Elementi lessicali
Un modello VHDL si compone di linee di testo.

Commenti

Si può aggiungere un commento dopo una linea di codice valido VHDL usando il doppio trattino "--". Tutto quello che viene scritto dopo il doppio trattino fino alla fine della riga viene ignorato.
Una riga che comincia con il doppio trattino è un commento.
Esempio di commenti

```
entity andport is -- Questa e' l'inizio della dichiarazione
dell'entita' andport
--  ora dobbiamo definire le porte
     port is (a, b: in bit; y: out bit);
-- non c'e' altro da fare ora si chiude le parte dichiarativa
end entity andport; -- Fine della dichiarazione
```

Identificatori

Sono identificatori i nomi degli "oggetti" che costituiscono un modello VHDL (segnali, entità, nomi di architettura ecc). Gli identificatori devono rispettare le regole seguenti:

- Gli identificatori possono contenere solo lettere, cifre decimali e il carattere "_";
- un identificatore deve cominciare con un carattere alfabetico;
- un identificatore non può terminare con il carattere "_";
- non possono essere presenti due o più caratteri "_" consecutivi;

Non c'è distinzione fra lettere maiuscole e minuscole: "Paolo" e "paolo" sono lo stesso identificatore.

Esempi di identificatori validi: `Cane, cane_e_gatto, gatto_e_2_cani`
Identificatori non validi: `_Cane, cane_, 1_cane_e_due_gatti, 33trentini, due__trattini`

Sono possibili i così detti identificatori estesi. Gli identificatori estesi iniziano e finiscono il carattere "\". Tutti i caratteri (inclusi gli spazi) possono essere usati all'interno di un identificatore esteso. Per esempio `\1 cane e 2 gatti\` è un identificatore valido. Attenzione: nel caso degli identificatori estesi, c'e' distinzione fra maiuscole e minuscole: `\Cane\, \cane\, \CANE\` sono 3 identificatori distinti.

Non possono essere usate come identificatori le parole chiave che fanno parte del linguaggio. Per esempio non si può usare la parola `entity` come identificatore.

Numeri

I numeri possono essere relativi a quantità *intere* o quantità *reali*.
Un numero intero si rappresenta come una sequenza di cifre senza punto decimale. La base di default è la base 10, ma possono essere specificate basi diverse come negli esempi seguenti:
123 numero 123 rappresentato in base 10
2#10010# numero 18 rappresentato in base 2
16#12# numero 18 rappresentato in esadecimale (in esadecimale, oltre alle cifre da 0 a 9 si usano le lettere a, b, c, d, e, f per rappresentare le cifre relative ai numeri 10,11,12,13,14,15).

Come aiuto per la lettura, specialmente nel caso della base 2, si può usare il carattere "_" per separare le cifre. Per esempio le due scritture seguenti sono equivalenti:
2#100111001#
2#1_0011_1001#
Per rappresentare i numeri *reali* sono possibili le consuete notazioni in virgola fissa e esponenziale.

Caratteri

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

Stringhe
Una costante di tipo stringa si indica racchiudendo i caratteri che la compongono fra virgolette. Esempio: `"Questa e' una stringa"`

Il carattere & viene usato per concatenare più costanti di tipo stringa.

Stringhe di bit
In VHDL si ha spesso a che fare con sequenze di bit. Si può rappresentare una sequenza di bit con riferimento alla notazione binaria, ottale o esadecimale. Esempi di stringhe di bit secondo le tre possibili notazioni sono:
B"10001101" oppure B"1000_1101" stringa di bit in notazione binaria
O"342" stringa di bit in notazione ottale (equivalente a B"011_100_010")
X"AF" stringa di bit in notazione esadecimale (equivalente a B"1010_1111"

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

Costanti e variabili

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

Dichiarazione di costante
dichiarazione_di_costante<=**constant** identificatore {,...}:indicatore_di_sottotipo[:=espressione];
La parte di assegnazione (:=espressione) è ovviamente obbligatoria nel caso della definizione di costanti, ma è indicata come opzionale perché ci sono alcuni casi particolari in cui si può definire una costante senza assegnare il valore.

Esempi di dichiarazione di costanti:
```
constant numero_di_byte:integer:=4;
constant numero_di_bit:integre:=8*numero_di_bytes;
constant e:real:=2.718281;
constant ritardo_di_propagazione: time:=3 ns;
```

Dichiarazione di variabile
dichiarazione_di_variabile<= **variable** identificatore
{,...}:identificatore_di_sottotipo[:=espressione];
Il valore iniziale, nel caso in cui non sia esplicitamente specificato, dipende dal tipo. Per i tipi scalari, il valore iniziale è il valore “più a sinistra” del tipo. L'espressione “più a sinistra” sarà più chiara in seguito.
Le variabili possono essere usate esclusivamente all'interno di process.

