INSTRUCȚIUNI SECVENȚIALE ÎN LIMBAJUL VHDL

O descriere în limbajul VHDL are două domenii: un domeniu secvențial și un domeniu concurent. Domeniul *secvențial* este reprezentat de un proces sau subprogram care conține instrucțiuni secvențiale. Aceste instrucțiuni sunt executate în ordinea în care apar în cadrul procesului sau subprogramului, ca și în cazul limbajelor de programare. Domeniul *concurent* este reprezentat de o arhitectură, care conține procese, apeluri concurente de proceduri, asignări concurente ale semnalelor și instanțieri de componente. Toate activitățile descrise de acestea au loc simultan.

În această lucrare se descrie formatul și modul de utilizare al unor instrucțiuni secvențiale. Sunt prezentate unele probleme legate de sinteza logică a instrucțiunilor secvențiale și sunt ilustrate circuitele sintetizate din aceste instrucțiuni.

În secțiunile următoare se descriu mai întâi unele aspecte legate de procese, cum sunt modul de specificare al unui proces, execuția unui proces, instrucțiunea `wait` și deosebirea dintre procesele combinaționale și cele secvențiale. Sunt prezentate apoi instrucțiunile secvențiale care pot apare într-un proces sau subprogram: asignarea secvențială a semnalelor, asignarea variabilelor, instrucțiunea `if`, instrucțiunea `case` și instrucțiunile de buclare. Pe lângă acestea, alte instrucțiuni secvențiale sunt instrucțiunea de apel a unei proceduri și instrucțiunea de revenire dintr-o procedură sau funcție.

1. Sintaxa și utilizarea instrucțiunilor secvențiale

1.1. Procese

Un proces este o secvență de instrucțiuni care sunt executate în ordinea specificată. Declarația unui proces delimitează un domeniu secvențial al arhitecturii în care apare declarația. Procesele se utilizează pentru descrieri funcționale.

1.1.1. Structura și execuția unui proces

Un proces poate apare oriunde în corpul unei arhitecturi (partea care începe după cuvântul cheie `begin`). Structura de bază a declarației unui proces este următoarea:

```
[nume:] process [(listă_de_sensibilitate)]
   [declarații_de_tipuri]
   [declarații_de_constante]
   [declarații_de_variabile]
   [declarații_de_subprograme]
begin
   instrucțiuni_secvențiale
end process [nume];
```

Declarația unui proces este cuprinsă între cuvintele cheie `process` și `end process`. Unui proces i se poate asigna în mod opțional un nume pentru identificarea mai ușoară a procesului în textul sursă. Numele este un identificator și trebuie urmat de caracterul ':' (cu rol de etichetă). Acest nume este util și pentru simulare, de exemplu, pentru setarea unui punct de întrerupere a execuției simulării. Numele poate fi repetat la sfârșitul declarației, după cuvintele cheie `end process`.

Lista de sensibilitate (opțională) este lista semnalelor la a căror modificare procesul este “sensibil”. Producerea unui eveniment asupra unuia din semnalele specificate în lista de sensibilitate va determina execuția instrucțiunilor secvențiale din cadrul procesului, similar cu

instrucțiunile dintr-un program obișnuit. Spre deosebire de un limbaj de programare, în limbajul VHDL clauza `end process` nu specifică terminarea execuției procesului. Procesul va fi executat într-un ciclu infinit, iar în cazul în care se specifică o listă de sensibilitate, procesul va fi doar suspendat după execuția ultimei instrucțiuni, până la producerea unui nou eveniment asupra semnalelor din listă. Se menționează că un eveniment are loc numai la modificarea valorii unui semnal. Astfel, asignarea aceleiași valori la un semnal nu reprezintă un eveniment.

În cazul în care lista de sensibilitate lipsește, procesul va fi executat în mod continuu. În acest caz, procesul trebuie să conțină o instrucțiune `wait` pentru a determina suspendarea procesului și activarea acestuia la apariția unui eveniment sau îndeplinirea unei condiții. În cazul în care lista de sensibilitate este prezentă, procesul nu poate conține instrucțiuni `wait`. Instrucțiunea `wait` este prezentată în secțiunea 1.1.2.

Partea declarativă a procesului este cuprinsă între cuvintele cheie `process` și `begin`. În această parte se pot declara tipuri, constante, variabile și subprograme (proceduri și funcții) care sunt locale procesului. Elementele declarate se pot utiliza deci numai în interiorul procesului.

Observație

- În interiorul unui proces nu pot fi declarate semnale, ci numai constante și variabile.

Partea de instrucțiuni a procesului începe după cuvântul cheie `begin`. Această parte conține instrucțiunile care vor fi executate la fiecare activare a procesului. Nu este permisă utilizarea instrucțiunilor concurente în interiorul unui proces.

În Exemplul 1 se prezintă declarația unui proces simplu format dintr-o singură instrucțiune de asignare secvențială.

Exemplul 1

```
proc1: process (a, b, c)
begin
   x <= a and b and c;
end process proc1;
```

1.1.2. Instrucțiunea `wait`

În locul unei liste de sensibilitate, un proces poate conține o instrucțiune `wait`. Utilizarea unei instrucțiuni `wait` are două scopuri:

- Suspendarea execuției unui proces;
- Specificarea condiției care va determina activarea procesului suspendat.

La întâlnirea unei instrucțiuni `wait`, procesul în care apare această instrucțiune este suspendat. Atunci când se îndeplinește condiția specificată în cadrul instrucțiunii `wait`, procesul este activat și se execută instrucțiunile acestuia până când se întâlnește din nou instrucțiunea `wait`. Limbajul VHDL permite ca un proces să conțină mai multe instrucțiuni `wait`. Atunci când se utilizează pentru modelarea logicii combinaționale în vederea sintezei, un proces poate conține însă o singură instrucțiune `wait`.

Dacă un proces conține o instrucțiune `wait`, nu poate conține o listă de sensibilitate. Procesul din Exemplul 2, care conține o instrucțiune `wait` explicită, este echivalent cu procesul din Exemplul 1, care conține o listă de sensibilitate. Ambele procese se vor executa atunci când apare o modificare a valorii semnalelor `a`, `b` sau `c`.

Exemplul 2

```
proc2: process
begin
   x <= a and b and c;
   wait on a, b, c;
end process proc2;
```

Formele instrucțiunii `wait`

Există trei forme ale instrucțiunii `wait`:

```
wait on listă_de_sensibilitate;
wait until expresie_condițională;
wait for expresie_de_timp;
```

Instrucțiunea `wait on` a fost ilustrată în exemplele precedente. Instrucțiunea `wait until` suspendă un proces până când condiția specificată devine adevărată datorită modificării unuia din semnalele care apar în expresia condițională. Se menționează că dacă nici un semnal din această expresie nu se modifică, procesul nu va fi activat, chiar dacă expresia condițională este adevărată. Exemplele următoare prezintă mai multe forme ale instrucțiunii `wait until`:

```
wait until semnal = valoare;
wait until semnal'event and semnal = valoare;
wait until not semnal'stable and semnal = valoare;
```

unde *semnal* este numele unui semnal, iar *valoare* este valoarea care se testează. Dacă semnalul este de tip `bit`, atunci pentru valoarea '1' se așteaptă frontul crescător al semnalului, iar pentru '0' frontul descrescător.

Instrucțiunea `wait until` se poate utiliza pentru implementarea unei funcționări sincrone. În mod obișnuit, semnalul testat este un semnal de ceas. De exemplu, așteptarea frontului crescător al unui semnal de ceas se poate exprima în următoarele moduri:

```
wait until clk = '1';
wait until clk'event and clk = '1';
wait until not clk'stable and clk = '1';
```

Pentru descrierile destinate sintezei, instrucțiunea `wait until` trebuie să fie prima din cadrul procesului. Din această cauză, logica sincronă descrisă cu o instrucțiune `wait until` nu poate fi resetată în mod asincron.

Instrucțiunea `wait for` permite suspendarea execuției unui proces pentru un timp specificat, de exemplu:

```
wait for 10 ns;
```

Observație

- Forma `wait for` *`expresie_de_timp`* a instrucțiunii `wait` nu se poate utiliza pentru sinteză.

Se pot combina mai multe condiții ale instrucțiunii `wait` într-o condiție mixtă. În Exemplul 3, procesul `proc3` va fi activat la modificarea valorii unuia din semnalele `a` sau `b`, dar numai atunci când valoarea semnalului `clk` este '1'.

