# VHDL - ca un limbaj de programare

VHDL seamana cu un limbaj de programare; cei care sint familiarizati cu limbajul de programare Ada vor observa similaritati cu acest limbaj.

## 1.Elemente lexicale

*a)Comentarii:*
Comentarife in VHDL incep cu '--' si se continua pina la sfirsitul rindului. Ele nu au semnificatie intr-o descriere VHDL.

*b)Identificatori:*
Identificatorii in VHDL sint cuvinte rezervate sau nume definite de programator. Se formeaza dupa regula:

identificator ::= litera {[_ ] litera_sau_cifra}

Nu se face diferenta intre litere mari si litere mici, de exemplu id si Id reprezinta acelasi lucru.

*c)Numere:*
Numerele sint reprezentate in baza 10 (numere zecimale) sau in alta baza de numeratie (de la 2 la 16). Numerele care contin '.' sint considerate numere reale, celelalte fiind numere intregi. Numerele zecimale sint definite de:

numar_zecimal ::= intreg[.intreg][exponent]
intreg ::= cifra{[_]cifra}
exponent :.= E[+]intreg |E[-]intreg

Exemple:

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 123_456 | 678E9 | --numere intregi |
| 0.0 | 0.1 | 2.345 67 | 12.3E-4 | --numere reale |

Numerele date intr-o baza de numeratie sint definite de:

numar bazat ::= baza#intreg bazat[.intreg-bazat]#[exponent]
baza ::= intreg
intreg_bazat ::= cifra_extinsa{[_]cifra extinsa}
cifra-extinsa ::= cifra | litera

Baza si exponentul sint in baza 10. Exponentul reprezinta puterea la care se ridica baza, cu care va fi inmultit numarul. Literele de la A la F (de la a la f) sint 'cifre extinse' si reprezinta numerele de la 10 la 15.

Exemple:

| | | |
|---|---|---|
| 2# 1100 0100# | 16#C4# | 4#301 #E1 | --nr. intreg 196 |
| 2# 1.1111_1111_111 | #E+ 11 | 16#F. FF#E2 | --nr. real 4095 |

*d) Caractere:*
Caracterele sint delimitate de ' '.
Exemple: 'A' 'a'

*e)Siruri de caractere:*
Sirurile de caractere sint delimitate de "". Pentru a include " intr-un sir, ghilimelele trebuie dublate. Un sir de caractere poate reprezenta valoarea unui obiect care a un vector de caractere.
Exemple:
“Un sir in sir: “”Un sir”” “ --sir care contine "

*f)Siruri de biti:*
VHDL permite o reprezentare convenabila a vectorilor de biti ('0' sau ' 1'). Sintaxa este:

sir biti ::= baza_de_reprezentare"valoare_bit"
baza de_reprezentare ::= B | O | X
valoare bit ::= cifra extinsa{[ ]cifra_extinsa}

Baza de reprezentare poate fi B (in binar), O (in octal) sau H (in hexazecimal).
Exemple:

B"1010110" --lungimea sirului e 7
O"126" --lungimea a 9, B”001_010_110"
H"56" --lungimea a 8, B"0101_0110"

## 2.Tipuri de date si obiecte

In VHDL exista doua feluri de tipuri: tipuri SCALARE si tipuri COMPUSE.

Tipurile scalare includ numere, cantitati fizice si enumerari, si tipuri predefinite. Tipurile compuse sint vectori si inregistrari. In VHDL sint definite si tipurile 'access' (pointeri) si 'file' (fisiere).

declaratie_de_tip :.= **type** identificator **is** tip
tip ::= tip_scalar
tip_compus
tip_access
tip_file
tip_scalar ::= tip_enumerare I tip-intreg I tip_real tip_fizic
tip-compus ::= tip_tablou I tip_inregistrare

*a)Tip intreg:*
Tipul intreg reprezinta o multime de numere intregi dintr-un interval specificat. Sintaxa este:

tip_intreg ::= multime_in_interval
multime_in_interval ::= range
interval interval ::= expresie_simpla directie expresie_simpla
directie :.= **to** I **downto**

Expresiile care specifica intervalul trebuie sa aiba valori intregi. Limitele intervalului sint cuprinse intre -2147483647 si +2147483647.

Exemple:

**type** byte_int **is range** 0 **to** 255;
**type** signed **is range** -32768 **to** 32767;
**type** bit_index **is range** 31 **downto** 0;

Exista tipul predefinit 'integer', care reprezinta numere intregi cuprinse intre -2147483647 si +2147483647.

*b)Tip fizic:*
Tipul fizic este un tip numeric de reprezentare a unor cantitati fizice (lungime, timp, volti). Declaratia de tip include specificarea unei unitati de masura de baza si eventual un numar de unitati de masura secundare, care reprezinta multiplii ai unitatii de baza. Sintaxa este:

tip_fizic ::= constructor_interval
**units**
unitate_de_baza
{unitati secundare}
**end units**
unitate_de baza ::= identificator;
unitati_secundare ::= identificator = literal_fizic;
literal_fizic :.= [literal_abstract]nume_unit;

Exemple:

```
type    length is range 0 to 1E9
        units
        um;
        mm = 1000 um;
        cm = 10 mm;
        m = 1000 mm;
end units;

type    resistance is range 0 to 1E8
        units
        ohms;
        kohms = 1000 ohms;
        Mohms = 1E6 ohms;
end units;
```

Exista tipul predefinit 'time', folosit in simulari VHDL pentru specificarea intirzierilor.

```
type    time is range interval_maxim_din_implementare
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;
```

Un numar de tip fizic se scrie: valoare unitate.

Exemple:
10 mm  1200 ohm  23 ns

*c)Tip real:*
Tipul real reprezinta o aproximare discreta a setului de numere reale ditrun interval specificat. Precizia de aproximare nu a definita de limbajul VHDL standard, dar numarul trebuie sa aiba maxim 6 cifre. Intervalul a cuprins intre -1E38 to + 1E38. Sintaxa: tip real :.= constructor interval

Exemple:

**type** signal level **is range** -10.00 **to** +10.00;
**type** probability **is** range 0.0 **to** 1.0;

Exista un tip predefinit 'real'. Intervalul de valori este predefinit si include valorile cuprinse intre -1E38 si +1E38.

*d)Tip enumerare:*
Tipul enumerare este o multime ordonata de identificatori sau caractere. Identificatorii si caracterele din cadrul unei enumerari trebuie sa fie distincti, dar pot fi 'refolositi' in enumerari diferite. Sintaxa este:

tip_enumerare ::= (enumeraref{,enumerare})
enumerare ::= identificator I caracter

Exemple:

**type** logic level **is** (unknown,low,undriven,high);
**type** alu_function **is** (disable,pass,add,substract,multiply,divide);
**type** octal_digit **is** (‘0’,’1’,’2’,’3’,’4’,'5',’6’,’7’);

Exista o serie de tipuri de enumerari predefinite:

**type** severity-level is (note,warning,error,failure);
**type** boolean is (false,true);
**type** bit is ('0','1');
**type** character **is** (
NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
BS, HT, LF, VT, FF, CR, SO, SI,
DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
CAN, EM, SUB, ESC, FSP, GSP, RSP, USP,
' ', '!', … 'z', '{', '|', '}' , '~', DEL);
I
*e)Tablouri :*
In VHDL, un tablou e o colectie indexata de elemente de acelasi tip. Tablourile pot fi unidimensionale sau multidimensionale. Un tablou poate avea dimensiunea definita la declarare sau nedefinita, urmind ca indicele sa is valori defente ulterior. Sintaxa este:

tip_tablou ::= tablou dim-nedefinita I tablou_dim_definita
tablou dim_nedefinita ::=**array** (subtip-index{,subtip-index})
**of** subtip element
tablou dim _definita ::= array multime_index **of** subtip_element
subtip-index ::= tip range <>
multime_index ::=(interval _discret{,interval_discret})
interval discret ::= subtip discret I interval

Exemple:
**type** word **is array** (31 **downto** 0) **of** bit;
**type** memory **is array** (address) **of** word;
**type** transform **is array** (1 to 4,1 to 4) **of** real;
**type** register _bank **is array** (byte range 0 to 132) **of** integer;
**type** vector **is array** (integer range <>) **of** real;
Simbolul ' < >' poate fi vazut ca o 'locatie' pentru index, care va fi 'umpluta' atunci cind a folosit vectorul. De exemplu, un obiect a declarat vector de 20 de elemente astfel:

vector(1 to 20).

Exista doua tipuri predefinite de tablouri (vectori):

**type** string **is array** (positive **range** <>) **of** character;
**type** bit vector **is array** (natural **range** <>) of bit;

Tipul 'bit vector' a folosit la simularea sistemelor digitale.

