PROIECTAREA STRUCTURALĂ ÎN LIMBAJUL VHDL

În această lucrare de laborator se descrie modul în care se poate realiza proiectarea structurală în limbajul VHDL. Se prezintă mai întâi avantajele proiectării structurale şi elementele unei descrieri structurale: declaraţia componentelor, instanţierea componentelor, instanţierea directă a entităţilor şi specificarea configuraţiei. În continuare se prezintă utilizarea bibliotecilor, a pachetelor, a genericelor. La sfârşitul lucrării se descriu formele instrucţiunii `generate` utilizate în cadrul descrierilor structurale.

1. Avantajele proiectării structurale

Descrierile structurale specifică un sistem sub forma unor componente interconectate. Aceste descrieri permit crearea unor *nivele ierarhice* multiple, în care un proiect este divizat în unităţi de proiectare de dimensiuni mai mici. Fiecare unitate de proiectare sau componentă poate fi specificată fie printr-o descriere funcţională, fie printr-o descriere structurală. În ultimul caz, o componentă poate fi formată din mai multe sub-componente, care pot fi specificate la rândul lor sub forma unor descrieri structurale. În final, fiecare componentă primitivă de la nivelul cel mai de jos este specificată sub forma unei descrieri funcţionale.

Proiectarea structurală poate fi realizată, cel puţin în cazul descrierilor la nivelul transferurilor între registre (RTL), prin utilizarea componentelor. Orice pereche entitate-arhitectură poate fi utilizată ca o componentă într-o arhitectură de nivel superior. Astfel, sistemele complexe pot fi construite în mai multe etape din componente de nivel inferior.

Principalele avantaje ale proiectării structurale sunt următoarele:

- Proiectarea structurală pe mai multe nivele ierarhice permite definirea detaliilor unei porţiuni a proiectului la un moment dat, de preferinţă în paralel cu alţi proiectanţi.
- Fiecare componentă poate fi proiectată şi testată individual, înainte de a fi integrată în nivelele superioare ale proiectului. Această testare a nivelelor intermediare este mai simplă decât testarea în cadrul sistemului, şi este de obicei mai completă. Aceasta înseamnă că proiectantul poate avea un grad mai ridicat de încredere în componentele utilizate, ceea ce contribuie şi la integritatea globală a sistemului.
- Componentele utile pot fi colectate în biblioteci, astfel încât ele pot fi reutilizate ulterior în acelaşi proiect sau în alte proiecte. Unul din avantajele sintezei logice este că asemenea componente sau module sunt independente de tehnologie. Gradul de reutilizare a componentelor creşte pe măsură ce sunt disponibile mai multe componente.

2. Elementele unei descrieri structurale

O descriere structurală constă din componente interconectate prin intermediul semnalelor. O componentă poate fi definită în cadrul unei arhitecturi cu ajutorul unei declaraţii `component`, sau poate fi reprezentată de un sistem separat, care este specificat sub forma unei entităţi şi a unei arhitecturi. Pentru utilizarea unei componente declarate anterior, aceasta trebuie *instanţiată* în cadrul descrierii structurale. Instanţierile componentelor reprezintă instrucţiunile de bază într-o arhitectură structurală. Aceste instanţieri sunt concurente unele faţă de altele. În cadrul instanţierii unei componente se specifică *maparea porturilor*, care indică semnalele conectate la porturile componentei. Aceste semnale pot fi specificate ca porturi sau pot fi semnale interne ale sistemului. În ultimul caz, acestea trebuie declarate în secţiunea declarativă a arhitecturii.

2.1. Exemplu de descriere structurală

Elementele unei descrieri structurale vor fi ilustrate mai întâi printr-un exemplu complet. Componentele descrierii structurale vor fi examinate apoi separat în secțiunile următoare. Exemplul prezentat constă din două bistabile de tip D conectate în serie, sub forma unui sistem *pipeline*. Structura circuitului este ilustrată în figura 1.

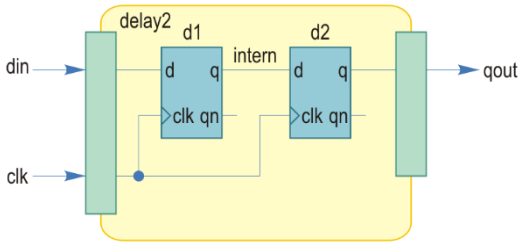

Figura 1. Exemplu de circuit pentru ilustrarea descrierii structurale.

Presupunem că bistabilul D este definit într-o bibliotecă, având definiția de entitate și arhitectură prezentată în Exemplul 1.

Exemplul 1

```
library ieee;
use ieee.std_logic_1164.all;

entity dff is
   port (d, clk: in std_logic;
         q, qn:  out std_logic);
end dff;

architecture arh_dff of dff is
   signal tmp: std_logic;
begin
   process (clk)
   begin
      if rising_edge (clk) then
         tmp <= d;
      end if;
   end process;
   q <= tmp;
   qn <= not tmp;
end arh_dff;
```

Există mai multe posibilități pentru descrierea circuitului considerat cu ajutorul componentelor. O posibilitate de descriere este prezentată în Exemplul 2.

Exemplul 2

```
library ieee;
use ieee.std_logic_1164.all;

entity delay2 is
   port (din, clk: in std_logic;
         qout:     out std_logic);
end delay2;

architecture structural of delay2 is
   signal intern: std_logic;
   -- Declarația componentei
   component dff is
      port (d, clk: in std_logic;
            q, qn:  out std_logic);
   end component dff;
   -- Specificarea configurației
   for all: dff use entity work.dff (arh_dff);
```

```
begin
   -- Instanţierile componentei
   d1: dff port map
      (d => din, clk => clk, q => intern, qn => open);
   d2: dff port map
      (d => intern, clk => clk, q => qout, qn => open);
end structural;
```

Arhitectura conţine trei părţi care se referă la utilizarea componentelor. Acestea sunt indicate prin comentarii, fiind următoarele: declaraţia componentei, specificarea configuraţiei şi instanţierile componentei. Cele trei părţi sunt descrise în secţiunile următoare.

