CERCETĂRI ALE EFICIENŢEI METODELOR DE CREŞTERE A DEPENDABILITĂŢII LA TREAPTA CACHE A UNEI IERARHII DE MEMORII

Teză destinată obţineri
titlului ştiinţific de doctor inginer
la
Universitatea „Politehnica” din Timişoara
în domeniul ŞTIINŢA CALCULATOARELOR
de către

Ing. Ovidiu-Constantin Novac

Conducător ştiinţific: prof.univ.dr.ing. Mircea Vlăduţiu
Referenţi ştiinţifici: prof.univ.dr.ing. Mircea Petrescu
prof.univ.dr.ing. Vladimir Creţu
prof.univ.dr.ing. Ştefan Vari-Kakas

Ziua susţinerii tezei: 01.02.2008

Seriile Teze de doctorat ale UPT sunt:

1. Automatică
2. Chimie
3. Energetică
4. Ingineria Chimică
5. Inginerie Civilă
6. Inginerie Electrică
7. Inginerie Electronică şi Telecomunicaţii
8. Inginerie Industrială
9. Inginerie Mecanică
10. Ştiinţa Calculatoarelor
11. Ştiinţa şi Ingineria Materialelor

Universitatea „Politehnica” din Timişoara a iniţiat seriile de mai sus în scopul diseminării expertizei, cunoştinţelor şi rezultatelor cercetărilor întreprinse în cadrul şcolii doctorale a universităţii. Seriile conţin, potrivit H.B.Ex.S Nr. 14 / 14.07.2006, tezele de doctorat susţinute în universitate începând cu 1 octombrie 2006.

România, 300159 Timişoara, Bd. Republicii 9,
tel. 0256 403823, fax. 0256 403221
e-mail: editura@edipol.upt.ro

Cuvânt înainte

Teza de doctorat a fost elaborată pe parcursul activităţii mele în cadrul Catedrei de Calculatoare din cadrul Facultăţii de Inginerie Electrică şi Tehnologia Informaţiei a Universităţii din Oradea.

Mulţumiri deosebite se cuvin conducătorului de doctorat profesor doctor inginer Mircea Vlăduţiu, conducătorul ştiinţific al stagiului de pregătire, pentru sprijinul moral şi ştiinţific acordat, pentru sfaturile şi îndrumarea pas cu pas în realizarea acestei teze precum şi pentru răbdarea, înţelegerea şi încrederea acordată.

Îmi exprim consideraţia şi recunoştinţa faţă de colegii din catedră şi în special d-lui Prof. dr. ing. Vari Kakas Ştefan de la Facultatea de Inginerie Electrică şi Tehnologia Informaţiei din Oradea, pentru tot sprijinul şi sfaturile acordate pe toată perioada de pregătire a tezei.

Timişoara, ianuarie 2008 Ovidiu - Constantin Novac

Destinatarii dedicaţiei.

Antoniu şi Mihaela Novac

Novac, Ovidiu Constantin

Cercetări ale eficienţei metodelor de creştere a dependabilităţii la treapta cache a unei ierarhii de memorii

Teze de doctorat ale UPT, Seria 10, Nr. 15, Editura Politehnica, 2008, 218 pagini, 102 figuri, 17 tabele

ISSN: 1842-7707

ISBN: 978-973-625-593-9

Cuvinte cheie: memorie cache, cache Ram cache Tag, hit, miss

Rezumat,

Teza este structurată pe şapte capitole. În capitolul 1, s-au prezentat date referitoare la stadiul actual al configuraţiilor de cache-uri, locul treptei cache într-o ierarhie de memorii, problemele cache-urilor şi perturbarea acestora prin defectare, analiza unor soluţii existente de cache-uri cu capabilităţi de îmbunătăţire a dependabilităţii. Capitolul 2 analizează defectele care pot afecta cache-urile, defecte la nivelul celulei SRAM, precum şi defecte la nivelul legăturilor dintre celulele SRAM. În capitolul 3, principala abordare o constituie configurarea structurii unui cache tipic urmărind stabilirea locului de introducere a elementelor de redundanţă menite a creşte dependabilitatea, interfaţarea cache-ului cu celelalte trepte ale unei ierarhii de memorii, detalierea constructivă a unui cache, sincronizarea funcţională a treptei cache. Capitolul 4, analizează metricile de performanţă şi fiabilitate la cache-uri, metode de îmbunătăţire a performanţei cache-urilor şi afectarea lor prin indicatorul CDLR. În capitolul 5 am prezentat mediul de experimentare reprezentat de simulatorul de cache, obiectivele de construcţie ale simulatorului, precum şi detalii constructive ale simulatorului – CACHE şi ale mediului de evaluare CDLR SPEC 2000. Capitolul 6 scoate în evidenţă rezultatele experimentale obţinute prin intermediul simulatorului CDLR SPEC 2000, evaluări de indici de fiabilitate, implementarea cu circuite programabile FPGA Xilinx a unui cod SEC-DED, iar în ultimul capitol, capitolul 7, am prezentat contribuţiile personale precum şi posibile dezvoltări ale cerecetărilor întreprinse.

CONŢINUT

LISTA PRESCURTĂRILOR UTILIZATE

ADR – Address
AMAT – Average Memory Acces Time
BILBO – Built In Logic Block Observer
BIST – Built In Self Test
CAM – Content Addressable Memory
CDLR – Cache Data Loss Rate
CFdr – Deceptive read disturb coupling fault
CFds – Disturb coupling fault
CFFCdr – Deceptive read disturb coupling fault from cache
CFFCir – Incorrect read coupling fault from cache
CFFCrd – Read disturb coupling fault from cache
CFICds – Disturb coupling fault in cache
CFid – Idempotent coupling fault
CFir – Incorrect read coupling fault
CFrd – Read disturb coupling fault
CF_{st} – State coupling fault - CF_{st}
CFtr – Transition coupling fault
CFwd – Write disturb coupling faults
CLB – Configurable Logic Block
CMPS –Cel mai puţin semnificativ
CMS –Cel mai semnificativ
CNTR –Control
CPI – Clock cycle Per Instruction
CPU – Central Processing Unit
CTR_LD –Control Load
DLR – Data Loss Rate
DRAM – Dinamic Random Acces Memory
DRDF – Deceptive read disturb faults
DRDFFC – Deceptive read disturb faults from cache
FFM – Functional Fault Model
FIFO – First In First Out
FPGA – Field Programmable Gate Arrays
GND – Ground
IC – Instruction count
IOB – Input Output Block
IRF – Incorrect read faults
IRFFC – Incorrect read faults from cache
LFSR – Linear Feedback Shift Register
LFU – Least Frequently Used
LRU – Least Recently Used

LSB – Least Significant Byte
MISR – Multiple Input Shift Register
MOS – Metal Oxid Semiconductor
MTBF – Mean Time Between Failures
MTTDL – Mean Time To Data Loss
MTTF – Mean Time To Failure
MTTFF – Mean Time To First Failure
MTTR – Mean Time To Repair
OBI – One-Bit Implementation
PSM – Programable Switch Matrices
RAM – Random Acces Memory
RDF – Read disturb faults
RDFFC – Dynamic read disturb faults from cache
RTL – Resistor Transistor Logic
SAF – Stuck-at faults
SAFIC – Stuck-at faults in cache
SBC – Stream-Based Compression
SBDT – Stream Based Data Trace
SBIT – Stream Based Instruction Trace
SEC-DED – Single bit Error Correcting and Double bit Error Detection
SF – State Faults
SOS – Sensitizing Operation Sequence
SPEC2000 – Standard Performance Evaluation Corporation 2000
SPEC’92 – Standard Performance Evaluation Corporation ’92
SRAM – Static Random Acces Memory
STF – Stream Table File
TF – Transition Faults
TLB – Translation Look-aside Buffers
VDD – Positive supply voltage
VLSI – Very Large Scale Integration
WDF – Write disturb faults

0. INTRODUCERE

0.1. ASUPRA OPORTUNITĂŢII ŞI ACTUALITĂŢII CERCETĂRILOR DIN TEZĂ

Un aspect care ridică deosebite probleme, este creşterea destul de lentă a vitezei memoriei comparativ cu creşterea vitezei procesoarelor [WuMc95], [HePa06]. Procesorul îşi alocă, din timpul de execuţie, o fracţiune de timp tot mai mare, în aşteptarea datelor ce urmează a fi aduse din memoria principală. Pentru reducerea diferenţei între viteza memoriei şi viteza procesorului, procesoarele actuale îşi alocă majoritatea resurselor hardware, treptei cache. De exemplu procesorul Itanium 2 alocă 86% din tranzistoare pentru nivelul cache L3 [CRST06],[Hung06].

0.2. OBIECTIVELE CERCETĂRILOR DIN TEZĂ

În analiza eficienţei metodelor de creştere a dependabilităţii la treapta cache a ierarhiei de memorii, motivul care m-a determinat să mă orientez spre aplicaţii critice din punctul de vedere al fiabilităţii este memoria cache. În cadrul acestei teze vom folosi atât denumirea de cache, cât şi denumirea de memorie cache.

În urma parcurgerii referinţelor bibliografice şi după modelul propus de Ad.J.van de Goor în [GoAl00], am făcut o clasificare a defectelor generale care se pot întâlni la memorii. Am selectat din această clasificare, după parcurgerea referinţelor bibliografice, defectele specifice cache-urilor. Soluţiile existente acoperă doar o parte a defectelor întâlnite la cache-uri.

După ce am studiat şi parcurs codurile detectoare şi corectoare a erorilor prezentate de T.R.N. Rao şi E. Fujiwara în [RaFu89], precum şi alte cărţi de specialitate care sunt citate în referinţele bibliografice, am ales codul detector şi corector al erorilor, codul Hsiao. Am ales acest cod Hsiao, datorită proprietăţilor sale. O proprietate importantă a acestui cod este numărul impar de biţi de 1 pe fiecare coloană a matricii generatoare. Această proprietate conduce la generarea unor ecuaţii echilibrate şi în consecinţă şi a unei scheme echilibrate de corecţie a erorilor. Astfel soluţia propusă prin codul Hsiao este acoperitoare pentru defecte specifice cache-urilor. Pentru eficientizarea dependabilităţii la treapta de cache am

introdus o nouă mărime de măsurare a dependabilităţii şi anume CDLR (CDLR reprezintă rata globală de pierdere a datelor). Pentru a putea verifica eficientizarea dependabilităţii la treapta cache, am realizat un mediu experimental CACHE, care are ca obiectiv simularea unei ierarhii de memorii. Acest mediu l-am construit luând ca model simulatorul SMP CACHE 2.0 [VSMZ00], el fiind construit pentru mai multe niveluri de cache, la ambele simulatoare folosindu-se trace-urile benchmark-urilor SPEC'92. Am îmbunătăţit acest simulator introducând calculul mărimii CDLR şi am introdus trace-urile benchmark-urilor SPEC 2000, rezultând astfel mediul de evaluare CDLR SPEC 2000. Cu acest nou mediu de evaluare, dedicat, am obţinut rezultate experimentale, prin care se poate verifica eficientizarea dependabilităţii la treapta cache.

Ca şi concluzie am putea spune că eforturile actuale în privinţa cercetărilor asupra cache-urilor atât în partea de procesare sau multiprocesare reprezintă o nouă cale care ţinteşte intensificarea şi exploatarea diferitelor tipuri de localităţi explicite sau implicite în partea de cod sau de date ale aplicaţiei. Se poate spune că o tratare eficientă a localităţii ne poate ajuta să obţinem o îmbunătăţire semnificativă în domeniul performanţă şi în domeniul raportului performanţă /complexitate.

În final doresc să mai menţionez faptul că în literatura de specialitate acest aspect este foarte puţin tratat.

0.3. ORGANIZAREA TEZEI

Teza este structurată pe şapte capitole. În capitolul 1, s-au prezentat date referitoare la stadiul actual al configuraţiilor de cache-uri, locul treptei cache într-o ierarhie de memorii, problemele cache-urilor şi perturbarea acestora prin defectare, analiza unor soluţii existente de cache-uri cu capabilităţi de îmbunătăţire a dependabilităţii. Capitolul 2 analizează defectele care pot afecta cache-urile, defecte la nivelul celulei SRAM, precum şi defecte la nivelul legăturilor dintre celulele SRAM. În capitolul 3, principala abordare o constituie configurarea structurii unui cache tipic urmărind stabilirea locului de introducere a elementelor de redundanţă menite a creşte dependabilitatea, interfaţarea cache-ului cu celelalte trepte ale unei ierarhii de memorii, detalierea constructivă a unui cache, sincronizarea funcţională a treptei cache. Capitolul 4, analizează metricile de performanţă şi fiabilitate la cache-uri, metode de îmbunătăţire a performanţei cache-urilor şi afectarea lor prin indicatorul CDLR. În capitolul 5 am prezentat mediul de experimentare reprezentat de simulatorul de cache, obiectivele de construcţie ale simulatorului, precum şi detalii constructive ale simulatorului – CACHE şi ale mediului de evaluare CDLR SPEC 2000.

Capitolul 6 scoate în evidenţă rezultatele experimentale obţinute prin intermediul simulatorului CDLR SPEC 2000, evaluări de indici de fiabilitate, implementarea cu circuite programabile FPGA Xilinx a unui cod SEC-DED, iar în ultimul capitol, capitolul 7, am prezentat contribuţiile personale precum şi posibile dezvoltări ale cerecetărilor întreprinse.

Autorul, împreună cu alte cadre didactice, au obţinut rezultate remarcabile la diferite conferinţe internaţionale şi naţionale cu recunoaştere internaţională [NoNo99], [NoNo01], [NovN01], [NoNo02], [VaNo03], [VNPN05], [NoGN05], [NVPN06], [NoNP06], [NVVN06], [NoVP07].

Mulţumesc pe această cale **d-lui Prof.dr.ing. MIRCEA VLĂDUŢIU** - conducătorul ştiinţific al stagiului de pregătire, pentru sprijinul moral şi ştiinţific acordat, pentru sfaturile şi îndrumarea pas cu pas în realizarea acestei teze precum şi pentru răbdarea, înţelegerea şi încrederea acordată. Îmi exprim consideraţia şi recunoştinţa faţă de colegii din catedră şi în special **d-lui Prof.dr.ing. Vari Kakas Ştefan**, pentru sprijinul şi sfaturile acordate pe toată perioada de pregătire a tezei.

Nu în ultimul rând doresc să mulţumesc familiei mele pentru întelegere, suport moral şi sacrificiile pe care le-au făcut pentru a-mi da posibilitatea de a elabora această teză.

1. ASUPRA STADIULUI ACTUAL AL CONFIGURAŢIILOR DE CACHE-URI

1.1. LOCUL TREPTEI CACHE ÎNTR-O IERARHIE DE MEMORII

O ierarhie de memorie este soluţia la nevoia programatorilor de a avea o memorie de capacitate mare şi foarte rapidă. Această ierarhie este organizată pe mai multe niveluri, fiecare având capacitate de stocare mai mică, viteză mai mare şi cost pe bit mai mare decât nivelul anterior. Obiectivul care se urmăreşte, în cazul unei ierarhii de memorii, este obţinerea unui sistem de memorie care are costul aproape la fel de mic ca şi cel mai ieftin nivel de memorie şi viteza aproape la fel de mare ca şi nivelul cel mai rapid [ZaMi04].

Ierarhia de memorii se bazează pe câteva proprietăţi fundamentale ale tehnologiei de stocare a informaţiei. Aceste proprietăţi le voi enumera în continuare. Diferite tehnologii de stocare au timpii de acces diferiţi. Tehnologiile mai rapide au un cost pe bit mai mare decât tehnologiile lente, dar acestea din urmă au o capacitate de stocare mai mare a informaţiei. În fig. 1.1. este prezentată o astfel de ierarhie de memorii.

Ierarhia are la baza piramidei nivelul cel mai lent, cel mai ieftin şi cu capacitatea de stocare cea mai mare. Pe măsură ce ne îndreptăm spre vârful piramidei, vom avea niveluri tot mai rapide, cu cost pe bit mai mare şi cu capacitate de stocare tot mai mică. În nivelul din vârful piramidei (nivelul 0, L0), avem un număr redus de regiştrii ai CPU, regiştrii ce au un timp de acces foarte mic, ei fiind accesaţi de către CPU într-un singur ciclu de clock.

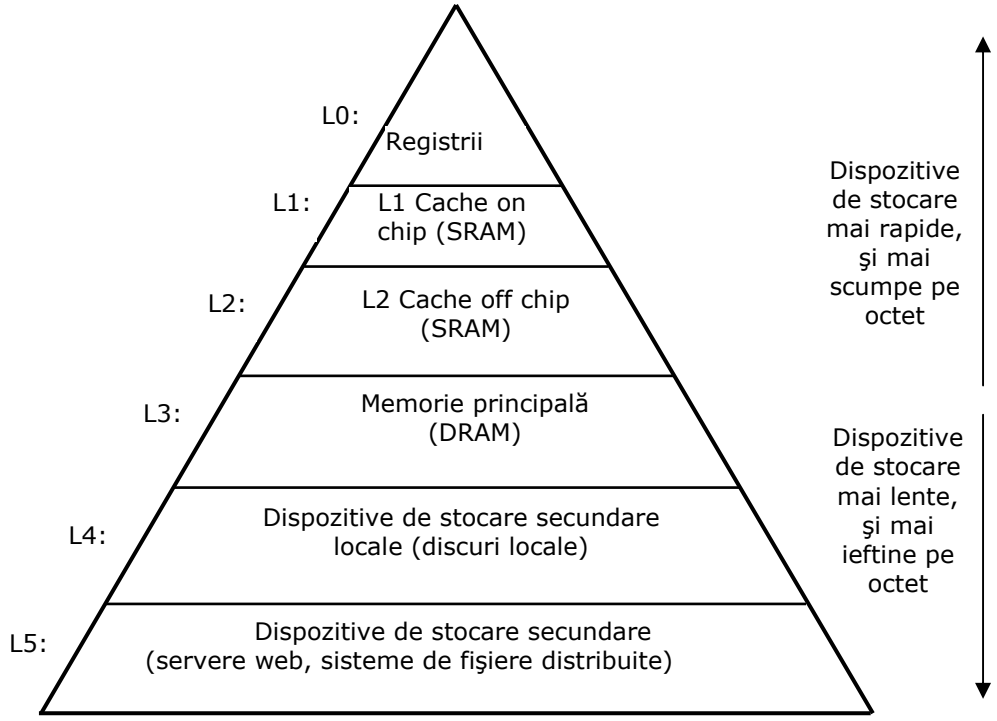


Fig. 1.1

Urmează una sau două niveluri unde avem memorii cache SRAM de dimensiuni medii, acestea putând fi accesate în câteva cicluri de clock CPU. Pe următorul nivel avem memoria principală DRAM, cu capacitate mare de stocare a informaţiei, această memorie putând fi accesată în zeci sau sute de cicluri de clock. În continuare urmează discurile locale care au dimensiuni foarte mari, ele având dezavantajul că sunt foarte lente. Pe ultimul nivel, unele sisteme includ un nivel adiţional de discuri sau servere remote, care pot fi accesate prin intermediul unei reţele. De exemplu World Wide Web-ul permite programelor să acceseze fişiere care sunt stocate pe servere Web oriunde în lume.

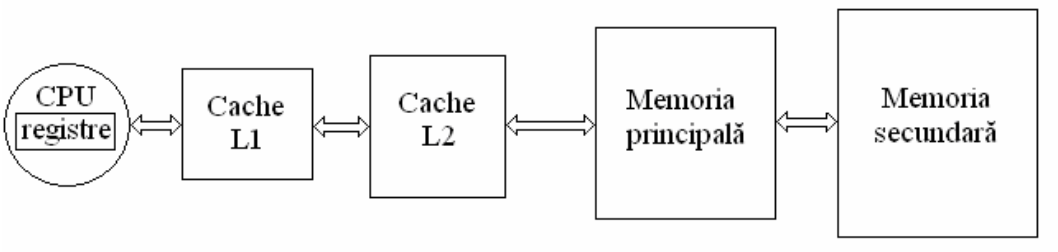


Fig. 1.2.

Există ierarhii de memorii cu două niveluri de cache-uri L1 şi L2. În fig. 1.2. este prezentată o astfel de ierarhie cu două niveluri de cache-uri L1 şi L2 exterioare CPU-ului, iar în fig. 1.3. este prezentată o altă ierarhie de memorie tot cu două niveluri de cache-uri în care nivelul L1 se găseşte în interiorul CPU, iar nivelul L2 în exteriorul acestuia.

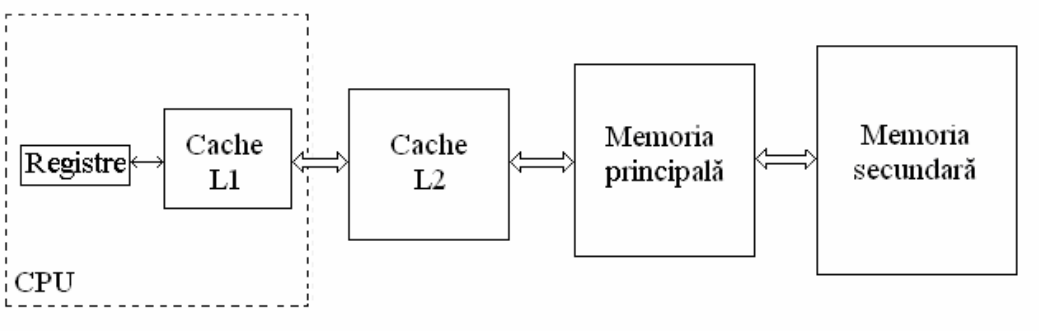


Fig. 1.3.

Interfaţa dintre memorie şi CPU joacă un rol important în stabilirea performanţei globale în orice sistem de calcul. Accesele la memorie sunt întotdeauna mai lente decât operaţiile din interiorul CPU, mai ales în cadrul microprocesoarelor unde CPU este încapsulată într-un singur chip, iar memoria este distribuită în alte circuite integrate. Accesul la resursele care se găsesc în interiorul aceluiaşi chip este mai rapid decât accesul la resursele din afara chipului, nefiind necesară traversarea amplificatoarelor de semnal asociate fiecărui pin de interfaţare cu exteriorul chip-ului. În sistemele de calcul care conţin microprocesoare se pot stabili un număr de niveluri din care este alcătuită ierarhia de memorii. Uzual primului nivel, i se alocă colecţia de registre ale CPU, acestea fiind cele mai apropiate de CPU, deci uşor de accesat. Cache-ul L1 se mai numeşte cache primar şi reprezintă, de obicei, cel de-al dolea nivel al ierarhiei de memorii, găsindu-se încapsulat în acelaşi chip de memorie cu CPU aşa cum se prezintă în fig. 1.3. sau în exteriorul CPU aşa cum se prezintă în fig. 1.2.

Am presupus până în acest moment că un cache stochează doar date. În sistemele de calcul actuale, cache-ul este split-at, memorând atât date cât şi instrucţii. Un cache care stochează date îl denumim d-cache, iar cel care stochează instrucţii îl denumim i-cache. Dacă cache-ul stochează atât instrucţii cât şi date este un cache unificat. Un sistem de calcul convenţional are un cache L1 de tip i-cache, un cache L1 de tip d-cache, ambele fiind în interiorul procesorului, şi un cache unificat, L2, în exteriorul procesorului. În fig. 1.4. se prezintă acest tip de cache.

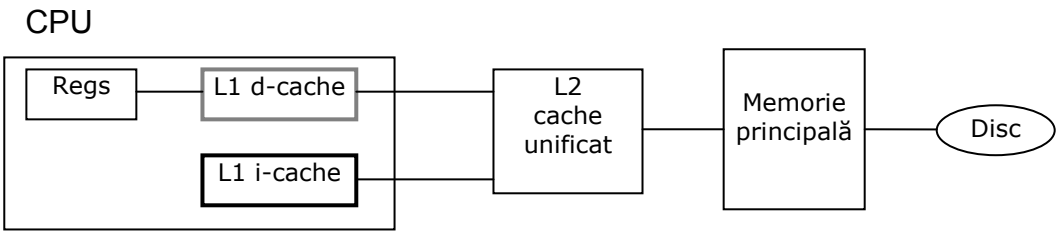


Fig. 1.4.

Cache-ul L2 se mai numeşte şi cache secundar. El reprezintă cel de-al treilea nivel al ierarhiei de memorii. Cache-ul secundar are o dimensiune mult mai mare decât cel primar. În unele sisteme această memorie este implementată în afara chipului CPU, fiind un cache unificat, atât de date cât şi de instrucţii. Odată cu creşterea densităţii de integrare în interiorul circuitelor integrate de memorie, în unele sisteme de calcul, cache-ul secundar poate fi încapsulat în cadrul aceluiaşi chip împreună cu CPU şi cache-ul primar. În acest caz însă apare necesitatea introducerii unui al treilea nivel de cache, L3, care este situat în exteriorul CPU, aşa cum se prezintă în fig. 1.5. Memoria principală este cel de-al patrulea nivel al ierarhiei. Această memorie poate fi accesată şi direct de către CPU, ea conţinând codul şi datele programelor care rulează la un moment dat pe sistemul de calcul. Memoria principală are însă un dezavantaj şi anume faptul că ea nu este suficient de mare pentru a putea conţine toată informaţia de care are nevoie un utilizator. Acest dezavantaj a dus la introducerea unui alt nivel în ierarhia de memorii, nivel denumit memorie secundară. Aceasta are dimensiunea mult mai mare decât a memoriei principale, şi este folosită pentru depozitarea informaţiei într-un sistem de calcul.

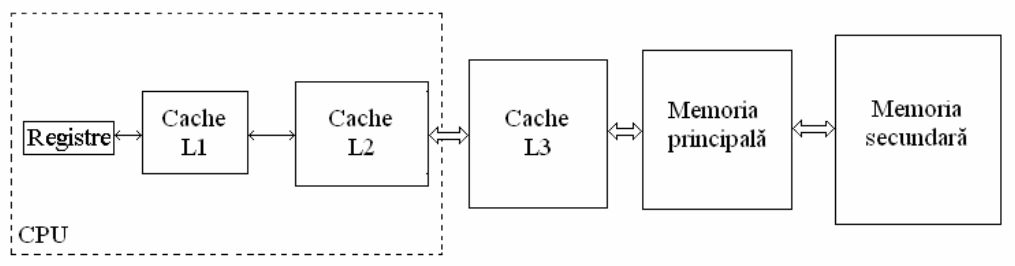


Fig. 1.5.

O mare parte a infrastructurii oricărui sistem de calcul este dedicată subsistemului de memorie. La procesoarele actuale (cum ar fi Intel Pentium) se alocă niveluri multiple de cache-uri de date şi de instrucţii, tabele de predicţii şi TLB (Translation Look-aside Buffers) [Inte06].

1.2. PROBLEMELE CACHE-URILOR ŞI PERTURBAREA ACESTORA PRIN DEFECTARE

Un cache este util numai dacă anumite informaţii sunt folosite frecvent şi mult. Atunci acele informaţii merită păstrate în această memorie. Din punct de vedere experimental s-a constatat că acest lucru este de cele mai multe ori adevărat. Această observaţie poate fi formulată în diferite moduri, unul dintre ele este principiul localităţii [LeLK00].

Eficienţa cache-ului este obţinută prin exploatarea potrivită a principiilor localităţii temporale şi spaţiale. Localitatea temporală înseamnă că probabilitatea este relativ mare, că o dată sau o instrucţie să fie reutilizată în viitorul apropiat. Localitatea spaţială înseamnă că probabilitatea este relativ mare ca următoarea dată sau instrucţie să fie utilizată dacă aceasta este în vecinătatea datei sau instrucţiei utilizată anterior. În sistemele tradiţionale, localitatea temporală este exploatată prin menţinerea a celor mai recent utilizate date / instrucţii în cache şi prin încorporarea în ierarhia de memorie. Localitatea spaţială este exploatată prin utilizarea unor blocuri de cache mai mari şi prin încorporarea unor mecanisme de prefech în sistemul cache. Pe măsură ce tehnologia a devenit din ce în ce mai sofisticată, este evident faptul că o mai bună performanţă poate fi obţinută prin încorporarea unor soluţii mai sofisticate pentru îmbunătăţirea şi exploatarea localităţii prezente în partea de cod sau de date. Pe măsură ce procesoarele au devenit tot mai complexe, performanţele şi proiectarea cache-urilor au avut un efect tot mai mare prin soluţii utilizate în alte domenii cum ar fi super pipelinizarea, superscalarea, predicţie, paralelizare.

Conceptul cache a apărut ca o soluţie la creşterea continuă în domeniul timp între tehnologia procesoarelor şi tehnologia memoriei. Acest concept a evoluat într-un sofisticat sistem de soluţii implementate hard şi soft. Astăzi cel mai bun raport performanţă / complexitate este obţinut printr-o interacţiune a unor soluţii bazate pe hard şi soft. Tot ce înseamnă cache este deci foarte solicitat, astfel apare posibilitatea tot mai mare ca cache-ul să se defecteze [Smit82].

Datele între cache şi CPU se transferă în cantităţi întregi cum ar fi octeţi, cuvinte sau cuvinte duble, iar informaţia între cache şi memoria principală se transferă de obicei în cantităţi numite blocuri uneori şi linii.

Conform principiului localităţii există patru aspecte care ne interesează la orice nivel al ierarhiei de memorii: funcţia de mapare, identificarea blocului, algoritmul de replasare şi politicile de scriere [HePa03], [HePa96].

1.2.1. Problema mapării adreselor

Funcţia de mapare ne furnizează informaţii referitoare la modul în care cache-ul poate fi mapat. Astfel cache-ul poate fi mapat: direct, set-asociativ sau complet asociativ. Prezint în continuare maparea adreselor din memoria principală în cache [HePa96].

1.2.1.1 Maparea directă

Maparea directă se caracterizează prin faptul că admiţând un cache cu dimensiunea S_1 adresele de memorie principală notate cu j se mapează în adrese de cache i după regula i=j modulo S_1.

Modalitatea de mapare este prezentată sugestiv în fig. 1.6.

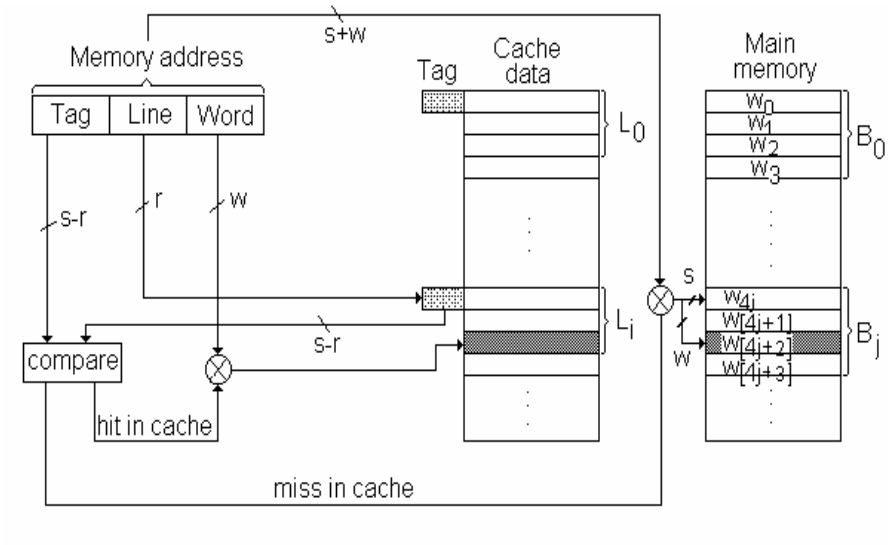


Fig. 1.6.

Adresa de memorie (Memory address) este split-ată în trei câmpuri: Tag, Line şi Word. Câmpul Tag identifică partea de tag din adresă, de dimensiune s-r biţi, câmpul Line identifică linia din cache şi are dimensiunea r biţi, iar ultimul câmp Word face identificarea cuvântului (Word) din cache si are dimensiunea de w biţi. La compararea celor s-r biţi ai adresei de memorie cu cei s-r biţi ai părţi de Tag din cache rezultă hit sau miss, practic se identifică linia mai întâi. În cazul unui hit cuvântul este identificat cu ajutorul celor w biţi ai adresei de memorie. În cazul unui

miss, cuvântul nefiind găsit în cache trebuie adus din memoria principală, folosind cei s+w biţi ai adresei de memorie. Dacă prin programare se prevăd accese repetate la un anumit bloc de cache acest lucru ar duce la un miss permanent [HePa03].

1.2.1.2. Maparea set-asociativă

Datorită scăderii anterior relevate la maparea directă la maparea set-asociativă, memoria principală se împarte în seturi, fiecare set fiind alcătuit dintr-un număr de blocuri.

Maparea unei adrese din memoria principală într-o linie de cache i se face după regula i = j modulo S_1' ($S_1'=2^d$), aspect prezentat în fig.1.7. [ChCh01], [ZhZY97], [Kess89]. Precizăm că S_1' este dimensiunea memoriei cache. În cazul acestui tip de mapare, adresa de memorie este split-ată tot în trei câmpuri: Tag, Set (index) şi Word. Câmpul Tag identifică partea de tag din adresă, de dimensiune s-d biţi, câmpul Set identifică setul din cache şi are dimensiunea d biţi, iar ultimul câmp Word face identificarea cuvântului (Word) din cache si are dimensiunea de w biţi.

Aceasta este o mapare set asociativă pe k – căi (k – way set associative cache). În măsura în care s-a făcut identificarea de tag (de exemplu setul i), blocul Bj poate fi plasat în oricare din k blocuri ale setului i, de exemplu în primul liber, k adrese ce aparţin unui set pot fi mapate fără probleme de suprapunere.

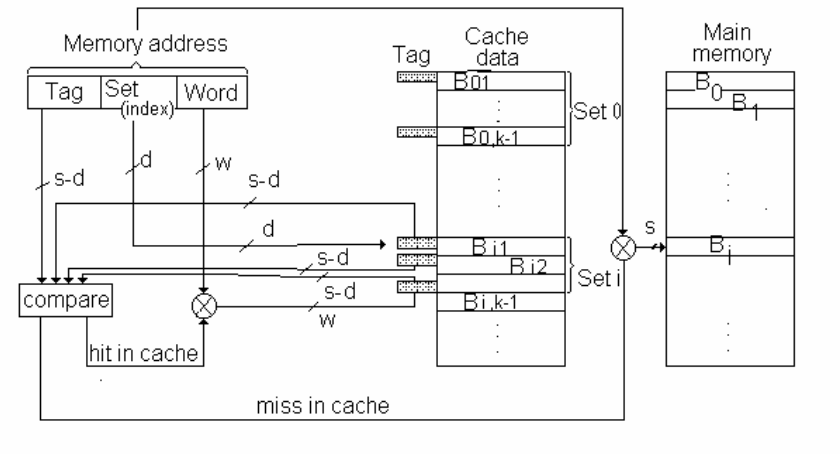


Fig. 1.7.

1.2.1.3. Maparea complet asociativă

În cadrul mapării complet asociative se permite maparea oricărei adrese de memorie principală în oricare linie liberă din memoria cache, aşa cum se poate observa în fig. 1.8. În acest caz adresa de memorie este split-ată doar în două câmpuri: Tag şi Word. Câmpul Tag identifică partea de tag din adresă şi are dimensiunea de s biţi, iar câmpul Word identifică cuvântul din cache, el are dimensiunea de w biţi. Spre deosebire de celelalte două tipuri de mapări anterior prezentate la maparea complet asociativă are loc compararea simultană a tag-ului pentru toate liniile.

Dacă în urma comparării a rezultat un hit, atunci este selectat doar cuvântul căutat cu ajutorul celor w biţi din adresa de memorie. Dacă în urma comparării a rezultat un miss atunci cuvântul căutat este adus din memoria principală cu ajutorul celor s+w biţi [HePa03].

Procesul de mapare poate fi afectat de malfuncţionări, sens în care considerăm următorul exemplu [Hung06]. Astfel, putem avea coruperea datelor în partea de tag a cache-ului şi astfel pot apărea hit-uri false şi miss-uri false. În fig. 1.9. este prezentat mecanismul prin care este posibilă apariţia unui hit fals.

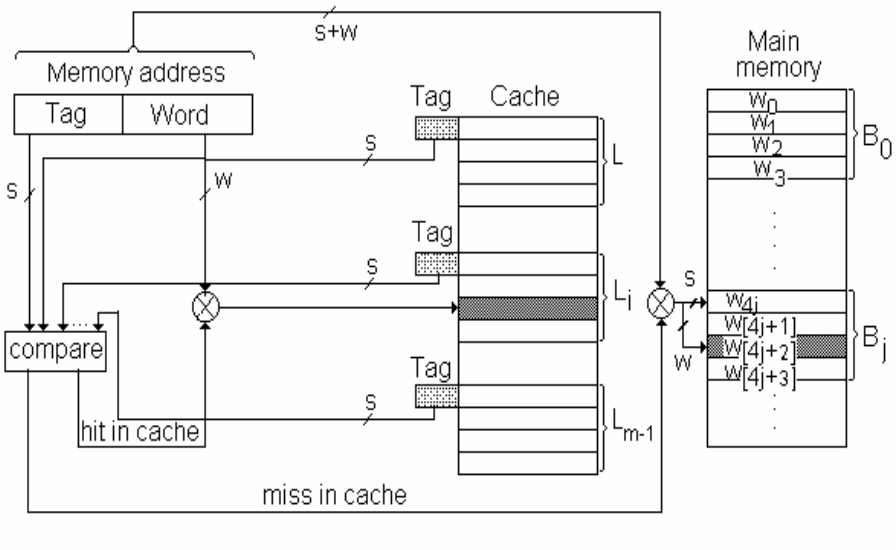


Fig. 1.8.

Astfel, să considerăm două elemente ale tag-ului iniţial, anume (0101) şi (0001). Un bit din primul element este corupt datorită, admiterii unei erori soft, valoarea acestui element fiind modificată din (0**1**01) în (0001).

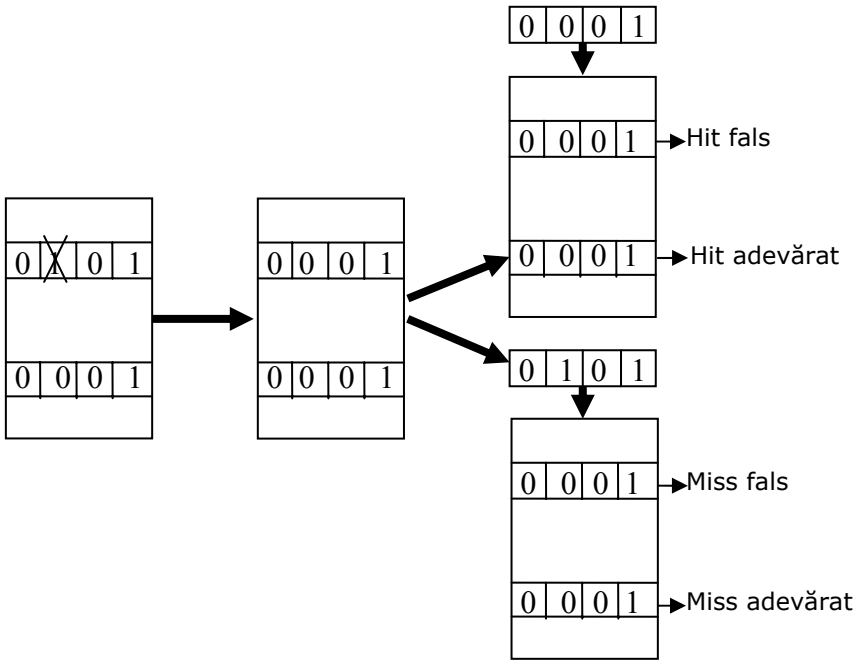


Fig. 1.9.

În această situaţie, tag-ul (0001) este introdus pentru căutare, în urma comparării va rezulta un succes, adică va rezulta un hit pentru elementul corupt, care este un hit fals. Acesta determină ca procesorul să încarce (citească) date sau să stocheze date, într-o locaţie incorectă. Erorile soft pot induce şi multi hit-uri.

Un miss fals este acela care la compararea de tag-uri ar trebui să fie un hit dacă erorile soft nu ar fi apărut. Şi în acest caz tag-ul are două elemente (0101) şi (0001). Admitem că un bit din primul element este corupt datorită unei erori soft determinând modificarea din (0**1**01) în (0**0**01). Dacă introducem pentru căutare elementul de tag (0001), el va cauza o nepotrivire, adică un miss, determinând probleme de integritate de date dacă intrarea care ar fi fost hit conţine date „dirty”. Datorită faptului că cache-urile write back pot conţine date „dirty”, aceste cache-uri sunt susceptibile să aibă probleme de integritate a datelor determinate de miss-uri false.

Există câteva metode care pot fi folosite în cazul hit-urilor false şi miss-urilor false care apar la CAM tag-ul unui cache CAM-RAM [Hung06]. O primă observaţie este faptul că miss-urile false şi hit-urile false pot fi detectate prin replicarea CAM tag-ului. Căutarea de tag se realizează în ambele CAM-uri. Vom avea ca rezultat al căutării atât informaţii despre hit şi miss cât şi despre compararea adreselelor de hit. Un insucces în rezultatele de căutare indică prezenţa unei erori soft într-una din memoriile CAM. Deşi această metodă este simplă ea implică un overhead destul de

mare. O celulă CAM are de câteva ori mai mare mărimea decât o celulă RAM. De exemplu, o celulă CAM ocupă o zonă de aproximativ patru ori mai mare decât o celulă RAM. Deşi replicarea de tag poate detecta erori, este necesar de asemenea şi o tehnică de codificare pentru a determina care din cele două CAM-uri nu este coruptă.

1.2.2. Problema replasării

Identificarea blocului ne specifică modul de găsire al blocului, în cadrul cache-ului. În cazul nostru, cache-ul are o adresă de tag la fiecare structură de blocuri care specifică adresa blocului. Adresa de tag de la fiecare bloc din cache, care ar putea conţine informaţia dorită, este verificată pentru a vedea dacă se potriveşte cu adresa de bloc de la CPU. Ca regulă toate tag-urile posibile sunt căutate în paralel pentru că viteza este critică în acest caz. Trebuie să existe o cale pentru a afla dacă blocul din cache conţine informaţia validă. Cea mai utilizată procedură este adăugarea unui bit de validare la tag, pentru a preciza dacă această intrare conţine sau nu o adresă validă. Dacă bitul nu e setat, nu poate exista o potrivire la această adresă.

În fig. 1.10. se prezintă împărţirea adresei ce provine de la CPU.

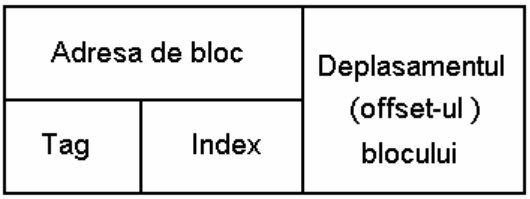


Fig. 1.10.

Prima divizare se produce la împărţirea adresei ce provine de la CPU în: *adresa de bloc* şi *offset-ul* (deplasamentul) blocului. *Adresa de bloc* poate fi divizată mai departe în două câmpuri: *câmpul tag* şi *câmpul index*. Câmpul offset bloc selectează data dorită din cadrul unui bloc, câmpul index selectează setul din care face parte blocul, iar câmpul tag este comparat cu tag-ul de la cache dacă avem un hit în cache. Dacă avem un miss, controler-ul de cache trebuie să selecteze blocul ce conţine data dorită pentru a-l putea replasa. În cazul în care este vorba de o replasare, când avem o mapare directă decizia hard este simplificată, numai un singur cadru de blocuri este verificat la un hit şi acel bloc va fi replasat. Dacă avem o plasare complet asociativă sau set-asociativă sunt mai multe blocuri de ales în

cazul unui miss. Există câţiva algoritmi folosiţi pentru selectarea blocului care urmează a fi înlocuit [NoNo01]:

- algoritmul FIFO (First In First Out);

- algoritmul LFU (Least Frequently Used);

- algoritmul Random;

- algoritmul LRU (Least Recently Used);

Algoritmul FIFO (Primul intrat este primul care iese) precizează că dacă în cadrul setului avem k elemente, totdeauna este ales pentru replasare acel element care a fost accesat cu k accese înainte indiferent de accesele intermediare care au avut loc [HePa03].

Algoritmul LFU (Cel mai puţin frecvent utilizat) precizează că trebuie să avem pentru fiecare bloc al setului un contor şi de fiecare dată când se accesează informaţia dintr-un bloc, contorul acestuia este incrementat, candidatul pentru replasare fiind acel bloc din cadrul setului care are valoarea cea mai mică a contorului.

Cele mai aplicate metode sunt metodele ce folosesc algoritmul random şi algoritmul LRU.

Algoritmul random prevede că din cadrul setului să fie ales pentru replasare un bloc într-un mod aleator din motive de depanare hardware. Nu se merge pe soluţia complet aleatoare ci pe o soluţie aleatoare cu o caracteristică de periodicitate. Prin urmare se va genera o secvenţă de numere pseudoaleatoare, secvenţă ce se repetă cu o anumită periodicitate. Dacă numerele aleatoare sunt generate prin rutine soft datorită faptului că registrele au număr limitat de ranguri, secvenţele generate prin rutine nu se supun unor reguli probabilistice date, ci se supun cu anumite abateri şi ele pot fi caracterizate prin denumirea pseudoaleatoare, în contextul nostru pseudo reprezintă aspectul de periodicitate deja amintit [FaOS03].

Algoritmul LRU (Cel mai puţin recent utilizat) prevede că pentru a reduce şansa pierderii informaţiei, care va fi necesară curând, trebuie ca accesele la blocuri să fie înregistrate. Blocul replasat este acela care a fost neutilizat pentru cea mai mare perioadă de timp. Algoritmul LRU apelează principiul localităţii care precizează că dacă blocurile utilizate recent, vor fi utilizate din nou, atunci cel mai bun candidat pentru replasare este cel mai puţin recent utilizat bloc.

1.2.3. Problema scrierii

Politicile de scriere, ne furnizează informaţii referitoare la strategiile de scriere. Într-un calculator citirile sunt mai frecvente decât scrierile, aceasta pentru că toate instrucţiunile sunt citite şi dintre instrucţiuni nu toate prevăd scrieri. Folosind principiul "Make the common case fast" – favorizarea cazului cel mai frecvent, vom acorda prioritate citirilor faţă de scrieri. Conform legii lui Amdahl [HePa03], pentru o performanţă globală nu trebuie neglijate scrierile. Din fericire cazul cel mai frecvent (al citirilor) este şi cazul cel mai simplu, pentru că concomitent cu citirea şi compararea de tag poate fi întreprinsă în paralel şi citirea de date. În situaţiile de hit datele sunt disponibile imediat, iar în situaţiile de miss nu se beneficiază de datele deja citite.

Scrierile sunt mai complicate pentru că operaţiile nu se pot executa în paralel, mai întâi se face citirea şi compararea de tag şi apoi scrierea în zona de date (datele ar fi alterate, s-ar pierde dacă scrierea şi citirea s-ar face simultan). În plus, scrierea se face doar într-o porţiune a blocului (1-8 B) şi nu în întreg blocul. Aspectele relevate mai sus fac ca scrierii să i se aloce tehnici specifice de implementare. Există astfel două tehnici şi anume: "write through" (numită şi "store through") şi "write back" (numită şi "copy back" sau "store in"). Metoda write through, implică scrierea atât în blocul din cache cât şi în blocul din memoria principală. A doua metodă, write back implementează scrierea doar în blocul din cache. Există unele referiri la deosebirea dintre metodele de scriere prin prisma posibilităţi de implementare a codului prin coduri ciclice. Fiecare din metodele analizate are avantaje şi dezavantaje.

Avantajele metodei write through sunt:

- la write miss nu mai este necesară scrierea blocului replasat în memoria principală pentru că acesta conţine copia cea mai actuală;
- implementare mai simplă;
- în multiprocesare prin existenţa în memoria principală a copiei cea mai actuală a datelor, este favorizată implementarea eficientă a operaţiilor de I/O.

Avantajele metodei write back sunt:

- la scrieri repetate într-un anumit bloc acestea se vor efectua în cache şi numai o singură dată în memoria principală lăsând magistrala liberă pentru mai multe operaţii;

- metoda este mai rapidă, deoarece se lucrează la viteza cache-ului;
- ocupă mai puţin timp magistrala sistemului, rezultând o gâtuire mai mică a acesteia, deci ea se utilizează pentru configuraţii multiprocesor.

Metoda write through revendică, datorită decalajului de viteză dintre CPU şi memoria principală, ca CPU să intre în stare inactivă (write stall). Pentru evitarea îngheţării CPU se utilizează soluţia write buffer existentă la memorie şi în care CPU trebuie să încarce atât adresa de înscris cât şi informaţia, iar CPU îşi reia activitatea în paralel cu desfăşurarea operaţiilor de scriere din write buffer.

Metoda write back implică prevederea fiecărui bloc de informaţii cu un bit suplimentar: clean/dirty. Iniţial este pus pe 0 clean şi dacă s-a făcut o scriere în acest bloc se va pune pe 1 dirty. Dacă urmează o replasare, urmează să fie replasate în memorie doar blocurile cu bitul pus pe dirty.

În cazul write-miss-urilor avem 2 soluţii:

1) write allocate (fetch on write)

2) no write allocate (write around)

În cazul write allocate se va scrie doar în blocul de cache, urmând ca actualizarea să se facă la primul miss de citire. În cazul write around scrierea se face pe lângă cache, doar în memoria principală. De obicei se combină metoda write back cu write allocate în speranţa captării a cât mai multor scrieri în cache aşa cum se combină write through cu write around pentru ca scrierile să fie cuprinse în memoria principală.

1.3. ANALIZA UNOR SOLUŢII EXISTENTE DE CACHE-URI CU CAPABILITĂŢI DE ÎMBUNĂTĂŢIRE A DEPENDABILITĂŢII

1.3.1. Toleranţă la defectare obţinută prin redimensionarea cache-ului

Agarwal et al. în [APMD05] propun o soluţie de reconfigurare a unui cache urmărind obţinerea unei structuri cu înalte capabilităţi de toleranţă la defectare. Autorii fac o analiză a modalităţilor de defectare mai probabile, şi conchid prin a face referire la următoarele categorii de defectare:

- a) creşterea timpului de acces la celula SRAM,
- b) citirea distructivă şi/sau scrierea ratată.

Cauza primordială pentru modurile de defectare, este reprezentată de abaterile parametrilor tranzistorilor din care este sintetizată celula, care provoacă, prin efectul de dopare aleator, fluctuaţii ale tensiunii de prag. Aceasta se constituie, în câte o variabilă aleatoare pentru fiecare din cele şase tranzistoare din care este alcăltuită celula, variabile care se admite că se supun unei legi de distribuţii normale (de tip Gauss).

Este luat în consideraţie un cache cu capacitate de 64 KB cu mapare directă, a cărui schemă bloc este prezentată în fig. 1.11. Se poate remarca împărţirea cuvântului de adresă în trei părţi, şi anume: Tag, Index şi Byte offset. Matricea de memorare este împărţită în două, una destinată tag-ului şi cealaltă destinată datelor.

Dimensiunea blocurilor de date este de 128 B, blocurile fiind alcătuite din 4 cuvinte, fiecare având 32 B. Câmpul index, prezintă un număr de 11 biţi, dintre care 9 permit selecţia unui bloc din cele 512 ale matricii, restul de 2 biţi fiind utilizaţi pentru selecţia cuvântului din cadrul unui bloc. Câmpul byte offset, alcătuit din 5 biţi, permite selecţia unui byte, din cele 32 ale cuvântului. Ceilalţi 16 biţi ai cuvântului de adresă din totalul de 32, sunt destinaţi câmpului tag.

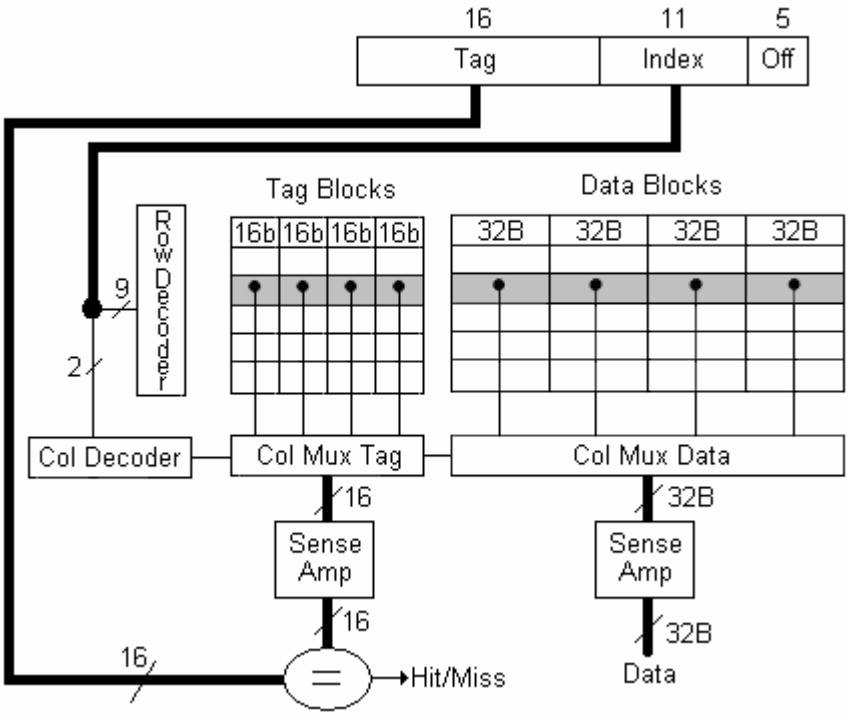


Fig. 1.11.

Selecţia unui bloc de tag se face concomitent cu cea a unui bloc de date prin decodificarea, cu Row Decoder, a unuia din cele 2^9 = 512 rânduri ale matricilor de tag, respectiv date. Biţi rămaşi, în număr de 2, sunt aplicaţi decodificatorului de coloană (Col Decoder) ale cărui ieşiri comandă câte un multiplexor, Col Mux Tag pentru partea de tag, respectiv Col Mux Data pentru partea de date.

Intrările celor două multiplexoare sunt reprezentate de cele 4 câmpuri, de câte 16 biţi la partea de tag, repectiv de câte 32 B la partea de date, iar ieşirile multiplexoarelor sunt conectate la amplificatoare (Sense Amp), menite a direcţiona informaţia şi a restaura nivelurile logice. Ieşirile amplificatoarelor sunt comparate bit cu bit cu câmpul tag al cuvântului de adresă şi, în caz de egalitate, se generează un semnal de hit, inegalitatea, chiar şi la nivelul numai a uneia din perechile de biţi provocând generarea unui semnal de miss. La o operaţie de citire, activarea semnalului de hit determină trimiterea porţiunii de date, corespunzător amplificate prin circuite driver, spre CPU, iar, la o operaţie de scriere, stocarea informaţiei provenite de la CPU, în locul blocului de date selectat. Pe de altă parte, activarea semnalului de miss determină direcţionarea accesului, ratat în această treaptă de cache, înspre o treaptă următoare, inferioară, a ierarhiei de memorii.

Urmărind acoperirea tipurilor de defecte amintite şi plecând de la structura generală descrisă, autorii [APMD05] configurează un cache cu capabilităţi de tolerare la defectare a cărei anatomie este prezentată în fig.1.12.

Elementul de structură esenţial, prin prisma atingerii dezideratului de tolerare la defectare, este constituit de aşa numitul bloc BIST, menit să testeze întregul cache, detectând malfuncţionarea până la nivelul celulei nefuncţionale. În [APMD05] nu se detaliază acest bloc, făcându-se afirmaţia că poate fi utilizată orice configuraţie BIST convenţională cu trimitere la [NHKK99], [TeNF01]. În ambele lucrări stimularea memoriei se realizează prin intermediul unui generator algoritmic de modele de biţi (Bit Pattern Generator), care au la baza construcţiei algoritmi ad-hoc convenţionali, utilizaţi în general, la testarea memoriilor semiconductoare [GoAl00]. Se cunoaşte însă, faptul că, chiar şi cei mai simpli algoritmi sunt prohibitivi, prin prisma parametrului critic reprezentat de timpul de testare. În consecinţă schemele BIST propuse, prezintă eficienţă doar în condiţiile unei testări efectuată off-line.

Am insistat asupra acestui aspect, întrucât el este esenţial prin prisma soluţiei propuse de autorul prezentei lucrări, a cărei esenţială caracteristică este constituită de utilizarea on-line a capabilităţilor de self-testare şi tolerare la defecte.

Analizând în continuare soluţia din [APMD05], mai menţionăm că numărul celulelor defecte şi locurile unde acestea se manifestă prezintă caracteristica de a se modifica, în dependenţă de condiţiile de operare reprezentate de valorile modificabile, în mod aleator, ale tensiunii de alimentare, frecvenţei, ş.a. Fiecare bloc

de date este testat prin intermediul circuistici BIST atunci când este activat semnalul Test Mod. Informaţia despre locaţiile defecte este aplicată unui aşa numit configurator, reprezentând aportul adus de lucrarea [APMD05], bloc care decide dacă un chip poate funcţiona într-o anumită condiţie de operare şi care generează un semnal Faulty/Non faulty. Informaţia despre defect furnizată de către configurator, este stocată într-o memorie Config Storage, de unde este salvată pe hard disk de fiecare dată când procesorul este deconectat.

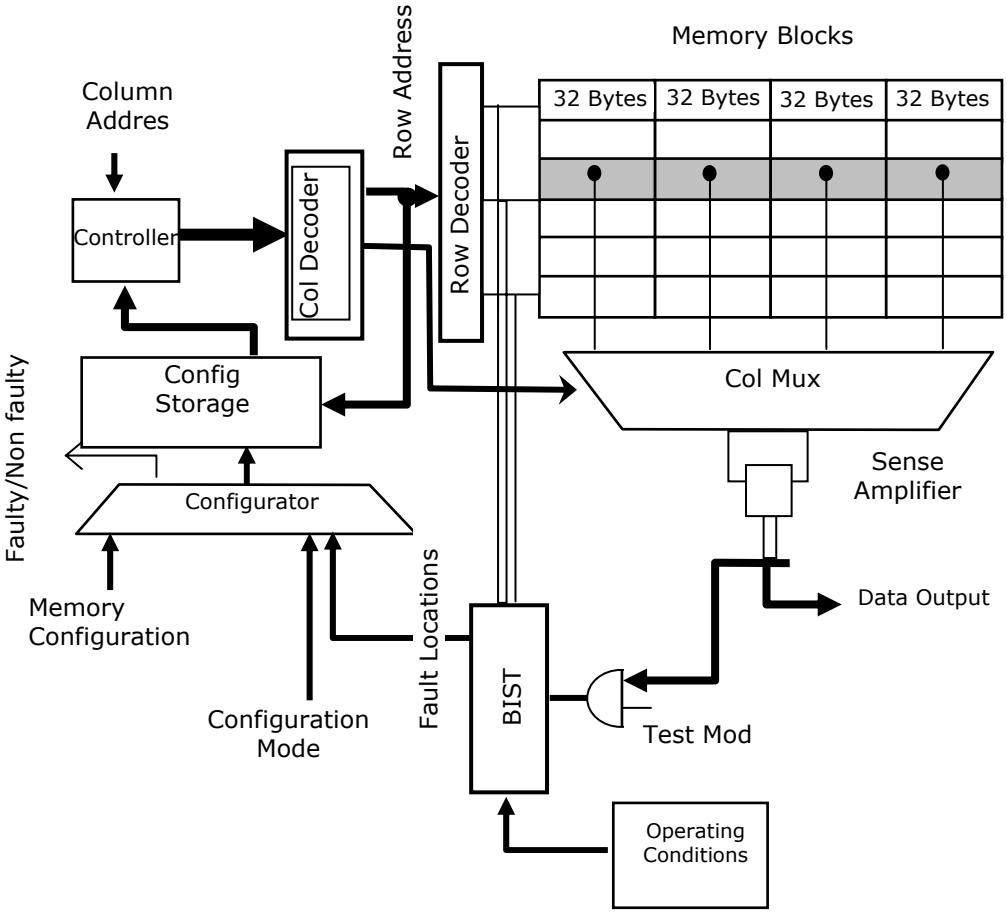


Fig. 1.12.

Ideea fundamentală a arhitecturii propuse, pe lângă elementele de structură, Configurator şi Config Storage, constă în intervenţia asupra multiplexoarelor de coloană (Col Mux Tag şi Col Mux Data) astfel încât, atunci când este accesat un cuvânt defect, să fie selectat un alt cuvânt din acelaşi rând. Atunci când sistemul este repornit, procesorul reîncarcă vechea informaţie despre defect în Config Storage. Dacă condiţiile de operare au fost modificate, este activat semnalul

Test Mod şi, prin intermediul BIST, în Config Storage este stocată noua informaţie despre potenţialele noi defecte. Aceasta este folosită pentru a redimensiona cache-ul prin utilizarea în locul cuvintelor defecte a altor cuvinte, astfel încât, cuvintele defecte să fie evitate pe parcursul operării normale a cache-ului.

Pentru a pune în relief penalităţile de performanţă care privesc strict funcţionarea cache-ului din momentul boot-ării sistemului, în fig.1.13.a) am înlănţuit timpii de acces la un sistem convenţional fără accesoriile propuse de Agarwal et al., iar în fig. 1.13.b) am prezentat, prin t_1, timpul necesar încărcării de pe hard disk în Config Storage, a informaţiei despre defect, prin t_2, timpul necesar tatonării faptului dacă condiţiile de operare s-au modificat faţă de anterioara boot-are a sistemului, iar, prin t_3, timpul necesar testării, prin BIST, a întregului cache în posibila situaţie că au fost modificate condiţiile de operare.

Desigur, t_1 şi t_2 se pot suprapune dar, ceea ce nu rezultă suficient de clar din fig. 1.13. este raportul dintre aceste intervale de timp, întrucât t_3 este cu ordine de mărime mai mare decât celelate, putând să ajungă, dependent de capacitatea cache-ului, de ordinul orelor sau mai mult chiar şi pentru teste ad-hoc moderate în complexitate. Agarwal et al. califică arhitectura lor ca fiind o soluţie tehnică necesitând un adaos minim de energie, precum şi neafectând prin vreo penalitate orice timp de acces (with minimum energy overhead and does not impose any cache acces time penalty [APMD05]).

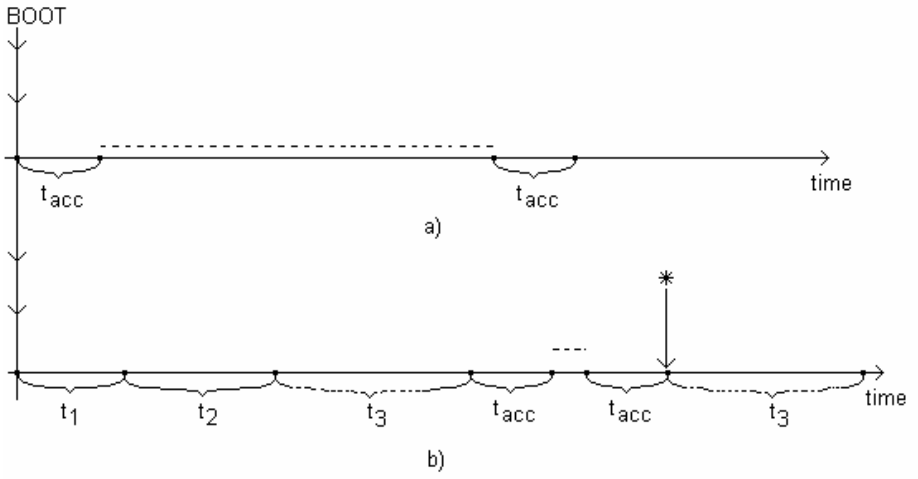


Fig.1.13.

Cele afirmate sunt corecte cu amendamentul, însă, că raportat la tipurile de defecte parametrice admise ca posibile, condiţiile de operare variabile pot provoca

respectivele defecte pe parcursul funcţionării normale, situaţie marcată cu * în fig. 1.13.b). Modificarea condiţiilor de operare ar trebuie să declanşeze, prin Test Mod, activarea schemei BIST cu concomitenta intrare într-un ciclu t_3 care va avea, drept consecinţă, o degradare drastică a performanţei accesării cache-ului. Neluarea în consideraţie a relevatei posibilităţi apreciem că reprezintă o omisiune a soluţiei propuse, evitată printr-o arhitectură BIST on-line de tipul celei prezentate de autor în cele ce urmează în prezenta teză.

Revenind la soluţia din [APMD05], în fig. 1.14. se arată cum controller-ul local, consultând dicţionarul de defecte din Config Storage, a obţinut informaţia constând din faptul că este defect cuvântul decodificat prin ”00” şi determină forţarea Col Mux-ului să selecteze alt cuvânt din acelaşi bloc, anume cel decodificat prin ”01”, alterând adresa de coloană.

Selecţia cuvântului dintr-un set de unele funcţional corecte este predefinită. În mod similar din partea de tag a cache-ului este selectat tag-ul corespunzător noului cuvânt (”01”). În felul acesta fiecare cuvânt defect din cache este replasat printr-unul funcţional corect, modalitate prin care, în mod evident, capacitatea cache-ului descreşte. Are loc, prin urmare o redimensionare (resizing) a dimensiunii cache-ului cu o granularitate la nivel de cuvânt.

