamento dell'istruzione *wait on condition* come ultima istruzione del processo.

Uso di *Wait for 0*

L'istruzione wait, come visto in precedenza, è spesso utilizzata per sospendere un processo in attesa di una condizione temporale o di un evento su un segnale, ma questo non è l'unico uso che se ne può fare. In certe situazioni,

può rendersi necessario variare il valore di un segnale ed utilizzarne il nuovo valore all'interno del loop corrente del processo; questo sembrerebbe impossibile da realizzarsi, a meno di copiare su una variabile d'appoggio il valore attuale del segnale e lavorare su quest'ultima. In realtà è possibile risolvere l'empasse, utilizzando l'istruzione *wait for 0*.

Usare il costrutto `wait for 0 ns`, significa attendere un delta cycle. Si consideri il seguente esempio:

```
process
begin
   wait on DATA;
   A <= DATA;
   wait for 0 ns;
   B <= A;
end process;
```

Se il segnale DATA varia all'istante 10 ns, A è schedulato per assumere il nuovo valore all'istante 10 + 1 delta. Il `wait for 0 ns` causa la sospensione del processo per un delta, quindi il tempo di simulazione sarà fatto proseguire all'istante 10 + 1 delta ed il processo sarà risvegliato. Chiaramente, il segnale A assumerà ora il nuovo valore. Il segnale B viceversa assumerà il nuovo valore all'istante 10 + 2 delta. Se non fosse stata presente l'istruzione wait for 0 ns, entrambe le istruzioni di assegnamento sarebbero state eseguite sequenzialmente all'istante temporale 10 ns e gli aggiornamenti dei valori sarebbero stati schedulati entrambi per l'istante 10 + 1 delta. Ma a quel punto i valori cui si sarebbe fatto riferimento sono i vecchi valori assunti dai segnali all'istante 10 ns. Il tutto è chiarificato dalle figure 2.7 e 2.8.

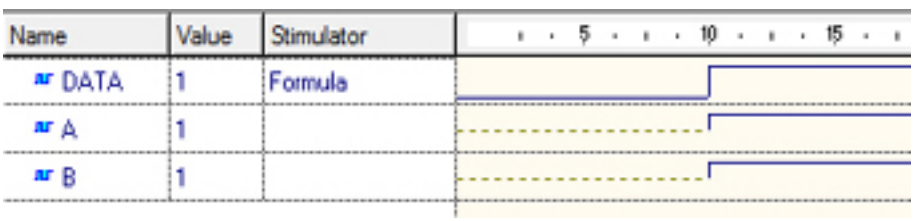


Figura 2.7: Forme d'onda per i segnali in esempio

Time	Delta	DATA	A	B
0.000	0	0	U	U
10.000 ns	0	1	U	U
10.000 ns	1	1	1	U
10.000 ns	2	1	1	1

Figura 2.8: Rappresentazione a lista per i segnali in esempio

Feedback volontari e involontari

La propagazione dei segnali all'interno dei processi può introdurre dei *feedback*, ossia un campionamento dei segnali d'uscita. Questo può essere perfettamento lecito e volontario, si pensi ad un contatore modellato dal seguente codice:

```
process
begin
   wait until ck = '1';
   count <= count + 1;
end process;
```

Al fronte positivo del clock, supponiamo all'istante T, il valore attuale del segnale *count* è ad esempio 4, l'espressione sarà valutata e il simulatore schedulerà per l'istante T + 1 delta l'assegnazione del valore 5 al segnale count. Ovviamente se ci fosse stata un'altra istruzione con il segnale count sul lato destro dell'espressione, il valore assunto dallo stesso sarebbe stato quello avuto nell'istante T, ossia 4 (e non il nuovo valore, 5), visto che l'avanzamento del tempo di simulazione avviene al termine del loop.

Il circuito equivalente al codice è riportato in fig. 2.9.

Viceversa, se non si modella correttamente il circuito potrebbero essere inseriti dei registri laddove non dovrebbero essere presenti. Un errore abbastanza comune è quello di modellare una rete combinatoria con un processo - cosa di per se lecita - senza riportare nella sensitivity list tutti i segnali che provocano la rivalutazione delle espressioni stesse. In tal caso, il variare di un segnale

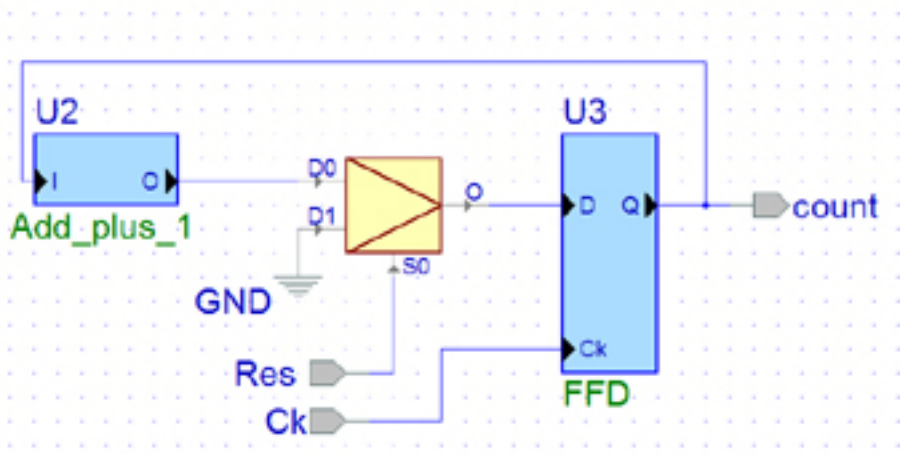


Figura 2.9: Schema circuitale contatore

non compreso nella sensitivity list non porta all'esecuzione del processo, con effetti potenzialmente erronei.

Un'altra situazione non ottimale, sempre nelle reti combinatorie, si ha quando in un costrutto IF..THEN..ELSE manca l'*else*. Si consideri il seguente esempio:

```
process(b, enable)
begin
   if enable = '1' then
      z <= b;
   end if;
end process;
```

In questo caso, sebbene tutti i segnali in ingresso alla rete combinatoria siano presenti nella sensitivity list, il comportamento simulato non sarà completamente combinatorio, in quanto se `enable = 0` il valore di z è mantenuto. Se l'istruzione IF è incompleta e non c'è un precedente assegnamento (come nel caso in esempio) prima della stessa, si crea un feedback involontario che continua a mantenere il valore precedente.

Un altro feedback involontario è introdotto se il valore di un segnale combinatorio non viene aggiornato in tutti i rami di un IF annidato. Si consideri

il seguente esempio:

```
process
begin
   wait on a, b, c;
   if c = '1' then
     z <= a;
   else
     y <= b;
   end if;
end process;
```

In questo caso, sebbene l'istruzione di IF sia completa, segnali (combinatori) differenti sono assegnati nei vari rami. Ne consegue che i segnali z e y manterranno alternativamente il valore precedente nelle varie esecuzioni del processo. Il valore è campionato infatti da un feedback asincrono che non dovrebbe esserci, perchè i segnali stessi sono combinatori.

Per evitare queste spiacevoli situazioni è necessario seguire due regole: assicurarsi che ogni assegnazione di un valore ad un segnale presente in un'istruzione IF sia presente in ogni ramo dell'IF stesso e che ci sia sempre l'else. In alternativa, si deve inizializzare ogni segnale coinvolto nell'istruzione di IF con un'assegnazione non condizionata, prima dell'IF stesso.

3.2 Forme alternative di assegnazione sequenziale

All'interno di un processo, è possibile utilizzare un insieme completo di istruzioni sequenziali, in maniera del tutto simile a quanto è possibile fare con un qualsiasi linguaggio di programmazione.

Il costrutto IF...THEN...ELSE

Il costrutto *if..then* permette di controllare il flusso d'esecuzione all'interno di un process body.

Come è possibile vedere nel seguente codice:

```
IF condition_1 THEN
   sequenza_istruzioni;
ELSIF condition_2 THEN
   sequenza_istruzioni;
ELSE
   sequenza_istruzioni;
END IF;
```

Le condizioni sono delle espressioni booleane che, se vere, abilitano l'esecuzione del ramo relativo composto da statement sequenziali. Il costrutto può anche contenere degli ELSIF che sono eseguiti quando le precedenti condizioni non sono verificate. E' possibile anche usare una clausola ELSE per raccogliere i casi esclusi da tutti i rami precedenti.
I costrutti IF...THEN sono utili quando i casi non sono tutti mutuamente esclusivi e si vuole stabilire una priorità di esecuzione nei confronti.

Il costrutto CASE...WHEN

Il funzionamento di questo costrutto è abbastanza simile al precedente, ma è preferibile quando i casi sono mutuamente esclusivi:

```
case segnale/variabile_di_selezione is
   when caso1 => istruzioni;
   when caso2 => istruzioni;
   when casoN => istruzioni;
end case;
```

L'istruzione *case* seleziona uno dei rami in maniera esclusiva, in base al valore dell'espressione. Il valore dell'espressione deve essere di tipo discreto o un array mono-dimensionale. Le scelte devono essere espresse come valori singoli, o con range di valori separati dal carattere '|' , o con l'uso della keyword *others*.
E'importante sottolineare che tutti i possibili valori dell'espressione devono essere univocamente coperti da un ramo di scelta, per questo è sempre buona norma indicare il ramo con la scelta di default, utilizzando la keyword others.

Il costrutto LOOP

Il costrutto *loop* è utilizzato per iterare attraverso un set di istruzioni sequenziali. La sintassi dell'istruzione è:

```
[loop-label:] iteration-scheme loop
                  sequential-statements;
              end loop;
```

Ci sono tre tipi di *iteration-scheme*. Il primo è l'iteratore *for*, il quale si esprime nella forma: `for <ident.> in <range>` (ad esempio: `for number in 0 to N loop`, il corpo del loop sarà eseguito (N - 1) volte con la variabile di conteggio (number) incrementata automaticamente di un'unità al termine del loop stesso). Similmente a quanto avviene in altri linguaggi, il controllo sulla variabile viene eseguito ad inizio ciclo (se l'espressione è vera, ossia se la variabile di incremento è minore del limite).

Il secondo iteratore è il *while*, che si esprime nella forma `while <condition>` (ad esempio: `while J < 20 loop`, il corpo del loop è eseguito finchè è verificata la condizione; quando essa diviene falsa, l'esecuzione continua con le istruzioni successive al loop).

Il terzo iteratore è quello in cui non è specificato alcuno schema di iterazione. In questa forma di loop, tutte le istruzioni contenute nel loop stesso sono eseguite ciclicamente, finchè qualche altra causa forzi il loop ad essere interrotto. L'azione di apertura potrebbe essere causata da un'istruzione di *exit* (uscita dal ciclo ed esecuzione della prima istruzione fuori ciclo) o da una di *next* (uscita dal ciclo e ritorno alla condizione di inizio ciclo, praticamente coincidente con la funzione *continue* presente nei linguaggi di alto livello).

Assert

Le istruzioni di *assert* sono utili per modellare vincoli imposti ad un'entità. Ad esempio, se è necessario verificare che il valore di un segnale si mantenga in uno specificato range, o si vuole controllare il rispetto dei tempi di hold e setup per i segnali che arrivano ai piedini di input dell'entità. Se la verifica

fornisce un esito negativo, il sistema genererà un opportuno messaggio.

La sintassi dell'istruzione di assert è la seguente:

```
assert boolean-expression
   [report string-expression]
   [severity expression];
```

Se il valore dell'espressione booleana è falso, il messaggio di report è stampato sullo standard di uscita (in fase di simulazione); l'espressione nella clausola severity deve essere un valore del tipo SEVERITY_LEVEL (un tipo di dato enumerativo predefinito con i valori NOTE, WARNING, ERROR and FAILURE). Il severity level è tipicamente usato dal simulatore per generare le azioni di risposta appropriate ad una condizione anomala. Ad esempio, in caso di fallimento di un'asserzione, se questa coinvolge segnali importanti, potrebbe essere preferibile abortire la simulazione stessa.

3.3 VHDL concorrente

L'assegnazione concorrente di segnali è una delle caratteristiche della modellazione *dataflow* di un'architettura; quest'ultima può avere qualsiasi numero di assegnazioni concorrenti e l'ordine delle stesse non è importante. Un'assegnazione concorrente è eseguita, quando un evento si verifica su almeno uno dei segnali presenti nell'espressione.

Una delle differenze più importanti tra segnali sequenziali e segnali concorrenti, che sarà ora discussa, riguarda la differente gestione delle tempistiche nella variazione dei segnali stessi.

Si consideri il seguente frammento di codice di un'architettura sequenziale:

```
architecture seq of fragment1 is
-- A, B e Z sono segnali,
-- e questo (con i due \"--\" iniziali)
-- è un commento!
```

```
begin
   process (B)
     begin
       A <= B;
       Z <= A;
   end process;
end architecture seq;
```

Nel momento in cui il segnale B ha un evento (ad esempio al tempo T), dapprima è eseguita la prima istruzione (`A <= B`), quindi la seconda (`Z <= A`), entrambe sono eseguite in un tempo nullo (senza ritardi di propagazione). Però, il segnale A è schedulato per acquisire il nuovo valore del segnale solo al tempo T + 1 delta, ed anche Z è schedulato per acquisire al tempo T + 1 delta il segnale di A. In definitiva, al termine dell'istante di simulazione T, A avrà il nuovo valore di B, mentre Z avrà il vecchio valore di A (si veda, a tal proposito la figura 2.10).

Time	Delta	A	B	Z
0.000	0	U	0	U
0.000	1	0	0	U
10.000 ns	0	0	1	U
10.000 ns	1	1	1	0
20.000 ns	0	1	0	0
20.000 ns	1	0	0	1

Figura 2.10: Rappresentazione a lista per i segnali in esempio

Si consideri, invece, quest'altro frammento di codice di un'architettura concorrente:

```
architecture conc of fragment1 is
begin
   A <= B;
   Z <= A;
end;
```

Quando un evento avviene sul segnale B, al tempo T, il segnale A assumerà il valore di B al tempo T + 1 delta. In questo caso per evitare ambiguità sull'effettivo valore di Z, il tempo di simulazione viene fatto avanzare a T + 1 delta prima che Z assuma il nuovo valore di A. Solo a questo punto A assume il nuovo valore, ciò evidentemente provoca un evento sulla seconda espressione, che causa l'assegnazione a Z del nuovo valore di A nell'istante T + 2 delta. Quindi solo nell'istante di simulazione T + 2 delta, Z sarà uguale a B.
Si noti che si considera sempre come valore stabile di un segnale, quello corrispondente all'ultimo delta dell'istante temporale in cui lo stesso è variato (si veda la figura 2.11).

Time	Delta	A	B	Z
0.000	0	U	0	U
0.000	1	0	0	U
0.000	2	0	0	0
10.000 ns	0	0	1	0
10.000 ns	1	1	1	0
10.000 ns	2	1	1	1
20.000 ns	0	1	0	1
20.000 ns	1	0	0	1
20.000 ns	2	0	0	0

Figura 2.11: Rappresentazione a lista per i segnali in esempio

Delta delay (in concorrenza)

In un'istruzione di assegnamento ad un segnale, se non è specificato un ritardo (o se il ritardo è di 0 ns), si assume sia presente un delta delay. Un delta delay è una quantità di tempo infinitesimale, non è un tempo fisico e non causa la variazione del tempo di simulazione reale. Il meccanismo del delta delay consente di imporre un ordinamento agli eventi che si manifestano su segnali nello stesso istante di simulazione.

Si consideri ad esempio, il tratto di circuito mostrato in figura 2.12, esso può essere modellato con il seguente codice:

```
entity fast_inverter is
   port(
       A: in std_logic;
       Z: out std_logic
       );
end;

architecture delta of fast_inverter is
signal B, C: std_logic;
begin
   Z <= not C;
   C <= not B;
   B <= not A;
end;
```

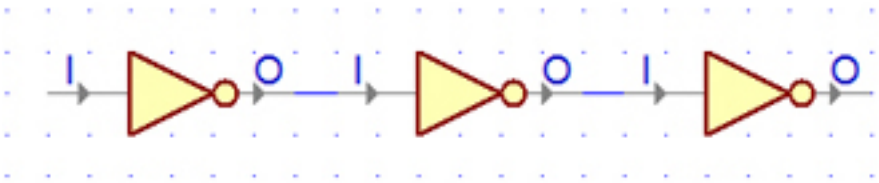


Figura 2.12: Tre invertitori ideali in cascata

Ci sono tre assegnamenti che utilizzano i delta delay. Quando un evento avviene sul segnale A, al tempo T; la terza istruzione rileva il cambiamento e schedula la variazione del segnale B all'istante di tempo T + 1 delta. Quando il tempo di simulazione avanza all'istante T + 1 delta, il valore di B viene aggiornato. Questa operazione però causa il ricalcolo della seconda espressione ed il nuovo valore segnale C sarà schedulato per essere applicato all'istante T + 2 delta. Quando il tempo di simulazione arriva a T + 2 delta, il valore di C è aggiornato e si deve ricalcolare la prima espressione: il segnale Z sarà schedulato per il cambiamento all'istante T + 3 delta.
In definitiva, anche il reale tempo di simulazione è rimasto fisso a T ns, il

segnale Z è stato aggiornato correttamente con una sequenza di 3 delta; se un altro processo o segnale avesse richiesto la lettura del segnale Z all'istante T, avrebbe comunque ottenuto il valore corretto (ossia quello risultante a T + 3 delta, si veda la figura 2.13).

Time	Delta	A	Z	B	C
0.000	0	0	U	U	U
0.000	1	0	U	1	U
0.000	2	0	U	1	0
0.000	3	0	1	1	0
10.000 ns	0	1	1	1	0
10.000 ns	1	1	1	0	0
10.000 ns	2	1	1	0	1
10.000 ns	3	1	0	0	1

Figura 2.13: Rappresentazione a lista per i segnali in esempio

3.4 forme alternative di assegnazione concorrente

Istruzioni di assegnamento Conditional

Un'istruzione di assegnamento *condizionato* seleziona differenti valori per il segnale cui è destinata in base a condizioni specifiche, differenti tra loro. Di fatto, assomiglia al costrutto IF sequenziale esaminato in 3.2.

Una tipica sintassi per questa istruzione è la seguente:

```
target-signal <= [waveform-elements when condition else]
                 [waveform-elements when condition else]
                     ....
                 waveform-elements [when condition];
```

Quando un evento si verifica su uno dei segnali utilizzati in una delle condizioni o su uno dei segnali che compaiono nella forma d'onda specificata,

è eseguita l'assegnazione condizionale valutando le condizioni una alla volta: per la prima condizione soddisfatta, il corrispondente valore della forma d'onda viene schedulato per essere assegnato al segnale.

Si consideri il seguente esempio:

```
Z <= IN0 after 10ns when S0 = '0' else
     IN1 after 10ns;
```

L'istruzione è eseguita ogni volta che un evento si verifica sui segnali IN0, IN1 e S0. La prima condizione è verificata, se falsa, si verifica la seconda: qui risulterà sempre vera e il valore di IN1 sarà schedulato per essere assegnato al segnale Z dopo 10ns.
Ovviamente, per ogni istruzione di assegnamento condizionato esiste l'equivalente istruzione sequenziale: essa si realizza all'interno di un processo, la cui sensitivity list deve essere sensibile a tutti i segnali coinvolti, attraverso il costrutto IF..THEN..ELSE.

Istruzioni di assegnamento Selected

Un'istruzione di assegnamento *selected* seleziona differenti valori per il segnale di destinazione in base al valore della espressione di select. Di fatto, assomiglia al costrutto CASE..SELECT sequenziale esaminato in 3.2.

Una tipica sintassi per questa istruzione è la seguente:

```
with expression select
   target-signal <= waveform-elements when choices,
                    waveform-elements when choices,
                        ...
                    waveform-elements when choices;
```

Quando un evento si verifica su uno dei segnali utilizzati in una delle condizioni o su uno dei segnali che compaiono nella forma d'onda specificata, l'istruzione è eseguita. Il valore della espressione nella clausola select corrisponderà ad uno dei valori delle possibili scelte (ossia ad una delle *choices*)

e il segnale sarà schedulato per assumere il valore corrispondente.
Si noti che le scelte non sono valuate in sequenza, ma parallelamente (così come accadeva nel costrutto CASE..SELECT). Inoltre, tutte i possibili valori assunti dalla espressione select devono avere la corrispondente choice (una e una sola), pertanto spesso si può usare la keyword *others* per raggruppare tutti i possibili valori non espressi altrimenti.

Un esempio, può essere il seguente:

```
type OP is (ADD, SUB, MUL, DIV);
signal OP_CODE: OP;
    ...
with OP_CODE select
   Z <= A+B after 10ns when ADD,
        A-B after 10ns when SUB,
        A*B after 10ns when MUL;
        A/B after 10ns when DIV;
```

Quando un evento si verifica su uno dei segnali OP_CODE, A o B, l'istruzione è eseguita. Assumendo che il valore corrente del segnale OP_CODE sia SUB, l'espressione A-B è calcolata e il suo valore è schedulato per essere assegnato al segnale Z dopo 10ns.

3.5 Registri

Il VHDL non prevede un'istruzione del linguaggio che descriva semanticamente un registro. Fisicamente, un progettista è abituato a lavorare con componenti elettronici basilari come flip-flop o registri, ma in VHDL non esiste un componente con memoria primitivo. L'unica soluzione per realizzare componenti con memoria è utilizzare un processo: grazie alla sequenzializzazione delle istruzioni e alla modalità di variazione dei segnali nel tempo (come esaminato in 3.1) è possibile implementare un set di istruzioni che ne modelli il comportamento.

Un primo esempio di registro modellato in VHDL è il flip-flop D, riportato nel seguente codice:

```
entity Dtype is
   port(
       d, ck : in bit;
       q : out bit
       );
end;

architecture behaviour of Dtype is
begin
   process
     begin
       wait on ck until ck = '1';
       q <= d;
     end process;
end;
```

Il modello appena descritto, se simulato, ha un comportamento equivalente ad un registro *edge-triggered* (ossia ad un flip-flop). In realtà, quella mostrata non è l'unica forma in cui può essere rappresentato un registro: esistono quattro differenti template, ovviamente basati su processi.

Basic template

E' il template più semplice, il suo corpo è costituito da un processo con solo due istruzioni sequenziali:

```
process
begin
   wait on ck until ck =  '1';
   q <= d;
end process;
```

Short template

Il secondo template opera esattamente come il primo, in simulazione, e si basa sul fatto che la clausola *on* può essere omessa: in questo caso il com-

pilatore sottintende una clausola on contenente tutti i segnali utilizzati poi nella condizione *until*.

```
process
begin
   wait until ck = '1';
   q <= d;
end process;
```

IF statement template

```
process
begin
   wait on ck;
   if ck = '1' then
      q <= d;
   end if;
end process;
```

Questo template si basa sull'uso di una condizione IF..THEN. L'istruzione di wait non ha una clausola until come nei casi precedenti: ciò significa che il processo sarà attivato ad ogni evento sul segnale di clock, indipendentemente dal tipo di evento (in altre parole, sia sul fronte positivo, sia sul fronte negativo). L'istruzione IF quindi agirà da filtro, considerando solamente i fronti positivi del clock. Se il segnale clock è basso, allora l'istruzione di assegnamento non sarà eseguita: il segnale q manterrà il suo valore. Ciò significa che il fronte negativo non ha effetto sulla variazione del valore dell'uscita, quindi il processo modella semanticamente un flip-flop. Si approfondirà ulteriormente la semantica del mantenimento nella sezione 3.1.

Sensitivity-list template

Un processo può avere una sensitivity list anzichè una condizione di wait per specificare il set di segnali che causano l'attivazione del processo stesso. La forma di questo template è data dal seguente codice:

```
process (ck)
begin
   if ck = '1' then
      q <= d;
   end if;
end process;
```

In questo esempio, il processo rimane sospeso finchè non avviene un evento su uno dei segnali della sensitivity list. Questo attiva il processo che esegue per una sola volta e quindi si sospende nuovamente. Anche in questo template, il processo è attivato su entrambi i fronti del clock, ma la condizione IF interna filtra via i fronti negativi.

Il *sensitivity-list template* è equivalente all'*IF-statement template*, ma con l'istruzione di wait spostata in fondo al processo. Ciò significa che i due template sono equivalenti in sintesi, ma differenti nella simulazione. Questa differenza può essere sfruttata in modo che le simulazioni si inizializzino correttamente senza alcun impatto sul risultato della sintesi.

La regola generale è che l'istruzione di wait possa essere posizionata ovunque nel processo, sebbene in generale sia posta o all'inizio o alla fine dello stesso. Alcuni sintetizzatori potrebbero non supportare tutte le possibili permutazioni, ma dovrebbero almeno supportare quelle ai due estremi.
I processi con l'istruzione di wait alla fine sono interamente eseguiti all'avvio della simulazione (inizializzazione automatica), mentre i processi con il wait all'inizio non lo sono (e quindi l'inizializzazione al valore corretto delle variabili interne va fatta manualmente).

Registri a più bit

E' possibile modellare anche registri di più bit, in maniera tale da campionare ad esempio il valore di un bus. La realizzazione di un registro ad 8 bit, ad esempio, è mostrata dal seguente codice:

```
entity Dtype is
   port(
```

```
       d : in signed(7 downto 0);
       ck : in bit;
       q : out signed(7 downto 0)
       );
end;

architecture behaviour of Dtype is
begin
   process
     begin
       wait on ck until ck = '1';
       q <= d;
   end process;
end;
```

Ma il modello non si limita solamente a campionare un segnale, qualsiasi numero di segnali può essere gestito nello stesso processo:

```
process
begin
   wait on Ck until Ck = '1';
   q0 <= d0;
   q1 <= d1;
   q2 <= d2;
end process;
```

Gated Register

I modelli visti fino ad ora non prevedono un segnale che abiliti o disabiliti il campionamento, praticamente seguono l'ingresso ad ogni fronte positivo del clock eciò non sempre è utile. Per questa ragione è necessario ampliare ulteriormente il modello, introducendo due modalità di controllo: *clock gating* e *data gating*.

Clock Gating

Questa modalità prevede di inserire un segnale di controllo che agisca sul clock ed eventualmente lo mascheri, affinchè il registro non possa aggiornare il suo stato. Tuttavia questa è una soluzione che non dovrebbe essere utilizzata, principalmente per due motivi. Il primo di essi è che gli strumenti di testing automatico, in fase di sintesi, utilizzano tecniche di scanning circuitale che necessitano di poter pilotare tutti i segnali di clock presenti. Il secondo motivo è che gli algoritmi utilizzati nella sintesi logica per la minimizzazione non garantirebbero il funzionamento *glitch-free* della logica di pilotaggio.

Data Gating

Questa è la soluzione più sicura ed utilizzata per dotare un registro di un segnale di enable. Il *data gating* è così chiamato perchè si inserisce sull'input dei dati al registro e non sul clock, quindi il registro continua ad essere normalmente alimentato dal clock, che non viene mai fermato. Il funzionamento avviene fornendo, con un ramo in *retroazione*, all'ingresso dati del flip-flop il valore dell'uscita, quando il segnale di enable è inattivo.

Il circuito in figura 2.14 utilizza un *multiplexer* per rendere possibile tale realizzazione; il codice VHDL che modella tale circuito è il seguente:

```
entity Dtype is
   port(
       d, ck, en : in bit;
       q : out bit
       );
end;

architecture behaviour of Dtype is
begin
   process
     begin
       wait on ck until ck = '1';
       if en = '1' then
```

```
            q <= d;
        end if;
    end process;
end;
```

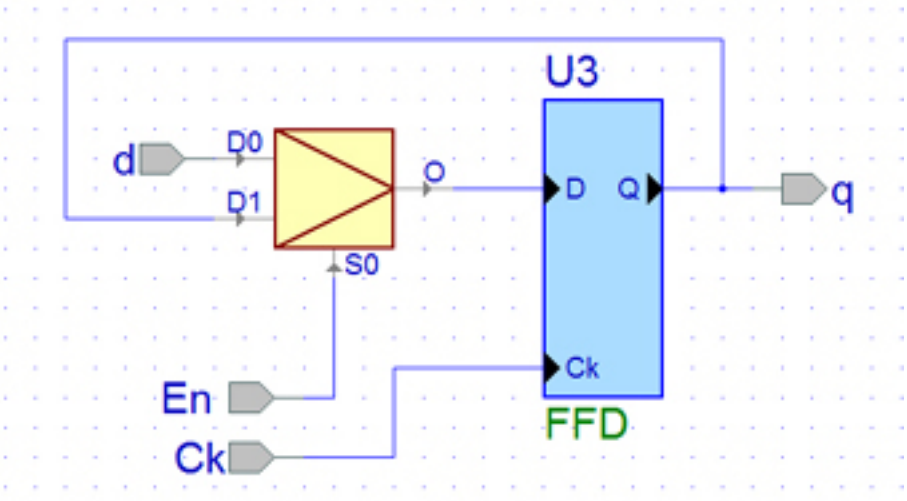


Figura 2.14: Flip flop D con data gating

Nella simulazione, il valore di q è mantenuto finchè un nuovo valore non gli è assegnato. In tal caso, l'assegnamento è bypassato finchè il segnale di enable è disattivato. Ciò è equivalente a riassegnare, al nuovo ingresso, il vecchio valore dell'uscita.

Reset

Ci sono due modalità per modellare un registro dotato di reset; un modo asincrono ed uno sincrono. E'importante fare una distinzione tra le due forme ed utilizzare quella più corretta alla circostanza.
Un reset asincrono scavalca il clock ed agisce immediatamente, modificando il valore del registro e quindi la sua uscita. Al contrario, i reset sincroni hanno effetto solo al fronte positivo del clock e devono quindi essere mantenuti fino al campionamento, per essere rilevati. I reset sincroni possono essere pilotati da qualsiasi segnale di controllo all'interno del circuito, quindi tutti quei dispositivi che possono essere resettati da segnali di controllo generati da un

circuito appartenente alla rete, dovrebbero essere dotati di reset sincrono.

Il modello VHDL prevede che, anzichè alimentare direttamente l'ingresso del registro con la sua uscita, sia fornito all'ingresso il valore di reset predefinito se il segnale di reset è attivo. L'equivalente circuitale è in fig.2.15, mentre il codice che lo rappresenta è il seguente:

```
entity FFD is
   port(
       d, ck, res : in bit;
       q : out bit
       );
end;

architecture behaviour of FFD is
begin
   process
     begin
       wait until ck = '1';
       if res = '1' then
          q <= '0';
       else
          q <= d;
       end if;
   end process;
end;
```

Un segnale di reset asincrono dovrebbe essere sempre pilotato da un input primario, nell'ambito di un sistema asincrono, per poi essere reso sincrono al clock (similmente a quanto accade nei calcolatori). I segnali di reset interni al circuito dovrebbero essere sempre sincroni.
Ci sono due buone ragioni affinchè componenti dotati di reset asincrono non vadano impegati in circuiti che genereranno segnali di reset sincroni: la prima ragione è l'introduzione di comportamento asincrono in un circuito, per sua natura, sincrono. Ciò renderebbe il circuito molto sensibile ai glitch che si

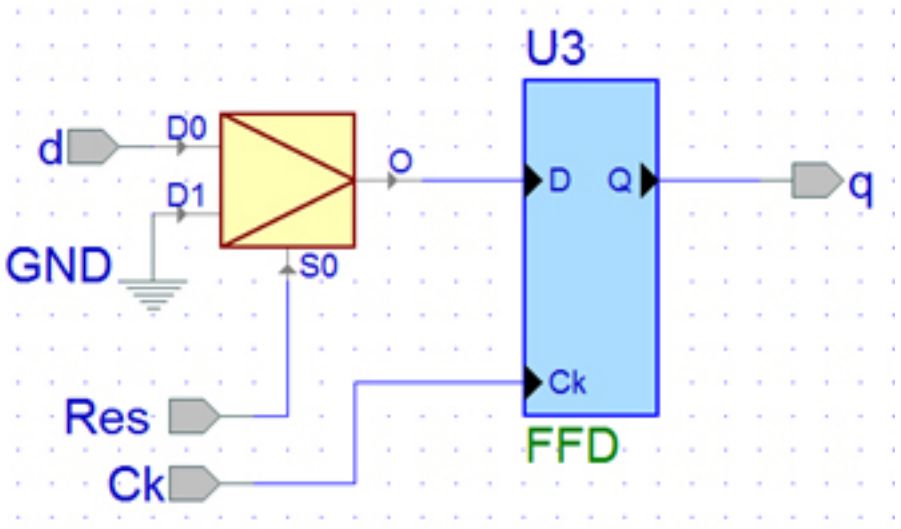


Figura 2.15: Flip flop con data gating e reset sincrono

verificherebbero nella logica di reset. La seconda motivazione è i *timing analysers* e i test automatici per la sintesi non possono pilotare correttamente reset asincroni sui componenti.

Un esempio di codice, in grado di modellare un registro con reset asincrono, è il seguente:

```
entity FFD is
   port(
       d, ck, res : in bit;
       q : out bit
       );
end;

architecture behaviour of FFD is
begin
   process (ck, res)
     begin
       if res = '1' then
          q <= '0';
       elsif ck'event and ck = '1' then
```

```
            q <= d;
        end if;
    end process;
end;
```

3.6 Driver

Si immagini di dover rispondere alla domanda: *"Cosa accadrebbe se all'interno di un processo viene fatto più di un assegnamento allo stesso segnale?"*; per capire quale sia l'effettivo valore del segnale, è importante capire il concetto di driver.

Un *driver* è creato per ogni segnale cui è assegnato un valore in un processo, esso è associato al segnale contenendo il suo valore corrente e tutti i suoi valori futuri, espressi come sequenza di una o più transizioni, dove ciascuna di esse identifica il valore che dovrà apparire nel tempo.

Si consideri il seguente assegnamento, con tre variazioni nel tempo, e il driver creato per quel segnale:

```
process
begin
   ...
   Reset <= 3 after 5 ns, 21 after 10 ns, 14 after 17 ns;
end process;

Reset  <-- | curr@now | 3@T+5ns | 21@T+10ns | 14@T+17ns |
```

Tutte le transizioni su un driver sono ordinate per valori di tempo crescenti, un driver contiene sempre almeno una transizione, che può essere il valore iniziale assegnato al segnale. Il valore di un segnale coincide con il valore del suo driver.

Nell'esempio precedente, quando il tempo di simulazione avanza a T + 5 ns, la prima transizione è cancellata dal driver, quindi il segnale assume il valore 3. Quando il tempo avanza ancora a T + 10 ns, la seconda transizione è

cancellata ed il segnale assume il valore 21. Infine, quando il tempo avanza a T + 17 ns, la terza transizione è raggiunta e cancellata: il segnale *Reset* assume il valore 14.

Se all'interno del processo fosse presente un'altra assegnazione sul segnale *Reset*, poichè un segnale in un processo ha un solo driver, le transizioni generate dalla seconda assegnazione modificano quelle già presenti sul driver.

In precedenza (nella sez. 3.1) si era fatto un accenno alle varie forme di ritardo nell'assegnazione ad un segnale, ora se ne darà una visione più approfondita nel loro comportamento sui driver dei segnali.

Effetti di un Transport Delay sui driver

Si consideri il seguente processo, con tre assegnazioni sullo stesso segnale, tutte modellate con un ritardo *transport*:

```
signal data : natural;
....
process
begin
   ...
   data <= transport 11 after 10 ns;
   data <= transport 20 after 22 ns;
   data <= transport 35 after 18 ns;
end process;
```

Si assuma che gli assegnamenti siano effettuati al tempo T, infatti essendo all'interno sono valutati sequenzialmente in un tempo nullo. Le transizioni sul driver del segnale *data* sono create come segue: quando la prima assegnazione è valutata, la transizione *11 @ T + 10 ns* è aggiunta al driver. Valutata la seconda assegnazione, la transizione *20 @ T + 22 ns* è appesa alla coda perchè il ritardo di questa transizione (22 ns) è più grande del ritardo della prima (la seconda avverrà comunque dopo la prima).

A questo punto, il driver del segnale *data* apparirà così:

```
data <-- | curr@now | 11@T+10ns | 20@T+22ns |
```

Infine, è eseguita la terza assegnazione, la nuova transizione *35 @ T + 18 ns* avviene prima della seconda, quindi la corrispondente *20 @ T + 22 ns* è cancellata, mentre la nuova è appesa nella coda. La ragione per questo comportamento è che il ritardo della nuova transizione (18 ns) è minore del ritardo dell'ultima transizione presente sul driver (22 ns). Questo effetto è causato perchè s'è usato il ritardo transport.

In generale, una nuova transizione cancellerà tutte le precedenti transizioni presenti in un driver che avvengano contemporaneamente o dopo il ritardo della nuova transizione. Ritornando sull'esempio, il driver del segnale *data* risulterà:

```
data <-- | curr@now | 11@T+10ns | 35@T+18ns |
```

In pratica, è possibile definire delle regole precise che governino le transizioni sul driver di un segnale in presenza di transport delay:

1. tutte le transizioni presenti sul driver che avverranno in corrispondenza o dopo il ritardo della nuova transizione, saranno cancellate.
2. tutte le nuove transizioni sono aggiunte alla fine della coda del driver.

Effetti di un Inertial Delay sui driver

Quando si utilizzano ritardi iniziali, sia il valore del segnale assegnato sia il ritardo influiscono sulla rimozione e l'aggiunta delle transizioni. Se il ritardo della prima nuova transizione è minore di una transizione esistente, quest'ultima è cancellata indipendentemente dal valore del segnale delle due transizioni. Si noti che ciò coincide con la regola 1 vista nel caso di ritardi transport. Viceversa, per le transizioni esistenti che cadono all'interno della finestra temporale della prima nuova transizione sono verificati i valori dei segnali: se i loro valori sono differenti, tali transizioni sono cancellate, viceversa sono mantenute.

