

## PREFATA

Aceasta lucrare se constituie într-o contribuție la instruirea și cercetarea arhitecturilor de procesare a informației pe baze moderne. În acest sens, sunt abordate probleme importante și actuale ale arhitecturilor cu paralelism la nivelul instrucțiunilor, atât din punct de vedere teoretic cât și practic. Caracterul pragmatic al lucrării este întărit de utilizarea unor simulatoare special dedicate aspectelor investigate, concepute în mare parte de către autorii lucrării sau de către colaboratorii lor. Aceste simulatoare sunt de altfel oferite cititorilor pe suportul unui CD atasat, în scopul unei autoinstruiri practice mai aprofundate. Precizăm că ele au fost folosite cu succes vreme de câțiva ani în procesul de instruire al studenților noștri din domeniul științei calculatoarelor.

Poate că cineva se întreaba dacă mai are rost să se abordeze în România această problemă a microarhitecturilor de calcul, azi când majoritatea celor cu preocupări în tehnologia informației (TI) sunt preocupați mai ales în domeniul dezvoltării de aplicații software dintre cele mai diverse. În mediile academice românești, preocupările par să fie preponderent legate de elaborarea unor sisteme inteligente tot mai sofisticate implementate, mai ales în software. Răspunsul nostru, adresat cu precădere profesioniștilor în devenire, este unul afirmativ. Iar argumentul esențial ar fi: nu doar din punct de vedere formativ, dar, mai ales, din punct de vedere aplicativ este necesar ca specialistul în TI să înțeleagă și să acopere “*prăpastia semantică*” actuală între tehnologie, văzută ca “*temelie*” a aplicațiilor și respectiv ceea ce se așteaptă de la ea, prin chiar prisma acestor aplicații inteligente [Bar02]. Putine sunt eforturile care se fac în această direcție, nu numai la noi ci și aiurea. Se preferă acestei viziuni integratoare (tehnologii, arhitecturi, aplicații) acea specializare strictă, poate eficientă pe termen scurt, dar aducătoare de crize cognitive și schimbări paradigmatică forțate, pe termen mai lung.

Ideea de esență care ne-a calauzit în elaborarea lucrării a fost aceea că domeniul arhitecturii microprocesoarelor nu este doar descriptiv (*read*) ci și constructiv (*write*). Altfel spus, investigația personală și creația sunt posibile, utilizând tehnici precum cele prezentate aici. Cartea calauzește cititorul interesat pe drumul de la o arhitectură puternic parametrizată la o anumită instanță optimă a acesteia, ce urmează să fie implementată în hardware. Alegerea acestei instanțe constituie o aventură formativă și aplicativă, care printre multe altele, trebuie să țină cont și de cerințele potențialului sau utilizator.

Pe scurt, lucrarea de față este structurată astfel:

În capitolul 1 se aprofundează arhitectura familiei de microprocesoare scalare RISC MIPS R2000/R3000 prin intermediul unui simulator numit SPIM, conceput pentru uz didactic de către James R. Larus de la Universitatea

din Wisconsin, SUA. Am optat pentru familia MIPS datorita faptului ca, pe lânga succesul sau comercial, constituie unele dintre cele mai inteligibile si elaborate microprocesoare RISC.

Capitolul 2 se focalizeaza pe studiul celebrului microprocesor virtual numit DLX, elaborat de catre profesorii John Hennessy (Universitatea Stanford, SUA) si David Patterson (Universitatea Berkeley, SUA), cercetatori reputati si autori ai faimoasei carti “*Computer Architecture: A Quantitative Approach*”, care a revolutionat modul de instruire în arhitecturile de calcul în întreaga lume. Acest microprocesor DLX este investigat prin intermediul unui excelent simulator de tip “*freeware*”, înzestrat cu facilitati didactice deosebite si cu o grafica arhitecturala atragatoare. Important, simulatorul ruleaza sub sistemul Windows, ca de altfel toate celelalte prezente în lucrare.

Capitolul 3 aprofundeaza înțelegerea tehnicilor de procesare pipeline si respectiv celor cu executie “*out-of-order*” a instructiunilor în cadrul unei masini superscalare tipice, înzestrate cu toate caracteristicile microarhitecturale aferente. Demersul este sustinut aici printr-un simulator numit SATSIM (dezvoltat de catre colectivul de cercetare în arhitecturi de la Georgia Institute of Technology, SUA), având facilitati grafice extrem de sugestive, în scopul înțelegerii tuturor subtilitatilor legate de circulatia informatiei prin structura la nivel de ciclu masina.

Urmatoarele doua capitole (4, 5) sunt destinate simulării functionarii memoriilor cache integrate în cadrul unor arhitecturi superscalare parametrizabile. Mentionam ca aceste simulatoare, ca de altfel toate cele care urmeaza a fi prezentate în carte, au fost dezvoltate în cadrul unui grup de cercetare în arhitecturi avansate, în cadrul Catedrei de Calculatoare a Universitatii “*Lucian Blaga*” din Sibiu, pe durata a mai bine de 7 ani. Revenind la simulatoarele de cache-uri, acestea abordeaza toate arhitecturile importante de asemenea memorii, incluzând si unele variante mai exotice bazate pe algoritmi de încarcare/evacuare euristici si predictivi (“*Selective Victim Cache*”).

Capitolele 6 si 7 abordeaza o problema deosebit de importanta în cadrul microprocesoarelor superscalare actuale si viitoare, anume aceea a predictiei branch-urilor si respectiv executiei speculative a instructiunilor. Prin intermediul unor simulatoare special dezvoltate sunt investigate cele mai cunoscute scheme de predictie adaptiva pe 2 nivele precum si un predictor neural, dezvoltat în premiera acum cca. 5 ani de catre unul din autorii acestei carti.

Urmatoarele doua capitole (8, 9) prezinta o foarte utila colectie de instrumente software numita generic SimpleScalar, pusa la dispozitie tuturor celor interesati în zona arhitecturilor moderne de calcul. Setul SimpleScalar a fost dezvoltat progresiv, pe baza de voluntariat, de catre diversi cercetatori

universitari. Setul cuprinde compilatoare, asamblatoare, link-editoare, depanatoare de cod, simulatoare si instrumente de vizualizare aferente unei arhitecturi superscalare tipice. Prin intermediul acestor simulatoare s-a încercat “*standardizarea*” metodologiei de simulare a microarhitecturilor, nu numai la nivelul benchmark-urilor (SPEC) ci, mai ales, la nivelul caracteristicilor microarhitecturale. Mai mult, autorii acestei carti prezinta si o metodologie generala de extindere a acestui set cu noi module de simulare, ceea ce este extrem de util pentru cei ce vor sa realizeze experiente noi, inedite, în acest domeniu fascinant al microprocesoarelor avansate. În acest sens, recomandam cititorului sa încerce sa-si dezvolte un simulator propriu, destinat unei probleme bine definite.

Capitolul 10 se concentreaza pe simularea unor microarhitecturi novatoare de tip predictiv si speculativ, dezvoltate relativ recent si cunoscute sub denumirea de procesoare cu predictie a valorilor instructiunilor. Succesul predictiei valorilor si, pe aceasta baza, a executiei speculative, se bazeaza pe caracteristica de “*vecinatate a valorilor instructiunilor*” (value locality) introdusa de Shen si Lipasti în 1996 [Lip96]. Întregul esafodaj al simulatorului este construit pe baza setului de simulatoare SimpleScalar si a benchmark-urilor standardizate SPEC, anterior prezentate.

Capitolele 11 si 12 sunt destinate propunerii si respectiv rezolvarii de probleme nu doar interesante si instructive ci, credem noi, chiar fecunde. Consideram ca solutionarea sistematica si graduala de probleme diverse poate determina aprofundarea serioasa a unei asemenea discipline tehnice de vârf, prin excelenta aplicativa, cum este arhitectura microprocesoarelor.

În fine, ultimul capitol (13) prezinta continutul CD-ului atasat cartii. Lucrarea se încheie cu un util set de anexe, legate de capitolele anterior prezentate si cu o bibliografie selectiva.

Autorii doresc sa multumeasca pe aceasta cale, tuturor celor care i-au ajutat, direct sau indirect, la scrierea acestei carti: fostilor lor profesori, actualilor colegi de breasla precum si unor fosti studenti. Gândurile noastre cele mai alese sunt îndreptate si asupra Editurii Matrix Rom care s-a implicat cu generozitate în editarea acestei lucrari. Desi cuvintele noastre sunt prea sarace, în final, adresam profunda recunostinta familiilor noastre, pentru tot.

Asteptam cu interes, sugestiile si criticile cititorilor, pentru care le multumim anticipat, pe adresele noastre de mail: [adrian.florea@mail.ulbsibiu.ro](mailto:adrian.florea@mail.ulbsibiu.ro), [lucian.vintan@mail.ulbsibiu.ro](mailto:lucian.vintan@mail.ulbsibiu.ro).

Sibiu,  
februarie 2003



## 0. SIMULATORUL. CE ESTE? ESTE NECESARA SIMULAREA? METODOLOGII DE SIMULARE.

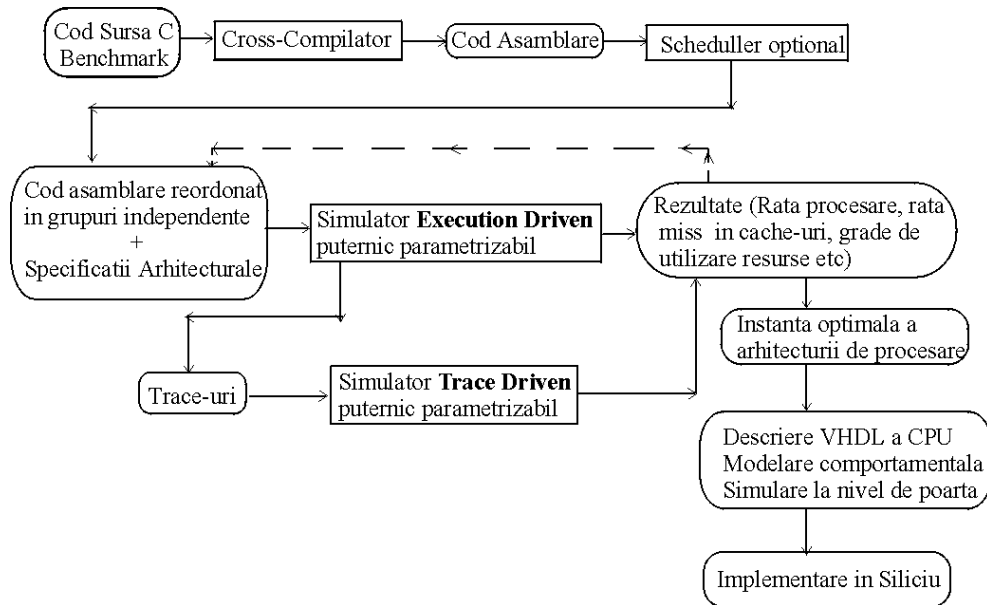
---

- ❑ Istoria procesoarelor contrapune doua paradigme pentru cresterea performantei, bazate pe software si respectiv pe hardware.
- ❑ În procesul de proiectare al procesoarelor, aferent generatiilor viitoare, accentul principal nu se mai pune pe implementarea hardware, ci pe proiectarea arhitecturii în strânsa legatura cu aplicatiile potentiale. Se porneste de la o arhitectura de baza (generica), puternic parametrizata, care este modificata si îmbunatatita dinamic, prin simulari laborioase pe programe de test (benchmark-uri) reprezentative.
- ❑ **Simulator dedicat unei arhitecturi de calcul** – instrument software (aplicatie/program) utilizat în exploatarea / cercetarea / si îmbunatatirea performantelor unei microarhitecturi. De obicei functionarea microarhitecturii se simuleaza la nivel de ciclu masina permitând vizualizarea tuturor resurselor hardware la finele fiecarui ciclu.
- ❑ Metodologia de simulare poate fi de doua tipuri:
  - **Execution driven simulation**
    - ⇒ caracterizata de cunoasterea în fiecare moment (ciclu "pipe") a continutului resurselor arhitecturale (registri, locatii de memorie, unitati functionale).
    - ⇒ Simularea se face foarte detaliat, la nivel de ciclu de executie al procesorului.
    - ⇒ **Outputs:** - continutul resurselor, gradul de încarcare al acestora, rate de procesare, de hit etc.

- Fișiere **trace** care contin toate instructiunile masina ale programelor de test în ordinea în care se executa.

➤ **Trace driven simulation**

⇒ analizeaza secvential toate instructiunile din trace-urile generate de simulatorul bazat pe execution driven, cu scopul de a determina instanta optima a arhitecturii - *procesorul ce urmeaza a fi implementat în hardware*. TDS se preteaza la *simularea cache-urilor* de date si instructiuni, mecanismelor de memorie virtuala etc., datorita faptului ca ofera pattern-uri reale de adrese, în urma executiei unor programe reprezentative.



**Figura 0.1.** Etapele de simulare, comparare si determinare efectiva ale unei arhitecturi optime, pornind de la sursa HLL (High Level Languages) a programelor de test si pâna la implementarea hardware a arhitecturii

# 1. ARHITECTURA MICROPROCESOARELOR

## *MIPS R2000/R3000*

---

### 1.1. SCOPUL LUCRARIII

Lucrarea de fata urmareste familiarizarea cu arhitectura procesorului scalar RISC MIPS R2000, un exemplu tipic de masina load/store. Pentru început se vor studia modurile de adresare, sintaxa asamblor, setul de instructiuni, partajarea memoriei. Lucrarea continua cu descrierea detaliata a simulatorului SPIM/SAL, scris pentru uz didactic de catre *James Larus* în cadrul Facultatii de Calculatoare a Universitatii din Wisconsin, simulator care ruleaza programe executabile sau scrise în limbaj de asamblare MIPS R2000. În finalul lucrarii sunt explicate doua aplicatii de complexitate medie si propuse alte trei probleme pentru rezolvare având ca scop înțelegerea aspectelor arhitecturale specifice procesoarelor RISC si formarea deprinderilor practice privind programarea în asamblare MIPS.

### 1.2. MEMENTO TEORETIC

Obiectivul declarat al procesorului MIPS este înalta performanta obtinuta prin pipelining, o implementare hardware usoara si compatibilitatea cu compilatoarele optimizate. Aceste scopuri conduc la instructiuni simple, moduri simple de adresare, formate de instructiuni de lungime fixa si multi registrii.

Procesorul contine 32 de registri de uz general si un set de instructiuni bine proiectat care-l face o tinta favorabila pentru generarea codului într-un compilator. Pe lângă unitatea de procesare pentru întregi, procesorul MIPS

contine si o colectie de coprocesoare care îndeplinesc sarcini auxiliare sau opereaza asupra altor tipuri de date, cum ar fi numere în flotant.

Complexitatea programelor se datoreaza în primul rând instructiunilor care implica întârzieri: branch-uri (instructiuni de ramificatie) si load-uri (instructiuni cu referire la memorie) precum si restrictiilor introduse de modulele de adresare. Un branch necesita doi cicli de întârziere, necesari stabilirii conditiei de salt si calculului de adresa tinta. În al doilea ciclu, instructiunea imediat urmatoare branch-ului se executa. Aceasta instructiune poate fi una utila care s-ar executa în mod normal înainte de branch sau poate fi un simplu *nop* (no operation). Similar, un load întârziat necesita doi cicli, astfel ca instructiunea imediat urmatoare load-ului nu va putea folosi valoarea din memorie.

MIPS ascunde complexitatea amintita mai sus, având implementat propriul asamblor (o masina virtuala). Acest calculator virtual nu are load-uri si branch-uri întârziate, în plus, are un set mai bogat de instructiuni decât hardware-ul actual. Asamblorul rearanjeaza instructiunile pentru a umple “delay slot-ul” - întârzierea pe care în mod normal ar fi introdus-o instructiunile întârziate (branch si load). De asemenea, sunt simulate macroinstructiuni sau *pseudoinstructiuni*, prin generarea unor secvente scurte de instructiuni actuale (de baza).

SPIM/SAL este o portare pe WIN32S a SPIM, un simulator scris pentru uz didactic în Facultatea de Calculatoare a Universitatii din Wisconsin. WIN32S este o extensie pe 32 de biti a sistemului Windows 3.1. Interfata programabila cu aplicatia a WIN32S este un subset al WIN32, un API suportat de sistemul de operare Windows NT. Aplicatii ce folosesc WIN32S API, cum sunt SPIM/SAL, sunt compilate într-un format executabil numit *executabil portabil* (PE) care ruleaza fie pe Microsoft Windows 3.1. cu extensie WIN32S fie pe Microsoft Windows NT.

SPIM apare în doua versiuni pentru sistemul de operare Linux: una simpla *spim*. (care ruleaza în mediu de comanda) si o versiune superioara *xspim* care ruleaza pe un sistem X – Window (mult mai usor de învatat si folosit deoarece comenzile sunt întotdeauna vizibile pe ecran si afiseaza permanent continutul registrilor). Atât optiunile din linia de comanda cat si cele de interfatare cu terminalul aferente versiunilor de SPIM pentru sistemul de operare Linux [VinFlor99] au corespondent în optiunile de meniu ale simulatorului SPIMSAL.

SPIM S20 este un simulator care ruleaza programe pentru calculatoare RISC cu procesor MIPS R2000 / R3000. SPIM poate citi si executa imediat fisiere scrise în limbaj de asamblare sau executabile MIPS. El contine un debugger si asigura câteva servicii specifice sistemelor de operare. SPIM este mult mai încet decât un calculator real (aproximativ 100 de ori). Totusi,



costul scazut, larga aplicabilitate si facilitatile didactice nu pot fi egalate de microprocesorul real.

Se pune deci întrebarea: *“De ce folosim un simulator când multi utilizatori au statii de lucru ce contin cip-uri MIPS care sunt semnificativ mai rapide decât SPIM ?”*

Un motiv ar fi ca aceste statii nu sunt universal folosite. Un altul ar fi progresul spre calculatoare noi si rapide, făcând aceste masini demodate. Tendinta curenta este de a face calculatoarele mai rapide prin executia concurenta a mai multor instructiuni. Aceasta face arhitecturile mai dificil de înțeles si programat. În plus, simulatoarele pot asigura un mediu mai bun de programare decât o masina existenta, deoarece ele pot detecta mai multe erori si pot asigura mai multe caracteristici decât un calculator actual. Deci, simulatoarele sunt unelte folosite în studierea calculatoarelor si programelor care ruleaza pe ele. Deoarece sunt implementate software si nu hardware, simulatoarele pot fi usor modificate adaugând instructiuni noi, construind sisteme noi cum ar fi cele multiprocesor sau se pot colecta si analiza date.

Implicit, SPIM simuleaza masina virtuala cu un bogat set de instructiuni. Totusi, el poate simula si hardware-ul simplu. În continuare vom descrie masina virtuala si vom mentiona pe scurt caracteristicile care nu apartin hardware-ului existent.

Desi SPIM este un simulator fidel calculatoarelor MIPS, unele lucruri nu sunt identice cu cele ale unui calculator adevarat. Cea mai evidenta diferenta consta în sincronizarea instructiunilor si sistemul de memorie. SPIM nu simuleaza memoria cache sau latentă memoriei, si nici nu reflecta exact întârzierile datorate operatiilor în virgula mobila sau instructiunilor de înmultire si împartire.

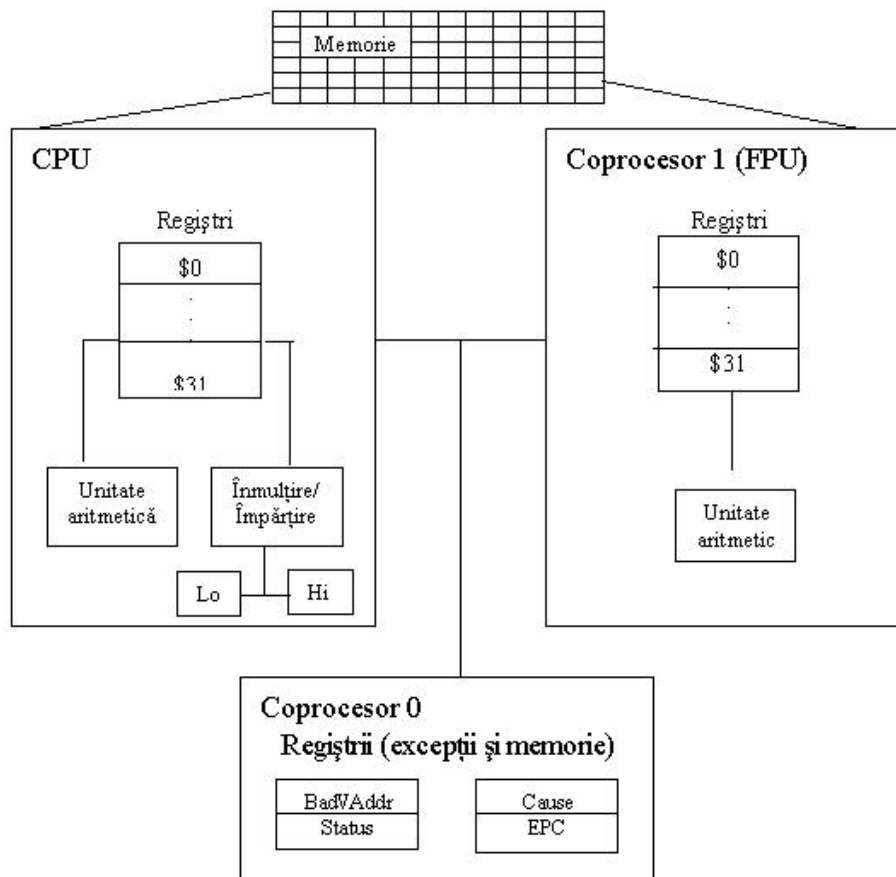
O caracteristica pe care o regasim si la masinile reale este aceea ca o pseudoinstructiune expandeaza în mai multe instructiuni masina. Când rulam pas cu pas programul sau examinam memoria, instructiunile pe care le vedem sunt diferite de cele ale programului sursa. Corespondenta dintre cele doua seturi de instructiuni este simpla întrucât SPIM nu reorganizeaza instructiunile pentru a umple “delay slot-ul” [VinFlor99].

Referitor la ordinea octetilor dintr-un cuvânt, SPIM opereaza atât cu ordinea “*big-endian*” cât si “*little-endian*”. Ordinea octetilor în cadru SPIM este identica cu ordinea de pe masina pe care ruleaza simulatorul. De exemplu, pe o statie DEC 3100, Intel Pentium II, SPIM foloseste “*little-endian*”, în timp ce pe un MacIntosh, HP Bobcat sau Sun SPARC, ordinea folosita este “*big-endian*”.

### 1.3. DESFASURAREA LUCRARIII

#### 1.3.1. SCHEMA BLOC SI REGISTRII PROCESORULUI MIPS R2000

Unitatea centrală de procesare a MIPS conține 32 de registre generali numerotați de la 0 la 31. Registrul  $n$  este desemnat ca  $\$n$ . Registrul  $\$0$  este întotdeauna cablat la valoarea 0. MIPS a stabilit un set de convenții despre cum ar trebui folosiți registrele. Aceste convenții sunt principii calauzitoare, care nu sunt forțate de către hardware. Totuși, un program care violează aceste principii nu va funcționa perfect cu un alt software.



**Figura 1.1.** Schema bloc a procesorului MIPS R2000

Numele Registrilor	Numarul Registrilor	Utilizarea Registrilor
Zero	0	Constanta 0.
at, k <sub>0</sub> , k <sub>1</sub>	1, 26, 27	Rezervati sistemului de operare; nu trebuie folositi de catre programele utilizator sau compilatoare.
v <sub>0</sub> , v <sub>1</sub>	2, 3	Pastreaza rezultatele returnate de functii în urma apelurilor sistem sau (în v <sub>0</sub> ) - codul apelurilor sistem.
a <sub>0</sub> ÷ a <sub>3</sub>	4 ÷ 7	Transmiterea primelor patru argumente rutinei apelate (restul argumentelor sunt puse pe stiva).
t <sub>0</sub> ÷ t <sub>9</sub>	8 ÷ 15, 24, 25	Registrarii de salvare ai rutinei apelante, memoreaza temporar valori care nu trebuie pastrate de-a lungul apelului.
s <sub>0</sub> ÷ s <sub>7</sub>	16 ÷ 23	Registrarii de salvare ai rutinei apelate, memoreaza valorile ce trebuie pastrate la iesirea din proceduri.
gp	28	Pointer global care indica spre mijlocul unui bloc de 64K octeti în segmentul de date statice, ce pastreaza constante si variabile globale.
sp	29	Indicator de stiva, care pointeaza pe prima locatie libera din stiva.
fp	30	Pointer de cadru
ra	31	Pastreaza adresa de revenire dintr-o procedura dupa apelul instructiunii jal.

Tabelul 1.1.

### Registrarii MIPS si conventii de utilizare

În plus, coprocesorul 0 contine registrarii care sunt folositi pentru tratarea exceptiilor, pe care îi prezentam în tabelul 1.2.

Numele Registrului	Număr	Utilizare
BadVAddr	8	Adresa de memorie la care apare adresa virtuala care a produs exceptia.
Status	12	Contine bitii de validare a întreruperii
Cause	13	Tipul exceptiei
EPC	14	Adresa instructiunii care a cauzat exceptia

Tabelul 1.2.

### Registrarii coprocesorului 0

Acesti registrarii sunt o parte din setul de registrarii ai coprocesorului 0 si pot fi accesati prin instructiuni de genul: *lcw0*, *mfc0*, *mtc0* si *swc0*.

### 1.3.2. ORDINEA OCTETILOR SI MODURILE DE ADRESARE

Procesoarele pot numerota octetii din interiorul unui cuvânt de 32 de biti astfel încât octetul cu numarul cel mai mic este fie cel mai din stânga fie cel mai din dreapta. Conventia folosita de catre o masina se numeste *ordinea octetilor*. Procesoarele MIPS pot opera fie cu ordinea octetilor *big-endian*

Byte #			
0	1	2	3

sau

*little-endian*

Byte #			
3	2	1	0

Instructiunile de calcul opereaza doar cu valorile din registre. O masina simpla asigura doar un singur mod de adresare a memoriei, si anume modul indexat: **c(rx)**, care foloseste suma imediata dintre constanta **c** si registrul **rx** ca o adresa. Masinile virtuale asigura urmatoarele moduri de adresare pentru instructiunile load/store (vezi tabelul 1.3).

FORMAT	CALCULUL ADRESEI
(Registru)	Continutul unui registru
Imm	Valoare imediata
Imm(registru)	Valoare imediata + Continutul registrului
Symbol	Adresa simbolului
Symbol imm	Adresa simbolului + sau – Valoarea imediata
Symbol imm(registru)	Adresa simbolului + sau - (Valoarea imediata + Continutul registrului)

*Tabelul 1.3.*

#### Modurile de Adresare la MIPS R2000/R3000

Majoritatea instructiunilor load si store opereaza doar cu date aliniate. O data este aliniata daca adresa sa de memorie este un multiplu a marimii sale în octeti. De aceea, un obiect de jumătate de cuvânt trebuie sa fie memorat la adrese pare iar obiecte pe un cuvânt trebuie memorate la adrese care sunt multiplu de 4. Totusi, MIPS asigura unele instructiuni pentru manipularea datelor nealiniate (*lwl*, *lwr*, *swl* si *swr*).

### 1.3.3. SINTAXA ASAMBLOR

Limbajul de asamblare este un limbaj de programare, principala sa deosebire fata de limbajele de nivel înalt, cum sunt BASIC, PASCAL si C/C++, fiind aceea ca el ofera doar câteva tipuri simple de comenzi si date. Limbajele de asamblare nu specifica tipul valorilor pastrate în variabile, lasând programatorul sa aplice asupra lor operatiile potrivite.

Comentariile în fisiere de asamblare încep cu simbolul `#`. Orice urmeaza acestui caracter pâna la sfârșitul liniei este ignorat. Identificatorii sunt o secventa de caractere alfanumerice, linie de subliniere (underbars), si punct, si sa nu înceapa cu un numar. Opcode-ul instructiunilor sunt cuvinte rezervate care nu pot fi folosite ca identificatori. Etichetele sunt declarate prin asezarea lor la începutul unei linii urmate de caracterul `:`.

*Exemplu:*

```

                                .data
item:    .word 1
                                .text
                                .globl main    # Trebuie sa fie global
main:    lw $t0, item

```

Numerele sunt implicit în baza 10. Daca sunt precedate de caracterul `0x`, ele sunt interpretate în sistemul de numeratie hexazecimal. Deci, 256 si `0x100` semnifica aceeasi valoare.

Sirurile sunt încadrate de ghilimele `""`. Caracterele speciale din siruri urmeaza conventia limbajului C. Astfel:

- linie noua     `\n`
- tab            `\t`
- ghilimele     `\"`

Prezentam, în continuare, câteva din directivele de asamblare ale MIPS.

**.align n** - aliniaza datele urmatoare într-un domeniu de  $2^n$  octeti. De exemplu,

*.align 2* aliniaza valorile urmatoare într-un domeniu limitat de 1 cuvânt (word).

*.align 0* dezactiveaza automat alinierea datorata directivelor *.half*, *.word*, *.float* si *.double* pâna la urmatoarea directiva *.data* sau *.kdata*.

**.asciiz str** - memoreaza sirul *str* în memorie împreuna cu terminatorul *null*.

**.byte b<sub>1</sub>, ..., b<sub>n</sub>** - memoreaza n valori în octeti succesivi în memorie.

- .data <addr>** - numere (articole) ulterioare sunt memorate în segmentul de date. Daca argumentul optional *addr* este prezent, valorile ulterioare sunt memorate începând de la adresa *addr*.
- .double  $d_1, \dots, d_n$**  - memoreaza cele  $n$  numere în flotant dubla precizie în locatii succesive de memorie.
- .float  $f_1, \dots, f_n$**  - memoreaza cele  $n$  numere în flotant simpla precizie în locatii succesive de memorie.
- .globl sym** - declara simbolul *sym* ca global si poate fi referit din alte fisiere.
- .half  $h_1, \dots, h_n$**  - memoreaza  $n$  locatii de 16 biti în semicuvinte succesive în memorie.
- .kdata <addr>** - date succesive sunt memorate în segmentul de date al *kernel*-ului. Daca argumentul optional *addr* este prezent, datele ulterioare sunt memorate începând cu adresa *addr*.
- .ktext <addr>** - date succesive sunt memorate în segmentul de text al *kernel*-ului. Daca argumentul optional *addr* este prezent, datele ulterioare sunt memorate începând cu adresa *addr*.
- .set noat .set at** - a doua directiva revalida avertismentele. Întrucât instructiunile false expandeaza în cod care foloseste registrul \$1, programatorii trebuie sa fie foarte atenti când parasesc valorile din registru.
- .space  $n$**  - alocă  $n$  octeti de spatiu în segmentul curent.
- .text <addr>** - valori ulterioare sunt puse în segmentul de text. Daca argumentul optional *addr* este prezent, atunci valorile sunt memorate începând cu aceasta adresa.
- .word  $w_1, \dots, w_n$**  - memoreaza cuvinte de 32 de biti în locatii succesive de memorie.

#### 1.3.4. FORMATUL INSTRUCIUNILOR SI SETUL DE INSTRUCIUNI AL PROCESORULUI MIPS R2000

Prezentam în continuare, atât instructiuni implementate în hardware-ul real al MIPS cât si pseudoinstructiuni oferite de asamblorul MIPS. Cele doua tipuri de instructiuni se disting usor. Instructiunile reale sunt descrise de câmpuri reprezentate binar. De exemplu:

$add\ R_d, R_s,$ $R_t$	<table><tr><td>0</td><td><math>R_s</math></td><td><math>R_t</math></td><td><math>R_d</math></td><td>0</td><td>0x20</td></tr><tr><td>6</td><td>5</td><td>5</td><td>5</td><td>5</td><td>6</td></tr></table>	0	$R_s$	$R_t$	$R_d$	0	0x20	6	5	5	5	5	6	adunare cu depasire (overflow)
0	$R_s$	$R_t$	$R_d$	0	0x20									
6	5	5	5	5	6									

Instructiunea *add* consta din 6 câmpuri. Dimensiunea fiecarui câmp în biti este indicata de numerele de sub câmp. Aceasta instructiune începe cu 6 biti de 0. Urmeaza codificarea registrului sursa pe 5 biti, registru tampon si registrul destinatie tot pe 5 biti. Un alt câmp general este  $Imm_{16}$ , care este o constanta imediata pe 16 biti. Formatul de mai sus se numeste R-tip si este o particularizare a formatului general urmator:

op	$R_s$	$R_t$	$R_d$	shamt	funct
6	5	5	5	5	6

Câmpurile au urmatoarea semnificatie:

- ☐ *Op* – operatia instructiunii
- ☐ *R<sub>s</sub>* – primul registru sursa, operand al instructiunii
- ☐ *R<sub>t</sub>* – al doilea registru sursa
- ☐ *R<sub>d</sub>* – registrul destinatie; pastreaza rezultatul operatiei
- ☐ *Shamt* – cantitate de deplasare (shiftare)
- ☐ *Func* - functia prin care se selecteaza varianta operatiei din câmpul *op*.

Un al doilea tip de format de instructiuni este I-tip si este folosit de catre instructiunile de transfer. Câmpurile formatului sunt urmatoarele:

op	$R_s$	$R_t$	Address
6	5	5	16

Desi multiplele formate complica hardware-ul, prin pastrarea unor formate similare se poate reduce aceasta complexitate. De exemplu, primele trei câmpuri ale formatelor prezentate mai sus (R-tip si I-tip) sunt identice, iar dimensiunea celui de-al patru-lea câmp al formatului I-tip este egal cu suma ultimelor trei câmpuri ale R-tip. Formatele se disting prin valorile din primul câmp.

Un al treilea format pentru instructiunile MIPS se numeste J-tip, care consta dintr-un câmp operatie pe 6 biti si unul de adresa de 26 de biti.

op	Address
6	26

Pseudoinstructiunile urmeaza aproximativ aceeasi conventie, dar omit informatia de codificare a instructiunilor. Astfel,  $S_{rc2}$  este fie registru fie constanta imediata. Asamblorul va transforma forma generala a unei instructiuni (Ex. *add*) în una intermediara (*addi*) daca cel de-al doilea parametru este o constanta imediata.

## 1.3.4.1. INSTRUCȚIUNI ARITMETICE SI LOGICE

*abs*  $R_{dest}, R_{src}$ 

Valoare absoluta

Pune valoarea absoluta a întregului aflat în registrul sursa în registrul destinatie.

*add*  $R_{dest}, R_{src1}, S_{rc2}$ 

Adunare cu depasire (overflow)

*addi*  $R_{dest}, R_{src1}, Imm$ 

Adunare imediata cu depasire

*addu*  $R_{dest}, R_{src1}, S_{rc2}$ 

Adunare fara depasire

*addiu*  $R_{dest}, R_{src1}, Imm$ 

Adunare imediata fara depasire

Calculeaza suma întregilor din registrii  $R_{src1}$  si  $S_{rc2}$  sau  $Imm$ , si depune rezultatul în  $R_{dest}$ .

*and*  $R_{dest}, R_{src1}, S_{rc2}$ 

SI logic

*andi*  $R_{dest}, R_{src1}, Imm$ 

SI logic cu o constanta imediata

Pune în registrul  $R_{dest}$  rezultatul obtinut în urma operatiei SI logic dintre întregii din registrii  $R_{src1}$  si  $S_{rc2}$ .

*and*  $R_{dest}, R_{src1}, S_{rc2}$ 

SI logic

*andi*  $R_{dest}, R_{src1}, Imm$ 

SI logic cu o constanta imediata

Pune în registrul  $R_{dest}$  rezultatul obtinut în urma operatiei SI logic dintre întregii din registrii  $R_{src1}$  si  $S_{rc2}$ .

*div*  $R_{src1}, RS_{rc2}$ 

Împartire cu depasire

*divu*  $R_{src1}, RS_{rc2}$ 

Împartire fara depasire

Împarte continutul celor doua registre. Câțul este depus în registrul LO si restul în registrul HI. De remarcat ca, daca un operand este negativ, restul este nespecificat într-o arhitectura MIPS si depinde de conventia masinii pe care este rulat programul.

*div*  $R_{dest}, R_{src1}, S_{rc2}$ 

Împartire cu depasire

*divu*  $R_{dest}, R_{src1}, S_{rc2}$ 

Împartire fara depasire

Pune câțul împartirii întregi dintre  $R_{src1}$  si  $S_{rc2}$  în registrul destinatie  $R_{dest}$ .



<i>mul</i> $R_{dest}, R_{src1}, S_{rc2}$	Înmultire fara depasire
<i>mulou</i> $R_{dest}, R_{src1}, S_{rc2}$	Înmultire fara semn cu depasire
<i>mulo</i> $R_{dest}, R_{src1}, S_{rc2}$	Înmultire cu depasire

Pune în registrul  $R_{dest}$  rezultatul obtinut în urma înmulțirii dintre întregii din registrii  $R_{src1}$  și  $S_{rc2}$ .

<i>mult</i> $R_{src1}, R_{src2}$	Înmultire
<i>multu</i> $R_{src1}, R_{src2}$	Înmultire fara semn

Înmulteste continutul celor doua registre. Partea mai puțin semnificativa a rezultatului e depusa în registrul LO, iar cea semnificativa în registrul HI.

<i>not</i> $R_{dest}, R_{source}$	NOT
-----------------------------------	-----

Pune negatia logica a întregului din registrul  $R_{source}$  în registrul  $R_{dest}$ .

<i>or</i> $R_{dest}, R_{src1}, S_{rc2}$	OR
<i>ori</i> $R_{dest}, R_{src1}, Imm$	OR imediat

Rezultatul operatiei SAU logic a întregilor din registrele  $R_{src1}$  și  $S_{rc2}$  (sau  $Imm$ ) este depus în registrul  $R_{dest}$ .

<i>rem</i> $R_{dest}, R_{src1}, S_{rc2}$	Rest
<i>remu</i> $R_{dest}, R_{src1}, S_{rc2}$	Rest fara semn

Pune restul împartirii dintre valoarea întreaga aflata în registrul  $R_{src1}$  la cea din  $S_{rc2}$ , în registrul  $R_{dest}$ . Daca un operand este negativ, restul este nespecificat pentru o arhitectura MIPS și depinde de conventiile masinii pe care ruleaza programul.

<i>rol</i> $R_{dest}, R_{src1}, S_{rc2}$	Roteste la stânga
<i>rор</i> $R_{dest}, R_{src1}, S_{rc2}$	Roteste la dreapta

Roteste la stânga sau dreapta continutul registrului  $R_{src1}$ , cu distanta indicata de registrul  $S_{rc2}$  și pune rezultatul în registrul destinatie.

<i>sll</i> $R_{dest}, R_{src1}, S_{rc2}$	Deplaseaza logic la stânga variabil
<i>sra</i> $R_{dest}, R_{src1}, S_{rc2}$	Deplaseaza aritmetic la dreapta

*srl*  $R_{dest}, R_{src1}, S_{rc2}$                       Deplaseaza logic la dreapta variabil

Deplaseaza la stânga sau dreapta continutul registrului  $R_{src1}$ , cu distanta indicata de registrul  $S_{rc2}$  si pune rezultatul în registrul destinatie.

*sub*  $R_{dest}, R_{src1}, S_{rc2}$                       Scadere cu depasire  
*subu*  $R_{dest}, R_{src1}, S_{rc2}$                       Scadere fara depasire

Diferenta dintre întregii din registrele  $R_{src1}$  si  $S_{rc2}$  este depusa în registrul destinatie.

*xor*  $R_{dest}, R_{src1}, S_{rc2}$                       XOR  
*xori*  $R_{dest}, R_{src1}, Imm$                       XOR imediat

Calculeaza XOR logic dintre întregii din registrii  $R_{src1}$  si  $S_{rc2}$ (sau  $Imm$ ), rezultatul fiind depus în  $R_{dest}$ .

#### 1.3.4.2. INSTRUCȚIUNI CU REFERIRE LA MEMORIE SI MANIPULARE A CONSTANTELOR

*la*  $R_{dest}, address$                       Încarcare adresa

Încarca adresa calculata, valoarea adresei nu continutul locatiei, în registrul  $R_{dest}$ .

*lb*  $R_{dest}, address$                       Încarcare octet  
*lbu*  $R_{dest}, address$                       Încarcare octet fara semn

Încarca octetul de la adresa *address* în registrul destinatie (si extensia de semn).

*ld*  $R_{dest}, address$                       Încarcare dublu cuvânt

Încarca 8 octeti de la adresa *address* în registrii  $R_{dest}$  si  $R_{dest}+1$ .

*lh*  $R_{dest}, address$                       Încarcare semicuvânt  
*lhu*  $R_{dest}, address$                       Încarcare semicuvânt fara semn

Încarca 2 octeti (un semicuvânt de 32 biti) de la adresa *address* în registrul destinație precum și extensia de semn.

***lw*  $R_{\text{dest}}$ , *address***                      Încarcare cuvânt

Încarca un cuvânt de 4 octeti de la adresa *address* în registrul  $R_{\text{dest}}$ .

***lwc*  $R_{\text{dest}}$ , *address***                      Încarcare cuvânt în coprocesor

Încarca un cuvânt de 4 octeti de la adresa *address* în unul din registrele coprocesorului z(0-3).

***lwl*  $R_{\text{dest}}$ , *address***                      Încarcare din stânga a cuvântului  
***lwr*  $R_{\text{dest}}$ , *address***                      Încarcare din dreapta a cuvântului

Încarca în registrul destinație octetii din stânga sau dreapta ai unui cuvânt în cazul unei adrese posibil nealiniate.

***sb*  $R_{\text{source}}$ , *address***                      Memorare octet

Memorează octetul mai puțin semnificativ al registrului sursă la adresa specificată de *address*.

***sd*  $R_{\text{source}}$ , *address***                      Memorare dublu cuvânt

Memorează 8 octeti (conținutul registrilor  $R_{\text{source}}$  și  $R_{\text{source}} + 1$ ) la adresa indicată de *address*.

***sh*  $R_{\text{source}}$ , *address***                      Memorare semicuvânt

Memorează primii doi octeti mai puțin semnificativi al registrului sursă la adresa specificată de *address*.

***sw*  $R_{\text{source}}$ , *address***                      Memorare cuvânt

Memorează cuvântul din registrul sursă la adresa *address*.

***swc*  $R_{\text{source}}$ , *address***                      Memorare cuvânt din coprocesor

Memorează cuvântul din  $R_{\text{source}}$  al coprocesorului z la adresa *address*.

<i>swl</i> $R_{source}$ , address	Memorare portiune stânga din cuvânt
<i>swr</i> $R_{source}$ , address	Memorare portiune dreapta din cuvânt

Memoreaza octetii din stânga sau dreapta ai registrului sursa în cazul unei posibile nealinieri a adresei.

***li*  $R_{dest}$ , Imm**                      Încarcare imediata

Transfera constanta imediata în registrul destinatie.

***li.d*  $FR_{dest}$ , float**                      Încarcare constanta imediata dubla precizie

Transfera un numar în virgula mobila dubla precizie în registrii flotanti  $FR_{dest}$  si  $FR_{dest} + 1$ .

***li.s*  $FR_{dest}$ , float**                      Încarcare constanta imediata simpla precizie

Transfera un numar în flotant simpla precizie în registrul flotant  $FR_{dest}$ .

***lui*  $R_{dest}$ , integer**                      Încarcare constanta întreaga

Încarca semicuvântul mai puțin semnificativ al unui întreg în semicuvântul semnificativ al registrului destinatie. Cei mai puțin semnificativi biti ai registrului sunt 0.

### 1.3.4.3. INSTRUCȚIUNI DE ÎNȚERUPERI SI EXCEȚII

***rfe***                      Revenire din exceptie

Reface registrul de stare al programului.

***syscall***                      Apel sistem

Registrul  $\$v_0$  contine numarul apelului.

*break n*

Exceptia n

Cauzeaza exceptia n. Exceptia 1 este rezervata debugger-ului.

*nop*

Nici o operatie

Nu executa nimic.

#### 1.3.4.4. INSTRUCIUNI DE COMPARATIE

În toate instructiunile de mai jos,  $S_{rc2}$  poate fi fie registru fie valoare imediata.

<i>seq</i> $R_{dest}, R_{src1}, S_{rc2}$	Seteaza flag în caz de egalitate
<i>sge</i> $R_{dest}, R_{src1}, S_{rc2}$	Seteaza flag pe relatia de mai mare sau egal
<i>sgt</i> $R_{dest}, R_{src1}, S_{rc2}$	Seteaza flag pe relatia de mai mare strict
<i>sle</i> $R_{dest}, R_{src1}, S_{rc2}$	Seteaza flag pe relatia de mai mic sau egal
<i>slt</i> $R_{dest}, R_{src1}, S_{rc2}$	Seteaza flag pe relatia de mai mic strict
<i>slti</i> $R_{dest}, R_{src1}, Imm$	Seteaza flag pe relatia de mai mic strict decât valoarea imediata
<i>sne</i> $R_{dest}, R_{src1}, S_{rc2}$	Seteaza flag în caz de inegalitate

Seteaza registrul destinatie la valoarea 1 daca operanzii  $R_{src1}$  si  $S_{rc2}$  (sau  $Imm$ ) sunt în relatia specificata si 0 altfel. Comparatiile pot fi cu sau fara semn.

#### 1.3.4.5. INSTRUCIUNI DE SALT SI RAMIFICATIE

În toate instructiunile de mai jos,  $S_{rc2}$  poate fi fie registru fie valoare imediata (întreg). Instructiunile de salt folosesc un câmp offset pe 16 biti cu semn. Deci pot exista salturi de  $2^{15}$  instructiuni “înainte” si  $2^{15}$  “înapoi”. Instructiunea *jump* contine un câmp de adresa de 26 de biti. Offset-ul instructiunilor nu este precizat, el fiind înlocuit cu eticheta. Acest lucru se întâmpla în majoritatea limbajelor de asamblare deoarece distanta dintre instructiuni este dificil de calculat când pseudoinstructiunile expandeaza în câteva instructiuni reale.

*b label*

Instructiunea branch

Se executa salt neconditionat la eticheta *label*.

<i>bczt</i> label	Branch pe conditie adevarata în coprocesor
<i>bczf</i> label	Branch pe conditie falsa în coprocesor

Se executa salt conditionat la eticheta *label* daca conditia din registrul coprocesorului este adevarata (sau falsa).

<i>beq</i> R <sub>src1</sub> , S <sub>rc2</sub> , label	Branch pe egalitate
<i>bge</i> R <sub>src1</sub> , S <sub>rc2</sub> , label	Branch pe conditie de mai mare sau egal
<i>bgt</i> R <sub>src1</sub> , S <sub>rc2</sub> , label	Branch pe conditie de mai mare strict
<i>ble</i> R <sub>src1</sub> , S <sub>rc2</sub> , label	Branch pe conditie de mai mic sau egal
<i>blt</i> R <sub>src1</sub> , S <sub>rc2</sub> , label	Branch pe conditie de mai mic strict
<i>bne</i> R <sub>src1</sub> , S <sub>rc2</sub> , label	Branch pe neegalitate

Salt conditionat la eticheta *label* daca continuturile celor doua registre sunt în relatia respectiva. Comparatia poate fi cu semn sau fara semn.

<b><i>beqz</i> R<sub>source</sub>, label</b>	Branch pe egalitate cu zero
<i>bgez</i> R <sub>source</sub> , label	Branch pe conditie de mai mare sau egal cu zero
<i>bgezal</i> R <sub>source</sub> , label	Branch pe conditie de mai mare sau egal cu zero si salvare legatura
<i>bltz</i> R <sub>source</sub> , label	Branch pe conditie de mai mic strict decât zero
<i>bnez</i> R <sub>source</sub> , label	Branch pe neegalitate cu zero
<i>bltzal</i> R <sub>source</sub> , label	Branch pe mai mic strict decât zero si salvare legatura
<i>bgtz</i> R <sub>source</sub> , label	Branch pe conditie de mai mare strict decât zero
<b><i>blez</i> R<sub>source</sub>, label</b>	Branch pe conditie de mai mic sau egal decât zero

Salt conditionat la eticheta *label* daca continutul registrului sursa îndeplineste conditiile respective. În plus unele instructiuni salveaza adresa urmatoarei instructiuni în registrul 31.

<b><i>j</i> label</b>	Jump
-----------------------	------

Salt neconditionat la eticheta.

<b><i>jal</i> label</b>	Jump si salvare legatura
<b><i>jalr</i> R<sub>source</sub></b>	Jump si salvare legatura prin registru

Salt neconditionat la eticheta sau la adresa specificata de registrul sursa. Adresa urmatoarei instructiuni e salvata în registrul 31. Deci un CALL/RET se transforma în JAL/JR \$31.

<b><i>jr</i> R<sub>source</sub></b>	Jump la adresa din registru
-------------------------------------	-----------------------------

Salt neconditionat la adresa specificata de registrul sursa.

#### 1.3.4.6. INSTRUCTIUNI DE TRANSFER

<b><i>move</i> R<sub>dest</sub>, R<sub>source</sub></b>	Transfer
---	----------

Transfera continutul registrului sursa în registrul destinatie. Rezultatul obtinut în urma executiei instructiunilor de înmultire si împartire este depus în doi registrii suplimentari, HI si LO. Aceste instructiuni transfera valori în si din registrii. Instructiunile de înmultire, împartire si rest, descrise anterior (vezi 1.3.4.1.), sunt pseudoinstructiuni care fac sa apara ca si cum unitatile de executie respective opereaza cu registrii generali si detecteaza conditii de eroare cum ar fi împartire cu 0 sau depasire de domeniu.

<b><i>mfhi</i> R<sub>dest</sub></b>	Transfera din HI
<b><i>mflo</i> R<sub>dest</sub></b>	Transfera din LO

Muta continutul registrului HI sau LO în registrul destinatie.

<b><i>mthi</i> R<sub>dest</sub></b>	Transfera în HI
<b><i>mtlo</i> R<sub>dest</sub></b>	Transfera în LO

Muta continutul registrului HI sau LO în registrul destinatie.

Fiecare coprocesor are propriul sau set de registre. Urmatoarele instructiuni transfera valori între aceste registre si registrele procesorului.

<b><i>mfcz</i> R<sub>dest</sub>, C<sub>opsource</sub></b>	Transfer din coprocesor z
---	---------------------------

Muta continutul registrului C<sub>opsource</sub> al coprocesorului z în registrul destinatie R<sub>dest</sub>.

*mfc1.d*  $R_{dest}, FR_{src1}$  Transfer dublu cuvânt flotant din coprocesorul 1

Transfera continutul registrelor flotante  $FR_{src1}$  si  $FR_{src1}+1$  în registrele procesorului  $R_{dest}$  si  $R_{dest}+1$ .

*mtcz*  $R_{source}, C_{opdest}$  Transfer în coprocesor z

Muta continutul registrului  $R_{source}$  al procesorului în registrul  $C_{opdest}$  al coprocesorului z.

**Nota:**

Instructiunile scrise cu litere aldine se vor întâlni mai des în practica.

#### 1.3.4.7. INSTRUCTIUNI ÎN VIRGULA MOBILA

Calculatorul MIPS are un coprocesor (notat cu 1) care opereaza în virgula mobila simpla precizie (valori pe 32 biti) si dubla precizie (valori pe 64 biti). Acest coprocesor are proprii sai registri, care sunt numerotati  $\$f_0$  -  $\$f_{31}$ . Deoarece aceste registre sunt doar pe 32 de biti, sunt necesare doua dintre ele pentru a pastra rezultatul în dubla precizie. Pentru a simplifica lucrurile, operatiile în virgula mobila folosesc doar registre numerotate par - chiar si instructiunile care opereaza în simpla precizie. Valorile sunt transferate în si din aceste registre pe cuvinte de 32 biti prin instructiunile: *lwc1*, *swc1*, *mtc1* si *mfc1* (descrise anterior) si *l.s*, *l.d*, *s.s* si *s.d* pseudoinstructiuni descrise în continuare. Flagul setat de catre operatiile de comparatie este citit de procesor prin instructiunile *bc1t* si *bc1f*. În toate instructiunile de mai jos  $FR_{dest}$ ,  $FR_{src1}$ ,  $FR_{src2}$  si  $FR_{source}$  sunt registrii flotanti.

*abs.d*  $FR_{dest}, FR_{source}$  Valoare absoluta registru flotant simpla precizie  
*abs.s*  $FR_{dest}, FR_{source}$  Valoare absoluta registru flotant dubla precizie

Calculeaza valoarea absoluta a registrului sursa flotant simpla (sau dubla) precizie si pune rezultatul în registrul destinatie.

*add.d*  $FR_{dest}, FR_{src1}, FR_{src2}$  Adunare în dubla precizie  
*add.s*  $FR_{dest}, FR_{src1}, FR_{src2}$  Adunare în simpla precizie

Calculeaza suma celor doi registri sursa în simpla sau dubla precizie si pune rezultatul în registrul destinatie.



*c.le.d*  $FR_{src1}, FR_{src2}$

Comparatie pe relatie de mai mic  
sau egal în dubla precizie

*c.le.s*  $FR_{src1}, FR_{src2}$

Comparatie pe relatie de mai mic  
sau egal în simpla precizie

Se seteaza flagul conditie în flotant pe true daca  $FR_{src1}$  este mai mic sau egal decât  $FR_{src2}$ .

*cvt.d.s*  $FR_{dest}, FR_{source}$

Conversie din simpla în dubla precizie

*cvt.d.w*  $FR_{dest}, FR_{source}$

Conversie din întreg în dubla precizie

Converteste un numar în simpla precizie (sau un întreg) aflat în registrul  $FR_{source}$  în dubla precizie si pune rezultatul în registrul  $FR_{dest}$ .

*cvt.s.d*  $FR_{dest}, FR_{source}$

Conversie din dubla în simpla precizie

*cvt.s.w*  $FR_{dest}, FR_{source}$

Conversie din întreg în simpla precizie

Converteste un numar în dubla precizie (sau un întreg) aflat în registrul  $FR_{source}$  în simpla precizie si pune rezultatul în registrul  $FR_{dest}$ .

*cvt.w.d*  $FR_{dest}, FR_{source}$

Conversie din dubla precizie în întreg

*cvt.w.s*  $FR_{dest}, FR_{source}$

Conversie din simpla precizie în întreg

Converteste un numar în dubla (sau simpla) precizie aflat în registrul  $FR_{source}$  în întreg si pune rezultatul în registrul  $FR_{dest}$ .

*div.d*  $FR_{dest}, FR_{src1}, FR_{src2}$

Împartire în dubla precizie

*div.s*  $FR_{dest}, FR_{src1}, FR_{src2}$

Împartire în simpla precizie

Calculeaza câtul împartirii celor doi registri  $FR_{src1}$  si  $FR_{src2}$  si pune rezultatul în registrul  $FR_{dest}$ .

*l.d*  $FR_{dest}, address$

Încarcare valoare dubla precizie de la adresa

*l.s*  $FR_{dest}, address$

Încarcare valoare simpla precizie de la adresa

Încarca o valoare în dubla (sau simpla) precizie de la adresa specificata.

*mov.d*  $FR_{dest}, FR_{source}$

Transfera valoare dubla precizie

*mov.s*  $FR_{dest}, FR_{source}$

Transfera valoare simpla precizie

Transfera o valoare dubla (sau simpla) precizie din registrul sursa în cel destinatie.

$mul.d \text{ FR}_{dest}, \text{FR}_{src1}, \text{FR}_{src2}$       Înmultire în dubla precizie  
 $mul.s \text{ FR}_{dest}, \text{FR}_{src1}, \text{FR}_{src2}$       Înmultire în simpla precizie

Calculeaza produsul celor doi registri  $\text{FR}_{src1}$  si  $\text{FR}_{src2}$  si pune rezultatul în registrul  $\text{FR}_{dest}$ .

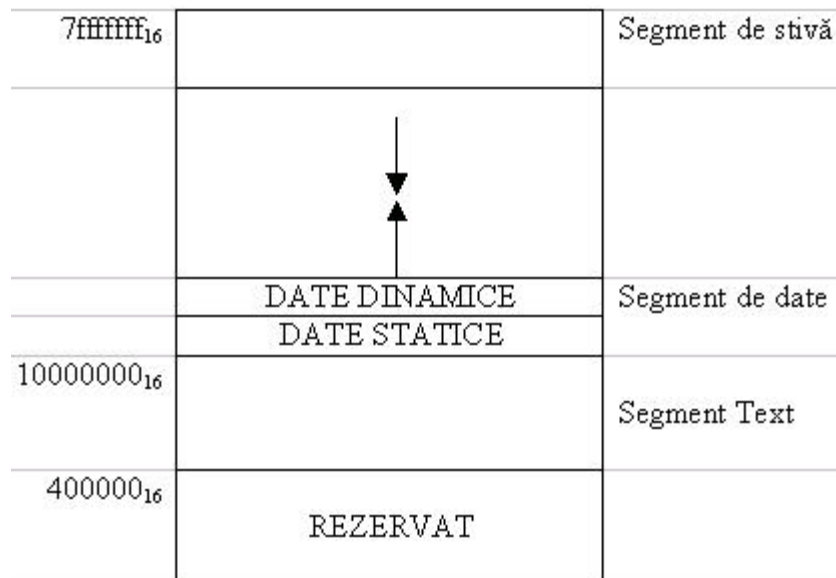
$s.d \text{ FR}_{dest}, \text{address}$       Memorare valoare dubla precizie la adresa  
 $s.s \text{ FR}_{dest}, \text{address}$       Memorare valoare simpla precizie la adresa

Scrie o valoare în dubla (sau simpla) precizie la adresa specificata.

$sub.d \text{ FR}_{dest}, \text{FR}_{src1}, \text{FR}_{src2}$       Scadere în dubla precizie  
 $sub.s \text{ FR}_{dest}, \text{FR}_{src1}, \text{FR}_{src2}$       Scadere în simpla precizie

Calculeaza diferenta celor doi registrii  $\text{FR}_{src1}$  si  $\text{FR}_{src2}$  si pune rezultatul în registrul  $\text{FR}_{dest}$ .

### 1.3.5. UTILIZAREA MEMORIEI SI CONVENTII DE APEL



**Figura 1.2.** Harta de memorie a microprocesorului MIPS R2000

Organizarea memoriei în sistemele MIPS este convetionala. Spatiul de adresa al unui program este compus din trei parti. Prima zona din spatiul de adresa utilizator (începe la 0x400000) este segmentul text, care memoreaza instructiunile programului. Deasupra segmentului de text este segmentul de date (începând de la adresa 0x1000000) si este împartita în doua parti. Zona de date statica contine obiecte a caror marime si adresa sunt cunoscute de catre compilator si link-editor. Imediat, deasupra acestei zone se afla datele dinamice. La alocarea spatiului dinamic de memorie pentru un program (prin **malloc** si apelul sistem **sbrk**) marginea superioara a segmentului de date se deplaseaza în sus. La marginea superioara a spatiului de adresa (0x7fffffff) este stiva de program, care creste în jos, spre segmentul de date.

Un *cadru* consta în memoria dintre indicatorul de cadru (\$fp), care pointeaza la cuvântul imediat urmator ultimului argument transmis pe stiva, si indicatorul de stiva (\$sp), care pointeaza la primul cuvânt liber pe stiva. Tipic pentru sistemele UNIX, stiva creste în jos de la adrese de memorie mai mari, astfel încât indicatorul de cadru este deasupra indicatorului de stiva.

Pentru a efectua un apel sunt necesari urmasorii pasi:

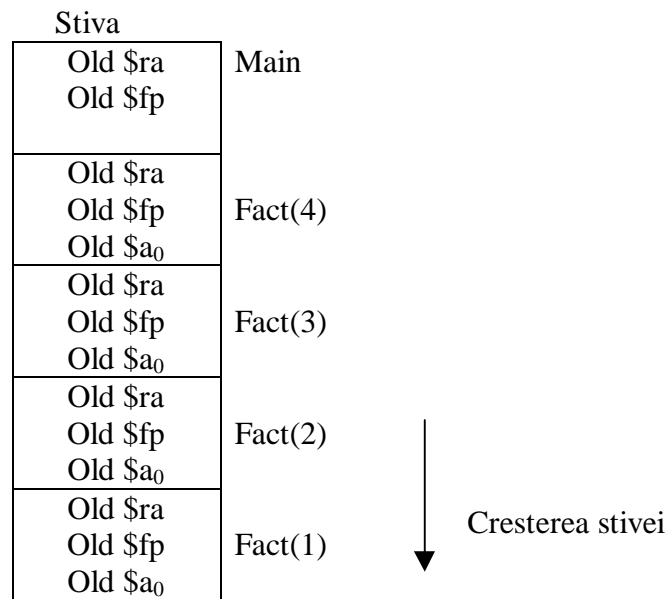
- a. Transmiterea argumentelor. Prin conventie, primele patru argumente sunt transferate în registrii \$a<sub>0</sub> - \$a<sub>3</sub> (chiar unele compilatoare mai simple simplifica aceasta conventie si transfera toate argumentele pe stiva). Argumentele ramase sunt puse pe stiva.
- b. Salveaza registrii \$t<sub>0</sub> - \$t<sub>9</sub>.
- c. Executa instructiunea *jal*.

În subrutina apelata sunt necesari urmasorii pasi:

- a. Se determina cadrul de stiva prin scaderea indicatorului de cadru din indicatorul de stiva.
- b. Salveaza registrii \$s<sub>0</sub> - \$s<sub>7</sub> în cadru. Registrul \$fp este întotdeauna salvat, iar \$ra e necesar a fi salvat doar când subrutina apeleaza alte subrutine.
- c. Stabileste indicatorul de cadru prin adaugarea dimensiunii cadrului de stiva la adresa din \$sp.

În final, la revenirea din subrutina, rezultatul unei functii este plasat în registrul \$v<sub>0</sub> si se executa urmasorii pasi:

- a. Restaureaza toti registrii care au fost salvati la intrarea în subrutina.
- b. Reface cadrul de stiva prin scaderea dimensiunii cadrului din \$sp.
- c. Se revine la adresa specificata de registrul \$ra.



