C U P R I N S

1

Proiectarea circuitelor secvențiale sincrone

1. Prezentare teoretică

În cadrul acestei lucrări de laborator se va prezenta metodologia de proiectare a unui circuit secvențial sincron. Circuitul final este obținut cu ajutorul diagramelor Karnough-Vength iar implementarea sa este realizată cu ajutorul utilitarului ***Schematic***, din pachetul software de dezvoltare Xilinx WebPACK ISE 6.2 i.

Calculatoarele sunt construite din circuite integrate care conțin elemente de comutare denumite porți. Porțile elementare sunt : **ŞI, SAU, ŞI-NU, SAU-NU** şi **NOT.** Circuitele simple pot fi realizate prin combinarea directă de porți individuale. Circuitele mai complexe sunt: multiplexoare, codificatoare, circuite de deplasare şi unitățile aritmetico-logice.

În 1975, Ron Cline de la Signetics (companie preluată mai târziu de Philips şi în cele din urmă de Xilinx) a avut ideea introducerii a două plane de programare. Cu ajutorul celor două plane de progamare, aşa cum se poate observa în figura 1.1, se poate realiza orice circuit descris printr-o combinație de porți ŞI şi SAU. Aceste dispozitive s-au numit dispozitive ***PLA*** *(Programmable Logic Array).*

Caracteristicile acestor tipuri de circuite constau în următoarele:

- conțin două planuri programabile;
- orice circuit compus dintr-o combinație de porți **ŞI** sau **SAU** poate fi implementat;
- ieşirile porților elementare **ŞI** sunt distribuite la intrările mai multor porți **SAU;**
- densitate mare de logică disponibilă pentru utilizator;

- timp de propagare mare *(T_{pd}),* deci viteza de funcţionare era relativ mică (mai mică decât a circuitelor PAL prezentate mai jos).

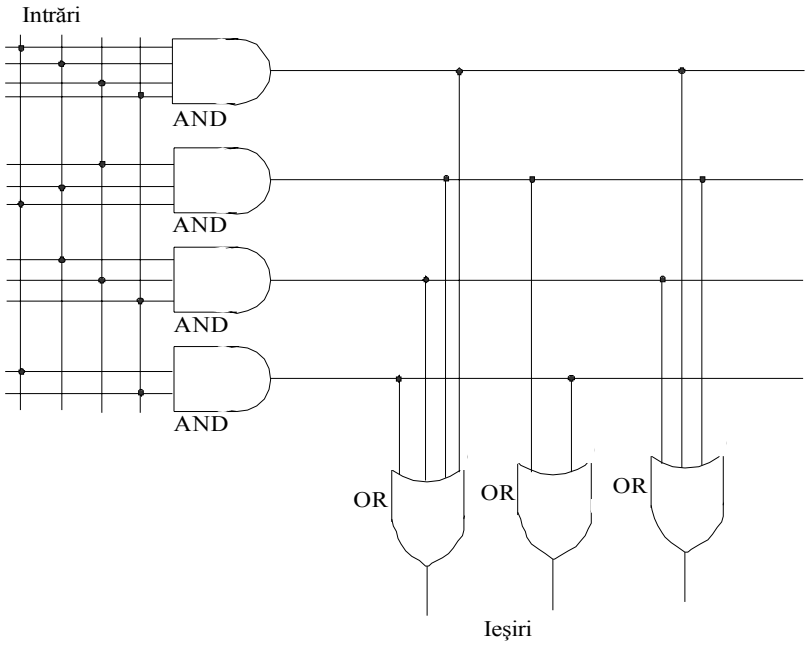


Figura 1.1: Exemplu de dispozitiv PLA.

O altă companie MMI (mai târziu preluată de compania AMD) a modificat această arhitectură obţinând arhitectura **PAL** (**P**rogrammable **A**rray **L**ogic - figura 1.2). Modificarea introdusă de MMI a constat în fixarea unui plan programabil (planul SAU), în acest fel valoarea lui T_{pd} a fost micşorată. Tot în urma acestei modificări, complexitatea circuitelor programabile a fost redusă. Dezavantajul a constat în pierderea flexibilităţii oferită de circuitele programabile **PLA**.

Alte arhitecturi au urmat acestora, (de exemplu **PLD** - **P**rogrammable **L**ogic **D**evice) dar fără a avea succesul comercial al arhitecturilor **PLA** sau **PAL**.

Toate circuitele din această familie erau programate electric şi erau şterse în aproximativ 20 de minute folosind lumină ultravioletă. Toate aceste circuite logice programabile au fost incluse în categoria circuitelor **SPLD** (**S**imple **PLD**).

Următoarele circuite logice apărute sunt circuitele CPLD - figura 1.3 (**C**omplex **P**rogrammable **L**ogic **D**evice). Conceptul acestor dispozitive presupune utilizarea de blocuri PLD sau macrocelule, interconectate între ele, într-un singur circuit.

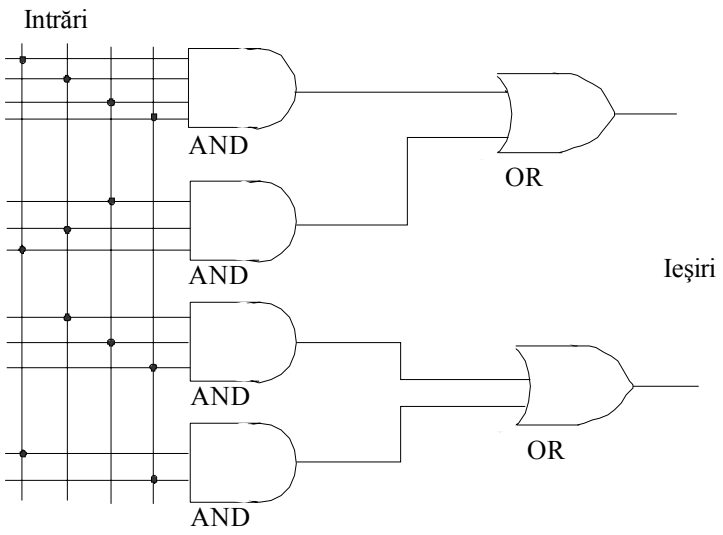


Figura 1.2: Arhitectură PAL produsă de MMI - Birkmer 1978.

Aceste circuite operează în părezent la viteze de 200MHz. O caracteristică importantă a acestor circuite este aceea că modelul de timp pentru circuitele proiectate este uşor de determinat.

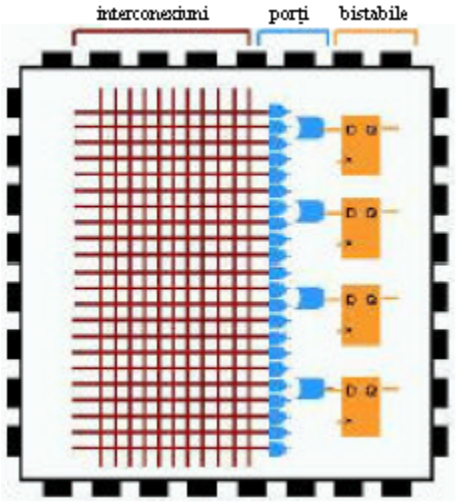


Figura 1.3: Arhitectură CPLD. Numărul maxim de porți este 200.

Dispozitivele CPLD posedă o serie de calități, care le fac utilizabile încă şi în prezent:

- oferă cea mai simplă cale de implementare a unui proiect. Odată ce proiectul a fost descris într-un limbaj HDL (**H**ardware **D**esign **L**anguage), programatorul utilizează un set de utilitare de dezvoltare CPLD în vederea optimizării şi simulării circuitului proiectat. Modificările ulterioare făcute circuitului proiectat sunt reimplementate în circuitul CPLD iar testarea poate avea loc imediat.

- costuri de dezvoltare reduse. Costurile achiziționării programelor de optimizare şi simulare în cazul circuitelor CPLD sunt reduse (programele de implementare, optimizare şi testare oferite de Xilinx sunt gratis).

- modificarea uşoară a circuitelor proiectate. Această calitate se datorează faptului că un circuit CPLD este reprogramabil. Este foarte uşor să se facă o modificare a proiectării, implementării şi testării chiar prin apeluri la distanță.

- aria de implementare este redusă. Această calitate este în directă corespondență cu calitatea uneltelor software folosite pentru optimizarea circuitului proiectat. Cu cât efortul de optimizare este mai mare cu atât timpul de procesare necesar este mai mare. Odată ce circuitul proiectat funcționează conform specificațiilor, se trece la implementarea în masă. Acum există o multitudine de firme producătoare de chip-uri care preiau doar fişierul obținut cu ajutorul programelor utilitare de implementare şi oferă circuitul hardware.

În 1985, compania numită Xilinx, aduce un nou concept. Realizarea unui circuit cu o structură regulată care să conțină celule logice sau module, interconectate între ele şi asupra cărora utilizatorul să aibă un control complet. Acest lucru implică faptul că utilizatorul poate proiecta, programa şi aduce modificări unui circuit oricând este necesar.

Circuitele de acest tip poartă numele de **F**ield **P**rogrammable **G**ate **A**rray ***FPGA*** (figura 1.4). Numărul de porți conținute de un circuit FPGA depăşeşte 10 milioane.

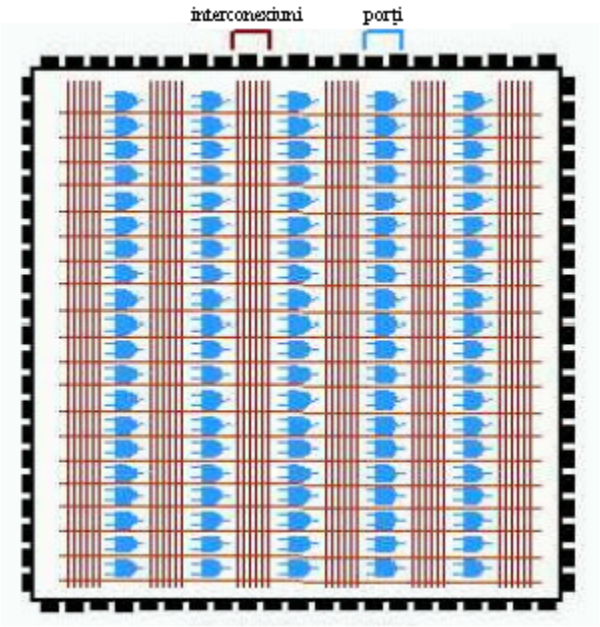


Figura 1.4: Arhitectură FPGA (Xilinx - Freeman - 1985); peste 10.000.000 de porți.

Actualmente există două tipuri de bază de circuite FPGA:

- SRAM FPGA
- OTP (**O**ne **T**ime **P**rogramable) FPGA

Aceste două tipuri diferă prin modalitatea de implementare a celulelor logice, precum şi prin mecanismul utilizat pentru realizarea conexiunilor în interiorul circuitului. Aşa cum se poate uşor intui, piața de circuite FPGA este dominată de către SRAM FPGA. Dacă la circuitele FPGA de tipul OTP se utilizează porți logice tradiționale, pentru implementarea circuitului proiectat, la cele de tipul SRAM se utilizează LUT (**L**ook **U**p **T**able).

Circuite secvențiale sincrone

Definiție: Sistemele digitale în a căror componență există elemente de memorie precum şi elemente logice combinaționale se numesc circuite secvențiale.

Ieşirea unui circuit secvențial la orice moment de timp este o funcție de intrările externe ale sale şi de starea sa internă la acel moment de timp. Starea circuitului este definită de conținutul elementelor de memorie din circuit şi este o funcție de stările anterioare şi de intrările circuitului.

Există două tipuri de circuite secvențiale: *sincrone* şi *asincrone.* Comportarea unui circuit sincron depinde de valorile semnalului la momente discrete de timp. Comportarea unui circuit asincron depinde de ordinea în care se modifică semnalele de intrare, aceste modificări putând apărea la orice moment de timp. Instanțele de timp discrete într-un circuit sincron sunt determinate de un semnal de control, uzual denumit ***ceas*.**

Circuitele secvențiale sincrone utilizează bistabilele ca elemente de memorie. Un bistabil este un circuit electronic care poate memora l sau 0. Un bistabil poate memora una din cele două stări logice până la apariția frontului de ceas. În figura 1.5 sunt prezentate simbolurile Xilinx ale celor mai utilizate bistabile: D, JK şi T, împreună cu tabelele lor de excitație.

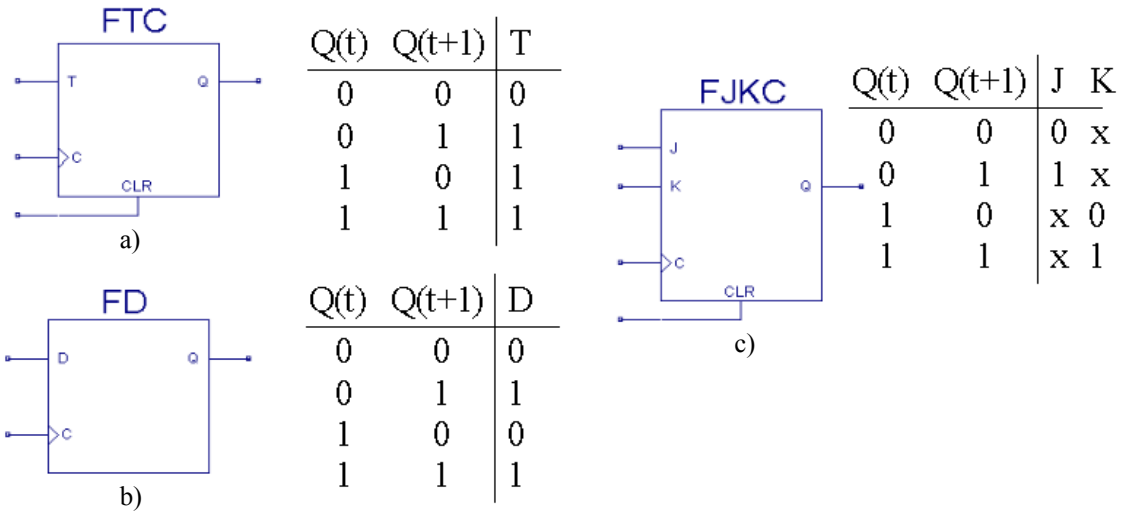


Figura 1.5: Bistabilele T (a), D (b), şi JK (c); simboluri şi tabele de excitație.

Procedura clasică urmată în cazul proiectării oricărui circuit secvențial sincron este următoarea:

1. se construieşte diagrama logică a circuitului pornind de la enunțul problemei;
2. se determină numărul de bistabile *p* necesare, pornind de la numărul de stări ale diagramei stabilite la pasul 1, utilizând formula: $2^{p-1} < n < 2^p$, unde *n* reprezintă numărul de stări;
3. se asigură câte o etichetă de lungime *p* biți pentru fiecare stare;
4. se stabileşte tabela stărilor de tranziție şi tabela de ieşire;
5. se determină o tabelă de intrare pentru fiecare bistabil, utilizând tabelele de excitație prezentate în figura 1.5;
6. se determină ecuațiile de intrare pentru fiecare intrare a bistabilelor ce compun circuitul secvențial sincron;
7. se desenează diagrama circuitului.

Pentru a exemplifica procedura enunțată se consideră un exemplu practic. Se doreşte proiectarea unui automat folosit în distribuirea băuturilor răcoritoare. Se presupune că fiecare sticlă costă 15.000 lei. Automatul acceptă monede de 5.000 lei precum şi bancnote de 10.000 şi 50.000 lei.

Se presupune existența unui mecanism care sortează banii primiți şi emite trei semnale, câte unul pentru fiecare tip. Cele trei semnale determină tranziții ale automatului dintr-o

stare în alta în funcție de valorile lor. Automatul trebuie să elibereze sticla şi restul corect în cazul in care suma acumulată este mai mare decât 15.000 de lei.

Pasul 1. Diagrama logică corespunzătoare enunțului problemei este prezentată în figura 1.6.

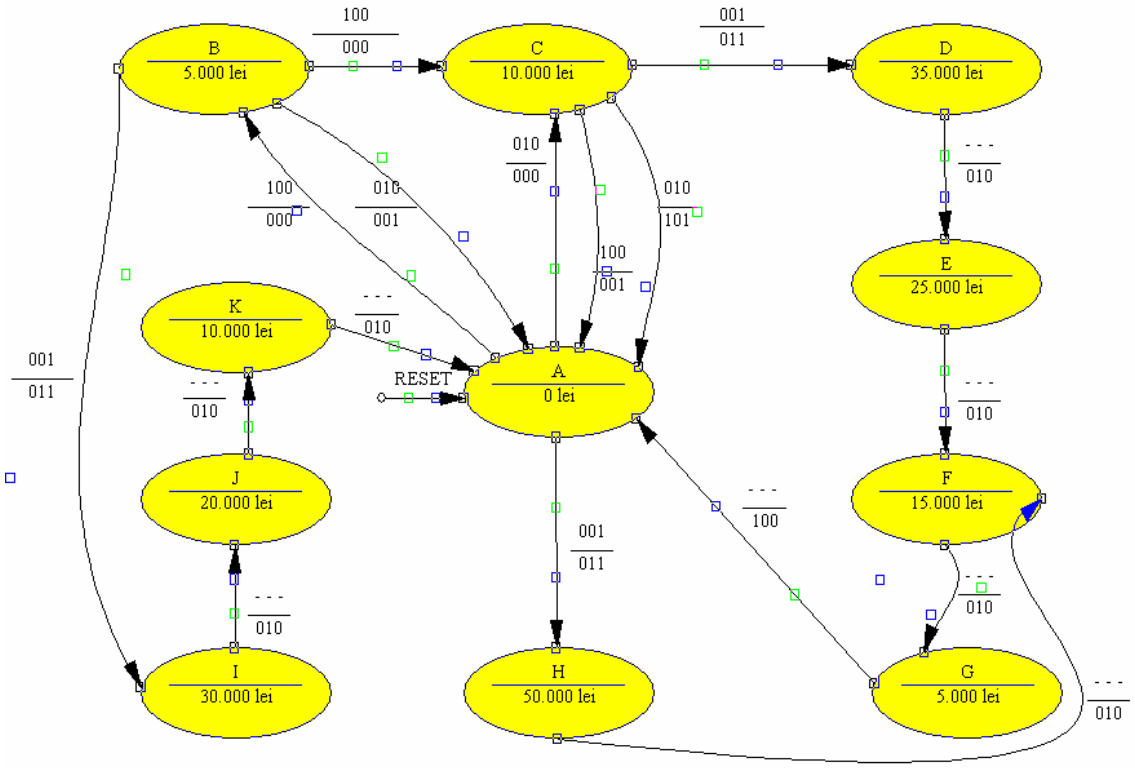


Figura 1.6: Diagrama logică pentru automatul de distribuit băuturi răcoritoare.

Pasul 2. În diagrama din figura 1.6 se poate observa că numărul de stări este 11. Deoarece 2^4 este primul număr mai mare decât 11, rezultă că vor fi necesare 4 bistabile. Bistabilele utilizate în implementare sunt bistabile de tipul D. Ele vor fi numerotate D_0, ... D_3 iar tabela de excitație utilizată este cea prezentată în figura 1.5.

Pasul 3. Codificarea stărilor este următoarea: A - 0000, B - 0001, C - 0010, D - 0011, E - 0100, F - 0101, G - 0110, H - 0111,I - 1000, J - 1001, K - 1010.

Pasul 4. Din diagrama de stări stabilită la pasul l, se poate deduce tabelul 1.1. Codificarea semnalelor cuprinse în tabelul 1.1 este următoarea:

- *M, B_1 , B_2* - semnale care se activează când o monedă de 5.000 lei, bancnotă de 10.000 / 50.000 lei este introdusă în aparat;

- *EM, EB_1* - semnal care se activează când o monedă de 5.000 lei, bancnotă de 10.000 lei trebuie returnată de către aparat;

- *ESB* - semnal care se activează $M \cdot \overline{B_1} \cdot \overline{B_2}$ când o sticlă de băutură răcoritoare trebuie eliberată de către aparat.

Stare inițială	Stare Finală	Condiții	Comenzi
A - 0000	B 0001	$M \cdot \overline{B_1} \cdot \overline{B_2}$	
	C 0010	$\overline{M} \cdot B_1 \cdot \overline{B_2}$	
	H 0111	$\overline{M} \cdot \overline{B_1} \cdot B_2$	EB_I & ESB
	A 0000	$\overline{M} \cdot \overline{B_1} \cdot \overline{B_2}$	
B - 0001	A 0000	$\overline{M} \cdot B_1 \cdot \overline{B_2}$	ESB
	I 1000	$\overline{M} \cdot \overline{B_1} \cdot B_2$	EB_I & ESB
	C 0010	$M \cdot \overline{B_1} \cdot \overline{B_2}$	
	B 0001	$\overline{M} \cdot \overline{B_1} \cdot \overline{B_2}$	
C - 0010	D 0011	$\overline{M} \cdot \overline{B_1} \cdot B_2$	EB_I & ESB
	A 0000	$M \cdot \overline{B_1} \cdot \overline{B_2} + \overline{M} \cdot B_1 \cdot \overline{B_2}$	ESB + EM & ESB
	C 0010	$\overline{M} \cdot \overline{B_1} \cdot \overline{B_2}$	
D - 0011	E 0100	- - -	EB_I
	D 0011	$\overline{M} \cdot \overline{B_1} \cdot \overline{B_2}$	
E - 0100	F 0101	- - -	EB_I
	E 0100	$\overline{M} \cdot \overline{B_1} \cdot \overline{B_2}$	
F - 0101	G 0110	- - -	EB_I
	F 0101	$\overline{M} \cdot \overline{B_1} \cdot \overline{B_2}$	
G - 0110	A 0000	- - -	EM
H - 0111	F 0101	- - -	EB_I
	H 0111	$\overline{M} \cdot \overline{B_1} \cdot \overline{B_2}$	
I - 1000	J 1001	- - -	EB_I
	I 1000	$\overline{M} \cdot \overline{B_1} \cdot \overline{B_2}$	
J - 1001	K 1010	- - -	EB_I
	J 1001	$\overline{M} \cdot \overline{B_1} \cdot \overline{B_2}$	
K - 1010	A 0000	- - -	EB_I

Tabelul 1.1: Tabelul stărilor de tranziție.

Pasul 5. În acest pas se elaborează tabelele *K-V* de minimizare pornind de la tabelul 1.1 şi tabelul de excitație pentru bistabilul D din figura 1.5. Rezultatul obținut este prezentat

în figura 1.7. Această tabelă este valabilă doar pentru bistabilul 4, corespunzător intrării D_3.

$y_3\ y_2$ \ $y_1\ y_0$	00	01	11	10
00	0	$\overline{M}\cdot\overline{B_1}\cdot B_2$	0	0
01	0	0	0	0
11	x	x	x	x
10	1	1	x	0

Figura 1.7: Tabela K-V de minimizare pentru D_3.

Rezultatul obținut în urma minimizării tabelei K-V din figura 1.7 este:

$$D_3 = \overline{y_1}\cdot\left(y_3 + y_0\cdot\overline{y_2}\cdot\overline{M}\cdot\overline{B_1}\cdot B_2\right) \qquad (1.1)$$

În mod analog se deduc şi celelalte funcții de excitație. Rezultatele obținute sunt:

$$D_2 = y_2\cdot\overline{y_1} + y_2\cdot y_0 + \overline{M}\cdot\overline{B_1}\cdot B_2\cdot\left(\overline{y_3}\cdot\overline{y_1}\cdot\overline{y_0} + y_2\cdot y_0\right) + \overline{\overline{M}\cdot\overline{B_1}\cdot\overline{B_2}}\cdot\left(\overline{y_3}\cdot y_1\cdot y_0 + y_2\cdot\overline{y_1}\right)$$

$$D_1 = \overline{\overline{M}\cdot\overline{B_1}\cdot\overline{B_2}}\cdot\left(y_2\cdot\overline{y_1}\cdot y_0 + y_3\cdot y_0\right) + \overline{M}\cdot\overline{B_1}\cdot\overline{B_2}\cdot\left(\overline{y_3}\cdot\overline{y_2}\cdot y_1 + y_1\cdot y_0\right) + \overline{y_3}\cdot\overline{y_2}\cdot$$
$$\cdot\left(M\cdot\overline{B_1}\cdot\overline{B_2}\cdot\overline{y_1}\cdot y_0 + \overline{M}\cdot\overline{B_1}\cdot B_2\cdot\overline{y_0} + \overline{M}\cdot B_1\cdot\overline{B_2}\cdot\overline{y_1}\cdot\overline{y_0}\right) \qquad (1.2)$$

$$D_0 = y_2\cdot y_1\cdot y_0 + \overline{\overline{M}\cdot\overline{B_1}\cdot\overline{B_2}}\cdot\overline{y_1}\cdot\overline{y_0}\cdot\left(y_2 + y_3\right) + \overline{M}\cdot\overline{B_1}\cdot\overline{B_2}\cdot y_0 + \overline{B_1}\cdot\overline{y_3}\cdot\overline{y_2}\cdot\overline{y_0}\cdot$$
$$\cdot\left(\overline{M}\cdot B_2 + M\cdot\overline{B_2}\cdot\overline{y_1}\right) + \overline{M}\cdot\overline{B_1}\cdot B_2\cdot y_2\cdot y_1\cdot y_0$$

Pasul 6. Pentru fiecare semnal de ieşire *(EM, EB_1 şi E S B)* se determină câte o tabelă K-V cu ajutorul căreia se va stabili forma canonică. Spre exemplu, pentru semnalul *EM,* pornind de la tabelul 1.1, se poate deduce tabela K-V prezentată în figura 1.8.

$y_3\ y_2$ \ $y_1\ y_0$	00	01	11	10
00	0	0	0	$\overline{M}\cdot B_1\cdot\overline{B_2}$
01	0	0	0	1
11	x	x	x	x
10	0	0	x	0

Figura 1.8: Tabela K-V de minimizare pentru *EM.*

Conform tabelei K-V prezentată în figura 1.8 se poate deduce următoarea formă canonică pentru *EM:*

$$EM = \overline{y_3}\cdot y_1\cdot\overline{y_0}\cdot\overline{M}\cdot B_1\cdot\overline{B_2} + y_2\cdot y_1\cdot\overline{y_0} \quad (1.3)$$

Procedând în mod analog pentru semnalele EB_1 si *ESB* se obțin următoarele forme canonice:

$$EB_1 = y_3\cdot\overline{y_2}\cdot y_1\cdot\overline{y_0} + \overline{M}\cdot\overline{B_1}\cdot B_2\cdot\left(\overline{y_3}\cdot\overline{y_2}\cdot\overline{y_1} + \overline{y_2}\cdot y_1\cdot\overline{y_0}\right) + \overline{\overline{M}\cdot\overline{B_1}\cdot\overline{B_2}}\cdot\left(y_3 + y_1\cdot y_0 + y_2\cdot\overline{y_1}\right) \quad (1.4)$$

$$ESB = \overline{y_3}\cdot\overline{y_2}\cdot\left[\overline{M}\cdot\overline{B_1}\cdot\overline{B_2}\cdot\left(\overline{y_1} + \overline{y_0}\right) + \overline{M}\cdot B_1\cdot\overline{B_2}\cdot\left(y_1 \oplus y_0\right) + M\cdot\overline{B_1}\cdot\overline{B_2}\cdot y_1\cdot\overline{y_0}\right] \quad (1.5)$$

Pasul 7 . Pentru desenarea diagramei circuitului se foloseşte o procedură destul de laborioasă care va fi prezentată în paragraful următor.

Implementarea unui circuit secvențial folosind Xilinx ISE

Cu ajutorul pachetului software WebPACK ISE, proiectele diverselor circuite pot fi introduse foarte uşor şi rapid utilizând limbaje de descriere cum sunt VHDL sau Verilog sau utilizând utilitarul ***Schematic*** . Acest utilitar foloseşte metoda tradiţională de proiectare, utilizând biblioteci de porți logice şi circuite logice programabile. Marele dezavantaj al acestei metode este faptul că proiectul rezultat este dependent total de tehnologia şi circuitul FPGA ales la începutul proiectării. Un alt dezavantaj major îl constituie faptul că circuitele elaborate cu acest utilitar nu mai respectă conceptul de ***reutilizare.***

Acest concept este deosebit de important în industrie, unde un circuit odată proiectat este folosit pentru orice platformă sau circuit FPGA/CPLD existent. Acest concept este valabil pentru circuitele proiectate folosind limbaje de descriere hardware cum sunt VHDL sau Verilog.

OBSERVAȚIE . Mediul de programare Xilinx WebPACK ISE este prezentat în detaliu în *Anexa A*. Se recomandă citirea cu atenție a acestei anexe înaintea începerii realizării oricărei proiectări.

Proiectarea unui circuit cu ajutorul utilitarului ***Schematic*** presupune parcurgerea următorilor paşi:

1. Stabilirea circuitului şi a familiei de circuite FPGA/CPLD utilizate în implementarea circuitului proiectat.

   Din meniul ***File*** se selectează opțiunea *New* ***Project*** . Aici se alege un nume pentru proiect, spre exemplu *laboratorl.* Pentru a putea implementa fizic proiectul în final, se introduce *XC9500 CPLDs* în câmpul *Device Family, Auto-PQ XC9500* în câmpul *Device* şi *XST Verilog* în câmpul *Design Flow*. În final se selectează opțiunea *New Source* din meniul *Project.*

   În noua fereastra deschisă se selectează opținea *Schematic* şi se introduce un nume pentru circuitul proiectat, spre exemplu *soft- drink.* Se execută *Next* şi apoi *Close.* Dacă toate operațiile au fost realizate cu succes, atunci o pagină nouă de lucru, ca în figura 1.9, devine disponibilă pentru a realiza circuitul dorit, utilizând doar circuitele din biblioteca XC9500 CPLD.

   În figura 1.9 sunt prezentate doar câteva din opțiunile disponibile. Restul opțiunilor pot fi uşor descoperite dacă săgeata mouse-ului este mutată deasupra opțiunii dorite.

2. Proiectarea circuitului dorit utilizând doar circuite componente ale bibliotecii XC9500 CPLD. Circuitul proiectat va conține patru bistabile *D* precum şi logica aferentă implementării ecuațiilor de la 1.1 până la 1.4. Este important pentru simulare ca proiectul să conțină porturile de intrare şi de ieşire.

Înainte de a trece la pasul următor, se verifică dacă implementarea este corectă. Pentru aceasta se alege opțiunea *Check Schematic* din meniul *Tools.*

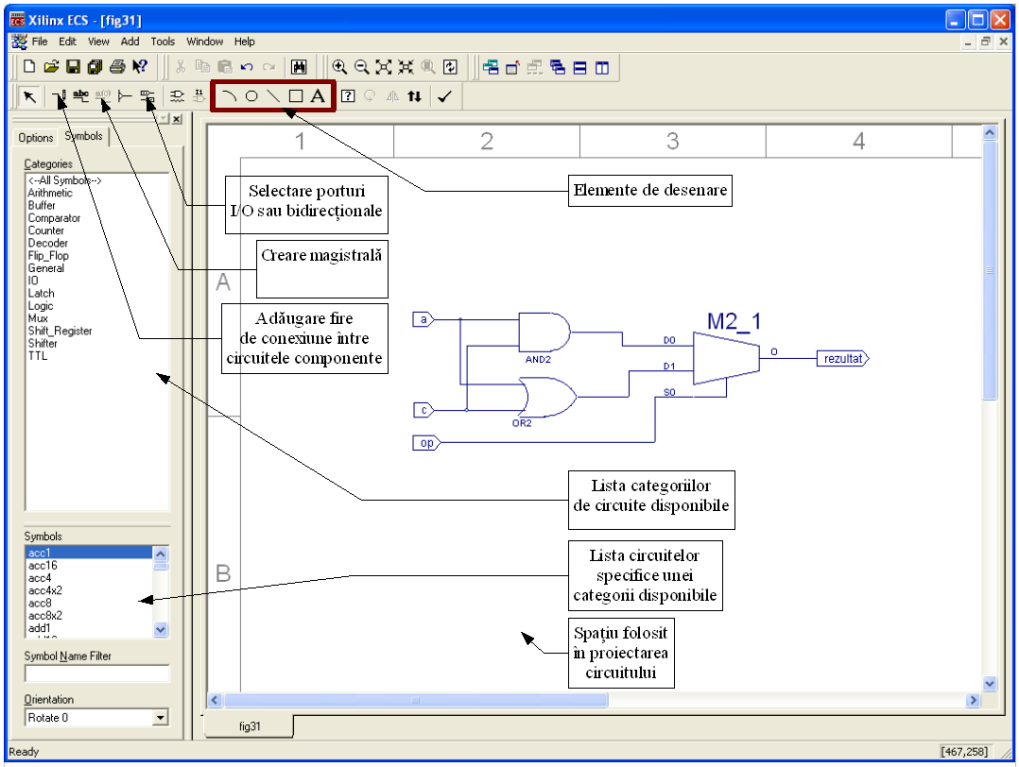


Figura 1.9: Mediul de proiectare ***Schematic .***

3. Ultimul pas, înaintea simulării, este crearea fişieirului *netlist.* Acesta este un fişier text echivalent cu circuitul generat la pasul anterior. El reprezintă o cale compactă pentru alte programe care doresc să "înțeleagă" ce porți logice există în proiect, cum sunt ele conectate şi care sunt denumirile porturilor de I/O.

Formatul ***EDIF*** (**E**lectronic **D**igital **I**nterchange **F**ormat) reprezintă standardul industrial pentru fişierele netlist. Xilinx oferă un format propriu pentru fişierul netlist, formatul ***XNF*** (**X**ilinx **N**etlixt **F**ormat).

2. Desfăşurarea lucrării

1. Fololosind ecuațiile 1.1 pâna la 1.5, utilitarul ***Schemtic*** precum şi circuitele disponibile în biblioteca XC9500CPLD, se va implementa circuitul secvențial prezentat în figura 1.6.

2. Se va simula circuitul obținut la punctual 1, folosind simulatorul ModelSim.

3. Se va implementa automatul obținut prin intermediul pachetului de dezvoltare Xilinx WebPACK ISE 6.2i în circuitul CPLD XC9500.

3. Probleme propuse

1. Se dă funcţia booleana: $F = x \cdot y + x + x \cdot y \cdot z$.

   - Determinaţi tabela de adevăr a acestei funcţii.
   - Determinaţi diagrama logică utilizând expresia booleana originală.
   - Simplificaţi expresia algebrică utilizând algebra booleana.
   - Determinaţi tabela de adevăr a expresiei simplificate şi arătaţi că este aceeaşi cu cea determinată la punctul a.
   - Desenaţi diagrama logică a expresiei simplificate şi comparaţi numărul total de porţi logice folosite cu numărul total de porţi logice folosite în diagrama stabilită la punctul b.

2. Simplicaţi următoarele funcţii booleene utilizând diagrame K-V:

   - $F(A, B, C, D) = \sum (4,6,7,15)$
   - $F(A, B, C, D) = \sum (3, 7, 11, 13, 14, 15)$
   - $F(A, B, C, D) = \sum (0, l, 2, 4, 5, 7, 11, 15)$
   - $F(A, B, C, D) = \sum (0, 2, 4, 5, 6, 7, 8, 10)$

3. Să se proiecteze şi să se simuleze utilizând WebPACK ISE, un numărător care numără crescător/descrescător de la 0 la 15.

4. Să se proiecteze şi să se simuleze utilizând WebPACK ISE, un circuit secvenţial care recunoaşte următoarea secvenţă: 0001 1000.

2

Proiectarea aplicațiilor folosind ierarhii de module

1. Prezentare teoretică

În cadrul acestui laborator se va prezenta metodologia de proiectare TOP DOWN utilizată în cazul proiectării oricărui circuit netrivial. Deasemenea, vor fi prezentate o parte din utilitarele cuprinse în pachetul de dezvoltare Xilinx WebPACK ISE 6.2i care reduc drastic timpul alocat implementării şi testării unui circuit.

Actualmente, în industrie, proiectele realizate sunt deosebit de complexe. Metoda utilizată în vederea proiectării lor este tehnica ***TOP DOWN*** . Conform acestei metode, proiectul este împărțit în module, fiecare modul fiind implementat şi testat separat, urmând ca la final modulele să fie combinate. WebPACK ISE 6.2 pune la dispoziția proiectantului o serie de utilitare care pot fi utilizate în proiectarea modulelor precum ***CORE Generator, State Diagram, HDL Design*** şi ***DCM Wizard*** . În final, combinarea tuturor modulelor se va realiza prin intermediul utilitarului ***Schematic*** **.** Pentru a exemplifica metoda ***TOP DOWN*** precum si utilitarele disponibile în WebPACK ISE 6.2, se va considera implementarea următoarei secvențe de program:

```
for (x = 0, y = 3, i = 0, i ≤ 10; i = i + 1)
     x = x + y;
if (x < 0)
     y = 0;
else
     x = 0;
```

Numerele se consideră binare de 8 biți iar valoarea sumei $x = x + y$ va fi afişată cu ajutorul unui dispozitiv LCD cu şapte segmente. Resursele hardware necesare implementării secvenței de program sunt prezentate în figura 2.1.

Toate aceste resurse vor fi implementate pentru circuitul CPLD XC9500 aflat în dotarea laboratorului. Primii paşi de creare a unui proiect, selectarea circuitului FPGA dorit, etc. se presupun cunoscuți şi realizați.

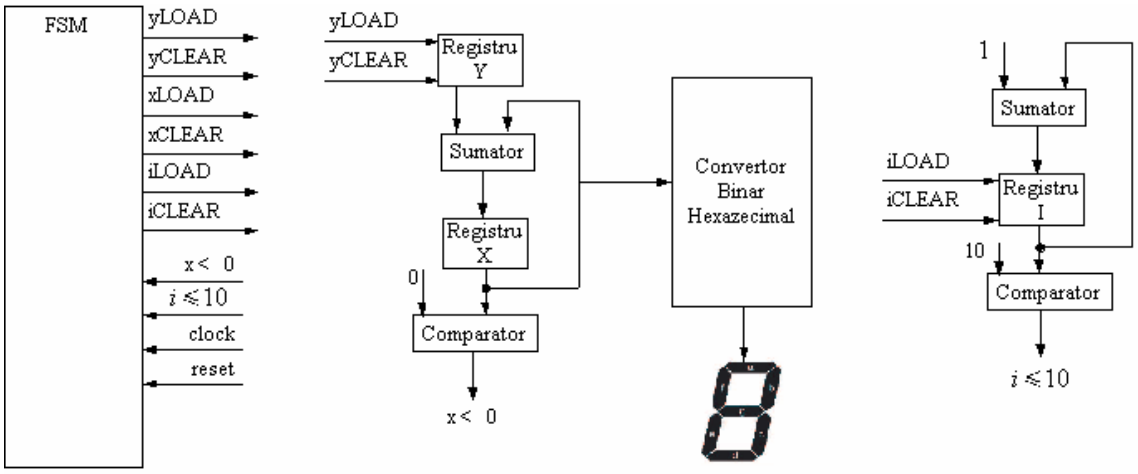


Figura 2.1: Resursele hardware necesare implementării secvenței de cod.

Întreaga implementare realizată ca exemplu păstrează principiul ***reutilizabilității.*** În consecință, modulele sunt implementate utilizând limbajul **VERILOG** şi nu se va face apel la librăria Xilinx, disponibilă pentru acest tip de circuit CPLD. Singurul modul implementabil cu ajutorul utilitarului ***State Diagram*** este modulul FSM.

Modulul registru - Se proiectează în Verilog conform următoarei proceduri:

- Se selectează opțiunea *New Source* din cadrul meniului *Project.* În fereastra nou apărută se alege *Verilog Module* urmând să se introducă numele modulului, spre exemplu *registru.*

- Următoarea fereastră este dedicată definirii intrărilor/ieşirilor modulului specificând pentru fiecare semnal numele, tipul semnalului (I/O), bitul cel mai semnificativ precum şi bitul cel mai puțin semnificativ. În vederea unei reutilizabilități cât mai mare se ignoră acest pas.

  Pentru implementările viitoare este mai bine să se definească dimensiunea oricărui semnal de I/O prin intermediul unui parametru *Width.* Dacă într-un alt proiect este nevoie de intrări de 32 biți, spre exemplu, este suficientă modificarea doar a acestui parametru, restul implementării rămânând nemodificată.

- Se implementează modulul dorit, utilizând sintaxa şi construcțiile permise de Verilog. Spre exemplu, implementarea registrului necesară proiectului poate fi observată în figura 2.2.

- Se creează un simbol care va fi utilizat în schema finală prin apelarea utilitarului

Create Schematic Symbol.

- În cazul în care modulul proiectat este complex, se va realiza o simulare comportamentală a modulului utilizând simulatorul ***ModelSim*** **.**

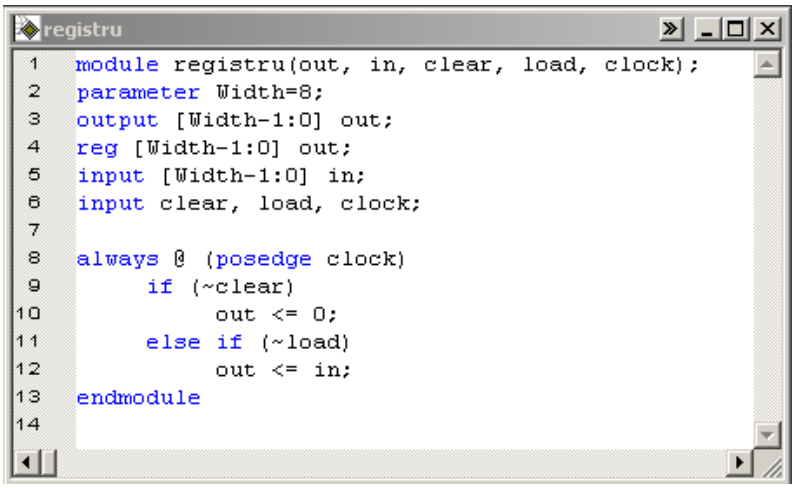


Figura *2.2:* Secvența de cod Verilog pentru implementarea unui registru.

Această procedură se repetă pentru proiectarea modulelor: *sumator, convertor, comparator-LEQ, comparatorLT, zero-gen, zece-generator* şi *unu-generator.*

Modulul FSM - Proiectarea lui se realizează cu ajutorul utilitarului *State Diagram.* Din meniul *Project* se selectează opțiunea *New Source.* În fereastra nou deschisă se introduce numele *fsm* şi se selectează opțiunea *State Diagram.* Panoul cu butoanele de lucru se află poziționat stânga față de foaia virtuală de lucru. Introducerea diagramei de stări se face conform următoarei proceduri:

- Se va selecta butonul *Add State;* apoi se va poziționa în cadrul foii virtuale de lucru starea *STATE0.*

    1. Pentru modificarea numelui acestei stări se execută dublu click asupra stării *STATE0.* În fereastra nou deschisă se introduce *A* în loc de *STATE0.* În cazul exemplului prezentat sunt necesare cinci stări: *A, B, C, D, E.*

    2. Se repetă pasul *2* până se introduc toate stările, care compun diagrama de stări a circuitului.

    3. Se selectează butonul *Add Transition* pentru a adăuga tranzițiile dintr-o stare într-o altă stare. În situația în care, datorită unor condiții, se rămâne

în aceeaşi stare, se va utiliza tot butonul *Add Transition*.

După selectarea butonului *Add Transition,* se alege starea inițială şi apoi starea următoare. În mod automat se va trasa o linie sau un arc de cerc terminate printr-o săgeată care va uni cele două stări selectate. Direcțiile săgeților vor indica modalitatea de parcurgere a stărilor.

În cazul exemplului prezentat, stările sunt parcurse în ordine de la starea *A* până la starea *E*. Există şi o excepție: din starea *C* este posibilă reîntoarcerea în starea *B*.

4. Adăugarea unei acțiuni pentru o stare: acțiunea unei stări arată cum trebuie să se comporte ieşirile circuitului într-o stare dată. Pentru a specifica comportarea ieşirilor într-o stare dată trebuie mai întâi să se selecteze starea dorită şi apoi să se efectueză un dublu click. Fereastra nou apărută are un buton, *Output Wizard,* utilizat pentru specificarea comportării ieşirilor.