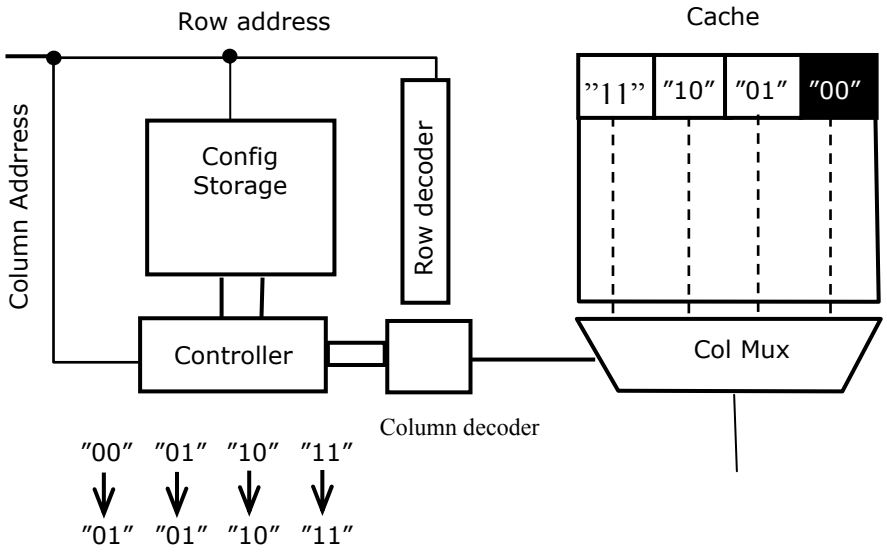


Fig. 1.14.

Una din problemele importante ale soluţiei din [APMD05] constă în accesarea de aşa manieră de către procesor a cache-ului redimensionat încât, să fie utilizată aceiaşi adresă la accesarea acestuia ca şi la o arhitectură convenţională. În acest sens, autorii apelează la o schemă de mapare ingenioasă a cărei introducere

se face prin exemplul dat în fig. 1.15. a). În acest sens se consideră două adrese de memorie care accesează două cuvinte diferite din acelaşi bloc.

Ambele vor avea aceeaşi adresă de rând, dar adresă de coloană diferită. Să presupunem că cele două adrese de cuvinte de date au asociate câmpuri de tag identice. Admiţând acum că adresa notată cu ”one” este defectă (FAULTY), în baza celor anterior prezentate, controller-ul va forţa Col Mux-ul să selecteze cuvântul corespunzător adresei ”two”. Ambele adrese sunt compuse dintr-un câmp de tag notat cu T, dintr-un câmp de index decodificat având concatenată la adresa de rând decodificată cei doi biţi (00 sau 01) ai adresei de coloană, precum şi din câmpul de offset notat cu Off. O acţiune de citire de la adresa de index R 00, adresă considerată defectă.

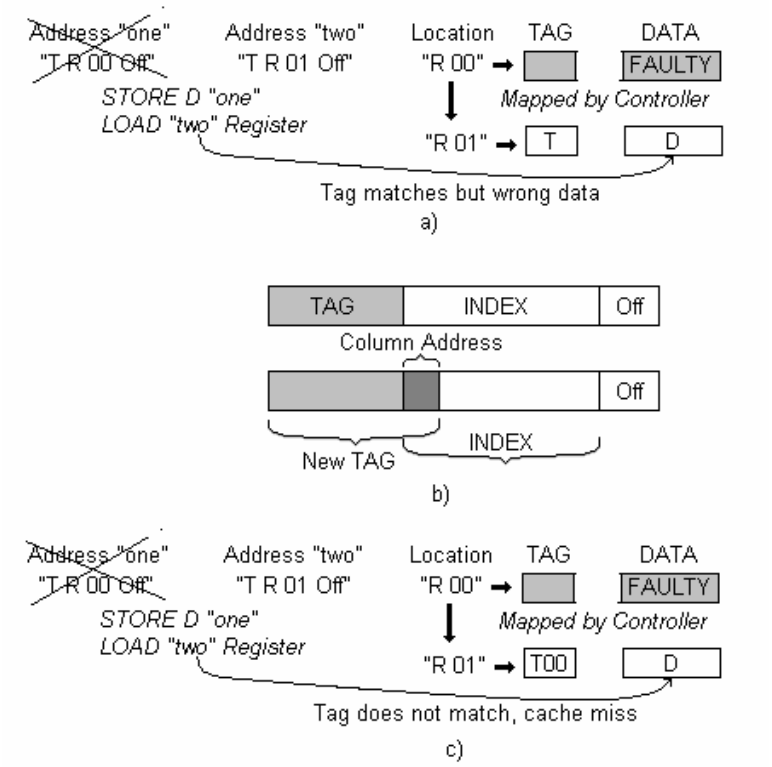


Fig. 1.15.

. Se desfăşoară în două trepte date de secvenţa de cod STORE D ”one” (prin care data D este scrisă, împreună cu tag-ul corespondent T, la locaţia ”two”), succedată de LOAD ”two” Register (prin care este citită informaţia de la adresa ”two”). Ambele adrese având acelaşi tag la micro-operaţia de comparaţie a acestuia, rezultând un cache hit, este citită o dată incorectă (wrong data) întrucât este accesată de fapt data de la adresa” one” şi nu data aşteptată de la adresa ”two”

care ar corespunde unei arhitecturi convenţionale de cache. În scopul evitării situaţiei nedorite relevate, autorii propun extinderea câmpului de tag, în cazul particular acceptat cu doi biţi peste câmpul de index. În general, extensia se face cu liniile de adresă de coloană, aşa cum se prezintă în figura 1.15.b).

Soluţia este asemănătoare cu una găsită în [HePa03], cu singura menţiune că în această referinţă suprapunerea apărea între câmpul de offset şi cel de index. Prin intermediul acestei soluţii, aşa cum se prezintă în figura 1.15.c) se soluţionează problema mapării în sensul că anterioara operaţie de STORE va memora în câmpul de tag informaţia extinsă T00 şi astfel succesivul acces de tip acces LOAD la adresa ”two”´compară noul tag, care este diferit pentru adresele ”one” şi ”two”, generând un cache miss cu citirea, în consecinţă, datei din nivelul inferior (fie următorul de cache, fie de memorie principală). Această nouă tehnică de mapare, care permite procesorului să acceseze cache-ul fară a fi sesizat că acesta a fost redimensionat, se bazeză deci, pe un adaos hardware şi o creştere relativ redusă a timpului de acces la cache dar, aceasta datorită creşterii numărului de intrări a circuitului de detecţie a stării de hit/miss.

Capacitatea de tolerare a defectelor pentru arhitectura propusă depinde de numărul cuvintelor dintr-un bloc. Controller-ul local este de aşa mamieră proiectat încât poate altera adresa de coloană pentru a selecta orice cuvânt dintr-un bloc. Schema funcţionează corect atâta timp cât există un cuvânt funcţional corect nonfaulty într-un bloc, modalitate prin care cache-ul poate fi redus până la 1/N din capacitatea pentru care a fost proiectat, unde N reprezintă numărul de cuvinte/bloc (facem observaţia că am utilizat, orientându-ne după referinţa de largă circulaţie şi accepţiune [HePa03], termenul de bloc alcătuit din mai multe cuvinte, ceea ce în [APMD05] este reprezentat de un rând alcătuit din mai multe blocuri). Se poate conchide că soluţia propusă în [APMD05] pentru problematica mapării implică, pe lângă adaosul de circuite necesar comparării câmpului extins de tag, şi o degradare, redusă, dar nenulă a timpului de acces la cache.

O altă parte importantă a soluţiei propuse în [APMD05] este constituită din memoria Config Storage, în care se stochează, aşa cum am vazut, informaţia despre defecte şi care este citită la fiecare acces de citire sau scriere în cache. Menţionăm însă că soluţia tehnică pentru acest element de structură trebuie să fie suficient de rapidă pentru ca informaţia despre defect să fie furnizată controller-ului înainte ca data să ajungă la Col Mux. Este motivul pentru care autorii propun soluţii costisitoare în ambele variante de implementare, atât prin intermediul unei memorii asociative, adresabile prin conţinut (content addressable memory-CAM), cât şi prin intermediul implementării one-bit (one-bit implementation-OBI). Fără a insista asupra detaliilor a nici uneia dintre cele două soluţii arătăm totuşi că, ele trebuie totuşi să răspundă la rezolvarea cât mai bună a compromisului excedent de energie

consumată (în ultimă instanţă, surplus de circuite) / capabilitate rezonabilă de tolerare la defectare (în ultimă instanţă, număr de defecte stocate). Astfel arătăm doar că, la versiunea CAM, sunt stocaţi biţii câmpului de index, capacitatea CAM-ului depinzând de numărul de cuvinte defecte, acceptat de autor la 100. CAM-ul memorează de asemenea adresa de coloană a cuvântului de adresă funcţional corect, unde trebuie memorată informaţia de la locaţia defectă în cazul în care aceasta din urmă este accesată. Pe de altă parte, la versiunea OBI, pentru fiecare cuvânt din cache este prevăzut un bit în memoria OBI, bit care are valoarea 1 în cazul în care cuvântul din cache este defect şi 0 în caz contar. Aceşti biţi sunt determinaţi de către BIST la momentul testării, evident off-line, şi sunt stocaţi în memoria OBI prin intermediul configuratorului. Arhitectura acestei memorii auxiliare este asemănătoare cu cea a cache-ului având blocuri alcătuite din câte 16 cuvinte, fiecare cuvânt fiind de 4 biţi, câte unul pentru fiecare dintre cuvintele de 32 B din cache (fig. 1.11.). Prin urmare, rezultă că şi această soluţie, ca şi cea bazată pe CAM, este mare consumatoare de circuite iar proiectarea ei astfel încât, să îndeplinească dezideratul de a răspunde mai rapid decît accesarea cache-ului, se constituie într-o activitate deosebit de pretenţioasă.

Concluziv, lucrarea [APMD05] dă o soluţie problemei proiectării unui cache cu capabilităţi de tolerare la defectare, dar se rezumă la acoperirea unei palete restrânse de defecte provocate de efectul dopant aleatoriu, dând o soluţie apreciată de noi cu carenţe prin prisma necesităţii activării on-line a schemei BIST în vederea stocării cuvintelor defecte în Config Storage, la care se adaugă şi alte penalităţi de performanţă şi cost relevate în cele de mai sus.

Încadrăm în clasa de soluţii asemănătoare cu [APMD05] şi cea din [Hori97], care se bazează pe matrici de cache cu rânduri şi coloane redundante care, prezentată rezumativ, se bazează pe înlocuirea rândurilor şi coloanelor defecte cu unele din cele redundante, cu funcţionare corectă, aflate în rezervă. Aici nu avem de a face cu o redimensionare a elementelor cache-ului excluzând din configuraţie elementele defecte, ci cu o replasare a acestora bazată pe tehnici de detecţie a defectelor asemănătoare. Această soluţie prezintă limitări prin numărul de rânduri şi coloane care pot fi considerate defecte şi, de asemenea, pot conduce la penalităţi de performanţă.

1.4. CONCLUZII

În acest capitol am prezentat stadiul actual al cunoaşterii în domeniul cache-urilor, locul treptei cache într-o ierarhie de memorii, arhitectura generală a cache-urilor, configuraţii şi exemple de cache-uri, problemele cache-urilor şi perturbarea acestora prin defectare.

În acest context am tratat: problema mapării, a replasării şi a scrierii în cache. În cadrul problemei mapării am prezentat în detaliu modul în care pot fi mapate adresele din memoria principală în cache pentru toate cele trei situaţii, ale mapării directe, set-asociative şi complet asociative. Tot în cadrul problemei mapării sunt prezentate şi malfuncţionalităţile determinate de hit-urile false şi miss-urile false şi câteva metode ce pot fi folosite pentru detecţia lor. În cadrul problemei replasării am prezentat cei patru algoritmi cei mai utilizaţi pentru selecţia blocului şi anume: FIFO, LFU, Random şi LRU. Am prezentat de asemenea şi aspecte ale politicilor de scriere prezentând avantajele celor două tehnici de scriere write through şi write back.

Am analizat soluţii existente de cache-uri cu capabilităţi de îmbunătăţire a dependabilităţii, unde se prezintă toleranţa la defectare prin redimensionarea cache-ului. Soluţia prezentată de [APMD05] se rezumă la acoperirea doar a unei palete restrânse de defecte, fiind o soluţie nesatisfăcătoare deoarece trebuie activată on-line schema BIST introducând penalităţi de performanţă şi cost.

2. ANALIZA DEFECTELOR CARE POT AFECTA CACHE-URILE

2.1. DEFECTE LA NIVELUL CELULEI SRAM

Paleta de defecte care pot afecta memoriile în general, şi cache-urile în particular, este foarte mare. Cum este exclus a elabora o strategie de testare destinată fiecărui mod de defectare, s-a căutat generarea unor modele de defectare, acoperitoare pentru cât mai multe dintre defectele potenţiale [GAMC02]. Restrângând pe cât posibil numărul modelelor de defectare, s-a reuşit construcţia unor echipamente de testare automată rezonabile totuşi ca investiţie, întrucât tehnicile de testare implementate pe acestea ţinteau doar detecţia şi diagnosticarea defectelor modelate. Ulterior, pe măsură ce s-a impus abandonarea tot mai accentuată la complexele echipamente de testare automată, metodele de verificare „glisând" sub formă de facilităţi de testare încorporată (built-in self-testing facilities) în interiorul subansamblelor, chiar în interiorul chip-urilor problematica restrângerii modelelor de defectare a devenit din ce în ce mai acută. În acest context complex, arătăm că literatura [NiOR97], [NHKK99], [OmRM03], [MeFR97], este bogată în încercări de modelare a defectelor din memorii principale, hard disc-uri, diferite tipuri de memorii pe suport magnetic, dar este săracă în ceea ce priveşte treapta unei ierarhii de memorare care constituie ţinta investigaţiilor din această teză, anume treapta cache. În căutarea individualizării problematicii de testare pentru cache-uri plecând de la suportul tehnologic distinct al acestor memorii, prezentul paragraf îşi propune analiza modurilor de defectare la nivelul cel mai intim al celulelor de memorare.

Astfel, vom demara cu o taxonomie mai generală care împarte modurile de defectare, după felul de manifestare în timp, în următoarele categorii:

- defecte permanente, care sunt definitive;

- defecte tranzitorii, care au o durată temporară scurtă;

- defecte intermitente, care sunt asemănătoare cu cele tranzitorii, dar ele apar şi dispar în mod repetat în timp, cu o comportare periodică [GAMC02].

Defectele permanente constau din modificări fizice ireversibile produse în componentele de memorare. Acestea se pot produce pe parcursul procesului de fabricaţie sau pe parcursul funcţionării normale în exploatare. Procesul de îmbătrânire al componentelor poate duce, pe termen lung, la apariţia iniţială a unor defecte intermitente, care se transformă, în timp, în unele permanente. Defectele de fabricaţie pot, de asemenea, produce malfuncţionalităţi care se manifestă pe parcursul funcţionării normale a componentelor memoriilor.

Defectele permanente au drept cauză, în marea lor parte, fenomene fizice care au loc la nivelul tranzistoarelor integrate. Prin prisma efectului care îl produc, din punct de vedere logic, putem considera următoarele modele de defectare caracteristice, de altfel, întregii familii de circuite integrate, nu doar chip-urilor de memorie:

a) Blocat - pe 0 sau 1 logic, (Stuck-at 0 or 1, s-a-0 or s-a-1)

b) Blocat pe stare deschisă (Stuck-open);

c) Linie deschisă (Open-line) în liniile de conectare a circuitelor logice;

d) Scurtcircuit (Short), între liniile de conectare a circuitelor logice;

Modelul de blocare_pe (0,1) este cel mai utilizat. Modelul blocat pe stare deschisă (Stuck-open), se aplică defectelor specifice porţilor realizate în tehnologie CMOS [ALRL04].

Există un al doilea grup în care avem câteva defecte care afectează întârzierea de comutare a tranzistoarelor MOS şi întârzierea de încărcare / descărcare a capacităţilor parazite pe conexiunile de intare / ieşire. Efectul acestor defecte la nivelul logic este modificarea permanentă a întârzierilor circuitului, astfel încât modelul de defect care se aplică aici este modelul de întârziere [GAMC02].

Defecte intermitente apar datorită unor cauze asemănătoare cu cele permanente, dar în cele mai multe situaţii apar datorită fenomenului de îmbătrânire (uzură). Din acest motiv modelele de defecte care se aplică acestor defecte intermitente sunt aceleaşi modele ca şi la defectele permanente [GAMC02].

Aceste defecte, mai poartă şi denumirea de erori soft (single event upsets (SEUs)) şi apar în timpul funcţionării circuitului fie datorită unor cauze interne, fie datorită unor cauze externe. Spre deosebire de defectele permanente, defectele tranzitorii nu introduc defecte fizice în circuit.

Studiul acestor defecte este dificil, deoarece ele nu pot fi localizate spaţial şi durata lor este scurtă. De asemenea acestea nu au un model bine definit de tratare

a defectelor, datorită diferitelor fenomene interne sau externe ce stau la baza generării lor [GAMC02].

Defectele funcţionale ale memoriilor (Functional Memory Faults) pot fi definite ca şi deviaţii de comportament ale memoriei ce se află sub observaţie faţă de un comportament specificat, în condiţiile aplicării unui set de operaţii ce trebuie executat [GoAl00].

Orice model funcţional de defecte (Functional Fault Model - FFM) este caracterizat prin două liste: o listă a operaţiilor ce trebuie executate de către memorie, numită şi secvenţă de operaţii, şi o listă corespunzătoare a deviaţiilor în comportarea observată faţă de comportarea aşteptată. Definim o secvenţă de stimulare (SS) o succesiune de operaţii menite a pune în relief diferenţa de comportament funcţional dintre răspunsul real obţinut şi cel aşteptat.

Operaţiile ce se pot efectua asupra unei memorii sunt: citirea notată cu r şi scrierea notată cu w. Pentru a putea accesa informaţia dintr-o celulă de memorie, trebuie să specificăm adresa celulei. De asemenea, dacă avem o operaţie de scriere trebuie se specificăm şi data ce urmează să fie scrisă în memorie. Mai putem menţiona în plus că o comportarea defectuoasă a memoriei este de asemenea afectată de către valoarea logică iniţială a celulei. Ţinând cont de cele mai sus menţionate, orice SS se poate reprezenta cu ajutorul notaţiei [GoAl00]:

$$a(viOPd)_1\ a(viOPd)_2\ \ldots\ a(viOPd)_k\ \ldots\ a(viOPd)_n$$

unde:

a - este adresa celulei utilizată de operaţia k,

vi - este valoarea iniţială memorată în celulă, vi $\in$ {0,1},

OP - reprezintă tipul operaţiei efectuate asupra celulei a, OP $\in$ {w,r},

d - este data ce se citeşte sau ce urmează să fie scrisă în celula cu adresa a, d$\in$ {0,1}.

Orice diferenţă între comportarea observată şi comportarea aşteptată a memoriei se poate face cu ajutorul notaţiei : <SS/M/O>

unde: - M este o mărime ce descrie starea memorată în celula defectă, M$\in$ {0,1},

- O descrie ieşirea logică a operaţiei de citire, O$\in$ {0,1,-}.
- O = - se utilizează dacă avem o operaţie de scriere şi nu o operaţie de citire, această operaţie punând în evidenţă defectul.

Multitudinea defectelor care pot să apară la nivelul tranzistoarelor integrate şi a interconexiunilor dintre acestea se impun modelate, scop în care preconizez următoarea taxonomie a modurilor de defectare prin analogie cu astfel de intreprinderi din domeniul memoriilor semiconductoare [Goor98]. Clasificăm mai

întâi în clasa modurilor de defectare statice, respectiv dinamice. Criteriul care stă la baza acestei divizări îl constituie numărul de operaţii aplicate celulei de memorare. Astfel, pentru modurile de defectare statice considerăm cel mult o operaţie de scriere sau citire iar pentru cele dinamice cel puţin două operaţii de scriere sau citire. Luând în consideraţie specificul celulei SRAM utilizată în cache-uri faţă de celula DRAM specifică memoriilor principale, preconizez clasificarea diferită faţă de cea din [GoAl00]. în defecte specifice citirii, scrierii, de blocare şi de stare.

Fundamental, vom pleca de la împărţirea defectelor în unele care afectează celula cache SRAM, notate cu A în fig. 2.1., respectiv care afectează două sau mai multe celule, notate cu B în fig. 2.3. În baza celor expuse propunem următoarea clasificare a defectelor care afectează celula SRAM.

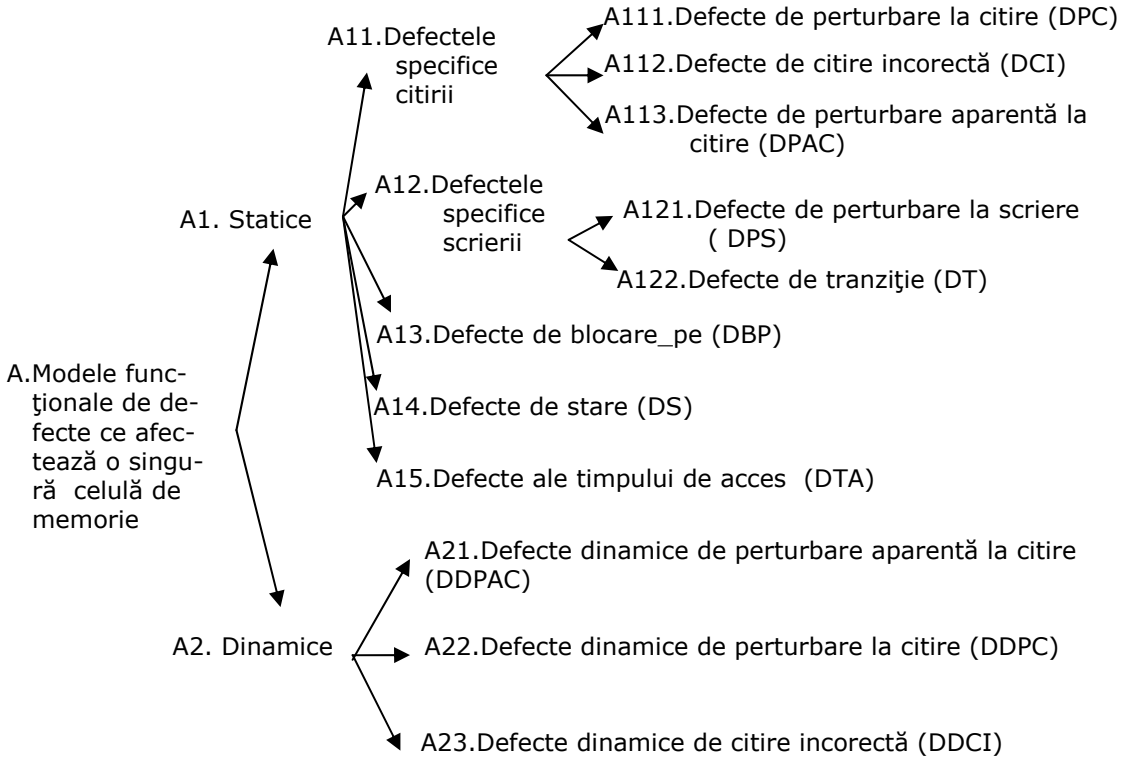


Fig.2.1.

A. Modele funcţionale de defecte ce afectează o singură celulă de memorie (Single-cell FFMs)

Acestea descriu o comportare incorectă, fie a unei date memorate într-o celulă, fie a unei operaţii ce se efectuează asupra ei. Ele se clasifică, în funcţie de numărul operaţii efectuate asupra celulei, în: statice şi dinamice.

***A.1. Modele funcţionale statice* de defecte ce afectează o singură celulă de memorie [GoAl00]**

Aceste modele, descriu defecte caracterizate prin efectuarea a cel mult unei operaţii asupra celulei defecte. Preconizăm clasificarea din fig. 2.1 a modelelor funcţionale de defectare ce afectează celula de memorare SRAM.

Referindu-ne, pentru început, la prima clasă, prezentăm în fig. 2.2 celula SRAM de bază.

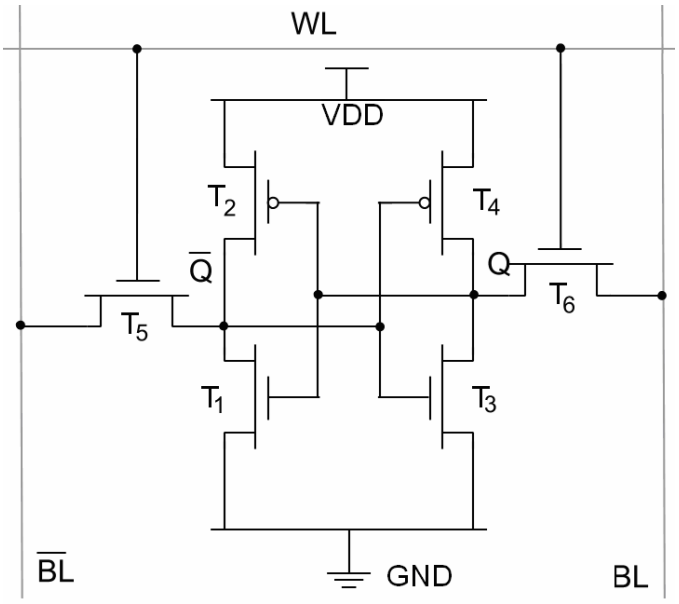


Fig. 2.2.

Celula constă din două inversoare CMOS conectate în cruce cuprinzând tranzistoarele de la T_1 la T_4 şi două tranzistoare de acces de tip N, T_5 şi T_6. Acestea din urmă sunt conectate la liniile de bit (bit line $\overline{BL}$, respectiv BL). Mai sunt puse în relief perechea de noduri $\overline{Q}$ respectiv Q, pe care se manifestă valorile logice complementare 0 şi 1. Imperfecţiunile mijloacelor optice de reproducere a măştilor folosite în tehnologia de fabricaţie a cache-urilor, precum şi densitatea din ce în ce

mai mare de împachetare a dispozitivelor semiconductoare face ca tranzistoarele care alcătuiesc celula să prezinte abateri dimensionale unele faţă de altele. Acestea conduc la funcţionarea eronată a celulei în ceea ce priveşte operaţiile de citire şi scriere pecum şi variaţii inadmisibile ale timpului de accces.

În ceea ce priveşte defectele de citire, în concordanţă şi cu [Hung06], ele constau în bascularea nedorită a celulei de memorie atunci când asupra celulei SRAM se efectuează o operaţie de citire. Uzual, tensiunea din nodul $\overline{Q}$, care în fig. 2.2. stochează 0, creşte de la 0 la o anumită valoare corespunzătoare citirii prin intermediul divizorului de tensiune dintre linia de bit BL, preîncărcată la VDD şi masă, GND, prin intermediul tranzistoarelor T_1 şi T_5. Dacă tensiunea de citire este mai mare decât excursia logică a tensiunii de la ieşirea inversorului T_2, T_4, atunci celula constă într-o citire defectă. Variaţiile tensiunilor de prag, ale tranzistoarelor T_1 şi T_2 (sau T_3 şi T_4) conduc la variaţii ale tensiunii de divizare şi implicit ale excursiei logice.

A.11. Defectele statice specifice citirii:

A.111. Defecte de perturbare la citire (DPCx) [AdCo96]. O celulă are un astfel de defect, dacă o operaţie de citire ce se efectuează asupra celulei modifică data din celulă şi returnează o valoare incorectă la ieşire.

A.112. Defecte de citire incorectă (DCIx). O celulă are un astfel de defect, dacă o operaţie de citire efectuată asupra celulei returnează o valoare logică incorectă, chiar dacă în celulă este menţinută data corectă.

A.113. Defecte de perturbare aparentă la citire (DPACx) [AdCo96]. O celulă are un astfel de defect, dacă o operaţie de citire ce se efectuează asupra celulei returnează valoarea logică corectă, dar acest defect are ca rezultat modificarea conţinutului celulei.

Referitor la defectele de scriere, în concordanţă şi cu [Hung06], ele constau în imposibilitatea de a efectua cu succes operaţia de scriere asupra celulei de memorie. Dacă efectuăm o scriere a valorii 0 în punctul Q, care conţinea valoarea iniţial 1, tensiunea în acest punct V_Q ajunge la valoarea tensiunii de scriere datorită divizorului de tensiune dintre BL şi VDD, la GND, prin tranzistoarele T_4 şi T_6. Dacă tensiunea de scriere este mai mare decât excursia logică a tensiunii de la ieşirea inversorului T_1, T_2, atunci se va efectua o scriere incorectă. De asemenea, deoarece tranzistoarele T_4 şi T_6 (respectiv T_2 şi T_5) sunt de obicei cele cu dimensiunea cea mai mică din celula SRAM, variaţii ale tensiunii de prag, în aceste tranzistoare, pot duce la scrieri incorecte.

A.12. Defecte specifice scrierii:

A.121. Defecte de perturbare la scriere (DPSx). O celulă are un astfel de defect, dacă o operaţie de scriere de tipul (0w0 sau 1w1) determină o tranziţie în celulă.

A.122. Defecte de tranziţie (DTx). O celulă are acest tip de defect, dacă are un eşec la operaţia de tranziţie (0→1 sau 1→0) în momentul când asupra ei se efectuează o operaţie de scriere.

A.13. Defecte de blocare_pe (DBPx). O celulă are acest defect, dacă rămâne întotdeauna blocată pe o anumită valoare, pentru toate operaţiile ce se efectuează asupra sa. Există două tipuri de defecte de blocare_pe: DBP0 = {<∀/0/->}, şi DBP1 = {<∀/1/->}.

A.14. Defecte de stare (DSx). O celulă are acest defect, dacă valoarea logică a celulei se modifică înainte de a fi accesată, chiar dacă nu s-a efectuat nici o operaţie asupra ei. Există două tipuri de defecte de stare şi anume DS0 = {<0/1/->} şi DS1 = {<1/0/->}.

Caracterul ∀, ne precizează faptul că valoarea celulei ramâne aceeaşi pentru toate operaţiile efectuate asupra ei. În consecinţă, SS = ∀ poate fi înlocuit doar de acele operaţii care pun în evidenţă defectul. Acest lucru duce la definiţii echivalente pentru defectul de bocare_pe: DBP0 = {<1/0/->,<0w1/0/->,<1w1/0/->} = DS1 U DT↑ U DPS1, şi DBP1 = {<0/1/->,<1w0/1/->,<0w0/1/->} = DS0 U DT↓ U DPS0. Semnul U este semnul matematic obişnuit de reuniune.

A.15. Defecte datorate timpului de acces (DTA) [Hung06]

Aşa cum se poate observa în figura 2.2., DTA reprezintă intervalul de timp necesar pentru a obţine o diferenţă de tensiune predefinită între liniile de bit BL şi $\overline{BL}$. Când în punctul $\overline{Q}$ avem 0, linia de bit $\overline{BL}$, se va descărca prin tranzistoarele T_5 şi T_1 la o operaţie de citire. Viteza de descărcare depinde de viteza de descărcare a tranzistoarelor T_5 şi T_1. Variaţii în nivelul tensiunii de prag al acestor tranzitoare determină o dispersie a timpului de acces. Se consideră că accesul este necorespunzător în cazul în care timpul de access este mai mare decât o valoare limită maxim tolerată.

A.2. Modele funcţionale dinamice de defecte ce afectează o singură celulă de memorie (Dynamic single-cell FFMs) *[GoAl00]*

Aceste modele, descriu defecte care efectuează mai multe operaţii asupra celulei defecte (adică defecte care respectă condiţia, #OP > 1, unde simbolul #OP reprezintă numărul de operaţii), astfel încât avem un număr infinit de modele. Noi vom limita studiul la cazul în care avem #OP = 2.

În continuare, avem o clasificare a acestor modele funcţionale dinamice de defecte, toate acestea fiind defecte specifice citirii.

A.21. Defecte dinamice:

A.22. Defecte dinamice de perturbare aparentă la citire (DDPACxy).

Acesta este un defect prin care orice secvenţă de tipul xwyry, dintr-o secvenţă de stimulare returnează valoarea logică corectă y, dar acest defect are ca rezultat modificarea conţinutului celulei. Cele patru tipuri de defecte dinamice de perturbare aparentă la citire, sunt: DDPAC00 = {<0w0r0/1/0>}, DDPAC11 = {<1w1r1/0/1>}, DDPAC01 = {<0w1r1/0/1>}, şi DDPAC10 = {<1w0r0/1/0>}.

A.23. Defecte dinamice de perturbare la citire (DDPCxy).

Acesta este un defect prin care orice secvenţă de tipul xwyry, dintr-o secvenţă de stimulare, schimbă valoarea logică stocată a unei celule în valoarea y şi ne returnează o valoare incorectă la ieşire. Există patru tipuri de defecte dinamice de perturbare la citire, pe care le prezint în continuare: DDPC00 = {<0w0r0/1/1>}, DDPC11 = {<1w1r1/0/0>}, DDPC01 = {<0w1r1/0/0>} şi DDPC10 = {<1w0r0/1/1>}.

A.24. Defecte dinamice de citire incorectă (DDCIxy).

Acesta este un defect prin care orice secvenţă de tipul xwyry, dintr-o secvenţă de stimulare, returnează valoarea logică y menţinând în acelaşi timp starea corectă a celulei. Există patru tipuri de defecte dinamice de citire incorectă, pe care le prezint în continuare: DDCI00 = {<0w0r0/0/1>}, DDCI11 = {<1w1r1/1/0>}, DDCI01 = {<0w1r1/1/0>} şi DDCI10 = {<1w0r0/0/1>}.

2.2. DEFECTE LA NIVELUL LEGĂTURILOR DINTRE CELULELE SRAM

B. Modele funcţionale de defecte ce afectează două celule de memorie (Two-cell FFMs) [GoAl00]

Acestea sunt modele de defecte ce afectează două celule de memorie. Celula de memorie care ne arată comportarea defectuoasă se numeşte *celulă victimă*, iar celula cu care interacţionează celula victimă pentru a produce defectul se numeşte *celulă agresoare*. Ele se clasifică, în funcţie de numărul operaţii efectuate asupra celulei, în: *statice* şi *dinamice*, iar în fig. 2.3. sunt prezentate toate modelele funcţionale de defecte ce afectează două celule de memorie.

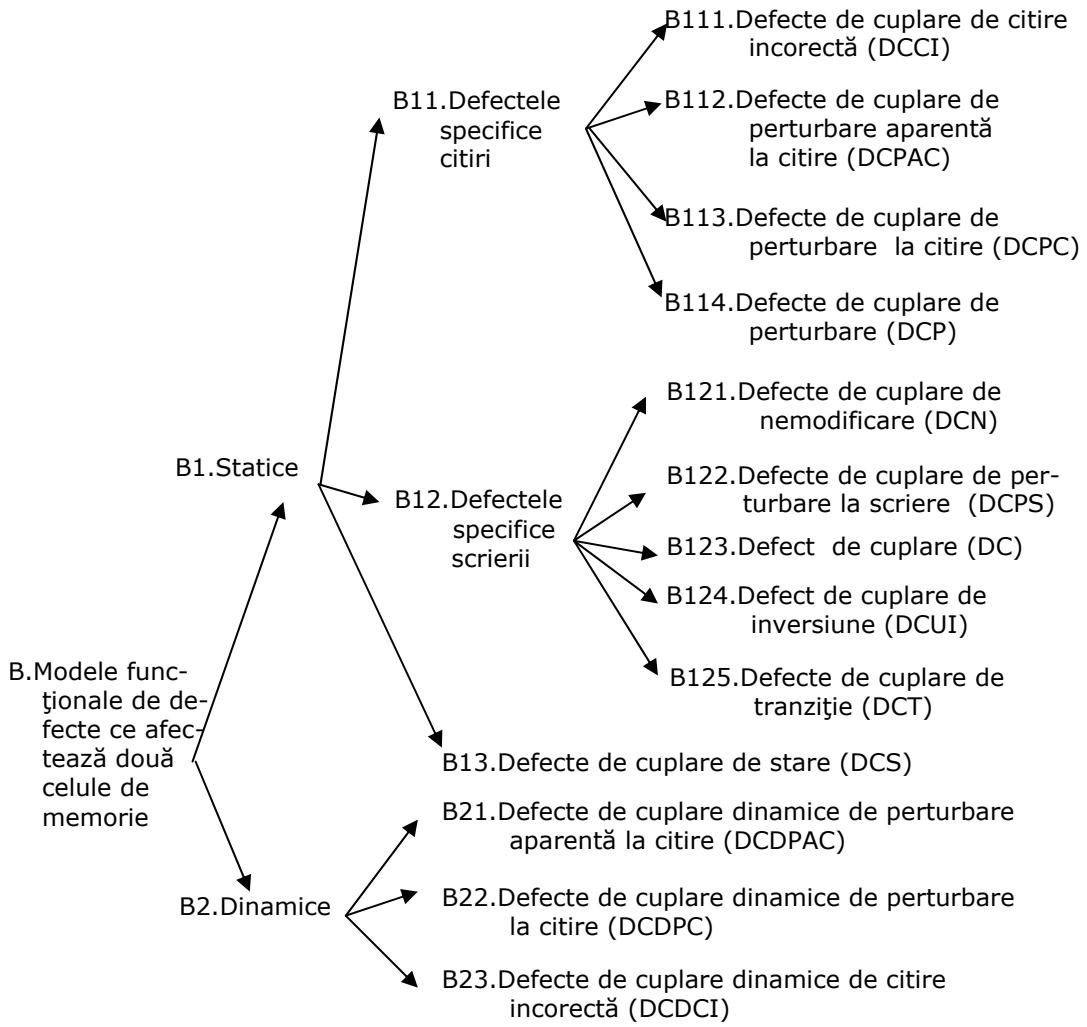


Fig. 2.3.

Luând în consideraţie şi în acest caz specificul celulei SRAM utilizată în cache-uri faţă de celula DRAM specifică memoriilor principale, preconizez clasificarea diferită faţă de cea din [GoAl00]. pentru cele statice, în defecte specifice citirii, scrierii, şi de stare, iar pentru cele dinamice în defecte specifice citirii.

În acest caz modelele descriu defecte ce pot fi puse în evidenţă dacă se efectuează cel puţin o operaţie asupra lor, în timp ce cele două celule acţionează una asupra celeilalte, astfel notaţia SS = $a(viOPd)_1$ $a(viOPd)_2$ ... $a(viOPd)_k$ ... $a(viOPd)_n$ poate fi simplificată prin una din următoarele secvenţe:

S_{sn} = ag(i) av(j): dacă nici o celulă nu este accesată,

S_{sag} = ag(iOPd) av(j): numai celula agresoare este accesată,

S_{sav} = ag(i) av(jOPd): numai celula victimă este accesată.

unde: i respectiv j sunt stările iniţiale ale agresorului respectiv ale victimei,

ag şi av sunt poziţia adresei agesorului, respectiv a victimei.

B1. Modele funcţionale statice de defecte ce afectează două celule de memorie (Static two-cell FFMs)

Indicele s indică faptul că avem o secvenţă statică, indicele n din S_{sn}, indică că avem o secvenţă fară operaţii, aceasta fiind situaţia în care doar observăm starea celor două celule. De asemenea, indicele ag din S_{sag} şi av din S_{sav} indică faptul că avem doar o singură operaţie ce se efectuează asupra agresorului, respectiv asupra victimei. Se observă faptul că în secvenţa de operaţii agresorul este pus întotdeauna primul. De obicei, secvenţele de două celule sunt date printr-o notaţie specială care separă partea de secvenţă privitoare la agresor faţă de partea de secvenţă privitoare la victimă, şi vom specifica ulterior şi adresele celulelor în ordinea ag; av. Rezultă astfel că orice secvenţă statică SS ce afectează două celule arată astfel: SS = S_{ag} ; S_{av} secvenţa SS fiind efectuată asupra celulelor ag, av.

Vom aplica o notaţie specială SS şi astfel vom avea următoarea reprezentare :

$<SS/M/O> = <S_{ag}; S_{av}/M/O>_{ag;av}$

B.11. Defecte specifice citirii

B.111. Defecte de cuplare de citire incorectă (DCCI). Două celule au un astfel de defect, dacă o operaţie de citire efectuată asupra victimei returnează o valoare logică incorectă când agresorul este setat într-o stare logică dată. Există patru tipuri de astfel de defecte: DCCIx;y = {<x; yry/y/$\bar{y}$ >}, unde x, y $\in$ {0,1}.

B.112. Defecte de cuplare de perturbare aparentă la citire (DCPAC). O celulă are un astfel de defect dacă o operaţie de citire efectuată asupra victimei returnează valoarea logică corectă şi schimbă conţinutul victimei când agresorul este setat într-o anumită stare logică dată. Există patru tipuri de astfel de defecte: DCPACx;y = {<x; yry/$\bar{y}$ /y>}, unde x, y $\in$ {0,1}.

B113. Defecte de cuplare de perturbare la citire (DCPC). Două celule au un astfel de defect dacă o operaţie de citire efectuată asupra victimei distruge data

stocată în victimă dacă agresorul este setat într-o anumită stare. Există patru tipuri de astfel de defecte DCPC = {<x; yry/$\overline{y}$ / $\overline{y}$ >}, unde x, y ∈ {0,1}.

B114. Defecte de cuplare de perturbare (DCP) Două celule au acest tip de defect dacă o operaţie (citire sau scriere) efectuată de către agresor forţează victima într-o stare logică dată. Orice operaţie efectuată de agresor se consideră că sensibilizează defectul. Există douăsprezece tipuri de astfel de defecte: DCPxwy;z = {<xwy; z/$\overline{z}$ /->} şi DCPxrx;y = {<xrx; y/$\overline{y}$ /->}, unde x, y, z ∈ {0,1}.

B.12. Defecte specifice scrierii

B.121. Defecte de cuplare de nemodificare (DCN). Două celule au un astfel de defect dacă o operaţie de scriere de nemodificare (0w0 şi 1w1), efectuată de către agresor aduce victima într-o stare dată. Cele patru tipuri de defecte de acest tip sunt date prin relaţia: DCN xwx;y = {<xwx; y/$\overline{y}$ /->}, unde x, y ∈ {0,1}.

B.122. Defecte de cuplare de perturbare la scriere (DCPS). O celulă are un astfel de defect dacă după ce se efectuează asupra victimei o operaţie de scriere (cu nemodificarea stării logice), rezultă o modificare a acestei stări, când agresorul este setat într-o stare logică dată. Există patru tipuri de astfel de defecte : DCPSx;y = {<x; ywy/$\overline{y}$ /->}, unde x, y ∈ {0,1}.

B.123. Defect de cuplare (DC) - Două celule au acest defect, dacă o operaţie de tranziţie de scriere (0w1 şi 1w0) a agresorului forţează victima într-o anumită stare. Acest defect este stimulat printr-o operaţie de scriere cu modificarea stării efectuată de agresor. Există patru tipuri de astfel de defecte, care se pot sintetiza cu ajutorul relaţiei : DCxw$\overline{x}$;y ={<xw$\overline{x}$; y/$\overline{y}$ /->}, unde x, y ∈ {0,1}.

B.124. Defect de cuplare de inversiune (DCUI) – Două celule au un astfel de defect dacă valoarea logică a victimei este inversată în cazul unei operaţii de scriere, efectuată de către agresor. Cele două tipuri de defecte de acest fel sunt sintetizate prin relaţia: DCUIxw$\overline{x}$ = {<xw$\overline{x}$; y/$\overline{y}$ /->,<xw$\overline{x}$; $\overline{y}$ /y/->}, unde x, y ∈ {0,1}.

B.125. Defecte de cuplare de tranziţie (DCT) - Două celule au acest tip de defect dacă, avem o valoare logică dată a agresorului, valoare rezultată în urma unui eşec al unei operaţii de tranziţie la scriere. Acest defect este stimulat printr-o operaţie de scriere asupra victimei şi setarea agresorului într-o stare dată. Există patru tipuri de astfel de defecte care pot fi sintetizate prin relaţia: DCTx;↑ = {<x; 0w1/0/->} şi DCTx;↓ = {<x; 1w0/1/->}, unde x ∈ {0,1}.

B.13. Defecte de cuplare de stare (DCS)

Două celule se spune că au un defect de cupare de stare numai dacă agesorul se găseşte într-o stare dată, fără a efectua nici o operaţie asupra victimei. Acest defect este special în sensul că nu trebuie să se efectueze nici o operaţie, mai mult chiar, el depinde numai de valoarea iniţială stocată în celule. Există patru tipuri de defecte de cuplare de stare care pot fi sintetizate cu ajutorul relaţiei: DCS x;y = {<x;y/$\bar{y}$/->}, unde x,y $\in$ {0,1}.

B.2. Modele funcţionale dinamice de defecte ce afectează două celule de memorie (Dynamic two-cell FFMs) [GoAl00]

Vom restrânge analiza doar pentru cazul modelelor de defecte dinamice ce afectează două celule de memorie. Vom folosi notaţia specială: < SS/M/O>, pentru clasa dinamică de defecte cu două operaţii, SS putând fi una din următoarele:

S_{agag} = ag(iOP1d1OP2d2) av(j);

S_{agav} = ag(iOP1d1) av(jOP2d2);

S_{avag} = av(jOP1d1) ag(iOP2d2);

S_{avav} = ag(i) av(jOP1d1OP2d2).

În continuare avem o clasificare a acestor modele funcţionale dinamice de defecte, toate fiind specifice citirii.

B.21. Defecte de cuplare dinamice de perturbare aparentă la citire (DCDPAC). Sunt defecte prin care un cuplu agresor-victimă ag(x) av(ywyry) sau ag(x) av($\bar{y}$ wyry)) din secvenţa de stimulare, returnează valoarea logică corectă y, în timp ce distrug starea celulei. Există opt tipuri de astfel de defecte: DCDPACx;yz = {<a(x) v(ywzrz)/ $\bar{z}$ /z>}, unde x, y, z $\in$ {0,1}.

B.22. Defecte de cuplare dinamice de perturbare la citire (DCDPC) Sunt defecte prin care un cuplu agresor-victimă ag(x) av(ywyry) sau ag(x) av($\bar{y}$ wyry), din secvenţa de stimulare, schimbă valoarea logică memorată în celula victimă, în $\bar{y}$ şi astfel vom avea o valoare incorectă la ieşire. Există opt tipuri de astfel de defecte: DCDPCx;yz = {<ag(x) av(ywzrz)/ $\bar{z}$ / $\bar{z}$ >}, unde x, y,z $\in$ {0, 1}.

B.23. Defecte de cuplare dinamice de citire incorectă (DCDCI). Sunt defecte prin care un cuplu agresor-victimă ag(x) av(ywyry) sau ag(x) av($\bar{y}$ wyry) din secvenţa de stimulare returnează valoarea logică $\bar{y}$, păstrând în acest timp starea

corectă a celulei. Există opt tipuri de astfel de defecte: DCDCIx;yz = {<a(x) v(ywzrz)/z/$\bar{z}$ >}, unde x,y,z $\in${0,1} [FaHi93] .

2.3. CONCLUZII

După parcurgerea literaturii de specialitate [NiOR97], [NHKK99], [OmRM03], [[MeFR97], care este bogată în încercări de modelare a defectelor din memorii principale, hard disc-uri, şi alte diferite tipuri de memorii pe suport magnetic, am constat că treapta cache, a unei ierarhii de memorii, care constituie ţinta investigaţiilor din această teză, este mai slab prezentată. În căutarea individualizării problematicii de testare pentru cache-uri plecând de la suportul tehnologic distinct al acestor memorii, am analizat în acest paragraf modurile de defectare la nivelul cel mai intim al celulelor de memorare SRAM şi la nivelul legăturilor dintre celulele SRAM şi astfel am realizat o clasificare a acestor defecte. Am efectuat o clasificare a modelelor fucţionale de defecte în clasa statică şi clasa dinamică, criteriul care a stat la baza acestei divizării fiind numărul de operaţii aplicate asupra celulei de memorare. Astfel modelele funcţionale statice de defecte au cel mult o operaţie de scriere sau citire, iar cele dinamice au cel puţin două operaţii de sciere sau citire. Ţinând cont de specificul celulei SRAM utilizată în cache-uri, faţă de celula DRAM utilizată la memoria principală am efectuat clasificarea modelelor funcţionale statice ce afectează celula SRAM în: defecte specifice citirii, defecte specifice scrierii, defecte de blocare_pe , defecte de stare şi defecte ale timpului de acces, iar pentru cele dinamice în defecte specifice citirii. În cazul defectelor care se manifestă la nivelul legăturilor dintre două celule SRAM am efectuat clasificarea modelelor funcţionale statice ce afectează aceste două celule în: defecte specifice citirii, defecte specifice scrierii şi defecte de cuplare de stare, iar pentru cele dinamice în defecte specifice citirii.

Putem spune că nu am făcut o prezentare exhaustivă, rezumându-ne la defectele specifice ale memoriilor cache, defecte care sunt acoperite de codurile corectoare şi detectoare de erori care s-au folosit ulterior.

3. CONFIGURAREA STRUCTURII UNUI CACHE TIPIC URMĂRIND STABILIREA LOCULUI DE INTRODUCERE A ELEMENTELOR DE REDUNDANŢĂ MENITE CREŞTERII DEPENDABILITĂŢII

3.1. INTERFAŢAREA CACHE-ULUI CU CELELALTE TREPTE ALE UNEI IERARHII DE MEMORII

Preconizăm traversarea în prezentul capitol a etapelor de proiectare specifice unui cache în scopul reliefării acelor puncte critice care reclamă protejarea lor prin elemente redundante în scopul creşterii atributului de dependabilitate. În acest sens, fară a piedere din generalitate şi urmărind o simplitate în urmărirea detaliilor constructive, vom adopta caracteristici de capacitate redusă pentru cache şi pentru memoria principală. Astfel, în calitate de date de intrare pentru soluţia de proiectare admitem că dispunem de un cache de 128KB (32K x 32b), (unde am folosit notaţia de B pentru un Byte (octet) şi b pentru bit) şi de o memorie principală de 16 MB implementată cu chip-uri DRAM de 1Kb.

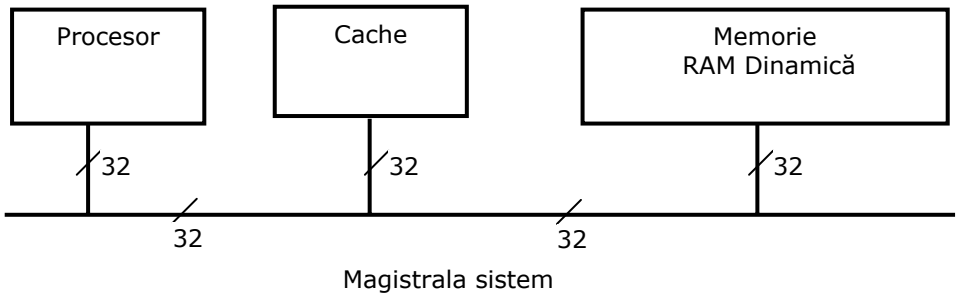


Fig. 3.1.

De asemenea, mai admitem că apelăm la o mapare directă a adreselor de memorie principală în cache, iar politica de scriere pe care o adoptăm este write-through, în sensul că, la scriere, aceasta se efectuează atât în cache, cât şi în memoria principală. Magistrala o adoptăm cu 32 linii, iar dimensiunea unui cache posedă 16 B. Structura supusă activităţii de proiectare este dată în fig. 3.1. La o astfel de configuraţie procesorul accesează informaţia, aceasta provenind fie din memoria RAM dinamică, fie din cache. Dacă informaţia este disponibilă în cache (în cazul unui hit), ea este accesată rapid şi procesorul îşi continuă activitatea. Dacă informaţia nu este prezentă în cache (în cazul unui miss), ea trebuie adusă din memoria RAM dinamică [Swaz87].

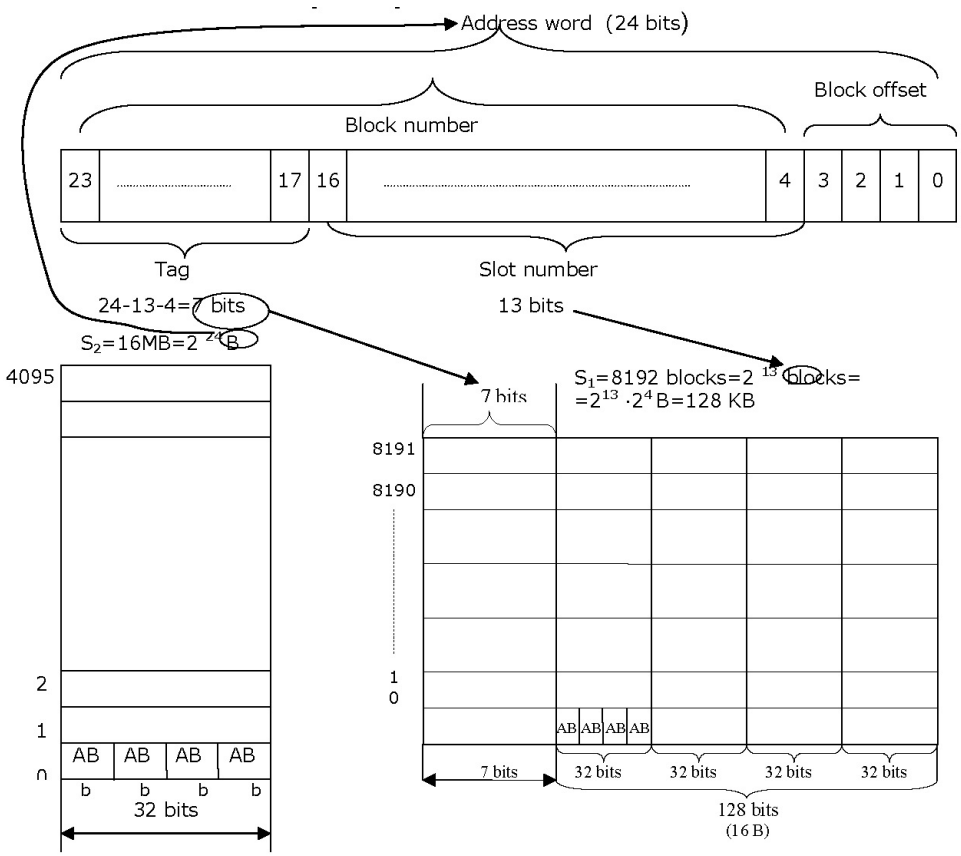


Fig. 3.2.

Un cache direct mapat specifică faptul că dacă informaţia este în cache, există o singură locaţie în care se poate găsi aceea informaţie. Cache-ul este organizat în unităţi mici, numite blocuri.

Acest sistem va efectua patru transferuri de cuvinte pentru a umple un bloc din cache. În conformitate cu datele de intrare constând din capacitatea memoriei principale de S_2 = 16MB, întrucât 16MB = 2^{24} B, magistrala de adresă va avea 24 de linii, notate de la 0 la 23, ele fiind reprezentate în fig. 3.2. De asemenea, având dimensiunea blocurilor egală cu 16 B = 2^4 B, pentru a asigura accesul la nivelul de B (octet), vom avea din cei 24 de biţi de adresă, 4 alocaţi aşa numitului block offset.

De asemenea, întrucât capacitatea cache-ului este S_1 = 8192 blocuri = 2^{13} blocuri = $2^{13} \cdot 2^4 \cdot$ B = 128 KB, din cei 24 de biţi de adresă câmpul corespunzător unui slot de cache este de 13 biţi. Concluziv, pentru câmpul de Tag rămâne 24 – 4 – 13 = 7 biţi. Putem spune că dimensiunea reală este egală cu $(7+32 \cdot 4) \cdot 8192$ biţi. Cache-ul se interfaţează cu procesorul şi memoria principală prin magistrala de adresă de 24 de biţi, una de date de 32 de biţi, precum şi prin linii de comandă care vor fi discutate în cele ce urmează.

3.2. DETALIERE CONSTRUCTIVĂ A UNUI CACHE

Schema bloc din fig. 3.1. ne spune cum funcţionează sistemul doar la modul general, această schemă nu cuprinde şi semnalele care trebuiesc activate pe parcursul procesului de citire, respectiv pe parcursul procesului de scriere.

3.2.1. Schema bloc a cache-ului

În fig. 3.3. este prezentată schema bloc a cache-ului.

Acestă schemă bloc conţine şi semnalele care trebuiesc activate la citirea din memorie, respectiv la scrierea în memorie [NovN01].

Sistemul cache se împarte în două, o parte de date, memorie pe care o numim în continuare cache RAM, cu dimensiunea de 32K x 32 biţi şi o parte de Tag, memorie pe care o vom denumi cache TAG, memorie care are dimensiunea de 8K x 8 biţi. Deoarece sunt patru cantităţi de 16 biţi pe o linie din cache, o locaţie de Tag va identifica patru locaţii în cache. Liniile de date ale cache-ului sunt conectate la magistrala de date a sistemului, această cale fiind folosită atât la scrierea datelor în cache cât şi la citirea datelor din cache. Adresele spre cache RAM se obţin din liniile ADR(16-2) ale magistralei de adrese.

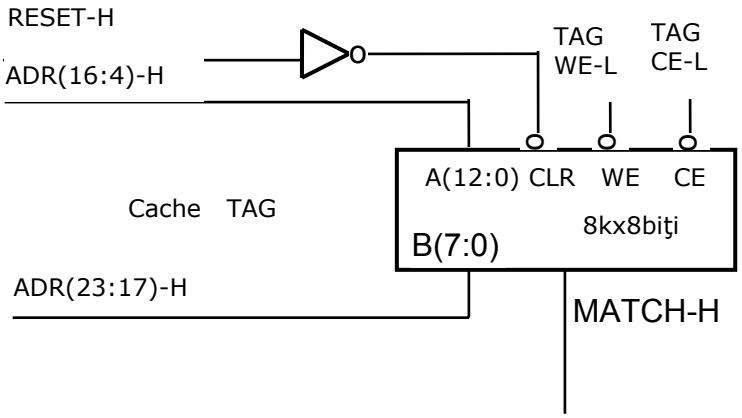


Fig. 3.3. a

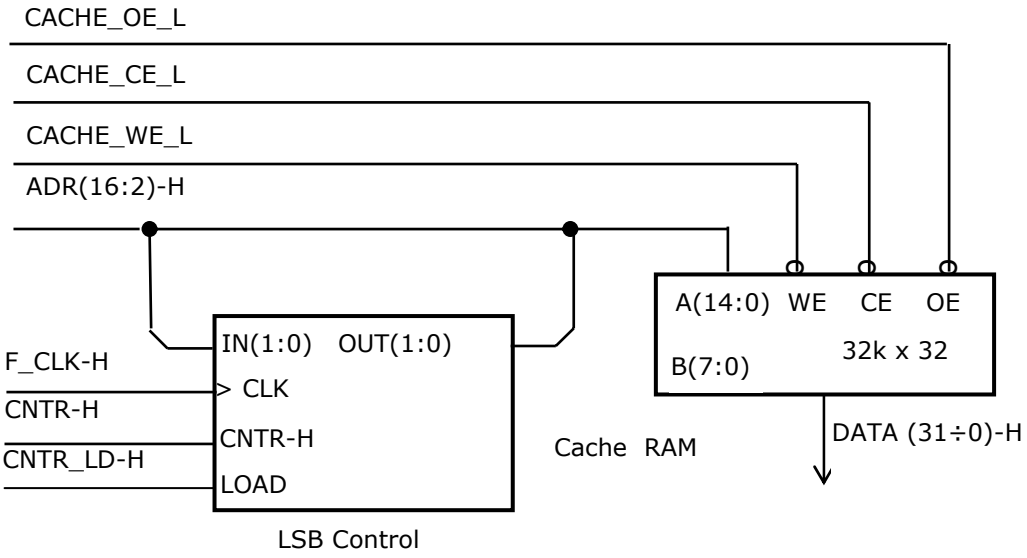


Fig. 3.3. b

Din cei 15 biţi, 13 vor fi folosiţi la identificarea liniei din cache, iar următorii biţi sunt utilizaţi pentru a identifica în care dintre locaţiile de 4 octeţi din cadrul unei linii se găseşte adresa octetului căutat. Cei doi biţi mai puţin semnificativi ai magistralei de adrese (CMPS), sunt intrări pentru unul din blocurile cache RAM-ului, acest bloc purtând denumirea de LSB Control. El conţine un numărător şi un multiplexor. Pe parcursul unei operaţii normale cei doi biţi CMPS sunt furnizaţi prin intermediul multiplexorului din adresă. Când se produce încărcarea unei linii de cache din memoria RAM dinamică, cei doi biţi CMPS sunt obţinuţi din numărător, care incrementează adresa cu patru locaţii. Adresa disponibilă în cache pe cele două linii CMPS este adresa care provine din sistem, cu excepţia cazului în care semnalul

CNTR-H este activ, situaţie în care adresa este furnizată de către partea care conţine numărătorul din cadrul cache. Numărătorul este încărcat cu adresa, când este activ semnalul CTR_LD-H.

Există şi alte linii de control, linii care se găsesc de obicei în orice sistem ce conţine memorie RAM. Datele se scriu în cache dacă semnalul CACHE_WE-L este activ, chipul de memorie RAM este selectat prin intermediul semnalului CACHE_CE-L, iar datele pot fi accesate din memoria RAM când semnalul CACHE_OE-L este activ. În fig. 3.4. sunt prezentate circuitele care alcătuiesc cache RAM-ul.

În această figură avem patru chip-uri de memorie SRAM, fiecare având capacitatea de 32 K x 8 biţi. Timpul de acces al acestor memorii este de 35 nsec. Semnalele CACHE_CE-L şi CACHE_OE-L sunt conectate la cele patru chip-uri SRAM. Semnalele write enable sunt separate şi sunt direcţionate către chip-urile de memorie corespunzătoare prin intermediul unui multiplexor (74157,'157).

Când avem un hit în cache, semnalele write enable care vor fi utilizate sunt furnizate de sistem prin intermediul semnalelor WEx. Dacă nu avem hit toate cele patru linii vor fi validate de semnalul CACHE_WE-L. În această figură mai există un bloc intitulat LSB Control, care este utilizat pentru a furniza cei mai puţini semnificativi doi biţi ai adresei. Dacă avem un hit în cache, fie la citire, fie la scriere, acei biţi sunt obţinuţi din magistrala de adrese. Dacă în schimb avem un miss în cache, cei doi biţi sunt furnizaţi de un numărător din LSB Control, cei doi biţi fiind obţinuţi din memoria DRAM.

Semnalele din porţiunea de TAG a cache-ului, TAG_WE-L şi TAG_CE-L ne permit scrierea datelor în memoria cache TAG, respectiv selecţia circuitelor ce compun această memorie cache TAG.

Memoria cache TAG este o memorie specială deoarece conţine pe lângă chip-urile de memorie şi o parte care realizează compararea componentei de TAG a adresei, un astfel de bloc de cache TAG este prezentat în fig. 3.5. Informaţia stocată la adresa identificată pe liniile de adresă este comparată cu informaţia prezentă pe liniile de date şi dacă cele două informaţii sunt identice semnalul MATCH-H este activat. Liniile din fig. 3.3., care sunt conectate la liniile de date ale sistemului de memorie TAG sunt cele mai semnificative (CMS) 7 linii ale magistralei de adrese. O condiţie care mai intervine este poziţionarea unei linii la "1" logic, scopul acestuia fiind punerea la dispoziţie a unui mecanism de iniţializare. La activarea semnalului RESET-H, conţinutul memoriei cache TAG este şters şi în privinţa adresei furnizate pe intrări o coincidenţă (match) nu va apărea până când va fi plasată o valoare în memoria TAG deoarece valoarea "0", valoarea prezentă în cache din momentul activării semnalului RESET-H, nu va coincide cu valoarea "1" care este prezentă la intrările de date.

