1. Noțiuni de descriere în VHDL, sinteză şi implementare în circuitele logice programabile

1. 1. Introducere

VHDL-ul este un limbaj de descriere hardware utilizat la descrierea comportamentului circuiteleor şi a sistemelor electronice. Acronimul VHDL provine de la VHSIC Hardware Description Language, iar VHSIC provenind de la Very High Speed Integrated Circuit. Limbajul a fost dezvoltat începând cu anii 80 de către Departamentul de Apărare al SUA. Prima versiune standardizată a fost în 1987, iar apoi revăzută şi îmbunătățită în anul 1993. Primul standard elaborat de IEEE (Institute of Electrical and Electronics Engineers) pentru limbajul VHDL a fost IEEE 1076, ulterior fiind adăugat şi standardul IEEE 1164.

Limbajul VHDL este utilizat atât la simularea circuitelor cât şi în sinteza acestora. De menționat faptul că toate construcțiile limbajului sunt simulabile dar nu toate sunt sintetizabile.

O motivație importantă în utilizarea unui limbaj HDL (VHDL sau Verilog) constă în faptul că nu depinde de tehologie, astfel secvențele de cod sunt reutilizabile pe circuite provenind de la fabricanți diferiți. Cea mai cunoscută aplicabilitate a limbajului VHDL este în proiectarea cu circuite CPLD (Complex Programmable Logic Devices), FPGA (Field Programmable Gate Arrays) şi ASIC (Application Specific Integrated Circuits). Odată scris un cod în VHDL acesta poate fi sintetizat şi implementat într-un circuit CPLD sau FPGA prvenind de la Altera, Xilinx, Atmel, etc sau poate fi trimis unui fabicant de circuite ASIC. În prezent multe circuite digitale complexe (ex. microcontrolerele) sunt proiectate utilizând un limbaj HDL.

1. 2. Sinopticul de proiectare în VHDL

În figura 1.1 sunt prezentate etapele de descriere a proiectelor în limbaj VHDL şi de implementare a acestora în circuite CPLD, FPGA sau ASIC. În prima fază se face descrierea proiectului în cod VHDL şi se salvează într-un fişier cu extensia .vhd, care va avea acelaşi nume cu cel declarat în cod după cuvântul cheie *entity.*

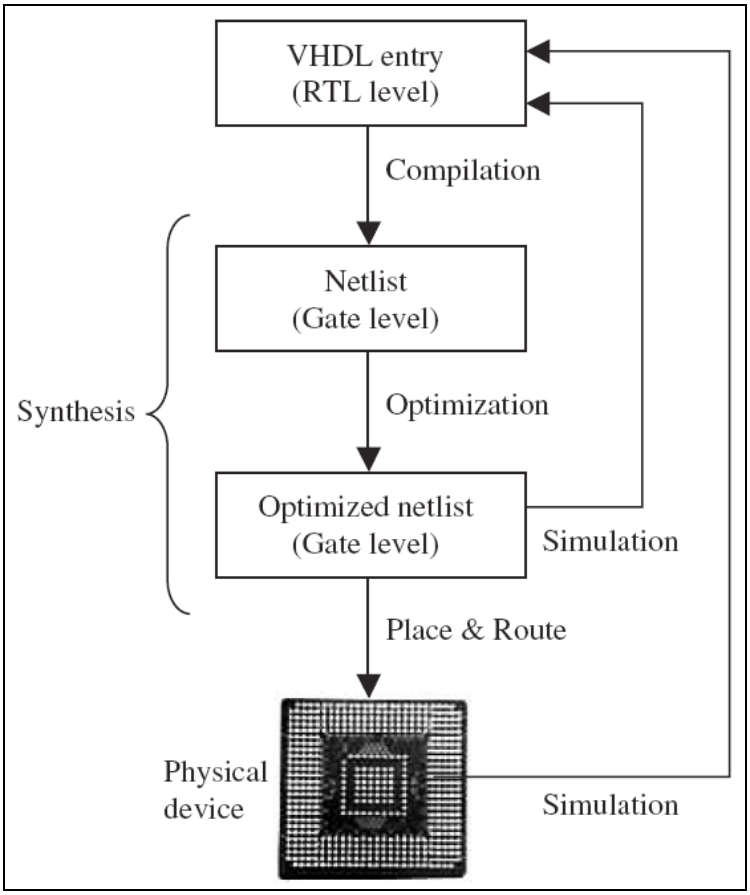


Figura 1.1 Sinopticul de proiectare în VHDL

Primul pas în procesul de sinteză este compilarea, care constă în conversia codului VHDL ce descrie circuitul la nivel RTL (Register Transfer Level) într-un format ce va descrie circuitul la nivel de poartă logică, aşa numita listă de conexiuni sau *netlist.* Cel de al doilea pas al sintezei constă în optimizarea circuitului descris la nivel de poartă logică, astfel încât el să funcționeze la frecvență maximă sau să ocupe o suprafață minimă în circuitul fizic în care va fi implementat. După aceste etape circuitul poate fi simulat. În ultima etapă un software de plasare/routare va genera layout-ul fizic pentru implementarea în CPLD/FPGA sau masca în cazul circuitelor ASIC.

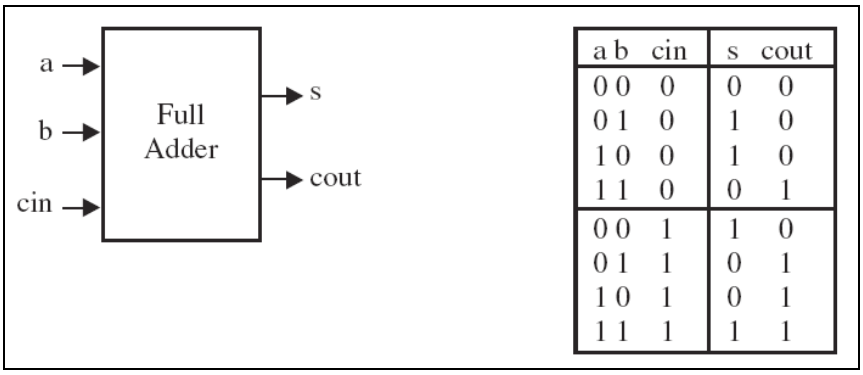
1. 3. Software de proiectare

Pe piață există mai multe software-uri de proiectare care utilizează limbajul VHDL, acestea mai sunt numite şi tools-uri EDA (Electronic Design Automation) şi acoperă întreg procesul de proiectare de la descriere şi simulare până la sinteză şi implementare. Unele dintre aceste programe de proiectare sunt oferite de producătorii de circuite CPLD/FPGA, astfel: Quartus II de la Altera, ISE de la Xilinx. Acesta permit sinteza şi implementarea doar pentru gama proprie de circuite. Există însă şi firme care oferă numai software pentru sinteză şi implementare, astfel de programe sunt: Leonardo Spectrum de la firma Mentor Graphics, Synplify de la firma Synplicity. Firma Model Tech parte a grupului Mentor Graphics oferă Model Sim, care este un software complex pentru simulare. O altă firmă cunoscută ce produce astfel de programe este Synopsys. Unul dintre avantajele utilizării unui program de proiectare produs de o firmă terță este faptul că acestea permit integrarea bibliotecilor cu componente tehnologice ale oricărui producător de circuite CPLD, FPGA şi ASIC, astfel că descrierea VHDL unui circuit va putea fi la orice moment sintetizată în tehnologia dorită.

1. 4. Conversia codului VHDL circuit fizic

În figura 1.2 este prezentat un sumator pe 1 bit împreună cu tabelul de adevăr al acestuia. Astfel, *a* şi *b* sunt intrările pe 1-bit, *cin* este intrarea de transport (carry in), *s* este ieşire şi reprezintă suma, iar *cout* este ieşirea de transport.

În figura 1.3 este prezentat codul VHDL corespunzător sumatorului pe 1-bit. Se poate observa că acesta este alcătuit dintr-un bloc *entity* în care se definesc pinii circuitului numărător (*port*) şi un bloc *architecture,* în care este definită funcționalitatea circuitului.



a b	cin	s	cout
0 0	0	0	0
0 1	0	1	0
1 0	0	1	0
1 1	0	0	1
0 0	1	1	0
0 1	1	0	1
1 0	1	0	1
1 1	1	1	1

Figura 1.2 Schema bloc a sumatorului pe 1-bit şi tabelul de adevăr

```
ENTITY sum IS
PORT (a, b, cin: IN BIT;
s, cout: OUT BIT);
END sum;
----------------------------
ARCHITECTURE sum_arch OF sum IS
BEGIN
s <= a XOR b XOR cin;
cout <= (a AND b) OR (a AND cin)
OR
(b AND cin);
END sum_arch;
```

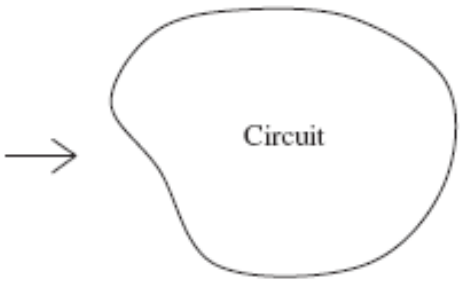


Figura 1.3 Exemplu de cod VHDL pentru sumatorul pe 1-bit

Codul VHDL prezentat în figura 1.3 va fi convertit într-un circuit, etapa de sinteză. Însă, există mai multe modalități de conversie în circuit a ecuațiilor descrise în arhitectură, astfel că circuitul final va depinde de software-ul de compilare/optimizare utilizat şi în egală măsură de tehnologia în care va fi implementat fizic. În figura 1.4 sunt prezentate câteva posibile rezultate ale conversiei din cod VHDL în circuit a sumatorului pe 1-bit.

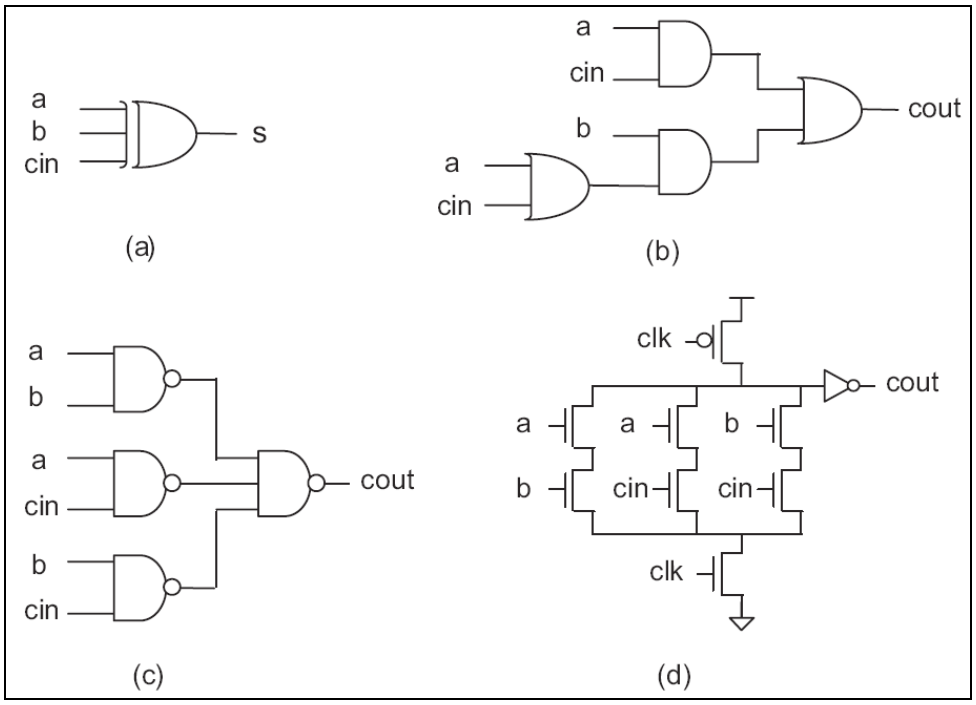


Figura 1.4 Variante de circuite rezultate în urma conversiei în circuit a codului VHDL ce descrie sumatorul pe 1-bit

Dacă dispozitivul final în care urmează să se facă implementarea este un circuit CPLD sau FPGA atunci două rezultate posibile ale conversiei ecuației corespunzătoare ieşirii *cout* sunt prezentate în figura 1.4, variantele b şi c. Dacă însă tehnologia țintă este ASIC, este posibil ca implementarea să se facă cu tranzistori CMOS respectând schema din figura 1.4, varianta d. Dacă în continuare se vor alege optimizări de viteză (frecvență de lucru mare) sau de arie (suprafață minimă pe siliciu), atunci forma finală a circuitului rezultat poate diferii.

Indiferent de forma finală a circuitului sintettizat, funcționarea acestuia trebuie verificată prin simulare. Se poate face verificare şi numia după implementarea fizică, dar corectarea eventualelor erori va fi mult mai costisitoare. În figura 1.5 sunt prezentate formele de undă rezultate în urma simulării funcționale a sumatorului pe 1-bit. După cum se poate observa pinii de intrare, definiți ca porturi în zona de arhitectură a codului VHDL sunt marcați cu o săgeată de forma  , iar pinii de ieşire sunt marcați cu săgeată de forma  . Starea stimulilor de la intrare poate fi stabilită în mod aleatoriu de către proiectant, dar starea ieşirilor trebuie întotdeauna să respecte valorile din tabelul de tabelul de adevăr, coloanele *s* şi *cout.*

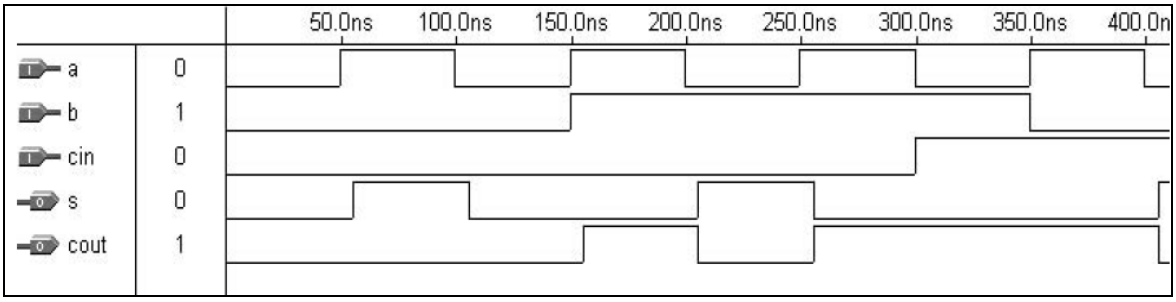


Figura 1.5 Rezultatul simulării codului VHDL ce descrie sumatorul pe 1-bit

1. 5. Nivele de abstractizare ale proiectării cu circuite digitale

Proiectarea cu circuite digitale se poate desfăşura la nivel de **tranzistor,** la nivel de **poartă logică**, la nivel de **registre** sau **RTL** (Register Transfer Level) şi la nivel **comportamental** (behavioral).

Nivelul cel mai de jos este cel de proiectare la nivel de tranzistor. La acest nivel tranzistoarele sunt conectate între ele pentru a alcătui structuri logice ca şi cele din figura 1.4.d.

Un nivel intermediar de proiectare este cel cu porți logice. La acest nivel se construiesc structuri de porți logice ca şi cele din figura 1.4.b şi c. Acest tip de proiectare se pretează pentru proiecte de dimensiuni mici în care componentele (sunt descrise din porți logice după care componentele sunt interconectate. Proiectarea la nivel de poartă logică se bazează pe tabele de adevăr sau pe ecuații booleane, proiectantul creând componentele combinaționale şi secvențiale standard sumatoare, multiplexoare, numărătoare, etc) pe care ulterior le va folosii în proiecte mai complexe. Acest tip de proiectare pune bazele proiectării ierarhice. Componentele standard create vor fi folosite la următorul nivel de proiectare, nivelul registru sau RTL. La acest nivel pot fi proiectate structuri de o complexitate ridicată, cum ar fi microprocesoarele. La nivel RTL proiectantul se concentrează pe modalitățile de transfer a datelor între registre şi alte blocuri funcționale ce fac parte dintr-un proiect.

Cel mai înalt nivel de abstractizare este cel comportamental, în care descrierea unui circuit se face utilizându-se un limbaj de descriere HDL (vezi codul din figura 1.3), proiectantul bazându-se strict pe definirea interfețelor circuitului (modulul entity) şi pe definirea comportării acestuia (modulul architecture).

1. 6. Sinteza proiectelor descrise în VHDL

Modul tradițional de proiectare cu circuite digitale se desfăşura la nivel de poartă logică sau în cel mai bun caz la nivel RTL. Datorită creşterii complexității proiectelor şi totodată a puterii de calcul şi dezvoltării software-ului de proiectare utilizarea limbajelor de descriere hardware (HDL) a devenit indispensabilă. Astfel că proiectații moderni de circuite digitale scriu linii de cod întocmai ca şi programatorii.

Odată dezvoltat un program utilizând un limbaj de nivel înalt acesta urmează a fi translatat în cod maşină specific procesorului ce urmează să-l execute. În această situație se foloseşte un software de compilare, iar fişierul rezultat în urma compilării va fi un fişier executabil. În situația în care un program este dezvoltat utilizând un limbaj de descriere hardware software-ul de compilare va înlocuit cu unul de **sinteză,** iar fişierul rezultat va fi un fişier netlist ce descrie un circuit digital. La fel cum procesul de translatare a codului unui limbaj de nivel înalt în cod maşină se numeşte compilare şi procesul de translatare din limbaj de descriere hardware în circuit digital se numeşte sinteză sau sintetizare.

Diferența care apare între cele două tipuri de programare în limbaj de nivel înalt şi limbaj de descriere hardware este că etapa de proiectare nu se încheie după faza de sinteză, ci continuă cu faza de mapare a porților logice în componente specifice unei tehnologii/producător, urmează faza de plasare/routare, faza de generare fişier de configurare şi în cele din urmă procesul de proiectare se încheie cu faza de programare (upload) a circuitului CPLD/FPGA.

Curs 6

Caracteristici avansate ale limbajului VHDL - part 1

1. **Subprograme**
2. **Supraîncărcare**
3. **Pachete**
4. **Vizibilitate**

Atunci când se elaborează modele comportamentale este util ca acestea să se împartă în secțiuni, astfel încât fiecare secțiune să trateze părți relativ independente ale comportamentului global al unui circuit. VHDL oferă posibilitatea folosirii conceptului de subprogram pentru a realiza acest lucru. In acest curs vom prezenta cele două tipuri de subprograme care se pot dezvolta în VHDL: proceduri şi funcții. în continuare, vom vedea ce înseamnă supraîncărcarea şi cum pot fi evitate posibilele conflictele generate de aceasta. Tot in legătură cu subprogramele, trebuie abordată problema folosirii numelor declarate în interiorul unui model. În acest scop, vom introduce ideea de vizibilitate a unei declarații.
Pachetele în VHDL oferă posibilitatea organizării datelor şi subprogramelor declarate într-un model. Vom descrie elementele de bază ale pachetelor şi vom arăta cum pot fi folosite. De asemenea, vom lua în discuție unele pachete predefinite care includ toate tipurile şi operatori predefiniți în VHDL.

1.1 Subprograme

În orice limbaj de programare se oferă posibilități sporite prin folosirea *funcțiilor* şi a *procedurilor.* În VHDL funcțiile şi procedurile sunt incluse în clasa generală a *subprogramelor. Funcțiile calculează şi returnează o valoare în momentul invocării unei expresii.* O funcție nu-şi modifică argumentele şi poate fi folosită numai într-o expresie. Procedurile au atât caracter secvențial cât şi concurent. Ele nu returnează o valoare la apelarea dintr-un program, şi îşi pot modifica argumentele. Următoarea arhitectură indică unde pot fi declarate subprogramele şi unde pot fi ele apelate.

```
architecture care_include_subprograme of nume_entitate is
-- subprogramele pot fi declarate aici
begin
  aici pot fi apelate subprogramele
end care_include_subprograme;
```

Subprogramele care sunt declarate în secțiunea de declarații a unei arhitecturi sunt vizibile numai în interiorul acelei arhitecturi. Ele pot fi de asemenea declarate în cadrul pachetelor şi vor fi vizibile pentru orice arhitectură prin intermediul unei instrucțiuni *use*. Subprogramele mai pot fi declarate în regiunile declarative ale proceselor, blocurilor şi chiar ale altor subprograme. Un subprogram este vizibil direct numai în interiorul construcției în care a fost declarat.

1.1.1 Funcții

Funcțiile pot fi declarate în VHDL specificând:

- numele funcției
- parametrii de intrare (dacă există)
- tipul valorii şi funcției returnate
- orice declarație cerută de funcția respectivă
- un algoritm pentru calculul valorii returnate

Formatul general pentru o declarație de funcție este:

```
function nume_funcție (declarații_parametrii_formali) return
    tip_return is
  — declarații de constante şi variabile
  -- aici nu sunt permise declarații de semnal begin
  — instrucțiuni secvențiale
    return (valoare_returnată); end
(nume_funcție);
```

în formalismul pe care l-am folosit până acum, regula de sintaxă pentru o funcție este:

function identificator
[(lista_parametrilor_de_interfață)] return marcă_tip is
{secțiune declarații}
begin
{instrucțiunisecvențiale}
end [function] [identificator];

Parametrii formali ai funcțiilor trebuie să fie în modul `in`. Deci, nu este necesar să se specifice modul parametrilor formali ai funcțiilor. Singurele clase de obiecte admise sunt constantele şi semnalele. Clasa implicită este `constant`. Pentru că parametrii formali sunt mereu în modul `in`, funcțiile nu au nici un efect secundar la apelare. Funcțiile returnează o valoare în expresia apelantă, dar nu modifică nici un parametru curent în acea expresie.

Variabilele şi constantele pot fi declarate în regiunea declarativă a funcțiilor. Variabilele din funcții sunt dinamice. Valoarea variabilelor se inițializează de fiecare dată când o funcție este apelată. Valorile lor nu sunt reținute între două apeluri. Pentru că o funcție specifică un

algoritm, toate instrucțiunile din corpul funcției trebuie să fie secvențiale. Pentru că la evocarea unei expresii trebuie să se returneze imediat o valoare, instrucțiunile `wait` nu sunt permise în corpul funcțiilor. Această restricție se aplică de asemenea oricărei funcții sau proceduri apelate de o funcție.

Într-un curs anteior am prezentat un corp arhitectural `MACRO` pentru entitatea `ONES_CNT`, unde s-au folosit funcțiile `MAJ3 (X)` şi `OPAR3 (X)`. Pentru a folosi aceste funcții ele trebuie să fie mai întâi declarate în modul următor:

```
function MAJ3(X: Bit_Vector(0 to 2))  return Bit is
begin
     return ((X(0) andX(l)) or (X(0) andX(2)) or (X(l) andX(2));
end MAJ3;
```

Parametrul `X` are clasa implicită `constant.` Valoarea returnată este de tipul `Bit.` `Valoarea` returnată poate fi specificată de o expresie, cum este cazul de `mai` sus, sau poate fi `calculată` prin executarea unei secvențe de instrucțiuni. După cum am arătat, pot fi făcute `declarații` locale dar ele nu sunt necesare în acest caz simplu. Funcțiile se utilizează ca părți lie unor expresii. Aşadar, funcția `MAJ3` a fost invocată în arhitectura `MACRO` pentru ONES_CNT după cum urmează:

```
C(1)  <= MAJ3(A);
```

Un exemplu tipic de declarație de funcție în VHDL este:

```
function rising_edge (signal clock: in std:logic) return boolean;
```

Structura acestei funcții respectă structura generală:

```
function  rising_edge    (signal  clock:  in  std_logic)
   return boolean  is
  – declarații variabile locale ale funcției begin
  – instrucțiuni secvențiale
   return (valoare_returnată);
end function rising_edge;
```

Funcția are un nume `(rising_edge)` şi un set de parametrii. Parametrii din definiția funcției sunt numiți *parametrii formali.* Când o funcție este apelată, argumentele care apar în apel sunt *parametrii actuali.* Funcția din acest exemplu poate fi apelată prin:

```
rising_edge  (enable);
```

Parametrul actual este semnalul `enable;` el ia locul parametrului formal `clock` din corpul funcției.

Exemplu. Detecția evenimentelor din semnale

De multe ori este util să realizăm teste asupra unor semnale pentru a determina dacă au avut loc anumite evenimente. De exemplu, *detecția frontului crescător* este foarte utilă **în** modelarea sistemelor secvențiale (automate Moore şi Mealy). În Figura 1 este prezentat modelul pentru un flip-flop de tip D care comută pe frontul crescător al semnalului de ceas. A fost inclusă o funcție pentru testarea acestui front, şi nu s-a ales varianta de a scrie codul funcției direct în corpul arhitectural (Figura 2). Observați locul în care s-a plasat funcția, în porțiunea declarativă a arhitecturii (Figura 5.2). În mod normal, această regiune este folosită pentru a declara semnale şi constante folosite în corpul arhitecturii. Deci, am putea să declarăm funcții (sau proceduri) care sunt folosite şi în arhitectură. Funcția ar putea fi declarată şi în regiunea declarativă a procesului care apelează funcția (adică, între cuvintele cheie begin şi end). Problema este dacă dorim să avem funcția vizibilă pentru (şi deci apelabilă din) toate procesele din corpul arhitectural, sau vizibilă numai pentru un proces.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity dff is
port (D, clk: in std_logic;
  Q, QN: out std_logic);
end entity dff;

architecture behavioral of dff is
  function rising_edge (signal clock: std_logic) return boolean is
  variable edge: boolean := false;
  begin
  edge := (clock = '1' and clock'event); return (edge);
  end function rising_edge;

begin
   output: process is begin
        wait until (rising_edge(clk));
        Q <= D after 5 ns;
        QN <= not Q after 5 ns;
   end process output;
end architecture behavioral;
```

Flg. 1 Un exemplu de folosire a funcțiilor pentru un flip-flop de tip D.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity dff is
port (D, elk: in std_logic;
  Q, QN: out std_logic);
 end entity dff;
architecture behavioral of dff is
begin
  output: process is
  begin
```

```
        wait until (clk'event and clk = '1');
          Q <= D after 5 ns;
          QN <= not Q after 5 ns;
    end process output;
    end architecture behavioral;
```

Fig. 2 Un model comportamental pentru un flip-flop de tip D care comută pe frontul pozitiv al semnalului de ceas.

Exemplu

În Figura 3 se prezintă o funcție care calculează dacă o valoare este între anumite limite şi returnează un rezultat limitat de cele două valori.

```
function limit ( value, min, max : integer ) return integer is
begin
  if value > max then
    return max;
   elsif value < min then
    return min;
   else
    return value;
  end if;
  end function limit;
```

Fig. 3 Un exemplu de funcție folosită pentru a limita o valoare între două extreme specificate.

Un apel al acestei funcții poate fi inclus într-o instrucțiune de asignare de variabilă după cum urmează:

```
new_temperature := limit(current_temperature + increment, 10, 100);
```

În această instrucțiune, expresia din partea dreaptă constă dintr-un simplu apel de funcție, şi rezultatul returnat este asignat variabilei new_temperature.

1.2 Proceduri

Procedurile sunt subprograme care pot modifica unul sau mai mulți parametri de intrare. Procedurile pot fi declarate specificând:

- numele procedurii
- parametrii de intrare şi cei de ieşire (dacă există)
- orice declarație cerută de procedura respectivă
- un algoritm

Formatul general pentru o declarație de procedură este:

```
procedure nume_procedură  (declarații_parametrii_formali)
  -- partea de declarații a procedurii
  -- declarații de constante şi variabile
  -- aici nu sunt permise declarații de semnal begin
  -- instrucțiuni secvențiale
  end (nume procedură);
```

În formalismul pe care l-am folosit până acum, regula de sintaxă pentru o procedură este:

procedura identificator [(lista_parametrilor_de_interfaţă)] is
{secţiune declaraţii}
begin
{instrucţiuni_secvenţiale}
end [procedura] [identificator];

Parametrii formali pot fi în modurile `in, out` sau `inout.` Dacă modul nu se specifică, se presupune implicit modul `in.` Clasele de obiecte acceptate sunt `constant,` variable, `signal.` Dacă modul este in şi nu se indică o clasă de obiecte, atunci se presupune implicit iasa `constant.` Dacă modul este `out` sau `inout,` clasa implicită este variable.

Variabilele şi constantele pot fi declarate în secțiunea de declarații; semnalele nu pot fi declarate aici. Variabilele din proceduri sunt dinamice; ele sunt inițializate de fiecare dată când procedura este apelată şi nu rețin valorile între două apeluri. In corpul procedurii se folosesc instrucțiuni secvențiale pentru a specifica algoritmul implementat (după cuvântul cheie begin). Spre deosebire de funcții, procedurile pot conține instrucțiuni `wait.` Totuşi, dacă o procedură este apelată de o funcție, ea nu trebuie să conțină instrucțiuni `wait.` O procedură nu returnează o valoare la o apelare; ea modifică unul sau mai mulți din parametrii a formali. Aceasta înseamnă că parametrul actual care a fost identificat cu parametrul formal `ifn` expresia apelantă va fi modificat în mediul apelant. Dacă parametrul formal este din clasa `signal,` semnalul nu se actualizează pe durata ciclului curent de simulare; este programat pentru actualizare într-un ciclu viitor de simulare.

`In` Figura `4` se prezintă o declarație pentru o procedură `ADD` care realizează suma a doi bit vectori. Parametrii formali `A, B, CIN` sunt în modul `in;` deci, ei pot fi numai citiți de procedură. Procedura nu poate asigna noi valori parametrilor formali care sunt în modul in. Pentru că nu se specifică o clasă pentru parametrii formali `A, B, CIN,` aceştia se presupun din `clasa constant.` Parametrii formali `SUM` şi `COUT` sunt în modul `out;` deci, lor li se pot turnai asigna valori de către procedură. Procedura nu poate citi valoarea curentă `a` parametrilor formali în modul `out.` Pentru că nu se specifică o clasă, parametrii `SUM` şi `COUT` au clasa implicită variable.

```
procedure ADD(A,B: in BIT_VECTOR; CIN: in BIT;
              SUM: out BIT_VECTOR; COUT: out BIT) is
```

```
    variable SUMV,AV,BV: BIT_VTCTOR(A'LENGTH-1 downto 0);
    variable CARRY: BIT;
  begin
    AV := A;
    BV := B;
    CARRY := CIN;
    for I in 0 to SUMV'HIGH loop
      SUMV(I)  := AV(I) xor BV(I) xor CARRY;
      CARRY := (AV(I) and BV(I)) or (AV(I) and CARRY) or (BV(I) and CARRY);
  end loop;
  COUT := CARRY;
  SUM := SUMV;
  end procedure ADD;
```

Fig. 4 Procedura de adunare a doi bit vectori.

Instrucțiunea care asignează noi valori lui `SUM` şi `COUT` foloseşte operatorul :=. Noile valori au efect imediat pe durata ciclului curent de simulare. Sunt declarate mai multe variabile interne procedurii, care vor fi vizibile numai în interiorul acesteia. Aceste variabile sunt dinamice şi sunt inițializate ori de câte ori procedura este apelată. Pentru că nu se specifică valori inițiale, valorile inițiale implicite sunt '0' pentru variabilele de tip `Bit` şi un vector zero pentru toate variabilele de tip `Bit_Vector`. Această implementare pentru operația de adunare funcționează pentru bit vectori cu game descendente sau ascendente, cu orice lungime. Aceasta se realizează folosind variabile interne de tipul `BitVector (A'length-1 downto 0)` şi folosind `SUMV'high` în specificarea gamei `loop`. Se presupune că bitul cel mai puțin semnificativ este în dreapta. Fiecare trecere prin buclă implementează ecuațiile unui sumator complet. Aplicarea repetată a buclei conduce la adunarea mai multor biți.

În general, procedurile pot fi apelate sub formă de instrucțiuni concurente în corpurile arhitecturale sau ca instrucțiuni secvențiale în procese, funcții sau alte proceduri. Procedura `ADD` de mai sus poate fi folosită numai ca instrucțiune secvențială în procese, funcții sau alte proceduri pentru că ieşirile `SUM` şi `COUT` au obiecte din clasa `variable` şi, prin urmare, acestea trebuie să fie identificate cu valori actuale din clasa `variable`. Pentru că variabilele nu pot fi declarate în regiunile declarative ale arhitecturii, procedura `ADD` nu poate fi folosită ca instrucțiune concurentă într-un corp arhitectural.

În Figura 5 se prezintă o declarație pentru o procedură care calculează media unor valori memorate într-un tablou numit `samples` şi asignează rezultatul unei variabile numită `average`. Această procedură are o variabilă locală total pentru a reține suma elementelor tabloului. Spre deosebire de variabilele din procese, variabilele locale ale procedurilor suni create şi inițializate de fiecare dată când procedura este apelată.

```
architecture test of fg_07_01 is
procedure average_test is
    variable average : real := 0.0;
    type sample_array is array (positive range <>) of real;
    constant samples : sample_array :=
       ( 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0 );
```

```
procedure average_samples is
      variable total : real := 0.0;
      begin
      assert samples'length > 0 severity failure;
      for index in samples'range loop
        total := total + samples(index);
       end loop;
    average := total / real(samples'length); end procedure
  average_samples; begin
  average_samples; end procedure average_test;
begin
  average_test; end architecture test;
```

Fig. 5. O declarație pentru o procedură de calcul a valorii medii.

Actiunile procedurii sunt declasate prin instrucțiunea de apelare, care este o altă instrucțijne secvențială în VHDL. O procedură fără parametrii este apelată simplu scriind numele ei:

instrucţiune_apelare_procedură <= [etichetă:] nume_procedură;

De exemplu, putem include următoarea instrucțiune într-un proces:

```
average_samples;
```

Efectul acestei instrucțiuni este apelarea procedurii averagesamples. Aceasta însemnă crearea şi inițializarea unei noi instanțe a variabilei locale total, apoi executarea instructiunilor din corpul procedurii. După executarea ultimei instrucțiuni a procedurii, controlul se transferă înapoi la programul apelant şi se va executa următoarea instrucțiune de aici.

2 Supraîncărcare

În VHDL există posibilitatea dea modifica semnificația variabilelor şi numele operatorilor, funcțiilor şi procedurilor prin *redeclarări.* Acest proces se numeşte *supraîncărcare.* Să considerăm următoarea instrucțiune de asignare de semnal:

```
F <= (A and B) or (C and D);
```

Să presupunem că A, B, C, D, F au fost declarate inițial de tipul Bit, adică s-a folosit logica binară. Mai târziu, s-a dorit să se modeleze aceeaşi ecuație logică folosind **un** tip cu patru valori logice conoscut sub numele MVL4; acesta are valorile 'X', '0', '1', 'Z'. Putem redeclara A, B, C, D şi F să fie de tipul MVL4. Dacă facem acest lucru, supraîncărcăm valorile literale ' 0 ' şi ' 1' pentru că acestea sunt de asemenea şi în tipul Bit. Totuşi, dacă am declarat A, B, C, D şi F de tipul MVL4, operatorii AND şi OR nu se mai pot aplica pentru că ei au fost construiți numai pentru tipurile Bit şi Boolean. Se pot defini două funcții, >ÍVL4_AND (X, Y) şi MVL4_OR (X, Y), pentru a

realiza operațiile indicate, dar aceasta necesită rescrierea ecuației folosind notația de apelare a funcțiilor:

```
F <= MVL4_OR(MVL4_AND(A,B), MVL4_AND(C,D));
```

Apar totuşi câteva probleme. Pentru expresii booleene complicate, această formă funcțională este dificil de citit şi de scris, iar scrierea ecuațiilor booleene în această formă poate să conducă uşor la erori. O metodă mai bună este supraîncărcarea semnificației operatorilor AND şi OR. Se poate declara un nou operator AND, după cum se arată în Figura 6.

În zona de declarații a funcției, se declară un tablou 2D. Elementele acestuia au doi indici şi unt de tip MVL4. Tabelul de adevăr pentru funcția AND este declarat sub formă de constantă. Funcția returnează numai valoarea accesată din table. Considerând declarația acestei funcții, se pot scrie expresii cum ar fi:

```
F <= A and B;
```

cu condiția ca F, A, B să fie de tip MVL4. Dacă din anumite motive se doreşte folosirea notația de apel de funcție, aceeali expresie se scrie sub forma:

```
F <= "and"(A, B);
```

```
function "and" (L, R: MVL4) return MVL4 is
-- Declare a two-dimensional table type.
  type MVL4_TABLE is array (MVL4, MVL4) of MVL4;
  --    truth table for "and" function constant table_AND:
  MVL4_TABLE :=
  -- X    0     1   Z
    (('X', '0','X', 'X'),      --X
    ('0','0','0','0'),    --0
    ('X',  '0','1', 'X'),    --1
     ('X','0','X','X')); --Z
begin
  return table_AND(L, R);
end function "and";
```

Fig. 6. Supraîncărcarea operatorului AND

```
function INTVAL (VAL: MVL4_VECTOR) return
INTEGER is variable SUM: INTEGER:= 0;
begin
  for   N in VAL'LOW to VAL'HIGH loop
```

```
        assert not(VAL(N) = 'X' or VAL(N) = *Z')
        report "INTVAL inputs not 0 or 1"
        severity WARNING;
        if   VAL(N) = '1' then SUM := SUM + (2**N);
      end if ; end loop; return
    SUM; end function INTVAL;
```

Fig. 7. Declarația unei funcții pentru conversia tipului MVL4_VECTOR la tipul integer

Apare o întrebare normală: Mai este accesibil operatorul obişnuit AND pentru a fi folosit cu obiecte de tip Bit sau Boolean? Răspunsul este DA. Modulul VHDL de analiză poate determina care operator AND este necesar într-o expresie prin examinarea profilelor parametrilor şi rezultatului pentru operații care au aceleaşi nume. Profilul parametrulu: specifică numărul, ordinea şi tipul parametrilor operanzi. Profilul rezultatului specifică tipul rezultatului returnat. Folosind aceste profiluri, modulul de analiză poate face diferența dintre diferiți operatori. De exemplu, tipul celor doi parametrii pentru funcția standard AND sun Bit sau Boolean. Pentru MVL4 cei doi operanzi sunt de tip MVL4.

Numele subprogramelor pot fi de asemenea supraîncărcate. De exemplu, să considerau funcția din Figura 7 care converteşte un MVL4_VECTOR la un întreg.

Funcția verifică mai întâi dacă biții sunt ' X' sau ' Z'. Presupunând că vectorul este alcătuit din '0' şi '1', funcția adună puterea lui doi potrivită pentru fiecare valoare '1' detectați în această funcție se presupune că bitul cel mai puțin semnificativ al cuvântului de convertit are indexul cel mai mic.

```
function INTVAL(VAL:BIT_VECTOR) return INTEGER is
  variable SUM: INTEGER:=0; begin
  for N in VAL'LOW to VAL'HIGH loop if VAL(N)=*1'
    then
  SUM := SUM + (2 ** N); end if; end loop;
return SUM; end function INTVAL;
```

Fig. 8. Declarația unei funcții pentru conversia tipului MVL4_VECTOR la tipul integer.

```
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_SIGNED.all;.
entity ADD_OVERLOAD is
port ( A, B, C: in   STD_LOGIC_VECTOR (7 downto 0);
SUM: out STD_LOGIC_VECTOR (7 downto 0));
end entity ADDjOVERLOAD;
architecture PACK_SIGNED of ADD_OVERLOAD is
begin
SUM <=   A + B + C; —Aceasta este acum o adunare
```

```
                        — in complement față de 2
    end architecture PACK_SIGNED;
```

Fig. 9. Supraîncărcarea ADD cu pachet cu semn.

Să presupunem că este declarată o altă funcție de conversie, aceea din Figura 5.9. Aceasta converteşte un parametru de tip Bit_Vector la un Integer. Pentru că numele celor două funcții sunt aceleaşi, tipul parametrilor de apelare va conduce la funcția corectă.

Mai poate apărea şi o altă problemă: dacă două subprograme sunt declarate cu nume identice şi profile ale parametrilor şi rezultatelor identice, care anume va fi vizibil? Răspunsul este că cea mai recentă declarație o ascunde pe cea precedentă.

Puterea supraîncărcării este cel mai bine ilustrată dacă arătăm modul în care pachetele IEEE pot fi folosite pentru a supraîncarcă operații. Mai întâi, toate operațiile booleene sunt supraîncărcate pentru tipurile std_logic şi std_logic_vector. Apoi, operatorii aritmetici pot fi supraîncărcați. în Figura 9, cele trei intrări ale entității sunt de tip std_logic_vector. Totuşi, pentru că se foloseşte pachetul std_logic_signed, operatorul supraîncărcat + din acest pachet interpretează intrările ca fiind numere cu semn şi realizează operații în complement față de doi şi returnează un rezultat de tip std_logic. Dacă se foloseşte în schimb pachetul std_logic_unsigned, operatorul supraîncărcat + din acest pachet realizează adunarea fără semn.

Numele subprogramelor supraîncărcate trebuie fie să aibă un număr diferit de parametrii sau cel puțin unul dintre parametrii formali trebuie să aibă un tip diferit de date. De exemplu, pot exista trei versiuni supraîncărcate ale unui flip-flop D, după cum se arată în următoarele apeluri de proceduri, unde toți parametrii formali sunt presupuşi de tip Bit:

```
DFF (CLK, D, Q);
DFF (CLK, D, Q, QN) ;
DFF (CLK, D, CLEAR, PRESET, Q);
```

Procedurile supraîncărcate următoare sunt invalide (se presupune că toți parametrii formali au tipul Bit):

```
DFF (CLK, D, CLEAR, Q, QN) ;
DFF (CLK, D, PRESET, Q, QN) ;
```

pentru că numărul parametrilor formali este acelaşi şi toți parametrii au acelaşi tip.

3 Pachete (package)

Scrierea unui program în care anumite declarații sunt scrise de fiecare dată când este nevoie de ele este o activitate anostă. VHDL foloseşte un *pachet (package)* pentru a reține declarațiile frecvente; fiecare pachet are un nume. Declarațiile din pachet pot fi făcute visibile prin referirea la numele pachetului. Totuşi, mai întâi trebuie făcută declarația ăachetului. Regula de sintaxă pentru scrierea unei declarații de pachet este:

declarație_de_pachet <=
package identificator is

{parte_declarativă_a „pachetului}
end [package] [identificator];

Identificatorul furnizează un nume pentru pachet, nume care se poate folosi oriunde într-un model pentru a face referire la pachetul respectiv. In interiorul declarației de pachet se scriu declarații care includ tipuri, subtipuri, constante, semnale şi subprograme.

Un exemplu de declarație de pachet este prezentat în Figura 10.