   În cazul exemplului prezentat, în starea A, ieşirile *xCLEAR* şi *iCLEAR* trebuie să aibă valoarea zero. Specificarea acestor cerințe se realizează cu ajutorul următoarei secvențe de operații:

   - Se selectează starea *A* şi se execută dublu click;
   - Se selectează butonul **Output Wizard.**
   - În secțiunea *Output Wizard* se selectează valorile **DOUT = xCLEAR, CONSTANT = 0;**
   - Se apasă butonul **OK** pentru a asigna valoarea constantă 0 ieşirii **xCLEAR;**
   - Se selectează butonul **Output Wizard.**
   - În câmpul *Output Wizard* se aleg valorile **DOUT = iCLEAR, CONSTANT = 0;**
   - Apăsarea butonului **OK** implică asignarea valorii 0 ieşirii **iCLEAR;**
   - Se selectează butonul **OK** pentru a închide fereastra **Edit State.**

Rezultatul aplicării unei astfel de secvențe de operații poate fi observat în figura 2.3.

Odată cu creşterea experienței în utilizarea utilitarului ***State Editor*** , se va observa că acelaşi efect se obține şi dacă în interiorul câmpului *Outputs* din cadrul ferestrei *Edit State* se introduce secvența *xCLEAR* = 0; *iCLEAR* = 0.

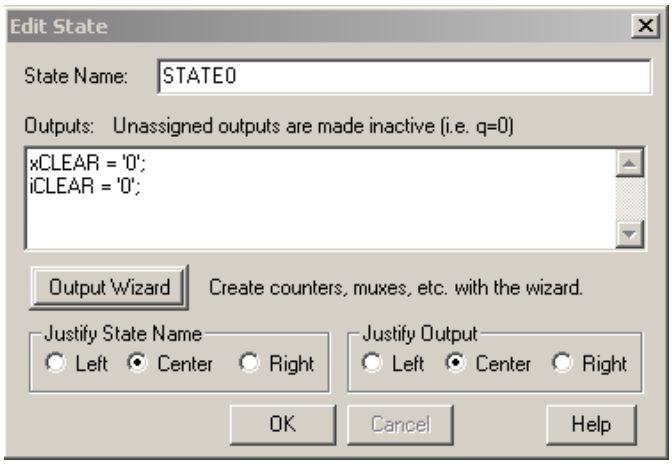


Figura 2.3: Introducerea de acţtiuni pentru starea A.

5. Se repetă pasul 5 până se epuizează toate stările, care compun graful stărilor circuitului.

6. Adăugarea condiţiei ***RESET*** - Această condiţie este deosebit de importantă deoarece prima stare care este activată (starea de intrare), este indicată de RESET. Pentru adăugarea ei trebuie selectat butonul *AddReset.* Se alege punctul iniţial în jurul stării de intrare şi apoi se selectează starea de intrare. În cazul exemplului prezentat, punctul de start al condiţiei *RESET* trebuie unit cu starea *A*.

7. Trecerea dintr-o stare în altă stare se poate realiza fie prin intermediul unor condiţii, fie automat. În cazul automat, linia de tranziţie care uneşte cele două stări nu conţine nici o condiţie. În cazul în care trecerea de la o stare într-o altă stare se realizează numai prin intermediul unor condiţii, aceste condiţii trebuie specificate.

   Este suficient să se realizeze un dublu click pe linia de tranziţie dorită pentru că fereastra *Edit Condition* să fie deschisă. În cadrul acestei ferestre, în câmpul *Condition* se introduc condiţiile care trebuie îndeplinite. Acest pas se repetă până când sunt introduse toate condiţiile care afectează funcţionarea circuitului.

   În cazul exemplului prezentat, circuitul trece din starea *C* în starea B numai dacă $i < 0$. Respectarea acestei condiţii este dată de valoarea semnalului *LT.* Dacă $LT = 1$, atunci inegalitatea $i < 0$ este adevărată, altfel $LT = 0$. Rezultatul implementării paşilor prezentaţi poate fi observat în figura 2.4.

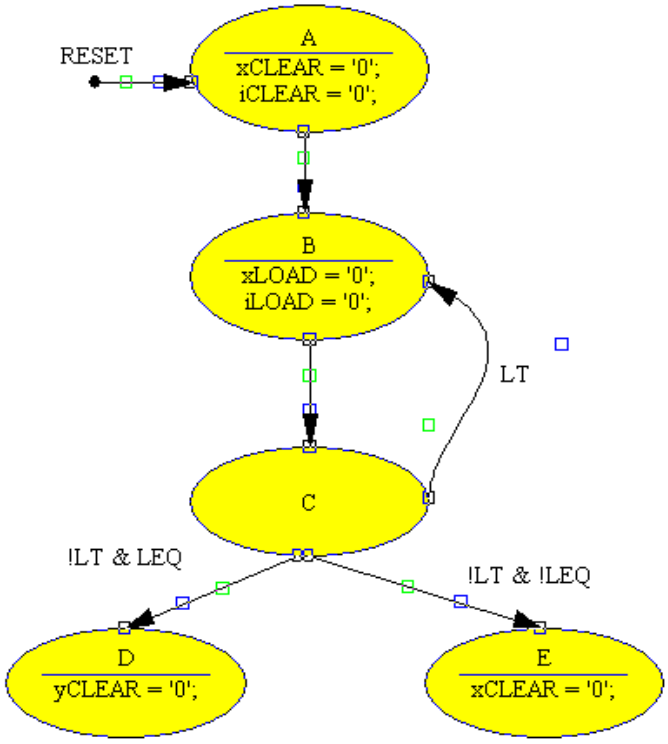


Figura 2.4: Modulul FSM proiectat cu ajutorul utilitarului **State Diagram.**

- Optimizarea circuitului *FSM* obținut - această opțiune este absolut necesar să fie selectată pentru a obține un circuit performant în funcție de un criteriu ales de noi, spre exemplu viteza de funcționare a circuitului rezultat sau minimizarea ariei ocupate de noul circuit. Intrarea în această procedură se realizează odată cu selectarea butonului *Optimize* situat în bara orizontală de butoane. În cazul exemplului prezentat, are loc următoarea succesiune de operații:

    1. Se selectează butonul *Optimize* aflat în bara orizontală de butoane;

    2. Circuitul utilizat pentru implementare este XC9500 care este un circuit Xilinx CPLD; de aceea se va alege opțiunea *CPLD/PAL;*

    3. Se doreşte ca circuitul să funcționeze cat mai rapid (implicit aria ocupată va fi mai mică), deci se va alege opțiunea *Speed(register outputs);*

4. Există şansa, în cazul proiectelor care conţin zeci sau sute de stări, ca nu toate condiţiile impuse să fie acoperite. Pentru a evita acest neajuns se va alege opţiunea *Guarantee coverage.* Automatul proiectat în figura 2.4 conţine un număr mic de stări ceea ce va conduce la selectarea opţiunii *Maximize Speed. Reduce Areea.*

5. Următoarea fereastră se referă la optimizarea modului de încărcare a semnalelor de intrare ale automatului. Se preferă utilizarea setărilor deja existente.

6. Automatul proiectat este translatat în cod Verilog/VHDL sau ABEL pentru a putea fi apelat sub formă de modul, într-un alt modul aflat pe un nivel superior, în cadrul ierarhiei de module.

7. Unul dintre utilitarele cu care se pot realiza toţi paşii de optimizare descrişi este utilitarul *Xilinx XST.*

Rezultatul procedurii de optimizare a automatului cu stări proiectat este un fişier Verilog, aşa cum se poate observa în figura 2.5.

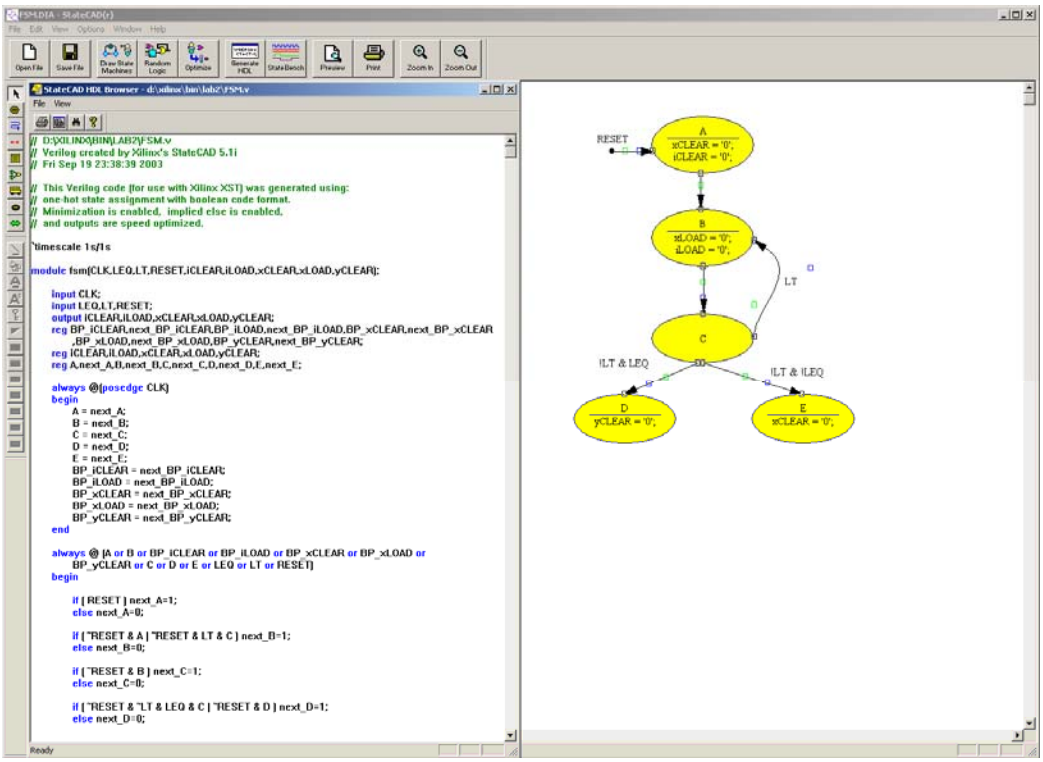


Figura 2.5: Optimizarea modulului FSM rezultă într-o descriere Verilog.

Modulul TOP– Este considerat modulul de vârf şi posedă ca submodule toate modulele proiectate până în acest moment. Conform procedurii de proiectare Xilinx, el este în mod obligatoriu proiectat cu ajutorul utilitarului *Schematic*. Proiectarea lui presupune conectarea tuturor submodulelor componente şi stabilirea semnalelor de intrare/ieşire.

Nu trebuie pierdut din vedere faptul că modulul proiectat anterior, FSM, trebuie adăugat la proiect. Pentru aceasta, se selectează opţiunea *Add Source,* din meniul *Project,* şi se alege fişierul FSM.v. Ultima etapă este selectarea sursei FSM.v şi lansarea procedurii *Create Schematic Symbol.* În urma acestui pas, sunt create toate modulele necesare implementării circuitului dorit.

OBSERVAŢIE – În codul sursă al fişierului *FSM.v* fost introdusă şi o linie de cod prin care ieşirea *yLOAD* primeşte valoarea 0 indiferent de starea în care se găseşte automatul.

Ultima etapă a procesului de proiectare o constituie verificarea întregului circuit proiectat. Această etapă presupune utilizarea unui simulator performant, de exemplu ModelSim XE.

Următorul proces este procesul de ***sinteză.*** Acesta presupune în primul rând specificarea exactă a tipului de circuit CPLD folosit în procesul de ***implementare.*** În fereastra *Sources in Projects* se poate observa că întreaga proiectare realizată până acum a fost executată pentru *Auto xc9500-*.

Deoarece proiectarea a fost realizată independent de platforma utilizată (proiectarea modulelor s-a executat în Verilog), este foarte simplu să se modifice tipul circuitului Xilinx fără însă a modifica modulele deja proiectate.

Pentru a schimba tipul circuitului Xilinx, se face dublu click în fereastra *Sources in Projects* pe *Auto xc9500-*. În noua fereastră se selectează în câmpul *Device* circuitul xc95108. Restul câmpurilor sunt completate automat şi nu este necesar să fie modificate. Pentru a putea lansa utilitarul *PACE*, este necesar să se specifice fişierul de constrângeri. Pentru aceasta, se adaugă un nou fişier sursă de tipul *Implementation Constraints File* la proiect, iar în fereastra de editare a fişierului de constrângeri, se realizează următoarele operaţii:

- Se selectează celula (dublu click) *Period* din linia asociată intrării de ceas CLK.
- În fereastra *Clock Period* se defineşte semnalul de ceas. În cadrul exemplului prezentat, au fost utilizate valorile deja existente.
- Se închide această fereastră selectând butonul OK.
- Se apasă butonul *Ports* pentru a defini şi celelalte porturi de intrare/ieşire. Spre exemplu, se selectează intrarea *RESET* şi se utilizează butonul dreapta al mouse-ului pentru a alege opţiunea *Pad to Setup* unde se va specifica

comportarea semnalului.

- Semnalele *ledout0* . . . *ledout6* au toate acelaşi comportament. Este indicat să se grupeze aceste semnale într-unul singur *ledout-gr* şi apoi, să se selecteze opţiunea *Clock to Pad,* pentru a specifica comportamentul grupului de semnale *ledout-gr.*

- Se salvează toate modificările făcute şi apoi se părăseşte editorul de constrângeri.

După realizarea fişierului de constrângeri trebuie invocat utilitarul grafic *PACE* care realizează asignarea porturilor de intrare/ieşire prezente în modulul *TOP* cu porturile fizice ale circuitului CPLD. Procesul este foarte simplu şi constă în selectarea intrării/ieşirii dorite şi apoi alegerea pinului corespunzător. Fiecare pin este reprezentat printr-un număr, semnificaţia fiecărui număr regăsindu-se în manualul circuitului oferit de către producător.

Ultimul proces este cel de implementare. Acesta este un proces automat si nu oferă (în cazul proiectării cu circuite CPLD) nici o metodă de a controla subprocesele sale. În urma lui va rezulta un fişier cu extensia ***bit*** care va fi ulterior transferat în circuitul CPLD fizic, testând astfel funcţionarea circuitului proiectat.

2. Desfăşurarea lucrării

1. Se vor proiecta în Verilog şi testa folosind ModelSim toate modulele care alcătuiesc schema funcţională din figura 2.1.

2. Se va implementa şi se va testa schema funcţională prezentată în figura 2.1 utilizând Xilinx WebPACK ISE 6.2i şi ModelSim.

3. Se va implementa fizic schema din figura 2.1 folosind circuitul CPLD XC9500, aflat în dotarea laboratorului.

3. Probleme propuse

1. Să se implementeze o maşină automată de eliberare a băuturilor răcoritoare folosind circuitele FPGA/CPLD existente în laborator. Să se elaboreze un raport cu diferenţele de proiectare observate.

2. Se consideră un numărător binar de 28 biţi. Să se proiecteze un circuit care afişează cei mai semnificativi patru biţi în format hexazecimal utilizând un dispozitiv cu şapte segmente. Pentru implementare se vor utiliza circuitele CPLD/FPGA din dotarea laboratorului.

3. Să se proiecteze şi implementeze folosind circuite CPLD/FPGA, un circuit secvențial care să îndeplinească următoarele cerințe:

- introducerea unei combinații de la tastatură ca o secvență de lungime 24.
- combinația introdusă se memorează ca un cod.
- orice altă combinație introdusă este ignorată (tastatura este blocată).
- introducerea combinației permite deblocarea tastaturii şi setarea unei alte combinații de cod.

4. Să se proiecteze şi implementeze jocul *BLACKJACK* utilizând Xilinx WebPack ISE 6.2i.

3

Proiectarea dispozitivelor aritmetice în virgulă fixă

1. Prezentare teoretică

Scopul acestui laborator este acela de a prezenta sumatorul ***carry look ahead*** precum şi o modalitate de implemetare a operaţiilor de adunare, scădere, înmulţire şi împărţire în virgulă fixă. Reprezentarea numerelor în virgulă fixă este descrisă în standardul IEEE 754 şi se presupune cunoscută.

Cuvintele calculatoarelor sunt compuse din biţi, deci ele pot fi reprezentate ca numere binare. Numărul de biţi din care este compus un cuvânt se alege în funcţie de precizia cu care se doreşte realizarea operaţiilor în calculator. Spre exemplu, un cuvânt de 32 biţi oferă o gamă de reprezentare cuprinsă între 0 şi 2^{32} - l (4.294.967.295).

Numerele într-un calculator pot fi fără semn sau fără semn. În cazul în care un număr este reprezentat fără semn, el se codieră număr pozitiv. Orice calculator foloseşte pentru numerele binare cu semn reprezentarea în cod complementar faţă de 2. Această reprezentare are avantajul că toate numerele negative au l în bitul cel mai semnificativ, ea devenind standardul universal pentru aritmetica de întregi a calculatoarelor.

În cazul în care se lucrează cu numere binare de dimensiuni mari: 32, 64 sau 128 biţi, este recomandată citirea/scrierea lor utilizând o bază mai mare decât baza 2 şi care să asigure ulterior conversia rapidă a numărului într-un număr binar.

Deoarece aproape toate dimensiunile de date de calculator sunt multipli de 4, utilizarea numerelor hexazecimale pentru operaţii de citire/scriere este o soluţie des întâlnită.

Conversia din binar în hexazecimal se realizează mecanic prin înlocuirea fiecărui grup de patru cifre binare printr-o singură cifră hexazecimală şi invers.

Proiectarea unei unități aritmetice / logice

Unitatea Aritmetică/Logică (UAL) este dispozitivul care efectuează operațiile aritmetice şi logice. Operațiile aritmetice şi logice executate de către orice calculator sunt: *adunarea, scăderea, ŞI la nivel de bit, SAU la nivel de bit.*

În calculator, adunarea se efectuează prin adunarea celor două numere bit cu bit de la dreapta la stânga, transportul trecând la următoarea cifră din stânga.

Scăderea foloseşte adunarea: înainte de a fi adunat, scăzătorul este transformat însă în negativul său.

Se va proiecta un circuit care implementeză operații aritmetice şi logice pe 1 bit (UAL de 1 bit), urmând ca acest circuit să fie multiplicat de 32 de ori, pentru a obține o unitate aritmetică/logică, care lucrează cu numere binare de lungime 32 biți (UAL de 32 biți).

Cele mai simple operații de implementat sunt operațiile *ŞI* şi *SAU,* care necesită din punct de vedere al resurselor hardware, doar o poartă *ŞI* cu două intrări, o poartă *SAU* cu două intrări şi un multiplexor, aşa cum se poate observa în figura: 3.1.a.)

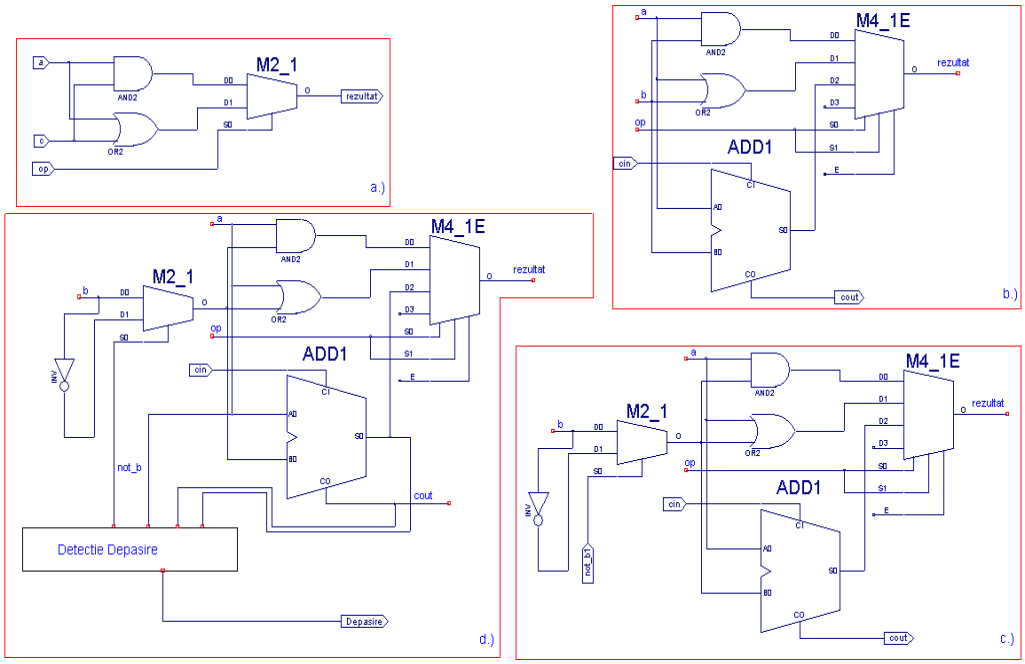


Figura 3.1: UAL de 1 bit care realizează operațiile *ŞI* şi *SAU.*

Următoarea operație care va fi implementată este operația de *adunare.* Pentru aceasta se va utiliza un sumator de 1 bit care va fi adăugat logicii deja existente. Unitatea aritmetică logică de 1 bit care realizează operațiile: *ŞI*, *SAU* şi *adunare* este prezentată în figura 3.1.b.)

Pentru implementarea operației de scădere mai trebuie adăugat un multiplexor şi un inversor resurselor hardware deja existente aşa cum se poate observa în figura 3.1.c. Conform acestei figuri, dacă semnalul *not_b* = 1 şi semnalul *cin* = 1, atunci se obține scăderea în cod complementar față de 2 a lui *b* din *a.*

Tot în figura 3.1 c.) se poate observa că, pentru realizarea operațiilor logice sau al operației de adunare, este suficient ca semnalele *not_b* şi *cin* să fie 0. Din această cauză se pot combina aceste două semnale într-o singură linie de control numită *not_b1,* ca în figura 3.1.d.)

În cazul în care se adună doi operanzi cu acelaşi semn pot să apară cazuri de depăşire. Acestea trebuie evidențiate pentru a elimina situațiile în care rezultatul obținut poate fi citit/interpretat eronat. O verificare simplă a depăşirii la adunare este să se constate dacă semnalul *cin* spre cel mai semnificativ bit este diferit de semnalul *cout* de la cel mai semnificativ bit. Schema finală pentru un UAL de 1 bit în care sunt semnalate situațiile de posibilă depăşire este prezentată în figura 3.1.d.)

Un circuit UAL de 32 de biți se realizează printr-o înlănțuire de 32 de unități UAL de 1 bit. În cazul în care se doreşte introducerea instrucțiunilor de ramificație condiționată, circuitului UAL de 32 biți i se mai adaugă un inversor şi o poartă logică *OR* cu 32 de intrări, aşa cum se poate observa în figura 3.2.

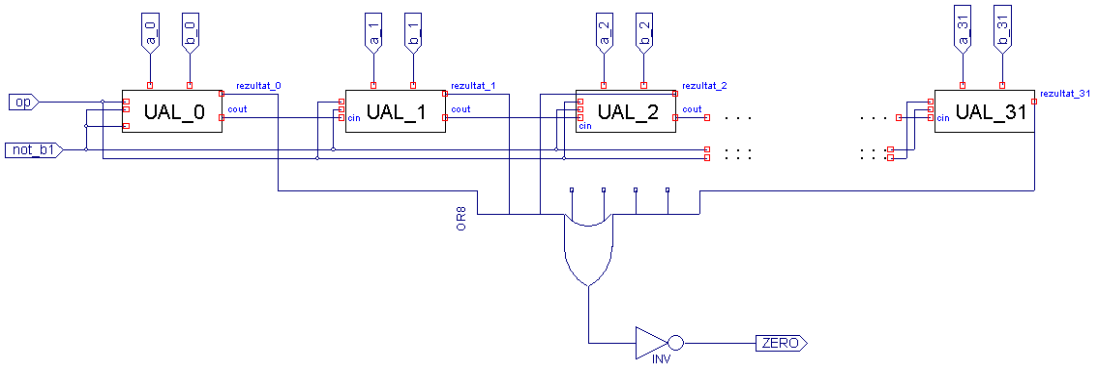


Figura 3.2: UAL de 32 biți.

Aceste instrucțiuni determină o ramificare a controlului, fie dacă două registre sunt egale în conținut, fie dacă ele sunt diferite. Verificarea egalității cu ajutorul circuitului UAL se realizează prin scăderea lui *b* din *a* şi apoi testarea rezultatului cu zero. Astfel, dacă se adaugă hardware pentru a se verifica dacă rezultatul este 0, se poate testa egalitatea. Metoda cea mai simplă de calcul presupune efectuarea unui *SAU* între toate ieşirile, apoi acest semnal să fie scos printr-un inversor.

$$ZERO = \overline{rezultat_{31} + rezultat_{30} + ... + rezultat_1 + rezultat_0} \quad (3.1)$$

Sumatorul cu anticiparea transportului

În vederea determinării factorilor *generare* şi *propagare* transport se porneşte de la ecuația care determină transportul de intrare pentru sumatorul 1.

$$cin_1 = b_0 \cdot cin_0 + a_0 \cdot cin_0 + a_0 \cdot b_0 \quad (3.2)$$

Factorizând ecuația 3.2, se obține următoarea ecuație:

$$c_{i+1} = b_i \cdot c_i + a_i \cdot c_i + a_i \cdot b_i = a_i \cdot b_i + c_i \cdot (a_i + b_i) \quad (3.3)$$

Dacă în această ecuație se notează termenul $a_i \cdot b_i$ cu g_i şi $a_i + b_i$ cu p_i atunci ecuația 3.3 devine:

$$c_{i+1} = g_i + p_i \cdot c_i \quad (3.4)$$

Termenii g_i şi p_i se numesc *generare transport* şi *propagare transport.* Dacă în ecuația 3.4 se presupune că g_i = l, atunci rezultă că c_{i+1} = 1. Aceasta înseamnă că sumatorul generează un transport către sumatorul următor, independent de valoarea intrării *cin.* Anolog, dacă se presupune în ecuația 3.4, g_i = 0 şi p_i = 1 atunci rezultă că $c_{i+1} = c_i$, adică sumatorul propagă transportul de la intrare la ieşire.

Exemplu Să se determine ecuțiile pentru un sumator cu anticiparea transportului care realizează suma a două numere binare de 4 biți.

Soluție*:* Ecuațiile sunt următoarele:

$$\begin{aligned} c_1 &= g_0 + p_0 \cdot c_0 \\ c_2 &= g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \\ c_3 &= g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \\ c_4 &= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0 \end{aligned} \quad (3.5)$$

Aşa cum se observă şi în exemplul prezentat, această formă simplificată conduce la un volum considerabil de circuite logice chiar şi pentru un sumator de 16 biți. Pentru a înlătura acest inconvenient este nevoie să se treacă la un nivel superior de abstractizare.

Pentru a mări viteza, în cazul folosirii unui sumator de 16 biți, este necesară efectuarea anticipării transportului cu sumatoare de 4 biți. Noile semnale, *P* şi G, vor semnifica propagarea şi generarea transportului la nivel de bloc. Ecuațiile la nivel de bloc pentru transportul

de intrare al fiecărui grup de 4 biți al sumatorului de 16 biți sunt prezentate în ecuațiile 3.6, dar ele sunt asemănătoare cu ecuațiile 3.5.

$$\begin{aligned} C_1 &= G_0 + P_0 \cdot c_0 \\ C_2 &= G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot c_0 \\ C_3 &= G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot c_0 \\ C_4 &= G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot c_0 \end{aligned} \quad (3.6)$$

unde:

$$\begin{aligned} P_0 &= p_3 \cdot p_2 \cdot p_1 \cdot p_0 \\ P_1 &= p_7 \cdot p_6 \cdot p_5 \cdot p_4 \\ P_2 &= p_{11} \cdot p_{10} \cdot p_9 \cdot p_8 \\ P_3 &= p_{15} \cdot p_{14} \cdot p_{13} \cdot p_{12} \\ G_0 &= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 \\ G_1 &= g_7 + p_7 \cdot g_6 + p_7 \cdot p_6 \cdot g_5 + p_7 \cdot p_6 \cdot p_5 \cdot g_4 \\ G_2 &= g_{11} + p_{11} \cdot g_{10} + p_{11} \cdot p_{10} \cdot g_9 + p_{11} \cdot p_{10} \cdot p_9 \cdot g_8 \\ G_3 &= g_{15} + p_{15} \cdot g_{14} + p_{15} \cdot p_{14} \cdot g_{13} + p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12} \end{aligned} \quad (3.7)$$

Implementarea hardware este prezentată în figura 3.3.

Exemplu Să se determine valorile g_i, p_i, P_i şi G_i ale următoarelor două numere de 16 biți: a = 0001 1010 0011 0011 şi b = 1110 0101 1110 1011. Să se determine şi valoarea termenului C_4.

Soluție *:* Se determină întâi valorile $g_i = a_i \cdot b_i$ şi $p_i = a_i + b_i$.

a: 0001 101000110011

b: 11100101 1110 1011

g_i : 0000000000100011

p_i : 1111 1111 1111 1011

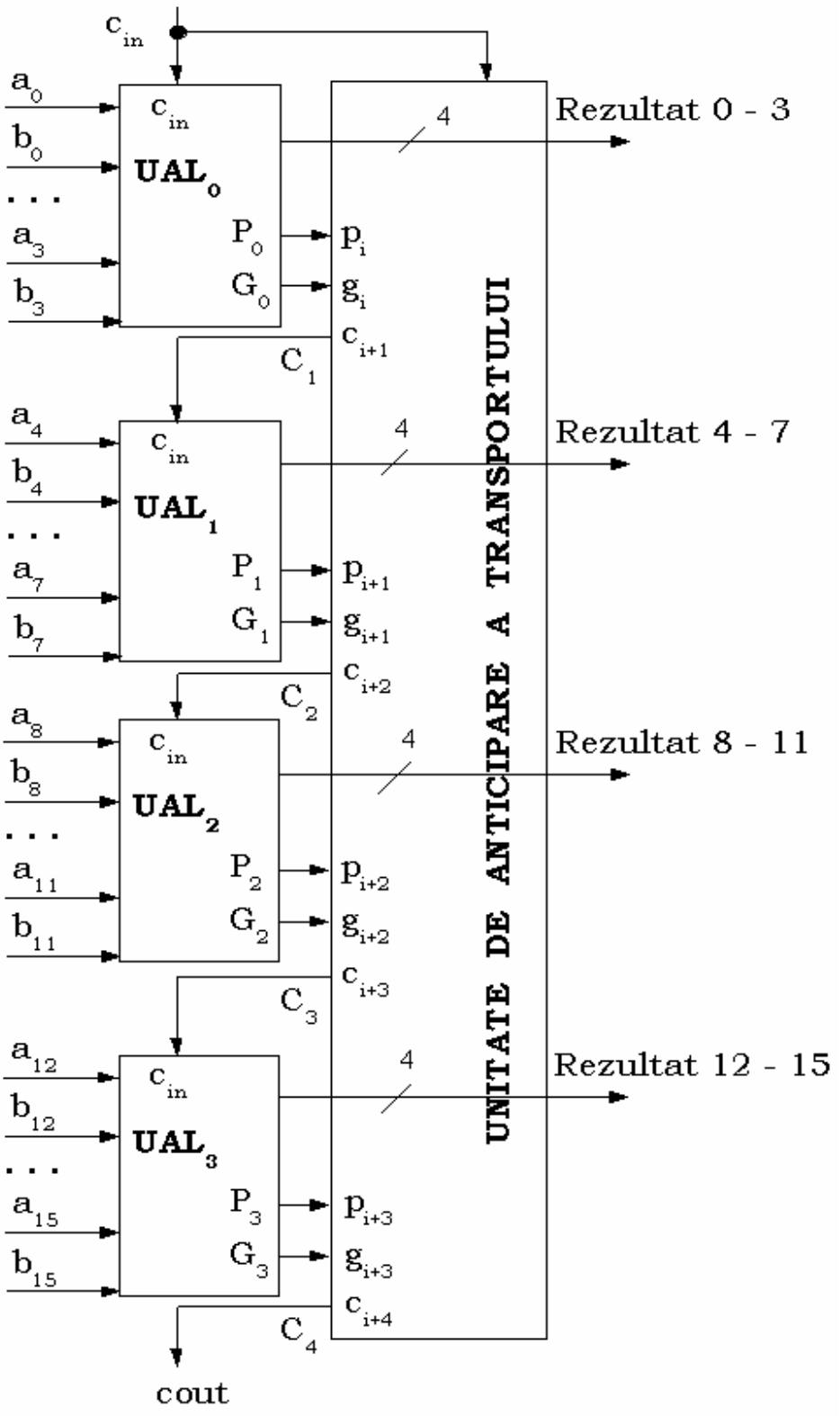


Figura 3.3: Sumator pe 16 biți

Determinarea semnalelor P_3, P_2, P_1 şi P_0 sunt simple operații *ŞI* între propagările nivelului inferior.

$$
\begin{aligned}
P_3 &= 1\cdot 1\cdot 1\cdot 1 = 1\\
P_2 &= 1\cdot 1\cdot 1\cdot 1 = 1\\
P_1 &= 1\cdot 1\cdot 1\cdot 1 = 1\\
P_0 &= 1\cdot 0\cdot 1\cdot 1 = 0\\
G_0 &= 0+(1\cdot 0)+(1\cdot 0\cdot 1)+(1\cdot 0\cdot 1\cdot 1) = 0+0+0+0 = 0\\
G_1 &= 0+(1\cdot 0)+(1\cdot 1\cdot 1)+(1\cdot 1\cdot 1\cdot 0) = 0+0+1+0 = 1\\
G_2 &= 0+(1\cdot 0)+(1\cdot 1\cdot 0)+(1\cdot 1\cdot 1\cdot 0) = 0+0+0+0 = 0\\
G_3 &= 0+(1\cdot 0)+(1\cdot 1\cdot 0)+(1\cdot 1\cdot 1\cdot 0) = 0+0+0+0 = 0\\
C_4 &= 0+(1\cdot 0)+(1\cdot 1\cdot 1)+(1\cdot 1\cdot 1\cdot 0)+(1\cdot 1\cdot 1\cdot 0\cdot 0) = 0+0+1+0+0 = 1
\end{aligned}
\qquad (3.8)
$$

Se observă că la adunarea numerelor a şi b de 16 biți există un transport de ieşire.

Algoritmul de înmulțire al lui Booth

Acest algoritm oferă o modalitate de a înmulți două numerelor binare întregi cu semn folosind reprezentarea cod complementar față de 2.

Algoritmul exploatează faptul că secvențele de 0 din înmulțitor necesită doar deplasare, iar secvențele de l din cadrul înmulțitorului, cuprinse între rangurile $2^k - 2^m$, pot fi tratate ca $2^{k+1} - 2^m$.

Exemplu. Se consideră numărul binar 001 110 (+14). Acest număr conține o secvență de l între pozițiile 2^3 şi 2^1. Rezultă $k = 3$ şi $m = l$. Numărul poate fi reprezentat ca: $2^{k+1} - 2^m = 2^4 - 2^1 = 16 - 2 = 14$.

Înmulțirea M × 14 (unde M reprezintă înmulțitorul iar 14 deînmulțitul) poate fi realizată ca: M x 2^4 - M × 2^1. Se observă că produsul poate fi obținut prin deplasarea înmulțitorului binar M de 4 ori la stânga şi scăderea lui M deplasat stânga, odată.

Algoritmul lui Booth necesită examinarea biților înmulțitorului şi deplasarea produsului parțial. Înainte de a deplasa produsul parțial, înmulțitorul poate fi adunat/scăzut la/din produsul parțial conform regulilor:

1. $Q_n = 0$, $Q_{n+1} = 0$ sau $Q_n = l$, $Q_{n+1} = l$ - produsul parțial nu se modifică;
2. $Q_n = l$, $Q_{n+1} = 0$ - se scade înmulțitorul din produsul parțial;
3. $Q_n = 0$, $Q_{n+1} = l$ - se adună înmulțitorul la produsul parțial.

În figura 3.4 sunt prezentate resursele hardware necesare implementării acestui algorithm precum şi biții Q_n şi Q_{n+1}.

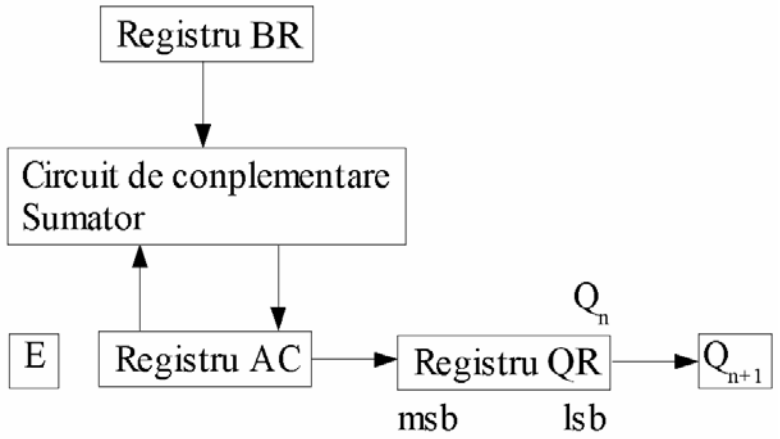


Figura 3.4: Resursele hardware necesare implementării algoritmului lui Booth.

Semnificația resurselor folosite în figura 3.4 este următoarea:

- **Registru BR** – registru care menține valoarea înmulțitorului;
- **E** – registru de 1 bit folosit pentru memorarea valorii transportului rezultat în urma operației de adunare;
- **Registru AC** – registru auxiliar de lucru;
- **Registrul QR** – registru care menține valoarea deînmulțitului

Q_n reprezintă cel mai puțin semnificativ bit al deînmulțitului memorat în QR. Un registru suplimentar Q_{n+1} este adăugat la QR pentru a facilita inspectarea celor 2 biți ai deînmulțitului. Acest registru este inițializat cu 0.

Un exemplu numeric pentru algoritmul lui Booth este prezentat în tabelul 3.1. Algoritmul lui Booth este prezentat sub formă de organigramă în figura 3.5. Exemplul numeric prezentat presupune că înmulțitorul are valoarea -9 şi deînmulțitul are valoarea -13 (-9 x -13).

			11001			
		ashr	11100	10110	0	010
0	0	ashr	11110	01011	0	001
1	0	se scade BR	01001			
			00111			
		ashr	00011	10101	1	000
0	1	se aduna BR	10111			

Tabelul 3.1: Algoritmul lui Booth. Exemplu. **ashr** = deplasare aritmetică la dreapta

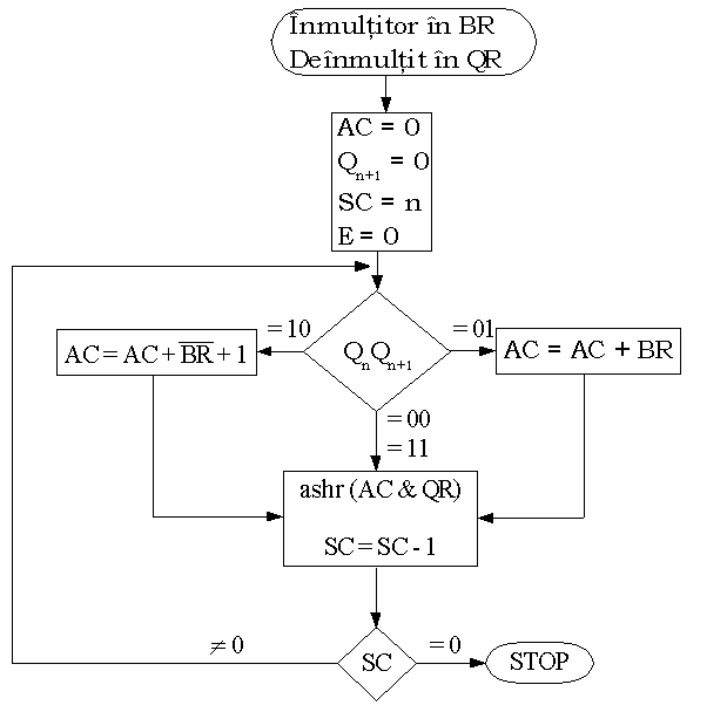


Figura 3.5: Algoritmul lui Booth de înmulțire a două numere binare cu semn.

Împărțirea numerelor binare în virgulă fixă

Uzual, operația de împărțire este realizată folosind comparări succesive, deplasări şi operații de scădere. Împărțirea binară este mai simplă decât împărțirea zecimală, deoarece

biții care formează câtul sunt 1 sau 0 şi de aceea nu este necesar să se estimeze de câte ori deîmpărțitul sau restul parțial este cuprins în împărțitor.

Resursele hardware necesare implementării operației de împărțire sunt cele prezentate în figura 3.6. Algoritmul hardware de împărțire a două numere binare în virgulă fixă este prezentat sub formă de organigramă în figura 3.7.

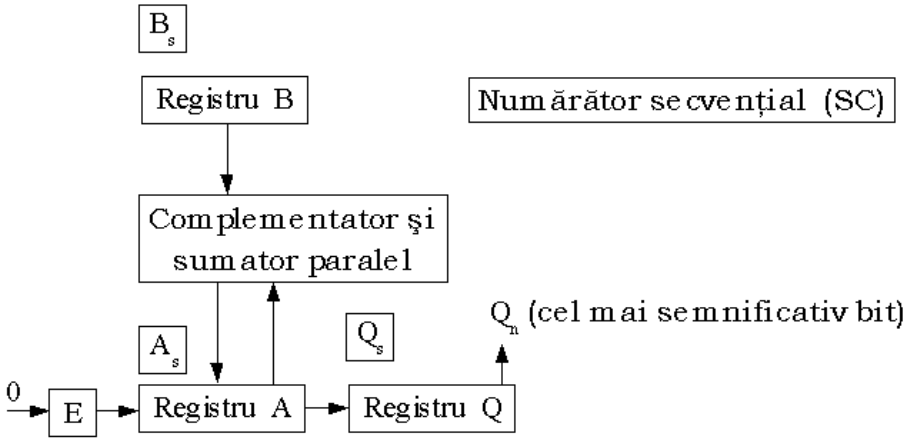


Figura 3.6: Resursele hardware necesare operației de împărțire.

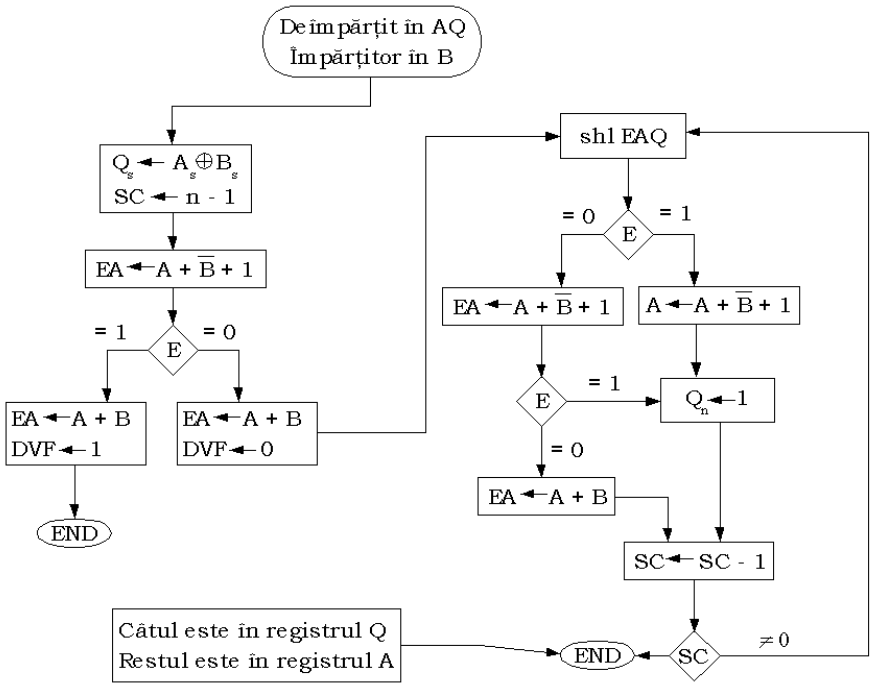


Figura 3.7: Organigrama operației de împărțire.

2. Desfăşurarea lucrării

1. Se va implementa în Verilog şi simula folosind simulatorul ModelSim o unitate aritmetică logică cu dimensiunea de 32 biți. Schema bloc este prezentată în figura 3.2.

2. Se va proiecta, implementa în Verilog şi testa funcționarea unui sumator cu transport anticipat (***carry look ahead***), care operează cu numere de 8 sau 32 de biți.

3. Se va implementa în Verilog şi testa cu ajutorul simulatorului ModelSim algoritmul de înmulțire al lui Booth.

4. Se va implementa în Verilog şi testa cu ajutorul simulatorului ModelSim algoritmul de împărțire prezentat în figura 3.7

3. Probleme propuse

1. Să se obțină negativul lui 2_{10} şi apoi să se verifice rezultatul. Numărul este reprezentat pe 32 de biți.

2. Să se convertească versiunea binară de 16 biți a lui 2_{10} şi - 2_{10} în numere binare de 32 biți.

3. Să se convertească următoarele numere haxazecimale şi binare în cealaltă bază: $eca86420_{16}$ şi $0001\ 0011\ 0101\ 0111\ 1001\ 1011\ 1101\ 1111_2$.