Exemplul 3

```
proc3: process
begin
   wait on a, b until clk = '1';
   ...
end process proc3;
```

Poziția instrucțiunii `wait`

De obicei, instrucțiunea `wait` apare fie la începutul unui proces, fie la sfârșitul acestuia. În Exemplul 2, instrucțiunea `wait` apare la sfârșitul procesului. Această formă a procesului este echivalentă cu forma care conține o listă de sensibilitate. Această echivalență se datorează modului în care se execută simularea unui model. În faza de inițializare a simulării, se execută toate procesele modelului. Dacă procesul conține o listă de sensibilitate, toate instrucțiunile procesului vor fi executate o singură dată. În forma procesului care conține o instructi-

une wait și aceasta apare la sfârșit, în faza de inițializare se execută o singură dată toate instrucțiunile până la instrucțiunea wait, ca și în cazul formei care conține o listă de sensibilitate.

În cazul în care instrucțiunea wait apare la începutul procesului, simularea se va executa în mod diferit, deoarece în faza de inițializare procesul va fi suspendat fără a se executa nici o instrucțiune a acestuia. Deci, un proces cu instrucțiunea wait plasată la început nu este echivalent cu un proces care conține o listă de sensibilitate. Deoarece faza de inițializare a simulării nu are echivalent hardware, nu vor exista diferențe la sinteza celor două procese cu instrucțiunea wait plasată la sfârșit, respectiv la început.

Totuși, din cauza diferenței între simulare și sinteză în ceea ce privește faza de inițializare a simulării, pot exista diferențe între funcționarea modelului la simulare și funcționarea circuitului rezultat prin sinteză. Este posibil ca modelul care funcționează corect la simulare să fie sintetizat într-un circuit a cărui funcționare este eronată.

1.2. Instrucțiunea secvențială de asignare a semnalelor

Semnalele reprezintă interfața dintre domeniul concurent al unui model în limbajul VHDL și domeniul secvențial din cadrul unui proces. Un model VHDL este format dintr-un număr de procese care comunică între ele prin intermediul semnalelor. La simulare, întreaga ierarhie din cadrul modelului este eliminată, rămânând numai procesele și semnalele. Simulatorul execută în mod alternativ actualizarea valorii unor semnale și rularea proceselor activate de modificarea valorii semnalelor aflate în listele lor de sensibilitate.

Asignarea valorilor la semnale se poate realiza printr-o instrucțiune secvențială sau o instrucțiune concurentă. Instrucțiunea secvențială poate apare doar în interiorul unui proces, iar cea concurentă poate apare doar în exteriorul proceselor. Instrucțiunea secvențială de asignare are o singură formă, cea simplă, care este o asignare necondiționată. Instrucțiunea concurentă are, pe lângă forma sa simplă, încă două forme: asignarea condițională și asignarea selectivă.

Instrucțiunea secvențială de asignare a unui semnal are sintaxa următoare:

```
semnal <= expresie [after întârziere];
```

Ca rezultat al execuției acestei instrucțiuni într-un proces, se evaluează expresia din dreapta simbolului de asignare și se planifică un eveniment care constă din modificarea valorii semnalului. Simulatorul va modifica însă valoarea semnalului doar în momentul în care procesul va fi suspendat, iar în cazul în care se utilizează clauza after, după întârzierea specificată din acest moment. Deci, într-un proces semnalele vor fi actualizate doar după execuția tuturor instrucțiunilor din cadrul procesului sau la întâlnirea unei instrucțiuni wait.

În mod tipic, sistemele de sinteză nu permit utilizarea clauzelor after, sau ignoră aceste clauze. Clauzele after sunt ignorate nu numai deoarece interpretarea lor pentru sinteză nu este specificată de standarde, ci și pentru că ar fi dificilă garantarea rezultatelor unor asemenea întârzieri. De exemplu, nu este clar dacă întârzierea ar trebui interpretată ca o întârziere de propagare minimă sau maximă. De asemenea, nu este clar cum trebuie să procedeze programul de sinteză dacă o întârziere specificată în codul sursă nu poate fi asigurată.

O consecință a modului în care se execută asignarea semnalelor în cadrul proceselor este că în cazul în care există mai multe asignări a unor valori diferite la același semnal, va avea efect doar ultima asignare. Astfel, cele două procese din Exemplul 4 sunt echivalente.

Exemplul 4

```
proc4: process (a)
begin
   z <= '0';
   z <= a;
end process proc4;

proc5: process (a)
begin
```

```
   z <= a;
end process proc5;
```

În concluzie, trebuie să se țină cont de următoarele aspecte importante la utilizarea instrucțiunilor de asignare a semnalelor în interiorul proceselor:

- Orice asignare a unei valori la un semnal devine efectivă numai atunci când procesul este suspendat. Până în acel moment, toate semnalele își păstrează vechea valoare.
- Numai ultima asignare a unei valori la un semnal va fi executată efectiv. Deci, nu are sens să se asigneze mai mult de o valoare la un semnal în același proces.

1.3. Variabile

Restricțiile impuse asupra semnalelor reduc posibilitățile de utilizare ale acestora. Deoarece semnalele pot păstra numai ultima valoare asignată, ele nu pot fi utilizate pentru memorarea rezultatelor intermediare sau temporare în cadrul unui proces. Un alt inconvenient este că noile valori sunt asignate semnalelor nu în momentul execuției instrucțiunii de asignare, ci după suspendarea execuției procesului. Aceasta determină ca analiza unei descrieri să fie dificilă.

Spre deosebire de semnale, variabilele pot fi declarate în interiorul unui proces și se pot utiliza pentru memorarea rezultatelor intermediare. Variabilele pot fi utilizate însă numai în domeniul secvențial al limbajului VHDL, deci în interiorul proceselor sau subprogramelor, și nu pot fi declarate sau utilizate direct într-o arhitectură. Ele sunt deci locale procesului sau subprogramului respectiv.

1.3.1. Declararea și inițializarea variabilelor

Ca și semnalele, variabilele trebuie declarate înainte de a fi utilizate. Declarația unei variabile este similară cu cea a unui semnal, cu deosebirea că se utilizează cuvântul cheie `variable`. Această declarație specifică tipul variabilei. În mod opțional, în cazul variabilelor scalare se poate specifica un domeniu restrâns, iar în cazul variabilelor de tip tablou se poate specifica un index restrâns. Pentru ambele tipuri de variabile, se poate specifica o valoare inițială. Exemplul 5 ilustrează declararea și inițializarea variabilelor.

Exemplul 5

```
variable a, b, c: bit;
variable x, y: integer;
variable index integer range 1 to 10 := 1;
variable t_ciclu: time range 10 ns to 50 ns := 10 ns;
variable mem: bit_vector (0 to 15);
```

Unei variabile i se atribuie o valoare inițială în faza de inițializare a simulării. Această valoare inițială este fie cea specificată în mod explicit la declararea variabilei, fie o valoare implicită, care este valoarea cea mai din stânga a tipului. De exemplu, pentru tipul `bit` valoarea inițială implicită este '0', iar pentru tipul `integer` această valoare este -2.147.483.647.

1.3.2. Instrucțiunea de asignare a variabilelor

O instrucțiune de asignare a unei variabile înlocuiește valoarea curentă a variabilei cu o nouă valoare specificată de o expresie. Expresia poate conține variabile, semnale și literale. Variabila și rezultatul evaluării expresiei trebuie să fie de același tip. Instrucțiunea de asignare a unei variabile are sintaxa următoare:

```
variabilă := expresie;
```

Această instrucțiune este similară cu asignările din majoritatea limbajelor de programare. Spre deosebire de o instrucțiune secvențială de asignare a semnalelor, asignarea unei variabile este executată instantaneu, deci într-un timp de simulare zero.

Există următoarele deosebiri principale între asignarea semnalelor și a variabilelor:

- În cazul asignării semnalelor, se planifică un eveniment pentru actualizarea valorii acestora, iar actualizarea se execută numai la suspendarea procesului, în timp ce pentru asignarea variabilelor nu se planifică un eveniment, iar actualizarea se execută instantaneu.
- La asignarea semnalelor se poate specifica o întârziere (numai pentru simulare), în timp ce asignarea variabilelor nu poate fi întârziată.
- Într-un proces, doar ultima asignare a unei valori la un semnal este efectivă. În schimb, pot exista numeroase asignări la o variabilă în același proces, toate fiind efective.

