LUCRAREA NR. 10
SUB-PROGRAME

1. Scopul lucrării

Lucrarea urmăreşte însuşirea noţiunilor de bază referitoare la sub-programe în VHDL. Există două tipuri de sub-programe: *procedurile* şi *funcţiile*, fiecare dintre ele fiind analizate şi comentate în detaliu. Se discută de asemenea problema *supraîncărcării* atât a funcţiilor şi a procedurilor, cât şi a operatorilor.

2. Consideraţii teoretice

2.1 Descrierea concurentă

Sub-programele permit scrierea unor algoritmi reutilizabili. Valorile parametrilor pot fi diferite la fiecare apel, obţinându-se astfel efecte diferite. Sub-programele pot fi folosite pentru mărirea lizibilităţii unui program, modularizând codul sursă, chiar dacă unele dintre ele nu vor fi apelate decât o singură dată.

În practică, sub-programele se folosesc atunci când apare necesitatea unei secvenţe de instrucţiuni (aşadar, respectivele instrucţiuni vor fi secvenţiale şi nu concurente) pentru descrierea unei anumite operaţii complexe, a unei conversii, a anumitor porţiuni din descrierea unor procese sau a unor funcţii de rezoluţie pentru semnale multi-sursă (funcţiile de rezoluţie au fost prezentate pe larg în lucrarea nr. 8).

Există două tipuri de sub-programe:

- *procedurile* (cuvântul cheie **procedure**);
- *funcţiile* (cuvântul cheie **function**).

Procedurile pot acţiona prin *efecte laterale*. Efectul lateral este, prin definiţie, o modificare (sau o consultare) a mediului printr-un mijloc care diferă de parametrii de ieşire ai procedurilor, semnalele de mod **out** ale entităţilor sau valorile returnate de către funcţii. De exemplu, asignarea unei valori unui semnal global (care nu este declarat local, ci în cadrul unui

pachet) constituie un efect lateral. Procedurile pot modifica, eventual, şi valoarea parametrilor transmişi la apel.

Funcţiile returnează un rezultat (şi numai unul) şi nu acţionează prin efecte laterale. Ele nu pot citi sau scrie variabile sau semnale decât dacă acestea sunt declarate în zona de declaraţii a funcţiei sau în zona declarativă a sub-programelor apelate. Cu alte cuvinte, dacă funcţia va fi apelată de mai multe ori cu aceleaşi valori ale parametrilor, ea va returna de fiecare dată acelaşi rezultat.

Observaţie

O excepţie la această regulă o constituie funcţia NOW care returnează data curentă a simulării şi care nu are parametri. Această funcţie se numeşte *impură*.

Începând cu VHDL'93 a fost introdusă o nouă categorie de funcţii, numite *impure* care au posibilitatea de a crea efecte laterale. Ele se declară folosind cuvântul cheie **impure**, fiind totodată supuse unor restricţii asupra utilizării lor:

- o funcţie de rezoluţie nu poate fi o funcţie impură;
- o funcţie impură nu poate fi apelată dintr-o funcţie pură (cu alte cuvinte, ne-impură).

Compilatorul va verifica întotdeauna aceste restricţii. În mod simetric, s-a introdus şi cuvântul cheie **pure**: el permite explicitarea caracterului pur al unei funcţii. Acest caracter este totodată atribuit în mod implicit oricărei funcţii, dacă nu se specifică cuvântul cheie **impure**.

Apelul unei proceduri este o instrucţiune, în vreme ce apelul unei funcţii se trece, la fel ca o valoare, în membrul drept al simbolului de asignare.

Un sub-program este alcătuit din două părţi: *declaraţia* (opţională) şi *corpul* său.

Declaraţia unui sub-program (numită şi „specificaţie de sub-program") va indica:

- *genul* sub-programului (procedură sau funcţie);
- *numele* acestuia;

- *lista parametrilor* săi (parametrii formali). Pentru fiecare dintre aceşti parametri, declaraţia sub-programului va preciza *modul* (de intrare, de ieşire, de intrare / ieşire) şi *tipul* său;
- *tipul valorii returnate* (în cazul unei funcţii).

Corpul sub-programului va conţine algoritmul implementat; el va fi perfect coerent cu declaraţia sub-programului. Compilatorul are sarcina de a garanta această compatibilitate.

