INSTRUCȚIUNI CONCURENTE ÎN LIMBAJUL VHDL

Operațiile din sistemele reale se execută în mod concurent. Limbajul VHDL modelează sistemele reale sub forma unui set de subsisteme care funcționează în mod concurent. Fiecare din aceste subsisteme poate fi specificat sub forma unui proces separat, iar comunicația dintre procese se poate realiza prin semnale. Complexitatea diferitelor procese poate fi foarte variată, de la o simplă poartă logică până la un procesor. Modelarea sistemelor reale sub această formă poate fi realizată cu ajutorul instrucțiunilor concurente.

În prima parte, această lucrare de laborator prezintă formatul, utilizarea și sinteza principalelor instrucțiuni concurente. În partea a doua, lucrarea de laborator descrie unele circuite combinaționale și secvențiale de bază: multiplexoare, decodificatoare, codificatoare prioritare, circuite de deplasare combinațională, bistabile, registre, registre de deplasare și numărătoare.

În secțiunile următoare se prezintă mai întâi structura și execuția unei arhitecturi, iar apoi sunt descrise principalele instrucțiuni concurente ale limbajului VHDL. Cea mai importantă instrucțiune concurentă este declarația unui proces. Procesele au fost prezentate în lucrarea dedicată instrucțiunilor secvențiale, astfel încât în această lucrare de laborator vor fi prezentate doar principalele caracteristici ale proceselor. Alte instrucțiuni concurente sunt instrucțiunea concurentă de asignare a semnalelor, instrucțiunea `block`, instrucțiunea concurentă de apel a unei proceduri, instanțierea unei componente și instrucțiunea `generate`. Instanțierea unei componente și instrucțiunea `generate` vor fi descrise în lucrarea de laborator dedicată proiectării structurale.

1. Sintaxa și utilizarea instrucțiunilor concurente

1.1. Structura și execuția unei arhitecturi

Definiția unei arhitecturi are două părți: o parte declarativă și o parte descriptivă. În partea declarativă se pot declara obiecte care sunt interne arhitecturii. Partea descriptivă conține instrucțiuni concurente. Acestea definesc procesele sau blocurile interconectate care descriu funcționarea sau structura globală a sistemului.

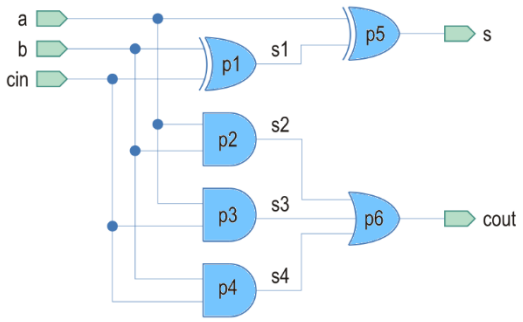

Figura 1. Schema unui sumator de 1 bit.

Toate procesele dintr-o arhitectură se execută în paralel unele față de altele, dar instrucțiunile din cadrul unui anumit proces se execută secvențial. Un proces suspendat este activat din nou atunci când unul din semnalele din lista sa de sensibilitate își modifică valoarea. Atunci când există mai multe procese într-o arhitectură, la modificarea valorii unui semnal

sunt activate toate procesele care conțin acest semnal în lista lor de sensibilitate. Instrucțiunile din cadrul proceselor activate sunt executate secvențial, dar independent de instrucțiunile din alte procese.

Figura 1 prezintă schema unui sumator de 1 bit. În Exemplul 1 fiecare poartă din această schemă este descrisă printr-un proces separat care se execută în mod concurent cu celelalte procese.

Exemplul 1

```
library ieee;
use ieee.std_logic_1164.all;
entity add_1 is
   port (a, b, cin: in std_logic;
         s, cout: out std_logic);
end add_1;

architecture procese of add_1 is
   signal s1, s2, s3, s4: std_logic;
begin

   p1: process (b, cin)
   begin
      s1 <= b xor cin;
   end process p1;

   p2: process (a, b)
   begin
      s2 <= a and b;
   end process p2;

   p3: process (a, cin)
   begin
      s3 <= a and cin;
   end process p3;

   p4: process (b, cin)
   begin
      s4 <= b and cin;
   end process p4;

   p5: process (a, s1)
   begin
      s <= a xor s1;
   end process p5;

   p6: process (s2, s3, s4)
   begin
      cout <= s2 or s3 or s4;
   end process p6;

end procese;
```

Comunicația între procese poate fi realizată cu ajutorul instrucțiunilor de asignare a semnalelor. Acestea se pot utiliza atât pentru activarea proceselor, cât și pentru sincronizarea între procese. Astfel, un semnal poate aștepta un eveniment asupra unui semnal de intrare, semnal care este asignat într-un alt proces. Acest semnal este declarat în partea declarativă a arhitecturii, și astfel este vizibil pentru toate procesele din cadrul arhitecturii.

Observație

- Pentru comunicația între procese se pot utiliza doar semnale, nu și variabile, deoarece acestea sunt obiecte locale în procesul în care sunt declarate.

1.2. Procese

Procesele sunt compuse din instrucțiuni secvențiale, dar declarațiile proceselor reprezintă instrucțiuni concurente. Declarația unui proces a fost prezentată în lucrarea de laborator dedicată instrucțiunilor secvențiale. Se pot formula următoarele caracteristici ale unui proces:

- Se execută în paralel cu alte procese;
- Nu poate conține instrucțiuni concurente;
- Definește o regiune a arhitecturii unde instrucțiunile se execută secvențial;
- Trebuie să conțină o listă de sensibilitate explicită sau o instrucțiune `wait`;
- Permite descrieri funcționale, asemănătoare limbajelor de programare;
- Permite accesul la semnalele definite în arhitectura în care apare procesul și la cele definite în entitatea cu care este asociată arhitectura.

1.3. Instrucțiuni concurente de asignare a semnalelor

O instrucțiune concurentă de asignare a valorii unui semnal este echivalentă cu un proces conținând acea instrucțiune. O asemenea instrucțiune este executată în paralel cu alte instrucțiuni concurente sau alte procese. Există trei tipuri ale instrucțiunilor concurente de asignare a semnalelor: instrucțiunea de asignare simplă, instrucțiunea de asignare condițională și instrucțiunea de asignare selectivă. Acestea sunt prezentate în continuare.

1.3.1. Instrucțiunea de asignare simplă

Instrucțiunea de asignare simplă este versiunea concurentă a instrucțiunii secvențiale de asignare a semnalelor, având aceeași formă cu aceasta. Ca și în cazul versiunii secvențiale, asignarea concurentă definește un nou driver pentru semnalul asignat. Deosebirea față de versiunea secvențială este că instrucțiunea concurentă de asignare apare în afara unui proces, în cadrul unei arhitecturi. O instrucțiune concurentă de asignare reprezintă o formă simplificată de scriere a unui proces, fiind echivalentă cu un proces care conține o singură instrucțiune secvențială de asignare.

Descrierea sumatorului de 1 bit din Exemplul 1 poate fi simplificată prin utilizarea instrucțiunilor concurente de asignare, după cum se arată în Exemplul 2.

