VHDL

Sintetizarea Sistemelor Digitale

Ing. Zărnescu George

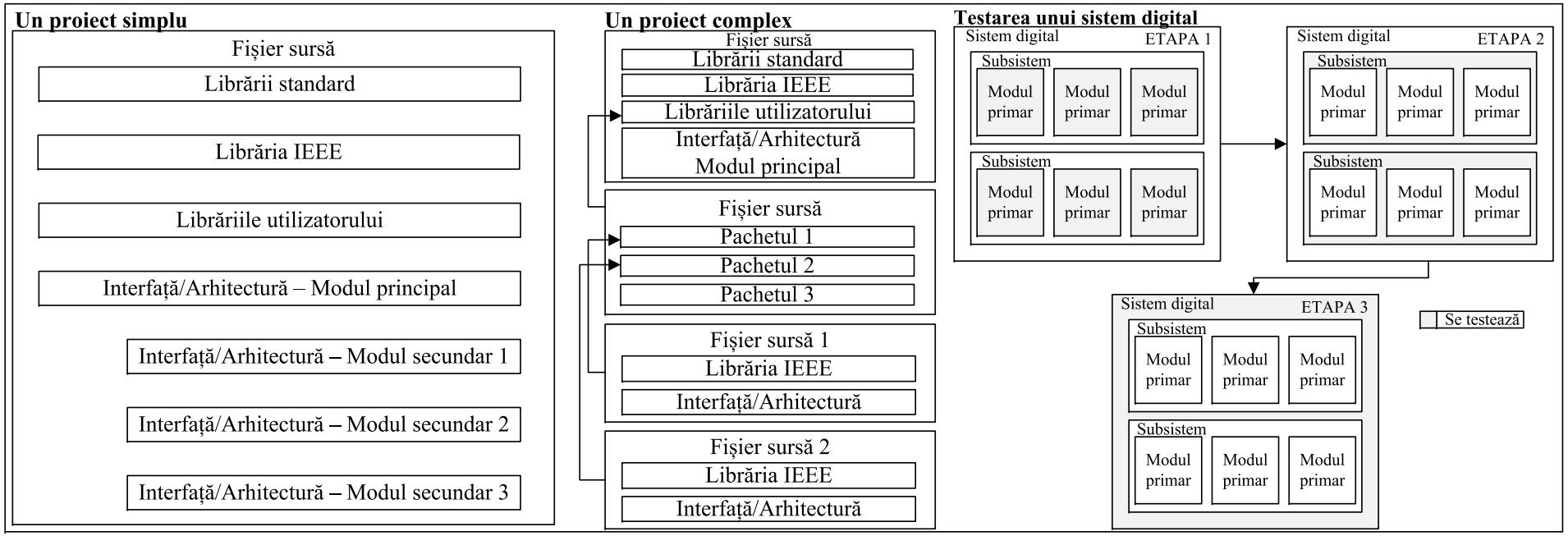
„However, the vast majority of commercial systems are now built using HDLs rather than schematics.”

D. Harris, S.Harris, Digital Design and Computer Architecture.

Cuprins

	Prelaborator	Laboratorul propriu-zis	Tutoriale
Laborator SD	**Îndrumar VHDL – Sintetizarea sistemelor digitale**	**Îndrumar sisteme digitale**	**Tutorial Altera Quatus**
Laboratorul 1		Introducere, Introducere în sistemele digitale	Introduction to Quartus II – Schematic *
Laboratorul 2	Capitolul 1; Anexa 2.		Introduction to Quartus II – VHDL*
Laboratorul 3	Capitolul 2 – 2.1, 2.2, 2.3, 2.4, 2.5.1.	Implementarea circuitelor logice combinaționale	
Laboratorul 4	Capitolul 2 – 2.5.1, 2.5.2, 2.5.3, 2.5.4, 2.5.5, 2.6, 2.7; Anexa 5.	Implementarea multiplexoarelor și a decodificatoarelor	
Laboratorul 5	Capitolul 2 – 2.5.6; Anexa 3 – A.3.4.	Implementarea circuitelor numerice de afișare	Quartus II Simulation - VHDL**
Laboratorul 6	Capitolul 3 – 3.1, 3.2, 3.3.1, 3.3.2, 3.3.3, 3.3.4, 3.3.5, 3.3.6, 3.3.7.	Implementarea circuitelor logice secvențiale	SignalTap II*, Debugging VHDL Hardware Designs – capitolul 1, 2, 3, 4.1.1, 4.1.2**
Laboratorul 7	Capitolul 3 – 3.3.8, 3.4.	Implementarea numărătoarelor	
Laboratorul 8	Capitolul 4; Anexa 1.	Proiecte I	Debugging VHDL Hardware Designs – capitolul 1, 2, 3, 4.2, 4.3, 5, 6, 7, 8**
Laboratorul 9	Capitolul 5; Anexa 3 – A.3.1, A.3.2, A.3.3.	Implementarea circuitelor aritmetico-logice	
Laboratorul 10	Capitolul 6.	Implementarea mașinilor cu stări finite	Timing Considerations – VHDL*, Debugging VHDL Hardware Designs – capitolul 1, 2, 3, 4.1.3**
Laboratorul 11	Capitolul 7, Anexa 4.	Implementarea controlerelor pentru memorii	
Laboratorul 12	Capitolul 9	Introducere în arhitectura MIPS	
Laboratorul 13	Capitolul 8	Proiecte II	Using Library Modules**
Laboratorul 14		Proiecte II	

* Tutorialele vor fi executate în laborator. ** Tutorialele vor fi executate acasă.



Regula celor 3 E

Ierarhizare
Modularitate
Regularitate

Organizarea standard a unui modul VHDL

Librării standard
Librării specifice

Interfață

Arhitectură

Declararea unei librării
```
LIBRARY numele_librăriei;
USE numele_librăriei.numele_pachetului.fragmentele_pachetului;
```

Declararea unei interfețe (entități)
```
ENTITY numele_interfeței is
   PORT (
            numele_portului : modul_semnalului tipul_semnalului;
            numele_portului : modul_semnalului tipul_semnalului;
            numele_portului : modul_semnalului tipul_semnalului;
            ...);
END numele_interfeței;
```

Declararea unei arhitecturi
```
ARCHITECTURE numele_arhitecturii OF numele_interfeței is
         [declarații semnale]
         [declarații constante]
         [declarații tipuri de date]
         [declarații componente]
         [declarații atribute]
BEGIN
          comportamentul modulului (corpul arhitecturii)
                              .
                              .
                              .
END numele_arhitecturii;
```

Declararea unei componente
```
COMPONENT numele_componentei
   PORT (
            numele_semnalului : modul_semnalului tipul_semnalului;
            numele_semnalului : modul_semnalului tipul_semnalului;
            numele_semnalului : modul_semnalului tipul_semnalului;
            ...);
END numele_componentei;
```

Utilizarea unei componente
```
numele_instanței : numele_componentei PORT MAP (
                        numele_semnalului_componentei => numele_semnalului_modulului,
                        numele_semnalului_componentei => numele_semnalului_modulului,
                                          .
                                          .
                                          .
                    numele_semnalului_componentei => numele_semnalului_modulului );
```

Declararea unui pachet
```
-- Partea declarativă
PACKAGE numele_pachetului is
          declararea tipurilor de date
          declararea constantelor
          declararea semnalelor globale
          declararea componentelor
          declararea atributelor
          declararea funcțiilor
          declararea procedurilor
END numele_pachetului;

-- Corpul pachetului
PACKAGE BODY numele_pachetului is
descrierea comportamentului unei funcții
END numele_pachetului;
```

Declararea unui semnal intern
```
architecture circ_arch of circ is

SIGNAL denumirea_semnalului : tipul_semnalului;

begin
...
end circ_arch;
```

Caracteristica generic
```
ENTITY numele_interfeței is
   GENERIC (WIDTH : INTEGER  N);
   PORT (
            numele_portului : modul_semnalului tipul_semnalului;
            numele_portului : modul_semnalului tipul_semnalului;
            numele_portului : modul_semnalului tipul_semnalului;
            ...);
END numele_interfeței;
```

Caracteristica generic map
```
-- declararea unei componente
COMPONENT numele_componentei
   GENERIC (WIDTH : INTEGER);
   PORT (
            numele_semnalului : modul_semnalului tipul_semnalului;
            numele_semnalului : modul_semnalului tipul_semnalului;
            numele_semnalului : modul_semnalului tipul_semnalului;
            ...);
END numele_componentei;

-- utilizarea unei componente
numele_instanței : numele_componentei GENERIC MAP (M)
                   PORT MAP (
                        numele_semnalului_componentei => numele_semnalului_modulului,
                        numele_semnalului_componentei => numele_semnalului_modulului,
                                          .
                                          .
                                          .
                    numele_semnalului_componentei => numele_semnalului_modulului );
```

FOR generate
```
numele_declarației_generate:
FOR index IN mulțime_de_valori GENERATE
        declarații;
        declarații;
        declarații;
            .
            .
            .
        declarații;
END GENERATE;
```

IF generate
```
numele_declarației_generate:
IF expresie_logică GENERATE
        declarații;
        declarații;
        declarații;
            .
            .
            .
        declarații;
END GENERATE;
```

Tipul de date array
```
ARCHITECTURE ...
TYPE mem IS ARRAY(R-1 DOWNTO 0) OF STD_LOGIC_VECTOR(C-1 DOWNTO 0);
SIGNAL mem_array : mem;
```

Atribuire selectată
```
WITH semnal(expresie)_logic(ă) SELECT
nume_semnal <= semnal(expresie)_logic(ă) WHEN valoare_constantă,
               semnal(expresie)_logic(ă) WHEN valoare_constantă,
                                  .
                                  .
                                  .
               semnal(expresie)_logic(ă) WHEN OTHERS;
```

Tipul de date std_logic și std_logic_vector
```
a : in std_logic;
d : in std_logic_vector(0 to 7);
e : out std_logic_vector(3 downto 0);
```

Atribuire condițională
```
nume_semnal <= semnal(expresie)_logic(ă) WHEN condiție_logică ELSE
               semnal(expresie)_logic(ă) WHEN condiție_logică ELSE
                                  .
                                  .
                                  .
               semnal(expresie)_logic(ă) (WHEN OTHERS);
```

Tipul de date enumeration
```
architecture...

TYPE states IS (S0, S1, S2, S4);
SIGNAL current_state : states;
SIGNAL next_state : states;
```

Declararea unui process
```
PROCESS ( numele_semnalului, numele_semnalului ... )
BEGIN
        [declarații simple]
        [declarații if/elsif/else]
        [declarații case]
END PROCESS [ numele procesului ];
```

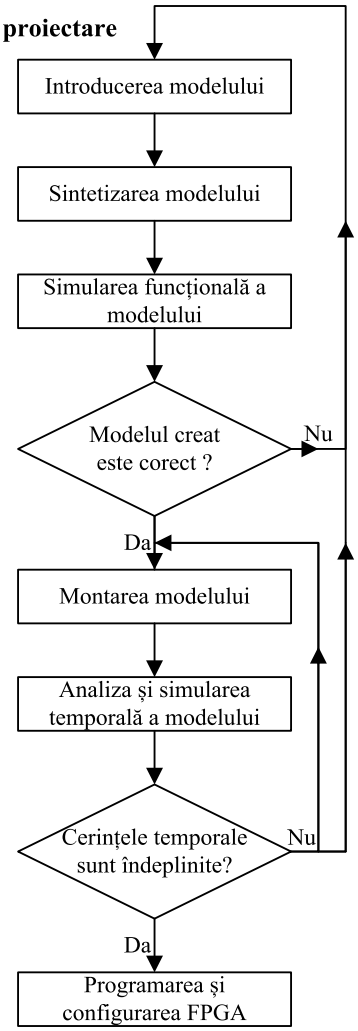
Declarația secvențială if/elsif/else
```
IF prima_condiție_logică THEN
        declarații;
        declarații;
ELSIF a_doua_condiție_logică THEN
        declarații;
        declarații;
ELSIF …
            .
            .
            .
ELSE   declarații;
        declarații;
END IF;
```

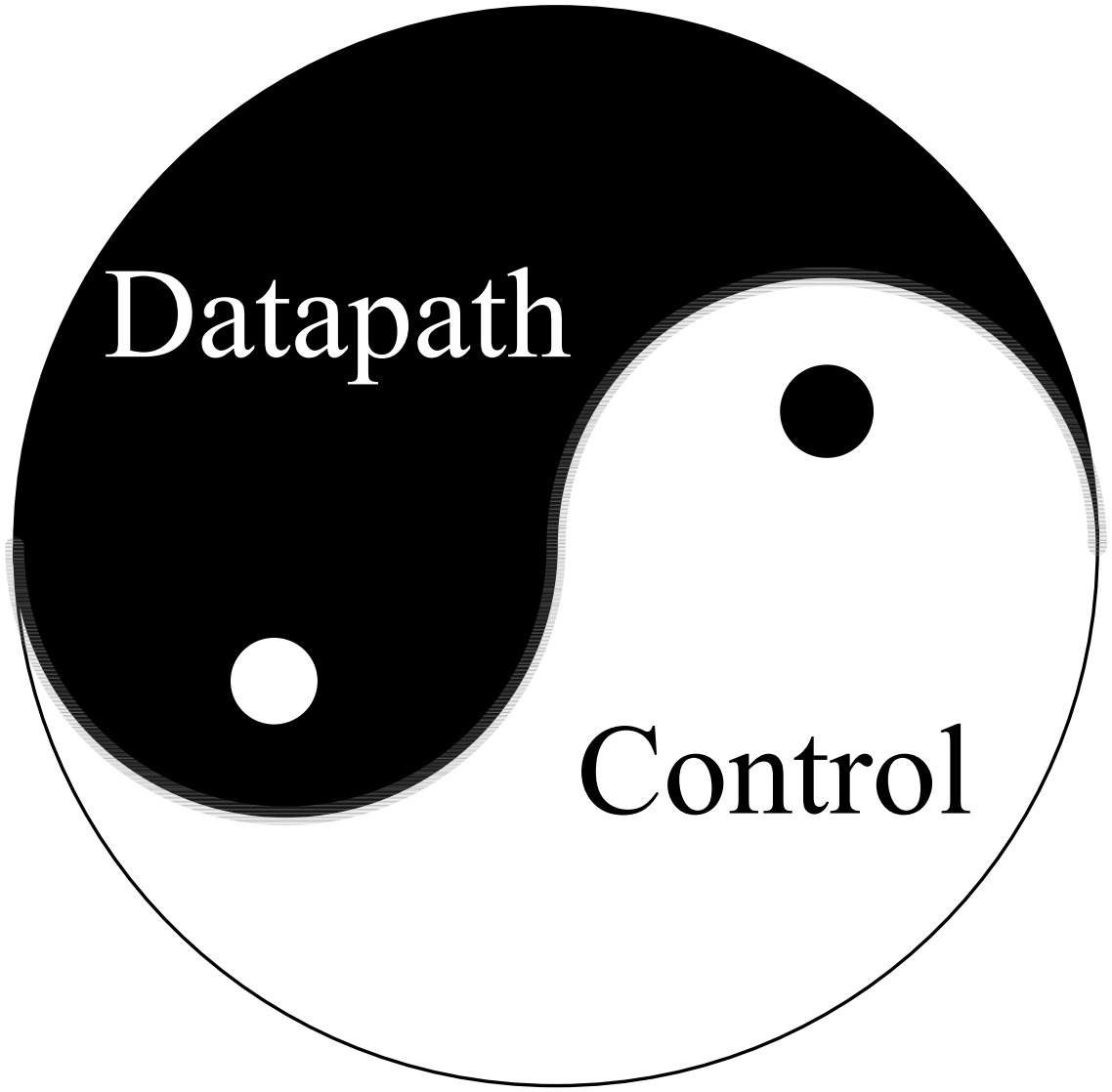
Declarația secvențială case
```
CASE expresie_logică IS

    WHEN valoare_constantă =>
            declarații;
            declarații;
    WHEN valoare_constantă =>
            declarații;
            declarații;
                .
                .
                .
    WHEN OTHERS =>
            declarații;
            declarații;

END CASE;
```

Operator	Simbolul operatorului	Operația efectuată
de atribuire	<=, =>	atribuire de valori
logic	NOT, AND, OR, NAND, NOR, XOR, XNOR	not, and, or, not and, not or, xor, not xor
relațional	=, /=, >, <, >=, <=	egalitate, inegalitate, mai mare, mai mic, mai mare sau egal, mai mic sau egal
aritmetic	+, -, *, /	adunare, scădere, înmulțire, împărțire
de deplasare	sll, srl, sla, sra, rol, ror	deplasare logică la stânga, deplasare logică la dreapta, deplasare aritmetică la stânga, deplasare aritmetică la dreapta, rotire la stânga, rotire la dreapta,
de concatenare	&	concatenare

Operator	Simbolul operatorului	Prioritate
diverși	NOT	ridicată
aritmetic	*, /	
aritmetic și de concatenare	+, -, &	
de deplasare	sll, srl, sla, sra, rol, ror	
relaționali	=, /=, <, <=, >, >=	
logic	AND, OR, NAND, NOR, XOR, XNOR	scăzută

Metodologia de proiectare




Datapath
Control

Împărțirea capitolelor în funcție de Datapath (calea de date) și Control (unitatea de control)

Capitolul 1 – Introducere în limbajul VHDL	Introducere
Capitolul 2 – Sintetizarea circuitelor logice combinaționale prima parte	Datapath (calea de date)
Capitolul 3 – Sintetizarea circuitelor logice secvențiale sincrone	
Capitolul 4 – Sintetizarea circuitelor logice combinaționale partea a doua	
Capitolul 5 – Sintetizarea circuitelor aritmetico-logice	
Capitolul 7 – Sintetizarea memoriilot și a controlerelor pentru memorii	
Capitolul 8 – Module parametrizabile	
Capitolul 6 – Sintetizarea mașinilor cu stări finite	Control (unitatea de control)
Capitolul 9 – Implementarea unui procesor MIPS single-cycle pe 32 de biți	Datapath & Control

Introducere

Limbajele de programare sunt organizate în două mari categorii: limbaje de programare software şi limbaje de descriere hardware. La început existau ingineri electronişti sau programatori specialişti doar în programare software sau doar în descriere hardware. La momentul actual inginerii electronişti sau programatorii cunosc un limbaj de descriere hardware şi cel puţin un limbaj de programare software.

Există două limbaje de descriere hardware: VHDL şi Verilog. Fiecare inginer electronist sau programator foloseşte, după preferinţă, un anumit limbaj. Limbajele de descriere hardware dau posibilitatea evidenţierii în cuvinte, sub forma unui program, a comportamentului circuitelor logice. Acest îndrumar va prezenta limbajul de descriere hardware VHDL. Se va prezenta în detaliu modul de sintetizare şi verificare al circuitelor logice.

VHDL este un limbaj elaborat de către o comisie şi de aceea se spune despre el că este prolix, adică descrierea unui circuit (modul) se exprimă cu prea multe cuvinte şi că este complicat în comparaţie cu limbajul Verilog. Companiile de telecomunicaţii, organizaţia ESA (European Space Agency – Agenţia Spaţială Europeană) sau contractanţii cu care armata S.U.A. colaborează, folosesc intens limbajul de descriere harware VHDL.

VHDL înseamnă VHSIC Hardware Description Language, iar VHSIC înseamnă Very High Speed Integrated Circuits, deci putem descrie şi implementa circuite integrate de mare viteză cu ajutorul unui limbaj de descriere hardware. VHSIC este un program al departamentului de apărare al S.U.A.. În anul 1981 departamentul de apărare a dezvoltat limbajul VHDL pentru a descrie structura şi funcţionarea circuitelor logice. În anul 1987 IEEE a standardizat limbajul VHDL (IEEE STD 1076). Până în acest moment acest limbaj a fost actualizat de câteva ori. Se dorea ca acesta să fie folosit pentru documentaţia sistemelor digitale, dar a fost adoptat foarte repede pentru simularea şi sintetizarea acestora.

Pentru a crea un circuit logic se vor folosi software-urile CAD existente. Acestea oferă posibilitatea sintetizării şi simulării circuitelor digitale. Comportamentul acestor circuite este descris printr-un program VHDL, introdus spre procesare în software-ul CAD. Procesul de sintetizare transformă cuvintele din programul VHDL într-un circuit logic.

Pentru a crea un circuit logic este recomandată parcurgerea a două etape. În prima etapă se vor schița pe o foaie de hârtie modulele care alcătuiesc circuitul logic și felul în care ele sunt conectate, deci este nevoie să se cunoască foarte bine circuitul. În a doua etapă se vor descrie în limbaj VHDL modulele circuitului logic și conexiunile dintre acestea. Sintetizatorul nu cunoaște circuitul ce se dorește a fi implementat, el doar interpretează cuvintele din programul VHDL și le transformă în hardware. Înainte ca circuitul logic să fie implementat comportamentul acestuia va fi simulat pentru a se găsi și remedia eventuale erori, denumite în limbaj de specialitate bug-uri.

Sisteme logice de complexitate redusă sau medie, care erau implementate în urmă cu 20 de ani în câteva zile, în momentul de față, se implementează în câteva minute sau ore cu ajutorul software-lor CAD și a limbajelor de descriere hardware. În anii 80, sistemele digitale erau implementate cu ajutorul circuitelor integrate. În momentul de față, inginerii electroniști și programatorii au posibilitatea de a fi mai productivi, deoarece limbajele de descriere hardware oferă posibilitatea implementării unor circuite de zece ori mai mari ca cele implementate în anii 80.

Limbajul de descriere hardware și software-ul CAD reprezintă "ustensilele" unui inginer electronist sau ale unui programator contemporan. Metodologia de gestionarea a complexității circuitelor logice determină modul în care aceste ustensile se pot folosi în implementarea sistemelor digitale.

VHDL este un limbaj bogat cu multe comenzi. Doar o parte din comenzi se pot sintetiza în circuite logice. Acest îndrumar va evidenția setul de comenzi VHDL ce pot fi sintetizate în circuite logice. Motto-ul acestui îndrumar este: **Think hardware!**

Capitolul 1

Introducere în limbajul VHDL

Introducere în limbajul VHDL

În momentul de față limbajele de descriere hardware au înlocuit metodele clasice de implementare a circuitelor logice. Aceste limbaje oferă posibilitatea inginerilor electroniști sau programatorilor de a fi productivi și de a crea sisteme digitale foarte mari și foarte complexe. Gestionarea complexității este punctul forte al limbajului VHDL.

Sistemele digitale sunt alcătuite din porți logice și circuite de memorie, care la rândul lor sunt alcătuite din milioane de tranzistoare. Descrierea sistemelor la nivel de tranzistor presupune o atenție sporită la detalii.

Sistemele digitale pot fi organizate în module interconectabile. Comportamentul acestor module este descris în cuvinte. Cuvintele folosite reprezintă comenzi VHDL sintetizabile. Descrierea modulelor la acest nivel ridicat de abstractizare este foarte simplă!

1.1 Gestionarea complexității

Gestionarea complexității sistemelor digitale este posibilă utilizând o tehnică foarte importantă, **abstractizarea**: ascunderea detaliilor atunci când acestea nu sunt importante. Un sistem poate fi privit din perspectiva mai multor nivele de abstractizare.

Figura 1 evidențiază nivelele de abstractizare pentru un computer, componentele principale fiecărui nivel și limbajele de programare folosite pentru descrierea comportamentului fiecărei componente.

Nivelul cel mai scăzut de abstractizare este reprezentat de fizica cuantică. Componenta principală a acestui nivel este reprezentată de **mișcarea electronilor**. Această mișcare este descrisă de mecanica cuantică și ecuațiile lui Maxwell. Studiul comportamentului electronilor este foarte dificil, deoarece acești au mișcări foarte complexe. Modele matematice care descriu mișcărea electronilor pot fi introduse în software-ului Comsol. Acesta ajută la simularea comportamentului electronilor și perfecționarea modelelor matematice care descriu mișcarea acestora.

În general sistemele electronice sunt alcătuite din **dispozitive** ca diodele, tranzistoarele sau de mult apusele tuburi electronice. Comportamentul acestor dispozitive poate fi evidențiat prin caracteristicile curent-tensiune sau prin simulări cu ajutorul programului Spice.

Circuitele analogice se pot crea cu ajutorul dispozitivelor electronice. Diodele și tranzistoarele intră în componența revoluționarul amplificator operațional. Aceste amplificatoare au intrări și ieșiri bine definite și prelucrează semnale (curenți sau tensiuni) continue. În acest moment putem construi sisteme cu ajutorul amplificatoarelor operaționale fără să ne intereseze structura lor internă.

Circuitele digitale cum ar fi porțile logice folosesc două valori de tensiune. Pentru a reprezenta valoare logică 0 se va folosi o tensiune de 0 volți iar pentru a reprezenta valoarea logică 1 se va folosi o tensiune de 3, 5, 10 sau 15 volți. **Sistemele digitale** complexe cum ar fi sumatoarele, comparatoarele sau memoriile sunt alcătuite din circuite digitale.

Nivelul **microarhitecturi** face legătura între nivelele de abstractizare digitale și nivelul de abstractizare **arhitecturi**. Cele două părți componente ale unei microarhitecturi sunt calea de date și unitatea de control. Acestea sunt alcătuite cu ajutorul sistemelor digitale complexe.

Nivelul de abstractizare arhitecturi descrie un computer din perspectiva programatorului. Spre exemplu arhitectura MIPS, folosită de microprocesoarele din consolele Playstation, este definită printr-un set de instrucțiuni și registre (locul unde sunt păstrate provizoriu variabilele), pe care un programator le poate folosi. Microarhitecturile execută instrucțiunile unei arhitecturi. Instrucțiunile unei anumite arhitecturi pot fi executate de mai multe microarhitecturi ținându-se cont de compromisul dintre preț, performanță și consumul de energie.

Nivelurile circuite digitale, sisteme digitale și microarhitecturi vor fi prezentate în detaliu în acest îndrumar din perspectiva limbajului de descriere hardware VHDL.

Un **sistem de operare**, precum Windows-ul, gestionează accesarea harddisk-ului, a dispozitivelor de intrare-ieșire sau a memoriei principale. **Aplicațiile software** utilizează avantajele unui sistem de operare și ușurează munca unui utilizator, fie că dorește să editeze un text sau să comunice cu o persoană aflată în altă parte a lumii.

Aplicații software	Programe	Office Messenger
Sisteme de operare	Drivere	Windows
Arhitecturi	Instrucțiuni Registre	MIPS IA-32
Microarhitecturi	Cale de date Unitate de control	VHDL
Sisteme digitale	Sumatoare Comparatoare Memorii	VHDL
Circuite digitale	Porți logice NOT AND OR	VHDL
Circuite analogice	Amp Op Filtre	Spice
Dispozitive	Diode Tranzistoare	Spice
Fizica cuantică	Electroni	Comsol

Figura 1 – Nivelurile de abstractizare pentru un computer.

Atunci când se lucrează la un anumit nivel de abstractizare este favorabilă cunoașterea nivelului imediat următor și a nivelului imediat anterior.

În gestionarea complexității sistemelor digitale proiectanții, în plus față de tehnica abstractizării, mai respectă **regula celor trei E**.

- **Ierarhizare** – acestă regulă propune împărțirea sistemului digital în module, apoi divizarea acestor module în submodule până în momentul în care acestea sunt alcătuite din fragmente, al căror comportament este ușor de înțeles. Acestă regulă este evidențiată în figura 2 (vaza din ceară este alcătuită din postamentul, corpul și gâtul vazei).
- **Modularitate** – acestă regulă propune ca modulele să aibă un comportament și o interfață bine definite, astfel încât acestea să se conecteze între ele cu ușurință, fără să existe ulterior complicații neanticipate. Acestă regulă este evidențiată în figura 4 (modulul de bază este reprezentat de o celulă de forma unei prisme hexagonale).
- **Regularitate** – acestă regulă propune crearea unor module simple, care să fie reutilizate frecvent în diferite proiecte și reducerea numărului de module, care trebuie să fie proiectate. Acestă regulă este evidențiată în figura 3 (peretele corpului vazei este alcătuit din modulul de bază, adică celule de forma unor prisme hexagonale).

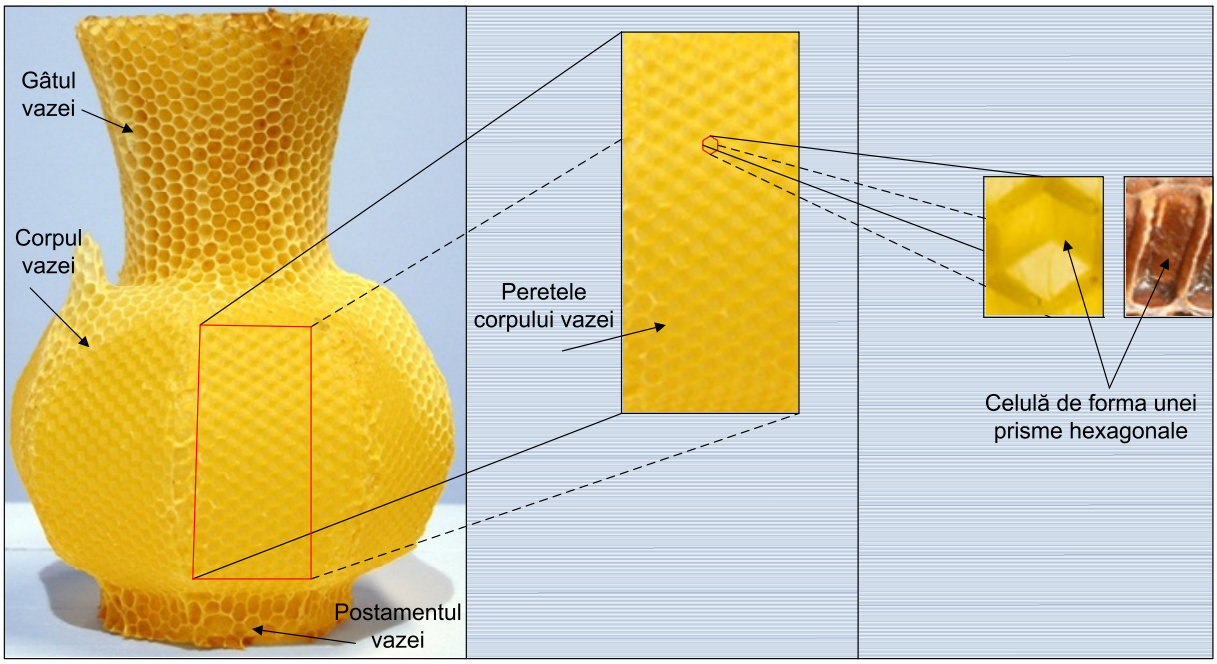


Figura 2 – Ierarhizare. **Figura 3 – Regularitate.** **Figura 4 – Modularitate.**

1.2 Structura unui program VHDL

Un program VHDL descrie comportamentul sau structura unui circuit logic. Un circuit logic cu o interfață (intrări și ieșiri), cu un comportament ideal descris prin limbajul VHDL și un comportament temporal bine definit, se numește **modul VHDL**. Există două stiluri prin care se poate descrie funcționarea unui modul VHDL: **stilul comportamental** și **stilul structural**. Stilul comportamental descrie relația intrare-ieșire (funcția logică) a unui modul. Stilul structural descrie felul în care un modul este alcătuit din module mai simple. Acest stil este o aplicație a ierarhizării. Adesea în cadrul proiectelor complexe cele două stiluri se vor combina. Un modul VHDL reprezintă o aplicație a modularității, adică modulul are o interfață bine definită și efectuează o anumită funcție. Felul în care a fost scris programul VHDL nu prezintă importanță pentru utilizatorii modulului atâta timp cât acesta îndeplinește funcția respectvă.

În figura 5.1 este reprezentată organizarea standard a unui modul VHDL, iar în figura 5.2 este descrisă funcția logică NAND. Modulul VHDL din figura 5.2 este descris în stil comportamental.

LIBRĂRII standard
LIBRĂRII specifice
	library ieee;
	use ieee.std_logic_1164.all;
	-- definirea unui comentariu
	-- entitatea nand
	entity nand is
	port (a, b : in std_logic;
	c : out std_logic);
	end nand;
	-- arhitectura nand_arch
	architecture nand_arch of nand is
	begin
	c <= a nand b;
	end nand_arch;

INTERFAȚĂ	
ARHITECTURĂ	

Figura 5.1 – Organizarea standard.

Figura 5.2 – Exemplu modul VHDL.

Un modul VHDL este alcătuit dintr-o listă de **librării**, o **interfață** unde sunt definite intrările și ieșirile modulului și o **arhitectură** unde este descris comportamentul acestuia. Cele 3 părți componente sunt evidențiate în continuare.

1.2.1 Librării

Librăriile sunt alcătuite din fragmente de cod folosite frecvent. Librăriile permit reutilizarea fragmentelor de cod în toate proiectele. În figura 6 este evidențiată organizarea unei librării. O librărie este alcătuită din pachete. Un pachet este alcătuit din tipuri de date, constante, semnale globale, funcții, proceduri și **componente**. Pentru a utiliza o librărie într-un modul trebuie respectate instrucțiunile din figura 7.

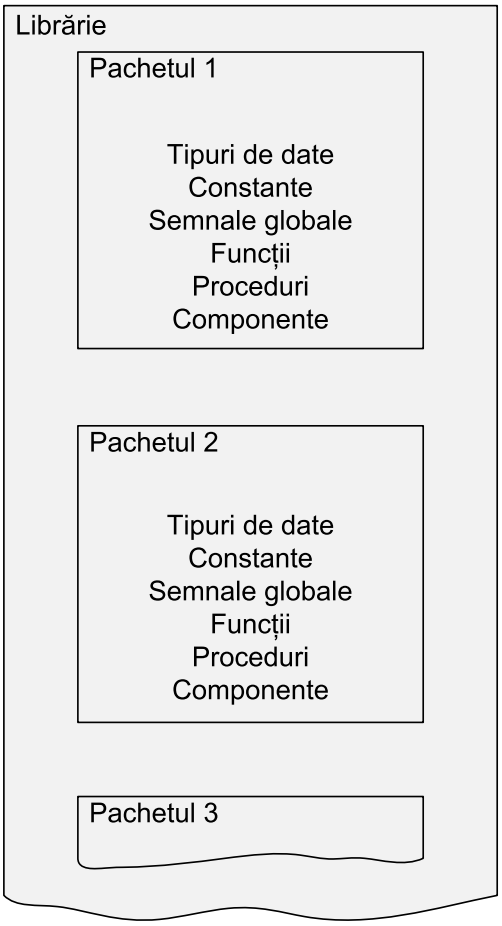


Figura 6 – Organizarea unei librării.

```
LIBRARY numele_librăriei;
USE numele_librăriei.numele_pachetului.fragmentele_pachetului;
```

Figura 7 – Declararea unei librării într-un modul.

O librărie standard este vizibilă implicit în orice modul. Aceste librării se numesc **std** și **work**. Librăriile specifice sunt create de instituții, cum ar fi societatea IEEE, și de utilizatori ai limbajului VHDL.

Într-un modul VHDL se va folosi cel puțin o librărie. În figura 5.2 este evidențiată instanțierea unei librării. Acestă librărie se numește **ieee**. Din acea librărie sunt necesare toate fragmentele de cod, de aceea s-a folosit comanda **all** în locul numelui unui fragment anume, din pachetul std_logic_1164.

Librăria ieee conține câteva pachete, printre care:

- **std_logic_1164** – acest pachet evidențiază tipurile de date **std_logic** și **std_logic_vector**, având 8 niveluri logice, și tipurile de date **std_ulogic** și **std_ulogic_vector**, având 9 niveluri logice. Tipuri de date std_logic și std_logic_vector se vor folosi pe parcursul întregului îndrumar și vor fi descrise în capitolul următor.
- **std_logic_arith** – în acest pachet sunt definite tipurile de date **signed** (numere întregi) și **unsigned** (numere naturale) împreună cu operațiile aritmetice specifice. De asemenea pachetul conține funcții de conversie a datelor: **conv_integer(s, b)**, **conv_unsigned(s, b)**, **conv_signed(s, b)**, **conv_std_logic_vector(s, b)**.
- **std_logic_signed** – acest pachet conține funcții ce permit efectuarea unor operații cu tipul de date std_logic_vector ca fiind date de tipul signed.
- **std_logic_unsigned** – acest pachet conține funcții ce permit efectuarea unor operații cu tipul de date std_logic_vector ca fiind date de tipul unsigned.

Toate aceste pachete împreună cu pachetul **numeric_std** sunt evidențiate în referința 12 din bibliografie. Modul în care utilizatorii limbajului VHDL pot crea librării este evidențiat în Anexa 1.

1.2.2 Interfață (entitate)

Într-o interfață sunt definite toate intrările și ieșirile unui modul VHDL. Pentru a defini o interfață trebuie respectate instrucțiunile din figura 8. Un exemplu de interfață este evidențiat în figua 5.2. Numele interfeței este nand. Semnalele modulului nand se numesc a, b, c și sunt de

tipul std_logic. Modul semnalelor a și b este in, fiind porturi de intrare, iar modul semnalului c este out, fiind port de ieșire.

```
ENTITY numele_interfeței is
   PORT (
            numele_portului : modul_semnalului tipul_semnalului;
            numele_portului : modul_semnalului tipul_semnalului;
            numele_portului : modul_semnalului tipul_semnalului;
            ...);
END numele_interfeței;
```

Figura 8 – Declararea unei entități într-un modul.

Numele entității sau numele porturilor pot fi alese ca nume obișnuite, exceptând cuvintele cheie ale limbajului VHDL. Aceste cuvinte cheie sunt evidențiate în Anexa 2. Modul unui semnal poate fi : **IN**, **OUT**, **BUFFER** și **INOUT**.

Modul IN este utilizat pentru un semnal de intrare. Modul OUT este utilizat pentru un semnal de ieșire și **nu** permite utilizarea valorilor semnalului **în interiorul** unui modul. Acest lucru înseamnă că semnalul poate apărea doar în partea stângă a operatorului <=. Modul BUFFER este utilizat pentru un semnal de ieșire și permite utilizarea valorilor semnalului în interiorul unui modul. Acest lucru înseamnă că semnalul poate apărea în ambele părți ale operatorului <=. Modurile IN, OUT și BUFFER sunt moduri unidirecționale. Modul INOUT este un mod bidirecțional, utilizat pentru un semnal care este și semnal de și semnal de ieșire. Dacă un port se definește fără un mod, atunci sintetizatorul va presupune acel port ca fiind de tip IN. Tipurile semnalelor utilizate într-un modul vor fi std_logic și std_logic_vector.

1.2.3 Arhitectură

Într-o arhitectură este descris comportamentul unui modul VHDL. Pentru a defini o arhitectură trebuie respectate instrucțiunile din figura 9. După cum se observă o arhitectură este alcătuită din două părți: o parte declarativă, opțională, unde sunt descrise semnale, constante, tipuri de date, componente sau atribute și o parte descriptivă unde este evidențiat

comportamentul modulului VHDL, de la cuvântul BEGIN în jos. În figura 5.2 este definită arhitectura interfeței nand. Această arhitectură poartă numele de arch_nand și nu deține parte declarativă. Comportamentul modulului este următorul: c<= a nand b.

```
ARCHITECTURE numele_arhitecturii OF numele_interfeței is
        [declarații semnale]
        [declarații constante]
        [declarații tipuri de date]
        [declarații componente]
        [declarații atribute]
BEGIN
            comportamentul modulului (corpul arhitecturii)
                                .
                                .
                                .
END numele_arhitecturii;
```

Figura 9 – Declararea unei arhitecturi într-un modul.

Numele arhitecturii poate fi ales ca nume obișnuit, exceptând cuvintele cheie ale limbajului VHDL.

1.3 Metodologia de proiectare

Implementarea sistemelor digitale într-un FPGA devine o sarcină foarte simplă în momentul în care un inginer electronist sau un programator respectă cu strictețe etapele metodologiei de proiectare. Acestă metodologia reprezintă un proces organizat de modelare, verificare și pregătire pentru fabricarea unui produs. Metodologia de proiectare și software-ul CAD înlesnesc procesul de implementare al sistemelor digitale. Acest proces devine unul automat și poartă numele de proiectare electronică automată (EDA – electronic design automation). Trebuie menționat că un modul este un circuit logic cu intrări și ieșiri și cu o interfață bine definită.

În figura 10 sunt evidențiate etapele metodologiei de proiectare pentru implementarea unui sistem digital în FPGA. Etapele acestui proces au următoarele semnificații:

- **Introducerea modulului** – circuitul dorit este descris printr-o diagramă schematică sau cu ajutorul limbajului de descriere hardware VHDL.
- **Sintetizarea modulului** – modulul introdus este sintetizat într-un circuit real alcătuit din elementele logice ale cipului FPGA.
- **Simularea funcțională a modulului** – circuitul sintetizat este testat pentru a se verifica dacă funcționează corect. Se analizează corectitudinea comportamentului ideal al circuitului. Acestă analiză nu ia în considerare comportamentul temporal al circuitului.
- **Montarea modulului** – software-ul CAD transformă elementele logice definite in netlist în elemente logice ale cipului FPGA și alege rutarea firelor pentru a face conexiunile necesare între elementele logice din FPGA.
- **Analiza temporală a modulului** – sunt analizate întârzierile prin propagarea semnalelor pe diferite căi ale circuitului montat pentru a se determina performanțele circuitului.
- **Simularea temporală a modulului** – circuitul montat este testat pentru a se verifica comportamentul ideal și cel temporal.
- **Programarea și configurarea FPGA** – în acest moment circuitul dorit poate fi implementat într-un FPGA prin programarea întrerupătoarelor configurabile. Acest lucru presupune configurarea elementelor logice utilizate și determinarea conexiunilor dintre acestea.

1.4 Organizarea unui proiect VHDL

1.4.1 Un proiect simplu

În figura 11 este evidențiată structura unui proiect simplu. Proiectanții debutanți sunt sfătuiți să creeze proiecte în acest fel. Un astfel de proiect conține un singur fișier sursă, fiind compus din librării standard, librăria ieee și librării create de utilizatori obișnuiți. Fișierul mai conține o interfață principală, unde sunt declarate intrările și ieșirile și o arhitectură, unde este descrisă funcționarea sistemului, structural sau comportamental. Figura 12 prezintă organizarea unui proiect simplu printr-un exemplu, circuit aritmetic adunare/scădere pentru numere întregi.

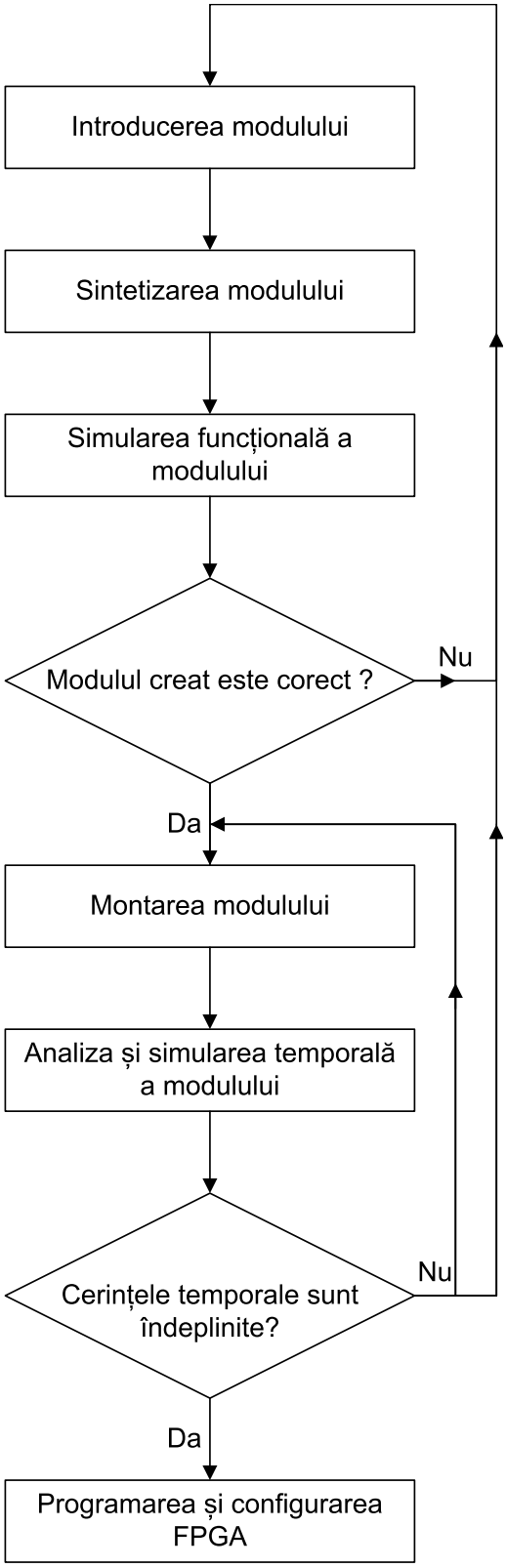


Figura 10 – Etapele metodologiei de proiectare.

În funcție de organizarea internă a sistemului digital se mai pot scrie module VHDL secundare în continuarea modulului principal. Aceste module secundare se vor instanția drept componente și se vor conecta în interiorul arhitecturii modulului principal. Instanțierea unei componente reprezintă capacitatea limbajului VHDL de a conecta referința unei componente într-o entitate cu declararea ei, care se află în altă parte (în același fișiere sau în alt fișier). Figura 13 evidențiază declararea și utilizarea unei componente.

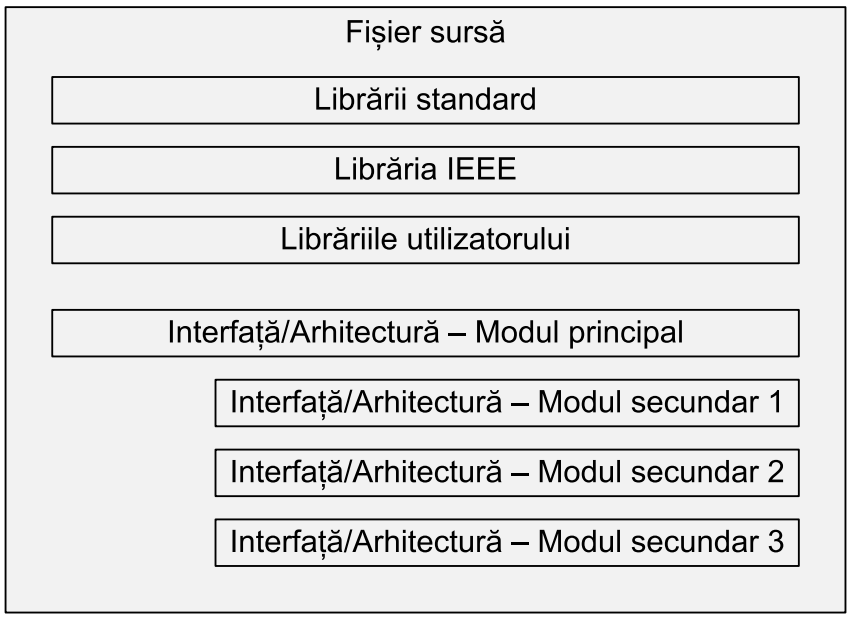


Figura 11 – Structura unui proiect simplu.

