LUCRAREA NR. 2
UNITĂŢI FUNDAMENTALE DE PROIECTARE

1. Scopul lucrării

Se prezintă unităţile fundamentale de proiectare în VHDL: *perechea entitate / arhitectură*, *configuraţiile*, *specificaţiile de pachete* şi *corpurile pachetelor*. Se studiază sintaxa declaraţiilor respective, precum şi cadrul lor de utilizare. Se detaliază avantajele şi dezavantajele specifice operării cu unităţile respective.

2. Consideraţii teoretice

2.1 Noţiuni introductive

VHDL este un limbaj bazat pe biblioteci, ceea ce înseamnă că după compilarea unui fişier sursă corect, rezultatul este stocat într-o bibliotecă, gestionată de către VHDL; cel mai adesea, ea este stocată într-un director sau subdirector pe care de obicei proiectantul nu are de ce să-l acceseze.

Există anumite părţi ale programelor care pot fi compilate separat: acestea sunt *unităţile de proiectare*. Fiecare compilare reuşită a unui fişier (realizată de către sistemul de operare sau de mediul de dezvoltare VHDL) va fi susceptibilă să producă în biblioteca curentă una sau mai multe unităţi de proiectare. Mai multe unităţi de proiectare pot fi scrise în acelaşi fişier, dar o unitate de proiectare nu poate fi divizată în mai multe fişiere.

Se recomandă delimitarea cât mai exactă a noţiunilor de *unitate de proiectare* şi de *fişier*: o unitate de proiectare va avea propriul său fişier, chiar dacă acest lucru nu este obligatoriu. Această abordare permite evitarea recompilării inutile a unităţilor de proiectare.

În cazul prezenţei mai multor unităţi de proiectare în acelaşi fişier, există posibilitatea decelării unei erori în cadrul uneia dintre ele; în consecinţă, analiza efectuată de compilator nu va fi dusă până la capăt. În acest caz, compilatorul va analiza şi va compila unităţile de proiectare din fişier care se află *înaintea* unităţii incriminate. În anumite cazuri, el va

încerca şi chiar va reuşi să compileze unităţile situate *după* unitatea de proiectare respectivă.

Deci noţiunea de fişier se pierde complet după compilare: nu se mai vorbeşte despre fişiere, ci numai de unităţi de proiectare (se va vorbi despre fişiere în VHDL la nivelul tipurilor de date - acest concept este cu totul diferit de cel discutat aici).

2.2 Unităţi de proiectare primare şi secundare

În VHDL există în total cinci tipuri de unităţi de proiectare diferite, dintre care trei sunt considerate „primare" şi două „secundare". O unitate primară descrie o vedere externă, pe când o unitate secundară descrie vederea internă.

De exemplu, *specificaţia unui pachet* va descrie sub-programele pe care le propune (pe care le *exportă*) pachetul, tipurile sau obiectele manipulate de acesta, însă fără a intra în detaliile (algoritmul) implementării. Specificaţia unui pachet este o unitate de proiectare primară.

Partea de implementare a pachetului (*corpul pachetului*) conţine algoritmii utilizaţi. Această informaţie nefiind primordială pentru cel care utilizează pachetul, ea îi este ascunsă. Corpul pachetului este o unitate de proiectare secundară. Entitatea este o unitate de proiectare primară, în vreme ce arhitectura este o unitate de proiectare secundară.

Există o analogie fundamentală, care merită să fie reţinută, între *specificaţia unui pachet* şi *entitate* - ca vederi externe ale unor cutii negre (hardware sau software). De altfel, se poate deduce imediat şi analogia dintre *corpul pachetului* şi *arhitectură*, aceste două unităţi de proiectare servind la descrierea realizării cutiei negre (hardware sau software).

Cele patru unităţi de proiectare sunt completate de o a cincia: *configuraţia*. Această unitate face legătura între instanţe de componente utilizate într-o arhitectură şi realizarea lor efectivă prin intermediul unor perechi entitate / arhitectură. Ea este considerată a fi o unitate primară.

În concluzie, cele cinci tipuri de unităţi de proiectare sunt:

- Entitatea;
- Arhitectura;
- Configuraţia;
- Specificaţia pachetului;
- Corpul pachetului.