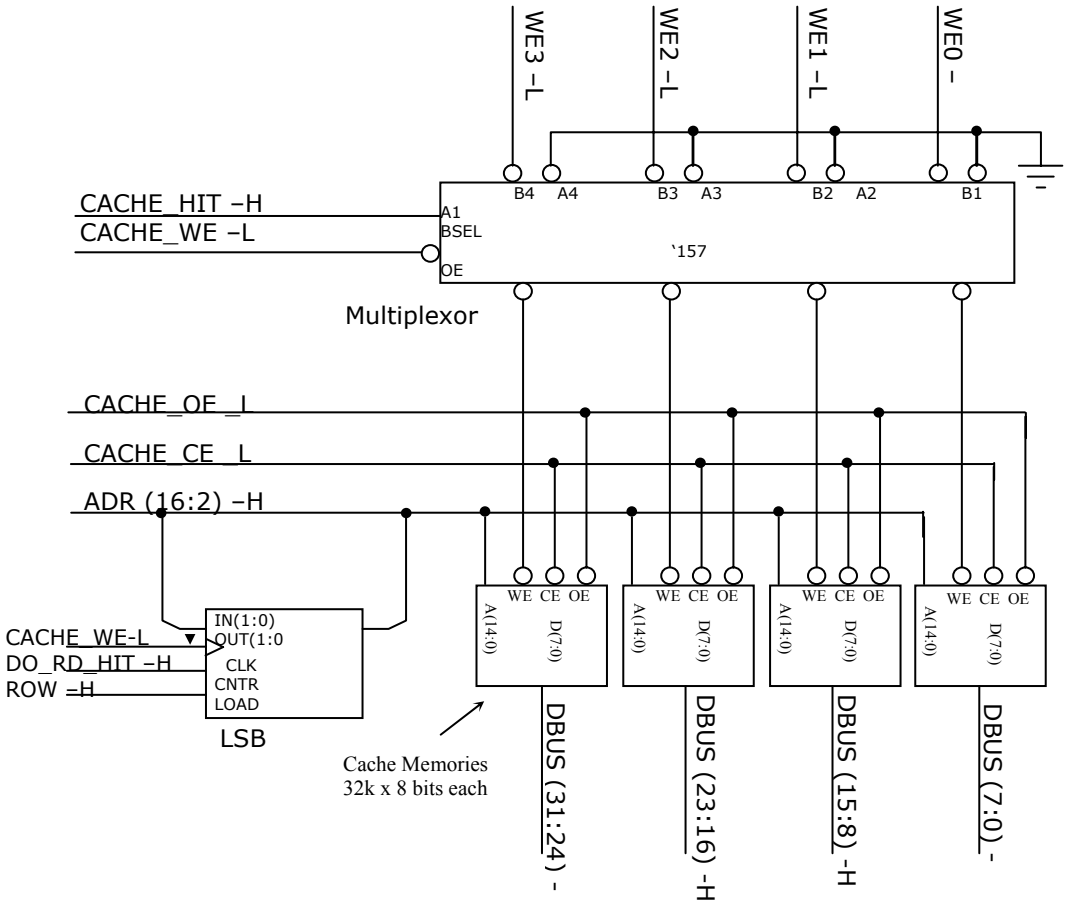


Fig.3.4.

Astfel, pot fi identificate locaţiile goale din cache. Liniile de adrese folosite în cadrul memoriei cache TAG sunt cele 13 linii de adresă care fac identificarea liniei în cauză. În final se face înscrierea informaţiei, de fapt adresa disponibilă pe liniile de date, în memoria TAG când se activează semnalul TAG_WE-L, iar validarea selecţiei acestei memorii se face activând semnalul TAG_CE-L. Aceste semnale pot fi asociate la scheme cu ajutorul unui mecanism, fie prin diagrame de stare, fie prin diagrame de timp.

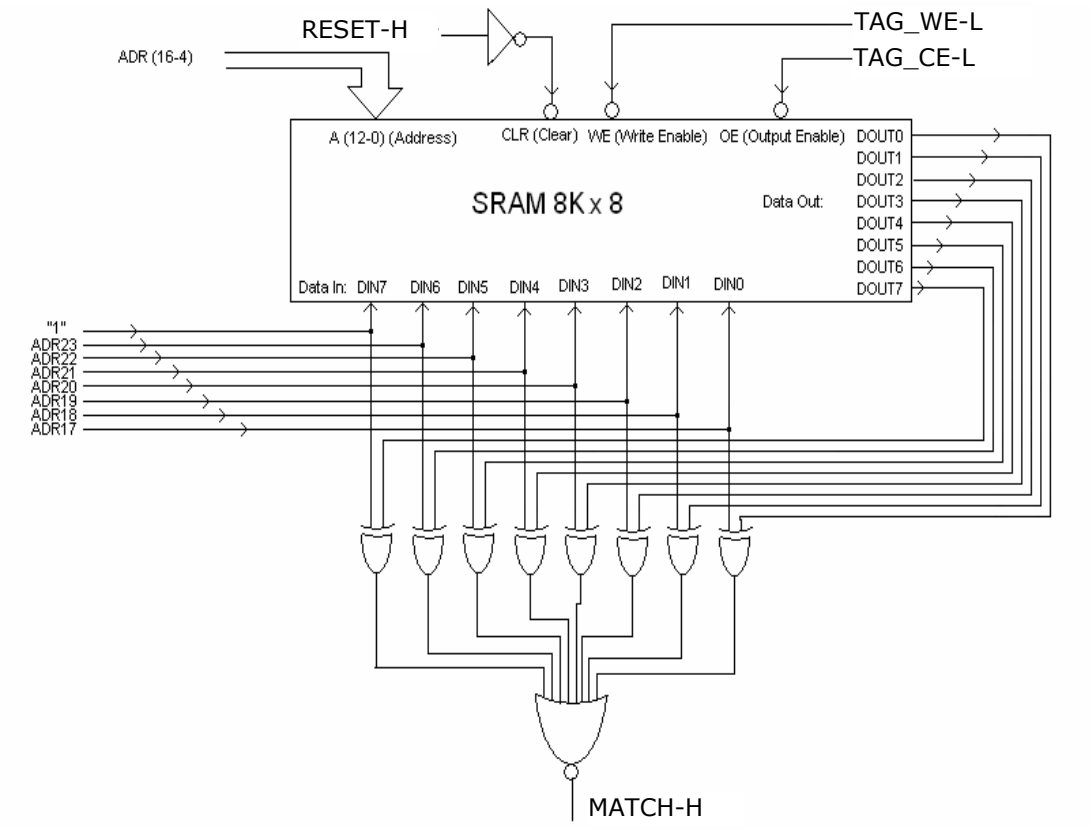


Fig. 3.5.

3.3. SINCRONIZAREA FUNCŢIONALĂ A TREPTEI CACHE

Pentru a sintetiza funcţionarea treptei cache vom utiliza în cazul nostru diagrame de timp. Aceste diagrame de timp, conţin informaţii despre circuitele ce alcătuiesc treapta cache precum şi ordinea de desfăşurare în timp a evenimentelor ce au loc. Astfel vom prezenta în continuare patru diagrame de timp, câte o diagramă de timp pentru cele patru cicluri în care se poate găsi cache-ul, şi anume: hit la citire (read-hit) (fig. 3.6.), miss la citire (read-miss)(fig. A.1), hit la scriere (write-hit)(fig. A.2) şi miss la scriere (write-miss) (fig. A.3).

De asemenea vom asocia câte o ordinogramă pentru fiecare diagramă de timp, acestea având rolul de descriere detaliată a funcţionării treptei cache. Activitatea din sistem este raportată la intervale de 15 nsec. Vom include în

diagramele de timp toate semnalele, chiar dacă unele dintre ele nu sunt activate într-un anumit ciclu. În continuare prezint în fig.3.6, ciclul hit la citire

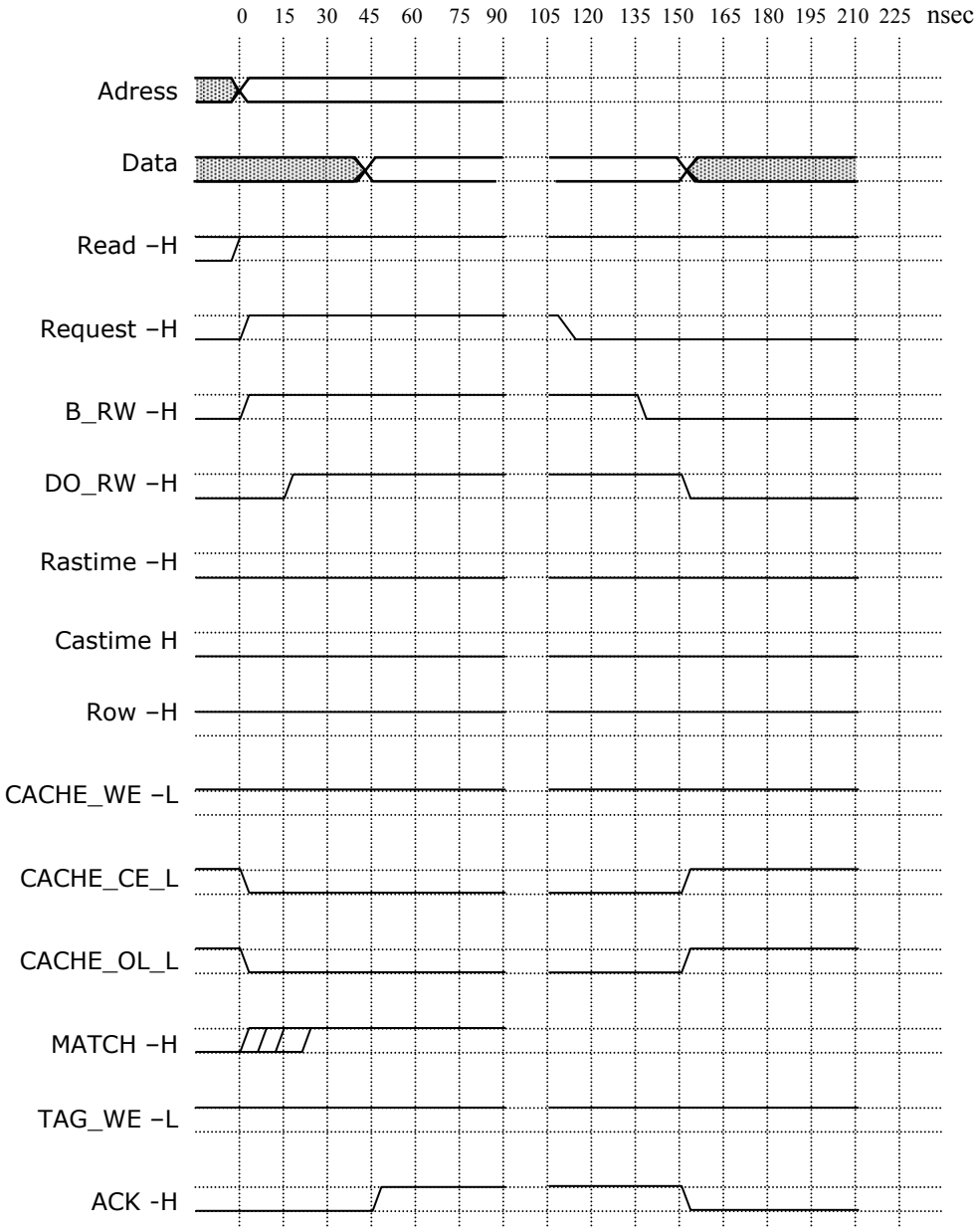


Fig. 3.6

Primul semnal care este inclus în aceste diagrame este Adress şi el reprezintă setul de linii de adresă care sunt direcţionate atât spre memoria principală DRAM cât şi spre treapta cache. Al doilea semnal, Data, reprezintă liniile de date care în acest caz sunt obţinute de la treapta cache. Al treilea semnal, Read-H, reprezintă linia de citire, care ne precizează tipul ciclului. Acest ciclu poate fi de citire (read-hit sau read-miss) sau de scriere (write-hit sau write-miss). Semnalul se poate obţine din semnalele de control ale magistralei sistemului. Următorul semnal, B_RW-H, este semnalul buffer-at de citire/scriere a unui sistem de memorie DRAM. Semnalul următor DO_RW-H, reprezintă semnalul de iniţiere a unui ciclu de citire/scriere. Urmează semnalele RAS time (Rastime-H) şi CAS time (Castime-H) folosite în cadrul unui ciclu de reîmprospătare (refresh) a memoriei DRAM. Semnalul Row-H ne precizează când adresa de rând este prezentă pe magistrala de adrese, semnal utilizat la memoria DRAM. Următoarele trei semnale sunt write enable (CACHE_WE-L), chip enable (CACHE_CE-L) şi output enable (CACHE_OE-L) ale cache RAM-ului. Următorul semnal este MATCH-H al memoriei cache TAG, acesta fiind urmat de semnalul write enable (TAG_WE-L) al memoriei cache TAG. Ultimul semnal este ACK-H (Acknowledge), acesta identifică un protocol de tip handshake care va realiza coordonarea ciclului cu un master de magistrală.

Primul ciclu pe care îl prezentăm este hit la citire (read-hit), ciclu prezentat în fig. 3.6. Ordinograma ciclului hit la citire este prezentată în fig. 3.7. În aceasta se observă la momentul t = 0 sec, că liniile de adresă sunt stabile pe magistrala de adrese. Tot acum se realizează activarea semnalului Read-H, după care urmează activarea semnalului Request-H. Prin activarea acestor două semnale cache-ul este informat că s-a solicitat o operaţie de citire din cache, pentru a verifica dacă informaţia respectivă este în cache.

Cache-ul va acţiona în consecinţă şi va activa semnalele corespunzătoare pentru a deservi această cerere. Acesta va activa semnalele chip enable (CACHE_CE-L) şi output enable (CACHE_OE-L) pentru a putea plasa pe magistrala de date informaţia dacă ea este prezentă în cache. Memoria cache TAG este verificată şi semnalul MATCH-H ne va indica, că avem un succes (hit), după 22 nsec de la începutul tranzacţiei, deoarece linia stocată la locaţia identificată prin liniile ADR (16:4) este identică cu adresa curentă. Vom presupune, conform specificaţiilor tehnice, că memoria RAM ne poate furniza datele după un interval de 40 nsec, Semnalul ACK-H va fi activat la începutul celui de al treilea ciclu de clock la 45 n sec de la începutul tranzacţiei. Activarea semnalului Request-H va determina activarea sermnalului buffer-at de citire/scriere B_RW-H, iar semnalul DO_RW-H, de iniţiere a unui ciclu de citire/scriere, va fi activat după 15 nsec de la activarea lui B_RW-H numai dacă nu se execută un ciclu de reâmprospătare.

După ce s-a activat semnalul DO_RW-H, poate urma activarea semnalului ACK-H. Odată ce linia ACK-H a fost activată sistemul va rămâne în această

configuraţie până când semnalul Request-H este dezactivat. Acest lucru este prezentat şi în fig. 3.7. în blocul de decizie unde se testează dacă mai este activă linia Request-H. Dacă linia este activă se aşteaptă următorul ciclu (după 15 nsec).

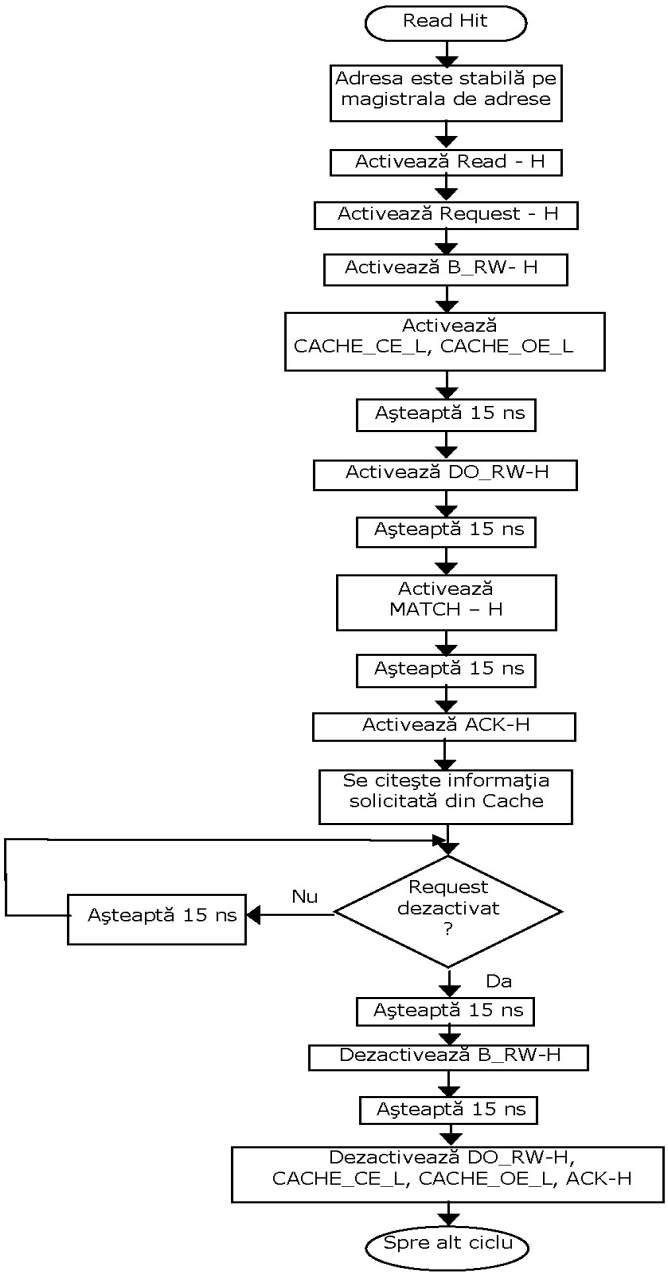


Fig. 3.7.

Dacă linia Request-H s-a dezactivat după 15 nsec se dezactivează linia B_RW-H, apoi după 15 nsec se dezactivează linia DO_RW-H, urmată la acelaşi moment de timp şi de dezactivarea liniilor de date, a semnalelor CACHE_CE-L, CACHE_OE-L şi a semnalului ACK-H lucru care marchează şi sfârşitul ciclului de hit la citire.

Al doilea ciclu pe care îl prezentăm, care este şi cel mai complex, este miss la citire (read-miss), ciclu prezentat în fig. A.1 şi comentat, în anexa A, unde este prezentată, în fig. A.4. şi ordinograma corespunzătoare. Ciclurile de scriere sunt asemănătoare, deoarece pentru orice scrire există şi o sciere în memoria principală DRAM. Scrierea se va realiza şi în cache doar dacă avem un hit. Acest aspect este prezentat în ciclul hit la scriere (write-hit) în fig. A.2, ordinograma acestuia fiind relatată în fig. A.5. în anexa A. Ciclul miss la scriere este prezentat în fig. A.3, iar în fig. A.6. din anexa A, se prezintă ordinograma corespunzătoare.

3.4. DETERMINAREA LOCULUI DE INTRODUCERE A ELEMENTELOR REDUNDANTE MENITE A ASIGURA CREŞTEREA DEPENDABILITĂŢII

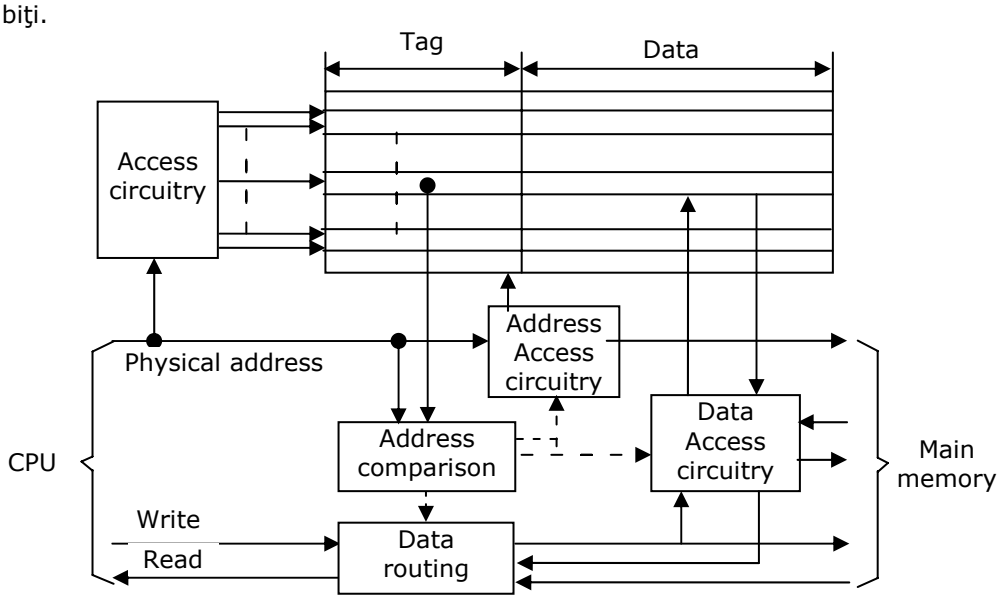
Vom prezenta în cele ce urmează o structură de cache cu logica de acces aferenetă în scopul de a putea observa ulterior care sunt locurile de introducere a elementelor redundante. În fig. 3.8. se poate observa modul de conectare al cache-ului cu CPU şi memoria principală. CPU-ul furnizează atât semnalele de scriere (Write), respectiv citire (Read), cât şi adresa fizică (Physical Address) unde, respectiv de unde, se va face accesul la informaţie.

Aşa cum este prezentat şi în [RaFu89] biţii de control adăugaţi biţilor utili pentru coduri polinomiale, în general, pot fi generaţi fie pe seama unei operaţii de înmulţire, fie pe seama unei operaţii de împărţire efectuată în câmpul Galois cu 2 elemente (GF(2)). În cele ce urmează vom considera că numărul biţilor utili este egal cu u, iar cel al biţilor de control este egal cu k. Pentru asigurarea corecţiei erorii singulare, între cele două, u şi k, există cunoscuta condiţie reprezentată de:

$$2^k \geq u+k+1 \qquad (3.1.)$$

În figura 3.9 este prezentat un cache cu o parte redundantă introdusă pe partea de date a cache-ului. Pentru a putea determina dimensiunea părţii redundante vom lua ca exemplu cache-ul cu dimensiunea de (7+128) KB = 135 KB, prezentat la începutul capitolului. Acest cache este mapat direct, şi politica de scriere este write-through. În acest caz particular numărul biţilor utili este u = 135. Conform relaţiei 3.1. va rezulta că numărul biţilor de control este k = 8. Astfel,

pentru cache-ul prezentat în fig. 3.9, numărul total al biţilor utilizaţi este de k+ u = 143 biţi.



Fir 3.8

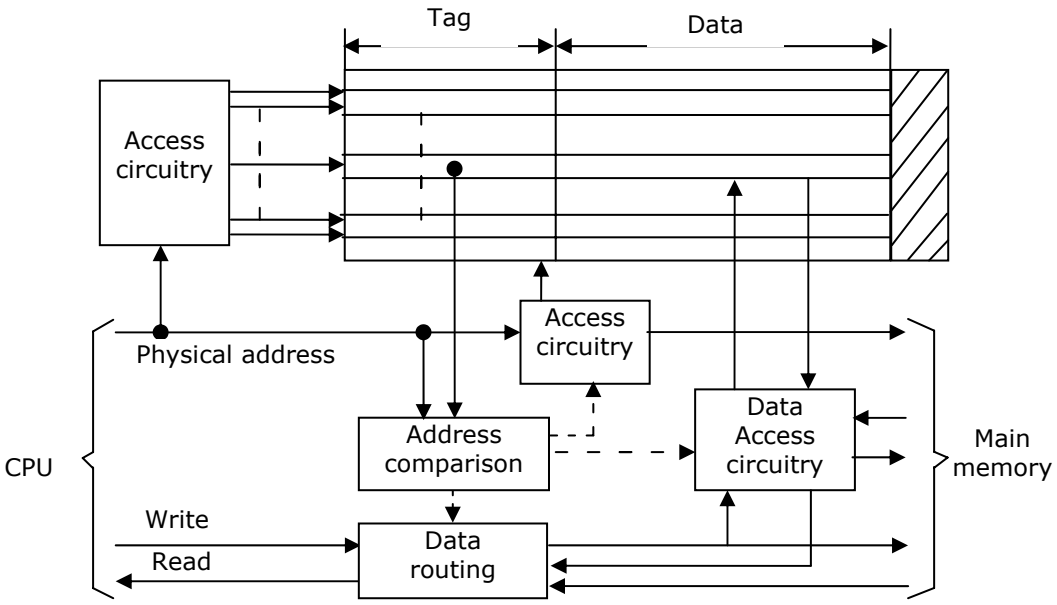


Fig.3.9.

În fig. 3.10 este prezentat un cache cu partea redundantă introdusă atât pe partea de date, cât şi pe partea de TAG a cache-ului. Pentru a putea determina dimensiunea totală a părţii redundante vom calcula numărul biţilor de control necesari pentru partea de TAG şi numărul biţilor de control necesari pentru partea de date. Pentru partea de TAG avem k1= 4 biţi. Pe partea de date cei 128 de biţi utili îi împărţim în 4 grupuri de câte 32 de biţii fiecare, şi pentru fiecare grup vom calcula numărul biţilor de control.

Pentru un astfel de grup vom avea nevoie de k2 = 6 biţi, dar deoarece sunt 4 astfel de grupuri pe partea de date vom avea, $4 \cdot 6 = 24$ biţi. Numărul total al biţilor de control pentru cache-ul din fig. 3.10. este $k = k1 + 4 \cdot k2 = 4 + 4 \cdot 6 = 28$ biţi. Concluziv, pentru cache-ul prezentat în fig. 3.10., numărul total al biţilor utilizaţi este de k+ u = 28 + 135 = 163 biţi.

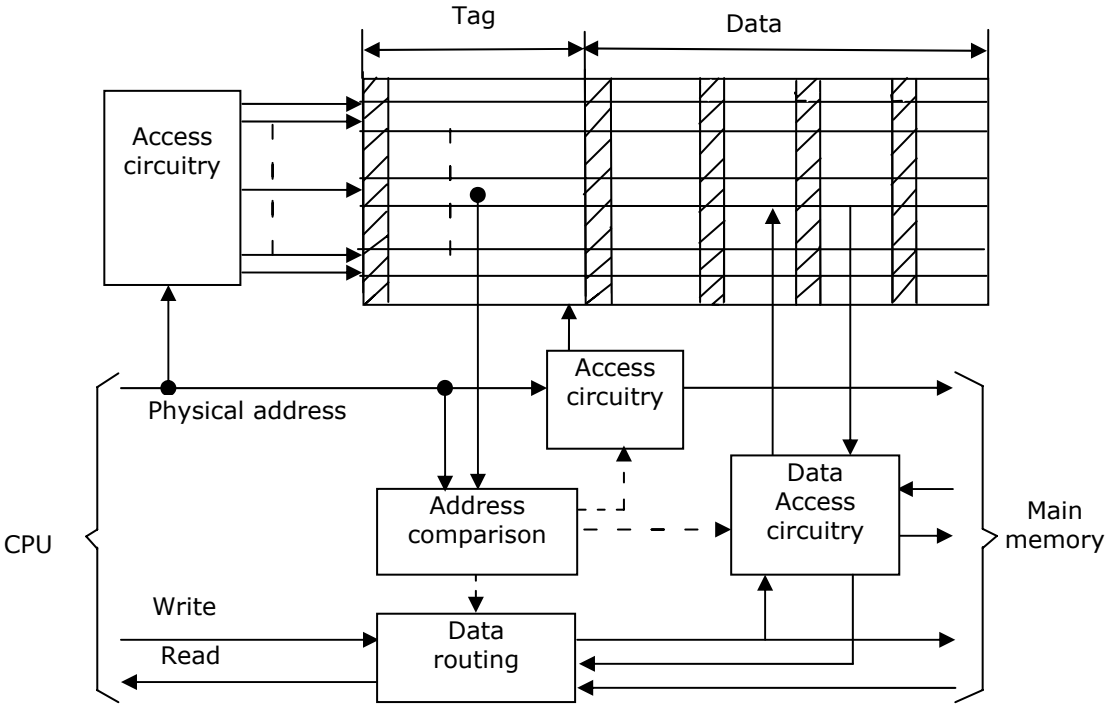


Fig.3.10.

În fig. 3.11. este prezentat din nou un cache cu partea redundantă introdusă atât pe partea de date, cât şi pe partea de TAG a cache-ului, diferenţa între cele două impementări, rezultă din implementarea pe partea de date.

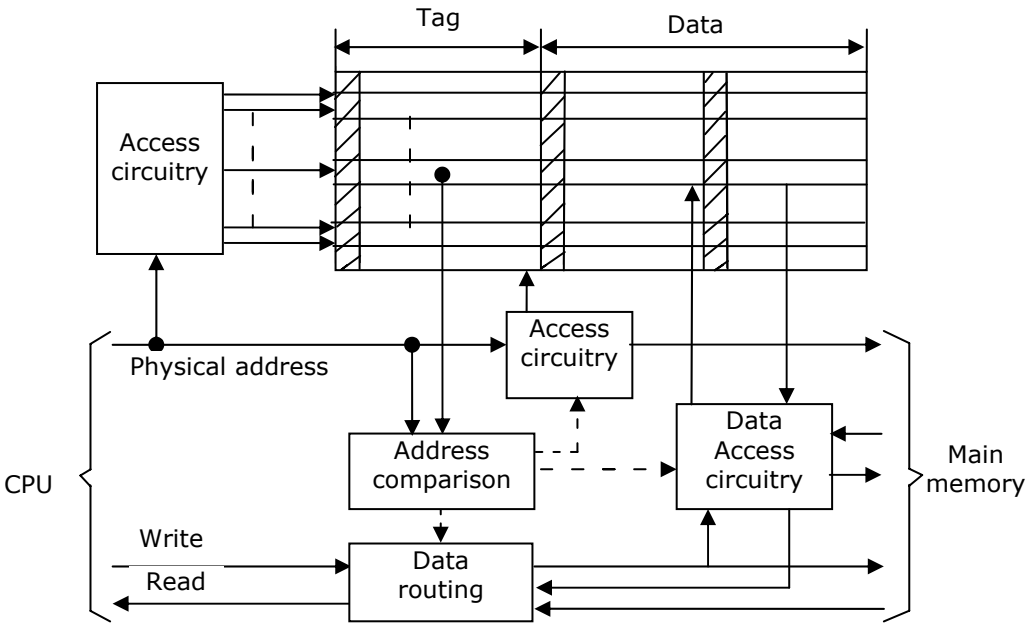


Fig. 3.11.

Pentru partea de TAG avem k1= 4 biţi. Pentru partea de date, avem 128 de biţi utili pe care îi împărţim în 2 grupuri de câte 64 de biţii fiecare, şi pentru fiecare grup vom calcula numărul biţilor de control. Pentru un astfel de grup vom avea nevoie de k2 = 7 biţi, dar deoarece sunt 2 astfel de grupuri pe partea de date vom avea, $2 \cdot 7 = 14$ biţi. Numărul total al biţilor de control pentru cache-ul din fig. 3.12. este $k = k1 + 2 \cdot k2 = 4 + 2 \cdot 7 = 18$ biţi.

Concluziv, pentru cache-ul prezentat în fig. 3.11, numărul total al biţilor utilizaţi este de k+ u = 18 + 135 = 153 biţi.

Am prezentat trei posibilităţi de introducere a părţii redundante. Am obţinut în cazul primei implementări un număr total de 143 biţi, La această implementare, prezentată în fig. 3.9., întrucât biţii de control sunt comuni la cele 4 cuvinte de 32 de biţi, şi noi lucrăm pe 32 de biţi, trebuie să citim întregul bloc (4·32 biţi şi partea de TAG) să recalculăm biţii de control şi pe urmă să rescriem informaţia, într-un ciclu Read Modify Write, care este cunoscut ca soluţie tehnică din alte contexte.

Dacă notăm cu timpul t_1, timpul de acces, (la citire), cu t_2 timpul de evaluare a biţilor de control şi cu t_3 timpul de acces (la scriere), vom avea, pentru situaţia din fig. 3.9, un interval de timp Δt, care se calculează ca suma între aceşti timpi, $\Delta t = t_1 + t_2 + t_3$. Logica de evaluare a biţilor de control este formată dintr-un

arbore de circuite SAU-EXCLUSIV, cu 143 de intrări, şi în consecinţă timpul t_2, este mult mai mare. Acest lucru determină ca şi intervalul Δt să fie mare.

Din acest motiv această soluţie este mai lentă, deoarece chiar dacă memoria are o capacitate mică adăugată în plus, trebuie să calculăm de fiecare dată, la fiecare ciclu de acces la memorie, biţii de control. Acest lucru este costisitor din punct de vedere al timpului de calcul. Partea redundantă de memorie, care se adaugă pe partea de date are dimensiunea k x nr. de linii (blocuri) din cache, adică 8 · 8192 = 8 · 1024· 8 biţi = 8 · 1K · 1B = 8 KB. Adăugăm această parte redundantă la un cache fară cod corector şi obţinem un cache redundant de dimensiune (8 + 135) KB = 143 KB. Overhead-ul datorat celulelor de memorie este de (143 - 135) / 135 = 5,92%. Overhead-ul datorat circuitelor suplimentare pentru corecţia erorii (porţi SI, porţi inversoare, porţi SAU-EX, porţi SAU) este neglijabil şi are valori între 1 - 2%.

În cazul implementări prezentată în fig. 3.10, numărul total al biţilor de control este k = 28 (4 biţi pe partea de TAG şi 24 biţi pe partea de date), iar numărul total al biţilor utilizaţi în acest caz este 163. Din punct de vedere al timpului nu există diferenţe deoarece biţii de control sunt evaluaţi pentru fiecare bloc. Partea redundantă de memorie, care se adaugă pe partea de date are dimensiunea k x nr. de linii (blocuri) din cache, adică 28 · 8192 = 28 · 1024 · 8 biţi = 28 · 1K · 1B.= 28 KB. Adăugăm această parte redundantă la un cache fară cod corector şi obţinem un cache redundant de dimensiune (28 + 135) KB = 163 KB. Overhead-ul datorat celulelor de memorie este de (163 - 135) / 135 = 20,74%. Putem spune că această implementare este mai rapidă decât implementarea din fig. 3.9 dar ea are un cost cu 20,74 % mai mare.

Overhead-ul datorat circuitelor suplimentare pentru corecţia erorii (porţi SI, porţi inversoare, porţi SAU - EX, porţi SAU) este neglijabil şi are valori între 1 - 2%.

În cazul implementări prezentată în fig. 3.11, numărul total al biţilor de control este k = 18 (4 biţi pe partea de TAG şi 14 biţi pe partea de date), iar numărul total al biţilor utilizaţi în acest caz este 153. Din punct de vedere al timpului nu există diferenţe deoarece biţii de control sunt evaluaţi pentru fiecare bloc. Partea redundantă de memorie, care se adaugă pe partea de date are dimensiunea k x nr. de linii (blocuri) din cache, adică 18 · 8192 = 18 · 1024 · 8 biţi = 18 · 1K · 1B= 18 KB. Adăugăm această parte redundantă la un cache fară cod corector şi obţinem un cache redundant de dimensiune (18 + 135) KB = 153 KB. Overhead-ul datorat celulelor de memorie este de (153 - 135) / 135 = 13,33%. Putem spune că această implementare este mai rapidă decât implementarea din fig. 3.9, dar ea are un cost cu 13.33 % mai mare.

Overhead-ul datorat circuitelor suplimentare pentru corecţia erorii (porţi SI, porţi inversoare, porţi SAU-EX, porţi SAU) este neglijabil şi are valori între 1-2%.

3.5. CONCLUZII

În acest capitol am realizat interfaţarea unui cache cu celelalte trepte ale unei ierarhii de memorii. În acest sens, fără a pierde din generalitate, am adoptat caracteristici de capacitate redusă pentru cache şi memoria principală. Am realizat proiectarea ordonată pe etape a unui cache, realizând schema bloc a cache-ului, schemă care conţine şi semnalele care trebuiesc activate la citirea sau scrierea din/în memorie. Am prezentat de asemenea împărţirea cache-ului în cele două părţi: memoria cache TAG şi memoria cache RAM. Am realizat sincronizarea funcţională a treptei cache cu celelalte trepte ale ierarhiei, utilizând diagrame de timp. Prin intermediul acestora, am prezentat ordinea desfăşurării în timp a evenimentelor în ierarhia de memorii. Am prezentat toate cele patru cicluri în care se poate găsi cache-ul: hit la citire (read-hit), miss la citire (read-miss), hit la scriere (write-hit) şi miss la scriere (write-miss). Am configurat mai multe structuri de cache, urmărind stabiliriea locului de introducere a elementelor de redundanţă menite a creşte dependabilitatea. În acest sens am implementat un cache cu partea redundantă introdusă pe partea de date, şi două implementări de cache cu partea redundantă introdusă atât pe partea de date cât şi pe partea de TAG a cache-ului. Aceste implementări le-am realizat pentru a calcula overhead-ul datorat celulelor de memorare şi overhead-ul datorat circuitelor suplimentare pentru corecţia erorii, şi am determinat care este implentarea ce introduce elementele de redundanţă optime pentru a obţine o creştere a dependabilităţii.

4. ANALIZA METRICILOR DE PERFORMANŢĂ ŞI FIABILITATE LA CACHE-URI

4.1. ANALIZA METRICILOR DE PERFORMANŢĂ ŞI FIABILITATE

4.1.1. Metrici de performanţă

Studiul vitezei unui sistem de calcul, se face ţinând cont de punctul de vedere al unui utilizator şi de punctul de vedere al unui manager de centru de calcul. Astfel utilizatorul apreciază că acel calculator este mai rapid care realizează în timp mai scurt o sarcină de calcul, deci calculatorul cu timpul de răspuns (execuţie) cel mai mic. Apare noţiunea de timp de răspuns (latenţă) care este intervalul de timp scurs între debutul şi terminarea unui eveniment de calcul. Managerul unui centru de calcul apreciază că este mai rapid acel calculator care realizează într-un timp prestabilit mai multe sarcini (task-uri), deci este mai rapid calculatorul care are o mai mare capacitate de trecere (lăţime de bandă). Această noţiune, capacitate de trecere sau lăţime de bandă reprezintă numărul de sarcini de calcul soluţionate într-un interval de timp dat. Vom utiliza la sistemele de memorii termenii de latenţă şi lăţime de bandă.

Diferenţa între viteza CPU şi viteza memoriei principale a atras atenţia multor proiectanţi de sisteme de calcul. Formula timpului mediu de acces la memorie (AMAT-Average Memory Acces Time) ne permite un studiu asupra cache-ului. Timpul mediu de acces la memorie se calculează ca sumă dintre Hit time şi produsul dintre Miss rate şi Miss penalty, conform relaţiei 4.1 [HePa03].

Timpul mediu de acces la memorie = Hit time + Miss rate * Miss penalty

sau

$$AMAT = \text{Hit time} + \text{Miss rate} * \text{Miss penalty} \qquad (4.1.)$$

unde:

- hit = accesare rapidă a informaţiei, dacă adresa este în cache;
- miss = adresa nu este în cache, şi rezultă că se aduce în cache;
- Hit time = timpul atingerii unui succes în memoria cache;

- Miss rate = este fracţia accesurilor care nu sunt în cache;
- Miss penalty = timpul adiţional pentru servirea miss-ului.

Evenimentele dintr-un calculator sunt exprimate în termeni discreţi daţi de un tren de impulsuri tact a cărui frecvenţă este constantă [NoNo99].

Se poate defini timpul CPU time, ca fiind timpul revendicat CPU pentru a executa un task (program), acest CPU time poate fi calculat în două moduri. CPU time este egal cu produsul dintre perioada de tact şi numărul de cicluri de tact CPU pentru un program.

CPU time = cicluri de tact CPU pentru un program * perioada tactului

Un alt mod de calcul al lui CPU time, CPU time este egal cu raportul dintre numărul de cicluri de tact pentru un program şi frecvenţa tactului [HePa03].

CPU time = cicluri de tact CPU pentru un program / frecvenţa tactului

Mărimea instruction count (IC), reprezintă numărul de instrucţii executate pentru un program. Dacă cunoaştem numărul de perioade de tact şi instruction count, putem calcula numărul mediu de perioade de tact pe instrucţiune CPI (Clock cycle Per Instruction).

Mărimea CPI, se calculează ca fiind egală cu raportul dintre ciclurile de tact CPU pentru un program şi instruction count.

CPI= cicluri de tact CPU pentru un program / instruction count

Din această formulă se poate observa că ciclurile de tact CPU pentru un program sunt egale cu produsul dintre CPI şi instruction count (IC). Dacă înlocuim în formula lui CPU time obţinem două relaţii de calcul pentru CPU time:

CPU time = instruction count * CPI * perioada tactului

Sau

CPU time = instruction count * CPI / frecvenţa tactului.

Performanţa CPU depinde prin prisma termenului CPU time de trei factori: instruction count, numărul mediu de perioade de tact pe instrucţiune (CPI) şi perioada tactului. Putem spune că CPU time este dependentă în mod egal de aceşti trei termeni, deci dacă realizăm o îmbunătăţire în oricare din aceşti trei termeni acest lucru se va resimţi şi în performanţa CPU [HePa03].

Putem preciza ca şi o observaţie că corelaţia dintre oricare din factori şi CPU time nu este directă. Acest lucru este cauzat de faptul că perioada de tact depinde

de tehnologia hardware şi de organizarea sistemului de calcul, mărimea CPI depinde de organizarea sistemului de calcul şi de setul de instrucţii, iar instruction count depinde atât de tehnologia de compilare cât şi de setul de instrucţii.

Unitatea de măsură a lui CPU time este secunde pe program.

$$CPU\ \text{time} = \sum_{i=1}^{n} \left(CPI_i \cdot \text{IC}_\text{i} \right)\ \text{perioada de tact}$$

4.1.2. Metrici de fiabilitate

4.1.2.1. Mărimea MTTDL

Vom introduce mărimea durată$_L$, ca fiind vârsta datelor înainte ca aceste date să fie transferate de pe nivelul L pentru a fi scrise în nivelul L+1. Politica de transfer între niveluri reprezintă un compromis între overhead şi fiabilitate. Întârzierile mari permit ca mai multe date să fie transferate simultan, aspect care creşte eficienţa transferului. De asemenea, aceste întârzieri mari reduc fiabilitatea deoarece datele rămîn mai mult timp pe un nivel mai puţin fiabil.

Rezultă astfel necesitatea utilizării unei metricii care să măsoare fiabilitatea globală a unei ierarhii. O astfel de metrică este MTTDL (Mean Time To Data Loss), adică **timpul mediu până la pierderea datelor** [ChLo99]. Presupunem o schemă bloc de fiabilitate de tip serial a sistemului (un defect în oricare nivel duce la defectarea sistemului) şi un timp de defectare distribuit exponenţial pentru fiecare nivel. În aceste condiţii, pentru o ierarhie cu N niveluri, MTTDL se calculează cu formula (4.2):

$$MTTDL = \frac{1}{\sum_{L=1}^{N} \sum \frac{1}{MTTF_L}} \qquad (4.2.)$$

În relaţia (4.2.) apare mărimea MTTF$_L$, care reprezintă timpul mediu până la defectarea nivelului L, În formula de calcul a MTTDL-ului se poate observa faptul că, acest indicator este limitat de fiabilitatea celui mai puţin fiabil nivel.

Putem spune că MTTDL-ul este relevant doar atunci când avem o pierdere semnificativă de date. Mărimea MTTDL nu face distincţie între pierderea unei cantităţi mari de date (datorate de exemplu unei defecţiuni a discului) şi pierderea unei cantităţi mici de date (datorată unei erori de memorie). [ChCh01].

4.1.2.2. Mărimea DLR

În [ChLo99], autorii propun o altă metrică pentru evaluarea fiabilităţii unei ierarhii de memorii: data loss rate - DLR. DLR-ul se calculează ca fiind egal cu timpul mediu de acces utilizat pentru a evalua o ierarhie de performanţă. Timpul mediu de acces pentru o ierarhie de performanţă cu N niveluri se calculează ca fiind media ponderată a timpilor de acces :

$$\sum_{L=1}^{N} hitRate_L \times TimpAcces_L \tag{4.3.}$$

Putem preciza că DLR-ul unei ierarhii de fiabilitate este egal cu suma ponderată a duratelor de pierdere a datelor când un nivel cedează şi se calculează cu formula (4.4.) [ChLo99]:

$$DLR = \sum_{L=1}^{N} \frac{durata_L}{MTTF_L} \tag{4.4.}$$

Putem spune că $durata_N$ implică un caz special, şi anume atunci când nu există un nivel inferior la care se pot transfera datele. O primă metodă de a aborda această problemă este aceea de a presupune că cel mai de jos nivel are $MTTF_N = \infty$, şi utilizarea unei tehnologii de stocare ce se apropie de acest ideal (relativ la fiabilitatea altor niveluri).

O altă abordare este aceea de a atribui mărimii $durata_L$, cantitatea de date utile care s-ar pierde dacă un nivel N, ar ceda. Spre exemplu, poate doar ultimul an de date este valabil pe caseta cu bandă magnetică.

Ambele abordări conduc spre formula:

$$\sum_{L=1}^{N} \frac{f_L}{MTTF_L} \tag{4.5.}$$

unde: - f_L reprezintă valoarea datelor care s-ar pierde la căderea nivelului L.

Putem considera de exemplu că f_L este o funcţie liniară ce depinde de $durata_L$, astfel încât data nouă să aibă o pondere mai mare decât data veche.

Formula prezentată mai sus nu ţine cont de defecţiunile corelate. Dacă o defecţiune afectează mai multe niveluri (o defecţiune de acest tip este inundarea), formula trebuie să o modificăm pentru a putea contabiliza un tip de defect o singură dată. Altfel, o singură defecţiune va scădea MTTF-ul mai multor niveluri şi va afecta de mai multe ori atât MTTDL-ul cât şi DLR-ul. Modalitatea prin care putem corecta

defectele corelate, este calculul MTTDL-ului şi DLR-ului, prin adunarea tipurilor de defectare mai degrabă decât a nivelurilor de fiabilitate:

$$MTTDL = \frac{1}{\sum_{toatetipurilededefecţiuniF} \frac{1}{MTTF_F}} \tag{4.6.}$$

$$DLR = \sum_{toatetipurilededefecţiuniF} \frac{defect_F}{MTTF_F} \tag{4.7.}$$

unde: - $MTTF_F$ este timpul mediu până la apariţia unei defecţiuni de tip F şi $defect_F$ este durata de pierdere a datelor, când apare un defect F.

4.1.2.3. Mărimea CDLR

Pentru a elimina neajunsurile prezentate mai sus vom elabora un alt indicator care exprimă rata de pierdere a datelor, pornind de la ideea formulată în [ChLo99]. Acest indicator îl vom numi CDLR, (Cache Data Loss Rate) şi el reprezintă de fapt, fracţia de date pierdute în timp datorită căderilor din ierarhia de memorie [VNPN05].

Presupunem că avem o ierarhie în care apar doar defecte tranzitorii (cum ar fi cele determinate de particolele α), şi timpii de reparare (MTTR - Mean Time To Repair, $MTTR_1$, $MTTR_2$,....,$MTTR_n$) sunt nuli în acest caz. În fig. 4.1 sunt prezentaţi timpii medii până la defectare ($MTTF_i$) şi timpii de reparare ($MTTR_i$), unde i=1,2, ..,.n.

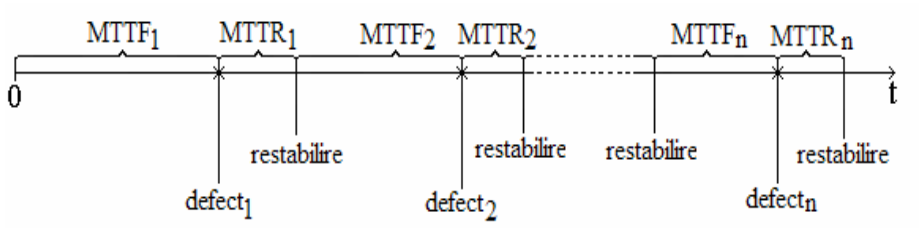


Fig 4.1

Mai presupunem că avem un acces generat de procesor, şi în acest caz vom face doar scrieri (nu şi citiri). Presupunem că se ignoră faptul că nu toate datele trebuiesc actualizate.

În continuare vom considera că T este durata totală luată în calcul.

Pentru determinarea relaţiei de calcul a CDLR-ului, ne vom folosi de următoarea ipoteză:

Numărul de pierderi în nivelul L = Numărul de pierderi produse de nivelul L + Numărul pierderi în nivelul (L+1) * [Numărul de pierderi induse în L de o pierdere în (L+1) – Numărul de pierderi deja produse de nivelul L din cele induse de (L+1)]

Acesta o vom scrie astfel :

$$N_L = \frac{T}{MTTF_L} + N_{L+1} \cdot \left(\frac{durata_{L+1}}{durata_L} - \frac{durata_{L+1}}{MTTF_L} \right) \quad (4.8.)$$

unde: - N_L este numărul de pierderi în nivelul L,

- N_{L+1} este numărul de pierderi în nivelul (L+1),

- $\frac{T}{MTTF_L}$ reprezintă numărul de pierderi produse de nivelul L,

- $\frac{durata_{L+1}}{durata_L}$ este numărul de pierderi induse în L de o pierdere în (L+1)

- $\frac{durata_{L+1}}{MTTF_L}$ este numărul de pierderi deja produse de nivelul L din cele induse de (L+1).

În continuare vom defini rata de pierdere a datelor în nivelul L, $CDLR_L$ [NVVN06], ca fiind egală cu raportul dintre numărul de pierderi în nivelul L şi numărul de mesaje a nivelului L, aşa cum se poate observa şi în relaţia (4.9.).

$$CDLR_L = \frac{N_L}{Nr.\_mesaje\_nivel\_L} \quad (4.9)$$

Numărul de mesaje a nivelului L, poate fi definit ca fiind egal cu raportul dintre durata totală luată în calcul, T, şi timpul cât o dată (un mesaj) rămâne pe nivelul L, mărime notată în continuare cu $durata_L$.

$$Nr.\_mesaje\_nivel\_L = \frac{T}{durata_L} \quad (4.10)$$

deci, dacă înlocuim numărul de mesaje din relaţia 4.10. în 4.9., obţinem pentru rata de pierdere a datelor relaţia 4.11:

$$CDLR_L = \frac{N_L}{\frac{T}{durata_L}} \quad (4.11)$$

Din (4.11.) se poate obţine pentru numărul de pierderi în nivelul L

$$N_L = CDLR_L \cdot \frac{T}{durata_L} \tag{4.12}$$

În mod analog pentru nivelul L+1 avem :

$$N_{L+1} = CDLR_{L+1} \cdot \frac{T}{durata_{L+1}} \tag{4.13}$$

Vom înlocui N_L, respectiv N_{L+1}, din 4.12. şi 4.13 în 4.8 şi obţinem:

$$CDLR_L \cdot \frac{T}{durata_L} = \frac{T}{MTTF_L} + CDLR_{L+1} \cdot \frac{T}{durata_{L+1}} \cdot \left(\frac{durata_{L+1}}{durata_L} - \frac{durata_{L+1}}{MTTF_L} \right)$$

vom trece în membrul drept al acestei relaţii numărul de mesaje a nivelului L şi obţinem:

$$CDLR_L = \frac{durata_L}{T} \cdot \left[\frac{T}{MTTF_L} + CDLR_{L+1} \cdot \frac{T}{durata_{L+1}} \cdot \left(\frac{durata_{L+1}}{durata_L} - \frac{durata_{L+1}}{MTTF_L} \right) \right]$$

reducând T avem:

$$CDLR_L = \frac{durata_L}{MTTF_L} + CDLR_{L+1} \cdot \frac{durata_L}{durata_{L+1}} \cdot \left(\frac{durata_{L+1}}{durata_L} - \frac{durata_{L+1}}{MTTF_L} \right)$$

şi în final obţinem relaţia:

$$CDLR_L = \frac{durata_L}{MTTF_L} + CDLR_{L+1} \cdot \left(1 - \frac{durata_L}{MTTF_L} \right) \tag{4.14.}$$

Aceasta este o formulă recurentă de calcul pentru mărimea CDLR.

Această formulă nu ne permite să luăm în calcul erorile apărute ulterior într-un câmp de date deja corupt în nivelul anterior. După cum am mai menţionat, ultimul nivel este un caz special, deoarece nu are un nivel mai jos unde să transmită datele. Din acest motiv, va trebui să considerăm $MTTF_N = \infty$, care înseamnă $CDLR_N=0$ (altfel CDLR-ul întregii ierarhii ar fi 100%, şi toate datele vor fi pierdute în timp). Evaluarea performanţei şi a fiabilităţii unor ierarhii de memorii poate fi făcută acum prin utilizarea celor două mărimi MTTF şi CDLR.

4.2. METODE DE ÎMBUNĂTĂŢIRE A PERFORMANŢEI CACHE-URILOR ŞI AFECTAREA LOR PRIN CDLR

Din formula timpului mediu de acces la memorie se observă deci că avem trei termeni, care pot fi reduşi pentru a optimiza performanţa cache-urilor. Conform relaţiei 4.1 în care s-a definit AMAT vor rezulta 17 metode de optimizare a performanţelor cache-urilor, metode care sunt prezentate în fig. 4.2.

Aceste metode pot fi împărţite în patru categorii [HePa03]:

- reducerea lui miss penalty;
- reducerea lui miss rate;
- reducerea miss penalty-ului sau miss rate-ului prin intermediul paralelismului;
- reducerea hit time-lui.

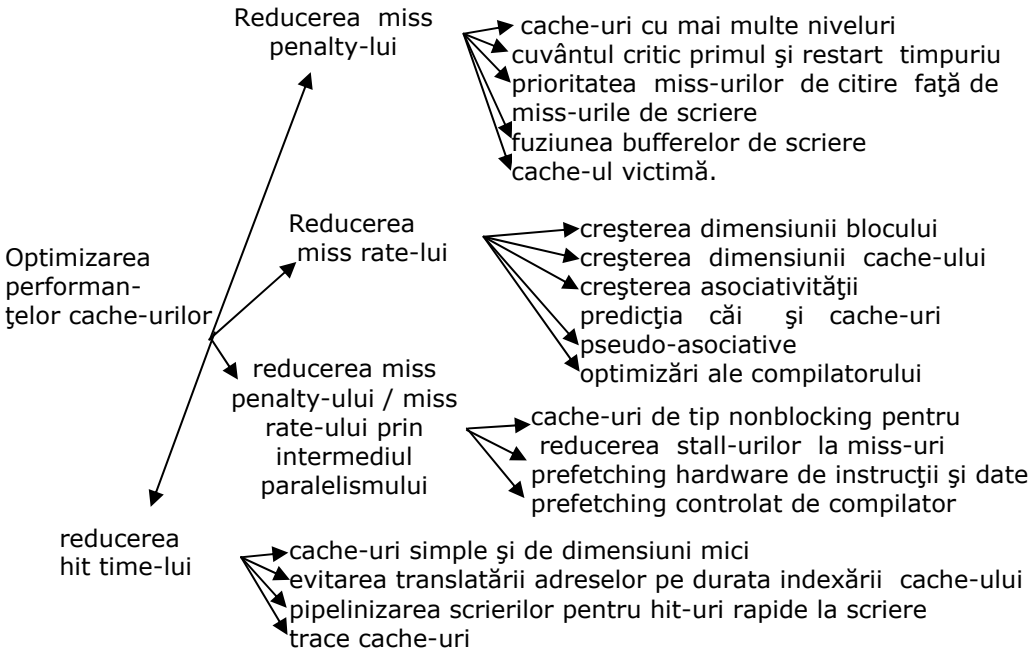


Fig. 4.2.

Propun o altă clasificare, în care sunt prezentate 18 metode de optimizare a performanţelor cache-urilor, clasificare care ţine cont de mărimea nou introdusă CDLR. Această nouă clasificare este prezentată în fig. 4.3.

Tehnicile prezentate mai detaliat în fig. 4.3, îmbunătăţesc miss rate, miss penalty şi hit time-ul şi au impact asupra unei componente sau a altei componente din formula timpului mediu de acces la memorie, dar au efect şi asupra complexităţii ierarhiei de memorii.

Vom prezenta cumulat în tabelul 4.1., toate cele 18 tehnici şi vom nota cu „+" dacă tehnica respectivă îmbunătăţeşte factorul, cu „-" dacă tehnica afectează în sens negativ factorul, iar „spaţiu" înseamnă că tehnica nu are nici un efect asupra factorului.

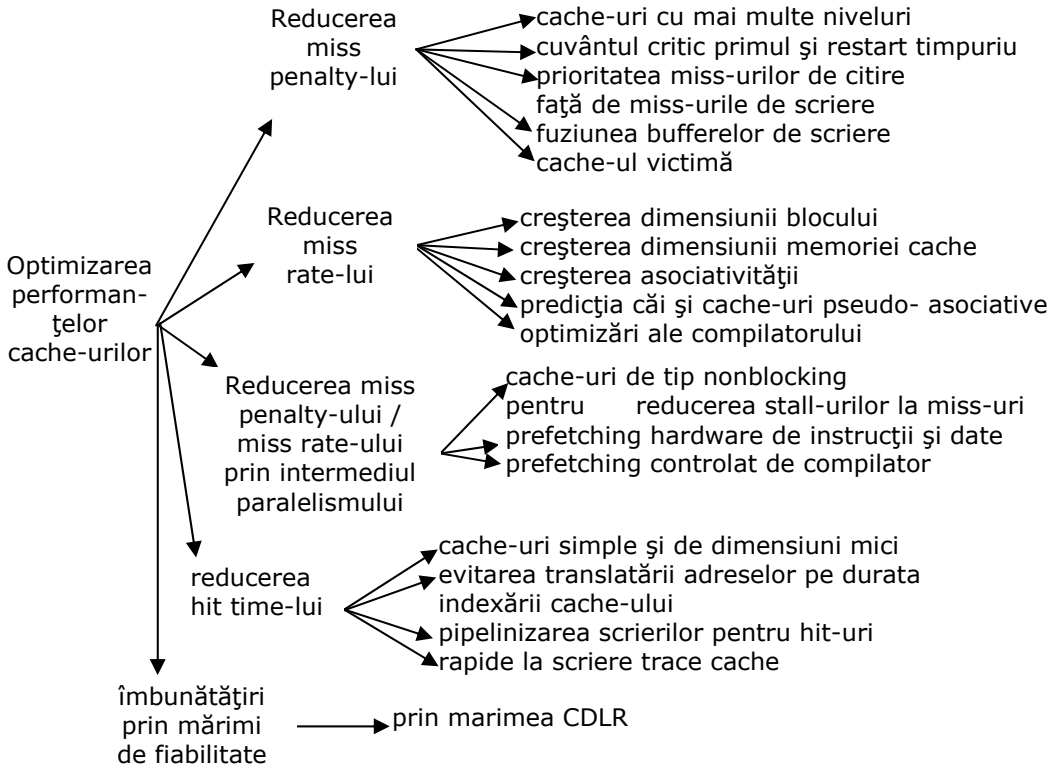


Fig. 4.3.

Tehnica	Miss penalty	Miss rate	Hit time	Complexitate Hardware
Cache-uri cu mai multe niveluri	+			2
Cuvântul critic primul şi restart timpuriu	+			2
Prioritatea miss-urilor de citire faţă de miss-urile de scriere	+			1
Fuziunea bufferelor de scriere	+			1
Cache-ul victimă	+	+		2
Creşterea dimensiunii blocului	-	+		0
Creşterea dimensiunii cache-ului		+	-	1
Creşterea asociativităţii		+	-	1
Predicţia căi şi cache-uri pseudo-asociative		+		2
Optimizări ale compilatorului		+		2
Cache-uri de tip nonblocking pentru reducerea stall-urilor la miss-uri	+			3
Prefetching hardware de instrucţii şi date	+	+		2 instr, 3 date
Prefetching controlat de compilator	+	+		3
Cache-uri simple şi de dimensiuni mici		-	+	0
Evitarea translatării adreselor pe durata indexării cache-ului			+	2
Pipelinizarea scrierilor pentru hit-uri rapide la scriere			+	1
Trace cache-uri			+	3
Îmbunătăţiri prin mărimea CDLR			+	1

Tabelul 4.1.

Majoritatea cercetărilor referitoare la optimizarea performanţelor cache-urilor s-au concentrat pe reducerea miss rate-ului pentru a obţine rezultate mai bune referitoare la acest tip de miss, astfel prezentăm un model care sortează toate miss-urile, în trei categorii simple şi anume [HePA03]:

– de constrângere – în acest caz primul acces la un bloc nu este în cache, astfel blocul va trebui adus în cache, aceste miss-uri se mai numesc miss-uri cu primă referinţă.

– de capacitate – dacă memoria cache nu poate conţine toate blocurile necesare în timpul execuţiei unui program, miss-urile de capacitate vor apărea datorită blocurilor care au fost ignorate şi ulterior s-au recuperat.

– de conflict – dacă strategia de plasare a blocurilor este set asociativă sau mapată direct, aceste miss-uri vor apărea deoarece un bloc poate fi ignorat şi ulterior recuperat dacă prea multe blocuri mapează setul acestuia. Aceste miss-uri se mai numesc miss-uri de coliziune sau miss-uri de interferenţă.

4.3. CONCLUZII

Am prezentat în acest capitol metrici de performanţă şi metrici de fiabilitate care sunt utilizate pentru studiul cache-urilor.

Am elaborat în cadrul studiului metricilor de fiabilitate, o nouă mărime, denumită CLDR (Cache Data Loss Rate). Cu ajutorul acestei mărimi se poate face evaluarea fiabilităţii unor ierarhii de memorii.

Am propus o nouă clasificare a metodelor de optimizare a performanţelor memoriilor cache, clasificare ce ţine cont de mărimea nou introdusă CDLR. De asemenea, am prezentat cumulat într-un tabel toate tehnicile de optimizare a performanţelor memoriilor cache, unde am prezentat şi faptul că tehnica respectivă îmbunătăţeşte sau nu termenul din formula timpului mediu de acces la memorie. în acest tabel am introdus şi îmbunătăţirile prin mărimea CDLR. În concluzie putem spune că locul optim pentru introducerea redundanţei este cel al cache-ului.

5. MEDIUL DE EXPERIMENTARE REPREZENTAT DE SIMULATORUL DE CACHE

5.1. OBIECTIVELE DE CONSTRUCŢIE ALE SIMULATORULUI

5.1.1. Obiectivul mediului experimental CACHE

Obiectivul mediului experimental CACHE este simularea funcţionării unei ierarhii de memorie. Orientându-mă după literatura de specialitate din acest domeniu [HePa03], [HePa06], precum şi [VeSG01] am considerat că cei mai importanţi parametri folosiţi la construcţia unui simulator sunt următorii: dimensiunea memoriei principale, numărul de niveluri de cache şi dimensiunea fiecărui nivel de cache, dimensiunea blocului din cache, funcţia de mapare, politicile de scriere şi algoritmul de înlocuire. Un alt parametru important care va fi folosit în cadrul simulării este fişierul trace.

Pentru a realiza această simulare trebuie în primul rând să configurăm ierarhia de memorie. În acest sens vom stabili dimensiunea memoriei principale, numărul de niveluri de cache, dimensiunea blocului din cache, selecţia tipului de mapare al memoriei cache, selecţia politicii de scriere şi selecţia algoritmului de înlocuire. În funcţie de numărul de niveluri de cache selectate se stabileşte dimensiunea fiecărui nivel de cache. Vom stabili în final şi fişierul trace care va fi folosit în cadrul simulării.

Un obiectiv al programului este trasarea unor grafice ce reprezintă hit rate şi miss rate pentru fiecare nivel de cache, precum şi hit rate-ul şi miss rate-ul global.

Acest simulator este construit pentru a verifica funcţionarea unei ierarhii de cache-uri cu toleranţă la defectare obţinută prin coduri detectoare şi corectoare de erori. Simulatorul l-am construit având ca model simulatorul SMP Cache 2.0. [VeSG01], prezentat în anexa E. Simulatorul nostru CACHE, are un dezavantaj şi anume faptul că el nu ţine cont şi de indicii performanţă şi fiabilitate ai cache-urilor[NoNo02].

5.1.2. Obiectivul mediului de evaluare CDLR SPEC 2000

CDLR SPEC 2000 este tot un simulator bazat pe traces-uri pentru ierarhii de cache-uri. Mediul nu are o interfaţă grafică, execuţia fişierelor se face din fereastra DOS. Obiectivul mediului de evaluare CDLR SPEC 2000 este studiul comportării memoriilor cache prin simularea funcţionării unei ierarhii de memorii şi calculul CDLR-ului global (CDLR-ul ierarhiei de memorii). Spre deosebire de simulatorul CACHE, mediul de evaluare ţine cont şi de indicii de performanţă şi fiabilitate prin calculul mărimii CDLR.

Pentru a putea realiza simularea, vom configura mai întâi ierarhia de memorii. În acest sens vom stabili: funcţia de mapare, dimensiunea blocului, politicile de scriere, algoritmul de înlocuire, dimensiunea memoriei cache, numărul seturilor de cache (pentru cache-urile set asociative), numărul cuvintelor dintr-un bloc. În continuare trebuie să specificăm pentru fiecare nivel din care este alcătuită ierarhia noastră următorii parametri: dimensiunea memoriei, MTTF-ul, timpul de fetch, timpul de citire, timpul de scriere şi miss penalty. După ce au fost specificaţi toţi parametri se încarcă un fişier trace, fişier pe care îl alegem dintre fişierele de tip trace ale benchmark-urilor SPEC 2000. Obiectivul final al simulatorului nostru este calculul CDLR-ului global (CDLR-ul ierarhiei de memorii) precum şi calculul MTTF-ului global al ierahiei.

5.2. DETALII CONSTRUCTIVE

5.2.1. Detalii constructive ale simulatorului – CACHE

Mediul de simulare Cache este scris cu ajutorul mediului de programare Microsoft Visual C++. În fig. 5.2. este prezentată interfaţa principală a programului de simulare Cache [VaNo03].

Programul permite configurarea ierarhiei de memorie. Prima setare care se poate face este alegerea dimensiunii memoriei principale. Memoria principală poate lua valori de la 256 KB până la 1048576 KB. Următorul parametru al programului ce trebuie configurat este dimensiunea blocului, dimensiume care poate fi de: 1 B, 2 B, 4 B sau 8 B.