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

-- interfața modulului principal

entity addersubtractor is
        port (
                A, B    : in std_logic_vector(7 downto 0);
                AddSub : in std_logic;
                S          : out std_logic_vector(7 downto 0);
                Cout     : out std_logic
               );

end addersubtractor;
```

```
-- arhitectura modulului principal

architecture arch_addersubtractor of addersubtractor is

	signal B_intermed : std_logic_vector(7 downto 0);

	component add_sub
		 port(
			add_sub : in std_logic;
			dataa	: in std_logic_vector(7 downto 0);
			datab	: in std_logic_vector(7 downto 0);
			result	: out std_logic_vector(7 downto 0);
			Cout	: out std_logic
		       );
	end component;

	component xor_module
		port (
			sel	: in std_logic;
			dataa : in std_logic_vector(7 downto 0);
			result : out std_logic_vector(7 downto 0)
		       );
	end component;

begin

XOR_MODULE0 : xor_module port map (Sel => AddSub,
                                   dataa => B,
                                   result => B_intermed);

ADDSUB0 : add_sub port map (add_sub => AddSub,
                            dataa => A,
                            datab => B_intermed,
                            result => S,
                            Cout => Cout);

end arch_addersubtractor;



-- adder/subtractor pe 8 biți – interfață/arhitectură modul secundar 1

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

```
entity add_sub is
	port(
		add_sub : in std_logic;
		dataa	: in std_logic_vector(7 downto 0);
		datab	: in std_logic_vector(7 downto 0);
		result	: out std_logic_vector(7 downto 0);
		Cout	: out std_logic
	    );

end add_sub;

architecture arch_addsub of add_sub is

signal sum : std_logic_vector(8 downto 0);

begin

sum   <= (‘0’ & dataa) + (’0’ & datab) + add_sub;
result <= sum(7 downto 0);
Cout  <= sum(8);

end arch_addsub;
```

-- modul xor pe 8 biți – interfață/arhitectură modul secundar 2

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

entity xor_module is
	port(
		sel	: in std_logic;
		dataa : in std_logic_vector(7 downto 0);
		result : out std_logic_vector(7 downto 0)
	    );

end xor_module;

architecture arch_xor_module of xor_module is

begin

result(0) <= dataa(0) xor sel;
result(1) <= dataa(1) xor sel;
result(2) <= dataa(2) xor sel;
```

```
result(3) <= dataa(3) xor sel;
result(4) <= dataa(4) xor sel;
result(5) <= dataa(5) xor sel;
result(6) <= dataa(6) xor sel;
result(7) <= dataa(7) xor sel;

end arch_xor_module;
```

Figura 12 – Organizarea unui proiect simplu, circuit aritmetic adunare/scădere pentru numere întregi.

```
-- declararea unei componente

COMPONENT numele_componentei
   PORT (
            numele_semnalului : modul_semnalului tipul_semnalului;
            numele_semnalului : modul_semnalului tipul_semnalului;
            numele_semnalului : modul_semnalului tipul_semnalului;
            ...);
END numele_componentei;

-- utilizarea unei componente

numele_instanței : numele_componentei PORT MAP (
                                numele_semnalului_componentei => numele_semnalului_modulului,
                                numele_semnalului_componentei => numele_semnalului_modulului,
                                                .
                                                .
                                                .
                               numele_semnalului_componentei => numele_semnalului_modulului
                                                );
```

Figura 13 – Declararea și utilizarea unei componente.

1.4.2 Un proiect complex

În figura 14 este evidențiată structura unui proiect complex. După cum se observă toate modulele VHDL folosite frecvent sunt organizate în pachete, apoi aceste pachete sunt organizate în librării. Fișierul sursă va conține librăriile standard, librăria ieee, librăriile utilizatorului, unde se găsesc tipurile de date, semnalele, constantele sau componentele folosite frecvent. Fișierul va conține o singură interfață și o singură arhitectură, comportamentul sistemului digital fiind descris cu ajutorul stilului structural.

O astfel de organizare permite o verificare rapidă a proiectului în cazul în care există erori în modulul principal sau alte module. Acestă organizare mai permite modificarea comportamentului modulelor utilizate frecvent fără a fi nevoie ulterior de a modifica modulul principal, dacă în prealabil s-a ținut cont de regula celor trei E.

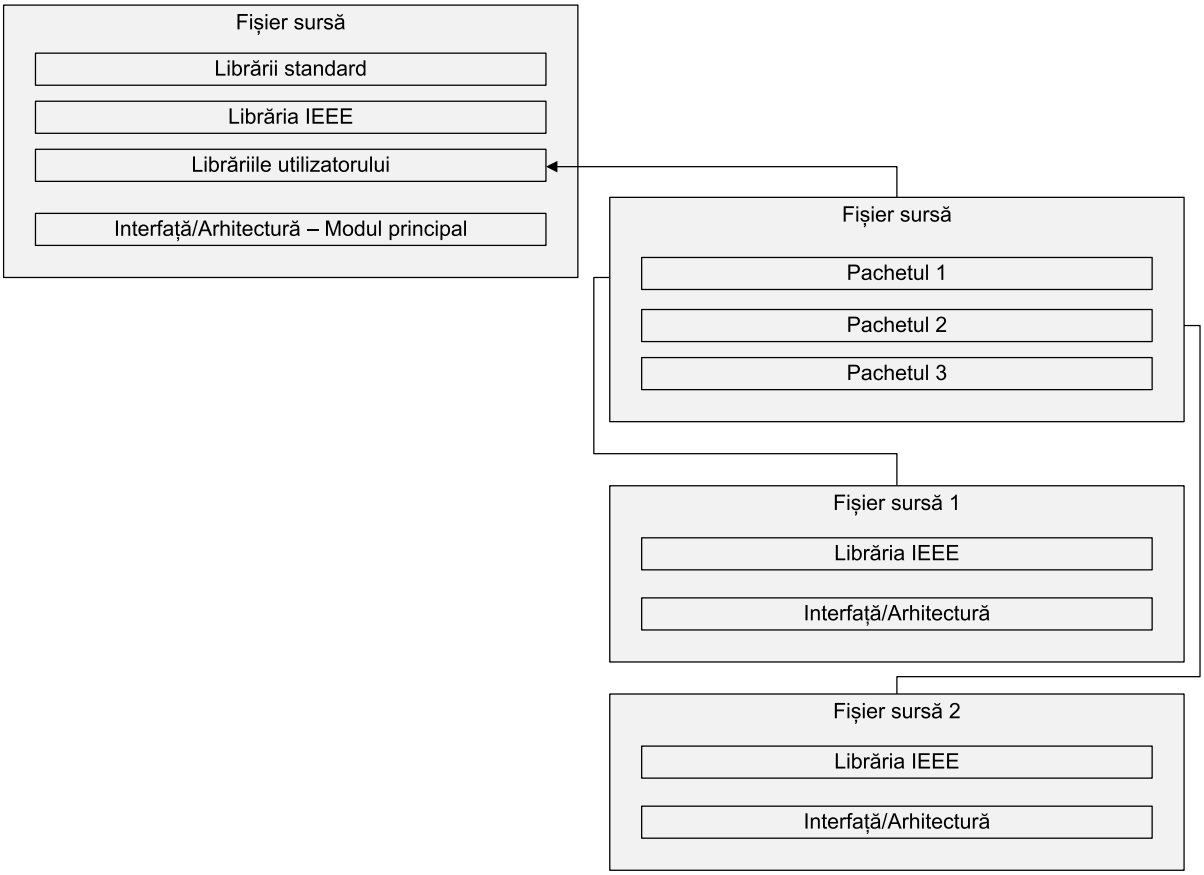


Figura 14 – Structura unui proiect complex.

Figura 15 prezintă organizarea unui proiect complex printr-un exemplu, circuit aritmetic pentru adunarea numerelor naturale.

fulladder.vhd – modul utilizat frecvent

```
library ieee;
use ieee.std_logic_1164.all;
entity fulladder is
        port(
              Cin, x, y : in std_logic;
              s, Cout   : out std_logic
             );
end fulladder;
architecture arch_fulladder of fulladder is
begin
         s      <= x xor y xor Cin;
         Cout <= (x and y) or (x and Cin) or (y and Cin);
end arch_fulladder;
```

mypack.vhd – pachet din librăria utilizatorului

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

package mypack is

    component fulladder
        port( Cin, x, y : in std_logic; s, Cout : out std_logic);
    end component;
```

```
end mypack;
```

adder_unsigned.vhd – proiect complex

```
library ieee;
use ieee.std_logic_1164.all;
library mypack;
use mypack.mypack.all;
entity adder_unsigned is
        port( Cin    : in std_logic;
              x, y  : in std_logic_vector(3 downto 0);
              Cout : out std_logic;
               s      : out std_logic_vector(3 downto 0)
              );
end adder_unsigned;

architecture arch_adder_unsigned of adder_unsigned is

signal c : std_logic_vector(1 to 3);

begin

stage0 : fulladder port map (Cin => Cin, x => x(0), y => y(0), s => s(0), Cout => c(1));
stage1 : fulladder port map (Cin => c(1), x => x(1), y => y(1), s => s(1), Cout => c(2));
stage2 : fulladder port map (Cin => c(2), x => x(2), y => y(2), s => s(2), Cout => c(3));
stage3 : fulladder port map (Cin => c(3), x => x(3), y => y(3), s => s(3), Cout => Cout);

end arch_my_pack;
```

Figura 15 – Organizarea unui proiect complex, circuit aritmetic pentru adunarea numerelor naturale.

1.5 Verificarea sistemelor digitale

Verificarea modulelor VHDL din acest îndrumar va fi făcută prin simulare funcțională și temporală cu ajutorul software-ului CAD. Tehnica de verificare folosită frecvent se numește **Bottom-Up**. Acestă tehnică impune testarea comportamentului modulelor primare, apoi testarea subsistemelor (module mai mari alcătuite din module primare), după care se va testa fiecare nivel de ierarhizare inclusiv sistemul digital principal. Figura 16 evidențiază tehnica bottom-up de verificare a comportamentului unui sistem digital.

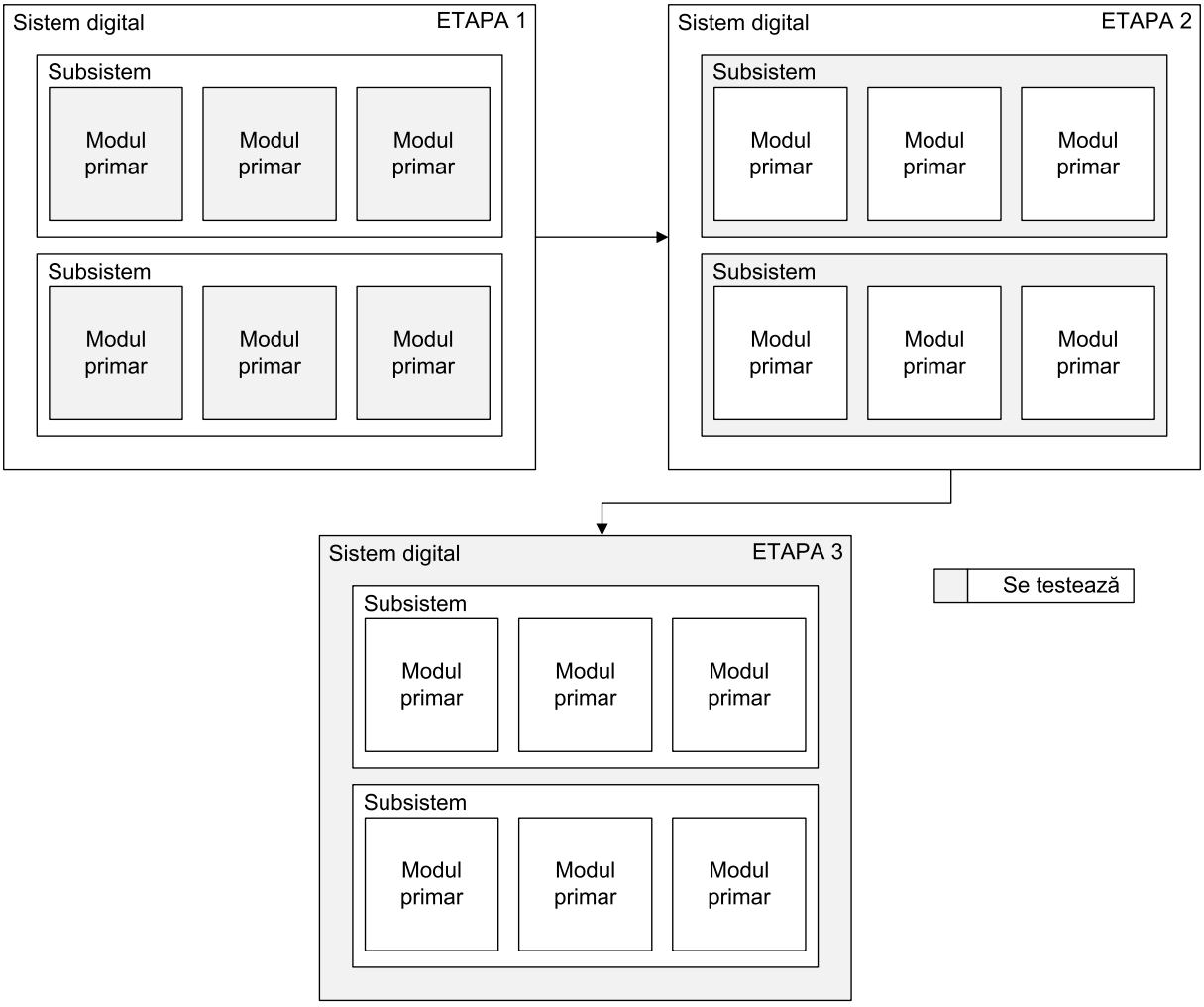


Figura 16 – Tehnica bottom-up de verificare a comportamentului unui sistem digital.

Verificarea modulelor VHDL se mai poate efectua și în hardware, folosind un **analizor digital**. Acest lucru presupune mai întâi implementarea în hardware a sistemului digital, după care folosirea analizorului digital pentru a verifica comportamentul sistemului.

Verificarea software oferă mult mai multe avantaje față de verificarea hardware. Acest lucru este evidențiat în figura 17. În practică se folosesc ambele moduri de verificare. După parcurgerea etapelor de proiectare, sistemul digital implementat în FPGA se testează hardware cu jutorul analizorului digital.

Verificare software	Verificare hardware
Viteza mică de execuție	Viteză mare de execuție
Viteză mare de corectare a erorilor	Viteză mică de corectare a erorilor
Posibilitatea testării tuturor semnalelor interne (acoperire de 90%-100%)	Imposibilitatea testării semnalelor interne

Figura 17 – Avantaje și dezavantaje pentru verificărea software și hardware.

Testarea modulelor combinaționale în software presupune încercarea tuturor combinațiilor de intrare și verificarea semnalelor de ieșire. Testatea modulelor secvențiale sincrone (mașini cu stări finite) în software presupune verificarea tuturor modurilor de tranziție. Se verifică fiecare stare și fiecare semnal de ieșire.

1.6 Tipuri de date

1.6.1 Tipurile de date std_logic și std_logic_vector

Std_logic și std_logic_vector sunt două tipuri de date ce pot fi accesate dacă în modulul VHDL se utilizează librăria ieee și pachetul std_logic_1164. În acest pachet se mai găsesc tipurile de date std_ulogic și std_ulogic_vector, însă acestea nu vor fi folosie în îndrumar.

Std_logic poate lua următoarele valori: ‘0’, ’1’, ’Z’, ’–’, ’L’, ’H’, ’U’, ’X’, ’W’. Doar primele patru valori se folosesc în procesul de sintetizare a circuitelor logice. Valoarea ’Z’ reprezintă un gol de circuit sau un fir întrerupt. – reprezintă valoarea „nu-mi pasă”. ’L’ înseamnă “slab 0”, ’H’ înseamnă “slab 1”, ’U’ înseamnă “neinițializat”, ’X’ înseamnă “necunoscut” iar ’W’ înseamnă “slab necunoscut”. Std_logic_vector reprezintă o mulțime (un array) de obiecte

std_logic. În figura 18 sunt prezentate câteva exemple de asignări folosind tipurile de date std_logic și std_logic_vector.

```
a, b, c : in std_logic; -- 3 semnale de intrare de tipul std_logic
d       : in std_logic_vector(0 to 7) – semnal de intrare pe 8 biți de tipul std_logic_vector
e, f    : out std_logic_vector(10 downto 0) – semnal de ieșire pe 11 biți de tipul std_logic_vector
```

Figura 18 – Exemple de asignări folosind tipurile de date std_logic și std_logic_vector.

Semnalele de tipul std_logic_vector pot fi utilizate în operații aritmetice, cum ar fi operațiile de adunare sau scădere, ca numere întregi sau ca numere naturale dacă se vor utiliza pachetele std_logic_signed, respectiv std_logic_unsigned, împreună cu pachetul std_logic_1164 în modulul VHDL. În acestă situație compilatorul VHDL va sintetiza un circuit care să funcționeze cu tipul de date respectiv (circuit aritmetic pentru numere întregi sau naturale).

1.6.2 Tipul de date boolean

Tipul de date boolean poate lua la un moment dat două valori TRUE (adevărat) și FALSE (fals). Valori boole-ene se obțin în urma operațiilor de comparație sau în atribuiri condiționale sau selectate, unde se folosește cuvântul cheie when.

Trebuie menționat că valoarea booleană true nu este echivalentă cu valoarea std_logic ‘1’, iar valoarea booleană fals nu este echivalentă cu valoarea std_logic ‘0’. Utilizarea acestui tip de date va fi evidențiată în capitolul următor.

1.6.3. Tipul de date integer

Datele de tipul integer pot lua valori întregi pozitive și negative, în intervalul de la -2^{31} la $2^{31} - 1$. Valorile integer se utilizează ca indici pentru bus-uri (semnale alcătuite din doi sau mai mulți biți). În figura 19 este evidențiată utilizarea tipului de date integer.

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

entity sample_circuit is
```

```
        port ( a     : in std_logic_vector(3 downto 0);
               c     : out std_logic);
end sample_circuit;

architecture arch_sample_circuit of sample_circuit is
begin
      c <= a(3) xor a(2) xor a(1) xor a(0);
end arch_sample_circuit;
```

Figura 19 – Utilizarea tipului de date integer.

Tipurile de date std_logic și std_logic_vector pentru a putea fi folosite într-un modul VHDL este necesar să fie atribuite explicit unor semnale, pe când tipurile de date boolean și integer se folosesc implicit, nefiind întotdeauna necesară atribuirea lor. În cazuri foarte rare se pot tipurile de date boolean sau integer se pot atribui explicit.

1.6.4 Tipul de date enumeration

Limbajul VHDL oferă posibilitatea utilizatorilor de a reprezenta informația într-un mod abstract. Folosind tipul de date enumaration informația se reprezintă sub formă de cuvinte. Cu acest tip de date putem defini stările unei maşini cu stări finite. Descrierea şi utilizarea acestui tip de date vor fi evidențiate în capitolul “Sintetizarea maşinilor cu stări finite”.

1.6.5 Tipul de date array

Tipul de date array descrie organizarea grupurilor de elemente de acelaşi tip. Cu ajutorul tipului de date array se pot crea alte tipuri de date având diferite dimensiuni. În figura 20 este reprezentat tipul de date **nibble**. Acest tip de date este unidimensional fiind alcătuit din patru elemente de tipul std_logic. În limbaj tehnic se spune că un nibble este jumătate de byte.

```
architecture arch_sample_circuit of sample_circuit is

type nibble is array(3 downto 0) of std_logic;
signal data_sample : nibble;


begin
```

```
            .
            .
            .
end arch_sample_circuit;
```

Figura 20 – Definirea și utilizarea tipului de date array.

În capitolul Sintetizarea memoriilor şi a controlerelor pentru memorii este evidențiată descrierea unui tip de date bidimensional cu ajutorul tipului de date array.

Capitolul 2

Sintetizarea circuitelor logice combinaționale

prima parte

Sintetizarea circuitelor logice combinaționale prima parte

Circuitele logice combinaționale sunt acele circuite pentru care ieșirile depind doar de intrările actuale. Aceste circuite nu au memorie. Porțile logice fundamentale, not, and, or, nand, nor, xor, xnor, sunt circuite logice combinaționale. În limbaj VHDL acest tip de circuit poate fi descris în stil comportamental sau în stil structural. Folosind tehnica de gestionare a complexității și regula celor trei E se pot crea circuite logice combinațioanle de diferite complexități. În acest capitol se vor prezenta circuitele logice combinațioanale fundamentale, modul în care sunt descrise în limbajul VHDL și comportamentul temporal al acestora.

2.1 Operatori binari

Limbajul VHDL permite utilizarea mai multor tipuri de operatori: **logici, relaționali, aritmetici, de deplasare, de concatenare și de atribuire**. În figura 1 sunt evidențiați operatorii binari ai limbajului VHDL. Trebuie menționat că nu toți operatorii pot fi sintetizați, cum ar fi /, sla sau sra. Operatorii VHDL sunt grupați, în figura 2, în funcție de prioritate.

Operator	Simbolul operatorului	Operația efectuată
de atribuire	<=, =>	atribuire de valori
logic	NOT, AND, OR, NAND, NOR, XOR, XNOR	not, and, or, not and, not or, xor, not xor
relațional	=, /=, >, <, >=, <=	egalitate, inegalitate, mai mare, mai mic, mai mare sau egal, mai mic sau egal
aritmetic	+, -, *, /	adunare, scădere, înmulțire, împărțire
de deplasare	sll, srl, sla, sra, rol, ror	deplasare logică la stânga, deplasare logică la dreapta, deplasare aritmetică la stânga, deplasare aritmetică la dreapta, rotire la stânga, rotire la dreapta,
de concatenare	&	concatenare

Figura 1 – Operatori binari.

Operator	Simbolul operatorului	Prioritate
diverşi	NOT	ridicată
aritmetic	*, /	
aritmetic şi de concatenare	+, -, &	
de deplasare	sll, srl, sla, sra, rol, ror	
relaţionali	=, /=, <, <=, >, >=	
logic	AND, OR, NAND, NOR, XOR, XNOR	scăzută

Figura 2 – Prioritatea operatorilor.

Operatorii cu prioritate egală sunt evaluaţi de la stânga la dreapta. Pentru a asigura o interpretare corectă a expresiilor logice descrise cu ajutorul operatorilor din figura 1 trebuie utilizate **parantezele**. În figura 3 este evidenţiată o expresie logică ce defineşte comportamentul unui circuit logic combinaţional.

```
f = (a · b) ⊕ (c+d) -- expresie logică ce descrie comportamentul unui circuit logic
-- combinaţional.

f <= (a and b) xor (c or d); -- expresie logică scrisă corect în limbaj VHDL folosind paranteze
-- ţinând cont de prioritatea egală a operatorilor.

f <= a and b xor c or d; -- expresie logică scrisă incorect, fără paranteze, în limbaj
-- VHDL. Această expresie logică are următoarea semnificaţie f = ((a · b) ⊕ c ) + d.
```

Figura 3 – Expresie logică scrisă corect.

2.1.1 Operatorul de atribuire

Operatorul de atribuire se foloseşte frecvent atunci când se doreşte atribuirea unei valori unui semnal. În figura 4 este evidenţiată utilizarea operatorului de atribuire.

```
... a     : out std_logic;
    b, c : out std_logic_vector(3 downto 0);
...
```

```
... a <= ’1’; -- această expresie se citește astfel: semnalul a ia valoarea logică 1.

   b <= ”1111”;

   c <= (3 => ’1’, OTHERS => ’0’); -- această expresie se citește astfel: bitul 3 al semnalului
-- c ia valoarea logică 1, iar biții 2, 1 și 0 (restul biților) iau valoarea logică 0. Termenul
-- OTHERS va fi explicat mai târziu în acest capitol.
```

Figura 4 – Utilizarea operatorului de atribuire.

2.1.2 Operatorul de concatenare

Acest operator conectează mai multe semnale pentru a crea un semnal mai mare.

```
... a, b : in std_logic_vector(3 downto 0);
   c, d : out std_logic_vector(7 downto 0);
…
... c <= a & b; -- această expresie se citește astfel: semnalul c este alcătuit din concatenarea
-- semnalelor a și b. După cum se observă 2 semnale pe 4 biți au fost conectate pentru a crea
-- un semnal pe 8 biți.

   d <= ”0000” & a; -- semnalul d este alcătuit din conectarea semnalului a împreună cu 4 biți
-- de valoare logică 0.
```

Figura 5 – Utilizarea operatorului de concatenare.

2.1.3 Operatori logici

Acești operatori se folosesc pentru a crea expresii logice, ce evidențiază circuite logice combinaționale. Operatorii logici se pot folosi pentru semnalele pe un singur bit sau mai mulți biți. Utilizarea lor este evidențiată în figura 6.

```
... a, b, c : in std_logic;
   d, e, f : in std_logic_vector(3 downto 0);
```

```
   g, h : out std_logic;
   i, j  : out std_logic_vector(7 downto 0);
…
... g <= not a; -- semnalul g este complementul semnalului a. Această expresie va fi
-- sintetizată ca un invertor pe 1 bit.

    h <= b and c; -- semnalul h este b și c. Această expresie va fi sintetizată ca o poartă logică
-- AND pe 1 bit.

    i <= not d; -- semnalul i este complementul semnalului d. Această expresie va fi
-- sintetizată ca un grup de 4 invertor pe 1 bit sau un invertor pe 4 biți.

    j <= e or f; -- semnalul j este e ori f. Această expresie va fi sintetizată ca un grup de 4
-- porți logice OR pe 1 bit sau o poartă logică OR pe 4 biți.
```

Figura 6 – Utilizarea operatorilor logici.

2.1.4 Operatori relaționali

Operatorii relaționali sunt utilizați pentru a compara expresii. Acest tip de operatori se folosesc în **atribuiri condiționale**, ce vor fi evidențiate mai târziu în acest capitol. Expresiile ce vor fi comparate trebuie să fie de același tip. Rezultatul unei comparații oarecare va fi **adevărat** (TRUE) sau **fals** (FALSE), adică rezultatul va fi de tip boolean. Operatorii relaționali sunt evidențiați în figura 7.

```
... a, b, : out std_logic_vector(3 downto 0); -- avem doua semnale de intrare pe 4 biți.

... a <= ”1111”;
   b <= ”0001”;
... a  /=  b – rezultatul comparației este adevărat, TRUE.
   a <= b – rezultatul comparației este fals, FALSE. Operatorul relațional se scrie la fel ca
-- operatorul de atribuire. „<=” în contexte diferite este interpretat difert. În acest caz este
-- vorba de operație de comparație și nu de o atribuire, deci expresia se citește: valoarea
-- semnalului a este mai mică sau egală cu valoarea semnalului b?.
```

a = ”1101” -- rezultatul comparației este **fals**, **FALSE**.

b >= ”0010” -- rezultatul comparației este **adevărat**, **TRUE**.

b > a -- rezultatul comparației este **fals**, **FALSE**.

a < b -- rezultatul comparației este **fals**, **FALSE**.

Figură 7 – Utilizarea operatorilor relaționali.

2.1.5 Operatori aritmetici

Operatorii aritmetici se folosesc în operații aritmetice obișnuite: adunare, scădere, înmulțire sau împărțire. În figura 8 sunt evidențiați operatorii aritmetici. Software-urilor CAD vor sintetiza corect operațiile de adunare, scădere sau înmulțire. De obicei operația de împărțire nu este suportată de software-urile CAD.

```
... a, b      : in std_logic_vector(3 downto 0);
    c, d, e   : out std_logic_vector(3 downto 0);
    f         : out std_logic_vector(7 downto 0);
...
... c <= a + b; -- rezultatul adunării semnalelor a și b este atribuit semnalului c.

    d <= a - b; -- rezultatul scăderii semnalului b din semnalul a este atribuit semnalului d.

    e <= a / b; -- rezultatul împărțierii semnalului a la semnalul b este atribuit semnalului e.

    f <= a * b; -- rezultatul înmulțirii semnalelor a și b este atribuit semnalului c. Rezultatul
-- înmulțirii a două semnale pe 4 biți este un semnal pe maxim 8 biți.
```

Figură 8 – Utilizarea operatorilor aritmetici.

2.1.6 Operatori de deplasare

Cu ajutorul operatorilor de deplasare biții unui semnal se pot deplasa la stânga, la dreapta sau se pot roti. În cazul în care deplasarea este logică, locul rămas vacant datorită deplasării biților va fi ocupat cu biți de valoare logică 0. Deplasarea aritmetică la stânga (sla) produce același rezultat ca deplasarea logică la stânga (sll). Deplasarea aritmetică la dreapta va permite ocuparea locurilor rămase libere cu biți având valoarea logică a primului bit al semnalului care se dorește a fi deplasat. Acest operator nu este suportat de standardele ieee pentru limbajul VHDL. Figura 9 evidențiază operatorii de deplasare.

```
... a     : buffer std_logic_vector(3 downto 0);
   b, c : out std_logic_vector(3 downto 0);
...
   a  <= ”1011”; semnalul a ia valoarea logică ”1011”.
   b <= conv_std_logic_vector(shl(conv_unsigned(conv_integer(a), 8), 2), 8); -- semnalul a
-- va fi deplasat logic 2 poziții la stânga, iar semnalul b va fi egal cu ”0010”.
-- shl este echivalent cu sll iar shr este echivalent cu srl.

   c <= a conv_std_logic_vector(rol(conv_unsigned(conv_integer(a), 8), 3), 8); -- semnalul a
-- va fi rotit 3 poziții, iar semnalul d va fi egal cu ”1101”.
-- conv_std_logic_vector, conv_unsigned, conv_interger sunt funcții de conversie.
```

Figura 9 – Utilizarea operatorilor de deplasare.

2.2 Declarații paralele

O declarație paralelă se utilizează atunci când se dorește atribuirea unei valori unui anumit semnal. Într-un modul VHDL declarațiile paralele se vor descrie în interiorul corpului unei arhitecturi, aceast loc poartă numele de **zonă simultană**. În figura 10 este evidențiată acestă zonă. Toate declarațiile din zona sunt executate în paralel (simultan) având aceeași prioritate, deci nu există o ordine de scriere a declarațiilor. Acest lucru este evidențiat în figura 11. În limbajul VHDL există patru tipuri de declarații paralele: **atribuire simplă**, **atribuire condițională**, **atribuire selectată** și **declarația generate** (descrisă în capitolul Module parametrizabile).

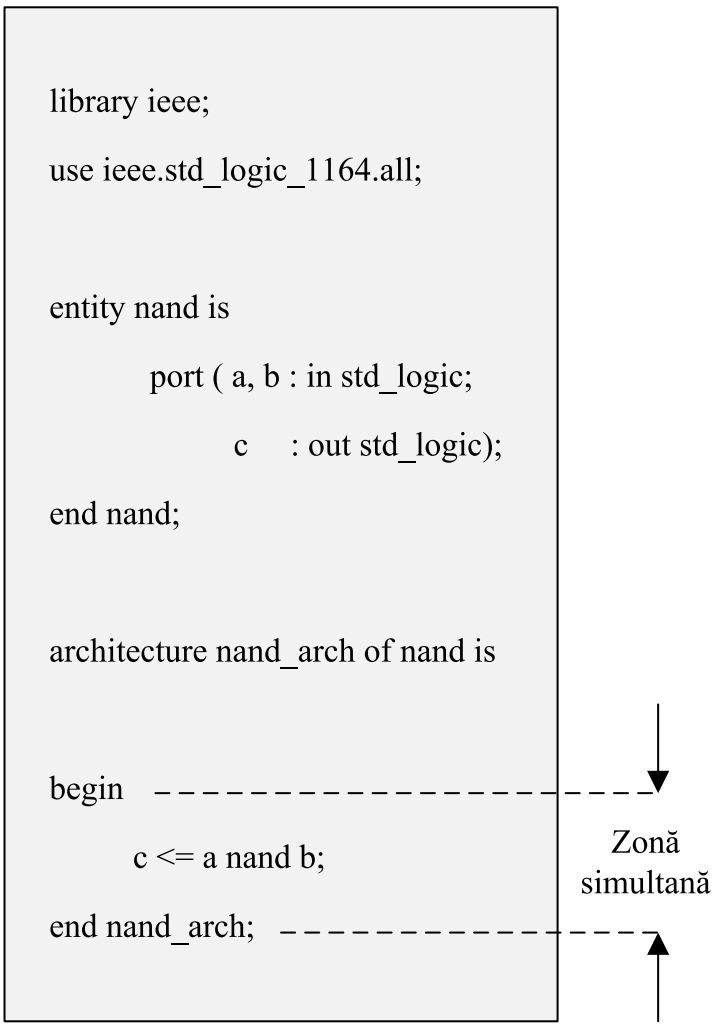


Figura 10 – Zona simultană

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

entity circ1 is
        port ( a, b : in std_logic;
               c, d : out std_logic);
end circ1;

architecture circ1_arch of circ1 is
begin
       c <= a nand b;
       d <= a nor b;

end circ_arch;
```

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

entity circ2 is
        port ( a, b : in std_logic;
               c, d : out std_logic);
end circ2;

architecture circ2_arch of circ2 is
begin
       d <= a nor b;
       c <= a nand b;

end circ2_arch;
```

Figura 11 – Declarațiile paralele au prioritate egală, rezultând în sintetizarea aceluiași circuit.

2.2.1 Atribuire simplă

Atribuirea simplă poate fi descrisă de operatorul de atribuire evidențiat în prima parte a acestui capitol.

Dacă se dorește ca toți biții unui semnal să fie egali cu valoarea logică 0 sau 1, putem proceda ca în figura 12. În acest caz semnalele sunt alcătuite din patru biți, atribuirea fiind ușor de realizat. În cazul în care într-un modul VHDL există semnale pe mai mult de patru biți și se dorește efectuarea aceluiași tip de atribuire, se poate folosi cuvântul cheie **OTHERS**. Utilizarea acestui cuvânt este evidențiată în figura 13.

```
... a, b : out std_logic_vector(3 downto 0);
…
… a <= ”0000”;
   b <= ”1111”;
…
…
```

Figura 12 – Atribuiri simple.

```
... a, b, c : out std_logic_vector(10 downto 0);
…
… a  <= (OTHERS => ’0’);
   b <= (OTHERS => ’1’); -- b <= (OTHERS => ’Valoare’), fiecare bit al semnalului b ia
-- valoarea Valoare.
   c <=  (OTHERS => ’Z’);
…
…
```

Figura 13 – Utilizarea cuvântului cheie OTHERS.

Atribuirea de valori logice folosind cuvântul cheie OTHERS poate fi efectuată pentru orice număr de biți dintr-un semnal.

2.2.2 Atribuire condiţională

Atribuirea condiţională poate fi utilizată când se doreşte selectarea valorii unui semnal din mai multe valori alternative pe baza unor condiţii de selecţie. Condiţiile logice trebuie să se excludă reciproc. Condiţia logică poate fi: o singură valoare, câteva valori separate prin simbolul | (or = sau) sau o mulţime de valori (discrete). Acestea vor putea fi evidenţiate cu ajutorul operatorilor relaţionali. Figura 14 evidenţiază forma generală a atribuirii condiţionale.

```
denumire_semnal <= semnal(expresie)_logic(ă) WHEN condiţie_logică ELSE
                   semnal(expresie)_logic(ă) WHEN condiţie_logică ELSE
                   .
                   .
                   .
                   semnal(expresie)_logic(ă) (WHEN OTHERS);
```

Figura 14 – Atribuire condiţională.

Atribuirea condiţională impune întotdeauna o anumită prioritate asupra condiţiilor logice. Într-o atribuire condiţională prima condiţie logică are prioritatea cea mai ridicată, iar ultima condiţie logică are prioritatea cea mai scăzută. În figura 15 este evidenţiat un modul VHDL unde s-a utilizat o atribuire condiţională.

```
library ieee; use ieee.std_logic_1164.all;
entity nand is
        port ( a, b, c, d : in std_logic; e : in std_logic_vector(2 downto 0);
               f          : out std_logic);
end nand;
architecture nand_arch of nand is
begin
       f <= a when  e(2) = '1' else
            b when e(1) = '1' else
            c when e(0) = '1' else
            d when OTHERS;
end nand_arch;
```

Figură 15 – Exemplu atribuire condiţională

Pentru ultima condiție s-a folosit cuvântul cheie „OTHERS” pentru a asigura includerea tuturor condițiilor (în limbajul VHDL semnalele pot avea 9 valori). Atunci când se dorește descrierea comportamentului unui circuit logic combinațional printr-o atribuire condițională este necesară evidențierea tuturor condițiilor logice.

2.2.3 Atribuire selectată

Atribuirea selectată poate fi utilizată atunci când se dorește selectarea valorii unui semnal din mai multe valori alternative pe baza unor criterii de selecție. Valorile constante trebuie să se excludă reciproc. Valoarea constantă poate fi: o singură valoare, câteva valori separate prin simbolul | (or = sau) sau o mulțime de valori (discrete). Figura 16 evidențiază modul general în care se poate crea o atribuire selectată. Diferența dintre o atribuire selectată și una condițională este aceea că atribuirea selectată nu impune o prioritate asupra condițiilor logice. În figura 17 este evidențiat un modul VHDL unde este utilizată o atribuire selectată.

```
WITH semnal(expresie)_logic(ă) SELECT
        denumire_semnal <= semnal(expresie)_logic(ă) WHEN valoare_constantă,
                                semnal(expresie)_logic(ă) WHEN valoare_constantă,
                                                .
                                                .
                                                .
                                semnal(expresie)_logic(ă) WHEN OTHERS;
```

Figura 16 – Atribuire selectată.

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

entity nand is
        port ( a, b, c, d : in std_logic; e : in std_logic_vector(2 downto 0);
                f          : out std_logic);
end nand;

architecture nand_arch of nand is
begin
      with e select
           f <= a when ”00”,
                b when ”01”,
```

```
                c when ”10”,
                d when OTHERS;
end nand_arch;
```

Figura 17 – Exemple atribuire selectată.

2.3 Semnale interne

În limbajul VHDL există 3 tipuri de semnale: de intrare, de ieșire și interne. Semnalele de intrare și ieșire sunt echivalente cu intrările și ieșirile unui circuit logic real. Aceste tipuri de semnale sunt declarate în interiorul unei entități.

Semnalele interne reprezintă echivalentul firelor de legătură dintre două sau mai multe circuite logice reale. Într-un modul VHDL semnalele interne se folosesc pentru a conecta porți logice, componente, procese (descrierea și utilizarea unui proces va fi evidențiată în capitolul următor). Acestea se mai pot folosi în interiorul unui proces.

Semnalele interne pot fi globale dacă sunt declarate într-un pachet sau locale dacă sunt declarate într-o arhitectură. În Anexa 1 este evidențiat modul de declarare al unui semnal intern global. Semnalele interne globale sunt vizibile în oricare modul VHDL într-un proiect. Semnalele interne locale sunt vizibile doar în arhitectura modulului VHDL în care au fost declarate. Figura 18 evidențiază locația și modul general de declarare al unui semnal intern local. În figura 19 este prezentat un modul VHDL unde se utilizează un semnal intern local pentru a face legătura între două porți logice.

```
...
architecture circ_arch of circ is
SIGNAL denumirea_semnalului : tipul_semnalului;
begin
...
end circ_arch;
```

Figură 18 – Locația și modul general de declarare al unui semnal intern local.

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

entity sum_of_products is
        port ( a, b, c, d : in std_logic;
               f          : out std_logic);
end sum_of_products;

architecture sum_of_products_arch of sum_of_products is

signal intern_a, intern_b : std_logic;

begin

intern_a <= a and b;
intern_b <= c and d;
f        <= intern_a or intern_b;

end sum_of_products_arch;
```

Figura 19 – Utilizarea semnalelor interne locale.

2.4 Numere binare

În limbajul VHDL numerele sunt șiruri de literali care se folosesc pentru a reprezenta numere în baza 2 (numere binare), baza 8 (numere octale) sau baza 16 (numere hexazecimale). Numerele pot fi scrise cu underscore-uri (linii de subliniere) pentru a fi lizibile.

În VHDL numerele având tipul std_logic sunt numere binare pe un singur bit, scrise cu apostrofi. Descrierea acestora este evidențiată în figura 20.

```
... a, b : out std_logic;
...
...

a <= '1';
b <= '0';
...
...
```

Figura 20 – Reprezentarea numerelor binare pe un singur bit.

În limbajul VHDL numerele având tipul std_logic_vector pot fi numere binare pe mai mulți biți, numere octale sau numere hexazecimale, scrise cu ghilimele. Atunci când sunt descrise numerele binare pe mai mulți biți trebuie precedate de litera B, numerele octale trebuie precedate de litera O, iar numerele hexadecimale trebuie precedate de litera X. Descrierea acestora este evidențiată în figura 21.

```
... a : out std_logic_vector(7 downto 0);
    b : out std_logic_vector(2 downto 0);
    c : out std_logic_vector(17 downto 0);
    d : out std_logic_vector(5 downto 0);
    e : out std_logic_vector(3 downto 0);
    f : out std_logic_vector(7 downto 0);
    g : out std_logic_vector(31 downto 0) );
...

a <= B”0110_1001”;
b <= O”7”;
c <= O”373_103”;
d <= O”23”;
e <= X”F”;
f <= X”2F”;
g <= X”FADE_FALL”;
...
```

Figura 21 – Reprezentarea numerelor binare pe mai mulți biți, numerelor octale și numerelor hexazecimale.

Trebuie reamintit că în cadrul arhitecturilor pe 32 de biți un byte este alcătuit din 8 biți, un nibble este alcătuit din 4 biți, un cuvânt este alcătuit din 32 de biți, iar jumătate de cuvânt este alcătuit din 16 biți. În arhitecturile pe 64 de biți un cuvânt este alcătuit din 64 de biți, jumătate de cuvânt este alcătuit din 32 de biți, numărul de biți pentru un byte sau un nibble rămânând același.

2.5 Circuite logice combinaționale – module comportamentale

În acest subcapitol este prezentat modul de sintetizare al circuitelor logice combinaționale reprezentate prin module comportamentale. Astfel de module descriu relația intrare-ieșire pentru un anumit circuit. Relația intrare-ieșire poate fi descrisă cu ajutorul unui tabel de adevăr sau o funcție logică. Acest subcapitol prezintă circuitele logice combinaționale fundamentale: porți logice fundamentale, multiplexoare, demultiplexoare, codificatoare, decodificatoare și convertoare.

2.5.1 Porți logice fundamentale

Circuitele logice combinaționale pot fi alcătuite din porți logice fundamentale. Aceste porți sunt: poarta NOT, poarta AND, poarta OR, poarta NAND, poarta NOR, poarta XOR și poarta XNOR. Aceste porți pot fi sintetizate cu ajutorul limbajului VHDL. Modulul VHDL din figura 22 evidențiază cele șapte porți logice fundamentale. În figura 23 sunt evidențiate cele șapte porți fundamentale sintetizate. În figura 24 este evidențiat modulul VHDL unde sunt instanțiate cele șapte porți logice fundamentale, având intrări pe 4 biți. În figura 25 sunt evidențiate porțile logice sintetizate. Se observă câte 4 porți de 1 bit pentru fiecare tip de poartă.

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

entity gates is
        port ( a, b : in std_logic; c, d, e, f, g, h, i : out std_logic);
end gates;

architecture gates_arch of gates is
begin
   c <= not a; d <= a and b; e <= a or b;
   f <= a nand b; g <= a nor b; h <= a xor b; i <= a xnor b;
end gates_arch;
```

Figura 22 – Cele șapte porți logice fundamentale.

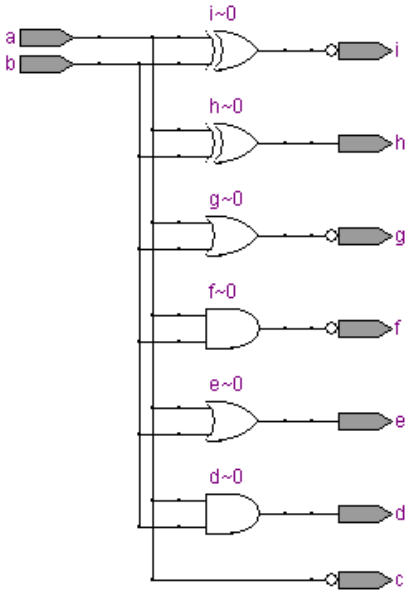
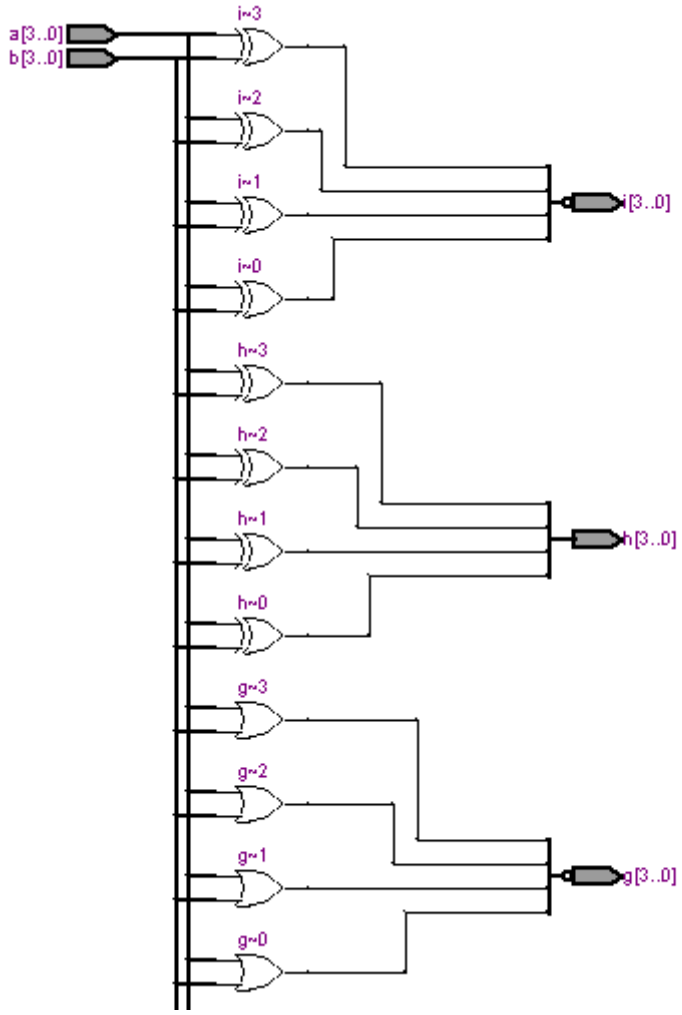


Figura 23 – Cele șapte porți logice fundamentale sintetizate.

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

entity gates_4bits is
        port ( a, b                 : in std_logic_vector(3 downto 0);
               c, d, e, f, g, h, i : out std_logic_vector(3 downto 0) );
end gates_4bits;

architecture gates_4bits_arch of gates_4bits is

begin

c <= not a ; d <= a and b; e <= a or b; f <= a nand b; g <= a nor b; h <= a xor b; i <= a xnor b;

end gates_4bits_arch;
```

Figura 24 – Cele șapte porți logice fundamentale cu intrări pe 4 biți.



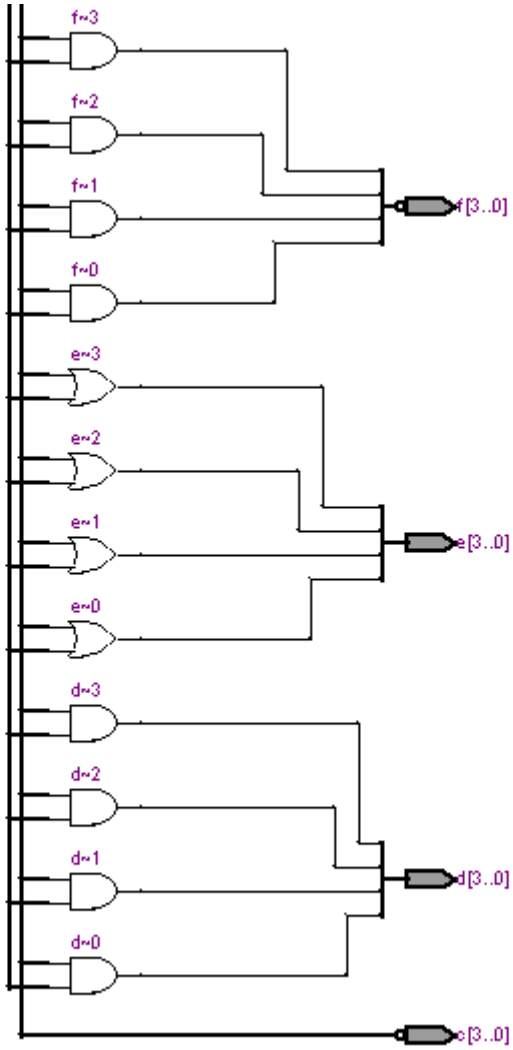


Figura 25 – Cele șapte porți fundamentale cu intrări pe 4 biți.

Porțile logice AND, OR, NAND, NOR, XOR, XNOR pot acționa asupra semnalelor pe mai mulți biți ca operatori de reducere. În figura 26 modulul VHDL evidențiază operatorul de reducere XOR. În figura 27 este reprezentat circuitul sintetizat.

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

entity xor_reduction_op is
        port ( a : in std_logic_vector(3 downto 0); b : out std_logic ); end xor_reduction_op;

architecture xor_reduction_op_arch of xor_reduction_op is
begin
b <= a(3) xor a(2) xor a(1) xor a(0); end xor_reduction_op_arch;
```

Figura 26 – Operatorul de reducere XOR.

