Universitatea TRANSILVANIA Brasov - România

Catedra de Electronica si Calculatoare

Dan Nicula

Nicolae Denes **Vlad Popescu**

Proiectarea circuitelor integrate digitale

- Lucrari de laborator -

Brasov - ROMÂNIA 2002

Prefaþã

Proiectarea modernã a circuitelor integrate digitale se face pe baza unei metodologii "top-down". Aceastã metodologie presupune începerea proiectãrii de la cel mai înalt nivel de abstractizare ºi ulterior partajarea ºi structurarea sistemului.
Avantajele acestei abordãri a proiectãrii are avantaje multiple:
· stãpânirea de cãtre proiectant a complexitãþii sistemului;
· posibilitatea simulãrii întregului sistem chiar în condiþiile în care existã blocuri incomplet definite sau proiectate parþial;
· obþinerea unui model HDL independent de tehnologie, care poate fi ulterior implementat într-o tehnologie disponibilã la un moment viitor.

Reprezentarea graficã a metodologiei de proiectare a circuitelor integrate digitale este prezentatã în figura 0.1.

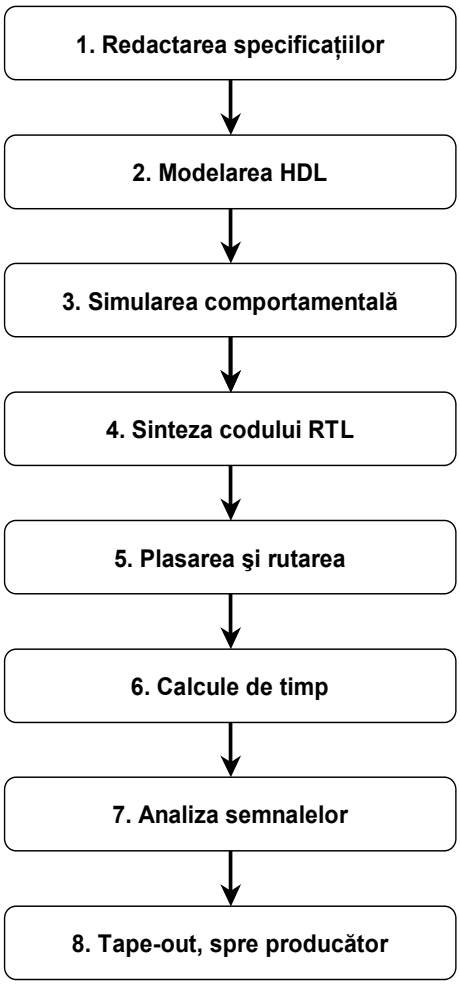


Figura 0.1: Metodologia top-down de proiectare a circuitelor integrate digitale

Etapa 1: Redactarea specificaþiilor, proiectarea arhitecturii

Specificaþiile reprezintã descrierea textualã a comportãrii sistemului ce trebuie proiectat. Acest document reprezintã interfaþa între beneficiar ºi proiectant. Experienþa a arãtat cã, deºi unora le pare inutil, acest document este deseori consultat în cursul proiectãrii pentru a rãspunde la întrebãri de detaliu. Conceperea acestui document poate consuma un timp de 10-25% din timpul dedicat proiectului. Cu cât specificaþiile sunt mai detaliate, cu atât este mai scurt timpul efectiv de scriere a modelului HDL.

Arhitectura reprezintã modul în care un chip este descompus în blocuri funcþionale interconectate. Blocurile pot fi optimizate individual. Din punct de vedere al layout-ului, arhitectura desemneazã blocurile, dimensiunea acestora, unde vor fi plasate ºi care sunt relaþiile logice dintre semnale.

Etapa 2: Modelarea HDL

Modelul HDL (Verilog sau VHDL) descrie comportamentul ºi structura sistemului. Se recomandã ca cel mai înalt nivel sã fie unul structural în care se instanþiazã blocurile funcþionale. Fiecare bloc funcþional va fi ulterior descompus la rândul lui sau modelat comportamental.

Etapa3: Simularea comportamentalã

Aceastã etapã are ca scop validarea modelului HDL. Este foarte important ca simularea sã se facã în condiþii cât mai apropiate de cele reale.
Testarea trebuie sã acopere:
· toate cazurile de funcþionare corectã;
· cazul cel mai nefavorabil;
· teste aleatoare.

Etapa 4: Sinteza codului RTL

Sinteza constã în conversia codului HDL de nivel RTL într-un cod la nivel de poartã logicã, pe baza unei biblioteci de componente specifice tehnologiei. Rezultatul sintezei este un netlist cu componente din bibliotecã interconectate. În etapa de optimizare, sintetizatorul alege din multitudinea de variante de implementare pe cea optimã în raport cu constrângerile impuse de proiectant.

Lucrarea a fost avizata de colectivul
Catedrei de Electronica si Calculatoare a
Universitatii Transilvania din Brasov

PARTEA I
NOTIUNI DE VHDL

Lucrarea

1 Modelarea circuitelor combinationale

1.1 Scopul lucrarii

· Descrierea comportamentala, structurala si cu blocuri cu garda a unui circuit combinational. Asocierea mai multor arhitecturi unei entitati.
· Crearea unui mediu de testare pentru compararea diferitelor tipuri de descrieri.

1.2 Modelarea unui MUX2:1

Multiplexorul realizeaza selectarea uneia din intrarile de date pe baza unei intrari de selectie.

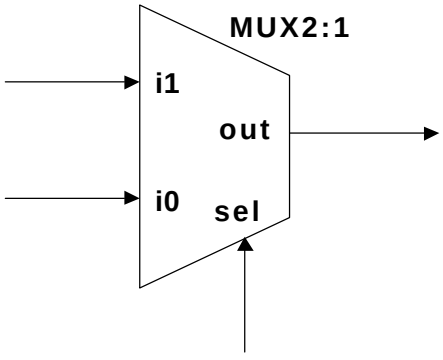


Figura 1.1: MUX2:1

Functia logica este:

$$out = sel \cdot i_1 + \overline{sel} \cdot i_0$$

1.2.1 Descrierea entitatii MUX2:1

Exemplul 1.1 prezinta descrierea entitatii MUX2:1 pentru care porturile de intrare si iesire sunt de tip **bit**.

```
--Exemplul 1.1: Entitatea MUX2:1
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY mux_2x1 IS
PORT (
          i1   : IN  bit;
          i0   : IN  bit;
          sel  : IN  bit;
          o    : OUT bit
      );
END mux_2x1;
```

1.2.2 Descrierea comportamentala a MUX2:1

Modelarea comportamentala a multiplexoarelor se poate face cu specificatii conditionale (**‘if’**) sau de selectie (**‘case’**). În exemplul 1.2 descrierea comportamentala a multipexorului 2:1 se face cu specificatia **‘if’**.
Iesirea multiplexorului depinde atât de intrarile de date (**i1** si **i2**) cât si de intrarea de selectie (**sel**). Lista de senzitivitati a procesului trebuie sa contina toate cele trei semnale.

```
--Exemplul 1.2: Descrierea comportamentala pentru MUX2:1
ARCHITECTURE mux_2x1_beh OF mux_2x1 IS
BEGIN
  PROCESS ( i1, i0, sel )
  BEGIN
    IF ( sel = ‘1‘ ) THEN
      o <= i1;
    ELSE
      o <= i0;
    END IF;
  END PROCESS;
END mux_2x1_beh;
```

1.2.3 Descrierea de tip "dataflow” a MUX2:1

Structura unui MUX2:1 este prezentata în figura 1.2.

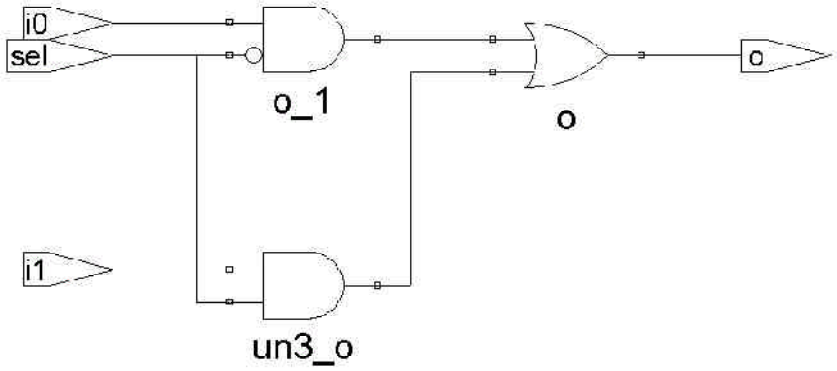


Figura 1.2: Structura MUX2:1 obtinuta în urma sintezei

```
--Exemplul 1.3: Descrierea de tip "dataflow” pentru MUX2:1
ARCHITECTURE mux_2x1_df OF mux_2x1 IS
BEGIN
  PROCESS ( i1, i0, sel )
  BEGIN
    o <= ( i1 AND sel ) OR ( i0 AND ( NOT sel ) );
  END PROCESS;
END mux_2x1_df;
```

1.2.4 Descrierea cu bloc cu garda a MUX2:1

Specificatia de bloc este o specificatie concurenta care grupeaza specificatii concurente. **Expresia de garda** este o expresie optionala care întoarce o valoare booleana. Expresia de garda este utilizata pentru determinarea valorii semnalului declarat implicit cu numele *guard*. Ori de câte ori are loc o tranzactie asociata unui semnal care apare în expresia de garda, expresia este evaluata si este actualizata valoarea semnalului *guard*.

```
--Exemplul 1.4: Descrierea cu bloc cu garda a MUX2:1
ARCHITECTURE mux_2x1_guard OF mux_2x1 IS
BEGIN
  B1: BLOCK ( ( NOT sel’STABLE ) OR ( NOT i1’STABLE ) OR
              ( NOT i0’STABLE ) )
  BEGIN
     o <= GUARDED i1 WHEN sel = ‘1‘ ELSE i0;
  END BLOCK B1;
END mux_2x1_guard;
```

Expresia de garda trebuie sa fie o expresie logica care depinde de toate intrarile care influenteaza iesirea. Expresia de garda din exemplul 1.4 va determina activarea blocului când unul din semnalele de intrare ale multiplexorului efectueaza o tranzitie.

1.2.5 Generarea vectorilor de test

Pentru testare este necesar un generator de vectori de test, care sa furnizeze semnale de intrare pentru MUX2:1 (**i1**, **i0** si **sel**).

```
--Exemplul 1.5: Generatorul de vectori de test pentru MUX2:1
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY mux_2x1_tb IS
PORT ( i1  : OUT bit;
       i0  : OUT bit;
       sel : OUT bit);
END mux_2x1_tb;

ARCHITECTURE mux_2x1_tb OF mux_2x1_tb IS
BEGIN
  i1 <= '0', '1' AFTER 10 ns,'0' AFTER 30 ns, '1' AFTER 80 ns,
             '0' AFTER 110 ns;
  i0 <= '1', '0' AFTER 40 ns,'1' AFTER 60 ns, '0' AFTER 70 ns,
             '1' AFTER 90 ns;
  sel<= '1', '0' AFTER 20 ns,'1' AFTER 50 ns, '0' AFTER 85 ns;
END mux_2x1_tb;
```

1.3 Mediul de testare

Figura 1.3 prezinta schema bloc a unui modul de testare care permite compararea rezultatelor pentru cele trei descrieri (comportamentala, de tip *"dataflow"* si cu bloc cu garda) asociate unei entitati.
Comparatorul **COMP** primeste la intrare semnalele de iesire generate de cele trei descrieri pe baza semnalelor de intrare generate de test-bench (**TB**).
COMP are ca iesire un semnal de eroare care semnaleaza aparitia unor rezultate diferite pe iesirile celor trei arhitecturi.

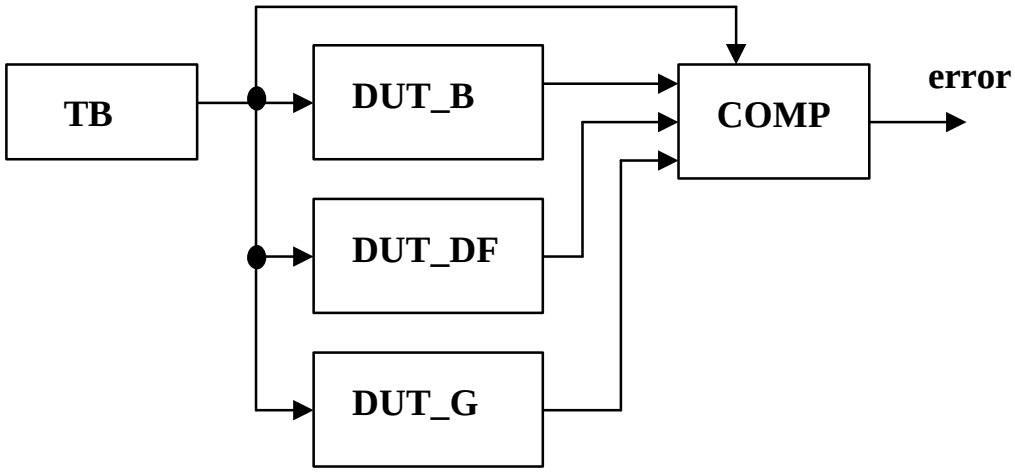


Figura 1.3: Mediu de testare pentru compararea rezultatelor celor trei descrieri asociate unei entitati

```
--Exemplul 1.6: Comparator pentru rezultatele generate de cele -
--trei descrieri asociate MUX2:1
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY mux_2x1_comp IS
PORT (i1      : IN  bit;
      i0      : IN  bit;
      sel     : IN  bit;
      o_beh   : IN  bit;
      o_df    : IN  bit;
      o_guard : IN  bit;
      error   : OUT bit);
END mux_2x1_comp;

ARCHITECTURE mux_2x1_comp OF mux_2x1_comp IS
BEGIN
  P2: PROCESS ( i1, i0, sel )
  BEGIN
    IF ( ( o_beh = o_df ) AND ( o_beh = o_guard ) ) THEN
      error <= '0';
    ELSE
      error <= '1';
    END IF;
  END PROCESS P2;
END mux_2x1_comp;
```

Descrierea VHDL a mediului de testare din figura 1.3 este prezentata în exemplul 1.7.

```
--Exemplul 1.7: Descrierea mediului de testare pentru
--compararea rezultatelor generate de cele trei descrieri
--asociate MUX2:1
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY mux_2x1_test IS
END mux_2x1_test;

ARCHITECTURE mux_2x1_test OF mux_2x1_test IS
  COMPONENT mux_2x1
  PORT ( i1  : IN  bit;
         i0  : IN  bit;
         sel : IN  bit;
         o   : OUT bit );
  END COMPONENT;

  COMPONENT mux_2x1_tb
  PORT ( i1  : OUT bit;
         i0  : OUT bit;
         sel : OUT bit);
  END COMPONENT;

  COMPONENT mux_2x1_comp
  PORT ( i1       : IN  bit;
         i0       : IN  bit;
         sel      : IN  bit;
         o_beh    : IN  bit;
         o_df     : IN  bit;
         o_guard  : IN  bit;
         error    : OUT bit);
  END COMPONENT;

  SIGNAL i1       : bit;
  SIGNAL i0       : bit;
  SIGNAL sel      : bit;
  SIGNAL o_beh    : bit;
  SIGNAL o_df     : bit;
  SIGNAL o_guard  : bit;
  SIGNAL error    : bit;

  FOR TB: mux_2x1_tb USE ENTITY work.mux_2x1_tb(mux_2x1_tb);
  FOR DUT_B: mux_2x1 USE ENTITY work.mux_2x1(mux_2x1_beh);
  FOR DUT_DF: mux_2x1 USE ENTITY work.mux_2x1(mux_2x1_df);
```

```
  FOR DUT_G: mux_2x1 USE ENTITY work.mux_2x1(mux_2x1_guard);
  FOR COMP: mux_2x1_comp USE ENTITY work.mux_2x1_comp(
  mux_2x1_comp);
BEGIN
  TB: mux_2x1_tb PORT MAP (   i1   => i1,
                              i0   => i0,
                              sel  => sel );
  DUT_B: mux_2x1 PORT MAP (   i1   => i1,
                              i0   => i0,
                              sel  => sel,
                              o    => o_beh );
  DUT_DF: mux_2x1 PORT MAP (  i1   => i1,
                              i0   => i0,
                              sel  => sel,
                              o    => o_df );
  DUT_G: mux_2x1 PORT MAP (   i1   => i1,
                              i0   => i0,
                              sel  => sel,
                              o    => o_guard );
  COMP: mux_2x1_comp PORT MAP(i1      => i1,
                              i0      => i0,
                              sel     => sel,
                              o_beh   => o_beh,
                              o_df    => o_df,
                              o_guard => o_guard,
                              error   => error );
END mux_2x1_test;
```

1.4 Modelarea unui decodificator

Functia logica a unui decodificator de 3 biti este descrisa de tabelul de adevar 1.1. Decodificatorul genereaza 1 doar pentru iesirea a carui indice este egal cu codul de intrare, celelalte iesiri fiind 0.

i2	i1	i0	o7	o6	o5	o4	o3	o2	o1	o0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Tabelul 1.1: Tabelul de adevar pentru DCD de 3 biti

Porturile decodificatorrului se definesc ca fiind de tipul **bit_vector** (vector de biti), tip de data definit în pachetul **std_logic_1164**. Descrierea comportamentala pentru DCD este prezentata în exemplul 1.9.

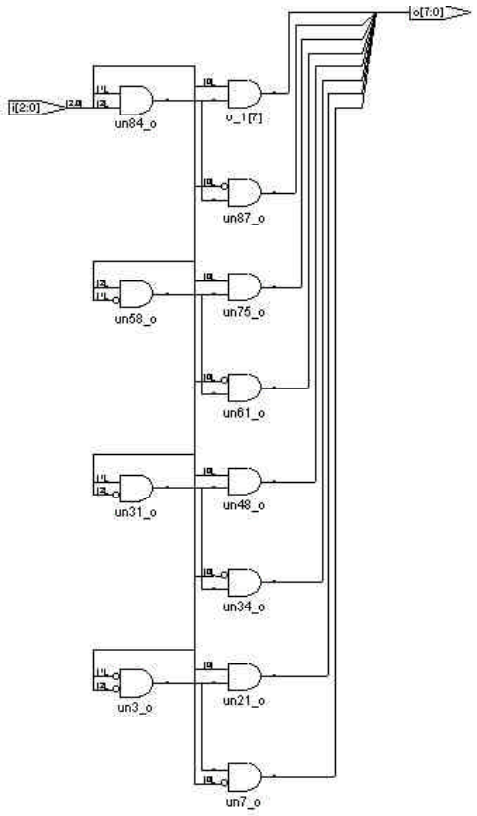
```
--Exemplul 1.8: Declararea entitatii DCD de 3 biti
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY dcd_3 IS
PORT ( i : IN  bit_vector(2 DOWNTO 0);
       o : OUT bit_vector(7 DOWNTO 0) );
END dcd_3;
```

```
--Exemplul 1.9: Descrierea comportamentala pentru DCD
ARCHITECTURE dcd_3_beh OF dcd_3 IS
BEGIN
  WITH i SELECT
    o <= "00000001” WHEN "000”,
         "00000010” WHEN "001”,
         "00000100” WHEN "010”,
         "00001000” WHEN "011”,
         "00010000” WHEN "100”,
         "00100000” WHEN "101”,
         "01000000” WHEN "110”,
         "10000000” WHEN "111”,
         "00000000” WHEN OTHERS;
END dcd_3_beh;
```

Observatie: Este necesara folosirea clauzei OTHERS pentru a asigura completitudinea specificatiei.

```
--Exemplul 1.10: Descrierea de tip "dataflow” pentru DCD
ARCHITECTURE struct OF dcd_3 IS
BEGIN
  PROCESS (i)
  BEGIN
    o(0) <= (NOT i(2)) AND (NOT i(1)) AND (NOT i(0));
    o(1) <= (NOT i(2)) AND (NOT i(1)) AND i(0);
    o(2) <= (NOT i(2)) AND i(1) AND (NOT i(0));
    o(3) <= (NOT i(2)) AND i(1) AND i(0);
    o(4) <= i(2) AND (NOT i(1)) AND (NOT i(0));
    o(5) <= i(2) AND (NOT i(1)) AND i(0);
    o(6) <= i(2) AND i(1) AND (NOT i(0));
    o(7) <= i(2) AND i(1) AND i(0);
  END PROCESS;
END struct;
```

Structura DCD este prezentata în figura 1.4 si s-a obtinut în urma sintezei descrierii din exemplul 1.10. Functiile pentru cele 8 iesiri se obtin direct din tabelul de adevar.



biti

Conditia de gardare pentru descrierea cu bloc cu garda (exemplul 1.11) trebuie sa se activeze când una din intrari se modifica. Ultimul ELSE corespunde clauzei OTHERS din descrierea comportamentala.

```
--Exemplul 1.11: Descrierea cu bloc cu garda pentru DCD
ARCHITECTURE dcd_3_guard OF dcd_3 IS
BEGIN
  B1: BLOCK ( i(2)'STABLE OR i(1)'STABLE OR i(0)'STABLE )
  BEGIN
    o <= GUARDED    "00000001" WHEN i = "000" ELSE
                    "00000010" WHEN i = "001" ELSE
                    "00000100" WHEN i = "010" ELSE
                    "00001000" WHEN i = "011" ELSE
                    "00010000" WHEN i = "100" ELSE
                    "00100000" WHEN i = "101" ELSE
                    "01000000" WHEN i = "110" ELSE
                    "10000000" WHEN i = "111" ELSE
                    "00000000";
  END BLOCK B1;
END dcd_3_guard;
```

1.5 Desfasurarea lucrarii

1.5.1 Simularea functionarii MUX2:1

Simulati functionarea MUX2:1, parcurgând urmatoarele etape:

· porniti *ModelSim* si schimbati directorul de lucru acolo unde exista sursele VHDL:
Fereastra principala: ***File > Change Directory >***
· creati un director de lucru:
Fereastra principala: ***Design > Create a New Library > work***
· compilati fisierele:
mux_2x1.vhd
mux_2x1_beh.vhd
mux_2x1_df.vhd
mux_2x1_quard.vhd
mux_2x1_tb.vhd
mux_2x1_comp.vhd
mux_2x1_test.vhd
· simulati entitatea **mux_2x1_test** cu arhitectura **mux_2x1_test**:
Fereasta principala: ***Design > Load Design > mux_2x1_test***
· deschideti fereastra cu forme de unda:
Fereastra principala: ***View > Signals, View > Wave > Signals in Region***
· rulati 800 ns prin comanda:
Prompter: ***run 800 ns***

Figura 1.5 prezinta formele de unda rezultate în urma simularii cu *ModelSim* a mediului de testare pentru MUX2:1.

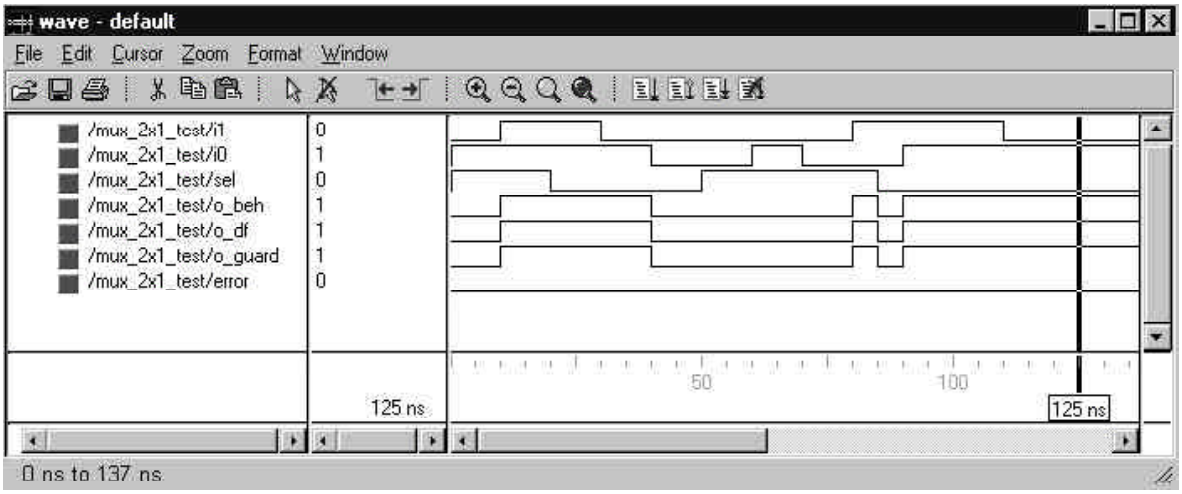


Figura 1.5: Forme de unda rezultate în urma simularii

1.5.2. Simularea functionarii DCD de 3 biti

Simulati decodificatorul descris în paragraful 1.4:

· scrieti un generator de vectori de test;
· construiti un mediu de testare care sa permita compararea rezultatelor celor trei descrieri asociate decodificatorului;
· simulati functionarea decodificatorului.

1.6. Probleme

1.6.1 Modelati în VHDL un codificator de 3 biti prioritar având intrarea **in** si iesirea **out** de tip **bit_vector**. Iesirea reprezinta codul binar al intrarii celei mai semnificative care are valoarea 1.

1.6.2 Modelati în VHDL un circuit MUX 2:1 pe 4 biti având în structura sa circuite MUX 2:1. Intrarile de date sunt **a0**, **a1**, **a2**, **a3** si **b0**, **b1**, **b2**, **b3**. Intrarea de selectie este **sel**. Iesirile **z0**, **z1**, **z2** si **z3** iau valorile **a0**-**a3** daca **sel**=’0' si iau valorile **b0**-**b3** daca **sel**=’1'. Scrieti un model de generator de vectori de test si testati modelul circuitului MUX prin simulare.

1.6.3 Scrieti un model comportamental pentru un circuit de limitare cu trei valori întregi: **data_in**, **lower** si **upper**. Sistemul are doua iesiri: **data_out** de tip întreg si **out_of_limits** de tip ‘bit’. Daca **data_in** este între **lower** si **upper** atunci **data_out** are aceiasi valoare cu **data_in**. Daca **data_in** este mai mica decât **lower** atunci **data_out** are valoarea **lower**. Daca **data_in** este mai mare decât **upper** atunci **data_out** are valoarea **upper**. Iesirea **out_of_limits** indica momentele când **data_in** este limitata.

Lucrarea

2 Modelarea circuitelor aritmetice

2.1 Scopul lucrarii

· Modelarea de tip *"dataflow"* a unitatiilor aritmetice si logice (**ALU**).
· Monitorizarea rezultatelor.
· Prezentarea bibliotecii aritmetice pentru operatii cu tipul de date **bit_vector**.

2.2. Modelarea unui sumator

Prin sinteza, operatorilor aritmetici le sunt asociate structuri complexe specifice tehnologiei pentru care se face sinteza.
Structura unui sumator pe 8 biti este prezentata în figura 2.1.

