Limbajul VHDL

VHDL este unul dintre limbajele standard folosite în industrie la ora actuală, pentru a descrie sistemele numerice. VHDL înseamnă **V**HSIC (Very High Speed Integrated Circuits) **H**ardware **D**escription **L**anguage adică un limbaj pentru descriere hardware a circuitelor integrate de foarte mare viteză. Inițial, acest limbaj **foarte asemănător cu limbajul ADA**, a fost destinat doar modelării şi simulării unor circuite ASIC complexe şi numai **ulterior a fost utilizat pentru sinteza şi implementarea logicii** corespunzătoare.

Aşa cum s-a menționat deja, un sistem numeric poate fi reprezentat la diferite nivele de abstractizare. Acest lucru facilitează mai ales descrierea şi proiectarea sistemelor complexe.

Cel mai înalt nivel de abstractizare este nivelul de descriere al comportării (funcționării) numit în engleză **behavioral**. La acest nivel de abstractizare un sistem este descris prin ceea ce face, adică modul cum se comportă şi nu prin componentele sale şi conexiunile dintre acestea. O descriere de acest tip specifică relațiile dintre semnalele de intrare şi ieşire. Descrierea poate fi o expresie booleană sau o descriere mai abstractă, la nivelul transferului între registre sau la nivelul algoritmilor. Ca un exemplu tipic să considerăm un circuit simplu care avertizează pasagerul atunci când uşa automobilului este deschisă sau centura nu este folosită, de fiecare dată când se introduce cheia în contact. Descrierea la nivel funcțional s-ar putea face în modul următor :

Avertizare utilizator = Contact AND (Usa_deschisa OR Nu_Centura)

Nivelul structural, spre deosebire de nivelul 'behavioral', descrie un sistem ca o colecție de porți şi componente care sunt conectate între ele pentru a realiza funcția dorită. O reprezentare structurală poate fi comparată cu o schemă de porți logice conectate între ele. Aceasta este o reprezentare care se apropie mai mult de realizarea fizică a sistemului.

Limbajul VHDL permite reprezentarea sistemelor la nivel funcțional (behavioral) sau structural. Nivelul behavioral poate fi împărțit în două stiluri de reprezentare: al fluxului de date (Data Flow) şi algoritmic. Reprezentarea de tip Data Flow descrie modul cum circulă datele prin sistem, aceasta realizându-se în termenii transferului de date între registre (RTL). Această descriere foloseşte instrucțiuni concurente, care se execută în paralel în momentul în care datele sunt prezente la intrare. Pe de altă parte în reprezentarea de tip algoritmic instrucțiunile sunt secvențiale şi se execută în ordinea specificată. VHDL permite atribuirea semnalelor în ambele moduri (secvențial şi concurent).

1. Structura unui fişier sursă VHDL

Un sistem numeric este descris ca **o entitate (entity) principală** care, la rândul ei, poate conține alte entități, acestea fiind considerate componente ale entității principale. Fiecare entitate este modelată de o **declarație a entității** şi de **corpul arhitectural**. Declarația entității se poate considera **o interfață cu mediul extern** care defineşte semnalele de intrare şi de ieşire, pe când corpul arhitectural conține descrierea entității şi este compus din alte entități, procese şi componente interconectate, toate funcționând în acelaşi timp.

La fel ca orice alt limbaj VHDL, foloseşte **cuvinte cheie**, iar acestea nu pot fi folosite ca nume de semnale sau identificatori (simboluri definite de utilizator).

O linie de comentariu, care va fi ignorată de compilator, începe cu `--` .

Declararea entităților

Declararea entității defineşte numele entității şi specifică porturile de intrare şi de ieşire. Forma generală este următoarea :

entity NUME_ENTITATE **is** [**generic** *declaratii_generice*);]
port (*nume_semnale* **: mode** *tip* ;
nume_semnale **: mode** *tip* ;
:
nume_semnale **: mode** *tip*;
end [NUME_ENTITATE] ;

O entitate începe întotdeauna cu cuvântul cheie **entity** (entitate) urmat de numele entității şi de cuvântul cheie **is** (este). În continuare sunt declarațiile porturilor cu cuvântul cheie **port**. Declarația entității

se termină întotdeauna cu cuvântul cheie **end**. NUME_ENTITATE este un identificator (simbol) ales de utilizator;

nume_semnale constă într-o listă de identificatori separați prin virgulă care specifică semnalele interfeței externe ;

mode este un cuvânt rezervat care indică sensul semnalelor ;

in - semnal de intrare;

out - semnal de ieşire; valoarea poate fi citită decât de alte entități

buffer - semnal este ieşire, dar valoarea poate fi citită în arhitectura entității

inout - semnal ce poate fi ieşire sau intrare (bi-direcțional);

tip este tipul de semnal; exemple de tipuri sunt: **bit, bit_vector, boolean, character, std_logic, std_ulogic**

Corpul arhitectural (architecture)

Corpul arhitectural specifică modul în care funcționează şi în care este implementată o entitate. După cum am spus mai devreme, o entitate sau circuit poate fi specificat în modul funcțional sau cel structural dar şi de o combinație dintre cele două. Generalizând, corpul arhitectural arată în felul următor:

```
architecture nume_arhitectura of NAME_ENTITATE is
-- Declaratii
      -- declaratii ale componentelor
      -- declaratii de semnale
      -- declaratii de constante
      -- declaratii de functii
      -- declaratii de proceduri
      -- declaratii de tipuri
begin
      -- Instructiuni
end nume_arhitectura;
```

Descrierea funcțională (behavioral) a arhitecturii pentru exemplul amintit mai sus, în care avertizarea s-ar realiza prin intermediul semnalului BUZZER este:

```
architecture behavioral of BUZZER is
begin
      Avertizare <= ( not Usa AND Contact ) OR ( not Centura AND Contact ) ;
end behavioral;
```

Descrierea structurală a aceluiaşi circuit, bazată pe utilizarea unor porți AND, OR cu două intrări şi a inversorului NOT va fi următoarea:

```
architecture structural of BUZZER is
-- Declaratii
component AND2
      port (in1, in2: in std_logic;
            out1: out std_logic);
  end component;
component OR2
      port (in1, in2: in std_logic;
            out1: out std_logic);
  end component;
component NOT1
      port (in1: in std_logic;
            out1: out std_logic);
  end component;
-- declaratiile semnalelor folosite la interconectare
signal USA_NOT, CENTURA_NOT, B1, B2: std_logic;
begin
```

```
-- Componentele trebuie sa fie si instantiate
      U0: NOT1 port map (USA, USA_NOT);
      U1: NOT1 port map (CENTURA, CENTURA_NOT);
      U2: AND2 port map (CONTACT, USA_NOT, B1);
      U3: AND2 port map (CONTACT, CENTURA_NOT, B2);
      U4: OR2  port map (B1, B2, AVERTIZARE);
end structural;
```