2.3 Biblioteci

În VHDL putem avea un număr mare de biblioteci, fiecare dintre ele conţinând mai multe unităţi de proiectare.

La un moment dat, una singură dintre aceste biblioteci este numită *bibliotecă de lucru*, celelalte fiind *biblioteci de resurse*. Alegerea bibliotecii de lucru este făcută în afara limbajului, printr-o comandă specifică platformei VHDL utilizate. Ea poate fi referită în VHDL prin numele logic WORK. Concret, în această bibliotecă vor fi introduse unităţile de proiectare compilate cu succes. Unităţile de proiectare primare aparţinând bibliotecilor de resurse pot fi referite. Pentru aceasta este necesar, mai întâi de toate, ca biblioteca în care se găsesc să fie referită de o clauză **library**, care trebuie să preceadă unitatea de proiectare care o apelează. În continuare, referirea unităţii se poate face printr-o clauză **use** într-o zonă de declaraţii oarecare (fără a uita sufixul **.all** pentru pachete, dacă dorim să accesăm totalitatea „exporturilor" unui pachet). Aşadar, se scrie:

```
library biblioteca_în_care_este_păstrată_unitatea_primară;
...
use numele_entităţii;   -- dacă unitatea primară e o
                        --specificaţie de entitate

use numele_configuraţiei;     -- dacă este o configuraţie
use numele_pachetului.all;    -- dacă este un pachet
```

Bibliotecile WORK şi STD (care conţine pachetele aşa-zis *standard*) fac obiectul unei clauze **library** implicite, ceea ce înseamnă că fiecare unitate de proiectare este implicit precedată de clauzele **library** şi **use**:

```
library WORK, STD;
use STD.STANDARD.all;
```

2.4 Entitate, arhitectură şi configuraţie

Orice sistem fizic comunică cu exteriorul prin intermediul unor semnale care sunt grupate în aşa-zisa sa *interfaţă* cu exteriorul. Semnalele primite de sistemul respectiv sunt orientate dinspre exterior către sistem şi sunt numite *semnale de intrare* sau - mai simplu - *intrări* sau *stimuli*. Reacţionând la stimulii primiţi, sistemul va genera, la rândul său, *semnale de ieşire* sau *răspunsuri*, transmise mediului său exterior.

Interfaţa este un element fundamental al oricărui sistem fizic (în categoria de sistem putem include, în această abordare, şi sistemele vii - fiinţele), în lipsa sa fiind de neconceput interacţiunea sistemului cu exteriorul. În realitate nu există sistem care să fie complet izolat de mediul său exterior.

În VHDL, interfaţa este descrisă cu ajutorul declaraţiei de *entitate,* iar structura internă a sistemului - cu ajutorul declaraţiei de *arhitectură*.

2.4.1 Entitatea

Entitatea descrie interfaţa unică dintre arhitecturi şi mediul exterior, iar arhitectura descrie funcţionalitatea entităţii. Ansamblul poate fi văzut şi utilizat, la un nivel superior, ca o componentă.

În funcţie de gradul de fineţe sau, pur şi simplu, de nivelul de descriere (structural, comportamental, flux de date sau hibrid) care a fost ales pentru descrierea funcţionalităţii, vom putea avea mai multe arhitecturi diferite pentru aceeaşi entitate.

Această grupare în entităţi şi arhitecturi conferă o mare modularitate modelării în VHDL. Modificarea unei arhitecturi sau înlocuirea sa nu va avea nici o repercusiune asupra unităţilor de proiectare care fac referire la entitatea corespunzătoare. Entitatea rămâne identică şi constituie singurul element cunoscut de către mediul exterior; mediul nu „vede" arhitectura.

Dimpotrivă, informaţiile cuprinse în entitate sunt cunoscute de către diversele sale arhitecturi. Toate aceste informaţii sunt opţionale. Ele sunt de trei feluri:

- declaraţii de parametri generici;
- declaraţii de porturi;
- instrucţiuni.

Sintaxa unei entităţi este următoarea:

```
entity numele_entităţii is
      {generic (listă de parametri generici)}
      {port (listă de porturi)}
      {begin
            listă de instrucţiuni concurente}
end {numele_entităţii}
```

Acoladele delimitează informaţiile opţionale.