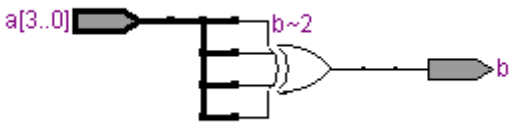


Figura 27 – Circuitul sintetizat pentru operatorul de reducere XOR.

2.5.2 Multiplexoare

Multiplexorul este un circuit logic combinațional ce deține două sau mai multe intrări de date, o intrare de selecție și o ieșire de date. În figura 28 este evidențiat comportamentul unui multiplexor 2 la 1 pe 4 biți. Intrarea de selecție sel este pe un bit. Atunci când sel = ’1’ ieșirea c este egală cu intrarea a, iar când sel = ’0’ ieșirea c este egală cu intrarea b. Circuitul sintetizat este reprezentat în figura 29. În limbaj VHDL comportamentul unui multiplexor se poate descrie cu ajutorul atribuirilor condiționale sau selectate.

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

entity mux21 is
        port ( a, b : in std_logic_vector(3 downto 0);
               sel  : in std_logic;
                 c    : out std_logic_vector(3 downto 0) );
end mux21;

architecture mux21_arch of mux21 is
begin

c <= a when sel = ’1’ else b; -- atribuire condițională

end mux21_arch;
```

Figura 28 – Multiplexor 2 la 1 pe 4 biți.

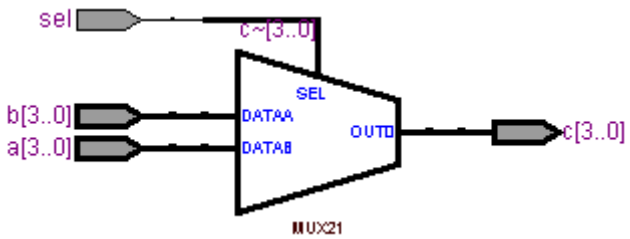


Figura 29 – Sintetizarea multiplexorului 2 la 1 pe 4 biți din figura 28.

În figura 30 este evidențiat un multiplexor 4 la 1 pe 4 biți descris printr-o atribuire selectată. Pentru a selecta 4 intrări avem nevoie de o intrare de selecție pe $\log_2 4$ (2) biți. În figura 31 este evidențiat simbolul multiplexorului 4 la 1 pe 4 biți.

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

entity mux41 is
        port ( a, b, c, d : in std_logic_vector(3 downto 0);
               sel        : in std_logic_vector(1 downto 0);
                 e          : out std_logic_vector(3 downto 0) );
end mux41;

architecture mux41_arch of mux41 is
begin

with sel select
      e <= a when ”00”,
            b when ”01”,
            c when ”10”,
            d when OTHERS;
end mux41_arch;
```

Figura 30 – Modul VDHL pentru un multiplexor 4 la 1 pe 4 biți.

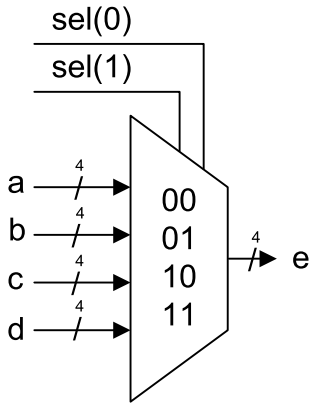


Figura 31 – Simbolul unui multiplexor 4 la 1 pe 4 biți.

2.5.3 Demultiplexoare

Demultiplexorul este un circuit logic combinațional ce deține o intrare de date, o intrare de selecție și mai multe ieșiri. Intrarea de selecție va face conexiunea între intrarea de date și ieșirea selectată. În figura 32 este evidențiată instanțierea unui demultiplexor 1 la 4 pe 4 biți. Simbolul acestui demultiplexor este reprezentat în figura 33.

```
library ieee; use ieee.std_logic_1164.all;
entity demux14 is
        port ( a          : in std_logic_vector(3 downto 0);
               sel        : in std_logic_vector(1 downto 0);
                b, c, d, e : out std_logic_vector(3 downto 0) );
end demux14;
architecture demux14_arch of demux14 is
begin
        with sel select
                b <= a          when ”00”,
                     ”0000” when OTHERS;
        with sel select
                c <= a          when ”01”,
                     ”0000” when OTHERS;
        with sel select
                d <= a          when ”10”,
                     ”0000” when OTHERS;
        with sel select
                e <= a          when ”11”,
                     ”0000” when OTHERS;
end demux14_arch;
```

Figura 32 – Modul VHDL pentru un demultiplexor 1 la 4 pe 4 biți.

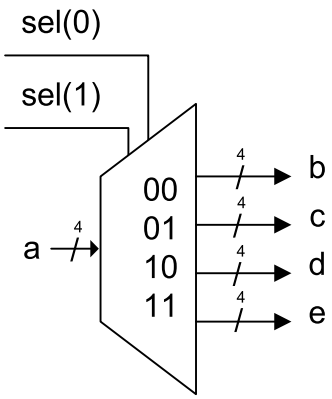


Figura 33 – Simbolul unui demultiplexor 1 la 4 pe 4 biți.

2.5.4 Codificatoare

Codificatorul este un circuit logic combinațional ce deține o intrate pe 2^n biți și o ieșire pe n biți. După cum se observă acest tip de circuit transformă o intrare pe un număr mare de biți și oferă la ieșire un număr mai mic de biți. Cele mai importante codificatoare sunt: codificatoarele binare și codificatoarele cu prioritate.

2.5.4.1 Codificator binar

În figura 34 este evidențiat tabelul de adevăr pentru un codificator binar pe 4 biți. Se observă că pe intrarea de date numai un bit este activ (1 logic) la un moment dat, iar ieșirea evidențiază indexul bit-ului activ. În figura 35 este evidențiat modulul VHDL ce descrie comportamentul codificatorului binar pe 4 la 2. În figura 36 este reprezentat circuitul sintetizat.

$a(3)$	$a(2)$	$a(1)$	$a(0)$	$b(1)$	$b(0)$
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Figura 34 – Tabel de adevăr pentru un codificator binar pe 4 biți.

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

entity bin_cod is
        port ( a  : in std_logic_vector(3 downto 0);
                b : out std_logic_vector(1 downto 0) );
end bin_cod;

architecture bin_cod_arch of bin_cod is
begin
with a select
      b <= ”00” when ”0001”,
           ”01” when ”0010”,
           ”10” when ”0100”,
           ”11” when ”1000”,
           ”--”  when OTHERS;
end bin_cod_arch;
```

Figura 35 – Modul VHDL ce descrie comportamentul codificatorului binar pe 4 biți.

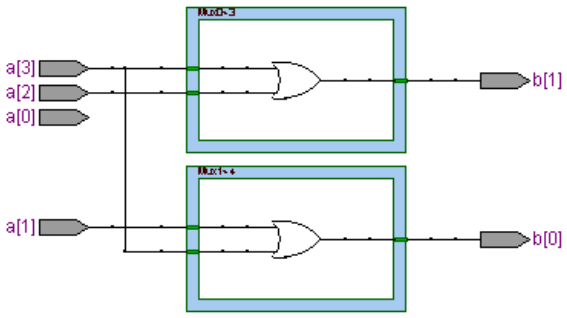


Figura 36 – Circuit sintetizat folosind modulul VHDL din figura 35.

2.5.4.2 Codificator cu prioritate

Un astfel de codificator impune o anumită prioritate asupra biților intrării de date. Ieșirea la un moment dat indică indexul bitului cu cea mai mare prioritate, ignorând biții cu index inferior. În figura 37 este reprezentat tabelul de adevăr pentru un codificator cu prioritate 4 la 2. În figura 38 este prezentat modulul VHDL unde este evidențiat comportamentul acestui codificator. Figura 39 prezintă circuitul sintetizat.

$a(3)$	$a(2)$	$a(1)$	$a(0)$	$b(1)$	$b(0)$
0	0	0	1	0	0
0	0	1	-	0	1
0	1	-	-	1	0
1	-	-	-	1	1

Figura 37 – Tabel de adevăr pentru un codificator cu prioritate 4 la 2.

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

entity pri_cod is

        port ( a  : in std_logic_vector(3 downto 0);
               b : out std_logic_vector(1 downto 0) );

end pri_cod;

architecture pri_cod_arch of pri_cod is

begin

      b <= ”11” when a(3) = ’1’ else
           ”10” when a(2) = ’1’ else
```

```
            ”01” when a(1) = ’1’ else
            ”00” when a(0) = ’1’ else
            ”--”;

end pri_cod_arch;
```

Figura 38 – Modul VHDL ce evidențiază comportamentul unui codificator cu prioritate 4 la 2.

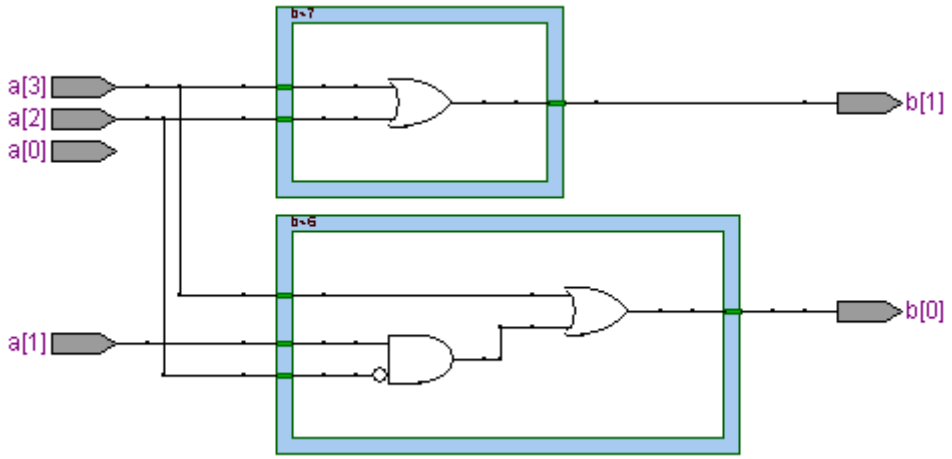


Figura 39 – Circuit sintetizat cu ajutorul modulului VHDL din figura 38.

2.5.5 Decodificatoare

Decodificatorul este un circuit logic combinațional ce implementează funcția logică inversă a codificatorului, adică decodifică semnale codificate. Acest tip de circuit are o intrare pe n biți și o ieșire pe 2^n biți. După cum se observă un decodificator va transforma o intrare pe câțiva biți într-o ieșire pe mai mulți biți. Decodificatorul va interpreta datele de la intrare și va oferi la ieșire semnale pentru care numai un bit va fi activ. Indexul bitului activ va fi egal cu valoarea semnalului de intrare. Acest lucru este evidențiat în tabelul de adevăr din figura 40. Modulul VHDL din figura 41 descrie comportamentul unui decodificator 2 la 4. Circuitul sintetizat este prezentat în figura 42.

$a(3)$	$a(2)$	$b(3)$	$b(2)$	$b(1)$	$b(0)$
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Figura 40 – Tabelul de adevăr al unui decodificator 2 la 4.

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

entity decod is

        port ( a  : in std_logic_vector(1 downto 0);
               b : out std_logic_vector(3 downto 0) );

end decod;

architecture decod_arch of decod is
begin

with a select
      b <= ”0001” when ”00”,
           ”0010” when ”01”,
           ”0100” when ”10”,
           ”1000” when ”11”,
           ”----”   when OTHERS;
end decod_arch;
```

Figura 41 – Modul VHDL ce descrie comportamenul unui decodificator 2 la 4.

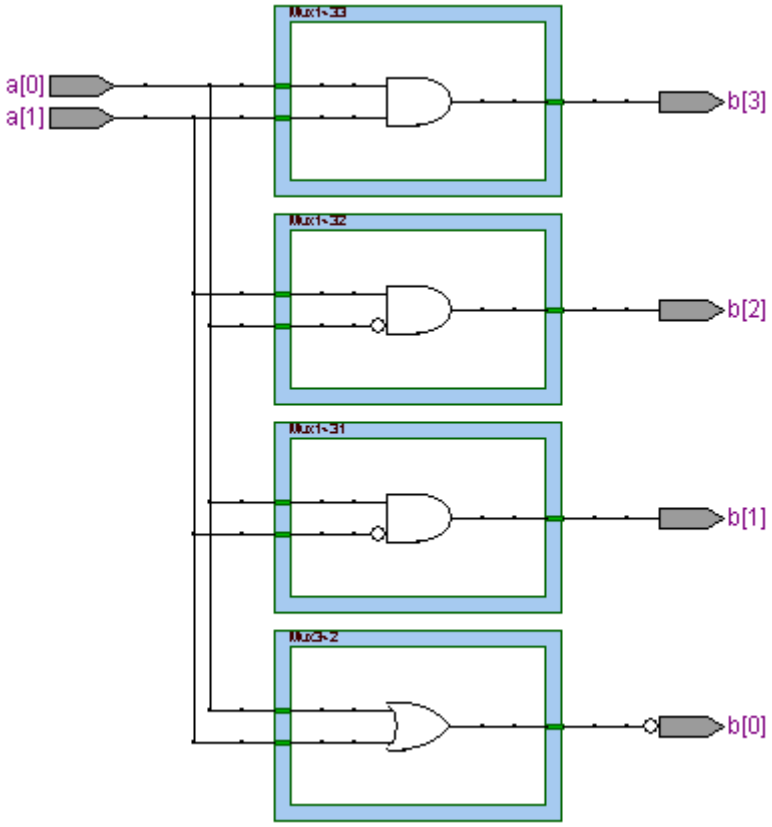


Figura 42 – Circuit sintetizat cu ajutoul modului VHDL din figura 41.

2.5.5.1 Decodificator cu prioritate

Un astfel de decodificator impune o anumită prioritate asupra biților intrării de date. Pe ieșirea de date numai un bit este activ la un moment dat. Bitul activ indică indexul bitului cu cea mai mare prioritate al intrării de date, ignorând biții cu index inferior. Acest lucru este evidențiat în tabelul de adevăr din figura 43. În acestă figură este evidențiată funcția logică a unui decodificator cu prioritate 4 la 4. Modulul VHDL ce descrie comportamentul decodificatorului este evidențiat în figura 44. Circuitul sintetizat este reprezentat în figura 45.

$a(3)$	$a(2)$	$a(1)$	$a(0)$	$b(3)$	$b(2)$	$b(1)$	$b(0)$
0	0	0	1	0	0	0	1
0	0	1	-	0	0	1	0
0	1	-	-	0	1	0	0
1	-	-	-	1	0	0	0

Figura 43 – Tabelul de adevăr pentru un decodificator cu prioritate 4 la 4.

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

entity pri_decod is

        port ( a  : in std_logic_vector(3 downto 0);
               b : out std_logic_vector(3 downto 0) );

end pri_decod;

architecture pri_decod_arch of pri_decod is

begin

      b <= ”1000” when a(3) = ’1’ else
           ”0100” when a(2) = ’1’ else
           ”0010” when a(1) = ’1’ else
           ”0001” when a(0) = ’1’ else
           ”----”;

end pri_decod_arch;
```

Figura 44 – Modul VHDL ce descrie comportamentul unui decodificator cu prioritate 4 la 4.

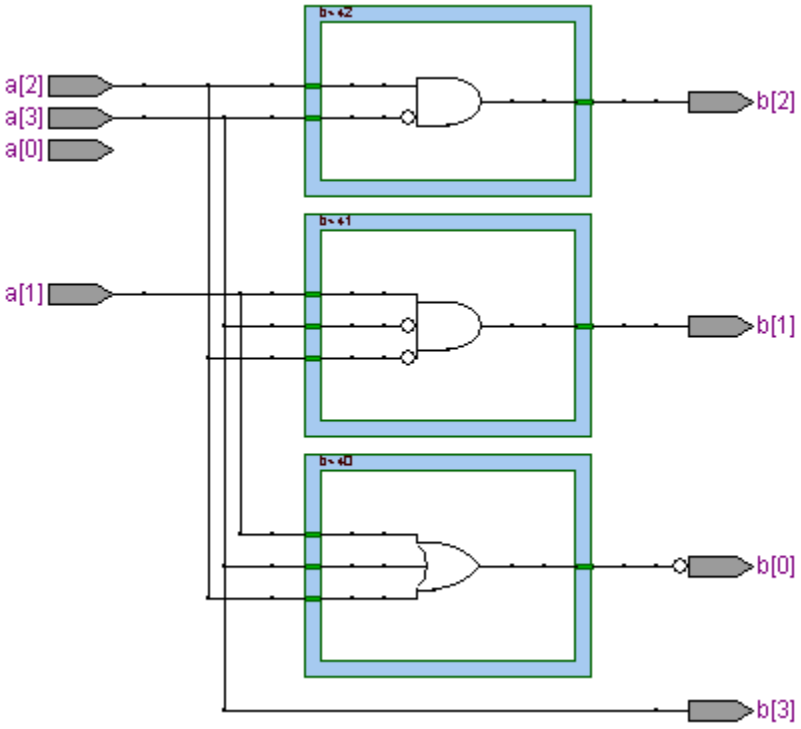


Figura 45 – Circuit sintetizat cu ajutorul modulului VHDL din figura 44.

2.5.6 Convertoare

Un convertor este un circuit logic combinațional ce transformă o codificare binară în altă codificare binară. Cel mai uzitat convertor transformă codul binar zecimal (BCD – binary coded decimal) într-un cod pentru un afișor cu șapte segmente. Tabelul de adevăr pentru convertorul prezentat mai sus este evidențiat în figura 46. În figura 47 este reprezentat un afișor cu șapte segmente. Cele șapte segmente sunt reprezentate fizic cu șapte LED-uri. Atunci când un segment ia valoarea logică 1, LED-ul corespunzător se va aprinde. Modulul VHDL pentru convertorul BCD - afișor cu șapte segmente este evidențiat în figura 47. Circuitul sintetizat este reprezentat în figura 48.

$s(3)$	$s(2)$	$s(1)$	$s(0)$	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1

0	1	1	0	0	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	0	1	0	1	1
1	0	1	0	-	-	-	-	-	-	-
1	0	1	1	-	-	-	-	-	-	-
1	1	0	0	-	-	-	-	-	-	-
1	1	0	1	-	-	-	-	-	-	-
1	1	1	0	-	-	-	-	-	-	-
1	1	1	1	-	-	-	-	-	-	-

Figura 46 – Tabel de adevăr pentru un convertor BCD – afișor cu șapte segmente.

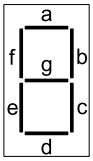


Figura 47 – Afișor cu șapte segmente.

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

entity BCD7seg is
        port ( s  : in std_logic_vector(3 downto 0); data_out : out std_logic_vector(0 to 6) );
end BCD7seg;

architecture BCD7seg_arch of BCD7seg is
begin
with s select
      data_out <= ”1111110” when ”0000”, -- data_out = abcdefg
                  ”0110000” when ”0001”,
                  ”1101101” when ”0010”,
                  ”1111001” when ”0011”,
                  ”0110011” when ”0100”,
                  ”1011011” when ”0101”,
                  ”0011111” when ”0110”,
                  ”1110000” when ”0111”,
                  ”1111111” when ”1000”,
                  ”1110011” when ”1001”,
                  ”-------”   when OTHERS;
end BCD7seg_arch;
```

Figura 48 – Modul VHDL ce descrie comportamentul convertor BCD – afișor cu șapte segmente.

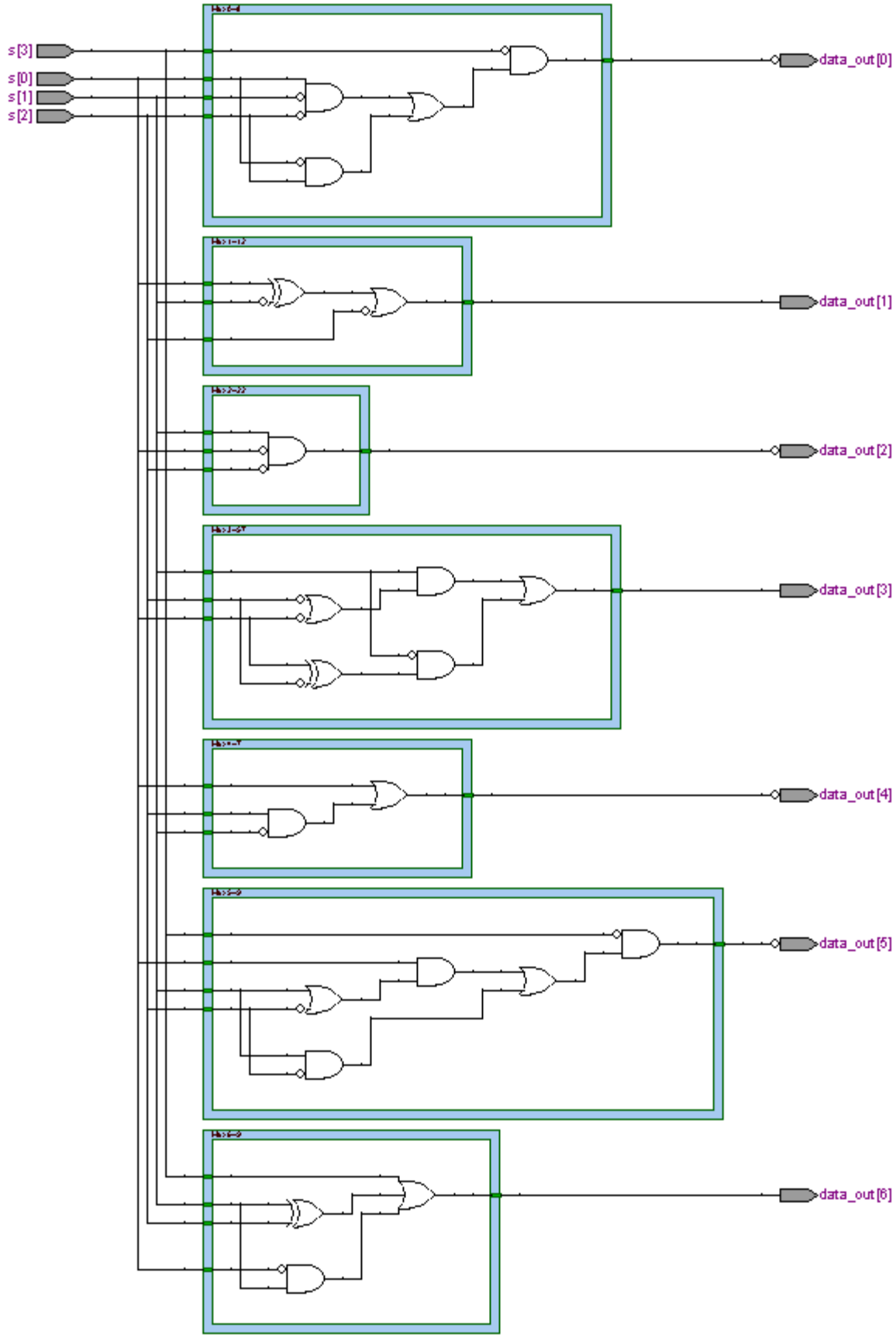


Figura 49 - Circuit sintetizat cu ajutorul modulului VHDL din figura 48.

2.6 Circuite logice combinaționale – module structurale

În acest subcapitolul vor fi descrise circuite logice combinaționale sub formă de module structurale. Un astfel de modul aste alcătuit din module mai mici, care la rândul lor sunt alcătuite din module structurale mai mici sau module comportamentale. Acest subcapitol evidențiază felul în care poate fi pusă în practică regula celor trei E. Trebuie evitată combinarea stilului comportamental și structural în același modul.

2.6.1 Multiplexor 4 la 1 pe 1 bit

În figura 50 este evidențiat un multiplexor 2 la 1 pe 1 bit ca modul comportamental. Cu ajutorul acestui modul se va crea un multiplexor 4 la 1 pe 1 bit. Acest multiplexor va fi alcătuit din 4 copii ale multiplexorului 2 la 1. O copie a unui modul se numește **instanță**. Modulul structural pentru un multiplexor 4 la 1 pe 1 bit este evidențiat în figura 51. Tabelul de adevăr este evidențiat în figura 52. În figura 53 este reprezentat circuitul sintetizat cu ajutorul modulului VHDL din figura 50.

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

entity mux21 is
        port ( a, b : in std_logic; sel  : in  std_logic;
               c    : out std_logic );
end mux21;

architecture mux21_arch of mux21 is
begin

c <= a when sel = '1' else b;

end mux21_arch;
```

Figura 50 – Multiplexor 2 la 1 pe 1 bit. Modul comportamental.

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

entity mux41 is
        port ( a, b, c, d : in std_logic;
               sel        : in  std_logic_vector(1 downto 0);
               e          : out std_logic );
```

```
end mux41;

architecture mux41_arch of mux41 is

component mux21 is
      port ( a, b : in std_logic;
               sel  : in  std_logic;
               c    : out std_logic );
end component;

signal intermed : std_logic_vector(1 downto 0);

begin

part1 : mux21 port map (a => a, b => b, sel => sel(0), c => intermed(0));

part2 : mux21 port map (a => c, b => d, sel => sel(0), c => intermed(1));

part3 : mux21 port map (a => intermed(0), b => intermed(1), sel => sel(1), c => e);

end mux41_arch;
```

Figura 51 – Multiplexor 4 la 1 pe 1 bit. Modul structural.

$sel(1)$	$sel(0)$	e
0	0	a
0	1	b
1	0	c
1	1	d

Figura 52 – Tabel de adevăr pentru un multiplexor 4 la 1 pe 1 bit.

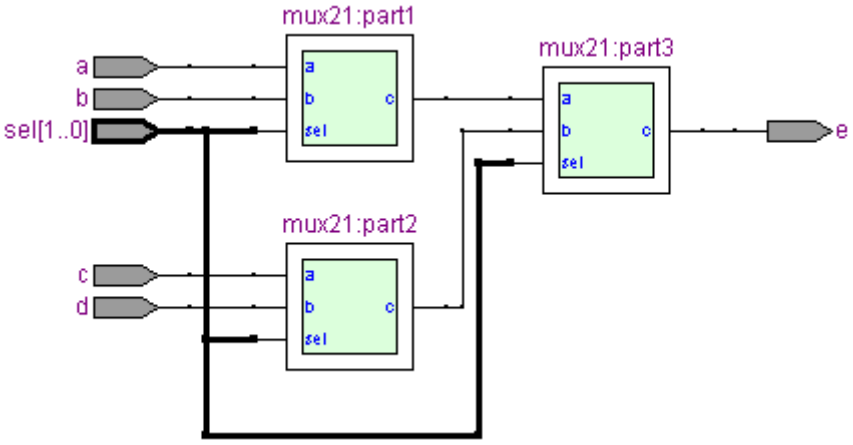


Figura 53 – Circuit sintetizat folosind modulul VHDL din figura 51.

2.6.2 Multiplexor 2 la 1 pe 4 biți

Cu ajutorul modulului VHDL din figura 50, ce evidențiază un multiplexor 2 la 1 pe 1 bit, se va crea un multiplexor 2 la 1 pe 4 biți. Se vor folosi patru instanțe ale multiplexorului 2 la 1 pe 1 bit. În figura 54 este evidențiată structura multiplexorului 2 la 1 pe 4 biți. În figura 55 este reprezentat circuitul sintetizat folosind modulul VHDL din figura 54.

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

entity mux21_4 is

        port (
              a, b : in std_logic_vector(3 downto 0);
              sel   : in std_logic;
              c     : out std_logic_vector(3 downto 0)
             );

end mux21_4;

architecture mux21_4_arch of mux21_4 is


component mux21 is

     port ( a, b : in std_logic;
            sel   : in  std_logic;
            c     : out std_logic
           );

end component;

begin

part1 : mux21 port map (a => a(0), b => b(0), sel => sel, c => c(0));

part2 : mux21 port map (a => a(1), b => b(1), sel => sel, c => c(1));

part3 : mux21 port map (a => a(2), b => b(2), sel => sel, c => c(2));

part4 : mux21 port map (a => a(3), b => b(3), sel => sel, c => c(3));

end mux21_4_arch;
```

Figura 54 – Multiplexor 2 la 1 pe 4 biți. Modul structural.

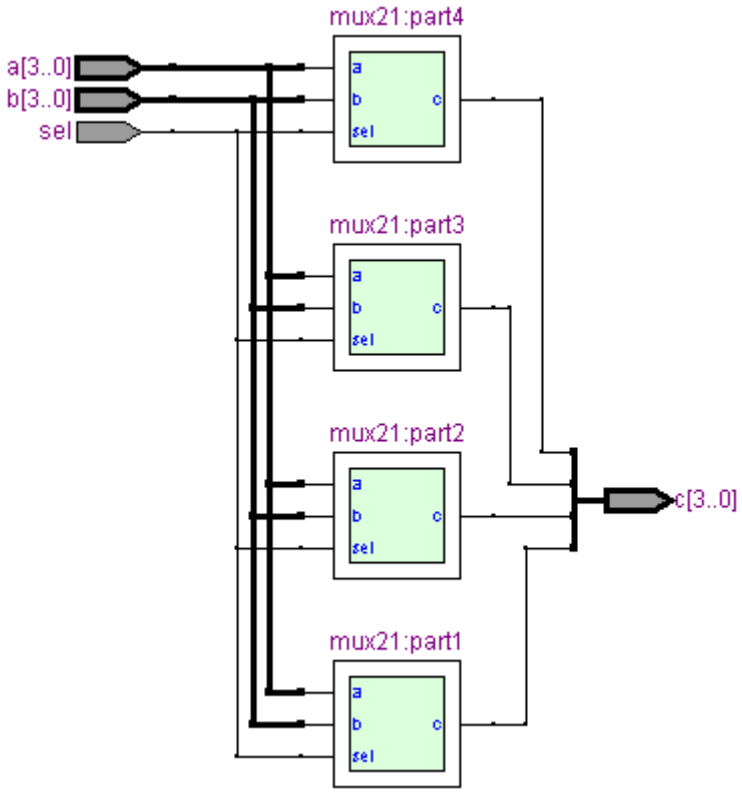


Figura 55 – Circuit sintetizat cu ajutorul modulului VHDL din figura 54.

2.6.3 Multiplexor 2 la 1 pe 8 biți

Cu ajutorul multiplexorului 2 la 1 pe 4 biți din figura 54 se va crea un multiplexor 2 la 1 pe 8 biți. Acest modul va cuprinde 2 instanțe ale multiplexorului 2 la 1 pe 4 biți. În figura 56 este evidențiată structura unui multiplexor 2 la 1 pe 8 biți. În figura 57 este reprezentat circuitul sintetizat folosind modulul din figura 56.

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

entity mux21_8 is

        port (
              a, b, c, d  : in std_logic_vector(7 downto 0);
              sel         : in std_logic;
              e           : out std_logic_vector(7 downto 0)
             );

end mux21_8;

architecture mux21_8_arch of mux21_8 is
```

```
component mux21_4 is

        port (
              a, b : in std_logic_vector(3 downto 0);
              sel  : in std_logic;
              c    : out std_logic_vector(3 downto 0)
             );

end component;

signal intermed : std_logic_vector(7 downto 0);

begin

part1 : mux21_4 port map (a => a(3 downto 0), b => b(3 downto 0), sel => sel,
                          c => intermed(3 downto 0));

part2 : mux21_4 port map (a => a(7 downto 4), b => b(7 downto 4), sel => sel,
                          c => intermed(7 downto 4));

c <= intermed;

end mux21_8_arch;
```

Figura 56 – Multiplexor 2 la 1 pe 8 biți. Modul structural.

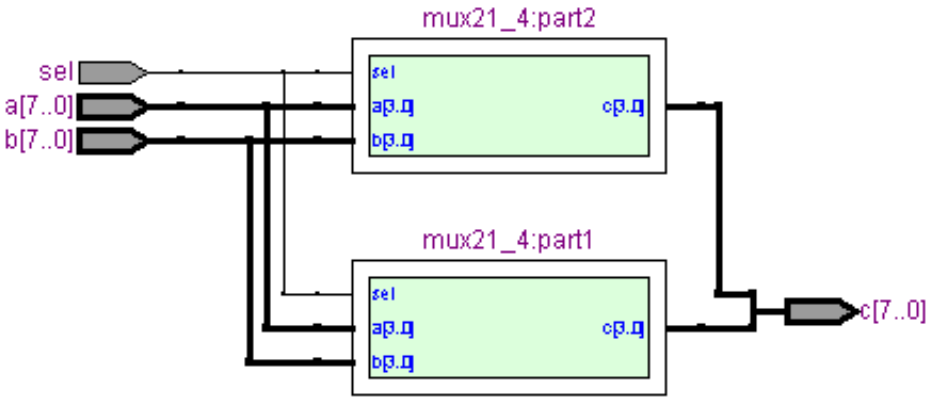


Figura 57 – Circuit sintetizat folosind modulul VHDL din figura 56.

2.6.4 Circuite de deplasare

Cu ajutorul multiplexorului 4 la 1 pe 1 bit, descris în figura 51, se vor crea circuitul de deplasare logică la stânga (figura 58), circuitul de deplasare logică la dreapta (figura 60), circuitul de deplasare aritmetică la dreapta (figura 62). În figurile 59, 61 și 63 sunt reprezentate circuitele sintetizate folosind modulele VHDL din figurile 58, 60, 62.

```
library ieee; use ieee.std_logic_1164.all;
entity sll_4 is
        port ( a       : in std_logic_vector(3 downto 0);
               shamt : in std_logic_vector(1 downto 0);
               b       : out std_logic_vector(3 downto 0));
end sll_4;
architecture sll_4_arch of sll_4 is
component mux41 is
        port ( a, b, c, d : in std_logic; sel  : in  std_logic_vector(1 downto 0);
               e          : out std_logic );
end component;
signal nul : std_logic;
begin
nul <= '0';
part1 : mux41 port map (a => a(3), b => a(2), c => a(1),  d => a(0), sel => shamt, e => b(3));
part2 : mux41 port map (a => a(2), b => a(1), c => a(0), d => nul,  sel => shamt, e => b(2));
part3 : mux41 port map (a => a(1), b => a(0), c => nul,  d => nul,  sel => shamt, e => b(1));
part4 : mux41 port map (a => a(0), b => nul,  c => nul,  d => nul,  sel => shamt, e => b(0));
end sll_4_arch;
```

Figura 58 – Circuit de deplasare logică la dreapta. Modul structural.

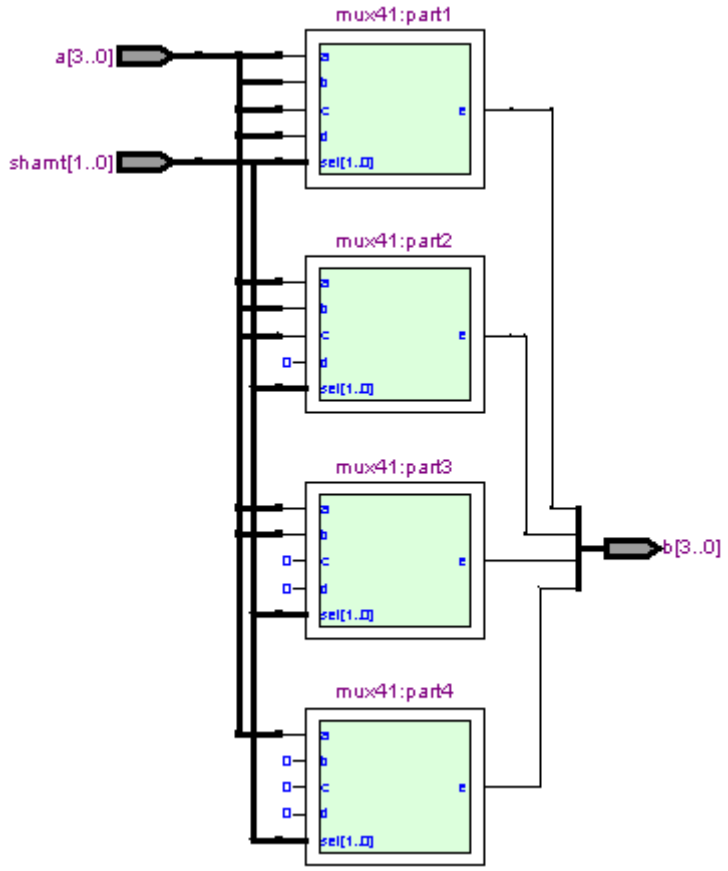


Figura 59 – Circuit sintetizat folosind modulul VHDL din figura 58.

```
library ieee; use ieee.std_logic_1164.all;
entity srl_4 is
        port ( a       : in std_logic_vector(3 downto 0);
               shamt : in std_logic_vector(1 downto 0);
               b       : out std_logic_vector(3 downto 0));
end srl_4;
architecture srl_4_arch of srl_4 is
component mux41 is
       port ( a, b, c, d : in std_logic; sel  : in  std_logic_vector(1 downto 0); e : out std_logic );
end component;
signal nul : std_logic;
begin
nul <= '0';
part1 : mux41 port map (a => a(3), b => nul,  c => nul,  d => nul,  sel => shamt, e => b(3));
part2 : mux41 port map (a => a(2), b => a(3), c => nul,  d => nul,  sel => shamt, e => b(2));
part3 : mux41 port map (a => a(1), b => a(2), c => a(3), d => nul,  sel => shamt, e => b(1));
part4 : mux41 port map (a => a(0), b => a(1), c => a(2), d => a(3), sel => shamt, e => b(0));
end srl_4_arch;
```

Figura 60 – Circuit de deplasare logică la stânga. Modul structural.

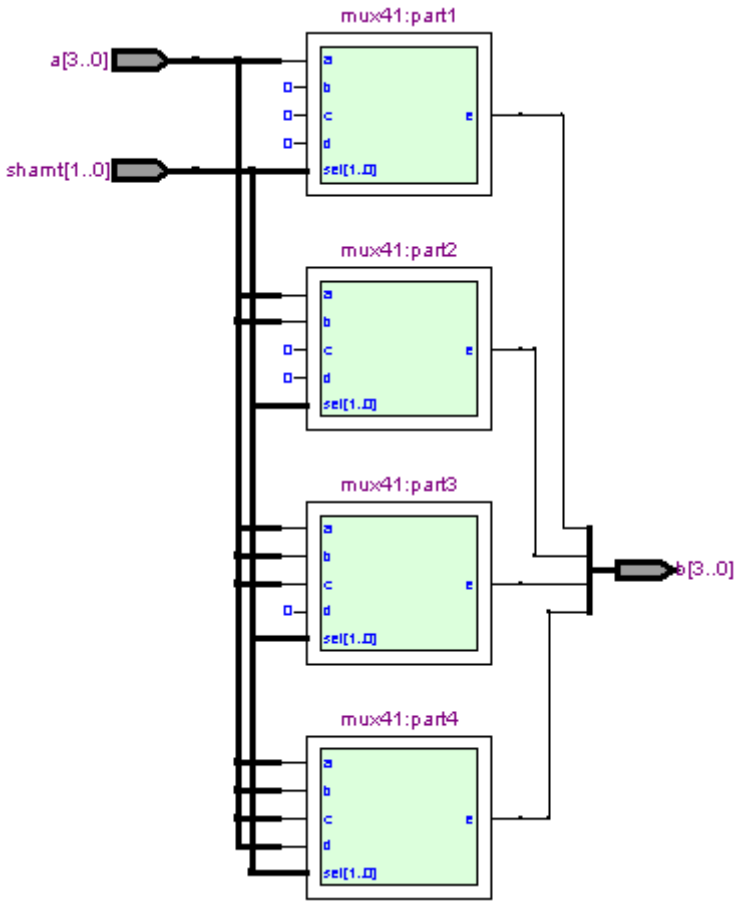


Figura 61 – Circuit sintetizat folosind modulul VHDL din figura 60.

```
library ieee; use ieee.std_logic_1164.all;
entity sra_4 is
        port ( a       : in std_logic_vector(3 downto 0);
               shamt : in std_logic_vector(1 downto 0);
               b       : out std_logic_vector(3 downto 0));
end sra_4;

architecture sra_4_arch of sra_4 is

component mux41 is
       port ( a, b, c, d : in std_logic; sel  : in  std_logic_vector(1 downto 0); e : out std_logic );
end component;

begin

part1 : mux41 port map (a => a(3), b => a(3), c => a(3), d => a(3), sel => shamt, e => b(3));
part2 : mux41 port map (a => a(2), b => a(3), c => a(3), d => a(3), sel => shamt, e => b(2));
part3 : mux41 port map (a => a(1), b => a(2), c => a(3), d => a(3), sel => shamt, e => b(1));
part4 : mux41 port map (a => a(0), b => a(1), c => a(2), d => a(3), sel => shamt, e => b(0));

end sra_4_arch;
```

Figura 62 – Circuit de deplasare aritmetică la dreapta. Modul structural.

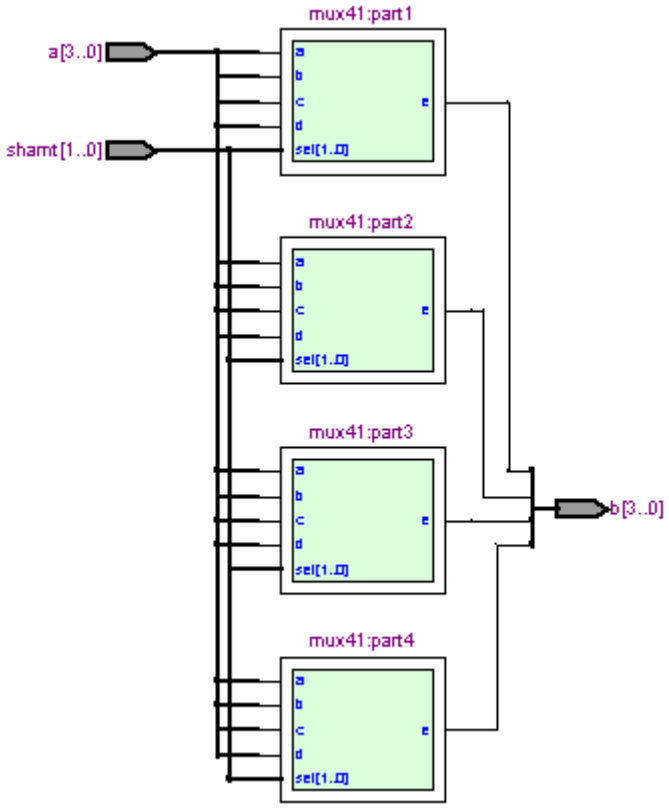


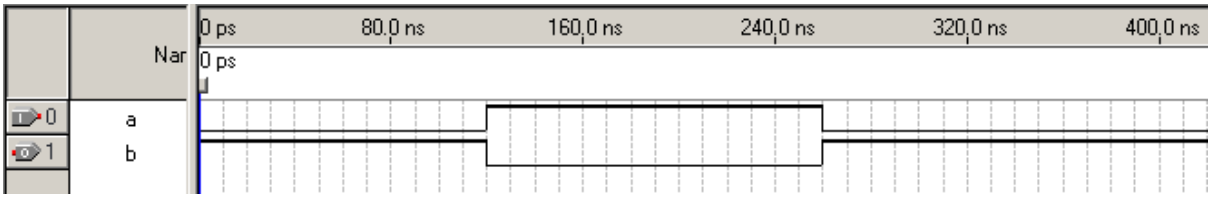
Figura 63 – Circuit sintetizat folosind modulul VHDL din figura 62.

2.7 Circuite logice combinaționale – comportament temporal

Ultimele două subcapitole au evidențiat funcționarea ideală a circuitelor logice combinaționale. În primul capitol s-a definit că un modul VHDL este caracterizat printr-o **funcționare ideală** și un **comportament temporal** sau o **funcționare reală**. În acest subcapitol va fi evidențiat comportamentul temporal (funcționarea reală) pentru circuitele logice combinaționale.

În figura 64 este evidențiată o **diagramă temporală ideală** unde este reprezentată funcționarea ideală a unui invertor. Atunci când valoarea semnalului de intrare se modifică, în același timp se va modifica și valoarea semnalului de ieșire. Se observă că o tranziție din valoarea logică 0 în valoarea logică 1 sau invers pentru semnalul de intrare va produce o tranziție instantanee din valoarea logică 1 în valoarea logică 0 sau invers pentru semnalul de ieșire.

În figura 65 este evidențiată o **diagramă temporală reală** unde este reprezentată funcționarea reală a unui invertor. Se observă existența unei **întârzieri** între tranziția valorii semnalului la intrare și tranziția valorii semnalului la ieșire, în concluzie diagrama temporală reală evidențiază răspunsul tranzitoriu al semnalelor. Valoarea semnalului de ieșire se modifică după 10 ns de la modificarea valorii semnalului de intrare.



Figură 64 – Diagrama temporală ideală a unui invertor.

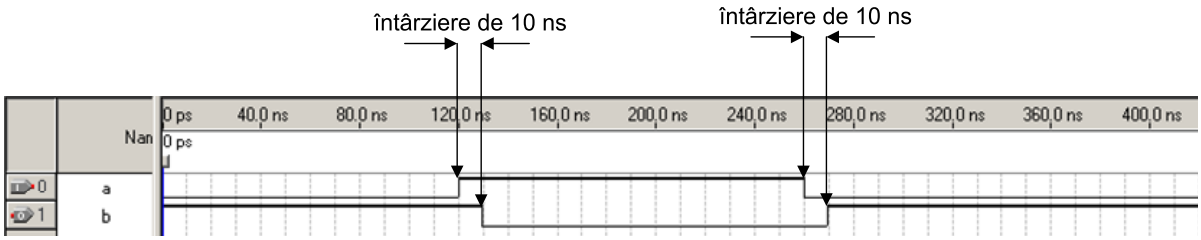


Figura 65 – Diagrama temporală reală a unui invertor.

Există două tipuri de întârzieri: **întârziere de contaminare** și **întârziere de propagare**. Întârzierea de contaminare, notată cu t_{cd}, reprezintă întârzierea minimă din

circuitul logic combinațional. Acestă întârziere se măsoară din momentul când valoarea de intrare se modifică până în momentul când valoare de ieșire începe să se modifice, datorită modificării valorii de intrare. Întârzierea de propagare, notată cu $\boldsymbol{t_{pd}}$, reprezintă întârzierea maximă din circuitul logic combinațional. Acestă întârziere se măsoară din momentul când valoarea de intrare se modifică până în momentul când valoare de ieșire a ajuns la valoarea finală (valoarea stabilă). În figura 65 se observă că $t_{cd} = t_{pd}$.

În momentul în care un inginer vorbește de calcularea întârzierii într-un circuit logic combinațional, în general el se referă la valoarea maximă a întârzierii, adică întârzierea de propagare. În figura 66 sunt reprezentate cele două tipuri de întârzieri pentru un invertor pe 4 biți. În acest caz cei doi timpi nu mai sunt egali. Producătorii de porți logice fundamentale furnizează în catalogul de produse aceste întârzieri.

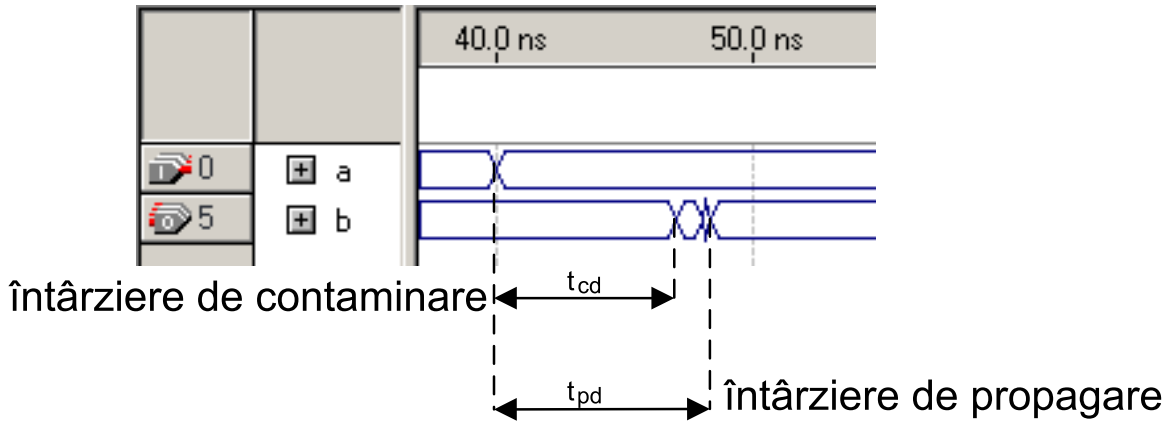


Figura 66 – Întârzierea de contaminare și întârzierea de propagare.

Cele două tipuri de întârzieri se manifestă atât pentru porți logice fundamentale cât și pentru circuite logice combinaționale. Întârzierile pentru astfel de circuite sunt determinate de propagarea semnalelor pe rutele ce fac legătura între intrare și ieșire. Întârzierea de contaminare este determinată de **ruta cea mai scurtă** de la intrare către ieșire, iar întârzierea de propagare este determinată de **ruta cea mai lungă** (**ruta critică**) de la intrare către ieșire.

În figura 67 este reprezentat un circuit logic combinațional unde sunt evidențiate cele două rute de propagare. Cea mai scurtă rută pornește de la intrarea **c**, trece prin porta **OR** și se oprește la ieșirea **d**. Această rută este cea mai rapidă, deoarece pe aici avem cea mai mică întârziere. Ruta critică pornește de la intrarea **a**, trece prin poarta **NOT**, apoi prin poarta **AND**, după care prin poarta **OR** și se oprește la ieșirea **d**. Această rută stabilește viteza

maximă la care circuitul poate funcționa, deoarece pe aici avem cea mai mare întârziere. Cu cât acestă întârziere este mai mică cu atât viteza de procesare a circuitului este mai mare.

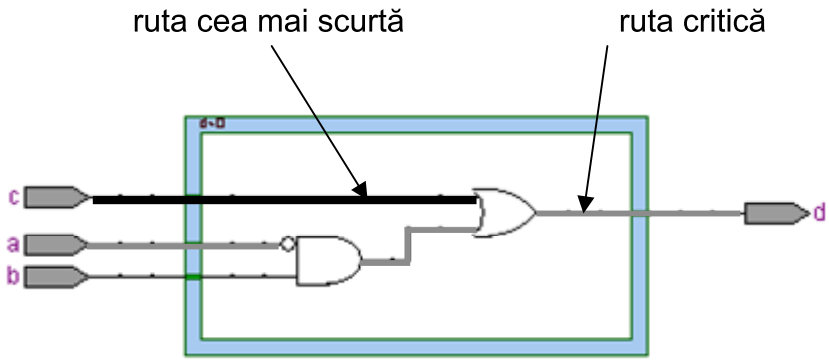


Figura 67 – Reprezentarea celor două rute pentru cele două tipuri de întârzieri.

Într-un circuit logic combinațional întârzierea de contaminare este reprezentată de suma întârzierilor de contaminare ale elementelor logice ce se găsesc pe ruta cea mai scurtă. Întârzierea de propagare este reprezentată de suma întârzierilor de propagare ale elementelor logice ce se găsesc pe ruta critică. Pentru circuitul din figura 67 avem că $t_{cd} = t_{cd\_OR}$, iar $t_{pd} = t_{pd\_NOT} + t_{pd\_AND} + t_{pd\_}OR$.

Capitolul 3

Sintetizarea circuitelor logice secvențiale sincrone

Sintetizarea circuitelor logice secvențiale sincrone

Circuitele logice secvențiale sunt circuite cu memorie. Ieșirea acestor circuite, la un moment dat, depinde de valoarea actuală a datelor de la intrare și de valorile anterioare ale intrării. În limbajul VHDL comportamentul acestor circuite este descris cu ajutorul **declarațiilor secvențiale** prin crearea de **procese**. Procesul este o structură specială a limbajului VHDL ce oferă posibilitatea descrierii comportamenului circuitelor logice secvențiale. Latch-ul, flip-flop-ul, registrul, counter-ul, registrul de deplasare sunt circuite logice secvențiale fundamentale. Acestea sunt împărțite în două categorii: asincrone și sincrone. În acest îndrumar se vor prezenta doar circuite logice secvențiale sincrone. În acest capitol va fi evidențiat comportamentul ideal și comportamentul temporal pentru această categorie de circuite logice secvențiale.

3.1 Procese

Un proces este o structură a limbajului VHDL cu ajutorul căreia se poate descrie comportamentul unui circuit logic secvențial și nu numai. În funcție de complexitatea circuitului se pot folosi unul sau mai multe procese. Procesele se definesc și se utilizează în corpul unei arhitecturi. După cum se știe corpul unei arhitecturi este o zonă simultană. Sintetizatorul va recunoaște procesele din cadrul modulelor VHDL ca declarații paralele. Cu ajutorul proceselor se poate descrie comportamentul latch-urilor, flip-flop-urilor, registrelor și counter-elor. Forma generală a unui proces este evidențiată în figura 1. Într-un proces se pot folosi declarații secvențiale. Acestea vor fi evidențiate în subcapitolul următor. Ordinea declarațiilor într-un proces contează enorm. Declarațiile sunt evaluate de sus în jos.

