LUCRAREA NR. 9
INSTRUCŢIUNI CONCURENTE

1. Scopul lucrării

Informaţiile din această lucrare vin în completarea celor din lucrarea precedentă. Sunt prezentate pe larg instrucţiunile concurente existente în VHDL, împreună cu aplicaţiile lor principale şi cu exemple de utilizare bogat documentate şi comentate. Vor fi analizate: instrucţiunea **block**, apelul concurent de procedură, instrucţiunea concurentă **assert**, instrucţiunea concurentă de asignare de valori semnalelor, instrucţiunea de instanţiere a unei componente şi instrucţiunea **generate**.

2. Consideraţii teoretice

2.1 Descrierea concurentă

Există o diferenţă importantă între noţiunile de *concurenţă* şi *paralelism*, care se cuvine a fi subliniată.

Instrucţiunile paralele se execută în funcţie de priorităţile atribuite de sistem, fără nici un alt punct de sincronizare în afară de cele programate explicit (prin construcţii specifice de genul: primitivele P şi V, semafoare, monitoare etc.). Acesta este paralelismul întâlnit în anumite limbaje de programare. Fondul acestui mecanism este nedeterminist şi noţiunea de „timp real" este adeseori prezentă „în fundal".

În cazul instrucţiunilor concurente, ordinea de execuţie este oarecare. Aceasta permite punerea în aplicare a unui paralelism real, dar noţiunea de timp real dispare: *simularea nu se efectuează în timp real*.

În concluzie, putem spune că paralelismul limbajelor de programare cunoscute este un paralelism *virtual* în timp *real*, în vreme ce concurenţa din VHDL este un paralelism *real* în timp *virtual* (timpul de simulare).

Descrierea unui sistem hardware se face în mod firesc sub o formă concurentă. Un asemenea sistem se proiectează ca un ansamblu de module,

de flux de date sau de algoritmi (în funcţie de nivelul de descriere ales) care interacţionează, funcţionând în paralel.

Aceste funcţii sunt descrise de instrucţiuni concurente care se execută în mod asincron; în plus, ordinea în care sunt scrise instrucţiunile nu are nici o influenţă asupra ordinii lor de execuţie.

„Puterea" unui limbaj de descriere hardware poate fi evaluată în funcţie de bogăţia setului său de instrucţiuni concurente. În cazul VHDL, acesta este deosebit de complet şi de universal. Instrucţiunile sunt utilizate pentru descrierea comportamentului diferitelor module din componenţa circuitului. Prin urmare, ele se vor regăsi la nivelul unităţilor de proiectare prin intermediul cărora se realizează descrierea: entităţile şi arhitecturile.

Într-o descriere concurentă, variabilele cunoscute din limbajele de programare clasice nu-şi mai au locul, interacţiunea (transportul de informaţie între procese) fiind modelată numai prin intermediul semnalelor.

2.2 Instrucţiunea *block*

Blocul reprezintă o modalitate de reunire a unui set de instrucţiuni concurente pentru a le face să aibă acces la o serie de declaraţii locale, invizibile din orice altă parte a restului descrierii. El permite de asemenea scrierea de instrucţiuni de asignare *gardate* sau ierarhizarea descrierii.

Cele trei funcţii principale ale unui bloc sunt următoarele:

- încapsularea declaraţiilor;
- utilizarea instrucţiunilor de asignare *gardate*;
- suport pentru ierarhizarea proiectului.

Aceste trei funcţii nu se exclud reciproc: un bloc poate foarte bine să-şi asume toate cele trei roluri deodată.

Sintaxa generală a unui bloc este următoarea:

```
etichetă: block {(condiţie_de_gardă)}
      {antet_parametri_generici_şi_porturi}
      ...
      ... Declaraţii locale
      ...
begin
      ...
      ... Instrucţiuni concurente
      ...
end block {etichetă};
```

Condiţia de gardă trebuie să fie o expresie booleană. Ea poate fi adeseori exprimată printr-o frază în limbaj natural, de genul „când semnalul X ia o anumită valoare”.

Antetul opţional indică valorile (pentru parametrii generici) şi semnalele (pentru porturi) pe care blocul le importă din mediul exterior.