Un pachet reprezintă o altă formă de unitate de proiectare, împreună cu declarațiile de entitate şi corpurile arhitecturale. El este analizat separat şi plasat în biblioteca de lucru ca unitate separată de modulul de analiză VHDL. De acolo, alte unități ale bibliotecii pot face referire la un element declarat în pachet folosind numele selectat al elementului. Numele selectat este format prin scrierea numelui bibliotecii, apoi numele pachetului, apoi numele elementului, toate separate de puncte; de exemplu:
work.cpu_types.status_value;

```
work.cpu_types.status_value;
  package cpu_types is
    constant word_size : positive := 16;
    constant address_size : positive := 24;
    subtype word is bit_vector(word_size - 1 downto 0);
    subtype address is bit_vector(address_size - 1 downto 0);
    type status_value is ( halted, idle, fetch, mem_read, mem_write,
                        io_read, io_write, int_ack );
  end package cpu_types;
```

Fig. 10. Un pachet care declară unele constante şi tipuri utile pentru o unitate centrală de prelucrare

Să presupunem că pachetul cpu_types din Figura 10 a fost analizat şi plasat în biblioteca work. Putem folosi elementele declarate atunci când modelăm un decodificator de adrese pentru CPU. Declarația de entitate şi corpul arhitectural pentru decodificator sunt prezentate în Figura 11.

Se observă că trebuie să folosim nume selectate pentru a face referire la subtipul adddress, tipul status_value, literalii enumerație din status_value şi la operatorul
declarat implicit, toate definite în pachetul cputypes. Aceasta deoarece ele nu sunt rect vizibile din interiorul declarației de entitate şi al corpului arhitectural.

În Figura 12 este prezentat un alt exemplu de pachet, HANDY, care declară subtipurile şi interfața pentru o funcție. Codul pentru funcție este dat în corpul pachetului. Dacă acesta nu conține subprograme, nu este nevoie de un corp arhitectural pentru pachet.

```
entity address_decoder is
port ( addr : in work.cpu_types.address;
        status : in work.cpu_types.status_value;
       mem_sel, int_sel, io_sel : out bit );
       end entity address_decoder;
```

```
architecture functional of address_deccder is
constant mem_low : work.cpu_types.address := X"000000";
constant mem_high : work.cpu_types.address := X"EFFFFF";
constant io_low : work.cpu_types.address : = X"F00000";
constant io_high : work.cpu_types.address := X"FFFFFF";
begin
mem_decoder :
   mem_sel <= '1' when (work.cpu_types."="(status,work.cpu_types.fetch) or
     work.cpu_types."="(status, work.cpu_types.mem_read) or work.cpu_types."="(status,
     work.cpu_types.mem_write) ) and addr >= mem_low and addr <= rtem_high else '0';
int_decoder :
   int_sel <=       when work.cpu_types."="(status,work.cpu_types.int_ack) else
             '0';
io_decoder :
   io_sel <= '1' when (work.cpu_types."="(status,work.cpu_types.io_read) or
     work.cpu_types."="(status, work.cpu_types.io_write)) and addr >= io_low and addr <=
     io_high else '0';
end architecture functional;
```

Fig. 11. Entitatea şi corpul arhitectural pentru un decodificator de adrese, folosind elemente declarate în pachetul cpu_types

```
   package HANDY is
     subtype BITVECT3 is BIT_VECTOR (0 to 2);
     subtype BITVECT2 is BIT_VECTOR (0 to 1);
     function MAJ3 (X: BIT_VECTOR (0 to 2)) return BIT;
    --- Alte declaraţii
   end HANDY;
   package body HANDY is
     function MAJ3 (X: BITjVECTOR (0 to 2))
       return BIT is begin
       return (X(0) andX(1)) or (X(0) andX(2)) or (X(1) andX(2));
     end function MAJ3;
    -  Alte declaraţii de subprogram
   end HANDY;
```

Fig. 12. Definirea unui pachet

Dacă se dă definiţia pachetului HANDY, să presupunem că dorim să dăm acces la pachet unei entităţi pe nume LOGSYS. Se poate face aceasta prin plasarea clauzei use înainte de descrierea de interfaţă pentru entitatea LOGSYS:

```
 use work:HANDY.all;
 entity LOGSYS is
port   (X:  in BITVECT3; Y:  out BITVECT2);
end entity LOGSYS;
```

Toate declarațiile conținute în HANDY sunt acum vizibile în entitatea LOGSYS, incluzând oricare din corpurile sale arhitecturale. Vom discuta despre vizibilitate ceva mai târziu în acest capitol.

Forma generală pentru o instrucțiune use este:

use nume_bibliotecă.nume_pachet.nume_element;

unde **nume_bibliotecă** este numele unei biblioteci care conține pachetul, nume_pachet este numele pachetului, iar nume_element este numele elementului specific din pachet, de exemplu,

```
use my_library.my_pckage.dff;  -- se preia o componentă
use my_library.my_package.all; --se preia totul din pachet
```

În exemplul HANDY, cuvântul cheie work se referă la biblioteca implicită care este asociată cu proiectul curent. Biblioteca conține toate pachetele, procedurile, componentele, etc, care au fost analizate ca parte a proiectului curent.

Limbajul VHDL defineşte un pachet STANDARD care poate fi folosit de toate entitățile. Printre alte lucruri, acest pachet conține definițiile pentru tipurile Bit, BitVector, Boolean, Integer, Real, Character, String, Tipe, ca şi pentru subtipurile Positive şi Natural.

IEEE a dezvoltat Standard 1164 ca un sistem standard cu nouă valori logice, care are aplicații particulare la sinteza circuitelor logice. Comitetul IEEE a elaborat două pachete: (1) std_logic_1164, care defineşte sistemul de valori de bază şi funcțiile asociate (este vândut ca atare de firmele comerciante) şi (2) numeric_std, care oferă elementele supraîncărcate de aritmetică şi alți operatori, pentru sinteză. Diferite companii vânzătoare au elaborat propriile versiuni ale acestui pachet. în particular, Synopsis a dezvoltat :

1. std_logic_arith, care defineşte două noi tipuri de vectori signed şi unsigned precum şi elementele de aritmetică şi operatorii relaționali;
2. std_logic_unsigned pentru elementele de aritmetică şi operatorii relaționali pentru obiecte de tip std_logic_vector care sunt interpretate ca numere fără semn,
3. std_logic_signed, pentru elementele de aritmetică şi operatorii relaționali pentru obiecte de tip std_logic_vector care sunt interpretate ca numere cu semn (complement față de doi).

4. Vizibilitate

In descrierea limbajului VHDL de până acum am dat numeroase exemple de declarații de puri, obiecte şi subprograme, dar nu am dat definiții exacte pentru locurile unde aceste elemente sunt declarate. Vom face acest lucru în continuare.

Regiune. O porțiune de text mărginită, continuă din punct de vedere logic.

Regiune de declarații. O regiune în care se poate folosi un nume pentru a face referire fără ambiguitate la un element declarat.

După ce un element a fost declarat într-un anumit punct dintr-o regiune de declarații, se rune că numele lui este *vizibil* până la sfârşitul regiunii de declarații. Efectul practic este acela că un element trebuie declarat înainte de a fi folosit. Numele sunt în mod normal vizibile numai

în interiorul regiunii unde ele sunt declarate, adică sunt *direct vizibile.* Totuşi, după cum vom vedea, numele pot fi făcute vizibile şi în afara regiunilor proprii de declarații prin *biblioteci* şi clauze *use.* Acest al doilea tip de vizibilitate se numeşte *vizibilitate prin selecție.*

Vom enumera următoarele construcții, care stabilesc regiuni de declarații. Vom împărți :este construcții în două categorii: *regiuni generale de declarații* care permit o gamă largă de declarații şi *regiuni specializate de declarații* care permit numai un set restrâns de tipuri de declarații.

Construcțiile care stabilesc o regiune generală sunt:

- o declarație de entitate şi o arhitectură asociată
- un proces
- un bloc
- un subprogram
- un pachet

În Figura 13 se prezintă un exemplu în care semnalele şi variabilele sunt declarate cu numele X în cinci regiuni declarative diferite: un pachet SIG, o entitate Y şi arhitectura ei Z, o funcție R, un bloc inclus B şi un proces în interiorul arhitecturii Z. Pentru că fiecare din aceste declarații este locală, ele pot fi executate fără conflict. In fiecare caz, numele X este direct vizibil până la sfârşitul regiunii de declarații în care se declară X.

Semnalul X (X = 1) declarat în pachetul SIG este vizibil direct pe tot cuprinsul pachetului SIG. Semnalul X (X = 2) declarat în entitatea Y, este direct vizibil pe tot cuprinsul entității Y şi al arhitecturilor ei. Deci, lui Z2 i se asignează valoarea 2. Instrucțiunea *return X* din funcția R. returnează valoarea lui X = 3 care este direct vizibilă pentru funcția R. Deci, lui Z3 i se asignează valoarea 3. Valoarea lui Z5 este 5 pentru că ea este asignată lui X (X = 5) care este direct vizibil pentru procesul PI.

În trei cazuri numele X este făcut vizibil prin selecție în afara regiunii declarative în care a fost declarat:

```
package SIG is
  signal X: INTEGER:= 1;
end package SIG;

use work.SIG.all; entity Y is
  signal X: INTEGER:= 2;
end entity Y;

architecture Z of Y is
  signal Z1,Z2,Z3,Z4,Z5: INTEGER = 0;
  function R return INTEGER is
    variable X: INTEGER := 3;
    begin
  return X; — Returns value of 3.
  end R;
  begin
  B: block
  signal X: INTEGER := 4; s
  ignal Z6: INTEGER := 0; begin
    Z6 <= X + Y.X;   -- Z6 = 6 end block B;

PI: process
```

```
                    variable X: INTEGER :=5;
                    begin
                   Z5 <= X;        -- Z5=5
                   wait;
                   end process PI;

                 Zl      <=      work.SIG.X;      --      Zl=l
                 Z2 <= X;          -- Z2=2
                 Z3 <= R;          -- Z3=3
                 Z4 <= B.X;        -- Z4=4
end architecture Z;
```

Fig. 12. Regiuni declarative

1. Numele pachetului este făcut vizibil prin instrucțiunea "use work.SIG". Apoi, num selectat SIG.X este folosit pentru a face referire la valoarea X din pachetul SI Valoarea acestui semnal (X = 1) este asignată lui Zl.
2. Numele selectat B.X este folosit pentru a face referire la valoarea lui X în interiori blocului B. Valoarea acestui semnal (X = 4) se asignează lui Z4.
3. In interiorul blocului B, numele selectat Y.X este folosit pentru a face referire la valoarea lui X declarată în entitatea Y. Valoarea acestui semnal (X = 2) este adunată a. valoarea locală a lui X (X = 4) care este direct vizibilă pentru blocul B, iar rezultata (6) este asignat lui Z6.

In fiecare caz, numele selectat este de forma P.S, unde P este prefixul care arată construcția în care apare declarația numelui, iar S este sufixul care este numele declarat. Selecția numelui se poate realiza folosind un număr nelimitat de niveluri incluse, de exemplu, un nume selectat de forma P1.P2.P3.S indică un nume declarat S din interiorul lui P3, care este inclusă în P2, are este inclusă în PI.

Dacă se simulează arhitectura Z a entității Y, se va găsi că semnalului Zi i se asignează valoarea i. Următoarele construcții stabilesc regiuni declarative speciale:

1. Tipuri înregistrare: numele câmpurilor de înregistrare sunt declarate în mod implicit.
2. Bucle: se declară implicit variabila de control ce reprezintă indexul buclei
3. declarații de componente: sunt implicit declarate numele componentei, numele porturilor şi numele generice.
4. Configurații: declarațiile de configurație au o regiune de declarații pentru specificarea atributelor şi a clauzelor use.

Curs 7

Caracteristici avansate ale limbajului VHDL – part 2

1. **Biblioteci**
2. **Configurații**
3. **Fişiere I/O**

În continuare, vom arăta cum pot fi folosite şi/sau dezvoltate bibliotecile, configurațiile şi fişierele de intrare/ieşire.

1. Biblioteci

Atunci când modelele VHDL sunt analizate fără erori, rezultatul va fi salvat într-o bibliotecă. Diferitele biblioteci pe care le creează grupuri de proiectare reține starea proiectelor pe care acel grup le elaborează. Existența acestor biblioteci permit ca modelele existente VHDL să fie folosite în descrieri VHDL ulterioare. După cum vom arăta, modelele rezidente într-o bibliotecă de proiectare pot fi accesate, astfel că nu este nevoie să avem codul respectiv în modelul curent.

Există două tipuri de biblioteci VHDL: biblioteca *work* şi biblioteca *resource.* Toate rezultatele analizei curente sunt memorate în biblioteca work. Bibliotecile resource se pot accesa pe durata analizei şi simulării, dar nu se poate scrie în ele. Totuşi, la un anumit moment, o bibliotecă resource trebuie să fie desemnată biblioteca work, astfel încât modelele să poată fi analizate. Vom ilustra în continuare cum se poate face acest lucru. Bibliotecile conțin atât unități *primare* cât şi *secundare,* unitățile primare sunt: declarațiile de entitate, pachet sau de configurație. Unitățile secundare sunt corpurile arhitecturale şi ale pachetelor. O unitate primară trebuie să fie analizată înaintea oricărei unități secundare corespunzătoare. O unitate secundară trebuie să fie analizată în aceeaşi bibliotecă precum şi corespondenta ei primară.

Bibliotecile au atât nume *logice* cât şi *fizice.* Numele logic se foloseşte în descrierea VHDL. Acest nume este portabil, şi este folosit pentru a face referire la bibliotecă fară a avea importanță sistemul în care biblioteca este instalată. Numele fizic este numele folosit de sistemul de operare gazdă pentru a face referire la bibliotecă. în general, el se schimbă în funcție de sistemul în care se instalează biblioteca. Trebuie să existe o metodă de corespondență între numele logice şi cele fizice. O posibiltate este aceea care este descrisă în continuare.

Numele logice ale bibliotecilor trebuie să fie declarate pentru a le face vizibile. Acest lucru se realizează cu ajutorul unei clauze *library.* Să presupunem că există o bibliotecă al cărei nume logic este DESIGN1. Acest nume este făcut vizibil folosind următoarea clauză *llibrary:*

```
library DESIGN1;
```

Apoi, se poate face referire la această bibliotecă în alte instrucțiuni. De exemplu, să presupunem că pachetul SIG, prezentat în paragraful anterior, se află în biblioteca DESIGN1. Semnalul X ar putea fi făcut vizibil prin următoarele două instrucțiuni:

```
use DESIGNI.SIG;
use SIG.X;
```

Sau, cu a singură instrucțiune:

```
use DESIGN1.SIG.X;
```

Este interesant să menționăm că, în ceea ce priveşte efectul lor total, cele două variante de mai sus nu sunt identice. Folosind varainta cu două instrucțiuni, vor fi vizibile atât pachetul SIG, cât şi semnalul X. A doua variantă face vizibil numai semnlul X, nu şi pachetul SIG.

Pentru numele logice ale bibliotecilor WORK şi STD nu este nevoie de o clauză library (numele bibliotecilor WORK şi STD) sunt întotdeauna vizibile. Biblioteca STD conține pachetele STANDARD şi TEXTIO. Toate numele din pachetul STANDARD sunt vizibile (clauza "use STD.STANDARD.all" este presupusă în mod implicit). Totuşi, este nevoie de o clauză use pentru a accesa elementele din pachetul TEXTIO.

2. Configurații

Arhitecturile structurale VHDL sunt dezvoltate prin instanțierea componentelor care sunt declarate în regiunea de declarații a arhitecturii .Înainte ca un model structural să poată fi simulat, fiecare componentă instanțiată trebuie să fie legată de o bibliotecă. În Figura 1 se arată modul în care se realizează acest lucra. Un model structural VHDL conține un set de pointeri la modele dintr-o bibliotecă de proiectare de tipul celor din paragraful anterior. O modalitate de a specifica pointerii este aceea de a plasa instrucțiuni de specificare a configurației direct în modelul structural.

În Figura 1 se prezintă un exemplu care foloseşte arhitectura de tip structural pentru entitatea TWO_CONSECUTIVE. Fiecare instrucțiune de specificare a configurației, comentată ca un pointer, este de forma:

for COMPONENTA_INSTANȚIATĂ use BIBLIOTECA_C0MPONENTĂ

Pentru cazul bistabililor D, **COMPONENTA_INSTANȚIATĂ** sunt **"all:EDGE_TRIGGERED_D".** Termenul **all** implică faptul că toate componentele instanțiate **EDGE_TRIGGERED_D** sunt transformate prin acelaşi model. Există şi posibilitatea transformării fiecărui flip-flop individual prin:

```
for CI:  EDGE_TRIGGERED_D
```

```
  use entity EDGE_TRIG_D_A(BEHAVIOR);
for C2:  EDGE_TRIGGERED_D
  use entity EDGE_TRIG_D_B(BEHAVIOR);
```

care ar transforma fiecare flip-flop într-un model de bibliotecă diferit.

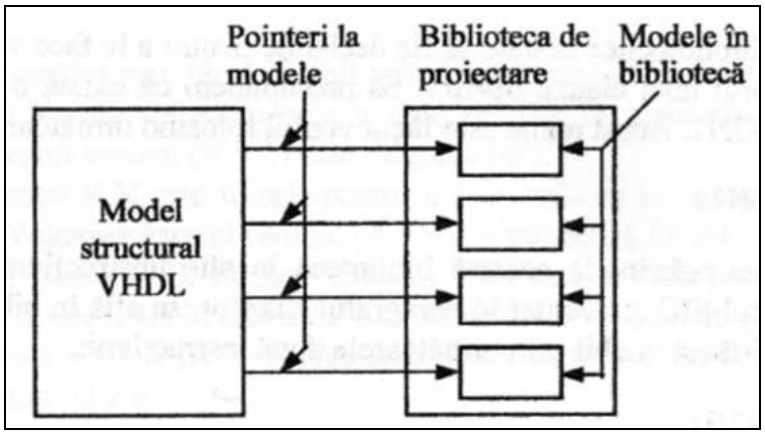


Fig. 1. Pointeri la modele din bibliotecă.

```
TWO_CONSECUTIVE is
  port(CLK,R,X: in BIT; Z: out BIT);
end entity TWO_CONSECUTIVE;          ;
use work.all;
architecture STRUCTURAL of TWO_CONSECUTIVE is
signal Y0,Y1,A0,A1: BIT : = 'OS-signal NY0,NX: BIT
  :='1\-signal ONE: BIT :='l';
 component EDGE_TRIGGERED_D
  port(CLK,D,NCLR: in BIT; Q,QN: out BIT);
  end component; for all: EDGE_TRIGGERED_D
    use entity EDGE_TRIG_D(BEHAVIOR); —model pointer component
  INVG
    port(I: in BIT;0: out BIT);
    end component;
    for all: INVG
    use entity INV(BEHAVIOR); —model pointer
  component AND3G
    port(II,12,13: in BIT;0: out BIT);
    end component;
    for all: AND3G
    use entity AND3(BEHAVIOR); —model pointer component
  0R2G
    port(II,12: in BIT;0: out BIT); end
```

```
component;
  for all: OR2G
  use entity OR2(BEHAVIOR); —model pointer begin
CI: EDGE_TRIGGERED_D
  port map(CLK,X,R,Y0,NY0);
  C2: EDGE_TRIGGERED_D
port map(CLK,ONE,R,Yl,open);
C3: INVG
  port map(X,NX); C4: AND3G
  port map(X,Y0,Yl,A0) ;
  C5: AND3G
port map(NYO,Yl,NX,Al) ;
C6: 0R2G
  port map(A0,Al,Z) ;
  end architecture STRUCTURAL;
```

Fig.2. Specificații de configurație pentru entitatea TWO_CONSECUTIVE.

În acest caz, biblioteca **WORK** conține modelele. După cum am discutat, numele **WORK** este întotdeauna vizibil. Instrucțiunea **"use work.all"** face vizibile numele entităților modelelor. O alternativă este renunțarea la **"use work. all"** şi folosirea numelui selectat al modelului componentei. De exemplu, specificația de componentă pentru bistabilii D poate fi:

```
for all:  EDGE_TRIGGERED_D
   use entity work.EDGE_TRIG_D(BEHAVIOR); -- pointer la model
```

Din nou, aceasta funcționează pentru că biblioteca **WORK** este mereu vizibilă.

O altă posibilitate de conectare împreună a componentelor este prin folosirea declarațiilor de configurație. In această metodă instanțele componentelor din arhitectura modelului sunt lăsate neconectate, iar instrucțiunile de specificare a componentelor sunt colectate într-o singură unitate analizabilă separat numită declarație de configurație. De exemplu, să presupunem că toate instrucțiunile de specificare a configurației (cele comentate ca pointeri) sunt eliminate din arhitectura STRUCTURAL a entității TWOCONSECUTIVE din Figura 2.

Specificațiile de componente pot fi grupate într-o declarație de componentă ca în Figura 3. Se observă că numele declarației este PARTS şi aceasta este o declarație de configurație pentru entitatea TWO_CONSECUTIVE. In interiorul declarației de configurație, instrucțiunea for exterioară dă numele arhitecturii care se va configura. Instrucțiunile for interioare sunt specificațiile de configurație pentru componentele instanțiate.

Declarațiile de configurații sunt unități primare ale bibliotecilor. Totuşi, ele trebuie să fie analizate după arhitectura pe care **o** configurează. De asemenea, când modelul este simulat, declarația de configurație se conectează la componenta de testat din programul de test (test bench) care testează modelul, aşa cum se ilustrează în Figura 4.

Declarațiile de configurație sunt utile atunci când se doreşte efectuarea unor simulări multiple ale aceleiaşi arhitecturi structurale cu componentele conectate la componente diferite din bibliotecă pentru fiecare simulare. în acest caz, se poate analiza arhitectura structurală o

singură fată. Apoi, se poate crea oricâte configurații diferite pentru acea arhitectura. Arhitectura structurală nu mai trebuie să fie analizată din nou. Acest mod de lucru reduce erorile şi timpul de analiză şi oferă posibilitatea reutilizării modelelor.

```
configuration PARTS of TWO_CONSECUTIVE is
  for STRUCTURAL
    for all: EDGE_TRIGGERED_D
       use entity work.EDGE_TRIG_D(BEHAVIOR);
    end for; for all: INVG
       use entity work.INV(BEHAVIOR);
    end for; for all: AND3G
       use entity work.AND3(BEHAVIOR);
    end for; for all: OR2G
    use entity work.0R2(BEHAVIOR); end
for; end for; end PARTS;
```

Fig.3. Declarația de configurație pentru entitatea TWO_CONSECUTIVE.

```
entity TB is end entity TB;
use WORK.ali;
architecture TCTEMPT of TB is signal
  X,R,CLK,Z: BIT; component TWIR
  port (CLK,R,X: in BIT; Z: out BIT);
  end component;
for CI: TWIR use configuration work.PARTS; begin CI: TWIR
   port map (CLK, R, X, Z ) ;
  CLK <='0',          'l'after   10 ns,'0'after   20ns,
'l'after   30 ns,'O'after   40 ns,'l'after   50ns, '0'after   60
ns,'l'after   70 ns,'O'after   80ns, 'l'after   90 ns,'O'after 100
ns,'l'after 110 ns, 'O'after 120 ns,'l'after 130 ns,'O'after 140 ns; X <=
'0','1' after   15 ns,'0' after 55 ns; R <= '1','0' after 125 ns,'1' after
127 ns; end architecture TCTEMPT;
```

Fig.4. Testarea unei configurații.

3. Fişiere l/O

O cerință de importanță deosebită în dezvoltarea modelelor VHDL este posibilitatea de a aplica unui model vectori de test. In discuția despre programe de test (test benches) am arătat cum se poate realiza acest lucru folosind elemente de forme de undă şi instrucțiuni de asignare de semnal. Aceste metode nu sunt practice pentru seturi mari de vectori de test. O problemă conexă este aceea că se doreşte inițializarea memoriilor la începutul simulării prin intermediul

unui fişier extern. în sfârşit, este de dorit să avem posibilitatea să scriem rezultate ale simulării formatate, uşor de înțeles, într-un fişier extern.

Pentru a aborda aceste probleme, VHDL oferă posibilitatea de a citi dintr-un şi a scrie într-un fişier extern pe durata simulării. Toate fişierele sunt secvențiale şi pot fi interpretate ca fluxuri de date. Există două tipuri de fişiere care pot fi scrise şi citite: fişiere text formatate şi fişiere text I/O. Fişierele formatate trebuie să fie scrise de un simulator VHDL; formatul fişierului este dependent de calculatorul gazdă şi nu poate fi citit de utilizator. Fişierele text pot fi citite de utilizator şi pot fi create folosind un editor de texte. Fişierele pot fi în modul *in* sau *out,* dar nu în modul inout. Deci, un fişier în care se scrie pe durata unei simulări nu poate fi citit pe durata aceleiaşi simulări.

3.1 Fişiere I/O formatate

Pentru fişierele I/O formatate sunt accesibile două declarații: o declarație de tip de fişier şi o declarație de fişier care foloseşte declarația de tip. de exemplu, un tip de fişier care reprezintă un flux de bit vectori folosit pentru a testa un model poate fi declarat prin:

type INP_COMB is file of Bit_Vector;

Tipul de bază al acestui fişier este **Bit_Vector.** În general, tipul de bază poate fi orice tip cu excepția tipului *access* sau un alt tip *fişier*. Cu această declarație, se poate apoi declara un fişier de ieşire:

file OUTVECT: INP_COMB is out "TEST.VECT";

În această declarație, **OUTVECT** este numele logic al fişierului, iar **"TEST.VECT"** este numele fişierului gazdă. Numele fişierului gazdă este un şir de caracter şi, prin urmare, trebuie să fie inclus între două semne de citare. Ori de câte ori se declară un fişier în modul **"out",** se defineşte în mod implicit o procedură **WRITE.** de exemplu, pentru fişierul de mai sus, procedura **WRITE** corespunzătoare este:

WRITE (OUTVECT: out INP_CCMB; V: in Bit_Vector);

unde **V** este o variabilă declarată în regiunea de declarații în care procedura **WRITE** este invocată. Ori de câte ori se execută o procedură **WRITE,** valoarea lui **V** este adăugată fişierului **OUTVECT.**

Similar, se poate declara un fişier de intrare prin:

file INVECT: INP_COMB is in "TEST.VECT";

Pentru toate fişierele de intrare se definesc în mod implicit **o** procedură **READ** şi **o** funcție **ENDFILE.** Pentru fişierul declarat mai **sus,** declarațiile **sunt:**

READ (INVECT, V, LENGTH)
ENDFILE (INVECT)

Fiecare invocare a procedurii **READ** va conduce la citirea unui bit vector din vârful fişierului **INVECT** şi asignarea acestuia variabilei **V.** Apoi, următorul bit vector din fişierul secvenţial se deplasează în vârful fişierului. Pentru fiecare citire, numărul natural **LENGTH** returnează lungimea bit vectorului. Acest parametru este prezent numai dacă tipul de bază al tipului fişierului este un tablou nerestricţionat. Funcţia **ENDFILE** returnează o valoare de tip boolean. Valoarea returnată este **TRUE** dacă s-a ajuns la sfârşitul fişierului; în caz contrar, se returnează **FALSE.**

Pentru a ilustra modul de folosire a fişierelor I/O formatate, să considerăm exemplul din Figura 5. Codul VHDL este alcătuit din două entităţi **OBVS** şi **IBVS.** Entitatea **OBVS** scrie o secvenţă de bit vectori într-un fişier gazdă numit **TEST.VECT.** Vectorii de test sunt generaţi de entitatea **PULSEGEN,** pe care am discutat-o în paragraful dedicat funcţiilor. Ne amintim că pentru **o** valoare dată *N,* această entitate generează o secvenţă care constă din toţ *2*N* vectori posibili cu lungimea *N.* Pe măsură ce se genereaz fiecare nou vector, comută ieşirea **SYNC.** De fiecare dată când **SYNC** se modifică, se execută procesul **WRITE_VECT,** şi se scrie bit vectorul într-un fişier.

Entitatea **OVBS** aplică vectorii de test. Când intrarea **PLAY** este '1', prima instrucţiune **wait** din procesul **READVECT** este îndeplinită, şi se intră în buclă. Fiecare trecere prin buclă citeşte un vector şi îl asignează variabilei **V.** Apoi **V** este asignată lui **BVOUT.** Apoi există o instrucţiune **wait** pentru perioada **PER** înainte de executarea următoarei citiri de fişier. Deci, **BVOUT** se schimbă la fiecare **PER** unităţi. Această buclă rulează până când se ajunge la sfârşitul fişierului. Apoi, se reia secvenţa completă de vectori de test.

```
entity OBVS is
generic(N:INTEGER;PER: TIME); port(GEN: in BIT);
end entity OBVS;
use work.all;
architecture FIO of OBVS is
  type INP_COMB is file of BIT_VECTOR; f
  ile OUTVECT: INP_COMB is out "TEST.VECT";
  signal VECTORS: BCTVECTOR(N-l downto 0);
  signal SYNC: BIT;
   component PG generic(N: INTEGER; PER: TIME);
  port(START: in BIT;
  PGOUT: out BIT_VECTOR(N-l downto 0); SYNC: inout BIT);
  end component;
for CI: PG   use entity work.PULSE_GEN(ALG);
begin
CI: PG
    generic map(N => N, PER => PER)
    port map(START => GEN, PGOUT => VECTORS, SYNC => SYNC);
    WRITE_VECT: process(SYNC)
    variable V: BIT_VECTOR(N-l downto 0);
    begin
    V := VECTORS;
  WRITE(OUTVECT, V);
  end process WRITE_VECT;
  end architecture   FIO;
```

```
  entity IBVS is
  generic(N:INTEGER;PER: TIME);
  port(PLAY: in BIT; BVOUT: out BIT_VECTOR(N-l downto 0)) ;
  end entity IBVS;
  architecture FIO of IBVS is
  type INP_C0MB is file of BIT_VECTOR;
  file INVECT: INP_COMB is in "TEST.VECT";
  begin
  READ_VECT: process
    variable LENGTH: NATURAL := N;
    variable V: BIT_VECTOR(LENGTH-1 downto 0);
begin
    wait on PLAY until PLAY = '1';
    loop
exit when ENDFILE(INVECT);
READ(INVECT,V,LENGTH);
BVOUT <= V;
wait for PER;
end loop;
end process READ_VECT;
end architecture FIO;
```

Fig.5.Scrierea şi citirea dintr-un fişier formatat.

Entitatea **OBVS** scrie în filierul **TEST.VECT** iar entitatea **IBVS** citeşte din el. Pentru ca aceste acțiuni să poată fi realizate, trebuie să se satisfacă două condiții:

1. Filierul de ieşire **(OUTVECT)** şi fişierul de intrare **(INVECT),** care sunt de fapt acelaşi fişier gazdă, **"TEST. VECT",** trebuie să fie declarate în regiuni declarative diferite.
2. Pe durata unei simulări, nu trebuie să se încerce scrierea şi apoi citirea din acelaşi fişier gazdă. Deci, în exemplul nostru, entitățile **OBVS** şi **IBVS** nu pot fi executate pe durata aceleiaşi simulări.

Aceste condiții asigură că un fişier nu este folosit în modul *inout* pe durata unei simulări.

3.2 Fişiere text l/O

După cum am arătat, fişierele formatate nu pot fi înțelese de utilizator. Mai mult, ele nu pot fi create cu uşurință în afara simulării VHDL în mediul gazdă al sistemului de operare. În multe situații, este nevoie să se poată citi fişiere pe durata simulării, fişiere care au fost generate cu un editor de texte sau care reprezintă codul obiect de ieşire al unui asamblor. VHDL oferă facilitatea text I/O pentru aceste cazuri.

Declarațiile de bază pentru fişierele text I/O sunt conținute în pachetul **TEXTIO,** care este inclus în biblioteca **STD.** Numele bibliotecii **STD** este întotdeauna vizibil; totuşi, numele sau conținutul pachetului **TEXTIO** trebuie să fie făcut vizibil prin clauze use. Primele două declarații din pachetul **TEXTIO** sunt:

type LINE is access STRING;

type TEXT is file of STRING;

Tipul **LINE** este declarat ca un tip access **STRING.** Tipurile access sunt folosite pentru variabile care reprezintă pointeri la memorie. Aceşti pointeri pot fi creați şi eliminați pe durata simulării. Deci, memoria nu este alocată în permanență pentru aceşti pointeri. detalii legate de

tipurile access pot fi găsite în cărți mai avansate dedicate VHDL . Tipul **TEXT** este un tip file al cărui tip de bază este tipul **STRING.** Tipul **STRING** se foloseşte pentru a reprezenta o gamă largă de date de intrare în mediul gazdă.

Cu aceste declarații, se poate declara un fişier text. De exemplu, în cazul citirii bit vectorilor dintr-un fişier dezvoltat extern, se poate scrie:

file INVECT: TEXT is "TVECT.TEXT" ;

Aşadar, fişierul text cu numele logic **INVECT** este transformat în fişierul gazdă cu numele **"TVECT. TEXT".**

Datele din fişierele text în VHDL sunt organizate în linii, cu un număr variabil de elemente pe fiecare linie. Pentru a citi date dintr-un fişier text, trebuie să se execute mai întâi o comandă **READLINE** pentru a citi o întreagă linie, apoi un anumit număr de comenzi **REA:** pentru elementele individuale din linia respectivă. Aceste funcții sunt declarate de asemenea în pachetul **TEXTIO.** Ca un exemplu de folosire a lor, să considerăm cazul citirii bit vectorilor din fişierul text declarat mai înainte. în acest caz, se poate invoca procesul de citire prin:

READLINE (INVECT, VLINE);

unde **VLINE** este o variabilă declarată de tipul **LINE.** Aceasta este urmată de un număr de apeluri de forma:

READ (VLINE, V);

unde **V** este o variabilă de tipul **Bit_Vector.** Fiecare invocare a acestei comenzi preia un bit vector din **VLINE** şi îl copiază în variabila **V.**

Pentru a ilustra întregul proces, vom arăta cum se citeşte un fişier text **"TVECT. TEXT"** care a fost generat folosind un editor de text şi care conține următoarele:

000
001
010
011
100
101
110
111
000

In Figura 6 se prezintă codul VHDL care conduce la citirea acestui fişier. Pentru exemplul nostru, valoarea generică *N* este setată la 3. Pe durata simulării, când **play** este **T,** se intră în bucla **while.** Pentru fiecare trecere prin buclă, se citeşte o linie şi este transmis la ieşire la **BVOUT** un bit vector. în acest caz, **READ** este invocat o singură dată pentru că în fiecare linie apare numai un singur bit vector. Dacă există mai mulți vectori pe o linie, **READ** este invocat de mai multe ori. Când se ajunge la sfârşitul fişierului, bucla se încheie. Procesul se

întoarce apoi la instrucțiunea wait de la începutul procesului şi îşi suspendă operarea până când apare un alt eveniment în semnalul **PLAY.**

Scrierea într-un fişier se realizează prin asamblarea liniilor folosind o comandă **WRITE, apoi** scriind liniile întregi într-un fişier folosind comanda **WRITELINE.** Nu vom ilustra aici acest proces ci vom propune un exercițiu la sfârşitul capitolului.

```
entity TBVS is
  generic(N:INTEGER;PER: TIME);
  port(PLAY: in BIT;
     BVOUT: out BIT_VECTOR(N-l downto 0));
     end entity TBVS;
     use STD.TEXTIO.all;
     architecture TIO of TBVS is
     begin
     process
  variabie VLINE: LINE;
  variable V:BIT_VECTOR(N-l downto 0);
  file INVECT: TEXT is "TVECT.TEXT";
  begin
wait on PIAY until PLAY = '1';
while not(ENDFTLE(INVECT)) loop
READLINE(INVECT,VLINE);
READ (VLINE, V); BVOUT <= V;
wait for PER;
end loop;
end process;
end architecture TIO;
```

Fig.6Codul VHDL care conduce la citirea unui fişier de intrare de tip TEXT.=

În acest exemplu, funcția **READ** realizează o conversie de tip de la tipul **TEXT** la tipul **BitVector.** Aceste tipuri "țintă" sunt restricționate la următoarele

Boolean
Bit
Bit_Vector
Character
Integer
Real
String
Time

Se observă că nu apar alte tipuri fizice în afara tipului **Time.** De asemenea, tipul enumerare definit de utilizator nu este inclus.

Curs 2

TIPURI DE DATE ŞI OPERAȚII – part 1

2.1 Constante şi variabile

2.2 Semnale

2.3 Tipuri scalare

Conceptul de tip este foarte important atunci când sunt descrise date într-un model VHDL. Tipul unor date defineşte setul de valori care pot fi atribuite, precum şi operațiile care pot fi realizate asupra datelor respective. Un tip scalar constă din valori singulare, indivizibile. În această parte vom lua în discuție tipurile scalare de bază oferite de VHDL şi vom arăta cum se pot folosi acestea pentru a defini obiecte care modelează starea internă a unor circuite electronice.

2.1 Constante şi variabile

Un obiect este un element cu nume într-un model VHDL care are o valoare de un tip specificat. *Există patru clase de obiecte: constante, variabile, semnale şi fişiere.* În acest capitol vom prezenta *constantele, variabilele şi semnalele*.

Constantele şi variabilele sunt obiecte în care se pot memora datele care se utilizează într-un model. Diferența dintre ele este aceea că valoarea unei constante nu se poate schimba după ce ea a fost creată, în timp ce valoarea unei variabile se poate modifica ori de câte ori este necesar, folosind instrucțiuni de asignare de variabile.

2.1.1 Declarații de constante şi de variabile

Atât constantele cât şi variabilele trebuie să fie declarate înainte de a putea fi folosite într-un model. O declarație introduce numele obiectului, defineşte tipul şi îi poate atribui o valoare inițială. *Regula de sintaxă pentru o declarație de constantă este*:

declarație_de_constantă <=
constant identificator {,...} : indicație_de_subtip [:= expresie];

Identificatorii listați sunt nume ale constantelor definite (câte una pentru fiecare nume), iar indicația de subtip indică tipul tuturor constantelor. Partea opțională este o expresie care specifică valoarea atribuită fiecărei constante. Iată câteva exemple de declarații de constante:

```
constant numar_de_octeti  :  integer := 4;
constant numar_de_biti :  integer := 8*număr_de octeți;
constant e : real  := 2.718281828;
constant intarziere_de_propagare : time := 3 ns;
constant dimensiune,  limita_de_numarare :  integer := 255;
```

Raţiunea pentru care se foloseşte o constantă este aceea de a avea un nume şi un tip definit explicit pentru o anumită valoare, în loc de a scrie valoarea respectivă ca un număr. Acest lucru face modelul mai uşor de înțeles pentru utilizator, pentru că numele şi tipul poartă mult mai multă informație despre intenția de utilizare a obiectului decât simpla valoare numerică. Mai mult, dacă dorim să schimbăm valoarea pe măsură ce modelul se dezvoltă, este nevoie să actualizăm numai declarația de constantă.

Forma unei declarații de variabile este similară cu aceea pentru constante. Regula de sintaxă este:

declarație_de_variabilă <=
variable identificator {,...}: indicație_de_subtip [:= expresie];

Şi de această dată, expresia pentru inițializare este opțională. Dacă este omisă, valoarea inițială implicită depinde de tip. De exemplu, pentru întregi ea este cel mai mic întreg reprezentabil. Iată câteva exemple de declarații de variabile:

```
variable index:  integer:= 0;
variable sum, average,  largest: real;
variable start, finish: time:= 0 ns;
```

```
architecture exemplu of ent is
  constant pi : real := 3.12159;
  begin
  process is
      variable counter : integer;
  begin
  ....      — instrucțiuni  care  folosesc  pi  şi  counter  end
process; end architecture exemplu;
```

Fig. 2.1 Un corp arhitectural care foloseşte o declarație de constantă şi o declarație de variabilă.

O declarație în care sunt incluşi mai mulți identificatori este echivalentă cu mai multe declarații câte una pentru fiecare identificator separat. De exemplu, ultima declarație de mai sus este echivalentă cu următoarele două declarații:

```
 variable   start:   time:=   0   ns;
vararіable finish: time:= 0 ns;
```

Aceasta înlocuire nu este semnificativă decât dacă expresia de inițializare poate produce valori diferite la două evaluări succesive.

Declarațiile de constante şi variabile pot apărea în mai multe locuri într-un model VHDL, inclusiv în părțile declarative ale proceselor. În acest caz, obiectul declarat poate **fi** folosit numai în interiorul procesului. O restricție asupra locului în care se face o declarație de variabilă este aceea că nu poate fi plasată astfel încât variabila să poată fi accesată de mai multe procese. Motivul este prevenirea efectelor neașteptate care pot apărea atunci când anumite variabile sunt modificate de un proces într-o ordine nedeterminată. Excepția de la această regulă este dată de variabilele care sunt declarate special ca variabile distribuite.

Exemplu

În Figura `2.1` este prezentat un corp arhitectural care arată cum pot fi incluse declarațiile de constante şi de variabile într-un model VHDL.

2.1.2 Asignarea variabilelor

După ce o variabilă a fost declarată, valoarea ei poate fi modificată folosind o instrucțiune de asignare. Sintaxa unei instrucțiuni de asignare de varaibilă este dată de regual următoare:

instrucţiune_de_asignare_de_variabilă <=

[etichetă:] nume := expresie;

Eticheta opțională furnizează o modalitate de identificare a instrucțiunii de asignare. Deocamdată, vom omite etichetele din exemplele discutate. Numele dintr-o instrucțiune de asignare de variabilă identifică variabila a cărei valoare va fi schimbată, iar expresia este evaluată pentru a determina noua valoare. Tipul acestei valori trebuie să fie compatibil cu tipul variabilei. Pe tot parcursul acestui curs vom prezenta detalii legate de modul în care se poate forma o expresie. *Pentru moment, vom înțelege expresiile ca fiind combinații de identificatori şi numere cu operatori*. Iată două exemple de instrucțiuni de asignare:

```
program_counter:= 0; index:= index + 1;
```

Prima asignare atribuie variabilei `program_counter` valoarea 0; valoarea anterioară a variabilei se şterge. Al doilea exemplu incrementează cu unu valoarea variabilei `index`.

Este important să accentuăm diferența dintre o instrucțiune de asignare de variabilă, prezentată în acest paragraf, şi instrucțiunea de asignare de semnal. O asignare de variabilă stabileşte imediat o nouă valoare pentru o variabilă. O asignare de semnal *planifică* o nouă valoare care va fi aplicată unui semnal la un anumit moment ulterior.

Cele două asignări au simboluri distincte:

<= pentru asignarea de semnal,

: = pentru asignarea de variabilă.

2.2 Semnale

Un semnal este un obiect care are şi o dimensiune timp. Instrucţiunile VHDL pot asigna valori viitoare obiectelor fără a afecta valoarea curentă. Cu ajutorul formelor de undă pot fi asignate o serie de valori care să apară în viitor. Valoarea curentă nu poate fi schimbată.