2.2. Declaraţia componentelor

Declaraţia unei componente defineşte interfaţa cu o entitate de proiectare care descrie componenta respectivă. Componenta declarată astfel poate fi utilizată ulterior în instrucţiunile de instanţiere a componentei respective. Declaraţia componentei nu specifică însă care este perechea entitate-arhitectură care descrie componenta şi nici porturile entităţii; aceste informaţii sunt conţinute în specificaţia configuraţiei sau în declaraţia configuraţiei.

Sintaxa simplificată a declaraţiei unei componente este următoarea:

```
component nume_componentă [is]
   generic (listă_generice);
   port (listă_porturi);
end component [nume_componentă];
```

Sintaxa declaraţiei unei componente este similară cu cea a declaraţiei de entitate. Clauza `generic` specifică lista genericelor componentei, iar clauza `port` specifică porturile acesteia. În practică, numele componentei, numele genericelor şi a porturilor acesteia, ca şi ordinea lor, sunt identice cu cele care apar în declaraţia entităţii corespunzătoare componentei.

O componentă poate fi declarată într-o arhitectură, un bloc, o entitate sau un pachet. În cazul în care componenta este declarată într-o arhitectură, aceasta trebuie plasată în partea declarativă a arhitecturii, înaintea cuvântului cheie `begin`. În acest caz, componenta poate fi utilizată (instanţiată) numai în cadrul arhitecturii respective. Atunci când componenta este declarată într-un pachet, ea va fi vizibilă în toate arhitecturile care utilizează pachetul respectiv.

Declaraţia componentei `dff` din Exemplul 2 este reprodusă în continuare:

```
component dff is
   port (d, clk: in std_logic;
         q, qn:  out std_logic);
end component dff;
```

2.3. Instanţierea componentelor

Instanţierea unei componente asociază semnale sau valori cu porturile unei componente definite anterior şi asociază valori cu genericele componentei respective. Sintaxa simplificată a unei instanţieri de componente este următoarea:

```
etichetă: [component] nume_componentă
          [generic map (listă_asociere_generice)]
           port map (listă_asociere_porturi);
```

Instanţierea unei componente introduce o relaţie cu o unitate declarată anterior ca o componentă. Numele componentei instanţiate trebuie să corespundă cu numele componentei declarate anterior. Pentru componenta instanţiată se specifică genericele şi porturile, care reprezintă parametrii actuali ai componentei declarate. Lista de asociere poate fi specificată prin nume sau poate fi poziţională.

Asocierea prin nume permite listarea genericelor şi a porturilor într-o ordine care este diferită de ordinea specificată în declaraţia componentei. În acest caz, fiecărui generic sau port i se asociază în mod explicit o valoare sau un semnal. Numele genericului, respectiv a portului, este urmat de simbolul =>, iar apoi de valoarea care i se atribuie genericului, respectiv de

semnalul la care este conectat portul. Porturile unei componente pot fi lăsate neconectate prin specificarea cuvântului cheie `open`.

În Exemplul 2, pentru porturi s-a utilizat asocierea prin nume. Instanțierile componentei din acest exemplu sunt reproduse mai jos:

```
d1: dff port map
   (d => din, clk => clk, q => intern, qn => open);
d2: dff port map
   (d => intern, clk => clk, q => qout, qn => open);
```

Într-o listă de asociere poziţională, parametrii actuali (genericele şi porturile) sunt specificaţi în aceeaşi ordine în care apar aceştia în declaraţia componentei. În acest caz, numele genericelor sau a porturilor şi simbolul => sunt omise. Instanţierile componentei din Exemplul 2 pot fi rescrise prin utilizarea asocierii poziţionale în felul următor:

```
d1: dff port map (din, clk, intern, open);
d2: dff port map (intern, clk, qout, open);
```

În Exemplul 2, există două instanţieri ale componentei `dff`, care sunt etichetate cu `d1` şi `d2`. Aceste etichete sunt obligatorii şi trebuie să fie unice. Fiecare instanţiere creează un sub-circuit conţinând componenta `dff` şi conexiunile cu această componentă.

Observaţii

- Instanţierile componentelor reprezintă instrucţiuni concurente.
- Instanţierea unei componente reprezintă instanţierea declaraţiei componentei şi nu a entităţii. Relaţia dintre declaraţia componentei şi entitatea care descrie componenta este controlată de specificaţia configuraţiei.

2.4. Instanţierea directă a entităţilor

Nu este necesară întotdeauna definirea unei componente pentru ca aceasta să fie instanţiată, deoarece versiunea VHDL ’93 a limbajului permite instanţierea directă a unei entităţi. Această instanţiere reprezintă forma cea mai simplă de specificare a unui sistem structural. Sintaxa instanţierii directe a unei entităţi este următoarea:

```
etichetă: entity nume_bibliotecă.nume_entitate
                [(nume_arhitectură)]
          [generic map (listă_asociere_generice)]
           port map (listă_asociere_porturi);
```

Instrucţiunea de instanţiere a unei entităţi specifică entitatea de proiectare şi, opţional, numele arhitecturii care trebuie utilizată pentru această entitate. Această entitate poate fi utilizată ulterior ca şi o componentă. Entitatea este specificată prin numele bibliotecii în care este compilată entitatea şi prin numele entităţii. Toate entităţile specificate de utilizator sunt compilate implicit în biblioteca `work`, astfel încât în instrucţiunea de instanţiere a entităţii se specifică, de obicei, această bibliotecă.

Numele arhitecturii trebuie specificat numai atunci când există mai multe arhitecturi definite pentru o singură entitate. Dacă nu este specificat numele arhitecturii şi există mai multe arhitecturi pentru entitatea instanţiată direct, se va utiliza arhitectura care a fost compilată ultima.

Presupunând că entitatea şi arhitectura corespunzătoare bistabilului de tip D din Exemplul 1 sunt compilate în biblioteca `work`, circuitul din figura 1 poate fi descris fără declararea unei componente, prin utilizarea instanţierii directe a entităţii, în modul prezentat în Exemplul 3.