Exemplul 2

```
library ieee;
use ieee.std_logic_1164.all;
entity add_1 is
   port (a, b, cin: in std_logic;
         s, cout: out std_logic);
end add_1;

architecture concurent of add_1 is
   signal s1, s2, s3, s4: std_logic;
begin
      s1 <= b xor cin;
      s2 <= a and b;
      s3 <= a and cin;
      s4 <= b and cin;
      s  <= a xor s1;
      cout <= s2 or s3 or s4;
end concurent;
```

După cum se observă din exemplul anterior, instrucțiunile concurente de asignare apar direct în cadrul arhitecturii și nu în interiorul unui proces. Ordinea în care sunt scrise instrucțiunile nu are importanță. La simulare toate instrucțiunile se execută în același ciclu de simulare.

Activarea execuției proceselor este determinată de modificarea valorii unui semnal din lista de sensibilitate a acestora sau de întâlnirea unei instrucțiuni `wait`. În cazul instrucțiunilor concurente de asignare, modificarea valorii unuia din semnalele care apar în partea dreaptă a asignării activează execuția asignării, fără să se specifice în mod explicit o listă de sensibilitate. Activarea unei instrucțiuni de asignare este independentă de activarea altor instrucțiuni concurente din cadrul arhitecturii.

Instrucțiunile concurente de asignare se utilizează pentru descrieri de tipul fluxului de date. Prin sinteza acestor instrucțiuni se generează circuite combinaționale.

Observație

- Dacă într-o arhitectură există mai multe asignări concurente la același semnal, vor fi create drivere multiple pentru acel semnal. În asemenea cazuri, trebuie să existe o funcție de rezoluție predefinită sau definită de utilizator pentru tipul semnalului respectiv. Spre deosebire de asignările concurente, dacă într-un proces există mai multe asignări secvențiale la același semnal, va avea efect doar ultima dintre acestea.

1.3.2. Instrucțiunea de asignare condițională

Instrucțiunea de asignare condițională este echivalentă funcțional cu instrucțiunea condițională `if`, având sintaxa următoare:

```
semnal <= [expresie when condiție else ...]
          expresie;
```

Valoarea uneia din expresiile sursă se atribuie semnalului destinație. Expresia atribuită va fi prima a cărei condiție booleană asociată este adevărată. La execuția unei instrucțiuni de asignare condițională, condițiile sunt testate în ordinea în care ele sunt scrise. La întâlnirea primei condiții care se evaluează la valoarea booleană `TRUE`, expresia corespunzătoare acesteia se asignează semnalului destinație. Dacă nici o condiție nu se evaluează la valoarea `TRUE`, semnalului destinație i se asignează ultima expresie, cea a ultimei clauze `else`. Dacă există două sau mai multe condiții care se evaluează la valoarea `TRUE`, va fi luată în considerare doar prima condiție.

Deosebirile dintre instrucțiunea de asignare condițională și instrucțiunea condițională `if` sunt următoarele:

- Instrucțiunea de asignare condițională este o instrucțiune concurentă, deci poate fi utilizată într-o arhitectură, în timp ce instrucțiunea `if` este secvențială, astfel că poate fi utilizată numai în interiorul unui proces.
- Instrucțiunea de asignare condițională poate fi utilizată numai pentru asignarea valorii unor semnale, în timp ce instrucțiunea `if` poate fi utilizată pentru execuția oricărei instrucțiuni secvențiale.

Exemplul 3 definește o entitate și două arhitecturi pentru o poartă SAU EXCLUSIV cu două intrări. Prima arhitectură utilizează o instrucțiune de asignare condițională, iar a doua utilizează o instrucțiune `if` echivalentă.

Exemplul 3

```
library ieee;
use ieee.std_logic_1164.all;
entity xor2 is
   port (a, b: in std_logic;
         x: out std_logic);
end xor2;

architecture arh1_xor2 of xor2 is
begin
   x <= '0' when a = b else
        '1';
```

```
end arh1_xor2;

architecture arh2_xor2 of xor2 is
begin
   process (a, b)
   begin
      if a = b then x <= '0';
               else x <= '1';
      end if;
   end process;
end arh2_xor2;
```

Instrucțiunea de asignare condițională este implementată printr-un multiplexor care selectează una din expresiile sursă. Figura 2 ilustrează circuitul rezultat pentru următoarea instrucțiune:

```
s <= a xor b when c = '1' else
     not (a xor b);
```

Circuitul din figura 2 este cel generat inițial de utilitarul de sinteză, dar operatorul de egalitate va fi minimizat ulterior la o simplă conexiune, astfel încât semnalul `c` să controleze multiplexorul în mod direct.

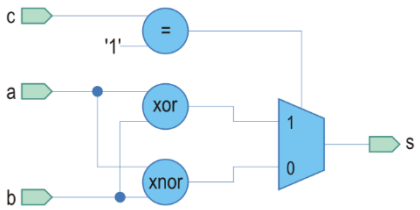

Figura 2. Implementarea unei instrucțiuni de asignare condițională.

Exemplul anterior reprezintă forma cea mai simplă a unei instrucțiuni de asignare condițională, în care există o singură condiție testată. Un alt exemplu în care există două condiții testate este următorul:

```
z <= a when s0 = '1' else
     b when s1 = '1' else
     c;
```

Condițiile sunt evaluate în ordinea în care sunt scrise, fiind selectată pentru asignare prima expresie a cărei condiție este adevărată. Aceasta echivalează din punct de vedere hardware cu o serie de multiplexoare cu două căi, prima condiție controlând multiplexorul cel mai apropiat de ieșire. Circuitul rezultat pentru acest exemplu este prezentat în figura 3. Pentru acest circuit, condițiile au fost deja optimizate, astfel încât semnalele `s0` și `s1` controlează multiplexoarele în mod direct. Din acest circuit se poate observa că atunci când `s0` este '1', este selectat semnalul `a` indiferent de valoarea semnalului `s1`. Dacă `s0` este '0', atunci `s1` selectează între intrările `b` și `c`.

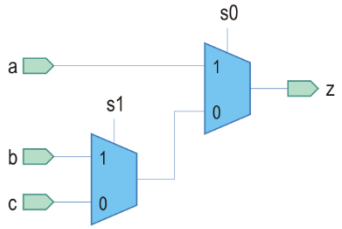

Figura 3. Implementarea unei instrucțiuni de asignare condițională cu două condiții.

Dacă există un număr mare de ramuri ale instrucțiunii, la implementare rezultă un șir lung de multiplexoare. De acest aspect trebuie să se țină cont la proiectare: cu cât o expresie

sursă apare mai târziu în lista de selecție, cu atât semnalele din această expresie vor traversa mai multe multiplexoare în circuitul rezultat la sinteză.

La implementare, se consideră că fiecare condiție dintr-o instrucțiune de asignare condițională este independentă de celelalte. Aceasta înseamnă că, în cazul în care condițiile sunt dependente (de exemplu, se bazează pe același semnal), este posibil să nu se realizeze nici o optimizare. De exemplu:

```
z <= a when sel = '1' else
     b when sel = '0' else
     c;
```

În acest exemplu, a doua condiție este dependentă de prima. De fapt, în a doua ramură, semnalul `sel` poate fi numai '0'. De aceea, a doua condiție este redundantă, iar ultima ramură `else` nu poate fi atinsă. La sinteză, această instrucțiune de asignare condițională va fi implementată totuși prin două multiplexoare, după cum se ilustrează în figura 4.