Un element al tabloului poate referit prin indexarea numelui tabloului. De exemplu, fie a si b doua tablouri de dimensiuni 1 si, respectiv, 2. Atunci a(1) si b(l,1) se refera la elementele tablourilor. Se pot referi si parti continue din tablouri, de exemplu a(8 to 15) este un vector de 8 elemente care a inclus in vectorul a.

Fie un tip de tablou declarat astfel:

**type** a **is array** (1 **to** 4) **of** character;

si vrem sa scriem un vector de acest tip, care sa contina elementele 'f, 'o', 'o', 'd' in aceasta ordine. Putem scrie astfel: ('f' 'o' 'o' 'd') in care elementele sunt in ordinea crescatoare a indicelui.

O alta varianta ar fi aceasta: (1=>'f',3=>'o',4=>'d',2=>'o')

In acest caz, este dat explicit indicele pentru fiecare element, care pot fi deci in orice ordine. Ambele variante pot fi folosite in cadrul scrierii aceluiasi tablou. Se poate folosi si cuvintul 'others', in locul indexului, care indica spre o valoare folosita pentru toate elementele care nu au fost mentionate explicit.

De exemplu: ('f',4=>'d',others=>'o')

*f)Inregistrari:*

Inregistrarile in VHDL sint colectii de elemente, care pot avea tipuri diferite. Sintaxa este:

tip_inregistrare ::=
**record**
element
{element}
**end record**
element ::= lista_identificatori : subtip_element;
lista identificatori ::= identificatorf{,identificator}
subtip element ::= subtip

Exemple:

**type** instruction **is**
**record**
op_code : processor_op;
address mode : mode;
operand1,operand2 : integer range 0 to 15;
**end record;**

Fie o inregistrare r si un cimp f in aceasta inregistrare. Acel cimp poate fi referit cu numele 'r.f'

*g)Subtipuri:*

Subtipurile reprezinta tipuri de baza restrictionate. Sintaxa este:

declaratie subtip ::= **subtype** identificator **is** subtip;
subtip ::= [functie de rezolutie] marcaj_tip [constringere]
marcaj_tip ::= nume_tip I nume-subtip
constringere ::= mult:wq :ime_inintervalimultime-index

Exista doua feluri de subtipuri:

1. Subtipul reprezinta valorile unui tip scalar, restrinse la un interval.

**subtype** pin_count **is** integer **range** 0 **to** 400;
**subtype** digits **is** character **range** '0' **to** '9';
2 . Subtipul reprezinta valorile unui tablou cu indici nedefiniti, restrins prin definirea indicilor. **subtype** id **is** string(1 **to** 20);
**subtype** word **is** bit_vector(31 **downto** 0);

Exista doua subtipuri numerice predefinite:

**subtype** natural **is** integer range 0 **to** MAXINT
**subtype** positive **is** integer range 1 **to** MAXINT

*h)Declaratii de obiecte:*

In VHDL exista trei clase de obiecte: CONSTANTE, VARIABILE si SEMNALE. Vom discuta despre constante si variabile. O constanta este un obiect care este initializat cu o valoare cind e creat, care nu mai este ulterior modificata. Sintaxa este:

constanta::=constant lista_identificatori:subtip[:=expresie];

Declaratiile de constante care nu au expresie de initializare se numesc 'constante aminate', si nu apar decit in declaratiile de package. Valoarea de initializare trebuie data bin package body corespunzator.

Exemple:

**constant** e : real := 2.13455;

**constant** max-size : natural;

B Variabila e un obiect a carui valoare se poate modifica.

Sintaxa este:
variabila::=variable lista_identificatori:subtip[:=expresie];

Daca expresia de initializare lipseste, i se asociaza variabilei o valoare implicita. Pentru tipurile scalare implicita este cea mai din stinga valoare pentru acel tip: prima dintr-o enumerare, cea mai mica dintr-un interval dat in ordine ascendenta, cea mai mare dintr-un interval dat in ordine descendenta. Daca variabila este un tip compus, valoarea implicita este formata din toate valorile implicite alr tipurilor componente.

Exemple:

**variable** count : natural := 0;
**variable** trace : trace_array;

Daca 'trace array' este un vector de booleeni, atunciu valoarea initiala a lui 'trace' este un vector de elemente cu valorile 'false'.

Mind dat un obiect, a posibil sa se defineasca nume alternative pentru obiect sau pentru o parte din el.
Sintaxa este:

nume_alternativ::=alias identificator:subtip **is** nume;

Exemple:

**variable** instr : bit_vector(31 **downto** 0);
alias op code : bit vector(7 **downto** 0) **is** instr(31 **downto** 24) ;

In acest exemplu, numele 'op code' este un nume alternativ pentru primii 8 biti din 'instr'.

*i) Atribute:*
Tipurile si obiectele declarate in VHDL pot avea asociate informatii suplimentare, numite atribute. Un atribut este referit cu '".

Atribute pentru orice tip sau subtip scalar T:

| ATRIBUT | REZULTAT |
|---|---|
| T'left | valoarea stinga a lui T |
| T'right | valoarea dreapta a lui T |
| T'low | valoarea cea mai mica a lui T |
| T'high | valoarea cea mai mare a lui T |

Atribute pentru orice tip T, X fiind membru al lui T si N un intreg:

| ATRIBUT | REZULTAT |
|---|---|
| T'pos(X) | pozitia lui X in T |
| T'val(N) | valoarea de la pozitia N din T |
| T'leftof(X) | valoarea de la stinga lui X in T |
| T'rightof(X) | valoarea de la dreapta lui X in T |
| T'pred(X) | valoarea imediat mai mica decit X in T |
| T'succ(X) | valoarea imediat mai mare decit X in T |

Atribute pentru orice tablou sau obiect A, N fiind un intreg cuprins intre 1 si numarul de dimensiuni ale lui A:

| ATRIBUT | REZULTAT |
|---|---|
| A'left(N)v | al.stinga a interv.index pt.dim.N in A |
| A'right(N) | val.dreapta a interv.index pt.dim.N in A |
| A'low(N) | val.cea mai mica int.index pt.dim.N in A |
| A'high(N) | val.cea mai mare int.index pt.dim.N in A |
| A'range(N) | int.de valori pt.index pt.dim.N in A |
| A'reverse_range(N) | int.valori pt.index, in ordine inversa |
| A'length(N) | lungimea intervalului index pt.dim.N in A |

## 3.Epresii si operatori

Iata lista operatorilor din VHDL in ordinea descrescatoare a precedentei

| | | | | | |
|---|---|---|---|---|---|
| ** | abs | not | | | |
| * | / | mod | rem | | |
| +(unar) | -(unar) | | | | |
| + | - | & | | | |
| = | /= | < | <= | > | >= |
| and | or | nand | nor | xor | |

Operatorii logici AND, OR, NAND, NOR, XOR si NOT opereaza pe valori de tip bit sau boolean si pe vectori (tablouri unidimensionale) de aceste tipuri. Operatiile pe vectori se aplica intre elementele corespondente ale fiecarui vector, rezultatul fiind tot un vector.

Operatorii relationali =, /=, < , < =, > si > = se aplica pe operanzi de acelasi tip si au ca rezultat o valoare booleana. Operatorii de egalitate (= si /=) permit operanzi de orice fel de tip. Ceilalti operatori permit numai operanzi de tip scalar sau vectori uni-dimensionali de tip discret. Operatorii + si - (unari sau binari) au semnificatia uzuala pentru operanzi numerici. Operatorul de concatenare (&) se aplica pe vectori, rezultatul fiind un nou vector format din alipirea celor doi vectori operanzi. Se pot concatena un element si un vector, sau doua elemente care prin concatenare formeaza un nou vector de dimensiune 2. '

Operatorii de inmultire si impartire (* si /) lucreaza pe numere intregi, reale si operanzi de tip fizic. Operatorii cit si rest (MOD si REM) lucreaza numai pe tipuri intregi. Operatorul valoare absoluta (ABS) lucreaza cu operanzi de orice tip. Operatorul ridicare la putere (**) poate avea un intreg sau un real in partea stinga, dar trebuie sa aiba neaparat un intreg in partea dreapta. Un exponent negativ a permis numai daca operandul din partea stinga a tip real.

## 4.Instructiuni secventiale

*a)Instructiunea de atribuire:*

Sintaxa este:

instr atribuire::=tinta:=expresie;
tinta::=nume l lista-elemente

Partea stinga si partea dreapta ale atribuirii trebuie sa aiba acelasi tip de baza. Daca in partea stinga se afla o lista de elemente, atunci ele trebuie sa fie nume de obiecte, iar in partea dreapta trebuie o valoare compusa de acelasi tip. ?Mai intii se se evalueaza lista, apoi expresia si, in final, componentele expresiei sint atribuite.

