LUCRAREA NR. 8
DOMENIUL CONCURENT

1. Scopul lucrării

Lucrarea urmăreşte familiarizarea cu noţiunile fundamentale referitoare la domeniul concurent al descrierilor VHDL. Se prezintă detaliat problematica arhitecturilor concurente, cea a proceselor elementare şi cea a semnalelor multi-sursă şi a funcţiilor lor de rezoluţie. Noţiunile expuse pot fi asimilate şi aprofundate cu ajutorul numeroaselor exemple oferite.

2. Noţiuni introductive

2.1 Arhitecturi concurente

În VHDL putem descrie sistemele sub forma unor „*mulţimi de subsisteme funcţionale operând în mod concurent*". Fiecare dintre aceste sisteme – care pot fi interdependente – vor fi specificate sub forma unor procese separate. Nivelul de detaliere (sau *granularitatea*) depinde de necesităţi – uneori un proces poate descrie funcţionarea unei componente complexe, cum ar fi un microprocesor, alteori vom avea câte un proces pentru fiecare componentă elementară de genul porţilor logice fundamentale.

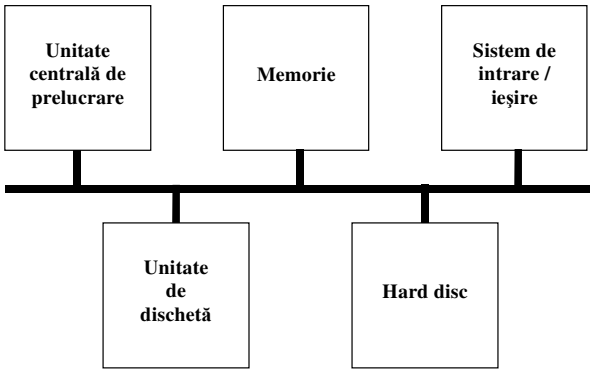

Figura 8.1 *Arhitectura clasică a unui microcalculator*

Descrierea comportamentală a unui sistem va consta deci dintr-o *mulţime de procese secvenţiale care operează în mod concurent*.

De exemplu, pentru arhitectura clasică a microcalculatorului din figura 8.1, descrierea VHDL va arăta astfel:

```
architecture EXEMPLU of MICROCALC is
signal DATABUS: BIT_VECTOR (31 downto 0);
begin
UCP: process -- Unitatea Centrală de Prelucrare
begin
...
end process UCP;
MEM: process -- Memoria
begin
...
end process MEM;
I_E: process -- Sistemul de intrare şi ieşire a datelor
begin
...
end process I_E;
DISC: process -- Hard discul
begin
...
end process DISC;
FLOPPY: process -- Unitatea de dischetă
begin
...
end process FLOPPY;
end architecture EXEMPLU;
```

Pentru înţelegerea domeniului concurent al VHDL este esenţială o foarte bună cunoaştere a noţiunilor legate de *arhitecturi*. Obiectul VHDL *arhitectură* este alcătuit dintr-un *antet* şi un *corp*.

Antetul specifică numele arhitecturii şi limitele corpului arhitecturii. De asemenea, se precizează entitatea căreia îi aparţine arhitectura. În plus, aici se declară eventualele obiecte care vor fi interne arhitecturii.

Corpul arhitecturii are o structură concurentă, care poate fi la început mai dificil de înţeles, datorită faptului că toate instrucţiunile (procesele şi conţinutul lor) sunt scrise în ordine secvenţială (este singura posibilitate existentă în cadrul actualelor sisteme de calcul, bazate pe fişiere). Cu toate acestea, trebuie să reţinem că *toate procesele din interiorul unei arhitecturi sunt executate concurent unele faţă de celelalte*. Toate procesele din cadrul

descrierii în VHDL a sistemului vor fi complet specificate în interiorul corpului arhitecturii.

Iată în continuare două exemple foarte scurte de descriere comportamentală. Primul exemplu arată modelarea în VHDL a unui bistabil D cu acţionare pe frontul ascendent, cu un generator de tact intern, care poate fi mascat de semnalul M, activ pe '0' logic (este un bistabil D cu semnalul M pe post de *clock enable*). Desigur, niciun bistabil fizic nu are un generator de tact intern, dar codul de mai jos reprezintă un model foarte util *numai pentru simularea* mai rapidă şi mai eficientă a bistabilului.

```
entity BIST_D is
      port (M, D: in BIT;
            Q, NQ: buffer BIT);
end BIST_D;

architecture ARH of BIST_D is
signal S1, S2, CLOCK: BIT;
begin
CLK: process -- Generatorul de semnal de tact
      begin
            CLOCK <= '0';
            wait for 5 ns;
            CLOCK <= '1';
            wait for 5 ns;
      end process CLK;
N1: process(M)
      begin
            S1 <= not(M);
      end process N1;
NAND1: process(S1, CLOCK)
      begin
            S2 <= S1 nand CLOCK;
      end process NAND1;
D1: process(S2)
      begin
            if S2 = '1' then
                  Q <= D;
                  NQ <= not(D);
            end if;
      end process D1;
end architecture ARH;
```

Al doilea exemplu arată implementarea în VHDL a arhitecturii din figura 8.2, reprezentând un sumator complet de numere pe un bit.