Exemplul 6 ilustrează utilizarea variabilelor pentru memorarea rezultatelor intermediare.

Exemplul 6

```
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 functional of add_1 is
begin
   process (a, b, cin)
      variable s1, s2, c1, c2: std_logic;
   begin
      s1 := a xor b;
      c1 := a and b;
      s2 := s1 xor cin;
      c2 := s1 and cin;
      s <= s2;
      cout <= c1 or c2;
   end process;
end functional;
```

Exemplul anterior descrie un sumator elementar. Acesta nu este modul optim de a descrie un sumator elementar, dar ilustrează utilizarea variabilelor. Rezultatul se generează prin crearea a două semisumatoare, primul generând ieșirile `s1` și `c1`, iar al doilea generând ieșirile `s2` și `c2`. În final, ieșirile sunt asignate celor două porturi de ieșire `s` și `cout` prin instrucțiuni de asignare a semnalelor, deoarece porturile sunt semnale.

1.4. Instrucțiunea `if`

Instrucțiunea `if` selectează pentru execuție una sau mai multe secvențe de instrucțiuni, în funcție de valoarea unei condiții corespunzătoare secvenței respective. Sintaxa acestei instrucțiuni este următoarea:

```
if condiţie then secvenţă_de_instrucţiuni
   [elsif condiţie then secvenţă_de_instrucţiuni ...]
   [else secvenţă_de_instrucţiuni]
end if;
```

Fiecare condiție este o expresie booleană, care se evaluează la valoarea `TRUE` sau `FALSE`. Pot exista mai multe clauze `elsif`, dar poate exista o singură clauză `else`. Se evaluează mai întâi condiția de după cuvântul cheie `if`, și dacă este adevărată, se execută secvența de instrucțiuni corespunzătoare. În caz contrar, dacă este prezentă clauza `elsif`, se evaluează condiția de după această clauză și dacă această condiție este adevărată, se execută secvența de instrucțiuni corespunzătoare. În caz contrar, dacă sunt prezente alte clauze `elsif`, se continuă cu evaluarea condiției acestor clauze. Dacă nici o condiție evaluată nu este adevărată, se execută secvența de instrucțiuni corespunzătoare clauzei `else`, dacă aceasta este prezentă.

Exemplul 7

```
process (a, b)
begin
   if a = b then
      rez <= 0;
   elsif a < b then
      rez <= -1;
   else
      rez <= 1;
   end if;
end process;
```

1.5. Instrucțiunea case

Ca și instrucțiunea `if`, instrucțiunea `case` selectează pentru execuție una din mai multe secvențe alternative de instrucțiuni pe baza valorii unei expresii. Spre deosebire de instrucțiunea `if`, în cazul instrucțiunii `case` expresia nu trebuie să fie booleană, ci poate fi reprezentată de un semnal, o variabilă sau o expresie de orice tip discret (un tip enumerat sau întreg) sau un tablou de caractere (inclusiv `bit_vector` sau `std_logic_vector`). Instrucțiunea `case` se utilizează atunci când există un număr mare de alternative posibile. Această instrucțiune este mai lizibilă decât o instrucțiune `if` cu un număr mare de ramuri, permițând identificarea ușoară a unei valori și a secvenței de instrucțiuni asociate.

Sintaxa instrucțiunii `case` este următoarea:

```
case expresie is
   when opțiuni_1 =>
      secvență_de_instrucțiuni
   ...
   when opțiuni_n =>
      secvență_de_instrucțiuni
   [when others =>
      secvență_de_instrucțiuni]
end case;
```

Instrucțiunea `case` conține mai multe clauze `when`, fiecare specificând una sau mai multe opțiuni. Opțiunile reprezintă fie o valoare individuală, fie un set de valori cu care se compară expresia instrucțiunii `case`. În cazul în care expresia este egală cu valoarea individuală sau cu una din valorile setului, se execută secvența de instrucțiuni specificată după simbolul =>. O secvență de instrucțiuni poate fi formată și din instrucțiunea nulă, `null`. Spre deosebire de anumite limbaje de programare, instrucțiunile dintr-o secvență nu trebuie incluse între cuvintele cheie `begin` și `end`. Clauza `others` se poate utiliza pentru a specifica execuția unei secvențe de instrucțiuni în cazul în care valoarea expresiei nu este egală cu nici una din valorile specificate în clauzele `when`.

În cazul în care o opțiune este reprezentată de un set de valori, se pot specifica fie valorile individuale din set, separate prin simbolul “|” având semnificația “sau”, fie domeniul valorilor, fie o combinație a acestora, după cum se arată în exemplul următor.

```
case expresie is
   when val =>
      secvență_de_instrucțiuni
   when val1 | val2 | ... | valn =>
      secvență_de_instrucțiuni
   when val3 to val4 =>
      secvență_de_instrucțiuni
   when val5 to val6 | val7 to val8 =>
      secvență_de_instrucțiuni
   ...
   when others =>
      secvență_de_instrucțiuni
end case;
```

Observații

- Într-o instrucțiune `case` trebuie enumerate toate valorile posibile pe care le poate avea expresia de selecție, ținând cont de tipul sau subtipul acesteia. De exemplu, dacă expresia de selecție este de tip `std_logic_vector` de doi biți, trebuie enumerate 81 de valori, deoarece pentru un singur bit pot exista nouă valori posibile. Dacă nu sunt enumerate toate valorile, trebuie utilizată clauza `others`.
- Clauza `others` trebuie să fie ultima dintre toate opțiunile.

Exemplul 8 prezintă un proces pentru secvențierea valorilor unui tip enumerat reprezentând stările unui semafor. Procesul este descris cu ajutorul unei instrucțiuni `case`.

Exemplul 8

```
type tip_culoare is (rosu, galben, verde);
signal culoare, culoare_urm: tip_culoare;
process (culoare)
   case culoare is
      when rosu =>
         culoare_urm <= verde;
      when galben =>
         culoare_urm <= rosu;
      when verde =>
         culoare_urm <= galben;
   end case;
end process;
```

Exemplul 9 definește o entitate și o arhitectură pentru o poartă SAU EXCLUSIV cu două intrări. Poarta este descrisă în mod funcțional cu ajutorul unui proces care conține o instrucțiune `case`.

Exemplul 9

```
library ieee;
use ieee.std_logic_1164.all;
entity xor_2 is
   port (a, b: in std_logic;
         x: out std_logic);
end xor_2;

architecture arh_xor_2 of xor_2 is
begin
   process (a, b)
      variable tmp: std_logic_vector (1 downto 0);
   begin
      tmp := a & b;
      case tmp is
         when "00"    => x <= '0';
         when "01"    => x <= '1';
         when "10"    => x <= '1';
         when "11"    => x <= '0';
         when others => x <= '0';
      end case;
   end process;
end arh_xor_2;
```

1.6. Instrucțiuni de buclare

Instrucțiunile de buclare permit execuția repetată a unei secvențe de instrucțiuni, de exemplu, prelucrarea fiecărui element al unui tablou în același mod. În limbajul VHDL, există trei tipuri de instrucțiuni de buclare: instrucțiunea de buclare simplă `loop`, instrucțiunea `while loop` și instrucțiunea `for loop`. Instrucțiunea de buclare simplă `loop` specifică repetarea unor instrucțiuni în mod nedefinit. În cazul instrucțiunii `while loop`, instrucțiunile

care formează corpul buclei sunt repetate cât timp condiția specificată este adevărată, iar în cazul instrucțiunii `for loop` instrucțiunile din corpul buclei sunt repetate de un număr de ori specificat de un contor.

Observație

- Singura instrucțiune de buclare care poate fi utilizată pentru sinteza logică este instrucțiunea `for loop`, deoarece la aceasta numărul de iterații este fix.

1.6.1. Instrucțiunea loop

Instrucțiunea de buclare simplă `loop` are sintaxa următoare:

```
[etichetă:] loop
   secvență_de_instrucțiuni
end loop [etichetă];
```

Instrucțiunea are o etichetă opțională, care poate fi utilizată pentru identificarea instrucțiunii. Instrucțiunea `loop` are ca efect repetarea instrucțiunilor din corpul buclei de un număr nelimitat de ori. În cazul acestei instrucțiuni, singura posibilitate de terminare a execuției este utilizarea unei instrucțiuni `exit` (prezentată în secțiunea 1.6.5).