Exemplu: ( a => r.b , b => r.a ) = r

*b)Instructiunea IF:*

Sintaxa este:

```
instr_if::=if conditie then
                secventa_instr
                { elsif conditie then
                secventa_instr}
        [else   secventa_instr]
        end if;
```

*c)Instructiunea CASE:*

Sintaxa este:

```
instr_case::=   case expresie is
                        alternativa case
                        {alternativa case}
                end case;
alternativa_case::=when alternative => secventa_instr
```

alternative::=alternativa{, alternativa}
alternativa::=expresie_simpla
interval discret
nume_simplu_element
**others**

Expresia de selectie trebuie sa aiba tip discret sau sa fie un vector de caractere. Alternativele trebuie sa fie distincte, deci nu trebuie sa existe valori duplicate. Toate valorile trebuie sa fie prezente in lista de alternative, sau OTHERS trebuie inclusa ca ultima alternativa. Daca expresia are ca valoare un vector, atunsi alternativele trebuie sa fie siruri de caractere sau siruri de biti.

Exemple:

```
case element colour of
        when red => instr. red;
        when green I blue => instr. green, blue;
        when orange to turquoise => instr;
end case;

case    opcode of
        when X"00" => perform add;
        when X"01" => perform substract;
        when others => signal_illegal_opcode;
end case;
```

*d)Implementarea ciclurilor:*
Sintaxa este:
ciclu::= [eticheta:]
[schema de iteratie] **loop**
secventa_instr
**end loop** [eticheta];
schema_de iteratie::= **while** conditie
| **for** specificare_parametrii_bucla
specificare_parametrii_bucla::= identificator in interval_discret

Daca lipseste schema de iteratie, avem cazul unei bucle infinite:

```
 loop ceva;
  end loop;
```

Exemplu ciclu WHILE:

```
while index < length and str(index) /= '' loop
        index := index + 1;
end loop;
```

Exemplu ciclu FOR:

```
        for item in 1 to last item loop table(item) := 0;
        end loop;
```

Exista doua tipuri de instructiuni aditionale care pot fi folosite in cadrul unui ciclu. Cu instructiunea 'next' se termina executia iteratiei curente si se trece la iteratia urmatoare. Cu instructiunea 'exit' se termina executia iteratiei curente si se iese din ciclu. Sintaxa este:

instr next::=next[eticheta][when conditie];
instr exit::=exit[eticheta][when conditie];

Daca este omisa eticheta, instructiunile se executa in cadrul buclei celei mai apropiate care le cuprinde, altfel se executa in cadrul buclei specificate de eticheta. Daca clauza WHEN este prezenta, dar conditia asociata a evaluata la FALSE, iteratife continua normal.

Exemple:

```
for i in 1 to max str_len loop
        a(i) .= buf(i);
```

```
        exit when buf(i) = NUL;
end loop;

outer loop: loop
        inner_loop: loop do_something;
        next outer_loop when temp = 0;
        do_something-else;
        end loop inner_loop;
end loop outer loop;
```

*e)Instructiunea vida:*
Instructiunea vida nu are nici un efect. Este folosita cind se doreste sa se arate explicit ca in unele cazuri nu se executa nimic. E folosita frecvent in instructiuni CASE, unde trebuie listate toate valorile alternative posibile ale unei expresii, cu toate ca pentru unele dintre ele nu trebuie sa se execute nimic. Sintaxa este: case controller command is when forward => engage motor forward; when reverse => engage-motorreverse; when idle => null; end case;

*f)Asertiuni:*
Asertiunile sint folosite pentru verificarea unor conditii
specificate si raportarea in caz ca acestea nu sint verificate.

Sintaxa este:
asertiuni::=assert conditie
        [report expresie]
        [severity expresie];

Daca a prezenta clauza REPORT, rezultatul expresiei trebuie sa fie un sir de caractere. Acesta reprezinta un mesaj ce va fi raportat in cazul in care conditia a evaluata la FALSE. Daca este omis, mesajul implicit este "Assertion violation". Daca e prezenta clauza SEVERITY atunci tipul expresiei trebuie sa fie 'severity-level'. Daca a omisa, valoarea implicita este 'error'. Un simulator termina executia daca o asertiune nu se mai respecta si valoarea specificata de 'severity' a mai mare decit un prag dat, dependent de implementare. De obicei acest prag este dat de catre utilizator.

## 5.Subprograme si pachete

Subprogramele VHDL pot fi proceduri sau functii. Exista si modalitati de incapsulare a datelor in package-uri(engl.pachet).

*a)Proceduri si functii:*

Sintaxa este:
subprogram::=specificare-subprogram;
specificare_subprogram::= procedure nume[(lista parametrii formali)] function nume[lista parametrii-formali)] return marcaj tip

Aceste declaratii de subprograme sint folosite de obicei in specificatii de package, unde corpul subprogramului este dat in corpul package-ului.

lista_parametrii_formali::=lista-interfata parametrii

lista_interfata::=element_interfata{;element interfata}

element_interfata::=declaratie_interfata

declaratie_interfata::=declaratie_constante_interfata declaratie -semnale_interfata declaratie variabile interfata

declaratie_constante_interfata::=
        [constant]lista_identificatori:**in** subtip[:=expresie_statica]

declaratie variabile interfata::=

[variable]lista_identificatori:[mod]subtip[:=expresie_statica]

Exemple:

1)
```
procedure reset;
…
rest; --apel procedura
```

2)
```
procedure increment_reg ( variable reg : inout word_32;
                          constant incr : in integer := 1 );
```

In al 2-lea exemplu, procedura are 2 parametrii: 'reg' si 'incr'. Modul lui 'reg' este INOUT, ceea ce inseamna ca 'reg' poate fi citit si asignat. Alte moduri posibile sint: IN (parametrul poate fi numai citit) si OUT, parametrul nu poate fi decat asignat. Daca modul este INOUT sau OUT, atunci cuvintul VARIABLE poate lipsi, fiind subinteles. Daca modul este IN, atunci CONSTANT poate lipsi. Un apel de procedura contine lista parametrilor actuali, data sub aceiasi forma ca la vectori.

Exemple:

```
increment reg ( index_reg , offset-2 );          --adaug o
valoare
        increment reg ( prog counter );   -
        incrementez(1=implicit)
sau
        increment_reg ( incr => offset-2 , reg => index_reg );
        increment_reg ( reg => prog_counter );
```

Exemplu de functie:

```
function byte_to_int ( byte : word-8 ) return integer;
```

Pentru functii, modul trebuie sa fie IN, si nu mai trebuie specificat. Daca nu se specifica clasa parametrului, se considera implicit CONSTANT.

Sintaxa pentru corpul unui subprogram este:

```
corp_subprogram::=
        subprogram is
                declaratii subprogram
                        begin
                        instructiuni_subprogram
                        end [nume];
declaratii_subprogram::={declaratie subprogram}
instructiuni_subprogram::={instructiune secventiala}
declaratie-subprogram::=subprogram
                                corp_subprogram
                                tip
                                subtip
                                constanta
                                variabila
                                alias
```

Numele listate dupa partea declarativa a subprogramului sint locale si nu sint vizibile in afara subprogramului. Ele chiar "ascund" semnificatia celor declarate in afara. Dupa apel, se executa instructiunile subprogramului pina se ajunge la sfirsit sau la un RETURN:

```
insructiunea-return::=RETURN[expresie];
```

Functiile nu trebuie sa aiba efecte laterale. O consecinta importanta a acestei reguli este ca functiile pot fi apelate fara sa aiba efect asupra mediului apelant.

Exemplu corp functie:

```
function byte_to_int ( byte : word_8 ) return integer is variable result : integer := 0
        begin
                for index in 0 to 7 loop
                        result := result * 2 + bit'pos ( byte ( index) );
                end loop;
```

```
            return result;
        end byte to_int;
```

REDENUMIRE: exista posibilitatea ca 2 subprograme sa aiba acelasi nume, diferentiindu-se prin numarul sau tipul parametrilor.

Exemple:

```
        function check limit ( value : integer ) return boolean;
        function check-limit ( value : word-32 ) return boolean;
        …
        testl := check-limit ( 4095 )                -- apel prima functie
        test2 := check-limit ( X"0000_OFFF")         --apel a doua functie
```

Numele subprogramului poate fi simbol de operator. Se pot redefini astfel operatori pentru tipuri not de operanzi.

Exemple:

```
 function "+"(a,b : word_32) return word_32 is
        begin
        return inttoword_32(word_32_to_int(a) + word-32 to int(b));
        end "+";
        …
        X"1000_0010" + X"0000_FFDO"          --e apelat noul operator
        "+"(X"1000_0010",X"0000_FFDO")       --notatie echivalenta
```

*b)Package-uri si declaratia corpului pentru package*

Un package - pachet - este o colectie de tipuri, constante, subprograme si eventual alte constructii care se folosesc de obicei pentru a implementa un anumit serviciu sau pentru a izola un anume grup de elemente intre care exista o legatura. Mai mult decit atit, detaliile implementarii unui package pot fi "ascunse" fata de utilizatorii acestuia, lasind vizibila pentru restul lumii numai partea de interfata.