2.2.1 Declararea semnalelor

Declaraţia de semnal este similară cu declaraţia unei variabile. Semnalele nu pot fi declarate în interiorul proceselor sau al subprogramelor. Semnalele pot fi declarate ca porturi în declaraţiile de entitate sau în secţiunea declarativă a declaraţiilor de arhitectură. Iată câteva declaraţii de semnale:

```
type ROM_type is array  (color range <>)  of Natural;

type counter is range 0 to 100; type reg is range 0 to 100;


type numele_lunii is (IAN, FEB, MAR, APR, MAI, IUN, IUL, AUG, SEPT, OCT, NOV,
DEC);

type DATA is record
  ZIUA:  Integer range 1 to 31;
  LUNA: numele_lunii;
  ANUL:  Integer range 0 to 3000; end record;

signal X1, X2, X3,  X4, X5: Bit;
signal SR1,  SR2,  SR3,  SR4: Reg;
signal down_count: counter := counter'right;
signal ROM_B: ROM_type  (0 to 3)   : = (X"FFFF_FFFF",  X"2222_CCCC",
                                        X"A03B_0020",  X"ABCC_506C");
signal date_register:  DATA :=  (1,  IAN,  1900);
```

Valoarea iniţială implicită pentru `X1` este '`0`', pentru `SR1` este '`0`'. Valoarea iniţială pentru `down_count` este `100`. Valoarea iniţială pentru `ROM_B(0)` este `X"FFFF_FFFF"`. Toate porturile trebuie să fie semnale. Sintaxa pentru o asignare de semnal este:

instrucţiune_asignare_de_semnal <= ., ,

[eticheta:] nume <= [mecanism_de_întârziere] forme_de_undă;

unde

forme_de_undă <= (valoareexpresie [after expresie_f/me]){,...}

Această regulă de sintaxă ne spune că putem specifica un mecanism de întârziere şi unul sau mai multe elemente de forme de undă, fiecare constând dintr-o nouă valoare şi un timp de întârziere opţional. Aceste întârzieri ne permit să specificăm când se va aplica noua valoare.

Valoarea curentă a semnalului nu se schimbă niciodată printr-o instrucțiune de asignare de semnal. *Dacă nu se specifică o valoare a timpului, valoarea implicită este infinitezimal de mică numită interval (sau timp)* **delta**. Pentru a face distincția între asignările de semnale şi cele de variabile, se foloseşte delimitatorul <= şi nu : =. Iată câteva exemple de instrucțiuni de asignare de semnale:

```
XI <= '1'  after 10 ns; SR1 <= 5
after 5 ns;
X2 <= '0'  after 10 ns,  '1' after 20 ns,  ,O' after 30
ns;
X5 <= ' 1';
```

în Figura 2.2 sunt arătate momentele implicate în asignările de mai sus, presupunând că toate instrucțiunile se execută la momemtul *t.* Valoarea curentă a lui `XI` la momentul / se presupune ' 0', valoarea lui `SR1` la momentul t se presupune `10,` a lui `X2` este ' 1' iar a lui `X5` este ' 0'. Prima instrucțiune determină schimbarea lui `XI` la ' 1' la momentul *t* + 10 ns. Similar, a doua instrucțiune determină schimbarea semnalului `SR1` de la `10` la `5` la momentul *t*+5 ns. Instrucțiunea care asignează valori semnalului `X2` foloseşte forme de undă care determină trei schimbări viitoare în semnalul `X2`. Pentru că nu există o specificație explicită a timpului pentru schimbarea în `X5,` aceasta va apărea la momentul *t* + *delta.*

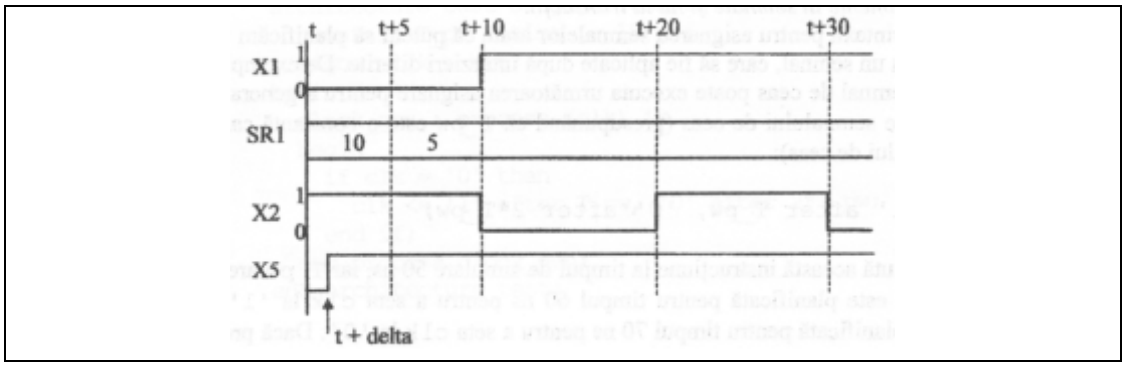


Fig. 2.2 Diagrame de timp pentru câteva instrucțiuni de asignare semnale

Tipul de bază al valorii asignate unui semnal trebuie să fie acelaşi cu tipul de bază declarat pentru semnal. De exemplu, instrucțiunea de asignare de semnal:

```
down_count <= 50.5 after 5 ns;
```

va produce o eroare pentru că tipul de bază al lui `downcount` este integer iar valoarea 50.5 este de tipul real. Un alt exemplu:

```
y <= not or_a_b after 5 ns;
```

Această instrucțiune specifică faptul că semnalul y trebuie să ia o nouă valoare cu 5 ns mai târziu decât momentul la care se execută instrucțiunea. întârzierea poate fi citită *în două moduri,* depinzând de faptul dacă modelul este folosit numai pentru valorile sale descriptive sau pentru simulare.

În *primul caz,* întârzierea poate fi considerată într-un sens *abstract* ca specificarea întârzierii de propagare introdusă de modul: când intrările se schimbă, ieşirea este actualizată cu 5 ns mai târziu. In *al doilea caz,* poate fi considerat în sens *operațional,* referitor la o maşină gazdă care simulează operarea modulului prin executarea unui model. Aşadar, dacă se execută asignarea de mai sus la momentul 250 ns, şi or_a_b are valoarea ' 1' la acel moment, atunci semnalul y va lua valoarea ' 0' la momentul 255 ns. Menționăm că instrucțiunea propriu-zisă se execută în timpul de modelare zero.

Dimensiunea timp la care ne referim atunci când modelul se execută este *timpul de simulare,* adică timpul în care circuitul de modelat considerat că funcționează. Acesta este diferit de timpul real de execuție pe maşina gazdă care execută simularea. Măsurăm timpul de simulare de la zero, la începutul execuției, şi crescând în paşi discreți pe măsură ce apar evenimente în model. Această tehnică se numeşte *simularea evenimentelor discrete.* Un asemenea simulator trebuie să aibă un ceas de simulare, şi atunci când se execută o asignare de semnal, întârzierea specificată se adună la timpul curent de simulare pentru a determina când noua valoare trebuie aplicată semnalului. Spunem că asignarea semnalului planifică o *tranzacție* pentru semnal, unde tranzacția constă din noua valoare şi din momentul la care ea trebuie aplicată. Atunci când timpul de simulare avansează la timpul la care este planificată tranzacția, semnalul este actualizat cu noua valoare. Spunem că semnalul este *activ* pe durata acelui ciclu de simulare. Dacă noua valoare este diferită de vechea valoare pe care o înlocuieşte în semnal, spunem că a apărut un *eveniment* în semnalul respectiv. *Procesele răspund la evenimente în semnale şi nu la tranzacții.*

Regulile de sintaxă pentru asignarea semnalelor arată că putem să planificăm un număr de tranzacții pentru un semnal, care să fie aplicate după întârzieri diferite. De exemplu, un proces condus de un semnal de ceas poate executa următoarea asignare pentru a genera următoarele două fronturi ale semnalului de ceas (presupunând că T_pw este o constantă care reprezintă lățimea impulsului de ceas):

```
clk <= '1' after T_pw,  '0' after 2*T_pw;
```

Dacă se execută această instrucțiune la timpul de simulare 50 ns, iar T p w are valoarea 10 ns, o tranzacție este planificată pentru timpul 60 ns pentru a seta clk la ' 1', iar a doua tranzacție este planificată pentru timpul 70 ns pentru a seta clk la ' 0'. Dacă presupunem că clk are valoarea [1] 0' atunci când se execută asignarea, ambele tranzacții produc evenimente asupra lui clk.

Instrucțiunea de asignare a semnalelor arată că dacă sunt incluse mai multe tranzacții, întârzierile sunt toate măsurate începând de la timpul curent. Mai mult, tranzacțiile din listă rebuie să aibă întârzieri strict crescătoare.

Exemplu

Putem scrie o declarație de proces pentru un generator de ceas folosind instrucțiunea de asignare de semnal de mai sus, pentru a genera un semnal de ceas simetric, cu durata pulsului T_pw. Dificultatea constă în obținerea unui proces care să se execute uniform, la fiecare ciclu de ceas. O cale pentru a face acest lucru este prin reluarea procesului ori de câte ori semnalul de ceas se schimbă şi programarea următoarelor două tranziții când se schimbă la '0'. Aceasta se poate realiza prin Figura 2.3.

Pentru că un proces este unitatea de bază a unei descrieri de tip *comportamental,* are sens | intuitiv) să se permită includerea uneia sau a mai multor instrucțiuni de asignare de semnal pentru un singur semnal în interiorul unui singur proces.

Exemplu

Putem scrie un proces care modelează un MUX cu doua intrări, ca în Figura 2.4. Valoarea portului sel se foloseşte pentru a selecta care asignare de semnal se va executa pentru a determina valoarea ieşirii.

Spunem că un proces defineşte un **driver** *pentru un semnal dacă şi numai dacă el conține cel puțin o instrucțiune de asignare pentru acel semnal*. Aşadar, exemplul de mai sus defineşte un driver pentru semnalul z. Dacă un proces conține instrucțiuni de asignare pentru mai multe semnale, el defineşte drivere pentru fiecare din acele semnale. *Un driver este o* **sursă** *pentru um semnal, în sensul că el furnizează valori care se aplică semnalului.*

!!!!O regulă importantă care trebuie amintită este aceea că pentru semnale normale poate exista numai o singură sursă.

Aceasta înseamnă că nu putem scrie două procese diferite, fiecare conținând instrucțiuni de asignare pentru acelaşi semnal.

Dacă dorim să modelăm bus-uri sau semnale *wired-or,* trebuie să folosim un tip special de semnal numit *semnal rezolvat* (resolved signal).

```
 entity generator is end
 entity generator ;
 architecture test of generator is
constant T_pw : time       10 ns;
signal clk : bit; begm
clock_gen : process (clk) is begin
  if clk = '0' then
    clk <= '1' after T_pw, '0' after 2*T_pw; end if;
end process clock_gen; end architecture test;
```

Fig. 2.3 Un proces care generează un semnal de ceas simetric

```
entity model is
end entity model ;

architecture test of model is
  constant prop_delay : time := 5 ns; signal a,
  b, sel, z : bit;

begin
  mux : process (a, b, sel) is begin
   case sel is
   when '0' => z <= a after prop_delay;
      when '1' => z <= b after prop_delay;
    end case; end process mux;

  stimulus : process is
subtype stim_vector_type is bit_vector(0 to 3);
type   stim_vector_array   is   array   (   natural   range   <>   )   of
stim_vector_type;
    constant stim_vector : stim_vector_array := ( "0000",
      "0010",  "0100",  "0111",  "1001",  "1010",  "1101",
      "1111" );
  begin
    for i in stim_vector'range loop
      (a, b, sel) <= stim_vector(i)(0 to 2);
     wait for 10 ns;
     assert z = stim_vector(i)(3); end loop; wait;
  end process stimulus;
  end architecture test;
```

Fig. 2.4 Un proces care modelează un multiplexor **2:1.**

2.2.3 Drivere de semnal

Dacă un proces conține una sau mai multe instrucțiuni de asignare de semnale care planifică valori viitoare pentru un anumit semnal X, simulatorul VHDL un singur element în care se reține valoarea semnalului; acest element se numeşte *driver de semnal* pentru X şi este asociat cu procesul. *Driverul menține o listă ordonată a valorilor planificate pentru semnalul X.* Fiecare asignare de semnal planificată se numeşte *tranzacție.* Noua valoare asignată unui semnal X printr-o tranzacție poate să fie diferită sau nu de valoarea curentă.

Dacă semnalul X se schimbă în urma unei tranzacții, spunem că a avut loc un eveniment *în semnalul X.*

Driverul menține valoarea semnalului între tranzacții. Simulatorul VHDL citeşte valoarea curentă a semnalului din driver.

Dacă mai multe procese conțin instrucțiuni de asignare pentru acelaşi semnal X, simulatorul formează un driver separat pentru semnalul X pentru fiecare proces, deci, la orice moment,

pot exista mai multe drivere, fiecare fiind asociat cu un proces diferit, care planifică valori pentru semnalul X. Evident, trebuie să existe *o cale de a determina valoarea corectă pentru semnalul X dacă diferite drivere specifică diferite valori pentru semnalul X la acelaşi moment de timp*. Problema se rezolvă prin specificarea unei *funcţii de rezoluţie* pentru semnalul X. Aceasta calculează ***valoarea rezolvată*** pentru toate perechile posibile de valori pe care două drivere diferite încearcă să le asigneze semnalului X la un moment dat. Semnalul X se numeşte *semnal rezolvat.* Funcţia de rezoluţie poate fi asociată cu un tip de date sau cu un semnal**.** Următoarele exemple arată cum se poate specifica un semnal rezolvat:

```
signal XI: wired_OR Bit;        —Exemplul 1
subtype resolved_Bit is wired_OR   Bit;
signal X2: resolved_Bit;
signal Yl: resolved std_ulogic;       —Exemplul 2 subtype std_logic is resolved std_ulogic;
signal Y2:  std_logic;
```

Prima instrucţiune din Exemplul 1 declară direct faptul că semnalul XI este un semnal rezolvat de tipul Bit cu funcţia de rezoluţie wired_OR. Numele funcţiei de rezoluţie este separat de numele tipului prin caracterul separator spaţiu. A doua instrucţiune asociază funcţia de rezoluţie wired_OR cu subtipul resolved_Bit care are acelaşi tip de bază Bit. Orice semnal declarat de subtipul resolved_Bit va fi automat un semnal rezolvat cu funcţia de rezoluţie wired_OR. A treia instrucţiune declară semnalul X2 de subtipul resolved_Bit.

Al doilea exemplu repetă procesul pentru standard logic IEEE. Mai întâi, se declară direct semnalul Yl, un semnal rezolvat de tipul st _ u logic care foloseşte funcţia de rezoluţie resolved. Apoi, funcţia de rezoluţie resolved este asociată cu tipul std_ulogic pentru a forma subtipul std_logic. Semnalul Y2 este declarat a avea subtipul stdlogic şi, prin urmare, va moşteni funcţia de rezoluţie resolved. În modelele dedicate sintezei, cele mai multe semnale şi variabile scalare sunt declarate de subtipul std_logic.

Atunci când drivere de semnal diferite pentru semnalul X specifică valori pentru X la un moment dat, simulatorul execută funcţia de rezoluţie pentru a determina valoarea corectă a semnalului. .

2.2.4 Atributele semnalelor

*Putem face referire la atributele semnalelor pentru a obţine informaţie legată de istoria tranzacţiilor sau a evenimentelor aces*tora. Dacă se dă un semnal **S,** şi o valoare **T** de tip **time,** VHDL defineşte atributele următoare.

S 'delayed(T)	Un *semnal* care ia aceleaşi valori cu S dar care este întârziat cu timpul T
S 'stable(T)	Un *semnal boolean* care este adevărat dacă nu a fost nici un eveniment în S în intervalul de timp T, până la timpul curent. în caz contrar este fals.
S 'quiet(T)	Un *semnal boolean* care este adevărat dacă nu a fost nici o tranzacţie în S în intervalul de timp T până la momentul curent. în caz contrar este fals.
S 'transaction	Un *semnal de tip bit* care îşi schimbă valoarea de la '0' la '1'

	sau invers de fiecare dată când există o tranzacție în S
S' 'event	TRUE dacă este un eveniment în S în ciclul curent de simulare, FALSE în caz contrar.

S'active	TRUE dacă este o tranzacție în S în ciclul curent de simulare, FALSE în caz contrar.
S'last_event	*Intervalul de timp* de la ultimul eveniment în S.
S'last_active	*Intervalul de timp* de la ultima tranzacție în S
S'last_value	*Valoarea* lui S chiar înainte de ultimul eveniment în S.

Primele trei atribute necesită un parametru opțional de tip time. Dacă omitem acest parametru, valoarea lui implicită este 0 fs. Aceste atribute sunt folosite adesea pentru verificarea comportării în timp a unui model. De exemplu, putem verifica dacă un semnal d respectă un timp **minim** de set-up, Tsu, înainte de frontul crescător al semnalului clk de tip std_ulogic:

```
if elk'event and  (clk = '1'  or clk = 'H'  )
and (clk'last_value = '0'  or clk'last_value =  ' L '   )  then assert d'last_event >= Tsu
report "Timing error: d changed within setup time of clk ";
end if;
```

În mod similar, putem verifica faptul că lățimea pulsului semnalului de ceas nu depăşeşte i frecvență maximă, prin testarea acestui puls:

```
assert  ( not clk'event )  or clk'delayed'last_event >= Tpw__cli report "Clock frequency too high";
```

Observăm că se testează timpul de la ultimul eveniment al unui semnal întârziat față semnalul de ceas. Atunci când există un eveniment într-un semnal, atributul ' last_event returnează valoarea 0 fs. în acest caz, se determină timpul de la evenimentul anterior prin aplicarea atributului ' last_event semnalului întârziat cu 0 fs. Putem interpreta aceasta < fiind o întârziere infinitezimală.

Exemplu

Putem folosi un test similar pentru frontul crescător al unui semnal de ceas pentru a modela un modul comandat pe frontul unui semnal de ceas, un flip-flop.

```
entity edge_triggered_Dff is port ( D : in bit;    clk : in bit;
clr  :  in  bit;  Q  :  out  bit  );  end  entity  edge_triggered_Dff;
architecture behavioral of edge_triggered_Dff is begin
  state_change : process (clk, clr) is begin
    if clr m '1' then
      Q <= '0' after 2 ns;
   elsif clk'event and clk = '1' then
Q <= D after 2 ns; end if; end process state_change; end architecture behavioral;
```

Fig. 2.5 O entitate şi un corp arhitectural pentru un un flip-flop de tip D.
Se foloseşte atributul 'event pentru a identifica schimbările în semnalul clk.

Acest flip-flop trebuie să încarce valoarea intrării de date D la aparitia unui front crescător în clk, trebuie să şteargă asincron ieşirea ori de câte ori clr este ' 1'. Declarația de entitate şi corpul arhitectural comportamental sunt prezentate în Figura 2.5.

Dacă flip-flop-ul nu are intrare asincronă de ştergere, modelul ar putea avea o instrucțiune wait simplă:

```
wait until clk = '1';
```

Dacă există intrare de ştergere, procesul trebuie să fie sensibil la clk şi clr în orice moment. Deci, foloseşte atributul ' event pentru a distinge între schimbarea lui clk la ' 1' şi schimbarea lui clr la ' 0' când ceasul este stabil la [1]1'.

În Figura 2.6 se prezintă diagramele de timp pentru un semnal X1 de tip Bit şi pentru unele din atributele acestuia. Există evenimente în semnalul XI la momentele 5, 10, 20 şi 30 ns. Semnalul XI' delayed (8 ns), care este de asemenea de tipul Bit, este o copie a semnalului X1 deplasată cu 8 ns înainte în timp. Aşadar, există evenimente în semnalul XI'delayed (8 ns) la momentele 13, 18, 28 şi 38 ns. Semnalul XI'stable (8 ns) este de tipul Booleean: Valoarea lui inițială este TRUE. El se schimbă la FALSE la *t* - 5 ns datorită evenimentului în XI. Se schimbă de la FALSE la TRUE la *t* = 18 ns pentru că semnalul XI a fost stabil 8 ns consecutive (nu au fost evenimente în precedentele 8 ns). Semnalul XI' stable (8 ns) devine FALSE din nou la / = 20 ns datorită evenimentului în semnalul XI din acel moment. La *t* = 28 ns, semnalul XI a fost stabil pentru 8 ns; deci, semnalul XI'stable(8 ns) devine TRUE din nou. Cititorul ar trebui să verifice evenimentele care au mai rămas din semnalul XI'stable(8 ns). Semnalul XI' stable (8 ns) are evenimente la momentele 18, 20,28,30 şi 38 ns.

În mod simular, semnalul XI'stable(2 ns) este inițializat la TRUE şi devine FALSE la *t* = 5 ns, la apariția evenimentului din XI. Semnalul XI' stable (2 ns) se schimbă de la FALSE la TRUE la *t* = 7 ns pentru că semnalul XI a fost stabil în precedentele 2 ns. Semnalul XI' stable (2 ns) este TRUE în cea mai mare parte a timpului. El devine FALSE la momentele 5, 10,20 şi 30 ns, care corespund evenomentelor în XI.

Atributul XI' event este diferit de celelalte atribute din Figura 2.6 pentru că el nu este un semnal; este o funcție care se evaluează la TRUE numai dacă există un eveniment în semnalul XI pe durata ciclului curent de simulare. XI' event şi XI' stable au valori complementare în fiecare moment.

Următoarele expresii ilustrează diferite aspecte ale atributelor semnalelor. Pentru că `XI' stable` şi `XI' delayed` sunt semnale, ele au şi atribute.

`XI' delayed (8 ns)`	- această expresie are valoarea '1' la f = 15 ns
`Xl'stable(8 ns)`	- această expresie are valoarea FALSE la r = 15 ns
`XI' stable (2 ns)`	- această expresie are valoarea TRUE la *t* = 15 ns
`XI' event`	- această expresie are valoarea FALSE la *t* — 15 ns
`XI[1] last_event`	- această expresie are valoarea 5 ns la / = 15 ns
`Xl'last_value`	- această expresie are valoarea' 1' la *t* = 15 ns
`XI' stable' last_event`	- valoarea este 5 ns Iar = 15 ns
`Xl'stable(2 ns) ' last_event`	- valoarea este 3 ns la t = 15 ns
`XI' delayed (8 ns) ' event`	- are valoarea TRUE numai la momentele 13, 18,28 şi 38 ns

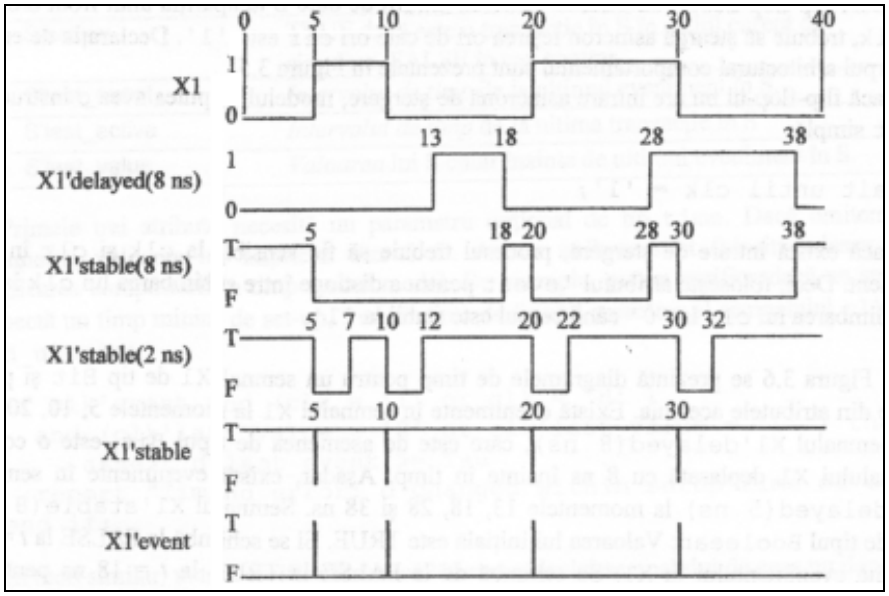


Fig. 2.6 Diagrame de timp pentru semnalul XI şi câteva din atributele acestuia

2.3 Tipuri scalare

Noțiunea de tip este foarte importantă în VHDL. Spunem că VHDL este un limbaj puternic tipizat, aceasta însemnând că fiecărui obiect i se pot asigna numai valori din tipul nominalizat (din tipul obiectului respectiv). In plus, definiția pentru fiecare operație include tipurile de valori la care se poate aplica oparația respectivă. Scopul unei puternice tipizări este acela de a permite detecția erorilor în etapele primare ale procesului de proiectare, atunci când modelul este analizat.

În acest paragraf vom arăta cum se poate declara un nou tip de date. Apoi, vom arăta cum se definesc diferite tipuri scalar. Un tip scalar este un tip ale cărui valori sunt indivizibile.

2.3.1 Declarații de tip

Putem introduce noi tipuri într-un model VHDL folosind declarații de tip. Declarația numeşte un tip şi specifică valorile care pot fi folosite pentru un obiect din acel tip. Regula de sintaxă pentru o declarație de tip este:

declarație_de_tip <= type identificator is definiție_de_tip;

Un lucru important care trebuie menționat este acela că dacă două tipuri sunt declarate separat cu definiții de tip identice, ele sunt totuşi tipuri distincte şi incompatibile. De exemplu, dacă avem două declarații de tip:

type mere is range 0 to 100;

type portocale is range 0 to 100;

nu putem asigna o valoare de tipul mere unei variabile de tipul portocale, pentru că ele sunt de tipuri diferite.

O utilizare importantă a tipurilor este specificarea valorilor admise pentru porturile unei entități. în exemplele din Capitolul 1 am văzut tipul Bit folosit pentru a specifica faptul că porturile pot lua numai valorile '0' şi T. Dacă definim propriile noastre tipuri pentru porturi, numele tipurilor trebuie declarate într-o declarație package, astfel încât ele să fie vizibile în declarația de entitate. De exemplu, să presupunem că dorim să definim o entitate small_adder care adună întregi din gama 0...255. în acest caz, trebuie să scriem o declarație package care să conțină declarația de tip, după cum urmează:

package int_types *is*
type small_int *is range 0 to 255;*
end package int_types*;*

Acest fragment defineşte un pachet numit int_types, care furnizează tipul numit small_int. Pachetul este privit ca o unitate de proiectare separată şi este analizat înainte de orice declarație de entitate care trebuie să folosească tipul respectiv. Putem folosi tipul introducând înaintea declarației de entitate o clauză use:

*use work.*int_types*.all;*

entity small_adder is
port (a, b: in small_int; s: out small_int); end entity small_adder;

2.3.2 Tipuri întregi

Un exemplu de tip întreg predefinit în VHDL este tipul **integer,** care include toate numerele întregi reprezentabile pe un calculator gazdă particular. Limbajul standard impune **ca** tipul ***integer*** *să includă cel puțin numerele între -2147483647 şi +2147483647 (-$2^{3,}$+l şi +2^{31}-1),* dar unele implementări pot extinde această gamă.

Putem defini un nou tip întreg folosind o definiție de tip cu restricție de gamă. Regula de sintaxă pentru definirea unui tip întreg este:

definiție_tip_întreg <=
range expresie simplă (**to** | **downto**) expresie simplă

care defineşte mulțimea de întregi dintre (inclusiv) valorile date de cele două expresii simple. Expresiile trebuie să fie evaluate la valori întregi. Dacă se foloseşte cuvântul cheie **to,** înseamnă că definim o **gamă crescătoare,** în care valorile sunt ordonate de la cea mai mică (în stânga) la cea mai mare (în dreapta). Dacă folosim cuvântul cheie **downto** înseamnă că definim o **gamă descrescătoare,** în care valorile sunt ordonate de la stânga la dreapta de la cea mai mare la cea mai mică.

Exemplu

Iată două declarații de tipuri întregi:

type day_of_month *is range 0 to 31;*

type year *is range 0 to 2020;*

Aceste două tipuri sunt destul de diferite, deşi ele includ unele valori comune. Deci, dacă vom declara variabile din aceste tipuri:

variable today: day_of_month := 7;

variable start_year: year : = 1989;

este ilegal să facem asignarea:

start year := today;

Deşi numărul 7 este şi în tipul year, în context el este tratat ca fiind de tipul dayofmonth, care este incompatibil cu tipul year. Această regulă de tip ajută la evitarea confuziilor legate de reprezentarea diferitelor feluri de elemente.

Dacă dorim să folosim o expresie aritmetică pentru a specifica limitele unei game, valorile care sunt folosite în expresie trebuie să fie local statice, adică, ele trebuie să fie cunoscute atunci când se analizează modelul. De exemplu, putem folosi valori constante într-o expresie, ca parte a unei definiții de gamă:

constant număr_de_biți : integer := 32;
type bit_index is range 0 to număr_de_biți - 1;

Operațiile care pot fi efectuate asupra tipurilor întregi includ operațiile aritmetice "obişnuite":

+	adunare, sau identitate
-	scădere, sau negare
*	înmulțire
/	împărțire
mod	modulo
rem	rest
abs	valoare absolută
**	exponențială

Rezultatul unei operații este un întreg de acelaşi tip cu operanzii. Pentru operatorii binari (care folosesc doi operanzi), operanzii trebuie să fie de acelaşi tip. Operatorul din dreapta operatorului exponențial trebuie să fie un întreg ne-negativ.

Operatorii identitate şi negație sunt unari, adică au nevoie de un singur operand (în dreapta lor). Rezultatul operatorului identitate este chiar operandul neschimbat; operatorul de negare produce un rezultat egal cu 0 - operandul. Deci, toate operațiile următoare produc acelaşi rezultat:

A + (- B) , A - (+ B) , A - B

Operatorul de împărțire produce un întreg care este rezultatul împărțirii, orice parte fracționară fiind trunchiată la zero. Operatorul rest este definit astfel încât relația următoare este adevărată:

*A = (A / B) * B + (A rem B)*

Rezultatul operației *A rem B* este restul împărțirii lui A la B. El are semnul lui A şi are valoarea absolută mai mică decât valoarea absolută a lui B. De exemplu,

5 rem 3 = 2, (- 5) rem 3 = -2,
5 rem (- 3) = 2, (- 5) rem (- 3) = - 2
în aceste exemple parantezele sunt impuse de regulile de gramatică pentru VHDL.

Operatorii rem şi negație nu pot fi folosiți unul lângă altul.
Operatorul modulo este conform cu definiția matematică.

Atunci când o variabilă este declarată a fi de un tip întreg, valoarea inițială este valoarea cea mai din stânga a gamei tipului respectiv. Pentru game ascendente, această valoarea este valoarea cea mai mică; pentru game descendente va fi valoarea cea mai mare. Dacă avem declarațiile următoare:

```
type set_index_range is range 21 downto 11;

type mode_pos_range is range 5 to 7;

variable set_index:  set_index_range;

variable mode_pos: mode_pos_range;
```

Valoarea inițială pentru set_index este valoarea cea mai din stânga a gamei lui set_index_range, adică 21; valoarea inițială pentru mode_pos este valoarea cea mai din stânga a gamei lui mode_pos_range, adică 5. Valoarea inițială a unei variabile de tipul integer este - 2147483647 sau mai mică, pentru că acest tip este prédéfinit cu o gamă crescătoare care trebuie să includă - 2147483647.

2.3.3 Tipuri în virgulă flotantă

Pentru a reprezenta numere reale în VHDL se folosesc tipuri de date în virgulă flotantă. Din punct de vedere matematic, în orice interval există o infinitate de numere reale, astfel încât nu este posibil să toate numerele reale exact pe un calculator. Tipurile în virgulă flotantă sunt doar aproximații ale numerelor reale. *Termenul "virgulă flotantă" se referă la faptul că aceste numere sunt reprezentate folosind o parte care se numeşte mantisă şi o parte care este exponentul.*

*În VHDL există un tip în virgulă flotantă predefinit, numit real, care include cel puțin gama de la - 1.0E+38 la + 1.0E+38, cu o precizie de cel putin şase cifre zecimal*e. Aceasta

corespunde *standardului IEEE cu reprezentare pe 32 biţi*, folosit în mod normal pentru numerele în virgulă flotantă. Unele implementări ale VHDL pot extinde această gamă şi pot oferi o precizie mai mare.

Definim un nou tip în virgulă flotantă folosind o definiţie de tip cu restricţia gamei. Regula de sintaxă este:

definiţie_tip_virgulă_flotantă <=
range expresie_simp!ă (to | downto) expresie_simplă

Aceasta este similară cu modalitatea prin care se declară un tip întreg, excepţia fiind dată de faptul că limitele trebuie să fie evaluate la numere în virgulă flotantă. Iată două exemple:

type input_level *is range -10.0 to +10.0;*

type probability *is range 0.0 to 1.0;*

Operaţiile care pot fi efectuate asupra numerelor în virgulă flotantă includ operaţiile aritmetice adunare şi identitate (+), scădere şi negaţie (-), înmulţire (*), împărţire (/), valoare absolută (abs) şi exponenţiala (**). Rezultatul unei operaţii este de acelaşi tip în virgulă flotantă ca şi operandul sau operanzii. Pentru operatori binari, operanzii trebuie să fie de acelaşi tip. Excepţia este dată de operandul din dreapta operatorului exponenţial care trebuie să fie întreg. Identitatea şi negaţia sunt operatori unari.

Variabilele care sunt declarate de tipul virgulă flotantă au o valoare iniţială implicită egală cu valoarea cea mai din stânga a gamei tipului. Aşadar, dacă declarăm o variabilă ca fiind de tipul input_level definit mai sus:

variable input_A: input_level;

valoarea ei iniţială este -10.0, adică valoarea cea mai din stânga a gamei lui input_level.

2.3.4 Tipuri fizice

Restul tipurilor numerice din VHDL sunt reprezentate de tipurile fizice. *Acestea sunt folosite pentru a reprezenta cantităţi fizice din lumea reală, cum sunt lungimea, masa, timpul, curentul, etc.* Definiţia unui tip fizic include unitatea primară de măsură şi poate include unităţi secundare care sunt multipli întregi ai unităţilor primare. Regula de sintaxă pentru definiţia unui tip fizic este:

definiţie_tip_fizic <=
type numetip range expresie simplá (to | downto) expresie_simplă
units
identificator;
{identificator = literal_fizic;}
end units [nume_tip]

Aceasta este similară cu definiția unui tip întreg, dar conține în plus partea de definire a unităților. Unitatea primară (primul identificator de după cuvântul cheie units) este cea mai mică unitate. Putem defini un număr de unități secundare. Gama (range) specifică multiplii unității primare care sunt incluşi în tipul respectiv. Dacă se include un identificator la sfârşitul definiției unităților, el trebuie să repete numele tipului care se defineşte.

Exemplu

Iată o declarație pentru un tip fizic ce reprezintă rezistența electrică:

```
type resistance is range 0 to 1E9

units
      ohm;
end units resistance;
```

Valorile numerice ale acestui tip sunt urmate de numele unității. De exemplu:

5 ohm, 35 ohm, 254 ohm

Trebuie să includem un spațiu înaintea numelui unității. Dacă valoarea este 1, ea poate fi omisă păstrând numai numele unității:

ohm, 1 ohm

Unitățile secundare se pot specifica prin indicarea numărului de unități primare pe care le alcătuiesc. Declarația pentru rezistența electrică poate fi extinsă la:

```
type resistance is range 0 to 1E9

units ohm;

kohm = 1000 ohm;

Mohm = 1000 kohm;

end units resiatance;
```

După ce o unitate secundară a fost definită, ea poate fi folosită pentru a specifica alte unități secundare. Desigur, unitățile secundare nu trebuie să fie puteri ale lui 10 înmulțite cu unitatea primară; totuşi, multiplicatorul trebuie să fie un întreg. De exemplu, un tip fizic pentru lungime poate fi declarat în forma următoare:

```
type length is range 0 to 1E9

units
      um;          ~ unitatea primară, micron
      mm = 1000 um;   --unități metrice
       m = 1000 mm;
```

```
mil = 254 um;   — unități imperiale
inch = 1000 mii;
end units length;
```

Putem scrie cantități fizice folosind unitățile secundare. De exemplu:

23 mm, 450 mil, 9 inch

Atunci când scriem cantități fizice putem folosi multipli care nu sunt întregi ai unităților primare sau secundare. în aceste cazuri valoarea se rotunjeşte în jos, la multiplul cel mai apropiat. De exemplu, putem scrie următoarele numere de tipul length, fiecare reprezentând aceeaşi valoare:

0.1 inch, 2.54 mm, 2.540528 mm

În ultimul caz, valoarea se rotunjeşte la 2540 pentru că unitatea primară pentru **length** este um. Dacă scriem 6.8 um, aceasta va fi rotunjită la 6 um.

Asupra tipurilor fizice se pot efectua multe operații aritmetice, dar cu anumite restricții. Adunarea, scăderea, identitatea şi negația pot fi aplicate valorilor de tipul fizic iar rezultatul va fi de acelaşi tip cu operandul sau operanzii.

O valoare de tipul fizic poate fi înmulțită cu un întreg sau cu un număr în virgulă flotantă, obținând o valoare de acelaşi tip fizic, de exemplu, 5 mm * 6 = 30 mm.

O valoare de tipul fizic poate fi împărțită la un întreg sau la un număr în virgulă flotantă ia rezultatul este o valoare de acelaşi tip fizic. Două valori de acelaşi tip fizic pot fi împărțite iar rezultatul este un întreg, de exemplu,

18 kohm / 2.0 = 9 kohm, 33 kohm / 22 ohm = 1500

Operatorul abs poate fi aplicat unei valori de tipul fizic şi se obține o valoare de acelaşi tip de exemplu,

abs 2 foot = 2 foot, abs (-2 foot) = 2 foot

VHDL nu acceptă înmulțirea a două tipuri fizice. Ar trebui ca mai întâi cele două valori si fie convertite la un întreg abstract, să se efectueze înmulțirea, apoi să se facă transformare inversă la tipul fizic inițial.

O variabilă care este declarată de tipul fizic are o valoare inițială implicită care este valoarea cea mai din stânga din gama tipului respectiv. De exemplu, valorile inițiale pentru tipurile declarate mai sus sunt 0 ohm pentru resistance şi 0 um pentru length.

Time

Tipul fizic predefinit în VHDL, time (timp), este foarte important pentru că este folosit în specificarea întârzierilor. Definiția acestui tip este:

type time is range în funcție de implementare *units*

```
 fs;       -- femptosecunda
ps = 1000 fs;    -- picosecunda
ns = 1000 ps;    -- nanosecunda
us = 1000 ns;    -- microsecunda
ms = 1000 us;    -- milisecunda
sec = 1000 ms;   -- secunda
min = 60 sec;    -- minutul
hr = 60 min;     -- ora
```

end units;

În mod implicit, unitatea primară f s este limita de rezoluție folosită atunci când se simulează un model, valorile timpului mai mici decât limita de rezoluție sunt rotunjite la zero unități. Un simulator poate permite selectarea unei unități secundare din time ca rezoluție limită. In acest caz, unitatea pentru toate valorile de tipul time din model nu trebuie să fie mai mică decât rezoluția limită. Atunci când un model este executat, se folosește rezoluția limită pentru a determina precizia cu care sunt reprezentate valorile time.

2.3.5 Tipuri enumerare

De multe ori, atunci când scriem modele ale unor părți hardware la un nivel abstract, este util să se folosească un set de nume pentru valorile codificate ale unor semnale, în locul unor codificări la nivel de bit. Acest lucru se poate realiza în VHDL folosind tipuri enumerare. De exemplu, să presupunem că modelăm un procesor şi dorim să defkiim nume pentru codurile funcțiilor unității arizmetice. O declarație de tip potrivită este:

type alu_function is (disable, pass, add,

subtract, multiply, divide);

Un astfel de tip se numește enumerare pentru căvalorile folosite sunt menționate una după cealaltă într-o listă. Regula de sintaxă pentru definirea tipului enumerare este:

definiție_tip_enumerare <=
((identificator | caracterjiterai) {,...})

Trebuie să existe cel puțin o valoare în tip, şi fiecare valoare poate fi un identificator, ca în exemplul de mai sus, sau un caracter literal. Un exemplu din acest al doilea caz este:

type octal_digit is ('0', '1', '2', '3', '4', '5', '6','7');

Presupunând ca au fost scrise declarațiile de mai sus, putem introduce noi variabile:

variable alu_op : alu_function;
variable last_digit : octal_digit := '0';

şi putem face următoarele asignări:

alu_op := subtract; last_digit := $^{1'}$7';

Diferite tipuri enumerare pot include acelaşi identificator (se numeşte supraîncărcare), istfel încât contextul trebuie să clarifice care tip trebuie folosit. Pentru a ilustra acest fapt, să considerăm următoarele declarații:

```
type logic_level is (unknown, low, undriven, high); variable control:
logic_level;
type water_level is (dangerously_low, low, ok); variable water_sensor:
water__level;
```

In acest fragment, low este supraîncărcat pentru că este membru al ambelor tipuri. Asignările:

```
control := low; water_sensor := low;
```

sunt ambele acceptabile pentru că tipurile variabilelor sunt suficiente pentru a determina care iow trebuie luat în considerare.

Atunci când se declară o variabilă de tipul enumerare, valoarea inițială implicită este elementul cel mai din stânga din lista de enumerare. Deci, unknown este valoarea inițială implicită pentru type_level, iar dangerously_low este valoarea inițială implicită pentru water_level.

Există trei tipuri enumerare predefinite, definite în modul următor:

```
type severity_level is (note, warning, error, failure);

type file_open_status is (open_ok, status_error, name_error, mode_error);
type file_open:kind is (read_mode, write_mode, append_mode);
```

Tipul severity level se foloseşte cu instrucțiunile assert, iar tipurile f ile_open_status şi f ile_open_kind se folosesc în operațiile cu fişiere.

Caractere

în Capitolul 1 am văzut cum se scriu valorile caracterelor literale, aceste valori aparțin tipului enumerare predefinit *character*, care include toate caracterele din setul ISO pe opt biți. definiția acestui tip este prezentată în Figura 2.7. Acest tip este un exemplu al unui tip enumerare care conține ca elemente combinații de identificatori şi caractere literale.

Primele 128 caractere din această enumerare sunt caractere ASCII, care formează un subset al setului ISO. Identificatorii de la nul la usp şi del sunt caractere ASCII de controL care nu pot fi tipărite. Caracterele de la cl28 la cl59 nu au nici un nume standard, aşa încât VHDL le dă numai nume nedescriptive pe baza poziției lor în setul de caractere. Caracterul de la poziția 160 este un caracter spațiu diferit de cel obişnuit, iar caracterul de la poziția 173 este un soft hyphen.

