Scopul acestui tutotrial este acela de a descrie limbajul de modelare VHDL. VHDL include facilități pentru descrierea structurilor logice și a funcțiilor sistemelor digitale pornind de la un nivel înalt până la nivelul porților elementare. Putem deasemenea, utiliza limbajul VHDL pentru sinteză hardware.

VHDL a apărut în urma programului VHSIC (Very High Speed Integrated Circuits), program susținut de către guvernul Statelor Unite ale Americii. În scurt timp de la lansarea programului a apărut necesitatea standardizării limbajului, standard care a dezvoltat sub auspiciile IEEE și adoptat în forma IEEE Standard 1076, Standard VHDL Language Refrence Manual, în 1987.

Ca și alte standarde, standardul VHDL este revizuit la cel mult cinci ani, astfel că limbajului i s-au adus îmbunătățiri în 1993 (VHDL-93) și în 2002 (VHDL-2002).

Acest tutorial descrie trăsăturile limbajului care sunt comune tuturor standardelor. Cele mai multe compilatoare VHDL suportă la acest moment (2005) cel putin standardul VHDL-93, deci diferentele sintactice nu ar trebui să cauzeze probleme.

Acest tutorial nu este o abordare exhaustivă a limbajului, el face o introducere în noțiunile fundamentale care sunt necesare pentru a realiza modelări relative simple ale sistemelor digitale. Pentru o acoperire completă a limbajului se recomandă studierea cărții The Designer’s Guide to VHDL, 2nd Edition de Peter J. Ashenden publicată de Morgan Kaufman Publishers (ISBN 1-55860-674-2).

CAPITOLUL 1. CONCEPTE FUNDAMENTALE

Modelarea sistemelor digitale

Termenul de sisteme digitale presupune o varietate de sisteme cu componente de nivel jos folosite pentru a completa un chip sau proiectări la nivel de placă. Având în vedere complexitatea sistemelor digitale este necesar să se găsească o metodă de a lucra cu astfel de sisteme.

Cea mai importantă metodă este aceea de a adopta o metodologie sistematică de proiectare a sistemelor digitale. Dacă se pornește de la un document de cerințe pentru sistemul ce se dorește proiectat, putem proiecta o structură abstractă care să îndeplinească cerințele inițiale. Apoi, putem descompune acestă structură intr-o colecție de componente care interacționează pentru a realiza funcțiile impuse. Rezultatul acestui proces este o ierarhie care compune sistemul digital, ierarhie construită din elemente primitive.

Avantajul acestei metodologii este acela că fiecare subsistem poate fi proiectat independent de celelalte subsisteme care alcatuiesc sistemul digital. Când vom utiliza un subsistem ne vom gândi la el ca la o abstractizare și vom ignora detaliile legate de implementarea lui. În acest fel, în orice etapă de proiectare ne vom concentra doar pe cantitatăți mici de informație relevantă pentru proiect reușind în acest fel să menținem atenția asupra proiectării generale.

Termenul model îl vom utiliza pentru a identifica un sistem. Modelul reprezintă acea informație care este relevantă dar și abstractizările pentru detaliile irelevante. Implicarea acestora apare atunci când avem mai multe modele pentru același sistem, deoarece diferite informații sunt relevante în contexte diferite. Un tip de model se poate concentra pe reprezentarea funcțiilor sistemului, cât timp un alt model poate reprezenta modalitatea în care sistemul este compus din subsisteme.

Ideea de model a apărut daorită următoarelor motivații importante:

- Exprimarea cerințelor sistemului într-o modalitate neambiguă și completă;
- Documentarea funcționalității unui sistem;
- Testarea unui proiect pentru a verifica funcționarea corectă a sa;
- Verificarea formală a proprietăților sistemului;

- Sinteza și implementarea sistemului într-o anumită tehnologie – ASIC sau FPGA

Factorul comun al acestor motivații este acela că dorim atingerea maximului de siguranță în procesul de proiectare având un cost minim al timpului de proiectare. Este necesar să ne asigurăm că cerințele sunt clar specificate și înțelese, că subsistemele sunt utilizate corect și că proiectarea rezolvă cerințele inițiale.

O contribuție majoră la mărirea costului unei proiectări este dată de corecția erorilor.

Concepte de modelare VHDL

În cadrul acestei secțiuni, vom studia conceptele de bază VHDL pentru modelarea comportamentală și structurală. Ca și exemplu vom considera un registru pe patru biți prezentat în figura de mai jos:

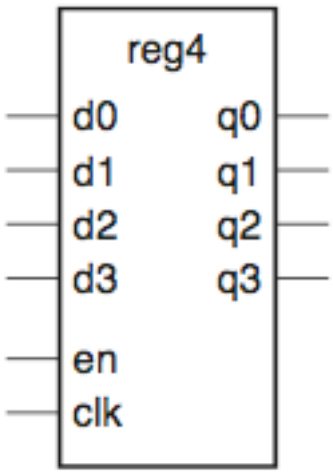

Figura 1. Registru pe 4 biți. Semnalul en este semnal ENABLE , iar semanlul clk este semnal de ceas. Biții de intrare sunt d0 ... d3 iar ieșirile modulului sunt q0 ... q3

Conform metodologiei VHDL vom avea:

- entity — modulul „reg4”
- ports — intrările și ieșirile d0 ... d3 / q0 ... q3
- entity declaration — codul prezentat mai jos. În fapt acest cod prezintă descrierea din exterior a entității.

```
entity reg4 is
        port (d0, d1, d2, d3, en, clk : in bit; -- valorile permise 0/1
                q0, q1, q2, q3 : out bit);
end entity reg4;
```

Descrierea comportamentală a unui sistem

În VHDL, o descriere a unei implementări pentru o entitate este denumită architecture body (corpul arhitecturii) pentru acea entitate. Pot exista mai multe corpuri de arhitecturi pentru aceeași entitate deoarece pot exista implementări alternative care realaizează aceeași funcție.

Putem scrie un copr de arhitectură comportamental pentru o entitate pentru a descrie funcția într-un mod abstract. Corpul unei arhitecturi include doar instrucțiunni de tip process care reprezintă colecții de acțiuni ce trebuiesc executate în ordinea descrisă. Aceste acțiuni se numesc instrucțiuni secvențiale și ele se aseamănă foarte mult cu instrucțiunile scrise în limbajele de programare convenționale.

În cadrul instrucțiunilor secvențiale sunt admise următoarele tipuri de acțiuni:

- evaluarea expresiilor;
- asignarea de valori unor variabile;
- execuții condiționale;
- execuții repetitive;
- apelarea de subprograme.

În plus, există o instrucțiune secvețială care este unică limbajelor de modelare hardware; instrucțiunea de asignare a unui semnal. Această instrucțiune este similară cu instrucțiunea de asignare de valori unei variabile cu excepția faptului că valoarea unui semnal este reînoită la același moment de timp. Această idee este foarte clar ilustrată în exemplul de mai jos:

```
architecture behav of reg4 is
begin
    storage : process is
    variable stored_d0, stored_d1, stored_d2, stored_d3 : bit;
    begin
        wait until clk = '1';
        if en = '1' then
            stored_d0 := d0;
            stored_d1 := d1;
            stored_d2 := d2;
            stored_d3 := d3;
        end if;

        q0 <= stored_d0 after 5 ns;
        q1 <= stored_d1 after 5 ns;
        q2 <= stored_d2 after 5 ns;
        q3 <= stored_d3 after 5 ns;
    end process storage;
end architecture behav;
```

În acest corp de arhitectură, partea de după primul cuvânt cheie begin, include o instrucțiune de tip proces care descrie implementarea de tip comportamental a registrului pe 4 biți. Procesul începe prin declararea numelui procesului storage și se încheie cu cuvintele cheie end process.

