

LUCRAREA NR. 2

UNITĂȚI FUNDAMENTALE DE PROIECTARE

1. Scopul lucrării

Se prezintă unitățile fundamentale de proiectare în VHDL: *perechea entitate / arhitectură*, *configurațiile*, *specificațiile de pachete* și *corpurile pachetelor*. Se studiază sintaxa declarațiilor respective, precum și cadrul lor de utilizare. Se detaliază avantajele și dezavantajele specifice operării cu unitățile respective.

2. Considerații teoretice

2.1 Noțiuni introductive

VHDL este un limbaj bazat pe biblioteci, ceea ce înseamnă că după compilarea unui fișier 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 părți ale programelor care pot fi compilate separat: acestea sunt *unitățile de proiectare*. Fiecare compilare reușită a unui fișier (realizată de către sistemul de operare sau de mediul de dezvoltare VHDL) va fi susceptibilă să producă în biblioteca curentă una sau mai multe unități de proiectare. Mai multe unități de proiectare pot fi scrise în același fișier, dar o unitate de proiectare nu poate fi divizată în mai multe fișiere.

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

În cazul prezenței mai multor unități de proiectare în același fișier, există posibilitatea decelării unei erori în cadrul uneia dintre ele; în consecință, analiza efectuată de compilator nu va fi dusă până la capăt. În acest caz, compilatorul va analiza și va compila unitățile de proiectare din fișier care se află *înaintea* unității incriminate. În anumite cazuri, el va

încerca și chiar va reuși să compileze unitățile situate *după* unitatea de proiectare respectivă.

Deci noțiunea de fișier se pierde complet după compilare: nu se mai vorbește despre fișiere, ci numai de unități de proiectare (se va vorbi despre fișiere în VHDL la nivelul tipurilor de date - acest concept este cu totul diferit de cel discutat aici).

2.2 Unități de proiectare primare și secundare

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

De exemplu, *specificația 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 (algoritmii) implementării. Specificația unui pachet este o unitate de proiectare primară.

Partea de implementare a pachetului (*corpul pachetului*) conține algoritmi utilizați. Această informație 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 reținută, între *specificația unui pachet* și *entitate* - ca vederi externe ale unor cutii negre (hardware sau software). De altfel, se poate deduce imediat și analogia dintre *corpul pachetului* și *arhitectură*, aceste două unități de proiectare servind la descrierea realizării cutiei negre (hardware sau software).

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

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

- Entitatea;
- Arhitectura;
- Configurația;
- Specificația pachetului;
- Corpul pachetului.

2.3 Biblioteci

În VHDL putem avea un număr mare de biblioteci, fiecare dintre ele conținând mai multe unități 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 unitățile de proiectare compilate cu succes. Unitățile de proiectare primare aparținând 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 precedă unitatea de proiectare care o apelează. În continuare, referirea unității se poate face printr-o clauză **use** într-o zonă de declarații oarecare (fără a uita sufixul **.all** pentru pachete, dacă dorim să accesăm totalitatea „exporturilor” unui pachet). Așadar, se scrie:

```
library biblioteca_în_care_este_păstrată_unitatea_primară;  
...  
use numele_entității;    -- dacă unitatea primară e o  
                        -- specificație de entitate  
  
use numele_configurației;    -- dacă este o configurație  
use numele_pachetului.all;    -- dacă este un pachet
```

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

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

2.4 Entitate, arhitectură și configurație

Orice sistem fizic comunică cu exteriorul prin intermediul unor semnale care sunt grupate în așa-zisa sa *interfață* cu exteriorul. Semnalele primite de sistemul respectiv sunt orientate dinspre exterior către sistem și sunt numite *semnale de intrare* sau - mai simplu - *intrări* sau *stimuli*.

Reacționând la stimulii primiți, sistemul va genera, la rândul său, *semnale de ieșire* sau *răspunsuri*, transmise mediului său exterior.

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

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

2.4.1 Entitatea

Entitatea descrie interfața unică dintre arhitecturi și mediul exterior, iar arhitectura descrie funcționalitatea entității. Ansamblul poate fi văzut și utilizat, la un nivel superior, ca o componentă.