Un package poate fi impartit in doua parti: partea de declaratii - care defineste interfata package-ului - si corpul package-ului, in care se define celelalte detalii. Corpul poate fi omis daca nu sint alte detalii nespecificate in partea de declaratii. Sintaxa declararii unui package este urmatoarea:

```
declaratie_package ::=
        package identificator is
                partea_declarativa_package
        end [ nume-simplu package ] ;

partea_declarativa_package ::=[ element_parte_declarativa ]

element_parte_declarativa ::=declaratie subprogram
        declaratie_tip
        declaratie_subtip
        declaratie_constanta
        declaratie_alias
        clauza_use
```

Declaratiile definesc acele elemente care vor fi vizibile pentru utilizatorii packageului si totodata in interiorul corpului package-ului. Trebuie notat de asemenea ca mai sint si alte tipuri de declaratii ce mai pot fi incluse, dar nu vor fi discutate aici.

Exemple:

```
package data_types is
        subtype address is bit_vector(24 downto 0);
        subtype data is bit _vector(15 downto 0);
        constant vector_table_loc : address;
        function data _to_int(value : data) return integer;
```

```
        function int_to_data(value : integer) return data;
end data_types;
```

In acest exemplu, declararea valorii constantei vector table precum si corpurile celor doua functii sint aminate, drept pentru care este necesar si corpul package-ului.

Sintaxa pentru corpul unui package este:

```
corp_package ::=
        package body nume_simplu package is
                parte declarativa_corp_package
        end [ nume-simplu package ] ;

        parte declarativa_corp package ::= [element corp package ]
        element_package ::=
                declaratie subprogram
                        corp_subprogram
                        declaratie_subtip
                        declaratie_constanta
                        declaratie_alias
                        clauza_use
```

In corpul package-ului poate fi inclus corpul unui subprogram, in timp ce in interfata package-ului nu poate fi inclusa decit partea declarativa a unui subprogram.

Exemple:

```
package body data types is
        constant vector_table_loc : address = X"FFFFF00";
        function data_to_int(value : data) return integer is corpul functiei data_to_int
        end data_to_int;

        function int_to_data(value : integer) return data is corpul functiei int_to_data;
        end int_to_data;
end data_types;
```

In corpul package-ului se specifica valoarea pentru constante si sint date corpurile functiilor. Declaratife de subtip nu sint repetate deoarece cele facute in partea declarativa a package-ului sint vizibile si in corpul package-ului.

Odata package-ul declarat, elementele sale pot fi referite prin prefixarea numelui for cu numele package-ului. Pentru exemplele anterioare:

```
variable PC : data_types.address;

int_vector_loc := data_types.vector_table_loc * 4 int_level;

offset := data types.data to int(offset reg);
```

Adeseori este convenabil sa poti referi elementele dintr-un package fara a le mai prefixa cu numele acestuia. Pentru aceasta se foloseste clauza use intr-o parte declarativa. Sintaxa acesteia este:

```
clauza_use ::= use nume selectat { , nume_selectat } ;
nume selectat ::= prefix . sufix
```

Efectul utilizarii acestei clauze este ca toate numele listate pot fi folosite apoi fara a mai fi nevoie sa le, prefixam cu numele package-ului. Se poate folosi sufixul *all* pentru referirea neprefixata la toate elementele declarate intr-un package.

Exemple:

```
use data types.all;
```

## 6. Structura sistemelor in VHDL

*6.1. Entitati*

Un sistem digital este format dintr-o colectie ierarhica de module. Fiecare modul are un set deporturi care constituie interfata lui cu exteriorul. Un asemenea modul se numeste in VHDL entity.

Sintaxa pentru descrierea unei entitati este:

```
entitate ::=        entity identificator is
                            antet_entitate
                            declaratii_entitate
                            [begin
                             instructiuni_entitate ]
                             end [ nume_simplu_entitate ] ;
antet entitate ::=
                    [ clauze_formale_generice ]
                    [ clauze_formale_porturi ]
```

clauze generice :.= **generic** ( lista generic ) ;
lista generic ::= lista interfata generica
clauze_porturi ::= **port** ( lista porturi ) ;
lista_porturi::= lista interfata porturi
declaratie_entitate ::= { elemente_declaratie_entitate}

In partea declarativa a unei entitati se declara elementele care vor fi folosite in implementarea acesteia. De obicei aceste declaratii sint incluse in partea de implementare. De obicei, declaratiile optionale din partea declarativa a unei entitati sint pentru a defini comportamente particulare ale entitatii.

Constantele generice pot fi folosite pentru a controla structura si comportamentul entitatii, iar porturile pentru a specifica canalele de intrare si iesire ale entitatii. Ele sint de tip **constant.** Valoarea actuala a unei constante generice este transmisa in momentul in care entitatea este folosita.

Porturile unei entitati sint de tip signal. Sintaxa este:

```
interfata_signal ::=
        [ signal ] lista-identificatori : [ mod ] subtip [ bus]
        [ := expresie statica ]
```

Cuvintul **bus** poate fi folosit daca portul poate fi conectat la mai mult de o iesire . La fel ca la constantele generice, semnalele care trebuie conectate la porturi sint specificate cind entitatea este folosita.

Exemple:

```
entity processor is
        generic (max_clock_freq: frequency := 30 MHz);
        port (clock: in bit; address: out integer; data: inout word_32; control: out proc control; ready in bit
);
        end processor;
```

In acest caz constanta generica max clock freq este folosita pentru a specifica comportamentul in timp al entitatii. In codul care descrie comportamentul entitatii, se va folosi aceasta valoare pentru a determina intirzierile in schimbarea valorilor semnalelor. In exemlul urmator se arata cum pot fi folositi parametrii generici pentru a specifica o clasa de entitati de structura variabila:

```
entity ROM is
        generic (width , depth: positive);
        port (enable : in bit; address: in bit_vector(depth - 1 downto 0; data : out bit_vector(width - 1
        downto 0);
end ROM;
```

Aici cele doua constante generice sint folosite pentru a specifica numarul de biti de date si respectiv de adresa pentru memoria "read-only". Trebuie notat ca nu este data nici o valoare implicita pentru aceste

constante. Aceasta inseamna ca atunci cind entitatea este folosita ca o componenta trebuie actualizate valorile acestora.

Exemplu de declaratie de entitate fara constante sau porturi generice:

**entity** test bench **is**
**end** test_bench

Cu toate ca acest exemplu pare la prima vedere sa nu aiba sens, de fapt el ilustreaza o utilizare des intilnita a entitatilor:

O entitate "top-level" pentru un design in curs de testare (design under test - DUT) este folosita ca o componenta intr-un circuit de tip "banc de lucru" impreuna cu o alta entitate generator de test (test generator - TG). Nu este nevoie de conexiuni externe, deci nu exista porturi.

*6.2. Arhitecturi*

Odata specificata interfata entitatii, una sau mai multe implementari ale acesteia pot fi descrise in sectiunea **architecture,** care reprezinta corpul entitatii. Pot fi date mai multe arhitecturi care descriu aceeasi entitate. De exemplu, o arhitectura poate descrie pur si simplu comportamentul entitatii, in timp ce alta poate descrie aceeasi entitate ca o colectie ierarhica de componente. In aceasta sectiune vom prezenta numai descrierile structurale.

arhitectura ::= **architecture** identificator **of** nume **entitate is** parte_declarativa_arhitectura
**begin**
instructiuni arhitectura
**end** [ nume arhitectura ];

parte_declarativa_arhitectura ::= { bloc declaratii }
instructiuni-arhitectura ::={ instructiune-concurenta }

bloc_declaratii ::= declaratie_subprogram
corp_subprogram
declaratie_tip
declaratie_subtip
declaratie_constanta
declaratie_alias
declaratie_signal
declaratie_componente
declaratie_configuratie
clauza_use

instructiune_concurenta ::=
instructiune_bloc
instructiune-instantiere_componenta

Declaratiile dintr-o arhitectura definesc elemente care vor fi folosite pentru a construi descrierea designului. In particular, semnalele si componentele pot fi declarate aici si folosite pentru a construi o descriere structurala, dupa cum a fost exemplificat.

*a)Declaratiile de semnal*

Semnalele sint folosite pentru a conecta submodule. Ele sint declarate cu sintaxa urmatoare:

declaratie_signal ::=
**signal** lista-identificatori : subtip [ tip-signal ][ :=expresie ] ;
tip-signal ::**=** register | bus

Expresia din declaratie este folosita pentru a initializa semnalul in timpul fazei de initializare a simulari. Daca expresia este omisa, atunci va fi asignata o valoare implicita.

