LUCRAREA NR. 4
OPERATORI. TIPURI DE DATE

1. Scopul lucrării

În lucrare se prezintă toţi operatorii din VHDL, precum şi literalii utilizaţi în cadrul limbajului. Sunt analizate pe larg tipurile de date utilizate în cadrul unei descrieri VHDL, împărţite în cele patru mari categorii: *tipul scalar, tipul compus, tipul acces* şi *tipul fişier*. Se subliniază diferenţele dintre anumite tipuri de date care pot crea confuzii.

2. Consideraţii teoretice

2.1 Operatori

În VHDL există şapte clase de operatori, fiecărei clase fiindu-i atribuit un nivel de prioritate unic. Această prioritate face ca, atunci când se evaluează o expresie complexă, operanzii să fie afectaţi operatorului cu prioritatea cea mai mare, apoi să se aplice acest operator, şi aşa mai departe.

Lista claselor de operatori, în ordinea crescătoare a priorităţilor (clasa cu prioritatea cea mai mică este prima) este următoarea:

1. *Operatorii logici*: **and**, **or**, **nand**, **nor**, **xor** şi **xnor**;
2. *Operatorii relaţionali*: =, /=, <, <=, > şi >=;
3. *Operatorii de deplasare*: **sll**, **srl**, **sla**, **sra**, **rol**, **ror**;
4. *Operatorii de adunare*: +, - şi **&**;
5. *Operatorii de semn*: + şi –;
6. *Operatorii de înmulţire*: ***, /, **mod** şi **rem**;
7. *Operatorii diverşi*: ****, **abs**, **not**.

Într-o expresie, ordinea priorităţilor poate evita scrierea de paranteze inutile. Cu toate acestea, putem folosi paranteze dacă nu suntem siguri de priorităţi sau dacă dorim să ameliorăm lizibilitatea programului.

În VHDL există posibilitatea de a *supraîncărca* operatorii, adică de a le conferi o nouă semnificaţie. Această operaţie nu va modifica însă prioritatea lor, care este fixată o dată pentru totdeauna.

2.1.1 Operatorii logici: **and**, **or**, **nand**, **nor**, **xor** şi **xnor**

Aceşti operatori sunt predefiniţi în vederea realizării operaţiilor logice „ŞI”, „SAU”, „ŞI-NU”, „SAU-NU”, „SAU-EXCLUSIV” şi „COINCIDENŢĂ” pe operanzi de tip BOOLEAN (FALSE, TRUE) şi BIT (‘0’, ‘1’);

Operatorii logici sunt funcţionali şi pe tablouri uni-dimensionale (numite şi *vectori*) de elemente de tip BOOLEAN sau BIT, cu condiţia ca operanzii să aibă aceeaşi lungime (acelaşi număr de elemente). O particularitate a primilor patru operatori: operandul lor din dreapta nu este obligatoriu evaluat. Într-adevăr, dacă după evaluarea operandului din stânga se obţine un rezultat care determină rezultatul operaţiei (de exemplu, o valoare FALSE pentru operatorul **and**), compilatorul nu va mai lua în considerare operandul din dreapta şi va câştiga astfel, în anumite cazuri, un timp preţios.

2.1.2 Operatorii relaţionali: =, /=, <, <=, > şi >=

Rezultatul acestor operatori este de tip BOOLEAN. Operatorii de egalitate şi de inegalitate (respectiv „=” şi „/=”) sunt definiţi pe toate tipurile de date din VHDL, cu excepţia tipului fişier (**file**).

La compararea a două tipuri compuse (tipuri a căror valoare este compusă din mai multe elemente), sunt luate în considerare toate elementele respective. În cazul comparării a două tipuri acces (poantori), compararea se referă exclusiv la adresele obiectelor referite. Aceşti operatori sunt definiţi pe toate tipurile enumerate. Ordinea prezentă la definirea tipului enumerat este păstrată, primul element numit fiind considerat drept cel mai mic. Astfel, la definirea tipului BOOLEAN, elementul FALSE, fiind declarat primul, este mai mic decât TRUE.

2.1.3 Operatorii de deplasare: **sll**, **srl**, **sla**, **sra**, **rol**, **ror**

Această clasă conţine şase operatori binari (cu doi operanzi) care operează exclusiv pe tablouri uni-dimensionale (vectori) de tip BIT sau BOOLEAN.

a) **sll** (**S**hift **L**eft **L**ogical) – efectuează o deplasare logică la stânga: elementul cel mai din stânga este pierdut (*lost value*), iar un alt element numit *valoare de umplere* (*fill value*) soseşte din partea dreaptă. Această valoare de umplere este ‘0’ pentru tablourile de tip BIT şi FALSE pentru cele de tip BOOLEAN. Singura

posibilitate de a schimba valoarea de umplere este supraîncărcarea operatorului.
Presupunând că A este un vector de elemente de tip BIT, avem:

```
B := A sll 0;    -- valoarea lui B este identică cu a lui A,
                 -- operaţie nulă
B := A sll 2;    -- valoarea lui B este identică cu a lui A
                 -- deplasată la stânga cu două poziţii; în
                 -- partea dreaptă apar doi biţi de '0'
B := A sll -3;   -- valoarea lui B este identică cu a lui A
                 -- deplasată la DREAPTA cu trei poziţii. O
                 -- valoare negativă inversează deci sensul
                 -- deplasării
```

b) **srl** (**S**hift **R**ight **L**ogical) – efectuează o deplasare logică la dreapta: este operaţia complementară celei precedente;
c) **sla** (**S**hift **L**eft **A**rithmetic) – efectuează o deplasare aritmetică la stânga: elementul cel mai din stânga (semnul) este deplasat şi duplicat;