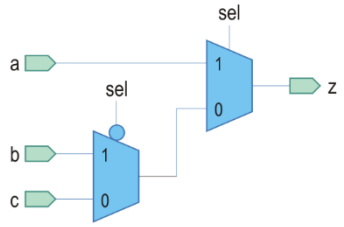

Figura 4. Implementarea unei instrucțiuni de asignare condițională cu o ramură redundantă.

În cazul unui exemplu simplu cum este cel anterior, este probabil ca utilitarul de sinteză să elimine multiplexorul redundant, dar în cazul unor exemple mai complexe această eliminare nu poate fi garantată. Motivul pentru care nu se obține o implementare optimizată este că, în cazul general, detectarea unui cod VHDL la care nu se poate ajunge nu este o problemă trivială.

În cazul în care condițiile sunt dependente unele de altele, este mai avantajoasă utilizarea unei instrucțiuni de asignare selectivă.

1.3.3. Instrucțiunea de asignare selectivă

Ca și instrucțiunea de asignare condițională, instrucțiunea de asignare selectivă permite selectarea unei expresii sursă pe baza unei condiții. Deosebirea constă în faptul că instrucțiunea de asignare selectivă utilizează o singură condiție pentru selecția dintre diferite opțiuni. Această instrucțiune este echivalentă funcțional cu instrucțiunea secvențială `case`. Sintaxa este următoarea:

```
with expresie_de_selecție select
   semnal <= expresie_1 when opțiuni_1,
             ...
             expresie_n when opțiuni_n,
             [expresie when others];
```

Semnalului destinație i se atribuie valoarea uneia din expresii. Expresia selectată este prima dintre cele ale căror opțiuni includ valoarea expresiei de selecție. Sintaxa opțiunilor este aceeași ca și în cazul instrucțiunii `case`. Astfel, fiecare opțiune poate fi reprezentată de o valoare individuală sau de un set de valori. În cazul în care o opțiune este reprezentată de un set de valori, se pot specifica fie valorile individuale din set separate prin simbolul "|", fie domeniul valorilor, fie o combinație a acestora. Tipul expresiei de selecție determină tipul fiecărei opțiuni.

Fiecare valoare din domeniul expresiei de selecție trebuie să fie acoperită de o opțiune. Ultima opțiune poate fi indicată prin cuvântul cheie `others`, care specifică toate valorile din domeniul expresiei de selecție rămase neacoperite de opțiunile anterioare.

Există următoarele restricții pentru diferitele opțiuni:

- Valorile din cadrul opțiunilor nu se pot suprapune.
- Dacă opțiunea `others` nu este prezentă, toate valorile posibile ale expresiei de selecție trebuie acoperite de setul de opțiuni.

Observație

- Opțiunile din cadrul instrucțiunii de asignare selectivă sunt separate prin virgule.

În Exemplul 4 se reia definiția porții SAU EXCLUSIV cu două intrări, dar în cadrul arhitecturii se utilizează o instrucțiune de asignare selectivă.

Exemplul 4

```
library ieee;
use ieee.std_logic_1164.all;
entity xor2 is
   port (a, b: in std_logic;
         x: out std_logic);
end xor2;

architecture arh_xor2 of xor2 is
   signal tmp: std_logic_vector (1 downto 0);
begin
   tmp <= a & b;
   with tmp select
      x <= '0' when "00",
           '1' when "01",
           '1' when "10",
           '0' when others;
end arh_xor2;
```

1.4. Instrucțiunea `block`

O instrucțiune `block` definește un grup de instrucțiuni concurente. Această instrucțiune este utilă pentru organizarea instrucțiunilor concurente în mod ierarhic sau pentru partiționarea unei liste de conexiuni structurale în scopul creșterii lizibilității descrierii. Sintaxa instrucțiunii `block` este următoarea:

```
etichetă: block [(expresie_de_gardă)]
   [declarații]
begin
   instrucțiuni_concurente
end block [etichetă];
```

Eticheta obligatorie denumește blocul. În partea de declarații se pot declara obiecte locale blocului. Declarațiile posibile sunt cele care pot apare și în partea declarativă a unei arhitecturi, și anume:

- Clauze `use`;
- Declarații de porturi și generice, ca și declarații pentru maparea acestora;
- Declarații și corpuri de subprograme;
- Declarații de tipuri și subtipuri;
- Declarații de constante, variabile și semnale;
- Declarații de componente;
- Declarații de fișiere, atribute și configurații.

Ordinea instrucțiunilor concurente dintr-un bloc nu este semnificativă, deoarece toate instrucțiunile sunt întotdeauna active. Într-un bloc pot fi declarate alte blocuri, pe mai multe nivele ierarhice. Obiectele declarate într-un bloc sunt vizibile în acel bloc și în toate blocurile interioare. Atunci când într-un bloc interior se declară un obiect cu același nume ca și un obiect dintr-un bloc exterior, este valabilă declarația din blocul interior.

Exemplul 5 ilustrează utilizarea blocurilor pe mai multe nivele ierarhice.

Exemplul 5

```
B1: block
   signal s: std_logic;          -- declaratia "s" in blocul B1
begin
   s <= a and b;                 -- "s" din blocul B1
   B2: block
      signal s: std_logic;       -- declaratia "s" in blocul B2
   begin
      s <= c and d;              -- "s" din blocul B2
      B3: block
      begin
         x <= s;                 -- "s" din blocul B2
      end block B3;
   end block B2;
   y <= s;                       -- "s" din blocul B1
end block B1;
```

Introducerea blocurilor în cadrul unei descrieri nu afectează execuția unui model la simulare, ci are doar rol de organizare a descrierii.

La declararea unui bloc se poate specifica o expresie booleană, numită *expresie de gardă*. Această expresie, specificată în paranteze după cuvântul cheie `block`, creează în mod implicit un semnal boolean numit `guard`, care se poate utiliza pentru controlul unor operații din cadrul blocului. Acest semnal poate fi citit ca orice alt semnal din cadrul instrucțiunii `block`, dar nu poate fi actualizat printr-o instrucțiune de asignare. De fiecare dată când apare o tranzacție asupra unuia din semnalele dintr-o expresie de gardă, expresia este evaluată și semnalul `guard` este actualizat imediat. Acest semnal ia valoarea `TRUE` dacă valoarea expresiei de gardă este adevărată și `FALSE` în caz contrar.

Semnalul `guard` poate fi declarat și în mod explicit ca un semnal de tip `boolean` în cadrul instrucțiunii `block`. Avantajul acestei declarări explicite este că se poate utiliza un algoritm mai complex pentru controlul semnalului `guard` decât cel permis de o expresie booleană. În particular, se poate utiliza un proces separat pentru controlul acestui semnal.

Dacă într-un bloc nu se specifică o expresie de gardă și semnalul `guard` nu este declarat în mod explicit, atunci acest semnal are întotdeauna valoarea `TRUE`.