Exemplul 3

```
library ieee;
use ieee.std_logic_1164.all;
```

```
entity delay2 is
   port (din, clk: in std_logic;
         qout:     out std_logic);
end delay2;

architecture structural of delay2 is
   signal intern: std_logic;
begin
   d1: entity work.dff (arh_dff)
       port map (d => din, clk => clk, q => intern, qn => open);
   d2: entity work.dff (arh_dff)
       port map (d => intern, clk => clk, q => qout, qn => open);
end structural;
```

2.5. Specificarea și declararea configurației

În cazul în care nu se utilizează instanțierea directă a entităților, declarațiile unor componente și instanțierile acestora nu sunt suficiente pentru o specificare completă a unei arhitecturi structurale, deoarece nu este specificată descrierea implementării componentelor. În acest caz se poate utiliza o specificație a configurației. O *configurație* este o construcție care definește modul în care instanțierile de componente sunt asociate cu entitățile de proiectare și arhitecturile corespunzătoare.

Motivul pentru separarea unei entități și a componentelor acesteia este de a permite ca asocierea ("legarea") dintre entitate și componente să fie realizată cât mai târziu posibil în procesul de simulare. Această asociere nu se realizează decât la începerea simulării, în faza de elaborare. În acest fel, modulele sursă ale unui proiect ierarhic pot fi compilate în orice ordine.

Sintaxa unei *specificații a configurației* este următoarea:

```
for etichetă_instanțiere: nume_componentă
   use entity nume_bibliotecă.nume_entitate
              [(nume_arhitectură)]
       [generic map (listă_asociere_generice)]
       [port map (listă_asociere_porturi)];
```

Mai multe specificații ale configurației unor componente pot fi incluse într-o *declarație a configurației*, care poate reprezenta o unitate separată de proiectare și astfel poate apare într-un fișier separat. Sintaxa unei declarații a configurației este următoarea:

```
configuration nume_configurație of nume_entitate is
   for nume_arhitectură
      -- specificații de configurații
   end for;
   -- alte clauze for
end [configuration nume_configurație];
```

Se observă că sintaxa unei specificații a configurației este asemănătoare cu cea a instanțierii directe a unei entități. Totuși, specificarea unei configurații reprezintă o metodă mai flexibilă în cazul în care trebuie utilizată o implementare diferită a aceleiași componente. Dacă trebuie efectuate anumite modificări, acestea vor fi introduse numai în fișierul de configurație, iar arhitectura structurală va rămâne neschimbată. Utilizarea instanțierii directe a entităților ar necesita ca toate modificările să fie introduse în cadrul arhitecturii.

O specificație a configurației are trei părți. Prima parte indică acele componente la care se referă configurația. Fiecare componentă este indicată prin eticheta instrucțiunii în care este instanțiată componenta respectivă. Este posibilă utilizarea cuvântului cheie `all` pentru selectarea tuturor componentelor cu numele specificat. Acest cuvânt cheie a fost utilizat și în Exemplul 2, specificația configurației din acest exemplu fiind reprodusă mai jos:

```
for all: dff use entity work.dff (arh_dff);
```

În locul specificării configurației pentru toate componentele cu numele `dff`, se puteau prevedea specificații separate pentru fiecare componentă instanțiată:

```
for d1: dff ...
for d2: dff ...
```

A doua parte a specificației unei configurații selectează entitatea care trebuie utilizată pentru o anumită componentă sau pentru toate componentele cu numele indicat, ca și biblioteca în care se află entitatea respectivă. Această parte poate specifica și arhitectura care va fi utilizată pentru entitatea selectată, în cazul în care există mai multe arhitecturi.

A treia parte a specificației este opțională. Această parte poate specifica în mod explicit modul în care genericele și porturile unei componente instanțiate sunt asociate cu genericele și porturile entității. Pentru aceasta se utilizează clauzele `generic map` și `port map`, iar asocierea poate fi pozițională sau prin nume. Asocierea explicită este necesară numai dacă numele genericelor și porturilor din declarația unei componente sunt diferite de numele genericelor și porturilor din declarația entității utilizate pentru componenta respectivă. În practică se recomandă însă ca aceste nume să fie aceleași.

În cazul în care pentru o componentă specificația unei configurații lipsește complet, se va realiza o asociere implicită. Aceasta înseamnă că pentru acea componentă va fi selectată o entitate cu același nume din biblioteca curentă, se va utiliza arhitectura compilată cel mai recent, iar genericele și porturile sunt asociate cu genericele și porturile cu aceleași nume din cadrul entității.

De cele mai multe ori, asocierea implicită este și cea dorită, astfel încât în aceste cazuri nu este necesară specificarea unei configurații. Există însă un caz în care specificația unei configurații este necesară, și anume atunci când o componentă trebuie asociată cu o entitate dintr-o bibliotecă diferită. O posibilitate pentru realizarea acestei asocieri ar fi utilizarea clauzelor `library` și `use` pentru ca toate entitățile din biblioteca respectivă să fie vizibile. De exemplu, în cazul în care entitatea `dff` a fost compilată în biblioteca numită `basic`, se pot adăuga arhitecturii clauzele `library` și `use`, după cum se ilustrează în Exemplul 4. În acest caz, specificația configurației nu este necesară.

Exemplul 4

```
library ieee;
use ieee.std_logic_1164.all;
library basic;
use basic.all;

entity delay2 is
   port (din, clk: in std_logic;
         qout:     out std_logic);
end delay2;

architecture structural of delay2 is
   signal intern: std_logic;
   component dff is
      port (d, clk: in std_logic;
            q, qn:  out std_logic);
   end component dff;
begin
   d1: dff port map (d => din, clk => clk, q => intern, qn => open);
   d2: dff port map (d => intern, clk => clk, q => qout, qn => open);
end structural;
```

Problema care apare în cazul utilizării acestei metode este că toate entitățile din biblioteca respectivă devin vizibile, indiferent dacă ele vor fi utilizate sau nu. Din acest motiv, pot apare conflicte între numele entităților din bibliotecă și alte nume din unitățile de proiectare în care biblioteca este vizibilă. O soluție mai avantajoasă este specificarea unei configurații prin care se asociază componenta cu entitatea, utilizându-se numele implicite pentru generice și porturi. Această soluție este ilustrată în Exemplul 5.