4. Să se scrie ecuația pentru logica transportului anticipat la un sumator cu 64 biți, utilizând sumatoare de 16 biți ca blocuri hardware de bază. Să se includă în soluție un desen asemănător cu cel prezentat în figura 3.3.

5. Formulați o procedură hardware pentru detecția depăşirii (overflow) prin compararea semnului sumei cu semnele celor două numere binare. Numerele sunt reprezentate în cod complement față de 2.

6. Elaborați un algoritm sub formă de organigramă pentru adunarea/scăderea a două

numere binare în virgulă fixă, dacă numerele negative sunt reprezentate în cod complement fată de l.

7. Demonstrați că produsul a două numere, fiecare având *n* biți, într-o bază *r* are drept rezultat un număr binar a cărui lungime nu este mai mare de *2n* biți. Demonstrați că afirmația de mai sus conduce la concluzia că în cazul operației de înmulțire nu putem avea depăşire.

8. Arătați conținutul registrelor E, A, Q şi SC (ca în tabelul 3.1) dacă înmulțitorul este 10101 iar deînmulțitul este 11111. Semnele nu sunt incluse.

9. Să se verifice cu ajutorul algoritmului lui Booth următoarea egalitate: 0010 x 1101 = 1111 1010.

4

Reprezentarea datelor. Operații aritmetice ZCB (Zecimal Codificat Binar).

1. Prezentare teoretică

În cadrul acestui laborator se vor prezenta modalitățile de reprezentare a numerelor în calculatoarele numerice, precum şi realizarea operațiilor de adunare şi scădere a numerelor ZCB.

În calculatoarele numerice, informația memorată în format binar se regăseşte în memorie sau în registrele procesorului. Registrele conțin date sau informații de control. Informația de control este formată dintr-un grup de biți utilizat în vederea specificării secvenței semnalelor de comandă necesare manipulării datelor în registre. Datele sunt numere şi alte informații codificate binar, care sunt prelucrate în vederea obținerii unui rezultat numeric.

Toate tipurile de date, cu excepția numerelor binare, sunt reprezentate în registrele calculatorului în formă binară, deoarece acestea sunt alcătuite din bistabile, care pot memora informație 0 sau l.

Reprezentarea numerelor

Un sistem de numerație, care utilizează baza ***r*** este un sistem, care foloseşte simboluri distincte pentru cifrele ***r***. Fiecărei cifre îi este asignată un simbol unic. Şirurile prin care se reprezintă numerele sunt compuse din simboluri. Pentru a determina cantitatea reprezentată de un număr este necesar să se înmulțească fiecare cifră cu un întreg putere a lui *r* şi apoi să se formeze suma tuturor cifrelor ponderate.

Sistemul de numerație ***zecimal*** este alcătuit din zece simboluri: 0, 1, 2, 3, 4, 5, 6, 7, 8 şi 9. Şirul de cifre 126.3 este interpretat ca: $l \times 10^2 + 2 \times 10^1 + 6 \times 10° + 3 \times 10^{-1}$.

Sistemul de numerație ***binar*** utilizează baza 2 rezultând astfel numai două simboluri 0 şi l, utilizate în cadrul acestui sistem de numeratie. Şirul de biți 101111 este interpretat ca: $l \times 2^5 + 0 \times 2^4 + l \times 2^3 + l \times 2^2 + l \times 2^1 + l \times 2^0$.

Sistemul de numerație ***octal*** utilizează baza opt şi este alcătuit din următoarele opt simboluri: 0, l, 2, 3, 4, 5, 6 şi 7. Şirul de cifre 736 este interpretat ca $(478)_{10}$ si $(111011110)_2$.

Sistemul de numerație ***hexazecimal*** utilizează baza 16 şi este alcătuit din următoarele simboluri: 0, l, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E şi F. Şirul de cifre F3 este interpretat $(243)_{10}$ şi $(11110011)_2$.

Sistemul de numerație binar este cel mai natural sistem pentru calculator (aşa cum s-a subliniat, circuitele bistabile pot memora doar 0 şi 1), în timp ce pentru utilizator, sistemul de numerație zecimal este considerat ca sistem natural. O metodă de soluționare a acestui conflict o constituie convertirea tuturor intrărilor din format zecimal în format binar, realizarea tuturor operațiilor aritmetice în binar şi apoi convertirea rezultatului în format zecimal.

O altă metodă este utilizarea numerelor ***binare codificate zecimal (ZCB).*** Este foarte important să nu se confunde *conversia* numerelor zecimale în binar si *codificarea binară* a numerelor zecimale. Când se realizează *conversia* numărului zecimal 99 într-un număr binar se obține șirul de biți 1100011, reprezentarea ZCB fiind 10011001. Singura diferență constă în simbolurile utilizate pentru reprezentarea biților.

În tabelul 4.1 se poate observa codificarea numerelor zecimale în diferite sisteme de numerație.

Codurile complementare

Reprezentarea în cod complementar este utilizată în calculatoarele numerice pentru simplificarea operației de scădere şi pentru manipulări logice. Există două tipuri de coduri complementare pentru fiecare bază ***r*** folosită: cod complementar față de ***r*** şi cod complementar față de ***r - 1.*** Dacă ***r*** este substituit cu 2 atunci se obține cod complementar față de 2 şi cod complementar față de 1. Analog dacă ***r*** este substituit cu 10.

Cod complement față de r - 1

Fie dat numărul N în baza r având n ranguri. Reprezentarea numărului N în cod complementar față de r - l este definită ca $(r^n - 1) - N$. În cazul numerelor zecimale, $r = 10$ şi $r - 1 = 9$, reprezentarea numărului N în cod complementar față de 9 este $(10^n - 1) - N$.

10^n reprezintă un număr care conține un singur l urmat de un număr de n digiți 0. $10^n - 1$ este un număr reprezentat de n digiți 9. Dacă $n = 4$, spre exemplu, $10^4 = 10000$ şi $10^4 - 1 = 9999$. Concluzia care se impune este următoarea: complementul față de 9 al unui număr zecimal este obținut prin scăderea fiecărui digit din 9. Ca exemplu, se consideră numărul $N = 546700$ dat. Reprezentarea numărului N în cod complementar față de 9 este: 999999 - 546700 = 453299.

Dacă substituim ***r*** cu 2 atunci $r - 1 = 1$, iar numărul N reprezentat în cod complementar față de l este $(2^n - 1) - N$. Raționând ca în cazul complementului față de 9, se ajunge la concluzia următoare: complementul față de l al unui număr binar este obținut scăzând fiecare bit din 1.

Număr hexazecimal	**Hexazecimal codificat binar**	**Număr octal**	**Octal codificat binar**	**Număr ZCB**	**Număr zecimal**
0	0000	0	000	0000	0
1	0001	1	001	0001	1
2	0010	2	010	0010	2
3	0011	3	011	0011	3
4	0100	4	100	0100	4
5	0101	5	101	0101	5

6	0110	6	110	0110	6
7	0111	7	111	0111	7
8	1000			1000	8
9	1001			1001	9
A	1010				10
B	1011				11
C	1100				12
D	1101				13
E	1110				14
F	1111				15

Tabelul 4.1: Reprezentarea numerelor zecimale în diferite sisteme de numerație.

Când se efectuează scăderea se observă că modificarea biților este 0 în l sau l în 0. Astfel, se poate concluziona: complementul față de l al unui număr binar se obține prin inversarea biților numărului *N*. Spre exemplu, reprezentarea numărului *N* = 1011001 în cod complementar față de l este 0100110. Dacă ***r*** este substituit cu 8 sau 16 atunci se scade fiecare digit al numărului *N* din 7 respectiv F.

Complementul față de r

Complementul față de r al unui număr de *n* digiți, *N,* în baza ***r*** este definit ca r^n - *N* în cazul în care $N \neq 0$ şi 0 dacă *N* = 0. Comparând cu complementul față de ***r - 1***, se observă că, complementul față de ***r*** este obținut prin adăugarea lui l la complementul față de ***r - 1***. Astfel, complementul față de 10 al numărului zecimal *N* = 2389 este 7610 + l = 7611, iar complementul față de 2 al numărului binar *N* = 101100 este 010011 + l = 010100.

Complementul față de 10 poate fi însă format şi aplicând următoarea procedură:

- rămân nemodificate toate cifrele zero cele mai puțin semnificative;
- se scade prima cifră diferită de zero (cea mai puțin semnificativă) din 10;
- se scad celelalte cifre până la cifra cea mai semnificativă.

Conform procedurii, reprezentarea numărului *N* = 246700 în cod complementar față de 10 este 753300. În mod similar se procedează în cazul reprezentării în cod complementar față de 2. Se păstrează toți biții cei mai puțin semnificativi neschimbați dacă valoarea lor este 0, precum şi primul bit cel mai puțin semnificativ care are valoarea 1. Se înlocuiesc toți ceilalți biți cu l dacă valoarea lor este 0 si cu 0 dacă valoarea lor este 1. Numărul binar *N* = 1101100 este reprezentat în cod complement față de 2 ca 0010100.

Alte coduri binare

În calculatoarele numerice precum şi în diverse aplicații este necesar să se utilizeze alte coduri precum codul Gray, codul 2421, codul excess-3 Gray, etc. Acestea sunt prezentate în tabelele 4.2 şi 4.3.

Cod Gray	Număr zecimal echivalent	Cod Gray	Număr zecimal echivalent
0000	0	1100	8
0001	1	1101	9
0011	2	1111	10
0010	3	1110	11
0110	4	1010	12
0111	5	1011	13
0101	6	1001	14
0100	7	1000	15

Tabelul 4.2: Codul Gray.

Digital zecimal	BCD 8421	2421	Excess-3	Excess-3 Gray
0	0000	0000	0011	0010
1	0001	0001	0100	0110
2	0010	0010	0101	0111
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1100
6	0110	1100	1001	1101
7	0111	1101	1010	1111
8	1000	1110	1011	1110
9	1001	1111	1100	1010

Tabelul 4.3: Diferite coduri pentru cifrele zecimale.

Operații aritmetice ZCB

Existența unei astfel de unități este justificată de faptul ca utilizatorul oricărui calculator oferă datele de intrare în format zecimal şi se aşteaptă ca şi rezultatul afişat să fie tot în format zecimal. Pentru calculatoarele numerice, soluția adoptată este de a converti datele de intrare în date binare, de a efectua toate operațiile solicitate în mod binar şi apoi de a converti rezultatul în format zecimal pentru a fi vizualizat de către utilizator.

Toate calculatoarele numerice au în componența lor o unitate aritmetică zecimală deoarece intrările si ieşirile sunt foarte frecvente. Există însă si multe calculatoare numerice, care au în componența lor hardware pentru calcule aritmetice în care datele sunt

reprezentate în format binar si zecimal. Utilizatorul poate specifica în acest caz cum să se efectueze calculele, binar sau zecimal.

Sumator ZCB

Considerăm adunarea a două numere ZCB împreună cu un posibil transport din stagiul anterior. Fiecare cifră care compune un număr ZCB nu poate depăşi valoarea 9, deci suma a două cifre nu poate depăşi valoarea 9 + 9 + 1 = 19, unde l reprezintă transportul din etajul/rangul anterior.

Pentru realizarea sumei celor două numere ZCB, există sumatoare binare de 4 biţi. Sumatorul va forma suma în *binar* şi valoarea ei se va încadra în rangul 0 ... 19. Numerele binare sunt prezentate în tabelul 4.4, folosind notaţiile K, Z_8, Z_4, Z_2 şi Z_1, unde *K* reprezintă transportul iar indicii literei *Z* sunt ponderile 8, 4, 2 şi l, asignate celor 4 biţi în codul ZCB.

Sumă binară					**Sumă ZCB**					**Zecimal**
K	**Z_8**	**Z_4**	**Z_2**	**Z_1**	**C**	**S_8**	**S_4**	**S_2**	**S_1**	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

Tabelul 4.4: Determinarea sumatorului ZCB.

Examinând conținutul tabelului 4.4, se deduce că atunci când suma binară este mai mică sau egală decât 1001, numărul ZCB corespunzător este identic; nici o conversie nu este necesară. Atunci când suma este mai mare decât 1001, se obține un număr invalid în reprezentare ZCB.

Procedura de adunare a două numere ZCB implică utilizarea unui sumator binar (la fiecare moment de timp este realizată adunarea a câte unei cifre a numărului ZCB) şi este următoarea:

1. Biții cei mai puțin semnificativi ai celor două numere ZCB sunt însumați rezultând o sumă binară;

2. Dacă suma obținută la pasul l este egală sau mai mare decât 1010, ea este corectată prin adăugarea valorii 0110;

3. Operația realizată în pasul 2 va genera automat un transport către pasul 4;

4. Următoarea pereche de biți ZCB este însumată împreună cu transportul generat la pasul 3;

5. Dacă rezultatul obținut în pasul 4 este mai mare sau egal cu 1010, atunci el se corectează prin însumare cu valoarea 0110;

6. Se repetă paşii 1 - 5 până când ultimile perechi de biți ZCB sunt adunate.

Circuitul logic care detectează corecția necesară este determinat pe baza tabelului 4.4. O corecție este necesară atunci când $K = 1$. Condiția pentru realizarea corecției poate fi exprimată ca o funcție booleana aşa cum se poate observa în ecuația 4.1.

$$C = K + Z_8 \cdot Z_4 + Z_8 \cdot Z_2 \qquad (4.1)$$

Un sumator ZCB care adună doi operanzi ZCB (adunarea biților ZCB este realizată în paralel) şi care include logica de corecție este prezentat în figura 4.1.

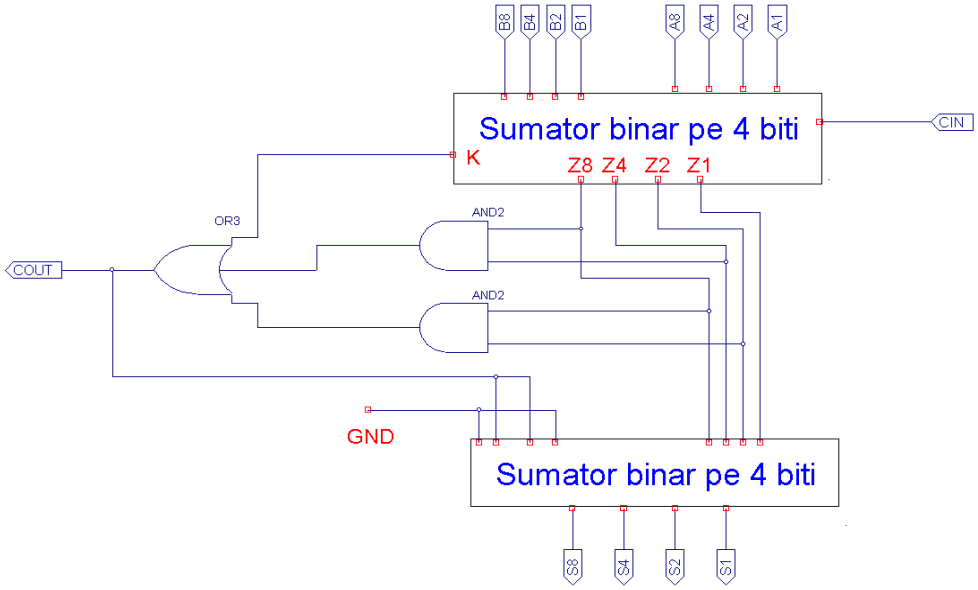


Figura 4.1: Schema bloc a sumatorului ZCB.

Adunarea şi scăderea ZCB

Datele zecimale pot fi adunate în trei moduri diferite aşa cum se poate observa în figura 4.2.

Metoda paralelă utilizează un număr de sumatoare ZCB egal cu numărul de cifre care formează numărul ZCB. Suma este realizată în paralel şi necesită doar o operație elementară.

În cadrul metodei 2, cifrele sunt aplicate serial la intrările unui singur sumator ZCB, cât timp biții pentru fiecare cifră codificată sunt transferate în paralel. Suma este formată prin deplasarea numerelor prin sumatorul zecimal la fiecare moment de timp. Dacă numărul ZCB este format din *k* cifre, atunci vor fi necesare *k* operații elementare, una pentru fiecare deplasare zecimală.

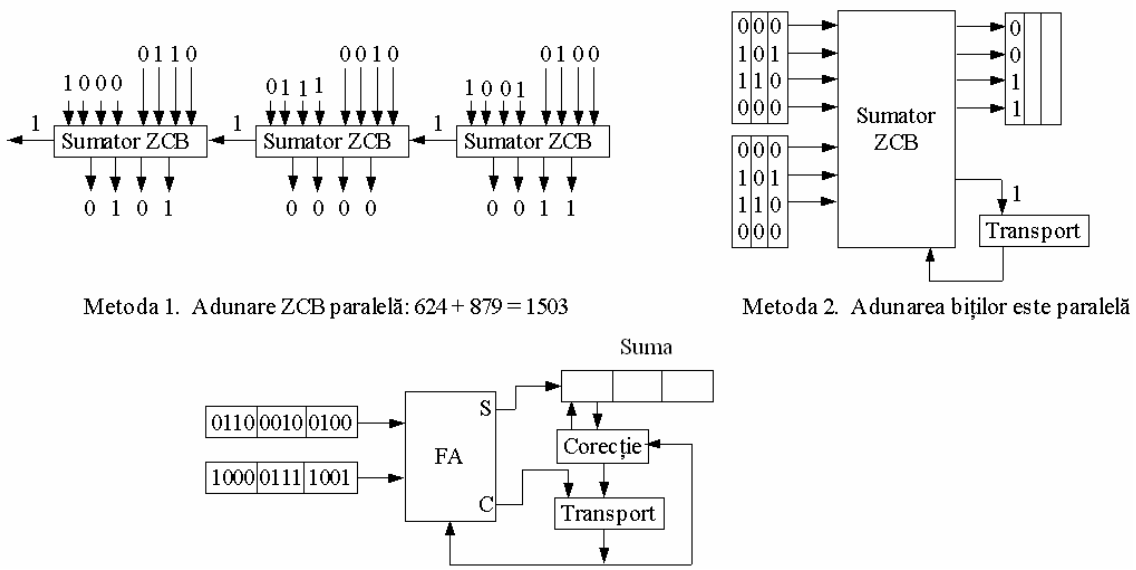


Figura 4.2: Modalități de adunare ZCB

În cadrul metodei 3, biții sunt deplasați câte unul la fiecare moment de timp prin sumatorul complet. Suma binară obținută după patru deplasări trebuie corectată pentru a deveni o cifră valid ZCB validă.

Scăderea ZCB se realizează tot prin intermediul adunării deoarece $A - B = A + \overline{B} + 1$, unde $\overline{B}$ înseamnă complementul lui B față de *9*.

2. Desfăşurarea lucrării

1. Se va implementa în Verilog şi simula schema bloc prezentată în figura 4.1 folosind Xilinx WebPACK ISE 5.1i şi simulatorul ModelSim.

2. Se vor implementa în Verilog şi se vor simula cele trei circuite de adunare ZCB prezentate în figura 4.2. Rezultatele obținute (timpii de funcționare) vor fi comparați şi se va stabili care dintre metode este cea mai eficientă.

3. Probleme propuse

1. Determinați o procedură hardware pentru detectarea apariției unei depășiri prin compararea semnului sumei cu semnele celor doi operanzi. Numerele sunt reprezentate în cod complementar față de 2.

2. Elaborați un algoritm sub formă de diagramă pentru adunarea şi scăderea a două numere binare dacă numerele negative sunt reprezentate în cod complementar față de 1.

3. Să se proiecteze şi să se simuleze utilizând Xilinx WebPACK ISE 6.2i un circuit, care realizează conversia din codul Gray în codul binar şi invers.

4. Să se proiecteze şi să se simuleze utilizând Xilinx WebPACK ISE 6.2i un circuit, care realizează conversia din codul BCD 8421 în codul binar şi invers.

5. Să se proiecteze şi să se simuleze utilizând Xilinx WebPACK ISE 6.2i un circuit, care realizează conversia din codul 2421 în codul binar şi invers.

6. Să se proiecteze, simuleze şi să se implementeze un sumator ZCB folosind Xilinx WebPack 6.2i.

7. Utilizând circuite combinaționale, să se implementeze şi simuleze folosind Xilinx WebPack 6. 2i un circuit complement ZCB față de 9.

8. Utilizând circuite combinaționale, să se implementeze şi simuleze folosind Xilinx WebPack 6.2i un circuit complement ZCB față de 9. Digiții zecimali sunt reprezentați în cod excess-3.

9. Proiectați circuitele hardware necesare operației de adunare/scădere a numerelor ZCB dacă numerele negative sunt reprezentate în cod complement față de 10. Indicați o modalitate de detecție a situației de depășire.

5

Unităţi de execuţie şi comandă integrate. AMD 2901 şi AMD 2909

1. Prezentare teoretică

Unităţile de execuţie se prezintă sub forma unor circuite integrate pe scară medie/largă. Ele sunt structurate pe un anumit număr de biţi astfel încât prin concatenarea lor şi utilizarea unor circuite adiţionale, se pot realiza sisteme de prelucrare pentru date organizate pe 4, 8, 16,24,32,48 sau 64 de biţi.

Unităţile de execuţie integrate s-au comercializat în asociaţie cu unităţile de comandă corespunzătoare şi cu o serie de circuite adiţionale, formând microprocesoarele pe tranşe de biţi, microprocesoarele "bit-slice", microprocesoarele "multi-chip" etc. Printre cele mai răspândite familii de microprocesoare "bit-slice" s-au aflat şi cele produse de compania ***Advanced Micro Devices*** sub numele de AMD 2900.

AMD 2901

Unitatea de execuţie AMD 2901 este organizată pe tranşe de 4 biţi/circuit şi este prevăzută cu elementele necesare cuplării în cascadă.

Semnalele de comandă se aplică sub forma unor vectori binari la terminalele circuitului, fiind, de regulă, preluate sub controlul unui circuit micro-secvențiator integrat (AMD 2909, 2911) de la o memorie cu conținut permanent.

În figura 5.1 se prezintă schema bloc a circuitului AMD 2901.

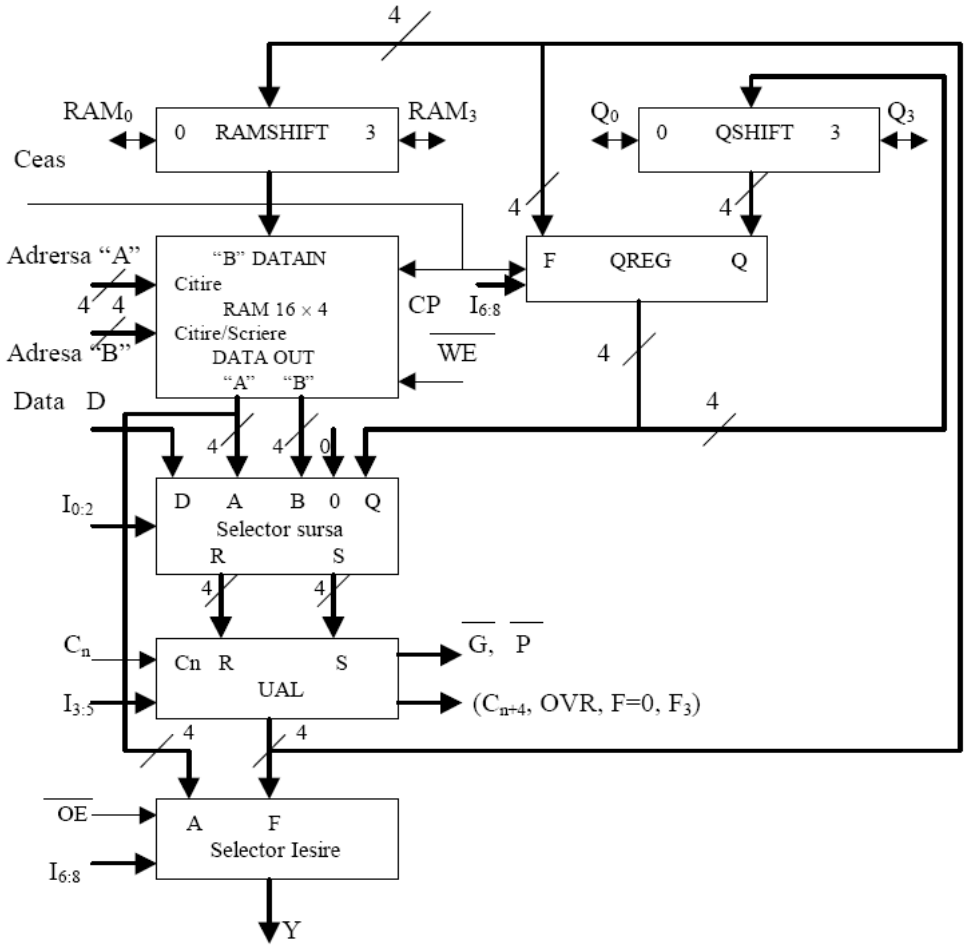


Figura 5.1: AMD 2901 - schemă bloc.

Semnificația resurselor hardware prezente în figura 5.1 este următoarea:

- un ansamblu de 16 registre generale de câte 4 biți, organizate sub forma unei memorii(RAM) biport, cu două intrări de adrese, o intrare de date şi două ieşiri de date;

- o unitate aritmetică-logică, cu transport anticipat, capabilă să efectueze 3 operații aritmetice binare şi 5 operații logice şi să genereze, atât indicatorii de condiții: depăşire, zero, semn, transport (C_{n+4}), cât şi condițiile de propagare ($\overline{P}$) şi generare ($\overline{G}$) ale transportului, la nivelul întregului circuit;

- un selector de date *(selector resurse UAL)* pentru cele două intrări ale unității aritmetice-logice, care pot reprezenta combinații între ieşirile memoriei biport *(A, B),* o intrare externă de date *(D),* constanta "ZERO" şi ieşirea unui registru suplimentar - extensie *(Q*);

- un selector de ieşire din circuit, care furnizează prin intermediul unor tampoane TS fie date de la ieşirea *A* a memorie biport, fie datele de la ieşirea UAL;

- un registru auxiliar-extensie (Q), care poate fi încărcat fie cu datele de la ieşirea *UAL,* fie cu propriul său conținut deplasat stânga/dreapta prin intermediul unei rețele logice de deplasare-multiplexor *QSHIFT;*

- o rețea de deplasare-multiplexor *RAMSHIFT,* plasată pe intrarea *B* a memoriei biport RAM.

Conform figurii 5.1, vectorul de comandă este compus din 9 biți după cum urmează: biții $I_{2:0}$ reprezintă sursa UAL, biții $I_{5:3}$ reprezintă funcția UAL iar biții $I_{8:6}$ reprezintă destinația UAL.

Unitatea aritmetică-logică poate efectua, sub controlul semnalelor $I_{3:5}$, trei operații aritmetice binare şi cinci operații logice, asupra operanzilor aplicați la intrările *R* şi *S.* În tabelul 5.1, sunt prezentate funcțiile logice realizate de unitatea de execuție AMD 2901.

Microcod octal		**Grup**	**Funcție**	**Microcod octal**		**Grup**	**Funcție**
$I_{2:0}$	$I_{5:3}$			$I_{2:0}$	$I_{5:3}$		
0	4	AND	$A \cap Q$	2	6	PASS	Q
1	4		$A \cap B$	3	6		B
5	4		$D \cap A$	4	6		A
6	4		$D \cap Q$	7	6		D
0	3	OR	$A \cup Q$	2	3	PASS	Q
1	3		$A \cup B$	3	6		B
5	3		$D \cup A$	4	3		A
6	3		$D \cup Q$	7	3		D
0	6	EXOR	$A \oplus Q$	2	4	"ZERO"	0
1	6		$A \oplus B$	3	4		0
5	6		$D \oplus A$	4	4		0
6	6		$D \oplus Q$	7	4		0
0	7		$A \equiv Q$	0	5		$\overline{A} \cap Q$
1	7	EXNOR	$A \equiv B$	1	5	MASK	$\overline{A} \cap B$
5	7		$D \equiv A$	5	5	MASK	$\overline{D} \cap A$

6	7		$D \equiv Q$	6	5		$\overline{D} \cap Q$
2	7	INVERT	$\overline{Q}$				
3	7		$\overline{B}$				
4	7		$\overline{A}$				
7	7		$\overline{D}$				

Tabelul 5.1: AMD 2901 - Funcții logice.

Funcțiile aritmetice realizate de către AMD 2901 sunt cele prezentate în tabelul 5.2.

Microcod octal		$C_n = 0$		$C_n = 1$	
$I_{2:0}$	$I_{5:3}$	**Grup**	**Funcție**	**Grup**	**Funcție**
0	0	ADD	A + Q	ADD plus 1	A + Q + 1
1	0		A + B		A + B + 1
5	0		D + A		D + A + 1
6	0		D + Q		D + Q + 1
2	0	PASS	Q	Incrementare	Q + 1
3	0		B		B + 1
4	0		A		A + 1
7	0		D		D + 1
2	1	Decrementare	Q – 1	PASS	Q
3	1		B – 1		B
4	1		A – 1		A
7	2		D – 1		D
2	2	Complementul	-Q – 1	Complementul	-Q
3	2	față de 1	-B – 1	față de 2	-B
4	2		-A – 1		-A
7	1		-D – 1		-D
0	1	Scădere în com-	Q – A – 1	Scădere în com-	Q – A
1	1	plementul față de	B – A – 1	plementul față de	B – A
5	1	1	A – D – 1	2	A – D
6	1		Q – D – 1		Q – D
0	2	Scădere în com-	A – Q – 1	Scădere în com-	A – Q
1	2	plementul față de	A – B – 1	plementul față de	A – B
5	2	1	D – A – 1	2	D – A
6	2		D – A – 1		D – Q

Tabelul 5.2: AMD 2901 - Funcții aritmetice.

Microinstrucțiunea care controlează unitatea de execuție posedă 24 de biți. Biții 3 - 0 au apărut ca urmare a înglobării datei în microinstrucțiune. Câmpul *D* al

microinstrucțiunii (în figura 5.3 Data "D") se va extinde în incremenți de 4 biți odată cu extinderea lungimii cuvântului prelucrat de către unitatea de execuție.

Câmpul C_n este asociat cu transportul în rangul cel mai puțin semnificativ al unității de execuție, fiind la latitudinea celui care scrie microprogramul.

AMD 2909

Secvențiatorul de microprogram **AMD 2909** face parte din familia de circuite **AMD 2900** şi constituie elementul de bază în jurul căruia este organizată unitatea de comandă microprogramată. Schema bloc a unității de comandă este prezentată în figura 5.2

Resursele hardware prezente în figura 5.2 sunt următoarele:

- Multiplexor (MUX) 4 x (4:1);
- Incrementator (INC);
- Contor de microprogram (CMP);
- Registru de microinstrucțiuni (RMI);
- Memorie de tip PROM (PROM);
- Comutator de adrese (CA);
- Registru de ramificare (RR);
- Indicatorul de stivă (IS);
- Stivă cu capacitatea 4 cuvinte x 4 biți (STV).

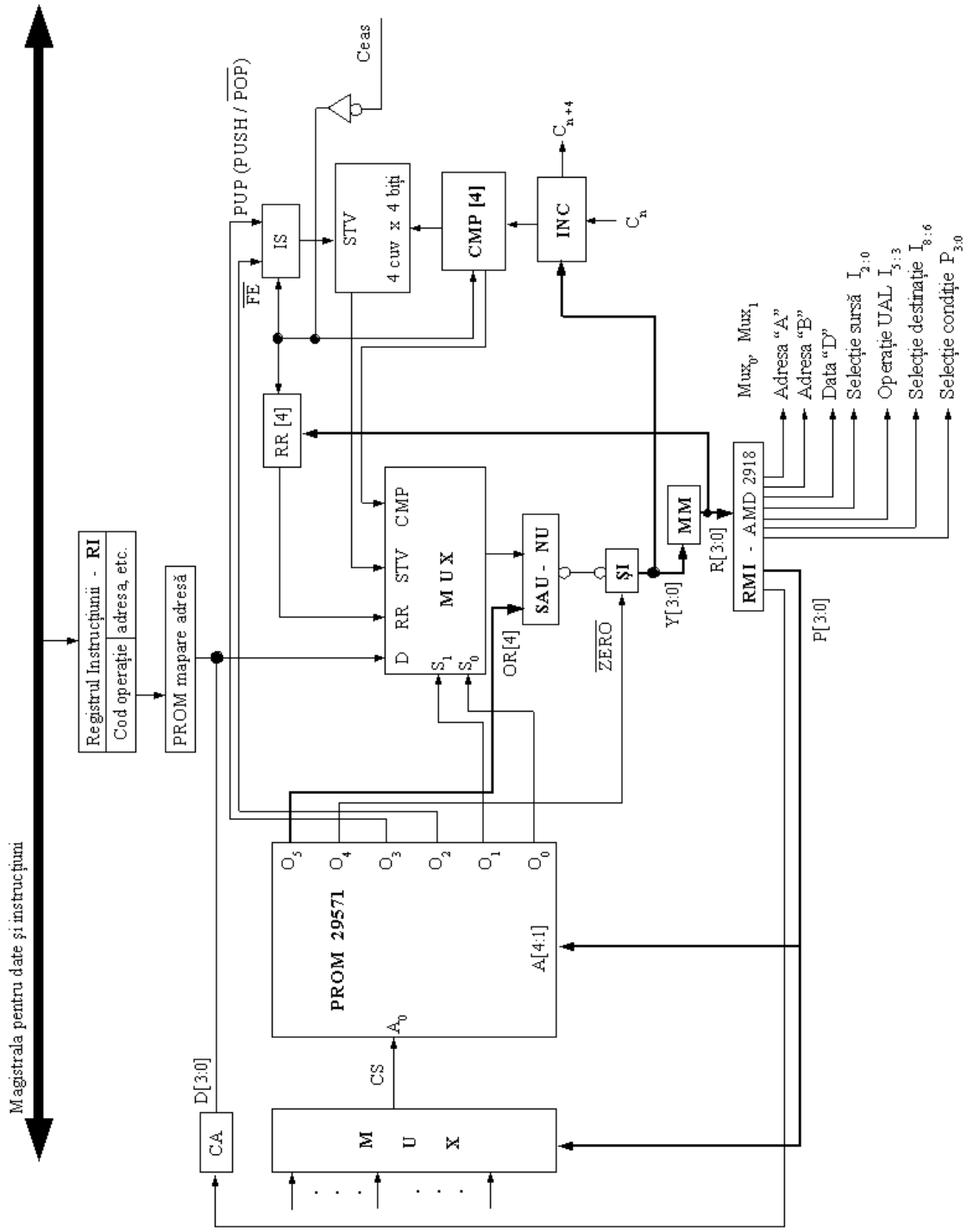


Figura 5.2: AMD 2909 - schemă bloc.

În locațiile memoriei PROM, adresate cu ajutorul câmpului $P_{3:0}$, sunt stocați vectori de comandă, care impun unității de comandă microprogramate efectuarea operațiilor prezentate în tabelul 5.3.

$P_{3:0}$ - cod hexa	Operație	Mnemonica
0	Ramificare la adresa R dacă F ≠ 0	JRNZF
1	Ramificare necondiționată la adresa R	JR
2	CONTINUĂ	CONT
3	Ramificare la adresa D	JD
4	Ramificare la subrutina cu adresa R dacă F ≠ 0	JSRNZF
5	Ramificare la subrutina cu adresa R	JSR
6	Revenire din subrutină	RS
7	Ramificare la adresa conținută în vârful stivei (fără POP)	JSTV
8	Terminare de ciclu şi POP dacă $F = 0$	TCPOZF
9	PUSH şi CONTINUĂ	PUCONT
A	POP şi CONTINUĂ	POCONT
B	Terminare de ciclu şi POP dacă $C_{n+4} = 1$	TCPOC
C	Ramificare la adresa R dacă $F = 0$	JRZF
D	Ramificare la adresa R dacă $F_3 = 1$	JRF3
E	Ramificare la adresa R dacă OVR = 1	JROVR
F	Ramificare la adresa R dacă $C_{n+4} = 1$	JRC

Tabelul 5.3: Setul de operații pentru AMD 2909.

Proiectarea şi simularea aplicațiilor dezvoltate pe circuitele AMD 2901 şi AMD 2909 se vor realiza cu ajutorul simulatorului ***C#_AMD.*** Aplicația ***C#_AMD*** oferă posibilitatea de a urmări funcționarea pas cu pas a execuției unui program.

Aşa cum se poate observa şi în figura 5.3, editorul este împărțit în două părți. Jumătatea superioară a ferestrei prezintă instrucțiunile linie cu linie, iar cea inferioară oferă posibilitatea operării unor modificări precum: introducere de linii noi *(InsertLine),* respectiv ştergerea anumitor linii din program (DeleteLine).

Începutul procesului de simulare se realizează odată cu selectarea butonului *Test.* În fereastra *Simulator* sunt afişate: programul sursă, instrucțiunea curentă în format binar, registrele unității de execuție AMD 2901 şi registrele de comandă AMD 2909. Toate resursele afişate în această fereastră au fost prezentate în figurile 5.1 şi 5.2.

Exemplu. Se consideră un microprocesor pe 4 biți realizat cu circuite din familia AMD 2900. Se doreşte proiectarea şi simularea unui contor pe 16 biți. Contorul este realizat prin concatenarea registrelor de câte 4 biți R_3, R_2, R_1, R_0.

Soluție. Primul pas este realizarea algoritmului care rezolvă cerințele impuse în enunț. Cu ajutorul ferestrei de editare, algoritmul este încărcat în simulator așa cum se observă în figura 5.3.

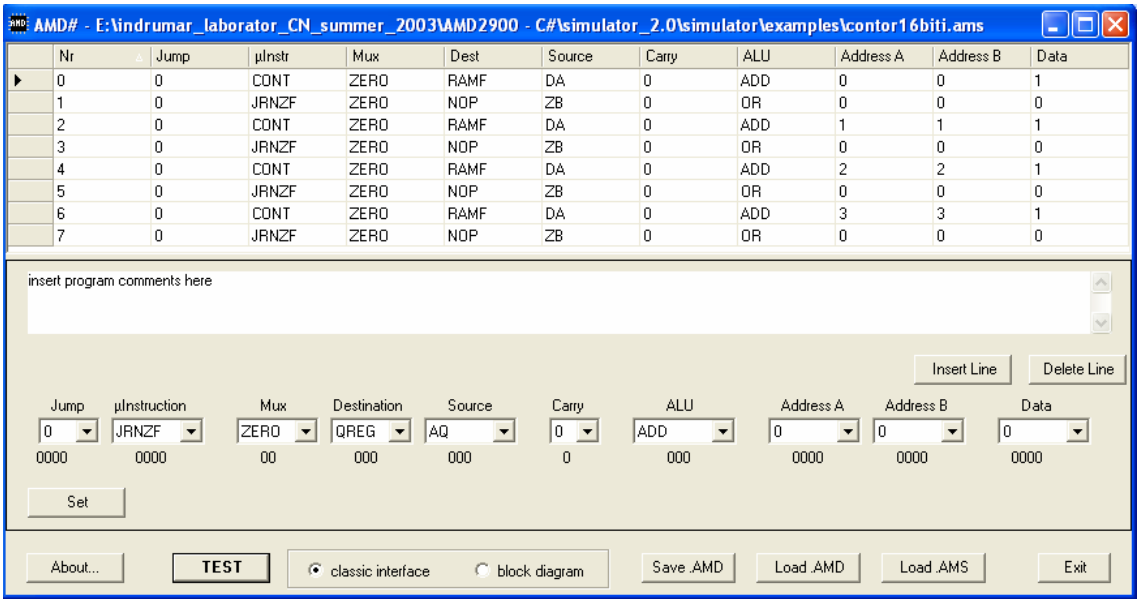
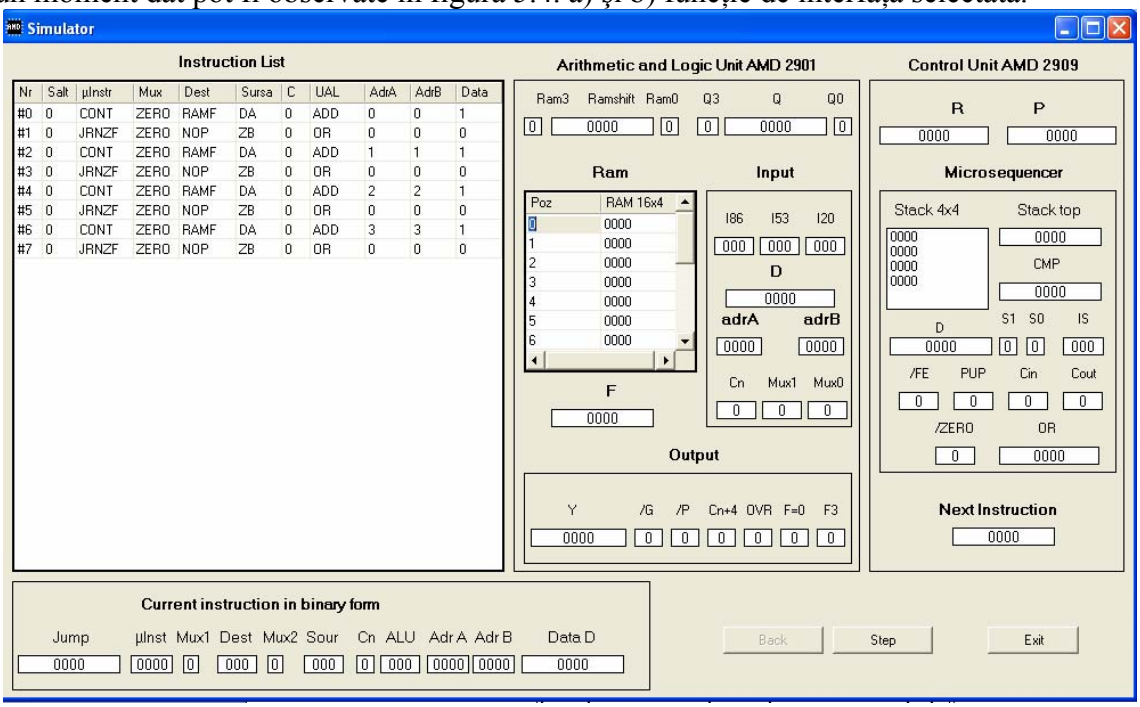


Figura 5.3: Algoritmul de implementare al unui contor pe 16 biți folosind circuite din familia AMD 2900.

Se selectează butonul *TEST* pentru a simula aplicația proiectată. Rezultatele simulării la un moment dat pot fi observate în figura 5.4. a) și b) funcție de interfața selectată.