Instrucțiunile procesului, definesc o secvență de acțiuni care vor avea loc când sistemul este simulat. Aceste acțiuni de control arată cum valorile porturilor entității se modifică în timp – ele controlează comporatrea entității. Acest process poate modifica valorile porturilor entității utilizând instrucțiuni de asignare a semnalelor.

Funcționarea procesului este următoarea:

- când simularea este pornită, valorile semnalelor sunt setate la „0” și procesul este activat;
- variabilele procesului (cele specificate după cuvântul cheie variable) sunt inițializate la „0”;
- se începe execuția instrucțiunilor în ordinea descrisă de codul sursă;
- prima instrucțiune – instrucțiunea wait – cauzează o suspendare a procesului.
- Cât timp procesul este suspendat, el este senzitiv la semnalul clk – când semnalul clk își modifică valoarea la „1”, procesul este reluat;
- Următoarea instrucțiune este o condiție care testează când semnalul en devine „1”. Dacă semnalul este „1”, atunci instrucțiunile dintre cuvintele cheie then și end if sunt executate; valorile variabilelor procesului vor fi suprascrise folosind valorile semnalelor de intrare. După instrucțiunea condițională if, se află 4 instrucțiuni de tip asignare de semnal care determină ca valoarea semnalelor de ieșire să fie reevaluată după 5ns;
- Când execuția procesului ajunge la finalul listei de instrucțiuni, ele vor fi executate din nou pornind de la cuvântul cheie begin și ciclurile se vor repeta.

  NOTĂ : Cât timp un proces este suspendat, valorile variabilelor de proces nu se pierd. În acest fel, procesul poate reprezenta starea unui sistem

Descrierea structurală a unui sistem

Corpul arhitectură (architecture body) care este compus doar din interconexiuni de subsisteme este denumit corp arhitectural structural. În figura 2, se prezintă modalitatea de realizare a unui registru pe 4 biți prin folosirea bistabilelor D.

Codul VHDL care descrie arhitectura prezentată în figura 2 este prezentat mai jos.

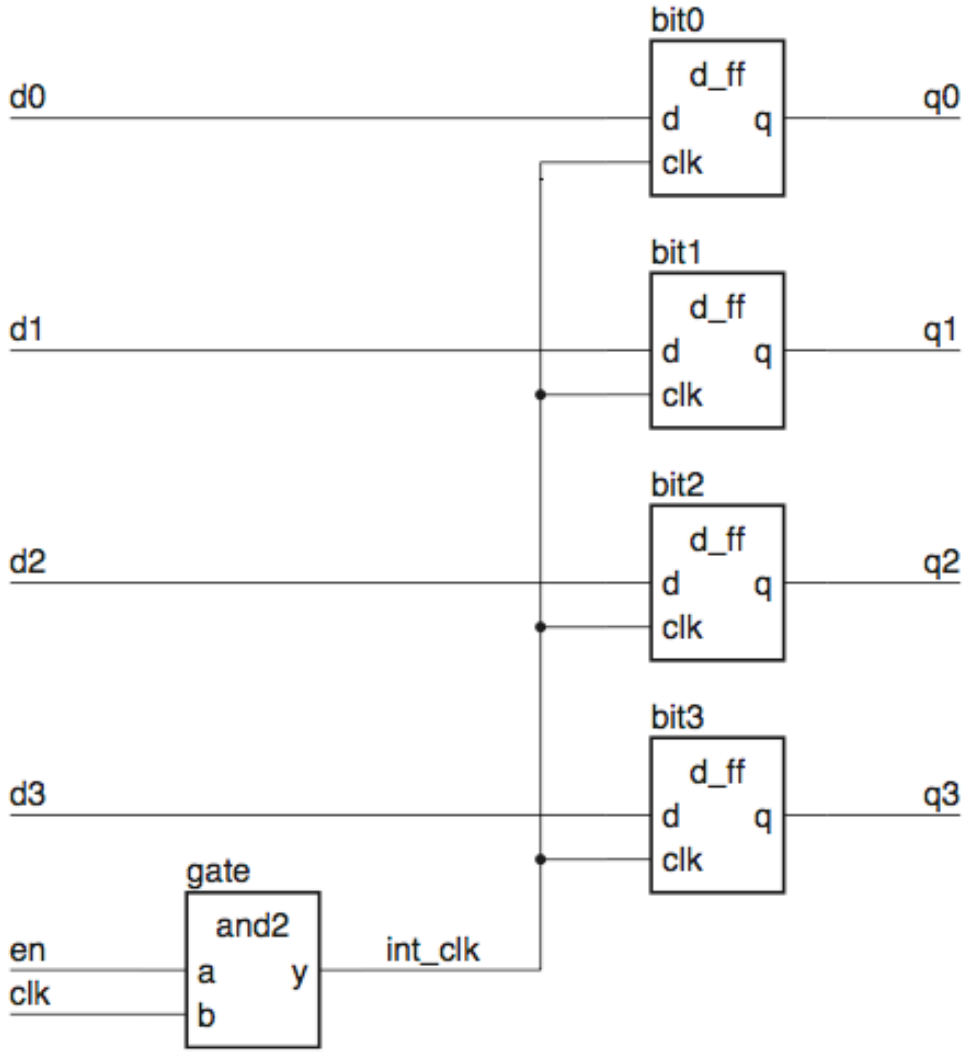

Figura 2. Implementarea unui registru pe 4 biți folosinf bistabile de tip D

```
entity d_ff is
        port ( d, clk : in bit;  q : out bit );
end d_ff;

architecture basic of d_ff is
begin
        ff_behavior : process is
        begin
                wait until clk = '1';
                q <= d after 2 ns;
        end process ff_behavior;
end architecture basic;


entity and2 is
        port ( a, b : in bit;  y : out bit );
end and2;
```

```
architecture basic of and2 is
begin
        and2_behavior : process is
        begin
                y <= a and b after 2 ns;
                wait on a, b;
        end process and2_behavior;
end architecture basic;

architecture struct of reg4 is
        signal int_clk : bit;
begin
        bit0 : entity work.d_ff(basic)
        port map (d0, int_clk, q0);
        bit1 : entity work.d_ff(basic)
        port map (d1, int_clk, q1);
        bit2 : entity work.d_ff(basic)
        port map (d2, int_clk, q2);
        bit3 : entity work.d_ff(basic)
        port map (d3, int_clk, q3);
        gate : entity work.and2(basic)
        port map (en, clk, int_clk);
end architecture struct;
```

Semnalele declarate înainte de cuvântul cheie begin, definesc semnalele interne ale arhitecturii. Spre exemplu, semnalul int_clk, este declarat ca semnal binar; el poate avea doar valorile „0” sau „1”. În VHDL semnalele pot fi declarate ca având valori complexe arbitrare.

Tot ca și semnale sunt tratate porturile entităților.

În partea a doua a corpului arhitecturii, un număr de instanțe de componente sunt create, reprezentând subsistemele din care este compusă entitatea reg4. Fiecare instanță de componentă este o copie a entității care descrie subsistemul, utilizând corpul arhitectură basic corespunzător.

Construcția port map specifică conexiunea porturilor pentru fiecare instanță a unei componente cu semnalele care sunt specifice corpului arhitecturii. Spre exemplu, bit0, este o instanță a entității d_ff, are portul d conectat la semnalul d0, portul clk conectat la semnalul int_clk și portul q conectat la semnalul q0.

Modulul de test

Cel mai adesea, testarea modulelor VHDL se face prin intermediul unui model denumit test bench – model de test. Un astfel de modul este compus dintr-un corp arhitectură care conține o instanță a componentelor ce vor fi testateși procesele care generează secvențe de valori pentru semnalele conectate la instanța componentei.