2.2.1 Funcţia de încapsulare a declaraţiilor

Fireşte, partea declarativă a unui bloc nu este vizibilă decât în interiorul blocului respectiv. Declaraţiile permise în interiorul unui bloc sunt următoarele:

- declaraţie de sub-program;
- corp de sub-program;
- declaraţie de tip;
- declaraţie de sub-tip;
- declaraţie de constantă;
- declaraţie de semnal;
- declaraţie de fişier;
- declaraţie de alias;
- declaraţie de componentă;
- declaraţie de atribut;
- specificaţie de atribut;
- declaraţie de grup;
- specificaţie de deconectare;
- specificaţie de configuraţie;
- clauza **use**.

De remarcat interdicţia declarării *variabilelor* locale blocului (datorită faptului că natura blocului este concurentă).

2.2.2 Utilizarea instrucţiunilor gardate

Această posibilitate răspunde unei nevoi care apare în mod frecvent la proiectarea circuitelor logice. O mare parte a circuitului este *sincronă*, ceea ce înseamnă că toate asignările de semnale sunt supuse unei condiţii de tact. În general, un circuit sincron este un circuit în cadrul căruia unul sau mai multe evenimente se pot produce în acelaşi timp. De exemplu, pentru un circuit sincronizat pe frontul ascendent al semnalului de tact, fiecare asignare de valoare unui semnal ar trebui să se scrie astfel:

```
A <= expresie when (CLOCK = '1' and not CLOCK'STABLE);
```

Rezultatul acestei asignări poate fi văzut în schema din figura 9.1. Asignarea rezultatului expresiei semnalului A nu se efectuează decât pe frontul ascendent al semnalului de tact CLOCK.

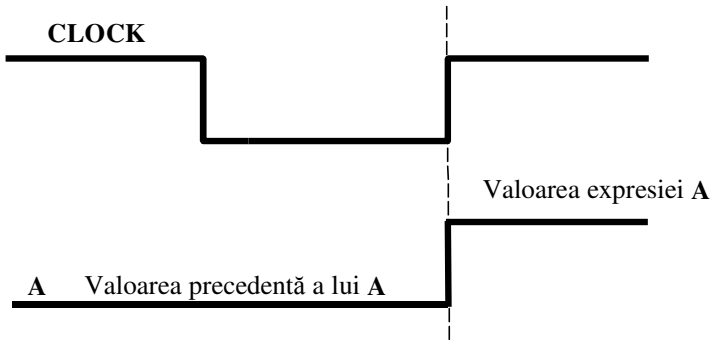

Figura 9.1 *Asignarea semnalului A pe frontul ascendent al semnalului CLOCK*

Observaţie

Atributul STABLE returnează valoarea booleană TRUE atunci când semnalul vizat nu şi-a modificat valoarea în pasul curent al simulării. Condiţia de mai sus înseamnă deci că semnalul de tact are valoarea '1' şi nu este stabilă (tocmai s-a modificat) în pasul curent al simulării. Atributul STABLE este prezentat în detaliu în lucrarea nr. 5.

Ar fi laborios să se scrie în VHDL descrieri care să repete aceeaşi condiţie de un mare număr de ori. *Garda* unui bloc oferă însă posibilitatea „factorizării" acestor condiţii. Aşadar, scrierea blocului gardat (condiţia de gardă este întotdeauna booleană) este echivalentă cu cea a unui bloc fără gardă în care se defineşte un semnal boolean numit GUARD, care ia valoarea expresiei de gardă şi în care instrucţiunile de asignare se efectuează condiţionat.

În exemplul de mai jos se prezintă două blocuri echivalente:

```
BLOC1: block (CLOCK = '1' and not CLOCK'STABLE)
begin
      S1 <= guarded DATA1 after 10 ns;
      S2 <= guarded DATA2 after 15 ns;
end block BLOC1;

BLOC2: block
signal GUARD: BOOLEAN;
begin
      GUARD <= CLOCK = '1' and not CLOCK'STABLE;
      S1 <= DATA1 after 10 ns when GUARD else S1;
      S2 <= DATA2 after 15 ns when GUARD else S2;
end block BLOC2;
```

Semnalul GUARD poate fi declarat explicit şi, cu toate acestea, poate fi recunoscut ca declanşator al asignării gardate (cu ajutorul cuvântului cheie **guarded**). În plus, fie că este sau nu declarat, acest semnal poate face obiectul unei asignări.

2.2.3 Suport pentru proiectarea ierarhică

Blocul este unitatea de bază echivalentă a structurării în VHDL.