```
PROCESS ( numele_semnalului, numele_semnalului ... ) -- listă de senzitivitate

BEGIN
        [declarații simple]
        [declarații if/elsif/else]
        [declarații case]
```

```
END PROCESS [ numele procesului ];
```

Figura 1 – Forma generală a unui proces.

După cum se observă în figura 1 pentru un proces trebuie declarată o listă de senzitivitate. Lista de senzitivitate cuprinde condițiile pentru care procesul va fi executat. Aceste condiții sunt reprezentate sub forma unor semnale. Procesul este executat tot timpul când semnalele din listă își modifică valoarea. Până la o modificare ulterioară a valorii semnalelor procesul memorează starea actuală, memorează valorile semnalelor din circuit. Exemple de circuite logice secvențiale descrise cu ajutorul proceselor vor fi prezentate în subcapitolele următoare.

3.2 Declarații secvențiale

Comportamentul unui circuit logic secvențial poate fi descris cu ajutorul unor declarații secvențiale. Locul acestor declarații este în interiorul unui proces între begin și end process. Există două tipuri de declarații secvențiale: declarația if/elsif/else și declarația case. Trebuie menționat că aceste două tipuri de declarații se pot imbrica.

3.2.1 Declarația if/elsif/else

În figura 2 este prezentată forma generală a declarației if/elsif/else. Acestă declarație impune o anumită prioritate asupra ramurilor ei, prioritate ce scade de sus în jos. Acestă declarație este asemănătoare cu atribuirea condițională din capitolul anterior. Condițiile logice se vor scrie cu ajutorul operatorilor relaționali.

```
IF prima_condiție_logică THEN
        declarații;
        declarații;
ELSIF a_doua_condiție_logică THEN
        declarații;
        declarații;
ELSIF …
            .
```

```
                .
                .
ELSE   declarații;
          declarații;
END IF;
```

Figura 2 – Forma generală a declarației if/elsif/else.

Exemplul din figura 3 evidențiază utilizarea declarației if/elsif/else. Modulul VHDL descrie comportamentul unui circuit logic secvențial oarecare.

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

entity seq_circ is
        port( clk            : in std_logic;
              data_input   : in std_logic_vector(3 downto 0);
               data_output  : out std_logic_vector(3 downto 0) );
end seq_circ;

architecture seq_circ_arch of seq_circ is
signal intermed : std_logic_vector(3 downto 0);
begin
        process(clk)
        begin
            if (clk'event and clk = '1') then
                intermed <= intermed xor data_input;
             end if;
        end process;
                data_output <= intermed;
end seq_circ_arch;
```

Figura 3 – Utilizarea declarației if/elsif/else.

3.2.2 Declarația case

Forma generală a declarației case este evidențiată în figura 4. Acestă declarație este asemănătoare cu atribuirea selectată din capitolul anterior. Și în acest caz valorile constante trebuie să se excludă reciproc. Valoarea constantă poate fi: o singură valoare, câteva valori separate prin simbolul | (or = sau) sau o mulțime de valori (discrete).

```
CASE expresie_logică IS

    WHEN valoare_constantă =>
            declarații;
            declarații;
    WHEN valoare_constantă =>
            declarații;
            declarații;
                .
                .
                .
    WHEN OTHERS =>
            declarații;
            declarații;

END CASE;
```

Figura 4 – Forma generală a declarației case.

În figura 5 este evidențiat comportamentul unui circuitului logic secvențial oarecare.

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

entity seq_circ is
        port(
            enable      : in std_logic;
```

```
                data_input   : in std_logic_vector(3 downto 0);
                data_output : out std_logic_vector(3 downto 0)
                 );
end seq_circ;

architecture seq_circ_arch of seq_circ is
signal intermed : std_logic_vector(3 downto 0);
begin
        process(clk)
        begin
                case enable is
                        when '1' => intermed <= intermed xor data_input;
                end case;
        end process;
        data_output <= intermed;
end seq_circ_arch;
```

Figură 5 – Utilizarea declarației case.

3.3 Circuite logice secvențiale sincrone – comportament ideal

3.3.1 Flip-flop de tip D

Flip-flop-ul este circuitul logic secvențial sincron cu ajutorul căruia se pot alcătui toate celelalte circuite logice secvențiale sincrone. Pe parcursul acestui îndrumar se va folosi flip-flop-ul de tip D, deoarece este cel mai utilizat flip-flop. Flip-flop-ul de tip D are o intrare de date, o intrare de sincronizare și o ieșire de date. În figura 6 este evidențiat comportamentul acestui circuit cu ajutorul unui modul VHDL. În figura 7 este reprezentat circuitul sintetizat.

```
library ieee; use ieee.std_logic_1164.all;
entity dff_circ is
        port ( d, clk : in std_logic;
                q       : out std_logic );
```

```
end dff_circ;
architecture dff_circ_arch of dff_circ is
begin
process(clk)
begin
      if (clk'event and clk = '1') then
          q <= d;
      end if;
end process;
end dff_circ_arch;
```

Figura 6 – Modul ce descrie comportamentul flip-flop-ului de tip D.

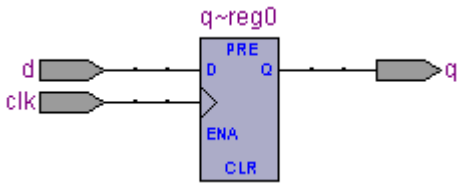


Figura 7 – Circuit sintetizat ulitizând modulul VHDL din figura 6.

Diagrama temporală ideală pentru flip-flop-ului de tip D este evidențiată în figura 8. Acest flip-flop transferă datele de la intrarea d către ieșirea q pe frontul crescător al tactului. Frontul crescător al tactului este evidențiat prin comanda clk'event and clk = '1'. Flip-flop-ul memorează datele transferate până în momentul când pe intrarea de sincronizare sesizează un nou front crescător. Dacă în acest moment datele de la intrarea d se modifică, flip-flop-ul transferă către ieşire noile date, dacă datele de la intrarea d nu se modifică flip-flop-ul va transfera vechile date.

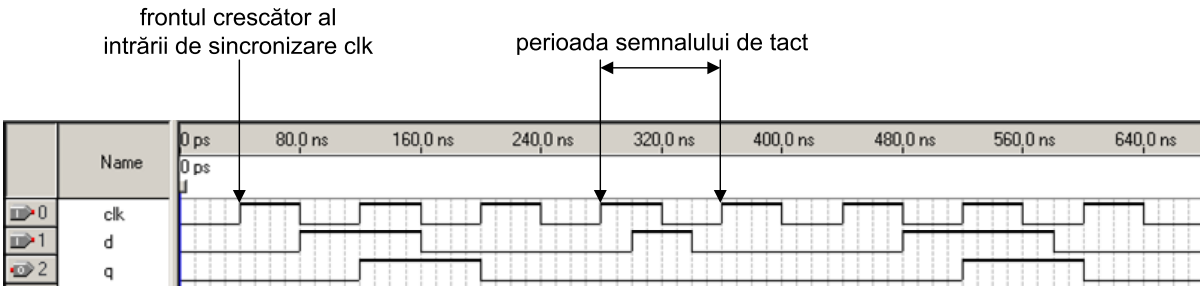


Figura 8 – Diagrama temporală ideală pentru flip-flop-ul de tip D.

3.3.2 Flip-flop de tip D cu reset asincron

Acest flip-flop are în plus o intrare de reset. Acestă intrare nu va fi sincronizată cu semnalul de tact, adică indiferent de valoarea logică a semnalului de tact când intrarea de reset va avea valoarea logică 1 va șterge (va reseta) datele din flip-flop. Dacă intrarea de reset va avea valoarea logică 0, flip-flop-ul va funcționa normal. În figura 9 este evidențiat comportamentul flip-flop-ului de tip D cu reset asincron printr-un modul VHDL. În figura 10 este reprezentat circuitul sintetizat, iar în figura 11 este prezentată diagrama temporală ideală pentru acest circuit.

```
library ieee; use ieee.std_logic_1164.all;
entity dff_circ is
        port ( d, clk, rst : in std_logic; q : out std_logic );
end dff_circ;
architecture dff_circ_arch of dff_circ is
begin
process (clk, rst)
begin
      if (rst = '1') then
            q <= '0';
      elsif (clk'event and clk = '1') then
             q <= d;
      end if;
end process;
end dff_circ_arch;
```

Figura 9 – Modul ce descrie comportamentul flip-flop-ului de tip D cu reset asincron.

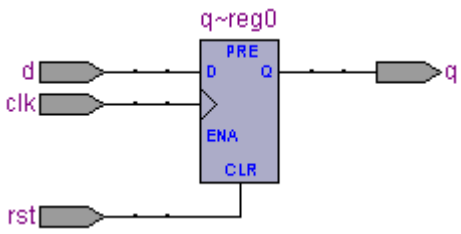


Figura 10 – Circuit sintetizat utilizând modulului VHDL din figura 9.

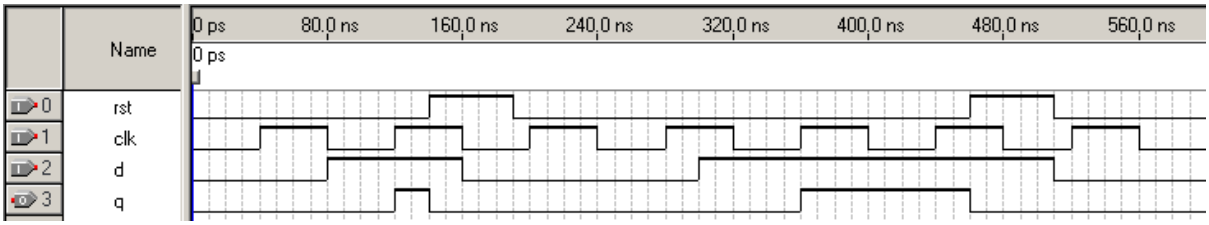


Figura 11 – Diagrama temporală ideală pentru flip-flop-ul de tip D cu reset asincron.

3.3.3 Flip-flop de tip D cu reset sincron

Pentru acest tip de circuit intrarea reset este sincronizată cu semnalul de tact. Datele vor fi șterse din flip-flop pe frontul crescător al semnalului de sincronizare numai în cazul când intrarea de reset are valoarea logică 1. Circuitul va funcționa normal dacă pe frontul crescător al semnalului de sincronizare intarea de reset va avea valoarea logic 0.

În figura 12 este evidențiat comportamentul flip-flop-ului de tip D cu reset sincron printr-un modul VHDL. În figura 13 este reprezentat circuitul sintetizat, iar în figura 14 este prezentată diagrama temporală ideală pentru acest circuit.

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

entity dff_circ is
        port (
                d, clk, rst : in std_logic;
                q           : out std_logic
              );
end dff_circ;

architecture dff_circ_arch of dff_circ is
begin
process (clk)
begin
      if (clk'event and clk = '1') then
         if (rst = '1') then
              q <= '0';
```

```
            else q <= d;
            end if;
        end if;
end process;
end dff_circ_arch;
```

Figura 12 - Modul ce descrie comportamentul flip-flop-ului de tip D cu reset sincron.

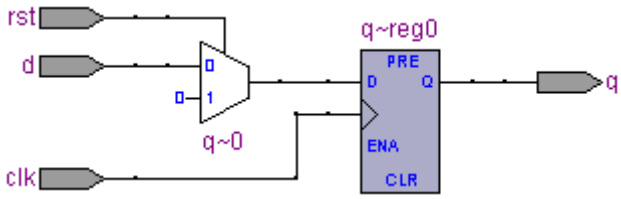


Figura 13 – Circuit sintetizat utilizând modulul VHDL din figura 12.

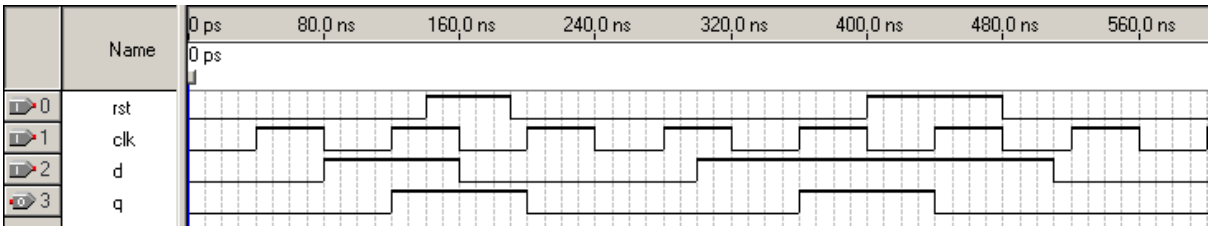


Figura 14 - Diagrama temporală ideală pentru flip-flop-ul de tip D cu reset sincron.

3.3.4 Flip-flop de tip D cu intrare de enable și reset asincron

Pentru acest circuit secvențial sincron vom aveam următoarea prioritate a intrărilor: intrarea de reset, intrarea de sincronizare și intrarea de enable. Intrarea de enable este sincronizată cu semnalul de tact.

Dacă intrarea de reset va avea valoarea logică 1, atunci flip-flop-ul va fi resetat, dacă intrarea de reset va avea valoarea logică 0 și dacă pe frontul crescător al semnalului de tact pe intrarea de enable avem valoarea logică 0 circuitul va fi blocat și va reține datele anterior memorate. Dacă intrarea de reset va avea valoarea logică 0 și dacă pe frontul crescător al semnalului de tact intrarea enable va avea valoarea logică 1 atunci circuitul va funcționa normal.

În figura 15 este evidențiat comportamentul flip-flop-ului de tip D cu intrare de enable și reset asincron printr-un modul VHDL. În figura 16 este reprezentat circuitul sintetizat, iar în figura 17 este prezentată diagrama temporală ideală pentru acest circuit.

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

entity dff_circ is
        port ( d, clk, rst, enable : in std_logic;
                q                  : out std_logic );
end dff_circ;

architecture dff_circ_arch of dff_circ is
begin
process (clk, rst)
begin
      if (rst = '1') then
              q <= '0';
      elsif (clk'event and clk = '1') then
            if (enable = '1') then
                q <= d;
            end if;
        end if;
end process;
end dff_circ_arch;
```

Figura 15 – Modul ce descrie comportamentul flip-flop-ului de tip D cu intrare de enable și reset asincron.

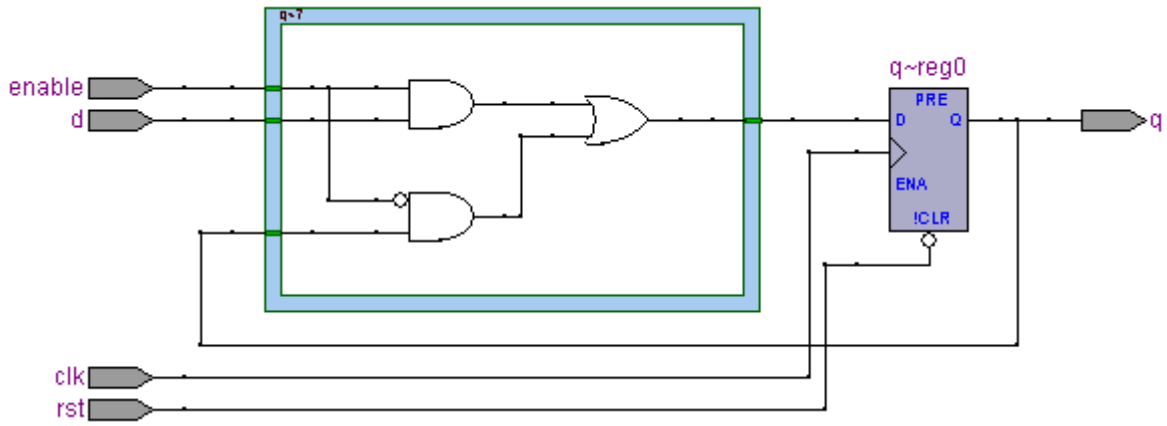


Figura 16 – Circuit sintetizat utilizând modulul VHDL din figura 15.

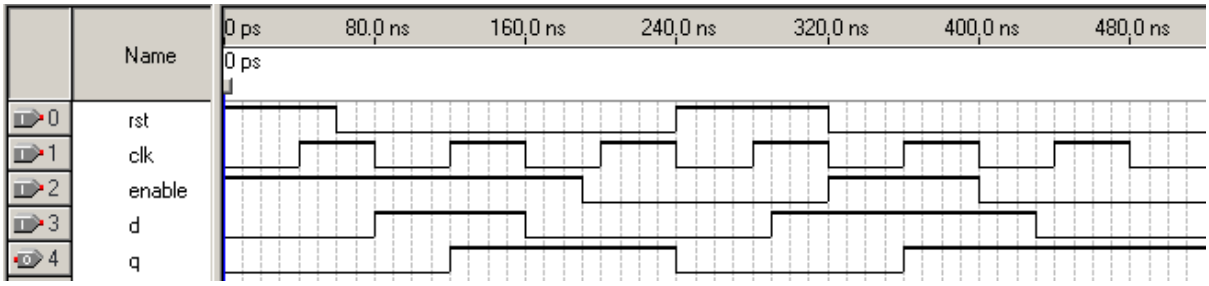


Figura 17 – Diagrama temporală ideală pentru flip-flop-ul de tip D cu intrare de enable și reset sincron.

3.3.5 Registru paralel

Registrul paralel este alcătuit dintr-un număr oarecare de flip-flop-uri de tip D cu intrare de enable și reset asincron. Dacă un registru paralel este alcătuit din 32 de flip-flop-uri, acesta poate să memoreze 32 de biți.

În figura 18 este evidențiat comportamentul unui registru paralel pe 4 biți printr-un modul VHDL. În figura 19 este reprezentat circuitul sintetizat, iar în figura 20 este prezentată diagrama temporală ideală pentru acest circuit. Acest registru paralel are o intrare de date pe 4 biți, o intrare de reset, o intrare de enable și o ieșire de date pe 4 biți. Dacă pe intrarea de reset avem valoarea logică 1 atunci datele memorate anterior în registru sunt șterse și înlocuite cu valoarea 0, dacă pe intrarea de reset avem valoarea logică 0 și dacă pe frontul crescător al semnalului de tact pe intrarea de enable avem valoarea logică 0 atunci circuitul este blocat și va reține valoarea memorată anterior. Dacă pe intrarea de reset avem valoarea logică 0 și dacă pe frontul crescător al semnalului de sincronizare pe intrarea de enable avem valoarea logică 1 atunci circuitul va funcționa normal.

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

entity reg_circ is
        port (
                d                : in std_logic_vector(3 downto 0);
              clk, rst, enable : in std_logic;
                q                : out std_logic_vector(3 downto 0)
              );
end reg_circ;
```

```
architecture reg_circ_arch of reg_circ is
begin

process (clk, rst)
begin
        if (rst = ’1’) then
                q <= (OTHERS => ’0’);
        elsif (clk’event and clk = ’1’) then
                if (enable = ’1’) then
                        q <= d;
                end if;
        end if;
end process;
end reg_circ_arch;
```

Figura 18 – Modul ce descrie comportamentul unui registru paralel pe 4 biți.

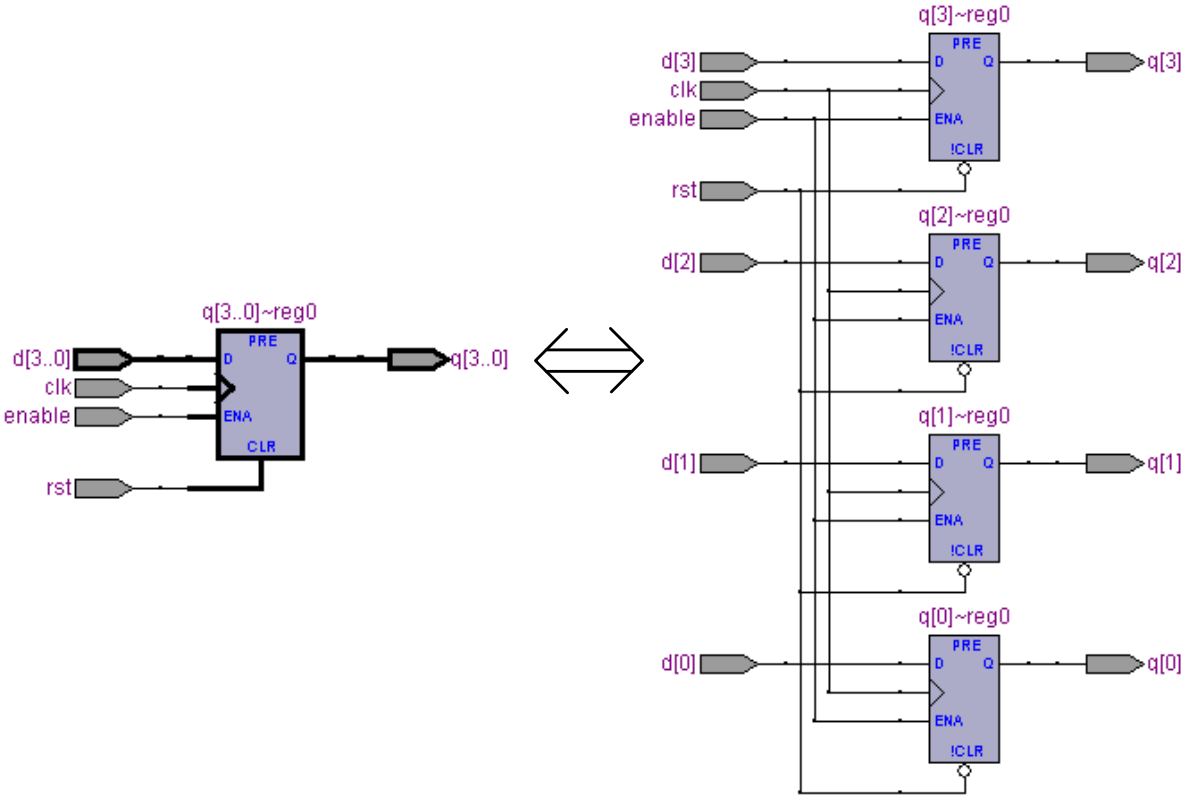


Figura 19 – Circuit sintetizat utilizând modulul VHDL din figura 18.

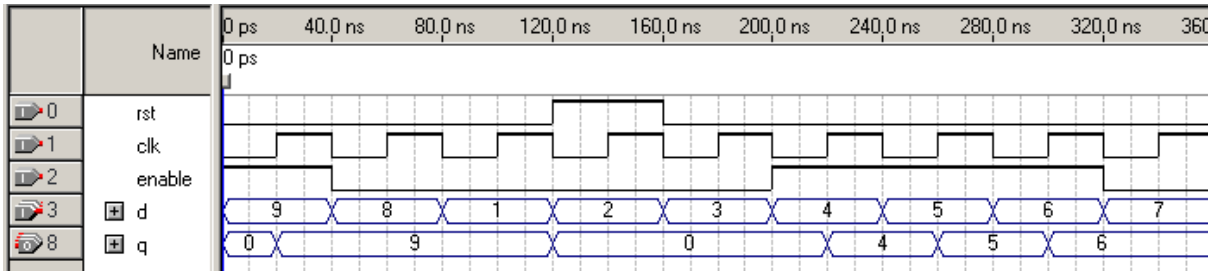


Figura 20 – Diagrama temporală ideală pentru un registru paralel pe 4 biți.

3.3.6 Registrul serial

Registrul serial este alcătuit dintr-un număr oarecare de flip-flop-uri de tip D legate în serie. Registrul serial se mai numește registru de deplasare sau registru serial-paralel. Acest circuit are o intrare serială de date (pe 1 bit), o intrare de reset, o intrare de sincronizare, o ieșire serială de date (pe 1 bit) și o ieșire paralelă pe N biți. Acest circuit poate deplasa datele de la intrarea serială de la stânga la dreapta și poate furniza N biți pe ieșirea paralelă.

În figura 21 este evidențiat comportamentul unui registru serial ce poate memora 4 biți printr-un modul VHDL. În figura 22 este reprezentat circuitul sintetizat, iar în figura 23 este prezentată diagrama temporală ideală pentru acest circuit.

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

entity reg_circ is
        port (
              sdata_input   : in std_logic;
              clk, rst      : in std_logic;
              sdata_output : out std_logic;
              data_output  : buffer std_logic_vector(3 downto 0)
             );
end reg_circ;

architecture reg_circ_arch of reg_circ is
begin
```

```
process (clk, rst)
begin
        if (rst = ’1’) then
                data_output <= (OTHERS => ’0’);
        elsif (clk’event and clk = ’1’) then
                data_output <= data_output(2 downto 0) & sdata_input;
        end if;
end process;

sdata_output <= data_output(3);

end reg_circ_arch;
```

Figura 21 – Modul ce descrie comportamentul unui registru serial.

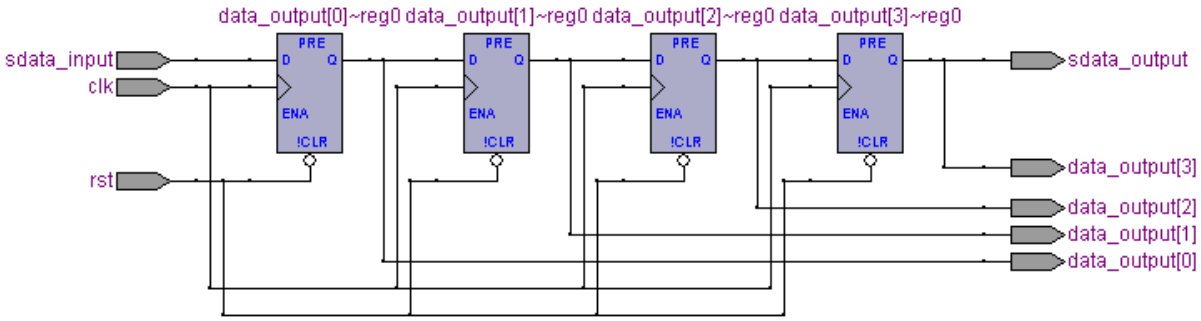


Figura 22 – Circuit sintetizat utilizând modulul VHDL din figura 21.

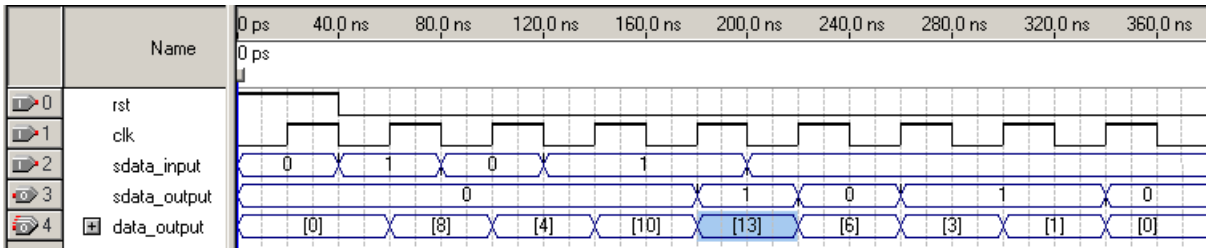


Figura 23 – Diagrama temporală ideală pentru un registru serial.

3.3.7 Registrul serial-paralel/paralel-serial

Un registru SP/PS deține o intrare serială de date, o intrare de reset, o intrare de sincronizare, o intrare de încărcare, o intrare paralelă, o ieșire paralelă și o ieșire serială de date. În figura 24 este evidențiat comportamentul unui registru SP/PS pe 4 biți printr-un

modul VHDL. În figura 25 este reprezentat circuitul sintetizat, iar în figura 26 este prezentată diagrama temporală ideală pentru acest circuit.

Intrarea de încărcare (load) este sincronizată cu semnalul de tact. Dacă pe intrarea de încărcare avem valoarea logică 1 registrul SP/PS, se comportă ca un registru paralel-serie și transferă pe frontul crescător al tactului datele de la intrarea paralelă către ieșirea paralelă. Datele de la intrarea paralelă mai pot fi deplasate către ieșirea serială. Dacă pe intrare de încărcare se găsește valoarea logică 0, registrul SP/PS se comportă ca un registru de deplasare.

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

entity reg_circ is
        port ( data_input     : in std_logic_vector(3 downto 0);
               sdata_input    : in std_logic;
               clk, rst, load : in std_logic;
               sdata_output : out std_logic;
               data_output  : buffer std_logic_vector(3 downto 0) );
end reg_circ;

architecture reg_circ_arch of reg_circ is
begin

process (clk, rst)
begin
      if (rst = '1') then
              data_output <= (OTHERS => '0');
      elsif (clk'event and clk = '1') then
            if (load = '1') then
               data_output <= data_input;
            else data_output <= data_output(2 downto 0) & sdata_input;
            end if;
```

```
        end if;
end process;
sdata_output <= data_output(3);
end reg_circ_arch;
```

Figura 24 – Modul ce descrie comportamenul registrului serial-paralel/paralel-serial.

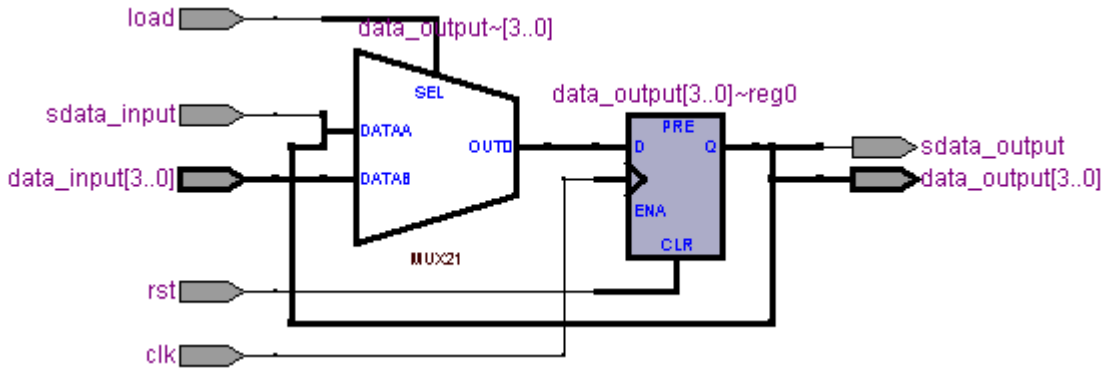


Figura 25 – Circuit sintetizat utilizând modulul VHDL din figura 24.

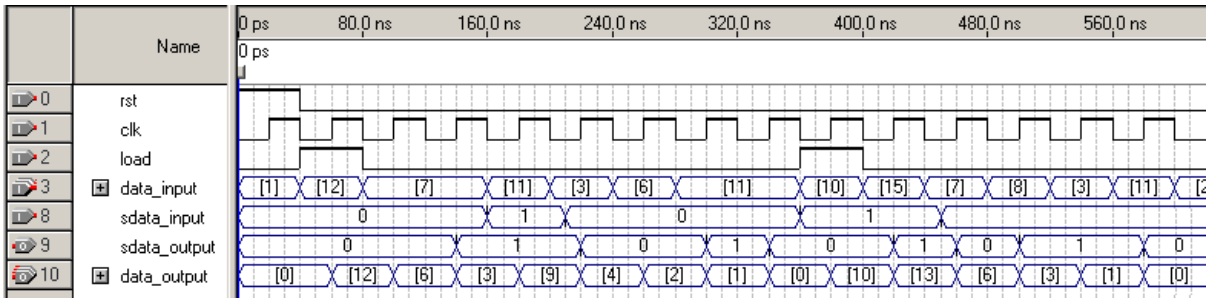


Figura 26 – Diagrama temporală ideală pentru un registru SP/PS.

3.3.8 Counter

Un counter deține o intrare de reset, o intrare de sincronizare o intrare de enable o intrare de load o intrare de date și o ieșire de date. În figura 27 este evidențiat comportamentul unui counter pe 4 biți printr-un modul VHDL. În figura 28 este reprezentat circuitul sintetizat, iar în figura 29 este prezentată diagrama temporală ideală pentru acest circuit. Un counter pe 4 biți numără de la 0 la $2^4 - 1$, cu pasul de o unitate. Counter-ul începe să numere din nou de la 0 atunci când ajunge la 15. Dacă pe intrarea de reset avem valoarea logică 1 atunci numărătoarea counter-ului este egală cu 0, dacă pe intrarea de reset avem valoarea logică 0 pe frontul crescător al tactului dacă pe intrarea de enable avem valoarea logică 0 circuitul memorează starea actuală, iar dacă intrarea pe intrarea de enable avem valoarea logică 1 atunci circuitul se comportă normal. Dacă pe intrarea de load avem valoarea

logică 1 atunci counterul este încărcat cu valoarea de pe intrarea de date, dacă pe intrarea de load avem valoarea logică 0 atunci circuitul se comportă normal.

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

entity counter_circ is
        port (
              data_input              : in std_logic_vector(3 downto 0);
              clk, rst, enable, load : in std_logic;
                data_output           : buffer std_logic_vector(3 downto 0)
              );
end counter_circ;

architecture counter_circ_arch of counter_circ is
begin

process (clk, rst)
begin
      if (rst = '1') then
                  data_output <= (OTHERS => '0');
      elsif (clk'event and clk = '1') then
              if (load = '1') then
                    data_output <= data_input;
              elsif (enable = '1') then
                    data_output <= data_output + '1';
              end if;
       end if;
end process;
end counter_circ_arch;
```

Figura 27 – Modul ce descrie comportamentul unui counter pe 4 biți.

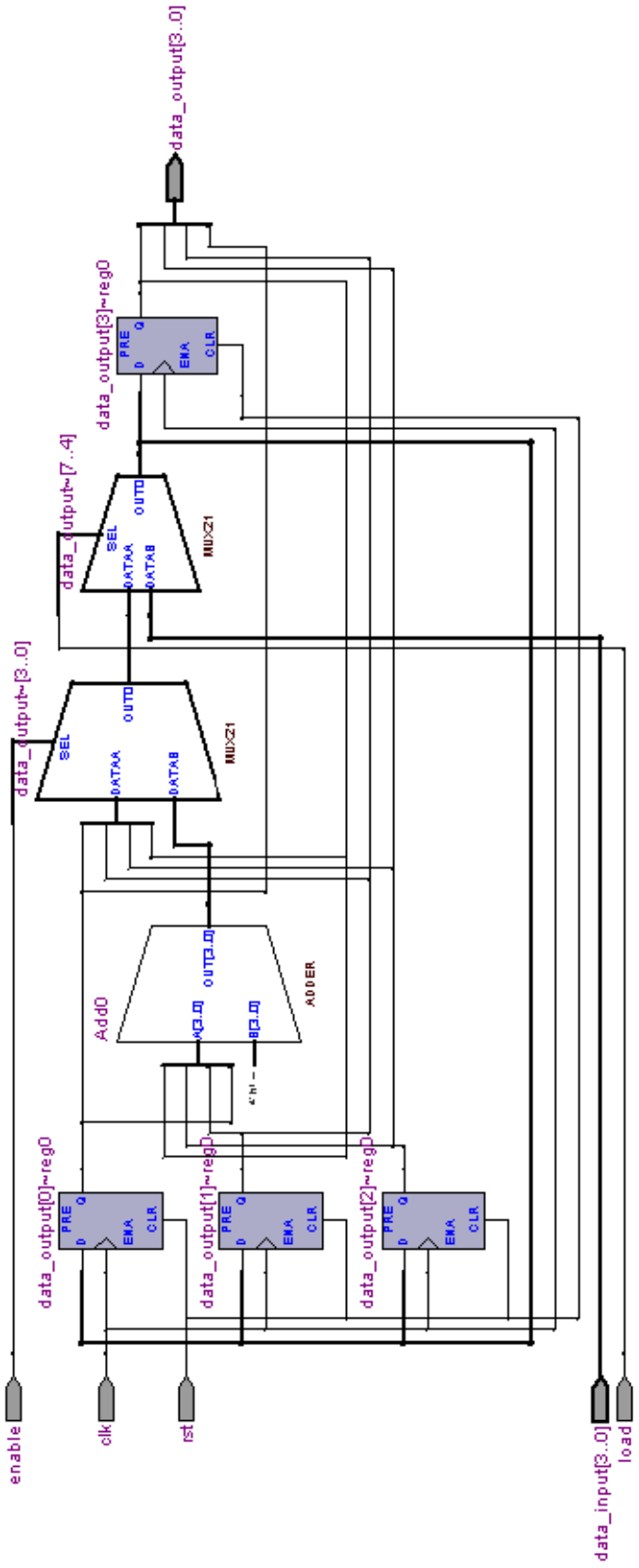


Figura 28 – Circuit sintetizat utilizând modulul VHDL din figura 27.

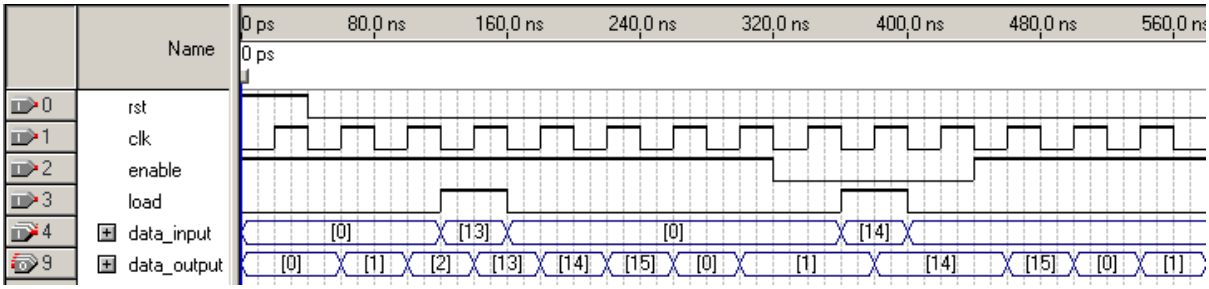


Figura 29 – Diagrama temporală ideală pentru un counter pe 4 biți.

3.4 Circuite logice secvențiale sincrone – comportament temporal

În figura 30 este prezentat comportamentul temporal al circuitelor logice secvențiale sincrone printr-o diagramă temporală reală. Aceste circuite logice eșantionează datele de la intrarea **d** pe frontul crescător al semnalului de tact. După cum se observă datele de la ieșirea **q** încep să se modifice după **întârzierea de contaminare** (t_{ccq}) și ajung la o valoare stabilă după **întârzierea de propagare** (t_{pcq}). Circuitele logice secvențiale sincrone vor eșantiona corect datele de la intrarea d, dacă aceste date vor fi stabile cel puțin un **timp de configurare** (t_{setup}) înainte de frontul crescător al semnalului de tact și vor rămâne stabile cel puțin un **timp de reținere** (t_{hold}) după frontul crescător al semnalului de tact.

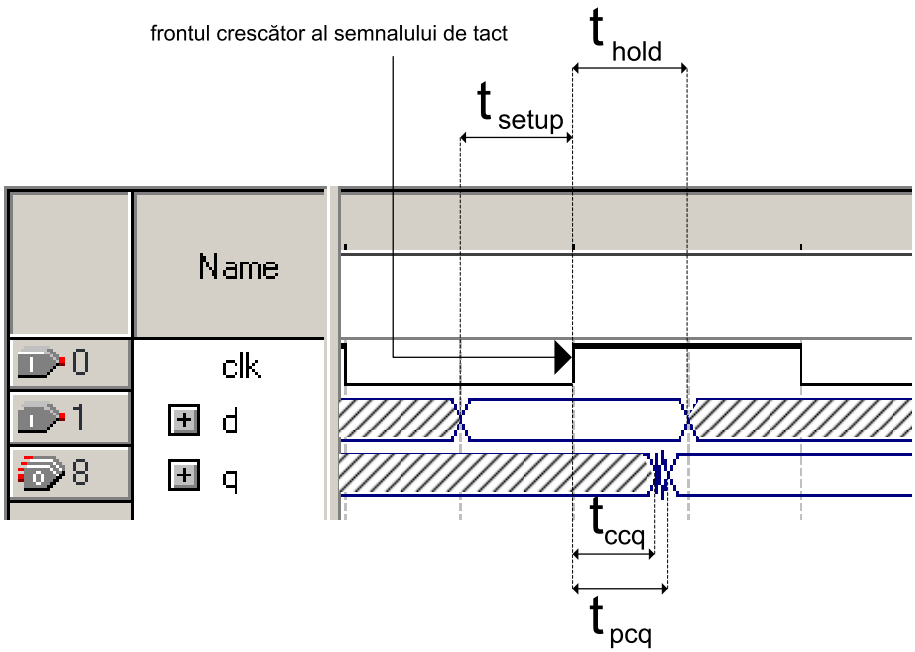


Figura 30 – Comportamentul temporal al circuitelelor logice secvențiale sincrone.

Capitolul 4

Sintetizarea circuitelor logice combinaționale

partea a doua

Sintetizarea circuitelor logice combinaționale partea a doua

Comportamentul circuitelelor logice combinaționale mai poate evidențiat cu ajutorul proceselor și a declarațiilor secvențiale. Crearea de module VHDL pentru circuitele logice combinaționale, utilizând procese și declarații secvențiale, este condiționată de respectarea unor reguli de bază. Aceste reguli sunt evidențiate în subcapitolul următor împreună cu exemple de circuite logice combinaționale fundamentale.

4.1 Circuite logice combinaționale – procese și declarații secvențiale

Pentru a descrie comportamentul circuitelor logice combinaționale cu ajutorul proceselor și declarațiilor secvențiale trebuie să se țină cont de două reguli de bază:

1. Lista de senzitivitate a proceselor trebuie să conțină toate semnalele de intrare in circuitul logic combinațional, care la un moment dat se pot modifica.
2. Declarațiile secvențiale trebuie să cuprindă toate combinațiile semnalelor de intrare pentru care semnalele de ieșire produc o anumită valoare.

În figura 1 este prezentat un exemplu de circuit logic combinațional evidențiat printr-un modul VHDL. Circuitul logic combinațional este un multiplexor 2 la 1 pe 4 biți. Comportamentul acestui circuit este evidențiat printr-o declarație case scrisă în interiorul unui proces.

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

entity mux21_4 is
        port ( a, b : in std_logic_vector(3 downto 0);
               sel  : in std_logic;
               c    : out std_logic_vector(3 downto 0) );
end mux21_4;

architecture mux21_4_arch of mux21_4 is
begin
```

```
process(a, b, sel)
begin
        case sel is
                when        '0'      => c <= a;
                when OTHERS => c <= b;
        end case;
end process;
end mux21_4_arch;
```

Figură 1 – Proces și declarație secvențială case pentru comportamenul multiplexorului 2 la 1 pe 4 biți.

Se observă că lista de senzitivități cuprince cele 3 semnale de intrare, iar declarația secvențială case cuprinde toate combinațiile semnalelor de intrare, deci cele două reguli de mai sus au fost îndeplinite.

În figura 2 este prezentat același circuit logic combinațional însă comportamentul acestuia este evidențiat printr-o declarație if/elsif/else scrisă în interiorul unui proces.