În partea de declarații se precizează componentele ce urmează a fi folosite în descrierea circuitelor. În exemplul nostru folosim o poartă ŞI cu două intrări, două porți SAU cu două intrări şi un inversor. Aceste porți trebuie definite în prealabil, adică vor avea nevoie de declararea entității şi de un corp arhitectural aşa cum s-a arătat mai sus. Acestea pot fi păstrate într-unul din package-uri după cum se va arăta în continuare. Instrucțiunile de după cuvântul cheie begin creează instanțieri ale componentelor şi descriu modul cum acestea sunt conectate între ele. Definirea unei instanțe creează un nou nivel ierarhic. Fiecare linie începe cu *numele* (de exemplu U0) urmat de două puncte, *numele componentei* şi cuvântul cheie **port map**. Acest cuvânt cheie defineşte modul în care sunt conectate componentele. În exemplul de mai sus, aceasta se face prin asocierea poziției: semnalul USA corespunde intrării in1 a porții NOT1, iar USA_NOT corespunde ieşirii. La fel este şi pentru poarta AND2 unde primele două semnale, CONTACT şi USA_NOT, corespund intrărilor in1 şi in2, iar semnalul B1 corespunde ieşirii out1.

Se poate folosi şi un mod alternativ de asociere a porturilor, astfel

eticheta: numele componentului **port map** (*port1=>semnal1, port2=> semnal2,… portn=>semnaln*);

```
U0: NOT1 port map (in1 => USA, out1 => USA_NOT);
```

Modelarea structurală impune o proiectare ierarhizată în care se pot defini componente care sunt folosite de mai multe ori. Odată ce au fost definite, aceste componente pot fi folosite ca blocuri, celule sau macro-uri într-o entitate la un nivel mai înalt. Aceasta poate reduce în mod semnificativ complexitatea proiectelor mari. **La fel ca la orice aplicație software proiectele ierarhizate sunt preferate întotdeauna în locul celor pe un singur nivel**.

Biblioteci (library) şi package-uri; cuvintele cheie library şi use

O bibliotecă poate fi considerată ca un loc unde compilatorul păstrează informații despre un proiect.

Un **package** VHDL este un fişier sau un modul care conține: declarațiile obiectelor folosite în mod curent, tipuri de date, declarații de componente, semnale, proceduri şi funcții care pot fi folosite de diferite modele VHDL. Spre exemplu **std_logic** este definit în package-ul **ieee.std_logic_1164** din biblioteca **ieee**. Pentru a folosi std_logic trebuie specificată biblioteca şi package-ul. Aceasta se face la începutul fişierului VHDL folosind cuvintele cheie **library** şi **use** după cum urmează :

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

Extensia **.all** indică faptul că se foloseşte tot package-ul ieee.std_logic_1164.
Luând ca exemplu mediul Xilinx ISE biblioteca ieee conține următoarele package-uri:
- **std_logic_1164**; defineşte tipurile de date standard
- **std_logic_arith**; pune la dispoziție funcții aritmetice, de conversie şi comparație pentru tipurile de date signed, unsigned, integer, std_ulogic, std_logic şi std_logic_vector
- **std_logic_unsigned**
- **std_logic_misc**; defineşte tipuri suplimentare, subtipuri, constante şi funcții pentru package-ul std_logic_1164 .

Pentru a folosi oricare dintre aceste package-uri trebuiesc folosite directivele library şi use:

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

Pentru a folosi funcțiile de bază AND2, OR2, NAND2, NOR2, XOR2, etc. acestea trebuiesc definite. Aceasta se poate face într-un package definit de utilizator, de exemplu functii_baza, pentru fiecare din aceste componente, după cum urmează:

```
-- Declararea package
library ieee;
use ieee.std_logic_1164.all;
package functii_baza is

-- Declarare AND2
component AND2
generic (DELAY: time :=5ns);
port (in1, in2: in std_logic; out1: out std_logic);
end component;

-- Declarare OR2
component OR2
generic (DELAY: time :=5ns);
port (in1, in2: in std_logic; out1: out std_logic);
end component;
end package functii_baza;

-- Declararea corpului package-ului
library ieee;
use ieee.std_logic_1164.all;
package body functii_baza is

-- poarta ŞI cu două intrări
entity AND2 is
generic (DELAY: time);
port (in1, in2: in std_logic; out1: out std_logic);
end AND2;

architecture model_conc of AND2 is
begin
out1 <= in1 and in2 after DELAY;
end model_conc;

--poarta SAU cu două intrări
entity OR2 is
generic (DELAY: time);
port (in1, in2: in std_logic; out1: out std_logic);
end OR2;

architecture model_conc2 of OR2 is
begin
out1 <= in1 or in2 after DELAY;
end model_conc2;
end package body basic_func;
```

Întârzierea intrare–ieşire de 5ns (delay) este ignorată la sinteză (Xilinx ISE). A fost folosit tipul predefinit std_logic care este declarat în package-ul std_logic_1164. S-au inclus directivele library şi use pentru acest package. Package-ul creat mai sus trebuie compilat şi plasat într-o bibliotecă pe care o putem numi spre exemplu, funct_pers. Pentru a putea folosi componentele din acest package, acesta la rândul lui trebuie declarat folosind directivele library şi use:

```
library ieee,funct_pers ;
use ieee.std_logic_1164.all, funct_pers.functii_baza.all ;
```

Directivele library şi use trebuiesc folosite de fiecare dată când se declară o entitate.

2. Elemente lexicale ale limbajului VHDL

Identificatori (simboluri utilizator)

Identificatorii sunt cuvinte definite de utilizator pentru a numi obiectele din modulele VHDL. Până acum s-au întâlnit exemple de identificatori pentru semnale de intrare şi ieşire, dar şi nume de entități şi de corpuri arhitecturale. La alegerea unui identificator trebuie sa se țină cont de câteva reguli de bază:

- identificatorii pot conține numai caractere alfanumerice (A-Z , a-z , 0-9) şi caracterul 'underscore' (_)
- primul caracter trebuie sa fie o literă iar ultimul nu poate fi ' _ '
- un identificator nu poate conține două caractere underscore consecutive
- nu contează tipul de literă (literele mari sau mici), spre exemplu And2 şi AND2 se referă la acelaşi obiect
- un identificator poate avea orice număr de caractere

Exemple de identificatori corecți : AND2 , poarta_1 , AND_2 .
Exemple de identificatori incorecți : _AND2 , poarta__2, AND-2, semnal_1_

Identificatorii de mai sus sunt numiți **identificatori de bază**. Regulile pentru identificatorii de bază sunt uneori prea restrictive pentru a indica semnale. De exemplu dacă se doreşte indicarea unui semnal activ în zero ca de exemplu un RESET activ în zero, acesta nu va putea fi denumit /RESET. Pentru a depăşi aceste limitări există un set de reguli pentru aşa zişii **identificatorii extinşi** care permit definirea de identificatori cu orice secvență de caractere.