Numele entităţii este un identificator care trebuie să fie unic în biblioteca dată. Pentru a desemna o pereche entitate / arhitectură, se notează:

```
numele_entităţii(numele_arhitecturii)
```

Numele entităţii, numit în mod formal *identificator*, are în primul rând un rol de documentare. Din acest motiv, se recomandă ca acest nume să fie alcătuit din substantive care să descrie cât mai exact menirea entităţii respective. Regulile pe care trebuie să le respecte orice identificator au fost prezentate în lucrarea nr. 1.

Lista parametrilor generici permite crearea unor entităţi foarte flexibile şi deci *reutilizabile* - aspect care va fi detaliat în continuare.

Lista porturilor oferă o mulţime de informaţii relative la semnalele care interfaţează entitatea. Se indică:

- numărul semnalelor;
- tipul acestora;
- valoarea lor implicită;
- sensul lor (intrare, ieşire, bidirecţional etc.).

Lista de instrucţiuni concurente permite scrierea o singură dată a prelucrărilor care sunt globale pentru toate arhitecturile aferente. În practică, restricţiile impuse[1] limitează această porţiune de cod la un rol de control al intrărilor entităţii (sau al parametrilor, în cazul entităţilor generice).

Să luăm exemplul următorului sumator complet:

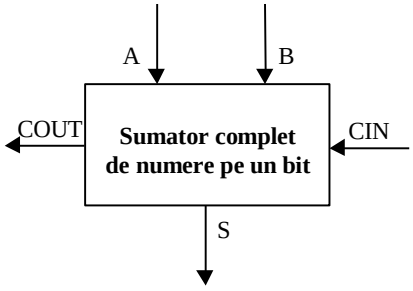

Figura 2.1 *Vederea externă a unui sumator complet de numere pe un bit*

Acestui sumator îi corespunde următoarea entitate:

[1] Sunt acceptate numai instrucţiunile concurente al căror proces echivalent este pasiv, adică: instrucţiunile concurente **assert**, anumite apeluri de proceduri şi anumite procese (aceste instrucţiuni sunt prezentate ulterior).

```
entity SUMATOR_COMPLET is
      port(A, B, CIN: in BIT;
           S, COUT: out BIT);
end SUMATOR_COMPLET;
```

Porturile (punctele de intrare / ieşire) A, B, CIN, S şi COUT se numesc *porturi formale*. Asta înseamnă că ele sunt relative la modelul SUMATOR_COMPLET. În momentul referirii acestui sumator ca şi componentă (**component**) sau în momentul utilizării sale (instanţiere), aceste nume vor fi asociate unor valori sau semnale numite *porturi actuale*.

În esenţă, *porturile sunt semnale* şi posedă un *tip* care este indicat în această specificaţie. Oarecum similar parametrilor procedurilor, fiecărui port îi va fi atribuit un *mod*. Acest mod indică un port de intrare (**in**), un port de ieşire (**out**) sau un port bidirecţional (**inout**). Mai există încă două moduri (**buffer** şi **linkage**) care sunt mai deosebite şi vor fi descrise ulterior.

Un port formal poate fi:

- conectat la un alt port;
- conectat la un semnal local;
- lăsat neconectat (se foloseşte cuvântul cheie **open**).

Conexiunile pot fi verificate în foarte multe feluri datorită *modurilor*. Astfel, un port formal în mod **in** al unui model nu poate fi conectat la un port de mod **out** al uneia dintre componentele sale: intrarea modelului ar fi atunci o ieşire a componentei înglobate de el. Figura 2.2 indică toate conexiunile autorizate.

Modul componentei / *Portul entităţii*	Modul **in**	Modul **out**	Modul **inout**	Modul **buffer**	Modul **linkage**
Port de mod **in**	**DA**	NU	NU	NU	**DA**
Port de mod **out**	NU	**DA**	NU	NU	**DA**
Port de mod **inout**	**DA**	**DA**	**DA**	NU	**DA**
Port de mod **buffer**	**DA**	NU	NU	**DA**	**DA**
Port de mod **linkage**	NU	NU	NU	NU	**DA**
Semnal local	**DA**	**DA**	**DA**	**DA**	**DA**
Port **open** (neconectat)	NU	**DA**	**DA**	**DA**	**DA**

Figura 2.2 *Conectarea porturilor formale*

2.4.2 Arhitectura

Entitatea defineşte interfaţa sau vederea externă a unei cutii negre; arhitectura descrie ceea ce se petrece (comportamentul) sau ce se găseşte în interiorul său (structura).