**Figura 1.3.** Imaginea stivei pentru calculul valorii fact(4)

### 1.3.6. APELURI SISTEM

SPIM asigura un mic set de servicii ale sistemului de operare prin instructiuni (*syscall*) – apeluri sistem. Pentru a apela un serviciu, programul încarca codul apelului în registrul \$v<sub>0</sub> si argumentele în registrele \$a<sub>0</sub> ... \$a<sub>3</sub> (sau \$f<sub>12</sub> pentru valori în flotant). Apelurile sistem care returneaza valori pun rezultatele în registrele \$v<sub>0</sub> sau \$f<sub>0</sub> pentru operatii în flotant.

Serviciul	Cod Apel Sistem	Argumentele	Rezultatul
Print_int	1	\$a <sub>0</sub> = integer	
Print_float	2	\$f <sub>12</sub> = float	
Print_double	3	\$f <sub>12</sub> = double	
Print_string	4	\$a <sub>0</sub> = string	
Read_int	5		Integer (în \$v <sub>0</sub> )
Read_float	6		Float (în \$f <sub>0</sub> )
Read_double	7		Double (în \$f <sub>0</sub> )
Read_string	8	\$a <sub>0</sub> = buffer, \$a <sub>1</sub> = lungimea	

Sbrk	9	\$a <sub>0</sub> = cantitate	Adresa (în \$v <sub>0</sub> )
Exit	10		
Print_char	11	\$a <sub>0</sub> = codul ASCII al caracterului '\n'	
Read_char	12		Codul ASCII al caracterului citit de la tastatura se depune în \$v <sub>0</sub>

Tabelul 1.4.

### Servicii Sistem

*Sbrk* întoarce un pointer la un bloc de memorie ce conține *n* octeți, iar *exit* oprește rularea programului.

Programele care folosesc aceste apeluri sistem pentru a citi de la terminal nu trebuie să folosească dispozitive de mapare a memoriei.

#### Exemplul 1.

Pentru exemplificare vom scrie codul necesar afisării unui mesaj de tipul: “*solutia corecta = 10*”.

```
.data
str:
    .asciiz "solutia corecta ="

.text
li $v0, 4           # codul apelului sistem pentru afisare de string
la    $a0, str       # adresa sirului de tiparit
syscall             # afiseaza sirul

li $v0, 1           # codul apelului sistem pentru afisare de
                    # întreg
li $a0, 10          # valoarea întreaga de tiparit
syscall             # afiseaza valoarea
```

În momentul rularii secvenței de instrucțiuni vom observa că instrucțiunea *li \$v0, 4* – care este de fapt o pseudoinstrucțiune, este înlocuită de instrucțiunea *ori \$2, \$0, 4*; instrucțiunea de încărcare imediată în registru se obține printr-o operație de SAU între registrul 0, care este întotdeauna 0, și valoarea respectivă.

Adresa etichetei *str*, unde începe sirul de date este 10010000h. Reamintim ca, segmentul de date începe la adresa 10000000h, iar compilatorul MIPS memoreaza variabilele globale în primii 64 Kocteti, deoarece aceste variabile sunt mai frecvent accesate decât alte date globale.

### Exemplul 2.

Urmatoarea secventa va afisa pe ecran mesajul: “*Dati un sir de caractere:* ” si asteapta tastarea unui string de la tastatura.

```
.data
d3: .asciiz "Tastati un sir de caractere: "
d4: .space 50

.text
la $a0, d3          # adresa sirului de tiparit
li $v0, 4            # codul apelului sistem pentru afisare de string
syscall             # afiseaza sirul

la $a0, d4           # adresa unde se va memora sirul
li $a1, 50           # lungimea maxima a sirului
li $v0, 8            # codul apelului sistem pentru citire string de la
                    # tastatura
syscall             # se citeste sirul
done                # se revine din program
```

### 1.3.7. PSEUDOINSTRUCTIUNI FOLOSITE ÎN LIMBAJ DE ASAMBLARE MIPS

Prezentam în continuare câteva din pseudoinstructiunile pe care le vom întâlni în lucrările de laborator ulterioare. Pseudoinstructiunile sunt asemanatoare cu instructiuni din limbajul C, cum ar fi cele de afisare caractere, valori întregi, siruri de caractere, citire de la tastatura a valori de diverse tipuri (**int**, **char**, **string**).

■ Afisare string pe consola:

**puts *label***      - are ca argument eticheta de la care se va afisa sirul de caractere

- Afisare caracter pe consola:  
**putc *vchar*** - are ca argument caracterul care se va afisa pe consola
- Afisare numar întreg pe consola:  
**puti \$a0** - are ca argument registrul al carui continut este numarul care se va afisa pe consola
- Citire caracter de la tastatura:  
**getc \$a0** - are ca argument registrul în care se va încarca codul ASCII al caracterului citit
- Citire întreg de la tastatura:  
**geti \$a0** - are ca argument registrul în care se va încarca valoarea întreaga citita
- Terminare program:  
**done** - nu are argumente si determina încheierea programului
- Apel sistem:  
**syscall** - nu are argumente si determina un apel sistem de citire de la tastatura/ afisare pe consola sau terminare program

### 1.3.8. GHID DE UTILIZARE AL SIMULATORULUI

Când pornim simulatorul pe ecran apare fereastra din figura 1.4. Pasul urmator ar fi încărcarea unui fisier care urmeaza a fi executat. Se selecteaza optiunea *Load* din meniul *File*, care va declansa aparitia ferestrei de selectie a fisierelor. SPIM/SAL accepta fisiere cu extensia “.s”, care sunt fisiere în limbaj de asamblare MIPS. Fisierul selectat este analizat sintactic, iar tipurile de erori depistate sunt scrise în fereastra Sesiune. Încărcarea fisierelor se poate face totodata si din linia de comanda.

Pentru reluarea unui program sau încărcarea unui nou fisier e necesara reinitializarea starilor masinii si a registrilor. Se selecteaza optiunea *Clear* din meniul *File*. Exista trei posibilitati: stergerea ferestrei consola, a registrilor sau a memoriei.

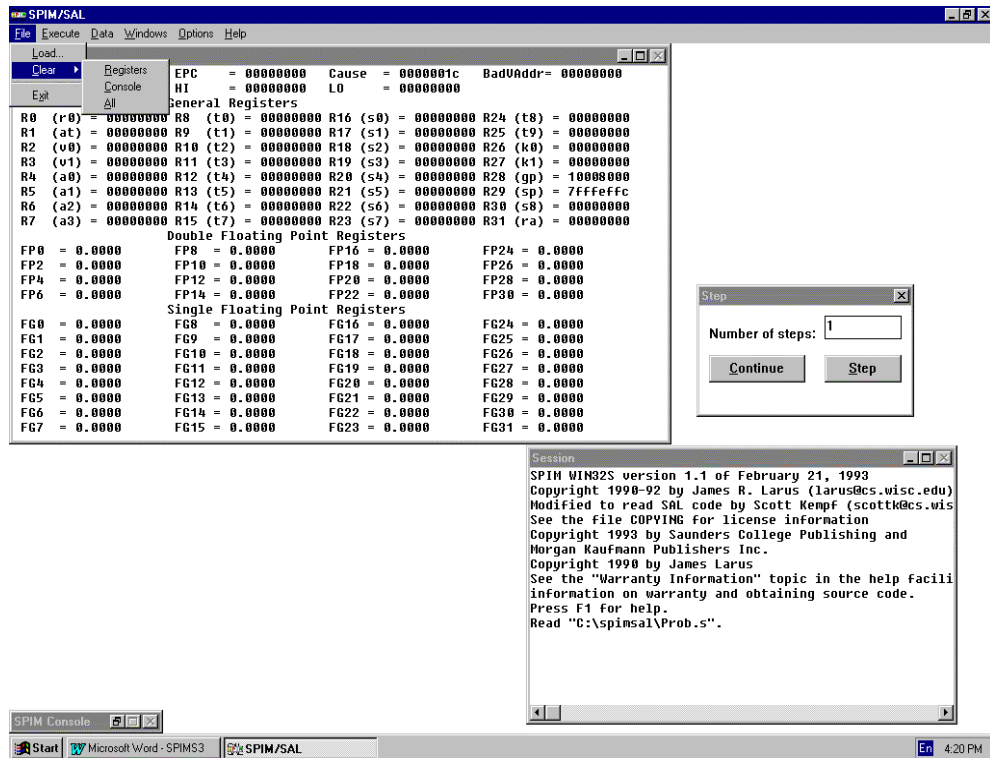


Figura 1.4. SPIMSAL: Interfata cu utilizatorul

Executia programului poate fi realizata în mod continuu sau pas cu pas. Pentru aceasta a doua varianta, se selecteaza *Step* din meniul *Execute*. Pe ecran apare o cutie de dialog care permite selectia adresei de start si numarul de pasi în care se executa. Cutia de dialog ramâne deschisa pâna la închiderea ei. Concomitent instructiunile sunt reflectate în fereastra Sesiune, atât instructiunile cât si adresa lor.

Managementul operatiilor de întrerupere executie program (*breakpoint*) se face prin selectia optiunii *Breakpoint* din meniul *Execute*. O cutie de dialog va apare pe ecran din care se va selecta operatia si adresa tinta pentru operatia de *breakpoint*. În cazul selectiei unei adrese la care nu exista instructiuni se genereaza un mesaj de eroare în fereastra Sesiune. Când este întâlnit un punct de întrerupere în executia programului, programul se opreste si apare o cutie de dialog care permite una din optiunile: de continuare sau oprire a programului.

Stabilirea unei date se face alegând optiunea *Set* din meniul *Data*. O cutie de dialog va apare pe ecran si prin intermediul ei se va specifica adresa unde vrem sa scriem si data pe care o setam la adresa respectiva.



Afisarea unei date în fereastra Sesiune se face prin selectarea optiunii *Print* din meniul *Data*. O cutie de dialog va permite alegerea domeniului de adrese.

Din meniul *Options* se poate alege modul de lucru simplu al asamblorului MIPS - *Bare* si *Quiet* pentru a nu afisa mesaje la exceptie. Prin selectia optiunii *Save Settings* din meniul *Options* se salveaza aceste moduri de functionare ale simulatorului SPIM în fisierul de initializare al Windows-ului - *win.ini*.

SPIM/SAL ofera permanent cinci ferestre: fereastra principala, fereastra Sesiune, fereastra Consola, fereastra Registru si fereastra Memorie. Fiecare are anumite caracteristici:

Fereastra principala este o fereastra simpla care contine un meniu. Ea poate fi redimensionata, minimizata, maximizata si închisa.

Fereastra Sesiune este o fereastra copil care obisnuieste sa afiseze mesaje sistem. Este singura fereastra copil care este initial vizibila. Ea poate fi maximizata, minimizata, redimensionata, ascunsa sau derulata. Exista un buffer de dimensiune fixa asociat ferestrei sesiune. Cele mai recent scrise mesaje din fereastra sunt pierdute la fel ca si mesajele sosite dupa ce capacitatea buffer-ului s-a atins.

Fereastra Memorie este o fereastra copil care obisnuieste sa afiseze continutul memoriei. Exista o sectiune separata pentru fiecare segment. Ea poate fi maximizata, minimizata, redimensionata, ascunsa si derulata.

Fereastra Registru este o fereastra copil care afiseaza valorile registrelor microprocesorului. Are aceleasi attribute cu fereastra Memorie.

Fereastra Consola este o fereastra de dimensiune fixa care este folosita la receptionarea iesirilor programului si asigura intrarea pentru programe. Poate fi minimizata si ascunsa dar nu maximizata, redimensionata sau derulata. Programele interactioneaza cu fereastra Consola prin apeluri sistem.

Cele patru ferestre (principala, Sesiune, Memorie, Registru) pot coexista pe parcursul rularii unui program. Oricare din acestea poate fi adusa în prim plan, printr-un click pe bara *Caption* a ferestrei respective, celelalte trecând în plan secundar. Acest lucru nu se poate realiza si cu fereastra Consola, ea ramânând în prim plan atât timp cât ea este activa si nu este minimizata.

## 1.4. PROBLEME DE LABORATOR PROPUSE SPRE REZOLVARE

### 1.4.1. NOTE EXPLICATIVE REFERITOARE LA BENCHMARK-URILE IMPLEMENTATE

Benchmark-urile prezentate sunt de dificultate medie (*Conjectura lui Goldbach*, *afisarea recursiva în ordine inversa a numerelor citite de la tastatura*) care utilizeaza pseudoinstructiunile ce expandeaza în apeluri system, lucrul cu stiva, prezentate anterior, pentru afisarea pe fereastra Consola a sirurilor de caractere, valori întregi, caractere, pentru citirea de la tastatura a variabilelor de diverse tipuri (**int**, **string**, **char**), pentru terminarea programului. Se vor observa cu ajutorul simulatorului prin intermediul ferestrelor **Data**, **Registru**, **Sesiune** si **Consola** valorile din registri, continutul locatiilor de memorie, instructiunile efectiv executate în urma rularii pas cu pas sau în mod continuu a programului.

Zona de date utilizator începe la adresa 10010000h. Initial, valorile din registrii generali sunt 0. Registrul *Program Counter* are valoarea PC=00400000h, care este adresa primei instructiuni a programului utilizator. Registrul pointer global la date, GP=10008000h, indica spre mijlocul unei zone de date de 64K octeti (de la adresa 10000000h - la adresa 1000FFFFh) în segmentul de date statice.

### 1.4.2. SOLUTII LA PROBLEME PROPUSE

**1. (*Conjectura lui Goldbach*)** Scrieti un program în limbaj de asamblare MIPS, care citește de la tastatura, prin intermediul modulului Input.s, un numar  $n$  par,  $n > 6$ . Sa se determine toate reprezentarile lui  $n$  ca suma de numere prime, suma cu numar minim de termeni. Afisarea se face pe consola pe fiecare linie câte o solutie.

#### Solutia problemei 1

```
.data
```

```
Mesaj:
```

```
.asciiz "Introduceti numarul N (par si mai mare ca 6): "
```

```

msg1:
    .asciiz "\n\tSuma: "
msg2:
    .asciiz "+"
err1:
    .asciiz "Va rog un numar par !"
err2:
    .asciiz "Va rog un nr mai mare ca 6 !"

    .text
Start:
    li    $v0,4          # Citire Numar
    la    $a0,Mesaj
    syscall
    li    $v0,5
    syscall
    sle    $a0,$v0,6
    bnez   $a0,eroare1
    addi   $a3,$v0,0      # in a3 numarul citit - fie N
    li    $a0,2
    div    $v0,$a0
    mfhi   $a0
    mflo   $a2            # in a2 mijlocul numarului N
    bnez   $a0,eroare2
    j      Cont
eroare1:
    li    $v0,4
    la    $a0,err2        # afisare mesaj ajutator pentru testarea unui
                          # numar par ≥ 6
    syscall
    j      Start
eroare2:
    li    $v0,4
    la    $a0,err1        # afisare mesaj ajutator pentru testarea unui
                          # numar par
    syscall
    j      Start
Cont:
    li    $t0,0
Cont1:
    addi   $t0,$t0,1

```

	sgt	\$s0,\$t0,\$a2	# s0 devine 1 daca s-a trecut de # mijlocul sirului si se incheie # programul
j	bnez	\$s0,Sfarsit	
verif_t0:	j verif_t0		# daca nu s-a trecut de mijlocul sirului verific # daca i (primul element din suma) este prim
	li	\$s2,1	
repete:			
	addi	\$s2,\$s2,1	# primul divizor s2=2
	sge	\$s5,\$s2,\$t0	# daca prezumtivul divizor a depasit numarul # - i - adica s5=1 inseamna ca nu s-a impartit # niciodata exact cu vreun divizor => numarul # este prim
	bnez	\$s5,EPrim	
	div	\$t0,\$s2	
	mfhi	\$s3	
	beqz	\$s3,Cont1	# daca t0 se imparte la unul din divizorii sai # (s2) atunci numarul curent nu este prim si se # trece la urmatorul numar - (i+1)
	j	repete	# se verifica in continuare impartirea # numarului - i - la un alt posibil divizor
EPrim:			
	li	\$s2,1	# daca primul numar a fost prim - i - atunci se # va testa daca si N-i este prim
	sub	\$s4,\$a3,\$t0	# s4=a3-t0
repete_2:			
	addi	\$s2,\$s2,1	# din nou primul divizor ese s2=2
	sge	\$s5,\$s2,\$s4	# daca prezumtivul divizor a depasit numarul # - (N-i) - # adica s5=1 inseamna ca nu s-a # impartit niciodata exact cu vreun divizor => # numarul este prim
	bnez	\$s5,Afisare	# daca s5=1 inseamna ca ambele numere sunt # prime # deci va fi afisata perechea (i, N-i)
	div	\$s4,\$s2	
	mfhi	\$s3	
	beqz	\$s3,Cont1	# daca (N-i) adica s4 se imparte la unul din # divizorii sai (s2) atunci numarul curent nu # este prim si se trece la urmatorul numar # - (i+1)

```

        j      repeta_2      # se verifica in continuare impartirea
                              # numarului - (N-i) - la un alt posibil divizor
Afisare:
        li     $v0,4
        la     $a0,msg1
        syscall
        li     $v0,1
        addi   $a0,$t0,0
        syscall      # afisare i (t0)
        li     $v0,4
        la     $a0,msg2
        syscall
        li     $v0,1
        addi   $a0,$s4,0      # afisare N-i (s4)
        syscall
        j      Cont1         # verific o urmatoare pereche de numere
                              # (i+1, N-i-1)
Sfarsit:
        li     $v0,10
        syscall

```

2. Scrieti un program folosind recursivitatea, care citeste caractere de la tastatura si le afiseaza în ordine inversa.

**Obs.** Nu se lucreaza cu siruri, nu se cunoaste numarul de caractere citite, sfârșitul sirului va fi dat de citirea caracterului '0'.

Modificati programul astfel încât '0' – care marcheaza sfarsitul sirului, sa nu fie tiparit.

### Solutia problemei 2

```

        .data
d3:     .asciiz "Dati un sir de caractere (sirul se incheie cu '0'): "
d4:     .asciiz "S-a tastat un sir vid !"
d5:     .asciiz "\nExecutia programului s-a incheiat !"
        .globl main

        .text
main:
        subu   $sp, $sp, 12
        sw     $ra, 8($sp)      # se pastreaza adresa de revenire in programul

```

```

                                # principal
sw $fp, 4($sp)                # se pastreaza pe stiva indicatorul de cadru cu
                                # valoarea din momentul inceperii executiei
                                # programului
addu $fp, $sp, 12             # se determina indicatorul de cadru
puts d3
li $t0, -1                    # prima citire de la tastatura

et:
getc $a0                      # se depune in a0 caracterul dorit
add $t0, $t0, 1
li $a1, 0x00000030            # codul ASCII al caracterului '0'
beq $a0, $a1, afis            # daca s-a tastat '0' atunci se indeplineste
                                # conditia de iesire din recursivitate si se trece
                                # la afisarea numerelor anterior citite
jal inv                        # Nu s-a tastat '0' => se continua cu apelul
                                # recursiv
                                # al rutinei de inversare

afis:
putc $a0                      # dupa apel se face afisarea
bne $t0, 0, more_write        # t0!=0 => mai sunt elemente de citit
                                # din stiva si de afisat
j end                          # t0=0 => nu mai sunt elemente in stiva se va
                                # afisa mesajul de sfarsit program

more_write:
lw $a0, 0($fp)                # se preia in a0 caracterul ce va fi afisat
lw $ra, 8($sp)                # se reface adresa de revenire din subrutina
lw $fp, 4($sp)                # se reface indicatorul de cadru
addu $sp, $sp, 12             # se reface indicatorul de stiva
add $t0, $t0, -1
j $ra

inv:
subu $sp, $sp, 12             # se determina cadrul de stiva
sw $ra, 8($sp)                # se salveaza pe stiva adresa de revenire - ra –
                                # fiind o functie recursiva
sw $fp, 4($sp)                # se salveaza indicatorul de cadru
addu $fp, $sp, 12
sw $a0, 0($fp)                # se salveaza pe stiva a0 (valoarea citita)

```

```

        j et

end:
        puts d5          # afisare mesaj 'Executia programului s-a
                          # incheiat!'

        done

```

Pentru ca programul sa nu mai afiseze caracterul ‘0’ care marcheaza sfarsitul sirului, în codul sursa al programului prezentat mai sus se vor face urmatoarele modificari:

- Dupa instructiunea **jal inv** (apelul de subrutina) nu se mai scrie instructiunea **putc \$a0** (afisarea caracterului curent) iar,
- În “subrutina” **more\_write** înainte de revenirea din subprogram **j \$ra** se va înscrie instructiunea **putc \$a0**.

### 1.4.3. PROBLEME PROPUSE SPRE REZOLVARE

1. Scrieti si testati un program în limbaj de asamblare MIPS care sa calculeze si afiseze primele 24 numere prime. Un numar  $n$  este prim daca el se divide exact doar cu 1 si cu el însusi. Pentru usurinta sa se implementeze întâi o rutina *test\_prime(n)*, care întoarce 1 (sau un mesaj “Numarul este prim!”) daca numarul este prim si 0 (sau mesaj “Numarul nu este prim!”), altfel. Folosindu-ne apoi de aceasta rutina se va scrie un program care parcurge numerele naturale si testeaza daca fiecare este prim. Se vor afisa primele astfel de 24 de numere, fiecare pe un rând nou [n (initial 24) - va putea lua ulterior orice valoare]. Testarea programului se va face pe simulatorul SPIM. Se vor rula programele pas cu pas, se vor modifica parametrii (numarul de test - în cazul rutinei *test\_prime* si numarul de numere prime generate - în cazul programului de *generare a numerelor prime*) si se va verifica corectitudinea rezultatelor. Întrucât în fereastra Consola nu încap decât maxim 25 numere prime pe o linie si consola prezinta maxim 24 de linii, se va modifica programul astfel încât sa poata fi afisate maxim  $n=24 \times 25$  numere prime.
2. Scrieti un program care sa calculeze factorialul unui numar natural  $n$  (de exemplu  $n=10$ ). Se va folosi o rutina de calcul recursiv *fact*, care-l obtine pe  $n!$  din înmultirea lui  $(n-1)!$  cu  $n$ . Codul scris în limbaj de asamblare al rutinei va ilustra cum programul afecteaza stiva, registrii \$sp, \$fp, \$ra. Programul principal *main* va creea cadrul de stiva si salveaza registrii \$fp

si \$ra (pointer-ul de cadru si adresa de revenire din procedura). Sa se ruleze programul pas cu pas, sa se urmareasca continutul stivei si al registrilor, actualizarea registrilor la revenirea din subrutina si corectitudinea rezultatelor.

3. Scrieti un program care sa sorteze un sir de  $n$  ( $n$  fiind citit de la tastatura) numere întregi prin metoda “bulelor” (*bubblesort*). Rulati programul pentru diverse siruri de numere de dimensiuni diferite.

**Obs:**

1. O deficiente a simulatorului SPIM consta în imposibilitatea de a varia diferiti parametri arhitecturali ai procesorului (latenta instructiunilor, numar unitati de executie), precum si validarea sau invalidarea unor tehnici gen *forwarding*.
2. Modul preferat de rulare al programelor este pas cu pas. Programele pot fi executate si prin puncte de întrerupere, dificultatea consta însa în cunoasterea exacta a adreselor instructiunilor pe care se stabileste întreruperea.



## 2. INVESTIGAȚII ARHITECTURALE UTILIZÂND SIMULATORUL *DLX*

---

### 2.1. SCOPUL LUCRĂRII

Scopul lucrării este de a ajuta la înțelegerea conceptelor legate de procesarea pipeline a instrucțiunilor precum și a aspectelor arhitecturale specifice procesoarelor RISC. În prima parte a lucrării este prezentată arhitectura procesorului didactic virtual DLX, formatul instrucțiunii și setul de instrucțiuni, modurile de adresare ale acestuia, apeluri sistem uzuale. În continuare este realizată o descriere, nu foarte detaliată, a simulatorului DLX, un simulator scris pentru sistemul Windows de către J. Hennessy și D. Patterson. Pentru utilizarea simulatorului dar și interpretarea rezultatelor sunt necesare cunostințe minime de operare în sistemul Windows dar și cunostințe privind depanarea la nivel de execuție a programelor de test. Interfața cu utilizatorul este extrem de simplă și intuitivă bazată pe ferestre de dialog și meniuri. În finalul lucrării sunt explicate două aplicații de complexitate redusă și propuse alte patru probleme pentru rezolvare având ca scop formarea unor deprinderi practice privind programarea în limbajul de asamblare al procesorului DLX, precum și o analiză cantitativă și calitativă a rezultatelor obținute în urma simulării efectuate pe benchmark-urile existente.

### 2.2. MEMENTO TEORETIC

DLX-ul este un microprocesor didactic care a fost conceput pe baza unor teste executate pe o serie de microprocesoare comerciale cu o filosofie asemănătoare cu cea a DLX-ului. (de ex. MIPS, SPARC, etc.) Setul de instrucțiuni al acestui microprocesor se bazează pe o serie de considerații

generale privitoare la caracteristicile microprocesoarelor moderne actuale RISC cum ar fi:

- Utilizarea tehnicilor de procesare pipeline a instructiunilor, ceea ce implica o rata teoretica de executie de o instructiune / ciclu, pe modelele de procesoare care pot lansa în executie la un moment dat o singura instructiune (procesoare scalare).
- Set relativ redus de instructiuni simple, majoritatea fara referire la memorie si cu putine moduri de adresare (set optimizat de instructiuni în vederea implementarii aplicatiilor propuse în limbajele de nivel înalt - C, C++, Pascal, etc.). În general, doar instructiunile LOAD / STORE sunt cu referire la memorie (arhitectura tip LOAD / STORE).
- Format fix al instructiunilor, codificate în general pe un singur cuvânt de 32 biti, mai recent pe 64 biti (Alpha 21264, IA-64 Merced etc.).
- Set de registre generale substantial mai mare decât la CISC-uri, în vederea lucrului "în ferestre" (register windows), util în optimizarea instructiunilor CALL / RET. Numarul mare de registre generale este util si pentru marirea spatiului intern de procesare, tratarii optimizate a evenimentelor de exceptie, modelului ortogonal de programare, etc.
- Registrul R0 este cablat la zero în majoritatea implementarilor, pentru optimizarea modurilor de adresare si a instructiunilor.

DLX ofera un model arhitectural bun pentru studiu nu doar datorita recenteii popularitati al acestui tip de masina dar si datorita arhitecturii simple si inteligibile.

Simulatorul DLX descrie modul de functionare al procesorului DLX (DeLuXe), un procesor pipeline, folosit drept exemplu în lucrarea "*Computer Architecture - A Quantitative Approach*", scrisa de *J. Hennessy* si *D. Patterson* [Hen96]. Procesorul DLX are cinci nivele distincte în procesarea instructiunilor. În nivelul **IF** (fetch instructiune) - se calculeaza adresa grupului de instructiuni ce trebuiesc citite din memoria principala. Al doilea nivel: **ID** (decodificare instructiune) - decodifica instructiunile aduse, se citesc operanzii din setul de registri generali, se calculeaza adresa de salt (pentru instructiunile de ramificatie). În timpul celei de-a treia faze de procesare a pipe-ului: **EX** (executa instructiune) se executa operatii aritmetico-logice, de deplasare si rotire asupra operanzilor care pot fi numere întregi sau flotante, se calculeaza adresa de acces la memorie (pentru instructiunile LOAD sau STORE). Accesarea / scrierea datelor din / în memoria principala, prin instructiunile LOAD sau STORE se face în nivelul patru al pipe-ului **MEM**. În final, avem al cincilea nivel **WB** (scriere date) - în care, unitatile functionale preiau rezultatul final al instructiunilor aritmetico-logice sau data citita din memorie, si o depun pe magistrala rezultat, de unde este copiată în registrul destinatie corespunzator.

Modul de lucru al simulatorului este urmatorul: dupa încărcarea unui program de test scris în asamblare pentru procesorul DLX, majoritatea informatiilor relevante pentru CPU (registrii, memorie, intrari/iesiri) pot fi vizualizate si modificate în timpul rularii pas cu pas sau continuu a benchmark-ului. WinDLX ofera totodata, statistici despre comportamentul pipeline în timp al procesorului. Cele trei benchmark-uri care fac parte din pachetul simulatorului sunt programe de calcul cu numere întregi, scrise în limbaj de asamblare DLX. Benchmark-ul PRIM.s genereaza un tabel cu primele *Count* (numar arbitrar) numere prime aflate în memorie începând cu adresa *Table*. Pentru calculul *celui mai mare divizor comun* a doua numere sunt necesare modulele INPUT.s - care citeste de la tastatura doua numere întregi si GCM.s - care realizeaza efectiv calculul divizorului si-l afiseaza pe ecran. Factorialul unui numar, introdus de la tastatura cu ajutorul modulului INPUT.s, memorat în registrul 1 al procesorului DLX, este calculat si afisat pe ecran în benchmark-ul FACT.s. Pe lângă cele trei surse existente plus rutina Input, autorii acestei carti au dezvoltat în cadrul laboratorului de *Organizarea si proiectarea microarhitecturilor* o multime de alte surse de la complexitate redusa la complexitate ridicata dintre care sunt prezentate în lucrare doar doua.

## 2.3. DESFASURAREA LUCRARIII

### 2.3.1. ARHITECTURA MICROPROCESOARELOR DLX

#### 2.3.1.1. REGISTRII MICROPROCESORULUI DLX

DLX-ul are un set de 32 de registri generali (GPR) pe 32 de biti denumiti R0, R1,...R31. Additional mai exista si un set de registri de virgula flotanta (FPR) care pot fi folositi registri pe 32 de biti în simpla precizie sau ca perechi (par-impar) pentru valori în dubla precizie. Registrii pentru virgula flotanta pe 64 de biti sunt denumiti F0, F1...F30. Sunt admise operatiile pe 32 si 64 de biti. Valoarea lui R0 este întotdeauna 0 (cablat la masa – caracteristica de altfel comuna tuturor microprocesoarelor RISC).

Exista câtiva registri speciali care pot fi convertiti în si din registrii de întregi (*Exemplu:* registrul de stare în virgula mobila folosit la pastrarea rezultatelor în virgula mobila). Exista de asemenea instructiuni pentru transferul între FPR si GPR.

### 2.3.1.2. TIPURI DE DATE

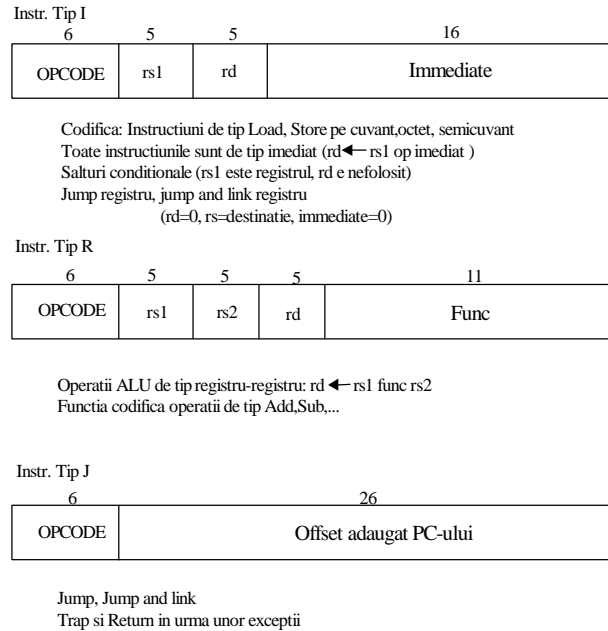
Datele sunt pe 8 biti, 16 biti (semicuvânt) si pe 32 de biti (cuvânt) pentru operanzi întregi si 32 de biti simpla precizie si 64 de biti dubla precizie pentru operanzi în virgula mobila. Suportul pentru semicuvinte a fost adaugat pentru ca ele se regasesc în limbaje ca C si în sisteme de operare datorita faptului ca aici se urmareste o dimensionare cât mai economica a structurilor. Operanzii în virgula flotanta simpla precizie au fost adaugati din aceleasi ratiuni. Operatiile lui DLX se pot efectua pe operanzi întregi de 32 de biti si pe operanzi de 32 si 64 de biti în virgula flotanta. Operanzii pe octet sau semicuvânt sunt încarcati în registri urmati fiind de umplerea cu zerouri sau cu bitul de semn a locatiilor libere ramase din cei 32 de biti ai registrului. Odata încarcati acesti operanzi se vor comporta ca un operand pe 32 de biti.

### 2.3.1.3. MODURI DE ADRESARE LA DLX

Singurele moduri de adresare suportate de catre DLX sunt cele imediat si respectiv indexat, ambele cu deplasamentul pe 16 biti. Modul de adresare direct registru se poate emula prin folosirea modului imediat cu valoarea zero în câmpul de deplasament pe 16 biti, iar modul de adresare absolut (direct) se poate emula prin folosirea modului indexat folosind registrul R0 ca registru de index. Astfel se pot folosi toate cele patru moduri de adresare majore desi fizic sunt suportate doar doua.

Memoria lui DLX este adresabila cu o adresa de 32 de biti. Fiind o arhitectura de tip LOAD-STORE toate accesele la / de la memorie se fac prin astfel de transferuri. Suportând tipurile de date descrise anterior, accesele la memorie care implica registri generali GPR se pot face pe octet, semicuvânt si cuvânt. Registrii pentru operanzii în virgula flotanta pot fi cititi sau scrisi din / în memorie folosind cuvinte în simpla si dubla precizie. Toate accesele la memorie trebuie aliniate.

### 2.3.1.4. FORMATUL INSTRUCȚIUNII LA DLX



**Figura 2.1.** Formatul instrucțiunii

Formatul instrucțiunii este proiectat în așa fel încât să asigure o decodificare optimă și o funcționare rapidă a structurii pipeline. Setul de instrucțiuni este ortogonal pe 32 de biți cu un OPCODE primar de 6 biți. Acest format permite un deplasament de 16 biți folosit ca index, constantă imediată sau adresă relativă (la PC) de salt.

### 2.3.1.5. INSTRUCȚIUNILE DLX

DLX-ul suportă o listă de instrucțiuni simple menționate anterior și în plus alte câteva. Există patru clase de bază în care se pot împărți instrucțiunile: LOAD-STORE, ALU, salturi și instrucțiuni de comparație și instrucțiuni în virgulă flotantă. Pentru toate aceste operații oricare dintre cei 32 de registre generali poate fi folosit cu specificarea faptului că încărcarea lui R0 cu orice valoare rămâne fără efect. Valorile în virgulă flotantă simplă

precizie vor ocupa un singur registru în simpla precizie iar un operand în dubla precizie va ocupa o pereche de registri. Conversia între simpla si dubla precizie trebuie facuta în mod explicit.

În continuare se va arata un exemplu pentru instructiunile de tip load – store.

Example instruction	Instruction name	Meaning
LW R1, 30 (R2)	Load word	$\text{Regs}[\text{R1}] \leftarrow_{32} \text{Mem}[30 + \text{Regs}[\text{R2}]]$
LW R1, 1000 (R0)	Load word	$\text{Regs}[\text{R1}] \leftarrow_{32} \text{Mem}[1000 + 0]$
LB R1, 40 (R3)	Load byte	$\text{Regs}[\text{R1}] \leftarrow_{32} (\text{Mem}[40 + \text{Regs}[\text{R3}]]_0)^{24} \text{ ## Mem}[40 + \text{Regs}[\text{R3}]]$
LBU R1, 40 (R3)	Load byte unsigned	$\text{Regs}[\text{R1}] \leftarrow_{32} 0^{24} \text{ ## Mem}[40 + \text{Regs}[\text{R3}]]$
LH R1, 40 (R3)	Load half word	$\text{Regs}[\text{R1}] \leftarrow_{32} (\text{Mem}[40 + \text{Regs}[\text{R3}]]_0)^{16} \text{ ## Mem}[40 + \text{Regs}[\text{R3}]] \text{ ## Mem}[41 + \text{Regs}[\text{R3}]]$
LF F0, 50 (R3)	Load float	$\text{Regs}[\text{F0}] \leftarrow_{32} \text{Mem}[50 + \text{Regs}[\text{R3}]]$
LD F0, 50 (R2)	Load double	$\text{Regs}[\text{F0}] \text{ ## Regs}[\text{F1}] \leftarrow_{64} \text{Mem}[50 + \text{Regs}[\text{R2}]]$
SW 500 (R4), R3	Store word	$\text{Mem}[500 + \text{Regs}[\text{R4}]] \leftarrow_{32} \text{Regs}[\text{R3}]$
SF 40 (R3), F0	Store float	$\text{Mem}[40 + \text{Regs}[\text{R3}]] \leftarrow_{32} \text{Regs}[\text{F0}]$
SD 40 (R3), F0	Store double	$\text{Mem}[40 + \text{Regs}[\text{R3}]] \leftarrow_{32} \text{Regs}[\text{F0}];$ $\text{Mem}[44 + \text{Regs}[\text{R3}]] \leftarrow_{32} \text{Regs}[\text{F1}]$
SH 502 (R2), R3	Store half	$\text{Mem}[502 + \text{Regs}[\text{R2}]] \leftarrow_{16} \text{Regs}[\text{R3}]_{16..31}$
SB 41 (R3), R2	Store byte	$\text{Mem}[41 + \text{Regs}[\text{R3}]] \leftarrow_8 \text{Regs}[\text{R2}]_{24..31}$

**Figura 2.2.** Instructiunile de tip Load-Store la DLX

Pentru o înțelegere mai buna a notatiilor din figura anterioara sa consideram exemplul urmator:

$$\text{Regs}[\text{R10}]_{16..31} \leftarrow_{16} (\text{Mem}[\text{Regs}[\text{R8}]]_0)^8 \text{ ## Mem}[\text{Regs}[\text{R8}]]$$

Semnificatia acestuia este: octetul de la adresa de memorie specificata de continutul lui R8 cu o extensie de semn la 16 biti este încarcat în jumatarea inferioara a lui R10 (jumatarea superioara ramânând neschimbata).

De asemenea, toate instructiunile ALU sunt de tip registru-registru. Acestea includ adunari, scaderi, operatii logice de tip AND, OR, XOR precum si deplasari. De asemenea este posibila folosirea adresarii imediate a tuturor acestor forme împreuna cu o extensie de semn pe 16 biti. Operatiile de tip LHI încarca jumatarea superioara a registrului în timp ce pune la zero pe cea inferioara. Aceasta facilitate ofera posibilitatea construirii unei constante de 32 de biti prin doua instructiuni distincte. Încarcarea unei constante este o simpla adunare a constantei cu registrul R0 si de asemenea o instructiune de tip *move* registru-registru este tot o adunare unde una din surse este registrul R0. Exista si instructiuni de comparare: <, >, ≤, ≥, ≠, =.

Daca conditia este îndeplinita se seteaza 1 în registrul destinatie altfel se înscrie valoarea 0.

Example instruction	Instruction name	Meaning
ADD R1, R2, R3	Add	$\text{Regs}[\text{R1}] \leftarrow \text{Regs}[\text{R2}] + \text{Regs}[\text{R3}]$
ADDI R1, R2, #3	Add immediate	$\text{Regs}[\text{R1}] \leftarrow \text{Regs}[\text{R2}] + 3$
LHI R1, #42	Load high immediate	$\text{Regs}[\text{R1}] \leftarrow 42 \# 0^{16}$
SLLI R1, R2, #5	Shift left logical immediate	$\text{Regs}[\text{R1}] \leftarrow \text{Regs}[\text{R2}] \ll 5$
SLT R1, R2, R3	Set less than	if $(\text{Regs}[\text{R2}] < \text{Regs}[\text{R3}])$ $\text{Regs}[\text{R1}] \leftarrow 1$ else $\text{Regs}[\text{R1}] \leftarrow 0$

**Figura 2.3.** Exemple de instrutiuni aritmetico-logice

Exista, de asemenea, instructiuni de ramificatie si salt. Din cele patru tipuri de instructiuni de salt, doua folosesc un offset pe 26 de biti cu semn care se adauga PC-ului instructiunii urmatoare din secventa determinând astfel adresa destinatie. Salturile sunt tot de doua tipuri: **Plain jump** (salt simplu) si **Jump and link** (salt cu legatura- folosit în cazul instructiunilor de tip CALL- apel de procedura- acesta din urma copiaza adresa de revenire în registrul R31).

Instructiunile de ramificatie sunt toate conditionale. Conditia de ramificatie este specificata de catre instructiunea care testeaza registrul sursa daca e sau nu zero; registrul poate contine o valoare data sau rezultatul unei operatii de comparare. Adresa de ramificatie este specificata ca un offset pe 16 biti cu semn care este adaugat PC-ului instructiunii urmatoare din secventa de program. În figura 2.4 se exemplifica acest tip de instructiuni.

Example instruction	Instruction name	Meaning
J name	Jump	$\text{PC} \leftarrow \text{name}; ((\text{PC}+4) - 2^{25}) \leq \text{name} < ((\text{PC}+4) + 2^{25})$
JAL name	Jump and link	$\text{R31} \leftarrow \text{PC}+4; \text{PC} \leftarrow \text{name}; ((\text{PC}+4) - 2^{25}) \leq \text{name} < ((\text{PC}+4) + 2^{25})$
JALR R2	Jump and link register	$\text{Regs}[\text{R31}] \leftarrow \text{PC}+4; \text{PC} \leftarrow \text{Regs}[\text{R2}]$
JR R3	Jump register	$\text{PC} \leftarrow \text{Regs}[\text{R3}]$
BEQZ R4, name	Branch equal zero	if $(\text{Regs}[\text{R4}] == 0)$ $\text{PC} \leftarrow \text{name}; ((\text{PC}+4) - 2^{15}) \leq \text{name} < ((\text{PC}+4) + 2^{15})$
BNEZ R4, name	Branch not equal zero	if $(\text{Regs}[\text{R4}] != 0)$ $\text{PC} \leftarrow \text{name}; ((\text{PC}+4) - 2^{15}) \leq \text{name} < ((\text{PC}+4) + 2^{15})$

**Figura 2.4.** Instructiuni de salt si ramificatie

Instruction type/opcode	Instruction meaning
<b>Data transfers</b>	<b>Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR</b>
LB, LBU, SB	Load byte, load byte unsigned, store byte
LH, LHU, SH	Load half word, load half word unsigned, store half word
LW, SW	Load word, store word (to/from integer registers)
LF, LD, SF, SD	Load SP float, load DP float, store SP float, store DP float
MOVI2S, MOVSI2I	Move from/to GPR to/from a special register
MOVFP, MOVPD	Copy one FP register or a DP pair to another register or pair
MOVFP2I, MOVI2FP	Move 32 bits from/to FP registers to/from integer registers
<b>Arithmetic/logical</b>	<b>Operations on integer or logical data in GPRs; signed arithmetic trap on overflow</b>
ADD, ADDI, ADDU, ADDUI	Add, add immediate (all immediates are 16 bits); signed and unsigned
SUB, SUBI, SUBU, SUBUI	Subtract, subtract immediate; signed and unsigned
MULT, MULTU, DIV, DIVU	Multiply and divide, signed and unsigned; operands must be FP registers; all operations take and yield 32-bit values
AND, ANDI	And, and immediate
OR, ORI, XOR, XORI	Or, or immediate, exclusive or, exclusive or immediate
LHI	Load high immediate—loads upper half of register with immediate
SLL, SRL, SRA, SLLI, SRLI, SRAI	Shifts: both immediate (S__I) and variable form (S__); shifts are shift left logical, right logical, right arithmetic
S__, S__I	Set conditional: “__” may be LT, GT, LE, GE, EQ, NE
<b>Control</b>	<b>Conditional branches and jumps; PC-relative or through register</b>
BEQZ, BNEZ	Branch GPR equal/not equal to zero; 16-bit offset from PC+4
BFPT, BFPF	Test comparison bit in the FP status register and branch; 16-bit offset from PC+4
J, JR	Jumps: 26-bit offset from PC+4 (J) or target in register (JR)
JAL, JALR	Jump and link: save PC+4 in R31, target is PC-relative (JAL) or a register (JALR)
TRAP	Transfer to operating system at a vectored address
RFE	Return to user code from an exception; restore user mode
<b>Floating point</b>	<b>FP operations on DP and SP formats</b>
ADDD, ADDF	Add DP, SP numbers
SUBD, SUBF	Subtract DP, SP numbers
MULTD, MULTF	Multiply DP, SP floating point
DIVD, DIVF	Divide DP, SP floating point
CVTF2D, CVTF2I, CVTD2F, CTD2I, CVTI2F, CVTI2D	Convert instructions: CVTx2y converts from type x to type y, where x and y are I (integer), D (double precision), or F (single precision). Both operands are FPRs.
__D, __F	DP and SP compares: “__” = LT, GT, LE, GE, EQ, NE; sets bit in FP status register

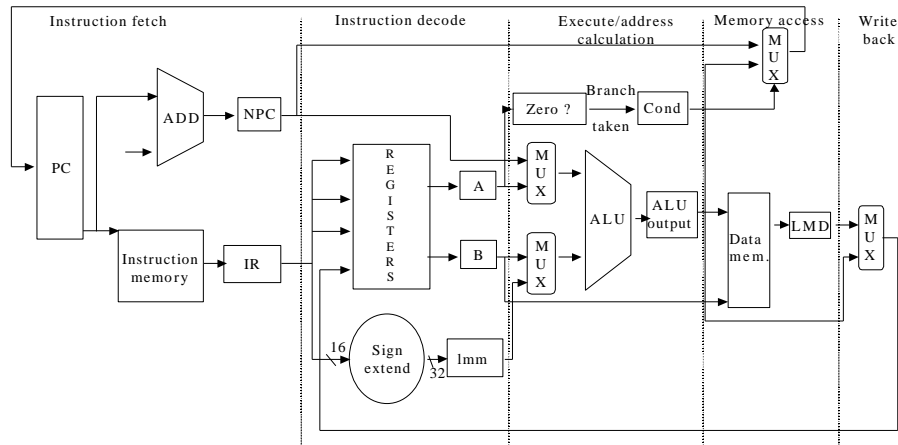
Figura 2.5. Setul complet de instructiuni DLX

### 2.3.1.6. IMPLEMENTAREA DLX

Sa pornim la început cu o schema a DLX-ului care *nu* implementeaza conceptul de pipeline. Ea este menita a ne face sa înțelegem mai bine toate etapele prin care trece o instructiune. În schema de mai jos se prezinta o implementare simpla unde orice instructiune se proceseaza în cel mult patru cicli. Nu este un caz optim de implementare dar este proiectat astfel încât sa realizeze un pas cât mai natural spre cazul *pipeline*. Pentru implementarea de fata s-au folosit o serie de registri temporari care nu apartin arhitecturii



originale; ei au menirea de a simplifica doar trecerea spre o structura pipeline.



**Figura 2.6.** Implementarea simpla DLX

Conform schemei de fata orice instructiune DLX se poate implementa în cel mult cinci cicli. Cei cinci cicli sunt:

#### 📁 Instruction Fetch (IF)

IR  $\leftarrow$  Mem[PC]  
NextPC  $\leftarrow$  PC+4

Obține PC-ul și aduce instrucțiunea respectivă din memorie în registrul IR; incrementează PC-ul cu 4, acesta conținând astfel adresa următoarei instrucțiuni din secvența de program. Registrul IR conține instrucțiunea curentă iar NPC conține PC-ul următor.

#### 📁 Instruction Decode (ID)

A  $\leftarrow$  Regs[IR<sub>6..10</sub>];  
B  $\leftarrow$  Regs[IR<sub>11..15</sub>];  
Imm  $\leftarrow$  ((IR<sub>16</sub>)<sup>16</sup>## IR<sub>16..31</sub>)

Decodifica instrucțiunea și accesează setul de registre pentru a citi conținutul acestora. Aceștia sunt citiți temporar într-un grup de doi registre temporari A și B pentru a putea fi folosiți ulterior în cadrul cicliilor ulterioare. Celor mai puțin semnificativi 16 biți ai registrului IR li se extinde semnul și

vor fi stocati în registrul temporar *Imm* pentru folosirea în urmatorul ciclu. Decodificarea se face în paralel cu citirea registrilor deoarece câmpurile acestora în formatul instructiunii se afla la locatii fixe. Tehnica poarta numele de *fixed-field decoding*. În acest stadiu se calculeaza si extensia de semn în caz ca aceasta este necesara deoarece portiunea imediata dintr-o instructiune se afla localizata mereu în acelasi loc în orice format DLX.

### Execution (EX)

În functie de tipul instructiunii DLX avem urmatoarele patru situatii:

- *Referinta la memorie:*

$$\text{ALUOutput} \leftarrow A + \text{Imm};$$

Se efectueaza adunarea în ALU si se obtine adresa efectiva de memorie care se plaseaza în ALUOutput

- *Instructiune ALU de tip Registru-Registru:*

$$\text{ALUOutput} \leftarrow A \text{ op } B;$$

Se efectueaza operatia specificata de opcode între valorile continute în cei doi registri si rezultatul se depune în registrul temporar ALUOutput.

- *Instructiune ALU de tip Registru-Imediat:*

$$\text{ALUOutput} \leftarrow A \text{ op } \text{Imm};$$

Se efectueaza operatia ALU specificata de opcode între registru si operandul imediat iar rezultatul se plaseaza în registrul temporar ALUOutput.

- *Branch:*

$$\begin{aligned} \text{ALUOutput} &\leftarrow \text{NPC} + \text{Imm}; \\ \text{Cond} &\leftarrow (A \text{ op } 0) \end{aligned}$$

Pentru a se calcula adresa de salt, ALU aduna NPC cu valoarea din IMM care în prealabil a suferit o extensie de semn. Registrul A care a fost citit în ciclul anterior este verificat pentru a se vedea daca saltul a fost facut sau nu. Op este operatorul relational determinat de opcode-ul branch-ului; de exemplu, pentru instructiunea BEQZ acesta este "==".

Arhitectura Load-Store a DLX-ului se bazeaza pe faptul ca adresele efective si ciclul de executie pot fi combinati într-un singur ciclu deoarece

nici o instructiune nu necesita sa calculeze simultan adresa unei date, adresa destinatie a unei instructiuni si sa execute operatia asupra datei. În aceeași categorie sunt incluse și instructiunile de tip Jump.

#### **Memory access/branch completion (MEM)**

În acest caz avem trei tipuri de instructiuni:

- *Referinte la memorie:*

LMD  $\leftarrow$  Mem[ALUOutput]  
sau  
Mem[ALUOutput]  $\leftarrow$  B;

Dacă instructiunea este un Load, data este citită din memorie în registrul LMD (Load Memory Register) iar dacă este o instructiune de tip Store atunci data din registrul B este scrisă în memorie.

- *Branch:*

If (cond)  
PC  $\leftarrow$  ALUOutput  
else  
PC  $\leftarrow$  NPC

Dacă saltul se face PC-ul este înlocuit cu adresa de salt aflată în registrul ALUOutput iar dacă saltul nu se face el este înlocuit cu PC-ul incrementat aflat în registrul NPC.

#### **Write Back (WB)**

- Instructiune ALU registru-registru:

Regs[IR<sub>16..20</sub>]  $\leftarrow$  ALUOutput;

- Instructiune ALU registru –immediat:

Regs[IR<sub>11..15</sub>]  $\leftarrow$  ALUOutput;

- Instructiuni de tip Load:

Regs[IR<sub>11..15</sub>]  $\leftarrow$  LMD;

Scris rezultatul în setul de registre fie că acesta provine din memorie sau din ieșirile ALU. Se observă din figura 2.6 că la sfârșitul fiecărui ciclu, orice valoare calculată și necesară într-un ciclu ulterior, se memorează într-o

locatie care poate fi de memorie, registru general, registrul PC sau unul din registrii temporari (LMD, Imm, A, B, IR, NPC, ALUOutput, sau Cond).

În aceasta implementare instructiunile de salt (branch) sunt singurele care reclama doar patru cicli, celelalte desfasurându-se de-a lungul a cinci cicli masina.

### 2.3.1.7. IMPLEMENTAREA PIPELINE LA DLX

Daca studiem mai cu atentie schema precedenta a DLX-ului se observa ca implementarea conceptului de pipeline se poate face aproape fara nici o modificare la schema de baza. În aceasta varianta fiecare ciclu masina va deveni o etapa în structura pipeline. Aceasta duce la o executie a instructiunilor ca în secventa de mai jos:

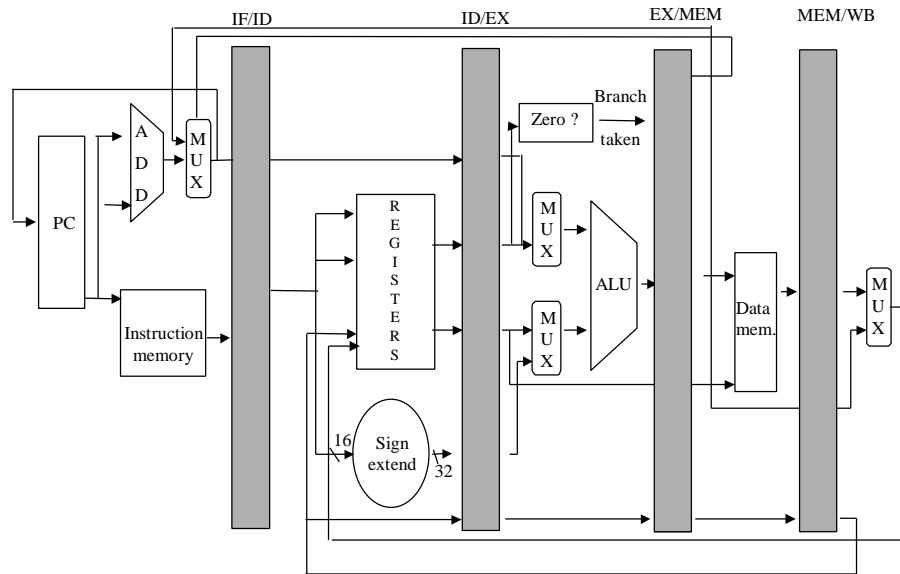
Numarul ciclului									
Nr. Instructiune	1	2	3	4	5	6	7	8	9
Instructiunea i	IF	ID	EX	MEM	WB				
Instructiunea i+1		IF	ID	EX	MEM	WB			
Instructiunea i+2			IF	ID	EX	MEM	WB		
Instructiunea i+3				IF	ID	EX	MEM	WB	
Instructiunea i+4					IF	ID	EX	MEM	WB

Tabelul 2.1.

#### Secventa de instructiuni în diferite faze de procesare

În realitate însa lucrurile sunt un pic mai complexe. Pentru început trebuie sa ne asiguram de faptul ca nu încercam sa utilizam o resursa a sistemului, una din unitatile sale functionale, de mai multe ori în acelasi ciclu în scopuri diferite. De exemplu, o singura unitate de adunare nu poate fi folosita concomitent la calculul unei adrese efective si la efectuarea unei scaderi oarecare (apare hazard structural). Ca si în structura anterioara vom folosi o memorie de instructiuni diferita fata de cea de date care va fi implementata prin cache-uri separate pe instructiuni si date (arhitectura Harvard). Se elimina astfel conflictul care ar putea apare între un fetch instructiune si un acces concomitent la memoria de date. Setul de registri se va folosi de doua ori în acest caz: odata pentru a citi (ID-ul instructiunii) si apoi pentru a scrie rezultatul (WB). Problema scrierii si citirii simultane se va ignora pentru moment.

Datorita structurii pipeline e necesar ca valorile transmise dintr-o etapa în alta sa fie stocate în registrii intermediari numiti registrii pipeline (pipeline latches). Astfel, ajungem la structura urmatoare:



**Figura 2.7.** Implementarea DLX cu structura pipeline

Prin registrele pipeline trec atât date cât și informații de control ale fluxului structurii pipeline. Aceste informații trebuie să fie transferate de la un nivel la altul al structurii pipeline până în momentul în care vor fi folosite. Nu se pot folosi doar registre temporare datorită faptului că se pot suprascrie valori care nu au fost folosite încă. De exemplu, câmpul unui registru necesar unei scrieri pe parcursul unei operații LOAD sau ALU, este luat din latch-ul MEM/WB și nu din IF/ID. Aceasta deoarece se dorește ca operația respectivă să scrie registrul destinație desemnat de operația respectivă și nu unul din registrele care tranzitează de la IF la ID. Acesta din urmă este copiat dintr-un latch în altul până în faza WB unde este folosit. De remarcat este faptul că primele două etape din structura pipeline sunt independente de tipul de instrucțiune deoarece orice instrucțiune este decodificată doar la finele ciclului ID. Activitatea ciclului IF depinde dacă instrucțiunea este un salt care se face sau care nu se face. Dacă saltul se face atunci adresa de salt este folosită ca PC și pentru a calcula adresa următoare. Pentru a controla fluxul datelor în schema de mai sus este necesar să cunoaștem funcționarea celor patru MUX-uri din figura 2.7.

Cele două MUX-uri de la intrările ALU sunt utilizate în funcție de tipul instrucțiunii care este dat de câmpul IR din registrul ID/EX. MUX-ul de la ieșirea sumatorului, înaintea registrului IF/ID este utilizat dacă

instrucțiunea este un salt. Astfel dacă instrucțiunea este un salt, PC-ul este încărcat cu noua valoare a adresei de salt în cazul în care acesta se face. Noua adresa se obține pe baza fazei EX/MEM. Multiplexorul alege dacă să folosească PC-ul curent sau valoarea din EX/MEM NPC (vezi registrul NPC figura 2.6). Acest multiplexor este controlat de către registrul EX/MEM cond (registrul *Cond* în figura 2.6). Cel de-al patrulea multiplexor este setat în funcție de tipul instrucțiunii din ciclul WB: ALU sau LOAD.

Se observă de asemenea necesitatea unui al cincilea multiplexor care să aleagă porțiunea corectă a registrului IR în latch-ul MEM/WB pentru a specifica registrul destinație. Acesta poate avea două variante: registrul-ALU respectiv ALU-imediat sau LOAD.

Stage	Any instruction		
IF	$IF/ID.IR \leftarrow Mem[PC];$ $IF/ID.NPC, PC \leftarrow (if\ EX/MEM.cond\ \{EX/MEM.NPC\}\ else\ \{PC+4\});$		
ID	$ID/EX.A \leftarrow Regs[IF/ID.IR_{6..10}];\ ID/EX.B \leftarrow Regs[IF/ID.IR_{11..15}];$ $ID/EX.NPC \leftarrow IF/ID.NPC;\ ID/EX.IR \leftarrow IF/ID.IR;$ $ID/EX.Imm \leftarrow (IR_{16})^{16}##IR_{16..31};$		
	ALU instruction	Load or store instruction	Branch instruction
EX	$EX/MEM.IR \leftarrow ID/EX.IR;$ $EX/MEM.ALUOutput \leftarrow ID/EX.A\ op\ ID/EX.B;$ or $EX/MEM.ALUOutput \leftarrow ID/EX.A\ op\ ID/EX.Imm;$ $EX/MEM.cond \leftarrow 0;$	$EX/MEM.IR \leftarrow ID/EX.IR$ $EX/MEM.ALUOutput \leftarrow ID/EX.A + ID/EX.Imm;$  $EX/MEM.cond \leftarrow 0;$ $EX/MEM.B \leftarrow ID/EX.B;$	$EX/MEM.ALUOutput \leftarrow ID/EX.NPC + ID/EX.Imm;$  $EX/MEM.cond \leftarrow (ID/EX.A\ op\ 0);$
MEM	$MEM/WB.IR \leftarrow EX/MEM.IR;$ $MEM/WB.ALUOutput \leftarrow EX/MEM.ALUOutput;$	$MEM/WB.IR \leftarrow EX/MEM.IR;$ $MEM/WB.LMD \leftarrow Mem[EX/MEM.ALUOutput];$ or $Mem[EX/MEM.ALUOutput] \leftarrow EX/MEM.B;$	
WB	$Regs[MEM/WB.IR_{16..20}] \leftarrow MEM/WB.ALUOutput;$ or $Regs[MEM/WB.IR_{11..15}] \leftarrow MEM/WB.ALUOutput;$	$Regs[MEM/WB.IR_{11..15}] \leftarrow MEM/WB.LMD;$	

Figura 2.8. Nivelele pipeline DLX

### 2.3.2. PORNIREA SI CONFIGURAREA WINDLX [Grü92]

După lansarea în execuție a simulatorului WinDLX pe ecran vor apărea o fereastră principală și șase ferestre copil minimizate. Pentru reinițializarea procesorului se alege opțiunea *Reset all* din meniul *File*. Pe

ecran va aparea fereastra “ResetDLX” si intentia de resetare a procesorului trebuie confirmata printr-un click pe butonul OK.

WinDLX este capabil sa lucreze cu câteva configuratii. Salvarea configuratiei poate fi facuta la sfârșitul unei sesiuni sau prin activarea optiunii *Save* în meniul de configurare. Poate fi modificata structura, timpul necesar nivelelor pipeline, capacitatea memoriei si alti parametri de control ai simulării. Pentru alegerea configuratiei standard avem de parcurs urmatorii pasi: din meniul *Configuration* selectam optiunea *Floating Point Stages* si verificam existenta urmatoarelor setari:

	Numar	Numar cicli întârziere
Unitate de adunare	1	2
Unitate de înmultire	1	5
Unitate de împartire	1	19

*Tabelul 2.2.*

### **Configuratia implicita a procesorului didactic virtual DLX**

Daca este necesar, schimbarea setarilor se face prin editarea valorilor corecte în câmpurile respective. Validarea noilor valori se face printr-un click pe butonul *OK*.

Setarea dimensiunii memoriei simulate a procesorului DLX se face printr-un click pe optiunea *Memory* din meniul *Configuration*. Aceasta trebuie sa fie 0x8000. Teoretic, capacitatea memoriei poate fi între 512 octeti (0x200) si 16 Mbytes (0x1000000). În practica, ea este limitata la configuratia si managementul memoriei din sistemul MS-Windows. Cu *OK* se revine în fereastra parinte.