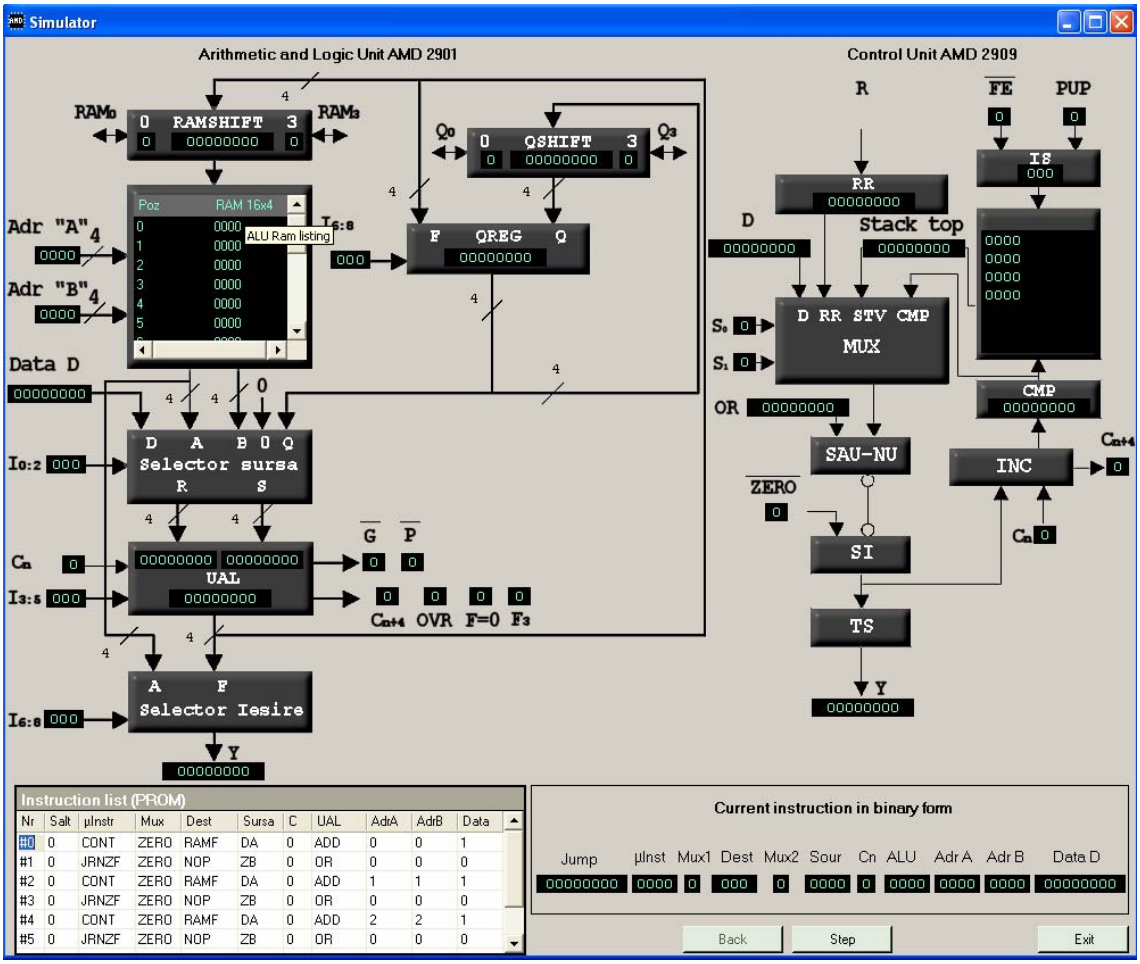


Figura 5.4: Fluxul informațional în cazul implementării unui contor pe 16 biți folosind circuite din familia AMD 2900.

2. Desfăşurarea lucrării

1. Se va elabora diagrama logică pentru exemplul prezentat în figura 5.3;

2. Cu ajutorul tabelelor 5.1, 5.2, 5.3, se va realiza programul care implementează funcționarea unui contor pe 16 biți;

3. Cu ajutorul simulatorului JAMD se va introduce programul obținut şi se va rula pas cu pas. Se va observa conținutul resurselor microprocesorului la fiecare pas.

3. Probleme propuse

1. Să se proiecteze şi să se simuleze folosind simulatorul JAMD, un numărător zecimal, hexazecimal, octal.

2. Să se proiecteze şi să se simuleze un program, care determină maximul a patru numere.

3. Să se proiecteze şi să se simuleze algoritmul de înmulțire al lui Booth.

6

Proiectarea dispozitivelor aritmetice în virgulă mobilă

1. Prezentare teoretică

În cadrul acestui laborator se vor prezenta algoritmii hardware pentru implementarea operațiilor de adunare, scădere, înmulțire şi împărțire a numerelor în virgulă mobilă utilizând conceptele benzii de asamblare.

Banda de asamblare

Banda de asamblare este o tehnică de descompunere a proceselor secvențiale în suboperații, fiecare suboperație putând fi executată într-un segment special dedicat şi în paralel cu alte suboperații. Rezultatul obținut în cadrul fiecărui segment este transferat următorului segment din banda de asamblare. La rezultatul final se ajunge atunci când datele au trecut prin toate segmentele benzii de asamblare.

Suprapunerea calculelor este posibilă datorită asocierii unui registru fiecărui segment al benzii de asamblare. Registrele au rolul de a izola segmentele astfel încât fiecare segment al benzii de asamblare poate opera pe date distincte simultan cu operarea altor segmente.

Exemplu Se doreşte aflarea rezultatul expresiei: $A_i \cdot B_i + C_i$ pentru $i = \overline{1 \ldots 7}$.

Soluție Fiecare operație poate fi implementată în câte un segment al benzii de asamblare. Fiecare segment conține unul sau două registre precum şi un circuit combinațional aşa cum se poate observa în figura 6.1. Registrele R_1 ... R_5 primesc date noi la fiecare apariție a

frontului de ceas. Circuitul de înmulțire precum şi sumatorul din figura 6.1 sunt circuite combinaționale.

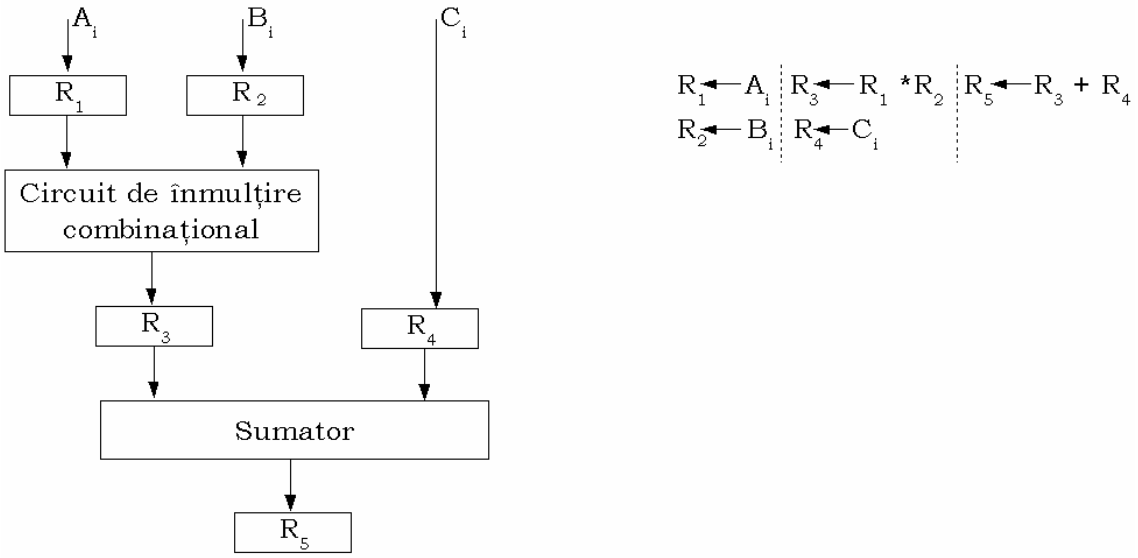


Figura 6.1: Calculul expresiei $A_i \cdot B_i + C_i$ folosind banda de asamblare.

Conținutul registrelor benzii de asamblare este prezentat în tabelul 6.1.

Ceas	Segmentul 1		Segmentul 2		Segmentul 3
	R_1	**R_2**	**R_3**	**R_4**	**R_5**
1	A_1	B_1			
2	A_2	B_2	$A_1 \cdot B_1$	C_1	
3	A_3	B_3	$A_2 \cdot B_2$	C_2	$A_1 \cdot B_1 + C_1$
4	A,	B_4	$A_3 \cdot B_3$	C_3	$A_2 \cdot B_2 + C_2$
5	A_5	B_5	$A_4 \cdot B_4$	C_4	$A_3 \cdot B_3 + C_3$
6	A_6	B_6	$A_5 \cdot B_5$	C_5	$A_4 \cdot B_4 + C_4$
7	A_7	B_7	$A_6 \cdot B_6$	C_6	$A_5 \cdot B_5 + C_5$
8			$A_7 \cdot B_7$	C_7	$A_6 \cdot B_6 + C_6$
9					$A_7 \cdot B_7 + C_7$

Tabelul 6.1: Conținutul registrelor benzii de asamblare.

Reprezentarea numerelor în virgulă mobilă

Reprezentarea numerelor în virgulă mobilă face obiectul standardului IEEE 754. Conform acestui standard reprezentarea lor este alcătuită din două părți. Prima parte reprezintă un

număr cu semn denumit *mantisă.* Partea a doua specifică poziția punctului zecimal şi se numeşte *exponent.*

Reprezentarea numerelor în virgulă mobilă poate fi făcută în precizie simplă sau dublă.

Folosirea preciziei simple impune o reprezentare a numărului utilizând 32 de biți. Primul bit (bitul cel mai semnificativ) este bitul de semn. Dacă valoarea acestui bit este 0, numărul este pozitiv, altfel numărul este negativ. Următorii 8 biți sunt alocați pentru valoarea exponentului, iar ultimii 23 de biți reprezintă mantisa. Gama de reprezenatre în cazul preciziei simple este: $-1.8 \times 10^{-38} \div 3.40 \times 10^{38}$.

În cazul în care se foloseşte precizia dublă, numărul biților utilizați pentru reprezentarea numărului se dublează devenind 64. Primul bit (bitul cel mai semnificativ) este bitul de semn. Exponentul este reprezentat pe următorii 11 biți, iar mantisa sau fracția pe 52 de biți.

Reprezentarea generală a unui număr în virgulă mobilă, folosind precizia simplă, este dată în ecuația 6.1.

$$(-1)^{s} \cdot (1 + MANTISA) \cdot 2^{Exponent - 127} \qquad (6.1)$$

Valoarea variabilei *Exponent* din cadrul ecuației 6.1 este *127* pentru a se evita valori negative ale exponentului. Pentru precizia dublă, această valoare se modifică devenind 1023.

Adunarea şi scăderea în virgulă mobilă

Resursele hardware necesare implementării acestor operații sunt prezentate în figura 6.2.

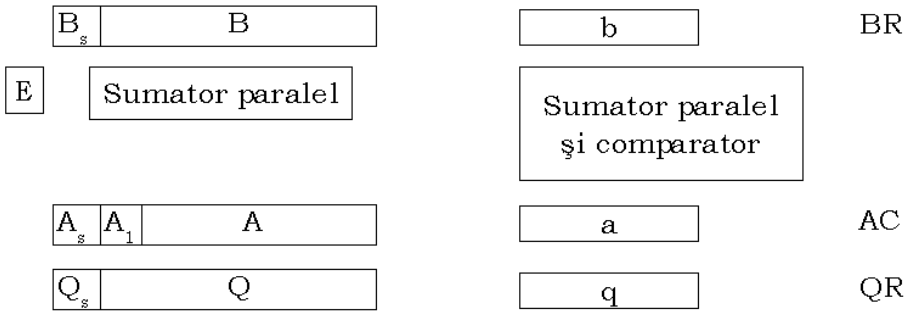


Figura 6.2: Resursele hardware necesare adunării/scăderii a două numere în virgulă mobilă.

Algoritmul necesită parcurgerea următoarelor etape:

1. Se verifică dacă unul dintre operanzi este zero sau nu;
2. Se aliniază mantisele;
3. Se adună sau se scad mantisele;
4. Se normalizează rezultatul obținut.

Algoritmul de adunare/scădere a două numere în virgulă mobilă utilizând tehnica bandă de asamblare poate fi observat în figura 6.3.

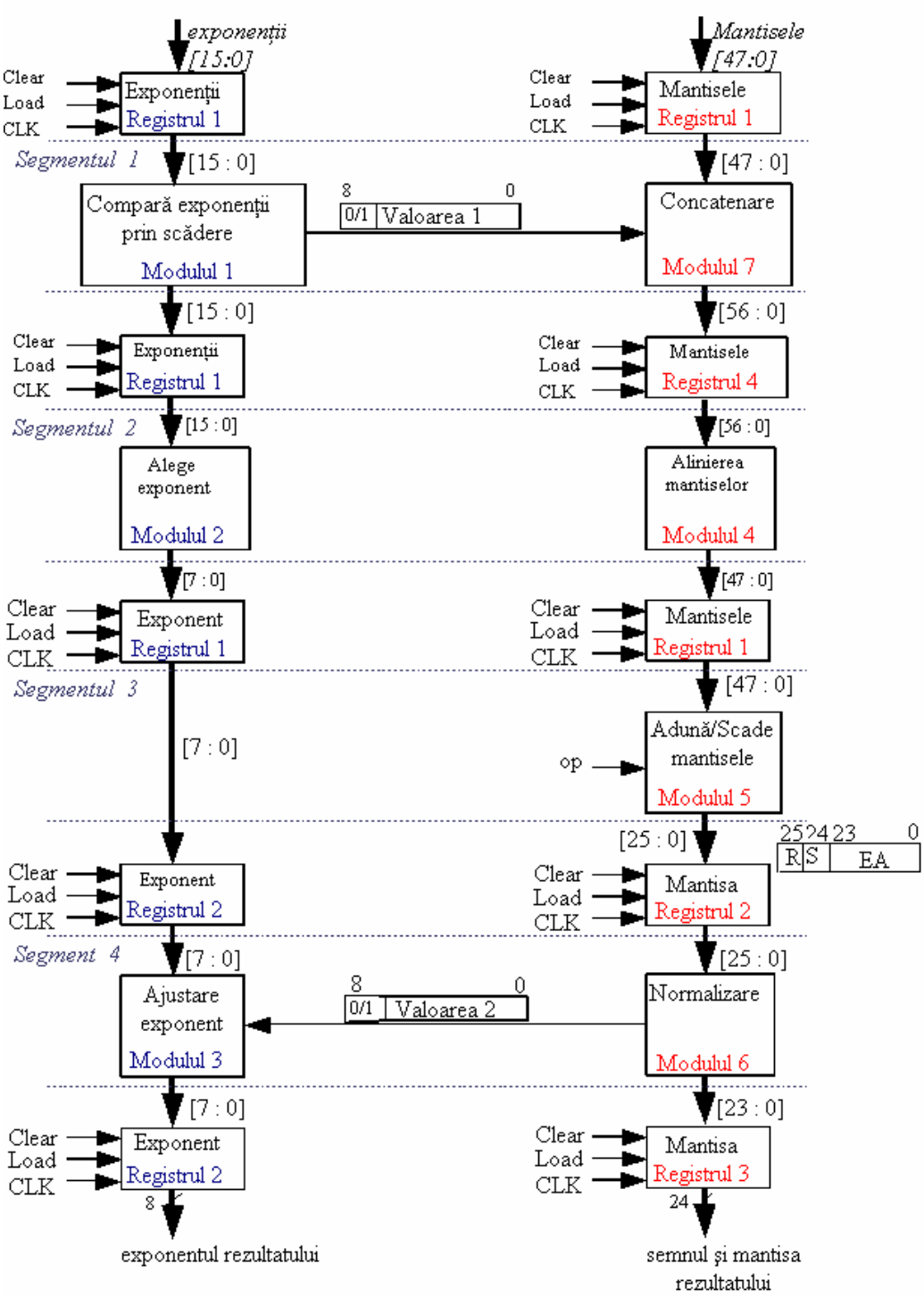


Figura 6.3: Adunarea şi scăderea a două numere reprezentate în virgulă mobilă.

În cadrul segmentului 1 al benzii de asamblare, exponenţii sunt comparaţi prin scădere. Exponentul mai mare este ales ca exponent al rezultatului. Rezultatul diferenţei indică de câte ori mantisa asociată cu exponentul mai mic trebuie deplasată către dreapta. Astfel se realizează alinierea mantiselor.

Variabila *Valoarea1* memorează rezultatul comparării. Dacă bitul cel mai semnificativ al acestei variabile este 0, mantisa primului număr trebuie deplasată dreapta. Pentru a memora numărul de deplasări care trebuie efectuate, au fost alocați 8 biți *(Valoarea1[7 : 0]*).

Dacă *Valoarea1[8]* este l, atunci mantisa asociată numărului 2 trebuie deplasată dreapta de un număr de ori egal cu valoarea lui *Valoarea1[7 : 0]*. Pentru implementarea sumatorului care apare în cadrul benzii de asamblare se recomandă utilizarea unui sumator cu transport anticipat (CARRY LOOK AHEAD).

În cadrul segmentului 3 al benzii de asamblare, cele două mantise sunt adunate sau scăzute în funcție de valoarea semnalului *OP*. Dacă *OP = 0* atunci trebuie realizată operația de adunare şi, în consecință, cele două mantise trebuie adunate. Dacă *OP = 1,* operația de scădere trebuie realizată şi, în consecință, cele două mantise trebuie scăzute.

Rezultatul pasului de adunare/scădere a mantiselor se reprezintă pe 26 de biți. Bitul cel mai semnificativ, bitul 25, este utilizat pentru a specifica dacă mantisa este egală sau nu cu zero. Restul de biți este folosit pentru a reprezenta registrele *E* şi *A* concatenate.

Rezultatul este normalizat în cadrul segmentului 4 al benzii de asamblare. Când apare o depăşire superioară, mantisa-rezultat este deplasată dreapta şi exponentul este incrementat cu o unitate. Când apare o depăşire inferioară, numărul de zerouri din cadrul mantisei (pozițiile cele mai semnificative) determină numărul de deplasări stânga al mantisei, număr care trebuie scăzut din exponent. Memorarea acestui număr se face prin intermediul variabilei *Valoarea2.*

Dacă bitul cel mai semnificativ al variabilei *Valoarea1* este 0, atunci exponentul trebuie deplasat spre stânga, altfel exponentul trebuie să fie deplasat spre dreapta. Numărul de deplasări este indicat de către *Valoarea2[7 :* 0].

Rezultatele simulării acestui algoritm folosind mediul de dezvoltare Xilinx pot fi observate în figura 6.4.

Înmulțirea în virgulă mobilă

Operația de înmulțire în virgulă mobilă presupune realizarea următorilor paşi:

1. Se verifică dacă unul dintre numere este zero sau nu;
2. Se adună exponenții;
3. Se înmulțesc mantisele;
4. Se normalizează produsul.

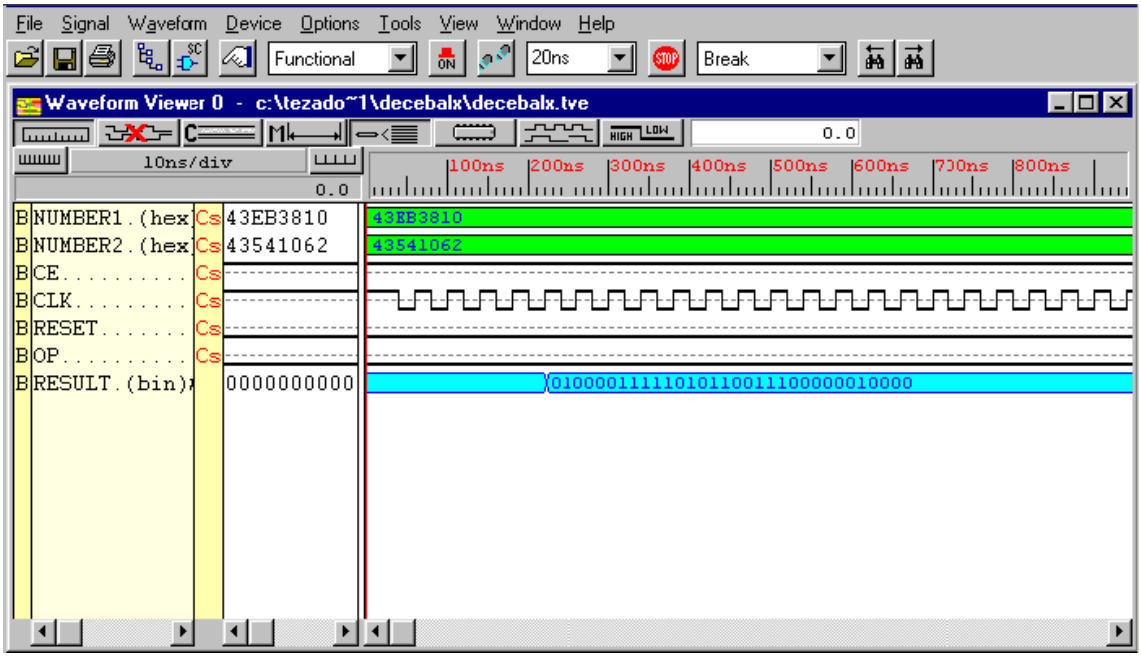


Figura 6.4: Rezultatele simulării algoritmului de adunare/scădere.

Implementarea în bandă de asamblare al acestui algoritm poate fi observată în figura 6.5.

Înmulțirea a două numere binare se realizează cu ajutorul unui circuit combinațional (denumit matrice de multiplicare) care formează toți biții produsului odată. Aceasta este cea mai rapidă metodă deoarece timpul necesar efectuării operației este egal cu timpul de propagare al semnalului luând în calcul ruta cea mai dezavantajoasă.

Circuitul *matrice de multiplicare* necesită un număr mare de porți, implementarea lui este neeconomică, acesta fiind principalul motiv pentru care acest circuit nu a avut o răpândire foarte mare până la apariția circuitelor integrate. Proiectarea unui asemenea circuit ale cărui intrări ar avea dimensiunile de j respectiv k biți va necesita $j \times k$ porți *ŞI* şi sumatoare de $(j-1)\cdot k$ biți pentru a obține un produs de *(j + k)* biți.

Un exemplu de circuit matrice de multiplicare este prezentat în figura 6.6.

Trebuie remarcat că exponentul rezultat corect este obținut prin scăderea valorii 127 din exponentul sumă.

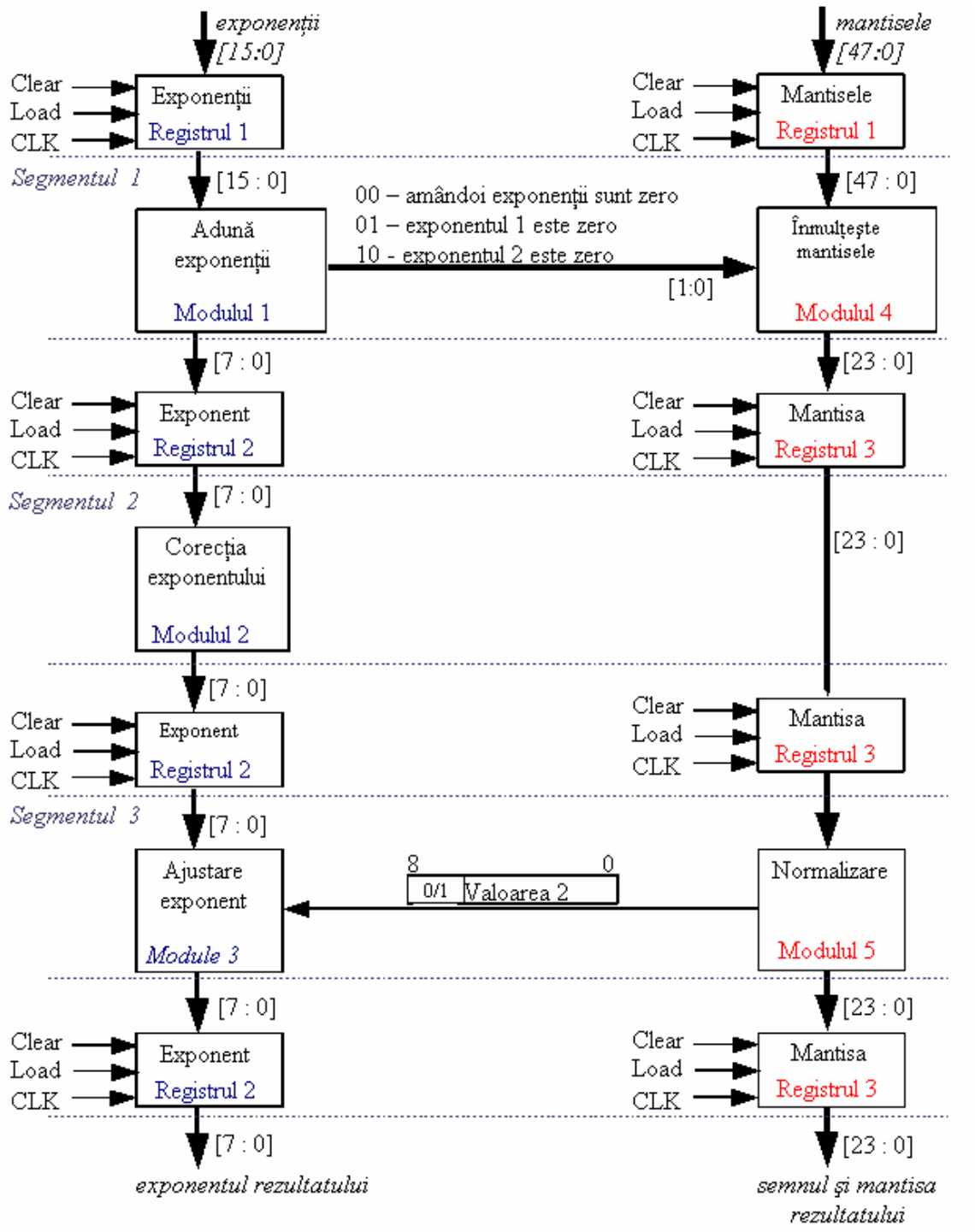


Figura 6.5: Algoritmul de înmulțire a două numere în virgulă mobilă.

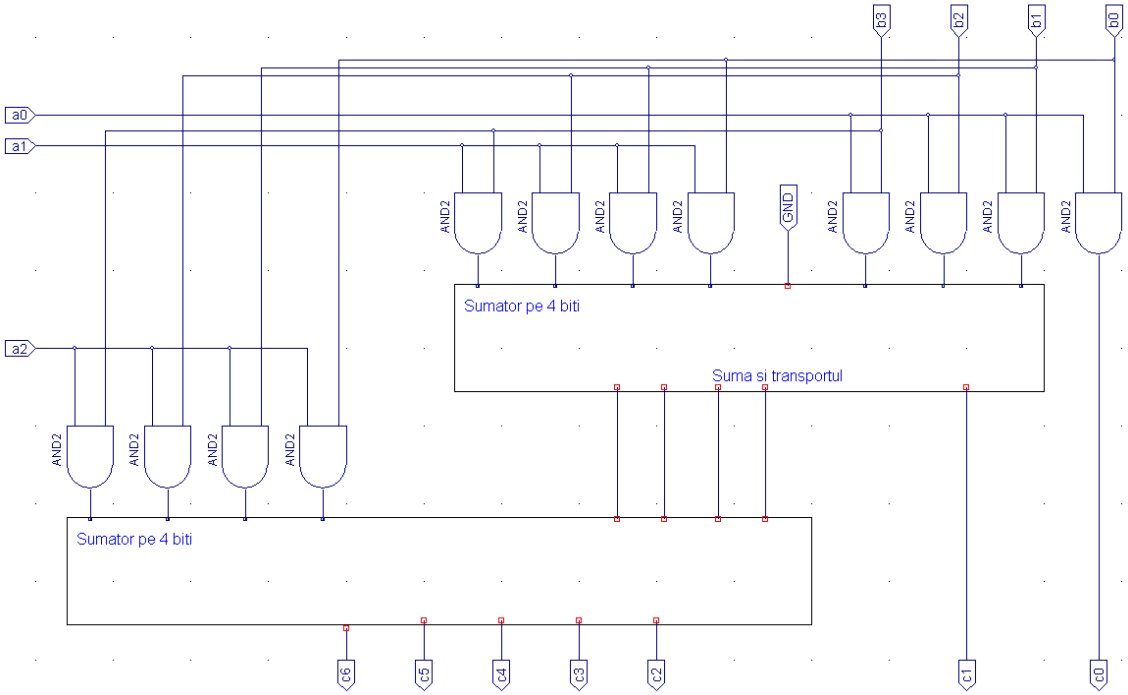


Figura 6.6: Circuitul matrice de multiplicare pentru *k = 4* şi *j = 3*.

Împărţirea în virgulă mobilă

Algoritmul de împărţire a două numere reprezentate în virgulă mobilă necesită parcurgerea următorilor paşi:

1. Se verifică dacă unul dintre cei doi operanzi este zero sau nu;
2. Se iniţializează registrele şi se evaluează semnul;
3. Se aliniază deîmpărţitul;
4. Se scad exponenţii;
5. Se împart mantisele.

Algoritmul în bandă de asamblare al acestei operaţii este prezentat în figura 6.7

Împărţirea a două numere în virgulă mobilă normalizate va produce întotdeauna un cât normalizat. Din acest motiv, spre deosebire de celelalte operaţii în virgulă mobilă, rezultatul operaţiei de împărţire nu necesită normalizare.

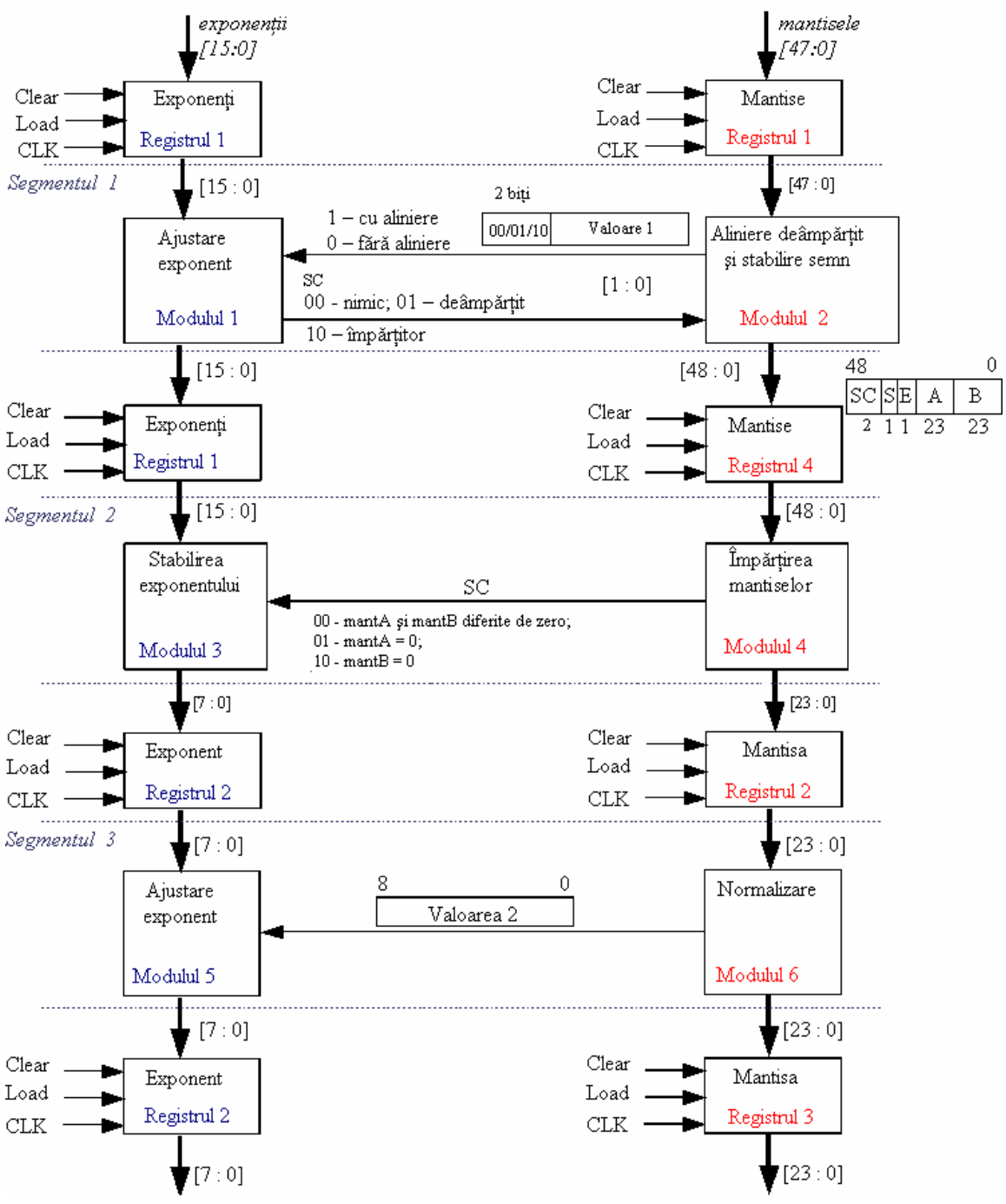


Figura 6.7: Algoritmul de împărțire a două numere în virgulă mobilă

2. Desfăşurarea lucrării

1. Se va proiecta în Verilog utilizând Xilinx WebPACK ISE 5.1i algoritmul de adunare/scădere a două numere reprezentate în virgulă mobilă, prezentat în figura 6.3.

2. Se va verifica funcționarea algoritmului utilizând simulatorul ModelSim.

3. Se va utiliza circuitul Xilinx Spartan XC2S200E pentru a verifica timpii reali de funcționare a circuitului proiectat.

3. Probleme propuse

1. Să se proiecteze şi simuleze utilizând Xilinx WebPACK 5.2i şi limbajul Verilog un circuit combinațional care să realizeze operația de înmulțire dintre două numere binare. Se va utiliza schema prezentată în figura 6.6;

2. Să se proiecteze şi simuleze utilizând Xilinx WebPACK 5.2i şi limbajul Verilog un circuit care să realizeze operația de înmulțire în virgulă mobilă precizie simplă prezentate în figura 6.5;

3. Să se proiecteze şi simuleze utilizând Xilinx WebPACK 5.2i şi limbajul Verilog un circuit care să realizeze operația de împărțire în virgulă mobilă precizie simplă prezentate în figura 6.7;

7

Procesorul didactic DLX

1. Prezentare teoretică

Procesorul didactic DLX este compus din următoarele resurse hardware:

1. UAL (Unitate arimetică - logică pentru întregi);
2. RG[32 : 31] - registre generale biport;
3. TS1, TS2, TD - registre tampon pentru registrele generale. Acestea sunt transparente programatorului;
4. TEMP - registru temporar, transparent pentru programator;
5. RAI - Registrul adresei de întrerupere (Registru Special);
6. CP - Contorul de Program;
7. RA, RD - Registrul de adrese, registrul de date;
8. RI - Registrul instrucțiunii;
9. M - memoria principala.

Operațiile executate de unitatea aritmetică şi logică sunt operații aritmetice, logice şi operații de deplasare. Operațiile de deplasare pot fi: $S_1 << S_2$ - deplasare logică stânga, $S_1 >> S_2$ - deplasare logică dreapta şi $S_1 >> aS_2$ - deplasare aritmetică dreapta.

Instrucțiunile procesorului DLX au 32 de biți, dintre care 6 sunt folosiți pentru codificarea operației si au următorul format:

1. **Instrucțiuni de tipul I** - acestea sunt instrucțiunile de tipul *imediat* şi au formatul prezentat în figura 7.1 a.). Instrucțiunile conțin câmpurile: *COP* - cod operație, *rs, rd* - adresele registrului sursă şi destinație, *imediat* - conține fie o adresă, fie un operand imediat.

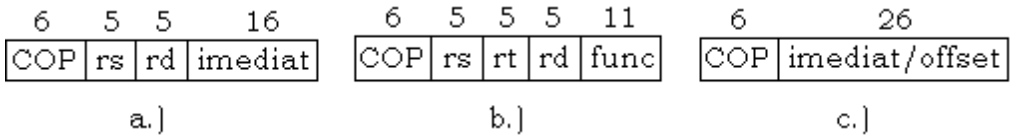


Figura 7.1: Formatul instrucțiunilor.

2. **Instrucțiuni de tip R** - aceste instrucțiuni sunt de tip *registru* şi au formatul prezentat în figura 7.1 b.). Operanzii se găsesc în registrele generale specificate de câmpurile *rs* şi *rt,* iar rezultatul este depus în registrul specificat de câmpul *rd.*

   Câmpul *func* reprezintă o extensie a codului de operație, pentru a putea codifica mai multe instrucțiuni decât ar permite un câmp de cod de operație de lungime 6 biți. Acest câmp codifică instrucțiunile operaționale aritmetice şi logice, instrucțiunile de scriere/ citire în/din registrul special RAI şi instrucțiunile de deplasare:
   $RG[rd] \leftarrow RG[rs](func)\,RG[rt]$.

3. **Instrucțiuni de tip J** - aceste instrucțiuni au formatul prezentat în figura 7.1 c.). Câmpul *imediat/offset* este utilizat pentru generarea adresei de salt. Instrucțiunile care intră în această categorie sunt cele de salt simplu, salt cu legătură, revenire (RET) şi întrerupere (TRAP).

Modurile de adresare prevăzute în instrucțiuni sunt:

1. *imediat* - operandul se află în instrucțiune;
2. *registru* - operandul se află într-un registru specificat de câmpul *rs*;
3. *cu deplasare bazată* - conținutul registrului se adună cu câmpul imediat pentru a se forma adresa;
4. *relativă la CP cu offset de 16 biți*. În acest caz se realizează şi extensia semnului;

5. *relativă la CP cu offset de 26 de biți*. În acest caz se realizează şi extensia semnului.

Setul de instrucțiuni al procesorului este compus dintr-un număr de 73 de instrucțiuni prezentate în tabelul 7.1. Acest set de instrucțiuni este folosit pentru scrierea şi testarea de aplicații folosind simulatorul java JDLX.

Nr	Instrucțiunea	Descriere	COP
Instrucțiuni - încarcă / Memorează			
1	LB	încarcă octet	6'b000000
2	LBU	încarcă octet fără semn	6'b000001
3	LH	încarcă semicuvânt	6'b000010
4	LHU	încarcă semicuvânt fără semn	6'b000011
5	LW	încarcă cuvânt	6'b000100
6	LF	încarcă float	6'b000101
1	LD	încarcă double float	6'b000110
8	LHI	încarcă imediat	6'b000111
9	SB	memorează octet	6'b001000
10	SH	memorează semicuvânt	6'b001001
1	SW	memorează cuvânt	6'b001010
12	SF	memorează float	6'b001011
13	SD5	memorează double	6'b001100
Instrucțiuni UAL cu operand imediat			
14	ADDI	Adunare cu operand imediat	6'b010000
15	SUBI	scădere cu operand imediat	6'b010001
16	Sil	ŞI logic imediat	6'b010010
17	SAUI	SAU imediat	6'b010011
18	XSAUI	SAU exclusiv imediat	6'b010100
19	SLLI	deplasare logică stânga imediat	6'b010101
20	SRLI	deplasare logică dreapta imediat	6'b010110
2	SRAI	shift aritmetic dreapta imediat	6'b010111
22	ADDUI	adunare fără semn imediată	6'b011110
23	SUBUI	scădere fără semn imediată	6'b011111
Instrucțiuni SET cu operand imediat			
24	SLTI	setează mai mic decât operand imediat	6'b011000
2	SGTI	setează mai mare decât operand imediat	6'b011001
26	SLEI	setează mai mic sau egal decât operand	6'b011010
2	SGEI	setează mai mare sau egal decât operand	6'b011011
2	SEQI	setează egal cu operand imediat	6'b011100
29	SNEI	setează diferit de operand imediat	6'b011101
30	JMP	salt la adresa specificată de operandul imediat	6'b100000
3	JR	întoarcere la adresa specificată de *rs*	6'b100001
32	JAL	cheamă de la adresa imediat	6'b100010

3	JALR	cheamă de la adresa *rd*	6'b100011
34	TRAP	instrucțiunea TRAP	6'bl00100
35	RET	revenire la adresa indicată de RG[31]	6'b100101
Instrucțiuni de ramificare			
36	BEQ	ramificare dacă egalitate	6'b110000
37	BNE	ramificare dacă nu există egalitate	6'bl10001
Alte instrucțiuni			
38	NOP	fără operație	6'b111101
39	R3	instrucțiune care necesită 3 registre	6'b111110
40	HALT	instrucțiunea HALT	6'b111111
Instrucțiuni UAL			
41	ADD	adunare	11'b00000000000
42	ADDU	adunare - operanzi fără semn	11'b00000000001
43	SUB	diferență	11'b00000000010
44	SUBU	diferență - operanzi fără semn	11'b00000000100
45	MULT	înmulțire	11'b00000000101
46	MULTU	înmulțire - operanzi fără semn	11'b00000000110
47	DIV	împărțire	11'b00000000111
48	DIVU	înpărțire - operanzi fără semn	11'b00000001000
49	ŞI	ŞI logic	11'b00000001001
50	SAU	SAU logic	11'b00000001010
51	XSAU	SAU-EXCLUSIV	11'b00000001011
52	SLL	deplasare stânga logică	11'b00000001100
53	SRL	deplasare dreapta logică	11'b00000001101
54	SRA	deplasare dreapta aritmetică	11'b00000001110
Instrucțiuni în virgulă mobilă			
55	ADDF	adunare în virgulă mobilă - precizie	11'b00000100000
56	ADDD	adunare în virgulă mobilă - precizie dublă	11'b00000100001
57	SUBF	scădere în virgulă mobilă - precizie simplă	11'b00000100010
58	SUBD	scădere în virgulă mobilă - precizie dublă	11'b00000100100
59	MULTF	înmulțire în virgulă mobilă - precizie	11'b00000100101
60	MULTD	înmulțire în virgulă mobilă - precizie	11'b00000100110
61	DIVF	împărțire în virgulă mobilă - precizie	11'b00000100111
62	DIVD	împărțire în virgulă mobilă - precizie	11'b00000101000
Istrucțiuni de setare			
63	SLT	setează mai mic	11'b00000110000
64	SGT	setează mai mare	11'b00000110001
65	SLE	setează mai mic sau egal	11'b00000110010
66	SGT	setează mai mare sau egal	11'b00000110011
67	SEQ	setează egal	11'b00000110100
68	SNE	setează diferit	11'b00000110101
Instrucțiuni de transfer date			

69	MOVS2I	mută din registrul RAI în registrul RG	11'b00001000001
70	MOVI2S	mută din registrul RG în registrul RAI	11'b00001000000
71	MOV	mută din RG în RG	11'b00001000010
72	MOVF	mută număr în virgulă mobilă - precizie	11'b00001000011
73	MOVD	mută număr în virgulă mobilă - precizie	11'b00001000100

Tabelul 7.1: Procesor didactic DLX - Setul de instruciuni.

Implementarea hardware a procesorului didactic DLX este realizată utilizând limbajul **VERILOG.** Sursele implementării sunt prea mari pentru a fi introduse în cadrul laboratorului. Sursele se găsesc la adresa http://www.csit-sun.pub.ro/resources. Acestea vor trebui să suporte unele modificări pentru a rezolva problemele propuse în cadrul acestei lucrări de laborator.

Un exemplu de program pentru procesorul didactic DLX este oferit în continuare.

000111_00 000_00011 11111111 11110001
0 : lhi 0 3 -15 --- rg[3] = (-15)(c)<< 16
010111_00 011_00100 00000000 00010000
4: srai 3 4 16 --- rg[4] = rg[3] >> 16(a) = -15(c)
000100_00 100_00010 00000000 00110111
8: lw 4 2 55 --- rg[2] = mem[55 + rg[4]] = mem[40] = 127 = 7fh
010000_00 010_00001 00000000 00010111
12: addi 2 1 23 --- rg[1] = rg[2] + 23 = 150 = 96h
001001_00 000_00001 00000000 01000000
16: sh 0 1 64 --- mem[64 + 0] = rg[1] = 98h -- scriere semicuv !!
001001_00 000_00010 00000000 01000100
16: sh 0 2 68 --- mem[68 + 0] = rg[2] = 7fh -- scriere semicuv !!
111110_00 010_00001 00011_000 00000101
20: mult 2 1 3 --- rg[3],rg[4] = rg[2] * rg[1]
001010_00 000_00011 00000000 01001000
24: sw 0 3 72 --- mem[72 + 0] = rg[3] = 0h
001010_00 000_00100 00000000 01001100
28: sw 0 4 76 --- mem[76 + 0] = rg[4] = 4a6ah
111111_00 00000000 00000000 00000000
32: halt
00000000 00000000 00000000 01111111
40: 127

Configurația memoriei înainte de lansarea în execuție a programului este:

00011100_00000011_11111111_11110001
01011100_01100100_00000000_00010000
00010000_10000010_00000000_00110111
01000000_01000001_00000000_00010111
00100100_00000001_00000000_01000000

00100100_00000010_00000000_01000100
11111000_01000001_00011000_00000101
00101000_00000011_00000000_01001000
00101000_00000100_00000000_01001100
11111100_00000000_00000000_00000000
00000000_00000000_00000000_01111111

După rularea programului conținutul memoriei va fi:

00011100_00000011_11111111_11110001
01011100_01100100_00000000_00010000
00010000_10000010_00000000_00110111
01000000_01000001_00000000_00010111
00100100_00000001_00000000_01000000
00100100_00000010_00000000_01000100
11111000_01000001_00011000_00000101
00101000_00000011_00000000_01001000
00101000_00000100_00000000_01001100
11111100_00000000_00000000_00000000
00000000_00000000_00000000_01111111
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_00000000_10010110
xxxxxxxx_xxxxxxxx_00000000_01111111
00000000_00000000_00000000_00000000
00000000_00000000_01001010_01101010
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx

2. Desfăşurarea lucrării

Se vor modifica sursele Verilog pentru procesorul didactic DLX, pentru a implementa exemplul prezentat în laborator. Să se compare conținutul memoriei obținut cu cel prezentat în cadrul laborator.