Această descriere poate fi *structurală* (cutia neagră este o interconectare a altor cutii negre), *funcţională*, „*flux de date*" (se dă algoritmul pe care-l implementează cutia neagră) sau *combinată*.

Descrierile se completează adeseori una pe cealaltă, iar la simulare se poate alege descrierea cea mai adecvată proiectului în curs. De aceea, aceeaşi entitate poate avea mai multe arhitecturi asociate.

În VHDL entitatea şi arhitectura trebuie să se găsească în aceeaşi bibliotecă.

Sintaxa unei arhitecturi este următoarea:

```
architecture numele_arhitecturii of numele_entităţii is
      ...
      ... Zona de declaraţii
      ...
begin
      ...
      ... Instrucţiuni concurente
      ...
end {numele_arhitecturii};
```

Întrucât arhitectura face parte din domeniul concurent, la acest prim nivel nu se pot face declaraţii de variabile (cu excepţia variabilelor partajate), ci numai de semnale. Tot astfel, funcţionalitatea acestei arhitecturi este descrisă de către un ansamblu de instrucţiuni concurente executate în manieră asincronă.

Iată câteva arhitecturi posibile ale entităţii a cărei specificaţie (SUMATOR_COMPLET) a fost prezentată mai sus.

a) *Descrierea structurală*

```
architecture DESCRIERE_STRUCTURALĂ of SUMATOR_COMPLET is
      component SEMI_SUMATOR
            port(A, B: in BIT;
                  COUT, S: out BIT);
      end component;

      component POARTA_SAU
            port(A, B: in BIT;
                  S: out BIT);
      end component;

      signal N1, N2, N3: BIT;
begin
--Interconectarea celor trei componente
C1: SEMI_SUMATOR port map (A, B, N1, N2);
C2: SEMI_SUMATOR port map (N2, CIN, N3, S);
C3: POARTA_SAU port map (N1, N3, COUT);
end DESCRIERE_STRUCTURALĂ;
```

Observaţie

Noţiunea de timp nu intervine într-o descriere structurală, care nu reprezintă decât interconectarea unor elemente.

b) *Descrierea „flux de date”*

Sumatorul complet este descris de către următoarele ecuaţii:

```
                  V = A XOR B
                S = V XOR CIN
      COUT = (A and B) OR (V and CIN)
```

```
architecture DESCRIERE_FLUX_DE_DATE of SUMATOR_COMPLET is
      signal INTER: bit;
begin
      INTER <= A xor B after 1 ns;
      S     <= INTER xor CIN after 1 ns;
      COUT  <= (A and B) or (INTER and CIN) after 2 ns;
end DESCRIERE_FLUX_DE_DATE;
```

Observaţie

Conceptul de timp (de temporizare) intervine în cadrul acestui exemplu numai pentru a ilustra posibilitatea temporizării unei asemenea descrieri. O descriere lipsită de temporizări este la fel de valabilă.

c) *Descrierea comportamentală*

Această descriere este bazată pe tabelul de adevăr al sumatorului:

A	B	CIN	S	COUT
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

de unde se poate deduce următoarea descriere:

```
architecture DESCRIERE_COMPORTAMENTALĂ of SUMATOR_COMPLET is
begin
      process
      variable L: integer;
      constant TABEL_S: bit_vector(0 to 3):= "0101";
      constant TABEL_COUT: bit_vector(0 to 3):= "0011";
            begin
            L:= 0;
            if A = '1' then L:= 1;  end if;
            if B = '1' then L:= L + 1;  end if;
            if CIN = '1' then L:= L + 1;  end if;
            --Atribuiri de semnale
            S <= TABEL_S(L) after 10 ns;
            COUT <= TABEL_COUT(L) after 15 ns;
-- Monitorizăm modificările (evenimentele) de pe A, B şi CIN
            wait on A, B, CIN;
            end process;
end DESCRIERE_COMPORTAMENTALĂ;
```