La selectarea optiunii *Symbolic addresses* din meniul de configurare, adresele vor fi afisate ca expresii de tipul: "symbol + offset". Altfel, adresele sunt reprezentate ca valori hexazecimale.

Optiunea *Absolute Cycle Count* permite contorizarea absoluta a ciclilor de ceas, începând cu ciclul 0 (de la restartarea procesorului). Altfel, ciclii de tact sunt numerotati relativ, spre exemplu, ciclul curent este 0, iar cei precedenti sunt -1, -2 etc.

Optiunea *Enable Forwarding* valideaza sau invalideaza mecanismul de forwarding. Reamintim ca forwarding-ul (numit de altfel si *bypassing* sau uneori *scurtcircuitare*) este o tehnica hardware simpla de reducere a penalitatilor cauzate de hazardurile de date, rebucând la o intrare, iesirea unei unitati functionale sau, la procesoarele superscalare folosind statii de rezervare [Hen94].

### 2.3.3. ÎNCARCAREA PROGRAMELOR DE TEST

Pentru a putea starta simularea, cel puțin un program trebuie încărcat în memoria principală. Astfel, se selectează opțiunea *Load Code or Data* din meniul *File*. Va apărea o fereastră de dialog din care se pot selecta un număr arbitrar de module cu extensia “.s”, reprezentând coduri sursă DLX. Dacă este confirmată selecția prin apăsarea butonului *Load*, toate modulele selectate vor fi asamblate și codul este scris în memoria DLX. Erorile raportate vor apărea într-o fereastră de dialog separată iar datele scrise în memorie vor fi invalide. După o încărcare cu succes putem reseta procesorul DLX prin intermediul unei ferestre de dialog.

Dacă s-a selectat un fișier nedorit, se inserează respectivul fișier și se apasă butonul *Delete*. Dacă nu se selectează nimic, atunci nici un fișier nu va fi încărcat în memorie.

Posibilitatea de încărcare multiplă a modulelor în același timp permite definirea simbolurilor globale care pot fi folosite în mai multe module. Ulterior, va fi posibil să încărcăm module fără a ține cont de ordinea de încărcare.

Pentru exemplificare vom considera benchmark-ul *fact.s* și rutina necesară *input.s*.

- se execută click pe **fact.s**.
- se apasă butonul *Select*.
- se execută click pe **input.s**.
- se apasă butonul *Select*.
- se apasă butonul *Load*.

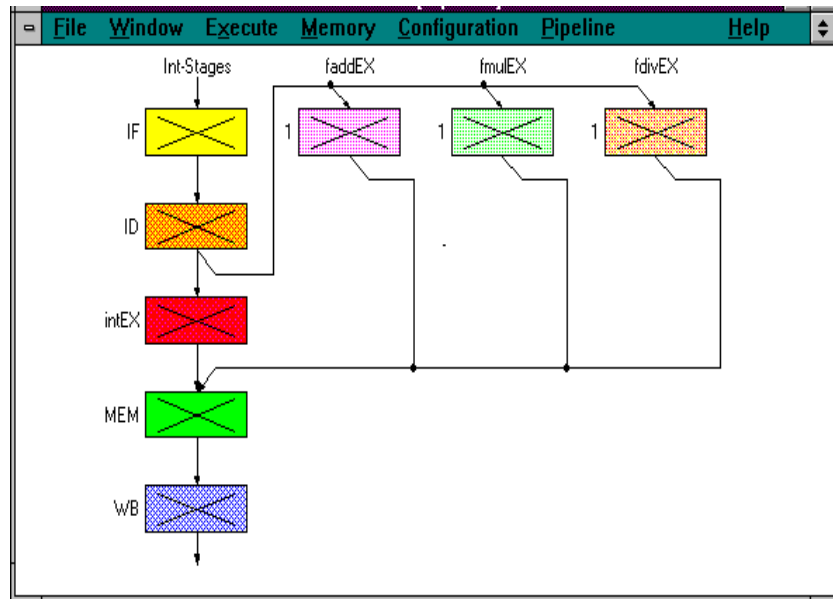
După executia acestor pași simularea poate începe.

### 2.3.4. SIMULAREA PROPRIU-ZISĂ

#### 2.3.4.1. FEREASTRA PIPELINE

În această fereastră sunt descrise cele 5 nivele pipeline ale execuției instrucțiunilor.





**Figura 2.9.** Fereastra pipeline

La nivelul **IF** în fiecare ciclu de tact o noua instrucțiune este extrasă din memorie și depusă în registrul instrucțiunii. PC este incrementat pentru a pointer la următoarea instrucțiune. La nivelul **ID** are loc executia branch-urilor pentru a reduce stagnerile. La nivelul **IntEX** au loc operații aritmetice (mai puțin înmulțire și împărțire) și calcul efectiv al adresei pentru instrucțiunile cu referire la memorie. În **FaddEX** se execută operații de adunare și scădere în simpla sau dubla precizie. Unitățile **FmulEX** și **FdivEX** execută operații de înmulțire și împărțire atât asupra întregilor (cu semn și fără semn) cât și asupra flotantilor în simpla și dubla precizie. Singurele instrucțiuni ale DLX active la nivelul **MEM** sunt LOAD și STORE. Calculul adresei de citire sau scriere s-a efectuat în nivelul anterior. Rezultatul este scris în registrul în nivelul **WB**. El provine fie din memorie fie dintr-o operație ALU. Operația de scriere se face în prima jumătate a ciclului de ceas, astfel încât o instrucțiune aflată pe nivelul ID să poată citi rezultatul în a doua jumătate a ciclului de ceas, eliminând necesitatea de *bypassing* a rezultatului instrucțiunii.

#### 2.3.4.2. FEREASTRA COD

Instrucțiunile DLX încărcate în memorie și adresele lor sunt afișate (în hexazecimal) și dezasamblate în fereastra de Cod.

\$TEXT	0x20011000	addi r1,r0,0x1000
main+0x4	0x0c00003c	jal InputUnsigned

Stabilirea unui punct de întrerupere pe o instructiune se face marcând cu Bxx respectiva instructiune, unde xx este tipul întreruperii. Daca o instructiune se afla pe un nivel pipeline de executie culoarea caracteristica a nivelului pipeline este folosita ca o culoare de fond pentru instructiune si este etichetata adecvat.

Defilarea sus/jos prin memorie se face folosind tastele cu sageti sau PgUp/PgDown. Dupa executia unui sau mai multor pasi ai codului DLX, ultimele instructiuni executate sunt afisate.

Meniul Code contine urmatoarele optiuni:

- *From Address* - adresa de început a codului ce va fi afisat în fereastra de cod; poate fi introdusa printr-o cutie de dialog. Specificarea adresei se face printr-o valoare întreaga, operatori sau simboluri.
- *Set Breakpoint* - pe instructiunea selectata din fereastra de Cod se stabileste un punct de întrerupere.
- *Delete Breakpoint* - punctul de întrerupere de pe instructiunea selectata este sters.

Pentru exemplificare, vom începe simularea secventei de instructiuni descrisa anterior.

Din meniul *Execute* se alege optiunea *Single Cycle*. Tasta F7 are acelasi efect. Vom observa ca prima linie din fereastra cu adresa \$TEXT este colorata galben. Apasând F7 vom avansa în simulare cu un pas. Aceasta va determina schimbarea culorii primei linii în portocaliu si urmatoarea linie este colorata galben. Aceste culori arata nivelul pipeline în care se afla fiecare instructiune. Astfel, instructiunea **jal InputUnsigned** este în nivelul IF iar precedenta **addi r1, r0, 0x1000** este în nivelul secund ID. Celelalte nivele sunt marcate cu o cruce, aratând ca nu se proceseaza nici o operatie semnificativa în ele. Tastând F7 în continuare, culorile din fereastra de cod vor fi rearanjate, introducând rosu pentru al treilea nivel intEX. Nivelul MEM va fi colorat în verde, iar WB în albastru.

#### 2.3.4.3. DIAGRAMA CICLILOR PROCESORULUI

În aceasta diagrama este afisat tot ce se întâmpla în fiecare ciclu si în fiecare nivel al pipe-ului. Fiecare coloana reprezinta starea pipe-ului în fiecare ciclu de tact. Starea curenta a pipe-ului (ultima coloana) este colorata

cu gri. Executând dublu-click pe o instrucțiune selectată vom obține mai multe detalii despre instrucțiunea respectivă, în fiecare ciclu de tact (instrucțiunea dezasamblată, adresa ei, codificarea în hexazecimal, starea de execuție - nivelul pipeline, modul în care s-a încheiat - normal, cu eroare, întrerupt, numărul ciclului când a început execuția, numărul de cicli de execuție etc).

Meniul Diagrama ciclului de tact conține următoarele opțiuni:

- *Display Forwarding*
- *Display Cause of Stalls*
- *Delete History*
- *Set History Length*

În continuare vom prezenta detalii privind stagnări sau forwarding în execuție. Stagnările sunt reprezentate prin chenare colorate în culoarea corespunzătoare nivelului pipeline în care se stagnează. Distingem următoarele tipuri de stagnări:

- **R-Stall** - stagnare datorată unui hazard RAW. O săgeată roșie închisă va indica instrucțiunea care cauzează stagnarea (după al cărui operand se așteaptă).
- **T-Stall** - stagnare datorată unei excepții Trap. O instrucțiune Trap rămâne în nivelul IF, până când nici o altă instrucțiune nu se mai află în pipe.
- **W-Stall** - stagnare datorată unui hazard WAW. Dependenta dintre instrucțiunile implicate este sugerată tot printr-o săgeată de culoare roșu închis.
- **S-Stall** - hazard structural. Nu există suficiente resurse pentru deservirea instrucțiunilor.
- **Stall** - dacă o instrucțiune în flotant se află în nivelul MEM, următoarea instrucțiune va fi oprită în nivelul intEX, etichetată cu "Stall", până la extragerea datelor necesare din memorie.

Deși este posibilă apariția oricărui tip de hazard menționat mai sus, pe cele trei benchmark-uri studiate nu vom întâlni hazarduri structurale sau de tipul WAW, în concluzie fiind suficiente doar câte o unitate de execuție pentru fiecare nivel al pipe-ului. Totuși, pentru programe de test care utilizează succesiv mai multe instrucțiuni de (adunare/scadere, înmulțire sau împărțire) în flotant pot apărea hazarduri structurale pentru anumite configurații arhitecturale.

*Exemplu:*

Considerăm următoarea configurație arhitecturală, fără mecanism de forwarding:

	Numar	Cicli Întârziere
Unitate de adunare	1	2
Unitate de înmulțire	1	10
Unitate de împartire	1	19

si urmatoarea secventa de instructiuni:

*i1: multd f2, f2, f0*

*i2: multd f4, f4, f0*

În urma executiei acestei secvente de instructiuni vom observa aparitia unor întârzieri în procesare datorate hazardului structural generat de prezenta celei de a doua instructiuni de înmulțire.

Hazardurile de tip WAW pot apare doar în cazul unui scheduling soft aplicat benchmark-urilor sau daca în vreunul din programele de test exista urmatoarea secventa de instructiuni:

*i1: ins\_float R2, R2, R0 se executa în 30 ciclui*

*i2: ins\_integer R2, R4, R6 se executa în 5 ciclui*

Instructiunea pe întregi (*i1*) se încheie mai repede decât cea în flotant (*i2*) si modifica continutul registrului *R2* înainte ca operatia în flotant sa se încheie; rezultatul ramas în *R2*, dupa executia celor doua instructiuni este astfel, eronat.

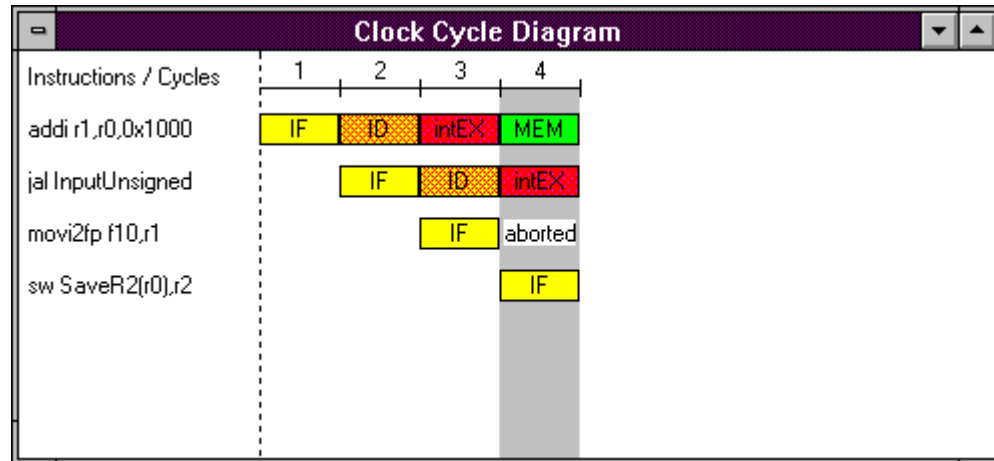
Daca optiunea *Display Forwarding* este validata, forwarding-ul este reprezentat printr-o sageata verde indicând spre sursa si destinatia mecanismului. Istoria contine informatii despre instructiunile deja încheiate. Se poate specifica dimensiunea istoriei (în instructiuni, de la 0 la 100). Daca aceasta este 0, înseamna ca doar instructiunea care se executa în pipe este afisata în diagrama ciclului de tact.

Pentru o înțelegere mai buna, vom exemplifica în continuare pe diagrama ciclului de tact de mai jos.

În diagrama din figura 2.10. se observa ca simularea este în al patrulea ciclu, prima instructiune este în nivelul MEM, a doua în intEX si a patra în IF. A treia instructiune este întrerupta. Motivatia consta în faptul ca a doua instructiune **jal**, este un salt (apel) neconditionat. Acest lucru e cunoscut abia dupa cel de-al treilea ciclu, dupa ce instructiunea de salt a fost decodificata. În acest timp, instructiunea imediat urmatoare celei de salt (**movi2fp**) a trecut de faza IF, însa instructiunea care se va executa efectiv dupa instructiunea de salt se afla la alta adresa. Astfel, executia instructiunii **movi2f** trebuie întrerupta, lasând loc gol în pipe.

Saltul se va face la eticheta *InputUnsigned*. Pentru a gasi valoarea adresei simbolice se alege optiunea *Symbols* din meniul *Memory*. Fereastra

care apare arata corespondenta dintre simbolurile folosite si valorile numerice ale adresei. Este preferabila sortarea dupa *nume* decât dupa *valoare* a adreselor, pentru o regasire mai usoara a acestora. Caracterul ‘G’ scris dupa valoare semnifica un simbol global, iar ‘L’ un simbol local. Astfel, *InputUnsigned* este un simbol global, existent în modulul Input.s, semnificând valoarea adresei 0x144.



**Figura 2.10.** Diagrama ciclului de tact

#### 2. 3.4.4. FEREASTRA BREAKPOINT

În fereastra Breakpoint, putem vizualiza, stabili, schimba si sterge punctele de întrerupere. E posibila stabilirea a maxim 20 de puncte de întrerupere. Pentru manevrarea unui punct de întrerupere, se selecteaza breakpoint-ul respectiv (prin tastele cu sageti sau mouse), iar printr-un dublu-click este permisa modificarea punctului de întrerupere sau a caracteristicilor acestuia. Meniul Breakpoint contine urmatoarele optiuni:

- *Set*
- *Delete*
- *Delete All*
- *Change*

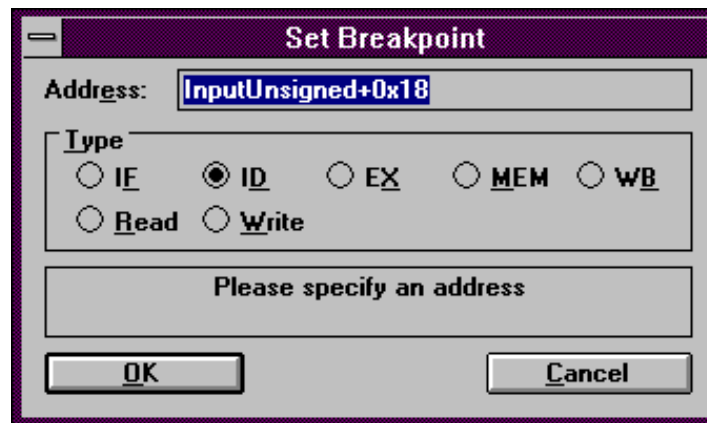
Dupa activarea optiunii Set, pe ecran apare o caseta de dialog prin intermediul careia se stabilesc:

- *adresa* punctului de întrerupere; poate fi o expresie întreaga arbitrara (poate include simboluri sau operatori). Adresa trebuie sa fie multiplu de 4 (aliniere la cuvânt de 4 octeti). În caz contrar, adresa e convertita la multiplu de 4.

- *tipul* punctului de întrerupere; poate fi IF, ID, EX, MEM sau WB, însemnând ca simularea s-a întrerupt daca instructiunea specificata a ajuns în nivelul pipeline specificat de tip. Un breakpoint pe o citire (în cazul instructiunilor Load/Store) se realizeaza în momentul în care în memorie am ajuns pe cuvântul (sau o parte a lui) specificat de adresa punctului de întrerupere. Un breakpoint pe o scriere (în cazul instructiunilor Store sau exceptiilor Trap) se realizeaza când un acces de scriere se face pe cuvântul (sau o parte a lui) specificat de adresa punctului de întrerupere.

Valorile introduse sunt implicite pentru data urmatoare când se va activa optiunea Set. Stergerea unui punct de întrerupere se face selectând optiunea Delete, iar stergerea tuturor punctelor de întrerupere se face cu varianta Delete All. În acest caz e necesara confirmarea actiunii. Prin optiunea Change poate fi modificata adresa sau tipul punctului de întrerupere selectat prin intermediul casetei de dialog.

Exemplificare:



**Figura 2.11.** Fereastra Breakpoint

Când examinam fereastra cod observam instructiuni similare care se repeta. Astfel, parcurgerea pas cu pas a codului sursa al benchmark-ului devine plictisitor si inefficient. Pentru accelerarea acestui proces vom introduce puncte de întrerupere.

În fereastra de cod vom selecta linia cu adresa 0x0000015c (instructiunea **trap 0x5** – instructiune analoaga întreruperilor **int n** de la procesoarele INTEL 80x86). Instructiunea reprezinta un apel sistem în urma caruia pe ecran va fi afisat un mesaj. Se selecteaza optiunea *Set Breakpoint*, iar pe ecran apare o fereastra identica cu cea din figura 2.11. Tipul implicit al breakpoint-ului este ID. Se confirma cu butonul OK.

În consecința, pe linia din fereastra de cod pe care se afla instrucțiunea **trap 0x5**, va apărea expresia “**BID**”, semnificând că programul se întrerupe când instrucțiunea respectivă este în faza de decodificare. Pentru examinarea punctelor de întrerupere deja definite se execută click pe iconița *Breakpoint*. Declansarea simulării se face selectând opțiunea *Run* din meniul *Execute* sau tasta F5. O fereastră ne va informa că ne aflăm pe nivelul ID și s-a atins primul punct de întrerupere. Dacă studiem diagrama ciclului de ceas vom observa că simularea este în ciclul 14, iar instrucțiunea **trap 0x5** se afla în stagnare.



Motivația constă în faptul că nivelele pipeline ale DLX sunt golite de fiecare dată când se întâlnește o instrucțiune trap, pentru a evita toate posibilitățile problemei. Dacă vom activa fereastra *Display DLX-IO* din meniul *Execute*, în fereastră va apărea mesajul: “An integer value:”.

#### 2.3.4.5. FEREASTRA REGISTRU

În această fereastră sunt afișate valorile tuturor registrilor. Este posibilă totodată și modificarea conținutului registrilor generali - GPR, a registrilor flotanti - FPR, a PC-ului și registrului de stare - FPSR, dar nu și conținutul registrilor speciali. Distingem mai multe tipuri de afișare și reprezentare a conținutului registrilor: hexazecimal, decimal, flotant simplă și dublă precizie. Selecția unui registru se face prin dublu click pe numele acestuia.

Procesorul DLX are următorii registri vizibili în programe:

- GPR (R0..R31) - 32 registri de uz general pe 32 de biți; R0 (ca și la procesorul MIPS R2000) este întotdeauna cablat la 0.
- FPR - set de registri flotanti care pot fi folosiți ca 32 de registri în simplă precizie (F0 ÷ F31), sau perechi de registri par-impar în dublă precizie (de la D0 la D30).
- FPSR - registru de stare folosit atât pentru comparații cât și pentru excepții în virgula flotantă. Toate instrucțiunile de transfer în/din registrul de stare lucrează cu registrii de uz generali. Instrucțiunile de salt condiționat testează bitul de comparație din registrul FPSR.
- PC – program counter-ul conține întotdeauna adresa următoarei instrucțiuni ce urmează a fi extrasă din memorie. Instrucțiunile de

ramificatie (salturi conditionate sau neconditionate) pot altera continutul PC.

De asemenea, DLX contine si registri interni, care nu sunt vizibili programelor DLX.

- IMAR - registrul adresei de memorie a instructiunii; este initializat cu continutul program counter-ului în timpul fazei IF. Spre deosebire de PC, acest registru este în conexiune directa cu memoria sistem.
- IR - registrul instructiunii; în timpul fazei IF acest registru se încarca cu urmatoarea instructiune ce va fi executata.
- A, B - registrii operanzi ai ALU; în faza de decodificare cei doi registri care sunt cititi si trimisi unitatii aritmetico-logice. WinDLX prezinta, de asemenea, doi pseudoregistrii AHI si BHI care contin cei mai semnificativi 32 de biti ai registrilor cu valori flotante în dubla precizie.
- BTA - registrul de adresa a tinteii branch-urilor; în nivelul ID se calculeaza adresa tinta a instructiunilor branch sau jump si se scrie în acest registru. Daca branch-ul este *taken* (se executa salt) adresa se încarca si în program counter.
- ALU - registrul rezultat al operatiilor aritmetico-logice; WinDLX prezinta un pseudoregistru ALUHI ce are acelasi rol ca si registrele AHI si BHI.
- DMAR - registrul adresa de memorie a datei; adresa datelor în cazul referintelor la memorie este transferata în acest registru. Accesul la memorie se face în timpul fazei MEM.
- SDR - registrul de memorare a datei; data care urmeaza a fi scrisa în memorie este transferata în acest registru. WinDLX prezinta, de asemenea, pseudoregistru SDRHI.
- LDR - registrul de încarcare a datei; data citita din memorie se încarca în acest registru. Registrul LDRHI este destinat operatiilor cu date în flotant dubla precizie.

Pentru a continua simularea activam fereastra de cod si defilam prin ea pâna la linia cu adresa 0x00000194, la instructiunea **lw r2, SaveR2(r0)**. Punem un punct de întrerupere (**Code / Set Breakpoint / Ok**) si pe aceasta linie. Totodata stabilim un punct de întrerupere si pe instructiunea de la adresa 0x000001a4 **jr r31** ( $PC \leftarrow (r31)$ ). Continuând simularea cu F5, pe ecran va apare fereastra DLX Standard-IO în care se asteapta introducerea de la tastatura a unui întreg. Dupa citirea unei valori întregi de la tastatura



(de exemplu: 20) si confirmarea cu Enter, simularea continua pâna se ajunge la cel de-al doilea breakpoint. Examinand simularea în ciclii 52 ÷ 56, în diagrama ciclului de ceas observam aparitia unor sageti rosii si verzi între instructiuni. Sageata rosie implica necesitatea unei stagnari datorate unui hazard de tip RAW (o instructiune are nevoie de rezultatul unei instructiuni precedente care nu este înca cunoscut). Sagetile verzi sugereaza folosirea mecanismului de forwarding (folosirea rezultatului unei instructiuni înainte ca acesta sa fie scris în registrul general destinatie).

În acest moment vom examina continutul registrilor generali activând fereastra Registru. Cu F5 continuam simularea pâna la urmatorul punct de întrerupere. Daca dorim ca simularea sa avanseze fara stabilirea unor noi puncte de întrerupere selectam optiunea Multiple Cycles din meniul Execute sau simplu F8. În fereastra nou creata se introduce numarul de cicli (de exemplu: 17) care se vor executa fara întrerupere. Defilam apoi prin diagrama ciclului de ceas pâna la cicli instructiune de la 72 la 78. Doua instructiuni în flotant dubla precizie (înmultire si scadere) sunt executate fiecare în unitati separate de executie în timpul fazei (EX), dar ambele necesita mai mult de un ciclu pentru executie. Astfel, instructiunea urmatoare acestora (j Fact.Loop) poate fi adusa din memorie, decodificata si executata, dar apoi trebuie sa astepte un ciclu pentru a permite instructiunii *subd* sa încheie faza MEM.

#### 2.3.4.6. FEREASTRA STATISTICA

Fereastra Statistica contine rezultatele statistice obtinute în urma simularii. Datele sunt clasificate în câteva grupuri si au urmatoarele semnificatii:

- Numarul total de cicli executati.
- Numarul de instructiuni care au fost deja transmise nivelului ID.
- Numarul de instructiuni curent executate în pipe.

Meniul Statistica contine unele optiuni pentru a controla afisarea informatiilor în fereastra Statistica:

- *Display* – cu optiunile:
  - *Hardware*
  - *Stalls*
  - *Conditional Branches*
  - *Load/Store-Instructions*
  - *Floating point stages instructions*
  - *Traps*

- All

- *Detail info*
- *Reset*

Optiunea *Display Hardware* permite afisarea urmatoarelor informatii despre configuratia hardware a DLX:

- capacitatea memoriei (în octeti).
- numarul de unitati de executie care opereaza asupra datelor flotante precum si latenta lor.
- starea mecanismului de forwarding (validat sau nu).

Daca optiunea *Display Stalls* este activata vor fi afisate urmatoarele date statistice (relative sau absolute):

- numarul de stagnari datorate hazardurilor de date RAW. Daca mecanismul de forwarding este validat si optiunea *Detail info* este activata, numarul de stagnari va fi împartite dupa tipul instructiunii cauzatoare de stagnare, în: numar de instructiuni de load, numar de instructiuni de salt (branch/jump), numar de instructiuni în virgula mobila.
- numarul de stagnari datorate hazardurilor de date WAW.
- numarul de stagnari datorate hazardurilor structurale înainte de intrarea în nivelul de executie cu numere flotante.
- numarul de stagnari datorate executiei instructiunilor de salt conditionat (branch-uri taken).
- numarul de stagnari datorate instructiunilor trap.

Optiunea *Display Conditional Branches* determina afisarea informatiilor (relative sau absolute) referitoare la instructiunile de salt conditionat.

- numarul de salturi conditionate. Daca optiunea *Detail info* este activata, informatia este separata în numar de instructiuni de salt conditionat care se fac si numar de instructiuni de salt care nu se fac.

Selectând *Display Load/Store* în fereastra vor fi afisate numarul de instructiuni Load si Store executate. Activând si *Detail info* informatia va fi împartita în doua: numar instructiuni Load si numar instructiuni Store.

Daca optiunea *Display Floating Point Stage Instructions* este activa, se pot obtine informatii despre numarul total de instructiuni executate care au folosit nivelele de executie în virgula flotanta (faddEX, fmulEX oder fdivEX). Daca e validata optiunea *Detail info*, în fereastra informatia va apare segmentat în:

- numarul de instructiuni care au trecut prin nivelul faddEX.
- numarul de instructiuni care au trecut prin nivelul fmulEX.
- numarul de instructiuni care au trecut prin nivelul fdivEX.

Numarul de instructiuni Trap executate este afisat cu ajutorul optiunii *Display Traps*.

Optiunea *Display All* implica afisarea tuturor grupurilor de date statistice.

Selectând *Detail Info* se permite afisarea a unor informatii detaliate în fereastra Statistica.

Pentru o reluare a testelor este necesara resetarea tuturor parametrilor din fereastra Statistica. De asemenea, la pornirea sau restartarea procesorului este necesara o resetare a parametrilor statistici. Acest lucru se face selectând optiunea *Reset*. Ultimul ciclu de ceas simulat devine 0. Istoria si continutul pipeline-ului vor ramâne neschimbate.

Continuam exemplificarea pe programul de test, urmarind la finele executiei programului actualizarea parametrilor în fereastra Statistica.

Executia programului continua cu F5. Pe ecran va apare o caseta cu mesajul "*Trap #0 occurred*", arătând ca instructiunea **trap 0** a fost executata. Instructiunea **trap 0** constituie apelul sistem (întreruperea) folosit pentru a sugera terminarea programului. Se confirma cu *OK* si se maximizeaza fereastra Statistica.

Fereastra Statistica este extrem de folositoare pentru a compara efectele modificarilor de configuratie asupra performantei procesorului (numar total cicli de executie). Vom examina avantajul introducerii mecanismului de forwarding. Folosind acest mecanism am obtinut urmatoarele rezultate: numarul total de cicli – 215, stagnari datorate (RAW – 17, Control – 25, Trap - 12), în total 54 de cicli. Închidem fereastra Statistica si vom reconfigura hardware procesorul. Pentru dezactivarea forwarding-ului se inhiba optiunea *Enable Forwarding*. Urmatorul avertisment care va apare pe ecran: "*OK resets automatically the processor! Disable Forwarding?*" trebuie confirmat cu *OK*. Stergem toate punctele de întrerupere cu optiunea *Delete All* din meniul *Breakpoint*. Executam simularea benchmark-ului fara întrerupere astfel: F5, 20 (numarul al carui factorial se calculeaza) Enter si OK pâna se executa instructiunea **trap 0**. Prin reexaminarea ferestrei Statistica observam ca, numarul de stagnari datorate instructiunilor de salt si *trap* ramâne acelasi dar numarul de stagnari datorate hazardurilor RAW creste de la 17 la 53, determinând cresterea numarului total de cicli de simulare la 236 fata de 215. Cunoscând aceasta informatie putem calcula *speed-up-ul* (câstigul de performanta obtinut prin forwarding). Raportul  $236 / 215 = 1.098$  este interpretat astfel: DLX cu forwarding este cu 9.8% mai rapid decât fara forwarding, pe benchmark-ul **fact.s**.

### 2.3.5. APELURI SISTEM

Pentru citirea de la tastatura si afisarea în fereastra “DLX I/O” a sirurilor de caractere, valorilor întregi, caractere, pentru terminarea programului, sunt utilizate apeluri sistem, un mic set de servicii ale sistemului de operare prin instructiuni – **trap** (*syscall* la procesorul MIPS, *int21* la Intel x86).

Instructiunile **trap** sunt similare apelurilor sistem UNIX/DOS, respectiv functiilor din biblioteca C - `open()`, `close()`, `read()`, `write()` si `printf()`. Orice fisier înainte de a fi prelucrat trebuie deschis. Functia `open()` determina deschiderea unui fisier existent si returneaza o valoare întreaga pozitiva numita *descriptorul de fisier*. Acesta identifica în continuare fisierul respectiv în toate operatiile realizate asupra lui. Functia `read()` primeste ca si parametru descriptorul unui fisier deschis în prealabil prin intermediul functiei `open` si realizeaza operatia de citire. Ea returneaza numarul de octeti cititi efectiv din fisier. Functia `write()` este similara cu `read()`, doar ca realizeaza transferul de date în sens invers, din memoria principala în fisier. La terminarea prelucrării unui fisier acesta trebuie închis cu ajutorul functiei `close()`. Aceasta returneaza 0 la o închidere reusita si -1 în caz de eroare.

Descriptorii de fisier 0,1 si 2 sunt rezervati fisierelor standard - `stdin`, `stdout` and `stderr`. Intrarile si iesirile DLX pot fi controlate cu acesti descriptori. Adresa parametrilor necesari apelurilor sistem trebuie încarcata în registrul R14. Toti parametrii trebuie sa fie pe 32 de biti, exceptie făcând registrii flotanti dubla precizie care sunt pe 64 de biti. Sirurile de caractere sunt referite cu adresa lor de început. Rezultatul apelului va fi returnat în registrul R1. Daca apare vreo eroare în timpul apelului sistem, registrul R1 este setat cu valoarea -1 si, daca simbolul “\_errno” este setat la valoarea A atunci se returneaza un cod de eroare la adresa de memorie A iar simularea continua, altfel simularea este întrerupta [Neg93].

Serviciul	Cod apel sistem	Argumentele	Rezultatul
Open file	1	1. Numele fisierului: adresa unui sir încheiat cu terminatorul null, care contine calea spre fisierul care va fi deschis. 2. Mod: modul de deschidere a fisierului. 3. Flaguri suplimentare: drepturi	Deschiderea unui fisier pentru citire sau scriere. Fisierele deschise sunt automat închise la resetarea DLX sau oprirea WinDLX. Descriptorul de fisier este

		de executie	returnat în registrul R1.
Close file	2	1. Descriptorul fisierului ce urmeaza a fi închis.	Un fisier descris anterior cu Trap 1 este închis. În caz de succes în registrul R1 se returneaza 0, altfel -1.
Read block from file	3	1. Descriptorul fisierului din care se citește blocul. 2. Adresa zonei unde se va stoca blocul citit. 3. Dimensiunea blocului ce va fi citit (în octeti)	Un bloc dintr-un fisier sau o linie de la intrare (stdin) poate fi citit cu acest apel sistem. Numarul actual de octeti citit este returnat în registrul R1..
Write block to file	4	1. Descriptorul de fisier în care se va scrie. 2. Adresa blocului ce va fi scris. 3. Dimensiunea blocului (în octeti).	Un bloc poate fi scris în memorie sau la iesirea standard. Numarul de octeti efectiv scrisi este returnat în registrul R1.
Formatted Output to Standard Output	5	1. Formatul de afisare al string-ului. 2. Argumente în conformitate cu formatul de afisare.	Echivalentul functiei de biblioteca printf(). Numarul de octeti transferati la iesire (stdout) este returnat în registrul R1.
Exit	0	-	Încheierea programului.

Tabelul 2.3.

### Apelurile sistem deservite de procesorul DLX

Detalii suplimentare legate de modurile de deschidere a fisierelor, drepturilor de acces si executie asupra acestora pot fi gasite studiind functiile de biblioteca C [Neg93].

## 2.4. PROBLEME DE LABORATOR PROPUSE SPRE REZOLVARE

### 2.4.1. NOTE EXPLICATIVE REFERITOARE LA BENCHMARK-URILE *MIN\_SUMA\_MAX.S* RESPECTIVE *INVERSAREA UNUI NUMAR CITIT DE LA TASTATURA*

Benchmark-ul *min\_suma\_max.s* este un program de test simplu, care calculeaza si afiseaza pe ecran minimul, maximul si suma unui sir de

numere pozitive citite de la tastatura (dimensiunea sirului este introdusa tot de la tastatura cu ajutorul modului *Input.s* si memorat în registrul  $R_1$  al procesorului DLX).

Se vor observa cu ajutorul simulatorului prin intermediul ferestrelor **Pipeline, Registru, Cod, Diagrama ciclului de tact, Statistica**, valorile din registri, continutul locatiilor de memorie, instructiunile efectiv executate în urma rularii pas cu pas sau în mod continuu a programului, nivelul pipeline în care se afla instructiunile în fiecare ciclu de tact, precum si rezultatele simulării.

```

;**** WINDLX: Calculul minimului/maximului/sumei unui sir de numere ****
;-----
;Programul necesita subrutina INPUT.s
; Rezultatele sunt afisate pe consola
;-----
.data
Prompt:
.asciiz "Introduceti numarul de elemente n = "
Prompt1:
.asciiz "Introduceti elementul = "
PrintfFormat1:
.asciiz "Maximul este %d :\n"
PrintfFormat2:
.asciiz "Minimul este %d :\n"
PrintfFormat3:
.asciiz "Suma este %d :\n"
PrintfMesaj_err:
.asciiz "\nNumar negativ! Reluati!\n"

.align 2
PrintfPar1:
.word PrintfFormat1 ; adresa formatului de afisare maxim: mesaj +
                    ; valoare

PrintfValue1:
.space 4            ; spatiu de stocare a valorii maximului

PrintfPar2:
.word PrintfFormat2 ; adresa formatului de afisare minim: mesaj +
                    ; valoare

PrintfValue2:
.space 4            ; spatiu de stocare a valorii minimului

PrintfPar3:

```

```

        .word PrintfFormat3 ; adresa formatului de afisare suma: mesaj +
                                ; valoare
PrintfValue3:
        .space 4             ; spatiu de stocare a valorii sumei
PrintfErr1:
        .word PrintfMesaj_err ; adresa formatului de afisare mesaj în cazul
                                ; tastarii de la intrare a unui numar negativ
        .text
        .global main
main:
        addi    r1,r0,Prompt  ; citire de la tastatura a dimensiunii sirului
        jal     InputUnsigned ; r1 = retine elementul current al sirului
        sle     r15,r1,r0
        bnez    r15, mesaj_err ; daca s-a tastat un numar negativ se afiseaza
                                ; mesaj urmat de reluarea introducerii
                                ; numarului de la tastatura
        addi    r2,r1,0        ; r2 = preia numarul de elemente al sirului
        addi    r3,r0,-32000   ; r3 = va stoca maximul
        addi    r4,r0,32000    ; r4 = va stoca minimul
        addi    r6,r0,0        ; r6 = suma
loop:
        addi    r1,r0,Prompt1  ; începe citirea sirului element cu element
        jal     InputUnsigned ; r1 = retine elementul current al sirului
        sle     r15,r1,r0      ; daca s-a tastat un numar negativ se afiseaza
                                ; mesaj
        bnez    r15, mesaj_err_el ; urmat de reluarea introducerii
                                ; numarului de la tastatura
        slt     r5,r1,r4        ; daca r1>r4 atunci minimul va deveni ultimul
                                ; element citit
        beqz    r5,maxim        ; altfel se continua executia fireasca
                                ; comparându-se elementul curent cu maximul
                                ; din sir obtinut pâna în acest moment
        addi    r4,r1,0        ; minimul devine ultimul element citit r4 ← r1
maxim:
        sgt     r5,r1,r3        ; daca r1>r3 atunci maximul va deveni ultimul
                                ; element citit
        beqz    r5,suma        ; altfel se continua executia fireasca
                                ; calculându-se suma partiala a elementelor
                                ; din sir citite pâna în acest moment
        addi    r3,r1,0        ; maximul devine ultimul element citit r3 ← r1

```

suma:

```

add  r6,r6,r1      ; r6 ← r6 + r1
subi  r2,r2,1      ; r2 ← r2 - 1: decrementare contor elemente
                        ; citite
bnez  r2,loop      ; mai sunt elemente de citit de la tastatura ?
sw    PrintfValue1,r3 ; salvare parametrii afisare pentru maxim
addi  r14,r0,PrintfPar1
trap  5            ; afisare maxim

sw    PrintfValue2,r4 ; salvare parametrii afisare pentru minim
addi  r14,r0,PrintfPar2
trap  5            ; afisare minim

sw    PrintfValue3,r6 ; salvare parametrii afisare pentru suma
addi  r14,r0,PrintfPar3
trap  5            ; afisare suma
j      gata

```

mesaj\_err:

```

sw    PrintfValue3,r6 ; salvare parametrii afisare pentru mesaj de
                        ; eroare
addi  r14,r0,PrintfErr1
trap  5            ; afisare mesaj eroare ' Dimensiune sir negativa !'
j      main        ; reluare cu tastarea dimensiunii sirului

```

mesaj\_err\_el:

```

sw    PrintfValue3,r6 ; salvare parametrii afisare pentru un al doilea
                        ; mesaj eroare
addi  r14,r0,PrintfErr1
trap  5            ; afisare mesaj eroare 'Element negativ!'
j      loop        ; reluare cu tastarea elementului curent

```

gata:

```

trap  0            ; încheiere program

```

---

### Comentariul subrutinei Input.s

\*\*\*\*\* WINDLX: Citirea unui numar întreg\*\*\*\*\*

```

;-----
;Apelul subprogramului se face prin salt la simbolul "InputUnsigned"
;În momentul apelului subrutinei, în R1 trebuie sa existe adresa unui sir care
; se ;încheie cu terminatorul null
;Valoarea citita de la tastatura este returnata în R1
;Se modifica continutul registrilor R1,R13,R14
;-----

```



```

        .data
;*** Zona de date necesara pentru instructiunea trap de citire ***

ReadBuffer:
        .space 80      ; 0x1034 – adresa la care se memoreaza
                        ; valoarea citita

ReadPar:
        .word 0,ReadBuffer,80
                        ; 0x1084 – adresa unde se gasesc parametrii
                        ; apel trap 3

;*** Zona de date necesara pentru instructiunea trap de scriere ***
PrintfPar:
        .space 4       ; 0x1090

SaveR2:
        .space 4       ; 0x1094 – 4 octeti necesari pentru memorarea
                        ; lui R2

SaveR3:
        .space 4       ; 0x1098 – 4 octeti necesari pentru memorarea
                        ; lui R3

SaveR4:
        .space 4       ; 0x109C – 4 octeti necesari pentru memorarea
                        ; lui R4

SaveR5:
        .space 4       ; 0x10A0 – 4 octeti necesari pentru memorarea
                        ; lui R5

        .text
;segmentul de cod al subrutinei începe la simbolul global InputUnsigned–0x144
InputUnsigned:
        .global InputUnsigned
;*** Salvarea continutului registrelor R2, R3, R4, R5 ***

        sw SaveR2,r2 ;memorare registrul R2 la adresa 0x1094
        sw SaveR3,r3 ;memorare registrul R3 la adresa 0x1098
        sw SaveR4,r4 ;memorare registrul R4 la adresa 0x109C
        sw SaveR5,r5 ;memorare registrul R5 la adresa 0x10A0

;*** Salvare registrul R1 – contine adresa sirului “An integer value >1 :” **

        sw PrintfPar,r1      ;memorare registrul R1 la adresa 0x1090

```

```

    addi r14,r0,PrintfPar ;pregatire parametru pentru apelul sistem trap 5
    trap 5                ;apel sistem de afisare mesaj în fereastra DLX-I/O

;*** Apel sistem de citire valoare întreaga de la tastatura ***
    addi r14,r0,ReadPar  ;pregatire parametru pentru apel trap 3
    trap 3                ;apel sistem de citire valoare în fereastra DLX – I/O

;*** Determinarea valorii citite; în urma apelului sistem valoarea citita
; [codul Ascii al fiecarui octet + terminatorul null (0x0A)] este memorat la
; adresa data de ReadBuffer ***
    addi r2,r0,ReadBuffer ;R2 – initializat cu adresa simbolului
                                ; ReadBuffer
    addi r1,r0,0            ; R1 – initializat cu 0
    addi r4,r0,10          ; R4 – specifica sistemul de numeratie zecimal

;*** Citeste într-o bucla toti octetii pâna la caracterul null ***
Loop:
    lbu r3,0(r2)            ;transfer în R3 octetul curent de la ReadBuffer
    seqi r5,r3,10          ; LF -> Exit. Setare registru R5 daca octetul
                                ; curent este terminatorul null
    bnez r5,Finish         ; daca R5<>0 s-a încheiat detectarea octetilor
    subi r3,r3,48          ; refacere în R3 a valorii întregi: scadere cod
                                ; Ascii numar 0
    multu r1,r1,r4         ; shift zecimal (R1<- R1*10)
    add r1,r1,r3            ; R1<- R1+R3
    addi r2,r2,1           ; incrementare pointer – adresare urmatorul
                                ; octet de la adresa data de ReadBuffer
    j Loop                 ;reapel bucla

;*** Refacere continut registri ***
Finish:
    lw r2,SaveR2          ;refacere registru R2 cu valoarea de la adresa 0x1094
    lw r3,SaveR3          ;refacere registru R3 cu valoarea de la adresa 0x1098
    lw r4,SaveR4          ;refacere registru R4 cu valoarea de la adresa 0x109C
    lw r5,SaveR5          ;refacere registru R5 cu valoarea de la adresa 0x10A0
    jr r31                ;revenire din subrutina în programul apelant

```

---

Benchmark-ul *numarInvers.s* este un program de test simplu, care citeste un numar întreg de la tastatura si afiseaza pe inversul acestuia

(valoarea numarului este introdusa de la tastatura prin intermediul modulului *Input.s* si memorat în registrul  $R_1$  al procesorului DLX).

```

;***** WINDLX: Inversarea unui numar citit de la tastatura *****
;-----
; Programul necesita subrutina INPUT.s
; Rezultatul este afisat pe consola
;-----
        .data
Prompt:  .ascii "Dati un numar intreg: "
PrintFormat: .ascii "Valoarea Inversa=%d\n\n"
MesajNegativ: .ascii "Numarul e negativ\n"
        .align 2
PrintfPar:  .word PrintFormat    ; adresa formatului de afisare rezultat:
                                           ; mesaj + valoare
PrintfValue: .space 4            ; spatiu stocare numar în ordine
                                           ; inversa
PrintfNeg:   .word MesajNegativ  ; adresa formatului de afisare mesaj de
                                           ; eroare

        .text
        .global main
main:
        addi   r1, r0, Prompt
        jal    InputUnsigned      ; în r1 se va afla numarul citit de la
                                           ; tastatura

        addi   r2, r1, 0          ; r2 ← r1
        slt    r3, r2, 0          ; se verifica daca numarul citit este negativ
        bnez   r3, NumarNegativ   ; dacă s-a tastat un numar negativ se va
                                           ; afisa un mesaj corespunzator urmat
                                           ; de reintroducerea unei valori pozitive
                                           ; de la intrare

        add    r3, r0, r0          ; initializarea noului numar
        add    r6, r0, 10         ; în r6 ← 10, baza sistemului zecimal
        add    r4, r1, r0         ; transfera în r4 numarul citit pentru a
                                           ; opera asupra lui

loop:
        add    r7, r4, r0         ; r7 ← r4 (salvare temporara a
                                           ; numarului ramas dupa eliminarea a
                                           ; cate unei cifre)
        div    r4, r4, r6         ; r4 ← r4 / 10 (câtul împartirii)

```

multu	r5, r4, r6	; secventa necesara pentru preluarea
		; restului împartirii
sub	r5, r7, r5	; $r5 \leftarrow r4 - [(r4/10)] * 10$ (restul
		; împartirii – cifra curenta)
multu	r3, r3, r6	; cifra curenta avanseaza în cadrul
		; numarului nou format cu o pozitie:
		; din unitati devine zeci, din zeci
		; devine sute, etc.
add	r3, r3, r5	; se adauga ultima cifra a numarului
		; initial (obtinut la fiecare pas) la
		; numarul nou creat
bnez	r4, loop	; numarul initial mai contine cifre
		; $(r4 \neq 0)$ ? Daca da se reia algoritmul cu
		; urmatoarea cifra.
add	r1, r3, r0	; în r1 se afla numarul inversat tocmai
		; calculat
Afisare:		
sw	PrintfValue, r1	; salvare parametrii afisare pentru
		; <i>numarul inversat</i>
addi	r14, r0, PrintfPar	
trap	5	; afisarea valorii întregi a numarului
		; inversat
j	Finish	; salt peste mesaj de eroare
NumarNegativ:		
addi	r14, r0, PrintfNeg	; salvare parametrii afisare pentru
		; mesaj de eroare
trap	5	; afisare mesaj eroare 'Dimensiune sir
		; negativa !'
j	main	; reluare program cu reintroducerea
		; unui numar pozitiv
Finish:		
trap	0	; încheiere program

## 2.4.2. PROBLEME PROPUSE SPRE REZOLVARE

- I. Pornind de la exemplele prezentat anterior, comentati si apoi rulati programele de test **fact.s**, **prim.s** si **gcm.s**.

- II.** Scrieti un program care citește două numere de la tastatură prin intermediul modulului **Input.s** ( $n$  și  $k$ ,  $n > k$ ) și calculează  $C_n^k$  și  $A_n^k$  și le depune succesiv în memoria DLX la adresa 0x1500.
- III.** Scrieti un program care afișează primele  $n$  perechi de numere prime impare consecutive ( $n$  - număr impar citit de la tastatură). Exemplu: (3,5), (5,7), etc.
- IV.** Numim CONFIGURATIE DE BAZA a procesorului RISC DLX, următoarea configurație arhitecturală:
- 1 unitate execuție numere întregi, latență 1 ciclu;
  - 1 unitate execuție adunare flotant (FPP ADD), latență 2 cicli;
  - 1 unitate execuție înmulțire flotant (FPP MUL), latență 5 cicli;
  - 1 unitate execuție împărțire flotant (FPP DIV), latență 19 cicli;
  - Nu e implementată tehnica "forwarding".

În acest context, în continuare, se cere:

1. Rata de procesare (IR, în instrucțiuni/ciclu), măsurată pe benchmark-ul dat, pentru configurația de bază DLX.
2. Cât devine rata de procesare (comparativ cu cazul precedent), dacă se consideră 2 respectiv 8 unități de execuție FPP ADD (în rest, sunt păstrate caracteristicile configurației de bază).
3. IR, dacă se consideră 2 respectiv 8 unități de execuție FPP MUL (în rest, configurația de bază).
4. IR, dacă se consideră 2 respectiv 8 unități de execuție FPP DIV (în rest, configurația de bază).
5. IR, dacă față de configurația de bază se activează opțiunea "forwarding".
6. IR, pentru variația latenței unității de execuție FPP ADD de la 1 la 5 cicli.
7. IR, pentru latența unității FPP MUL de 1 respectiv 20 cicli.
8. IR, pentru latența unității FPP DIV de 1 respectiv 32 cicli.

**Obs:**

1. Rezultatele obținute la punctele IV.2, IV.3, IV.4 sunt identice cu cel obținut la punctul IV.1. Explicația o regăsim dacă privim în fereastra Statistica, și observăm că nu există stagnări datorate

hazardurilor structurale. Aceasta implica faptul ca, oricât de multe unitati de executie ar exista, pentru simularea optima a acestor benchmark-uri sunt necesare doar câte o unitate de executie în flotant de fiecare tip (ADD, MUL, DIV) [VinFlor99].

2. Tehnica de *forwarding* determina cresterea ratei de procesare în procente variabile (pâna la 22.97% pe **fact.s**), în functie de benchmark si functie de valorile parametrilor cu care sunt apelate programele de test. Cu cât aceste valori sunt mai mici, cu atât cresterea de performanta este mai accentuata (estimatie facuta pe acelasi benchmark - **fact.s**).

3. Este evidenta o diminuare a ratei de procesare odata cu cresterea latentei de executie a instructiunilor în virgula mobila. Varianta optima din acest punct de vedere este cea oferita de configuratia de baza.

## SCURTE CONCLUZII

Lucrarea prezinta caracteristicile importante ale procesorului DLX, urmarind înțelegerea conceptului de pipeline în general si a modului de operare a procesorului DLX în particular. Configuratia poate suferi modificari. Este interesant de observat daca introducerea unui sumator în virgula mobila este de folos, sau daca o unitate de împartire mai rapida (executia instructiunilor se face în mai putini cicli) justifica costul suplimentar. Ulterior, pot fi simulate efectele unei compilari optimizate prin rearanjarea liniilor în codul sursa, evitând astfel stagnarile datorate hazardurilor RAW.

### 3. SATSIM: SIMULAREA ARHITECTURILOR SUPERSCALARE FOLOSIND ANIMATIE INTERACTIVA

---

#### 3.1. SCOPUL LUCRARIII

Scopul lucrării este de a ajuta la înțelegerea conceptelor legate de procesarea *pipeline* a instrucțiunilor, executia *out of order*, impactul negativ asupra performanței al predicției gresite a branch-urilor și al miss-urilor în cache. Totodată se urmărește evidențierea, prin simulare la nivel de execuție, a aspectelor arhitecturale specifice procesoarelor RISC superscalare (statii de rezervare, unitati functionale de execuție, buffer de reordonare și buffer de redenumire). Lucrarea de față prezintă o descriere amplă a simulatorului SATSim, un simulator vizual și extrem de animat, scris pentru sistemul de operare Windows. Pentru a utiliza și înțelege cât mai clar modul de lucru al simulatorului sunt necesare cunostinte minime de operare în Windows dar și cunostinte bogate de microarhitectura calculatoarelor, modele de procesare existente, superscalaritate, simulare de tip *execution* respectiv *trace driven*. Înțelegerea acestor concepte fundamentale trebuie să stea la baza viitoarelor teme de cercetare gen: *trace cache*, *localitatea* și *predicția valorilor*, *reutilizarea dinamica a instrucțiunilor*, *multiprocesare* și *multithreading*.

#### 3.2. INTRODUCERE

SATSim (*‘Superscalar Architecture Trace Simulator’*) reprezintă un simulator hibrid, adică simularea se face pe trace-uri (*“trace-driven”*), dar sunt prezentate în fiecare ciclu de tact conținutul fiecărei unități de execuție,

a statiilor de rezervare, a structurii pipeline('execution driven'), fiind o unealta de animatie interactiva care exploateaza conceptele avansate ale arhitecturii superscalare: executia 'out-of-order' a instructiunilor, 'branch prediction', studiul interfetei procesor-cache [Wol00].

SATSim este puternic parametrizabil, permite utilizatorilor sa modifice interactiv parametrii configuratiilor hardware si sa vizualizeze efectele lor într-o maniera mult mai accesibila decât este posibil cu ajutorul altor simulatoare existente (spre ex. simulatoarele setului SimpleScalar, simulatorul Out of Order aferent arhitecturii HSA [Col93] etc.).

Implementarea simulatorului s-a realizat în mediul Developer Studio din Visual C++, versiunea 6.0, oferind utilizatorului o interfata prietenoasa, bazata pe meniuri, ferestre de dialog, imagini grafice edificatoare. Programul ruleaza pe platforme Windows 9x sau NT (sub sistemul de operare WinNT performanta fiind vizibil mai buna!). SATSim a fost realizat pentru uzul studentilor de catre Mark Wolf si Linda Wills si este folosit la "Georgia Institute of Technology" într-un curs de arhitectura calculatoarelor [Wol00]. Actuala versiune a simulatorului poate fi descarcata gratuit de pe Internet de la adresa: <http://www.ece.gatech.edu/research/pica/SATSim/satsim.html>.

Pe masura ce complexitatea arhitecturilor superscalare creste, înțelegerea conceptelor de 'scheduling' dinamic si de executie speculativa a instructiunilor sunt tot mai greu de explicat fara o forma de vizualizare a lor. Mijloacele statice vizuale ajutatoare, ca de exemplu diagramele (sau graficele) realizate în Word sau Excel, prezentarile din PowerPoint etc. sunt limitate în capacitatea lor de a exprima simultan relatiile structurale între componentele unei arhitecturi superscalare si relatiile temporale între instructiuni asa cum se executa ele parcurgând structura 'pipe'.

O alternativa poate fi utilizarea unui simulator pentru o arhitectura superscalara care sa înglobeze tehnici avansate de exploatare si crestere a paralelismului la nivelul instructiunilor. Oricum, o parte din simulatoarele existente [Bur97, Mou, Host] au fost proiectate în primul rând pentru cercetare, accentul punându-se pe modelarea exacta a efectelor mecanismelor arhitecturale.

Cele mai multe dintre aceste simulatoare nu încearca sa exprime în mod vizual comportarea mecanismelor arhitecturale si modul de interactiune dintre ele. Ele sunt adesea proiectate sa modeleze o arhitectura speciala, bine precizata si sunt de asemenea prea complexe pentru a fi corespunzatoare studiului studentilor care sunt începatori în studiul conceptelor arhitecturilor superscalare.

Interactivitatea simulatorului SATSim permite o cunoastere în fiecare moment a continutului resurselor (statiile de rezervare, unitati functionale,



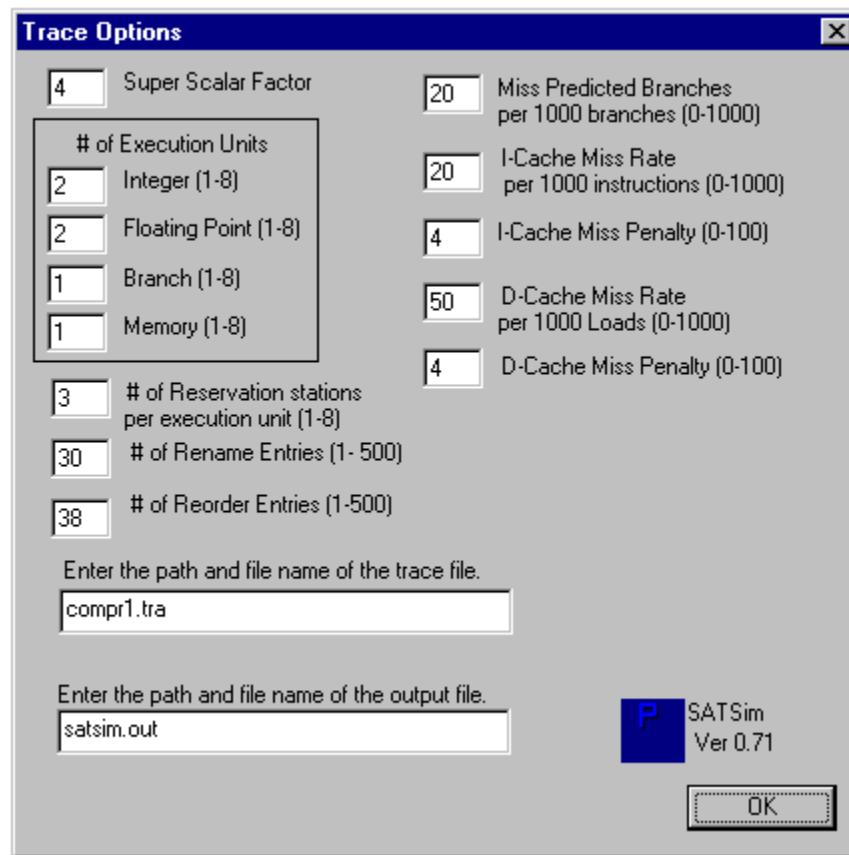
buffer de reordonare, buffer de redenumire, structura pipe, precum si experimentarea unor situatii neprevazute: fortarea unei predictii gresite a instructiunii de salt curente, fortarea unui miss în cache-ul de instructiuni/date).

### 3.3. INTERFATA CU UTILIZATORUL. DESCRIERE DETALIATA.

#### 3.3.1. PARAMETRII SIMULATORULUI

Lansarea în executie a simulatorului **SATSim** se realizeaza efectuând "dublu click" cu mouse-ul pe numele aplicatiei. Utilizatorul va introduce apoi datele initiale (parametrii arhitecturii, trace-ul de simulat, fisierul în care se va stoca configuratia arhitecturala si rezultatele) necesare la startarea programului. Odata lansata în executie simularea unei configuratii arhitecturale, simulatorul trebuie restartat pentru a putea modifica oricare din parametrii de configurare.

SATSim analizeaza secvential instructiunile din fisierul trace selectat si le afiseaza asa cum se executa ele în cadrul microarhitecturii simulate, observând în fiecare ciclu 'pipe' continutul registrelor arhitecturale. Utilizatorul poate sa configureze arhitectura selectând factorul superscalar al arhitecturii, numarul de statii de rezervare per unitate de executie, numarul intrarilor 'buffer'-elor de redenumire si reordonare, numarul de unitati de executie de un anumit tip. În plus, utilizatorul poate sa specifice (oarecum nenatural, ce-i drept...) acuratetea de predictie a salturilor, ratele de *miss* în cache, penalitatile de timp pentru *miss*-uri în cache, prin intermediul ferestrei de configurare (vezi figura 3.1).



**Figura 3.1.** Fereastra de configurare a arhitecturii superscalare simulate

În continuare, se prezintă semnificația principalilor parametri ai arhitecturii simulate:

- **Factorul superscalar al arhitecturii** - Reprezintă numărul maxim de instrucțiuni independente care pot fi extrase din cache-ul de instrucțiuni, decodificate și finalizate (scrierea rezultatelor în setul de registre generali) într-un ciclu de tact. Valori uzuale disponibile sunt în intervalul 1..16 (Astfel  $IR=FR$ , unde  $IR$  – *issue rate* și  $FR$  – *fetch rate*).
- **Numărul stațiilor de rezervare per unitate de execuție:** - Fiecare unitate de execuție are nevoie de cel puțin o stație de rezervare pentru a colecta datele de intrare în vederea execuției instrucțiunilor. Cu mai mult de o stație de rezervare, instrucțiunile pot fi lansate în execuție de îndată ce operanzii lor sursă sunt disponibili, chiar dacă o instrucțiune anterioară nu a intrat în faza de execuție (execuție "out-of-order"). Domeniul de valori disponibile utilizatorului este de la 1 la 8.

**Obs:** Valoarea stabilita (*numarul statiilor de rezervare per unitate de executie*) se aplica tuturor unitatilor de executie aferente procesorului, indiferent de tipul acestora.

În cadrul procesului general de “*executie*” a instructiunilor se disting 3 subprocese (faze) consecutive:

- ↳ **Faza de dispatch (issue – 1<sup>st</sup> stage)** – procesul de rutare a instructiunilor din bufferul de prefetch înspre statiile de rezervare multiple. Acest proces este implementat de tipul “*in order*”. Dispatch-ul unei instructiuni se face de îndata ce exista o statie de rezervare libera, aferenta tipului de instructiune.
  - ↳ **Faza de issue propriu-zis (issue – 2<sup>nd</sup> stage)** – procesul de trimitere a instructiunilor din statiile de rezervare înspre unitatile de executie (acest lucru poate fi realizat si în faza *dispatch* daca exista unitati de executie si operanzi disponibili). Lansarea în executie are loc de îndata ce operanzii sursa ai unei instructiuni devin disponibili în statia de rezervare. Acest proces se desfasoara “*out of order*”, în principal datorita penalizarilor aferente miss-urilor în cache-ul de date, latentelor unitatilor de executie etc.
  - ↳ **Faza de executie propriu-zisa** – procesarea efectiva a instructiunilor în cadrul unitatilor functionale. Ca si consecinta la faza anterioara (*issue*) si acest proces este unul de tip “*out of order*”. Aceasta executie “*out of order*” ar fi fost amplificata daca ar fi existat posibilitatea asignarii unor latente specifice diferitelor unitati de executie. Aceasta facilitate ar fi contribuit si mai mult la realismul arhitecturii simulate.
- **Numarul unitatilor de executie (pentru operatii cu numere întregi)** - reprezinta numarul total de unitati aritmetico-logice dedicate operatiilor cu numere întregi pe care procesorul le contine. Aceasta valoare reprezinta numarul de instructiuni de numere întregi pe care procesorul le poate executa într-un ciclu de tact. Domeniul de valori al parametrului este de la 1 la 8.
  - **Numarul de unitati de executie pentru instructiuni în virgula flotanta:** - Are aceeasi semnificatie ca parametrul anterior, dar cu mentiunea ca operanzii instructiunilor sunt reprezentati în virgula flotanta. Domeniul de valori este identic(1..8).
  - **Numarul de unitati de executie aferent instructiunilor de salt** - reprezinta numarul total de unitati functionale dedicate executiei instructiunilor de salt pe care procesorul le contine (numarul de

instructiuni de salt pe care procesorul le poate executa într-un ciclu de tact). Domeniul de valori îl constituie intervalul 1..8.