Exemplul 5

```
library ieee;
use ieee.std_logic_1164.all;
library basic;

entity delay2 is
   port (din, clk: in std_logic;
         qout:     out std_logic);
end delay2;

architecture structural of delay2 is
   signal intern: std_logic;
   component dff is
      port (d, clk: in std_logic;
            q, qn:  out std_logic);
   end component dff;
   for all: dff use entity basic.Dff;
begin
   d1: dff port map (d => din, clk => clk, q => intern, qn => open);
   d2: dff port map (d => intern, clk => clk, q => qout, qn => open);
end structural;
```

Observații

- În general, utilitarele de sinteză nu permit specificarea unei configurații. Proiectantul trebuie să se asigure că numele entităților și a componentelor, ca și a genericelor și a porturilor, sunt aceleași.
- Pentru configurația unei entități de proiectare, atât entitatea cât și configurația trebuie declarate în aceeași bibliotecă.

3. Biblioteci

O bibliotecă reprezintă un director sau un fișier în care se pot compila unitățile de proiectare. De exemplu, o declarație de entitate și arhitectura corespunzătoare, aflate într-un fișier, pot fi compilate într-o bibliotecă. Formatul intern al bibliotecii poate fi specific unui anumit sistem de proiectare. Unitățile de proiectare (de exemplu, entitățile) dintr-o bibliotecă pot fi utilizate în cadrul altor entități, dacă biblioteca și unitățile respective sunt vizibile.

Pentru a se utiliza unitățile de proiectare dintr-un pachet, în primul rând trebuie ca biblioteca din care face parte pachetul să fie accesibilă, ceea ce se obține prin utilizarea unei clauze `library`. În această clauză se pot specifica mai multe nume de biblioteci, separate prin virgule. Pentru utilizarea unităților de proiectare dintr-o anumită bibliotecă, trebuie ca pachetele, componentele, declarațiile, funcțiile și procedurile respective să fie vizibile, ceea ce se obține prin utilizarea unei clauze `use`.

Există două biblioteci predefinite care sunt utilizate în mod implicit în fiecare proiect: `std` și `work`. Biblioteca `std` conține pachetele cu numele `standard` și `textio`. Biblioteca `work` este locul în care se plasează, în mod implicit, unitățile de proiectare care sunt compilate în timpul dezvoltării proiectului. După verificarea corectitudinii unei unități de proiectare aflată în biblioteca `work`, aceasta poate fi compilată într-o altă bibliotecă, dacă unitatea de proiectare trebuie reutilizată în același proiect sau în proiectele ulterioare. Bibliotecile `std` și `work` sunt vizibile în mod implicit pentru toate proiectele, astfel încât pentru aceste biblioteci nu este necesară utilizarea clauzei `library`.

În lucrările de laborator anterioare a fost utilizată biblioteca predefinită `ieee`. Aceasta conține unitățile de proiectare standard definite de IEEE (altele decât cele definite de standardul IEEE 1076), cum sunt pachetele `std_logic_1164`, `numeric_std` și `numeric_bit`.

Observații

- O bibliotecă specificată într-o clauză `library` înainte de o unitate primară de proiectare (entitate, configurație sau pachet) este vizibilă în fiecare unitate secundară de proiectare (arhitectură sau corp al pachetului) asociată cu unitatea primară de proiectare.

4. Pachete

Un pachet este o unitate de proiectare având rolul de a grupa diferite declarații care sunt vizibile pentru alte unități de proiectare. Din acest punct de vedere, declarațiile din cadrul pachetelor se deosebesc de cele din cadrul arhitecturilor, care nu sunt vizibile pentru alte unități de proiectare.

Un pachet constă dintr-o *declarație a pachetului* și, în mod opțional, dintr-un *corp al pachetului*. Declarația pachetului este utilizată pentru a declara tipuri, componente, funcții și proceduri. Corpul pachetului conține definițiile funcțiilor și procedurilor pachetului.

4.1. Declarația pachetelor

Declarația unui pachet definește interfața cu pachetul respectiv. Sintaxa declarației unui pachet este următoarea:

```
package nume_pachet is
   [tipuri]
   [constante]
   [semnale]
   [fișiere]
   [funcții]
   [proceduri]
   [clauze use]
   [declarații]
end [package nume_pachet];
```

În cadrul declarației pachetului se pot declara tipuri, subtipuri, constante, semnale, fișiere, subprograme, componente și atribute. După declararea pachetului, acesta poate fi utilizat în diferite proiecte independente. Pentru ca elementele dintr-o declarație a unui pachet să fie vizibile în alte unități de proiectare trebuie să se utilizeze o clauză `use` de forma:

```
use nume_bibliotecă.nume_pachet.element;
```

Dacă se dorește ca toate elementele din pachet să fie vizibile, se poate utiliza cuvântul rezervat `all`. Cu excepția cazului în care există un element în cadrul pachetului a cărei definiție este în conflict cu cea a unui element dintr-un alt pachet, de obicei toate elementele pachetului sunt declarate vizibile. Aceasta nu are ca efect creșterea timpului de compilare.

Specificarea în două etape a unui pachet (declarația, respectiv corpul pachetului) permite definirea unor așa-numite *constante întârziate*, care nu au valori asignate în declarația pachetului. Valoarea unei constante întârziate trebuie definită însă în corpul pachetului care este asociat cu declarația pachetului.

Declarația unui pachet poate conține declarația unui subprogram (funcție sau procedură). Corpul subprogramului nu poate apare însă în cadrul declarației pachetului, ci trebuie să apară în corpul pachetului.