L'esempio, espresso da seguente codice, dovrebbe rendere più chiaro il concetto:

```
process
begin
   data <= 1 after 5 ns, 21 after 9 ns, 6 after 10 ns,
           12 after 19 ns;
   data <= reject 4 ns inertial 6 after 12 ns, 20 after 19 ns;
wait;
end process;
```

Lo stato del driver del segnale *data* dopo la prima assegnazione risulta:

```
data <-- | curr@now | 1@5ns | 21@9ns | 6@10ns | 12@19ns |
```

L'esecuzione della seconda istruzione causa la cancellazione di tutte le transizioni sul driver dopo 12 ns (perchè avviene prima delle altre) e sono aggiunte le due nuove transizioni *6 @ 12 ns* e *20 @ 19 ns*. La transizione *21 @ 9 ns* è rimossa anche essa perchè il suo ritardo cade nella finestra compresa tra 12 ns e (12 ns - 4 ns), e il suo valore (21) è diverso dal valore della nuova transizione (6), mentre l'altra è mantenuta perchè il valore del segnale non cambia. Il driver risultante è:

```
data <-- | curr@now | 1@5ns | 6@10ns | 6@12ns | 20@19ns |
```

Concludendo, è possibile definire delle regole per l'assegnazione a segnali usando ritardi inerziali:

1. tutte le transizioni su un driver schedulate per avvenire in corrispondenza o dopo il ritardo della prima nuova transizione sono eliminate dal driver (come nel caso *transport*);
2. le nuove transizioni si aggiungono in coda al driver;
3. per tutte le vecchie transizioni presenti sul driver schedulate per un tempo incluso nella finestra tra il tempo della prima nuova transizione (supponiamo T) e T meno il limite di reject, sono eliminate quelle con valori differenti del segnale rispetto alla nuova transizione.

Un discorso molto simile si ha anche al di fuori dei processi, nell'ambito di un'esecuzione concorrenziale. In tal caso, può capitare di avere istruzioni di assegnazione concorrenti che tentino di assegnare ad uno stesso segnale valori diversi con tempistiche differenti. Il tal caso la risoluzione del conflitto e la scelta del valore effettivo da assegnare è delegata ad una *resolution function* definita appositamente dall'utente. Il suo uso è poco consigliato, per una descrizione più approfondita in merito, si rimanda quindi ad [1] in bibliografia.

4 FSM

La forma base di una *macchina a stati finiti* (*FSM*) è un circuito sequenziale in cui lo stato futuro e l'uscita del circuito stesso dipendono sia dallo stato presente che dagli ingressi. La più comune applicazione di una macchina a stati finiti è all'interno di un circuito di controllo.

Lo schema base di una FSM è mostrata in fig. 2.16.

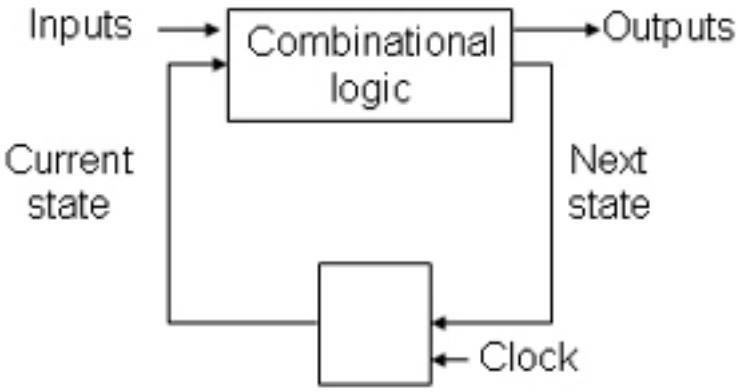


Figura 2.16: FSM

Una FSM generalmente è modellata in VHDL come l'unione di un blocco combinatorio e di un blocco di memoria. La caratteristica fondamentale del pattern realizzativo è che lo stato corrente e lo stato successivo sono rappresentati da un *tipo enumerativo*, con un valore per ciascun stato.

4.1 Ambiente di sviluppo FSM in Active HDL

L'IDE utilizzato nel lavoro di tesi (*Active-HDL 7.2* prodotto dalla *Aldec Inc.*) fornisce al progettista un ambiente di design grafico, chiamato *State Diagram Editor* (esempio in fig. 2.17). L'utente può utilizzare questo supporto per modellare graficamente l'automa a stati finiti nella sua completezza: sarà poi il compilatore a generare automaticamente il linguaggio VHDL corrispondente a quanto modellato. Inoltre, l'editor è collegato anche al simulatore per poterne permettere la simulazione e il debugging grafico del diagramma.

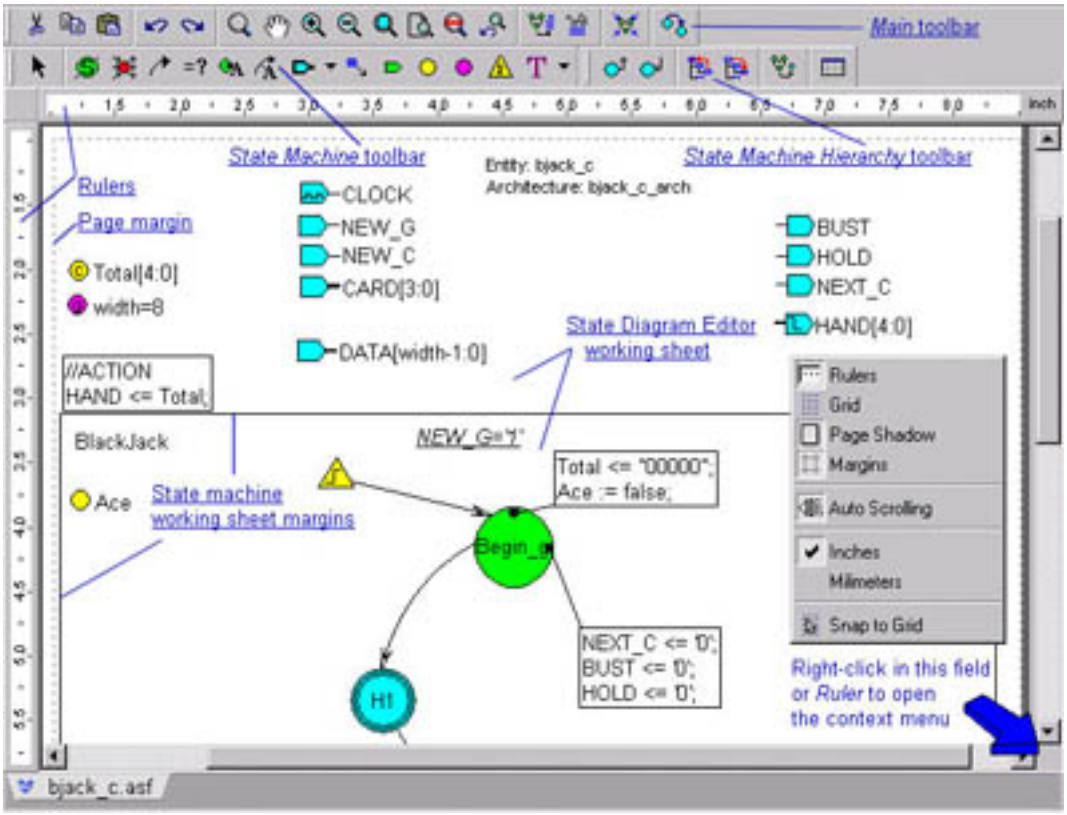


Figura 2.17: Esempio di State Diagram Editor

Segnali di ingresso e uscita

I segnali di ingresso ed uscita, si inseriscono attraverso il relativo bottone presente nella *State Machine toolbar*. Subito dopo l'inserimento nello schema, è possibile cliccare sull'icona del segnale per aprirne l'utilità di configurazione (fig. 2.18).

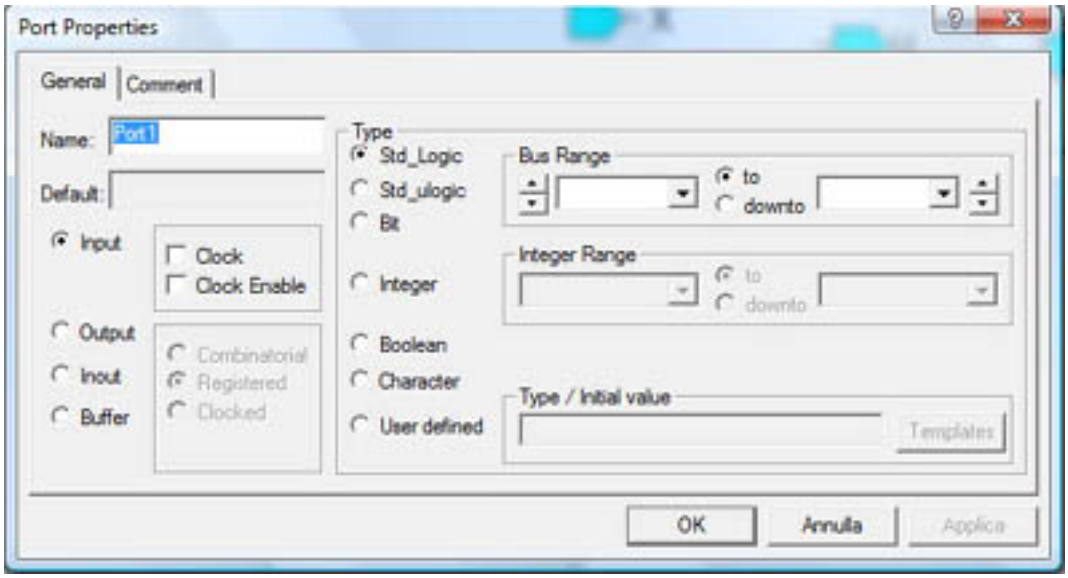


Figura 2.18: Finestra 'port properties'

Come si può vedere in figura, è possibile definire sia la tipologia di porta (ossia se *input*, *output*, *inout* o *buffer* - per le differerenze tra queste, si rimanda alla sez. 2.1), sia il tipo di dato ad essa associato (`std_logic`, `std_ulogic`, `bit`, `std_logic_vector`, `std_ulogic_vector`, `bit_vector`, `boolean`, `character`, `integer` (vincolato)). In particolare, se il segnale è un input alla rete, si può specificare se debba essere assunto come segnale di Clock o come *Clock Enable*. Viceversa, se è un segnale di output, si può definire la natura delle sue uscite.

Un'uscita (anche se di tipo *inout* o *buffer*) può essere modellata in tre modalità differenti:

- *uscita combinatoria*: una macchina con l'output combinatorio non include alcun elemento addizionale di memoria (come ad esempio un registro) nel percorso d'uscita. Un valore di una uscita combinatoria è aggiornato immediatamente ad ogni variazione dello stato interno (macchine di *Moore*) e degli ingressi (macchine di *Mealy*), ciò significa che l'uscita combinatoria può variare sincronamente in *Moore* e asincronamente in *Mealy*. Un esempio è mostrato nella figura 2.19.
- *uscita campionata*: una macchina con l'uscita campionata contiene un registro addizionale sull'uscita. La macchina e il registro finali agiscono

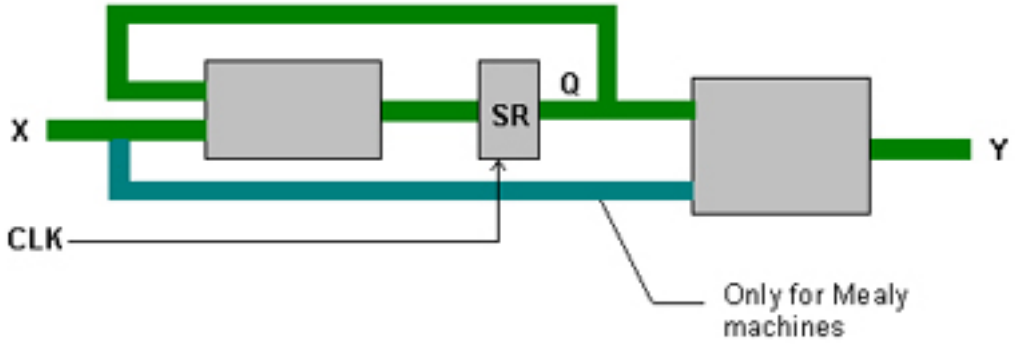


Figura 2.19: Uscita combinatoria

sincronamente sullo stesso fronte di clock. L'uscita di questa macchina è inevitabilmente ritardata di un clock, rispetto all'equivalente macchina con uscita combinatoria. Un esempio è mostrato nella figura 2.20.

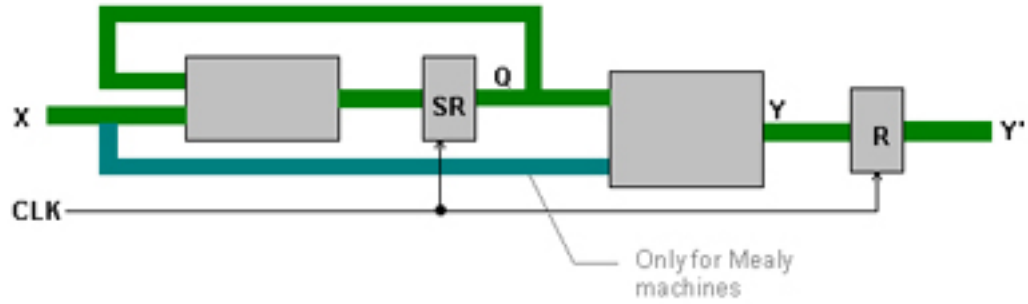


Figura 2.20: Uscita campionata

- *uscita in base al clock*: una macchina a stato finito con uscita legata al clock è molto simile alla macchina con uscita campionata. In questo caso, il valore del registro di stato corrisponde al valore disponibile sul registro d'uscita. Viceversa, nelle macchine con campionamento dell'uscita, il valore del registro di stato non segue questa regola ed è necessario un ciclo di clock ulteriore per ottenere sull'uscita il valore dello stato attuale. Un esempio è mostrato nella figura 2.21.

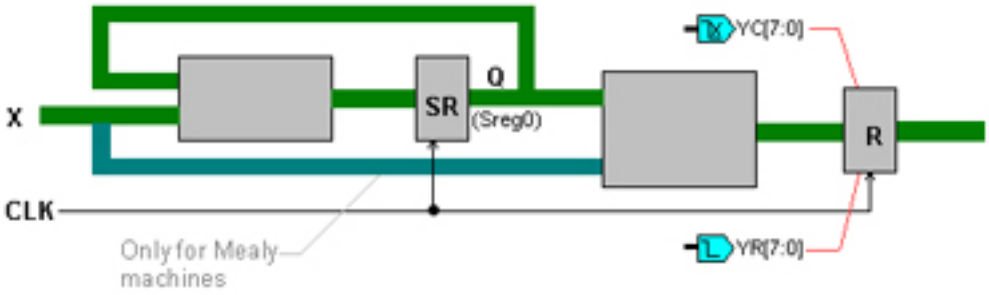


Figura 2.21: Uscita in base al clock

Stati

Dopo aver definito i collegamenti della rete con l'esterno, si possono inserire gli stati dell'automa. Uno *stato* è una condizione interna che determina le uscite della macchina. Ogni macchina sequenziale ha almeno due stati (altrimenti sarebbe combinatoria), legati tra loro da transizioni.

L'oggetto stato nel visualizzatore grafico è rappresentato (per default) da un cerchio verde. Esso è identificato dal suo nome univoco (all'inetrno della rete sequenziale cui appartiene). Per default, l'editor assegna nomi generici, quali S1, S2, S3, ecc., ma è possibile modificarli inserendo una stringa testuale più opportuna. L'oggetto 'stato' può essere inserito nel diagramma utilizzando il comando presente nel menu `FSM -> State` o cliccando sull'apposito pulsante nella toolbar.

E' possibile definire due proprietà aggiuntive per uno specifico stato:

- *default state*: è possibile dichiarare uno stato perchè sia uno stato di *default*. In questa condizione, la macchina transiterà per questo stato se tutte le condizioni assegante alle transizioni uscenti dallo stato corrente sono risultate tutte *false*. Se lo stato di default non fosse presente, evidentemente la macchina si bloccherebbe, in questo senso l'uso di detto stato costiuisce una forma di debug.
- *trap state*: è possibile che uno stato sia dichiarato come *trap*, la macchina transiterà in questo al successivo fronte attivo del clock se, per qualche ragione, il suo stato corrente non corriponde ad alcuno sta-

to definito sul diagramma. Ciò protegge la macchina da un'eventuale perdita di passo, in cui le variabili che codificano lo stato interno - probabilmente a causa di un disturbo - hanno assunto una configurazione illecita o non prevista.

Si rimanda al manuale utente dell'IDE per ulteriori utilizzi dell'oggetto stato, che esulano dal lavoro di tesi.

Al suo interno, lo stato consente di definire le azioni che devono essere intraprese (modellando una macchina di Moore). Una azione è un set di istruzioni HDL che assegnano nuovi valori alle porte, ai segnali interni e alle variabili. Per aggiungere un'azione al diagramma, si deve scegliere l'opzione più appropriata attraverso il menu `FSM -> Action`. La sintassi definita nelle azioni deve essere conforme a quella definita nel linguaggio VHDL.

Esistono anche altre due azioni direttamente legate ad uno stato:

- *entry actions*: un'entry action è una azione effettuata in occasione di una transizione in entrata nello stato.
- *exit actions*: un'exit action è una azione effettuata in occasione di una transizione in uscita dallo stato.

Transizioni

Le *transizioni* sono azioni che permettono l'avanzamento del flusso tra stati differenti. L'avanzamento dello stato, quindi lo scattare di una transizione, dipende dal clock e/o dal soddisfacimento di una specifica condizione sulla stessa. E' possibile che ad una transizione sia associato un cambiamento delle uscite (in accordo al modello di *Mealy*).

Come nel caso delle azioni interne allo stato, anche la scrittura delle condizioni e degli assegnamenti presenti nelle transizioni deve rispettare la sintassi VHDL. Questo è motivato dal fatto che il compilatore, nella generazione del codice, copierà nel listato generato le condizioni e le assegnazioni associate alle transizioni stesse. Cliccando su una transizione, è possibile visualizzare una finestra con le sue proprietà, in modo da poterle settare opportunamente.

Reset

La transizione di *reset* è uno strumento fondamentale per garantire il funzionamento di una macchina sincrona, perchè ne garantisce la corretta inizializzazione. L'editor grafico, ed il linguaggio VHDL stesso, non permettono di evidenziare uno stato come *stato iniziale*, c'è quindi bisogno di garantire la corretta inizializzazione in altre forme: una di questa è appunto quella di utilizzare un reset.

Il reset, come ogni altra transizione, si verifica se la condizione è valutata *true*; la transizione può essere poi asincrona o sincrona: nel primo caso, il passaggio al nuovo stato avviene immediatamente, viceversa avviene al primo fronte attivo del clock, ossia al ciclo successivo. Qualsiasi condizione *booleana* può essere utilizzata come condizione di reset, nell'immagine seguente (fig. 2.22) la macchina subisce un reset quando il segnale RST assume il valore '1'.

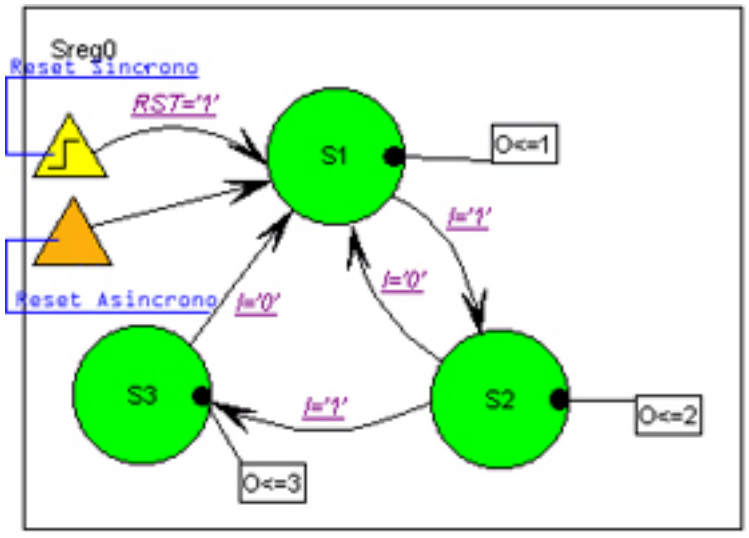


Figura 2.22: Esempio di modellazione di reset sincrono e asincrono

Un altro modo per impostare lo stato iniziale è quello di variare l'ordine degli stati; il segnale che rappresenta lo stato, sia l'attuale sia il futuro, è associato ad un tipo enumerativo (vedi pag. 87) contenente tutti gli stati dell'automa. Essendo un tipo enumerativo, il primo valore nell'elenco sarà quello assegnato di default all'avvio della simulazione. E' possibile modificare l'ordine degli stati, attraverso il menu `FSM -> View/Sort Objects -> States`.

4.2 Generazione automatica del codice

La FSM modellata graficamente con l'editor grafico, dopo aver settato tutte le opzioni necessarie alle funzioni che deve compiere, può essere convertita in VHDL. La conversione produce un'unità composta da una entity e da un'architecture. L'entità contiene la lista dei segnali in ingresso e in uscita alla FSM, mentre l'architettura contiene la traduzioni in assegnazioni e processi della macchina, secondo lo stile *comportamentale* (si veda a proposito la sezione 2.2).

Solitamente in letteratura, i pattern più utilizzati utilizzano due variabili distinte per rappresentare lo stato (stato presente e stato futuro) e due processi - operanti in modo concorrenziale - a gestirle: un processo (sincronizzato con il fronte attivo del clock) si occupa di aggiornare la variabile stato futuro in base allo stato attuale ed ai valori in uscita, inoltre in caso di una macchina di *Moore* si provvede anche ad assegnare l'uscita corrispondente. L'altro processo, sincronamente al clock, ha il solo compito di assegnare il valore futuro allo stato attuale. Chiaramente, essendo i due processi eseguiti parallelamente (per il modello di funzionamento proprio del linguaggio), lo stato futuro sarà assegnato al clock successivo rispetto quando è stato calcolato.
In una macchina di *Mealy* è presente un ulteriore blocco di istruzioni, che agiscono concorrentemente ai processi, che si occupano di aggiornare immediatamente i valori delle uscite in rapporto agli ingressi ed allo stato interno.

In realtà, l'IDE utilizzato utilizza un pattern lievemente differente, ma equivalente. Il compilatore genera un solo processo, generalmente contraddistinto dalla *label* `Sreg0_machine`: in esso, mediante delle istruzioni `CASE`, in relazione allo stato attuale e agli ingressi sono calcolati gli stati futuri. Per memorizzare lo stato viene utilizzata una sola variabile, a cui viene assegnato via via un nuovo valore all'interno del processo, chiaramente questa assumerà il valore dello stato futuro solo alla successiva riesecuzione del processo, cioè al clock successivo.

Si rimanda il lettore alle prove effettuate (cap. 6) ed alle conclusioni (cap. 7), in cui è evidenziata la generazione del codice e ne è valutata la bontà.

5 Design grafico

Parlando di *stili architetturali* (nella sez. 2.2) si era citato lo stile di modellazione strutturale: questo prevede l'istanziazione, all'interno di un'architettura, di componenti primitivi per utilizzarli nella formazione di una rete di livello superiore. A sua volta l'entità così definita potrà essere utilizzata essa stessa come componente, in processo a cascata.

Questo modo di procedere è particolarmente intuitivo e permette di limitare la *descrizione comportamentale* ai soli componenti di più basso livello, rendendo più semplice e immediatamente comprensibile la progettazione del componente. In particolare, questa modalità è ben rappresentabile graficamente. L'IDE utilizzato nel lavoro di tesi, fornisce al progettista un pratico ambiente di modellazione grafica, chiamato *Block Diagram Editor*.

5.1 Ambiente di modellazione grafica

Aggiungendo un documento del tipo *Block Diagram* al progetto, è possibile caricare l'ambiente di modellazione grafico. Questo si presenta come un ampio spazio libero, contornato da barre di strumenti. Un esempio è mostrato in figura 2.23.
L'area di lavoro rappresenta l'interno del componente che si sta modellando; al suo interno andranno posizionati i terminali di connessione con l'esterno, i componenti di livello inferiore, eventuali processi e tutto l'insieme di connessioni e bus.

Terminali e connessioni

I morsetti (o meglio i *terminali*, seguendo la nomenclatura del programma) costituiscono la via per collegare l'entità ad altri componenti ed al mondo esterno; essi possono essere singoli o raggruppati in bus. Un terminale bus, consente di collegare ad esso un bus, ossia un raggruppamento di più segnali distinti, ma con semantica comune.

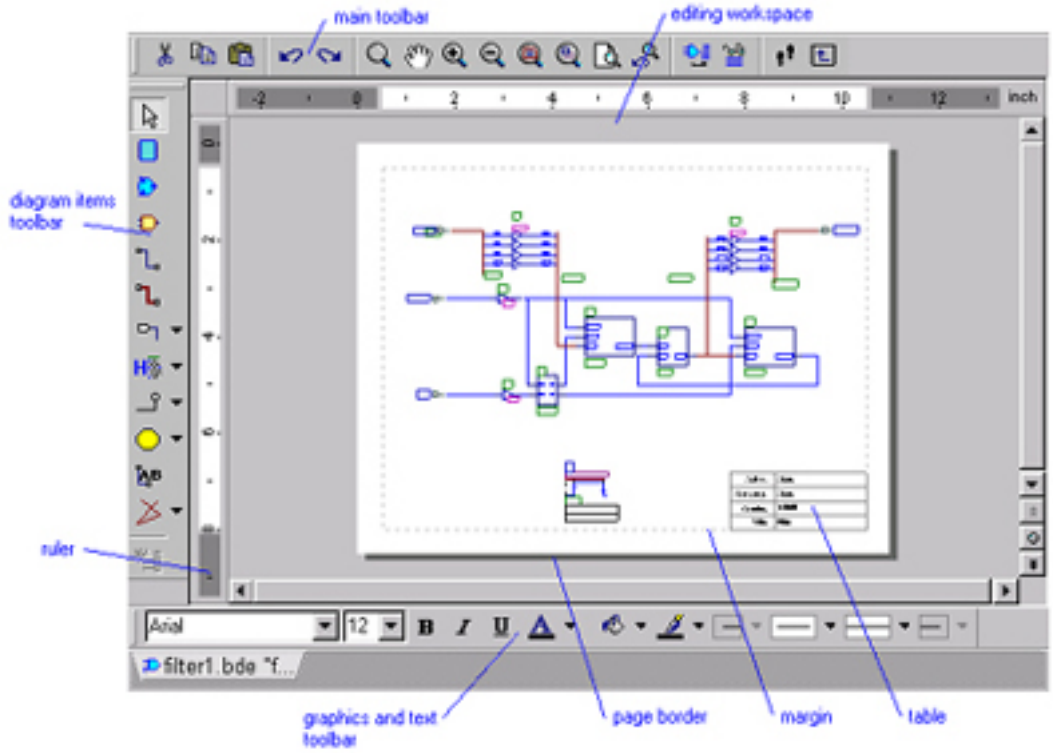


Figura 2.23: Block Diagram Editor

Similmente a quanto visto in precedenza, i segnali hanno sempre una direzione, in particolare esse sono quattro:

- *in*: input port. Una variabile o un segnale può leggere un valore da questo tipo di porta, ma non è possibile assegnare ad essa un valore.
- *out*: output port. E' consentito assegnare uno stato logico in uscita a questa porta, ma non è possibile leggere un valore da essa.
- *inout*: è una porta bidirezionale, è possibile assegnarle un valore o leggere un valore da essa.
- *buffer*: è una porta di output, ma con possibilità di lettura. Differisce da una porta inout perchè il suo valore può essere aggiornato solo da una sorgente, mentre una porta inout può essere aggiornata da zero o più sorgenti.

Si definisce *net* una connessione logica tra i simboli presenti nel diagramma. Ci sono due tipi di collegamenti: discreti e bus. In un diagramma, un collegamento *discreto* è rappresentato da una connessione singola (*Wire*). Un

collegamento a *bus* è una collezione di connessioni, ognuna delle quali ha un numero univoco che la contraddistingue dalle altre e che permette di identificarla singolarmente.

In VHDL, un collegamento è rappresentato da un segnale, i collegamenti discreti corrispondono a segnali scalari (ad esempio *std_logic*), mentre i bus corrispondono ad array mono-dimensionali (ad esempio *std_logic_vector*). Le connessioni (rispettivamente i bus) sono composte dall'unione di più segmenti in sequenza, essi assumono tutti lo stesso nome logico; in particolare, il nome assunto è quello del morsetto, o della porta (in caso di un componente), cui sono collegati.

L'unione di più connessioni o bus, anche graficamente disgiunti, ma con almeno un'intersezione in comune, definisce una rete (*net*). Ciascuna rete è identificata da un ID univoco, questo è automaticamente assegnato quando è tracciato una nuova connessione o bus. E' anche possibile (ma non obbligatorio) modificare il nome logico assegnato, per assegnarne uno più significativo (che ad esempio espliciti la funzione del collegamento stesso).

L'accorpamento e il disaccoppiamento di connessioni attraverso i bus è una operazione molto semplice e pratica. E' possibile suddividere ulteriormente il bus in sotto-bus ed estrapolarne i relativi segnali, i segmenti così ottenuti possono essere collegati agilmente ad altri dispositivi. Il bus, nella figura 2.24, è composto da tre segmenti: ogni segmento ha il proprio nome e il proprio range. I due sottoinsiemi `A[7:0]` e `A[15:8]` sono connessi a terminali a 8-bit, mentre l'intero bus `A[15:0]` è un bus a 16-bit.

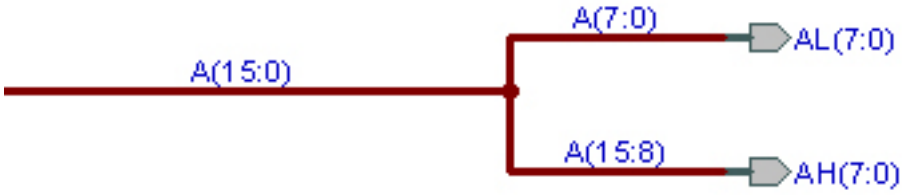


Figura 2.24: Bus slicing

E' possibile anche estrarre singolarmente le connessioni dal bus, ad esempio per connetterle alle singole porte di un componente. Nella nomenclatura del software utilizzato, tale concetto è espresso col nome di *bus tap* e rappresenta graficamente la giunzione tra una connessione e un bus, quest'ultimo diviene un membro del bus cui è collegato. Ciò implica che sia il bus sia la connessione debbano avere un nome, questo deve essere correttamente indicizzato: ad esempio, se il nome del bus fosse `DXBUS[7:0]`, i nomi validi per le connessioni ad esso collegati sarebbero `DXBUS(0)`, `DXBUS(1)`,..., `DXBUS(7)` (si veda anche la fig. 2.25).

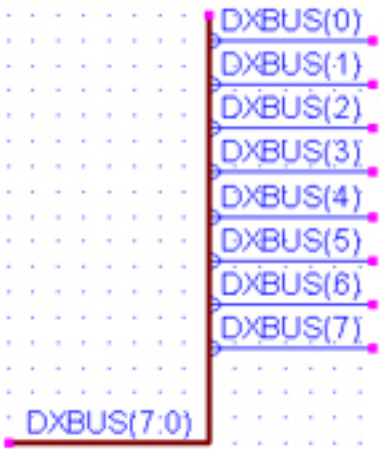


Figura 2.25: Bus taping

Fub e Symbol

Un *fub* è una rappresentazione grafica di un blocco logico creato ed editato direttamente nell'editor grafico. La logica interna al blocco si interfaccia con l'esterno attraverso le porte, rappresentate nel blocco funzionale mediante *pin*.

Ci sono varie proprietà che distinguono i fub dai simboli veri e propri:

- è possibile modificare le dimensioni grafiche di un fub, così come è semplice aggiungere o rimuovere pin direttamente dal diagramma.
- i pin sono aggiunti automaticamente quando si connette una connessione o un bus ad un fub. La label iniziata associata al pin è la stessa del nome della connessione stessa.

- i pin sono automaticamente rimossi quando si disconnette una connessione o un bus da un fub. Questa regola non si applica se il nome del pin è diverso dal nome della connessione stessa.

Inoltre è importante specificare che i fub esistono solamente in una sola istanza, quindi non è possibile progettare un componente che al suo interno contenga due istanze di uno stesso blocco funzionale. Ciò sarebbe semanticamente scorretto: un blocco funzionale ha il senso di racchiudere in esso delle istruzioni destinate a fare una specifica operazione, spesso dipendente dal contesto; se quindi si volesse creare un componente (quindi del codice) riutilizzabile o che si adatti a situazioni differenti si dovrebbe ricorrrere ad un simbolo. Per questa ragione, i fub non compaiono nella finestra *Symbols Toolbox* contenente la lista dei simboli di default e creati dall'utente per il progetto corrente.

I fub (rappresentazione in fig. 2.26) possono essere creati solo quando si impiega la metodologia di design *top-down*, in altre parole, il simbolo fub è creato prima di definire il suo contenuto. Una volta disegnato, facendo doppio click sulla sua forma è possibile scegliere come implementarlo (se con del codice VHDL, con un altro Block Diagram o addirittura con una macchina a stati).

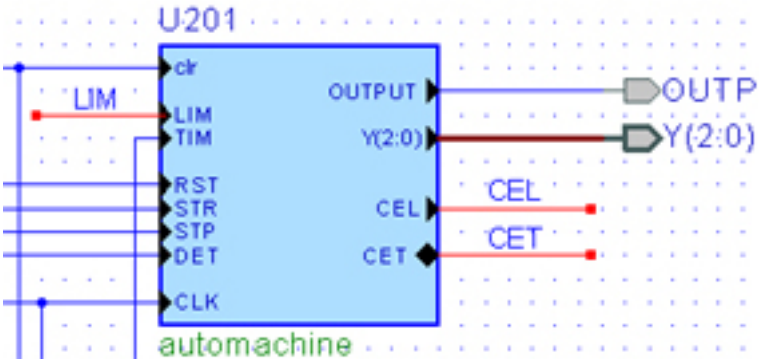


Figura 2.26: Fub

Un *symbol* è una rappresentazione grafica di un blocco logico. La sua complessità può variare da un gate elementare fino ad arrivare ad un blocco funzionale complesso. In particolare, l'intero design prodotto può essere modellato da un simbolo, richiamato poi nel testbench. La logica contenuta nel

simbolo si interfaccia con l'ambiente esterno attraverso le porte, rappresentate per mezzo dei pin.