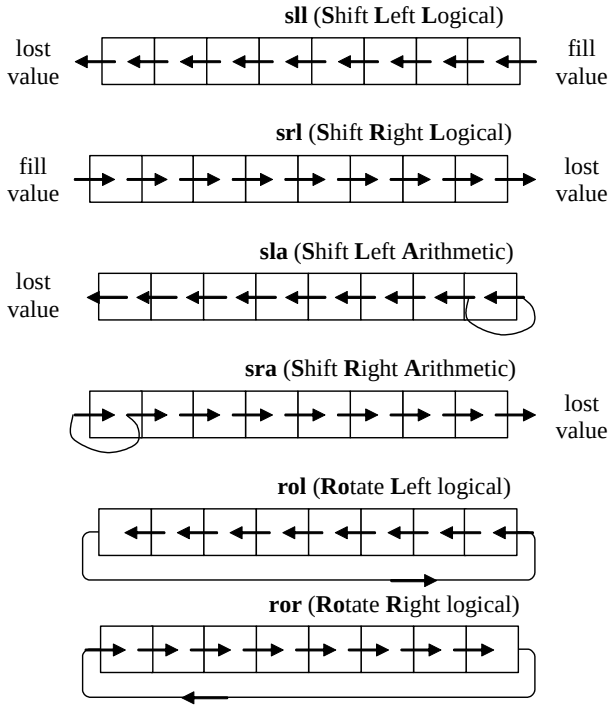

Figura 4.1 *Operatorii de deplasare şi de rotire pe biţi*

d) **sra** (**S**hift **R**ight **A**rithmetic) – efectuează o deplasare aritmetică la dreapta. Este operatorul complementar celui precedent: elementul cel mai din dreapta este deplasat şi duplicat;
e) **rol** (**Ro**tate **L**eft) – efectuează o rotaţie spre stânga: elementul cel mai din stânga dispare şi reapare în partea dreaptă;
f) **ror** (**Ro**tate **R**ight) – efectuează o rotaţie spre dreapta: elementul cel mai din dreapta dispare şi reapare în partea stângă.

2.1.4 Operatorii de adunare: **+, -, &**

Aceştia sunt operatori binari (cu doi operanzi) de adunare şi scădere („+" şi „-"), la care se adaugă operatorul de concatenare („&").

Concatenarea este definită pe tablourile uni-dimensionale (vectori). Un şir de caractere (tipul STRING) este un vector de caractere; este deci evident modul de aplicare a operatorului de concatenare. De exemplu, operaţia "ROMÂ" & "NIA" are drept rezultat "ROMÂNIA".

2.1.5 Operatorii de semn: **+, -**

Aceştia sunt operatori unari, adică se aplică pe un operand unic. Dacă vom supraîncărca operatorii de adunare, operatorii de semn nu vor fi supraîncărcaţi în mod automat, ci această supraîncărcare a lor va trebui făcută în mod explicit.

Observaţie

Operatorii de semn au o prioritate mai mică decât operatorii de înmulţire, ceea ce nu este normal.

Astfel, expresia 2* -3, care ar trebui să ia valoarea –6, va genera o eroare (nu se poate evalua înmulţirea înaintea evaluării operatorului unar „-"). Introducerea de paranteze va rezolva problema (se va scrie: 2*(-3)).

2.1.6 Operatorii de înmulţire: ***, /,* mod, rem**

Operaţia de împărţire pe tipuri de date întregi efectuează o rotunjire prin trunchierea părţii fracţionare.

Următoarele relaţii sunt verificate în VHDL:

```
(-A)/B = -(A/B) = A/(-B)
```

Operatorul **rem** (de la *remainder*) reprezintă restul împărţirii întregi definite mai sus. Prin urmare, presupunând că A şi B sunt numere întregi, vom avea:

```
A = (A/B) * B + (A rem B)
```

A **rem** B are semnul lui A.

Operatorul **mod** (de la *modulo*) este definit de următoarea relaţie:

```
A = B * N + (A mod B) --N este un număr întreg oarecare
```

A **mod** B e mai mic ca B (în valoare absolută) şi are semnul lui B.

2.1.7 Operatorii diverşi: ******, **abs, not**

Operatorul ** este numit *exponenţial* (ridicare la putere). Acest operator se aplică unui operand aflat la stânga sa, care este de un tip întreg sau flotant. Operandul din dreapta (puterea) trebuie obligatoriu să fie un întreg.

Operatorul **abs** returnează valoarea absolută a oricărui tip numeric.

not este un operator logic unar. El operează deci asupra obiectelor de tip BOOLEAN, BIT şi asupra vectorilor cu elemente de aceste tipuri.

2.2 Literali

În VHDL, noţiunea de literal este puţin diferită faţă de alte limbaje de programare cunoscute. De exemplu, 123 este un literal. Literalii se clasifică în:

- literali numerici (123, 3.14);
- literali enumeraţi;
- literali şiruri de caractere sau şiruri de biţi;
- literalul **null** – reprezintă orice obiect de tip acces neiniţializat.

În mod evident, literalul nu poartă cu el tipul său, ci numai familia sa de tipuri.

În VHDL sunt autorizate numai 95 de caractere ASCII (din setul ASCII complet au fost excluse caracterele de control). Aceste caractere sunt notate între apostrofuri: ‘x’, ‘Y’, ‘7’, ‘*’, ‘ ‘ (spaţiu) etc.

Şirurile de caractere se notează între ghilimele. Ele nu trebuie să depăşească o linie de cod sursă. În cadrul lor se face deosebirea dintre litere mari şi mici. Câteva exemple: “Salut”, “T78%0-#”, “1010”.