Pentru a ilustra modul în care se poate folosi tipul character vom declara două variabile după cum urmează:

```
variable cmd_char, terminator : character; type character is (
```

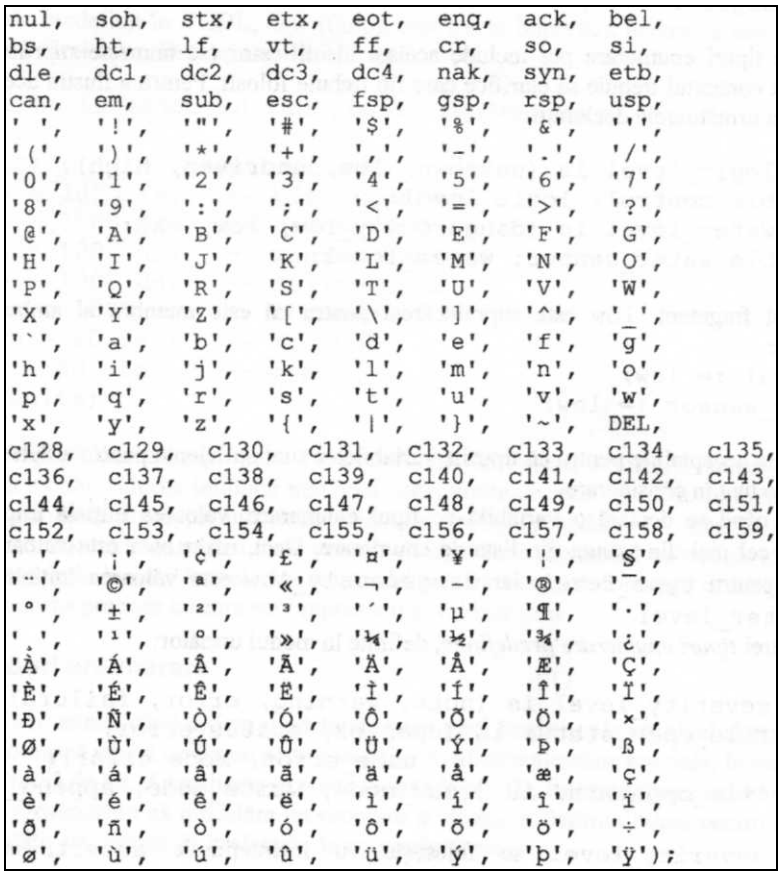


Fig. 2.7 Definiția tipului enumerare predefinit character

şi vom face următoarele asignări:

cmd_char := 'P'; terminator := cr; Boolean

Unul din tipurile enumerare predeftnite în VHDL este tipul boolean definit prin:

type boolean is (false, true);

Acest tip se foloseşte pentru a reprezenta valorile unor condiții prin care se poate controla execuția unui model comportamental. Există mai mulți operatori care se pot aplica valorilor de diferite tipuri şi care conduc la valori booleene; aceştia sunt operatorii relaționali şi logici. Operatorii relaționali, egalitatea ("=") şi neegalitatea ("/="), pot fi aplicați operanzilor de orice tip (cu excepția fişierelor), inclusiv tipurilor compuse. Ambii operanzi trebuie să fie de acelaşi tip iar rezultatul este o valoare booleana. De exemplu, expresiile:

123 = 123, 'A' = 'A', 7 ns = 7 ns,

conduc toate la valoarea TRUE, iar expresiile:

123 = 456, 'A' = 'D[1], 7 ns = 3 ns,

conduc la valoarea FALSE.

Operatorii relaționali, care testează ordinea sunt "mai mic decât" ('<*), "mai mic sau egal cu" ('<='), "mai mare decât" ('>'), "mai mare sau egal cu" ('>=').Aceştia pot fi aplicați numai valorilor tipurilor care sunt ordonate, inclusiv tuturor tipurilor scalare descrise în acest capitol. Ca şi în cazul operatorilor de egalitate şi ncegalitatc, operanzii trebuie să fie de acelaşi tip, iar rezultatul este o valoare booleana. De exemplu, expresiile:

123 < 456, 789 ns <= 789 ns, '1' > '0',
au toate ca rezultat TRUE, iar expresiile:
96 >- 102, 2 ms < 4 s, 'X' < 'X',

au toate ca rezultat FALSE.

Operatorii logici AND, OR, NAND, NOR, XOR, XNOR şi NOT acceptă operanzi care trebuie să aibă valori booleene şi produc rezultate de tip boolean. In Figura 2.8 sunt prezentate rezultatele produse de operatorii logici binari. Rezultatul operatorului NOT este TRUE dacă operandul este FALSE şi este FALSE dacă operandul este TRUE.

Operatorii AND, OR NAND, NOR sunt numiți operatori "de scurt-circuit" pentru că evaluează operandul din dreapta numai dacă operandul din stânga nu impune rezultatul. De exemplu, dacă operandul din stânga operatorului and este FALSE, ştim că rezultatul este FALSE, aşa încât nu mai este nevoie să se considere şi celălalt operand. Această observație este utilă atunci când operandul din stânga este o condiție de test care gardează față de operandul din dreapta, conducând la o eroare.

Să considerăm expresia:
(b /= 0) and (a/b > 1)

Dacă b este zero şi evaluăm operandul din dreapta, putem ajunge la o eroare datorată împărțirii prin zero. Totuşi, pentru că AND este un operator de scurt-circuiț, dacă b este zero, operandul din stânga se va evalua la FALSE, aşa încât operandul din dreapta nu mai este evaluat. Pentru operatorul NAND, operatorul din dreapta nu va fi evaluat dacă operandul din stânga este FALSE. Pentru OR şi NOT, operandul din dreapta nu este evaluat dacă cel din stânga este TRUE.

A	B	A and B	A nand B	A or B	A nor B	A xor B	A xnor B
false	false	false	true	false	true	false	true
false	true	false	true	true	false	true	false
true	false	false	true	true	false	true	false
true	true	true	false	true	false	fasle	true

Fig. 2.8 Tabelul de adevăr pentru operatorii logici binari.

Bit

Pentru că VHDL este folosit la modelarea sistemelor digitale, este util să existe un tip de date pentru a reprezenta valorile biților. Tipul enumerare prédéfinit, bit, serveşte acestui scop. El este definit ca:

type bit is ('0', '1');

Caracterele '0' şi '1' devin astfel supraîncărcate, pentru că ele aparțin atât tipului bit cât şi tipului character. Acolo unde '0' şi T apar într-un model, contextul va determina tipul care va fi folosit.

Operatorii logici care au fost menționați pentru valorile booleene pot fi aplicați şi valorilor de tip bit, şi vor produce rezultate de tip bit. Valoarea '0' corespunde lui FALSE, iar '1' lui TRUE. Aşadar, de exemplu:

'0' and '1' = '0', '1' xor '1' = '0'

Şi de această dată operanzii trebuie să aibă acelaşi tip. Adică, este ilegal să se scrie:

'0' and true

Diferența dintre tipurile boolean şi bit este aceea că valorile booleene se folosesc pentru a modela condiții abstracte, în timp ce valorile bit sunt folosite pentru a modela niveluri logice hardware. Deci, '0' reprezintă un nivel logic LOW iar '1' reprezintă un nivel logic HIGH. Atunci când se aplică valorilor de tip bit, operatorii sunt definiți în logica pozitivă, cu '1' reprezentând starea validată iar '0' starea negată. Dacă este nevoie să lucrăm în logică negativă, trebuie să avem grijă la scrierea expresiilor logice pentru a obține sensu logic corect. De exemplu, dacă write_enable_n, select_reg_n şi write_reg_n sunt variabile bit în logică negativă, putem face asignarea:

write_reg_n := not (not write_enable_n and not select_reg_n) ;

Variabila write_reg_n este validată ('0') numai dacă write_enable_n este validați şi select_reg_n este validată. în caz contrar, nu este validată, adică are valoarea T.

Standard logic

Întrucât VHDL este proiectat pentru modelarea sistemelor digitale, este necesar să fie incluse tipuri pentru a reprezenta valori codificate digital. Tipul predefinit bit, de mai înainte, se foloseşte în acest scop în modele mai abstracte, unde nu suntem interesați de detalii legate de semnalelor electrice. Totuşi, pe măsură ce modelele sunt rafinate şi sunt incluse mai multe detalii, este nevoie să se ia în considerare proprietățile electrice atunci când sunt reprezentate semnale. Sunt mai multe moduri în care se pot defini tipuri de date care să folosească acestui scop, dar IEEE a standardizat o metodă sub forma unui pachet numit ;r.d_logic_1164. Unul din tipurile definite în acest pachet este tipul enumerare numit std_ulogic, definit prin:

```
type std_ulogic is (   'U',     -- Neinițializat
                       'X',     -- Forțat necunoscut
                       '0',     -- Forțat zero
                       '1',     -- Forțat unu
                       'Z',     -- Înaltă impedanță
                       'W',     -- Slab necunoscut
                       'L',     -- Slab zero
                       'H',     -- Slab unu
                       '-' );   -- Don't care
```

Acest tip poate fi folosit pentru a reprezenta semnale stabilite prin intermediul unor drivere îctive (forțarea puterii), drivere rezistive cum sunt rezistoarele *pull-up* şi *pull-down* (putere slabă) sau drivere **three-state** care includ o stare de înaltă impedanță. Fiecare tip de driver poate stabili o valoare "zero", "unu" sau "necunsocută". O valoare "necunoscută" este stabilită

ie un model atunci când nu este capabil să determine dacă semnalul trebuie să fie "zero" sau 'unu". De exemplu, ieşirea unei porți and este necunoscută dacă intrările sunt comandate de drivere care stabilesc starea de înaltă impedanță. în plus față de aceste valori, valoarea cea mai din stânga a tipului reprezintă valoarea "neinițializat". Dacă declarăm semnale de tipul stdulogic, valoarea lor implicită va fi ' U[1]. Dacă un model încearcă să opereze cu această valoare în locul unei valori logice reale, se va detecta o eroare: sistemul proiectat nu va porni cum trebuie. Valoarea finală din tipul stdulogic este valoarea "don' t care". Aceasta este uneori utilizată de uneltele de sinteză şi poate fi utilizată de asemenea atunci când se definesc vectori de test, pentru a arăta că valoarea unui semnal care trebuie comparat cu un vector de test nu este importantă.

Chiar dacă tipul std_ulogic precum şi alte tipuri din pachetul std_logic_1164 nu sunt efectiv construite în limbajul VHDL, putem scrie modele ca şi cum ele ar fi, cu anumite precauții. Deocamdată, vom introduce la începutul modelului pe care dorim să-1 dezvoltăm următoarea linie:

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

înaintea declarației de entitate sau corpului arhitectural.

Cu această observație, putem crea constante, variabile şi semnale de tipul std_ulogic. Putem asigna valori din acest tip, puetm folosi operatorii AND, OR, NOT, etc. Fiecare dintre aceştia operează asupra valorilor std_ulogic şi returnează ' U', ' X', ' O ' sau ' 1' din std_ulogic. Operatorii sunt "optimişti", în sensul că dacă pot determina un rezultat ' 0 'sau ' 1', în ciuda intrărilor necunoscute, atunci ei fac acest lucru. In caz contrar, ei returnează ' X' sau ' U[1]. De exemplu, ' 0' and ' Z ' returnează '0' pentru că, dacă intrare a unei porți *and* este '0', ieşirea va fi '0' indiferent de cealaltă intrare.

Curs 3

TIPURI DE DATE ŞI OPERAȚII – part 2

3.1 Clasificarea tipurilor

3.2 Atributele tipurilor scalare

3.3 Expresii şi operatori

3.4 Tipuri compuse de date

3.1 Clasificarea tipurilor

În paragrafele anterioare am luat în discuție *tipurile scalare* oferite de VHDL. În Figura 1. sunt prezentate relațiile dintre tipurile scalare şi celelalte tipuri pe care le vom lua în discuție în paragrafele următoare următoare.

3.1.1 Subtipuri

Adesea, modelele conțin obiecte care trebuie să ia valori într-o *gamă restrânsă a unui set mai larg. Putem reprezenta astfel de obiecte prin declararea unui* **subtip**, *care defineşte un set restricționat de valori ale unui* **tip de bază**. Condiția care determină care anume valori sunt în subtipul definit se numeşte *restricție.* Folosind o declarație de subtip, va deveni clar intenția noastră de a folosi numai anumite valori şi va fi posibil să se verifice faptul că valori invalide nu sunt folosite. Regula de sintaxă pentru o declarație de subtip este:

declarație_de_subtip <=
***subtype* identificator *is* indicație_de_subtip;**
unde
indicație_de_subtip <=
nume [range expresie simplă (*to* | *downto*) expresie_simpiă]

Declarația defineşte identificatorul ca un subtip al tipului de bază, cu precizarea unei game restrânse de valori, restricția este opțională, ceea ce înseamnă că *este posibil să avem un subtip care include toate valorile tipului de bază.*

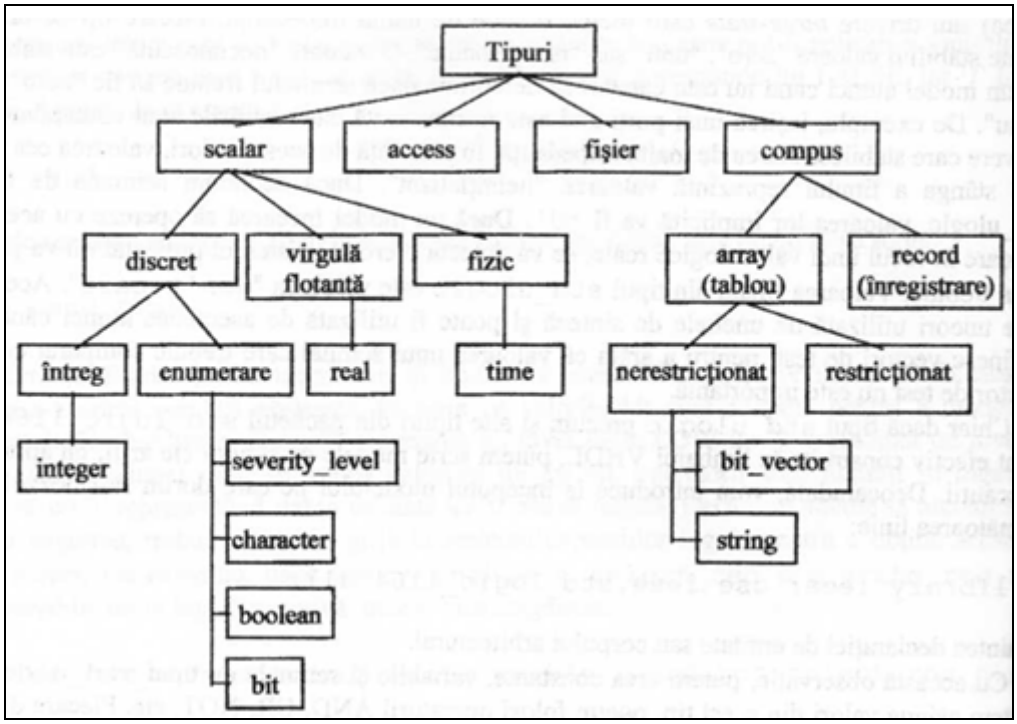


Figura 1 Clasificarea tipurilor de date în VHDL

Exemplu

Iată o declarație care definește un subtip al tipului integer:

```
subtype small:int is range -128 to 127;
```

Valorile lui *small_int* sunt restricționate să fie în gama -128...127. Dacă declarăm următoarele variabile:

```
variable deviation: small_int;
variable adjustment: integer;
```

putem să le folosim in calcule precum:

```
deviation := deviation + adjustment;
```

Se observă că în acest caz este posibil să combinăm valorile subtipului cu cele ale tipului de bază (în sumă) pentru a produce o valoare de tip integer, dar rezultatul trebuie să fie în gama -128...127 pentru ca asignarea să fie corectă. În caz contrar, se va semnala o eroare la asignarea variabilei. Toate operațiile care se aplică tipului de bază pot fi folosite şi pentru valorile unui subtip. Operațiile produc valori ale tipului de bază şi nu ale subtipului. Totuşi, operația de asignare nu va asigna o valoare unei variabile din subtip dacă valoarea obținută nu satisface restricția impusă.

Un alt lucru care trebuie menționat este că *dacă tipul de bază are precizată o gamă într-o anumită direcție (ascendent sau descendent), iar un subtip este specificat cu o restricție de gamă în direcție opusă, specificația de subtip este aceea care contează*. De exemplu, tipul predefinit integer are gama ascendentă. Dacă declară un subtip prin:

```
subtype bit_index is integer range 31 downto 0;
```

acest subtip are gama descendentă.

*Standardul VHDL include două **subtipuri predefinite** pentru tipul integer, definite prin:*

```
subtype natural is integer range 0 to cel mai mare întreg;
```

```
subtype positive is integer range 1 to cel mai mare întreg;
```

În cazul în care logica unui proiect indică faptul că un număr trebuie să fie nenegativ, se obişnuieşte să se folosească unul din aceste două subtipuri în locul tipului de bază *integer*. În acest fel putem detecta orice eroare care va conduce la producerea unor numere negative. Există şi un subtip predefinit pentru tipul fizic *time*, definit prin:

```
subtype delay_length is time range 0 fs to cea mai mare valoare time;
```

Acest subtip ar trebui să fie folosit ori de câte ori este nevoie de o întârziere nenegativă.

3.1.2 Calificarea tipului

Uneori, nu este clar din context care este tipul unei valori particulare. În cazul valorilor de tip enumerare supraîncărcate, *poate fi necesar să se specifice în mod explicit tipul* despre care este vorba. Putem face acest lucru folosind ***expresia de calificare a tipului***, care *constă din scrierea numelui tipului urmat de un singur caracter apostrof, şi de o expresie inclusă între paranteze.* De exemplu, dacă se dau tipurile enumerare:

```
type logic_level is  (unknown,  low, undriven, high);
```

```
type system_state is  (unknown,  ready, busy);
```

putem face distincția între valorile comune scriind:

```
logic_level'(unknown), system_state'(unknown)
```

Calificarea tipului poate fi folosită şi pentru a restrânge o valoare la un subtip particular al unui tip de bază.

De exemplu, dacă definim un subtip al tipului *logic_level*:

```
subtype valid_level is logic_level range low to high;
```

putem specifica în mod explicit o valoare fie a tipului fie a subtipului:

```
logic_level'(high),        valid_level'(high)
```

3.1.3 Conversia de tip

Atunci când am introdus operatorii aritmetici în paragraful precedent, am spus că *operanzii trebuie să fie de acelaşi tip*. Aceasta face imposibilă combinarea valorilor întregi şi a celor în virgulă flotantă în expresiile aritmetice. Dacă avem nevoie de astfel de combinații, trebuie să folosim conversia de tip pentru a efectua conversia de la valori întregi la virgulă flotantă. *Forma unei conversii de tip este: numele tipului la care dorim să convertim, urmat de o valoare inclusă între paranteze*. De exemplu, pentru a converti între tipurile integer şi real, putem scrie

```
real (123),     integer (3.6)
```

Conversia unui întreg la o valoare în virgulă flotantă este doar o schimbare a reprezentării, deşi se poate pierde în precizie. Conversia de la o valoare în virgulă flotantă la un întreg presupune rotunjirea la cel mai apropiat întreg. Conversiile tipurilor numerice nu sunt singurele acceptate; în general, putem converti între orice tipuri apropiate.

Un lucru căruia trebuie să-i acordăm atenție este distincția dintre calificarea tipului şi conversia tipului. Prima declară pur şi simplu tipul unei valori, în timp ce a doua schimbă valoarea, (posibil) la un tip diferit.

3.2 Atributele tipurilor scalare

Un tip defineşte un set de valori şi un set de operații aplicabile. Există de asemenea un set de *atribute* predefinite care sunt folosite pentru a oferi informație despre valorile incluse în acel tip. *Numele atributelor sunt scrise după numele tipului şi un caracter apostrof. Valoarea unui atribut poate fi folosită în calculele din interiorul unui model.* Vom lua în discuție unele atribute pentru tipurile pe care le-am introdus în acest capitol.

Mai întâi, există un număr de atribute care sunt aplicabile la toate tipurile scalare şi oferă informații despre gama valorilor din acel tip. Dacă notăm cu T un tip (oarecare) sau subtip scalar, prin x o valoare din acel tip şi prin S un şir, atributele sunt:

T'left	prima (cea mai din stânga) valoare din T
T'right	ultima (cea mai din dreapta) valoare din T
T'low	cea mai mică valoare din T
T'high	cea mai mare valoare din T
T'ascending	true, dacă **T** este în gamă crescătoare, false în caz contrar
T'image(x)	un şir care reprezintă valoarea lui x
T'value(s)	valoarea din **T** care este reprezentată de s

Şirul produs de atributul 'image este un literal . Şirurile acceptate în atributul 'value trebuie să respecte regulile şi pot include spații. Aceste două atribute sunt utile pentru intrările şi ieşirile unui model .

Exemplu

Pentru a ilustra modul de folosire a atributelor listate mai sus, vom include unele declarații din exemplele anterioare:

```
type resistance is range 0 to 1E9 units
ohm;
kohm = 1000 ohm;
Mohm = 1000 kohm;
end units resistance;

type set_index_range is range 21 downto 11;

type logic_level is   (unknown,  low,  undriven,  high);
```

Pentru aceste tipuri putem determina următoarele atribute:

```
resistance'left = 0 ohm
resistance'right = 1E9 ohm
resistance'low = 0 ohm
resistance'high = 1E9 ohm
resistance'ascending = true
resistance'image  (2 kohm) = "2000 ohm"
resistance'value ("5 Mohm")=5_000_000 ohm

set_index_range'left        =        21
set_index_range'right       =        11
set_index_range'low         =        11
set_index_range'high        =        21
set_index_range'ascending    =    false
set_index_range'image    (14)    =  "14"
set_index_range'value  ("20") = 20

logic_level'left = unknown
logic__level[1] right = high
logic_level[1] low = unknown
logic_level'high = high
logic_level'ascending = true
logic_level'image  (undriven) = "undriven"
logic level'value  ("low")  = low
```

*Există atribute care sunt aplicabile numai **tipurilor fizice şi discrete***. Pentru orice astfel de tip **T,** o valoare **x** din acel tip şi un întreg **n,** atributele sunt:

T'pos(x)	numărul poziției lui x în T
Tval(n)	valoarea din T de la poziția n
T'succ(x)	valoarea din T la poziția mai mare cu unu decât a lui x
T'pred(x)	valoarea din T la poziția mai mică, cu unu decât a lui x
Tleftof(x)	valoarea din T la poziția din stânga lui x
T'rightof(x)	valoarea lui T la poziția din dreapta lui x

Pentru tipurile enumerare, numerele pozițiilor încep cu zero, pentru primul element din listă, şi creşte cu unu pentru fiecare element către dreapta. Deci, pentru tipul *logic_level* definit mai înainte, unele atribute sunt:

```
logic_level'pos(unknown) = 0
logic_level'val(3)  = high
logic_level'succ(unknown)=low
logic_level'pred(undriven)  = low
```

Pentru tipurile întregi, numărul poziției este acelaşi cu valoarea întreagă, dar tipul numărului poziției este un tip special anonim numit *întreg universal.* Acesta este acelaşi tip ca şi cel al întregilor literali şi, dacă este necesar, este convertit implicit la orice alt tip întreg declarat. *Pentru tipuri fizice, numărul poziției este numărul întreg al unităților de bază în valoarea fizică.* De exemplu:

```
time'pos(4 ns)  = 4_000_000
```

pentru că unitatea de bază este fs.

Exemplu

Putem folosi atributele ' pos şi ' val în combinație, pentru a efectua operații aritmetice cu operanzi de tip fizic cu dimensiuni diferite, producând un rezultat corect dimensional. Să presupunem că definim tipuri fizice care reprezintă lungimea (length) şi aria (area):

```
type length is range integer'low to integer'high
  units mm;
  end units length;
type area is range integer'low to integer'high units
     square_mm;  end  units
  area;
```

şi variabile de aceste tipuri:

```
variable LI, L2 :  length;
 variable A : area;
```

Restricțiile asupra valorilor de multiplicat în cazul tipurilor fizice ne împiedică să scriem.

A : = LI *L2; - această asignare este incorectă

Pentru a obține răspunsul corect, putem să convertim aceste valori la întregi abstracți folosind atributul ' pos apoi să convertim rezultatul înmulțirii la o valoare din area folosind ' va 1, după cum urmează:

A:= area'val(length'pos(LI)*length'pos(L2));

Observăm în acest exemplu simplu că nu este nevoie să includem un factor de scală în înmulțire pentru că unitatea de bază pentru area este pătratul unității de bază din length.

Pentru game ascendente, T'succ(x) şi T'rightof (x) produc aceeaşi valoare iar T'pred(x) şi T'leftof(x) produc aceeaşi valoare. Pentru game descendente, T'pred(x) şi T' rightof (x) produc aceeaşi valoare iar T'succ(x) şi I'leftof(x) produc aceeaşi valoare. Pentru toate gamele, T' succ (T' high), r'pred(T'low), T'rightof (T' right) şi T'leftof (T' left) determină apariția unei erori.

Ultimul atribut pe care-1 introducem aici este T' base. *Pentru orice subtip T, acest atribut produce tipul de bază al lui T.* Singurul context în care acest atribut se poate folosi este ca prefix pentru un alt atribut. De exemplu, dacă avem declarațiile:

```
type opcode is  (nop,  load,  store,  add,  subtract,
               negate,  branch,  halt);
subtype arith_op is opcode range add to negate;
```

atunci

```
arith_op'base'left = nop;
arith_op'base'succ(negate)  = branch;
```

3.3 Expresii şi operatori

În acest paragraf vom prezenta un sumar al regulilor care guvernează expresiile. *Putem gândi o expresie ca fiind o formulă care specifică modul în care se calculează o valoare. O expresie constă din valori primare combinate cu operatori.* Valorile primare care pot fi folosite în expresii includ:

- valori literale
- identificatori care reprezintă obiecte (constante, variabile, etc.)
- atribute care conduc la valori
- expresii calificate
- expresii convertoare de tip
- expresii în paranteze

În Figura 2 sunt prezentați toți *operatorii şi toate tipurile cărora le pot fi aplicați*. *Operatorii sunt grupați după prioritate, cu* **, `abs` *şi* `not` *având prioritatea cea mai mare iar operatorii logici având prioritatea cea mai mică* . Aceasta înseamnă că dacă o expresie conține o combinație de operatori, mai întâi se aplică aceia cu prioritatea cea mai mare. Se pot utiliza paranteze pentru a schimba ordinea de evaluare.

Operator	*Operație*	*Tip operand stânga*	*Tip operand dreapta*	*Tip rezultat*
**	exponențială	întreg sau virgulă flotantă	întreg	același cu al operandului din dreapta (AOD)
abs	valoare absolută		numeric	același cu operandul
not	negație		bit, boolean sau tablou 1D de elemente bit sau boolean	același cu operandul
*	multiplicare	fizic întreg sau virgulă flotantă	același cu al operandului din stânga (AOS)	același cu operanziii (AO)
/	divizare	întreg sau virgulă flotantă	AOS	AOS
mod	modulo	fizic întreg sau virgulă flotantă	AOS	AO
rem	rest	întreg	AOS	AO
+	identitate		numeric	același cu al operandului
-	negație		numeric	același cu al operandului
+	adunare	numeric	AOS	AO
-	scădere	numeric	AOS	AO
&	concatenare	tablou 1D	AOS	AO
sll	deplasare logică stânga	tablou 1D de elemente bit sau boolean	întreg	AOS
srl	deplasare logică dreapta			
sla	deplasare aritmetică stânga			
sra	deplasare aritmetică dreapta			
rol	rotație stânga			
ror	rotație dreapta			
=	egalitate	oricare cu excepția file	AOS	boolean
/=	neegalitate			
<	mai mic decât		AOS	boolean
<=	mai mic sau egal cu			
>	mai mare decât			
>=	mai mare sau egal cu			
and	AND logic	bit, boolean sau tablou 1D de elemente bit sau boolean	AOS	AO
or	OR logic			
nand	NAND logic			
nor	NOR logic			
xor	OR exclusiv			
xnor	NOR exclusiv			

Fig. 2 Operatorii VHDL în ordinea priorității

3.4 Tipuri compuse de date

Datele compuse au valori complexe. De exemplu, un tablou unidimensional de întregi poate avea valoarea (15, 35, 2, 295). *Valoarea complexă are mai multe componente şi reprezintă un tablou de valori între care există anumite relaţii*.

3.4.1 Tablouri

Un **tablou** *este o colecţie de valori de* ***acelaşi tip****. Spunem că elementele unui tablou sunt* **omogene**. *Poziţia fiecărui element în tablou este dată de valoarea unui scalar numit index.* Pentru a forma un tablou într-un model, mai întâi se defineşte tipul tabloului într-o declaraţie de tip. Regula de sintaxă pentru definirea unui tip tablou este:

definire_tip_tablou <= *array* **(gamă_discretă {**...**}**) *of* **indicaţie_subtip_element**

În acest mod se poate defini un tablou prin specificarea uneia sau a mai multor game de indici (lista de game discrete) împreună cu tipul sau subtipul elementelor. Ne amintim din cursurile anterioare că o gamă discretă este un subset de valori dintr-un tip discret (un tip întreg sau enumerare), şi că poate fi specificat prin regula:

gamă discretă <=
indicaţie_subtip_discret | expresie_simplă (to | downto**) expresie_simplă**

indicaţie_subtip <=
marca_tip [range **expresie_simplă** (to | downto **) expresie_simplă]**

Există două tipuri predefinite de date de tip tablou.

Tipul de date *String* este un tablou cu elemente de tip *Character*. Acesta este folosit pentru a forma texte care trebuie tipărite şi tipul de date *Bit_Vector* care este un tablou de biţi. *Acesta este folosit pentru a descrie date memorate în registre sau date care se transferă pe bus-uri paralele*. Definiţiile acestora sunt:

```
type String is array  (Positive range <>)  of Character;

type Bit_Vector is array  (Natural range <>)  of Bit;
```

Gama indexului este plasată între paranteze după cuvântul cheie *array*. Gama indexului pentru tipul *String* este de tipul *Positive. Notaţia <> înseamnă că gama este nerestricţionată.* Efectul este acela că utilizatorul trebuie să indice gama indexului atunci când declară un obiect de tipul *String*, respectând condiţia de întreg pozitiv. In mod similar, gama indexului pentru *Bit_Vector* este nerestricţionată. Obiectele de tip *Bit_Vector* pot avea indexul 0 dar cele de tip *String* nu pot avea indexul 0.

Vom ilustra modul în care se declară tablourile cu ajutorul unor exemple:

```
type color is (RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET);
type std_logic_vector is array (natural range <>) of std_logic;
type register_32_Bit is array (31 downto 0) of Bit;
                                        -- gamă descrescătoare a indexului

type counter_16_Bit is array (0 to 15) of Bit;
                                        -- gamă crescătoare a indexului

subtype BitVect3 is Bit_Vector (O to 2);
type color_count is array (color range <>) of Natural;
type array_of_colors is array (Natural range <>) of color;
type RCM_type is array (Natural range <>) of register_32JBit;
type array_2D is array (Natural range <>, Natural range <>) of
Bit;
```

Tipul `std_logic_vector` *este un tip de dată nerestricționat, folosit pentru a reprezenta registre sau bus-uri atunci când se foloseşte sistemul de valori logice (de exemplu,* ***în modele care trebuie să fie sintetizate****).* Tipul tablou `register_32_bit` este un tablou unidimensional restricționat, de `32` biți, numerotați de la 31 la 0 (de la stânga la dreapta). De exemplu, dacă o dată din tipul `register 32 bit` are valoarea:

```
B"llll 1101 0101 1110 0000 0101 1100 0110"
```

atunci bitul 31 (cel mai din stânga) are valoarea ‚1', bitul 23 are valoarea '0', bitul 2 are valoarea T, bitul 0 (cel mai din dreapta) are valoarea '0'. Spunem că gama indexului pentru tipul `register_32_bit` este descrescătoare.

Similar, tipul tablou `counter_16_bit` este un tablou *unidimensional* de 16 biți. restricționat, cu pozițiile biților numerotate în ordine crescătoare de la 0 la 15 (de la stânga la dreapta). Dacă un obiect de tipul `counter_16_bit` are valoarea:

```
B"1011_0010_1011_0010"
```

atunci bitul 0 (din stânga) are valoarea '1', bitul 4 are valoarea '0', iar bitul 15 (din dreapta) are valoarea '0'.

Pentru că `Bit_Vector` este un tip de date nerestricționat, se pot declara subtipuri cu gama specificate de indici. `Bit_Vect3` este un subtip al tipului `Bit_Vector` cu o gama crescătoare a indexului de la 0 la 2.

Tipul tablou `color_count` este un tablou de numere naturale. Gama indexului este nerestricționată, de tipul `color`, ceea ce permite utilizatorului să selecteze orice gama doreşte atunci când declară un obiect de tipul `color_count`. Să presupunem că a fost selectată gama `RED to YELLOW` pentru an obiect. în acest caz, tabloul constă din trei numere naturale. Secvența de indici este `RED, ORANGE, YELLOW`. De exemplu, valoare

tabloului poate fi (24, 35, 0). în acest caz, elementul cu indexul RED este 24, elementul cu indexul ORANGE este 35 iar elementul cu indexul YELLOW este 0.

În mod similar, tipul tablou array_of_colors este un tablou de colors. Gama indexului este nerestricționată dar trebuie să fie numere naturale. Să presupunem că a fost declarată o gamă a indexului 3 to 5. Dacă valoarea tabloului este (BLUE, ORANGE VIOLET), elementul cu indexul 3 are valoarea BLUE, elementul cu indexul 4 are valoare ORANGE iar elementul cu indexul 5 are valoarea VIOLET.

Sunt două moduri posibile pentru a declara un tablou unidimensional. Tipul tablou ROM_type *este un tablou de tablouri. Fiecare element din tablou are tipul* register_32_bit. Cu alte cuvinte, ROM_type este un tablou unidimensional de cuvinte de 32 biți. O valoare tipică pentru o entitate de tipul ROM_type constă din patru cuvinte, cum ar fi:

(X"2F3C 5456", X"FF22_A5B9", X"9900_AD51", X"FFFF_FFFF")

Cu alte cuvinte, *fiecare element al tabloului este o cantitate reprezentată pe 32 biți.*

Spre deosebire de această declarație, următoarea arată că array_2D este un tablou bidimensional în care fiecare element este de tipul Bit. Dacă un obiect este declarat ca având cinci linii şi 10 coloane, o valoare tipică pentru această entitate este:

(1,0, 1, 1, 0, 0, 0, 1, 1, 1,
0, 1, 1, 1, 0, 1, 1, 0, 1,
1, 1, 0, 1, 0, 1, 0, 0, 1, 1,
1, 1, 1, 0, 0, 0, 1, 1, 0,0,
0, 0, 0, 0, 1, 1, 1, 0, 0, 1)

In acest caz, fiecare element al tabloului este un singur bit.

3.4.2 Atributele tablourilor

Atributele aplicabile tablourilor fac referire la informație legată de gama indexului. Să considerăm un tip tablou sau obiect **A,** şi un întreg **N** între 1 şi numărul de dimensiuni ale lui **A.** VHDL defineşte următoarele atribute :

A'range(N)	returnează gama indexului dimensiunii N din A
A'high(N)	returnează cea mai mare valoare a gamei indexului pentru dimensiunea N din A
A'low(N)	returnează cea mai mică valoare a gamei indexului pentru dimensiunea N din A
A'right(N)	returnează valoarea cea mai din dreapta a gamei indexului pentru dimensiunea N din A
A**'left(N)**	returnează valoarea cea mai din stânga a gamei indexului pentru dimensiunea N din A
A'length(N)	returnează numărul de elemente (lungimea) din gama

	dimensiunii N din A
A'reverse_range(N)	inversează gama indexului dimensiunii N din A
A'ascending(N)	returnează TRUE dacă gama indexului dimensiunii N din A este crescătoare, FALSE în caz contrar.

`A ' range` este o funcție care returnează gama indexului pentru un tablou A. Aceasta permite crearea unui proces care examinează elementele `unui` tablou fără a avea grijă care ar putea fi gama curentă a tabloului. De exemplu, următorul proces contorizează numărul de 1 dintr-un tablou unidimensional `DBUS` cu elemente `Bit_Vector.` Gama poate fi crescătoare sau descrescătoare. Procesul funcționează fără a avea importanță numărul de elemente din gama indexului.

```
process_range: process   (DBUS)  variable
    count3: integer := 0;
begin
count3 := 0;
  for I in DBUS'range loop
   if DBUS(I)  = '1' then count3 := count3 + 1;
   end if;
  end loop;
end process process_range;
```

Pentru tabloul *bidimensional* declarat în modul următor:

```
type A is array (1 to 4, 31 downto 0) of boolean;
```

unele valori ale atributelor sunt:

```
A'left(1) = 1
A'right (1)  = 0
A'range (1) = 1 to 4
A'length(1) = 4
A'ascending(l) = true
A'low(l) = 1
A'high(2)  = 31
A'reverse_range(2) = 0 to 31
A'length(2) = 32
A'ascending(2) = false
```

Pentru toate aceste atribute, dacă dorim să ne referim la prima dimensiune (sau este numai o singură dimensiune), putem omite numărul dimensiunii din paranteze, de exemplu:

A'low = 1 A'legth = 4

Dacă `DBUS` este declarat prin:

```
  signal DBUS: Nit_Vector    (15 downto 0);

atributele tabloului au următoarele valori:
  DBUS'right = 0
  DBUS'left = 15
  DBUS'high = 15
  DBUS'low = 0
  DBUS'length = 1 6
```

Următorul proces foloseşte două dintre aceste atribute pentru a calcula numărul de 1 din semnalul `DBUS`. Procesul funcționează pentru orice semnal de tip `Bit_Vector`, fără a avea importanță gama valorilor indexului. Gama poate fi crescătoare sau descrescătoare:

```
process_length: process   (DBUS)
variable count2:  integer := 0;
begin
     count2   := 0;
  for I in 0 to DBUS'length -l loop
    if DBUS(DBUS'low+1)    = '1'  then
    count2   := count2 + 1;
    end if;
  end loop;
end process;
```

3.4.3 Înregistrări *(record)*

Tipul record *este un tip compus în care elementele sunt* **heterogene**; *adică, elementele pot avea tipuri diferite*. Ca şi în tipurile tablou, elementele formează o listă ordonată. De exemplu să considerăm următoarea declarație pentru o înregistrare folosită pentru a reține o dată:

```
  type numele_lunii is   (IAN,   FEB, MAR, APR, MAI,   IUN,
                      IUL, AUG,   SEPT,   OCT,   NOV,   DEC);
  type DATA is record
    ZIUA: Integer range 1 to 31;
    LUNA: numele_lunii;
    ANUL:  Integer range 0 to 3000;
end record;
```

O valoare tipică pentru un obiect de tipul DATA este:

```
(16, AUG,  2006)
```

```
unde ZIUA = 16, LUNA = AUG,   ANUL = 2006.
```

3.4.4 Tipuri *access*

Aceste tipuri se folosesc în conjuncție cu procesele de alocare şi dealocare a memoriei în aplicațiile dinamice, legate de liste şi arbori de decizie. Aceste tipuri nu vor fi luate în discuție în aceast curs.

3.4.5 Tipuri *file*

Aceste tipuri sunt folosite pentru a defini obiecte care reprezintă conținutul unor fişiere din sistemul de calcul. *Fişierele pot fi utile pentru a reține date folosite de programele VHDL*. De exemplu, vectorii de test sunt memorați frecvent în fişiere text. Aceste tipuri vor fi luate în discuție în următoarele cursuri.

Curs 4

Instrucțiunile limbajului VHDL

Instrucțiuni secvențiale I

1. **Instrucțiunea CASE**
2. **Istrucțiunea WAIT**
3. **Instrucțiunea IF**

Începând cu acest curs vom prezenta instrucțiunile limbajului VHDL. Există două clase de instrucțiuni. ***Instrucțiunile secvențiale*** *se folosesc în procese şi subprograme pentru a descrie algoritmi*. Ele se execută secvențial, cu excepția cazurilor când se întâlnesc instrucțiuni de salt., în mod similar cu instrucțiunile oricărui alt limbaj procedural (cum este de exemplu C). ***Instrucțiunile concurente*** sunt folosite în corpurile arhitecturale pentru a modela semnale. *Spre deosebire de limbajele tipice de programare, aceste instrucțiuni* ***nu sunt executate în ordinea în care sunt scrise; ele sunt executate numai când se modifică semnalele care le comandă***. In plus, *toate instrucțiunile concurente sunt executate o dată la începutu simulării*. *Ordinea în care instrucțiunile concurente apar în corpul arhitectural nu este importantă*. Unele instrucțiuni sunt atât concurente cât şi secvențiale. În Figura 1 se prezintă clasificarea intsrucțiunilor VHDL. Unele dintre acestea au fost introduse şi prezentate în cursurile anterioare.

Instrucțiuni secvențiale	**Instrucțiuni concurente**
Asignare de variabile	
Asignare de semnale	Asignare de semnale
Assert	Assert
Apelare de proceduri şi funcții	Apelare de proceduri
If...then...else	Process
Case	Block
Loop	Instanțiere componente
Wait	Generate
Null	
Next	
Exit	
Return	

Fig. 1 Instrucțiuni secvențiale şi concurente.

4.1 Instrucțiuni secvențiale

Aceste instrucțiuni sunt folosite în procese şi subprograme pentru a implementa algoritmi. Ele sunt executate în ordinea în care sunt scrise.

4.1.1 Instrucțiunea CASE

Atunci când avem un model în care comportarea trebuie să depindă de valoarea unei singure expresii, putem folosi o instrucțiune case. Regulile de sintaxă sunt următoarele:

```
instructiune_case <=

[eticheta_case:] case expresie is
    ( when variante => {instructiune_secventiala})
    {...}
  end case [ eticheta_case ];

  variante <= ( expresie_simpla | gama_discreta | others ) { |...}
```

Eticheta poate fi folosită pentru a identifica instrucțiunea case. Mai întâi, să presupunem că modelăm **o unitate aritmetică şi logică**, a cărei intrare de control, *func*, este declarată de tipul enumerare:

```
type alu_function is  (pass1, pass2,  add,  subtract);
```

Putem descrie comportarea folosind o instrucțiune *case* :

```
case func is
when pass1 => result := operand1;
when pass2 => result  := operand2;
when add => result := operandl+operand2;
when pass2 => result := operandl-operand2;
end case;
```

La începutul acestei instrucțiuni se află expresia care ***selectează*** variantele posibile; este situată între cuvintele cheie *case* şi *is*. In acest exemplu, este vorba despre o expresie simplă care constă numai dintr-o valoare primară. Valoarea acestei expresii se foloseşte pentru a selecta instrucțiunile care vor fi executate. Corpul instrucțiunii *case* constă dintr-o serie de *alternative,* fiecare începând cu cuvântul cheie *when*, urmat de una sau mai multe variante şi o secvență de instrucțiuni. Variantele sunt valori ce sunt comparate cu valoarea expresiei selectoare. *Trebuie să existe exact o variantă pentru fiecare valoare posibilă a expresiei selectoare.* Instrucțiunea *case* găseşte alternativa cu aceeaşi valoare a variantei ca şi expresia selectoare şi execută instrucțiunile acelei variante. În acest exemplu, variantele sunt toate expresii simple de tipul *alu_func*. Dacă valoarea lui *func* este *passl*, se execută instrucțiunea

result := operandl; dacă valoarea este *pass2*, se va executa instrucțiunea *result := operand2*; etc.

