LUCRAREA NR. 7
INSTRUCŢIUNI SECVENŢIALE

1. Scopul lucrării

Lucrarea prezintă pe larg toate instrucţiunile secvenţiale permise în cadrul domeniului secvenţial: instrucţiunea **assert**, instrucţiunea **report**, instrucţiunea de asignare de valori semnalelor, apelul secvenţial de procedură, structura condiţională **if-then-else**, instrucţiunea **case**, structura de buclă **loop**, instrucţiunea **next**, instrucţiunea **exit**, instrucţiunea **return** şi instrucţiunea nulă. Pentru toate instrucţiunile se pune la dispoziţie sintaxa lor completă, însoţită de exemple semnificative.

2. Consideraţii teoretice

2.1 Instrucţiunea *assert*

Această instrucţiune permite supravegherea unei condiţii şi emiterea unui mesaj în cazul în care condiţia este *falsă* (FALSE).

Sintaxa instrucţiunii este următoarea:

```
assert condiţie {report mesaj}{severity nivel_de_severitate};
```

În cursul execuţiei acestei instrucţiuni se evaluează condiţia monitorizată. Dacă această condiţie (booleană) este adevărată, nu se întâmplă nimic - instrucţiunea este echivalentă cu instrucţiunea nulă. Dimpotrivă, dacă ea se dovedeşte falsă, atunci mesajul indicat şi nivelul de severitate al erorii sunt utilizate pentru a „confecţiona” un mesaj care va fi afişat imediat. Execuţia programului se reia apoi cu instrucţiunea imediat următoare instrucţiunii **assert**.

Mesajul este în mod obligatoriu un şir de caractere de un sub-tip al tipului STRING. În caz de absenţă a clauzei **report** (care este opţională), mesajul implicit este „Assertion Violation”.

Nivelul de severitate al erorii este o expresie de tipul SEVERITY_LEVEL. Acest tip este definit în pachetul STANDARD; este un tip enumerat definit astfel:

```
type  SEVERITY_LEVEL is (NOTE, WARNING, ERROR, FAILURE);
```

În caz de absenţă a clauzei **severity**, se alege implicit nivelul de severitate ERROR. Numeroase implementări decid oprirea simulării în cazul întâlnirii unui asemenea nivel de severitate.

Se recomandă ca mesajul emis să conţină, pe lângă informaţiile expuse mai sus, şi numele entităţii în care se găseşte instrucţiunea **assert**. Această recomandare este deosebit de utilă mai ales în fazele incipiente de proiectare a sistemului numeric în VHDL.

De obicei, o instrucţiune **assert** cu nivelul de severitate FAILURE opreşte simularea. Prin urmare, putem controla sfârşitul simulării cu ajutorul unei condiţii. Un exemplu al acestei practici este următorul:

```
assert NOW < 1 min report "Sfârşitul normal al simulării"
severity FAILURE;
```

Această linie va opri simularea, generând mesajul dat de îndată ce timpul curent (returnat de către NOW, o funcţie din pachetul STANDARD) va fi egal cu 1 minut. Cu alte cuvinte, se va simula numai primul minut de funcţionare a sistemului.

2.2 Instrucţiunea *report*

Această instrucţiune a fost introdusă numai din motive de uşurare a scrierii în VHDL. Ea nu aduce funcţionalităţi suplimentare faţă de cele oferite de instrucţiunea **assert**, ci permite pur şi simplu afişarea unui mesaj pe ecran. Implicit, simularea se desfăşoară apoi normal, dar există posibilitatea specificării, la fel ca în cazul instrucţiunii precedente, a unui nivel de severitate al mesajului.

Aşadar, instrucţiunea:

```
report "Salut!";
```

este echivalentă cu:

```
assert FALSE report "Salut!" severity NOTE;
```

Sintaxa completă a instrucţiunii **report** este următoarea:

```
[etichetă:] report mesaj [nivel_de_severitate_al_mesajului];
```

În VHDL se permite utilizarea unei etichete pentru fiecare instrucţiune secvenţială.