În practică, rolul blocului în cadrul structurării este echivalent cu cel al procesului în cazul simulării. Orice altă formă de structurare a proiectului nu este, de fapt, decât un artificiu echivalent cu un bloc, căci în cele din urmă orice construcţie VHDL se reduce la un bloc.

De exemplu, o componentă instanţiată de către o entitate (ansamblul specificaţie de entitate-arhitectură referit într-o configuraţie) este echivalentă cu două blocuri imbricate.

Următorul exemplu este extras din Manualul de referinţă (LRM) al VHDL. Se dă un pachet PAC care exportă o procedură CHECKTIMING a cărei specificaţie poate fi scrisă după cum urmează:

```
package PAC is
      procedure  CHECKTIMING (signal A, B: BIT; T: TIME);
end PAC;
```

Fie următoarea pereche alcătuită din specificaţia de entitate X şi din arhitectura Y:

```
use WORK.PAC.all; -- Referirea pachetului PAC
entity X is
      port(P1, P2: inout BIT);
      constant DELAY: TIME := 1 ms;
begin
      CHECKTIMING(P1, P2, 2*DELAY);
end X;
architecture Y of X is
      signal P3: BIT;
begin
      P3 <= P1 after DELAY;
      P2 <= P3 after DELAY;
      B: block
            ...
         begin
            ...
         end block;
end Y;
```

şi fie o componentă COMP a cărei declaraţie este următoarea:

```
component COMP
      port (A, B: inout BIT);
end component;
```

şi pe care o instanţiem într-o componentă C plecând de la perechea specificaţie de entitate X - arhitectură Y. Această entitate având porturile P1 şi P2 şi nu A şi B, trebuie să specificăm corespondenţele:

```
for C: COMP use entity X(Y) port map (P1 => A, P2 => B);
```

şi în final urmează să specificăm configuraţia componentei C (în care S1 şi S2 sunt două semnale care trebuie conectate la această instanţă):

```
C: COMP port map (A => S1, B => S2);
```

Această instrucţiune va fi echivalentă cu următoarele două blocuri imbricate:

```
C: block
      port(A, B: inout BIT);
      port map (A => S1, B => S2);
begin
      X: block
            port(P1, P2: inout BIT);
            port map (P1 => A, P2 => B);
            constant DELAY: TIME := 1 ms;
            signal P3: BIT;
         begin
            CHECKTIMING(P1, P2, 2*DELAY);
            P3 <= P1 after DELAY;
            P2 <= P3 after DELAY;
            B: block
                  ...
                  begin
                  ...
               end block;
      end block X;
end block C;
```

Partea declarativă a blocului înglobat (X) conţine atât declaraţiile specificaţiei de entitate cât şi cele ale arhitecturii Y. De asemenea, partea rezervată instrucţiunilor blocului înglobat (X) conţine atât instrucţiunile specificaţiei de entitate cât şi cele ale arhitecturii Y.

În practică, această ierarhie tradusă sub formă de blocuri este interesantă mai ales pentru programatorii de compilatoare / simulatoare VHDL. În cazul modelării, ierarhia va fi rareori tradusă prin instrucţiuni **block** (se lasă acest lucru pe seama compilatorului), preferându-se noţiunea de *componentă*.

2.3 Apelul concurent de procedură

Apelul concurent de procedură are aceeaşi sintaxă ca şi apelul secvenţial de procedură (singura diferenţă care apare este posibilitatea specificării unei etichete opţionale).

Întrucât ne aflăm în domeniul concurent, parametrii procedurii nu pot fi decât semnale sau constante (NU şi variabile!). Un apel de procedură se efectuează întotdeauna prin indicarea numelui procedurii urmat, între paranteze, de lista parametrilor de apel (apelul poate fi poziţional, prin nume

sau combinat). Opţional, numele acestei proceduri poate fi precedat de către o etichetă.