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

entity mux21_4 is
          port (  a, b : in std_logic_vector(3 downto 0);
                  sel   : in std_logic;
                  c     : out std_logic_vector(3 downto 0) );
end mux21_4_1;

architecture mux21_4_1_arch of mux21_4_1 is
begin
process(a, b, sel)
begin
        if (sel = '0') then c <= a;
        else                c <= b;
        end if;
end process;
```

```
end mux21_4_1_arch;
```

Figură 2 - Proces și declarație secvențială if/elsif/else pentru comportamenul multiplexorului 2 la 1 pe 4 biți.

Pentru ambele module VHDL sintetizatorul va produce același circuit. Acesta este evidențiat în figura 3.

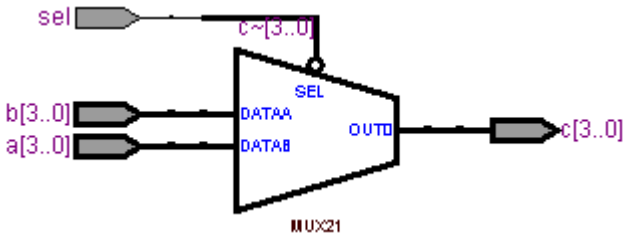


Figura 3 – Circuit sintetizat utilizând modulul din figura 1 sau figura 2.

Cele două reguli evidențiate mai sus trebuie respectate cu strictețe atunci când se dorește crearea de circuite logice combinaționale cu ajutorul proceselor și a declarațiilor secvențiale. Dacă aceste reguli nu se respectă modulul VHDL creat nu va descrie comportamentul unui circuit logic combinațional ci va descrie comportamentul unui circuit logic secvențial asincron.

De obicei sintetizatorul observă greșelile proiectantului și îl avertizează că circuitul sintetizat nu este unul combinațional ci unul secvențial asincron. În figurile 4 și 5 sunt evidențiate astfel de avertismente.

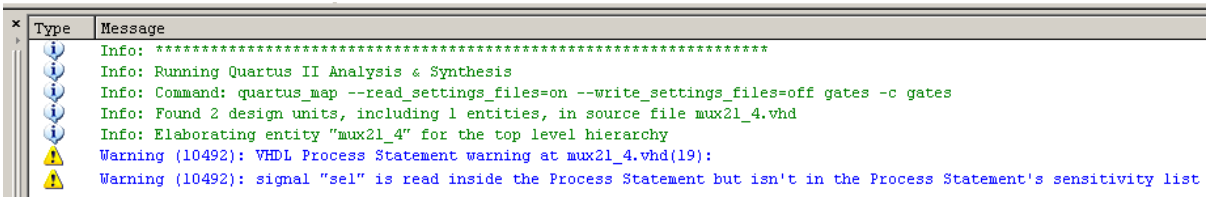


Figura 4 – Semnalul sel nu a fost trecut în lista de senzitivitate.

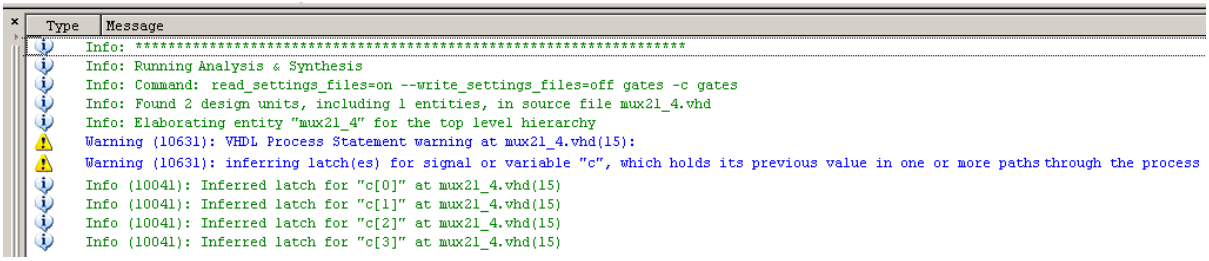


Figura 5 – Declarația secvențială nu cuprinde toate combinațiile semnalelor de intrare.

4.1.1 Circuite logice combinaționale – declarația case

4.1.1.1 Multiplexoare

În figura 6 este evidențiat un multiplexor 4 la 1 pe 4 biți descris cu ajutorul unui proces și declarației secvențiale case. În figura 7 este prezentat circuitul sintetizat.

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

entity mux41 is
        port ( a, b, c, d : in std_logic_vector(3 downto 0);
               sel        : in std_logic_vector(1 downto 0);
               e          : out std_logic_vector(3 downto 0) );
end mux41;

architecture mux41_arch of mux41 is
begin

process(a, b, c, d, sel)
begin
      case sel is
            when ”00”       => e <= a;
            when ”01”       => e <= b;
            when ”10”       => e <= c;
            when OTHERS => e <= d;
      end case;
end process;

end mux41_arch;
```

Figura 6 – Modul VHDL pentru un multiplexor 4 la 1 pe 4 biți.

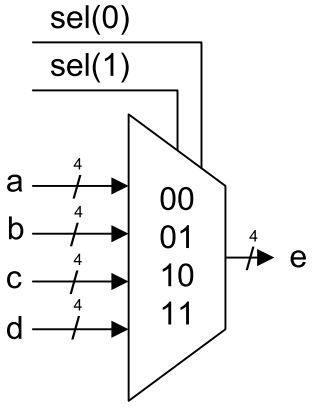


Figura 7 – Circuit sintetizat utilizând modulul VHDL din figura 6.

4.1.1.2 Demultiplexoare

În figura 8 este evidențiat un demultiplexor 1 la 4 pe 4 biți descris cu ajutorul unui proces și declarației secvențiale case. În figura 9 este prezentat circuitul sintetizat.

```
library ieee; use ieee.std_logic_1164.all;
entity demux14 is
        port ( a : in std_logic_vector(3 downto 0); sel : in std_logic_vector(1 downto 0);
                b, c, d, e : out std_logic_vector(3 downto 0) );
end demux14;
architecture demux14_arch of demux14 is
begin
process(a, sel)
begin
      case sel is
            when "00"         => b <= a; c <= "0000"; d <= "0000"; e <= "0000";
            when "01"         => c <= a;  d <= "0000"; e <= "0000"; b <= "0000";
            when "10"         => d <= a;  e <= "0000"; b <= "0000"; c <= "0000";
            when OTHERS => e <= a;  b <= "0000"; c <= "0000"; d <= "0000";
      end case;
end process;
end demux14_arch;
```

Figura 8 – Modul VHDL pentru un demultiplexor 1 la 4 pe 4 biți.

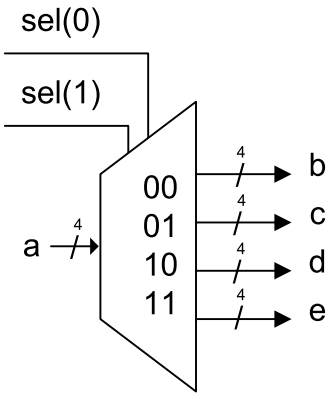


Figura 9 – Circuit sintetizat utilizând modulul VHDL din figura 8.

4.1.1.3 Codificatoare

4.1.1.3.1 Codificator binar

În figura 10 este prezentat tabelul de adevăr pentru un codificator binar 4 la 2. În figura 11 este evidențiat comportamentul codificatorului binar printr-un modul VHDL descris cu ajutorul unui proces și declarației secvențiale case. În figura 12 este prezentat circuitul sintetizat.

$a(3)$	$a(2)$	$a(1)$	$a(0)$	$b(1)$	$b(0)$
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Figura 10 – Tabel de adevăr pentru un codificator binar 4 la 2.

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

entity bin_cod is
        port ( a  : in std_logic_vector(3 downto 0);  b : out std_logic_vector(1 downto 0) );
end bin_cod;

architecture bin_cod_arch of bin_cod is
begin

process( a )
begin

      case a is
            when ”0001”     => b <= ”00”;
            when ”0010”     => b <= ”01”;
            when ”0100”     => b <= ”10”;
            when ”1000”     => b <= ”11”;
            when OTHERS => b <= ”--”;
      end case;

end process;

end bin_cod_arch;
```

Figura 11 – Modul VHDL ce descrie comportamentul codificatorului binar 4 la 2.

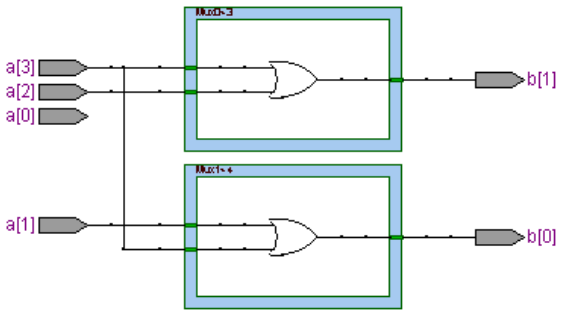


Figura 12 – Circuit sintetizat folosind modulul VHDL din figura 11.

4.1.1.3.2 Codificator cu prioritate

În figura 13 este prezentat tabelul de adevăr pentru un codificator cu prioritate 4 la 2. În figura 14 este evidențiat comportamentul codificatorului cu prioritate printr-un modul VHDL descris cu ajutorul unui proces și declarației secvențiale case. În figura 15 este prezentat circuitul sintetizat.

$a(3)$	$a(2)$	$a(1)$	$a(0)$	$b(1)$	$b(0)$
0	0	0	1	0	0
0	0	1	-	0	1
0	1	-	-	1	0
1	-	-	-	1	1

Figura 13 – Tabel de adevăr pentru un codificator cu prioritate 4 la 2.

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

entity pri_cod is
        port ( a  : in std_logic_vector(3 downto 0);
                b : out std_logic_vector(1 downto 0) );
end pri_cod;

architecture pri_cod_arch of pri_cod is
signal a_convert : integer range 0 to 15;
begin
a_convert <= conv_integer(a); -- conv_integer transformă valori binare în valori discrete
process( a_convert )
begin
      case a_convert is
            when 8 to 15      => b <= ”11”; -- 8 to 15 este o mulțime discretă
            when 4 to 7       => b <= ”10”;
```

```
            when 2 to 3        => b <= ”01”;
            when 1             => b <= ”00”;
            when OTHERS  => b <= ”--”;
        end case;
end process;

end pri_cod_arch;
```

Figura 14 – Modul VHDL ce evidențiază comportamentul unui codificator cu prioritate 4 la 2.

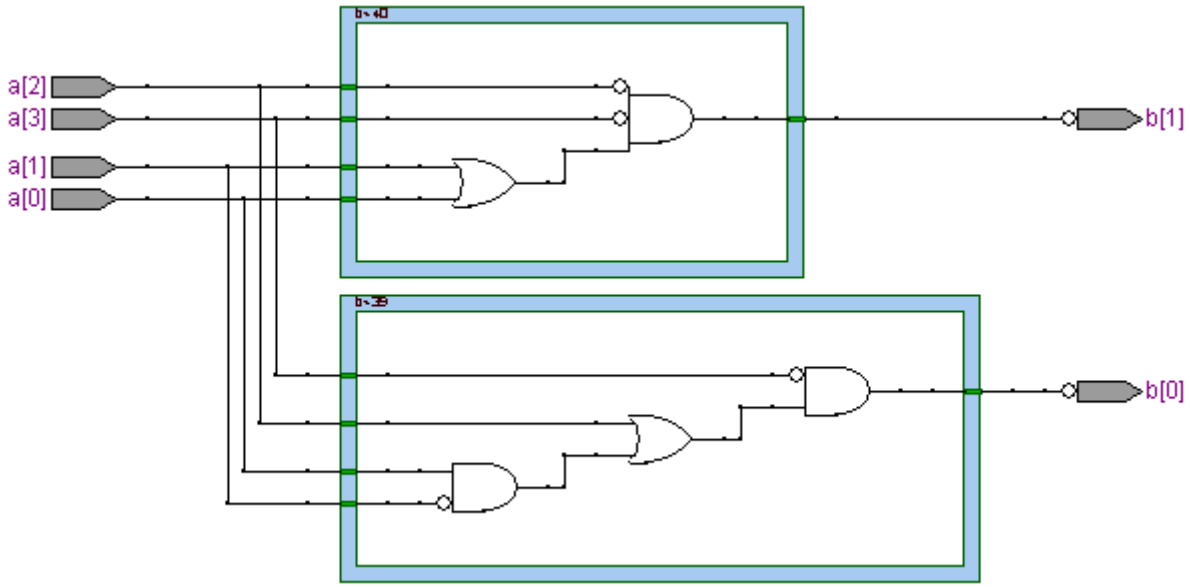


Figura 15 – Circuit sintetizat cu ajutorul modulului VHDL din figura 14.

4.1.1.4 Decodificatoare

În figura 16 este prezentat tabelul de adevăr pentru un decodificator 2 la 4. În figura 17 este evidențiat comportamentul decodificatorului printr-un modul VHDL descris cu ajutorul unui proces și declarației secvențiale case. În figura 18 este prezentat circuitul sintetizat.

$a(3)$	$a(2)$	$b(3)$	$b(2)$	$b(1)$	$b(0)$
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Figura 16 – Tabelul de adevăr al unui decodificator 2 la 4.

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

entity decod is
        port ( a  : in std_logic_vector(1 downto 0);
               b : out std_logic_vector(3 downto 0) );
end decod;

architecture decod_arch of decod is
begin
process( a )
begin
      case a is
            when ”00”       => b <= ”0001”;
            when ”01”       => b <= ”0010”;
            when ”10”       => b <= ”0100”;
            when ”11”       => b <= ”1000”;
            when OTHERS => b <= ”----”;
      end case;
end process;
end decod_arch;
```

Figura 17 – Modul VHDL ce descrie comportamenul unui decodificator 2 la 4.

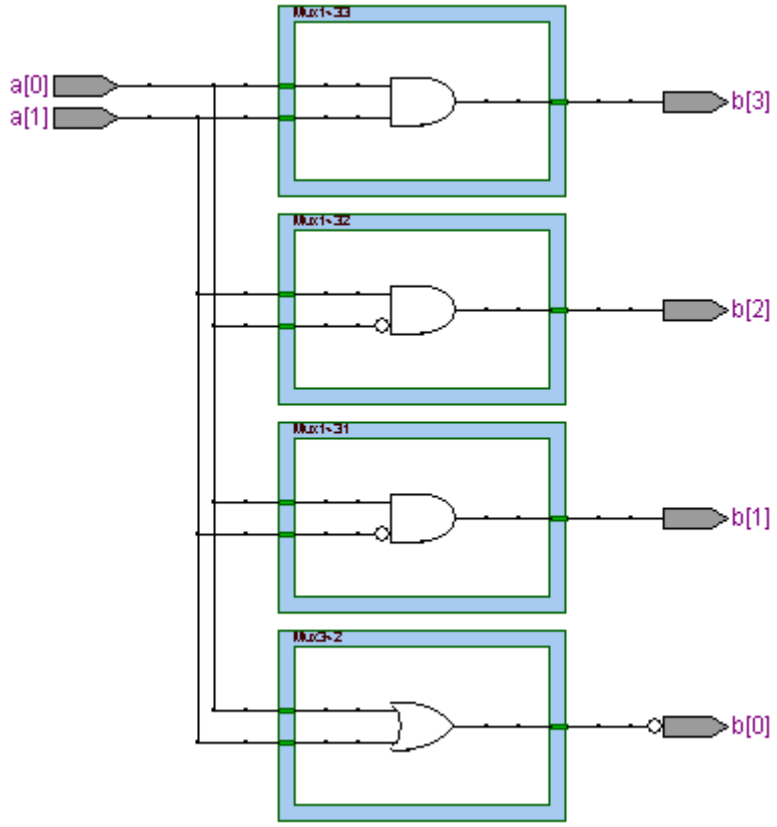


Figura 18 – Circuit sintetizat cu ajutoul modului VHDL din figura 17.

4.1.1.4.1 Decodificator cu prioritate

În figura 19 este prezentat tabelul de adevăr pentru un decodificator cu prioritate 4 la 4. În figura 20 este evidențiat comportamentul decodificatorului cu prioritate printr-un modul VHDL descris cu ajutorul unui proces și declarației secvențiale case. În figura 21 este prezentat circuitul sintetizat.

$a(3)$	$a(2)$	$a(1)$	$a(0)$	$b(3)$	$b(2)$	$b(1)$	$b(0)$
0	0	0	1	0	0	0	1
0	0	1	-	0	0	1	0
0	1	-	-	0	1	0	0
1	-	-	-	1	0	0	0

Figura 19 – Tabelul de adevăr pentru un decodificator cu prioritate 4 la 4.

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

entity pri_decod is
        port ( a  : in std_logic_vector(3 downto 0);
               b : out std_logic_vector(3 downto 0) );
end pri_decod;

architecture pri_decod_arch of pri_decod is

signal a_convert : integer range 0 to 15;

begin

a_convert <= conv_integer(a); -- conv_integer transformă valori binare în valori discrete

process( a_convert )
begin
      case a_convert is
            when 8 to 15      => b <= ”1000”; -- 8 to 15 este o mulțime discretă
            when 4 to 7       => b <= ”0100”;
            when 2 to 3       => b <= ”0010”;
            when 1            => b <= ”0001”;
            when OTHERS  => b <= ”----”;
      end case;
end pri_decod_arch;
```

Figura 20 – Modul VHDL ce descrie comportamentul unui decodificator cu prioritate 4 la 4.

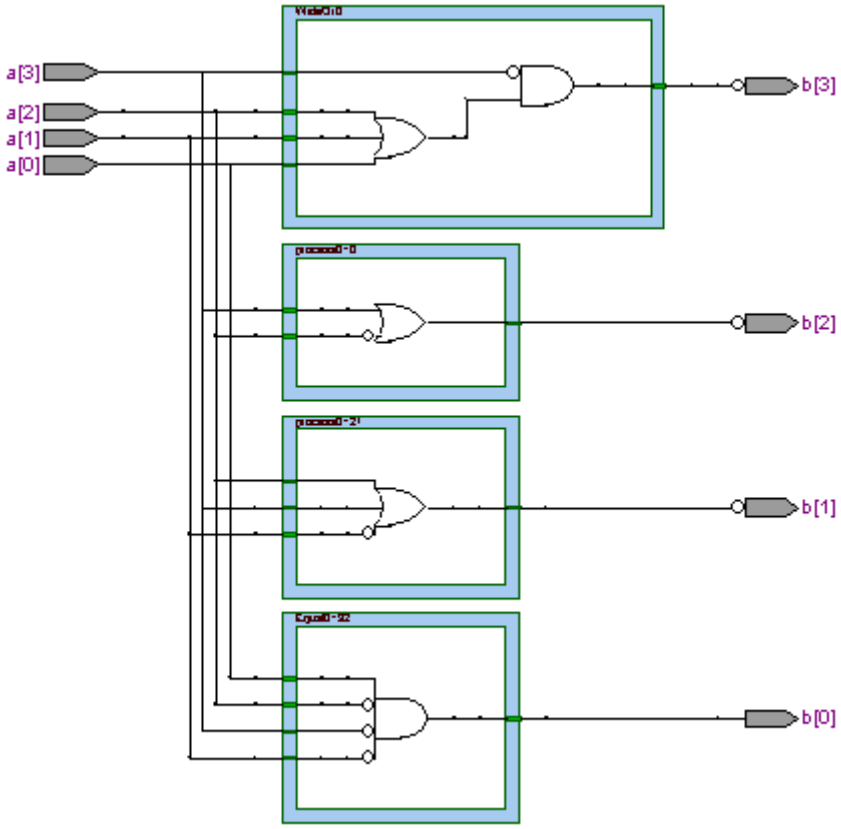


Figura 21 – Circuit sintetizat cu ajutorul modulului VHDL din figura 20.

4.1.1.5 Convertoare

În figura 22 este prezentat tabelul de adevăr pentru un convertor BCD-afișor cu șapte segmente. În figura 23 este evidențiat comportamentul convertorului printr-un modul VHDL descris cu ajutorul unui proces și declarației secvențiale case. În figura 24 este prezentat circuitul sintetizat.

$s(3)$	$s(2)$	$s(1)$	$s(0)$	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	0	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1

1	0	0	1	1	1	0	1	0	1	1
1	0	1	0	-	-	-	-	-	-	-
1	0	1	1	-	-	-	-	-	-	-
1	1	0	0	-	-	-	-	-	-	-
1	1	0	1	-	-	-	-	-	-	-
1	1	1	0	-	-	-	-	-	-	-
1	1	1	1	-	-	-	-	-	-	-

Figura 22 – Tabel de adevăr pentru un convertor BCD – afișor cu șapte segmente.

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

entity BCD7seg is
        port (
                s            : in std_logic_vector(3 downto 0);
                data_out : out std_logic_vector(0 to 6)
              );
end BCD7seg;

architecture BCD7seg_arch of BCD7seg is

begin

process( s )
begin
      case s is
            when ”0000”     => data_out <= ”1111110”; -- data_out = abcdefg
            when ”0001”     => data_out <= ”0110000”;
            when ”0010”     => data_out <= ”1101101”;
            when ”0011”     => data_out <= ”1111001”;
            when ”0100”     => data_out <= ”0110011”;
            when ”0101”     => data_out <= ”1011011”;
            when ”0110”     => data_out <= ”0011111”;
            when ”0111”     => data_out <= ”1110000”;
            when ”1000”     => data_out <=”1111111”;
            when ”1001”     => data_out <=”1110011”;
            when OTHERS  => data_out <= ”-------”;
      end case;
end process;

end BCD7seg_arch;
```

Figura 23 – Modul VHDL ce descrie comportamentul convertorului BCD – afișor cu șapte segmente.

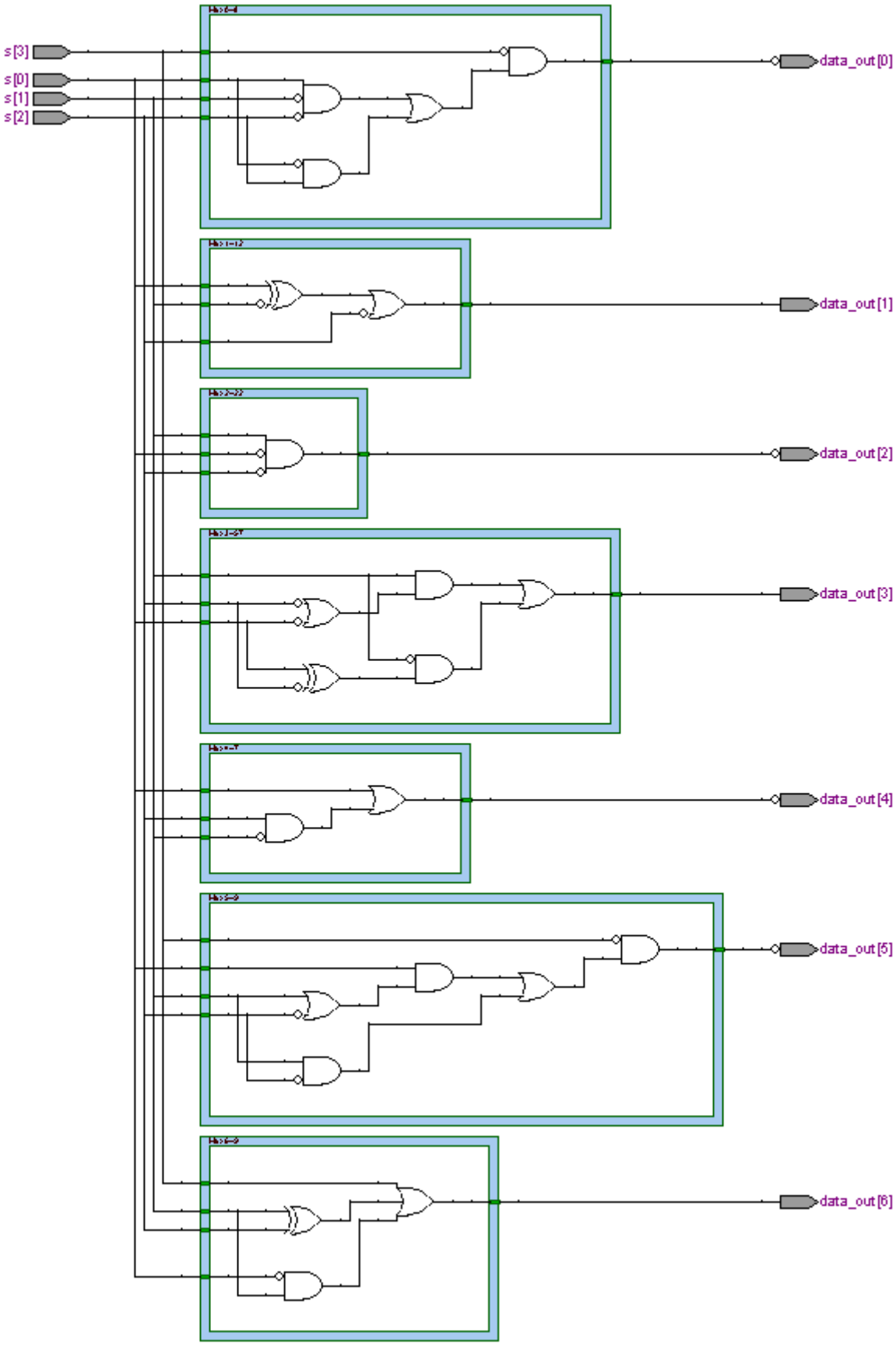


Figura 24 - Circuit sintetizat cu ajutorul modulului VHDL din figura 23.

4.1.2 Circuite logice combinaționale – declarația if/elsif/else

4.1.2.1 Multiplexoare

În figura 25 este evidențiat un multiplexor 4 la 1 pe 4 biți descris cu ajutorul unui proces și declarației secvențiale if/elsif/else.

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

entity mux41 is
        port ( a, b, c, d : in std_logic_vector(3 downto 0);
               sel        : in std_logic_vector(1 downto 0);
               e          : out std_logic_vector(3 downto 0) );
end mux41;

architecture mux41_arch of mux41 is
begin

process(a, b, c, d, sel)
begin
      if (sel = ”00”) then
            e <= a;
      elsif (sel = ”01”) then
            e <= b;
      elsif (sel = ”10”) then
            e <= c;
      else
            e <= a;
     end if;
end process;
end mux41_arch;
```

Figura 25 – Modul VHDL pentru un multiplexor 4 la 1 pe 4 biți.

4.1.2.2 Demultiplexoare

În figura 26 este evidențiat un demultiplexor 1 la 4 pe 4 biți descris cu ajutorul unui proces și declarației secvențiale if/elsif/else.

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

```
entity demux14 is
        port (
                a            : in std_logic_vector(3 downto 0);
                sel          : in std_logic_vector(1 downto 0);
                b, c, d, e   : out std_logic_vector(3 downto 0)
              );
end demux14;

architecture demux14_arch of demux14 is
begin
process(a, sel)
begin
      if (sel =  "00") then
            b <= a; c <= "0000"; d <= "0000"; e <= "0000";
      elsif (sel = "01") then
            c <= a;  d <= "0000"; e <= "0000"; b <= "0000";
      elsif (sel = "10") then
            d <= a;  e <= "0000"; b <= "0000"; c <= "0000";
      else
            e <= a;  b <= "0000"; c <= "0000"; d <= "0000";
      end if;
end process;
end demux14_arch;
```

Figura 26 – Modul VHDL pentru un demultiplexor 1 la 4 pe 4 biți.

4.1.2.3 Codificatoare

4.1.2.3.1 Codificator binar

În figura 27 este evidențiat comportamentul unui codificator binar 4 la 2 printr-un modul VHDL descris cu ajutorul unui proces şi declarației secvențiale if/elsif/else.

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

```
entity bin_cod is
        port ( a  : in std_logic_vector(3 downto 0);
               b : out std_logic_vector(1 downto 0) );
end bin_cod;

architecture bin_cod_arch of bin_cod is
begin

process( a )
begin
      if (a = ”0001”) then
            b <= ”00”;
      elsif (a = ”0010”) then
            b <= ”01”;
      elsif (a = ”0100”) then
            b <= ”10”;
      elsif (a = ”1000”) then
            b <= ”11”;
      else
            b <= ”--”;
      end if;
end process;
end bin_cod_arch;
```

Figura 27 – Modul VHDL pentru un codificator binar 4 la 2.

4.1.2.3.2 Codificator cu prioritate

În figura 28 este evidențiat comportamentul unui codificator cu prioritate 4 la 2 printr-un modul VHDL descris cu ajutorul unui proces și declarației secvențiale if/elsif/else.

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

entity pri_cod is
        port ( a  : in std_logic_vector(3 downto 0);
               b : out std_logic_vector(1 downto 0) );
end pri_cod;

architecture pri_cod_arch of pri_cod is
begin
process( a )
begin
      if (a(3) = ’1’) then
            b <= ”11”;
```

```
        elsif (a(2) = ’1’) then
                b <= ”10”;
        elsif (a(1) = ’1’) then
                b <= ”01”;
        elsif (a(0) = ’1’) then
                b <= ”00”;
        else
                b <= ”--”;
        end if;
end process;
end pri_cod_arch;
```

Figura 28 – Modul VHDL pentru un codificator cu prioritate 4 la 2.

4.1.2.4 Decodificatoare

În figura 29 este evidențiat comportamentul unui decodificator 4 la 2 printr-un modul VHDL descris cu ajutorul unui proces și declarației secvențiale if/elsif/else.

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

entity decod is
          port ( a  : in std_logic_vector(1 downto 0);
                   b : out std_logic_vector(3 downto 0) );
end decod;

architecture decod_arch of decod is
begin
process( a )
begin
        if (a = ”00”) then
                b <= ”0001”;
        elsif (a = ”01”) then
                b <= ”0010”;
        elsif (a = ”10”) then
                b <= ”0100”;
        elsif (a = ”11”) then
                b <= ”1000”;
        else
                b <= ”----”;
        end if;
end process;
end decod_arch;
```

Figura 29 – Modul VHDL pentru un decodificator 2 la 4.

4.1.2.4.1 Decodificator cu prioritate

În figura 30 este evidențiat comportamentul unui decodificator cu prioritate 4 la 4 printr-un modul VHDL descris cu ajutorul unui proces și declarației secvențiale if/elsif/else.

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

entity pri_decod is
        port ( a  : in std_logic_vector(3 downto 0);
               b : out std_logic_vector(3 downto 0) );
end pri_decod;

architecture pri_decod_arch of pri_decod is
begin
process( a )
begin
      if (a(3) = '1') then
            b <= "1000";
      elsif(a(2) = '1') then
            b <= "0100";
      elsif (a(1) = '1') then
            b <= "0010";
      elsif (a(0) = '1') then
            b <= "0001";
      else
            b <= "----";
      end if;
end pri_decod_arch;
```

Figura 30 – Modul VHDL pentru un decodificator cu prioritate 4 la 4.

4.1.2.5 Convertoare

În figura 31 este evidențiat comportamentul unui convertor BCD-afișor cu șapte segmente printr-un modul VHDL descris cu ajutorul unui proces și declarației secvențiale if/elsif/else.

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

entity BCD7seg is
      port ( s        : in std_logic_vector(3 downto 0);
             data_out : out std_logic_vector(0 to 6) );
```

```
end BCD7seg;

architecture BCD7seg_arch of BCD7seg is
begin
process( s )
begin
        if (s = ”0000”) then
                data_out <= ”1111110”;
        elsif (s = ”0001”) then
                data_out <= ”0110000”;
        elsif (s = ”0010”) then
                data_out <= ”1101101”;
        elsif (s = ”0011”) then
                data_out <= ”1111001”;
        elsif (s = ”0100”) then
                data_out <= ”0110011”;
        elsif (s = ”0101”) then
                data_out <= ”1011011”;
        elsif (s = ”0110”) then
                data_out <= ”0011111”;
        elsif (s = ”0111”) then
                data_out <= ”1110000”;
        elsif (s = ”1000”) then
                data_out <=”1111111”;
        elsif (s = ”1001”) then
                data_out <=”1110011”;
        else
                data_out <= ”-------”;
        end if;
end process;
end BCD7seg_arch;
```

Figură 31 – Modul VHDL pentru un convertor BCD – afișor cu șapte segmente.

Capitolul 5

Sintetizarea circuitelor aritmetico-logice

Sintetizarea circuitelor aritmetico-logice

Circuitele aritmetico-logice reprezintă “mușchii” calculatoarelor. Aceste circuite se pot crea foarte ușor utilizând regula celor 3 E. În alcătuirea circuitelor atirmetico-logice intră circuitele logice prezentate în capitolele 2, 3 și 4. Aceste circuite sunt parametrizabile, se pot modifica foarte ușor pentru a acomoda intrări de diferite dimensiuni. Capitolul Module parametrizabile va oferi mai multe informații cu privire la crearea modulelor VHDL generale. Circuitele aritmetico-logice îndeplinesc operațiile aritmetice și logice de bază.

Operațiile aritmetice sunt: **adunare**, **scădere**, **înmulțire**, **împărțire** și **comparație**. Operațiile logice sunt: **NOT**, **AND**, **OR**, **NAND**, **NOR**, **XOR**, **XNOR** și **operația de deplasare**. Toate aceste operații sunt reunite într-o unitate aritmetico-logică, denumită și **ALU**. Un ALU poate procesa diferite tipuri de numere: numere naturale sau întregi, sub diferite reprezentări binare.

Anexa 3 – Reprezentarea numerelor va evidenția diferitele reprezentări binare ale numerelor. Trebuie precizat că toate microprocesoarele au în componență un ALU.

5.1 Operații aritmetice

5.1.1 Adunare

5.1.1.1 Sumator parțial

Sumatorul parțial este un circuit aritmetic ce va aduna două semnale pe un singur bit. În figura 1 este evidențiat tabelul de adevăr pentru acest circuit. Atunci când semnalul A are valoarea logică 1 și când semnalul B are aceiași valoarea, suma celor două semnale este egală cu 2. Valoarea 2 se reprezintă în formă binară pe doi biți ca 10, deci semnalul de ieșire S va avea valoarea logică 0, iar C_{out} va avea valoarea logică 1. C_{out} reprezintă bitul de transport sau depășirea. În cazul în care semnalul A va avea valoarea logică 0, iar semnalul B va avea valoarea logică 1, atunci semnalul de ieșire S va avea valoarea logică 1, iar depășirea va avea valoarea logică 0. În figura 2 este reprezentat modulul VHDL pentru sumatorul parțial. În figura 3 este prezentat circuitul sintetizat.

A	B	S	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figura 1 – Tabelul de adevăr pentru un sumator parţial.

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

entity halfadder is
        port (
                A, B     : in std_logic;
                S, Cout : out std_logic
               );
end halfadder;

architecture halfadder_arch of halfadder is
begin

process(A, B)
begin
      if (A = ’0’ and B = ’0’) then
              S     <=  ’0’;
              Cout <= ’0’;
      elsif (A = ’1’ and B = ’0’) then
              S      <= ’1’;
              Cout <= ’0’;
      elsif (A = ’0’ and B = ’1’) then
              S      <= ’1’;
              Cout <= ’0’;
      elsif (A = ’1’ and B = ’1’) then
              S      <= ’0’;
              Cout <= ’1’;
      else
              S      <= ’-’;
              Cout <= ’-’;
      end if;
end process;

end halfadder_arch;
```

Figura 2 – Modul VHDL pentru un sumator parţial.

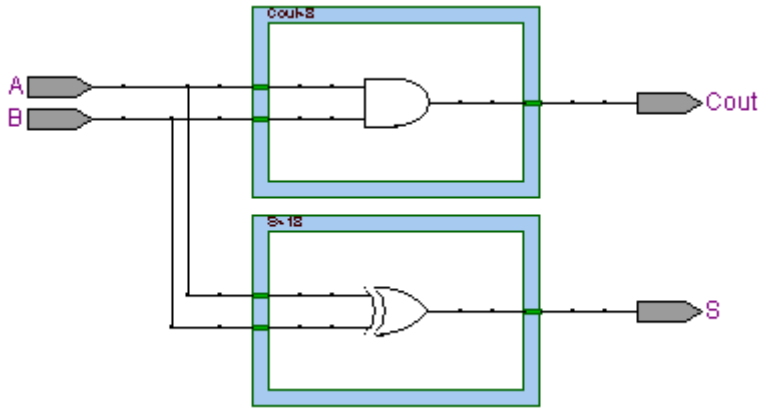


Figura 3 – Circuit sintetizat folosind modulul VHDL din figura 2.

Dacă se doreşte adunarea a două semnale pe mai mulţi biţi circuitul din figura 3 trebuie modificat.

În figura 4 este evidențiată printr-un exemplu adunarea a două numere naturale. Modul de calcul este următorul. Se va aduna prima coloană. Din adunarea numărelor 9 și 5 vom obține ca rezultat 4 și "'ținem minte" 1, adică depășirea va fi 1. În continuare vom aduna cea de a doua coloană la care se va adăuga depășirea anterioară. Adică vom aduna 1 cu 1 cu 1, rezultatul fiind 3, iar depășirea va fi 0.

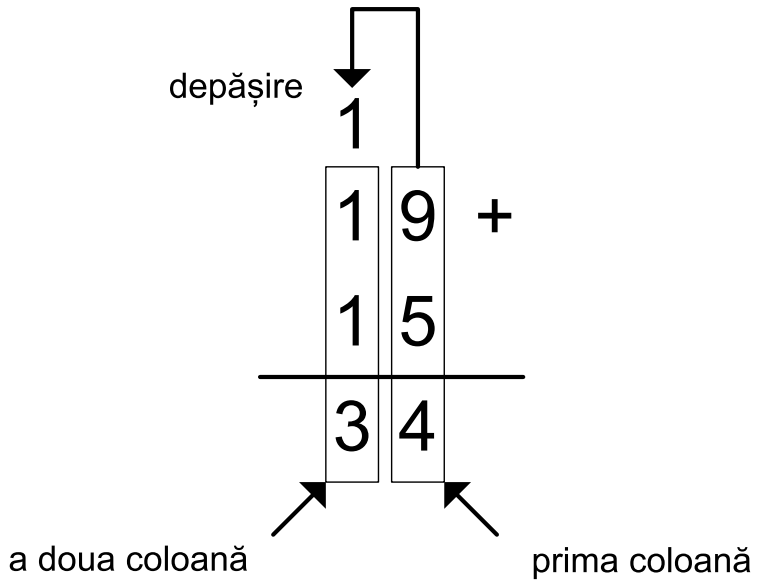


Figura 4 – Adunarea a două numere naturale.

În același mod se vor aduna două numere binare pe mai mulți biți. Din figura 5 se observă că depășirea primului sumator parțial nu poate fi introdusă în următorul modul, deoarece sumatorul parțial a fost proiectat cu doar două intrări. Pentru a remedia acest

neajuns sumatorul parțial va fi modificat și va avea o a treia intrare, care va accepta depășirea unui sumator pațial anterior. Acest circuit va purta numele de sumatorul complet.

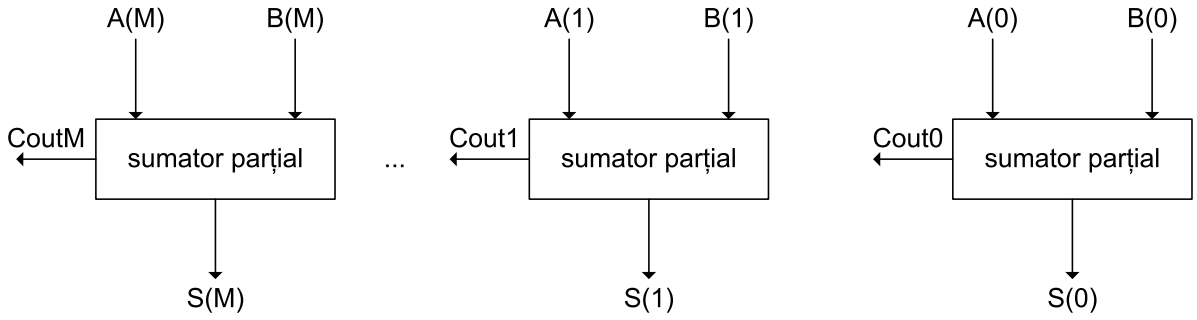


Figura 5 – Adunarea a două semnale pe n biți și reprezentarea neajunsurilor sumatorului parțial

5.1.1.2 Sumator complet

În figura 6 este prezentat tabelul de adevăr al sumatorului complet. După cum se observă acest circuit are 3 intrări, două intrări de date A și B și depășirea anterioră, C_{in}, respectiv 2 ieșeri, rezultatul S și depășirea actuală, C_{out}. În figura 7 este evidențiat modulul VHDL pentru un sumatot complet pentru intrări de date pe un bit. În figura 8 este reprezentat circuitul sintetizat. În figura 9 este reprezentat simbolul sumatorului complet.

C_{in}	A	B	S	C_{out}
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

Figura 6 – Tabel de adevăr pentru un sumatorul complet.

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

entity fulladder is
```

```
        port ( A, B, Cin  : in std_logic;
               S, Cout    : out std_logic);
end fulladder;

architecture fulladder_arch of fulladder is
begin

process(A, B, Cin)
begin
    if (A = '0' and B = '0' and Cin = '0') then
        S    <= '0';
        Cout <= '0';
    elsif (A = '1' and B = '0' and Cin = '0') then
        S    <= '1';
        Cout <= '0';
    elsif (A = '0' and B = '1' and Cin = '0') then
        S    <= '1';
        Cout <= '0';
    elsif (A = '1' and B = '1' and Cin = '0') then
        S    <= '0';
        Cout <= '1';
    elsif (A = '0' and B = '0' and Cin = '1') then
        S    <= '1';
        Cout <= '0';
    elsif (A = '1' and B = '0' and Cin = '1') then
        S    <= '0';
        Cout <= '1';
    elsif (A = '0' and B = '1' and Cin = '1') then
        S    <= '0';
        Cout <= '1';
    elsif (A = '1' and B = '1' and Cin = '1') then
        S    <= '1';
        Cout <= '1';
```

```
        else
                S       <= '-';
                Cout <= '-';
        end if;
end process;
end fulladder_arch;
```

Figura 7 – Modul VHDL pentru un sumator complet.

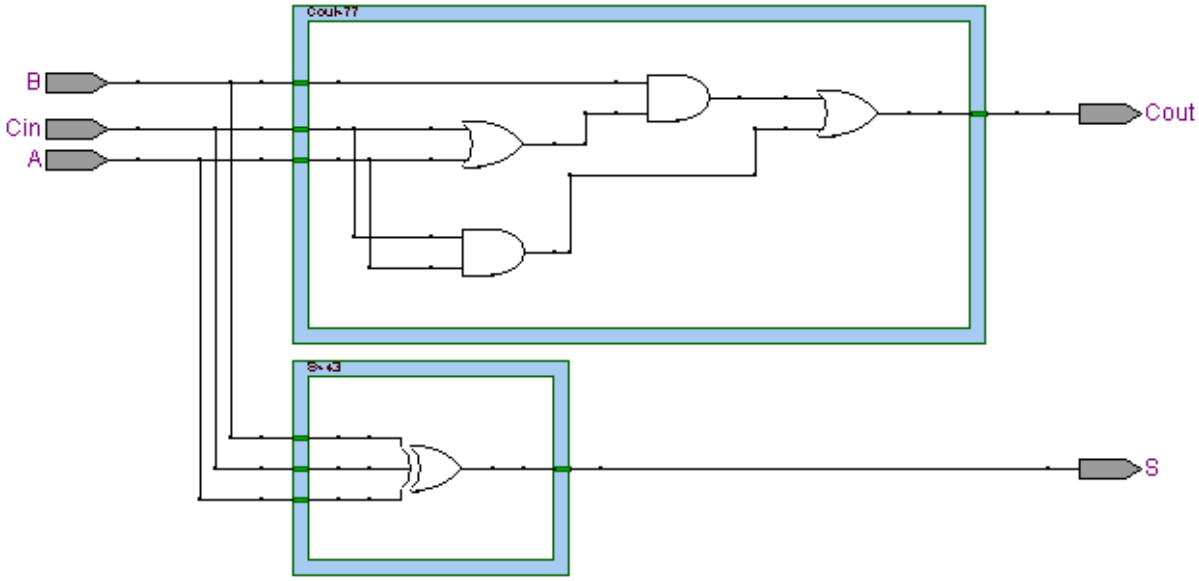


Figura 8 – Circuit sintetizat cu ajutorul modulului din figura 7.

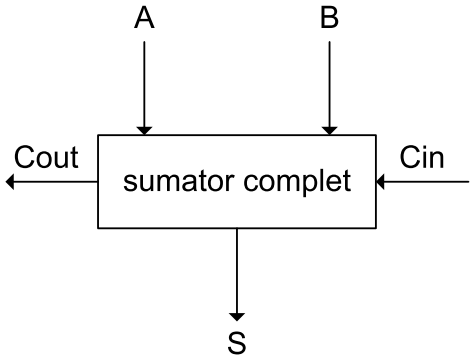


Figura 9 – Simbolul unui sumator complet.

Trebuie precizat că C_{out} îndeplinește o funcție logică ce poartă numele de "majoritatea decide". C_{out} ia valoarea logică 1 în cazul în care două sau trei (mai multe) intrări au valoarea logică 1. În cazul în care doar o intrare este activă la un moment dat C_{out} ia valoarea logică 0.

5.1.1.3 Sumator cu propagarea depășirii

Un sumator cu propagarea depășirii deține trei intrări, două intrări de date, A și B, pe mai mulți biți și o intrare pentru depășirea anterioară, C_{in}, respectiv două ieșiri, rezultatul adunării semnalelor A și B, notat cu S, și depășirea actuală, C_{out}. Semnalele A, B și S sunt alcătuite din M biți, iar C_{in} și C_{out} sunt semnale pe un singur bit.

Denumirea acestui tip de sumator relevă faptul că adunarea binară a două semnale este efectuată prin propagarea depășirii actuale. Depășirea actuală a primului modul, C_{out0}, rezultată din sumarea primilor biți, A(0) și B(0), și a depășirii anterioare, (C_{in}), va deveni depășirea anterioară a modulului secund, C_{in1}. Ea va fi folosită împreună cu biții secunzi, A(1), B(1), pentru calcularea rezultatului, S(1), și a depășirii actuale, C_{out1}, procesul se va repeta pentru toți biții semnalelor de la intrare. În figura 10 este prezentat simbolul unui sumator cu propagarea depășirii.

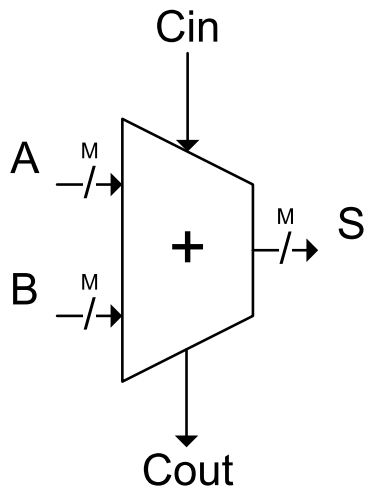


Figura 10 – Simbolul sumatorului cu propagarea depășirii.

În continuare se vor prezenta trei implementări pentru un sumator cu propagarea depășirii. Aceste circuite sunt prezentate în ordinea crescătoare a vitezei de procesare, adică în ordinea descrescătoare a timpului de propagare, t_{pd}.

5.1.1.3.1 Sumator cu propagare în cascadă

În figura 11 este evidențiată implementarea sumatorului cu propagare în cascadă. Pentru ca acest sumator să sumeze două semnale de intrare pe M biți, va fi nevoie să înseriem M sumatoare complete. În figura 12 este reprezentat un modul VHDL pentru un sumator cu propagare în cascadă pe 4 biți. În figura 13 este evidențiat circuitul sintetizat.

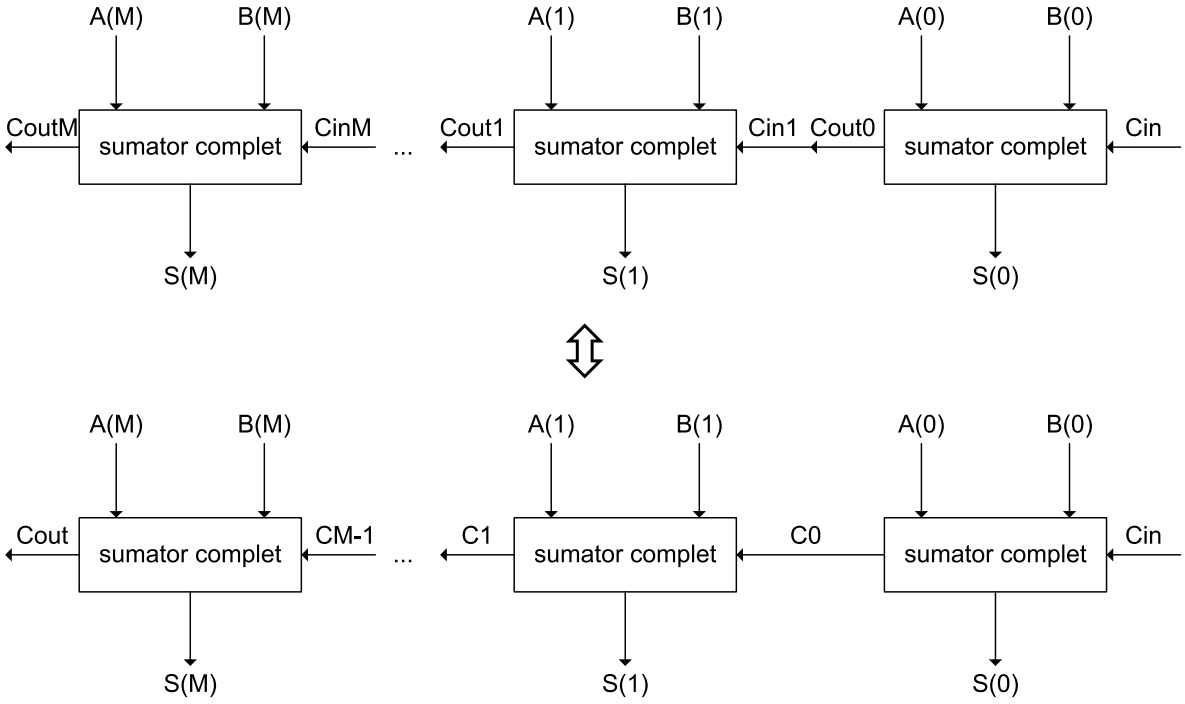


Figura 11 – Implementarea sumatorului cu propagare în cascadă.