Corpul arhitectură poate să conțină procese caretestează că instanța componentei produce valorile dorite prin intermediul semnalelor de ieșire.

```
entity test_bench is
end entity test_bench;

architecture test_reg4 of test_bench is
                        signal d0, d1, d2, d3, en, clk, q0, q1, q2, q3 : bit;
    begin
    dut : entity work.reg4(behav)
        port map ( d0, d1, d2, d3, en, clk, q0, q1, q2, q3 );
        stimulus : process is
        begin
                        d0 <= '1';  d1 <= '1';  d2 <= '1';  d3 <= '1';
                        en <= '0';  clk <= '0';
                        wait for 10 ns;
                        en <= '1';  wait for 10 ns;
                        clk = '1', '0' after 10 ns;  wait for 20 ns;
                        d0 <= '0';  d1 <= '0';  d2 <= '0';  d3 <= '0';
                        en <= '0';  wait for 10 ns;
                        clk <= '1', '0' after 10 ns;  wait for 20 ns;
                                    …
                        wait;
        end process stimulus;
end architecture test_reg4;
```

Declarația entității nu are o listă de porturi, deoarece această entitate este în întregime autoconținută. Corpul arhitectură, conține semnale care sunt conectate la intrărirel/ieșirile porturilor instanței componentă dut – cea care reprezintă dispozitivul care trebuie testat.

Procesul denumit stimulus oferă o secvență de valori de test ale semnalelor de intrare specificată prin instrucțiuni de asignare de semnal intercalate cu instrucțiuni wait. În interiorul unui simulator se pot observa valorile semnalelor q0 ... q3 pentru a verifica dacă registrul operează correct. Finalul simulării lasă procesul stimulus într-o stare de așteptare nedefinită.
Analiză, Elaborare și Execuție

Odată scris un model al unui sistem este recomandat să se efectueze o simulare a modului său de lucru. Acestă simulare presupune 3 etape:

- analiză;
- eloborare;
- execuție.

Analiza și elaborarea sunt deasemenea necesare și pentru alte scopuri – spre exemplu sinteza modelului proiectat.

În prima etapă, analiza, descrierea VHDL a unui sistem este verificată în vederea depistării diferitelor tipuri de erori. Ca cele mai multe limbaje comune de programare, la VHDL sintaza și semnatica sunt fix definite. Sintaxa este un set de reguli care guvernează modul cum se scrie un model. Regulile semantice guvernează înțelesul unui program. Spre exemplu, face sens să adunăm două numere, dar nu are sens să adunăm două procese.

Pe durata fazei de analiză, descrierea VHDL este examinată, iar erorile sintactice și semantice sunt localizate. Nu este necesară naliza întregului model odată. Este posibil să se analizeze separat unitățile de proiectare precum entitățile sau declarațiile corpurilor de arhitectură. Dacă analizorul nu găsește erori într-o unitate de proiectare, el creează o reprezentare intermediară a unității și o memorează într-o librărie.

A doua etapă în simularea modelului, elaborarea, este rezultatul parcurgerii ierarhiei modulelor sistemului proiectat și crearea tuturor obiectelor definite în declarații. Un sistem trebuie redus la o colecție de semnale și procese pentru a putea fi simulat.

A treia etapă este etapa de execuție a unui model. Intervalul de timp pe care se efectuează simularea este simulat în pași discreți în strânsă corelație cu evenimentele care pot apărea, din acest motiv vom folosi termenul de simulare de evenimente discrete.

La anumite momente de timp de simulare, un proces poate fi activat prin schimbarea valorii unui semnal aflat în lista semnalelor senzitive. Procesul este reluat și poate programa noi valori care vor fi date semnalelor la un moment ulterior de timp al simulării. Această tehnică se numește tranziție programată pentru un semnal. Dacă noua valoare este diferită de valoarea anterioară a semnalului, va apărea un eveniment și alte procese existente în lista de senzitivități a semnalului pot fi reluate.

Simularea începe cu o fază de inițializare, urmată de o execuție repetitivă a unui ciclu de simulare. Pe durata fazei de inițializare, fiecare semnal premește o valoare inițială, valoare care depinde de tipul declarat al semnalului. Timpul de simulare este setat la zero, apoi fiecare instanță de proces precum și instrucțiunile secvențiale aferente sunt executate. Uzual, un proces va include o instrucțiune de asignare a unui semnal pentru a programa tranziția unui semnal la un moment de timp de simulare ulterior celui actual. Execuția unui proces continuă până când trebuie executată o instrucțiune wait, instrucțiune care va cauza suspendarea procesului.

Procesul de simulare în VHDL se desfășoară conform următorilor pași:

Pasul 1: Pe durata ciclului de simulare, timpul de simulare este mărit până la momentul următor în care o tranziție a unui semnal a fost planificată.

Pasul 2: Toate tranzițiile planificate pentru acest timp de simulare sunt executate. Acest lucru poate cauza apariția unor anumite evenimente prin intermediul unor semnale.

Pasul 3: Toate procesele care sunt senzitive pentru aceste evenimente sunt repornite și este permisă continuarea execuției lor până când o instrucțiune wait este executată. Din nou, procesele execută instrucțiuni de asignare a semnalelor, astfel planificându-se următoarele tranziții ale semnalelor. Când toate procesele au fost suspendate, ciclul de simulare este repetat.

Pasul 4: Oprirea simulării se realizează în momentul în care nu mai sunt planificate tranzacții spre execuție.

CAPITOLUL 2. VHDL este ca un limbaj de programare

Elemente lexicale și de sintaxă

Comentariile

Comentariile sunt deosebite de importante în orice cod sursă, deoarece ele au rolul de a lămuri anumite decizii luate de cel care face implementarea la un moment dat.

Comentariile în VHDL pot fi făcute pe o singiră linie sau pe mai multe linii. Exemple sugestive pentru ambele situații sunt prezentate în cele ce urmează:

O line de cod VHDL --comentariu pe o singură linie

-- Prima line de comentarii
-- A doua linie de comentarii
O linie de cod VHDL

Identificatori

Identificatorii sunt utilizați pentru a denumi obiectele într-un model VHDL. În VHDL un identificator poate fi declarat doar dacă respectă următoarele condiții:

- litere „A-Z” și „a-z”, cifrele „0-9” și caracterul underline „_”;
- trebuie să înceapă cu o literă;
- nu se poate termina cu carcaterul „_”;
- nu poate include două caractere „_” succesive;
- identificatorii nu sunt case sensitive.

Exemple de identificatori valizi în VHDL: A, X0, counter, Next_Value, generate_read_cycle

Cuvinte rezervate

Cuvintele rezervate sunt acei identificatori care pot fi folosiți doar în situații speciale.

abs	**access**	**after**	**alias**	**all**	**and**
architecture	**array**	**assert**	**attribute**	**begin**	**block**
body	**buffer**	**bus**	**case**	**component**	**configuration**
constant	**disconnect**	**downto**	**else**	**elsif**	**end**

entity			**exit**		**file**		**for**		**function**		**generate**								
generic		**group**		**guarded**		**if**		**impure**	**in**										
inertial				**inout**		**is**		**label**		**library**		**linkage**							
literal					**loop**		**map**		**mod**		**nand**		**new**						
next					**nor**		**not**		**null**		**of**		**on**						
open					**or**		**others**		**out**		**package**		**port**						
postponed				**procedure**		**process**		**protected**		**pure**		**range**							
record				**register**		**reject**		**rem**		**report**		**return**							
rol					**ror**		**select**		**severity**		**shared**	**signal**							
sla					**sll**		**sra**		**srl**		**subtype**		**then**						
to					**transport**		**type**		**unaffected**		**units**		**until**						
use					**variable**		**wait**		**when**		**while**		**with**						
xnor					**xor**														

Numere

Într-un cod VHDL numrele pot apărea ca numere întregi sau ca numere reale. Semnificația numerelor întregi şi a numrelor reale este aceeaşi ca şi în alte limbaje de programre (spre exemplu limbajul C). Câteva exemple de numere întregi şi numere reale sunt prezentate mai jos:

Numere întregi:	23	0	146
Numere reale:	23.1	0.0	3.14159

Cele două tipuri de reprezentări pot utiliza notația exponențială în care un număr este urmat de litera ”E„ sau ”e„ și o valoare exponențială (1.234E09, 98.6E + 21, 34.0e-08)

Caractere

Un caracter poate apărea într-un cod VHDL în cazul în care el este prezente între apostroafe. Orice succesiune de caractere printabile pot apărea între apostroafe.