*b)Blocuri*

Submodulele dintr-o arhitectura pot fi descrise ca blocuri. Un bloc este o unitate ce contine propria interfata, conectat cu alte blocuri sau porturi prin semnale. Sintaxa este:

bloc ::= eticheta_bloc
**block** [ (expresie_garda) ]
antet_bloc
parte_declarativa_bloc
**begin**
instructiuni bloc
**end block** [ eticheta bloc ] ;

antet_bloc ::=
[ clauza generica [ map_generic ;] ]
[ clauza port ]
[ map port ; ] ]

map-generic ::=
**generic map** ( lista asociatie generic )

map_port ::=
**port map** ( lista asociatie port )
parte_declarativa_bloc ::= declaratie bloc }

instructiuni_bloc ::=
{ instructiune concurenta }

Antetul blocului defineste interfata cu blocul respectiv in mod similar cu modul in care antetul entitatii defineste interfata unei entitati. Lista de asociatie generica specifica valorile pentru constantele generic evaluate in blocul inconjurator sau in arhitectura. Lista de asociatie a porturilor specifica care dintre semnalele sau porturile blocului inconjurator sau arhitecturii sint conectate la porturile blocului. Trebuie notat ca instructiunile dintr-un bloc pot de asemenea contine instructiuni bloc, astfel ca un proiect (design) poate fi privit ca o ierarhie de blocuri ce contine descrierea comportamentului la baza ierarhiei.

Ca si exemplu, sa presupunem ca se doreste descrierea structurii unei arhitecturi a entitatii procesor descrise intr-o sectiune anterioara. Daca separam procesorul intr-o structura de control si o sectiune ce se ocupa cu calea datelor, putem scrie o descriere ca o pereche de blocuri inlantuite.

Unitatea de control are porturile clk, bus control si bus ready, care sint conectate la porturile entitatii procesor. Exista de asemenea un port de iesire pentru controlul caii de date, acesta fiind conectat la un semnal declarat in sectiunea de arhitectura. Acest semnal este conectat de asemenea la un port de control din blocul caii de date. Porturile de date si de adrese ale blocului caii de date sint conectate la porturile corespunzatoare ale entitath respective. Avantajele acestei abordari modulare se refera la faptul ca odata precizate blocurile, fiecare dintre acestea poate fi dezvoltat separat. Aici putem observa aplicarea cu succes a separarii partii de interfata in mod clar, lucru ce permite astfel dezvoltarea in paralel.

**architecture** block structure **of** processor **is**
**type** data-path-control **is** .... ;
signal internal control : data-path-control;
**begin**
control unit: **block**
**port** (clk: **in** bit; bus _control : **out** proc_control;bus _ready: **out** data-path-control);
**port map** (clk => clock, bus_control => control, bus_ready => ready,
control => internal control);
declaratii pentru control_unit
**begin**
instructiuni pentru control-unit
**end block** control unit;
data_path: **block**
**port** (address: **out** integer;data: **inout** word 32; control: **in** data_path_control);
**port map** (address => address, data=> data, control => internal-control);
declaratii pentru data_path

```
                    begin
                    instructiuni pentru data_path;
                    end block_data_path;
end block_structure;
```

*c)Declaratii de componente*

O arhitectura poate folosi de asemenea si alte entitati descrise separat si plasate in biblioteci de proiectare. Pentru aceasta, intr-o arhitectura trebuie sa se declare o componenta la care ne putem gindi ca la un model ce defineste o entitate de proiectare virtuala. Aceasta componenta virtuala va fi instantiata mai tirziu in cadrul arhitecturii.

Sintaxa este:

componenta ::=
**component** identificator
[ clauza generic-locala ]
[ clauza port locala ]
**end** component;

Exemple:

```
component nand3
        generic (Tpd : Time := 1 ns );
        port (a,b,c :in logic level; y :out logic-level);
        end component;

component read_only_memory
        generic(data bits, addr_bits : positive);
        port(   en : in bit;
                addr : in bit _vector(depth-1 downto 0);
                data : out bit_vector(width-1 downto 0);
        end component;
```

Acest exemplu declara o poarta cu 3 porturi si un parametru generic care specifica intrizierea de propagare. Instante diferite pot fi folosite apoi cu intirzieri de propagare diferite. Al doilea exemplu declara o componenta de memorie "read-only" ale carei dimensiuni sint dependente de constante generice.

*b)Instantierea componentelor*

O componenta definita intr-o arhitectura poate fi definita astfel:

instructiune instantiere_componenta::=
eticheta_instantiere: nume_componenta
[ map generic ]
[ map port ] ;

Aceasta inseamna ca arhitectura contine o instanta a unei componente pentru care sint specificate valorile actuale ale constantelor generice. De asemenea, porturile componentei respective sint conectate la semnalele corespunzatoare sau la porturile entitatii.

enable gate: nand3 **port map** (a => enl, b => en2, c => int_req, y => interrupt);

```
 parameter rom: read_only_memory
          *generic ma* (data bits => 16, addr_bits => 8);
          port map (en => rom_sel, data => param, addr => a(7 downto 0) );
```

In primul exempla nu este specificat map-generic, asa ca constanta generica Tpd timpul de propagare) este folosita. In al doilea exemplu sint specificate valorile pentru porturile de adrese si date.

## 7. Comportamente VHDL

Comportamentul unui sistem digital poate fi descris in termenii unui limbaj de programare. Aspectele legate de paradigma secventiala a VHDL ca limbajj de programare au detaliate in capitolul 2. In acest capitol vom incerca sa descriem modul in care aceste caracteristici ale VHDL sint extinse pentru a include instructiuni ce permit modificarea valorilor unor semnale si mijloace de a raspunde la modificarea valorilor unor semnale.

*a)Atribuiri pentru semnale*

Sintaxa este:

instructiune_atribuire_semnal ::=
destinatie <= [ **transport** ] forma unda ;
destinatie ::= nume | nume-compus
forma unda ::= element forma unda {, element forma unda }

element_forma_unda::= valoare_expresie [ **after** expresie_timp | **null** [ **after** expresie time ]

Destinatia trebuie sa reprezinte un semnal sau un grup de semnale. Daca expresie timp ce corespunde intirzierii este omisa, atunci se va considera valoarea implicita de 0 fs. Aceasta inseamna ca se va considera ca tranzactia a avut loc simultan cu executia operatiei de atribuire.

Asociem fiecarui semnal valoarea prevazuta a formei sale de unda. Atribuirile introduc tranzitii in aceasta forma de unda. De exemplu:

s **<=** ‘0' **after** 10 ns;

Aceasta atribuire va avea ca efect trecerea la '0' a valorii semnalului s la 10 ns dupa executia acesteia. Putem reprezenta valorile prevazute ale unui semnal dat, desenind tranzitiile care apar de-a lungul unei axe pe care consideram timpul. Spre exemplu, daca atribuirea de mai sus ar fi executata la momentul t=5 ns, atunci reprezentarea grafica ar fi cea de mai sus:

Cind simularea a atins momentul de timp t=15ns, tranzitia corespunzatoare va fi procesata si valoarea semnalului va fi modificata. Sa presupunem in continuare ca la momentul de timp t=16 ns, executam atribuirea urmatoare:

s **<=** ‘1' **after** 4 ns, '0' after 20 ns;

Vor apare astfel doua not tranzitii. Trebuie notat faptul ca atunci cind intr-o atribuire sint prezente mai multe tranzitii, momentele de timp ce specifica intirzierea trebuiesc sa fie in ordine crescatoare.

Este interesant de notat ce se intimpla in cazul executiei unor atribuiri de valori pentru semnale pentru care existau si alte atribuiri anterioare. Aici distingem doua cazuri, dupa cum cuvintul rezervat *transport* este sau nu inclus in instructiunea de atribuire. A stfel

daca *transport* apare, atunci intirzierea se numeste intirziere de transport.In aceasta situatie, orice alte tranzitii anterioare tranzitiei cauzate de executia atribuirii curente vor fi sterse. Este ca si cum tranzitiile anterioare ar fi suprascrise de tranzitia cauzata de atribuirea curenta.

daca nu se foloseste transport**,** atunci vom avea de-a face cu al doilea tip de intirziere ce se numeste intirziere inertiala. Acest tip de atribuire a valorii unui semnal corespunde dispozitivelor fizice care nu prezinta un raspuns la excitarea cu un impuls de durata mai mica decit intirzierea pe care o necesita propagarea semnalului de excitatie pina la iesire. In aceasta situatie, vor fi sterse toate acele tranzitii care ar fi trebuit sa aiba loc inainte de tranzitia nou introdusa si vor ramine numai cele care vor avea loc in timp de la momentul de timp corespunzator tranzitiei nou introduse inainte.

Cind se executa o atribuire a valorii unui semnal in care se specifica mai multe tranzitii, prima tranzitie va fi considerata de tip inertial, iar restul de tip intirziere de transport.