4.2. Corpul pachetelor

Corpul unui pachet definește corpul subprogramelor și valorile constantelor întârziate specificate în declarația pachetului. Sintaxa definiției corpului unui pachet este următoarea:

```
package body nume_pachet is
   [subprograme]
   [constante]
   [tipuri]
   [semnale]
```

```
   [declarații]
end [package body nume_pachet];
```

Pe lângă corpul subprogramelor și valorile constantelor întârziate, corpul pachetului mai poate conține declarații similare cu cele din declarația pachetului, dar acestea sunt vizibile numai în cadrul corpului pachetului.

Observații

- În corpul unui pachet, declarațiile diferite de specificarea valorii constantelor întârziate și corpul subprogramelor nu sunt vizibile în exteriorul corpului pachetului, astfel încât pot fi utilizate numai local.
- Fiecărei declarații a unui pachet i se poate asocia un singur corp al pachetului.

4.3. Pachete cu declarații de componente

Una din utilizările frecvente ale pachetelor este pentru păstrarea declarațiilor unor componente care pot fi refolosite în proiectele viitoare. După plasarea declarației unei componente într-un pachet și compilarea pachetului într-o bibliotecă, pentru utilizarea componentei trebuie specificată doar o clauză `use`.

Pentru ilustrarea utilizării pachetelor care conțin declarații de componente, considerăm exemplul utilizat anterior al bistabilului de tip D. Declarația pachetului conținând componenta `dff` este prezentată în Exemplul 6.

Exemplul 6

```
library ieee;
use ieee.std_logic_1164.all;

package dff_pkg is
   component dff is
      port (d, clk: in std_logic;
            q, qn:  out std_logic);
   end component dff;
end package dff_pkg;
```

Presupunând că pachetul `dff_pkg` este compilat în biblioteca curentă `work`, componenta `dff` poate fi utilizată în circuitul `delay2` din exemplele anterioare în modul indicat în Exemplul 7.

Exemplul 7

```
library ieee;
use ieee.std_logic_1164.all;
use work.dff_pkg.all;

entity delay2 is
   port (din, clk: in std_logic;
         qout:     out std_logic);
end delay2;

architecture structural of delay2 is
   signal intern: std_logic;
begin
   d1: dff port map (d => din, clk => clk, q => intern, qn => open);
   d2: dff port map (d => intern, clk => clk, q => qout, qn => open);
end structural;
```

În acest exemplu nu există o specificație a configurației, astfel încât se va utiliza configurația implicită.

Un pachet poate fi compilat și într-o altă bibliotecă, diferită de cea curentă `work`. Aceasta nu trebuie să fie aceeași bibliotecă în care este compilată entitatea asociată componentei. Totuși, în practică pachetul cu declarația unei componente se compilează în aceeași

bibliotecă ca și perechea entitate-arhitectură corespunzătoare componentei. Asemenea biblioteci sunt puse la dispoziție de producătorii diferitelor familii de circuite. De obicei, numele pachetului care conține declarațiile componentelor este același cu numele bibliotecii.

5. Generice și componente parametrizate

5.1. Principiul genericelor

Genericele reprezintă un mecanism general pentru transmiterea unor parametri la diferitele instanțieri ale unei entități sau ale unei componente. Parametrii transmiși unei entități sau componente reprezintă date statice. După elaborarea modelului, aceste date nu se modifică în timpul simulării. Datele conținute în genericele transmise entității sau componentei se pot utiliza pentru a modifica rezultatele simulării, dar rezultatele nu pot modifica genericele.

Genericele pot fi declarate în cadrul entităților sau a declarațiilor componentelor. Declarația genericelor trebuie plasată înaintea declarației porturilor. Sintaxa declarației este următoarea:

```
generic (listă_generice);
```

Lista genericelor conține numele genericelor declarate, tipul acestora și, opțional, valorile inițiale care li se asignează.

Pentru simulare, cei mai utilizați parametri transmiși unei entități sau componente sunt timpii de întârziere ai unor semnale. Genericele pot fi utilizate și pentru transmiterea unor tipuri de date definite de utilizatori, cum sunt capacitatea de încărcare sau rezistența.

Pentru sinteză, genericele pot fi utilizate pentru proiectarea unor componente parametrizate, pentru care dimensiunile și caracteristicile sunt specificate prin valorile parametrilor de instanțiere. Genericele sunt foarte utile mai ales atunci când se dorește reutilizarea unor componente sau subsisteme proiectate anterior. Utilizarea cea mai frecventă a genericelor este pentru parametrizarea dimensiunii unor porturi, buffere, registre sau numărătoare. De exemplu, același model al unei unități aritmetice și logice poate fi utilizat pentru cazul în care operanzii au o dimensiune de 16 biți sau de 32 biți. Se poate parametriza, de asemenea, numărul etajelor *pipeline* ale unei căi de date.

5.2. Definirea entităților generice

În Exemplul 8 se prezintă declarația de entitate și de arhitectură a unei porți ȘI cu două intrări căreia i se asociază trei generice: întârzierile pentru frontul crescător și descrescător (`rise`, `fall`) și încărcarea ieșirii (`load`). Cu aceste informații, descrierea poate modela în mod corect o poartă ȘI.

Exemplul 8

```
library ieee;
use ieee.std_logic_1164.all;

entity and2 is
   generic (rise, fall: time; load: integer);
   port (i, j: in std_logic;
         k:    out std_logic);
end and2;

architecture arh_and2 of and2 is
   signal tmp: std_logic;
begin
   tmp <= i and j;
   k <= tmp after (rise + (load * 2 ns))
        when tmp = '1'
        else tmp after (fall + (load * 3 ns));
end arh_and2;
```

În Exemplul 9 se prezintă descrierea unui sumator cu transport succesiv în care se utilizează un generic `n`, care specifică dimensiunea sumatorului. În cadrul entității, acest generic este utilizat pentru definirea dimensiunii porturilor. În cadrul arhitecturii, genericul `n` este utilizat pentru specificarea domeniului buclei `for`.