Uneori se poate dovedi utilă scrierea unui şir de caractere pe două linii – acest lucru se poate face folosind simbolul de concatenare. Iată un exemplu:

“Se poate” &
“scrie pe două linii”.

Pentru numerele zecimale utilizate, un element interesant îl constituie posibilitatea folosirii liniuţei de subliniere (*underscore*) pentru delimitarea grupelor de cifre zecimale, atât pentru numere întregi cât şi pentru numere reale. Iată câteva exemple: 22, 1971 sau 1_971, 84000000 sau 84_000_000 sau 84E6 sau 84E+6, 7.0, 1971.0 sau 1_971.0, 84000000.0 sau 84_000000.0 sau 84.0E6 sau 84.0E+6.

Numerele întregi şi cele reale pot fi exprimate în diferite baze, de la baza 2 (sistemul numeric binar) până la baza 16 (sistemul numeric hexazecimal). Notaţia este următoarea: întâi se exprimă baza (în zecimal), după care urmează numărul propriu-zis, încadrat de caracterul # (diez).

Iată câteva exemple de reprezentare a numărului întreg 255:

2#1111_1111# sau 4#3333# sau 10#255# sau 16#FF#

Iată câteva exemple de reprezentare a numărului real 255.25:

2#1111_1111.01# sau 2#1.111111101#E7 sau 4#3333.1# sau 16#FF.4#

În cazul folosirii tipului BIT standard, în VHDL se obişnuieşte să se folosească notaţia prin şiruri de biţi (aşa-numitul *bit string*). Acest tip de date este exportat de pachetul STANDARD şi poate lua valorile ‘0’ sau ‘1’.

Notaţia prin şiruri de biţi permite iniţializarea vectorilor de biţi. Valoarea se scrie între ghilimele (la fel ca în cazul şirurile de caractere) şi este precedată de litera B pentru scrierea binară, O pentru scrierea octală şi X pentru scrierea hexazecimală. Astfel, X”ABC” este echivalent cu B”101010111100”, iar O”427” este echivalent cu B”100010111”. X”ABC” este echivalent şi cu ”101010111100” şi poate deci să fie utilizat oriunde ”101010111100” este valid.

Fiecare expresie are un tip şi o valoare în momentul evaluării sale.

2.3 Tipuri de date

VHDL este un limbaj puternic tipizat: fiecare constantă, variabilă sau semnal trebuie să aibă un tip înainte de utilizarea sa. De asemenea, parametrii unei proceduri sau ai unei funcţii, precum şi valoarea returnată de funcţia respectivă, sunt obiecte tipizate.

Există patru mari tipuri de date în VHDL:

- *tipurile scalare* – valoarea lor este constituită dintr-un singur element;
- *tipurile compuse* – valoarea lor este constituită din mai multe elemente;
- *tipurile acces* sau *poantorii* (*pointers*);
- *tipurile fişier*.

Observaţie

Semnalele nu pot avea tipul acces; nici măcar un sub-element al unui semnal nu poate avea acest tip.

Numai variabilele pot avea tipul acces; fişierele constituie o familie cu totul specială de tipuri. Semnul de întrebare din figura 4.1 indică faptul că fişierele pot fi văzute ca variabile a căror valoare nu poate fi modificată.

	Constante	**Variabile**	**Semnale**	**Fişiere**
Scalare	DA	DA	DA	NU
Compuse	DA	DA	DA	NU
Acces	NU	DA	NU	NU
Fişier	NU	NU (?)	NU	DA

Figura 4.1 *Clase de obiecte şi familii de tipuri*

2.3.1 Tipurile scalare

Din această familie fac parte următoarele tipuri:

a) tipurile enumerate;
b) tipurile întregi;
c) tipurile flotante;
d) tipurile fizice.

Toate aceste tipuri sunt ordonate (există o relaţie de ordine definită asupra lor), aşadar valorile lor vor putea fi comparate cu ajutorul operatorilor relaţionali: =, /=, <, <=, > şi >=.

Valorile posibile ale unui tip scalar pot fi restrânse prin indicarea unui interval de validitate. Notaţia pentru acest interval este următoarea:

```
range expresie1 to expresie2
```

Dacă expresie1 este mai mare decât expresie2, intervalul va fi nul (vid). În acest caz se pot inversa capetele intervalului de definiţie:

```
range expresie3 downto expresie4
```

a) Tipurile enumerate

Toate declaraţiile de tip încep cu cuvântul cheie **type**. Pentru declararea acestui tip de date, este suficientă indicarea valorilor simbolice (identificatori sau caractere) pe care le poate lua. De exemplu:

```
type CULOARE is (ROŞU, GALBEN, ALBASTRU);
type STARE is (INIŢIALIZARE, CITIRE, SCRIERE, AŞTEPTARE);
type ZI is (LUNI,MARŢI,MIERCURI,JOI,VINERI,SÂMBĂTĂ,DUMINICĂ);
type CULOARE_MAŞINĂ is (ROŞU, ALB, VERDE);
```

Un acelaşi simbol (în exemplul de mai sus, simbolul ROŞU) poate să aparţină mai multor tipuri enumerate (se spune că apare o *supraîncărcare*). Contextul acestei supraîncărcări trebuie să permită eliminarea ambiguităţii.

În pachetul STANDARD sunt predefinite următoarele tipuri enumerate: BOOLEAN, BIT, CHARACTER şi SEVERITY_LEVEL.

```
type BOOLEAN is (FALSE, TRUE);
type BIT is ('0', '1');
type SEVERITY_LEVEL is (NOTE, WARNING, ERROR, FAILURE);
```

Tipul CHARACTER conţine enumerarea tuturor caracterelor admise într-un program VHDL.