2.2 Declaraţia de sub-program

Declaraţia de sub-program este opţională. De regulă, în partea de specificare a pachetelor, vom găsi declaraţia tuturor sub-programelor pe care respectivul pachet le exportă. Corpul acestor sub-programe se va găsi în corpul pachetului.

Declaraţia furnizează numeroase informaţii despre funcţionalitatea sub-programului, în special dacă beneficiază de câteva linii de comentarii. Ea poate fi deci văzută ca „manualul de utilizare" a sub-programului.

Sintaxa declaraţiilor de sub-program este următoarea:

```
-- Declaraţia unei proceduri
procedure nume_procedură (lista_parametrilor_formali);

-- Declaraţia unei funcţii
{pure/impure} function nume_funcţie (lista_parametri_formali)
return tipul_rezultatului;
```

În cazul procedurii, numele este pur şi simplu un identificator (la fel ca în cazul variabilelor sau semnalelor).

În cazul funcţiei, numele poate fi un identificator, însă şi simbolul unui operator (cum ar fi +, =, & sau **and**). În acest caz, simbolul se va scrie între ghilimele ("+") şi funcţia va putea fi activată sub forma unui operator.

Lista parametrilor formali are următoarea sintaxă:

```
{clasă_obiect} nume_param_1{, nume_param_2}: {mod_transmitere}
      type valoare_implicită;
```

această declaraţie putându-se repeta de un anumit număr de ori.

Există trei moduri de transmitere a parametrilor:

1. *Modul intrare* (cuvântul cheie **in**) este modul de transmitere implicit. El corespunde unor parametri care pot fi citiţi în cadrul sub-programului, dar care nu pot fi modificaţi. Prin urmare, în interiorul acestui sub-program, ei nu pot apărea în membrul stâng al nici unei instrucţiuni de asignare şi nici ca parametri de apel ai vreunei proceduri în alt mod decât modul **in**;
2. *Modul ieşire* (cuvântul cheie **out**) este modul unui parametru formal a cărui valoare este returnată de către sub-program. În cazul de faţă, este vorba despre o procedură, căci acest mod le este interzis parametrilor formali ai unei funcţii (o funcţie nu returnează decât valoarea sa). În interiorul unei proceduri, este interzisă citirea unui parametru formal având modul ieşire – prin urmare, acest parametru nu va fi niciodată întâlnit în membrul drept al unei instrucţiuni de asignare. În plus, el nu poate fi folosit nici ca parametru actual de intrare sau de intrare / ieşire în cazul apelării unui alt sub-program intern;
3. *Modul combinat intrare / ieşire* (cuvântul cheie **inout**) autorizează orice citire şi scriere în interiorul unei proceduri. Din aceleaşi motive ca şi cele expuse în cazul modului de ieşire (**out**), el nu poate fi un parametru formal al unei funcţii.

Un parametru care posedă o valoare implicită poate fi omis la apelul unui sub-program. Orice semnal are întotdeauna o valoare implicită asociată. Valoarea implicită este acea valoare care îi este atribuită unui parametru atunci când apelul nu precizează nimic în această direcţie. Această valoare este o expresie care trebuie să fie de acelaşi tip ca şi parametrul şi este evaluată o dată pentru totdeauna în decursul elaborării (înaintea execuţiei programului VHDL).

În VHDL există trei clase de obiecte:

- Constante;
- Variabile;
- Semnale.

Restricţiile existente asupra clasei unui parametru în funcţie de modul său sunt date în tabelul de mai jos (figura 10.1):

Clasă / Mod	Obiect de clasă constantă	Obiect de clasă variabilă	Obiect de clasă semnal
Parametru de mod **in** în proceduri	Permis şi luat implicit	Permis	Permis
Parametru de mod **out** în proceduri	Permis	Permis şi luat implicit	Permis
Parametru de mod **inout** în proceduri	Permis	Permis şi luat implicit	Permis
Parametru de mod **in** în funcţii	Permis şi luat implicit	NU	Permis
Parametru de mod **out** în funcţii	NU	NU	NU
Parametru de mod **inout** în funcţii	NU	NU	NU

Figura 10.1 *Relaţia dintre clase şi modul de transmitere a parametrilor formali*

Sintaxa prezentată în figura 10.1 arată că, în cazul definirii unui parametru formal, nu este obligatoriu să se precizeze dacă este vorba despre o constantă, o variabilă sau un semnal. Tabelul de mai sus ilustrează clasele care sunt luate implicit în funcţie de modul parametrului.