Semnalul `guard` poate fi utilizat pentru controlul instrucțiunilor de asignare a semnalelor din cadrul blocului. O asemenea instrucțiune de asignare conține cuvântul cheie `guarded` după simbolul de asignare, care determină ca execuția instrucțiunii de asignare să fie condițională:

```
semnal <= guarded expresie;
```

Această asignare se execută numai dacă semnalul `guard` al blocului care conține expresia de gardă are valoarea `TRUE`.

În Exemplul 6, semnalului `out1` i se va asigna valoarea `not in1` numai dacă valoarea expresiei `clk'event and clk = '1'` va fi adevărată.

Exemplul 6

```
front_crescator: block (clk'event and clk = '1')
begin
   out1 <= guarded not in1 after 5 ns;
   ...
end block front_crescator;
```

În Exemplul 7, semnalul `guard` este declarat in mod explicit, astfel încât i se poate asigna o valoare ca și oricărui alt semnal.

Exemplul 7

```
UAL: block
   signal guard: boolean := FALSE;
begin
```

```
   out1 <= guarded not in1 after 5 ns;
   ...
   p1: process
   begin
      guard <= TRUE;
      ...
   end process p1;
end block UAL;
```

Observații

- În general, utilitarele de sinteză nu permit utilizarea blocurilor cu expresii de gardă. Un asemenea bloc este echivalent cu un proces cu o listă de sensibilitate care conține instrucțiuni condiționale. Se poate utiliza un asemenea proces în locul unui bloc cu o expresie de gardă.
- De obicei, blocurile simple sunt ignorate de utilitarele de sinteză.
- Deși blocurile se pot utiliza pentru partiționarea descrierilor, limbajul VHDL permite utilizarea unui mecanism mai puternic pentru partiționare, și anume instanțierea componentelor.

2. Descrierea unor circuite combinaționale

2.1. Multiplexoare

Pentru descrierea multiplexoarelor se pot utiliza diferite metode. Exemplul 8 prezintă descrierea multiplexorului 4:1 pentru magistrale de 4 biți din figura 5 utilizând o instrucțiune de asignare selectivă.

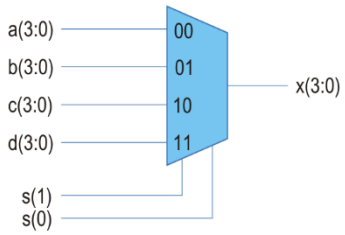

Figura 5. Schema unui multiplexor 4:1 pentru magistrale de 4 biți.

Exemplul 8

```
library ieee;
use ieee.std_logic_1164.all;
entity mux is
   port (a, b, c, d: in  std_logic_vector (3 downto 0);
         s:          in  std_logic_vector (1 downto 0);
         x:          out std_logic_vector (3 downto 0));
end mux;

architecture arh_mux of mux is
begin
   with s select
      x <= a when "00",
           b when "01",
           c when "10",
           d when "11",
           d when others;
end arh_mux;
```

Motivul pentru care se utilizează cuvântul cheie `others` este că semnalul de selecție `s` este de tip `std_logic_vector` și există nouă valori posibile ale unui obiect de acest tip. Toate valorile posibile ale semnalului de selecție trebuie acoperite. În cazul în care nu s-ar fi

utilizat opțiunea `others`, doar patru din cele 81 de valori posibile ar fi acoperite de setul de opțiuni. Alte valori posibile ale semnalului `s` sunt, de exemplu, `"1X"`, `"UX"`, `"Z0"`, `"U-"`. Pentru sinteză `"11"` este singura valoare utilă, însă pentru simulare semnalul `s` poate avea alte 77 de valori posibile. Se poate utiliza și valoarea metalogică `"--"` pentru asignarea unei valori indiferente semnalului `x`.

Multiplexorul 4:1 poate fi descris cu ajutorul unei instrucțiuni `if` în modul indicat în Exemplul 9.

Exemplul 9

```
architecture arh_mux of mux is
begin
mux4_1: process (a, b, c, d, s)
   begin
      if s = "00" then
         x <= a;
      elsif s = "01" then
         x <= b;
      elsif s = "10" then
         x <= c;
      else
         x <= d;
      end if;
   end process mux4_1;
end arh_mux;
```

Deoarece condițiile implică valori mutual exclusive ale semnalului `s`, prin sinteza acestei descrieri se generează același circuit ca și în cazul utilizării unei instrucțiuni de asignare selectivă. Însă, deoarece condițiile conțin o prioritate, instrucțiunea `if` nu este avantajoasă atunci când condițiile implică semnale multiple care sunt mutual exclusive. Utilizarea unei instrucțiuni `if` în aceste cazuri poate determina generarea unei logici suplimentare pentru a asigura faptul că precedentele condiții nu sunt adevărate. În locul unei instrucțiuni `if`, este mai avantajoasă utilizarea unei ecuații booleene sau a unei instrucțiuni `case`.

2.2. Decodificatoare

Un decodificator este un circuit combinațional care identifică un cod de intrare prin activarea unei singure linii de ieșire, corespunzătoare codului de intrare. Un decodificator cu n linii de intrare are, în general, 2^n linii de ieșire și se notează cu DCD n:2^n.

Exemplul 10 descrie un decodificator 3:8 pentru care liniile de ieșire sunt active în starea 1 logic. Pentru descriere se utilizează o instrucțiune de asignare condițională.

Exemplul 10

```
library ieee;
use ieee.std_logic_1164.all;
entity decodif_3_8 is
   port (a: in std_logic_vector (2 downto 0);
         y: out std_logic_vector (7 downto 0));
end decodif_3_8;

architecture decod of decodif_3_8 is
begin
   y <= "00000001" when a = "000" else
        "00000010" when a = "001" else
        "00000100" when a = "010" else
        "00001000" when a = "011" else
        "00010000" when a = "100" else
        "00100000" when a = "101" else
        "01000000" when a = "110" else
        "10000000";
end decod;
```

Atunci când se utilizează programul de sinteză XST, pentru a se genera un decodificator din descrierea HDL trebuie să se specifice toate combinațiile intrărilor și trebuie utilizate toate ieșirile (de exemplu, nu trebuie specificate valori 'X' pentru liniile de ieșire).

2.3. Codificatoare prioritare

Un exemplu de codificator prioritar este prezentat în figura 6.

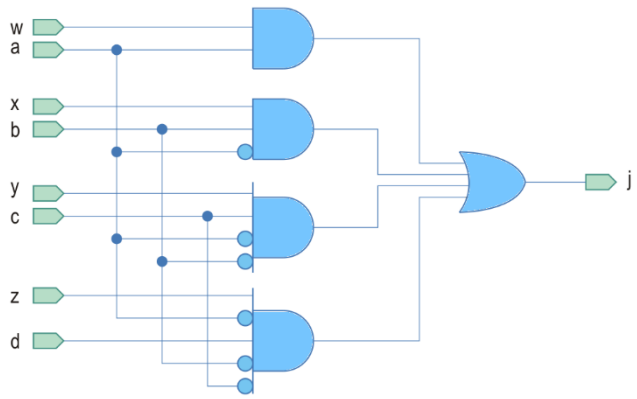

Figura 6. Schema unui codificator prioritar.

Acest codificator prioritar poate fi descris în mod concis cu ajutorul unei instrucțiuni de asignare condițională, ca în Exemplul 11.