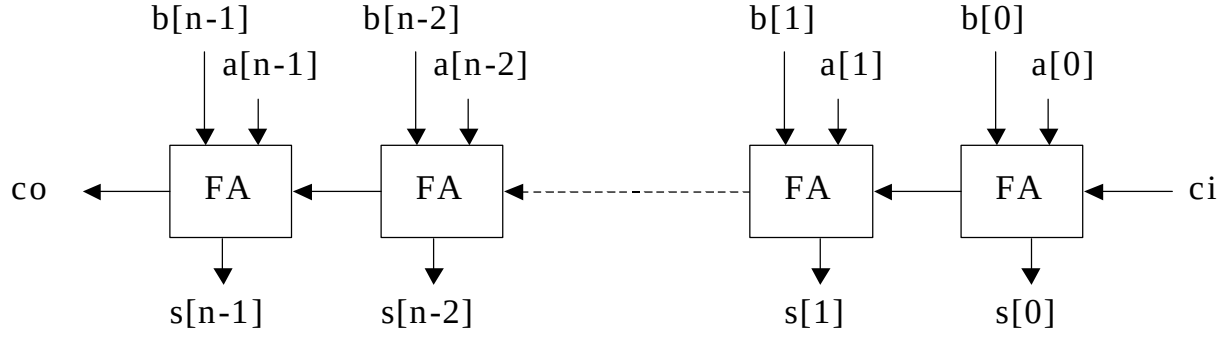


Figura 2.1: Structura unui sumator cu propagarea transportului

Un sumator cu propagarea transportului (**ripple-carry adder**) pe **n** biti este format din **n** celule de sumator complet de un bit (**FA** – Full Adder). Tabelul de adevar pentru un sumator complet de un bit este prezentat în tabelul 2.1.

a	b	cr_in	s	cr_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tabelul 2.1: Tabelul de adevar pentru sumatorul complet de un bit

Ecuatiile iesirilor sunt:

$$s_i = cr_i \oplus a_i \oplus b_i$$
$$cr_{i+1} = a_i \cdot b_i + cr_i \cdot (a_i + b_i)$$

```
--Exemplu 2.1: Entitatea sumatorului de 8 biti.
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY add_8 IS
PORT ( a  : IN  bit_vector(7 DOWNTO 0);
       b  : IN  bit_vector(7 DOWNTO 0);
       s  : OUT bit_vector(7 DOWNTO 0) );
END add_8;
```

În descrierea de tip *"dataflow"* (exemplul 2.2) se declara semnalul intermediar **carry** care corespunde bitului de transport.

```
--Exemplu 2.2: Descrierea "dataflow" a sumatorului de 8 biti.
ARCHITECTURE add_8_df OF add_8 IS
  SIGNAL carry : bit_vector(8 DOWNTO 0);
BEGIN
  carry(0)          <= '0';
  s                 <= a XOR b XOR carry(7 DOWNTO 0);
  carry(8 DOWNTO 1) <= (a AND b) OR (carry(7 DOWNTO 0) AND
                       (a OR b));
END add_8_df;
```

În urma sintezei descrierii din exemplul 2.2 rezulta structura din figura 2.2.

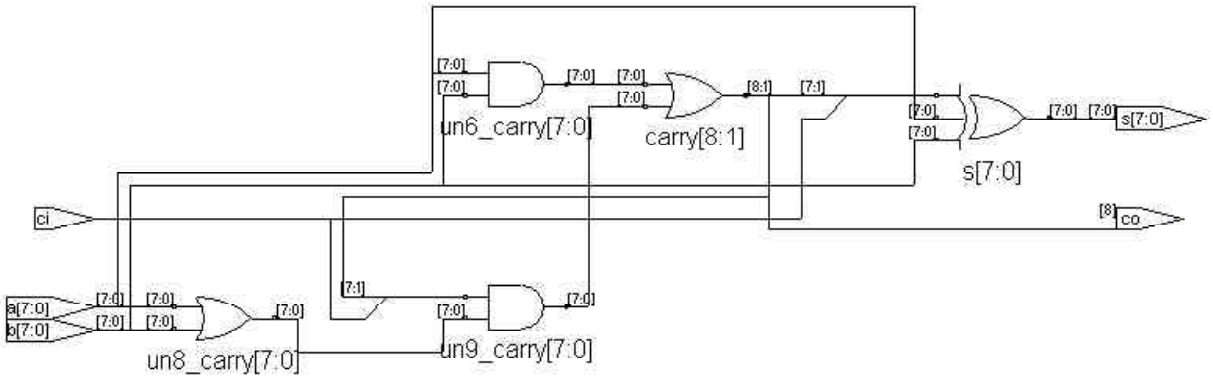


Figura 2.2: Structura sumatorului pe 8 biti rezultata în urma sintezei

2.3 Monitorizarea rezultatelor

Testarea circuitelor complexe se face prin monitorizarea rezultatelor. Aceasta presupune compararea rezultatelor generate de circuitul supus testarii cu rezultatele unui circuit martor (monitor). Descrierea monitorului nu trebuie sa se supuna constrângerilor de modelare pentru sintetizabilitate.
Figura 2.3 prezinta schema bloc a unui modul de test care permite monitorizarea rezultatelor.

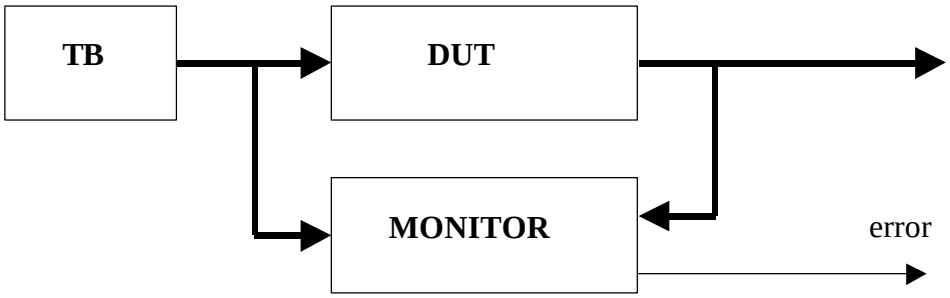


Figura 2.3: Modul de test pentru monitorizarea rezultatelor.

Generatorul de vectori de test (**TB**) trebuie sa genereze semnalele de intrare pentru sumator. Pentru testarea completa a unui circuit combinational cu **n** intrari, **TB** trebuie sa genereze toate combinatiile posibile pe **n** biti. Daca acest lucru nu este posibil, din cauza numarului mare de vectori de test, se încearca o acoperire cât mai uniforma a întregului spatiu al vectorilor de intrare.

```
--Exemplu 2.3: Test-bench pentru sumatorul de 8 biti
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE work.bit_arith.ALL;
ENTITY add_8_tb IS
GENERIC (per : TIME := 10 ns);
PORT ( a  : OUT bit_vector(7 DOWNTO 0);
       b  : OUT bit_vector(7 DOWNTO 0) );
END add_8_tb;

ARCHITECTURE add_8_tb OF add_8_tb IS
  SIGNAL a_int : bit_vector(7 DOWNTO 0) := "00000000”;
  SIGNAL b_int : bit_vector(7 DOWNTO 0) := "11111111”;
BEGIN
  a_int <= a_int + "00000001” AFTER per;
  b_int <= b_int - "00000001” AFTER per;
  a <= a_int;
  b <= b_int;
END add_8_tb;
```

Parametrul generic **per** permite parametrizarea timpului dupa care se incrementeaza sau se decrementeaza cele doua numaratoare. Semnale generate de **TB** se vor aplica si monitorului care, pe baza acestor semnale, calculeaza iesirea asteptata de la sumator.

```
--Exemplu 2.4: Monitor pentru sumatorul de 8 biti.
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE work.bit_arith.ALL;

ENTITY add_8_monitor IS
PORT ( a   : IN  bit_vector(7 DOWNTO 0);
       b   : IN  bit_vector(7 DOWNTO 0);
       s   : IN  bit_vector(7 DOWNTO 0);
       err : OUT bit );
END add_8_monitor;

ARCHITECTURE add_8_monitor OF add_8_monitor IS
BEGIN
  PROCESS( a, b, s )
  BEGIN
    IF( s = a + b ) THEN
       err <= ‘0‘;
    ELSE
       err <= ‘1‘;
    END IF;
  END PROCESS;
END add_8_monitor;
```

Descrierea comportamentala a sumatorului se face folosind operatorul **'+'** pentru tipul de data **bit_vector** definit în pachetul **bit_arith**.
Monitorul compara rezultatul generat intern **s_int** cu iesirea **s** al **DUT** si genereaza un semnal de eroare (**error**).

2.4 Pachetul bit_arith

Pentru ca circuitele aritmetice cu porturi de tip **bit_vector** sa poata fi descrise comportamental cu ajutorul operatorilor aritmetici este necesara folosirea pachetului **bit_arith.** O parte din declararea acestui pachet este prezentata în exemplul 2.5.

```
--Exemplu 2.5: Declararea pachetului bit_arith
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

PACKAGE bit_arith IS

  FUNCTION "+"(op1, op2: BIT_VECTOR) RETURN BIT_VECTOR;
  FUNCTION "-"(op1, op2: BIT_VECTOR) RETURN BIT_VECTOR;
  FUNCTION sh_left(op1: BIT_VECTOR; op2: NATURAL)
                                        RETURN BIT_VECTOR;
  FUNCTION sh_right(op1: BIT_VECTOR; op2: NATURAL)
                                        RETURN BIT_VECTOR;
  …
  FUNCTION To_BitVector (oper: INTEGER; length: NATURAL)
                                        RETURN BIT_VECTOR;
  …
END bit_arith;
```

Pachetul **bit_arith** contine:
· declararea operatorilor aritmetici pentru tipul de data **bit_vector**;
· definirea functiilor de deplasare pentru tipul de data **bit_vector**;
· functii de conversie a diferitelor tipuri de date în **bit_vector**.

2.5 Desfasurarea lucrarii

Simulati functionarea sumatorului pe 8 biti parcurgând urmatoarele etape:

Compilati fisierele:

- **add_8.vhd**
- **add_8_df.vhd**
- **bit_arith.vhd**
- **add_8_tb.vhd**
- **add_8_monitot.vhd**
- **add_8_test.vhd**

Simulati entitatea **add_8_test** cu arhitectura **add_8_test**.

2.6 Probleme

2.6.1 Modelati o unitate logico-aritmetica (ALU) cu intrari si iesiri de date de tip *bit_vector*. Operatiile pe care trebuie sa le execute depind de intrarea **op_code** si sunt codificate astfel:

op_code	**functia**
00	a + b
01	a - b
10	a **AND** b
11	a **OR** b

2.6.2 Concepeti un decodificator de adrese pentru un sistem cu microprocesor. Decodificatorul are un port de adresa de tip natural si un numar de iesiri active în zero, fiecare activându-se când adresa este într-un anumit domeniu. Iesirile si domeniile de adrese corespunzatoare sunt:

ROM_sel_n	16#0000# - 16#3FFF#
RAM_sel_n	16#4000# - 16#5FFF#
PIO_sel_n	16#8000# - 16#8FFF#
SIO_sel_n	16#9000# - 16#9FFF#
INT_sel_n	16#F000# - 16#FFFF#

Lucrarea

3 Modelarea latch-urilor si a bistabilelor

3.1 Scopul lucrarii

· Modelarea comportamentala si cu blocuri cu garda a circuitelor secventiale.
· Descrierea comportamentala a unui bistabil.
· Compararea comportamentului unui latch cu cel al unui bistabil.

3.2 Descrierea unui latch

Figura 3.1 prezinta schema bloc a unui latch activ pe palierul de ‘1’ al semnalului **clk**.

Figura 3.1: Schema bloc a unui latch de tip D

```
--Exemplul 3.1: Entitatea asociata latch-ului din figura 3.1
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY latch_d IS
PORT ( q   : OUT bit;
       d   : IN  bit;
       clk : IN  bit );
END latch_d;
```

Descrierea VHDL comportamentala si cu blocuri cu garda pentru latch este prezentata în continuare.

```
--Exemplul 3.2: Descrierea comportamentala pentru latch
ARCHITECTURE latch_d_beh OF latch_d IS
BEGIN
  PROCESS ( d, clk )
  BEGIN
    IF ( clk = '1' ) THEN
      q  <= d;
    END IF;
  END PROCESS;
END latch_d_beh;
```

Lista de senzitivitati pentru descrierea comportamentala trebuie sa contina nu numai semnalul de ceas (**clk**) ci si semnalul de date (**d**). Cât timp **clk = 1**, iesirea **q** trebuie sa urmareasca intrarea **d**.

```
--Exemplul 3.3: Descrierea cu bloc cu garda pentru latch.
ARCHITECTURE latch_d_guard OF latch_d IS
BEGIN
  B1: BLOCK ( clk = '1' )
  BEGIN
    q  <= GUARDED d;
  END BLOCK B1;
END latch_d_guard;
```

3.3 Modelarea unui bistabil de tip D

Schema bloc asociata unui bistabil de tip D activ pe frontul pozitiv al semnalului de ceas este prezentata în figura 3.2.
La nivelul porturilor de intrare-iesire, entitatea bistabilului de tip D este identica cu cea a latch-ul de tip D, doar comportarea fiind diferita.

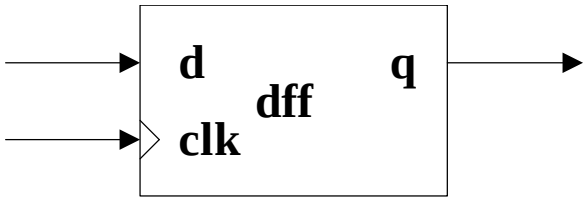


Figura 3.2: Schema bloc a unui bistabil de tip D

```
--Exemplul 3.4: Entitatea asociata bistabilului de tip D
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY dff_d IS
PORT ( q   : OUT bit;
       d   : IN  bit;
       clk : IN  bit );
END dff_d;


--Exemplul 3.5:Descrierea comportamentala a bistabilului de
--tip D
ARCHITECTURE dff_d_beh OF dff_d IS
BEGIN
  PROCESS ( d, clk )
  BEGIN
    IF ( ( clk'EVENT ) AND ( clk = '1' ) ) THEN
      q <= d;
    END IF;
  END PROCESS;
END dff_d_beh;


--Exemplul 3.6:Descrierea cu bloc cu garda a bistabilului
--tip D
ARCHITECTURE dff_d_guard OF dff_d IS
BEGIN
  B1: BLOCK ( ( NOT clk'STABLE ) AND ( clk = '1' ) )
  BEGIN
    q  <= GUARDED d;
  END BLOCK B1;
END dff_d_guard;
```

3.4 Comparatie latch-bistabil

Pentru a compara functionarea latch-ului de tip D cu functionarea bistabilului de tip D este necesara construirea unui mediu de testare ca cel din figura 3.3. Acest modul de test este descris în exemplul 3.7.

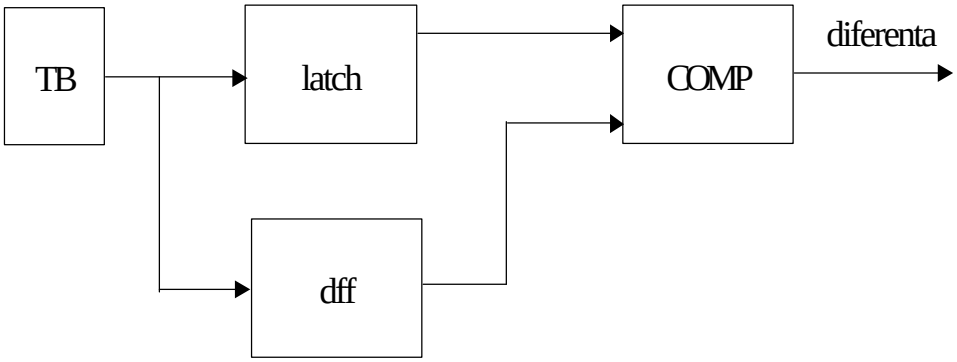


Figura 3.3: Mediu de testare pentru comparatia latch-bistabil de tip D

```
--Exemplul 3.7: Mediul de testare pentru comparatia latch
--bistabil de tip D.
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY latch_dff_d_comp IS
END latch_dff_d_comp;

ARCHITECTURE latch_dff_d_comp OF latch_dff_d_comp IS
  COMPONENT dff_d
  PORT ( q   : OUT bit;
         d   : IN  bit;
         clk : IN  bit );
  END COMPONENT;

  COMPONENT latch_d
  PORT ( q   : OUT bit;
         d   : IN  bit;
         clk : IN  bit );
  END COMPONENT;

  COMPONENT latch_d_tb
  GENERIC (per : time := 10 ns );
  PORT ( d   : OUT   bit;
         clk : INOUT bit );
  END COMPONENT;
```

```
  SIGNAL d         : bit;
  SIGNAL clk       : bit;
  SIGNAL q_latch   : bit;
  SIGNAL q_dff     : bit;
  SIGNAL diferenta : bit;

  FOR TB    : latch_d_tb USE ENTITY work.latch_d_tb
              (latch_d_tb);
  FOR DUT_L : latch_d USE ENTITY work.latch_d(latch_d_beh);
  FOR DUT_F : dff_d USE ENTITY work.dff_d(dff_d_beh);

BEGIN

  TB : latch_d_tb GENERIC MAP (per => 40 ns)
                  PORT MAP ( d   => d,
                             clk => clk );

  DUT_L : latch_d PORT MAP ( q   => q_latch,
                             d   => d,
                             clk => clk );

  DUT_F : dff_d PORT MAP ( q   => q_dff,
                           d   => d,
                           clk => clk );

  PROCESS (q_latch, q_dff)
  BEGIN
    IF (q_latch = q_dff) THEN
      diferenta <= '0';
    ELSE
      diferenta <= '1';
    END IF;
  END PROCESS;
END latch_dff_d_comp;
```

3.5 Desfasurarea lucrarii

3.5.1 Simularea latch-ului

Pentru simulare parcurgeti urmatoarele etape:

Compilati fisierele:

- **latch_d.vhd** – descrierea entitatii
- **latch_d_beh.vhd** – descrierea comportamentala
- **latch_d_guard.vhd** – descrierea cu bloc cu garda
- **latch_d_tb.vhd** – descrierea generatorului de forme de unda
- **latch_d_comp.vhd** – descrierea comparatorului
- **latch_d_test.vhd** – descrierea mediului de testare

Simulati entitatea de nivel înalt **latch_d_test** asociata cu arhitectura **latch_d_test.**

Figura 3.4 prezinta compararea semnalelor generate de cele doua descrieri asociate latch-ului. Semnalul generat de comparator **error = 1** daca cele doua descrieri genereaza rezultate diferite.

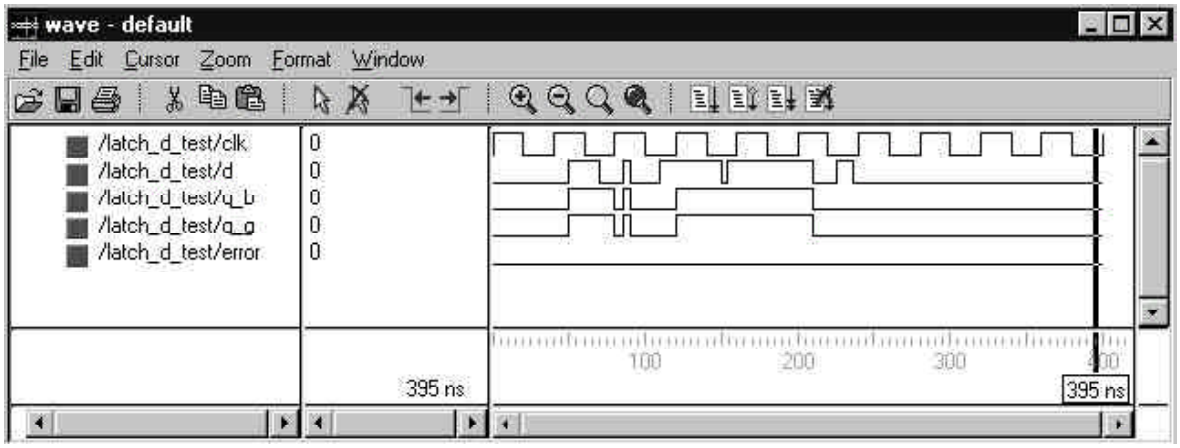


Figura 3.4: Forme de unda comparative pentru cele doua descrieri asociate latch-ului

3.5.2 Simularea bistabilului de tip D

Pentru a compara rezultatele generate de cele doua descrieri asociate bistabilului parcurgeti urmatoarele etape:

Compilati fisierele:

- **dff_d.vhd**
- **dff_d_beh.vhd**
- **dff_d_guard.vhd**
- **dff_d_tb.vhd**
- **dff_d_comp.vhd**
- **dff_d_test.vhd**

Simulati entitatea de nivel înalt **dff_d_test** asociata cu arhitectura **dff_d_test**.

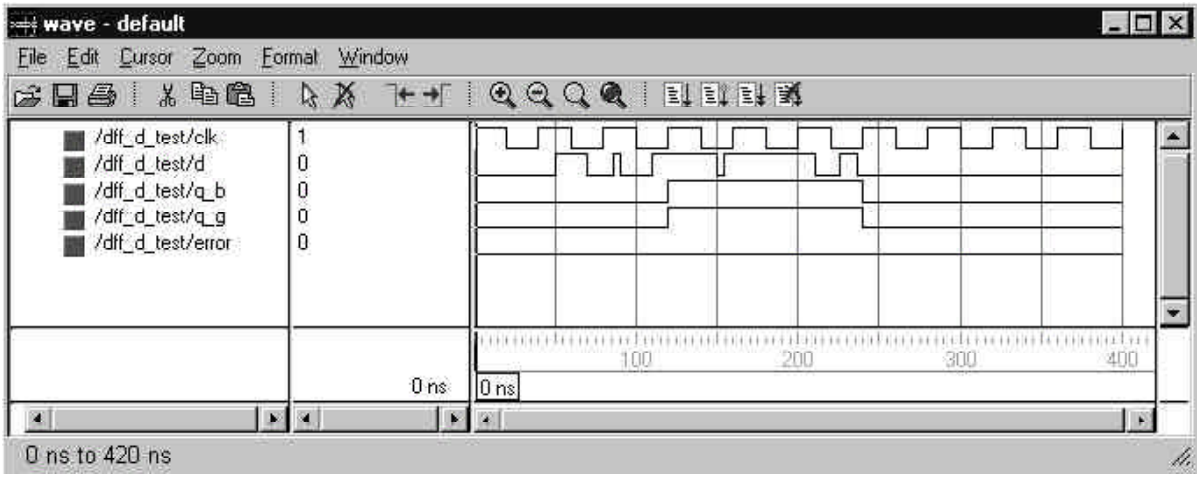


Figura 3.5: Forme de unda pentru cele doua descrieri asociate bistabilului de tip D

3.5.3. Comparatie latch-bistabil

Compilati fisierele:

- **latch_d.vhd**
- **latch_d_beh.vhd**
- **dff_d.vhd**
- **dff_d_beh.vhd**
- **latch_d_tb.vhd**
- **latch_dff_d_comp.vhd**

Simulati entitatea de nivel înalt **latch_dff_d_comp** si arhitectura **latch_dff_d_comp**.

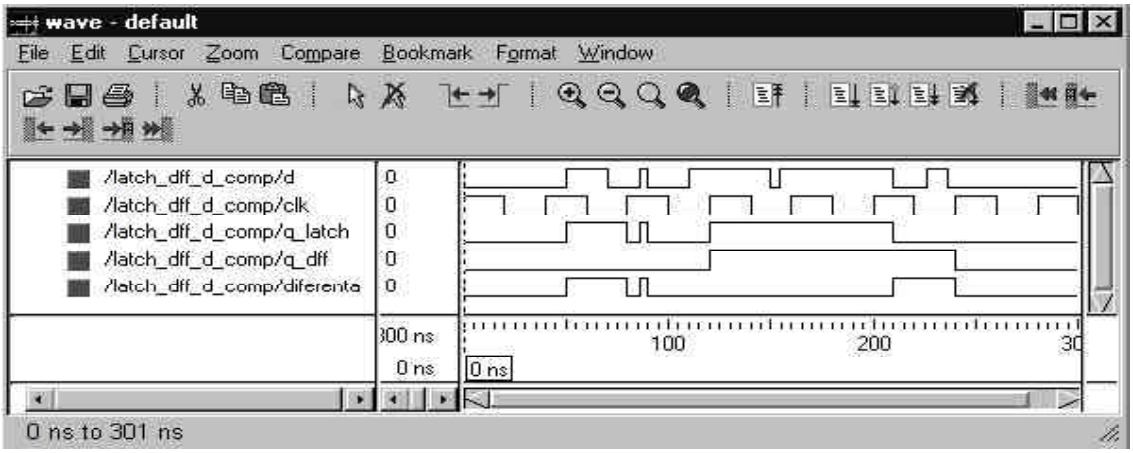


Figura 3.6: Forme de unda pentru comparatia latch-bistabil de tip D

3.6. Probleme

3.6.1 Modelati în VHDL un bistabil de tip D, activ pe frontul negativ al semnalului de ceas, cu intrari asincrone de **set** si **reset**.

3.6.2 Modelati un registru pe **n** biti cu intrare de **reset** sincrona. Intrarea **load**, activa pe 1, permite memorarea datelor de intrare (**data_in**) pe urmatorul front crescator al semnalului de ceas (**clk**). Pentru **load** inactiv, iesirea (**data_out**) ramâne neschimbata.

3.6.3 Modificati descrierea registrului din problema 3.6.2 astfel încât sa poata fi folosita ca registru de deplasare. Daca intrarea **shf_r = 1,** iesirea registrului este valoarea iesirii anterioare deplasata cu o pozitie spre dreapta.

Lucrarea

4 Modelarea circuitelor secventiale

4.1 Scopul lucrarii

· Modelarea registrelor si a numaratoarelor.
· Utilizarea pachetului **STD.TEXTIO** pentru scrierea în fisiere a rezultatelor monitorizate.
· Scrierea unor test-bench-uri complexe.

4.2 Modelarea unui registru de deplasare

Schema bloc a unui registru de deplasare (shift-are) este prezentata în figura 4.1.

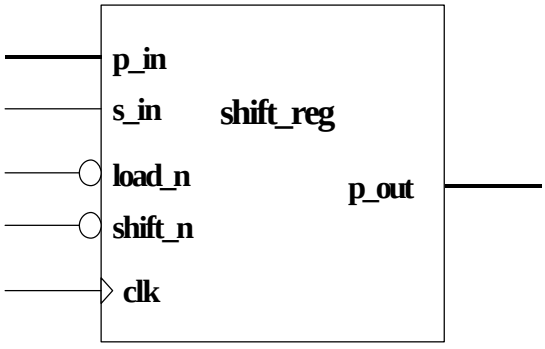


Figura 4.1 Schema bloc a unui registru de deplasare

nume_pin	dim	tip	functia
clk	1	intrare	ceas
s_in	1	intrare	intrare de deplasare seriala
p_in	8	intrare	intrare paralela
shift_n	1	intrare	validare deplasare, activ în 0
load_n	1	intrare	validare încarcare paralela, activ în 0
p_out	8	iesire	iesire paralela

Tabelul 4.1: Descrierea pinilor pentru registrul de deplasare din figura 4.1

load_n	shift_n	operatia
0	X	încarcare paralela
1	0	deplasare seriala
1	1	pastreaza starea

Tabelul 4.2: Descrierea functionarii registrului de deplasare din figura 4.1

Exemplul 4.1 prezinta descrierea entitatii registrului de deplasare. Porturile de intrare si iesire sunt de tipul **bit_vector** pe 8 biti.
Deplasarea la stânga se face prin concatenarea primilor 7 biti ai iesirii anterioare cu intrarea seriala.

```
--Exemplul 4.1: Entitatea registrului de deplasare
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY shf_reg_8 IS
PORT ( p_out   : OUT bit_vector(7 DOWNTO 0);
       p_in    : IN  bit_vector(7 DOWNTO 0);
       s_in    : IN  bit;
       load_n  : IN  bit;
       shift_n : IN  bit;
       clk     : IN  bit );
END shf_reg_8;


--Exemplul 4.2: Descrierea comportamentala a registrului de
--deplasare.
ARCHITECTURE shf_reg_8_beh OF shf_reg_8 IS
  SIGNAL reg     : bit_vector(7 DOWNTO 0);
BEGIN
  PROCESS(clk)
  BEGIN
```

```
    IF (clk'EVENT AND (clk = '1')) THEN
      IF (load_n = '0') THEN
        reg <= p_in;
      ELSIF (shift_n = '0') THEN
        reg <= reg(6 DOWNTO 0) & s_in;
      END IF;
    END IF;
  END PROCESS;
  p_out <= reg;
END shf_reg_8_beh;

--Exemplul 4.3: Descrierea cu bloc cu garda a registrului de
--deplasare.
ARCHITECTURE shf_reg_8_guard OF shf_reg_8 IS
  SIGNAL reg : bit_vector(7 DOWNTO 0);
BEGIN
  B1: BLOCK (NOT clk'STABLE AND (clk = '1'))
      BEGIN
        reg <= GUARDED p_in WHEN load_n = '0' ELSE
               reg(6 DOWNTO 0) & s_in WHEN shift_n = '0'ELSE
               reg;
      END BLOCK B1;
  p_out <= reg;
END shf_reg_8_guard;
```

4.3 Scrierea unui monitor complex

Testarea circuitelor complexe este foarte greoaie daca se face pe baza formelor de unda. Se recomanda scrierea unor monitoare care sa permita înscrierea datelor monitorizate într-un fisier text (figura 4.2).