2.3 Instrucţiunea de asignare de valori variabilelor

Asignarea semnalelor face parte tot din domeniul secvenţial; aceasta a fost însă prezentată pe larg în cadrul lucrării nr. 3.

Asignarea de valori variabilelor se notează cu „:=". Iată câteva exemple simple:

```
VAR1 := 7;
VAR2 := VAR1 / (VAR1 + 22);
```

Aşa cum am arătat mai sus, variabila ia imediat valoarea indicată (spre deosebire de semnal).

Variabilele pot fi asignate încă de la declarare, cu ajutorul aceluiaşi operator de asignare:

```
-- Zona declarativă
variable X: INTEGER := 7;
variable Y: REAL := RADICAL(X);
-- Presupunem că RADICAL este o funcţie definită anterior
-- care returnează un rezultat de tip REAL
```

În ceea ce priveşte poantorii (tipul acces), este posibilă efectuarea de asignări dinamice. Iniţializarea „în trecere" a obiectului spre care poantează o variabilă de acest tip se face astfel:

```
-- Zona declarativă
type P_POANTOR is acces INTEGER; -- Declaraţia unui tip acces
-- care poantează spre un obiect de tip întreg
variable UN_POANTOR: P_POANTOR;
begin
      UN_POANTOR := new INTEGER (3);
-- S-a creat un obiect întreg iniţializat cu valoarea 3.
-- Tipul acces care poantează la acest întreg este stocat în
-- variabila UN_POANTOR.
end;
```

Observaţie

Instrucţiunea **new** „consumă" spaţiu de memorie deoarece ea creează un obiect. La crearea fiecărui tip acces, se creează implicit o procedură DEALLOCATE (a se vedea lucrarea nr. 4) care permite recuperarea spaţiului de memorie ocupat de către obiectele de acest tip.

La asignarea tablourilor şi chiar a porţiunilor de tablouri unidimensionale, trebuie să urmărim ca dimensiunile tablourilor din membrul stâng şi din membrul drept al instrucţiunii de asignare să fie identice. O diferenţă între dimensiunile celor două porţiuni ale tablourilor va provoca o eroare la compilare (sau la execuţie, dacă este vorba despre porţiuni dinamice de tablouri). Iată un exemplu:

```
type TABLOU is array (1 to 100) of INTEGER;
variable TAB1, TAB2: TABLOU;
...
TAB1 := TAB2;
TAB1(3 to 5):= (1, 2, 3);
```

2. 4 Instrucţiunea secvenţială de apel de procedură

Să luăm procedura O_PROCEDURĂ a cărei specificaţie este următoarea:

```
procedure O_PROCEDURĂ (NUMĂR: INTEGER; MESAJ: STRING);
```

Apelarea acestei proceduri se realizează prin indicarea numelui său, urmată de lista de parametri de apel, care se scrie între paranteze. Această listă poate fi dată:

- într-o formă poziţională (de exemplu: O_PROCEDURĂ(4, "START"););
- prin numirea parametrilor formali ai procedurii (de exemplu: O_PROCEDURĂ(NUMĂR =>4, MESAJ =>"START"););
- prin combinarea celor două forme menţionate mai sus, urmărind însă să plasăm mai întâi partea poziţională (de exemplu, O_PROCEDURĂ(4, MESAJ => "START");). În acest caz, ultimii parametri ai apelului pot fi omişi dacă au o valoare implicită.

Apelul de procedură este detaliat în lucrarea nr. 10 dedicată sub-programelor.

2.5 Structura condiţională

Aceasta este o instrucţiune structurată care permite executarea condiţionată a unor secvenţe de instrucţiuni (care pot fi, la rândul lor, structurate). Sintaxa sa de bază este următoarea:

```
if Condiţie_booleană then
      Secvenţa_de_instrucţiuni_1
else
      Secvenţa_de_instrucţiuni_2
end if;
```

Secvenţa de instrucţiuni numărul 1 nu va fi executată decât atunci când condiţia booleană este TRUE, pe când secvenţa de instrucţiuni numărul 2 nu va fi executată decât atunci când condiţia booleană este FALSE. Aşadar, secvenţele de instrucţiuni numărul 1 şi numărul 2 sunt mutual exclusive.