Să luăm exemplul unei proceduri care verifică dacă un semnal A respectă un timp de prepoziţionare (*set-up*) TS (valoare declarată ca o constantă) faţă de frontul ascendent al unui alt semnal, B. Semnalele A şi B vor fi transmise în mod **in** (este o verificare – valorile acestor semnale nu vor suferi modificări). Apelul concurent al procedurii se va putea scrie astfel:

```
VERIF1: VERIF_SETUP (A, B, TS);
```

Un apel concurent de procedură va fi tradus în domeniul secvenţial printr-un proces care înglobează apelul secvenţial corespondent, urmat de o instrucţiune **wait** care va monitoriza semnalele de mod **in** şi **inout** date ca parametri. Cu alte cuvinte, procedura va fi apelată la fiecare eveniment care va apărea pe parametrii săi care sunt semnale de intrare. Instrucţiunea **wait** este indispensabilă pentru evitarea unei bucle infinite, în care procesul respectiv ar intra în lipsa ei. În cazul în care printre parametrii săi nu există nici un semnal de mod **in** sau **inout**, instrucţiunea **wait** nu va avea nici o condiţie: aceasta înseamnă că procedura nu va fi apelată decât o singură dată, la iniţializare.

În cazul exemplului precedent, procesul echivalent se rezumă la:

```
VERIF1: process
begin
      VERIF_SETUP(A, B, TS);
      wait on A, B;
end process VERIF1;
```

Aşadar, numărul de procese create este egal cu numărul de apeluri ale procedurii respective. Următoarea porţiune de cod creează trei procese:

```
VERIF1: VERIF_SETUP(A, B, TS);
VERIF2: VERIF_SETUP(C, D, TS_CD);
VERIF3: VERIF_SETUP(A, B, TS);
```

Aceste procese vor „trăi” până la sfârşitul simulării.

Utilizarea imediată a apelului concurent de procedură survine atunci când trebuie să creăm mai multe procese implementând aceiaşi algoritmi, dar care fie nu se aplică aceloraşi semnale, fie au constante diferite. Algoritmul se scrie deci o singură dată sub formă de procedură (de exemplu, într-un pachet) şi fiecare instrucţiune de apel va crea un proces specific pe baza parametrilor transmişi.

Printr-o scriere adecvată a procedurii apelate, putem obţine două efecte foarte interesante:

- Dacă procedura se încheie printr-o instrucţiune **wait** fără condiţii, ea se va executa o singură dată, la iniţializare;
- Dacă scriem o procedură sub formă de buclă infinită având grijă să introducem o instrucţiune **wait** (sensibilă la semnalele de intrare, de mod **in** şi **inout**) imediat înainte de sfârşitul buclei, vom suprapune astfel iteraţia procesului peste iterativitatea derulării procedurii respective.

De exemplu, prin apelul concurent al procedurii:

```
procedure VERIF_SETUP (signal S, REF: in BIT; TIMP: TIME) is
begin
      loop
            ...   -- Algoritmul
            ...
            wait on S, REF;
      end loop;
end VERIF_SETUP;
```

se obţine acelaşi comportament ca şi la apelul aceleiaşi proceduri fără buclă şi fără instrucţiune **wait**, cu o singură diferenţă: apelul procedurii VERIF_SETUP nu este efectuat decât o dată şi prin urmare obiectele eventual declarate în partea declarativă a acestei proceduri (în cazul de faţă nu există nici unul) sunt remanente. Acest aspect este deosebit de important în cazul oricărei proceduri care gestionează starea internă a unui automat finit.

2.4 Instrucţiunea concurentă *assert*

Instrucţiunea are o sintaxă identică (cu excepţia etichetei, care este opţională) cu cea a instrucţiunii secvenţiale **assert**.

```
{etichetă:} assert condiţie {report mesaj} {severity
nivel_de_eroare};
```

La fel ca în cazul instrucţiunii secvenţiale, această instrucţiune are ca efect monitorizarea unei condiţii şi emiterea unui mesaj în cazul în care aceasta este *falsă*. De fapt, ea reprezintă o simplă versiune simplificată de scriere a următorului proces:

```
{etichetă:} process
begin
assert condiţie {report mesaj}{severity nivel_de_eroare};
wait on lista_semnalelor_care_apar_în_condiţie;
end process {etichetă};
```

Observaţie

Este important de reţinut că acest proces echivalent nu are listă de sensibilitate, ci conţine o instrucţiune secvenţială **wait** după instrucţiunea secvenţială **assert**. Prin urmare, chiar dacă lista semnalelor care apar în condiţie este vidă, procesul nu se va putea repeta în buclă.

Instrucţiunea **assert** se poate afla într-o specificaţie de entitate sau într-o arhitectură. În cadrul unei specificaţii de entitate, ea permite de exemplu controlarea validităţii semnalelor prezente la porturi sau a valorilor parametrilor generici.