3. Probleme propuse

1. Să se implementeze şi să se simuleze următorul program folosind sursele VERILOG pentru procesorul didactic DLX:

```
lhi 0 3 -15          rg[3] = (-15)(c) << 16
srai 3 4 16          rg[4] = rg[3] >> 16(a) = -15(c)
lw 4 2 28            rg[2] = mem[43 + rg[4]] = mem[28] = 127
addi 2 1 23          rg[1] = rg[2] + 23 = 150
halt
lb rg[3] rg[4] 10
zona libera
127
```

2. Să se implementeze şi să se simuleze următorul program folosind sursele VERILOG pentru procesorul didactic DLX:

```
0: lf 0 4 72         f[4] = mem[72 + 0]] = 0,25
4: lf 0 3 76         f[3] = mem[76 + 0]] = 0,75
8: addf 4 3 1        f[1] = f[3] + f[4] = 1
12: sf 0 1 100       mem[100 + 0] = f[1]
16: lf 0 4 80        f[4]  =  mem[80+0]] = -0,5
20: multf 3 4 1      f[1]  =  f[3]  *  f[4]
24: sd 0 1 120       mem[120 + 0]  =  f[1]  =  0,375 =
                     1_01111110_1100_0000_0000_0000
28: halt
0 0 0 0   //32:
0 0 0 0   //36:
0 0 0 0   //40:
0 0 0 0   //44:
0 0 0 0   //48:
0 0 0 0   //52:
0 0 0 0   //56:
0 0 0 0   //60:
0 0 0 0   //64:
0 0 0 0   //68:
          //72: 0,25
          //76: 0,75
          //80: -0,5
```

8

Memorii şi ierarhii de memorii

1. Prezentare teoretică

Memoria reprezintă una din componentele esențiale ale unui calculator numeric. Rolul său este de a memora programe şi date. La modul general, memoria se împarte în memorie principală şi memorie secundară. Memoria principală stochează informațiile şi datele curente necesare procesorului cât timp memoria secundară memorează date care nu sunt folosite în mod curent.

Subsistemul de memorie al unui calculator numeric este văzut ca o ierarhie de module de memorie. La baza ierarhiei se află memoria auxiliară/secundară (lentă dar de capacitate foarte mare, de ordinul zecilor de GB), iar în vârful ierarhiei se află memoria **cache** (foarte rapidă dar de capacitate foarte mică, de ordinul KB).

Memoria cache este o memorie specială, utilizată pentru mărirea vitezei de prelucrare a procesorului. Aceasta este dispusă între procesor şi memoria principală în vederea compensării diferenței de viteză dintre cele două componente. Viteza de funcționare a memoriei cache este

foarte apropiată de viteza de funcționare a procesorului, acesta fiind un motiv pentru care, de obicei, această memorie este încapsulată împreună cu procesorul.

Memoria principală

Este o memorie cu un timp de acces de aproximativ 700ns. Capacitatea ei variază de la 32MB până la 512MB. Tehnologia folosită în construcția acestui tip de memorie se bazează pe circuite integrate semiconductoare. Circuitele integrate RAM pot opera în două moduri:

1. ***static*** - memoria statică RAM este formată din bistabile care memorează informația în formă binară. Informația rămâne disponibilă atât timp cât circuitul este alimentat cu curent electric.

2. ***dinamic*** - memoria dinamică RAM, memorează informația binară în formă de sarcini electrice ce sunt aplicate unor condensatori. Condensatorii, realizați cu ajutorul tranzistoarelor MOS, tind să se descarce în timp şi astfel apare riscul ca informația să se distrugă. Pentru a elimina acest inconvenient, în aceste memorii se realizează un ciclu de reîmprospătare a informației la intervale de câteva milisecunde.

Memoria ***ROM*** (Read Only Memory) este o memorie cu acces aleator şi este utilizată pentru memorarea programelor, care sunt rezidente permanent în calculator şi pentru stocarea tabelelor de constante definite de către producătorul calculatorului. Tot în această memorie se păstrează şi programul inițial de pornire a calculatorului (bootstrap loader) responsabil cu lansarea sistemului de operare. Conținutul acestei memorii se păstrează chiar dacă circuitul nu este alimentat. Singura metodă de a şterge informația dintr-un astfel de circuit constă în iradierea sa cu raze ultraviolete.

Simbolul şi tabela de funcționare a unui circuit de memorie RAM sunt prezentate în figura 8.1.

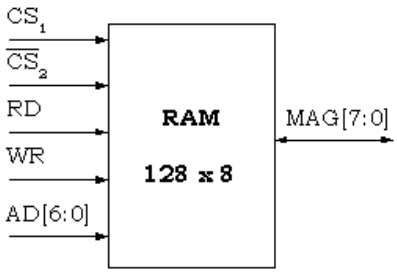


Tabela de funcționare

CS_1	$\overline{CS_2}$	RD	WR	Funcționare	MAG Magistrală de memorie
0	0	x	x	inhibată	Z
0	1	x	x	inhibată	Z
1	0	0	0	inhibată	Z
1	0	0	1	scriere	introducere date
1	0	1	x	citire	citire date
1	1	x	x	inhibată	Z

Figura 8.1: Memorie RAM 128 x 8.

Odată ce proiectantul unui calculator dispune de modulele de memorie fizice, el trebuie să asigneze memoriilor RAM sau ROM întreaga cantitate de memorie, determinată în prealabil, necesară pentru rularea aplicațiilor. Adresarea memoriei de către procesor este stabilită pe baza unei tabele în care se specifică adresele de memorie

asignate fiecărui dispozitiv/tip de memorie. Această tabelă poartă denumirea de "harta adreselor de memorie". Modalitatea de construire a acestei tabele este prezentată prin intermediul unui exemplu.

Exemplu Se presupune că un calculator necesită 512 octeți de RAM şi 128 octeți de ROM. Pentru implementare, la dispoziția proiectantului există circuitele RAM şi ROM cu capacitate de 128 x 8 fiecare. Numărul circuitelor de memorie este limitat la 5. Harta adreselor de memorie este prezentată în tabelul 8.1.

Componenta	**Adresa hexazecimală**	**Magistrala de adrese**									
		10	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**
RAM_1	0000 - 007F	0	0	0	x	x	x	x	x	x	x
RAM_2	0080 - 00FF	0	0	1	x	x	x	x	x	x	x
RAM_3	0100 - 017F	0	1	0	x	x	x	x	x	x	x
RAM_4	0180 - 01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200 - 03FF	1	x	x	x	x	x	x	x	x	x

Tabelul 8.1: Harta adreselor de memorie.

Memoria totală este: 512 + 128 = 640 octeți. Pentru a accesa o memorie de 640 octeți este necesar un cuvânt-adresă de 10 biți. Astfel, dacă se va considera o magistrală de adrese de 16 biți, primii 6 biți vor fi întotdeauna egali cu 0. Având în vedere că memoria RAM ocupă 512 octeți rezultă necesitatea utilizării a 9 linii de adresă. Deoarece sistemul conține si memorie RAM si memorie ROM, o linie de adresă este necesară pentru a se realiza o distincție între ele. Deci în total vor fi 10 linii de adresă ocupate. Restul de 6 linii de adresă vor fi tot timpul 0. Pentru uşurința proiectării, valoarea de pe magistrala de adrese este exprimată în hexazecimal.

Memoria asociativă

Memoria asociativă se bazează pe conceptul că timpul de căutare a unei informații date într-o memorie este redus considerabil dacă datele memorate pot fi identificate după conținut şi nu după adresă.

Când se scrie un cuvânt într-o memorie asociativă, nici o adresă nu este furnizată. Memoria este capabilă să determine locația goală şi neutilizată în vederea memorării noului cuvânt. La citirea unui cuvânt dintr-o memorie asociativă, conținutul cuvântului sau o parte a cuvântului este specificată. Memoria localizează toate cuvintele care se potrivesc cu conținutul specificat şi ele vor fi marcate pentru citire.

Organizarea hardware a unei memorii asociative este prezentată în figura 8.2.

În figura 8.2 registrele *A* şi *K* sunt registre de *n* biți. Registrul M, registrul de potrivire, este de *m* biți, unul pentru fiecare cuvânt de memorie.

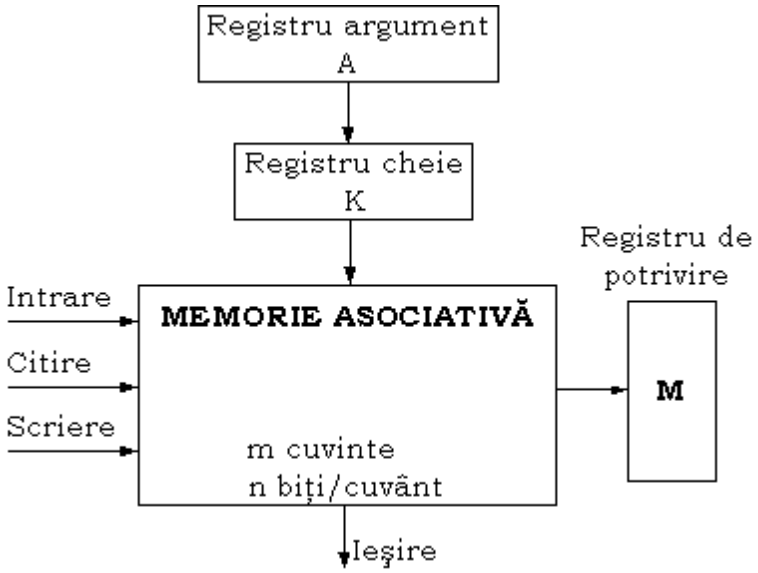


Figura 8.2: Organizarea hardware a unei memorii asociative

Fiecare cuvânt din memorie este comparat cu conținutul registrului *A*. Cuvintele care se potrivesc setează un bit corespunzător în registrul *M* a cărui valoare este 1. În final, registrul *M* va conține doar biții care indică ce cuvinte s-au potrivit. Citirea se realizează printr-un acces secvențial la memorie, doar pentru cuvintele ai căror biți corespunzători în registrul *M* sunt 1.

Registrul *K* este un registru de mascare în vederea stabilirii unui câmp particular sau a unei chei din cuvântul memorat în registrul *A*. Comparația întregului cuvânt de memorie cu conținutul registrului *A* se va realiza doar dacă registrul *K* conține toți biții egali cu 1. În mod normal se compară doar cuvintele care au biții l în pozițiile în care există l în registrul *K*.

Organizarea hardware a unei celule de memorie asociative precum şi relaţia dintre vectorul memorie şi registrele externe sunt prezentate în figura 8.3.

Aşa cum se observă în figura 8.3, fiecare bit A_j din registrul A este comparat cu toţi biţii coloanei j din matricea vector, dacă $k_j = 1$, $\forall\ j = \overline{1 \ldots n}$. Dacă toţi biţii registrului K sunt egali cu biţii din cuvântul *i,* atunci se setează $M_i = 1$, altfel $M_i = 0$.

Circuitul de potrivire logică din figura 8.3 se poate deduce matematic. Cuvântul *i* este egal cu conţinutul registrului *A* dacă:

$$A_j \quad = \quad F_{ij} \quad \forall\ j = \overline{1 \ldots n} \tag{8.1}$$

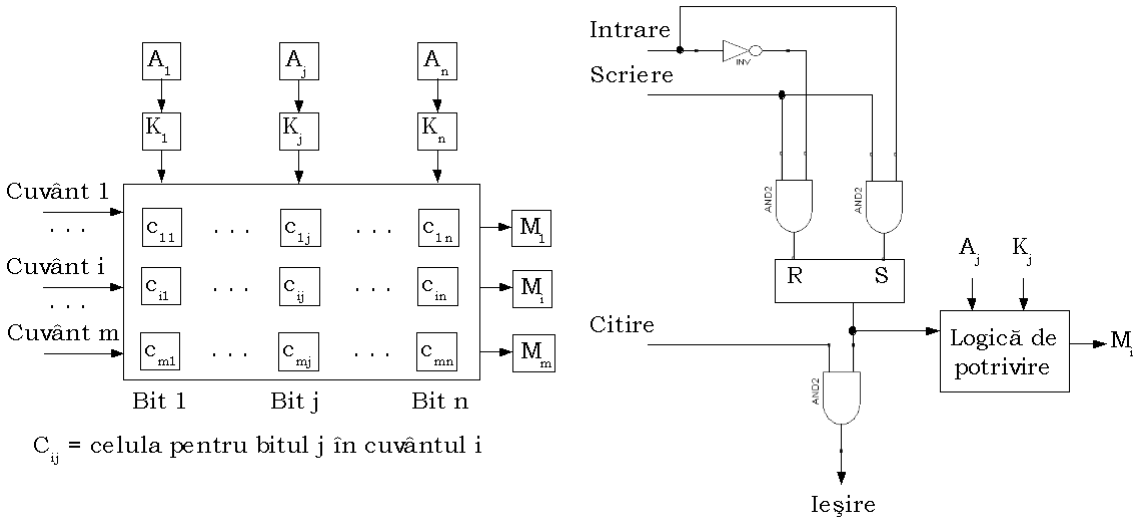


Figura 8.3: Organizarea hardware a unei celule de memorie

Egalitatea a doi biţi poate fi exprimată ca $x_j = A_j \cdot F_{ij} + \overline{A_j} \cdot \overline{F_{ij}}$, unde $x_j = 1$ dacă toţi biţii de pe poziţia *j* sunt egali. Pentru ca bitul M_i să fie 1 trebuie respectată următoarea egalitate:

$$M_i \quad = \quad x_1 \cdot x_2 \cdot x_3, ..., x_n \tag{8.2}$$

În funcţie de valoarea lui K_j, există două situaţii posibile:

1. $k_j = 0$, atunci biţii corespunzători A_j şi F_{ij} nu trebuie comparaţi;

2. $k_j = 1$, atunci biții corespunzători A_j şi F_{ij} trebuie comparați.

Se poate concluziona că:

$$x_j + \overline{k_j} = \begin{cases} x_j & daca \quad k_j = 1 \\ 1 & daca \quad k_j = 0 \end{cases} \tag{8.3}$$

Ecuația (8.2) se rescrie ca:

$$M_i = \left(x_1 + \overline{k_1}\right) \cdot \left(x_2 + \overline{k_2}\right) \cdot \left(x_3 + \overline{k_3}\right) \cdot \ldots \cdot \left(x_n + \overline{k_n}\right) \tag{8.4}$$

sau într-o formă mult mai compactă:

$$M_i = \prod_{j=1}^{n} \left(A_j \cdot F_{ij} + \overline{A_j} \cdot \overline{F_{ij}} + \overline{k_j}\right) \quad \forall\, i = \overline{1, m} \tag{8.5}$$

În vederea realizării operației de citire din memoria asociativă, se scanează conținutul registrului *M* şi se citeşte doar un bit al său la un moment dat, obținând astfel o secvență de cuvinte aflate în memoria asociativă şi care se potrivesc cu cuvântul de memorie.

În cazul operației de scriere apar următoarele situații posibile:

1. *memoria este vidă* - în acest caz scrierea poate fi realizată prin adresarea fiecărei locații într-o anumită secvență. Astfel memoria devine o memorie cu acces aleator în cazul scrierii şi o memorie de tip adresabilă după conținut în cazul operației de citire. Avantajul este că adresele cuvintelor de intrare pot fi decodificate ca în cazul memoriilor cu acces aleator, rezultând astfel numai *d* linii de adrese în loc de *m* ($m = 2^d$).

2. *suprascrierea unui cuvânt* - este situația în care memoria este complet ocupată şi se doreşte introducerea unui nou cuvânt în memorie. În această situație se mai adaugă un registru special (registrul de etichete) care memorează cuvintele active şi cele inactive asignând valoarea 1 pentru cuvintele active şi 0 pentru cuvintele inactive. Numărul biților acestui registru trebuie să fie acelaşi cu numărul de cuvinte din memorie. Se înlocuiesc doar cuvintele cu eticheta 0 utilizând în acest scop un algoritm FIFO, LRU etc. După realizarea operației de suprascriere, eticheta corespunzătoare va avea valoarea 1.

Memoria cache

Memoria cache este o memorie de mică capacitate dar cu o viteză de accesare foarte mare. Când CPU doreşte să acceseze memoria principală, întâi se examinează memoria cache. Dacă, cuvântul solicitat se găseşte stocat în memoria cache, atunci el este preluat

de către procesor, din această memorie. În caz contrar este accesată memoria principală, se citeşte cuvântul dorit şi este stocat apoi în memoria cache.

Pentru a respecta principiul "localității spațiale", se transferă din memoria principală în memoria cache nu doar cuvântul solicitat de procesor, ci un bloc de cuvinte care conține cuvântul solicitat. Dimensiunea blocului transferat variază de la o arhitectură la alta, dar uzual, blocul transferat are o dimensiune cuprinsă între l şi 16 cuvinte.

Performanța unei memorii cache este măsurată în termeni de "hit ratio". Dacă, cuvântul solicitat de CPU se află memorat în memoria cache, atunci este vorba de un succes ***HIT,*** iar altfel, de un insucces ***MISS***. ***HIT RATIO*** se defineşte ca fiind raportul dintre numărul de succese şi numărul total de referiri CPU la memorie.

Procesul de amplasare a datelor citite din memoria principală, în memoria cache se numeşte *mapare* . Uzual în proiectarea şi organizarea memoriilor cache se folosesc următoarele tipuri de proceduri de mapare:

1. mapare complet asociativă;

2. mapare directă;

3. mapare asociativă pe mai multe căi.

Pentru o descriere cât mai completă a fiecărui mod de mapare se va utiliza următorul exemplu.

Exemplu Se presupunem existența unui calculator care posedă o memorie principală cu capacitatea de 32k × 12 şi o memorie cache cu capacitatea 512 cuvinte × 12. Procesorul calculatorului poate comunica cu ambele memorii. El trimite o adresă de 15 biți către memoria cache. Dacă se obține HIT, CPU acceptă 12 biți de date de la memoria cache. În caz de MISS, CPU citeşte cuvântul din memoria principală şi cuvântul va fi apoi memorat în memoria cache.

Maparea complet asociativă

Acest tip de mapare este cea mai flexibilă şi mai rapidă organizare a memoriei cache. Organizarea acestui tip de memorie poate fi observată în figura 8.4. În memoria complet asociativă sunt memorate adresele şi conținutul cuvintelor de memorie.

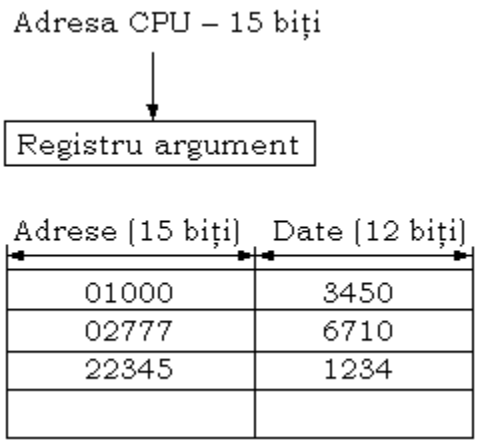


Figura 8.4: Maparea asociativă.

Adresa CPU de 15 biţi este memorată în registrul argument/descriptor şi apoi se încearcă depistarea unei potriviri de adrese dintre această adresă şi o adresă din memoria cache. Dacă procesul se încheie cu HIT, atunci datele de 12 biţi sunt citite şi transmise către CPU.

Dacă procesul se încheie cu MISS, atunci perechea adresă-dată este citită din memoria principală, trimisă la procesor şi memorată în memoria cache. În cazul cel mai defavorabil, memoria este complet ocupată şi noua pereche adresă-dată trebuie suprascrisă în locul altei perechi stabilită de către algoritmul de replasare utilizat (uzual se folosesc algoritmii LRU şi FIFO).

Mapare directă

O versiune mai ieftină de memorie cache este prezentată în figura 8.5. În cadrul acestei implementări, adresa CPU se împarte în două câmpuri: INDEX (numărul biţilor acestui câmp este egal cu numărul biţilor necesari pentru a accesa memoria cache) şi ETICHETĂ. În concluzie, cei 15 biţi ai adresei CPU se împart după cum urmează: 6 biţi pentru câmpul ETICHETĂ şi 9 biţi pentru câmpul INDEX.

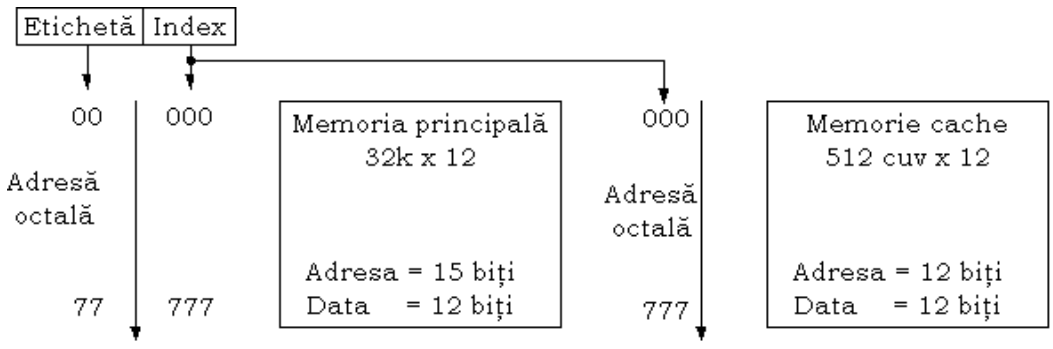


Figura 8.5: Maparea directă.

Fiecare cuvânt memorat în memoria cache este format din câmpul de date şi eticheta asociată. Atunci când o cerere este lansată, câmpul INDEX este utilizat pentru adresa

de acces a memoriei cache. Câmpul ETICHETĂ al adresei CPU este comparat cu "eticheta" cuvântului citit din memoria cache.

Dacă procesul returnează MISS, cuvântul cerut se citeşte din memoria principală şi este memorat în memoria cache împreună cu o nouă etichetă, înlocuindu-se astfel valoarea anterioară.

Dezavantajul major constă în scăderea valorii HIT RATIO dacă două sau mai multe cuvinte ale căror adrese au acelaşi index dar etichete diferite sunt accesate în mod repetat.

Un exemplu numeric este prezenata în figura 8.6.

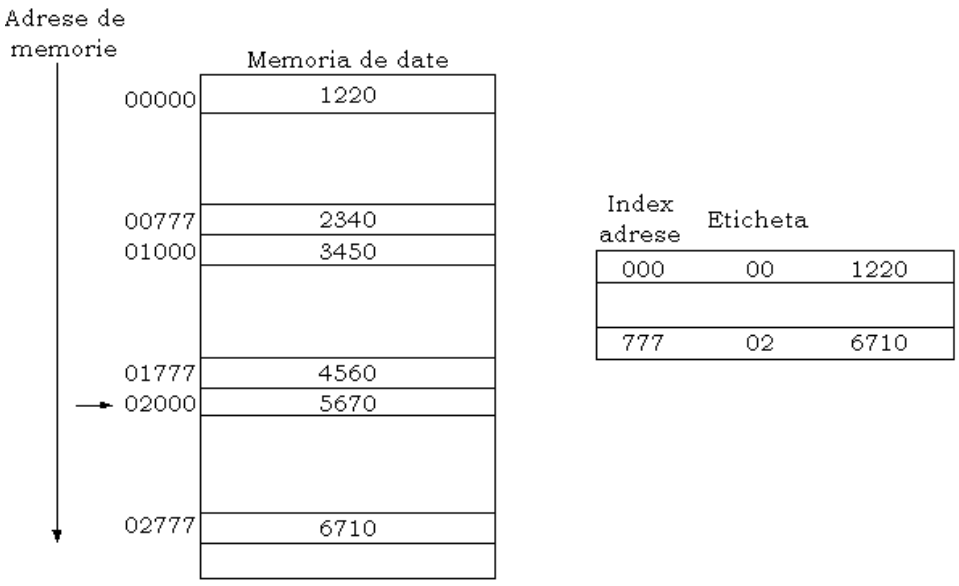


Figura 8.6: Exemplu mapare directă.

Cuvântul aflat la adresa 00000 este memorat în memoria cache. Procesorul doreşte să acceseze cuvântul aflat la adresa 02000. Indexul de adrese este 000 şi el va fi utilizat pentru accesarea memoriei cache. În urma comparaţiei dintre cele două etichete, se constată că rezultatul întors este MISS, deoarece eticheta memoriei cache este 00, iar eticheta adresă este 02.

Se accesează memoria principală şi cuvântul 5670 este transferat către CPU. Cuvântul din memoria cache de la 000 este replasat cu eticheta 02 şi data 5670.

Maparea asociativă pe mai multe căi

În cadrul acestei metode de organizare a memoriei cache, fiecare cuvânt al memoriei cache poate memora unul sau mai multe cuvinte din memorie sub acelaşi index de

adresă. În figura 8.7 este prezentat un exemplu de memorie cache cu o organizare asociativă pe mai multe căi.

Index adrese	Eticheta	Date	Eticheta	Date
000	01	3450	02	5670
777	02	6710	00	2340

Figura 8.7: Maparea asociativă pe mai multe căi.

Fiecare index de adresă referă două cuvinte de date şi etichetele asociate lor. Fiecare etichetă necesită 6 biți, iar fiecare cuvânt 12 biți, rezultând lungimea cuvântului de $2 \cdot (6+12) = 36$ biți. Un index de adrese de 9 biți poate acoperi 512 cuvinte deci dimensiunea memoriei cache este de 512 x 36. Această memorie cache poate acoperi un număr de 1024 cuvinte ale memoriei principale deoarece un cuvânt din memoria cache conține două cuvinte dată.

Când CPU generează o cerere de memorie, valoarea index a adresei este utilizată pentru accesarea memoriei cache. Câmpul etichetă din adresa CPU este comparat cu cele două etichete din memoria cache în vederea stabilirii unei potriviri. Compararea logică este realizată printr-o căutare asociativă de etichete într-o mulțime. În caz de MISS, algoritmii de înlocuire cei mai frecvent utilizați sunt FIFO şi LRU.

Scrierea în memoria cache

În cazul în care CPU generează semnal de scriere în memoria principală, există două metode care se pot aplica:

1. reactualizarea informației din memoria principală în paralel cu reactualizarea informației din memoria cache dacă cuvântul se găseşte în memoria cache, la adresa specificată. Această procedură se numeşte WRITE-THROUGH.

2. WRITE-BACK. În cadrul acestei metode, doar locația memoriei cache este reactualizată în cadrul operației WRITE. Locația reactualizată este marcată şi doar în caz de suprascriere a informației se va face reactualizarea locației memoriei principale.

2. Desfăşurarea lucrării

1. Se va utiliza un simulator de memorie cache pentru a simula diferite organizări

ale memoriei cache pentru primele 1 milion de referinţe în monitorizarea execuţiei programului ***gcc***. Sunt disponibile (http://www.csit-sun.pub.ro/resources) atât ***dinero*** (simulatorul de memorie cache), cât şi ***gcc.*** Se presupune o memorie cache de instrucţiuni de 32kB şi o memorie cache de date de 32kB folosind aceeaşi organizare. Să se aleagă cel puţtin două tipuri de asociativităţi şi două dimensiuni de bloc. Să se deseneze o diagramă ce prezintă organizarea unei memorii cache cu cea mai bună rată de eşec.

2. Se va studia influenţa folosirii unui nivel secundar de memorie cahe asupra performanţei unui procesor. Se presupune existenţa unui procesor cu un CPI (numărul de cicluri/instrucţiune) de bază de 1.0 şi o frecvenţă de ceas de 500MHz. Se consideră că toate referinţele la memorie vor avea succes în memoria cache primară. Memoria principală are un timp de acces de 200ns, incluzând aici şi timpul necesar tratării cazurilor de eşec. Frecvenţa de eşec pe instrucţiune în memoria cache primară este de 5%. Care va fi creşterea de viteză a maşinii dacă este adăugat un nivel suplimentar de memorie cache cu un timp de acces de 20ns, atât pentru eşec, cât şi pentru succes şi suficient de mare pentru a reduce frecvenţa de eşec la memoria principală la 2%?

3. Probleme propuse

1. Pe baza hărţii adreselor de memorie stabilită în cadrul exemplului l, să se proiecteze circuitul care realizează conexiunea memoriei cu CPU.

2. Pe baza organizării hardware a unei celule de memorie asociativă prezentată în figura 8.3, să se proiecteze o memorie asociativă cu m = 4 şi n = 3. Să se realizeze simularea operaţiilor de scriere/citire în/din memoria asociativă proiectată.

3. Se reconsideră exemplul prezentat în figura 8.6. În cadrul exemplului mărimea blocului de memorie are dimensiunea de l cuvânt. Cum arată aceeaşi proiectare dacă se utilizează un bloc de capacitate 8 cuvinte ?

4. Enunţaţi cel puţin un avantaj al folosirii metodei WRITE-THROUGH de scriere în memoria cache.

5. Folosind limbajul de asamblare pentru MIPS şi simulatorul SPIM exemplicaţi toate modurile de adresare la memorie cunoscute. Spre exemplu, pentru a exemplifica modul de adresare indirectă cu autodecrementare se poate scrie programul MIPS din figura 8.8

```
# adresare indirectă cu autodecrementare
     .text
     la         $a1,  0x0040001c
```

```
      sub         $a1, $a1, 0x00000004
      lw          $t1, ($a1)
      lw          $t2, ($t1)
pc:   not         $t1, $t2
adr:  .word       0x00400020
      li          $v0, 1
op:   .word 7
      move $a0, $t1
      syscall
end:  mop
```

Figura 8.8: Program MIPS pentru exemplificarea modului de adresare indirectă cu autodecrementare a memoriei principale.

9

Transmisia şi recepția serială a informației

1. Prezentare teoretică

La sistemele de calcul se cuplează o mare varietate de echipamente periferice. Aceste echipamente, fiind produse de diverse firme, nu sunt complet compatibile între ele ca mod de dialog şi transfer de informație, deşi din punct de vedere logic realizează aceleaşi funcții. Prin urmare, a apărut necesitatea unei standardizări şi unificări a echipamentelor de transmisie de date.

Transferul de date între echipamentul periferic şi calculator se poate face din punct de vedere logic, paralel sau serial folosind diverse tehnologii: infraroşu, usb, Bluetooth etc.

Transferul serial are avantajul că asigură o fiabilitate mai mare transmisiei în comparație cu transferul paralel, în schimb necesită resurse fizice mai complexe.

Norme de transmitere serială a informației. Protocolul de transmisie.

Între două dispozitive cuplate ce transmit datele serial asincron trebuie să existe o linie de referință (masa electrică), 2 linii pentru date şi 2 linii pentru comenzi şi stări, după cum se poate observa în figura 9.1. Conform normei de transmisie CCITT, semnalele de legătură au următoarea semnificație:

- 103 = date transmisie
- 104 = date recepționate
- 105 = cerere pentru emisie
- 106 = gata de emisie
- 107 = conectat la linie
- 108 = conectează la linie

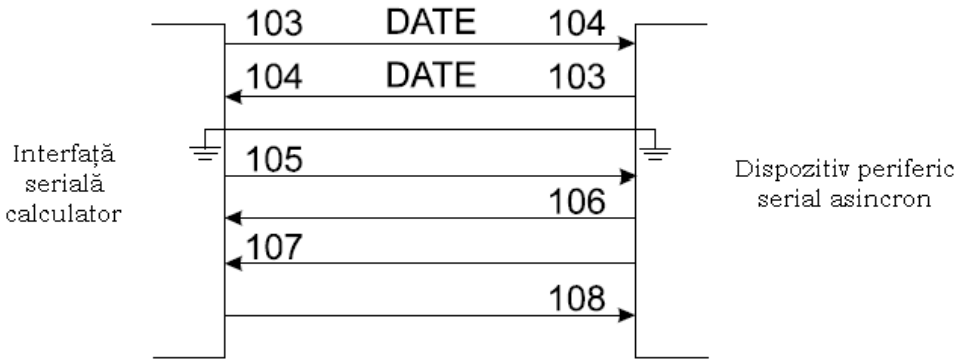


Figura 9.1: Norma de transmisie CCITT.

Semnalele ce se transmit pe aceste linii sunt sub forma unor nivele de tensiune în logică negativă. Astfel, 1 logic este considerat între -6V şi -12V, iar 0 logic este considerat între +6V şi +12V. Avantajele unei astfel de alegeri sunt:

- tensiunea de referință 0;
- o pană de alimentare se poate deosebi de oricare din cele două stări;

- imunitate la zgomot.

Protocolul de transmitere date

Acest protocol poate fi urmărit în figura 9.2 şi este compus din următoarele secvențe:

- Dispozitivul transmite continuu l logic când este inactiv.
- Datele transmise pe o singură linie (103) sunt precedate de un bit de start (0 logic) şi urmate de l sau 2 biți de stop (l logic).

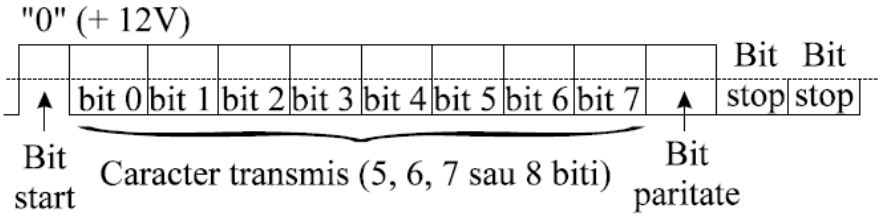


Figura 9.2: Protocolul de transmisie a datelor.

Semnalele de comandă şi stare servesc numai pentru stabilirea legăturii între dispozitive. Dintre dispozitivele care lucrează serial asincron, care corespund standardului CCITT, se pot menționa: DISPLAY-uri, CONSOLE, MODEM-uri.

Proiectarea unui dispozitiv de transmisie serială asincronă

Un dispozitiv de transmisie serială asincronă se compune din:

1. registru de transmisie
2. logica de generare a parității
3. numărător de biți

Schema bloc a unui dispozitiv de transmisie este prezentată în figura 9.5. Semnificația semnalelor prezente în schema bloc este următoarea:

- rxrdy = recepție terminată;

- rdy = transmițător liber;
- txrdy = transmisie terminată;
- DDISP = date disponibile;
- DREQ = cerere date;
- DPL = deplasare date.

Proiectarea unității de comandă prezentată în figura 9.4 se realizează utilizând diagrama logică prezentată în figura 9.3.

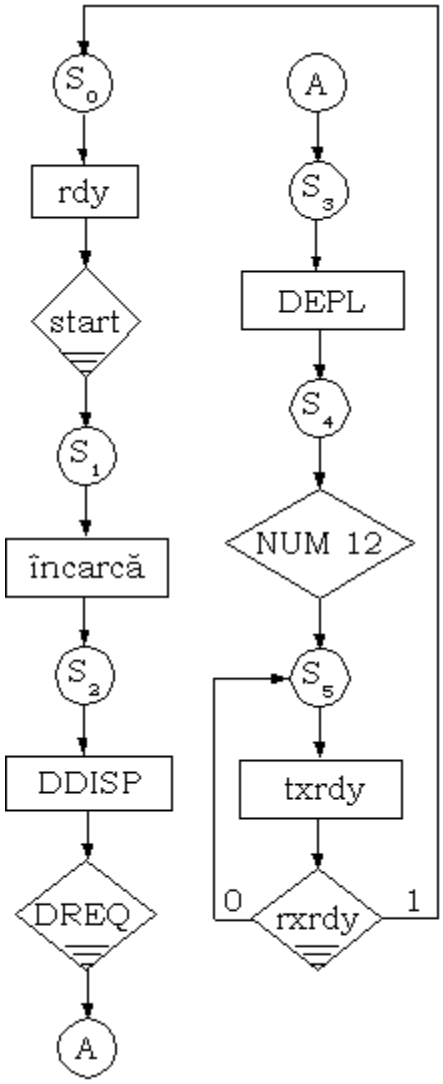


Figura 9.3: Diagrama logică a unității de comandă pentru transmisia serială a informației

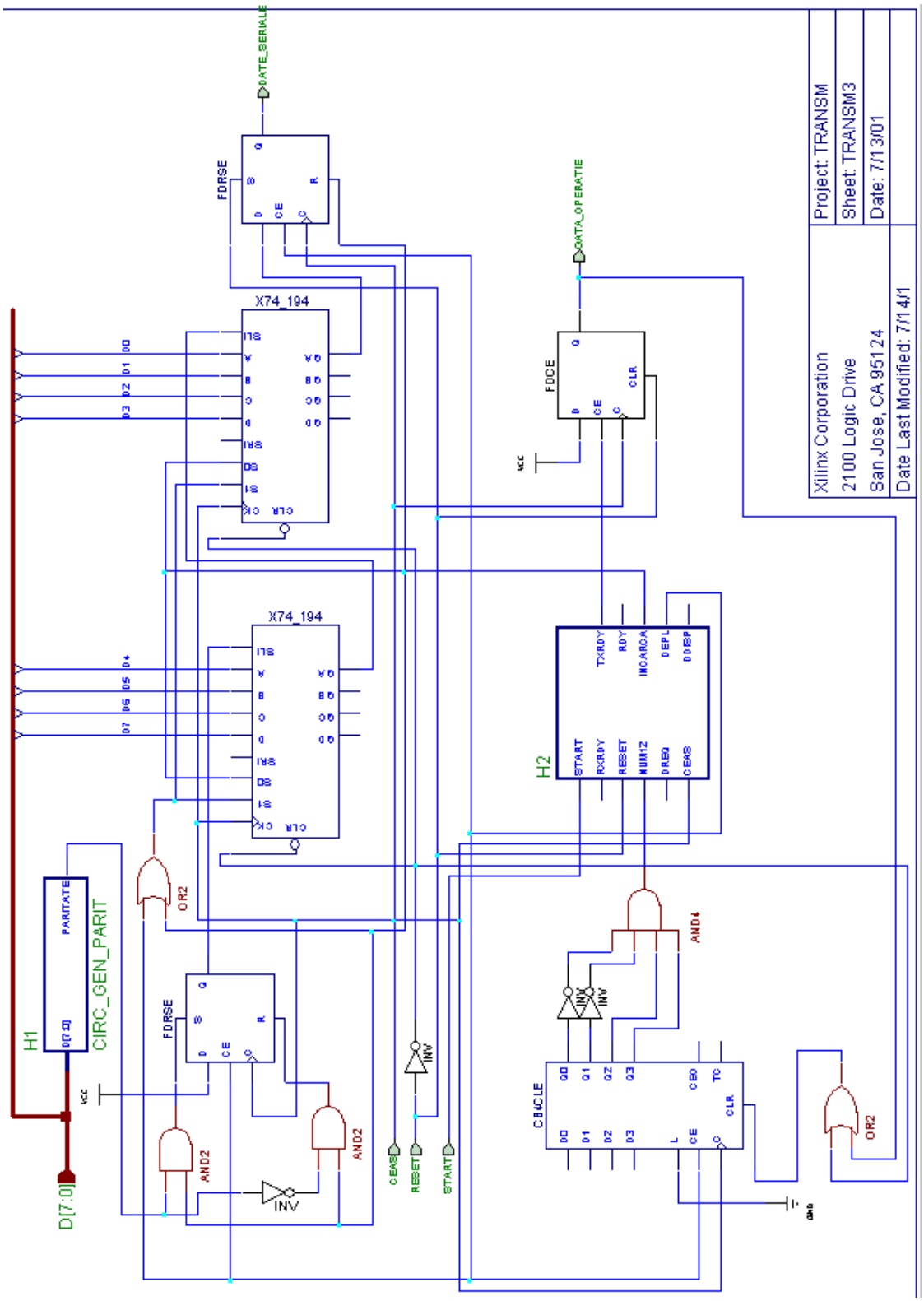


Figura 9.4: Schema bloc a protocolului de transmisie.

Recepția serială a informației

În proiectarea unui dispozitiv de recepție serială a informației se pornește de la protocolul de transmisie serială asincronă prezentat în figura 9.5.

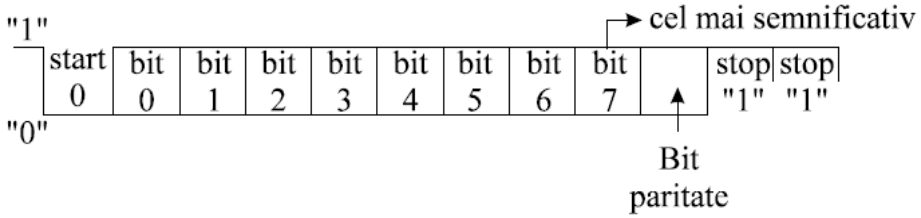


Figura 9.5: Protocolul de transmisie serială asincronă.

Resursele harware utilizate în proiectare sunt:

- Două registre de deplasare de 4 biți în care se asamblează informația recepționată serial;
- Trei bistabile care indică modul de desfăşurare a operației de recepție;
- Bistabilul DA = bistabil care indică faptul că datele au fost acceptate;
- Bistabilul PE = bistabil care indică apariția unei erori de paritate;
- Bistabilul SE = bistabil care indică bit de stop invalid.

Schema bloc a dispozitivului de recepție serială a informației este prezentată în figura 9.7. Informația recepționată serial este asamblată în registrele de deplasare de 4 biți. Având în vedere faptul că informația este precedată de un bit de start egal cu 0, se poate detecta recepția completă a caracterului, fără numărarea biților în felul următor:

- se încarcă inițial registrele de deplasare cu l logic;
- se testează valoarea bitului BIT_9. Când aceasta a devenit 0 (bitul de start a ajuns în această poziție) înseamnă că s-a terminat recepționarea caracterului.

Pentru a asigura o bună funcționare a dispozitivului de recepție este bine ca preluarea unui bit de informație să se facă la un moment de timp cât mai aproape de mijlocul bitului. Preluarea bitului se face prin aplicarea semnalului de deplasare *DEPL* registrelor de deplasare.

Pentru a genera semnalul *DEPL* cât mai aproape de mijlocul bitului, se aplică automatului de comandă o frecvență de 8 sau 16 ori mai mare decât frecvența de transmisie serială. După sesizarea bitului de *START,* se lasă automatul să evolueze prin 4, respectiv 8 stări, şi apoi se generează semnalul *DEPL*.

Pentru biții următori, automatul de comandă trece prin 8 respectiv 16 stări, pentru a genera noi semnale *DEPL*. Diagrama unității de comandă este prezentată în figura 9.6.