- un identificator extins este cuprins între caractere backslash " \ "
- pentru un identificator extins **contează tipul de literă** (litere mari sau mici)
- un identificator extins este diferențiat de cuvintele rezervate (cuvinte cheie) sau de orice identificator de bază (ex. identificatorul \identity\ este permis)
- între două caractere backslash se poate folosi orice secvență de caractere cu excepția faptului că un caracter " \ " în interiorul unui identificator extins trebuie dublat. Ca un exemplu, pentru a folosi identificatorul BUS:\data acesta se va declara în felul următor \BUS:\\data\

Cuvinte rezervate

Anumiți identificatori sunt folosiți de sistem ca şi **cuvinte cheie**. Aceste cuvinte nu pot fi folosite ca identificatori de bază pentru semnalele sau obiectele definite de utilizator. Am întâlnit deja câteva cuvinte rezervate cum ar fi in, out, or, and, port, map, end, etc. Cuvintele cheie sunt de cele mai multe ori listate cu caractere îngroşate. Identificatorii extinşi pot folosi cuvintele cheie deoarece aceştia sunt priviți ca şi cuvinte separate (de exemplu identificatorul extins \end\ este permis)

Reprezentările numerice

Reprezentarea numerică obişnuită este reprezentarea zecimală. VHDL permite reprezentări de tip **întreg** şi **real**. Reprezentarea de tip întreg constă în număr, fără punct zecimal, iar reprezentarea reală include întotdeauna punctul zecimal. Notația exponențială este permisă folosindu-se 'E' sau 'e'. Pentru tipul întreg exponentul trebuie să fie întotdeauna pozitiv. Exemple de reprezentări sunt :

- tipul întreg : 12 , 10 , 256E3 , 12e+6
- tipul real : 1.2 , 256.24 , 3.14E-2

Pentru a exprima un număr în altă bază decât baza 10, se foloseşte următoarea convenție gnerală: baza#număr#. Câteva exemple sunt date în continuare.

baza 2 : 2#10010# (reprezentând numărul zecimal 18)
baza 16 : 16#12#
baza 8 : 8#22#
baza 16 : 16#1D#

Pentru a face citirea numerelor cu mai multe ranguri mai uşoară se pot insera caractere underscore atâta timp cât nu sunt inserate la început sau la sfârşit : 2#1001_1101_1100_0010# , 215_123 .

Caractere, şiruri de caractere, şiruri de biţi

Pentru a se putea folosi caractere literale în codul VHDL, se pune caracterul între ghilimele simple: 'a' , 'B' , ',' . Pe de altă parte şirurile de caractere sunt plasate între ghilimele duble: "Acesta este un sir de caractere". Pentru a folosi ghilimelele în interiorul unul şir de caractere acestea se vor dubla: "Acesta este un ""sir""."

Orice caracter ASCII ce are un echivalent tipăribil poate fi introdus în interiorul unui şir de caractere.

Un şir de biţi reprezintă o secvenţă de valori 0 sau 1. Pentru a se indica faptul că acesta este un şir de biţi, se plasează litera 'B' în faţa şirului: B"1001" . Se pot folosi de asemenea şiruri în baza 16 sau 8 folosind specificatorii X respectiv O. Câteva exemple sunt următoarele:

binar : `B"1100_1001" , b"1001011"`

hexazecimal : `X"C9" , X"4b"`

baza 8 : O"311" , o"113"

Trebuie menţionat că în sistemul hexazecimal fiecare digit reprezintă exact 4 biţi. De aceea numărul b"1001011" nu este acelaşi cu X"4b" deoarece primul are doar 7 biţi, iar al doilea reprezintă o secvenţă de 8 biţi.

Obiecte de tip date: semnale, variabile şi constante

Un obiect de tip dată este creat de o declaraţie a obiectului şi are asociate o valoare şi un tip. Un obiect poate fi o constantă, o variabilă, un semnal sau un fişier(file). Până acum s-au întâlnit semnale care erau folosite ca porturi de intrare/ieşire sau conexiuni interne. Semnalele pot fi considerate trasee într-o schemă care au o valoare curentă şi valori viitoare şi care sunt funcţii ale instrucţiunilor de asignare a semnalelor. De cealaltă parte, variabilele şi constantele sunt folosite pentru a modela funcţionarea circuitului şi sunt folosite în procese, proceduri şi funcţii la fel cum se folosesc în orice limbaj de programare.

Constante - O constantă poate avea o singură valoare de un tip specificat şi nu se poate modifica pe parcursul modelării. O constantă se declară în felul următor:

constant numele_constantei : tipul [:= valoare_iniţială]

unde valoarea iniţială este opţională. Constantele se pot declara la începutul unei arhitecturi şi pot fi folosite apoi oriunde în arhitectură. Constantele care se declară într-un proces pot fi folosite numai în procesul respectiv. Ca exemple:

```
constant const_1 : integer := 24 ;
constant const2 : time := 2 ns ;
constant const_1,const2 : integer := 24;
constant DATA_BUS : integer := 16;
```

Variabile - O variabilă poate avea o singură valoare la un moment dat, ca o constantă, dar ea poate fi şi actualizată folosind o instrucţiune de actualizare. Variabila este actualizată fără nici o întârziere, imediat ce instrucţiunea este executată. Variabilele trebuie declarate în interiorul proceselor. Declararea variabilelor se face în modul următor:

variable lista_nume_variabile : tipul [:= valori initiale];

Câteva exemple de declaraţii de variabile sunt următoarele:

```
variable CNTR_BIT: bit :=0;
variable VAR1: boolean :=FALSE;
variable SUM: integer range 0 to 256 :=16;
variable STS_BIT: bit_vector (7 downto 0);
```

Variabila SUM, din exemplul de mai sus, este un număr întreg care poate lua valori între 0 şi 256 cu valoarea iniţială 16 la începutul modelării. Al patrulea exemplu defineşte un vector de 8 biţi: STS_BIT(7), STS_BIT(6), ... , STS_BIT(0).