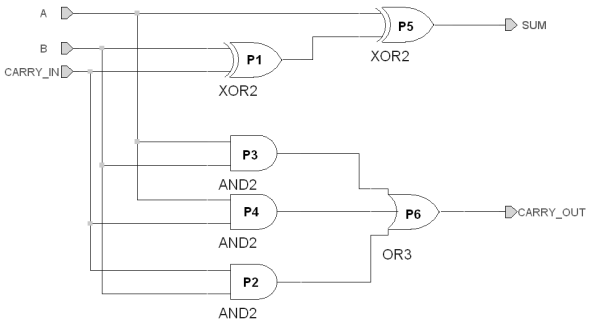

Figura 8.2 *Sumator complet de numere pe un bit*

```
entity SUMATOR_COMPLET is
      port (A, B, CARRY_IN: in BIT;
            SUM, CARRY_OUT: out BIT);
end SUMATOR_COMPLET ;
architecture ARH1 of SUMATOR_COMPLET is
signal S1, S2, S3, S4: BIT;
begin
P1: process(B, CARRY_IN)
      begin
            S1 <= B xor CARRY_IN;
      end process P1;
P2: process(B, CARRY_IN)
      begin
            S2 <= B and CARRY_IN;
      end process P2;
P3: process(A, B)
      begin
            S3 <= A and B;
      end process P3;
P4: process(A, CARRY_IN)
      begin
            S4 <= A and CARRY_IN;
      end process P4;
P5: process(A, S1)
      begin
            SUM <= A xor S1;
      end process P5;
P6: process(S2, S3, S4)
      begin
            CARRY_OUT <= S2 or S3 or S4;
      end process P6;
end architecture ARH1;
```

După cum s-a prezentat deja în lucrarea nr. 6, un proces suspendat va fi reactivat atunci când oricare dintre semnalele de pe lista sa de sensibilitate îşi modifică valoarea. Această regulă se aplică şi în cazul în care există mai multe procese într-o arhitectură: atunci când un semnal îşi modifică valoarea, *toate* procesele care au respectivul semnal în lista lor de sensibilitate vor fi reactivate. Instrucţiunile din interiorul proceselor reactivate vor fi aşadar executate secvenţial, în ordine. Subliniem însă că aceste instrucţiuni vor fi executate *independent* de instrucţiunile aflate în interiorul celorlalte procese.

Conceptul de concurenţă ar fi probabil mult mai uşor de înţeles dacă am putea scrie procesele unul lângă celălalt, şi nu unul după celălalt, aşa cum suntem obligaţi să o facem în fişierul sursă, pe orice sistem de calcul existent în momentul de faţă.

Întrucât procesele nu fac deosebirea între semnalele generate extern (provenite din mediul exterior) şi cele generate intern (declarate în interiorul arhitecturii), rezultă că semnalele care activează procesele pot fi generate şi de către alte procese existente în interiorul aceleiaşi arhitecturi.

De fiecare dată când un semnal aflat pe lista de sensibilitate a unui anumit proces îşi modifică valoarea, procesul respectiv va fi activat. Acest lucru se întâmplă indiferent dacă modificarea valorii semnalului a fost produsă de către mediul exterior sau de către un alt proces.

Observaţie

Pentru a realiza transferuri de informaţie între procese nu se pot utiliza decât *semnale*. Datorită faptului că variabilele sunt obiecte locale proceselor, ele nu pot fi utilizate ca purtătoare de informaţie între procese.