Exemplul 9

```
library ieee;
use ieee.std_logic_1164.all;

entity add_succ is
   generic (n: natural);
   port (a, b: in std_logic_vector (n-1 downto 0);
         cin:  in std_logic;
         s:    out std_logic_vector (n-1 downto 0);
         cout: out std_logic);
end add_succ;

architecture structural of add_succ is
begin
   process (a, b, cin)
      variable c: std_logic;
   begin
      c := cin;
      for i in 0 to n-1 loop
         s(i) <= a(i) xor b(i) xor c;
         c := (a(i) and b(i)) or (a(i) and c) or
              (b(i) and c);
      end loop;
      cout <= c;
   end process;
end structural;
```

5.3. Utilizarea componentelor generice

O entitate generică poate fi utilizată ca şi o componentă în acelaşi mod în care se utilizează o entitate fără generice. Singura diferență este că trebuie specificate valori pentru parametrii generici. Aceste valori trebuie să fie constante și pot fi specificate ca valori numerice direct în clauza `generic map` a instanțierii componentei sau a entității.

Descrierea din Exemplul 10 conține instanțierea directă a entității `and2` din Exemplul 8 utilizând maparea genericelor cu ajutorul clauzei `generic map`.

Exemplul 10

```
library ieee;
use ieee.std_logic_1164.all;

entity test is
   generic (rise, fall: time; load: integer);
   port (a, b, c, d: in std_logic;
         x, y:       out std_logic);
end test;

architecture arh_test of test is
begin
   u1: entity work.and2
            generic map (rise => 5 ns, fall => 6 ns, load => 3)
            port map (i => a, j => b, k => x);
   u2: entity work.and2
            generic map (rise => 4 ns, fall => 5 ns, load => 5)
            port map (i => c, j => d, k => y);
end arh_test;
```

Genericele pot fi inițializate cu valori implicite, care sunt înlocuite de valorile actuale dacă acestea sunt transmise la instanțiere. În caz contrar, se vor utiliza valorile implicite. Un exemplu de inițializare a genericelor este următorul:

```
generic (rise, fall: time := 5 ns; load: integer := 0);
```

În Exemplul 11 se utilizează entitatea `add_succ` din Exemplul 9 pentru realizarea unui sumator cu transport succesiv de 8 biți. Entitatea sumatorului este instanțiată direct, indicându-se valoarea genericului `n`.

Exemplul 11

```
library ieee;
use ieee.std_logic_1164.all;

entity add_8 is
   port (a, b: in std_logic_vector (7 downto 0);
         cin:  in std_logic;
         s:    out std_logic_vector (7 downto 0);
         cout: out std_logic);
end add_8;

architecture structural of add_8 is
begin
   add: entity work.add_succ
      generic map (n => 8);
      port map (a => a, b => b, cin => cin, s => s, cout => cout);
end add_8;
```

5.4. Tipul parametrilor generici

În limbajul VHDL, parametrii generici pot fi de orice tip. Totuși, utilitarele de sinteză limitează tipul parametrilor generici care pot fi utilizați. În general, singurele tipuri care pot fi utilizate sunt tipurile întregi, deși unele utilitare de sinteză permit și utilizarea tipurilor enumerate, dintre care tipurile cele mai utile sunt `bit` și `boolean`.

Modelul porții ȘI din Exemplul 8 nu poate fi utilizat pentru sinteză, deoarece în instrucțiunile de asignare se utilizează clauza `after`. De asemenea, în Exemplul 10 se utilizează generice de tip `time` la declararea porții ȘI sub forma unei componente. Exemplele care prezintă sumatorul cu transport succesiv pot fi utilizate pentru sinteză, deoarece genericul utilizat este de tip `natural`, care este un subtip al tipului `integer`. Utilizarea tipului `natural` nu va permite generarea unui sumator cu un număr de biți care este negativ.

Toate tipurile de tablouri utilizate pentru sinteză sunt indexate utilizând tipul `natural`. Asemenea tipuri sunt `bit_vector`, `std_logic_vector`, ca și tipurile `signed` și `unsigned` definite în pachetele `numeric_bit` și `numeric_std`. Astfel, nu apar probleme la utilizarea genericelor pentru specificarea domeniului porturilor de tip tablou, ca în exemplele sumatorului cu transport succesiv.

Utilizarea parametrilor generici de alte tipuri permite parametrizarea și a altor caracteristici ale circuitelor. De exemplu, tipul `boolean` poate fi utilizat în diferite construcții condiționale. Un exemplu de acest tip este prezentat în secțiunea 6.2, care descrie instrucțiunea `if generate`. Se menționează că un parametru de tip `boolean` poate fi înlocuit cu un parametru de tip `integer`. De exemplu, valoarea booleană `FALSE` poate fi reprezentată prin valoarea 0, iar valoarea `TRUE` prin valoarea 1.

6. Instrucțiunea `generate`

Instrucțiunea `generate` simplifică descrierea unor circuite cu o structură repetitivă. Această instrucțiune se utilizează în mod obișnuit pentru specificarea unui grup de componente identice printr-o singură specificație a componentelor și repetarea acestora prin mecanismul de repetare al instrucțiunii. Instrucțiunea `generate` este similară cu o instrucțiune `loop`, cu deosebirea că, în timp ce `loop` este o instrucțiune secvențială, `generate` este o instrucțiune concurentă. Aceasta înseamnă că instrucțiunea `generate` nu poate apare în interiorul unui proces.

Există două tipuri de instrucțiuni `generate`: instrucțiunea `for generate`, utilizată pentru structuri repetitive, și instrucțiunea `if generate`, utilizată pentru structuri condiționale.

6.1. Instrucțiunea `for generate`

Instrucțiunea `for generate` specifică repetarea unui grup de instrucțiuni concurente de un anumit număr de ori. Această instrucțiune este de fapt o formă concurentă a instrucțiunii `for loop`. Fiind însă o instrucțiune concurentă, poate fi utilizată pentru repetarea altor instrucțiuni concurente, cum sunt procesele, blocurile, instrucțiunile concurente de apel a procedurilor, instrucțiunile concurente de asignare a semnalelor, instanțierile componentelor sau alte instrucțiuni `generate`.