**Obs:** Daca predictia nu ar fi **perfecta** (exceptând cazurile de miss-predictie datorate generatorului intern) ar fi dificil de executat chiar si numai 2 instructiuni de salt simultan, executia presupunând mecanisme de aducere a mai multor instructiuni de la adrese diferite si necontigue, precum si structuri de predictie multiple. Practic, pot exista mai multe branch-uri simultan deoarece adresele target se cunosc (în fapt, acest simulator nu realizeaza practic nici o predictie, el simulând doar ciclîi necesari restaurarii starii procesorului în urma unei predictii gresite).

- **Numarul de unitati de executie dedicate instructiunilor cu referire la memorie (*load-store*)** - reprezinta numarul total de unitati dedicate executiei instructiunilor *load-store* pe care procesorul le contine (numarul maxim de instructiuni *load-store* pe care procesorul le poate executa simultan într-un ciclu de tact). Domeniul de valori al parametrului este de la 1 la 8.
- **Numarul de intrari ale *buffer*-ului de redenumire** - reprezinta numarul de intrari în *buffer*-ul de '*renaming*' aferent registrilor microprocesorului. Aceasta valoare limiteaza numarul de instructiuni, care ar putea modifica eventual un registru al arhitecturii, si care se afla în curs de executie ('*in-flight*'). Orice instructiune care a fost expediata spre executie (a parasit nivelul '*issue*') si nu a fost finalizata (nu a fost scris înca rezultatul în setul de registri generali, adica nu s-a executat faza '*commit*') se considera a fi în stadiul '*in-flight*'. Domeniul de valori al parametrului îl constituie intervalul 0...500.
- **Numarul de intrari în *buffer*-ul de reordonare** - reprezinta numarul de instructiuni care pot fi simultan '*in-flight*'. Instructiunile care nu modifica un registru al arhitecturii (mai precis instructiunile de salt si cele de tip *store*) nu au nevoie de o intrare în *buffer*-ul de renaming, dar au nevoie de o intrare în cel de reordonare, deoarece trebuie retinuta ordinea exacta a tuturor instructiunilor, dar si starea acestora (daca instructiunea este executata speculativ, daca branch-ul s-a rezolvat instructiunea se afla pe calea corecta sau trebuie evacuata structura pipeline si reluata procesarea), pentru mentinerea coerenta a contextului procesorului (si a memoriei). De exemplu, daca o instructiune de tip *store*, care nu s-ar afla în *ReorderBuffer* se executa în alta ordine decât cea initiala si apare o întrerupere, atunci nu se va sti exact daca trebuie reluata executia ei. La fel si în cazul unei instructiuni de salt: daca nu s-ar

înscrie în *ReorderBuffer (RB)* si la finele executiei ei se dovedeste predictia a fi gresita sau apare o exceptie, nu se va sti exact cu care instructiune sa se reia procesarea. Domeniul de valori uzuale este de la 0 la 500. Atât buffer-ul de renaming cât si cel de reordonare reprezinta resurse hardware necesare mecanismului de executie 'out-of-order' a instructiunilor [Hen96, Vin00].

**Obs:** Teoretic, o alternativa la utilizarea unui *ReorderBuffer (RB)* consta în utilizarea tehnicii de **redenumire dinamica a registrilor**. Aceasta se bazeaza pe exploatarea unui set extins de registri fizici care înlocuiesc din punct de vedere functional atât RB-ul cât si statiile de rezervare. Pe timpul fazei de lansare în executie a instructiunilor, registrul destinatie este redenumit cu un registru liber (nealocat) din acest set extins. Astfel, dependentele de tip WAR si WAW sunt eliminate. Acest registru fizic alocat destinatiei instructiunii, devine "*registru logic*" abia în momentul în care instructiunea se încheie din punct de vedere al procesarii *in-order* (asadar faza "*commit*", ulterioara fazei "*write-back*"). Conversia logic-fizic se face printr-o simpla tabela de mapare.

Un avantaj al redenumirii dinamice fata de tehnica RB consta în faptul ca procesele aferente terminarii *in-order* a instructiunilor sunt simplificate. Aceasta terminare implica faptul ca maparea respectivului registru logic în registrul fizic corespondent nu mai este activa, si, desigur, registrul fizic redevine disponibil. În cazul RB, statia de rezervare se elibera în momentul încheierii executiei instructiunii în unitatea de executie iar locatia din RB se elibera la finele executiei *in-order* a instructiunii (faza "*commit*"). Totusi, în cazul dealocarii unui registru fizic trebuie verificat sa nu mai existe instructiuni în curs care sa-l utilizeze pe post de registru-sursa. Ca alternativa pentru dealocarea unui registru fizic, procesorul poate astepta ca o alta instructiune sa modifice respectivul registru. În acest caz, un registru poate fi tinut alocat mai mult decât ar fi stricat necesar (cazul MIPS R10000 [Yeag96]). O alta simplificare a tehnicii de redenumire fata de cea a utilizarii unui RB consta în faptul ca operandii sursa ai instructiunii se afla în setul extins de registrii (în cazul filozofiei RB, sursele puteau fi gasite în RB sau în registrii logici). Redenumirea dinamica a registrilor este implementata la microprocesoarele MIPS R10000/120000, Alpha 21264, Pentium III, IV) [Yeag96, Ung99].

În cadrul acestui simulator structura de redenumire dinamica pur si simplu conlucreaza cu un RB simplificat. În esenta, prima e raspunzatoare

de eliminarea dependentelor WAR si WAW în timp ce RB-ul se ocupa numai de gestiunea încheierii *in order* a instructiunilor (faza *Commit*), în vederea implementarii unui mecanism precis de exceptii.

- **Numarul de predictii gresite la 1000 de instructiuni de salt**

**rata de miss = 1-rata de hit**

Rata de miss aferenta predictiei salturilor reprezinta numarul de predictii gresite la 1000 de instructiuni de salt executate ( $1 \Leftrightarrow 0.1\%$ ). De fiecare data când simulatorul întâlnește o instructiune de salt, un numar aleator de la 0 la 999 este generat. Daca numarul aleator e mai mic decât numarul introdus aici, instructiunea e marcata ca si gresit predictionata. Generatorul de numere aleatoare substituie scheme de predictie clasice (BTB, GAg, PAg) implementate în procesoarele reale (comerciale). Aria permisa a valorilor acestui parametru al arhitecturii superscalare e de la 0 la 1000.

- **Numarul de miss-uri în cache-ul de instructiuni la 1000 de instructiuni citite** - reprezinta numarul de miss-uri în cache la 1000 de instructiuni ( $1 \Leftrightarrow 0.1\%$ ). De fiecare data când simulatorul extrage o instructiune din cache, un numar aleator de la 0 la 999 este generat. Daca numarul aleator e mai mic decât numarul prestabilit, instructiunea e marcata ca si miss în cache. Valorile permise pentru acest parametru sunt de la 0 la 1000.
- **Penalitatea introdusa în cazul unui miss în cache-ul de instructiuni** - reprezinta numarul de cicli de tact în care unitatea de *fetch* stagneaza, adica întârzie procesarea (*stalls*) în caz de miss în cache. Poate sa ia valori de la 1 la 100 (valori realiste în functie de caracteristicile sistemului ierarhizat de memorie respectiv procesorului).
- **Rata de miss în cache-ul de date la 1000 de instructiuni load** - semnificatie identica ca cea de la rata de miss în cache ul de instructiuni. Trebuie stiut ca în caz de miss în cache-ul de date cauzat de o instructiune de tip *load*, orice acces în acelasi bloc din cache, va fi întârziat pâna la rezolvarea miss-ului. Daca accesul se va face în alt bloc din cache, instructiunea se executa normal daca exista unitati functionale, dedicate instructiunilor cu referire la memorie, disponibile.
- **Penalitatea introdusa în caz de miss în cache-ul de date** – analog cu penalitatea de timp în caz de miss în cache-ul de instructiuni din punct de

vedere al semanticii, doar ca de data aceasta nu unitatea de fetch stagneaza, ci instructiunile cu referire la memorie aflate în statiile de rezervare.

- **Fisierul de intrare (trace file)** - reprezinta calea de directoare si numele fisierului "trace" utilizat ca parametru de intrare al simulatorului. Calea de directoare e relativa la locatia executabilului simulatorului. Un exemplu de fisier trace este prezentat în figura 3.2.
- **Fisierul de iesire (output file)** - calea de directoare si numele fisierului în care sunt pastrate rezultatele în vederea interpretarii calitative si cantitative. Programul va scrie rezultatele în acest fisier când se termina parcurgerea fisierului de intrare sau la terminarea programului (din diverse cauze, dorite sau nedorite). Fisierul este de tip text, valorile fiind delimitate prin <Tab> si este deschis în mod "append". Va contine informatiile existente în Tabelul 3.1.

PARAMETRU	SEMNIFICATIE
<b>Date and Time</b>	Data la care a fost efectuata simularea
<b>Trace File Name</b>	Numele fisierului de intrare în format trace (contine valorile PC, IHI, ILO, mnemonica ASM)
<b>SATSim.out.</b>	Constituie fisierul ASCII care retine rezultatele simularii.
<b>Pilog.out</b>	În caz de eroare va contine cauza încheierii eronate a simularii si efectul rezultat.
<b>Total Cycles</b>	Numarul total de cicli executati
<b>Number Instructions Committed</b>	Numarul de instructiuni care au fost complet executate (au scris rezultatele în setul de registri generali)
<b>Number of Integer Instruction Fetched</b>	Numarul instructiunilor cu operanzi întregi extrase din cache.
<b>Number of Store Instruction Fetched</b>	Numarul instructiunilor de tip store citite din cache.
<b>Number of Load Instruction Fetched</b>	Numarul instructiunilor de tip load citite din cache.
<b>Number of Branch Instruction Fetched</b>	Numarul instructiunilor de salt extrase din cache.
<b>Number of Float Instruction Fetched</b>	Numarul instructiunilor cu operanzi flotanti extrase din cache.
<b>ICache Misses</b>	De câte ori faza de fetch instructiune s-a încheiat cu miss în cache

<b>Pipe Stall Cycles</b>	Numarul total de cicli în care procesarea a stagnat (întârzierea unitatii de fetch datorita faptului ca unitatea de decodificare nu a fost libera)
<b>DCache Misses</b>	Numarul de accese cu miss în cache-ul de date
<b>Mis-Predicted branches</b>	Numarul de branch-uri gresit predictionate.
<b>Mis-Predicted branch cycles</b>	Numarul total de cicli de tact datorat branch-urilor gresit predictionate.
<b>Reorder Utilization</b>	<b>Reorder use/#Cycles/#Reorder</b> (Numarul total de cicli în care fiecare intrare în buffer-ul de reordonare a fost activa).
<b>Rename Utilization</b>	Numarul total de cicli în care fiecare intrare în buffer-ul de <i>renaming</i> a fost activa
<b>Integer Execution Utilization</b>	Numarul total de cicli în care a fost activa fiecare din unitatile de executie pentru întregi ( <b>Res int /#Cycles/#IntUnits</b> )
<b>Floating Point Execution Utilization</b>	Numarul total de cicli în care a fost activa unitatea de executie pentru instructiuni în virgula mobila
<b>Branch Execution Utilization</b>	Numarul total de cicli în care a fost activa unitatea de executie pentru instructiuni de salt
<b>Memory Execution Utilization</b>	Numarul total de cicli în care a fost activa unitatea de executie pentru instructiuni cu referinta la memorie
<b>Integer Reservation Utilization</b>	Numarul total de cicli în care fiecare statie de rezervare pentru instructiuni de întregi a fost activa ( <b>Res int/#Cycles/#Res per Exe</b> )
<b>Floating Point Reservation Utilization</b>	Numarul total de cicli în care fiecare statie de rezervare pentru instructiuni cu operanzi în virgula mobila a fost activa
<b>Branch Reservation Utilization</b>	Numarul total de cicli în care fiecare statie de rezervare pentru instructiuni de salt a fost activa
<b>Memory ReservationUtilization</b>	Numarul total de cicli în care fiecare statie de rezervare pentru instructiuni cu referire la memorie a fost activa

Tabelul 3.1

### Parametrii si rezultatele simularii

Predictia salturilor si functionarea *cache*-ului pot fi ilustrate statistic sau bazat pe actiunea interactiva a utilizatorului cu ajutorul “*butoanelor*” din bara cu instrumente a simulatorului (pentru fortarea unei predictii gresite a unei instructiuni de salt sau pentru fortarea unui miss în *cache*-ul de date).



```

PC:401eb8 HI:4d LO: 40418 ;sra r4,r4,24
PC:401ec0 HI: 3 LO: 1007f6 ;jal 0x401fd8
PC:401fd8 HI:49 LO: 40418 ;sll r4,r4,24
PC:401fe0 HI:4d LO: 4040e ;sra r4,r4,14
PC:401fe8 HI:71 LO: 110e5 ;lui r1,0x10e5
PC:401ff0 HI:36 LO: 4010100 ;addu r1,r4,r1
PC:401ff8 HI:15 LO: 100c1e8 ;l.d f0,-15896(r1)
PC:402000 HI:76 LO: 2060000 ;dmtc1 r6,f2
PC:402008 HI:36 LO: 600 ;addu r6,r0,r0
PC:402010 HI:6a LO: 20000 ;c.lt.d f0,f2
PC:402018 HI:37 LO: 7007f ;addiu r7,r0,127
PC:402020 HI: c LO: 8 ;bc1f 0x402048
PC:402048 HI:36 LO: 500 ;addu r5,r0,r0
PC:402050 HI:71 LO: 210e5 ;lui r2,0x10e5
PC:402058 HI:37 LO: 202bdf0 ;addiu r2,r2,-16912
PC:402060 HI:36 LO: 4020400 ;addu r4,r4,r2
PC:402068 HI:36 LO: 6070200 ;addu r2,r6,r7
PC:402070 HI:4d LO: 20301 ;sra r3,r2,1
PC:402078 HI:49 LO: 30203 ;sll r2,r3,3
PC:402080 HI:36 LO: 2040200 ;addu r2,r2,r4
PC:402088 HI:15 LO: 2000000 ;l.d f0,0(r2)
PC:402090 HI:6a LO: 2000000 ;c.lt.d f2,f0
PC:402098 HI: c LO: 4 ;bc1f 0x4020b0
PC:4020b0 HI:6a LO: 20000 ;c.lt.d f0,f2
PC:4020b8 HI: c LO: ffde ;bc1f 0x402038
PC:4020c0 HI:37 LO: 3060001 ;addiu r6,r3,1
PC:4020c8 HI:37 LO: 5050001 ;addiu r5,r5,1
PC:4020d0 HI:50 LO: 5020007 ;sllti r2,r5,7
PC:4020d8 HI: 7 LO: 200ffe2 ;bne r2,r0,0x402068

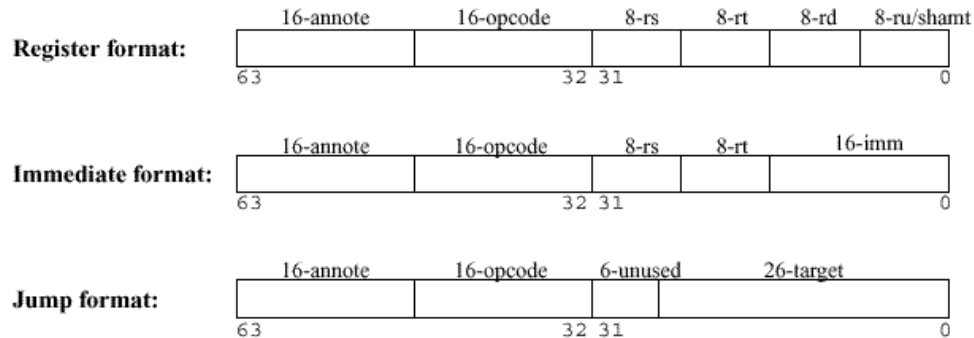
```

Figura 3.2. Continutul fisierului trace (*compr1.tra*)

Fisierul de intrare este în formatul MIPS. Se disting 5 tipuri de instrucțiuni (și implicit unități funcționale): întregi, virgula flotantă, de ramificație, *load* și *store*. Registrul sursă și destinație sunt decodificați, astfel încât dependențele de date (hazardurile WAR, WAW) să fie rezolvate cu acuratețe prin ‘*renaming*’. Instrucțiunile sunt codificate pe 8 octeți. Prima coloană (figura 3.2) reprezintă valoarea registrului PC (adresa instrucțiunii curente). A doua coloană cuprinde valoarea registrului instrucțiunii (partea HI, adică primii 4 octeți), practic “*opcode*”-ul instrucțiunii. Următoarea coloană conține valoarea registrului instrucțiunii (partea LO), practic codificarea operanzilor instrucțiunii, moduri de adresare etc. Toate valorile numerice care apar în cod sunt în format hexazecimal. Ultima coloană din fisierul trace (succesoarea simbolului “;”) reprezintă instrucțiunea curentă în mnemonica de asamblare.

Există trei formate de instrucțiuni: *registru*(R-tip), *imediat*(I-tip) și de *salt*(J-tip). Formatul registru este specific instrucțiunilor de calcul (aritmetic – logice). Formatul imediat este folosit de către instrucțiunile de transfer și de instrucțiunile de salt condiționat. Adresa de salt reprezintă un offset pe 16 biți față de PC-ul curent. Cuprinde un câmp constantă imediată

pe 16 biti. Al treilea format se numeste J-tip, si cuprinde printre altele si un câmp de adresa de 24 de biti. Câmpurile alocate fiecarui registru sunt pe 8 biti în scopul suportarii unei extensii ulterioare a arhitecturii de registre de 256 registri întregi si 256 flotanti. Câmpul opcode este pe 16 biti facilitând o decodificare rapida a instructiunii.



**Figura 3.3.** Formatul instructiunii pentru arhitectura simulata

Pentru exemplificare consideram instructiunile:

**a) addu r2, r6, r7**

Formatul instructiunii de adunare este R-tip rezultând ca în registrul ILO va trebui sa existe codificat în ordine: primul operand sursa, al doilea operand sursa, registrul destinatie si valoarea 0 (deoarece nu exista deplasare (shiftare) si nici instructiunea nu are 3 operanzi sursa). În IHI se va gasi codul operatiei de adunare.

Daca verificam în fisierul compr1.tra la adresa PC: 402068h se observa ca:

ILO = 06070200h;

IHI = 36h;

**b) sw r3, -31600(r28);**

Formatul instructiunii cu referire la memorie este I-tip, rezultând ca în registrul IHI se afla opcode-ul instructiunii , iar în ILO se vor afla în ordine: primul operand sursa (adresa de baza), operandul sursa ce va fi înscris în memorie si respectiv valoarea imediata.

În ce priveste valoarea imediata, aceasta fiind negativa, va fi reprezentata în complement fata de 2 (reprezentare hexagesimala).

$(-31600 = 65535 + 1 - 31600 = 33936 = 0x8490)$ .

Într-adevar, daca urmarim în fisierul trace la adresa PC: 402178h se observa ca IHI=1Ah si ILO=1C038490h.

- c) **jal 0x401FD8** (adresa de salt e cea absoluta, adica valoarea PC-ului instructiunii destinatie)  
 ILO= 0x1007F6 (0x1007F6 \* 4=0x401FD8)

### 3.3.2.ALTE RESURSE SOFTWARE UTILIZATE. MENIUL SIMULATORULUI SI BARA DE INSTRUMENTE

#### 3.3.2.1 MENIUL SIMULATORULUI

La lansarea în executie a simulatorului si dupa închiderea ferestrei de configurare a arhitecturii, pe ecranul calculatorului va apare meniul principal, cuprinzând submeniurile: **File**, **View**, **Window**, **Cycle**, **Timer**, **Force** si **Help**.

Acestea pot fi activate si apăsând combinatia de taste formata din tasta <Alt> si de literele subliniate din submeniu.

NUME SUBMENIU	OPTIUNI SUBMENIU	SEMNIFICATIA
FILE	New Trace	Starteaza o noua simulare. Pe ecran va apare o noua fereastră de configurare .Pot fi realizate simultan mai multe simulari.
	Open Batch	Nu este încă implementata; are acelasi efect ca si New Trace.
	Close	Va avea ca efect oprirea simulării în curs.
	Print Print Preview Print Setup	Reprezinta functii <i>Windows</i> standard. Permite setarea imprimantei, previzualizarea si printarea ferestrei de simulare active(starea curenta a simulatorului).
	Exit	Determina terminarea programului (închiderea aplicatiei SATSim.exe).
VIEW MENU		Permite utilizatorului sa controleze daca se afiseaza sau nu bara de instrumente sau bara de stare.
WINDOW MENU		Permite utilizatorului sa deschida si sa controleze ferestre multiple pentru afisarea stadiului simulării aflate în curs de desfasurare.

<b>CYCLE MENU</b>	<b>Single step</b>	Va opri timer-ul daca acesta este activ, determinând ca procesorul sa execute un singur ciclu de tact.
	<b>Cycle timer</b>	Va activa timer-ul, fortând procesorul sa treaca automat de la un ciclu de procesare la altul, actualizând totodata si ecranul.
	<b>10 Cycle Timer</b>	Va activa timer-ul, fortând procesorul sa execute continuu, ecranul fiind actualizat la fiecare 10 cicli.
	<b>100 Cycle Timer</b>	Identice ca mai sus, cu precizarea ca actualizarea ecranului se va face la fiecare 100 de cicli parcursi.
	<b>1000 Cycle Timer</b>	Identice ca mai sus, cu precizarea ca actualizarea ecranului se va face la fiecare 1000 de cicli parcursi.
	<b>Go To End</b>	Are ca si efect simularea neîntrerupta pâna la finalul trace-ului, dar fara actualizarea ecranului. Reprezinta cel mai rapid mod de simulare.
<b>TIMER MENU</b>	<b>10 ms</b>	Cel mai rapid mod de simulare, viteza variind în functie de sistemul de operare si de calculatorul folosit, între 15 si 100 de cicli pe secunda.
	<b>100 ms</b>	Simularea va rula la o viteza de 10 cicli pe secunda.
	<b>500 ms</b>	Simularea va rula la o viteza de 2 cicli pe secunda.
	<b>1 s</b>	Simularea va rula la o viteza de 1 ciclu pe secunda. Reprezinta cel mai lent, dar si cel mai didactic mod de simulare.
<b>FORCE M4ENU</b>	<b>Branch MissPredict</b>	Urmatoarea instructiune de salt extrasa din cache va fi etichetata ca si gresit predictionata, incrementându-se contorul de predictii eronate.

	<b>ICache miss</b>	Urmatoarea linie de instructiuni extrasa va cauza un miss în cache-ul de instructiuni. Instructiunile vor fi aduse din memoria principala. Daca nu exista instructiuni în statiile de rezervare sau unitatile functionale, procesarea va stagna cu un numar de cicli de penalizare prestabilit.
	<b>DCache miss</b>	Urmatoarea instructiune de tip <i>load</i> , ajunsa în ultimul ciclu al fazei de executie(acces la memorie) va determina un miss în cache-ul de date. Procesarea instructiunilor cu referire la memorie si a celor dependente de rezultatul acesteia va întârzia cu un numar de cicli de tact prestabilit în fereastra de configurare(Figura 3.1).
<b>HELP MENU</b>		Furnizeaza informatii despre simulator si folosirea lui.

Tabelul 3.2.

### Meniul simulatorului SATSim

#### 3.3.2.2 BARA DE INSTRUMENTE A SIMULATORULUI

Bara de instrumente poate fi ascunsă sau mutată în locația dorită prin mecanismul "*Drag and drop*"(specific sistemului de operare Windows). Ea este compusă din următoarele butoane:



**Single-Step** ⇨ Oprește timer-ul și impune procesorului modul de lucru pas cu pas cu cicluri de 1 tact. Tasta funcțională corespunzătoare este F3.



**Timer** ⇒ Activare timer. Simulatorul va rula automat. Tasta funcțională corespunzătoare este F4.



**Go To End of Trace** ⇒ Efectul e similar cu optiunea de meniu corespunzatoare din submeniul CYCLE MENU.



**Miss Predict Branch** ⇒ Efectul e similar cu optiunea de meniu corespunzatoare din submeniul FORCE.



**ICache Miss** ⇒ Efectul e similar cu optiunea de meniu corespunzatoare din submeniul FORCE.



**DCache Miss** ⇒ Efectul e similar cu optiunea de meniu cu același nume din submeniul FORCE.



**Printare** ⇒ Listează ecranul curent la imprimantă.



**Ajutor sensibil la context** ⇒ Nu este activ în versiunea de simulator curentă.

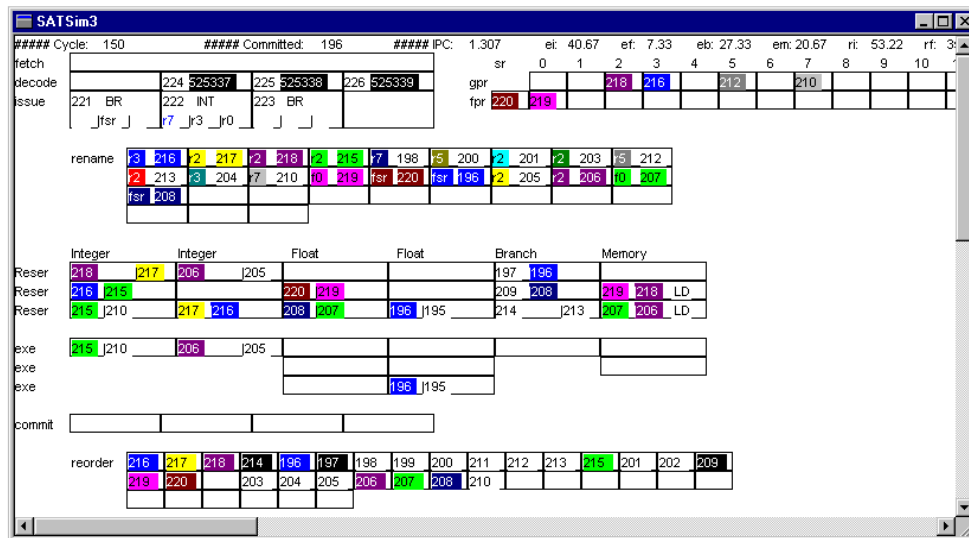
### 3.4. DESCRIEREA FUNCTIONALITATII PROCESORULUI

Interactiv poate fi vizualizat conținutul fiecărei resurse arhitecturale, stările tuturor instrucțiunilor aflate în curs de procesare pe măsura ce acestea ocupă locațiile din *pipe*, în *buffer*-ul de redenumire, în *buffer*-ul de reordonare, în stațiile de rezervare, și în unitățile de execuție. Instrucțiunilor li se aplică culori când registrul lor destinație este redenumit, și pierd culoarea odată ce rezultatul lor a fost emis (preluat de instrucțiunile dependente).

Culoarea, sau lipsa de culoare, pentru instrucțiunile sursă (furnizoare de operanți), este afișată pentru instrucțiunile care ocupă o stație de

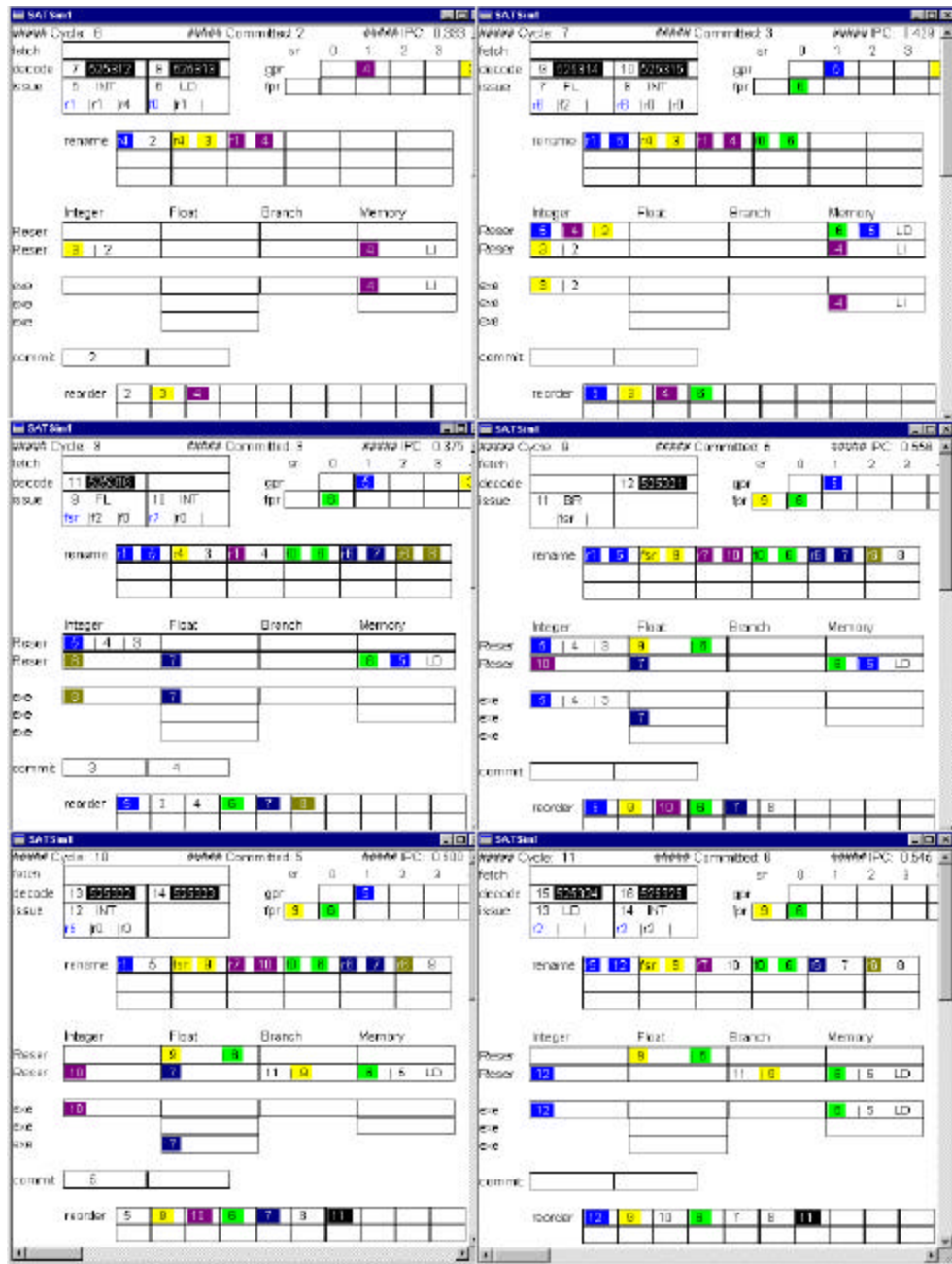
rezervare. Daca exista, culoarea curenta asignata pentru fiecare registru este de asemenea ilustrata. Starea fiecarei instructiuni este actualizata cu fiecare ciclu de ceas simulat. Prin executia trace-ului metricile de performanta aferente (rata de procesare, ratele de hit etc.) sunt actualizate si afisate în fiecare ciclu de tact.

Animatia are trei caracteristici interactive care sunt folositoare pentru explicarea si înțelegerea conceptelor avansate de arhitectura a calculatoarelor (fortarea unei predictii gresite, unui miss în Icache, sau executia unei instructiuni de load cu miss în Dcache). În timpul simulării, utilizatorul detine controlul vitezei de animatie, aceasta putând fi setata sa actualizeze ecranul la fiecare 1, 10, sau 1000 cicluri de ceas. Simularea poate fi facuta pas cu pas (interactiv), sau rulând un numar multiplu de pasi (1, 2, ..., 10) prestabilit de utilizator. Exista posibilitatea de executie în mod continuu pâna la sfârșitul trace-ului fara actualizarea ecranului.



**Figura 3.4.** Instantaneu cu resursele hardware ale arhitecturii

Figura 3.4 prezinta un instantaneu cu microarhitectura într-un anumit ciclu de tact (o fereastra de animatie). Figura 3.5 indica stadiile de procesare succesive (sase cicluri de ceas) aferente grupului de instructiuni care cuprind instructiunea 5 - de la extragerea din cache pâna la scrierea rezultatului în setul de registri generali. Fiecarei instructiuni îi este asignat un nume pe trei digiti, astfel încât aceasta sa poata fi afisata si urmarita de catre utilizator pe masura ce ea progreseaza prin microarhitectura simulata.



**Figura 3.5.** Stadii succesive de procesare (sase cicluri de ceas) aferente unui grup de instructiuni

Primele 3 valori afisate în fereastra de dialog (figura 3.4) reprezinta numarul de cicluri (**Cycle**), numarul de instructiuni executate (**Committed**) si rata de procesare curenta (**IPC**). Valorile ramase reprezinta ratele de



utilizare aferente unitatilor de executie întregi (**ei**), unitatilor de executie în virgula flotanta (**ef**), unitatilor de executie pentru instructiuni de salt (**eb**), unitatilor de executie pentru instructiuni cu acces la memorie (**em**), pentru statiile de rezervare pentru valori întregi (**ri**), pentru statiile de rezervare pentru valori în virgula mobila (**rf**), pentru statiile de rezervare dedicate salturilor (**rb**), pentru statiile de rezervare aferente instructiunilor cu referire la memorie (**rm**), rata de utilizare corespunzatoare locatiilor din "buffer-ul de renaming"(**rn**), si rata de utilizare corespunzatoare locatiilor din "buffer-ul de reordonare"(**ro**).

### 3.4.1 FAZA FETCH-INSTRUCTIUNE

Realizeaza extragerea instructiunilor din cache sau din memoria principala, în caz de miss în cache. Daca procesul de fetch este cu miss, atunci acest lucru este semnalizat printr-un mesaj de culoare rosie edificator, "IcacheMiss". Stagnarea (latenta procesului de fetch în acest caz) dureaza pâna la aducerea instructiunilor din memoria principala (un numar de cicli precizat prin parametul IcacheMiss Penalty). Daca nu se semnaleaza un miss în cache-ul de instructiuni si modulul de decodificare este gol, atunci o noua linie de instructiuni este extrasa (din fisierul trace în format ASCII). Daca modulul de decodificare nu este gol (adica cel putin o instructiune este în faza de decodificare), atunci faza fetch-instructiune stagneaza (stalls).

Adresele instructiunilor sunt aliniate în unitatea *pipe* de fetch la valori multiplu de factorul superscalar al arhitecturii. Adresa primei instructiuni determina locul unde va fi plasata. Instructiunile sunt extrase pâna când se umple si ultima locatie a unitatii de fetch (adresa ultimei instructiuni din grup este divizibila cu factorul superscalar), sau pâna se întâlnește un salt care se predictioneaza ca se face (taken), sau pâna se întâlnește o instructiune de salt gresit predictionata. O instructiune de salt care se face (taken) este întâlnita când adresa instructiunii urmatoare difera de adresa instructiunii precedente cu o valoare diferita de 8 octeti. Daca predictia este taken procesarea continua cu instructiunea destinatie a saltului. Un salt gresit predictionat este determinat de un generator de numere aleatoare sau poate fi fortat de catre utilizator.

### 3.4.2 FAZA DE DECODIFICARE A INSTRUCȚIUNII

Reprezintă nivelul pipeline în care instrucțiunile inutile (adrese *tinta* - destinația instrucțiunilor de salt - nealiniate, instrucțiunile următoare celei de salt taken în grupul de instrucțiuni citite) aduse în faza de fetch-instrucțiune sunt îndepărtate. De asemenea, instrucțiunile sunt decodificate pentru a se determina operanzii sursă și destinație, tipul operației ce va fi executată, informații necesare fazelor următoare de procesare (*issue*, execuție).

Se disting două valori numerice afișate în cadrul fiecărei unități de decodificare (*Ex*: 166 525325). Numărul instrucțiunii scris cu culoarea neagră pe fond alb reprezintă doar un identificator pe care simulatorul îl asignează instrucțiunii pentru ca aceasta să poată fi "*urmarita*" prin fazele de procesare (valori uzuale de la 0 la 499), pentru a cunoaște eventualele dependente de date etc. Adresa instrucțiunii curente reprezintă numărul afișat cu culoarea albă pe fond negru și se obține împărțind PC-ul la 4. Adresa este afișată pentru a indica modul de asignare a instrucțiunilor extrase din cache unităților de decodificare respective în această fază de procesare.

Instrucțiunile aflate pe nivelul de decodificare pot fi transmise nivelului "*issue*" în funcție de spațiul liber existent pe acest nivel. Ideal ar fi ca numărul de instrucțiuni care tranzitează de pe un nivel pe altul să fie egal cu factorul superscalar al arhitecturii.

### 3.4.3 NIVELUL "*DISPATCH-ISSUE*" DE PROCESARE. RUTAREA INSTRUCȚIUNILOR SPRE STATIILE DE REZERVARE ȘI UNITĂȚILE DE EXECUȚIE CORESPUNZATOARE

În funcție de numărul stațiilor de rezervare libere (disponibile), instrucțiunile aflate pe nivelul *issue* sunt direcționate spre stațiile corespunzătoare tipului de instrucțiuni. Întrucât modelul de procesare este "*in-order*", în momentul când o instrucțiune nu poate fi lansată în execuție, atunci și instrucțiunile succesoare sunt stagnate în faza "*issue*". Instrucțiunile care nu pot fi asignate unei SR (stații de rezervare) vor fi deplasate la stânga pentru a se face loc pentru mai multe instrucțiuni venind din faza de decodificare în următorul ciclu de tact.

Există 2 linii de informație afișate pentru fiecare instrucțiune aflată în faza de "*issue*". Prima linie arată numărul instrucțiunii și tipul ei. A doua

linie afiseaza destinatia arhitecturala si registri sursa daca acestia exista. Registrul destinatie este afisat în stânga si este scris cu litere albastre. Registrul sursa sunt afisati cu litere negre.

O instructiune va fi expediata unei statii de rezervare, corespunzatoare unei unitati de executie de tip similar ei, exceptând cazul când în care una din propozitiile urmatoare este adevarata:

1. O instructiune de acelasi tip a fost deja expediata în vederea executiei în ciclul de tact urmator;
2. Toate statiile de rezervare ale unitatilor de executie corespunzatoare sunt pline;
3. Instructiunea modifica un registru arhitectural si nu mai exista intrari în buffer-ul de redenumire disponibile;
4. Nu mai exista intrari disponibile în buffer-ul de reordonare.

Când o instructiune este trimisa spre executie, are loc urmatoarea secventa de evenimente:

1. Daca instructiunea are un registru destinatie, i se acorda o culoare din partea buffer-ului de "*renaming*". Aceasta e indicata în buffer-ul de redenumire de culoarea numarului instructiunii si a registrului arhitectural corespunzator.
2. De asemenea, daca instructiunea are un registru destinatie, numarul colorat al instructiunii este afisat în locatia registrului din setul de registri generali *gpr* sau *fpr* (având rol de **tag** – informatie de recunoastere, în procesul de redenumire a registrilor, putând fi folosit de statiile de rezervare sau unitatile functionale de executie care au nevoie de ultima valoare a respectivului registru). Aceasta pentru ca orice instructiune viitoare care depinde de rezultatul instructiunii sa "stie" ce intrare de "*renaming*" sa îi fie acordata.
3. Toate instructiunile au asignata o intrare în buffer-ul de reordonare indicata prin colorarea numarului instructiunii în prima intrare libera a acestuia.
4. Instructiunii îi este asignata o statie de rezervare, fapt remarcat prin numarul instructiunii afisat în partea stânga a statiei de rezervare.
5. Daca o instructiune este extrasa în vederea executiei si toti operanzii sursa sunt disponibili imediat si daca unitatea de executie nu primeste alta instructiune din statiile ei de rezervare, instructiunea va fi executata imediat de catre unitatea de executie (lucru remarcat prin aparitia numarului instructiunii afisat în partea stânga a primei faze pipeline din unitatea de executie corespunzatoare).

### 3.4.4 SETUL DE REGISTRI GENERALI

Describe starea curenta a fiecarui registru arhitectural (redenumit sau disponibil). Exista 32 de registri (pentru numere întregi  $<0..31>$ ), 32 de registri pentru numere în virgula mobila  $<0..31>$  si un registru de stare pentru coprocesorul în virgula flotanta (fsr). La fel ca la toate microprocesoarele RISC registrul R0 este cablat la masa ( $R0=0$ ), neexistând nici un motiv sa redenumim instructiuni care scriu în acest registru.

O locatie libera în setul de registri generali indica faptul ca registrul pentru moment nu este redenumit, registrul reținând valoarea corecta. Daca registrul a fost redenumit, casuta va arata culoarea numarului instructiunii care va produce cea mai recenta valoare pentru acest registru (tag-ul, care va fi folosit pentru eliminarea posibilelor hazarduri WAR si WAW din secventele succesoare de instructiuni). Culoarea corespunde intrarii în buffer-ul de "renaming".

### 3.4.5 BUFFER-UL DE REORDONARE (RB)

Bufferul de reordonare simplificat împreuna cu mecanismul de redenumire dinamica a registrilor concura la executia 'out-of-order' a instructiunilor, precum si asigura implementarea unui mecanism precis de tratare a evenimentelor de exceptie (derute, devieri, întreruperi hard-soft etc.). Acesta contine un numar de locatii care sunt alocate în mod dinamic instructiunilor. Simulatorul modelând în linii mari arhitectura procesorului Power PC, **bufferul de reordonare (RB)** retine doar indexul si starea instructiunilor, rezultatul acestora fiind pastrat în **bufferul de redenumire (RenB)** [Ung99, Sim97, Bhan96].

Bufferul RB poate fi gestionat ca o memorie FIFO (First In First Out). În momentul decodificarii unei instructiuni, indexul (a câta instructiune din trace este instructiunea curenta) acesteia este alocat în coada RB. Când aceasta instructiune ajunge în prima pozitie a RB, daca între timp nu au aparut exceptii, are loc o cautare asociativa dupa indexul instructiunii în *fpr* sau *gpr*, dar si în statiile de rezervare. În caz de hit rezultatul instructiunii (aflat în RenB) este înscris în setul de registri generali. De asemenea, instructiunile aflate în asteptare dupa acest rezultat în statiile de rezervare vor fi deblocate si pot trece în faza de executie efectiva. În cazul în care, cautarea se încheie cu miss, nu se opereaza nimic în *fpr* sau *gpr*, ci doar se evacueaza intrarile din buffer-ul de reordonare si din cel de redenumire.

Daca instructiunea nu s-a încheiat atunci când locatia alocata în RB a ajuns prima, buffer-ul RB nu va mai avansa pâna când executia acestei instructiuni nu se va încheia. Instructiunile sunt retrase (*committed*) – rezultatele sunt scrise în setul de registri generali *in order* (toate instructiunile anterioare, din codul sursa, celei aflate pe prima pozitie a buffer-ului de reordonare s-au încheiat cu succes, scrierea rezultatelor făcându-se fie în memorie, fie în registrul destinatie corespunzator). Decodificarea instructiunilor poate însa continua atât timp cât mai exista locatii disponibile în RB.

Daca apare o exceptie, bufferul RB este golit, procesorul bazându-se pe contextul memorat *In Order* în setul de registri generali, dar si pe coerenta memoriei principale. Astfel, desi proceseaza *Out of Order*, procesorul superscalar implementeaza un mecanism de exceptii precise. Mecanismul este similar cu cel numit "*history buffer*". În general, capacitatea RB se stabileste pe baza simularii unei arhitecturi superscalare, pe diverse programe de test reprezentative (benchmark-uri) [Vin00].

Altfel spus, buffer-ul de reordonare retine numarul instructiunii aferent tuturor instructiunilor în curs de executie (trecute de faza de decodificare si care nu au scris înca rezultatele în setul de registri generali). Faptul ca numarul instructiunii este afisat cu litere negre pe fond alb, sugereaza ca instructiunea si-a încheiat executia si asteapta sa scrie rezultatul în setul de registri generali (parcurea fazei *commit/write-back*).

### 3.4.6 BUFFER-UL DE REDENUMIRE (*RENB*)

În urma decodificarii unei instructiuni, rezultatul acesteia este asignat unei locatii din buffer-ul de redenumire [Ung99, Bhan96], iar numarul registrului destinatie este asociat acestei locatii. În acest mod, registrul destinatie este practic redenumit printr-o locatie din RenB. Rezultatul instructiunii este disponibil abia în momentul în care unitatea de executie corespunzatoare îl genereaza. În urma decodificarii se creeaza prin hard un "tag" care reprezinta numele unitatii de executie care va procesa rezultatul instructiunii respective. Acest tag va fi scris în aceeași locatie din RenB. Din acest moment, când o instructiune urmatoare face referire la respectivul registru pe post de operand sursa, ea va apela în locul acestuia valoarea înscrisa în RenB sau, daca valoarea nu a fost înca procesata, tag-ul aferent locatiei. Daca mai multe locatii din RenB contin acelasi numar de registru (mai multe instructiuni în curs au avut acelasi registru destinatie), se va înscrie (în *fpr* sau *gpr*) locatia din RenB cea mai recent înscrisa (tag sau

valoare).

RenB se implementeaza sub forma unei memorii asociative, cautarea făcându-se după numărul registrului destinat la scriere, respectiv sursa la citire. Dacă accesarea RenB se soldează cu miss, atunci operandul sursa va fi citit din setul de registre. În caz de hit, valoarea sau tag-ul citite din RenB sunt memorate în SR corespunzătoare. Când o unitate de execuție generează un rezultat, acesta se va înscrie în SR și în locația din RenB care are tag-ul identic cu cel emis de către respectiva unitate. Rezultatul înscris într-o SR poate debloca anumite instrucțiuni aflate în așteptare. După ce rezultatul a fost scris în RenB, instrucțiunile următoare vor continua să-l citească din RenB ca operand sursa până când va fi evacuat și scris în setul de registre (în faza **commit/write-back** de procesare). Redenumirea unui registru cu o locație din RenB se termină prin evacuarea acestei locații.

Locațiile libere indică intrările de "*renaming*" care nu sunt folosite momentan. Cât timp este în uz o anumită locație, casuta va reține numărul instrucțiunii pentru instrucțiunea care va produce rezultatul și numele registrului arhitectural pentru care instrucțiunea așteaptă.

Dacă numărul instrucțiunii este colorat înseamnă că instrucțiunea nu și-a încheiat execuția. Un număr de instrucțiune care e afișat cu litere negre pe fond alb indică o instrucțiune care și-a încheiat execuția și a înaintat rezultatul instrucțiunilor dependente aflate în așteptare în stările de rezervare, dar care nu a scris rezultatele în setul de registre generali.

### 3.4.7. STATILE DE REZERVARE (SR)

Constituie o structură de date implementată în cadrul procesoarelor superscalare cu scopul facilitării procesării *Out of Order* a instrucțiunilor. Lansarea în execuție a unei instrucțiuni rezidentă în buffer-ul de prefetch înseamnă mutarea acesteia în SR aferentă. De aici, instrucțiunea se lansează efectiv în execuție atunci când toți operanzii săi sursa sunt disponibili. SR efectuează o redenumire dinamică a registrilor în vederea eliminării hazardurilor de date tip WAR și WAW. De asemenea aceste stări facilitează execuția prin captarea agresivă a unor rezultate așteptate de către instrucțiunile aflate în stare așteptare.

Stările de rezervare conțin instrucțiunile care așteaptă să își înceapă execuția sau a căror execuție este în curs. Primul număr care apare este numărul colorat al instrucțiunii. Instrucțiunile de salt și cele de tip *Store* nu primesc nici o culoare deoarece nu modifică valoarea unui registru arhitectural. Al doilea, respectiv al treilea număr, dacă sunt prezente, sunt

numerele colorate ale instructiunilor care furnizeaza operanzii sursa. Daca nu exista nici un numar sau daca toate numerele nu sunt colorate (litere negre pe fond alb), atunci instructiunea are disponibili toti operanzii necesari si deci asteapta sa intre în executie sau executia ei este în desfasurare. Instructiunile Load vor afisa "LD" în locul celui de-al doilea operand sursa. Aceasta se face pentru a specifica mai clar deosebirea dintre Load si Store. Fireste ca, instructiunile Load nu au niciodata mai mult de un operand sursa.

Instructiunile ramân în statiile de rezervare pâna ce faza lor de executie se încheie. O instructiune va fi executata de îndata ce toti operanzii ei sursa sunt disponibili. O unitate functionala de executie poate fi ocupata într-un ciclu de tact de catre o singura instructiune. În cazul concurenței la resurse (doua sau mai multe instructiuni de acelasi tip sunt gata de executie si doresc sa acceseze aceeasi unitate de functionala), i se va acorda prioritate acelei instructiuni care a devenit prima gata de executie.

O instructiune de numere întregi, în virgula mobila sau de salt poate începe executia doar în unitatea de executie corespunzatoare amplasata sub statia de rezervare în care a fost emisa. Tratarea instructiunilor cu referire la memorie este mult mai complexa. Acestea nu se pot executa out-of-order la fel de usor precum celelalte instructiuni. Dependentele cauzate de variabilele aflate în memorie reprezinta o frâna în calea cresterii performantei arhitecturilor de calcul. Exista motive ca o instructiune LD amplasata dupa o instructiune ST sa se execute înaintea acesteia din motive de eficienta a executiei (maskare latentă, reducerea necesarului de largime de banda a memoriei prin **bypassing**). Acest lucru este posibil numai daca cele 2 adrese de memorie sunt întotdeauna diferite. Este evident ca daca la un anumit moment ele sunt identice, semantica secventei se modifica inacceptabil. Atunci când acest lucru este posibil, problema se rezolva static, de catre compilator. Rutina de *dezambiguizare* (antialias), componenta a compilatorului, compara cele 2 adrese de memorie si returneaza una dintre urmatoarele 3 posibilitati:

- a) adrese întotdeauna distincte;
- b) adrese întotdeauna identice;
- c) cel puțin 2 adrese identice sau nu se poate determina.

Doar în primul caz putem fi siguri ca executia anterioara a instructiunii LD fata de instructiunea ST (sau simultana în cazul procesoarelor cu executie multipla) îmbunatateste performanta fara a cauza alterarea semantica a programului. Din pacate, nu se poate decide întotdeauna acest lucru în momentul compilarii.

Dezambiguizarea statica da rezultate bune în cazul unor adresari liniare si predictibile ale memoriei (accesari de tablouri, matrici). Prin

urmare un reorganizator de program bazat pe dezambiguizarea statica va fi deosebit de conservativ în actiunile sale. Când compararea adreselor de memorie se face pe parcursul procesarii programului prin hardware, se realizeaza o dezambiguizare dinamica. Aceasta este mai performanta decât cea statica dar necesita resurse hardware suplimentare si implicit costuri sporite [Vin00, Ste96]. Se considera ca progresele în aceasta problema pot duce la cresteri semnificative de performanta în domeniul paralelismului la nivelul instructiunilor.

În cadrul acestui simulator toate instructiunile Store sunt executate "in-order". În situatia în care doua instructiuni Store scriu în aceeași locatie de memorie, valoarea finala ramasa în memorie trebuie sa provina de la ultima (din punct de vedere lexical al aparitiei). De asemenea, toate instructiunile Load trebuie sa starteze executia înaintea oricarei instructiuni Store plasata ulterior (din punct de vedere al aparitiei în trace). Este necesar acest lucru pentru prevenirea încarcării unei valori eronate din memorie care abia mai târziu s-ar memora la respectiva adresa (dupa executia instructiunii Load). Toate instructiunile Store trebuie sa fie executate înaintea Load-urilor care urmeaza în trace, pentru a putea preveni citirea unei valori neactualizate din memorie. Doar instructiunile Load consecutive se pot executa "out-of-order". Nu conteaza ordinea în care se executa Load-uri consecutive care acceseaza aceeași locatie de memorie. Se va obtine aceeași valoare.

Statiile de rezervare aferente instructiunilor cu acces la memorie actioneaza ca o singura coada FIFO indiferent de câte unitati de executie pentru instructiuni cu referire la memorie exista. Din faza "issue" sunt expediate catre SR si unitatile functionale aferente instructiunilor Load / Store cât mai multe instructiuni posibil dar fara a viola regulile amintite anterior.

### **3.4.8. UNITATILE FUNCTIONALE DE EXECUTIE IMPLEMENTATE PIPELINE**

Unitatile de executie descriu progresul unei instructiuni prin structura pipeline de executie. Unitatile întregi si de branch au doar un singur stagi pipeline (executia respectivelor tipuri de instructiuni se face într-un ciclu de tact). Unitatile în virgula mobila prezinta 3 stagii pipeline si unitatile de memorie 2 stagii. Rezultatele sunt monitorizate de îndata ce instructiunea paraseste unitatea de executie pipeline, si orice instructiune dependenta care asteapta rezultatul poate fi executata in ciclul urmator.



Aparitia unui miss în cache-ul de date este indicata prin scrierea cu litere de culoare rosie sub unitatea de executie. S-ar putea spune ca miss-ul în D-Cache reprezinta a treia faza pipeline a unitatii de memorie. Doar instructiunile Load cauzeaza miss în cache-ul de date în aceasta versiune de simulator. Load-ul în cauza va ramâne pe nivelul al doilea al structurii pipeline un numar de cicli necesar aducerii blocului respectiv din memorie (penalitatea introdusa pentru miss în D-Cache).

Pentru a distinge instructiunile Load si Store, primele sunt marcate cu valoarea "LD" în spatiul dedicat celui de-al doilea operand sursa.

### 3.4.9. FAZA COMMIT (WRITE BACK)

Specifica numarul instructiunii care se va încheia si va scrie (write back) rezultatul în setul de registri generali în acest ciclu. Numarul maxim de instructiuni care pot face acest lucru este determinat de factorul superscalar al arhitecturii. Instructiunile se termina în ordinea în care au fost expediate spre executie. Instructiunile trebuie sa astepte în buffer-ul de reordonare (si redenumire) pâna la terminarea tuturor instructiunilor precedente. Dupa ce procesarea unei instructiuni se încheie, au loc urmatoarele evenimente:

1. Instructiunea (indexul ei) este înlaturata din buffer-ul de reordonare si locatia respectiva poate fi folosita de o alta instructiune în acelasi ciclu.
2. Daca instructiunea foloseste o intrare în buffer-ul de redenumire, este eliberata respectiva locatie, putând fi folosita de catre o alta instructiune (care a trecut de faza *issue*) în acelasi tact.
3. Daca instructiunea respectiva reprezinta ultima redenumire pentru registrul arhitectural destinatie, atunci rezultatul ei este scris în setul de registri generali, acest fapt fiind sesizat prin disparitia culorii din locatia aferenta în setul de registri generali.
4. Numarul de instructiuni complet procesate este incrementat în partea superioara a ecranului.

În urma descrierii functionarii microarhitecturii se desprind câteva întrebări evidente:

#### Unde este faza write back?

Simulatorul este modelat în linii mari dupa arhitectura procesorului Power PC. În cadrul acestui procesor majoritatea instructiunilor executa fazele *commit* si *write back* în acelasi ciclu de tact. Faza etichetata ca si *commit* arata de fapt instructiunile care efectueaza practic *faza write back*.

Aceasta politica este una conservatoare, cu repercursiuni defavorabile asupra performantei.

#### ❏ De ce nu sunt afisate instructiuni în faza fetch instructiune?

Fiecare faza de procesare a simulatorului evidentiaza instructiunile prezente în structura pipeline. La începutul fiecarui ciclu de tact unitatea de fetch detine întreaga linie din cache-ul de instructiuni; singura informatie afisata în faza de fetch este daca avem miss în cache-ul de instructiuni sau un branch gresit predictionat. Instructiunile care au fost aduse cu succes din cache/memorie sunt afisate abia în faza de decodificare, în urmatorul ciclu de procesare (din punctul de vedere al procesarii reale este normal, întrucât în faza de fetch nu se cunosc nici operanzi, nici codul operatiei, ci doar PC-ul corespunzator fiecărei instructiuni care va fi aduse).

#### ❏ De ce întârzie procesul de fetch-instructiune la un branch gresit predictionat?

Într-un procesor real, unitatea de fetch nu întârzie dupa un branch gresit predictionat. Mai mult, unitatea de fetch va continua sa aduca instructiuni de pe calea incorecta si sa le transmita fazei de decodificare. De îndata ce unitatea de executie a branch-ului a determinat ca instructiunea de salt a fost gresit predictionata, toate aceste instructiuni vor fi sterse împreuna cu eventualele lor efecte. În cazul de fata, metodologia de simulare fiind "trace-driven", singurele instructiuni disponibile sunt cele de pe calea corecta.

SATSim simuleaza branch-ul gresit predictionat întârziind unitatea de fetch pâna când acesta își încheie executia. În acest mod se implementeaza penalizarea în "*timing*" pe care o predictie eronata ar determina-o, chiar daca nu în aceasta maniera simplista.

### 3.5. DISFUNCTIONALITATI RECUNOSCUTE ÎN ACEASTA VERSIUNE A SIMULATORULUI

- Fonturile caracterelor sunt afisate diferit pe fiecare tip de "calculator" pe care a fost rulat programul. Ocazional pot apare efecte secundare, dar acestea sunt doar de natura estetica.
- Procesorul PowerPC 620 implementeaza un cache tip *non-blocking* [Bhan96, Song94], în sensul ca, memoria cache de date poate fi atacata de catre o instructiune Load cu hit, în timp ce deserveste un miss anterior. Daca apare un nou miss (în timp ce mai exista unul la cache-ul

de date), instructiunea în cauza este retinuta în statiile de rezervare aferente instructiunilor cu referire la memorie si va fi lansata în executie abia dupa ce primul miss a fost rezolvat. În cazul simulatorului SATSim unitatea functionala de executie aferenta instructiunilor cu referire la memorie este partial "*non-blocking*" (pentru instructiuni Store) si partial "*blocking*" (pentru instructiuni Load). Daca apare un miss la o instructiune Load, si exista o alta în statiile de rezervare cu operanzii disponibili, dar cu un index ulterior primeia, se va astepta pâna la rezolvarea miss-ului - *blocking*. În opinia realizatorilor acestui simulator, cea mai buna alternativa consta în folosirea unei unitati de executie "*non-blocking*" pentru ambele tipuri de instructiuni cu referire la memorie: Load si Store.

- Rata de miss în cache-ul de instructiuni este exprimata prin accese cu miss per linia de instructiuni extrasa. Rata de miss ar trebui interpretata ca si accese cu miss per instructiune efectiv.
- Din punct de vedere al interfetei software a simulatorului, buffer-ul de reordonare contine instructiunile într-o ordine diferita de cea corecta (Ex: 5, 3, 4, 6, 7, 8 - vezi figura 3.5, ciclul 8 de procesare), însa retragerea (evacuarea) din RB se face conform principiului FIFO (asa cum trebuie facut de fapt). Totusi, acest amanunt poate deruta utilizatorul.

### 3.6. CONCLUZII SI PROIECTE DE VIITOR

SATSim este de un real folos pentru înțelegerea conceptelor moderne aferente arhitecturii superscalare. Dupa cum intentioneaza autorii simulatorului, viitoarea versiune a simulatorului SATSim ar trebui sa înglobeze un sistem de ajutor - "**help-on-line**", un "**browser**" care sa faciliteze selectia fisierelor de intrare, respectiv de iesire, un sistem de vizualizare în timp real al gradului de utilizare a resurselor etc. Actuala versiune a simulatorului SATSim poate fi descarcata gratuit de pe Internet de la adresa: <http://www.ece.gatech.edu/research/pica/SATSim/satsim.html>

Simulatorul SATSim este însoțit de un pseudo-translator/simulator la nivel de executie - **PseudoExSAT**, scris în CBuilder versiunea 3.0 sub îndrumarea unuia dintre autorii acestei lucrari (Adrian FLOREA) de catre un grup de studenti calculatoristi (vezi Anexa 2). PseudoExSAT primeste ca si intrare codul sursa asamblare în format MIPS - compatibil SATSim - al unui program de test, îl executa si genereaza trace-ul aferent, necesar simulării "*hibride*" cu ajutorul SATSim. Setul de instructiuni implementate

la ora actuala cuprinde instructiunile aritmetico-logice (partial), de salt si cele cu referire la memorie (partial). Referitor la PseudoExSAT în viitor se intentioneaza implementarea tuturor tipurilor de instructiuni - apeluri sistem, instructiuni înmultire / împartire flotant si o eventuala colaborare cu colectivul de cercetare de la Georgia Institute of Technology. Actuala versiune a simulatorului PseudoExSAT poate fi descarcata gratuit de pe Internet de la adresa: <http://vectra.ulbsibiu.ro/~adrian.florea>

### 3.7. PROBLEME PROPUSE SPRE REZOLVARE

- A.** Anexa 1 de la sfârșitul lucrării descrie functionarea simulatorului pe o secvența de instructiuni din trace-ul “*compr1.tra*” în primii 11 cicli de tact. S-a ales o configuratie arhitecturala modesta (**FR=IR=2**, **Nr.unit.exec=1** (pe fiecare tip de instructiune), **Nr. statii de rezervare / unitati functionala** = 3, **RenB=30** locatii, **RB=38** locatii etc.). Descrieti analog simularea instructiunilor în primii 15 cicli de tact pe urmatoarea configuratie arhitecturala:
- a) **FR=IR=4**, **Nr.unit.exec** = 2 (Int/Float)  
                     **Nr.unit.exec** = 1 (Branch/Mem)  
                     Ceilalti parametri sunt cei folositi în Anexa D.
  - b) **FR=IR=4**, **Nr.unit.exec** = 2 (toate tipurile de instructiuni)  
                     Ceilalti parametri sunt cei folositi în Anexa D.
  - c) **FR=IR=4**, **Nr.unit.exec** = 1 (toate tipurile de instructiuni)  
                     **Nr. statii de rezervare / unitati functionala** = 2  
                     Ceilalti parametri sunt cei folositi în Anexa D.
  - d) Analog cu a) dar fortati în ciclul 4, 8 si 10 accese cu miss la cace-ul de instructiuni.
- B.** Studiati efectele acuratetii de predictie a salturilor si efectul ratei de hit în cache asupra performantelor arhitecturii superscalare.
- C.** Determinati optimul de performanta din punct de vedere al ratei de procesare, variind urmatorii parametri arhitecturali: factorul superscalar al arhitecturii, numarul de intrari în “buffer-ul de *renaming*”, numarul de intrari în buffer-ul de *reordonare*, numarulde statii de rezervare per unitate de executie, numarul unitatilor de executie.
- D.** Scrieti doua programe (Suma a *n* numere si Determinarea maximului dintre 3 numere) în asamblare MIPS, generati trace-urile corespunzatoare cu pseudo-translator/simulator-ul **PseudoExSAT** si simulati-le cu ajutorul SATSim. Verificati corectitudinea functionarii programelor simulate.