O instrucțiune *case* prezintă anumite similarități cu o instrucțiune *if* (pe care o vom studia mai târziu): ambele realizează o selecție dintre grupuri alternative de instrucțiuni secvențiale. Diferența constă în modul în care sunt alese instrucțiunile care trebuie să fie executate. Vom vedea că o *instrucțiune* ***if*** *evaluează* ***condiții booleene succesive*** *până când se găseşte una adevărată. Apoi se execută grupul de instrucțiuni ce corespunde acelei condiții. O instrucțiune* ***case*** *evaluează* ***o singură expresie selectoare*** *pentru a obține o valoare selectoare*. Valoarea expresiei se compară cu valorile diferitelor variante pentru a determina ce instrucțiune trebuie să fie executată. O instrucțiune *if* oferă un mecanism mai general pentru a face selecția diferitelor alternative, pentru că condițiile pot fi expresii booleene arbitrar de complexe. Totuşi, instrucțiunile *case* reprezintă un mecanism important şi util**,** după cum vom arăta în exemplele următoare.

Expresia selectoare din instrucțiunea ***case*** *trebuie să conducă la o valoare de tipul discret sau la un tablou unidimensional de elemente de tip caracter*, cum sunt şirurile de întregi sau şirurile de biți. Astfel, putem avea o instrucțiune case care selectează o alternativă pe baza unei valori întregi. Dacă presupunem că index_mode şi instruction_register sunt declarate ca:

```
subtype index_mode is integer range 0 to 3;
variable instruction_register :  integer range 0 to 2**16-1;
```

Atunci**,** putem scrie o instrucțiune *case* care foloseşte o valoare de acest tip:

```
case index_mode'((instruction_register/2**12) rem 2**2) is
when 0 =>index_value := 0;
when 1 =>index_yalue := accumulator_A;
when 2 =>index_value := accumulator_B;
when 3 =>index_value := index_register;
end case;
```

De menționat că, în acest exemplu, folosim o expresie *calificată* în expresia selectoare. Dacă am fi omis acest lucru, rezultatul evaluării expresiei ar fi fost un întreg şi ar fi trebuit să includem alternative pentru a acoperi toate valorile întregi posibile. Tipul *calificare* elimină această necesitate prin limitarea valorilor posibile ale expresiei.

O altă regula care trebuie amintită este aceea că ***tipul fiecărei alternative trebuie să fie de acelaşi tip ca şi tipul care rezultă din expresia selectoare***. Aşadar, în exemplul de mai sus, este ilegal să introducem o alternativă cum ar fi

```
when 'a' =>….   -- ilegal
```

pentru că alternativa listată nu poate fi un întreg. Putem include mai multe alternative (variante) prin scrierea acestora separate prin simbolul |. De exemplu, dacă tipul opcodes se declară prin:

```
type opcodes is  (nop, add, subtract,  load, store, jump,
                  jumpsub, branch, halt);
```

putem scrie o alternativă care să includă trei dintre aceste valori ca variante posibile:

when load | add | subtract =>

operand := memory_operand;

Dacă avem un număr de alternative într-o instrucțiune case şi dorim să includem o alternativă pentru a acoperi toate variantele posibile ale expresiei selectoare care nu au fost menționate în alternativele precedente, putem folosi o variantă specială ***others***. de exemplu, dacă variabila *opcode* este o variabilă de tipul *opcodes*, declarată mai sus, putem scrie:

```
case opcode is
  when load  |  add  |  subtract =>
    operand := memory_operand;
  when store  |  jump  |  jumpsub  |  branch =>
operand := address_operand;
when others => operand := 0;
end case;
```

În acest exemplu, dacă valoarea lui *operand* este oricare alta decât cele listate în prima şi a doua alternativă, se va selecta ultima alternativă. Trebuie să existe numai o singură alternativă care foloseşte varianta *others*; aceasta trebuie să fie ultima alternativă a instrucțiunii *case*. O alternativă care include varianta *others* nu poate include alte variante. De menționat că dacă toate valorile posibile ale expresiei selectoare sunt acoperite de variantele anterioare, putem totuşi include varianta *others*, dar aceasta nu va fi abordată niciodată.

Ultima formă a variantelor pe care nu am menționat-o este o gamă discretă, specificată prin următoarele reguli sintactice:

```
gamă_discretă <=
 indicația_subtip_discret
 | expresie_simplă (to | downto ) expresie_simplă
indicația_subtip <= type_mark
 [ range expresie_simplă (to | downto) expresie_simplă ]
```

Aceste forme ne permit să specificăm o *gamă de valori într-o alternativă a unei instrucțiuni* ***case***. Dacă valoarea expresiei de selecție se potriveşte cu una din valorile din gamă, se execută instrucțiunile din alternativa respectivă. Cea mai simplă cale de a specifica o gamă discretă este scrierea limitelor din stânga şi din dreapta ale gamei, separate printr-un cuvânt cheie de direcție. De exemplu, instrucțiunea ***case*** de mai sus poate fi rescrisă sub forma echivalentă:

```
case opcode is
  when add to load =>    --gamă de valori
     operand := memory_operand;
  when branch downto store =>
```

```
    operand := address_operand;
  when others => operand := 0;
end case;
```

O altă cale de a specifica o *gamă discretă este prin folosirea numelui tipului discret* şi, posbil, a unei restricţii de gamă pentru a reduce valorile la o submulțime de acest tip. de exemplu, dacă declarăm un subtip de opcodes ca:

```
subtype control_transfer_opcodes is opcodes range jump to
branch;
```

putem rescrie a doua alternativă sub forma

```
when control_transfer_opcodes  |  store =>
  operand := address_operand;
```

Menționăm că putem folosi o gamă discretă ca variantă numai dacă expresia selectoare este de tip discret. Nu avem voie să folosim o gamă discretă dacă expresia selectoare este de un tip tablou, cum ar fi tipul bit-vector. Dacă specificăm o gamă prin scrierea limitelor şi o direcție, direcția nu are altă semnificație decât să identifice conținutul gamei.

O mențiune importantă legată de variantele dintr-o instrucțiune ***case*** *este aceea că ele trebuie să fie scrise toate folosind valori* **statice locale**. Aceasta înseamnă că ***valorile variantelor trebuie*** *să* ***fie determinate*** *pe* ***durata fazei*** *de* ***analiză a procesului de proiectare***. Toate exemplele de mai **sus** satisfac această condiție. Pentru a da un exemplu de instrucțiune *case* care nu îndeplineşte această cerință, să presupunem că avem o variabilă întreagă N, declarată prin:

```
variable N : integer := 1;
```

Dacă scriem instrucțiunea **case** sub forma

```
case expresie is    -- exemplu de instrucțiune case ilegală
  when N | N+l => . . .
  when N+2 TO N+5 => . . .
  when others => ...
  end case;
```

valorile variantelor depind de valoarea variabilei N. *Pentru că aceasta se poate schimba pe parcursul execuției, aceste variante nu sunt local statice*. Aşadar, instrucțiunea **case** este scrisă incorect Pe de altă parte, dacă am fi declarat C ca fiind o constantă întreagă, de exemplu cu ajutorul declarației:

```
constant C : integer := 1;
```

atunci am putea să scriem o instrucțiune *case* legală:

```
case expresie is
  when C | C+l => . . .
  when C+2 to C+5 => . . .
  when others => . . .
```

```
end case;
```

Aceasta este legală pentru că putem determina, analizând modelul, că prima alternativă include variantele 1 şi 2, a doua alternativă include numerele dintre 3 şi 6 iar a treia acoperă toate celelalte valori posibile ale expresiei.

Exemplele anterioare prezintă toate numai o instrucțiune în fiecare alternativă. Ca şi la instrucțiunea *if*, putem scrie un număr arbitrar de instrucțiuni secvențiale de orice tip în fiecare alternativă. Aceasta presupune scrierea instrucțiunilor *case* incluse, a instrucțiunilor *if* sau a oricărei forme de instrucțiuni secvențiale în oricare alternativă.

Deşi regulile precedente care guvernează scrierea instrucțiunilor *case* pot părea complexe, în practică există numai câteva *lucruri care trebuie reținute*:

- ***toate valorile posibile ale expresiei selectoare trebuie să fie acoperite numai de o singură variantă;***
- ***valorile din variante trebuie să fie local statice;***
- ***dacă se foloseşte `others`, acest cuvânt cheie trebuie să fie ultima alternativă şi trebuie să fie unica variantă în acea alternativă.***

Exemplu

Putem scrie un model comportamental pentru un multiplexor cu o intrare de selecție **`sel;`** multiplexorul are patru intrări de date `d0, dl, d2, d3` şi o ieşire de date `z.` Intrările şi ieşirile de date sunt de tipul **`IEEE standard_logic,`** iar intrarea de selecție este de tipul **`sel_range,`** pe care îl presupunem declara prin:

```
type sel_range is range 0 to 3;
```

Declarația de entitate, care defineşte porturile şi corpul arhitectural comportamental sunt prezentate în Figura 4.2. Corpul arhitectural conține numai o declarație de *proces*. Pentru că ieşirea multiplexorului trebuie să se schimbe dacă oricare din datele de intrare sau intrarea de selecție se modifică, procesul trebuie să fie sensibil la toate intrările; toate intrările sunt prezente în lista de sensibilitate a procesului. Am făcut apel la o instrucțiune *case* pentru a selecta intrarea de date care trebuie asignată ieşirii.

```
library ieee;
use ieee.std_logic_1164.all;
entity mux4 is
port ( sel : in sel_range;
         d0, d1,d2, d3: in std_ulogic;
         z : out std_ulogic );
end entity mux4; -
architecture demo of mux4 is
begin
out_select : process (sel, d0, d1, d2,d3) is
begin
```

```
case sel is

when 0 => z <= d0;

when 1 => z <= d1;

when 2 -> z <= d2;

when 3 => z <= d3;

end case;

end process out_select;

end architecture demo;
```

Fig. 2 Declarația de entitate şi un corp arhitectural pentru un multiplexor cu patru intrări de date.

4.1.2 Instrucțiunea WAIT

Următorul pas în modelarea comportamentului unui circuit este *specificarea momentelor **în** care procesul răspunde la schimbările valorilor semnalelor*. Acest lucru se realizează folosind instrucțiuni *wait*. O instrucțiune *wait* este o instrucțiune secvențială cu următoarele reguli de sintaxă:

instructiune_wait <=
[eticheta :] wait [on nume_semnal {,…}**]**
[until expresie_booleana **]**
[for expresie_time **];**

Eticheta opțională este folosită pentru a identifica instrucțiunea. *Scopul acestei instrucțiuni este acela de a determina procesul care execută instrucțiunea să-şi suspende execuția*. Clauza de senzitivitate, clauza de condiție şi aceea de *timeout* arată când procesul îşi reia execuția. Putem include orice combinație a acestor clauze, sau le putem omite pe toate trei.

Clauza de senzitivitate, care începe cu cuvântul cheie **on,** ne ajută să specificăm o listă de semnale la care procesul răspunde. *Dacă includem numai o listă de senzitivitate într-o instrucțiune **wait,** procesul se va relua ori de câte ori unul din semnale îşi schimbă valoarea, adică ori de câte **ori** apare un eveniment în oricare dintre semnale*. Această variantă a instrucțiunii *wait* este utilă într-un proces care modelează un *bloc de tip combinațional,* pentru **că** orice schimbare în semnalele de intrare poate conduce la modificarea ieşirilor. De exemplu,

```
half_add : process is begin
sum <= a xor b after T_pd;
carry <= a and b after T_pd;
wait on a,b; --aşteaptă modificările lui a sau b
```

```
end process half_add;
```

Procesul îşi începe execuţia prin generarea valorilor pentru *sum* şi *carry* pe baza valorilor iniţiale ale intrărilor *a* şi *b*, apoi se suspendă până când *a* sau *b* (sau ambele) schimbă valoarea. Atunci când aceasta se întâmplă, procesul se reia de la început.

Această formă de proces este folosită la modelarea sistemelor digitale, iar VHDL oferă o notaţie prescurtată pe care am folosit-o în exemple din capitolele anterioare.

Un proces cu o listă de senzitivitate este echivalent cu un proces care conţine o instrucţiune *wait* ***la sfârşit, instrucţiune care conţine clauza de senzitivitate care prezintă semnalele din lista senzitivitate.***

Aşadar, procesul half_adder de mai sus poate fi rescris sub forma:

```
half_add : process  (a,b)  is
begin
sum <= a xor b after T_pd;
carry <= a and b after T_pd;
end process half_add;
```

Instrucţiunea următoare:

```
wait on X, Y until Z = 0 for 100 ns;
```

suspendă execuţia procesului pentru maxim 100 ns. Dacă apare un eveniment în `X` sau înainte de 100 ns, condiţia "`Z = 0`" este evaluată. Dacă "`Z = 0`" este TRUE când apare evenimentul în `X` sau `Y`, procesul se reia în acel moment, în caz contrar, se continuă suspendarea. Procesul se reia după ce au trecut 100 ns sau când apare un eveniment în `X` sau Y cu `Z = 0`, oricare se întâmplă primul.

Exemplu

Ne întoarcem la modelul unui MUX cu două intrări din Figura 2. Procesul din acel model este sensibil la toate cele trei semnale de intrare. Aceasta înseamnă că el se va relua **la** schimbările de la oricare intrare de date, chiar dacă numai una dintre acestea este selectată la a moment dat. Dacă suntem îngrijoraţi de această mică lipsă de eficienţă în simulare, putem scrie un alt proces, folosind instrucţiuni *wait* pentru a oferi mai multă rigurozitate referitor la semnalele la care procesul este sensibil după ce a fost suspendat. Modelul modificat este prezentat în Figura 3. În acest model, atunci când este selectată intrarea *a*, procesul aşteaptă numai schimbări în intrarea de selecţie şi în intrarea *a*. Orice schimbare în b este ignorată. Dacă *b* este selectată, procesul aşteaptă schimbări în *sel* sau *b*, ignorând schimbările în *a*.

Clauza de condiţie dintr-o instrucţiune *wait*, începe prin cuvântul cheie *until* şi **ne** permite să specificăm o condiţie care trebuie să fie adevărată pentru ca procesul să se reia. In exemplu, instrucţiunea wait:

```
wait until clk = '1'
```

determină suspendarea execuției procesului până când valoarea semnalului *clk* devine '1'. Expresia care reprezintă condiția este testată pe durata suspendării procesului pentru a determina când se reia procesul suspendat In consecință, chiar atunci când condiția este adevărată când instrucțiunea *wait* este executată, procesul va fi totuşi suspendat până că semnalele necesare se schimbă şi determină evaluarea condiției din nou la adevărat. *Dacă instrucțiunea wait nu include o clauză de senzitivitate, condiția se testează ori de câte apare un eveniment în oricare din semnalele menționate în condiție.*

```
entity mux2 is
port ( a, b, sel : in bit; z:out
bit );
end entity mux2;

architecture behavioral of mux2 is
 constant prop_delay : time := 2 ns;
begin
 s_mux : process is begin
  case sel is
  when '0' => z<=a after prop_delay;
   wait on sel, a;
when '1'=> z <= b after prop_delay;
    wait on sel, b;
end case;
end process s_mux;
end architecture behavioral;
```

Fig. 3 Entitatea şi corpul arhitectural pentru un multiplexor **2:1.**

Exemplu

În Figura 4 este descris un proces de generare a unui semnal de ceas, utilizând instrucțiunea *wait.*.

```
entity gene is
end entity gene;
architecture EXA of gene is
 constant T_pw : time := 10 ns;
 signal elk : bit;
 begin
 clock_gen : process is
 begin
    clk <= '1' after T_pw,  '0' after 2*T_pw;
        wait until clk = '0';
end process clock_gen;
end architecture EXA;
```

Fig. 4 Un proces care generează un semnal de ceas.

De fiecare dată când procesul execută instrucțiunea `wait, clk` are valoarea '0'. Totuşi, procesul rămâne suspendat iar condiția se testează, de fiecare dată când există un eveniment în `clk`. Atunci când `clk` se schimbă la '1', nu se întâmplă nimic, dar când se schimbă din nou la '0', procesul se reia şi programează tranzacții pentru următorul ciclu.

Dacă o instrucțiune wait include o clauză de senzitivitate şi o clauză de condiție, condiția este testată numai dacă apare un eveniment în oricare din semnalele din clauza de senzitivitate.

De exemplu, dacă procesul se suspendă la următoarea instrucțiune wait:

```
wait on clk until reset = ' 0';
```

condiția se testează la fiecare schimbare de valoare a semnalului *clk*, fără a face referire la schimbările în *reset*.

Clauza de timeout *dintr-o instrucțiune* ***wait****, care începe cu cuvântul cheie for, ne permite să specificăm un interval maxim al timpului de simulare pentru care procesul ar trebui suspendat*.

Dacă includem şi o clauză de senzitivitate sau de condiție, acestea pot determina reluarea procesului mai târziu. De exemplu, instrucțiunea wait:

```
wait until trigger = '1'  for 1 ms;
```

determină suspendarea execuției procesului până când trigger se schimbă la ,'1', sau până când s-a scurs 1 ms din timpul de simulare, oricare ar apărea mai întâi. Dacă includem numai o clauză de timeout, procesul se suspendă pentru timpul indicat.

Exemplu

Putem rescrie generatorul de ceas, de data aceasta folosind o instrucțiune wait cu clauză de timeout, ca în Figura 5. În acest caz specificăm perioada ceasului ca timeout, după care procesul trebuie să se reia.

```
entity gen_mod is
end entity gen_mod;
architecture test of genjnod is
  constant Tjpw : time := 10 ns;
  signal clk : bit;
  begin
```

```
clockjgen : process is
begin
clk <= '1' after T_pw, '0' after 2*T_pw;
            wait for 2*T_pw;
   end process clock_gen;
 end architecture test;
```

Fig. 5 O a treia versiune a unui generator de ceas.

Dacă ne referim din nou la regula de sintaxă pentru *wait*, menționăm că este permisă scrierea

wait;

Această formă determină suspendarea procesului pentru timpul de simulare rămas, deşi aceasta poate părea ciudat, în practică se foloseşte destul de des . *Unul din cazurile în care se foloseşte este în procesele al căror scop este generarea stimulilor pentru o simulare*. Un astfel de proces ar trebui să genereze o secvență de tranzacții în semnalele conectate cu alte părți ale modelului şi apoi să se oprească. De exemplu, procesul:

```
test_gen: process is begin
  test0 <= '0' after 10 ns, '1' after 20 ns,
          '1' after 30 ns, '1' after 40 ns;
   test1 <= '0' after 10 ns, '1' after 30 ns;
                            wait;
end process test_gen;
```

generează toate cele patru combinații posibile de valori pentru semnalele test0 şi test1.

Dacă instrucțiunea wait din final ar fi fost omisă, procesul ar fi ciclat în permanență, repetând instrucțiunile de asignare de semnale fără suspendare, ***iar simularea nu ar fi progresat.***

4.1.3 Instrucțiuni IF

În multe modele, comportarea depinde de un set de condiții care pot fi îndeplinite sau nu pe durata simulării. Putem folosi instrucțiuni if pentru a exprima această comportare. Regula de sintaxă pentru o instrucțiune ***if*** este:

```
Instrucțiune_if <=
[eticheta_if:]if expresie_booleană then
   {instrucțiune_secvențială} {
  elsif expresie_booleană then
   {instrucțiune_secvențială} }
  [ else
   {instrucțiune_secvențială} ]
```

```
end if [eticheta_if];
```

La prima vedere, această definiție pare complicată, aşa încât vom începe prin câteva exemple mai simple. Eticheta poate fi folosită pentru a identifica instrucțiunea *if*. Un prim exemplu simplu este:

```
if en = '1'    then
   stored_value := data_in;
end if ;
```

Expresia de după cuvântul cheie *if* reprezintă condiția care se foloseşte pentru a controla dacă instrucțiunea de după cuvântul cheie *then* se execută sau nu. Dacă această condiție se evaluează la TRUE, instrucțiunea se execută în acest exemplu, dacă valoarea obiectului *en* este '1', se realizează asignarea.

Putem specifica şi acțiunile care au loc atunci când condiția este falsă. De exemplu:

```
if sel = 0   then
  result <= input_0;   -- se executa daca sel = 0
else
  result <= input_1;  -- se executa daca sel /= 0
end if ;
```

Aici, după cum arată comentariile, prima instrucțiune de asignare de semnale se execută în cazul în care condiția este TRUE; a doua instructiune de asignare se execută în cazul în care condiția este falsă.

În multe modele, este nevoie să se verifice un număr de condiții diferite şi să se execute o secvență diferită de instrucțiuni pentru fiecare caz. Putem construi o formă mai elaborată a instrucțiunii ***if*** pentru a realiza acest lucru, de exemplu:

```
if mode = immediate   then
  operand := immed_operand;
elsif opcode = load or opcode = add
        or opcode = subtract then
  operand   :=   memory_operand;
else
  operand := address_operand;
end if ;
```

În acest exemplu, se evaluează prima condiție şi, dacă este adevărată, se execută instrucțiunea de după primul cuvânt cheie *then*. Dacă prima condiție este falsă, se evaluează a doua condiție, şi dacă aceasta este TRUE, se execută instrucțiunea de după al doilea cuvânt cheie *then*. Dacă a doua condiție este falsă, se execută instrucțiunea de după cuvântul cheie *else*.

În general, putem construi o instrucțiune *if* cu oricâte cazuri *elsif* (inclusiv nici unul), şi putem include sau omite clauza *else*.

*Execuția instrucțiunii **if** începe cu evaluarea primei condiții. Dacă este falsă, sunt evaluate condițiile succesive, în ordine, până când se găseşte una adevărată. In acest caz se execută instrucțiunile corespunzătoare. Dacă nici una dintre condiții nu este adevărată, şi am inclus o clauză **else**, se execută instrucțiunile de după cuvântul cheie **else**.*

*Nu suntem limitați la o singură instrucțiune în orice secțiune a instrucțiunii **if**.* Acest lucru este ilustrat de următorul exemplu:

```
if opcode = halt_opcode    then
  PC := effective_address;
  executing := false;
  halt_indicator <= true;
end if ;
```

Dacă este adevărată condiția, se execută toate cele trei instrucțiuni, una după celaltă. Pe de altă parte, dacă se evaluează condiția la fals, nu se execută nici una din instrucțiuni.

Mai mult, *fiecare instrucțiune conținută într-o instrucțiune **if** poate fi orice instrucțiune secvențială. Aceasta însemna că putem* **include** *instrucțiuni if unele în altele* **(nested** *if)*. de exemplu:

```
if phase = wash   then
        if cycle_select = delicate_cycle
        then agitator_speed <= slow;
        else
        agitator_speed <= fast;
        end if;
    agitator_on <= true;
end if ;
```

În acest exemplu, se evaluează mai întâi condiția phase = wash; dacă este TRUE, se execută instrucțiunea *if* inclusă şi următoarea instrucțiune de asignare de semnal. Aşadar, asignarea *agitator_speed* <= *slow* se execută numai dacă ambele condiții sunt evaluate la TRUE, iar asignarea *agitator_speed* <= *fast* se execută numai dacă prima condiție este adevărată şi a doua condiție este falsă.

Exemplu

Vom dezvolta un model comportamental pentru un încălzitor cu termostat. Dispozitivul poate fi modelat sub forma unei entități cu două intrări întregi**,** una care specifică temperatura dorită iar celaltă care este conectată la un termometru, şi o ieşire de tip boolean, care comută încălzitorul ON sau OFF. Termostatul comută încălzitorul ON dacă temperatura măsurată este la două grade sub temperatura dorită, şi comută OFF dacă temperatura măsurată creşte cu mai

mult de două grade peste temperatura dorită. În Figura 6 sunt prezentate entitatea şi corpul arhitectural pentru acest termostat.

Declarația de entitate defineşte porturile de intrare şi pe cele de ieşire. Pentru **că** acesta este un model comportamental, corpul arhitectural conține numai o instrucțiune **de** proces **care** implementează comportarea cerută. Declarația de proces include o listă de sensibilitate, după cuvântul cheie *process*. Aceasta este o listă de semnale la care procesul este sensibil. Atunci când oricare din semnalele din listă îşi schimbă valoarea, procesul se reia şi se execută instrucțiunile secvențiale. După ce a fost executată ultima instrucțiune, procesul se suspendă din nou. în acest exemplu, procesul este sensibil la oricare dintre porturile de intrare. Deci, dacă ajustăm temperatura dorită sau dacă temperatura măsurată variază, procesul se reia. Procesul conține o instrucțiune *if* care compară temperatura curentă cu temperatura dorită. Dacă temperatura curentă este prea mică, procesul execută prima asignare de semnal şi comută încălzitorul ON. Dacă temperatura curentă este prea mare, procesul execută a doua asignare pentru a comuta încălzitorul OFF. Dacă temperatura curentă este în gama dorită, sarea încălzitorului nu se schimbă, pentru că nu există clauza *else* în instrucțiunea *if*.

```
entity thermostat is
   port ( desired_temp, actual_temp : in integer;
          heater_on : out boolean );
   end entity thermostat;
architecture example of thermostat is
begin
 controller : process (desired_tenp, actual_temp) is
 begin
   if actual_temp < desired_temp - 2 then
      heater_on <= true;
    elsif actual_temp > desired_temp + 2 then
      heater_on <= false;
    end if ;
 end process controller;
end architecture example;
```

Fig. 6 Entitatea şi corpul arhitectural pentru un termostat

Curs 5

Instrucțiunile limbajului VHDL

1. Instrucțiuni secvențiale II

2. Declarații de arhitectură şi instrucțiuni concurente

1. Instrucțiuni secvențiale II

Reamintim din cursul anterior că ***instrucțiunile secvențiale*** se folosesc în procese şi subprograme pentru a descrie algoritmi. Ele se execută secvențial, cu excepția cazurilor când se întâlnesc instrucțiuni de salt., în mod similar cu instrucțiunile oricărui alt limbaj procedural (cum este de exemplu C). În continuare vor fi tratate următoarele instrucțiuni secvențiale.

1. **Instrucțiunea LOOP**
2. **Instrucțiunea EXIT**
3. **Instrucțiunea NEXT**
4. **Instrucțiunea WHILE**
5. **Instrucțiunea FOR**
6. **Instrucțiunea NULL**
7. **Instrucțiunea ASSERT şi REPORT**
8. **Instrucțiunea PROCESS**

1. Instrucțiunea LOOP

Adesea, avem nevioe să scriem o secvență de instrucțiuni care este executată în mod treptat. în aceste cazuri se folosesc instrucțiuni *loop*. În VHDL există mai multe forme diferite de instrucțiuni *loop*; cea mai simplă repetă o secvență de instrucțiuni în mod indefînit; se mai numeşte ***buclă infinită.*** Regula de sintaxă pentru o astfel de buclă este:

instrucțiuneajoop <= [
[eticheta_loop:]

loop
{instrucțiune_secvențială}
end loop [eticheta_loop]

În cele mai multe limbaje de programare, nu este de dorit o buclă infinită, pentru că înseamnă că programul nu se termină niciodată. Totuşi, atunci când modelăm sisteme digitale, o buclă infinită poate fi utilă, pentru că multe dispozitive hardware efectuează în mod repetat aceeaşi funcție până când se întrerupe tensiunea de alimentare. In mod tipic, un model pentru un astfel de sistem include o instrucțiune *loop* în corpul unui proces; bucla, în schimb, conține o instrucțiune *wait*.

Exemplu

În Figura 1 se prezintă un model pentru un numărător care începe să-şi incrementeze conținutul, de la zero, pe fiecare tranziție a semnalului de ceas din '0' în '1'. Atunci când numărătorul atinge 15, se întoarce la zero la următoarea tranziție a semnalului de ceas. Corpul arhitectural pentru acest numărător conține un proces care inițializează mai întâi ieşirea count la zero, apoi aşteaptă în mod repetat o tranziție a semnalului de ceas pentru a-şi incrementa valoarea.

```
entity counter is
 port ( clk : in bit;
 count : out natural );
end entity counter;
architecture behavior of counter is
begin
 incrementer : process is
  variable count_value : natural := 0;
 begin
  count <= count_value;
  loop
      wait until clk = '1' ;
      count_value := (count_value + 1) mod 16;
      count <= count_value;
   end loop;
 end process incrementer;
end architecture behavior;
```

Fig. 1 Declarația de entitate şi corpul arhitectural pentru un numărător.

Instrucțiunea *wait* din acest exemplu determină suspendarea procesului în mijlocul buclei. Atunci când semnalul de ceas se schimbă din '0' în '1', procesul se reia şi actualizează

valoarea numărătorului şi ieşirea *count*. Bucla este apoi repetată începând cu instrucțiunea *wait*, astfel că procesul se suspendă din nou.

Un alt lucru care trebuie menționat este faptul că instrucțiunea ***process*** *nu include o listă de sensibilitate. Aceasta se întâmplă pentru că el contine o instrucțiune* ***wait****.* Un proces poate conține fie o listă de sensibilitate, fie o instrucțiune *wait*, dar nu poate conține ambele instrucțiuni.

2. Instrucțiunea EXIT

În exemplul anterior, se executau în mod repetat instrucțiunile din buclă, fără nici o posibilitate de oprire. În mod uzual, avem nevoie de o ieşire dintr-o buclă atunci când sunt împlinite anumite condiții. Putem folosi instrucțiunea exit pentru a părăsi o buclă de program. Regula de sintaxă este:

istrucțiunea_exit <=
[eticheta_exit:] exit [eticheta_loop] [when expresie_booleană];

Eticheta opțională la începutul instrucțiunii *exit* serveşte la identificarea instrucțiunii. Cea mai simplă formă a instrucțiunii *exit* este:

```
EXIT;
```

Atunci când se execută această instrucțiune, toate instrucțiunile care au mai rămas în buclă sunt suspendate, iar controlul se transferă instrucțiunii de după cuvintele cheie *end loop*. Astfel, într-o buclă putem scrie:

```
IF condiție THEN
    EXIT;
END IF;
```

unde ***condiție*** este o expresie booleană. Pentru că aceasta este, poate, cea mai comună utilizare intsrucțiunii *exit*, VHDL furnizează o modalitate mai scurtă pentru a o indica, folosind cuvântul cheie *when*. Folosim o instrucțiune *exit* împreună cu *when* într-o buclă de forma:

```
LOOP
    ...
    EXIT WHEN condiție;
END LOOP;
...      -- controlul se transferă aici
      --când condiție devine adevărată în interiorul buclei
```

Exemplu

Vom modifica modelul anterior al numărătorului astfel încât să includem o intrare *reset* care, atunci când este ’1’, resetează la zero ieşirea *count*. Ieşirea stă la zero atâta timp cât *reset* este '1'; numărătoarea se reia la următorul front activ al semnalului de ceas după ce *reset* s-a schimbat la '0'. Declarația modificată de entitate din Figura 2 include acest nou port de intrare. Corpul arhitectural este modificat prin includerea unei bucle în alta şi adăugarea semnalului reset la instrucțiunea *wait* inițială. Bucla interioară realizează aceeaşi funcție ca şi mai înainte, cu excepția faptului că atunci când *reset* se schimbă la '1', procesul se reia, iar instrucțiunea *exit* determină terminarea buclei interioare. Controlul se transferă instrucțiunii de după sfârşitul buclei interne.

```
entity counter is
  port ( elk, reset : in bit;   count : out natural );
 end entity counter;

architecture behavior of counter is

begin
  incrementer : process is
    variable count_value : natural := 0;
    begin
    count <= count_value;
    loop
        loop
         wait until elk = '1' or reset = '1';
         exit when reset = '1';
         count_value := (count_value + 1) mod 16;
         count <= count_value;
        end loop;
-- in acest punct, reset = '1'
     count_value := 0;
     count <= count_value;
     wait until reset = ' 0' ;
    end loop;
  end process incrementer;
end architecture behavior;
```

Fig. 2 Declarația de entitate şi un corp arhitectural pentru un numărător cu intrare de reset

După cum arată comentariile, ştim că acest punct poate fi atins numai când *reset* este '1'. Valoarea numărătorului şi ieşirea *count* sunt resetate, iar procesul aşteaptă ca semnalul *reset* să revină la '0'. In timp ce este suspendat în acest punct, orice schimbări în semnalul de intrare de ceas sunt ignorate. Când *reset* se schimbă la '0' procesul se reia, iar bucla exterioară se repetă.

Acest exemplu ilustrează de asemenea un alt lucru inportant. Atunci când avem bucle *loop* incluse, cu o instrucțiune *exit* în bucla interioară, instrucțiunea *exit* determină transferul controlului afară numai din bucla interioară, nu în afara buclei exterioare. În mod implicit, o instrucțiune *exit* transferă controlul la bucla imediat circumscrisă.

În unele cazuri, dorim transferul controlului în afara buclei interioare, dar şi în afara celei exterioare. Putem realiza acest lucru prin etichetarea buclei exterioare şi folosind această etichetă în instrucțiunea *exit***.** Putem scrie:

nume_buclă : LOOP
…
    EXIT nume_buclă;
...
END LOOP. nume_buclă;

În această secvență am etichetat bucla cu numele *numebuclă*, astfel încât putem indica ce buclă să părăsim în instrucțiunea *exit***.** Eticheta buclei poate fi orice identificator valid.

Instrucțiunea *exit* care se referă la această etichetă se poate afla în interiorul instrucțiunilor *loop* incluse.

Pentru a arăta modul în care buclele pot fi incluse, etichetate şi părăsite, vom considera următoarele instrucțiuni:

```
outer : LOOP
…
  inner : LOOP
  …
    EXIT outer WHEN condiție-1; -- exit 1
    ...
    EXIT WHEN condiție-2;       -- exit 2
    ...
  END LOOP inner;
  ...          -- target A
  EXIT outer WHEN condiție-3;      -- exit 3
  …
  END LOOP outer;
   ...         -- target B
```

Această secvență conține două instrucțiuni *loop***,** una etichetată *inner* şi aflată în interiorul celeilalte care este etichetată *outer***.** Prima instrucțiune *exit***,** marcată prin comentariul *exitl***,** transferă controlul instrucțiunii etichetată *target B***,** dacă este adevărată condiția proprie. A doua instrucțiune *exit,* marcată prin comentariul *exit2*, transferă controlul la *target A***.** Pentru că

nu se face referire la o etichetă, există numai instrucțiunea *loop* imediat circumscrisă, adică bucla *inner***.** In sfârşit, instrucțiunea *exit* marcată *exit 3* transferă controlul la *target B.*

3. Instrucțiunea NEXT

Un alt tip de instrucțiune pe care putem să o folosim pentru a controla execuția buclelor este instrucțiunea **next.** Atunci când se execută această instrucțiune, iterația curentă a buclei este încheiată fără a se mai executa alte instrucțiuni, şi începe iterația următoare.

Regula de sintaxă este:

Instructiune_next <=
[eticheta_next:] next [eticheta_loop] [when expresie_booleana];

Eticheta opțională de la începutul instrucțiunii *next* serveşte la identificarea instrucțiunii. Instrucțiunea este foarte asemănătoare cu instrucțiunea *exit***,** diferența fiind dată de cuvântul cheie *next* în loc de *exit***.** Cea mai simplă formă este:

```
NEXT ;
```

care declanşează iterația următoare din bucla imediat circumscrisă. Putem să introducem de asemenea o condiție de testat înainte de sfârşitul iterației:

```
NEXT WHEN condiție;
```

şi putem include o etichetă loop pentru a indica pentru care *loop* se încheie iterația.

NEXT eticheta_loop;
sau
NEXT eticheta_loop WHEN condiție;

O instructiune *next* care detennină sfârşitul buclei imediat circumscrise poate fi rescrisă sub forma unei bucle echivalente care include o instrucțiune *if*. De exemplu, următoarele două secvențe sunt echivalente:

loop instructiune-1; **next when** condiție; instructiune-2; **end loop;**	**loop** instructiune-1; **if** not condiție **then** instructiune-2; **end if;** **end loop;**

4 Instrucțiunea WHILE

Putem complica instrucțiunea *loop* de bază introdusă anterior pentru a forma o buclă *while***,** care testează o condiție înainte de fiecare iterație. In cazul în care condiția este adevărată, iterația se execută. Dacă este falsă, bucla se încheie. Regula de sintaxă pentru o buclă *while* este:

Instructiune_loop <=
[eticheta_loop:]
while condiție loop
{instructiuni_secventiale}
end loop [eticheta_loop};

Singura diferență dintre această formă şi forma de bază *loop* este aceea că am adăugat cuvântul cheie *while* şi o condiție înainte de cuvântul cheie *loop***.** Toate observațiile pe care le-am făcut în legătură cu bulca de bază *loop* se aplică şi buclei *while***.** Putem scrie orice instrucțiuni secvențiale în corpul instrucțiunii, inclusiv *exit* şi *next***,** şi putem eticheta bucla înainte de cuvântul cheie *while***.**

Există trei situații importante care trebuie amintite în legătură cu buclele *while***.** Prima: condiția este testată înainte de fiecare iterație a buclei, inclusiv prima iterație. Aceasta înseamnă că dacă această condiție este falsă înainte să începem bucla, ea se terrnină imediat, fără a fi executată nici o itera^e. De exemplu, dacă se consideră bucla *while***:**

```
while index > 0 loop
   .... --instructiune_A:realizează o operație asupra
  indexului
end loop;
 ....        -- instructiune_B
```

dacă putem demonstra că *index* nu este mai mare decât zero înainte de începerea buclei atunci ştim că instrucțiunile din interiorul buclei nu se vor executa, iar controlul se va tranfera direct la *intructiune_B***.**

A doua: în absența instrucțiunilor *exit* în interiorul unei bucle *loop***,** bucla se încheie numai atunci când condiția devine falsă. Aşadar, ştim că negația condiției trebuie să se îndeplinească atunci când controlul este transferat la instrucțiunea imediat următoare buclei. În mod similar, în absența instrucțiunilor *next* în interiorul unei bulce *loop***,** bucla realizează o iterație numai când condiția este adevărată. Aşadar, condiția este îndeplinită dacă se porneşte o instrucțiune din corpul buclei. în exemplul de mai sus, ştim că *index* trebuie să fie mai mare decât zero atunci când se execută *instrucțiune_A,* şi *index* trebuie să fie mai mic sau egal decât zero când se

execută *instructiune_B.* Acest lucru ne ajută să judecăm corectitudinea modelului pe care îl scriem.

A treia: atunci când scriem instrucțiunile din corpul unei bucle *while***,** trebuie să fim siguri că condiția va deveni în cele din urmă falsă, sau ca o instrucțiune *exit* va opri bucla în cele din urmă. In caz contrar, bucla *while* nu se va sfârşi niciodată.

Exemplu

Se poate dezvolta un model pentru o entitate **cos** care poate fi folosit ca parte a unui sistem special de procesare a semnalelor. Entitatea are o intrare, *theta***,** care este un număr real care reprezintă un unghi în radiani, şi o ieşire, *result***,** care reprezintă funcția cosinus a valorii lui *theta***.** Putem folosi relația:

$$\cos\theta = 1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - \frac{\theta^6}{6!} + \ldots$$

Astfel încât să adăugăm termeni succesivi ai seriei până când termenii devin mai mici decât o milionome din rezultat. Declarațiile entitatății şi ale corpului arhitectural sunt prezentate în figura 3.

```
entity cos is
  port ( theta : in real;   result : out real );
end entity cos;
architecture series of cos is
begin
  sumare : process (theta) is
    variable sum, term : real;
    variable n : natural; begin
    sum := 1.0;
    term := 1.0;
    n := 0;
    while abs term > abs (sum / 1.0E6) loop
     n := n + 2;
     term := (-term) * theta**2 / real(((n-1) * n));
     sum := sum + term;
    end loop;
   result <= sum;
 end process sumare;
end architecture series;
```

Fig. 3 O entitate şi un corp arhitectural pentru un modul cos

Corpul arhitectural constă dintr-un proces care este sensibil la schimbările din semnalului de intrare theta. Inițial, variabilele *sum* şi *term* sunt setate la 1.0, ceea ce reprezintă primul termen din serie. Variabila *n* începe cu valoarea 0 pentru primul termen. Funcția *cosinus* se calculează cu ajutorul unei bulce *while* care incrementează pe *n* cu doi şi îl foloseşte pentru a calcula următorul termen pe baza termenului anterior. Iterația are loc atâta timp cât ultimul termen calculat este mai mare (în amplitudine) decât o milionime din sumă. Când ultimul termen este mai mic decât acest prag, bucla *while* se încheie. Putem aprecia că bucla se va termina, pentru că valorile termenilor succesivi ai seriei devin din ce în ce mai mici. Aceasta se întâmplă pentru că funcția factorială creşte cu o rată mai mare decât funcția exponențială.

5. Instrucțiunea FOR

O altă cale prin care putem dezvolta bucla de bază este prin folosirea instrucțiunii for. O astfel de buclă include o specificație care arată de câte ori trebuie să se execute corpul buclei. Regula de sintaxă este:

Instructiune_loop <=
[eticheta_loop:]
  for identificator **in** gama_discreta **loop**
    {instructiune_secventiala}
end loop [eticheta_loop];

Am văzut în capitolele anterioare că o gamă discretă poate fi de forma:

expresie_simpla (**to**|**downto**) expresie_simpla

care reprezintă toate valorile situate între limita din stânga şi cea din dreapta inclusiv.. Identificatorul se numeşte ***parametrul buclei*** şi, pentru fiecare iterație, ia valori succesive din gama discretă, începând cu elementul cel mai din stânga. De exemplu, în următoarea buclă for:

```
for count_value in 0 to 127 loop
  count_out <= count_value;
  wait for 5 ns;
end loop;
```

identificatorul count_value ia valorile 0, 1, 2 ş.a.m.d., şi pentru fiecare valoare se execută instrucțiunea de asignare şi instrucțiunea *wait*. Aşadar, semnalului *count_out* i se vor asigna valorile 0,1,2,... până la 127, la intervale de 5 ns.