Simbolurile utilizate în cadrul unui tip enumerat vor fi numai identificatori sau caractere. În cadrul tipurilor enumerate apare noţiunea de

poziţie. Primul element din definiţia tipului primeşte poziţia 0, următorul poziţia 1 etc. Atributul predefinit POS, care se apelează la fel ca o funcţie, returnează poziţia unui element al unui tip enumerat. Astfel, BOOLEAN'POS(TRUE) are valoarea 1. Această poziţie este cea care induce relaţia de ordine asupra elementelor. Operatorii de comparare sunt disponibili în cadrul tipurilor enumerate, ceea ce va face ca, de exemplu, TRUE să fie mai mare decât FALSE.

b) Tipurile întregi

Tipurile numerice întregi pot fi văzute ca tipuri enumerate (ele reprezintă un şir ce poate fi enumerat). Pe toate aceste tipuri sunt definiţi operatorii aritmetici. Toate tipurile numerice întregi pot fi convertite *implicit* într-un tip virtual numit UNIVERSAL_INTEGER, aspect care asigură compatibilitatea tuturor tipurilor numerice.

Pentru a declara un tip numeric întreg este suficient să indicăm intervalul pe care este definit cu ajutorul cuvântului cheie **range**. De exemplu, porţiunea de cod VHDL de mai jos reprezintă declaraţia unui tip enumerat întreg care variază între 1 şi 22, inclusiv capetele intervalului:

```
type DISTANŢA is range 1 to 22;
```

În pachetul STANDARD există un tip predefinit şi deosebit de util: INTEGER.

```
type INTEGER is range -2_147_483_648 to 2_147_483_647;
```

De fapt, capetele acestui interval (care pot fi aflate cu ajutorul atributelor INTEGER'LOW şi INTEGER'HIGH) variază cu lăţimea cuvintelor calculatorului pe care rulează compilatorul VHDL.

c) Tipurile flotante

La fel ca în cazul tipurilor numerice întregi, declaraţia unui tip flotant se face prin specificarea capetelor intervalului său de definiţie (aceste capete ale intervalului de definiţie trebuie să fie şi ele flotante):

```
type FLOTANTUL_MEU is range 1.63 to 3.14;
```

Operatorii aritmetici se pot aplica pe toate aceste tipuri. Toate tipurile numerice flotante pot fi convertite *implicit* într-un tip virtual numit UNIVERSAL_REAL. Există un tip real predefinit în pachetul STANDARD:

```
type REAL is range -16#0.7FFFFF8#E+32 to 16#0.7FFFFF8#E+32;
```

Capetele acestui interval depind de implementarea specifică a analizorului VHDL, însă standardul VHDL precizează că intervalul trebuie să fie de cel puţin [–1E38, 1E38] (în notaţie zecimală), cu o precizie de minimum 6 cifre după virgulă (această condiţie este respectată în definiţia de mai sus, deoarece notaţia utilizată este cea hexazecimală).

d) Tipurile fizice

Aceste tipuri sunt foarte apropiate de tipurile întregi. În limbajul VHDL există noţiunea de *unitate de cantitate*. De exemplu, în pachetul STANDARD este definit tipul TIME, care este un tip fizic. Acesta este singurul tip fizic predefinit şi el este utilizat de către simulator.

Un tip fizic este caracterizat de:

- Unitatea de bază (femtosecunda, pentru tipul TIME);
- Intervalul valorilor autorizate;
- O eventuală colecţie de sub-unităţi împreună cu corespondenţele lor.

În cele ce urmează este prezentată definiţia tipului TIME din pachetul STANDARD. Intervalul valorilor sale autorizate depinde de implementare. În varianta de mai jos este propusă o codificare pe 64 de biţi:

```
--Tipul TIME predefinit
type TIME is range -9_223_372_036_854_775_808 to
-9_223_372_036_854_775_807    -- Codificare pe 64 de biţi!
      units fs;               -- femtosecundă
            ps = 1000 fs;     -- picosecundă
            ns = 1000 ps;     -- nanosecundă
            us = 1000 ns;     -- microsecundă
            ms = 1000 us;     -- milisecundă
            sec = 1000 ms;    -- secundă
            min = 60 sec;     -- minut
            hr = 60 min;      -- oră
end units;
```

Astfel, atunci când utilizăm o expresie de tip TIME, putem scrie la fel de bine 20 fs, sau 30 us, sau 3 min. De asemenea, expresia 345 fs + (347 ps/ 20) + 2 ns are sens şi valoarea sa este 2_017_695 fs.

Observaţie

Între număr şi unitatea de măsură există întotdeauna un caracter „spaţiu". Acest aspect nu este specificat explicit în norma VHDL, aşa că fiecare constructor de instrumente VHDL realizează propria sa implementare, în care poate să ţină sau nu cont de aceasta.

Pe tipurile fizice sunt definite toate operaţiile aritmetice. Toate valorile unui tip fizic şi toate valorile de conversie între unităţi trebuie să fie întregi. Practic, aceasta implică să avem întotdeauna unitatea de bază – cea mai mică dintre toate unităţile.