Assegnazione di variabile

assegnazione_di_variabile<=[label:]nome:=espressione;
Per esempio:
se var_a è una variabile di tipo bit, allora per assegnare il valore '1' si scrive `var_a:='1';`
se var_b è una variabile di tipo intero si può scrivere `var_b:=3+7;`

Tipi scalari

Dichiarazione di tipo

dichirazione_di_tipo<=**type** identificatore **is** definizione_del_tipo;
esempio di definizione di tipo:
```
type mele is range 0 to 100;
type arance is range 0 to 100;
```
Si noti che, nonostante mele e arance prevedano gli stessi insiemi di valori, si tratta di due tipi diversi!
Pertanto, se definissimo le seguenti variabili:

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

l'assegnazione seguente sarebbe scorretta:

```
frutto:=mela+arancia;
```

Si ponga attenzione al fatto che se si vogliono usare tipi definiti dall'utente nella dichiarazione delle porte di una entità, i tipi devono essere in qualche modo già definiti prima della dichiarazione dell'entità stessa. Questo si ottine definendo i tipi che si intendono sfruttare all'interno di un "package". Per il momento un package può essere visto come in insieme di definizioni di tipo. Un package può essere contenuto in un file di testo separato.

Esempio di definizione di package:

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

Una volta che il file che contiene il package viene elaborato, i tipi definiti sono "visibili" se immediatamente prima della definizione dell'entità si scrive:

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

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

Tipi interi

I tipi interi inVHDL hanno valori nell'insieme dei numeri interi. Lo standard del linguaggio prevede che l'intervallo dei valori possibili si estenda da un minimo di $-2^{31}+1$ a un massimo di $2^{31}-1$.
Si possono definire nuovi tipi "intero" nel modo seguente:

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

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

Esempi di definizione di tipi interi:

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

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

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

Ricordiamo che non è permesso scrivere espressioni come:

```
anno_di_nascita:=oggi;
```

Sono possibili definizioni del tipo:
```
constant numero_di_bit:integer:=32;
type bit_index is range 0 to numero_di_bit-1;
```

Sui valori del tipo intero sono definite alcune operazioni:

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

Attenzione: la divisione (A/B) e il resto sono tali da soddisfare la relazione seguente:
A=(A/B)*B+(A rem B) con (A rem B) che ha lo stesso segno di A e un valore assoluto inferiore al valore assoluto di B. L'operazione modulo (mod) coincide con la omonima funzione matematica.

Tipo Floating Point

Lo standard prevede che l'intervallo dei valori rappresentabili deve essere almeno pari a (-1.8E+308 +1.8E308)
Floating_point_definition <= **range** simple_expression {**to** | **downto**} simple_expression

Esempio di definizione di un tipo floating point
type real_value **is range** 0.0 **to** 1e6;

Tipi "fisici"

physical_type_definition<= range simple_expression {**to** | **downto** } simple_expression
units
identifier;
{identifier=physical_literal;}
end units [identifier]
physical_literal<=[decimal_literal | based_literal] unit_name

Esempio:

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

Assegnazione di una variabile del tipo resistance
```
r23:= 330 ohm;
```

Esempi di definizione con unità principale e secondarie
```
type resistance is range 0 to 1e9
     units
          ohm;
```

```
        kohm= 1000 ohm;
        Mohm=1000 kohm;
    end units;


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

Il tipo time è un tipo fisico predefinito nel seguente modo

```
type time is range (definizione dipendente dall'implementazione)
    units
        fs;
        ps= 1000 fs;
        ns= 1000 ps;
        us= 1000 ns;
        ms=1000 us;
        sec=1000 ms;
        min= 60 sec;
        hr= 60 min;
    end units;
```

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

Tipi definiti per enumerazione

Vengono esplicitamente elencati gli elementi del tipo mediante l'elencazione di identificatori o di caratteri alfanumerici.

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

Esempi:

```
type colori_semaforo is (rosso, giallo, verde);

type colori is ('b','w','g','y','r');
```

Il tipo character è un tipo predefinito definito per enumerazione:

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

Un importante tipo predefinito definito per enumerazione è il tipo boolean

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

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

Le espressioni `123=123, 'A'='A', 7 ns=7 ns` producono il valore true;
Le espressioni `123=443, 'A'='z', 7 ns=17 us` producono il valore false
Gli operatori logici **and**, **or**, **nand**, **nor**, **xor**, **xnor** e **not** si applicano a espressioni o variabili di tipo boolean e producono risultati di tipo boolean.
Gli operatori **and**, **or**, **nand**, e **nor** sono chiamati operatori di tipo "short.circuit". Questa denominazione deriva dal fatto che nella valutazione dell'espressione logica viene valutata prima l'espressione a sinistra dell'operatore. Se il risultato è tale che il valore dell'espressione logica è determinato (per esempio nel caso dell' and, se l'espressione a sinistra vale false, il risultato dell'operazione and è comunque false) l'espressione a destra non viene valutata. Questo può essere utile in un caso come quello seguente:

```
(b/=0) and (a/b>1)
```

L'espressione precednete non produce errore se b=0. Infatti in questo caso l'espressione a destra non viene valutata.
Al contrario, l'espressione
`(a/b>1) and (b/=0)`
produrrebbe errore se venisse valutata con b=0.

Un altro importante tipo definito per enumerazione è il tipo bit

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

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

Tipo standard logic
Tipo standar_ulogic è un tipo definito per enumerazione in una package standardizzato (std_logic_1164).

```
type std_ulogic is (     'U', --Unitialized
               'X', – Forcing Unknown
               '0', –  Forcing zero
               '1', – Forcing one
               'Z',-- high impedence
               'W',-- Weak Unknown
               'L', – Weak zero
               'H',-- weak one
               '-'); – don't care
```

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

Sottotipi

subtype_declaration**<=subtype** identifier **is** subtype_indication**;**
subtype_indication<= type_mark **[range** simple_expression **{to | downto}** simple expession**]**

Come suggerisce il nome, un sottotipo è un tipo ottenuto a partire da un tipo già definito. L'insieme dei valori di un sottotipo è generalmente un sottoinsieme dei valori del tipo. Variabili di un sottotipo

e del tipo di partenza possono essere combinati in operazioni di assegnamento o di calcolo.

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

Supponiamo ora che sia
```
var1:small_int:=37;
var2:med_int:=528;
```

L'espressione
```
var2:=var2+var1;
```
ha perfettamente senso.
Si noti che, invece, l'espressione
```
var1:=var1+var2;
```
è lecita, ma produce un errore perché il risultato dell'operazione non appartiene al sottotipo della variabile a sinistra dell'operazione di assegnazione.

Qualificatori di tipo

Supponiamo di aver definito i seguenti tipi:
```
type colore is (rosso, verde,giallo);
type semaforo is (rosso,verde,giallo);
```
I tipi sono distinti e pertanto il valore “rosso” del tipo colore è cosa diversa dal valore “rosso” del tipo semaforo. Per distinguere chiaramente i due valori si può fare esplicito riferimento al tipo di appartenenza mediante le espressioni:
```
colore'(rosso)
semaforo'(rosso)
```

Attributi applicabili ai tipi scalari

Sia TIPO l'identificatore di un tipo o sottotipo scalare.
Allora
TIPO'left indica l'elemento più a sinistra del tipo
TIPO'right indica l'elemento più a destra del tipo
TIPO'low indica il valore più piccolo del tipo
TIPO'high indica il valore più grande del tipo
TIPO'ascending è un valore boolean che vale true se il tipo è ordinato in modo crescente, altrimenti vale false
TIPO'image(x) produce una stringa che rappresenta il valore di x
TIPO'value(s) produce il valore x del tipo “TIPO” che è rappresentato dalla stringa s

Attributi applicabili a tipi discreti e fisici.
Sia ancora TIPO l'identificatore di un tipo discreto o fisico

TIPO'pos(x) produce un intero che indica la posizione di x all'interno dell'insieme dei valori del tipo (il primo valore del tipo ha indice di posizione 0)
TIPO'val(n) produce il valore x che ha indice di posizione n
TIPO'succ(x) produce il valore del tipo di indice di posizione immediantamente superiore a quello di x
TIPO'pred(x) produce il valore del tipo di indice di posizione immediantamente inferiore a quello di

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

Tipi composti

Array
Un array è una collezione ordinata di valori tutti appartenenti allo stesso tipo.
La definisione di un tipo array segue la sintassi:
array_type_definition<=**array (**discrete_range {,....} **of** element_subtype_indication**)**
discrete_range<=discrete_subtype_indication | simple_expression (to | downto) simple_expression
subtype_indication<=type_mark [range simple_expression (to|downto) simple_expression]

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

Supponiamo sia
```
type colori is (nero,rosso,verde,blu,giallo,bianco);
```
possiamo ora definire un array di numeri naturali (sottotipo predefinito del tipo intero) come:
type quantita is array (rosso to giallo) of natural;
In questa definizione ci si affida al fatto che il tipo del range è chiaro dal contesto. Se non fosse cosi' (per esempio se rosso e giallo sono elementi di più tipi definiti per enumerazione) allora si puà scrivere:
```
type quantita is array (colori range rosso to giallo) of natural;
```

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

Se si definisce una variabile di tipo array