Exemplul 11

```
library ieee;
use ieee.std_logic_1164.all;
entity codif_prioritar is
   port (a, b, c, d: in std_logic;
         w, x, y, z: in std_logic;
         j: out std_logic);
end codif_prioritar;

architecture prioritar of codif_prioritar is
begin
   j <= w when a = '1' else
        x when b = '1' else
        y when c = '1' else
        z when d = '1' else
        '0';
end prioritar;
```

Instrucțiunea `when-else` din exemplul precedent indică faptul că semnalului `j` i se asignează valoarea semnalului `w` atunci când `a` este '1', chiar dacă `b`, `c` sau `d` sunt '1'. Semnalul `b` este prioritar față de semnalele `c` și `d`, iar semnalul `c` este prioritar față de semnalul `d`. Dacă semnalele `a`, `b`, `c` și `d` sunt mutual exclusive (deci, se cunoaște că numai unul din acestea va fi activ la un moment dat), atunci este mai avantajoasă descrierea din Exemplul 12.

Exemplul 12

```
library ieee;
use ieee.std_logic_1164.all;
entity fara_prioritate is
   port (a, b, c, d: in std_logic;
         w, x, y, z: in std_logic;
         j: out std_logic);
end fara_prioritate;

architecture fara_prioritate of fara_prioritate is
begin
```

```
   j <= (a and w) or (b and x) or (c and y) or (d and z);
end fara_prioritate;
```

Logica generată prin sinteza descrierii din Exemplul 12 necesită porți ȘI cu doar două intrări. Deși în cazul circuitelor CPLD porțile ȘI cu un număr mai mare de intrări nu necesită, de obicei, resurse suplimentare, în cazul circuitelor FPGA aceste porți pot necesita celule logice și nivele logice suplimentare. Descrierile din Exemplul 11 și Exemplul 12 nu sunt echivalente funcțional, această echivalență existând doar în cazul în care semnalele `a`, `b`, `c` și `d` sunt mutual exclusive. În acest caz, descrierea din Exemplul 12 generează o logică echivalentă cu un număr mai redus de resurse.

2.4. Circuite combinaționale de deplasare

Un circuit combinațional de deplasare realizează o operație de deplasare logică sau aritmetică asupra datelor de intrare. Intrările circuitului de deplasare sunt datele care trebuie deplasate și selectorul a cărui valoare binară specifică distanța de deplasare. Ieșirea circuitului de deplasare este rezultatul operației de deplasare.

Atunci când se utilizează programul de sinteză XST, există următoarele restricții pentru a se genera un circuit combinațional de deplasare din descrierea HDL:

- Se pot utiliza numai operatori de deplasare logică (`sll`, `srl`), deplasare aritmetică (`sla`, `sra`), rotire (`rol`, `ror`) și concatenare (&). Operațiile de deplasare care completează pozițiile eliberate cu valori dintr-un alt semnal nu sunt recunoscute.
- Pentru un circuit de deplasare se poate utiliza un singur tip de operație de deplasare.
- Valoarea care specifică distanța de deplasare în operația de deplasare trebuie să fie pozitivă și trebuie incrementată sau decrementată numai cu 1 pentru fiecare valoare binară consecutivă a selectorului.
- Trebuie să fie prezente toate valorile selectorului.

Exemplul 13 descrie un circuit combinațional de deplasare pentru vectori de 8 biți care pot fi deplasați la stânga cu una, două sau trei poziții. Pentru descrierea circuitului de deplasare se utilizează o instrucțiune de asignare selectivă.

Exemplul 13

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity depl_stanga is
   port (din:  in unsigned (7 downto 0);
         sel:  in unsigned (1 downto 0);
         dout: out unsigned (7 downto 0));
end depl_stanga;

architecture arch_depl of depl_stanga is
begin
   with sel select
      dout <= din when "00",
              din sll 1 when "01",
              din sll 2 when "10",
              din sll 3 when others;
end arch_depl;
```

3. Descrierea unor circuite secvențiale

3.1. Circuite secvențiale sincrone și asincrone

Circuitele secvențiale reprezintă acea categorie de circuite logice care cuprind elemente de memorare. Efectul de memorare se datorează unor legături inverse (bucle de reacție)

prezente în schemele logice ale acestor circuite. Semnalele generate la ieșirile unui circuit secvențial depind atât de semnalele de intrare, cât și de starea circuitului.

Starea prezentă a circuitului este determinată de o stare anterioară și de valorile semnalelor de intrare. În cazul circuitelor secvențiale sincrone, modificarea stării se poate realiza la momente bine definite de timp sub controlul unui semnal de ceas. În cazul circuitelor secvențiale asincrone, modificarea stării poate fi cauzată de schimbarea aleatoare în timp a valorii unui semnal de intrare. Comportamentul unui circuit asincron este mai puțin sigur, evoluția stării fiind influențată și de timpii de întârziere ai componentelor circuitului. Trecerea între două stări stabile se poate realiza printr-o succesiune de stări instabile, aleatoare.

Circuitele secvențiale sincrone sunt mai fiabile și au un comportament predictiv. Toate elementele de memorare ale unui circuit sincron își modifică simultan starea, ceea ce elimină apariția unor stări intermediare instabile. Prin testarea semnalelor de intrare la momente bine definite de timp se reduce influența întârzierilor și a eventualelor zgomote.

Există două tehnici de proiectare a circuitelor secvențiale: *Mealy* și *Moore*. În cazul circuitelor secvențiale Mealy, semnalele de ieșire depind atât de starea curentă, cât și de intrările prezente. În cazul circuitelor secvențiale Moore, ieșirile sunt dependente numai de starea curentă, fără să depindă în mod direct de intrări. Metoda Mealy permite implementarea unui anumit circuit printr-un număr minim de elemente de memorare (bistabile), însă eventualele variații necontrolate ale semnalelor de intrare se pot transmite semnalelor de ieșire. Proiectarea prin metoda Moore necesită mai multe elemente de memorare pentru același tip de comportament, dar funcționarea circuitului este mai sigură.

3.2. Bistabile

Exemplul 14 descrie un bistabil sincron de tip D acționat pe frontul crescător al semnalului de ceas (figura 7).

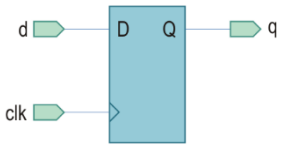

Figura 7. Simbolul unui bistabil de tip D.

Exemplul 14

```
library ieee;
use ieee.std_logic_1164.all;
entity bist_d is
   port (clk: in std_logic;
         d:   in std_logic;
         q:   out std_logic);
end bist_d;

architecture exemplu of bist_d is
begin
   process (clk)
   begin
      if (clk'event and clk = '1') then
         q <= d;
      end if;
   end process;
end exemplu;
```

Procesul utilizat pentru descrierea bistabilului este sensibil numai la modificările semnalului de ceas `clk`. Tranziția semnalului de intrare `d` nu determină activarea procesului. Expresia `clk'event` și lista de sensibilitate sunt redundante, deoarece ambele detectează modificarea semnalului de ceas. Unele utilitare de sinteză ignoră însă lista de sensibilitate a procesului, motiv pentru care trebuie inclusă expresia `clk'event` pentru descrierea evenimentelor acționate pe frontul semnalului de ceas.

Pentru descrierea unui circuit *latch* acționat pe nivel (figura 8), se elimină condiția `clk'event` și se inserează intrarea de date `d` în lista de sensibilitate a procesului, după cum se arată în Exemplul 15.

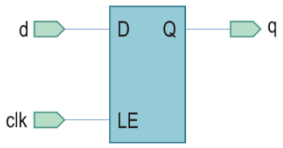

Figura 8. Simbolul unui circuit *latch* de tip D.

Exemplul 15