În funcție de gradul de finețe sau, pur și simplu, de nivelul de descriere (structural, comportamental, flux de date sau hibrid) care a fost ales pentru descrierea funcționalității, vom putea avea mai multe arhitecturi diferite pentru aceeași entitate.

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

Dimpotrivă, informațiile cuprinse în entitate sunt cunoscute de către diversele sale arhitecturi. Toate aceste informații sunt opționale. Ele sunt de trei feluri:

- declarații de parametri generici;
- declarații de porturi;
- instrucțiuni.

Sintaxa unei entități este următoarea:

```
entity numele_entității is
  {generic (listă de parametri generici)}
  {port (listă de porturi)}
  {begin
    listă de instrucțiuni concurente}
end {numele_entității};
```

Acoladele delimitează informațiile opționale.

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

```
numele_entității (numele_arhitecturii)
```

Numele entității, 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 entității respective. Regulile pe care trebuie să le respecte orice identificator au fost prezentate în lucrarea nr. 1.

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

Lista porturilor oferă o mulțime de informații relative la semnalele care interfațează entitatea. Se indică:

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

Lista de instrucțiuni concurente permite scrierea o singură dată a prelucrărilor care sunt globale pentru toate arhitecturile aferente. În practică, restricțiile impuse¹ limitează această porțiune de cod la un rol de control al intrărilor entității (sau al parametrilor, în cazul entităților generice).

Să luăm exemplul următorului sumator complet:

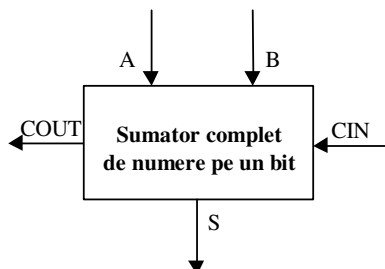


Figura 2.1 Vederea externă a unui sumator complet de numere pe un bit

¹ Sunt acceptate numai instrucțiunile concurente al căror proces echivalent este pasiv, adică: instrucțiunile concurente **assert**, anumite apeluri de proceduri și anumite procese (aceste instrucțiuni sunt prezentate ulterior).

Acestui sumator îi corespunde următoarea entitate:

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

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

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

Un port formal poate fi:

- conectat la un alt port;
- conectat la un semnal local;
- lăsat neconectat (se folosește 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 ieșire a componentei înglobate de el. Figura 2.2 indică toate conexiunile autorizate.

<i>Modul componentei</i> <i>Portul entității</i>	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 definește interfața sau vederea externă a unei cutii negre; arhitectura descrie ceea ce se petrece (comportamentul) sau ce se găsește în interiorul său (structura).

Această descriere poate fi *structurală* (cutia neagră este o interconectare a altor cutii negre), *funcțională*, „*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, aceeași entitate poate avea mai multe arhitecturi asociate.

În VHDL entitatea și arhitectura trebuie să se găsească în aceeași bibliotecă.

Sintaxa unei arhitecturi este următoarea:

```
architecture numele_arhitecturii of numele_entității is
    ...
    ... Zona de declarații
    ...
begin
    ...
    ... Instrucțiuni concurente
    ...
end {numele_arhitecturii};
```

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

Iată câteva arhitecturi posibile ale entității a cărei specificație (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;
```

Observație

Noțiunea 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 ecuații:

$$\begin{aligned} V &= A \text{ XOR } B \\ S &= V \text{ XOR } CIN \\ COUT &= (A \text{ and } B) \text{ OR } (V \text{ and } CIN) \end{aligned}$$

```
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;
```


Observație

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_COMPORAMENTALĂ 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;
    --Atribuirii de semnale
    S <= TABEL_S(L) after 10 ns;
    COUT <= TABEL_COUT(L) after 15 ns;
    -- Monitorizăm modificările (evenimentele) de pe A, B și CIN
    wait on A, B, CIN;
  end process;
end DESCRIERE_COMPORAMENTALĂ;
```

Și în acest caz este posibilă apariția explicită a timpului.

d) *Descrierea combinată*

Am ales o descriere structurală pentru semi-sumator și o descriere comportamentală a porții 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 *Configurația*

Pentru a simula o instanță a unei componente este necesară cunoașterea perechii entitate / arhitectură pe care aceasta o „utilizează”. Această informație poate fi dată direct sau poate proveni de la *configurație*.