```
variable parola:word;
```

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

Se si ha
```
variable parola1,parola2:word;
allora l'espressione
parola1:=parola2;
```
copia tutti gli elementi di parola2 in parola1.

E' possibile definire array multidimensionali.

Assegnazione di valori ad un array.

L'inizializzazione di costanti o variabili di tipo array può essere ottenuta come segue
```
type point is array (1 to 3) of real;
constant origin:point:=(0.0,0.0,0.0);
variable view_point:point:=(10.0,20.0,0.0);
```
In alternativa alla “aasociazione posizionale”, si può specificare il valore da assegnare alle variabili corrispondenti a specifici indici.

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

Altra interessante possibilità:

```
type coeff_array is array (coeff_ram_address) of real;
variable coeff:coeff_array:=(0=>1.6, 1 =>2.3, 2 to 63 =>0.0);
```
o ancora:
```
variable coeff:coeff_array:=(0|3|5|9 =>1.2, 1=>2.3, others=>0.0);
```

Il simbolo "|" serve per separare indici a cui corrisponde lo stesso valore. "others" deve essere sempre messo in fondo).

Descrizione di un modulo di un sistema digitale

La descrizione di un modulo in un sistema digitale può essere pensata come divisa in due parti: la descrizione "esterna" e la descrizione "interna". La descrizione esterna specifica il numero e il tipo delle "porte" che sono presenti per il collegamento al resto del sistema digitale; la descrizione interna specifica il modo nel quale il modulo svolge la sua funzione.
In VHDL si usa la dichiarazione di entità (entity declaration) per ottenere la descrizione esterna; per la descrizione interna si usa il corpo di architettura (architecture body). In generale, a ogni entità possono corrispondere più di un "architecture body", ciascuno dei quali descrive una diversa implementazione interna dello stesso modulo. I vantaggi offerti dalla possibilità di disporre di più di una architettura per la descrizione interna di una unità sono molteplici. In fase di definizione di un sistema digitale complesso ci si può limitare a descrivere il comportamento di un modulo mediante una descrizione funzionale ad alto livello. Solo in un secondo momento, definite le specifiche funzionali generali, si può passare a una descrizione più dettagliata del modulo in termini di sottomoduli o di porte elementari. Poiché possono essere in generale individuate più soluzioni circuitali, a ognuna di esse è possibile far corrispondere una architettura distinta. In questo modo è possibile, per esempio, confrontare nel dettaglio le differenze di comportamento di due o più soluzioni circuitali distinte per ottenere la stessa funzione.

Dichiarazione di una entità

Vediamo intanto la struttura della dichiarazione di entità in una forma semplificata:

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

port_interface_list<=(identifier{, …}:[mode]subtype_indication[:=expression]) {;.....}
mode<= **in** | **out** | **inout**

Alcuni esempi:
```
entity adder is
     port (a: in word;
           b:in word;
           sum:out word);
end entity adder;
```

Nell'esempio precedente si assume che il tipo (o sottotipo, in generale) word sia stato definito in un package.