’A’	caracterul A
’z’	caracterul z
’,’	caracterul virgulă
’’’	caracterul apostrof
’ ’	caracterul spațiu

Șirurile de caractere

Un șir reprezintă o secvență de caractere, secvență care trebuie să apară între ghilimele. Șirul poate conține orice secvență de caractere, inclusiv 0, dar este limitat la o singură linie. Dacă se dorește concatenarea a două șiruri atunci se va folosi operatorul de concatenare &. Următoarele exemple definesc șiruri de caractere.

”Un șir”
”Șirurile sunt scrise pe o singură linie”

”000011110000”
”Șirul nr. 1”
& ”Șirul nr. 2”

Șirurile de biți

Limbajul VHDL include valori care reprezintă biți, valoarea unui bit poate fi ’0’ sau ’1’. Un șir de biți reprezintă secvența de valori pentru biții cuprinși în șir.Șirul de biți este cuprins între ghilimele și este precedat de un caracter special care reprezintă baza de reprezentare: B pentru baza 2, O pentru baza 8 și X pentru baza 16.

B”000011110000” b”0000_1111_0000”

Din exemplul de mai sus se poate observa că într-un șir de biți este permisă folosirea caracterului ’_’. Folosirea acestui caracter permite o citire mai ușoară a șirului de biți și nu va afecta în nici un fel valoarea șirului de biți.

O”372” echivalent cu B”011_111_010”
o”00” echivalent cu B”000_000”
X”FA” echivalent cu B”1111_1010”
x”0d” echivalent cu B”0000_1101”

Sintaxa

Regulile de sintaxă prezentate în acest tutorial se bazează pe EBNF (Extended Backus-Naur Form). Ideea este să împărțim limbajul în categorii sintactice. Pentru fiecare categorie sintactică se va scrie o regulă care descrie modalitatea de construire a unei clauze VHDL pentru aceea categorie, prin combinarea elementelor lexicale și a clauzelor altor categorii. Modalitatea de scriere a regulilor este următoarea:

variabila_de_asignat ⇐ ținta := expresie;

Precizare: partea din stânga semnului ⇐ trebuie citită ”este definită să fie”

Regula prezentată în exemplul de mai sus indică că o clauză VHDL din categoria ”variabila_de_asignat” este definită să fie o clauză în categoria ”țintă” urmată de simbolul ”:=” urmat și el la rândul lui de o clauză din categoria ”expresie”. Regula se termină prin caracterul ”;”.

Următorul tip de regulă este acela care ia în considerare posibilitatea apariției unui element opțional într-o clauză. Partea opțională dintr-o clauză va fi delimitată prin intermediul caracterelor ”[” ”]”.

function_call ⇐ name [(lista_asociații)]

Regula prezentată mai sus indică faptul că apelul unei funcții se face prin numele funcției urmat de lista asocierilor cuprinsă între paranteze.

```
process_instrucțiunea ⇐
    process is
            {partea declarativă a procesului}
    begin
            {partea secvențială a procesului}
    end process;
```

Semnele acoladă (”{” ”}”) reprezintă faptul că un proces poate include mai multe părți declarative sau niciuna. Similar, este posibil să avem zero mai multe instrucțiuni secvențiale în cadrul unui proces.

```
case_instrucțiunea ⇐
    case expresie is
            instrucțiune_case_alternativă
            { ... }
    end case
```

Regula de mai sus indică faptul că o instrucțiune case trebuie să conțină cel puțin o instrucțiune case alternativă, dar poate conține un număr arbitrar de instrucțiuni case alternative suplimentare. În exemplul de mai sus, acoladele se referă numai la instrucțiunile case alternative.

```
listă_identificatori ⇐ identificator {, ...}
```

Această regulă specifică faptul că o listă de identificatori poate fi compusă din unul sau mai mulți identificatori separați virgulă.

```
mode ⇐ in | out | inout
```

Acestă regulă specifică pentru categoria ”mode”, faptul că ea poate fi formată de la o clauză formată din unul sau mai multe cuvinte rezervate.

Ultima notație permisă este cea care folosește parantezele ”(” ”)”.

```
term ⇐ factor {(* | / | mod | rem ) factor}
```

În acest exemplu, un factor poate fi urmat de un simbol operator și apoi de un alt factor.

Constante și variabile

Constantele și variabilele sunt obiecte în care datele pot fi memorate pentru a putea fi utilizate într-un model. Diferența dintre o constantă și o variabilă este aceea că valoarea unei constante nu poate fi modificată. Valoarea unei variabile poate fi modifcată doar prin intermediul unei instrucțiuni de asignare de valori.

Atât constantele cât și variabilele trebuiesc declarate înainte de a fi utilizate ân interiorul unui model. O declarație conține numele variabilei/constantei, tipul său precum și valoarea inițială.

declarație_constantă ⇐

constant identificator {, ...} : indicare_subtip := expresie;

Exemple de declarare a unei constante:

```
constant number_of_bytes : integer := 4;
constant number_of_bits : integer := 8 * number_of_bytes;
constant e : real := 2.718281828;
constant prop_delay : time := 3 ns;
constant size_limit, count_limit : integer := 255;
```

Declararea unei variabile este similară cu declararea unei constante:

declarație_variabilă ⇐

variable identificator {, ...} : indicare_subtip [:= expresie];

În cazul în care nu se specifică valoarea inițială, VHDL va asigna variabilei, cea mai mică valoare posibilă care aparține tipului de variabilă declarat. Spre exemplu, pentru întregi, valoarea cea mai mică este 0.

```
variable index : integer := 0;
variable sum, average, largest : real;
variable start, finish : time := 0 ns;
```

În cazul declarării în interiorul unui model, variabilele pot fi utilizate doar în cadrul unui proces. O restricție de care trebuie ținut seama, este aceea că o variabilă nu poate fi accesibilă pentru mai mult de un proces.

Modificarea valorii unei variabile se poate face doar prin intermediul unei instrucțiuni de asignare de variabilă:

instrucțiune_asignare_variabilă ⇐ nume : expresie;

Tipul valorii produse pentru modificarea valorii variabilei trebuie să fie identic cu tipul declarat al variabilei.

Tipul de date intreg

Tipul de date întreg are gama de reprezentare $-2^{31}+1$: $2^{31}-1$. În VHDL există două subtipuri predefinite de întregi:

natural acest tip de date include întregii de la 0 la cel mai mare număr întreg reprezentabil;
positive acest tip de date include întregii de la 0 la cel mai mare număr întreg reprezentatbil

Aceste tipuri de date este recomandat de folosit ori de câte ori nu se lucrează cu date negative. În acest fel o parte din erorile de implementare pot fi înlăturate deoarece se exclude posibilitatea de producere a unor numere negative.