Putem construi orice tip fizic pe baza modelului oferit de tipul TIME. De exemplu, în manualul de referinţă al VHDL este oferit următorul exemplu:

```
type DISTANCE is range 0 to 1 E16
      units A;                        -- angstrom
            nm    = 10         A;     -- nanometru
            um    = 1000       nm;    -- micron
            mm    = 1000       um;    -- milimetru
            cm    = 10         mm;    -- centimetru
            m     = 1000       mm;    -- metru
            km    = 1000       m;     -- kilometru
            mil   = 254000     A;     -- mil
            inch  = 1000       mil;   -- inch
            ft    = 12         inch;  -- foot
            yd    = 3          ft;    -- yard
            mi    = 5280       ft;    -- milă
            lg    = 3          mi;    -- leghe
      end units;
```

Aşadar, se poate scrie:

```
variable Dis1, Dis2 : DISTANCE;
........
Dis1 := 18 mm;
Dis2 := 2 cm - 1 mm;
if Dis1 < Dis2 then ...
```

2.3.2 Tipurile compuse

Din această familie fac parte următoarele tipuri:

- *Tablourile* - care reprezintă o colecţie de obiecte de acelaşi tip;
- *Articolele* - care reprezintă o colecţie de obiecte de tipuri diferite.

a) Tablourile

Structurile omogene se descriu în VHDL prin tablouri. De exemplu, o magistrală de semnale văzută ca o structură omogenă de biţi poate fi reprezentată printr-un tablou. Acest lucru este făcut în pachetul STANDARD prin declararea tipului BIT_VECTOR.

Elementele acestor structuri sunt accesibile cu ajutorul unuia sau mai multor *indecşi* (numărul acestora poate fi oarecare). Un tabel cu un singur index este numit *vector*.

Definirea unui tablou necesită specificarea tipului şi a numărului indecşilor şi a tipului elementelor tabloului. Tipul indecşilor poate fi orice tip discret (enumerat sau întreg). Tipul elementelor poate fi oricare, cu excepţia tipului fişier.

Există două tipuri de tablouri:

- *tablourile constrânse*, în care intervalul de variaţie a indecşilor este cunoscut cu anticipaţie. Sensul de variaţie al indecşilor este specificat cu ajutorul cuvintelor cheie **to** şi **downto**. Iată câteva definiţii de tablouri constrânse:

```
type CUVÂNT is array (0 to 21) of BIT;
type LOGIC4 is ('X', '0', '1', 'Z');
type INTRARE is array (0 to 7) of LOGIC4;
--Utilizând tipul POSITIVE din pachetul STANDARD:
constant JOS: INTEGER := 7;
constant SUS: INTEGER := 22;
type INTERVAL is array (POSITIVE range JOS to SUS) of BIT;
```

- *tablourile neconstrânse*, în care intervalul de variaţie al indecşilor nu va fi cunoscut decât în momentul execuţiei codului (în timpul simulării). Pentru tablourile neconstrânse, simbolul „<>”, care se citeşte box, permite amânarea definirii unui interval de indexare şi a unei direcţii de variaţie până în momentul execuţiei. În pachetul STANDARD sunt definite două tablouri neconstrânse:

```
type BIT_VECTOR is array (NATURAL range <>) of BIT;
type STRING is array (POSITIVE range <>) of CHARACTER;
```

b) Articolele

Spre deosebire de tablouri, articolele pot conţine elemente de tipuri diferite. Aceste elemente se mai numesc şi câmpuri şi sunt desemnate prin nume şi nu prin index, ca în cazul tablourilor. Pentru a descrie un articol, este suficient să enumerăm câmpurile care fac parte din el între cuvintele cheie **record** şi **end record**.

Iată un exemplu de definiţie a unui tip articol:

```
type CAPACITATE is range 0 to 1_000_000_000
      units pF;                -- picofarad
            nF = 1000 pF;      -- nanofarad
            uF = 1000 nF;      -- microfarad
      end units;
type REZISTENŢĂ is range 0 to 1_000_000_000
      units Ohm;               -- ohm
            kOhm = 1000 Ohm;   -- kiloohm
            MOhm = 1000 kOhm; -- megaohm
      end units;
type CIRCUIT_RC is
 record
 PREŢ: INTEGER; -- preţul de vânzare
 CAP_INTERNĂ: CAPACITATE; -- tipul fizic definit anterior
 REZ_INTERNĂ: REZISTENŢĂ; -- tipul fizic definit anterior
 end record;
```

Elementele sau câmpurile sunt selectate prin „notaţia cu punct”. Să presupunem că avem următorul semnal şi următoarea variabilă:

```
signal A: CIRCUIT_RC; -- un semnal de tipul declarat anterior
variable B: CIRCUIT_RC; -- o variabilă de acelaşi tip
```

În cazul semnalului A: *A. PREŢ* este de tipul INTEGER, *A. CAP_INTERNĂ* este de tipul CAPACITATE şi *A.REZ_INTERNĂ* este de tipul REZISTENŢĂ. Aceleaşi notaţii sunt valabile şi în cazul variabilei B.

Asignările pot fi făcute individual:

```
A. CAP_INTERNĂ <= B. CAP_INTERNĂ after 100 ns;
```

În pilotul (*driver*-ul) semnalului A, elementului CAP_INTERNĂ i se va atribui valoarea elementului CAP_INTERNĂ a variabilei B, la 100 ns de la începerea simulării (ne referim la timpul simulatorului, nu este vorba despre un timp real).

Este de asemenea posibilă asignarea simultană de valori tuturor câmpurilor:

```
A <= B;
```

De asemenea, este posibil ca tuturor câmpurilor articolului să le fie asignate valori dintr-o dată, cu ajutorul unui agregat. Iată două exemple:

```
B : = ('10_000', 35 pF, 7 kOhm); --Notaţia poziţională
B := (PREŢ => '20_000', REZ_INTERNĂ => 700 kOhm, CAP_INTERNĂ
=> 70 pF); --Notaţia după nume
```

c) Notaţia prin agregare

Un *agregat* constituie o modalitate de a indica valoarea unui tip compus. Agregatul se scrie între paranteze, elementele fiind separate prin virgule. Verificarea corespondenţei dintre tipurile valorilor asignate şi tipurile elementelor tipului compus vizat este efectuată de către compilator.