```
entity and_or_inv is
   port (a1,a2,b1,b2: in bit :='1'; y : out bit);
end entity and_or_inv;
```

Nella dichiarazione precedente, l'aver assegnato un valore alle porte a1,a2,b1,b2 ha esclusivamente il seguente significato: tutte le porte che non saranno collegate a segnali nella entità di rango superiore assumono il valore “1”. Se le porte sono collegate a dei segnali, il valore specificato nella dichiarazione viene ignorato.

Nella dichiarazione di entità possono essere incluse dichiarazioni di segnali, costanti, tipi che saranno poi utilizzabili in tutte le architetture che si riferiscono all'entità.

Prima di esaminare la struttura più completa della dichiarazione di entità, è opportuno discutere della forma in cui può essere descritta la struttura interna di una entità. Ci occupiamo quindi del modo in cui si costruiscono gli *architecture body*.

architecture_body<=
architecture identifier **of** entity_name **is**
{block_declarative_item}
begin
{concurrent_statement}
end [**architecture**] [identifier];

Come già detto, possono essere presenti più architetture, identificate da un diverso “identifier” per ogni entità, identificata da “entity_name”. Una architettura si riferisce in ogni caso a una specifica enitità, per cui possono essere presenti architetture con lo stesso “identifier” che si riferiscono a enità distinte.

Nella sezione dedicata ai {block_declarative_item} possono essere definite costanti, variabili, segnali, tipi e sottotipi. Posso essere anche indicate le altre eventuali entità nella forma di “component” che saranno usate come parte della descrizione così detta “strutturale” (ovvero descrizione in termini del collegamento di altre entità). Come vedremo, tuttavia, il ricorso alla definizione di “component” non è strettamente necessario ai fini di una descrizione strutturale.

Nella sezione dedicata ai concurrent_statement è contenuta la descrizione del comportamento dell'entità in termini di uno o più comandi che specificano l'evoluzione dei segnali interni all'entità. Più precisamente, sulla base dei meccanismi generali di simulazione, nella sezione concurrent_statement si programmano eventi futuri sulla base degli eventi correnti. Semplificando al massimo, si programmano nuove transizioni in funzione della o delle transizioni appena avvenute. Se sono presenti due o più concurrent_statement, essi operano indipendentemente l'uno dall'altro. Il “process”, già esaminato in alcuni esempi elementari, è una particolare forma di concurrent_statement. In realtà, tutte le altre forme di concurrent_statement possono ridursi in ultima analisi a un process, per cui il process può essere visto come il concurrent_statement fondamentale.

Esempio di architettura

```
architecture abstract of adder is
```

```
begin
     process (a,b) is
     begin
          sum<=a+b;
     end process;
end architecture abstract;
```

Dichiarazione di segnali all'interno di una architettura:

signal_declaration <= signal identifier {, …}:subtype_indication[:=expression];

Esempio di architettura

```
architettura esempio of and_or_inv is
     signal and_a, and_b:bit;
     signal or_a_b:bit;
begin
     and_gate_a:process (a1,a2) is
     begin
          and_a<=a1 and a2;
     end process and_gate_a;

     and_gate_b:process(b1,b2) is
     begin
          and_b<=b1 and b2;
     end process and_gate_b;

     or_gate:process (and_a, and_b) is
     begin
          or_a_b<=and_a or and_b;
     end process or_gate;

     inv_gate:process (or_a_b) is
     begin
          y<=not or_a_b;
     end process inv_gate;
end architecture esempio;
```

Assegnamento di segnali

signal_assugnment_statement <= [label:] name <=[delay_mechanism] waveform:
waveform<=(value_expression [after time_expression]){, ….}
delay_mechanism<= transport | [reject time_expression] inertial

per esempio

```
     y<=not or_a_b after 5 ns;

     clk<='1' after 10 ns, '0' after 20 ns;
```

Per la discussione dei diversi meccanismi di ritardo implementati in VHDL, si faccia riferimento alla nota specifica sull'argomento.

Se non compare la specifica del meccanismo di ritardo si assume che esso sia inerziale.

Attributi dei segnali
Sia S il nome di un segnale e T una costante che specifica un tempo. Allora:

S'delayed(T) è un segnale che assume gli stessi valori di S ma con un ritardo pari a T

S'stable(T) è un valore Boolean che è “true” se non c'e' stato alcun evento su S nell'intervallo di tempo di durata T immediatamente precedente al tempo attuale.

S'quiet(T) è un valore Boolean che è “true” se non c'e' stato alcuna transazione su S nell'intervallo di tempo di durata T immediatamente precedente al tempo attuale.

S'transaction è un segnale di tipo bit cha cambia valore da '0 ' a '1' o viceversa tutte le volte che c'e' una transazione du S

S'event è un valore Boolean che è “true” se c'e' un evento su S nel circlo di simulazione corrente.

S'active è un valore Boolean che è “true” se c'e' una transazione su S nel circlo di simulazione corrente.

S'last_event indica il tempo dall'ultimo evento
S'last_active indica il tempo dall'ultima transazione
S'last_value indica il valore di S prima dell'ultimo evento su S

Questi attributi sono particolarmente utili nella fase di simulazione per il controllo del rispetto delle specifiche di sistema mediante “assert”