```
architecture exemplu of latch_d is
begin
   process (clk, d)
   begin
      if (clk = '1') then
         q <= d;
      end if;
   end process;
end exemplu;
```

În exemplele 14 și 15 nu există o condiție `else`. Fără această condiție, este specificat în mod implicit un element de memorie (care va păstra valoarea semnalului `q`). Cu alte cuvinte, următorul fragment:

```
if (clk'event and clk = '1') then
   q <= d;
end if;
```

are aceeași semnificație pentru simulare ca și fragmentul:

```
if (clk'event and clk = '1') then
   q <= d;
else
   q <= q;
end if;
```

Aceasta este în concordanță cu modul în care funcționează un bistabil de tip D. Cele mai multe utilitare de sinteză nu permit utilizarea unei expresii `else` după condiția `if (clk'event and clk = '1')`, deoarece implementarea unei asemenea descrieri poate fi ambiguă.

3.3. Registre

Exemplul 16 descrie un registru de 8 biți printr-un proces similar celui din Exemplul 14, cu deosebirea că `d` și `q` sunt vectori. În plus, acest registru are un semnal de validare a ceasului (`ce`).

Exemplul 16

```
library ieee;
use ieee.std_logic_1164.all;
entity reg8 is
   port (clk: in std_logic;
         ce:  in std_logic;
         d:   in std_logic_vector (7 downto 0);
         q:   out std_logic_vector (7 downto 0));
end reg8;

architecture ex_reg of reg8 is
begin
   process (clk)
   begin
```

```
      if (clk'event and clk = '1') then
         if (ce = '1') then
            q <= d;
         end if;
      end if;
   end process;
end ex_reg;
```

3.4. Registre de deplasare

Un registru de deplasare este un circuit secvențial care deplasează la stânga sau la dreapta conținutul registrului cu o poziție în fiecare ciclu de ceas. De obicei, intrările unui registru de deplasare sunt reprezentate de semnalul de ceas, o intrare serială de date, un semnal de setare/resetare sincronă sau asincronă și un semnal de validare a ceasului. În plus, un registru de deplasare poate avea semnale de control și de date pentru încărcarea paralelă sincronă sau asincronă. Datele de ieșire ale unui registru de deplasare pot fi accesate fie serial, atunci când este accesibil numai conținutul ultimului bistabil pentru restul circuitului, fie în paralel, atunci când este accesibil conținutul mai multor bistabile.

Circuitele FPGA Xilinx conțin resurse dedicate (primitivele SRL16 și SRL32) care permit o implementare eficientă a registrelor de deplasare fără utilizarea unor bistabile suplimentare. Totuși, aceste resurse permit numai operații de deplasare la stânga și au un număr limitat de semnale de intrare/ieșire: ceas, validarea ceasului, intrare serială de date și ieșire serială de date. În primitivele SRL nu sunt disponibile semnale de setare/resetare sincronă sau asincronă. De aceea, dacă în descriere se utilizează orice logică de setare, resetare sau de încărcare paralelă, este posibil ca utilitarul de sinteză XST să nu poată beneficia de avantajul primitivelor dedicate pentru o implementare eficientă.

Există mai multe posibilități pentru descrierea registrelor de deplasare în limbajul VHDL:

- Utilizarea operatorului de concatenare:

  ```
  reg <= reg (6 downto 0) & si;
  ```

- Utilizarea construcțiilor `for loop`;
- Utilizarea operatorilor de deplasare predefiniți (`sll`, `srl`, `sla`, `sra`).

Exemplul 17 descrie un registru de deplasare la stânga de 8 biți cu semnale de validare a ceasului, intrare serială și ieșire serială. Pentru descrierea registrului de deplasare se utilizează o construcție `for loop`.

Example 17

```
library ieee;
use ieee.std_logic_1164.all;
entity reg8_depl is
   port (clk: in std_logic;
         ce:  in std_logic;
         si:  in std_logic;
         so:  out std_logic);
end reg8_depl;

architecture reg_depl of reg8_depl is
   signal tmp: std_logic_vector (7 downto 0);
begin
   process (clk)
   begin
      if (clk'event and clk = '1') then
         if (ce = '1') then
            for i in 0 to 6 loop
               tmp(i+1) <= tmp(i);
            end loop;
            tmp(0) <= si;
         end if;
```

```
      end if;
   end process;
   so <= tmp(7);
end reg_depl;
```

3.5. Numărătoare

Exemplul 18 descrie un numărător de 3 biți.

Exemplul 18

```
library ieee;
use ieee.std_logic_1164.all;
entity num3 is
   port (clk: in std_logic;
         num: out integer range 0 to 7);
end num3;

architecture num3_integer of num3 is
   signal tmp: integer range 0 to 7;
begin
   cnt: process (clk)
   begin
      if (clk'event and clk = '1') then
         tmp <= tmp + 1;
      end if;
   end process cnt;
   num <= tmp;
end num3_integer;
```

În exemplul anterior, pentru semnalul `num`, care este de tip `integer`, se utilizează operatorul de adunare. Majoritatea utilitarelor de sinteză permit această utilizare, convertind tipul `integer` la tipul `bit_vector` sau `std_logic_vector`. Utilizarea tipului `integer` pentru porturi pune însă unele probleme:

1) Pentru a utiliza valoarea `num` într-o altă porțiune a proiectului pentru care interfața are porturi de tip `std_logic`, trebuie efectuată o conversie de tip.
2) Vectorii aplicați în timpul simulării codului sursă nu pot fi utilizați pentru simularea modelului generat în urma sintezei. Pentru codul sursă, vectorii trebuie să fie valori întregi. Modelul generat în urma sintezei necesită vectori de tip `std_logic`.

Deoarece operatorul nativ + al limbajului VHDL nu este definit pentru tipurile `bit` sau `std_logic`, acest operator trebuie redefinit înainte de adunarea operanzilor care au aceste tipuri. Standardul IEEE 1076.3 definește funcții pentru redefinirea operatorului + pentru următoarele perechi de operanzi: (`unsigned`, `unsigned`), (`unsigned`, `integer`), (`signed`, `signed`) și (`signed`, `integer`). Aceste funcții sunt definite în pachetul `numeric_std` al standardului 1076.3.

Exemplul 19 reprezintă Exemplul 18 modificat pentru a se utiliza tipul `unsigned` pentru ieșirea numărătorului.

Exemplul 19

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity num3 is
   port (clk: in std_logic;
         num: out unsigned (2 downto 0));
end num3;

architecture num3_unsigned of num3 is
   signal tmp: unsigned (2 downto 0);
begin
   cnt: process (clk)
```

```
   begin
      if (clk'event and clk = '1') then
         tmp <= tmp + 1;
      end if;
   end process cnt;
   num <= tmp;
end num3_unsigned;
```

De obicei, utilitarele de sinteză pun la dispoziție pachete suplimentare care redefinesc operatorii pentru tipul `std_logic`. Deși acestea nu sunt pachete standard, ele se utilizează adesea de către proiectanți, deoarece permit operații aritmetice și relaționale cu tipul `std_logic`, din acest punct de vedere fiind chiar mai utile decât pachetul `numeric_std`. De asemenea, aceste pachete nu necesită utilizarea a două tipuri suplimentare (`signed`, `unsigned`) în plus față de tipul `std_logic_vector` și nici a funcțiilor necesare conversiei între aceste tipuri. La utilizarea unuia din aceste pachete pentru operațiile aritmetice, utilitarul de sinteză va utiliza pentru tipul `std_logic_vector` o reprezentare fără semn sau una cu semn (în complement față de 2) și va genera componentele aritmetice corespunzătoare.