Asemănarea cu instrucţiunea secvenţială **assert** nu trebuie să ascundă puterea acestei instrucţiuni. Într-adevăr, caracterul concurent şi iterativ al procesului face ca monitorizarea condiţiei date să fie permanentă. Simpla scriere a unei instrucţiuni concurente **assert**, de exemplu pentru monitorizarea verificării timpului de prepoziţionare (*set-up*) al unui bistabil, va crea un control permanent al acestei caracteristici.

La fiecare modificare a valorii unui semnal care ar putea modifica valoarea condiţiei, se va efectua un test şi se va genera, eventual, mesajul de eroare specificat. Acest caracter de „viaţă proprie” al controlului este foarte util, cu atât mai mult cu cât absenţa ieşirilor acestui proces (singura „ieşire” este – eventual – mesajul specificat) evită complicarea inutilă a simulării.

Observaţie

Scrierea procesului echivalent al acestei instrucţiuni ascunde o capcană! În cazul în care condiţia nu depinde de nici un semnal, lista semnalelor care intervin în cadrul condiţiei este vidă şi aceasta va fi verificată doar o singură dată, la iniţializare, şi niciodată după aceea.

De exemplu, prin utilizarea condiţiei NOW < TIMP_MAX (unde TIMP_MAX este o constantă şi NOW este funcţia care returnează timpul curent al simulării), nu vom ajunge niciodată la sfârşitul simulării.

Dacă dorim să testăm o condiţie care nu utilizează nici un semnal, atunci vom fi nevoiţi să scriem un proces care va conţine instrucţiunea *secvenţială* **assert** şi va avea o listă de sensibilitate adecvată.

2.5 Instrucţiunea concurentă de asignare de valori semnalelor

Asignarea de valori semnalelor poate îmbrăca o *formă condiţională* sau o *formă selectivă*. Fiecăreia dintre aceste forme îi corespunde câte o formă secvenţială, cea a instrucţiunii de asignare conţinută în procesul echivalent.

Sintaxa *formei condiţionale* este următoarea:

```
{etichetă:} nume_sau_agregat <= {guarded}
      formă_de_undă_1 when condiţie_booleană_1 else
      formă_de_undă_2 when condiţie_booleană_2 else
      ...
      formă_de_undă_n;
```

Presupunând că se omite opţiunea **guarded**, această formă se traduce în următorul proces echivalent (deci, în instrucţiuni secvenţiale):

```
if condiţie_booleană_1 then
      nume_sau_agregat <= {transport} formă_de_undă_1;
elsif condiţie_booleană_2 then
      nume_sau_agregat <= {transport} formă_de_undă_2;
...
else
      nume_sau_agregat <= {transport} formă_de_undă_n;
end if;
```

Această formă se numeşte *forma condiţională echivalentă*.

Sintaxa *formei selective* este următoarea:

```
{etichetă:} with expresie select
      nume_sau_agregat <= {guarded}{transport}
            formă_de_undă_1 when alegere_1,
            formă_de_undă_2 when alegere_2,
            ...
            formă_de_undă_n when alegere_n;
```

La fel ca în cazul formei condiţionale, presupunând că se omite opţiunea **guarded**, această formă se traduce în următorul proces echivalent (*forma selectivă echivalentă*):

```
case expresie is
      when alegere_1 =>
            nume_sau_agregat <= {transport} formă_de_undă_1;
      when alegere_2 =>
            nume_sau_agregat <= {transport} formă_de_undă_2;
            ...
      when alegere_n =>
            nume_sau_agregat <= {transport} formă_de_undă_n;
end case;
```

După cum se poate remarca, există două opţiuni comune acestor două forme:

- Opţiunea **transport**, care a fost deja prezentată în cadrul lucrării nr. 3, specifică un model de propagare non-inerţial (toate impulsurile sunt transmise). Dacă apare în cadrul instrucţiunii concurente de asignare, această opţiune va fi prezentă la nivelul fiecărei asignări secvenţiale a formelor echivalente;
- Opţiunea **guarded** arată că asignarea nu va fi executată decât dacă semnalul GUARD de tip BOOLEAN (declarat implicit într-un bloc gardat sau explicit de către proiectant) are sau este pe cale să primească valoarea TRUE. Eventual se va genera o instrucţiune de deconectare. O asignare care utilizează această opţiune se numeşte *asignare gardată*.