## 4. SIMULAREA INTERFETEI PROCESOR - CACHE PENTRU O ARHITECTURA RISC SUPERSCALARA PARAMETRIZABILA

---

### 4.1. SCOPUL LUCRARI

Tematica acestei lucrari o constituie **modelarea si simularea procesului de scriere** în cache prin prisma celor doua strategii posibile: *write back* si *write through*. De asemenea, se urmareste descrierea si utilizarea unui simulator parametrizabil pentru o arhitectura superscalara.(determinarea diferitilor parametri de performanta, configuratii optime, pentru o arhitectura Harvard de memorie - caracterizata prin spatii separate pentru instructiuni si date cât si prin bus-uri separate între procesor si memoria cache de date respectiv de instructiuni).

### 4.2. MEMENTO TEORETIC

Una din cele mai studiate zone ale cercetarii, proiectarii si implementarilor actuale în ingineria calculatoarelor o reprezinta problematica procesarii paralele la nivelul instructiunilor masina. Limitarea principala a performantei arhitecturilor RISC la o instructiune / ciclu masina, a determinat aparitia conceptului de masina cu executii multiple ale instructiunilor (MEM) având ca principal deziderat depasirea barierei de o instructiune / tact si atingerea unor rate de procesare de mai multe instructiuni / tact. Procesoarele MEM extind paralelismul temporal de tip pipeline la unul spatial bazat pe aducerea si executia simultana a mai multor instructiuni masina. Evident ca acest model presupune existenta mai multor unitati functionale de procesare. Arhitecturile MEM sunt implementate în

doua variante distincte: procesoare *superscalare* si respectiv procesoare *VLIW* (very long instruction word).

În varianta superscalara cade în sarcina hardului sa verifice independenta instructiunilor aduse în buffer-ul de prefetch si sa le lanseze în executii multiple spre unitatile de executie pipeline-izate. Caracterizate de o complexitate hardware deosebita determinata de detectia si solutionarea hazardului între instructiuni, procesoarele superscalare sunt limitate în posibilitatea de a exploata paralelismul la nivelul instructiunilor, de capacitatea limitata a buffer-ului de prefetch [Vin00].

La procesoarele VLIW si EPIC, mai rare decât cele superscalare în implementarile comerciale, cade în sarcina compilatorului de a reorganiza programul original în scopul “împachetarii” într-o singura instructiune multipla a mai multor instructiuni RISC primitive si independente, care vor fi alocate unitatilor de executie în conformitate stricta cu pozitia lor în instructiunea multipla. Avantajul principal fata de modelul superscalar consta în simplitatea hardware, în special în ceea ce priveste logica de detectie a hazardurilor si lansare în executie [Vin00].

Decalajul accentuat între perioada de tact procesor si timpul mediu de acces la memoria principala au determinat utilizarea eficienta a unui sistem ierarhic de memorie. Sistemul de memorie la procesoarele superscalare este ierarhizat pe câteva nivele - fiecare mai mic, mai rapid si mai scump per byte decât nivelul urmator. Cache este numele dat în general primului nivel al memoriei ierarhice întâlnit odata ce adresa paraseste CPU si reprezinta un mecanism arhitectural destinat sa acopere decalajul amintit anterior prin satisfacerea celor mai multe accese la memoria RAM (latenta ridicata) la o viteza egala cu cea a procesorului. Termenul de cache e folosit oricând sunt cautate spre a fi refolosite elemente accesate anterior pe parcursul procesarii. Unitatea minima de informatie, care poate fi prezenta (*hit*) sau nu (*miss*) în cache, se numeste bloc. În functie de modul de plasare al blocurilor în cache, acestea se clasifica în:

- ☐ **Cu mapare directa (direct mapped)** - daca fiecare bloc are doar un loc unde poate aparea în cache. Maparea în cache se face dupa formula:  

$$(Adresa\ blocului) \bmod (Numarul\ de\ blocuri\ în\ cache).$$
- ☐ **Complet asociative (full associative)** - daca blocul poate fi plasat oriunde în cache.
- ☐ **Semi-asociative (set associative)** - daca blocul poate fi plasat numai într-un set predeterminat, dar oriunde în acest set. Un set este un grup de blocuri într-un cache. Un bloc este întâi mapat într-un set, si apoi el poate fi plasat oriunde în interiorul setului. Setul e de obicei ales astfel:

$$(Adresa\ blocului) \bmod (Numarul\ de\ seturi\ în\ cache).$$

Daca avem  $n$  blocuri într-un set, cache-ul se numeste în literatura de specialitate,  **$n$  - way associative**.

Un cache *direct mapped* este un cache simplu, *one - way associative*, iar un cache *full associative* considerat ca fiind un singur set cu  $m$  blocuri poate fi numit *m - way set associative*. Cu alte cuvinte, un cache *direct mapped* poate fi gândit sa aiba  $m$  seturi de asociativitate 1 si *full associative* sa aiba doar un singur set, de asociativitate  $m$ . Majoritatea cache-urilor integrate în procesor sunt de tip *direct mapped*, *two - way associative*, *four - way associative*.

Verificarea existentei blocurilor în cache se face dupa *tag*. Cache-urile au un tag (identificator) de adresa în fiecare *block frame*. Tag-ul fiecarui bloc din cache, care poate sa contina informatia dorita este verificat daca se potriveste cu adresa blocului din cadrul adresei (fizice sau virtuale) emise de catre procesor. Ca o regula, toate tag-urile posibile sunt cautate în paralel pentru ca viteza este critica. Trebuie verificat daca informatia dintr-un bloc este sau nu valida. Procedura cea mai folosita este de a adauga un **bit de validare (V)** la tag, pentru a spune daca aceasta intrare contine sau nu adresa valida. Daca bitul nu e setat, atunci nu putem avea hit la aceasta adresa, informatia fiind invalida (scriere DMA, multimicroprocesare) chiar daca "tag-urile" coincid. Bitul are rol în mentinerea coerentei cache-urilor în sistemele multiprocesor.

Accesele la cache pot fi cu hit, la potrivirea tag-ului si  $V=1$ , sau miss, în caz contrar. În ultimul caz, controller-ul de cache trebuie sa selecteze un bloc pentru a fi înlocuit cu data dorita - proces de evacuare din cache. La cache-urile mapate direct, decizia hardware e simplificata în asa fel încât doar un singur *block frame* e verificat pentru hit si numai blocul care poate fi înlocuit. La cache-urile *full associative* sau *set associative*, vor exista mai multe blocuri candidate la evacuare. Exista multe strategii de baza pentru selectarea blocului care trebuie înlocuit, amintim aici doar urmatoarele:

- ☐ **Random** (aleatoare) - pentru a întinde alocarea uniforma, blocurile candidate sunt selectate aleator. Un atu al înlocuirii aleatoare este ca e simplu de implementat în hardware.
- ☐ **Least Recently Used** (cel mai putin recent folosit) - pentru a reduce sansa de a evacua o informatie de care va fi nevoie în curând, blocul înlocuit este cel nefolosit de cel mai mult timp. LRU face uz de *principiul de localitate*: Daca exista probabilitate mare ca blocurile recent folosite sa fie folosite din nou, atunci cel mai bun candidat pentru transfer este blocul LRU.
- ☐ O politica de înlocuire în ordinea **FIFO** (*first in first out*) poate da roade în anumite contexte desi în general este mai putin performanta decât anterioarele, conform literaturii [Hen96].

Prin anii 60, *Belady* a propus **un algoritm teoretic cvasi-optimal de evacuare** (OPT) [Bel66]. Algoritmul OPT, înlocuieste întotdeauna blocul care va fi adresat cel mai târziu în viitor (eventual nu va mai fi adresat deloc). Un astfel de algoritm s-a dovedit a fi cvasi-optimal pentru toate pattern-urile de program, ratele de miss fiind cele mai mici în acest caz, dintre mai toate politicile de înlocuire folosite. Politica se dovedeste optima doar pentru fluxuri de instructiuni *read-only*. Pentru cache-urile cu modalitate de scriere *write-back* algoritmul de înlocuire nu este întotdeauna optim (spre exemplu poate fi mai costisitor sa se înlocuiasca blocul cel mai târziu referit în viitor daca blocul trebuie scris si în memoria principala, fiind "*murdar*", fata de un bloc "*curat*" referit în viitor putin mai devreme decât blocul "*murdar*" anterior; în plus, blocul curat nu mai trebuie evacuat, ci doar supra-scris). Este evident un **algoritm speculativ**, practic imposibil de implementat în practica. Totusi el are doua calitati majore: (1) reprezinta o metrica de evaluare teoretica a eficientei algoritmului de evacuare implementat în realitate, absolut necesara si (2) induce ideea fecunda a predictibilitatii valorilor de folosinta ale blocurilor din cache, conducând astfel la algoritmi predictivi rafinati de evacuare (vezi memoria *Selective Victim Cache*, în capitolul urmator, care implementeaza un astfel de algoritm). Mai nou, se implementeaza **algoritmi de evacuare predictivi**, care anticipeaza pe baze euristice utilitatea de viitor a blocurilor memorate în cache, evacuându-l pe cel mai putin valoros.

Cu privire la politica de scriere în cache se disting doua strategii posibile: *write back* si *write through*. Write back e preferata în majoritatea implementarilor actuale deoarece scrierea are loc la viteza memoriei cache iar multiplele scrieri în bloc necesita doar o scriere în nivelul cel mai de jos al memoriei. Cu *write through*, informatia e scrisa în ambele locuri: în blocul din cache si în blocul din memoria principala. Prin *write back* informatia e scrisa doar în blocul din cache. Write back implica evacuare efectiva a blocului - cu penalitatile de rigoare - în memoria principala. Rezulta ca este necesar un bit **Dirty** asociat fiecarui bloc din cache-ul de date. Starea acestui bit indica daca blocul e *Dirty* (modificat cât timp a stat în cache), sau *Clean* (nemodificat). Daca bitul este "curat", blocul nu e scris la miss, deoarece nivelul inferior – memoria principala - contine copia fidela a informatiei din cache. Daca avem *citire din cache cu miss* si *Dirty* setat pe '1' atunci vom avea o penalizare egala în timp cu timpul necesar evacuarii blocului - existent în cache dar nu cel solicitat - la care se adauga timpul necesar încarcarii din memorie în cache a blocului necesar în continuare. În cazul strategiei *write through* nu exista practic evacuare de bloc, dar exista penalitati la fiecare scriere în memorie în lipsa unui procesor specializat de iesire (*Data Write Buffer*). *Write through* are de asemenea avantajul ca,



urmatorul nivel inferior are majoritatea copiilor curente ale datei. Acest lucru e important pentru sistemele de intrare / iesire (I/O) si pentru multiprocesoare în vederea pastrarii coerentei variabilelor stocate în cache. Dispozitivele I/O si multiprocesoarele au oarecum cerinte contradictorii: ele vor sa foloseasca *write back* pentru cache-ul procesorului si pentru a reduce traficul memoriei si vor sa foloseasca *write through* pentru a pastra cache-ul consistent cu nivelul inferior al ierarhiei de memorie. Oricare din cele doua strategii de scriere pot fi asociate cu cele de mentinere a coerentei cache-urilor în cadrul sistemelor multiprocesor, anume strategia "*write invalidate*" respectiv "*write broadcast*".

Dintre metodele de îmbunatatire a performantei cache-urilor se disting [Hen96]:

1. Reducerea ratei de miss în cache
2. Reducerea penalitatii în caz de miss în cache
3. Reducerea timpului de executie în caz de hit în cache.

Cu privire la accesele cu miss în cache exista trei tipuri de miss:

- ☐ **Compulsory** - sau de start rece. Apar la primul acces al unui anumit bloc în cache.
- ☐ De **capacitate** - apar în situatia în care cache-ul nu poate retine toate blocurile necesare iar cele care sunt evacuate la un moment dat vor fi necesare ulterior.
- ☐ De **conflict** - sau de interferenta. Apar daca strategia de plasare a blocurilor în cache este mapata direct sau semi-asociativa, situatie în care mai multe blocuri din memoria principala pot interfera (accesa) acelasi bloc din cache.

Dintre metodele de reducere a ratei de miss în cache amintim:

- ⇒ Cresterea dimensiunii blocurilor
- ⇒ Cresterea gradului de asociativitate
- ⇒ Utilizarea conceptului de *victim cache* respectiv *selective victim cache* (a se vedea capitolul urmator)
- ⇒ Utilizarea mecanismelor de prefetch asupra instructiunilor (*buffer de prefetch*) si datelor (*data write buffer*)
- ⇒ Folosirea tehnicilor de optimizare gen (*merging array, loop interchange, loop fusion*)

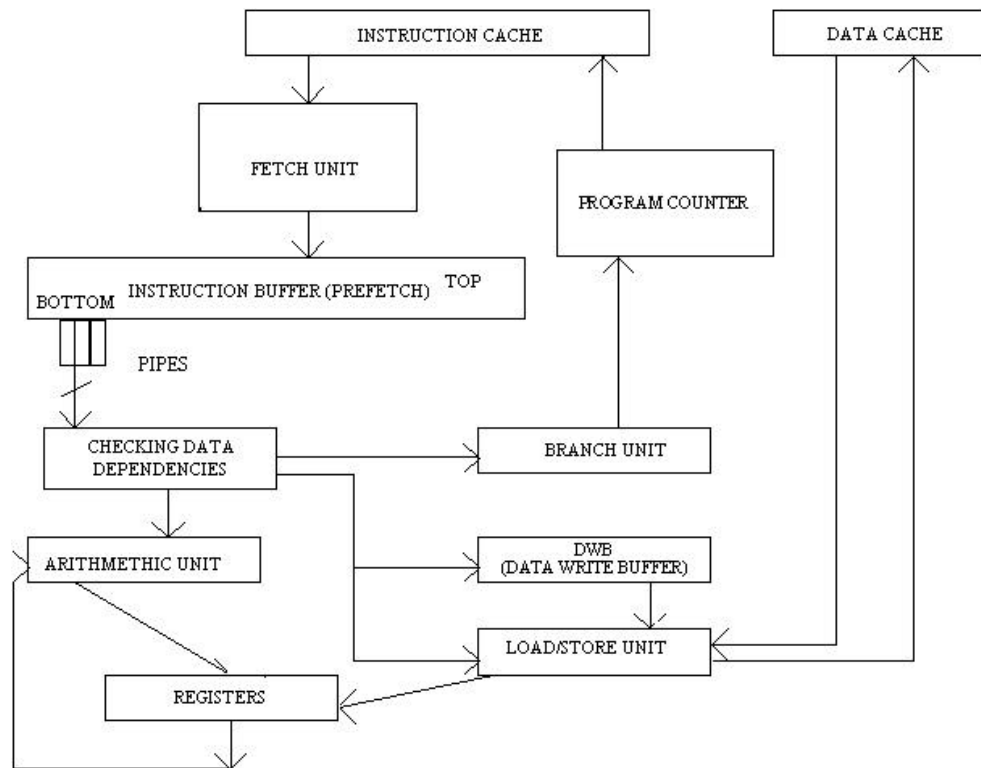
### 4.3. DESFĂȘURAREA LUCRĂRII

#### 4.3.1. DESCRIEREA SIMULATORULUI

##### 4.3.1.1. ELEMENTELE SIMULATORULUI. SCHEMA BLOC

Arhitectura superscalara are patru nivele distincte în procesarea instrucțiunilor. În nivelul **IF** (fetch instrucțiune) - se calculează adresa grupului de instrucțiuni ce trebuie citite din cache-ul de instrucțiuni sau din memoria principală; după citire, blocul de instrucțiuni este plasat în partea superioară a buffer-ului de prefetch. Instrucțiunile din partea inferioară a buffer-ului sunt selectate și trimise celui de-al doilea nivel: **ID** (decodificare instrucțiune) - în care sunt decodificate instrucțiunile aduse, se citesc operanții din setul de registre generali, se calculează adresa de salt (pentru instrucțiunile de ramificație) și respectiv se calculează adresa de acces la memorie (pentru instrucțiunile LOAD sau STORE). Instrucțiunile sunt apoi pasate unităților funcționale potrivite, care folosesc operanții sursă în timpul celei de-a treia faze de procesare a pipe-ului: **ALU/MEM**. În această fază se execută operația ALU asupra operanților selectați în cazul instrucțiunilor aritmetico-logice și se accesează memoria cache de date sau memoria principală (în caz de miss în cache), pentru instrucțiunile LOAD sau STORE. Unitățile de execuție aferente instrucțiunilor LOAD și STORE sunt singurele unități funcționale care interacționează în mod direct cu cache-ul de date. În final, avem cel de-al patrulea nivel **WB** (scriere date) - în care, unitățile funcționale preiau rezultatul final al instrucțiunilor aritmetico-logice sau data citită din memorie, și o depun pe magistrala rezultat, de unde este copiată în registrul destinație din setul de registre generali [Col93].

Simulatorul realizat trebuie să elimine gâtuirile care limitează performanța și să investigheze posibile schimbări (arhitecturale sau tehnici de optimizare) în scopul creșterii acesteia. Prin realizarea unui model de simulare detaliat pentru fiecare procesor, performanța obținută prin simulare este capabilă să asigure un rapid *feedback* în legătură cu schimbările propuse.



**Figura 4.1.** Schema bloc a arhitecturii superscalare simulate

Principalii parametri ai simulatorului, ce pot fi modificati de catre utilizator sunt [VinFlor99, VinFlor00]:

- **FR (rata de fetch)** - defineste marimea blocului accesat din cache-ul de instructiuni, mai precis numarul de instructiuni citite simultan din cache sau memorie (în caz de miss în cache) într-un ciclu de tact; poate lua valori de 4, 8 sau 16 instructiuni.
- **IBS (instruction buffer size)** - dimensiunea buffer-ului de prefetch, masurata în numar de instructiuni; plaja de valori: 4 (minim FR, altfel, nici o instructiune nu va putea fi plasata cu succes în buffer), 8, 16, 32, ≤ capacitatea cache-ului de instructiuni; buffer-ul de prefetch este o coada ce lucreaza dupa principiul FIFO (*first in first out*). Vor fi citite FR instructiuni simultan de la adresa specificata de PC (program counter) si depuse în partea superioara a buffer-ului. În acelasi ciclu de executie, instructiuni din partea inferioara sunt expediate spre unitatile de decodificare si executie. O intrare în buffer va contine câmpurile:
  - OPCODE - codul operatiei executata de instructiunea respectiva;
  - PC\_crt - adresa (Program Counter-ul) instructiunii curente;

DATE / INSTR - adresa din / la care se citesc / se scriu date din sau în memorie, în cazul instructiunilor cu referire la memorie, respectiv adresa instructiunii tinta în cazul instructiunilor de salt.

- **capacitatea memoriilor cache** care variaza între 64 cuvinte si 16 Kcuvinte. Aceste capacitati relativ mici ale memoriilor cache se datoreaza particularitatilor benchmark-urilor Stanford utilizate. Acestea folosesc o zona restrânsa de instructiuni, limitata la cca. 2 Ko si o zona de date mai mare dar mai "rarefiata", care se întinde pe un spatiu de cca 24 Ko. Cache-ul de date este parametrizabil din punct de vedere al porturilor de citire / scriere (uniport / biport - **NR\_UNIT\_LS=1÷2**), favorizând executia a doua instructiuni cu referire la memorie de genul: Load + Load sau Load + Store.
- **IRmax (issue rate maxim)** - numarul maxim de instructiuni, lansate în executie simultan într-un ciclu de executie, din buffer-ul de prefetch. Poate lua valorile: 2, 4, 8, 16 (maxim FR) instructiuni. Daca rata de fetch este mai mica decât numarul maxim de instructiuni executate concurrent într-un ciclu, atunci performanta este limitata de procesul de *fetch instructiune*. Simulatorul implementat considera executia instructiunilor "in order" - ordinea initiala a instructiunilor. O instructiune va fi executata abia dupa ce toate celelalte instructiuni anterioare, de care ea depinde au fost executate.
- **Strategia de scriere în cache:** *write back* sau *write through*. Cu *write through*, informatia e scrisa atât în blocul din cache cât si în blocul din memoria principala. Prin *write back* informatia e scrisa doar în blocul din cache.
- **Utilizare / neutilizare DWB (data write buffer)** - o coada FIFO de lungime parametrizabila, a carei valoare trebuie sa fie minim IRmax. Fiecare locatie contine adresa de memorie (virtuala) si data de scris. Consideram ca DWB contine suficiente porturi de scriere pentru a sustine cea mai defavorabila situatie (STORE-uri multe, independente si simultane în fereastra IRmax fiind deci posibile), oferind deci porturi de scriere virtuale multiple. În schimb D-Cache va contine un singur port de citire (LOAD) si un singur port de scriere (STORE), reflectând o situatie cât mai reala. Cu DWB sunt posibile deci STORE-uri simultane, fara el acestea trebuind serializate cu penalitatile de rigoare. În plus DWB va putea rezolva prin "bypassing" foarte elegant hazarduri de tip "LOAD after STORE" cu adrese identice, nemaifiind deci necesara accesarea sistemului de memorie de catre instructiunea LOAD.

- ❑ **Memoria principala** - (care se acceseaza numai la *miss* în cache) va avea o latenta parametrizabila de **N\_PEN** (10, 15, 20) impulsuri de tact procesor.
- ❑ Se presupune existenta unui numar suficient de mare (maxim **IRmax**) de **seturi de registri generali (NR\_REG\_GEN)**: un set de registri generali este necesar pentru executia unei instructiuni de tip aritmetico-logic sau cu referire la memorie.
- ❑ Se presupune o predictie perfecta a instructiunilor de ramificatie, adica cunoasterea în permanenta a adresei corecte a urmatoarei instructiuni ce se va executa. Mecanismul de *forwarding* este inhibat.

#### 4.3.1.2. INTERFATA CU UTILIZATORUL. DESCRIEREA RESURSELOR SOFTWARE

În acest mini ghid de implementare / utilizare nu se va insista asupra conceptelor de simulare care au stat la baza implementarii acestui simulator, accentul punându-se pe detalierea procesului de elaborare / concepere / implementare a software-ului. Metodologia de simulare este *trace driven simulation*. Simulatorul (aplicatia software) executa multithreaded instructiunile benchmark-urilor Stanford (algoritmi de sortare - *bubble*, *quicksort*, *tree*, problema turnurilor din Hanoi, problema celor 8 regine etc.) compilate pe arhitectura HSA (Hatfield Superscalar Architecture).

Simulatorul este conceput într-o maniera moderna si este implementat folosind mediul Borland Delphi® versiunea 5 care ofera un suport puternic pentru programarea orientata pe obiecte: mosteniri multiple, supraîncarcarea operatorilor si functiilor, clase prieten etc. Conceptele de mostenire si polimorfism creeaza premisele dezvoltarii ulterioare (extinderii) a variantei actuale de simulator. De asemenea, implementarea a fost astfel realizata încât, orice modificare (adaugare) în hardware sau software sa fie facuta cu minim de efort. Interfata cu utilizatorul, caracterizata de ferestre de dialog, imagini grafice edificatoare este "*user-friendly*" etc.

Simulatorul este organizat în jurul unei ferestre principale, care contine toti parametrii de simulare, permite controlul simularii si afiseaza rezultatele simularii. Pe lângă fereastra principala mai exista o resursa similara, dedicata afisarii graficelor rezultate în urma simularii.

Aplicatia este conceputa pe baza unui fir principal, care prin actiunea utilizatorului asupra unor butoane dedicate, lanseaza în executie fire secundare, în functie de trace-urile si optiunile de simulare selectate de utilizator.

Elementul de baza al simulatorului îl constituie clasa **TSimThread**. Aceasta clasa contine declaratiile unui fir de executie, al carui scop este acela de a realiza simularea completa, functie de parametrii de intrare. Structurile de date folosite de program sunt definite ca attribute membre ale clasei **TSimThread**, instantiind clasele sau diverse variabile definite global. Astfel, pentru memorarea unei instructiuni s-a utilizat structura **TInstructiune**, care contine trei câmpuri, **TipInstr** (tipul instructiunii), **Adr\_Crt** (PC-ul curent) si **Adr\_Dest** (adresa de salt respectiv adresa locatiei de memorie).

Structurile **TDCacheEntry** si **TICacheEntry** sunt utilizate pentru implementarea cache-urilor de date respectiv de instructiuni, prin folosirea unor tablouri dinamice derivate din aceste tipuri. Structura **OneFetch** retine un grup de FR instructiuni citite din cache sau memorie (în caz de miss în cache), Buffer-ul de prefetch (**IB**) este o coada ce lucreaza dupa principiul FIFO. Vor fi citite FR instructiuni simultan (instructiunile din **OneFetch**) de la adresa specificata de PC si depuse în partea superioara a buffer-ului. În acelasi ciclu de executie, instructiuni din partea inferioara sunt expediate spre unitatile de decodificare si executie.

Pe lânga structurile anterior mentionate s-au mai folosit un numar de variabile si constante în functie de necesitati (dimensiune cache-uri, rata de fetch, latente instructiuni / accese la memoria principala , variabile pentru memorare rezultate etc.)

Simularea se realizeaza în urmatoarii pasi:

1. Se apeleaza functia **StartSimulare**. Aceasta realizeaza simularea completa a trace-ului, de la citirea instructiunilor din trace si pâna la generarea rezultatelor. Functia apeleaza metodele clasei **TSimThread**.
2. Se apeleaza metoda **Initializari**, care transfera în variabilele de program (parametrii de simulare) valorile controalelor utilizator ale formei de configurare.
3. **Atâta timp cât** mai exista instructiuni în trace, se vor executa ciclic urmatoarii pasi:
  4. Se aduc din fisierul trace instructiuni FR instructiuni si se memoreaza în structura **OneFetch**.
  5. In momentul în care buffer-ul **OneFetch** este plin se apeleaza functia **Check\_ICache** pentru a verifica daca accesul în cache de instructiuni este cu hit sau miss (încarcarea / evacuarea se face la nivel de bloc).
  6. Se intra într-o bucla în care se asteapta adaugarea buffer-ului de instructiuni **OneFetch** in buffer-ul de prefetch, apelându-se metoda **AdaugaOneFetchInIB** pâna când rezultatul returnat de functie este afirmativ. La fiecare ciclu al buclei (corespunzator unui ciclu de tact)

se executa maxim  $IR_{max}$  instructiuni din buffer-ul de prefetch, apelându-se metoda **Executa InstrDinIB**. Aceasta functie va verifica si va penaliza accesele in cache-ul de date conform cu strategia aleasa.

7. In momentul in care se semnalizeaza de catre metoda **GetInstructiuneDinTrace** sfarsitul trace-ului, se vor executa toate instructiunile din buffer-ul de prefetch.

Pe tot parcursul acestui ciclu se actualizeaza informatia furnizata de indicatorul de progres; de asemenea se verifica în permanenta apasarea butonului de oprire a simularii pentru a evita blocarea programului si distrugerea thread-ului de simulare.

Dupa executia si simularea completa a trace-ului se vor afisa rezultatele in tabelul de rezultate, in coloana specifica, apelandu-se metoda **UpdateScreenInfo** in mod sincronizat cu firul principal pentru a evita conflictele la resursele programului (tabelul cu rezultate in speta).

In functie de trace-urile selectate se creeaza cate un thread cu prioritate Higher pentru fiecare trace, pornindu-se simularea paralela a trace-urilor selectate.

Pentru generarea graficelor se folosesc exact aceleasi metode, valorile necesare afisarii graficelor fiind memorate in structurile specifice (tablouri dinamice) si preluate din tabelul cu rezultate.

#### 4.3.1.3. INTERFATA CU UTILIZATORUL. GHID DE UTILIZARE.

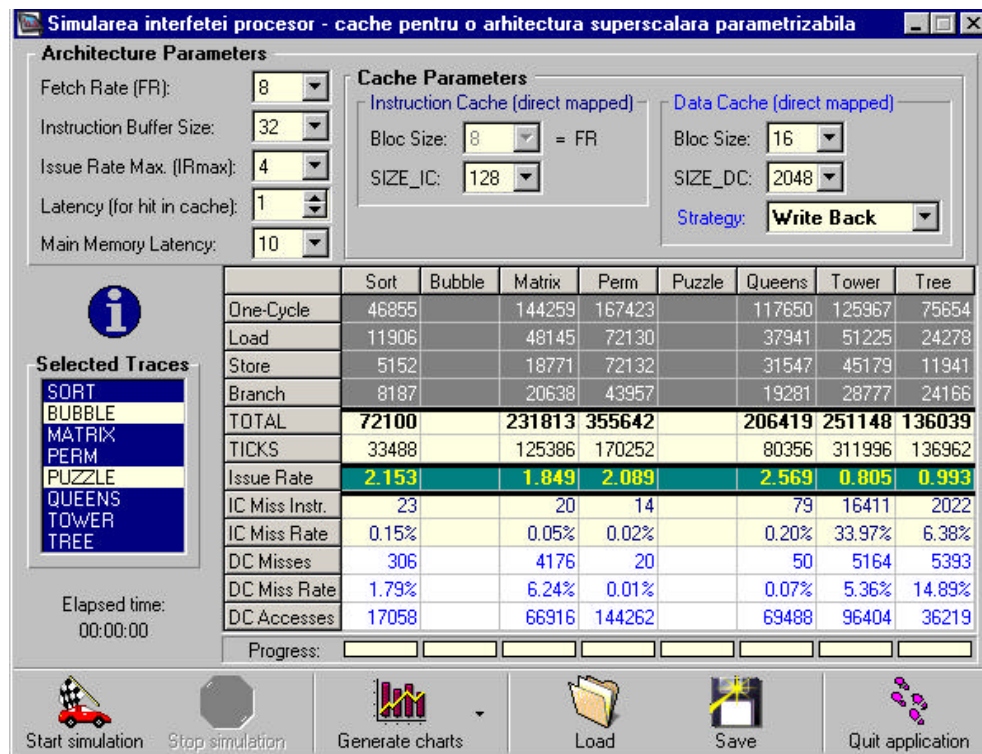
Fereastra principala a simulatorului (vezi figura 4.2) contine toate elementele necesare simularii. S-a optat pentru aceasta varianta din motive ergonomice, si anume de a avea în orice moment la indemâna absolut toti parametrii de simulare împreuna cu paleta de butoane de control a simularii.

În partea superioara a ferestrei se afla zona dedicata configurarii parametrilor de simulare (rata de fetch, dimensiune buffer de prefetch, configurare cache-uri, strategia de scriere etc.).

În partea stânga se afla lista benchmark-urilor disponibile, utilizatorul putând opta pentru selectii multiple. Orice simulare se va efectua, aceasta va opera exclusiv cu trace-urile selectate din aceasta lista. Selectia multipla se realizeaza cu ajutorul mouse-ului si al tastelor Shift / Ctrl, deselectarea fiind procedura inversa selectiei.

Central se afla tabelul cu rezultatele simularii, ce va fi completat progresiv pe parcursul unei simularii; fiecare coloana corespunde unui trace, la sfârșitul simularii ea fiind completata de thread-ul corespunzator simularii trace-ului respectiv.

În sfârșit, în partea de jos a ecranului se afla dispusa paleta de controale utilizator, care contine butoane de pornire/oprire a simulării, un buton pentru selectia graficelor care se doresc a fi generate, butoane pentru salvare/restaurare de configuratii si rezultate. Utilizatorul poate opta pentru salvarea unei configuratii (împreuna cu rezultatele simulării până în acel moment) într-un fisier text pentru o consultare ulterioara. De remarcat faptul ca structura fisierului de tip *csn* este una standard, text, asemanatoare fisierelor *ini*, permitând importul relativ simplu în alte programe de simulare. Încarcarea unui fisier salvat anterior pe disc va avea ca efect modificarea parametrilor de simulare precum si a rezultatelor din tabelul cu rezultate.



**Figura 4.2.** Fereastra principala a simulatorului: configurare si rezultate statistice

**Pentru a efectua o simulare simpla**, fara generare de grafice, utilizatorul va trebui sa efectueze urmatoorii pasi:

1. Se vor stabili parametrii de simulare în partea superioara a ferestrei. La modificarea parametrilor de configurare, tabelul cu rezultate va fi golit



automat, deoarece rezultatele afisarii nu mai corespund noilor valori ale parametrilor de intrare.

2. Se vor selecta trace-urile pentru simulare din lista aflata în stânga ferestrei.
3. Se apasa butonul **Start Simulation** din bara de butoane. Simularea poate fi oprita în orice moment prin apasarea butonului **Stop Simulation**. Dupa oprirea fortata a simularii, în tabelul cu rezultate vor aparea rezultatele parțiale ale simularii până în acel punct.

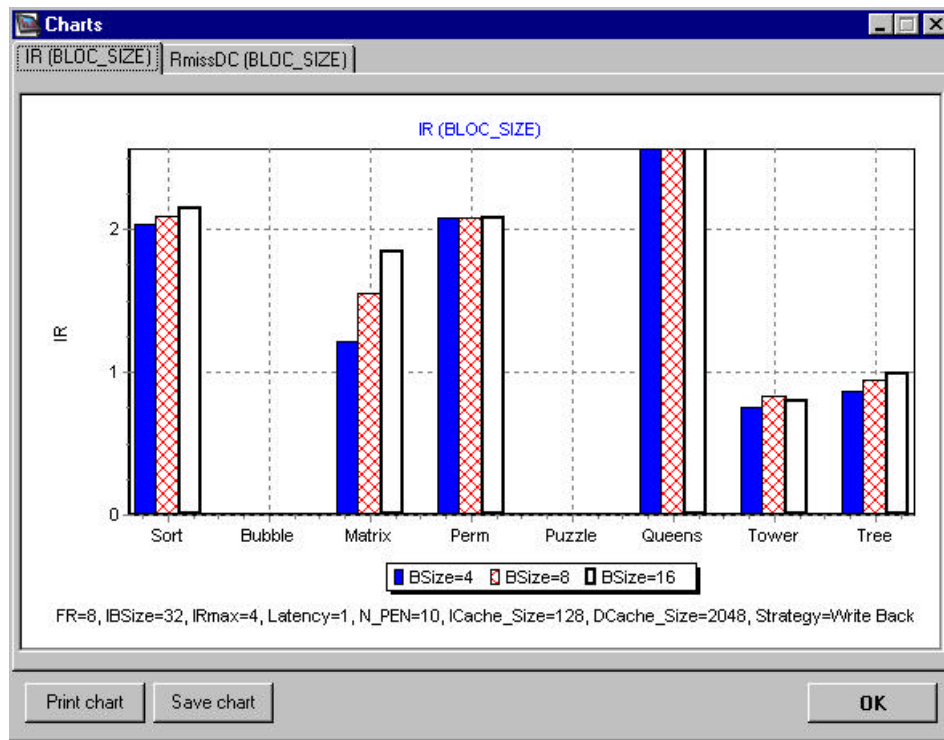
**Pentru a efectua o simulare complexa**, cu generare de grafice, utilizatorul va trebui sa execute urmatoarea secventa:

4. Se vor stabili parametrii de simulare. În functie de graficul solicitat anumiti parametrii se vor modifica automat pe parcursul simularii (*dimensiunea blocului de date* în cazul graficelor IR(BLOC\_SIZE) si RmissDC(BLOC\_SIZE) sau strategia de scriere în cache pentru graficul IR(Strategy)).
5. Se vor selecta trace-urile pentru simulare.
6. Se apasa butonul **Start Simulation** din bara de butoane. Simularea poate fi oprita în orice moment prin apasarea butonului **Stop Simulation**. Dupa oprirea fortata a simularii, în tabelul cu rezultate vor aparea rezultatele parțiale ale simularii până în acel punct.

Operatiunea de simulare fiind mai complexa necesita un timp suplimentar de executie.

Fereastra de afisare a graficelor este simpla (vezi figura 4.3), organizata sub forma unor pagini, fiecare dedicata unui anumit tip de grafic. S-a optat pentru aceasta varianta din cauza extensibilitatii sporite în viitor, prin adaugarea cu usurinta de noi grafice. În aceasta fereastra, utilizatorul poate consulta graficele generate, le poate tipari la imprimanta (apasând butonul **Print**) sau poate salva graficul curent pe disc în format \*.bmp (apasând butonul **Save**).

De retinut ca la apasarea butonului OK (dupa confirmarea actiunii) se vor pierde graficele generate, pentru reafisarea lor procesul de simulare fiind reluat de la început. S-a optat pentru aceasta solutie din cauza consumului de memorie redus si datorita vitezei de simulare relativ ridicata.



**Figura 4.3.** Rezultatele simulării sub forma grafică

#### 4.3.2. PROGRAMELE DE TEST STANFORD

Benchmarking reprezintă procesul de evaluare și comparare a performanțelor diverselor arhitecturi sau sisteme de calcul prin simulare. Metoda folosită este *trace driven simulation*. Programele benchmark de numere întregi Stanford, sunt o suită de opt programe C care necesită puțină inițializare de date și care sunt gândite să manifeste comportamente similare cu scopul general al programelor de calculator, deși unele au o natură destul de recursivă. Programele implică executia a 100 până la 900 de mii de instrucțiuni mașină HSA. Codul original C este întâi trecut printr-un compilator “*gnu C*” care produce formatul corect al codului în mnemonica de asamblare (fișiere \*.ins), precum și directive de asamblare, comenzi de alocare a datelor. Codul este apoi executat pe un simulator la nivel de instrucțiuni, care produce la ieșire un fișier trace de instrucțiuni (fișiere \*.trc). În final aceste trace-uri constituie programe de test în simularea unui sir de arhitecturi de cache, pentru determinarea optimului de performanță în

anumite conditii de intrare. De remarcat ca, la rulara repetata a aceluiași program C aferent oricarui din cele opt benchmark-uri se obtine acelasi fisier trace.

Spre exemplificare prezentam primele doua linii de cod din programul de test **fsort.trc** [Col93].

```
B 2 151      S 152 3968    B 153 120      S 121 3840    B 122 18
S 19 3712    S 20 3720    S 21 3724    B 22 4        S 6 6328
```

Întregul fisier trace este o înlantuire de triplete <**TipInstr** **AdrCrt** **AdrDest**>, unde **TipInstr** poate lua una din valorile ‘B’ – branch, ‘L’ – load, ‘S’ – store; **AdrCrt** reprezinta valoarea registrului PC – adresa instructiunii curente, iar **AdrDest** reprezinta adresa de memorie a datei accesate – în cazul instructiunilor cu referire la memorie (‘L’ / ‘S’) sau adresa destinatie a saltului – în cazul instructiunilor de salt si ramificatie (‘B’). Instructiunile care nu apar în trace sunt instructiuni aritmetico-logice, relationale, de deplasare si rotire; consideram pentru aceste instructiuni **TipInstr** = ‘A’ si **AdrDest** = xxxx - nesemnificativa în acest caz.

Prima instructiune din trace este <B 2 151> semnificând urmatoarele: PC-ul instructiunii de salt este 2, iar adresa urmatoarei instructiuni citite si ulterior executate este 151. Întrucât programul începe cu instructiunea al carei PC=0, si aceasta nu exista în trace, rezulta ca primele doua instructiuni din program sunt aritmetice. Secventa reala de instructiuni executate ar fi:

```
A 0 xxxx      A 1 xxxx      B 2 151.
```

Urmatoarea instructiune este cea de la adresa 151, dar cum ea nu se gaseste în trace, înseamna ca la aceasta adresa exista tot o instructiune aritmetica, iar PC\_next (PC-ul urmatoarei instructiuni) este incrementat, neexistând nici o instructiune de salt care sa schimbe cursul programului. Instructiunea urmatoare având PC=152, este cu referire la memorie “Store la adresa 3968”. Urmeaza o noua instructiune de salt, PC\_next devenind 120. La aceasta adresa întâlnim o noua instructiune aritmetica, urmata de un Store iar apoi un nou salt samd.

Concomitent putem urmari în fisierul **fsort.ins** mnemonica în asamblare a instructiunilor citite si ulterior executate, trace-ul reprezentând cursul exact al programului - instructiune cu instructiune - în conditiile unui branch prediction perfect.

Prezentam desfasurarea - modul de citire si executie - al instructiunilor, în paralel, (trace si asamblare) a primelor doua linii din fisierul trace.

Fsort.trc			fsort.ins
A	0	xxxx	MOV GP, #4096
A	1	xxxx	MOV SP, #4096
B	2	151	BSR RA, _main (#0)
A	151	xxxx	SUB SP, SP, #128
S	152	3968	ST 0(SP), RA
B	153	120	BSR RA, _Quick (#0)
A	120	xxxx	SUB SP, SP, #128
S	121	3840	ST 0(SP), RA
B	122	18	BSR RA, _Initarr (#0)
A	18	xxxx	SUB SP, SP, #128
S	19	3712	ST 0(SP), RA
S	20	3720	ST 8(SP), R17
S	21	3724	ST 12(SP), R18
B	22	4	BSR RA, _Initrand (#0)
A	4	xxxx	SUB SP, SP, #128
A	5	xxxx	MOV R13, #74755
S	6	6328	ST _seed, R13

Tabelul 4.1.

### Executia primelor 17 instructiuni din trace

Cele opt benchmark-uri - *bubble*, *sort*, *perm*, *puzzle*, *queens*, *matrix*, *tree* si *tower* - desi relativ scurte (contin între 100 si 810 de mii de instructiuni dinamice), sunt caracterizate de calcul intensiv si au o dinamica ridicata a numarului de instructiuni. Un numar de benchmark-uri folosesc recursivitatea, *perm*, *tower*, *sort* si *tree* fiind puternic recursive. Dinamica distributiei instructiunilor este tipica: 53% instructiuni aritmetice, logice sau de rotatie si deplasare, 13% relationale, 18% instructiuni Load, 12% Store si 17% instructiuni de salt [VinFlor99, VinFlor00, Col93].

Benchmark-ul *tower* reprezinta programul de rezolvare a problemei turnului din Hanoi pentru sapte discuri. Benchmark-urile *bubble*, *tree* si *sort* reprezinta trei programe bazate pe tehnici de sortare diferite (interschimbare, insertie binara si sortare rapida prin partitionare dupa un pivot). Benchmark-ul *matrix* presupune calcularea produsului a doua matrici. Programul de test *queens* rezolva problema celor opt regine de pe tabla de sah. Benchmark-ul *perm* realizeaza permutari de grupuri de sapte numere (de la 0 la 6) de mai multe ori, iar *puzzle* rezolva probleme de puzzle (solutia finala obtinându-se când numerele ajung în pozitii consecutive în matricea ce reprezinta starile puzzle-lului). Totusi, aria de aplicabilitate a benchmark-urilor Stanford nu se extinde si la aplicatiile grafice si multimedia, rutine critice ale sistemelor de operare etc.

Mai jos, se prezinta o descriere succinta a benchmarkurilor utilizate în cercetare.

Benchmark	Total instr.	Descriere benchmark-uri
Puzzle	804.620	Rezolva o problema de "puzzle"
Bubble	206.035	Sortare tablou prin metoda "bulelor"
Matrix	231.814	Inmultiri de matrici
Permute	355.643	Calcul recursiv de permutari
Queens	206.420	Problema de sah a celor 8 regine
Sort	72.101	Sortare rapida a unui tablou aleator (quick sort)
Towers	251.149	Problema turnurilor din Hanoi (recursiva)
Tree	136.040	Sortare pe arbori binari

Tabelul 4.2.

### Caracteristicile benchmark-urilor Stanford

**Obs:** Din punct de vedere al implementarii software, solutia implementarii multifir prezinta un avantaj net ca timp de executie, în defavoarea solutiei clasice de simulare secventiala (trace-urile simulate unul dupa altul). Experimental, prin implementarea ambelor variante, s-a constatat o îmbunatatire a timpului de simulare de aproximativ 50%. S-a optat pentru o prioritate marita a thread-urilor de simulare, în detrimentul firului principal, datorita vitezei simtitor îmbunatatite, precum si a faptului ca singura optiune a utilizatorului pe firul principal (în timpul simularii) este oprirea proceselor prin actiunea butonului *Stop Simulation*.

## 4.4. PROBLEME PROPUSE SPRE REZOLVARE

Rezultatele generate sunt rata de procesare medie – *average issue rate* – (numar de instructiuni raportat la numar de cicli de executie), rate de miss în cache-uri (IC, DC). Se vor determina parametri optimi si factorii de limitare în fiecare din cazuri.

Cu ajutorul simulatorului *Simcache.exe* generati [VinFlor00]:

1. Rezultate urmate de grafice privind influenta ratei de fetch (FR) asupra ratei de procesare  $IR(FR)$  si asupra ratei de miss în cache-ul de instructiuni  $R_{missIC}(FR)$ .

2. Studiați influența capacității cache-ului de instrucțiuni asupra ratei de procesare  $IR(SIZE\_IC)$  și asupra ratei de miss la cache-ul de instrucțiuni  $R_{missIC}(SIZE\_IC)$ .
3. Studiați influența capacității cache-ului de date asupra ratei de procesare  $IR(SIZE\_DC)$  și asupra ratei de miss la cache-ul de date  $R_{missDC}(SIZE\_DC)$ .
4. Determinați influența numărului maxim de instrucțiuni ce pot fi trimise simultan în execuție asupra ratei de procesare  $IR(IR_{max})$ .
5. Se vor genera graficele  $IR(BLOC\_SIZE)$  și  $R_{missDC}(BLOC\_SIZE)$  în cele două ipostaze: scriere în cache prin *write back* și scriere în cache prin *write through*.
6. Se va studia comparativ realismul, prin rata de procesare, introdus prin cele două tehnici de scriere față de situația când nu se folosește nici una din aceste tehnici IR (tehnica de scriere în cache).

## 5. ÎMBUNĂTĂȚIREA SISTEMULUI IERARHIZAT DE MEMORIE PRIN ARHITECTURI DE TIP “*SELECTIVE VICTIM CACHE*”

---

### 5.1. SCOPUL LUCRĂRII

Lucrarea de față urmărește îmbunătățirea performanțelor memoriilor cache cu mapare directă, integrate într-un procesor superscalar utilizând conceptul de “victim cache” – cache victima (simplu și selectiv). Aplicația realizată implementează o arhitectură de procesor superscalar caracterizată atât prin spații separate pentru instrucțiuni și date cât și prin busuri separate între procesor și memoria cache de date respectiv de instrucțiuni (Harvard).

### 5.2. MEMENTO TEORETIC

În momentul de față se disting mai multe abordări moderne de exploatare și creștere a paralelismului la nivelul instrucțiunii:

- ⇒ Procesoarele superscalare extrag paralelismul dinamic prin execuție speculativă sau trimitere spre procesare “**out of order**”.
- ⇒ Schemurile de instrucțiuni exploatează paralelismul în momentul compilării prin rearanjarea codului sursă, dezambiguizarea referințelor la memorie (antialias), metode de *in lining* aplicate procedurilor, tehnici de optimizare locală și/sau globală (loop unrolling, list/trace scheduling, software pipelining) etc.
- ⇒ Prin comprimarea dependențelor reale de date între instrucțiuni se urmărește eliminarea / reducerea parțială a efectelor defavorabile cauzate de hazardurile RAW (*read after write*), folosind dacă este posibil unități aritmetico-logice cu mai mult de două intrări.

Tehnicile hardware recente de **reutilizare dinamica a instructiunilor** si **predictia valorilor** sunt reprezentative în acest sens [Vin02].

- ⇒ Eforturile de îmbunatatire tehnologica si arhitecturala sunt canalizate spre reducerea decalajului tehnologic dintre un procesor avansat si sistemul ierarhic de memorie (selective victim cache).
- ⇒ Dezvoltarea de noi instrumente de cercetare care apartin si altor domenii (inteligenta artificiala, algoritmi genetici, retele neurale) pentru cresterea acuratetii de predictie a ramificatiilor de program - predictoare neuronale, genetice.

Ultima decada este caracterizata de o accentuare a decalajului dintre viteza procesorului si viteza nivelului inferior de memorie (DRAM, disc). Acesta este rezultatul contributiei a doi factori: (i) perioada de tact a procesorului a scazut mult mai rapid decât timpul de acces la memorie (DRAM) si (ii) tehnicile arhitecturale de proiectare pentru procesoarele pipeline si superscalare au cauzat o reducere dramatica a numarului mediu de cicli per instructiune (CPI).

Modul general de a depasi “gâtuirile” datorate latentei mari a memoriei consta în organizarea memoriei sistem într-o ierarhie, cu un cache foarte rapid de tip SRAM (timp acces de 0.5-3 ns la ora actuala, daca e integrata în procesor) apropiat de CPU si o memorie principala de tip DRAM, aflata pe un nivel inferior. Ierarhiile de memorie exploateaza localizarea intrinseca prezenta în programe tipice prin pastrarea celor mai frecvent accesate informatii în cache, reducându-se numarul de accese în memoria principala. Datorita cresterii decalajului dintre viteza procesorului si viteza nivelului inferior de memorie, influenta performantei ierarhiei de memorie asupra memoriei cache în particular, este considerabil mai puternica decât a fost la începutul ultimei decade [Vin00]. Aceasta tendinta a continuat si cu generatia de procesoare DEC Alpha, PowerPC, Intel Pentium. Realizarea potentialului de performanta al acestor procesoare necesita o proiectare atenta, în special a primului nivel de cache.

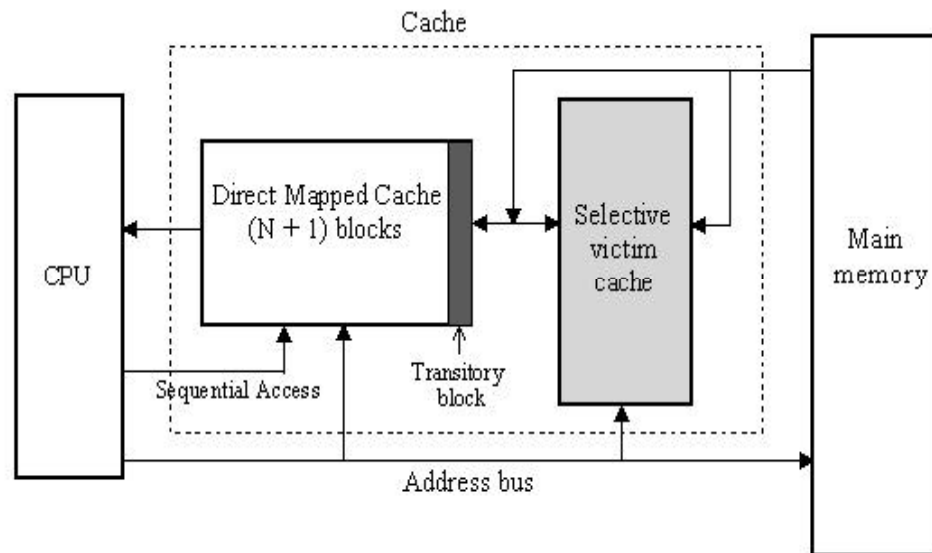
Cache-ul procesorului continua sa fie în topul cercetarii actuale deoarece importanta sa asupra performantei sistemului creste iar îmbunatatirile tehnologice asupra procesorului altereaza caracteristicile sale. În aplicatia de fata primul nivel al ierarhiei de memorie îl constituie cache-ul direct mapat. Desi cache-urile mapate direct au o rata de miss relativ ridicata - datorita conflictelor între referintele la memorie, mai ales daca cache-ul este de dimensiuni mici - comparativ cu cele (semi)asociative, ele sunt atractive pentru procesoarele zilelor noastre prin: costul redus de implementare, timpul de hit mai rapid, timp de acces superior si posibilitate



de integrare on-chip [Flor98]. Cache-urile semiasociative îmbunătățesc rata de miss prin eliminarea interferențelor dar cu pretul creșterii timpului de acces. Cercetătorul american Hill argumentează ca, atunci când se ține cont atât de rata de miss cât și de timpul de acces la cache, cache-urile mapate direct oferă deseori o mai bună performanță - în termenii timpului mediu de acces la memorie - superioară cache-urilor semiasociative. În plus, cum perioada de tact a procesorului se micșorează, un acces într-un singur ciclu la cache poate fi făcut doar într-o configurație cu cache mapat direct. Un alt avantaj al cache-urilor mapate direct îl reprezintă compatibilitatea lor cu procesoarele pipeline. La citirea unei date sau fetch-ul (extragerea din ICache) unei instrucțiuni, data poate fi extrasă pentru CPU înainte de a fi cunoscut rezultatul verificării tag-ului sau în paralel cu aceasta. Dacă rezultatul verificării este miss, procesarea instrucțiunii se oprește la un stadiu ulterior în pipeline. Ideea a fost aplicată la scrierea în cache-ul de date [Joup90]. Pe de altă parte, cache-ul semiasociativ necesită, normal, ca și compararea tag-ului să fie încheiată înainte ca locația datei referite în cache să fie determinată.

Pentru a reduce rata de miss a cache-urilor mapate direct (fără să se afecțeze însă timpul de hit sau penalitatea în caz de miss), cercetătorul Norman Jouppi (DEC) a propus conceptul de **victim cache**. Aceasta reprezintă o memorie mică complet asociativă, plasată între primul nivel de cache mapat direct și memoria principală. Blocurile înlocuite din cache-ul principal datorită unui miss sunt temporar memorate în victim cache. Dacă sunt referite din nou înainte de a fi înlocuite din victim cache, ele pot fi extrase direct din victim cache cu o penalitate mai mică decât cea a memoriei principale [Joup90]. Deoarece victim cache-ul este complet asociativ, multe blocuri care ar genera conflict în cache-ul principal mapat direct, ar putea rezida în victim cache fără să dea naștere la conflicte. Decizia de a plasa un bloc în cache-ul principal sau în victim cache (în caz de miss) este făcută cu ajutorul unei informații de stare asociate blocurilor din cache. Bitii de stare conțin informații despre istoria blocului. Aceasta idee a fost propusă de McFarling, care folosește informația de stare pentru a exclude blocurile sigure din cache-ul mapat direct, reducând înlocuirile ciclice implicate de același bloc. Aceasta schema, numită **excludere dinamică**, reduce miss-urile de conflict în multe cazuri. O predicție greșită implică un acces în nivelul următor al ierarhiei de memorie contrabalansând eventuale câștiguri în performanță. Schema este mai puțin eficientă cu blocuri mari, de capacități tipice cache-urilor microprocesoarelor curente.

Pentru a reduce numărul de interschimbări dintre cache-ul principal și victim cache, D. Stiliadis și A. Varma au introdus un nou concept numit **selective victim cache(SVC)** [Sti94].



**Figure 5.1.** Ierarhia de memorie pentru schema cu *Selective Victim Cache*

Arhitectura SVC încearcă să izoleze două blocuri conflictuale prin memorarea unuia în MCP și a celuilalt în SVC. Cu SVC, blocurile aduse din memoria principală sunt plasate selectiv fie în cache-ul principal cu mapare directă fie în selective victim cache, folosind un algoritm de predicție euristic bazat pe istoria folosirii sale. Blocurile care sunt mai puțin probabil să fie accesate în viitor sunt plasate în SVC și nu în cache-ul principal. Predicția este de asemenea folosită în cazul unui miss în cache-ul principal pentru a determina dacă este necesară o schimbare a blocurilor conflictuale. Obiectivul algoritmului este de a plasa blocurile, care sunt mai probabil să fie referite din nou, în cache-ul principal și celelalte în victim cache. Pe baza algoritmului de predicție a blocului care va fi mai utilizat în viitor, se va reduce numărul de interschimbări de blocuri între MCP și SVC, comparativ cu arhitectura "cache victimă" originală, propusă de *N. Jouppi*, cu consecințe favorabile asupra timpului global de acces la MCP respectiv SVC.

La referirea unui cache mapat direct, victim cache-ul este adresat în paralel; dacă rezultă miss în cache-ul principal, dar hit în victim cache, instrucțiunea (în cazul ICache-ului) este extrasă din victim cache. Penalitatea pentru miss în cache-ul principal, în acest caz este mult mai redusă decât costul unui acces în nivelul următor de memorie. Algoritmul de victim cache încearcă să izoleze blocurile conflictuale și să le memoreze doar unul în cache-ul principal restul în victim cache. Dacă numărul blocurilor conflictuale este suficient de mic să se potrivească în victim cache, atât rata de miss în nivelul următor de memorie cât și timpul mediu

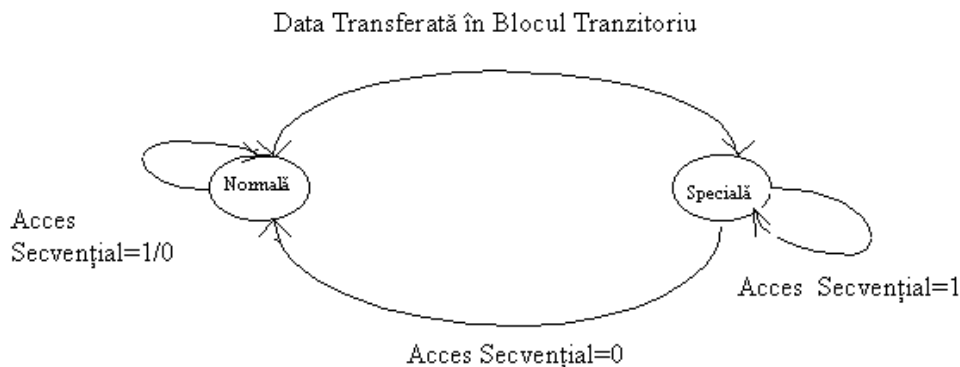
de acces va fi îmbunătățit datorita penalității reduse implicate de prezenta blocurilor în victim cache.

Ca și cache-ul victima simplu (VC), SVC este o memorie de capacitate redusă (5-8 locații !), cu un grad sporit de asociativitate, interpusă între cache-ul principal cu mapare directă și nivelul superior de memorie. Memoria cache principală (MCP) și SVC-ul sunt adresate în paralel la orice acces la memorie al procesorului. Latenta SVC este considerată - în baza articolului original [Sti94]- mai mare decât cea a MCP dar mai mică decât latenta nivelului superior de memorie. Acest lucru are sens în opinia noastră, numai dacă se consideră o MCP adresată cu adrese virtuale (cache virtual, avantajos dar dificil de implementat) și o SVC adresată cu adrese fizice, fapt necesar având în vedere că în general această memorie este complet asociativă. În acest sens se precizează că o memorie asociativă este mai dificil să fie adresată cu adrese virtuale decât una cu mapare directă, întrucât problema aliasurilor de memorie (două adrese virtuale distincte mapate pe aceeași adresă fizică) este mai dificil de rezolvat în acest caz. Oricum, în această abordare s-ar putea într-adevăr considera că latenta SVC este mai mare decât cea a MCP, având în vedere procesul adițional de traducere a adresei virtuale în adresă fizică prin accesarea unor tabele de traducere din memoria principală sau a unor TLB-uri (*Translation Lookaside Buffers*) rezidente "on-chip". Articolul lui *Stiliadis* și *Varma* nu realizează aceste aspecte relativ subtile, esențiale pentru timing-ul structurii propuse. Memoria SVC se asociază cu memorii cache principale mapate direct, în vederea îmbunătățirii performanțelor acestora din urmă. Reamintim că aceste memorii cache cu mapare directă sunt cele mai simple și mai ușor de integrat în circuit, având timpul de acces cel mai redus dar și o rată de miss mai ridicată decât la tipurile asociative, datorită în principal interferențelor blocurilor conflictuale. Ele se pretează cel mai bine la procesarea pipeline care impune (la nivelul anului 2002) timpi de acces mai mici de 1 ns (Pentium IV, Itanium). În plus, cu astfel de memorii cache, data (instrucțiunea) poate fi adusă de către procesor înainte ca verificarea tag-urilor să se fi încheiat, ceea ce la cele asociative este imposibil întrucât procesul de căutare al datei în cache se face chiar după tag.

Cache-ul mapat direct crește cu un bloc pentru a implementa conceptul de selective victim cache. Acest bloc adițional se numește *bloc tranzitoriu*, și este necesar pentru două motive. Primul ar fi acela că, blocul tranzitoriu este folosit de algoritmul de predicție pentru referiri secvențiale într-un același bloc. Hardware-ul este capabil să determine accese secvențiale, folosind semnalul "Acces Secvențial" activat de CPU, când referirea curentă se face în același bloc ca și cel anterior. Semnalul este folosit de către cache pentru a evita actualizarea bitilor de stare folosiți de

algoritmul de predicție la referințe repetate în același bloc tranzitoriu. Al doilea motiv constă în faptul că, atunci când are loc un hit în victim cache și algoritmul de predicție decide să nu se interschimbe blocurile, blocul corespondent este copiat din victim cache în blocul tranzitoriu. În acest caz blocul tranzitoriu servește ca buffer astfel încât accese secvențiale la acel bloc pot fi satisfăcute direct din acest buffer la timpul de acces al cache-ului principal. Similar, atunci când are loc un miss în următorul nivel de memorie, algoritmul de predicție ar trebui să decidă să plaseze blocul sosit în victim cache și în blocul tranzitoriu.