Un alt parametru ce trebuie configurat este numărul de niveluri ale ierarhiei de memorie. Ierarhia poate avea: una, două, trei sau patru niveluri de cache. Tipul mapării este un alt parametru ce trebuie setat. Maparea poate fi astfel: directă, set asociativă sau complet asociativă. Următorul parametru ce trebuie setat este algoritmul de înlocuire.

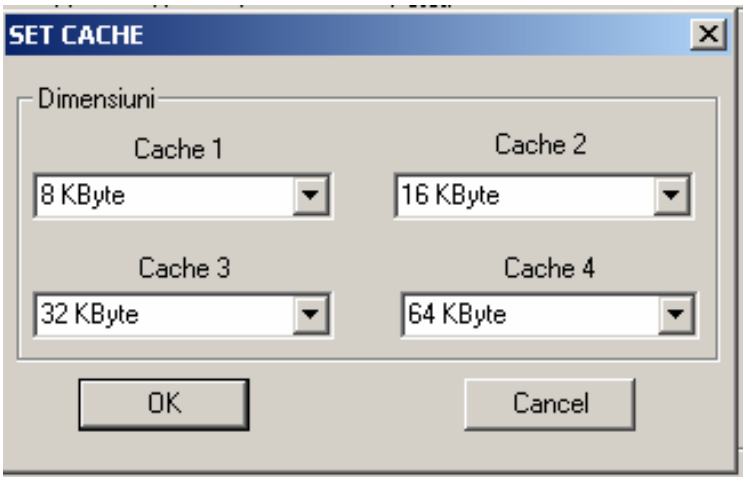


Fig. 5.1.

Agoritmul de înlocuire se poate alege dintre LRU, LFU, FIFO sau RANDOM. Politica de scriere este un alt parametru ce trebuie configurat. Putem alege pentru politica de scriere, fie politica write through, fie politica write back. Dacă numărul de niveluri de cache ales este, de exemplu, patru trebuie să acţionăm butonul Setup Cache.

Fig. 5.2.

Efectul apăsării butonului Setup Cache este deschiderea ferestrei de setare a dimensiunilor celor patru niveluri de cache, fereastră prezentată în fig. 5.1. Dimensiunea unui nivel se poate alege între 8 KB şi 32768 KB, cu observaţia că suma dimensiunilor celor patru niveluri de cache nu trebuie să depăşească dimensiunea memoriei principale. După efectuarea acestei selecţii apare în plus în fereastra principală butonul Simulare, aşa cum se observă în fig. 5.3.

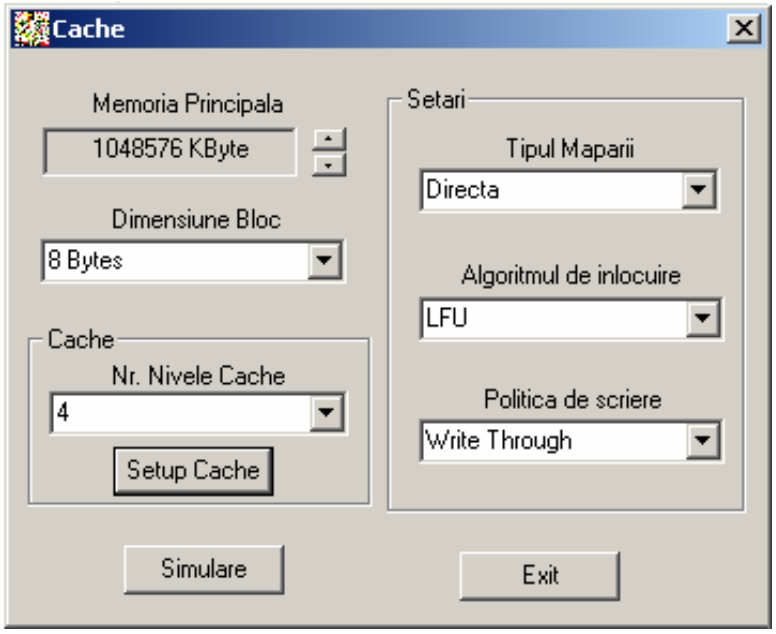


Fig. 5.3.

După apăsarea butonului Simulare se intră într-o altă fereastră unde trebuie să selectăm unul din traces-urile disponibile. Această fereastră este prezentată în fig. 5.4.

În exemplul din figură selecţia traces-ului se face acţionând butonul Browse. După apăsarea acestui buton se deschide o fereastră Open ce cuprinde lista fişierelor de tip trace. În cazul nostru această listă cuprinde următoarele fişiere: CEXP.PRG, COMP.PRG, EAR.PRG, HYDRO.PRG, MDLJD.PRG, NASA7.PRG, SWM.PRG, UCOMP.PRG şi WAVE.PRG [BYUC01].

Codurile sursă ale programelor: Cache.cpp, CacheDlg.cpp, CacheDimensiune.cpp, MemCache.cpp şi Simulare.cpp sunt prezentate în anexa F. Vom prezenta în continuare ordinogramele programelor ce alcătuiesc mediul experimental CACHE.

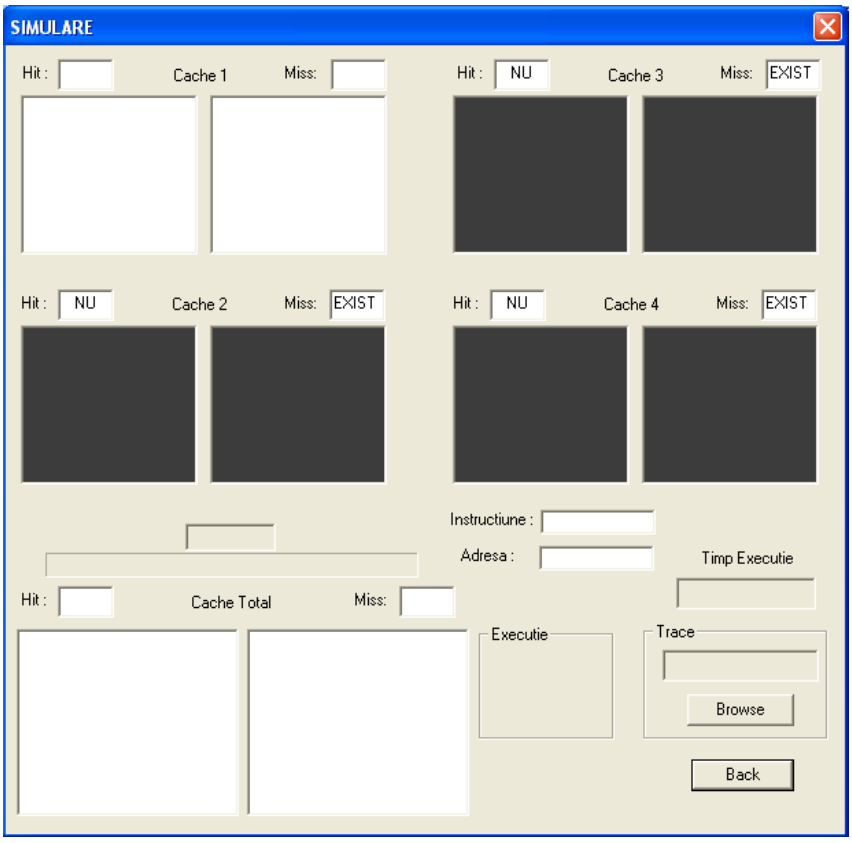


Fig. 5.4.

Primul program, pe care-l prezentăm este Cache.cpp. În fig 5.5. se prezintă ordinograma funcţiei CCacheApp, care este funcţia ce este prezentată în fişierul Cache.cpp. În primul rând se defineşte clasa CCacheApp şi comportarea ei pentru aplicaţia noastră.

Urmează definirea clasei de dialog CCacleDlg şi acest dialog devine programul principal. Se deschide dialogul nResponse==IDOK şi se verifică dacă variabila Response este OK. Dacă nu este OK se verifică dacă variabila Response este CANCEL şi în ambele variante funcţia ne va returna un FALSE, acesta reprezentând ieşirea din aplicaţie.

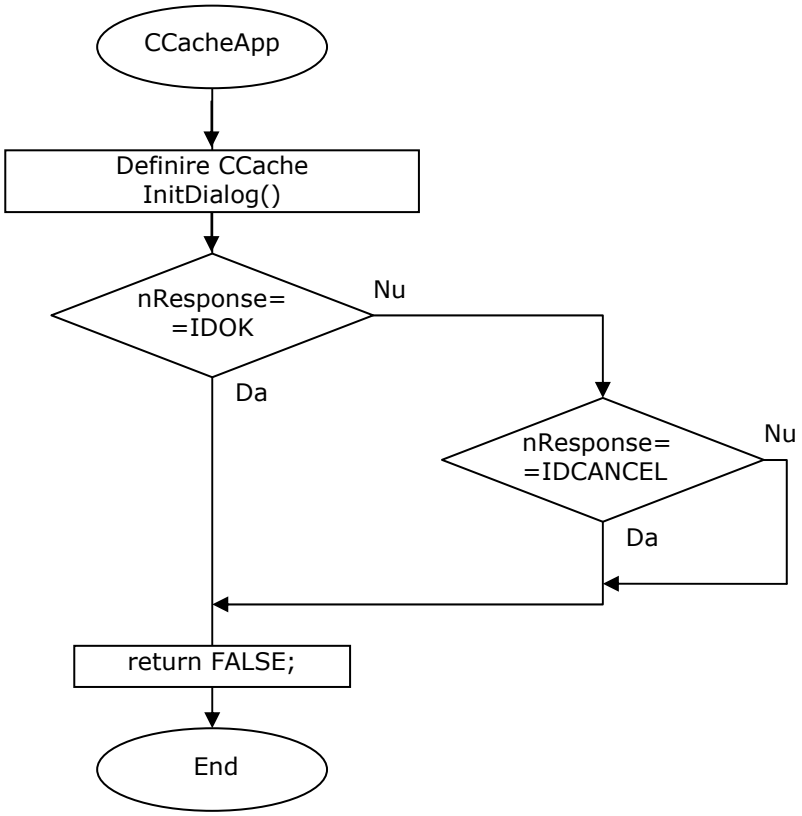


Fig. 5.5

Al doilea program, pe care-l prezentăm este CacheDlg.cpp. În fişierul CacheDlg.cpp sunt prezentate pe lângă funcţia principală CCacheDlg şi alte funcţii (DoDataExchange(), OnInitDialog(), OnSysCommand(), OnPaint(), OnSpinChange()). Funcţia DoDataExchange() este o funcţie de legătură între controalele MFC şi variabilele care conţin parametrii simulatorului. Funcţia OnInitDialog() defineşte meniul About. Funcţia OnSysCommand() preia comenzile de la meniu la fereastra About şi crează dialogul About. Funcţia OnPaint() efectuează centrarea ferestrei de dialog. Funcţia OnSpinChange() este funcţia care face selecţia dimensiunii memoriei principale. În fig 5.6. se prezintă ordinograma funcţiei CCacheDlg.

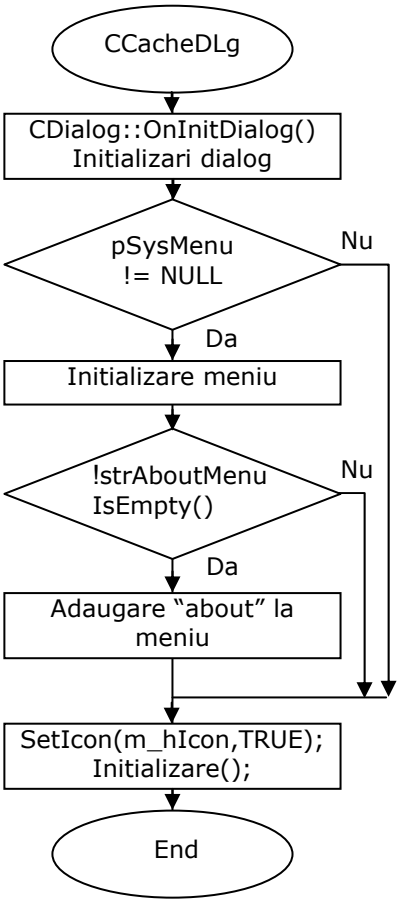


Fig. 5.6. Ordinograma funcţiei CCacheDlg()

În acest subcapitol am prezentat doar o parte din ordinogramele ce descriu construcţia simulatorului CACHE, cealaltă parte este prezentată în anexa B.

La apăsarea butonului Setup Cache se intră în funcţia OnSetupCache(). Se defineşte Dialogul CCacheDimensiune, după care se preia valoarea ce a fost selectată pentru numărul de niveluri din care este alcătuită ierarhia. Se execută DoModal() şi se testează dacă ddl.DoModal() are valoarea 1. Se deschide dialogul şi la ieşirea din dialog se face preluarea variabilei din clasă. În final se face reafişarea dialogului Setup Cache.

În primul rând se iniţializează variabilele de dialog. Se aplelează funcţia extinsă CDialog: OnInitDialog(). Se defineşte şi se adaugă meniul About după care urmează definirea icoanei dialogului şi afişarea acestei ferestre. În final se apelează funcţia de iniţializare care crează listele de opţiuni pentru utilizator. O funcţie

importantă care se găseşte în fişierul CacheDlg.cpp este OnSetupCache(). Ordinograma funcţiei OnSetupCache() este prezentată în fig. 5.7.

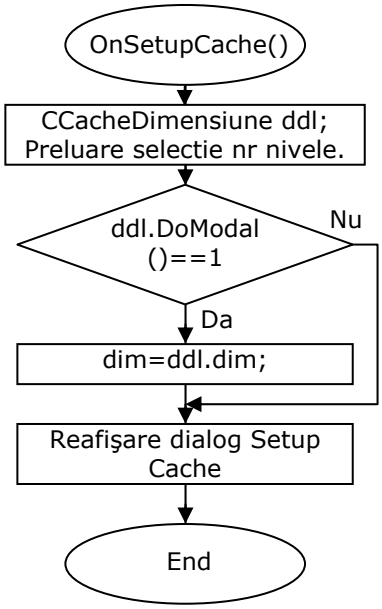


Fig. 5.7. Ordinograma funcţiei OnSetupCache()

5.2.2. Detalii constructive ale mediului de evaluare CDLR SPEC 2000

CDLR SPEC 2000 este un mediu pentru evaluarea indicilor de performanţă / fiabilitate la memoriile cache. Acest mediu de evaluare l-am elaborat utilizând iniţial sistemul de operare Windows. La rularea mediului în acest sistem de operare au apărut o serie de probleme. Una din probleme este timpul foarte mare de calcul, datorită faptului că mediul CDLR SPEC 2000 foloseşte traces-uri ale benchmark-urilor SPEC 2000, care au dimensiuni foarte mari de ordinul sutelor de Mocteţi sau chiar Gocteţi. O altă problemă este faptul că pentru anumite configuraţii de memorii cache acest mediu se bloca. Datorită acestor aspecte am apelat la sistemul de operare LINUX, rezultând versiunea CDLR SPEC 2000 [NVVN06].

Mediul este alcătuit din următoarele programe: main.c, read.h, run.h, data.h, write.h. În anexa B sunt prezentate ordinogramele ce descriu construcţia mediului de evaluare CDLR SPEC 2000.

Am creat fişierele de intrare, iar fişierele de ieşire sunt obţinute cu ajutorul simulatorului CDLR SPEC 2000. Astfel avem fişierul de intrare - spec1win.in, unde s-a prezentat structura unei ierarhii de memorii cu două niveluri. Se observă în acest exemplu că pe primele patru linii avem configuraţi cei patru parametrii:

funcţia de mapare (asociativitate de tip 1), dimensiunea blocului (4), politica de scriere (write back) şi algoritmul de înlocuire (random). Următoarele două linii ne specifică faptul că ierarhia noastră de memorii are două niveluri, şi ne precizează paramerii fiecărui nivel. Primul nivel are dimensiunea de 1024 octeţi, MTTF = 10000000 ore, timpul de fetch=1ns, timpul de citire=3 ns, timpul de scriere=5ns şi miss penalty=10ns. Al doilea nivel are dimensiunea de 4096 octeţi, MTTF = 900000000ore, timpul de fetch = 10ns, timpul de citire = 20ns, timpul de scriere = 50ns şi miss penalti = 150ns. Pe ultima linie se încarcă trace-ul şi se poate specifica de asemenea şi arhivele trace-ului pentru care se execută simularea. În cazul nostru trace-ul pe care-l încărcăm este fft1.prg. Ieşirea programului este fişierul spec1win.out.

Avantajele simulatorului CDLR SPEC 2000.

CDLR SPEC 2000 este tot un simulator bazat pe traces-uri pentru sisteme cu memorii cache. Acest simulator poate fi instalat pe calculatoare personale ce au instalate sistemele de operare Windows sau LINUX. Cu ajutorul mediului, vom putea studia următorii parametrii: funcţia de mapare, dimensiunea blocului, politicile de scriere, algoritmul de înlocuire, dimensiunea memoriei cache, numărul seturilor de cache (pentru cache-urile set asociative), numărul cuvintelor dintr-un bloc. Acest simulator poate fi folosit pentru studiul comportării memoriilor cache. De asemenea, programul nostru, introduce şi calculul mărimii CDLR asupra ierarhiei de memorii. Mediul nu are o interfaţă grafică, execuţia fişierelor se face din fereastra DOS.

Acest mediu este conceput, ca şi programul SMP Cache 2.0, pentru lucrul cu fişiere de tip Trace, deosebirea majoră fiind faptul că programul CDLR SPEC 2000 foloseşte traces-uri ale benchmark-urilor SPEC 2000, care au dimensiuni foarte mari de ordinul sutelor de Mocteţi sau chiar Gocteţi [SPEC00].

Parametrii programului CDLR SPEC 2000 pot fi modificaţi de către utilizator. În carul unei rulări a programului CDLR SPEC 2000 trebuie să specificăm atât parametrii de simulare cât şi contextul, acest lucru se va specifica în cadrul unui fişier de intrare (fişier ce va avea extensia .in). Astfel vom stabili funcţia de mapare (asociativitatea), memoria poate fi: mapată direct (-1), mapată complet asociativ (1) sau mapată set asociativ. Următorul parametru ce trebuie specificat este dimensiunea blocului. Al treilea parametru ce trebuie specificat este politica de scriere, aceasta poate fi: write back (1) sau write through (2). Al patrulea parametru ce trebuie specificat este algoritmul de înlocuire: LRU (1), LFU (2), Random (3) sau FIFO (4).

În continuare trebuie să specificăm pentru fiecare nivel din care este alcătuită ierarhia noastră următorii parametrii: dimensiunea memoriei, MTTF, timpul de fetch, timpul de citire, timpul de scriere şi miss penalty. După ce au fost specificaţi toţi parametri se încarcă un fişier trace, fişier pe care îl alegem dintre

fişierele de tip trace ale benchmark-urilor SPEC 2000 [NoNP06]. În continuare prezint structura unui astfel de fişier de intrare:

- asociativitatea 4
- dimensiune_bloc 16
- politica_scriere back
- algoritm_inlocuire fifo
- nivel 4096 1000000 1 3 5 10
- nivel 16384 90000000 10 20 50 150
- nivel 32768 900000000 20 40 90 300
- sbcrun swim_m2b 1 21

În exemplul de mai sus avem structura fişierului de intrare spec1.in. Se observă în acest exemplu că pe primele patru linii avem configuraţi cei patru parametrii: funcţia de mapare, dimensiunea blocului, politica de scriere şi algoritmul de înlocuire. Următoarele trei linii ne specifică faptul că ierarhia noastră de memorii are trei niveluri, precum şi care sunt parametrii fiecărui nivel. Primul nivel are dimensiunea de 4.096 octeţi, MTTF=1000000 ore, timpul de fetch = 1 ns, timpul de citire = 3 ns, timpul de scriere = 5 ns şi miss penalty = 10 ns. Al doilea nivel are dimensiunea de 16384 octeţi, MTTF= 90000000 ore, timpul de fetch = 10 ns, timpul de citire = 20 ns, timpul de scriere = 50 ns şi miss penalty = 150ns. Al treilea nivel are dimensiunea de 32768 octeţi, MTTF = 900000000 ore, timpul de fetch = 20 ns, timpul de citire = 40ns, timpul de scriere = 90ns şi miss penalty = 300 ns. Pe ultima linie se încarcă trace-ul şi se specifică de asemenea şi arhivele trace-ului între care se execută simularea.

Lansarea în execuţie a programului se realizează cu comanda:

main fişier_intrare fişier_iesire

Programul main încarcă datele pe care le-am scris din fişierul de intrare în simulator, iar rezultatele simulării le va scrie în fişierul de ieşire. Dacă folosim de exemplu, fişierul de intrare spec1.in, fişierul de ieşire, spec1.out obţinut în urma simulării va arăta astfel:

```
----------------[ LEVEL 1 ]-------------------------
MTTF Level : 1000000
Delay Average Level: 33516
Times Level: 519963

----------------[ LEVEL 2 ]-------------------------
MTTF Level : 90000000
Delay Average Level: 3034236
```

Times Level: 22642

----------[LEVEL 3]----------
MTTF Level : 900000000
Delay Average Level: 31605791
Times Level: 1680

MTTF Ierarhie: 987925
CDLR ierarhie: 6.764735
CDLR statisic: 10.349626

Simulatorul CDLR SPEC 2000 foloseşte trace-urile benchmark-urilor SPEC 2000 care au dimensiuni foarte mari. Datorită acestor dimensiuni foarte mari ale trace-urilor (de exemplu 10 Gocteţi) este necesară utilizarea unui algoritm de compresie specific trace-urilor, care trebuie să reducă atât dimensiunea trace-ului cât şi timpul de simulare. Astfel am identificat două condiţii:

- reducerea semnificativă a dimensiunii trace-ului;
- reducerea timpului de simulare, pentru a obţine rezultate cât mai bune.

Simulatorul foloseşte algoritmul SBC (Stream-Based Compression). Acesta este un algoritm de compresie a traces-urilor de adrese şi instrucţii, şi se bazează pe extragerea aşa numitelor fluxuri de instrucţii (instruction streams). Un flux de instrucţii reprezintă o rulare a unor instrucţiuni secvenţiale, acesta fiind situat între o ramificaţie ţintă şi prima ramificaţie care apare în secvenţa de instrucţii. O tabelă a fluxului de instrucţii (stream table) va păstra informaţii importante despre fluxurile de instrucţii: lungimea fluxului de instrucţii, cuvintele de instrucţii şi tipul instrucţiei. Toate instrucţiile dintr-un flux sunt înlocuite prin indexul lor în tabela fluxului de instrucţii, creând astfel un trace al fluxurilor de instrucţii [MiMi03].

Algoritmul utilizat de noi, are o foarte bună rată de compresie, dar şi un timp foarte bun de decompresie atât pentru traces-urile de instrucţii cât şi pentru traces-urile adrese de date. Acesta poate fi uşor implementat cu ajutorul unor limbaje de programare de nivel înalt, acest algoritm fiind implementat în limbajul C, el putând fi combinat cu algoritmi de compresie pentru reducerea dimensiunii traces-urilor [MiMi03].

Algoritmul se bazează pe utilizarea următoarelor trei fişiere: fişierul STF (Stream Table File), fişierul SBIT (Stream Based Instruction Trace) şi fişierul SBDT (Stream Based Data Trace). [MiMi03].

5.3. CONCLUZII

Pornind de la referinţele bibliografice şi de la simulatorul de cache SMP Cache 2.0, prezentat în anexa E, am elaborat un simulator de CACHE. La începutul acestui capitol am precizat care sunt obiectivele mediului experimental CACHE. De asemenea, am precizat care sunt obiectivele mediului de evaluare CDLR SPEC 2000. Deoarece simulatorul CACHE avea un dezavantaj major şi anume faptul că el nu ţine cont de indicii de performanţă şi fiabilitate ai cache-urilor, am construit un alt simulator mai performant, numit CDLR SPEC 2000, care ţine cont de aceşti indici. Acest capitol mai cuprinde şi detaliile constructive ale simulatorului CACHE şi a mediului de evaluare CDLR SPEC 2000, Ambele medii de evaluare sunt construite prin intermediul unor programe care sunt prezentate prin ordinograme

6. REZULTATE EXPERIMENTALE OBŢINUTE PRIN INTERMEDIUL SIMULATORULUI CDLR SPEC 2000

6.1. IMPLEMENTAREA CU CIRCUITE PROGRAMABILE FPGA XILINX A UNUI COD SEC-DED

6.1.1. Alegerea tipului de cod

În sistemele de calcul, la ierarhiile de memorii se pot aplica cu succes mai multe coduri de corecţie a erorilor. Aceste coduri de corecţie şi detecţie a erorii sunt adăugate la memorii cu scopul de a îmbunătăţii fiabilitatea [PIVN04], [PVND05] ,[PVNI06], [PoVN07], [PoNV07].

O trăsătură importantă a codurilor pentru memorii foarte rapide îl constituie faptul că se solicită o codificare şi decodificare paralelă pentru menţinerea unor rate înalte a capacităţii de trecere. Prin urmare, codificarea şi decodificarea schemelor, sunt implementate cu ajutorul schemelor combinaţionale în locul registrelor de deplasare liniare cu reacţie. În memoriile foarte rapide, cele mai utilizate coduri sunt codurile corectoare a erorilor singulare de bit şi detectoare a erorilor duble de bit (numite şi coduri SEC-DED – Single bit Error Correcting and Double bit Error Detection) [RaFu89]. Aceste coduri sunt utilizate deoarece multe chipuri semiconductoare RAM sunt organizate pentru un bit de date la ieşire la un moment dat şi prin urmare orice defecţiune în chip se manifestă ca şi o eroare de bit [Imai90]. Am ales acest cod Hsiao, datorită proprietăţilor sale. O proprietate importantă a acestui cod este numărul impar de biţi de 1 pe fiecare coloană a matricii generatoare. Această proprietate conduce la generarea unor ecuaţii echilibrate şi în consecinţă şi a unei scheme echilibrate de corecţie a erorilor. Astfel soluţia propusă prin codul Hsiao este acoperitoare pentru defecte specifice cache-urilor.

6.1.2. Aplicarea codului la memoria principală

În cazul memoriei noastre DRAM, vom folosi un cod Hsiao (13,8,5) pentru corecţia erorilor singulare şi detecţia erorilor duble. Acest cod are o matrice H în care nu avem două coloane identice. Codul Hsiao ales de noi (13,8,5), se codifică astfel: (t,u,k), unde k este numărul biţilor de control, u este numărul biţilor utili, iar t este numărul total de biţi ai codului t = k + u. În cazul nostru k = 5, u = 8 şi t = 13. Matricea H este definită prin relaţia 6.1. Pentru asigurarea corecţiei erorii singulare, între cele două mărimi, u şi k, există condiţia (4.1), $2^k > u+k+1$. În mod normal ar fi fost suficienţi un număr de k= 4 biţi de control, dar noi vom folosi un număr de k = 5 biţi de control, al cincilea bit fiind folosit pentru detecţia erorilor duble.

Un cuvânt de cod tipic $u=(c_0c_1c_2c_3c_4u_0u_1u_2u_3u_4u_5u_6u_7)$ are biţii de paritate în poziţiile 1,2,3,4 şi 5 şi biţi de date în rest (de la poziţia 6 la 13)

Am ales pentru matricea H, această amplasare a biţilor de 1 şi 0 pentru a obţine ecuaţii echilibrate pentru biţii de control. Precizez de asemenea că matricea H are pe primele cinci linii şi cinci coloane o matrice unitate, iar în continuare această matrice se construieşte din coloane cu un număr impar de biţii de 1. În acest caz particular, numărul biţilor de 1 din fiecare coloană este egal cu 3. Se observă faptul că în cazul oricărei matrici H, nu avem două coloane identice.

$$H = \begin{array}{c} \text{Poziţia} \rightarrow \\ \\ \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccccccccc} c_0 & c_1 & c_2 & c_3 & c_4 & u_0 & u_1 & u_2 & u_3 & u_4 & u_5 & u_6 & u_7 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \end{array} \quad (6.1)$$

Biţii de control se calculează prin ecuaţiile de paritate date în continuare:

$$\begin{aligned} c_0 &= u_0 \oplus u_3 \oplus u_4 \oplus u_6 \oplus u_7 \\ c_1 &= u_0 \oplus u_1 \oplus u_5 \oplus u_7 \\ c_2 &= u_0 \oplus u_1 \oplus u_2 \oplus u_3 \oplus u_6 \\ c_3 &= u_1 \oplus u_2 \oplus u_3 \oplus u_4 \oplus u_5 \\ c_4 &= u_2 \oplus u_4 \oplus u_5 \oplus u_6 \oplus u_7 \end{aligned} \quad (6.2.)$$

Decodificarea vectorului recepţionat utilizează ecuaţiile de sindrom care sunt bazate pe ecuaţiile de paritate (6.2), iar aceste ecuaţii de sindrom (6.3) se dau în continuare:

$$
\begin{aligned}
s_0 &= c_0 \oplus u_0 \oplus u_3 \oplus u_4 \oplus u_6 \oplus u_7 \\
s_1 &= c_1 \oplus u_0 \oplus u_1 \oplus u_5 \oplus u_7 \\
s_2 &= c_2 \oplus u_0 \oplus u_1 \oplus u_2 \oplus u_3 \oplus u_6 \\
s_3 &= c_3 \oplus u_1 \oplus u_2 \oplus u_3 \oplus u_4 \oplus u_5 \\
s_4 &= c_4 \oplus u_2 \oplus u_4 \oplus u_5 \oplus u_6 \oplus u_7
\end{aligned}
\qquad (6.3.)
$$

Vom aplica acest cod SEC-DED [NVPN06] la memoria RAM dinamică a sistemului ce are capacitatea de 16 MB. La citire vom citi din memoia DRAM atât biţii utili de date (u_0 u_1 u_2 u_3 u_4 u_5 u_6 u_7) cât şi cei de control (c_0 c_1 c_2 c_3 c_4). În continuare se implementează cu ajutorul unor porţi SAU-EXCLUSIV, ecuaţiile 6.2. Vom genera biţii de control c_0' c_1' c_2' c_3' c_4' din biţii utili citiţi. De exemplu, pentru generarea bitului de control c_0', vom folosi ecuaţia $c_0' = u_0 \oplus u_3 \oplus u_4 \oplus u_6 \oplus u_7$, iar pentru implementarea acestei ecuaţii vom folosi patru porţi SAU-EX cu 2 intrări situate pe trei niveluri, aşa cum se poate observa în fig. 6.1.

Vom proceda în mod analog pentru a genera restul biţiilor de control c_1', c_2', c_3', c_4'. Biţii de control generaţi (c_0' c_1' c_2' c_3' c_4') se pot compara cu biţii de control citiţi din memoria DRAM (c_0 c_1 c_2 c_3 c_4), tot cu ajutorul unor porţi SAU-EX cu 2 intrări, rezultând ecuaţiile de sindrom $s_0 = c_0 \oplus c_0'$, $s_1 = c_1 \oplus c_1'$, $s_2 = c_2 \oplus c_2'$, $s_3 = c_3 \oplus c_3'$, $s_4 = c_4 \oplus c_4'$.

În continuare, aşa cum se poate observa în fig. 6.1. vom conecta cinci porţi inversoare pe fiecare linie de sindrom şi vom construi cu ajutorul a opt porţi ŞI cu 5 intrări decodorul de sindrom. Ecuaţiile care stau la baza construcţiei decodorului de sindrom sunt:

$$
\begin{aligned}
u'_0 &= s_0 \cdot s_1 \cdot s_2 \cdot \overline{s_3} \cdot \overline{s_4} \\
u'_1 &= \overline{s_0} \cdot s_1 \cdot s_2 \cdot s_3 \cdot \overline{s_4} \\
u'_2 &= \overline{s_0} \cdot \overline{s_1} \cdot s_2 \cdot s_3 \cdot s_4 \\
u'_3 &= s_0 \cdot \overline{s_1} \cdot s_2 \cdot s_3 \cdot \overline{s_4} \\
u'_4 &= s_0 \cdot \overline{s_1} \cdot \overline{s_2} \cdot s_3 \cdot s_4 \\
u'_5 &= \overline{s_0} \cdot s_1 \cdot \overline{s_2} \cdot s_3 \cdot s_4 \\
u'_6 &= s_0 \cdot \overline{s_1} \cdot s_2 \cdot \overline{s_3} \cdot s_4 \\
u'_7 &= s_0 \cdot s_1 \cdot \overline{s_2} \cdot \overline{s_3} \cdot s_4
\end{aligned}
\qquad (6.4.)
$$

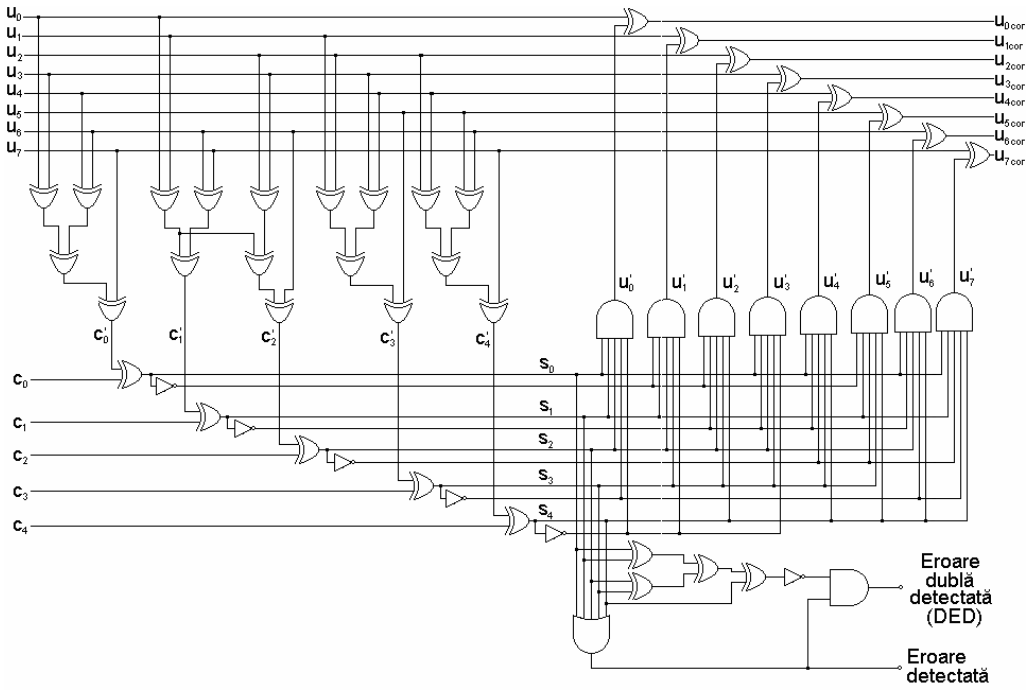


Fig. 6.1.

Vom conecta ieşirile porţilor ŞI la câte o intrare a unei porţi SAU-EX, cealaltă intrare a porţii având bitul util citit de la memoria DRAM. Corecţia biţilor utili se face prin utilizarea a opt porţi SAU-EX cu 2 intrări, şi această corecţie se realizează cu ecuaţiile:

$$
\begin{aligned}
u_{0cor} &= u_0 \oplus u_0', \\
u_{1cor} &= u_1 \oplus u_1', \\
u_{2cor} &= u_2 \oplus u_2', \\
u_{3cor} &= u_3 \oplus u_3', \\
u_{4cor} &= u_4 \oplus u_4', \\
u_{5cor} &= u_5 \oplus u_5', \\
u_{6cor} &= u_6 \oplus u_6', \\
u_{7cor} &= u_7 \oplus u_7'
\end{aligned}
\qquad (6.5.)
$$

Vom prezenta în continuare algoritmul care l-am implementat, algoritm care corectează erorile singulare şi detectează erorile multiple prin utilizarea sindromului $S=(s_0,s_1,s_2,s_3,s_4)$. În situaţia în care avem sindromul S egal cu zero (S=0), atunci cuvântul care este citit din memorie se presupune că este fără erori. În situaţia în care $S \neq 0$, se încearcă o identificare exactă între sindromul S şi o coloană a matricii H. Identificarea aceasta este realizată cu cele opt porţi ŞI cu 5 intrări. Dacă

sindromul S este identic cu a k-a coloană a matricii H, atunci al k-lea bit din cuvântul citit din memorie este eronat şi acest bit va fi inversat, executându-se astfel corecţia erorii.

În situaţia în care S ≠ 0, şi paritatea globală este egală cu zero, o eroare dublă sau un număr par de erori sunt detectate. Dacă un sindrom S diferit de zero, este egal cu una din coloanele matricii H şi paritatea globală este egală cu 1, atunci trei sau un număr impar de erori sunt detectate.

La scrierea în memoria DRAM se generează biţii de control (c_0 c_1 c_2 c_3 c_4) din biţii utili de date (u_0 u_1 u_2 u_3 u_4 u_5 u_6 u_7) care sunt furnizaţi din magistrala de date cu ajutorul ecuaţiilor 6.2, iar în memorie se scriu cuvinte de cod de forma: $u=(c_0c_1c_2c_3c_4u_0u_1u_2u_3u_4u_5u_6u_7)$.

Datorită faptului că avem patru blocuri de câte 4 MB, şi fiecare din aceste blocuri are conectare la o magistrală de date de opt biţi, trebuie să implementăm patru coduri Hsiao de acest fel, câte un cod pe fiecare magistrală de date care este conectată cu chip-urile de memorii DRAM. Datorită circuitelor suplimentare de corecţie a erorii vom avea nevoie de circuite de memorie cu capacitatea mai mare, care să poată memora în loc de 9 biţi (8 biţi de date şi un bit de paritate) un număr de 13 biţi (8 biţi de date şi 5 biţi de paritate).

La prima implementare, fară cod Hsiao, avem 4 circuite de memorie care memorează fiecare 9 biţi (8 biţi de date şi un bit de paritate), deci vom memora 9 biţi x 4 circuite = 36 biţi. În cazul celei de-a doua implementări, cu cod Hsiao (13,8,5), avem 4 circuite de memorie care memorează fiecare 13 biţi (8 biţi de date şi 5 biţi de paritate), deci vom memora 13 biţi x 4 circuite = 52 biţi. Apare un overhead de (52 - 36) x 100 / 52 = 30,77 % [NoVP07].

Overhead-ul datorat circuitelor suplimentare pentru corecţia erorii, porţi SI, porţi inversoare, porţi SAU-EX, porţi SAU, este neglijabil având valori între 1 şi 2%.

6.1.3. Aplicarea codului la memoria cache TAG

Implementarea versiunii cu coduri corectoare la memoria cache, cod Hsiao (13,8,5), la memoria cache RAM şi cod Hsiao (22,16,6), la memoria cache TAG.

În cazul cache-ului vom folosi acelaşi cod Hsiao (13,8,5), cu matricea H aşa cum s-a definit cu relaţia 6.1. Codul Hsiao este folosit pentru corecţia erorilor singulare şi detecţia erorilor duble şi el se aplică la ieşirile de date ale memoriei cache RAM. În cadrul cache-ului, memoria cache RAM are dimensiunea de 128 KB (128K x 8 biţi) şi este construită cu ajutorul a 4 circuite de dimensiune 32K x 8 biţi la care vom aplica codul Hsiao, definit prin relaţia 6.1 de 4 ori, pe liniile de date, între circuitele de memorie şi magistrala de date.

În cazul memoriei cache RAM cu dimensiunea de 128K x 8 biţi, fără cod corector, avem 4 circuite cu capacitatea de 32K x 8 biţi, deci vom memora 8 biţi x 4 circuite = 32 biţi. În cazul memoriei cache RAM cu dimensiunea de 128K x 8 biţi, cu cod corector Hsiao, avem nevoie de 4 circuite care memorează fiecare 13 biţi (8 biţi de date şi 5 biţi de paritate), deci vom memora 13 biţi x 4 circuite = 52 biţi. Apare un overhead de (52 - 32) x 100 / 52 = 38,46 %.

Overhead-ul datorat circuitelor suplimentare pentru corecţia erorii, porţi SI, porţi inversoare, porţi SAU-EX, porţi SAU, este neglijabil având valori între 1 şi 2%.

Pentru partea de TAG a memoriei cache vom folosi un alt cod Hsiao (22,16,6), cod care are 6 biţi de control şi 16 biţi utili de date. În cazul acestui cod (22,16,6), avem k = 6 biţi de control, u = 16 biţi utili iar numărul total de biţi ai codului este t = 22. Şi în acest caz pentru asigurarea corecţiei erorii singulare, între cele două mărimi, u şi k, este satisfăcută condiţia $2^k > u + k + 1$. În mod normal ar fi fost suficienţi un număr de k= 5 biţi de control, dar noi vom folosi un număr de k = 6 biţi de control, al şaselea bit fiind folosit pentru detecţia erorilor duble.

Codul Hsiao folosit pentru partea cache TAG, este definit prin matricea H dată de relaţiile (6.6):

$$H = \begin{array}{c} poz \\ \\ \\ \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{cccccccccccccccccccccc} c_0 & c_1 & c_2 & c_3 & c_4 & c_5 & u_0 & u_1 & u_2 & u_3 & u_4 & u_5 & u_6 & u_7 & u_8 & u_9 & u_{10} & u_{11} & u_{12} & u_{13} & u_{14} & u_{15} \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \end{array} \quad (6.6)$$

Un cuvânt de cod tipic al acestei matrici are următoarea formă:

$u = (c_0c_1c_2c_3c_4c_5u_0u_1u_2u_3u_4u_5u_6u_7u_8u_9u_{10}u_{11}u_{12}u_{13}u_{14}u_{15})$

şi are biţii de paritate în poziţiile 1,2,3,4,5 şi 6 iar biţi de date în rest (de la poziţia 7 la 22).

Am ales pentru matricea H, această amplasare a biţilor de 1 şi 0 pentru a obţine ecuaţii echilibrate pentru biţii de control. Precizez în acest caz că matricea H are pe primele şase linii şi şase coloane o matrice unitate, iar în continuare această matrice se construieşte din coloane cu un număr impar de biţii de 1. În acest caz, numărul biţilor de 1 din anumite coloane este egal cu 3, iar numărul biţilor de 1 din celelalte coloane este egal cu 5.

Biţii de control se calculează prin ecuaţiile de paritate date în continuare:

$$\begin{aligned}
c_0 &= u_0 \oplus u_4 \oplus u_5 \oplus u_6 \oplus u_8 \oplus u_9 \oplus u_{10} \oplus u_{12} \oplus u_{13} \oplus u_{14} \oplus u_{15} \\
c_1 &= u_0 \oplus u_1 \oplus u_7 \oplus u_8 \oplus u_9 \oplus u_{10} \oplus u_{11} \oplus u_{13} \oplus u_{14} \oplus u_{15} \\
c_2 &= u_0 \oplus u_1 \oplus u_2 \oplus u_4 \oplus u_7 \oplus u_{10} \oplus u_{11} \oplus u_{12} \oplus u_{14} \oplus u_{15} \\
c_3 &= u_1 \oplus u_2 \oplus u_3 \oplus u_4 \oplus u_5 \oplus u_9 \oplus u_{11} \oplus u_{12} \oplus u_{13} \oplus u_{15} \\
c_4 &= u_2 \oplus u_3 \oplus u_5 \oplus u_6 \oplus u_{10} \oplus u_{11} \oplus u_{12} \oplus u_{13} \oplus u_{14} \\
c_5 &= u_3 \oplus u_6 \oplus u_7 \oplus u_8 \oplus u_{10} \oplus u_{11} \oplus u_{12} \oplus u_{13} \oplus u_{14} \oplus u_{15}
\end{aligned} \tag{6.7}$$

Decodificarea vectorului recepţionat utilizează ecuaţiile de sindrom care sunt bazate pe ecuaţiile de paritate (6.7), iar aceste ecuaţii de sindrom se dau în continuare (6.8):

Vom aplica acest cod SEC-DED la memoria cache TAG a sistemului ce are capacitatea de 8 KB.

$$\begin{aligned}
s_0 &= c_0 \oplus u_0 \oplus u_4 \oplus u_5 \oplus u_6 \oplus u_8 \oplus u_9 \oplus u_{10} \oplus u_{12} \oplus u_{13} \oplus u_{14} \oplus u_{15} \\
s_1 &= c_1 \oplus u_0 \oplus u_1 \oplus u_7 \oplus u_8 \oplus u_9 \oplus u_{10} \oplus u_{11} \oplus u_{13} \oplus u_{14} \oplus u_{15} \\
s_2 &= c_2 \oplus u_0 \oplus u_1 \oplus u_2 \oplus u_4 \oplus u_7 \oplus u_{10} \oplus u_{11} \oplus u_{12} \oplus u_{14} \oplus u_{15} \\
s_3 &= c_3 \oplus u_1 \oplus u_2 \oplus u_3 \oplus u_4 \oplus u_5 \oplus u_9 \oplus u_{11} \oplus u_{12} \oplus u_{13} \oplus u_{15} \\
s_4 &= c_4 \oplus u_2 \oplus u_3 \oplus u_5 \oplus u_6 \oplus u_{10} \oplus u_{11} \oplus u_{12} \oplus u_{13} \oplus u_{14} \\
s_5 &= c_5 \oplus u_3 \oplus u_6 \oplus u_7 \oplus u_8 \oplus u_{10} \oplus u_{11} \oplus u_{12} \oplus u_{13} \oplus u_{14} \oplus u_{15}
\end{aligned} \tag{6.8}$$

La citirea din memoia cache TAG, vom citi atît biţii utili de date (u_0 u_1 u_2 u_3 u_4 u_5 u_6 u_7 u_8 u_9 u_{10} u_{11} u_{12} u_{13} u_{14} u_{15}) cât şi biţii de control (c_0 c_1 c_2 c_3 c_4 c_5). În continuare se implementează cu ajutorul unor porţi SAU-EXCLUSIV, ecuaţiile 6.7. Vom genera biţii de control c_0' c_1' c_2' c_3' c_4' c_5' din biţii utili citiţi. De exemplu, pentru generarea bitului de control c_0', vom folosi ecuaţia,

$$c_0' = u_0 \oplus u_4 \oplus u_5 \oplus u_6 \oplus u_8 \oplus u_9 \oplus u_{10} \oplus u_{12} \oplus u_{13} \oplus u_{14} \oplus u_{15}$$

iar pentru implementarea acestei ecuaţii vom folosi nouă porţi SAU-EX cu 2 intrări situate pe patru niveluri, aşa cum se poate observa în fig. 6.2. Vom proceda în mod analog pentru a genera restul biţiilor de control c_1', c_2', c_3', c_4', c_5'. Biţii de control generaţi (c_0' c_1' c_2' c_3' c_4' c_5') se pot compara cu biţii de control citiţi din memoria cache TAG (c_0 c_1 c_2 c_3 c_4 c_5), tot cu ajutorul unor porţi SAU-EX cu 2 intrări, rezultând ecuaţiile de sindrom $s_0 = c_0 \oplus c_0'$, $s_1 = c_1 \oplus c_1'$, $s_2 = c_2 \oplus c_2'$, $s_3 = c_3 \oplus c_3'$, $s_4 = c_4 \oplus c_4'$, $s_5 = c_5 \oplus c_5'$. În continuare, aşa cum se poate observa în fig. 6.2., vom onstrui decodorul de sindrom.

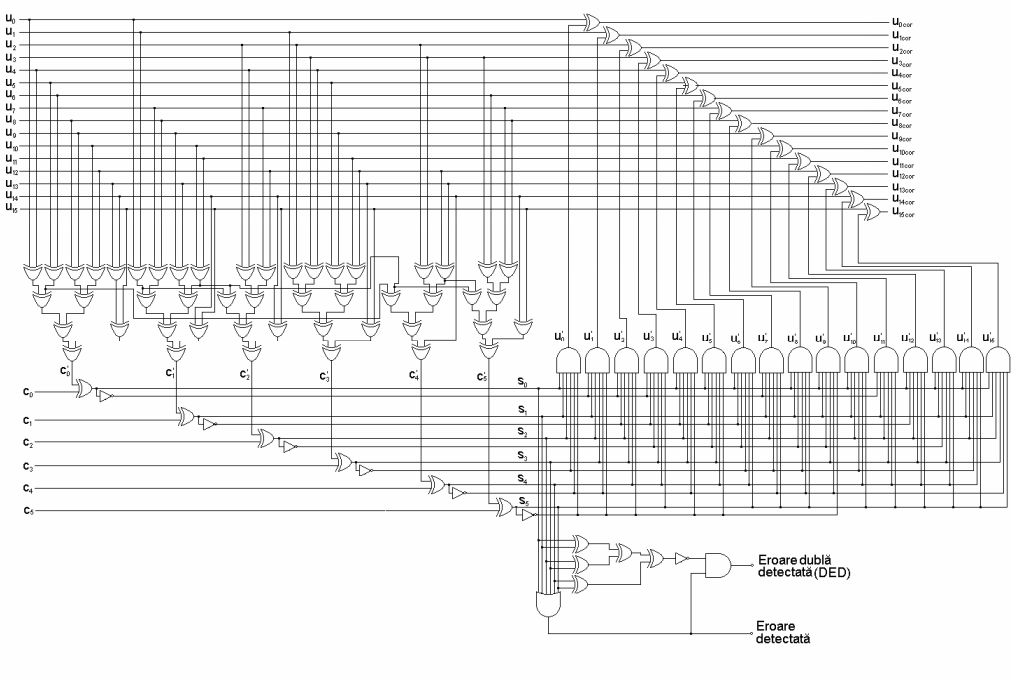


Fig. 6.2.

Vom conecta şase porţi inversoare pe fiecare linie de sindrom şi vom construi cu ajutorul a 16 porţi ŞI cu 6 intrări decodorul de sindrom. Ecuaţiile care stau la baza construcţiei decodorului de sindrom sunt:

$$
\begin{aligned}
u_0' &= s_0 \cdot s_1 \cdot s_2 \cdot \overline{s_3} \cdot \overline{s_4} \cdot \overline{s_5} \\
u_1' &= \overline{s_0} \cdot s_1 \cdot s_2 \cdot s_3 \cdot \overline{s_4} \cdot \overline{s_5} \\
u_2' &= \overline{s_0} \cdot \overline{s_1} \cdot s_2 \cdot s_3 \cdot s_4 \cdot \overline{s_5} \\
u_3' &= \overline{s_0} \cdot \overline{s_1} \cdot \overline{s_2} \cdot s_3 \cdot s_4 \cdot s_5 \\
u_4' &= s_0 \cdot \overline{s_1} \cdot s_2 \cdot s_3 \cdot \overline{s_4} \cdot \overline{s_5} \\
u_5' &= s_0 \cdot \overline{s_1} \cdot \overline{s_2} \cdot s_3 \cdot s_4 \cdot \overline{s_5} \\
u_6' &= s_0 \cdot \overline{s_1} \cdot \overline{s_2} \cdot \overline{s_3} \cdot s_4 \cdot s_5 \\
u_7' &= \overline{s_0} \cdot s_1 \cdot s_2 \cdot \overline{s_3} \cdot \overline{s_4} \cdot s_5 \\
u_8' &= s_0 \cdot s_1 \cdot \overline{s_2} \cdot \overline{s_3} \cdot \overline{s_4} \cdot s_5 \\
u_9' &= s_0 \cdot s_1 \cdot \overline{s_2} \cdot s_3 \cdot \overline{s_4} \cdot \overline{s_5} \\
u_{10}' &= s_0 \cdot s_1 \cdot s_2 \cdot \overline{s_3} \cdot s_4 \cdot s_5
\end{aligned}
\qquad (6.9)
$$

$$u_{11}' = \overline{s_0} \cdot s_1 \cdot s_2 \cdot s_3 \cdot s_4 \cdot s_5$$
$$u_{12}' = s_0 \cdot \overline{s_1} \cdot s_2 \cdot s_3 \cdot s_4 \cdot s_5$$
$$u_{13}' = s_0 \cdot s_1 \cdot \overline{s_2} \cdot s_3 \cdot s_4 \cdot s_5$$
$$u_{14}' = s_0 \cdot s_1 \cdot s_2 \cdot \overline{s_3} \cdot s_4 \cdot s_5$$
$$u_{15}' = s_0 \cdot s_1 \cdot s_2 \cdot s_3 \cdot \overline{s_4} \cdot s_5$$

Vom conecta ieşirile porţilor ŞI la câte o intrare a unei porţi SAU-EX, cealaltă intrare a porţii având bitul util citit din cache TAG. Corecţia biţilor utili se face prin utilizarea a 16 porţi SAU-EX cu 2 intrări, această corecţie realizându-se cu ecuaţiile:

$$\begin{aligned}
u_{0cor} &= u_0 \oplus u_0', \\
u_{1cor} &= u_1 \oplus u_1', \\
u_{2cor} &= u_2 \oplus u_2', \\
u_{3cor} &= u_3 \oplus u_3', \\
u_{4cor} &= u_4 \oplus u_4', \\
u_{5cor} &= u_5 \oplus u_5', \\
u_{6cor} &= u_6 \oplus u_6', \\
u_{7cor} &= u_7 \oplus u_7' \\
u_{8cor} &= u_8 \oplus u_8', \\
u_{9cor} &= u_9 \oplus u_9', \\
u_{10cor} &= u_{10} \oplus u_{10}', \\
u_{11cor} &= u_{11} \oplus u_{11}', \\
u_{12cor} &= u_{12} \oplus u_{12}', \\
u_{13cor} &= u_{13} \oplus u_{13}', \\
u_{14cor} &= u_{14} \oplus u_{14}' \\
u_{15cor} &= u_{15} \oplus u_{15}'
\end{aligned} \qquad (6.10)$$

Vom prezenta în continuare algoritmul pe care l-am implementat, algoritm care corectează erorile singulare şi detectează erorile duble prin utilizarea sindromului S = (s_0,s_1,s_2,s_3,s_4,s_5). În situaţia în care avem sindromul S egal cu zero (S = 0), atunci cuvântul care este citit din memorie se presupune că este fără erori. În situaţia în care S ≠ 0, se încearcă o identificare exactă între sindromul S şi o coloană a matricii H. Identificarea aceasta este realizată cu cele 16 porţi ŞI cu 6 intrări. Dacă sindromul S este identic cu a k-a coloană a matricii H, atunci al k-lea bit din cuvântul citit din memorie este eronat şi acest bit va fi inversat, executându-se astfel corecţia erorii.

În situaţia în care S ≠ 0, şi paritatea globală este egală cu zero, o eroare dublă sau un număr par de erori sunt detectate. Dacă un sindrom S diferit de zero, este egal cu una din coloanele matricii H şi paritatea globală este egală cu 1, atunci trei sau un număr impar de erori sunt detectate.

La scrierea în memoria cache TAG se generează biţii de control (c_0 c_1 c_2 c_3 c_4 c_5), din biţii utili de date (u_0 u_1 u_2 u_3 u_4 u_5 u_6 u_7 u_8 u_9 u_{10} u_{11} u_{12} u_{13} u_{14} u_{15}), care sunt furnizaţi din magistrala de date cu ajutorul ecuaţiilor 6.7, în memorie scriindu-se cuvinte de cod de forma:

$$u=(c_0c_1c_2c_3c_4c_5u_0u_1u_2u_3u_4u_5u_6u_7u_8u_9u_{10}u_{11}u_{12}u_{13}u_{14}u_{15}).$$

Datorită faptului că avem patru blocuri de câte 4K x 4 biţi, şi fiecare două blocuri sunt conectare la partea superioară a magistralei de adrese de ADR(23:17), trebuie să implementăm un singur cod Hsiao de acest fel. Datorită circuitelor suplimentare de corecţie a erorii vom avea nevoie de circuite de memorie cu capacitatea mai mare, care să poată memora în loc de 4 biţi un număr de 5 sau 6 biţi (4 biţi de date şi 1 sau 2 biţi de paritate).

La prima implementare a memoriei cache TAG cu dimensiunea de 8k x 8 biti avem 2 grupuri de 2 circuite de memorie care memorează fiecare 4 biţi, deci vom memora 4 biţi x 2 circuite = 8 biţi. În cazul celei de-a doua implementări a memoriei cache TAG cu cod Hsiao (22,16,6) avem 4 circuite de memorie, din care 2 circuite memorează 5 biţi şi 2 circuite memorează 6 biţi, deci vom memora 10 biţi + 12 biţi = 22 biţi. Apare un overhead de (22 - 16) x 100 / 22 = 27,27%

Overhead-ul datorat circuitelor suplimentare pentru corecţia erorii, porţi SI, porţi inversoare, porţi SAU-EX, porţi SAU, este neglijabil şi are valori între 1 - 2%.

6.1.4. Aplicarea codului la memoria Cache RAM.

Implementarea versiunii cu coduri corectoare la memoria cache, cod Hsiao (39,32,7), la memoria cache RAM şi cod Hsiao (22,16,6), la memoria cache TAG [NVVH08].

Pentru partea cache RAM a memoriei cache vom folosi un alt cod Hsiao (39,32,7), cod care are 7 biţi de control şi 32 biţi utili de date. Codul Hsiao folosit pentru partea cache RAM, este definit prin matricea H dată de relaţiile (6.11).

Un cuvânt de cod tipic al acestei matrici are forma:

$$u=(c_0c_1c_2c_3c_4c_5c_6u_0u_1u_2u_3u_4u_5u_6u_7u_8u_9u_{10}u_{11}u_{12}u_{13}u_{14}u_{15}u_{16}u_{17}u_{18}u_{19}u_{20}u_{21}u_{22}u_{23}u_{24}u_{25}u_{26}u_{27}u_{28}u_{29}u_{30}u_{31})$$

şi are biţii de paritate în poziţiile 1,2,3,4,5,6 şi 7, şi biţi de date în rest (de la poziţia 8 la 39)

$$H=\begin{bmatrix}
1&0&0&0&0&0&0&1&0&1&0&0&0&1&1&1&0&0&0&0&1&1&1&0&0&1&0&0&1&0&0&0&1&0&1&1&1&0&1\\
0&1&0&0&0&0&0&1&1&0&1&0&0&0&1&1&1&0&0&0&0&0&1&1&0&0&1&0&1&1&0&0&0&1&0&1&1&0&0\\
0&0&1&0&0&0&0&0&1&1&0&1&0&0&0&1&1&1&0&0&0&0&0&1&1&0&0&1&0&1&1&0&0&0&1&1&1&1&0\\
0&0&0&1&0&0&0&0&0&1&1&0&1&0&0&0&1&1&1&0&0&1&0&0&1&1&0&0&1&0&1&1&0&0&0&1&0&1&1\\
0&0&0&0&1&0&0&0&0&0&1&1&0&1&0&0&0&1&1&1&0&0&1&0&0&1&1&0&0&1&0&1&1&0&0&0&1&0&1&1\\
0&0&0&0&0&1&0&1&0&0&0&1&1&0&0&0&0&0&1&1&1&0&0&1&0&0&1&1&0&0&1&0&1&1&0&0&1&1&1\\
0&0&0&0&0&0&1&0&1&0&0&0&1&1&1&0&0&0&0&1&1&1&0&0&1&0&0&1&0&0&0&1&0&1&1&0&1&1&1
\end{bmatrix} \tag{6.11}$$

Biţii de control se calculează prin ecuaţiile de paritate date în continuare:

$$\begin{aligned}
c_0 &= u_0 \oplus u_2 \oplus u_6 \oplus u_7 \oplus u_8 \oplus u_{13} \oplus u_{14} \oplus u_{15} \oplus u_{18} \oplus u_{21} \oplus u_{25} \oplus u_{27} \oplus u_{28} \oplus u_{29} \oplus u_{31}\\
c_1 &= u_0 \oplus u_1 \oplus u_3 \oplus u_7 \oplus u_8 \oplus u_9 \oplus u_{15} \oplus u_{16} \oplus u_{19} \oplus u_{21} \oplus u_{22} \oplus u_{26} \oplus u_{28} \oplus u_{29}\\
c_2 &= u_1 \oplus u_2 \oplus u_4 \oplus u_8 \oplus u_9 \oplus u_{10} \oplus u_{16} \oplus u_{17} \oplus u_{20} \oplus u_{22} \oplus u_{23} \oplus u_{27} \oplus u_{28} \oplus u_{29} \oplus u_{30}\\
c_3 &= u_2 \oplus u_3 \oplus u_5 \oplus u_9 \oplus u_{10} \oplus u_{11} \oplus u_{14} \oplus u_{17} \oplus u_{18} \oplus u_{21} \oplus u_{23} \oplus u_{24} \oplus u_{28} \oplus u_{30} \oplus u_{31}\\
c_4 &= u_3 \oplus u_4 \oplus u_6 \oplus u_{10} \oplus u_{11} \oplus u_{12} \oplus u_{15} \oplus u_{18} \oplus u_{19} \oplus u_{22} \oplus u_{24} \oplus u_{25} \oplus u_{28} \oplus u_{30} \oplus u_{31}\\
c_5 &= u_0 \oplus u_4 \oplus u_5 \oplus u_{11} \oplus u_{12} \oplus u_{13} \oplus u_{15} \oplus u_{19} \oplus u_{20} \oplus u_{23} \oplus u_{25} \oplus u_{26} \oplus u_{29} \oplus u_{30} \oplus u_{31}\\
c_6 &= u_1 \oplus u_5 \oplus u_6 \oplus u_7 \oplus u_{12} \oplus u_{13} \oplus u_{14} \oplus u_{17} \oplus u_{20} \oplus u_{24} \oplus u_{26} \oplus u_{27} \oplus u_{29} \oplus u_{30} \oplus u_{31}
\end{aligned} \tag{6.12}$$

Decodificarea vectorului recepţionat utilizează ecuaţiile de sindrom care sunt bazate pe ecuaţiile de paritate (6.12), iar aceste ecuaţii de sindrom se dau în continuare:

$$\begin{aligned}
s_0 &= c_0 \oplus u_0 \oplus u_2 \oplus u_6 \oplus u_7 \oplus u_8 \oplus u_{13} \oplus u_{14} \oplus u_{15} \oplus u_{18} \oplus u_{21} \oplus u_{25} \oplus u_{27} \oplus u_{28} \oplus u_{29} \oplus u_{31}\\
s_1 &= c_1 \oplus u_0 \oplus u_1 \oplus u_3 \oplus u_7 \oplus u_8 \oplus u_9 \oplus u_{15} \oplus u_{16} \oplus u_{19} \oplus u_{21} \oplus u_{22} \oplus u_{26} \oplus u_{28} \oplus u_{29}\\
s_2 &= c_2 \oplus u_1 \oplus u_2 \oplus u_4 \oplus u_8 \oplus u_9 \oplus u_{10} \oplus u_{16} \oplus u_{17} \oplus u_{20} \oplus u_{22} \oplus u_{23} \oplus u_{27} \oplus u_{28} \oplus u_{29} \oplus u_{30}\\
s_3 &= c_3 \oplus u_2 \oplus u_3 \oplus u_5 \oplus u_9 \oplus u_{10} \oplus u_{11} \oplus u_{14} \oplus u_{17} \oplus u_{18} \oplus u_{21} \oplus u_{23} \oplus u_{24} \oplus u_{28} \oplus u_{30} \oplus u_{31}\\
s_4 &= c_4 \oplus u_3 \oplus u_4 \oplus u_6 \oplus u_{10} \oplus u_{11} \oplus u_{12} \oplus u_{15} \oplus u_{18} \oplus u_{19} \oplus u_{22} \oplus u_{24} \oplus u_{25} \oplus u_{28} \oplus u_{30} \oplus u_{31}\\
s_5 &= c_5 \oplus u_0 \oplus u_4 \oplus u_5 \oplus u_{11} \oplus u_{12} \oplus u_{13} \oplus u_{15} \oplus u_{19} \oplus u_{20} \oplus u_{23} \oplus u_{25} \oplus u_{26} \oplus u_{29} \oplus u_{30} \oplus u_{31}\\
s_6 &= c_6 \oplus u_1 \oplus u_5 \oplus u_6 \oplus u_7 \oplus u_{12} \oplus u_{13} \oplus u_{14} \oplus u_{17} \oplus u_{20} \oplus u_{24} \oplus u_{26} \oplus u_{27} \oplus u_{29} \oplus u_{30} \oplus u_{31}
\end{aligned} \tag{6.13}$$

Vom aplica acest cod corector la memoria cache RAM a sistemului care are capacitatea de 128K x 8 biţi. Implementarea acestui cod corector, este prezentat în anexa fig. C.1. din anexa C.

La citirea din memoia cache RAM, vom citi atît biţii utili de date ($u_0u_1u_2u_3u_4u_5u_6u_7u_8u_9u_{10}u_{11}u_{12}u_{13}u_{14}u_{15}u_{16}u_{17}u_{18}u_{19}u_{20}u_{21}u_{22}u_{23}u_{24}u_{25}u_{26}u_{27}u_{28}u_{29}u_{30}u_{31}$) cât şi biţii de control (c_0 c_1 c_2 c_3 c_4 c_5 c_6). În continuare se implementează cu ajutorul unor porţi SAU-EXCLUSIV, ecuaţiile 6.12. Vom genera biţii de control c_0' c_1' c_2' c_3' c_4' c_5' c_6' din biţii utili citiţi.