*b)Procese. Instructiunea wait*

Unitatea de baza intr-o descriere comportamentala de tip VHDL este procesul. Un proces este un corp de cod cu executie secventiala ce poate fi activat ca raspuns la o schimbare de stare. Atunci cind se executa mai mult de un proces la un moment dat, executia va avea loc concurent. Procesele sint specificate conform sintaxei urmatoare:

instructiune_proces ::= [ eticheta proces : ]
**process** [ ( lista semnale ) ]
parte declarativa proces
**begin**
instructiuni_proces
**end process** [ eticheta proces ] ;

parte_declarativa_proces :.= { element declaratie proces }

element_declaratie_proces ::= declarare subprogram
corp_subprogram
declarare_tip
declarare_subtip
declarare_constante
declarare_variabile
declarare_alias
clauza_use

instructiuni proces ::= { instructiune_secventiala }

instructiune secventiala ::= {instructiune wait
instructiune_atribuire
instructiune_atribuire_semnal
instructiune_atribuire_variabile
instructiune_if
apel_procedura
instructiune_case
instructiune_loop
instructiune_next
instructiune_exit
instructiune_return
instructiune_nula

O instructiune de tip proces este o instructiune concurenta ce poate fi folosita in corpul unei arhitecturi sau al unui bloc. Partea declarativa defineste elementele ce pot fi folosite local in cadrul procesului. Trebuie notat ca variabilele definite intr-un proces pot fi folosite la pastrarea starii in cadrul unui model.

Un proces poate contine instructiuni de atribuire a valorilor unui semnal dat. Aceste instructiuni de atribuire formeaza impreuna ceea ce se numeste un driver pentru acel semnal. In mod normal ar trebui sa existe un singur driver pentru orice semnal considerat, astfel incit codul ce defineste valorile pe care le poate avea un semnal sa fie restrins in interiorul unui singur proces.

Un proces este activat initial in cadrul fazei de initializare a simularii. Procesul va executa toate instructiunile secventiale si apoi va relua executia de la inceput. Eventual, procesul va fi blocat in executia unei instructiuni de tip wait. Aceasta instructiune are urmatoarea sintaxa:

instructiune wait wait [ clauza-semnale ] [ clauza-conditii ] [clauza timeout];
clauza_semnale::=**on** lista_semnale
lista_semnale::= nume_semnal {, nume-semnal}
clauza_conditii::= **until** conditie
clauza_timeout::= **for** expresie_timp

Lista de semnale a unei instructiuni de tip wait specifica un set de semnale fata de care procesul respectiv este sensibil in timpul perioadei in care este blocat. Atunci cind se petrece un eveniment ce afecteaza oricare din aceste semnale - adica se schimba valoarea semnalului - procesul respectiv va iesi din blocare si va reevalua conditia. Daca conditia este adevarata, sau daca conditia este omisa, atunci executia va continua cu urmatoarea instructiune, altfel procesul se va reintoarce in starea de blocare in care se afla anterior.

Daca lista de semnale este omisa, atunci procesul respectiv este sensibil la valoarea tuturor semnalelor ce apar in expresia conditiei. Cu alte cuvinte, o modificare a valorii unui semnal dih expresia conditiei va duce la "trezirea" procesului respectiv si la reevaluarea conditiei. In cazul in care si lista de semnale si conditia sint omise dintr-un proces, este posibil ca acesta sa ramina blocat un timp nedefinit.

Daca lista de semnale apare explicit in antetul unui proces, atunci se presupune ca acel proces va contine ca ultima instructiune o instructiune de tip wait a carei lista de semnale este identica cu lista specificata in antet. In acest caz este posibil ca procesul sa nu mai contina nici o alta instructiune de tip wait explicita.

Iata in continuare un exemplu:

```
process (reset, clock)
        variable state : bit := false;
begin
        if reset then state := false;
        elsif clock = true then state := not state;
        end if q <= state after prop_delay; -- wait implicit pentru modificarea valorii semnalelor reset si
clock.
end process;
```

In faza de initializare a simularii procesul este activat si se executa atribuirea valorii initiale a semnalului q. Apoi procesul se va bloca la executia instructiunii wait implicite indicate in comentariu. La orice modificare a valorii unuia dintre semnalele reset sau clock, procesul va fi "trezit" si se va reevalua valoarea semnalului q.

Urmatorul exemplu descrie comportarea unui circuit de sincronizare numit "Muller _C". Acest dispozitiv se foloseste in construirea circuitelor logice asincrone. Iesirea dispozitivului este initial '0' si ramine la aceasta valoare pina cind ambele intrari ale circuitului sint ' 1'. In acest moment iesirea devine ' 1' si ramine la aceasta valoare pina cind ambele intrari ale circuitului devin '0', moment in care va lua valoarea '0' din nou. Descrierea VHDL este urmatoarea:

```
muller_c_2: process
begin
        wait until a='1' and b='1';
        q<='1';
        wait until a='0' and b='0';
        q<=’0’;
end process muller_c_2;
```

Din cauza ca acest proces nu contine o lista de semnale este necesara folosirea explicita a instructiunilor de tip wait. In ambele instructiuni wait, lista semnalelor este formata din a si b, derivate din conditia testata.

*c)Instructiuni concurente de atribuire a valorilor unui semnal.*

Adesea, un proces ce descrie un driver pentru un semnal contine o singura instructiune de atribuire a valorii acelui semnal. Limbajul VHDL ofera posibilitatea de a folosi o notatie scurta pentru acest lucru. Aceasta notatie poarta numele de instructiune concurenta de atribuire a valorilor unui semnal. Sintaxa este:

```
instructiune_concurenta_semnal ::=
                [eticheta : ] atribuire_conditionala_semnal | [ eticheta : ] atribuire_selectiva_semnal
```

Pentru fiecare tip de atribuire concurenta a valorii unui semnal exista o varianta corespunzatoare cu lists de semnale in antetul unui proces.

*a)Atribuire conditionala*

Acest tip de atribuire a valorii unui semnal corespunde situatiei in care in cadrul unui proces se face atribuirea de valori unui semnal in interiorul unei instructiuni if. Sintaxa este:

```
        atribuire_conditionals_semnal ::= destinatie <= optiuni forma_unda_conditionala ;

optiuni ::= [ guarded ] [ transport ]

forma_unda_conditionala :.= { forma_unda when conditie else } forma_unda
```

Daca se include cuvintul cheie *transport*, atunci atribuirile corespunzatoare din proces vor fi facute cu intirziere de tip transport.

*b)Atribuire selectiva*

O instructiune de atribuire selectiva a valorii unui semnal corespunde unei instructiuni de atribuire ce se executa in interiorul unei instructiuni de tip case. Sintaxa este:

atribuire_selectiva ::= **with** expresie **select**
destinatie <= optiuni forma_unda_selectie;
forma_unda_selectie ::= { forma_unda when posibilitati ,} forma_unda **when** posibilitati
posibilitati ::= posibilitate { I posibilitate }

Optiunile sint aceleasi ca si pentru atribuirea conditionala. Cu alte cuvinte, daca cuvintul rezervat *transport* apare, atunci atribuirile corespunzatoare din proces vor fi facute cu intirziere de tip transport.

Lista de semnale pentru instructiunea wait se determina in felul urmator: daca in instructiunea de atribuire selectiva nu apare nici o referinta explicita la vreun semnal, atunci lista de semnale pentru acea instructiune wait va fi vida. In caz contrar, lista de semnale a instructiunii wait va contine acele semnale care au fost referite in interiorul instructiunii de atribuire selectiva.

Iata in continuare un exemplu de instructiune de atribuire selectiva:

**with** alu_function **select**
alu-result <= op1 + opt2 **when** alu_add I alu-incr,
opl - op2 when alu_substract,
opl and opt when alu_and,
opl or opt when alu_or,
opt and not opt alu mask;

In acest exemplu, valoarea semnalului alu function este folosita pentru a selecta care dintre instructiunile de atribuire se va executa. Instructiunea contine in lista ei de semnale urmatoarele semnale alu function, op 1, opt, astfel incit oricare dintre aceste trei semnale isi va schimba valoarea, instructiunea de atribuire selectiva isi va relua executia.

## 8.Organizare model

In acest capitol se va arata cum se poate scrie o descriere completa VHDL a unui sistem digital.

*a)Unitati si biblioteci de design*

Descrierile VHDL se scriu intr-un "design file" , se analizeaza cu un compiler care le introduce intrun "design library". O biblioteca este formata din mai multe unitati ("library units"). Unitatile de biblioteca primare sint declarathle de unitati, de package-uri si de configuratie. Unitatile de biblioteca secundare sint corpurile de package-uri si arhitecturile. Aceste unitati depind de specificarea interfetei lor, care se face in unitati primare. Deci unitatile de biblioteca primare trebuie analizate inaintea oricarei unitati secundare corespunzatoare.