Operațiile posibile care pot fi efectuate cu aceste tipuri de date sunt următoarele: +, -, *, / (cu trunchiere), mod (acelasi semn ca și operandul din dreapta), rem (restul unei împărțiri – același semn ca operandul din stânga), abs (valoarea absolută), ** (exponențial – operandul din dreapta trebuie să fie pozitiv).

Exemplu:

- declararea unui subtip întreg

```
subtype small_int is integer range -128 to 127;
```

- declararea variabilelor de tipul small_int:

```
variable deviation : small_int;
variable adjustment : integer;
```

- folosirea variabilelor declarate:

```
deviation := deviation + adjustment;
```

Tipuri de date în virgulă mobilă

Aceste tipuri de date reprezintă numerele reale în conformitate cu standardul IEEE (se utilizează mantisa și exponentul). Tipul de date predefinit se numește real și domeniul de reprezentare este în conformitate cu standardul IEEE 64 biți dublă precizie. Operațiile posibile cu acest tip de date sunt cele prezentate la tipul de date întreg cu observația că operanziitrebuie să fie de același tip. Doar în cazul operației exponențiale operandul din dreapta trebui să aibă o valoare întreagă.

Tipul de date time

Acest tip de date predefinit este folosit pentru reprezentarea timpilor de simulare și a întârzierilor. Declararea unei valori de tip time, se face prin scrierea valorii timpului urmată de un spațiu și apoi se va specifica unitatea de timp.

5 ns 22 us 471.3 msec

Unitățile de timp permise sunt următoarele:

fs, ps, ns, us, ms, sec, min, hr

Valorile posibile pentru acest tip de date sunt valorile pozitive și valorile negative. În VHDL există un subtip de date de tip time, redefinit, denumit delay_lenght care include doar valorile pozitive.

Operatorii aritmetici acceptati pentru acest tip de date sunt:

- adunare, scădere, identitate și negare – rezultatul aplicării acestor operatori este o valoare de tip time.
- multiplicare și împărțire – unul din operanzi poate fi de tipul integer sau real
- absolut

Precizare: În cazul împărțirii a două valori de tipul time va rezulta o valoare de tipul integer.

18 ns / 2.0 = 9.0 ns; 33 ns / 22 ps = 1500
abs 2 ps = 2 ps; abs (-2 ps) = 2 ps

Tipul de date enumerare

Cel mai adesea, când se implementează modulele hardware la nivel abstract, este deosebit de util să se utilizeze o mulțime de nume pentru codificarea valorilor anumitor semnale. Sintaxa este următoarea:

declarație_type ⇐ type identificator is definiția_tipului

O declarație de tipul type permite introducerea de tipuri noi de date, distincte de alte tipuri de date. Definirea tipului enumerare este:

definire_tip_enumerare ⇐ ((identificator | caracter) {, ...})

Acestă definire implică în fapt creearea unei liste ale tuturor valorilor permise. Fiecare valoare poate fi un identificator sau un caracter, ca în exemplul de mai jos:

type alu_function is (disable, pass, add, substract, multiply, divide);
type octal_digit is ('0', '1', '2', '3', '4', '5', '6', '7');

Pe baza declarațiilor făcute mai sus putem declara, spre exemplu, următoarele variabile:

```
variabile alu_op : alu_function;
variable last_digit : octal_digit := ’0’;
```

În conformitate cu declarațiile de mai sus, următoarele instrucțiuni de asignare de variabile sunt valide:

```
alu_op := subtract;
last_digit := ’7’;
```

Caractere

Tipul predefinit caracter include toate caracterele din setul de caractere ISO 8859. Definiția tipului de date caracter este prezentată mai jos. Ea conține o mixtură de identificatori (pentru controlul caracterelor) şi literalii caracter (pentru caracterele grafice). Caracterul din poziția 160 este un caracter spațiu neseparabil, distinct de caracterul aflat la poziția 173.

```
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,
    ' ',    '!',    '"',    '#',    '$',    '%',    '&',    ''',
    '(',    ')',    '*',    '+',    ',',    '–',    '.',    '/',
    '0',    '1',    '2',    '3',    '4',    '5',    '6',    '7',
    '8',    '9',    ':',    ';',    '<',    '=',    '>',    '?',
    '@',    'A',    'B',    'C',    'D',    'E',    'F',    'G',
    'H',    'I',    'J',    'K',    'L',    'M',    'N',    'O',
    'P',    'Q',    'R',    'S',    'T',    'U',    'V',    'W',
    'X',    'Y',    'Z',    '[',    '\',    ']',    '^',    '_',
    '`',    'a',    'b',    'c',    'd',    'e',    'f',    'g',
    'h',    'i',    'j',    'k',    'l',    'm',    'n',    'o',
    'p',    'q',    'r',    's',    't',    'u',    'v',    'w',
    'x',    'y',    'z',    '{',    '|',    '}',    '~',    del,
    c128,   c129,   c130,   c131,   c132,   c133,   c134,   c135,
    c136,   c137,   c138,   c139,   c140,   c141,   c142,   c143,
    c144,   c145,   c146,   c147,   c148,   c149,   c150,   c151,
    c152,   c153,   c154,   c155,   c156,   c157,   c158,   c159,
    ' ',    '¡',    '¢',    '£',    '¤',    '¥',    '¦',    '§',
    '¨',    '©',    'ª',    '«',    '¬',    '-',    '®',    '¯',
    '°',    '±',    '²',    '³',    '´',    'µ',    '¶',    '·',
    '¸',    '¹',    'º',    '»',    '¼',    '½',    '¾',    '¿',
    'À',    'Á',    'Â',    'Ã',    'Ä',    'Å',    'Æ',    'Ç',
    'È',    'É',    'Ê',    'Ë',    'Ì',    'Í',    'Î',    'Ï',
    'Ð',    'Ñ',    'Ò',    'Ó',    'Ô',    'Õ',    'Ö',    '×',
```

```
'Ø',    'Ù',    'Ú',    'Û',    'Ü',    'Ý',    'Þ',    'ß',
'à',    'á',    'â',    'ã',    'ä',    'å',    'æ',    'ç',
'è',    'é',    'ê',    'ë',    'ì',    'í',    'î',    'ï',
'ð',    'ñ',    'ò',    'ó',    'ô',    'õ',    'ö',    '÷',
'ø',    'ù',    'ú',    'û',    'ü',    'ý',    'þ',    'ÿ');
```

Pentru o ilustra utilizarea tipului caracter, este nevoie de următoarea declarație de variabilă:

```
varaible cmd_char, terminator : character;
```

şi apoi se vor face asignările:

```
cmd_char := 'P';
terminator := cr;
```

Tipul de date Boolean

Tipul de date boolean este definnit ca:

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

Acest tip este utilizat pentru a reprezenta valorile condițiilor, care pot controla execuția modelului comportamental. Există un număr de operatori care pot fi aplicați valorilor diferitelor tipuri de date, numiți operatori relaționali şi logici. Operatorii relaționali "=" şi "/=" pot fi aplicați operanzilor de orice tip, atâta timp cât cei doi operatori au acelaşi tip. Spre exemplu, expresiile

```
123 = 123,      'A' = 'A',      7 ns = 7ns
```

sunt toate adevărate, cât timp expresiile

```
123 = 456,      'A' = 'z', 7 ns = 2 us
```

sunt toate false.

Operatorii relaționali care testează ordinea sunt: "<", "<=", ">", ">=". Aceştia pot fi aplicați doar valorilor care au tipul scalar.