De exemplu, pentru generarea bitului de control c_0', vom folosi ecuaţia:

$$c_0' = u_0 \oplus u_2 \oplus u_6 \oplus u_7 \oplus u_8 \oplus u_{13} \oplus u_{14} \oplus u_{15} \oplus u_{18} \oplus u_{21} \oplus u_{25} \oplus u_{27} \oplus u_{28} \oplus u_{29} \oplus u_{31},$$

iar pentru a implementa acestă ecuaţie vom folosi 14 porţi SAU-EX cu 2 intrări situate pe patru niveluri, aşa cum se poate observă în anexa C în fig. C.1. Vom proceda în mod analog pentru a genera restul biţiilor de control c_1', c_2', c_3', c_4', c_5',

c_6'. Biţii de control generaţi ($c_0'\ c_1'\ c_2'\ c_3'\ c_4'\ c_5'\ c_6'$) se pot compara cu biţii de control citiţi din memoria cache RAM ($c_0\ c_1\ c_2\ c_3\ c_4\ c_5\ c_6$), tot cu ajutorul unor porţi SAU-EX cu 2 intrări, rezultând ecuaţiile de sindrom:

$$s_0 = c_0 \oplus c_0',\ s_1 = c_1 \oplus c_1',\ s_2 = c_2 \oplus c_2',\ s_3 = c_3 \oplus c_3',\ s_4 = c_4 \oplus c_4',\ s_5 = c_5 \oplus c_5',\ s_6 = c_6 \oplus c_6'.$$

În continuare, aşa cum se poate observa în fig. C.1. vom conecta 7 porţi inversoare pe fiecare linie de sindrom şi vom construi cu ajutorul a 32 porţi ŞI cu 7 intrări decodorul de sindrom. Ecuaţiile care stau la baza construcţiei decodorului de sindrom sunt:

$$\begin{aligned}
u_0' &= s_0 \cdot s_1 \cdot \overline{s_2} \cdot \overline{s_3} \cdot \overline{s_4} \cdot s_5 \cdot \overline{s_6} \\
u_1' &= \overline{s_0} \cdot s_1 \cdot s_2 \cdot \overline{s_3} \cdot \overline{s_4} \cdot \overline{s_5} \cdot s_6 \\
u_2' &= s_0 \cdot \overline{s_1} \cdot s_2 \cdot s_3 \cdot \overline{s_4} \cdot \overline{s_5} \cdot \overline{s_6} \\
u_3' &= \overline{s_0} \cdot s_1 \cdot \overline{s_2} \cdot s_3 \cdot s_4 \cdot \overline{s_5} \cdot \overline{s_6} \\
u_4' &= \overline{s_0} \cdot \overline{s_1} \cdot s_2 \cdot \overline{s_3} \cdot s_4 \cdot s_5 \cdot \overline{s_6} \\
u_5' &= \overline{s_0} \cdot \overline{s_1} \cdot \overline{s_2} \cdot s_3 \cdot \overline{s_4} \cdot s_5 \cdot s_6 \\
u_6' &= s_0 \cdot \overline{s_1} \cdot \overline{s_2} \cdot \overline{s_3} \cdot s_4 \cdot \overline{S_5} \cdot s_6 \\
u_7' &= s_0 \cdot s_1 \cdot \overline{s_2} \cdot \overline{s_3} \cdot \overline{s_4} \cdot \overline{S_5} \cdot s_6 \\
u_8' &= s_0 \cdot s_1 \cdot s_2 \cdot \overline{s_3} \cdot \overline{s_4} \cdot \overline{s_5} \cdot \overline{s_6} \\
u_9' &= \overline{s_0} \cdot s_1 \cdot s_2 \cdot s_3 \cdot \overline{s_4} \cdot \overline{s_5} \cdot \overline{s_6} \\
u_{10}' &= \overline{s_0} \cdot \overline{s_1} \cdot s_2 \cdot s_3 \cdot s_4 \cdot \overline{s_5} \cdot \overline{s_6} \\
u_{11}' &= \overline{s_0} \cdot \overline{s_1} \cdot \overline{s_2} \cdot s_3 \cdot s_4 \cdot s_5 \cdot \overline{s_6} \\
u_{12}' &= \overline{s_0} \cdot \overline{s_1} \cdot \overline{s_2} \cdot \overline{s_3} \cdot s_4 \cdot s_5 \cdot s_6 \\
u_{13}' &= s_0 \cdot \overline{s_1} \cdot \overline{s_2} \cdot \overline{s_3} \cdot \overline{s_4} \cdot s_5 \cdot s_6 \\
u_{14}' &= s_0 \cdot \overline{s_1} \cdot \overline{s_2} \cdot s_3 \cdot \overline{s_4} \cdot \overline{s_5} \cdot s_6 \\
u_{15}' &= s_0 \cdot s_1 \cdot \overline{s_2} \cdot \overline{s_3} \cdot s_4 \cdot \overline{s_5} \cdot \overline{s_6} \\
u_{16}' &= \overline{s_0} \cdot s_1 \cdot s_2 \cdot \overline{s_3} \cdot \overline{s_4} \cdot s_5 \cdot \overline{s_6} \\
u_{17}' &= \overline{s_0} \cdot \overline{s_1} \cdot s_2 \cdot s_3 \cdot \overline{s_4} \cdot \overline{s_5} \cdot s_6 \\
u_{18}' &= s_0 \cdot \overline{s_1} \cdot \overline{s_2} \cdot s_3 \cdot s_4 \cdot \overline{s_5} \cdot \overline{s_6} \\
u_{19}' &= \overline{s_0} \cdot s_1 \cdot \overline{s_2} \cdot \overline{s_3} \cdot s_4 \cdot s_5 \cdot \overline{s_6} \\
u_{20}' &= \overline{s_0} \cdot \overline{s_1} \cdot s_2 \cdot \overline{s_3} \cdot \overline{s_4} \cdot s_5 \cdot s_6 \\
u_{21}' &= s_0 \cdot s_1 \cdot \overline{s_2} \cdot s_3 \cdot \overline{s_4} \cdot \overline{s_5} \cdot \overline{s_6} \\
u_{22}' &= \overline{s_0} \cdot s_1 \cdot s_2 \cdot \overline{s_3} \cdot s_4 \cdot \overline{s_5} \cdot \overline{s_6} \\
u_{23}' &= \overline{s_0} \cdot \overline{s_1} \cdot s_2 \cdot s_3 \cdot \overline{s_4} \cdot s_5 \cdot \overline{s_6}
\end{aligned} \qquad (6.14)$$

$$u_{24}' = \overline{s_0} \cdot \overline{s_1} \cdot \overline{s_2} \cdot s_3 \cdot s_4 \cdot \overline{S_5} \cdot s_6$$
$$u_{25}' = s_0 \cdot \overline{s_1} \cdot \overline{s_2} \cdot \overline{s_3} \cdot s_4 \cdot s_5 \cdot \overline{s_6}$$
$$u_{26}' = \overline{s_0} \cdot s_1 \cdot \overline{s_2} \cdot \overline{s_3} \cdot \overline{s_4} \cdot s_5 \cdot s_6$$
$$u_{27}' = s_0 \cdot \overline{s_1} \cdot s_2 \cdot \overline{s_3} \cdot \overline{s_4} \cdot \overline{s_5} \cdot s_6$$
$$u_{28}' = s_0 \cdot s_1 \cdot s_2 \cdot s_3 \cdot s_4 \cdot \overline{s_5} \cdot \overline{s_6}$$
$$u_{29}' = s_0 \cdot s_1 \cdot s_2 \cdot \overline{s_3} \cdot \overline{s_4} \cdot s_5 \cdot s_6$$
$$u_{30}' = \overline{s_0} \cdot \overline{s_1} \cdot s_2 \cdot s_3 \cdot s_4 \cdot s_5 \cdot s_6$$
$$u_{31}' = s_0 \cdot \overline{s_1} \cdot \overline{s_2} \cdot s_3 \cdot s_4 \cdot s_5 \cdot s_6$$

Vom conecta ieşirile porţilor ŞI la câte o intrare a unei porţi SAU-EX, cealaltă intrare a porţii având bitul util citit din memoria cache RAM. Corecţia biţilor utili se face prin utilizarea a 32 porţi SAU-EX cu 2 intrări, această corecţie realizându-se cu ecuaţiile (6.15):

$$\begin{aligned}
u_{0cor} &= u_0 \oplus u_0', \\
u_{1cor} &= u_1 \oplus u_1', \\
u_{2cor} &= u_2 \oplus u_2', \\
u_{3cor} &= u_3 \oplus u_3', \\
u_{4cor} &= u_4 \oplus u_4', \\
u_{5cor} &= u_5 \oplus u_5', \\
u_{6cor} &= u_6 \oplus u_6', \\
u_{7cor} &= u_7 \oplus u_7' \\
u_{8cor} &= u_8 \oplus u_8', \\
u_{9cor} &= u_9 \oplus u_9', \\
u_{10cor} &= u_{10} \oplus u_{10}', \\
u_{11cor} &= u_{11} \oplus u_{11}', \\
u_{12cor} &= u_{12} \oplus u_{12}', \\
u_{13cor} &= u_{13} \oplus u_{13}', \\
u_{14cor} &= u_{14} \oplus u_{14}' \\
u_{15cor} &= u_{15} \oplus u_{15}', \\
u_{16cor} &= u_{16} \oplus u_{16}', \\
u_{17cor} &= u_{17} \oplus u_{17}', \\
u_{18cor} &= u_{18} \oplus u_{18}', \\
u_{19cor} &= u_{19} \oplus u_{19}', \\
u_{20cor} &= u_{20} \oplus u_{20}', \\
u_{21cor} &= u_{21} \oplus u_{21}', \\
u_{22cor} &= u_{22} \oplus u_{22}' \\
u_{23cor} &= u_{23} \oplus u_{23}', \\
u_{24cor} &= u_{24} \oplus u_{24}', \\
u_{25cor} &= u_{25} \oplus u_{25}',
\end{aligned} \qquad (6.15)$$

$$u_{26cor} = u_{26} \oplus u_{26}',$$
$$u_{27cor} = u_{27} \oplus u_{27}',$$
$$u_{28cor} = u_{28} \oplus u_{28}',$$
$$u_{29cor} = u_{29} \oplus u_{29}'$$
$$u_{30cor} = u_{30} \oplus u_{30}'$$
$$u_{31cor} = u_{31} \oplus u_{31}'$$

Vom prezenta în continuare algoritmul care l-am implementat, algoritm care corectează erorile singulare şi detectează erorile duble prin utilizarea sindromului S = (s_0,s_1,s_2,s_3,s_4,s_5,s_6). În situaţia în care avem sindromul S egal cu zero (S = 0), atunci cuvântul care este citit din memorie se presupune că este fără erori. În situaţia în care S ≠ 0, se încearcă o identificare exactă între sindromul S şi o coloană a matricii H. Această identificare este realizată cu cele 32 porţi ŞI cu 7 intrări. Dacă sindromul S este identic cu a k-a coloană a matricii H, atunci al k-lea bit din cuvântul citit din memorie este eronat şi acest bit va fi inversat, executându-se astfel corecţia erorii.

În situaţia în care S ≠ 0, şi paritatea globală este egală cu zero, o eroare dublă sau un număr par de erori sunt detectate. Dacă un sindrom S diferit de zero, este egal cu una din coloanele matricii H şi paritatea globală este egală cu 1, atunci trei sau un număr impar de erori sunt detectate.

La scrierea în memoria cache RAM se generează biţii de control (c_0 c_1 c_2 c_3 c_4 c_5 c_6) din biţii utili de date (u_0 u_1 u_2 u_3 u_4 u_5 u_6 u_7 u_8 u_9 u_{10} u_{11} u_{12} u_{13} u_{14} u_{15} u_{16} u_{17} u_{18} u_{19} u_{20} u_{21} u_{22} u_{23} u_{24} u_{25} u_{26} u_{27} u_{28} u_{29} u_{30} u_{31}), care sunt furnizaţi din magistrala de date cu ajutorul ecuaţiilor 6.12, în memoria cache RAM scriindu-se cuvinte de cod de forma:

$$u=(c_0c_1c_2c_3c_4c_5c_6u_0u_1u_2u_3u_4u_5u_6u_7u_8u_9u_{10}u_{11}u_{12}u_{13}u_{14}u_{15}u_{16}u_{17}u_{18}u_{19}u_{20}u_{21}u_{22}u_{23}u_{24}u_{25}u_{26}u_{27}u_{28}u_{29}u_{30}u_{31}).$$

La memoria cache RAM cu dimensiunea de 128K x 8 biţi, fără cod corector, avem 4 circuite cu capacitatea de 32K x 8 biţi, deci vom memora 8 biţi x 4 circuite = 32 biţi. În cazul memoriei cache RAM cu dimensiunea de 128K x 8 biţi, cu cod corector Hsiao (39,32,7), avem nevoie de circuite care memorează 39 biţi (32 biţi de date şi 7 biţi de paritate). Apare un overhead pe partea de date de (39 - 32) x 100 / 39 = 17,94%. O altă problemă care apare în acest caz, este overhead-ul datorat circuitelor suplimentare pentru corecţia erorii, porţi SI, porţi inversoare, porţi SAU-EX, porţi SAU, acest overhead având valoarea de 6%.

6.1.5. Implementarea cu circuitele programabile FPGA Xilinx a codului corector pentru memoria cache RAM.

În continuare vom implementa schema de corecţie a memoriei cache RAM din fig. C.1, cu ajutorul circuitelor programabile FPGA din Xilinx. Procesul de proiectare cu circuite FPGA Xilinx este rapid şi eficient. Structura internă a unei arhitecturi FPGA este asemănătoare cu cea a ariilor generice şi este organizată într-o matrice alcătuită din blocuri logice configurabile (CLB) şi matrici de comutare programabile (Programable Switch Matrices - PSM), înconjurate la periferie de pini de intrare/ieşire (I/O Pin). Segmentele de interconexiuni din metal pot realiza prin intermediul punctelor de conexiune, legături între celulele logice configurabile şi celulele de I/O. Implementarea se va realiza cu circuite din seria XC4000. Seria XC4000 este implementată printr-o structură regulară, flexibilă, cu o arhitectură programabilă, realizată utilizând celulele logice programabile (CLB) şi resurse de conexiuni abundente programabile [MaMŢ99], [MaŢi00].

Structura internă programabilă de utilizator include două elemente configurabile: *blocurile logice configurabile (CLB),* care furnizează elementele funcţionale şi realizează structura logică proiectată şi *blocurile de intrare/ieşire (IOB),* care furnizează interfaţa între semnalele interne şi exteriorul circuitului (prin intermediul pinilor). Structura cuprinde încă trei elemente programabile: fiecărui bloc logic configurabil îi este asociat un buffer tri-state a cărei ieşire este conectată la liniile lungi asociate, decodificatoare rapide implementate pentru viteze ridicate, un oscilator intern cu frecvenţa de 8 MHz şi interconexiunile programabile. Funcţia logică realizată de fiecare CLB este implementată prin intermediul memoriei statice de configurare. Valorile din aceste memorii determină starea blocurilor şi interconexiunilor. În fig. C.2. din anexa C, sunt prezentate elementele unui bloc logic configurabil, neconfigurat.

Un CLB conţine două elemente de stocare (realizate cu bistabile de tip D) care se pot utiliza pentru stocarea rezultatelor date de generatoarele de funcţii. Generatoarele de funcţii se pot utiliza şi independent. Generatoarele de funcţii sunt implementate ca şi tabele de memorii. Astfel timpul de propagare este independent de funcţia propagată [MaMŢ99].

Am realizat implementarea codului corector pentru memoria cache RAM, cu ajutorul circuitele programabile FPGA Xilinx din seria XC4000. Această implementare este prezentată în fig. C.3., C.4. şi C.5., în anexa C.

În fig. C.6. (din anexa C) este prezentat un bloc logic configurabil care este deja configurat de către programul Xilinx.

Overhead-ul datorat circuitelor suplimentare pentru corecţia erorii, porţi SI, porţi inversoare, porţi SAU-EX, porţi SAU, are o valoare de 6,25 %, acest rezultat este obţinut cu ajutorului unui raport din Xilinx, denumit map.mrp, raport care ne precizează gradul de utilizare a blocurilor CLB din numărul total al blocurilor CLB. În cazul nostru avem utilizate un număr de 64 de CLB-uri dintr-un număr total de 1024 CLB-uri.

Pe partea de memorie cache TAG la implementarea memoriei cache TAG fară cod Hsiao, avem 4 circuite de memorie care memorează fiecare 4 biţi, deci vom memora 4 biţi x 4 circuite = 16 biţi.

În cazul implementării memoriei cache TAG cu cod Hsiao (22,16,6) avem 4 circuite de memorie, din care 2 circuite memorează 5 biţi şi 2 circuite memorează 6 biţi, deci vom memora 10 biţi + 12 biţi = 22 biţi. Apare un overhead de (22 - 16) x 100 / 22 = 27,27%.

Overhead-ul datorat circuitelor suplimentare pentru corecţia erorii, porţi SI, porţi inversoare, porţi SAU-EX, porţi SAU, şi are valori între 5-6%.

6.2. EVALUĂRI DE INDICI DE FIABILITATE

Implementarea codului corector Hsiao la memoria cache RAM

În tabelul 6.1 este prezentat calculul overhead-ului, în cazul implementării codului corector Hsiao (13,8,5), (22,16,6) şi (39,32,7) la memoria cache RAM cu capacitatea de 128Kx8 biţi.

Tipul memoriei	Biţi utili u	Biţi de control C	Overhead O_1 = c / u [%]	Overhead O_2 = [%]
Fără cod de Paritate	8	0	-	-
	16	0	-	-
	32	0	-	-
Cu cod Hsiao	8	5	62,50	38,46
	16	6	37,50	27,27
	32	7	28,12	21,95

Tabelul 6.1

Se observă în tabelul 6.1 că, cu cât codul Hsiao folosit are o dimensiune mai mare (biţi utili şi biţi de control), cu atât overhead-ul scade. De exemplu dacă folosim codul Hsiao (13,8,5) cu 5 biţi de control avem un overhead de 38,46 %, dacă folosim codul Hsiao (22,16,6) cu 6 biţi de control avem un overhead de 27,27 %, iar în cazul folosirii codului Hsiao (39,32,7) cu 7 biţi de control avem un overhead de 21,95 %.

Se observă în fig. 6.3. că odată cu creşterea numărului de biţi utili ai codului Hsiao, scade overhead-ul pe partea de date.

Implementarea codului corector Hsiao la memoria cache TAG .

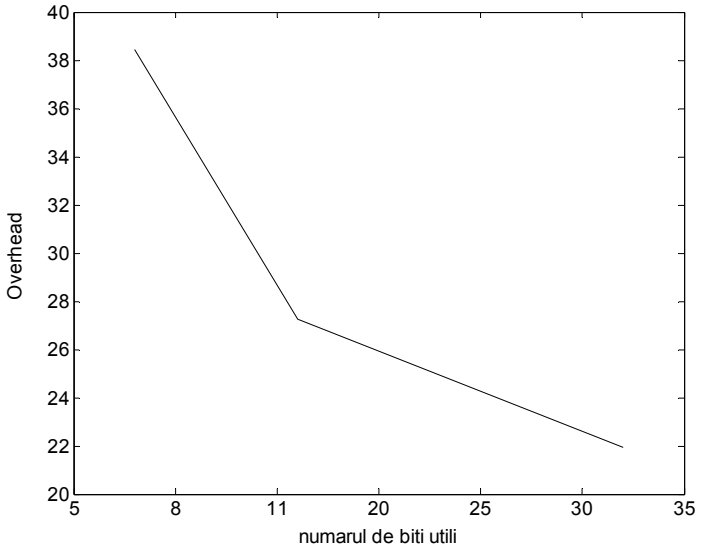


Fig. 6.3.

Tipul memoriei cache TAG	Biţi utili u	Biţi de control c	Overhead O_1=c/u [%]	Overhead O_2=[%]
Fără cod de Paritate	8	0	-	-
	16	0	-	-
	32	0	-	-
Cu cod de paritate pe C biţi	8	1	12,50	11,11
	16	2	12,50	11,11
	32	4	12,50	11,11
Cu cod de Paritate Hsiao	8	5	62,50	38,46
	16	6	37,50	27,27
	32	9	28,12	21,95

Tabelul 6.2

Tabelul 6.2, ce descrie introducerea codului Hsiao la memoria cache TAG de 128 K x 32 de biţi este identic cu tabelul 6.1. ce descrie introducerea codului Hsiao la memoria cache RAM.

6.2.1. Grafice. Tabele. Rezultate experimentale

Vom efectua simulări cu ajutorul mediului experimental CACHE. Vom alege pentru simulare următorii parametrii: dimensiunea memoriei principale - 1048576 KB, un singur nivel de cache, mapare directă, algoritm de înlocuire - LFU, politică de scriere - write through, fişier trace - cexp.prg. Dimensiunea blocului este de 8 B.

În fig. 6.4. este prezentată o simulare a funcţionării unei ierarhii de memorii, cu parametrii pe care i-am specificat, iar ca rezultat al simulării se pot observa Hit Rate şi Miss Rate.

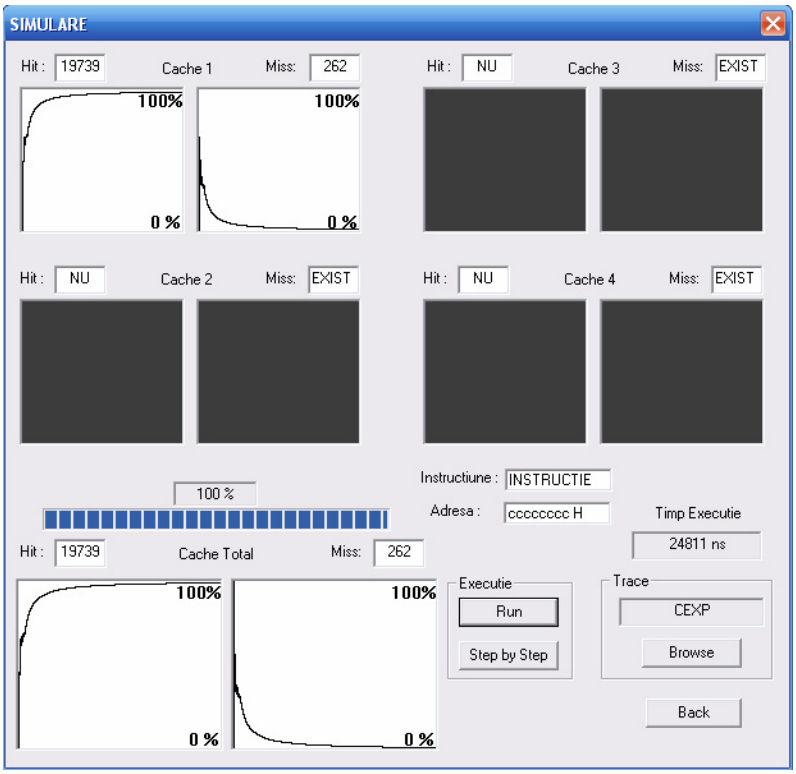


Fig. 6.4.

Exemplul 1.

Pentru această simulare alegem următorii parametrii: dimensiunea memoriei principale 1024 KB, un singur nivel de cache, mapare directă, algoritm de înlocuire - LFU, politică de scriere - write through, fişier trace - wave.prg. Pentru toate simulările care sunt prezentate în cadrul acestui exemplu, avem aceeaşi dimensiune a blocului de 8 B.

În fig. 6.5. am prezentat o simulare a funcţionării unei ierarhii de memorii, cu parametrii pe care i-am specificat, dimensiunea cache-ului este de 8 KB. Ca şi rezultat al simulării se pot observa Hit Rate şi Miss Rate.

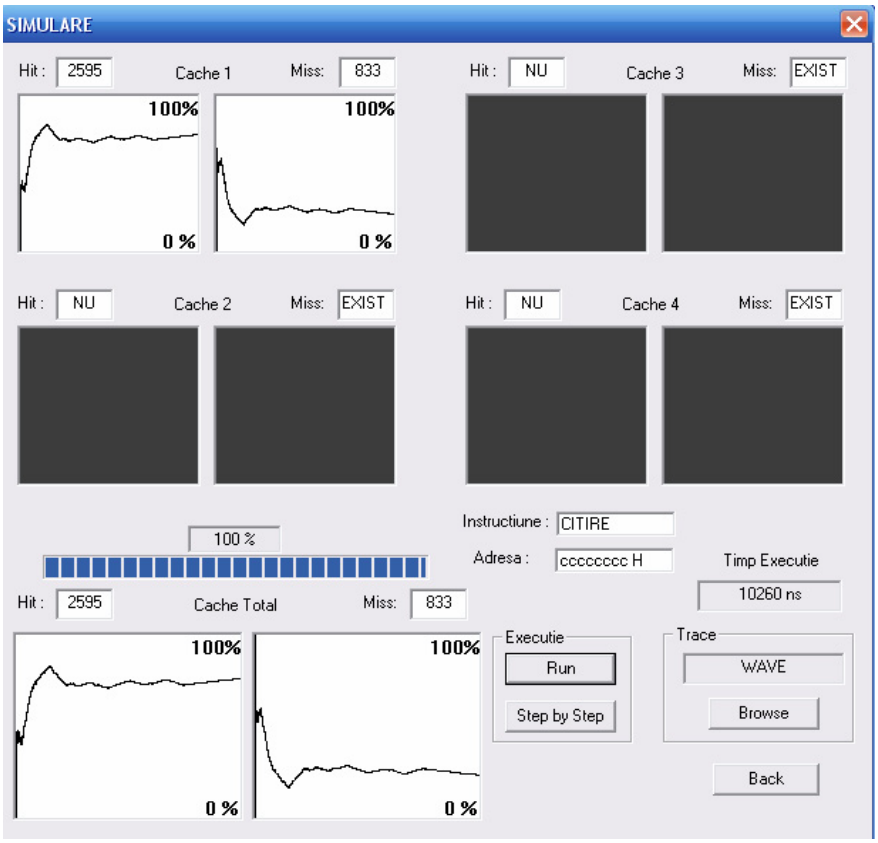


Fig. 6.5.

Dimensiune cache (KB)	Hit Rate Dimensiune bloc (8 B)
8	0,7570
16	0,7578
32	0,7651
64	0,7660
128	0,7660
256	0,7660
512	0,7660
1024	0,7660

Tabelul 6.3.

În tabelul 6.3. se observă la diferite dimensiuni ale cache-ului, valorile Hit

Rate-ului, obţinute cu simulatorul CACHE, pentru o dimensiune constantă a blocului de 8 B.

Exemplul 2.

Alegem pentru această simulare următorii parametrii: dimensiunea memoriei principale 1024 KB, un singur nivel de cache, mapare directă, algoritm de înlocuire - LFU, politică de scriere - write through, fişier trace - wave.prg. Pentru toate simulările care sunt prezentate în cadrul acestui exemplu avem aceeaşi dimensiune a blocului de 4 B.

Dimensiune cache (KB)	Hit Rate Dimensiune bloc (4 B)
8	0,6102
16	0,6108
32	0,6193
64	0,6196
128	0,6196
256	0,6196
512	0,6196
1024	0,6196

Tabelul 6.4.

În tabelul 6.4. se observă, la diferite dimensiuni ale cache-ului, valorile Hit Rate-ului, obţinute cu simulatorul CACHE, pentru o dimensiune constantă a blocului de 4 B.

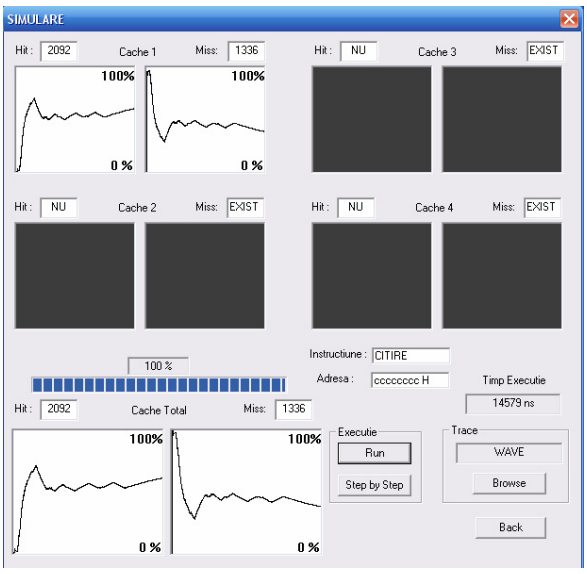


Fig. 6.6.

În fig. 6.6. am prezentat o simulare a funcţionării unei ierarhii de memorii cu parametrii pe care i-am specificat, dimensiunea cache-ului este de 8 KB. Ca şi rezultat al simulării se pot observa Hit Rate şi Miss Rate.

Exemplul 3.

Vom alege pentru această simulare următorii parametrii: dimensiunea memoriei principale 1024 KB, un singur nivel de cache, mapare directă, algoritm de înlocuire - LFU, politică de scriere - write through, fişier trace - wave.prg. Pentru toate simulările care sunt prezentate în cadrul acestui exemplu avem aceeaşi dimensiune a blocului de 2 B.

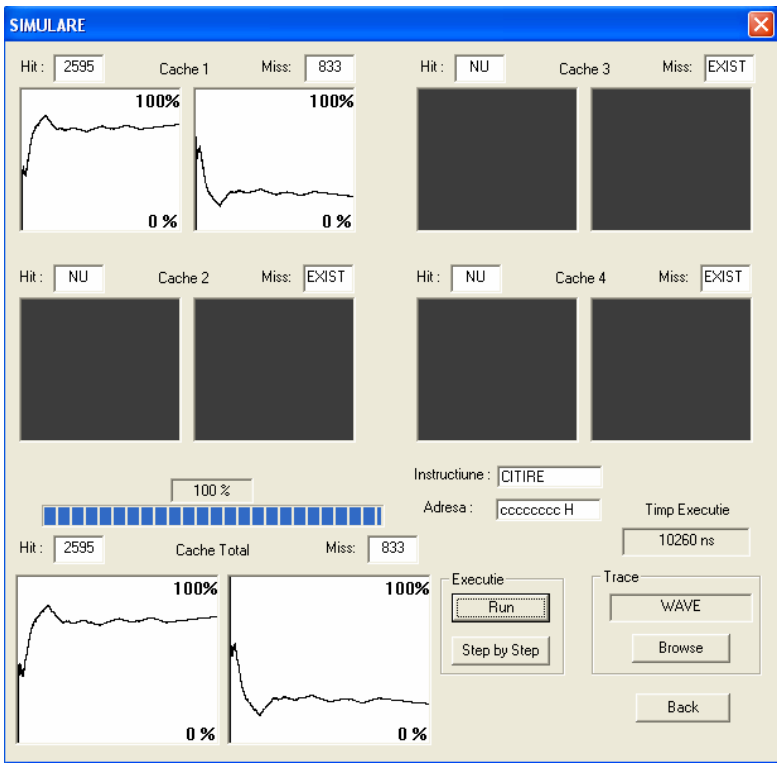


Fig. 6.7.

În fig. 6.7. am prezentat o simulare a funcţionării unei ierarhii de memorii cu parametrii pe care i-am specificat, dimensiunea cache-ului este de 8 KB. Ca şi rezultat al simulării se pot observa Hit Rate şi Miss Rate

Dimensiune cache (KB)	Hit Rate Dimensiune bloc (2 B)
8	0,6093
16	0,6096
32	0,6181
64	0,6184
128	0,6184
256	0,6184
512	0,6184
1024	0,6184

Tabelul 6.5.

În tabelul 6.5. se observă la diferite dimensiuni ale cache-ului, valorile Hit Rate-ului, obţinute cu simulatorul CACHE, pentru o dimensiune constantă a blocului de 2 B.

Exemplul 4.

Considerăm în cazul acestei simulări următori parametri: dimensiunea memoriei principale 1024 KB, un singur nivel de cache, mapare directă, algoritml de înlocuire - LFU, politică de scriere - write through, fişier trace - wave.prg. Pentru toate simulările care sunt prezentate în cadrul acestui exemplu, avem aceeaşi dimensiune a blocului de 1 B.

În fig. 6.8. am prezentat o simulare a funcţionării unei memorii cu parametrii pe care i-am specificat, dimensiunea cache-ului este de 8 KB.

Dimensiune cache (KB)	Hit Rate Dimensiune bloc (1 B)
8	0,6076
16	0,6079
32	0,6163
64	0,6166
128	0,6166
256	0,6166
512	0,6166
1024	0,6166

Tabelul 6.6.

În tabelul 6.6. se observă la diferite dimensiuni ale cache-ului, valorile Hit Rate-ului, obţinute cu simulatorul CACHE, pentru o dimensiune constantă a blocului de 1 B.

Fig. 6.8.

Ca şi rezultat al simulării se pot observa Hit Rate şi Miss Rate.

În tabelul 6.7, am prezentat sintetizat simulările pe care le-am realizat. Pe a doua coloană a acestuia, se prezintă Hit Rate-ul obţinut în urma simulărilor pentru dimensiunea blocului egală cu 1 B, în a treia coloană, avem Hit Rate-ul obţinut în situaţia în care blocul este egal cu 2 B. În a patra coloană, se prezintă Hit Rate-ul obţinut pentru dimensiunea blocului egală cu 4 B, iar în ultima coloană se prezintă Hit Rate-ul obţinut în urma simulărilor, pentru dimensiunea blocului egală cu 8 B.

Dimensiune cache (KB)	Hit Rate Dimensiune bloc 1 B	Hit Rate Dimensiune bloc 2 B	Hit Rate Dimensiune bloc 4 B	Hit Rate Dimensiune bloc 8 B
8	0,6076	0,6093	0,6102	0,7570
16	0,6079	0,6096	0,6108	0,7578
32	0,6163	0,6181	0,6193	0,7651
64	0,6166	0,6184	0,6196	0,7660
128	0,6166	0,6184	0,6196	0,7660
256	0,6166	0,6184	0,6196	0,7660
512	0,6166	0,6184	0,6196	0,7660
1024	0,6166	0,6184	0,6196	0,7660

Tabelul 6.7.

În continuare vom reprezenta grafic, în fig. 6.9., Hit Rate versus dimensiunea blocului din cache, pentru simulările din tabelul 6.7. În acest grafic se observă că, cu cât dimensiunea blocului din cache este mai mare, cu atât avem un Hit Rate mai mare. Reprezentarea acestor grafice este realizată în Matlab.

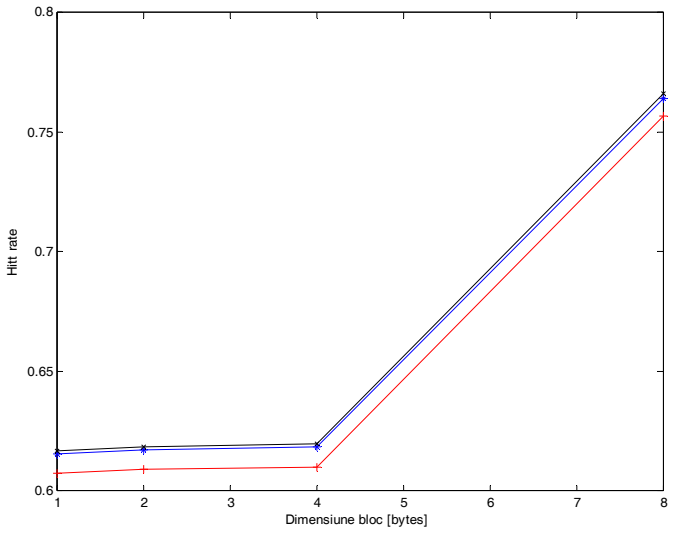


Fig. 6.9.

În fig. 6.10. vom reprezenta grafic, Hit Rate versus dimensiunea cache-ului, pentru simulările din tabelul 6.7. În acest grafic se poate observa că dacă dimensiunea blocului din cache este mai mare atunci avem un Hit Rate mai mare.

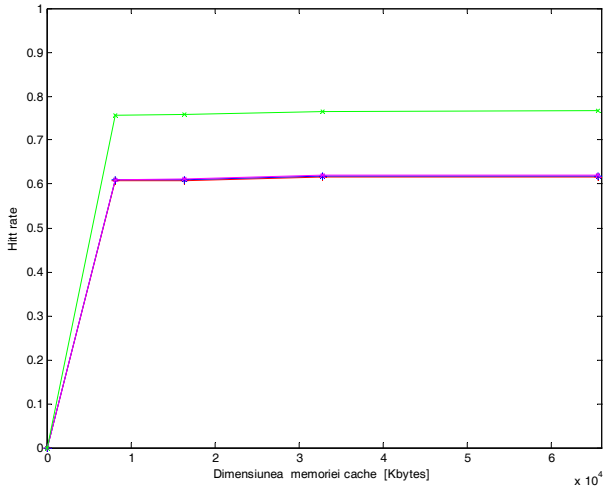


Fig. 6.10.

Exemplul 5.

Verificăm în cadrul acestui exemplu influenţa mapării cache-ului cu ajutorul simulatorului CACHE, astfel alegem pentru această simulare următorii parametrii: dimensiunea memoriei principale 1024 KB, un singur nivel de cache, mapare directă, algortim de înlocuire - LFU, politică de scriere - write through, fişier trace - wave.prg. Pentru toate simulările care sunt prezentate în cadrul acestui exemplu avem aceeaşi dimensiune, de 8 KB a cache-ului. După efectuarea simulărilor, vom obţine Hit Rate-urile, pe care le vom sintetiza în tabelul 6.8. Pe a doua coloană a acestuia, se prezintă Hit Rate-ul obţinut în urma simulărilor în cazul mapării directe, în a treia coloană, se prezintă Hit Rate-ul obţinut în urma simulărilor, în cazul mapării set-asociative pe 2 căi, iar în a patra coloană, se prezintă Hit Rate-ul obţinut în urma simulărilor în cazul mapării complet-asociative.

În fig. 6.11. vom reprezenta grafic Hit Rate versus dimensiunea blocului din cache, pentru mapările din tabelul 6.8. În acest grafic se poate observa că se obţine un Hit Rate mai bun pentru maparea complet asociativă decât Hit Rate-ul obţinut pentru maparea directă.

Dimensiune bloc (B)	Hit Rate Mapare directă	Hit Rate Mapare set-asociativă pe 2 căi	Hit Rate Mapare complet-asociativă
1	0,6076	0,6155	0,6166
2	0,6093	0,6172	0,6184
4	0,6102	0,6184	0,6196
8	0,7570	0,7642	0,7660

Tabelul 6.8.

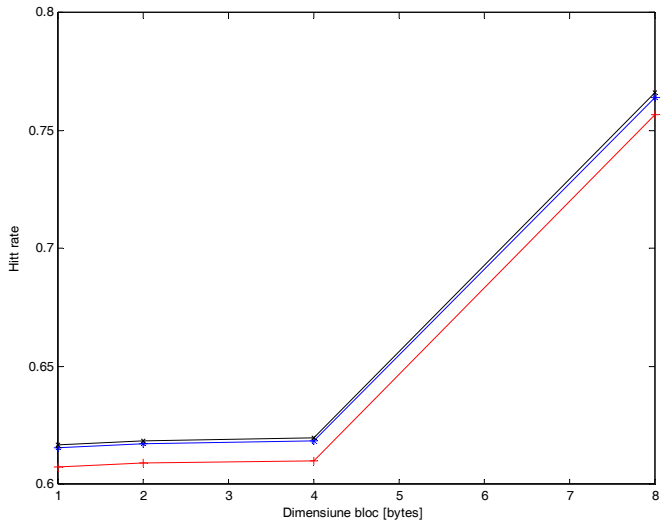


Fig. 6.11.

Exemplul 6.

Considerăm următorii parametrii de simulare: dimensiunea memoriei principale 1024 KB, un singur nivel de cache, mapare directă, algoritm de înlocuire - LFU, politică de scriere - write through, fişierul trace - swim.prg (trace-ul swim.prg are în total un număr de 20001 accese la memorie).

Pentru toate simulările care sunt prezentate în cadrul acestui exemplu avem aceeaşi dimensiune, de 8 KB a cache-ului. După efectuarea simulărilor cu ajutorul mediului CACHE, pentru un cache cu mapare directă, vom obţine Hit Rate şi Miss Rate.

În tabelul 6.9 am prezentat sintetizat simulările pe care le-am realizat. Pe a doua coloană a acestuia, se prezintă Hit Rate-ul obţinut în urma simulărilor în cazul mapării directe, în a treia coloană, se prezintă Hit Rate-ul obţinut în urma simulărilor în cazul mapării set-asociative pe 2 căi, iar în final, în a patra coloană, se prezintă Hit Rate-ul obţinut în urma simulărilor în cazul mapării complet-asociative.

Dimensiune bloc (B)	Hit Rate Mapare directă	Hit Rate Mapare set-asociativă pe 2 căi	Hit Rate Mapare complet-asociativă
1	0,5753	0,5814	0.6315
2	0,5770	0,5836	0.6351
4	0,5767	0,5787	0.6360
8	0,7469	0,7783	0.7895

Tabelul 6.9.

În fig. 6.12. vom reprezenta grafic Hit Rate versus dimensiunea blocului din cache. pentru mapările din tabelul 6.9. În acest grafic se poate observa că se obţine un Hit Rate mai bun pentru maparea complet asociativă decât Hit Rate-ul obţinut pentru maparea directă. De asemenea, se poate observa că Hit Rate-ul obţinut pentru maparea set - asociativă pe 2 căi este aproape identic cu Hit Rate-ul obţinut pentru maparea directă.

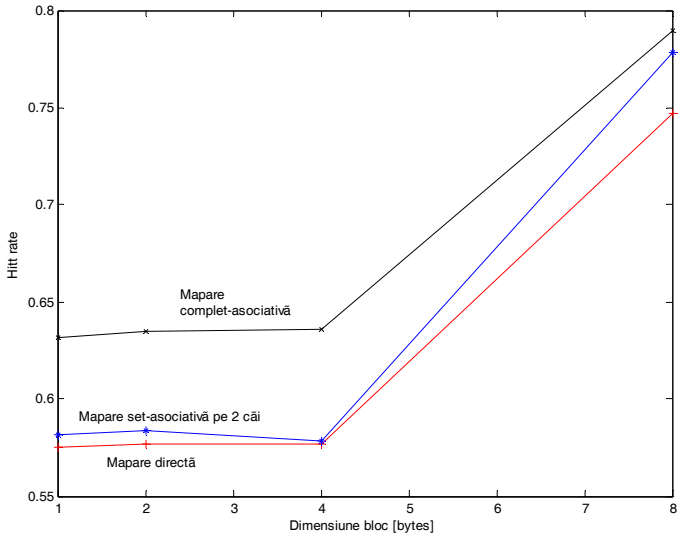


Fig. 6.12.

Exemplul 7.

În acest exemplu am considerat următorii parametrii: dimensiunea memoriei principale 1024 KB, un singur nivel de cache, mapare directă, algoritm de înlocuire - LFU, politică de scriere - write through, fişier trace - mdljd.prg (traces-ul mdljd.prg are în total un număr de 20001 accese la memorie). Pentru toate simulările care sunt prezentate în cadrul acestui exemplu avem aceeaşi dimensiune, de 8 KB a cache-ului. După efectuarea simulărilor, cu ajutorul mediului CACHE, vom obţine Hit Rate-urile din tabelul 6.10. În acest tabel pe a doua coloană se prezintă Hit Rate-ul obţinut în urma simulărilor în cazul mapării directe, în a treia coloană se prezintă Hit Rate-ul obţinut în urma simulărilor în cazul mapării set-asociative pe 2 căi, iar în a patra coloană, se prezintă Hit Rate-ul obţinut în urma simulărilor în cazul mapării complet-asociative.

Dimensi-une bloc (B)	Hit Rate Mapare directă	Hit Rate Mapare set-asociativă pe 2 căi	Hit Rate Mapare complet-asociativă
1	0,6807	0,7178	0,7299
2	0,6943	0,7304	0,7322
4	0,7024	0,7385	0,7396
8	0,8127	0,8310	0,8367

Tabelul 6.10.

În fig. 6.13. vom reprezenta grafic Hit Rate versus dimensiunea blocului din cache pentru mapările din tabelul 6.10. În acest grafic se poate observa că se obţine un Hit Rate mai bun pentru maparea complet asociativă decât Hit Rate-ul obţinut pentru maparea directă. De asemenea, se poate observa că Hit Rate-ul obţinut pentru maparea set-asociativă pe 2 căi este aproape identic cu Hit Rate-ul obţinut pentru maparea complet asociativă.

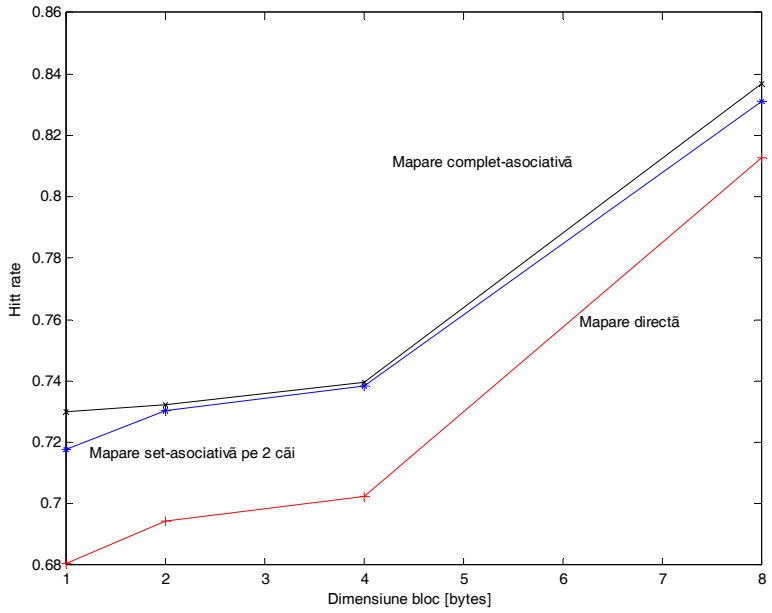


Fig. 6.13.

Exemplul 8.

Parametrii aleşi pentru această simulare sunt următorii: dimensiunea memoriei principale 1024 KB, un singur nivel de cache, mapare directă, algoritm de înlocuire - LFU, politică de scriere - write through, fişier trace - ear.prg (trace-ul ear.prg are în total un număr de 5309 accese la memorie). Pentru toate simulările care sunt prezentate în cadrul acestui exemplu avem aceeaşi dimensiune, de 8 KB a cache-ului. După efectuarea simulărilor cu ajutorul mediului CACHE, pentru un cache cu mapare directă, vom obţine Hit Rate-urile din tabelul 6.11. În acest tabel, pe a doua coloană, se prezintă Hit Rate-ul obţinut în urma simulărilor în cazul mapării directe, în a treia coloană, se prezintă Hit Rate-ul obţinut în urma simulărilor în cazul mapării set-asociative pe 2 căi, iar în a patra coloană, se prezintă Hit Rate-ul obţinut în urma simulărilor în cazul mapării complet-asociative.

Dimensiune bloc (B)	Hit Rate Mapare directă	Hit Rate Mapare set-asociativă pe 2 căi	Hit Rate Mapare complet-asociativă
1	0,7447	0,7666	0,7668
2	0,7534	0,7850	0,7852
4	0,7632	0,7950	0,7952
8	0,8592	0,8828	0,8834

Tabelul 6.11.

În fig. 6.14. vom reprezenta grafic Hit Rate versus dimensiunea blocului din cache pentru mapările din tabelul 6.11. În acest grafic, se poate observa că se obţine un Hit Rate mai bun pentru maparea complet asociativă decât Hit Rate-ul obţinut pentru maparea directă. De asemenea, se poate observa că Hit Rate-ul obţinut pentru maparea set-asociativă pe 2 căi este aproape identic cu Hit Rate-ul obţinut pentru maparea complet asociativă.

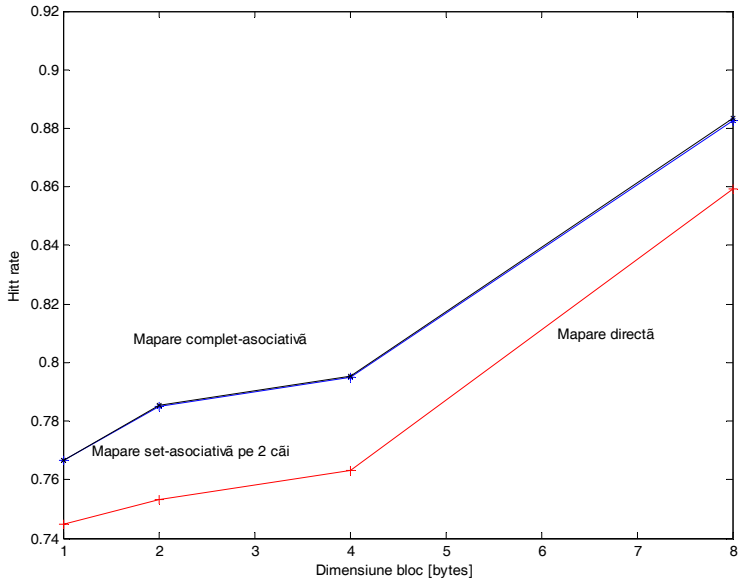


Fig. 6.14.

Exemplul 9.

Simularea din acest exemplu are următorii parametrii: dimensiunea memoriei principale 1024 KB, un singur nivel de cache, mapare directă, algoritm de înlocuire - LFU, politică de scriere - write through, fişier trace - hydro.prg (traces-ul hydro.prg are în total un număr de 5309 accese la memorie). Pentru toate simulările care sunt prezentate în cadrul acestui exemplu avem aceeaşi dimensiune, de 8 KB a cache-ului. După efectuarea simulărilor cu ajutorul mediului CACHE, pentru un

cache cu mapare directă, vom obţine Hit Rate-urile din tabelul 6.12. În acesta, pe a doua coloană, se prezintă Hit Rate-ul obţinut în urma simulărilor în cazul mapării directe, în a treia coloană, se prezintă Hit Rate-ul obţinut în urma simulărilor în cazul mapării set-asociative pe 2 căi, iar în a patra coloană, se prezintă Hit Rate-ul obţinut în urma simulărilor în cazul mapării complet-asociative.

Dimensi-une bloc (B)	Hit Rate Mapare directă	Hit Rate Mapare set-asociativă pe 2 căi	Hit Rate Mapare complet - asociativă
1	0,5441	0,5660	0,5671
2	0,5465	0,5682	0,5695
4	0,5484	0,5669	0,5714
8	0,7401	0,7546	0,7589

Tabelul 6.12.

În fig. 6.15. vom reprezenta grafic Hit Rate versus dimensiunea blocului din cache, pentru mapările din tabelul 6.12. În acest grafic se poate observa că se obţine un Hit Rate mai bun pentru maparea complet asociativă decât Hit Rate-ul obţinut pentru maparea directă.

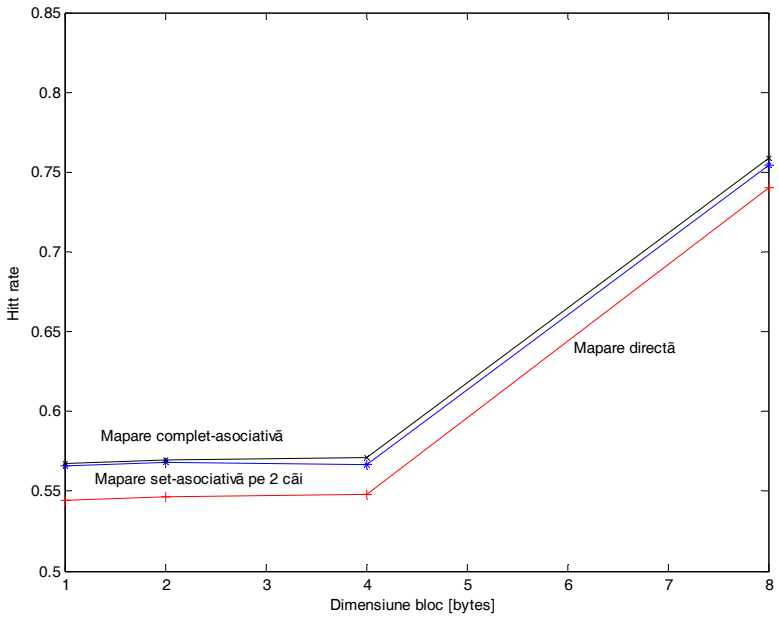


Fig. 6.15.

De asemenea se poate observa că Hit Rate-ul obţinut pentru maparea set-asociativă pe 2 căi este aproape identic cu Hit Rate-ul obţinut pentru maparea complet asociativă.

6.2.2. Mediul de simulare anterior simulatorului CDLR SPEC 2000

Intrările programului nostru sunt: N (numărul de niveluri), $MTTF_L$ - timpul mediu până la defectare pe nivelul L, $delay_L$, n (numărul de paşi de simulare). Ieşirea programului este mărimea CDLR.

Pentru a putea genera timpii de cădere aleatori pe fiecare nivel considerăm o distribuţie exponenţială a timpilor de cădere conform cu următoarea funcţie cumulativă de distribuţie:

$$F_L = 1 - e^{-\lambda_L t} \tag{6.16}$$

unde: $\lambda_L = 1 / MTTF_L$ reprezintă rata de defectare a nivelului L.

Pentru a obţine un timp de cădere aleator pentru nivelul L (t_L), începând cu variabila u, variabilă ce are valori aleatoare cuprinse în intervalul [0,1], putem scrie:

$$u = 1 - e^{-\lambda_L \cdot t_L}$$

$$t_L = \frac{1}{\lambda_L} \cdot \ln \frac{1}{1-u} \tag{6.17}$$

Pentru calculul mărimii CDLR în pasul i vom folosi următoarea formulă de calcul:

$$CDLR_i = \frac{\overline{Z_1} + \overline{Z_2} + \ldots + \overline{Z_i}}{i} \tag{6.18}$$

unde: - i este pasul simulării din tabelul 6.13., iar Z_i este 1 dacă avem succes şi 0 în caz contrar.

Am considerat în cadrul programului de simulare un număr n = 5000 paşi de simulare.

Rezultatele comparative între mărimea CDLR, obţinută prin modelul matematic (CDLR analitic) şi mărimea CDLR obţinută prin simulare (CDLR simulat), pentru trei cazuri ce sunt prezentate în tabelul 6.14. Eroarea relativă ε, se calculează cu formula 6.19. şi este egală cu raportul dintre valoarea absolută a diferenţei între CDLR-ul analitic şi CDLR-ul simulat, şi CDLR-ul analitic.

Rezultatele parţiale pentru câteva simulări sunt date în tabelul 6.13 [NoGN05].

Pasul Nr.	t1 t1 < 2	Succes	t2 t2 <300	Succes	t3 t3 < 86400	Succes	t4 t4 < 1576800000	Succes	CDLR [%]
1	3897025	1	23343054	1	38065129	1	33426393219557040000	1	0.000000000
2	20416098	1	67086060	1	41379366	1	2551845032950944200	1	0.000000000
3	11919781	1	1629043	1	25024637	1	28605754501451772000	1	0.000000000
4	35517632	1	11275845	1	5650021	1	10154498572615608000	1	0.000000000
5	18596521	1	137420722	1	105440484	1	60937820941717320000	1	0.000000000
.....									
2612	28085836	1	200	0	21943970	x	1740184782032714500	x	0.038284839
3826	6760802	1	8562634	1	46211	0	397452068417875520	x	0 .052273915
3881	18739564	1	3225363	1	76263	0	11808395109897003000	x	0.077299665
3978	14109592	1	408178	1	43900	0	12513132994014900000	x	0.100553042
4733	17793714	1	16958944	1	78575	0	15732256004728312000	x	0.105641242
4919	60183794	1	65691306	1	23102	0	3165487150413436900	x	0.121976011
.....									
4999	2845723	1	23453231	1	143679403	1	15015473684324770000	1	0.120024005
5000	22188127	1	32012677	1	31884861	1	45659748193829200000	1	0.120000000

Tabelul 6.13.

$$\varepsilon = \frac{\left|CDLR\ analitic - CDLR\ simulat\right|}{CDLR\ analitic} \qquad (6.19)$$

NIVELURI	NIVEL MTTF [s]	DELAY NIVEL[s]	CDLR analitic[%]	CDLR simulat[%]	Epsilon[%]
5	25920000	2	0.21462231	0.20683682|	3.62753111
	50457600	300			
	75686400	86400			
	100012100	100000			
	9223372036854775800	1576800000			
4	25920	2	11.46819529	11.08006408	3.38441401
	504576	300			
	756864	86400			
	92233720368547758	1576800000			
4	25920000	2	0.11474915	0.11858830	3.34568939
	50457600	300			
	75686400	86400			
	9223372036854775800	1576800000			

Tabelul 6.14.

Se poate observa că eroarea relativă are valori sub 4% (practic are valori între 3.34% si 3.62%).

6.2.3. Exemplu

Considerăm în acest exemplu o ierarhie de memorii cu patru niveluri. Pentru datele din tabelul 6.14. am calculat atât MTTDL–ul cât şi CDLR-ul. MTTDL-ul întregii ierarhii nu este afectat de politica de transfer între niveluri.

Nivel	MTTF	Delay Întârzierea
1. Cache	10 luni	2 secunde
2. Memorie	1.6 ani	5 minute
3. Disc	2.4 ani	1 zi
4. Banda	50 ani	∞
MTTDL=5.3 luni DLR=0.001163		

Tabelul 6.15.

Pentru politica de scriere aleasă, delayed-write, CDLR-ul global rezultat este de 0,1163%, care este de aproximativ 10 ore de date pierdute pe an. Este interesantă observaţia că, crescând fiabilitatea unui nivel, putem lăsa datele să stea mai mult timp pe acel nivel, menţinând acelaşi CDLR global.

6.3. CONCLUZII

În acest capitol am ales un cod corector SEC-DED, Hsiao, pe care l-am aplicat ulterior atât pentru detecţia cât şi pentru corecţia erorilor din cadrul cache-ului şi din cadrul memoriei principale. Am implementat acest cod la memoria principală DRAM şi la memoria cache TAG, folosind un cod Hsiao (13,8,5). Am determinat pentru aceste memorii, overhead-ul pe partea de date şi overhead-ul datorat circuitelor supliimentare pentru corecţia erorii.

Am implementat un cod Hsiao (39,32,7), la memoria cache RAM şi am determinat overhead-ul atât pe partea de date, cât şi overhead-ul datorat circuitelor supliimentare pentru corecţia eroriii. De asemenea, am implementat codul corector pentru memoria cache RAM, cu ajutorul circuitelor programabile FPGA Xilinx din

seria XC4000, pentru a putea determina overhead-ul datorat circuitelor suplimentare pentru corecţia erorii.

Am analizat prin intermediul unor tabele şi grafice aceste implementări şi am ajuns la concluzia că, cu cât creşte numărul de biţi utili ai codului cu atât scade overhead-ul pe partea de date. Am prezentat prin graficce şi tabele rezultate experimentale obţinute cu ajutorul mediilor de evaluare.

7. CONCLUZII

7.1. CONCLUZII

Obiectivul urmărit în această teză de doctorat este încercarea autorului de a contribui la dezvoltarea şi la eficientizarea metodelor de creştere a dependabilităţii la ierarhii de memorii. Punctul de plecare a acestei teze a fost dezvoltarea stadiului memoriilor cache prin creşterea dependabilităţii şi a fiabilităţii acestor memorii. Lucrarea este structurată pe şapte capitole ce au următorul conţinut:

Capitolul 1: *„Asupra stadiului actual al configuraţiilor de cache-uri”* cuprinde stadiile actuale ale cunoaşterii în domeniul cache-urilor, locul treptei cache într-o ierarhie de memorii, arhitectura generală a cache-urilor, configuraţii şi exemple de cache-uri, problemele cache-urilor şi perturbarea acestora prin defectare, unde sunt tratate: problema mapării adreselor, problema replasării, problema scrierii. Tot aici se realizează analiza unor soluţii existente de cache-uri cu capabiltăţi de îmbunătăţire a dependabilităţii, unde se prezintă toleranţa la defectare prin redimensionarea cache-ului.

Capitolul 2: *„Analiza defectelor care pot afecta cache-urile”* tratează defectele la nivelul celulei SRAM şi la nivelul legăturilor dintre celulele SRAM. Analiza tipurilor de defecte prezintă defecte generale ale memoriilor, defecte permanente, defecte intermitente, defecte tranzitorii, defecte specifice ale cache-urilor. Putem spune că nu am făcut o prezentare exhaustivă, rezumându - ne la defectele specifice ale cache-urilor, defecte care sunt acoperite de codurile corectoare şi detectoare de erori care s-au folosit ulterior.

Capitolul 3: *„Configurarea structurii unui cache tipic urmărind stabilirea locului de introducere a elementelor de redundanţă menite creşterii dependabilităţii”.* În acest capitol se tratează interfaţarea cache-ului cu celelalte trepte ale unei ierarhii de memorii, se realizează detalierea constructivă a unui cache, sincronizarea funcţională a treptei cache şi se determină locul de introducere a elementelor redundante care asigură creşterea dependabilităţii. S-a realizat un exerciţiu de proiectare a unui cache în vederea stabilirii locului de introducere a redundanţei, acesta cuprinzând interfaţa cache-ului cu alte blocuri ale configuraţiei de calcul. Proiectarea este realizată printr-o metodă ordonată pe paşi. Se aplică

metoda pe un exemplu concret de proiectare care conţine inclusiv interconectarea cache-ului cu memoria principală, în vederea stabilirii locului de introducere a redundanţei în ierarhia de memorii.

Capitolul 4: *„Analiza metricilor de performanţă şi fiabilitate la cache-uri”.* În acest capitol se realizează analiza metricilor de performanţă şi fiabilitate, se propune un nou indicator de evaluare a dependabilităţii şi se prezintă metode de îmbunătăţire a performanţei cache-urilor şi afectarea lor prin CDLR.

Capitolul 5: *„Mediul de experimentare reprezentat de simulatorul de cache*". Acest capitol prezintă mediul experimental CACHE şi mediul de evaluare CDLR SPEC 2000, obiectivele de construcţie ale simulatorului şi detalii constructive ale acestora.

Capitolul 6: *„Rezultate experimentale obţinute prin intermediul simulatorului CDLR SPEC 2000”* . În acest capitol se realizează implementarea cu circuite programabile FPGA Xilinx a unui cod SEC-DED şi se evaluează indicii de fiabilitate. Sunt prezentate structuri de cache cu facilităţi de detecţie şi corecţie pe baza proprietăţilor favorabile ale codului Hsiao, se face alegerea acestui cod, care se va aplica ulterior atât pentru detecţia şi corecţia erorilor din cadrul cache-ului, cât şi pentru detecţia şi corecţia erorilor din cadrul memoriei principale. Se realizează implementarea codului Hsiao la memoria principală DRAM şi se determină overhead-ului rezultat. Se implementează codul Hsiao la memoria cache TAG, şi se determină overhead-ul pe partea de TAG. De asemenea, este implementat codul Hsiao la memoria cache RAM, şi este determinat overhead-ul. Se implementează codul corector pentru o memorie cache RAM, cu ajutorul circuitelor programabile FPGA Xilinx şi se determină overhead-ul datorat circuitelor suplimentare pentru corecţia erorii. Tot aici sunt prezentate rezultatele experimentale ale aplicării acestor medii, prin grafice şi tabele.

Capitolul 7:”Concluzii”. În acest capitol se prezintă concluziile şi contribuţiile personale.

7.2. CONTRIBUŢII PERSONALE

În cadrul tezei de docorat intitulată „***Cercetări ale eficienţei metodelor de creştere a dependabilităţii la treapta cache a unei ierarhii de memorii***” au fost urmărite şi realizate următoarele deziderate care sunt totodată contribuţii ale lucrării:

1. Elaborarea unui studiu privind stadiul actual în domeniul cache-urilor;

2. Întocmirea unei clasificări a defectelor la nivelul celulei SRAM şi la nivelul legăturilor dintre celulele SRAM;

3. În cadrul studiului metricilor de fiabilitate o altă contribuţie este introducerea mărimii CLDR (Cache Data Loss Rate);

4. Elaborarea unei metode de proiectare ordonată pe etape a unui cache;

5. Stabilirea locului de introducere a redundanţei în cadrul unui cache;

6. Alegerea codului corector Hsiao (SEC-DED), care va fi aplicat ulterior atât pentru detecţia şi corecţia erorilor din cadrul cache-ului cât şi pentru detecţia şi corecţia erorilor din cadrul memoriei principale;

7. Implementarea codului Hsiao la memoria principală DRAM, la memoria cache TAG, la cache RAM şi determinarea overhead-ului pe partea de date pentru aceste memorii;

8. Implementarea codului corector pentru memoria cache RAM, cu ajutorul circuitelor programabile FPGA Xilinx şi determinarea overhead-ului datorat circuitelor suplimentare pentru corecţia erorii;

9. Implementarea mediului CACHE şi simularea cu acesta a funcţionării unei ierarhii de memorie;

10. Implementarea mediului CDLR SPEC 2000 şi simularea cu acesta a funcţionării unei ierarhii de memorie, precum şi calculul mărimii CDLR;

O parte din preocupările actuale sunt axate pe creşterea disponibilităţii şi în acest sens a fost prezentată o soluţie general aplicată. De asemenea au fost discutate şi câteva soluţii care duc la creşterea disponibilităţii prin toleranţa la defecte.