Ramura **elsif** permite înlănţuirea condiţiilor, fără a creşte nivelul de imbricare:

```
if A>B then
      ...
      -- Secvenţa_de_instrucţiuni_1
      ...
else
      if A=B then
            ...
            -- Secvenţa_de_instrucţiuni_2
            ...
      else
            ...
            -- Secvenţa_de_instrucţiuni_3
            ...
      end if;
end if;
```

Acest exemplu conţine două niveluri de imbricare care pot fi reduse la unul singur prin utilizarea ramurii **elsif**. Numărul de ramuri **elsif** nu este limitat şi această formă poate fi folosită în mod conjugat sau nu cu o ramură **else** (şi numai una). Câştigul rezultat din punct de vedere al indentării poate ameliora semnificativ lizibilitatea codului în cazul structurilor cu un număr mare de ramuri.

```
if A>B then
      ...
      -- Secvenţa_de_instrucţiuni_1
      ...
elsif A=B then
      ...
      -- Secvenţa_de_instrucţiuni_2
      ...
else
      ...
      -- Secvenţa_de_instrucţiuni_3
      ...
end if;
```

2.6 Instrucţiunea *case*

Această instrucţiune permite selectarea, în funcţie de valoarea unei expresii, a unei secvenţe de instrucţiuni dintre mai multe alternative. Sintaxa sa generală este următoarea:

```
case expresie is
      when Valoare_1 =>...          -- Secv_instrucţiuni_1
      when Val_2|Val_3|Val_4 =>...  -- Secv_instrucţiuni_2
      when Val_5 to Val_6  =>...    -- Secv_instrucţiuni_3
      ...
      when others =>...             -- Secv_instrucţiuni_n
end case;
```

Câmpul „expresie” şi valorile trebuie să aibă acelaşi tip discret (enumerat). Ele pot avea şi un tip de structură tablou de biţi sau de caractere, uni-dimensional. În toate aceste cazuri, valorile trebuie să fie statice (cu alte cuvinte, care pot fi calculate în faza de elaborare).

Semnificaţia instrucţiunii este aceea că se doreşte executarea ramurii a cărei valoare corespunde valorii expresiei.

Ordinea ramurilor nu are importanţă, cu o singură excepţie: ramura **others** trebuie să se găsească la sfârşitul instrucţiunii, pentru a indica o alegere "în ultimă instanţă". Această ramură (**others**) este obligatorie în toate cazurile în care nu este enumerat întregul ansamblu al valorilor posibile ale expresiei.

2.7 Structura de buclă

Bucla se foloseşte pentru repetarea secvenţei de instrucţiuni din cadrul ei. Fiecare trecere se numeşte *iteraţie*.

Sintaxa generală a acestei instrucţiuni este următoarea:

```
{etichetă:} {schemă de iteraţie} loop
      Secvenţă_de_instrucţiuni
end loop {etichetă};
```

Schema de iteraţie arată de câte ori va fi repetată secvenţa de instrucţiuni. Dacă această schemă de iteraţie nu este precizată, atunci

numărul de iteraţii va fi infinit. De exemplu, următoarea buclă va fi executată până la depăşirea limitei INTEGER'HIGH de către variabila N:

```
variable N: INTEGER;
...
loop
      N := N + 1;
end loop;
```

Există două scheme de iteraţie diferite:

1. Prima formă continuă iteraţiile „atâta timp cât” o condiţie dată este adevărată; această condiţie este testată din nou la începutul fiecărei iteraţii:

```
while condiţie loop
      Secvenţă_de_instrucţiuni
end loop;
```

De exemplu:

```
variable R, M, N: INTEGER;
...
while R /= 0 loop
      R := M mod N;
      M := N;
      N := R;
end loop;
```

2. Cea de-a doua formă repetă secvenţa de instrucţiuni de un anumit număr de ori care nu poate fi cunoscut decât în momentul execuţiei. Există o variabilă numită *variabilă de buclă* care permite contorizarea numărului de cicluri. Această contorizare se efectuează prin parcurgerea unui tip enumerat oarecare, de exemplu a unui sub-tip al lui INTEGER. Nu există posibilitatea specificării unui pas (*step*), însă se poate utiliza eventual o variabilă intermediară pentru rezolvarea acestei probleme.

```
for INDICE in 1 to 100 loop
      ... Secvenţă de instrucţiuni
end loop;
```