Observaţie

La declararea sa, un semnal poate fi clasificat ca având tipul registru (cuvântul cheie **register**) sau magistrală (cuvântul cheie **bus**). Atunci este vorba despre un semnal gardat (*guarded signal*). Aceste calificări, foarte apropiate de preocupările fizice, hardware, permit descrierea comportamentului unui semnal la deconectarea sa.

Aşadar, în cazul unei instrucţiuni concurente de asignare de semnal, este posibilă apariţia unuia dintre următoarele patru cazuri:

1. Semnal ne-gardat, obiect al unei asignări ne-gardate. Asignarea concurentă se reduce la forma echivalentă (condiţională sau selectivă);
2. Semnal ne-gardat, obiect al unei asignări gardate. Asignarea concurentă se reduce la următoarea condiţie:

```
if GUARD then
      formă echivalentă (condiţională sau selectivă)
end if;
```

3. Semnal gardat, obiect al unei asignări ne-gardate. Acest caz este interzis: compilatorul sau simulatorul VHDL va genera o eroare. Un semnal gardat nu poate fi asignat decât prin intermediul unei instrucţiuni gardate;
4. Semnal gardat, obiect al unei asignări gardate. Asignarea concurentă devine foarte puternică în acest caz, căci ea va gestiona implicit şi deconectarea semnalului, în cazul în care asignarea (care este gardată) nu va fi executată (cu alte cuvinte, dacă condiţia de gardă este falsă).

```
if GUARD then
      formă echivalentă (condiţională sau selectivă)
else
      instrucţiune de deconectare
end if;
```

Instrucţiunea de deconectare îi asignează semnalului gardat un element de formă de undă nul după întârzierea indicată în specificaţia de

deconectare a acestui semnal. Asignarea unui element de formă de undă nul este rezervată semnalelor gardate şi se scrie astfel:

```
semnal_sau_agregat <= null after {expresie_de_tip_TIME};
```

Semnalul astfel asignat rămâne fără sursă („firul este întrerupt") după un interval de timp dat. Această instrucţiune înlătură din pilotul semnalului tot ceea ce este datat „mai târziu" decât timpul dat în momentul specificării deconectării.

Observaţie

O arhitectură este un bloc (fără condiţie de gardă). Într-un bloc fără condiţie de gardă putem însă declara un semnal GUARD şi apoi putem folosi instrucţiuni gardate. Aşadar, exemplul de mai jos este perfect valabil:

```
architecture ARH1 of ENT1 is
signal A: BIT;
signal GUARD: BOOLEAN;
begin
      A <= guarded '1' after 10 ns;
end ARH1;
```

Aşadar, este inutil să creăm un bloc în interiorul unei arhitecturi, dacă acest bloc (numit şi *instrucţiune* ***block***) va fi singura instrucţiune a respectivei arhitecturi.

2.6 Instrucţiunea de instanţiere a unei componente

2.6.1 Instanţierea

A *instanţia* o componentă înseamnă a face o copie (*o instanţă*) a unei componente deja declarate („modelul") şi de a o personaliza pentru a da curs unei necesităţi particulare. De exemplu, dacă am declarat deja o componentă INVERSOR, atunci putem instanţia o componentă INV1, apoi o componentă INV2 etc. Sintaxa instrucţiunii este următoarea:

```
etichetă: numele_componentei_model
 {corespondenţa parametrilor generici}
 {corespondenţa porturi efective / porturi locale modelului};
```

Această instrucţiune permite descrierea unui modul al circuitului ca fiind o instanţă a unui model. Modelul este definit printr-o declaraţie de componentă. Se pot preciza următoarele informaţii:

- *Valoarea parametrilor generici ai componentei*. Acest aspect reprezintă adaptarea componentei la nevoile de modelare curente;
- *Corespondenţa dintre porturile formale ale componentei model şi porturile efective*, cele pe care le vom conecta în circuitul nostru actual. Există posibilitatea de a nu conecta toate porturile modelului, caz în care porturile efective corespunzătoare nu vor exista (vor fi „în aer"). În cazul porturilor de intrare (de mod **in**), acestora li se vor asigna valorile implicite prevăzute eventual pentru ele.