Din acest studiu a reieşit că cercetările încă continuă, deoarece aceste cache-uri care au atribute de creşterea disponibilităţii prin toleranţă la defecte sunt de actualitate şi pot fi aplicabile, o astfel de metodă putând fi aplicarea unui cod corector asupra unui cache.

LISTA LUCRĂRILOR

A. Lucrări reprezentative:

1. **Ovidiu Novac**, Mihaela Novac, - *Comparative study of cache memory performance improvement*, International Conference on Engineering of Modern Electric Systems, EMES'99, Felix-Spa, România, pp.127 - 130, ISSN-1223 – 2106, 27 - 29 Mai 1999.

2. **Ovidiu Novac**, Mihaela Novac*, - Aspects regarding the improvement of cache memory performance,* International Conference on Engineering of Modern Electric Systems, EMES'01, Universitatea Oradea, România, pp. 153 - 156, ISSN - 1223-2106, Mai 2001.

3. **Ovidiu Novac,** Mihaela Novac, - *Aspecte privind capacitatea de testare a sistemelor de memorii,* Analele Universităţii din Oradea, Fascicola Colegiului Tehnic, Economic şi de Admin., Oradea, pp. 73 - 76, 2001.

4. **Ovidiu Novac**, Mihaela Novac, - *Aspects regarding cache memory simulators*, International Conference on Renewable Sources and Environmental ElectroTehnologies RSEE 2002, România, Universitatea Oradea, pp. 199 - 203, ISSN 1223 – 2106, 2002.

5. Ştefan Vari Kakas, **Ovidiu Novac**, - *Cache modeling and performance comparison by simulations,* 7' th International Conference on Engineering of Modern Electric Systems, EMES'03, Universitatea Oradea, România, pg. 199 - 203, ISSN 1223 – 2106, 29 - 31 Mai, 2003.

6. **Ovidiu Novac**, Ştefan Vari Kakas, Otto Poszet, Mihaela Novac, *Aspects regarding the EVASMXP work environment,* International Conference on Renewable Sources and Environmental ElectroTehnologies RSEE 2004, România, Universitatea Oradea pp 173 - 175, ISSN 1223 – 2106, 2004.

7. Otto Poszet, Iosif Ignat, Ştefan Vari Kakas, **Ovidiu Novac**, - *Methods for cyclic codes decodification,* International Conference on Renewable Sources and Environmental ElectroTehnologies RSEE 2004, România, Universitatea din Oradea, pp. 61 - 65, ISSN 1223 – 2106, 2004.

8. Otto Poszet, Ştefan Vari Kakas**, Ovidiu Novac**, Horatiu Drăgan, Iosif Ignat, - *Efficiency Of Identification Protocols in Electronic Payment System,* International Conference on Engineering of Modern Electric Systems, EMES'05, University of Oradea, România, pg. 118 - 121, ISSN 1223 – 2106, 2005.

9. Ştefan Vari Kakas, **Ovidiu Novac**, Otto Poszet, Mihaela Novac, *Reliability Considerations in Memory Hierarchies*, International Conference on Engineering of Modern Electric Systems, EMES'05, University of Oradea, România, pp. 149 - 152, ISSN 1223 – 2106, 2005.

10. **Ovidiu Novac,** Mircea Gordan, Mihaela Novac, - *Data Loss Rate versus Mean Time To Failure in Memory Hierarchies,* International Conferences on Computer, Information, and Systems Sciences, and Engineering (CIS^2E 05), University Bridgeport USA., pp. 305 - 307, Published by Springer, ISBN 1 – 4020 – 5262 – 6, December 10 – 20, 2005.

11. **Ovidiu Novac**, Ştefan Vari Kakas, Otto Poszet, Mihaela Novac, *Consideration about single error correction and double error detection in a memory hierarchy*, International Conference on Renewable Sources and Environmental ElectroTehnologies, RSEE 2006, România, Universitatea Oradea, pp 32 - 34, ISSN 1841 – 7221, 2006.

12. Otto Poszet, Ştefan Vari Kakas, **Ovidiu Novac,** Iosif Ignat, - *Fault Tolerant Protocols Used in Electronic Payment System,* International Conference on Renewable Sources and Environmental ElectroTehnologies, RSEE 2006, România, Universitatea Oradea, pp 39 - 42, ISSN 1841 – 7221, 2006.

13. **Ovidiu Novac,** Mihaela Novac, - *Teoretical and experimental study regarding the simulation of a memory hierarchy,* EPE, Buletinul Institutului Politehnic Iaşi, Tomul LII (LVI), fasc. 5, Electrotehnică, Energetică, Electronică, pp. 915 - 920**,** ISSN 1223 – 8139, 2006.

14. **Ovidiu Novac**, Mircea Vlăduţiu, Ştefan Vari Kakas, Mihaela Novac, Mircea Gordan, *A Comparative Study Regarding a Memory Hierarchy with the DLR SPEC 2000 Simulator*, International Conferences on Computer, Information, and Systems Sciences, and Engineering (CIS^2E 06), University Bridgeport USA, pp.369-372, Published by Springer, ISBN 978 – 1 - 4020-6267 – 4, December 10 - 20, 2006.

15. **Ovidiu Novac**, Ştefan Vari Kakas, Otto Poszet, Mihaela Novac, - *Aspects Regarding the use of Error Detecting and Error Correcting Codes in Cache Memories*, International Conference on Engineering of Modern Electric Systems, EMES'07, University of Oradea, România, pg. 72 – 74, ISSN 1223-2106, 2007.

16. Otto Poszet, Ştefan Vari Kakas, **Ovidiu Novac**, - *Performance assessment of an Electronic Payment system by simulation,* International Conference on Engineering of Modern Electric Systems, EMES'07, University of Oradea, România, pg. 100 – 106, ISSN 1223-2106, 2007.

17. Otto Poszet **, Ovidiu Novac**, Ştefan Vari Kakas, - *Analysis of an electronic Payment System Using Simulation*, Proceedings of the 8th International Symposium of Hungarian Researchers on Computational Intelligence and Informatics, CINTI 2007, Budapest, Hungary, ISBN 978-963-7154-65-2,15-17 November, 2007.

18. **Ovidiu Novac**, Mircea Vlăduţiu, Ştefan Vari Kakas, Ioan Francisc Hathazi, Mihaela Novac, - *Aspects Regarding the use of SEC-DED Codes to the Cache Level of a Memory Hierarchy*, AIKED-Knowledge Engineering, ISI Books, Cambridge (UK), (Lucrare acceptată spre publicare) , 20-22 February, 2008.

REFERINŢE BIBLIOGRAFICE

[AdCo96] R.D. Adams and E.S. Cooley, *"Analysis of a Deceptive Destructive Read Memory Fault Model and Recommended Testing"*, *In Proc. IEEE North Atlantic Test Workshop*, 1996.

[AgPu93] A. Agarwal and S. D.Pudar, *"Column-Associative Caches: a Technique for Reducing the Miss Rate of Direct-Mapped Caches", Int'l Symp. on Computer Architecture*, pp. 179 - 190, 1993.

[APMD05] A. Agarwal, B. C. Paul, H. Mahmoodi, A. Datta and K. Roy - *"A Process-Tolerant Cache Architecture for Improved Yield in Nanoscale Technologies"*, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 13, No. 1, pp. 27 - 38, January 2005.

[AMZE04] G. Asadi, S. G. Miremadi, H. R. Zarandi, A. R. Ejlali, *"Evaluation of Fault-Tolerant Designs Implemented on SRAM-based FPGAs"*, *Proceedings IEEE/IFIP Pacific Rim International Symposium on Dependable Computing, French*, pp. 327 - 333, 2004.

[ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, *"Basic Concepts and Taxonomy of Dependable and Secure Computing"*, *IEEE Transactions on Dependable and Secure Computing*, Vol.1, No.1, pp 11 - 33, January-March 2004.

[BYUC01] Brigham Young University: BYU Cache Simulator. http://tds.cs.byu.edu, 2001.

[CaGr96] B. Calder, D. Grunwald, *"Predictive Sequential Associative Cache"*, *Proceedings of 2nd Int'l Symposium. High Performance Computer Architecture*, pp. 244 - 253, 1996.

[CRST06] J. Chang, Şt. Rusu, J. Shoemaker, S. Tam, M. Huang, M. Haque, et al., *"A 130-nm Trimple-Vt 9-MB Third-Level On-Die Cache for the 1.7-GHz Itanium 2 Processor",* Journal on Solid State Circuits, vol. 40, no. 1, pp. 195 – 203, 2006.

[ChLo99] P. M. Chen, D. E. Lowell *„Reliability Hierarchies", Workshop in Operating Systems*, 1999.

[ChCh01] H. Chen, J. Chiang, *"Design of an Adjustable-way Set-Associative Cache"*, *Proceedings Pacific Rim Communications, Computers and signal Processing*, pp. 315 - 318, 2001.

[De94] K. De et al., *"RSYN: A system for automated synthesis of reliable multilevel circuits", IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 2, pp. 186 - 195, June 1994.

[FaHi93] A. Faridpour, M. Hill, *"Performance Implications of Tolerating Cache Faults"*, *IEEE Transactions on Computers*, Vol. 42, No. 3, pp 257 - 267, 1993.

[FaOS03] A. A. Farooqui, V. G. Oklobdzija, S. M. Sait, *"Area-Time Optimal Adder with Relative Placement Generator"*, *Proceedings of International Symposium on Circuits and Systems*, Vol. 5, pp. 141 - 144, 2003.

[GAMC02] P. Gil, J. Arlat, H. Madeira, Y. Crouzet, T. Jarboui, K. Kanoun, T. Marteau, J. Durães, M. Vieira, D. Gil, J.-C. Baraza, J. Gracia, *"Fault Representativeness"*, Dependability Benchmarking, IST-2000-2542, LAAS-CNRS (France) and Partners, 2002.

[GoAl00] Ad J. van de Goor, Zaid Al-Ars, *"Functional Memory Faults: A Formal Notation and a Taxonomy"*, *Proceedings IEEE International Test Conference*, 2000.

[Goor98] Ad J. van de Goor, *"Testing Semiconductor Memories, Theory and Practice"*, ComTex Publishing, Gouda, The Nederlands,1998.

[HePa96] J. L. Hennessy, D. A. Patterson, *"Computer Architecture. A Quantitative Approach"*, 2nd Edition, *San Mateo, Morgan-Kaufmann Publishing Co.*, 1996.

[HePa03] J. L. Hennessy, D. A. Patterson, *"Computer Arhitecture. A Quantitative Approach"*, 3rd Edition, *San Mateo, CA, Morgan-Kaufmann Publishing Co.,* 2003.

[Hori97] M. Horiguchi, *"Redundacy techniques for high-density DRAMS", Proceedings of 2nd Annual IEEE International Conference Systems Silicon*,, pp. 22 - 29, October, 1997.

[Hung06] L. D. Hung, *"Soft Error Tolerant Cache Architectures",* PhD Thesis, Department of Information Science and Technology, University of Tokyo, December 2006.

[Imai90] H. Imai, *"Essentials of Error-Control Coding Techniques" Academic Press*, San Diego, 1990.

[Inte06] *http://intel.com/design/Pentium4/specupdt/249199.htm*, 2006

[JhWa93] N. K. Jha, S.-J. Wang, *„Design and synthesis of self-checking VLSI circuits"*, *IEEE Transaction on Computer-Aided Design*, vol. 12, pp. 878 - 887, June 1993.

[KaGh97] M. B. Kamble, K. Ghose, "Energy-Efficiency of VLSI Cache: A Comparative Study". *Proceedings of IEEE 10th International. Conference on VLSI Design,* pp. 261-267, 1997.

[Kess89] R. R. Kessler, et al., *"Inexpensive Implementations of Associativity"*, *Proceedings of International Symposium Computer Architecture*, pp. 131 - 139 , 1989.

[KiSo99] S. Kim, A. Somani, *"Area Efficient Architectures for Information Integrity Checking in the Cache Memories"*, *Proceedings of International Symposium Computer Architecture,* pp. 246 - 256, 1999.

[LaWe95] C.-S. Lai, C.-L. Wey, *"SOLiT: an automated system for synthesizing reliable sequential circuits with multilevel logic implementation"*, *IEEE Proc.-Comput. Digit. Tech*., vol. 142, pp. 49 - 54, January 1995.

[LeLK00] J. H. Lee, J. S. Lee, S. D. Kim, *"A New Cache Architecture based on Temporal and Spatial Locality"*, *Journal of Systems Architecture,* Vol. 46 , pp. 1452 - 1467, 2000

[MaMŢ99] G. E. Mang, I. Mang , R. Ţirtea, *„Proiectarea logică cu FPGA", Editura Universităţii din Oradea*, 1999.

[MaŢi00] G. E. Mang, R. Ţirtea, *„Proiectarea logică în VHDL. Aplicaţii practice", Editura Universităţii din Oradea*, 2000.

[MiMi03] A. Milenkovic, M. Milenkovic, *"Exploiting Streams in Instruction and Data Address Trace Compression"*, *Electrical and Computer Engineering Department*, University of Alabama, 2003.

[MeFR97] C. Metra, M. Favalli, and B. Ricco, *"Highly Testable and Compact Single Output Comparator"*, In *Proc. of IEEE VLSI Test Symp.*, pages 210 – 215, 1997.

[NiOR97] D. Niggemeyer, J. Otterstedt, M. Redeker, *"A Defect-Tolerant DRAM employing a Hierarchical Redundancy Scheme, Built-In Self-Test and Self-Reconfiguration"*, *IEEE International Workshop on Memory Technology, Design, and Testing*, San Jose, pp. 33 - 40, 1997.

[NHKK99] S. Nakahara, K. Higeta, M. Kohno, T. Kawamura, K. Kakitani , *"Built-in-self-test for GHz Embedded SRAMS Using Flexible Pattern Generator And New Rapair Algorithm"*, In *Proceedings of International Test Conference*, pp. 301 - 310, 1999.

[NoON96] P. Nordholz, J. Otterstedt, D. Niggemeyer, *"A Defect-Tolerant Word-Oriented Static RAM with Built-In Self-Test and Self-Reconfiguration", Innovative Systems in Silicon (ISIS)*, pp. 124 -132, October 1996.

[NoNo99] **O**. **Novac**, Mihaela Novac, *"Comparative study of cache memory performance improvement",* 5'th International Conference on Engineering of Modern Electric Systems, EMES'99, pp. 127-130, Felix-Spa, România, pp.127 - 130, ISSN-1223-2106, 27-29 Mai, 1999.

[NoNo01] **O**. **Novac**, Mihaela Novac, *"Aspects regarding the improvement of cache memory performance",* International Conference on Engineering of Modern Electric Systems, EMES'01, pp. 153 - 156, University of Oradea, ISSN-1223 – 2106, Mai, 2001.

[NovN01] **O**. **Novac**, Mihaela Novac, *„Aspecte privind capacitatea de testare a sistemelor de memorii"*, Analele Universităţii din Oradea, Fascicola Colegiului Tehnic, Economic şi Administrativ, pp. 73 - 76, 2001.

[NoNo02] **O**. **Novac**, Mihaela Novac, *„Aspects regarding cache memory simulators"*, *International Conference on Renewable Sources and Environmental Electro-Technologies*, RSEE 2002, pp. 199 - 203, University of Oradea, 2002.

[NoGN05] **O. Novac,** M. Gordan, Mihaela Novac, *"Data Loss Rate versus Mean Time To Failure in Memory Hierarchies"*, International Conferences on Computer, Information, and Systems Sciences, and Engineering (CIS^2E 05), University of Bridgeport, USA, ISBN 1 – 4020 – 5262 – 6, December 10 - 20, 2005.

[NVPN06] **O**. **Novac**, Şt. Vari Kakas, O. Poszet, Mihaela Novac, *"Consideration about single error correction and double error detection in a memory hierarchy", International Conference on Renewable Sources and Environmental Electro-Technologies RSEE 2006,* pp. 32 - 34, University of Oradea, 2006.

[NoNP06] **O**. **Novac**, Mihaela Novac, V. Paşca, *"Theoretical and experimental study regarding the simulation of a memory hierarchy*, EPE, Buletinul Institutului Politehnic Iaşi, Tomul LII (LVI), fascicola 5B, Electrotehnică, Energetică, Electronică, pp. 915 - 920, 2006.

[NVVN06] **O**. **Novac**, M. Vlăduţiu, Şt. Vari Kakas, Mihaela Novac, M. Gordan, *"A Comparative study Regarding a Memory Hierarchy with the CDLR SPEC 2000 Simulator", International Conferences on Computer, Information, and Systems Sciences, and Engineering (CIS^2E 06),* University of Bridgeport, USA, ISBN 978 – 1 – 4020 - 6267 - 4, December 2006.

[NoVP07] **O**. **Novac**, Şt. Vari Kakas, O. Poszet, *"Aspects Regarding the use of Error Detecting and Error Correcting Codes in Cache Memories", EMES'07*, University of Oradea, ISSN 1223 – 2106, June 2007.

[NVVH08] **O. Novac**, M. Vlăduţiu, Şt. Vari Kakas, I. F. Hathazi, Mihaela Novac, - *Aspects Regarding the use of SEC-DED Codes to the Cache Level of a Memory*

Hierarchy, AIKED-Knowledge Engineering, ISI Books, Cambridge (UK), (Lucrare acceptată spre publicare) , 20-22 February, 2008.

[OmRM03] M. Omaina, D. Rossi and C. Metra, *"High Speed and Highly Testable Parallel Two-Rail Code Checker"*, *Proceedings IEEE International Test Conference*, 2003.

[Poll90] P. L. Howard, "*The Design Book: Techniques and Solutions for Digital Computer Systems"*, *Prentice-Hall Inc.,* Englewood Cliffs, N. J. 1990.

[PIVN04] O. Poszet, I. Ignat, Şt. Vari Kakas, **O. Novac**, - *Methods for cyclic codes decodification,* International Conference on Renewable Sources and Environmental ElectroTehnologies RSEE 2004, România, Universitatea din Oradea, pp. 61 - 65, ISSN 1223 – 2106, 2004.

[PVND05] O. Poszet, Şt. Vari Kakas**, O. Novac**, H. Drăgan, I. Ignat, - *Efficiency Of Identification Protocols in Electronic Payment System,* International Conference on Engineering of Modern Electric Systems, EMES'05, University of Oradea, România, pg. 118 - 121, ISSN 1223 – 2106, 2005.

[PVNI06] O. Poszet, Şt. Vari Kakas, **O. Novac,** I. Ignat, - *Fault Tolerant Protocols Used in Electronic Payment System,* International Conference on Renewable Sources and Environmental ElectroTehnologies, RSEE 2006, România, Universitatea Oradea, pp 39 - 42, ISSN 1841 – 7221, 2006.

[PoVN07] O. Poszet, Şt Vari Kakas, **O. Novac**, - *Performance assessment of an Electronic Payment system by simulation,* International Conference on Engineering of Modern Electric Systems, EMES'07, University of Oradea, România, pg. 100 – 106, ISSN 1223-2106, 2007.

[PoNV07] O. Poszet **, O. Novac**, Şt Vari Kakas, - *Analysis of an electronic Payment System Using Simulation*, Proceedings of the 8th International Symposium of Hungarian Researchers on Computational Intelligence and Informatics, CINTI 2007, Budapest, Hungary, ISBN 978-963-7154-65-2,15-17 November, 2007.

[RaAJ00] P. Ranganathan, S. Adve, N. P. Jouppi, *"Reconfigurable Caches and their Application to Media Processing"*, *Proceedings International Symposium. Computer Architecture*, pp. 214-224, 2000.

[RaFu89] T.R.N. Rao, E. Fujiwara, *"Error-Control Coding for Computer Systems"*, *Prentice-Hall Inc., Englewood Cliffs*, N. J. 1989.

[Sezn93] A. Seznec, *"A Case for Two-Way Skewed-Associative Caches", Proceedings of International Symposium on Computer Architecture,* pp. 169 -178,1993.

[ShMc99] P. Shirvani, E. J. McCuskey, *"PADded Cache: A New Fault-Tolerance Technique for Cache Memories"*, *Proceedings of 17th IEEE VLSI Test Symposium*, pp. 440 - 445, 1999.

[Smit82] A. J. Smith, " *Cache memories. Computing Survey",* Vol. 14, No. 4, pp. 473 - 530, 1982.

[SPEC92] Standard Performance Evaluation Corporation. SPEC'92 benchmarks,*http://www.spec.org/osg/news/articles/news96q1/access.html*.

[SPEC00] Standard Performance Evaluation Corporation. SPEC CPU 2000 benchmarks, *http://www.specbench.org/osg/cpu2000*.

[Swaz87] P. Swazey ,"*SRAM Organization, Control, and Speed, and Their Effect on Cache Memory Design"*, *Midcon/87*, pp. 434 - 437, 1987.

[TeNF01] M. H. Tehranipour, Z. Navabi, S.M. Fakhraie, *"An Efficient BIST Method For Testing of Embedded SRAMs"*, In *Proceedings IEEE International Symp. Circuits and Sistems*, vol. 5, pp. 73 - 76, 2001.

[VaNo03] Şt. Vari Kakas, **O**. **Novac**, *"Cache modeling and performance comparison by simulations"*, *7' th International Conference on Engineering of Modern Electric Systems*, EMES'03, University of Oradea, pp. 199 - 203, 29-31 Mai, 2003.

[VNPN05] Şt. Vari Kakas, **O**. **Novac**, O. Poszet, Mihaela Novac, *„Reliability Considerations in Memory Hierarchies", International Conference on Engineering of Modern Electric Systems*, EMES'05, University of Oradea, pp. 149 - 152, 2005.

[VSMZ00] M. A. Vega-Rodríguez, J. M. Sánchez-Pérez, R. M. de la Montaña, F. A. Zarallo-Gallardo, *"Simulation of Cache Memory Systems on Symmetric Multiprocessors with Educational Purposes", Proceedings of the I International Congress in Quality and in Technical Education Innovation*, vol. III, pp. 47 - 59. Donostia-San Sebastián, Spain. 4-6 September 2000.

[VeSG01] M. A. Vega-Rodríguez, J. M. Sánchez-Pérez, J. A. Gómez-Pulido, *"An Educational Tool for Testing Caches on Symmetric Multiprocessors"*. *Microprocessors and Microsystems, Elsevier Science,* vol. 25, no. 4, pp. 187 – 194, June 2001.

[WuMc95] Wm. A. Wulf and S. A. McKee, *"Hitting the Memory Wall: Implications of the Obvious,"* vol. 23, no. 1, pp. 20 – 24, 1995.

[ZaMS04] H. R. Zarandi, S. G. Miremadi, H. Sarbazi-Azad, *"Fault Detection Enhancement in Cache Memories Using a High Performance Placement Algorithm"*, *Proceedings of the 10th IEEE International On-Line Testing Symposium* (IOLTS'04), 2004.

[ZaMi04] H. R. Zarandi, S. G. Miremadi, *"A Highly Fault Detectable Cache Architecture for Dependable Computing"*, M. Heisel et al. (Eds.), *SAFECOMP 2004*, LNCS 3219, pp. 45 – 59, 2004.

[ZGKS03] W. Zhang, S. Gurumurthi, M. Kandemir, A. Sivasubramaniam, *"ICR: In-Cache Replication for Enhancing Data Cache Reliability"*, *In Proceedings of the International Conference on Dependable Systems and Networks (DSN),* pp. 291 - 300, 2003.

[ZhVN03] C. Zhang, F. Vahid, W. Najjar, *"A Highly Configurable Cache Architecture for Embedded Systems"*, *International Symposium on Computer Architecture,* pp. 136 - 146, 2003.

[ZhZY97] C. Zhang, X. Zhang, Y. Yan, *"Two Fast and High-Associativity Cache Schemes"*, *IEEE micro*, pp. 40 - 49, 1997.

ANEXA A

DIAGRAME DE TIMP

Diagrama ciclului miss la citire este prezentată în fig. A.1.

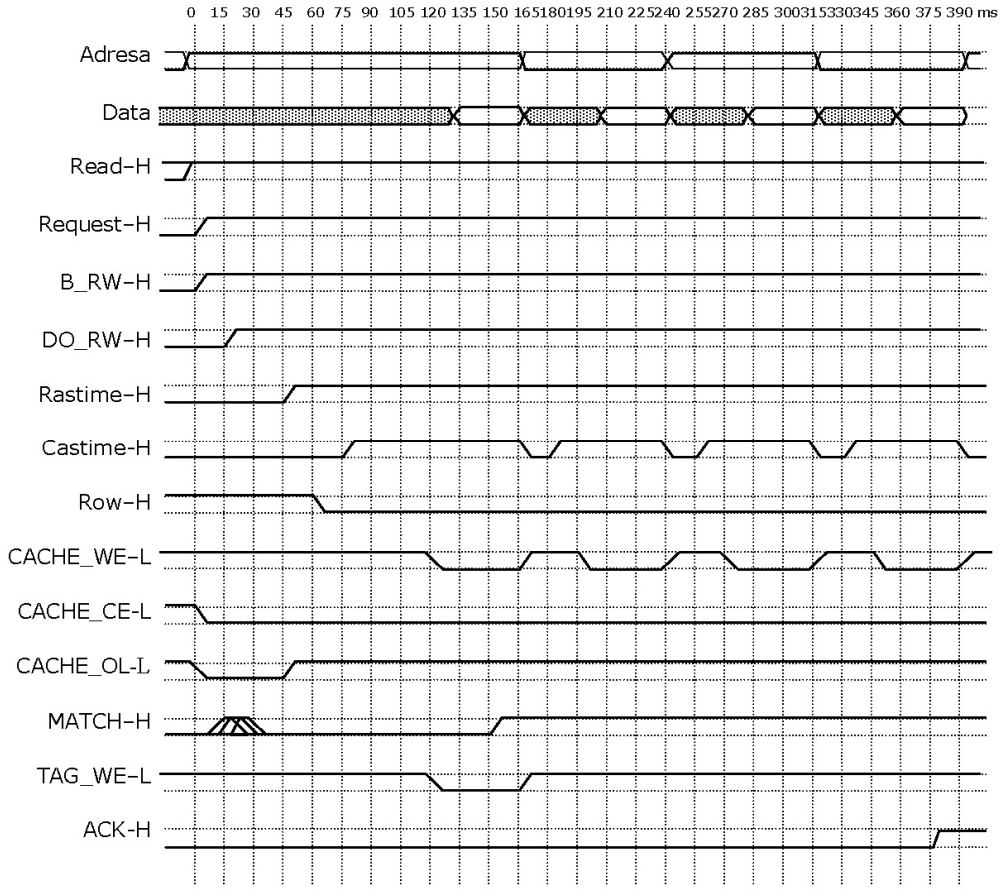


Fig. A.1

Diagrama ciclului hit la scriere este prezentată în fig. A.2.

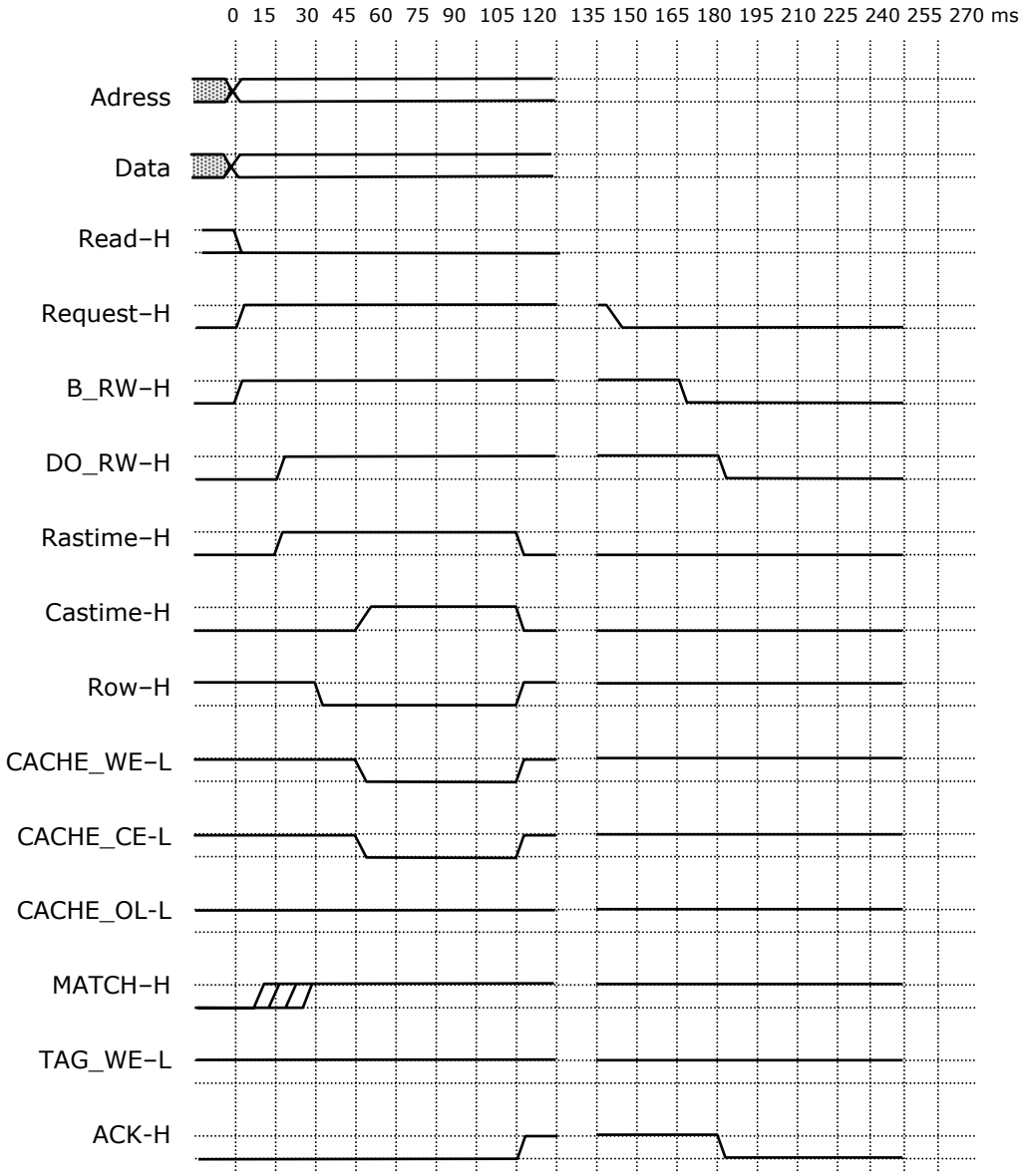


Fig. A.2.

Diagrama ciclului miss la scriere este prezentată în fig. A.3

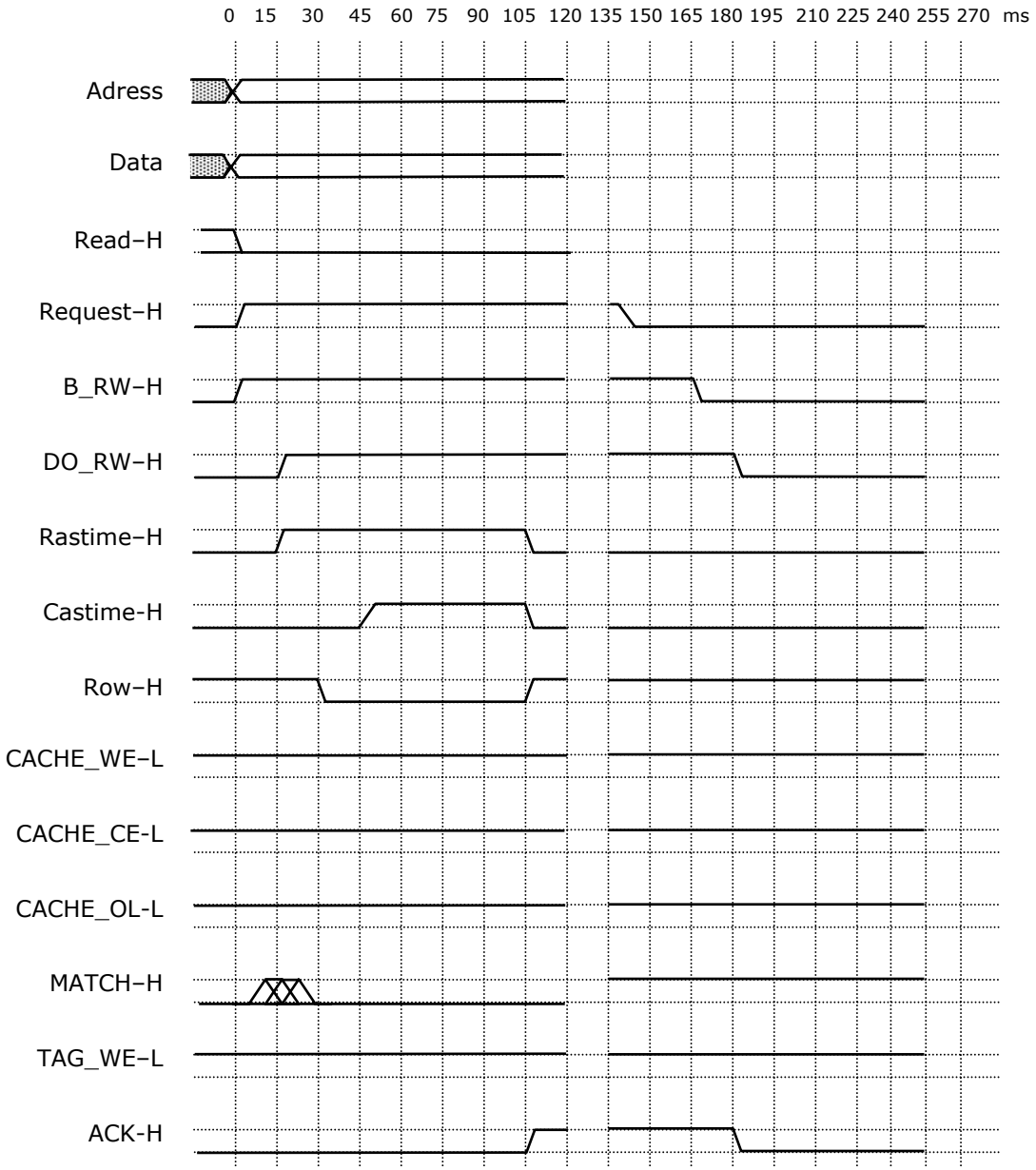


Fig. A.3.

În ciclul din fig. A.1 procesorul (sau alt dispozitiv de pe magistrala de date) solicită o informaţie care nu se găseşte în cache. Ordinograma ciclului miss la citire este prezentată în fig. A.4.

Acest ciclu, (miss la citire) este identic cu ciclul anterior, astfel la momentul t = 0 sec adresele sunt stabile pe magistrala de adrese, linia Read-H activă, ne indică că avem un ciclu de citire şi semnalul Request-H este de asemenea activat.

Şi în acest caz activarea semnalului Request-H va determina activarea semnalului buffer-at de citire/scriere B_RW-H, iar semnalul DO_RW-H, de iniţiere a unui ciclu de citire / scriere, va fi activat după 15 nsec de la

activarea lui B_RW-H numai dacă nu se execută un ciclu de reîmprospătare. Prin activarea semnalelor Read-H şi Request-H, cache-ul este informat că s-a solicitat o operaţie de citire. El va activa semnalele CACHE_CE-L şi CACHE_OE-L pentru a putea plasa pe magistrala de date informaţia, dacă ea este prezentă în cache. Memoria TAG RAM este verificată şi semnalul MATCH-H ne va indica, la începutul celui de al treilea ciclu, adică la 45 nsec de la începutul tranzacţiei, că avem un insucces (miss), deoarece linia stocată la locaţia identificată prin liniile ADR (16:4) nu este identică cu adresa curentă. Deoarece informaţia solicitată nu se găseşte în cache, ea trebuie adusă din memoria DRAM. Urmează aşa cum se observă şi în ordinograma din fig. A.4. dezactivarea semnalului CACHE_OE-L şi iniţializarea unui ciclu de reîmprospătare. Iniţializarea ciclului de reîmprospătare se realizează prin activarea semnalului Rastime-H.

După o întârziere de un ciclu de tact (15 nsec) se dezactivează semnalul Row-H şi apoi după încă un ciclu tact se activează semnalul Castime-H. După un interval de trei cicluri de tact se realizează activarea liniilor CACHE_WE-L şi TAG_WE-L. După încă un interval de tact datele vor fi stabile pe magistrala de date. Activarea semnalului MATCH-H se realizează după un alt interval de tact (t = 150 nsec). În ciclul următor de tact la t = 165 nsec un cuvânt de 32 de biţi (un sfert din linia de cache) se citeşte din memoria DRAM şi se scrie în cache.

În acest moment se dezactivează atât semnalul Castime-H cât şi semnalele CACHE_WE-L şi TAG_WE-L. Se aşteaptă 15 nsec şi în acest moment pe magistrala de adrese, care duce la cache adresa, se modifică şi avem o altă adresă stabilă. În acelaşi moment se activează semnalul Castime-H şi după 15 nsec se activează semnalul CACHE_WE-L. După trei intervale de tact (45 nsec) se citeşte un alt cuvânt de 32 de biţi din memoria DRAM şi se scrie în cache. În acest moment avem o jumătate din linia de cache adusă din DRAM în cache. Se repetă, aşa cum se poate observa şi din ordinograma din fig. A.1. paşii, de la dezactivarea semnalului Castime-H, până la citirea unui cuvânt de 32 biţi din memoria DRAM în cache. În momentul când vom avea adusă o linie întreagă (patru cuvinte de 32 de biţi) în

cache, ciclul miss la citire, este complet şi se poate trece la următorul ciclu, care poate fi un ciclu hit la citire.

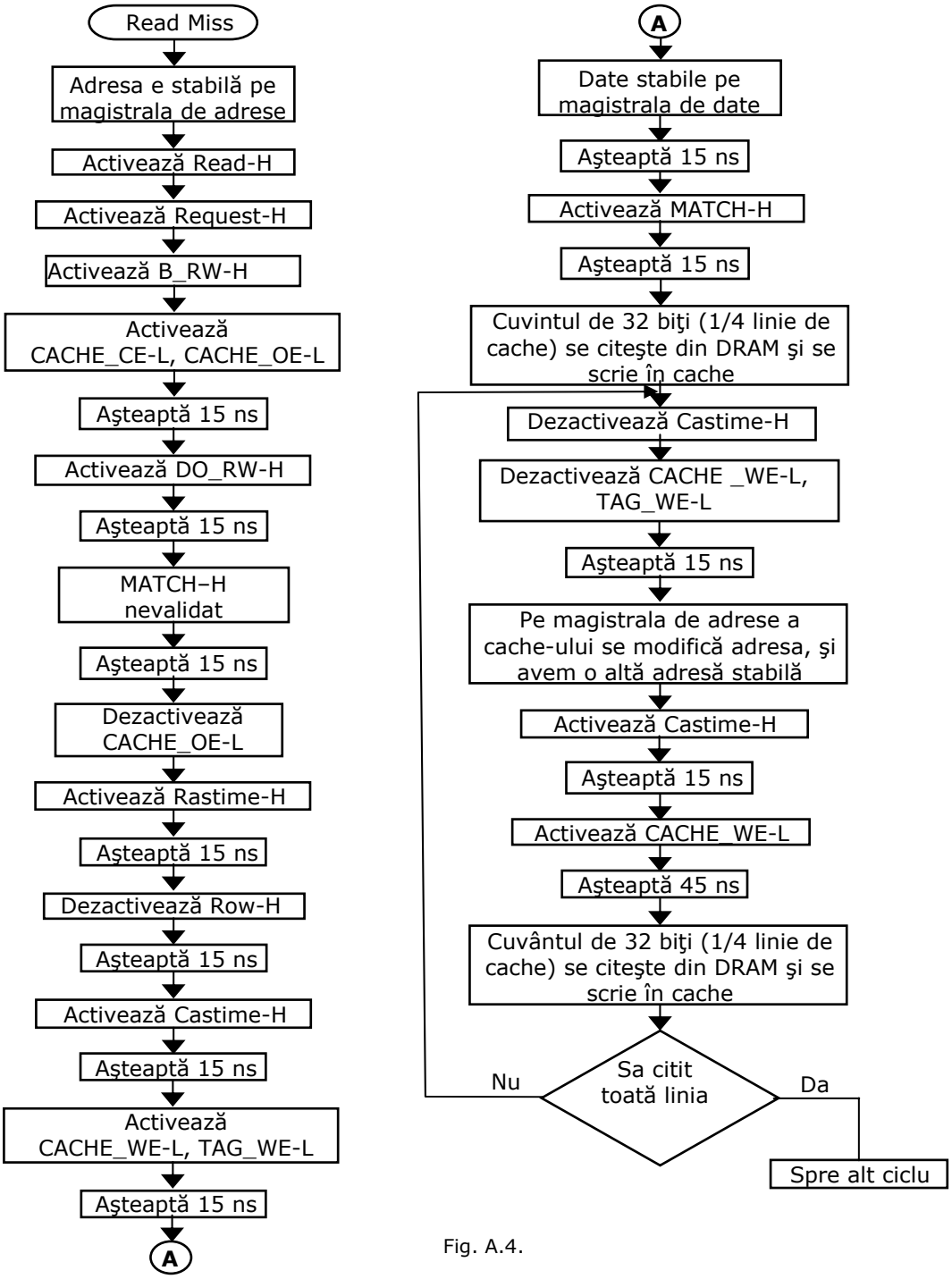


Fig. A.4.

Ordinograma ciclului hit la scriere este prezentată în fig. A.5

În cazul unei scrieri, la momentul t = 0 sec adresele sunt stabile pe magistrala de adrese, linia Read-H este inactivă şi ne indică că avem un ciclu de scriere, iar semnalul Request-H este activat.

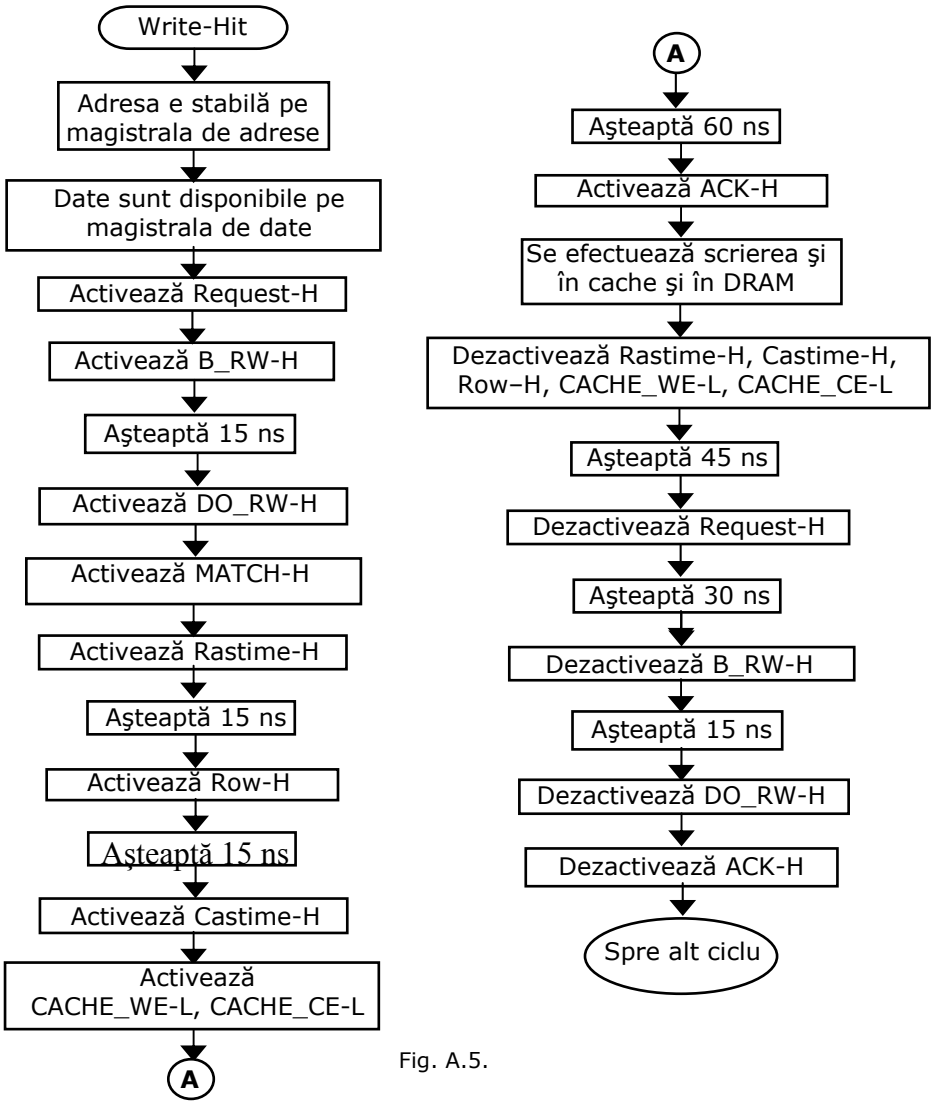


Fig. A.5.

Şi în acest caz activarea semnalului Request-H va determina activarea sermnalului buffer-at de citire/scriere B_RW-H, iar semnalul DO_RW-H, de iniţiere a unui ciclu de citire/scriere, va fi activat după 15 nsec de la activarea lui B_RW-H numai dacă nu se execută un ciclu de reîmprospătare. Deoarece este un ciclu de scriere presupunem că datele sunt stabile şi disponibile de la începutul tranzacţiei.

Se activează semnalul MATCH-H. În acest ciclu de tact se iniţializează un ciclu de reîmprospătare. Se activează semnalul Rastime-H, după un ciclu de tact se activează Row-H şi după încă un ciclu de tact se activează şi semnalul Castime-H.

În acest moment sunt activate şi semnalele CACHE_WE-L şi CACHE_CE-L. După un interval de 60 nsec se activează semnalul ACK-H şi se iniţiază în acest moment un protocol master-slave handshake. Deoarece semnalul MATCH-H este activat încă din al treilea ciclu de tact, acest lucru indicându-ne că locaţia este de asemenea disponibilă şi în cache, vom efectua scrierea atât în cache cât şi în memoria DRAM. Se dezactivează semnalele Rastime-H, Castime-H, Row-H, CACHE_WE-L şi CACHE_CE-L şi după o întârziere de 45 nsec se dezactivează semnalul Request-H. Urmează dezactivarea semnalului B_RW-H după 30 nsec, şi după încă 15 nsec şi dezactivarea semnalului ACK-H, moment în care ciclul hit la scriere, este complet şi se poate trece la următorul ciclu.

Acest ciclu diferă de ciclul anterior, aşa cum se poate vedea şi în ordinograma din fig. A.6. doar prin activitatea liniilor din cache. Deoarece adresa ce dorim să o înscriem nu are un hit în cache, semnalele CACHE_WE-L şi CACHE_CE-L nu sunt activate. De asemenea în ciclul al treilea se testează semnalul MATCH-H şi deoarece nu avem un hit, acest semnal va rămâne inactiv pe toată perioada ciclului.

Ordinograma ciclului miss la scriere este prezentată în fig. A.6

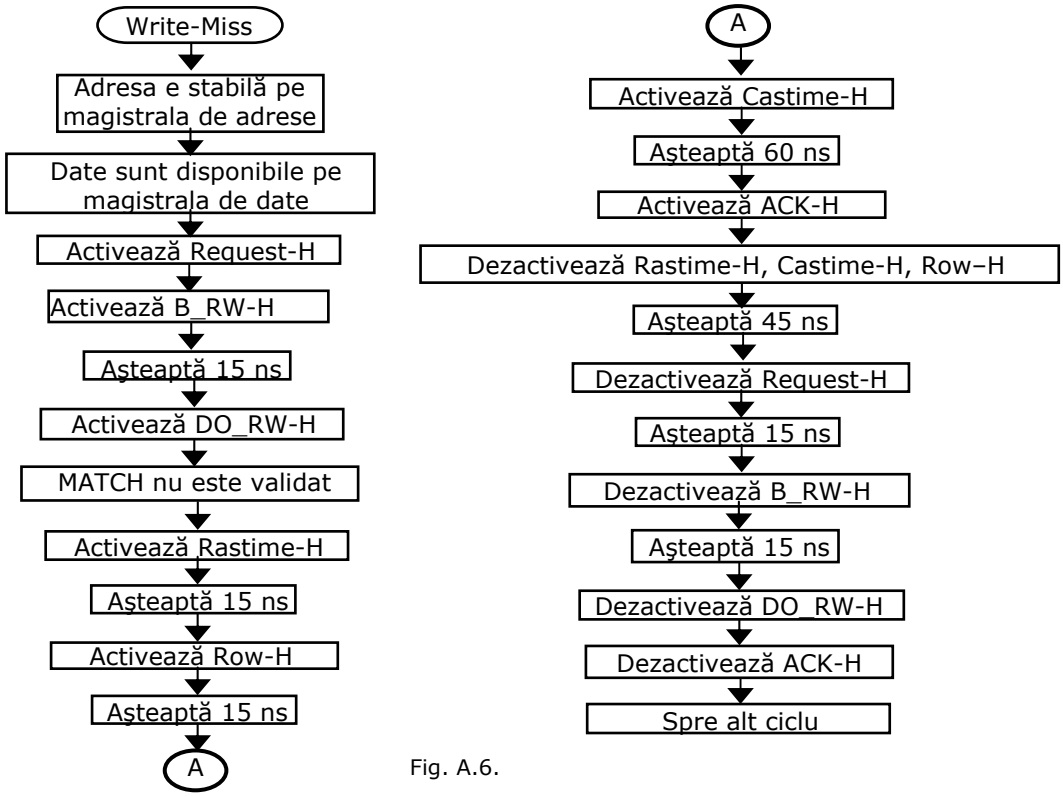


Fig. A.6.

ANEXA B

ORDINOGRAMELE SIMULATOARELOR

Ordinograma funcţiei OnOK() este prezentată în fig. B.1.

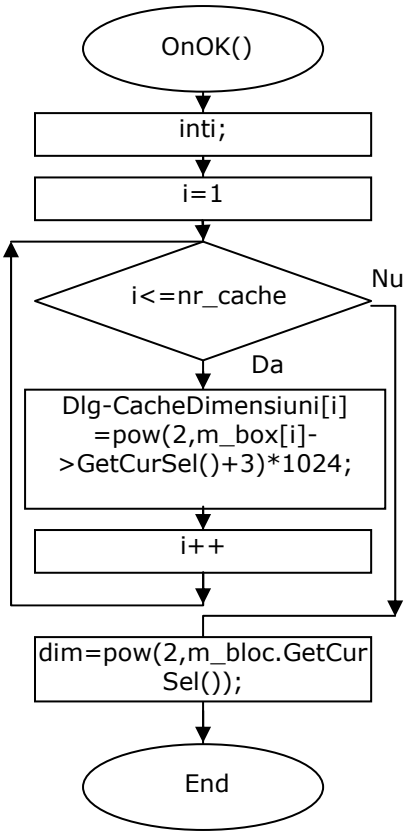


Fig. B.1.

În funcţia OnOK() se iau doar combox-urile vizibile şi se atribuie valorile alese în combox-uri vectorului CacheDimensiuni din dialogul principal CCacheDlg şi de asemenea se atribuie valoarea blocurilor dintr-un set în variabila dim.

Ordinograma funcţiei OnInitDialog() este prezentată în fig. B.2. Această funcţie realizează apelarea clasei extinse CDialog. Se inţializează, de asemenea, legătura dintre controale şi variabilele corespunzătoare. Se setează vizibilitatea celor patru combox-uri ce reprezintă dimensiunile fiecărui nivel de cache şi se selectează pentru fiecare valoarea default.

Funcţia OnSimulare() defineşte dialogul CSimulare. În acestă funcţie se realizează transmitere numărului de niveluri dialogului şi se efectuează şi calculul dimensiunii blocului. Se face selecţia tipului de mapare asociat ierahiei de memorie şi se porneşte al doilea dialog CMemCache. Se atribuie valori mărimilor T_ACC_FETCH, T_ACC_RD, T_ACC_WR şi T_MISS_PENALTY pentru fiecare nivel al ierarhiei. În final se deschide dialogul de simulare.

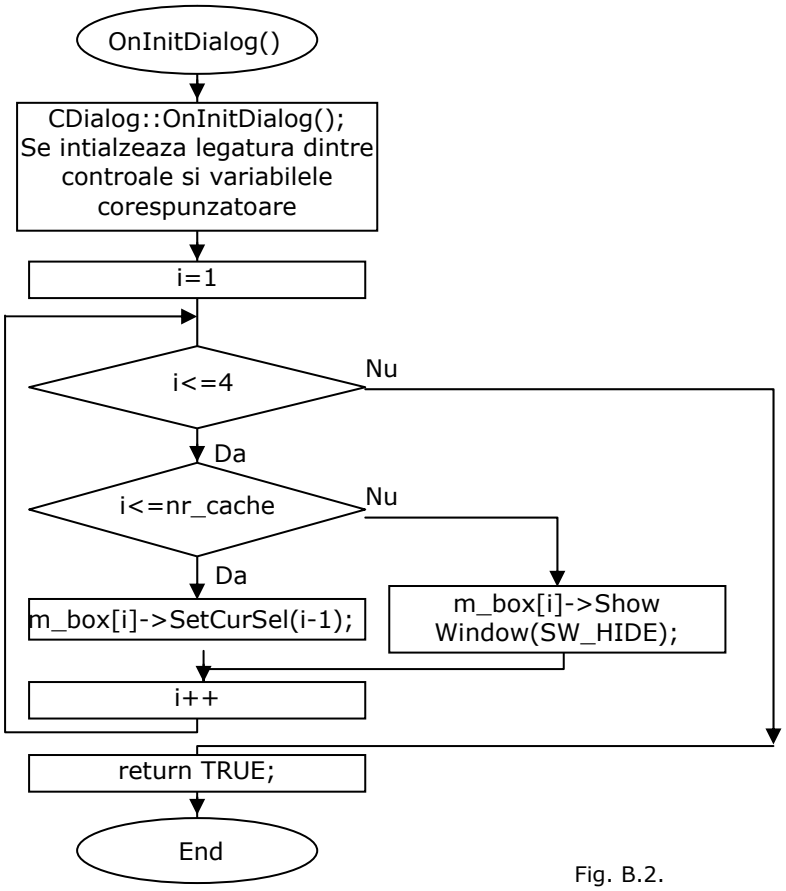


Fig. B.2.

Ordinograma funcţiei OnSimulare() este prezentată în fig. B.3.

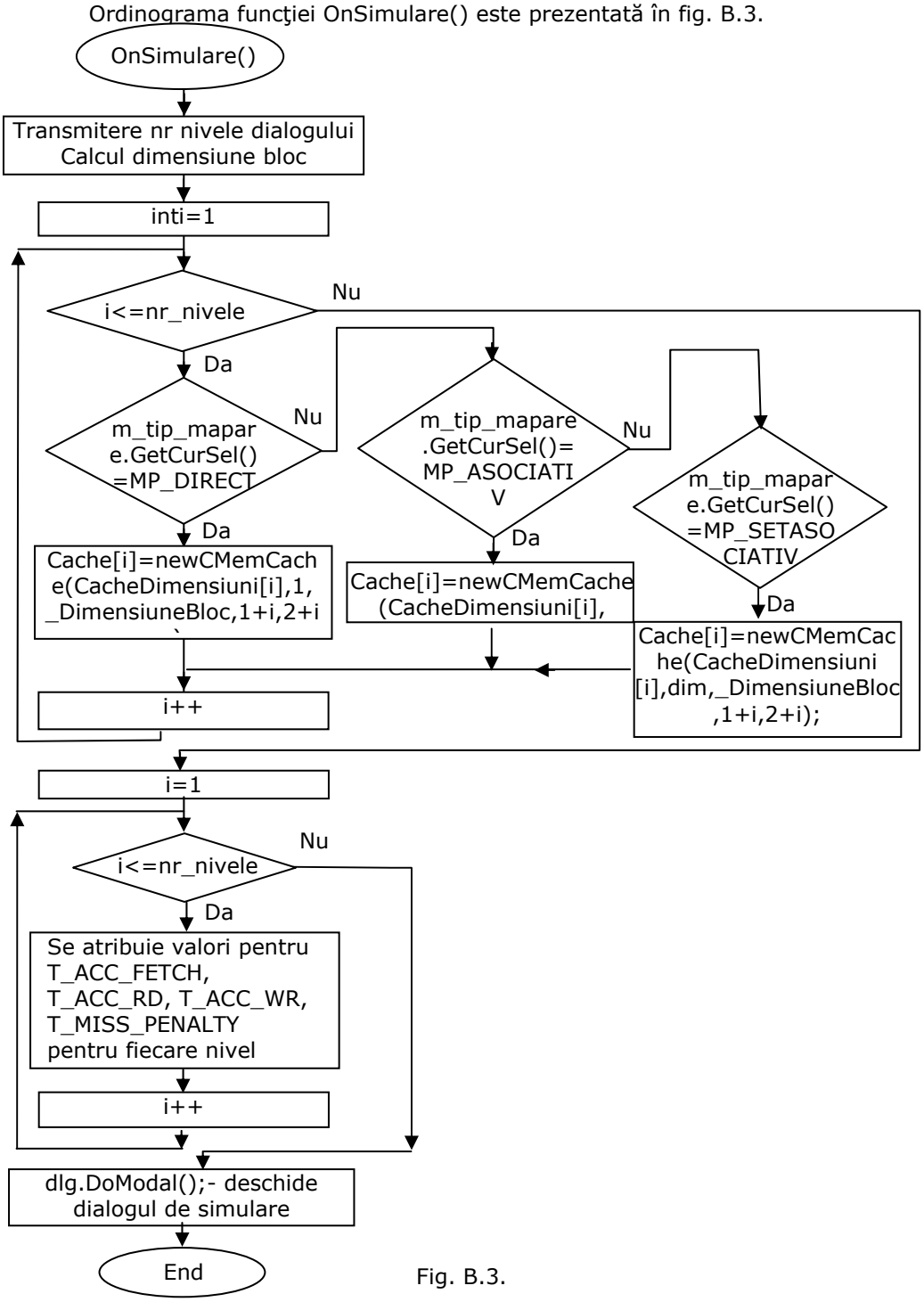


Fig. B.3.

În ordinograma din fig B.4., este prezentat constructorul CMemCache().

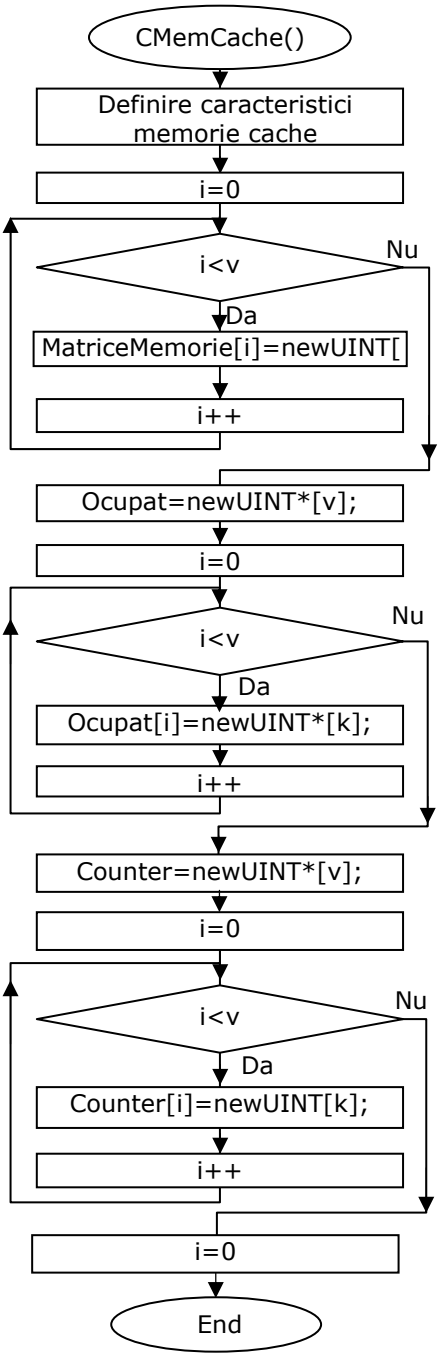


Fig. B.4.

În ordinograma constructorului CMemCache se definesc prima dată caracteristicile cache-ului, adică se definesc: T_ACC_RD, T_ACC_WR, dimensiunea cache-ului, dimensiunea blocului, numărul de blocuri în set, numărul de blocuri.

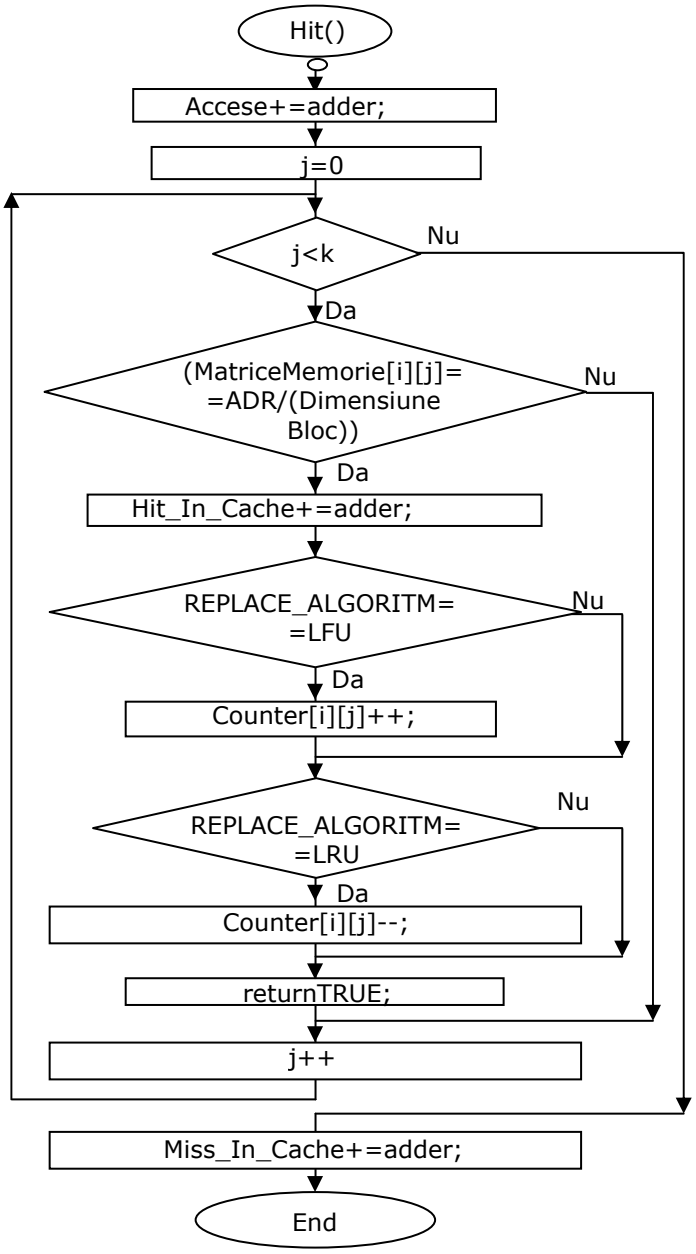


Fig. B.5.

După aceasta se crează un vector pentru stocarea datelor memorate. Vectorul Ocupat indică dacă blocul de memorie este ocupat sau nu, iar vectorul Counter indică numărul de accesări al unui anumit bloc. În continuare se iniţializează vectorii: Ocupat, MatriceMemorie şi Counter cu zero. Tot în acest constructor se iniţializează variabilele: Accese, Miss_In_Cache şi Hit_In_Cache cu zero.

Ordinograma funcţiei Hit() este prezentată în fig B.5. În primul rând se contorizează numărul de accese (adder). Se calculează apoi adresa blocului curent, numărul blocului în set şi adresa setului. Se caută în toate blocurile din set adresa blocului, adică dacă avem un hit.

Dacă s-a găsit adresa în cache (Hit), se returnează TRUE. Se contorizează numărul de hit-uri din cache (adder). Se testează dacă avem algoritmul LFU. Dacă da, se incrementează counter-ul blocului pentru care am avut hit. Se testază apoi dacă avem algoritm LRU. Dacă da, se decrementează counter-ul blocului pentru care am avut hit. Dacă nu s-a găsit adresa în cache, adică dacă avem un miss, se returnează FALSE.

Ordinograma funcţiei Add() este prezentată în fig B.6. Se calculează adresa blocului curent, numărul blocului în set şi adresa setului. Se caută în toate blocurile din set primul bloc liber (neocupat). Daca s-a găsit o locaţie liberă pentru un bloc din set, atunci se adaugă adresa în set. Se marchează locaţia ca fiind ocupată. Se incrementează contorul pentru locaţia respectivă şi în final se returnează valoarea 1.