Noţiunile expuse mai sus sunt ilustrate în următorul exemplu: un bistabil JK construit cu bistabil D, având un generator de semnal de tact descris în interiorul arhitecturii.

```
entity BIST_JK is
      port (J, K: in BIT;
            Q, NQ: buffer BIT);
end BIST_JK;
architecture ARH of BIST_JK is
signal S1, S2, S3, S4, CLOCK: BIT;
begin
CLK: process -- Generatorul de semnal de tact
      begin
            CLOCK <= '0';
            wait for 5 ns;
            CLOCK <= '1';
            wait for 5 ns;
      end process CLK;
NAND1: process(J, NQ)
      begin
            S1 <= J nand NQ;
      end process NAND1;
NAND2: process(S1, S4)
      begin
            S2 <= S1 nand S4;
      end process NAND2;
NAND3: process(S3, Q)
      begin
            S4 <= S3 nand Q;
      end process NAND3;
NOT1: process(K)
      begin
            S3 <= not(K);
      end process NOT1;
process(CLOCK)
      begin
            if CLOCK = '1' then
                  Q <= S2;
                  NQ <= not(S2);
            end if;
      end process;
end architecture ARH;
```

2.2 Procese elementare

În anumite cazuri este necesar să utilizăm porţi logice elementare ca module separate în cadrul arhitecturii, precum în exemplul de mai jos:

```
IEŞIRE_ŞI <= IN1 and IN2;
```

Specificarea unui asemenea comportament foarte simplu cu ajutorul unui proces ar necesita încă trei instrucţiuni suplimentare (antetul procesului, clauza **begin** şi clauza **end process**), ceea ce este de prisos în cazul unei simple porţi logice ŞI. Limbajul VHDL permite simplificarea proceselor de acest gen (cele care conţin o singură instrucţiune) prin utilizarea *instrucţiunilor singulare de asignare concurentă de valori semnalelor* (numite astfel deoarece ocupă, de regulă, o singură linie de cod).

O instrucţiune de asignare concurentă de semnal poate apărea în interiorul unei arhitecturi, eventual în paralel cu unele procese, şi poate fi executată concurent cu alte instrucţiuni, în mod similar proceselor. Dacă două sau mai multe instrucţiuni de asignare apar *în cadrul unui proces*, ele vor fi executate *secvenţial în ordinea în care au fost scrise*. Însă, dacă reprezintă instrucţiuni de asignare *concurentă* de semnal (deci apar în afara oricărui proces), ele vor fi executate *în mod concurent*.

Observaţie

Instrucţiunile de asignare concurentă a semnalelor sunt echivalente cu un proces conţinând o singură instrucţiune de asignare de semnal. Dacă în cadrul unui proces apar două sau mai multe instrucţiuni de asignare, ele vor fi executate într-o anumită secvenţă predefinită. Cu toate acestea, dacă aceleaşi instrucţiuni vor apărea ca asignări concurente de semnal (deci apar în afara oricărui proces), ele vor fi executate concurent.

Implementarea unui bistabil RS asincron constituie o ilustrare foarte bună a utilizării instrucţiunilor de asignare concurentă:

```
entity BIST_RS is
      port (SET, RESET: in BIT;
            Q, NQ: buffer BIT);
end BIST_RS;
architecture ARH of BIST_RS is
begin
            Q <= not(NQ and SET) after 1 ns;
            NQ <= not(Q and RESET) after 1 ns;
end architecture ARH;
```

Cele două arhitecturi din exemplul de mai jos sunt echivalente:

```
architecture ARH1 of ENTITATE is
signal IN1, IN2: BIT;
begin
P1: process (A, C)
      begin
            IN1 <= A or C;
      end process P1;
P2: process (B, D)
      begin
            IN2 <= B or D;
      end process P2;
P3: process (IN1, IN2)
      begin
            OUT <= IN1 and IN2;
      end process P3;
end architecture ARH1;

architecture ARH2 of ENTITATE is
signal IN1, IN2: BIT;
begin
      IN1 <= A or C;
      IN2 <= B or D;
      OUT <= IN1 and IN2;
end architecture ARH2;
```

Pentru activarea proceselor am arătat deja că se folosesc listele de sensibilitate şi instrucţiunile **wait**. Atunci, cum stau lucrurile cu instrucţiunile de asignare concurentă a semnalelor, care nu au nici instrucţiuni **wait** şi nici liste de sensibilitate?

După cum se poate observa în exemplul de mai sus, listele de sensibilitate ale tuturor proceselor conţin semnale care apar ulterior în membrul drept al instrucţiunilor de asignare. Prin urmare, instrucţiunea de asignare concurentă de semnal este sensibilă la orice modificare a oricărui semnal care apare în membrul ei drept. Această asignare poate fi întârziată cu ajutorul clauzei **after**, folosindu-se în acest scop modelul de întârziere *inerţial* sau *transport* (prezentate în lucrarea nr. 3).