1.6.2. Instrucțiunea while loop

Instrucțiunea `while loop` este o instrucțiune de buclare condiționată. Sintaxa acestei instrucțiuni este următoarea:

```
[etichetă:] while condiție loop
   secvență_de_instrucțiuni
end loop [etichetă];
```

Condiția este testată înaintea fiecărei execuții a buclei. Dacă această condiție este adevărată, se execută secvența de instrucțiuni din corpul buclei, după care controlul este transferat la începutul buclei. Execuția buclei se termină atunci când condiția testată devine falsă, caz în care se execută instrucțiunea care urmează după clauza `end loop`.

Exemplul 10

```
process
   variable contor: integer := 0;
begin
   wait until clk = '1';
   while nivel = '1' loop
      contor := contor + 1;
      wait until clk = '0';
   end loop;
end process;
```

În procesul din Exemplul 10 se numără fronturile crescătoare ale semnalului de ceas `clk` atât timp cât semnalul `nivel` are valoarea '1'. Stabilitatea semnalului `nivel` nu este testată. Se testează doar dacă acest semnal are valoarea '1' la detectarea unui front crescător al semnalului `clk`, și nu se testează dacă valoarea semnalului `nivel` s-a modificat de la testarea anterioară.

Instrucțiunile de buclare pot fi imbricate pe mai multe nivele. Deci, corpul buclei unei instrucțiuni `while loop` poate conține o altă instrucțiune de buclare, în particular o instrucțiune `while loop`, după cum se arată în exemplul următor.

```
E1:     while i < 10 loop
E2:        while j < 20 loop
           ...
           end loop E2;
        end loop E1;
```

1.6.3. Instrucțiunea `for loop`

Pentru execuția repetată a unei secvențe de instrucțiuni de un număr de ori specificat se poate utiliza instrucțiunea `for loop`. Sintaxa acestei instrucțiuni este următoarea:

```
[etichetă:] for contor in domeniu loop
   secvență_de_instrucțiuni
end loop [etichetă];
```

Într-o instrucțiune `for loop` se specifică un contor de iterații și un domeniu. Instrucțiunile din corpul buclei sunt executate cât timp contorul se află în domeniul specificat. După terminarea unei iterații, contorului i se asignează următoarea valoare din domeniu. Domeniul poate fi unul crescător, specificat prin cuvântul cheie `to`, sau descrescător, specificat prin cuvântul cheie `downto`. Acest domeniu poate fi specificat și sub forma unui tip enumerat sau subtip, caz în care nu se specifică în mod explicit limitele domeniului în cadrul instrucțiunii `for loop`. Limitele domeniului vor fi determinate de compilator din declarația tipului sau subtipului respectiv.

Instrucțiunea `for loop` din Exemplul 11 calculează pătratele valorilor întregi cuprinse între 1 și 10, pe care le depune în tabloul `i_patrat`.

Exemplul 11

```
for i in 1 to 10 loop
   i_patrat (i) <= i * i;
end loop;
```

Contorul de iterații din acest exemplu este în mod implicit de tip `integer`, deoarece tipul acestuia nu a fost definit în mod explicit. Forma completă a declarației domeniului pentru contorul de iterații este similară cu cea a unui tip. Pentru exemplul anterior, clauza `for` poate fi scrisă și sub forma următoare:

```
for i in integer range 1 to 10 loop
```

În unele limbaje de programare, în cadrul buclei se poate asigna o valoare contorului de iterații (în exemplul anterior, `i`) pentru a-i modifica valoarea. Limbajul VHDL nu permite însă asignarea unei valori contorului de iterații sau utilizarea acestuia ca parametru de intrare sau de ieșire al unei proceduri. Contorul poate fi utilizat însă într-o expresie, cu condiția să nu i se modifice valoarea. Un alt aspect legat de contorul de iterații este că acesta nu trebuie declarat în mod explicit în cadrul procesului. Acest contor este declarat în mod implicit ca o variabilă locală a instrucțiunii de buclare prin specificarea sa după cuvântul cheie `for`. Dacă există o altă variabilă cu același nume în cadrul procesului, cele două vor fi tratate ca variabile separate.

Din cauza interpretării sintezei instrucțiunii `for loop` (descrisă în secțiunea 2.8), limitele domeniului buclei trebuie să fie constante. Aceasta înseamnă că limitele nu pot fi definite utilizând valoarea unei variabile sau semnal. Această restricție determină ca descrierea unor circuite să fie dificilă. De exemplu, considerăm cazul în care trebuie contorizat numărul de zerouri de la începutul unei valori întregi, și deci numărul de iterații necesar nu este cunoscut dinainte. Descrierea unor asemenea circuite cu ajutorul instrucțiunilor de buclare este facilitată prin utilizarea instrucțiunilor `next` și `exit`.

1.6.4. Instrucțiunea `next`

Există situații în care este necesară oprirea execuției instrucțiunilor dintr-o buclă în iterația curentă și trecerea la următoarea iterație. Pentru aceasta se poate utiliza instrucțiunea `next`. Sintaxa acestei instrucțiuni este următoarea:

```
next [etichetă] [when condiție];
```

La întâlnirea unei instrucțiuni `next` în cadrul corpului unei bucle, execuția iterației curente este oprită și controlul este transferat la începutul instrucțiunii de buclare, fie necondi-

ționat, dacă clauza `when` nu este prezentă, fie condiționat, dacă această clauză este prezentă. Contorul de iterații va fi actualizat, iar dacă limita domeniului nu a fost atinsă, execuția va continua cu prima instrucțiune din corpul buclei. În caz contrar, execuția instrucțiunii de buclare se va termina.

În cazul în care există mai multe nivele de instrucțiuni de buclare (o buclă conținută într-o altă buclă), în cadrul instrucțiunii `next` se poate specifica o etichetă, aceasta fiind eticheta instrucțiunii de buclare de nivel imediat superior în care este cuprinsă instrucțiunea `next`. Această etichetă are doar rolul de a crește claritatea descrierii și nu poate fi diferită de cea a instrucțiunii de buclare curente.

O instrucțiune `next` se poate utiliza în locul unei instrucțiuni `if` pentru execuția condiționată a unui grup de instrucțiuni. Utilizarea instrucțiunii `next` este ilustrată în secțiunea 2.9.

1.6.5. Instrucțiunea `exit`

Există situații în care execuția unei instrucțiuni de buclare trebuie oprită complet, fie datorită apariției unei erori în timpul execuției unui model, fie datorită faptului că prelucrarea trebuie terminată înaintea depășirii domeniului de către contorul de iterații. În asemenea situații, se poate utiliza instrucțiunea `exit`. Sintaxa acestei instrucțiuni este următoarea:

```
exit [etichetă] [when condiție];
```

Există trei forme ale instrucțiunii `exit`. Prima este cea în care nu apare o etichetă și nici o condiție specificată printr-o clauză `when`. În acest caz, se va opri în mod necondiționat execuția instrucțiunii curente de buclare. În cazul în care instrucțiunea `exit` apare într-o instrucțiune de buclare aflată în interiorul unei alte instrucțiuni de buclare, va fi oprită doar execuția instrucțiunii interioare de buclare, dar execuția instrucțiunii exterioare de buclare va continua.

Dacă în instrucțiunea `exit` se specifică o etichetă a unei instrucțiuni de buclare, la întâlnirea instrucțiunii `exit` controlul va fi transferat la eticheta specificată.

Dacă instrucțiunea `exit` conține o clauză `when`, execuția instrucțiunii de buclare va fi oprită numai în cazul în care condiția specificată de această clauză este adevărată. Următoarea instrucțiune executată depinde de prezența sau absența unei etichete în cadrul instrucțiunii. Dacă se specifică o etichetă a unei instrucțiuni de buclare, următoarea instrucțiune executată va fi prima din instrucțiunea de buclare specificată de acea etichetă. Dacă nu se specifică o etichetă, următoarea instrucțiune executată va fi cea de după clauza `end loop` a instrucțiunii de buclare curente.

Instrucțiunea `exit` se poate utiliza pentru a termina execuția unei instrucțiuni `loop` simple, după cum se arată în Exemplul 12.