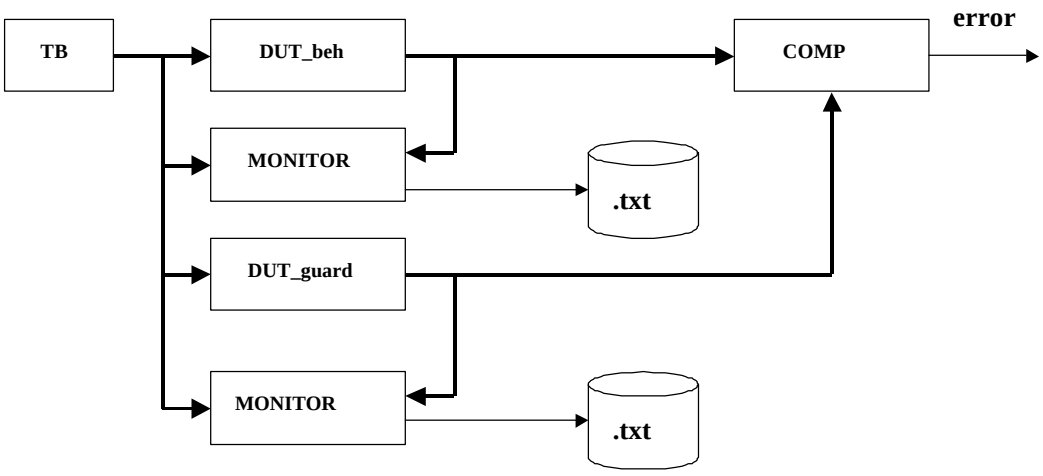


Figura 4.2 Mediu de testare ce permite compararea si înscrierea datelor monitorizate într-un fisier

În VHDL, interfata cu fisierele este asigurata de pachetul **TEXTIO**. Acest pachet defineste o serie de rutine utilizate pentru cititea si scrierea fisierelor ASCII. În exemplul 4.4 se foloseste si pachetul **io_utils** care completeaza pachetul **TEXTIO**.

```
--Exemplul 4.4: Descrierea monitorului care permite scrierea
--rezultatelor în fisier.
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
LIBRARY STD;
USE STD.TEXTIO.ALL;
USE work.io_utils.ALL;

ENTITY shf_reg_8_monitor IS
GENERIC ( eval_delay : time := 1 ns);
PORT ( p_out   : IN bit_vector(7 DOWNTO 0);
       p_in    : IN bit_vector(7 DOWNTO 0);
       s_in    : IN bit;
       load_n  : IN bit;
       shift_n : IN bit;
       clk     : IN bit );
END shf_reg_8_monitor;

ARCHITECTURE shf_reg_8_monitor OF shf_reg_8_monitor IS
  FILE out_file : TEXT IS OUT “shf_reg_8.txt”;
  SIGNAL reg    : bit_vector(7 DOWNTO 0);
BEGIN
  P1:PROCESS
    VARIABLE buff  : LINE;
  BEGIN
    WRITE_STRING (buff, “clk  p_in      s_in  load_n  shift_n
      p_out     error”);
    WRITELINE (out_file, buff);
    WAIT FOR 1 ns;
    WRITE_STRING (buff, “==================================”);
    WRITELINE (out_file, buff);
    WAIT;
  END PROCESS P1;


P2: PROCESS( clk )
    VARIABLE sel   : bit_vector(1 DOWNTO 0);
  BEGIN
    sel := load_n & shift_n;
    IF (NOT clk‘STABLE AND (clk = ‘1‘)) THEN
      CASE sel IS
         WHEN “00” => reg <= p_in;
```

```
        WHEN “01” => reg <= p_in;
        WHEN “10” => reg <= reg(6 DOWNTO 0) & s_in;
        WHEN “11” => reg <= reg;
      END CASE;
    END IF;
  END PROCESS P2;


 P3: PROCESS( p_out )
   VARIABLE error : bit := ‘0‘;
   VARIABLE buff  : LINE;
 BEGIN
   IF (reg = p_out) THEN
      error := ‘0‘;
    ELSE
      error := ‘1‘;
    END IF;

    WRITE (buff, clk);
    WRITE_STRING (buff, “    “);
    WRITE (buff, p_in);
    WRITE_STRING (buff, “  “);
    WRITE (buff, s_in);
    WRITE_STRING (buff, “     “);
    WRITE (buff, load_n);
    WRITE_STRING (buff, “       “);
    WRITE (buff, shift_n);
    WRITE_STRING (buff, “        “);
    WRITE (buff, p_out);
    WRITE_STRING (buff, “  “);
    WRITE (buff, error);
    WRITELINE (out_file, buff);
  END PROCESS P3;

END shf_reg_8_monitor;
```

Procesul **P1** înscrie header-ul fisierului, iar procesul **P2** este o descriere comportamentala a circuitului monitorizat (registru de deplasare). Compararea rezultatelor si înscrierea rezultatelor în fisier se face în procesul **P3**. În exemplul 4.5 este ilustrat o parte din fisierul generat de monitor.

```
--Exemplul 4.5: Fisierul generat de monitor.

clk  p_in      s_in  load_n  shift_n  p_out     error
================================================
0    00000000  0     0       1        00000000  0
1    00000000  0     0       1        00000000  0
0    00000001  0     0       1        00000000  0
1    00000001  0     0       1        00000001  0
0    00000010  0     0       1        00000001  0
…
```

4.4 Modelarea unui numarator binar

Numaratorul din figura 4.3 are pinii descrisi în tabelul 4.3. Tabelul 4.4 descrie functionarea acestuia.

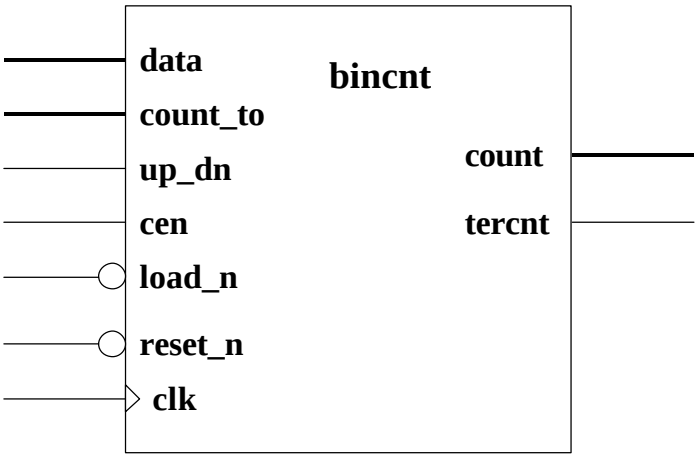


Figura 4.3: Schema bloc a unui numarator binar.

Nume pin	Dimensiune	Tip	Functia
data	8	intrare	date de intrare
count_to	8	intrare	intrare de comparare pentru numarare
up_dn	1	intrare	sensul de numarare
load_n	1	intrare	validare încarcare, activ în 0
cen	1	intrare	validare numarare, activ în 1
clk	1	intrare	ceas
reset_n	1	intrare	reset sincron, activ în 0
count	8	iesire	date de iesire
tercnt	1	iesire	terminare numarare, activ în 1

Tabelul 4.3: Descriera pinilor pentru numaratorul din figura 4.3

reset_n	load	cen	up_dn	Operatia
0	X	X	X	reset sincron
1	0	X	X	încarcare paralela
1	1	0	X	pastreaza starea
1	1	1	0	numarare în sens crescator
1	1	1	1	numarare în sens descrescator

Tabelu 4.4: Descrierea functionarii numaratorului din figura 4.3

O posibila descriere comportamentala pentru numaratorul binar este prezentata în continuare:

```
--Exemplul 4.6: Descrierea comportamentala a numaratorului
--binar.
ARCHITECTURE bicnt_dcto_8_beh OF bicnt_dcto_8 IS
  SIGNAL reg : bit_vector(7 DOWNTO 0);
BEGIN
  PROCESS (clk)
  BEGIN
    IF ( clk'EVENT AND (clk = '1')) THEN
      IF ( reset_n = '0' ) THEN
        reg <= "00000000";
      ELSIF ( load_n = '0' ) THEN
        reg <= data;
      ELSIF ( cen = '1' ) THEN
        IF ( up_dn = '0' ) THEN
          reg <= reg - "00000001";
        ELSE
          reg <= reg + "00000001";
        END IF;
      END IF;

  IF (reg = count_to) THEN
        tercnt <= '1';
      ELSE
        tercnt <= '0';
      END IF;
    END IF;
  END PROCESS;
  count <= reg;
END bicnt_dcto_8_beh;
```

În descrierea cu bloc cu garda se foloseste o singura atribuire concurenta cu conditii multiple.

```
--Exemplul 4.8: Descrierea cu bloc cu garda a numaratorului
--binar.
ARCHITECTURE bicnt_dcto_8_guard OF bicnt_dcto_8 IS
  SIGNAL reg : bit_vector(7 DOWNTO 0);
BEGIN
  B1: BLOCK ((NOT clk'STABLE) AND (clk = '1'))
  BEGIN
    reg    <= GUARDED "00000000" WHEN (reset_n = '0') ELSE
              data WHEN ((reset_n = '1') AND
 (load_n = '0')) ELSE
              reg - "00000001" WHEN ((reset_n = '1') AND
                                     (load_n = '1') AND
                                     (cen = '1') AND
                                     (up_dn = '0')) ELSE
              reg + "00000001" WHEN ((reset_n = '1') AND
                                           (load_n = '1') AND
                                     (cen = '1') AND
                                     (up_dn = '1')) ELSE
            reg;
    tercnt <= GUARDED '1' WHEN reg = count_to ELSE '0';
  END BLOCK B1;
  count <= reg;
END bicnt_dcto_8_guard;
```

4.5 Desfasurarea lucrarii

4.5.1 Simularea registrului de deplasare

Compilati fisierele:

- **io_utils.vhd**
- **bit_arith.vhd**
- **shf_reg_8.vhd**
- **shf_reg_8_beh.vhd**
- **shf_reg_8_guard.vhd**
- **shf_reg_8_tb.vhd**
- **shf_reg_8_monitor.vhd**
- **shf_reg_8_comp.vhd**
- **shf_reg_8_test.vhd**

Simulati entitatea de test **shf_reg_8_test** asociata cu arhitectura **shf_reg_8_test**.

4.5.2 Simularea numaratorului binar

Deoarece în descrierea numamarorului s-au folosit operatorii aritmetici pentru tipul de data **bit_vector**, iar rezultatul monitorizarii se înscrie în fisier, mai întâi trebuie compilate:

- **bit_arith.vhd** – pachet pentru operatii cu **bit_vector**
- **io_utils.vhd** – pachet cu functii pentru interfata cu fisiere.

Blocurile necesare pentru simularea celor doua descrieri asociate numaratorului binar (comportamentala si cu bloc cu garda) sunt descrise în fisierele urmatoare:

- **bincnt_dct_8.vhd**
- **bincnt_dct_8_beh.vhd**
- **bincnt_dct_8_guard.vhd**
- **bincnt_dct_8_monitor.vhd**
- **bincnt_dct_8_comp.vhd**
- **bincnt_dct_8_tb.vhd**
- **bincnt_dct_8_test.vhd**.

4.6 Probleme

4.6.1 Folosind modelul de numarator binar cu indicator dinamic **count_to** (paragraful 4.4) realizati un numarator modulo 8.

4.6.2 Scrieti un model comportamental pentru un numarator. Numaratorul decrementeaza pe frontul pozitiv al semnalului de ceas.

4.6.3 Modelati comportamental un numarator pe 4 biti activ pe frontul pozitiv al semnalului de ceas, cu facilitati de reset sincron. Declaratia entitatii este:

```
entity counter is
    port (clk_n, load_en: in  std_logic;
            d: in  std_logic_vector(3 downto 0);
            q: out std_logic_vector(3 downto 0));
end counter;
```

Lucrarea

5 Modelarea automatelor

5.1 Scopul lucrarii

· Modelarea automatelor cu reset sincron sau asincron.
· Compararea automatelor.

5.2 Modelarea automatelor cu reset sincron

Automatul Mealy cu întârziere cu graful de tranzitie din figura 5.1 are intrarile **red**, **green** si **blue**. Pe baza combinatiei culorilor de intrare, automatul determina daca se formeaza o noua culoare sau nu (iesirea **newColour**). Semnificatia combinatiilor înscrise pe sagetile de tranzitie este urmatoarea: **red green blue / newColour**.

```
--Exemplul 5.1 Entitatea asociata automatului din figura 5.1
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY mealy IS
PORT ( red       : IN  bit;
       green     : IN  bit;
       blue      : IN  bit;
       clk       : IN bit;
       reset     : IN bit;
       newColour : OUT bit );
END mealy;
```

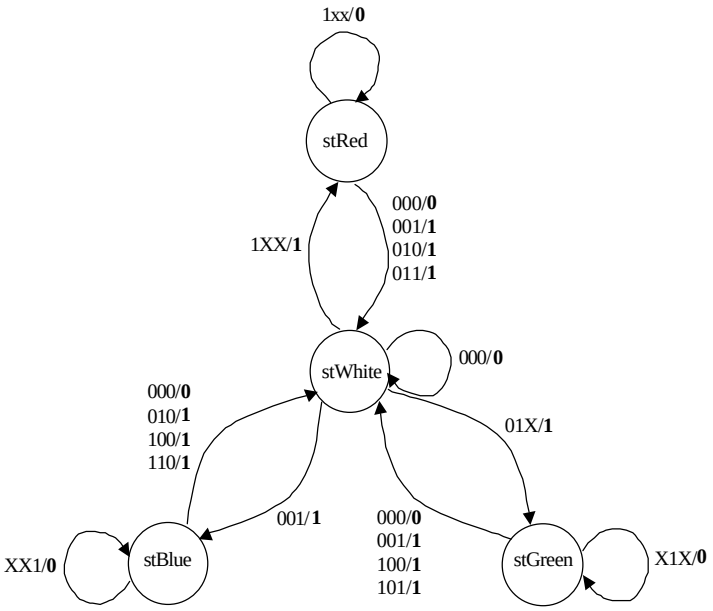


Figura 5.1 Graful de tranzitie pentru un automat Mealy

În exemplul 5.2 se prezinta o posibila descriere VHDL pentru automatul cu graful de tranzitie din figura 5.1. Activarea semnalului sincron de reset determina trecerea automatului în starea initiala **stWhite**.

```
--Exemplul 5.2: Descrierea automatului Mealy cu reset sincron
ARCHITECTURE mealy_syn OF mealy IS
  TYPE state IS (stWhite, stRed, stGreen, stBlue);
  SIGNAL currentState : state;
  SIGNAL nextState    : state;
  SIGNAL colour       : bit;
BEGIN
  CLC: PROCESS ( currentState, reset, red, green, blue ) BEGIN
    CASE currentState IS
      WHEN stWhite => IF ( reset = '1' ) THEN
                        colour    <= '0';
                        nextState <= stWhite;
                      ELSIF ( red   = '0' AND
                              green = '0' AND
                              blue  = '0' ) THEN
                        colour    <= '0';
                        nextState <= stWhite;
                      ELSIF ( red   = '1' ) THEN
                        colour    <= '1';
                        nextState <= stRed;
                      ELSIF ( red   = '0' AND
                              green = '1' ) THEN
                        colour    <= '1';
                        nextState <= stGreen;
```

```
                    ELSIF ( red   = '0' AND
                            green = '0' AND
                            blue  = '1' ) THEN
                      colour    <= '1';
                      nextState <= stBlue;
                    END IF;

WHEN stRed =>       IF ( reset = '1' ) THEN
                      colour    <= '0';
                      nextState <= stWhite;
                    ELSIF ( red   = '1' ) THEN
                      colour    <= '0';
                      nextState <= stRed;
                    ELSIF ( red   = '0' AND
                            green = '0' AND
                            blue  = '0' ) THEN
                      colour    <= '0';
                      nextState <= stWhite;
                    ELSE
                      colour    <= '1';
                      nextState <= stWhite;
                    END IF;

WHEN stGreen =>     IF ( reset = '1' ) THEN
                      colour    <= '0';
                      nextState <= stWhite;
                    ELSIF ( green   = '1' ) THEN
                      colour    <= '0';
                      nextState <= stGreen;
                    ELSIF ( red   = '0' AND
                            green = '0' AND
                            blue  = '0' ) THEN
                      colour    <= '0';
                      nextState <= stWhite;
                    ELSE
                      colour    <= '1';
                      nextState <= stWhite;
                    END IF;

WHEN stBlue =>      IF ( reset = '1' ) THEN
                      colour    <= '0';
                      nextState <= stWhite;
                    ELSIF ( blue   = '1' ) THEN
                      colour    <= '0';
                      nextState <= stBlue;
                    ELSIF ( red   = '0' AND
                            green = '0' AND
                            blue  = '0' ) THEN
```

```
                        colour    <= '0';
                        nextState <= stWhite;
                      ELSE
                        colour    <= '1';
                        nextState <= stWhite;
                      END IF;
    END CASE;
  END PROCESS CLC;

  REG: PROCESS ( clk ) BEGIN
    IF( ( NOT clk'STABLE ) AND ( clk = '1' ) ) THEN
      currentState <= nextState;
      newColour    <= colour;
    END IF;
  END PROCESS CLS;

END mealy_syn;
```

Descrierea automatului contine doua procese:
· **CLC** corespunzator circuitului logic combinational pentru calculul starii viitoare si al iesirilor.
· **REG** corespunzator registrului de stare si registrului de întârziere a iesirilor.

În urma sintezei rezulta structura din figura 5.2. Se observa ca **reset**-ul sincron actioneaza asupra logicii combinationale.

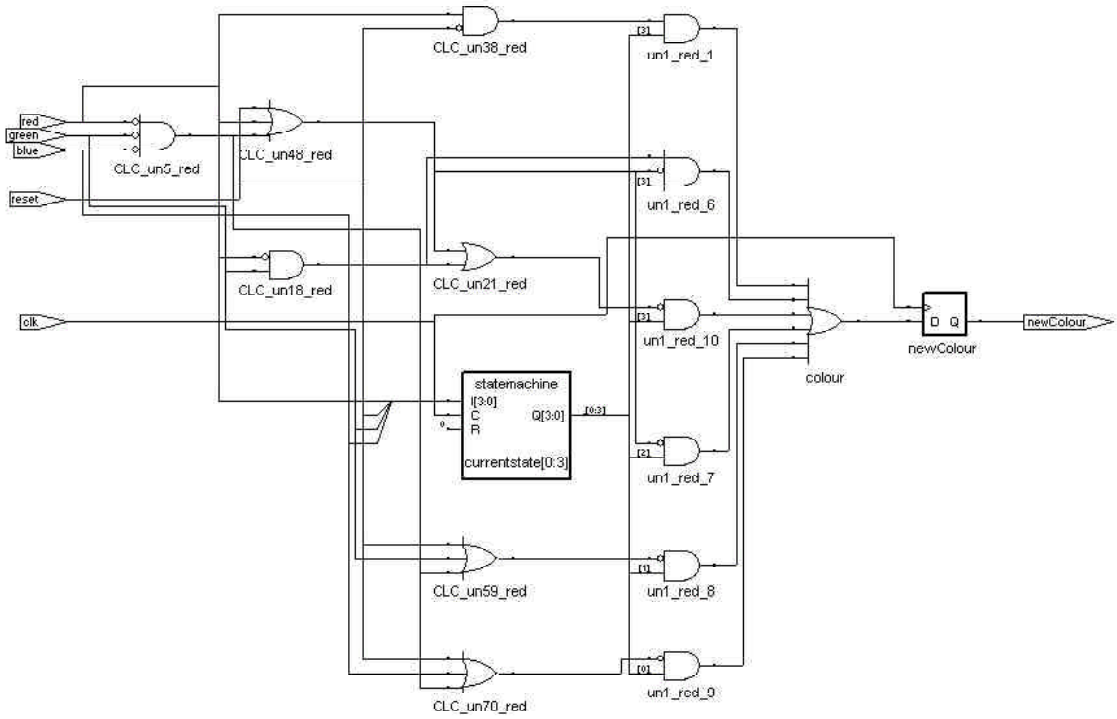


Figura 5.2: Structura rezultata în urma sintezei automatului Mealy cu reset sincron

5.3 Modelarea automatelor cu reset asincron

Modelarea **reset**-ului asincron presupune modificarea registrului de stare. Reset-ul asincron, actioneaza direct asupra registrului de stare.

```
--Exemplul 5.3: Descrierea automatului Mealy cu reset asincron
ARCHITECTURE mealy_asyn OF mealy IS

  TYPE state IS (stWhite, stRed, stGreen, stBlue);
  SIGNAL currentState : state;
  SIGNAL nextState    : state;
  SIGNAL colour       : bit;

BEGIN

  CLC: PROCESS ( currentState, red, green, blue ) BEGIN
    CASE currentState IS
      WHEN stWhite => IF ( red   = '0' AND
                           green = '0' AND
                           blue  = '0' ) THEN
                        colour    <= '0';
                        nextState <= stWhite;
                      ELSIF ( red   = '1' ) THEN
                        colour    <= '1';
                        nextState <= stRed;
                      ELSIF ( red   = '0' AND
                              green = '1' ) THEN
                        colour    <= '1';
                        nextState <= stGreen;
                      ELSIF ( red   = '0' AND
                              green = '0' AND
                              blue  = '1' ) THEN
                        colour    <= '1';
                        nextState <= stBlue;
                      END IF;

      WHEN stRed =>   IF ( red   = '1' ) THEN
                        colour    <= '0';
                        nextState <= stRed;
                      ELSIF ( red   = '0' AND
                              green = '0' AND
                              blue  = '0' ) THEN
                        colour    <= '0';
                        nextState <= stWhite;
```

```
                    ELSE
                      colour <= '1';
                      nextState <= stWhite;
                    END IF;

      WHEN stGreen => IF ( green   = '1' ) THEN
                      Colour    <= '0';
                      nextState <= stGreen;
                    ELSIF ( red   = '0' AND
                            green = '0' AND
                            blue  = '0' ) THEN
                      colour    <= '0';
                      nextState <= stWhite;
                    ELSE
                      Colour    <= '1';
                      nextState <= stWhite;
                    END IF;
      WHEN stBlue =>  IF ( blue   = '1' ) THEN
                      Colour    <= '0';
                      nextState <= stBlue;
                    ELSIF ( red   = '0' AND
                            green = '0' AND
                            blue  = '0' ) THEN
                      colour    <= '0';
                      nextState <= stWhite;
                    ELSE
                      Colour    <= '1';
                      nextState <= stWhite;
                    END IF;
    END CASE;
  END PROCESS CLC;

  CLS: PROCESS ( clk, reset ) BEGIN
    IF ( reset = '1' ) THEN
      newColour    <= '0';
      currentState <= stWhite;
    ELSIF(( NOT clk'STABLE ) AND ( clk = '1' )) THEN
      currentState <= nextState;
      newColour    <= colour;
    END IF;
  END PROCESS CLS;

END mealy_asyn;
```

Structura automatului cu **reset** asincron este prezentat în figura 5.3. Reset-ul asincron determina folosirea bistabilelor cu intrari asincrone de **reset**.

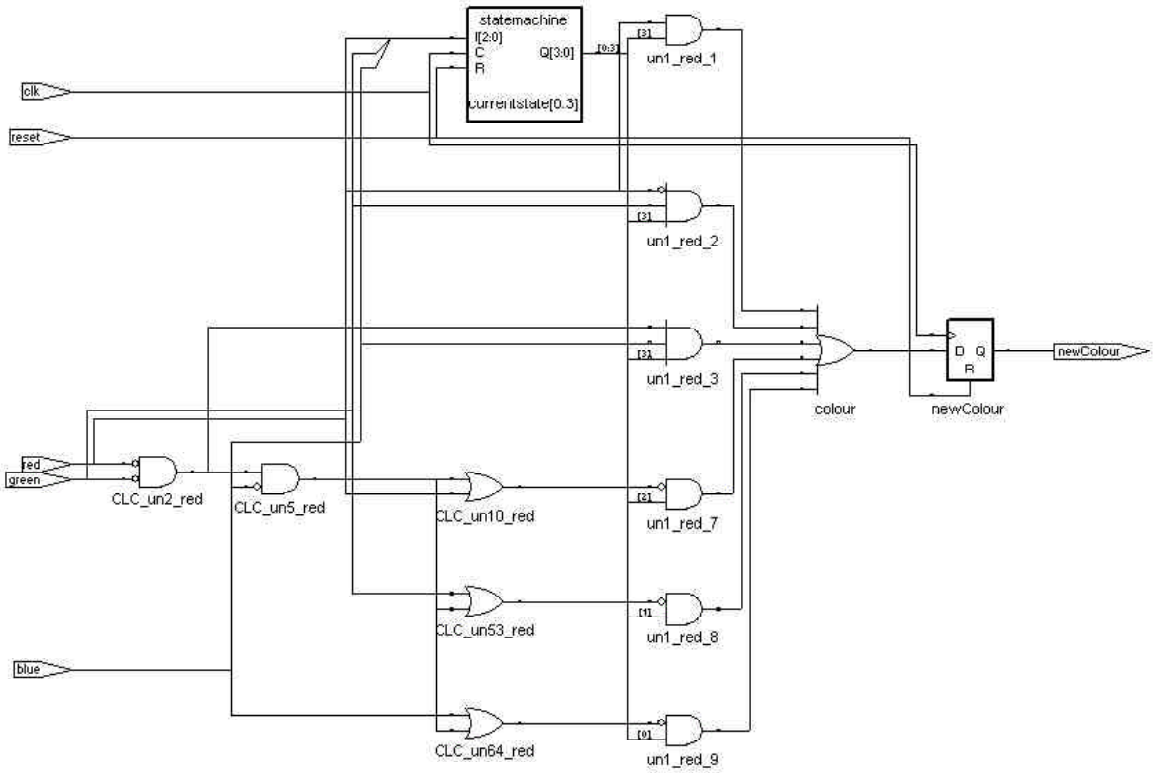


Figura 5.3: Structura rezultata în urma sintezei automatului Mealy cu reset asincron

5.4 Desfasurarea lucrarii

5.4.1 Simularea automatului cu reset sincron

Compilati fisierele:

- **mealy.vhd**
- **mealy_syn.vhd**
- **mealy_tb.vhd**
- **mealy_syn_test.vhd**

Simulati entitate **mealy_syn_test** asociata cu arhitectura **mealy_syn_test**.