Iată câteva exemple de declaraţii de sub-programe:

```
procedure VERIF_PERIOADA (signal TACT: in BIT);
procedure CONVERSIE (INTRARE : in REAL; IEŞIRE: out INTEGER);
function CONVERSIE (INTRARE : in REAL) return INTEGER;
function "+"(A, B: INTEGER) return INTEGER;
```

Următoarele porţiuni de cod sunt variante perfect echivalente:

```
procedure MIN (A, B: INTEGER; C: out INTEGER);
procedure MIN (A: INTEGER; B: INTEGER; C: out INTEGER);
procedure MIN (A, B: in INTEGER; C: out INTEGER);

function MIN (A, B: INTEGER) return INTEGER;
function MIN (A: INTEGER; B: INTEGER) return INTEGER;
function MIN (A: in INTEGER, B: in INTEGER) return INTEGER);
```

2.3 Corpul de sub-program

Corpul sub-programului are menirea de a realiza funcţionalitatea sub-programului. Sintaxa sa este următoarea:

```
Antet_sub-program is
{zona declarativă} --Acoladele delimitează părţile opţionale
begin
{zona rezervată instrucţiunilor}
end {nume_sub-program}
```

Corpul unui sub-program va conţine algoritmul ataşat sub-programului respectiv. Antetul acestui corp începe prin copierea specificaţiei sub-programului, singura diferenţă constând în faptul că ea se încheie cu cuvântul cheie **is** şi nu cu punct şi virgulă („;”).

Urmează apoi o parte declarativă care va conţine declaraţiile interne acestui sub-program. Declaraţiile nu vor fi vizibile din exteriorul sub-programului.

În această zonă se pot găsi şi alte declaraţii sau corpuri de sub-program, tipuri şi sub-tipuri, constante şi variabile, fişiere, alias-uri şi atribute. Este permisă specificarea de atribute şi prezenţa clauzei **use**. Cea mai importantă restricţie o constituie aceea referitoare la semnale: în cadrul unui sub-program este interzisă declararea semnalelor, deoarece ne aflăm în domeniul secvenţial (aici, după încheierea execuţiei sale, sub-programul „moare”, în timp ce semnalul ar continua să existe!). Zona de declaraţii se termină o dată cu cuvântul cheie **begin,** care indică începutul porţiunii care conţine algoritmul sub-programului.

Această porţiune este alcătuită dintr-o suită de instrucţiuni care se execută secvenţial. Ea începe deci cu cuvântul cheie **begin** şi se încheie cu cuvântul cheie **end**, urmat eventual de numele sub-programului. În zona de algoritmi, o funcţie returnează valoarea sa prin intermediul instrucţiunii **return**, pe când o procedură va asigna valori parametrilor de mod **out** şi **inout**.

De exemplu: *funcţia* MIN returnează cel mai mic dintre cei doi întregi pe care îi primeşte ca parametri. Partea sa declarativă (opţională) este vidă. Se foloseşte instrucţiunea de selecţie simplă **if … then… else… end if**.

```
function MIN (A, B: INTEGER) return INTEGER is
begin
      if A < B then
            return A;
      else
            return B;
      end if;
end MIN;
```

Să examinăm în continuare *procedura* MIN. Partea declarativă a rămas în continuare vidă:

```
-- Prima variantă a procedurii MIN
procedure MIN (A, B: in INTEGER; C: out INTEGER) is
begin
      if A < B then
            C := A;
      else
            C := B;
      end if;
end MIN;
-- A doua  variantă a procedurii MIN: se foloseşte în plus o
-- variabilă locală (numai din considerente demonstrative)
procedure MIN (A, B: in INTEGER; C: out INTEGER) is
variable VAR_LUCRU: INTEGER;
begin
      if A < B then
            VAR_LUCRU := A;
      else
            VAR_LUCRU := B;
      end if;
      C := VAR_LUCRU;
end MIN;
```

2.4 Apelul de sub-program

Deşi un sub-program este de natură secvenţială (instrucţiunile pe care le conţine sunt executate secvenţial), apelul său se poate face în două moduri:

a) ca o acţiune secvenţială;
b) ca o acţiune concurentă.