Exemplul 12

```
E3:     loop
           a := a + 1;
           exit E3 when a > 10;
        end loop E3;
```

2. Sinteza instrucțiunilor secvențiale

2.1. Procese cu liste de sensibilitate incomplete

Unele programe utilitare de sinteză nu verifică listele de sensibilitate ale proceselor. Aceste utilitare presupun că toate semnalele din partea dreaptă a asignărilor secvențiale la semnale se află în lista de sensibilitate. Astfel, aceste utilitare vor interpreta cele două procese din Exemplul 13 ca fiind identice.

Exemplul 13

```
proc6: process (a, b, c)
begin
   x <= a and b and c;
end process proc6;

proc7: process (a, b)
begin
   x <= a and b and c;
end process proc7;
```

Toate utilitarele de sinteză vor interpreta procesul `proc6` ca o poartă ŞI cu 3 intrări. Unele utilitare de sinteză vor interpreta procesul `proc7` de asemenea ca o poartă ŞI cu 3 intrări, chiar dacă la simularea acestui cod procesul nu se va comporta ca atare. La simulare, o modificare a valorii semnalului `a` sau `b` va determina execuția procesului, iar valoarea funcției ŞI logic între semnalele `a`, `b` şi `c` se va asigna semnalului `x`. Totuşi, dacă se modifică valoarea semnalului `c`, procesul nu este executat, iar semnalul `x` nu este actualizat.

Deoarece nu este clar modul în care un utilitar de sinteză ar trebui să genereze un circuit pentru care o tranziție a semnalului `c` nu determină o modificare a semnalului `x`, dar o modificare a semnalelor `a` sau `b` determină actualizarea semnalului `x` cu valoarea funcției ŞI logic între semnalele `a`, `b` şi `c`, există următoarele alternative pentru programele de sinteză:

- Interpretarea procesului `proc7` în mod identic cu procesul `proc6` (cu o listă de sensibilitate care cuprinde toate semnalele din partea dreaptă a instrucțiunilor de asignare a semnalelor din cadrul procesului);
- Indicarea unei erori la compilare, specificând faptul că nu se poate realiza sinteza procesului fără o listă de sensibilitate completă.

A doua variantă este preferabilă, deoarece proiectantul va trebui să modifice codul sursă astfel încât funcționarea circuitului generat să fie aceeaşi cu rezultatul simulării funcționale a codului sursă.

Deşi din punct de vedere sintactic procesele pot fi declarate fără o listă de sensibilitate şi fără o instrucțiune `wait`, asemenea procese nu vor fi suspendate niciodată. De aceea, dacă se va simula un asemenea proces, timpul de simulare nu va avansa, deoarece faza de inițializare, în care toate procesele sunt executate până când acestea sunt suspendate, nu se va termina niciodată.

2.2. Procese combinaționale şi procese secvențiale

Atât procesele combinaționale, cât şi procesele secvențiale sunt interpretate în acelaşi fel la sinteză, singura deosebire dintre acestea fiind că la procesele secvențiale semnalele de ieşire se înscriu în bistabile. Un proces combinațional simplu este prezentat în Exemplul 14.

Exemplul 14

```
proc8: process
begin
   wait on a, b;
   z <= a and b;
end process proc8;
```

Acest proces va fi implementat prin sinteză ca o simplă poartă ŞI cu două intrări.

Pentru ca un proces să modeleze un circuit combinațional, acesta trebuie să conțină în lista de sensibilitate toate semnalele care sunt intrări ale procesului. Cu alte cuvinte, procesul trebuie reevaluat de fiecare dată când una din intrările circuitului pe care îl modelează se modifică. În acest fel se modelează în mod corect un circuit combinațional.

Dacă un proces nu este sensibil la toate intrările sale şi nu este un proces secvențial, atunci nu poate fi sintetizat, deoarece nu există un echivalent hardware al unui asemenea proces. Nu toate utilitarele de sinteză impun o asemenea regulă, astfel încât trebuie acordată aten-

ție la scrierea proceselor combinaționale pentru a nu introduce erori. Asemenea erori vor avea ca efect diferențe subtile între modelul simulat și circuitul obținut prin sinteză, deoarece un proces non-combinațional este interpretat de utilitarul de sinteză ca un circuit combinațional.

Dacă un proces conține o instrucțiune `wait until` sau o instrucțiune `if semnal'event`, procesul va fi interpretat ca un proces secvențial. Astfel, procesul din Exemplul 15 va fi interpretat ca un proces secvențial.

Exemplul 15

```
proc9: process
begin
   wait until clk = '1';
   z <= a and b;
end process proc9;
```

Prin sinteza acestui proces, rezultă circuitul din Figura 1, unde s-a adăugat un bistabil la ieșire.

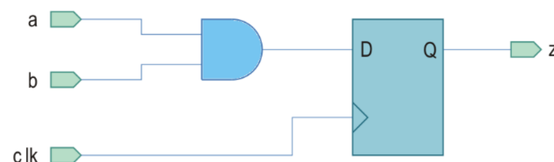

Figura 1. Rezultatul sintezei unui proces secvențial.

2.3. Reacții inverse la asignarea semnalelor

O consecință a modului în care se execută asignarea semnalelor este că prin citirea unui semnal căruia i se asignează o valoare în același proces se va obține valoarea care i-a fost asignată semnalului la execuția precedentă a procesului. Citirea unui semnal și asignarea unei valori la acel semnal în același proces este echivalentă cu o *reacție inversă*. Într-un proces combinațional, valoarea precedentă este o ieșire a logicii combinaționale, astfel încât reacția este asincronă. Într-un proces secvențial, valoarea precedentă este valoarea memorată într-un bistabil sau registru, astfel încât reacția este sincronă.

În Exemplul 16 se prezintă un proces în care se utilizează reacția inversă sincronă. Procesul descrie un numărător de 4 biți, semnalul `num` fiind de tip `unsigned`, tip care este declarat în pachetul `numeric_std`. De observat că semnalul `num` este declarat în afara procesului.

Exemplul 16

```
library ieee;
use ieee.numeric_std.all;

signal num: unsigned (3 downto 0);
process (clk)
begin
   if (clk'event and clk = '1') then
      if rst = '1' then
         num <= "0000";
      else
         num <= num + "0001";
      end if;
   end if;
end process;
```

Circuitul rezultat în urma sintezei procesului din exemplul anterior este prezentat în Figura 2.

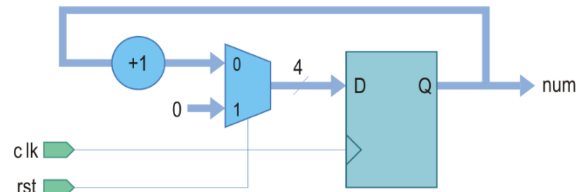

Figura 2. Exemplu de reacție inversă sincronă.

2.4. Sinteza instrucțiunilor `if`

O instrucțiune `if` poate fi implementată printr-un multiplexor. Considerăm mai întâi forma instrucțiunii `if` fără nici o ramură `elsif`. Exemplul 17 prezintă utilizarea unei asemenea instrucțiuni pentru descrierea unui comparator.

Exemplul 17

```
library ieee;
use ieee.std_logic_1164.all;
entity comp is
   port (a, b: in std_logic_vector (7 downto 0);
         egal: out std_logic);
end comp;

architecture functional of comp is
begin
   process (a, b)
   begin
      if a = b then
         egal <= '1';
      else
         egal <= '0';
      end if;
   end process;
end functional;
```

În exemplul anterior, se testează egalitatea a două semnale `std_logic_vector`, reprezentând doi vectori de câte 8 biți, rezultând o valoare de tip `std_logic`. Circuitul rezultat este prezentat în Figura 3. De notat că, la fel ca și în cazul altor exemple, în practică utilitarul de sinteză va elimina ineficiențele din circuit (în acest caz, intrările constante la multiplexor) pentru a rezulta o soluție minimală.

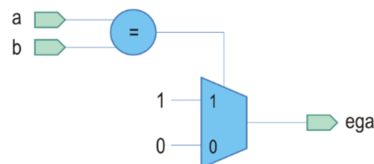

Figura 3. Implementarea unei instrucțiuni `if`.

O instrucțiune `if` cu ramuri multiple, în care apare cel puțin o clauză `elsif`, este implementată prin mai multe etaje de multiplexoare. Considerăm instrucțiunea `if` din Exemplul 18.