Tipul bit

Deoarece VHDL este utilizat pentru modelarea sistemelor digitale, este util să avem un tip de date care reprezintă valoarea unui bit. Definiția acestui tip de date este următoarea:

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

Operatorii logici menționați pentru valorile boolean-e pot fi deasemenea aplicate valorilor de tip bit, rezultatul fiind tot de tipul bit. Valoarea ‘0’ corespunde lui FALSE, cât timp valoarea ‘1’ corespunde lui TRUE.

Exemplu

‘0’ and ‘1’ = ‘0’, ‘1’ xor ‘1’ = 0

Diferența dintre tipul boolean şi tipul bit este aceea că valorile boolean-e sunt utilizate pentru a modela condițiile abstracte, cât timp valorile de tipul bit sunt utilizate pentru a modela nivelele logice hardware. Deci, ‘0’ repreyintă nivelul de jos al unui semnal cât timp ‘1’ repreyintă nivelul de sus al unui semnal.

Tipul standard logic

IEEE a standardizat un pachet denumit std_logic_1164 care permite modelarea semnalelor digitale luând în considerare anumite efecte electrice. Unul din tipurile definite în acest pachet este un tip enumerare, denumit std_logic definit astfel:

```
type std_ulogic is (    'U', — neinițializat
                        'X', — necunoscut forțat
                        '0', — zero forțat
                        '1', — unu forțat
                        'Z', — mare impedanță
                        'W', — necunoscut slab
                        'L', — zero slab
                        'H', — unu slab
                        '–' ); — nu contează
```

Acest tip poate fi utilizat pentru a reprezenta semnalele conduse de conductorii activi, conductorii rezistivi (precum trage-sus şi trage-jos) sau conductorii tri-state inclusiv starea de mare impedanță. Fiecare tip de conductor, poate conduce una din valorile „zero”, „unu” sau „necunoscut”. O valoare „necunoscut” este condusă de un model când el este în imposibilitatea de determina dacă semnalul este „zero” sau „unu”. La declararea unui semnal de tipul **std_ulogic**, valoarea inițializată pentru acest semnal va fi “U”. Valoarea “nu contează” este utilizată de programele de sinteză logică şi poate fi de asemenea utilizată la definirea vectorilor de test, pentru a specifica faptul că valoarea unui semnal ce trebuie comparată cu vector de test un este importantă.

Dacă se va include linia:

library ieee; **use** ieee.std_logic_1164.**all**;

înaintea descrierii oricărei entități sau corp arhitectural care utilizează pachetul, vom putea scrie module al căror tip aparțin definiției limbajului VHDL.

În acest mod, se pot crea constante, variabile şi semnale de tipul **std_ulogic**. Cât timp avem asignate valori default, vom putea utiliza operatorii logici **and**, **or**, **not** etc. Fiecare operator care operează cu valori de tipul **std_ulogic**, va returna rezultate de tipul **std_ulogic** "U", "X", "0" sau "1".

Instrucțiuni secvențiale

În această secțiune vom vedea cum datele pot fi manipulate în interiorul unui proces utilizând *instrucțiuni secvențiale*, numite aşa deoarece, instrucțiunile se executa în ordine secvențială. Una din instrucțiunile secvențiale de bază este instrucțiunea de asignare, instrucțiune prezentată în cadrul asignării unei variabile. Modelele care conțin acest tip de instrucțiuni sunt cel mai adesea denumite structuri controlate. Ele permit selecția între acțiuni alternative precum şi acțiuni repetitive.

Instrucțiunea IF

În multe modele, comportarea modelului depinde de un set de condiții care pot fi adevărate sau false pe durata simulării. Acest tip de comportament este modelat cel mai bine prin intermediul instrucțiunii IF. Sintaxa instrucțiunii este următoarea:

```
if_statement
        [ if_label : ]
        if boolean_expression then
                { sequential_statement }
        { elsif boolean_expression then
                { sequential_statement } }
        [ else
                { sequential_statement } ]
        end if [ if_label ] ;
```

Exemplu

```
if en = '1' then
        valoare1 := data_in;
end if;
```

Expresia booleană după cuvântul cheie **if** este condiția care este utilizată pentru a stabili dacă instrucțiunea ce urmează cuvântului cheie **then** este executată sau nu. Dacă condiția este evaluată ca adevărată, atunci instrucțiunea este executată. De asemenea, se poate specifica acțiunea ce va fi executată dacă condiția este falsă.

Exemplu

```
if sel = 0 then
        result <= input_0;              -- executată dacă sel = 0
else
        result <= input_1;              -- executată dacă sel /= 0
end if;
```

În acest exemplu, prima isntrucțiune de asignare a semnalului este executată dacă condiția este adevărată, pe când cea de a doua instrucțiune de asignare este executată dacă condiția este falsă.

Putem construi o formă mult mai elaborată a instrucțiunii **if** pentru verificarea unui număr diferit de condiții, ca în exemplul următor.

Exemplu

```
if mode = immediate then
        operand := immed_operand;
elsif opcode = load or opcode = add or opcode = subtract then
        operand := memory_operand;
else
        operand := address_operand;
end if;
```

În general, se poate construi o instrucțiune **if** cu un număr arbitrar de caluze **elsif** şi putem include sau omite clauzele **else**. Execuția instrucțiunii **if** porneşte prin evaluarea primei condiții. Dacă ea este falsă, condițiile succesive sunt evaluate în ordine, până când una din ele este găsită adevărată, caz în care instrucțiunile corespunzătoare sunt executate.

Dacă nici una dintre condiții nu este adevărată, şi există inclusă o clauză **else**, instrucțiunile care urmează cuvântului cheie **else** vor fi executate.

Exemplu

Un termostat de căldură poate fi modelat ca o entitate cu două intrări întregi, una care specifică temperatura dorită şi alta care este conectată la un termometru. Ieşirea este de tipul boolean şi acționează asupra căldurii – o porneşte sau o opreşte. Termostatul porneşte căldura dacă temperatura măsurată este cu două grade mai mică decât temperatura dorită şi opreşte căldura, dacă temperatura măsurată este mai mare cu două grade decât temperatura dorită.

```
entity thermostat is
        port ( desired_temp, actual_temp : in integer;
                heater_on : out boolean );
end entity thermostat;
```

```
architecture example of thermostat is
begin
        controller : process (desired_temp, actual_temp) is
        begin
                if actual_temp < desired_temp – 2 then
                        heater_on <= true;
                elsif actual_temp > desired_temp + 2 then
                        heater_on <= false;
                end if;
        end process controller;
```

```
end architecture example;
```

Instrucțiunea CASE

Dacă avem un model a cărui comportare depinde de valoarea unei singure expresii, putem utiliza o instrucțiune case. Sintaxa instrucțiunii este următoarea:

```
case_statement
        [ case_label : ]
        case expression is
                ( when choices => { sequential_statement } )
                { … }
        end case [ case_label ] ;
choices  ( simple_expression I discrete_range I others ) { | … }
```

Spre exemplu, ne dorim modelarea unei unități aritmetice/logice care are o intrare de control **func**, declarată ca un tip enumerare.

```
        type alu_func is (pass1, pass2, add, subtract);
```

Descrierea comportamentală se poate realiza prin intermediul unei instrucțiuni case.

```
case func is
        when pass1 =>
                result := operand1;
        when pass2 =>
                result := operand2;
        when add =>
                result := operand1 + operand2;
        when subtract =>
                result := operand1 – operand2;
end case;
```

La începutul instrucțiunii case este “expresia selecție”, între cuvintele cheie **case** şi **is**. Valoarea expresiei este utilizată pentru a selecta care instrucțiuni vor fi executate. Corpul instrucțiunii **case** este constituit dintr-o serie de *alternative*. Fiecare alternativă începe cu cuvântul **when** şi este urmată de una sau mai multe *alegeri* şi o secvență de instrucțiuni.