Verificati, pe formele de unda, daca automatul functioneaza conform grafului de tranzitie.

5.4.1 Simularea automatului cu reset asincron

Compilati fisierele:

- **mealy.vhd**
- **mealy_asyn.vhd**
- **mealy_tb.vhd**
- **mealy_asyn_test.vhd**

·Simulati entitatea **mealy_asyn_test** asociata cu **mealy_asyn_test**.

·Verificati, pe formele de unda, daca automatul functioneaza conform grafului de tranzitie.

5.4.1 Compararea automatelor cu reset sincron si asincron

Compilati fisierele:

- **mealy.vhd**
- **mealy_syn.vhd**
- **mealy_asyn.vhd**
- **mealy_tb.vhd**
- **mealy_syn_asyn_test.vhd**

Simulati entitatea **mealy_syn_asyn_test** cu arhitectura **mealy_syn_asyn_test**.

În urma comparatiei celor doua tipuri de automate (figura 5.4) se observa ca starea curenta a automatului asincron devine starea initiala (**stWhite**) odata cu activarea semnalului de **reset**. Starea curenta pentru automatul cu reset sincron se modifica doar pe urmatorul front activ al semnalului de ceas.

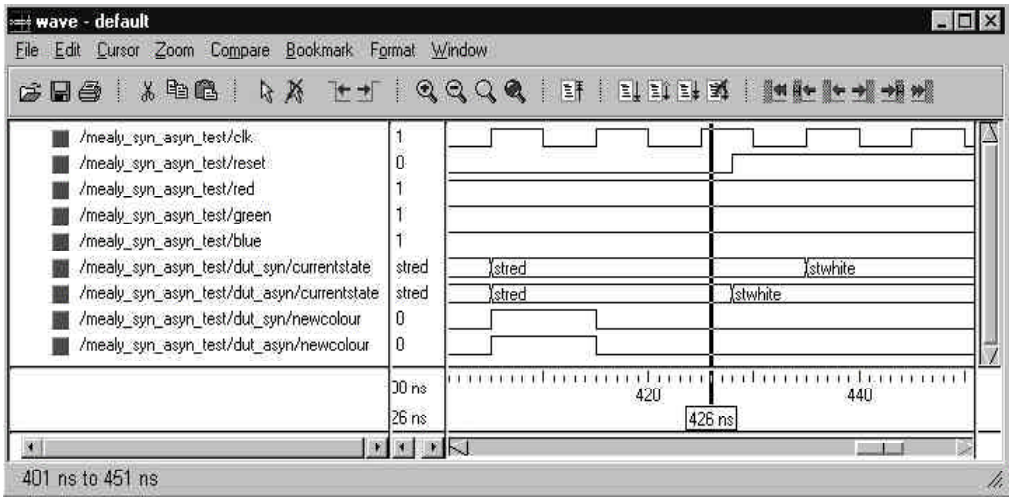


Figura 5.4: Forme de unda pentru compararea celor doua tipuri de automate

5.5 Probleme

5.5.1 Modelati un controller pentru un motor pas-cu-pas care are schema bloc din figura 5.7 si functionarea descrisa de tabelul 5.2.

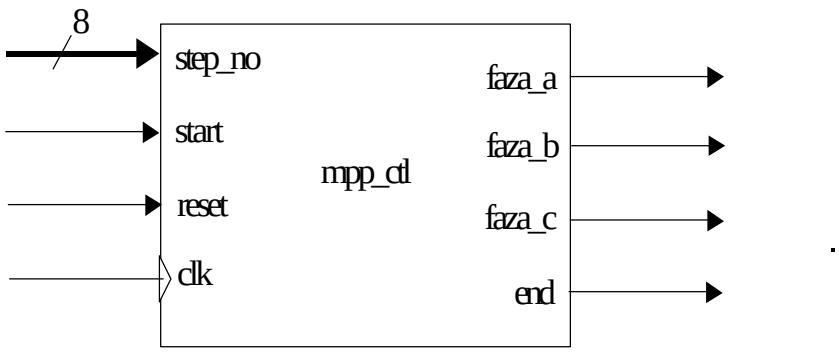


Figura 5.5: Schema bloc a controller-ului pentru motorul pas cu pas

reset	start	step_no	operatia
1	X	X	reset
0	0	X	pastreaza starea
0	1	0…255	efectueaza numarul de pasi specificati

Tabelul 5.2: Descrierea functionarii controller-ului

Efectuarea unui pas de catre MPP presupune aplicarea urmatoarei secvente pe cele trei faze ale motorului:

faza_a	faza_b	faza_c
1	0	0
0	1	0
0	0	1

PARTEA a II-a

ALLIANCE

Lucrarea

6 Mediul de proiectare ALLIANCE

6.1 Introducere

Pachetul Alliance este un set complet de utilitare EDA (Electronic Design Automation) pentru specificarea, proiectarea si validarea circuitelor digitale VLSI. Pe lânga aceste unelte, pachetul Alliance mai include si un set de biblioteci de celule, de la celule elementare folosite de utilitarele de rutare si plasare automata si pâna la blocuri întregi de circuit. Acest pachet este în domeniul public putând fi gasit la adresa *ftp:// asim.lip6.fr/pub/alliance/.*

6.1.1 Independenta de proces

Pachetul Alliance ofera suport pentru implementarea circuitelor integrate în tehnologie CMOS de celule standard (de la specificatii pâna la layout). Celulele bibliotecilor se bazeaza pe o abordare simbolica a layout-ului oferind o independenta completa de parametrii tehnologici ai procesului de implementare în siliciu. Trecerea de la un layout simbolic la un layout real se poate face complet automatizat folosind ca parametru un fisier tehnologic propriu implementarii în siliciu.

6.1.2 Portabilitate software

Pachetul Alliance a fost proiectat pentru a rula pe o serie de platforme de calcul, singurele necesitati fiind un compilator C si sistemul de operare Unix. Pentru aplicatiile

grafice este necesara si biblioteca XWindows. Pachetul a fost testat si ruleaza în prezent pe o gama larga de platforme de calcul, de la calculatoare compatibile PC si pâna la statii Sparc, Sun sau DEC. Începând cu versiunea 4.0, Alliance este disponibil si pentru Windows, folosind emulatorul de SO Linux denumit Cygwin.

6.1.3 Modularitate

Fiecare unealta de proiectare din pachetul Alliance poate opera atât în cadrul mediului Alliance, cât si ca program executabil de sine statator. Din aceste motive utilitarele de proiectare Alliance permit folosirea diverselor standarde de descriere VLSI: SPICE, EDIF, VHDL, CIF, GDS II, etc.

6.2 Metologia de proiectare

Prin metologie de proiectare (Design Flow) se întelege un set de operatii secventiale care trebuie executate pentru a realiza un layout de circuit VLSI. Pachetul Alliance suporta modalitatea de proiectare top-down (modelul Mead-Conway), formata din 4 parti distincte descrise în continuare.

6.2.1 Descrierea si simularea reprezentarii comportamentale

Primul pas al unui proiect consta în realizarea descrierii comportamentale a unui circuit, folosind în acest scop primitive ale limbajului VHDL. Subsetul VHDL cu care opereaza Alliance se numeste VBE si este destul de restrictiv, neoferind o prea mare libertate de miscare utilizatorului. Cea mai restrictiva particularitate a subsetului VHDL o constituie absenta proceselor, ele fiind suplinite de blocurile cu garda.
Modelul circuitului ce trebuie proiectat este vazut ca un *“black box”*, o cutie neagra, caracterizata de un set de intrari, un set de iesiri si un set de functii logice. Fisierele continând descrieri comportamentale au extensia .VBE.

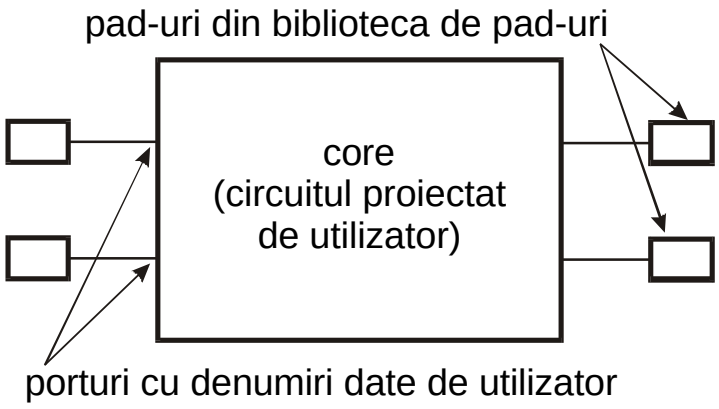


Figura 6.1: Structura unui circuit VLSI

Simularea descrierilor VBE se poate face cu ajutorul unui fisier de pattern-uri în care este descris comportamentul intrarilor. Acest fisier poate fi generat atât manual, cât si semi-automat prin folosirea unui utilitar care opereaza pe baza unor primitive C.
Fisierul de pattern-uri, împreuna cu descrieirea VBE, sunt parametri de intrare ai simulatorului **asimut** care are ca iesire un fisier de pattern-uri care descrie comportamentul iesirilor. Acelasi utilitar **asimut** poate fi folosit si la verificarea sintaxei fisierului VBE.

6.2.2 Descrierea si simularea reprezentarii structurale

Reprezentarea structurala este calea de legatura între domeniul comportamental si cel fizic. Aici sunt specificate conexiunile între module, dar nici un fel de parametri fizici. Ierarhia este cea cunoscuta: entitate, arhitectura, componente, semnale.
Simularea descrierii structurale se face cu acelasi simulator si cu acelasi fisier de pattern-uri ca si la descrierea comportamentala, pattern-urile rezultante trebuind sa fie identice.

6.2.3 Sinteza fizica

Netlist-ul generat din descrierea structurala este plasat si rutat prin folosirea unui router de celule standard, celule care se gasesc în bibliotecile Alliance. Uzual, unul sau mai multe module formeaza asa numitul “core”, care în urmatorul pas este conectat cu exteriorul (adica la pini) prin intermediul unor pad-uri. Exista pad-uri de intrare, de iesire si bidirectionale. Operatia este executata automat de utilitarul **ring,** pe baza unor

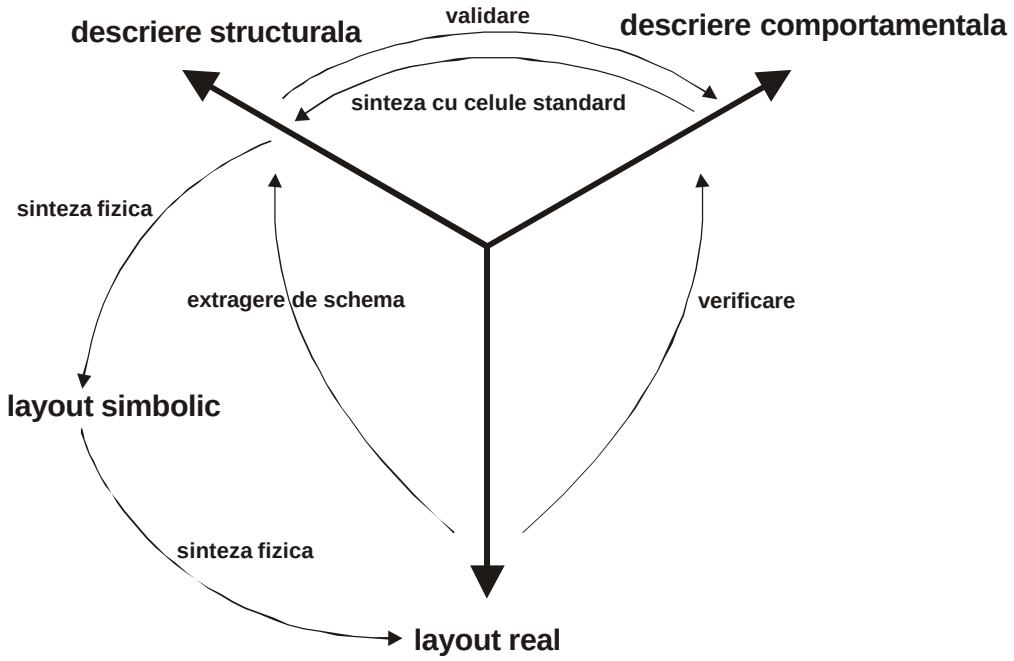


Figura 6.2: Procesul de proiectare al unui circuit VLSI

parametri de orientare si a unei biblioteci de pad-uri. Rezultatul legarii pad-urilor de core-ul circuitului constituie layout-ul simbolic, în care sunt specificate atât pozitiile relative, cât si conexiunile între elementele structurale ale circuitului.
Toate operatiile executate pâna în acest punct sunt independente de constrângeri fizice si de parametri tehnologici. Trecerea la un layout real, propriu unei anumite tehnologii si unui proces de fabricatie, se face pe baza unui fisier tehnologic, folosit ca parametru al utilitarului **s2r.** Layout-ul real contine pozitiile reale, absolute, ale componentelor circuitului în stratul de siliciu .

6.2.4 Verificare

În cadrul proiectarii VLSI procesul verificarii este cel putin la fel de important ca si proiectarea propriu-zisa. Din aceste motive, mediul Alliance dispune de o serie de utilitare de verificare ce pot fi folosite în fiecare etapa a procesului de proiectare.
Verificarea la nivelul layout-ului se poate face fie prin extragerea de schema din layout - LVS (Layout Versus Schematic), fie prin extragerea de descriere comportamentala. Descrierile comportamentale sau structurale nu vor fi identice cu cele initiale, dar pot fi verificate cu acelasi set de pattern-uri sau cu un utilitar special, **proof**.

Procedeele specifice de realizare a unui proiect prin proiectare top-down constituie trecerea de la o etapa la alta a procesului de proiectare. Transformarea reprezentarii comportamentale în reprezentare structurala se realizeaza prin sinteza. Trecerea de la reprezentarea structurala la un layout real se face prin etapa de **sinteza fizica**, care, în cazul Alliance include o etapa intermediara de **layout simbolic**. Din reprezentarea geometrica (layout real) se poate trece înapoi la o reprezentare comportamentala prin etapa de verificare.
Alte transformari inverse sunt:
- extragerea de schema, care desemneaza trecerea de la layout real la reprezentare structurala, si
- validarea, care desemneaza trecerea de la reprezentarea structurala la reprezentarea comportamentala.

6.3 Arhitectura unui chip

Proiectarea unui circuit VLSI este bazata pe o abordare ierarhizata si incrementala. Pentru a putea face fata complexitatii unui circuit, cât si din motive de consum de putere, fiecare chip este compus din doua componente majore:
- core (miezul) circuitului care include întreaga sa structura functionala;
- pad-urile care constituie interfata core-ului cu exteriorul circuitului, adica legatura dintre core si pinii chip-ului.

Se mai poate efectua o partitionare si a core-ului în cale de control si cale de date. Aceasa segregare se regaseste în toate circuitele de o complexitate mai ridicata, pornind de la simple unitati ALU si pâna la microprocesoare. Scopul segregarii îl constituie posibilitatile îmbunatatite de proiectare si depanare prin independenta celor doua unitati de baza. În mod uzual, calea de control contine circuitele secventiale de comanda (automate) iar calea de control contine circuitele combinationale (porti, registri, bistabile, numaratoare etc.).

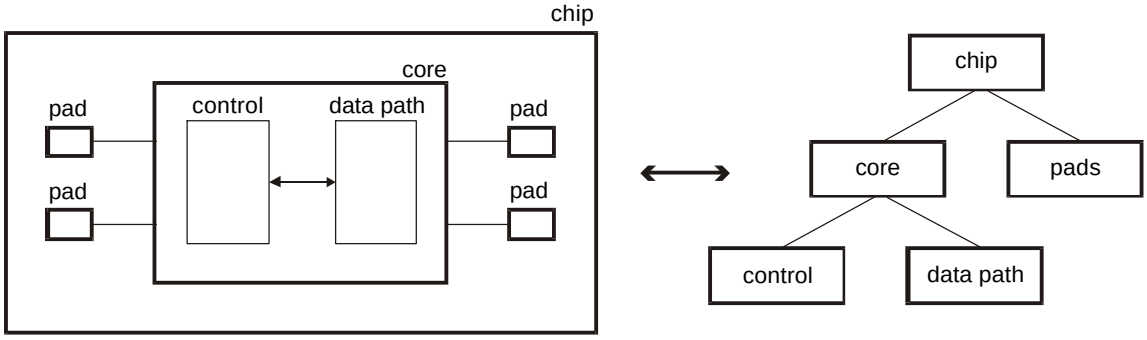


Figura 6.3: Arhitectura unui chip

Pachetul Alliance se preteaza foarte bine la abordarea top-down a proiectarii unui circuit VLSI partitionat dupa modelul descris anterior. Astfel, în mod uzual calea de control si cea de date sunt descrise comportamental, pentru a fi ulterior sintetizate structural. La acest nivel, tot printr-un procedeu structural, calea de control si cea de date sunt legate pentru a crea core-ul. Acesta este la rândul lui legat, tot structural, de pad-uri, formând astfel structura chip-ului. Descrierea în acest punct este tot structurala, fiind urmata de trecerea la un layout simbolic si ulterior la unul real.

6.4 Pachetul Alliance

Cu toate ca pachetul Alliance nu dispune de un mediu cu interfata grafica de tip Windows, el este usor de folosit, una dintre cele mai importante caracteristici ale sale constituind-o structura comuna interna a datelor pentru a reprezenta cele trei abordari de baza ale unui circuit VLSI:

- abordarea comportamentala;
- abordarea structurala;
- abordarea fizica.

Independenta de procesul tehnologic pâna la nivelul de layout simbolic este realizata folosind pentru toate componentele existente în biblioteci o abordare simbolica cu grila fixa. Implementarea fizica a unui layout a fost facuta cu succes de catre creatorii acestui program pentru tehnologii în siliciu de 2 µm, 1.5 µm, 1.2 µm, 1.0 µm, 0.7 µm si chiar 0.5 µm. Tehnologiile comerciale actuale sunt: 0.18 µm, 0.15 µm si 0.13 µm.

6.4.1 Utilitare

asimut este un simulator VHDL, subsetul VHDL suportat permitând atât descrieri structurale cât si comportamentale (fara întârzieri). Descrierile comportamentale (fisiere cu extensia VBE), cât si cele structurale (fisiere cu extensia VST) pot fi verificate sintactic si logic folosind ca parametru de intrare un fisier de pattern-uri în care se descriu stimulii de intrare. Fisierul de pattern-uri rezultant poate fi ulterior verificat si chiar vizualizat cu ajutorul utilitarului **xpat** (sub interfata X).

syf este un sintetizator de automate. Mai exact, **syf** asociaza valori starilor folosite pentru descrierea functionarii automatului si încearca sa minimizeze functiile booleene de iesire si de tranzitie. Fisierul de intrare al utilitarului **syf** (cu extensia .FSM) este o descriere a automatului folosind un subset VHDL dedicat care include totusi si specificatia de proces. Fisierul de iesire va contine o descriere comportamentala a automatului folosind acelasi subset VHDL ca si simulatorul logic **asimut**.

scmap este un utilitar de sinteza logica care are ca iesire o descriere structurala (netlist) generata pe baza unei biblioteci de celule standard. Este folosit acelasi subset VHDL ca la descrierea comportamentala. Se pot folosi si alte componente decât cele provenind din biblioteca de celule standard, acestea trebuind însa sa fie descrise atât comporamental cât si în termeni de layout simbolic.

genlib este un generator de descrieri structurale pe baza folosirii unui set consistent de primitive C, dând posibilitatea utilizatorului de a descrie circuite VLSI pe baza porturilor, semnalelor si a instantierilor de componente.

scr este un utilitar care se ocupa de plasarea si rutarea descrierilor structurale. Intrarea utilitarului o constituie o descriere structurala (fisier .VST), iesirea fiind un layout simbolic ierarhic (canalele sunt instantiate) sau flattened (structuri din care s-au eliminat granitele dintre module). Extensia fisierului de iesire este în ambele cazuri .AP.

ring este un utilitar de rutare dedicat operatiunii finale de rutare a pad-urilor împreuna cu core-ul unui circuit. Utilitarul se ocupa automat de problema plasamentului optim al pad-urilor, cât si de alimentarea lor cu semnale de ceas, alimentare si masa.

druc este un utilitar de verificare a regulilor de proiectare (Design Rule Check – DRC). Intrarea o constituie un layout simbolic, iar verificarea se face pe baza setului de reguli pentru proiectarea simbolica.

s2r este un utilitar folosit în ultima faza a procesului de generare a unui layout, facând trecerea de la descrierea ierarhica simbolica a unui layout, la un layout fizic adaptat cerintelor tehnologice. Procesul de translatie implica operatii complexe de adaptare a straturilor în siliciu, aliniere si uniformizare a structurii. Intrarea utilitarului este un layout simbolic si un fisier de tehnologie în care sunt listati toti parametrii tehnologici necesari tranzitiei la layout real. Fisierul de iesire este standardizat fie în formatul CIF fie în formatul GDS II.

druc este un utilitar de verificare a regulilor de proiectare (Design Rule Check – DRC). Intrarea o constituie un layout simbolic, iar verificarea se face pe baza setului de reguli pentru proiectarea simbolica.

lynx este un extractor de schema dintr-un layout simbolic sau real. Intrarea este fie un fisier continând un layout simbolic, fie unul continând un layout real. Iesirea este un netlist continând si capacitatile parazite.

yagle este un dezasamblor pentru circuite CMOS, generând o descriere comportamentala pornind de la un netlist de tranzistoare. Aceasta transformare inversa are scop de verificare, descrierea comportamentala generata putând fi comparata din punct de vedere logic cu cea initiala prin folosirea unui alt utilitar, **proof**.

6.5 Desfasurarea lucrarii

- Parcurgeti structura de directoare ale pachetului Alliance si vizualizati diversele formate de fisiere (.VBE, . VST, .FSM, .AP).

- Verificati functionarea utilitarelor descrise în paragraful 6.4.1. Vizualizati paginile de manual si listele de parametri ale acestora.

Lucrarea

7 Proiectarea unui sumator

7.1 Introducere

Aceasta lucrare prezinta proiectarea unui sumator de un bit cu generare a transportului. Proiectul are ca scop crearea unui core de circuit. Legarea pad-urilor la core se va face în lucrarea 9. Circuitul de adunare va fi proiectat în doua etape:

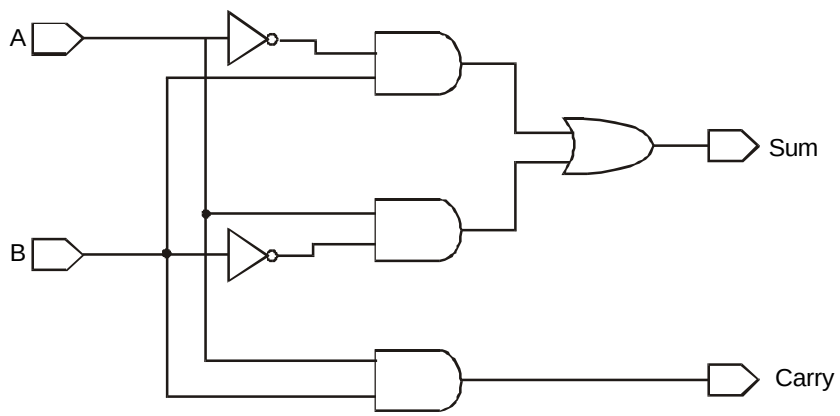
- proiectarea si sintetizarea unui circuit de adunare fara transport la intrare (semisumator);
- legarea structurala si ulterior sinteza a doua circuite de adunare pentru a obtine un sumator complet de 1 bit (full adder).

7.2 Semisumatorul

Prima parte a circuitului de adunare o constituie descrierea si sintetizarea unui semi-sumator, având schema si tabelul de adevar din figura 7.1.

Operatii

1. Descrierea comportamentala a componentei Half Adder folosind subsetul VHDL Alliance (halfadder.vbe).
2. Verificarea sintaxei descrierii prin trecerea fisierului prin utilitarul **asimut**.
3. Scrierea unui fisier de vectori de test (halfadder.pat).
4. Simularea descrierii comportamentale cu ajutorul fisierului de pattern-uri si salvarea rezultatului simularii într-un nou fisier de pattern-uri (r1.pat).
5. Generarea unei descrieri structurale (halfadders.vst) folosind biblioteca de celule standard cu utilitarul **scmap**.
6. Plasarea si rutarea descrierii structurale cu utilitarul **scr** (halfadderl.ap).
7. Vizualizarea layout-ului cu utilitarul **graal**.