Întrucât un al doilea sau un al n-lea acces consecutiv în același bloc în cache-ul principal poate fi servit din blocul tranzitoriu, acestuia îi este adăugat un bit de stare pentru a adresa cache-ul principal. Acest bit de stare urmărește starea datei din blocul tranzitoriu. Când starea este *normală*, adresa sosită pe bus este decodificată pentru a accesa cache-ul principal în mod obișnuit; când starea este *specială*, accesul se face în blocul tranzitoriu. Figura următoare arată tranzițiile dintre cele două stări. Mai jos se prezintă acest algoritm sub forma de “mașină secvențială de stare” (*Sequential State Machine – SSM*).



**Figura 5.2** Mașina secvențială de stare și tranzițiile ei

Inițial starea mașinii este resetată în stare *normală*. Dacă avem un miss în cache-ul mapat direct, acesta este servit fie de victim cache fie de nivelul următor de memorie. În fiecare din cazuri, algoritmul de predicție este folosit pentru a determina care bloc urmează a fi memorat în cache-ul principal. Dacă algoritmul de predicție plasează blocul accesat în cache-ul principal, starea mașinii rămâne în stare *normală*. Altfel, blocul este copiat în blocul tranzitoriu din cache și mașina tranzitează în starea *specială*. Referința secvențială a aceluiași bloc păstrează semnalul “Acces Secvențial” activat iar mașina în starea *specială*. Datele se extrag din blocul tranzitoriu.

Primul acces nesecvențial resetează starea mașinii în stare *normală*, distingându-se trei cazuri distincte, pe care le vom discuta mai jos.

### 5.2.1. ALGORITMUL DE SELECTIVE VICTIM CACHE

1. *Hit în cache-ul principal*: dacă cuvântul este găsit în cache-ul principal, el este extras pentru CPU. Nu este nici o diferență față de cazul cache-ului mapat direct. Singura operație suplimentară este o posibilă actualizare a bitilor de stare folosiți de schema de predicție. Actualizarea se poate face în paralel cu operația de fetch și nu introduce întârzieri suplimentare.
2. *Miss în cache-ul principal, hit în victim cache*: în acest caz, cuvântul este extras din victim cache în cache-ul mapat direct și înaintat CPU. Un algoritm de predicție este invocat pentru a determina dacă va avea loc o interschimbare între blocul referit și blocul conflictual din cache-ul principal. Dacă algoritmul decide că blocul din victim cache este mai probabil să fie referit din nou decât blocul conflictual din cache-ul principal se realizează interschimbarea; altfel blocul din victim cache este copiat în blocul tranzitoriu al cache-ului principal iar mașina secvențială de stare trece în starea specială. Datele pot fi înaintate CPU. În ambele cazuri blocul din victim cache este marcat drept cel mai recent folosit din lista LRU. În plus, bitii de predicție sunt actualizați pentru a reflecta istoria acceselor.
3. *Miss atât în cache-ul principal cât și în victim cache*: dacă cuvântul nu este găsit nici în cache-ul principal nici în victim cache, el trebuie extras din nivelul următor al ierarhiei de memorie. Aceasta înseamnă că fie blocul corespondent din cache-ul principal este “gol”, fie noul bloc este în conflict cu un alt bloc memorat în cache (mai probabil). În primul caz, noul bloc este adus în cache-ul principal iar victim cache-ul nu este afectat. În cel de-al doilea caz, trebuie aplicat algoritmul de predicție pentru a determina care din blocuri este mai probabil să fie referit pe viitor. Dacă blocul care sosește din memoria centrală are o probabilitate mai mare decât blocul conflictual din cache-ul principal, ultimul este mutat în victim cache și noul bloc îi ia locul în cache; altfel, blocul sosit este direcționat spre victim cache și copiat în blocul tranzitoriu al cache-ului mapat direct, de unde poate fi accesat mai ușor de către CPU. Mașina secvențială de stare trece în starea specială iar bitii de predicție sunt actualizați.

Diferenta introdusa prin schema prezentata (*Selective Victim Cache*) se observa în cazurile 2 si 3. În cazul 2, blocurile conflictuale din cache-ul principal si cele din victim cache sunt întotdeauna schimbate în cazul folosirii victim cache-ului traditional, pe când schema prezentata face acest lucru într-un mod selectiv, euristic. Similar, în cazul 3, prin folosirea victim cache-ului obisnuit blocurile din memorie sunt întotdeauna plasate în cache-ul principal, pe când în cazul *Selective Victim Cache*-ului plaseaza aceste blocuri selectiv în cache-ul principal sau în victim cache.

Orice algoritm de înlocuire poate fi folosit pentru victim cache. LRU (cel mai putin recent referit) pare sa fie cea mai buna alegere, întrucât scopul victim cache-ului este de a captura cele mai multe victime recent înlocuite si victim cache-ul este de dimensiune mica.

### 5.2.2. ALGORITMUL DE PREDICTIE

Scopul algoritmului de predictie este de determina care din cele doua blocuri conflictuale este mai probabil sa fie referit pe viitor. Blocul considerat cu o probabilitate mai mare de acces în viitor este plasat în cache-ul principal, celalalt fiind plasat în victim cache. Astfel, daca blocul din victim cache este pe viitor înlocuit datorita capacitatii reduse a victim cache-ului, impactul ar fi mai putin sever decât alegerea opusa (interschimbarea permanenta a blocurilor din cazul schemei cu victim cache obisnuit).

Algoritmul de predictie se bazeaza pe algoritmul de excludere dinamica propus de McFarling [McFar92]. Algoritmul foloseste doi biti de stare asociati fiecarui bloc, numiti *hit bit* si *sticky bit*. Hit bit este asociat logic cu blocul din nivelul L1 al cache-ului care se afla pe nivelul L2 sau în memoria centrala. Hit bit egal cu 1 indica, faptul ca a avut cel putin un acces cu hit la blocul respectiv de când el a parasit cache-ul principal (cache-ul de pe nivelul L1). Hit bit egal cu 0 înseamna ca blocul corespunzator nu a fost deloc accesat de când a fost înlocuit din cache-ul principal. Într-o implementare ideala, bitii de hit sunt mentinuti în nivelul L2 de cache sau în memoria principala si adusi în nivelul L1 de cache cu blocul corespondent. Daca blocul este înlocuit din cache-ul principal (L1 cache), starea bitului de hit trebuie actualizata în L2 cache sau în memoria centrala. Când un bloc, sa-l numim  $\alpha$ , a fost adus în cache-ul principal, bitul sau sticky este setat. Fiecare secventa cu hit la blocul  $\alpha$  reîmprospateaza bitul sticky la valoarea 1. La referirea unui bloc conflictual, fie acesta  $\beta$ , daca algoritmul de predictie decide ca blocul sa nu fie înlocuit din cache-ul principal atunci bitul sticky este resetat. Daca un acces ulterior în cache-ul principal intra în

conflict cu blocul care are bitul *sticky* resetat, atunci blocul va fi înlocuit din cache-ul principal. De aceea, *sticky* bit de valoare 1 pentru blocul  $\alpha$  semnifică faptul că nu a avut loc nici o referire la un bloc conflictual cu  $\alpha$ , de la ultima referire a acestuia.

Este ușor de înțeles rolul blocului tranzitoriu în algoritmul de predicție. Dacă algoritmul tratează toate fetch-urile în același fel, accesările secvențiale în același bloc vor seta întotdeauna bitul *sticky*. Algoritmul de predicție va fi incapabil să determine dacă blocul a fost referit repetat în interiorul unei bucle, sau dacă mai mult decât un cuvânt din același bloc a fost extras din cache fără o referință intervenită la un alt bloc. O soluție similară a acestei probleme a fost propusă și în [McFar92].

În algoritmul *Selective Victim Cache* prezentat în figura 5.3, se disting trei cazuri: în primul caz, un hit în cache-ul principal setează bitii de stare hit și *sticky*. În al doilea caz, blocul accesat, fie acesta  $\beta$ , se consideră rezident în victim cache. Acesta implică un conflict între blocul  $\beta$  și cel din cache-ul principal, notat  $\alpha$ . În acest caz, algoritmul de predicție este aplicat pentru a determina dacă va avea loc o interschimbare. Dacă bitul *sticky* al lui  $\alpha$  este 0, semnificând faptul că blocul nu a fost accesat de la conflictul anterior la acest bloc, noul bloc  $\beta$  primește o prioritate superioară lui  $\alpha$ , determinând o interschimbare. De asemenea, dacă bitul hit al lui  $\beta$  este setat pe 1, acestuia îi este dată o prioritate mai mare decât lui  $\alpha$ , și ele sunt interschimbate. Dacă bitul *sticky* al lui  $\alpha$  este 1 și bitul hit al lui  $\beta$  este 0, accesul este satisfăcut din victim cache și nu are loc nici o interschimbare (se consideră că blocul  $\beta$  nu este suficient de „valoros” pt. a fi adus în cache-ul principal). Bitul *sticky* aferent lui  $\alpha$  este resetat astfel încât o secvență următoare care implică conflict la acest bloc va determina mutarea lui  $\alpha$  din cache-ul principal. În final, cazul 3 al algoritmului prezintă secvența de acțiuni care au loc în cazul unor accese cu miss atât în cache-ul principal cât și în victim cache. Secvența este similară cu cea de la cazul 2, cu excepția faptului că, destinația blocului sosit se alege fie cache-ul principal fie victim cache-ul. În situația cu victim cache simplu, blocul conflictual din cache-ul principal era mutat în victim cache înainte să fie înlocuit. În cazul de față când blocul sosit este plasat în victim cache, el este de asemenea plasat și în blocul tranzitoriu pentru a servi eventualele viitoare referințe secvențiale.

```

Cazul 1: Accesarea blocului  $\beta$ , hit în cache-ul principal
  Actualizează biții hit și sticky
  hit[ $\beta$ ] ← 1; sticky[ $\beta$ ] ← 1;

Cazul 2: Accesarea blocului  $\beta$ , hit în victim cache
  Fie  $\alpha$  blocul conflictual din cache-ul principal
  if sticky[ $\alpha$ ] = 0 then
    interschimbă  $\alpha$  cu  $\beta$ 
    sticky[ $\beta$ ] ← 1; hit[ $\beta$ ] ← 1 ( hit[ $\beta$ ] ← 0;
                                modificarea autorilor )
  else
    if hit[ $\beta$ ] = 0 then
      sticky[ $\alpha$ ] ← 0;
      copiază  $\beta$  în blocul tranzitoriu
    else
      interschimbă  $\alpha$  cu  $\beta$ 
      sticky[ $\beta$ ] ← 1; hit[ $\beta$ ] ← 0;
    endif
  endif

Cazul 3: Accesarea blocului  $\beta$ , miss atât în cache-ul principal cât și în victim cache
   $\alpha$  este blocul conflictual din cache-ul principal
  if sticky[ $\alpha$ ] = 0 then
    mută  $\alpha$  în victim cache
    transferă  $\beta$  în cache-ul principal
    sticky[ $\beta$ ] ← 1; hit[ $\beta$ ] ← 1; ( hit[ $\beta$ ] ← 0;
                                modificarea autorilor )
  else
    if hit[ $\beta$ ] = 0 then
      transferă  $\beta$  în victim cache
      copiază  $\beta$  în blocul tranzitoriu
      sticky[ $\alpha$ ] ← 0;
    else
      mută  $\alpha$  în victim cache
      transferă  $\beta$  în cache-ul principal
      sticky[ $\beta$ ] ← 1; hit[ $\beta$ ] ← 0;
    endif
  endif

```

**Figura 5.3** Algoritmul *Selective Victim Cache*

Operațiile algoritmului *Selective Victim Cache* pot fi ilustrate printr-o secvență de instrucțiuni repetate  $(\alpha^m \beta \gamma)^n$  implicând trei blocuri conflictuale  $\alpha$ ,  $\beta$  și  $\gamma$ . Notatia  $(\alpha^m \beta \gamma)^n$  reprezintă executia unei secvențe compuse din două bucle de program imbricate, bucla interioară constând în  $m$  referințe la



blocul  $\alpha$ , urmate de accesul la blocurile  $\beta$  și  $\gamma$  în bucla exterioară, care se execută de  $n$  ori. Primul acces îl aduce pe  $\alpha$  în cache-ul principal și atât bitul hit cât și cel sticky sunt setați după cel mult două referiri ale acestuia. Când  $\beta$  este referit, bitul sau hit este inițial 0. De aceea el nu-l înlocuiește pe  $\alpha$  în cache-ul principal și este memorat în victim cache. Conflictul generat determină resetarea bitului sticky al lui  $\alpha$ . Când  $\gamma$  este referit, bitul sau hit este 0, dar bitul sticky al lui  $\alpha$  este tot 0. Deci,  $\gamma$  îl înlocuiește pe  $\alpha$ . Blocul  $\alpha$  este transferat în victim cache și bitul sau hit rămâne 1 datorită referinței sale anterioare. În ciclul următor când  $\alpha$  este referit din nou, el este mutat înapoi în cache-ul principal datorită bitului sau de hit, ramas setat. Astfel, dacă victim cache-ul este suficient de mare pentru a încapa atât  $\alpha$  și  $\beta$ , sau  $\beta$  și  $\gamma$ , doar trei referințe ar fi servite de către al doilea nivel de cache. Numărul total de interschimbări nu va depăși  $2n$ . În cazul unei scheme simple de predicție fără victim cache, numărul total de referiri cu miss ar fi  $2n$ , în cazul în care schema poate rezolva doar conflicte între două blocuri. Un victim cache simplu, fără predicție ar fi capabil să reducă numărul de accese cu miss la cel de-al doilea nivel de cache la 3, dar ar necesita  $3n$  interschimbări în timpul execuției buclei exterioare, cu influențe evident defavorabile asupra timpului global de procesare. Aceasta arată avantajul Selective Victim Cache-ului superioară altor scheme care tratează conflicte implicând mai mult de două blocuri. De reținut că, penalitatea pentru o predicție greșită în această schema este limitată la accesul în victim cache și o posibilă interschimbare, presupunând că victim cache-ul este suficient de mare pentru a reține blocurile conflictuale între accese.

### 5.2.3. METRICI DE PERFORMANȚĂ

Metricile de performanță folosite sunt rata de miss la nivelul L1 de cache și timpul mediu de acces la ierarhia de memorie. În cazurile cu victim cache simplu și cel cu victim cache selectiv, folosim de asemenea și numărul de interschimbări între cache-ul principal și victim cache ca metrică de comparație. Rata de miss la nivelul L1 de cache se bazează pe numărul de referințe propagate în nivelul următor al ierarhiei de memorie. În caz de miss în cache-ul principal acestea sunt servite de către victim cache, fiind plată o penalitate pentru accesul în victim cache precum și pentru interschimbările de blocuri rezultate între cache-ul principal și victim cache. Timpul mediu de acces la ierarhia de memorie ia în calcul și aceste

penalizari si de aceea este un bun indicator al performantei memoriei sistemului, desigur mai complet decât rata de miss în nivelul L1 de cache.

Deoarece obiectivul principal al victim cache-ului este de a reduce numarul de *miss-uri de conflict* în cache-ul mapat direct, este de asemenea important sa comparam procentul de miss-uri de conflict eliminate prin fiecare din scheme. Miss-urile de conflict sunt de obicei calculate ca miss-uri suplimentare ale unui cache, comparate cu un cache complet asociativ de aceeasi marime si care dezvolta un acelasi algoritm de înlocuire. Algoritmul folosit este LRU (Least Recently Used, cel mai putin recent folosit) [Hill88, Hen96]. Desi acest model pare intuitiv corect, el poate genera si rezultate eronate uneori. De exemplu, numarul total de accese cu miss poate uneori sa creasca când creste asociativitatea, iar politica de înlocuire LRU este departe de a fi cea optima pentru unele din programe. Aceasta anomalie poate fi evitata prin folosirea algoritmului optim OPT în loc de LRU ca baza pentru clasificarea miss-urilor în cache [Sug93]. Algoritmul OPT, prima data studiat în [Bel66], înlocuieste întotdeauna blocul care va fi înlocuit cel mai târziu în viitor. Un astfel de algoritm s-a dovedit a fi optimal cvasioptimal, dupa cum am aratat în capitolul precedent.

### **Modelarea timpului de acces la ierarhia de memorie**

Estimarea timpului de acces se face ca o functie de marimea cache-ului, dimensiunea blocului, asociativitatea si organizarea fizica a cache-ului. Presupunem ca penalitatea medie pentru un miss în cache-ul de pe nivelul L1 este acelasi pentru toate arhitecturile si este de  $p$  ori ciclul de baza al cache-lui principal, unde  $p \in [1, 100]$ .

Parametrii	Cache Mapat Direct	Victim Cache Simplu	Selective Victim Cache	2-way cache
Referinte totale	$R$			
Numar total de miss-uri în L1 cache	$M_d$	$M_v$	$M_s$	$M_2$
Hit-uri în victim cache		$h_v$	$h_s$	
Interschimbări între victim cache si cache-ul principal		$I_v$	$I_s$	
Timp mediu de acces	$T_d$	$T_v$	$T_s$	$T_2$
Timp mediu de penalizare (în cicli CPU)	$P$			
Perioada de tact CPU	$Clk$			$Clk_{2-way}$

Tabelul 5.1.

### **Notatiile folosite în calculul timpului de acces**

Tabelul 5.1, rezuma toate notațiile privitoare la calculul timpului de acces la memorie.  $R$  este numărul total de referințe generate de programele de tip trace. În cazul cache-ului simplu mapat direct,  $M_d$  reprezintă numărul total de accese cu miss în cache. În cazul folosirii unui victim cache obisnuit sau a unui Selective Victim Cache,  $M_v$  și  $M_s$  sunt folosite pentru a nota numărul de accese cu miss în primul nivel de cache care sunt servite de al doilea nivel al ierarhiei de memorie. Numărul total de hituri în victim cache pentru aceste scheme le-am notat cu  $h_v$  și respectiv  $h_s$ .

Timpul mediu de acces pentru un cache mapat direct se calculează astfel:

$$T_d = clk \times \frac{R + p \times M_d}{R} = clk \left( 1 + p \cdot \frac{M_d}{R} \right) \quad (5.1)$$

Pentru fiecare miss, sunt necesari  $p$  cicli suplimentari. Presupunem ca cei  $p$  cicli includ toate “cheltuielile” CPU-ului. În cazul victim cache-ului simplu, o penalitate de  $p$  cicli este produsă la fiecare miss în primul nivel al ierarhiei de memorie. În plus, pentru fiecare referință servită de către victim cache, o penalizare suplimentară de cel puțin un ciclu este plătită pentru accesarea în victim cache, iar operația de interschimbare dintre cache-ul principal și victim cache necesită o penalitate de câțiva cicli (presupunem 3 cicli, de altfel minimali). Aceasta penalitate ar fi chiar mai mare dacă matricea memoriei cache-ului mapat direct sau a victim cache-ului este organizată fizic în cuvinte egale cu o fracțiune din mărimea blocului de cache. În acest caz penalitatea pentru interschimbare va fi multiplicată cu raportul:

$$\frac{\text{Dimensiunea blocului de cache}}{\text{Marimea cuvântului de date al cache - ului}}$$

Astfel, timpul mediu de acces la memorie pentru un sistem cu victim cache simplu, se calculează astfel:

$$T_v = clk \left( 1 + p \cdot \frac{M_v}{R} + \frac{h_v + 3 \times I_v}{R} \right) \quad (5.2)$$

Într-un sistem cu *Selective Victim Cache*, timpul mediu de acces la memorie poate fi calculat în același fel ca în cazul victim cache-ului simplu. O penalitate de  $p$  cicli este aplicată de câte ori este accesat nivelul următor al ierarhiei de memorie. Un ciclu suplimentar este necesar la un hit în victim

cache si 3 cicli suplimentari pentru operatia de interschimbare de blocuri. Timpul mediu de acces la memorie este dat de formula:

$$T_s = clk \left( 1 + p \cdot \frac{M_s}{R} + \frac{h_s + 3 \times I_s}{R} \right) \quad (5.3)$$

Se observa ca, chiar daca rata de miss  $M_s$  este foarte aproape de cea a victim cache-ului simplu, sistemele ce folosesc *selective victim cache* pot totusi oferi o îmbunatatire substantiala a performantei superioara sistemelor cu victim cache simplu, din urmatoarele doua motive:

1. *Rata de miss locala în cache-ul principal poate fi îmbunatatita printr-un plasament mai bun al blocurilor.*
2. *Numarul de interschimbari poate descreste ca rezultat al algoritmului de predictie.* Aceasta reduce media penalizarii pentru accesele care sunt servite de victime cache, în special când numarul de cicli folositi la o interschimbare este ridicat.

Se foloseste timpul mediu de acces la memorie pentru un sistem cu un cache “2-way asociative”, ca o referinta pentru evaluarea performantei sistemului cu *selective victim cache*. Pentru estimarea timpului de acces la un cache cache “2-way asociative”, se presupune ca penalitatea în nanosecunde pentru al doilea nivel al ierarhiei de memorie ramâne aceeași ca și în cazul cache-ului mapat direct. Pot exista unele constrângeri de implementare care afecteaza penalitatea în caz de miss. Accesarea magistralei sistem, poate implica o secventa de operatii care necesita un numar fix de perioade de tact. Astfel, numarul de cicli necesari pentru deservirea unui miss nu poate descreste proportional cu cresterea perioadei de tact CPU, rezultând într-o penalizare mai mare în cazul cache-ului cache “2-way asociative”. Timpul mediu de acces la memorie acestui cache este estimat de relatia:

$$T_2 = clk \left( \frac{clk_{2-way}}{clk} + p \cdot \frac{M_2}{R} \right) \quad (5.4)$$

Primul termen reprezinta timpul de acces la cache iar al doilea termen este timpul de acces la nivelul urmator de memorie. Comparând aceasta ecuatie cu (5.1), orice îmbunatatire a performantei se datoreaza celui de-al doilea termen, în timp ce primul termen reprezinta câstigul introdus prin asociativitatea cache-ului asupra timpului de acces. Daca îmbunatatirea datorata celui de-al doilea termen nu este adecvata pentru a compensa acest

câștig, performanța cache-ului 2-asociativ poate fi inferioară celei a cache-ului mapat direct.

Îmbunătățirea în performanță obținută atât prin *victim cache* cât și prin *selective victim cache* variază în funcție de trasee, depinzând de mărimea lor și de numărul de conflicte de acces pe care schema de predicție le elimină.

O problemă potențială cu algoritmul de predicție dezvoltat în *selective victim cache* este aceea că, performanța sa se poate degrada odată cu creșterea dimensiunii blocului, ca rezultat al partajării bitilor de stare de cuvinte din interiorul aceluiași bloc. *Selective victim cache* asigură o îmbunătățire semnificativă a ratei de miss indiferent de dimensiunea blocului. Aceste rezultate contrazic comportamentul excluziunii dinamice [McFar92], unde reducerea ratei de miss scade cu creșterea dimensiunii blocului. Rezultatele se datorează menținerii la aceeași dimensiune a *victim cache*-ului în termenii numărului de blocuri, astfel că, o creștere a dimensiunii blocului determină o creștere efectivă a capacității cache-ului. Această creștere în capacitate compensează mai mult decât orice degradare a ratei de predicție prin creșterea dimensiunii blocului. Acest fapt nu crește semnificativ complexitatea implementării *victim cache*-ului, deoarece asociativitatea rămâne aceeași. Indiferent de mărimea cache-ului, numărul de interschimbări prin folosirea *selective victim cache*-ului este redus cu 50% sau mai mult față de folosirea unui *victim cache* simplu [Sti94]. Când dimensiunea blocului este mai mare, în funcție de implementare, operația de interschimbare poate necesita câțiva cicli. Prin îmbunătățirea atât a ratei de hit cât și a numărului de interschimbări, *selective victim cache*-ul poate crește semnificativ performanța primului nivel de cache, superioară *victim cache*-ului simplu și cache-ului “two-way set associative”. Pentru diverse dimensiuni de cache, îmbunătățirea ratei de miss la cache-ul two-way semiasociativ nu este suficientă pentru a compensa creșterea timpului de acces, rezultând într-o creștere netă a timpului mediu de acces la memorie superior cache-urilor mapate direct.

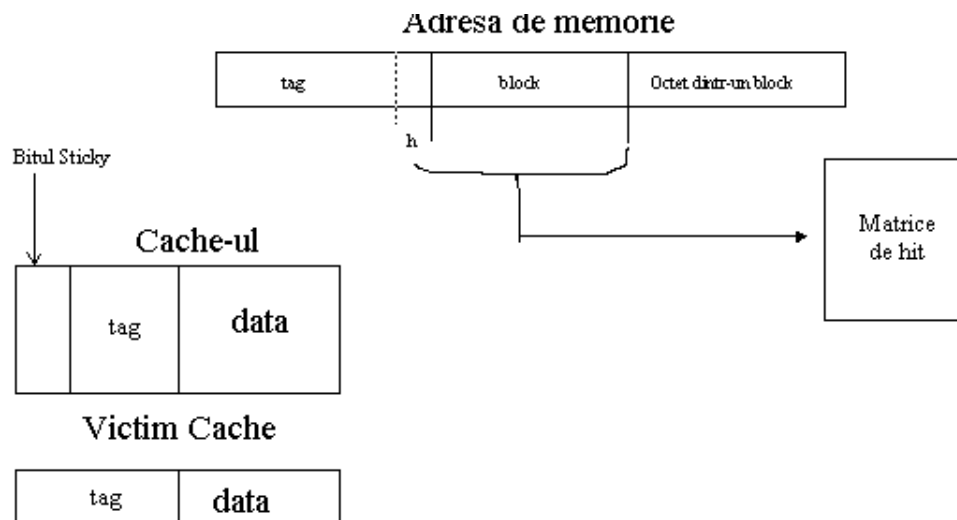
Politica de scriere implementată este *write back* cu *write allocate*. Pentru a menține proprietatea de incluziune multinivel, blocurile din cache-ul de pe nivelul L1 au fost invalidate când au fost înlocuite pe nivelul L2 de cache. Deși *selective victim cache*-ul produce îmbunătățiri semnificative ale ratei de hit comparativ cu cache-urile mapate direct de dimensiune redusă, performanța sa este inferioară celei obținută folosind *victim cache* simplu. De fapt, îmbunătățirile ratei de miss variază semnificativ în funcție de caracteristicile traseelor simulate. Sunt două motive care explică acest raționament: primul este natura acceselor la memorie a programelor folosite. Programele care implică o alocare statică a datelor și structurilor de date, arată o îmbunătățire cu *selective victim cache*, ca rezultat al folosirii

algoritmului de predictie. Structurile de date principale ale acestor programe sunt vectori. Algoritmul de predictie este capabil sa rezolve un numar mare de conflicte în aceste cazuri, fara sa acceseze al doilea nivel. În programele cu alocare dinamica a memoriei si folosire a extensiei de pointeri, conflictele sunt mai greu de rezolvat de catre algoritmul de predictie. Ratele de miss în situatia folosirii selective victim cache-ului pentru aceste trace-uri sunt mai mari decât în cazul folosirii unui simplu victim cache. În timp ce pentru selective victim cache presupunem un al doilea nivel de cache si mentinem proprietatea de incluziune, simularea victim cache-ului simplu presupune ca al doilea nivel al ierarhiei este memoria principala.

Chiar daca îmbunatatirile asupra ratei de miss sunt mai putin convingatoare în cazul cache-urilor de date comparativ cu cel de instructiuni, selective victim cache-ul poate reduce timpul mediu de acces la memorie pentru primul nivel al cache-ului de date prin **reducerea numarului de interschimbari**. Performanta cache-ului semiasociativ este inferioara ambelor (simplu victim cache si selective victim cache), dar superioara cache-ului mapat direct.

Trace-urile de date sunt caracterizate de o rata de miss mai mare decât trace-urile de instructiuni. În plus miss-urile de conflict sunt raspunzatoare de procentul ridicat din rata totala de miss. Sunt doua consecinte ale acestui fapt: primul, efectul reducerii ratei de miss asupra timpului de acces la memorie este mai pronuntat, iar al doilea, cache-urile semiasociative pe 2 cai asigura îmbunatatiri în timpul mediu de acces chiar si pentru cache-uri mari, spre deosebire de cache-urile de instructiuni, unde avantajul obtinut prin reducerea ratei de miss datorata cresterii asociativitatii este mai mare decât câstigul obtinut asupra timpului de acces la cache. Concluzionam ca, atât victim cache-ul simplu cât si *selective victim cache*-ul sunt mult mai putin atractive pentru folosire în cache-ul de date comparativ cu cel de instructiuni.

În continuare se analizeaza modul în care informatiile de stare de care are nevoie schema de predictie dezvoltata în *selective victim cache* pot fi stocate în interiorul ierarhiei de memorie. Dupa cum s-a aratat schema *selective victim cache*-ului necesita doi biti de stare pentru a pastra informatii despre istoria blocurilor din cache - bitul *sticky* (SB) si bitul *hit* (HB). Bitul *sticky* este asociat logic cu blocul din cache-ul principal. De aceea este normal sa se memoreze acest bit în cache-ul mapat direct ca parte a fiecarui bloc. Pe de alta parte, bitul hit este asociat logic cu fiecare bloc din memoria principala. Astfel, într-o implementare perfecta, bitii de hit trebuie memorati în memoria principala. Aceasta abordare este impracticabila în majoritatea cazurilor.



**Figura 5.4.** Implementarea schemei de memorare a bitilor de hit

Dacă ierarhia de memorie include un al doilea nivel de cache, este posibil să se memoreze bitii de hit în cadrul blocurilor din acest nivel. Când un bloc este adus pe nivelul L1 de cache din nivelul L2, o copie locală a bitului de hit este memorată în blocul de pe nivelul L1. Aceasta elimină nevoia de acces a nivelului L2 de cache de fiecare dată când bitul hit este actualizat de către algoritmul de predicție. Când blocul este înlocuit din nivelul L1 de cache, bitul hit corespundent este copiat în nivelul L2. O problemă ar fi însă aceea că, multiple locații din memoria principală sunt forțate să împartă același bit de pe nivelul L2. Astfel, când un bloc este înlocuit de pe nivelul L2 de cache, toate informațiile lui de stare se pierd, reducând eficacitatea algoritmului de predicție. De fiecare dată când un bloc este adus pe nivelul L2 de cache din memoria principală, bitul hit al său trebuie setat la o valoare inițială. Pentru o secvență specifică de acces, valori inițiale diferite pot produce rezultate diferite. Cu cache-urile de pe nivelul L2 de dimensiuni mari, efectul valorilor inițiale este probabil mai mic.

O tratare alternativă este de a menține bitii de hit în interiorul CPU, în cadrul nivelului L1 de cache. În [Sti94] se propun anumite implementări "aproximative", mai realiste, în sensul reducerii necesităților de memorare pentru bitii de hit. Astfel de exemplu, se propune implementarea unui "*hit array*", ca parte a MCP și care permite printr-un mecanism tip "mapare directă", mai multor biti HB corespunzători blocurilor memoriei principale să fie mapati în aceeași locație a acestui "*hit array*". Lungimea sirului este inevitabil mai mică decât numărul maxim de blocuri care pot fi adresate. Deci, mai mult de un bloc va fi mapat aceluiași bit de hit, cauzând datorită interferențelor un aleatorism ce trebuie introdus în procesul de predicție. Cu

alte cuvinte se realizeaza o mapare a bitilor de hit teoretic rezidenti în memoria principala, într-un cache dedicat, cu rezultate obtinute prin simulari, foarte apropiate de cele generate printr-o implementare ideala. Utilizarea unei tabele de dispersie cu rezolvarea coliziunilor prin înlantuire sau adresare deschisa cu dispersie dubla conduce la obtinerea unei performante similare cu situatia ideala (câte un bit de hit pentru fiecare bloc din memoria principala).

Implementarea nivelului L1 de cache sistem este prezentata în Figura 4. Bitul *sticky* este mentinut cu fiecare bloc în cache-ul principal. Nici un bit de stare nu este necesar în *victim cache*. Bitii de hit sunt pastrati în *hit array*, adresati de o parte a adresei de memorie. Dimensiunea sirului de hit bit este aleasa ca un multiplu al numarului de blocuri din cache-ul principal. Astfel, avem relatia:

$$\text{Dimensiunea sirului hit array} = \text{Numar de blocuri în cache-ul principal} \times H$$

unde H determina gradul de partajare a bitilor de hit de catre blocurile memoriei principale. Se presupune ca H este o putere a lui 2,  $H=2^h$ . *Hit array* poate fi adresat de adresa de bloc concatenata cu cei mai putin semnificativi biti  $h$ , din partea de tag a adresei. O valoare mare pentru H implica mai putine interferente între blocurile conflictuale la bitii de hit. Daca H este ales ca raport dintre dimensiunea cache-ului de pe nivelul L2 si cea a cache-ului de pe nivelul L1 (principal), atunci efectul este similar cu mentinerea bitilor hit în nivelul L2 de cache.

În [Vin98, Flor00], autorii acestei monografii au aratat ca o mai mare gradualitate a lui "hit" si "sticky", prin implementarea acestora ca vectori binari cu modificarea corespunzatoare a algoritmului de predictie, poate conduce la performante superioare ale conceptului de SVC. Astfel s-a aratat ca 2 biti de "hit" respectiv "sticky", conduc la rezultate optimele în conditii fezabile ale implementarii hardware.

O problema a implementarii schemei atât a victim cache-ului cât si a selective victim cache-ului este costul implementarii victim cache-ului complet asociativ. Chiar si atunci când aceste cache-uri sunt foarte mici, costul hardware al memoriei adresabila contextual (CAM) poate fi semnificativ. Cache-urile complet asociative cu algoritm de înlocuire LRU pot uneori suferi de o rata de miss mai ridicata decât cache-urile two-way asociative deoarece algoritmul de înlocuire nu este cel optimal [Sug93, Sti94]. Efectul ambelor probleme de mai sus poate fi diminuat prin reducerea asociativitatii victim cache-ului. Cu un victim cache semiasociativ pe 2 cai, nu se observa nici o crestere a ratei de miss la nivelul urmator al ierarhiei de memorie pentru nici o instructiune din trace-urile simulate, atât în victim cache simplu cât si în selective victim cache [Sti94]. Surprinzator,



victim cache-ul semiasociativ pe 2 cai poate îmbunătăți rata de miss și timpul mediu de acces pentru mai multe trace-uri. Acest comportament se datorează algoritmului de înlocuire LRU dezvoltat în victim cache-ul complet asociativ. Blocurile mutate în victim cache-ul complet asociativ ca rezultat al conflictelor din cache-ul principal sunt înlocuite frecvent înainte de a fi accesate din nou. Victim cache-ul semiasociativ pe 2 cai asigură o mai bună izolare pentru blocurile sale în multe cazuri, micșorând rata de miss în victim cache. În plus, datorită dimensiunii sale reduse, miss-urile de conflict formează doar o mică fracțiune din numărul total de accese cu miss în *victim cache* comparativ cu miss-urile de capacitate. Aceasta limitează îmbunătățirea ratei de miss prin creșterea asociativității victim cache-ului, chiar cu un algoritm optimal de înlocuire. Se observă că, victim cache-ul complet asociativ poate îmbunătăți dramatic rata de miss în cazul conflictelor ce implică mai mult de trei blocuri, blocurile conflictuale fiind reținute în victim cache între accese.

Cu un *victim cache* simplu conținutul cache-ului principal mapat direct este neafectat de asociativitatea acestuia. Astfel, rata de miss locală rămâne neschimbată în timp ce se variază asociativitatea victim cache-ului. Prin urmare toate îmbunătățirile efectuate asupra ratei de miss la nivelul L1 de cache pot fi atribuite îmbunătățirii ratei de miss locale a victim cache-ului. Cu victim cache-ul selectiv, asociativitatea poate afecta potențial atât rata de miss locală a cache-ului principal cât și numărul de interschimbări dintre cele două cache-uri. Comparatia timpului de acces ține cont de schimbările aparute în rata de miss și numărul de interschimbări (substanțial micșorat) și de aceea timpul mediu de acces reprezintă o măsură mai bună pentru caracterizarea efectului de asociativitate al victim cache-ului asupra performanței sistemului. Chiar cu un victim cache mapat direct, timpul mediu de acces este mai mare sau egal decât cel din cazul victim cache-ului complet asociativ. Când folosim un victim cache de date semiasociativ pe 2 cai, rezultatele sunt mai proaste decât celea cu un victim cache complet asociativ, atât pentru victim cache simplu cât și pentru victim cache-ul selectiv. Acest lucru nu surprinde, dând conflictelor de acces la date o natură aleatoare. Astfel, un cache complet asociativ poate fi încă atractiv când este folosit ca și cache de date. Totuși, îmbunătățirile observate sunt mai mici. [Sti94].

Chiar dacă nu este nici o îmbunătățire a ratei de hit în cache-ul principal, schema victim cache-ului selectiv poate totuși asigura o îmbunătățire a performanței superioară victim cache-ului simplu. Pentru schema cu SVC rezultatele demonstrează că îmbunătățirile de performanță sunt puternic determinate de impactul algoritmului de predicție asupra numărului de interschimbări cu cache-ul mapat direct. Algoritmul poate de

asemenea contribui la o mai buna plasare a blocurilor în cache, reducând numarul de accese în victim cache si generând rate de hit ridicate în cache-ul mapat direct.[Vin98]

Simularile au utilizat 3 metrici de evaluare a performantelor: *rata de miss* (locala),  *timpul mediu de acces* (globala) si *numarul de interschimbari de blocuri* între MCP respectiv SVC. Concluziile au aratat îmbunatatiri semnificative ale acestor 3 metrici fata de cazul VC simplu. Astfel, de exemplu, numarul de interschimbari a scazut cu 50% pe cache-ul de instructiuni. Pe cache-ul de date, rezultatele au fost mai slabe (benchmark-urile SPECint '92) datorita structurilor de date neregulate. Îmbunatatiri ale conceptului de SVC (vezi modificarea initializarii bitului de hit din cadrul algoritmului, benefica, reducând numarul de interschimbari între MCP si SVC fata de algoritmul original cu cca. 3%) precum si detalii cantitative ale unor cercetari efectuate de catre autori, pot fi gasite în [Vin98, Flor00].

Folosirea victim cache-ului selectiv determina îmbunatatiri ale ratei de miss precum si ale timpului mediu de acces la memorie, atât pentru cache-uri mici cât si pentru cele mari (4Kocteti - 128 Kocteti). Simulari facute pe trace-uri de instructiuni a 10 benchmark-uri SPEC'92 arata o îmbunatatire de aproximativ 21% a ratei de miss, superioara folosirii unui victim cache simplu de 16 Kocteti cu blocuri de dimensiuni de 32 octeti.

### 5.3. DESFASURAREA LUCRARIII

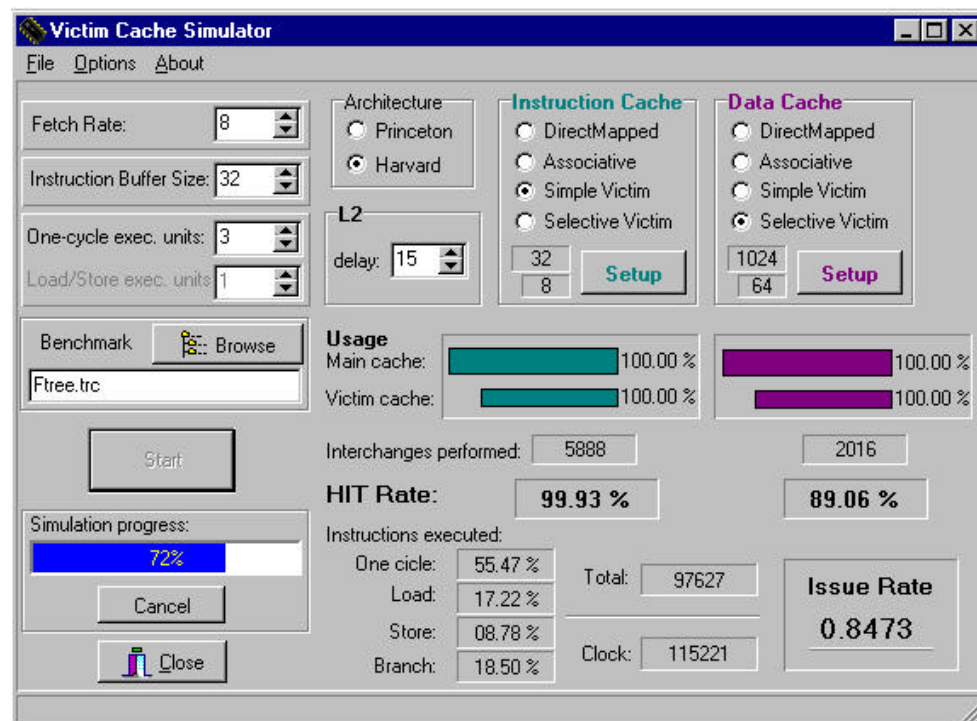
#### 5.3.1. DESCRIEREA SIMULATORULUI

##### 5.3.1.1. ELEMENTELE SIMULATORULUI. DETALII DE IMPLEMENTARE.

Fereastra principala a simulatorului (vezi figura 5.5) contine toate elementele necesare simularii (nucleul de executie al procesorului - rata de fetch, dimensiune buffer prefetch, numar unitati functionale de executie, arhitectura Harvard/Princeton de memorie, selectie benchmark etc.) mai putin cele ce privesc sistemul ierarhic de memorie. Cu ajutorul butoanelor

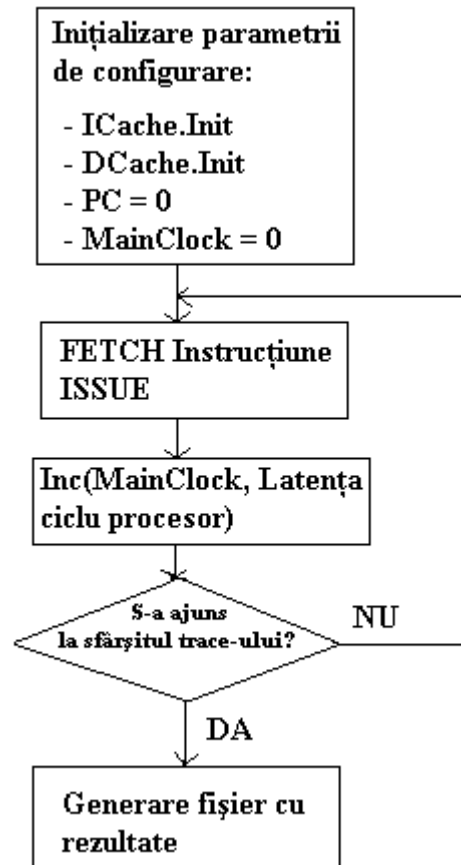
de comanda **Setup** se ajunge în fereastra de configurare a memoriilor cache principale, respectiv (selective) victim cache de unde se aleg valorile corespunzătoare pentru fiecare din parametrii (vezi figura 5.7).

Simularea benchmark-urilor se realizează individual. Orice simulare se va efectua, aceasta va opera exclusiv cu trace-ul selectat cu ajutorul butonului **Browse**.



**Figura 5.6.** Instantaneu cu parametrii de configurare și rezultatele simulării

Arhitectura superscalară implementată are patru nivele distincte în procesarea instrucțiunilor: **fetch** instrucțiune (**IF**), decodificare instrucțiune (**ID**), execuție operație aritmetico-logică sau accesare memorie (**ALU/MEM**) și înscriere rezultat (**WB**). Pe durata unui ciclu procesor se execută în paralel două procese distincte: **fetch** (aducere din memorie) instrucțiune și **issue** (lansare în execuție a instrucțiunilor independente din bufferul de prefetch). Cele două procese, deși implementate secvențial în software, sunt independente și rulează teoretic simultan, conform organigramei din figura 5.6.



**Figura 5.6.** Acțiunile desfășurate pe durata unui ciclu procesor

Procesul de **fetch** consta în extragerea unui grup de instrucțiuni din cache-ul de instrucțiuni (principal sau victima) sau din memoria centrală. Marimea blocului accesat (număr de instrucțiuni citite simultan din cache sau memorie) este specificată de parametrul **FR** (rata de fetch) și poate lua valori din mulțimea {2, 4, 8, 16} instrucțiuni. Expedierea instrucțiunilor spre unitățile de decodificare și execuție se realizează prin intermediul entității “issue”. Pentru evitarea fenomenului de “gâtuire” (bottleneck) introdus de procesul de *fetch instrucțiuni*, e necesar ca rata de fetch să fie mai mare sau cel puțin egală cu numărul de instrucțiuni executate concurrent.

Cele două procese (*fetch* și *issue*) comunică prin intermediul bufferului de prefetch (**IB**) – o structură de tip coadă, parametrizabilă, ce lucrează după principiul FIFO. Cele **FR** instrucțiuni citite din cache sau memoria principală sunt depuse în partea superioară (TOP) a bufferului de prefetch. În același ciclu procesor, din partea inferioară (BOTTOM) a IB

sunt expediate instrucțiuni spre decodificare și execuție. Într-un ciclu procesor, în varianta de simulator implementată, este permisă execuția unei singure instrucțiuni cu referire la memorie (fie Load, fie Store). Execuția instrucțiunilor se face “*in order*” (în ordinea inițială a instrucțiunilor din program). Se presupune o predicție perfectă a ramificațiilor în pipeline, adică cunoașterea în permanență a adresei corecte a următoarei instrucțiuni ce se va executa. Hazardurile RAW sunt complet ignorate în simulatorul implementat. În cazul în care accesul la cache-ul principal se soldează cu miss, accesul este direcționat spre victim cache cu o latență intermediară (victim delay) și dacă și în acest caz rezultatul este miss, instrucțiunile respectiv datele vor fi citite din memoria centrală, cu penalitate maximă (L2 delay). În cazul execuției unei instrucțiuni Load/Store care a produs miss în cache-ul de date (principal și victimă), procesarea instrucțiunilor următoare din “*fereastra de execuție*” [Flor98, VinFlor99], stagnează până la terminarea instrucțiunii cu referire la memorie. Instrucțiunile considerate în simulare sunt de trei tipuri: aritmetico-logice (executate cu latență procesorului – cache-ului), cu referire la memorie (Load/Store) și de salt (branch).

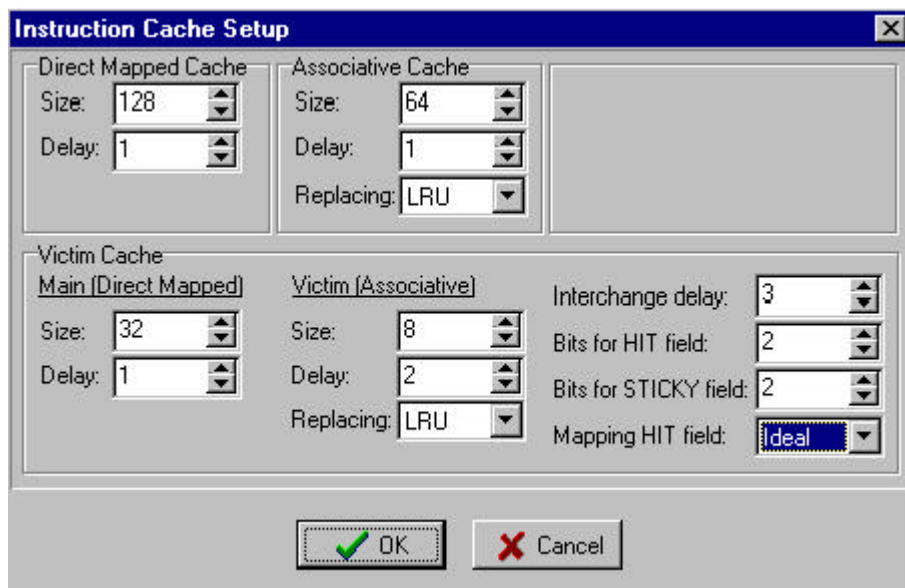
### ***Implementarea memoriilor cache***

Cache-urile principale sunt de tip mapare directă iar victim cache-urile sunt complet asociativ. Memoriile cache sunt implementate sub forma unor blocuri în care procesele *fetch* și *issue* scriu și citesc instrucțiunile / datele necesare. Capacitatea cache-urilor este presetată la valorile 64 locații (principal) + 8 locații (victim) pentru cache-ul de instrucțiuni, respectiv la valorile de 1024 locații (principal) + 32 locații (victim) în cazul cache-ului de date. Capacitatea maximă a cache-urilor în ambele cazuri este de 64k locații pentru cache-ul principal și 64k locații pentru cache-ul victimă. Întârzierea produsă de memoriile cache la fiecare acces poate varia între 1 ciclul și 255 de cicluri, aceeași întârziere putându-se alege și pentru durata secvenței de interschimbare.

**Citirea** din cache cu HIT durează un număr de cicluri corespunzător tipului de memorie cache utilizată. Astfel, în cazul unei memorii cache cu mapare directă, durata presetată de citire este de 1 ciclu. În cazul unei memorii care utilizează victim cache, citirea durează 2 cicluri în cazul unei memorii cache victimă de 8 locații și 3 cicluri în cazul unei memorii cache victimă de 32 de locații, în situația în care nu a avut loc o interschimbare decisa de algoritmul *selective victim cache*. Această întârziere este introdusă de faptul că memoria cache victimă este de tip complet asociativ, căutarea în aceasta făcându-se într-un timp mai mare. În cazul în care algoritmul utilizat decide o interschimbare, durata de citire din cache este egală cu durata

secvenței de interschimbare (presetată la 3 cicluri). La citirea cu miss, răspunsul cache-ului vine după 1 ciclu. Această valoare nu este parametrizabilă, ea fiind stabilită prin program. Este necesar în acest caz accesarea memoriei centrale.

În cazul în care **scrierea** se face în memoria cache principală, procesul durează un număr de cicluri corespunzător întârzierii introduse de accesul la respectivul cache. În cazul în care scrierea se datorează unei interschimbări (hit în victim cache), scrierea durează un număr de cicluri egal cu durata unei interschimbări. În cazul în care scrierea se face în cache-ul direct mapat iar vechea valoare se salvează în *victim cache*, timpul necesar este egal cu suma dintre timpul de acces al cache-ului *victim* (pentru scrierea locației înlocuite în cache-ul principal) și timpul de acces al cache-ului direct mapat. Asadar, scrierea unei locații în cache-ul descris mai sus nu reflectă conceptul de *write-back* sau *write-through*, aceasta făcându-se în implementarea de faptă într-un timp corespunzător ‘destinației’ scrierii.



**Figura 5.7.** Configurarea sistemului ierarhic de memorie

### ***Detalii de implementare***

Memoria cache implementează un bit care specifică disponibilitatea acesteia. În cazul în care bitul respectiv nu este *busy*, cache-ul este ocupat și marcat *busy*. Programul implementează structura de memorii cache precum și un *Instruction Buffer* sub forma unor câmpuri de tip *array* ale unor

obiecte din clasele corespunzătoare. Se prezintă în continuare structura de clase care implementează memoriile cache:

```

TCache = class
-> TDirectMapped = class(TCache) pentru implementarea cache-ului
    principal
-> TAssociative = class(TCache) pentru implementarea victim cache-
    ului

```

```

TInstructionBuffer = class

```

Clasele menționate implementează metode corespunzătoare structurilor hardware cu același nume. Astfel, clasa corespunzătoare memoriei cache implementează metode *Read(addr)*, *Write(addr)*, rezultând în setarea unor biți (*busy*, *hit*) în funcție de rezultat, precum și în gestionarea unor contoare (*HitCnt*, *AccessCnt*) necesare evaluării ulterioare a performanțelor (statistici cantitative și calitative - vezi subcapitolul 5.4. *Probleme propuse spre rezolvare*).

Clasa corespunzătoare buffer-ului de instrucțiuni implementează metode de genul *HasRooms* pentru a verifica un eventual ‘gap’ în IB (test inițiat de secvența de fetch la începutul fiecărui ciclu) sau *AcceptFetchBuffer* pentru copierea instrucțiunilor aduse din cache sau memoria centrală în IB.

Rezultatele simulării vor fi înscrise într-un fișier text al cărui nume și cale sunt stabilite de către utilizator, la încheierea procesului de simulare. Fișierul respectiv conține în primul rând parametrii de configurare ai procesorului: rata de fetch, capacitatea buffer-ului de prefetch, latentă nivelurilor ierarhiei de memorie (cache principal, victim cache, memorie centrală - RAM), capacitatea cache-urilor de instrucțiuni și date, capacitatea victim cache-urilor (simplu sau selectiv), algoritmul de înlocuire în victim cache. Rezultatele calculate sunt: rata de hit în cache, rata de utilizare a cache-ului principal și a celui victimă, numărul de interschimbări produse între cache-ul principal și cel victimă etc. Toate aceste rezultate vor fi generate pentru ambele tipuri de cache-uri (de instrucțiuni și date). Totodată se calculează numărul total de instrucțiuni procesate, numărul total de cicli de execuție, procentajul instrucțiunilor cu referire la memorie (load, store) și de salt (branch).

Metodologia simulării este similară celei descrise în capitolul precedent. Evaluarea performanțelor arhitecturii se face prin metoda *trace driven simulation*. Programele de test sunt aceleași benchmarkuri Stanford, o suită de opt programe relativ scurte, dar caracterizate de calcul intensiv, cu

o dinamica ridicata a numarului de instructiuni, parte dintre ele fiind si puternic recursive. Dinamica distributiei instructiunilor este tipica: 53% instructiuni aritmetico – logice, 13% relationale, 18% instructiuni Load, 12% Store si 17% instructiuni de salt [Flor98, VinFlor99, Col93].

Cu toate acestea, aria de aplicabilitate a programelor de test nu se extinde si la aplicatii grafice, multimedia sau rutine critice ale sistemelor de operare.

#### 5.4. PROBLEME PROPUSE SPRE REZOLVARE

Metricile de performanta utilizate sunt rata medie de procesare (numar total de instructiuni raportat la numar total de cicli de executie), rata de miss în cache (numar de referinte propagate în nivelul urmator al ierarhiei de memorie), numarul de interschimbari produse între cache-ul principal si victim cache si rata de utilizare (numar intrari ocupate raportat la capacitatea cache-ului) a cache-ului principal respectiv a celui victima. Fiecare rezultat este ilustrat atât pentru cache-ul de instructiuni cât si pentru cel de date. Se vor determina parametrii optimi în fiecare din cazuri.

Cu ajutorul simulatorului *hsvictp2.exe* generati [Colc98] atât pentru cache-ul de instructiuni (ICache) cât si pentru cel de date (DCache):

1. Rezultate urmate de grafice privind influenta capacitatii cache-ului asupra ratei de procesare **IR(DM\_size)** si asupra ratei de miss în cache-ul de instructiuni **R<sub>missIC</sub>(DM\_size)** în cele trei situatii:
  - a) fara victim cache.
  - b) cu victim cache simplu.
  - c) cu selective victim cache.
2. Determinati în ce masura selective victim cache-ul reduce numarul de interschimbari dintre cache-ul principal si cel victima **Interchgs(DM\_size)** în situatiile:
  - a) cu victim cache simplu.
  - b) cu selective victim cache.
3. Studiati influenta capacitatii cache-ului de instructiuni asupra ratei de utilizare a respectivului cache **Usage(DM\_size)** în situatiile:
  - a) fara victim cache.
  - b) cu victim cache simplu.
  - c) cu selective victim cache.



4. Determinați influența dimensiunii victim cache-ului de instrucțiuni asupra ratei de procesare  $IR(\text{Victim\_size})$  și asupra ratei de miss în cache-ul de instrucțiuni  $R_{\text{missIC}}(\text{Victim\_size})$  în cele două situații:
  - a) cu victim cache simplă.
  - b) cu selective victim cache.
5. Studiați variația numărului de interschimbări produse între cache-ul principal și victim cache în cazul creșterii capacității celui din urmă, în ipotezele:
  - a) cu victim cache simplă.
  - b) cu selective victim cache.
6. Pentru configurația optimă găsită determinați ce algoritm de înlocuire este mai bun în Victim Cache: LRU, Random sau FIFO.
7. Studiați diferența cantitativă dintre implementarea ideală (pastrarea bitilor de hit în memoria principală) și cea care utilizează tabela de dispersie memorată în primul nivel de cache.
8. Determinați dimensiunea optimă (în biți) a vectorilor binari (hit respectiv sick) pentru o mai bună (graduală) pastrare a istoriei folosirii blocurilor.

#### 5.4.1. INDICAȚII DE REZOLVARE

Acolo unde nu se specifică, se vor considera parametrii optimi de procesare (obținuți de autor prin simulări anterioare) ai arhitecturii, după cum urmează [Flor98, Colc98]:

Fetch Rate  $FR = 8$   
 Instruction Buffer Size  $IB = 32$   
 One-cycle exec. units = 3  
 Load / Store exec. units = 1  
 Second Level Memory Delay  $L2 = 10$

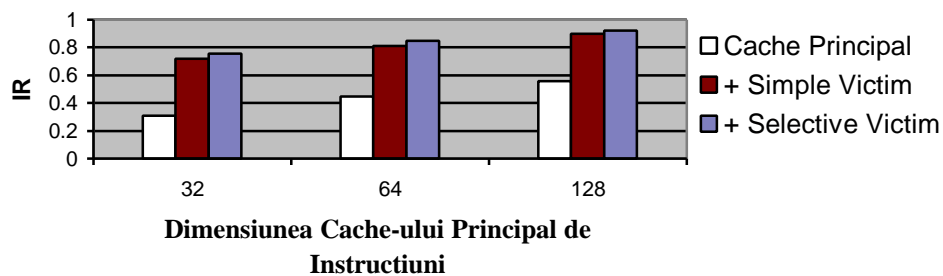
**ICACHE:**  
 Cache Principal Size  $DM = 64$   
 DM Delay = 1  
 Victim Cache Size = 8  
 Victim Cache Delay = 2  
 Victim Cache Replacing = LRU  
 Ticks for interchange = 3

**DCACHE:**  
 DM Cache = 1024  
 DM Delay = 1  
 Victim Cache Size = 32  
 Victim Cache Delay = 3  
 Victim Replacing = LRU  
 Ticks for interchg. = 3

**Bits for HIT field = 1**  
**Bits for STICKY field = 1**  
**HIT Mapping = ideal**

**Bits for HIT field = 1**  
**Bits for STICKY field = 1**  
**HIT Mapping = ideal**

În continuare se exemplifica grafic si se comenteaza influenta capacitatii cache-ului de instructiuni asupra ratei de procesare  $IR(DM\_size)$  în cele trei ipostaze enumerate la punctul 1. paragraful 5.4.



**Figura 5.8.** Variatia ratei de procesare variind capacitatea cache-ului principal, în cazul cache-ului de instructiuni

Asadar capacitatea cache-ului principal de instructiuni variaza între 32 (valoarea maxima aleasa pentru  $IB\_size$ ,  $DM\_size \geq IB\_size$ ) si 128 de locatii. Rezultatele obtinute reprezinta media armonica a ratelor de executie obtinute pentru fiecare program de test in particular, respectiv pentru fiecare capacitate a cache-ului principal (DM) de instructiuni. Se observa în figura 5.8 o rata de procesare optima în cazul cache-ului de instructiuni de 128 de locatii. Din simulari laborioase se va vedea ulterior ca aceasta rata se obtine cu costul unui cache relativ putin utilizat, valoarea optima a capacitatii acestui cache parând a fi cea de 64 de locatii (caz în care cache-ul este mult mai 'bine' utilizat).

## 6. IMPLEMENTAREA SCHEMELOR CLASICE DE PREDICTIE ÎN PROCESOARELE SUPERSCALARE AVANSATE.

---

### 6.1. SCOPUL LUCRĂRII

Capitolele 6 și 7 ale prezentei cărți insistă pe investigarea unor arhitecturi moderne de predictoare ale ramificațiilor de program (**branch**), în vederea reducerii penalităților introduse de instrucțiunile de ramificație în procesoarele pipeline superscalare. Vor fi explorate în mod critic metodologiile de predicție existente, vor fi îmbunătățite, optimizate și stabilite limitările fundamentale. Distingem trei părți (primele două fiind integrate în capitolul 6 iar ultima parte aparținând capitolului 7), fiecare tratând scheme de predicție diferite: de la clasicul BTB, continuând cu predictorul adaptiv corelat pe două nivele GAg și finalizând cu o idee relativ nouă, revoluționară - cea de predictor neuronal.

### 6.2. LUNG MEMENTO TEORETIC

În cadrul procesoarelor pipeline pot apărea situații defavorabile (hazarduri) care determină stagnarea (întârzierea) procesării, având o influență negativă asupra ratei de execuție a instrucțiunilor. Se disting trei categorii de hazarduri: *structurale*, *de date* și *de ramificație*.

Hazardurile structurale sunt determinate de conflicte la resurse comune, adică atunci când mai multe procese simultane aferente mai multor instrucțiuni în curs de procesare accesează o resursă comună. Eliminarea (sau reducerea) prin hardware se realizează prin multiplicarea resurselor sau modificări arhitecturale (bus-uri și cache-uri separate vs. unificate).

Hazardurile de date apar când o instructiune depinde de rezultatele unei instructiuni anterioare din banda de asamblare. Se clasifica în 3 categorii, dependent de ordinea acceselor de citire, respectiv scriere în cadrul instructiunilor: RAW, WAR, WAW. Ultimele 2 tipuri de hazarduri nu reprezinta conflicte reale, ci doar conflicte de nume care pot fi eliminate prin „renaming” aplicat registrilor. Reducerea efectelor defavorabile provocate de hazardurile RAW se realizeaza hardware prin „forwarding” (pasarea anticipata a rezultatului instructiunii  $i$ , nivelului de procesare aferent instructiunii  $j$  care are nevoie de acest rezultat).