Vom exemplifica pe următoarele tipuri de date:

```
--Tablou de şapte elemente de tip întreg
type TAB is array (1 to 7) of INTEGER;
--Articol de trei elemente
type ART is record
            C1: INTEGER;
            C2: BIT;
            C3: INTEGER;
            end record;
```

În cazul unui agregat există trei posibilităţi distincte de notare:

- *Asocierea poziţională* constă pur şi simplu în a înşira elementele ca într-o listă. Asocierea dintre câmpul articolului sau al indicelui tabloului se face graţie poziţiei sale din listă. Această ordine corespunde celei din declaraţia tipului compus. Pe exemplele de mai sus, putem nota:
  - pentru tipul TAB: (2,4,6,8,10,12,14)

- pentru tipul ART: (6,'0',8)

- *Asocierea prin denumire* permite precizarea numelui elementului (numele câmpului sau al indexului) şi valoarea corespunzătoare. Se utilizează simbolul „=>" pentru marcarea acestei corespondenţe. Ordinea câmpurilor nu mai are importanţă. Pentru a realiza aceleaşi asignări ca şi mai sus, notaţia este următoarea:
  - pentru tipul TAB: (1=>2, 2=>4, 3=>6, 4=>8, 5=>10, 6=>12, 7=>14)
  - pentru tipul ART: (C1=>6, C2=>'0', C3=>8)

  sau:
  - pentru tipul TAB: (1=>2, 7=>14, 3=>6, 5=>10, 2=>4, 6=>12, 4=>8)
  - pentru tipul ART: (C2 => '0', C3 =>8, C1 => 6)
- *Asocierea mixtă* este un amestec al primelor două notaţii. Ea trebuie neapărat să înceapă cu partea sa poziţională, apoi să se încheie cu asocierile după nume. Utilizarea cuvântului cheie **others**, care desemnează toate câmpurile sau indecşii (în cazul tabloului) neasignaţi încă, permite efectuarea „dintr-o dată" a acestei asignări. De exemplu:
  - pentru tipul TAB: (2, **others** =>0)
  - pentru tipul ART: (C2 => '1', **others** =>0)

  ceea ce este echivalent cu:
  - pentru tipul TAB: (2, 0, 0, 0, 0, 0, 0)
  - pentru tipul ART: (0, '1', 0)

  Există câteva cazuri particulare:
  - Un agregat poate să nu conţină decât cuvântul cheie **others** urmat de valoarea pe care dorim să o asignăm tuturor elementelor. Această notaţie se utilizează adeseori atunci când se urmăreşte iniţializarea tuturor elementelor unui tablou cu o aceeaşi valoare;
  - Un agregat care conţine un singur element trebuie să fie exprimat folosind asocierea prin denumire, pentru a evita orice ambiguitate cu o expresie dată între paranteze;
  - În cazul utilizării clauzei **others** pentru a asigna o valoare câmpurilor unui articol, aceste câmpuri trebuie să fie toate de acelaşi tip.

2.3.3 Tipurile acces

Un tip acces este adeseori numit *poantor* (*pointer*). Un tip acces „indică" („poantează") spre un obiect de un tip definit anterior. Rostul unui asemenea tip este, în principal, cel de a permite crearea dinamică (adică, în timpul rulării) a unor noi obiecte. Cuvântul cheie utilizat pentru desemnarea acestui tip este **access**.

Acest tip de date permite *alocarea dinamică a memoriei*. Stilul de manipulare a obiectelor de acest tip este identic cu cel al limbajelor de programare clasice (C, Pascal etc.). Prin urmare, sunt valabile toate regulile şi metodele de programare referitoare la: alocarea dinamică a obiectelor de tip acces, dealocarea zonei de memorie corespunzătoare ocupate, crearea structurilor de date dinamice specifice (liste simplu şi dublu înlănţuite, arbori etc.) şi modul de operare cu ele (inserarea unui element, ştergerea unui element etc.) Aceste aspecte fiind foarte cunoscute în limbajele de programare clasice, care constituie totuşi o cerinţă preliminară pentru însuşirea limbajului VHDL, ele nu vor fi detaliate aici.

Toate tipurile acces sunt de aceeaşi natură (fizic, ele reprezintă *adresa unui obiect*) dar, pentru o mai mare siguranţă, ele sunt tipizate în funcţie de tipul obiectului spre care poantează. Aşadar, un poantor (*pointer*) spre un întreg va fi diferit de un poantor (*pointer*) spre un flotant. La declarare, tipurile acces sunt iniţializate cu o valoare implicită care este **null**.

În continuare este necesar să alocăm o zonă de memorie pentru stocarea obiectelor spre care se poantează. Aceasta se face cu ajutorul instrucţiunii speciale **new**. De foarte multe ori – dar nu obligatoriu – putem profita de această alocare pentru a iniţializa valoarea obiectului poantat.

După ce obiectul şi-a încheiat ciclul de viaţă, procedura DEALLOCATE (declarată automat) va elibera zona de memorie ocupată.

Iată câteva declaraţii de tip acces:

```
type P_INT is access INTEGER; -- Poantor la întreg
variable A,B: P_INT;
type COMPLEX is
      record
            NUME: STRING (1 to 7);
            DATA: TIME;
            VALOAREA: BIT;
      end record;
type P_ COMPLEX is access COMPLEX; -- Poantor la un articol
variable ELEM: P_ COMPLEX;
```

La acest nivel al declaraţiilor, egalităţile A = **null**, B = **null** şi ELEM = **null** sunt adevărate.