Am văzut de asemenea că o gamp discretă poate fi specificată folosind un nume de tip sau subtip discret. De exemplu, dacă avem tipul enumerare:

```
type controller_state is (inițial, idle, active, error);
```

putem scrie o buclă *for* care iterează peste fiecare valoare de acest tip:

```
for state in controller_state loop
...
end loop;
```

În interiorul secvenței de instrucțiuni din corpul buclei, parametrul buclei este o constantă al cărui tip este tipul de bază din gama discretă. Aceasta înseamnă că putem folosi valoarea lui indicându-1 într-o expresie, dar nu îi putem face asignări. Spre deosebire de alte constante, nu avem nevoie să îl declarăm. Parametrul buclei este definit implicit pe bucla respectivă. El există numai atâta timp cât se execută bucla; nu există nici înainte şi nici după execuția ei. De exemplu, următoarea declarație de proces arată cum NU se foloseşte parametrul buclei:

```
eronat : process is
  variable i, j : integer;
  begin
  i := loop_param;    -- eroare!!
  for loop_param in 1 to 10 loop
    loop_param := 5;    -- eroare! !
   end loop;
  j := loop_param;    -- eroare!!
end process eronat;
```

Asignările pentru *i* şi *j* sunt ilegale pentru că parametrul buclei nu este definit nici înainte nici după buclă. Asignarea din interiorul buclei este ilegală pentru că *loop_param* este o constantă şi nu poate fi modificat.

O consecință a modului în care se defineşte parametrul buclei este aceea că el ascunde orice obiect cu acelaşi nume definit în afara buclei. De exemplu, în procesul:

```
exemplu_ascunde: process is
  variable a, b : integer;
  begin
  a:= 10;
  for a in 0 to 7 loop
    b:=a;
  end loop;
  -- a=10 si b=7
  ...
```

```
end process exemplu_ascunde;
```

variabilei *a* i se asignează inițial valoarea 10, apoi se execută bucla, formând un parametru al buclei numit de asemenea *a.* În interiorul buclei, asignarea pentru *b* foloseşte parametrul buclei, astfel încât valoarea finală a lui *b* după ultima iterație este 7. După buclă, parametrul buclei nu mai există, astfel că dacă folosim numele *a,* ne referim de fapt la variabila a cărei valoare este, încă, 10.

După cum am precizat mai înainte, bucla f o r iterează asupra parametrului buclei, atribuind acestuia valori succesive din gama discretă, începând cu valoarea cea mai din stânga. O mențiune importantă este legată de cazul în care specificăm o gamă nulă. În acest caz, corpul buclei nu se execută de loc. O gamă nulă, vidă, poate apărea dacă specificăm o gamă crescătoare cu limita din stânga mai mare decât limita din dreapta, sau o gamă descrescătoare cu limita din stânga mai mică decât cea din dreapta. De exemplu, pentru bucla:

```
for i in 10 to 1 loop
...
end loop;
```

se sfârşeşte imediat, fără a executa instrucțiunile incluse. Dacă dorim ca bucla să se execute cu *i* luând valorile 10,9, 8,ar trebui să scriem:

```
for i in 10 downto 1 loop

end loop;
```

O mențiune finală legată de buclele for este aceea că, la fel ca la instrucțiunile loop de bază, ele pot include instrucțiuni secvențiale arbitrare, inclusiv next, exit, şi putem folosi etichete pentru buclă înaintea cuvântului cheie for.

Exemplu

Vom rescrie modelul cosinus din Figura 3 astfel încât să calculăm rezultatul prin sumarea primilor 10 termeni ai seriei. Declarația de entitate este neschimbată. Corpul arhitectural modificat este prezentat în Figura 4. Acesta constă dintr-un proces care foloseşte o buclă *for* şi nu *while*. Ca şi mai înainte, variabilele *sum* şi *term* sunt setate la 1.0, care reprezintă primul termen al seriei. Variabila *n* este înlocuită cu parametrul buclei *for*. Bucla iterează de 9 ori, calculând restul de nouă termeni ai seriei.

```
architecture
serie_lungime_fixă of cos is
begin
 sumare    :    process
```

```
      (theta)  is  variable
      sum, term : real;
      begin
        sum := 1.0;
        term := 1.0;
        for n in 1 to 9 loop
    term := (-term) * theta**2 / real( ( (2*n-l) *
    2*n));
     sum := sum + term;
    end loop;
     result <= sum;
    end process sumare;
    end architecture serie_lungime_fixă;
```

Fig. 4 Un corp arhitectural modificat pentru modulul cos.

6. Instrucțiunea NULL

Uneori, atunci când scriem modele, avem nevoie să stabilim că, dacă sunt îndeplinite anumite condiții, nu se realizează nici o acțiune. Este nevoie de aceasta atunci când folosim instrucțiuni *case*, pentru că trebuie să includem o alternativă pentru fiecare valoare posibilă a expresiei selectoare. în loc să lăsăm necompletată partea aferentă a instrucțiunii, putem folosi o *instrucțiune nuli* pentru a indica în mod explicit că nu trebuie să se efectueze nimic. Sintaxa pentru instrucțiunea *null* este:

instrucțiune_null <= [etichetă :] null;

Eticheta opțională servește la identificarea instrucțiunii. O instrucțiune *null* simplă, neetichetată, este:

```
NULL;
```

Un exemplu de folosire a acesteia într-o instrucțiune *case* este:

```
case opcode is
 when add =>
   Acc := Acc + operand;
 when subtract =>
   Acc : = Acc - operand;\
 when nop => null;
 end case;
```

Putem folosi o instrucțiune *null* în orice loc în care este nevoie de o instrucțiune secvențială, nu numai în cazul unei alternative *case*. O instrucțiune *null* se poate folosi pe durata fazei de dezvoltare din scrierea modelului. Dacă ştim, de exemplu, că vom avea nevoie de o entitate ca parte a unui sistem, dar nu suntem încă pregătiți să scriem un model detaliat pentru aceasta, putem scrie un model comportamental care nu face nimic. Un astfel de model include numai un proces cu o instrucțiune *null* în corpul său:

```
secțiune_de_control : process (lista_de_sensibilitate) is
     begin
     null;
end process secțiune_de_control;
```

Menționăm că acest proces trebuie să conțină lista de sensibilitate, din rațiuni care vor fi explicate în paragraful 4.2.

7. Instrucțiunile ASSERT şi REPORT

Unul din motivele pentru care dezvoltăm modele pe un sistem de calcul este acela de a verifica, înainte de realizarea fizică, faptul că un proiect funcționează corect. Putem testa parțial un model aplicând semnale de test la intrare şi verificând că ieşirile au valorile pe care le aşteptăm. În caz contrar, trebuie să găsim cauza pentru care proiectul a funcționat greşit. Această sarcină poate fi simplificată folosind instrucțiuni *assert* care verifică faptul că anumite condiții sunt îndeplinite în model. O instrucțiune *assert* este o instrucțiune secvențială, deci poate fi inclusă oriunde în corpul unui proces. Regula de sintaxă este următoarea:

instrucțiune_assert <=
[etichetă:] assert expresie_booleana
[report expresie] [severity expresie];

Eticheta opțională ne ajută să identificăm instrucțiunea. Cea mai simplă formă include numai cuvântul cheie *assert* urmat de expresia condiție la care ne aşteptăm să fie TRUE atunci când se execută instrucțiunea *assert*. Dacă această condiție nu este îndeplinită, adică dacă a apărut o violare a condiției *assert*, simulatorul raportează acest fapt Pe durata sintezei, condiția dintr-o instrucțiune *assert* poate fi interpretată ca o condiție pe care sintetizatorul poate să o presupună TRUE. Pe durata verificării formale, condiția poate fi interpretată ca o condiție care trebuie verificată. De exemplu, dacă scriem:

```
assert initial_value <= max_value;
```

iar *initial_value* este mai mare decât *max_value* atunci când instrucțiunea se execută pe durata simulării, simulatorul ne va informa despre acest lucru. Pe durata sintezei,sintetizatorul poate

presupune că *initial_value* <= *max_value* şi va optimiza circuitul pe baza acestei informații. Pe durata verificării formale, se poate încerca demonstrarea *initial_value* <= *max_value* pentru toți stimulii posibili şi pentru toate cazurile care conduc la instrucțiunea *assert*.

Dacă avem un număr de instrucțiuni *assert* în model, este util să ştim care anume instrucțiune a fost violata. Putem determina simulatorul să furnizeze informație suplimentară dacă includem o clauză report în instrucțiunea *assert*. De exemplu,

```
assert   initial_value   <=   max_value
  report "initial_va1ue_too_large";
```

Şirul pe care îl indicăm este folosit pentru a forma un mesaj. Putem scrie orice expresie în clauza report cu condiția ca aceasta să conducă la un şir. De exemplu,

```
assert current_character >= '0' and current_character <= '9'
  report "Input_number" & input_string & "contains a non-
  digit";
```

În acest caz, mesajul este obținut prin *concatenarea* a trei valori şir împreună. Tipul enumerare predefinit severity_level poate avea următoarele stări:

```
type severity_level is (note, warning, error, failure);
```

Putem include o valoare din acest tip într-o clauză severity a unei instrucțiuni *assert*. Această valoare va indica gradul în care violarea instrucțiunii *assert* afectează funcționarea modelului. Valoarea *note* poate fi folosită pentru a transmite mesaje informative de la simulator, de exemplu,

```
assert free_memory >= low_water_limit
  report "low on memory, about to start garbage
  collect" severity note;
```

Nivelul de severitate *warning* poate fi folosit dacă apare o situație neobişnuită în care modelul poate continua să se execute dar, poate produce rezultate neaşteptate. De exemplu,

```
assert packet_length /= 0
  report "empty network packet received"
  severity warning;
```

Putem folosi nivelul de severitate *error* pentru a indica faptul că ceva a funcționat greşit cu siguranță şi că se impune o acțiune corectivă. De exemplu,

```
assert  clock_pulse_width  >=  min_clock_width
  severity error;
```

În sfârşit, *failure* se poate folosi atunci când se detectează o inconsistență care nu ar trebuie să apară niciodată. De exemplu,

```
assert (last_position - first position + 1) =
  number_of_entries report "inconsistency in buffer model"
  severity failure;
```

Dacă sunt prezente ambele clauze, *report* şi *severity*, regula de sintaxă ne arată faptul că *report* trebuie să fie considerată prima. Dacă omitem *report*, şirul implicit este "Assertion violation". Dacă omitem **severity,** valoarea implicită este **error.** Valoarea **severity** este folosită de simulator pentru a determina dacă execuția se continuă sau nu după o violare de **assert.** Cele mai multe simulatoare permit utilizatorului să specifice un prag de severitate după care execuția se opreşte.

Exemplu

Un flip-flop S/R are două intrări, **S** şi **R,** şi o ieşire Q. Dacă **S** este ’1’**,** ieşirea este setată la **’1’;** dacă **R** este '1', ieşirea este resetată la '0'. Există o restricție: **S** şi **R** nu pot fi ambele '1' simultan. Dacă se întâmplă acest lucru, valoarea de ieşire nu este specificată. In Figura 5 am prezentat un model comportamental pentru un flip-flop SR, model care include o verificare a acestei condiții ilegale.

Corpul arhitectural conține un proces sensibil la intrările **S** şi **R.** În interiorul procesului am scris o instrucțiune *assert* care cere ca **S** şi **R** să nu fie ambele T. Dacă ambele sunt ‘1’, **assert** este violată şi simulatorul va scrie un mesaj "Assertion violation" cu **severity** având valoarea **error.** Dacă execuția se continuă după această etapă, se va asigna mai întâi valoarea '1' ieşirii Q, urmată de valoarea '0'. Valoarea care se obține este '0'. Acest lucru este permis pentru că starea lui Q nu a fost specificată pentru această condiție ilegală, aşadar avem libertatea să alegem orice valoare. Dacă instrucțiunea **assert** nu este violată, atunci se **execută** cel mult una din instrucțiunile următoare, modelând astfel corect comportarea **bistabilului** SR.

```
entity SR_flipflop is
  port ( S, R : in bit;    Q : out bit
 );
end entity SR_flipflop;
architecture checking of SR_flipflop
 is
begin
   set__reset : process (S, R) is
   begin
     assert S = '1' nand R = '1';
     if S = '1' then
       Q <= '1';
      end if;
```

```
            if R = '1' then
          Q <= '0';
          end if;
          end process set_reset;
          end architecture checking;
```

Fig.5 O entitate şi un corp arhitectural pentru un flip-flop set/reset, care include o verificare a condițiilor corecte

Exemplu

În Figura 6 se prezintă modelul pentru o entitate care are trei intrări întregi, a, b, c, şi produce o ieşire întreagă, z, care este egală cu valoarea cea mai mare dintre cele trei intrări.

Corpul arhitectural este dezvoltat folosind un proces care conține instrucțiuni if incluse. Am introdus o eroare "accidentală" în model. Dacă simulăm acest model şi aplicăm valorile a= 7, b = 3 şi c = 9 la porturile de intrare ale entității, ne aşteptăm ca valoarea lui result, şi deci la portul de ieşire, să fie 9. Instrucțiunea assert arată că valoarea lui result trebuie să fie mai mare sau egală decât toate intrările. Totuşi, codul nostru de eroare determină ca valoarea 7 să fie asignată lui result, astfel că instrucțiunea assert este violată. Această violare ne determină să examinăm mai atent modelul şi să-1 corectăm.

```
entity max3 is
 port ( a, b, c : in integer;   z : out integer
);  end entity max3;
architecture check_error of max3 is
begin
   maximizer : process (a, b, c)
   variable result : integer;
   begin
    if a > b then if a > c
     then result := a;
    else
       result := a; -- Atenție!   Ar trebui result := c;
     end if;
    elsif  b > c then
     result := b;
    else
     result := c;
    end if;
   assert result >= a and result >= b and result >= c
report "inconsistent result for maximum" severity failure;
z <= result; end process maximizer;
end architecture check_error;
```

Fig. 6 O entitate şi un corp arhitectural pentru un circuit care selectează valoarea maximă

VHDL ne oferă şi o instrucțiune report, care este destul de asemănătoare cu assert. Regula de sintaxă este:

instrucțiune_report <=
[etichetă:] report expresie [severity expresie];

Diferența este dată de faptul că de această dată nu există o condiție de verificat şi că dacă nu se specifică un nivel de severitate, acesta este note în mod implicit. Instrucțiunea report poate fi gândită ca o instrucțiune assert în care condiția este valoarea FALSE iar severity este note, aşa încât se va produce mereu un mesaj. O modalitate de folosire a instrucțiunii report este pentru includerea etapelor parcurse într-un model, ca ajutor în depanare.

Să presupunem că scriem un model complex şi nu suntem siguri că am gândit modelul prea bine. Putem folosi o instrucțiune report pentru a determina procesul să scrie mesaje, astfel incât să putem vedea când acestea sunt activate şi ce realizează. Un exemplu este:

```
transmit_element: process  (transmit_data)  is
  ... -- declarații de variabile
  begin
  report  "transmit_element:  data  ="  &
    data_type'image(transmit_data);
  ...
end process transmit_element;
```

2. Declarații de arhitectură şi instrucțiuni concurente

2.1 Intrucțiunea PROCESS

Instrucțiunile process pe care le-am folosit ca exemple suntt instrucțiuni concurente. Ele sunt tipul fundamental de instrucțiuni folosit în arhitecturi. De fapt, toate celelalte instrucțiuni concurente pot fi scrise sub forma unei instrucțiuni process echivalente. Instrucțiunile *process* au două forme. *Una dintre ele* conține o listă de sensibilitate. Aceasta este forma cel mai des întâlnită în modelele de până acum:

```
eticheta: process  (lista_de_sensibilitate)
   — declarații de constante
```

```
   — declarații de variabile
   — declarații de subprograme
   — declarațiile de semnale nu sunt permise aici
   — alte elemente, mai specializate, pot fi declarate aici
   begin
   — instrucțiuni secvențiale
   end process eticheta;
```

In general, orice instrucțiune de proces din interiorul unei arhitecturi se execută o dată la începutul simulării, apoi se execută numai când un semnal din lista de sensibilitate își schimbă valoarea (adică, dacă există un eveniment în unul sau mai multe semnale din lista de sensibilitate). Constantele și variabilele pot fi declarate în procese; nu și semnalele. Instrucțiunile din interiorul proceselor trebuie să fie secvențiale. De fiecare dată când un proces este activat, se execută instrucțiunile secvențiale, cum este cazul în limbajele procedurale C, C++.

Variabilele declarate în interiorul unui proces sunt statice. Ele sunt inițializate o singură dată la începutul simulării și își mențin valorile între două activări succesive ale procesului.

Cealaltă formă a instrucțiunii de proces nu are listă de sensibilitate :

```
eticheta: process
   — declarații de constante
   — declarații de variabile
   — declarații de subprograme
   — declarațiile de semnale nu sunt permise aici
   — alte elemente, mai specializate, pot fi declarate aici
   begin
   — instrucțiuni secvențiale
   end process eticheta;
```

Acest proces se execută o dată la începutul simulării și continuă să se execute până când se întâlnește o instrucțiune *wait*. Când s-a executat ultima instrucțiune secvențială, procesul se reia automat cu prima instrucțiune secvențială. Deci, cel puțin una din instrucțiunile secvențiale trebuie să fie *wait* pentru a preveni o buclă infinită. O instrucțiune *process* cu listă de sensibilitate este echivalent cu un proces fără listă de sensibilitate care are un *wait* ca ultimă instrucțiune din proces. De exemplu, procesul:

```
s_list: process  (SI,  S2)
   — declarații de constante
   — declarații de variabile begin
   — instrucțiuni secvențiale
end process s_list;
```

este echivalent cu procesul

```
no_list: process
   — declarații de constante— aceleaşi ca în s_list
   — declarații de variabile— aceleaşi ca în s_list
   begin
   — instrucțiuni secvențiale —aceleaşi ca în s_list
   wait on SI,  S2
end process no_list;
```

presupunând că toate declarațiile şi instrucțiunile secvențiale sunt aceleaşi, cu excepția ultimei instrucțiuni *wait*. In ambele cazuri, procesul se execută o dată la începutul simulării, apoi, ori de câte ori apare un eveniment în semnalele SI sau S2.

Este clar că un proces fără listă de sensibilitate trebuie să aibă un *wait*, iar un proces cu listă de sensibilitate poate să nu aibă un *wait***.** Prima cerință previne buclele infinite, a doua este necesară pentru a preveni conflictele în activarea proceselor.

2.2 Instrucțiuni ASSERT concurente

Instrucțiunea **assert** poate fi secvențială sau concurentă. Forma concurentă este:

eticheta: assert expresie_booleană report "şir_mesaj" severity severity_level;

Atunci când este un eveniment în oricare din semnalele din expresie_booleană, se evaluează expresie**_booleană.** Dacă expresia se evaluează la FALSE, se scrie **şir_mesaj** la dispozitivul standard de ieşire, **severity_level** este tipul predefinit enumerare din limbajul VHDL standard. Nivelul implicit este **warning.** Interpretarea lui **severity_level** este dependentă de implementare. De exemplu, anumite implementări încetează simularea dacă **severity_level** este prea mare. Iată un exemplu:

```
eticheta: assert (A or B)  = C
   report "warning: C is NOT equal to (A or B)"
    severity warning;
```

Se presupune că **A, B, C** sunt semnale de tipul **Bit.** în condiții normale de operare, **C** se presupune mereu egal cu **(A or B).** Dorim să detectăm orice violare a acestei condiții

Instrucțiunea concurentă *assert* este activată o dată la începutul simulării, apoi, de fiecare dată când există un eveniment în oricare din semnalele A, **B, C.** Când condiția este FALSE, adică dacă **(A or B)** NU este egal cu **C,** mesajul ***warning*** va fi generat la dispozitivul standard de ieşire. Ieşirea se poate redirecționa la un fişier.

Instrucțiunea **assert** de tip concurent este echivalentă cu următorul proces:

```
eticheta: process  (A,  B,  C)
begin
 assert  (A or B)  = C
report "warning: C is NOT equal to  (A or B)"
 severity warning; end process eticheta;
```

2.3 Asignări de semnal de tip concurent

Asignările de semnale pot fi secvențiale sau concurente. Dacă sunt secvențiale, ele sunt executate numai când algoritmul ajunge la respectivele instrucțiuni. Atunci când sunt concurente, ele sunt executate de fiecare dată când apare un eveniment în oricare din semnalele din partea dreaptă a asignării. De exemplu, instrucțiunea concurentă de asignare:

label: C <= A or B;

este activată o dată la începutul simulării, apoi, atunci când apare un eveniment în semnalul **A** sau în semnalul **B.** Această instrucțiune concurentă este echivalentă cu următorul proces:

```
label: process  (A, B)
 begin
    C <= A or B;
end process label;
```

Instrucțiunile concurente de asignare de semnale pot fi şi conditionale. Formatul pentru acestea este:

```
labei: signal_name <= [transport]
  forma_de_undal when condiție1 else
  forma_de_unda2 when condiție2 else
…
 forma_de_undaN when condițieN else
  forma_de_undaQ;
```

Instrucțiunea concurentă este activată o dată la începutul simulării, apoi, dacă apare un eveniment în oricare din semnalele din oricare forma de unda sau din oricare semnal din condiții. Când condiție1 este TRUE, se asignează forma_de_undă1 lui signal_name**.** Când condiție1 este FALSE iar condiție2 este TRUE, se asignează forma_de_unda2 lui signal_name. Forma_de_undaQ se asignează lui signal_name numai când condiției**,** şi... şi condițieN sunt toate FALSE. Numai una din forme_de_unda se asignează semnalului, chiar dacă mai multe condiții sunt TRUE. Se asignează formade_unda care este condiționată de prima condiție care este TRUE. Iată un exemplu:

```
LL1:  S <= A or B when XX = 1 else
 A and B when XX = 2 else A xor B;
```

În acest exemplu, se presupune că **A, B** şi **S** sunt semnale de tip **Bit,** iar **XX** este un semnal de tip **Integer.** Instrucțiunea se execută o dată la începutul simulării, apoi se execută din nou ori de câte ori apare un eveniment în semnalele **A, B** sau **XX.** Această instrucțiune concurentă este echivalentă cu următoarea instrucțiune **process:**

```
LL1: process   (A,  B, XX)
      begin
   if XX = 1 then S <= A or B;
   elsif XX = 2 then S <= A and B;
   else S <= A xor B; end
 process LL1;
```

Formatul pentru o instrucțiune concurentă de asignare selectată este:

```
labei: with expresie select
    signal_name <= [transport]
  forma_de_undal when opțiuneal,
  forma_de_unda2 when opțiunea2,
    ...
 forma_de_undaN when opțiuneaN,
  forma_de_undaQ when others;
```

Trebuie să se includă oricare din valorile posibile pentru expresie în exact una din opțiuni Setul de opțiuni trebuie să fie mutual exclusiv, iar reuniunea tuturor seturilor de opțiuni trebuie să fie mulțimea tuturor valorilor posibile pentru expresie, **others** este cuvântul cheie care include toate valorile expresiei care nu au fost incluse în nici o opțiune. Instrucțiunea concurentă este activată o dată la începutul simulării, apoi, ori de câte ori apare un eveniment în oricare semnal din expresie sau în oricare semnal din oricare forma_de_unda**.** De exemplu, instrucțiunea:

```
LL2: with (S1 + S2) select
C <= A after 5 ns when 0,
    B after 10 ns when 1 to integer'high,
    D after 15 ns when others;
```

se activează la $t = 0$ şi ori de câte ori există un eveniment în oricare dintre semnalele: **S1, S2, A, B** sau **D.** Toate semnalele se presupun de tip **Integer.** Această instrucțiune de selecție este echivalentă cu următorul proces:

```
LL2: process  (S1,  S2, A,  B,  D)
```

```
begin
     case (S1 + S2)  is
  when 0 => C <= A after 5 ns;
   when 1 to Integer'high => C <= B after 10 ns;
   when others => C <= D after 15 ns; end case; end
   process LL2;
```

CURS 10

Etapele de proiectare a unui microprocesor pe 8-biți

1. Obiective

În această secțiune se va prezenta un exemplu de proiectare a unui microprocesor simplu cu un set minim de instrucțiuni şi cu o magistrală de date pe 8 biți. Pe tot parcursul acestei secțiuni denumirea prescurtată a microprocesorului va fi *μP8*. În continuare se vor prezenta caracteristicile microprocesorului *μP8*:

- magistrala de date pe 8-biți;
- magistrala de adrese pentru accesarea unei memorii RAM externe pe 15 biți;
- setul de instrucțiuni format din 22 instrucțiuni;
- modurile de adresare permise: imediat, direct, indirect şi indexat;
- 16 regiştrii interni pe 8-biți organizați în două bancuri;
- port de intrare/ieşire;
- acceptă o întrerupere externă.

2. Setul de instrucțiuni

În tabelul T.1 este prezentat setul de instrucțiuni al microprocesorului.

Tabelul T.1 Setul de instrucțiuni

Nr.	Mnemonică	Operație	Descriere
1.	load Rd	ACC<-Rd	Încarcă ACC (acumulatorul) cu, conținutul reg. Rd, d este adresă în domeniul [0...7] (**adresare directă prin registre**).
2.	load (Rd)	ACC<-RAM[Rd]	Încarcă ACC cu, conținutul unei locații din DATE-RAM(zona de date a memoriei. RAM) a cărei adresă se află în reg. Rd (**adresare indexată**).
3.	load #d	ACC<-d	Încarcă ACC cu valoarea d, unde d este o constantă în domeniul [0...255] (**adresare imediată**).
4.	load d	ACC<-RAM[d]	Încarcă ACC cu, conținutul unei locații din DATE- RAM a cărei adresă este d, unde d este o constantă în domeniul [0...255] (**adresare directă**).
5.	load (d)	ACC<-RAM[RAM[d]]	Încarcă ACC cu, conținutul unei locații din DATE- RAM a cărei adresă se află în locația cu adresa d din DATE-RAM, d este în domeniul [0...255] (**adresare indirectă**).
6.	store Rd	Rd<-ACC	Încarcă conținutul ACC în reg. Rd, d este adresă în domeniul

			[0...7] (adresare directă prin registre).
7.	store (Rd)	RAM[Rd]<-ACC	Încarcă ACC în DATE-RAM la adresa din Rd, d este adresă în domeniul [0...7] (adresare indexată).
8.	store d	RAM[d]<-ACC	Încarcă ACC în DATE-RAM la adresa d, d este în domeniul [0...255] (adresare directă).
9.	store (d)	RAM[RAM[Rd]]<-ACC	Încarcă ACC într-o locație din DATE-RAM a cărei adresă se află în altă locație din DATE-RAM a cărei adresă se află în reg. Rd, d este adresă în domeniul [0...7] (adresare indirectă).
10	in Rd	Rd<-IN_PORT	Încarcă reg. Rd cu o valoare de la portul de intrare.
11	out	OUT_PORT<-ACC	Pune la portul de ieşire valoarea din ACC.
12	xor Rd	ACC<-ACC$Rd	Operație SAU-EXCLUSIV între conținutul registrului Rd şi ACC, d este adresă în domeniul [0...7].
13	add Rd	ACC<-ACC+Rd+C; C<-carry out	Adună conținutul reg. Rd, al flagului C şi al ACC, păstrează rezultatul în ACC. d este adresă în domeniul [0...7].
14	test Rd	Z<-ACC&Rd	Operație ŞI logic între conținutul reg. Rd şi ACC, rezultatul nu se va păstra în ACC. Flagul Z va fi setat dacă rezultatul operației este 0, altfel va fi şters. d este adresă în domeniul [0...7].
15	clear_c	C<-0	Setează flagul C la „0”
16	set_c	C<-1	Setează flagul C la „1”
17	jc#a	IF C=1->PC<-a ELSE PC<-PC+2	Dacă C=1 continuă cu instr. de la adresa a, dacă C=0 continuă cu instr. următoare, a este în domeniul [0...255].
18	jz #a	IF Z=1->PC<-a ELSE PC<-PC+2	Dacă Z=1 continuă cu instr. de la adresa a, dacă Z=0 continuă cu instr. următoare, a este în domeniul [0...255].
19	jump #a	PC<-a	Salt necondiționat la adresa a, a este în domeniul [0...255].
20	jsr #a	RAM[SP]<-PC+2 SP<-SP+1 PC<-a	Salvează adresa următoarei instr. în stivă şi incrementează indicatorul de stivă, apoi salt la adresa a, a este în domeniul [0...255].
21	ret	SP<-SP-1 PC<-RAM[SP]	Decrementează indicatorul de stivă (SP) şi încarcă PC cu adresa aflată la „vârful” stivei
22	reti	SP<-SP-1 ACC<-RAM[SP] SP<-SP-1 PC<-RAM[SP]	Decrementează SP şi încarcă ACC cu valoarea din „vârful” stivei, apoi decrementează din nou SP şi încarcă PC cu adresa instrucțiuni de la care s-a întrerupt programul la intrare în ISR.

Codificarea în binar a instrucțiunilor prezentate în tabelul T.1 este prezentată în tabelul T.2. Se poate observa că ultimii trei biți ai codului instrucțiunii sunt folosiți pentru a stoca adresa registrelor interne. Instrucțiunile care nu lucrează cu registrele nu vor folosi aceşti biți. Instrucțiunile se disting una de cealaltă prin combinațiile binare ale celor mai semnificativi

cinci biți. Unele instrucțiuni ocupă câte doi octeți datorită faptului că datele şi adresele sunt pe 8-biți.

Tabelul T.2 Codificarea setului de instrucțiuni

Mnemonică	**Nr. de octeți**	**Codificare în binar**
load Rd	1	00100d2d1d0
load (Rd)	1	00101d2d1d0
load #d	2	01000000d7d6d5d4d3d2d1d0
load d	2	00110000d7d6d5d4d3d2d1d0
load (d)	2	00111000d7d6d5d4d3d2d1d0
store Rd	1	00000d2d1d0
store (Rd)	1	00001d2d1d0
store d	2	00010000d7d6d5d4d3d2d1d0
store (d)	2	00011000d7d6d5d4d3d2d1d0
in Rd	1	01100d2d1d0
out	1	01101000
xor Rd	1	10000d2d1d0
add Rd	1	10001d2d1d0
test Rd	1	10010d2d1d0
clear_c	1	10100000
set_c	1	10101000
jc#a	2	11000000d7d6d5d4d3d2d1d0
jz #a	2	11001000d7d6d5d4d3d2d1d0
jump #a	2	11010000d7d6d5d4d3d2d1d0
jsr #a	2	11011000d7d6d5d4d3d2d1d0
ret	1	11100000
reti	1	11101000

3. Arhitectura microprocesorului

În Figura 1 este prezentată arhitectura microprocesorului *μP8*, în continuare se va studia şi se va proiecta fiecare bloc al acestei arhitecturi.

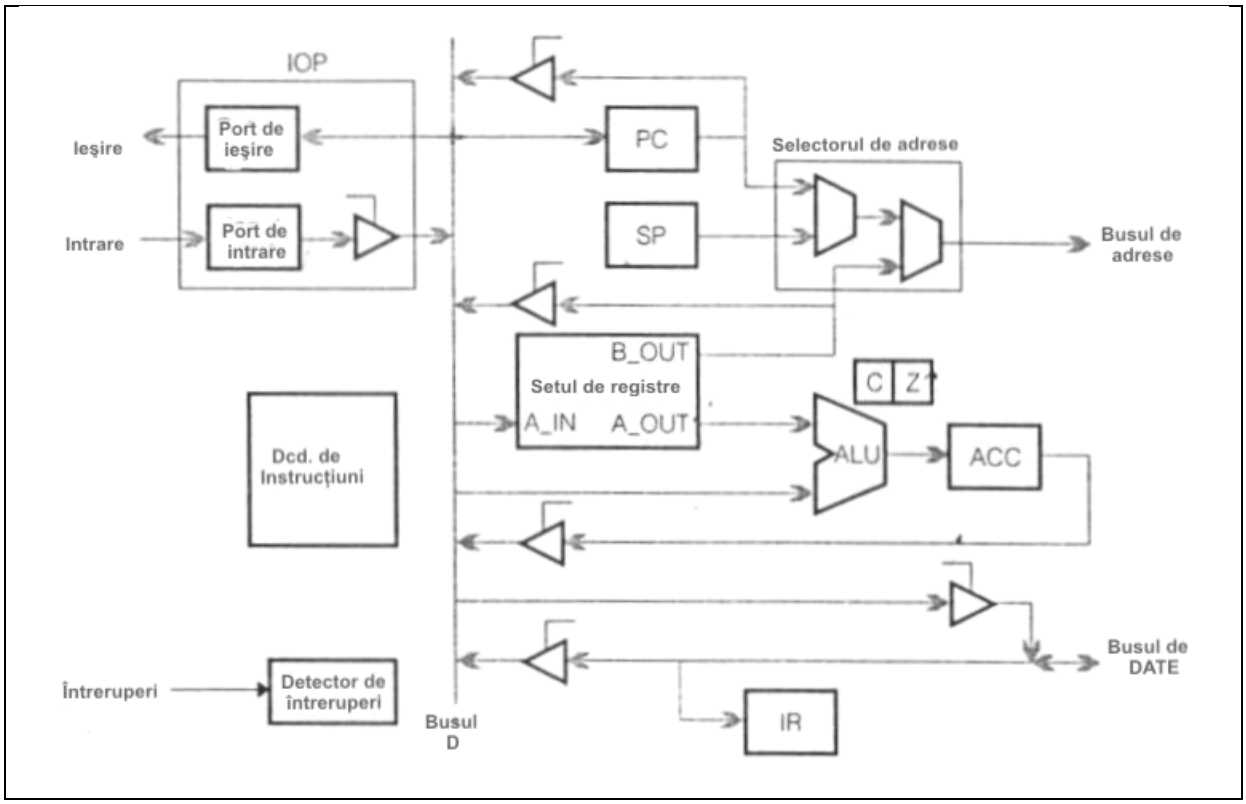


Figura 1 Arhitectura µP8

3.1. Magistrala internă de date D

Această magistrală conectează împreună toate componentele microprocesorului *µP8*. Pe această magistrală pot să fie puse date de către: PC, setul de registre prin intermediul ieşirii B, ACC, portul de intrare şi magistrala externă de date care conectează memoria externă la *µP8*. De asemenea valorile de pe magistrala internă D pot fi preluate de: PC, intrarea A_IN a setului de registre, ACC, portul de ieşire şi magistrala externă de date.

3.2. Setul de registre (R0-R7)

Microprocesorul *µP8* are 16 registre interne pentru stocarea biţilor de date. Cele 16 registre sunt organizate în două bancuri, unul este bancul normal de lucru, iar celălalt este accesat pe durata unei întreruperi. Setul de registre va fi proiectat ca şi o memorie dual-port, cu două ieşiri şi o intrare, toate pe 8-biţi. Acesta permite ca într-un singur ciclu de tact să fie citite două locaţii din memorie, iar una dintre ele să fie scrisă. Setul de registre dual port, notat *uP8_REG* (vezi Figura2) are nevoie de trei biţi de adrese pentru a adresa cele două bancuri a câte opt registre (intrările A_ADDR şi B_ADDR). Cei trei biţi de adrese provin de la cei trei biţi mai puţini semnificativi ai registrului de instrucţiuni (IR2...IR0). Cele două intrări de adrese A_ADDR şi B_ADDR activează registrele a căror conţinut va fi citit prin intermediul ieşirilor A_OUT şi B_OUT.

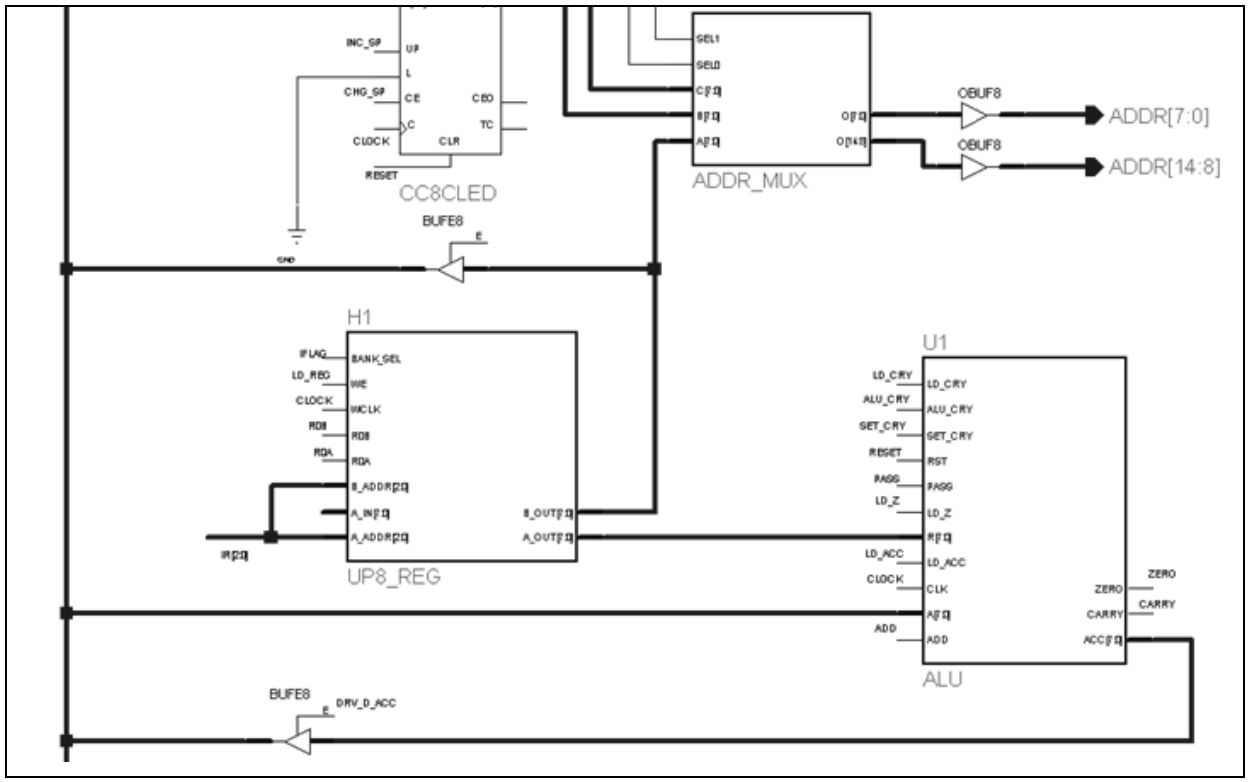


Figura2 Setul de registre şi ALU

De asemenea intrarea A_ADDR va selecta registrul care va fi activ pentru scrierea datelor prin intermediul intrării A_IN, date provenind de pe magistrala internă D. Valoarea prezentă la intrarea A_IN va fi scrisă la următorul front crescător de tact (registrele sunt proiectate ca şi memorie sincronă), dacă la intrarea LD_REG semnalul are nivel logic „1”. Pe durata executării programelor microprocesorul are adesea nevoie de un registru în care să stocheze valori imediate şi adrese, pentru acest scop este selectat registrul R0. Modalitatea de accesare a acestui registru de către decodorul de instrucțiuni este de a aplica un „1” logic la intrările R0A şi R0B, forțând astfel adresa din R0 la intrarea A_ADDR sau B_ADDR ne mai ținându-se astfel cont de conținutul lui IR.

Intrarea BANK_SEL permite activarea la un moment dat doar a unuia din cele două bancuri de registre. Semnalul de intrare IFLAG este activ pe „1” logic pe durata executării unei subrutine de tratare a întreruperilor şi este folosit pentru a comuta între cele două bancuri de registre. Astfel se oferă posibilitatea ca subrutina de tratare a întreruperilor să lucreze cu propriul set de registre, prevenindu-se prin aceasta alterarea valorilor din registrele folosite în mod curent de microprocesor. Valoarea de la ieşirea B_OUT poate fi transmisă pe busul intern de date D dacă este activat, de către decodorul de instrucțiuni şi bufferul tristate BUFE8 prin intermediul semnalului DRV_D_REG. Această operație este necesară când se doreşte transmiterea datelor de la setul de registre către celelalte componente ale microprocesorului.

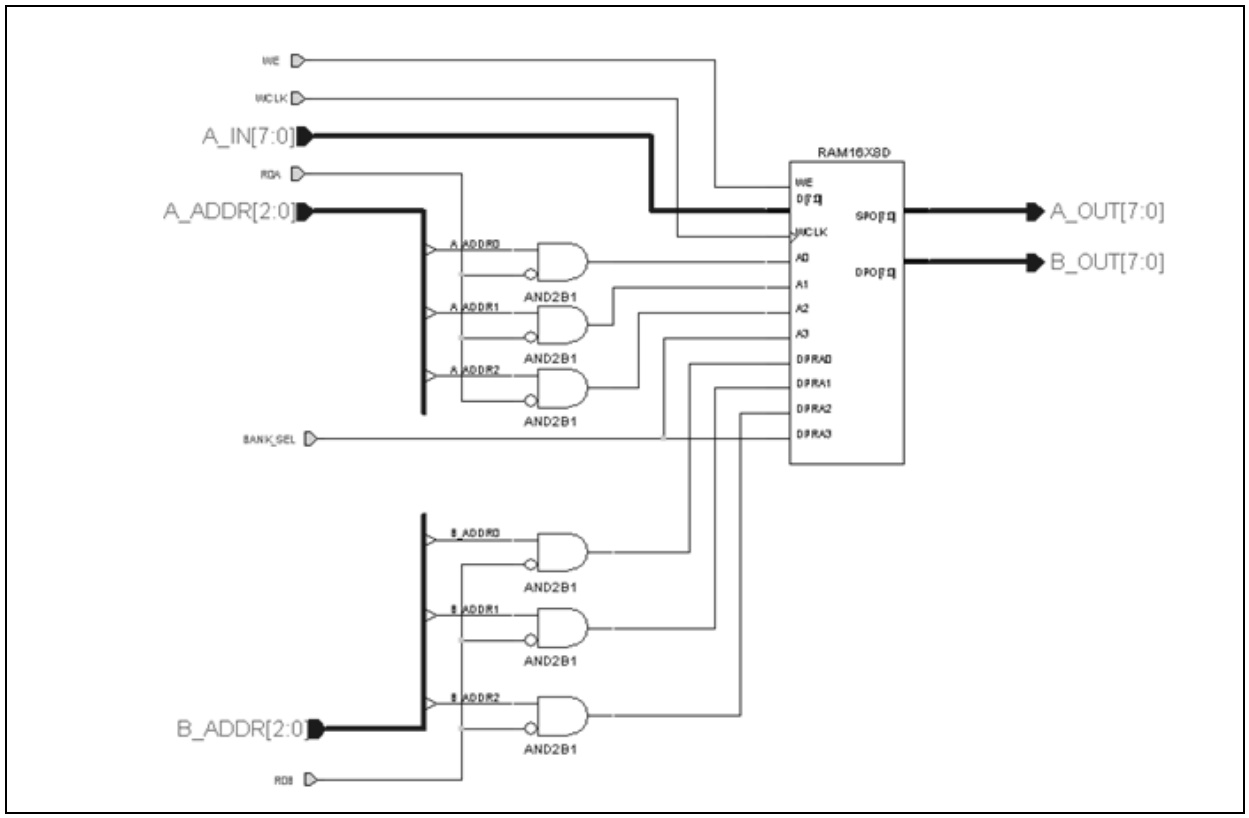


Figura3 Schema detailată a setului de registre

Detaliile de proiectare ale setului de registre sunt prezentate în Figura 3. Componenta principală a setului de registre este o memorie RAM sincronă dual port (RAM16X8D) selectată din lista de componente standard a mediului Xilinx Foundation. Bitul cel mai semnificativ al intrărilor de adrese pentru cele două bancuri de registre provine de la intrarea BANK_SEL. Pentru a forța adresele registrelor în zero se folosesc două seturi de porți logice ŞI controlate de semnalele R0A şi R0B.