Hazardurile de ramificatie sunt generate de instructiunile de ramificatie (branch-uri) si determina pierderi de performanta mai importante decât hazardurile structurale si de date. Reducerea efectelor defavorabile se poate face prin metode software (scheduling = reorganizarea programului sursa) sau prin metode hardware (predictie = determinarea în avans - daca saltul se face - a noului PC *program counter*). Frecventa instructiunilor de salt în programele de calcul este de  $2 \div 8\%$  din instructiunile programului (pentru salturi neconditionate) si  $11 \div 17\%$  din instructiunile programului (pentru salturi conditionate). De asemenea statisticile arata ca, salturile conditionate simple se fac cu probabilitate de 50%, buclele (*loops*) se fac cu probabilitate de 90% si salturile orientate pe bit nu se prea fac.

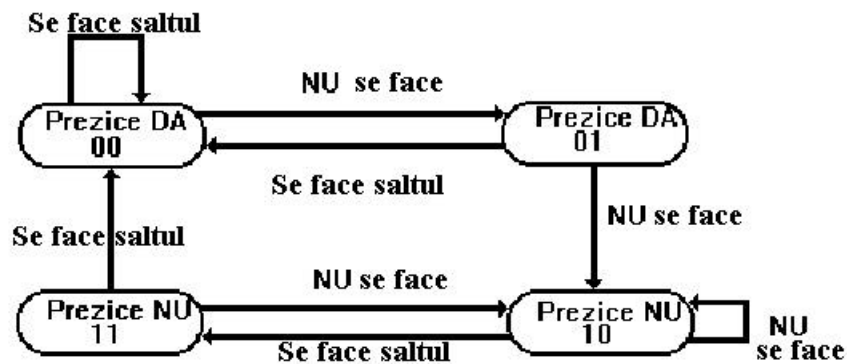
*Predictia prin hardware* reprezinta una din cele mai performante strategii actuale de gestionare a ramificatiilor de program. Strategiile hardware de predictie a branch-urilor au la baza un proces de predictie "run - time" a ramurii de salt conditionat precum si determinarea în avans a noului PC. Ele sunt comune atât procesoarelor scalare cât si celor cu executii multiple ale instructiunilor. Avantajul metodelor hardware fata de cele software îl constituie independenta fata de masina. Cercetari recente insista pe aceasta problema, întrucât s-ar elimina necesitatea reorganizarii soft ale programului sursa si deci s-ar obtine o independenta fata de masina.

Necesitatea predictiei, mai ales în cazul procesoarelor cu executii multiple ale instructiunilor (VLIW, EPIC, superscalare) este imperios necesara. Este o dovada clara ca sunt necesare acurateti ale predictiilor foarte apropiate de 100% pentru a nu se "simti" efectul defavorabil al ramificatiilor de program asupra performantei procesoarelor avansate. O metoda consacrata în acest sens o constituie metoda "**branch prediction buffer**" (BPB). BPB-ul reprezinta o mica memorie adresata cu cei mai putin semnificativi biti ai PC-ului aferent unei instructiuni de salt conditionat. Cuvântul BPB este constituit în principiu dintr-un singur bit. Daca acesta e 1 logic, atunci se prezice ca saltul se va face, iar daca e 0 logic, se prezice ca saltul nu se va face. Evident ca nu se poate sti în avans daca predictia este corecta. Oricum, structura va considera ca predictia este

corecta și va declanșa aducerea instrucțiunii următoare de pe ramura prezisă. Dacă predicția se dovedește a fi fost falsă structura pipeline se evacuează și se va iniția procesarea celeilalte ramuri de program. Totodată, valoarea bitului de predicție din BPB se inversează.

Dezavantajul schemei BPB cu un singur bit se manifestă îndeosebi în cazul buclelor de program. Bazat pe tehnica BPB în acest caz vom avea uzual 2 predicții false: una la intrarea în buclă (prima parcurgere) și alta la ieșirea din buclă (ultima parcurgere a buclei). Astfel, considerând o buclă de program care va fi executată de  $N$  ori, acurătatea predicției va fi de  $(N - 2) * 100 / N \%$ , iar saltul se face în proporție de  $(N - 1) * 100 / N \%$ .

Pentru a elimina acest dezavantaj se utilizează 2 biți de predicție modificabili conform grafului de tranziție de mai jos (numerator saturat). În acest caz acurătatea predicției unei bucle care se face de  $(N - 1)$  ori va fi  $(N - 1) * 100 / N \%$ .

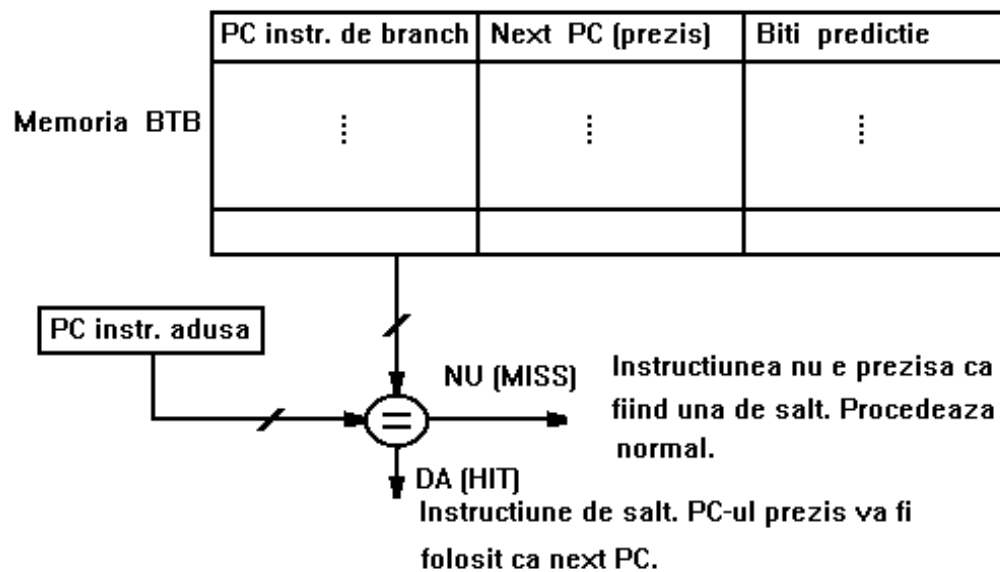


**Figura 6.1.** Automat de predicție de tip numărător saturat pe 2 biți

Prin urmare, în cazul în care se prezice ca branch-ul se va face, aducerea noii instrucțiuni se face de îndată ce conținutul noului PC e cunoscut. În cazul unei predicții incorecte, se evacuează structura pipeline și se ataca cealaltă ramură a instrucțiunii de salt. Totodată, biții de predicție se modifică în conformitate cu graful din figura numit și numărător saturat (vezi Figura 6.1).

Datorită faptului că predictorul propriu-zis este implementat prin automate finite de stare, mai mult sau mai puțin complexe, aceste scheme au un caracter dinamic, comportarea lor adaptându-se funcției de „istoria” saltului respectiv. Există și scheme de predicție statice, caracterizate prin faptul că predicția saltului pentru o anumită istorie a sa este fixă, bazată în general pe studii statistice ale comportamentului saltului

O alta problema delicata consta în faptul ca desi predictia poate fi corecta, de multe ori adresa de salt (noul PC) nu este disponibila în timp util, adica la finele fazei de aducere IF. Timpul necesar calculului noului PC are un efect defavorabil asupra ratei de procesare. Solutia la aceasta problema este data de metoda de predictie numita "**branch target buffer**" (BTB). Un BTB este constituit dintr-un BPB care contine pe lânga bitii de predictie, noul PC de dupa instructiunea de salt conditionat si eventual alte informatii. De exemplu, un cuvânt din BTB ar putea contine si instructiunea tinta a saltului. Astfel ar creste performanta, nemaifiind necesar un ciclu de aducere a acestei instructiuni, dar în schimb ar creste costurile de implementare. Diferenta esentiala între memoriile BPB si BTB consta în faptul ca prima este o memorie operativa iar a 2-a poate fi asociativa, ca în figura urmatoare.



**Figura 6.2.** Structura de predictie BTB asociativa

La începutul fazei IF se declanseaza o cautare asociativa în BTB dupa continutul PC-ului în curs de aducere. În cazul în care se obtine hit se obtine în avans PC-ul aferent instructiunii urmatoare. Mai precis, considerând o structura pipeline pe 3 faze (IF, RD, EX) algoritmul de lucru cu BTB-ul este în principiu urmatorul [Hen96]:

**IF)** Se trimite PC-ul instructiunii ce urmeaza a fi adusa spre memorie si spre BTB. Daca PC-ul trimis corespunde cu un PC din BTB (hit) se trece în pasul RD2, altfel în pasul RD1.

- RD1)** Dacă instrucțiunea adusă e o instrucțiune de branch, se trece în pasul EX1, altfel se continuă procesarea normală.
- RD2)** Se trimite PC-ul prezis din BTB spre memoria de instrucțiuni. În cazul în care condițiile de salt sunt satisfăcute, se trece în pasul EX 3, altfel în pasul EX2.
- EX1)** Se introduce PC-ul instrucțiunii de salt precum și PC-ul prezis în BTB. De obicei această alocare se face în locația cea mai de demult neaccesată (Least Recently Used- LRU).
- EX2)** Predicția s-a dovedit eronată. Trebuie reluată faza IF de pe cealaltă ramură cu penalizările de rigoare datorate evacuării structurilor pipeline.
- EX3)** Predicția a fost corectă, însă numai dacă și PC-ul predictionat este într-adevăr corect, adică neschimbat. În acest caz, se continuă execuția normală.

#### **Avantajele și dezavantajele tehnicii BTB**

- 1) BTB nu îmbunătățește performanța pentru o predicție corectă de tipul "saltul nu se face", întrucât în acest caz structura se comportă în mod implicit la fel ca și o structură fără BTB.
- 2) Un branch trebuie introdus în BTB, **cu prima ocazie când el se va face**. Un salt care ar fi prezis că nu se va face nu trebuie introdus în BTB pentru că nu are potențialul de a îmbunătăți performanța. Din acest motiv, există strategii care atunci când trebuie evacuat un branch din BTB îl evacuează pe cel cu potențialul de performanță minim, care nu coincide neapărat cu cel mai puțin folosit (vezi [Dub91, Per93]). Astfel în [Per93] se construiește câte o variabilă MPP (Minimum Performance Potential), implementată în hardware, asociată fiecărui cuvânt din BTB. Evacuarea din BTB se face pe baza MPP-ului minim. Acesta se calculează ca un produs între probabilitatea ca un branch din BTB să fie din nou accesat (LRU) și respectiv probabilitatea ca saltul să se facă. Aceasta din urmă se obține pe baza unei istorii a respectivului salt (taken / not taken). Minimizarea ambilor factori duce la minimizarea MPP-ului și deci la evacuarea respectivului branch din BTB, pe motiv că potențialul său de performanță este minim.

În literatura [Hen96, Per93], bazat pe testări laborioase, se arată că se obțin predicții corecte în cca. 88% din cazuri folosind un bit de predicție și respectiv în cca.93% din cazuri folosind 16 biți de predicție. Microprocesorul Intel Pentium avea un predictor de ramificații bazat pe un BTB cu 256 de intrări.

O problemă dificilă este determinată de **instrucțiunile de tip RETURN** întrucât o aceeași instrucțiune, poate avea adrese de revenire

diferite, ceea ce va conduce în mod normal la predictii eronate, pe motivul modificarii adresei eronate în tabela de predictii. Desigur, problema se pune atât în cazul schemelor de tip BTB cât si a celor de tip corelat, ce vor fi prezentate în continuare. Solutia de principiu [Kae91], consta în implementarea în hardware a unor asa zise "stack - frame"- uri diferite. Acestea vor fi niste stive, care vor contine perechi CALL/ RETURN cu toate informatiile necesare asocierii lor corecte. Astfel, o instructiune CALL poate modifica dinamic în tabela de predictii adresa de revenire pentru instructiunea RETURN corespunzatoare, evitându-se astfel situatiile nedorite mai sus schitate.

Schemele de predictie anterior prezentate se bazeau pe comportarea recenta a unei instructiuni de salt, de aici predictionându-se comportarea viitoare a acelei instructiuni de salt. Este posibila îmbunatatirea acuratetii predictiei daca aceasta se va baza pe comportarea recenta a altor instructiuni de salt, întrucât frecvent aceste instructiuni pot avea o comportare corelata în cadrul programului. Schemele bazate pe aceasta observatie se numesc **scheme de predictie corelata** si au fost introduse pentru prima data în 1992 în mod independent de catre *Yeh* si *Patt* si respectiv de Pan [Hen96, Yeh92, Pan92]. Daca doua branch-uri sunt corelate, cunoscând comportarea primului se poate anticipa comportarea celui de al doilea, ca în exemplul de mai jos:

```
if (cond1)
....
if (cond1 AND cond2)
```

Se poate observa ca functia conditionala a celui de al doilea salt este dependenta de cea a primului. Astfel, daca prima ramificatie nu se face atunci se va sti sigur ca nici cea de a doua nu se va face. Daca însa prima ramificatie se va face atunci cea de a doua va depinde exclusiv de valoarea logica a conditiei "cond2". Asadar în mod cert aceste doua ramificatii sunt corelate, chiar daca comportarea celui de al doilea salt nu depinde exclusiv de comportarea primului.

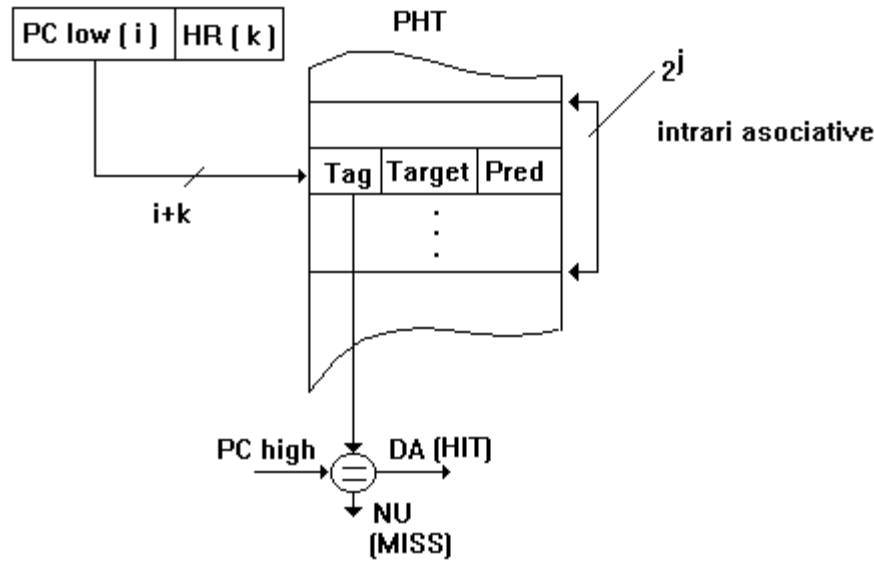
Un alt avantaj al acestor scheme este dat de simplitatea implementarii hardware, cu putin mai complexa decât cea a unui BPB clasic. Aceasta se bazeaza pe simpla observatie ca "istoria" celor mai recent executate m salturi din program, poate fi memorata într-un registru binar de deplasare pe m ranguri (registru de predictie). Asadar adresarea cuvântului de predictie format din n biti si situat într-o tabela de predictii, se poate face foarte simplu prin concatenarea c.m.p.s. biti ai PC-ului instructiunii de salt curente cu acest registru de deplasare în adresarea BPB-ului de predictie. Ca si în



cazul BPB-ului clasic, un anumit cuvânt de predicție poate corespunde la mai multe salturi. Exista în implementare 2 nivele deci: un **registru de predicție** al carui conținut concatenat cu PC- ul c.m.p.s. al instrucțiunii de salt pointeaza la un cuvânt din **tabela de predicții** (aceasta conține bitii de predicție, adresa destinație, etc.). În [Yeh92], nu se face concatenarea PC - registrului de predicție și în consecință se obțin rezultate nesatisfăcătoare datorită **interferenței diverselor salturi** la aceeași locație din tabela de predicții.

În [Pan92], o lucrare de referință în acest plan, se analizează calitativ și cantitativ într-un mod foarte atent, rolul informației de corelație, pe exemple concrete extrase din benchmark-urile SPECint '92. Se arată că bazat pe predictor de tip număratoare saturate pe 2 biți, schemele corelate (5-8 biți de corelație utilizați) ating acurateți ale predicțiilor de până la 11% în plus față de cele clasice.

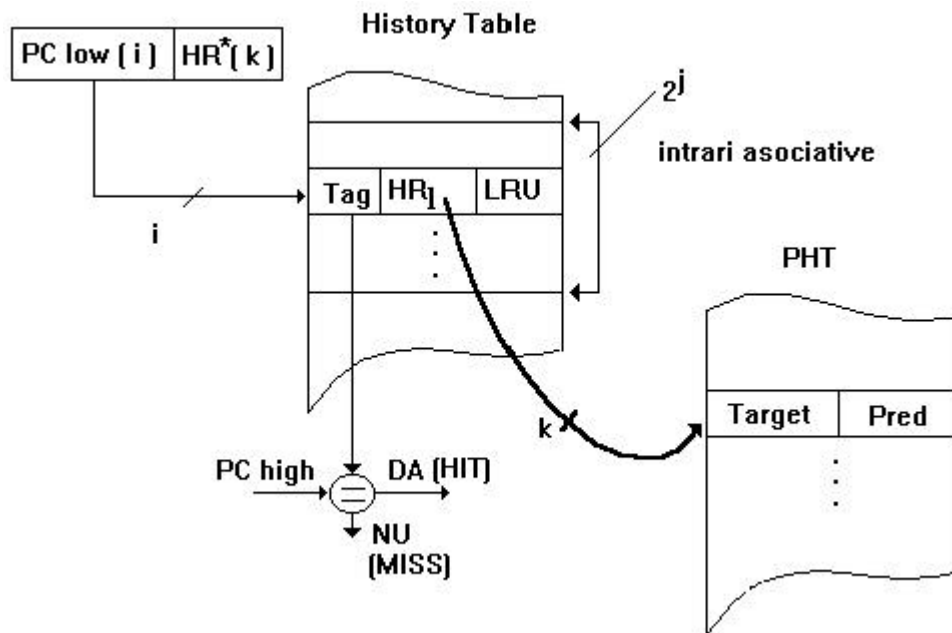
Exista citate în literatura mai multe implementări de scheme de predicție a ramificațiilor, prima implementare comercială a unei astfel de scheme făcându-se în microprocesorul Intel Pentium Pro. Astfel, implementarea tipică a unui predictor corelat de tip **GAg** (Global History Register, Global Prediction History Table) este prezentată în figura 6.3. Tabela de predicții PHT (Prediction History Table) este adresată cu un index rezultat din concatenarea a **două informații ortogonale**: PC<sub>low</sub> (i biți), semnificând gradul de localizare al saltului, respectiv registrul de predicție (HR - History Register pe k biți), semnificând "contextul" în care se situează saltul în program. Ambele contribuții s-au făcut cu scopul eliminării interferențelor branch-urilor în tabela de predicție. Adresarea PHT exclusiv cu HR ca în articolul [Yeh92], ducea la serioase interferențe (mai multe salturi puteau accesa aceleași automat de predicție din PHT), cu influențe evident defavorabile asupra performanțelor. Desigur, PHT poate avea diferite grade de asociativitate. Un cuvânt din această tabelă are un format similar cu cel al cuvântului dintr-un BTB. Se pare că se poate evita concatenarea HR și PC<sub>low</sub> în adresarea PHT, cu rezultate foarte bune, printr-o funcție de dispersie tip SAU EXCLUSIV între acestea, care să adreseze tabela PHT. Aceasta are o influență benefică asupra capacității tablei PHT.



**Figura 6.3.** Structura de predicție de tip GAg

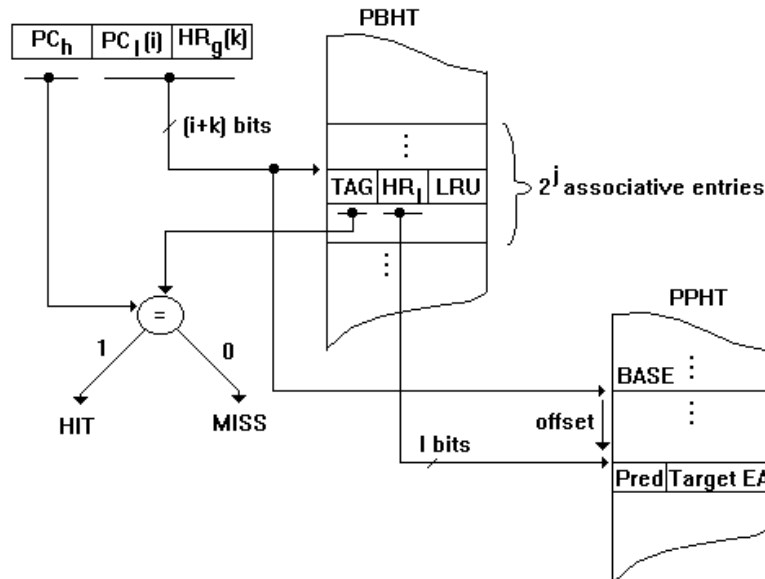
În scopul reducerii interferențelor diverselor salturi în tabela de predicții, în [Yeh92] se prezintă o schema numită **PAG**- Per Address History Table, Global PHT, a cărei structură este oarecum asemănătoare cu cea a schemei GAg. Componenta  $HR^*(k)$  a fost introdusă în [Vin97], având semnificația componentei HR de la varianta GAg, adică un registru global care memorează comportarea ultimelor  $k$  salturi. Fără această componentă, schema PAG și-ar pierde din capacitatea de adaptare la contextul programului în sensul în care schema GAg o face. *Yeh* și *Patt* renunță la informația de corelație globală (HRg) în trecerea de la schemele de tip GAg la cele de tip PAG, în favoarea exclusivă a informației de corelație locală (HRI). În schimb, componenta HR din structura History Table, conține "istoria locală" (taken / not taken) a saltului curent, ce trebuie predicționat. După cum se va arăta mai departe, performanța schemei PAG este superioară celei obținute printr-o schemă de tip GAg, cu tributul de rigoare plătit complexității hardware.

Schema de predicție de tip PAG predicționează pe baza a **3 informații ortogonale**, toate disponibile pe timpul fazei IF: istoria HRg a anterioarelor salturi corelate (taken / not taken), istoria saltului curent HRI și PC-ul acestui salt.



**Figura 6.4.** Structura de predicție de tip PAg

Dupa cum se observa din figura 6.4 (PAg) desi tabela de pe primul nivel este specifica unui salt (PC concatenate cu HRg), tabela de pe nivelul al doilea este globala, fiind adresata doar cu istoria locala a unui salt (HR). acest fapt conduce la existenta de interferente între instructiunile de salt.



**Figura 6.5.** Structura de predicție de tip PAP

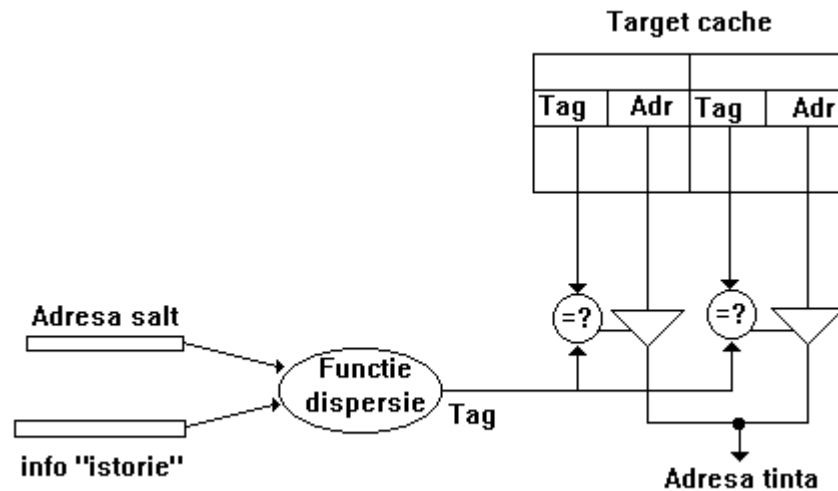
Daca adresarea tabelii PHT s-ar face în schema PAg cu HR concatenat cu PClow(i), atunci practic fiecare branch ar avea propria sa tabela PHT, rezultând deci o schema si mai complexa numita **PAp** (Per Address History Table, Per Address PHT) [Yeh92], a carei schema de principiu este prezentata anterior (figura 6.5). Complexitatea acestei scheme o face practic neimplementabila în siliciu la ora actuala, fiind doar un model utilizat în cercetare.

Desigur, este posibil ca o parte dintre branch-urile memorate în registrul HR, sa nu se afle în corelatie cu branch-ul curent, ceea ce implica o serie de dezavantaje. În astfel de cazuri pattern-urile din HR pot pointa în mod inutil la intrari diferite în tabela de predictii, fara beneficii asupra performantei predictiei, separându-se astfel situatii care nu trebuiesc separate. Mai mult, aceste situatii pot conduce la un timp de "umplere" a structurilor de predictie mai îndelungat, cu implicatii defavorabile asupra performantei [Eve96].

Reducerea efectelor defavorabile datorate interferentei predictiilor se poate realiza prin extinderea informatiei de corelatie (asocierea fiecarui bit din HRg cu PC-ul afferent saltului respectiv si determinarea predictiei pe baza acestei informatii mai complexe). In acest fel, rezulta ca la contexte diferite de aparitie a unui salt, se vor apela automate diferite de predictie, asociate corect contextelor. Comprimarea acestui complex de informatie este posibila si chiar necesara, avand în vedere necesitatea unor costuri rezonabile pentru aceste scheme. Se recomanda utilizarea unor functii de dispersie simple (OR, XOR).

O problema dificila în predictia branch-urilor o constituie salturile codificate în moduri de adresare indirecte, a caror acuratete a predictiei este deosebit de scazuta prin schemele anterior prezentate (cca.50%). În [Cha97] se propune o structura de predictie numita "target cache" special dedicata salturilor indirecte. În acest caz predictia adresei de salt nu se mai face pe baza ultimei adrese tinta a saltului indirect ca în schemele de predictie clasice, ci pe baza alegerii uneia din ultimele adrese tinta ale respectivului salt, memorate în structura. Asadar, în acest caz structura de predictie, memoreaza pe parcursul executiei programului pentru fiecare salt indirect ultimele N adrese tinta.

Predictia se va face deci în acest caz pe baza urmatoarelor informatii: PC-ul saltului, istoria acestuia, precum si ultimele N adrese tinta înregistrate. Structura de principiu a target cache-ului e prezentata în figura 6.6. O linie din acest cache contine ultimele N adrese tinta ale saltului împreuna cu tag-ul aferent.



**Figura 6.6.** Predicția adresei în cazul salturilor indirecte

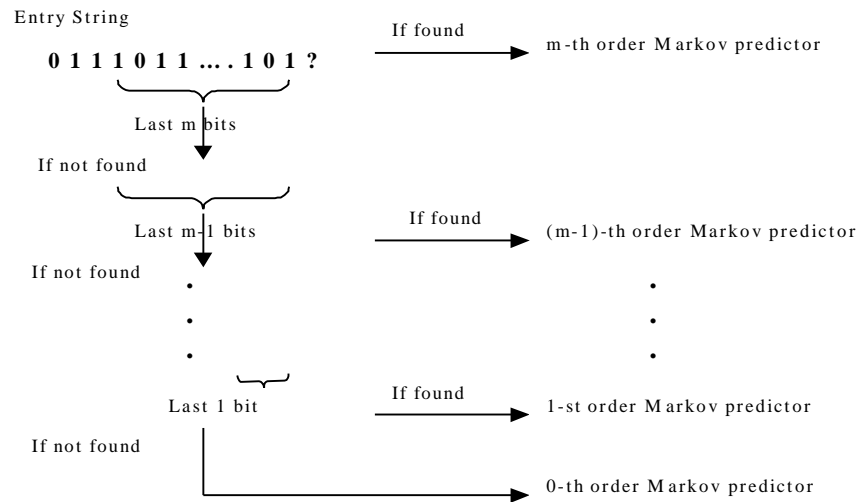
Informația "istorie" provine din două surse: istoria saltului indirect sau a anterioarelor salturi și respectiv ultimele  $N$  adrese țintă, înscrise în linia corespunzătoare din cache. Aceste două surse de informație binară sunt prelucrate prin intermediul unei funcții de dispersie (SAU EXCLUSIV), rezultând indexul de adresare în cache și tag-ul aferent. După ce adresa țintă a saltului devine efectiv cunoscută, se va introduce în linia corespunzătoare din cache. Schema mizează pe faptul că la același context de apariție a unui salt indirect se va asocia o aceeași adresă țintă. Prin astfel de scheme, măsurat pe benchmark-urile SPECint '95 acurătatea predicției salturilor indirecte crește și ca urmare, câștigul global asupra timpului de execuție este de cca 4.3% - 9% [Cha97].

O altă idee nouă în predicția branch-urilor o reprezintă predicția pe baza de lanțuri Markov utilizând algoritmul PPM (Prediction by Partial Matching). Parintele conceptului de "predictor markovian", introdus pentru prima dată în 1997, este Trevor Mudge, de la Universitatea din Michigan. Predicția se bazează pe algoritmul universal de compresie prin potrivire parțială (PPM), care s-a dovedit a fi optim în compresii și extrageri anticipate de date, și în recunoașterea vorbirii. De asemenea, PPM reprezintă un instrument de diagnoză în măsurarea limitei "inerente" de predictibilitate din programele de test.

Algoritmul PPM de ordinul  $m$  (predictor PPM complet) este alcătuit din  $(m+1)$  predictoare Markov. Un predictor Markov de ordinul  $j$  predicționează bit-ul următor pe baza pattern-urilor precedente de  $j$  biti (în concordanță cu cel mai frecvent bit care urmează pattern-ului). De câte ori

apare un miss (nu regasim pattern-ul de  $j$  biti în sirul de intrare), PPM reduce pattern-ul cu 1 bit si foloseste predictorul Markov de ordin imediat inferior pentru predictie. Procedeul continua pâna când se identifica un pattern si se poate realiza predictia corespudenta.

În esenta pentru predictie, se cauta pattern-ul memorat în registrul HRg pe  $k$  biti într-un sir binary mai lung al istoriei salturilor anterioare. Daca acest pattern este gasit în sirul respective cel putin o data, predictia se face corespunzator, pe baza unei statistici care determina de câte ori a fost urmat de 1 logic (taken) - *valoarea predictionata este aceea care a urmat cu cea mai mare frecventa contextului considerat*. Daca însa pattern-ul din HRg nu a fost gasit în sirul de istorie, se construiesc un nou pattern mai scurt prin eliminarea ultimului bit din HRg si algoritmul se reia pe cautarea acestui nou pattern, s.a.m.d. pâna la gasirea unui anumit pattern în sir. Se observa ca predictia este functie si de contextul considerat, un context mai “*bogat*” (“*lung*”) conducând adeseori la o acuratete mai ridicata a predictiei (nu întotdeauna însa: câteodata contextul se poate comporta ca “*zgomot*”). Se arata ca desi complexitatea implementarii acestei noi scheme creste de cca. 2 ori fata de o schema corelata, eficienta sa - la acelasi buget al implementarii - este clar superioara. La nivelul tehnologiei actuale, implementarea unui asemenea predictor PPM complet ar putea fi prohibita.



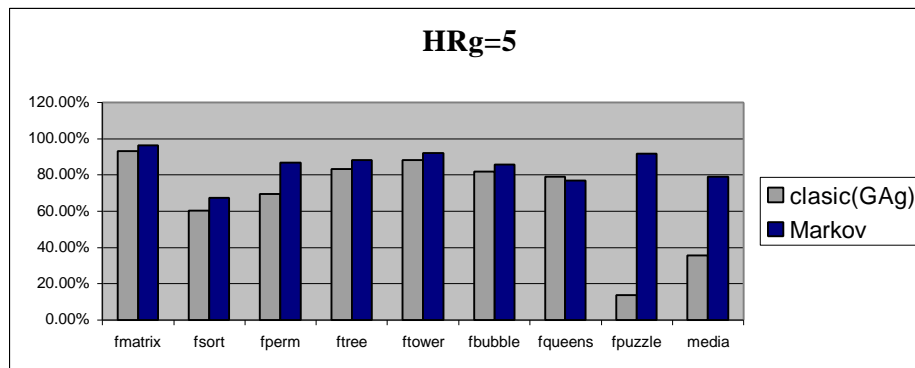
**Figure 6.7.** Predictor PPM de ordin  $m$

Exista similitudini puternice între predictoarele adaptive pe 2 nivele si predictoarele Markov [Mud96] (primul nivel este identic - ambele scheme

de predicție utilizează comportarea ultimelor  $m$  salturi (T/NT) pentru cautarea în structurile de date corespondente, iar numărătoarele saturate de pe al doilea nivel sunt approximate cu număratorul de frecvențe Markov (frecvența de apariție a valorii 1 și 0). Cele două scheme de predicție diferă doar prin informația folosită pentru actualizarea celor două numărătoare.

Un predictor PPM complet de ordinul  $m$  cuprinde  $(m+1)$  predictor Markov (pentru predicție sunt cautate pattern-uri de lungime  $m$  până la 0 biti). Datorită asemănărilor enunțate anterior, predictorul PPM poate fi văzut ca o mulțime de predictor adaptive pe două nivele, având nu doar un registru de istorie a salturilor pe  $m$  biti ci un set de registre cu dimensiunile de la  $m$  la 0. Rezultă în acest fel, generarea a  $(m+1)$  tabele PHT având dimensiuni între  $2^m \cdot k$  biti (unde  $k$  reprezintă numărul de biti pentru codificarea contorului de frecvențe Markov) și  $2^0 \cdot k$  biti. Spațiul total de memorie necesar este, în acest sens:  $2^m \cdot k + 2^{m-1} \cdot k + \dots + 2^0 \cdot k = k \cdot (2^{m+1} - 1)/(2 - 1) \gg 2^{m+1} \cdot k = 2 \cdot (2^m \cdot k) = 2 \cdot (\text{spațiul de memorie necesar unui GAg cu istorie globală pe } m \text{ biti})$ . În concluzie, costul hardware al unui predictor PPM de ordinul  $m$  complet este echivalent cu dublul costului unui predictor adaptiv corelat pe două nivele utilizând tot  $m$  biti de istorie globală, ceea ce îl face dificil de implementat în siliciu.

Încercând să evite complexitatea acestui circuit, autorii acestei lucrări au investigat [Vin99] comportarea unui predictor PPM simplificat, constând într-un predictor Markov de ordin  $m$ , chiar fezabil [Mud96] pentru a fi implementat în hardware. Rezultatele simulărilor de tip trace driven pe benchmark-urile Stanford au arătat că acurătatea predicției obținută cu predictorul PPM este substanțial mai bună față de folosirea unui predictor GAg clasic (vezi figura 6.8).



**Figure 6.8.** Predictorul PPM simplificat surclasează pe schema GAg corespondentă din punct de vedere al acurătății de predicție

Tabelul urmator (tabelul 6.1) prezinta câteva moduri de tratare a instructiunilor de salt si implementarile comerciale în care apar[Ung99].

<b>Tehnica</b>	<b>Exemple de implementare</b>
<i>Fara branch prediction</i>	Intel 8086
<b>Predictie statica</b> <input type="checkbox"/> Întotdeauna “nu se face” <input type="checkbox"/> Întotdeauna “se face” <input type="checkbox"/> Saltul <u>înapoi</u> “se face”, saltul <u>înainte</u> “nu se face”	Intel i486 Sun SuperSPARC HP PA-7x00
<b>Predictie dinamica</b> BTB – 1 bit  BTB – 2 biti  Predictoare adaptive pe doua nivele - GAg	DEC Alpha 21064, AMD K5  Power PC 604 si MIPS R10000 ( BTB cu 512 intrari, acuratetea predictiei=87%, simulari efectuate pe SPEC'92)  Intel Pentium PRO, Pentium II AMD – K6
<b>Predictie hibrida</b> Static +Dinamic	DEC Alpha 21264
<b>Executie predicativa a instr.</b>	INTEL / HP Merced si majoritatea procesoarelor de semnal, procesorul ARM

Tabelul 6.1

### Implementari comerciale ale schemelor clasice de predictie



### 6.3. DESFASURAREA LUCRARIII

#### 6.3.1. PREDICTOR DE SALTURI DE TIP BTB

Lucrarea de fata insista pe investigarea unor arhitecturi de predictoare a ramificatiilor de program (**branch**) de tip BTB (*branch target buffer*), în vederea reducerii penalitatilor introduse de instructiunile de ramificatie în procesoarele pipeline superscalare.

Partea practica a lucrarii consta în simulare efectuata pe benchmark-urile HSA (*Hatfield Superscalar Architecture*) Stanford. Pentru simulare, sunt folosite 8 fisiere trace identice ca nume dar cu extensia (\*.tra). Aceste fisiere sunt o prelucrare a programelor scrise în mnemonica de asamblare (\*.ins) si a trace-urilor originale (\*.trc), cu scopul de a evidentia toate salturile (inclusiv cele care nu se fac).

Întregul fisier trace (\*.trc) este o înlantuire de triplete <**TipInstr** **AdrCrt** **AdrDest**>, unde **TipInstr** poate lua una din valorile 'B' – branch, 'L' – load, 'S' – store; **AdrCrt** reprezinta valoarea registrului PC – adresa instructiunii curente, iar **AdrDest** reprezinta adresa de memorie a datei accesate – în cazul instructiunilor cu referire la memorie ('L' / 'S') sau adresa destinatie a saltului – în cazul instructiunilor de salt si ramificatie ('B').

Desi simularea poate ignora instructiunile Load / Store din trace, deoarece în acest caz suntem interesati numai de comportarea branch-urilor, exista totusi o deficiente a acestor trace-uri: nu evidentiaza salturile care nu se fac. Din acest motiv s-au generat noile trace-uri (fisierele \*.tra). Acestea contin doar branch-urile (atât cele care se fac cât si cele care nu se fac) si exclude instructiunile Load / Store. Forma acestor fisiere este urmatoarea:

BT	12	30
BS	32	98
BM	100	33
NT	36	37
BRA	2	100
MOV	PC, RA	(RETURN)

Reprezentarea salturilor se face tot sub forma unor triplete <**TipBr** **AdrCrt** **AdrDest**>, unde **TipBr** se prezinta sub forma unei codificari pe doua

caractere, primul dintre ele indica dacă saltul se face ('B' – saltul se face, 'N' – saltul nu se face), iar al doilea caracter indica tipul saltului: 'T' sau 'F' – salturi conditionate, 'S' – apeluri de tip Call, 'M' – apeluri de tip Return, 'R' – salturi neconditionate. Alegerea acestei codificări a fost inspirată de mnemonicile întâlnite în sursa în limbaj de asamblare (BT, BF – salt conditionat, BSR – instrucțiune de tip Call, BRA – salt neconditionat și MOV PC, RA – instrucțiune de tip Return).

Automatul de predicție este descris printr-un sir de caractere cu un format mai special, ce prezintă atât numărul de stări, tranzițiile între stări cât și predicția aferentă fiecărei stări. Pentru o mai bună înțelegere a funcționării automatului exemplificăm pe două situații diferite:

1. automatul **ABAB:2** - pe un bit
2. automatul **BCBAADCD:12** - pe 2 biți

Tabelul care descrie funcționarea primului automat este următorul:

Stare curentă	Stare următoare		Predicție
	Pentru intrare = 0	Pentru intrare = 1	
A	A	B	0
B	A	B	1

*Tabelul 6.2.*

### Funcționarea automatului de predicție pe 1 bit

Prima parte a sirului până la caracterul ':' reprezintă tranzițiile pentru starea 'A' cu intrare 0, apoi cu intrare 1, apoi tranzițiile din 'B' pentru aceleași intrări. Numărul din a doua parte este "văzut" în binar sub forma "0010" și reprezintă ieșirile asociate fiecărei stări în parte (stării 'A' îi este asociat cel mai puțin semnificativ bit, în acest caz bitul '0', următorul bit lui 'B', bitul '1' etc).

Tabelul care descrie funcționarea celui de-al doilea automat arată astfel:

Stare curentă	Stare următoare		Predicție
	Pentru intrare = 0	Pentru intrare = 1	
A	B	C	0
B	B	A	0
C	A	D	1
D	C	D	1

*Tabelul 6.3.*

### Tranzițiile automatului de predicție pe 2 biți

### 6.3.1.1. METODOLOGIA DE SIMULARE ÎN CADRUL SCHEMEI *BRANCH TARGET BUFFER* (BTB)

BTB reprezintă o memorie care conține pe lângă bitii de predicție, noul PC de după instrucțiunea de salt condiționat și eventual alte informații. De exemplu, un cuvânt din BTB ar putea conține și instrucțiunea țintă a saltului, crescând astfel performanța, nemaifiind necesar un ciclu de aducere a acestei instrucțiuni, dar în schimb crescând și costurile de implementare (vezi figura 6.2). Memoria BTB poate fi implementată fie asociativ fie mapat direct.

În principiu, simulatorul dezvoltat va funcționa după următorul algoritm, astfel:

#### a) **Initializare simulator:**

- Dimensiunea tabelului (*număr de intrări*)
- Initializarea cu 0 a adreselor destinație
- Initializare stare automat de predicție
- Tipul de arhitectură ales (mapat direct / complet asociativ)

#### b) **Simularea propriu-zisă**

Se stabilește de către utilizator benchmark-ul de tip *trace* care va fi utilizat. Din acest benchmark, se citește secvențial instrucțiunile de ramificație și se compară predicția reală din trace cu cea propusă din tabelă. În funcție de rezultatul predicției se actualizează informațiile necesare în tabelă. Algoritmul se reia până la epuizarea instrucțiunilor de salt din trace.

În cazul în care memoria este mapată direct adresarea se va face cu adresa obținută prin calcularea ( $PC \bmod \text{Locații\_BTB}$ ), unde PC reprezintă adresa instrucțiunii de salt curente, iar Locații\_BTBT este numărul de locații din BTB. Astfel, în cazul unei evacuări se știe exact ce instrucțiune va fi evacuată. În cazul memoriei complet asociative, orice salt poate fi memorat oriunde în BTB, iar pentru evacuare s-a folosit un algoritm de tip LRU.

Pentru exemplificare presupunem ca instrucțiunea curentă de salt din trace este **BT 12 30**.

1. Se preia PC-ul instrucțiunii de salt taken (începe cu B) din trace și se caută în BTB o intrare corespundentă. Dacă BTB este mapat direct pot găsi o singură intrare iar dacă este asociativ mai multe intrări.

a) Dacă se găsește o intrare corespundentă (hit pe direcție) atunci se caută dacă target-ul din trace (30) este egal cu target-ul din BTB (la cel mapat direct). Dacă este egalitate și predicția este egală cu 1 (bitul corespunzător din BTB) atunci predicția este corectă și intrarea pentru automatul de predicție este 1, acesta trecând într-o nouă stare.

Presupunând ca era în B înainte, va trece în starea  $S = f(B,1)$  rezultând si predictia viitoare.

La BTB asociativ se alege din grupul instructiunilor care au generat hit pe directie, acea instructiune care are hit si pe target (BTB cu trace). Daca exista, se verifica daca bitul de predictie este 1 si în acest caz predictia fiind corecta, altfel (bitul  $P=0$ ) predictia fiind gresita. Se actualizeaza doar automatul de predictie (starea si predictia viitoare). Daca nu exista în BTB si mai este loc se adauga în BTB. Daca nu, va fi înlocuita în functie de LRU o alta locatie, nu neaparat una care a avut hit pe directie. Predictia a fost eronata. La inserarea în BTB se initializeaza automatul de predictie.

La BTB, în caz ca s-a gasit hit pe directie si target, dar bitul de predictie este 0, rezulta predictie gresita, se trece în starea corespunzatoare automat si predictia se modifica si ea corespunzator. În cazul în care la BTB mapat direct nu s-a gasit hit pe target, indiferent de predictie ea nu este corecta (pentru ca si daca ar fi 1, ea ar duce la alta adresa destinatie. Daca  $pred=0$  semnifica saltul nu se face la respectivul target, ceea ce înseamna corect (saltul se face dar la alta adresa), rezulta în BTB se înlocuieste target-ul corect cu cel din trace si predictia se actualizeaza.

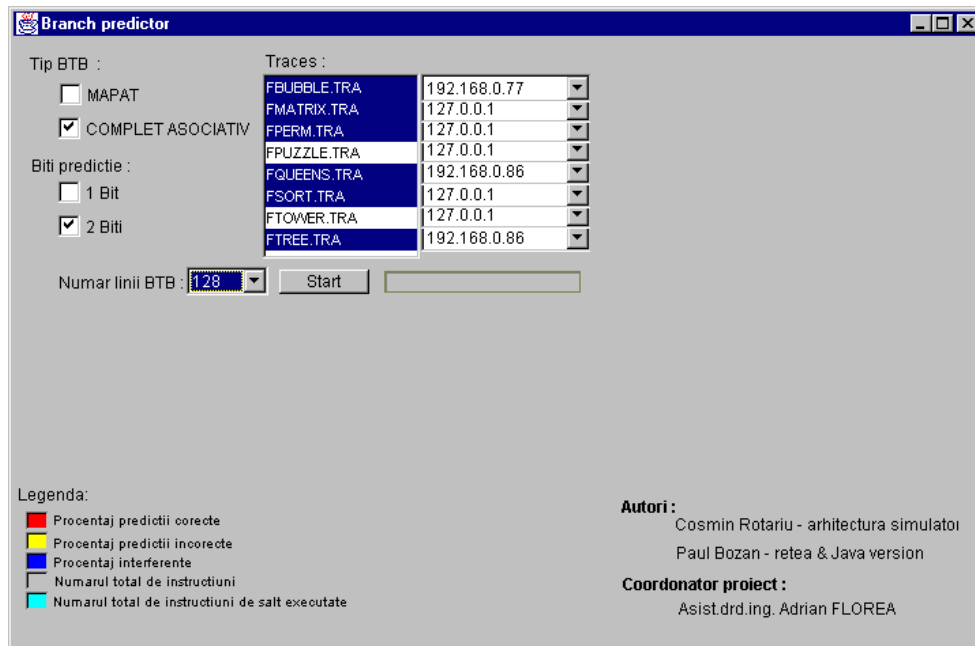
b) Daca nu a existat hit pe directie si branch-ul din trace este taken (se face, începe cu B nu cu N) este adaugat în BTB (daca e mapat direct în locatia conflictuala, daca e asociativ în locatia care are LRU cel mai mic).

2. Daca branch-ul din trace este NotTaken (nu se face), atunci nu se adauga în BTB (întrucat nu îmbunatateste predictia). Totusi se verifica daca TAG-ul este în BTB (12) =hit pe directie si se actualizeaza automatul de predictie. Daca nu exista nici nu se introduce în BTB, se continua procesarea normal.

### 6.3.1.2. PREZENTARE PROGRAM DE SIMULARE

#### Interfata cu utilizatorul:

La lansarea în executie a simulatorului (de exemplu: *“runClient.bat”*, pe ecranul calculatorului gazda apare o fereastră de dialog înzestrata cu butoane si liste de diverse tipuri (vezi figura 6.9).



**Figura 6.9.** Pregătirea pentru simulare - configurarea predictorilor și a stațiilor server

*Locații BTB* = numărul de linii din BTB

*Tip BTB* = se va alege între o arhitectura mapată direct sau una complet asociativ

*Biti de predicție* = numărul de biti folosiți în predicție de către automatul de predicție

*Fisiere pentru simulare* = se vor afișa toate fișierele cu extensia \*.tra din directorul curent. Dintre acestea se vor selecta acelea pe care se dorește să se facă simularea (se pot selecta și toate).

Simularea se va efectua în modul următor:

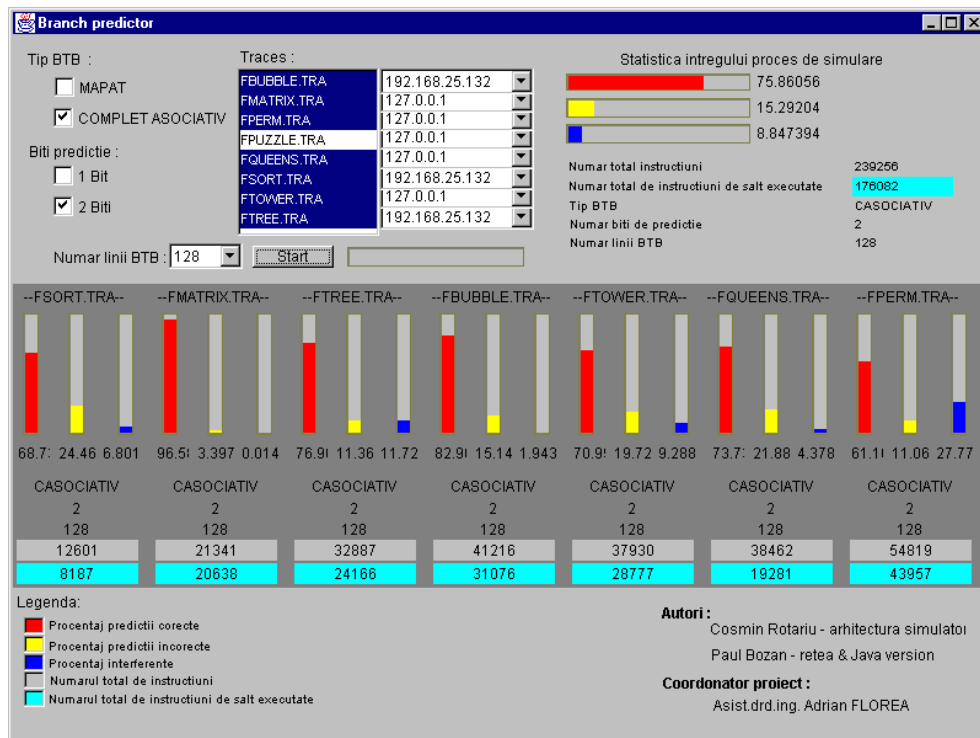
- ☐ se va alege configurația BTB-ului (locatii, biti de predicție, arhitectura)
- ☐ se vor alege fișierele *trace*
- ☐ se stabilește IP-ul fiecărui server pe care se va simula trace-ul selectat
- ☐ se va porni simularea la apăsarea butonului “Start”

În funcție de tipul simulării pot exista următoarele cazuri:

- ⇒ **Acuratetea predicției în funcție de arhitectura:** se vor face două simulări - pentru BTB mapat direct și pentru BTB complet asociativ, restul parametrilor rămânând constante.

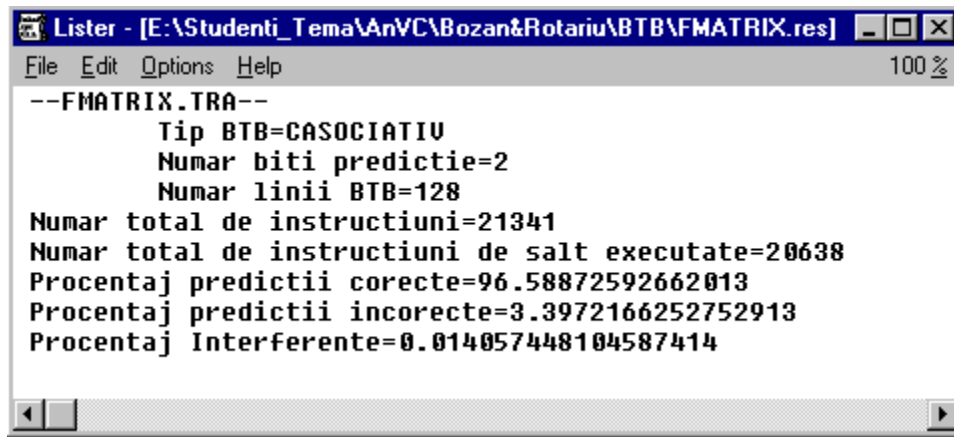
- ⇒ **Acuratetea predictiei în functie de *locatii*:** se vor face simulări pentru mai multe dimensiuni ale memoriei BTB, restul parametrilor rămânând constanti.
- ⇒ **Acuratetea predictiei în functie de *automat*:** se vor face două simulări pentru automat cu un bit de predictie și apoi cu doi biti de predictie.

Rezultatele simulării sunt disponibile atât sub forma grafică cât și text.  
Un rezultat sub forma grafică arată astfel:



**Figura 6.10.** Rezultatele simulării a 7 din 8 benchmark-uri Stanford

Rezultate în urma simulării pot fi vizualizate și în format text, la fiecare simulare creându-se un fișier cu extensia **\*.res**, având același nume cu cel al fișierului trace simulat.



```

--FMATRIX.TRA--
    Tip BTB=CASOCIATIV
    Numar biti predicție=2
    Numar linii BTB=128
    Numar total de instructiuni=21341
    Numar total de instructiuni de salt executate=20638
    Procentaj predicții corecte=96.58872592662013
    Procentaj predicții incorecte=3.3972166252752913
    Procentaj Interferente=0.014057448104587414

```

**Figura 6.11.** Rezultatele simulării disponibile în fișiere text

#### Descriere implementare software

Programul se bazează pe mai multe clase care descriu automatul de predicție, tabela BTB, funcționarea simulatorului, execuția multithreading. În continuare se vor prezenta clasele și rolul fiecăreia în cadrul aplicației.

#### **Implementarea automatului de predicție**

Automat:

```

public class Automat {
    int stare ;
    public Automat();
    public boolean Predicție(boolean biti);    // returneaza predicția pentru
                                                // un anumit automat
    public void UpdateAutomat(boolean salt, boolean biti); // modifica starea
                                                            automatului în
                                                            funcție de
                                                            parametrii primiti
    void SetStareAutomat(int stare);           // seteaza starea automatului la A
}

```

#### **Implementarea memoriei BTB**

##### **LinieBTB:**

// structura unei linii din BTB

```

public class LinieBTB {
    int FPc; // PC-ul instructiunii
    int FPcPrezis; //PC-ul Prezis
    Automat FPredictie = new Automat();// automatul de predictie
    // asociat
    int FLru; // folosit pentru evacuare // in cazul mem. asociative
};

/* structura BTB */
public class Btb {
    public Vector linii = new Vector();
    String arhitectura = "MAPAT";//"CASOCIATIV";// tipul arhitectura cache
    // (mapat sau complet
    // asociativ)
    String rezultat= "CORECT"; // rezultatul predictiei
    // CORECT -> predictie corecta
    // INCORECT -> predictie incorecta
    // INCORECTADR -> predictie incorecta
    // datorata modificarii adresei destinatie a
    // saltului
    int locatii = 0; // nr. de locatii din tabela (BTB)
    int index = 0; // indexul din BTB la care sa gasit PC-ul cautat
    boolean biti = false; // nr. de biti de predictie utilizati
    boolean InBtb = false; // indica daca PC-ul cautata s-a gasit in BTB -
    // hit
    public void SetDimensiune(int nr_linii); // seteaza dimensiune tablei BTB
    public void InitializareZero(); // initializeaza cu 0 toate adresele
    // destinatie si PC-ul salturilor
    public void SetBitiPredictie(boolean biti); // seteaza numarul de biti de
    // predictie folositi de automat:
    // False-1 bit respectiv True-2 biti
    public void InitializareAutomat(); // initializeaza starea automatului
    // de predictie într-o stare
    // implicita (DA Slab)
    public void InitBtb( int nr_linii, String arhitectura, boolean biti);
    // initializeaza structura BTB
    public boolean GetPredictie( int pc); // returneaza predictia pentru o
    // anumita instructiune
    public boolean CautareMapat(int pc); // cauta PC-ul primit ca
    // parametru în memoria mapata
    // direct

```





```

public void DecodificareInstructiune( String instr);
                                // decodifica instructiunea primita în
                                // PC, si stabileste adresa de salt
}

```

### Simulare:

Pentru a putea fi simulate mai multe fisiere în acelasi timp s-au folosit fire de executie. Astfel pentru fiecare fisier selectat, se va crea câte un fir de executie care va face simularea fisierului respectiv si salvarea rezultatelor. Clasa este derivata din *Threads*, o clasa specifica *JBUILDER*, care implementeaza firele de executie.

```

public class Simulare {
    Simulator sim = null;//new Simulator();// o clasa de tip Tsimulator care are
                                // metodele si datele specifice
                                // simularii

    String btbType;
    int nrBp ;
    int liniiBTB;
    public Simulare(String trace,String btbType,int nrBitiPredictie,int liniiBTB);
    public void SetSimulator(Simulator simulator);    // seteaza parametrii
                                                        // simulatorului
    public void Execute(java.io.PrintWriter out);    // functia executata de
                                                        // firul de executie
}

```

### Implementarea clasei de afisare a rezultatelor sub forma grafica si salvarea în mod text a acestora:

#### MainFrame

Pe forma principala apar diverse controale de tip *eticheta*, *checkbox*, *button*, *list*.

```

public class MainFrame extends Frame {
    java.awt.List TracesList = new java.awt.List();
    Label label1 = new Label();
    Label label2 = new Label();
    CheckboxGroup checkboxGroup1 = new CheckboxGroup();
    Checkbox cMapat = new Checkbox();
    Checkbox CA asociativ = new Checkbox();
    Label label3 = new Label();
}

```

```
Checkbox c1Bit = new Checkbox();
Checkbox c2Bit = new Checkbox();
Label label4 = new Label();
JProgressBar progress = new JProgressBar();
Button bStart = new Button();
String nrBiti = "1";
Label label6 = new Label();
Label label5 = new Label();
Choice tnrLinii = new Choice();
Choice choice1 = new Choice();
Choice choice2 = new Choice();
Choice choice3 = new Choice();
Choice choice4 = new Choice();
Choice choice5 = new Choice();
Choice choice6 = new Choice();
Choice choice7 = new Choice();
Choice choice8 = new Choice();
Panel panelPlot = new Panel();
BorderLayout borderLayout1 = new BorderLayout();

public MainFrame();
private void jbInit() throws Exception;
public void setProgress();
void cMapat_itemStateChanged(ItemEvent e);
void cAsociativ_itemStateChanged(ItemEvent e);
void c1Bit_itemStateChanged(ItemEvent e);
void c2Bit_itemStateChanged(ItemEvent e);
int bStart_actionPerformed(ActionEvent e);
void loadIps();
void loadTraces();
void this_windowClosing(WindowEvent e);
}
```

### 6.3.1.3. IMPLEMENTAREA RETELEI DE SIMULATOARE

#### Scurt memento teoretic privind modelul CLIENT-SERVER

Modelul architectural de referință pentru interconectarea sistemelor deschise (*open system interconnection* – OSI), elaborat și adoptat de *Organizația Internațională de Standarde* (ISO) are la bază 3 elemente:

⇒ procesele de aplicație, care realizează prelucrările de date;

- ⇒ sistemele de calcul care gazduiesc procesele de aplicatie si care sunt conectate printr-un mediu de comunicare;
- ⇒ conexiunile logice care permit un schimb de informatie uniform între procesele de aplicatie, indiferent de localizarea acestora în calculatoarele gazda.

La baza stabilirii nivelelor arhitecturale ale modelului ISO OSI au stat o serie de principii generale, cum ar fi:

- ⇒ crearea unui numar redus de nivele cu putine interactiuni între ele;
- ⇒ alegerea taieturilor dintre nivele în conformitate cu necesitatile de standardizare sau cu standardele deja existente;
- ⇒ colectarea functiilor înrudite în acelasi nivel;
- ⇒ crearea posibilitatii de modificare a functiilor unui nivel, fara afectarea celorlalte.

Cele 7 nivelele OSI sunt:

- ☐ **Aplicatie**
- ☐ **Prezentare**
- ☐ **Sesiune**
- ☐ **Transport**
- ☐ **Retea**
- ☐ **Legatura de date**
- ☐ **Fizic**

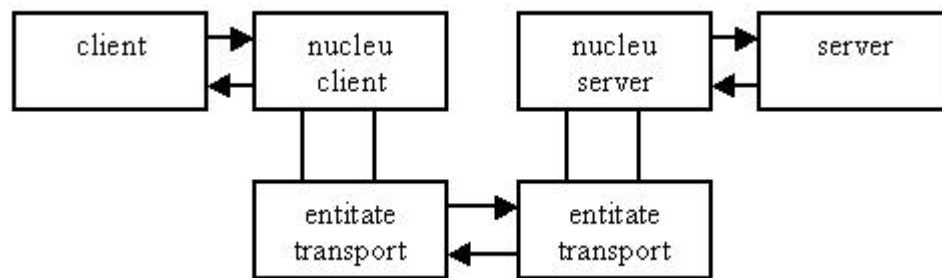
În conformitate cu modelul arhitectural OSI, rolul nivelului sesiune este acela de a permite utilizatorilor (care pot fi entitati ale nivelului prezentare sau procese de aplicatie) sa stabileasca conexiuni, numite sesiuni si sa transfere date într-o maniera ordonata. O sesiune poate fi folosita pentru accesul unui terminal la un nod îndepartat, pentru transfer de fisiere etc.

Serviciile nivelului sesiune si ale nivelelor superioare se refera mai putin la problemele de comunicare a datelor si mai mult la cele specifice aplicatiilor. În particular, nivelul sesiune realizeaza, printre altele, gestiunea dialogului, sincronizarea si gestiunea activitatilor, concepute ca servicii ce se adauga transportului corect al datelor, asigurat de nivele inferioare.

În multe retele în special în retelele locale, unele resurse sunt concentrate în anumite noduri dar sunt accesibile celorlalte noduri prin serviciile oferite de nivelele transport sau sesiune. Se stabileste o relatie asimetrica între **solicitantul serviciului** numit **client** si **furnizorul** acestuia, numit **server**. În aceasta **relatie** numita **client - server**, comunicarea ia forma unui perechi cerere – raspuns, fiind initiata întotdeauna de client si completata de server.