Exemplul 20 prezintă descrierea modificată a numărătorului din exemplele precedente pentru a se utiliza pachetul `std_logic_unsigned` și tipul `std_logic_vector` pentru ieșirea numărătorului.

Exemplul 20

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity num3 is
   port (clk: in std_logic;
         num: out std_logic_vector (2 downto 0));
end num3;

architecture num3_std_logic of num3 is
   signal tmp: std_logic_vector (2 downto 0);
begin
   cnt: process (clk)
   begin
      if (clk'event and clk = '1') then
         tmp <= tmp + 1;
      end if;
   end process cnt;
   num <= tmp;
end num3_std_logic;
```

3.5. Resetarea componentelor sincrone

Exemplele anterioare nu fac referire la resetarea componentelor descrise sau la condițiile inițiale. Standardul VHDL nu specifică faptul că un circuit trebuie resetat sau inițializat. Pentru simulare, standardul specifică faptul că, dacă un semnal nu este inițializat explicit, acesta va fi inițializat cu valoarea având atributul `'left` a tipului semnalului respectiv. Pentru ca circuitele reale să fie aduse într-o stare cunoscută la inițializare, trebuie utilizate semnale de resetare și setare (preset).

Figura 9 ilustrează un bistabil de tip D cu un semnal de resetare asincronă. Acest bistabil poate fi descris în modul prezentat în Exemplul 21.

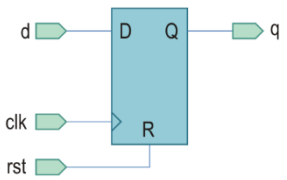

Figura 9. Simbolul unui bistabil de tip D cu un semnal de resetare asincronă.

Exemplul 21

```
architecture exemplu_r of bist_d is      --  1
begin                                    --  2
   process (clk, rst)                    --  3
   begin                                 --  4
      if (rst = '1') then                --  5
         q <= '0';                       --  6
      elsif rising_edge (clk) then       --  7
         q <= d;                         --  8
      end if;                            --  9
   end process;                          -- 10
end exemplu_r;                           -- 11
```

Dacă semnalul `rst` este activat, semnalul `q` va fi setat la '0', indiferent de valoarea semnalului de ceas. Funcția `rising_edge` este definită în pachetul `std_logic_1164`, având rolul de a detecta frontul crescător al unui semnal. Această funcție se poate utiliza în locul expresiei (`clk'event and clk = '1'`), dacă semnalul `clk` este de tip `std_logic`. În același pachet este definită și funcția `falling_edge`, care detectează fronturile descrescătoare ale semnalelor. Aceste funcții sunt preferate de către unii proiectanți deoarece la simulare funcția `rising_edge`, de exemplu, va asigura că tranziția este de la '0' la '1' și nu va ține cont de alte tranziții, cum este cea de la 'U' la '1'.

Pentru a descrie un bistabil cu un semnal de setare asincronă, liniile 5-7 din exemplul anterior se modifică astfel:

```
      if (set = '1') then                --  5
         q <= '1';                       --  6
      elsif rising_edge (clk) then       --  7
```

De obicei, circuitele FPGA au semnale dedicate de resetare sau setare. De exemplu, circuitele FPGA Xilinx au un semnal dedicat de resetare asincronă numit *Global Set/Reset* (GSR). Acest semnal este activat în mod automat la sfârșitul configurării circuitului FPGA. Pentru simularea la nivelul porților logice, în modelul generat pentru simulare este inserat și semnalul GSR pentru a permite simularea cu acuratețe a proiectului inițializat.

Atunci când este disponibil un semnal dedicat de resetare asincronă, utilizarea unui semnal de resetare asincronă într-un proiect nu este recomandat din următoarele motive:

- Semnalul dublează doar semnalul de resetare dedicat;
- Analiza temporală este mai dificilă;
- Circuitul sintetizat de utilitarul de sinteză este mai puțin optim.

Se pot utiliza semnale de resetare (sau de setare) sincrone prin includerea condiției respective în interiorul porțiunii procesului care este sincronă cu ceasul, după cum se indică în Exemplul 22.

Exemplul 22

```
architecture exemplu_r_sinc of bist_d is
begin
   process (clk)
   begin
      if rising_edge (clk) then
         if (rst = '1') then
            q <= '0';
         else
            q <= d;
         end if;
      end if;
   end process;
end exemplu_r_sinc;
```

Execuția procesului din exemplul anterior depinde numai de modificările semnalului de ceas. În urma sintezei se generează un bistabil D care resetat în mod sincron atunci când semnalul `rst` este activ și apare un front crescător al semnalului de ceas. Deoarece bistabilele

circuitelor CPLD nu dispun, de obicei, de intrări de setare sau resetare sincronă, implementarea acestor intrări necesită utilizarea unor resurse logice suplimentare (figura 10).

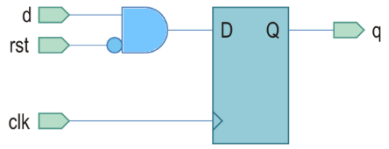

Figura 10. Resurse logice suplimentare necesare pentru un semnal de resetare sincronă.

Se pot utiliza și combinații de semnale sincrone și asincrone de resetare (sau setare). Uneori sunt necesare două semnale asincrone: atât un semnal de resetare, cât și unul de setare. Exemplul 23 prezintă un numărător de 8 biți cu semnale asincrone de resetare și setare.

Exemplul 23

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity num8 is
   port (clk:       in std_logic;
         rst, set: in std_logic;
         en, load: in std_logic;
         data:     in std_logic_vector (7 downto 0);
         num:      out std_logic_vector (7 downto 0));
end num8;

architecture arh_num8 of num8 is
   signal tmp: std_logic_vector (7 downto 0);
begin
   cnt: process (rst, set, clk)
   begin
      if (rst = '1') then
         tmp <= (others => '0');
      elsif (set = '1') then
         tmp <= (others => '1');
      elsif (clk'event and clk = '1') then
         if (load = '1') then
            tmp <= data;
         elsif (en = '1') then
            tmp <= tmp + 1;
         end if;
      end if;
   end process cnt;
   num <= tmp;
end arh_num8;
```

În exemplul anterior, ambele semnale `rst` și `set` sunt utilizate pentru asignarea asincronă a unor valori la registrele numărătorului. Combinația de semnale de resetare și setare din acest exemplu ridică o problemă legată de sinteză. Construcția `if-then-else` utilizată în cadrul procesului implică o precedență – faptul că numărătorului trebuie să i se asigneze valoarea "11111111" numai atunci când semnalul `set` este activ și semnalul `rst` nu este activ. Logica din figura 11 (*a*) asigură această condiție.