Tabel de adevar

A	B	Sum	Carry
0	1	0	1
0	1	1	0
1	0	1	0
1	1	0	1

Ecuatii

$Sum=\overline{A}B+\overline{B}A$

$Carry=AB$

Figura 7.1: Schema, tabelul de adevar si ecuatiile semisumatorului

Înainte de a începe proiectarea propriu-zisa, trebuie verificate si, la nevoie, setate variabilele de mediu fie direct în fisierul de configuratie *alc_env.sh*, fie cu comanda *export* din Linux.
Variabila MBK_CATA_LIB marcheaza calea (sau caile) în care se gasesc celulele de baza care se folosesc la instantierile din proiect, la trecerea de la o descriere comportamentala la una structurala si mai departe.
Variabilele MBK_IN_LO si MBK_OUT_LO specifica formatul de intrare si de iesire pentru utilitarele generatoare si utilizatoare a layout-ului simbolic.
Variabilele MBK_IN_PH si MBK_OUT_PH sunt responsabile de setarea formatului de fisier folosit de layout-ul real.

```
MBK_CATA_LIB=:$TOP/cells/sclib:$TOP/cells/padlib;
MBK_IN_LO=vst; MBK_OUT_LO=vst;
MBK_IN_PH=ap;  MBK_OUT_PH=ap;
```

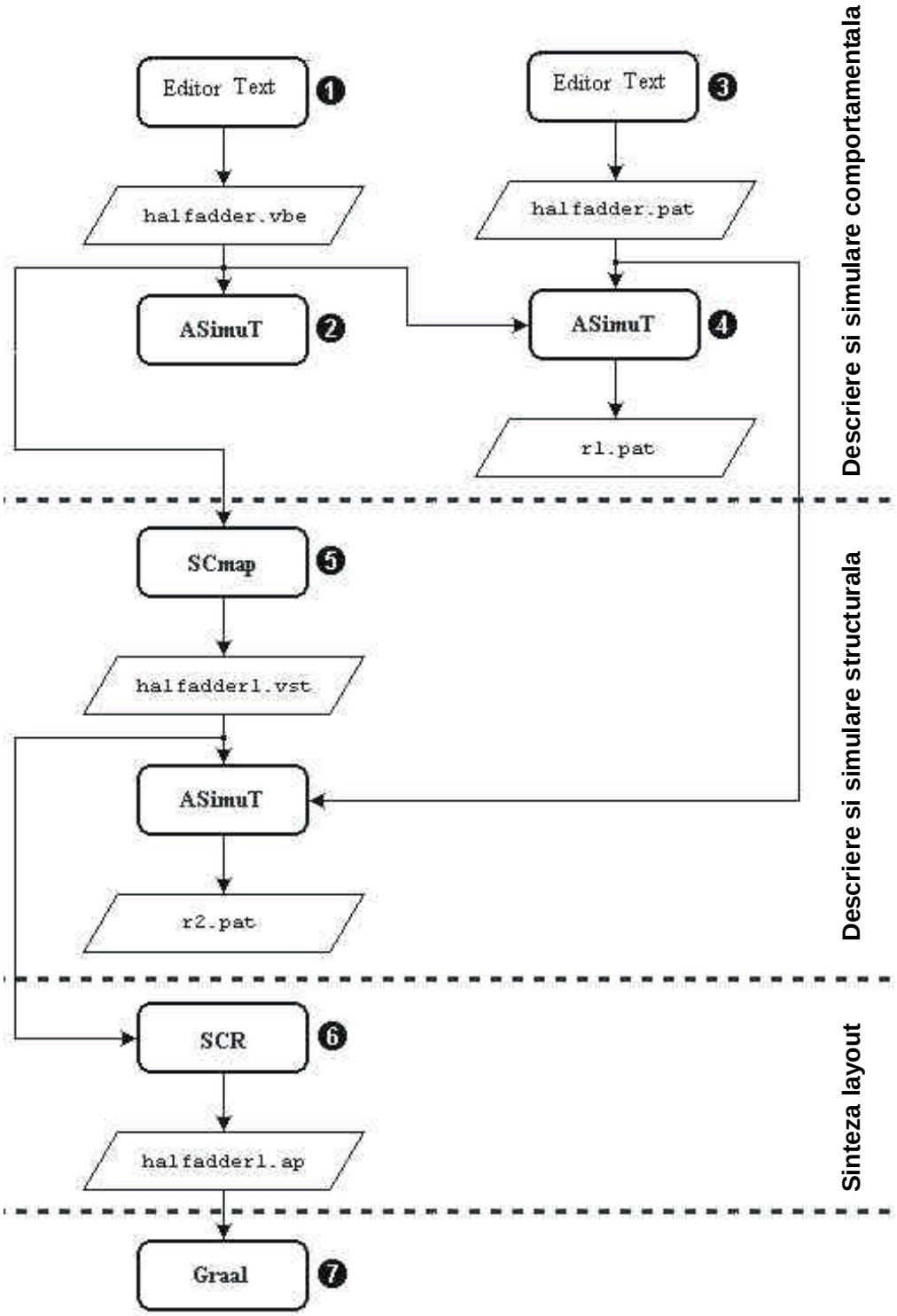


Figura 7.2: Operatiile necesare pentru proiectarea semi sumatorului

În mod normal, variabilele sunt setate implicit pentru a putea permite o sinteza completa pornind de la o descriere comportamentala. Pentru modificarea valorilor se poate folosi comanda Linux *export* urmata de numele variabilei si de valoarea ei. Pentru mai multe detalii despre variabilele de mediu se pot consulta paginile de manual ale fiecarei variabile în parte.

7.2.1 Descrierea comportamentala

Comportamentul componentei Half Adder va fi descris în fisierul *halfadder.vbe* folosind subsetul VHDL Alliance. Nu sunt permise declaratii secventiale ci doar concurente. Nu pot fi folosite procese, ci doar blocuri cu garda.

```
_************************************************
ENTITY halfadder IS
PORT ( A, B, Vdd, Vss: IN BIT; Sum, Carry: OUT BIT );
END halfadder;
_************************************************

ARCHITECTURE halfadder_data_flow OF halfadder IS

SIGNAL A_bar, B_bar: BIT;
BEGIN
      A_bar <= NOT A;
      B_bar <= NOT B;
      Sum   <= ( A_bar AND B ) OR ( A AND B_bar );
      Carry <= A AND B;

END halfadder_data_flow;
```

Fisierul, odata scris, trebuie verificat atât din punct de vedere sintactic, cât si al corectitudinii comportamentale. Ambele verificari pot fi facute cu **asimut**, utilitarul de simulare logica a descrierilor hardware.

```
asimut –b –c halfadder

-b – utilitarul asteapta o descriere comportamentala (*.vbe)
-c – optiune de compilare
halfadder – numele fisierului comportamental (fara extensia
VBE).
```

7.2.2 Scrierea fisierului cu vectori de test

Daca pasul anterior nu a generat aparitia pe ecran a unor mesaje de eroare, atunci descrierea comportamentala poate fi simulata cu un fisier de vectori de test (pattern-uri), fiind necesara scrierea unui fisier *halfadder.pat*. Fisierul de pattern-uri este compus din doua parti:

- declaratiile de semnale continând lista de intrari, iesiri,
- semnale interne si registri, urmate de descrierea acestor semnale. Intrarilor li se atribuie anumite valori, iesirile fiind ulterior completate automat de utilitarul **asimut**.

```
-- lista de intrari/iesiri:
in a;;
in b;;
in vdd;;
in vss;;
out sum;;
out carry;;
begin

-- descrierea pattern-urilor :

--    a b v v  s  c
--        d s  u  a
--        d s  m  r

pat0: 0 0 1 0 ?* ?* ;
pat1: 0 1 1 0 ?* ?* ;
pat2: 1 0 1 0 ?* ?* ;
pat3: 1 1 1 0 ?* ?* ;

end;
```

Fisierul de vectori de text *halfadder.pat* poate fi vizualizat ca forme de unda cu utilitarul **xpat** care trebuie lansat sub interfata X.

7.2.3 Simularea descrierii comportamentale

Odata scrise descrierea comportamentala (*halfadder.vbe*) si fisierul cu vectorii de test (*halfadder.pat*), se poate trece la simularea descrierii.

```
asimut -b halfadder halfadder r1

-b – descriere comportamentala
primul halfadder – fisierul VBE
al doilea halfadder – fisierul PAT
r1 – fisier PAT rezultand.
```

Fisierul *r1.pat* poate fi vizualizat din nou cu utilitarul **xpat** sau si în mod text. Astfel, se poate observa ca semnalele de iesire au fost completate cu valori logice ce au înlocuit simbolurile *.
În fisierul de pattern-uri de intrare se pot specifica pe lânga valorile penru intrari, si valori pentru iesiri, precedate, la fel ca si asteriscul, de un semn de întrebare. Daca valorile de iesire specificate apriori coincid cu cele calculate prin simulare, utilitarul **asimut** nu afiseaza nici un mesaj de avertizare. În caz contrar, **asimut** va semnala neconcordanta.

În fisierul initial de pattern-uri fiecare linie din lista de semnale de intrare si iesire se termina cu cel putin un simbol ";". Fiecare ";" suplimentar va introduce un spatiu liber între componentele vectorului de test generat de trecerea fisierului cu pattern-uri prin **asimut**, facilitate utila pentru marirea lizibilitatii codului.

7.2.4 Generarea descrierii structurale

Descrierea comportamentala din fisierul *halfadder.vbe* poate fi convertita într-o descriere structurala folosind utilitarul **scmap**. Procedeul mai este cunoscut si sub numele de **"mapare"** cu celule standard (standard cell mapping). Având în vedere ca circuit de adunare prezentat este simplu, nu se vor face optimizari si se va folosi biblioteca de celule standard.

```
scmap halfadder halfadders

halfadder – fisierul VBE (halfadder.vbe)
halfadders – descrierea structurala generata (halfadders.vst)
```

Fisierul VST poate fi vizualizat atât în format text, cât si în format grafic. În format text, vor putea fi observate structuri de porti logice interconectate prin asocieri de porturi si semnale auxiliare generate automat. Aceste porti logice sunt apelate din biblioteca de celule standard (sclib) si sunt instantiate ca si componente ale entitatii halfadder.

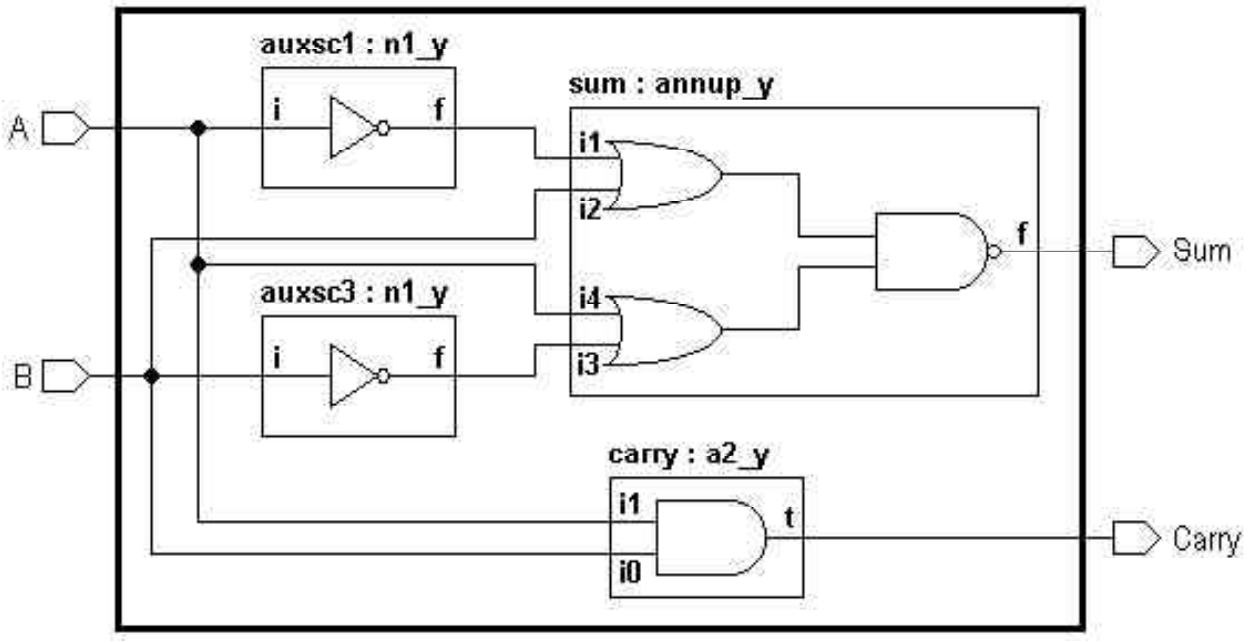


Figura 7.3: Structura componentei halfadder

Varianta grafica a descrierii structurale poate fi vizualizata cu utilitarul **xsch**. Componentele sunt prezentate ca niste cutii negre, caracterizate de numele sub care pot fi gasite în biblioteca de celule de baza si de o lista de intrari si iesiri. Semnalele auxiliare sunt etichetate. Structura este prezentata în figura 7.3.
În aceasta etapa se poate verifica daca descrierea structurala a fost generata corect, prin simularea ei cu acelasi fisier de pattern-uri ca si în cazul descrierii comportamentale. În mod normal fisierele de iesire sunt identice.

```
asimut halfadder halfadder r2

halfadders – fisierul VST
halfadder – fisierul PAT
r2 – fisier PAT rezultand.
```

Parametrul –b lipseste la apelul **asimut** pentru ca nu este vorba de o descriere comportamentala (behavioural) ci de una structurala. Din acest motiv, **asimut** cauta un fisier cu extensia .VST si nu unul cu extensia .VBE.

7.2.5 Plasarea si rutarea descrierii structurale

Pentru operatiile de plasare si rutare a core-ului circuitului descris structural este folosit utilitarul **scr** (Standard Cell Router). Acesta genereaza un fisier cu extensia. AP.

```
scr –p –r –i 1000 halfadder

-p – apelarea procesului de rutare automata
-r – apelarea procesului de plasare automata
-i 1000 – nr. de iteratii
halfadder – fisierul VST
```

7.2.6 Vizualizarea layout-ului

Layout-ul generat de **scr** poate fi vizualizat cu utilitarul **graal** (interfata X). Din meniul File – Open se încarca fisierul halfadders.ap si pe ecran va aparea structura din figura 7.4, care prezinta doar celulele standard folosite.
Pentru a vizualiza toate detaliile, pâna la nivel de tranzistor MOS, trebuie selectat meniul Tools-Flat si tras cu mouse-ul o fereastra în jurul întregii figuri. Pe ecran vor aparea toate straturile (layer) existente în structura circuitului. Pentru navigare se pot folosi si direct tastele cursorului, tasta Z (zoom in) si M (mooz – zoom out).
Cele doua trasee mai groase din mijlocul layout-ului sunt traseele VDD si VSS care distribuie tensiunea de alimentare la toate tranzistoarele din schema. Intrarile si iesirile circuitului sunt plasate întotdeauna la marginea layout-ului, pentru a permite o ulterioara conectare a pad-urilor.

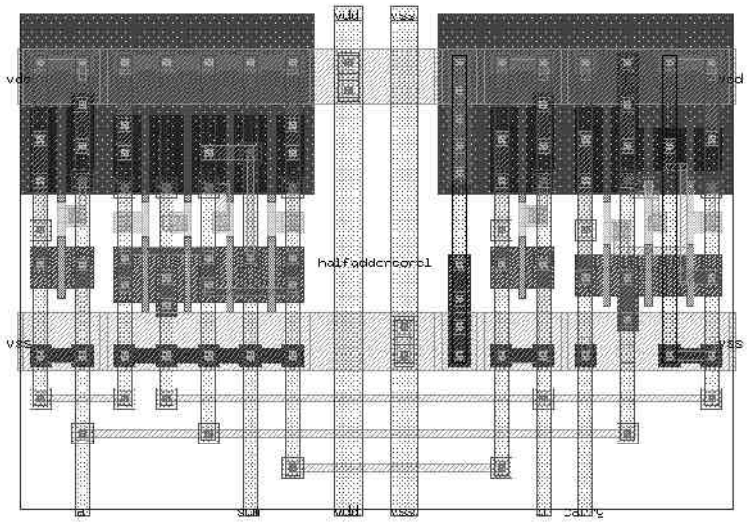


Figura 7.4: Layout-ul componentei halfadder

7.3 Sumatorul complet

Odata completat procesul de proiectare a componentei Half Adder, se poate face cel de-al doilea pas în realizarea unui sumator complet cu intrare si iesire de transport (carry). Schema ne-optimizata a acestuia este prezentata în figura 7.5, împreuna cu tabelul de adevar si ecuatiile logice.

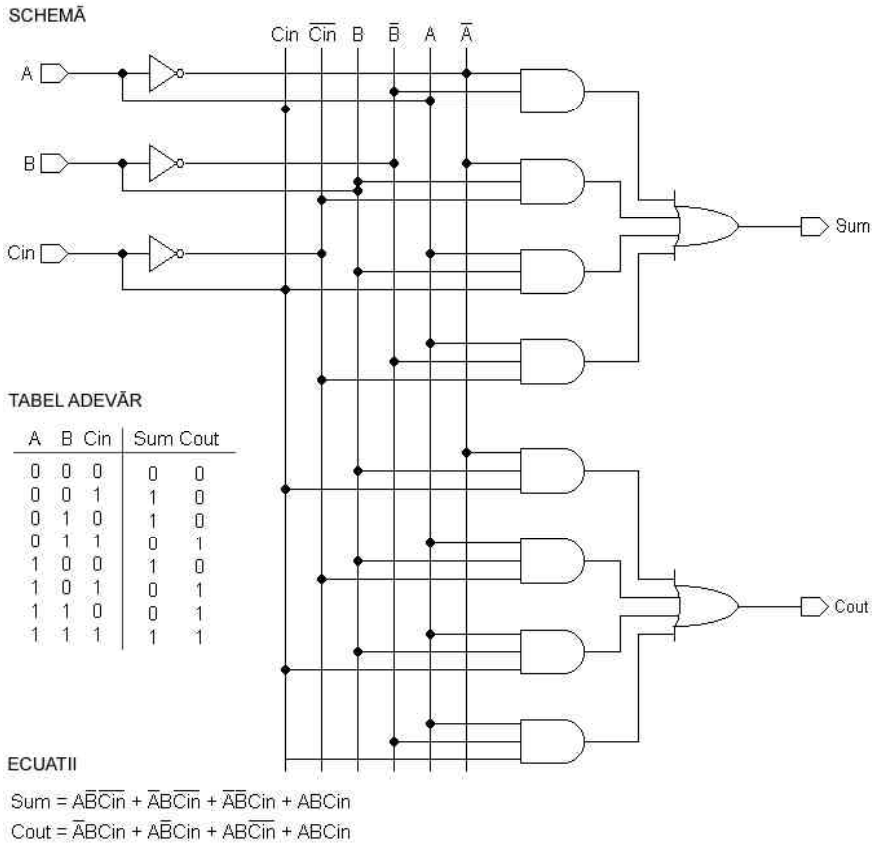


Figura 7.5: Sumatorul complet de 1 bit

Sumatorul complet va fi descris structural cu doua componente de tip semi-sumator (figura 7.6). Prin folosirea a doua asemenea componente, împreuna cu o poarta SAU cu doua intrari din biblioteca de celule standard, se poate sintetiza un sumator complet fara a fi nevoie de o noua descriere comportamentala.

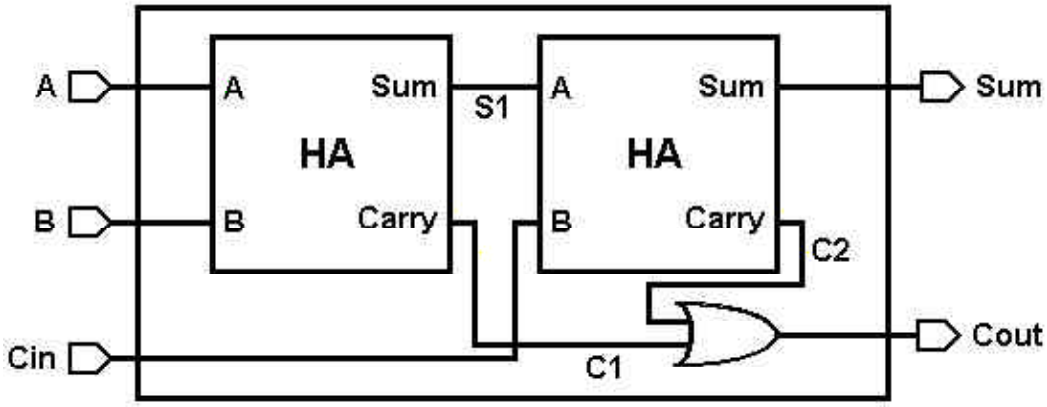


Figura 7.6: Sumatorul complet de 1 bit descris structural cu doua semisumatoare

Pasii care trebuie urmati în realizarea layout-ului sunt:

1.Descrierea structurala a circuitului fulladder (*fulladder.vst*) folosind componenta halfadder.
2. Verificarea sintactica a fisierului.
3. Scrierea unui nou fisier de vectori de test (*fulladder.pat*).
4. Simularea descrierii structurale folosind fisierul de pattern-uri.
5. Plasarea si rutarea descrierii structurale cu utilitarul **scr** (*fulladder.ap*).
6. Verificarea regulilor de proiectare cu utilitarul **druc**.
7. Vizualizarea layout-ului simbolic cu utilitarul **graa**l.
8. Generarea layout-ului real cu utilitarul **s2r** (*fulladder.cif*).
9. Vizualizarea layout-ului real cu utilitarul **dreal**.

7.3.1 Descrierea structurala a sumatorului complet

Fisierul cu descrierea structurala se va numi *fulladder.vst.*

```
--Porturi externe
ENTITY fulladder IS
PORT ( A, B, Cin, Vdd, Vss: IN BIT;
Sum, Cout: OUT BIT );
END fulladder;
_-*****************************************
```

```
--Structura interna

ARCHITECTURE fulladder_structural OF fulladder IS

--declararea componentelor utilizate

COMPONENT halfadders

PORT ( A: IN BIT; B: IN BIT;
Sum: OUT BIT;
Carry: OUT BIT;
Vdd: IN BIT;
Vss: IN BIT );

END COMPONENT;

COMPONENT o2_y

PORT ( i0: IN BIT; i1: IN BIT;
t: OUT BIT;
vdd: IN BIT;
vss: IN BIT );
END COMPONENT;

--declararea semnalelor interne

SIGNAL c1, s1, c2: BIT;

BEGIN

--instantierea componentelor si legarea lor

ha1: halfadderl
PORT MAP (    Vss => Vss,
Vdd => Vdd,
A => A,
B => B,
Sum => s1,
Carry => c1 );

ha2: halfadderl
PORT MAP (    Vss => Vss,
Vdd => Vdd,
A => s1,
B => Cin,
Sum => Sum,
Carry => c2 );
```

```
or1:o2_y PORT MAP(vss => Vss,
vdd => Vdd,
i0 => c1,
i1 => c2,
t => Cout );

END fulladder_structural;
```

Subsetul VHDL Alliance pentru descrieri structurale este ceva mai restrâns, înglobând totusi toate elementele necesare pentru a putea crea descrieri structurale complexe. Sunt instantiate doua semisumatoare, intrarile si iesirile acestora fiind similare celor descrise comportamental în fisierul *halfadder.vbe* în prima parte a laboratorului. Componenta **o2_y** este o poarta SAU cu doua intrari si se gaseste deja descrisa comportamental în biblioteca de celule standard.
Dupa instantiere, cele trei componente sunt legate cu ajutorul semnalelor auxiliare conform schemei din figura 7.6 pentru a forma o structura de sumator complet de 1 bit.

7.3.2 Verificarea sintactica

Cu acelasi utilitar **asimut** se poate si de aceasta data verifica corectitudinea descrierii din punct de vedere sintactic.

```
asimut –c fulladder

-c – optiunea de compilare
fulladder – fisierul VST
```

Optiunea **–b** lipseste pentru ca este vorba de descriere structurala.

7.3.3 Scrierea fisierului cu vectori de test

Fisierul cu vectori de test (*fulladder.pat*) poate fi adaptat din fisierul *halfadder.pat*, inserând o intrare suplimentara de tip bit, numita Cin (prescurtarea de la Carry In). Evident, si descrierea vectorilor de test trebuie suplimentata în fiecare linie cu intrarea amintita.

```
--lista de intrari/iesiri:
in a;;
in b;;
in vdd;;
in vss;;
out sum;;
out carry;;
```

```
begin

-- descrierea pattern-urilor
--    a b v v c  s  c
--        d s i  u  o
--        d s n  m  u
--                  t

pat0: 0 0 1 0 0 ?* ?* ;
pat1: 0 1 1 0 0 ?* ?* ;
pat2: 1 0 1 0 1 ?* ?* ;
pat3: 1 1 1 0 1 ?* ?* ;

end;
```

7.3.4 Simularea descrierii structurale folosind fisierul de pattern-uri

Fisierul *fulladder.pat*, împreuna cu descrierea structurala din fisierul *fulladder.vst* trebuie trecute prin **asimut** pentru a genera un fisier de pattern-uri care ulterior poate fi verificat pentru corectitudine.

```
asimut fulladder fulladder r3

primul fulladder – fisierul VST
al doilea fulladder – fisierul PAT
r3 – fisierul PAT generat
```

7.3.5 Plasarea si rutarea descrierii structurale

Utilitarul **scr** genereaza pe baza unui netlist (descriere structurala *fulladder.vst*) un layout simbolic.

```
scr -p -r -l 2 -i 1000 fulladder

-p - apelarea procesului de rutare automata
-r – apelarea procesului de plasare automata
-l 2 – numarul de rânduri pe care va fi realizat circuitul
-i 1000 – numarul de iteratii în care sunt realizare cele doua
operatii
```

Daca optiunea **-l** lipseste, utilitarul **scr** încearca sa genereze un layout în care raportul lungime/latime este cât mai apropiat de 1, adica un layout în forma de patrat. Optiunea permite însa generarea de layout-uri pe lungime, utile mai ales în cazul în care se doreste realizarea unor circuite reale care au pini doar pe doua din laturi.

7.3.6 Verificarea regulilor de proiectare

Utilitarul **druc** (design rule checker) este conceput pentru a verifica daca un layout a fost sintetizat conform cu regulile circuitelor VLSI, care specifica anumite constrângeri în ceea ce priveste dimensiunile traseelor de masa, suprapunerile de layer-e etc.

```
druc fulladder
```

Druc nu are parametri si în caz de eroare genereaza doua fisiere:

- *fulladder.drc* în care sunt listate erorile existente, si
- *fulladder_drc.cif* în care sunt specificate structurile geometrice (dreptunghiurile) ce contin erori.

7.3.7 Vizualizarea layout-ului simbolic

Graal este utilitarul cu care layout-ul simbolic poate fi vizualizat si, la nevoie, chiar modificat. O vedere completa asupra fisierului *fulladder.ap* se poate obtine prin selectarea din meniul **Tools** a optiunii **Flat** si înconjurarea cu o fereastra a întregului layout. Se poate remarca rezultatul sintetizarii layout-ului pe doua rânduri orizontale, vizibil delimitate în figura 7.7.

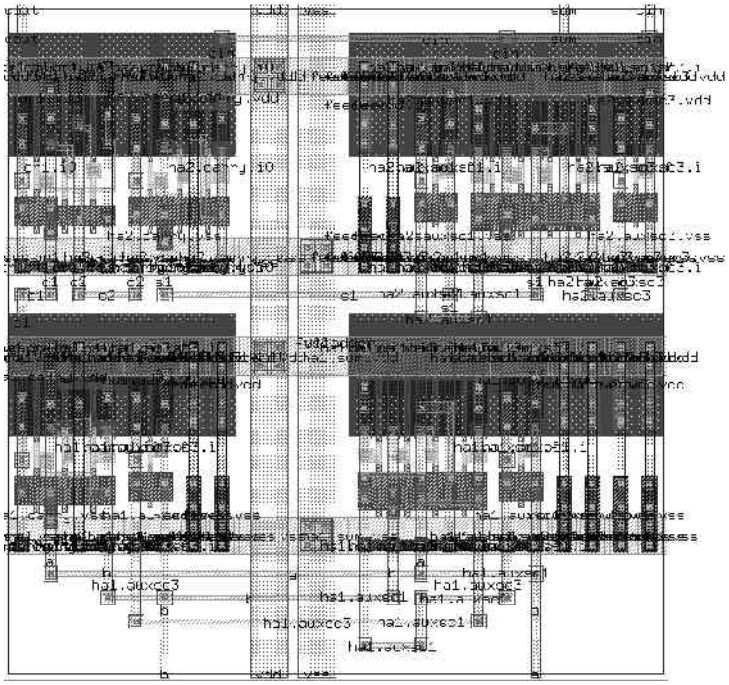


Figura 7.7: Layout-ul simbolic al componentei fulladder

7.3.8 Generarea unui layout real

Pâna acum layout-urile generate descriau circuitul doar la nivel de celule simbolice. Turnatoria de siliciu are însa nevoie de o descriere a layout-ului folosind straturi (layer-e) si dreptunghiuri. Trecerea de la layout simblolic la layout real se face cu utilitarul **s2r** (symbolic to real), pe baza unui fisier de tehnologie ce contine datele necesare tranzitiei la o anumita tehnologie specifica turnatoriei. Numele acestui fisier este specificat de variabila de mediu RDS_TECHNO_NAME.

```
s2r fulladder fulladder

primul fulladder – fulladder.ap (layout simbolic)
al doilea fulladder – fulladder.cif (layout real)
```

7.3.9 Vizualizarea layout-ului real

Layout-ul real poate fi vizualizat cu utilitarul **dreal**, care are o functionare identica cu utilitarul **graal**. Diferentele constau doar în formatul fisierului. Pentru a vizualiza întreaga structura a circuitului, se foloseste aceeasi optiune **Flat** din meniul **Tools**.