Şi în acest caz este posibilă apariţia explicită a timpului.

d) *Descrierea combinată*

Am ales o descriere structurală pentru semi-sumator şi o descriere comportamentală a porţii SAU.

```
architecture DESCRIERE_COMBINATĂ of SUMATOR_COMPLET is
      component SEMI_SUMATOR
            port(A, B: in bit;
                  COUT, S: out bit);
      end component;
      signal N1, N2, N3: bit;
begin
 C1: SEMI_SUMATOR port map (A, B, N1, N2);
 C2: SEMI_SUMATOR port map (N2, CIN, N3, S);
 C3: COUT <= N1 or N3; -- aici ar putea interveni o
                       -- temporizare
end DESCRIERE_COMBINATĂ;
```

2.4.3 Configuraţia

Pentru a simula o instanţă a unei componente este necesară cunoaşterea perechii entitate / arhitectură pe care aceasta o „utilizează”. Această informaţie poate fi dată direct sau poate proveni de la *configuraţie*.

Începând cu versiunea '93 a VHDL există posibilitatea asocierii directe şi într-o singură linie a unei perechi entitate / arhitectură cu o instanţă a unei componente. Aceasta se realizează cu ajutorul unei instrucţiuni numite „instanţiere directă” (descrisă în lucrarea nr. 9). De fapt, însăşi noţiunea de „componentă” nu mai este necesară. Instrucţiunea este uşor de pus în aplicare, dar suferă de lipsă de forţă şi de flexibilitate: numele entităţii şi al arhitecturii sunt fixate într-un loc precis al codului sursă. Configuraţia – deşi este un concept mai greu de asimilat - oferă mult mai multă flexibilitate.

Conceptele de „componentă” şi de „configuraţie” sunt primordiale în VHDL. De fapt, există două tipuri de configuraţii posibile:

a) De fiecare dată când utilizăm o componentă într-o descriere VHDL, există posibilitatea de a o configura imediat, ceea ce înseamnă să specificăm modelul al cărei instanţă este, parametrii săi (dacă este generic) şi corespondenţa dintre porturile sale (numite *porturi formale*) şi porturile actuale.

b) Cu toate acestea, în unele cazuri poate părea mai judicios să amânăm această configuraţie pentru a o plasa în unitatea de proiectare cu acelaşi nume (cu ajutorul cuvântului cheie **configuration**).

Forţa acestei unităţi de proiectare creşte atunci când este separată de restul descrierii. Astfel, fără a modifica (deci fără a mai recompila) alte unităţi de proiectare, va fi posibil, de exemplu, să schimbăm nivelul descrierii unei componente (configurând o nouă arhitectură, poate chiar o altă entitate).

Aspectele prezentate anterior în cazul unei componente sunt la fel de adevărate şi în cazul unui *bloc*, element ierarhic intern unei arhitecturi care va fi prezentat ulterior (lucrarea nr. 9).

Sintaxa unei configuraţii este următoarea:

```
configuration numele_configuraţiei of numele_entităţii is
{Zona declarativă (numai clauza use şi specificarea atributelor)}
{Zona rezervată configuraţiei}
end {numele_configuraţiei};
```

Zona declarativă este extrem de redusă: în ea nu sunt autorizate decât clauza **use** şi specificaţiile de atribute.

Zona rezervată configuraţiei propriu-zise descrie modul în care vor fi realizate fizic componentele din interiorul blocurilor[2].

Aceste aspecte se traduc printr-o serie de clauze **for ... end for**; imbricate care desemnează blocul în care se găseşte componenta sau componentele de configurat. În interiorul imbricării de clauze **for**, configuraţia fiecăreia dintre componente este indicată precis şi are o sintaxă de tipul:

```
for eticheta_instanţei_componentei: nume_componentă use
entity
numele_entităţii(numele_arhitecturii) {parametri generici}
      {corespondenţa porturi formale / porturi actuale};
end for;
```

Configuraţia poate fi ea însăşi ierarhică. În acest caz, în loc să facă referire la o pereche entitate / arhitectură ca mai sus, ea poate pur şi simplu să facă referire la o altă configuraţie de nivel inferior:

---

[2] Blocul constituie elementul fundamental al ierarhiei în VHDL. O arhitectură este un bloc şi putem avea blocuri (eventual imbricate) în interiorul unei arhitecturi.

```
for eticheta_instanţei_componentei: numele_componentei
      use configuration numele_configuraţiei
end for;
```

Reluând exemplul precedent al sumatorului complet şi în special arhitectura structurală (DESCRIERE_STRUCTURALĂ) este posibil să configurăm componentele semi-sumator prin entitatea HALF_ADDER de arhitectură CN78975S (corespunzătoare unui circuit integrat disponibil pe piaţă) şi poarta logică SAU prin entitatea OR_GATE de arhitectură SN74LS32, scriind:

```
use WORK.SUMATOR_COMPLET; -- Clauza permite „vederea” entităţii
                          -- SUMATOR_COMPLET din biblioteca de
                          -- lucru WORK
configuration TEST1 of SUMATOR_COMPLET is
for DESCRIERE_STRUCTURALĂ
      for  C1, C2: SEMI_SUMATOR
            use entity WORK.HALF_ADDER(CN78975S);
      end for;
      for C3: POARTA_SAU
            use entity WORK.OR_GATE(SN74LS32);
      end for;
end for;
end TEST1;
```

În VHDL este posibilă configurarea incrementală a unei descrieri. De exemplu, se pot configura mai întâi valorile parametrilor generici, apoi se poate reveni, în altă parte a codului, pentru a se modifica aceste valori. Porturile neconectate pot de asemenea să fie configurate la un nivel mai înalt al ierarhiei.

Configuraţia este un concept VHDL puternic, dar care pune probleme începătorilor în ceea ce priveşte aplicarea sa concretă.

Apelarea unei biblioteci se efectuează prin intermediul unei clauze **use**..., eventual precedată de o clauză **library**..., dacă numele bibliotecii nu este WORK sau STD. De exemplu:

```
--apelarea tuturor perechilor entitate / arhitectură
library MY_PROJECT_SIM;
use MY_PROJECT_SIM.all;
```

Pentru ca această apelare să fie luată în considerare, trebuie ca ea să preceadă contextul pe care-l are în vedere.

Există două situaţii posibile:

a) Întreaga configuraţie are nevoie de biblioteci care trebuie referite.

În acest caz, cel mai simplu este să scriem clauzele **library** şi **use** la început, adică înaintea cuvântului cheie **configuration**. De exemplu:

```
library MY_PROJECT_SIM;
use MY_PROJECT_SIM.all;
configuration cfg_test_PROIECT of test_PROIECT is...
```

b) Modelul are foarte multe nivele ierarhice şi fiecare dintre sub-blocuri nu apelează în mod necesar aceleaşi biblioteci, cu riscurile de ambiguitate pe care le implică această abordare. Sunt posibile două soluţii:

b1) Configuraţia este „spartă" în atâtea sub-configuraţii câte sunt necesare. Fiecare dintre ele face atunci referire, fără ambiguităţi, la una sau mai multe biblioteci şi astfel am revenit întrucâtva la cazul anterior.

b2) Bibliotecile sunt referite în momentul când este nevoie de ele în cadrul modelului. Concret, aceste referiri la biblioteci sunt plasate chiar înainte de stipularea numelui arhitecturii care conţine instanţele de configurat. De exemplu:

```
library MY_PROJECT_SIM; -- Referinţe globale
use MY_PROJECT_SIM.all;

configuration CFG_TEST_PROIECT of TEST_PROIECT is
--numele configuraţiei şi al entităţii
       for A_TEST   -- numele arhitecturii folosite
        for all: B1 use entity BIB1.OBIECT1(ARC_RTL);
        end for;
         for all: B2 use entity BIB2.OBIECT1(ARC_RTL);
              use MY_PROIECT_SIM.all;
              for ARC_RTL
              end for;
         end for;
       end for;
end CFG_TEST_PROIECT;
```

Observaţie

Soluţia cea mai uşor aplicabilă o constituie trecerea prin sub-configuraţii. Într-adevăr, după fiecare analiză VHDL a unei configuraţii a unui sub-bloc, este suficient să se lanseze simularea pentru a verifica dacă instrumentul reuşeşte să construiască reţeaua completă de componente şi de cupluri entităţi / arhitecturi la care trebuie să le lege (aşa-zisa „netlist").