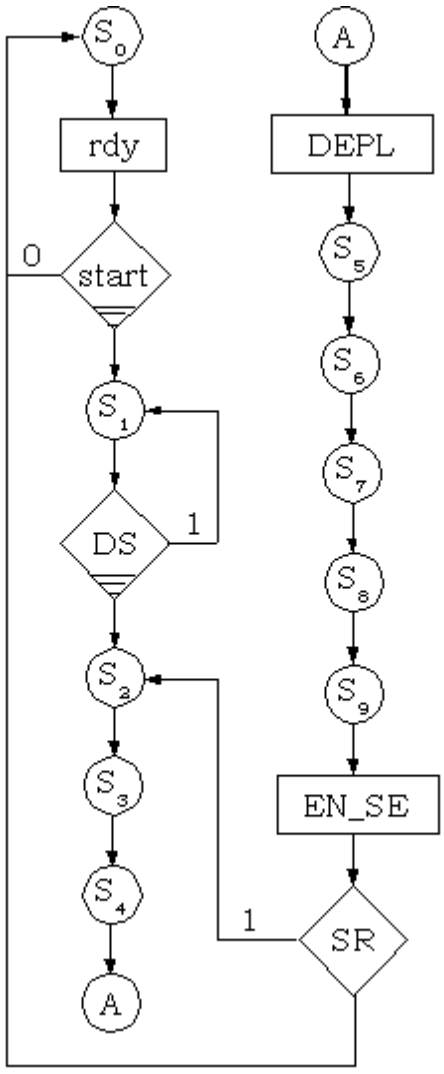
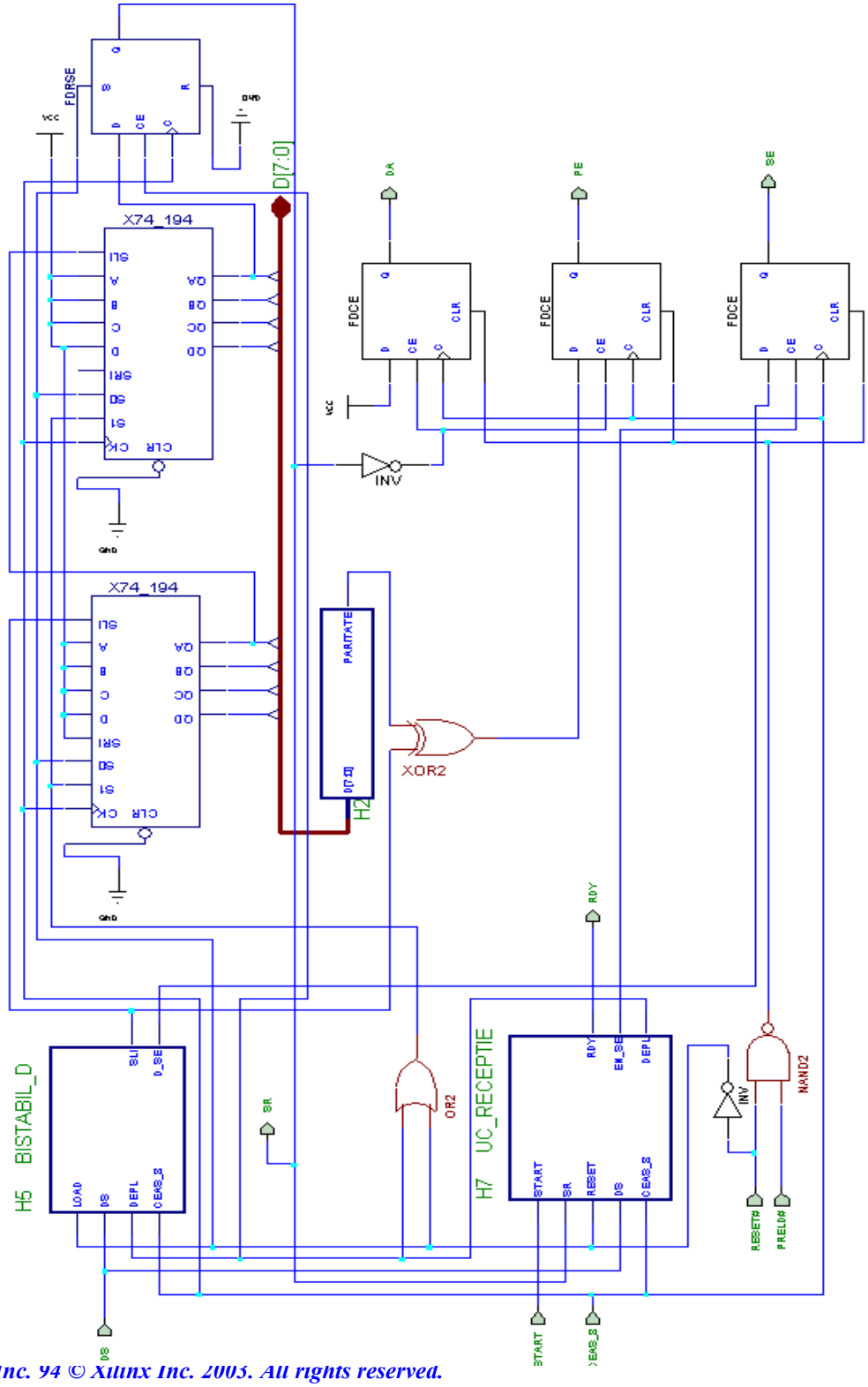


Figura 9.6: Diagrama logică a unității de comandă pentru recepția informației.



[Figure

Figura 9.7: Schema bloc pentru recepția serială a informației

2. Desfăşurarea lucrării

1. Se va proiecta în Verilog şi realiza utilizând Xilinx WebPACK ISE 6.2i schema bloc a protocolului de transmisie prezentată în figura 9.4. Verificarea funcționării schemei bloc se va realiza cu ajutorul simulatorului ModelSim.

3. Probleme propuse

1. Se va proiecta în Verilog şi realiza utilizând Xilinx WebPACK ISE 6.2i schema bloc pentru recepția serială a informației prezentată în figura 9.7. Verificarea funcționării schemei bloc se va realiza cu ajutorul simulatorului ModelSim.

10

Coduri detectoare/corectoare de erori. Criptarea informației

1. Prezentare teoretică

În cadrul acestei lucrări de laborator se vor prezenta algotimii CRC şi Reed-Solomon folosiți la detectarea şi corectarea erorilor care pot apărea într-o transmisie de date. Algoritmii RSA şi IDEA prezentați sunt uzual folosiți pentru criptarea informației şi se bazează pe chei publice. Implementările hardware ale altor algoritmi de criptare, care se bazează pe metode tradiționale (de exemplu algoritmul de criptare DES), pot fi studiate la http://www.csit-sun.pub.ro/resources.

Sume de control

Scopul unei tehnici de detecție a erorilor este acela de a pune la dispoziția receptorului unui mesaj, transmis printr-un canal cu zgomote (pasibil de introducere de erori), o metodă de a determina dacă mesajul a fost corupt sau nu. Pentru a face posibil acest lucru, emițătorul construieşte o valoare numită sumă de control care este o funcție de mesaj şi o anexează acestuia. Receptorul poate să folosească aceeaşi funcție pentru a calcula suma de control pentru mesajul primit, iar apoi să o compare cu suma de control anexată (concatenată mesajului) pentru a vedea dacă mesajul a fost receptat corect.

Exemplu Să se aleagă o funcție care are ca rezultat (sumă de control) suma octeților din mesaj modulo 256:

$$f(x) = \sum(octeti \quad mesaj) \bmod 256 \qquad (10.1)$$

Considerând toate valorile în zecimal, se obține:

mesaj	:	7 24 3
mesaj cu suma de control	:	7 24 3 34
mesaj după transmisie	:	7 28 3 38

Al doilea octet al mesajului a suferit o modificare în timpul transmisiei, de la 24 la 28. Cu toate acestea, receptorul poate determina prezența unei erori comparând suma de control transmisă (34) cu cea calculată (38 = 7 + 28 + 3).

Dacă însăşi suma de control este coruptă, un mesaj transmis corect poate fi (incorect) interpretat drept unul eronat. Acesta nu este însă un eşec periculos. Un eşec periculos are loc atunci când atât mesajul cât şi suma de control se modifică astfel încât rezultă într-o transmisie consistentă intern (interpretată ca neavând erori).

Din păcate, această posibilitate nu poate fi evitată şi cel mai bun lucru care se poate realiza este de a minimiza probabilitatea ei de apariție prin creşterea cantității de informație din suma de control (de exemplu, lărgind dimensiunea ei la doi octeți în loc de unul).

Coduri CRC

Ideea de bază pentru algoritmii CRC este de a trata mesajul drept un număr reprezentat în binar, de a-l împărți la un alt număr binar fixat şi de a considera restul drept sumă de control. La primirea mesajului, receptorul poate efectua aceeaşi împărțire şi poate compara restul cu suma de control primită (restul transmis).

Exemplu Considerând că mesajul care trebuie transmis este alcătuit din 2 octeți (6, 23), el este reprezentat în baza 16 ca numărul 0617 şi în baza 2 ca 0000_0110_0001_0111. Se presupune folosirea unei sume de control de 1 octet şi a unui împărțitor constant 1001. Atunci suma de control va fi restul împărțirii 0000_0110-0001_0111 : 1001 = ... 0000010101101, rest 0010. Mesajul transmis de fapt va fi: 06172, unde 0617 este mesajul inițial (informația utilă), iar 2 este suma de control (restul).

Aritmetica binară fără transport

Toate calculele executate în cadrul algoritmilor CRC sunt realizate în binar, fără transport. Deseori se foloseşte denumirea de aritmetică polinomială, dar în continuare se va folosi denumirea de aritmetică CRC deoarece la implementarea cu polinoame s-a renunțat.

Adunarea a două numere în aritmetica CRC, aşa cum se poate observa în figura 10.1, este asemănătoare cu adunarea binară obişnuită, însă nu există transport. Aceasta înseamnă că fiecare pereche de biți corespondenți determină bitul corespondent din rezultat, fără

nici o referință la alt bit din altă poziție (așa cum se poate observa din exemplul prezentat în figura 10.2 a).).

Definiția operației de scădere este identică cu operația de adunare şi poate fi observată în figura 10.1, iar un exemplu este prezentat în figura 10.2 b).

Se poate concluziona că atât adunarea cât şi scăderea în aritmetica CRC sunt echivalente cu operația SAU EXCLUSIV (XOR), iar operația XOR este propria sa inversă. Acest fapt reduce operațiile primului nivel de putere (adunare, scădere) la una singură, care este propria sa inversă (o proprietate foarte convenabilă a acestei aritmetici).

a	b	a + b
0	0	0
0	1	1
1	0	1
1	1	0

a	b	a - b
0	0	0
0	1	1
1	0	1
1	1	0

Figura 10.1: Definirea operațiilor de adunare/scădere.

Pe baza adunării, se poate defini şi înmulțirea, care se realizează natural, fiind suma dintre primul număr deplasat corespunzător şi cel de-al doilea număr (se foloseşte adunarea CRC). Un exemplu pentru această operație este prezentat în figura 10.2 c).

Pentru realizarea operației de împărțire, este nevoie să se cunoască când *un număr este cuprins în altul*. De aceea, se va considera următoarea definiție: *X* este mai mare decât sau egal cu *Y* dacă poziția celui mai semnificativ bit l al lui X este mai mare sau aceeaşi cu poziția celui mai semnificativ bit l al lui Y. Un exemplu complet este prezentat în figura 10.2 d).

Transmisia - recepția datelor folosind CRC

Aşa cum s-a arătat până acum, calculul CRC este de fapt o simplă împărțire. Pentru realizarea unui calcul CRC este nevoie de un divizor, denumit în limbaj matematic *polinom generator*. Lungimea polinomului uzuală este de 16 sau 32 de biți, CRC-16, CRC-32, şi aceste dimensiuni sunt folosite în calculatoarele digitale moderne. *Lungimea unui polinom - W-* este de fapt poziția celui mai semnificativ bit l (lungimea polinomului 10011 este 4).

La transmițător, înainte de calculul CRC, se adaugă *W* biți cu valoarea 0 la sfârşitul mesajului care va fi împărțit folosind aritmetica CRC la polinom, astfel încât toți biții mesajului să participe la calculul CRC. Un exemplu este prezentat în figura 10.2 d). Împărțirea produce un cât, care nu va fi ignorat şi un rest, care este suma de control calculată (CRC-ul). În mod uzual CRC-ul este apoi adăugat mesajului, iar rezultatul este trimis către receptor, în acest caz se transmite 11010110111110.

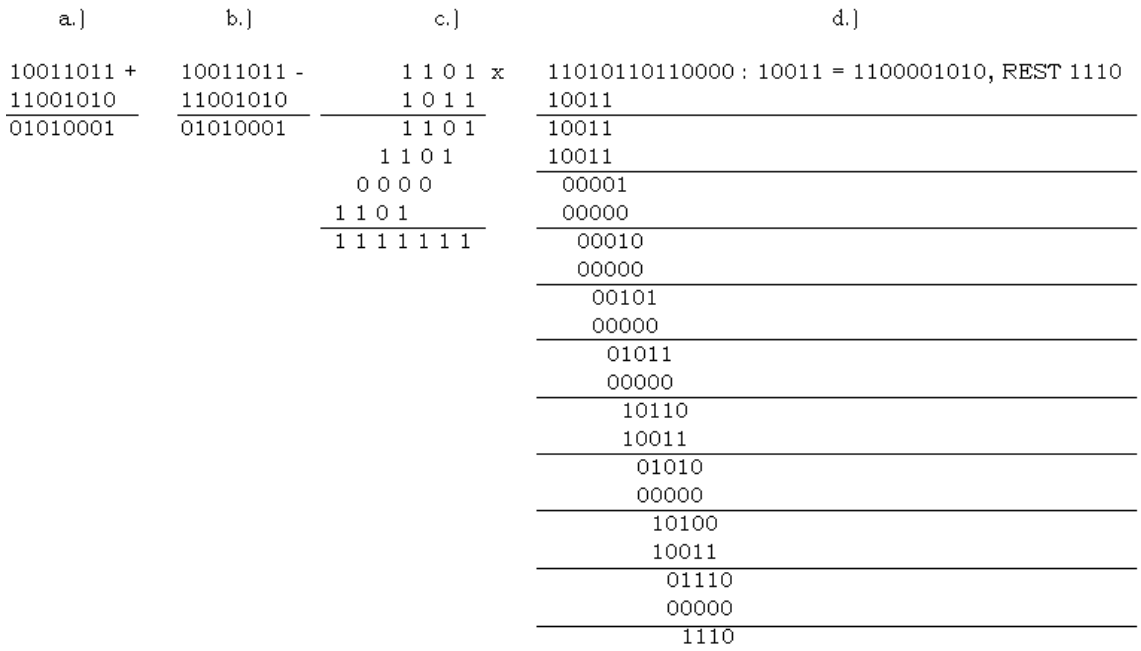


Figura 10.2: Exemplificarea operațiilor binare fără transport

Receptorul calculează suma de control pentru întreg mesajul primit (fără adăugare de zerouri) şi compară restul cu 0. Realizarea acestei operații este motivată de faptul că mesajul transmis *T* este multiplu de polinomul folosit drept divizor.

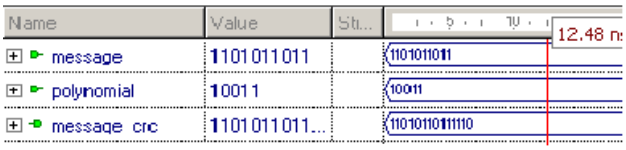
Implementarea directă

CRC-ul se poate calcula utilizând noțiunile teoretice prezentate până acum. Algoritmul, implementarea Verilog precum şi rezultatele simulării pot fi observate în figura 10.3.

Implementarea bazată pe tabelă

Acest algoritm este o variantă îmbunătățită a algoritmului anterior, el fiind foarte eficient deoarece implică doar o deplasare, o operație SAU, o operație SAU EXCLUSIV şi un acces la memorie pentru fiecare octet. Algoritmul precum şi rezultatele implementării sale în Verilog sunt prezentate în figura 10.4.

```
module crc_simple(message, polynomial, message_crc);
        /* Parametrii: */
        parameter crc_width = 4;  /* lungimea CRC-ului; */
        parameter msg_width = 10; /* lungimea mesajului; */
        /*Intrari, iesiri si resurse fizice interne*/
        /* -> masajul initial: */
        input  [msg_width - 1 : 0] message;
        wire   [msg_width - 1 : 0] message;
        /* -> polinomul generator: */
        input  [crc_width : 0] polynomial;
        wire   [crc_width : 0] polynomial;
        /* -> mesajul cu CRC-ul adaugat: */
        output [msg_width + crc_width - 1 : 0] message_crc;
        reg    [msg_width + crc_width - 1 : 0] message_crc;
        reg    [crc_width - 1 : 0] crc; /* the CRC register; */
        reg    msb_crc;/*cel mai semnificativ bit al registrului CRC; */
        /* resurse logice: */
        integer count; /* numarator; */
        /* Calculeaza CRC-ul si atasare: */
        always @(message | polynomial) begin
                /* Initializare: */
                message_crc = message;
                message_crc = message_crc << crc_width;
                crc = 'b0;
                /* Calculeaza CRC-ul: */
                for (count = msg_width + crc_width - 1; count >= 0; count = count - 1)
                 begin
                        msb_crc = crc[crc_width - 1];
                        crc = crc << 1;
                        crc[0] = message_crc[count];
                        if (msb_crc == 1) begin
                                crc = crc ^ polynomial;
                        end
                end
                /* Adauga CRC-ul mesajului: */
                message_crc[crc_width - 1 : 0] = crc;
        end
endmodule
```



Algoritm:

1. încarcă registrul cu biți 0
2. adaugă la sfârşitul mesajului W biți 0.
3. cât timp (mesajul mai are biți) deplasează registrul stânga cu un bit, introducând următorul bit din mesaj în poziția 0
4. dacă (un bit 1 a fost scos din registru) registru = registru XOR polinom
5. registrul conține restul

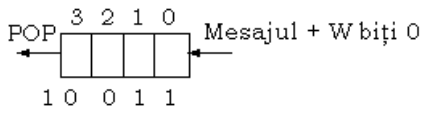


Figura 10.3: Implementare directă

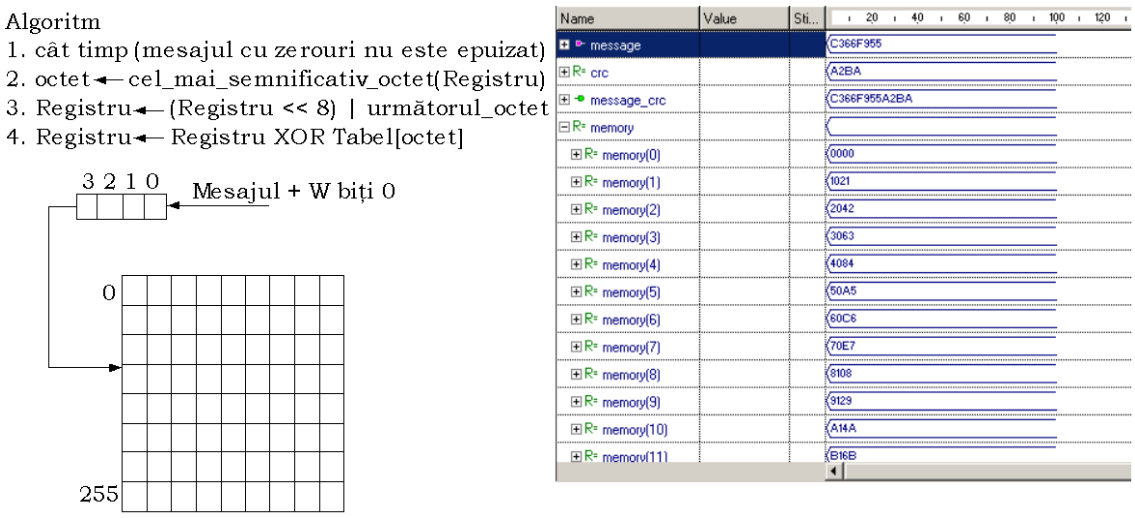


Figura 10.4: Implementarea bazată pe tabelă

Coduri Reed-Solomon

Codurile Reed-Solomon (RS) sunt coduri corectoare de erori în bloc inventate în 1960 de Irving Reed şi Gustave Solomon. Aceste coduri au început să fie utilizate începând cu 1990, atunci când progresele tehnologice au făcut posibilă trimiterea datelor în cantități mari si la viteze ridicate. Actualmente aceste coduri sunt utilizate într-o gamă largă de echipamente electronice cum sunt:

- dispozitivele pentru stocarea datelor (CD, DVD, hard-disk);
- telefoanele mobile;
- echipamentele folosite în comunicațiile prin satelit;
- televiziunea digitală;
- modemurile de mare viteză (ADSL, xDSL).

Realizarea unei transmisii folosind codurile RS presupune ca, codificatorul RS să preia un bloc de date şi să adauge o informație suplimentară caracteristică. Una dintre caracteristicile importante ale codului RS constă în faptul că acest cod va codifica grupuri de simboluri de date.

Decodificatorul RS procesează fiecare bloc şi încearcă să corecteze erorile apărute şi să recupereze datele trimise original.

Un cod RS este specificat ca *RS(n, k)* cu simboluri de *s* biți. Această descriere semnifică faptul că, codificatorul preia *k* simboluri de paritate astfel încât să rezulte un cuvânt de cod de *n* simboli. Sunt *n* - *k* simboluri de paritate, de câte *s* biți fiecare. Un decodificator RS poate corecta până la *t* simboluri ce conțin erori, cu $2t = n - k$.

Un cod RS este obținut împărțind mesajul original în blocuri de lungime fixă. Fiecare bloc este apoi împărțit în simboluri de *m* biți. Fiecare simbol are lungime fixă (între 3 si 8 biți). Natura liniară a acestui cod asigură faptul că fiecare cuvânt de *m* biți este valid pentru codificare astfel încât se pot transmite date binare sau text.

Exemplu Un cod des folosit este RS(255, 233) cu simboluri de 8 biți. Fiecare cuvânt de cod conține 255 de simboluri din care 233 sunt de date şi 22 sunt de paritate. Pentru acest cod se pot stabili următoarele relații: $n = 255$, k — 233, $s = 8$, $t = 16$.

Codurile RS pot fi scurtate dacă la codificator se fac anumiți biți zero, nu se transmit dar sunt adăugați la decodificator. Spre exemplu, codul RS(255, 233) poate fi scurtat la (200, 168). Operațiile realizate de codificator sunt următoarele:

- se preia un bloc de 168 de biți de date;
- se adaugă virtual 55 de biți de zero creând astfel un cod (255, 233);
- se transmit doar 168 biți de date şi 32 biți de paritate.

Un decodificator RS poate corecta un număr de *t* erori şi până la *2t* ştersături. La decodificarea unui cuvânt RS pot apărea următoarele variante:

- dacă $2s + r < 2t$ atunci codul original transmis poate fi corectat în întregime;
- decodificatorul indică faptul că nu poate reface codul original;
- decodificatorul va genera un cuvânt decodat cu erori şi nu va fi semnalat acest lucru.

Arhitectura decodorului poate fi urmărită în figura 10.5.

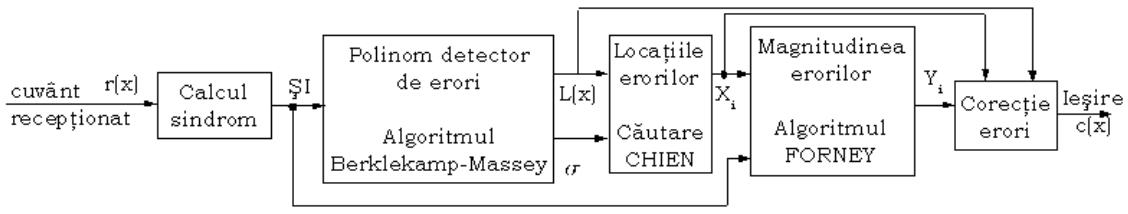


Figura 10.5: Arhitectura unui decodificator RS.

Algoritmul de criptare RSA

Algoritmul RSA este un sistem criptografic ce utilizează chei publice şi a fost creat de un grup de cercetători de la MIT (Massachusetts Institute of Technology) cu scopul de a asigura securitatea datelor schimbate prin intermediul Internet-ului.

Metodele tradiționale de criptare (spre exemplu algoritmul DES - implementările hardware şi JAVA precum şi simulările acestor implementări pot fi vizualizate la http://www.csit-sun.pub.ro) folosesc un număr de $\frac{n \cdot (n-1)}{2}$ chei, în timp ce algoritmii bazați pe chei publice utilizează un număr de cel mult n chei publice.

O altă deosebire constă în faptul că în sistemele tradiționale de criptare, cheia de criptare trebuie ținută secretă deoarece ea trebuie utilizată în cadrul procesului de decriptare. În cazul criptării cu chei publice, cheia de criptare/decriptare nu mai este trimisă receptorului, deci canalul de comunicație dintre transmițător şi receptor poate să nu fie securizat.

Utilizarea algoritmului RSA implică crearea a două chei de către transmițător: *una publică* şi una *privată.* Cheia publică este trimisă oricărui destinatar la care trebuie trimis mesajul criptat. Cheia privată sau secretă este utilizată pentru decriptarea mesajului criptat cu ajutorul cheii publice.

Modalitatea de realizare a unei comunicații criptate cu ajutorul algoritmului RSA este prezentată în figura 10.6.

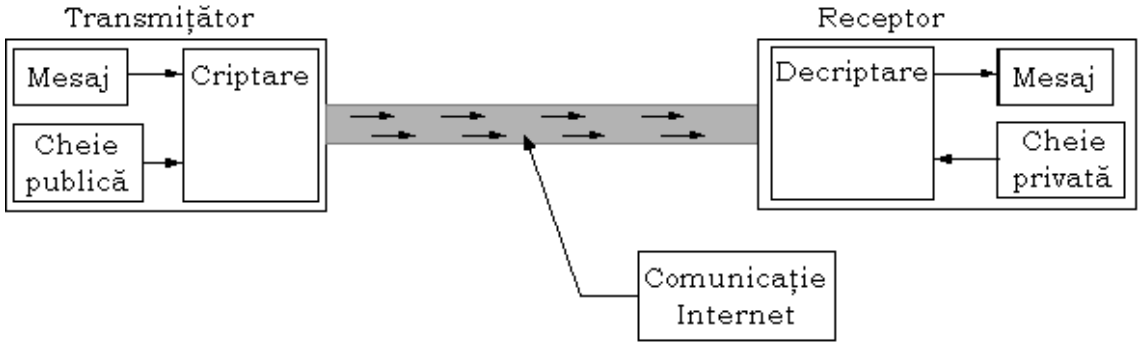


Figura 10.6: Arhitectura unui decodor RS.

Transmisia folosind algoritmul RSA necesită parcurgerea a două etape importante:

1. Generarea cheilor - se generează două chei una publică şi una privată. Pentru aceasta trebuie parcurşi următorii paşi:

    1. se aleg două numere prime p şi q cu aceeaşi magnitudine (lungime) şi se generează numărul $n = p \cdot q$;

2. se determină $\Phi = (p-1)\cdot(q-1)$;

3. se alege *e* ca fiind un număr prim în raport cu Φ, deci cel mai mare divizor comun (notat gcd(e, Φ)) al celor două numere trebuie să fie 1. În implementările practice valoarea lui *e* este aleasă ca fiind un număr prim Fermat (3, 5, 17, 65537,...);

4. se determină valoarea *d* care reprezintă inversiunea modulară a lui *e* şi Φ:

$$d = rest\left(e - \frac{1}{\Phi}\right)$$

Cheia publică este alcătuită din perechea *(n, e),* cât timp cheia privată este formată din perechea *(n, d).* Implementarea hardware a celui mai mare divizor comun se realizează cu ajutorul algoritmului lui Euclid.

```
Algoritm EuclidExtins(a, b)
     if b = 0 then
          return (a, 1, 0)
     else
          (d', x', y') = EuclidExtins(b, rest(a/b))
     return (d', y', x' - a/b · y')
```

2. Transmisia informației - În cadrul acestei etape, atât transmițătorul cât şi receptorul trebuie să execute câteva operații distincte. Transmițătorul realizează următoarele operații:

   a. obține cheia publică *(n, e)* de la receptor;

   b. convertește mesajul într-o mulțime de întregi pozitivi;

   c. calculează textul criptat conform relației: $c = m^e \ mod$ n;

   d. transmite mesajul c la receptor.

   Receptorul realizează următoarele operații:

   a. utilizează cheia privată *(n, d)* pentru a calcula $m = c^d \ mod \ n$;

   b. extrage textul din colecția de numere întregi *m*.

Algoritmul de criptarea IDEA

IDEA este un algoritm bazat pe chei publice care criptează blocuri de câte 64 de biți folosind o cheie de criptare de lungime 128 de biți. Criptarea şi decriptarea presupun utilizarea aceluiaşi algoritm. Implementarea acestui algoritm impune utilizarea a trei operații: *XOR, adunarea modulo 65536* şi *înmulțirea modulo 65537* care operează pe sub-blocuri de dimensiune 16 biți.

Funcționarea algoritmului constă în parcurgerea a opt paşi. Blocul de date de dimensiune 64 de biți este împărțit în 4 părți X_0, X_1, X_2 şi X_3, fiecare parte având dimensiunea de 16 biți. În fiecare pas, între cele 4 sub-blocuri se realizează o operație *XOR,* de adunare sau de înmulțire, împreună cu 6 subchei de dimensiune 16 biți fiecare.

Între paşii 2 şi 3, sub-blocurile sunt interschimbate, iar în final cele 4 sub-blocuri sunt combinate împreună cu 4 subchei pentru a forma ieşirea. În cadrul fiecărui pas al algoritmului se execută următoarea succesiune de operații:

- se înmulțeşte X_0 cu prima subcheie;
- se adună X_1 la a doua subcheie;
- se adună X_2 la a treia subcheie;
- se înmulțeşte X_3 cu a patra subcheie;
- XOR între rezultatele paşilor l şi 3;
- XOR între rezultatele paşilor 2 şi 4;
- se înmulțeşte rezultatul pasului 5 cu subcheia numărul 5;
- se adună rezultatele obținute în cadrul paşilor 6 şi 7;
- se înmulțeşte rezultatul de la pasul 8 cu subcheia numărul 6;
- se adună rezultatele obținute la paşii 7 şi 9;
- XOR între rezultatele paşilor l şi 9;
- XOR între rezultatele paşilor 3 şi 9;
- XOR între rezultatele paşilor 2 şi 10;
- XOR între rezultatele paşilor 4 şi 10.

Cele patru rezultate sunt sub-blocurile obținute în urma paşilor 11, 12, 13 si 14. Se interschimbă cele două sub-blocuri din mijloc şi astfel se obține intrarea pentru următorul pas. Excepție face ultimul pas în care nu se mai execută interschimbarea celor două sub-blocuri din mijloc. După pasul opt se execută următoarea secvență de operații pentru a determina rezultatul final:

- se înmulțeşte X_0 cu prima subcheie;
- se adună X_1 la a doua subcheie;
- se adună X_2 la a treia subcheie;
- se înmulțeşte X_3 cu a patra subcheie.

În final cele patru sub-blocuri se vor concatena pentru a forma blocul criptat de lungime 64 de biți.

Algoritmul utilizează 52 de subchei: 6 subchei pentru fiecare pas şi 4 subchei pentru pasul final. Generarea subcheilor porneşte de la cheia de lungime 128 de biți care se împarte în opt subchei. Acestea reprezintă primele opt subchei utilizate în algoritm. La pasul următor cheia este deplasată la stânga 25 de poziții şi apoi împărțită în opt părți. Acest proces de generare a subcheilor este continuat până se generează toate cele 52 de subchei necesare funcționării algoritmului.

Schema generală a algoritmului de criptare ***IDEA*** este prezentată în figura 10.8.

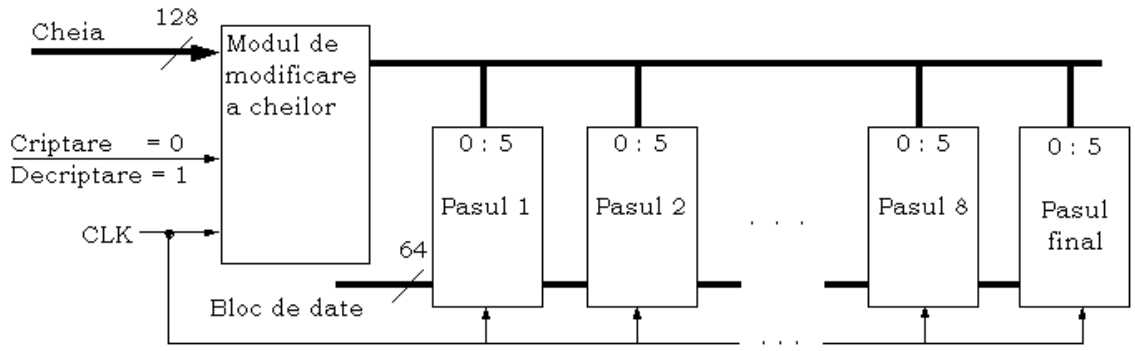


Figura 10.7: Schema generală a algoritmului de criptare cu chei publice IDEA.

2. Desfăşurarea lucrării

Se va proiecta în Verilog utilizând Xilinx WebPACK ISE 5.1i şi se va simula un circuit, care implementează algoritmul IDEA. Se va folosi schema generală prezentată în figura 10.7.

3. Probleme propuse

1. Să se proiecteze în Verilog utilizând Xilinx WebPack ISE 6.2i şi să se simuleze un circuit, care implementează algoritmul CRC bazat pe tabelă.

2. Să se proiecteze în Verilog utilizând Xilinx WebPack ISE 6.2i şi să se simuleze un circuit, care implementează algoritmul de criptare RSA.

Indicații

- Este bine să se calculeze o tabelă de conversie pentru fiecare dintre cele 256 valori de intrare posibile. Pentru a cripta mesajul va fi necesar doar accesul la o memorie locală care memorează tabela determinată. La decriptare se va utiliza acelaşi artificiu.

- Pentru a putea implementa în hardware expresia $m^e \ mod \ n$ se va utiliza următorul algoritm:

```
res = m;
for (i = 2; i<= e; i = i + 1) begin
  res = res * m;
  if (res > m) begin
        res = res % n;
  end
end
cypher(m, n, e) = res;
```

Anexa A

Prezentarea mediului de programare şi testare Xilinx WebPACK ISE 5.1i / 5.2i

A.1 Introducere

Xilinx WebPACK ISE 6. 2i reprezintă o soluție de proiectare a sistemelor numerice deosebit de complexă. Aceasta integrează pachete software pentru proiectarea cu ajutorul circuitelor FPGA sau CPLD, utilizând, în acest scop programe proprietare şi industriale.

Cu ajutorul pachetului software ISE 6.2 se poate proiecta, testa şi implementa o aplicație într-un timp foarte scurt. După ce testarea practică a aplicației, ce se realizează cu ajutorul circuitelor FPGA sau CPLD, demonstrează o funcționare corectă, se poate trece la implementarea în serie a structurii numerice respective, sub forma unui circuit specializat (ASIC).

În această anexă se va prezenta în detaliu mediul de proiectare şi testare Xilinx ISE 6.2.

Versiunea Xilinx ISE 6.2 poate fi obținută de la adresa: http://www.xilinx.com/ise_ classics/index.html.

Cu ajutorul mediului ISE 6.2 pot fi realizate următoarele operații:

1. Crearea de cod sursă în limbajul Verilog sau VHDL;

2. Generarea automată de cod pentru circuite combinaționale, circuite secvențiale şi

memorii;

3. Verificarea, din punct de vedere sintactic, a codului elaborat;

4. Simulare comportamentală şi la nivelul transferurilor între registre a proiectului sau numai a unor module ale proiectului;

5. Sinteză;

6. Plasarea şi rutarea proiectului într-un FPGA sau CPLD selectat de către proiectant;

7. Analiza timpilor de execuţie şi de comutare ai circuitului proiectat;

8. Impunerea de contrângeri de tipul arie sau constrângeri de tipul timp asupra circuitului proiectat;

9. Configurarea circuitului proiectat.

Fereastra principală a aplicaţiei se numeşte **Project Navigator** şi este împărţită în cinci secţiuni principale după cum se poate observa în figura A.l.

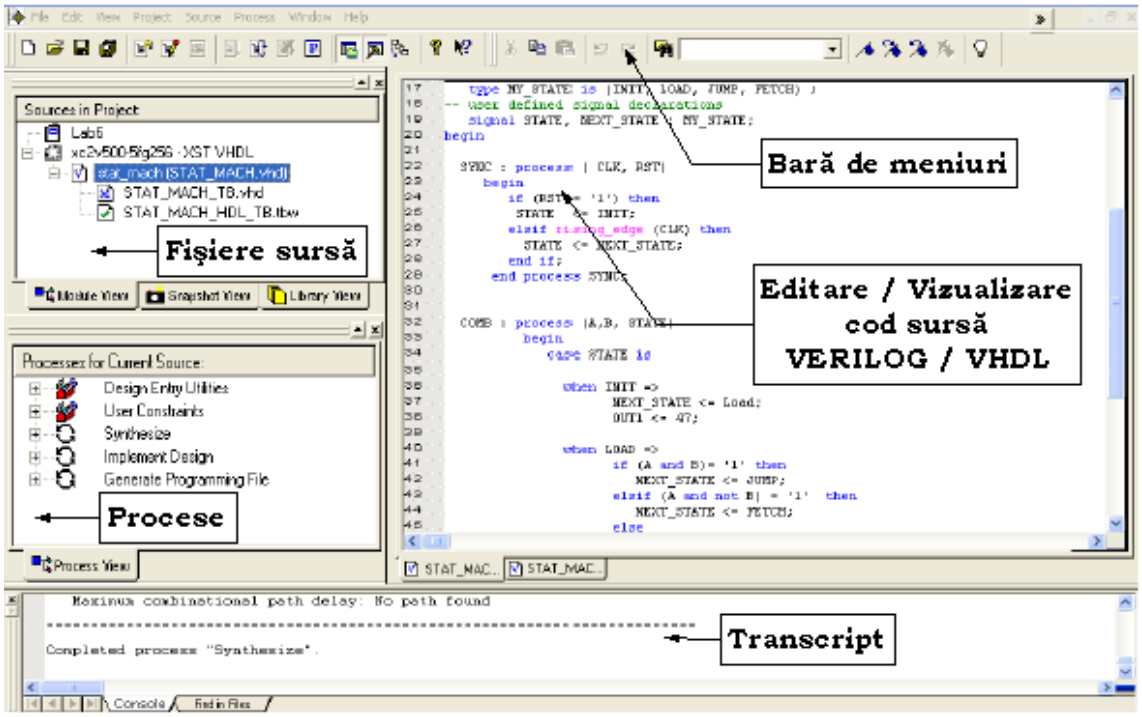


Figura A.1: Fereastra principală a aplicaţiei

- Prima secțiune – **Bară de meniuri** este folosită pentru:

    1. Creare sau deschidere de proiecte;

    2. Creare sau deschidere de fişiere sursa;

    3. Setarea anumitor parametri de funcționare pentru diferitele utilitare cuprinse în pachetul software ISE 6.2.

- În cea de a doua secțiune principală - ***Fişiere sursă*** sunt vizibile toate fişierele sursă din cadrul proiectului. Fişierele sunt prezentate sub forma unei ierarhii pentru a se putea selecta uşor şi rapid orice fişier sursă din cadrul proiectului. Tot în acesta secțiune sunt prezentate şi relațiile dintre fişiere, în figura A.l se pot observa cele trei subsecțiuni ale sale.

În prima subsecțiune *Module View* pot fi create şi vizualizate următoarele tipuri de fişiere:

1. Entități sau arhitecturi VHDL;

2. Module Verilog;

3. Generatoare de teste VHDL/Verilog;

4. Module generate automat de către ISE cu ajutorul utilitarului - ***Xilinx CORE Generator Modules*** **;**

5. Pachete;

6. Scheme grafice utilizând circuite care se regăsesc în bibliotecile Xilinx;

7. Fişiere de constrângere definite de către programator;

8. Fişiere de tipul EDIF Netlist;

9. Orice document de tip text definit de către programator.

Subsecțiunea *Snapshoot View* este dedicată salvării stărilor codului proiectului. Vizualizarea conținutului acestei subsecțiuni poate fi făcută în orice moment.

Subsecțiunea *Library* este utilă programatorului pentru a putea observa, adăuga/şterge, bibliotecile utilizate în cadrul proiectului. Această subsecțiune este utilă în special programatorilor care utlizează VHDL, îşi creează propriile biblioteci şi care vor putea fi reutilizate oricând în diferite alte proiecte.

- În secțiunea a treia ***Procese*** sunt prezentate toate procesele utilizate în crearea şi implementarea unui proiect. Aceste procese sunt:

    1. Verificarea sintaxei codului HDL;
    2. Sinteza şi elaborarea proiectului;
    3. Introducerea de constrângeri asupra proiectului;
    4. Verificare RTL;
    5. Plasarea şi rutarea modulelor proiectului;
    6. Analiza timpului;
    7. Simularea proiectului în timp şi afişarea formelor de undă;
    8. Planificarea proiectului în circuitul FPGA/CPLD şi definirea de contrângeri de tip arie;
    9. Configurarea dispozitivului.

Fiecare proces are o listă de parametri care pot fi modificați în funcție de cerințele proiectului, înainte de a rula procesul. Pentru a putea modifica parametrii unui proces, se selectează procesul dorit şi cu ajutorul butonului dreapta al mouse-ului se deschide fereastra care conține parametrii procesului.

- Secțiunea a patra ***Transcript*** este dedicată vizualizării progreselor realizate de fiecare proces implicat în realizarea circuitului FPGA/CPLD. Tot în această secțiune mai pot fi vizualizate mesajele de eroare sau de atenționare generate de fiecare proces, precum şi fişierul de comandă.

- Ultima secțiune, ***Editare/Vizualizare cod sursă VHDL/Verilog****, e*ste destinată vizualizării sau editării de fişiere sursă utilizate în cadrul proiectului. În această secțiune vor fi deschise spre consultare şi fişierele de tip raport, care sunt generate la terminarea fiecărui proces prezentat în secțiunea ***Procese .***

A.2 Proiectarea, sinteza şi implementarea unui circuit utilizând ISE 6.2

Procesele prezente în secțiunea ***Proces*** sunt afişate în ordinea în care ele sunt programate spre execuție. Proiectarea unui circuit în vederea implementării, în Xilinx ISE, presupune crearea unui proiect. Pentru a putea proiecta un circuit de dimensiuni mari, se recomandă utilizarea tehnicii top-down. în acest fel circuitul este descompus în module.

Modulele care alcătuiesc proiectul pot fi realizate utilizând fie un limbaj de programare HDL, fie folosind utilitarul *Schematic,* în cele ce urmează se vor prezenta toate procesele implicate în proiectarea, sinteza şi implementarea unui circuit digital.

Procesul Design Entry Utilities este cel care permite definirea modulelor care compun proiectul. Acest proces presupune existenta următoarelor subprocese:

- *Create Schematic Symbol* - creează simbolul unui modul deja definit. Acest simbol (un exemplu de simbol este prezentat în figura A.2) nu poate fi utilizat în ISE Schematic Editor. Acest subproces este util în cazul în care se doreşte vizualizarea porturilor unui modul proiectat. Dacă se va realizează un dublu-click pe acest simbol, atunci în secțiunea ***Editare/Vizualizare cod sursă VHDL/Verilog*** vom putea vizualiza codul sursă care a generat modulul, în cazul în care proiectul este de tip mixt (include module VHDL, Verilog şi schematic), atunci schema generală a circuitului va conține mai multe asemenea simboluri interconectate între ele.

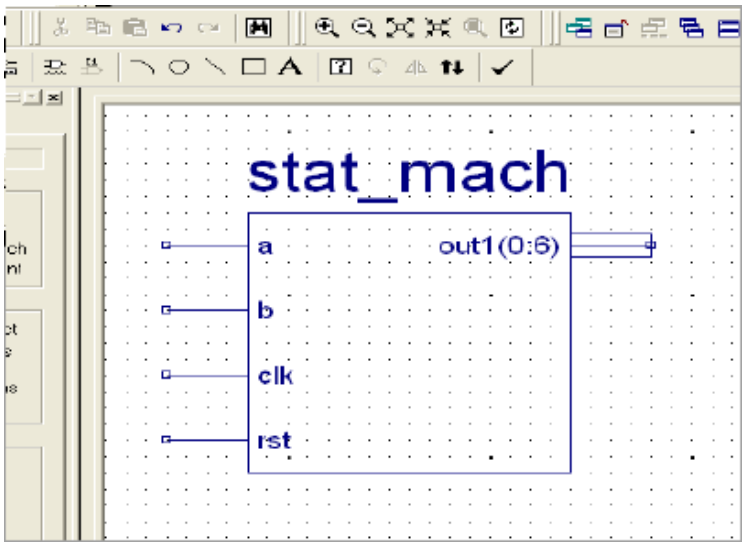


Figura A.2: Exemplu simbol generat de ISE.

2. *Launch ModelSim Simulator.* Acest subproces este necesar atunci când se doreşte testarea funcţionării unui modul. ISE nu beneficiează de un simulator propriu, dar se recomandă folosirea simulatorului produs de compania ModelTech şi denumit ModelSim care se integrează perfect cu ISE.

3. *View Command Line Log File.* Aşa cum se poate observa şi în figura A.3 acest subproces este folosit pentru vizualizarea sintaxei fiecărui proces lansat de ISE. Comenzile vizualizate sunt comenzi specifice Xilinx. O facilitate deosebită constă în faptul că aceste comenzi pot fi copiate şi editate într-un fişier script, care poate fi rulat.

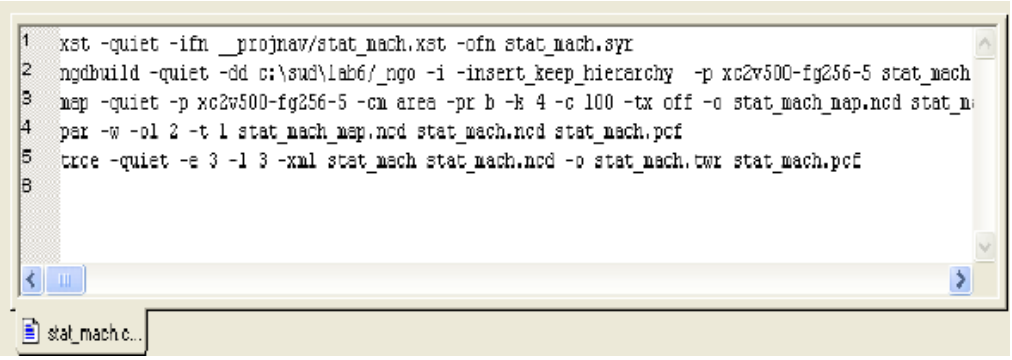


Figura A.3: Vizualizarea sintaxei proceselor ISE

4. *View Verilog Instantiation Template.* Acest subproces este utilizat pentru a crea rapid declaraţii de componente şi instanţieri de templet-uri, utilizând un limbaj HDL. Acestă facilitate este foarte utilă deoarece templet-urile definite pot fi copiate în proiecte ierarhice pentru utilizări viitoare.