7.4 Desfasurarea lucrarii

- Sintetizati componenta halfadder parcurgând operatiile descrise în paragraful 7.2 si în figura 7.2.

- Pe baza componentei **halfadder**, sintetizati componenta **fulladder** cu structura din figura 7.6. Parcurgeti operatiile descrise în paragraful 7.3.

Lucrarea

8 Proiectarea unui automat sincron

8.1 Introducere

Aceasta lucrare prezinta proiectarea unui automat sincron care returneaza la iesire ”1” logic dupa ce numara la intrare patru valori de ”1” logic consecutive. Proiectul are ca scop crearea unui core de circuit, fara a lega pad-uri. Circuitul va fi descris sub forma unui automat.

8.2 Automatul sincron

Graful de tranzitii al automatului este ilustrat în figura 8.1. Automatul este de tip Moore imediat cu reset sincron (iesirea depinde doar de starea prezenta a automatului). Semnificatia starilor este urmatoarea:

- S0 - nu s-a primit nici o valoare de ‘1’;
- S1 - s-a primit o singura valoare de ‘1’;
- S2 - s-au primit 2 valori consecutive de ‘1’;
- S3 - s-au primit 3 valori consecutive de ‘1’;
- S4 - s-au primit 4 sau mai multe valori consecutive de ‘1’.

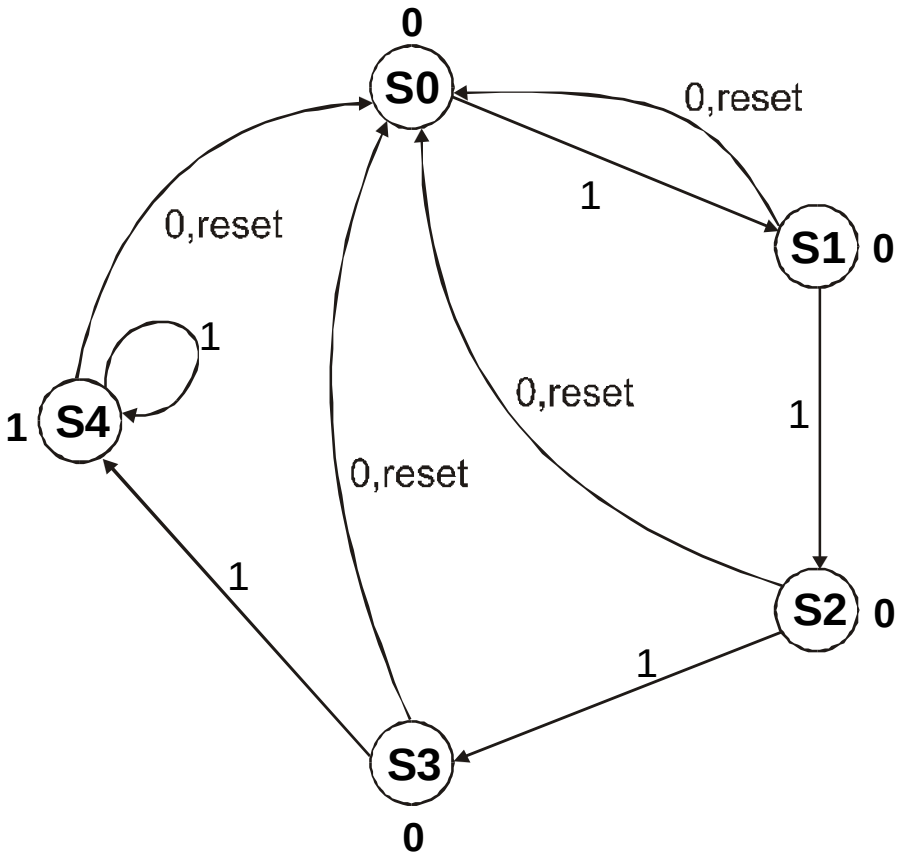


Figura 8.1: Graful de tranzitie al automatului finit

Operatii:

1. Descrierea automatului folosind subsetul Alliance FSM. În scop de depanare, se vor adauga câte o iesire corespunzatoare fiecarei stari a automatului (*dcounter.fsm*).
2. Generarea descrierii comportamentale folosind utilitarul **syf** (*dcounterm.vbe*).
3. Scrierea unui fisier de vectori de test (*dcounter.pat*).
4. Simularea descrierii comportamentale cu ajutorul fisierului de pattern-uri si salvarea rezultatului simularii într-un nou fisier de pattern-uri (*r1.pat*).
5. Descrierea automatului fara iesirile aditionale dar cu intrari de alimentare (*counter.fsm*).
6. Generarea descrierii comportamentale folosind utilitarul **syf** (*counterm.vbe*).
7. Scrierea unui fisier de vectori de test (*counter.pat*) modificând fisierul deja existent (*dcounter.pat*).
8. Generarea unei descrieri structurale (*counterl.vst*) folosind biblioteca de celule standard cu utilitarul **scmap**.
9. Simularea descrierii structurale cu ajutorul fisierului de pattern-uri si salvarea rezultatului simularii într-un nou fisier de pattern-uri (*r2.pat)*.
10. Plasarea si rutarea descrierii structurale cu utilitarul **scr** (*counterl.ap*).
11. Verificarea layout-ului cu utilitarul **druc**.
12. Generarea unui layout real cu utilitarul **s2r** (*counterl.cif*).

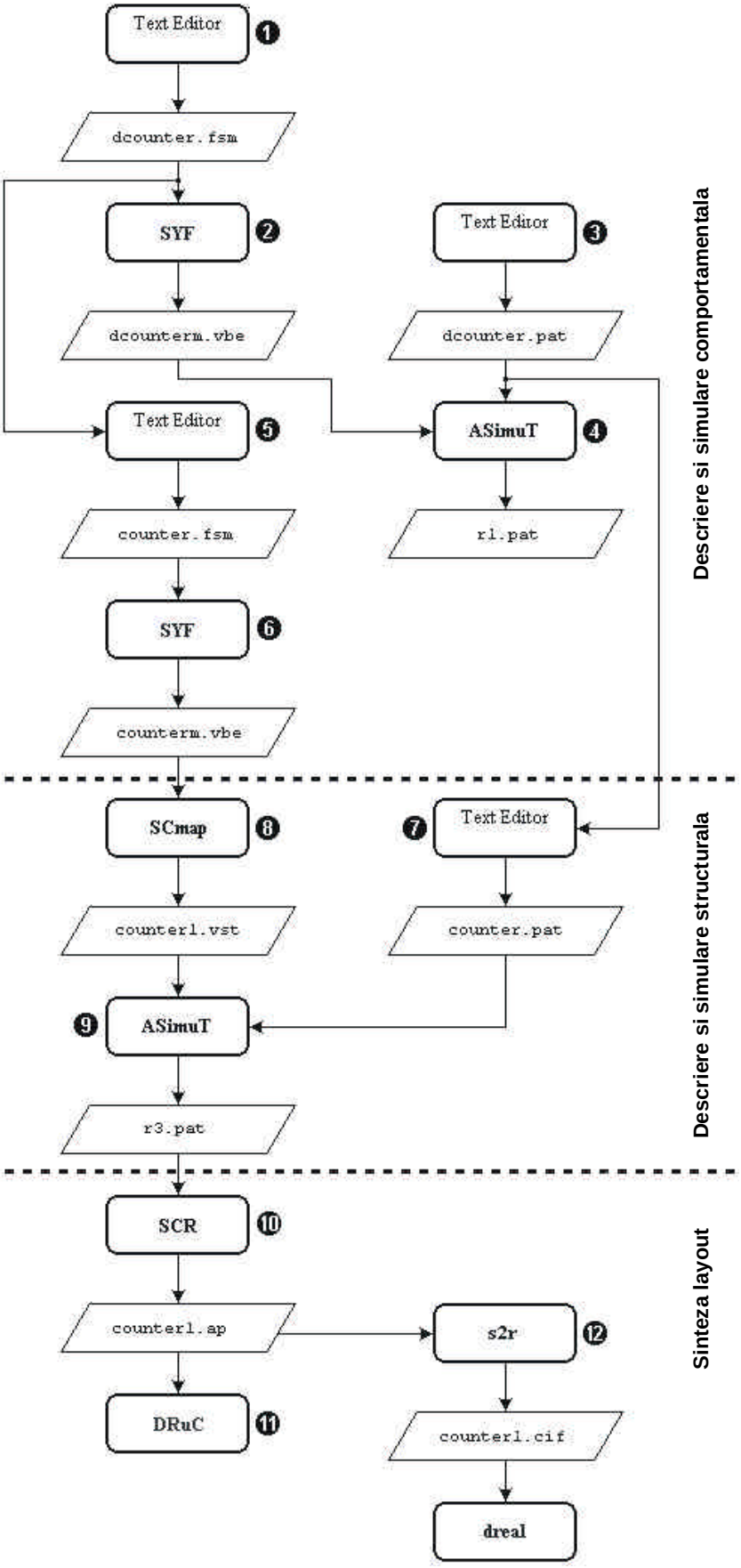


Figura 8.2: Operatiile necesare proiectarii automatului

8.2.1 Descrierea automatului

Subsetul VHDL folosit pentru descrierea automatului (FSM – finite state machine – automat finit) este putin diferit fata de subsetul standard (VBE) Alliance, incluzând în plus un element fara de care automatul nu ar fi putut fi descris comportamental: procesul. Definitia entitatii este standard, deosebirile aparând în partea de arhitectura. Astfel, din listing se pot observa noi definitii de variabile si de semnale. S0, S1, S2 si S3 fac parte din multimea starilor, iar semnalele CURRENT_STATE si NEXT_STATE sunt de tip stare. Cele trei linii care încep cu "-- pragma" nu sunt comentarii, ci stabilesc denumirea locala a celor trei semnale (ceas, starea actuala si starea urmatoare). CLOCK, CUR_STATE si NEX_STATE sunt cuvinte rezervate din subsetul FSM si sunt identificate cu denumirile din dreapta.

```
ENTITY counter IS
  PORT ( ck, i, reset: IN BIT;
         o: OUT BIT;
          s0_flag, s1_flag, s2_flag, s3_flag, s4_flag : OUT BIT
         );
END counter;


ARCHITECTURE automate OF counter IS

TYPE STATE_TYPE IS ( S0, S1, S2, S3, S4 );
-- pragma CLOCK ck
-- pragma CUR_STATE CURRENT_STATE
-- pragma NEX_STATE NEXT_STATE

SIGNAL CURRENT_STATE, NEXT_STATE: STATE_TYPE;

BEGIN
  PROCESS ( CURRENT_STATE, i, reset )
  BEGIN
    IF ( reset = '1' ) THEN
      NEXT_STATE <= S0;
      o <= '0';
    ELSE
     CASE CURRENT_STATE IS

       WHEN S0 => s0_flag <= '1';
         IF ( i = '1' ) THEN
           NEXT_STATE <= S1;
         ELSE
           NEXT_STATE <= S0;
         END IF;
         o <= '0';
```

```
      WHEN S1 => s1_flag <= '1';
        IF ( i = '1' ) THEN
          NEXT_STATE <= S2;
        ELSE
          NEXT_STATE <= S0;
        END IF;
        o <= '0';

      WHEN S2 => s2_flag <= '1';
        IF ( i = '1' ) THEN
          NEXT_STATE <= S3;
        ELSE
          NEXT_STATE <= S0;
        END IF;
        o <= '0';

      WHEN S3 => s3_flag <= '1';
        IF ( i = '1' ) THEN
          NEXT_STATE <= S4;
        ELSE
          NEXT_STATE <= S0;
        END IF;
        o <= '0';

      WHEN S4 => s4_flag <= '1';
        IF ( i = '1' ) THEN
          NEXT_STATE <= S4;
        ELSE
          NEXT_STATE <= S0;
        END IF;
        o <= '1';

      WHEN OTHERS =>
        ASSERT ( '1' )
         REPORT "Illegal State";

    END CASE;
  END IF;
  END PROCESS;

  PROCESS ( ck )
  BEGIN
    IF ( ck = '0' AND NOT ck'STABLE ) THEN
       CURRENT_STATE <= NEXT_STATE;
    END IF;
  END PROCESS;
END  automate;
```

Functionarea automatului este descrisa de doua procese. Unul dintre procese are în lista de senzitivitati doar semnalul de ceas (ck) si asigura tranzitia automatului în starea urmatoare la fiecare front descrescator de ceas, înlocuind practic starea actuala (CURRENT_STATE) cu starea urmatoare (NEXT_STATE). Acest proces modeleaza registrul de stare.

Al doilea proces are în lista de senzitivitati intrarea automatului, intrarea RESET si starea curenta. În cadrul acestui proces se testeaza semnalul RESET, care, activ pe ”1” logic, va aduce automatul în starea initiala S0 si aloca iesirii valoarea “0”. Acest proces modeleaza circuitul combinational.

Pe ramura de ELSE a procesului se testeaza succesiunea de biti de “1” la intrare, trecând prin toate starile automatului. În fiecare din aceste stari se activeaza (“1” logic) iesirile de control (s0_flag … s3_flag), acestea fiind folosite pentru urmarirea si depanarea functionarii automatului. Iesirea automatului este validata (”0” sau “1” în functie de stare) în cadrul starii si nu în cadrul tranzitiei (IF … THEN), facând ca automatul sa fie de tip Moore.

8.2.2 Generarea descrierii comportamentale si simularea ei

Odata descris automatul cu subsetul FSM, trebuie facuta tranzitia catre o descriere comportamentala în subsetul Alliance (VBE) care sa permita sinteza. Utilitarul responsabil pentru aceasta tranzitie este **syf** (synthesizer FSM), care accepta urmatoarea sintaxa si genereaza un fisier *docunterm.vbe:*

```
syf -m -C -V -E dcounter

-m - algoritm de codare "Mustang”
-C - verifica consistenta tranzitiilor
-V - Verbose (cu mesaje)
-E - salveaza rezultatul codarii starilor în dcounter.enc
dcounter - dcounter.fsm
```

Fisierul de pattern-uri pentru verificarea descrierii comportamentale cuprinde toate semnalele de intrare si iesire, inclusiv pe cele de control. Semnalul de ceas este generat prin specificarea alternativa a valorilor “0” si “1” logic.

```
-- lista de intrari/iesiri
in    ck B;;
in    reset B;;
in    i B;;;;
out   o B;;;;
out   s0_flag B;;;
```

```
out   s1_flag  B;;;
out   s2_flag  B;;;
out   s3_flag  B;;;
out   s4_flag  B;;

begin

--  descrierea  pattern-urilor

--         c r i   o    s  s  s  s  s
--         k e         0  1  2  3  4
--           s         _  _  _  _  _
--           e         f  f  f  f  f
--           t         l  l  l  l  l
--                     a  a  a  a  a
--                     g  g  g  g  g

pat_0  :   0 1 0   ?0   ?0 ?0 ?0 ?0 ?0 ;
pat_1  :   1 1 0   ?0   ?0 ?0 ?0 ?0 ?0 ;
pat_2  :   0 1 0   ?0   ?0 ?0 ?0 ?0 ?0 ;
pat_3  :   1 0 0   ?0   ?1 ?0 ?0 ?0 ?0 ;
pat_4  :   0 0 0   ?0   ?1 ?0 ?0 ?0 ?0 ;
pat_5  :   1 0 1   ?0   ?1 ?0 ?0 ?0 ?0 ;
pat_6  :   0 0 1   ?0   ?0 ?1 ?0 ?0 ?0 ;
pat_7  :   1 0 1   ?0   ?0 ?1 ?0 ?0 ?0 ;
pat_8  :   0 0 1   ?0   ?0 ?0 ?1 ?0 ?0 ;
pat_9  :   1 0 1   ?0   ?0 ?0 ?1 ?0 ?0 ;
pat_10 :   0 0 1   ?0   ?0 ?0 ?0 ?1 ?0 ;
pat_10 :   1 0 1   ?0   ?0 ?0 ?0 ?1 ?0 ;
pat_11 :   0 0 1   ?1   ?0 ?0 ?0 ?0 ?1 ;
pat_12 :   1 0 1   ?1   ?0 ?0 ?0 ?0 ?1 ;
pat_13 :   0 0 1   ?1   ?0 ?0 ?0 ?0 ?1 ;
pat_14 :   1 0 1   ?1   ?0 ?0 ?0 ?0 ?1 ;
pat_15 :   0 0 1   ?1   ?0 ?0 ?0 ?0 ?1 ;
pat_16 :   1 1 1   ?0   ?0 ?0 ?0 ?0 ?0 ;
pat_17 :   0 1 1   ?0   ?0 ?0 ?0 ?0 ?0 ;
pat_18 :   1 0 1   ?0   ?1 ?0 ?0 ?0 ?0 ;
pat_19 :   0 0 1   ?0   ?0 ?1 ?0 ?0 ?0 ;

end;
```

În acest fisier de pattern-uri s-au specificat si valorile corecte ale iesirilor. Daca aceste valori difera de cele generate de urmatoarea linie de comanda, atunci utilitarul **asimut** va semnala neconcordantele. Atât *r1.pat* cât si *dcounter.pat* pot fi vizualizate cu ajutorul utilitarului **xpat**.

```
asimut -b dcounterm dcounter r1

-b       - simulare comportamantala
dcounterm -  dcounterm.vbe
dcounter  -  dcounter.pat
r1        - rezultatul simularii în fisierul r1.pat
```

8.2.3 Descrierea si verificarea automatului fara semnale de control

Prin folosirea semnalelor de control suplimentare a putut fi verificata functionarea corecta a automatului. Odata verificata functionarea corecta a automatului se poate trece la descrierea propriu-zisa a automatului care va fi ulterior sintetizat. În acest scop, se vor elimina semnalele de control (**s0_flag** … **s3_flag)** si se vor adauga doua noi iesiri circuitului: alimentarile **vdd** si **vss**. Declaratia entitatii împreuna cu restul descrierii automatului vor constitui fisierul *counter.fsm*.
Descrierea comportamentala (*counter.vbe*) poate fi generata prin folosirea aceluiasi utilitar **syf**. Va fi generat un fisier numit *counterm.vbe*.

```
syf -m -C -V -E counter

-m - algoritm de codare "Mustang”
-C - verifica consistenta tranzitiilor
-V - Verbose
-E - salveaza rezultatul codarii starilor în counter.enc
counter -  counter.fsm
```

Corectitudinea descrierii comportamentale poate fi usor verificata prin analizarea continutului fisierului *counterm.vbe*. De exemplu, la capatul fisierului se afla descrierea functiei logice de iesire a circuitului, care depinde doar de starea curenta a automatului si de semnalul **reset**. În consecinta automatul a fost corect sintetizat ca un automat Moore, asa cum a si fost descris în fisierul cu extensia .FSM.

```
o <= (current_state_S4 and not(reset));
```

Verificarea functionala a descrierii generate se poate face cu ajutorul fisierului anterior de pattern-uri din care trebuie eliminate semnalele de iesire de control (**s0_flag** … **s3_flag**) si introduse doua semnale de intrare noi, alimentarile **vdd** si **vss**, a caror valoare trebuie sa fie ‘1’ si respectiv ‘0’. În cazul în care perioada de ceas din fisierul de pattern-uri este prea mica în raport cu intrarile si iesirile si nu se pot observa în detaliu tranzitiile automatului, se poate mari aceasta perioada, prin multiplicarea valorilor de ‘0’ si ‘1’ în pattern-uri succesive. Noul fisier de pattern-uri se va numi *counter.pat.*

8.2.4 Generarea descrierii structurale si a layout-ului

Pe baza descrierii comportamentale (*counterm.vbe*) si a bibliotecii de celule standard se poate trece la generarea descrierii structurale:

```
scmap counterm counterl
```

Fisierul generat se va numi *counterl.vst* si poate fi vizualizat cu utilitarul **xsch**. Se pot observa cele trei bistabile de stare, ceea ce înseamna ca au fost folositi trei biti pentru codificarea celor patru stari ale automatului, deci automatul este optimizat, numarul de biti de stare fiind minim.
Descrierea comportamentala poate fi verificata cu acelasi fisier de pattern-uri folosit anterior (*counter.pat*) prin utilitarul **asimut**:

```
asimut counterl counter r3

counterl  -  counterl.vst
counter   -  counter.pat
r3        -  fisier de pattern-uri rezultand (r3.pat)
```

Layout-ul simbolic poate fi generat folosind utilitarul **scr**. Se va obtine fisierul *counterl.ap*.

```
scr -p -r -l 3 counterl

-p – apelarea procesului de rutare automata
-r – apelarea procesului de plasare automata
-l – specifica numarul de rânduri (3) ale layout-ului
counterl - fisierul generat
```

Fisierul *.ap* poate fi vizualizat cu utilitarul **graal**. Se pot identifica cele trei rânduri specificate anterior la sinteza layout-ului.
La nivel de layout se poate face o verificare a respectarii regulilor de proiectare si implemetare în siliciu, DRC. În categoria regulilor de proiectare intra regulile geometrice, cum ar fi distantele minime între NWELL si PWELL, latimile traselor de aluminiu si interdictii de suprapunere ale diverselor layere. Utilitarul care se ocupa de aceasta verificare este **druc**. Eventualele erori sunt listate într-un fisier cu extensia *drc*.

```
druc counterl
```

Trecerea de la layout-ul simbolic la cel real se face, la fel ca si în lucrarea anterioara, cu utilitarul **s2r**. Fisierul rezultat se numeste *counterl.cif* si poate fi vizualizat cu utilitarul **dreal**. Imaginea grafica a layout-ului este prezentata în figura 8.3.

```
s2r -v counterl counterl
```

8.3 Desfasurarea lucrarii

- Sintetizati automatul parcurgând operatiile descrise în paragraful 8.2.

- Modificati automatul astfel încât sa fie de tip Mealy. Comparati numarul de perioade de ceas în care automatul genereaza ‘1’ pe iesire pentru variantele Mealy si Moore.

- Modificati automatul Mealy astfel încât sa comute pe frontul pozitiv al semnalului de ceas.

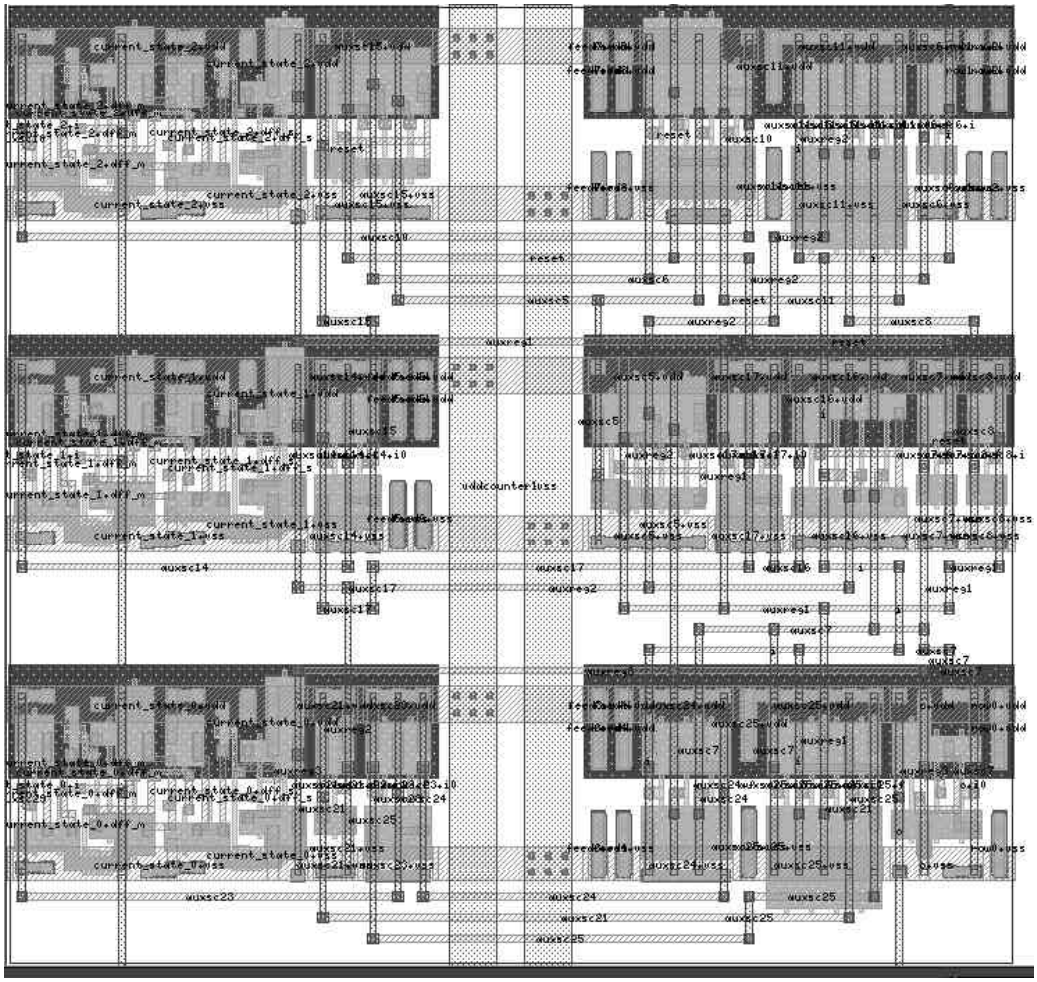


Figura 8.3: Layout-ul real al automatului

Lucrarea

9 Legarea pad-urilor

9.1 Introducere

Atât circuitele combinationale cât si cele secventiale sintetizate pâna acum au fost limitate la descrierea core-ului, adica a miezului circuitului, fara a tine cont de necesitatea interfatarii acestuia cu lumea exterioara.

Orice circuit, indiferent de complexitate si de functionalitate trebuie sa fie legat de restul circuitelor si, evident, alimentat cu tensiune. Interfata circuitului cu exteriorul se realizeaza prin intermediul pad-urilor, care pot fi de mai multe tipuri:

- pad-uri de intrare;
- pad-uri de iesire;
- pad-uri bidirectionale;
- pad-uri de alimentare;
- pad-uri de ceas.

Din punct de vedere comportamental, pad-urile nu sunt altceva decât niste buffere de semnal, deosebindu-se în functie de tip doar prin diversi parametri cum ar fi capacitatea si rezistenta.

Mediul Alliance dispune de o biblioteca de pad-uri continând pentru toate tipurile necesare o descriere comportamentala (*.vbe) si un layout simbolic (*.ap). Cele mai des folosite modele de pad-uri sunt:

- pvdde_sp, pvddi_sp, pvsse_sp, pvssi_sp – pad-uri de alimentare interne si externe;
- pi_sp – pad de intrare;
- po_sp – pad de iesire;
- pio_sp – pad bidirectional;
- pck_sp – pad de ceas intern;
- pvddeck_sp – pad de ceas extern.