În exemplul de mai jos, se obţin două componente, C1 şi C2, care asigură interconectarea semnalelor A, B, C şi D de tip BIT.

```
signal A, B, C, D: BIT;

-- Declararea componentei INVERSOR
-- INTRARE şi IEŞIRE sunt porturile modelului, iar A, B, C şi
-- D sunt porturile efective ale circuitului nostru
component: INVERSOR
      port(INTRARE: in BIT; IEŞIRE: out BIT);
end component;

-- Componenta INVERSOR va fi instanţiată de două ori
C1: INVERSOR port map (INTRARE => A, IEŞIRE => B);
C2: INVERSOR port map (INTRARE => C, IEŞIRE => D);
```

2.6.2 Diferenţa dintre bloc şi componentă

Ce diferenţe există între un bloc şi o instanţă a unei componente? Când trebuie creat un bloc şi când trebuie instanţiată o componentă?

Nu există nici o diferenţă fundamentală între un bloc şi o instanţă a unei componente: după cum am văzut, o componentă este un bloc pentru că ea este imaginea unei entităţi. Ambele construcţii sunt folosite pentru ierarhizarea *descrierii structurale*.

Un bloc poate fi văzut ca o componentă deja instanţiată. În acest sens, el reprezintă o variantă simplificată de scriere: crearea unui bloc evită recurgerea la referirea unei entităţi, la declararea unei componente, la

configurarea şi apoi la instanţierea sa. Deşi este ceva mai greu de utilizat în practică, componenta are un mare avantaj faţă de bloc: ea este reutilizabilă.

Instanţierile componentelor se fac cu ajutorul unor parametri (porturi) diferite. Această instanţiere este un mecanism foarte puternic, pentru că permite omiterea anumitor porturi şi mai ales schimbarea dinamică (prin configuraţie) a entităţii care va fi luată drept model.

Să remarcăm că blocul are o altă funcţie, efemeră în viaţa ciclului de proiectare: el permite, într-o fază intermediară, crearea şi testarea rapidă a ceea ce – mai târziu – va deveni o componentă. Acest rol nu este deloc neglijabil pentru proiectant!

În concluzie, este foarte util să alegem o structurare în blocuri a proiectului – dacă urmărim o scriere cât mai concisă, o proiectare rapidă sau dacă excludem orice reutilizare a codului. Dimpotrivă, structurarea în componente va aduce gradul de generalitate necesar unor aplicaţii complexe.

2.7 Instrucţiunea *generate*

Această instrucţiune permite elaborarea iterativă sau condiţională a liniilor de cod sursă VHDL; ea NU efectuează o execuţie condiţională sau iterativă a instrucţiunilor înglobate.

Cele două forme (condiţională şi iterativă) ale acestei instrucţiuni sunt:

```
-- Forma condiţională
etichetă: if condiţie_booleană generate
      ...
      Secvenţă de instrucţiuni concurente
      ...
end generate {etichetă};
-- Forma iterativă
etich: for nume_param_generare in interval_discret generate
      ...
      Secvenţă de instrucţiuni concurente
      ...
end generate {etich};
```

Semantica formei condiţionale este foarte simplă: instrucţiunile concurente înglobate nu există decât dacă, la elaborare, condiţia respectivă este adevărată (TRUE).

Extrem de utilă, forma iterativă creează un număr de ansambluri de instrucţiuni înglobate egal cu numărul de elemente cuprins în intervalul discret dat (eventual nici unul). Pentru fiecare ansamblu de instrucţiuni, parametrul generării (un identificator) este înlocuit, oriunde apare, cu valoarea iteraţiei.

Prezentăm în continuare un exemplu care utilizează ambele forme ale acestei instrucţiuni.

Să presupunem că dorim să conectăm o serie de N inversoare, ieşirea unuia fiind conectată la intrarea următorului. Vom folosi în continuare exemplul din secţiunea 2.6.1, în care am instanţiat două componente INVERSOR. În acest caz, problema este diferită, deoarece nu se cunoaşte dinainte numărul de componente necesare.