Forma minimală a configuraţiei este următoarea:

```
{scrierea referinţelor, clauzele library şi use}
configuration CFG_D of D is
       for A_D
       end for;
end CFG_D;
```

Regulile necesare utilizării acestei forme simplificate se pot rezuma astfel:

a) Trebuie definite *profiluri* identice pentru entităţile şi componentele VHDL. Astfel se poate utiliza mecanismul configuraţiei implicite.

Prin profil se înţelege:

- acelaşi nume (componentă şi entitate);
- aceeaşi interfaţă la porturi;
- aceeaşi interfaţă în ceea ce priveşte parametrii generici; aceştia pot totuşi să fie omişi din vederea componentei dacă au o valoare implicită definită în entitate.

b) Trebuie definită o listă de biblioteci sau de elemente de referinţă astfel încât să se evite orice ambiguitate: o aceeaşi entitate nu trebuie să existe în două biblioteci diferite.

c) Este necesar să definim o sub-configuraţie (având o formă simplificată) atunci când:

- este posibilă apariţia unei ambiguităţi;
- o entitate posedă mai multe arhitecturi operaţionale;
- entitatea reprezintă un bloc compact (RAM, ROM etc.).

Această nouă configuraţie va fi bineînţeles utilizată într-o configuraţie de nivel superior.

Dacă proiectantul respectă regulile enunţate mai sus, el nu va avea nevoie să explicitezе, nici măcar parţial, structura ierarhică a blocului în

decursul scrierii configuraţiei. Va fi necesară doar enunţarea listei de referinţe care vor fi utile pentru ca asocierea „instanţei de componentă” cu „perechea entitate / arhitectură” să se efectueze corect.

Referinţele pot fi de mai multe tipuri:

a) Referirea unei biblioteci complete în sens VHDL.

Aceste biblioteci pot reprezenta obiecte de naturi foarte diferite:

- un catalog de celule standard;
- lista (totală sau parţială) a vederilor ierarhice ale circuitului.

b) Referirea unei configuraţii deja definite. Avantajul acestei abordări constă în faptul că este ierarhică.

c) Referirea unei entităţi deja definite. Această abordare prezintă interes în cazul în care proiectantul nu doreşte să facă referire la totalitatea unei biblioteci (pentru a evita problemele de ambiguitate), ci numai la una sau mai multe entităţi.

Observaţie

Dacă această entitate, referită în mod explicit, are mai multe arhitecturi, nu este posibil s-o alegem direct (compilatorul va selecta arhitectura cea mai recent compilată). O soluţie constă în definirea unei configuraţii specifice pentru o anumită pereche entitate / arhitectură şi în alegerea perechii corecte.

2.4.4 Specificarea pachetelor

Specificaţia unui pachet (numită şi partea declarativă sau vederea externă a pachetului) prezintă tot ce exportă pachetul: în primul rând obiecte (semnale, constante, fişiere sau variabile partajate), tipuri şi sub-tipuri, precum şi sub-programe. Se pot exporta şi tipuri fişier, declaraţii de componente şi de alias-uri, specificaţii sau declaraţii de atribute, specificaţii de conectare şi clauze **use**. Toate aceste obiecte îi vor fi accesibile oricărui proiectant care apelează acest pachet printr-o clauză **use**.

Specificaţia următorului pachet oferă o imagine asupra acestor posibilităţi:

```
package P1_PKG is
```

```
--Definirea unui tip enumerat
type CIT_SCR is (SCRIERE,  CITIRE);
--Definirea unei constante
constant DIM_MAX: INTEGER:=200;
--Definirea de sub-tipuri
subtype DIM is INTEGER range 0 to DIM_MAX;
subtype CIT_SCR_BIT is BIT;
--Definirea de constante cu iniţializare ulterioară
constant SCRIERE_BIT: CIT_SCR_BIT;
constant CITIRE_BIT: CIT_SCR_BIT;
--Declararea unui semnal
signal COMANDĂ_MEMORIE: CIT_SCR;
--Declararea unei funcţii
function CDĂ_SPRE_BIT (C: in CIT_SCR) return CIT_SCR_BIT;
--Declararea unei proceduri
procedure CDĂ_SPRE_BIT (C: in CIT_SCR; B: out CIT_SCR_BIT);
end P1_PKG;
```

Un pachet poate avea un corp asociat, lucru care este chiar obligatoriu atunci când se declară sub-programe sau constante cu iniţializare ulterioară. În acest caz, corpul este unic şi conţine algoritmica (strict secvenţială) necesară realizării sub-programelor, precum şi eventuale declaraţii de uz intern ale pachetului.