Un pad tipic de intrare are în biblioteca de pad-uri (directorul cells/padlib) urmatoarea descriere:

```
— VHDL data flow description generated from ‘pi_sp‘
ENTITY pi_sp IS
  GENERIC (
    CONSTANT area       : NATURAL := 86000;     --area
    CONSTANT cin_pad    : NATURAL := 654;       --cin_pad
    CONSTANT tpll_pad   : NATURAL := 1487;      --tpll_pad
    CONSTANT rdown_pad  : NATURAL := 234;       --rdown_pad
    CONSTANT tphh_pad   : NATURAL := 233;       --tphh_pad
    CONSTANT rup_pad    : NATURAL := 273        --rup_pad
  );

PORT (
  pad  : in BIT;   --pad
  t    : out BIT;  --t
  ck   : in BIT;   --ck
  vdde : in BIT;   --vdde
  vddi : in BIT;   --vddi
  vsse : in BIT;   --vsse
  vssi : in BIT    --vssi
  );
END pi_sp;

— Architecture Declaration
ARCHITECTURE behaviour_data_flow OF pi_sp IS

BEGIN
   ASSERT((((vddi and vdde) and not(vssi)) and not (vsse))=’1’)
   REPORT "power supply is missing on pi_sp"
   SEVERITY WARNING;
t <= pad;
END;
```

Declaratia entitatii contine, în plus fata de o descriere comportamentala clasica, declaratiile unor constante de proces tipice pentru fiecare tip de pad în parte. În lista de semnale de intrare si de iesire apare atât o intrare de ceas (nefolosita în descrierea

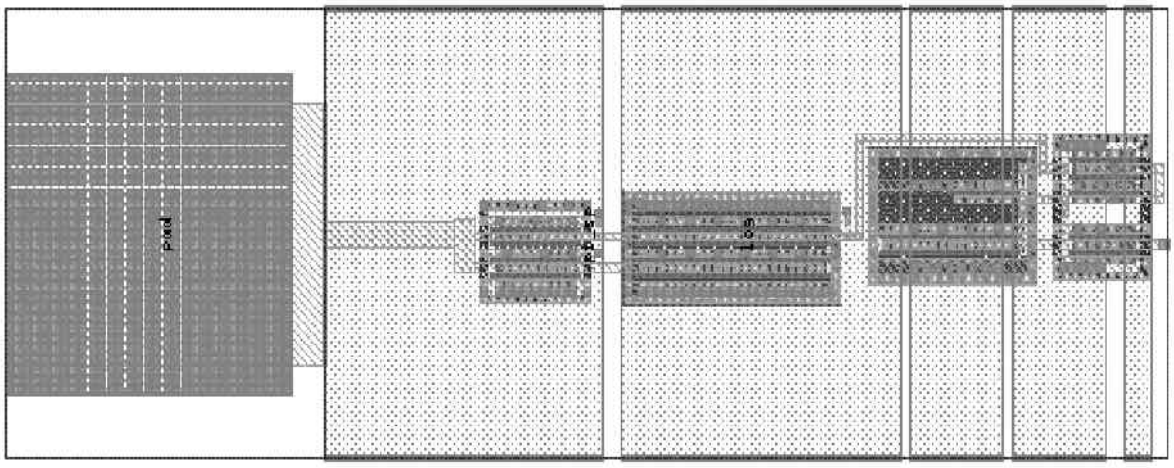


Figura 9.1: Layout-ul unui pad de intrare (pi_sp.ap)

arhitecturala) cât si doua perechi de alimentari: interne si externe. Legarea acestora va fi discutata în paragrafele urmatoare. Descrierea arhitecturala nu face altceva decât sa asocieze intrarea iesirii. Layout-ul pad-ului descris mai sus este prezentat în figura 9.1. În etapa de productie finala a chip-ului, în partea stânga a pad-ului va fi lipit firul de aur care va face conexiunea electrica cu pinii exteriori ai circuitului. Pe lânga suprafata destinata lipirii firului, pad-ul mai contine circuite de buffer-are a semnalului cât si de protectie.

9.2 Legarea pad-urilor de core

Pad-urile din biblioteca Alliance pot fi folosite exact ca orice componente descrise comportamental sau structural, adica pot fi legate printr-o descriere structurala de celelalte componente (în general core) ale circuitului.

Având în vedere particularitatile pad-urilor enumerate în paragraful anterior, legarea lor la un core trebuie facuta dupa anumite reguli, ilustrate in figura 9.2. Exemplul este ales pentru automatul proiectat în laboratorul trecut, care este folosit în figura pe post de core. Semnalele suplimentare interne sunt scrise cursiv, denumirile componentelor corespunzând celor ale fisierelor VBE sau VST.

Trebuie remarcata distributia semnalului de ceas, care este întâi distribuit la toate pad-urile si doar ulterior aplicat la core, prin intermediului unui pad special (pvddeck_sp) care reface frontul semnalului.

Legarea pad-urilor la restul de componente ale circuitului poate fi facuta fie direct, prin intermediul unei descrieri structurale (fisier VST), fie generata automat de utilitarul **genlib** pe baza unei descrieri folosind niste functii C++ din biblioteca *genlib.h*. Ultima

varianta are avantajul unei lizibilitati mai bune a codului si în plus permite folosirea de bucle pentru legarea unor componente de acelasi tip (de exemplu vectori de intrare-iesire). Fisierul are extensia *.C si trecut prin utilitarul **genlib** va genera o descriere structurala, adica un fisier .VST.

```
#include <genlib.h>
main()
{
int i;
      DEF_LOFIG("counter");
      LOCON("vdd",IN,"vdd");
      LOCON("vss",IN,"vss");
      LOCON("vdde",IN,"vdde");
      LOCON("vsse",IN,"vsse");
      LOCON("in",IN,"in");
      LOCON("reset",IN,"reset");
      LOCON("out",OUT,"out");
      LOCON("ck",IN,"ck");

LOINS("pvsse_sp","p1","cki","vdde","vdd","vsse","vss",0);
LOINS("pvdde_sp","p2","cki","vdde","vdd","vsse","vss”,0);
LOINS("pvddeck_sp","p3","clock","cki","vdde","vdd","vsse","vss",0);
LOINS("pvssi_sp","p4","cki","vdde","vdd","vsse","vss",0);
LOINS("pvddi_sp","p5","cki","vdde","vdd","vsse","vss",0);

    LOINS("pi_sp", "p6",
          "in", "inin",
           "cki", "vdde", "vdd", "vsse", "vss", 0);

    LOINS("pi_sp", "p7",
          "reset", "resetreset",
           "cki", "vdde", "vdd", "vsse", "vss", 0);

    LOINS("pck_sp", "p8",
          "ck",
           "cki", "vdde", "vdd", "vsse", "vss", 0);

    LOINS("po_sp", "p9",
          "outout", "out",
           "cki", "vdde", "vdd", "vsse", "vss", 0);

    LOINS("core", "core",
          "vdd", "vss",
           "clock", "inin", "resetreset", "outout", 0);

    SAVE_LOFIG();
   exit(0);   }
```

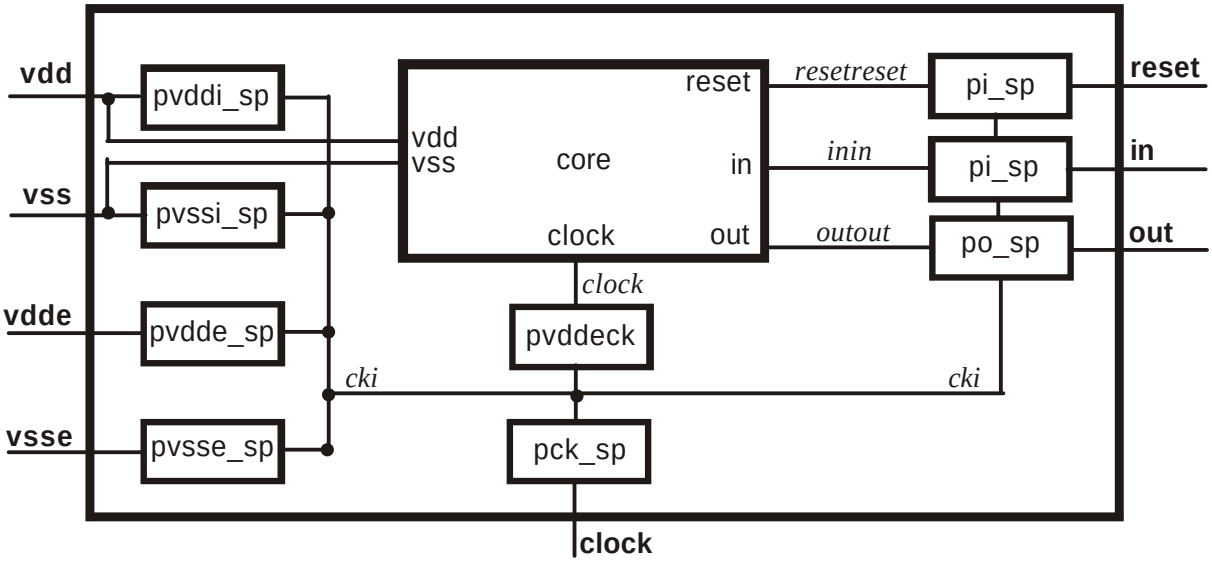


Figura 9.2: Schema de legare a pad-urilor de core

Fisierul *counter.c* are structura clasica a unui fisier sursa C, functiile scrise cu majuscule fiind continute în header-ul *genlib.h*:

DEF_LOFIG (DEFine LOgical FIGure) stabileste numele descrierii structurale generate, si implicit numele fisierului VST generat, în cazul de fata, counter.

LOCON (LOgical CONnector) specifica semnalele de intrare si de iesire ale descrierii structurale. Primul argument al functiei defineste portul extern al circuitului, al doilea tipul de semnal, iar ultimul numele semnalului în interiorul circuitului. Pentru o lizibilitate mai mare a codului, este bine ca cele doua nume sa coincida.

LOINS (LOgical INStance) defineste componentele folosite în interiorul descrierii structurale. Primul argument al functiei specifica numele componentei ce va fi folosita, componenta ce trebuie sa existe ca si fisier de descriere comportamentala sau structurala. Al doilea nume este folosit doar intern la instantierea componentei, înainte de maparea semnalelor care urmeaza enumerate în lista de argumente ale functiei. Ordinea semnalelor din LOINS trebuie sa corespunda cu cea din descrierea initiala a componentei. Ultimul semnal este urmat de un ”0” care specifica încheierea listei.

SAVE_LOFIG (SAVE LOgical FIGure) salveaza întreaga descriere si genereaza fisierul .VST.

Semnalele auxiliare folosite în cadrul descrierii nu trebuie declarate explicit, ele fiind automat detectate ca fiind cele care nu apar în lista de intrari/iesiri de la începutul fisierului.

Pentru un pad de intrare (**pi_sp**, p6), responsabil pentru semnalul reset, intrarea **reset** a circuitului este legata direct la pad, care are ca si iesire semnalul auxiliar **resetreset**, legat în cele din urma la core. În sens invers, pentru un pad de iesire (**po_sp**, p9), iesirea **outout** din core este legata la intrarea pad-ului care are iesirea **out**, adica exact iesirea circuitului.

Semnalul extern de ceas (**ck**) este întâi aplicat pad-ului **pck_sp** care va avea ca iesire un semnal intern **cki**. Acesta este aplicat tuturor pad-urilor folosite în circuit. Semnalul intern de ceas **cki** este aplicat înca unui pad **pvddeck_sp** si de abia de la iesirea acestuia (semnalul **clock**) este aplicat la core. Pad-ul intermediar are ca scop refacerea frontului semnalului de ceas.

Pentru pad-urile de alimentare (care au doar semnale de intrare în descrierea comportamentala) nu trebuie avut grija decât la aplicarea semnalului de ceas. În plus, alimentarea core-ului se face de la sursele de tensiune interne, adica cu **vdd** si **vss**.

Avantajul folosirii acestor functii C pentru a genera o descriere structurala nu este atât de evident în cazul unor circuite simple cu putine intrari sau iesiri. Fiind un fisier sursa C, se pot însa folosi orice tip de constructii repetitive, care usureaza mult munca în cazul unor intrari sau iesiri de tip vector. De exemplu, daca intrarea circuitului nu ar fi pe un singur bit ci pe 8 biti, s-ar putea folosi urmatoarea secventa de cod:

```
for (i = 1; i < 9; i++)
     LOINS("pi_sp", NAME("p%d", i),
            NAME("a[%d]", i), NAME("aa[%d]", i),
           "cki", "vdde", "vdd", "vsse", "vss", 0);
```

Apare în plus functia NAME, care genereaza un sir de caractere pe baza unei variabile numerice. În cazul de fata, bucla *for* va instantia opt componente de tipul pi_sp, numite p1 ... p8, care au ca intrare a[1] ... a[8] si ca iesire semnalele intermediare aa[1] ... aa[8].

Odata scris fisierul *counter.c*, se poate genera descrierea structurala cu urmatoarea comanda:

```
genlib -v counter
```

Fisierul *counter.vst* poate fi vizualizat atât în mod text, cât si în mod grafic, folosind utilitarul **xsch**.

9.3 Generarea layout-ului cu pad-uri

Descrierea structurala existenta incluzând si pad-urile nu permite o generare de layout uzuala, cu ajutorul utilitarului **scr**, deoarece mai este nevoie de date suplimentare referitoare la pozitia fizica a pad-urilor în jurul circuitului. Se va folosi utilitarul **ring** care însa necesita un fisier suplimentar cu extensia RIN. Fisierul (*counter.rin*) poate avea urmatoare structura:

```
width (vdd 20 vss 20)
######################
west  (p1 p2 p4 p5 p3)
north (p6 p7)
east  (p8)
south (p9)
```

Functia **width** specifica grosimea traseelor de alimentare în unitati lambda. Prin varierea acestui parametru se pot rezolva unele probleme sau evita unele mesaje de eroare aparute la operatie de rut-are. Functia este optionala.

Cele patru functii de pozitionare west, north, east, south pozitioneaza pad-urile în jurul circuitului. Pad-urile trebuie sa fie plasate pe cel putin o latura a circuitului si sa aiba cel putin un pad în lista. Ordonarea pentru laturile east() si west() este de sus în jos, iar pentru north() si south() de la stânga la dreapta. Numele din lista corespund cu numele componentelor din descrierea structurala si, implicit, cu cele din fisierul C. Singurele restrictii la aranjarea pad-urilor pe laturile core-ului sunt cele referitoare la pad-urile de alimentare (**pvvdi_sp** si **pvssi_sp**), care trebuie sa fie plasate cât mai aproape de mijlocul laturii core-ului pe cât posibil, si nici într-un caz pe colt. Daca pad-urile sunt amplasate prea aproape de marginea circuitului, ele nu mai pot fi legate de inelele de vdd si vss care înconjoara core-ul. Este indicat ca numarul de pad-uri pe o latura sa fie adaptat la lungimea si latimea core-ului.

Înainte de generarea layout-ului simbolic trebuie verificat daca mediul Alliance are setata calea atât pentru biblioteca de celule standard, cât si pentru cea de pad-uri. Cele doua setari se gasesc în variabila de mediu MBK_CATA_LIB care trebui modificata în $TOP/cells/sclib:$TOP/cells/padlib:

```
export $TOP/cells/sclib:$TOP/cells/padlib
```

Comanda pentru generarea layoutului (*counter.ap*) este:

```
ring counter counter
```

```
primul counter     – counter.vst si counte.rin
al doilea counter  - counter.ap (layout simbolic)
```

Din layout-ul simbolic se poate trece la un layout real cu utilitarul **s2r**.

În figura 9.3 este prezentat layout-ul simbolic al unui circuit cu 40 de pad-uri. Partea de vest a circuitului este ocupata de pad-urile de intrare. Pad-urile de iesire sunt grupate pe partea de est si pot fi identificate pe baza ariilor mai mari ocupate de capacitoare, necesare generarii unui semnal de iesire cu front abrupt si cu fan-out superior.

9.4 Desfasurarea lucrarii

- Generati un layout cu pad-uri pornind de la descrierea structurala a automatului counter (*counter.vst*) din lucrarea 8. Folositi în acest scop fisierele *counter.c* (paragraful 9.2) si *counter.rin* (paragraful 9.3).

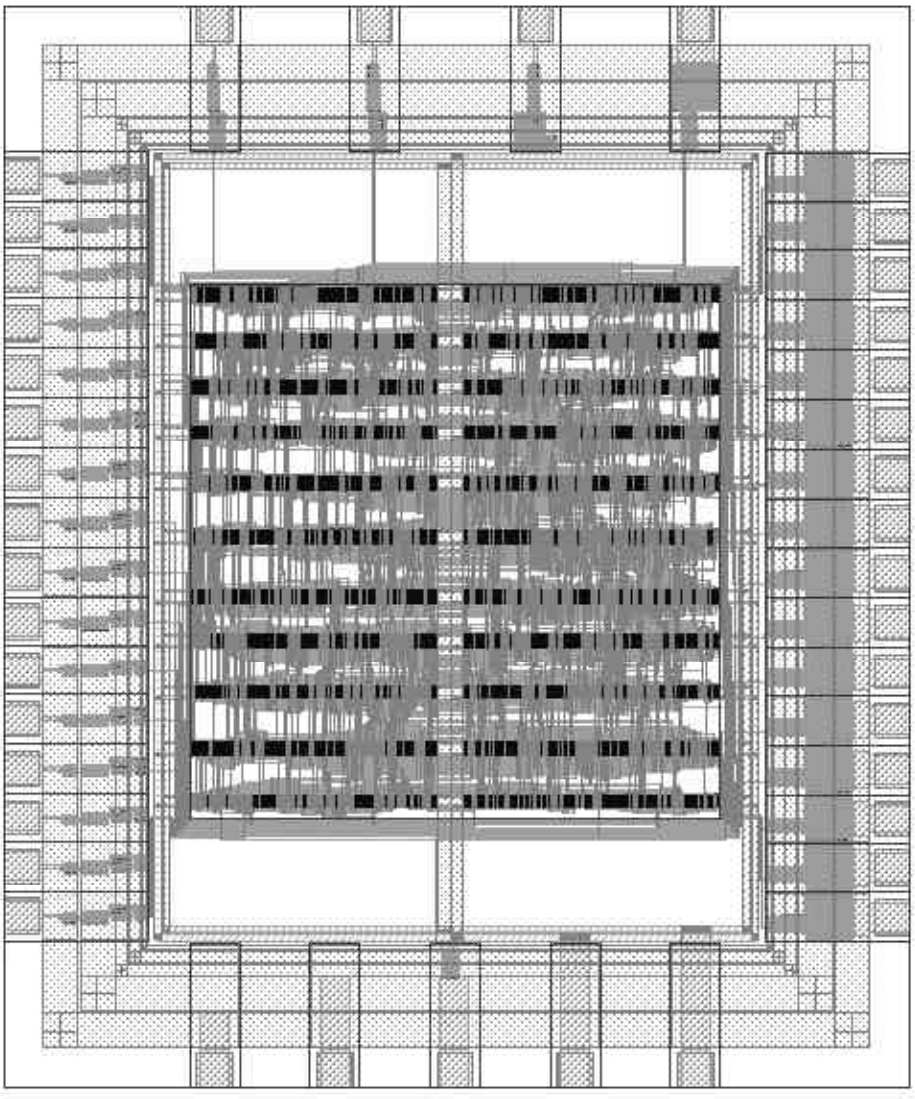


Figura 9.3: Layout complet cu pad-uri

Lucrarea

10 Proiectarea unui multiplicator

10.1 Introducere

Proiectul are ca scop modelarea unui sistem riguros sincron care accepta la intrare doi vectori binari pe 4 biti si returneaza produsul acestora reprezentat pe 8 biti. Schema de nivel înalt a multiplicatorului este prezentata în figura 10.1.

Denumire port	Sens	Dim	Semnificatie
a	IN	4	primul operand
b	IN	4	al doilea operand
prod	OUT	8	rezultatul
Start	IN	1	startul operatiei de multiplicare
Ready	OUT	1	sfârsitul operatiei de multiplicare
Reset	IN	1	initializare asincrona
Ck	IN	1	semnal de ceas

Proiectul va trebui sa respecte constrângerile unui sistem sicron. Semnalul de ceas va fi aplicat tuturor registrelor folosite atât în calea de date cât si în calea de control. În plus, nu se vor folosi circuite de divizare a frecventei semnalului de ceas, ci circuite de generare a unor semnale de activare (enable). Descrierea sistemului se va face în întregime în subsetul VHDL propriu pachetului Alliance, instantiindu-se doua componente în descrierea de nivel înalt: cale de date si cale de control.

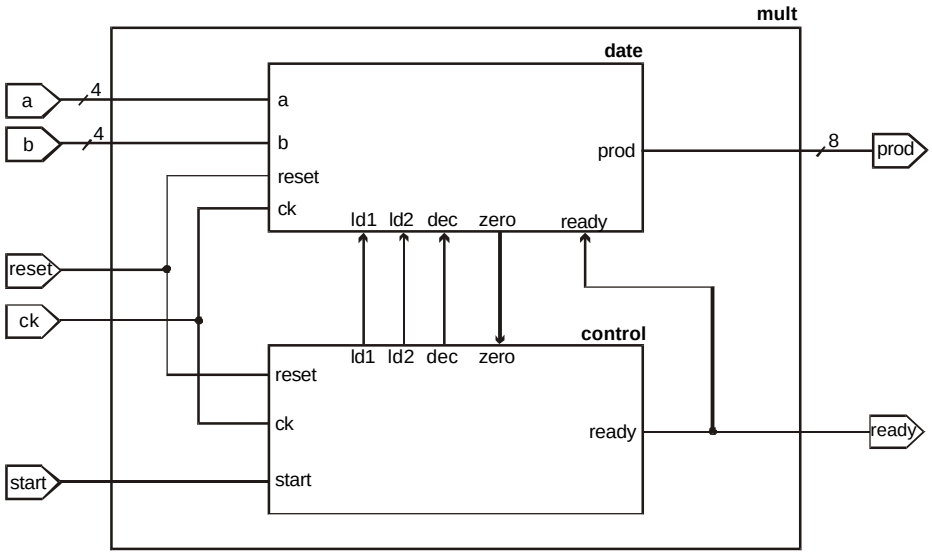


Figura 10.1 Schema bloc a multiplicatorului

Calea de date contine toate elementele necesare stocarii datelor (registre), cât si unitatea aritmetico-logica (ALU).
Calea de control este descrisa sub forma unui automat care va implementa un algoritm de înmultire prin adunari repetate.

10.2 Algoritmul de înmultire

Înmultirea prin adunari repetate constituie cel mai simplu algoritm de multiplicare cunoscut. Deînmultitul se aduna cu el însusi de un numar de ori reprezentat de înmultitor. În cazul concret al acestui multiplicator, cele doua numere sunt reprezentate pe 4 biti, produsul lor fiind pe 8 biti. Unul din cele doua numere (registrul A) va fi adunat cu el însusi în registrul P, cel de-al doilea (registrul B) urmând a fi decrementat pâna când ajunge la zero. Succesiunea operatiilor, împreuna cu un exemplu, este prezentata în continuare.

1. Reseteaza P (8 biti)
2.Încarca deînmultitul în A (4 biti)
3. Încarca înmultitorul în B (4 biti)
4. Repeta cât timp B este diferit de zero B=B-1
   P=P+A
5. P contine produs

```
7 x 10 = 70    7=00000111    10=00001010
Registrul P    Registrul A   Registrul B
0000 0000          0111          1010
0000 0000                                  - iteratia 1
0000 0111
---- ----                        ----
```

```
      0000  0111                        1001

      0000  0111                                - iteratia 2
      0000  0111
      ----  ----                        ----
      0000  1110                        1000

      0000  1110                                - iteratia 3
      0000  0111
      ----  ----                        ----
      0001  0101                        0111

      0001  0101                                - iteratia 4
      0000  0111
      ----  ----                        ----
      0001  1100                        0110

      0001  1100                                - iteratia 5
      0000  0111
      ----  ----                        ----
      0010  0011                        0101

      0010  0011                                - iteratia 6
      0000  0111
      ----  ----                        ----
      0010  1010                        0100

      0010  1010                                - iteratia 7
      0000  0111
      ----  ----                        ----
      0011  0001                        0011

      0011  0001                                - iteratia 8
      0000  0111
      ----  ----                        ----
      0011  1000                        0010

      0011  1000                                - iteratia 9
      0000  0111
      ----  ----                        ----
      0011  1111                        0001

      0011  1000                                - iteratia 10
      0000  0111
      ----  ----                        ----
      0100  0110                        0000

Produs = 0100 0110 = 16x4 + 6 = 70
```

10.3 Unitati functionale

Multiplicatorul este structurat în:
- cale de date
- cale de control

Calea de control este implementata ca un automat finit pe baza algoritmului de înmultire cu adunari repetate. Calea de date contine restul de elemente necesare functionarii circuitului si anume registre, ALU etc. Comunicatia între cele doua unitati functionale va fi realizata prin semnale interne.

10.3.1 Calea de date

Pentru calea de date sunt necesari doi registri de patru biti în care sa fie încarcati cei doi operanzi la începutul operatiei de înmultire. Sunt necesare doua registre de câte 8 biti, unul pentru a memora rezultatul intermediar al adunarilor repetate, iar cel de-al doilea pentru a memora rezultatul la terminarea operatiei de multiplicare. Unitatea aritmetico-logica este un sumator pe 8 biti, având ca intrare 4 biti ai deînmultitului si 8 biti din registrul imtermediar. Rezultatul adunarii este încarcat din nou în registrul temporar la fiecare iteratie, exceptând-o pe ultima, în care rezultatul este încarcat în registrul de iesire, odata cu activarea semnalului **ready**.

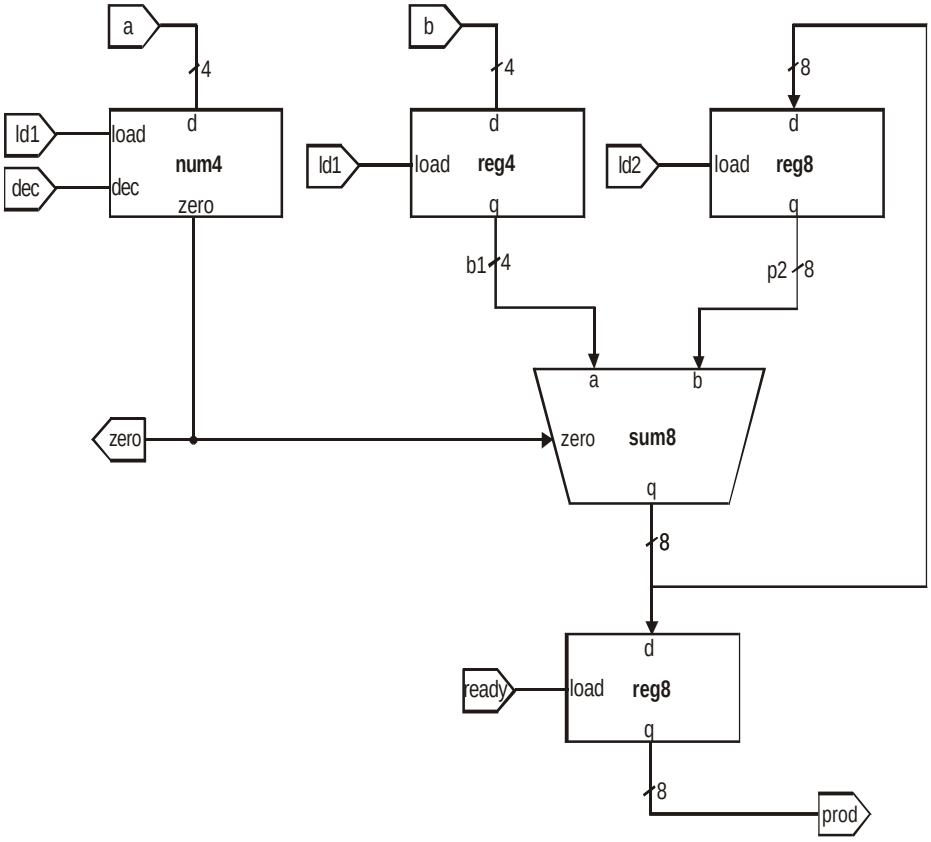


Figura 10.2: Calea de date a multiplicatorului

Operandul **a** este încarcat într-un registru la comada **start**. La fiecare iteratie, acest registru este decrementat. Când contorul registrului ajunge la ‘0’ semnalul **zero** anunta calea de control despre aparitia acestui eveniment. Calea de date este prezentata în figura 10.2.