Valoarea unei variabile poate fi actualizată printr-o instrucţiune de asignare de genul: nume_variabilă := expresie;

Semnale - Semnalele sunt declarate cu următoarele instrucţiuni:

```
signal lista_nume_semnale : tipul [ := valori initiale ];
```

Exemple de astfel de declaraţii sunt:

```
signal SUMA,CARRY : std_logic ;
```

```
signal TRIGGER : integer :=0 ;
signal CLOCK : bit ;
signal DATA_BUS : bit_vector (0 to 7);
signal VALOARE : integer range 0 to 100 ;
```

Semnalele sunt actualizate când este executată instrucțiunea de asignare, cu o anumită întârziere, cum se arată mai jos:

```
SUMA <= (A xor B) after 2 ns;
```

În particular se poate astfel specifica şi o formă de undă:

```
signal unda : std_logic;
unda <= '0','1' after 5ns, '0' after 10ns, '1' after 20ns ;
```

Diferențele dintre variabile şi semnale sunt importante şi constă în special în momentul la care acestea îşi schimbă valoarea. O variabilă este actualizată imediat când este executată instrucțiunea de actualizare. Semnalele îşi modifică valoarea cu o anumită întârziere după ce a fost evaluată expresia de asignare. Dacă nu este specificată întârzierea, semnalul va avea o întârziere generică *delta*. Aceasta are consecințe importante pentru valorile actualizate ale semnalelor şi ale variabilelor. Pentru a ilustra acest lucru, se va face o comparație între două surse VHDL în care se foloseşte un proces pentru a calcula semnalul RESULT, odată folsosind variabile şi odată semnale. Exemplul de proces în care se folosesc variabile este:

```
architecture VAR of EXEMPLU is
signal TRIGGER, RESULT: integer := 0;
begin
process
variable variable1: integer :=1;
variable variable2: integer :=2;
variable variable3: integer :=3;
begin
wait on TRIGGER;
variable1 := variable2;
variable2 := variable1 + variable3;
variable3 := variable2;
RESULT <= variable1 + variable2 + variable3;
end process;
end VAR
```

Iar exemplul în care se folosesc semnale este:

```
architecture SIGN of EXEMPLU is
signal TRIGGER, RESULT: integer := 0;
signal signal1: integer :=1;
signal signal2: integer :=2;
signal signal3: integer :=3;
begin
process
begin
wait on TRIGGER;
signal1 <= signal2;
signal2 <= signal1 + signal3;
signal3 <= signal2;
RESULT <= signal1 + signal2 + signal3;
end process;
end SIGN;
```

În primul caz, variabilele 'variable1' , 'variable2' şi 'variable3' sunt calculate secvențial şi valorile lor actualizate instantaneu după apariția semnalului TRIGGER. În continuare semnalul RESULT este calculat folosind noile valori ale variabilelor. Rezultă următoarele valori: variable1 = 2 , variable2 = 5 , variable3 = 5. Deoarece RESULT este un semnal, acesta va fi calculat la momentul TRIGGER şi actualizat la momentul TRIGGER + delta, iar valoarea sa va fi RESULT = 12.

În al doilea exemplu însă, semnalele vor fi calculate la momentul TRIGGER. Toate aceste semnale sunt calculate în acelaşi timp, folosind vechile valori ale signal 1, 2 şi 3. Toate semnalele vor fi actualizate la

momentul delta după sosirea semnalului TRIGGER. În acest mod, semnalele vor avea următoarele valori: signal1 = 2, signal2 = 4, signal3 = 2, iar RESULT va avea valoarea 7.

3. Tipuri de date în VHDL

Fiecare obiect de natura unei date are un tip asociat. Tipurile definesc un set de valori pe care obiectul le poate avea şi implicit un set de operaţii care sunt permise pe acesta. **Noţiunea de tip este noţiune cheie pentru VHDL deoarece acesta este un limbaj în care fiecare obiect trebuie să aibă un tip riguros definit (strongly typed).** În general nu este permisă asignarea unei valori de un tip unui obiect de un alt tip.

Există patru categorii de tipuri: scalar, compus (composite), acces (access) şi fişier (file).

Tipurile scalar reprezintă o singură valoare şi sunt ordonate astfel încât se pot efectua operaţii relaţionale. Tipul scalar include subtipurile discret, virgulă mobilă (real) şi fizic. Subtipul discret cuprinde integer şi tipuri gen enumerare de valori booleene, biţi şi caractere.

Tipurile compuse sunt cele vectoriale (array) şi înregistrare (record).

Tipurile de date predefinite care există în VHDL sunt predefinite în package-ul standard (std) după cum se arată în tabelul 1. Pentru a folosi acest package trebuie inclusă următoarea directivă :

```
library std, work;
use std.standard.all;
```

Tabelul 1

Tipurile definite în package-ul *standard* din biblioteca *std*		
Tip	**Domeniul de valori**	**Exemplu**
Bit	'0', '1'	`signal A: bit :=1;`
Bit_vector	un şir de elemente de tip bit	`signal INBUS: bit_vector(7 downto 0);`
Boolean	FALSE, TRUE	`variable TEST: Boolean :=FALSE'`
character	orice caracter permis VHDL	`variable VAL: character :='$';`
Integer	domeniul depinde de implementare dar include cel puţin $-(2^{31}-1)$: $+(2^{31}-1)$	`constant CONST1: integer :=129;`
Natural	întreg începând cu 0 până la valoarea maximă admisă	`variable VAR1: natural :=2;`
Positive	întreg începând cu 1 până la valoarea maximă admisă	`variable VAR2: positive :=2;`
String	un şir de elemente de tip caracter	`variable VAR4: string(1 to 12):= "@$#ABC*()_%Z";`

Tipuri de date definite de utilizator se pot introduce ca noi tipuri folosind declaraţia de tip, care denumeşte tipul şi specifică domeniul de valori. Sintaxa pentru definirea tipurilor este :

type identificator **is** definitia_tipului;

În continuare vor fi date câteva exemple de definiţii de tipuri.