Revenind la exemplul anterior cu descrierea sumatorului complet de numere pe un bit din figura 8.2, o implementare echivalentă este cea care urmează:

```
entity SUMATOR_COMPLET is
      port (A, B, CARRY_IN: in BIT;
            SUM, CARRY_OUT: out BIT);
end SUMATOR_COMPLET;
architecture ARH2 of SUMATOR_COMPLET is
signal S1, S2, S3, S4: BIT;
begin
            S1 <= B xor CARRY_IN;
            S2 <= B and CARRY_IN;
            S3 <= A and B;
            S4 <= A and CARRY_IN;
            SUM <= A xor S1;
            CARRY_OUT <= S2 or S3 or S4;
end architecture ARH2;
```

2.3 Pilotul (*driver*-ul) unui semnal

În cadrul acestei secţiuni vom relua anumite noţiuni referitoare la semnale, în special cele referitoare la semnalele multi-sursă.

Semnalele primesc noi valori în cadrul proceselor numai în momentul suspendării acestora, suspendare realizată cu ajutorul instrucţiunii **wait** explicite sau implicite (prin lista de sensibilitate a procesului). În plus, dacă există mai multe asignări de valori pentru un acelaşi semnal, numai ultima va fi luată în considerare. Atunci, cum se pot stoca informaţiile referitoare la evenimentele apărute pe un anumit semnal?

Această funcţie este îndeplinită de *pilotul* (*driver*-ul) semnalului respectiv. Compilatorul VHDL creează câte un pilot pentru fiecare semnal care primeşte o valoare în interiorul unui proces. Regula este foarte simplă: indiferent câte valori sunt atribuite semnalului în cadrul procesului, există un singur pilot *per* semnal *per* proces. Toate operaţiile sunt efectuate asupra pilotului, care este copiat în semnalul propriu-zis numai atunci când procesul este suspendat.

Datorită pilotului său, fiecare semnal cunoaşte valorile sale trecute, prezente şi viitoare (acestea din urmă pot fi cunoscute deoarece fiecare pilot poate specifica şi *forma de undă proiectată să apară* pe semnalul respectiv). Fiecărui pilot îi poate fi asignată o asemenea formă de undă care va consta dintr-o secvenţă de una sau mai multe *tranzacţii*. O tranzacţie constă dintr-o valoare a semnalului împreună cu o valoare de tip TIME, care va specifica

momentul de timp când pilotului îi va fi asignată noua valoare specificată de către tranzacţie.

Forma de undă poate fi specificată în mod explicit ca o secvenţă de valori împreună cu întârzierile asociate. Formele de undă pot fi considerate drept *viitorul proiectat* al semnalului. Întrucât simulatoarele stochează tranzacţiile fiecărui semnal, ele creează într-adevăr *istoria semnalelor*.

Un semnal căruia i se poate determina cu uşurinţă istoria şi viitorul este semnalul CLOCK din cadrul exemplelor anterioare de bistabili cu generator de tact intern (bistabil D şi bistabil JK).

Atunci când un semnal are un singur pilot, valoarea sa este foarte uşor de determinat. În foarte multe aplicaţii, semnalele cu mai multe surse sunt o apariţie obişnuită. De exemplu, în interiorul unui calculator cu arhitectură clasică von Neumann, magistrala internă primeşte valori din partea procesorului, a memoriei, a discurilor şi a dispozitivelor de intrare / ieşire. Fiecare fir metalic din componenţa magistralei va fi deci *pilotat* de mai multe dispozitive. De vreme ce VHDL este un limbaj de descriere hardware specializat pentru sistemele numerice, astfel de situaţii sunt controlate cu uşurinţă.

Este greu de stabilit dacă un semnal cu mai multe surse va fi întotdeauna pilotat (comandat), la un anumit moment dat, de către un singur dispozitiv. În unele sisteme această situaţie trebuie evitată cu stricteţe, pe când în altele ea este chiar dorită (de exemplu, în cazul implementărilor de „poartă ŞI cablată" sau „poartă SAU cablată" (*wired AND*, respectiv *wired OR*)). În general, semnalele cu surse multiple necesită stabilirea unei metode de determinare a valorii rezultante atunci când există mai multe dispozitive sursă care furnizează date în mod concurent.