Sumatorul ete modelat pe baza functiei logice, dupa cum urmeaza:

```
-- file: sum8.vbe
ENTITY sum8 IS
    PORT(  vdd :IN bit;
           vss :IN bit;
              a :IN bit_vector(3 DOWNTO 0);
              b :IN bit_vector(7 DOWNTO 0);
           zero :IN bit;
              q :OUT bit_vector(7 DOWNTO 0));
END sum8;

ARCHITECTURE behave OF sum8 IS

   SIGNAL carry:bit_vector(7 DOWNTO 0);
   SIGNAL q_s: bit_vector(7 DOWNTO 0);
   SIGNAL  a1: bit_vector(3 DOWNTO 0);

BEGIN
  WITH zero SELECT
     a1 <= a WHEN ,0‘, "0000" WHEN ,1‘;

  carry(0) <= ,0‘;
  q_s(0) <= (a1(0) XOR b(0)) XOR carry(0);
  carry(1) <= (a1(0) AND b(0))
            OR (a1(0) AND carry(0))
                 OR (b(0) AND carry(0));
  q_s(1) <= (a1(1) XOR b(1)) XOR carry(1);
  carry(2) <= (a1(1) AND b(1))
              OR (a1(1) AND carry(1))
              OR (b(1) AND carry(1));
  q_s(2) <= (a1(2) XOR b(2)) XOR carry(2);
  carry(3) <= (a1(2) AND b(2))
              OR (a1(2) AND carry(2))
              OR (b(2) AND carry(2));
  q_s(3) <= (a1(3) XOR b(3)) XOR carry(3);
  carry(4) <= (a1(3) AND b(3))
              OR (a1(3) AND carry(3))
              OR (b(3) AND carry(3));
  q_s(4) <=  b(4) XOR carry(4);
  carry(5) <= b(4) AND carry(4);
```

```
  q_s(5) <= b(5) XOR carry(5);
  carry(6) <= b(5) AND carry(5);
  q_s(6) <= b(6) XOR carry(6);
  carry(7) <= b(6) AND carry(6);
  q_s(7) <= b(7) XOR carry(7);

  q <= q_s;

  ASSERT((vdd = '1') and (vss = '0'))
    REPORT "Power supply is missing on sum8"
    SEVERITY WARNING;

END behave;
```

Semnalul **ld1** este aplicat atât registrului b, cât si numaratorului si este generat doar la începutul operatiei de înmultire pentru a încarca cei doi operanzi. Semnalul **ld2** este aplicat doar registrului intermediar pe 8 biti. Semnalul **zero** este generat de numarator dupa un numar de perioade de ceas egal cu înmultitorul si este aplicat atât sumatorului, cât si trimis mai departe la calea de control. Numaratorul este comandat de semnalul **dec** ce vine de la calea de control.

Registrul de iesire se încarca doar când semnalul **ready** (provenind de la calea de control) este activ, adica atunci când operatia de înmultire este încheiata, datele fiind asftel valide la iesirea circuitului. Toate registrele sunt comutate de acelas semnal de ceas si initializate de acelas semanl **reset**.

Descrierea comportamentala a unui registru poate fi facuta conform exemplului urmator. Extinderea la un registru de 8 biti se face prin simpla modificare a dimensiunilor porturilor si a vectorilor.

```
-- file: reg4.vbe

ENTITY reg4 IS
  PORT(   vdd :IN bit;
          vss :IN bit;
           d :IN bit_vector(3 DOWNTO 0);
        reset :IN bit;
           ck :IN bit;
         load :IN bit;
            q :OUT bit_vector(3 DOWNTO 0));
END reg4;

ARCHITECTURE behave OF reg4 IS
```

```
SIGNAL q_s: reg_vector(3 DOWNTO 0) register;

BEGIN
    block1: BLOCK(ck='1' and not ck'STABLE)
           BEGIN
          q_s <= GUARDED "0000"    WHEN reset='1'
                    ELSE      d    WHEN reset='0' and load='1'
                    ELSE    q_s;
          END BLOCK block1;
    q <= q_s;

ASSERT((vdd = '1') and (vss = '0'))
   REPORT "Power supply is missing on reg4"
   SEVERITY WARNING;

END   behave;
```

10.3.2 Calea de control

Calea de control are urmatoarele intrari si iesiri:
- intrarea **start** provenind din exteriorul circuitului si intrarea **zero** de la numarator;
- iesirile **ld1**, **ld2** si iesirea **ready** pentru registrul de iesire si deasemnea pentru exteriorul circuitului.

Schema caii de control este prezentata în figura 10.3.

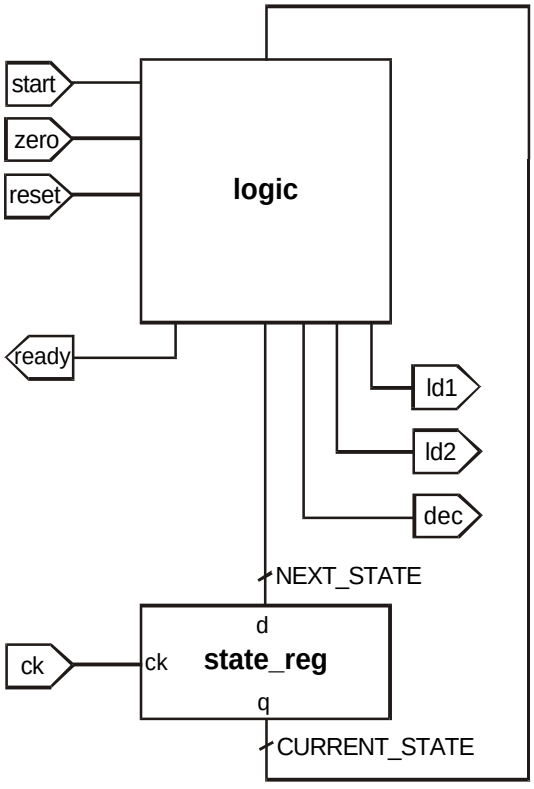


Figura 10.3: Calea de control si conexiunile cu calea de date

Numaratorul pe patru biti poate fi descris pe baza functiei logice, folosind un bloc cu garda având în lista de senzitivitati semnalul de ceas:

```
-- file: num4.vbe

ENTITY num4 IS
    PORT(   vdd :IN bit;
            vss :IN bit;
              d :IN bit_vector(3 DOWNTO 0);
          reset :IN bit;
             ck :IN bit;
           load :IN bit;
            dec :IN bit;
           zero :OUT bit);
END num4;

ARCHITECTURE behave OF num4 IS

     SIGNAL borrow: bit_vector(3 DOWNTO 0);
     SIGNAL b: bit_vector(3 DOWNTO 0);
     SIGNAL d1 : bit_vector(3 DOWNTO 0);
     SIGNAL d2 : reg_vector(3 DOWNTO 0) register;

BEGIN
  borrow(0) <= '0';

  WITH dec SELECT
  b           <= "0001" WHEN '1', "0000" WHEN '0';

  d1(0)       <= (d2(0) XOR b(0)) XOR borrow(0);
  borrow(1)   <=(((NOT d2(0)) AND b(0)) OR ((NOT d2(0))
                   AND borrow(0))OR (b(0) AND borrow(0)));
  d1(1)       <= (d2(1) XOR b(1)) XOR borrow(1);
  borrow(2)   <= (((NOT d2(1)) AND b(1))
                    OR ((NOT d2(1)) AND borrow(1))
                   OR (b(1) AND borrow(1)));
  d1(2)       <= (d2(2) XOR b(2)) XOR borrow(2);

  borrow(3)   <= (((NOT d2(2)) AND b(2)) OR ((NOT d2(2))
                   AND borrow(2)) OR (b(2) AND borrow(2)));
  d1(3)       <= (d2(3) XOR b(3)) XOR borrow(3);
```

```
  L1: BLOCK ((ck = '1') AND NOT ck'STABLE)
    BEGIN
        d2 <= GUARDED d WHEN reset = '0' and load = '1'
              ELSE   d1 WHEN reset = '0' and load = '0'
             ELSE   "1111";
    END BLOCK L1;

  zero <=  NOT (d2(0) OR d2(1) OR d2(2) OR d2(3));

  ASSERT((vdd = '1') and (vss = '0'))

     REPORT "Power supply is missing on num4"
     SEVERITY WARNING;

END  behave;
```

Automatul este de tip Mealy imediat. Graful de tranzitie al automatului este prezentat în figura 10.3.
Trecerea din starea S0 în S1 se face la activarea semnalului **start**. Automatul ramâne în starea S1 pâna când semnalul **zero** devine activ, moment în care semnalul **ready** este activat, semnalizând terminarea operatiei de înmultire.

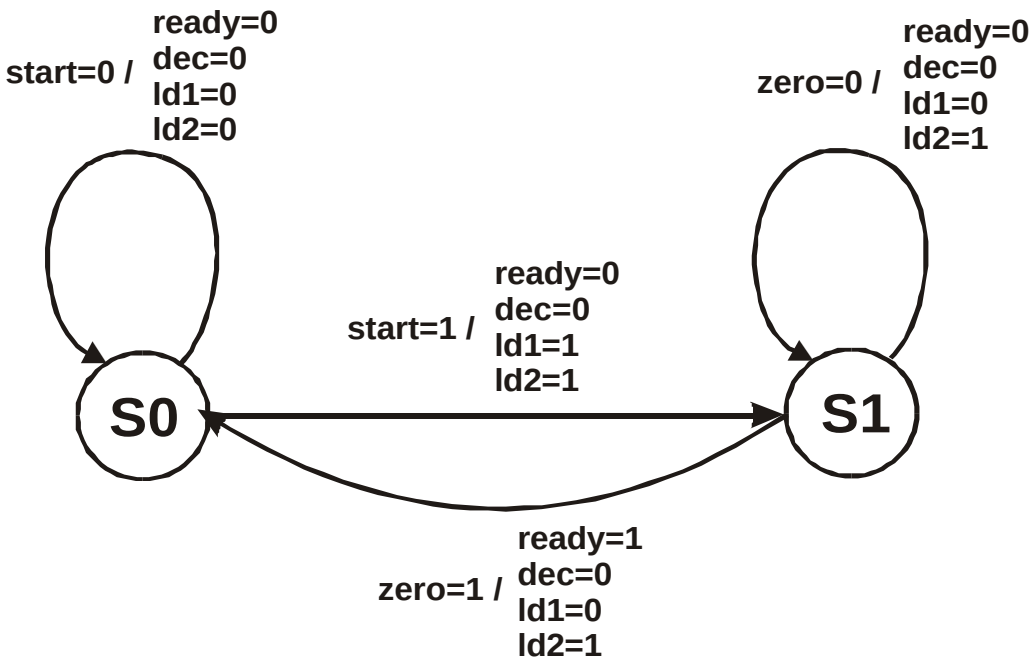


Figura 10.4: Graful de tranzitie al automatului

```
-- file: automat.fsm

ENTITY automat IS
  PORT(
         reset :IN bit;
         start :IN bit;
            ck :IN bit;
          zero :IN bit;
           vdd :IN bit;
           vss :IN bit;
           ld1 :OUT bit;
           ld2 :OUT bit;
           dec :OUT bit;
         ready :OUT bit
      );
END automat;


ARCHITECTURE automat_a OF automat IS

TYPE STATE_TYPE IS (S0, S1);

 — pragma CLOCK ck
 — pragma CUR_STATE CURRENT_STATE
 — pragma NEX_STATE NEXT_STATE


SIGNAL CURRENT_STATE, NEXT_STATE: STATE_TYPE;

BEGIN
   PROCESS (CURRENT_STATE, start, reset, zero)
  BEGIN

     IF(reset = '1') THEN
       NEXT_STATE <= S0;
       ld1 <= '0';
       ld2 <= '0';
       dec <= '0';
       ready <= '0';

     ELSE
      CASE CURRENT_STATE IS

        WHEN S0 =>
         IF(start = '1') THEN
           NEXT_STATE <= S1;
           ld1 <= '1';
```

```
          ld2 <= '1';
          dec <= '0';
          ready <= '0';
        ELSE
          NEXT_STATE <= S0;
          ld1 <= '0';
          ld2 <= '0';
          dec <= '0';
          ready <= '0';
        END IF;

       WHEN S1 =>
        IF(zero = '0') THEN
          NEXT_STATE <= S1;
          ld1 <= '0';
          ld2 <= '1';
          dec <= '1';
          ready <= '0';
        ELSE
          NEXT_STATE <= S0;
          ld1 <= '0';
          ld2 <= '0';
          dec <= '0';
          ready <= '1';
        END IF;

     END CASE;
    END IF;
  END PROCESS;

  PROCESS(ck)
  BEGIN
     IF(ck='1' and not ck'STABLE) THEN
       CURRENT_STATE <= NEXT_STATE;
    END IF;
  END PROCESS;
END automat_a;
```

10.3.3 Generarea descrierilor structurale

Pornind de la fisierele cu descrierile comportamentale (*reg4.vbe*, *reg8.vbe*, *sum4.vbe* si *count4.vbe*) se genereaza descrierile structurale (fisiere cu extensia vst). Operatiunea se face prin executarea utilitarului **scmap**.

```
scmap  reg4 reg4
```

În cazul automatului de control, *control.fsm*, apare o etapa suplimentara: convertirea subsetului fsm în subsetul vbe, prin folosirea utilitarului **syf**:

```
syf –M –c control control
```

Atât descrierile structurale generate, cât si cele comportamentale trebuie verificate cu ajutorul unul set de vectori de test care pot fi gasiti în fisierele cu acelasi nume ca si descrierile, dar cu extensia .PAT.

La capatul acestei etape, în directorul de lucru trebuie sa existe urmatoarele fisiere:

```
reg4.vbe, reg4.vst, reg8.vbe, reg8.vst, sum4.vbe, sum4.vst,
count4.vbe, count4.vst, control.fsm, control.vbe, control.vst.
```

10.3.4 Legarea descrierilor structurale

Revenind la structurile din figurile 10.1 si 10.2, odata avute toate componentele descrise structural, ele trebuie legate astfel încât sa formeze cele doua unitati functionale, calea de control si cea de date. Pentru început trebuie legate componentele caii de date conform schemei din figura 10.1. Descrierea structurala aferenta este listata în fisierul *date.vst*:

```
--file:  date.vst

ENTITY date IS
  PORT     a :IN bit_vector(3 DOWNTO 0);
           b :IN bit_vector(3 DOWNTO 0);
           ck :IN bit;
         reset :IN bit;
           ld1 :IN bit;
           ld2 :IN bit;
           dec :IN bit;
           vdd :IN bit;
           vss :IN bit;
         ready :IN bit;
          zero :INOUT bit;
          prod :INOUT bit_vector(7 DOWNTO 0));
END date;

ARCHITECTURE date_struct OF date IS

      COMPONENT num4
        PORT(    vdd :IN bit;
                 vss :IN bit;
```

```
              d :IN bit_vector(3 DOWNTO 0);
            reset :IN bit;
               ck :IN bit;
             load :IN bit;
              dec :IN bit;
             zero :INOUT bit);
    END COMPONENT;


    COMPONENT reg4
      PORT(   vdd :IN bit;
              vss :IN bit;
               d :IN bit_vector(3 DOWNTO 0);
            reset :IN bit;
               ck :IN bit;
             load :IN bit;
              q :OUT bit_vector(3 DOWNTO 0));
    END COMPONENT;


    COMPONENT reg8
      PORT(   vdd :IN bit;
              vss :IN bit;
               d :IN bit_vector(7 DOWNTO 0);
            reset :IN bit;
               ck :IN bit;
             load :IN bit;
              q :OUT bit_vector(7 DOWNTO 0));
    END COMPONENT;


    COMPONENT sum8
      PORT(  vdd :IN bit;
             vss :IN bit;
              a :IN bit_vector(3 DOWNTO 0);
              b :IN bit_vector(7 DOWNTO 0);
             zero:IN bit;
              q :OUT bit_vector(7 DOWNTO 0));
    END COMPONENT;


SIGNAL b1 : bit_vector(3 DOWNTO 0);
SIGNAL p1 : bit_vector(7 DOWNTO 0);
SIGNAL p2 : bit_vector(7 DOWNTO 0);

BEGIN
```

```
num: num4
  PORT MAP(
              vdd => vdd,
              vss => vss,
                d => a,
            reset => reset,
               ck => ck,
             load => ld1,
              dec => dec,
             zero => zero
           );
reg1: reg4
  PORT MAP(
              vdd => vdd,
              vss => vss,
                d => b,
            reset => reset,
               ck => ck,
             load => ld1,
                q => b1
          );
reg2: reg8
  PORT MAP(
              vdd => vdd,
              vss => vss,
                d => p1,
            reset => reset,
               ck => ck,
             load => ld2,
                q => p2
          );
reg3: reg8
  PORT MAP(
              vdd => vdd,
              vss => vss,
                d => p1,
            reset => reset,
               ck => ck,
             load => ready,
                q => prod
          );

sum: sum8
  PORT MAP(
              vdd => vdd,
              vss => vss,
                a => b1,
```

```
                b => p2,
             zero => zero,
                q => p1
           );
END  date_struct;
```

Fisierul *core.vst* descrie legarea caii de date cu cea de control:

```
-- file: core.vst

ENTITY mult IS
  PORT(    a   :IN bit_vector(3 DOWNTO 0);
           b   :IN bit_vector(3 DOWNTO 0);
          ck   :IN bit;
       reset   :IN bit;
       start   :IN bit;
         vdd   :IN bit;
         vss   :IN bit;
       ready   :INOUT bit;
        prod   :INOUT bit_vector(7 DOWNTO 0));
END mult;

ARCHITECTURE mult_struct OF mult IS

    COMPONENT date
      PORT(
            a :IN bit_vector(3 DOWNTO 0);
            b :IN bit_vector(3 DOWNTO 0);
             ck :IN bit;
          reset :IN bit;
            ld1 :IN bit;
            ld2 :IN bit;
            dec :IN bit;
            vdd :IN bit;
            vss :IN bit;
          ready :IN bit;
           zero :OUT bit;
          prod :INOUT bit_vector(7 DOWNTO 0)
           );
    END COMPONENT;

    COMPONENT control
      PORT (
              reset : in BIT; -- reset
              start : in BIT; --start
                 ck : in BIT; -- ck
               zero : in BIT; -- q
```

```
               vdd : in BIT;--vdd
               vss : in BIT;--vss
               ld1 : out BIT;--ld1
               ld2 : out BIT;--ld2
               dec :out BIT;  --dec
              ready : out BIT--ready
            );
     END COMPONENT;

SIGNAL l1, l2, dec1, zero1: bit;

BEGIN

  d: date
    PORT MAP(
               a => a,
               b => b,
              ck => ck,
           reset => reset,
             ld1 => l1,
             ld2 => l2,
             dec => dec1,
             vdd => vdd,
             vss => vss,
           ready => ready,
            zero => zero1,
            prod => prod
            );

  c: control
    PORT MAP(
              reset => reset,
              start => start,
                 ck => ck,
               zero => zero1,
                vdd => vdd,
                vss => vss,
                ld1 => l1,
                ld2 => l2,
                dec => dec1,
              ready => ready
            );

END mult_struct;
```

10.3.5. Generarea layout-ului simbolic

Pe baza descrierii structurale a core-ului circuitului, se poate testa întreg sistemul cu ajutorul unui set de vectori de test si a utilitarului **asimut**. Ulterior se genereaza layoutul simbolic:

```
scr –p –r –i 1000 core
```

Rezultatul (*core.ap*) poate fi vizualizat cu utilitarul **graal**.

10.3.6 Legarea pad-urilor

Setul de pad-uri de intrare, iesire, de ceas si de alimentare poate fi legat de core prin doua metode: fie în mod clasic, printr-o descriere structurala, fie folosind o etapa suplimentara în care, pe baza unor primitive C se genereaza automat descrierea structurala. A doua metoda este în cazul de fata (16 pad-uri) mai simpla si în plus genereaza un cod de o lizibilitate sporita. Fisierul este *mult.c*:

```
-- file: mult.c

#include <genlib.h>
main()
{
int i;

    DEF_LOFIG("mult");

LOCON("vdd",            IN,         "vdd");
LOCON("vss",            IN,         "vss");
LOCON("vdde",           IN,        "vdde");
LOCON("vsse",           IN,        "vsse");
LOCON("a[3:0]",         IN,      "a[3:0]");
LOCON("b[3:0]",         IN,      "b[3:0]");
LOCON("reset",          IN,       "reset");
LOCON("start",          IN,       "start");
LOCON("ck",             IN,          "ck");
LOCON("ready",         OUT,       "ready");
LOCON("prod[7:0]",     OUT,   "prod[7:0]");

LOINS ("pvsse_sp", "p20", "cki", "vdde", "vdd", "vsse", "vss",
0);
LOINS ("pvdde_sp", "p21", "cki", "vdde", "vdd", "vsse", "vss",
0);
```

```
LOINS ("pvddeck_sp", "p22", "clock", "cki", "vdde", "vdd",
"vsse", "vss",0);
LOINS ("pvssi_sp", "p23", "cki", "vdde", "vdd", "vsse", "vss",
0);
LOINS ("pvddi_sp", "p24", "cki", "vdde", "vdd", "vsse", "vss",
0);

    for (i = 0; i < 4; i++)
      LOINS("pi_sp", NAME("p%d", i),
            NAME("a[%d]", i), NAME("aa[%d]", i),
           "cki", "vdde", "vdd", "vsse", "vss", 0);

    for (i = 0; i < 4; i++)
      LOINS("pi_sp", NAME("p%d", i + 4),
            NAME("b[%d]", i), NAME("bb[%d]", i),
           "cki", "vdde", "vdd", "vsse", "vss", 0);

    for (i = 0; i < 8; i++)
      LOINS("po_sp", NAME("p%d", i + 8),
            NAME("prodprod[%d]", i), NAME("prod[%d]", i),
           "cki", "vdde", "vdd", "vsse", "vss", 0);

    LOINS("pi_sp", "p16",
          "start", "startstart",
          "cki", "vdde", "vdd", "vsse", "vss", 0);

    LOINS("pi_sp", "p17",
          "reset", "resetreset",
          "cki", "vdde", "vdd", "vsse", "vss", 0);

    LOINS("pck_sp", "p18",
         "ck",
          "cki", "vdde", "vdd", "vsse", "vss", 0);

    LOINS("po_sp", "p19",
          "readyready", "ready",
          "cki", "vdde", "vdd", "vsse", "vss", 0);

    LOINS("core", "core",
          "vdd", "vss",
           "aa[3:0]", "bb[3:0]", "startstart", "resetreset",
"clock",
           "readyready", "prodprod[7:0]", 0);

    SAVE_LOFIG();
    exit(0);
}
```

Descrierea structurala, continând si padurile, poate fi generata, pornind de la fisierul anterior, cu ajutorul utilitarului **genlib**:

```
genlib mult
```

Înainte de generarea layout-ului simbolic trebuie verificat daca exista si fisierul cu lazout-ul simbolic al core-ului *(core.ap)*. Pentru generarea layout-ului mai trebuie doar specificata pozitia pad-urilor pe cele patru laturi ale circuitului in fisierul *mult.rin*:

```
#file: mult.rin

width (vdd 20 vss 20)
west (p0 p1 p2 p3 p4 p5 p6 p7)
north (p16 p17 p18 p19)
east (p8 p9 p10 p11 p12 p13 p14 p15)
south (p20 p21 p23 p24 p22)
```

Utilitarul pentru plasarea padurilor este **ring**, fisierul rezultat fiind un layout simbolic:

```
ring mult mult
```

Layout-ul simbolic (*mult.ap*) poate fi vizualizat cu utilitrul **graal** si ulterior convertit la un layout real cu **s2r**.

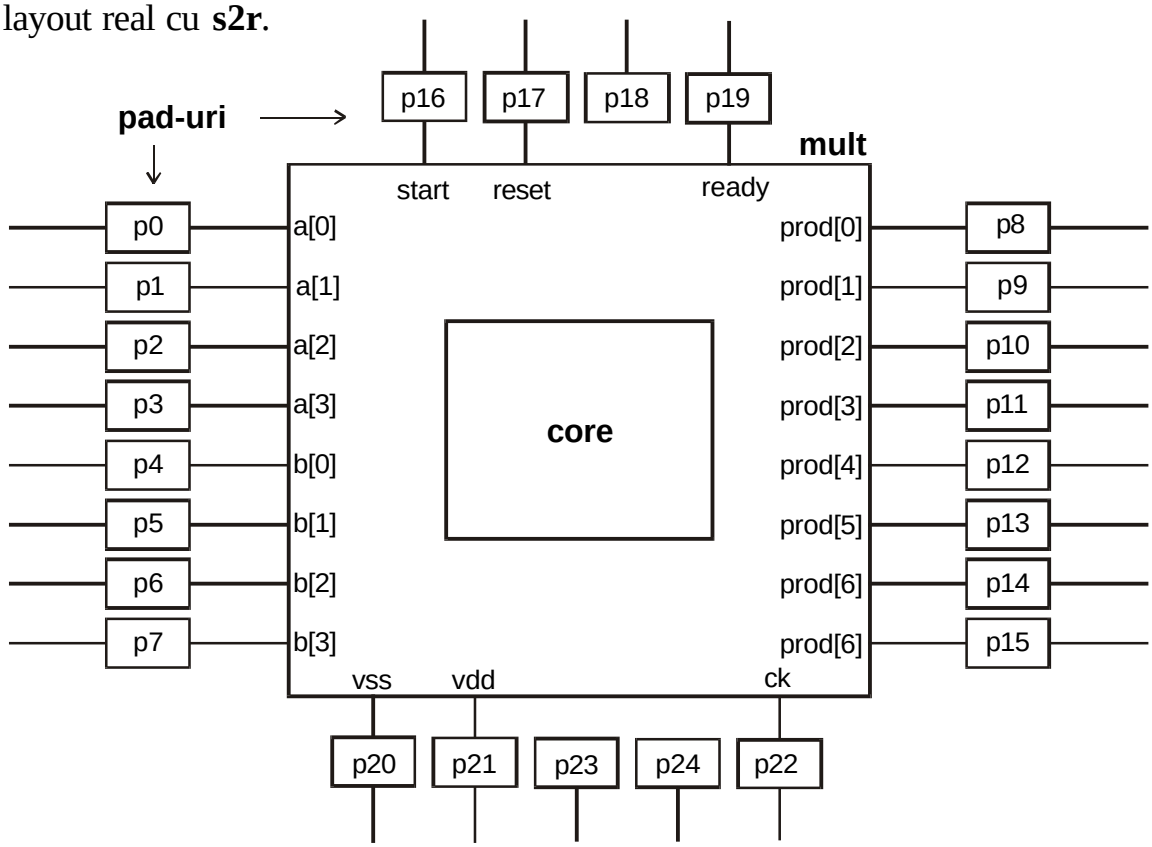


Figura 10.5: Distribuirea pad-urilor pe laturile circuitului