Un fisier de design ("design file") contine mai multe unitati de biblioteca. Sintaxa este:
fisier_design ::= unitate design {unitate design }
unitate_design ::= clauza_context unitate_biblioteca
clauza_context ::={context }
context ::= clauza biblioteca I clauza use
clauza biblioteca ::= **library** lista nume logic ;
lista_nume_logic ::= nume logic { , nume_logic }
unitate_biblioteca ::= unitate primara unitate secundara
unitate_primara ::=declaratie_entitate declaratie-configuratie declaratie_package
unitate_secundara ::= arhitectura I corp_package

Bibliotecile sint referite folosind identificatori numiti NUME LOGICE. Numele logic trebuie tradus de sistemul de operare gazda in alt nume, dependent de implementare.
De exemplu, bibliotecile pot fi niste fisiere de baze de date, numele logic reprezentind numele unui fisier de baze de date. Unitatile dintr-o biblioteca pot fi referite prin prefixarea numelui for cu numele logic al bibliotecii. "ttl lib.ttl_10" se poate referi h unitatea "ttl_10" din biblioteca "ttl lib".

Clauza care precede fiecare unitate specifica ce alte biblioteci sau package-uri sint folosite. Scopeul numelor specifcate in clauza de context se intinde pina la sfirsitul unitatii de design.
Exista 2 biblioteci speciale care sint disponibile implicit tuturor unitatilor, si deci nu a necesar sa fie numite in clauze de context. WORK este biblioteca de lucru, in care vor fi plasate de catre analizor unitatile de design curente. Intr-o unitate se pot referi unitatile analizate anterior folosind numele bibliotecii "work". STD este o biblioteca de design care contine package-urile "standard" si "textio". "Standard" contine toate tipurile si functiile predefinite. Toate elementele din acest package sint implicit vizibile, deci nu e necesara clauza use pentru a le accesa.

*b)Configuratii.*

Am aratat cum se poate declara o specificare de componente intr-o descriere structurala, si cum se pot crea instante ale componentelor. Legatura dintre o entitate si componente se face prin declararea de configuratie. Intr-o astfel de declaratie se pot specifics constantele generice actuale pentru componente si blocuri.

Sintaxa este:

declarare_configuratie ::=**configuration** identificator of nume_entitate **is**
parte_declarativa_configuratie
configuratie bloc
**end** [ nume_configuratie ];
parte_declarativa_configuratie ::={ declarare configuratie }
declarare_configuratie ::= clauza_use
configuratie_bloc ::=
**for** specificare bloc
{ clauza_use }
{ conf iguratie }
**end for** *;*

specificare_bloc ::= nume-arhitectura | eticheta_instructiune_bloc
configuratie ::= configuratie_bloc | configuratie_componenta
configuratie_componenta ::= **for** specificare_componenta
[ **use** legatura ; ]
[ configuratie_bloc ]
**end for** *;*
specificare_componenta ::= lista_instantieri nume_componenta
lista_instantieri ::= eticheta_instantiere { , eticheta_instantiere }
**others all**
legatura ::=
aspect_entitate
[ map_generic ]
[ map port ]
aspect_entitate :.= **entity** nume_entitate [(identificator arhitectura)]
***configuration*** nume-configuratie
**open**
map-generic ::= **generic** map (lista asociatii generic)
map port ::= **port map** (lists asociatii port)
Configuratia unei astfel de arhitecturi este:

**configuration** test_config **of** processor **is**
**use** work.processor_types. **all**
**for** block_structure
… … … … … … … …
elemente configuratie
… … … … … … … …
**end for**
end test_config;

In acest exemplu, continutul package-ului "processor types" din biblioteca de lucru curenta este vizibil, iar configurarea de bloc se refera la arhitectura "block structure" a entitatii "processor".
Se pot configura submodulele dintr-o arhitectura. Acestea contin blocuri si instante de componente. Iata o configurare de blocuri pentru arhitectura prezentata mai sus:

```
configuration test_config of processor is
        use work.processor_types. all
        for block_structure
                for control_unit
                … … … … … … … … …
                elemente_configuratie
                … … … … … … … … …
                end for;
                for data_path
                … … … … … … … … …
                elemente_configuratie
                … … … … … … … … …
                end for;
        end_for;
end test_config;
```

Daca submodulul este o instanta a unei componente, configuratia componentei este folosita pentru a lega o entitate de instanta componentei.
Fie blocul "data-path", care contine o instanta a componentei "alu", declarata astfel:

```
data_path: block port(lista porturi);
        port map (lista asociatii);
component alu
        port(function: in alu_function; opl,op2: in bit_vector_32; result: out bit-vector-32);
end component ;
… … … … … … … …
alte declaratii pentru data_path
… … … … … … … …
begin
        data_alu:alu
port map(function=>alu fn,opl=>bl,op2=>b2, result=>alu r);
… … … … … … … …
alte declaratii pentru data_path
… … … … … … … …
end block data_path;
```

Fie biblioteca "project cells" cu entitatea "alu cell" definita astfel:

```
entity alu_cell is
          generic(width : positive);
port( function code : in alu_function; operand1, operand2 : in bit_vector(width-1 downto 0);
          result : out bit_vector(width-1 downto 0); flags : out alu_flags); end alu_cell;
```

Entitatea are o arhitectura numita "behaviour". Aceasta entitate poate fi legata cu instanta "alu" daca porturile operanzi si rezultat pot fi puse in corespondenta cu porturile componentei, porturile "flags" putind fi lasate neconectate.

```
for data_path
        for data_alu:alu
        use entity project cells.alu_cell (behaviour)
        generic map (width=>32)
        port map (function code=>function, operandi=>opl, operand2=>op2, result=>result, flags=> open );
        end for ;
… … … … … … … …
alte declaratii pentru data path
… … … … … … … …
end for ;
```

Daca biblioteca include si o configuratie "alu-struct" pentru o arhitectura a entitatii "alu_cell" atunci configuratia de bloc arata astfel:

```
          for data_path
                  for data alu:alu
                  use configuration project_cells.alu_struct
                  generic map (width => 32)
```

```
                port map (function code=>function, operandi=>opl, operand2=>op2,
                                result=>result, flags=> open );
                end for ;
                … … … … … … …
                alte declaratii pentru data path
                … … … … … … …
        end for ;
```

*c)Exemplu complet*

Prezentam in continuare un fisier de design pentru exemplul din capitolul 1. Fisierul contine mai multe unitati de design care sint analizate in ordine. Prima unitate reprezinta declararea entitatii "count2". Urmeaza 2 entitati secundare - arhitecturi ale entitatii "count2". Urmeaza o alta deeclarare de entitate, un test pentru numarator ("test bench"). Urmeaza o unitate secundara, care reprezinta descrierea structurala a bancii de test. Urmeaza o declarare de configuratie, care face referiri la unitatile de bibliteca definite anterior, din biblioteca de lucru si deci nu a nevoie de clauza de context. Se observa ca entitatea "cunt2" este referita in cadrul configuratiei cu numele "work. count2". Urmeaza o declrare de configuratie cu arhitectura "structure" a entitatii "count2". Ea utilizeaza 2 unitati("misc.t flipflop", "misc. inverter") din biblioteca separata "misc", deci e necesara o clauza de biblioteca.
Aceasta descriere cuprinde toate unitatile intr-un singur fisier. Dar a posibil ca unitatile de design sa fie impartite pe mai multe fisiere ( cazul extrem fiind o unitate intrun fisiewr). Daca exista mai multe fisiere, ele trebuie compilate in ORDINEA corecta, si trebuie recompilate fisierele dependente de o unitate in care s-au operat modificari.

```
--unitate primara: declaratia entitatii count2

entity count2 is
        generic(prop_delay: Time := 10 ns);
        port(clock: in bit; q1,q0 . out bit);
end count2;

--unitate secundara: architectura-comportament pentru count2

architecture behaviour of count2 is
begin count_up: process (clock)
        variable count-value: natural := 0;
        begin   if clock='1' then
                        count value:=(count value+1) mod4;
                        q0<=bit'val(count_value mod 2) after prop_delay;
                        q1<=bit'val(count_value/2) after prop delay;
                end if ; end process count_up;
end behaviour ;

--unitate secundara: architectura-structura pentru count2

architecture structure of count2 is
        component t_flipflop port(ck: in bit; q: out bit);
end component;
component inverter port(a: in bit;y: out bit);
end component;
signal ffo,ffi,inv_ff0: bit;
        begin
                bit-0: t-flipflop port map (ck=>clock, q=>ff0);
                inv: inverter port map (a=>ff0, y=>inv ff0);
                bit 1:t flipflop port map (ck=>inv-ff0, q=>ffl);
                q0<=ff0; q1<=ff1;
        end structure ;

--unitate primara: declaratia entitatii de testare

entity test _count2 is
end test count2;
```

```
--unitate secundara: arhitectura-structura pentru test

architecture structure of test_count2 is
        signal clock,q0,q1: bit;
        component count2
                port(clock: in bit; q1,q0: out bit);
        end component;
        begin counter : count2
        port map (clock => clock, q0 => q0, q1 => q1);
        clock driver : process
                begin wait for 100 ns;
                        end process clock_driver;
                end structure;

-- unitate primara: configuratie cu architectura  comportament

configuration test count2 behaviour of test_count2 is
        for structure pentru test_count2
                for counter : count2
                        use entity work.count2(behaviour);
                end for;
        end for ;
end test_count2_behaviour;

-- unitate primara: configuratie cu architectura

structura library misc;
configuration test_count2_structure of test_count2 is
        for structure --pentru test_count2
                for counter : count2
                                use entity work.count2(structure);
                        for structure --pentru count_2
                                for all :t_flipflop
                                        use entity misc.t_flipflop(behaviour);
                                end for;
                                for all :inverter
                                        use entity misc.inverter(behaviour);
                                end for;
                        end for;
                end for;
        end for;
        end test_count2_structure;
```