În cele ce urmează prezentăm exemplul unui controler de magistrală utilizat pentru citirea şi scrierea în / din dispozitive de intrare / ieşire. Magistrala poate fi accesată şi din exterior. De remarcat semnalele multi-sursă şi asignarea semnalelor în procese diferite.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity SISTEM is
 port (INSTR: in INTEGER range 0 to 4; -- codul instrucţiunii
       DATA_BUS: inout STD_LOGIC_VECTOR(7 downto 0));
end SISTEM;
```

```
architecture ARH of SISTEM is
signal CIT_SCR: STD_LOGIC; -- '1' = citire, '0' = scriere
signal SELECT_DISP: STD_LOGIC;
-- '1' memoria, '0' interfaţa de I/E

begin
CTRL: process(INSTR) -- Procesul de control
 begin
  case INSTR is
   when 0 => CIT_SCR <= '1';         -- Instrucţiune de citire
             SELECT_DISP <= '1';     -- din memorie
   when 1 => CIT_SCR <= '0';         -- Instrucţiune de scriere
             SELECT_DISP <= '1';     -- în memorie
   when 2 => CIT_SCR <= '1';         -- Instrucţiune de citire
             SELECT_DISP <= '0';     -- din dispozitivul de I/E
   when 3 => CIT_SCR <= '0';         -- Instrucţiune de scriere
             SELECT_DISP <= '0';     -- în dispozitivul de I/E
   when 4 => CIT_SCR <= 'Z';         -- Instrucţiune nulă
             SELECT_DISP <= 'Z';
  end case;
end process CTRL;

MEM: process(SELECT_DISP, CIT_SCR) -- Emularea memoriei
variable CEL: STD_LOGIC_VECTOR (7 downto 0);
begin
 if SELECT_DISP = '1' then
  if CIT_SCR = '1' then
    DATA_BUS <= CEL after 7 ns;      -- Citire din memorie
  else CEL := DATA_BUS;              -- Scriere în memorie
  end if;
 else DATA_BUS <= (others => 'Z');  -- Înaltă impedanţă
 end if;
end process MEM;
```

```
I_E: process(SELECT_DISP, CIT_SCR)  -- Emularea interfeţei de I/E
variable INTR_IES: STD_LOGIC_VECTOR (7 downto 0);
begin
 if SELECT_DISP = '0' and SELECT_DISP'EVENT then
  if CIT_SCR = '1' then
   DATA_BUS <= INTR_IES after 12 ns;-- Citire de la portul de I/E
  else INTR_IES:= DATA_BUS;          -- Scriere la portul de I/E
  end if;
 else DATA_BUS <= (others => 'Z');  -- Înaltă impedanţă
 end if;
end process I_E;
end architecture ARH;
```

2.4 Rezolvarea semnalelor multi-sursă

Simulatorul VHDL nu poate „şti" cu anticipaţie dacă un semnal multi-sursă va fi sau nu activat din două sau mai multe surse simultan. Datorită acestui fapt, simulatorul trebuie să fie „pregătit" pentru a mixa valorile semnalului. Această „mixare" a semnalelor se numeşte în VHDL *rezolvare* (*resolving*).

Regulile de mixare a valorilor semnalelor sunt specificate sub forma unui tabel, care se numeşte *funcţie de rezoluţie*. Tabelul conţine toate valorile posibile ale semnalului şi fiecare poziţie din el conţine informaţii referitoare la valoarea care va fi generată dacă cele două valori corespunzătoare liniei şi coloanei respective vor fi mixate.

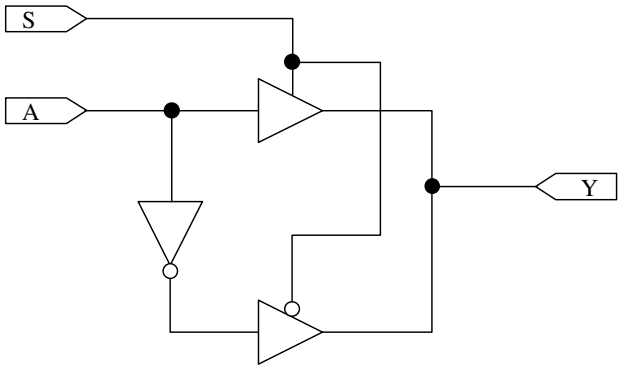

Figura 8.3 *Semnal multi-sursă (Y) cu doi piloţi*

În figura 8.3 este prezentat un exemplu foarte simplu de semnal multi-sursă. Semnalul Y este comandat de doi piloţi. Semnalul de selecţie S va determina dacă ieşirea Y va fi conectată la A sau la **not**(A).

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity PILOŢI1 is
      port (A, S: in STD_LOGIC;
            Y: out STD_LOGIC);
end PILOŢI1;
```

```
architecture ARH1 of PILOŢI1 is
begin
      P1: process(A, S)
      begin
            if S = '1' then Y <= A;
            else Y <= 'Z';
            end if;
      end process P1;
      P2: process(A, S)
      begin
            if S = '0' then Y <= not(A);
                       else Y <= 'Z';
            end if;
      end process P2;
end architecture ARH1;
```