Procesul User Constraints este următorul proces invocat. Acest proces precum şi toate subprocesele sale folosesc ca intrare un fişier a cărui extensie este *ucf* (User Constraints File). Acest fişier este editat de către proiectant ceea ce înseamnă că ori de câte ori se modifică acest fişier, întregul proiect se va recompila.

Pentru a crea acest fişier se selectează meniurile: File, New Source, Implementation Constraints Editor. Acum trebuie să se selecteze modulul de vârf al proiectului deoarece fişierul de constrângeri este în directă corespondenţă cu acesta. În acest fişier, fiecărui port al modulului de top îi va corespunde un pin fizic în circuitul FPGA/CPLD, în care se va realiza implementarea.

Observaţie: Fiecărui proiect îi este ataşat numai un singur fişier de tip *ucf*.

Procesul *User Constraints* conține următoarele subprocese:

1. *Create Timing Constraints.* Acest subproces invocă editorul de constrângeri care este prezentat în figura A.4. Tab-ul *General* este utilizat pentru a putea vizualiza semnalele globale ale proiectului. Folosirea tab-ului *Ports* asignează fiecărui semnal global câte un port în corespondență cu specificațiile circuitului FPGA/CPLD utilizat.

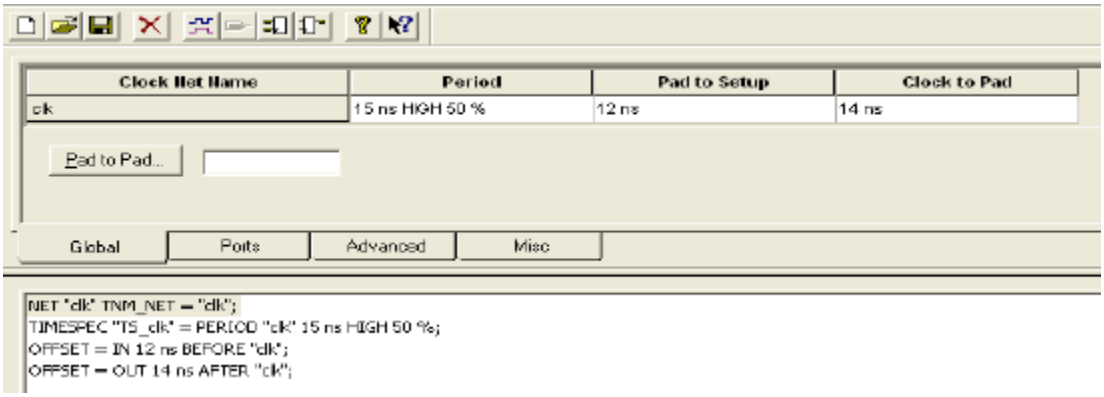


Figura A.4: Editorul ISE pentru crearea constrângerilor de timp.

2. *Assign Package Pines.* Este un subproces nou denumit şi **PACE** . Cu ajutorul acestui subproces se pot vizualiza, aşa cum se poate observa în figura A.5, pinii I/O ai circuitului FPGA/CPLD în care se va face implementarea. Fiecare tip de pin este codificat cu o anumită culoare (aşa cum se poate observa în figura A.6) sau cu un anumit simbol în modul de vizualizare pachet. ***PACE*** este recomandat pentru introducerea pinilor şi constrângerilor legate de arie.

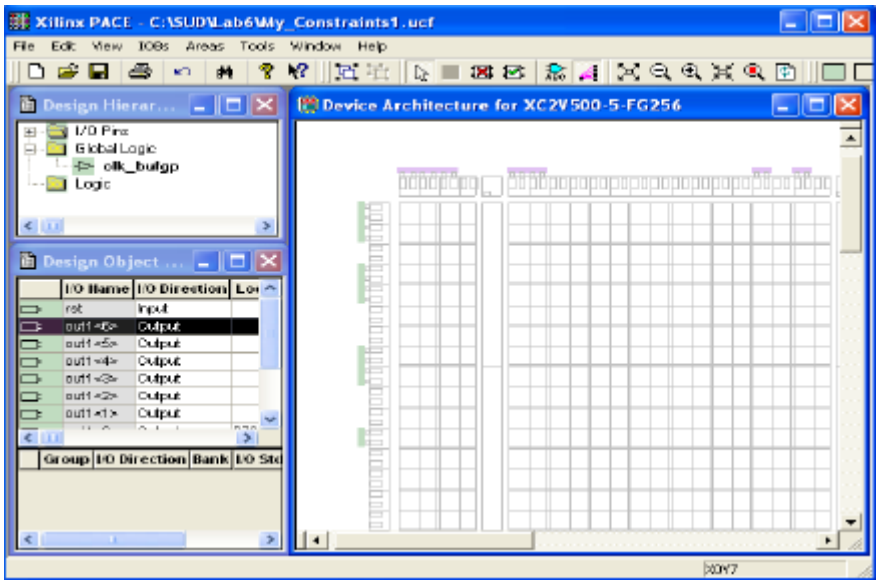


Figura A.5: Xilinx ***PACE .***

În fişierul cod sursă (modulul de vârf) trebuie conectate intrările şi ieşirile. În caz contrar se va obţine o implementare care nu are nici o funcţionalitate, ceea ce implică faptul că fişierul rezultat în urma procesului ***ngdbuild*** va fi vid.

ngdbuild este un proces care identifică modulul de vârf şi porturile sale cu pinii circuitului FPGA/CPLD.

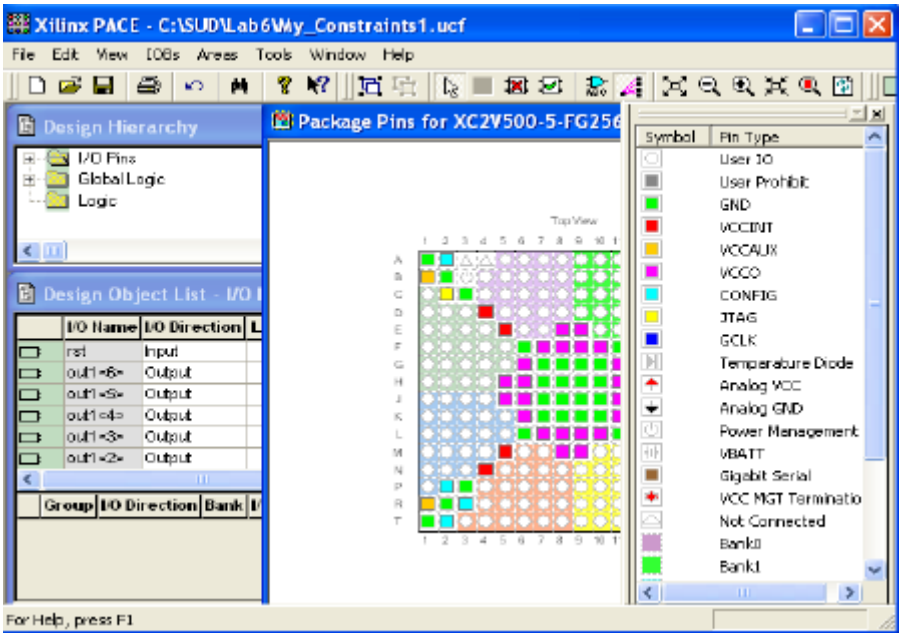


Figura A.6: Codificare ***PACE*** bazată pe culori.

Subprocesul ***PACE*** este utilizat pentru asignarea pinilor la porturile modulului de vârf, într-o etapă introductivă din cadrul proiectării circuitului FPGA/CPLD (aşa cum se poate observa în figura A.7). De asemenea, acest subproces oferă posibilitatea de a seta tensiunile care vor fi aplicate porturilor de I/O. Utilizarea acestui subproces trebuie făcută cu atenţie deoarece toate mofificările efectuate sunt suprascrise în fişierul ***ucf***.

3. *Create Area Constraints*. Este o facilitate nouă oferită de utilitarul ***Xilinx Floorplan***. Fiecare submodul din cadrul proiectului poate avea propriile constângeri legate de dimensiunea ariei ocupate. Aria ocupată de fiecare modul este afişată cu o culoare separată, aşa cum se poate observa în figura

A.8, pe aria totală disponibilă a circuitului FPGA/CPLD

4. *Edit Constraints File.* Acest subproces este folosit în momentul în care se doreşte o editare directă a fişierului de constrângeri. Conținutul unui fişier de constrângeri este prezentat în figura A.9. In fereastra deschisă de subproces, se pot edita, şterge sau adăuga constrângerile care se aplică circuitului final.

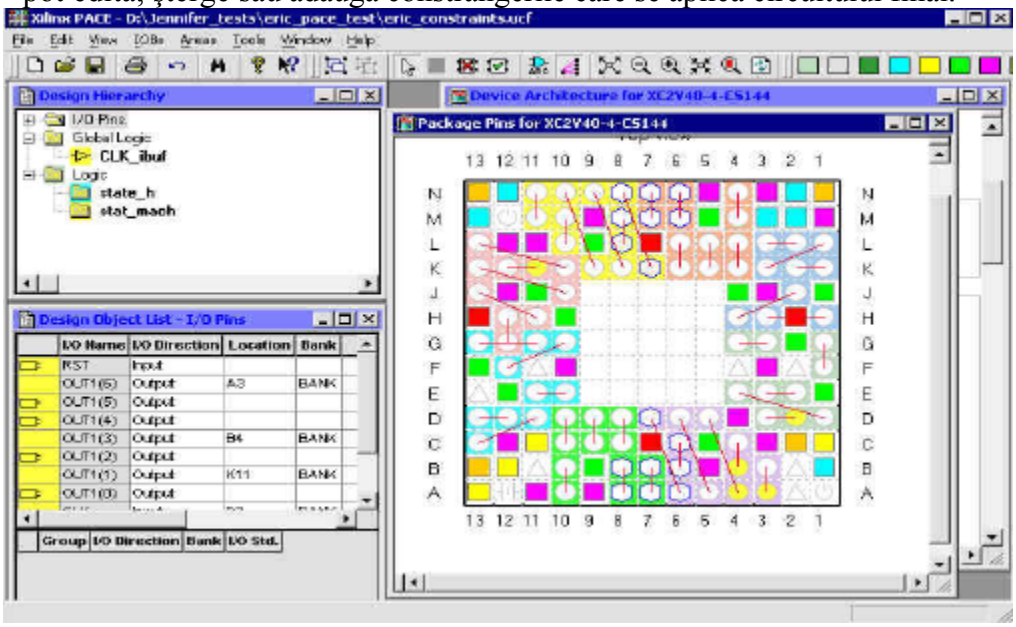


Figura A.7: Asignarea pinilor I/O folosind ***PACE***.

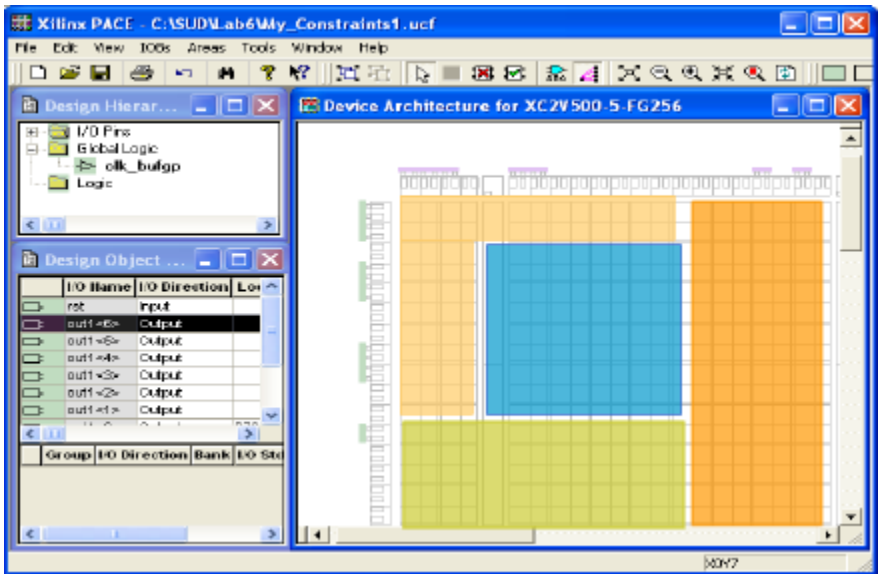


Figura A.8: Aria ocupată de fiecare submodul din cadrul proiectului este afişată folosind o culoare distinctă.

Metoda cea mai simplă, este aceea de a utiliza facilitatea de generare automată a fişierului de constrângeri (se va utiliza opţiunea BUILT-IN). Odată ce modelul care ne interesează a fost ales, copiem codul corespunzător modelului într-un fişier nou, a cărui extensie trebuie să fie *ucf* .

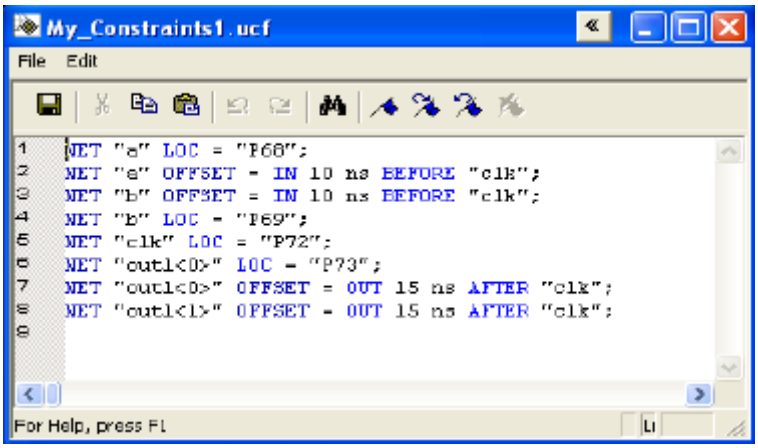


Figura A.9: Exemplu de fişier de constrângeri.

Procesul Synthesize este următorul proces invocat. Sinteza unor circuite complexe (de exemplu sinteza unui circuit pentru implementarea operaţiilor aritmetice în virgulă mobilă) poate dura ore iar spaţiul liber necesar pe harddisk este de ordinul zecilor de GB.

Rularea acestui proces implică existenţa a cel puţin unuia din următoarele utilitare: *Xilinx XST Compile, Synplicity sau Leonardo Spectrum.* Xilinx XST Compile este un utilitar proprietar Xilinx şi dedicat mediilor de testare hardware care conţin doar circuite FPGA/CPLD Xilinx. Celelalte două utilitare realizează sinteza si pentru alte tipuri de circuite FPGA/CPLD, spre exemplu circuitele ALTERA.

Acest proces conţine un număr de patru subprocese. Trebuie remarcat faptul că toate subprocesele se execută într-o ordine prestabilită. Rularea unui subproces specificat implică execuţia în ordine a tuturor subproceselor care preced subprocesul ales.

Subprocesele procesului *Synthesize* sunt următoarele:

1. *View Synthesis Report.* Acest subproces oferă o imagine detaliată despre rezultatele obţinute de toate subutilitarele apelate de utilitarul de sinteză. În raportul generat, aşa cum se poate observa în figura A. 10, se regăsesc informaţii privitoare la aria utilizată de circuit, opţiunile selectate precum şi rezultatele

estimărilor legate de stabilirea căii de întârziere. Informațiile referitoare la aria utilizată de către circuit sunt exprimate în unități de măsură interne Xilinx.

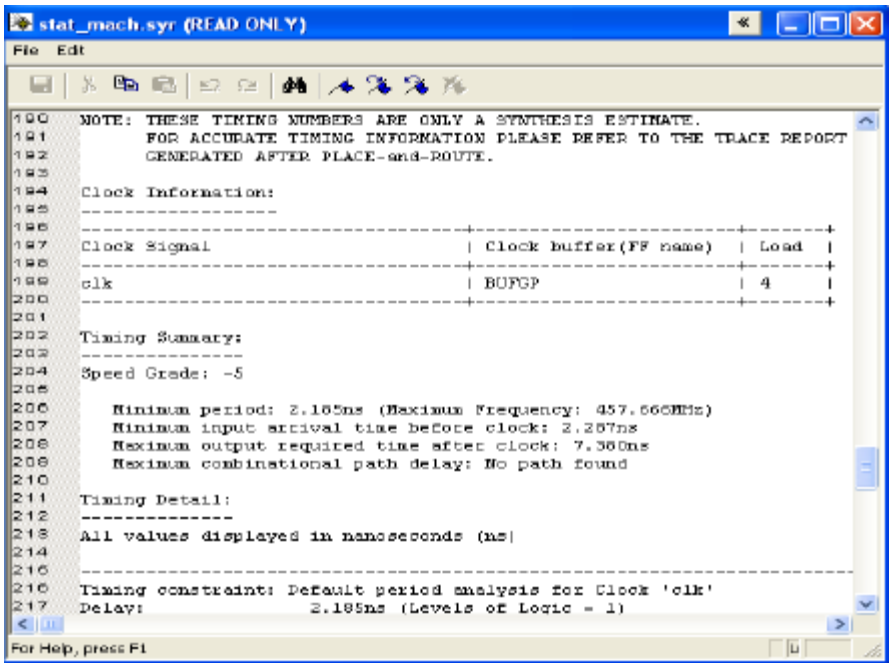


Figura A.10: Raport de sinteză generat de ISE.

2. *View RTL Schematic.* Acest subproces oferă posibilitatea de a vizualiza codul sursă Verilog/VHDL sub forma unei scheme. Un exemplu este prezentat în figura A. 11. Acest subproces are o utilitate deosebită deoarece el oferă posibilitatea unei verificări logice si ierarhice rapide a întregului circuit proiectat.

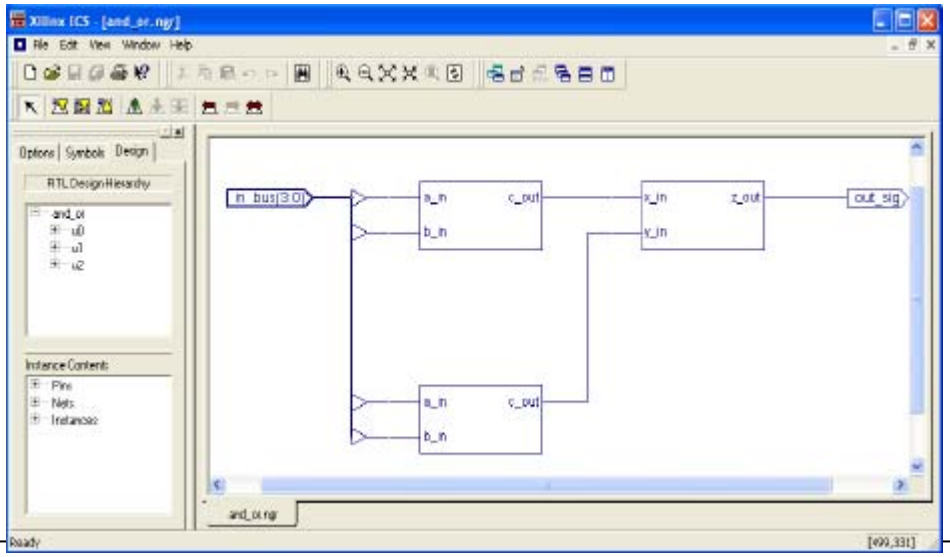


Figura A.11: Schemă generată de ISE pornind de la cod sursă Verilog/VHDL.

3. *Analyze Hierarchy.* Acest subproces este responsabil cu elaborarea părții RTL a proiectului evidențiind relațiile dintre module şi specificațiile între ierarhiile de module unde modulele sunt instanțiate. Tot în cazul acestui subproces se reliefează şi biblioteca specifică care este utilizată în proiectarea circuitului. Un exemplu de astfel de raport este prezentat în figura A.12.

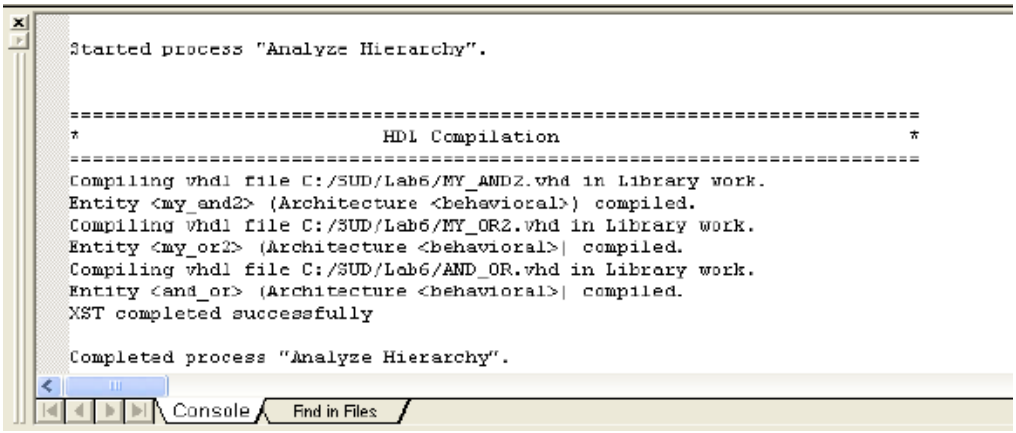


Figura A.12: Raport cu privire la relațiile dintre module.

4. *Check Syntax.* După cum sugerează chiar numele, acest subproces este utilizat pentru vizualizarea erorilor de sintaxă. La compilarea fişierelor sursă pot apărea erori care pot fi vizualizate în secțiunea ***TRANSCRIPT*** . După cum se poate observa şi in figura A.13, erorile sunt marcate cu ajutorul unui pătrat. Se selectează cu ajutorul mouse-ului eroarea ce se doreşte a fi remediată şi automat în secțiunea ***Editare/Vizualizare cod sursă VHDL/Verilog*** este deschis fişierul sursă corespunzător erorii. Această facilitate este deosebit de utilă în cazul proiectelor care conțin multe fişiere sursă.

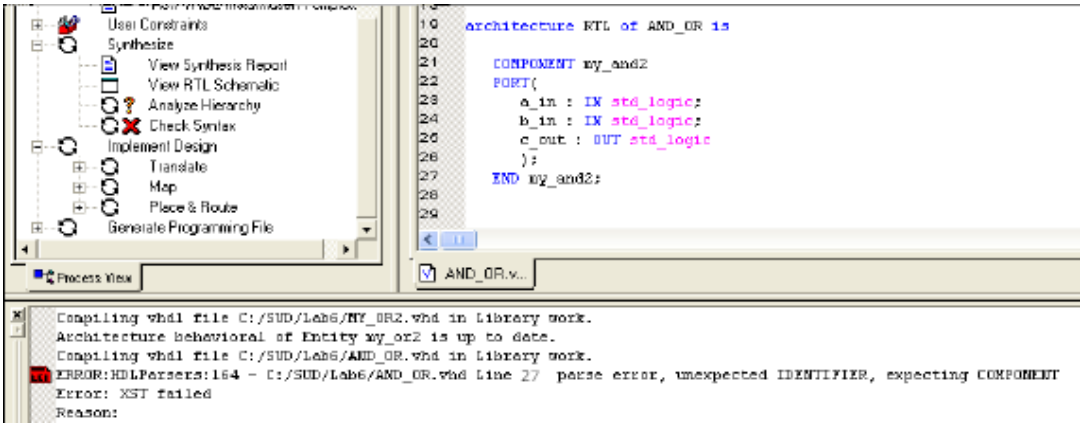


Figura A.13: Vizualizarea erorilor conținute în fişiere sursă HDL.

Procesul Implement Design. La finalul acestui proces întregul circuit proiectat se va regăsi într-un fişier special cu extensia ***bit***. Fişierul care conţine prototipul circuitului implementat va fi mutat în circuitul Xilinx FPGA/CPLD disponibil în vederea testării funcţionării sale reale.

Acest proces conţine trei subprocese importante: *Translate, Map* şi *Place Route*. Fiecare subproces conţine unul sau mai multe subsubprocese, care vor fi detaliate. O remarcă importantă este aceea că, ordinea acestor subprocese este prestabilită şi selectarea rulării unui subproces aleator implică rularea tuturor subproceselor anterioare lui.

În cazul în care implementarea se va face într-un circuit Xilinx CPLD vom avea doar două subprocese: *Translate* şi *Fit*. În concluzie procesul de implementare a unui circuit într-un circuit Xilinx CPLD este uşor diferit faţă de procesul de implementare a unui circuit într-un circuit Xilinx FPGA.

Procesul *Implement Design* dispune de un număr mare de proprietăţi care pot fi modificate în funcţie de cerinţele fiecărui proiect (aşa cum se poate observa în figura A. 14). Pentru a putea avea acces la aceste proprietăţi în vederea modificării lor, este suficient să se selecteze procesul *Implement Design* şi apoi se acţionează butonul din dreapta al mouse-ului.

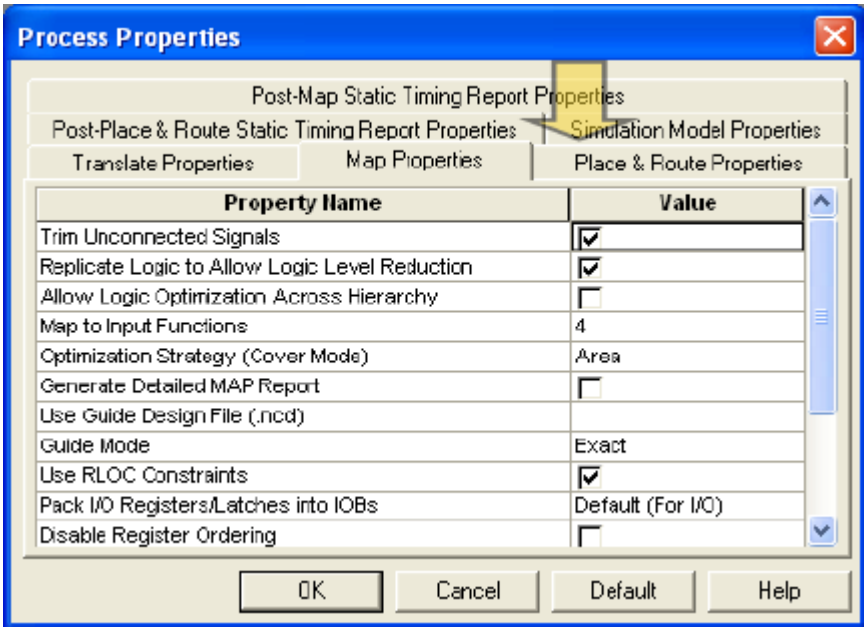


Figura A.14: Vizualizarea şi editarea proprietăţilor procesului *Implement Design*.

1. *Translate,* în ISE translatarea unui proiect se efectuează prin intermediul utilitarului *ngdbuild.* Acest utilitar realizează combinarea tuturor modulelor existente în cadrul unui proiect. Una din erorile frecvente este imposibilitatea localizării fişierului de tip *netlist* al unei componente de către procesul de translatare. Acest fişier de tip *netlist* trebuie generat în mod obligatoriu pentru fiecare modul component al proiectului, după etapa de proiectare a modulului.

   Acest subproces conține următoarele subsubprocese:

   - *Translation Report* - în acest raport sunt cuprinse informații referitoare la modulele care vor fi combinate. Alte informații care se regăsesc în raport sunt cele referitoare la ieşirile obținute în urma rulării utilitarului ***ngdbuild*** **.**

   - *Floorplan Design* - acest proces ne oferă posibilitatea vizualizării modului în care logica proiectată ocupă resursele disponibile din cadrul circuitului Xilinx FPGA. Aşa cum se poate observa şi în figura A.15 se pot vizualiza şi porturile de intrare/ieşire.

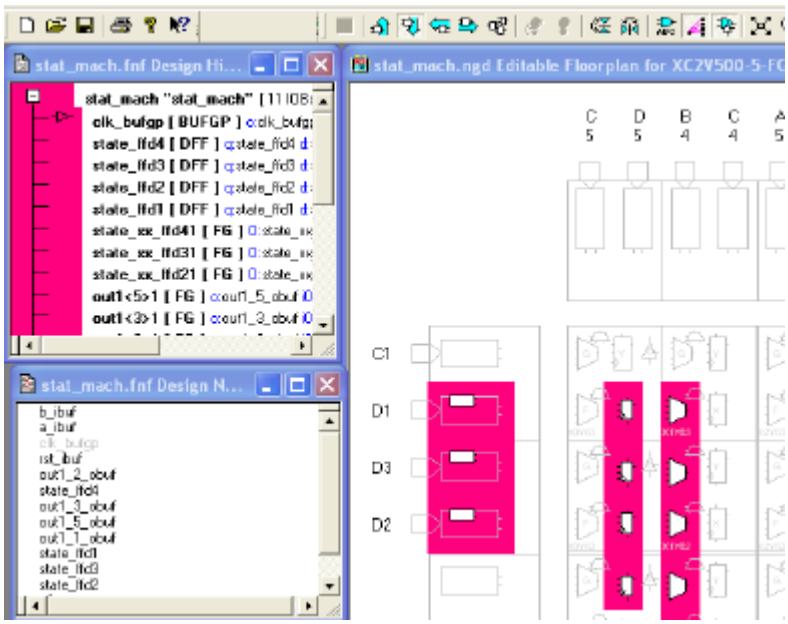


Figura A.15: Vizualizarea resurselor ocupate.

Tot cu ajutorul acestui proces, putem realiza constrângeri de tip arie prin desemnarea unor arii specifice pentru implementarea proiectului. Toate aceste facilități nu sunt oferite de către nici un alt proces.

- Generate Post-Translate Simulation Model - acest proces oferă un model în limbaj HDL rezultat în urma procesului de traslatare, aşa cum se poate observa în figura A. 16.

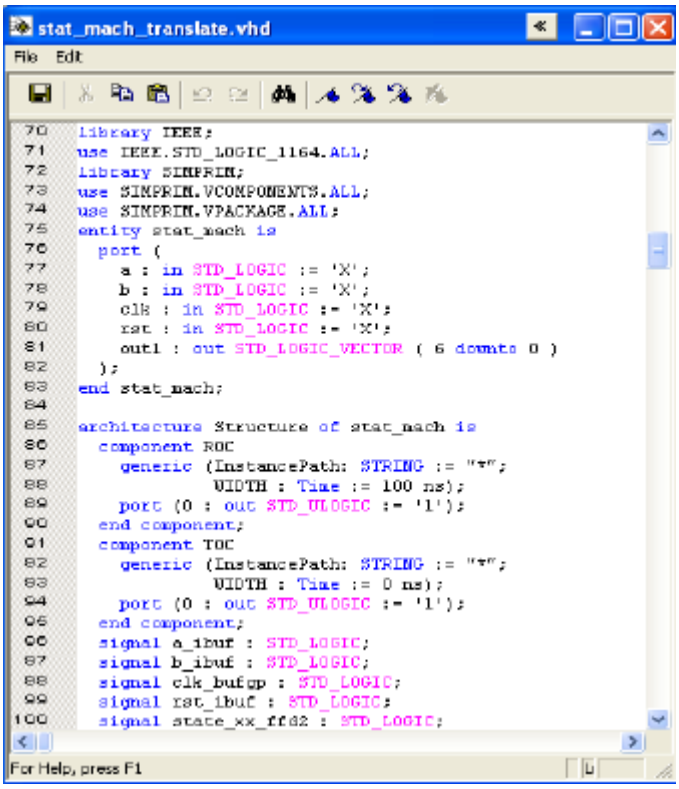


Figura A.16: Model rezultat în urma procesului de translatare

Deoarece procesul *Place Route* nu a fost rulat, nici o informaţie referitoare la întârzierile prin circuit nu este disponibilă. Rezultatul acestui proces este utilizat în vederea realizării unei verificări post-sinteză cu ajutorul fişierul ***netlist*** pentru proiectare. Acest fişier oferă o implementare structurală modulului VHDL/Verilog deoarece în componenţa sa se regăsesc numai semnale, declaraţii de componente si instanţieri de componente.

3. *Map.* Acest subproces conţine următoarele procese:

- *Map Report* - rezultatul acestui proces este un raport care include toate informaţiile legate de maparea circuitului aşa cum se poate oberva în figura A.17. În raport se regăsesc informaţii legate de înlăturarea blocurilor logice

redundante. Cele mai multe optimizări sunt realizate în timpul procesului de sinteză.

- Tot în acest raport se mai găsesc şi informaţii referitoare la plasarea relaţională a macro-urilor proiectului. Foarte important este faptul că acest raport cuprinde informaţii referitoare la aria totală ocupată de către proiect. Aria este exprimată în unităţi interne Xilinx. O listă completă a numărului de componente Xilinx utilizate (IOB, LUT etc.) poate fi consultată tot în cadrul acestui raport.

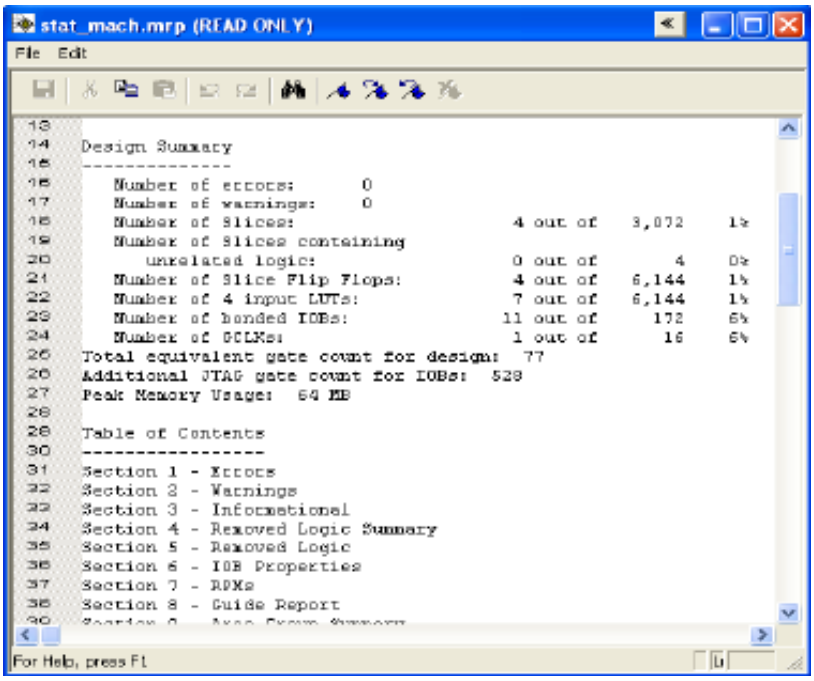


Figura A.17: Raport complet asupra mapării proiectului.

- Generate Post-Map Static Timing Report - acest proces conţine următoarele sub-procese:

  - *Post-Map Static Timing Report* - acest raport este prezentat în format ***xml*** (aşa cum se poate observa în figura A.18) pentru a facilita navigarea prin document. El este esenţial pentru o analiză în vederea realizării unei hărţi de preplasare a circuitului în structura ariei - Xilinx FPGA. Această preplasare se realizează în funcţie de întârzierile determinate.

Pe baza acestui raport se poate verifica dacă constrângerile impuse circuitului sunt realiste sau se înscriu într-o anumită tabelă care trebuie respectată. În raport se poate observă valoarea fiecărei constrângeri obținută în cazul utilizării celei mai defavorabile căi din punct de vedere al întârzierilor. Tot în acest raport se poate observa dacă o anumită constrângere poate fi sau nu îndeplinită.

- *Text-based Post-Map Static Timing Report* - acest proces oferă acelaşi raport prezentat anterior cu deosebirea că formatul fişierului este ***txt*** în loc de ***xml*** Acest proces a fost introdus pentru a se putea tipări raportul obținut într-o formă grafică facilă.

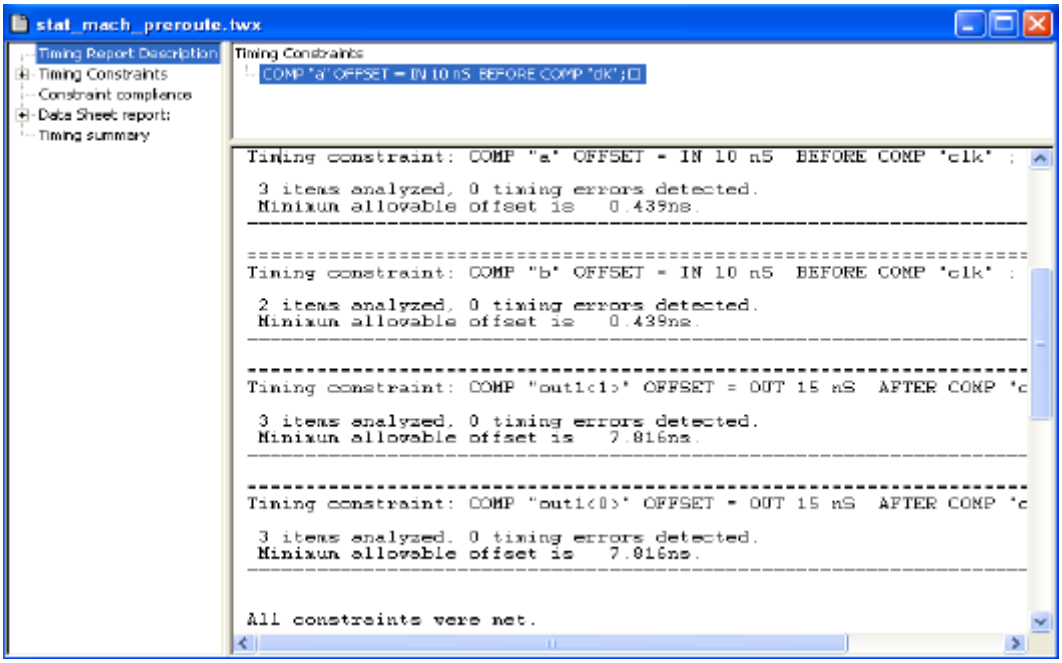


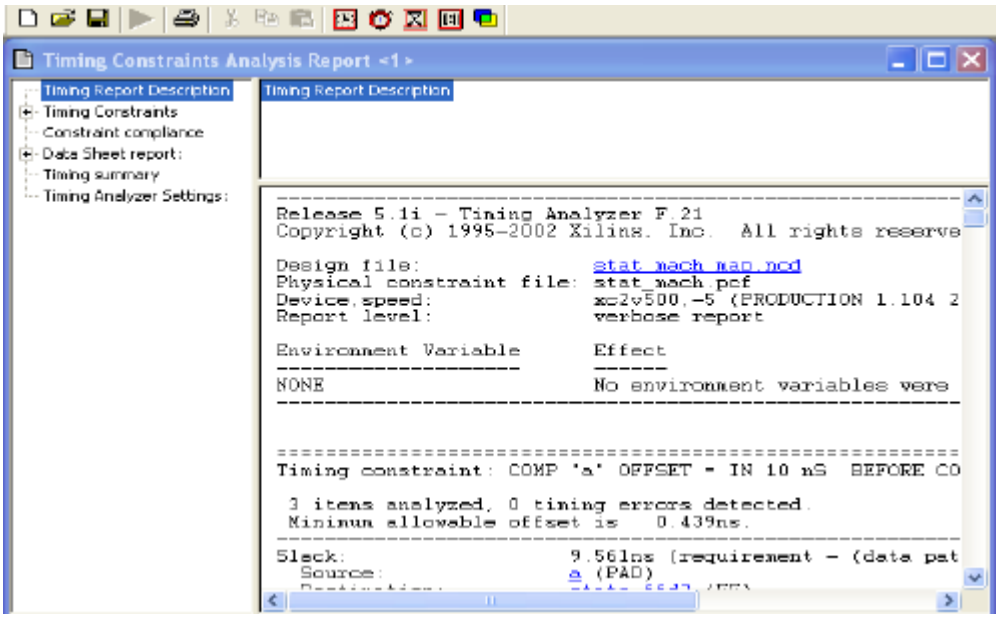
Figura A.18: Raport complet în versiune xml.

- *Text-based Post-Map Static Timing Report* - acest proces oferă acelaşi raport prezentat anterior cu deosebirea că formatul fişierului este ***txt*** în loc de ***xml*** Acest proces a fost introdus pentru a se putea tipări raportul obținut într-o formă grafică facilă.

- *Analyze Post-Map Static Timing* - În urma rulării acestui proces putem analiza harta finală a implementării circuitului din punct de vedere al timpilor statici. Xilinx oferă metode noi de analiză a timpilor, aceste metode putând fi uşor apelabile (butoanele încadrate într-un dreptunghi în figura A.19).

  Se pot obține rapoarte diferite dacă se selectează de fiecare dată alte semnale sau dacă se modifică diferitele opțiuni disponibile.

- Manually Place Route (FPGA Editor) - acest proces este utilizat dacă se doreşte realizarea procesului *Place Route* manual. Pentru aceasta, ISE oferă un utilitar numit *FPGA Editor,* aşa cum se poate observa în figura A.20. Acest utilitar oferă acces la nivel logic pentru circuitul Xilinx FPGA selectat



precum şi la resursele de rutare.

Figura A.19: Analiza timpilor statici.

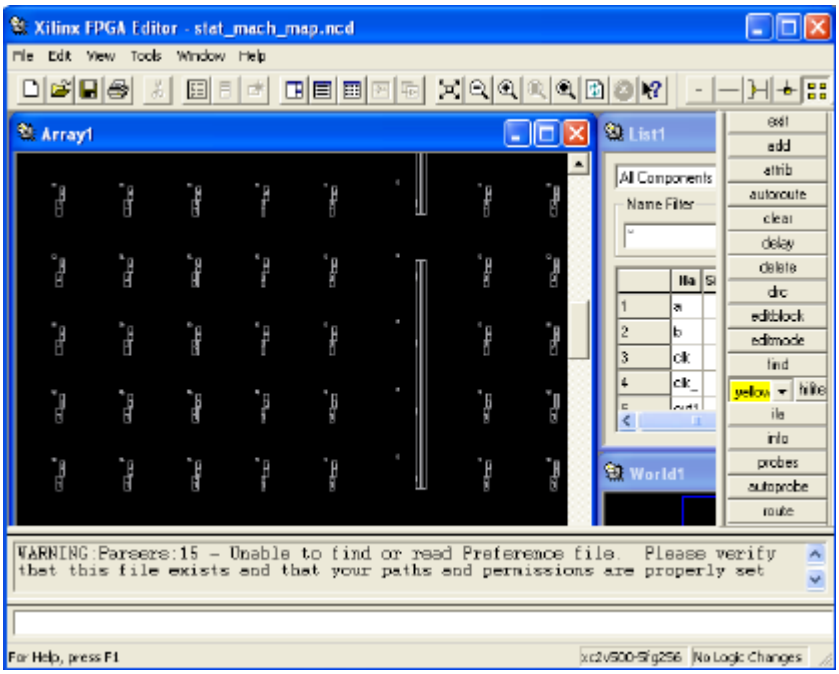


Figura A.20: FPGA Editor

Chiar dacă acest utilitar este perfect funcțional, utilizarea sa este recomandată doar pentru inginerii cu vaste cunoştinţe VLSI. În mod normal, rularea procesului *Place Route* în modul automat este recomandată deoarece ieşirea obținută este optimizată.

Modificările făcute cu ajutorul acestui utilitar afectează funcționarea circuitului. Modul în care funcționarea circuitului este afectată nu este reflectat nici în codul sursă HDL nici în rezultatele obținute în urma simulării circuitului.

- *Generate Post-Map Simulation Model* - în momentul lansării în execuție a acestui proces, proiectul nu a trecut de etapa *Place Route.* Conform acestei observații, nu se poate realiza decât o simulare de timp parțială bazată pe întârzierile blocurilor actuale precum şi pe baza întârzierilor de rutare estimate.

  Ieşirile obținute în urma acestui proces sunt salvate în două fişiere: un fişier cu extensia ***vhd*** în care se regăseşte o reprezentare structurală a implementării si al doilea fişier cu extensia ***sdf*** (Standard Delay Format) care conține informații legate de întârzierile prin circuit.

  Ambele fişiere pot fi deschise într-un simulator HDL (ModelSim spre exemplu) pentru a putea realiza o verificare a logicii proiectării pentru harta obținută într-un proces anterior.