Există două moduri de a indica parametrii unui sub-program în momentul apelării sale: *prin poziţie* sau *prin denumire*.

Apelul prin poziţie constă în a enumera toţi parametrii, în ordine, separaţi prin virgule. Corespondenţa dintre parametrii actuali şi parametrii formali se efectuează în funcţie de această ordine.

Apelul prin denumire constă în a indica numele parametrului formal înaintea fiecărui parametru actual (parametrul de apel). Corespondenţa este atunci explicită şi se utilizează simbolul „=>”. În acest caz, ordinea parametrilor nu mai este semnificativă. Apelul prin denumire se utilizează pentru ameliorarea lizibilităţii programelor sau pentru a nu modifica decât câţiva parametri, presupunându-se că ceilalţi au o valoare implicită.

Prezentăm în continuare câteva exemple: procedura şi respectiv funcţia MIN va fi apelată în ambele moduri (prin poziţie şi prin denumire).

```
-- Procedura MIN
MIN (VAR, 5, REZULTAT); -- Apel poziţional
MIN (A => VAR, B => 5, C => REZULTAT); -- Apel prin denumire
MIN (B => 5, C => REZULTAT, A => VAR); -- Apel prin denumire

-- Funcţia MIN
REZULTAT := MIN(VAR, 5); -- Apel poziţional
REZULTAT := MIN(A => VAR, B => 5); -- Apel prin denumire
```

2.5 Supraîncărcarea

Două sub-programe se numesc *supraîncărcate* dacă au acelaşi nume, însă profilurile lor diferă.

Profilul unui sub-program este un ansamblu de informaţii care cuprinde numărul, ordinea şi tipul parametrilor formali, precum şi – în cazul funcţiilor – tipul rezultatului returnat.

Supraîncărcarea a două sub-programe de profiluri diferite este deosebit de utilă şi măreşte uşurinţa scrierii de cod sursă VHDL, asigurând în acelaşi timp o foarte bună lizibilitate a programelor. Pentru a determina sub-programul pe care trebuie efectiv să-l apeleze, compilatorul va alege profilul care corespunde apelului. Dacă nici un profil nu corespunde sau dacă există mai multe profiluri identice candidate, acest fapt va fi semnalat ca eroare la compilare.

Observaţie

Anumite informaţii conţinute în specificaţia unui sub-program, cum ar fi numele parametrilor formali, modul lor de transmitere, clasa lor sau valoarea lor implicită nu fac parte din profilul sub-programului. Două sub-programe care nu diferă decât prin una sau mai multe dintre aceste informaţii vor provoca o ambiguitate care va fi semnalată de către compilator. Prin urmare, următoarele două funcţii nu se vor supraîncărca (se va genera o eroare la compilare, deoarece simbolul MIN este definit de două ori):

```
function MIN(A, B: INTEGER) return INTEGER;
function MIN(C, D: INTEGER) return INTEGER;
```

Iată câteva exemple de supraîncărcare:

```
-- Supraîncărcarea a trei proceduri
procedure CONVERSIE (INTRARE: in REAL, IEŞIRE: out INTEGER);
procedure CONVERSIE (INTRARE: in BIT, IEŞIRE: out INTEGER);
procedure CONVERSIE (INTRARE: in OCTET, IEŞIRE: out INTEGER);

-- Supraîncărcarea a trei funcţii
function MIN(A, B: INTEGER) return INTEGER;
function MIN(A: REAL, B: REAL) return REAL;
function MIN(A: in BIT, B: in BIT) return BIT;
```

Supraîncărcarea poate fi aplicată şi operatorilor. De exemplu, putem supraîncărca operatorul logic **and** folosind următoarea funcţie:

```
function "AND"(A,B: TIP_LOGIC) return BOOLEAN;
```

Această declaraţie va autoriza scrierea unor expresii precum:

```
A and B
```

unde A şi B sunt de tipul TIP_LOGIC.

Funcţia astfel definită trebuie să aibă respectiv unul singur sau doi parametri, în funcţie de tipul operatorului: unar sau binar. Operatorii „+" şi „-" pot fi supraîncărcaţi atât ca operatori binari cât şi ca operatori unari.