## 9.VHDL - notiuni avansate

*a) Rezolutie de semnale si magistrate*

In majoritatea sistemelor digitale magistralele conecteaza un numar de iesiri ale dispozitivelor pe aceeasi linie. VHDL de obicei permiteun singur dispozitiv pentru un semnal. Pentru a modela semnale pentru mai multe dispozitive VHDL foloseste notiunea de "resolved types"(tipuri rezolvate) pentru semnale. Un tip rezolvat include la definire o functie de rezolutie care is valori pentru toate dispozitivele legate la o linie, rezultind valoarea finala a semnalului. Un tip rezolvat pentru un semnal este declarat cu sintaxa pentru subtip:

subtip::=[ nume functie rezolutie ] tip [ constringere ]

Functia de rezolutie trebuie definita anterior. Functia are ca parametru un tablou cu valori de subtipul semnalului si terbuie sa intoarca ca rezultat acelasi subtip. De exemplu:

```
type logic_level is (L,Z,H);
type logic_array is array (integer range < >) of logic_level;
subtype resolved_level is resolve_logic logic_level;
```

In acest exemplu, tipul logic level reprezinta 3 posibile stari ale unui semnal digital : L(low), Z(high-impedance), H(high). Subtipul resolved-level este folosit pentru declararea semanalului rezolvat de acest tip. Functia de rezolutie poate fi implementata astfel:

```
function resolve_logic(drivers:in logic array) return logic_level;
begin
        for index in drivers'range loop
                if (drivers(index) = L then return L;
                end if
        end loop;
        return H;
end resolve logic;
```

Aceasta functie 'bucleaza' pe un tablou de dispozitive, si daca unul dintre ele are valoarea L, functia intoarce valoarea L. Altfel functia intoarce H.

*b) Tranzitii in 0*

VHDL furnizeaza facilitatea de a modela iesiri care pot trece in inalta impedanta. Un semnal de acest gen poate specifica absenta unei valori specificate pentru semnal, aceasta insemnind ca driveul poate fi deconectat. In scopul realizarii acestui lucru se folosesc elemente cu forma de unda nula. Sintaxa pentru forma de unda a elementului este:

waveform_element ::= **value** expresion [ **after** time_expresion | **null** [ **after** time expresion ]

Un exemplu despre cum se face asignarea semnalului este:

d_out < = **null after** Toz;

Daca toate drivere-le semnalului sint deconectate, se naste intrebarea care este valoarea semnalului ? Apar doua posibilitati, depinzind de tipul declararii semnalului: ca semnal de tip *register* sau *bus*. Pentru semnale de tip *registru*, valoarea cea mai recent determinata ramine ca valoare a semnalului. Pentru semnalele de tip bus, functia de rezolutie trebuie sa determine valoarea semnalului cind nici un driver nu este activ pe magistrala. Este o modalitate prin care magistrale de tip open-colector sau tri-state pot sa fie modelate.

*c) Instructiunea generate*

VHDL are in plus instructiuni concurente care pot fi folosite in arhitecturi pentru a descrie structuri regulate, precum vectori de blocuri, instante ale unor componente sau procese. Sintaxa este:

```
instructiunea_generate ::=eticheta_generate
        schema generare generate
                { instructiune_concurenta } end generate [eticheta-generate] ;
        schena_generare ::=
                for generarespecificamet_paramie_rparametru-generare | if conditie
```

Schemele de generare **for** sint folosite pentru a descrie structurii repetitive. Schemele de generare **if** sint de obicei folosite pentru a descrie cazuri de exceptie in cadrul structurilor cum ar fi cum ar fi conditiile care apar la limita. Aceste sint exemplificate in exemplul urmator. Sa presupunem ca descriu structura unui sumator construit cu celule **full_adder** cu exceptia celulei care are repartizat bitul cel mai putin semnificativ, celula care este de tip **half_adder.**

```
adder : for i in 0 to width-1 generate
        is_bit : if i = 0 generate
                is_cell half_addre port map (a(0), b(0), sum(0), c_in(1));
        end generate is_bit;
                middle bit: if = width -1 generate ms_cell : full adder port map (a(i), bn(i), sum(i), c in(i+1));
        end_generate middle bit;
        ms_bit : if = width -1 generate
                ms_cell : full adder port map (a(i), bn(i), sum(i), carry);
        end_generate ms_bit;
end generate adder;
```

Prima instructiune de generare iteraza dupa i de la 0 pina la width - 1. Pentru cel mai putin semnificativ bit (i = 0) se genereaza o instanta a unui "half adder". Bitii de intrare sint conectati la bitii cei mai putini semnificativi biti ai lui a si b, iar iesirea este conectata la cel mai putin semnificativ bit al sumei, carry find conectat carry-in al urmatorului nivel. Pentru bitii intermediari, se genereaza instante a "full adder-ului", cu intrarile si iesirile conectate in mod asemanator cu cele ale primului nivel. Pentru ultimul nivel se genereaza o noua instanta pen tru"full adder", dar carry este conectat la iesirea Carry a sumatorului.

*d) Asertiuni concurente si apeluri de proceduri*

Sint doua tipuri de instructiuni concurente care nu au fost discutate in capitolele anterioare: asertiuni concurente si apeluri de proceduri concurente. O asertiune concurenta este echivalenta cu un proces care contine o asertiune urmata de o instructiune wait.

Sintaxa este:

asertiune_concurenta :: = [ eticheta: ] asertiune

Asertiunea concurenta de semnal este:

L : **assert** conditie **report** eroare *severity* valoare si este echivalenta cu procesul:

```
L : process
        begin
                assert conditie report eroare severity valoare;
        wait [clauza];
        end process L;
```

Clauza instructiunii wait include toate semnalele care au fost referite in conditie. Daca nici un semnal nu este referit se poate spune despre proces ca a fost activat la initierea simularii, s-au verificat conditiitle si a fost suspendat.

Cealalta instructiune concurenta, apel de procedura concurenta, este echivalenta cu un proces care contine un apel de procedura urmat de o instructiune de asteptare (wait). Sintaxa este:

apel_procedura concurenta ::= [ eticheta: ] apel procedura

Procedura nu trebuie sa aiba parametrii formali din clasa **variable** intrucit nu este posibil ca o variabila sa fie vizibila in orice loc unde estefolosita o instructiune concurenta. Lista de clauze a instructiunii de asteptare (wait) include toate semnalelele care sint parametrii actuali de tip **in** sau **inout** in apelul de procedura. Apelurile concurente sint pentru definirea acelor comportamente care pot fl folosite in locuri diferite sau in module diferite. De exemplu sa presupunem ca in package-ul bit_vect arith se declara procedura:

**procedure** add(**signal** a,b : in bit vector; **signal** result : **out** bit vector);

Un exemplu al apelului procedurii concurente este:

adder : bit_vector_arith.add(sample, old accum, new accum);

Aceasta declaratie este echivalenta cu procesul:

```
begin
        bit_vector_arith.add(sample, old accum, new accum);
        wait on sample, old accum;
end process adder;
```

*e) Instructiuni sub entitati*

Sintaxa pentru declararea unei entitati este:

entitate ::= *entity* identificator **is**
antet_entitate
parte_declarativa_entitate

[ **begin** instructiuni_entitate ]
**end** [ nume_entitate ];

instructiuni entitate ::={ instructiune entitate }
instructiune_entitate ::= asertiune_concurenta | apel concurent_pasiv | proces-pasiv

Instructiunile concurenta care sint permise intr-o declaratie de entitate trebuie sa fie fie pasive, aceasta insemna ca nu vor contine nici o asignare de semnal. Un rezulat al acestei reguli este ca asemenea procese nu pot modifica starea entitatii sau a circuitului in care este folosita entitatea. Ele pot monitoriza starea si deci pot fi folosite pentru a raporta conditii de operare eronate ale entitatii.