3. *Place Route* - Acest proces ca şi procesul *Map* posedă un număr mare de parametri care pot fi modificați în funcție de necesitățile proiectului. Modalitatea de accesare şi modificare este similară cu cea descsrisă în cadrul procesului *Map.* Din cadrul acestui proces fac parte următoarele subprocese:

   - *Place Route Report* - acest raport conține informații importante referitoare la proiect, după executarea procesului de implementare. Informații privitoare la semnalul de ceas sau eventualele anomalii ale acestui semnal pot fi regăsite în acest raport, aşa cum se poate observa în figura A.21. Tot aici apare şi scorul de proiectare.

     Scorul de proiectare este influențat de mulți factori. Totuşi un scor de proiectare mic indică în general o implementare bună. Pe lângă aceste informații, raportul mai conține informații privitoare la media întârzierilor

prin circuit, întârzierea maximă a pinului precum şi media întârzierilor semnalelor în cazul utilizării celei mai defavorabile rute.

- *Asynchronous Delay Report* - acest raport se concentrează asupra celor mai proaste 20 de căi din implementare şi arată întârzierile obținute pentru fiecare semnal. Acest raport este foarte util în cazul în care apar probleme legate de aria implementării.

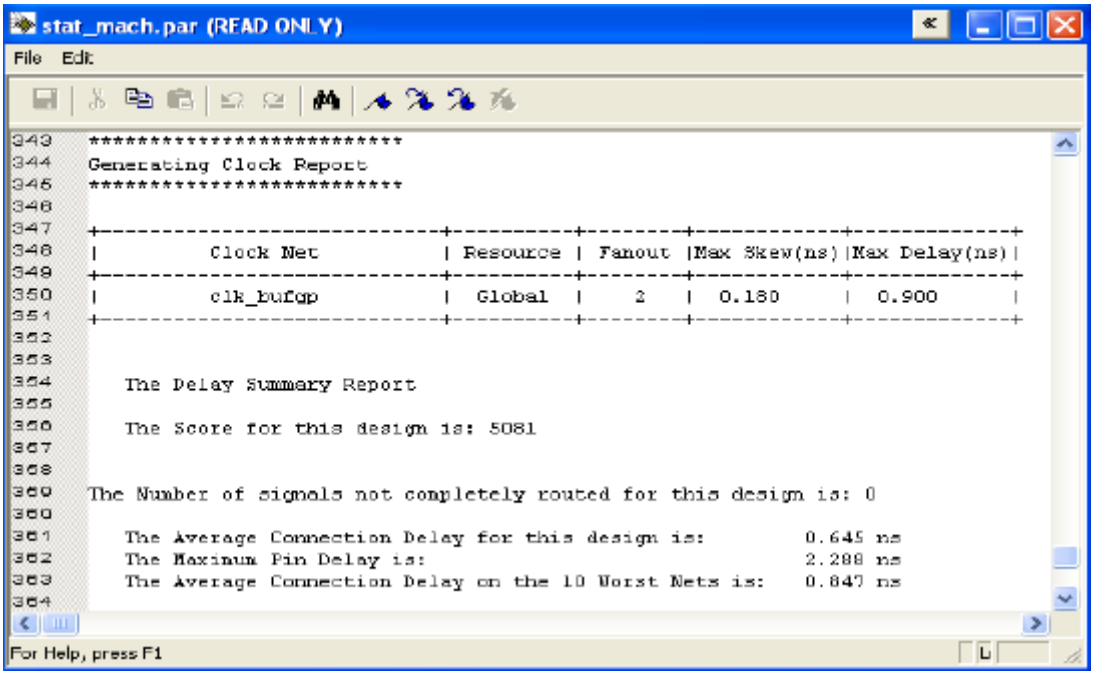


Figura A.21: Raport obținut în urma procesului de *Place Route* .

- *Pad Report* - este cel mai cuprinzător raport şi se referă la căile prin care se propagă semnalele prin circuit precum şi la pinii pachetelor A.22.

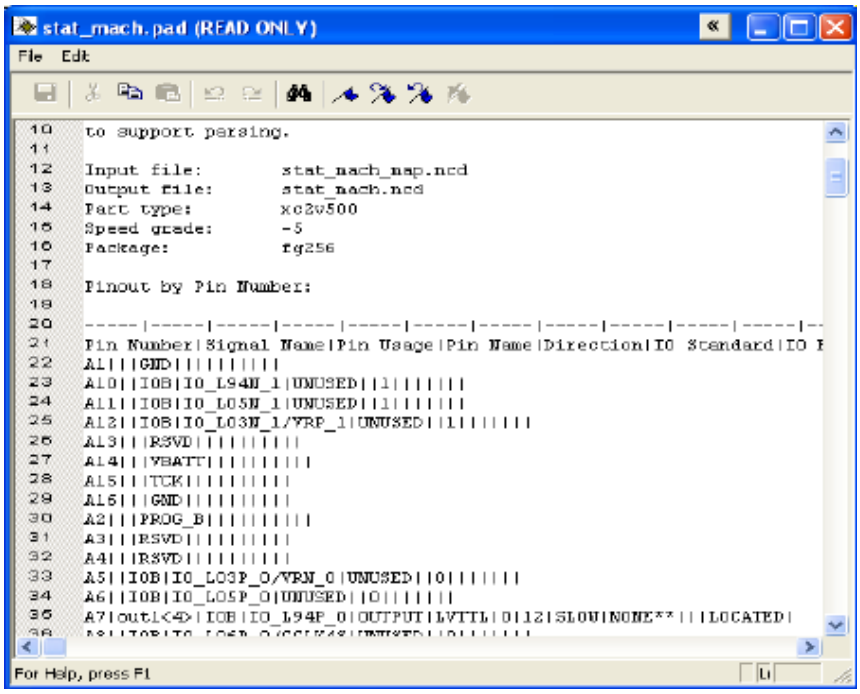


Figura A.22: Raport cu privire la căile de propagare a semnalelor prin circuit.

Acest raport este uşor de importat în Excel pentru a se putea realiza o analiză mai atentă şi pentru o eventuală listare a informațiilor conținute în raport.

- *Guide Results Reports* - acest raport face posibilă urmărirea rezultatelor obținute în urma rulării procesului *Place Route,* aşa cum se poate observa şi în figura A.23. Scopul urmărit este acela de a menține implementarea anterioară cât mai mult posibil. Avantajul acestei abordări îl constituie păstrarea nemodificată a layout-ului implementării. Consecința directă a acestui fapt este că timpul de funcționare al circuitului rămâne nemodificat.

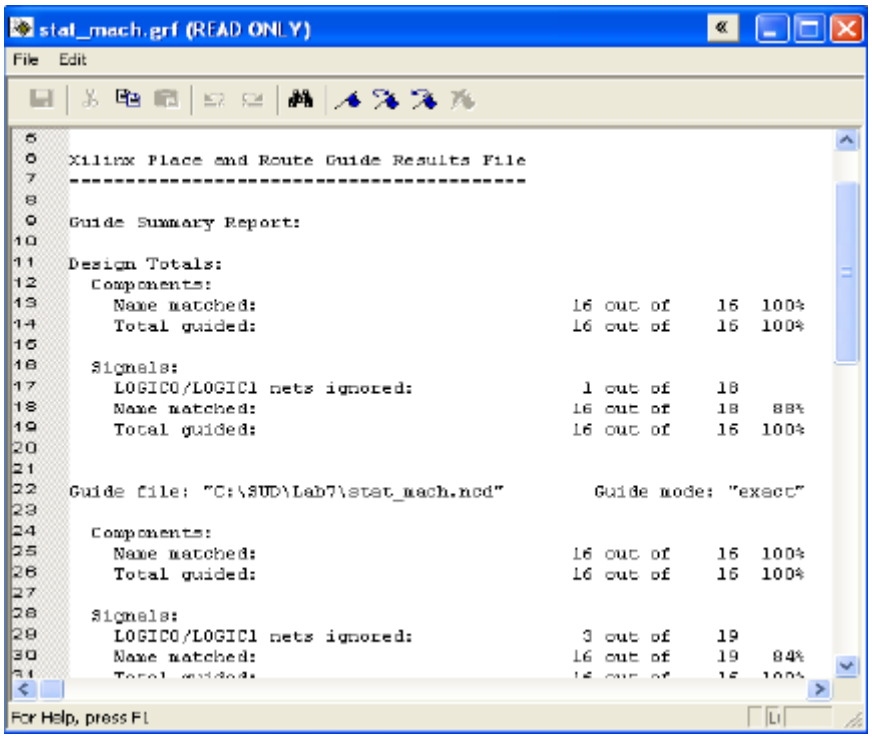


Figura A.23: Raport cu privire la gradul de optimizare.

Din analiza acestui raport se poate deduce gradul de optimizare (realizat de către programele software Xilinx) obținut între două implementări succesive. Obținerea unui raport mai complet implică activarea opțiunii *Guide Results Reports.* Este una dintre proprietățile proceselor *Translate* şi *Map.*

- *Generate Post-Place Route Static Timing* - acest proces împreună cu cele trei sub-procese ale sale ne oferă valorile timpilor fiecărei constrângeri aplicabile circuitului. Aceste valori sunt obținute în urma finalizării procesului *Place Route* şi indică cu exactitate funcționarea circuitului.

- *Post Place Route Static Time* - acest proces oferă un raport în care se poate observa o trecere în revistă a fiecărei constrângeri (figura A.24).

Calea cea mai defavorabilă este arătată automat numai dacă acea cale nu reuşeşte să respecte cerinţele impuse de constrângere. Raportul este prezentat în format *xml* pentru a facilita consultarea rezultatelor obţinute. Selectând hiperlink-urile pentru semnale sau componente conţinute în raport se poate invoca utilitarul *Xilinx Floorplane.*

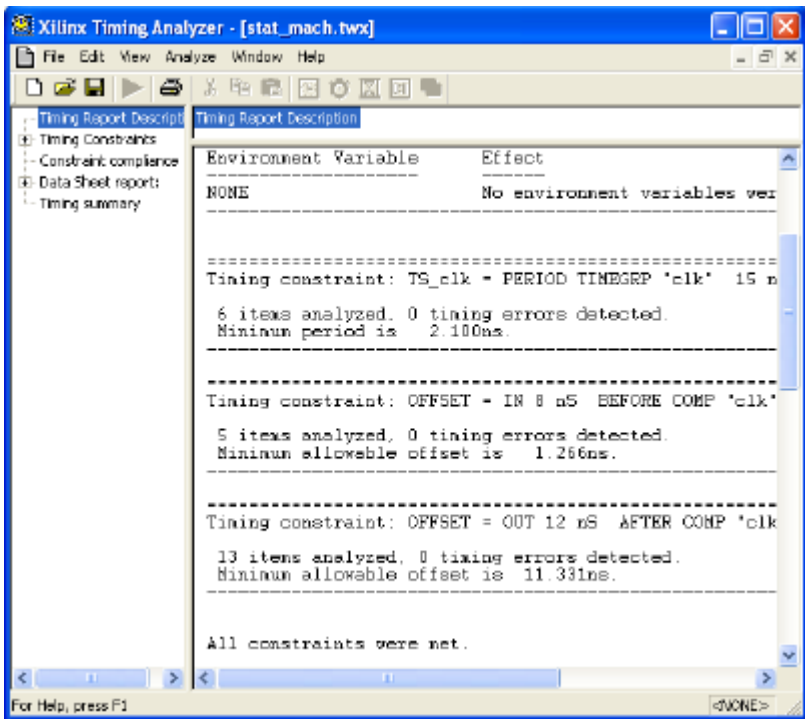


Figura A.24: Raport asupra constrângerilor impuse circuitului.

- *Text-base Post Place Route Static Time* - acest raport este identic cu raportul obţinut de către procesul anterior cu deosebirea că formatul utilizat acum este *txt.* Acest raport poate fi uşor tipărit dacă este necesar.

- *Analyze Post-Place Route Static Timing* - acest utilitar ne oferă toate opţiunile şi capabilităţile utilitarului *Xilinx Timing Analyzer Tool.* Astfel se pot genera rapoarte, în diferite formate, selectând un grup particular de semnale sau chiar un subgrup de semnale ce se doresc a fi examinate.

View Edit Placed Design acest proces invocă utilitarul *Floor Planner* (figura A.25).

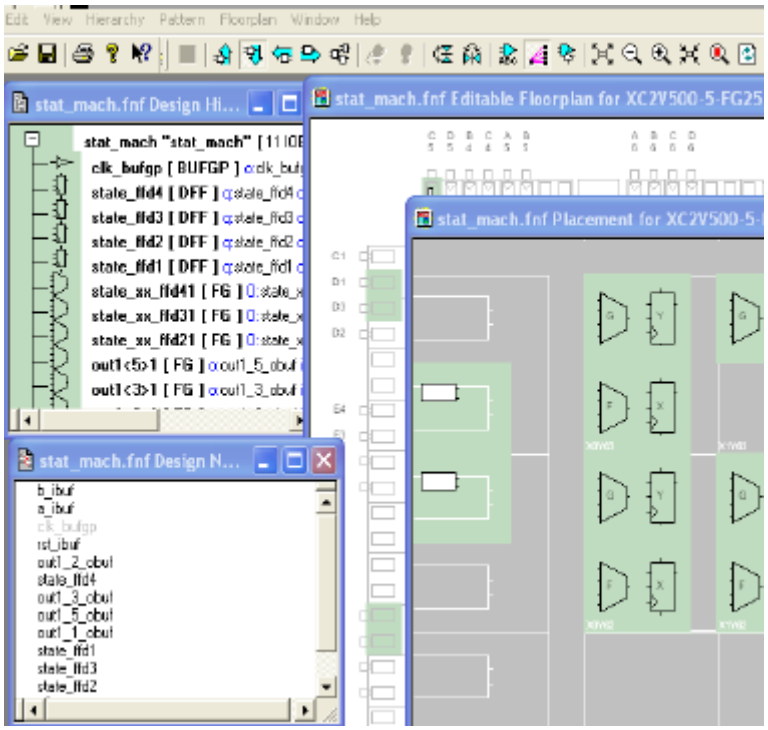


Figura A.25: Visualizare modului de plasare al circuitului în circuitul Xilinx FPGA/CPLD disponibil.

- *View Edit Routed Design (FPGA Editor)* - acest utilitar permite realizarea operațiilor de plasare şi rutare manuale folosind editorul FPGA oferit de Xilinx. Cu ajutorul lui este prezentată logica actuală a circuitului precum şi rutarea resurselor procesului Xilinx, aşa cum se poate observa în figura: A.26.

  Printre facilitățile oferite se pot menționa: realizarea unei operații de Place Route manual pentru orice/toată logica conținută de către circuit, modificarea layout-ului şi funcționalității circuitului, adăugarea de blocuri logice la logica deja existentă.

  Orice modificare realizată cu ajutorul acestui utilitar nu se va reflecta în codul RTL sau în rezultatele simulării comportamentale. Recomandarea este ca acest utilitar să se utilizeze cu atenție şi doar în următoarele cazuri:

- eliminarea erorilor din circuitului proiectat; se poate realiza o evaluare grafică a implementării;

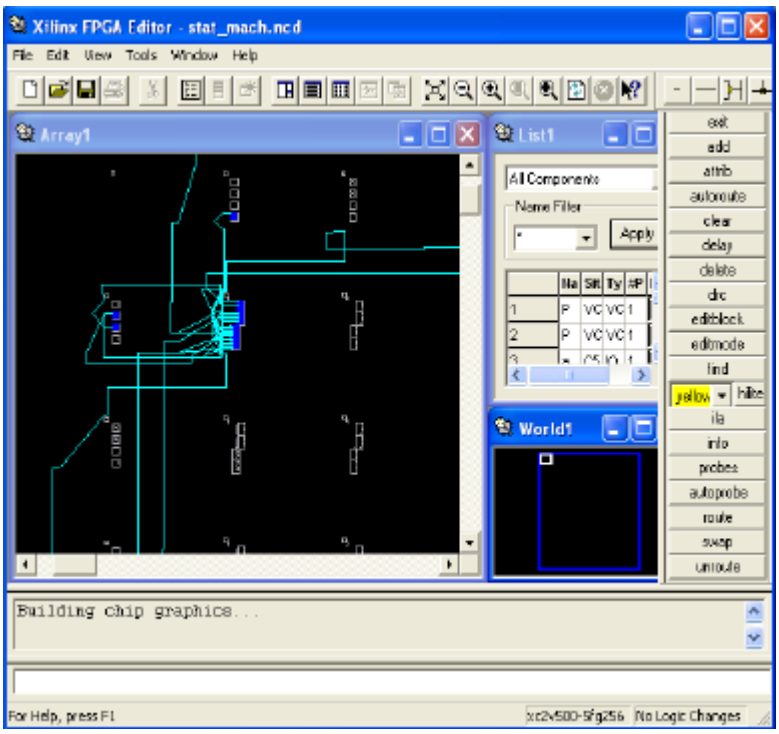


Figura A.26: FPGA Editor.

- eliminarea erorilor din circuitului proiectat; se poate realiza o evaluare grafică a implementării;

- evaluarea căilor constrângerilor; se pot modifica atributele pentru intrări/ieşiri fără a fi necesară o compilare a codului sursă;

- realizarea de modificări care totuşi nu modifică funcţionalitatea circuitului. În acest caz se doreşte doar modificarea implementării la nivel de circuite.

- modificarea atributelor pentru *Xilinx Device Resource* fără a modifica funcţionarea circuitului.

Analyze Power (XPower) - este un utilitar folosit pentru estimarea consumului de putere al unui circuit FPGA/CPLD. Un exemplu poate fi observat în figura A.27.Acest utilitar preia fişierele de implementare ale circuitului FPGA/CPLD precum şi datele de intrare ale proiectului pentru a produce o estimare foarte precisă pentru fiecare tensiune de alimentare.

XPower poate procesa fişiere rezultate în urma procesului de simulare, fişiere în format ***vcd*** . Acesta facilitate este foarte importantă deoarece detaliile de funcţionare reală ale circuitului proiectat sunt memorate în fişiere cu format ***vcd,*** aceste detalii fiind esenţiale în obţinerea unei precizii foarte ridicate a estimării puterii consumate.

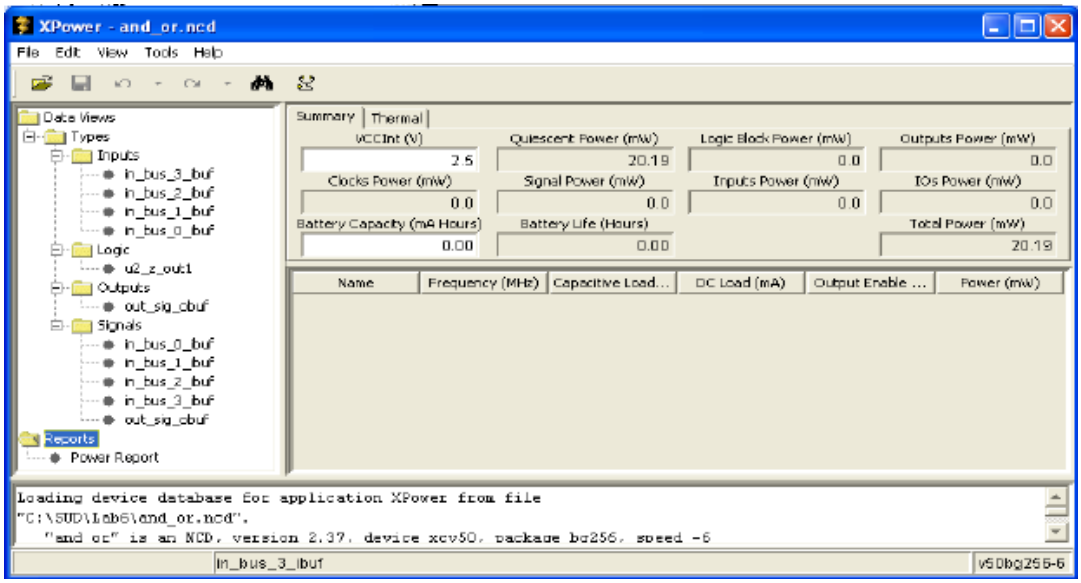


Figura A.27: Rezultate obţinute cu ajutorul utilitarului *XPower* .

- *Generate Post-Place Route Simulation Model* - invocarea acestui proces are drept scop obţinerea a două tipuri de fişiere: VHDL/Verilog (circuitul este descris la nivel structural) şi fişiere în format ***sdf*** (**S**tandard **D**elay **F**ormat). Fişierele obţinute pot fi simulate utilizând *ModelSim,* obţinând astfel simulări de timp pornind de la căile de întârziere pe care le posedă circuitul proiectat.

- *Generate IBIS Model (I/O Buffer Information Specification)* - Acest utilitar generează un model care permite simularea pe nivel precum şi analiza intrărilor/ieşirilor circuitului. Raportul astfel obţinut este personalizat pentru circuitul Xilinx FPGA/CPLD utilizat pentru implementarea circuitului.

- *Multi Pass Place Route* - este unul dintre cele mai importante utilitare oferite de Xilinx. Acest utilitar este folosit pentru a realiza multiple iteraţii ale utilitarului *Place Route.* Fiecare iteraţie implică utilizarea de intrări diferite pentru utilitarul *Place Route.* Fiecare intrare este aleasă aleator şi determinată doar de un număr particular ales de către proiectant. Valoarea acestui număr particular nu poate fi mai mare de 100, aşa cum se poate observa în figura A.28.

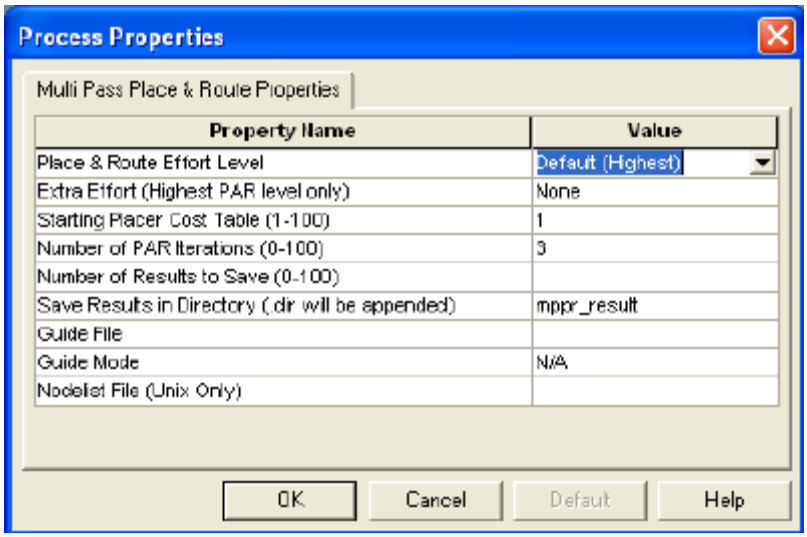


Figura A.28: Fereastra principală a utilitarului *Mulți Pass Place Route* .

Acest utilitar determină prin mai multe iterații aleatoare cea mai bună configurație pentru circuitul proiectat, urmând ca această configurație să fie cea finală. Acest utilitar posedă următorii parametrii care pot fi modificați în funcție de caracteristicile care trebuie îndeplinite de circuitul final:

- Starting Placer Cost Table (1-100) - se alege valoarea l, deoarece numărul iterației următoare este dat de numărul iterației curente la care se adaugă această valoare;

- Number of PAR iterations (0 - 100) - reprezintă numărul de iterații ales de către proiectant. Aşa cum s-a specificat mai sus, limita maximă a acestui număr este 100.

- Number of Results to Save (O - 100) - se alege o valoare diferită de zero dacă se doreşte un raport final cu rezultatele obținute în urma rulării acestui proces.

- Save Results in Directory - se alege directorul unde vor fi salvate rapoartele rezultate.

După terminarea rulării acestui proces, se poate selecta opțiunea *MFPR Raport* pentru a putea vizualiza şi analiza rezulatele obținute. Acest raport va conține informații referitoare la sinteza rezultatelor obținute pentru fiecare situație testată. Simbolul „*" este cel care ne indică exact configurata ce a fost salvată, aşa cum se poate observa în figura A.29.

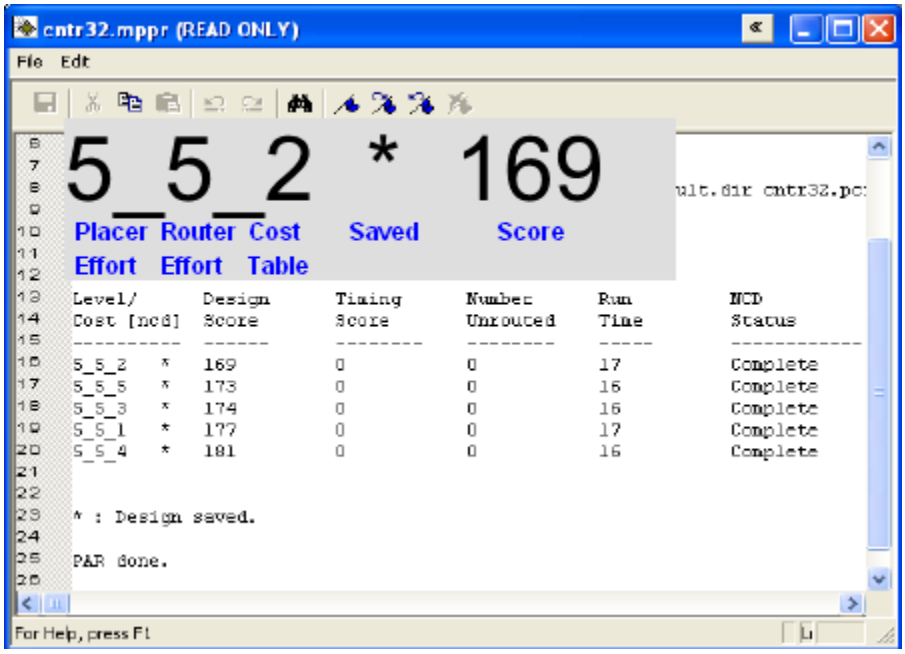


Figura A.29: MFPR Raport.

Back-annotate Pin Locations - Acest process permite verificarea porturilor de intrare/ieşire ale circuitului, în urma rulării acestui proces se vor obține două rapoarte. Primul raport se referă la denumirile pinilor circuitului. Cu ajutorul acestui raport putem verifica dacă există mai multe denumiri pentru acelaşi pin (această situație creează un conflict) sau dacă avem mai multe semnale asignate aceluiaşi port de intrare/ieşire al circuitului.

Raportul al doilea obținut ne permite să vizualizăm modalitatea de asignare a semnalelor de intrare/ieşire. Acest raport este în corespondență directă cu fişierul ***ucf.*** Dacă fişierul ***ucf*** nu a fost elaborat, atunci întregul raport poate fi copiat într-un fişier cu extensia ***ucf,*** selectând din meniul *Projet* opțiunea New Source File.

Procesul Generate Programming File. În momentul în care acest proces este rulat, proiectarea circuitului este în faza de programare la nivel de circuit Xilinx FPGA/CPLD.

Funcționalitatea circuitului a fost testată, iar cerințele legate de timpii de funcționare au fost soluționate.

Ca şi procesele anterioare, şi acest proces conține un număr semnificativ de parametri care pot fi configurați. Modalitatea de acces şi setare este identică cu cea explicată la procesele de mai sus. Acest proces conține trei subprocese şi anume:

- Programming File Generation Report - acest raport este rezultatul executării procesului *BITGEN*. In acest raport se pot examina setările salvate în urma rulării tuturor proceselor descrise anterior.

- Generate PROM or ACE or JTAG File - rezultatul rulării acestui proces va fi transferat în circuitul Xilinx FPGA/CPLD si reprezintă varianta fizică a circuitului proiectat. De fapt rularea acestui proces implică realizarea unei conversii dintr-un fişier cu extensia ***bit*** care este un fişier cu format proprietar Xilinx, într-un fişier al cărui format este unul utilizat în industrie.

- Configure Device (iMPACT) - acest process este utilizat numai pentru transferarea efectivă a circuitului proiectat într-un circuit Xilinx FPGA/CPLD disponibil. Transferarea între calculator şi circuitul Xilinx se va realiza cu ajutorul unui cablu proprietar. La finalul acestui proces, circuitul Xilinx FPGA/CPLD va conține prototipul circuitului proiectat.

Anexa B

Prezentarea simulatorului software SPIM

B.1 Prezentare generală

SPIM este un simulator software care execută programe scrise pentru procesoarele MIPS R2000/R3000. SPIM poate citi şi executa imediat fişiere în limbaj de asamblare sau fişiere executabile. El conţine un utilitar de depanare şi oferă câteva servicii asemănătoare cu cele ale sistemului de operare.

Odată lansată comanda ***xspim*** el afişează pe ecran fereastra prezentată în figura: B.l. Fereastra principală este împărţită în cinci panouri:

1. Panoul superior se numeşte *afişare registre.* El prezintă valoarea registrelor în UCP MIPS şi UVM (Unitatea de Virgulă Mobilă). Acest afişaj este actualizat de fiecare dată când programul se opreşte din execuţie.
2. Panoul doi conţine *butoanele de control* pentru operarea simulatorului. Butoanele realizează următoarele operaţii:
   - **quit** - ieşire din simulator;
   - **load** - citeşte un fişier sursă sau executabil în SPIM;
   - **run** - porneşte execuţia programului;
   - **step** - execută un pas din program;

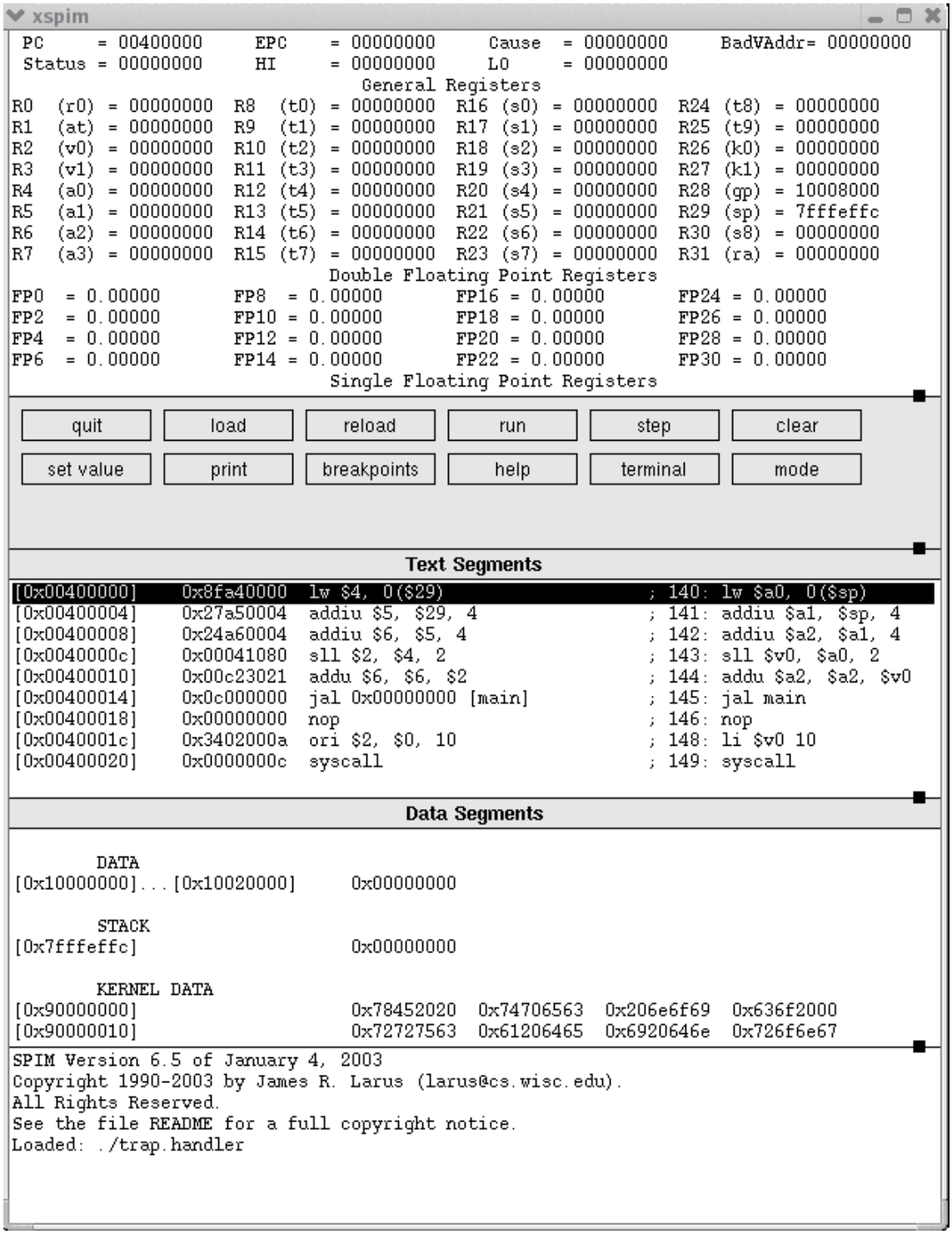


Figura B.1: Fereastra principală a simulatorului SPIM.

- **clear** - reinitializează registrele si memoria;
- **set value** - stabileşte valoarea dintr-un registru sau o locație de memorie;

- **print** - tipăreşte valoarea dintr-un registru sau o locaţie de memorie;
- **breakpoint** - stabileşte sau anulează un punct de întrerupere sau enumera toate punctele de întrerupere;
- **help** - afişează un mesaj de ajutor;
- **mode** - stabileşte modul de operare SPIM

3. următorul panou se numeşte *segment de text.* In acest panou sunt afişate instrucţiuni aparţinând programului curent, cât şi codul sistemului încărcat automat în momentul lansării simulatorului. Fiecare instrucţiune este afişată pe câte o singură line:

```
[0x00400000] Ox8fa40000 lw $4, 0($29) ; 89 : lw $aO, 0($sp)
```

Primul număr, cel între paranteze drepte, reprezintă adresa hexazecimală a locaţiei de memorie unde se află memorată instrucţiunea. Cel de al doilea număr rezintă codificarea numerică a instrucţiunii. Următoarea parte din instrucţiune (până la punct şi virgulă) este descrierea mnemonică a instrucţiunii. Linia efectivă din fişierul scris în asamblare începe după punct şi virgulă. Valoarea 89 reprezintă numărul liniei din fişierul care conţine instrucţiuni în limbaj de asamblare. Dacă linia nu mai conţine nimic după punct şi virgulă, atunci instrucţiunea a fost produsă de SPIM ca parte din translatarea unei pseudoinstrucţiuni.

4. următorul panou, *stive de date si segment,* oferă spre vizualizare datele încărcate în memoria programului precum şi datele din stiva programului.
5. ultimul panou cuprinde mesajele de eroare.

B.2 Încărcarea şi rularea unui program

În primul rând trebuie să se încarce programul scris în limbaj de asamblare. Această operaţie se realizează prin selectarea butonului **load.** Odată selectat acest buton, se va deschide o nouă fereastră în care se introduce numele fişierului şi se va selecta noul buton **assembly file.** Rolul noului buton **abort** este evident.

Odată selectat butonul **assembly file,** programul este încărcat în simulator, rularea sa realizându-se odată cu selecţia butonului **run.** Odată selectat butonul **run,** se va activa o fereastră cu două casete de text şi două butoane. Aceste casete conţin valorile corecte pentru executarea programului şi, în concluzie, ele pot fi ignorate, deci se selectează butonul **ok.**

Observație în timpul rulării programului, conținutul panoului de afişare al registrelor este inactiv deoarece aceste registre sunt modificate continuu. Dacă se doreşte oprirea programului se va tasta CTRL-C. Acum se va activa o nouă fereastră, înainte de a acționa în această nouă fereastră, se va observa conținutul registrelor şi al memoriei pentru a observa cum rulează programul. Dacă este prea complicat, rulați programul pas cu pas.

Dacă programul conține instrucțiuni care scriu/citesc la terminal, simulatorul afişează o nouă fereastră denumită consolă. Toate caracterele scrise de către program sunt afişate la consolă. Toate intrările în program se fac tot prin intermediul consolei.

B.3 Deosebiri față de un procesor normal

SPIM-ul nu simulează latența cache-ului sau a memoriei şi nici nu reflectă cu exactitate întârzierile operațiilor de virgulă mobilă sau ale instrucțiunilor de înmulțire şi împărțire.

O pseudoinstrucțiune este expandată la mai multe instrucțiuni maşină, în execuția pas cu pas sau când se examinează memoria, instrucțiunile care apar diferă de programul sursă. Corespondența între cele două seturi de instrucțiuni este simplă.

Procesoarele pot număra octeții din cuvânt, octetul cu numărul cel mai mic este sau cel mai din stânga sau cel mai din dreapta. Procesoarele MIPS pot opera cu ordinea octeților big-endian sau little-endian. Directiva *.byte 0, l, 2, 3* într-o maşină de tip big-endian va rezulta într-un cuvânt de memorie al cărui conținut este de la stânga la dreapta 0, 1,2, 3, în timp ce la o maşină de tip little-endian ordinea este inversă.

B.4 Apeluri sistem

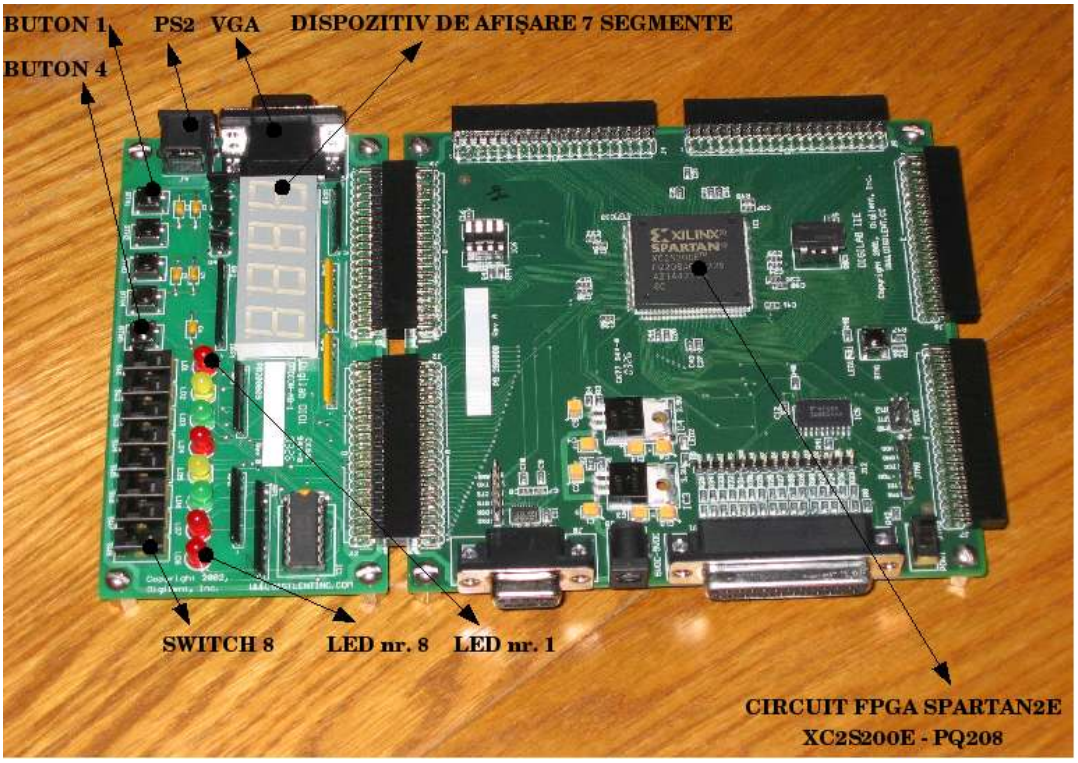
SPIM oferă câteva servicii asemănătoare cu cele pe care sistemul de operare le oferă prin intermediul instrucțiunii de apel de sistem - *syscaii.* Programul încarcă codul apelului de sistem în registrul *$ VO* şi argumentele în registrele *$ aO - $ a3 ($ f!2* pentru valorile de virgulă mobilă). Valorile întoarse de apelurile sistem se regăsesc în registrul *$ vO ($ fO* pentru rezultatele de virgulă mobilă).

Serviciile oferite de SPIM sunt prezentate în tabelul B. L

Serviciu	Codul apelului de sistem	Argumente	Rezultat
print_int	1	$ a0 = integer	
print_float	2	$ f12 = real	
print_double	3	$ f12 = double	
print_string	4	$ a0 = string	
read_int	5		integer (în $ v0)
read_float	6		float (în $ f0)
read_double	7		double (în f0)
read_string	8	$ a0 = buffer, $ a1 = lungime	
sbrk	9	$ a0 = cantitate	adresă (în $ v0)
exit	10		

Tabelul B.1: Serviciile oferite de SPIM.

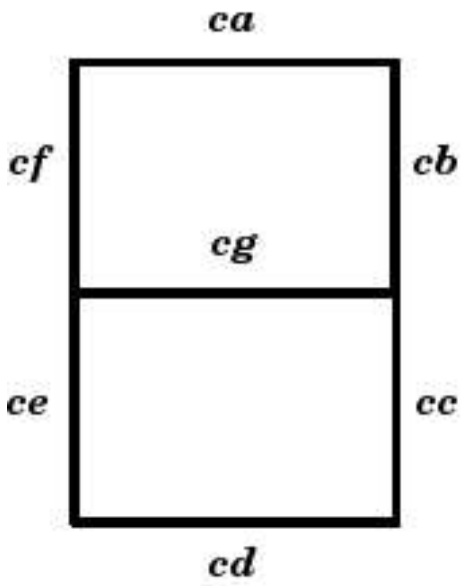
Valoriile pinilor programabili pentru DIGILAB DIO1 şi DIGILAB 2E



Denumire resursă	*Pin*	*Denumire resursă*	*Pin*	*Denumire resursă*	*Pin*		
SW 1	*P 16*	*BTN 2*	*P 41*	*LED nr. 7*	*P 61*	B0	P 188
SW 2	*P 18*	*BTN 3*	*P 42*	*LED nr. 8*	*P 63*	BTN	P 77
SW 3	*P 21*	*BTN 4*	*P 43*	PS2CLK	P 189	LED	P 69
SW 4	*P 23*	*LED nr. 1*	*P 44*	PS2D	P 187		
SW 5	*P 27*	*LED nr. 2*	*P 46*	CLOCK	P 80		
SW 6	*P 30*	*LED nr. 3*	*P 48*	HSYNC	P 192		
SW 7	*P 33*	*LED nr. 4*	*P 55*	VSYNC	P 194		
SW 8	*P 35*	*LED nr. 5*	*P 57*	R0	P 193		
BUTON (BTN) 1	*P 40*	*LED nr. 6*	*P 59*	G0	P 191		

Dispozitive de afişare 7 segmente

Placa de I/O DIGILAB DIO1 este prevăzutăcu un număr de 4 dispozitive de afişsare 7 segmente. Pentru a le referi sunt necesare 4 semnale codificate dupăcum urmează: AN1 – dispozitivul 1, AN2 – dispozitivul 2, AN3 – dispozitivul 3şi AN4 dispozitivul 4.
Segmentele sunt codificate conform figurii de mai jos, iar punctul zecimal este referit prin intermediul semnalului DP.



De reţinut este că dacă se doreşte iluminăm unui anumit segment atunci semnalul segmentului respectiv trebuie să aibă valoarea 0 (zero).

Denumire resursă	*Pin corespunzător*	*Denumire resursă*	*Pin corespunzător*
ca	*P 17*	*AN 2*	*P 47*
cb	*P 20*	*AN 3*	*P 49*
cc	*P 22*	*AN 4*	*P 56*
cd	*P 24*		
ce	*P 29*		
cf	*P 31*		
cg	*P 34*		
DP	*P 36*		
AN 1	*P 45*		

Pinii pentru LCD – Seiko (2 rânduri, 16 caractere)

```
NET "db<0>"  LOC = "P58"; # 10   data 0   a11
NET "db<1>"  LOC = "P59"; #  9   data 1   a10
NET "db<2>"  LOC = "P60"; #  8   data 2   a9
NET "db<3>"  LOC = "P61"; #  7   data 3   a8
NET "db<4>"  LOC = "P62"; #  6   data 4   a7
NET "db<5>"  LOC = "P63"; #  5   data 5   a6
NET "db<6>"  LOC = "P64"; #  4   data 6   a5
NET "db<7>"  LOC = "P68"; #  3   data 7   a4
NET "db<8>"  LOC = "P57"; #  2   en       a12
NET "db<9>"  LOC = "P56"; #  1   rw       a13
NET "db<10>" LOC = "P55"; #  0   rs       a14
```