Alegerile sunt valori, care sunt comparate cu valoarea expresiei selector. Trebuie să existe doar o singură alegere pentru fiecare valoare posibilă. Instrucțiunea case găseşte alternativele ale căror valoare este egală cu valoarea expresiei selector şi execută instrucțiunile din aceea alternativă.

Putem include mail mult de o alegere în fiecare alternativă prin scrierea alegerilor separate de simbolul “|”.

Exemplu

Tipul opcode este declarat astfel:

```
type opcodes is (nop, add, subtract, load, store, jump, jumpsub, branch, halt);
```

putem scrie o alternativă incluzând trei dintre valorile definite astfel:

```
when load | add | subtract => operand := memory_operand;
```

Dacă avem un număr de alternative într-o instrucțiune **case** şi dorim să includem o alternativă care să trateze valorile posibile ale variabilei selctor nemenționate în nici o alternativă, se va utiliza valoarea specială **others**. Spre exemplu, dacă variabila **opcode** este o variabilă de tipul **opcodes**, declarată mai sus, se poate scrie:

```
case opcode is
        when load | add | subtract =>
                operand := memory_operand;
        when store | jump | jumpsub | branch =>
                operand := address_operand;
        when others =>
                operand := 0;
end case;
```

În acest exemplu, dacă valoarea **opcode** este orice altceva decât alegerile listate în prima şi a doua alternativă, ultima alternativă este selectată. Există doar o singură alternativă care utilizează alegerea **others**. În cazul în care această alternativă există, ea trebuie să fie ultima alternativă din instrucțiunea **case**. O alternativă inclusă în **others**, nu mai poate fi inclusă în nici o altă alegere.

*Alegerile intr-o instrucțiune **case**, vor fi scrise utilizând valori statice locale. Aceasta înseamnă ca valorile alegerilor trebuie să fie determinate pe durata fazei de analiză a procesului de proiectare.*

Exemplu

Putem scrie un model comportamental pentru un multiplexor cu o intrare de selecție, **sel**, două intrări **cc_z** şi **cc_c**; şi o ieşire **taken**. Intrările şi ieşirile sunt de tipul IEEE standard-logic, cât timp intrarea de selecție este de tipul **branch_fn**, tip declarat astfel:

```
type branch_fn is (br_z, br_nz, br_c, br_nc);
```

Declarația entității care defineşte porturile precum şi arcitectura corpului comportamental sunt prezentate mai jos. Corpul arhitectural conține un proces care este senzitiv pe intrări. Procesul cuprinde o instrucțiune case pentru a selecta valoarea ce va fi asignată ieşirii.

```
library ieee; use ieee.std_logic_1164.all;
entity cond_mux is
        port ( sel : in barnch_fn;
```

```
                cc_z, cc_c : in std_ulogic;
                taken : out std_ulogic );
end entity cond_mux;
-------------------------------------------------------------
architecture demo of cond_mux is
begin
        out_select : process (sel, cc_z, cc_c) is
        begin
                case sel is
                        when br_z =>
                                taken <= cc_z;
                        when br_nz =>
                                taken <= not cc_z;
                        when br_c =>
                                taken <= cc_c;
                        when br_nc =>
                                taken <= not cc_c;
                end case;
        end process out_select;
end architecture demo;
```

Instrucțiunile LOOP şi EXIT

Adesea este nevoie să se scrie o secvență de instrucțiuni care sunt executate în mod repetat. Pentru aceaste situații se va utiliza *intsrucțiunea loop*. Sintaxa pentru un ciclu care iterează nedefinit este următoarea:

```
loop_statement
        [ loop_ label : ]
        loop
                {  sequential_statement }
        end loop [ loop_ label ] ;
```

Pentru ieşirea din ciclu atunci când anumite condiții sunt îndeplinite se realizează prin intermediul instrucțiunii *exit*. Sintaxa folosită este următoarea:

```
exit_statement
        [  label : ] exit [ loop_ label ] [ when boolean_ expression ] ;
```

Cea simplă utilizare a instrucțiunii *exit* este:

```
exit;
```

Când această instrucțiune este executată, orice instruiuni rămase în ciclu sunt ignorate iar controlul este transferat instrucțiunii care apare după cuvintele cheie **end loop**. Ca atare, într-un ciclu se poate scrie:

```
if condition then
        exit;
end if;
```

unde *condition* este o expresie booleană. VHDL oferă însă şi o altă cale de folosire a instrucțiuni **exit**, prin folosirea clauzei **when**.

```
loop
        …
        exit when condition;
        …
end loop;
…       -- controlul este transferat către această instrucțiune
        -- când condition devine adevărată în ciclu.
```

Exemplu

Se doreşte realizarea unui numărător care porneşte de la zero şi valoarea sa se incrementează cu 1 la fiecare tranziție a ceasului de la '0' la '1'. Când numărătorul ajunge la valoarea 15, el va fi resetat la următorul front pozitiv al ceasului. Numărătorul are o intrare asincronă **reset** care atunci când devine '1' cauzează resetarea numărătorului. Ieşirea va rămâne '0', atâta timp intrarea **reset** este '1' şi se reâncepe numărătoarea pe următorul front pozitiv al ceasului de când intrarea **reset** a devenit '0'.

```
entity counter is
        port ( clk, reset : in bit; count : out natural );
end entity counter;
-------------------------------------------------------------
architecture behavior of counter is
begin
        incrementer : process is
                variable count_value : natural := 0;
        begin
                count <= count_value;
                loop
                        loop
                                wait until clk = '1' or reset = '1';
                                exit when reset = '1';
                                count_value := (count_value + 1) mod 16;
                                count <= count_value;
                        end loop;
                        -- în acest punct, reset = '1'
                        count_value := 0;
                        count <= count_value;
                        wait until reset = '0';
                end loop;
        end process incrementer;
end architecture behavior;
```

Corpul arhitecturii conține două cicluri **loop**. Al doilea ciclu se ocupă de operația de numărare. Când **reset** devine '1', instrucțiunea **exit** cauzează terminarea acestui ciclu şi transferarea controlului primei instrucțiuni care se găseşte după descrierea acestui

ciclu. Valoarea *count* dar şi ieşirea *count* sunt resetate şi procesul aşteaptă până când **reset** va comuta în '0', după care procesele sunt reluate şi ciclurile se vor repeta.

În anumite cazuri, vom dori sa transferîm controlul la ieşirea celui de al doilea ciclu **loop** şi să continuăm ciclul. Acest lucru poate fi realizat prin eti chetarea primului ciclu şi utilizarea etichetei într-o instrucțiune **exit**.

```
loop_name : loop
        …
        exit loop_name;
        …
end loop loop_name ;
```

Ciclul WHILE

Acest ciclu testează o condiție înainte de fiecare iterație. Dacă condiția este adevărată, iterația se va executa, în caz contrar ciclul este terminat. Sintaxa folosită este următoarea:

```
loop_statement
        [ loop_ label : ]
        while boolean_ expression loop
                { sequential_statement }
        end loop [ loop_ label ] ;
```

Singura diferență dintre această formă de scriere şi instrucțiunea de bază **loop**, este accea că a fost adăugat cuvântul cheie **while** şi condiția înainte de cuvântul cheie **loop**. Toate noțiunile prezentate la ciclul **loop**, sunt aplicabile ciclului **while**. Condiția este testată înainte de fiecare iterație a ciclului **while**, inclusiv la prima iterație. În cazul în care condiția este falsă înainte de a se intra în ciclu, execuția ciclului este terminată imediat fară ca vreo iterație să se execute.