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

entity spc is
        port( Cin    : std_logic;
              A, B  : in std_logic_vector(3 downto 0);
              S      : out std_logic_vector(3 downto 0);
              Cout : out std_logic );
end spc;

architecture spc_arch of spc is
component fulladder
        port ( A, B, Cin  : in std_logic;
               S, Cout      : out std_logic);
end component;
signal C : std_logic_vector(0 to 2);
begin
```

```
stage0 : fulladder port map (A(0), B(0), Cin,  S(0), C(0));
stage1 : fulladder port map (A(1), B(1), C(0), S(1), C(1));
stage2 : fulladder port map (A(2), B(2), C(1), S(2), C(2));
stage3 : fulladder port map (A(3), B(3), C(2), S(3), Cout);
end spc_arch;
```

Figura 12 - Modul VHDL pentru un sumator cu propagare în cascadă.

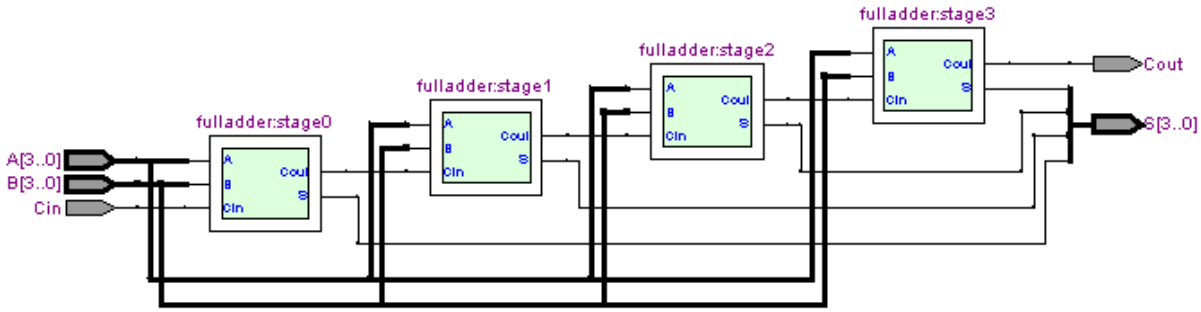


Figura 13 – Circuit sintetizat cu ajutorul modulului VHDL din figura 12.

Trebuie precizat că viteza de procesare a sumatorului cu propagare în cascadă scade o dată cu extinderea sumatorului. Bitul final al rezultatului se calculează adunând biții finali de intrare și depășirea anterioară provenită din penultimul bloc. Biții finali de intrare sunt accesibili imediat, însă pentru ca depășirea anterioară provenită din penultimul bloc să fie accesibilă ea trebuie calculată. Acestă depășire va fi calculată prin obținerea și utilizarea depășirilor anterioare, calcul ce durează destul de mult dacă sumatorul este extins pe foarte mulți biți. Acest neajuns este remediat prin implementarea sumatorului cu propagare anticipată, sumator ce procesează date cu o viteză mai mare ca sumatorul cu propagare în cascadă.

5.1.1.3.2 Sumator cu propagare anticipată

Sumatorul cu propagare anticipată prezintă două îmbunătățiri față de sumatorul cu propagare în cascadă. Sumatorul este împărțit în **blocuri** iar fiecare bloc are capacitatea de a calcula foarte repede depășirea actuală imediat ce se cunoaște depășirea anterioară. Avantajul împărțirii în blocuri a sumatorului este acela că sumatorul va fi ușor de implementat. Implementarea acestui sumator evidențiază utilizarea regulii celor 3 E.

Acest sumator are la bază tabelul din figura 6, ce evidențiază funcția logică a sumatorului complet. Sumatorul cu propagare anticipată utilizează două semnale numite **semnal de generare** și **semnal de propagare**. Aceste semnale sunt folosite pentru a calcula depășirea actuală determinată de fiecare bit al intrărilor sau de fiecare bloc în parte Semnalul de generare se notează cu G, iar semnalul de propagare se notează cu P.

Coloana k a intrărilor A și B va genera o depășire actuală dacă acestă depășire nu va depinde de o depășire anterioară. Acest lucru este evidențiat în figura 14. După cum se observă coloana k va genera o depășire actuală independentă de o depășire anterioară în cazul în care $A(k)$ și $B(k)$ vor avea valoarea logică 1, deci semnalul de generare pentru coloana k a intrărilor are funcția logică $G(k) = A(k)B(k)$.

C_{in}	$A(k)$	$B(k)$	C_{out}
0	1	1	1
1	1	1	1

Figura 14 – Tabelul de adevăr pentru semnalul de generare al coloanei *k* a intrărilor.

Coloana k a intrărilor A și B va propaga o depășire actuală ori de câte ori există o depășire anterioară. Acest lucru este evidențiat în figura 15. După cum se observă coloana k va propaga o depășire actuală în cazul în care bit-ul k al intrării A sau bit-ul k al intrării B va avea valoarea logică 1, deci semnalul de propagare pentru coloana k a intrărilor are funcția logică $P(k) = A(k) + B(k)$.

C_{in}	$A(k)$	$B(k)$	C_{out}
1	0	1	1
1	1	0	1
1	1	1	1

Figura 15 – Tabelul de adevăr pentru semnalul de generare al coloanei *k* a intrărilor.

În acest moment se poate calcula depășirea actuală dacă se cunoaște coloana k a intrărilor A și B și depășirea anterioară, C_{in}. În figura 16 este evidențiată funcția logică a depășirii actuale pentru coloana k a intrărilor.

$$C_{out} = G(k) + P(k)C_{in}$$

Figura 16 – Funcția logică a depășirii actuale pentru coloana *k* a intrărilor.

În figura 17 este rescrisă funcția logică din figura 16 sub o formă convenabilă. C_{out} este înlocuit de C_l iar C_{in} este înlocuit de C_{l-1}. C_l este depășirea actuală, respectiv C_{l-1} este depășirea anterioară.

$$C_l = G(k) + P(k)C_{l-1}$$

Figură 17 – Funcția logică a depășirii actuale pentru coloana k a intrărilor.

Semnalele de generare și propagare se vor calcula și pentru fiecare bloc. Astfel că un bloc va genera o depășire actuală dacă acestă depășire va fi independentă de o depășire provenită de la un bloc anterior. Un bloc va propaga o depășire actuală ori de câte ori există o depășire provenită de la un bloc anterior. Pentru un bloc ce va cu prinde coloane de la i la j, semnalul de generare se va nota cu $G(j \leftarrow i)$, iar semnalul de propagare se va nota cu $P(j \leftarrow i)$.

În acestă parte va fi evidențiată implementarea unui sumator cu propagare anticipată pe 4 biți. Acest sumator pe 4 biți va fi împărțit în două blocuri pe 2 biți. Semnalul de generare,$G(1 \leftarrow 0)$, va genera o depășire actuală în cazul în care coloana 1 va genera o depășire actuală sau în cazul în care coloana 1 va propaga o depășire actuală și coloana 0 va genera o depășire actuală. Funcția logică a semnalului de generare, $G(1 \leftarrow 0)$, este evidențiată în figura 18.

$$G(1 \leftarrow 0) = G(1) + P(1)G(0)$$

Figura 18 – Semnalul de generare, $G(1 \leftarrow 0)$.

Semnalul de propagare, $P(1 \leftarrow 0)$, va propaga o depășire actuală ori de câte ori coloanele semnalelor de intrare vor propaga o depășire actuală. Funcția logică a semnalului de propagare, $P(1 \leftarrow 0)$, este evidențiată în figura 19.

$$P(1 \leftarrow 0) = P(1)P(0)$$

Figura 19 – Semnalul de propagare, $P(1 \leftarrow 0)$.

Cunoscând semnalul de generare, $G(i \leftarrow j)$, semnalul de propagare, $P(i \leftarrow j)$ și depășirea provenită de la un bloc anterior, $C(j)$, se va putea calcula depășirea actuală a blocului, $C(i)$. Funcția logică a depășirii actuale pentru un bloc oarecare este evidențiată în figura 20.

$$C(i) = G(i \leftarrow j) + P(i \leftarrow j)C(j)$$

Figura 20 – Funcția logică a depășirii actuale pentru un bloc oarecare.

În figura 21 este reprezentat schematic sumatorul cu propagare anticipată pe 4 biți. În figură se observă cele două blocuri și modul în care este alcătuit un bloc, adică dintr-un sumator complet pe 2 biți respectiv logica ce calculează depășirea actuală a blocului. În figura 22 este evidențiat un modul VHDL pentru sumatorul cu propagare anticipată pe 4 biți. În figura 23 este reprezentat circuitul sintetizat.

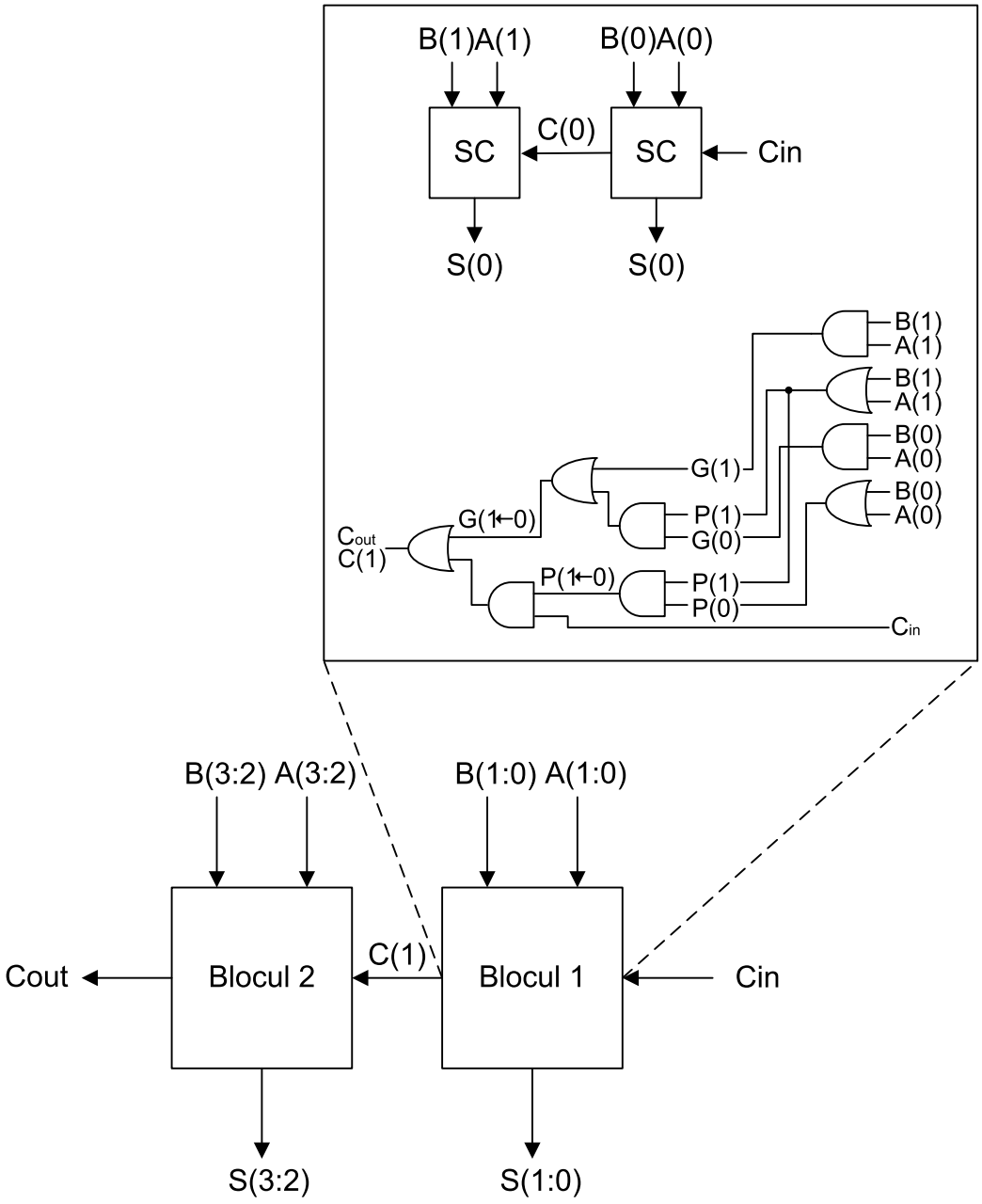


Figura 21 – Reprezentarea schematică a sumatorului cu propagare anticipată pe 4 biți.

spa_block.vhd

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

entity spa_block is
        port ( Cin    : in std_logic;
               A, B  : in std_logic_vector(1 downto 0);
               S      : out std_logic_vector(1 downto 0);
               Cout : out std_logic);
end spa_block;

architecture spa_block_arch of spa_block is

component fulladder
        port ( A, B, Cin  : in std_logic;
               S, Cout     : out std_logic);
end component;

signal C0 : std_logic;
signal G1, G0, P1, P0, G1_0, P1_0, C1 : std_logic;

begin
G0 <= A(0) and B(0);
G1 <= A(1) and B(1);

P0 <= A(0) or B(0);
P1 <= A(0) or B(0);

G1_0 <= G1 or (P1 and G0);
P1_0 <= P1 and P0;

C1 <= G1_0 or (P1_0 and Cin);
Cout <= C1;
stage0 : fulladder port map (A(0), B(0), Cin, S(0), C0);
```

```
stage1 : fulladder port map (A(1), B(1), C0,  S(1));
end spa_block_arch;
```

spa.vhd

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

entity spa is
        port ( Cin    : in std_logic;
               A, B  : in std_logic_vector(3 downto 0);
               S      : out std_logic_vector(3 downto 0);
               Cout : out std_logic);
end spa;

architecture spa_arch of spa is

component spa_block is
        port ( Cin   : in std_logic;
               A, B : in std_logic_vector(1 downto 0);
               S     : out std_logic_vector(1 downto 0);
               Cout : out std_logic);
end component;

signal C1 : std_logic;

begin
stage0 : spa_block port map (Cin, A(1 downto 0), B(1 downto 0), S(1 downto 0), C1);
stage1 : spa_block port map (C1,  A(3 downto 2),  B(3 downto 2), S(3 downto 2), Cout);
end spa_arch;
```

Figura 22 – Modul VHDL pentru un sumator cu propagare anticipată pe 4 biți.

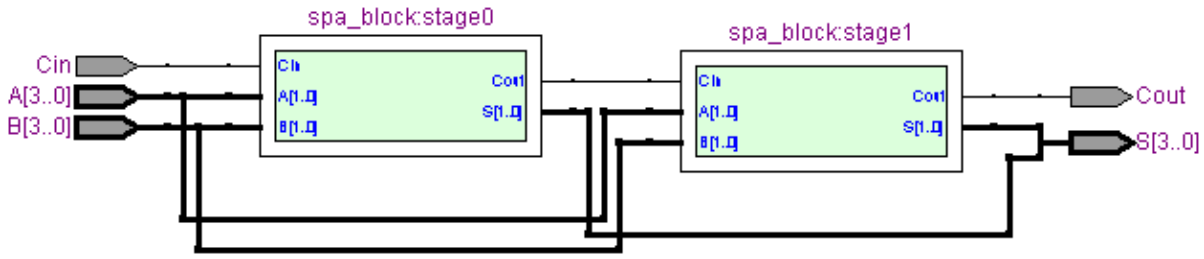


Figura 23 – Circuit sintetizat cu ajutorul modulului VHDL din figura 22.

Dacă se dorește implementarea unui sumator pentru semnale de intrare cu dimensiuni mai mici de 16 biți se poate folosi un sumator cu propagare în cascadă. Dacă se dorește implementarea unui sumator pentru semnale de intrare cu dimensiuni mai mari de 16 biți, soluția este sumatorul cu propagare anticipată.

Acest sumator este mai rapid ca sumatorul cu propagare în cascadă pentru semnale de intrare cu dimensiuni mai mari de 16 biți.

5.1.2 Scădere

Anexa 3 - Reprezentarea numerelor va evidenția diferitele tipuri de reprezentări ale numerelor binare. În practică se utilizează des numerele naturale binare și numerele întregi binare. Operația de scădere pentru aceste două tipuri de numere binare va fi implementată diferit.

5.1.2.1 Scăderea numerelor naturale binare

În cazul în care se dorește scăderea a două numere naturale binare A și B, rezultatul va fi $T = A - B$. Se cunoaște că operația de scădere derivă din operația de adunare. Astfel că rezultatul scăderii numerelor naturale binare A și B se mai poate scrie $T = A + (-B)$. Se știe că opusul unui număr B este $-B$. În logica binară, pentru a găsi opusul unui număr natural binar B, acel număr va trebui negat. Vom obține numărul natural binar$\bar{B}$, iar rezultatul scăderii se va rescrie astfel $T = A + \bar{B}$.

În figura 24 este evidențiat modulul VHDL ce implementează operația de scădere pentru numere naturale binare. Acest modul va folosi unul dintre cele două sumatoare cu propagarea depășirii. În acest moment, utilizând tehnica abstractizării, structura internă a sumatorului este neimportantă. Importantă este funcționarea sumatorului, adică relația intrare-ieșire. În figura 25 este reprezentat circuitul sintetizat.

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

entity unsig_sub is
        port ( Cin   : in std_logic; -- pentru acesta operație întotdeauna Cin <= '0';
               A, B : in std_logic_vector(3 downto 0);
               S     : out std_logic_vector(3 downto 0);
               Cout : out std_logic);
end unsig_sub;

architecture unsig_sub_arch of unsig_sub is

component spc
        port( Cin    : std_logic;
              A, B  : in std_logic_vector(3 downto 0);
              S      : out std_logic_vector(3 downto 0);
              Cout : out std_logic );
end component;

signal not_B : std_logic_vector(3 downto 0);
begin

not_B <= not B;
stage : spc port map (Cin, A, not_B, S, Cout);
end unsig_sub_arch;
```

Figura 24 – Modul VHDL ce implementează operația de scădere pentru numere naturale binare.

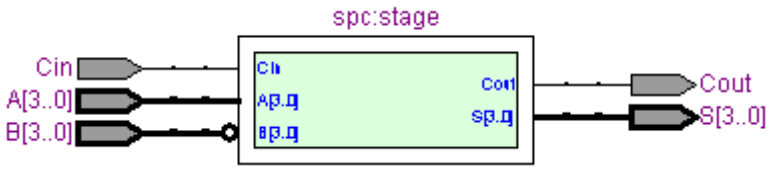


Figura 25 – Circuit sintetizat cu ajutorul modulului VHDL din figura 24.

5.1.2.2 Scăderea numerelor întregi binare

În cazul în care se dorește scăderea a două numere întregi binare A și B, rezultatul va fi $T = A - B$. Această relație se mai poate scrie $T = A + (-B)$. Se știe că opusul unui număr B este $-B$. În Anexa 3 sunt evidențiate trei reprezentări pentru numerele întregi binare. Reprezentarea în **complement față de 2** este singura reprezentare în uz în zilele noastre. În logica binară, pentru a găsi opusul unui număr întreg binar B reprezentat în complement față de 2, numărul va fi negat după care i se va aduna valoarea logică 1. În acest caz rezultatul scăderii se va rescrie astfel $T = A + \bar{B} + 1$.

În figura 26 este evidențiat modulul VHDL ce implementează operația de scădere pentru numere întregi binare reprezentate în complement față de 2. Acest modul va folosi unul dintre cele două sumatoare cu propagarea depășirii. În figura 25 este reprezentat circuitul sintetizat.

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

entity sig_sub is
        port ( Cin    : in std_logic; -- pentru acesta operație întotdeauna Cin <= '1';
                A, B  : in std_logic_vector(3 downto 0);
                S     : out std_logic_vector(3 downto 0);
                Cout : out std_logic);
end sig_sub;

architecture sig_sub_arch of sig_sub is

component spc
        port( Cin    : std_logic;
                A, B  : in std_logic_vector(3 downto 0);
                S     : out std_logic_vector(3 downto 0);
                Cout : out std_logic );
end component;
```

```
signal not_B : std_logic_vector(3 downto 0);
begin

not_B <= not B;
stage : spc port map (Cin, A, not_B, S, Cout);
end sig_sub_arch;
```

Figura 26 – Modul VHDL ce implementează operația de scădere pentru numere întregi binare reprezentate în complement față de 2.

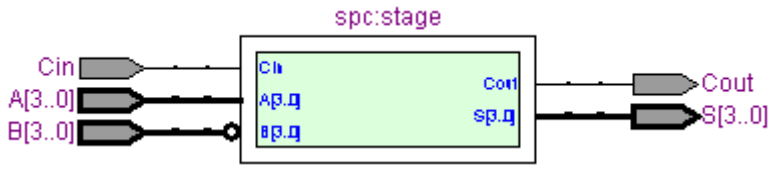


Figura 27 – Circuit sintetizat cu ajutorul modulului VHDL din figura 25.

5.1.3 Înmulțire

Operația de înmulțire se poate implementa utilizând operatorul artimetic ∗. În figura 28 este evidențiat modulul VHDL pentru circuitul de înmulțire pe 4 biți. Dacă semnalele de intrare sunt pe 4 biți, rezultatul operației de înmulțire va fi pe 8 biți. Operația de înmulțire este foarte costisitoare pentru a fi implementată. În figura 29 este evidențiat circuitul sintetizat.

```
library ieee; use ieee.std_logic_1164.all; use ieee.std_logic_unsigned.all;
entity mult is
        port (A, B : in std_logic_vector(3 downto 0);  P : out std_logic_vector(7 downto 0) );
end mult;
architecture mult_arch of mult is
begin
P <= A*B;
end mult_arch;
```

Figură 28 – Modul VHDL ce implementează operația de înmulțire.

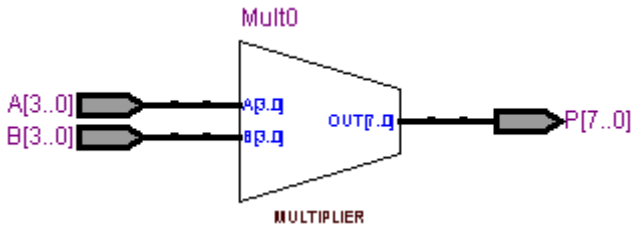


Figura 29 – Circuit sintetizat cu ajutorul modulului VHDL din figura 28.

5.1.4 Împărțire

Operația de împărțire se poate implementa utilizând operatorul artimetic /. În figura 30 este evidențiat modulul VHDL pentru circuitul de împărțire pe 4 biți. Dacă semnalele de intrare sunt pe 4 biți, rezultatul operației de împărțire va fi pe 4 biți. Operația de împărțire este foarte costisitoare pentru a fi implementată.

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

entity div is
        port ( A, B : in std_logic_vector(3 downto 0);
                Q      : out std_logic_vector(3 downto 0) );
end div;

architecture div_arch of div is
begin
Q <= A/B;
end div_arch;
```

Figura 30 – Modul VHDL ce implementează operația de împărțire.

5.1.5 Comparație

Comparația a două semnale se poate implementa foarte ușor utilizând operatorii relaționali ai limbajului VHDL. Acești operatori sunt =, /=, >, <, >=, <=. Figura 31 evidențiază modulul VHDL ce implementează comparația a două semnale cu ajutorul operatorilor relaționali. În figura 32 este reprezentat circuitul sintetizat.

```
library ieee; use ieee.std_logic_1164.all;
entity comp is port ( A, B : in std_logic_vector(3 downto 0);
egal, diferit, mai_mare, mai_mare_sau_egal, mai_mic, mai_mic_sau_egal : out std_logic );
end comp;
architecture comp_arch of comp is
begin
egal         <= '1' when A = B else '0'; diferit                <=  '1' when A /= B   else '0';
mai_mare <= '1' when A > B else '0'; mai_mare_sau_egal <=  '1' when A >= B  else '0';
mai_mic   <= '1' when A < B else '0'; mai_mic_sau_egal   <=  '1' when A <= B  else '0';
end comp_arch;
```

Figura 31 – Modul VHDL ce implementează comparația a două semnale.

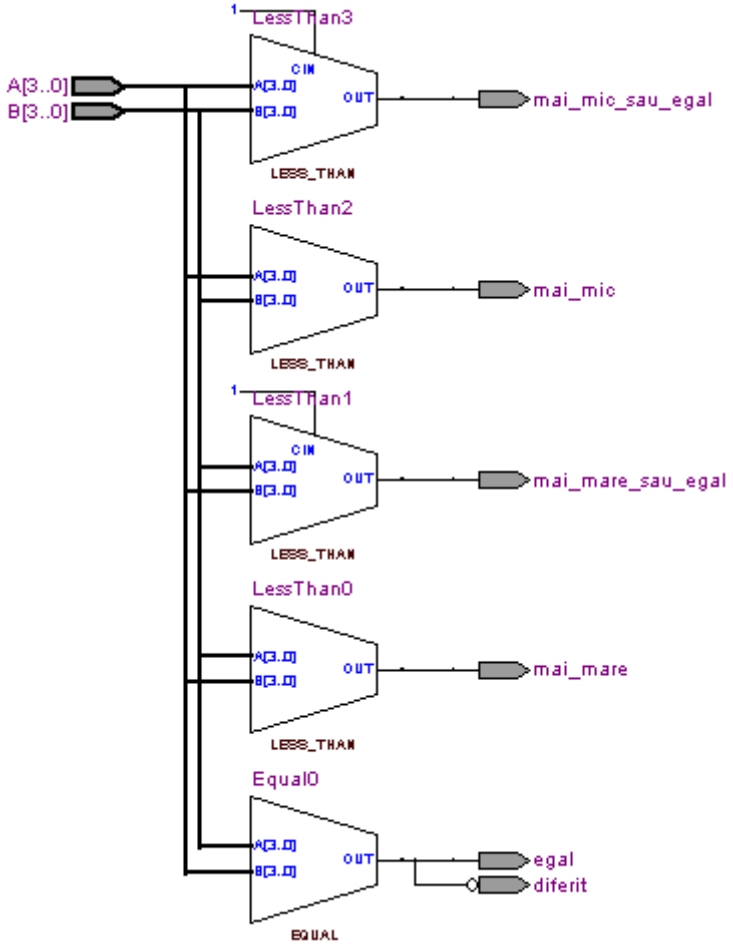


Figura 32 – Circuit sintetizat cu ajutorul modulului VHDL din figura 31.

În practică operatorii = și < se folosește cel mai des. Operatorul relațional = este ușor de implementat, însă operatorul relațional este destul de costisitor pentru a fi implementat.

5.2 Operații logice

5.2.1 Not, and, or, nand, nor, xor și xnor

Operațiile logice uzuale sunt: NOT, AND, OR, NAND, NOR, XOR și XNOR. Aceste operații sunt evidențiate prin operatorii logici NOT, AND, OR, NAND, NOR, XOR și XNOR. Acești operatori sunt definiți în subcapitolul 2.1 Operatori binari. Operațiile logice se implementează cu ajutorul porților logice fundamentale evidențaite în subcapitolul 2.5 Circuite logice combinaționale – module comportamentale.

5.2.2 Circuite de deplasare și rotire

Circuitele de deplasare și rotire sunt evidențiate prin operatorii de deplasare descriși în subcapitolul 2.1 Operatori binari. Operatorii de deplasare sunt sll, srl, sla, sra, rol, ror. Cei mai uzitați operatori sunt sll, srl, sra. Circuitele de deplasare evidențiate de acești operatori sunt reprezentate în subcapitolul 2.6 Circuite logice combinaționale – module structurale.

5.3 ALU

Un ALU aduce laolaltă circuite logice ce implementează operațiile aritmetice și logice prezentate în subcapitolele anterioare. Un ALU este o unitate aritmetico-logică cu două intrări pe un număr M de biți, cu o intrare de selecție, Sel, ce va selecta operația dorită, și o ieșire pe același număr de biți, M.

În figura 33 este reprezentat tabelul de adevăr al unui ALU. În funcție de intrarea de selecție se poate opta pentru o operație aritmetică sau logică. În figura 34 este evidențiat modulul VHDL ce implementează tabelul din figura 33. Trebuie precizat că semnalele de intrare vor fi numere întregi binare cu reprezentare în complement față de 2. Semnalele de intrare și ieșire vor avea o capacitate de 8 biți iar semnalul de selecție va avea o capacitate de 3 biți. Operațiile aritmetice sunt: adunare, scădere, egalitate și mai mic. Operațiile logice sunt: OR, AND, NOR, sll, și srl. Unitatea aritmetico-logică are ca operație implicită operația aritmetică de egalitate.

Trebuie menționat că acestă configurare, evidențiată în figura 33, nu este unică. Există ALU-uri ce implementează și alte funcții logice sau aritmetice. ALU-urile pot avea ieșiri suplimentare ce indică faptul că o operație de adunare a depășit sau că ieșirea ALU-ului are valoarea 0.

Sel	Operația aritmetico-logică	Observații
000	adunare	$A + B$
001	scădere	$A - B$ sau $A + \bar{B} + 1$
010	OR	$A \mid B$
011	AND	$A \& B$
100	sll	$A \ll B$, deplasare logică la stânga
101	srl	$A \gg B$, deplasare logică la dreapta
110	NOR	$A\ NOR\ B = \bar{A}\ OR\ \bar{B}$
111	mai mic	dacă $A < B$, $S = 1$, dacă nu, $S = 0$
- - -	egalitate	A = = B

Figura 33 – Tabel de adevăr pentru operațiile aritmetico-logice ale unui ALU.

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

entity ALU is
        port ( A, B : in std_logic_vector(7 downto 0);
                sel    : in std_logic_vector(2 downto 0);
                S      : out std_logic_vector(7 downto 0);
                Egal : out std_logic );
end ALU;

architecture ALU_arch of ALU is

begin
process(A, B, sel)
```

```
begin
case sel is
        when "000" => S <= A + B;
        when "001" => S <= A + not(B) + '1';
        when "010" => S <= A or B;
        when "011" => S <= A and B;
        when "100" => S <= conv_std_logic_vector (shl(conv_unsigned(conv_integer(A), 8),
conv_unsigned(conv_integer(B), 8)), 8);
        when "101" => S <= conv_std_logic_vector (shr(conv_unsigned(conv_integer(A), 8),
conv_unsigned(conv_integer(B), 8)), 8);
        when "110" => S <= A nor B;
        when OTHERS =>
                if (A < B) then
                    S <= "00000001";
                else S <= (others => '0');
                end if;
        end case;
end process;

Egal <= '1' when A = B else '0';

end ALU_arch;
```

Figura 34 – Modul VHDL ce implementează tabelul din figura 33.

Capitolul 6

Sintetizarea mașinilor cu stări finite

Sintetizarea mașinilor cu stări finite

Mașinile cu stări finite sunt circuite logice secvențiale sincrone. Există două tipuri de mașini cu stări finite: **de tip Moore** și **de tip Mealy**. Aceste mașini sunt alcătuite dintr-un modul secvențial și două module combinaționale. Mașinile cu stări finite, după cum reiese din denumirea acestora, au un **număr finit de stări**, stări ce se succed sincron cu impulsul de tact ($S0, S1, S2, S3, \dots, S0, S1, \dots$). O mașină cu stări finite prezintă două tipuri de stări: **starea curentă** (starea la momentul prezent) și **starea următoare** (starea viitoare). Starea următoare este memorată cu ajutorul modulului secvențial și oferită la ieșire ca stare curentă. Starea următoare este calculată pe baza stării curente și a semnalului de intrare cu ajutorul unui modul combinațional. În limbajul VHDL starea curentă și starea următoare pot fi definite cu ajutorul tipului de date enumaration, evidențiat în subcapitolul 6.1. Subcapitolele 6.2, 6.3 și 6.4 prezintă comportamentul ideal al mașinilor cu stări finite. În subcapitolul 6.5 este prezentat comportamentul temporal al acestor mașini.

Figura 1 prezintă o analogie între stărilor mașinilor cu stări finite și stările fețelor vesele din cadrul unui software de socializare.

Figura 1 – Starea fețelor vesele.

6.1 Tipul de date enumeration

În figura 2 este evidențiat modul în care se pot defini stările unei mașini cu stări finite în limbaj VHDL. Tipul de date enumeration permite reprezentarea stărilor într-o formă abstractă, sub formă de cuvinte și nu sub forma unei anumite codificări binare. Acest fapt reprezintă un avantaj deoarece software-ul ce va sinteza mașina cu stări finite poate înlocui stările, definite abstract prin cuvinte, cu diferite codificări binare, pentru a găsi configurația care se va implementa cu un minim de circuite logice. Unele software-uri ce sintetizează circuite digitale oferă utilizatorilor posibilitatea alegerii unei anumite codificări binare.

```
...
architecture FSM_circ_arch of FSM_circ is

TYPE states IS (S0, S1, S2, S4, S5);
SIGNAL current_state, next_state : states;

begin
…
end FSM_circ_arch;
```

Figura 2 – Modul de definire al stărilor unei mașini cu stări finite.

În figura 2 s-a definit tipul de date **states**, fiind de tip enumaration, putând lua valorile $\boldsymbol{S0, S1, S2, S3, S4, S5}$. S-au definit două semnale de tipul states ce poartă nume sugestive: **current_state** (starea curentă) și **next_state** (starea următoare). Se observă că locul de definiție al celor două linii de cod este între architecture și begin.

6.2 Mașina cu stări finite de tip Moore

În figura 3 este prezentată structura generală a unei mașini cu stări finite (FSM) de tip Moore. Blocul numit starea următoare este un circuit logic combinațional ce determină starea următoare pe baza stării curente și a intrării în FSM. Blocul numit ieșire este un circuit logic combinațional ce determină ieșirea la un moment dat pe baza stării curente. Blocul numit FF este un registru, un circuit logic secvențial, ce memorează starea următoare și oferă la ieșire

starea curentă. Se observă că datorită blocului FF stările FSM-ului de tip Moore se succed sincron cu impulsul de tact (CLK).

În figura 4 este prezentată forma generală a unui FSM de tip Moore printr-un modul VHDL. Este evidențiată definirea stărilor sistemului cu ajutorul tipului de date enumaration. Arhitectura modulului VHDL, pentru a fi ușor de debug-uit, a fost împărțită în trei procese.

Primul proces evidențiază comportamentul blocului secvențial FF (registru). Datele din registru sunt șterse asincron cu ajutorul semnalului de reset, rst. Next_state este memorată în registru pe frontul crescător al semnalului de tact (clk'event and clk = '1') și oferită la ieșire drept current_state.

Al doilea proces evidențiază comportamentul blocului combinațional starea următoare. Se observă că lista de senzitivitate cuprinde două semnale: semnalul de intrare, data_input și starea curentă, current_state. Cu ajutorul acestor două semnale se vor calcula valorile semnalului next_state.

Al treilea proces determină comportamentul blocului combinațional ieșire. Lista de senzitivitate cuprinde semnalul current_state cu ajutorul căruia se vor calcula valorile semnalului data_output.

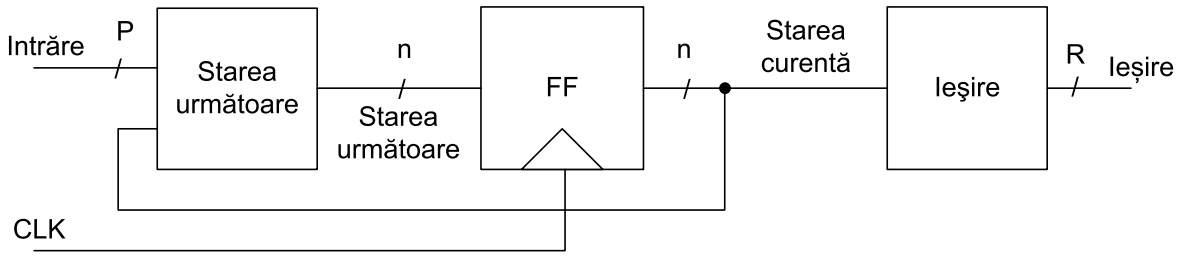


Figura 3 – Structura generală a unui FSM de tip Moore.

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

entity FSM_circ is
        port( clk, rst, data_input : in std_logic; data_output : out std_logic);
end FSM_circ;
```

```
architecture FSM_circ_arch of FSM_circ is

type statetype is (S0, S1, S2, ... );
signal current_state, next_state : statetype;

begin

---------- blocul secvențial FF ----------
        process (rst, clk)
        begin
                if (rst = ’1’) then
                        current_state <= S0;
                elsif(clk'event and clk = '1') then
                        current_state <= next_state;
                end if;
        end process;

---------- starea următoare ----------
        process (current_state, data_input)
        begin
                case current_state is
                        when S0 => if data_input = '1' then
                                                next_state <= S1;
                                        else
                                                next_state <= S0;
                                        end if;
                        when S1 => if data_input = '1' then
                                                next_state <= S2;
                                        else
                                                next_state <= S1;
                                        end if;
                                         .
                                         .
```

```
                                    .
                        when OTHERS => next_state <= S0;
                    end case;
            end process;

---------- ieșire ----------
            process (current_state)
            begin
                    if current_state = S0 then
                            data_output <= '0';
                    elsif current_state = S1 then
                            data_output <= '1';
                    elsif current_state = S2 then
                            data_output <= '0';
                     .
                     .
                     .
                    end if;
            end process;

end FSM_circ_arch;
```

Figura 4 – Modul VHDL ce prezintă forma generală a unui FSM de tip Moore.

Exemplul 1

În figura 5 este prezentată diagrama de stare pentru o mașină cu stări finite de tip Moore. Cu ajutorul acestei diagrame se poate deduce ușor tabelul de adevăr, evidențiat în figura 6, pentru starea următoare, respectiv ieșirea FSM-ului de tip Moore. Se observă, în figura 6, că ieșirea la un moment dat depinde doar de starea curentă.

Cu ajutorul diagramei de stare sau cu ajutorul tabelelor de adevăr se poate scrie modulul VHDL pentru FSM-ul de tip Moore. Acest modul este evidențiat în figura 7. Circuitul sintetizat este reprezentat în figura 8.

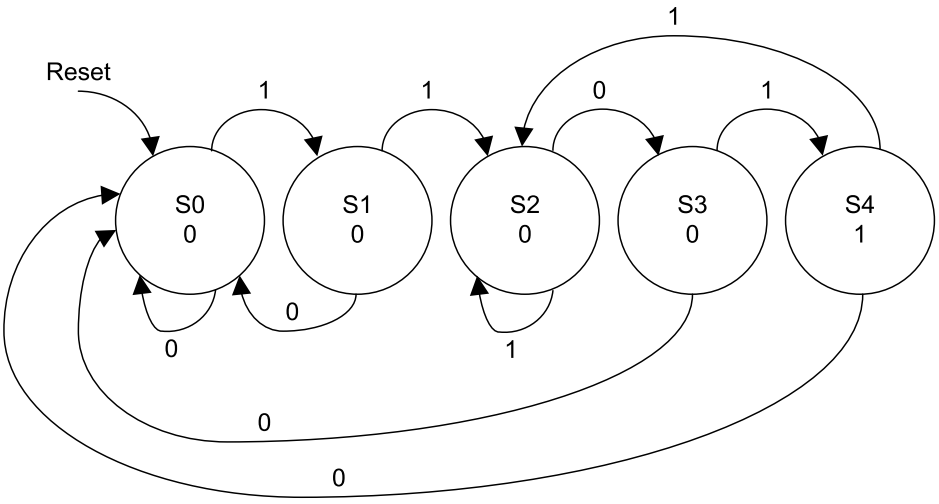


Figura 5 – Diagrama de stare pentru o maşină cu stări finite de tip Moore.

Starea actuală	Intrare	Starea următoare
S0	0	S0
S0	1	S1
S1	0	S0
S1	1	S2
S2	0	S3
S2	1	S2
S3	0	S0
S3	1	S4
S4	0	S0
S4	1	S2

Starea actuală	Ieşire
S0	0
S1	0
S2	0
S3	0
S4	1

Figura 6 – Tabele de adevăr pentru starea următoare şi ieşire pentru o maşină cu stări finite de tip Moore.

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

entity FSM_circ is
        port( clk, rst, data_input : in std_logic; data_output : out std_logic);
end FSM_circ;
architecture FSM_circ_arch of FSM_circ is

type statetype is (S0, S1, S2, S3, S4);
signal current_state, next_state : statetype;
```

```
begin

---------- blocul secvențial FF ----------
        process (rst, clk)
        begin
                if (rst = ’1’) then
                        current_state <= S0;
                elsif(clk'event and clk = '1') then
                        current_state <= next_state;
                end if;
        end process;

---------- starea următoare ----------
        process (current_state, data_input)
        begin
                case current_state is
                        when S0 => if data_input = '1' then
                                        next_state <= S1;
                                   else
                                        next_state <= S0;
                                   end if;
                        when S1 => if data_input = '1' then
                                        next_state <= S2;
                                   else
                                        next_state <= S0;
                                   end if;
                        when S2 => if data_input = '0' then
                                        next_state <= S3;
                                   else
                                        next_state <= S2;
                                   end if;
                        when S3 => if data_input = '1' then
```

```
                                    next_state <= S4;
                                else
                                    next_state <= S0;
                                end if;
                    when S4 => if data_input = '1' then
                                    next_state <= S2;
                                else
                                    next_state <= S0;
                                end if;
                    when OTHERS => next_state <= S0;
                end case;
        end process;

---------- ieșire ----------
        process (current_state)
        begin
            if current_state = S0 then
                data_output <= '0';
            elsif current_state = S1 then
                data_output <= '0';
            elsif current_state = S2 then
                data_output <= '0';
            elsif current_state = S3 then
                data_output <= '0';
            elsif current_state = S4 then
                data_output <= '1';
            else  data_output <= '0';
            end if;
        end process;

end FSM_circ_arch;
```

Figura 7 – Modul VHDL ce evidențiază diagrama de stare a FSM-ului de tip Moore din figura 5.

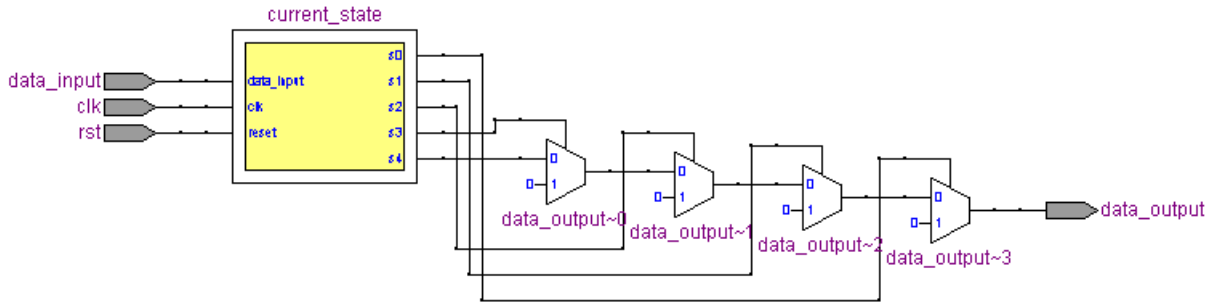


Figura 8 – Circuit sintetizat cu ajutorul modulului VHDL din figura 7.

6.3 Mașina cu stări finite de tip Mealy

În figura 9 este prezentată structura generală a unei mașini cu stări finite (FSM) de tip Mealy. Blocul numit starea următoare este un circuit logic combinațional ce determină starea următoare pe baza stării curente și a intrării în FSM. Blocul numit ieșire este un circuit logic combinațional ce determină ieșirea la un moment dat pe baza stării curente și a intrării. Blocul numit FF este un registru, un circuit logic secvențial, ce memorează starea următoare și oferă la ieșire starea curentă. Se observă că datorită blocului FF stările FSM-ului de tip Mealy se succed sincron cu impulsul de tact (CLK).

În figura 10 este prezentată forma generală a unui FSM de tip Mealy printr-un modul VHDL. Este evidențiată definirea stărilor sistemului cu ajutorul tipului de date enumaration. Arhitectura modulului VHDL, pentru a fi ușor de debug-uit, a fost împărțită în trei procese.

Primul proces evidențiază comportamentul blocului secvențial FF (registru). Datele din registru sunt șterse asincron cu ajutorul semnalului de reset, rst. Next_state este memorată în registru pe frontul crescător al semnalului de tact (clk'event and clk = '1') și oferită la ieșire drept current_state.

Al doilea proces evidențiază comportamentul blocului combinațional starea următoare. Se observă că lista de senzitivitate cuprinde două semnale: semnalul de intrare, data_input și starea curentă, current_state. Cu ajutorul acestor două semnale se vor calcula valorile semnalului next_state.

Al treilea proces determină comportamentul blocului combinațional ieșire. Lista de senzitivitate cuprinde semnalul current_state și data_input cu ajutorul cărora se vor calcula valorile semnalului data_output.

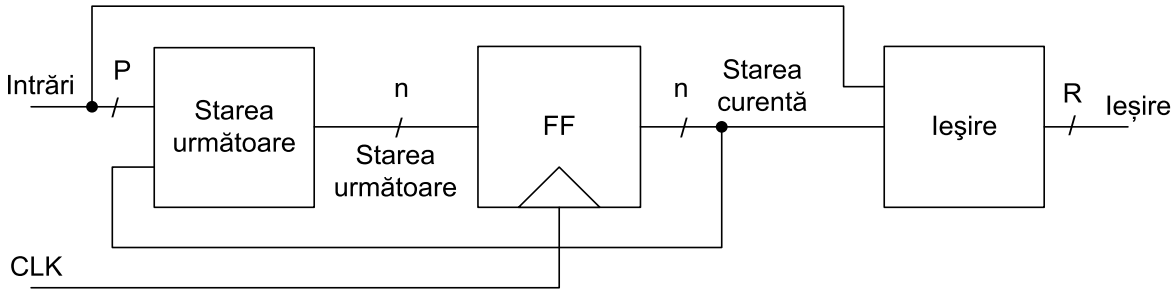


Figura 9 – Maşină cu stări finite de tip Mealy.

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

entity FSM_circ is
        port( clk, rst, data_input : in std_logic; data_output : out std_logic);
end FSM_circ;

architecture FSM_circ_arch of FSM_circ is

type statetype is (S0, S1, S2, ... );
signal current_state, next_state : statetype;

begin

---------- blocul secvențial FF ----------
        process (rst, clk)
        begin
                if (rst = ’1’) then
                        current_state <= S0;
                elsif(clk'event and clk = '1') then
                        current_state <= next_state;
                end if;
        end process;

---------- starea următoare ----------
```

```
        process (current_state, data_input)
        begin
                case current_state is
                        when S0 => if data_input = '1' then
                                        next_state <= S1;
                                   else
                                        next_state <= S0;
                                   end if;
                        when S1 => if data_input = '1' then
                                        next_state <= S2;
                                   else
                                        next_state <= S1;
                                   end if;
                                   .
                                   .
                                   .
                        when OTHERS => next_state <= S0;
                end case;
        end process;

---------- ieșire ----------
        process (current_state, data_input)
        begin
                if (current_state = S0 & data_input = ’1’) then
                        data_output <= '0';
                elsif (current_state = S0 & data_input = ’0’) then
                        data_output <= '1';
                elsif (current_state = S1 & data_input = ’1’) then
                        data_output <= '0';
                elsif (current_state = S1 & data_input = ’0’) then
                        data_output <= '0';
                  .
                  .
```

```
                .
            end if;
        end process;

end FSM_circ_arch;
```

Figura 10 – Modul VHDL ce prezintă forma generală a unui FSM de tip Mealy.

Exemplul 2

În figura 11 este prezentată diagrama de stare pentru o mașină cu stări finite de tip Mealy. Cu ajutorul acestei diagrame se poate deduce ușor tabelul de adevăr, evidențiat în figura 12, pentru starea următoare, respectiv ieșirea FSM-ului de tip Mealy. Se observă, în figura 12, că ieșirea se modifică ori de câte ori intrarea își modifica valoarea sau ori de câte ori există o tranziție dintr-o stare în altă stare.