Utilizarea acestei funcţii va putea întotdeauna să se efectueze sub forma sa clasică:

```
"AND" (A, B) -- Nu trebuie uitate ghilimelele!
```

Supraîncărcarea operatorilor prezintă un interes deosebit pentru simplificarea scrierii programelor care operează pe semnale de alte tipuri decât tipul BIT.

Iată trei exemple de supraîncărcare a operatorului „+":

```
function "+" (A,B: BIT) return BIT;
function "+" (A,B: BIT_VECTOR) return BIT_VECTOR;
function "+" (A,B: IMPEDANŢĂ) return IMPEDANŢĂ;
```

2.6 Consideraţii suplimentare

2.6.1 Recursivitatea

În VHDL există posibilitatea de a scrie programe recursive. Într-un limbaj de programare clasic, în general vorbind, recursivitatea reprezintă proprietatea unui sub-program de a se putea apela pe el însuşi. În VHDL este permisă şi recursivitatea încrucişată (sub-programul A apelează sub-programul B, care la rândul său apelează sub-programul A). În cazul aplicării recursivităţii, trebuie să ne asigurăm că am scris o instrucţiune de test pentru stoparea sa.

Astfel, este posibil să descriem un sumator complet pe N biţi în funcţie de el însuşi, instanţiat pe N-1 biţi, şi a unui modul de un singur bit care va avea rolul de test de stopare a recursivităţii.

Trebuie însă să avem în vedere că, în VHDL, scopul final îl reprezintă în general obţinerea unei structuri hardware. De aceea, este bine să ne punem mereu întrebarea: la ce ne ajută acest gen de descriere? Va exista oare o realitate materială (hardware) corespunzătoare? În acest caz, răspunsul este din păcate negativ…

2.6.1 Vizibilitatea

În figura 10.2 sunt prezentate regulile de vizibilitate a sub-programelor în VHDL:

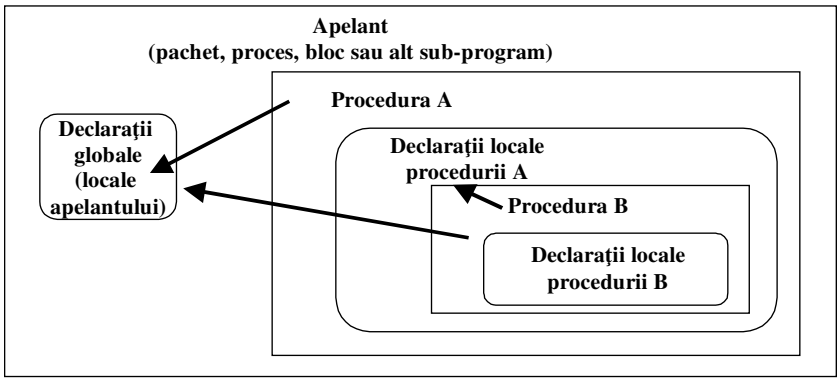

Figura 10.2 *Reguli de vizibilitate*

Declaraţiile efectuate în interiorul unui sub-program nu sunt vizibile (deci nu sunt accesibile) din exteriorul acestuia, însă sunt vizibile de către procedurile înglobate. Săgeţile din figura 10.2 pornesc de la obiectul care „vede" către obiectul „văzut".

2.6.3 Logica sub-programelor

O procedură se execută până la **end**-ul său final.

O funcţie nu trebuie să ajungă niciodată la instrucţiunea sa **end** finală, lucru care ar genera o eroare în cursul execuţiei. Ea trebuie să întâlnească întotdeauna cuvântul cheie **return** şi să predea atunci controlul (împreună cu valoarea pe care trebuie s-o returneze) apelantului.

3. Desfăşurarea lucrării

3.1 Se vor supraîncărca operatorii de adunare şi scădere definiţi pe întregi pentru valori de tip STD_LOGIC şi STD_LOGIC_VECTOR.

3.2 Se vor supraîncărca funcţiile MIN şi MAX din lucrare pentru parametri de tip STD_LOGIC şi STD_LOGIC_VECTOR.

3.3 Se vor defini funcţiile de conversie între următoarele tipuri de date: BIT_VECTOR şi INTEGER; BIT_VECTOR şi STRING.

3.4 Se va defini un sistem numeric care rezolvă ecuaţii de gradul 2 folosind proceduri.