Studiind cu mai multă atenţie această problemă, ne putem da seama că funcţia de rezoluţie are nevoie de mai multe valori decât cele din componenţa setului de bază. O situaţie similară apare atunci când încercăm să reprezentăm semnale multi-sursă în VHDL. De exemplu, pentru tipul BIT, prima problemă care se pune este următoarea: ce se va întâmpla dacă vom mixa valorile '0' şi '1' (cu alte cuvinte, *care este valoarea rezolvată pentru '0' şi '1'*?). Problema nu poate fi rezolvată folosind numai două valori – prin urmare, întrebarea precedentă nu are răspuns.

Acest aspect are o serie întreagă de consecinţe: dacă vom folosi numai tipurile de date BIT şi BIT_VECTOR, atunci NU vom putea specifica un microprocesor (de exemplu) în VHDL. Tipurile de date nerezolvate (aşa cum sunt cele două tipuri menţionate mai sus, BIT şi BIT_VECTOR) nu pot fi folosite pentru semnale multi-sursă, care sunt însă indispensabile pentru specificarea magistralelor de date (care există, în mod evident, în orice microprocesor). Pentru rezolvarea acestei probleme, trebuie să utilizăm alt tip de date, care să conţină mai mult de două valori şi care să aibă definită o funcţie de rezoluţie pentru toate combinaţiile valorilor semnalelor.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity PILOŢI2 is
      port (A: in BIT;
            Y: out BIT;
            B: in STD_LOGIC;
            Y1: out STD_LOGIC);
end PILOŢI2;
```

```
architecture ARH2 of PILOŢI2 is
begin

P1: process(A)
begin -- Această condiţie este suficientă pentru detectarea
      -- frontului ascendent în cazul tipului BIT
      if A = '1' and A'EVENT then Y <= A after 1 ns;
      end if;
end process P1;