Trebuie precizat că exemplele 1 și 2 sunt identice, adică prezintă același comportament ideal. Ceea ce diferă sunt: diagramele de stare, tabelele de adevăr și modulele VHDL. Se observă că FSM-ul de tip Mealy, din figura 11, are 4 stări, cu una mai puțin ca FSM-ul de tip Moore din figura 12. Cu ajutorul diagramei de stare sau cu ajutorul tabelului de adevăr se poate scrie modulul VHDL pentru FSM-ul de tip Mealy. Acest modul este evidențiat în figura 13. Circuitul sintetizat este reprezentat în figura 14.

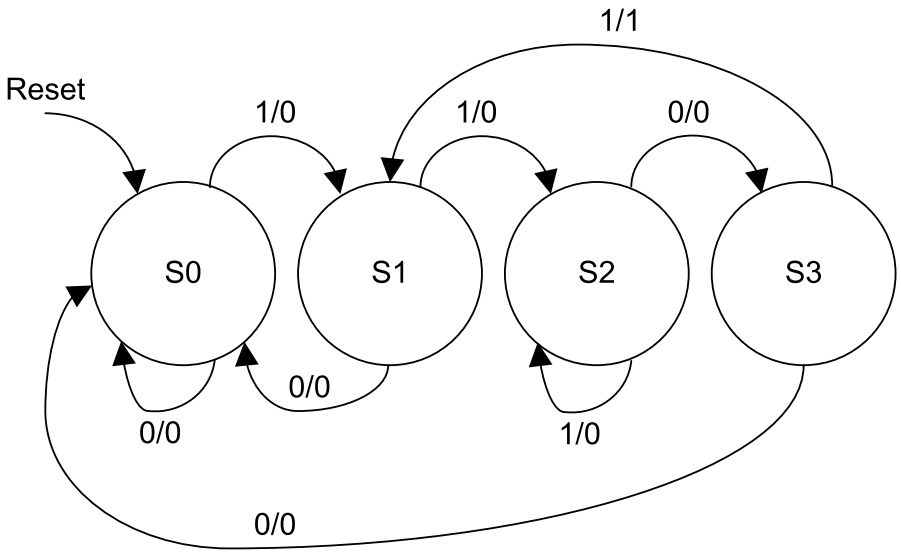


Figura 11 – Diagrama de stare pentru o maşină cu stări finite de tip Mealy.

Starea actuală	Intrare	Starea următoare	Ieşire
S0	0	S0	0
S0	1	S1	0
S1	0	S0	0
S1	1	S2	0
S2	0	S3	0
S2	1	S2	0
S3	0	S0	0
S3	1	S1	1

Figura 12 – Tabelul de adevăr pentru starea următoare şi ieşire pentru o maşină cu stări finite de tip Mealy.

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

entity FSM_circ is
        port(
                clk, rst, data_input : in std_logic;
                data_output          : out std_logic
               );
end FSM_circ;

architecture FSM_circ_arch of FSM_circ is

type statetype is (S0, S1, S2, S3);
signal current_state, next_state : statetype;

begin

---------- blocul secvențial FF ----------
        process (rst, clk)

        begin
                if (rst = '1') then
                        current_state <= S0;
```

```
        elsif(clk'event and clk = '1') then
            current_state <= next_state;
        end if;
    end process;

---------- starea următoare ----------
    process (current_state, data_input)

    begin
        case current_state is
            when S0 => if data_input = '1' then
                           next_state <= S1;
                       else
                           next_state <= S0;
                       end if;

            when S1 => if data_input = '1' then
                           next_state <= S2;
                       else
                           next_state <= S0;
                       end if;

            when S2 => if data_input = '0' then
                           next_state <= S3;
                       else
                           next_state <= S2;
                       end if;

            when S3 => if data_input = '1' then
                           next_state <= S1;
                       else
                           next_state <= S0;
                       end if;
```

```
                    when OTHERS => next_state <= S0;
                end case;
        end process;

---------- ieşire ----------
        process (current_state, data_input)

        begin
                if (current_state = S0 and data_input = ’0’) then
                        data_output <= '0';
                elsif (current_state = S0 and data_input = ’1’) then
                        data_output <= '0';
                elsif (current_state = S1 and data_input = ’0’) then
                        data_output <= '0';
                elsif (current_state = S1 and data_input = ’1’) then
                        data_output <= '0';
                elsif (current_state = S2 and data_input = ’0’) then
                        data_output <= '0';
                elsif (current_state = S2 and data_input = ’1’) then
                        data_output <= '0';
                elsif (current_state = S3 and data_input = ’0’) then
                         data_output <= '0';
                elsif (current_state = S3 and data_input = ’1’) then
                        data_output <= '1';
                else
                       data_output <= '0';
        end if;
        end process;

end FSM_circ_arch;
```

Figura 13 - Modul VHDL ce evidențiază diagrama de stare a FSM-ului de tip Mealy din figura 11.

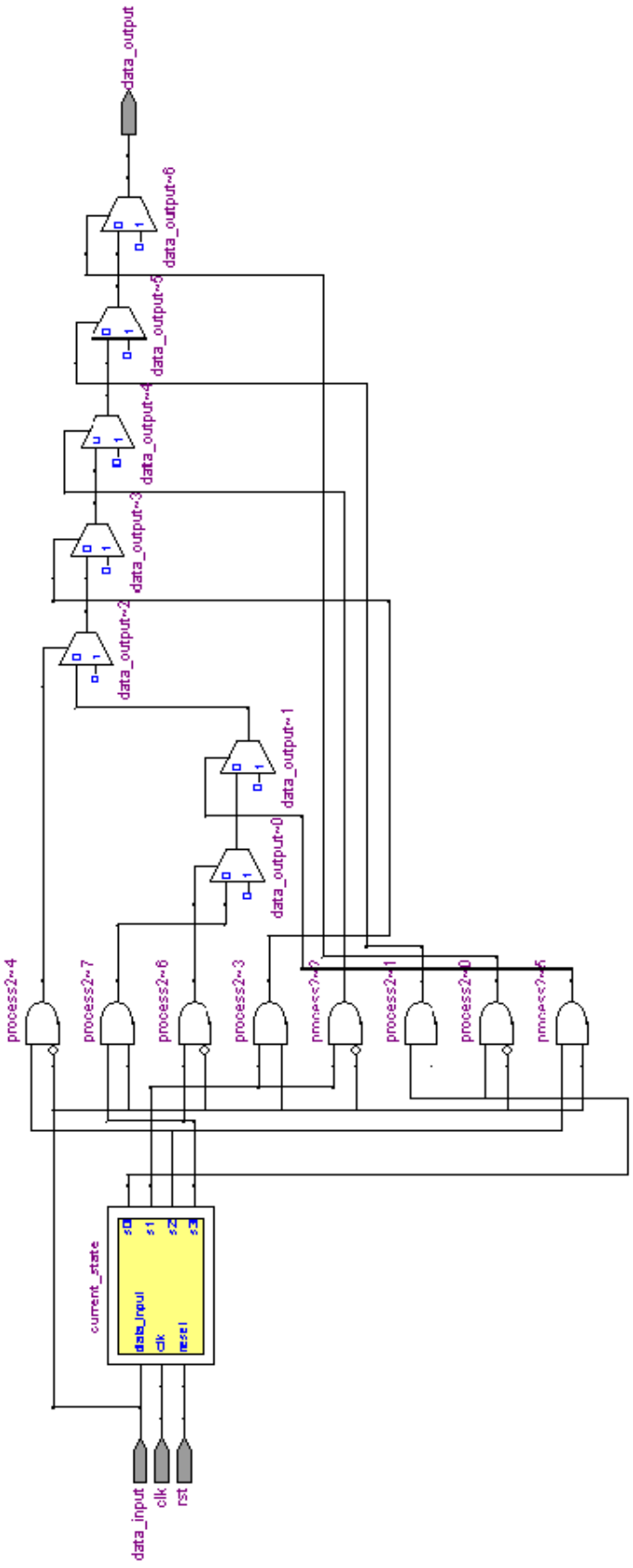


Figura 14 – Circuit sintetizat cu ajutorul modulului VHDL din figura 13.

6.4 Maşina cu stări finite de tip Mealy sincronă

În figura 15 este prezentată structura generală a unei maşini cu stări finite (FSM) de tip Mealy sincronă. Se observă că acest tip de FSM are în plus faţă de FSM-ul de tip Mealy, un bloc secvenţial FF (registru) pentru datele de ieşire. Scopul introducerii acestui bloc este acela de a sincroniza datele de la ieşire cu impulsul de tact. În figura 16 este prezentată forma generală a unei maşini cu stări finite de tip Mealy sincronă printr-un modul VHDL.

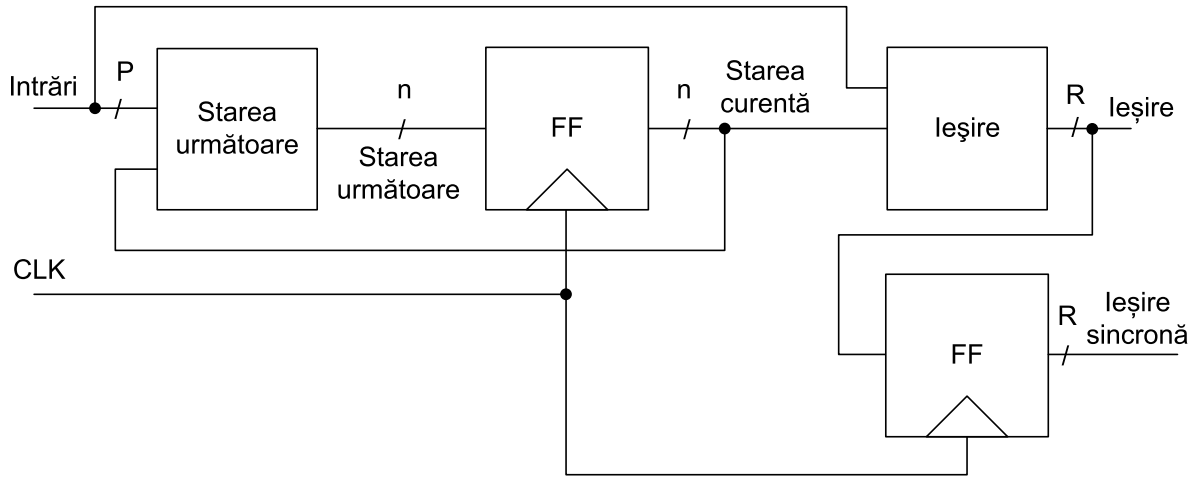


Figura 15 – Maşină cu stări finite de tip Mealy sincronă.

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

entity FSM_circ is
        port(
                clk, rst, data_input              : in std_logic;
                data_output, data_output_reg : out std_logic
              );
end FSM_circ;

architecture FSM_circ_arch of FSM_circ is

type statetype is (S0, S1, S2, ... );
signal current_state, next_state : statetype;
```

```
begin

---------- blocul secvențial FF ----------
        process (rst, clk)
        begin
                if (rst = ’1’) then
                        current_state <= S0;
                elsif(clk'event and clk = '1') then
                        current_state <= next_state;
                end if;
        end process;

---------- starea următoare ----------
        process (current_state, data_input)
        begin
                case current_state is
                        when S0 => if data_input = '1' then
                                        next_state <= S1;
                                   else
                                        next_state <= S0;
                                   end if;
                        when S1 => if data_input = '1' then
                                        next_state <= S2;
                                   else
                                        next_state <= S1;
                                   end if;
                                    .
                                    .
                                    .
                        when OTHERS => next_state <= S0;
                  end case;
        end process;
```

```
---------- ieșire ----------
        process (current_state, data_input)
        begin
                if (current_state = S0 & data_input = ’1’) then
                        data_output <= '0';
                elsif (current_state = S0 & data_input = ’0’) then
                        data_output <= '1';
                elsif (current_state = S1 & data_input = ’1’) then
                        data_output <= '0';
                elsif (current_state = S1 & data_input = ’0’) then
                        data_output <= '0';
                  .
                  .
                  .
                end if;
        end process;

---------- blocul secvențial FF ieșire sincronă ----------
        process (rst, clk)

         begin
                if (rst = ’1’) then
                        data_output_reg <= (OTHERS => ’0’);
                elsif (clk'event and clk = '1') then
                        data_output_reg <= data_output;
                end if;
        end process;

end FSM_circ_arch;
```

Figura 16 – Modul VHDL ce prezintă forma generală a unei mașini cu stări finite de tip Mealy sincronă

6.5 Mașini cu stări finite – comportament temporal

Viteza de procesare a mașinilor cu stări finite este determinat de perioada, T_c, sau frecvența impulsurilor de tact, f_c. Perioada sau frecvența impulsurilor de tact este determinată de comportamentul temporal al blocului combinațional starea următoare și comportamentul temporal al blocului secvențial FF (registru). Pentru o funcționare corectă din punct de vedere temporal a unei mașini cu stări finite trebuie îndeplinite două condiții: condiția de configurare și condiția de reținere.

În figura 17 este prezentată structura comună a mașinilor cu stări finite de tip Moore și de tip Mealy și sensul de deplasare al datelor. Acestă structură comună este alcătuită din blocul combinațional starea următoare și blocul secvențial FF.

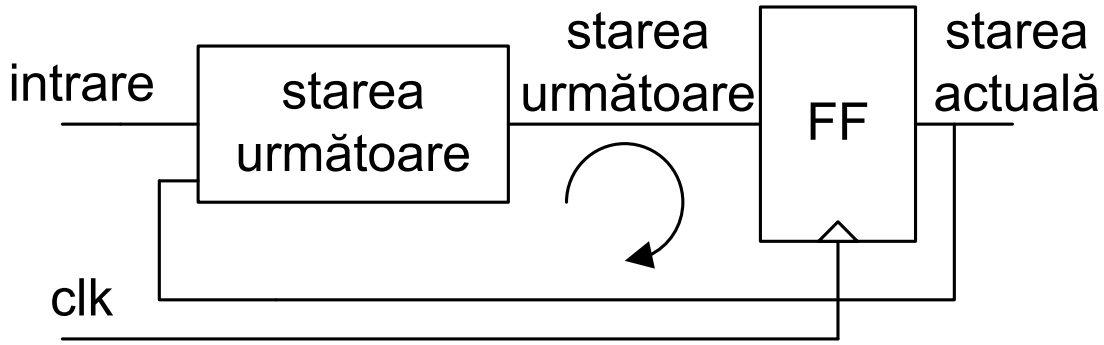


Figura 17 – Structura comună a celor două tipuri de mașini cu stări finite.

6.5.1 Îndeplinirea condiției de configurare

Comportamentul temporal al unui circuit logic combinațional este determinat de întârzierea de contaminare, t_{cd}, și întârzierea de propagare, t_{pd}. Comportamentul temporal al unui circuit logic secvențial, cum ar fi un flip-flop sau registru, este determinat de timpul de configurare, t_{setup}, timpul de reținere, t_{hold}, întârzierea de contaminare, t_{ccq}, și întârzierea de propagare, t_{pcq}.

Pentru îndeplinirea condiției de configurare sunt considerați importanți următorii timpi: timpul de configurare al blocului secvențial FF și întârzierile de propagare, adică întârzierile maxime prin blocul combinațional și prin cel secvențial. Dacă se cunosc cei trei timpi, există posibilitatea calculării perioadei minime sau frecvenței maxime a impulsurilor de tact. Acești timpi sunt evidențiați în figura 18. Perioada minimă se poate calcula utilizând următoarea relație:

$$T_c \geq t_{pd} + t_{setup} + t_{pcq}$$

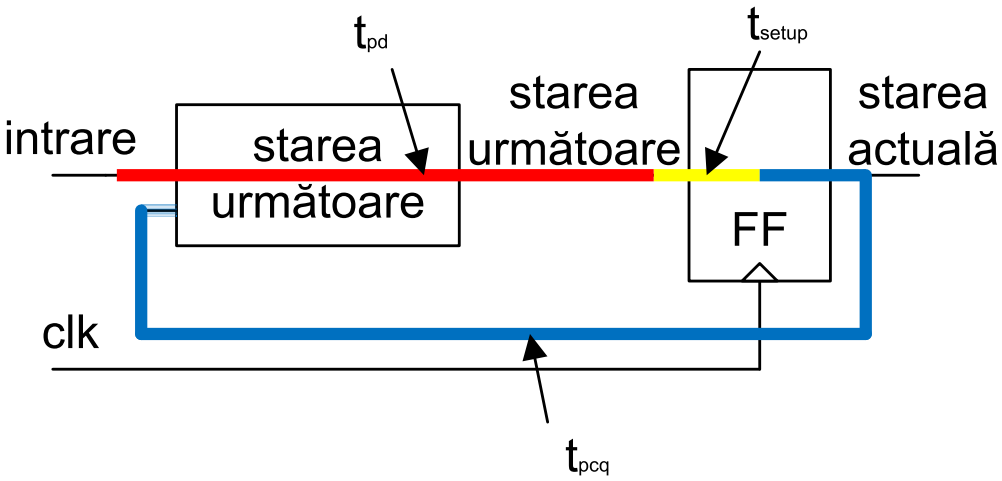


Figura 18 – Cei trei timpi pentru îndeplinirea condiției de configurare.

Relația de mai sus reprezintă condiția de configurare. În figura 19 este evidențiată funcționarea mașinii cu stări finite, din figura 18, printr-o diagramă temporală reală.

La un moment dat datele de la intrare se pot modifica. Acestă modificare duce la calcularea stării următoare într-un timp maxim, t_{pd}, fiind întârzierea de propagare a modulului combinațional starea următoare. Apoi la intrarea blocului secvențial FF valoarea stării următoare va trebui să se mențină constantă un timp, t_{setup}, fiind timpul de configurare. Blocul secvențial FF va memora starea următoare și o va furniza către ieșire ca starea actuală după un timp, t_{pcq}, fiind întârzierea de propagare a registrului.

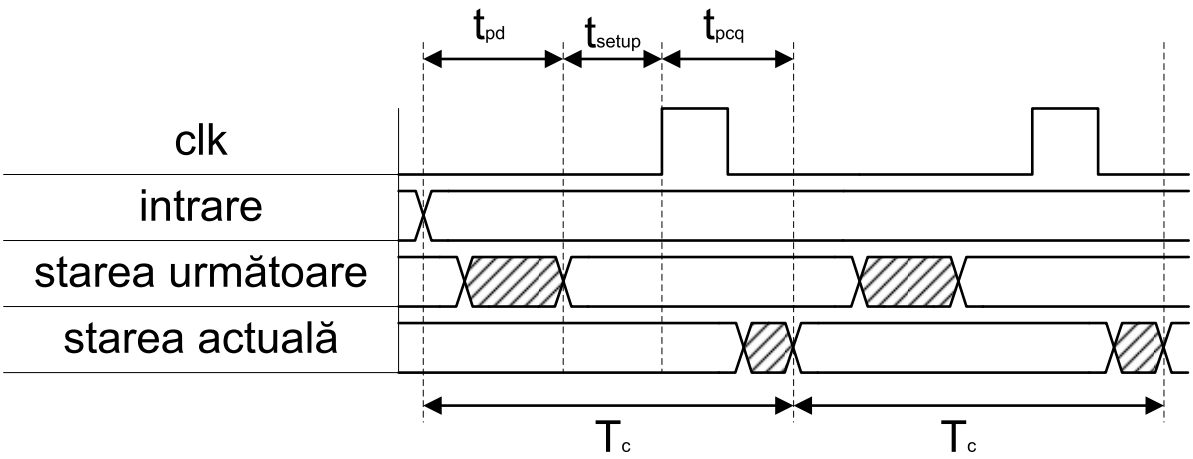


Figura 19 – Diagrama temporală reală a unui FSM și evidențierea timpilor pentru îndeplinirea condiției de configurare.

De obicei t_{setup} și t_{pcq} se cunosc din cataloagele cu componente secvențiale ale diferiților producători. Dacă perioada de repetiție a impulsului de tact, T_c ,a fost stabilită la începutul proiectării mașinii cu stări finite, nu ne rămâne de făcut decât să calculăm întârzierea de propagare (întârzierea maximă) pentru blocul combinațional starea următoare. Relația de calcul este următoarea:

$$t_{pd} \leq T_c - t_{setup} - t_{pcq}$$

6.5.2 Îndeplinirea condiției de reținere

Pentru îndeplinirea condiției de reținere sunt considerați importanți următorii timpi: timpul de reține al blocului secvențial FF și întârzierile de contaminare, adică întârzierile minime prin blocul combinațional și prin cel secvențial. Relația următoare reprezintă condiția de reținere:

$$t_{hold} \leq t_{ccq} + t_{cd}$$

Cei trei timpi sunt evidențiați în figura 20, în care se prezintă diagrama temporală reală a unei mașini cu stări finite.

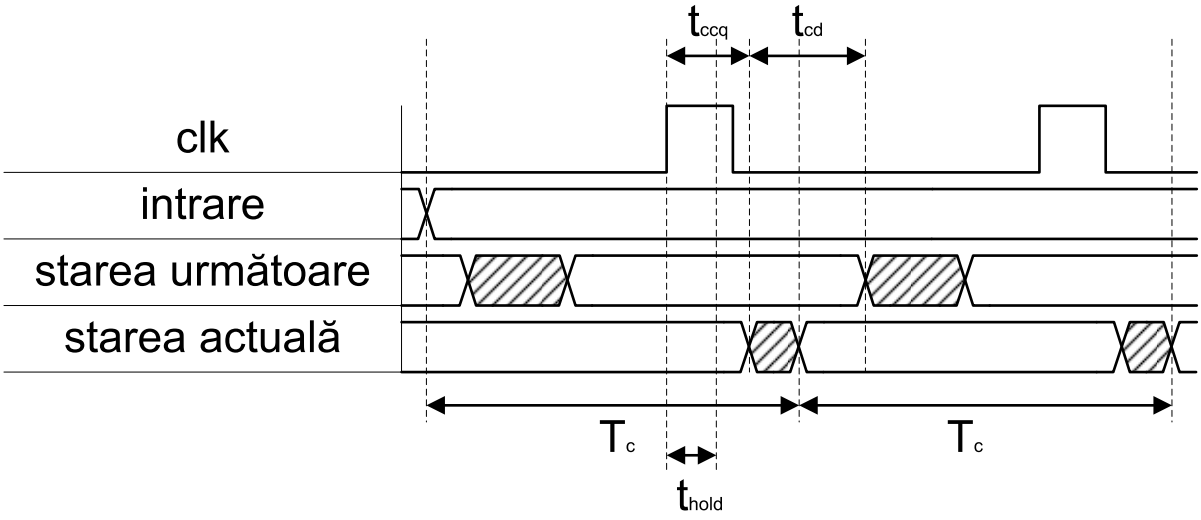


Figura 20 – Diagrama temporală reală a unui FSM și evidențierea timpilor pentru îndeplinirea condiției de configurare.

De obicei $\boldsymbol{t_{hold}}$ și $\boldsymbol{t_{ccq}}$ se cunosc din cataloagele cu componente secvențiale ale diferiților producători. nu ne rămâne de făcut decât să calculăm întârzierea de contaminare

(întârzierea minimă) pentru blocul combinațional starea următoare. Relația de calcul este următoarea:

$$t_{cd} \geq t_{hold} - t_{ccq}$$

Capitolul 7

Sintetizarea memoriilor și a controlerelor pentru memorii

Sintetizarea memoriilor și a controlerelor pentru memorii

Tipul de date array oferă posibilitatea descrierii memoriilor în limbaj VHDL. În funcție de volatilitate există două tipuri de memorii: memoria RAM (random access memory) și memoria ROM (read only memory). Memoria RAM este volatilă, iar memoria ROM este non-volatilă. Termenul de volatil se referă la faptul că memoria pierde datele stocate din momentul când nu mai este alimentată de la o sursă de tensiune. În funcție de tipul și numărul porturilor, memoria RAM poate fi: uniport, biport sau triport.

7.1 Tipul de date array

În figura 1 este reprezentată schematic organizarea internă a unei memorii. Pe verticală avem **numărul de rânduri**, notat cu **R**, iar pe orizontală avem **numărul de biți** sau **numărul de coloane**, notat cu **C**. Cu ajutorul tipului de date array se pot crea alte tipuri de date, având diferite dimensiuni. În figura 2 este evidențiat modul în care se poate crea memoria din figura 1, în limbaj VHDL, cu ajutorul tipului de date array. După cum se observă memoria din figura 1 este bidimensională. În consecință, în figura 2, s-a definit tipul de date bidimensional, **mem**, utilizând tipul de date array.

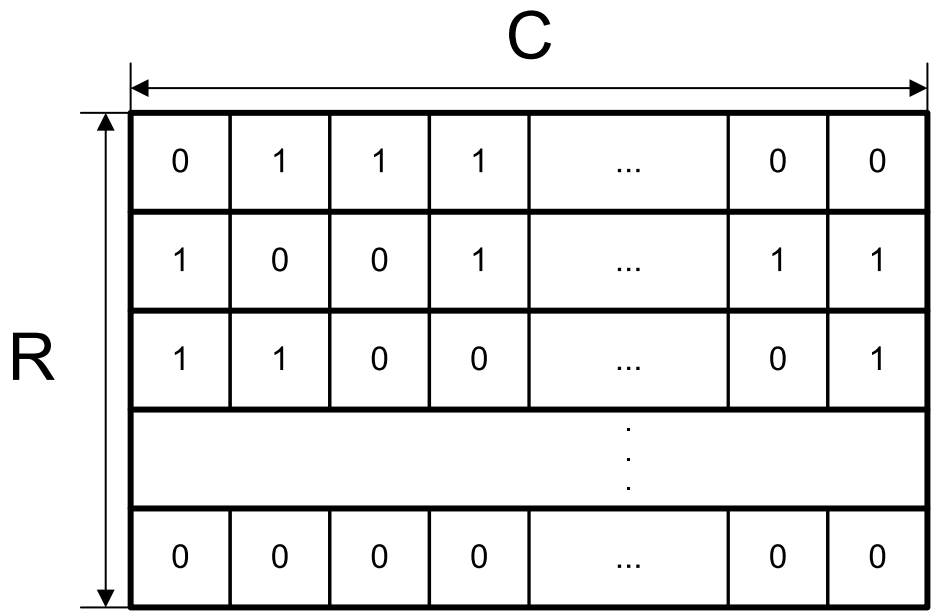


Figura 1 – Organizarea internă a unei memorii.

```
ARCHITECTURE ...

TYPE mem IS ARRAY(R-1 DOWNTO 0) OF STD_LOGIC_VECTOR(C-1 DOWNTO 0);
SIGNAL mem_array : mem;

BEGIN
        .
        .
        .
END ... ;
```

Figura 2 – Definirea unei memorii în limbaj VHDL.

7.2 Memoria RAM uniport

7.2.1 Memorie RAM cu port distinct de intrare și ieșire

În figura 3 este prezentat schematic acest tip de memorie RAM uniport. În figura 4 este evidențiat modulul VHDL pentru acest tip de memorie. Memoria RAM cu port distinct de intrare și ieșire funcționează în felul următor, atunci când semnalul $\overline{WE}$ are valoarea logică 0, se spune că memoria este scrisă, adică datele de la intrarea de date vor fi stocate în memorie în locația ce se găsește pe intrarea adresă. În același timp la ieșire vor fi furnizate datele stocate în memorie în locația ce se găsește pe intrarea adresă. Atunci când $\overline{WE}$ are valoarea logică 1, se spune că memoria este citită, adică la ieșire vor fi furnizate datele stocate în memorie în locația ce se găsește pe intrarea adresă. În figura 5 este reprezentat circuitul sintetizat cu ajutorul modulului din figura 4.

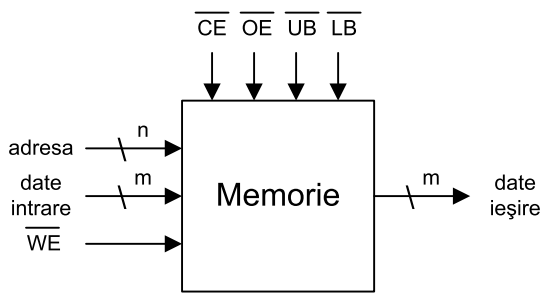


Figura 3 – Memorie RAM cu port distinct de intrare și ieșire.

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

entity mem is
        port ( clk          : in std_logic;
               address      : in std_logic_vector(3 downto 0);
               data_input   : in std_logic_vector(3 downto 0);
               wren         : in std_logic;
               data_output : out std_logic_vector(3 downto 0)
              );
end mem;

architecture arch_mem of mem is

type mem is array (15 downto 0) of std_logic_vector(3 downto 0);
signal memory_array : mem;

begin

process(clk)
begin
      if (clk'event and clk = '1') then
         if(wren = '0') then
            memory_array(conv_integer(address)) <= data_input;
         end if;
      end if;
end process;
            data_output <= memory_array(conv_integer(address));
end arch_mem;
```

Figura 4 – Modul VHDL ce evidențiază memoria RAM uniport cu port distinct de intrare și ieșire.

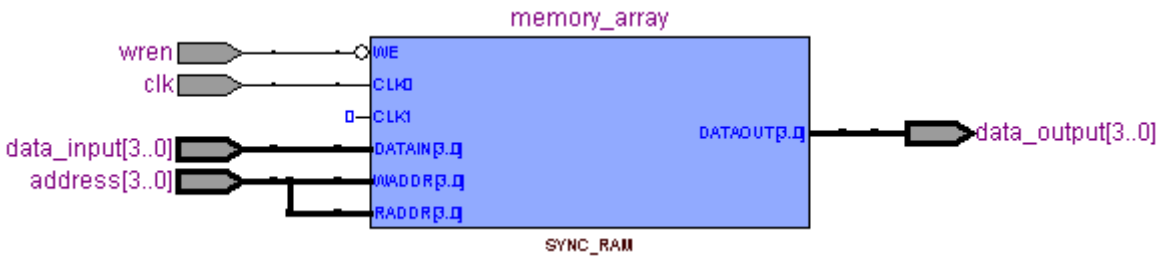


Figura 5 – Circuit sintetizat cu ajutorul modulului VHDL din figura 4.

7.2.2 Memorie RAM cu port bidirecțional intrare/ieșire

În figura 6 este prezentat schematic acest tip de memorie RAM uniport. În figura 7 este evidențiat modulul VHDL pentru controlererul acestui tip de memorie. Memoria RAM cu port bidirecțional intrare/ieșire funcționează în felul următor, atunci când semnalul $\overline{WE}$ are valoarea logică 0, se spune că memoria este scrisă, adică datele ce se găsesc pe portul bidirecțional vor fi stocate în memorie în locația ce se găsește pe intrarea adresă. În acest moment portul bidirecțional se comportă ca un port de intrare. Atunci când $\overline{WE}$ are valoarea logică 1, se spune că memoria este citită, adică pe portul bidirecțional se vor găsi datele stocate în memorie în locația ce se găsește pe intrarea adresă. În acest moment portul bidirecțional se comportă ca un port de ieșire. În figura 8 este reprezentat circuitul sintetizat cu ajutorul modulului din figura 7.

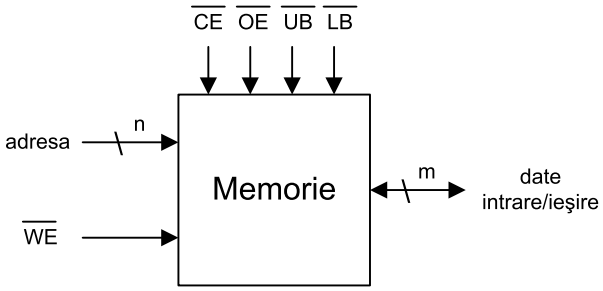


Figura 6 – Memorie RAM cu port bidirecțional intrare/ieșire.

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

entity mem is
        port ( data_in      : in std_logic_vector(3 downto 0);
```

```
            address        : in std_logic_vector(3 downto 0);
            wren           : in std_logic;
            address_out    : out std_logic_vector(3 downto 0);
            data_out       : out std_logic_vector(3 downto 0);
            data_inout     : inout std_logic_vector(3 downto 0)
            );
end mem;

architecture arch_mem of mem is
begin

process (data_in, wren)
begin
    if (wren = '0') then
        data_inout <= data_in;
    else
        data_inout <= (OTHERS => 'Z');
    end if;
end process;

process (data_inout, wren)
begin
    if (wren = '1') then
        sdata_out <= data_inout;
    else
        data_out <= (OTHERS => 'Z');
    end if;
end process;
        address_out <= address;
end arch_mem;
```

Figura 7 – Modul VHDL ce evidențiază controlerul pentru memoria RAM cu port bidirecțional intrare/ieșire.

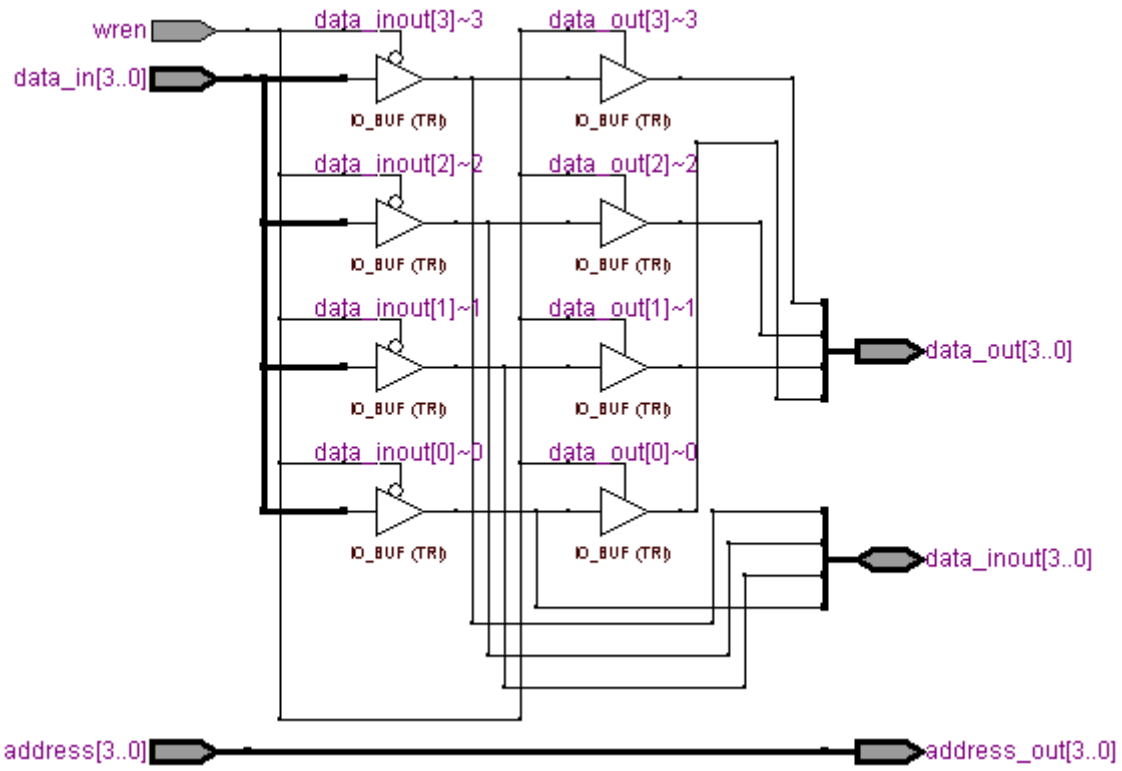


Figura 8 – Circuit sintetizat cu ajutorul modulului VHDL din figura 7.

7.2.3 Memorie RAM cu port distinct pentru intrarea de adresă

În figura 9 este reprezentat acest tip de memorie RAM uniport. În figura 10 este evidențiat modulul VHDL pentru memoria RAM cu port distinc pentru intrarea de adresă. Acestă memorie funcționează în felul următor, atunci când semnalul $\overline{WE}$ are valoarea logică 0, se spune că memoria este scrisă, adică datele de la intrarea de date vor fi stocate în memorie în locația ce se găsește pe intrarea adresă date intrare. În același timp la ieșire vor fi furnizate datele stocate în memorie în locația ce se găsește pe intrarea adresă date ieșire. Atunci când $\overline{WE}$ are valoarea logică 1, se spune că memoria este citită, adică la ieșire vor fi furnizate datele stocate în memorie în locația ce se găsește pe intrarea adresă date ieșire.

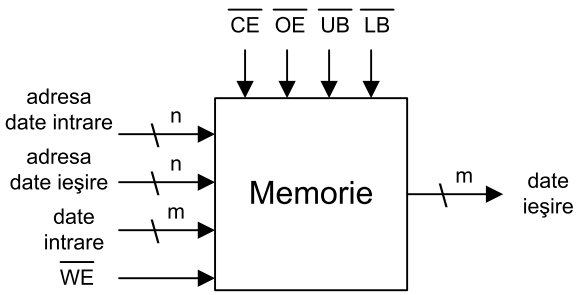


Figura 9 – Memorie RAM cu port distinct pentru intrarea de adresă.

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

entity mem is
        port ( clk          : in std_logic;
               address_in   : in std_logic_vector(3 downto 0);
               address_out : in std_logic_vector(3 downto 0);
               data_input   : in std_logic_vector(3 downto 0);
               wren         : in std_logic;
               data_output : out std_logic_vector(3 downto 0)
             );
end mem;

architecture arch_mem of mem is

type mem is array (15 downto 0) of std_logic_vector(3 downto 0);
signal memory_array : mem;

begin

process(clk)
begin
        if (clk'event and clk = '1') then
            if(wren = '0') then
                memory_array(conv_integer(address_in)) <= data_input;
            end if;
        end if;
end process;
               data_output <= memory_array(conv_integer(address_out));
end arch_mem;
```

Figura 10 – Modul VHDL ce evidențiază memoria RAM cu port distinct pentru intrarea de adresă.

În figura 11 este reprezentat circuitul sintetizat cu ajutorul modulului din figura 10.

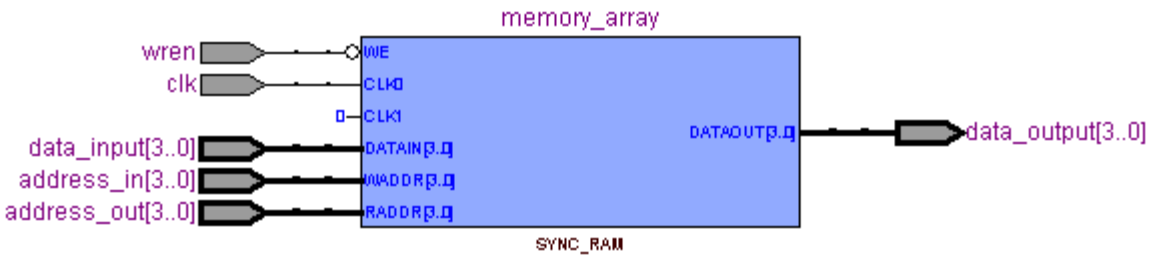


Figura 11 – Circuit sintetizat cu ajutorul modulului VHDL din figura 10.

7.3 Memoria RAM biport

În figura 12 este reprezentată memoria RAM biport. În figura 13 este evidențiat modulul VHDL pentru memoria RAM biport. Acestă memorie funcționează în felul următor, atunci când semnalele $\overline{WE}1$ și $\overline{WE}2$ au valoarea logică 0, se spune că memoria este scrisă, adică datele de la intrarea de date 1 și datele de la intrarea de date 2 vor fi stocate în memorie în locațiile ce se găsește pe intrările adresă date intrare1 și adresă date intrare 2. În același timp la ieșire pe porturile date ieșire 1 și date ieșire 2 vor fi furnizate datele stocate în memorie în locațiile ce se găsesc pe intrarea adresă date intrare 1 respectiv adresă date intrare 2. Atunci când $\overline{WE}$ are valoarea logică 1, se spune că memoria este citită, adică la ieșire pe porturile date ieșire 1 și date ieșire 2 vor fi furnizate datele stocate în memorie în locațiile ce se găsesc pe intrarea adresă date intrare 1 respectiv adresă date intrare 2. În figura 14 este reprezentat circuitul sintetizat cu ajutorul modulului din figura 13.

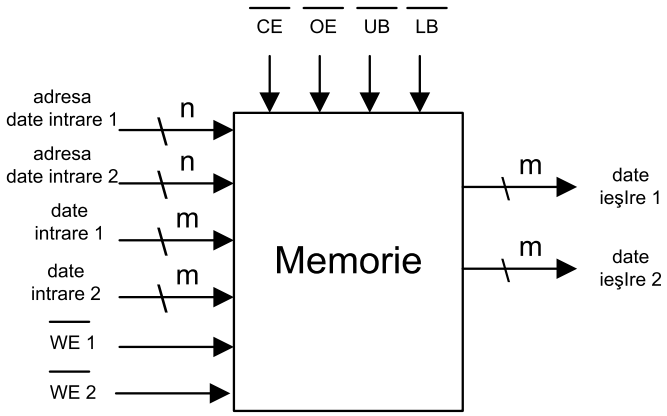


Figura 12 – Memoria RAM biport.

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

```
use ieee.std_logic_unsigned.all;

entity mem is
        port ( clk              : in std_logic;
               address_in_1     : in std_logic_vector(3 downto 0);
               address_in_2     : in std_logic_vector(3 downto 0);
               data_input_1     : in std_logic_vector(3 downto 0);
               data_input_2     : in std_logic_vector(3 downto 0);
               wren_1           : in std_logic;
               wren_2           : in std_logic;
               data_output_1 : out std_logic_vector(3 downto 0);
               data_output_2 : out std_logic_vector(3 downto 0)
             );
end mem;

architecture arch_mem of mem is

type mem is array (15 downto 0) of std_logic_vector(3 downto 0);
signal memory_array : mem;

begin

process(clk)
begin
      if (clk'event and clk = '1') then
         if(wren_1 = '0') then
            memory_array(conv_integer(address_in_1)) <= data_input_1;
         end if;
      end if;
end process;

process(clk)
begin
```

```
        if (clk'event and clk = '1') then
            if(wren_2 = '0') then
                memory_array(conv_integer(address_in_2)) <= data_input_2;
            end if;
        end if;
end process;

                data_output_1 <= memory_array(conv_integer(address_in_1));
                data_output_2 <= memory_array(conv_integer(address_in_2));

end arch_mem;
```

Figura 13 – Modul VHDL ce evidențiază memoria RAM biport.

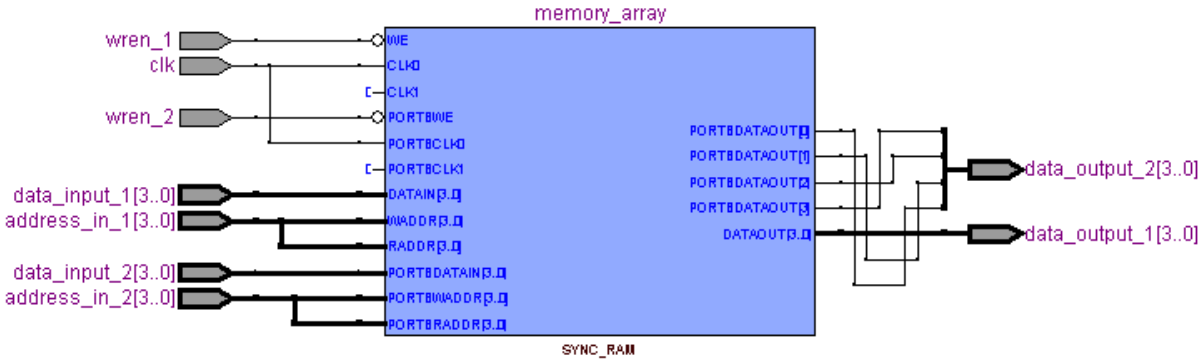
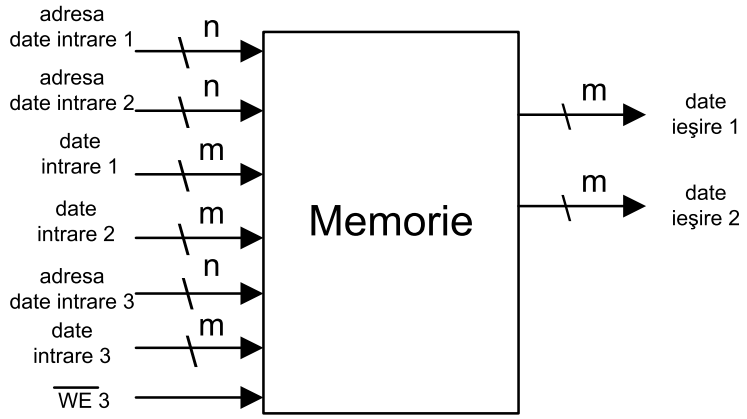


Figura 14 – Circuit sintetizat cu ajutorul modulului VHDL din figura 13.

7.4 Memoria RAM triport

În figura 15 este reprezentată memoria RAM triport. Acest tip de memorie, în limbaj de specialitate, se numește register file. Orice arhitectură modernă de microprocesor deține un register file. Un register file este alcătuit din 32 de registrii, fiecare registru putând avea o capacitate de 8, 16, 32 sau 64 de biți. În figura 16 este evidențiat modulul VHDL pentru memoria RAM triport. Acestă memorie funcționează în felul următor, atunci când semnalul $\overline{WE}3$ are valoarea logică 0, se spune că memoria este scrisă, adică datele de la intrarea de date 3 vor fi stocate în memorie în locația ce se găsește pe intrarea adresă date intrare 3. În același timp la ieșire pe porturile date ieșire 1 și date ieșire 2 vor fi furnizate datele stocate în memorie în locațiile ce se găsesc pe intrarea adresă date intrare 1 respectiv adresă date intrare

2. Atunci când $\overline{WE}3$ are valoarea logică 1, se spune că memoria este citită, adică la ieșire pe porturile date ieșire 1 și date ieșire 2 vor fi furnizate datele stocate în memorie în locațiile ce se găsesc pe intrarea adresă date intrare 1 respectiv adresă date intrare 2.



Figură 15 – Memorie RAM triport sau register file.

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

entity mem is
        port ( clk              : in std_logic;
               address_in_1     : in std_logic_vector(4 downto 0);
               address_in_2     : in std_logic_vector(4 downto 0);
               address_in_3     : in std_logic_vector(4 downto 0);
               data_input_1     : in std_logic_vector(31 downto 0);
               data_input_2     : in std_logic_vector(31 downto 0);
               data_input_3     : in std_logic_vector(31 downto 0);
               wren_3           : in std_logic;
               data_output_1 : out std_logic_vector(31 downto 0);
               data_output_2 : out std_logic_vector(31 downto 0)
             );
end mem;

architecture arch_mem of mem is
```

```
type mem is array (31 downto 0) of std_logic_vector(31 downto 0);
signal memory_array : mem;

begin

process(clk)
begin
        if (clk'event and clk = '1') then
            if(wren_3 = '0') then
                memory_array(conv_integer(address_in_3)) <= data_input_3;
            end if;
        end if;
end process;

                data_output_1 <= memory_array(conv_integer(address_in_1));
                data_output_2 <= memory_array(conv_integer(address_in_2));

end arch_mem;
```

Figura 16 – Modul VHDL ce evidențiază memoria RAM triport.

7.5 Memoria ROM

În figura 17 este reprezentat simbolul memoriei ROM. În figura 18 este evidențiat modulul VHDL pentru memoria ROM. Acestă memorie funcționează în felul următor, datele stocate în memoria ROM la o anumită adresă sunt oferite la ieşire, atunci când pe intrarea adresă există adresa respectivă. În figura 19 este reprezentat circuitul sintetizat cu ajutorul modulului VHDL din figura 18.

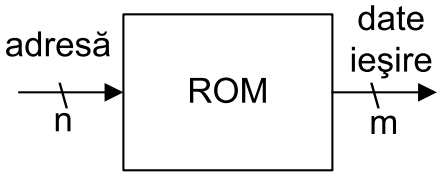


Figura 17 – Memoria ROM.