Exemplul 18

```
process (a, b, c, s0, s1)
begin
   if s0 = '1' then
      z <= a;
   elsif s1 = '1' then
      z <= b;
```

```
      else
         z <= c;
      end if;
   end process;
```

Rezultatul implementării instrucțiunii `if` din exemplul anterior este prezentat în Figura 4. Acest circuit este echivalent cu cel rezultat prin implementarea unei instrucțiuni de asignare condițională, care este o instrucțiune concurentă.

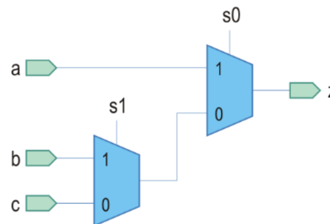

Figura 4. Implementarea unei instrucțiuni `if` cu ramuri multiple.

Condițiile din ramurile succesive ale unei instrucțiuni `if` sunt evaluate independent. În exemplul anterior, condițiile implică cele două semnale `s0` și `s1`. Pot exista oricâte condiții, fiecare din acestea fiind independentă de celelalte. Structura instrucțiunii `if` asigură faptul că primele condiții vor fi testate înaintea celorlalte. În acest exemplu, semnalul `s0` a fost testat înaintea semnalului `s1`. Această prioritate se reflectă în circuitul rezultat, în care multiplexorul controlat de semnalul `s0` este mai apropiat de ieșire decât multiplexorul controlat de semnalul `s1`.

Este important să se rețină existența acestei priorități cu care se testează condițiile, astfel încât să fie eliminate condițiile redundante. Considerăm instrucțiunea `if` din Exemplul 19, care este echivalentă cu instrucțiunea `if` din Exemplul 18.

Exemplul 19

```
process (a, b, c, s0, s1)
begin
   if s0 = '1' then
      z <= a;
   elsif s0 = '0' and s1 = '1' then
      z <= b;
   else
      z <= c;
   end if;
end process;
```

Condiția suplimentară `s0 = '0'` este redundantă, deoarece ea va fi testată numai dacă prima condiție a instrucțiunii `if` este falsă. Se recomandă evitarea unor asemenea redundanțe, deoarece nu există garanția că ele vor fi detectate și eliminate de către utilitarul de sinteză.

În cazul instrucțiunilor `if` cu ramuri multiple, în mod normal fiecare condiție va fi dependentă de semnale și variabile diferite. Dacă fiecare ramură este dependentă de același semnal, atunci este mai avantajos să se utilizeze o instrucțiune `case`.

2.5. Instrucțiuni `if` incomplete

În exemplele prezentate până acum, toate instrucțiunile `if` au fost complete. Cu alte cuvinte, semnalului destinație i s-a asignat o valoare în toate condițiile posibile. Sunt posibile însă două situații în care unui semnal nu i se asignează o valoare: atunci când lipsește clauza `else` a instrucțiunii `if` sau atunci când unui semnal nu i se asignează o valoare într-o ramură a instrucțiunii `if`. În ambele cazuri, interpretarea este aceeași. În situațiile în care unui semnal nu i se asignează o valoare, semnalul își păstrează valoarea precedentă.

Se pune însă problema care este valoarea precedentă a semnalului. Dacă există o instrucțiune anterioară de asignare în care semnalul apare ca destinație, valoarea precedentă

provine de la acea instrucțiune de asignare. În caz contrar, valoarea provine din execuția precedentă a procesului, ceea ce conduce la existența unei reacții inverse în cadrul circuitului.

Primul caz este ilustrat prin Exemplul 20.

Exemplul 20

```
process (a, b, en)
begin
   z <= a;
   if en = '1' then
      z <= b;
   end if;
end process;
```

În acest caz, instrucțiunea `if` este incompletă deoarece lipsește clauza `else`. În instrucțiunea `if`, semnalul `z` primește o valoare în cazul în care condiția `en = '1'` este adevărată, dar rămâne la valoarea precedentă în cazul în care condiția este falsă. Valoarea precedentă provine din instrucțiunea de asignare aflată înaintea instrucțiunii `if`.

Instrucțiunea `if` din Exemplul 20 este echivalentă cu instrucțiunea `if` din Exemplul 21.

Exemplul 21

```
process (a, b, en)
begin
   if en = '1' then
      z <= b;
   else
      z <= a;
   end if;
end process;
```

În cazul în care instrucțiunea `if` este incompletă și nu există o instrucțiune anterioară de asignare, se va insera un circuit *latch* la ieșire și va exista o reacție inversă de la ieșirea circuitului la intrare. Aceasta deoarece valoarea semnalului de la execuția precedentă a procesului este păstrată și aceasta devine valoarea semnalului la execuția curentă a procesului.

O asemenea formă a instrucțiunii `if` este utilizată pentru a descrie un bistabil sau un registru cu o intrare de validare, ca în Exemplul 22.

Exemplul 22

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

Semnalul `q` este actualizat cu noua valoare a semnalului `d` în cazul în care condiția este adevărată, dar nu este actualizat în cazul în care condiția este falsă. În acest caz, valoarea precedentă a semnalului `q` este păstrată prin reacția inversă secvențială a semnalului `q`. Circuitul rezultat este prezentat în Figura 5.

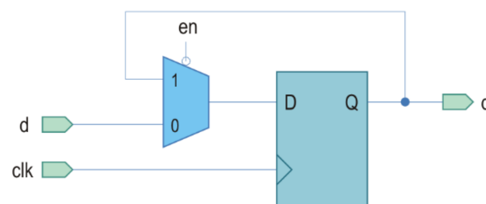

Figura 5. Implementarea unei instrucțiuni `if` incomplete.

Instrucțiunea `if` din Exemplul 22 este echivalentă cu instrucțiunea `if` completă din Exemplul 23.

Exemplul 23

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

În cazul în care condiția este falsă, se asignează semnalului `q` valoarea anterioară a acestuia, ceea ce este echivalent cu păstrarea valorii sale anterioare.

Una din cele mai obișnuite erori întâlnite în descrierile VHDL destinate sintezei logice este introducerea neintenționată a reacției inverse în circuit datorită unei instrucțiuni `if` incomplete. Aceasta va insera circuite *latch* în proiectul sintetizat, ceea ce poate fi problematic pentru circuitele FPGA deoarece întârzierile pentru căile care conțin circuite *latch* sunt dificil de analizat. Utilitarele de sinteză raportează, de obicei, inserarea unui circuit *latch*.

Pentru a se evita inserarea unor circuite *latch*, proiectantul trebuie să se asigure că fiecare semnal căruia i se asignează o valoare într-un proces combinațional (deci, care este un semnal de ieșire din proces) primește o valoare în fiecare combinație posibilă a condițiilor. În practică, există două posibilități pentru aceasta: asignarea unei valori unui semnal de ieșire în fiecare ramură a instrucțiunii `if` și includerea clauzei `else`, sau inițializarea semnalului printr-o instrucțiune de asignare necondiționată înaintea instrucțiunii `if`.

În exemplul următor, deși instrucțiunea `if` pare completă, în cele două ramuri ale instrucțiunii asignările se efectuează la semnale diferite. De aceea, ambele semnale `z` și `y` vor avea o reacție inversă asincronă.

Exemplul 24

```
process (a, b, c)
begin
   if c = '1' then
      z <= a;
   else
      y <= b;
   end if;
end process;
```

Un alt exemplu este cel în care există un test redundant al unei condiții care trebuie să fie adevărată (Exemplul 25).

Exemplul 25

```
process (a, b, c)
begin
   if c = '1' then
      z <= a;
   elsif c = '0' then
      z <= b;
   end if;
end process;
```

În acest caz, deși instrucțiunea `if` pare completă (presupunând că semnalul `c` este de tip `bit`), deoarece pentru fiecare condiție testată sinteza se realizează în mod independent, este posibil ca sistemul de sinteză să nu detecteze faptul că a doua condiție este redundantă. În acest caz, instrucțiunea `if` va fi implementată printr-un multiplexor cu trei căi, a treia intrare

fiind condiția ramurii `else` care lipsește, aceasta fiind reacția inversă a valorii anterioare. Circuitul implementat pentru acest exemplu este prezentat în Figura 6.

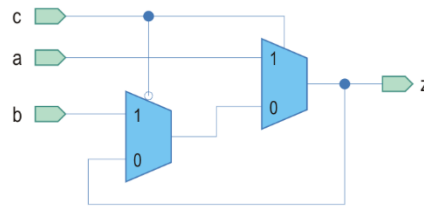

Figura 6. Implementarea unei instrucțiuni `if` cu o condiție redundantă.