În cazul unui interval vid sau negativ, secvenţa de instrucţiuni este pur şi simplu ignorată. Variabila de buclă (în exemplul anterior este vorba despre INDICE) nu trebuie să fie declarată şi ea nu este cunoscută decât în interiorul buclei. De asemenea, ei nu-i poate fi atribuită nici o altă valoare şi deci nu poate fi modificată.

Bucla poate avea (opţional) o etichetă, care, dacă există, trebuie să apară şi la sfârşitul buclei (**end loop**). Pe lângă avantajele legate de lizibilitate pe care această etichetă le procură, ea poate fi utilizată şi în cadrul instrucţiunilor **exit** şi **next**, care vor fi prezentate în cele ce urmează.

2.8 Instrucţiunea *next*

Instrucţiunea **next** permite oprirea iteraţiei în curs de desfăşurare a unei bucle. După această instrucţiune, execuţia va continua cu iteraţia următoare (dacă aceasta există).

Prin urmare, instrucţiunea **next** poate fi:

- imperativă:

```
next {eticheta_buclei};
```

- condiţională:

```
next {eticheta_buclei} when condiţie;
```

În acest ultim caz, întreruperea iteraţiei nu are loc decât dacă respectiva condiţie este adevărată.

Această instrucţiune, dacă nu prezintă o etichetă, se referă la bucla de nivelul cel mai de jos care o înglobează. Dacă prezintă o etichetă, atunci instrucţiunea **next** permite ieşirea din iteraţia curentă a buclei care poartă aceeaşi etichetă.

2.9 Instrucţiunea *exit*

La fel ca în cazul instrucţiunii **next,** instrucţiunea **exit** permite ieşirea dintr-o buclă (şi deci întrerupe toate iteraţiile restante ale buclei). După execuţia acestei instrucţiuni, programul va continua cu instrucţiunea imediat următoare sfârşitului buclei (**end loop**).

Prin urmare instrucţiunea **exit** poate fi:

- imperativă;

```
exit {eticheta_buclei};
```

- condiţională.

```
exit {eticheta_buclei} when condiţie;
```

În acest ultim caz, ieşirea din buclă nu are loc decât dacă respectiva condiţie este adevărată.

La fel ca în cazul instrucţiunii **next**, această instrucţiune, dacă nu are specificată o etichetă, nu se referă decât la bucla de nivelul cel mai de jos care o înglobează. Dacă prezintă o etichetă, atunci instrucţiunea **exit** permite ieşirea din bucla care poartă aceeaşi etichetă.

Iată un exemplu de utilizare a instrucţiunii **exit**:

```
variable R, M, N: INTEGER;
...
loop
      R := M mod N;
      M := N;
      N := R;
      exit when R = 0;
end loop;
```

2.10 Instrucţiunea *return*

Această instrucţiune este rezervată sub-programelor. La execuţia acestei instrucţiuni, sub-programul este imediat suspendat, controlul revenindu-i din nou apelantului.

O funcţie nu trebuie niciodată să fie executată până la cuvântul său cheie final **end**, căci aceasta ar provoca o eroare la execuţie. Orice funcţie se termină în mod dinamic prin instrucţiunea **return**, căreia i se asociază întotdeauna valoarea returnată.

```
return VALOARE;
```

Tipul lui VALOARE trebuie, bineînţeles, să fie cel declarat în specificaţia funcţiei. Într-o funcţie pot exista mai multe instrucţiuni **return** pentru că pot exista mai multe ramuri de decizie.