Una caratteristica particolarmente significativa dei simboli è data dalla possibilità di definire configurazioni *generiche*, ovvero sia realizzare il componente in modo che non sia limitato, ad esempio, dal numero di segnali in ingresso e uscita o dal valore interno del suo stato, bensì che sia provvisto di parametri configurabili e adattabili dall'esterno. Per rendere possibile ciò, esistono delle istruzioni dedicate, come mostrato nell'esempio seguente:

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

entity SReg is
    generic(
        n : integer
        );
    port(
        D : in std_logic_vector(n - 1 downto 0);
        R : in std_logic;
        E : in std_logic;
        Ck : in std_logic;
        Q : inout std_logic_vector(n - 1 downto 0)
        );
end SReg;

architecture behaviour of SReg is
begin
    process (Ck)
    begin
      if Ck'event and Ck = '1' then  --CLK rising edge
         if R = '1' then    --synchronous RESET active High
             Q <= (others => '0');
         elsif E = '1' then
             Q <= D;
```

```
        end if;
      end if;
    end process;
end behaviour;
```

Il codice descrive un registro generico, dove n è il parametro, che si riflette sul numero di terminali disponibili. Si noti che anche la modellazione interna è stata impostata in modo da non inserire vincoli, ad esempio il reset è realizzando utilizzando la *keyword* `others` consentendo di assegnare un numero di zeri pari alla dimensione attuale del segnale stesso.
L'istanziazione avviene definendo i segnali, in modo classico, ed il valore del parametro:

```
U : SReg
    generic map(3)
    port map(
        Ck => Ck,
        D => CtoR,
        E => Ud,
        Q => BUS1,
        R => Rd
        );
```

Dove *3* è il valore del parametro, espresso in notazione *posizionale*, e i vari segnali collegati ai terminali di I/O sono array di 3-bit. Graficamente, il simbolo corrispondente è quello in fig. 2.27

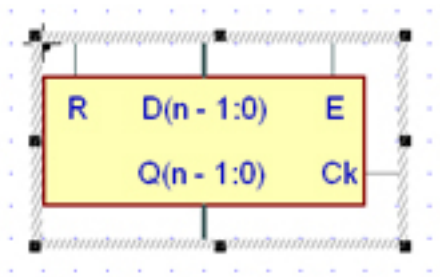


Figura 2.27: Registro generico

Il file sorgente che descrive la logica rappresentata dal simbolo è detta implementazione del simbolo e ne descrive il suo contenuto; similmente al fub,

l'implementazione può essere fatta in VHDL, con un altro diagramma a blocchi o con una macchina a stati finiti).

Ciascun simbolo memorizzato nella libreria associata al progetto è identificato univocamente con il suo nome. Quando si inserisce un simbolo nel diagramma, è assegnato automaticamente un nome univoco all'istanza, per identificarlo all'interno del diagramma. Poichè un simbolo rappresenta del codice - e più in generale un comportamento - riutilizzabile più volte nella stessa architettura, è possibile istanziarlo più volte (al contrario di un fub).

Processi

Una delle caratteristiche della programmazione grafica è la possibilità di inserire nello schema un oggetto che modelli un *processo* (si veda anche la sezione 3.1), si tratta dei cosiddetti *Graphical Process text blocks*. La procedura di inserimento di un processo è simile all'inserimento di un fub e il suo contenuto può essere editato direttamente cliccando sullo stesso.

Il blocco processuale così definito può essere quindi collegato agli altri componenti dello schema attraverso connessioni singole o bus; chiaramente in fase di simulazione, il processo sarà considerato alla stregua di una *macroistruzione* eseguita concorrenzialmente alle altre (poichè, si ricorda, che i vari componenti sono anche esse considerate delle macro istruzioni concorrenti, come per altro descritto nella sez. 3.3).

Immediatamente dopo aver completato le connessioni tra blocco processuale e altri elementi, il nome dei segnali così connessi è inserito nella *sensitivity list* visibile nel codice del processo stesso. A tal proposito, si veda la fig. 2.28.

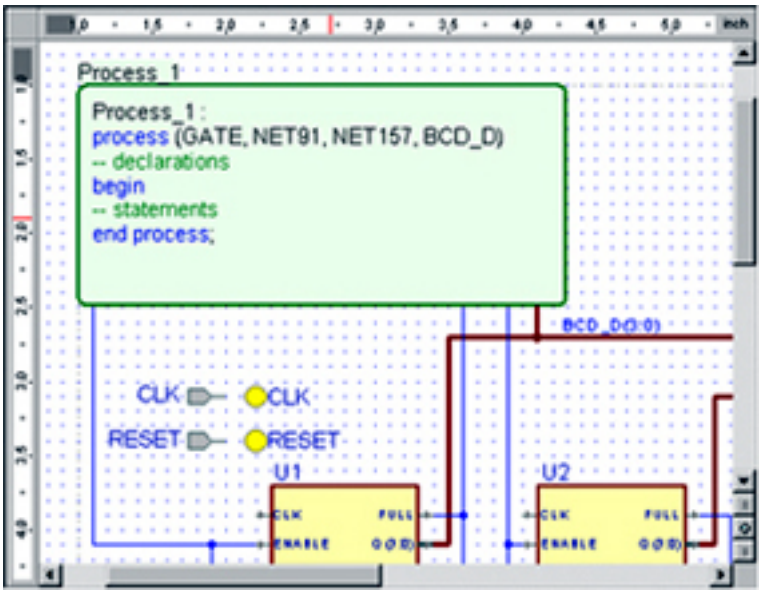


Figura 2.28: Blocco processuale e connessioni agli altri componenti

5.2 Generazione automatica del codice

Una volta completato il disegno dello schema, è possibile generare automaticamente il codice ad esso associato. Dapprima l'analizzatore controlla il rispetto dei vincoli sintattici e semantici (segnalando gli eventuali errori: uno dei più comuni è leggere un segnale uscente da un pin etichettato *out* anzichè *inout*) quindi genera il codice corrispondente.

L'architettura così creata rispecchia lo stile di modellazione *strutturale*, quindi dapprima saranno riportate le dichiarazioni dei componenti, quindi gli stessi sono istanziati e collegati tra loro secondo lo schema, con i relativi segnali. Ovviamente sono inseriti anche i processi, modellati nei blocchi processuali.
Una volta compilata l'unità derivante dalla coppia entità - relazione può essere utilizzata essa stessa come componente.

Anche quest'editor è collegato al simulatore per permettere la simulazione e il debugging grafico del diagramma. Se si fa procedere la simulazione per istanti precisi (ad esempio con il comando *Run Until* o con il comando *Run For*) è possibile visualizzare lo stato di ogni segnale presente nel diagramma,

compreso il valore attuale di registri e contatori (esempio, riportato in fig. 2.29). Chiaramente, si tratta di un'opzione molto utile in fase di debugging.

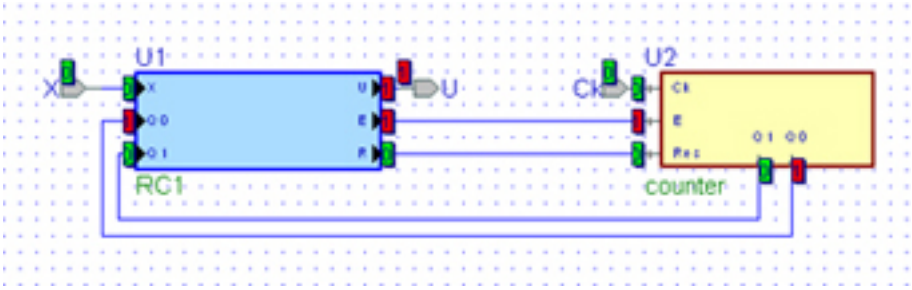


Figura 2.29: Probing dei segnali durante la simulazione, utilizzando il Block Diagram Editor

Si rimanda il lettore alle prove effettuate (cap. 6) ed alle conclusioni (cap. 7), in cui è evidenziata la generazione del codice e ne è valutata la bontà.

Capitolo 3

Tipi di dato

L'uso dei tipi è fondamentale per comprendere il VHDL, in quanto è un linguaggio fortemente tipato. Ciò significa che ogni tipo di flusso di dati (input, output, segnali interni, ecc.) ha un tipo associato e non ci può mai essere ambiguità squest'ultimo.

1 Standard types

Un cospicuo numero di tipi di dato è predefinito nel linguaggio. La tabella seguente riporta le definizioni di questi tipi predefiniti, la classe di tipologia a cui essi appartengono e se sono supportati dai sintetizzatori.

Tipo	Classe	Sintetizzabile
boolean	Enumeration type	Si
bit	Enumeration type	Si
character	Enumeration type	Si
severity_level	Enumeration type	No
integer	Integer type	Si
natural	Subtype of Integer	Si
positive	Subtype of Integer	Si
real	Floating - point type	No
time	Physical type	No
string	Array of character	Si
bit_vector	Array of bit	Si

Questi si trovano nel package, detto *standard*, che deve essere presente in ogni sistema VHDL. In aggiunta ai tipi standard, ne sono stati aggiunti altri per supportare la modellazione di gate logici e sono anche utilizzati nella sintesi. Questi tipi sono comunemente raggruppati sotto il nome *std_logic* e sono memorizzati in un package detto *std_logic_1164*, così chiamato perchè standardizzato dalla IEEE con il numero 1164.

Nelle seguenti sezioni saranno esaminati i tipi più significativi del tipo std_logic.

1.1 Type Bit

Il tipo *bit* assume solamente due valori, rappresentati dai caratteri '0' e '1'. In altre parole, la definizione di tipo è:

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

Si noti l'uso degli apici, questi sono fondamentali e non vanno omessi, questo perchè i valori sono dei caratteri e non dei numeri. In VHDL i caratteri sono distinti dai numeri, in quanto racchiusi da un apice.

Gli operatori che si applicano al tipo bit sono:

- *booleani*: `not, and, or, nand, nor, xor`;
- *di confronto*: `=, /=, <, <=, >, >=`.

Il tipo bit ha un ricco set di operatori logici, i quali forniscono sempre un risultato di tipo bit. Ciò significa che gli operatori logici possono essere composti a formare espressioni complesse e tutti i risultati intermedi saranno di tipo bit.

Il tipo bit ha anche un ricco set di operatori di confronto, che forniscono tutti un risultato booleano. E' possibile confrontare due segnali di tipo bit (ad es. `a = b` per vedere se hanno lo stesso valore ed il risultato sarà di tipo boolean - ossia *true* o *false*).

1.2 Boolean

Il tipo *boolean* è il tipo predefinito per i confronti in VHDL. E' raramente utilizzato direttamente come tipo logico, poichè il tipo *bit* ricopre completamente tale ruolo. In altre parole è improbabile trovare un'assegnazione di un valore ad un segnale espressa tramite un valore booleano (ad es. anzichè `a = false`, si utilizzerà `a = '0'`). E' comunque possibile un booleano per settare il valore di un segnale, se lo si desidera.
Il tipo boolean è predefinito ed ha la seguente definizione:

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

Ciò significa che il tipo ha solamente due possibili valori, *false* e *true*.
Gli operatori che si applicano al tipo boolean sono:

- *booleani*: `not, and, or, nand, nor, xor`;
- *di confronto*: `=, /=, <, <=, >, >=`.

Il tipo boolean è solitamente utilizzato indirettamente quando è eseguito un confronto tra altri due segnali di tipo differente. Come visto già nella sezione precedente, una comparazione della forma `a = b` produce come risultato un valore booleano, indipendentemente dal tipo dei segnali *a* e *b*.

1.3 Integer

E' molto importante quando ci si riferisce agli interi distinguere tra il tipo di dato chiamato *integer* (definito nello standard) e gli altri tipi di dato interi (definiti dall'utente). D'ora in avanti, quando si citerà il tipo integer, si farà riferimento al tipo predefinito *integer* (proprio del linguaggio VHDL standard).

Il tipo integer

Il tipo *integer* è il tipo di dato predefinito che rappresenta i valori numerici interi. Il range di valori coperto dal tipo integer non è definito dallo standard VHDL, ma deve essere almeno compreso tra -2.147.483.647 e +2.147.483.647.

Questo infatti è il range per i numeri di 32-bit in complemento ad uno. La ragione per cui si è scelto tale range in complemento ad uno come standard è, presumibilmente, quella di consentire la massima flessibilità nella scelta della rappresentazione, in quanto è lievemente più piccolo del corrispondente range in complemento a due. In pratica, la maggior parte delle implementazioni utilizzano il range in complemento a due, che ha un lower value di -2.147.483.648.

La definizione del tipo *integer* è la seguente:

```
type integer is range -2147483648 to +2147483647;
```

Gli operatori che si applicano al tipo integer sono:

- *di confronto*: `=, /=, <, <=, >, >=`;
- *aritmetici*: `sign +, sign -, abs, +, -, *, /, rem, mod, **`.

Interi definiti dall'utente

In aggiunta al tipo integer predefinito, gli utenti possono definire altri tipi di interi. Ad esempio, se si realizzasse un sistema in cui tutte le operazioni fossero eseguite con un'aritmetica ad 8-bit, sarebbe possibile definire un tipo di interi ad 8-bit ad hoc per eseguire detti calcoli. In generale, è possibile definire tipi di interi solo con range minori di quelli previsti dal tipo integer predefinito dall'implementazione.

Un esempio di definizione di interi che definisca un nuovo tipo con un range di 8-bit in complemento a due è dato dal seguente codice:

```
type short is range -128 to 127;
```

Nelle espressioni che contengono interi, non è possibile mescolare differenti tipi di interi. Per esempio, non è possibile utilizzare contemporaneamente i tipi *integer* e *short* nella stessa espressione. In questo senso, il VHDL è un linguaggio fortemente vincolato ai tipi e tale chiusura consente di rilevare rapidamente e fin dalle prime fasi della progettazione, possibili errori nel casting di tipi di dato differenti. L'essere fortemente tipizzato promuove un uso attento dei tipi e richiede una chiara comprensione di quale tipo utilizzare

per adattarsi ad una certa informazione.

Le regole del VHDL insistono sul fatto che il risultato di un'operazione tra interi debba rientrare nel range previsto: in altre parole, se si decidesse di effettuare un'operazione che ritorni un valore *short*, l'operazione stessa deve produrre un risultato compreso tra -128 e 127. Viceversa, se il risultato dell'operazione è fuori range, si otterrà un'eccezione durante la simulazione. Se ciò si verificasse, la maggior parte dei simulatori assegnerebbe all'operazione un risultato indefinito e la simulazione verrebbe interrotta.

Si deve sempre tenere in considerazione che gli interi in VHDL non possono sforare il range e non vale il *wrap round*. In altre parole, sommare 2 a 127 non dà come risultato -127, bensì un'eccezione, a differenza di quanto avviene in linguaggi di più alto livello. Questa è una diretta conseguenza della forte tipizzazione del linguaggio stesso.

Quando l'utente definisce un nuovo tipo di intero, il VHDL assume automaticamente che possano essere applicati gli operatori del tipo *integer*.

Integer subtype

Un sottotipo è la restrizione sul range di un tipo di dato. Il tipo di dato su cui si basa un sottotipo è noto come *basetype*. Ad esempio, esistono due predefiniti sottotipi del tipo *integer* chiamati rispettivamente *natural* e *positive*. Le loro definizioni sono:

```
subtype natural is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;
```

Il valore `integer'high` rappresenta il limite superiore del tipo integer (si veda anche il par. 6 di questo capitolo), quindi almeno 2.147.483.647.
Un segnale dichiarato *natural* è di fatto un integer, ma con un range d'utilizzabilità ridotta.

Tutti gli operatori del tipo integer sono ereditati dai suoi sottotipi: ciò significa che se si esegue un'operazione su due valori natural, dapprima sarà

eseguita una somma di interi, quindi il simulatore verificherà che il risultato prodotto sia nel range ammissibile per il sottotipo natural. Questa verifica è effettuata però solo all'effettivo assegnamento (e non nel calcolo); ciò significa che, in un'espressione complessa, i valori intermedi potrebbero eccedere i range del sottotipo ma sono tollerati a patto che non si ecceda il range del tipo integer e che il risultato finale dell'operazione rientri nel range del sottotipo.

Un esempio aiuterà a capire meglio il concetto. Si crei dapprima il sottotipo di 4-bit *nat4* di natural (a sua volta sottotipo di integer):

```
subtype nat4 is natural range 0 to 15;
```

Quindi si prendano quattro segnali appartenenti al sottotipo nat4, tali che:

```
w <= x - y + z;
```

con x = 3, y = 4 e z = 5.

Non essendoci parentesi o operatori più prioritari, l'espressione è calcolata da destra a sinistra: dapprima è eseguita la sottrazione 3 - 4, il cui risultato è -1 (valore non ammissibile per *nat4*, ma lecito per *integer*), questo perchè l'espressione è calcolata come se gli operandi fossero degli integer, quindi a questo valore viene aggiunto 5 e il risultato finale è 4. Questo valore è quindi assegnato (al successivo delta cycle, si veda anche cap. 2, sez. 3.3) al segnale w verificando che il valore sia ammesso nel range di *nat4*.

Si noti che il successo di questa operazione deriva dall'uso di integer come *basetype* rispetto cui fare le operazioni intermedie: se si fosse ottenuto nat4 come sottotipo di un tipo di interi definito dall'utente - senza valori negativi -, il calcolo dell'espressione sarebbe fallito alla valutazione della sottrazione, perchè il risultato sarebbe stato fuori dal range ammissibile del basetype stesso.

2 Enumeration

Un *tipo enumerativo* è un tipo di dato composto da un set di valori letterali. L'esempio più ovvio, in grado di illustrare un tipo enumerativo è la variabile di stato di una macchina a stati finita (*FSM* - si veda anche cap. 2, sez. 4). I valori letterali sono nomi, quindi il tipo può essere pensato come un set di nomi.

Ad esempio, il codice seguente definisce un tipo enumerativo che modella il controller di un semaforo:

```
type state is (main_green, main_yellow, farm_green, farm_yellow);
```

Questo tipo è composto da quattro letterali, identificati dai nomi *main_green, main_yellow, farm_green* e *farm_yellow*. Qualsiasi nome, può essere usato come letterale ad eccezione delle keyword del linguaggio (ad esempio non è possibile chiamare *type* un letterale).

Il tipo enumerativo può essere anche definito da letterali composti da un singolo carattere; ci sono dei vantaggi nell'uso di caratteri, specialmente nell'uso di array di caratteri (vedi sez. 5), perchè il VHDL fornisce alcune scorciatoie per la definizione di array di caratteri. Per questa ragione, la maggior parte dei tipi logici sono definiti come caratteri. Ad esempio, una logica a quattro valori può essere definita come segue:

```
type mlv4 is ('X', '0', '1', 'Z');
```

Si noti che, come al solito in VHDL, i caratteri sono racchiusi tra apici. Non si deve fare confusione tra i singoli caratteri e lo specifico tipo di dato *character*: quest'ultimo contiene l'intero set di caratteri a 7-bit in codice ASCII ed è solitamente utilizzato per sintetizzare hardware in grado di manipolare testo ASCII.

I letterali presenti in un tipo enumerativo assumono una notazione posizionale: il primo letterale occupa la posizione 0, i successivi i numeri seguenti. Ad esempio, la posizione numerica del tipo *mvl4* definito precedentemente è la seguente:

```
main_green = 0
main_yellow = 1
farm_green = 2
farm_yellow = 3
```

Comunque, similmente a quanto avviene nei linguaggi più noti, i tipi enumerativi non sono di tipo intero e non è possibile riferirsi ai singoli letterali attraverso i loro valori posizionali. Inoltre, non è possibile eseguire operazioni aritmetiche sui letterali (tramite il valore posizionale loro associato). Infine, tale valore posizionale è predeterminato dalla definizione del linguaggio e non può essere ridefinito dall'utente, in altre parole non è possibile far corrispondere, ad esempio, al primo letterale il valore posizionale 1 anzichè 0.

I soli operatori predefiniti per il tipo enumerativo sono solo i sei operatori di confronto: `=, /=, <, <=, >, >=`. Tali operatori sono definiti nei termini dei valori posizionali dei singoli letterali. Ciò significa che il primo letterale è trattato come il valore più piccolo di tutti quelli del set, mentre l'ultimo è considerato quello più grande.

3 Multi-valued logic type

Un tipo di logica *multi-valued* (a più valori) è una logica che include i cosiddetti valori *metalogici*. Questi sono valori che non esistono fisicamente, ma che si rivelano molto utili nella simulazione. Un classico esempio è il valore associato all'*alta impedenza 'Z'*. Ovviamente, collegando un multimetro ad un morsetto non si misurerà mai un valore 'Z', ma tale valore è noto in letteratura per il suo nella modellazione di tristate bus. Anche i valori '0' e '1' sono delle astrazioni, che modellano i corrispondenti valori di tensione presenti fisicamente nel circuito reale.

Il tipo di dati standard *std_ulogic* utilizza una logica a 9 valori, è stato progettato principalmente per la simulazione a livello di gate e supporta la modellazione di alcune caratteristiche come i pull-up e i pull-down resistivi. Esiste anche un sottotipo, chiamato *std_logic* che ha lo stesso comportamento logico, ma può essere usato anche per modellare i segnali in tristate. In realtà,

tale sottotipo è usato universalmente per tutte le operazioni, non solo per quelle che coinvolgono valori tristate, quindi la combinazione del tipo e del sottotipo è generalmente riferita attraverso il nome del sottotipo, *std_logic*.

Std_logic non fa parte del linguaggio VHDL nativo, ma è un'estensione al linguaggio, standardizzata da IEEE; essa è presente nel package *std_logic_1164*, ed è utilizzabile nel codice richiamandola attraverso le istruzioni:

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

Il tipo *std_ulogic* ha la seguente definizione:

```
type std_ulogic is (
   'U', -- Uninitialized
   'X', -- Forcing Unknown
   '0', -- Forcing 0
   '1', -- Forcing 1
   'Z', -- High Impedance
   'W',-- Weak Unknown
   'L', -- Weak 0
   'H', -- Weak 1
   '-'  -- Don't care
);
```

L'uso di un tipo multi-valued è pieno di potenziali trabocchetti. La maggior parte di essi riguarda l'uso improprio di valori metalogici al posto dei valori reali, ad esempio l'assegnazione del valore 'L' o 'X' ad un segnale uscente, anzichè utilizzare un valore reale (come uno 0 o un 1).

Un'altra fonte di possibili problemi è data dall'uso dei *don't care*. Questo valore è stato introdotto per consentire all'utente di render flessibile la sintesi del circuito da progettare, ma può presentare problemi perchè non esiste il concetto di don't care in VHDL, che quindi è affidato al singolo compilatore. Ad esempio, se fosse prevista la gestione dei don't care, il seguente test di uguaglianza nel tipo *std_logic* darebbe sempre *true*: `match <= s = '-';`

Viceversa (come avviene nell'IDE utilizzato in questo elaborato), il linguaggio interpreterebbe ciò non come un test di matching con ogni valore, ma come un test di match esatto con il letterale '-': quindi il test darebbe esito *false*, a meno che il segnale s sia esso stesso in stato '-'.
Per questa ragione, si dovrebbe evitare l'utilizzo dei don't care: si mostrerà negli esempi come modellare istruzioni di assegnamento contenenti don't care.

4 Record

Un *record* è una collezione di elementi, ciascuno dei quali può essere di qualsiasi tipo o sottotipo anche diversi l'uno dall'altro. Un record è dichiarato come segue:

```
type complex is record
   real : integer;
   imag : integer;
end record;
```

Una volta che il tipo record è stato dichiarato, i segnali possono essere dichiarati di tale tipo, esattamente come se fossero di qualsiasi altro tipo esaminato precedentemente: `signal a, b, c : complex;`

Un segnale di tipo record è effettivamente una collezione di segnali, uno per ciascun elemento. Le regole per l'interpretazione di ciascun elemento sono le regole per quel tipo di elemento. Ad esempio, nel tipo *complex* definito sopra, gli elementi sono tipi *integer* a 32-bit.

Il solo operatore che può essere applicato su un tipo record (nel suo complesso) è quello di comparazione `=, /=`: il confronto avviene elemento per elemento, le uguaglianze tra gli argomenti sono quindi messe in *AND* tra loro per determinare l'uguaglianza complessiva. In altre parole, due segnali di tipo record sono uguali se i loro corrispondenti elementi sono uguali.

Per accedere agli elementi interni ad un record, si utilizza la *dot notation*: ad esempio, per assegnare il valore 0 alla parte reale del segnale *complex* si ha

la seguente istruzione: `a.real <= 0;`
Il tipo di *a.real* è un *integer*, poichè *integer* è il tipo dell'elemento *real* del segnale *complex*. Tutti gli operatori previsti dal tipo *integer* possono essere utilizzati sull'elemento *real*.

4.1 Aggregazione

Per assegnare i valori a tutti gli elementi di un record si utilizza una notazione chiamata *aggregazione*, ossia una collezione di valori. Un esempio di aggregazione è: `a <= (real => 0, imag => 0);`. Questa è la forma completa, in cui ciascun elemento è esplicitamente nominato ed associato ad un valore. Il simbolo `=>` è detto *finger* e associa il valore che segue con il nome dell'elemento nel record.

La modalità di funzionamento di un aggregazione prevede che l'espressione `(real => 0, imag => 0)` crei un valore temporaneo di tipo *complex*, e che questo valore temporaneo sia quindi assegnato al segnale come se si trattasse dell'intero record.

Una forma di aggregazione più semplice, utilizza la notazione posizionale. I valori degli elementi sono semplicemente scritti nello stesso ordine degli elementi definiti della definizione del record. Ad esempio: `a <= (0, 0);`

L'aggregazione può anche essere utilizzata per unire segnare al fine di assegnarli ad un record. Ad esempio, supponendo di avere due segnali r ed i di tipo *integer* e di doverli assegnare al segnale a di tipo *complex*. Ciò può essere fatto con un unico assegnamento, sfruttando l'aggregazione: `a <= (real => r, imag => i);`

L'operazione inversa può essere eseguita usando un'aggregazione come obiettivo di un'assegnazione. L'operazione non è però così semplice: l'analizzatore VHDL non può estrarre il tipo associato alla assegnazione target perchè normalmente si agisce all'opposto; il tipo del target è utilizzato per determinare il tipo del valore che gli sarà assegnato. Ad esempio, l'aggregazione `(real => r, imag => i)` è riconosciuta di tipo *complex* perchè è stata as-

segnata ad un segnale di tipo *complex*.
Per risolvere la questione, il VHDL utilizza il concetto di *type qualification*. Si tratta di una modalità per dire all'analizzatore VHDL di che tipo sia un'espressione, evitando così l'ambiguità. L'uso della type qualification è sempre richiesto esplicitamente quando si assegna qualcosa ad un aggregazione.

Si consideri il seguente esempio, i segnali (integer) r e i possono essere assegnati a partire dal segnale (complex) a in un solo assegnamento:

```
complex'(r, i) <= a;
```

Il type qualification non ha effetto sul circuito modellato, è solamente una guida per l'analizzatore al fine di risolvere l'ambiguità nel tipo dell'assegnazione. Nello specifico, i segnali i e r sono stati raggruppati assieme in un'aggregazione. Il type qualification (ossia il costrutto `complex'`) specifica il tipo di segnale che sarà passato, in questo caso di tipo *complex*. All'aggregazione è quindi assegnato il valore del segnale a; l'effetto complessivo è lo stesso del tener separate le due istruzioni:

```
r <= a.real;
i <= a.imag;
```

5 Array

Un *array* è una collezione di elementi, tutti dello stesso tipo. A differenza dei record, gli elementi di un array sono acceduti attraverso un indice, sia esso un intero sia un tipo enumerativo.
Un array può essere vincolato o non vincolato. Un tipo è *non vincolato* se la sua dimensione non è specificata: di fatto definisce una famiglia di sottotipi array, tutti con lo stesso tipo di elementi, ma con dimensione diversa. Viceversa, in un array *vincolato* sia il tipo di indice sia il suo range sono completamente definiti.

Nel VHDL, gli array vincolati sono implementati come sottotipi di un tipo array 'anonimo' non vincolato. Un *tipo anonimo* è un tipo senza nome, che non può essere quindi utilizzato direttamente dall'utente, ma che esite come

supporto per l'analizzatore sintattico. In questo senso, tutti i tipi di array in VHDL sono non vincolati. Ad ogni modo, poichè - come detto - non è possibile riferirsi direttamente al basetype anonimo, l'utente piò utilizzare solamente il sottotipo vincolato.

In pratica, un array non vincolato è utilizzato dal linguaggio per la definizione di altri tipi di dato. Ad esempio, il tipo predefinito *bit_vector* definisce un array di bit non vincolato:

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

Il simbolo `<>` è noto come *box* ed indica un range non vincolato. Il resto della dichiarazione dell'indice evidenzia che lo stesso varia in un range di valori appartenenti al sottotipo *natural*, in questo modo non sono permessi indici negativi.

Di fatto, l'utente utilizzerà poi, all'atto della dichiarazione di un segnale, un'istanza vincolata del sottotipo:

```
signal a : bit_vector (3 downto 0);
```

L'istruzione sovra-riportata definisce un segnale di quattro elementi, indicizzati dal range `3 downto 0`: tecnicamente, si definisce *descending range*; significa che il primo elemento dell'array è l'elemento numero 3, il secondo è l'elemento 2, e così via. L'uso di un range discendente è una prassi molto comune nella modellazione circuitale, ma ha una spiegazione ben precisa: il suo uso è associato ai bus di dati e alla rappresentazione di interi. In questi casi, il bit più significativo (*MSB*) è quello più a sinistra ed ha l'indice più alto, viceversa il bit meno significativo (*LSB*) è quello di indice zero. Gli array vincolati possono usare indifferentemente indici con range ascendenti o discendenti.

Un modo alternativo di ottenere lo stesso effetto è dichiarare un sottotipo vincolato di un array non vincolato, e quindi dichiarare il segnale come appartenente a detto sottotipo. Tale concetto si ottiene con la seguente coppia di istruzioni:

```
subtype bv4 is bit_vector (3 downto 0);
signal a : bv4;
```

Ogni segnale che è stato definito come *bit_vector* o di qualsiasi suo sottotipo ha come basetype *bit_vector*. I sottotipi possono avere qualsiasi lunghezza. Poichè il basetype non varia, segnali di differente lunghezza possono essere utilizzati contemporaneamente nella stessa espressione, senza causare un errore. Ad ogni modo, è richiesto che quando un array è assegnato ad un altro array, i due array siano della stessa misura.

Un esempio è la concatenazione di due bus a formarne uno unico, si supponga di avere due segnali *D0* e *D1* entrambi definiti come `bit_vector(3 downto 0)` e il segnale *D*, definito come `bit_vector(7 downto 0)`, rappresentante la concatenazione dei due bus (ad esempio della parte alta e della parte bassa di un dato ad 8-bit); è possibile scrivere - e simulare senza alcun problema - la seguente istruzione:

```
D <= D0 & D1;
```

nella quale al segnale D sono assegnati i valori dell'array D0 nel range `[7..4]` (rispettivamente: `3 => 7`, `2 => 6`, `1 => 5`, `0 => 4`) e i valori dell'array D1 nel range `[3..0]` (rispettivamente `3 => 3`, `2 => 2`, `1 => 1`, `0 => 0`).
Gli elementi di un array possono essere acceduti staticamente, ad esempio con un'istruzione tipo: `a(0) <= '1'`, o dinamicamente: l'array è acceduto dal valore attuale del suo indice, ad esempio con un segnale anzichè un intero. Ad esempio:

```
signal item : natural range 0 to 3;
a(item) <= '0';
```

I soli *operatori* disponibili nel tipo array sono gli operatori di comparazione (`=, /=, <, <=, >, >=`) e di concatenazione (`&`). L'operatore d'uguaglianza (`=`) verifica l'uguaglianza di ogni singolo elemento dei due array, il confronto avviene elemento per elemento, iniziando *sempre* da sinistra a destra. Gli operatori di ordinamento forniscono un ordinamento di array di lunghezze differenti.

La concatenazione permette di concatenare array differenti a formare un array più lungo, avviene prendendo gli elementi del primo operando nell'ordine da sinistra a destra e quindi appendendo a questi gli elementi del secondo operando nell'ordine da sinistra a destra. Ancora una volta, è la posizione degli elementi in ciascun array, non i loro indici, a determinare il risultato dell'operazione.

5.1 Aggregazioni

Similmente ai record, i valori degli elementi di un array possono essere assegnati in un'unica istruzione,utilizzando le aggregazioni:

```
a <= (3 => '1', 2 => '0', 1 => '0', 0 => '0');
```

Il tipo dell'aggregazione è dedotto dall'analizzatore in base al tipo del segnale cui è assegnato; nel caso in esame, il tipo dell'aggregato è *bit_vector*, in quanto il segnale *a* era definito come *bit_vector*. Il rispetto dei vincoli di range è verificato partendo dal set di indici presenti nell'aggregazione. Perchè l'assegnamento sia lecito è necessario che gli indici forniti abbiano valori ammissibili nel tipo previsto e che la lunghezza degli array sia la stessa.

Il funzionamento delle aggregazioni è abbastanza simile all'equivalente per i record (sez. 4.1). E' formato un valore intermedio (nell'esempio in questione, evidentemente di tipo *bit_vector* che è poi assegnato al segnale *a*. Ciò introduce un potenziale trabocchetto, perchè gli indici utilizzati nell'aggregazione non riflettono gli indici effettivamente usati nel segnale *a*, essi sono solamente gli indici interni all'aggregazione; in altre parole è solamente un caso che gli indici dell'array corrispondente al segnale *a* e dell'aggregazione siano gli stessi. Ad esempio, il seguente assegnamento - per quanto a prima vista potrebbe non sembrare - è esattamente equivalente al precedente:

```
a <= (0 => '1', 1=> '0', 2=> '0', 3 => '0');
```

La differenza è evidente nella fig. 3.1.

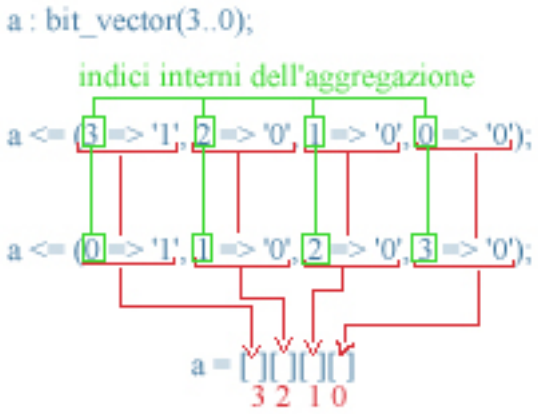


Figura 3.1: Indici di un array

Si otterrebbe il medesimo risultato con i seguenti quattro assegnamenti:

```
a(3) <= '1';
a(2) <= '0';
a(1) <= '0';
a(0) <= '0';
```

Un'altra conseguenza del meccanismo di assegnamento è che l'aggregazione potrebbe non avere lo stesso range di riferimento (non come ampiezza, che deve essere necessariamente uguale) del segnale target, ad esempio:

```
a <= (10 => '1', 11 => '0', 12 => '0', 13 => '0');
```

Per evitare incomprensioni, è fortemente raccomandato che le aggregazioni abbiano esattamente lo stesso range di valori del target e la stessa direzione.

Ovviamente è possibile anche l'operazione inversa, ovvero un'aggregazione può essere utilizzata come bersaglio di un assegnamento. Similmente a quanto avveniva nei record (sez. 4.1) c'è bisogno di utilizzare la *type qualification* per aiutare l'analizzatore a determinare il tipo del segnale oggetto di assegnazione:

```
bit_vector'(3 => elem3, 2 => elem2, 1 => elem1, 0 => elem0) <= a;
```

Ci sono anche alcune forme alternative di esprimere gli elementi di un'aggregazione, tutte equivalenti tra loro:

```
a <= (3 => '1', 2 | 1 | 0 => '0');
a <= (3 => '1', 2 downto 0 => '0');
a <= (3 => '1', others => '0');
```

La keyword *others*, se utilizzata, deve essere l'ultima scelta nell'aggregazione e deve definire tutti i rimanenti elementi dell'array da definire. Si noti che se si ricorre alla keyword *others*, allora il range dell'aggregazione non ha bisogno di fare match esatto con il range dell'array cui è assegnato.

In aggiunta alla notazione *nominale*, è possibile utilizzare anche con gli array la notazione *posizionale*; le assegnazioni viste precedentemente possono essere espresse come:

```
a <= ('1', '0', '0', '0');
```

Poichè l'assegnamento è effettuato dalla sinistra alla destra, in accordo alla posizione, il valore '1' sarà assegnato all'elemento più a sinistra di *a*, in questo caso all'elemento 3, e così via.

5.2 Array come stringhe

In realtà, per gli array di caratteri (ricordando che i valori dei segnali quali *bit* e *std_logic* sono espressi tramite caratteri) è possibile utilizzare un'ulteriore notazione più comoda e rapida, universalmente utilizzata per assegnare valori ad array di caratteri (quali *bit_vector* e *std_logic_vector*). Questa notazione è espressa nella seguente assegnazione:

```
a <= "1000";
```

Si noti l'uso delle doppie virgolette e non degli apici, in quanto qui si utilizzano stringhe di caratteri, differenziate dai caratteri stessi (e dei numeri) con l'uso delle virgolette.

Per agevolare l'utente c'è un'ulteriore notazione. E' possibile inserire i valori da dare ai segnali in formato numerico binario, ottale, decimale o esadecimale: è sufficiente anteporre al valore rispettivamente la lettera *B, O, D, X*.

6 Attributi

Un *attributo* è un modo per ottenere informazioni da un tipo di dato o dai valori associati ad un tipo di dato. Gli attributi sono utili, ad esempio, per trovare il primo o l'ultimo valore di un tipo o per determinare il valore posizionale di letterale enumerativo. In alcune circostanze, è una buona pratica riferirsi ad un valore, utilizzando un attributo, piuttosto che utilizzando direttamente il valore stesso.

Esistono diversi attributi che si applicano ai tipi interi, enumerativi ed array. Per mantenere chiaro il loro comportamento, se ne discuterà in due sezioni distinte.

6.1 Attributi (integer ed enumeration)

I seguenti attributi sono disponibili per i tipi scalari, ossia quelli che coinvolgono interi o enumerazioni:

```
type'left
type'right
type'high
type'low
type'pred(value)
type'succ(value)
type'leftof(value)
type'rightof(value)
type'pos(value)
type'val(value)
```

Per poter descrivere come questi operano, si farà riferimento ai seguenti tre tipi di dato:

```
type state is (main_green, main_yellow, farm_green, farm_yellow);
type short is range -128 to 127;
type backward is range 127 downto -128;
```

I valori più a sinistra e più a destra di un tipo possono essere determinati utilizzando, rispettivamente, gli attributi *left* e *right*:

```
state'left    -> main_green      state'right    -> farm_yellow
short'left    -> -128            short'right    -> 127
backward'left -> 127             backward'right -> -128
```

E' anche possibile trovare il valore più basso e quello più alto di un tipo, usando rispettivamente gli attributi *low* e *high*. Come si può vedere, c'è una sostanziale differenza rispetto agli attributi *left* e *right*, questa è evidenziata sul tipo *backward*: il valore all'estremo sinistro (destro) non necessariamente è il più piccolo (grande) del range.

```
state'low    -> main_green      state'high    -> farm_yellow
short'low    -> -128            short'high    -> 127
backward'low -> -128            backward'high -> 127
```

In altre parole, l'attributo *low* equivale all'attributo *left* per range ascendenti, mentre equivale all'attributo *right* per i range discendenti.

Gli attributi *pos* e *val* convertono un valore enumerativo in un valore numerico e viceversa. Questi attributi accettano un solo argomento, che è il valore da convertire, sia esso una costante o un segnale; in quest'ultimo caso si comportano come vere e proprie funzioni di conversione tra i tipi interi ed enumerativi.