2.6. Instrucțiuni `if` în care apar variabile

Până acum, în instrucțiunile `if` au fost utilizate doar semnale. Aceleași reguli se aplică și în cazul utilizării variabilelor, cu o singură diferență. Ca și în cazul unui semnal, dacă unei variabile i se asignează valori numai în anumite ramuri ale instrucțiunii `if`, se păstrează valoarea anterioară a variabilei prin reacție inversă. Spre deosebire de cazul utilizării unui semnal, citirea și scrierea unei variabile în același proces va determina o reacție inversă numai dacă citirea apare înaintea scrierii. În acest caz, valoarea citită este valoarea precedentă a variabilei. În cazul citirii și scrierii în același proces a unui semnal, va rezulta întotdeauna o reacție inversă.

Această observație se poate utiliza pentru a crea registre sau numărătoare utilizând variabile. Reamintim că un proces secvențial este interpretat la sinteză prin plasarea unui bistabil sau registru înaintea ieșirii fiecărui semnal căruia i s-a asignat o valoare în procesul respectiv. Aceasta înseamnă că în mod normal variabilele nu sunt înscrise în bistabile sau registre. Dacă există însă o reacție inversă a valorii precedente a unei variabile, această reacție se realizează printr-un bistabil sau registru pentru ca procesul să fie sincron.

În Exemplul 26 se descrie un numărător utilizând tipul întreg `unsigned`. La incrementarea unei valori de tip `unsigned`, dacă valoarea este cea maximă a domeniului se obține valoarea minimă a domeniului.

Exemplul 26

```
process (clk)
   variable num: unsigned (7 downto 0);
begin
   if (clk'event and clk = '1') then
      if rst = '1' then
         num := "00000000";
      else
         num := num + 1;
      end if;
      rez <= num;
   end if;
end process;
```

În acest exemplu, în ramura `else` a instrucțiunii `if` se citește valoarea precedentă a variabilei `num` pentru a calcula valoarea următoare. Rezultă astfel o reacție inversă.

De notat că în acest exemplu se creează de fapt două registre. Conform regulilor pentru reacția inversă, variabila `num` va fi înscrisă într-un registru. Semnalul `rez` va fi de asemenea înscris într-un registru, deoarece toate semnalele cărora li se asignează o valoare într-un proces secvențial vor fi înscrise în registre. Acest registru suplimentar va conține întotdeauna aceeași valoare ca și registrul variabilei `num`. Utilitarul de sinteză va elimina în mod normal acest registru redundant.

2.7. Sinteza instrucţiunilor `case`

O instrucţiune `case` este implementată printr-un multiplexor, asemănător instrucţiunii `if`. Deosebirea este că toate opţiunile depind de aceeaşi intrare şi sunt mutual exclusive. Aceasta permite anumite optimizări ale condiţiilor de control comparativ cu o instrucţiune `if` cu ramuri multiple, la care nu se va presupune nici o dependenţă între diferitele condiţii. Efectul este că vor fi necesare mai puţine resurse şi întârzierea semnalelor va fi mai redusă.

2.8. Sinteza instrucţiunilor `for loop`

Interpretarea sintezei unei instrucţiuni `for loop` este aceea că se realizează o nouă copie a circuitului descris de conţinutul instrucţiunii la fiecare iteraţie a buclei. Utilizarea instrucţiunii `for loop` pentru generarea unui circuit este ilustrată în Exemplul 27.

Exemplul 27

```
library ieee;
use ieee.std_logic_1164.all;
entity potrivire_biti is
   port (a, b: in std_logic_vector (7 downto 0);
         potriviri: out std_logic_vector (7 downto 0));
end potrivire_biti;

architecture functional of potrivire_biti is
begin
   process (a, b)
   begin
      for i in 7 downto 0 loop
         potriviri (i) <= not (a(i) xor b(i));
      end loop;
   end process;
end functional;
```

Prin procesul din acest exemplu, se generează un set de comparatoare de câte un bit, care compară biţii de acelaşi rang ai vectorilor `a` şi `b`. Rezultatul este înscris în vectorul `potriviri`, care va conţine '1' în poziţiile în care biţii celor doi vectori sunt identici şi '0' în celelalte poziţii.

Procesul din exemplul anterior este echivalent cu următorul proces:

```
process (a, b)
begin
   potriviri (7) <= not (a(7) xor b(7));
   potriviri (6) <= not (a(6) xor b(6));
   potriviri (5) <= not (a(5) xor b(5));
   potriviri (4) <= not (a(4) xor b(4));
   potriviri (3) <= not (a(3) xor b(3));
   potriviri (2) <= not (a(2) xor b(2));
   potriviri (1) <= not (a(1) xor b(1));
   potriviri (0) <= not (a(0) xor b(0));
end process;
```

În acest caz, ordonarea domeniului contorului de iteraţii nu are importanţă, deoarece nu există conexiune între blocurile generate ale circuitului. Această ordonare devine importantă atunci când există o conexiune între aceste blocuri. Această conexiune este creată de obicei de către o variabilă care păstrează o valoare într-o iteraţie a buclei, valoare care este citită apoi într-o altă iteraţie. De obicei, este necesară iniţializarea unei asemenea variabile înaintea intrării în buclă. Exemplul 28 prezintă un asemenea circuit.

Exemplul 28

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

```
entity contorizare_unu is
   port (v: in std_logic_vector (15 downto 0);
         num: out signed (3 downto 0));
end contorizare_unu;

architecture functional of contorizare_unu is
begin
   process (v)
      variable rez: signed (3 downto 0);
   begin
      rez := (others => '0');
      for i in 15 downto 0 loop
         if v(i) = '1' then
            rez := rez + 1;
         end if;
      end loop;
      num <= rez;
   end process;
end functional;
```

Acest exemplu este un circuit combinațional care contorizează numărul biților din vectorul `v` care sunt '1'. Rezultatul este acumulat pe parcursul execuției procesului într-o variabilă numită `rez` și este asignată apoi semnalului de ieșire `num` la sfârșitul procesului. Corpul buclei – o instrucțiune `if` conținând o asignare a unei variabile – reprezintă un bloc format dintr-un multiplexor și un sumator, care va fi generat la sinteză pentru fiecare iterație a buclei. Ieșirea unui bloc devine intrarea `rez` a următorului bloc. Figura 7 prezintă circuitul generat.

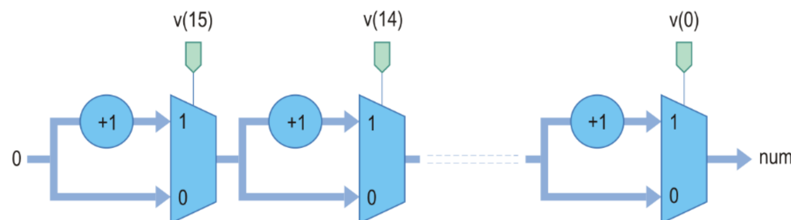

Figura 7. Implementarea unei instrucțiuni `for loop`.

În Exemplul 28, domeniul contorului de iterații al instrucțiunii `for loop` a fost specificat în mod explicit ca fiind `15 downto 0`. În practică, această specificare explicită este utilizată foarte rar pentru accesarea tablourilor, fiind recomandată utilizarea atributelor pentru tablouri. Există patru posibilități, în funcție de domeniul crescător sau descrescător al tabloului sau de ordinea în care trebuie parcurse elementele tabloului.

1. Dacă elementele unui tablou trebuie parcurse de la stânga la dreapta, indiferent de domeniul crescător sau descrescător al tabloului, se utilizează atributul `'range`:

   ```
   for i in v'range loop
   ```

2. Dacă elementele unui tablou trebuie parcurse de la dreapta la stânga, se utilizează atributul `'reverse_range`:

   ```
   for i in v'reverse_range loop
   ```

3. Dacă elementele unui tablou trebuie parcurse de la indexul minim la cel maxim, indiferent de domeniul crescător sau descrescător al tabloului, se utilizează atributele `'low` și `'high`:

   ```
   for i in v'low to v'high loop
   ```

4. În sfârșit, dacă elementele unui tablou trebuie parcurse de la indexul maxim la cel minim, se utilizează atributele `'high` și `'low`, cu specificarea cuvântului cheie `downto`:

   ```
   for i in v'high downto v'low loop
   ```

La scrierea subprogramelor, în cazul în care nu se cunoaşte dinainte dacă o variabilă sau un semnal de tip tablou va avea un domeniu crescător sau descrescător, devine foarte importantă alegerea limitelor corecte pentru instrucţiunile de buclare. Pentru tipurile tablou care reprezintă valori întregi, cum sunt tipurile `signed` şi `unsigned` definite în pachetele `numeric_std` şi `std_logic_arith`, convenţia este că bitul din stânga reprezintă bitul c.m.s. şi bitul din dreapta reprezintă bitul c.m.p.s., indiferent de domeniul tabloului. Aceasta înseamnă că modul corect pentru accesul acestor tablouri este utilizarea atributului `'range` sau `'reverse_range`.