2.4.5 Corpul unui pachet

Corpul unui pachet conţine algoritmii sub-programelor exportate de către pachet, a căror declarare a fost făcută în specificaţia de pachet corespunzătoare.

În corpul pachetului sunt declarate şi tipurile, sub-tipurile, constantele, fişierele, alias-urile sau sub-programele (declaraţie şi corp) utilizate local. Aceste declaraţii nu sunt văzute din exteriorul pachetului, nici măcar din specificaţia sa. În schimb, toate declaraţiile efectuate în specificaţie sunt cunoscute în corpul pachetului.

Observaţie

Este interzisă declararea semnalelor: corpul unui pachet face parte din domeniul secvenţial.

Vom considera un exemplu extrem de simplu: următorul pachet va trebui să exporte două funcţii care returnează, respectiv, cel mai mic şi cel mai mare dintre două numere întregi.

```
package SIMPLU is
      function MIN(A,B: INTEGER) return INTEGER;
      function MAX(A,B: INTEGER) return INTEGER;
end SIMPLU;

package body SIMPLU is
      function MIN(A,B: INTEGER) return INTEGER is
      begin
            if A < B then return A;
            else return B;
            end if;
      end MIN;
      function MAX(A,B: INTEGER) return INTEGER is
      begin
            if A > B then return A;
            else return B;
            end if;
      end MAX;
end SIMPLU;
```

În acelaşi mod, corpul corespunzător pachetului P1_PKG, a cărui specificaţie a fost prezentată mai sus, dă valorile constantelor cu iniţializare ulterioară CITIRE_BIT şi SCRIERE_BIT, apoi descrie algoritmii funcţiei şi procedurii CDĂ_SPRE_BIT.

```
Package body P1_PKG is

--Definirea valorii constantelor cu iniţializare ulterioară
constant CITIRE_BIT: CIT_SCR_BIT:= '0';
constant SCRIERE_BIT: CIT_SCR_BIT:= '0';

--Definirea funcţiei
function CDĂ_SPRE_BIT (C: in CIT_SCR) return CIT_SCR_BIT is
begin
      case C is
            when CITIRE => return CITIRE_BIT;
            when SCRIERE => return SCRIERE_BIT;
      end case;
end;
```

```
--Definirea procedurii
procedure CDĂ_SPRE_BIT(C: in CIT_SCR;B: out CIT_SCR_BIT) is
begin
      case C is
            when CITIRE => B:= CITIRE_BIT;
```

```
            when SCRIERE => B:= SCRIERE_BIT;
      end case;
end;
end P1_PKG;
```

3. Desfăşurarea lucrării

3.1 Se vor crea pachetele P1_PKG şi SIMPLU prezentate în cuprinsul lucrării şi se va testa corectitudinea lor.

3.2 Se va crea o entitate numită SUMATOR_1 de numere pe 1 bit şi se vor specifica trei arhitecturi diferite (structurală, comportamentală şi „flux de date") ale acesteia. Se vor simula toate arhitecturile.

3.3 Se va crea o entitate numită SUMATOR_2 de numere pe 2 biţi în cadrul căreia se vor utiliza instanţe ale entităţii SUMATOR _1, folosind mecanismul specific configuraţiilor.

3.4 Se va crea o entitate numită COMPARATOR_2 de numere pe 2 biţi şi se vor specifica trei arhitecturi diferite (structurală, comportamentală şi „flux de date") ale acesteia. Se vor simula toate arhitecturile.

3.5 Se va crea o entitate numită COMPARATOR_4 de numere pe 4 biţi în cadrul căreia se vor utiliza instanţe ale entităţii COMPARATOR_2, folosind mecanismul specific configuraţiilor.

3.6 Se va implementa un circuit de comandă a unui decodificator BCD-7 segmente.