Tipurile întregi

```
type small_int is range 0 to 1024;
type my_word_length is range 31 downto 0;
subtype data_word is my_word_length range 7 downto 0;
```

Un subtip este un subset al unui tip definit anterior. Ultimul exemplu ilustrează modul de folosire al subtipurilor. Subtipul numit data_word este un subtip al my_word_length al căriu domeniu este restricţionat între 7 şi 0. Alt exemplu de subtip ar fi: subtype int_small is integer range -1024 to +1024;

Tipurile în virgulă mobilă

```
type cmos_level is range 0.0 to 3.3;
type pmos_level is range -5.0 to 0.0;
type probability is range 0.0 to 1.0;
subtype cmos_low_V is cmos_level range 0.0 to +1.8;
```

Tipurile fizice

Definirea tipului fizic include un identificator al unității de măsură a mărimii fizice în cauză dar acest tip prezintă o importanță mai mică pentru sinteză (de exemplu el nu este un tip permis de programul de sinteză Xilinx Foundation).

Tipurile enumerative

Un tip enumerativ constă într-o listă de caractere sau identificatori. Tipul enumerativ poate sa fie foarte util la scrierea modelelor la un nivel abstract.

Sintaxa pentru tipurile enumerative este,

type nume_tip **is** (identificatori sau listă de caractere);

Iată câteva exemple,

```
type my_3values is ('0', '1', 'Z');
type PC_OPER is (load, store, add, sub);
type oct_digit is ('0','1','2','3','4','5','6','7';
type state_type is (S0, S1, S2, S3);
```

Exemple de obiecte care folosesc tipurile de mai sus sunt date în continuare,

```
signal SIG1: my_3values;
variable ALU_OP: pc_oper;
variable first_digit: hex_digit :='0';
signal STATE: state_type :=S2;
```

Dacă nu se inițializează semnalul, inițializarea implicită este valoarea extremă din stânga domeniului.

Tipurile enumerative trebuiesc definite în corpul arhitectural sau în interiorul unui package după cum se arată în continuare:

```
type STD_ULOGIC is (
                'U',   -- neinițializat
                'X',   -- impus nedeterminat
                '0',   -- impus 0
                '1',   -- impus 1
                'Z',   -- înaltă impedanță
                'W',   -- slab nedeterminat
                'L',   --  slab 0
                'H'.   -- slab 1
                '-');  -- neimportant
```

Pentru a se folosi acest tip trebuie să se includă clauza înainte de fiecare declarație:

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

Este posibil ca mai multe porți sa impună valoarea unui semnal (cazul unui SI cablat la ieşire). În acest caz ar putea exista un conflict şi semnalul ar putea fi nedeterminat. Ca un exemplu ieşirile porților AND şi NOT sunt conectate împreună la portul de ieşire OUT1. Pentru a rezolva valoarea ieşirii, se poate apela o funcție specială. Aceste funcții sunt de obicei funcții scrise de utilizator şi care vor rezolva problema. Package-ul std_logic_1164 are o astfel de funcție pre definită, numită RESOLVED. Se poate folosi următoarea declarație pentru semnalul OUT1:

```
signal OUT1: resolved: std_logic;
```

Dacă există conflict, funcția RESOLVED va fi folosită pentru intermedierea conflictului şi determinarea valorii semnalului.

Tipurile compuse

Obiectele compuse constau într-o colecție de date legate între ele, sub forma unui vector sau a unei înregistrări(record). Înainte de a se putea folosi, acest tip trebuie declarat.

Un tip **array** este declarat în felul următor:

```
type nume is array (schema indexare) of tip_element;
```

```
type MY_WORD is array (15 downto 0) of std_logic;
type YOUR_WORD is array (0 to 15) of std_logic;
type VAR is array (0 to 7) of integer;
type STD_LOGIC_1D is array (std_ulogic) of std_logic;
```

În primele două exemple este definită un vector unidimensional de elemente de tipul std_logic indexat de la 15 la 0, şi respectiv de la 0 la 15. Ultimul exemplu defineşte tot un vector unidimensional de elemente de tipul std_logic care foloseşte tipul std_ulogic pentru a defini index-ul. Această vector arată astfel: 'U' 'X' '0' '1' 'Z' 'W' 'L' 'H' '-'.

Câteva exemple de obiecte de acest tip sunt:

```
signal MEM_ADDR: MY_WORD;
signal DATA_WORD: YOUR_WORD :=  B"1101100101010110";
constant SETTING: VAR := (2,4,6,8,10,12,14,16);
```

În primul exemplu, semnalul MEM_ADDR este un vector de 16 biți inițializați toți cu 0. Pentru a accesa elemente individuale ale vectorului, trebuie specificat indexul. Astfel M_ADDR(15) accesează bitul cel mai din stânga al vectorului, DATA_WORD(15) accesează bitul cel mai din dreapta al şirului, care are valoarea '0'. Pentru a accesa un subdomeniu, se specifică indexul subdomeniului, MEM_ADDR(15 downto 8) sau DATA_WORD(0 to 7).

Vectorii multidimensionali pot fi de asemenea declarați folosind o sintaxă similară cu următoarele declarații:

```
type MATRICE3X2 is array (1 to 3, 1 to 2) of natural;
type MATRICE4X2 is array (1 to 4, 1 to 2) of integer;
type STD_LOGIC_2D is array (std_ulogic,std_ulogic) of std_logic;
variable DATA_ARR: MATRICE4X2 :=((0,2), (1,3), (4,6), (5,7));
```

Pentru a accesa un element se va specifica indexul, de exemplu DATA_ARR(3,1) care returnează valoarea 4. Uneori este util să nu se specifice dimensiunea vectorului când se declară tipul. Sintaxa este următoarea:

type nume **is array** (tip **range** <>) **of** tip_elemente;

Tipul înregistrare (record) este al doilea tip compus este. O înregistrare (record) constă în mai multe elemente care pot fi de tipuri diferite. Sintaxa pentru tipul record este :

type nume_tip **is**
record
identificator :indicarea_subtipului;
.....
identificator : indicarea_subtipului;
end record;

Ca un exemplu se poate da :

```
type MY_MODULE  is
record
            RISE_TIME   :time;
            FALL_TIME   : time;
            SIZE        : integer range 0 to 200;
            DATA        : bit_vector (15 downto 0);
end record;
signal A, B: MY_MODULE;
```

Pentru a accesa valorile sau a atribui valori unui record, se poate folosi una dintre metodele următoare:

```
A.RISE_TIME <= 5ns;
A.SIZE <= 120;
            B <= A;
```

Conversiile de tip au o mare importanță deoarece VHDL este un limbaj puternic bazat pe tipuri. Astfel nu se poate asigna o valoare de un anumit tip, unui semnal de alt tip. În general, se preferă folosirea

aceluiaşi tip de date pentru semnalele dintr-un design, spre exemplu std_logic (în loc de combinații de std_logic şi bit). Uneori însă nu se poate evita folosirea tipurilor diferite. Pentru a permite atribuirea datelor între diferite tipuri este nevoie de o conversie de la un tip la altul.