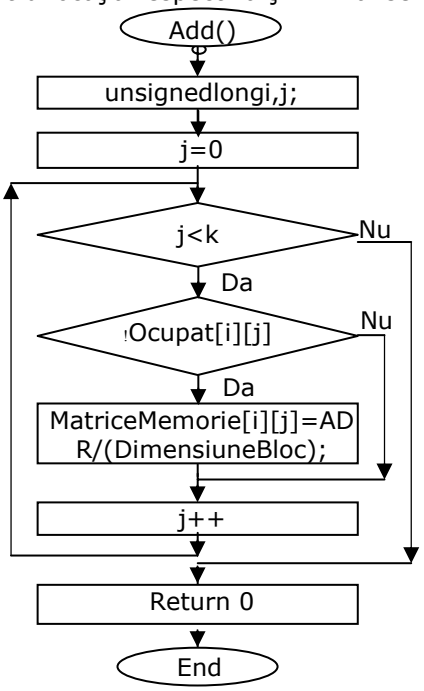
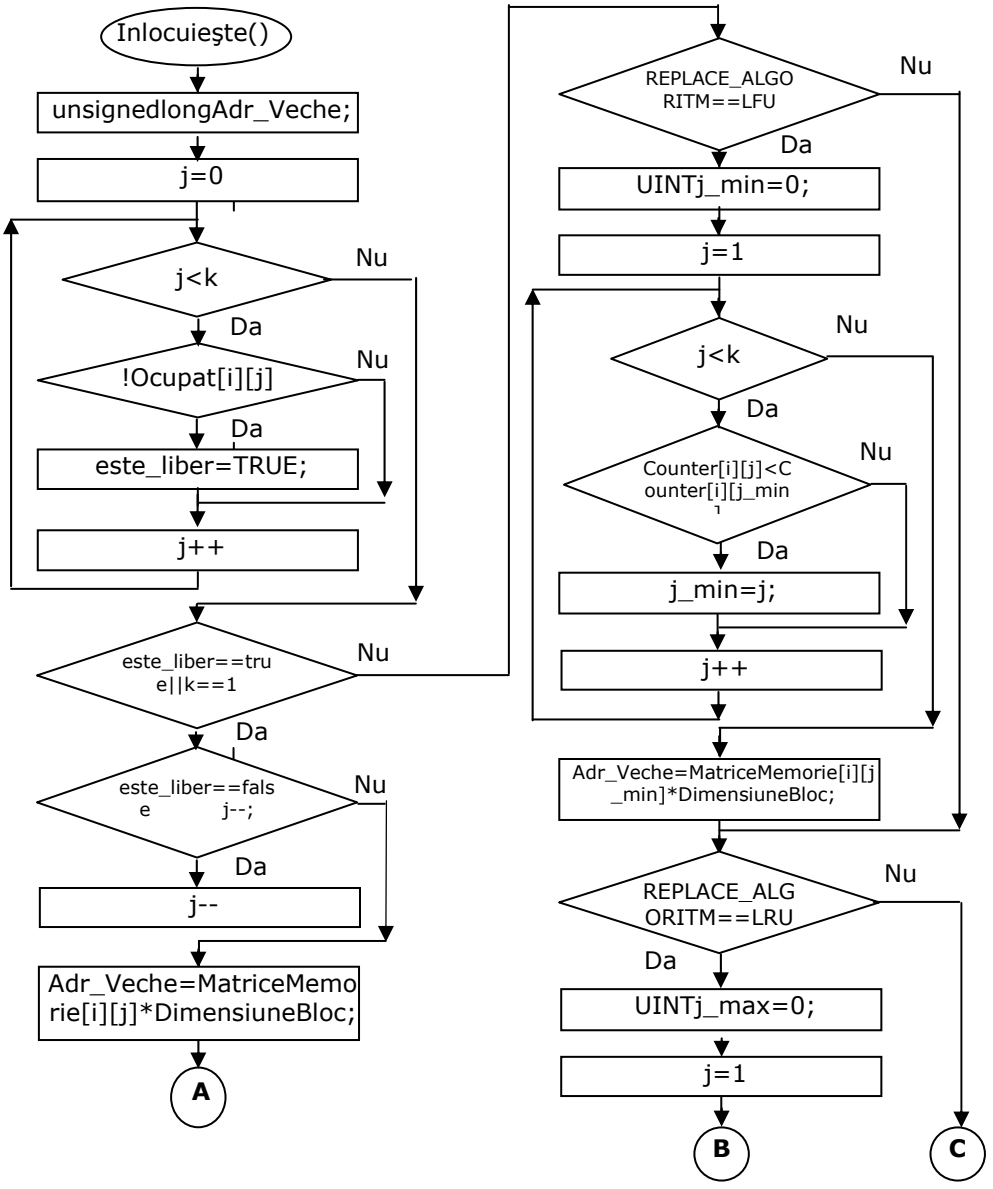


Fig. B.6.

Ordinograma funcţiei înlocuieşte() este prezentată în fig B.7. Se calculează adresa blocului curent, numărul blocului în set şi adresa setului. Se caută în toate blocurile din set primul bloc liber (neocupat). Dacă s-a găsit o locaţie liberă atunci variabila este_liber ia valoarea TRUE. Dacă s-a găsit o locaţie liberă sau avem o mapare direct asociativă, în acest caz se calculează blocul unde se va face înlocuirea. Apoi se aduce din memorie valoarea care se va şterge. Se reţine noua adresă în memorie. Se marchează locaţia ca fiind ocupată. Dacă avem algoritmul FIFO, contorul pentru locaţia respectivă ia valoarea 1.



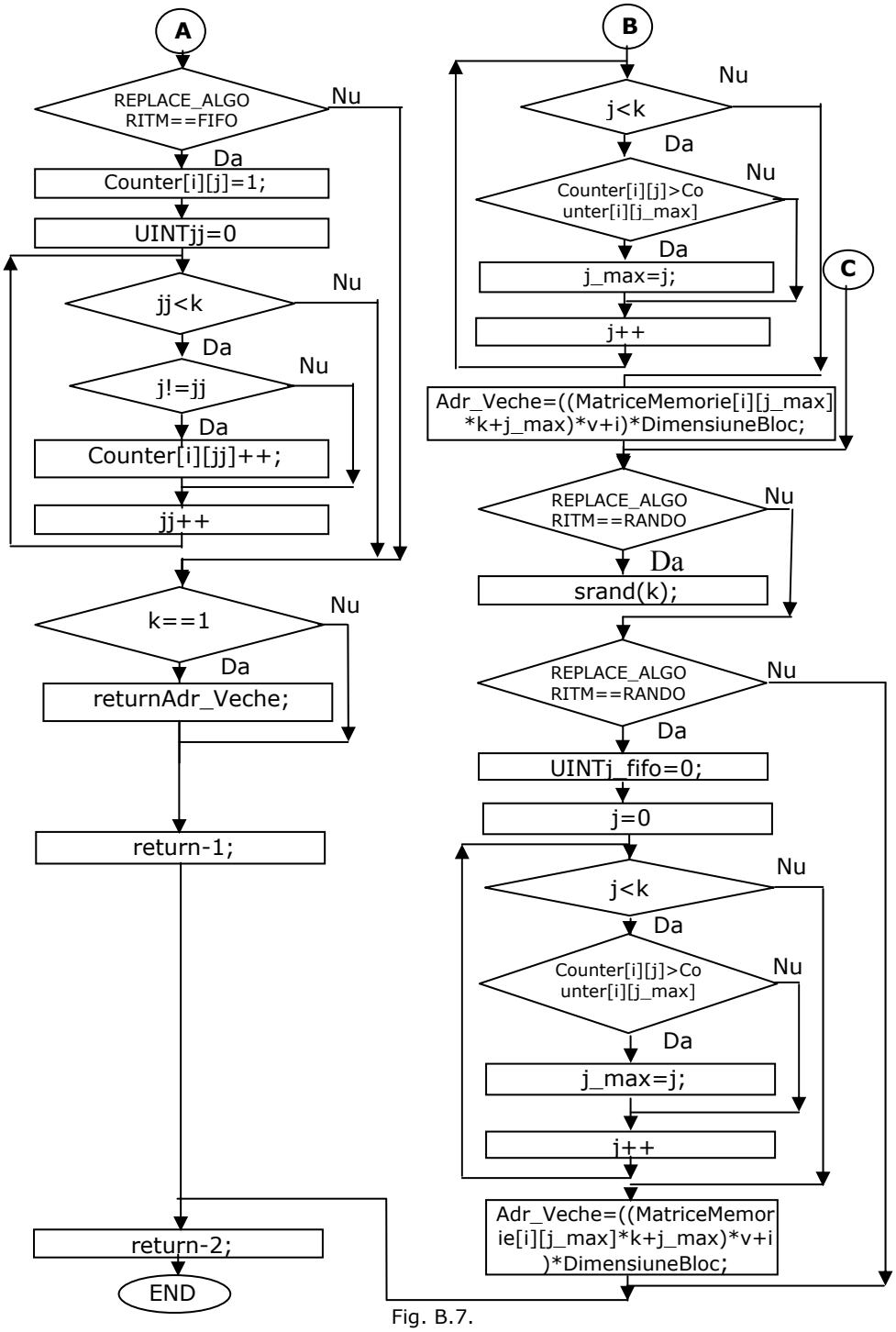

A
REPLACE_ALGO RITM==FIFO
Nu
Da
Counter[i][j]=1;
UINTjj=0
jj<k
Nu
Da
j!=jj
Nu
Da
Counter[i][jj]++;
jj++
k==1
Nu
Da
returnAdr_Veche;
return-1;
return-2;
END
B
j<k
Nu
Da
Counter[i][j]>Co unter[i][j_max]
Nu
Da
j_max=j;
j++
C
Adr_Veche=((MatriceMemorie[i][j_max] *k+j_max)*v+i)*DimensiuneBloc;
REPLACE_ALGO RITM==RANDO
Nu
Da
srand(k);
REPLACE_ALGO RITM==RANDO
Nu
Da
UINTj_fifo=0;
j=0
j<k
Nu
Da
Counter[i][j]>Co unter[i][j_max]
Nu
Da
j_max=j;
j++
Adr_Veche=((MatriceMemor ie[i][j_max]*k+j_max)*v+i)*DimensiuneBloc;

Fig. B.7.

Pentru toate celelalte locaţii din set se incrementează contorul, dacă a avut loc o înlocuire atunci se returnează adresa veche, altfel se retunează -1. Ordinograma funcţiei OnInitDialog(), este prezentată în fig. B.8. Se apelează OnInitDialog din clasa extinsă CDialog. Se face legătura între controale şi variabilele corespunzătoare, pentru nivelurile inexistente se invalidează căsuţele de afişare a graficelor Hit Rate şi Miss Rate şi se afişează textul NU EXISTA.

Se setează în obiectele cache-ului, algoritmul de înlocuire şi politica de scriere în funcţie de selecţiile din dialogul principal. Se creează ferestrele valide de desenare şi se resetează contoarele de accese: Hit_In_Cache şi Miss_In_Cache.

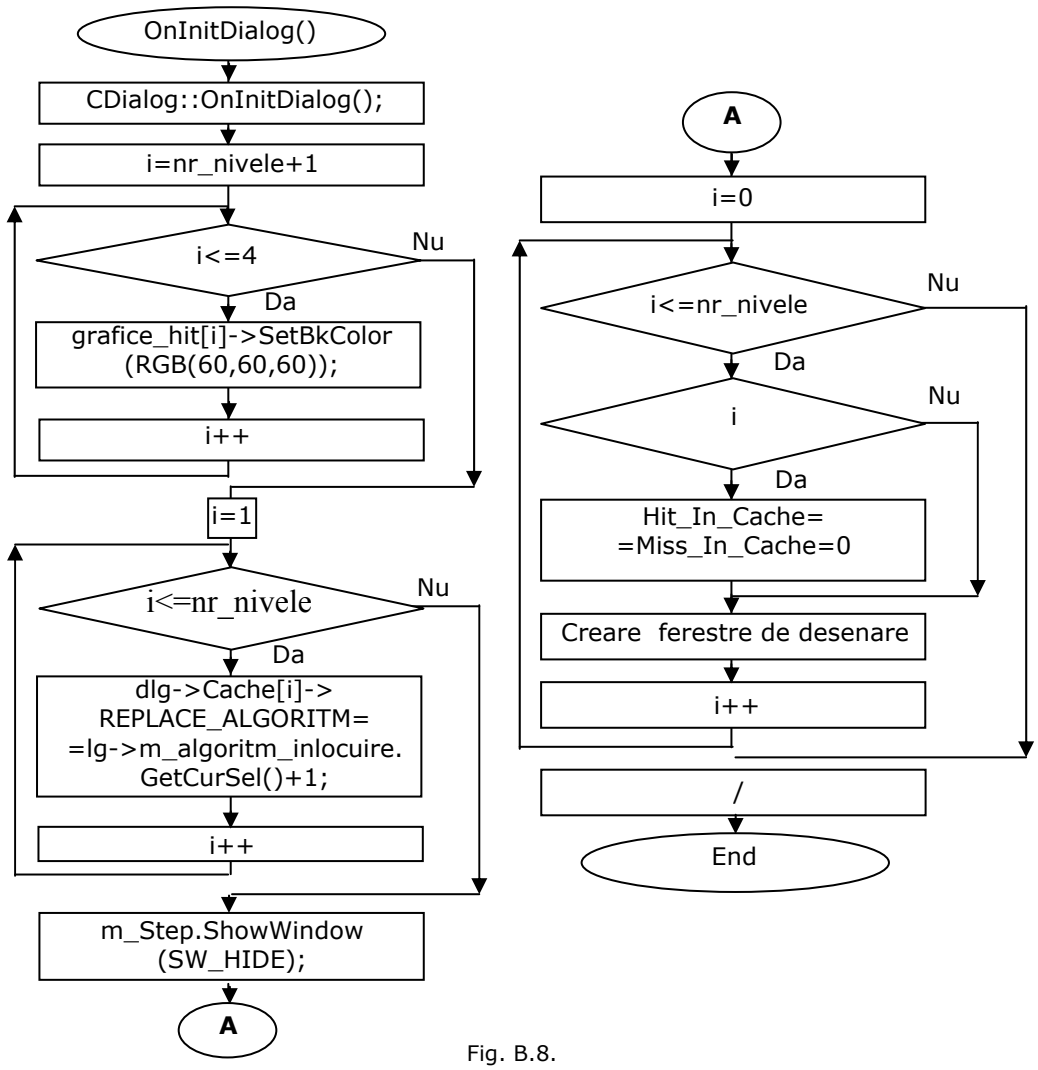


Fig. B.8.

Ordinograma funcţiei MakeDCs(), este prezentată în fig. B.9.

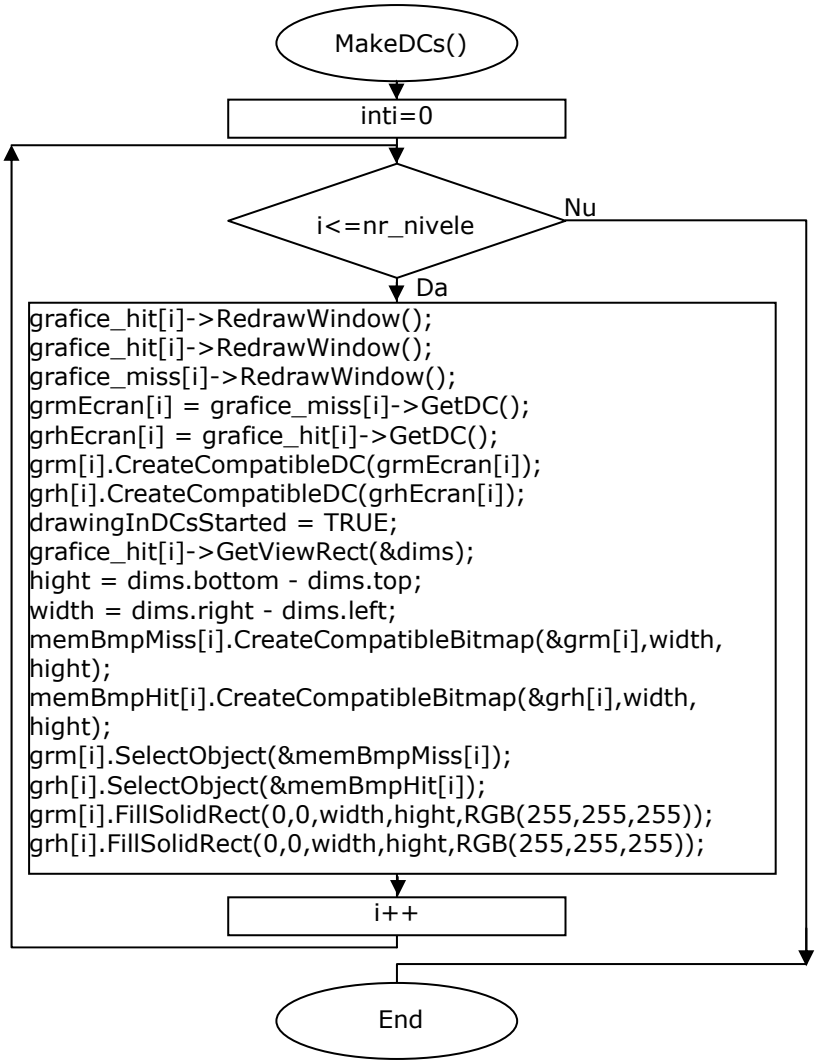


Fig. B.9.

Se eliberează zona de desenare. Se crează o unealtă de desenare. Se trece în modul desenare prin setarea unei variabile pe valoarea TRUE. Se calculează apoi mărimea zonei de desenare. Se colorează cu alb zona de desenare.

Ordinograma funcţiei OnBrowse(), este prezentată în fig. B.10. Se crează dialogul de deschidere a unui fişier. Se deschide fereastra de dialog Open. Se memorează într-o variabiă numele fişierului de tip trace, calea spre acel fişier se memorează într-o altă variabilă şi se afişează de asemenea numele fişierului în

dialogul CSimulare. Se deschide fişierul şi se număra liniile acestui fişier. Dacă nu se deschide fişierul se afişează mesajul ”ERROR”. Se inchide fişierul. Butoanele Run şi Step by Step devin vizibile.

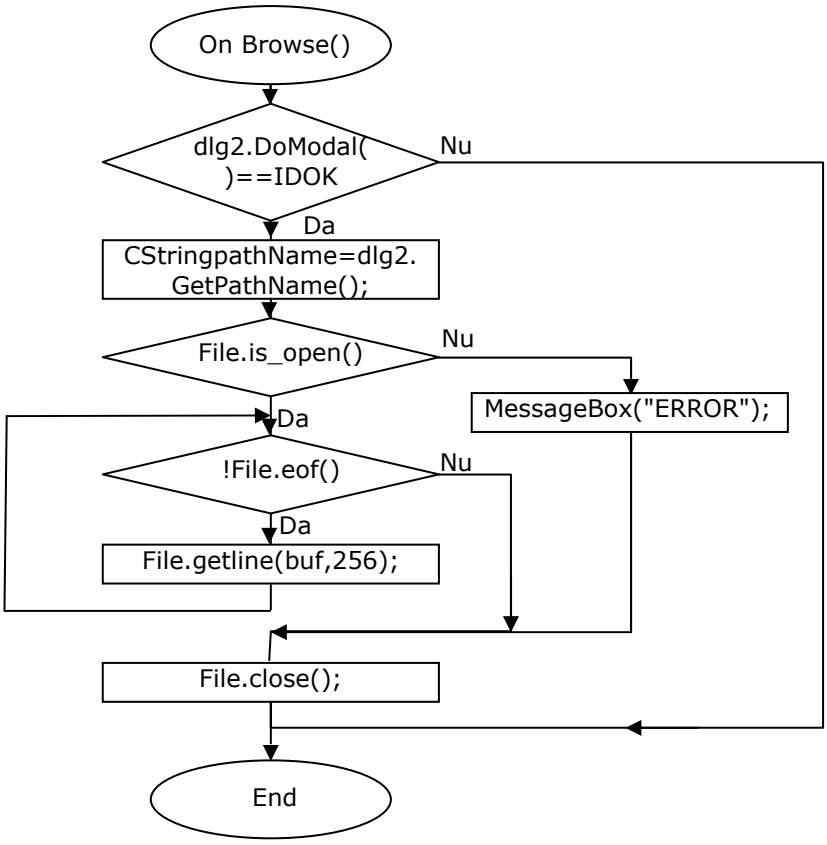


Fig. B.10.

Ordinograma funcţiei OnRun(), este prezentată în fig. B. 11. Se resetează zonele de desenare în funcţie de variabila mustClearDCs. Se resetează variabilele Accese, Hit_In_Cache, Miss_In_Cache pentru toate nivelurile active. Se testează dacă e deschis fişierul de tip trace şi în cazul în care nu e deja deschis se deschide şi se resetează variabilele globale: TIMP_EXECUŢIE, MISS, HITS. Se apelează funcţia OnStepbyStep() până la sfărşitul fişierului. Se inchide fişierul trace. Variabila mustClearDCs ia valoarea TRUE.

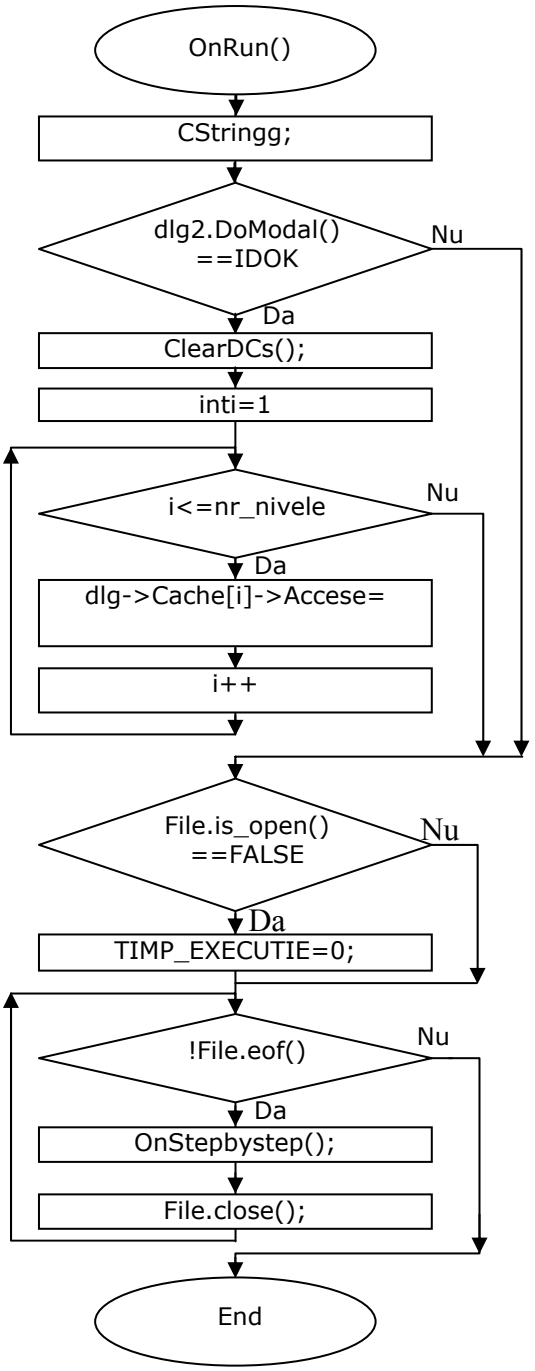
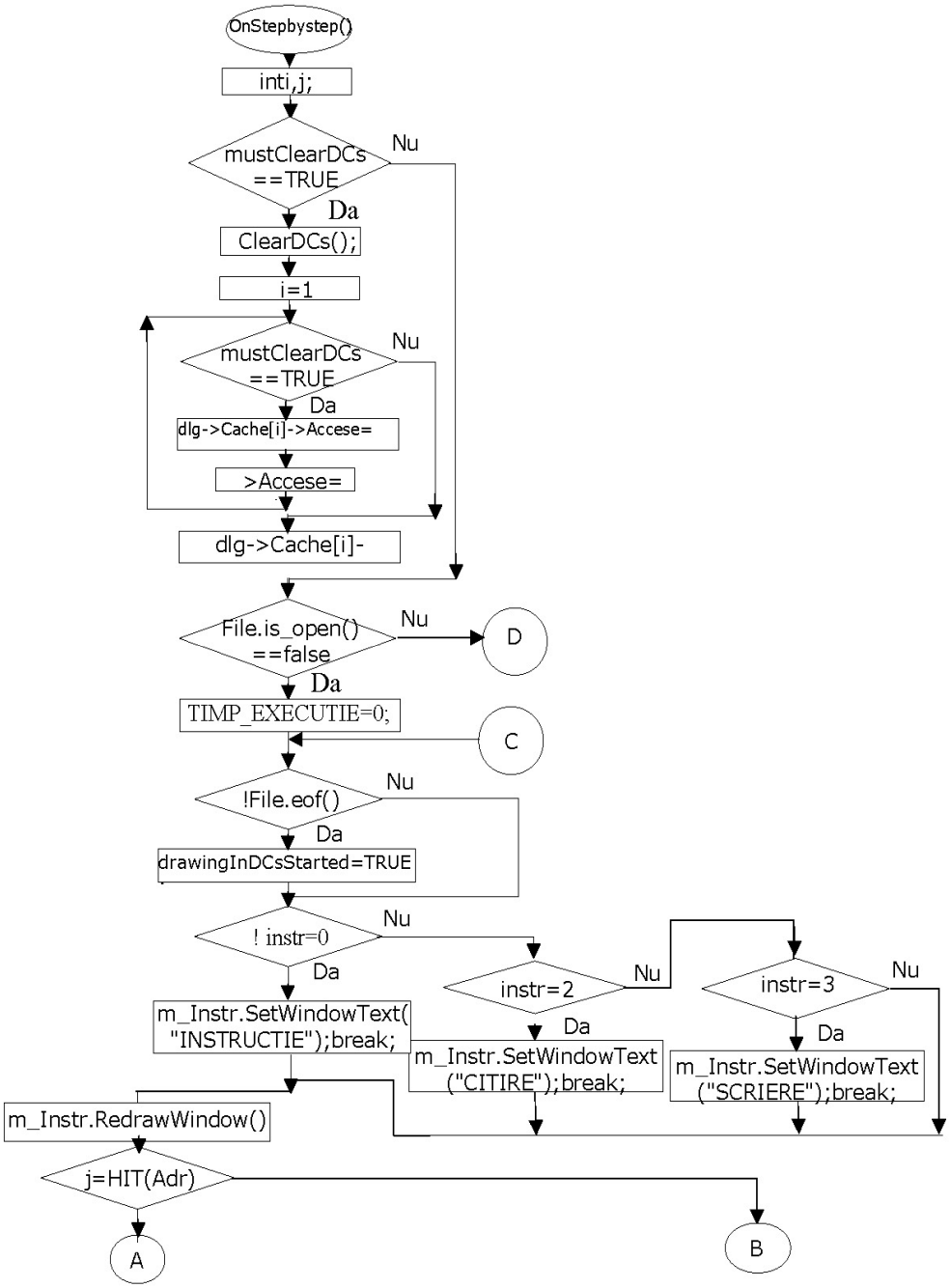

OnRun()
CStringg;
dlg2.DoModal()
==IDOK
Nu
Da
ClearDCs();
inti=1
i<=nr_nivele
Nu
Da
dlg->Cache[i]->Accese=
i++
File.is_open()
==FALSE
Nu
Da
TIMP_EXECUTIE=0;
!File.eof()
Nu
Da
OnStepbystep();
File.close();
End

Fig. B.11

Ordinograma funcţiei OnStepbystep(), este prezentată în fig. B.12.



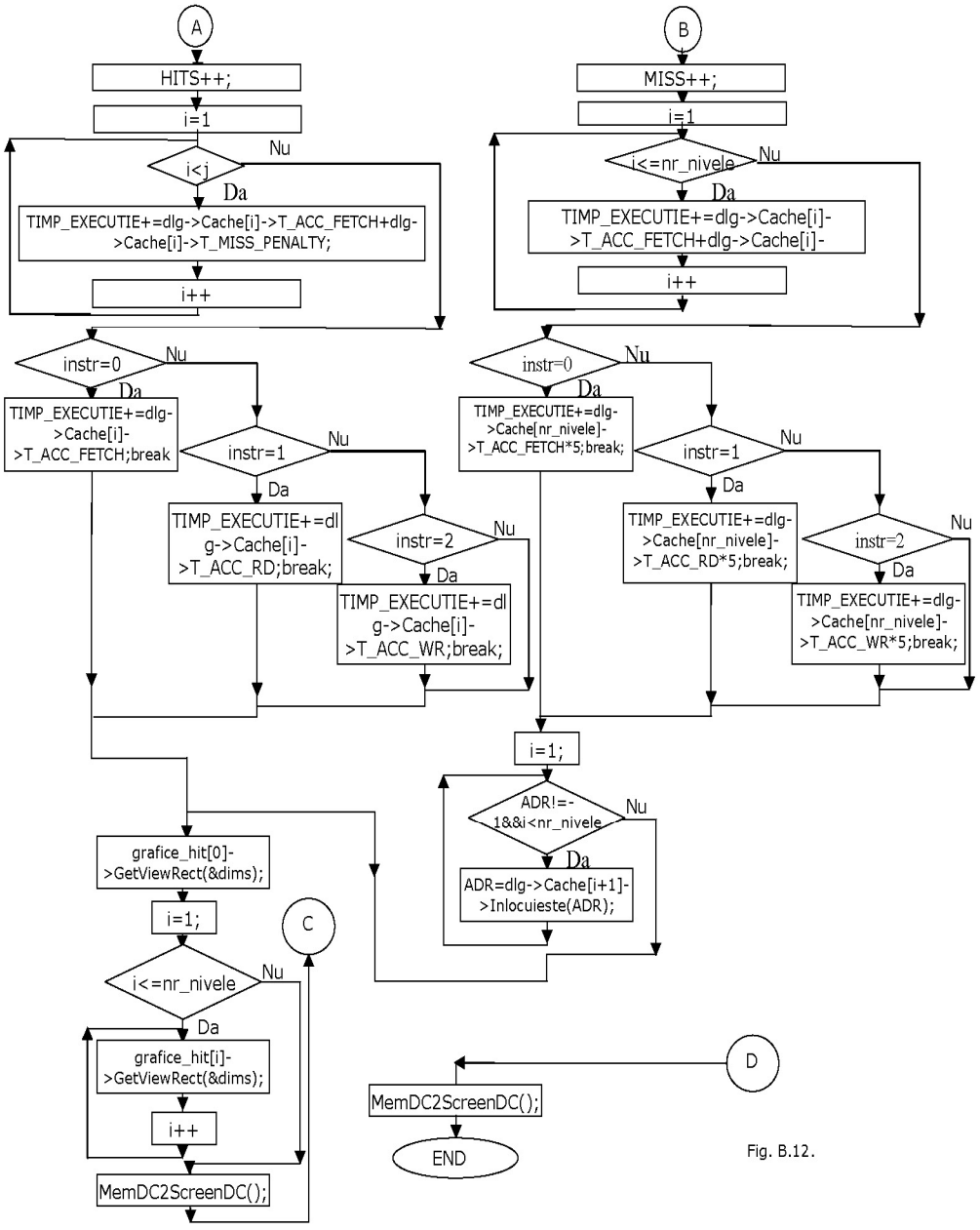


Fig. B.12.

Fig. B.12.

Se resetează zonele de desenare în funcţie de variabila mustClearDCs. Se resetează variabilele Accese, Hit_In_Cache, Miss_In_Cache pentru toate nivelurile active. Se testează dacă e deschis fişierul de tip trace şi în cazul în care nu e deja deschis se deschide şi se resetează variabilele globale: TIMP_EXECUŢIE, MISS, HITS.

Ordinograma funcţiei Hit(), este prezentată în fig. B.13.

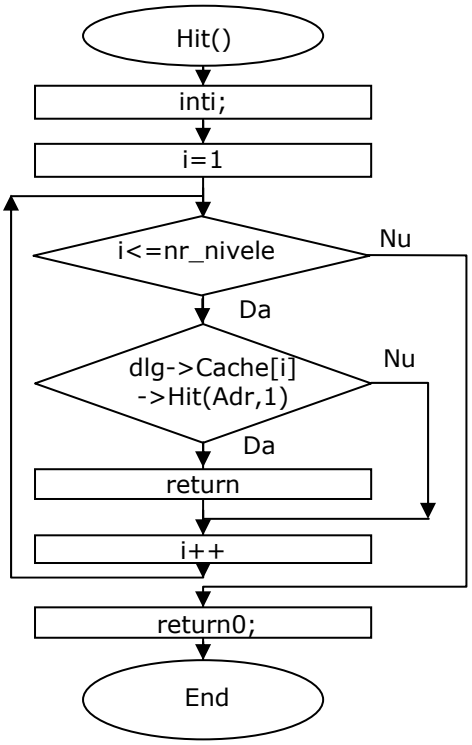


Fig. B.13.

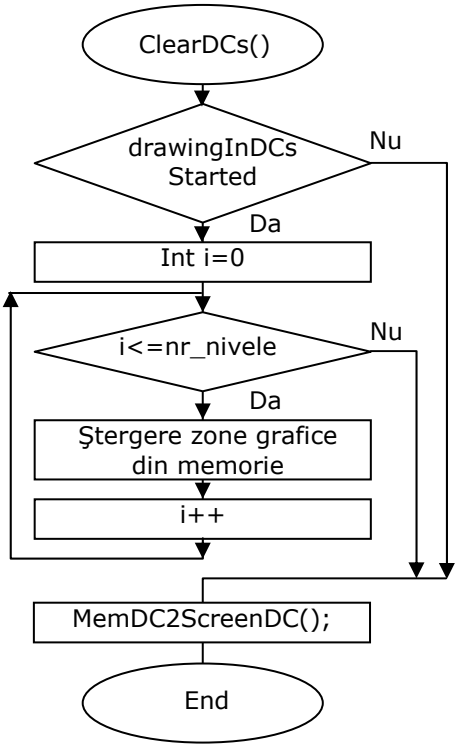


Fig. B.14.

Se verifică nivelul curent de cache i, apoi se apelează funcţia Hit() pentru fiecare nivel i. Dacă se returnează o valoare diferită de zero, atunci se returneză nivelul curent, iar dacă nu avem succes pe nici un nivel, se returnează valoarea zero.

Ordinograma funcţiei ClearDCs(), este prezentată în fig. B.14.

Dacă suntem în mod desenare, adică se testează variabila drawingInDCsStarted, atunci se şterg toate graficele.

Se testează nivelul curent şi se calculează dimensiunile graficelor curente (Hit şi Miss). Se şterg apoi graficele prin acoperirea lor cu un dreptunghi de culoare albă. Se apelează în final, funcţia MemDC2ScreenDC().

Ultima ordinogramă este cea a funcţiei MemDCs2ScreenDCs() şi ea este prezentată în fig. B.15.

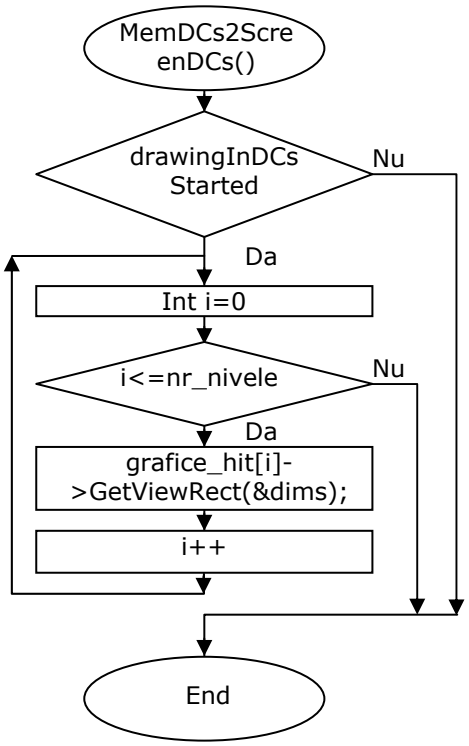


Fig. B.15.

Se testează variabila drawingInDCsStarted. Se testează nivelul curent şi se calculează dimensiunile graficelor curente (Hit şi Miss). Se afişează graficele din memorie pe ecran.

În continuare sunt prezentate ordinogramele ce descriu construcţia mediului de evaluare CDLR SPEC 2000. Mediul este alcătuit din următoarele programe: main.c, read.h, run.h, data.h, write.h.

Ordinograma funcţiei principale main.c, este prezentată în fig. B.16.

Se iniţializează timpul t1, după care se citesc argumentele care se dau în linia de comandă. Se testează valoarea parametrului ARGN, care reprezintă numărul argumentelor din linia de comandă. Dacă ARGN < 1, atunci se afişează mesajul "Nu exisă fişierul de intrare" şi se iese din program, returnându-se valoarea 1. Dacă ARGN > 1, atunci se aplează procedura read_data() (notată cu R în această ardinogramă). Se testează din nou valoarea lui ARGN şi dacă ARGN > 2, atunci se aplează procedura de scriere writedata() (notată cu W în această ordinogramă). În

caz contrar, se afişează fişierul main.out, se iniţializează timpul t2 şi se afişează mesajul "EXECUŢIA A DURAT (t2-t1) sec", după care se iese din programul principal.

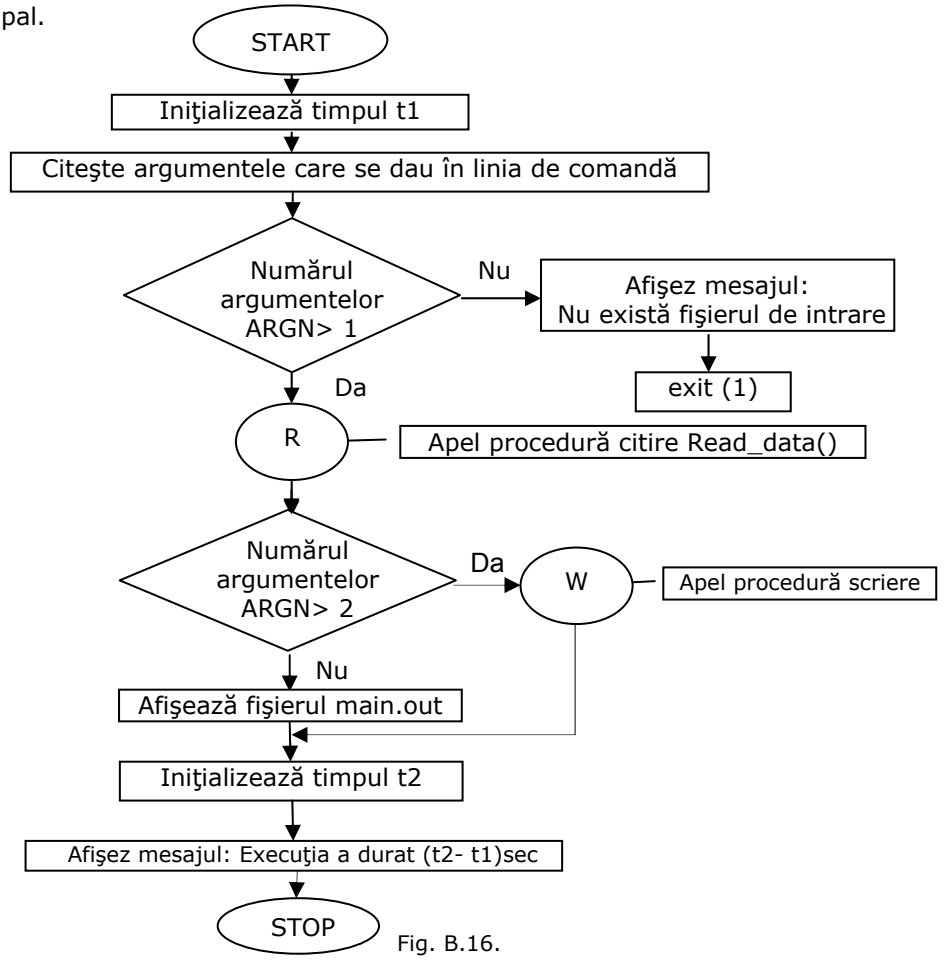
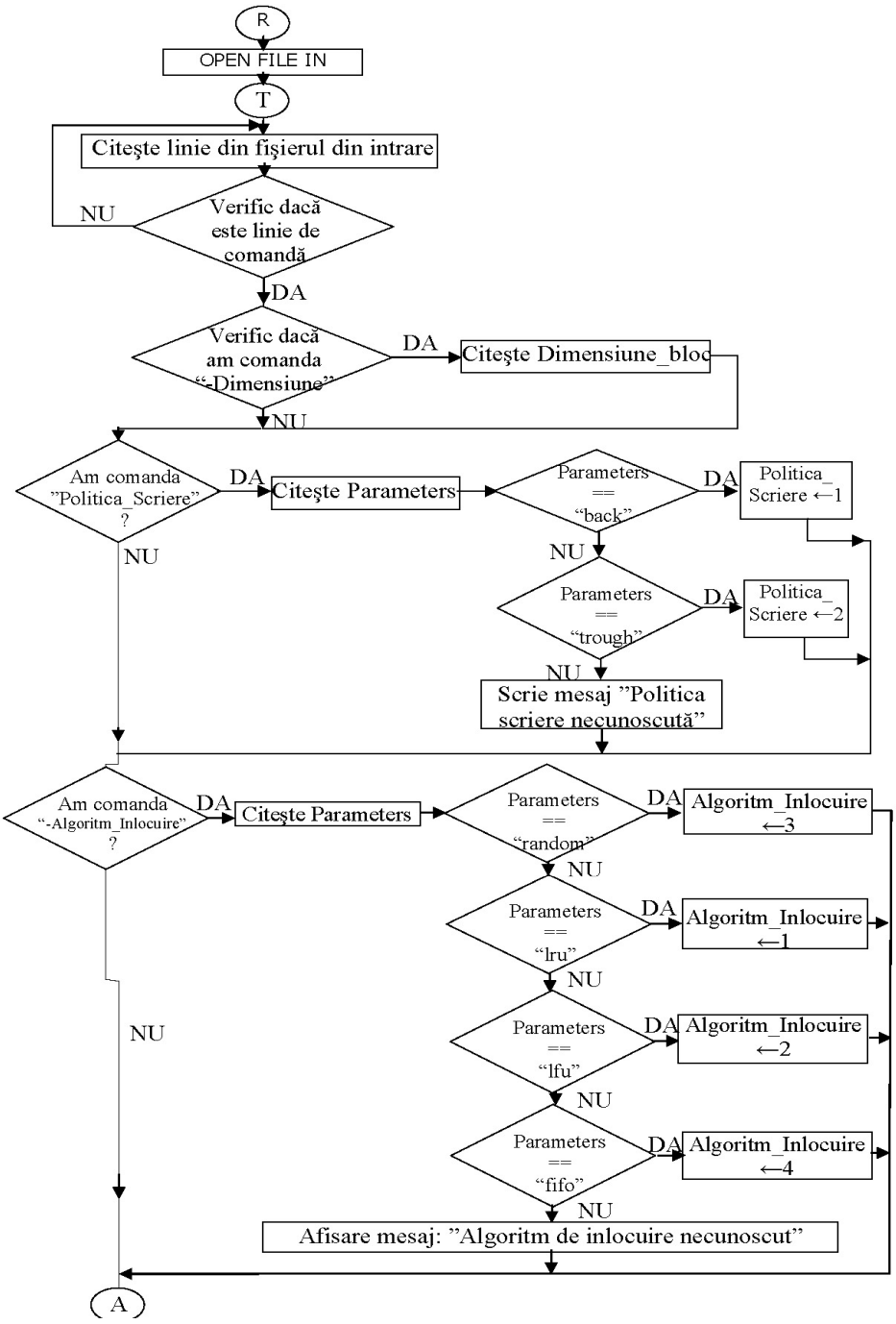
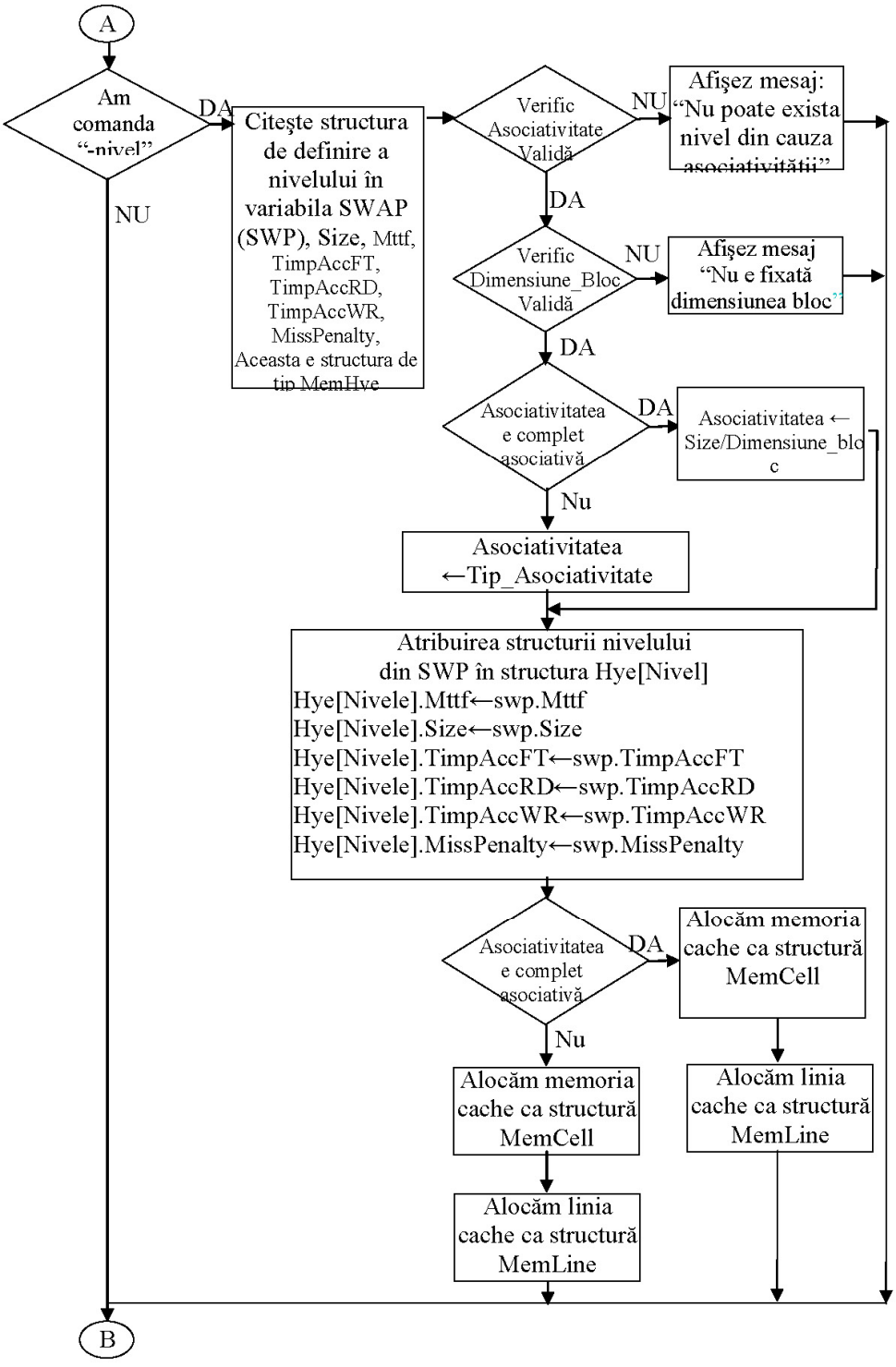


Fig. B.16.

În fig. B.17. este ilustrată ordinograma funcţiei read_data(). Se deschide fişierul de intrare şi se citeşte o linie din acest fişier. Se verifică dacă aceasta este o linie de comandă. Dacă nu avem linie de comandă se citeşte următoarea linie din fişierul de intrare. Se verifică dacă aceasta este linie de comandă şi în caz afirmativ se citeşte parametrul Dimensiune_bloc. În continuare se defineşte ierarhia de memorie. Se testează dacă avem comanda "Politica_Scriere" şi se citeşte variabila Parameters. În funcţie de valoarea citită, se atribuie valoarea 1, pentru write back, valoarea 2 pentru write trough, sau se afişează mesajul "Politica scriere necunoscută" dacă avem alte valori. Se testează în continuare, dacă avem comanda "Algoritm_Inlocuire" şi se citeşte variabila Parameters. În funcţie de valoarea citită, se atribuie valoarea 3, pentru random, valoarea 1 pentru lru, valoarea 2 pentru lfu, valoarea 4 pentru fifo, sau se afişează mesajul "Algoritm de înlocuire necunoscut"

dacă avem alte valori. Se testează dacă avem comanda nivel şi în caz afirmativ se citeşte structura de definire a nivelului în variabila SWAP (SWP).




A
Am comanda "-nivel"
DA
NU
Citeşte structura de definire a nivelului în variabila SWAP (SWP), Size, Mttf, TimpAccFT, TimpAccRD, TimpAccWR, MissPenalty, Aceasta e structura de tip MemHye
Verific Asociativitate Validă
NU
Afişez mesaj: "Nu poate exista nivel din cauza asociativităţii"
DA
Verific Dimensiune_Bloc Validă
NU
Afişez mesaj "Nu e fixată dimensiunea bloc"
DA
Asociativitatea e complet asociativă
DA
Asociativitatea ← Size/Dimensiune_bloc
Nu
Asociativitatea ←Tip_Asociativitate
Atribuirea structurii nivelului din SWP în structura Hye[Nivel]
Hye[Nivele].Mttf←swp.Mttf
Hye[Nivele].Size←swp.Size
Hye[Nivele].TimpAccFT←swp.TimpAccFT
Hye[Nivele].TimpAccRD←swp.TimpAccRD
Hye[Nivele].TimpAccWR←swp.TimpAccWR
Hye[Nivele].MissPenalty←swp.MissPenalty
Asociativitatea e complet asociativă
DA
Alocăm memoria cache ca structură MemCell
Nu
Alocăm memoria cache ca structură MemCell
Alocăm linia cache ca structură MemLine
Alocăm linia cache ca structură MemLine
B

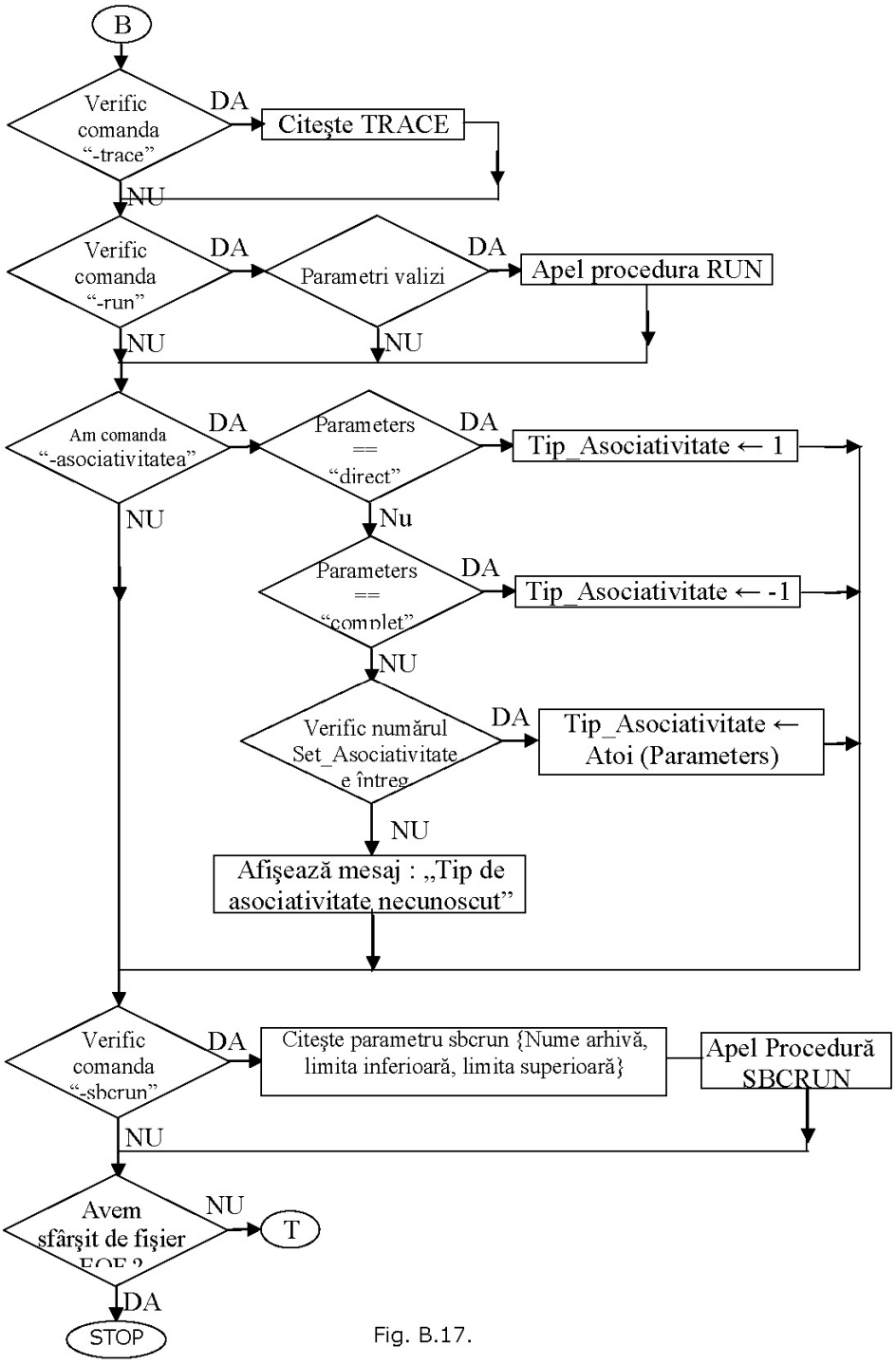

B
Verific comanda "-trace"
DA
Citeşte TRACE
NU
Verific comanda "-run"
DA
Parametri valizi
DA
Apel procedura RUN
NU
NU
Am comanda "-asociativitatea"
DA
Parameters == "direct"
DA
Tip_Asociativitate ← 1
NU
Nu
Parameters == "complet"
DA
Tip_Asociativitate ← -1
NU
Verific numărul Set_Asociativitate e întreg
DA
Tip_Asociativitate ← Atoi (Parameters)
NU
Afişează mesaj : „Tip de asociativitate necunoscut"
Verific comanda "-sbcrun"
DA
Citeşte parametru sbcrun {Nume arhivă, limita inferioară, limita superioară}
Apel Procedură SBCRUN
NU
Avem sfârşit de fişier EOF ?
NU
T
DA
STOP

Fig. B.17.

Structura de date are forma : Size (dimensiune), Mttf, TimpAccFT (timp acces fetch), TimpAccRD (timp acces read) TimpAccWR (timp acces write) şi MissPenalty. Aceasta este o structura de tip MemHye. Se testează dacă asociativitatea e validă. Dacă nu este validă atunci afişez mesajul: "Nu poate exista nivel din cauza asociativităţii". Se testează dacă Dimensiune_Bloc este validă. Dacă nu este validă atunci afişez mesajul: "Nu este fixată dimensiunea bloc". Se testează tipul asociativităţii. Dacă asociativitatea este complet asociativă atunci se atribuie variabilei Asociativitatea, raportul Size/Dimensiune_bloc. Dacă nu avem o asociativitate compet asociativă, se atribuie structura nivelului din SWP în structura Hye[Nivel]. Se testează din nou dacă asociativitatea este complet asociativă şi în ambele situaţii se alocă cache-ul ca structură MemCell şi se alocă linia cache ca structură MemLine. Se verifică comanda "trace" şi în caz afirmativ se citeşte trace-ul. Verificăm comanda "run" şi dacă parametri sunt valizi se face apelul procedurii run. În continuare se verifică comanda "asociativitatea", în funcţie de valoarea citită, se atribuie valoarea 1, pentru maparea directă şi valoarea -1 maparea pentru complet asociativă.

Se testează dacă parametrul Set_Asociativitate este întreg şi în caz afirmativ se atribuie parametrului Tip_Asociativitate valoarea Atoi(Parameters), în caz contrar se afişează mesajul " Tip de asociativitate necunoscut". Se verifică dacă avem comanda "sbcrun". În caz afirmativ, se citeşte parametrul sbcrun (Nume arhivă, limita inferioară, limita superioară) şi se face apelul procedurii SBCRUN. În final se testează, dacă am ajuns la sfârşitul fişierului (EOF?), dacă nu s-a ajuns, se revine în punctul notat cu T, situat la începutul ordinogramei, dacă da, se iese din funcţie.

Ordinograma funcţiei WriteData(), este prezentată în fig. B.18. Se face afişarea numărului de niveluri ale ierarhiei, şi se iniţializează variabilele Halt cu 0 şi count cu 1. Se face afişarea structurei ierarhiei sub forma:

----------[LEVEL count]----------

MTTF Level : Hye[count]. MTTF

Se testează variabila Hye[count].Times, şi în cazul în care este egală cu zero, se afişează mesajul "Cache Delay Average Level Nedeterminat" şi se iese din program returnându-se valoarea 1. Dacă variabila Hye[count].Times este diferită de zero, se afişează structura:

Delay Average Level: Hye[count].Delay

Times Level: Hye[count].Times

Se incrementează variabila count şi se testează dacă count ≤ Nr. niveluri. Se repetă procedura de afişare, pentru toate nivelurile din care este alcătuită ierarhia, şi în momentul când avem count > Nr. niveluri, se trece la calculul MTTF-ului ierarhiei cu funcţia compute_MTTF() din programul dlr.h. Se afişează MTTF-ul Ierarhiei, şi se testează dacă avem un indicator CDLR valid. Dacă CDLR-ul nu este

valid se afişează mesajul: ”Imposibil de calculat CDLR”, se iese din program şi se returnează valoarea -1.

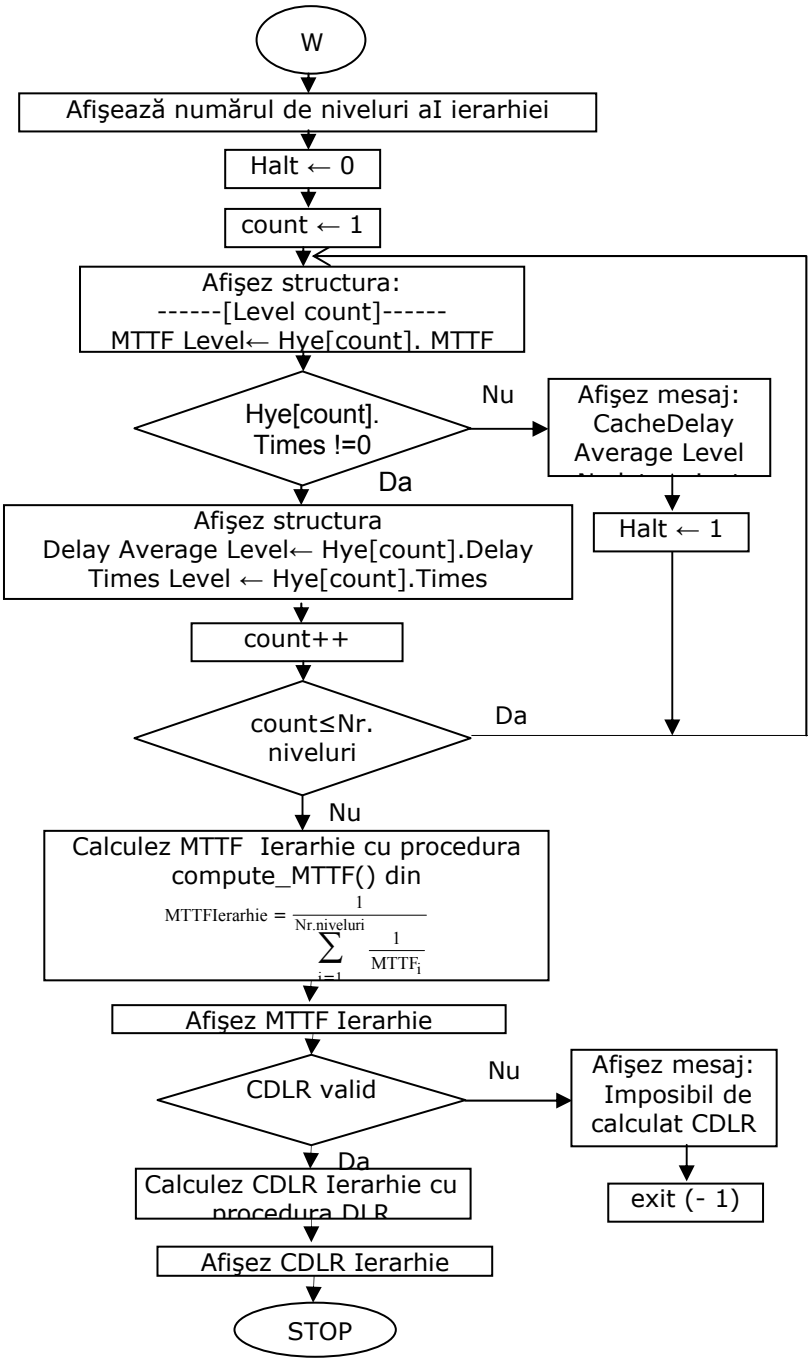


Fig. B.18.

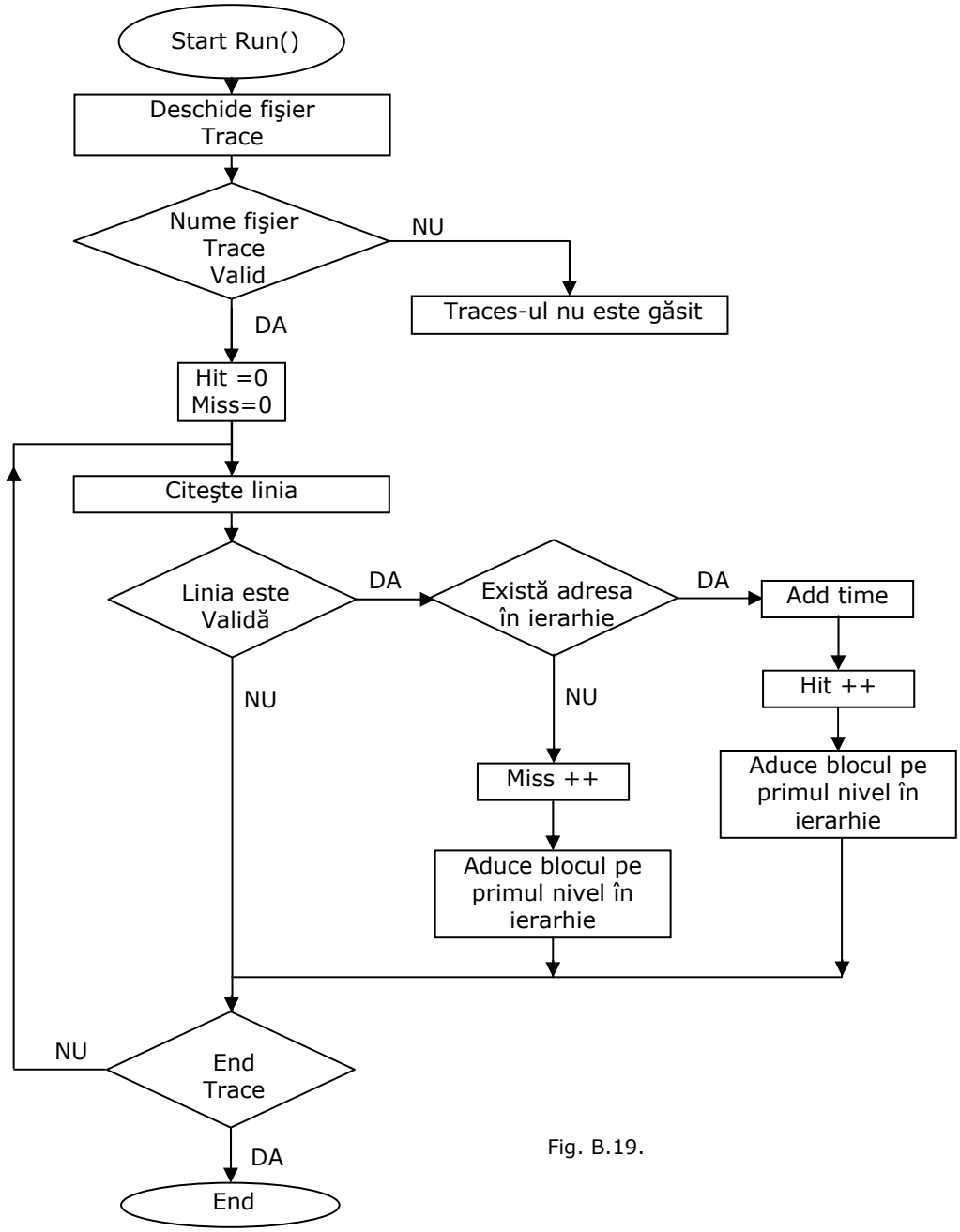


Fig. B.19.

Dacă în schimb CDLR-ul este valid, atunci se calculează CDLR-ul ierarhiei cu procedura DLR şi se afişează CDLR-ul ierarhiei, moment în care se termină execuţia acestui program.

Ordinograma funcţiei Run(), este prezentată în fig. B.19.