Câteva exemple de alocări:

```
A := new INTEGER'(22); -- A poantează la întregul 22;
B := new INTEGER; -- B poantează la un întreg. Iniţializarea
                  -- implicită este la INTEGER'LOW, cel mai
                  -- mic număr întreg care poate fi codificat
-- ELEM poantează la un articol: câmpul NUME va fi
-- iniţializat la "ADUNAT", câmpul DATA va fi iniţializat la
-- 50 ns, iar câmpul VALOAREA va fi iniţializat la '1'
ELEM := new COMPLEX'("ADUNAT", 50 ns, '1');
```

Dealocarea se face simplu:

```
DEALLOCATE(A);
```

Obiectul A va deveni astfel inaccesibil, iar locul ocupat de el în memorie va putea fi utilizat în alte scopuri. Această operaţie trebuie efectuată prin program, altminteri existând riscul ca toată zona de memorie rezervată alocărilor dinamice să fie „consumată".

2.3.4 Tipurile fişier

Fişierele sunt foarte importante în VHDL. Ele pot fi folosite, de exemplu, pentru încărcarea conţinutului unui modul de memorie ROM sau pentru a defini un ansamblu de stimuli. Acest stil de lucru va evita necesitatea de a recompila proiectul, ceea ce va duce la o importantă reducere a timpului de proiectare efectiv. De asemenea, fişierele sunt folosite pentru interfaţarea cu alte instrumente de dezvoltare, ele aflându-se la baza interacţiunii cu utilizatorul (pentru că acesta e „văzut" ca un fişier).

Toate fişierele sunt cu acces secvenţial: valorile scrise în / citite dintr-un fişier sunt plasate în secvenţă, unele după celelalte, ca pe o bandă magnetică.

La declararea unui fişier trebuie indicat şi tipul de date al elementelor din care este alcătuit:

```
type FIŞIER_DE_PROGRAMARE is file of BIT;
type FIŞIER_TEXT is file of STRING;
type FIŞIER_DE_DATE is file of INTEGER;
```

Există câteva limitări în privinţa tipurilor de date care se pot afla într-un fişier. Un fişier nu poate conţine:

- poantori (*pointer*-i) sau tipuri compuse;
- alte obiecte de tip fişier;
- tablouri (cu excepţia vectorilor).

Declararea unui tip fişier are ca efect crearea implicită a trei subprograme care vor permite citirea din şi scrierea în fişierul respectiv.

a) Procedura de citire (READ)

Această procedură are ca parametru de intrare un tip fişier şi returnează valoarea următoare citită din acest fişier. Aici, parametrul formal corespunzător fişierului de intrare se numeşte F. El este de tipul fişier TF care se presupune că a fost declarat anterior. VALUE este parametrul formal de ieşire; el este de tipul TM (fişierul TF a fost definit ca un fişier conţinând date de tipul TM).

```
procedure READ (F: in TF; VALUE: out TM);
```

b) Procedura de scriere (WRITE)

Este complementara procedurii de citire (READ). Ea primeşte o valoare pe care o va scrie la sfârşitul fişierului dat.

```
procedure WRITE(F: out TF; VALUE: in TM);
```

c) Funcţia ENDFILE

Această funcţie primeşte un parametru de intrare de tip fişier şi ne permite să aflăm dacă am ajuns la sfârşitul fişierului. Ea returnează valoarea FALSE dacă se mai poate citi cel puţin încă o valoare din fişierul respectiv.

Dacă fişierul a fost utilizat în mod **out** (în scriere), rezultatul boolean este întotdeauna adevărat. În cazul în care se va încerca efectuarea unei operaţii de citire (READ) asupra unui fişier pentru care apelul funcţiei ENDFILE ar returna valoarea „adevărat”, se va genera o eroare la simulare.

```
function ENDFILE (F: in TF) return BOOLEAN;
```

Observaţie

Pentru tablourile neconstrânse, procedura READ returnează, pe lângă valoarea citită din fişier, şi lungimea tabloului. Atunci, profilul funcţiei nu este cel de mai sus, ci următorul:

```
procedure READ(F: in TF; VALUE: out TM; LENGTH: out NATURAL);
```

Majoritatea intrărilor curente făcându-se sub formă de şiruri de caractere, pachetul TEXTIO permite o gestiune mai fină a acestora. Acest pachet standard este descris în lucrarea nr. 12.

Începând cu versiunea VHDL'93, au apărut două proceduri de deschidere, respectiv de închidere de fişiere: FILE_OPEN şi FILE_CLOSE. Foarte utilă în anumite cazuri, această facilitate trebuie totuşi utilizată cu precauţie. Într-adevăr, a avea posibilitatea de a deschide şi de a închide un fişier le permite unor procese concurente să comunice, „ascunzând" însă acest fapt mecanismului de simulare al VHDL, care se bazează pe conceptele de semnal şi de eveniment. Rezultatele pot fi neaşteptate…

Cu ocazia efectuării declaraţiei:

```
type tipul_meu_de_fişier is file of tipul_meu_de_obiect;
```

vor fi efectuate implicit următoarele declaraţii:

```
procedure FILE_OPEN (file F: tipul_meu_de_fişier;
                     EXTERNAL_NAME: in STRING;
                     OPEN_KIND: in FILE_OPEN_KIND := READ_MODE);

procedure FILE_OPEN (STATUS: out FILE_OPEN_STATUS;
                     file F: tipul_meu_de_fişier;
                     EXTERNAL_NAME: in STRING;
                     OPEN_KIND: in FILE_OPEN_KIND := READ_MODE);
```

Se observă că a doua supraîncărcare oferă un parametru de ieşire (STATUS) care permite testarea unei erori de tipul „fişier inexistent".