Din fericire există funcții disponibile în câteva package-uri din biblioteca ieee, cum ar fi package-urile std_logic_1164 şi std_logic_arith. De exemplu package-ul std_logic_1164 permite conversiile de tip din tabelul 2.

Tabelul 2

Conversii suportate de package-ul std_logic_1164	
Conversia	**Funcția**
std_ulogic -> bit	to_bit(*expresie*)
std_logic_vector -> bit_vector	to_bitvector(*expresie*)
std_ulogic_vector -> bit_vector	to_bitvector(*expresie*)
bit -> std_ulogic	To_StdULogic(*expresie)*
bit_vector -> std_logic_vector	To_StdLogicVector(*expresie*)
bit_vector -> std_ulogic_vector	To_StdUlogicVector(*expresie*)
std_ulogic -> std_logic_vector	To_StdLogicVector(*expresie*)
std_logic -> std_ulogic_vector	To_StdUlogicVector(*expresie*)

Package-urile std_logic_unsigned şi std_logic_arith permit conversii adiționale cum ar fi integer -> std_logic_vector şi invers.

Sintaxa unei conversii de tip este :

nume_tip (expresie);

Pentru a fi permisă conversia, expresia trebuie să întoarcă un tip ce poate fi convertit în tipul nume_tip. Condițiile ce trebuiesc îndeplinite pentru a fi posibilă conversia sunt:

- conversiile sunt posibile între tipuri întregi şi tipuri array similare;
- conversia între tipurile array este posibilă dacă au aceeaşi lungime şi dacă au acelaşi tip de elemente sau tipuri de elemente convertibile;
- tipurile enumerative nu pot fi convertite.

Atributele (attributes) există în VHDL în două categorii: unele pre definite ca parte a standardului 1076 şi unele introduse în afara standardului, eventual de către utilizator. Atributele pre definite sunt întotdeauna **aplicate ca un prefix** numelui unui semnal,variabile sau tip. Atributele sunt folosite pentru a returna diferite tipuri de informație despre un semnal, variabilă sau tip. Atributele constau într-un prefix (‘) urmat de numele atributului. Ele sunt prezentate în tabelul 3.

Tabelul 3

Atribut	**Funcție**
nume_semnal’event	întoarce valoarea booleană True dacă semnalul a avut o tranziție şi False dacă nu
nume_semnal’active	întoarce True dacă a existat o atribuire a valorii semnalului, dacă nu, întoarce False
nume_semnal’transaction	întoarce un semnal de tip 'bit' care se schimbă din 1 în 0 sau din 0 în 1 la fiecare tranziție a semnalului.
nume_semnal’last_event	întoarce intervalul de timp care a trecut de la ultimul eveniment al semnalului
nume_semnal’last_active	întoarce intervalul de timp care a trecut de la ultima atribuire a valorii semnalului
nume_semnal’last_value	furnizează valoarea semnalului înainte de ultimul eveniment al acestuia
nume_semnal’delayed(T)	furnizează un semnal care este întârziat cu (T) față

	de semnalul original [T este opțional, implicit T=0]
nume_semnal'stable(T)	întoarce True, dacă nu a existat nici un eveniment în intervalul de timp T, altfel întoarce False. [T este opțional, implicit T=0]
nume_semnal'quiet(T)	întoarce True, dacă nu a existat nici o atribuire a valorii semnalului în intervalul de timp T, altfel întoarce False. [T este opțional, implicit T=0]

Un exemplu ar fi următoarea expresie care verifică apariția unui front crescător de ceas:

```
if (CLOCK'event and CLOCK='1') then ...
```

4. Operatori VHDL

Limbajul VHDL suportă diferite clase de operatori care operează pe semnale, variabile şi constante. Clasele de operatori sunt prezentate în tabelul 4.

Tabelul 4

Clasa						
1. Operatori logici	and	or	nand	nor	xor	xnor
2. Operatori relaționali	=	/=	<	<=	>	>=
3. Operatori de deplasare	sll	srl	Sla	sra	rol	ror
4. Operatori aditivi	+	=	&			
5. Operatori unari	+	-				
6. Operatori multiplicativi	*	/	mod	rem		
7. Alți operatori	**	abs	Not			

Prioritatea operatorilor este maximă pentru operatorii din clasa 7, urmată de clasa 6, iar cea mai mică prioritate o au operatorii din clasa 1. Dacă nu sunt folosite paranteze, operatorii din clasa 7 sunt aplicați prima dată, după care urmează celelalte clase în ordinea priorității. Operatorii din aceeaşi clasă au aceeaşi prioritate şi sunt aplicați de la dreapta la stânga într-o expresie.

Ca un exemplu să considerăm următorii vectori : X (='010'), Y('10'), şi Z ('10101'). În acest caz expresia: not X & Y xor Z rol 1 este echivalentă cu ((not X) & Y) xor (Z rol 1) = ((101) & 10) xor (01011) =(10110) xor (01011) = 11101. Operatorul xor este executat la nivel de bit.

Operatorii logici (**and**, **or**, **nand**, **nor**, **xor** şi **xnor**) sunt definiți pentru tipurile "bit", "boolean", "std_logic", "std_ulogic" şi pentru vectorii de aceste tipuri. Aceştia sunt folosiți pentru a definii expresii logice booleene sau pentru a face operații la nivel de bit între şiruri de biți. Rezultatul acestor operatori este de tipul operanzilor (Bit sau Boolean). Aceşti operatori pot fi aplicați semnalelor, variabilelor sau constantelor.

Trebuie reținut că operatorii nand şi nor nu sunt asociativi şi trebuiesc folosite paranteze într-o secvență de nand şi nor pentru a preveni erori de sintaxă.

De exemplu X nand Y nand Z va da o eroare de sintaxă şi va trebui scrisă (X nand Y) nand Z.

Operatorii relaționali (tabelul 5) testează relația dintre două tipuri scalare şi oferă ca rezultat o ieşire booleană True sau False.