```
state'pos(main_green) -> 0       state'val(3) -> farm_yellow
```

Infine c'è un set di quattro attributi che possono essere utilizzati per incrementare o decrementare un valore. Questi sono: `succ`, `prec`, `leftof` e `rightof`. L'attributo *succ* trova il successore del suo argomento, che è il successivo valore più alto del tipo di dato considerato - indipendentemente che il range sia crescente o decrescente. L'attributo *pred* trova il valore precedente, con semantica duale a quella dell'attributo *succ*. L'attributo *leftof* trova il successivo valore, più a sinistra, rispetto all'argomento fornito: questo sarà il successivo valore più basso per un range ascendente e il successivo valore più alto per un range discendente. Infine, l'attributo *rightof* trova il successivo valore più a destra, rispetto al suo argomento, il comportamento è il duale dell'attributo *leftof*.

Ad esempio:

```
state'succ(main_green) -> main_yellow
short'pred(0)          -> -1
short'leftof(0)        -> -1
backward'pred(0)       -> -1
backward'leftof(0)     -> -1
```

Si noti che il VHDL non consente il *wrap round* in caso di sforamento del range, quindi un'istruzione del tipo: `state'succ(farm_yellow)` causerebbe un'eccezione run-time, in quanto essendo un estremo del range, non ammette un valore successivo.

6.2 Attributi (integer ed enumeration)

Gli attributi degli array sono utilizzati per ottenere informazioni sulla loro dimensione, sul range di valori contenuti e sul loro indirizzamento interno. Questi attributi sono più frequentemente utilizzati dei loro corrispondenti per i tipi interi ed enumerativi, perchè è generalmente considerata buona pratica utilizzare gli attributi per riferirsi alla dimensione o al range di un array di segnali. Ad esempio, nel caso di istruzioni `for..loop`, gli attributi sono la sola modalità per specificare quali elementi sono visitati in ordine da sinistra a destra, indipendentemente dall'ordinamento del range.

Gli attributi esaminati in questa sezione sono:

```
signal'left
signal'right
signal'low
signal'high
signal'range
signal'reverse_range
signal'length
```

Si utilizzeranno inoltre i seguenti segnali, all'interno degli esempi:

```
signal up : bit_vector (0 to 3);
signal down : bit_vector (3 downto 0);
```

L'attributo *left* fornisce l'indice dell'elemento più a sinistra dell'array, mentre l'attributo *right* fornisce l'indice dell'elemento più a destra. L'attributo *low* restituisce l'indice dell'elemento numerato più basso contenuto nell'array, viceversa l'elemento *high* quello più alto. Ad esempio:

```
up'left -> 0                down'left -> 3
up'low  -> 0                down'low  -> 0
```

Tutti questi attributi restituiscono un valore che può essere utilizzato per accedere direttamente all'array. Per esempio, supponendo che il segnale *down* rappresenti un numero dotato di segno (usando la consueta interpretazione che l'elemento più a sinistra sia il più significativo e quindi rappresenti il segno) sarebbe possibile utilizzare un segnale di tipo *bit* per assegnargli il valore del segno del segnale *down*, attraverso la seguente istruzione:

```
sign <= down(down'left);
```

Gli attributi *range* e *reverse_range* sono utilizzati principalmente per controllare i cicli `for..loop`, ma trovano anche impiego nella definizione di sottotipi di segnali in quanto restituiscono il range vincolato di un array. Questo non sarebbe un valore assegnabile ad un segnale, perchè in fase di definizione di un sottotipo non c'è modo di dichiarare un segnale per quello scopo. Ad esempio:

```
signal a : bit_vector (3 downto 0);
signal b : bit_vector (a'range);
```

In questo esempio, il segnale *b* è stato definito con lo stesso range del segnale *a*, questa è una modalità sicura di operare perchè in caso di cambiamento nel range del primo segnale, le modifiche sarebbero automaticamente propagate al secondo. Sempre in questo esempio, l'attributo *range* assegna al segnale il range di valori `3 downto 0`, viceversa l'uso dell'attributo *reverse_range* avrebbe assegnato il range `0 to 3`.

Infine, l'attributo *length* restituisce il numero di elementi presenti in un array, questo valore può essere assegnato a qualsiasi segnale di tipo *integer*. Un uso abbastanza comune di questo attributo è per la cosiddetta *normalizzazione di un segnale*. Ad esempio, dato un segnale con un range "non convenzionale":

```
signal c : bit_vector (13 to 24);
```

è possibile creare un altro segnale, con il medesimo range del segnale *c*, ma normalizzato alla convenzione comune di avere un range discendente, con 0 come estremo inferiore:

```
signal d : bit_vector (c'length - 1 downto 0);
```

Capitolo 4

Validazione e simulazione

1 Validazione ed analisi

Una volta descritta in VHDL, un'entity può essere validata usando un analizzatore ed un simulatore, i quali sono parte di un sistema VHDL. Il primo passo del processo di validazione è l'analisi. L'analizzatore riceve in input un file contenente una o più design unit (si ricordi che una *design unit* è formata da un'entity, una o più architetture associate, ed eventualmente da una configurazione ed un package) e lo compila in una forma intermedia. Il formato di questa rappresentazione non è definita dal linguaggio e quindi è propria del simulatore stesso.

Durante la compilazione, l'analizzatore controlla la sintassi, validandola, ed esegue verifiche statiche sulla semantica. La forma intermedia così generata è mantenuta nella libreria (*design library*) del progetto corrente, la cosiddetta *working library*.

Una design library è una locazione di memoria nell'ambiente in esecuzione, dove sono conservate le descrizioni compilate delle unità. Ciascuna design library ha un nome logico, che è utilizzato per riferirla all'interno di una descrizione in VHDL. Il mapping tra il nome logico della libreria e la locazione dove essa è fisicamente memorizzata non è definito dal linguaggio, bensì dalla logica interna del compilatore; ad esempio, una design library può essere implementata dall'IDE di sviluppo come una directory sul file system in cui

le varie unità sono memorizzate come singoli file.

Un numero arbitrario di design library può esistere simultaneamente, ma solo una di esse è contrassegnata come *working library* (con il nome logico di WORK), ossia la libreria contenente le unità del progetto corrente. L'analizzatore del linguaggio compila sempre le unità in questa libreria, quindi, ad ogni istante temporale, una sola libreria è aggiornata.

Se c'è la necessità di depositare le unità compilate in una libreria differente, il riferimento alla working library deve essere cambiato per puntare alla nuova libreria di destinazione, prima di avviare il processo di compilazione dei sorgenti.

E' ovviamente possibile includere riferimenti a librerie remote sia di sistema, sia definite dall'utente. Il progettista potrebbe, ad esempio, definire una libreria contenente le design unit di alcuni gate basilari (che prevede di utilizzare spesso) e aggiungerla nell'ambiente di progetto corrente, per istanziarne direttamente i componenti all'interno di design unit più complesse. In questo modo si evita di dover ridefinire da zero componenti già realizzati, garantendo una maggiore riusabilità del codice e una maggior efficenza.
Più banalmente, l'uso di librerie esterne è necessario anche nella progettazione di componenti semplici: basti pensare all'uso del tipo *std_logic* che richiede l'importazione della libreria *IEEE.std_logic_1164* o all'uso di funzioni per la lettura e scrittura su file, che utilizzano il package *TEXTIO* presente nella libreria *STD*.

Per importare una libreria all'interno della unit che si sta progettando, si utilizza l'istruzione *include* seguita dal nome della libreria. Quindi è necessario dichiarare quali package e quali funzioni o entità si vogliono utilizzare: questo è fatto per mezzo dell'istruzione *use*. La sintassi dell'istruzione è semplice e ricorda quella presente in linguaggi di più alto livello (quale il linguaggio Java): procedendo da sinistra a destra, la prima keyword è il nome della libreria utilizzata, quindi è presente il package interno alla libreria, per concludere è riporata la funzione o la entity da utilizzare. E' possibile includere tutte le

funzioni o entità presenti nel package riferito, mediante la keyword `all`.

Un esempio contribuirà a rendere più chiaro quanto detto:

```
library IEEE;                    --statement 1
use IEEE.std_logic_1164.all;     --statement 2

entity ffd is
    port(
        Ck, D : in std_logic;
        Q: out std_logic
        );
end;
```

Per poter utilizzare nel codice della unit in realizzazione (si ricordi che la unit è composta come minimo dalla coppia entity e architecture) il tipo di dato *std_logic*, è necessario importare la libreria IEEE (tramite statement 1), quindi utilizzare il contenuto del package *std_logic_1164* (in questo caso, statement 2, si è caricato l'intero contenuto del package, tramite la keyword `all`).

L'IDE utilizzato nello svolgimento di questa tesi (*Active HDL 7.2 Student Edition*, della *Aldec Inc.*) prevede un pratico tool per la gestione delle librerie collegate al progetto. Tramite il *Library Manager* è possibile vedere quali librerie sono attualmente in uso, crearne di nuove o attaccare librerie presenti sul file system, è possibile sganciare (o anche eliminare fisicamente dal disco) le librerie che non sono più utili ed esplorare il loro contenuto.

La finestra del library manager è costituita da due pannelli (fig. 4.1). Il pannello di sinistra mostra la lista delle librerie attualmente attaccate al progetto e i loro parametri.

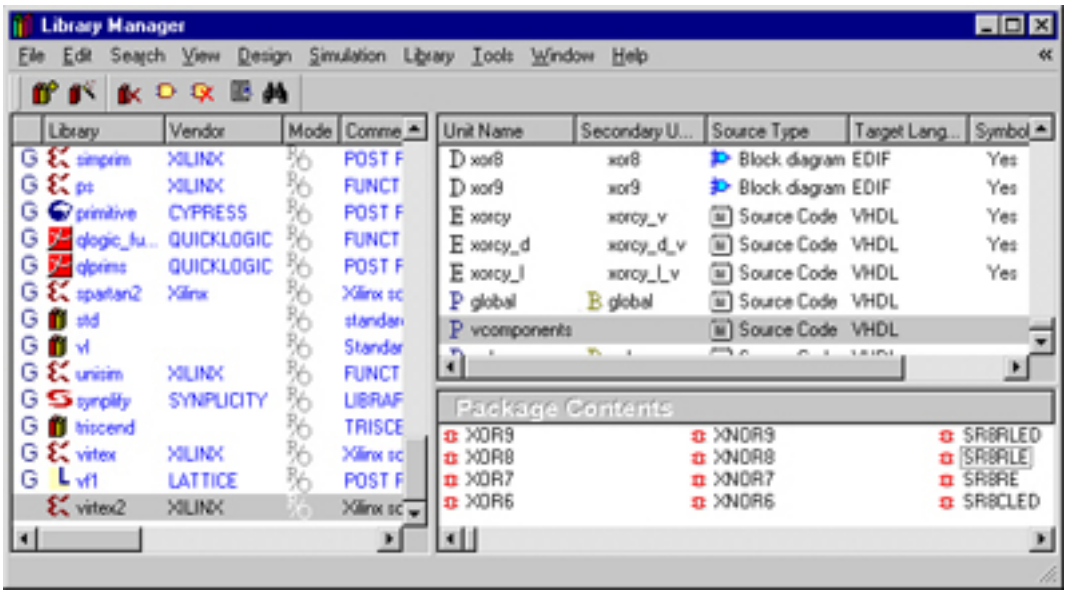


Figura 4.1: Library Manager

Sono presenti quattro colonne:

- *Library type*: mostra il simbolo del tipo di libreria. Può trattarsi infatti di una liberia *globale* (ossia accessibile da tutti i progetti realizzati con *Active HDL* e sempre visualizzata nel library manager - anche se non è stato ancora caricato un workspace. Si tratta delle librerie fornite a corredo del programma, registrate in fase di installazione) o di una libreria *locale* (private per il workspace e il design in cui sono state create). Un progetto non può accedere alle librerie locali di un altro progetto finchè non è eseguita un'esplicita importazione (come visto in precedenza) o finchè non sono converite in librerie globali.
- *Library*: mostra il nome logico della libreria.
- *Mode*: mostra la modalità d'uso della libreria. Può essere accessibile in lettura/scrittura (R/W) o in sola lettura (R/O). Per default, le librerie locali operano in modalità lettura/scrittura, mentre quelle globali in sola lettura. E' comunque possibile variarne la modalità.
- *Comment*: mostra un commento opzionale per fornire una breve descrizione del contenuto della libreria.
- *Directory*: mostra il percorso completo fisico dove è memorizzata la libreria.

Il pannello di destra mostra le unità contenute all'interno della libreria selezionata nel pannello di sinistra. Cliccando con il tasto destro del mouse su una unità è possibile:

- *View Source*: caricare il codice sorgente della design unit selezionata;
- *Open simbol o Delete Symbol*: caricare l'editor dei simoli o cancellare il simbolo selezionato dalla libreria;
- *Copy Declaration*: permette di copiare la dichiarazione del codice VDHL selezionato;
- *Copy Istantiation*: permette di copiare le istruzioni di istanziazione di componenti presenti all'interno dell'unità selezionata;
- *Delete*: rimuove l'unità selezionata;
- *Find*: ricerca l'unità all'interno della libreria selezionata.

Il pannello di destra contiene i seguenti campi:

- *Unit Name*: mostra le unità primarie contenute nella libreria selezionata;
- *Secondary Unit Name*: mostra le unità secondarie contenute nella libreria selezionata. Le sole unità primarie che potrebbero contenere delle unità secondarie sono le entità e i package. Per le altre tipologie di unità, la colonna ripete il nome mostrato nella colonna Unit Name. Ciò è giustificato, dal punto di vista logico, perchè quando tali unità sono referenziate da codice VHDL in un design misto, si assume di avere delle unità secondarie "virtuali" con lo stesso nome.
- *Source Type*: mostra il tipo di documento contenente la descrizione di una specifica unità. Se l'unità primaria è un'entità VHDL, questa colonna mostra il tipo del file sorgente che descrive una specifica architettura, non l'entità stessa. I tipi di sorgente disponibili sono: *Source Code* (se l'unità è descritta direttamente in VHDL), *NetList* (se l'unità è descritta tramite una NetList di tipo EDIF), o *State - Block Diagram* (se l'unità è rappresentata da una macchina a stati o graficamente con blocchi logici).
- *Target Language*: Indica il linguaggio con cui l'unità è stata effettivamente compilata. I linguaggi disopinibili sono VHDL, Verilog, SystemC e EDIF.

- *Symbol*: indica se l'unità primaria ha un simbolo (dalla modellazione grafica a blocchi) nella sua libreria. Se si, è presente il valore *YES*, viceversa *NO*. Il valore FUB è utilizzato se è stato istanziato un blocco funzionale (si veda anche il cap. 2, sez. 5.1) e quest'ultimo è stato compilato nella libreria.

- *Simulation Data*: indica se l'unità contiene i dati di simulazione. Le librerie potrebbero includere unità che non hanno i dati di simulazione al loro interno.

Le *unità primarie* sono costituite da *Entities*: ossia dal risultato della compilazione di una dichiarazione di entità, simulabile solamente in unione con una sua architettura (rappresentato da una *E maiuscola*), *Packages*: ossia dal risultato della compilazione di una dichiarazione di un package (rappresentato da una *P maiuscola*) e *Configurations*: ossia dalla compilazione di una dichiarazione di configurazione (rappresentato da una *C maiuscola*).

Le *unità secondarie* sono costituite da *Architectures*: ossia dalla compilazione di un architecture body. Un'architettura descrive il contenuto della corrispondente entità. Una singola entità può avere diverse architetture che ne definiscono il comportamento (rappresentate da una *A maiuscola*); e da *Package Bodies*: ossia dalla compilazione del corpo di un package (rappresentate da una *B maiuscola*).

Cliccando con il mouse su uno dei package presenti nella libreria selezionata (quindi nel pannello di destra) è possibile vedere (all'interno di un sotto-pannello) il contenuto del singolo package: ossia la lista delle funzioni (simbolo: *f minuscolo*), procedure (simbolo: *p minuscolo*), componenti (simbolo: *freccina rossa*), costanti (simbolo: *c minuscolo*), segnali (simbolo: *s minuscolo*) e variabili condivise (simbolo: *v minuscolo*) presenti al suo interno e istanziabili da codice mediante l'istruzione `use` come precedentemente descritto.

2 Simulazione

Una volta che la descrizione del modello è stata correttamente compilata in una o più design library, il passo successivo del processo di validazione è la *simulazione*. Se a dover essere simulata è un'entità sviluppata gerarchicamente, è necessario che per tutte le sue componenti a basso livello sia fornita la descrizione comportamentale.

Prima di avviare la simulazione, è necessaria una fase di *elaborazione*: in questa fase, la gerarchia dell'entità è espansa, i componenti sono associati alle entità presenti nella libreria e l'entità di più alto livello è costruita come una rete di modelli comportamentali (propri dei sotto-componenti di cui l'unità di alto livello è l'aggregazione) pronta ad essere simulata. Inoltre, nella memoria del calcolatore è assegnato lo spazio per allocare i segnali, le variabili e le costanti dichiarate nelle design unit. Infine, se nelle dichiarazioni delle unità sono presenti istruzioni per maneggiare file, questi vengono caricati e aperti.

Segue quindi la fase di *inizializzazione*: il simulatore parte dall'istante temporale 0. A questo punto tutti i segnali sono inizializzati al loro valore di default, compresi quei segnali e quelle variabili per le quali è stata fatta una dichiarazione esplicita dei valori iniziali. Il simulatore utilizza una lista degli eventi (*event list*) basata sui tempi impostati e una matrice contenente i segnali sensibili (*sensitivity list*) per avviare l'esecuzione dei processi.

All'istante di simulazione zero, tutti i processi sono schedulati per l'esecuzione. Solo uno di essi sarà però avviato e tutte le sue istruzioni sequenziali saranno eseguite, inclusi i loop. Quando l'esecuzione del processo è sospesa (raggiungimento di una condizione di *WAIT*), un altro processo è avviato e così via finchè tutti i processi sono stati eseguiti. Quando tutti i processi sono stati eseguiti si ha il completamento di un ciclo di simulazione.

Durante la sua esecuzione, un processo potrebbe assegnare nuovi valori ad un segnale: questi non sono assegnati immediatamente, ma sono inseriti nella

event list e schedulati per diventare effettivi dopo un certo tempo.

In questo modello temporale, i segnali (non le variabili) sono caratterizzati da una forma d'onda (per i valori precedenti al tempo di simulazione corrente) non modificabile, ma consultabile, da un valore attuale e da eventi schedulati in tempi (o cicli) di simulazione futuri, ovviamente questi eventi non sono consultabili, ma possono essere modificati.

Al completamento di un ciclo di simulazione, la lista degli eventi è scansionata per cercare quei segnali che cambieranno al successivo istante temporale presente nella lista stessa. Può trattarsi di un delta delay o di un vero ritardo (come sarà spiegato nelle sezioni seguenti), in ogni caso il tempo di simulazione è incrementato e le variazioni sui valori dei segnali sono eseguite. A questo punto, tutti i processi che hanno nella sensitivity list un segnale che è appena cambiato sono schedulati per essere eseguiti nel ciclo di simulazione immediatamente successivo.

Il simulatore procede finchè sono presenti eventi schedulati nella event list. La presenza della event list rende possibile simulare le operazioni di processi concorrenti in uni sistema mono-processore. Tutte le operazioni racchiuse in un *process* avvengono in un tempo (*tempo di simulazione*) nullo. In pratica il simulatore esegue le istruzioni all'interno di un processo fino a che non trova l'istruzione *WAIT*, poi passa agli altri processi.

2.1 Il simulatore Aldec

Il simulatore utilizzato in questo elaborato è quello fornito con l'IDE di sviluppo, ossia con *Active HDL 7.2 Student Edition*, sviluppato dalla *Aldec Inc.* Il progettista ha a disposizione diverse modalità per impostare la simulazione e per visualizzarne i risultati.

Prima di discutere le varie modalità di simulazione, è importante chiarire quali siano i passi fondamentali per eseguire una simulazione nell'IDE di riferimento. Un progetto solitamente è composto da numerose coppie entità - architettura in relazione tra loro, ovviamente sia per il testing dei singoli

componenti, sia per il testing della rete finale è necessario eseguire delle simulazioni. In entrambi i casi è necessario dapprima compilare i sorgenti, quindi selezionare dal menù a sinistra quale sia l'architettura *Top-level* (ossia di più alto livello) da utilizzare nella simulazione (esempio in fig. 4.2).

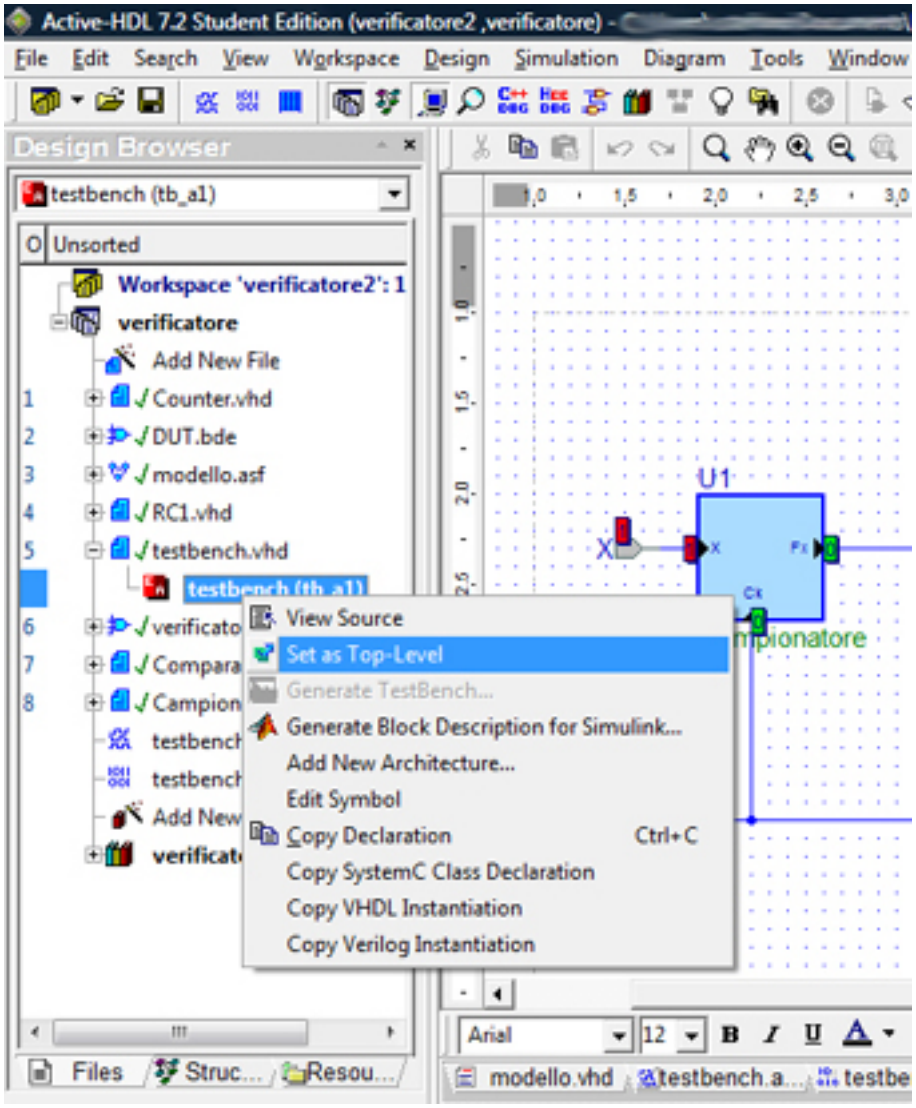


Figura 4.2: Impostazione dell'architettura *Top - level*

In questo modo i dati di simulazione (*simulation data*) creati in fase di compilazione vengono caricati nel simulatore ed è possibile, scegliendo uno dei visualizzatori in seguito citati, definire quali siano i segnali (tra quelli definiti nella entity e nella architecture definita Top-level) da visualizzare e quali

stimoli applicare su di essi. Ovviamente, alla simulazione dell'intera rete (integrazione dei componenti di più basso livello) si sceglierà come coppia entità - architettura quella di un apposito testbench.

Le due sicuramente più importanti sono il visualizzatore di forme d'onda (noto come *Standard Waveform Viewer/Editor Window*) ed il visualizzatore a lista (noto come *List Viewer*).

La Standard Waveform Viewer/Editor Window consiste di due pannelli, come mostrato in fig. 4.3, e consente di visualizzare l'andamento dei valori dei segnali come forme d'onda nel tempo.

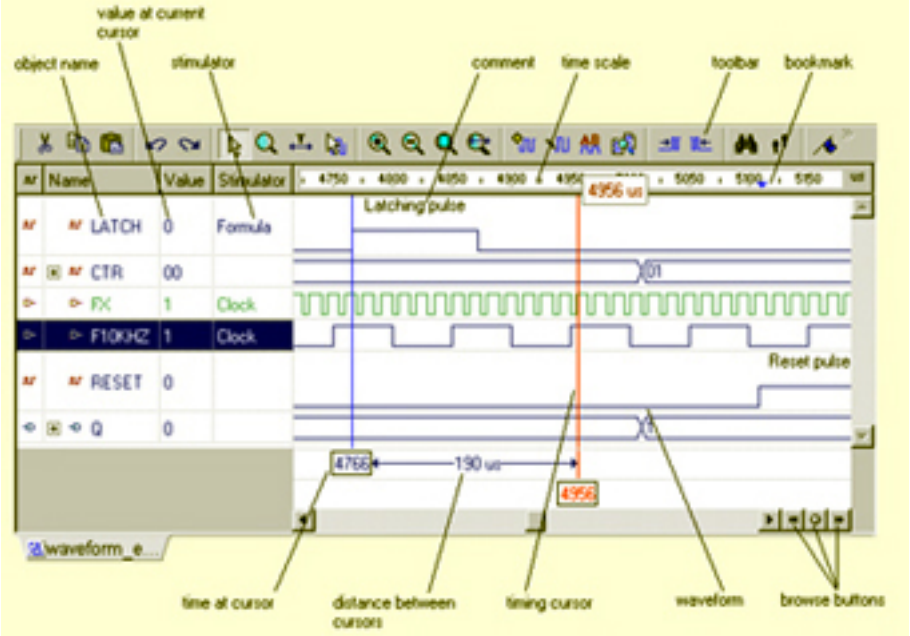


Figura 4.3: Standard Waveform Viewer/Editor Window

Il pannello *Signal Grid* raggruppa le informazioni principali sui segnali, le cui forme d'onda sono visualizzate nel relativo pannello. Il pannello Signal Grid contiene le seguenti voci:

- *Mode*: questa colonna mostra le icone indicanti il tipo di oggetto (ossia se si tratta di un segnale di input, di output o di inout rispetto all'entità che si sta simulando o viceversa se si tratta di un segnale interno all'architettura in simulazione).

- *Name*: questa colonna mostra gli identificatori degli oggetti, ossia i segnali esterni e interni all'unità (o meglio alla coppia entità - architettura) che si sta simulando.
- *Type*: per il VHDL, questa colonna mostra il tipo di oggetto; nel Verilog, il tipo di rete o di registro.
- *Value*: questa colonna mostra il valore attuale del segnale nell'istante di simulazione corrente.
- *Stimulator*: questa colonna mostra il tipo di stimolatore associato a quel segnale, si discuterà in seguito con maggior dettaglio.

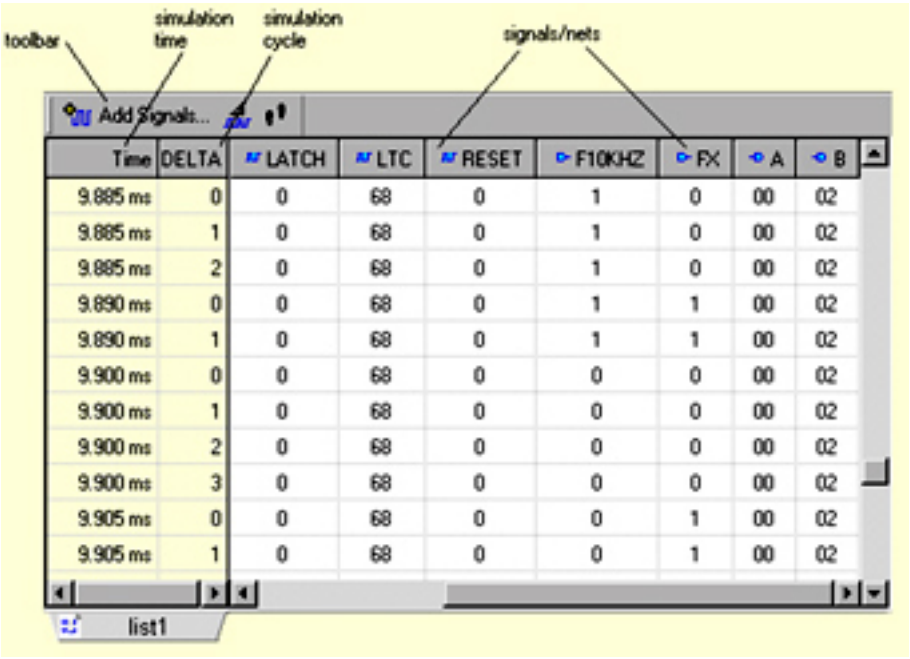
Il pannello *Waveform* mostra le forme d'onda tracciate per i segnali durante la simulazione. Il pannello comprende i seguenti elementi:

- *Time Scale*: la scala temporale è un righello rappresentante l'unità di tempo impostata e i bookmark impostati dall'utente.
- *Timing cursor*: il cursore consente di testare i valori dei segnali lungo l'asse temporale. I valori assunti dai segnali nell'istante temporale marcato dal cursore sono visualizzati accanto ai nomi dei segnali stessi.

La *List Viewer* mostra i valori dei segnali nel tempo, rappresentandoli all'interno di una tabella di verità. Questa visualizzazione, collegata alla precedente, è particolarmente utile per il progettista perchè consente di esplicitare il valore dei segnali in corrispondenza dei vari delta cycle di ciascun istante di simulazione. I delta cycle riportati sono solo quelli collegati a segnali presenti nella coppia entità-architettura che si sta attualmente simulando. I delta cycle possono essere omessi (o meglio, non visualizzati) abilitando l'opzione *collapse delta cycle* (fig. 4.4).

L'IDE di sviluppo è molto completo e offre al progettista anche altre soluzioni per visualizzare l'andamento dei segnali nel tempo.

Se ad esempio, è stata realizzata una *macchina a stati finiti*, utilizzando il compositore grafico per diagrammi di stato, è possibile visualizzare su quest'ultimo la successione delle transizioni sugli stati e lo stato corrente (fig. 4.5). Nella versione per studenti è presente una limitazione che permette di visualizzare entrambe le informazioni solo all'ultimo istante di simulazione,



Time	DELTA	LATCH	LTC	RESET	F10KHZ	FX	A	B
9.885 ms	0	0	68	0	1	0	00	02
9.885 ms	1	0	68	0	1	0	00	02
9.885 ms	2	0	68	0	1	0	00	02
9.890 ms	0	0	68	0	1	1	00	02
9.890 ms	1	0	68	0	1	1	00	02
9.900 ms	0	0	68	0	0	0	00	02
9.900 ms	1	0	68	0	0	0	00	02
9.900 ms	2	0	68	0	0	0	00	02
9.900 ms	3	0	68	0	0	0	00	02
9.905 ms	0	0	68	0	0	1	00	02
9.905 ms	1	0	68	0	0	1	00	02

Figura 4.4: List dei segnali e visualizzazione dei *delta cycle*

mentre nelle versioni complete il processo è mostrato run-time. Si noti che la visualizzazione è possibile anche se l'architettura definita come top-level non è quella che implementa il diagramma degli stati, l'importante è che quest'ultima sia in qualche modo collegata al grafo stesso (ad esempio, utilizzando il componente descritto dalla macchina a stati).

Anche il *diagramma a blocchi* (*block diagram*) può essere utile in fase di simulazione, perchè in esso sono riportati i valori dei segnali (per ogni pin o cavo di collegamento) run-time, similmente a quanto avveniva nel diagramma degli stati (fig. 4.6).

Ci sono tre differenti modalità per lanciare una simulazione, ciascuna di esse con le proprie caratteristiche:

- il comando *Run* lancia la simulazione e la fa proseguire per un periodo di tempo non definito. La simulazione termina quando si verifica una delle due seguenti condizioni: il tempo di simulazione corrente è uguale a TIME'HIGH (ossia l'estremo superiore del tipo *time*: massimo valore assumibile dal tempo di simulazione), oppure non ci sono più

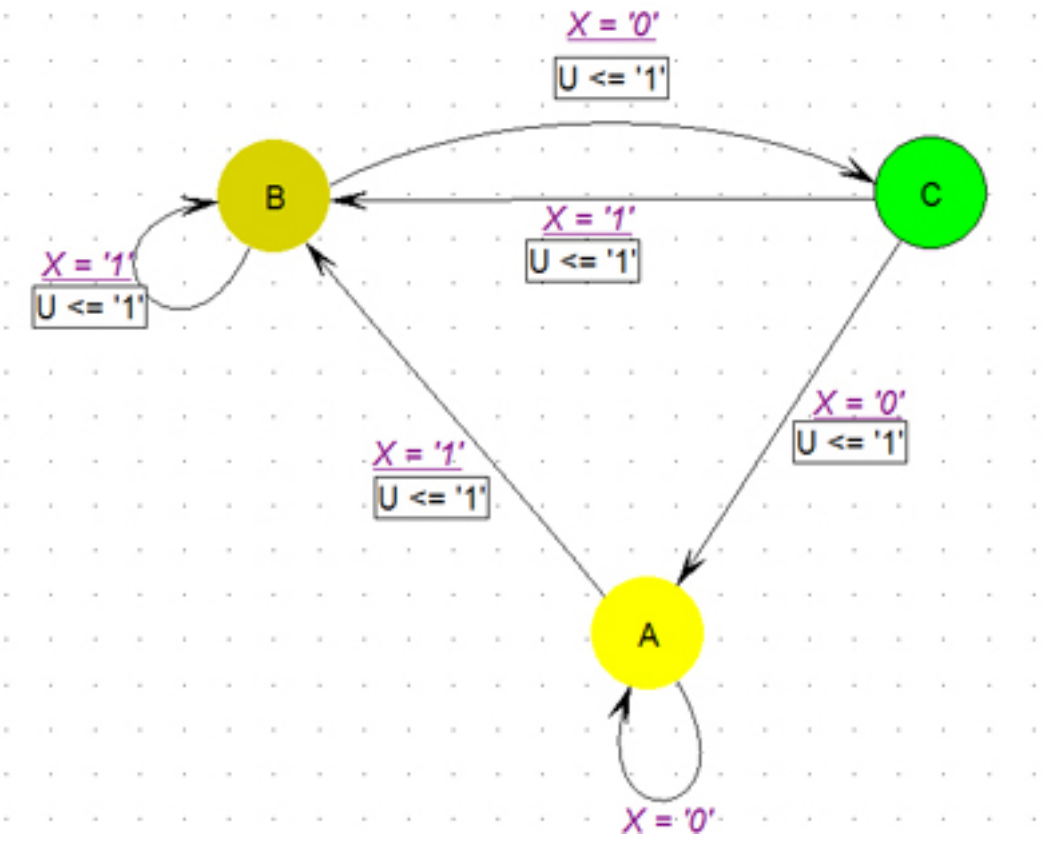


Figura 4.5: Editor grafico FSM durante la simulazione

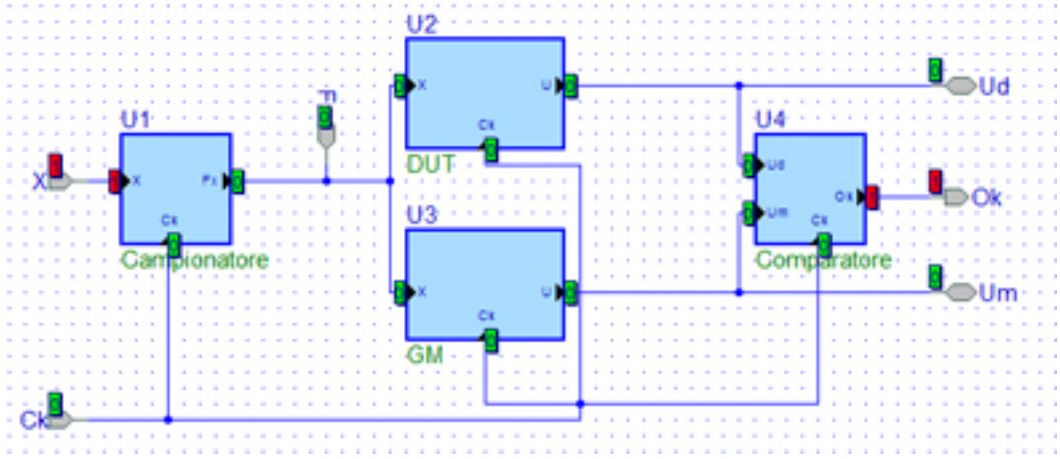


Figura 4.6: Block Diagram con visualizzazione dei *probe* sui segnali, durante la simulazione

eventi o processi da risvegliare schedulati. Con questa modalità non è possibile variare i segnali manualmente (con il toggle da tastiera), perchè il simulatore ha bisogno di definire la lista degli eventi in maniera deterministica e predefinita.

- il comando *Run For* lancia la simulazione e la fa proseguire per un determinato periodo di tempo. Tale periodo di tempo è quello inserito nel *Simulation Step box* localizzato nella barra di simulazione. Premendo più volte il pulsante la simulazione prosegue per il periodo di tempo previsto a partire dall'ultimo istante simulato precedentemente (ad esempio due pressioni del pulsante, con un tempo di simulazione pari a 50ns, originano una simulazione di durata complessiva 100 ns). Solitamente, quando si utilizzano variazioni toggle dei segnali, si agisce sfruttando la sovrapposizione degli effetti: dapprima viene simulato l'intervallo temporale col segnale originario, quindi agendo da tastiera (a simulazione ferma) si varia il valore del segnale e si rilancia la simulazione a partire da quell'istante temporale. Il risultato è la simulazione equivalente.
- il comando *Run Until* lancia la simulazione e la fa proseguire fino al raggiungimento dell'istante temporale specificato.

Sono inoltre disponibili anche i pulsanti (fig. 4.7) per resettare la simulazione (o meglio, per riportare all'istante zero il cursore temporale) e per fermarla.

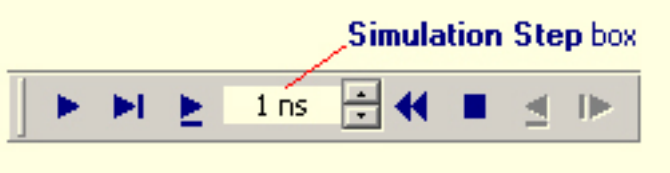


Figura 4.7: Simulation toolbar

In realtà, fino ad ora si è tralasciata una componente fondamentale per l'esecuzione sensata di una simulazione: è infatti necessario fornire ai segnali entranti nell'entità (o ai segnali propri dell'architettura) degli stimoli, affinchè essi varino e generino così dei nuovi eventi da simulare. Gli stimoli possono essere forniti al simulatore o attraverso appositi tool grafici o codificandoli all'interno del codice, si esamineranno ora entrambe le soluzioni.

La definizione di stimoli con la *modalità grafica* è possibile utilizzando sia il visualizzatore di forme d'onda sia il visualizzatore a lista. Per ogni segnale cui si vuole aggiungere uno stimolo è sufficiente cliccare la cella corrispondente alla colonna *Stimulator*, ciò provoca l'apertura di una nuova finestra, dove si può selezionare una delle opzioni disponibili:

- *Clock*: pilota il segnale con un'onda pulsante (un clock): è possibile settare il valore iniziale dello stesso, un offset, il periodo (o la frequenza) e il duty cycle dell'onda così ottenuta.
- *Counter*: pilota il segnale con una sequenza di valori rappresentata dagli stati di conteggio di un contatore. Sono disponibili vari attributi configurabili dall'utente quali il modulo del contatore, il valore iniziale, la velocità di conteggio, ecc.
- *Custom*: permette all'utente di definire manualmente nell'editor la forma d'onda da assegnare al segnale.
- *Formula*: permette all'utente di definire una formula con cui assegnare i valori al segnale. Utilizzare la formula equivale a settare il valore dei segnali con una semantica *"after X unità di tempo"*, ossia specificando il valore assunto del segnale a partire da X unità di tempo (il riferimento è assoluto, quindi sempre rispetto all'origine). Utilizzando una formula è possibile definire anche cicli di ripetizioni (anteponendo il parametro *-r* al l'istante temporale da cui si vuol ripetere la sequenza) o stabilire per quante volte ripetere un'assegnazione di un valore.
- *Hotkey*: consente di variare il valore di un segnale (in modalità toggle) con la pressione di uno specifico tasto della tastiera (si veda anche la sez. 2.1).
- *Value*: forza il segnale ad assumere il valore costante impostato per tutta la durata della simulazione.
- *Random*: pilota il segnale con una sequneza di valori interi, distribuiti in accordo ad una funzione di probabilità standard.

Infine, dopo aver selezionato uno dei comportamenti, è possibile impostare come gli stimoli contribuiscano al valore del segnale pilotato, in altre parole si definisce la forza d'applicazione (*Strength*):

- *Override*: il valore applicato dallo stimolatore sovrascrive quelli forniti al segnale nel codice della architecture.

- *Drive*: lo stimolo si comporta come un driver aggiuntivo collegato al segnale, quindi viene assegnato parallelamente ai valori codificati nell'architettura.
- Deposit: lo stimolo applicato sovrascrive i valori forniti nel codice finchè non c'è una successiva transizione nel driver collegato al segnale.

Spesso però il progettista si trova a dover simulare componenti con molti segnali in ingresso o con segnali che variano molto spesso, in questo senso l'uso della modalità grafica di applicazione degli stimoli può rivelarsi troppo onerosa. In questi casi, è preferibile applicare gli stimoli ai segnali *codificandoli internamente al codice*; per far ciò è bene creare un testbench in cui inserire come componente la rete da simulare e fornire i valori degli ingressi attesi mediante processi dedicati (che operino concorrenzialmente alla rete stessa): questi processi potrebbero leggere i valori da file testuali, oppure da vettori codificati nell'architettura del testbench o potrebebro assegnare i valori tramite formule (utilizzo di `after` e `wait for`). Si parlerà più dettagliatamente dei testbench e dei loro pattern costruttivi nel capitolo loro dedicato (cap. 5).

Capitolo 5

I testbench

Un *testbench* è uno strumento utilizzato per simulare e verificare la correttezza di un modello hardware. La potenza espressiva del linguaggio VHDL permette al progettista di scrivere il testbench nello stesso linguaggio con cui ha modellato il componente da simulare.

In definitiva, un testbench ha tre compiti fondamentali:

1. *generare* gli stimoli per la simulazione (forme d'onda);
2. *applicare* gli stimoli all'entità sotto test e raccoglierne gli output;
3. *confrontare* gli output prodotti con i valori attesi.

Il modo migliore di operare una simulazione è creare un modello esterno (quindi dotato di una propria coppia entità - architettura) che istanzi al suo interno l'entità da simulare. Questo è giustificato per valutare il comportamento dell'entità dall'esterno, senza alterarne il contenuto con istruzioni di debug o non proprie della stessa; allo stesso modo è logico istanziare il livello più alto di un circuito con molti componenti al suo interno, proprio come se si avesse fisicamente sottomano una scheda e si collegassero le sonde dell'oscilloscopio ai suoi terminali.

La struttura tipica di un testbench è la seguente:

```
entity test_bench is
end;
```

```
architecture tb_behaviour of test_bench is
   component entity_under_test is
      port(< list_of_ports >);
   end component;
   < local_signal_declarations >;
begin
   < Waveform generation >
    EUT : entity_under_test port map (< port-associations >);
   < Results emission >
end;
```

Come si può notare, l'entità non contiene la dichiarazione di segnali; questo perchè un testbench non modella un dispositivo fisico, quindi non ha bisogno di definire segnali verso l'esterno. I segnali utilizzati per operare sull'unità in testing sono specificati all'interno della singola architettura. Chiaramente, è perfettamente lecito definire varie architetture nell'ambito dello stesso testbench, qualora si vogliano agevolmente considerare differenti configurazioni dei segnali.

Prima di entrare nel dettaglio sulla scrittura di un testbench, è bene esaminare come utilizzare un testbench per condurre delle simulazioni.

Una prima idea, può essere quella di limitare il testbench alla semplice generazione degli ingressi (codificandoli direttamente nel codice) e alla loro applicazione al *DUT* (d'ora in avanti si utilizzerà questa sigla, in luogo di *Device Under Test*), sarà poi compito del progettista visualizzare i risultati (con le forme d'onda o come lo ritiene più opportuno) e confrontarli con un riferimento. Il procedimento è riassunto nella fig. 5.1.

Una soluzione del genere ha valenza prettamente didattica, ma è inapplicabile per sistemi complessi con molti segnali, per l'evidente difficoltà di dover esaminare manualmente i dati.

Un miglioramento alla soluzione precedente si ottiene con la lettura e la scrittura degli input su file di testo; lo schema di funzionamento è analogo,

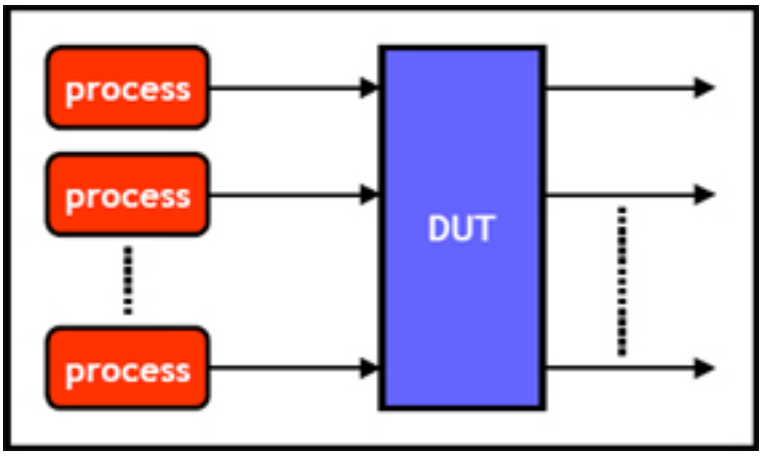


Figura 5.1: Generazione degli ingressi interna al testbench e applicazione al DUT

in questo caso però gli ingressi anzichè essere modellati nel codice, sono presenti in file, e le uscite sono memorizzabili per usi futuri. Il vantaggio è quello di avere un codice indipendente dalla singola configurazione, più generale e riutilizzabile per prove differenti (eventualmente sostituendo la DUT), ma presenta comunque lo svantaggio di dover effettuare manualmente l'analisi (fig. 5.2).

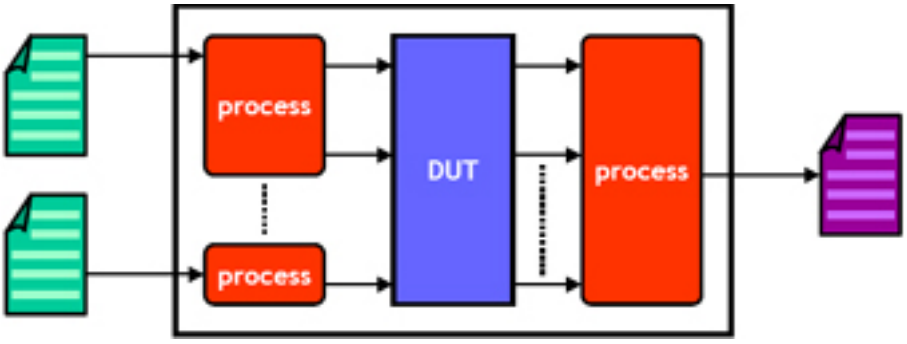


Figura 5.2: Lettura degli ingressi da file e salvataggio delle uscite

Un modus operandi più professionale è quello di prevedere la *verifica automatica* dei risultati generati. Questo schema prevede la lettura da file degli ingressi da applicare al DUT e delle uscite attese, quindi l'uscita del dispositivo è collegata ad un comparatore che verifica la concordanza della stessa

con quella prevista. Il controllo di correttezza deve avvenire in un processo sincronizzato in base ad un evento opportuno, tipicamente il segnale di clock, oppure tramite un'entità definita a tale scopo (ma sempre sincronizzata); in caso contrario il testbench potrebbe rivelarsi inutilizzabile, perchè la simulazione (e il DUT) potrebbero introdurre ritardi è necessario tenere conto. Tale pattern è riassunto dalla fig. 5.3.

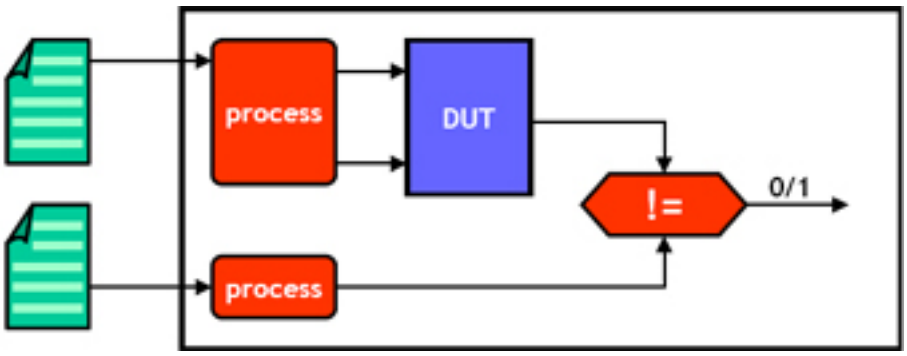


Figura 5.3: Verifica automatica, interna al testbench

In alternativa, anzichè fornire le uscite previste tramite un file esterno, si può ricorrere ad un golden model. Un *golden model* è una visione del sistema ad alto livello che ne modella il comportamento. Si immagini di modellare un circuito realizzato con la connessione di molte componenti (quindi stile strutturale - cap. 2, sez. 2.2) - ad esempio in RTL, parallelamente è possibile modellare anche la macchina a stati che ne descrive il comportamento (ad esempio tramite l'editor FSM incluso nell'IDE); a questo punto è sufficiente fornire ad entrambe i segnali in input per la simulazione e confrontare automaticamente le risposte emesse. Tale pattern, che sarà analizzato tramite un esempio concreto nel capitolo 6, sez. 5, è riassunto dalla fig. 5.4

Infine, un'altra modalità di verificare un sistema è la cosiddetta *verifica intrinseca*. Alcuni sistemi sono costituiti da due parti che svolgono funzionalità inverse, ad esempio può essere presente un modulo di codifica e un modulo di decodifica, oppure una trasformazione e la relativa anti-trasformazione, è possibile utilizzare una delle due sezioni per verificare l'altra. Un pratico esempio può essere quello di una coppia encoder - decoder, il testbench ge-

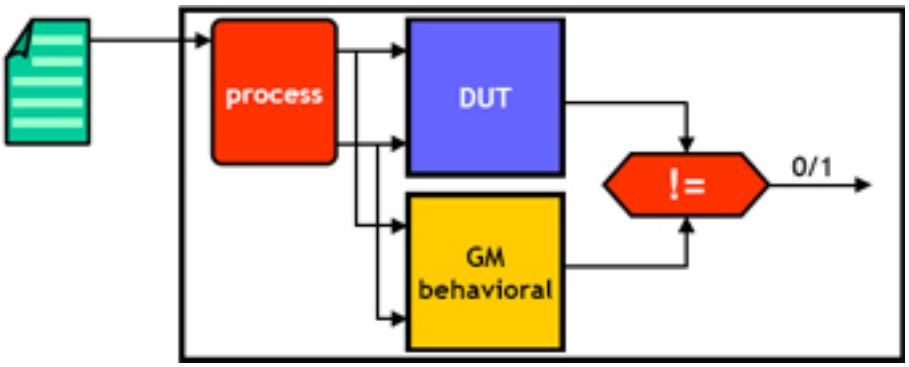


Figura 5.4: Verifica automatica con Golden Model

nera gli ingressi e li applica all'encoder cui in cascata è applicato il decoder, quindi confronta gli ingressi originari - opportumanente ritardati - con l'uscita della rete in testing (fig. 5.5).

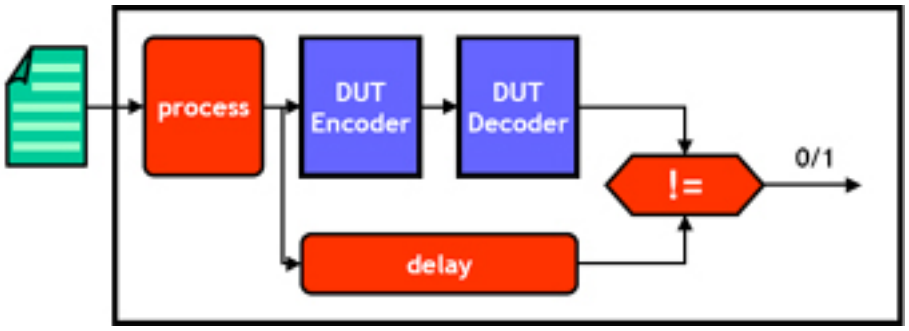


Figura 5.5: Verifica intrinseca

1 Generazione di forme d'onda

Per generare uno stimolo sui segnali di ingresso al DUT è necessario definire le forme d'onda su tali segnali ed applicare lo stimolo ad intervalli di tempo discreti. Una soluzione può essere inserire manualmente la configurazione voluta nel simulatore attraverso l'editor grafico messo a disposizione dall'IDE (vedi cap. 4, sez. 2.1), questa operazione però è abbastanza inefficiente se si hanno molti segnali e scarsamente portabile, soprattutto se si deve lavorare

su ambienti di sviluppo differenti.

Una soluzione senza dubbio migliore, è quella di utilizzare un processo - all'interno dell'architettura del testbench - che generi gli stimoli sui segnali tramite istruzioni di assegnamento o statiche o generate da una lettura da file. Se ne darà ora una visione più approfondita.

1.1 Assegnamenti statici

Le istruzioni di *assegnamento statico* sono quelle istruzioni in cui il valore del segnale è fisso, impostato dal programmatore, assieme alle tempistiche di variazione. Esse, solitamente, sono inserite all'interno di processi e le tempistiche sono modellate tramite istruzioni `wait for`; i processi per la generazione dei segnali di ingresso hanno la forma seguente:

- *segnali aperiodici*:

```
GEN_X : process
begin
   X <= value1;
   wait for time1 ns;
   X <= value2;
   wait for time2 ns;
   ...
   X <= valueN;
   wait;
end process;
```

- *segnali periodici*:

```
GEN_X : process
begin
   X <= value1;
   wait for time1 ns;
   X <= value2;
   wait for time2 ns;
   ...
   X <= valueN;
```

```
        wait for timeN ns;
    end process;
```

La differenza è proprio sull'ultima istruzione di `wait`: nel caso di segnali aperiodici, il processo completati gli assegnamenti viene sospeso indefinitamente, viceversa nei segnali periodici - non essendoci un wait indefinito - il processo, al risveglio dall'ultima istruzione di wait inizia un nuovo loop. Uno schema del genere è generalmente utilizzato per la creazione, rispettivamente, dei segnali di *reset* e di *clock*.

Un problema che si può manifestare è che il simulatore non fornisce la possibilità di controllare quanto tempo di simulazione è trascorso, in altre parole, se la simulazione è stata avviata in *Run* non vincolato, essa proseguirà fino al massimo tempo predefinito dall'ambiente (*time'high*) indipendentemente dalla assenza di altri istanti nella *event list*. Esistono un paio di *work around* per risolvere questo problema: una prima forma prevede l'utilizzo di un costrutto `assert` che verifichi l'arrivo ad un certo istante temporale o la variazione di uno specifico segnale (ad esempio, la commutazione ad '1' del segnale *End*). Ad esempio:

```
assert NOW <= 1000ns
   report "Simulation completed successfully"
   severity error;
```

Un altro modo è interrompere il processo di generazione del clock, con una condizione del tipo (vincolata al raggiungimento di un particolare istante temporale o ad una condizione sul valore di un segnale):

```
if NOW > 1000ns then
   wait;
end if;
```

1.2 Assegnamenti dinamici

Utilizzando il package `textio`, presente nella libreria *STD*, è possibile operare su file, ossia accedere in lettura o scrittura a file presenti sul filesystem. Ovviamente, rispetto a linguaggi di più alto livello, le possibilità di gestione

sono molto limitate: il file da caricare è codificato staticamente all'interno del codice e non sono disponibili metodi di supporto per fare altre operazioni sul filesystem (ad esempio la cancellazione di un file, ecc. - anche perchè le stesse non rientrano nell'ottica di principio del VHDL stesso).

Il linguaggio VHDL definisce il tipo file solo come strumento di supporto per la memorizzazione e lo scambio di dati, infatti un file non ha un corrispettivo fisico e le sue istruzioni non sono sintetizzabili.

Un file è gestito come un insieme di righe, il package definisce quindi due funzioni (sia per l'input che per l'output) per operare con essi. L'apertura di un file avviene con la dichiarazione `file <handle>: text is in <file>;` un file può aperto in lettura/scrittura (dipende solo dalle operazioni effettuate). Da un file si leggono unicamente stringhe (non interpretate) ed è necessario dichiarare almeno una variabile (di tipo *line*) per la lettura di linee di testo.

Solitamente la *lettura* di un file è realizzata con un ciclo loop, la cui condizione è data dalla funzione *endfile* (valore *true* alla lettura dell'ultima riga del file). Ad ogni loop è letta una riga del file, attraverso la funzione `readline()` (che restituisce un tipo di dato *line*, concettualmente un buffer della dimensione di una linea), la lettura dei singoli valori è fatta dalla linea così estrapolata attraverso l'istruzione `read()` (che restituisce il tipo di dato atteso dal target dell'assegnazione). Questo comportamento evidenzia la necessità di conoscere in anticipo il tipo di dati presenti nel file e la loro posizione; in altre parole è necessario definire una semantica per il contenuto del file stesso.

La scrittura su un file avviene simmetricamente, con la scrittura dei singoli valori sulla linea buffer (tramite la funzione `write()`) e la successiva scrittura del buffer nel file stesso, mediante la funzione `writeline()`.

2 Pattern per la scrittura di testbench

Scrivere un testbench che operi correttamente non è particolarmente difficile, ma - soprattutto all'aumentare dei segnali coinvolti - è bene seguire sempre

delle regole di riferimento; queste sono presentate all'interno della sezione corrente.

Quando si progetta un processo per la *generazione del clock*, è importante utilizzare *istruzioni di sospensione* (`wait for`) e non *istruzioni di scheduling* (`after`) per creare il timing. Ciò è dovuto al seguente motivo: il ciclo di simulazione ignora le clausole after all'interno di istruzioni di assegnamento ai segnali, poichè il riferimento temporale è assoluto (quindi sempre riferito all'origine); ad esempio il seguente codice non è corretto:

```
junk : process
begin
   CLK <= '0', '1' after 25 ns; -- errore
   wait for 50 ns;
end process;
```

La prima esecuzione del loop avviene correttamente, perchè la variazione del segnale è schedulata correttamente a 25 ns, ma successivamente all'istruzione `wait for 50 ns` il tempo di simulazione si porta a 75 ns (si ricordi che il tempo in una istruzione wait for è relativo, mentre nella after è appunto assoluto), chiaramente al riavvio del loop l'istruzione di assegnazione non è più valida (la soluzione corretta è mostrata nella sez. 1.1).

Può presentarsi la necessità di avere più clock sincronizzati tra loro, ad esempio generandoli a partire dal clock principale. In questo caso, bisogna prestare attenzione ai delta cycle introdotti tra il clock principale e quelli derivati. Questi delta potrebbero causare comportamenti indesiderati se all'interno del circuito sono utilizzati sia il clock principale che quelli secondari. E' possibile prevenire il problema con il seguente tipo di codice:

```
divider: process
begin
   clk50 <= '0';
   clk100 <= '0';
   clk200 <= '0';
   loop -- forever
```

```
      for j in 1 to 2 loop
          for k in 1 to 2 loop
               wait on clk;
               clk200 <= not clk200;
          end loop;
          clk100 <= not clk100;
      end loop;
      clk50 <= not clk50;
   end loop;
end process divider;
```

Si noti che, essendo all'interno di un unico processo, i clock derivati (*clk50*, *clk100* e *clk200*) hanno la stessa frequenza del clock principale (*clk*), ma sono ritardati rispetto al principale di un delta cycle (questo è perfettamente normale, per quanto visto in cap. 2, sez. 3.1). Ad ogni modo, i clock derivati sono aggiornati contemporaneamente e devono essere gli unici ad essere utilizzati nel circuito, in altre parole il clock principale deve essere utilizzato solo per la generazione dei derivati (eventualmente un suo derivato può essere in rapporto 1:1 con il principale).

Nella scrittura di un testbench è importante prevedere un processo che si occupi della *generazione del segnale di reset*, questo ha la caratteristica di far partire il sistema da una configurazione nota e stabile. Un processo del genere è equivalente alla generazione di un segnale asincrono, come mostrato nella sez. 1.1.

Nella *generazione di dati* è necessario evitare corse critiche tra gli stessi e il clock: l'applicazione contemporanea di dati e del fronte attivo del clock può causare risultati inaspettati. Per mantenere i dati sincronizzati ed evitare le corse critiche, i dati vanno applicati in un momento differente dal fronte attivo, ad esempio al fronte inattivo.

La figura 5.6 riassume le regole sovra-citate.

Figura 5.6: Processi indipendenti per la generazione dei segnali di simulazione

2.1 Semantiche temporali

Quando si leggono i valori da applicare ai segnali di un circuito da simulare (ad esempio da un file dati) è necessario definire una semantica, ossia come essi debbano essere applicati run-time.

Si consideri ad esempio il seguente file, in cui sono riportati i valori di un segnale e degli istanti temporali:

```
#val #delay
1 0
0 10
1 30
0 40
0 50
0 60
1 70
0 80
0 90
0 100
1 110
```

in questo caso, si è deciso che i valori nella colonna più a destra siano gli istanti di applicazione del nuovo valore del corrispondente segnale (chiaramente, il segnale manterrà immutato il valore fino alla successiva variazione).

Con questa soluzione (peraltro analizzata più in dettaglio nel cap. 6, sez. 5.5), il progettista ha la responsabilità di assegnare dei valori che rispettino (oltre la sintassi e i tipi di dato) la semantica imposta dal processo stesso. In particolare, si impegna a fornire valori che evitino corse critiche con il segnale di clock e che siano accettabili (ad esempio, che il tempo di permanenza del segnale su un certo valore sia maggiore del tempo di campionamento del circuito).

Una semantica alternativa potrebbe essere quella di definire la colonna più a destra come il numero di unità di tempo per cui il segnale deve assumere il corrispondente valore. Questa soluzione (anche essa descritta in maggior dettaglio nel cap. 6, sez. 5.5) ha il vantaggio di scaricare il progettista dalle responsabilità sulle corse critiche e sui tempi di hold, ma rende più difficoltoso modellare segnali con variazioni non regolari. Un altro aspetto vantaggioso di questa soluzione è la possibilità di variare la *frequenza* o il *duty cycle* del clock, senza dover modificare la temporizzazione dei segnali, che risultano così *conformanti*.

2.2 Altre utili funzioni di conversione

Nella scrittura di un testbench può capitare di dover fare conversioni ai valori dei segnali, ad esempio per definire istanti temporali (quindi tipo di dato *time*) a partire da valori interi, oppure per poter stampare valori numerici.

Se si vuole convertire un integer in un tipo time, il modo più semplice per farlo è moltiplicare l'intero con l'unità di tempo richiesta. Ad esempio, si consideri il seguente codice:

```
variable data : integer;
variable timer : time;
...
timer := data * 1 ns;
```

La variabile data è un integer, moltiplicandola per l'unità (1 ns), l'espressione `data * 1 ns` restituisce un valore del tipo time. Il viceversa si ottiene agevolemente dividendo il valore di tipo time per l'unità di riferimento, ottenendo

così l'equivalente intero.

Infine, se si avesse necessità di stampare a video o su file una stringa, è necessario passare come parametro in input alla funzione `write()` una stringa definita con la seguente sintassi (ovvero avvertendo esplicitamente che il parametro passato è di tipo *string*, con un discorso simile a quello visto per le aggregazioni): `string'("stringa")`.

Capitolo 6

Casi di studio

1 Controller semaforico

Questo primo caso di studio si basa sulla progettazione di una semplice centralina semaforica. Il sistema di controllo di un impianto semaforico posto all'incrocio di due strade deve operare secondo due distinte modalità di funzionamento, selezionate rispettivamente dal valore logico 1 (funzionamento diurno) o dal valore logico 0 (funzionamento notturno) di un segnale di ingresso X, proveniente da un timer elettromeccanico esterno.

Il funzionamento diurno prevede la generazione ciclica delle seguenti quattro segnalazioni:

1. semaforo verde sulla direttrice 1 - rosso sulla 2;
2. semaforo giallo sulla direttrice 1 - rosso sulla 2;
3. semaforo rosso sulla direttrice 1 - verde sulla 2;
4. semaforo rosso sulla direttrice 1 - giallo sulla 2.

Il funzionamento notturno prevede la generazione ciclica ed ordinata delle seguenti due segnalazioni:

1. semaforo spento su entrambe le direttrici;
2. semaforo giallo su entrambe le direttrici.

A seguito di una variazione del segnale X, la commutazione dall'una all'altra modalità di funzionamento può aver luogo solamente al termine della generazione di un semaforo giallo su una delle due direttrici nel caso di passaggio

al funzionamento notturno, della segnalazione tutto spento nel caso di passaggio al funzionamento diurno.

Allorchè il segnale di *Reset* è attivo (a seguito dell'accensione dell'impianto o della pressione di un pulsante previsto a tale scopo), tutte le lampade devono essere mantenute spente. Il sistema di controllo dovrà poi attuare la modalità di funzionamento selezionata dal segnale X, a partire dalla corrispondente prima segnalazione, non appena il segnale Reset si disattiva.

Il sistema è stato modellato con le seguenti unità funzionali (FUB) descritte nelle seguenti sezioni.

1.1 Unità di temporizzazione

L'*unità di temporizzazione* ha il compito di generare un segnale di clock di frequenza 1 Hz a partire da un segnale digitale di frequenza 50 Hz e duty cycle del 50% derivato tramite apposito circuito di rettificazione della tensione di alimentazione del sistema.

La rete è stata progettata con lo stile *strutturale*, utilizzando l'editor grafico fornito dall'IDE (fig. 6.1).

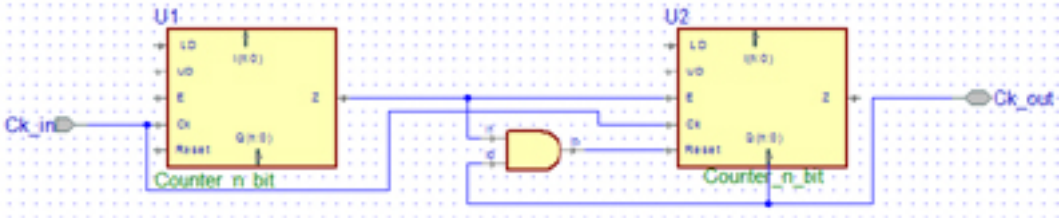


Figura 6.1: Unità di temporizzazione

Come si vede, sono presenti due istanze di contatori generici ad n-bit (sono istanze di simboli, in quanto è normale prevedere l'utilizzo del simbolo più volte nella stessa architettura) collegate con delle retroazioni. L'uso di *generici* permette di riutilizzare il componente in situazioni differenti, semplicemente assegnando il valore del parametro in fase di istanziazione (attraverso

l'istruzione `generic map(<valore>)`). Nello specifico i due contatori sono a 4-bit, *U1* conta modulo 16, mentre *U2* è modellato nel codice per resettarsi (attraverso il *simbolo AND*) quando la sua uscita è `Q = "0100"`.

Il segnale *Ck_out* costituisce l'uscita dell'unità; è stata settata come *inout* (si noti il simbolo grafico differente) perchè la connessione, cui il segnale è collegato, è anche entrante nell'AND. Se si fosse settata l'uscita come *out* il compilatore avrebbe segnalato un errore, perchè è impossibile che un segnale uscente sia sul lato destro di un'espressione.

1.2 Unità di controllo

L'*unità di controllo* ha il compito di identificare via via la segnalazione corrente da generare, in dipendenza del valore assunto dal segnale di ingresso *X*, opportunamente sincronizzato tramite un circuito di campionamento, ed in accordo alle precedenti regole delineate.

Si è deciso di modellare l'unità di controllo tramite una macchina a stati finita (in fig. 6.2), sfruttando così le capacità di sintesi del compilatore. La scelta di modellare un controllore con una *FSM* piuttosto che a più basso livello (es. *RTL*) deriva dalla volontà di rendere facilmente comprensibile il comportamento del circuito e di poterlo modificare o sviluppare facilmente.

Le uscite sono tutte *combinatorie* (cap. 2, sez. 4.1), nello schema si vedono alcune istruzioni circondate da un quadrato nero: si tratta delle istruzioni associate alle *transizioni* o allo stato (di fatto si è realizzata una macchina di *Mealy*), mentre le istruzioni in viola sono le condizioni che permettono alle transizioni di scattare. Tutte le istruzioni presenti all'interno dello schema devono rispettare la sintassi del linguaggio, pena il fallimento nella generazione del corrispondente VHDL modellante lo schema.

1.3 Unità di memoria

Il tempo di permanenza della segnalazione corrente è reso disponibile dall'*unità di memoria*, la quale comprende quattro locazioni di otto bit indirizzabili tra-

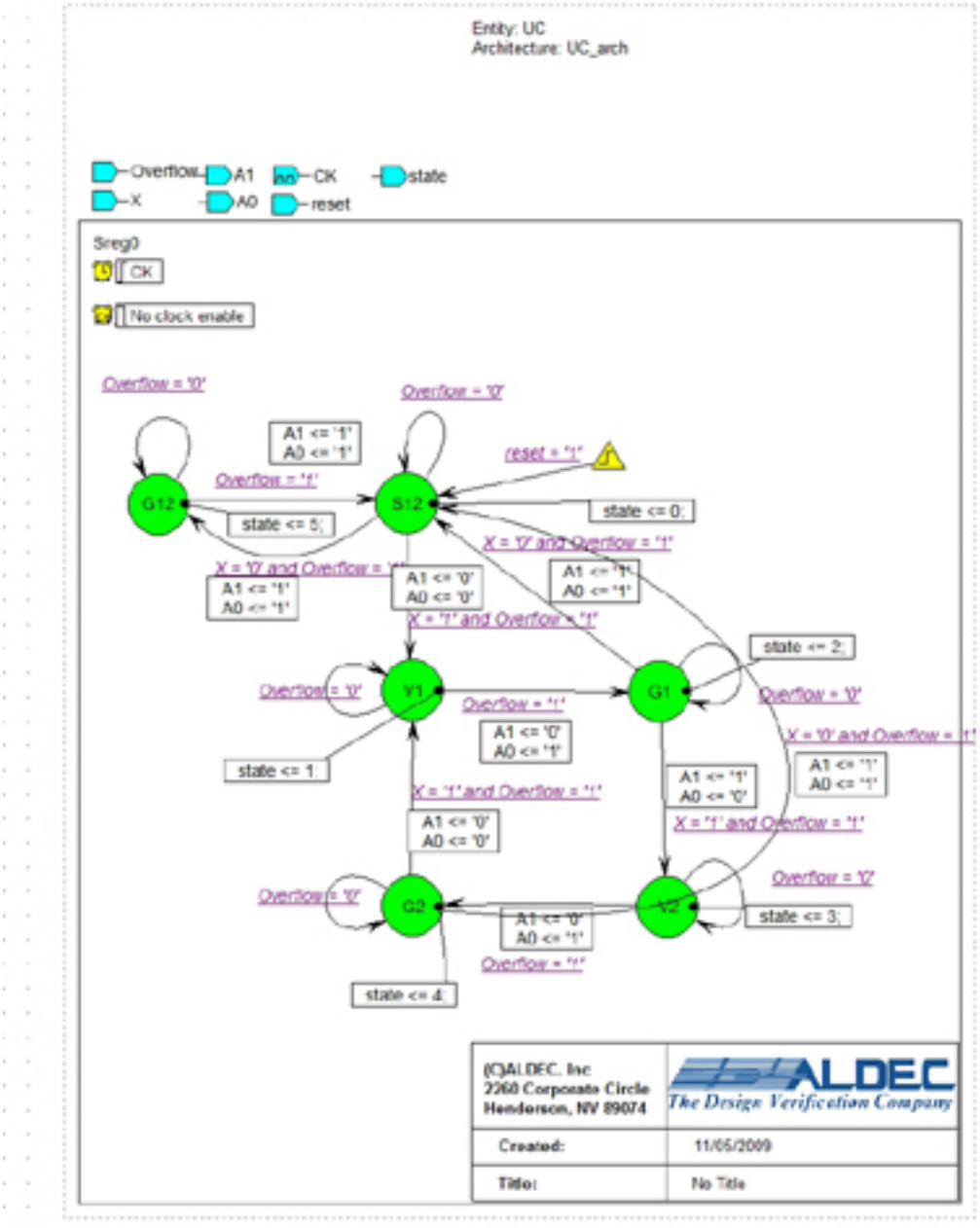


Figura 6.2: Unità di controllo

mite i segnali *A1* e *A0*, in cui sono nell'ordine riportati i valori della durata di ogni singola fase, ciascuno espresso in secondi e rappresentato mediante due cifre *BCD*.

L'unità è stata modellata direttamente in VHDL, con uno stile *comportamentale*, espresso dal seguente codice:

```
architecture behaviour of UM is
signal sel : std_logic_vector(1 downto 0);
type table_type is array (1 to 4) of std_logic_vector(7 downto 0);
constant OUTPUT_VECTORS: table_type :=
                ("00010101", "00000101", "00010001", "00000101");

begin
   sel <= A1 & A0;
   process(sel)
   begin
      case sel is
           when "00" => --Tempo V1
                Data <= OUTPUT_VECTORS(1);
           when "01" => --Tempo G1 e G2
                Data <= OUTPUT_VECTORS(2);
           when "10" => --Tempo V2
                Data <= OUTPUT_VECTORS(3);
           when "11" => --Tempo Spento
                Data <= OUTPUT_VECTORS(4);
           when others =>
                Data <= "XXXXXXXX";
      end case;
   end process;
end behaviour;
```

Come si può vedere è stato definito un nuovo tipo di dato *table_type* come un array di quattro celle, ognuna delle quali è capace di memorizzare un vettore *std_logic* ad 8-bit. In questo modo di fatto, si è realizzata una tabella indirizzabile per righe. Attraverso la coppia di segnali *A1* e *A0* (concatenati a formare il segnale *sel* per una maggiore manegevolezza) è possibile selezionare una precisa riga e leggerne il valore memorizzato.

Si modella così il funzionamento di una *ROM*; il dato restituito è la durata temporale della fase semaforica corrente.

Si noti la presenza della clausola `others`: essa è presente nonostante, apparentemente, siano già state coperte tutte le scelte possibili. Ciò è dovuto al fatto che il segnale *sel* è di tipo *std_logic*, quindi ammette 9 possibili valori; in altre parole la clausola others va a coprire i casi (difficilmente manifestabili) in cui il valore del segnale non sia definito correttamente (ad esempio la configurazione `1Z`).

1.4 Unità di elaborazione

L'*unità di elaborazione* ha il compito di indicare all'unità di controllo, tramite l'attivazione del segnale *Overflow*, il completamento dell'intervallo di generazione della segnalazione corrente. In altre parole, essa riceve in ingresso il valore corrispondente alla durata temporale della singola fase, ne fa il countdown e al termine avvisa l'unità di controllo tramite il segnale *Overflow* (fig. 6.3).

Il circuito, modellato graficamente in stile *strutturale*, è realizzato con l'istanziazione di due tipi di simboli (contatore generico e rete di rilevamento '1'). I due contatori *U1* ed *U2* sono configurati in modo da lavorare e prendere in ingresso 4-bit ciascuno. In questo caso si ricorre alla tecnica del *bus slicing*: il bus dati `D[7..0]` è suddiviso in due bus a 4-bit, `D[7..4]` e `D[3..0]`; il primo sotto-bus contiene le cifre "decimali" dei secondi, mentre il secondo sotto-bus contiene le unità.

Il componente *U3* modella una rete di rilevamento della configurazione `"00000001"` e genera il segnale *Overflow*, la generazione di tale segnale permette (essendo collegato all'ingresso *load* di entrambi i contatori) al successivo fronte positivo del clock di caricare nei contatori il nuovo dato dalle memorie.

Il codice dell'architettura è nella pagina seguente:

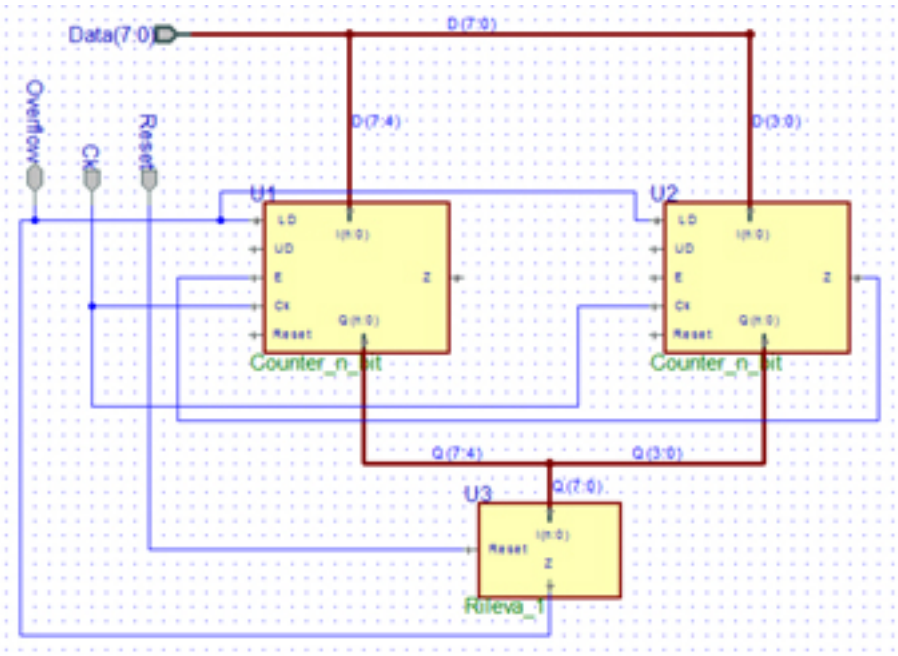


Figura 6.3: Unità di elaborazione

```
architecture behaviour of Rileva_1 is
signal value: std_logic;
signal confronto : std_logic_vector(n - 1 downto 0)
                        := (0 => '1', others => '0');
begin
   process(I, reset)
   begin
      if (reset = '1') then
         Z <= '1';
      else
         if (I = confronto) then
            Z <= '1';
         else Z <= '0';
         end if;
      end if;
   end process;
end Rileva_1;
```

La presenza del segnale di *Reset* è necessaria al fine di inizializzare correttamente entrambi i contatori, ricaricando l'ultimo valore temporale loro assegnato.

1.5 Unità di uscita

L'*unità di uscita*, infine, ha il compito di gestire (in dipendenza della segnalazione corrente), i segnali *V1*, *G1*, *R1*, *V2*, *G2* e *R2* che comandano l'accensione delle lampade per le due direttrici di marcia.

La modellazione è fatta direttamente in stile *comportamentale*, senza ricorrere all'editor grafico. Il codice architetturale che ne descrive il comportamento è il seguente:

```
architecture behaviour of UU is
begin
   P0: process(state)
   begin
      case state is
          when 0 =>
               v1 <= '0'; g1 <= '0'; r1 <= '0';
               v2 <= '0'; g2 <= '0'; r2 <= '0';
          when 1 =>
               v1 <= '1'; g1 <= '0'; r1 <= '0';
               v2 <= '0'; g2 <= '0'; r2 <= '1';
          when 2 =>
               v1 <= '0'; g1 <= '1'; r1 <= '0';
               v2 <= '0'; g2 <= '0'; r2 <= '1';
          when 3 =>
               v1 <= '0'; g1 <= '0'; r1 <= '1';
               v2 <= '1'; g2 <= '0'; r2 <= '0';
          when 4 =>
               v1 <= '0'; g1 <= '0'; r1 <= '1';
               v2 <= '0'; g2 <= '1'; r2 <= '0';
          when 5 =>
```

```
                v1 <= '0'; g1 <= '1'; r1 <= '0';
                v2 <= '0'; g2 <= '1'; r2 <= '0';
            when others =>
                v1 <= '0'; g1 <= '0'; r1 <= '0';
                v2 <= '0'; g2 <= '0'; r2 <= '0';
        end case;
    end process;
end behaviour;
```

Di fatto si opera una decodifica a partire dal segnale *state*, ossia un tipo di dato *integer*, estratto dall'unità di controllo; al solito si è utilizzata la clausola `others` per recuperare le configurazioni altrimenti non modellate.

1.6 Testbench

In questo primo esempio, visto anche lo scarso numero di segnali, si è preferito impostare i parametri della simulazione, facendo direttamente uso del tool grafico di supporto alla simulazione (si veda in proposito anche il cap. 4, sez. 2.1). L'architettura del *testbench* si limita all'istanziazione del componente *semaforo* nella sua interezza, andando così a misurare i segnali presenti in uscita sui suoi morsetti, come se ci si ponesse di fronte all'impianto installato.

La figura 6.4 mostra uno screenshot dell'andamento dei segnali in simulazione.

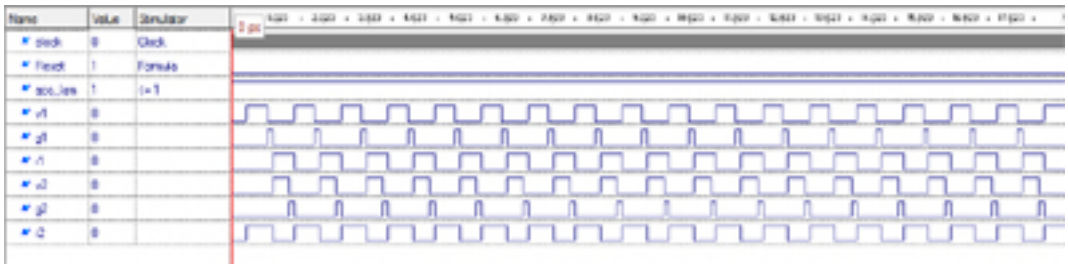


Figura 6.4: Dettaglio della simulazione

2 Game Controller

Questo secondo caso di studio modella una versione elementare di un controller per un videogame da bar. Il videogioco prevede che un oggetto mobile su una griglia di sedici celle segua ciclicamente il percorso schematizzato in fig. 6.5, senza arretrare o andare a sbattere contro le pareti.

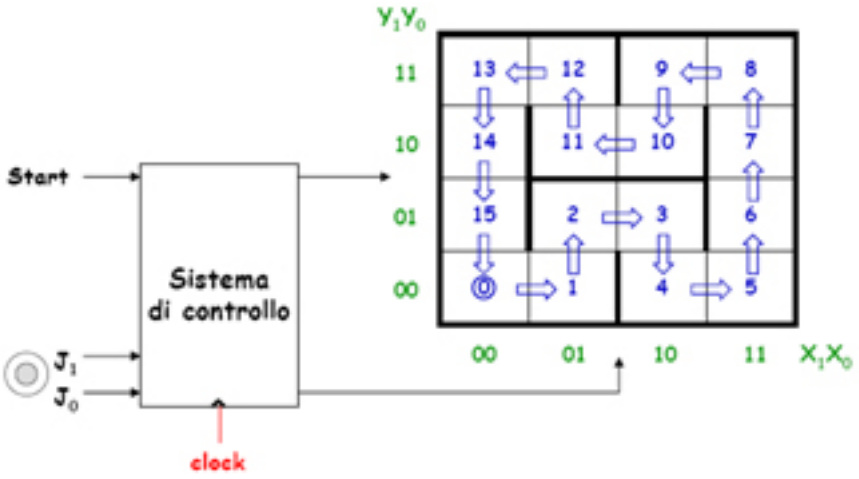


Figura 6.5: Percorso del gioco

A gioco fermo l'oggetto permane nella cella 0 di coordinate *(Y1,Y0) = (X1,X0) = (00)*. Il gioco ha inizio nell'intervallo di clock successivo a quello in cui il segnale sincrono *Start* assume il valore 1 (per ipotesi tale valore ha durata unitaria e si può presentare soltanto a gioco fermo in conseguenza dell'introduzione di un gettone).

Corrispondentemente il sistema di controllo del videogioco prende atto dei comandi di spostamento dell'oggetto che il giocatore seleziona tramite un joystick. I comandi possibili sono 4, codificati tramite 2 bit *(J1, J0)* come segue:

- (01) = spostamento a destra;
- (10) = spostamento a sinistra;
- (11) = spostamento in alto;
- (00) = spostamento in basso.

Se il comando è corretto, l'oggetto è spostato nella cella successiva del percorso; se è errato, l'oggetto è riportato nella cella 0. Il gioco prosegue da tale cella nel caso che gli errori non siano stati più di due; in caso contrario il gioco è considerato concluso.

Il sistema di controllo del videogioco deve essere strutturato, come indicato in fig. 6.6, nelle quattro unità che saranno descritte nelle sezioni seguenti:

1. unità di rilevazione di errore;
2. unità di conteggio degli errori;
3. unità di conteggio dei passi;
4. unità di transcodifica delle coordinate dell'oggetto in base al numero di passi eseguiti.

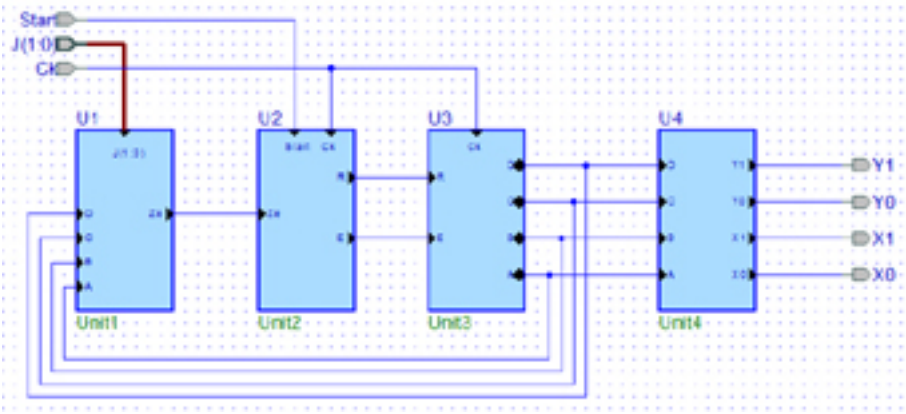


Figura 6.6: Modellazione del controller per il videogioco

2.1 Unità di rilevazione d'errore

L'*unità di rilevazione d'errore* è un'unità combinatoria a due stadi con il compito di evidenziare, tramite il segnale di uscita *Error*, la selezione da parte del giocatore di uno spostamento corretto (*Error = 0*) o errato (*Error = 1*), tenendo conto del passo corrente (indicato dai segnali di uscita *D* (MSB), *C*, *B*, *A* (LSB) dell'unità di conteggio dei passi) e del valore attuale dei segnali *J1*, *J0*.

Si è ritenuto appropriato modellare questa unità con l'editor grafico. L'idea è stata quella di disporre in cascata tre reti logiche per ottenere il comportamento richiesto: una prima rete combinatoria *RC1*, partendo dalla posizione attuale del cursore mobile, restituisce l'unica configurazione lecita che può essere ottenuta dal joystick. Quest'ultima costituisce l'ingresso di selezione di un *decoder* (connesso a valle alla RC1) che attiva così solamente una delle sue uscite. Le uscite del decoder sono negate e collegate agli ingressi dati di un *multiplexer*, i cui ingressi di selezione sono la configurazione attuale fornita dal joystick. Se il joystick ha fornito la stessa configurazione prevista dalla rete RC1, l'uscita del multiplexer sarà uno '0' (mossa corretta), viceversa sarà un '1' (mossa erronea). La rete risultante è mostrata in fig. 6.7

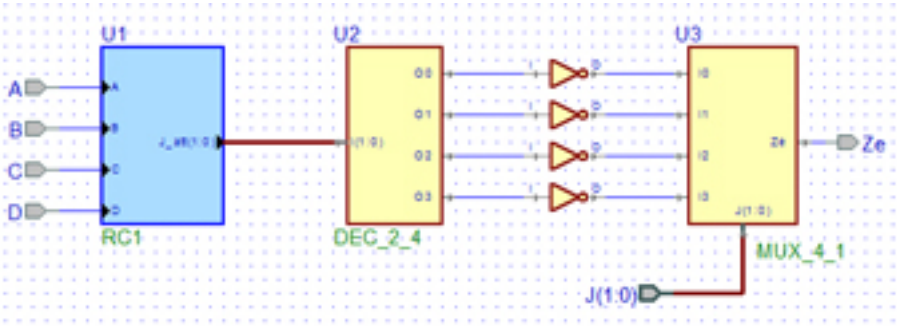


Figura 6.7: Rete interna all'unità di rilevazione d'errore

Per definire il funzionamento interno di *RC1* si è proceduto creando la *mappa di Karnaugh* per la funzione combinatoria da realizzare, quindi si sono sintetizzate manualmente le equazioni logiche:
$J0' = \overline{DB} + \overline{D}C + \overline{D}B\overline{A} + D\overline{C}BA$
$J1' = D\overline{BA} + \overline{DB}A + \overline{D}CB + D\overline{C}B$

L'architettura è stata quindi definita manualmente con l'uso di uno stile *dataflow*, ossia mediante l'uso di istruzioni di assegnazione concorrenziali.

Il *multiplexer* e il *decoder* sono stati modellati utilizzando i relativi pattern, suggeriti dal *Code Assistant* proprio dell'IDE (consistente in un repository contenente il codice per modellare i dispositivi più comuni).

2.2 Unità di conteggio degli errori

L'*unità di conteggio degli errori* ha il compito di elaborare i segnali *Start* ed *Error* in modo da indicare all'unità di conteggio (a valle), tramite i comandi *Reset* (R) ed *Enable* (E), se l'oggetto deve:

1. restare nella cella 0 in attesa dell'evento *Start = 1*;
2. ritornare nella cella 0 in caso di spostamento selezionato errato;
3. avanzare di un passo lungo il percorso in caso di spostamento selezionato corretto.

Nel caso specifico, sono state realizzate due architetture per l'unità: seguendo il processo di sviluppo si è dapprima realizzata la macchina a stati finita che ne descrivesse il comportamento, modellandola per mezzo dell'editor grafico fornito dall'IDE di sviluppo. Per evitare incomprensioni, il segnale in input *Error* (proveniente dall'unità a monte) è stato rinominato in *Ze* (fig. 6.8).

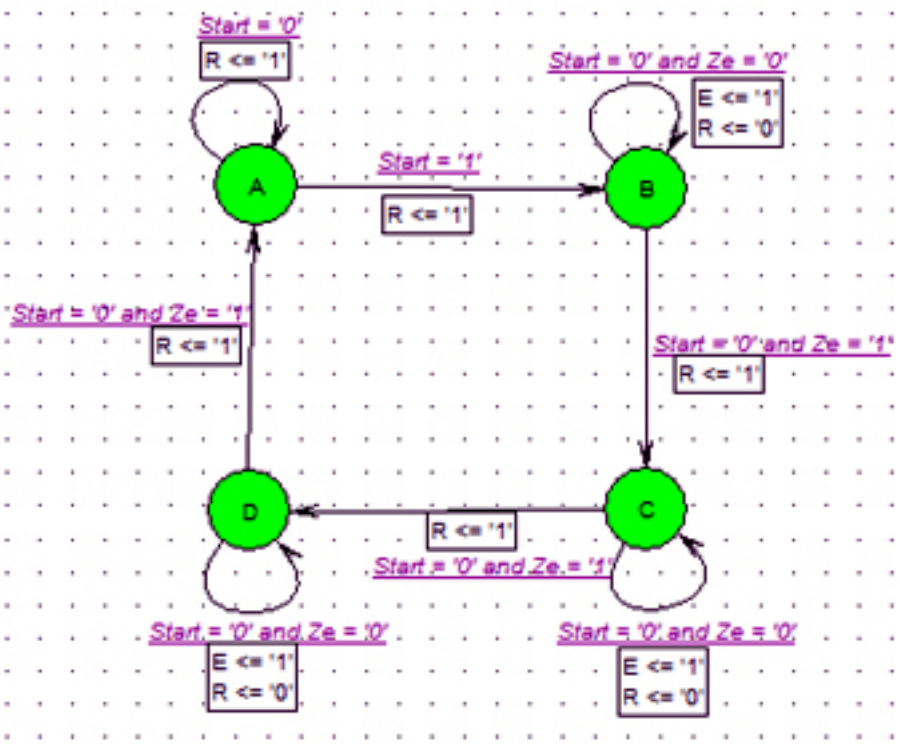


Figura 6.8: FSM per l'unità di conteggio degli errori

Quindi se ne è sviluppata (manualmente) la corrispondente versione *strutturale* modellandola con l'ambiente grafico (fig. 6.9).

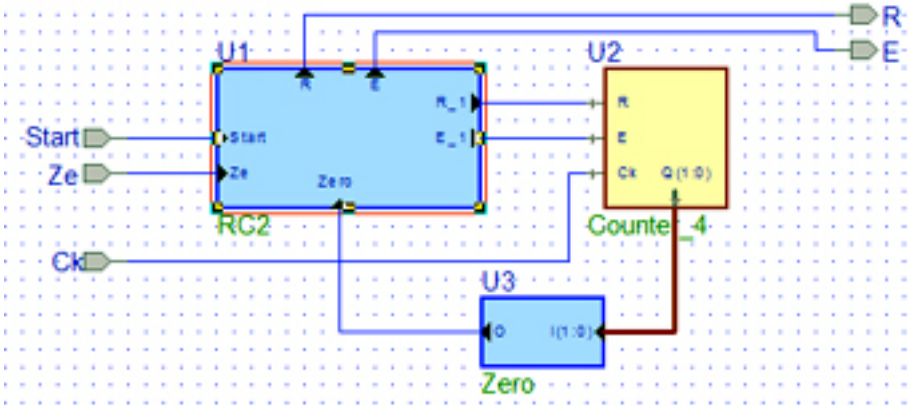


Figura 6.9: Architettura strutturale per l'unità di conteggio degli errori

Nello schema è presente un contatore e la rete combinatoria che ne fornisce i comandi, in accordo al modello *Data Path & Control Unit*; è presente inoltre anche una rete combinatoria in grado di rilevare l'uscita `"00"` del contatore 2-bit, che identifica lo stato iniziale.

La rete combinatoria *RC2*, modellata con lo stile *dataflow* contiene le seguenti assegnazioni concorrenziali:
$R_1 = Zero\overline{Start}$
$E_1 = Zero + Ze$
$R = Zero + Ze$
$E = 1$

Ovviamente, nelle fasi successive si dovrà utilizzare una sola delle due architetture, ma le prove effettuate hanno mostrato (come per altro prevedibile) che il comportamento è lo stesso.

2.3 Unità di conteggio dei passi

L'*unità di conteggio dei passi*, *U3* all'interno di questo progetto, è costituita da un contatore 4-bit, modulo 16. Il contatore riceve in ingresso i segnali di

controllo generati dall'unità di conteggio degli errori e fornisce in uscita il numero della cella futura in cui si troverà il cursore.

L'architettura è modellata in stile *strutturale* dal seguente codice (per il commento del quale, si rimanda ad un caso analogo presente nella sez. 5.2, pag. 177).

```
architecture Counter_n_bit of Counter_n_bit is
signal count_int : std_logic_vector(3 downto 0) := (others => '0');
begin
process (CK)
   begin
     if CK'event and CK= '1' then
        if R = '1' then
           count_int <= (others => '0');
        elsif E = '1' then
              count_int <= count_int + 1;
        end if;
     end if;
   end process;

   A <= count_int(0);
   B <= count_int(1);
   C <= count_int(2);
   D <= count_int(3);
end Counter_n_bit;
```

2.4 Unità di transcodifica

L'*unità di transcodifca* è costituita da una rete combinatoria con il compito di generare le coordinate dell'oggetto in base al valore fornito dal contatore, presente nell'unità a monte.

L'unità è stata modellata con un'architettura in stile *dataflow*, dopo aver sintetizzato mediante mappa di mappa di Karnaugh (fig. 6.10) le espressioni relative alla variabili delle uscite *X0*, *X1*, *Y0* ed *Y1*.

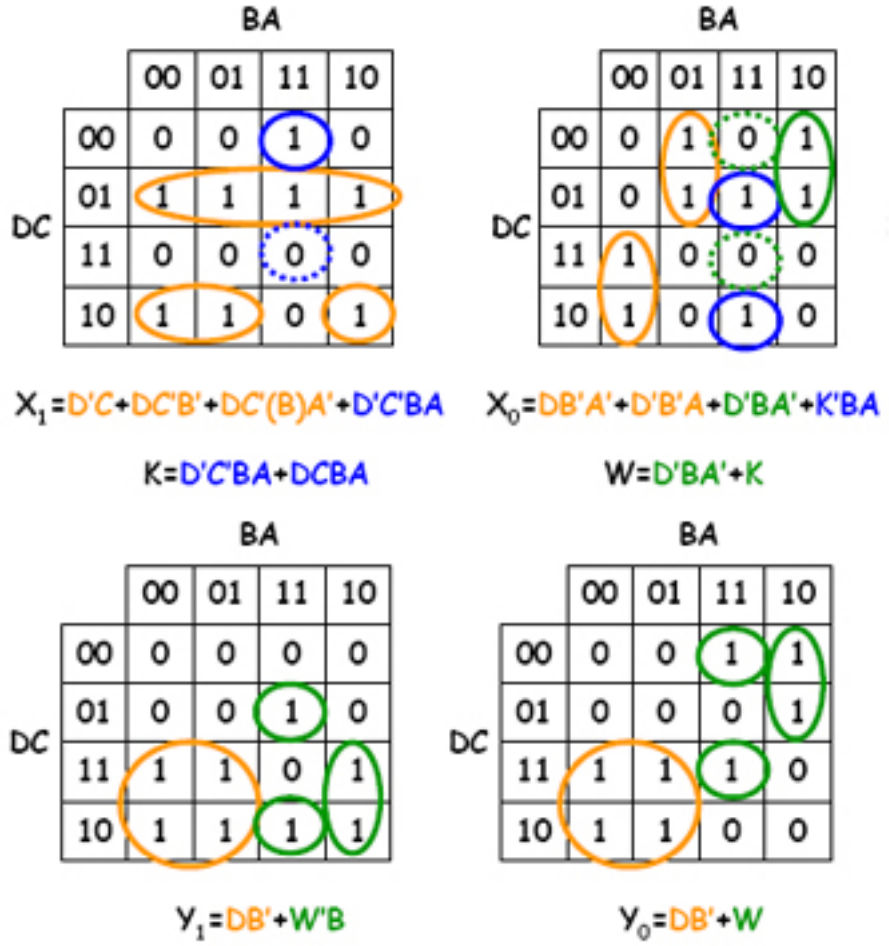


Figura 6.10: Mappe di Karnaugh per la generazione delle uscite

Le relative espressioni sono:
$X0 = (D\overline{B}\overline{A}) + (\overline{D}\overline{B}A) + (\overline{D}B\overline{A}) + (\overline{K}BA)$
$X1 = (\overline{D}C) + (D\overline{C}B) + (D\overline{C}B\overline{A}) + (\overline{D}\overline{C}BA)$
$Y0 = (D\overline{B}) + W$
$Y1 = (D\overline{B}) + \overline{W}$

$K = (\overline{D}\overline{C}BA) + (DCBA)$
$W = (\overline{D}B\overline{A}) + K$

2.5 Testbench

Il *testbench* per il progetto è stato realizzato nel rispetto delle indicazioni presentate nel cap. 5, sez. 2. In particolare sono presenti i processi di generazione del clock e della sua interruzione al raggiungimento di un preciso istante del tempo di simulazione; il processo per generare il segnale aperiodico di *Start* ed il processo per caricare da file testuale la sequenza di comandi da dare al joystick.

In realtà, sono presenti due differenti architetture (simulabili in alternativa): i processi relativi al clock e alla generazione del segnale aperiodico sono comuni, mentre la differenza risiede nel comportamento della lettura da file. Nella prima architettura si ha la sola lettura della configurazione, la relativa conversione in segnali e l'applicazione di questi ai terminali del *controller* (ovviamente il controller è istanziato come componente di alto livello). Il tutto è modellato dal seguente codice:

```
architecture tb_1 of controller_tb is
   component controller is
     port(
         Ck : in std_logic;
         Start : in std_logic;
         J : in std_logic_vector(1 downto 0);
         X0 : out std_logic;
         X1 : out std_logic;
         Y0 : out std_logic;
         Y1 : out std_logic
         );
   end component controller;

   signal Clock, Start : std_logic;
   signal J : std_logic_vector(1 downto 0);
   signal X0, X1, Y0, Y1 : std_logic;
   signal N1 : integer;
   constant period : time := 10 ns;
```

```
   for u1: controller use entity work.controller;
   file input : text open read_mode is "input_TB_A1.txt";

begin
   u1: controller
    port map(
        Ck => Clock,
        Start => Start,
        J => J,
        X0 => X0,
        X1 => X1,
        Y0 => Y0,
        Y1 => Y1
        );

   clock_p: process
   begin
     Clock <= '0';
     wait for period / 2;
     Clock <= '1';
     wait for period / 2;
     if(now > 200 ns) then
        wait;
     end if;
   end process;

   start_p: process
   begin
     Start <= '1';
     wait for 10 ns;
     Start <= '0';
     wait;
   end process;
```

```
   read_file: process
   variable buf : line;
   variable J_var : std_logic_vector(1 downto 0);
   begin
     wait for 5 ns;
     while not endfile(input) loop
       readline(input, buf);
       if buf(1) = '#' then
          next;
       end if;
       read(buf, J_var);
       J <= J_var;
       wait for 10 ns;
     end loop;
     wait;
   end process;
end tb_1;
```

Per ulteriori commenti sul processo di lettura da file, si rimanda alla spiegazione dettagliata riportata nella sez. 5.5, pag. 183.

La seconda architettura, si differenzia da quella sopra riportata, perchè all'interno del processo di acquisizione da file è presente anche una verifica sui valori prodotti in uscita. Il codice che la descrive è il seguente: (si riporta solo il processo)

```
read_file: process
   variable buf, outbuf: line;
   variable J_var : std_logic_vector(1 downto 0);
   variable X1_var, X0_var, Y1_var, Y0_var: std_logic;
   begin
      wait for 5 ns;
      while not endfile(input) loop
         readline(input, buf);
         if buf(1) = '#' then
```