Exemplu

Se doreşte dezvoltarea unui model pentru o entitate **cos** care calculează funcția cosinus a unei intrări **theta** utilizând relația:

$$\cos(\theta) = 1 - \frac{\theta^2}{2} - \frac{\theta^4}{4} - \frac{\theta^6}{6} \ldots$$

Vom adăuga termeni succesivi la serie până când termenii vor deveni mai mici decât a milioanea parte din rezultat.

```
entity cos is
        port ( theta : in real; result : out real );
end entity cos;
```

```
architecture series of cos is
begin
```

```
        summation : process (theta) is
                variable sum, term : real;
                variable n : natural;
        begin
                sum := 1.0;
                term := 1.0;
                n := 0;
                while abs term > abs (sum / 1.0E6) loop
                        n := n + 2;
                        term := (–term) * theta**2 / real(((n–1) * n));
                        sum := sum + term;
                end loop;
                result <= sum;
        end process summation;
end architecture series;
```

Cilul FOR

Ciclul **for** include în specificația sa, de câte ori corpul ciclului este executat. Sintaxa este următoarea:

```
loop_statement
        [ loop_ label : ]
        for  identifier in  discrete_range loop
                {  sequential_statement }
        end loop [ loop_ label ] ;
```

unde *discrete_range* poate fi de forma următoare:

```
simple_expression ( to I downto )  simple_expression
```

reprezentând toate valorile dintre cele două capete de interval, inclusiv aceste valori.

Exemplu

```
for count_value in 0 to 127 loop
        count_out <= count_value;
        wait for 5 ns;
end loop;
```

identificatorul *count_value* ia valorile 0, 1, 2 etc. Pentru fiecare valoare, cele două instrucțiuni din interiorul ciclului **for** sunt executate. Semnalul *cout_out* va avea asigante valorile 0, 1, 2, …, 127 la un interval de 5ns. Se observă că variabila *count_value* nu trebuie declarată şi ea este considerată ca şi o constantă (programatorul nu îi poate modifca valoarea). Această variabilă are domeniul de vizibilitate doar în interiorul ciclului **for** nu şi în afara acestuia. Ca şi în cazul ciclului de bază **loop**, in interiorul ciclului **for** poate fi introdusă instrucțiunea **exit** sau putem eticheta ciclurile.

Exemplu

Vom rescrie modelul pentru calculul funcției cosinus, în noul model se va dori calculul doar a primilor 10 termeni ai seriei. Declararea entității va rămâne neschimbată. Corpul arhitecturii se va modifca astfel:

```
architecture fixed_length_series of cos is
begin
        summation : process (theta) is
                variable sum, term : real;
        begin
                sum := 1.0;
                term := 1.0;
                for n in 1 to 9 loop
                        term := (–term) * theta**2 / real(((2*n–1) * 2*n));
                        sum := sum + term;
                end loop;
                result <= sum;
        end process summation;
end architecture fixed_length_series;
```

Instrucțiuni aserțiune

Unul dintre scopurile pentru care se scriu modele ale sistemelor de calcul este acela de a verifica că ele funcționează corect. Putem testa parțial un model prin aplicarea de valori pe intrări şi verificând că ieşirile se comportă conform cerintelor. În cazul în care ieşirile nu se comportă conform cerințelor, se trece la verificarea proiectării pentru a depista eroarea. Acest task poate fi făcut mult mai uşor prin intermediul *instrucțiunilor aserțiune* care verifică dacă condițille cerute sunt îndeplinite sau nu de către model. O instrucțiune aserțiune este o instrucțiune secvențială, ea putând fi inclusă în orice corp de proces. Sintaxa este următoarea:

```
assertion_statement
        assert boolean_ expression
                [ report  expression ] [ severity  expression ] ;
```

Forma cea mai simplă a unei instrucțiuni de aserțiune include cuvântul cheie **assert** urmat de o expresie booleană care se presupune a fi adevărată când instrucțiunea aserțiune este executată. În cazul în care condiția nu este îndeplinită, apare o excepție. În cazul în care excepția apare pe durata simulării modelului, simulatorul raportează acest fapt.

Exemplu

Dacă vom scrie:

```
assert initial_value <= max_value;
```

şi *initial_value* este mai mare decât *max_value*, când instrucțiunea este executată de către simultor el va anunța o excepție.

Putem forța simulatorul să ne ofere extra informații prin introducerea clauzei **report** în cadrul instrucțiuni de aserțiune astfel:

```
assert initial_value <= max_value
        report "valoarea inițială este prea mare";
```

VHDL predefineşte un tip enumerare **severity_level** astfel:

```
type severity_level is (note, warning, error, failure);
```

O valoare a acestui tip poate fi inclusă într-o clauză **severity** a unei instrucțiuni aserțiune. Această valoare ne va indica gradul cu care excepția va afecta operarea normală a modelului.

Exemplu

```
assert packet_length /= 0
        report "pachet de rețea primit este gol"
        severity warning;

assert clock_pulse_width >= min_clock_width
        severity error;
```

Dacă se omite clauza **report**, mesajul default întors de simulator este “Assertion violation”. Dacă se va omite clauza **severity**, valoarea default întoarsă de simulator este **error**. Valoarea **severity** este uzual utilizată de simulator pentru a determina dacă se va continua execuția modelului după apariția unei excepții.

Exemplu

Presupunem un registru care lucrează pe frontul pozitiv al ciclului de ceas. Datele de intrare sunt eşantionate, memorate şi transmise către ieşire. Presupunem că intrarea de ceas rămâne în '1' logic pentru cel puțin 5ns. Modelul este prezentat mai jos:

```
entity edge_triggered_register is
        port ( clock : in bit;
                d_in : in real; d_out : out real );
end entity edge_triggered_register;
-------------------------------------------------------
architecture check_timing of edge_triggered_register is
begin
        store_and_check : process (clock) is
                variable stored_value : real;
                variable pulse_start : time;
        begin
                case clock is
                        when '1' =>
                                pulse_start := now;
                                stored_value := d_in;
```

```
                    d_out <= stored_value;
                when '0' =>
                    assert now = 0 ns or (now – pulse_start) >= 5 ns
                        report "pulsul de ceas prea scurt";
            end case;
        end process store_and_check;
end architecture check_timing;
```

Când ceasul comută din '0' în '1', intrarea este memorată dar şi timpul curent de simulare (prin inetrmediul funcției predefinite **now**) prin intermediul variabilei **pulse_start**. Când ceasul comută din '1' în '0', diferența dintre **pulse_start** şi timpul curent de simulare este verificată prin intermediul instrucțiunii aserțiune.

Tipul ARRAY şi operații

Un *array* conține o colecție de valori, care sunt toate de acelaşi tip. Poziția fiecărui element într-un array este dată de o valoare scalară denumita *index*. Pentru a crea un array de obiecte într-un model, mai întâi trebuie definit tipul array-ului printr-o declarație de tip. Sintaxa este următoarea:

```
array_type_definition
    array (  discrete_range ) of element_ subtype_indication
```

Această declarație defineşte un tip de array prin specificarea rangului index-ului şi tipul elementelor ce compun array-ul sau subtipul. *discrete_range* poate fi definit ca un subset de valori dintr-un tip discret.

```
discrete_range
    type_mark
    I  simple_expression ( to I downto )  simple_expression
```

Exemple

Declararea unui array care reprezintă datele sub formă de cuvinte

```
type word is array (0 to 31) of bit;
```

Fiecare element este un bit, iar elementele sunt indexate de la 0 până la 31. O declarație alternativă este (mai apropiată de sistemele *little-endian*)

```
type word is array (31 downto 0) of bit;
```

Valorile index ale unui array pot să nu fie numerice. Dacă avem următoarea declarare a tipului enumerare:

```
type controller_state is (initial, idle, active, error);
```

putem declara un array astfel:

```
type state_counts is array (idle to error) of natural;
```