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

entity rom is
port ( address   : in std_logic_vector(3 downto 0);
         data_out : out std_logic_vector(3 downto 0)
       );
end rom;

architecture rom_arch of rom is
begin

process (address)
begin
        case address is
                when "0000"     => data_out <= "0011";
                when "0001"     => data_out <= "1110";
                when "0010"     => data_out <= "0100";
                when "0011"     => data_out <= "1010";
                when "0100"     => data_out <= "0011";
                when "0101"     => data_out <= "1110";
                when "0110"     => data_out <= "0100";
                when "0111"     => data_out <= "1100";
                when "1000"     => data_out <= "0010";
                when "1001"     => data_out <= "0011";
                when "1010"     => data_out <= "1110";
                when "1011"     => data_out <= "1100";
                when "1100"     => data_out <= "1010";
                when "1101"     => data_out <= "1011";
                when "1110"     => data_out <= "0110";
                when "1111"     => data_out <= "0100";
                when OTHERS => data_out <= (OTHERS => '0');
```

```
        end case;
end process;
end rom_arch;
```

Figura 18 – Modul VHDL ce implementează memoria ROM.

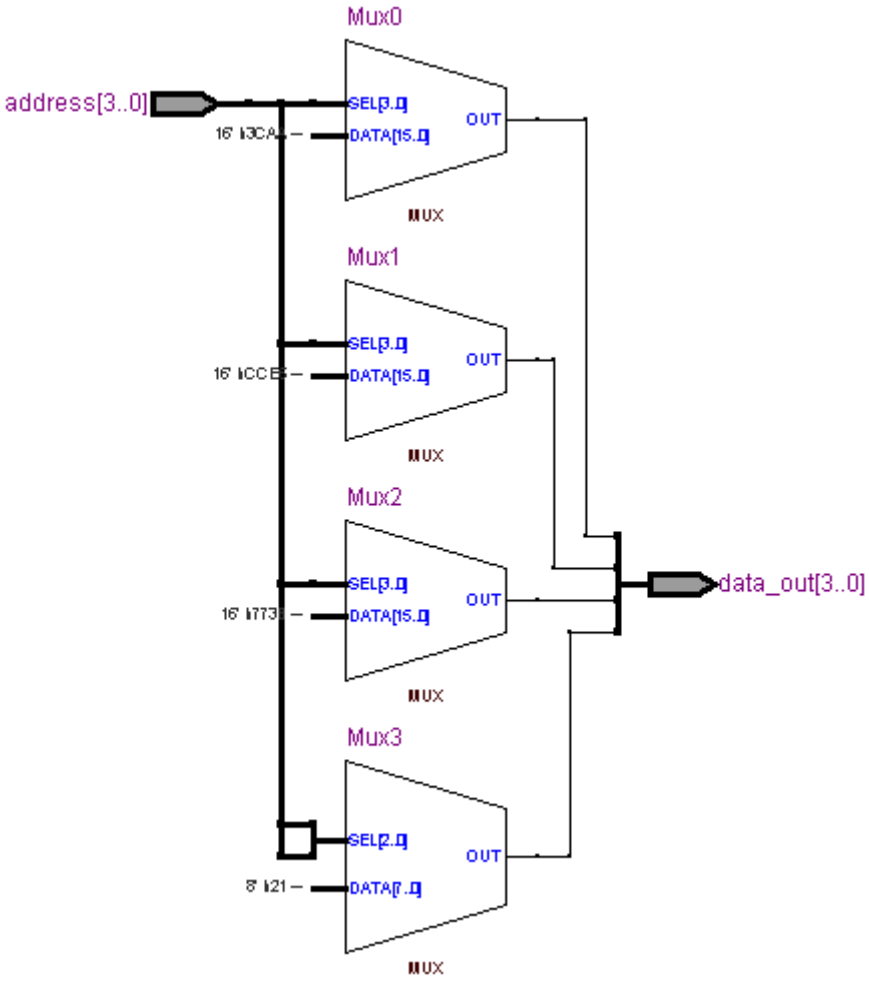


Figura 19 – Circuit sintetizat cu ajutorul modulului VHDL din figura 18.

Capitolul 8

Module parametrizabile

Module parametrizabile

Până în acest moment au fost create module VHDL unde semnalele de intrare sau ieșire aveau capacități fixe. Limbajul VHDL deține o caracteristică numită **generic** cu ajutorul căreia putem crea module parametrizabile unde semnale de intrare și ieșire au capacități variabile. În acest capitol vor fi parametrizate următoarele module VHDL: multiplexorul, registrul paralel, ALU-ul și memoria uniport. **Declarația** paralelă **generate** este foarte utilă în crearea de module VHDL unde este necesară utilizarea repetată a unei componente sau a unei expresii logice. De obicei în crearea de module VHDL parametrizabile caracteristica generic se folosește împreună cu declarația generate.

8.1 Parametrizarea modulelor VHDL

8.1.1 Caracteristica generic și generic map

În figura 1 este evidențiată utilizarea caracteristicii generic. După cum se observă acestă caracteristică se definește în cadrul unei entități. Caracteristica generic poate cuprinde unul sau mai mulți parametrii întregi. În figura 2 este evidențiată instanțierea unei componente parametrizate. Dacă se dorește folosirea unei alte valori pentru parametrul definit în cadrul caracteristicii generic, se va introduce caracteristica **generic map** împreună cu valoarea dorită a parametrului, în momentul instanțierii componentei respective.

```
ENTITY numele_interfeței is
   GENERIC (WIDTH : INTEGER := N);
   PORT (
            numele_portului : modul_semnalului tipul_semnalului;
            numele_portului : modul_semnalului tipul_semnalului;
            numele_portului : modul_semnalului tipul_semnalului;
        ...);
END numele_interfeței;
```

Figura 1 – Cacarteristica generic.

```
-- declararea unei componente

COMPONENT numele_componentei
   GENERIC (WIDTH : INTEGER);
   PORT (
            numele_semnalului : modul_semnalului tipul_semnalului;
            numele_semnalului : modul_semnalului tipul_semnalului;
            numele_semnalului : modul_semnalului tipul_semnalului;
        ...);
END numele_componentei;

-- utilizarea unei componente

numele_instanței : numele_componentei GENERIC MAP (M)
                    PORT MAP (
                            numele_semnalului_componentei => numele_semnalului_modulului,
                            numele_semnalului_componentei => numele_semnalului_modulului,
                                                    .
                                                    .
                                                    .
                            numele_semnalului_componentei => numele_semnalului_modulului
                                                    );
```

Figura 2 – Caracteristica generic map.

8.1.1.1 Multiplexor

În figura 3 este evidențiată organizarea unui multiplexor parametrizat. Capacitatea implicită a intrărilor și ieșirilor este de 8 biți. Cu ajutorul multiplexorului 2 la 1 din figura 3 se poate crea un multiplexor 4 la 1 cu o capacitate mai mare a intrărilor și ieșirilor. Multiplexorul 4 la 1 este evidențiat în figura 4, având o capacitate implicită de 16 biți.

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

```
entity mux21 is
        generic (width : integer := 8);
        port ( a, b : in std_logic_vector(width-1 downto 0);
                sel  : in std_logic;
                c    : out std_logic_vector(width-1 downto 0) );
end mux21;

architecture mux21_arch of mux21 is
begin

c <= a when sel = '1' else b;

end mux21_arch;
```

Figura 3 – Multiplexor 2 la 1 parametrizat cu capacitate implicită de 8 biți.

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

entity mux41 is
        generic (width : integer := 16);
        port ( a, b, c, d : in std_logic_vector(width-1 downto 0);
                sel       : in std_logic_vector(1 downto 0);
                e         : out std_logic_vector(width-1 downto 0) );
end mux41;

architecture mux41_arch of mux41 is

component mux21
        generic (width : integer);
        port ( a, b : in std_logic_vector(width-1 downto 0);
                sel  : in std_logic;
                c    : out std_logic_vector(width-1 downto 0) );
end component;

signal intermed1, intermed2 : std_logic_vector(width-1 downto 0);

begin

mux21_stage0 : mux21 generic map (16) port map (a, b, sel(0), intermed1);
mux21_stage1 : mux21 generic map (16) port map (c, d, sel(0), intermed2);
mux21_stage2 : mux21 generic map (16) port map (intermed1, intermed2, sel(1), e);

end mux41_arch;
```

Figura 4 – Multiplexor 4 la 1 parametrizat cu capacitate implicită de 16 biți alcătuit cu 3 multiplexoare 2 la 1.

8.1.1.2 Registrul paralel

În figura 5 este evidențiată organizarea unui registru paralel parametrizat cu capacitate implicită de 16 biți.

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

entity reg_circ is
        generic (width : integer := 16);
         port (
                d                  : in std_logic_vector(width-1 downto 0);
                clk, rst, enable : in std_logic;
                q                  : out std_logic_vector(width-1 downto 0) );
end reg_circ;

architecture reg_circ_arch of reg_circ is
begin

process (clk, rst)
begin
      if (rst = '1') then
              q <= (OTHERS => '0');
      elsif (clk'event and clk = '1') then
            if (enable = '1') then
                q <= d;
            end if;
      end if;
end process;

end reg_circ_arch;
```

Figura 5 – Registrul paralel parametrizat cu capacitate implicită de 16 biți.

8.1.1.3 ALU

În figura 6 este evidențiată organizarea unui ALU parametrizat cu capacitate implicită de 16 biți.

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

entity ALU is
        generic (width : integer := 16);
        port ( A, B : in std_logic_vector(width-1 downto 0);
               sel  : in std_logic_vector(2 downto 0);
               S    : out std_logic_vector(width-1 downto 0);
               Egal : out std_logic );
end ALU;

architecture ALU_arch of ALU is

begin
process(A, B, sel)
begin
case sel is
      when "000" => S <= A + B;
      when "001" => S <= A + not(B) + '1';
      when "010" => S <= A or B;
      when "011" => S <= A and B;
      when "100" => S <= conv_std_logic_vector(shl(conv_unsigned(conv_integer(A), width), conv_unsigned(conv_integer(B), width)), width);
      when "101" => S <= conv_std_logic_vector(shr(conv_unsigned(conv_integer(A), width), conv_unsigned(conv_integer(B), width)), width);
      when "110" => S <= A nor B;
```

```
      when OTHERS =>
            if (A < B) then
                S(width-1 downto 1) <= (OTHERS => '0');
                   S(0) <= '1';
            else S <= (OTHERS => '0');
            end if;
end case;
end process;

Egal <= '1' when A = B else '0';

end ALU_arch;
```

Figura 6 – ALU parametrizat cu capacitate implicită de 16 biți.

8.1.1.4 Memorie uniport

În figura 7 este evidențiată organizarea unei memorii uniport parametrizate cu capacitate implicită de 32 rânduri și 16 de coloane.

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

entity mem is
          generic (R : integer := 5; C : integer := 16);
          port ( clk              : in std_logic;
                 address          : in std_logic_vector(R-1 downto 0);
                 data_input       : in std_logic_vector(C-1 downto 0);
                 wren             : in std_logic;
                 data_output : out std_logic_vector(C-1 downto 0)
                );
end mem;
```

```
architecture arch_mem of mem is

type mem is array (2**R-1 downto 0) of std_logic_vector(C-1 downto 0);
signal memory_array : mem;

begin

process(clk)

begin
        if (clk'event and clk = '1') then
            if(wren = '0') then
                memory_array(conv_integer(address)) <= data_input;
            end if;
        end if;
end process;

                data_output <= memory_array(conv_integer(address));

end arch_mem;
```

Figura 7 – Memorie uniport parametrizată cu 32 de rânduri și 16 coloane.

8.2 Declarația generate

Există două variante ale declarației generate: **for generate** și i**f generate**. În figura 8 este definită declarația for generate, iar în figura 9 este definită declarația if generate. Cele două variante ale declarației generate se pot utiliza pentru a crea circuite unde este necesară instanțierea repetată a unei componente sau atunci când se dorește utilizarea repetată a unor porți logice.

Declarațiile if și for generate trebuie utilizare cu mare grijă deoarece pot genera un număr mare de circuite sau porți logice fără să fie nevoie de un așa număr mare.

```
numele_declarației_generate:
FOR index IN mulțime_de_valori GENERATE
        declarații;
        declarații;
        declarații;
            .
            .
            .
        declarații;
END GENERATE;
```

Figura 8 – Declarația for generate.

```
numele_declarației_generate:
IF expresie_logică GENERATE
        declarații;
        declarații;
        declarații;
            .
            .
            .
        declarații;
END GENERATE;
```

Figura 9 – Declarația if generate.

În figura 10 este evidențiată organizarea unui sumator pe 16 biți. Circuitul este asamblat cu ajutorul declarației for generate.

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

```
entity adder is
        generic(width : integer := 16);
        port(
                Cin    : std_logic;
                A, B  : in std_logic_vector(width-1 downto 0);
                S      : out std_logic_vector(width-1 downto 0);
                Cout : out std_logic
              );
end adder;

architecture adder_arch of adder is

component fulladder
        port ( A, B, Cin : in std_logic;
                S, Cout     : out std_logic);
end component;

signal carry : std_logic_vector(0 to 16);

begin
carry(0) <= Cin;

generate_fulladder_component:
for i in 0 to width-1 generate
fulladder_stage : fulladder port map (A(i), B(i), carry(i), S(i), carry(i+1));
end generate;

Cout <= carry(16);

end adder_arch;
```

Figura 10 – Utilizarea declarației for generate.

În figura 11 este evidențiată organizarea unui funcții logice oarecare. Circuitul este asamblat cu ajutorul declarațiolor if generate și for generate. Circuitul sintetizat cu ajutorul modulului VHDL din figura 11 este evidențiat în figura 12.

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

entity random_circ is
        generic(width : integer := 4);
         port( data_in    : in std_logic_vector(width-1 downto 0);
                data_out : out std_logic );
end random_circ;

architecture random_circ_arch of random_circ is

signal intermed : std_logic_vector(3 downto 0);

begin

generate_random_circ:
for i in 0 to width-1 generate

generate_not_gate:
if i = 0 generate
intermed(i) <= not data_in(i);
end generate;

generate_and_signals:
if i /= 0 generate
intermed(i) <= data_in(i) and intermed(i-1);
end generate;

generate_or_signals:
```

```
if i = width-1 generate
data_out <= data_in(0) or intermed(i);
end generate;

end generate;

end random_circ_arch;
```

Figură 11 – Implementarea unei funcții logice oarecare cu ajutorul declarațiilor if și for generate.

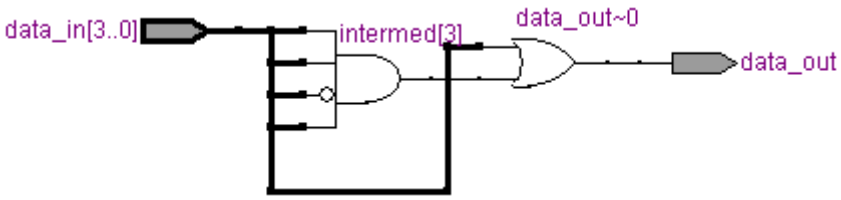


Figura 12 – Circuit sintetizat cu ajutorul modulului VHDL din figura 11.

Capitolul 9

Implementarea unui procesor MIPS single-cycle pe 32 de biți

Implementarea unui procesor MIPS single-cycle pe 32 de biți

Microprocesoarele și microcontrolerele reprezintă sisteme digitale complexe. Toate tipurile de procesoare sunt organizate cu ajutorul a două componente standard: componenta datapath (calea de date) și componenta control (unitatea de control). Limbajul pe care un procesor îl înțelege se numește arhitectură. Limbajul poate avea mai multe dialecte sau în limbaj tehnic o arhitectură poate avea mai multe microarhitecturi, adică o arhitectură poate fi implementată în mai multe feluri, fiecare microarhitectură având performanțele ei.

9.1 Datapath și control

În figura 1 este evidențiată organizarea standard a unui procesor oarecare. Părțile componente sunt datapath (calea de date) și control (unitatea de control). Datapath reprezintă pentru un procesor mușchii, iar control reprezintă creierul. Funcționarea componentei control este evidențiată în limbaj VHDL printr-o mașină cu stări finite. Componenta datapath poate cuprinde mai multe subcomponente cum ar fi: ALU, registru, circuit de deplasare logică la stânga, sumatoare, multiplexoare și memorii sau orice alt circuit logice care nu reprezintă o mașină cu stări finite.

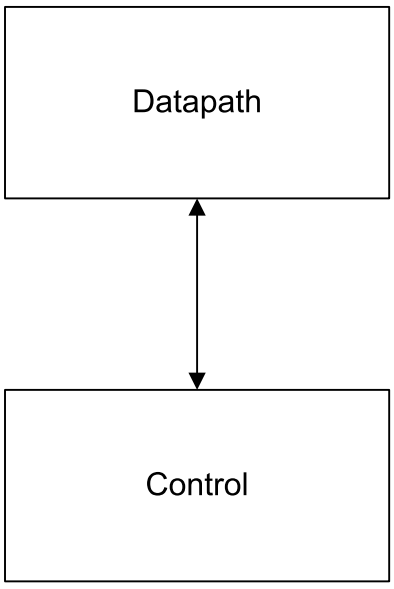


Figura 1 – Organizarea standard a unui procesor oarecare: component datapath și componenta control.

Trebuie precizat că orice alt sistem digital complex poate fi organizat în așa fel încât să fie altcătuit dintr-o componentă datapath și o componentă control.

Mai sus a fost evidențiat faptul că memoria este deținută de componenta datapath. Memoria unui procesor poate fi de patru feluri: memoria pentru date, memoria pentru instrucțiuni (instrucțiunile reprezintă cuvintele limbajului nostru), registrii și counterul pentru instrucțiuni (program counter sau PC).

O microarhitectură poate avea o memorie comună pentru date și instrucțiuni, această implementare purtând numele de microarhitectura Princeton, sau poate avea două memorii separate, una pentru date iar cealaltă pentru instrucțiuni, această implementare purtând numele de microarhitectura Harvard. Alcătuirea celor două microarhitecturi este prezentată în subcapitolul următor

9.2 Microarhitectura Harvard și microarhitectura Princeton

În figurile 2 și 3 este evidențiată organizarea microarhitecturilor Harvard și Princeton. Arhitectura Princeton este bine cunoscută sunt numele de arhitectura von Neumann.

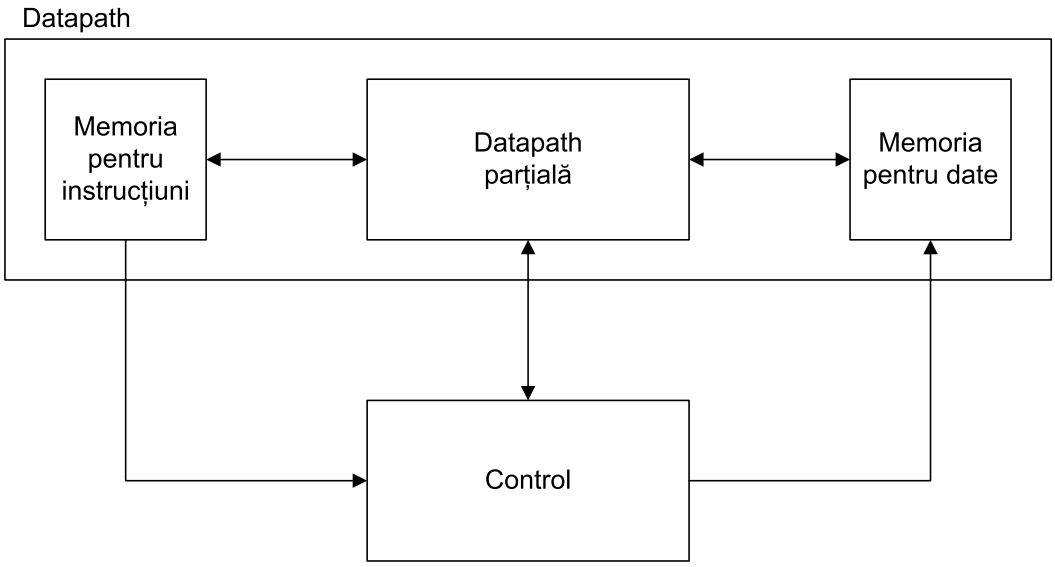


Figura 2 – Organizarea microarhitecturii Hardvard.

Trebuie precizat că în zilele noastre procesoarele sunt implementate cu ajutorul microarhitecturii Princeton. Privind totul dintr-o perspectivă pedagogică arhitectura Harvard este ușor de implementat și de înțeles de cădre viitori ingineri electroniști sau programatori.

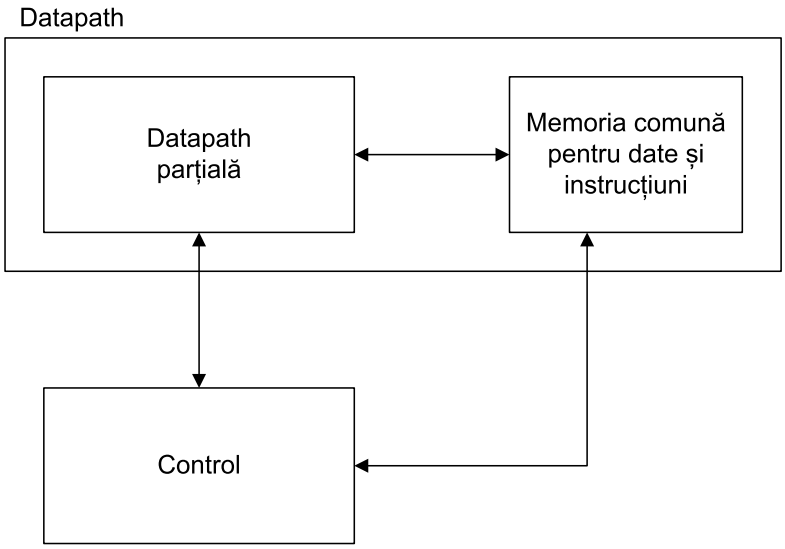


Figura 3 – Organizarea microarhitecturii Princeton (von Neumann).

9.3 Microarhitectura MIPS single-cycle pe 32 de biți

Atunci când se dorește implementarea în hardware a unui procesor, inginerul proiectant trebuie "să respecte termenii din contractul semnat" între arhitectură și microarhitectură. În acest subcapitol va fi evidențiată implementarea unei microarhitecturi MIPS single-cycle pe 32 de biți. Microarhitectura ce va fi implementă va fi de tip Harvard (memorie separată pentru date și instrucțiuni).

În cadrul unei arhitecturi MIPS pe 32 de biți instrucțiunile ce trebuie executate au o lungime de 32 de biți, iar datele ce trebuie stocate în memoria pentru date la rândul lor au o lungime de 32 de biți. Microarhitectura ce va fi implementată este single-cycle, acest lucru însemnând că instrucțiunile vor fi executate cu o viteză de o instrucțiune per perioadă. Perioada de execuție se notează cu T_s, inversul acestei perioade reprezintă frecvența procesorului, $f_s = \frac{1}{T_s}$. Microarhitectura MIPS single-cycle pe 32 de biți va fi alcătuită din două componente: componenta datapath și componenta control.

În figura 4 este evidențiată organizarea internă a componentei datapath. După cum se observă acestă componentă este alcătuită din circuite digitale create în subcapitolele anterioare, cum ar fi: registru, ALU, sumatoare, multiplexoare, memorii (pentru date și pentru instrucțiuni), "dosarul cu registre" sau register file-ul și circuit de deplasare logică la stânga.

În figura 5 este evidențiată componenta control. Acestă componentă asigură funcționarea corectă a componentei datapath și bineînțeles a întregului procesor, prin

furnizarea unor semnale de control. pentru o arhitectură single-cycle componenta control va fi un circuit logic combinațional.

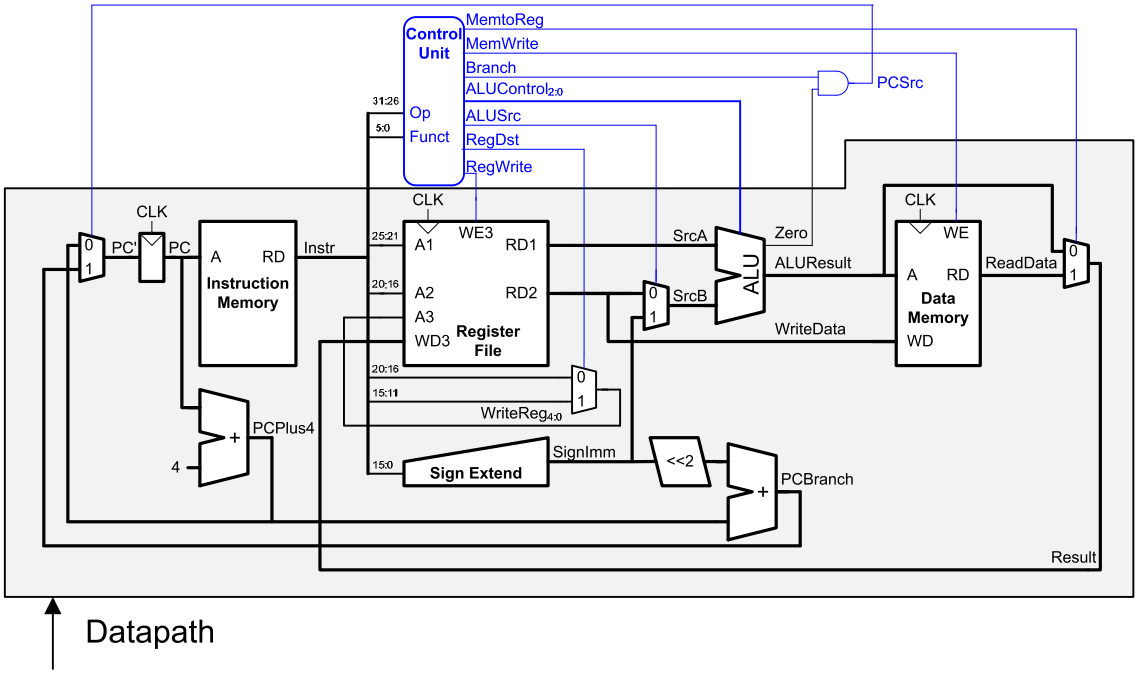


Figura 4 – Componenta datapath. Figura este preluată din Digital Design and Computer Architecture – Harris and Harris.

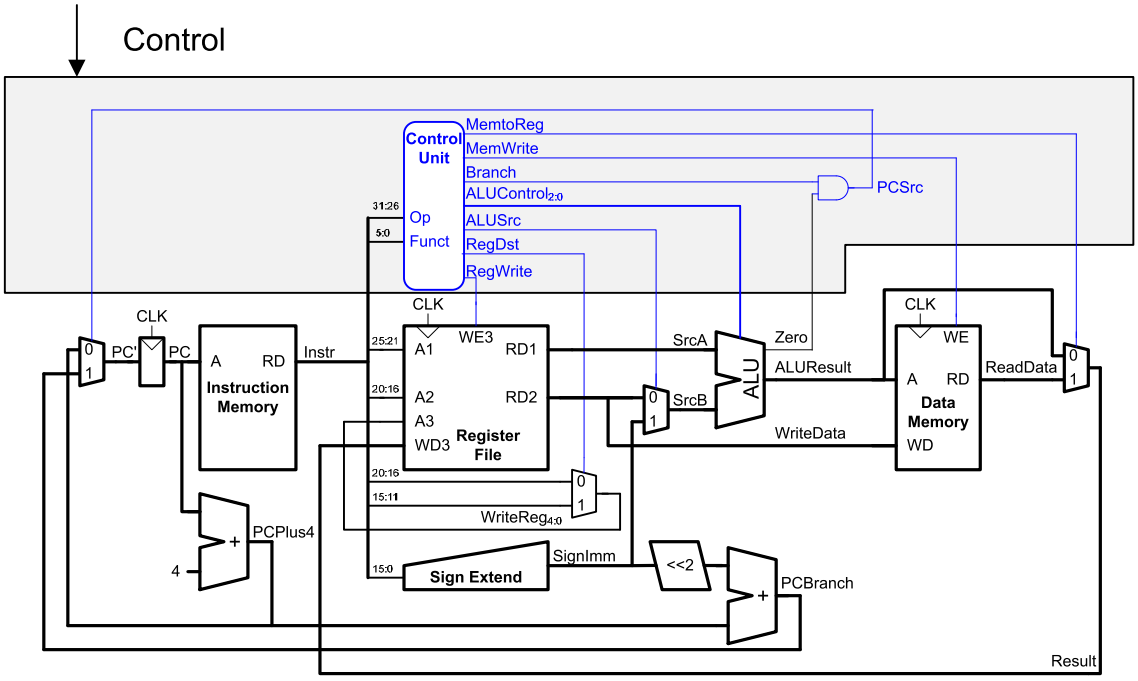


Figura 5 – Componenta control. Figura este preluată din Digital Design and Computer Architecture – Harris and Harris.

Anexa 1

Crearea unei librării

Anexa 1 – Crearea unei librării

Librăriile sunt alcătuite din pachete. Un pachet este alcătuit din două părți: zona de declarări și corpul pachetului. Corpul pachetului reprezintă o parte opțională, în care se poate defini comportamentul unor funcții, cum ar fi funcțiile de conversie pentru diferite tipuri de date. În partea declarativă se pot declara tipuri de date, constante, semnale globale, componente, atribute, funcții și proceduri. Forma generală a unui pachet este evidențiată în figura 1.

```
-- Partea declarativă
PACKAGE numele_pachetului is
        declararea tipurilor de date
        declararea constantelor
        declararea semnalelor globale
        declararea componentelor
        declararea atributelor
        declararea funcțiilor
        declararea procedurilor
END numele_pachetului;

-- Corpul pachetului
PACKAGE BODY numele_pachetului is
        descrierea comportamentului unei funcții
END numele_pachetului;
```

Figura 1 – Forma generală a unui pachet.

Se va crea o librărie ce va cuprinde un pachet în care se vor defini un tip de date, o constantă, un semnal global și o componentă. Componenta descrie un sumator complet, comportamentul circuitului fiind evidențiat în figura 2.

```
library ieee; use ieee.std_logic_1164.all;
entity fulladder is
        port( Cin, x, y : in std_logic; s, Cout : out std_logic);
end fulladder;
architecture arch_fulladder of fulladder is
begin
s<=x xor y xor Cin; Cout<=(x and y) or (x and Cin) or (y and Cin);
end arch_fulladder;
```

Figura 2 – Modul VHDL ce descrie comportamentul unui sumator complet.

Sumatorul se va salva cu numele de fulladder.vhd. Pachetul este definit în figura 3 şi va purta numele my_pack. Pachetul se va salva cu numele my_pack.vhd. În acest momentul pachetul devine o librărie şi va putea fi folosit într-un modul VHDL. Modul în care se utilizează acest pachet (acestă librărie) este evidenţiat în figura 4. Modulul este salvat cu numele adder.vhd.

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

package my_pack is

    -- declararea unui tip de date
    type my_byte is array (7 downto 0) of std_logic;

    -- declararea unei constante
    constant all_ones : std_logic_vector(7 downto 0) := (others => '1');

    -- declararea unui semnal global
    signal internal_signal : std_logic_vector(7 downto 0);
```

```
    -- declararea unei componente
    component fulladder
            port( Cin, x, y : in std_logic; s, Cout : out std_logic);
    end component;

end my_pack;
```

Figura 3 – Pachetul my_pack.

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

entity adder is
          port(
                Cin    : in std_logic;   x, y    : in std_logic_vector(3 downto 0);
                Cout  : out std_logic; s        : out std_logic_vector(3 downto 0)
               );
end adder;

architecture arch_adder of adder is

signal c : std_logic_vector(1 to 3);

begin

stage0 : fulladder port map (Cin , x(0), y(0), s(0), c(1));
stage1 : fulladder port map (c(1), x(1), y(1), s(1), c(2));
```

```
stage2 : fulladder port map (c(2), x(2), y(2), s(2), c(3));
stage3 : fulladder port map (c(3), x(3), y(3), s(3), Cout);

end arch_adder;
```

Figura 4 – Modul de utilizare al librăriei my_pack.

În figura 5 este evidențiat circuitul reprezentat în figura 4. În figura de mai jos se observă instanțierea a patru module. Circuitul a fost creat utilizând stilul structural, după cum se observă în figura 4.

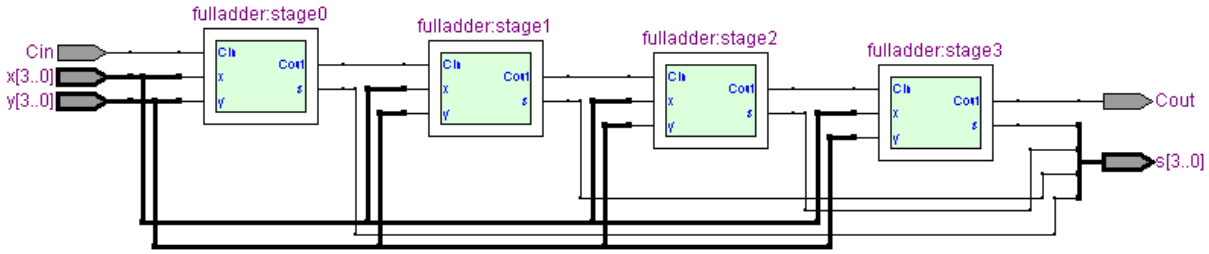


Figura 5 – Descrierea structurală a unui sumator pe 4 biți.

Anexa 2

Cuvinte cheie VHDL

Anexa 2 – Cuvinte cheie VHDL

VHDL 87 – Cuvinte cheie:		
ABS	LOOP	WHILE
ACCESS	MAP	WITH
AFTER	MOD	XOR
ALIAS	NAND	
ALL	NEW	VHDL 93 – Cuvinte cheie:
AND	NEXT	
ARCHITECTURE	NOR	GROUP
ARRAY	NOT	IMPURE
ASSERT	NULL	INERTIAL
ATTRIBUTE	OF	LITERAL
BEGIN	ON	POSTPONED
BLOCK	OPEN	PURE
BODY	OR	REJECT
BUFFER	OTHERS	ROL
BUS	OUT	ROR
CASE	PACKAGE	SHARED
COMPONENT	PORT	SLA
CONFIGURATION	PROCEDURE	SLL
CONSTANT	PROCESS	SRA
DISCONNECT	RANGE	SRL
DOWNTO	RECORD	UNAFFECTED
ELSE	REGISTER	XNOR
END	REM	
ENTITY	REPORT	
EXIT	RETURN	
FILE	SELECT	
FOR	SEVERITY	
FUNCTION	SIGNAL	
GENERATE	SUBTYPE	
GENERIC	THEN	
GUARDED	TO	
IF	TRANSPORT	
IN	TYPE	
INOUT	UNITS	
IS	UNTIL	
LABEL	USE	
LIBRARY	VARIABLE	
LINKAGE	WAIT	
	WHEN	

Anexa 3

Reprezentarea numerelor binare

Anexa 3 – Reprezentarea numerelor binare

Valorile binare sunt: 0 și 1. Cu ajutorul acestor valori se pot crea numere binare reprezentate prin semnale pe mai mulți biți. Există semnale pe 128, 64, 32, 16, 8, 4, 2 sau 1 bit. În figura 1 sunt evidențiate semnalele obișnuite dintr-un microprocesor.

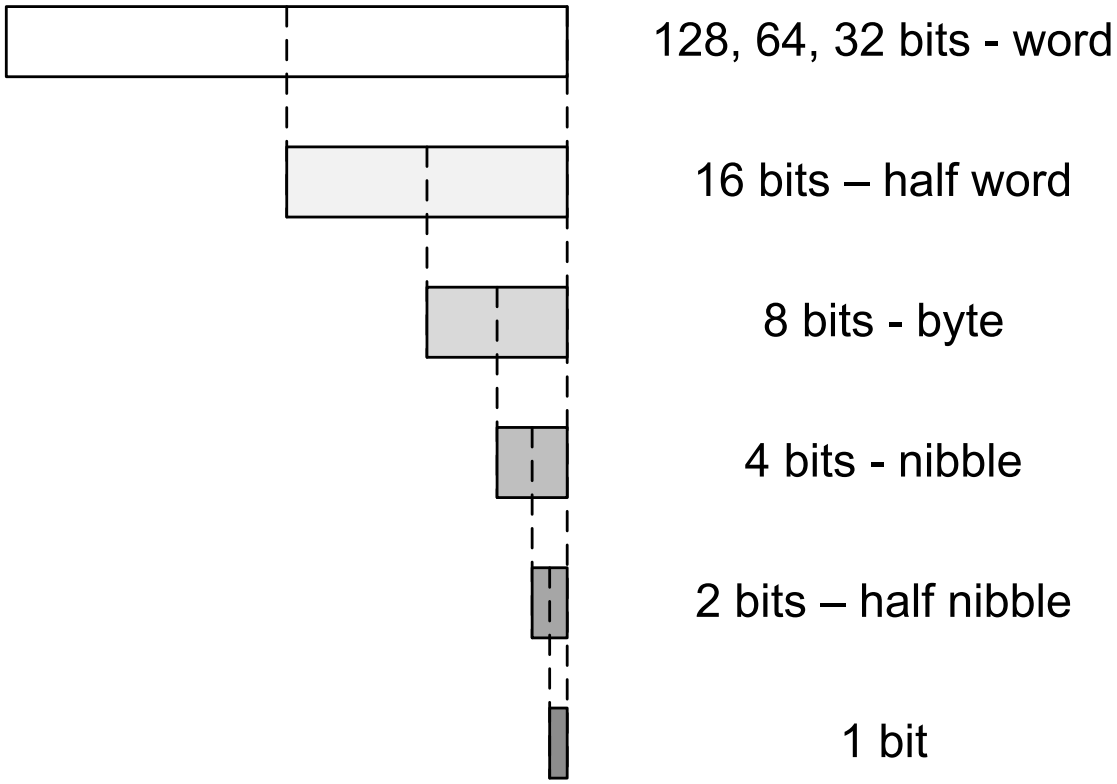


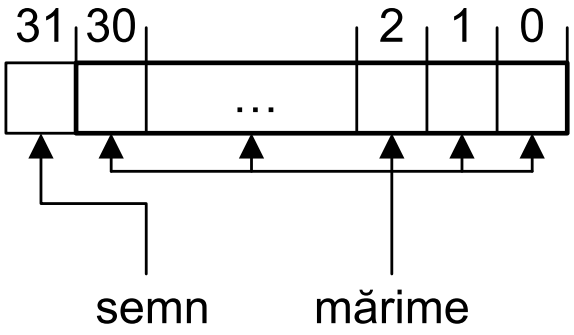
Figura 1 – Semnale obișnuite într-un microprocesor.

Vom lua ca exemplu un semnal pe 32 de biți (word). Pe cei 32 de biți putem crea numere binare evidențiate prin diferite reprezentări binare. Există trei reprezentări binare importante: sign-magnitude (semn și mărime), 1 complement (se citește first complement sau complement față de 1) și 2 complement (se citește second complement sau complement față de 2).

A.3.1. Sign-magnitude

Reprezentarea sign-magnitude este evidențiată în figura 2. Cei 32 de biți au următoarea însemnătate: primul bit (cel mai semnificativ bit) va reprezenta semnul, restul

biților vor reprezenta mărimea. Dacă bitul de semn va avea valoarea logică 1 atunci numărul va fi întreg negativ, dacă bitul de semn va avea valoarea logică 0 atunci numărul va fi întreg pozitiv.



Figură 2 – Reprezentarea binară sign-magnitude.

În figura 3 sunt date câteva exemple de numere binare evidențiate prin reprezentarea sign-magnitude. Numerele binare din figura de mai jos sunt organizate în 8 grupuri a câte 4 biți. S-a optat pentru acestă organizare pentru o vizibilitate mai bună a numerelor binare.

$$0000\_0000\_0000\_0000\_0000\_0000\_0000\_0001_2 = +1_{10}$$
$$0000\_0000\_0000\_0000\_0000\_0000\_0000\_0000_2 = +0_{10}$$
$$1000\_0000\_0000\_0000\_0000\_0000\_0000\_0000_2 = -0_{10}$$
$$1000\_0000\_0000\_0000\_0000\_0000\_0000\_0010_2 = -2_{10}$$

Figura 3 – Exemple de numere binare evidențiate prin reprezentarea sign-manitude.

A.3.2. 1 complement

Reprezentarea 1 complement este evidențiată în figura 4 printr-un exemplu. Putem obține un număr binar negativ în reprezentare 1 complement negând toți biții unui număr binar negativ în reprezentare 1 complement. Procesul este reversibil.

$$0000\_0000\_0000\_0000\_0000\_0000\_0000\_0001_2 = +1_{10}$$
$$1111\_1111\_1111\_1111\_1111\_1111\_1111\_1110_2 = -1_{10}$$
$$0000\_0000\_0000\_0000\_0000\_0000\_0000\_0000_2 = +0_{10}$$
$$1111\_1111\_1111\_1111\_1111\_1111\_1111\_1111_2 = -0_{10}$$

Figura 4 – Exemple de numere binare evidențiate prin reprezentarea 1 complement.

A.3.3. 2 complement

În momentul de față reprezentarea 2 complement este utilizată de către toate microprocesoarele comerciale. Acestă reprezentare este evidențiată în figura 5 printr-un exemplu. Putem obține un număr binar negativ în reprezentare 2 complement negând toți biții unui număr binar pozitiv în reprezentare 2 complement la care apoi adunăm valoarea logică 1. Procesul este reversibil.

$$0000\_0000\_0000\_0000\_0000\_0000\_0000\_0001_2 = +1_{10}$$
$$1111\_1111\_1111\_1111\_1111\_1111\_1111\_1111_2 = -1_{10}$$
$$0000\_0000\_0000\_0000\_0000\_0000\_0000\_0000_2 = +0_{10}$$

Figura 5 – Exemple de numere binare evidențiate prin reprezentarea 2 complement.

A.3.4. Conversia numerelor binare în numere hexazecimale

În figura 6 este reprezentată conversia numerelor binare în numere hexazecimale. În figura 7 este evidențiată transformarea numerelor binare în numere hexazecimale printr-un exemplu.

Număr binar pe 4 biți (baza 2)	Număr hexazecimal (baza 16)
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D

1110	E
1111	F

Figura 6 – Conversia numerelor binare pe 4 biți în reprezentare hexazecimală.

1111_1111_1111_1111_1111_0000_0000_1101

F_F_F_F_F_0_0_D

Figura 7 – Conversia numerelor binare pe 32 de biți în reprezentare hexazecimală.

Anexa 4

Buffer-ul cu 3 stări

Anexa 4 – Buffer-ul cu 3 stări

În figura 1 este evidențiat tabelul de adevăr pentru un buffer cu 3 stări. În figura 2 este evidențiat simbolul acestui tip de buffer. După cum se observă acest circuit are o intrare de date, o ieșire de date și o intrare de enable. Dacă intrarea de enable are valoarea logică 1 atunci ieșirea de date este conectată cu intrarea de date. Dacă intrarea de enable are valoarea logică 0 atunci ieșirea de date nu va fi conectată cu intrarea de date, între intrare și ieșire existând un gol de circuit, adică ieșirea de date va lua valoarea logică Z, numită și HIGH Z, sau impedanță mare. De obicei tensiunea valorii logice Z se află în intervalul de tensiuni 0V, însemnând valoarea logică 0 și U_{max}, însemând valoarea logică 1. În funcție de tehnologia de fabricație U_{max} poate avea următoarele valori: 5V, 3.6V, 1V.

În figura 3 este prezentat modulul VHDL ce evidențiază comportamentul buffer-ului cu 3 stări.

ena	data_in	data_out
0	0	Z
0	1	Z
1	0	0
1	1	1

Figura 1 – Tabel de adevăr pentru un buffer cu 3 stări.

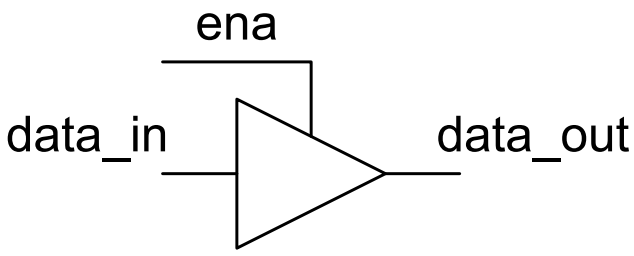


Figura 2 – Simbolul buffer-ului cu 3 stări.

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

```
use ieee.std_logic_unsigned.all;

entity tristate_buffer is
        port( ena      : in std_logic;
              data_in  : in std_logic_vector(7 downto 0);
              data_out : out std_logic_vector(7 downto 0));
end tristate_buffer;

architecture tristate_buffer_arch of tristate_buffer is

begin
process(data_in, ena)
begin
      if (ena = '1') then
          data_out <= data_in;
      else
          data_out <= (others => 'Z');
      end if;
end process;

end tristate_buffer_arch;
```

Figura 3 – Modul VHDL ce evidențiază comportamentul buffer-ului cu 3 stări.

Anexa 5

Maparea porturilor unei componente

Anexa 5 – Maparea porturilor unei componente

Porturile unei componente pot fi mapate în două feluri, prin: mapare poziţională sau mapare nominalizată. În figura 1 este evidenţiată utilizarea mapării nominalizate, iar în figura 2 este evidenţiată utilizarea mapării poziţionale. Trebuie precizat că maparea poziţională este uşor de efectuat, dar semnalele pot fi mapate greşit cu destul de multă uşurinţă. Maparea nominalizată este efectuată într-un timp mai lung, însă acestă mapare se va face fără erori neprevăzute.

Este bine ca maparea fiecărui port să fie efectuată pe o linie de cod diferită, pentru ca modulul VHDL să fie uşor de debug-uit, iar sintetizatorul să indice clar linia de cod unde a fost efectuată o mapare greşită.

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

entity adder is
        port( Cin    : std_logic;
              A, B  : in std_logic_vector(3 downto 0);
              S      : out std_logic_vector(3 downto 0);
              Cout : out std_logic );
end adder;

architecture adder_arch of adder is

component fulladder
        port ( A, B, Cin  : in std_logic;
               S, Cout      : out std_logic);
end component;

signal carry : std_logic_vector(0 to 4);
```

```
begin

carry(0) <= Cin;

stage0 : fulladder port map ( A => A(0),
                              B => B(0),
                              Cin => carry(0),
                              S => S(0),
                              Cout => carry(1) );
stage1 : fulladder port map ( A => A(1),
                              B => B(1),
                              Cin => carry(1),
                              S => S(1),
                              Cout => carry(2) );
stage2 : fulladder port map ( A => A(2),
                              B => B(2),
                              Cin => carry(2),
                              S => S(2),
                              Cout => carry(3) );
stage3 : fulladder port map ( A => A(3),
                              B => B(3),
                              Cin => carry(3),
                              S => S(3),
                              Cout => carry(4) );
Cout <= carry(4)  ;

end adder_arch;
```

Figura 1 – Maparea nominalizată.

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

```
entity adder is
        port( Cin    : std_logic;
              A, B  : in std_logic_vector(3 downto 0);
              S      : out std_logic_vector(3 downto 0);
              Cout : out std_logic );
end adder;

architecture adder_arch of adder is

component fulladder
        port ( A, B, Cin  : in std_logic;
              S, Cout      : out std_logic);
end component;

signal carry : std_logic_vector(0 to 4);

begin

carry(0) <= Cin;

stage0 : fulladder port map (A(0), B(0), carry(0), S(0), carry(1) );
stage1 : fulladder port map (A(1), B(1), carry(1), S(1), carry(2) );
stage2 : fulladder port map (A(2), B(2), carry(2), S(2), carry(3) );
stage3 : fulladder port map (A(3), B(3), carry(3), S(3), carry(4) );

Cout <= carry(4)  ;

end adder_arch;
```

Figura 2 – Maparea poziţională.

Bibliografie

Bibliografie

1. Brown S., Vranesic Z., *Fundamentals of Digital Logic with VHDL*, 2nd ed., McGraw Hill, 2005.

2. Chu P., *RTL Hardware Design using VHDL*, John Wiley & Sons, 2000.

3. Ciletti M., *Advanced Digital Design with the Verilog HDL*, Prentice Hall, 2003.

4. Harris D., Harris S., *Digital Design and Computer Architecture*, Morgan Kaufmann, 2007.

5. Katz, R., *Modern Logic Design*, 2nd ed., Addison-Wesley, 2004.

6. Mano M., Ciletti M., *Digital Design*, Prentice Hall, 4th ed., 2006.

7. Patterson D., Hennessy J., *Computer Organization and Design*, Morgan Kaufmann, 4th ed., 2009.

8. Pedroni V., *Circuit Design with VHDL*, MIT Press, 2004.

9. Roth C., *Digital System Design with VHDL*, PWS Publishing Company, 1998.

10. Wakerly J., *Digital Design: Principles and Practices,* 3rd ed., Prentice Hall, 2000.

11. www.altera.com

12. http://www.cs.umbc.edu/portal/help/VHDL/stdpkg.html

13. http://esd.cs.ucr.edu/labs/tutorial

14. http://micro.magnet.fsu.edu/chipshots/mips/index.html

15. http://www-inst.eecs.berkeley.edu/~cs150/sp04/Calendar.htm

16. www.mips.com

17. www.elsevier.com