```
            next;
         end if;
         read(buf, J_var);
         read(buf, Y1_var);
         read(buf, Y0_var);
         read(buf, X1_var);
         read(buf, X0_var);
         J <= J_var;
         wait for period;
         if(Y1 /= Y1_var or Y0 /= Y0_var or X1 /= X1_var or X0 /= X0_var)then
            write(outbuf, string'("Errore all'istante: "));
            write(outbuf, now);
            write(outbuf, string'(" --> Valori ottenuti: (Y1,Y0,X1,X0) "));
            write(outbuf, Y1);
            write(outbuf, string'(","));
            write(outbuf, Y0);
            write(outbuf, string'(","));
            write(outbuf, X1);
            write(outbuf, string'(","));
            write(outbuf, X0);
            write(outbuf, string'(" "));
    [...]
         end if;
      end loop;
      wait;
   end process;
```

Come si può vedere, dal file viene letta la configurazione da applicare al segnale J che modella il joystick ed inoltre sono letti i valori corrispondenti alla posizione del cursore successiva alla mossa impartita. Poichè la posizione viene aggiornata ad ogni clock, il confronto tra il valore fornito in uscita dal controller ed il valore previsto è ritardato di un periodo, attraverso l'istruzione `wait for period`.

Se i valori generati e previsti non coincidono, ad esempio perchè il giocatore

ha sbagliato mossa rispetto a quella prevista, viene scritto l'evento in un apposito file dati, attraverso le istruzioni di scrittura presenti nel processo stesso.

2.6 Visualizzazione grafica

L'utilizzo dell'editor grafico per la modellazione di un circuito, soprattutto se utilizzato per il debugging, ha suggerito la possibilità di realizzare una forma elementare di visualizzazione grafica alternativa a quella ottenibile con le forme d'onda nel simulatore.

Si è realizzata quindi una nuova unità aggiuntiva, collegabile alle uscite del macro-componente *game controller*, in grado di mostrare run-time la posizione del cursore nel labirinto, modellando così un display a matrice di led. (fig. 6.11)

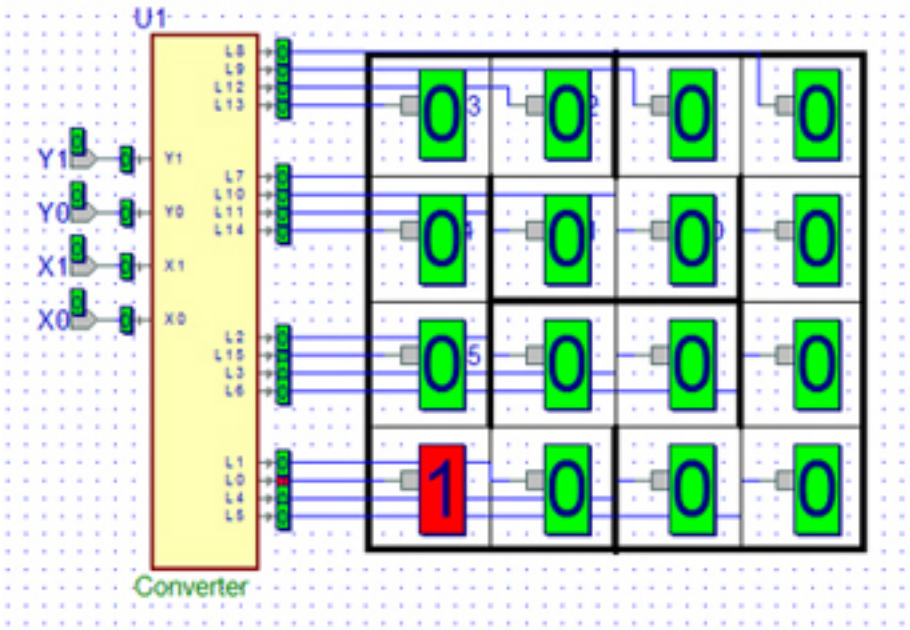


Figura 6.11: Display simulato

Per rendere l'effetto, è necessario eseguire segmenti di simulazione in sequenza, per mezzo del comando *Run for* (come per altro spiegato nel cap. 4, sez.

2.1) e con la possibilità di pilotare l'andamento dei segnali *J0* e *J1* da tastiera. Seppure lievemente macchinoso, l'effetto ottenuto è discreto, permette meglio di apprezzare il funzionamento del sistema ed aiuta nella fase di testing.

Chiaramente questa unità fittizia non modella realmente un componente fisico, non si deve dimenticare infatti che il VHDL descrive e modella componenti fisici: ad esempio un integrato che piloti un display, ma non il display stesso. Infatti, i led sono simulati utilizzando (impropriamente) i *probe*, ossia gli strumenti di debugging che permettono di visualizzare con un valore grafico lo stato di un segnale associato ad un componente o ad una connessione.

3 Sudoku Controller

Il terzo caso di studio prevede la modellazione e la simulazione di un controller da installare su un dispositivo portatile che riproduca il gioco del *Sudoku*. Il gioco consiste nel riempire una matrice di 9x9 celle, o meglio nel completarne il riempimento essendo già predefinito il contenuto di un certo numero di esse in base all'annesso livello di difficoltà, in modo tale che nelle 9 celle di ciascuna riga R_i (i = 1, 2, ..., 9), di ciascuna colonna C_j (j = 1, 2, ..., 9) e di ciascuna sottomatrice S_k (k = 1, 2, ..., 9) siano presenti tutte, e quindi ognuna una sola volta, le nove cifre decimali 1, 2, ..., 9.

Nelle versioni portatili disponibili in commercio, la console del "Sudoku Game" comprende, oltre al display della matrice di celle, alcuni tasti funzionali per mezzo dei quali il giocatore può avviare un nuovo gioco selezionandone il livello di difficoltà, definire il valore numerico da associare ad ogni cella della matrice il cui contenuto non sia già predefinito, segnalare la conclusione del gioco sottoponendo così a verifica la soluzione individuata.

Limitando il progetto agli aspetti realizzativi connessi con quest'ultima funzionalità, il (sotto)sistema di controllo può essere strutturato, in accordo al modello "Data-Path & Control Unit", come indicato in fig. 6.12.

Definendo le seguenti componenti:

- unità di controllo;
- unità di elaborazione;
- unità di memoria.

3.1 Unità di controllo

L'unità di controllo *UC*, avvalendosi delle risorse previste a livello di Data-Path e coordinandone opportunamente il funzionamento, ha il compito di gestire il processo di verifica della correttezza di una soluzione del gioco ogni qual volta viene dall'esterno attivato (livello logico 1, durata unitaria) il segnale di ingresso *Start*.

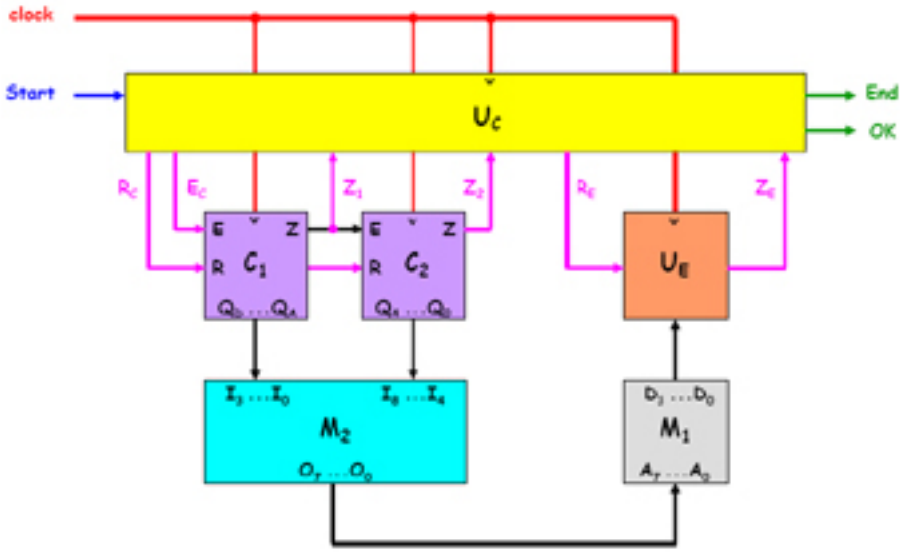


Figura 6.12: Modello Data-Path & Control Unit per il controller

Completata la verifica, UC deve prontamente notificarne in uscita l'esito tramite il segnale *OK* (OK = 1 in caso di esito positivo, OK = 0 in caso contrario), contestualmente attivando (livello logico 1, durata unitaria) il segnale di uscita *End*.

Il processo di verifica consiste nell'esaminare in sequenza i 27 sottoinsiemi di 9 celle della matrice (le 9 righe, le 9 colonne, le 9 sottomatrici), al fine di accertare se il relativo contenuto viola o meno le regole del gioco (nel primo caso è chiaramente del tutto inutile estendere l'indagine ai successivi sottoinsiemi). L'ordine secondo cui procedere nella scansione dei sottoinsiemi è R1, R2, ..., R9, C1, C2, ..., C9, S1, S2, ..., S9, mentre quello delle celle nell'ambito di un sottoinsieme è evidenziato in fig. 6.13.

L'unità è stata modellata in VHDL attraverso una macchina a stati finita, disegnata facendo uso dell'editor grafico presente nell'IDE. A tal proposito, si sono definiti dapprima i segnali di ingresso e di uscita alla rete, quindi si è proceduto con il tracciamento degli stati e delle transizioni tra essi (fig. 6.14).

La macchina a stati finiti che definisce il comportamento della rete è modellata secondo *Mealy*; ovviamente le condizioni sulle transizioni e le uscite sono

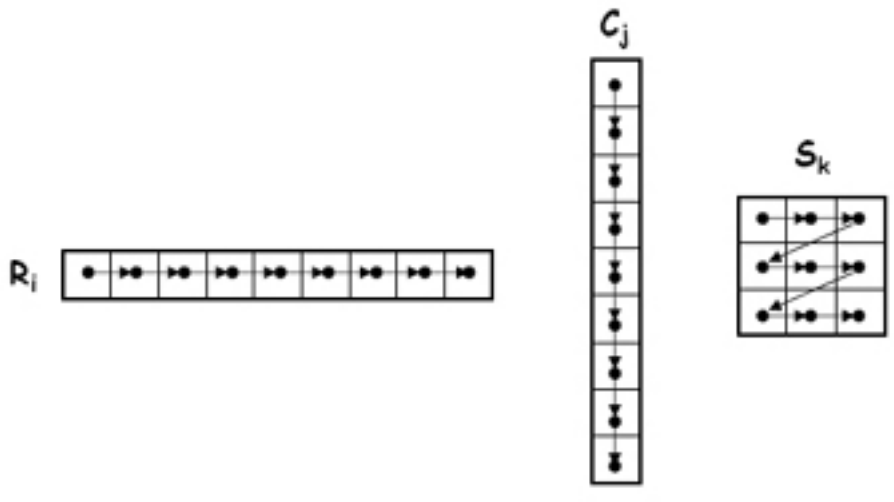


Figura 6.13: Il processo di verifica

codificate secondo la sintassi propria del linguaggio.

3.2 Unità di elaborazione

Il compito di evidenziare, una volta completata la scansione delle 9 celle di un sottoinsieme, se esse contengano valori numerici distinti è affidato all'*unità di elaborazione UE* (segnale di uscita `ZE = 1`). Il ruolo di discriminare quale sia, in ogni intervallo di clock, il particolare sottoinsieme oggetto di indagine, e nell'ambito di esso la specifica cella, è affidato ai due contatori binari *C1* (base di conteggio 9) e *C2* (base di conteggio 27), dotati di segnale di reset sincrono ed opportunamente disposti in cascata. La modellazione degli stessi, come oggetti generici, e l'istanziazione è realizzata allo stesso modo di quanto esaminato negli esempi precedenti.

Più precisamente, le variabili di stato di *C2* (`Q4`$_{(MSB)}$, `Q3`, `Q2`, `Q1`, `Q0` $_{(LSB)}$) identificano il sottoinsieme di celle (il 1° - ovvero *R1* - allorché `Q4 Q3 Q2 Q1 Q0 = 00000`, ..., il 27° - ovvero *S9* - allorché `Q4 Q3 Q2 Q1 Q0 = 11010`), mentre quelle di *C1* (`QD`$_{(MSB)}$, `QC`, `QB`, `QA`$_{(LSB)}$) la specifica cella (la 1a allorché `QD QC QB QA = 0000`, ..., la 9a allorché `QD QC QB QA = 1000`).

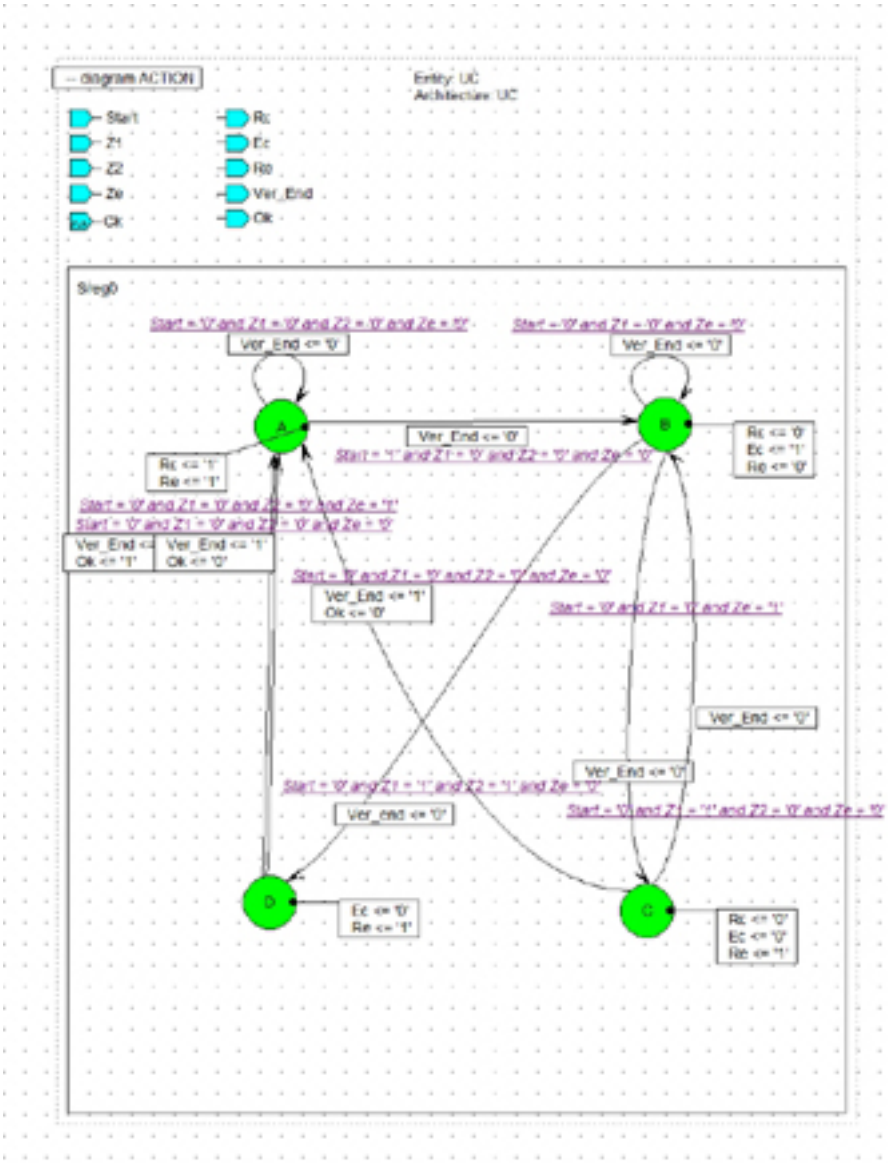


Figura 6.14: FSM relativa all'unità di controllo

Di fatto alla unità di elaborazione arrivano in ingresso, uno dopo l'altro, i 9 valori presenti nella singola cella e la rete al suo interno deve verificare che questi siano tutti diversi tra loro. L'implementazione è stata realizzata con uno stile *strutturale* per mezzo dell'editor grafico. Il risultato della modellazione è in fig. 6.15.

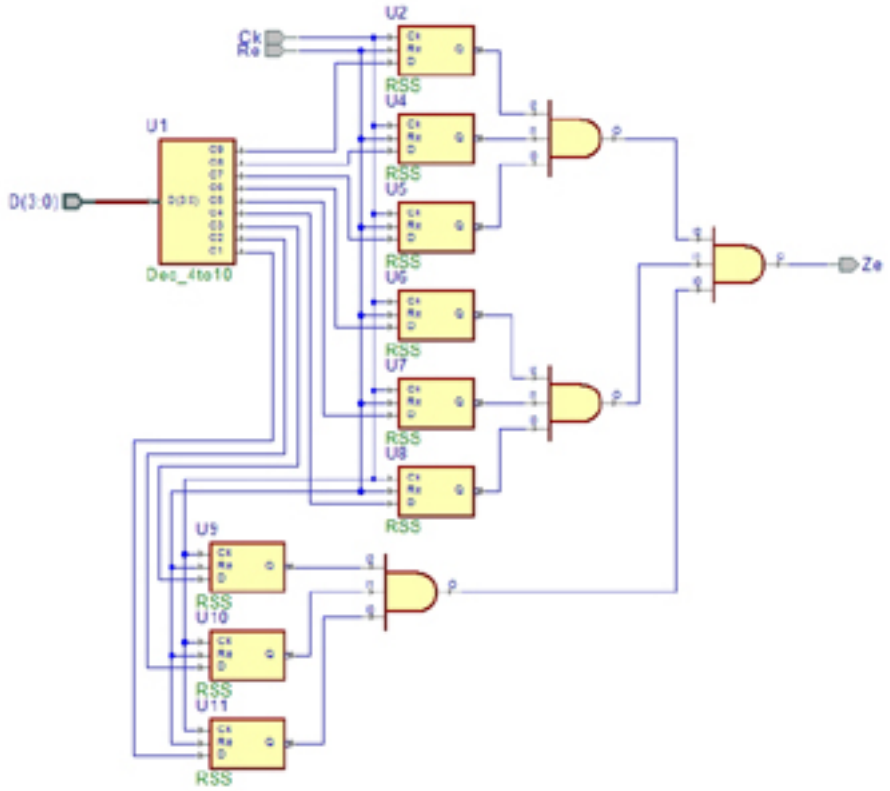


Figura 6.15: Schema a blocchi dell'unità di elaborazione

Come si può vedere, è presente un decoder 4-to-10, pilotato dai segnali `D[3..0]`: ad ogni passo della verifica, la memoria *M1* emette un valore, questo seleziona la corrispondente uscita del decoder che viene settata ad 1, ciò significa che quel valore è presente almeno una volta nella cella attualmente in corso di verifica. Ciascuna uscita del decoder è memorizzata da una RSS. L'unità emetterà il segnale di uscita *ZE = 1* se tutte le RSS hanno memorizzato il valore 1 nell'ambito dei 9 passi: ciò significa che tutti i numeri si sono presentati almeno una volta nella cella (quindi si sono presentati una e una sola volta). Viceversa, se nella cella fossero presenti due numeri uguali, dopo i 9 passi di conteggio, una delle RSS avrebbe la sua uscita non settata. Il controllo che tutte le RSS siano settate ad 1 è effettuato con una serie di *AND* a cascata, sul lato destro della figura 6.15.

Chiaramente, il valore del segnale *ZE* sarà significativo solo al termine del conteggio di 9 passi, eseguito dal contatore *C1*; sarà compito dell'unità di

controllo considerare il valore di questo segnale quando significativo.

La *RSS* è stata modellata come simbolo, in quanto istanziata più volte all'interno dell'architettura dell'unità di elaborazione, ed è costituita dalla rete (in stile *strutturale*) modellata in fig. 6.16.

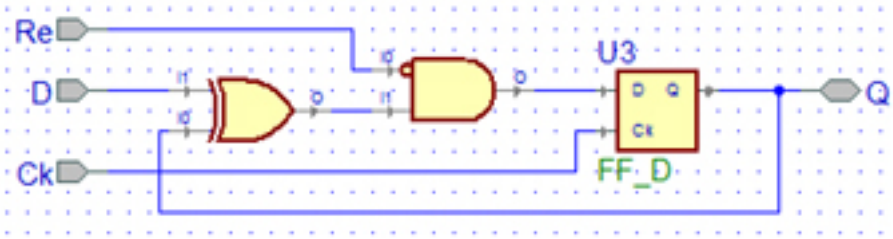


Figura 6.16: RSS per la memorizzazione dei singoli elementi individuati

I componenti utilizzati all'interno sono predefiniti dal linguaggio (*porta XOR* e *porta AND*) o modellati (*FF_D*) utilizzando il codice proposto dal *code assistant* (di cui si era parlato a pag. 144).

3.3 Unità di memoria

Per ipotesi, i valori numerici associati alle 81 celle della matrice, ciascuno dei quali è rappresentato secondo il codice *BCD*, sono disponibili nell'unità di memoria (del tipo *RAM*) *M1*.

Tale memoria, contraddistinta da 8 bit di ingresso `A7`$_{(MSB)}$, `A6`, `A5`, `A4`, `A3`, `A2`, `A1`, `A0`$_{(LSB)}$ e da 4 bit di (ingresso-)uscita `D3`$_{(MSB)}$, `D2`, `D1`, `D0`$_{(LSB)}$, fornisce in uscita il valore numerico associato alla cella corrispondente alla riga R_i e alla colonna C_j (i, j = 1, 2, ..., 9) a fronte della presentazione in ingresso dell'indirizzo `A =` A_{MSD} A_{LSD} `= (ij)`$_{BCD}$, essendo A_{MSD} `= A7 A6 A5 A4` e A_{LSD} `= A3 A2 A1 A0` (l'indirizzo `A7 A6 A5 A4 A3 A2 A1 A0 = 00010001` seleziona il contenuto della prima cella (in alto a sinistra) della matrice, ..., l'indirizzo `A7 A6 A5 A4 A3 A2 A1 A0 = 10011001` quello dell'ultima cella (in basso a destra)).

L'unità *M1* è stata modellata con un'architettura in stile *comportamentale*, di cui si riportano le parti più importanti:

```
architecture behaviour of M1 is
   signal i : std_logic_vector(3 downto 0);
   signal j : std_logic_vector(3 downto 0);
   type row is array (1 to 9) of std_logic_vector(3 downto 0);
   type memoria is array (1 to 9) of row;
   constant matrice1: memoria := (
   ("0110", "0010", "0101", "0100", "0001", "1001", "0011", "1000","0111"),
   ("1000", "0100", "0011", "0101", "0111", "0110", "1001", "0010","0001"),
   ("1001", "0001", "0111", "1000", "0010", "0011", "0101", "0110","0100"),
   ("0100", "1000", "0110", "1001", "0101", "0010", "0111", "0001","0011"),
   ("0111", "0101", "0001", "0110", "0011", "0100", "0010", "1001","1000"),
   ("0010", "0011", "1001", "0001", "1000", "0111", "0110", "0100","0101"),
   ("0001", "0111", "1000", "0010", "0110", "0101", "0100", "0011","1001"),
   ("0101", "0110", "0100", "0011", "1001", "1000", "0001", "0111","0010"),
   ("0011", "1001", "0010", "0111", "0100", "0001", "1000", "0101","0110")
   );

begin
   i <= A(7 downto 4);
   j <= A(3 downto 0);
   process(i, j)
     variable riga_estrapolata: row;
     begin
       case i is
          when "0001" =>
               riga_estrapolata := matrice1(1);
               case j is
                    when "0001" =>
                        D <= riga_estrapolata(1);
                    when "0010" =>
                        D <= riga_estrapolata(2);
                    when "0011" =>
```

```
                    D <= riga_estrapolata(3);
                when "0100" =>
                    D <= riga_estrapolata(4);
                when "0101" =>
                    D <= riga_estrapolata(5);
                when "0110" =>
                    D <= riga_estrapolata(6);
                when "0111" =>
                    D <= riga_estrapolata(7);
                when "1000" =>
                    D <= riga_estrapolata(8);
                when "1001" =>
                    D <= riga_estrapolata(9);
                when others =>
                D <= "XXXX";
            end case;

        [...]

        when others =>
          D <= "XXXX";
      end case;
   end process;
end architecture behaviour;
```

Di fatto, è stato dapprima definito il tipo *row*, corrispondente alla singola riga della matrice (riga costituita da 9 celle, dalla 1 alla 9, ciascuna delle quali di 4 bit). Quindi è stato definito il tipo *memoria*, corrispondente ad una tabella di 9 righe, ciascuna contenente 9 row. Infine è stata definita una costante *matrice1*, di tipo memoria, riempita con i valori inseriti nelle celle del sudoku da valutare. La fase di inserimento dei valori nelle celle del sudoku stesso (e quindi nella memoria) esula dal caso di studio e per questa ragione non è stata modellata.

L'architettura si compone di due istruzioni concorrenti, in grado di operare un *bus slicing*, per separare i bit relativi all'indirizzo della riga da quelli della colonna e da un processo, nel quale si ha l'effettiva estrazione estrazione del dato dalla memoria, in relazione alla cella selezionata.

I segnali i e j sono generati con le istruzioni concorrenti, fuori dal processo in modo da essere subito disponibili al processo stesso (si veda anche cap. 2, sez 3.1 e quanto detto nell'esempio a pag. 177).

Il processo, attivato in caso di variazione di uno dei segnali di indirizzamento, è costituito da un'istruzione `case` esterna in grado di selezionare ed estrapolare in una variabile temporanea (quindi il cui valore è subito utilizzabile) una delle 9 linee della matrice, ad es. `riga_estrapolata := matrice1(1);` . Oltre a questa operazione è presente anche un'altra istruzione `case` annidata, con il compito di estrarre dalla linea uno dei 9 valori in essa contenuti, il dato così estratto viene assegnato all'uscita della memoria.

Nell'istruzione di case esterno sono ovviamente riportati tutti i bracci alternativi, qui per esigenze di spazio si è riportato solo quello equivalente alla prima linea della matrice.

L'unità di memoria *M2* (del tipo *ROM*), contraddistinta da 9 bit di ingresso $\texttt{I8}_{(MSB)}$, `I7`, `I6`, `I5`, `I4`, `I3`, `I2`, `I1`, $\texttt{I0}_{(LSB)}$, nell'ordine connessi a `Q4 Q3 Q2 Q1 Q0 QD QC QB QA`, e da 8 bit di uscita $\texttt{O7}_{(MSB)}$, `O6`, `O5`, `O4`, `O3`, `O2`, `O1`, $\texttt{O0}_{(LSB)}$, nell'ordine connessi a `A7 A6 A5 A4 A3 A2 A1 A0`, è quindi prevista allo scopo di generare l'indirizzo della cella oggetto di indagine in ogni intervallo di clock a partire dalla corrispondente modalità di identificazione adottata nell'ambito dell'unità di conteggio.

3.4 Testbench

Il testbench si prefigge l'obiettivo di testare il componente *sudoku*, ossia il controller, mostrandone il comportamento ai morsetti. La modellazione del testbench segue i pattern presentati nel cap. 5 e nello specifico (all'interno della sua architettura) sono presenti i processi di generazione del clock e del

segnale aperiodico di *Start*.

L'architettura del testbench è la seguente:

```
architecture tb of sudoku_tb is
   signal Ck, Ok, Ver_End : std_logic;
   signal Start : std_logic := '0';
   constant period : time := 10 ns;

   component sudoku
       port(
           Ck : in std_logic;
           Start : in std_logic := '0';
           Ok : out std_logic;
           Ver_End : out std_logic
           );
    end component sudoku;

begin
   clock_p: process
   begin
     Ck <= '0';
     wait for period / 2;
     Ck <= '1';
     wait for period / 2;
     if Ver_End = '1' then
        wait;
     end if;
   end process;

   Start_p: process
   begin
     Start <= '1';
     wait for 10 ns;
     Start <= '0';
```

```
      wait;
   end process;

   U1: sudoku
   port map(
      Ck => Ck,
      Start => Start,
      Ok => Ok,
      Ver_End => Ver_End
      );
end architecture tb;
```

Come si può vedere, il componente (ad alto livello) sudoku viene dichiarato ed istanziato. Contrariamente agli altri casi di studio, in questo non sono generati i valori corrispondenti ai segnali presenti nella griglia del sudoku. Questi ultimi sono stati codificati direttamente nell'architettura della memoria *M2*: la ragione di ciò è che si vuol concentrare l'attenzione sul processo di verifica, piuttosto che sull'inserimento dei valori.

Nelle fasi di testing, si sono realizzate diverse architetture, con diverse permutazioni dei valori nella griglia, per valutare il comportamento del sistema in occasione di errori nell'inserimento dei valori stessi nel gioco. Chiaramente, in occasione di tali errori la verifica si interrompeva in una delle tre fasi di scansione della matrice (per righe, per colonne, per singola cella).

4 Pig Game

Il caso di studio esaminato in questa sezione prevede la realizzazione di un controllore digitale che gestisca il corretto funzionamento di una consolle per il gioco del PIG.

Il *gioco del "PIG"*, utilizzato nelle scuole elementari per consentire agli alunni di acquisire una intuitiva percezione del concetto di probabilità, prevede che ciascuno degli N (N≥2) giocatori partecipanti ad un incontro, a rotazione, effettui il lancio di un dado anche più volte consecutivamente, accumulando via via un punteggio parziale uguale alla somma dei valori corrispondentemente esibiti dal dado. Tale punteggio parziale è tuttavia prontamente azzerato ed il turno del giocatore ritenuto concluso senza alcun incremento del punteggio totale conseguito negli eventuali precedenti turni, se a seguito di un lancio il dado esibisce il valore 1.

Al fine di prevenire questo indesiderabile evento, tanto più probabile quanto maggiore è il numero di lanci effettuati, un giocatore può decidere di capitalizzare il punteggio parziale fino al momento conseguito nell'ambito di un turno passando spontaneamente la mano al giocatore successivo. Vince l'incontro il giocatore che per primo raggiunge un punteggio complessivo ≥ 100.

Con riferimento al caso di N = 2 giocatori, la console del "PIG", evidenziata in fig. 6.17, comprende:

- due display numerici a 7-segmenti (*Total0*, *Total1*), preposti alla visualizzazione del punteggio totale conseguito nei precedenti turni da ciascun giocatore;
- due led (*Player0*, *Player1*), adibiti ad indicare in maniera mutuamente esclusiva quale sia il giocatore correntemente di turno (*Player*);
- tre pulsanti (*Roll*, *Hold*, *NewGame*), tramite i quali il *Player* può, rispettivamente, richiedere il lancio del dado, notificare la propria decisione di passare la mano ritenendosi soddisfatto del punteggio parziale accumulato nel turno corrente, avviare un nuovo incontro in caso di vincita;

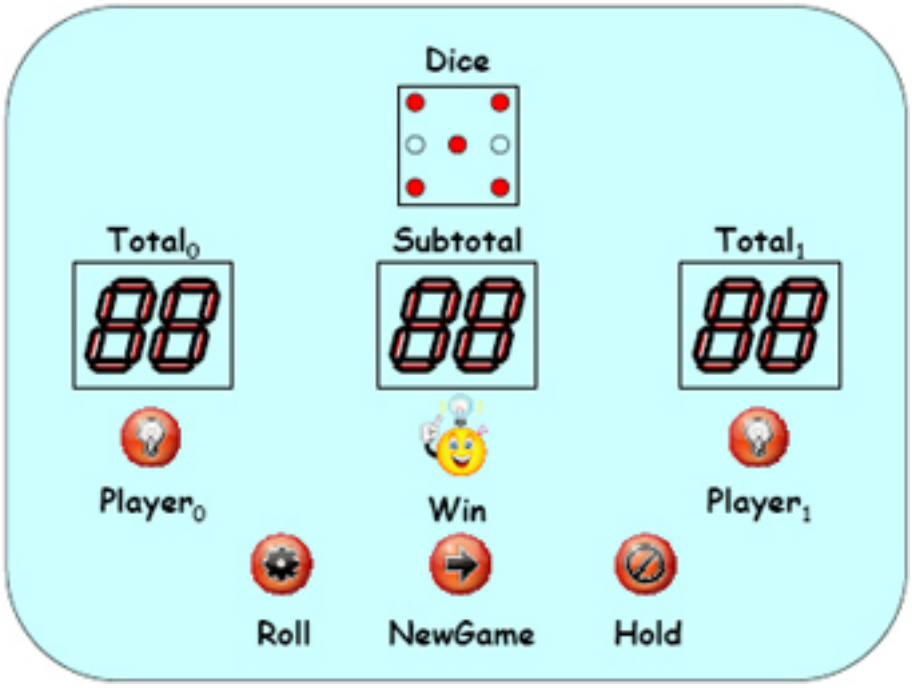


Figura 6.17: Consolle del "Pig Game"

- un display a matrice di punti (*Dice*), preposto alla visualizzazione del punteggio conseguito dal *Player* a seguito del lancio del dado;
- un display numerico a 7-segmenti (*Subtotal*), preposto alla visualizzazione del punteggio parziale correntemente accumulato dal *Player* nell'ambito del turno;
- un led (*Win*), tramite il quale viene segnalata al *Player*, una volta raggiunto un punteggio complessivo ≥ 100, la vincita dell'incontro.

Sia all'accensione che a seguito della pressione del pulsante *NewGame* al termine di un incontro, il sistema, dopo aver selezionato quale primo player il giocatore successivo a quello che ha vinto l'incontro precedente (all'accensione la scelta è operata in maniera casuale), procede ad azzerare tutti i totalizzatori dei punteggi ed a spegnere i dispositivi di visualizzazione *Win* e *Dice*.

Prontamente quindi, o al rilascio del pulsante *NewGame* in caso di avvio di un nuovo incontro, il sistema si pone in attesa della richiesta da parte del player di operare il lancio del dado. A seguito della pressione del pulsante

Roll, il sistema genera un numero casuale compreso nel range `[1..6]`, campionando lo stato di un contatore free-running e visualizzandone il valore in *Dice*. Al rilascio del pulsante *Roll*, se il valore del dado è 1, il gioco, previo azzeramento di *Subtotal*, passa nelle mani del successivo giocatore; in caso contrario il sistema, accumulato il valore del dado in *Subtotal*, si pone in attesa di una richiesta di *Hold* o di *Roll* da parte del player.

A fronte della richiesta di *Hold*, il sistema somma al punteggio totale conseguito dal player nei precedenti turni il punteggio parziale accumulato in *Subtotal* nel turno corrente, quindi opera la commutazione del player; a fronte della richiesta di *Roll*, il sistema procede alla generazione di un nuovo numero casuale, reiterando poi la sequenza. Il player è dichiarato vincitore non appena il suo punteggio complessivo raggiunge o eccede la soglia prestabilita.

4.1 Data-path & Control Unit

Il progetto di una rete sequenziale sincrona, strutturato in accordo al modello *data-path & control unit*, può essere modellato come in fig. 6.18.

Come si può vedere, sono presenti diversi simboli con funzioni differenti e il blocco funzionale (FUB) dell'unità di controllo (*UC*). Partendo da sinistra, si trova un contatore generico *U1* (nel codice - fase di istanziazione del componente nell'architettura - modellato con 3-bit per i segnali di ingresso-uscita e 6 stati interni di conteggio) *"free flow"*, ossia con l'ingresso *Enable* fisso a 1, la cui uscita è collegata ad un registro a 3-bit (*U2*) che ne congela il valore se opportunamente comandato dall'unità di controllo (pressione del pulsante *Roll*).

L'unità *U3* è multiplexer a 4 canali che consente di selezionare quale sia il secondo ingresso ad alimentare il sommatore U15, il pilotaggio è operato dall'unità di controllo. Il sommatore *U15* ha la funzione di sommare al punteggio temporaneo attuale o il nuovo valore ottenuto dal campionamento del dado o il precedente valore depositato per aggiornarlo in caso di pressione del pulsante *Hold* (a seconda dei dati inviati attraverso il multiplexer). La modellazione di tale unità è fatta con una coppia entità-architettura in un

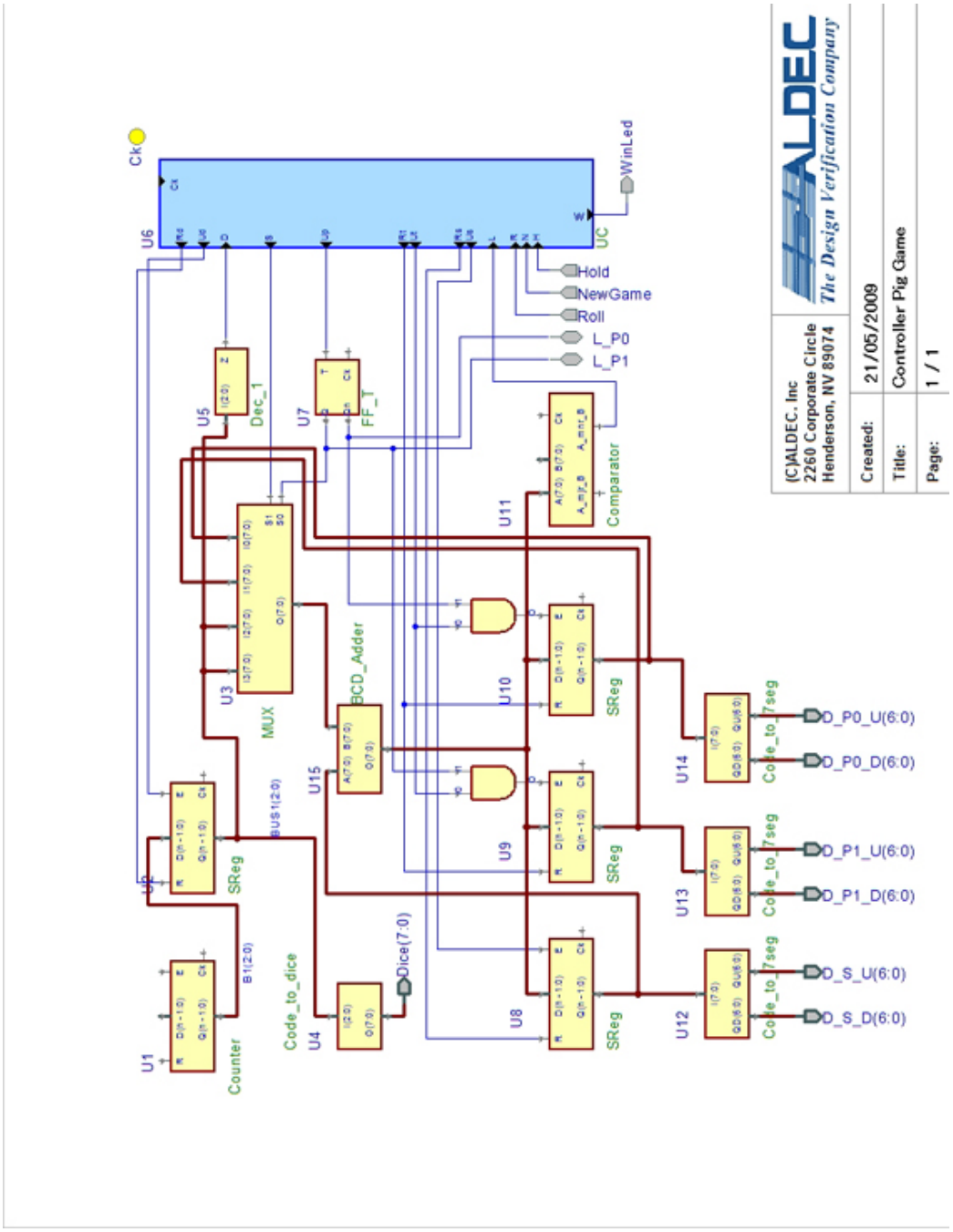


Figura 6.18: RSS per il controllo del gioco

file VHDL separato.

Il comparatore *U11* serve a verificare che il punteggio attuale (componente incamerata e componente temporanea) ottenuto da uno dei due giocatori sia ≤ 100, comunicandolo all'unità di controllo mediante il segnale *L*. Il secondo termine di confronto non è riportato graficamente, ma è stato modellato nel codice dell'architettura in fase di istanziazione del componente, esso è la codifica binaria del valore 100.

Le unità *U8*, *U9* e *U10* sono dei registri, pilotati dall'unità di controllo, con il compito di memorizzare i valori attuali rispettivamente del punteggio temporaneo, ottenuto dal giocatore 2 (lett. Player1) e ottenuto dal giocatore 1 (lett. Player0).

L'unità *U5* ha il compito di identificare e segnalare all'unità di controllo, per mezzo del segnale *D*, l'ottenimento del valore 1 dal campionamento del dado. Il flip-flop T (*U7*) ha il compito di selezionare di volta in volta quale sia il player corrente e di abilitrae quindi il campionamento sui relativi registri di memorizzazione del risultato.

Nel progetto sono state modellate anche opportune unità di transcodifica per fornire all'utente una rappresentazione dei valori dei punteggi più familiare rispetto al codice binario. L'unità *U4* si occupa di convertire i 3-bit in ingresso (valore campionato del dado) in una rappresentazione composta da sei led, disposti a riprodurre la faccia di un dado da gioco. Le unità *U12*, *U13* ed *U14* permettono la rappresentazione dei relativi punteggi su display a 7 segmenti (nel formato cifra decine, cifra unità) in notazione decimale.

Sono inoltre presenti i terminali di ingresso e uscita da collegare ai pulsanti ed ai led previsti dalle specifiche. Si noti infine la presenza del simbolo *Ck* in alto a sinistra vicino ad un cerchietto giallo: tale simbolo grafico permette di definire un segnale come se fosse idealmente collegato ad ogni ingresso 'Ck' presente sul componente. In altre parole, anzichè collegare ogni componente che utilizzi il segnale di clock con una connessione al relativo terminale,

è possibile definire graficamente il segnale come *Global Wire* affinchè sia il compilatore in fase di generazione del codice ad interpretarlo come collegato ad ogni componente con un piedino che lo richiami. Ciò (può essere ovviamente fatto per ogni tipologia di segnale) semplifica nettamente la leggibilità del circuito.

4.2 Unità di controllo

L'*unità di controllo* è stata modellata con una macchina a stati finiti, inserita nel progetto per mezzo dell'editor grafico di modellazione. A tal proposito, si sono definiti dapprima i segnali di ingresso e di uscita alla rete, quindi si è proceduto con il tracciamento degli stati e delle transizioni tra essi (fig. 6.19).
La macchina a stati finiti che definisce il comportamento della rete è modellata secondo *Mealy*; ovviamente le condizioni sulle transizioni e le uscite sono codificate secondo la sintassi propria del linguaggio. Poichè l'uso della stessa arricchisce lo schema e ne satura lo spazio diminuendo la leggibilità in fase di stampa, se ne riporta anche la versione concettuale (fig. 6.20).

4.3 Testbench

Il testbench si prefigge l'obiettivo di testare il componente *controller*, mostrandone il comportamento ai morsetti. La modellazione del testbench segue i pattern presentati nel cap. 5 e nello specifico (all'interno della sua architettura) sono presenti i processi di generazione del clock e degli altri segnali.

Di fronte ad un componente del genere, il testing può essere condotto con approcci differenti. Una prima modalità consiste nel creare all'interno dell'architettura due processi, concorrenti tra loro, che settino il valore dei segnali *Roll* ed *Hold* ad intervalli di tempo definiti (ad esempio mediante lettura da un file dati o mediante codifica degli stessi nel codice), in modo da simulare l'andamento di una partita tra due giocatori.

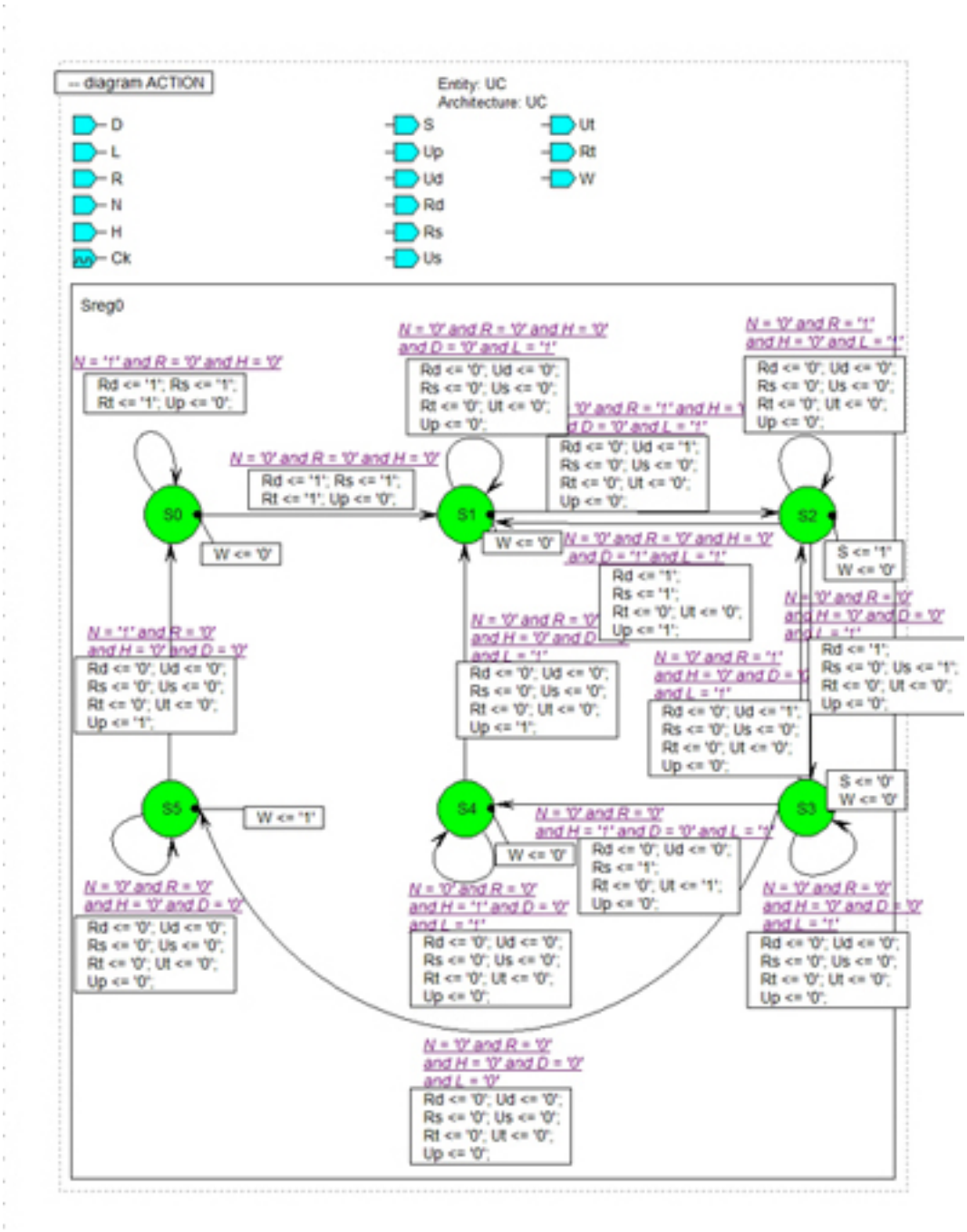


Figura 6.19: FSM relativa all'unità di controllo

Ovviamente questa modalità, benchè lecita, è abbastanza tediosa (soprattutto se si vogliono simulare partite lunghe) e richiede calcoli manuali per derivare le tempistiche, da farsi prima di eseguire la simulazione.

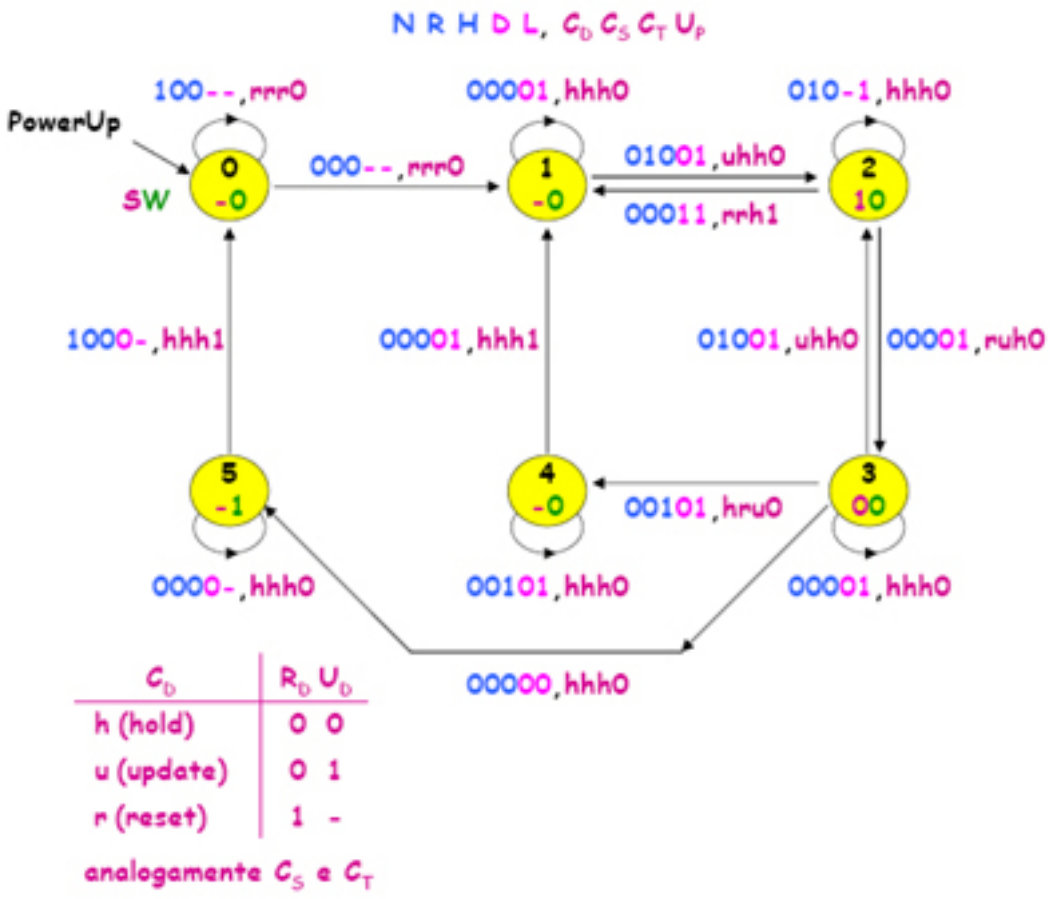


Figura 6.20: FSM (concettuale) relativa all'unità di controllo

Un esempio di simulazione è riportato nelle forme d'onda della fig. 6.21.

Una soluzione decisamente più pratica è testare il sistema con la simulazione passo-passo, variando il valore dei segnali coinvolti direttamente da tastiera, ciò è illustrato nella sottosezione seguente.

4.4 Visualizzazione grafica

Come già descritto nel cap. 4, sez. 2.1, la simulazione passo-passo è possibile lanciando il comando *Run for*, specificando un'opportuna durata temporale. In questa modalità, è possibile assegnare, tramite l'editor grafico del simu-

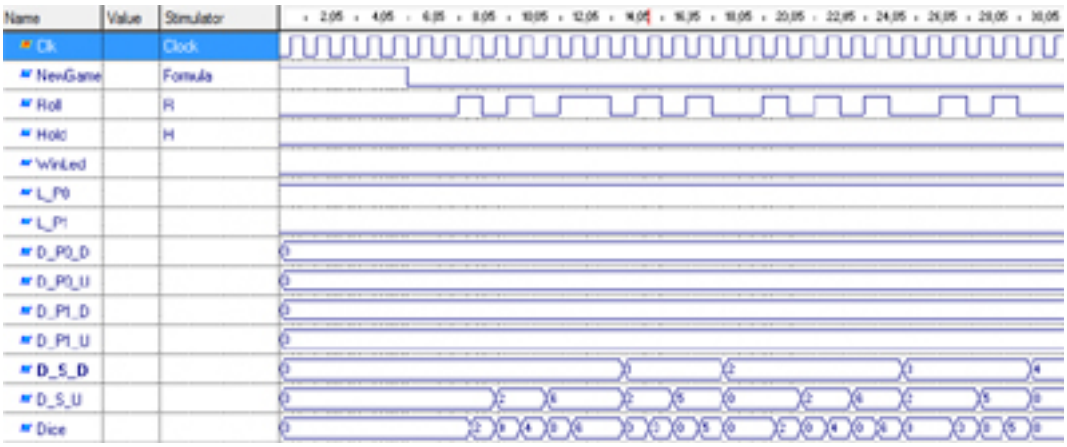


Figura 6.21: Forme d'onda per il controller (particolare)

latore, una combinazione di tasti per variare (in modalità *toggle*) il valore attuale di un segnale in input al sistema. In altre parole è possibile far evolvere la partita (quasi) come se si stesse realmente operando sull'unità.

Per rendere più semplice la visualizzazione dei risultati, e quindi capire se il sistema sta operando correttamente, si è pensato di sfruttare la modalità di debug associata all'editor grafico con cui si è modellato il componente nella sua interezza: mediante l'utilizzo di *probe* sui segnali uscenti dalle varie componenti, è stato possibile disegnare un'interfaccia grafica basilare che simulasse i led e i display a 7 segmenti, presenti nell'equivalente fisico.
Il risultato è mostrato nella fig. 6.22.

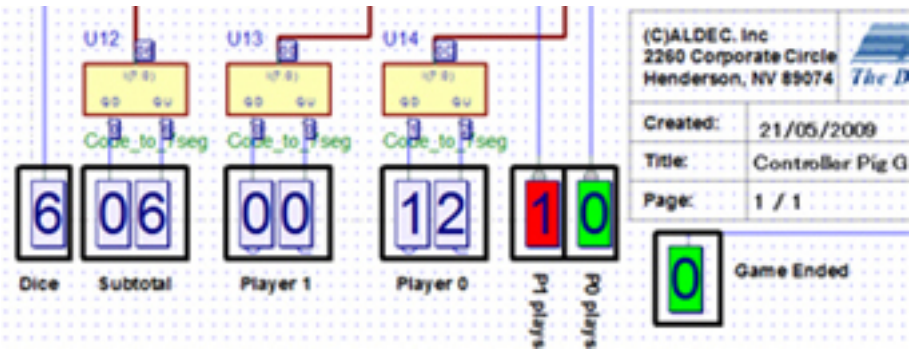


Figura 6.22: La console di comando (particolare)

5 Verificatore

Il caso di studio esaminato in questa sezione è ispirato ad un problema reale. Si è realizzato un semplice sistema di correzione automatica per gli esami del corso di *"Reti Logiche"* partendo da requisiti precisi.

Il docente modella il comportamento della rete sincrona, definita dalle specifiche, attraverso il grafo degli stati (ad esempio): questa rappresentazione - supposta, ovviamente, corretta - costituisce il *Golden Model*, ossia il modello di riferimento. Lo studente a questo punto realizzerà la propria rete, con le nozioni di sintesi apprese durante il corso, ad esempio attraverso uno schema grafico: modellerà quindi la connessione di componenti da lui stesso definiti in VHDL.

Fornendo al sistema un file con le opportune configurazioni di ingresso, verificherà che le due reti operino con lo stesso comportamento e che le uscite siano sempre congruenti. In questo caso, la verifica da esito positivo e lo studente ha realizzato un buon elaborato.

Per realizzare il sistema (da qui in avanti chiamato *verificatore*) si è proceduto con una logica top-down, facendo ricorso dell'ambiente di modellazione grafica. Nello specifico, sono stati inseriti 4 blocchi funzionali, ciascuno dei quali con una propria funzionalità univoca (di qui la scelta di definire FUB anzichè simboli, come per altro esaminato in cap. 2, sez. 5.1): *Campionatore*, *DUT*, *GM* e *Comparatore*, come mostrato in fig. 6.23.

L'uso dell'ambiente grafico non era ovviamente necessario, infatti il circuito VHDL può essere tranquillamente realizzato semplicemente scrivendo le entità e le architetture manualmente e collegandole assieme; una pratica del genere però diviene troppo complessa e prona ad errori quando si modella un circuito costituito da molte componenti, inoltre l'editor grafico può anche essere utilizzato come strumento di debug per la correzione di errori.

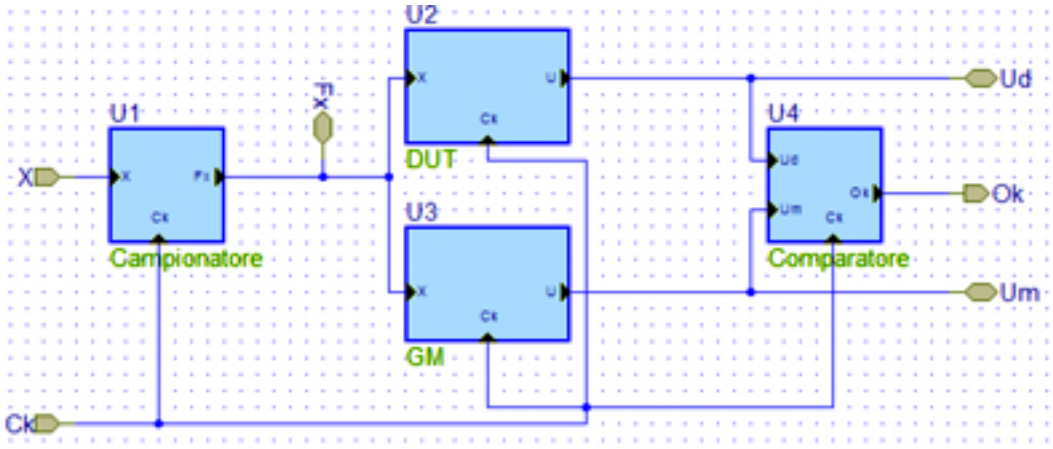


Figura 6.23: Il verificatore nel suo complesso

Nelle sottosezioni seguenti si darà una visione più dettagliata del funzionamento e delle principali caratteristiche dei singoli componenti costituenti il verificatore.

5.1 Campionatore

Il campionatore è un Flip-flop D, pilotato dal clock, con la funzione di sincronizzare l'ingresso fornito alla rete (che potrebbe essere asincrono) e mantenerlo stabile per tutto il ciclo di clock, in maniera da consentire alle reti a valle di gestire il dato e di operare correttamente. Questa è una procedura nota in letteratura quando si devono collegare segnali esterni ad una rete sincrona.

Nello specifico, trattandosi di una macchina a stati finiti (modellata nei componenti seguenti) in forma di *Mealy*, ossia dipendente combinatoriamente anche dagli ingressi, si è sperimentalmente verificato che l'eliminazione del campionatore provoca un'evoluzione erronea nel comportamento delle reti a valle. La causa dell'errore dipende dal repentino modificarsi dell'ingresso, che - se non mantenuto stabile dal registro - si ripercuote sulle reti combinatorie interne mentre è ancora in corso il calcolo dello stato futuro.

5.2 Rete da verificare

La *DUT* è stata anche essa modellata con uno stile *strutturale*, utilizzando l'editor grafico; la proposta di soluzione da testare prevede l'uso di un contatore a due bit e di una rete combinatoria per generare i segnali di controllo del contatore e l'uscita. La rete è riportata in figura 6.24.

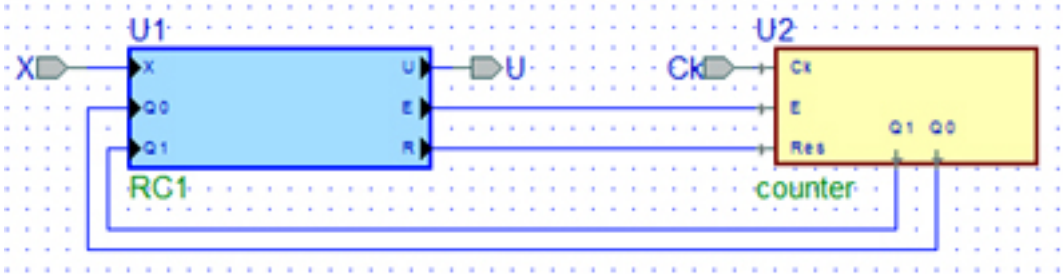


Figura 6.24: DUT

La rete combinatoria è stata modellata con un FUB e la sua implementazione interna è affidata ad un'architettura in stile *dataflow*, descritta dal seguente codice:

```
R <= X;
E <= not Q1;
U <= not Q1 or X;
```

Viceversa, il contatore è un symbol - perchè concettualmente è possibile utilizzarne più istanze nello stesso design - ed è modellato con lo stile *comportamentale*, attraverso un processo:

```
architecture behaviour of Counter is
signal count : std_logic_vector(1 downto 0) := "00";
begin
   process (Ck)
   begin
     if Ck'event and Ck= '1' then
        if Res = '1' then
           count <= "00";
```

```
        elsif E = '1' then
              count <= count + 1;
        end if;
     end if;
   end process;
   Q0 <= count(0);
   Q1 <= count(1);
end behaviour;
```

Come si può notare nel codice, il valore del contatore è memorizzato in un segnale, aggiornato all'interno del processo. Al fronte positivo del clock (quindi al primo delta) il processo è eseguito e l'eventuale variazione del segnale *count* è schedulata per il secondo delta. Lo stato del contatore è portato all'esterno attraverso i segnali *Q0* e *Q1*, questi segnali sono assegnati attraverso due istruzioni esterne al processo e concorrenti, ciò è stato fatto consciamente: il simulatore, in questo modo, renderà effettivo il valore del segnale *count* al secondo delta e schedulerà al terzo delta l'aggiornamento di *Q0* e *Q1* ed in delta successivi l'eventuale ricalcolo di altre istruzioni contenenti tali segnali (se modificati).

Se gli assegnamenti fossero stati inseriti internamente al processo, si sarebbe commesso un errore. In tal caso, all'esecuzione del processo (primo delta) il segnale *count* sarebbe eventualmente schedulato per assumere il nuovo valore al secondo delta, ma (per quanto già descritto in cap. 2, sez. 3.1) i segnali *Q0* e *Q1* continuerebbero a leggere il valore precedente del contatore, perchè il tempo di simulazione si incrementerà solo al successivo loop del processo.

In realtà, è possibile inserire le due assegnazioni all'interno del processo sfruttando il costrutto `wait for 0 ns`; in altre parole, è necessario modificare la struttura del processo (si ricordi che non si possono utilizzare le istruzioni di wait assieme ad una *sensitivity list*) come segue:

```
architecture behaviour of Counter is
signal count : std_logic_vector(1 downto 0) := "00";
begin
```

```
   process
   begin
     if Ck'event and Ck= '1' then
        if Res = '1' then
           count <= "00";
        elsif E = '1' then
              count <= count + 1;
        end if;
     end if;
     wait for 0 ns;
     Q0 <= count(0);
     Q1 <= count(1);
     wait on Ck;
   end process;
end behaviour;
```

con questa soluzione, l'istruzione `wait for 0 ns` sospende il processo per un delta cycle, consentendo così al segnale *count* di rendere effettivo l'eventuale nuovo valore del segnale (parallelamente il tempo di simulazione è portato al secondo delta cycle) e facendo si che i segnali *Q0* e *Q1* assumano il valore corretto.

Ad alto livello, la *RC1* ed il *contatore*, sono considerate dal simulatore come due macro-istruzioni combinatorie eseguite concorrentemente, quindi saranno valutate in parallelo (indipendentemente dall'ordine con cui sono scritte), ma riordinate nell'ordine corretto mediante l'introduzione di opportuni delta cycle e il ricalcolo delle espressioni gia valutate, se almeno uno dei segnali è variato nel delta precedente.

5.3 Golden Model

Il *Golden Model* rappresenta la soluzione dell'esercizio, fornita dal docente. Nello specifico si è ritenuto opportuno modellarne il comportamento direttamente con una macchina a stati finiti, piuttosto che fornire un'implementazione di più basso livello, per poter confrontare la conversione eseguita dal

compilatore con l'equivalente realizzata a mano (riportata nella DUT). Per far ciò si è fatto ricorso all'editor fornito dall'IDE, fig. 6.25.

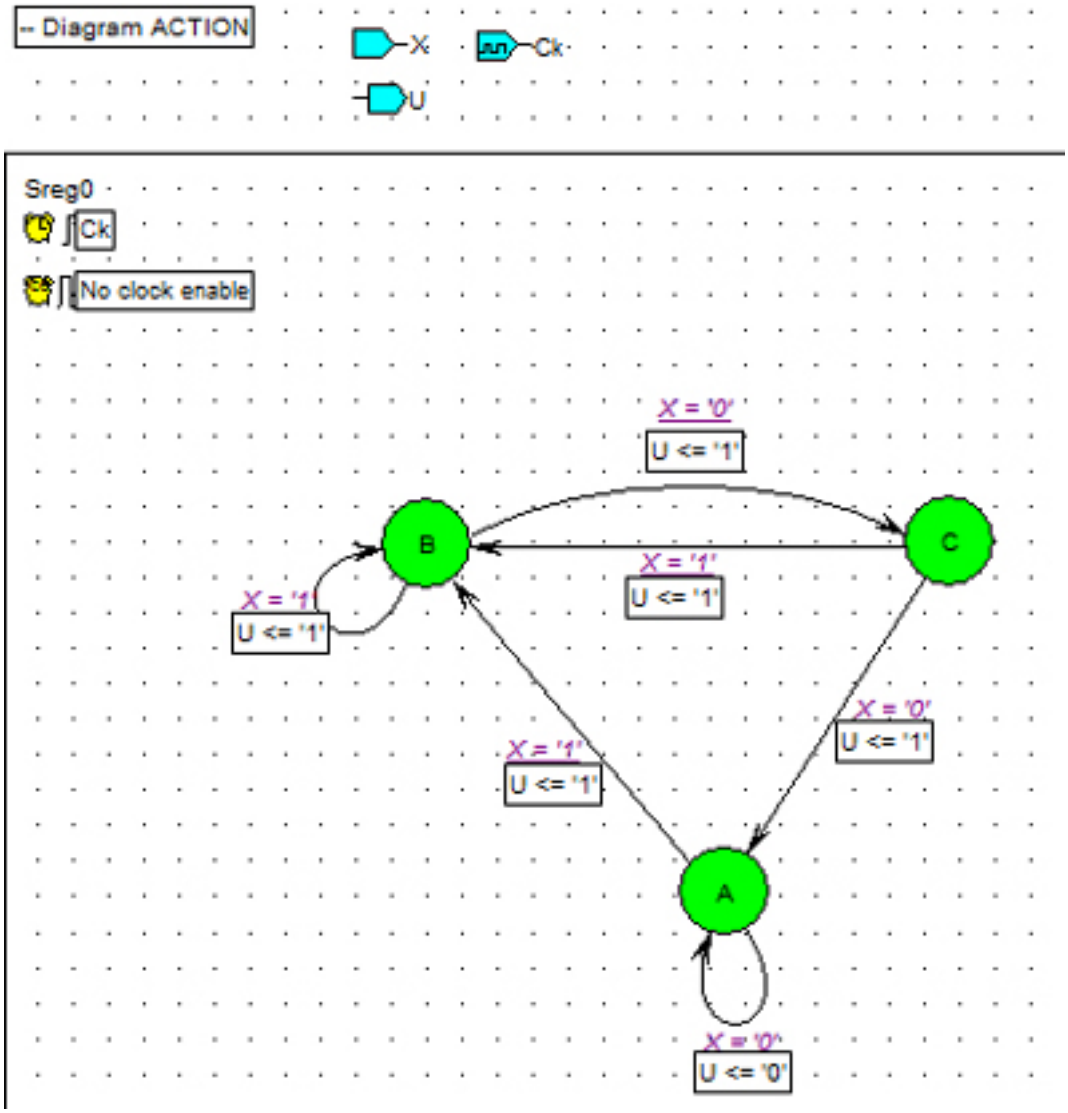


Figura 6.25: GM: macchina a stati finiti modellante il comportamento della rete da progettare

Nella modellazione, si è avuto cura di definire lo stato B come stato iniziale, ordinando manualmente gli stati per mezzo dell'apposita utility (menù `FSM` | `View/Sort Objects` | `States`). Questa operazione è molto importante, perchè se non si definisce esplicitamente lo stato iniziale, ad esempio con un

reset, (si veda anche il cap. 2, sez. 4) si corre il rischio che lo stato di partenza non sia assegnato correttamente.

Il problema è causato dal tipo di dato cui è associato lo stato: essendo un tipo *enumerativo*, implicitamente definisce un ordinamento ed il valore di default, in fase di inizializzazione, è quello dell'elemento di posto 0 (ossia quello più a sinistra). Chiaramente se la macchina partisse con un valore di default erroneo, potrebbe bloccarsi o assumere un comportamento non regolare.

5.4 Comparatore

Questa componente, modellata direttamente in VHDL con uno stile *comportamentale*, ha il compito di verificare che le uscite prodotte dalle reti procedano di pari passo. In caso di verifica positiva, il segnale *Ok* assumerà il valore '1', viceversa il valore '0'. La verifica è resa sincrona al clock, viceversa il comportamento prodotto non sarebbe valido perchè si rischierebbe di confrontare dati intermedi non stabili e quindi di dare informazioni erronee in uscita.

Essendo modellato con un processo, la valutazione delle espressioni è fatta al primo delta di ogni istante temporale per cui si ha il fronte positivo del clock (attivazione del processo), quindi l'eventuale difformità delle uscite è rilevata al clock successivo (come si può vedere in fig. 6.26).

Il *Comparatore* è definito dal seguente codice:

```
architecture behaviour of Comparatore is
begin
   check: process (Ck)
   begin
     if Ck = '1' and Ck'event then
        if (Ud = Um) then
            Ok <= '1';
        else
            Ok <= '0';
```

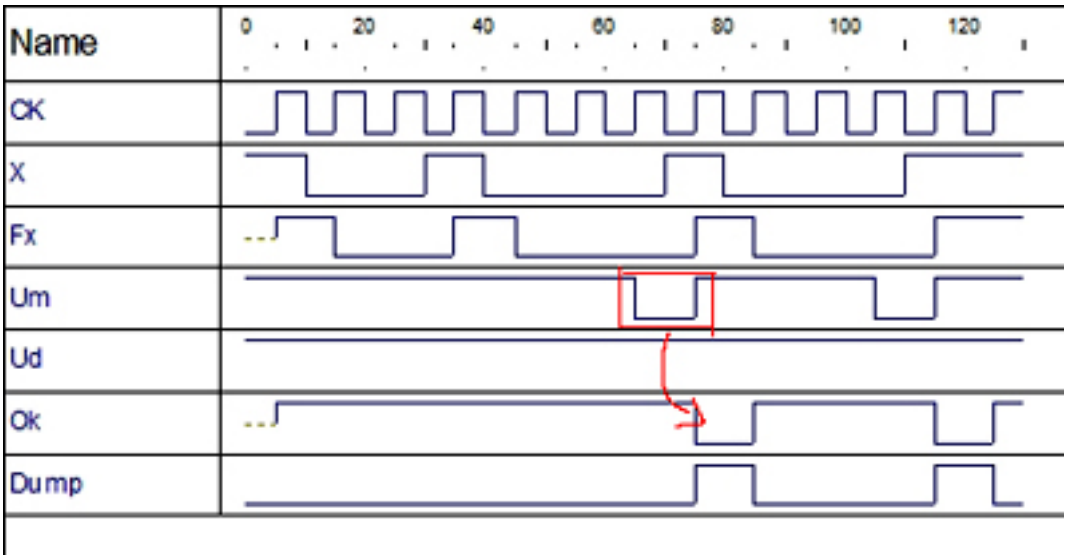


Figura 6.26: Andamento dei segnali, nel caso di uscite difformi

```
        end if;
      end if;
   end process;
end behaviour;
```

5.5 Testbench

Come spiegato precedentemente nel cap. 5, il *testbench* ha il compito di permettere la simulazione e l'analisi del verificatore realizzato. Nella scrittura del testbench si è fatto riferimento alle regole presenti nel cap. 5, sez. 2, in tal senso si è diviso lo sviluppo in processi differenti a seconda della funzione da realizzare.

Generazione del clock

La generazione del clock avviene mediante un processo dedicato, realizzato con il seguente codice:

```
clk_p: process
begin
   Ck <= '0';
   wait for period / 2;
```

```
   Ck <= '1';
   wait for period / 2;
   if (Sim_end = '1') then
       wait;
   end if;
end process;
```

per rendere il codice più riutilizzabile possibile, si è utilizzata una *costante* `period` di tipo *time* pari a 10 ns (periodo totale del clock); inoltre il processo di generazione del clock si interrompe quando il segnale *Sim_end* assume valore '1', attraverso l'istruzione di `wait` indefinita.

Acquisizione degli ingressi da file

Gli ingressi al verificatore sono introdotti dalla lettura di un file testuale, presente sul file-system ed opportunamente formattato. Nella realizzazione dell'architettura, si è previsto di utilizzare due diverse semantiche per i valori letti: la prima semantica prevede che ogni linea letta contenga una coppia di valori, costituita dal valore del segnale e dall'istante di applicazione (assoluto, rispetto all'origine); la seconda semantica invece definisce la coppia come valore del segnale e numero di unità di tempo per cui quel valore deve essere mantenuto.
La selezione tra quale delle due modalità utilizzare è realizzata inserendo come prima linea del file un codice numerico a n bit a seconda del numero di configurazioni possibili.
La lettura dei valori dal file, avviene come descritto in cap. 5, sez. 1.2 e seguenti, ed è espressa con il seguente codice:

```
data: process
variable linea_attuale : line;
variable value1 : std_logic;
variable selezione: std_logic_vector(1 downto 0);
variable t1 : integer;
variable t_delay : time;
begin
   readline(input, linea_attuale);
```

```
read(linea_attuale, selezione);
case (selezione) is
     when "00" =>
        while not endfile(input) loop
           readline(input, linea_attuale);
           if linea_attuale(1) = '#' then
              next;
           end if;
           read(linea_attuale, value1);
           read(linea_attuale, t1);
           t_delay := (t1 * 1 ns) - now;
           wait for t_delay;
           X <= value1;
        end loop;
        Sim_end <= '1';
        wait;

     when "01" =>
        while not endfile(input) loop
           readline(input, linea_attuale);
           if linea_attuale(1) = '#' then
              next;
           end if;
           read(linea_attuale, value1);
           read(linea_attuale, t1);

           for i in 1 to t1 loop
              X <= value1;
              wait for period;
           end loop;
        end loop;
        Sim_end <= '1';
        wait;
```

```
        when others =>
   end case;
end process;
```

Il processo è realizzato per essere eseguito una sola volta, infatti in ogni branch del costrutto *case* di selezione è presente un'istruzione `wait` indefinita. La lettura della prima linea consente di effettuare la selezione tra le due differenti modalità, ciascuna delle quali è definita in uno dei branch (si noti la presenza dell'istruzione `when others =>` per le configurazioni non gestite).

All'interno di ciascun branch sono presenti delle istruzioni per scartare eventuali commenti, questi possono presentarsi su più linee del file, ma il loro riconoscimento (se una linea è contrassegnata dal simbolo iniziale `#` si intende commento e quindi priva di valori da leggere) e la mancata valutazione (attraverso l'istruzione `next` che abortisce l'esecuzione del ciclo corrente e riporta il puntatore all'inizio del loop stesso) dei valori non aumenta il tempo di simulazione. Si ricordi infatti che essendo all'interno di un processo, le istruzioni che coinvolgono variabili sono eseguite in un tempo nullo, quindi non sono inseriti nuovi istanti da simulare nella *event list*; da questo punto di vista, un commento di una riga o di dieci righe non fa alcuna differenza.

Il processo si sospende per la durata prevista in entrambe le configurazioni, permettendo così alla rete di eseguire (i vari processi e la rete verificatore agiscono come macro-istruzioni concorrenti tra loro) e di far avanzare il tempo di simulazione. Al raggiungimento dell'ultima riga del file, viene anche settato il segnale *Sim_end* che arresterà la generazione del clock a partire dall'intervallo successivo.

Scrittura dei risultati su file

Al fine di rendere più completo il processo di verifica, si è realizzato un processo in grado di stampare su un file di testo eventuali discordanze tra l'uscita prodotta dalla rete sotto test ed il Golden Model.

```
dump_p: process (Ok)
variable linea : line;
```

```
begin
   if Ok = '0' then
      write(linea, string'("Comportamento NON corretto all'istante: "));
      write(linea, time'image(now - period));
      write(linea, string'(" - Valore previsto: "));
      write(linea, Um);
      write(linea, string'(" Valore ottenuto: "));
      write(linea, Ud);
      writeline(output, linea);
      dump <= '1';
   else
      dump <= '0';
   end if;
end process;
```

Di significativo, si può notare una forma alternativa conversione in intero di un tipo *time*: utilizzando `time'image(...)`e la presenza del segnale *Ok* nella sensitivity list, ciò significa che questo processo non è pilotato dal clock; la sua esecuzione può essere tranquillamente avviata dal segnale Ok (che, uscendo da una macchina sequenziale è già sincrono).

Conclusioni

L'attività di tesi, condotta in questi mesi, ha permesso di comprendere le caratteristiche e le possibilità offerte dalla modellazione di circuiti elettronici attraverso il linguaggio VHDL.

Nello specifico ci si è concentrati non solo sulla comprensione delle sue strutture fondamentali e sui pattern con cui rappresentare le varie componenti (si pensi, ad esempio, alla progettazione di un *multiplexer* o di un *contatore*), ma anche sulla approfondita conoscenza del processo di funzionamento proprio del simulatore.
Comprendere come avviene la simulazione, perchè il tempo di simulazione si incrementa in certe condizioni e non in altre, capire come cambia il valore di un segnale a seconda della posizione di un'assegnazione in un processo, piuttosto che all'esterno di esso, permette di scrivere codice funzionante, senza *"dover sperare che funzioni"*.

La modellazione dei casi di studio riportati ha permesso di analizzare le prestazioni dell'IDE utilizzato (*Active HDL 7.2 - Student Edition* della *Aldec, inc.*). Questo si è dimostrato un prodotto valido e facilmente apprendibile (almeno nelle funzionalità più classiche).

La possibilità di definire file diversi per ogni componente (includendo in ciascuno di essi l'entità e le relative architetture), ha permesso uno sviluppo ordinato e di facile gestione, con la possibilità di testare singolarmente le singole unità. Il sistema di gestione delle librerie consente di inserire le librerie esterne (ad esempio la *IEEE*, la *STD* o altre create dall'utente) all'interno del workspace corrente rendendole disponibili al progettista per l'inclusione

nel codice.

La guida in linea fornita con il software è abbastanza completa e spiega nel dettaglio le singole strutture del linguaggio, dell'IDE e dei suoi comandi; tuttavia, soprattutto agli inizi, può risultare dispersiva perchè gli argomenti non sono ben legati tra loro, mancando in parte la visione d'insieme.

La modellazione dei casi di studio riportati ha permesso di analizzare le prestazioni dell'IDE utilizzato (*Active HDL 7.2 - Student Edition* della *Aldec, inc.*). Questo si è dimostrato un prodotto valido e facilmente apprendibile (almeno nelle funzionalità più classiche).

Gli editor di modellazione grafica si sono rivelati abbastanza semplici da utilizzare, ma in grado di modellare circuiti anche complessi. Viceversa, la generazione automatica del codice ha fornito risultati diversi: molto buona quella ottenuta dall'editor *FSM* (cap.2, sez. 4), tanto da non richiedere integrazioni al codice; mentre quella ottenuta dall'editor *a blocchi* (cap.2, sez. 5) ha dato risultati nettamente peggiori: spesso il codice risultante richiede l'intervento del progettista, ad esempio per settare i parametri generici dei componenti, ma anche per definire in modo più ottimale le connessioni tra gli stessi.

Uno dei difetti emersi è che il software non riesce a mantenere la sincronia tra il circuito disegnato e la corrispondente trasposizione in VHDL. In altre parole, tutte le modifiche effettuate manualmente nel codice generato (e nel caso del *Block Editor* generalmente sono una quota significativa) non sono propagate all'indietro; ciò richiede al progettista di aggiornare manualmente lo schema (dove possibile). Inoltre, si deve prestare attenzione a non rigenerare automaticamente il codice dopo aver fatto modifiche sullo stesso, perchè altrimenti queste sono sovrascritte senza avvertimenti.

Possibili sviluppi futuri, a partire dal lavoro svolto, sono legati alla sintesi dei componenti modellati (in fase di analisi e simulazione) mediante appositi tool di simulazione che estendano l'IDE: in particolare la affidabilità della sintesi

stessa e le possibilità di ottenere un equivalente maggiormente ottimizzato rispetto a quanto potrebbe fare il progettista. Altri aspetti da approfondire sono il sottoinsieme di linguaggio VHDL sintetizzabile e i test da effettuarsi a fine sintesi per valutare la bontà di quanto realizzato.

Infine si può utilizzare quanto sintetizzato per la programmazione di una FPGA reale, completando così l'intero processo di sviluppo.

Bibliografia

[1] J. Bhasker, *A VHDL primer* - terza edizione. (Prentice Hall - Modern semiconductor design series)

[2] A. Rushton, *VHDL for logic synthesis - An introdution guide for achieving design requirements*. (McGraw Hill)

[3] M. Zwolinski, *VHDL progetto di sistemi digitali*. (Pearson Education Italia)

[4] E. Faldella, materiale del corso di Sistemi Digitali (DEIS, Università di Bologna)

[5] S. Mattoccia, Introduzione al linguaggio VDHL. (DEIS, Università di Bologna)

[6] C. Brandolese, Introduzione al linguaggio VHDL. (Politecnico di Milano)

[7] J. Swift, VHDL Simulation Coding and Modeling Style Guide.
(da `www.emba.uvm.edu/~jswift/uvm_class/labs/vcs/simcg/simcg_2.pdf`)

[8] L. Merola, Guida al VHDL.
(da `http://www.webalice.it/luigimerola/file/archi/vdhl.pdf`)

[9] P.J. Ashenden, VHDL cookbook.

[10] C. Gouldstone, slide sulla gestione di segnali asincroni, macchine di Mealy e Moore.

[11] P. Chodowiec, Finite State Machines and their Testing. (George Mason University)

[12] M. Williamson, E. Lee, Specification and modelling of reactive real-time systems. (UC Berkley, dept. of EECS)

[13] Behavioural Simulation Training Lab - tutorial sull'ambiente IDE, a cura del produttore Aldec, Inc.

[14] Active HDL training - slide di introduzione all'ambiente IDE, a cura del produttore Aldec, Inc.

Rigraziamenti

In poche righe è difficile ricordare tutti coloro che in questi anni mi sono stati vicini, ma ci voglio provare.

Ringrazio di cuore la mia famiglia per tutto l'affetto e il sostegno che mi hanno sempre dato, aiutandomi nelle difficoltà dello studio, della permanenza a Bologna e della vita. Li ringrazio per il loro amore, che ho sentito sempre forte anche nella distanza. Vi voglio bene.

Desidero ringraziare il prof. Eugenio Faldella non solo per avermi affidato questo progetto e per avermi seguito nel lavoro di tesi, ma per il modo in cui lo ha fatto: sempre con grande tranquillità e rispetto, aiutandomi nei punti critici e trasmettendomi la passione per l'analisi e la comprensione delle cose studiate. Affrontare il lavoro di tesi con la sua supervisione non è mai stato nè banale, nè *palloso*.

Voglio ringraziare e ricordare i miei amici più cari di Ancona, perchè anche se ci siamo visti di meno, non hanno mai smesso di farmi sentire il loro calore e la loro vicinanza. Devo anche a loro ed ai loro consigli se oggi sono riuscito ad arrivare a questo importante traguardo.

Un pensiero particolare è per Pietro (il *Dominus*), Giovanni (*Giovà*), Riccardo (il *Pellix*), Luca (il *Pincer*) e Andrea (il *Primix*) per tutti i momenti bellissimi passati assieme.

Qui in facoltà, a Bologna, ho avuto la fortuna di conoscere delle bellissime persone, che sono state sia buoni compagni di studio sia buoni amici.

Ringrazio Francesco (*Frà*), Alessandro (*Makkio*), Marco (*Passa*), Diego (*Kwesi*), Chiara (*Kia*), Francesco (il *Ciaba*) per avermi fatto scoprire un altro modo di vedere le cose, e Davide. Grazie di cuore, ragazzi.

Voglio salutare e ringraziare inoltre Luigi, Antonio, Andrea e Valentina, perchè con la loro amicizia, il loro affetto e i loro consigli mi sono stati sempre vicini sia nei momenti difficili, sia quando c'era da *sfanzare*.

Grazie a tutti.
Stefano *"Sturi"* Suraci.