Sintaxa instrucțiunii `for generate` este următoarea:

```
etichetă: for contor in domeniu generate
   [declarații
begin]
   instrucțiuni_concurente
end generate [etichetă];
```

Sintaxa acestei instrucțiuni este similară cu cea a instrucțiunii `for loop`. Eticheta care precede construcția `for` este obligatorie. Contorul de iterații este numit constantă de generare, deoarece valoarea acestuia este tratată ca o constantă în cadrul buclei. După terminarea unei iterații, contorului de iterații i se asignează următoarea valoare a domeniului. Domeniul poate fi unul crescător sau descrescător. Domeniul contorului de iterații trebuie să fie constant, deoarece sistemul de sinteză trebuie să cunoască numărul de generări ale structurii. De aceea, nu este posibilă utilizarea unui semnal pentru definirea domeniului. Partea declarativă a instrucțiunii permite declararea locală a unor tipuri, atribute, constante, semnale, componente, subprograme, configurații sau fișiere.

Observație

- Dacă instrucțiunea `for generate` nu conține declarații locale, nu este necesară utilizarea cuvântului cheie `begin`. Aceeași observație este valabilă și pentru instrucțiunea `if generate`.

Pentru ilustrarea instrucțiunii `for generate`, se va rescrie exemplul sumatorului cu transport succesiv utilizând instrucțiuni concurente de asignare a semnalelor. Descrierea modificată este prezentată în Exemplul 12.

Exemplul 12

```
library ieee;
use ieee.std_logic_1164.all;

entity add_succ is
   generic (n: natural := 4);
   port (a, b: in std_logic_vector (n-1 downto 0);
         cin:  in std_logic;
         s:    out std_logic_vector (n-1 downto 0);
         cout: out std_logic);
end add_succ;
architecture structural of add_succ is
   signal c: std_logic_vector (n downto 0);
begin
   c(0) <= cin;
   gen: for i in 0 to n-1 generate
      s(i) <= a(i) xor b(i) xor c(i);
      c(i+1) <= (a(i) and b(i)) or
                (a(i) and c(i)) or
                (b(i) and c(i));
   end generate;
   cout <= c(n);
```

```
end structural;
```

În acest exemplu, semnalul de transport `c` s-a transformat într-un vector. Fiecare bit de transport este un semnal separat, iar numărul de semnale necesare depinde de parametrul generic `n`. Dimensiunea vectorului este dată de parametrul generic, această dimensiune fiind mai mare cu 1 față de dimensiunea operanzilor care se adună pentru a furniza un transport de ieșire.

Presupunând că valoarea genericului `n` este 4, arhitectura echivalentă cu cea în care se utilizează instrucțiunea `for generate` de mai sus este următoarea:

```
architecture structural of add_succ is
   signal c: std_logic_vector (4 downto 0);
begin
   c(0) <= cin;
   s(0) <= a(0) xor b(0) xor c(0);
   c(1) <= (a(0) and b(0)) or (a(0) and c(0))
           or (b(0) and c(0));
   s(1) <= a(1) xor b(1) xor c(1);
   c(2) <= (a(1) and b(1)) or (a(1) and c(1))
           or (b(1) and c(1));
   s(2) <= a(2) xor b(2) xor c(2);
   c(3) <= (a(2) and b(2)) or (a(2) and c(2))
           or (b(2) and c(2));
   s(3) <= a(3) xor b(3) xor c(3);
   c(4) <= (a(3) and b(3)) or (a(3) and c(3))
           or (b(3) and c(3));
   cout <= c(4);
end structural;
```

Circuitul rezultat prin sinteza descrierii din exemplul anterior este ilustrat în figura 2.

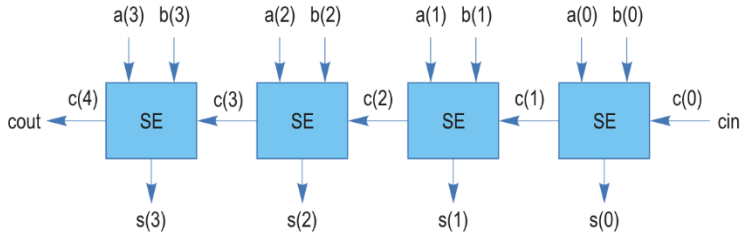

Figura 2. Sumator de 4 biți cu transport succesiv.

Figura 3 prezintă un registru de deplasare de 8 biți realizat cu bistabile D. Acest registru are o intrare de ceas *clk*, o intrare serială *sin* și o ieșire serială *sout*.

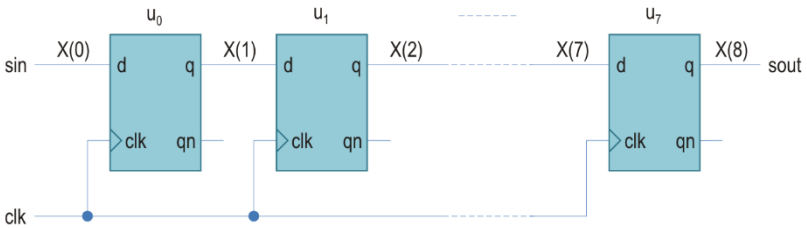

Figura 3. Registru de deplasare de 8 biți realizat cu bistabile D.

Exemplul 13 prezintă descrierea registrului de deplasare de 8 biți din figura 3 utilizând o instrucțiune `for generate`. Declarațiile de entitate și arhitectură ale componentei `dff` au fost prezentate în Exemplul 1.