Există posibilitatea ca aceasta să nu reprezinte comportarea dorită. Unele utilitare de sinteză pot recunoaște faptul că acesta nu reprezintă efectul dorit și că prin construcția bistabilelor este dominant fie semnalul `rst`, fie semnalul `set`. Astfel, în funcție de algoritmul utilizat de programul de sinteză, descrierea din Exemplul 23 va genera fie logica din figura 11 (*a*), fie cea din figura 11 (*b*). Multe circuite CPLD care permit resetarea sau setarea prin termeni produs pot implementa ambele variante. În timp ce majoritatea circuitelor FPGA permit resetarea sau setarea eficientă prin semnale globale, de obicei acestea nu dispun de resurse pentru resetarea sau setarea eficientă prin termeni produs, caz în care implementarea din figura 11 (*b*) este preferată.

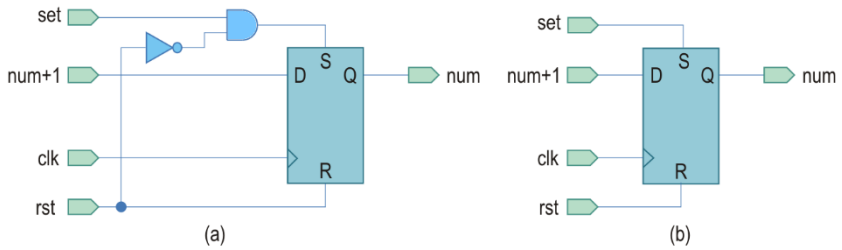

Figura 11. Rezultatul sintezei descrierii din Exemplul 20: (*a*) logica suplimentară asigură ca semnalul `rst` să fie dominant; (*b*) rezultatul dacă se presupune că semnalul `rst` este dominant.

În toate exemplele anterioare în care există semnale de resetare sau setare s-a utilizat instrucțiunea `if` sau funcția `rising_edge` pentru descrierea circuitelor sincrone. Pentru descrierea acestor circuite se poate utiliza și instrucțiunea `wait until`, dar în acest caz semnalele de resetare și setare trebuie să fie sincrone. Aceasta deoarece pentru descrierile destinate sintezei instrucțiunea `wait` trebuie să fie prima din cadrul unui proces, astfel încât toate instrucțiunile care urmează vor descrie o logică sincronă.

3.6. Buffere cu trei stări și semnale bidirecționale

Majoritatea circuitelor programabile dispun de ieșiri cu trei stări sau semnale bidirecționale de I/E. În plus, anumite circuite dispun de buffere interne cu trei stări. Un semnal cu trei stări poate avea valorile `'0'`, `'1'` și `'Z'`, toate acestea fiind permise de tipul `std_logic`.

Exemplul 24 prezintă descrierea modificată a numărătorului din Exemplul 23 pentru a utiliza ieșiri cu trei stări. Acest numărător nu dispune de un semnal de setare asincronă.

Exemplul 24

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity num8 is
   port (clk, rst: in std_logic;
         en, load: in std_logic;
         oe:       in std_logic;
         data:     in std_logic_vector (7 downto 0);
         num:      out std_logic_vector (7 downto 0));
end num8;

architecture arh_num8 of num8 is
   signal tmp: std_logic_vector (7 downto 0);
begin
   cnt: process (rst, clk)
   begin
      if (rst = '1') then
         tmp <= (others => '0');
      elsif rising_edge (clk) then
         if (load = '1') then
            tmp <= data;
         elsif (en = '1') then
            tmp <= tmp + 1;
         end if;
      end if;
   end process cnt;

   oep: process (oe, tmp)
   begin
      if (oe = '0') then
         num <= (others => 'Z');
      else
         num <= tmp;
```

```
      end if;
   end process oep;
end arh_num8;
```

Comparativ cu descrierea din Exemplul 23, în această descriere se utilizează un semnal suplimentar `oe` pentru controlul ieșirilor cu trei stări. Procesul etichetat cu `oep` descrie ieșirile cu trei stări ale numărătorului. Dacă semnalul `oe` nu este activat, ieșirile sunt trecute în starea de înaltă impedanță. Descrierea procesului `oep` este în concordanță cu funcționarea unui buffer cu trei stări (figura 12).

Figura 12. Buffer cu trei stări.

Numărătorul din exemplele precedente poate fi modificat astfel încât pentru ieșirile acestuia să se utilizeze semnale bidirecționale. În acest caz, numărătorul poate fi încărcat cu valoarea curentă a ieșirilor acestuia, ceea ce înseamnă că valoarea încărcată atunci când semnalul `load` este activ va fi valoarea precedentă a numărătorului sau o valoare aplicată din exterior, în funcție de starea semnalului `oe`.

În Exemplul 25, semnalul de validare a ieșirilor unui buffer cu trei stări este definit în mod implicit.

Exemplul 25

```
mux: process (adr_lin, adr_col, stare_prez)
begin
   if (stare_prez = linie or stare_prez = RAS) then
      dram <= adr_lin;
   elsif (stare_prez = coloana or stare_prez = CAS) then
      dram <= adr_col;
   else
      dram <= (others => 'Z');
   end if;
end process mux;
```

Bufferele cu trei stări ale semnalului `dram` sunt validate dacă valoarea semnalului `stare_prez` este `linie`, `RAS`, `coloana` sau `CAS`. Pentru orice altă valoare a acestui semnal, bufferele de ieșire nu sunt validate.

În exemplele anterioare, pentru bufferele cu trei stări s-au utilizat descrieri funcționale. Pentru generarea acestor buffere se pot utiliza și descrieri structurale, cum este construcția `for generate`. Această construcție va fi descrisă în lucrarea de laborator dedicată proiectării structurale.

4. Aplicații

4.1. Modificați următoarea secvență pentru a utiliza o instrucțiune de asignare condițională:

```
process (a, b, j, k)
begin
   if a = '1' and b = '0' then
      pas <= "0100";
   elsif a = '1' then
      pas <= j;
   elsif b = '1' then
      pas <= k;
   else
      pas <= "----";
   end if;
end process;
```

4.2. Transformați următoarea secvență într-o instrucțiune `case`:

```
with stare select
   data <= "0000" when inactiv | terminat,
           "1111" when creste,
           "1010" when mentine,
           "0101" when scade,
           "----" when others;
```

4.3. Transformați următoarea secvență în două instrucțiuni de asignare selectivă:

```
case stare is
   when inactiv => a <= "11"; b <= "00";
   when terminat | creste => a <= "01"; b <= "--";
   when mentine | scade => a <= "10"; b <= "11";
   when others => a <= "11"; b <= "01";
end case;
```

4.4. Rescrieți următoarea secvență utilizând o instrucțiune condițională `if`:

```
iesire <= a when stare = inactiv else
          b when stare = receptie else
          c when stare = transmisie else
          d;
```

4.5. Implementați memoria FIFO pe o placă de dezvoltare, urmărind etapele descrise în documentul Aplicatie-FIFO.pdf.