În acelaşi mod, sunt definite implicit şi disponibile următoarele proceduri şi funcţii:

```
procedure FILE_CLOSE (file F: tipul_meu_de_fişier);
procedure WRITE (file F: tipul_meu_de_fişier,
                 VALUE: in tipul_meu_de_obiect);
function ENDFILE (file F: tipul_meu_de_fişier)return BOOLEAN;
procedure READ(file F: tipul_meu_de_fişier, --Pentru obiecte
               VALUE: out tipul_meu_de_obiect); --constrânse
-- Pentru obiecte neconstrânse, cum ar fi STRING sau
-- BIT_VECTOR; astfel se returnează şi lungimea acestora
procedure READ(file F: tipul_meu_de_fişier,
      VALUE: out tipul_meu_de_obiect, LENGTH: out NATURAL);
```

2.4 Expresii calificate

Rolul calificării expresiilor este de a înlătura o ambiguitate referitoare la tipul unei expresii sau al unui agregat, indicând explicit numele acestui tip. Nu este o conversie de tip, ci o informaţie transmisă compilatorului pentru a-l ajuta să rezolve o ambiguitate.

Sintaxa calificării este următoarea:

```
tipul_obiectului'(expresie) -- Prima variantă
tipul_obiectului'agregat -- A doua variantă
```

Elementul important (care nu trebuie uitat!) este apostroful; fără el, calificarea se transformă într-o conversie de tip.

În continuare prezentăm un exemplu de calificare a expresiilor în contextul pachetului TEXTIO (pachetul standard de intrări şi ieşiri). În cadrul acestui pachet sunt declarate mai multe proceduri WRITE, printre care şi următoarele două:

```
procedure WRITE (L: inout LINE; VALUE: in BIT_VECTOR;
            JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH:=0);
procedure WRITE (L: inout LINE; VALUE: in STRING;
            JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH:=0);
```

Acesta este un exemplu de supraîncărcare, cele două proceduri fiind diferite din punct de vedere al profilului. Diferenţa este datorată exclusiv celui de-al doilea parametru, care în prima procedură are tipul BIT_VECTOR, iar în cea de-a doua are tipul STRING.

Următoarea instrucţiune, în care LINIE_CRT este o variabilă de tip LINE, va crea o ambiguitate pe care compilatorul nu o poate rezolva de unul singur:

```
WRITE (LINIE_CRT, "1010");
```

Ambiguitatea constă în faptul că nu se ştie dacă se face referire la vectorul de patru biţi 1,0,1,0 sau la şirul de caractere "1010". Compilatorul nu va putea alege pe care dintre cele două proceduri s-o apeleze.

Soluţia constă în utilizarea calificării expresiei. De exemplu, dacă ne referim la un şir de caractere, vom scrie:

```
WRITE (LINIE_CRT, STRING'("1010")); -- Atenţie la apostrof!
```

Această calificare reprezintă o indicaţie, un ajutor oferit compilatorului. Ea nu este o operaţie şi nu consumă nici spaţiu de memorie, nici timp de execuţie suplimentar.

Observaţie

Notaţia aleasă, "1010" este deosebit de ambiguă. Acelaşi exemplu rămâne valabil şi pentru notaţia "QWERTY", chiar dacă ambiguitatea cu tipul BIT_VECTOR pare exclusă (din punct de vedere uman): compilatorul nu va face presupuneri în privinţa conţinutului, ci va reacţiona la notaţia "”!

2.5 Conversii de tipuri

Între două tipuri cu structuri apropiate se pot efectua conversii explicite. Conversia este foarte restrictivă şi nu este autorizată decât în următoarele trei cazuri:

- Conversii de tipuri cu reprezentare întreagă sau flotantă;
- Conversii de tablouri (cu anumite restricţii);
- Conversia unui tip în el însuşi.

Restricţiile menţionate în cazul tablourilor se referă la faptul că acestea trebuie să aibă:

- aceleaşi dimensiuni;
- acelaşi tip de elemente;
- indecşii lor trebuie să fie cel puţin convertibili.

Sintaxa generală a conversiei de tip este următoarea:

```
Tipul_spre_care_se_face_conversia (valoarea_de_convertit)
```

Trebuie ca tipul valorii de convertit să fie cunoscut fără ambiguitate şi ca respectiva conversie să fie autorizată.

Iată un exemplu de conversie:

```
variable A,B: REAL;
variable I: INTEGER;
...
A := B*REAL(I);          -- În ambele cazuri este vorba despre
I : = INTEGER(A*B);      -- o operaţie de înmulţire reală
```

Spre deosebire de expresiile calificate, în acest caz se realizează efectiv o operaţie. De pildă, în cazul liniei a doua din exemplul de mai sus, variabilei I îi este atribuit numărul întreg cel mai apropiat de rezultatul real al înmulţirii lui A cu B.

3. Desfăşurarea lucrării

3.1 Se va exersa definirea tuturor tipurilor de date şi operatori. Se va crea un tip fizic numit VOLUM şi un tip fizic VALUTĂ pentru conversia între diferite monede.

3.2 Se va crea o listă simplu înlănţuită cu elemente de tip **acces**, cu alocări şi dealocări.

3.3 Se va crea un modul REGISTRU_DE_DEPLASARE, folosind operatorii cei mai adecvaţi pentru gestionarea datelor interne.

3.4 Se va crea o unitate aritmetică-logică, în care intrările şi ieşirile se vor exprima sub formă de articole şi care poate efectua următoarele operaţii, pe cei doi operanzi de intrare de tip întreg INTEGER: AND, OR, Înmulţire cu o putere a lui 2, Împărţire la un număr putere a lui 2, adunare şi scădere.