Exemplul 13

```
library ieee;
use ieee.std_logic_1164.all;
```

```
entity reg_depl is
   generic (n: natural := 8);
   port (sin, clk: in std_logic;
         sout:     out std_logic);
end reg_depl;

architecture structural of reg_depl is
   signal X: std_logic_vector (0 to n);
begin
   X(0) <= sin;
   gen_depl: for i in 0 to n-1 generate
      u: entity work.dff
         port map (d => X(i), clk => clk, q => X(i+1), qn => open);
   end generate;
   sout <= X(n);
end structural;
```

6.2. Instrucțiunea `if generate`

Instrucțiunea `if generate` specifică repetarea condiționată a unui grup de instrucțiuni concurente. Această instrucțiune este utilă pentru generarea unor circuite cu o structură repetitivă, dar care conțin anumite iregularități, de exemplu, pentru primul și ultimul element al structurii. O altă utilizare a acestei instrucțiuni este pentru descrierea unor structuri opționale. De exemplu, unei componente de uz general i se poate adăuga un registru de ieșire opțional, iar această adăugare poate fi controlată cu un parametru generic de tip `boolean`.

Sintaxa instrucțiunii `if generate` este următoarea:

```
etichetă: if condiție generate
   [declarații
begin]
   instrucțiuni_concurente
end generate [etichetă];
```

Condiția este reprezentată de o variabilă booleană, un parametru generic de tip boolean, o expresie booleană sau o expresie condițională care se evaluează la o constantă booleană. Nu este posibilă utilizarea unui semnal în cadrul expresiei condiționale.

Instrucțiunea `if generate` va fi ilustrată printr-un exemplu reprezentând descrierea unui registru de ieșire opțional (Exemplul 14). Exemplul constă doar din descrierea registrului, dar această descriere poate fi adăugată în orice entitate generică.

Exemplul 14

```
library ieee;
use ieee.std_logic_1164.all;

entity reg_optional is
   generic (n: natural; mem: boolean);
   port (a:   in std_logic_vector (n-1 downto 0);
         clk: in std_logic;
         z:   out std_logic_vector (n-1 downto 0));
end reg_optional;

architecture structural of reg_optional is
begin
   gen: if mem generate
      process (clk)
      begin
         if rising_edge (clk) then
            z <= a;
         end if;
      end process;
   end generate;

   ngen: if not mem generate
      z <= a;
```

```
   end generate;
end structural;
```

Dacă parametrul generic `mem` are valoarea `TRUE`, ieșirea `z` va fi o versiune memorată într-un registru a intrării `a`. Dacă însă parametrul generic are valoarea `FALSE`, ieșirea `z` va fi conectată direct la intrarea `a`.

Este important să se considere ambele condiții posibile ale unei instrucțiuni `if generate`, după cum s-a ilustrat și în exemplul anterior. Aceasta deoarece în limbajul VHDL nu există o instrucțiune `else generate`.

Exemplul 15 prezintă descrierea modificată a registrului de deplasare de 8 biți din Exemplul 13 în care se utilizează instrucțiuni `if generate`. Aceste instrucțiuni permit conectarea diferită a primului și ultimului bistabil față de celelalte bistabile ale registrului.

Exemplul 15

```
library ieee;
use ieee.std_logic_1164.all;
entity reg_depl is
   generic (n: natural := 8);
   port (sin, clk: in std_logic;
         sout:     out std_logic);
end reg_depl;

architecture structural of reg_depl is
   signal X: std_logic_vector (0 to n);
begin
   gen_depl: for i in 0 to n-1 generate
      gen_A: if i = 0 generate
         UA: entity work.dff port map (d => sin, clk => clk,
                                       q => X(i+1), qn => open);
      end generate gen_A;
      gen_B: if (i > 0) and (i < n-1) generate
         UB: entity work.dff port map (d => X(i), clk => clk,
                                       q => X(i+1), qn => open);
      end generate gen_B;
      gen_C: if i = n-1 generate
         UC: entity work.dff port map (d => X(i), clk => clk,
                                       q => sout, qn => open);
      end generate gen_C;
   end generate gen_depl;
end structural;
```

7. Aplicații

7.1. Desenați schema bloc a unui numărător binar asincron de *n* biți care numără în sus, realizat cu bistabile de tip D având o intrare de resetare asincronă. Descrieți numărătorul utilizând instrucțiunea `for generate`. Utilizați simulatorul ISim pentru a simula funcționarea numărătorului pentru *n*=8.

7.2. Modificați numărătorul binar asincron din aplicația 7.1 pentru a realiza numărarea în jos. Utilizați o instrucțiune `for generate` și instrucțiuni `if generate`. Simulați funcționarea numărătorului cu simulatorul ISim pentru *n*=8.

7.3. Utilizați bistabile de tip T pentru a construi un numărător binar sincron de *n* biți care numără în sus (figura 4). Numărătorul are o intrare de validare *en*, o intrare de ceas *clk* și o intrare de resetare sincronă *rst*. Fiecare bistabil T are o intrare de validare a ceasului *ce* și comută starea ieșirii la fiecare impuls de ceas, dacă intrarea *ce* este activată. Creați un modul VHDL pentru bistabilul de tip T. Apoi creați un modul VHDL pentru numărător și instanțiați bistabilul utilizând instrucțiunea `for generate`. Simulați funcționarea numărătorului cu simulatorul ISim pentru *n*=8.

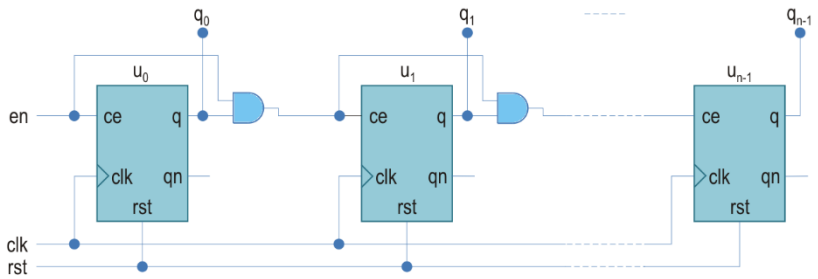

Figura 4. Schema bloc a unui numărător sincron binar de *n* biți cu numărare în sus.

7.4. Proiectați o unitate aritmetică și logică simplă care permite efectuarea următoarelor operații cu operanzi de 4 biți: adunare, ȘI logic, SAU logic, complement logic, deplasare logică la dreapta cu o poziție și deplasare la stânga cu o poziție. Simulați funcționarea unității aritmetice și logice cu simulatorul ISim.