Tabelul 5

Op.	**Descriere**	**Tipul Operanzilor**	**Tipul Rezultatului**
=	Egalitate	orice tip	Boolean
/=	Inegalitate	orice tip	Boolean
<	Mai mic	scalar sau array discret	Boolean
<=	Mai mic sau egal	scalar sau array discret	Boolean
>	Mai mare	scalar sau array discret	Boolean
>=	Mai mare sau egal	scalar sau array discret	Boolean

De menționat că simbolul operatorului "<=" (mai mic sau egal) este acelaşi cu cel al operatorului de atribuire pentru semnale şi variabile. În exemplul următor primul simbol "<=" este operator de atribuire.

```
      variable STS      : Boolean;
      constant A        : integer :=24;
      constant B_COUNT  : integer :=32;
      constant C        : integer :=14;
      STS <= (A < B_COUNT) ;
-- va atribui valoarea "TRUE" variabilei STS
      STS <=  ((A >= B_COUNT) or (A > C));
-- STS va fi "TRUE"
STS <= (std_logic ('1', '0', '1') < std_logic('0', '1','1'));
-- STS va fi "FALSE"
      type new_std_logic is ('0', '1', 'Z', '-');
      variable A1: new_std_logic :='1';
      variable A2: new_std_logic :='Z';
     STS <=  (A1 < A2);
```

În ultimul exemplu STS va rezulta "TRUE" deoarece '1' este la stânga lui 'Z' deoarece la tipurile enumerative discrete, comparația se face la nivel de element, începând de la stânga spre dreapta..

Operatorii unari '+' şi '-' (tabelul 6) sunt folosiți pentru a specifica semnul unui tip numeric.

Tabelul 6

Op.	Descriere	Tip Operand	Tip Rezultat
+	Identitate	Orice tip numeric	Acelaşi tip
-	Negație	Orice tip numeric	Acelaşi tip

Operatoriii de deplasare execută o deplasare sau rotire la nivel de bit într-un vector de elemente de tip bit (std_logic) sau boolean.

În tabelul 7 sunt prezentați operatorii de deplasare, descrierea şi tipul operanzilor. Ca exemplu după aplicarea operatorului din următorul exemplu:

```
      variable NUM1 :bit_vector := "10010110";
      NUM1 srl 2;
```

NUM1 va fi evaluat ca "00100101".

Dacă argumentul este un întreg negativ deplasarea stânga va deveni deplasare dreapta: NUM1 srl –2 este echivalent cu NUM1 sll 2 vor da ca rezultat "01011000".

Tabelul 7

Op.	Descriere	Tip Operand	Tip Rezultat
sll	Deplasare logică la stânga - spațiile rămase în dreapta sunt completate cu '0'	Stânga: orice tip array unidimensional cu elemente de tip bit sau boolean Dreapta: întreg	Tipul operandului din stânga
srl	Deplasare logică la dreapta spațiile rămase libere completate cu '0'	idem	idem
sla	Deplasare aritmetică stânga - spațiile sunt completate cu bitul din marginea dreaptă	Idem	idem
sra	Deplasare aritmetică dreapta - spațiile sunt completate cu bitul din marginea stângă	Idem	idem
rol	Rotire stânga (circular)	Idem	idem
ror	Rotire dreapta (circular)	idem	idem

Operatorii aditivi (tabelul 8) sunt folosiți pentru operații aritmetice (adunare şi scădere) pe orice tip numeric de operanzi. Operatorul de concatenare (&) este folosit pentru a concatena doi vectori având ca

rezultat un vector mai mare. Pentru a folosi aceşti operatori trebuie specificate package-ul std_logic_unsigned sau std_logic_arith pe lângă package-ul std_logic_1164.

Tabelul 8

Op.	Descriere	Tip operand stânga	Tip operand dreapta	Tip rezultat
+	Adunare	Tip numeric	Tipul op din stânga	Acelaşi tip
-	Scădere	Tip numeric	Tipul op din stânga	Acelaşi tip
&	Concatenare	Tip Array sau element	Tipul op din stânga	Acelaşi tip

Operatorii de multiplicare (tabelul 9) sunt folosiți pentru a executa funcții matematice pe tipurile numerice (întreg sau virgulă mobilă)

Tabelul 9

Op.	Descriere	Tip operand stga.	Tip operand dreapta	Tip rezultat
*	înmulțire	întreg sau virgulă mobilă	Acelaşi tip	Acelaşi tip
		Orice tip fizic	Întreg sau real	Acelaşi tip
		Întreg sau real	Tip fizic	Acelaşi tip
/	împărțire	întreg sau virgulă mobilă	întreg sau virgulă mobilă	Acelaşi tip
		Orice tip fizic	Întreg sau real	Tip op. stânga
		Orice tip fizic	Acelaşi tip	Întreg
mod	modul	Orice tip întreg		Acelaşi tip
rem	rest	Orice tip întreg		Acelaşi tip

Alți operatori relevanți mai sunt prezentați în tabelul 10.

Tabelul 10

Op.	Descriere	Tip operand Stânga	Tip operand Dreapta	Tip rezultat
**	Ridicare la putere	Întreg	Întreg	tip operand stânga
		virgulă mobilă	Întreg	tip operand stânga
abs	Valoare absolută	orice tip numeric		acelaşi tip
not	Negare logică	orice tip bit sau boolean		acelaşi tip

5. Exemple

Exemplele de sinteză prezentate in continuare sunt orientate pe circuite CPLD (eventual prezintă anumite particularitati in cazul implementării cu circuite FPGA) şi pot compilate folosind mediul Xilinx ISE Foundation/Webpack şi compilatorul XST VHDL.

a. Multiplexoare

Este prezentat un circuit de multiplexare 4:1 implementat clasic, cu nivele de porți şi ieşire normală, precum si unul cu ieşire de tip tri-state.

```
library ieee;
use ieee.std_logic_1164.all;
entity MUX4_1 is
port
(
Sel         : in std_logic_vector(1 downto 0);
A, B, C, D  : in std_logic;
Y           : out std_logic
```

```
);
end MUX4_1;

architecture behavior of MUX4_1 is
begin
process (Sel, A, B, C, D)
begin
 case Sel is
      when "00" => Y<=A;
      when "01" => Y<=B;
      when "10" => Y<=C;
      when "11" => Y<=D;
      when others => Y<=A;
 end case;
end process;
end behavior;
```

O varianta similara cu ieşire tri-state:

```
library ieee;
use ieee.std_logic_1164.all;
entity mux is
  port (a, b, c, d: in std_ulogic;
        s: in std_ulogic_vector(1 downto 0);
        y: out std_logic);
end mux;

architecture three_state of mux is
begin
  y <= a when s="00" else 'Z';
  y <= b when s="01" else 'Z';
  y <= c when s="10" else 'Z';
  y <= d when s="11" else 'Z';
end three_state;
```

b. Codificatoare şi decodificatoare

Exemplele prezintă un codificator 8:3, un decodificator 3:8 şi un decodificator pentru adresare.