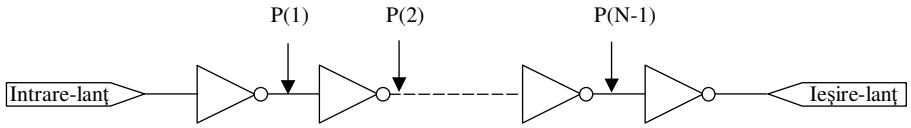

Figura 9.2 *Lanţ de N inversoare: entitatea LANŢ_DE_INVERSOARE*

N va fi un parametru generic transmis entităţii LANŢ_DE_INVERSOARE care trebuie proiectată. Specificaţia acestei entităţi este următoarea:

```
entity LANŢ_DE_INVERSOARE is
      generic (N: INTEGER);
      port (INTRARE_LANŢ: in BIT; IEŞIRE_LANŢ: out BIT);
end LANŢ_DE_INVERSOARE;
```

Entitatea conţine două porturi: INTRARE_LANŢ şi IEŞIRE_LANŢ. Mai rămâne de descris arhitectura sa asociată, ştiind că numărul de inversoare din lanţ este N.

```
architecture STRUCTURALĂ of LANŢ_DE_INVERSOARE is

-- Declararea unui semnal care memorează cele N-1 noduri
-- interne ale lanţului de inversoare
signal P: BIT_VECTOR (1 to N-1);

--Declararea componentei INVERSOR
component INVERSOR
      port (INTRARE: in BIT; IEŞIRE: out BIT);
end component;

-- Toate instanţele componentei INVERSOR precedente vor fi
-- configurate pentru a utiliza specificaţia entităţii cu
-- acelaşi nume(INVERSOR) asociată arhitecturii COMP.
for all: INVERSOR use entity INVERSOR(COMP);
begin

-- Instanţierea primului inversor: forma condiţională a
-- instrucţiunii generate
PRIMUL_INVERSOR: if N > = 1 generate
      INVERSOR_1: INVERSOR port map (INTRARE_LANŢ, P(1));
end generate PRIMUL_INVERSOR;

-- Instanţierea inversoarelor intermediare: forma iterativă a
-- instrucţiunii generate
ALTE_INVERSOARE: for I in 1 to N-2 generate
      INVERSOR_I: INVERSOR port map (P(I), P(I+1));
end generate ALTE_INVERSOARE;

-- Instanţierea ultimului inversor: forma condiţională a
-- instrucţiunii generate
ULTIMUL_INVERSOR: if N > = 2 generate
      INVERSOR_N: INVERSOR port map (P(N-1), IEŞIRE_LANŢ);
end generate ULTIMUL_INVERSOR;
end STRUCTURALĂ;
```

Mai există încă o posibilitate de specificare a arhitecturii: vom utiliza numai forma iterativă a instrucţiunii **generate** şi vom memora N+1 noduri. Cele două noduri suplimentare sunt legate la intrare şi la ieşire prin intermediul a două instrucţiuni de asignare concurentă de semnal.

```
architecture STRUCTURALĂ_2 of LANŢ_DE_INVERSOARE is
-- Declararea unui semnal care memorează cele N+1 noduri
signal P: BIT_VECTOR (1 to N+1);
component INVERSOR
      port (INTRARE: in BIT; IEŞIRE: out BIT);
end component;
for all: INVERSOR use entity INVERSOR(COMP);
begin
-- Instanţierea tuturor componentelor
INVERSOARE: for I in 1 to N generate
      INVERSOR_K: INVERSOR port map (P(I), P(I+1));
end generate INVERSOARE;
-- Primul nod se conectează la intrarea lanţului
P(1) <= INTRARE_LANŢ;
-- Al (N+1)-lea nod se conectează la ieşirea lanţului
IEŞIRE_LANŢ <= P(N+1);
end STRUCTURALĂ_2;
```

Acest exemplu ne permite să facem două observaţii importante:

1. Parametrul N trebuie să fie cunoscut în momentul elaborării: prin urmare, N poate fi o constantă sau un parametru generic;
2. În cazul de faţă, configurarea inversoarelor se face „din mers" pentru toate componentele (**for all...**).

3. Desfăşurarea lucrării

3.1 Se va implementa un DEMULTIPLEXOR 1 la 8 descriind structura sa internă sub formă de porţi logice, unde fiecare poartă logică va constitui un bloc intern.

3.2 Se va proiecta şi implementa un registru de deplasare universal BARREL SHIFTER care poate efectua deplasări la stânga sau la dreapta de maximum 3 poziţii binare. Se va studia oportunitatea utilizării instrucţiunii concurente assert şi a blocurilor gardate.

3.3 Pentru implementarea bistabililor din componenţa registrului de deplasare universal BARREL SHIFTER de la punctul anterior, se vor instanţia componente BISTABIL_D.

3.4 Se va testa exemplul LANŢ_DE_INVERSOARE din lucrare.

3.5 Se va implementa un sumator de numere pe N biţi folosind instrucţiunea **generate**.