Se deschide fişierul trace şi se verifică dacă numele fişierului Trace este valid. Dacă numele Trace-ului nu este valid se afişează mesajul "Traces-ul nu este găsit". Dacă în schimb acesta este valid se iniţializează variabilele Hit şi Miss cu valoarea 0. Se citeşte linia din trace şi se testează dacă ea este validă. Dacă ea este validă, se testează dacă există adresa în ierarhie şi se incrementează variabila Hit, după care se aduce blocul pe primul nivel în ierarhie. Dacă adresa nu este găsită în ierarhie atunci se incrementează variabila Miss, după care se aduce blocul pe primul nivel în ierarhie. Testăm dacă am ajuns la sfârşitul fişierului Traces (End Trace), dacă da, se iese din program, în caz contrar se repetă acest proces pentru toate liniile din Trace.

În figura B.20 este reprezentată ordinograma programului dlr(). Se calculează CDLR-ul statistic al ierarhiei cu funcţia incercare(), iar MTTF-ul îl calculăm conform formulei $\mathrm{MTTF} = \frac{1}{\sum \frac{1}{\mathrm{MTTF_n}}}$.

Calculăm şi CDLR-ul analitic al ierarhiei.

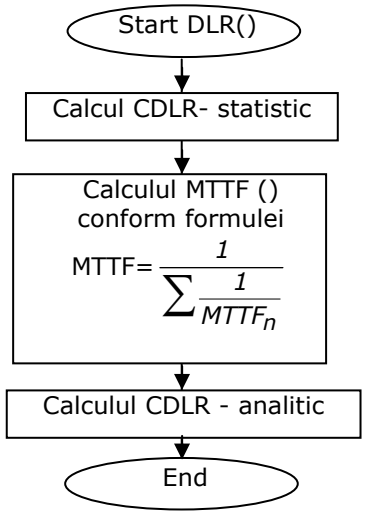


Fig. B.20.

Ordinograma următoare din fig. B.21, prezintă programul data.h. Ea defineşte structura MemCell, ce are câmpurile: Adress, Counter, Plin, Timer.

Tot aici este definită structura MemHye, care conţine câmpurile: Mttf, Size, TimpAccRD, TimpAccWR, TimpAccFT, MissPenalty, Delay, Times, Total. Definim celulele de memorie ale ierarhiei cu relaţia: MemCell ** Level [NIVELE_MAXIM], se

defineşte structura pentru fiecare nivel de memorie MemHye Hye[NIVELE_MAXIM], după care se defineşte linia din memorie MemLine *Line[NIVELE_MAXIM] şi în finalul ordinogramei se definesc variabilele globale ale programului

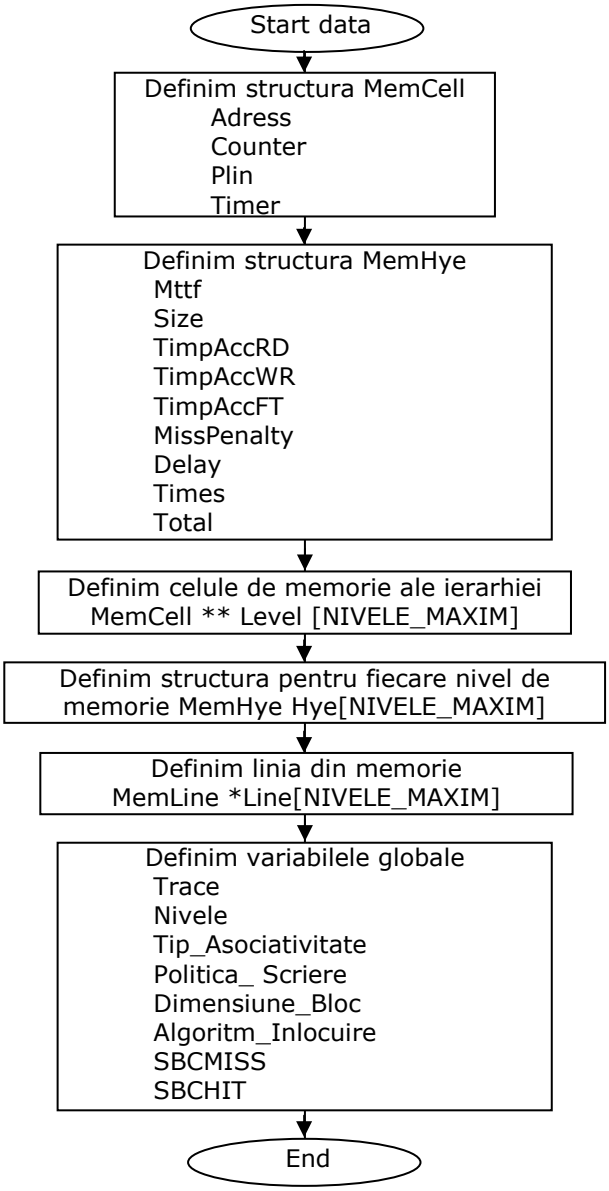


Fig. B.21.

ANEXA C

FIGURI

Implementarea unui cod corector la memoria cache RAM a sistemului ce are capacitatea de 128M x 8 biţi

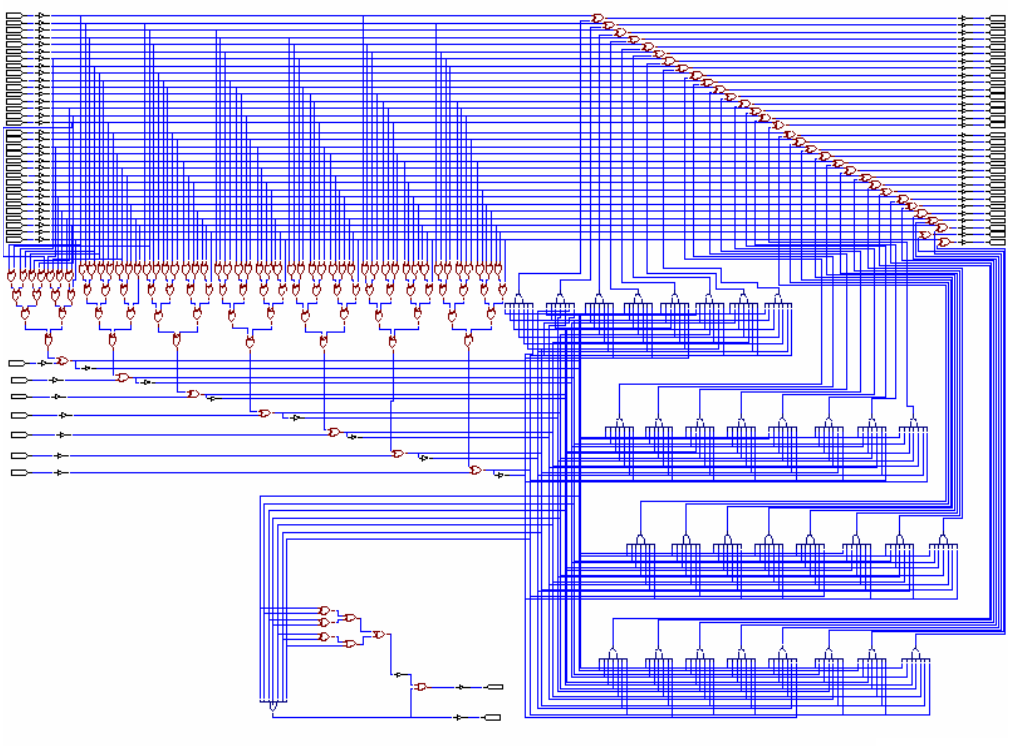


Fig C.1

Implementarea codului corector pentru memoria cache RAM, cu cod Hsaio (39,32,7) pentru determinarea overheadului pe partea de date.

„Number of CLBs 64 out of 1024” adică 6,25% cu ajutorul circuitelor programabile FPGA Xilinx din seria XC4000.

Overhead-ul este de 6,25% (64 CLB-uri configurate dintr-un total de 1024 CLB-uri)

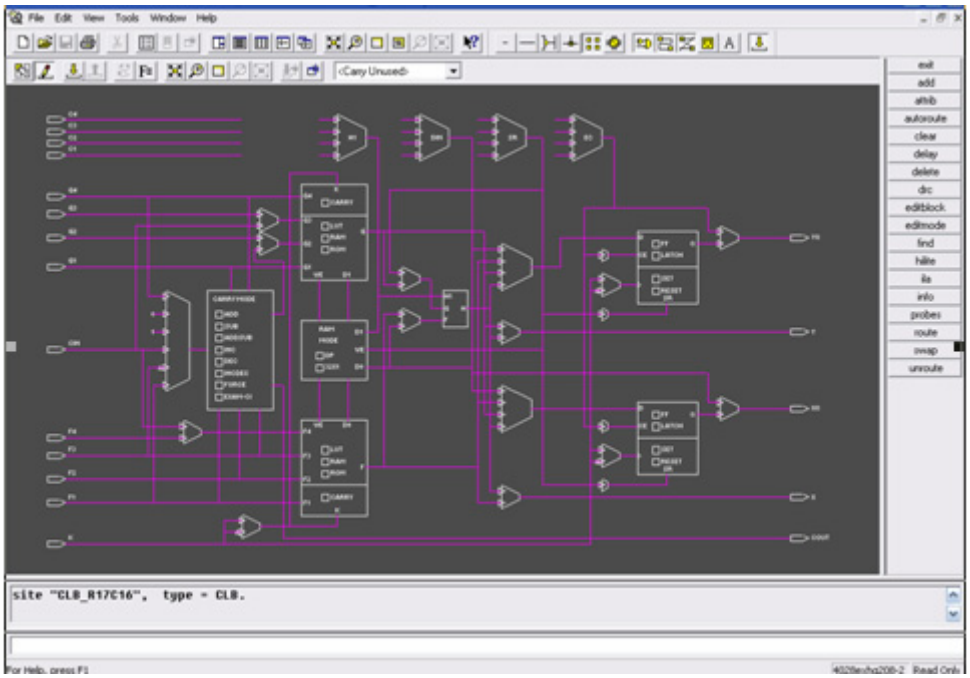


Fig. C.2.

În fig. C2. sunt prezentate elementele unui bloc logic configurabil (CLB – Configurable Logic Block). În acestă figură, se observă că CLB-ul nu este încă configurat de către programul Xilinx. Configurarea CLB-urilor se va face după implementarea schemei din fig. C.1.

Implementarea codului corector pentru memoria cache RAM, cu ajutorul circuitelor programabile FPGA Xilinx din seria XC4000

În fig. C.3. se observă acoperira plăcii cu blocurile logice configurabile şi de asemenea se pot observa şi conexiunile dintre ele.

Implementarea codului corector pentru memoria cache RAM, cu ajutorul circuitelor programabile FPGA Xilinx din seria XC4000.

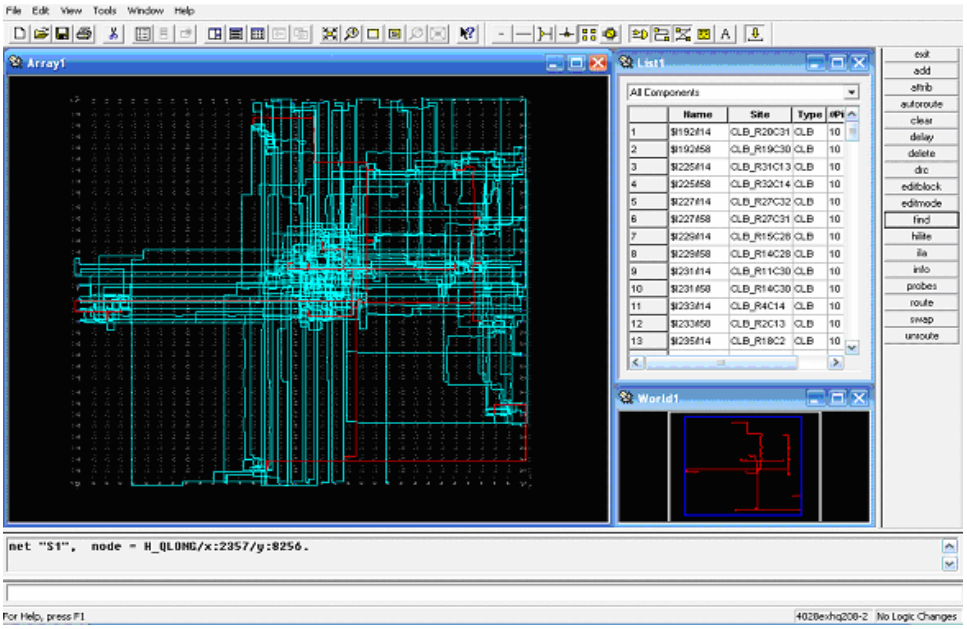


Fig. C.3.

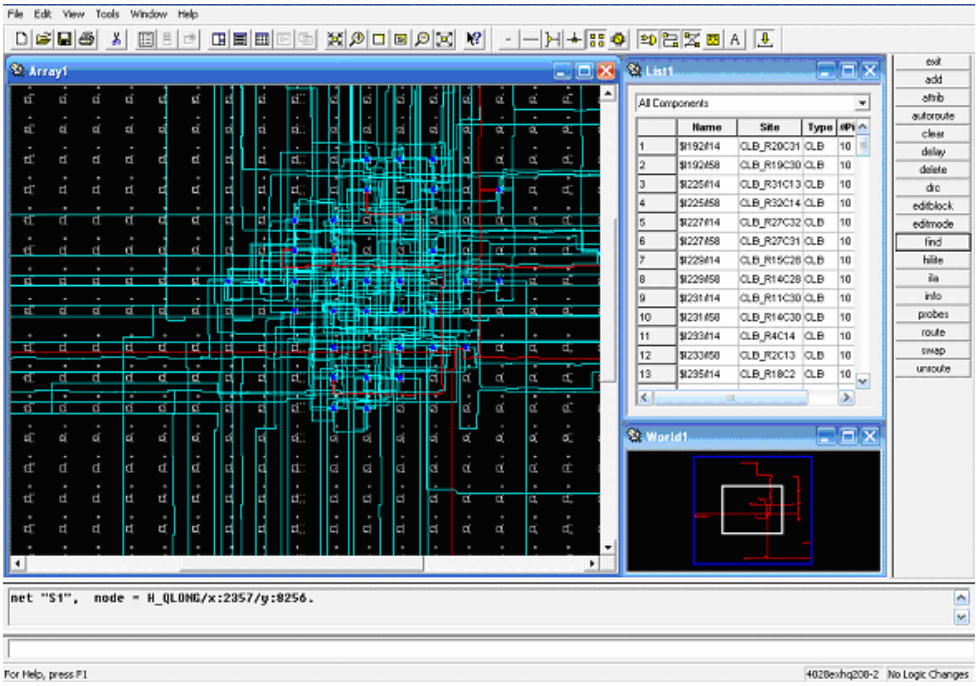


Fig. C.4.

În acest detaliu din fig. C.4. se observă o parte din blocurile logice configurabile ale schemei precum şi conexiunile dintre ele. Implementarea codului corector pentru memoria cache RAM, cu ajutorul circuitelor programabile FPGA Xilinx din seria XC4000

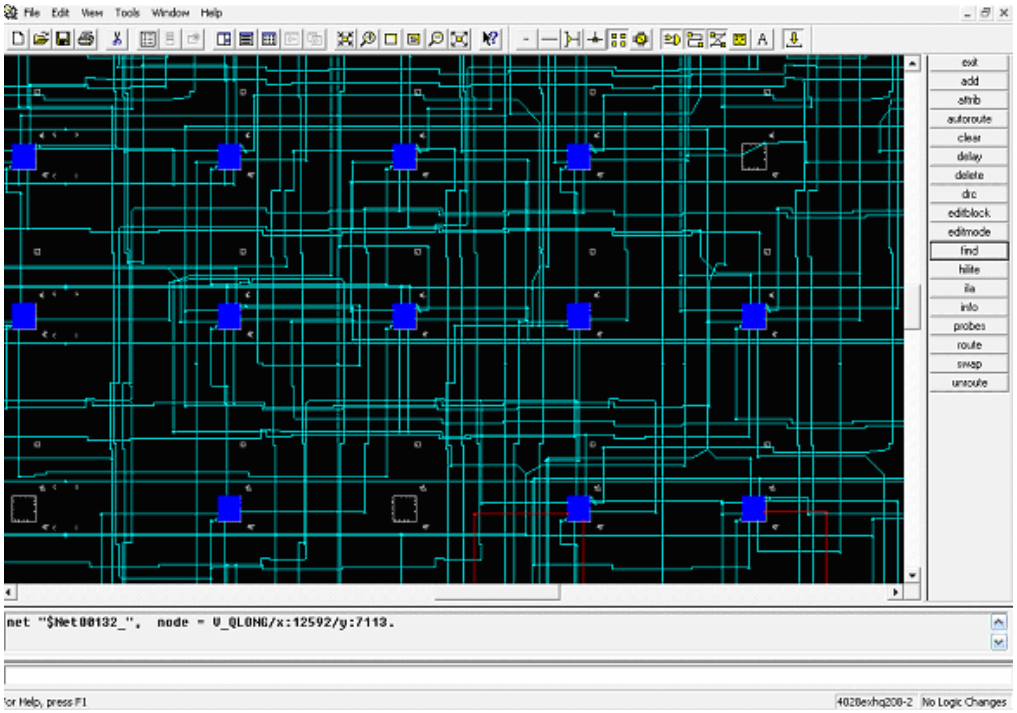


Fig. C.5.

În detaliul din fig. C.5. se observă un număr de 12 blocuri logice configurabile precum şi conexiunile dintre ele.

Implementarea codului corector pentru memoria cache RAM, cu ajutorul circuitelor programabile FPGA Xilinx din seria XC4000

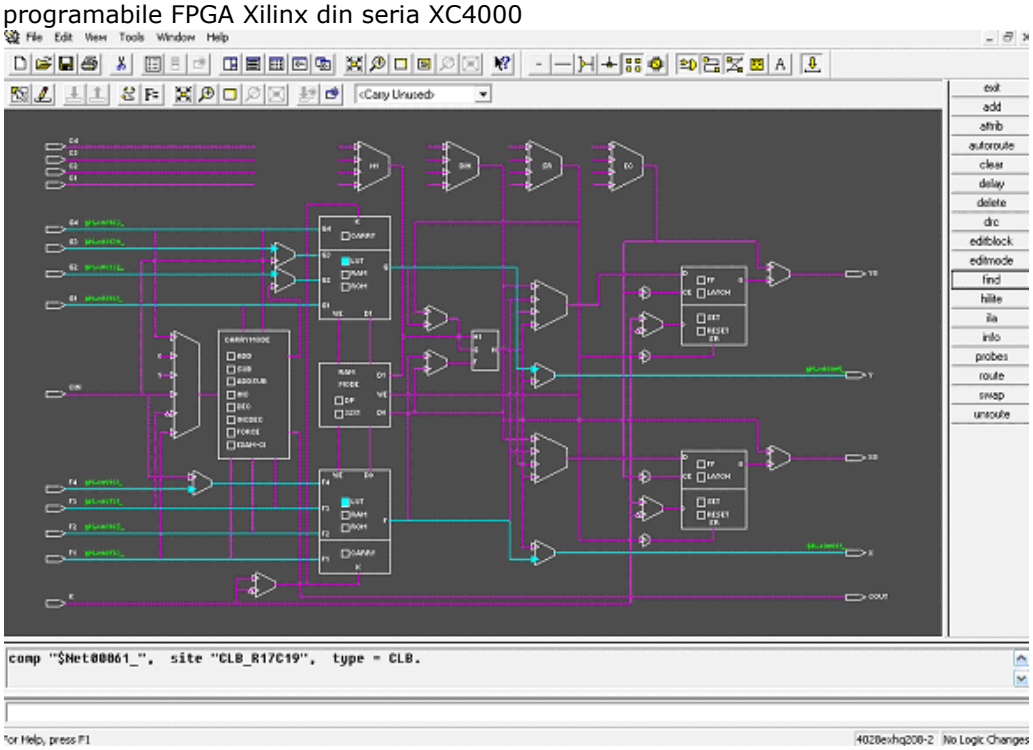


Fig. C.6.

Bloc logic configurabil care este deja configurat de către programul Xilinx.

Implementare cu circuite Xilinx FPGA dim familia XC4000 a memoriei cache RAM cu cod Hsaio

(13,8,5)pt determinarea overheadului pe partea de date

Number of CLBs	12 out of 1024	1%

Overhead-ul este de 1,367 % (14 CLB-uri configurate dintr-un total de 1024 CLB-uri)

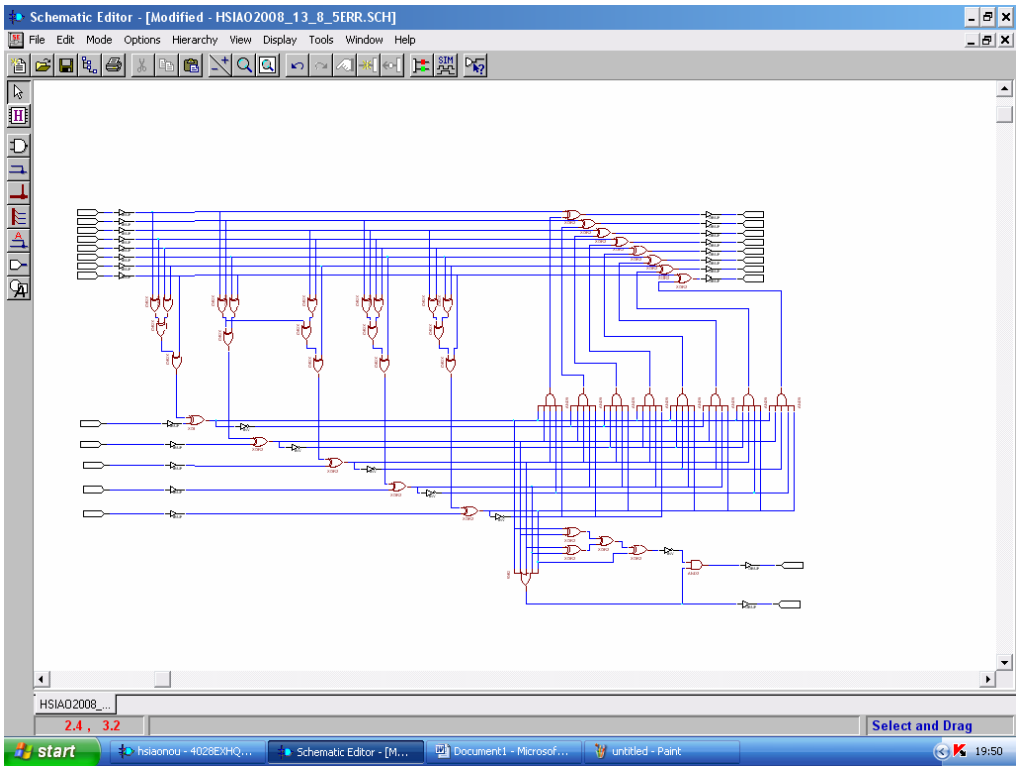


Fig. C.7.

În figura C.7.este prezentatâ o altâ implementarea a unui cod corector la o memorie cache RAM

ANEXA D

FIŞIERE DE INTRARE ŞI FIŞIERE DE IEŞIRE REZULTATE ÎN URMA SIMULĂRII

fişier spec1win.in

```
- asociativitatea 1
- dimensiune_bloc 4
- politica_scriere back
- algoritm_inlocuire random
- nivel 1024 10000000 1 3 5 10
- nivel 4096 900000000 10 20 50 150
- trace fft1.prg
- run
```

Dacă folosim de exemplu, fişierul de intrare specwin1.in, fişierul de ieşire, spec1win.out obţinut în urma simulării va arăta astfel:

fişier spec1win.out

```
----------------[ LEVEL 1 ]--------------------------
MTTF Level : 10000000
Delay Average Level: 752
Times Level: 7220101
----------------[ LEVEL 2 ]--------------------------
MTTF Level : 900000000
Delay Average Level: 1612
Times Level: 7214810

MTTF Ierarhie: 9890109
CDLR ierarhie: 0.000179
```

fişier spec2win.in

```
- asociativitatea 4
- dimensiune_bloc 64
- politica_scriere back
- algoritm_inlocuire lru
- nivel 1024 1000000 1 3 5 10
- nivel 4096 90000000 10 20 50 150
- nivel 8192 200000000 20 50 150 300
```

- trace fft1.prg
- run

fişier spec2win.out

```
----------[ LEVEL 1 ]-------------------------
MTTF Level : 1000000
Delay Average Level: 664
Times Level: 7237767

----------[ LEVEL 2 ]-------------------------
MTTF Level : 90000000
Delay Average Level: 1663
Times Level: 7233696

----------[ LEVEL 3 ]-------------------------
MTTF Level : 200000000
Delay Average Level: 24326
Times Level: 6229528

MTTF Ierarhie: 984144
CDLR ierarhie: 0.014011
```

fişier spec3win.in

- asociativitatea complet
- dimensiune_bloc 64
- politica_scriere back
- algoritm_inlocuire fifo
- nivel 1024 254231 1 1 1 5
- nivel 4096 2542310 1 3 5 10
- trace fft1.prg
- run

fişier spec3win.out

```
----------[ LEVEL 1 ]-------------------------
MTTF Level : 254231
Delay Average Level: 21
Times Level: 7237753

----------[ LEVEL 2 ]-------------------------
MTTF Level : 2542310
Delay Average Level: 21
Times Level: 7236919

MTTF Ierarhie: 231119
CDLR ierarhie: 0.000827
```

fişier spec4win.in

- asociativitatea 8
- dimensiune_bloc 64
- politica_scriere back

- algoritm_inlocuire lru
- nivel 1024 1000000 1 3 5 10
- nivel 4096 90000000 10 20 50 150
- nivel 8192 200000000 20 50 150 300
- trace fft1.prg
- run

fişier spec4win.out

```
----------------[ LEVEL 1 ]--------------------------
MTTF Level : 1000000
Delay Average Level: 664
Times Level: 7237763

----------------[ LEVEL 2 ]--------------------------
MTTF Level : 90000000
Delay Average Level: 1663
Times Level: 7233417

----------------[ LEVEL 3 ]--------------------------
MTTF Level : 200000000
Delay Average Level: 13265
Times Level: 5778471

MTTF Ierarhie: 984144
CDLR ierarhie: 0.008480
```

fişier spec5win.in

- asociativitatea 8
- dimensiune_bloc 64
- politica_scriere back
- algoritm_inlocuire fifo
- nivel 1024 1000000 1 3 5 10
- nivel 4096 90000000 10 20 50 150
- nivel 8192 200000000 20 50 150 300
- trace fft1.prg
- run

fişier spec5win.out

```
----------------[ LEVEL 1 ]--------------------------
MTTF Level : 1000000
Delay Average Level: 664
Times Level: 7237763
----------------[ LEVEL 2 ]--------------------------
MTTF Level : 90000000
Delay Average Level: 1663
Times Level: 7233417
----------------[ LEVEL 3 ]--------------------------
MTTF Level : 200000000
Delay Average Level: 13265
Times Level: 5778471
```

MTTF Ierarhie: 984144
CDLR ierarhie: 0.008480

fişier spec6win.in

- asociativitatea 8
- dimensiune_bloc 4
- politica_scriere back
- algoritm_inlocuire random
- nivel 1024 1000000 1 3 5 10
- nivel 4096 90000000 10 20 50 150
- nivel 8192 200000000 20 50 150 300
- trace fft1.prg
- run

fişier spec6win.out

```
----------------[ LEVEL 1 ]--------------------------
MTTF Level : 1000000
Delay Average Level: 1327
Times Level: 7169886
----------------[ LEVEL 2 ]--------------------------
MTTF Level : 90000000
Delay Average Level: 7039
Times Level: 7096246
----------------[ LEVEL 3 ]--------------------------
MTTF Level : 200000000
Delay Average Level: 9380
Times Level: 7089579

MTTF Ierarhie: 984144
CDLR ierarhie: 0.012511
```

fişier spec7win.in

- asociativitatea 8
- dimensiune_bloc 16
- politica_scriere back
- algoritm_inlocuire lru
- nivel 1024 1000000 1 3 5 10
- nivel 4096 90000000 10 20 50 150
- nivel 8192 200000000 20 50 150 300
- trace fft1.prg
- run

fişier spec7win.out

```
----------------[ LEVEL 1 ]--------------------------
MTTF Level : 1000000
Delay Average Level: 664
Times Level: 7237763
----------------[ LEVEL 2 ]--------------------------
MTTF Level : 90000000
```

```
Delay Average Level: 4662
Times Level: 7213897
----------------[ LEVEL 3 ]--------------------------
MTTF Level : 200000000
Delay Average Level: 7260
Times Level: 6617355

MTTF Ierarhie: 984144
CDLR ierarhie: 0.008810
```

fişier spec8win.in

```
- asociativitatea direct
- dimensiune_bloc 16
- politica_scriere back
- algoritm_inlocuire lru
- nivel 1024 1000000 1 3 5 10
- nivel 4096 90000000 10 20 50 150
- nivel 8192 200000000 20 50 150 300
- trace fft1.prg
- run
```

fişier spec8win.out

```
----------------[ LEVEL 1 ]--------------------------
MTTF Level : 1000000
Delay Average Level: 2326
Times Level: 7220101
----------------[ LEVEL 2 ]--------------------------
MTTF Level : 90000000
Delay Average Level: 4984
Times Level: 7214810
----------------[ LEVEL 3 ]--------------------------
MTTF Level : 200000000
Delay Average Level: 45115
Times Level: 6061816

MTTF Ierarhie: 984144
CDLR ierarhie: 0.028095
```

fişier spec9win.in

```
- asociativitatea 16
- dimensiune_bloc 4
- politica_scriere back
- algoritm_inlocuire random
- nivel 1024 1000000 1 3 5 10
- nivel 4096 90000000 10 20 50 150
- nivel 8192 200000000 20 50 150 300
- trace fft1.prg
- run
```

fişier spec9win.out

```
----------------[ LEVEL 1 ]--------------------------
MTTF Level : 1000000
Delay Average Level: 1327
Times Level: 7169819
----------------[ LEVEL 2 ]--------------------------
MTTF Level : 90000000
Delay Average Level: 4689
Times Level: 7098947
----------------[ LEVEL 3 ]--------------------------
MTTF Level : 200000000
Delay Average Level: 9378
Times Level: 7087549

MTTF Ierarhie: 984144
CDLR ierarhie: 0.009900
```

fişier spec10win.in

```
- asociativitatea 16
- dimensiune_bloc 16
- politica_scriere back
- algoritm_inlocuire lru
- nivel 1024 1000000 1 3 5 10
- nivel 4096 90000000 10 20 50 150
- nivel 8192 200000000 20 50 150 300
- trace fft1.prg
- run
```

fişier spec10win.out

```
----------------[ LEVEL 1 ]--------------------------
MTTF Level : 1000000
Delay Average Level: 664
Times Level: 7237753
----------------[ LEVEL 2 ]--------------------------
MTTF Level : 90000000
Delay Average Level: 3833
Times Level: 7213638
----------------[ LEVEL 3 ]--------------------------
MTTF Level : 200000000
Delay Average Level: 3630
Times Level: 6618873

MTTF Ierarhie: 984144
CDLR ierarhie: 0.006074
```

fişier spec2000_2win.in

```
- asociativitatea 4
- dimensiune_bloc 16
- politica_scriere back
- algoritm_inlocuire lru
- nivel 4096  900000000 1 3 5 10
- nivel 16384 1800000000 2 8 10 20
- nivel 16384 18000000000 3 5 10 20
```

- trace fft1.prg
- run
fişier spec2000_2win.out

```
----------[ LEVEL 1 ]-------------------------
MTTF Level : 900000000
Delay Average Level: 262
Times Level: 7220041
----------[ LEVEL 2 ]-------------------------
MTTF Level : 1800000000
Delay Average Level: 525
Times Level: 7214364
----------[ LEVEL 3 ]-------------------------
MTTF Level : 820130816
Delay Average Level: 2843
Times Level: 6127684

MTTF Ierarhie: 346502226
CDLR ierarhie: 0.000376
```

ANEXA E

MEDII DE EVALUARE A INDICILOR DE PERFORMANŢĂ LA CACHE-URI. ASPECTE TEORETICE ALE SIMULATORULUI SMP CACHE 2.0

SMP Cache este un simulator bazat pe traces-uri pentru sisteme cu cache şi cu multiprocesoare simetrice (SMP-uri) care utilizează un sistem de memorie bazată pe partajarea magistralei. Acest simulator poate fi instalat pe calculatoare personale ce au instalate sistemele de operare Windows, având interfaţă grafică asemănătoare sistemului de operare Windows.

Parametrii simulatorului ce pot fi studiaţi sunt: influenţa numărului de procesoare, politicile de înlocuire, localitatea programelor, protocoalele de coerenţă a cache-ului, scheme de arbitrare a magistralei, funcţia de mapare, dimensiunea cache-ului (blocuri în cache), numărul seturilor de cache (pentru cache-urile set asociative), numărul cuvintelor dintr-un bloc (dimensiunea blocului). Interfaţa simulatorului este destul de uşoară şi astfel simulatorul poate fi folosit în scopuri didactice şi pentru studiul comportării cache-urilor.

Simulatorul SMP Cache ne permite să vizualizăm într-un mod grafic clar cum evoluează microprocesorul pe măsura execuţiei programelor (adică pe măsură ce traces-urile de memorie traces sunt citite) [VeSG01].

În tabelul E.1. sunt prezentate caracteristicile arhitecturale ale simulatorului SMPCache 2.0 [VeSG01].

Procesoare în SMP	1, 2, 3, 4, 5, 6, 7 sau 8
Protocoale de coerenţă a cache-ului	MSI, MESI sau DRAGON
Scheme pentru arbitrarea magistralei	Random (aleatoare), LRU sau LFU
Dimensiunea cuvântului (biţi)	8, 16, 32 sau 64
Numărul cuvintelor dintr-un bloc (dimensiunea blocului)	1, 2, 4, 8, 16, 32, 64, 128, 256, 512 sau 1024
Numărul de blocuri din memoria principală	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288, 1048576, 2097152 sau 4194304
Numărul de blocuri din memoria cache	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 sau 2048
Funcţia de mapare	directă, set asociativă sau complet asociativă
Numărul seturilor de cache (dacă cache-ul este set asociativ)	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 sau 2048
Politicile de înlocuire	Random, LRU, FIFO sau LFU
Strategiile (politicile) de scriere	Write-Back (pentru protocolul de coerenţă a cache-ului)
Numărul de niveluri de cache din ierarhia de memorii	1
Referinţe	La cuvinte din memorie
Dimensiunea maximă a blocului	8 Kocteţi
Dimensiunea maximă a memoriei principale	32 Gocteţi
Dimensiunea maximă a cache-ului (excludem etichetele, biţii de stare a blocului, numărătoarele, etc.)	16 Mocteţi

Tabelul E.1.

Fişierele de configurare au formatul prezentat în fig. E.1. După cum se poate observa, "comentariile" (indicând tipul de configurare) apar în liniile impare iar selecţia caracteristicilor în liniile pare (sunt valorile numerice) [VSMZ00].

În continuare, în fig. E.1. se prezintă structura unui fişier de configurare:

Processors in SMP:
3
Cache coherence protocol:
2
Scheme for bus arbitration:
1
Word wide (bits):
64
Words by block:
128
Blocks in main memory:

1024
Blocks in cache:
64
Mapping:
3
Number of cache sets:
0
Replacement policy:
2
Cache levels:
1
Writing strategy:
2

Fig. E.1.

În cazul în care avem configuraţii numerice, configurarea numerică a SMP Cache-ului se face prin selecţia opţiunii în măsura scrierii acesteia în fişier, de exemplu: numărul de procesoare - 3, dimensiunea cuvântului – 64 biţi, numărul cuvintelor dintr-un bloc - 128. Pentru restul configuraţiilor, un cod numeric este asociat cu una din opţiunile posibile.

Tabelul E.2. ne prezintă opţiunile de configurare şi codurile numerice asociate acestora.

	Opţiuni posibile de configurare	**Cod numeric asociat**
Protocolul de coerenţă a cache-ului	MSI MESI DRAGON	1 2 3
Scheme pentru arbitrarea magistralei	Random LRU LFU	1 2 3
Funcţia de mapare	Directă Set-asociativă Complet-asociativă	1 2 3
Numărul de seturi a cache-ului	Nu există 1, 2, 4, 8, 16, 32, 64, 128,256, 512, 1024 sau 2048	0,1, 2, 4, 8, 16, 32, 64, 128,256, 512, 1024 sau 2048
Politica de înlocuire	Nu există Random LRU FIFO LFU	0 1 2 3 4
Strategia de scriere	Write-Through Write-Back	1 2

Tabelul E.2.

Voi prezenta în continuare o descriere a pachetului de programe SMPCache [VeSG01].

1. Fişiere de tip Trace

Pentru a putea lucra cu simulatorul, trebuie să utilizăm fişiere de date care să facă apeluri la adrese de memorie, apeluri solicitate de procesor pe parcursul execuţiei unui program: aceste apeluri se numesc traces-uri de memorie. Fişierele trace sunt fişiere ASCII de date cu extensia ".prg", aceste fişiere sunt alcătuite din mai multe linii, fiecare linie fiind alcătuită din două câmpuri numerice separate printr-un spaţiu. Aceste două câmpuri sunt Eticheta şi Valoare. Formatul unei linii din fişierul trace:

Eticheta Valoare

unde:

Eticheta - este un număr zecimal care identifică tipul operaţiei de acces la memorie solicitat de către procesor (CPU), într-un timp dat: (0) instrucţiuni de Fetch, (2) citirea unei date din memorie şi (3) scrierea unei date în memorie.

Valoare - este un număr hexazecimal care indică adresa efectivă a cuvântului de memorie care trebuie să fie accesat de către procesor (CPU). Această adresă va fi transferată de simulator pentru localizarea cuvântului în ierarhia de memorii [VSMZ00].

În fig. E.2. se prezintă un exemplu de fişier trace, fişier ce urmează a fi încărcat într-un anumit procesor din Simulatorul SMPCache 2.0.

Pentru a putea înţelege mai bine vom prezenta un fişier de tip trace. În fig. E.2. este prezentată o porţiune din acest fişier, care conţine 6 instrucţii de Fetch a unui anumit program. Trei instrucţii sunt instrucţii de citire a datelor şi o instrucţie de scriere în memorie. În total, aceste 6 instrucţii implică 10 accese la memorie.

```
0       00004be2
0       00004ca5
3       0000eb51
2       0000ef00
0       0000ed35
0       0000fa36
2       0000ef01
0       0000782b
2       0000efcd
0       0000782f
```

Fig. E.2.

2. *Trace-uri cu un singur procesor*

Nume Trace	Clasificare	Limbajul de programare	Comentarii
Hydro	Punct flotant	-	Astrofizica: ecuaţia hidrodinamică Naiver Stokes
Nasa7	Punct flotant	Fortran	O colecţie de 7 nuclee. Pentru fiecare nucleu, programul îşi generează propriile date de intrare, execută nucleul şi compară rezultatul obţinut cu rezultatul aşteptat.
Cexp	Întreg	C	O porţiune de compilator Gnu C care manifesta un puternic comportament aleatoriu
Mdljd	Punct flotant	Fortran	Rezolvă ecuaţia de mişcare pentru un model de 500 de atomi care interacţionează prin potenţialul idealizat al lui Lennard-Jones. Este un program numeric care expune comportamentul combinat aleatoriu şi ciclic.
Ear	Punct flotant	---	Acest trace, ca şi restul, a fost dat de Nadeem Malik de la IBM
Comp	Întreg	C	Foloseşte codarea Lempel-Ziv pentru compresia datelor. Comprimă un fişier de 1Moctet de 20 de ori.
Wave	Punct flotant	Fortran	Rezolvă ecuaţia Maxwell şi ecuaţia particulelor electromagnetice de mişcare
Swm	Punct flotant	Fortran	Rezolvă un sistem de ecuaţie SHALLOW WATER utilizând aproximarea diferenţelor finite pe un grid de 256*256
Ucomp	Întreg	C	Versiunea necomprimată a lui Comp

Tabelul E.3.

Ca urmare vom configura simulatorul SMPCache 2.0 cu un singur procesor, şi vom folosi traces-uri cu un singur procesor.

Acest simulator lucrează cu traces-urile benchmark-urilor SPEC'92 (Hydro, Nasa7, Cexp, Mdljd, Ear, Comp. Wave, Swm şi Ucomp). Traces-urile folosite reprezintă o varietate extinsă de aplicaţii de programe „reale". Aceste traces-uri provin de la Paralel Architecture Research Laboratory (PARL), New Mexico State Univeristy (NMSU), şi sunt publicate pe serverul ftp anonim tracebase.nmsu.edu. Aceste traces-uri au avut formate diferite, de exemplu Dinero sau PDATS, dar au fost schimbate în formatul de trace al SMPCache. O scurtă prezentare a traces-urilor este dată în Tabelul E.3.

ANEXA F

LISTING-URILE PROGRAMELOR CE COMPUN MEDIUL CDLR SPEC 2000

1. LISTING-UL PROGRAMULUI PRINCIPAL: ***main.cpp***

```
# include "read.h"


 int main(int argn, char **argv)
{
 unsigned long t1,t2;
 t1 = (long)time(NULL);
 srand(t1);
 if(argn > 1) read_data(argv[1]);
 else printf("Nu exista fisierul de intrare");
 if(argn > 2) WriteData(argv[2]);
 else WriteData("main.out");
 t2 = (long)time(NULL);
 for (int i=1;i<=Nivele;i++)
   delete Level[i];
 printf("EXECUTIA A DURAT: %ld sec",t2-t1);
 return 0;
}
```

---

2. LISTING-UL PROGRAMULUI: **read.h**

```
# ifndef __READ_DATA_H
# define __READ_DATA_H
# include <stdio.h>
# include <string.h>
# include <stdlib.h>
# include "data.h"
# include "run.h"

int read_data(char *Path_In)
{
  FILE *file;
  MemHye swp;
  char String[120],Command[20],Parameters[100],Size[10];
  //   printf("%s\n",Path_In);
  if ( !(file = fopen(Path_In,"r")) ) return -1;
  while(! feof(file))
  {
    strcpy(String,"");
    strcpy(Command,"");
    strcpy(Parameters,"");
    fgets(String,120,file);
    sscanf(String,"%s %s",Command,Parameters);
    if( Command[0] != '-') continue;
    if( strcmp(Command,"-dimensiune_bloc") == 0)
      sscanf(Parameters,"%d",&Dimensiune_Bloc);
    else if( strcmp(Command,"-politica_scriere") == 0)
```

```
        {
          if( strcmp(Parameters,"back") == 0)  Politica_Scriere = 1;
          else if( strcmp(Parameters,"trough") == 0) Politica_Scriere = 2;
          else printf("Politica de scriere <%s> necunoscuta...\n",
                    Parameters);
          printf("Politica de scriere: %d\n",Politica_Scriere);
        }
    else if( strcmp(Command,"-algoritm_inlocuire") == 0)
        {
          if( strcmp(Parameters,"random") == 0) Algoritm_Inlocuire = 3;
          else if( strcmp(Parameters,"lru") == 0) Algoritm_Inlocuire = 1;
          else if( strcmp(Parameters,"lfu") == 0) Algoritm_Inlocuire = 2;
          else if( strcmp(Parameters,"fifo") == 0) Algoritm_Inlocuire = 4;
          else printf("Algoritm de inlocuire <%s> necunoscut...\n",
                    Parameters);
          printf("Algoritmul ales: %d\n",Algoritm_Inlocuire);
        }
    else if( strcmp(Command,"-nivel") == 0)
        {
           //             printf("---------- %s\n",Parameters);
          sscanf(String,"-nivel %ld %ld %ld %ld %ld %ld",&swp.Size,&swp.Mttf,
                &swp.TimpAccFT,&swp.TimpAccRD,&swp.TimpAccWR,
                &swp.MissPenalty);
/*---------------------------
          if( (swp.Size = atoi(Size)) == 0)
          {
            if(Size[strlen(Size)-1]=='k' || Size[strlen(Size)-1]=='K')
            {
```

```
                Size[strlen(Size)-1] = 0;
                swp.Size = atoi(Size)*1024;
              }
              else if( Size[strlen(Size)-1]=='M' )
                  {
                    Size[strlen(Size)-1] = 0;
                    swp.Size = atoi(Size)*1024*1024;
                  }
                  else printf("Dimensiune <%s> necunoscuta",Size);
            }
----------------------------------*/
          printf("PARAMETRII:\nSIZE:          %ld\nMTTF:          %ld\nTMP_FETCH:
%ld\nTMP_RD: %ld\nTMP_WR: %ld\nMISS_PENALTY: %ld\n",

swp.Size,swp.Mttf,swp.TimpAccFT,swp.TimpAccRD,swp.TimpAccWR,swp.MissPenalty
);
            Nivele++;
               printf("Numărul de niveluri: %d\n",Nivele);
            if(Tip_Asociativitate == 0)
              printf("Nu poate exista nivelul din cauza asociativitatii...\n");
            else if( Dimensiune_Bloc == 0)
              printf("Nu este fixata dimensiunea blocului...\n");
            else
            {
              Hye[Nivele].Mttf = swp.Mttf;
              Hye[Nivele].Size = swp.Size;
              Hye[Nivele].TimpAccFT = swp.TimpAccFT;
              Hye[Nivele].TimpAccRD = swp.TimpAccRD;
              Hye[Nivele].TimpAccWR = swp.TimpAccWR;
              Hye[Nivele].MissPenalty = swp.MissPenalty;
```

```
            if (Tip_Asociativitate == -1)
                  {
                    printf("Alocarea memoriei1 \n");
                 Level[Nivele] = new MemCell* [Hye[Nivele].Size/Dimensiune_Bloc];
                    for(int i=0;i<Hye[Nivele].Size/Dimensiune_Bloc;i++)
                       {
                         Level[Nivele][i] = new MemCell;
                         Level[Nivele][i]->next = 0;
                         Level[Nivele][i]->Adress = 0;
                         Level[Nivele][i]->Timer = 0;
                         Level[Nivele][i]->Counter = 0;
                       }
                  }
            else
                  {
                    Level[Nivele] = new MemCell*
[Hye[Nivele].Size/(Dimensiune_Bloc*Tip_Asociativitate)];
                    for(int
i=0;i<Hye[Nivele].Size/(Dimensiune_Bloc*Tip_Asociativitate);i++)
                     {
                       Level[Nivele][i] = new MemCell;
                         Level[Nivele][i]->next = 0;
                         Level[Nivele][i]->Adress = 0;
                         Level[Nivele][i]->Timer = 0;
                         Level[Nivele][i]->Counter = 0;
                     }
                   }
          }
        }
```

```
    else if( strcmp(Command,"-trace") == 0) strcpy(Trace,Parameters);
    else if( strcmp(Command,"-run") == 0)
        {
            if(Tip_Asociativitate == 0 || Algoritm_Inlocuire == 0 ||
              Politica_Scriere == 0)
              printf("Nu sunt fixati parametrii...\n");
            else if (Dimensiune_Bloc == 0) printf("Nu este fixata dimensiunea
blocului...\n");
            else
                  {
                    printf("PORNIM PROGRAMUL\n");
                    Run();
                  }
        }
    else if( strcmp(Command,"-asociativitatea") == 0)
        {
          if( strcmp(Parameters,"direct") == 0)  Tip_Asociativitate = -1;
          else if( strcmp(Parameters,"complet") == 0)
                                        Tip_Asociativitate = 1;
          else if( (Tip_Asociativitate = atoi(Parameters)) == 0 )
          printf("Tipul de asociativitate <%s> necunosut..\n",Parameters);
            printf("Asociativitate %d\n",Tip_Asociativitate);
        }
   else if( strcmp(Command,"-clock") == 0) Clock = atoi(Parameters);
  }
  fclose(file);
  return 0;
}
# endif
```

3. LISTING-UL PROGRAMULUI: ***data.h***

```
# ifndef __DATA_H_
# define __DATA_H_
# define NIVELE_MAXIM 7
typedef struct MemCell{
   unsigned long Adress;
   unsigned int Counter;
   unsigned int Plin;
   unsigned int Timer;
   MemCell *next;
}MemCell;

typedef struct MemHye{
   unsigned long Mttf;
   unsigned long Size;
   unsigned long TimpAccRD,TimpAccWR,TimpAccFT,MissPenalty;
   unsigned long Delay;
   unsigned long Times;
}MemHye;

/*pointer dublu */MemCell **Level[NIVELE_MAXIM];
MemHye  Hye[NIVELE_MAXIM];
char Trace[20];
int Nivele = 0;
int Tip_Asociativitate = 0;
int Politica_Scriere = 0;
```

```
int Dimensiune_Bloc = 0;
int Algoritm_Inlocuire = 0;
int Clock = 0;
unsigned long Adresa_Swap;
//int Fall_Level;
# endif
```

4. LISTING-UL PROGRAMULUI: ***run.h***

```
# ifndef __RUN_H_
# define __RUN_H_
# include "data.h"
# include "write.h"
# include <stdio.h>
# include <stdlib.h>
# include <time.h>

int AddTime(int);

int InNivel(int nivel,unsigned long Adresa_Bloc)
{
  int i,j;
  MemCell *p;
  if( Tip_Asociativitate == -1 ) i = 0;
  else i = Adresa_Bloc % (Hye[nivel].Size/(Tip_Asociativitate*Dimensiune_Bloc));
  p = Level[nivel][i];
  while(p)
    {
```

```
      if( p->Adress==Adresa_Bloc) return 1;
      p = p->next;
    }
  return 0;
}

int Cauta(unsigned long Adresa)
{
  int i;
  for(i=1;i<=Nivele;i++)
    if(InNivel(i,Adresa/Dimensiune_Bloc)) return i;
    else AddTime(Hye[i].MissPenalty);
  return 0;
}

/*
Asoc pentru selectia liniei din nivelul "i"
*/
int AddTime(int time_step)
{
  MemCell *p;
  int i,j,Asoc;
  for(i=1;i<=Nivele;i++)
    {
      if(Tip_Asociativitate != -1) Asoc = Hye[i].
Size/(Dimensiune_Bloc*Tip_Asociativitate);
      else Asoc = 0;
      for(j=0;j<Asoc;j++)
      {
```

```
        p = Level[i][j];
        while( p )
        {
          p->Timer+= time_step;
          p = p->next;
        }
      }
   }
   return 0;
}

/*
Asoc pentru selectia linei din nivelul "i"
 */
int PrintTime()
{
  MemCell *p;
  int i,j,Asoc;
  for(i=1;i<=Nivele;i++)
     {
       if(Tip_Asociativitate != -1) Asoc = Hye[i].
Size/(Dimensiune_Bloc*Tip_Asociativitate);
       else Asoc = 0;
       for(j=0;j<Asoc;j++)
       {
         p = Level[i][j];
         while( p )
         {
           printf("%8d",p->Adress);
```

```
          p = p->next;
        }
          printf("");
      }
   }
  return 0;
}

/*
Asoc pentru numarul de elementa maxim dintr-o linie "i"
 */
int Scrie(int nivel,unsigned long Adresa)
{
   if(nivel>Nivele) return -1;
   //   printf("Nivelul: %d ",nivel);
   unsigned long Adresa_Bloc = Adresa/Dimensiune_Bloc,Asoc,timer = 0;
   MemCell *prim,*p,*np;
   int i,j;
   if( Tip_Asociativitate == -1) i = 0;
   else i = Adresa_Bloc % (Hye[nivel]. Size/(Tip_Asociativitate*Dimensiune_Bloc));
   prim = Level[nivel][i];
   p = prim;
   j = 0;
   if(Tip_Asociativitate == -1) Asoc = Hye[nivel].Size/Dimensiune_Bloc;
   else Asoc = Tip_Asociativitate;
   while(p)
   {
     if(p->Adress == Adresa_Bloc) break;
```

```
  p = p->next;
  j++;
}
//printf("%5d\n",j);
if(p == NULL)
{
  if( j<Asoc )
    {
       np = new MemCell;
       np->Adress = Adresa_Bloc;
       np->Timer = 0;
       np->Plin = 1;
       switch(Algoritm_Inlocuire){
          case 1:
           {
               p = prim;
               while(p)
                {
                  p->Counter++;
                  p = p->next;
                }
               np->Counter = 0;
               break;
           }
          case 2:
           {
               p = prim;
               while(p)
                {
```

```
                p->Counter++;    return 1;
                p = p->next;
               }
            np->Counter = 0;
            break;
          }
        case 3:
          {
            break;
          }
      case 4:
          {
            p = prim;
         while(p)
               {
                 p->Counter = 0;
           p = p->next;
                }
            np->Counter =1;
            break;
          }
        }
      np->next = Level[nivel][i];
      Level[nivel][i] = np;
   }
 else/* j == Asoc */
  {
      //printf("\n****************\n");
```

```
switch(Algoritm_Inlocuire){
case 1:
 {
   p = prim;
   np = prim;
   while(p)
    {
        if( p->Counter > np->Counter) np = p;
        p->Counter++;
        p = p->next;
    }
   Adresa_Swap = np->Adress;
   np->Adress = Adresa_Bloc;
 np->Counter = 0;
   timer = np->Timer;
   np->Timer = 0;
   break;
 }
case 2:
 {
  p = prim;
  np = prim;
  while(p)
   {
       if( p->Counter < np->Counter) np = p;
       p->Counter++;
       p = p->next;
   }
  Adresa_Swap = np->Adress;
```

```
    np->Adress = Adresa_Bloc;
    np->Counter = 0;
    timer = np->Timer;
 np->Timer = 0;
    break;
  }
case 3:
  {
    j = rand() % Asoc;
    p = prim;
    while(j)
      {
          p = p->next;
          j--;
      }
    Adresa_Swap = p->Adress;
    p->Adress = Adresa_Bloc;
    timer = p->Timer;
    p->Timer = 0;
    break;
  }
case 4:
  {
    p = prim;
    while(p->Counter == 0) p = p->next;
    Adresa_Swap = p->Adress;
    p->Adress = Adresa_Bloc;
    timer = p->Timer;
```

```
            p->Timer = 0;
            break;
          }
        }/*switch*/
        //printf("%d\n",timer);
        Hye[nivel].Delay+= timer;
        Hye[nivel].Times++;
        AddTime(Hye[nivel].TimpAccWR);
        Scrie(nivel+1,Adresa_Swap);
      }/*else*/
  }/*if j<Asoc */
  else /* p!= null */
    AddTime(Hye[nivel].TimpAccWR);
  return 0;
}

int Run()
{
  FILE *file;
  char String[120];
  file = fopen(Trace,"r");
  if(!file)
    {
      printf("Trace-ul %s nu este gasit...\n");
      return -1;
    }
  int Mod,nivel;
  unsigned long Adress,Hit = 0,Miss = 0;
  do{
```

```
    fgets(String,120,file);
    if(sscanf(String,"%d %ld",&Mod,&Adress) == 2)
    {
        //printf("%d %ld\n",Mod,Adress);
      if( nivel = Cauta(Adress) )
          {
            if(Mod == 2)AddTime(Hye[nivel].TimpAccRD);
            else if(Mod == 0) AddTime(Hye[nivel].TimpAccFT);
            else if(Mod == 3) AddTime(Hye[nivel].TimpAccWR);
            Hit++;
            if( nivel > 1) Scrie(1,Adress);
          }
      else
      {
          Miss++;
          Scrie(1,Adress);
      }
    }
  }while(!feof(file));
  printf("Misses: %d\nHits: %d\nTotal: %d\n",Miss,Hit,Hit+Miss);
  //   WriteData("main.out");
  return 0;
}
# endif
```

5. LISTING-UL PROGRAMULUI: ***write.h***

```
# ifndef _WRITE_H__
```

```
# define _WRITE_H__
# include "dlr.h"
# include "data.h"
# include <stdio.h>
# include <stdlib.h>

int WriteData(char *out)
{
  FILE *file;
  int count,halt;
  long double Dlr,Dlr_S,Mttf;
  file = fopen(out,"w");
  printf("Nivele: %d\n\n",Nivele);
  halt = 0;
  for(count=1; count<=Nivele ; count++)
  {
    fprintf(file,"MTTF Level %d: %ld\n",count,Hye[count].Mttf);
    fprintf(file,"Delay Overall Level %d: %ld\n",count,Hye[count].Delay);
    if(Hye[count].Times != 0) fprintf(file,"Delay Average Level %d:
%ld\n",count,Hye[count].Delay = Hye[count].Delay/Hye[count].Times);
    else
      {
          fprintf(file,"Delay Average Level %d: Nedeterminat\n",count);
          halt = 1;
      }
    fprintf(file,"\n");
  }
  Mttf = compute_MTTF();
  fprintf(file,"MTTF Ierarhie: %f\n",Mttf);
```

```
  if(halt)
    {
      fprintf(file,"Imposibile de calculat DLR\n");
      return -1;
    }
  else
  {
    Dlr = compute_DLR();
    Dlr_S = compute_statistic_DLR();
    fprintf(file,"DLR ierarhie[%%]: %.16lf\n",Dlr);
    fprintf(file,"DLR statisic[%%]: %.16lf\n",Dlr_S);
  }
  fclose(file);
  return 0;
}
# endif
```

6. LISTING-UL PROGRAMULUI: ***dlr.h***

```
# ifndef __DLR_H_
# define __DLR_H_
# include "data.h"
# include <stdlib.h>
# include <math.h>

long double  compute_MTTF()
{
  long double MTTF = 0;
```

```
  for(int i=1;i<=Nivele;i++)
      MTTF+= (long double)1/Hye[i].Mttf;
  MTTF = 1/MTTF;
  return MTTF;
}

long double compute_DLR()
{
  long double DLR_L,DLR_L1;
  DLR_L1 = 0.00;
  for(int i=Nivele;i>1;i--)
    {
      if(Hye[i].Delay == 0) return -1;
      DLR_L = (Hye[i].Delay+DLR_L1*(Hye[i].Mttf-Hye[i]. Delay))/Hye[i].Mttf;
      DLR_L1 = DLR_L;
    }
  return DLR_L*100;
}

long double compute_statistic_DLR()
{
  long k;
  long double t,z,u,DRUF,DRUI,DLR;
  int ng,i;
  k = 0;
  n = 51754;
  DRUF = 0.00000000;

  do{
```

```
    ng = 1;
    DRUI = DRUF;
    z = 0.000;
     for(i=1;i<=Nivele;i++)
      {
           u = (double)rand()/0x7FFF;
           t = -Hye[i].Mttf*log(1-u);
           if(t <= Hye[i].Delay && ng == 1)
            {
              z=1.00;
              ng = 0;
              continue;
            }
     }
    DRUF = (DRUI*k+z)/(k+1);
    k++;
   }while(k<587124);
  DLR = DRUF*100;
  return DLR;
}
# endif
```

7. LISTING-UL PROGRAMULUI: ***dataloss.cpp***

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <time.h>
```

```
__int64 N,T[100],MTTF[100];
__int64 k,n;
FILE *in,*out;

int Incercare(char path[10],double &DLR)
{
	FILE *solution;
	__int64 i;
	bool ng;
	char path_in[20],path_out[20],path_sol[20];
	double eps = 0.00005,u,z,t;
	double DRUF,DRUI;
	sprintf(path_in,"%s.in",path);
	in = fopen(path_in,"r");
	if(!in)
	{
		printf("NU EXISTA FISIERUL DE INTRARE...\n");
		return 1;
	}

	sprintf(path_out,"%s.out",path);
	sprintf(path_sol,"%s.sol",path);
	out = fopen(path_out,"w");
	solution = fopen(path_sol,"w");

	fscanf(in,"LEVELS: %I64d \nINCERCARI: %I64d\n",&N,&n);
	fprintf(out,"LEVELS : %I64d \nINCERACRI : %I64d\n",N,n);
	fprintf(solution,"%7s|","Run No.");
```

```
        __int64 m;
        do{
                fscanf(in,"\nLEVEL %I64d:\n",&m);
                fscanf(in,"TIME: %I64d\nMTTF: %I64d\n",&T[m],&MTTF[m]);
                fprintf(out,"---------------------\n LEVEL %I64d \n TIME : %I64d \n
MTTF: %I64d\n",m,T[m],MTTF[m]);
        }while(!feof(in));

        for(i=1;i<=N;i++)
        {
                fprintf(solution,"timp%1I64d %15I64d|Succes|",i,T[i]);
                printf("%I64d\n",T[i]);
        }
        fprintf(solution,"DataLossRate[%%]\n");

        k = 0;
        DRUF = 0.00000000;
        do{
                ng = true;
                DRUI = DRUF;
                z = 0.000;
                fprintf(solution,"%7d|",k+1);
                for(i=1;i<=N;i++)
                {
                        u = (double)(rand())/0x7FFF;
                        t = -MTTF[i]*log(1-u);
                        fprintf(solution,"%21.0lf|",t);
                        if(t <= T[i] && ng == true)
                        {
```

```
                    fprintf(solution,"%6c|",'0');
                    z=1.00;
                    ng = false;
                    continue;
                }
                if(ng == true) fprintf(solution,"%6c|",'1');
                else fprintf(solution,"%6c|",'x');
            }
            DRUF = (DRUI*k+z)/(k+1);

            k++;
            fprintf(solution,"%.9lf\n",DRUF*100);
            DLR = DRUF*100;
        }while(k<n);
        fprintf(out,"\n      \n--------------------------------\nDLR      :      %.15lf
%%\n",DRUF*100);
        fclose(in);
        fclose(out);
        return 0;
}

double compute_DLR()
{
        double DLR_L,DLR_L1;
        DLR_L1 = 0.00;
        for(__int64 i=N;i>1;i--)
        {
            DLR_L = (T[i]+DLR_L1*(MTTF[i]-T[i]))/MTTF[i];
            DLR_L1 = DLR_L;
```

```
        }
        return DLR_L*100;
}

double compute_epsilon(double DLR_SIM,double DLR_TEO)
{
        return 100*fabsl(DLR_SIM-DLR_TEO)/DLR_TEO;
}
int main()
{
        FILE *rezultat;
        double DLRS,DLRT;
        int caz;
        char caz_path[20];
        printf("NUMARUL DE FISIERE data*.in : ");
        srand((unsigned)time( NULL ) );
        scanf("%d",&caz);
        rezultat = fopen("rezultat_final.out","w");
        fprintf(rezultat,"LEVELS|      MTTF LEVEL[s] |     DELAY LEVEL[s] |     DLR
model[%%]   |  DLR simulat[%%]   |     Epsilon[%%]    |\n");
        fprintf(rezultat,"-----------------------------------------------------------------------
---------------------------------------\n");
        while(caz)
        {
                sprintf(caz_path,"Data%d",caz);
                printf("INCERCARE %s\n",caz_path);
                if(Incercare(caz_path,DLRS))
                {
                        printf("INCERCAREA A ESUAT.....\n");
```

```
                continue;
            }
            DLRT = compute_DLR();
        fprintf(rezultat,"%6I64d|%19I64d|%19I64d|%19.8lf|%19.8lf|%19.8lf|\n",N
,MTTF[1],T[1],DLRT,DLRS,compute_epsilon(DLRS,DLRT));
            for(__int64 i=2;i<=N;i++)
                fprintf(rezultat,"       |%19I64d|%19I64d|                    |
|                |\n",MTTF[i],T[i]);
            fprintf(rezultat,"-------------------------------------------------------------
-----------------------------------------------\n");
            caz--;
        }
        fclose(rezultat);
        return 0;
}
```

Titluri recent publicate în colecția „TEZE DE DOCTORAT”
seria 10: Ştiința Calculatoarelor

1. **Norbert Neidenbach -** *Das Service-Management eines IT-Outsourcing-Projektes durch ITIL-Best-Practices, IT-Outsourcing kostenoptimiert planen und steuern, ISBN 978-973-625-660-8*, (2008);

2. **Edwin Hans Wolf -** *Das Geschäftsmodell (Business model) MDS (Managed Desktop Support) im IT-Outsourcing, Leistungserbringung im Rahmen des MDS-Geschäftsmodells, ISBN 978-973-625-661-5,* (2008);

3. **Adrian Zafiu –** *Minimizarea sistemelor decizionale multivalente deterministe şi nedeterministe, ISBN 978-973-625-678-3,* (2008);

4. **Daniel Iercan –** *Contributions to the Development of Real-Time Programming Techniques and Technologies, ISBN 978-973-625-719-3,* (2008);

5. **Laurenția Timar –** *Contribuții referitoare la configurarea optimală prin prisma performanță-fiabilitate a unor rețele de dispozitive de achiziția datelor cu aplicabilitate la excavatoarele cu cupe, ISBN 978-973-625-775-9,* (2008);

6. **Dan Cireşan –** *Recunoaşterea şirurilor numerice scrise de mână, ISBN 978-973-625-777-3,* (2008);

7. **Emanuel Țundrea –** *Contributions to the modelling and the use of software product lines, ISBN 978-973-625-793-3,* (2008);

8. **Alexandru Amăricăi –** *On the Design of Floating Point Units for Interval Arithmetic, ISBN 978-973-625-795-7,* (2008);

9. **Oana Boncalo –** *Simulation Based Reliability Assessment of Quantum Circuits, ISBN 978-973-625-796-4,* (2008);

10. **Cristian Ruican –** *Developing Automatic Synthesis Methodologies for Quantum Circuits Using Genetic Algorithms, ISBN 978-973-625-819-0,* (2009).

EDITURA POLITEHNICA