Începând cu versiunea '93 a VHDL există posibilitatea asocierii directe și într-o singură linie a unei perechi entitate / arhitectură cu o instanță a unei componente. Aceasta se realizează cu ajutorul unei instrucțiuni numite „instanțiere directă” (descrișă în lucrarea nr. 9). De fapt, însăși noțiunea de „componentă” nu mai este necesară. Instrucțiunea este ușor de pus în aplicare, dar suferă de lipsă de forță și de flexibilitate: numele entității și al arhitecturii sunt fixate într-un loc precis al codului sursă. Configurația – deși este un concept mai greu de asimilat - oferă mult mai multă flexibilitate.

Conceptele de „componentă” și de „configurație” sunt primordiale în VHDL. De fapt, există două tipuri de configurații 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 instanță este, parametrii săi (dacă este generic) și corespondența dintre porturile sale (numite *porturi formale*) și porturile actuale.

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

Forța acestei unități de proiectare crește atunci când este separată de restul descrierii. Astfel, fără a modifica (deci fără a mai recompila) alte unități 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 și în cazul unui *bloc*, element ierarhic intern unei arhitecturi care va fi prezentat ulterior (lucrarea nr. 9).

Sintaxa unei configurații este următoarea:

```
configuration numele_configurației of numele_entității is
{Zona declarativă (numai clauza use și specificarea
atributelor)}
{Zona rezervată configurației}
end {numele_configurației};
```

Zona declarativă este extrem de redusă: în ea nu sunt autorizate decât clauza **use** și specificațiile de atribut.

Zona rezervată configurației propriu-zise descrie modul în care vor fi realizate fizic componentele din interiorul blocurilor².

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

```
for eticheta_instanței_componentei: nume_componentă use
entity
numele_entității(numele_arhitecturii) {parametri generici}
{corespondența porturi formale / porturi actuale};
end for;
```

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

Configurația poate fi ea însăși ierarhică. În acest caz, în loc să facă referire la o pereche entitate / arhitectură ca mai sus, ea poate pur și simplu să facă referire la o altă configurație de nivel inferior:

```
for eticheta_instanței_componentei: numele_componentei
    use configuration numele_configurației
end for;
```

Reluând exemplul precedent al sumatorului complet și 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 piață) și poarta logică SAU prin entitatea OR_GATE de arhitectură SN74LS32, scriind:

```
use WORK.SUMATOR_COMPLET; -- Clauza permite „vederea” entității
                           -- 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.

Configurația este un concept VHDL puternic, dar care pune probleme începătorilor în ceea ce privește 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 situații posibile:

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

În acest caz, cel mai simplu este să scriem clauzele **library** și **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 și fiecare dintre sub-blocuri nu apelează în mod necesar aceleași biblioteci, cu riscurile de ambiguitate pe care le implică această abordare. Sunt posibile două soluții:

b1) Configurația este „spartă” în atâtea sub-configurații câte sunt necesare. Fiecare dintre ele face atunci referire, fără ambiguități, la una sau mai multe biblioteci și 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 conține instanțele de configurat. De exemplu:

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

configuration CFG_TEST_PROIECT of TEST_PROIECT is
--numele configurației și al entității
  for A_TEST -- numele arhitecturii folosite
    for all: B1 use entity BIB1.OBJECT1 (ARC_RTL);
    end for;
    for all: B2 use entity BIB2.OBJECT1 (ARC_RTL);
      use MY_PROJECT_SIM.all;
      for ARC_RTL
        end for;
    end for;
  end for;
end CFG_TEST_PROIECT;
```

Observație

Soluția cea mai ușor aplicabilă o constituie trecerea prin sub-configurații. Într-adevăr, după fiecare analiză VHDL a unei configurații a unui sub-bloc, este suficient să se lanseze simularea pentru a verifica dacă instrumentul reușește să construiască rețeaua completă de componente și de cupluri entități / arhitecturi la care trebuie să le lege (așa-zisa „netlist”).