P2: process(B)
begin -- Această condiţie este suficientă pentru detectarea
      -- frontului ascendent în cazul tipului STD_LOGIC
      if B = '1' and B'EVENT and B'LAST_VALUE = '0'
       then Y1 <= B after 2 ns;
      end if;
end process P2;
end architecture ARH2;
```

Uneori, chiar şi semnalele uni-sursă necesită mai mult de două valori pentru reprezentarea „obiectelor” hardware. Iată câteva dintre cele mai des întâlnite valori care pot să apară:

- Dacă ştim despre un semnal că a fost iniţializat, dar nu putem determina cu precizie dacă în acest moment valoarea sa este ‘0’ sau ‘1’ sau ‘Z’, atunci valoarea semnalului va fi notată cu ‘X’;
- Tampoanele (*buffers*) tri-state deconectează dispozitivele generatoare de liniile de semnal prin intermediul stării de „*înaltă impedanţă*”, notată cu 'Z';
- Uneori, un semnal poate avea o valoarea „*neasignată*” sau „*necunoscută*”, care este diferită de valoarea indiferentă. Această stare poate apărea, de exemplu, în cazul unui bistabil, a cărui stare internă nu este cunoscută în momentul punerii sub tensiune a sistemului, dacă nu există o logică de auto-iniţializare; ea se notează cu 'U';
- Valoarea notată cu 'L' înseamnă starea '0' slabă (de exemplu, ieşire emitor deschis în starea logică '0');
- Valoarea notată cu 'H' înseamnă starea '1' slabă (de exemplu, ieşire emitor deschis în starea logică '1');
- Valoarea notată cu 'W' înseamnă starea necunoscută slabă;

- Uneori, valoarea semnalului nu este importantă. Atunci, spunem că valoarea sa este *indiferentă* (*don't care*) şi se notează cu '-'.

Aceste valori şi altele, mai puţin frecvente, sunt specificate de tipul de date STD_ULOGIC, definit în cadrul pachetului STD_ULOGIC_1164 (litera U vine de la termenul ***U****nresolved types*). Pachetul conţine, de asemenea, o definiţie a tipului vector, care se bazează pe tipurile STD_ULOGIC şi STD_ULOGIC_VECTOR. Ambele tipuri au şi un set de operaţii logice de bază definite asupra lor. Aceste valori nu pot fi utilizate în cazul semnalelor multi-sursă.

Un semnal care posedă o funcţie de rezoluţie se numeşte *semnal rezolvat*, ceea ce înseamnă că poate avea mai multe surse - cu alte cuvinte, acestui semnal îi pot fi asignate mai multe valori, în cadrul unor procese diferite.

Funcţia de rezoluţie poate fi menţionată la nivelul unei declaraţii de sub-tip de date (situaţie în care ea se va aplica tuturor semnalelor de acest sub-tip) sau a unei declaraţii de semnal.

În decursul simulării, această funcţie este apelată pentru calcularea valorii pe care o va lua semnalul. Pentru calcul, ea utilizează toate valorile surselor semnalului respectiv, cu excepţia celor care fac obiectul unei tranzacţii nule, adică cele care sunt deconectate. În VHDL este posibilă specificarea deconectării unui semnal, funcţionalitate deosebit de preţioasă pentru modelarea unei magistrale.

De ce este asociată funcţia de rezoluţie unui sub-tip de date sau unui semnal şi nu unui tip de date? Această particularitate a limbajului VHDL permite păstrarea compatibilităţii cu tipul de date de bază, oferind totodată utilizatorului mai multe sub-tipuri de date, fiecare cu propria sa funcţie de rezoluţie.

Pentru tipurile compuse, putem avea o funcţie de rezoluţie la nivelul elementelor şi o alta la nivelul tipului compus însuşi. Apare deci un conflict: care dintre aceste funcţii de rezoluţie va fi aplicată? Problema este soluţionată astfel: dacă există o funcţie de rezoluţie definită la nivelul tipului compus, ea va fi cea luată în considerare, şi va masca funcţia de rezoluţie a elementelor constituente ale respectivului tip de date.

Funcţia de rezoluţie nu are decât un singur parametru de intrare care este un vector (tablou uni-dimensional) neconstrâns (numărul de elemente nu este impus), ale cărui elemente sunt de acelaşi tip ca şi semnalul. În

timpul ciclului de simulare, ea va fi utilizată automat de fiecare dată când se va pune problema calculării valorii rezultante.

Observaţie

O funcţie de rezoluţie trebuie în principiu să fie comutativă şi asociativă, dacă se urmăreşte modelarea corectă a unui conflict de valoare. Dacă nu se doreşte aşa ceva, atunci ordinea apelurilor, care nu se află sub controlul proiectantului, va influenţa valoarea rezultantă a unui conflict. Cu toate acestea, aceste proprietăţi (comutativitatea şi asociativitatea) nu fac obiectul unor verificări din partea compilatorului.

Cele mai importante aspecte de reţinut în cazul funcţiilor de rezoluţie sunt următoarele:

1. Argumentul funcţiei este unic şi este un vector cu elemente de acelaşi tip ca şi valoarea returnată de funcţie;
2. Proiectantul nu controlează apelul acestei funcţii, care se face automat de către simulator atunci când apare necesitatea rezolvării (la conflicte).

Vom prezenta în continuare două exemple. Primul este cel al unei funcţii de rezoluţie aplicabilă unor semnale de tipul BIT_REZOLV. BIT_REZOLV este un tip de date care conţine patru valori: '0', '1', 'X' şi 'Z'. BIT_REZOLV_VECTOR este definit ca un vector cu un număr oarecare de elemente de tip BIT_REZOLV.

Reamintim că se recomandă includerea funcţiei de rezoluţie într-un pachet care să conţină şi declaraţia tipului de date rezolvat corespunzător.

```
function REZ(SURSE: BIT_REZOLV_VECTOR) return BIT_REZOLV is
variable  REZULTAT: BIT_REZOLV := 'Z';
begin
-- Algoritmul calculează valoarea returnată (REZULTAT) în
-- funcţie de valorile vectorului SURSE
for i in SURSE'RANGE loop
      case SURSE(i) is
       when 'X' => return 'X';
       when '0' => if REZULTAT = '1' then
                        return 'X';
                   else REZULTAT := '0';
                   end if;
```

```
       when '1' => if REZULTAT = '0' then
                        return 'X';
                   else REZULTAT := '1';
                   end if;
       when 'Z' => null;
      end case;
end loop;
return REZULTAT;
end REZ;
```

Al doilea exemplu este de fapt un extras din codul sursă original din pachetul IEEE STD_LOGIC şi reprezintă funcţia şi tabelul de rezoluţie pentru tipul de date STD_LOGIC. Aceasta este cea mai simplă modalitate de reprezentare a funcţiei de rezoluţie pentru logica cu 9 valori:

```
package body  STD_LOGIC_1164 is
constant RESOLUTION_TABLE: STD_LOGIC_TABLE := (
------------------------------------------------------------
|  U    X     0     1     Z     W     L     H     -      |   |
------------------------------------------------------------
('U', 'U',  'U',  'U',  'U',  'U',  'U',  'U',  'U'),-- | U |
('U', 'X',  'X',  'X',  'X',  'X',  'X',  'X',  'X'),-- | X |
('U', 'X',  '0',  'X',  '0',  '0',  '0',  '0',  'X'),-- | 0 |
('U', 'X',  'X',  '1',  '1',  '1',  '1',  '1',  'X'),-- | 1 |
('U', 'X',  '0',  '1',  'Z',  'W',  'L',  'H',  'X'),-- | Z |
('U', 'X',  '0',  '1',  'W',  'W',  'W',  'W',  'X'),-- | W |
('U', 'X',  '0',  '1',  'L',  'W',  'L',  'W',  'X'),-- | L |
('U', 'X',  '0',  '1',  'H',  'W',  'W',  'H',  'X'),-- | H |
('U', 'X',  'X',  'X',  'X',  'X',  'X',  'X',  'X'),-- | - |
);
```

```
function RESOLVED(S: STD_ULOGIC_VECTOR) return STD_ULOGIC is
variable  RESULT: STD_ULOGIC := 'Z'; -- implicit e starea cea
                                     -- mai „slabă”
begin

-- Este esenţial să se efectueze un test pentru detectarea
-- cazului în care există un singur pilot, altminteri bucla
-- ar returna valoarea 'X' în cazul în care ar exista un
-- singur pilot de semnal de valoare '-'
if (S'LENGTH = 1) then return S(S'LOW);
else
```

```
      for I in S'RANGE loop
            RESULT := RESOLUTION_TABLE(RESULT, S(I));
      end loop;
end if;
return RESULT;
end RESOLVED;
```

Recapitulând: tipul de date STD_ULOGIC suportă toate valorile ce s-ar putea dovedi necesare pentru specificarea oricărui sistem numeric. Totuşi, acest tip de date este nerezolvat, fiind inutilizabil în cazul semnalelor multi-sursă. De aceea, în pachetul STD_LOGIC_1164 s-a definit un tip de date suplimentar: STD_LOGIC, care îmbină puterea expresivă a celor nouă valori ale lui STD_ULOGIC cu avantajele funcţiei de rezoluţie, oferind proiectantului un tip de date universal. În momentul de faţă, STD_LOGIC s-a impus ca un adevărat standard industrial *de facto*.

Singura diferenţă dintre STD_LOGIC şi STD_ULOGIC este aceea că primul este o versiune rezolvată a celui de-al doilea. Din acest motiv, toate operaţiile şi funcţiile (inclusiv RISING_EDGE şi FALLING_EDGE) definite pentru STD_ULOGIC pot fi utilizate şi pentru STD_LOGIC, fără nici o altă declaraţie suplimentară. Tipul STD_LOGIC_VECTOR este o versiune rezolvată a lui STD_ULOGIC_VECTOR.

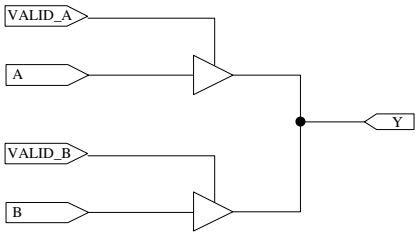

Figura 8.4 *Circuit a cărui ieşire este controlată de doi piloţi independenţi*

În circuitul din figura 8.4, ieşirea Y este controlată din două surse independente. Specificaţia sa în VHDL este prezentată mai jos:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity PILOŢI3 is
      port (VALID_A, VALID_B: in BOOLEAN;
            A, B: in STD_LOGIC;
            Y: out STD_LOGIC);
end PILOŢI3;
```

```
architecture ARH3 of PILOŢI3 is
signal SA, SB: STD_LOGIC;
begin
      SA <= A when VALID_A else 'Z';
      SB <= B when VALID_B else 'Z';
      P1: process(SA)
      begin
            Y <= SA;
      end process P1;
      P2: process(SB)
      begin
            Y <= SB;
      end process P2;
end architecture ARH2;
```

3. Desfăşurarea lucrării

3.1 Se va desena structura internă (cu porţi) a bistabilului D descris în primul exemplu din lucrare.

3.2 Se va testa sumatorul complet de numere pe un bit din figura 8.2. Cum se poate face trecerea la un sumator de numere pe 2, 3, 4 etc. biţi?

3.3 Se va desena structura internă (cu porţi) a bistabilului JK descris în exemplul din lucrare.

3.4 Se va proiecta o componentă "*buffer tri-state*" cu semnalul de validare (*enable*) activ pe 0.

3.5 Se va implementa şi testa exemplul funcţiei de rezoluţie pentru tipul BIT_REZOLV.