3.3. Unitatea aritmetică şi logică, registrul acumulator şi flagurile de stare

Unitatea aritmetică şi logică (ALU) are rolul de a efectua toate operații logice (ŞI, SAU, SAU-Exclusiv etc.) şi aritmetice (adunare, scădere etc.) cu octeții de date. Rolul registrului acumulator (ACC) este acela de a stoca valorile intermediare rezultate în urma operațiilor făcute de ALU. Registrul ACC este asemeni unei locații de memorie şi poate stoca un singur octet la un moment dat. Flagul sau fanionul de stare este reprezentate fizic printr-o celulă de memorie care are posibilitatea să stocheze la un moment dat un singur bit. În cazul microprocesorului nostru există două astfel de flaguri de stare şi anume flagul de transport sau de carry (C) şi flagul de zero (Z). Flagul de carry stochează valoarea de la ieşirea de transport (carry) a lui ALU, iar flagul Z este setat sau nu în funcție de rezultatul operațiilor efectuate de ALU.

Modulul ALU din Figura 2 conține componentele necesare pentru a efectua operațiile de adunare şi sau-exclusiv. De asemenea în acest modul sunt incluse şi acumulatorul şi flagurile de stare. Operanzii (date sau adrese) pe 8-biți intră în ALU prin intermediul

magistralelor R7...R0 şi A7...A0. Operanzii pot să provină de la setul de registre sau de pe magistrala internă D. Intrările de ADD şi PASS controlează operațiile aritmetice care se efectuează cu operanzii mai sus amintiți (vezi tabelul T.3).

Tabelul T.3 Codificarea operațiilor aritmetice

PASS	ADD	Operație ALU
0	0	ACC<-ACC$REGA
0	1	ACC<-ACC + REGA + CARRY
1	X	ACC<- BUS-ul D

Detaliile în ceea ce priveşte structura ALU sunt prezentate în Figura 4.a. Prin intermediul intrărilor ADD şi PASS sunt controlate o pereche de multiplexoare (H2 şi H5) fiecare cu intrări pe 8-biți (vezi Figura 4.b). Aceste multiplexoare pot să aleagă între: octetul provenind de la intrarea A7...A0, ieşirea sumatorului ADD8 sau ieşirea din modulul XOR_8X8 care face operația SAU_EXCLUSIV bit-cu-bit (vezi Figura 4.c). Când intrarea LD_ACC are valoarea „1” logic şi concomitent are loc o tranziție din „0” în „1” a semnalului de tact, modulul acumulator ACCUM0 va stoca valoarea de la ieşirea multiplexorului H2. Flagurile Z şi C îşi vor reîmprospăta de asemenea valoarea pe durata unei tranziții pozitive a semnalului de clock cu condiția ca semnalele LD_Z şi LD_CRY să aibă valoarea „1” logic. Flagul Z este întotdeauna încărcat cu valoarea de la ieşirea modulului ZEROTEST care face operația logică SAU-NU între biții rezultați în urma unui ŞI logic bit-cu-bit (vezi Figura 5). Octeții prinşi în operația ŞI bit-cu-bit provin de la setul de registre şi de pe magistrala D. Dacă intrarea ALU_CRY are valoarea „1” logic, flagul de transport C va stoca valoarea de la ieşirea de transport a modulului sumator (ieşirea CO a modulului ADD8), vezi Figura 6. Acesta permite propagarea transportului în operațiile aritmetice multi-octet.

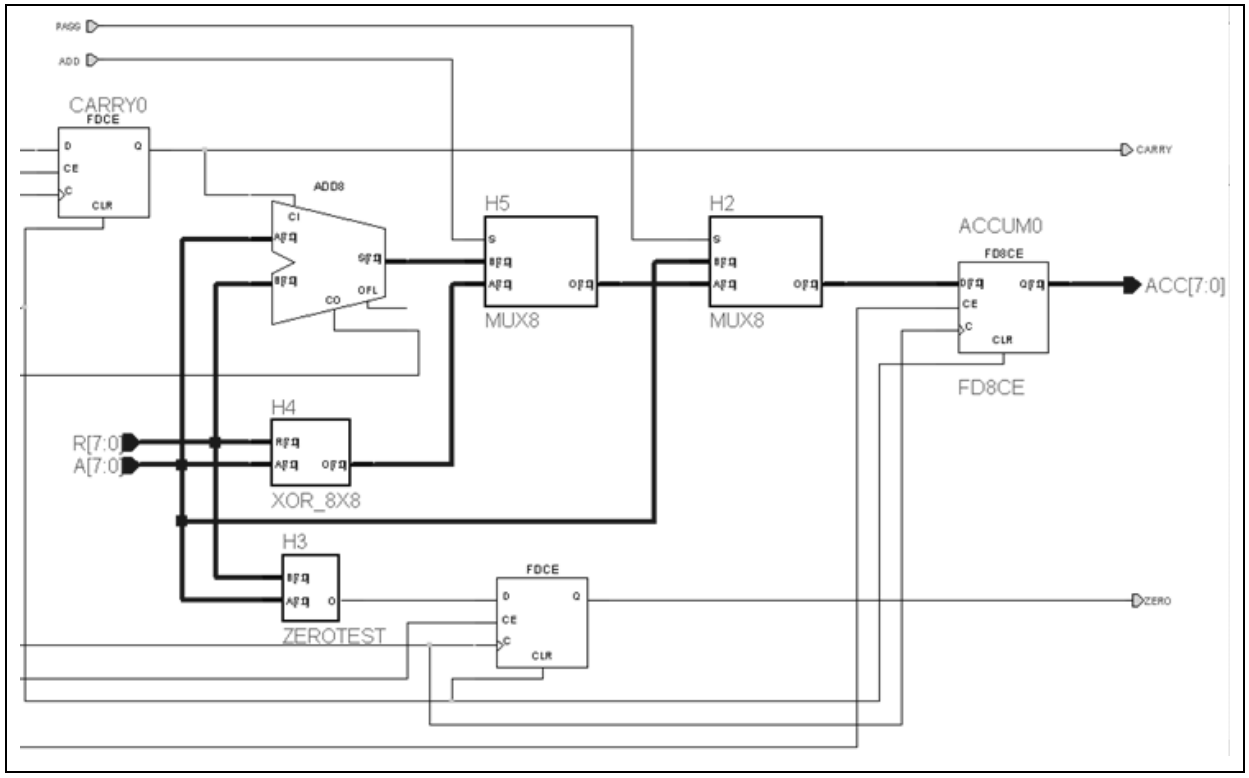


Figura 4.a Unitatea Logică şi Aritmetică

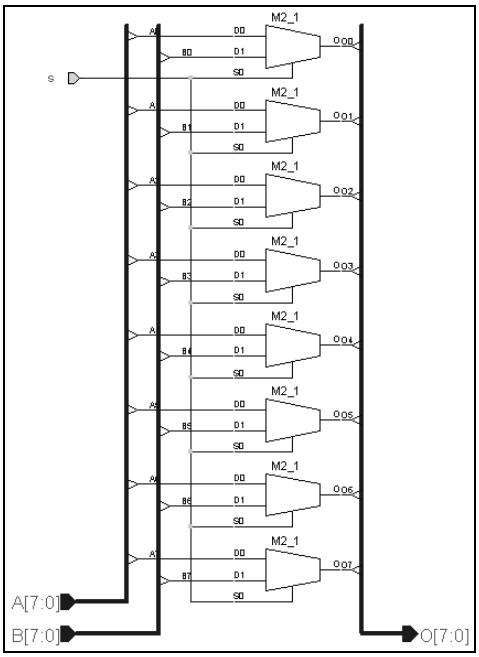


Figura 4.b Circuit detailat MUX8

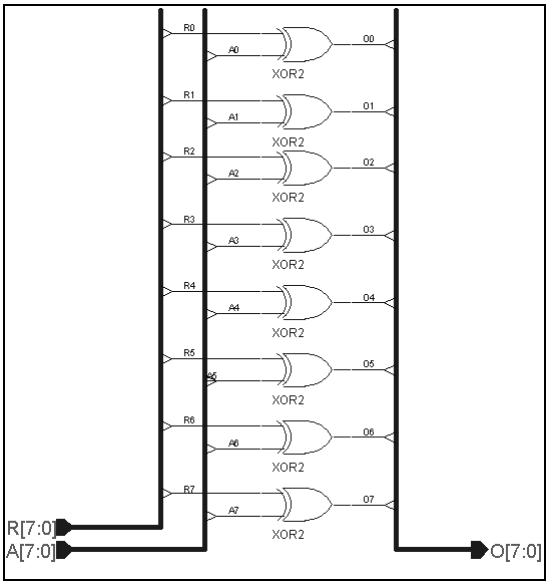


Figura 4.c Circuit detailat XOR 8x8

Dacă intrarea ALU_CRY = „0” flagul C va fi încărcat cu valoarea de la intrarea SET_CRY care îl va seta în „0” sau în „1” logic.

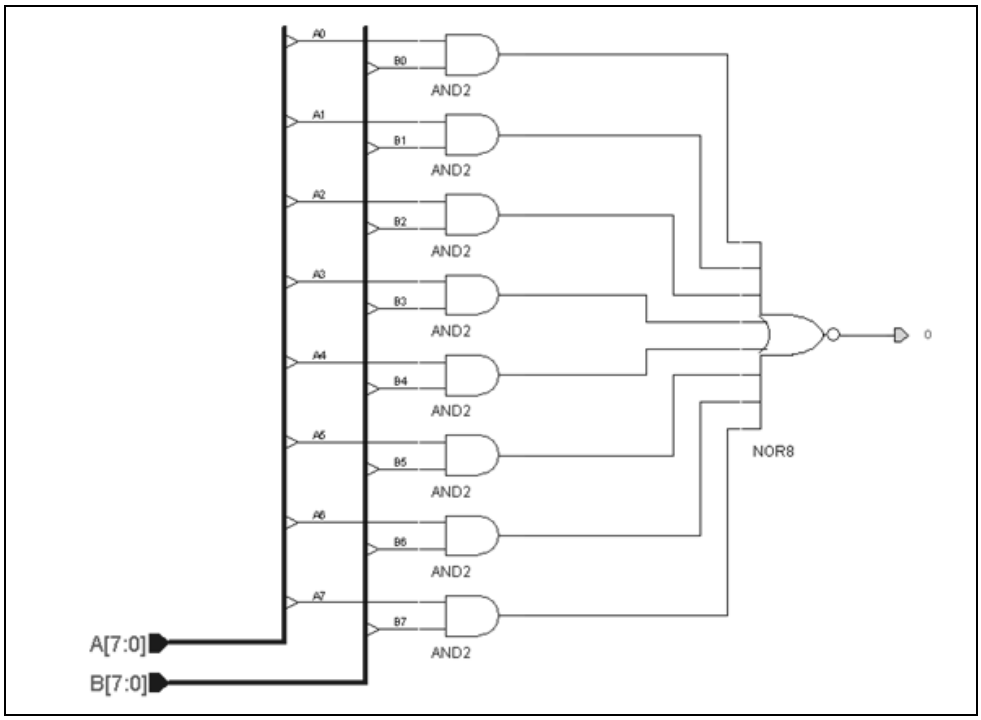


Figura 5 Modulul ZEROTEST

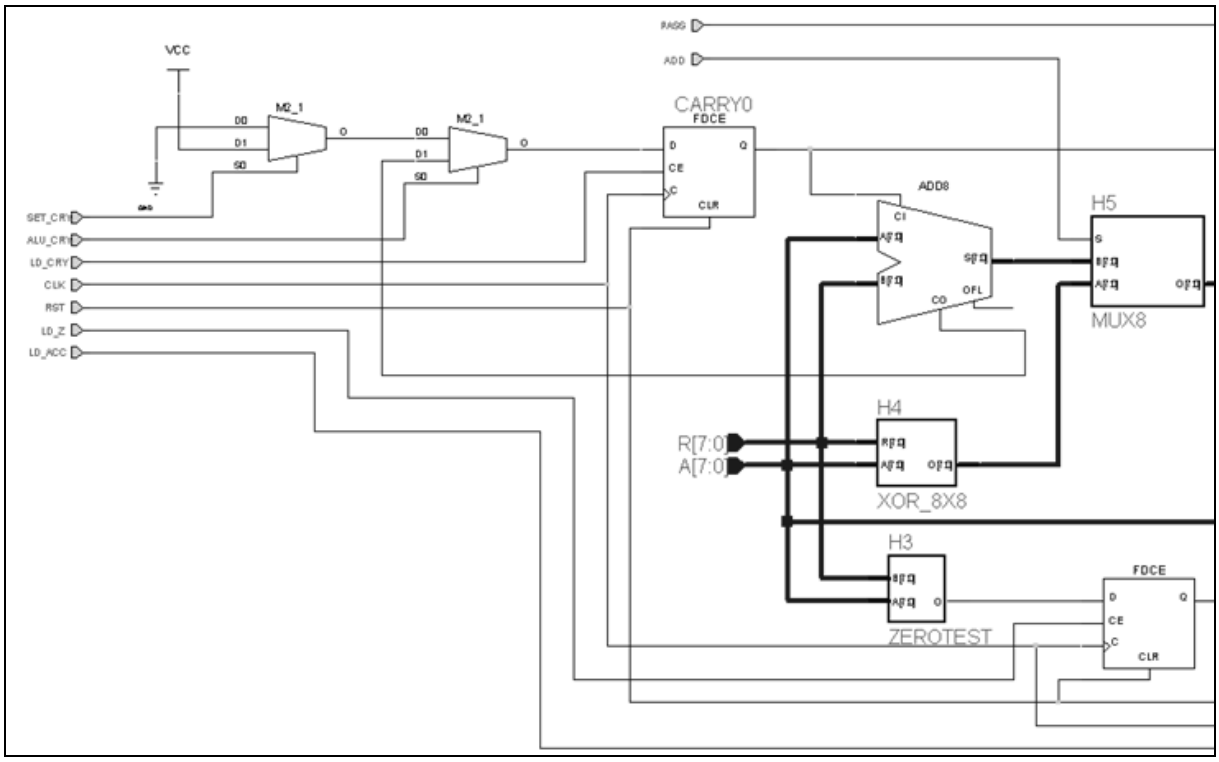


Figura 6 Circuit detailat corespunzător flag-ului de transport (Carry) C

O valoare de „1” logic la intrarea RESET va avea ca rezultat ştergerea conţinutului acumulatorului şi a celor două flaguri de stare. Ieşirile CARRY şi ZERO ale modulului ACC sunt asociate celor două flaguri. De asemenea ieşirea ACC7...ACC0 este asociată acumulatorului, setând semnalul DRV_D_ACC=”1” valoarea de la ieşirea acumulatorului va apărea pe magistrala internă D vezi Figura 2. Aceasta permite stocarea valorii provenind de la ACC în memoria externă, în setul de registre sau încărcarea ei din nou înapoi în ALU ca şi operand.

3.4. Contorul de program

Contorul de program (PC= Program Counter) are rolul de a păstra adresa locației de memorie din zona de PROGRAM, locație de memorie care conține o instrucțiune în curs de executare. Odată procesată instrucțiunea curentă adresa din PC se incrementează, astfel că aceasta va stoca adresa următoarei instrucțiuni ce urmează a fi adusă, decodificată şi executată.

În Figura 7 este prezentat circuitul de generare a adreselor. Una dintre componentele acestui circuit este şi contorul de program (modulul)PC0. Din figură se poate observa că acesta este un numărător reîncărcabil pe 8-biți (CB8CLE), luat din lista de componente standard. PC-ul se poate încărca cu o nouă valoare de pe magistrala internă de date D dacă la intrarea LD_PC este prezent un „1” logic, iar la intrarea de clock urmează un front crescător. Această operație este folosită atunci când au loc salturi în program.

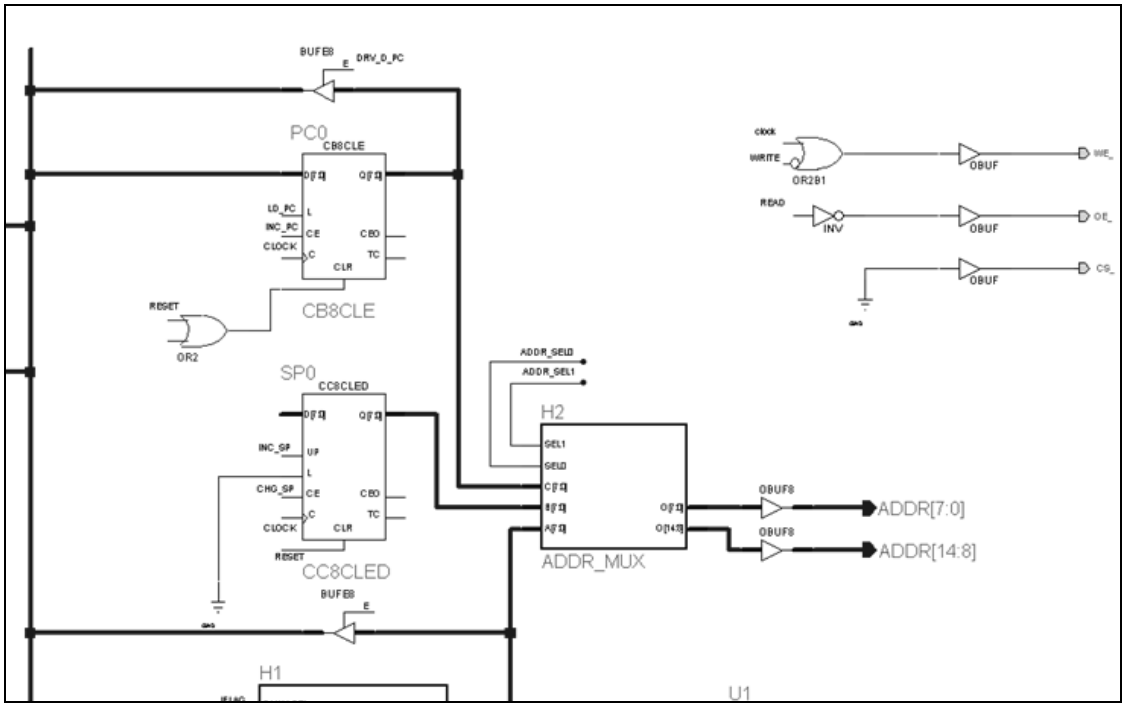


Figura 7 PC, SP, selectorul de adrese şi logica de control a memorie RAM externe

Dacă însă se urmăreşte executarea unui set de instrucţiuni de la adrese succesive, se menţine un „1” logic la intrarea INC_PC ceea ce va avea ca efect incrementarea conţinutului program counterului. Valoarea de la ieşirea program counterului poate să fie trimisă pe magistrala D activând prin intermediul semnalului DRV_D_PC = „1” bufferul tristate BUFE8. Această operaţie este necesară când se doreşte salvarea valorii PC-ului în memoria stivă. Un semnal cu valoarea „1” logic la intrările RESET sau CLR_PC au ca rezultat ştergerea conţinutului program counterului. Acest lucru este necesar înaintea executării unui program, pentru ca acesta să se deruleze de la adresa 0. De asemenea valoarea din PC mai trebuie reiniţializată şi în cazul începerii unei subrutine de tratare a întreruperilor (ISR = Interrupt Service Routine).

3.5. Indicatorul de stivă

Indicatorul de stivă sau stack pointer-ul (SP) păstrează adresele locaţiilor de memorie în care se vor stoca conţinutul program counterului PC (contor de program) şi a acumulatorului ACC.

În Figura 7 este prezentat indicatorul de stivă (componenta SP0) ca făcând parte din circuitul de generare a adreselor. Din figură se poate observa că modulul SP0 nu este altceva decât un circuit numărător reversibil (up/down)**,** reîncărcabil pe 8-biţi care face parte din lista de componente standard (CC8CLED). Un semnal „1” logic la intrarea CHG_SP va permite SP-ului să-şi modifice conţinutul. Astfel, dacă se aplică un „1” logic şi la intrarea INC_SP, conţinutul indicatorului de stivă se va incrementa, altfel (INC_SP = „0”) conţinutul SP-ului se va decrementa. Aceste operaţii de incrementare / decrementare a indicatorului de stivă sunt necesare atunci când sunt salvate valori în memoria stivă (aşa numita operaţie de *push*) sau

când sunt citite (operația *pop*) valori din memoria stivă. Dacă semnalul RESET=”1” conținutul indicatorului de stivă va fi şters, acest lucru este necesar să se facă în cadrul secvenței de pornire a microprocesorului *μP8*. Deoarece nu este necesară accesarea memoriei stivă cu adrese aleatoare intrările D7...D0 sunt lăsate neconectate iar intrarea (L) este ținută la „0” logic.

3.6. Selectorul de adrese

Microprocesorul *μP8* împarte memoria externă în trei zone şi anume: 256 octeți memorie de PROGRAM, 256 octeți memorie de DATE şi 256 octeți memorie STIVĂ. Rolul selectorului de adrese este de a activa la un moment dat una din cele trei zone şi de ai trimite o adresă care poate să provină din una din cele trei surse:

- memoria de PROGRAM este adresată de PC (Program Counter);
- memoria STIVĂ este adresată de SP (Stack Pointer);
- memoria de DATE este adresată prin intermediul unei valori provenind din unul din registrele interne.

Blocul de selectare de adrese cu cele trei intrări menționate mai sus este prezentat în Figura 7 (blocul ADDR_MUX). Prin intermediul intrărilor ADDR_SEL0 şi ADDR_SEL1 se face selectarea unei dintre intrările de date (pe 8-biți) din ADDR_MUX şi a celor 7-biți mai semnificativi care se vor concatena ceilalți 8 amintiți anterior. Astfel se va obține o adresă validă pe 15-biți. În tabelul T.4 sunt prezentate posibilitățile de selectare a celor trei zone din memoria RAM externă. De asemenea în Figura 8 este prezentată o „hartă” a memoriei RAM externe.

Tabelul T.4 Împărțirea pe zone a memoriei RAM externe

ADDR_SEL0	ADDR_SEL1	ADDR14...ADDR8	ADDR7...ADDR0	Zona RAM
0	0	1111111(7Fh)	REGB7...REGB0	DATE
0	1	1111110(7Eh)	SP7...SP0	STIVA
1	0	0000001	-	nefolosită
1	1	0000000(00h)	PC7...PC0	PROGRAM

Dacă semnalele de selecție ADDR_SEL0 = ADDR_SEL1 =”0” atunci se va accesa zona de DATE din memoria RAM externă. Pentru formarea unei adrese valide se vor concatena 8-biți (vor reprezenta cei mai nesemnificativi 8-biți ai adresei) provenind de la ieşirea B a setului de registre şi setul de 7-biți 7Fh. La fel se va proceda şi pentru adresarea celorlalte două zone ale memoriei RAM externe. Fiecare din cele trei zone de memorie ocupă câte 256 de locații din memoria RAM externă.

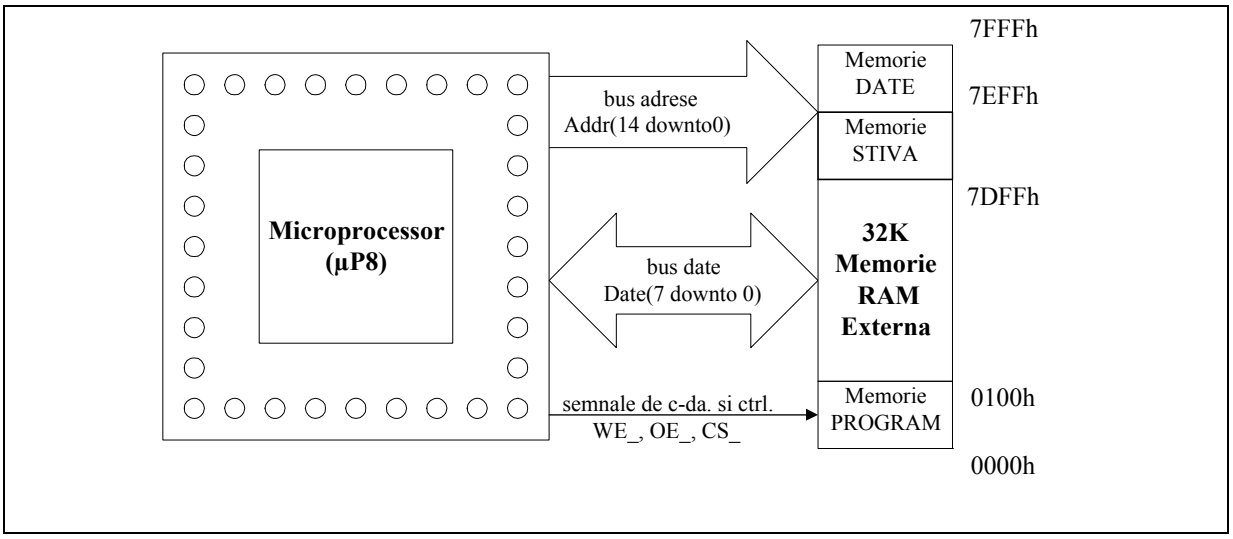


Figura 8 Diagrama bloc a microprocesorul *µP8* şi harta memoriei externe

În Figura 9 sunt prezentate detalii ale circuitului selector de adrese. Se poate observa că, componentele principale sunt două multiplexoare MUX8 conectate în serie.

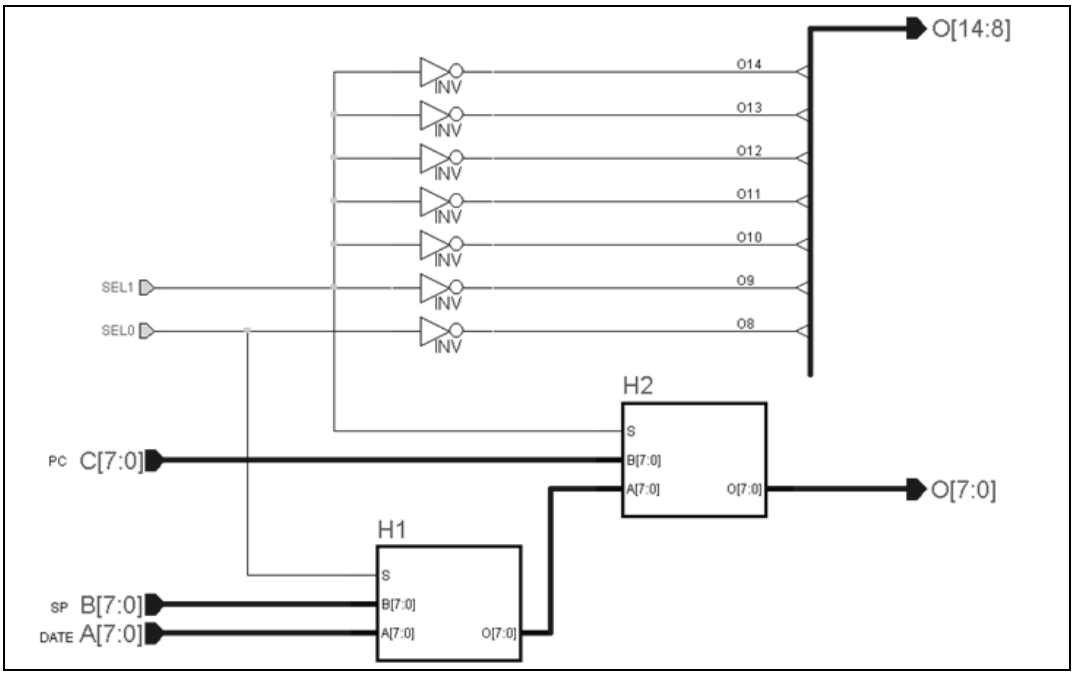


Figura 9 Circuit detailat pentru multiplexorul de adrese

Modalitatea de formare a adresei prezentată în acest paragraf a fost impusă de faptul că, se foloseşte memoria RAM externă adresabilă pe 15-biți (capacitate 32kB) de pe placa XS40. În cazul în care se folosesc alte resurse RAM externe modalitatea de adresare poate fi reconsiderată pe baza celor prezentate mai sus.

3.7. Logica de control a memoriei RAM externe

Pentru o bună sincronizare a microprocesorului cu memoria RAM mai sunt necesare câteva circuite de control, acestea sunt prezentate în Figura 7. Prin conectarea ieşirii de chip select CS_ la „0” logic memoria RAM va fi tot timpul activă. Pe durata de timp cât se citeşte memoria externă ieşirea OE_ va fi ţinută pe nivel logic „0”. Circuitul pentru generarea semnalului de scriere în memoria RAM (semnal de ieşire WE_) generează un puls „0” logic pe durata cât semnalele CLOCK= „0” şi WRITE=”1”. Această temporizare oferă timp suficient pentru ca adresa şi datele să fie stabile în momentul în care se primeşte semnal de scriere.

3.8. Portul de intrare/ieşire

Există un singur port de intrare/ieşire (IOP), acesta are separate cele două magistrale, de intrare şi de ieşire, ambele fiind pe 8-biţi. Componentele logice din care este alcătuit portul I/O sunt prezentate în Figura 10.

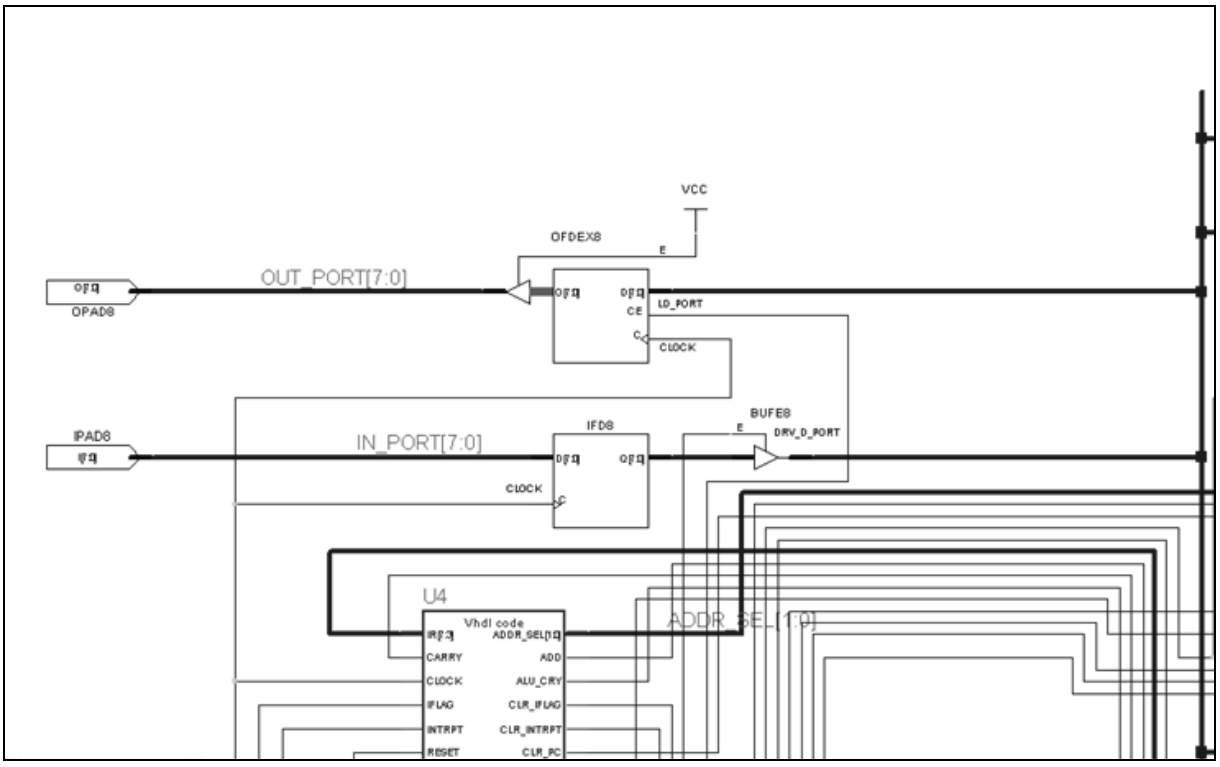


Figura 10 Portul de intrare/ieşire

Portul de ieşire este alcătuit din opt bistabile (OFDEX8) localizate în blocurile de intrare/ieşire (IOB, vezi cap.3) ale circuitului FPGA, rezultă deci că pentru implementarea acestui port nu se consumă resursele blocurilor logice configurabile (CLB). Dacă intrarea LD_PORT este „1” logic, la un front crescător a semnalului de tact la portul de ieşire va apărea valoarea de pe magistrala internă de date D.

Portul de intrare este alcătuit de asemenea din opt circuite bistabile care fac parte din blocurile IOB (IFD8) ale FPGA-ului, acestea încarcă valoarea de la intrarea IN_PORT[7...0]

la fiecare front crescător de tact. Valoarea de la ieşirea portului de intrare va apărea pe magistrala D numai când semnalul DRV_D_PORT este „1” logic şi numai sincron cu frontul pozitiv al semnalului de tact. Aceasta are ca efect sincronizarea portului de intrare cu celelalte blocuri ale microprocesorului *μP8*.

3.9. Detectorul de întreruperi

O tranziție a semnalului din „0” logic în „1” logic la pinul INT al microprocesorului va activa circuitul de sesizare a întreruperilor, care la rândul său va „alerta” microprocesorul obligându-l să-şi altereze secvența de program.

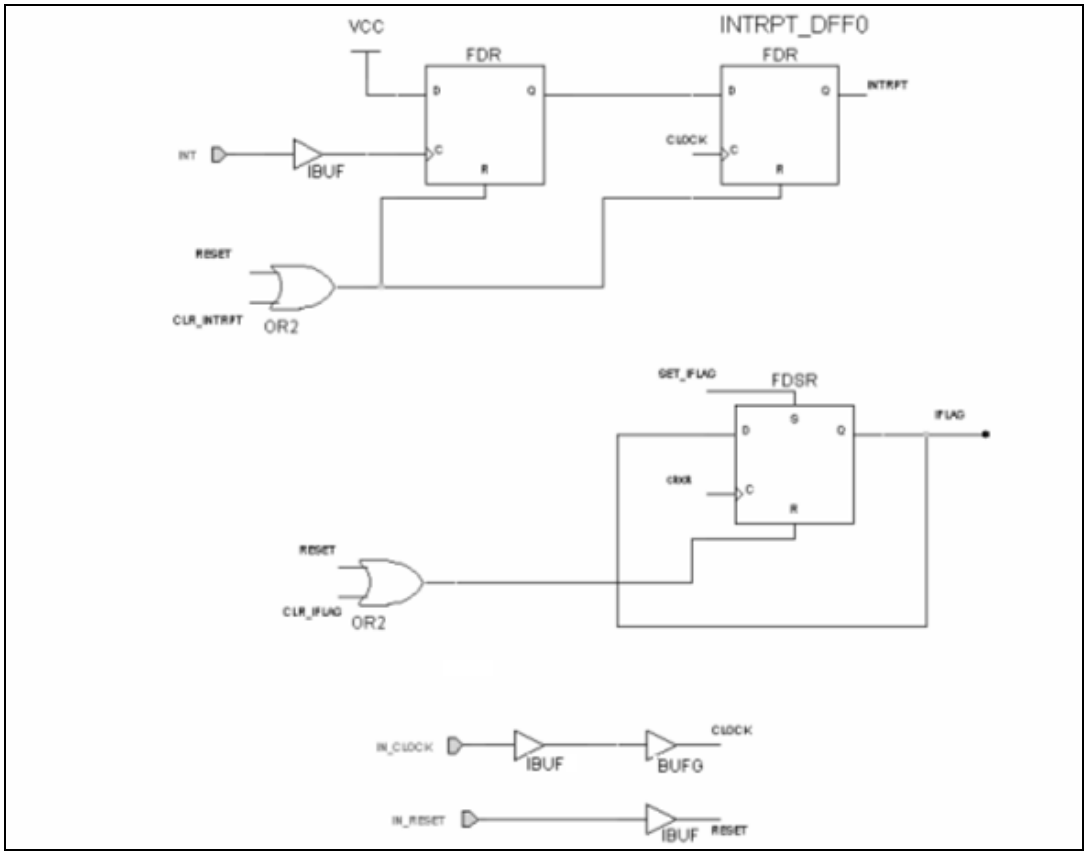


Figura 11 Circuitul de detectare şi înregistrare a întreruperilor

În Figura 11 este prezentat în detaliu circuitul de sesizare şi înregistrare a unei întreruperi, pentru microprocesorul *μP8*. Un front crescător al semnalului de la intrarea INT are ca rezultat încărcarea primului bistabil cu valoarea „1” logic, care se va propaga prin bistabilul INTRPT_DFF0 şi va apărea la ieşirea acestuia pe următorul front crescător de clock. În acest fel semnalul de întrerupere extern va fi sincronizat cu semnalul de tact, prevenind astfel apariția stărilor metastabile.

Pe lângă bistabilele care înregistrează apariția unei întreruperi mai este necesar un bistabil pentru a indica executarea unei rutine de tratare a întreruperilor, acest bistabil este IFLAG0. Dacă intrarea SET_IFLAG=”1” atunci bistabilul IFLAG0 va stoca valoarea „1” logic. Ieşirea IFLAG este folosită pentru a comuta între cele două bancuri ale setului de

registre. Ieşirile INTRPT şi IFLAG sunt intrări în decodorul de instrucţiuni, dacă INTRPT= „1” şi IFLAG=”0” (adică a apărut o întrerupere şi nu există o alta în curs de procesare) atunci decodorul de instrucţiuni va întrerupe executarea secvenţei normale de instrucţiuni şi va iniţializa secvenţa de program (subrutina) de tratare a întreruperii. Dacă ambele ieşiri INTRPT şi IFLAG sunt „1” logic microprocesorul va ignora întreruperea care apărut deoarece înseamnă că este în curs de procesare a unei întreruperi anterioare. Când intrările CLR_INTRPT şi CLR_IFLAG sunt în stare logică „1” bistabilele din circuitul de detectare şi înregistrare a întreruperilor vor fi reiniţializate cu „0” logic. Reiniţializarea acestor bistabile va ave loc şi în cazul în care se face un reset general al microprocesorului (RESET=”1”).

În Figura 11 pe lângă circuitul de detectare a întreruperilor mai este prezentată şi modalitatea de conectare a semnalelor de clock şi reset în microprocesorul *μP8*. Semnalele IN_RESET şi IN_CLOCK sunt trecute prin buffere de intrare de tipul IBUF pentru a beneficia de resursele blocurilor IOB ale circuitului FPGA. În plus semnalul IN_CLOCK mai este trecut şi printr-un buffer global BUFG, aceasta pentru a permite semnalului de clock să folosească o linie globală (vezi cp.3 resurse de interconectare FPGA). Liniile globale nu trec prin matricea programabilă de conexiuni, semnalul de clock având în acest caz distorsiuni şi întârzieri minime, ajungând aproape instantaneu la toate bistabilele.

3.10. Registrul de instrucţiuni

Registrul de instrucţiuni (IR) este pe 8-biţi şi are rolul de a stoca adresa instrucţiuni în curs de executare. În Figura 12 este prezentat registru de instrucţiuni modulul IR0 care este conectat direct la busul numit DATE. Registrul de instrucţiuni este încărcat cu toate valorile de pe magistrala DATE, pe care sunt prezente opcodurile instrucţiunilor de program stocate în RAM-ul extern., încărcarea are loc numai dacă semnalul LD_IR=”1” şi apare o tranziţie a semnalului de tact. Registrul de instrucţiuni este reiniţializat când semnalul RESET este pe „1” logic.

Magistrala DATE menţionată mai sus este bidirecţională şi asigură legătura cu memoria RAM externă. Valorile de pe magistrala internă D vor fi puse pe magistrala DATE când semnalul WRITE=”1”. Acest semnal comandă bufferul tristate OBUFE8. Datele din memoria RAM externă pot să ajungă pe magistrala D dacă semnalul de la decodorul de instrucţiuni DRV_DATA este „1” logic.

3.11. Decodorul de instrucţiuni

Decodorul de instrucţiuni (ID) este componenta cea mai importantă şi cea mai complexă a unui microprocesor. ID are rolul de a interpreta instrucţiunea curentă care se află în IR (magistrală de intrare pe 8-biţi). La interpretarea instrucţiunii curente se va ţine cont şi de starea flagurilor C şi Z cât şi de starea semnalului de la ieşirea circuitului de detectare a întreruperilor, astfel că în funcţie de acestea decodorul de instrucţiuni va controla funcţionarea celorlalte componente ale microprocesorului *μP8*.