Astfel, pentru accesul unui tablou `n` reprezentând o valoare întreagă începând de la bitul c.m.s. spre bitul c.m.p.s., se va utiliza atributul `'range`:

```
for i in n'range loop
```

Pentru accesul aceluiaşi tablou începând de la bitul c.m.p.s. spre bitul c.m.s., se va utiliza atributul `'reverse_range`:

```
for i in n'reverse_range loop
```

2.9. Sinteza instrucţiunilor `next`

Pentru sinteza unei instrucţiuni `next` sunt necesare aceleaşi circuite ca şi cele necesare pentru sinteza unei instrucţiuni `if`. Alegerea uneia din cele două instrucţiuni rămâne la latitudinea proiectantului.

Ca un exemplu, considerăm acelaşi circuit care contorizează numărul biţilor de '1' dintr-un vector. Exemplul 29 prezintă descrierea modificată a acestui circuit. În locul instrucţiunii `if` se utilizează o instrucţiune `next`, prin care se abandonează o iteraţie în cazul în care valoarea elementului curent este '0', astfel că nu se execută incrementarea contorului.

Exemplul 29

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity contorizare_unu is
   port (v: in std_logic_vector (15 downto 0);
         num: out signed (3 downto 0));
end contorizare_unu;

architecture functional of contorizare_unu is
begin
   process (v)
      variable rez: signed (3 downto 0);
   begin
      rez := (others => '0');
      for i in v'range loop
         next when v(i) = '0';
         rez := rez + 1;
      end loop;
      num <= rez;
   end process;
end functional;
```

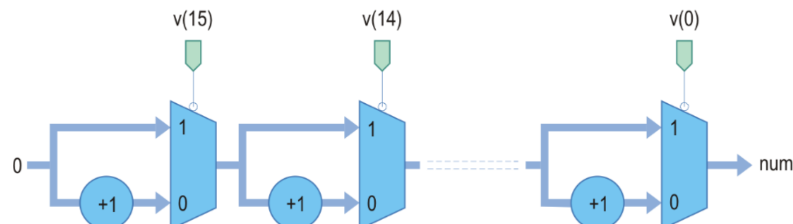

Figura 8. Implementarea unei instrucţiuni `for loop` care conţine o instrucţiune `next`.

Figura 8 prezintă circuitul generat prin sinteza descrierii din exemplul anterior. Singura diferență dintre acest circuit și cel din Figura 7 este că este inversată logica de control a multiplexoarelor. Cele două circuite sunt însă echivalente funcțional.

2.10. Sinteza instrucțiunilor `exit`

Exemplul 30 prezintă descrierea unui circuit pentru contorizarea numărului de zerouri de la sfârșitul unui vector de biți. Se testează fiecare element al vectorului reprezentând o valoare întreagă, iar dacă un element este '1', bucla se termină cu ajutorul unei instrucțiuni `exit`.

Exemplul 30

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity contorizare_zero is
   port (v: in std_logic_vector (15 downto 0);
         num: out signed (3 downto 0));
end contorizare_zero;

architecture functional of contorizare_zero is
begin
   process (v)
      variable rez: signed (3 downto 0);
   begin
      rez := (others => '0');
      for i in v'reverse_range loop
         exit when v(i) = '1';
         rez := rez + 1;
      end loop;
      num <= rez;
   end process;
end functional;
```

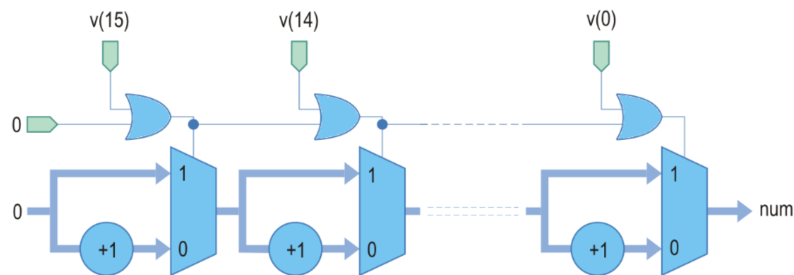

Figura 9. Implementarea unei instrucțiuni `for loop` care conține o instrucțiune `exit`.

Circuitul generat prin sinteza descrierii din exemplul anterior este prezentat în Figura 9. Instrucțiunea `exit` este implementată prin porțile SAU care determină, pentru iterația curentă și toate iterațiile rămase, selectarea intrării multiplexoarelor care nu conțin circuitul de incrementare atunci când condiția instrucțiunii `exit` devine adevărată. Se poate observa că implementarea unei instrucțiuni `exit` dintr-o buclă este similară cu cea a unei instrucțiuni `if` dintr-o buclă.

3. Aplicații

3.1. Răspundeți la următoarele întrebări:

a. Care sunt deosebirile dintre asignarea semnalelor și asignarea variabilelor?

b. Care este problema utilizării instrucțiunilor `if` incomplete în procesele combinaționale?

c. Care este deosebirea dintre instrucțiunile `if` cu semnale și instrucțiunile `if` cu variabile?

d. Când este mai avantajoasă utilizarea unei instrucțiuni `case` în locul unei instrucțiuni `if` și care este avantajul utilizării unei instrucțiuni `case`?

3.2. Identificați și corectați erorile din următoarea descriere:

```
library ieee;                                             --  1
use ieee.std_logic.all;                                   --  2
entity t_c is                                             --  3
   port (clock, reset, enable: in bit;                    --  4
         data: in std_logic_vector (7 downto 0);          --  5
         egal, term_cnt: out std_logic);                  --  6
end t_c;                                                  --  7
architecture t_c of t_c is                                --  8
   signal num: std_logic_vector (7 downto 0);             --  9
begin                                                     -- 10
   comp: process                                          -- 11
   begin                                                  -- 12
      if data = num then                                  -- 13
         egal = '1';                                      -- 14
      end if;                                             -- 15
   end process;                                           -- 16
                                                          -- 17
   count: process (clk)                                   -- 18
   begin                                                  -- 19
      if reset = '1' then                                 -- 20
         num <= "111111111";                              -- 21
      elsif rising_edge (clock) then                      -- 22
         num <= num + 1;                                  -- 23
      end if;                                             -- 24
   end process;                                           -- 25
                                                          -- 26
   term_cnt <= 'z' when enable = '0' else                 -- 27
               '1' when num = "1-------" else             -- 28
               '0';                                       -- 29
end t_c;                                                  -- 30
```

Utilizați mediul de proiectare Xilinx ISE Design Suite pentru compilarea cu succes a acestei descrieri. Corectați apoi erorile semantice ale descrierii.

3.3. Scrieți o secvență pentru setarea semnalului `x` la valoarea funcției ȘI logic între toate liniile unei magistrale de 8 biți `a_bus(7:0)`.

3.4. Descrieți un multiplexor 4:1 de 8 biți utilizând o instrucțiune `case`. Intrările multiplexorului sunt `a(7:0)`, `b(7:0)`, `c(7:0)`, `d(7:0)`, `s(1:0)`, iar ieșirile sunt `q(7:0)`.

3.5. Realizați un decodificator din codul BCD pentru afișajul cu 7 segmente. Intrările decodificatorului sunt `bcd(3:0)`, iar ieșirile sunt `led(6:0)`.

3.6. Descrieți memoria FIFO ale cărei specificații și schemă bloc sunt prezentate în secțiunea 1 din documentul Aplicatie-FIFO.pdf, disponibil pe pagina laboratorului. Creați entitatea și arhitectura pentru modulul *fifo8x8*, după cum se indică în secțiunea 2 din același document.