În cazul decodificatorului de adresare spațiul de adrese de 64k adresat cu 16 biți de adresă, este împărțit liniar din punct de vedere al decodificării în 16 secțiuni egale de 4k şi/sau 4 secțiuni de 16k; din acestea 3 secțiuni egale de 16k sunt identificate prin semnalele de decodificare AddDec active în „1”, iar o a doua zonă de 16 k este la rândul ei împărțită în patru cu semnale de decodificare similare.

```
Library IEEE;
Use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;
entity ENCODER8 is
port (A: in std_logic_vector (7 downto 0);
      Y: out std_logic_vector (2 downto 0));
end entity ENCODER8;

architecture ARCH of ENCODER8 is
begin
 with A select
 Y <= "000" when "00000001",
      "001" when "00000010",
      "010" when "00000100",
      "011" when "00001000",
      "100" when "00010000",
      "101" when "00100000",
```

```
      "110" when "01000000",
      "111" when "10000000",
      "XXX" when others;
end architecture ARCH;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity Decoder3_8 is
Port ( A : in integer range 0 to 7;
       Y : out std_logic_vector(7 downto 0));
end Decoder3_8;

architecture behavioral of Decoder3_8 is
begin
process (A)
begin
 for N in 0 to 7 loop
      if (A = N) then
       Y(N) <= '1';
      else
       Y(N) <= '0';
       end if;
end loop;
end process;
end behavioral;


library IEEE;
use IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.all;
entity decoder_4_bit is
Port ( Address       : in integer range 0 to 15;
       AddDec_0to3   : out std_logic;
       AddDec_8to11  : out std_logic;
       AddDec_12to15 : out std_logic;
       AddDec_4to7   : out std_logic_vector(3 downto 0));
end decoder_4_bit;

architecture behavioral of decoder_4_bit is
begin
process (Address)
      begin
      AddDec_0to3 <= '0';
      AddDec_4to7 <= (others => '0');
      AddDec_8to11 <= '0';
      AddDec_12to15 <= '0';
      case Address is
--prima zona
      when 0 to 3 =>
      AddDec_0to3 <= '1';
      --a doua zona, 4 iesiri dec.
      when 4 => AddDec_4to7(0) <= '1';
      when 5 => AddDec_4to7(1) <= '1';
      when 6 => AddDec_4to7(2) <= '1';
      when 7 => AddDec_4to7(3) <= '1';
--a treia zona
      when 8 to 11 =>
      AddDec_8to11 <= '1';
--a patra zona
```

```
      when 12 to 15 =>
      AddDec_12to15 <= '1';
      end case;
end process;
end behavioral;
```

c. Circuite secvențiale

Primul exemplu este un numărător binar sincron de 16 biți, cu intrare de reset . Intrarea de reset este activă in „1”, iar semnalul de ceas este activ pe frontul crescător.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL; --
entity num16 is
Port ( rst : in std_logic;
       clk : in std_logic;
       count: out std_logic_vector(15 downto 0));
end num16;

architecture behavioral of num16 is
signal temp: std_logic_vector(15 downto 0);
begin
      process (clk, rst)
      begin
      if (rst = '1') then
      temp <= "0000000000000000";
      elsif (clk'event and clk='1') then
      temp <= temp + 1;
      end if;
      end process;
count <= temp;
end behavioral;
```

Următorul exemplu este un registru paralel-paralel de 8 biți (D - intrări, Q - ieşiri) prevăzut si cu o intrare de Reset, activă in „0”. Semnalul de ceas este activ pe frontul crescător.

```
library ieee;
use ieee.std_logic_1164.all;
entity reg is
  port (D: in std_ulogic_vector(7 downto 0);
        Clock, Reset: in std_ulogic;
        Q: out std_ulogic_vector(7 downto 0));
end reg;

architecture behavioral of reg is
begin
    process (Clock, Reset)
    begin
      if (Reset = '0') then
        Q <= (others => '0');
      elsif rising_edge(Clock) then
        Q <= D;
      end if;
    end process;
end behavioral;
```

Un alt exemplu este un registru serie-paralel de 8 biți, intrarea serială de date este a, iar ieşirile sunt q. Semnalul de ceas clk este activ pe frontul crescător.

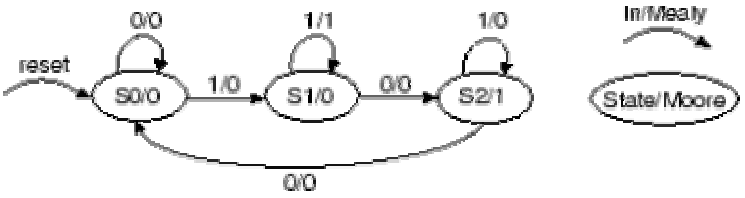
```
library ieee;
use ieee.std_logic_1164.all;
entity sipo is

  port(a : in std_ulogic;
       q : out std_ulogic_vector(7 downto 0);
       clk : in std_ulogic);
end sipo;

architecture rtl of sipo is
begin
   process (clk)
      variable reg : std_ulogic_vector(7 downto 0);
    begin
      if rising_edge(clk) then
        reg := reg(6 downto 0) & a;
        q <= reg;
      end if;
    end process;
end rtl;
```

In exemplul următor este prezentată o maşină secvențială cu 3 stări (S0, S1, S2) cu o intrare In si 2 ieşiri notate Mealy si Moore. Ieşirea Moore depinde doar de starea prezentă (de exemplu in starea S0 ea este întotdeauna „0” indiferent de starea intrării In) pe când ieşirea Mealy depinde de starea prezentă **si** de intrarea In.

In figura alăturată este prezentata diagrama de stări a maşinii.




```
LIBRARY ieee;
  use ieee.std_logic_1164.all;

entity sm is
    port (
        clk, In, reset: in std_logic;
        Mealy: out std_logic;
        Moore: out std_logic
    );
end sm;

architecture behavior of sm is
    type states is (S0,S1,S2);
    signal present_state, next_state : states;
begin
    state_register: process (clk)
    begin
        if (clk'event and clk='1') then
            if (reset = '1') then
                present_state <= S1;
            else
                present_state <= next_state;
            end if;
```

```
        end if;
    end process;

    next_state_transition: process (present_state, In)
    begin

        next_state <= present_state;
        Mealy <= '0';
        Moore <= '0';

        case (present_state) is
            when S0 =>
                if (In='1') then
                    next_state <= S1;
                else
                    next_state <= S0;
                end if;

            when S1 =>
                if (In='0') then
                    next_state <= S2;
                else
                    Mealy <= '1';
                    next_state <= S1;
                end if;

            when S2 =>
                Moore <= '1';
                if (In='1') then
                    next_state <= S2;
                else
                    next_state <= S0;
                end if;
        end case;
    end process;

end behavior;
```