Instrucţiunea **return** poate fi utilizată pentru întreruperea cursului unei funcţii şi pentru a reveni în programul apelant. În acest caz, nu trebuie să i se asocieze nici o valoare.

```
return;
```

2.11 Instrucţiunea nulă

Sintaxa instrucţiunii nule este următoarea:

```
null;
```

Semantica acestei instrucţiuni este clară: se trece la executarea liniei de cod următoare. În practică, această instrucţiune este utilă pentru scrierea anumitor instrucţiuni de selecţie (**case**) atunci când toate cazurile trebuie luate în considerare, chiar dacă nu implică efectuarea nici unei acţiuni.

Instrucţiunea ;; care simbolizează instrucţiunea nulă în anumite limbaje de programare este interzisă în VHDL.

Instrucţiunea nulă nu este necesară la compilarea unui proces sau a unui corp de procedură „vid”. Următoarele porţiuni de cod sunt corecte din punct de vedere sintactic:

```
procedure P is -- Prima variantă
begin
end;
Q: process -- A doua variantă
begin
end process Q;
```

3. Desfăşurarea lucrării

3.1 Se va implementa exemplul prezentat în continuare, care gestionează traficul la o trecere simplă de pietoni. Dacă există erori în descriere, acestea se vor detecta şi corecta.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all
entity SEMAFOR is
        port(
             CLOCK, RESET, SENZOR1, SENZOR2: in STD_LOGIC;
             ROSU1, ROSU2, GALBEN1, GALBEN2, VERDE1, VERDE2: out STD_LOGIC;
        );
end SEMAFOR;

architecture SEMAFOR of SEMAFOR is
type STARE_T is (ST0, ST1, ST2, ST3, ST4, ST5, ST6, ST7);
signal STARE, NXSTARE: STARE_T;
begin
ACTUALIZEAZĂ_STARE: process (RESET, CLOCK)
        begin
                if (RESET = '1') then
                        STARE <= ST0;
                elsif CLOCK'EVENT and CLOCK = '1' then
                        STARE <= NXSTARE;
                end if;
        end process ACTUALIZEAZĂ_STARE;
TRANSITIONS: process (STARE, SENZOR1, SENZOR2)
        begin
        -- inițializări
                ROSU1 <= '0'; GALBEN1 <= '0'; VERDE1 <= '0';
                ROSU2 <= '0'; GALBEN2 <= '0'; VERDE2 <= '0';
                case STARE is
                        when ST0 => VERDE1 <= '1'; ROSU2 <= '1';
                              if SENZOR2 = SENZOR1 then NXSTARE <= ST1;
                              elsif (SENZOR1 = '0' and SENZOR2 = '1') then
                                        NXSTARE <= ST2;
                              else      NXSTARE <= ST0;
                              end if;
                        when ST1 => VERDE1 <= '1'; ROSU2 <= '1';
                                        NXSTARE <= ST2;
                        when ST2 => VERDE1 <= '1'; ROSU2 <= '1';
                                        NXSTARE <= ST3;
                        when ST3 => GALBEN1 <= '1'; ROSU2 <= '1';
                                        NXSTARE <= ST4;
                        when ST4 => ROSU1 <= '1'; VERDE2 <= '1';
                              if (SENZOR1 = '0' and SENZOR2 = '0') then
                                        NXSTARE <= ST5;
                              elsif (SENZOR1 = '1' and SENZOR2 = '0') then
                                        NXSTARE <= ST6;
                              else      NXSTARE <= ST4;
                              end if;
                        when ST5 => ROSU1 <= '1'; VERDE2 <= '1';
                                        NXSTARE <= ST6;
                        when ST6 => ROSU1 <= '1'; VERDE2 <= '1';
                                        NXSTARE <= ST7;
                        when ST7 => ROSU1 <= '1'; GALBEN2 <= '1';
                                        NXSTARE <= ST0;
                end case;
        end process TRANSITIONS;
end SEMAFOR;
```

3.2 Se va implementa următorul exemplu şi se va detecta dacă numărătorul comută pe frontul ascendent sau descendent al tactului. Se va defini interfaţa numărătorului.

```
NUMĂRĂTOR: process
variable COUNT: INTEGER := 0;
       begin
       wait until CLK = '1';
       while LEVEL = '1' loop COUNT := COUNT + 1;
                              wait until CLK = '0';
                       end loop;
end process NUMĂRĂTOR;
```