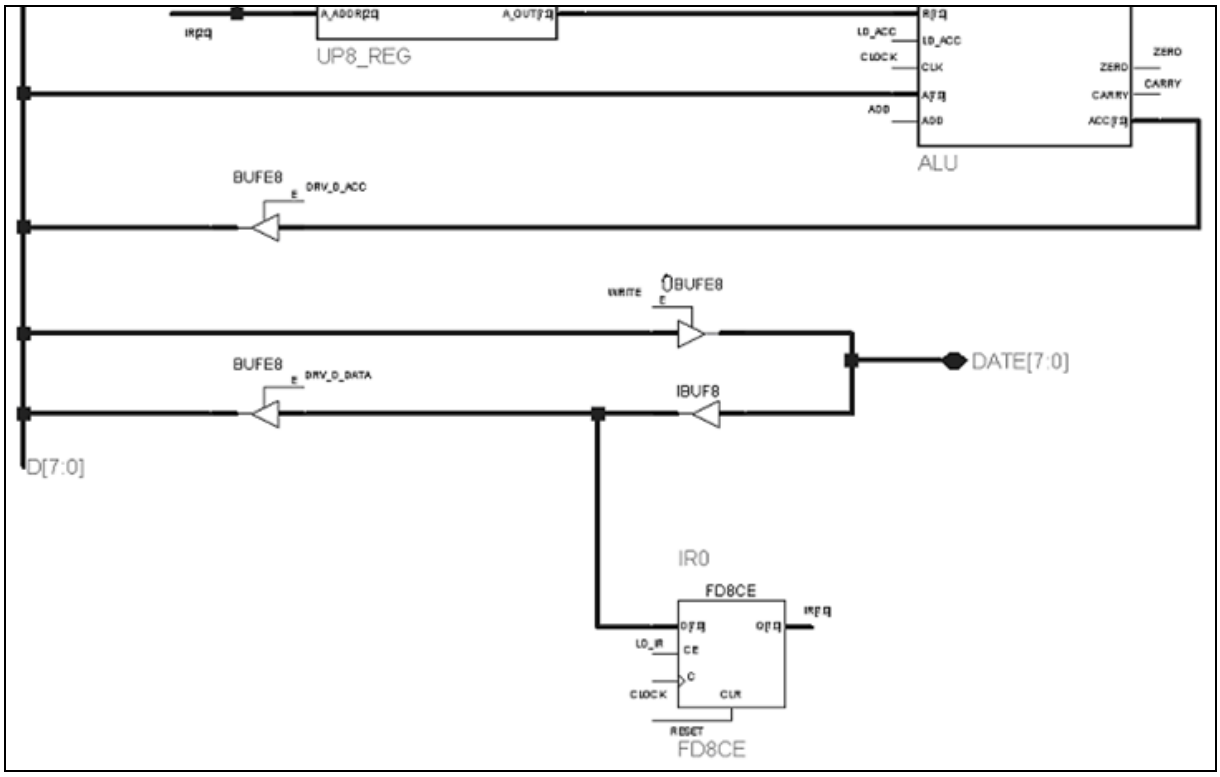


Figura 12 Registru de instrucțiuni şi interfața la magistrala externă de date

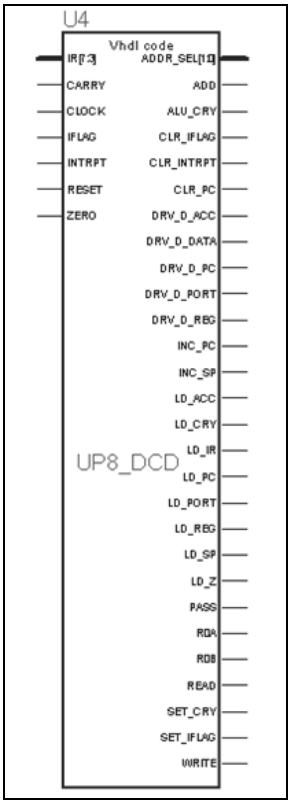


Figura 13 Decodificatorul de instrucțiuni

După cum se poate observa din Figura 13 decodorul de instrucțiuni are 14 linii de intrare în funcție de care furnizează 29 de semnale de control pentru toate componentele microprocesorului tratate în acest paragraf. Modulul up8_IDCD din Figura 13 a fost creat prin importarea unui fişier VHDL sintetizat. Conținutul fişierului VHDL se poate vedea în tabelul T.5. În continuare vor fi tratate câteva instrucțiuni, restul codului pe urmă putându-se descifra uşor.

Pentru executarea unei instrucțiuni sunt necesare trei etape, aceste etape formează aşa numitul **ciclu maşină**:

- Etapa FETCH (aducerea instrucțiuni) – microprocesorul citeşte instrucțiunea din zona de PROGRAM a memoriei RAM şi o încarcă în registrul de instrucțiuni IR;
- Etapa DECODE (decodificarea instrucțiuni) – microprocesorul (mai bine zis decodificatorul de adrese) trebuie să identifice tipul instrucțiuni care trebuie executată şi să aducă din memorie operanzii dacă este cazul;
- Etapa EXECUTE (executarea instrucțiuni) – microprocesorul trebuie să efectueze operațiile cu operanzii şi să stocheze rezultatul în memorie.

Codul VHDL al decodificatorului de instrucțiuni începe cu declararea semnalelor de intrare (liniile 9-15) şi a semnalelor de ieşire (liniile 16-45). În linia 51 se declară ca şi semnale interne (acestea vor fi sintetizate ca şi bistabile) stările microcontrolerului, după care ca şi constante se vor declara valorile pe care le pot lua cele două stări: curr_st şi next_st (liniile 54-60). De asemenea tot ca şi valori constante vor fi declarate şi cele trei zone de memorie (liniile 62-65) şi setul de instrucțiuni (liniile 68-89).

Tabelul T.5 Codul VHDL al decodificatorului de instrucțiuni

```
1   -- UP8 controller -- V1.0
2   library IEEE;
3   use IEEE.std_logic_1164.all;
4   use IEEE.std_logic_unsigned.all;
5
6   entity uP8 is
7           PORT
8           (
9                   clock: in STD_LOGIC;
10                  reset: in STD_LOGIC;                    -- reset global
11                  ir: in STD_LOGIC_VECTOR (7 downto 3); -- registru de instrucțiuni (IR)
12                  carry: in STD_LOGIC;                    -- flag carry (C)
13                  zero: in STD_LOGIC;             -- flag zero (Z)
14                  intrpt: in STD_LOGIC;                   -- sesizare întreruperi
15                  iflag: in STD_LOGIC;                    -- înregistrare întreruperi
16                  read: out STD_LOGIC;                    -- 1 când se citeşte RAM
17                  write: out STD_LOGIC;                   -- 1 când se scrie RAM
18                  addr_sel: out STD_LOGIC_VECTOR (1 downto 0); -- 2=PROG;1=DATE;0=STIVA
19                  inc_pc: out STD_LOGIC;                  -- 1 pentru incrementare PC
20                  ld_pc: out STD_LOGIC;                   -- 1 pentru încărcare PC
21                  clr_pc: out STD_LOGIC;                  -- 1 pentru ştergere PC
22                  inc_sp: out STD_LOGIC;                  -- 0=--sp; 1=sp++
23                  ld_sp: out STD_LOGIC;                   -- 1 pentru schimbarea valorii din SP
24                  ld_ir: out STD_LOGIC;                   -- 1 pentru încărcare IR
25                  ld_port: out STD_LOGIC;                 -- 1 pentru încărcare port I/O
26                  drv_d_DATE: out STD_LOGIC; -- 1 când se pun valori pe D de pe busul DATE
27                  drv_d_acc: out STD_LOGIC;   -- 1 când se pun valori pe D din acumulator
28                  drv_d_pc: out STD_LOGIC;    -- 1 când se pun valori pe D din PC
29                  drv_d_port: out STD_LOGIC;  -- 1 când se pun valori pe D de la portul I/O
30                  drv_d_reg: out STD_LOGIC;   -- 1 când se pun valori pe D de la setul de registre
31                  ld_reg: out STD_LOGIC;                          -- 1 când se încarcă setul de registre
32                  R0A: out STD_LOGIC;                             -- 1 forțează adresa din reg. A la 0
33                  R0B: out STD_LOGIC;                             -- 1 forțează adresa din reg. B la 0
34                  set_cry: out STD_LOGIC;     -- 0=resetează flagul de carry; 1=setează flagul de carry
35                  alu_cry: out STD_LOGIC;     -- 1 pentru ca în urma unei operații ALU carry să fie reîmrpospătat
36                  ld_cry: out STD_LOGIC;      -- 1 activare flag carry
37                  ld_z: out STD_LOGIC;        -- 1 activare flag zero
38                  add: out STD_LOGIC;         -- 0=XOR; 1=ADD
```

```
40                              pass: out STD_LOGIC;    -- 0=ieşire sumator trece în ACC; 1=operandul trece prin ALU la ACC
41                              ld_acc: out STD_LOGIC;          -- 1 activare acumulator
42                              set_iflag: out STD_LOGIC;       -- 1 setează pe 1 flag de întrerupere
43                              clr_iflag: out STD_LOGIC;       -- 1 ştergere flag întrerupere
44                              clr_intrpt: out STD_LOGIC       -- 1 resetare flip-flop întrerupere
45                      );
46      end uP8;
47
48
49      ARCHITECTURE uP8_arch of uP8 is
50
51      SIGNAL curr_st, next_st: STD_LOGIC_VECTOR (2 downto 0);   -- uP8
52
53      -- stările uP8 -----
54      constant R0: STD_LOGIC_VECTOR (2 downto 0) := "000";        -- stare reset 0
55      constant R1: STD_LOGIC_VECTOR (2 downto 0) := "001";        -- stare reset 1
56      constant T0: STD_LOGIC_VECTOR (2 downto 0) := "011";        -- starea normală 0
57      constant T1: STD_LOGIC_VECTOR (2 downto 0) := "010";        -- starea normală 1
58      constant T2: STD_LOGIC_VECTOR (2 downto 0) := "110";        -- starea normală 2
59      constant T3: STD_LOGIC_VECTOR (2 downto 0) := "100";        -- starea normală 3
60      constant I1: STD_LOGIC_VECTOR (2 downto 0) := "111";        -- stare de tratare a întreruperilor 1
61
62      -- definirea zonelor de memorie -----
63      constant PROGRAM: STD_LOGIC_VECTOR (1 downto 0) := "11"; -- regiunea de PROGRAM
64      constant DATE:    STD_LOGIC_VECTOR (1 downto 0) := "00";  -- regiunea de DATE
65      constant STIVA:   STD_LOGIC_VECTOR (1 downto 0) := "01";  -- regiunea de STIVA
66
67      -- uP8 opcodurile instrucţiunilor -----
68      constant STORE_REG: STD_LOGIC_VECTOR (4 downto 0) := "00000";
69      constant STORE_INX: STD_LOGIC_VECTOR (4 downto 0) := "00001";
70      constant STORE_DIR: STD_LOGIC_VECTOR (4 downto 0) := "00010";
71      constant STORE_IND: STD_LOGIC_VECTOR (4 downto 0) := "00011";
72      constant LOAD_REG: STD_LOGIC_VECTOR (4 downto 0)  := "00100";
73      constant LOAD_INX: STD_LOGIC_VECTOR (4 downto 0)  := "00101";
74      constant LOAD_DIR: STD_LOGIC_VECTOR (4 downto 0)  := "00110";
75      constant LOAD_IND: STD_LOGIC_VECTOR (4 downto 0)  := "00111";
76      constant LOAD_IMM: STD_LOGIC_VECTOR (4 downto 0)  := "01000";
77      constant IN_REG: STD_LOGIC_VECTOR (4 downto 0)    := "01100";
78      constant OUT_OP: STD_LOGIC_VECTOR (4 downto 0)    := "01101";
79      constant XOR_OP: STD_LOGIC_VECTOR (4 downto 0)    := "10000";
80      constant ADD_OP: STD_LOGIC_VECTOR (4 downto 0)    := "10001";
81      constant TEST: STD_LOGIC_VECTOR (4 downto 0)      := "10010";
82      constant CLEAR_C: STD_LOGIC_VECTOR (4 downto 0)   := "10100";
83      constant SET_C: STD_LOGIC_VECTOR (4 downto 0)     := "10101";
84      constant JC: STD_LOGIC_VECTOR (4 downto 0)        := "11000";
85      constant JZ: STD_LOGIC_VECTOR (4 downto 0)        := "11001";
86      constant JUMP: STD_LOGIC_VECTOR (4 downto 0)      := "11010";
87      constant JSR: STD_LOGIC_VECTOR (4 downto 0)       := "11011";
88      constant RET: STD_LOGIC_VECTOR (4 downto 0)       := "11100";
89      constant RETI: STD_LOGIC_VECTOR (4 downto 0)      := "11101";
90
91      BEGIN
92
93      process(clock,next_st,reset)
94      begin
95                      if reset='1' then           -- resetare asincronă
96                              curr_st <= R0;
97                      elsif (clock'event and clock='1') then
98                              curr_st <= next_st;-- la următorul front crescător de clock treci la starea următoare (next_state)
99                      end if;
100     end process;
101
102     process(curr_st,ir,carry,zero,iflag,intrpt)
103     begin
104       -- iniţializare ieşiri pentru a prevenii sintetizarea bistabilelor
105       read<='0';
106       write <= '0';
107       addr_sel <= PROGRAM;
108       inc_pc <= '0';
109       ld_pc <= '0';
110       clr_pc <= '0';
111       inc_sp <= '0';
112       ld_sp <= '0';
113       ld_ir <= '0';
114       ld_port <= '0';
```

```
115     drv_d_DATE <= '0';
116     drv_d_acc <= '0';
117     drv_d_pc <= '0';
118     drv_d_port <= '0';
119     drv_d_reg <= '0';
120     ld_reg <= '0';
121     R0A <= '0';
122     R0B <= '0';
123     set_cry <= '0';
124     alu_cry <= '0';
125     ld_cry <= '0';
126     ld_z <= '0';
127     add <= '0';
128     pass <= '0';
129     ld_acc <= '0';
130     set_iflag <= '0';
131     clr_iflag <= '0';
132     clr_intrpt <= '0';
133     next_st <= R0;
134     case curr_st is
135             when R0 =>                      -- tratare stare reset prin incrementare PC la adresa 0x02
136                     inc_pc <= '1';          -- incrementare PC la 0x01
137                     next_st <= R1;
138             when R1 =>                      -- terminare incrementare PC to 0x02
139                     inc_pc <= '1';          -- incrementare PC la 0x02
140                     next_st <= T0;          -- începere procese de aducere şi executare instrucţiuni
141             when T0 =>                      -- tratarea instrucţiunilor în desfăşurarea normală a programelor
142                     if (intrpt='1' and iflag='0') then    -- depistare întrerupere: stocare PC în STIVA
143                             set_iflag <= '1';         -- setare flag întrerupere pt. a semnaliza ca o întrerupere este în
145                                                               --curs de executare
146                             drv_d_pc<='1'; addr_sel<=STIVA; write<='1';             -- salvează PC în STIVA
147                             ld_sp<='1'; inc_sp<='1';                                -- incrementează STIVA pointer
148                             next_st <= I1;          -- continuă cu starea care tratează întreruperile
149                     else – desfăşurarea obişnuită a programului: aducerea unui opcod (instrucţiune)
150                             addr_sel<=PROGRAM; read<='1';             -- citeşte din RAM, regiunea de PROGRAM
151                             drv_d_DATE<='1'; ld_ir<='1';    -- încarcă opcodul în registrul de instrucţiuni
152                             inc_pc<='1';                    -- incrementează program counterul
153                             next_st <= T1;                  -- mergi la starea următoare :executare instrucţiune
154                     end if;
155             when I1 =>                                                      -- tratare întreruperi
156                      clr_intrpt <= '1';                     -- ştergere întreruperea curentă
157                     drv_d_acc<='1'; addr_sel<=STIVA; write<='1';  -- salvează ACC în STIVA
158                     ld_sp<='1'; inc_sp<='1';                -- incrementează indicatorul de STIVA
159                     clr_pc<='1';                            -- resetare PC
160                     next_st <= T0;                          -- se începe citirea codului de tratare a întreruperii
161             when T1 =>              -- procesarea instrucţiunilor
162                     case ir(7 downto 3) is
163                             when IN_REG =>
164                                     drv_d_port<='1'; ld_reg<='1';-- registrele se încarcă cu valorile de la portul de
165                                                                  --intrare
166                                     next_st <= T0;                  -- terminat instrucţiune: treci la următoarea
167
168                             when OUT_OP =>
169                                     drv_d_acc<='1'; ld_port<='1';     -- portul de ieşire se încarcă cu valoarea din ACC
170                                     next_st <= T0;                    -- terminat instrucţiune: treci la următoarea
171
172                             when JSR =>             -- obţine adresa subrutinei şi incrementează PC la următoarea instr.
173                                     addr_sel<=PROGRAM; read<='1';                   -- citeşte adrs. subr. din  RAM
174                                     drv_d_DATE<='1'; ld_reg<='1'; R0A<='1';   -- încarcă adrs. în reg. R0
175                                     inc_pc<='1';
176                                     next_st <= T2;          -- executarea instrucţiunii continuă şi în starea următoare
177
178                             when RET =>             -- decrementează indicatorul stivă pentru a indica adrs. de reîntoarcere
179                                     ld_sp<='1'; inc_sp<='0';            -- decrementează indicatorul de STIVA
180                                     next_st <= T2;          -- executarea instrucţiunii continuă şi în starea următoare
181
182                             when RETI =>-- dec. indic. de stivă pt. a indica locaţia unde a fost salvat ACC
183                                     ld_sp<='1'; inc_sp<='0';            -- decrementează indicatorul de STIVA
184                                     next_st <= T2;          -- executarea instrucţiunii continuă şi în starea următoare
185
186                             when JUMP =>
187                                     addr_sel<=PROGRAM; read<='1';             -- citeşte adresa de salt din RAM
188                                     drv_d_DATE<='1'; ld_pc<='1';              -- încarcă-o în PC
189                                     next_st <= T0;                  -- terminat instrucţiune: treci la următoarea
190
```

```
191                     when JC =>
192                             if carry='1' THEN               -- daca carry=1 încarcă PC cu adresa de salt
193                                     addr_sel<=PROGRAM; read<='1';-- citeşte adresa de salt din RAM
194                                     drv_d_DATE<='1'; ld_pc<='1';    -- încarcă-o în PC
195                             ELSE                                    -- altfel treci la următoarea instr.
196                                     inc_pc<='1';                    -- incrementează PC
197                             end if;
198                             next_st <= T0;                  -- terminat instrucţiune: treci la următoarea
199
200                     when JZ =>
201                             if zero='1' THEN        -- dacă flagul zero este setat, încarcă PC cu adresa de salt
202                             addr_sel<=PROGRAM; read<='1';           -- citeşte adresa de salt din RAM
203                                     drv_d_DATE<='1'; ld_pc<='1';    -- încarcă-o în PC
204                             ELSE                            -- altfel treci la următoarea instrucţiune.
205                                     inc_pc<='1';            -- incrementează  PC
206                             end if;
207                             next_st <= T0;                  -- terminat instrucţiune: treci la următoarea
208
209                     when LOAD_REG =>                -- încarcă ACC cu date din registre
210                             pass<='1'; ld_acc<='1'; drv_d_reg<='1';
211                             next_st <= T0;          -- terminat instrucţiune: treci la următoarea
212
213                     when LOAD_INX => -- încarcă ACC cu date ale căror adresa este în registre
214                             addr_sel<=DATE; read<='1';      -- adresează memoria cu adresă din registre
215                             drv_d_DATE<='1'; pass<='1'; ld_acc<='1';-- stochează  RAM DATE în ACC
216                             next_st <= T0;                  -- terminat instrucţiune: treci la următoarea
217                     when LOAD_IMM =>                -- încarcă ACC cu valori din zona de PROGRAM
218                             addr_sel<=PROGRAM; read<='1'; drv_d_DATE<='1';-- citeşte datele din RAM
219                             pass<='1'; ld_acc<='1';         -- treci datele prin ALU în ACC
220                             inc_pc<='1';                    -- inc. PC la instr. următoare.
221                             next_st <= T0;                  -- terminat instrucţiune: treci la următoarea
222                     when LOAD_DIR =>                -- încarcă R0 cu o adresă aflată în RAM
223                             addr_sel<=PROGRAM; read<='1'; drv_d_DATE<='1'; -- adu adresa din RAM
224                             R0A<='1'; ld_reg<='1';          -- salvează-o in R0
225                             inc_pc<='1';                    -- inc. PC la instr. următoare
226                             next_st <= T2;          -- executarea instrucţiunii continuă şi în starea următoare
227
228                     when LOAD_IND =>                -- încarcă R0 cu adresa unei adrese din RAM
229                             addr_sel<=PROGRAM; read<='1'; drv_d_DATE<='1';  -- adu adresa din RAM
230                             R0A<='1'; ld_reg<='1';-- salveaz-o in R0
231                             inc_pc<='1';            -- inc. PC la instr. următoare
232                             next_st <= T2;          -- executarea instrucţiunii continuă şi în starea următoare
233
234                     when STORE_REG =>               -- salvează ACC într-un registru
235                             drv_d_acc<='1'; ld_reg<='1';
236                             next_st <= T0;          -- terminat instrucţiune: treci la următoarea
237                     when STORE_INX =>               -- salvează ACC în memorie la adresa aflată în registru
238                             addr_sel<=DATE;     -- adresează RAM cu adresă din registru
239                             drv_d_acc<='1'; write<='1';         -- pune valoarea din ACC pe busul DATE
240                             next_st <= T0;          -- terminat instrucţiune: treci la următoarea
241                     when STORE_DIR =>               -- adu adresa la care va fi stocat ACC
242                             addr_sel<=PROGRAM; read<='1'; drv_d_DATE<='1';-- adu adresa din  RAM
243                             R0A<='1'; ld_reg<='1';          -- încarcă adresa în R0
244                             inc_pc<='1';            -- inc. PC la instr. următoare
245                             next_st <= T2;          -- executarea instrucţiunii continuă şi în starea următoare
246
247                     when STORE_IND =>-- adu adrs. din RAM care conţine adrs. la care va fi stocat ACC
248                             addr_sel<=PROGRAM; read<='1'; drv_d_DATE<='1';-- obţine adresa din RAM
249                             R0A<='1'; ld_reg<='1';                  -- stochează adrs. în R0
250                             inc_pc<='1';                            --inc. PC la instr. următoare
251                             next_st <= T2;          -- executarea instrucţiunii continuă şi în starea următoare
252
253                     when ADD_OP =>                  -- ACC <- ACC + Registru
254                             pass<='0'; add<='1'; ld_acc<='1'; drv_d_acc <= '1';
255                             ld_cry<='1'; alu_cry<='1';          -- încarcă în flag  carry  ALU carry out
256                             next_st <= T0;                  -- terminat instrucţiune: treci la următoarea
257                     when XOR_OP =>                  -- ACC <- ACC $ Registru
258                             pass<='0'; add<='0'; ld_acc<='1'; drv_d_acc <= '1';
259                             next_st <= T0;                  -- terminat instrucţiune: treci la următoarea
260                     when TEST =>   -- AND bit-cu-bit, testează conţinutul ACC în funcţie de un registru
261                             ld_z <= '1'; drv_d_acc<='1';        -- dacă rezultatul testări este zero flagul zero va fi 1
262                             next_st <= T0;                  -- terminat instrucţiune: treci la următoarea
263                     when SET_C =>                           -- set flag  carry pe 1
264                             ld_cry<='1'; set_cry<='1';
265                             next_st <= T0;                  -- terminat instrucţiune: treci la următoarea
```

```
266                                     when CLEAR_C =>                         -- şterge flag  carry , 0
267                                             ld_cry<='1'; set_cry<='0';
268                                             next_st <= T0;                  -- terminat instrucţiune: treci la următoarea
269                                     when others =>
270                             end case;
271                     when T2 =>                                              -- procesarea instrucţiunilor (cont.)
272                             case ir(7 downto 3) is
273                                     when JSR =>             -- salvează PC în STIVA şi inc. SP
274                                             drv_d_pc<='1'; addr_sel<=STIVA; write<='1';             --salvează PC în STIVA
275                                             ld_sp<='1'; inc_sp<='1';                                --inc. indicatorul de  STIVA
276                                             next_st <= T3;          -- executarea instrucţiunii continuă şi în starea următoare
277
278                                     when RET =>                     -- încarcă adresa de reîntoarcere din subrutină în PC
279                                             addr_sel<=STIVA; read<='1';     --citeşte adresa de reîntoarcere din STIVA
280                                             drv_d_DATE<='1'; ld_pc<='1';    --încarcă adresa in PC
281                                             next_st <= T0;                  -- terminat instrucţiune: treci la următoarea
282                                     when RETI =>                            -- reîncarcă ACC din stivă şi decrementează STIVA
283                                     addr_sel<=STIVA; read<='1';     --citeşte val. ACC din STIVA
284                                             drv_d_DATE<='1'; pass<='1'; ld_acc<='1';   --încarc-o în ACC
285                                             ld_sp<='1'; inc_sp<='0';                --decrementează indicatorul de STIVA
286                                             next_st <= T3; -- executarea instrucţiunii continuă şi în starea următoare
287
288                                     when LOAD_DIR =>        -- încarcă ACC cu date din RAM ale căror adresă se află în R0
289                                             R0B<='1'; addr_sel<=DATE; read<='1';--citeşte datele din RAM cu adrs. din R0
290                                             drv_d_DATE<='1'; pass<='1'; ld_acc<='1';   --încarcă valorile în ACC
291                                             next_st <= T0;          -- terminat instrucţiune: treci la următoarea
292                                     when LOAD_IND => -- încarcă date din RAM în R0 de la adresa aflată în R0
293                             R0B<='1'; addr_sel<=DATE; read<='1';        --obţine adresa din RAM
294                                             drv_d_DATE<='1'; R0A<='1'; ld_reg<='1';--stochează adresa în R0
295                                             next_st <= T3;          -- executarea instrucţiunii continuă şi în starea următoare
296
297                                     when STORE_DIR =>                       -- stochează ACC în RAM la adresa din R0
298                                     R0B<='1'; addr_sel<=DATE; write<='1';       --adresa de RAM est adusă în R0
299                                             drv_d_acc<='1'; write<='1';         --valoarea din ACC este  trimisă în  RAM
300                                             next_st <= T0;                  -- terminat instrucţiune: treci la următoarea
301                                     when STORE_IND =>               -- obţine adresa la care se stochează ACC
302                     R0B<='1'; addr_sel<=DATE; read<='1';       --adresa RAM este stocată în R0
303                                             drv_d_DATE<='1'; R0A<='1'; ld_reg<='1';   --stochează noua adresă în R0
304                             next_st <= T3;          -- executarea instrucţiunii continuă şi în starea următoare
305                                     when others =>
306                             end case;
307                     when T3 =>              -- procesarea instrucţiunilor (cont.)
308                             case ir(7 downto 3) is
309                                     when JSR =>                             -- încarcă adresa subrutinei în PC
310                     drv_d_reg<='1'; R0B<='1';                               --citeşte adresa subrutinei din R0
311                                             ld_pc<='1';--şi încarc-o în PC
312                                             next_st <= T0;          -- terminat instrucţiune: treci la următoarea
313                                     when RETI =>            -- PC încărcat cu adrs. de reîntoarcere din intrpt., flag intrpt. şters
314                                        addr_sel<=STIVA; read<='1';--citeşte adresa de reîntoarcere din STIVA
315                                                drv_d_DATE<='1'; ld_pc<='1';            --încarcă adresa în PC
316                                       clr_iflag<='1';--indică că s-a încheiat procesarea rutinei de tratare a întreruperi
317                                 next_st <= T0;                  -- terminat instrucţiune: treci la următoarea
318                                     when LOAD_IND =>  -- încarcă ACC cu date ale căror adresă este în R0
319                                       R0B<='1'; addr_sel<=DATE; read<='1';   --citeşte adresa de RAM în R0
320                                                 drv_d_DATE<='1'; pass<='1'; ld_acc<='1';            --încarcă valoarea în ACC
321                                                  next_st <= T0;                 -- terminat instrucţiune: treci la următoarea
322                                     when STORE_IND =>               -- stochează ACC în RAM la adresa stocată în R0
323                                                R0B<='1'; addr_sel<=DATE; write<='1';--adresa de RAM este încărcată în R0
324                                                  drv_d_acc<='1'; write<='1';  --trimite valoarea din ACC în RAM
325                             next_st <= T0;          -- terminat instrucţiune: treci la următoarea
326                                     when others =>
327                             end case;
328                     when others =>
329       end case;
330     end process;
331
332     end uP8_arch;
```

Arhitectura decodificatorului cuprinde descrierea a două procese. În primul proces se face „reinițializarea” stării curente cu starea următoare şi resetarea asincronă a decodificatorului (liniile 93-100). Cel de al doilea proces inițializează toate semnalele de ieşire ale decodificatorului (liniile 105-133), după care va fi descrisă funcționarea

decodificatorului în fiecare din cele şapte stări. Stările R0 şi R1 vor fi activate numai în faza de reinițializare după ce a fost setat semnalul de reset. Stările T0, T1, T2, T3 vor fi active pe măsură ce sunt executate instrucțiunile. Pe durata stării T0 se verifică starea logică a pinului prin intermediul căruia se detectează întreruperile, dacă se detectează o întrerupere starea care va inițializa tratarea acesteia va fi I1.

În liniile de cod 134-333 este descris modul în care se face tranziția între stări şi semnalele de ieşire care sunt afectate la fiecare tranziție. Când are loc un semnal de reset microprocesorul *μP8* va trece în starea R0 (linia 135, bistabilii care stochează stările vor fi setați cu valoarea 000 care este codul stării R0). După resetare PC este inițializat cu valoarea „0". În starea R0 PC se incrementează o dată după care se va trece în starea următoare (liniile 135-137). În starea R1, PC se va mai incrementa o dată (va conține adresa 02), iar automatul de stări (decodificatorul de instrucțiuni) va trece în starea T0 (138-141). În starea T0 sunt aduse instrucțiunile şi începe executarea programului. În următorul paragraf se va explica de ce este necesar ca programele să înceapă la adresa 02 şi se va prezenta modul de preluare a unei întreruperi.

Faptul că programele trebuie să înceapă de la adresa 02 are legătură tocmai cu nevoia de a procesa întreruperi. Astfel în starea T0 se verifică dacă a avut loc o întrerupere (linia 142, intrpt=1) şi dacă nu cumva o subrutină de tratare a întreruperilor este în curs de executare (iflag=0). Dacă condițiile prezentate mai sus sunt adevărate, semnalul *iflag* va fi setat pentru a indica că are loc executarea unei subrutine de tratare a întreruperilor (linia 143), adresa instrucțiunii curente va fi salvată în memoria RAM în zona de stivă (linia 146), după care stiva este incrementată (linia 147). În continuare decodificatorul de instrucțiuni va trece în starea I1. În această stare bistabili din circuitul de sesizare a întreruperilor vor fi inițializați cu zero (linia 156, prin aceasta se dă posibilitatea microprocesorului dea sesiza o nouă întrerupere), apoi se salvează conținutul ACC în stivă (linia 157), se incrementează indicatorul de stivă (linia 158), se şterge conținutul PC (linia 159), după care la următoarea perioadă de clock se trece în starea T0 şi se va începe procesarea programelor începând cu adresa 0 din PC. Se poate observa că toate programele (serviciile) de tratare a întreruperilor vor începe la adresa 0, iată de ce secvența normală de program începe la adresa 02. De asemenea mai este de observat faptul că microprocesorul nu va răspunde la o nouă întrerupere atâta timp cât se află într-o subrutină de tratare a întreruperilor, prin aceasta se previne cazul în care întreruperile ar fi tot timpul sesizate, dar niciodată apelată subrutina de tratare.

În continuare se va prezenta instrucțiunea *RETI* (Return from Interrrupt) care apare la sfârşitul oricărei subrutine de tratare a întreruperilor, este folosită pentru a încheia aceste subrutine şi pentru a trece la desfăşurarea normală a programului rulat de microprocesor. Executarea acestei instrucțiuni începe în starea T0 prin aducerea ei (faza „fetch" a instrucțiuni) din memoria RAM, zona de PROGRAM (linia 150). Opcodul din RAM este adus în microprocesor şi încărcat în registrul de instrucțiuni (linia 151), PC se va incrementa după care se trece în starea T1.

Acțiunile care au loc în starea T1 depind de opcodul (instrucțiunea) stocat în IR. Deoarece în acest moment acest opcod corespunde instrucțiuni *RETI* se vor executa instrucțiunile din liniile 182-184. În această stare indicatorul de stivă va fi decrementat pentru a indica locația de memorie unde a fost salvat acumulatorul. Reactualizarea valorii din acumulatorul va avea loc numai la sfârşitul stării T2 (liniile 283-286) deoarece reactualizarea valorii din indicatorul de stivă are loc numai la sfârşitul stării T1. În linia 283 zona de stivă a memoriei RAM este adresată cu valoarea din indicatorul de stivă. Valoarea din RAM este

adusă pe busul intern D, trecută (pass) prin ALU şi stocată în ACC (linia 284). Indicatorul de stivă este decrementat din nou (linia 285) şi se trece în starea T3.

La intrarea în starea T3 (liniile 313-317) SP indică locația de memorie unde a fost salvat conținutul contorului de program la intrare în ISR. Astfel PC va fi reactualizat în acelaşi mod ca şi ACC (liniile 314-315), iar flagul care indica procesarea unei întreruperi este setat pe „0" (linia 316). Decodificatorul de instrucțiuni va revenii înapoi în starea T0, în IR va fi adusă o nouă instrucțiune din RAM zona de PROGRAM de la adresa conținută de PC, astfel se reia desfăşurarea normală a programului care se executa înainte de întrerupere.

Pentru o mai bună înțelegere a funcționării decodificatorului se va mai analiza executarea unei instrucțiuni. În continuare se va analiza versiunea de adresare indirectă a instrucțiuni *LOAD*. Plecând din starea T1 după ce opcodul instrucțiuni (LOAD_IND) a fost adus din memoria de program, decodificatorul trebuie să citească adresa care urmează opcodului (linia 229). Această adresă va fi stocată temporar în registru R0, pentru aceasta la portul A se va forța adresa *0* (R0A=1) şi se va activa scrierea în bancul de registre (linia 230). În linia următoare se va incrementa PC pentru a se trece la executarea instrucțiuni următoare. Decodificatorul va trece în starea T2, în această stare decodificatorul va adresa memoria RAM (zona de DATE) cu adresa conținută de registrul R0 (linia 293). Datele din RAM vor fi aduse pe busul intern D, de unde vor fi încărcate în registrul R0 (linia 294). Astfel conținutul lui R0 a fost înlocuit cu date din RAM de la adresa care a fost stocată în R0 în starea anterioară. Pentru că este vorba de adresare indirectă în aceasta fază R0 va conține o altă adresă.

În starea T3 decodificatorul de instrucțiuni va accesa din nou zona de DATE a memoriei RAM la o adresă conținută de registrul R0 (linia 320). În linia următoare de program datele vor fi aduse în decodificator pe busul intern D, vor trece prin ALU şi se vor stoca în ACC. Cu aceasta ia sfârşit executarea instrucțiunii LOAD_IND, iar decodificatorul de instrucțiuni se va întoarce înapoi în starea T0 pentru a aduce o nouă instrucțiune.

În aceeaşi manieră ca şi cea prezentată mai sus se pot interpreta toate instrucțiunile din codul VHDL al decodificatorului. Examinând valorile logice atribuite semnalelor de la ieşirea decodificatorului se poate urmării funcționarea tuturor modulelor microprocesorului *µP8*, pe parcursul executării unei instrucțiuni.

În tabelul T.6 este prezentată structura fişierului UCF. Se poate observa că trei dintre ieşirile PC-ului pe portul paralel sunt conectate la intrările de clock, de întrerupere şi de reset ale microprocesorului *µP8*. Cei şapte biți mai puțini semnificativi ai portului de ieşire sunt conectați la afişajul 7-segmente.

Tabelul T.6 Conținutul fişierului UCF

Nume pin	Locație pin FPGA	Comentarii
NET IN_CLOCK	LOC=P44;	#argument GXSPORT D0
NET INT	LOC=P45;	#argument GXSPORT D1
NET IN_RESET	LOC=P46;	#argument GXSPORT D2
NET ADDR<0>	LOC=P3;	
NET ADDR<1>	LOC=P4;	
NET ADDR<2>	LOC=P5;	
NET ADDR<3>	LOC=P78;	
NET ADDR<4>	LOC=P79;	
NET ADDR<5>	LOC=P82;	

```
NET     ADDR<6>          LOC=P83;
NET     ADDR<7>          LOC=P84;
NET     ADDR<8>          LOC=P59;
NET     ADDR<9>          LOC=P57;
NET     ADDR<10>         LOC=P51;
NET     ADDR<11>         LOC=P56;
NET     ADDR<12>         LOC=P50;
NET     ADDR<13>         LOC=P58;
NET     ADDR<14>         LOC=P60;
NET     DATA<0>          LOC=P41;
NET     DATA<1>          LOC=P40;
NET     DATA<2>          LOC=P39;
NET     DATA<3>          LOC=P38;
NET     DATA<4>          LOC=P35;
NET     DATA<5>          LOC=P81;
NET     DATA<6>          LOC=P80;
NET     DATA<7>          LOC=P10;
NET     CS_              LOC=P65;
NET     OE_              LOC=P61;
NET     WE_              LOC=P62;
NET     OUT_PORT<0>      LOC=P25;   #segment LED S0
NET     OUT_PORT<1>      LOC=P26;   #segment LED S1
NET     OUT_PORT<2>      LOC=P24;   #segment LED S2
NET     OUT_PORT<3>      LOC=P20;   #segment LED S3
NET     OUT_PORT<4>      LOC=P23;   #segment LED S4
NET     OUT_PORT<5>      LOC=P18;   #segment LED S5
NET     OUT_PORT<6>      LOC=P19;   #segment LED S6
NET     OUT_PORT<7>      LOC=P6;
NET     IN_PORT<0>       LOC=P7;
NET     IN_PORT<1>       LOC=P8;
NET     IN_PORT<2>       LOC=P9;
NET     IN_PORT<3>       LOC=P77;
NET     IN_PORT<4>       LOC=P70;
NET     IN_PORT<5>       LOC=P66;
NET     IN_PORT<6>       LOC=P67;
NET     IN_PORT<7>       LOC=P69;
```

În continuare se va sintetiza întregul proiect şi se vor citii fişierele raport. Pentru verificarea proiectului este necesar un program de testare. Acest program face adunarea unei liste de numere şi este prezentat în tabelul T.7. Se poate observa că programul începe la adresa 02 din memoria RAM, conform celor expuse anterior.

Tabelul T.7 Programul de testare a microprocesorului

Adresa	**Opcod**	**Operand**		**Mnemonică**	**Comentariu**
0002	40	00		load #0	; R1<- adresa de start a şirului
0004	01			store R1 ;	de numere din zona de DATE
0005	40	10		load #10 ; R2<-	numărul de valori din şir
0007	02			store R2	
0008	40	00		load #0	;R3<- 0 , şterge registru de însumare (*sum*)
000A		03		store R3	
			loop:		; etichetă
000B	29			load (R1)	; ACC<- următorul nr. din listă şterge
000C	A0			clear_c	; rezultatul anterior

```
000D    8B          add R3          ; adună conținutul reg. sum cu ACC
000E    03          store R3 ;stochează rezultatul înapoi în reg. sum
000F    40  01      load #1         ; incrementează R1 astfel încât să indice
0011    A0          clear_c         ;       următorul nr. din listă
0012    89          add R1
0013    01          store R1
0014    40  FF      load #FF; decrementează R2 pt. că tocmai s-a
0016    A0          clear_c         ; adăugat încă un număr
0017    8A          add R2
0018    02          store R2
0019    40  FF      load #FF; acum, testează R2 dacă este zero
001B    92          test R2         ; (ex. pt. a vedea dacă şirul de nr. este gata)
001C    C8  20      jz  afişează    ; ieşi din buclă (loop) dacă numărarea
001E    D0  0B      jump loop       ; a luat sfărşit dacă nu continuă procesarea
            afişează:       ; etichetă
0020    23          load R3         ; ACC <- stochează suma în reg. R3
0021    68          out             ;afişează biții sumei folosind segm. LED
            halt:               ; etichetă
0022    D0  22      jump halt       ; execută această instr. şi stai în buclă
```

Opcodurile instrucțiunilor vor fi scrise în fişierul ADDLIST.HEX (numele fişierului poate fi definit de utilizator, extensia este însă obligatorie) după cum este prezentat în tabelul T. 8.

Tabelul T.8 Conținutul fişierului HEX

```
- 0E 0002 40 00 01 40 10 02 40 00 03 29 A0 8B 03 40
- 10 0010 01 A0 89 01 40 FF A0 8A 02 40 FF 92 C8 20 D0 0B
- 04 0020 23 68 D0 22
- 10 7F00 01 23 45 67 89 AB CD EF FE DC BA 98 76 54 32 10
```

În formatul fişierului din tabelul T.8, linia (-) de la începutul fiecărui rând ne spune că nu se face verificare de tip „*check sum*" a conținutului acestuia, prima valoare reprezentată în sistem hexazecimal de la începutul fiecărei linii (ex. 0E) reprezintă numărul de înregistrări (valori hexazecimale) dintr-o linie, iar a doua valoare reprezintă adresa locației din memoria RAM de la care se va face scrierea următoarelor valori. În primele trei linii ale fişierului sunt cuprinse opcodurile instrucțiunilor şi operanzii programului, aceste valori hexazecimale vor fi încărcate în zona de PROGRAM a memoriei RAM, începând cu adresa 02. Cele 16 valori hexazecimale din ultima linie vor fi încărcate în zona de DATE a memoriei începând cu adresa 0x7F00, aceste valori vor forma şirul de caractere cu care va opera programul.

Fişierul *up8.bit* care a rezultat în urma sintezei va fi adus împreună cu fişierul ADDLIST.HEX în fereastra programului GXSLOAD, iar prin procedura *drag and drop* se va face încărcarea microprocesorului *µP8* în placa XS40. Pentru resetarea microprocesorului, pentru inițierea unei întreruperi cât şi pentru rularea unui ciclu de clock se poate folosi programul GXSPORT sau o secvență de comenzi la fel ca şi cele din tabelul T.9. Respectiva secvență de comenzi se poate înscrie într-un fişier cu extensia *.bat* care se poate executa.

Tabelul T.9 Secvența de comenzi ce va fi înscrisă în fișierul BAT

Acțiune	Secvență de comenzi XSPORT
Reset	XSPORT 100 XSPORT 000
Generarea unei întreruperi	XSPORT 010 XSPORT 000
Executarea unui ciclu de tact	XSPORT 001 XSPORT 000

În timpul rulării programul nu va afişa valori intermediare pe afişajul 7-segmente. Afişarea unei valori va avea loc doar când programul va fi rulat complet, pentru aceasta sunt necesare câteva zeci de secvențe de comenzi de tipul celei din ultima linie a tabelului T.9, acestea pot fi înscrise direct în fişierul de tip *.bat*. Rezultatul însumării şirului de valori este 0x7F8 (sistem hexazecimal) şi va fi păstrat în ACC. Ultima comandă a programului va trimite acest rezultat trunchiat (doar F8) la portul de ieşire sub forma „11111000”, din care numai cei mai nesemnificativi 7-biți vor fi afişați. Astfel va trebui să observăm afişajul 7-segmente la care se vor aprinde cele patru leduri din partea superioară, celelalte patru rămânând stinse. Dacă se schimbă în fişierul UCF pinul la care este conectată intrarea de clock, după cum urmează:

```
NET IN_CLOCK LOC=P13;      #oscilatorul de 12MHz de pe placa XS40
```

atunci microprocesorul *μP8* va lucra la o frecvență de 12 MHz.

4. Observații. Teme

- În această secțiune se vor proiecta şi asambla toate modulele microprocesorului, se va face o simulare funcțională a fiecărui modul, în continuare se va sintetiza şi implementa proiectul, urmărindu-se indicațiile de la finalul secțiuni.
- Se va extinde setul de instrucțiuni, cu instrucțiuni de incrementare, decrementare, înmulțire, împărțire şi eventual comparare.
- Se va dezvolta un program pentru tratarea întreruperilor. Indicație: la adresa 00 din zona de PROGRAM se va încărca o instrucțiune de salt urmată de adresa de la care va începe propriu-zis ISR-ul.
- Să se dezvolte un program asamblor simplu care să traducă instrucțiunile microprocesorului reprezentate ca şi mnemonici, în opcoduri (format hexazecimal).