Forma minimală a configurației este următoarea:

```
{scrierea referințelor, clauzele library și 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 entitățile și componentele VHDL. Astfel se poate utiliza mecanismul configurației implicite.

Prin profil se înțelege:

- același nume (componentă și entitate);
- aceeași interfață la porturi;
- aceeași interfață în ceea ce privește parametrii generici; aceștia pot totuși să fie omiși din vederea componentei dacă au o valoare implicită definită în entitate.

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

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

- este posibilă apariția unei ambiguități;
- o entitate posedă mai multe arhitecturi operaționale;
- entitatea reprezintă un bloc compact (RAM, ROM etc.).

Această nouă configurație va fi bineînțeles utilizată într-o configurație de nivel superior.

Dacă proiectantul respectă regulile enunțate mai sus, el nu va avea nevoie să expliciteze, nici măcar parțial, structura ierarhică a blocului în

decursul scrierii configurației. Va fi necesară doar enunțarea listei de referințe care vor fi utile pentru ca asocierea „istanței de componentă” cu „perechea entitate / arhitectură” să se efectueze corect.

Referințele 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 parțială) a vederilor ierarhice ale circuitului.
- b) Referirea unei configurații deja definite. Avantajul acestei abordări constă în faptul că este ierarhică.
- c) Referirea unei entități deja definite. Această abordare prezintă interes în cazul în care proiectantul nu dorește să facă referire la totalitatea unei biblioteci (pentru a evita problemele de ambiguitate), ci numai la una sau mai multe entități.

Observație

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

2.4.4 Specificarea pachetelor

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

Specificarea următorului pachet oferă o imagine asupra acestor posibilități:

```
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 inițializare ulterioară
constant SCRIERE_BIT: CIT_SCR_BIT;
constant CITIRE_BIT: CIT_SCR_BIT;
--Declararea unui semnal
signal COMANDĂ_MEMORIE: CIT_SCR;
--Declararea unei funcții
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 inițializare ulterioară. În acest caz, corpul este unic și conține algoritmică (strict secvențială) necesară realizării sub-programelor, precum și eventuale declarații de uz intern ale pachetului.

2.4.5 Corpul unui pachet

Corpul unui pachet conține algoritmică sub-programelor exportate de către pachet, a căror declarare a fost făcută în specificația de pachet corespunzătoare.

În corpul pachetului sunt declarate și tipurile, sub-tipurile, constantele, fișierele, alias-urile sau sub-programele (declarație și corp) utilizate local. Aceste declarații nu sunt văzute din exteriorul pachetului, nici măcar din specificația sa. În schimb, toate declarațiile efectuate în specificație sunt cunoscute în corpul pachetului.

Observație

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

Vom considera un exemplu extrem de simplu: următorul pachet va trebui să exporte două funcții care returnează, respectiv, cel mai mic și 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 același mod, corpul corespunzător pachetului P1_PKG, a cărei specificație a fost prezentată mai sus, dă valorile constantelor cu inițializare ulterioară CITIRE_BIT și SCRIERE_BIT, apoi descrie algoritmi funcției și procedurii CDĂ_SPRE_BIT.

```
Package body P1_PKG is

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

--Definirea funcției
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. Desfășurarea lucrării

- 3.1 Se vor crea pachetele P1_PKG și SIMPLU prezentate în cuprinsul lucrării și se va testa corectitudinea lor.
- 3.2 Se va crea o entitate numită SUMATOR_1 de numere pe 1 bit și se vor specifica trei arhitecturi diferite (structurală, comportamentală și „flux de date”) ale acesteia. Se vor simula toate arhitecturile.
- 3.3 Se va crea o entitate numită SUMATOR_2 de numere pe 2 biți în cadrul căreia se vor utiliza instanțe ale entității SUMATOR _1, folosind mecanismul specific configurațiilor.
- 3.4 Se va crea o entitate numită COMPARATOR_2 de numere pe 2 biți și se vor specifica trei arhitecturi diferite (structurală, comportamentală și „flux de date”) ale acesteia. Se vor simula toate arhitecturile.
- 3.5 Se va crea o entitate numită COMPARATOR_4 de numere pe 4 biți în cadrul căreia se vor utiliza instanțe ale entității COMPARATOR_2, folosind mecanismul specific configurațiilor.
- 3.6 Se va implementa un circuit de comandă a unui decodificator BCD-7 segmente.