De cele mai multe ori faptul ca serviciul solicitat este realizat în alt nod, nu de cel în care este localizat clientul, este transparent pentru utilizator. Astfel un program poate folosi discul unui server prin comenzi similare utilizării unui disc local. Din acest punct de vedere relația dintre client și server apare ca apelul unui subprogram; clientul inițiază o acțiune și așteaptă primirea rezultatului. Diferența constă în aceea că subprogramul apelat se află la distanță.

Implementarea unui astfel de apel se poate face conform schemei prezentate în figura 6.12. Clientul apelează un subprogram, **nucleu client**, aflat în spațiul său de adresare. Acesta compune un mesaj cu parametrii apelului și-l transmite, prin serviciul de transport, unui nucleu **server**. Acesta apelează serverul și obține rezultatele dorite, pe care le transmite înapoi nucleului client. În fine, rezultatele sunt livrate clientului care le așteaptă.

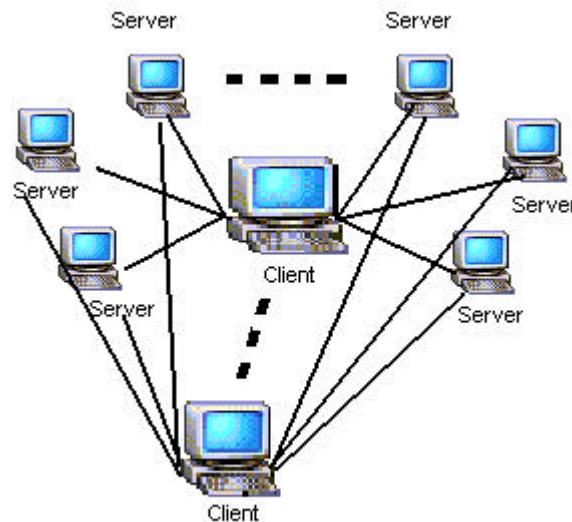


**Figura 6.12.** Implementarea unui apel în relația Client - Server

Reteaua de simulatoare este construită în jurul unei arhitecturi clasice **Client – Server**. Reteaua este compusă dintr-o serie de servere instalate pe calculatoarele din rețea și o aplicație client care comandă simulări serverelor din rețea.

Atât serverele cât și clientul sunt multi-thread ceea ce face ca rezultatele să fie disponibile în timp real și în plus un client poate comanda mai multe simulări unei singure stații iar simulările vor fi executate în paralel. Mai mult serverul este proiectat să suporte mai mulți clienți simultan.

O posibilă structură a rețelei este înfățișată în figura următoare:



**Figura 6.13.** Structura rețelei de simulatoare

### Atributiile Clientului

#### Client:

```
public class SendRequest extends Thread {
    String host = "";
    String trace = "";
    String btbSize = "";
    String nrBiti = "";
    String btbType = "";
    MainFrame progress;
    public SendRequest(String host, String trace, String nrBiti, String
                        btbType, String btbSize, MainFrame progress);
    public void run();
    public void StartSimulation();
}
```

### Atributiile Serverului

#### Server:

```
public class Server{
    public static final int          DEFAULT_PORT = 51966;
    public static final boolean REDIRECT_OUT = false;
    public static final boolean REDIRECT_ERR = true;
```

```

    private ServerSocket server = null;
    private boolean running = false;
    public Server();
    void run();
}

```

#### **Server\_Local:**

```

public class startLocalServer extends Thread {
    public startLocalServer();
    public void run();
}

```

#### **6.3.1.4. RESURSE HARDWARE MINIME NECESARE**

Având în vedere ca simulatorul este scris în întregime în limbajul de programare JAVA necesitând un JDK 1.3 calculatoarele care compun rețeaua de simulatoare trebuie să aibă următoarele resurse hardware minime:

- ☐ Frecvența procesor - cel puțin 133 MHz (recomandat mai mult de 266).
- ☐ Memorie RAM - 16 MB RAM (recomandat 32 iar ideal este să fie mai mare).
- ☐ Spațiu disponibil pe disc – 40 MB (este luat în calcul și spațiul necesar Mașinii Virtuale JAVA).
- ☐ Rezoluție monitor minimă 800x600.

Simulatorul a fost testat și rulează în condiții foarte bune pe calculatoare cu frecvență cuprinsă între 300 – 800 MHz și 128 RAM. Simulatorul fiind scris integral JAVA este portabil și rulează sub următoarele sisteme de operare:

- ☐ Windows 9x
- ☐ Windows NT
- ☐ Windows 2000
- ☐ Linux

**Obs:** Sub Linux JRE1.3 (Mașina Virtuală pentru platforma Windows ) trebuie înlocuită cu Mașina Virtuală specifică acestui sistem de operare (ex. Linux – JRE1.3 pentru Linux )

### 6.3.1.5. GHID DE INSTALARE SI CONFIGURARE

Atât clientul cât si serverul sunt “împachetate” în arhive auto-extractabile. Pasii necesari instalarii sunt urmatoarii:

1. Se copiaza fisierul dorit (arhivele) pe statia gazda.
2. Se executa fisierul si din caseta de dialog ce va fi afisata se alege directorul în care se doreste instalarea fisierelor.
3. Dupa extragerea fisierelor se editeaza fisierul “**Ips.txt**”, aflat în directorul radacina al simulatorului, cu IP-urile statiilor pe care vor fi instalate server-ele.
4. În directorul radacina al simulatorului se mai gasesc doua fisiere unul “**runClient.exe**” iar altul “**runServer.exe**” dintre care-l vom alege pe cel care ne intereseaza.

**Obs:** Pe statia pe care va fi rulat un client este rulat automat si un server. Nu mai rulati un al doilea server .

### 6.3.1.6. GHID DE UTILIZARE

Se verifica faptul ca toate server-ele ale caror IP-uri le-am setat în fisierul “IPs.txt” ruleaza, apoi, din fereastra clientului alegem configuratia dorita pentru simulare apoi alegem din lista “**Traces**” programele de test în format trace necesare simularii, iar pentru fiecare trace si statia pe care se doreste sa se execute simularea trace-ului. (**Obs:** 127.0.0.1 este IP-ul statiei locale).

Se apasa butonul “**Start**” si simularea începe, evolutia ei putând fi urmarita cu ajutorul control-ului **progressBar** din dreptul butonului “**Start**”.

Rezultatele simularii vor fi afisate atât în fereastra Clientului cât si în fisierele cu extensia “.res” din directorul radacina al aplicatiei.

### 6.3.1.7. NE-REALIZARI ACTUALE

1. Nu sunt tratate toate exceptiile care pot aparea în decursul unei simulari aceasta ducând la o comportare ciudata a retelei de simulare la aparitia unei exceptii. În versiunea actuala au fost “*prinse*” doar exceptii considerate la momentul respectiv ca fiind cele care pot aparea cu frecventa mai mare, celelalte fiind considerate ne-esentiale.
2. Nu exista posibilitatea efectuarii de catre utilizator a unor setari suplimentare si anume:
  - ⇒ cât timp sa astepte clientul dupa un rezultat daca statia întârzie sa-l dea. În versiunea actuala se asteapta un timp nelimitat.



⇒ setarea portului pe care asculta server-ul (asteapta raspuns). În versiunea actuala 51966.

3. Ca si posibilitati de dezvoltare a simulatorului amintim repararea "bug"-urilor cunoscute si eventual îmbunatatirea interfetei cu utilizatorul.

#### 6.3.1.8. PROBLEME PROPUSE SPRE REZOLVARE

Sa se reprezinte sub forma grafica functiile utilizând implicit automatul de predicție pe doi biti:

a)  $A_p = f(\text{tip\_arhitectura})$

b)  $A_p = f(\text{dimensiune\_tabela\_predictie})$

Sa se repete rezultatele utilizând automatul de predicție pe 1 bit definit de expresia: **ABAB:2**.

c) Reprezentati  $A_p = f(\text{nr\_biti\_automat\_predictie})$  considerând parametrul **dimensiune\\_tabela\\_predictie** - valoarea optima rezultata în urma simulării efectuată la b) si arhitectura - optima de la a).

### 6.3.2. PREDICTOR CORELAT DE SALTURI, ADAPTIV PE DOUA NIVELE - GAG

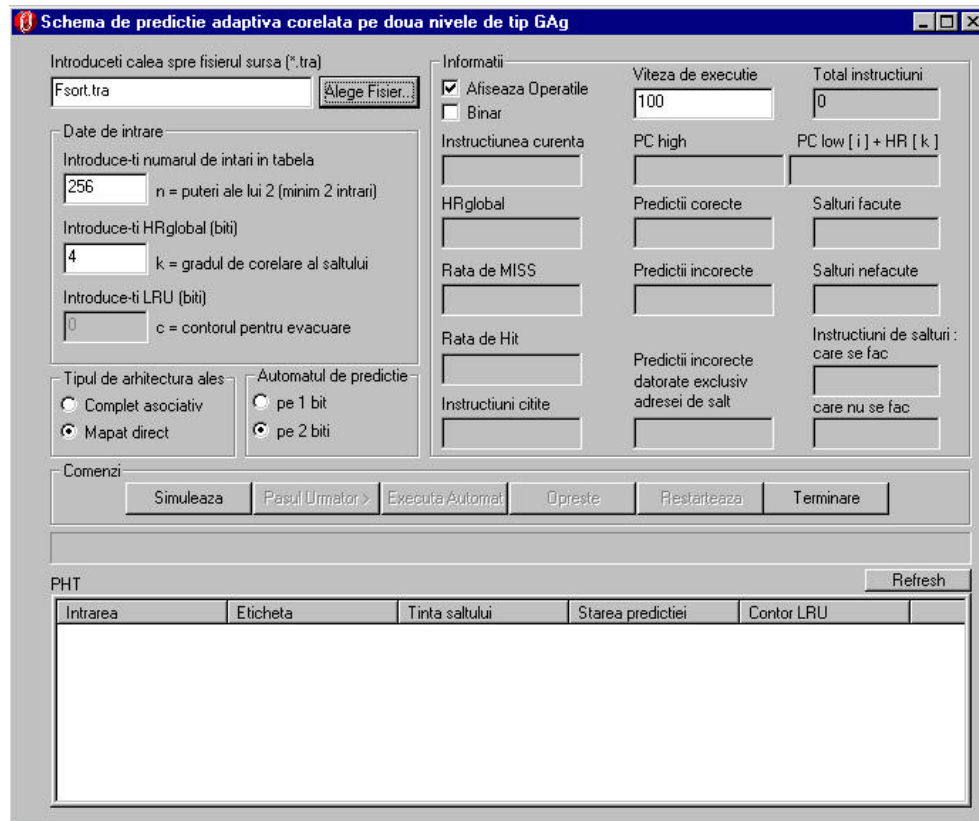
#### 6.3.2.1. PREZENTARE PROGRAM DE SIMULARE

##### Interfata cu utilizatorul:

Programul realizat reprezintă un simulator al schemei de predicție GAg. Implementarea software s-a făcut în mediul Developer Studio din Visual C++ versiunea 6.0, care permite extinderea actualei variante de simulator (generarea altor scheme de predicție adaptivă corelate pe două nivele - GAp, PAg, PAp, utilizarea controalelor ActiveX și COM pentru prelucrarea și redarea sub formă grafică a rezultatelor simulării). Din acest motiv, portabilitatea actualei versiuni de simulator se mărginește doar la sistemul de operare Windows cu variantele 9x, NT, Me, 2000. Metodologia de simulare este *trace driven* efectuată pe benchmark-urile Stanford.

Vor fi prezentate în continuare opțiunile care se pot alege în cadrul programului. Privind din punct de vedere al utilizatorului interfata

simulatorului (vezi figura de mai jos) este extrem de atractiva si intuitiva, bazata pe ferestre de dialog, butoane de comanda, controale de editare statica / dinamica, casete de tip lista etc.



**Figura 6.14.** Simulatorul schemei de predictie GAg - instantaneu

Se poate observa ca interfața software a simulatorului este formata din mai multe grupuri fiecare ocupându-se cu anumite parti necesare simulatorului.

O prima optiune ar fi “**Introduceti calea spre fisierul sursa (\*.tra)**”. Cu ajutorul acesteia se poate alege benchmark-ul asupra caruia se vor face testele (fisierul sursa). În cazul în care nu se stie o locatie exacta catre fisier se poate apasa butonul aflat în dreapta controlului de editare cu inscriptia “**Alege fisier...**” deschizându-se astfel un browser de navigare si selectie în care vor aparea în special fisierele cu extensia “**.tra**”.

Sub acest control de editare se observa grupul “**Date de intrare**”, grup ce contine zone de editare ce trebuie completate cu parametrii de intrare necesari simulatorului.

“**Introduceti numarul de intrari în tabela**” este controlul de editare prin intermediul caruia se introduce de la tastatura dimensiunea tabelii **PHT**. Dimensiunea trebuie sa fie minim 2 si maxim 65536. De amintit ca orice alta valoare trebuie sa fie aleasa ca si putere a lui 2 (2, 4, 8, 16, ..., 256, ..., 2048, ..., etc.), din motive de implementare software dar si hardware.

Prin intermediul controlului de editare “**Introduceti HR global (biti)**”, se va introduce numarul de biti al registrului de istorie globala ( $k = \text{gradul de corelare al saltului}$ ). El poate fi reprezentat minim pe un bit si maxim cu un bit mai mic decât puterea numarului de intrari în **PHT** pentru a putea fi concatenat cu o parte din **PC** (fie ea si un bit) în momentul accesarii tabelii de predicție.

De retinut ca aceasta optiune poate fi completata doar în cazul alegerii unui tip de arhitectura “**mapat direct**”, deoarece în cazul “**complet asociativ**”, eliminarea înregistrărilor din tabela se rezolva cu ajutorul contorului de tip “**LRU**”(Least Recently Used).

“**Introduceti LRU (biti)**”, reprezinta numarul de biti pe care poate fi codificata valoarea maxima a contorului pentru evacuarea locatiei corespunzătoare din tabela. Aceasta optiune este activata doar în cazul alegerii tipului de arhitectura “**complet asociativ**”, deoarece în cazul mapării directe se cunoaste locatia care va fi evacuată. Contorul aferent fiecărei înregistrări din tabela **PHT** este initializat la început cu valoarea maxima. Spre exemplu, pentru un **LRU** pe 8 biti valoarea initiala a contorului va fi 255. Contorul fiecărei înregistrări se va decrementa la fiecare acces în tabela iar locatiei accesate i se va seta contorul la valoarea maxima. Astfel, se va sti în permanenta care locatie a fost cel mai de demult ne-accesata din tabela, aceasta putând fi înlocuită când intrările în tabela vor fi complet ocupate.

Urmatorul grup din interfata este “**Tipul de arhitectura ales**”, care contine doua butoane radio, putându-se astfel selecta doar unul dintre ele si anume “**Complet asociativ**” sau “**Mapat direct**”, alegându-se astfel modelul de arhitectura dorit pentru simularea ce va urma. Implicit, se simuleaza o schema de predicție mapata direct.

Alt grup de butoane specifice interfetei cu utilizatorul îl reprezinta cel privitor la “**Automatul de predicție**”. Acesta contine tot doua butoane radio, din care se poate alege doar unul: “**pe 1 bit**”, “**pe 2 biti**”. Automatul de predicție este descris printr-un sir de caractere cu un format mai special, ce prezinta atât numarul de stari, tranzitiile între stari cât si predictia aferenta fiecărei stari. Ca si exemplu pot fi folosite automatele urmatoare (**ABAB:2** - pe un bit si automatul **BCBAADCD:12** - pe 2 biti, descrise si în capitolul 6.3.1.). Suplimentar, în cazul automatului de predicție pe 2 biti, reprezentarea în tabela **PHT** va fi astfel:

00 – NU\_Tare

01 – NU\_Slab

10 – DA\_Slab

11 – DA\_Tare

<b>Intrarea</b>	– este indexul intrării în tabela PHT
<b>Eticheta</b>	– reprezintă <i>tagul</i> (PCHigh aferent instrucțiunii în cazul tipului de arhitectura cu mapare directă și PC-ul instrucțiunii în cazul complet asociativ)
<b>Tinta saltului</b>	– este PCnext (adresa următoarei instrucțiuni din program)
<b>Starea predicției</b>	– constituie starea predicției respectivei instrucțiuni, generată de automatul de predicție
<b>Contor LRU</b>	– este valoarea contorului LRU, pentru instrucțiunea de salt supusă analizei

Tabelul 6.4.

#### Semnificația câmpurilor tabelii “PHT”

Grupul “Comenzi” conține 6 butoane din care inițial sunt validate doar două “Simulează” și “Terminare” și simulează un mediu de depanare (execuție pas cu pas) la nivel de instrucțiune.

“Simulează” – este practic butonul la a cărui apăsare începe simularea fișierului de test ales.

“Pasul următor >” – este butonul care determină trecerea simulatorului la următorul pas, adică procesarea unei noi instrucțiuni de salt din fișier.

“Execută automat” – permite execuția continuă a simulării în cazul în care se dorește mai mult rezultatele finale decât urmărirea și înțelegerea algoritmului pas cu pas.

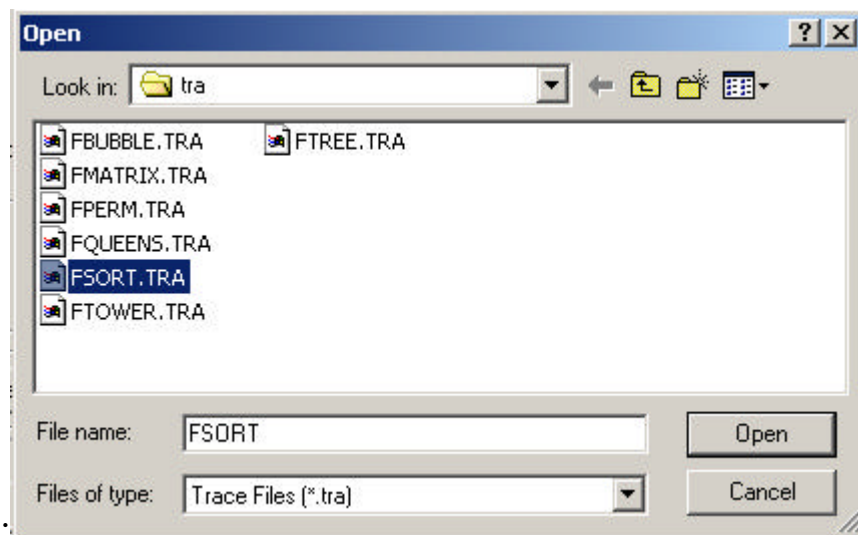
“Oprește” – este butonul cu ajutorul căruia se poate opri execuția automată la un moment dat dacă ne interesează o anumită parte din fișier pentru a o urmări pas cu pas în continuare.

“Restartează” – este butonul cu care putem relua din nou simularea trace-ului curent sau simularea altui fișier care poate fi încărcat cu ajutorul butonului “Alege fișier ...” explicat anterior. De amintit că, apăsarea acestui buton nu salvează datele simulării până la acel moment.

“Terminare” – este butonul la a cărui apăsare se închide aplicația (simulatorul).

### 6.3.2.2. GHID DE UTILIZARE

Cu aceste lucruri explicate se poate trece la simularea fisierului trace (“*.tra*”) și fie acesta spre exemplu “*fsort.tra*”. Fisierul îl putem cauta cu ajutorul “browser-ului” care apare după apăsarea butonului “*Alege fisier ...*”. Odată selectat programul de test apasam butonul “*Open*” pentru încărcarea acestuia (deschiderea în mod citire de către simulator).



**Figura 6.15.** Selectia benchmark-urilor pentru simulare

Se apasă butonul “*Simuleaza*” pentru începerea simulării și afișarea informațiilor în tabela PHT. De observat că, orice eroare care există în datele completate în controalele de editare puse la dispoziție de simulator va fi anunțată explicit cu un mesaj corespunzător care va specifica clar eroarea și unde se află aceasta, după care programul ne va poziționa în zona de editare respectivă, pentru reintroducerea corectă a datelor.

Pentru acest exemplu de simulare datele vor rămâne nemodificate, așa cum au fost setate implicit de către simulator și anume: numărul de intrări este 256, numărul de biți ai registrului de istorie globală este 4, tipul de arhitectură ales este “*mapat direct*” iar automatul de predicție selectat este pe 2 biți. În continuare se apasă butonul “*Simuleaza*” care va avea ca efect initializarea tabelului *PHT* și ale altor câmpuri din grupul “*Informatii*” care vor fi explicate mai jos, după figura următoare.

**Schema de predicție adaptivă corelată pe două nivele de tip GAg**

Introduceți calea spre fișierul sursă (\*.tra)

Date de intrare  
 Introduceți numărul de intrări în tabela  
 n = puteri ale lui 2 (minim 2 intrări)  
 Introduceți HRglobal (biti)  
 k = gradul de corelare al saltului  
 Introduceți LRU (biti)  
 c = contorul pentru evacuare

Tipul de arhitectura ales  
☐ Complet asociativ  
☒ Mapă directă

Automatul de predicție  
☐ pe 1 bit  
☒ pe 2 biti

Comenzi

Informații  
☒ Afisează Operatiile  
☐ Binar

Viteza de execuție

Total instrucțiuni

Instrucțiunea curentă

PC high

PC low [ i ] + HR [ k ]

HRglobal

Predictii corecte

Salturi facute

Rata de MISS

Predictii incorecte

Salturi nefacute

Rata de Hit

Predictii incorecte datorate exclusiv adresei de salt

Instrucțiuni de salturi :  
 care se fac  
  
 care nu se fac

Instrucțiuni citite

Refresh

Intrarea	Eticheta	Tinta saltului	Starea predicției	Contor LRU
25	-1	0	DA_Slab = 2 => 1	0
26	-1	0	DA_Slab = 2 => 1	0
27	-1	0	DA_Slab = 2 => 1	0
28	-1	0	DA_Slab = 2 => 1	0
29	-1	0	DA_Slab = 2 => 1	0
30	-1	0	DA_Slab = 2 => 1	0
31	-1	0	DA_Slab = 2 => 1	0
32	-1	0	DA_Slab = 2 => 1	0
33	-1	0	DA_Slab = 2 => 1	0

**Figura 6.16.** Accesarea primei locații din PHT – după startarea simulării

De altfel, există posibilitatea de acces și la butoanele “*Pasul Urmator* >”, “*Execută automat*” și “*Restartează*”, pe când butonul “*Simulează*” nu se mai poate accesa decât după restartare.

Se observă în tabela *PHT* initializarea câmpului “**Eticheta**” cu **-1**, a “destinației instrucțiunii de salt” și a “**contorului LRU**” - care în acest caz (tip de arhitectura “cu mapare directă”) nu este folosit - cu **0**, și a câmpului “**Starea predicției**” cu **DA\_Slab = 10<sub>2</sub> = 2** **P predicție Taken**, considerându-se ca toate salturile se introduc în *PHT* cu prima ocazie în care acestea se fac. Deasupra tabelului *PHT*, există un buton numit “*Refresh*”, care face actualizarea tabelului; în acest caz dacă se apasă pe el, la intrarea 32 din tabela vor fi înlocuite valorile corespunzătoare, putându-se urmări în continuare, pas cu pas algoritmul simulatorului.

În grupul “*Informații*”, există o serie de controale de editare care se referă în mod special la datele de ieșire corespunzătoare calculelor pentru instrucțiunea în cauză. În afara acestora mai există două butoane de tip

*checkbox* “**Afiseaza operatiile**” si “**Binar**”, care pot fi selectate sau deselectate la cerere.

“**Afiseaza operatiile**” – selectat implicit, afiseaza operatiile executate în timpul simulării si are efect doar în cazul în care ne aflam în executie automata (s-a apasat butonul “**Executa automat**” din grupul “**Comenzi**”). Dacă acesta este selectat pot fi identificate operatiile ce se executa automat. Acestea din urma se executa cu o viteza selectata în zona de editare “**Viteza de executie**” care implicit are valoarea **100ms**. Viteza reprezinta practic diferenta de timp între 2 instructiuni ce vor fi aduse în executiei (timpul între doua afisari succesive a informatiilor din fereastra utilizator).

Dacă în schimb se deselecteaza optiunea “**Afiseaza operatiile**”, atunci simulatorul nu va mai afisa nici o informatie din tabela **PHT** dar nici în grupul “**Informatii**”. Acest lucru este benefic dacă se urmareste o viteza superioara de executie. În acest caz simulatorul nu va mai tine cont de valoarea aflata în controlul de editare “**Viteza de executie**”, si va executa simularea la viteza maxim posibila.

Al doilea buton de tip *checkbox* “**Binar**”, transforma toate afisarile din zecimal în binar pentru o mai buna observare a împartirii tag-urilor (HRglobal, PClow etc), la nivel de bit. Restul controalelor de editare ramase sunt specifice simulării benchmark-ului selectat:

“**Total instructiuni**” – contine numarul de instructiuni existente în fisierul sursa.

“**Instructiunea curenta**” – prezinta chiar instructiunea aflata în simulare.

“**PChigh**” – este partea superioara a PC-ului instructiunii curente determinata pe baza dimensiunii lui HR si a numarului de intrari în tabela PHT. Astfel, se poate calcula numarul de biti necesari pentru PClow (în acest caz tabela având 256 de intrari - deci poate fi accesata pe 8 biti, si cum 4 biti sunt folositi din registrul HR înseamna ca PClow va fi tot pe 4 biti, ceea ce face ca partea mai semnificativa (PChigh) a oricarui PC cu valori între 0 si 15 va fi tot timpul 0. PChigh reprezinta chiar *tag-ul emis* care se compara în tabela PHT la adresa data de PClow + HRglobal cu *tag-ul de comparat* (semnul '+' are semnificatia de concatenare, desi uneori poate fi înlocuit cu o functie de dispersie - vezi simulatorul **bpred** al setului de instrumente SimpleScalar 3.0 – capitolul 8). În cazul de fata PChigh este 32. În cazul tipului de arhitectura

”*complet asociativ*” PChigh va fi substituit de întreg PC-ul instrucțiunii curente.

“*PClow[i] + HR[k]*” – este controlul de editare ce contine valoarea concatenata a celor doua componente care se pot vedea mai clar trecând afisarea în binar de la butonul de tip check “*Binar*”. De amintit ca, în cazul tipului de arhitectura “*complet asociativ*” acest câmp nu va mai fi folosit deoarece tot PC-ul se va folosi pentru cautarea tag-ului în tabela.

“*HRglobal*” – reprezinta comportamentul ultimelor instructiuni de salt (T/NT). Valoarea registrului HR se va actualiza astfel: în urma executiei fiecarei instructiuni de salt se face o deplasare cu o pozitie la stânga. Daca saltul s-a facut (*taken*) atunci bitul cel mai putin semnificativ va fi setat la valoarea 1. În cazul în care saltul nu se face (*not taken*) bitul respectiv va lua valoarea 0.

Câmpurile “*Predictii corecte*”, “*Predictii incorecte*” si “*Predictii incorecte datorate exclusiv adresei de salt*” contorizeaza numarul de predictii facute corect, numarul de predictii gresite, respectiv numarul de predictii gresite când desi tag-ul si predictia au coincis, adresa de salt memorata în tabela PHT nu a corespuns cu cea a instructiunii în cauza (target-ul a diferit: salturi indirecte, reveniri din subrutina).

Câmpurile “*Salturi facute*” si “*Salturi nefacute*” contorizeaza numarul de salturi care s-au facut, respectiv al celor care nu s-au facut, aceste stari ale salturilor fiind date de starea automatului de predictie în momentul determinarii predictiei.

Câmpurile “*Rata de HIT*” respectiv “*Rata de MISS*” exprima numarul de accese cu hit respectiv miss în tabela PHT (cazurile când tag-ul instructiunii respective a fost sau nu gasit în tabela - hit / miss pe directie de salt).

Câmpurile “*Instructiuni de salt:*” “*care se fac*”, respectiv “*care nu se fac*” contorizeaza tipurile de instructiuni de salt citite din trace de-a lungul simularii. De exemplu instructiuni de salt care se fac ar fi instructiuni cu mnemonica “*BR, BM, BS, BT, ...*” iar cele care nu se fac sunt “*NT*”.

Un ultim câmp utilizat pentru controlul simularii ar fi “*Instructiuni citite*”, ce contine numarul instructiunii curente extrase din fisier. Simularea se va termina în momentul în care acest contor va fi egal cu cel din controlul de



editare “**Total instructiuni**”. Procentul de simulare efectuat până la un moment dat este ilustrat cu ajutorul “*progress bar*-ului” situat deasupra tabelului **PHT**.

Execuția pas cu pas a simulatorului pune în evidență toate modificările survenite în tabela PHT și în celelalte controale de editare. Pentru exemplificare, se efectuează primii 12 pași de simulare ajungând astfel la următoarea stare:

**Schema de predicție adaptivă corelată pe două nivele de tip GAg**

Introduceți calea spre fișierul sursă (\*.tra)

**Date de intrare**  
 Introduceți numărul de intrări în tabela  
 n = puteri ale lui 2 (minim 2 intrări)  
 Introduceți HRglobal (biti)  
 k = gradul de corelare al saltului  
 Introduceți LRU (biti)  
 c = contorul pentru evacuare

**Tipul de arhitectura ales:**  
☐ Complet asociativ  
☒ Mapat direct

**Automatul de predicție:**  
☐ pe 1 bit  
☒ pe 2 biti

**Informații**  
☒ Afisează Operatiile  
☐ Binar

Viteza de execuție:   
 Total instrucțiuni:

Instrucțiunea curentă:   
 PC high:   
 PC low [i] + HR [k]:   
 HRglobal:   
 Predicții corecte:   
 Salturi facute:   
 Rata de MISS:   
 Predicții incorecte:   
 Salturi nefacute:   
 Rata de Hit:   
 Predicții incorecte datorate exclusiv adresei de salt:   
 Instrucțiuni citite:   
 Instrucțiuni de salturi :  
 care se fac:   
 care nu se fac:

**Comenzi**

**PHT**

Intrarea	Eticheta	Tinta saltului	Starea predicției	Contor LRU
31	1	28	DA_Slab = 2 => 1	0
32	0	151	DA_Slab = 2 => 1	0
33	-1	0	DA_Slab = 2 => 1	0
34	-1	0	DA_Slab = 2 => 1	0
35	-1	0	DA_Slab = 2 => 1	0
36	-1	0	DA_Slab = 2 => 1	0
37	-1	0	DA_Slab = 2 => 1	0
38	-1	0	DA_Slab = 2 => 1	0
39	-1	0	DA_Slab = 2 => 1	0

**Figura 6.17.** Modificările survenite în tabela PHT după primii 12 pași de simulare

Se observă că instrucțiunea curentă “**BM 17 28**” are **PChigh** = 1, **PClow[i] + HR[k]** = 31 și destinația saltului este 28. Astfel, la intrarea 31 în tabela PHT, se verifică tag-ul instrucțiunii memorate acolo cu tag-ul emis (dat de PChigh) și se observă egalitatea lor. De asemenea, se determină că instrucțiunea curentă este una de salt care se face, ceea ce este în concordanță cu automatul de predicție aferent intrării respective, care este setat pe **DA\_Slab**. Atunci mai rămâne de verificat doar tinta saltului aflată

în tabela cu cea reala rezultata din trace, observându-se ca si acestea sunt egale. Astfel, daca se va apasa butonul **Refresh** simulatorul va modifica în tabela la aceasta intrare doar starea automatului de predictie pe care îl va trece din DA\_Slab în **DA\_Tare**, ca în figura de mai jos:

**Schema de predictie adaptiva corelata pe doua nivele de tip GAg**

Introduceti calea spre fisierul sursa (\*.tra)

**Date de intrare**  
 Introduce-ti numarul de intrari in tabela  
 n = puteri ale lui 2 (minim 2 intrari)  
 Introduce-ti HRglobal (biti)  
 k = gradul de corelare al saltului  
 Introduce-ti LRU (biti)  
 c = contorul pentru evacuare

**Tipul de arhitectura ales**  
☒ Complet asociativ  
☐ Mapat direct

**Automatul de predictie**  
☐ pe 1 bit  
☒ pe 2 biti

**Comenzi**

**Informatii**  
☒ Afiseaza Operatile  
☐ Binar

Viteza de executie:   
 Total instructiuni:

Instructiunea curenta:   
 HRglobal:   
 Rata de MISS:   
 Rata de Hit:   
 Instructiuni citite:

PC high:   
 PC low [i] + HR [k]:   
 Predictii corecte:   
 Predictii incorecte:   
 Predictii incorecte datorate exclusiv adresei de salt:

Salturi facute:   
 Salturi nefacute:   
 Instructiuni de salturi :  
 care se fac:   
 care nu se fac:

**PHT**

Intrarea	Eticheta	Tinta saltului	Starea predictiei	Contor LRU
31	1	28	DA_Tare = 3 => 1	0
32	0	151	DA_Slab = 2 => 1	0
33	-1	0	DA_Slab = 2 => 1	0
34	-1	0	DA_Slab = 2 => 1	0
35	-1	0	DA_Slab = 2 => 1	0
36	-1	0	DA_Slab = 2 => 1	0
37	-1	0	DA_Slab = 2 => 1	0
38	-1	0	DA_Slab = 2 => 1	0
39	-1	0	DA_Slab = 2 => 1	0

**Figura 6.18.** Actualizarea automatului de predictie din tabela PHT si a celorlalte controale de editare în cazul unei predictii corecte

Astfel se poate urmări functionarea simulatorului în fiecare moment de timp. Analog se poate trece simularea pe un tip de structura **“complet asociativ”**, cu un automat de predictie pe un bit sau pe mai multi si cu dimensiuni diferite ale tabelor PHT, pentru gasirea unei configuratii optime pentru un trace-ul în cauza , sau pentru comparatii ce se pot realiza între fisier diferite sau pe acelasi fisier în configuratii diferite. Acest lucru poate fi facut deoarece simulatorul salveaza la sfârșitul simulării rezultatele obtinute într-un fisier al carui nume este compus din datele de intrare alese, tocmai pentru a se putea salva cât mai multe simulari diferite pe acelasi fisier.

În cazul de față dacă se apasă butonul “*Executa automat*” la sfârșitul simulării se obțin rezultatele afișate mai jos. Fisierul cu rezultate se va deschide automat la terminarea simulării pentru ilustrarea acestora.

```

FSORT_256intrari_4bitHR_MD_2bitAP - Notepad
File Edit Format Help
-----
Rezultatele simulării fișierului FSORT.TRA
-----
Datele de intrare
-----
Numarul de intrari în tabela (PHT) = 256
Numarul de biti folositi pentru registrul de istorie globala (HR) = 4
Tipul de arhitectura ales = MAPAT DIRECT
Numarul de biti ales pentru automatul de predicție (AP) = 2 biti
-----
Rezultatele
-----
Numarul total de instrucțiuni de salt procesate = 12601
Numarul total de instrucțiuni de salt,
care se fac (Ex: EM, BR, BS, BT, etc.) = 8187 (64.971%)
Numarul total de instrucțiuni de salt,
care nu se fac (Ex: NT) = 4414 (35.028%)
Numarul salturilor facute = 7691 (73.507%)
Numarul salturilor nefacute = 2772 (26.493%)
Numarul salturilor predictiionate corect = 7600 (72.637%)
Numarul salturilor predictiionate incorect = 2697 (25.777%)

```

**Figura 6.19.** Exprimarea rezultatelor simulării în format text

### 6.3.2.3. DEZVOLTARI ULTERIOARE

1. Îmbunătățirea interfeței cu utilizatorul în vederea extinderii procesului de simulare a mai multor benchmark-uri simultan (aplicație multi-document).
2. Reprezentarea grafică a rezultatelor simulării cu ajutorul controalelor de tip ActiveX.
3. Realizarea unui sistem de ajutor “*Help on line*” care să faciliteze o înțelegere mai bună a termenilor specifici arhitecturii calculatoarelor, a modului de depanare pas cu pas, explicarea rezultatelor simulării.

**6.3.2.4. PROBLEME PROPUSE SPRE REZOLVARE**

Sa se reprezinte sub forma grafica functiile utilizând implicit automatul de predictie pe doi biti:

1.  **$A_p = f(\text{tip\_arhitectura})$**
2. Analizati influenta gradului de localizare al saltului asupra acurateti de predictie:  **$A_p = f(i)$**  unde  **$i$**  = dimensiunea  **$PC_{low}$** .
3. Stabiliti influenta contextului în care se situeaza saltul în program:  **$A_p = f(HR_{global})$** .
4. Reprezentati  **$A_p = f(nr\_biti\_automat\_predictie)$**  considerând parametrii optimi ( $PC_{low}$ ,  $HR_{global}$ ) rezultati în urma simulării efectuate la 1), 2) si 3).
5. Cuantificati câstigul introdus printr-o implementare ideala, în care nu ar exista adresa tinta modificata în tabele.

## 7. PREDICTOR NEURONAL DE RAMIFICATII PROGRAM

---

### 7.1. SCOPUL LUCRARIII

Scopul lucrării îl constituie introducerea în domeniul rețelelor neuronale și aplicabilitatea acestora în predicția salturilor. De asemenea, se urmărește descrierea și utilizarea unui simulator parametrizabil pentru o arhitectura superscalară având înglobat un *predictor neuronal de ramificatii program* care să înlocuiască clasicele scheme de predicție (BTB, corelate pe două nivele - GAg, PAg, GAp etc.), destul de limitate din punct de vedere al acuratății de predicție.

### 7.2. INTRODUCERE ÎN REȚELE NEURALE ȘI ALGORITMI GENETICI

#### 7.2.1. REȚELE NEURALE. GENERALITĂȚI.

Din punct de vedere al scopului, domeniul rețelelor neuronale, se încadrează în domeniul mai mare al recunoașterii formelor, iar acesta în inteligența artificială prin necesitatea **învățării**, iar din punct de vedere al metodei aplicate, se încadrează în cel al proceselor distribuite paralel (PDP).

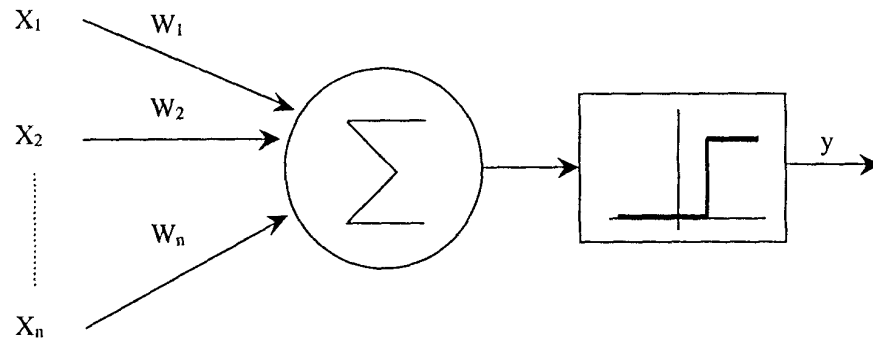
Rețelele neuronale încearcă să simuleze structura neurofiziologică a creierului uman. Creierul este alcătuit dintr-o multitudine de neuroni interconectați. Fiecare neuron primește informații de la alți neuroni prin intermediul dendritelor și transmite la rândul său un semnal prin intermediul

axonului. Acest model poate fi implementat pe calculator, considerând legăturile dintre neuroni numere (ponderi) care se modifică în funcție de cât de des este activată legătura.

O rețea neurală este un sistem de procesare a informației format dintr-o mulțime de neuroni (puternici) interconectați. Conceptul de rețele neurale artificiale a fost inspirat de rețelele neurale biologice. Neuronii biologici sunt considerați a fi constituanții structurali ai creierului care deși sunt mult mai înceti decât o poartă logică, implementată în siliciu, realizează inferențe și gândesc mult mai rapid și, în orice caz, mai profund și mai nuanțat decât cel mai bun calculator actual. Creierul compensează operațiile relativ încete (operații luate individual) printr-un număr imens de neuroni interconectați, prin paralelism masiv, reușind să se adapteze mediului înconjurător, să „mânuiască” informații vagi, imprecise și probabilistice, să generalizeze de la situații și exemple cunoscute la situații necunoscute, robuste, toleranță la erori. Rețele neurale artificiale încearcă să imite o parte a caracteristicilor descrise mai sus ale rețelelor neurale biologice și constă dintr-o mulțime de neuroni artificiali interconectați, de mici procesoare care execută doar operații de bază.

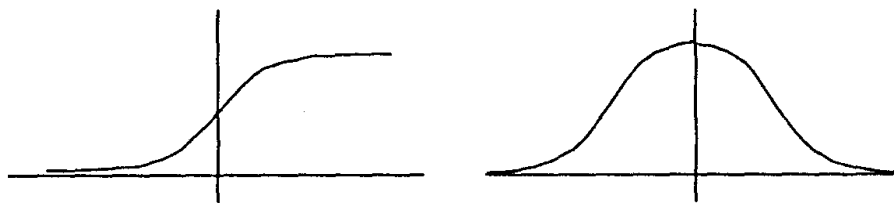
Primul model de rețea neuronală a apărut în 1958, când Frank Rosenblatt publică prima sa lucrare despre *perceptron*, denumire dată rețelei sale neuronale [Burj01]. Lumea oamenilor de știință și nu numai, a ramas fascinată de posibilitățile pe care le ofereau rețelele neuronale, care puteau fi folosite la formelor. Dar în 1969, Marvin Minsky și S. Papert au analizat posibilitățile de învățare ale perceptronului și au ajuns la concluzii destul de sceptice, dovedind imposibilitatea rezolvării de către perceptronul cu un singur strat a unor probleme simple, cum ar fi învățarea funcției XOR (funcție care nu este liniar separabilă). Ca urmare, interesul pentru acest domeniu de cercetare a scăzut dramatic. Relansarea interesului oamenilor de știință s-a produs în 1986, odată cu apariția unor modele și tehnici noi de învățare supervizată (*metodele propagării înapoi*). În fapt, paternitatea metodei aparține lui Paul Werbos care a prezentat-o în premieră în teza sa de doctorat, la Universitatea din Berkeley, în 1974.

Modelul matematic al unui neuron artificial propus de McCulloch și Pitts calculează o sumă ponderată de  $n$  semnale de intrare și generează o singură ieșire de 1 dacă această sumă este mai mare decât o anumită valoare (numită *prag*) și 0 dacă este mai mică [Sbe00].



**Figura 7.1.** Modelul matematic al unui neuron artificial

Una dintre cele mai evidente îmbunătățiri care pot fi aduse modelului McCulloch-Pitts, care de altfel reprezintă o simplificare grosolană a neuronului biologic, este folosirea unei *funcții de activare* diferite de funcția treaptă cum ar fi *funcția sigmoidă* sau *funcția Gaussiană*.



**Figura 7.2.** Doua funcții de activare: sigmoidă și Gaussiană

Funcția sigmoidă este de departe cea mai folosită în rețelele neurale. Este strict crescătoare și asimptotică având expresia matematică  $g(x) = 1/(1 + \exp(-\beta x))$ . Prin folosirea unei funcții de activare neliniare pot fi rezolvate un număr mai mare de probleme decât cu funcții liniare iar neuronul artificial este adus mai aproape de cel biologic, care este analogic și nu binar.

**Caracterizarea unei rețele** neurale poate fi făcută după:

- **Arhitectura.** Rețelele neurale artificiale pot fi văzute ca și grafuri orientate în care neuronii artificiali sunt noduri iar muchiile sunt conexiuni între ieșirile neuronilor și intrările acestora. Astfel există două categorii de rețele:
  - ☐ **rețele feed-forward**, în a căror grafuri nu există bucle; (cu reglare înainte; cu reacție pozitivă) unde fluxul de date de la unitățile de intrare la cele de ieșire se desfășoară strict înainte. Procesarea datelor poate fi extinsă pe mai multe straturi dar, fără a fi prezente conexiuni

înapoi (feed-back) de la iesirile unitatilor la intrarile celor din acelasi strat sau din straturile anterioare. Într-una din cele mai cunoscute familii de retele feed-forward numita *multilayer perceptron*, neuronii sunt organizati pe nivele care au conexiuni unidirectionale între ele. Diferite configuratii determina diferite comportamente si capabilitati ale retelei.

- **retele recurente** (feedback) în care apar bucle datorita conexiunii înapoi. Spre deosebire de retele feed-forward, aici sunt importante proprietatile dinamice ale retelei. În unele cazuri, valorile activarii unitatilor trec printr-un proces de relaxare astfel încât, reseaua va evolua într-o stare stabila în care aceste valori nu se vor mai modifica. Exemple de retele de acest tip sunt *retelele Hopfield* si *retelele Kohonen*.

- **Regulile de actualizare a ponderilor** (*algoritmul de învățare*). Abilitatea de a învăța este o caracteristica fundamentala a inteligentei. Nu exista o definitie precisa a procesului de învățare însă acest proces poate fi vazut în cazul retelelor neurale artificiale ca si o problema de adaptare a ponderilor astfel încât reseaua sa poata rezolva o anumita problema. Sunt trei mari paradigme de învățare: *supervizata*, *nesupervizata* si *hibrida*. În învățarea supervizata, sau învățarea cu un „profesor”, retelei i se furnizeaza un raspuns pentru fiecare tip de intrare. Ponderile se vor modifica astfel încât reseaua sa dea un raspuns cât mai aproape de cel corect. Prin contrast, învățarea nesupervizata, sau învățarea fara profesor, nu necesita un raspuns corect asociat fiecarui tip de intrare din multimea de intrari de antrenament. În schimb reseaua exploateaza corelatiile întâlnite în cadrul datelor de intrare si organizeaza sabloanele în categorii rezultate din aceste corelatii. Învățarea hibrida combina învățarea supervizata si cea nesupervizata. O parte dintre ponderi sunt determinate prin învățarea supervizata iar celelalte sunt obtinute prin învățarea nesupervizata;
- **Funcția de activare**. Poate fi folosita o plaja întinsa de functii de activare dar cele mai eficiente în rezolvarea problemelor dificile sunt functiile de activare neliniare. Studiul acestor functii este dificil fapt pentru care nici retelele care se bazeaza pe astfel de functii nu se stie exact cum functioneaza, similar cu sistemul neural uman care este necunoscut în proportie de 80%.



**Domeniile de aplicabilitate / utilitate al rețelelor neurale:**

- ☐ **Clasificarea paternurilor.** Sarcina de clasificare a paternurilor se refera la a cataloga un patern reprezentat printr-un vector de trasaturi la una din clasele prespecificate (recunoasterea caracterelor, recunoasterea de voce, clasificarea celulelor de sânge, etc.)
- ☐ **Clustering sau gruparea.** În grupare, care se mai numeste si clasificare a paternurilor nesupervizata nu exista date de antrenament si clase cunoscute. Un algoritm de grupare exploateaza similaritatile dintre paternuri si plaseaza paternuri similare într-o grupare (compresia de date).
- ☐ **Aproximare de functii.** Fiind data o multime de perechi  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$  generate de o functie necunoscuta. Sarcina aproximarii unei functii este de a gasi un estimator a acesteia cu care sa poate sa fi generate si alte valori cu o anumita aproximatie (predictia la bursa, prognoza vremii).
- ☐ **Predictia.** Dându-se o multime de  $n$  exemple  $(y(t_1), y(t_2), \dots, y(t_n))$  în ordinea temporală  $t_1, t_2, \dots, t_n$  problema este de a predictiona valoarea de la momentul  $t+1$ .
- ☐ **Optimizarea.** Scopul optimizarii este de a gasi o solutie la o anumita problema dar care sa satisfaca anumite restrictii cum ar fi *maximizarea* sau *minimizarea*.
- ☐ **Control.** Având un sistem caracterizat de o marime de intrare si control  $u(t)$  si una de iesire  $y(t)$  scopul este de a genera o intrare de control astfel încât sistemul urmeaza o anumita traiectorie dupa un anumit model.

**7.2.2. MODELE DE RETELE NEURALE****7.2.2.1. PERCEPTRONUL**

*Perceptronul* este una dintre cele mai simple modele de retele neurale fiind utilizat la clasificarea paternurilor. Perceptronul consta dintr-un singur neuron cu ponderi ajustabile,  $(w_1, w_2, \dots, w_n)$ , si functie de activare de tip treapta, asa cum se arata în figura x.1. Dându-se un vector de intrare  $x=(x_1, x_2, \dots, x_n)$  intrarea neuronului este:

$$v = \sum_{j=1}^n w_j x_j \quad \text{si} \quad y = u(v)$$

iar iesirea perceptronului este +1 pentru  $v > 0$  si 0 altfel. Într-o problema de clasificare, perceptronul asigneaza un patrn de intrare unei clase daca  $y=1$  si altei clase daca  $y=0$ . Rosenblatt a dezvoltat un algoritm pentru adaptarea ponderilor folosind o multime de patrnuri de antrenament.

1. Ponderile sunt initializate cu numere aleatoare mici
2. La intrare este plasat un vector  $(x_1, x_2, \dots, x_n)$ , se calculeaza

$$v = \sum_{j=1}^n w_j x_j \text{ si } y = u(v) \text{ (se prezinta retelei date de intrare pentru care}$$

se cunoaste raspunsul dorit).

3. Sunt modificate ponderile dupa formula  $w_j(t+1) = w_j(t) + \eta(d - y)x_j$ , unde  $d$  este iesirea dorita,  $t$  este numarul iteratiei si  $\eta$  ( $0 < \eta < 1$ ) este câstigul sau rata de învățare. Daca raspunsul rețelei este diferit fata de raspunsul dorit, atunci se ajusteaza ponderile, în asa fel încât perceptronul sa clasifice corect datele de intrare.

Dupa ce rețeaua clasifica corect toate exemplele, se poate considera ca antrenarea s-a încheiat si se asteapta ca în momentul introducerii unor seturi de date noi, perceptronul sa raspunda corect. Conform algoritmului de învățare ponderile sunt ajustate doar când raspunsul este incorect si le lasa neschimbate daca clasificarea a fost corecta. Rosenblatt a demonstrat ca atunci când patrnurile de antrenament provin din doua clase separate algoritmul de învățare al perceptronului converge dupa un numar finit de pasi. Multe variatii ale acestui algoritm de învățare au fost propuse în literatura fiecare ducând la diferite caracteristici ale algoritmului de învățare.

*Teorema de convergenta* a perceptronului, a carei demonstratie presupune elemente de matematica superioara afirma ca, pentru *clase liniar separabile* exista întotdeauna un vector pondere astfel încât rețeaua sa clasifice corect exemplele de învățare. Prin *clase liniar separabile*, se înțelege doua clase ce pot fi despartite printr-un hiperplan (o dreapta în plan, un plan în spatiul 3D etc.). Ecuatia hiperplanului de separatie este furnizata de valorile ponderilor perceptronului.

O alta problema care se pune este cum trebuie modificate ponderile rețelei, în asa fel încât sa ajungem la solutie, atunci când ea exista. Ideea de baza pentru antrenarea oricarui tip de rețea este folosirea unei functii criteriu care trebuie minimizata. O astfel de functie poate determina suma distantelor pâna la hiperplanul de separatie a punctelor gresit clasificate. Cu cât valoarea acestei functii este mai mica (chiar zero), cu atât hiperplanul ales separa mai bine cele doua clase, deci este mai apropiat de solutie.

Asadar strategia de învățare este redusă la minimizarea unei funcții multivariabile, ceea ce este remarcabil [Burj01].

Minimizarea funcției criteriu se poate face prin metode de tip gradient.

Liniar separabilitatea este una din cele mai importante limitări ale perceptronului. De multe ori, exemplele de învățare în practică, nu constituie clase liniar separabile (cazul celebrei probleme pe care un perceptron cu un singur strat nu o poate rezolva: funcția XOR).

O modificare a algoritmului clasic de învățare a fost adusă de către Gallant în anul 1986, astfel încât acesta să poată fi aplicat și claselor liniar inseparabile. Ideea este de a antrena rețeaua printr-un număr mare de pași, și de a reține configurația care a clasificat corect cele mai multe date de intrare. De multe ori însă, acest algoritm nu prezintă aplicabilitate practică, deoarece sunt necesare extrem de multe iterații pentru a ajunge la o soluție acceptabilă.

Un alt algoritm de învățare a perceptronului cu un singur strat este algoritmul Widrow-Hoff, cunoscut și sub numele de *regula delta*. Acest algoritm alege drept funcție criteriu suma erorilor pătratice:

$$\sum_{k=1}^n (d_k - o_k)^2$$

unde  $n$  reprezintă numărul exemplurilor de antrenare,  $d_k$  ieșirea dorită pentru exemplul  $k$ , iar  $o_k$  rezultatul obținut. Se urmărește minimizarea ei, folosind din nou metode de tip gradient și se ajunge la următoarea regulă de învățare:

$$\begin{cases} p_{t+1} = p_t + \frac{c}{t} \cdot \mathbf{d} \cdot x \\ \mathbf{d} = d_t - o_t \end{cases}$$

unde:

- $p_{t+1}$  este vectorul de ponderi la pasul  $t+1$ ;
- $p_t$  este vectorul de ponderi la pasul  $t$ ;
- $d_t$  este rezultatul dorit la pasul  $t$ ;
- $o_t$  este rezultatul obținut la pasul  $t$ ;
- $x_t$  este vectorul valorilor de intrare la pasul  $t$ ;
- $c$  este constanta de corectie ( $0 < c \leq 1$ )

În acest caz, rețelei  $i$  se prezintă exemple de instruire, până când eroarea de clasificare este mai mică decât o eroare acceptabilă, moment în care se încheie procesul de antrenare.

În general, algoritmul de învățare al lui Widrow-Hoff produce o soluție mai bună decât algoritmul clasic, deoarece există posibilitatea de a controla eroarea hiperplanului de separație. În cazul algoritmului clasic,

învățarea se oprește în momentul în care toate exemplele de instruire au fost corect clasificate, chiar dacă soluția obținută este acceptabilă "*la limita*". Dacă clasele nu sunt liniar separabile, algoritmul delta va produce cu siguranță o soluție mai bună, deoarece algoritmul clasic va oscila și nu va converge niciodată, și chiar și în varianta Gallant va produce rezultate mai puțin satisfăcătoare [Burj01].

Deși gama de probleme pe care un perceptron cu un singur strat le poate rezolva este destul de limitată, el reprezintă punctul de plecare pentru rețele mult mai complexe cu aplicații practice.

#### 7.2.2.2. MODELUL LVQ (LEARNING VECTOR QUANTIZATION)

LVQ este o altă rețea neurală simplă folosită pentru clasificarea paternelor. Acest mod de învățare este unul supervizat bazat pe competiție. În cadrul algoritmului vor fi atâtia vectori binari câte clase diferite sunt. Numărul de biți folosiți la codificarea unui singur patern determină lungimea acestor vectori. Pentru a clasifica un patern este calculată distanța Hamming dintre paternul de intrare și fiecare vector  $D = \sum (X - V)^2$ , unde  $X$  este paternul de intrare și  $V$  este unul dintre vectori. Paternul  $X$  este clasificat ca făcând parte din clasa asociată vectorului care are distanța Hamming cea mai mică. Fiind un algoritm de învățare ponderile vectorilor inițiali trebuie modificate, dar într-un singur ciclu se modifică doar vectorul câștigător, cel cu distanța Hamming minimă, după formula:

$$V(t) = V(t-1) + \eta [X - V(t-1)],$$

unde  $X$  este paternul de intrare,  $V$  este vectorul câștigător iar  $\eta$  este rata de învățare  $\eta \in [0, 1]$ .

Restul vectorilor rămân nemodificați. De aceea acest tip de învățare este bazat pe competiție; numai unele ponderi sunt modificate. Algoritmul a fost aplicat de noi în implementarea unor predictoare de branch-uri speciale (vezi în continuare).

#### 7.2.2.3. MODELUL DE ÎNVĂȚARE BACKPROPAGATION

Backpropagation este un algoritm de învățare folosit în rețelele de tip feed-forward. Backpropagation definește doi pași: primul în care informația trece de la intrare către ieșire și apoi un pas înapoi de la nivelul de ieșire la nivelul de intrare. Pasul de trecere înainte propaga vectorul de intrare în primul nivel al rețelei, ieșirile din acest nivel produc un nou vector care vor fi intrări pentru nivelul următor, până când se ajunge la ultimul nivel al caror

iesiri produc iesirile rețelei. Pasul înapoi este similar cu cel înainte exceptând faptul ca erorile sunt propagate înapoi prin rețea pentru a determina adaptarea ponderilor. Algoritmul Backpropagation poate fi formulat astfel:

1. Initializeaza ponderile cu valori mici aleatoare.
2. Plaseaza la intrare un vector X.
3. Propaga acest vector de intrare înainte prin rețea.
4. Calculeaza eroarea în nivelul de iesire.  

$$\mathbf{d}^n = f'(h_i^m)[d_i^n - y_i^m]$$
, unde  $f'$  este derivata functiei de activare  $f$ ,  $h_i^m$  reprezinta intrarile perceptronului  $i$  din nivelul  $m$ ,  $d$  este rezultatul dorit iar  $y$  este rezultatul produs de ultimul nivel al rețelei.
5. Calculeaza erorile pentru nivelele precedente prin propagarea erorilor înapoi:  

$$\mathbf{d}^l = f'(h_i^m) \sum_j w_{ji}^{l+1} \mathbf{d}_j^{l+1}$$
, unde  $l$  ia valori de la numarul nivelului ultim -1 pâna la 1.
6. Actualizeaza ponderile folosind  $\Delta w_{ji}^l = \eta \mathbf{d}_i^l y_j^{l-1}$ .
7. Repeta de la pasul 2 cu urmatorul patern pâna când eroarea din nivelul de iesire scade sub un anumit nivel.

Alegerea lui  $\eta$  influenteaza într-o mare masura algoritmul de învățare Backpropagation. Totusi alegerea lui  $\eta$  este dependenta de specificul problemei. Metoda backpropagation fiind o metoda de minimizare a erorii medii patratice sufera de *deficientele generale ale tehnicilor de gradient*:

- ☐ Rata de învățare,  $\rho$ , care da marimea pasului pe directia gradientului trebuie sa fie suficient de mica. Aceasta deoarece gradientul este o masura locala, iar daca  $\rho$  este prea mare s-ar putea ca masura  $J$  a erorii sa nu scada ci sa oscileze, vectorul coeficientilor sinaptici sarind de pe o parte pe cealalta a "gropii" în care se gaseste minimul local. Pe de alta parte, un  $\rho$  prea mic conduce la o viteza prea mica de convergenta ceea ce conduce la o învățare prea lenta.
- ☐ Indiferent de cât de rapid atinge minimul, acesta va fi minimul **local** cel mai apropiat si din acesta nu va putea iesi pentru ca tehnica backpropagation fiind o metoda de gradient este o metoda ce exploateaza proprietatile locale ale functiei criteriu. În plus, cu cât dimensiunea spatiului de cautare (spatiul ponderilor) este mai mare (avem un numar mai mare de neuroni în straturile ascunse), cu atât creste numarul minimelor locale si deci si "sansa" de a esua într-unul dintre acestea.

La metoda backpropagation clasica ramâne la alegerea utilizatorului precizarea valorii ratei de învățare, adica a marimii pasului în directia

gradientului functiei criteriu  $J$  (masura erorii). Desi nu exista o metoda universala de alegere a lui  $\rho$  într-o problema data se recomanda ca acesta sa fie subunitar sau, eventual, descrescator odata cu cresterea numarului iteratiei. Viteza de scadere a functiei criteriu depinde foarte puternic de valoare aleasa pentru  $\rho$ . În general se recomanda sa se testeze evolutia retelei pentru diverse valori ale lui  $\rho$  si apoi sa se aleaga o valoare convenabila. Se poate porni cu o valoare relativ mica pentru  $\rho$  si sa se creasca aceasta valoare, atât timp cât prin aceasta crestere scade  $J$ . În momentul în care  $\rho$  depaseste valoarea optima si devine prea mare apar oscilatii ale valorii lui  $J$  care uneori creste în loc sa scada.

Una dintre retelele în care este utilizata învatarea backpropagation este reteaua feed-forward multi-layer-perceptron. Dezvoltarea algoritmului backpropagation pentru determinarea ponderilor într-o retea multi-layer-perceptron a dus la popularitatea ei în rândul utilizatorilor de retele neurale. În general o retea multi-layer-perceptron este o retea feed-forward formata dintr-un numar de nivele ascunse si un nivel de iesire, fara conexiuni între perceptronii aflati pe acelasi nivel si fara conexiuni înapoi între nivele.

### 7.2.3. PREDICTIA DE SALTURI PRIN METODE NEURALE

Poate fi abordata si prin metode de inteligenta artificiala datorita apropierei problematiei predictiei de salturi de recunoasterea de paternuri. Prin abordarea neurala a predictiei tabela de predictie este înlocuita cu informatia, distribuita, înmagazinata în ponderile retelei iar predictia automatului cu rezultatul produs de nivelul de iesire al retelei neurale. Revenind la domeniul recunoasterii de paternuri, reteaua neurala clasifica paternurile în doua categorii: salturi care ar trebui predictionate ca *taken* si salturi care ar trebui predictionate ca *not taken*.

#### 7.2.4. ALGORITMI GENETICI (EVOLUTIONARY COMPUTING)

Algoritmii genetici constituie o parte a calculului evolutiv care la rândul sau este o componenta a inteligenței artificiale. Ca și rețelele neurale, algoritmii genetici se bazează pe o metaforă biologică (asocierea soluției unei probleme unui cromozom uman și folosirea operatorilor genetici pentru determinarea de noi soluții). Prin acești algoritmi se percepe învățarea ca o competiție între membrii unei populații de soluții posibile a unei probleme în evoluție. Este utilizată o funcție de evaluare a fiecărei soluții care să estimeze dacă o soluție va contribui la următoarea generație de soluții. Apoi prin operații similare cu transferul de gene din reproducerea sexuală algoritmul creează o nouă populație de posibile soluții. O formă generală a unui algoritm genetic este următoarea:

Fie  $P(t)$  o populație de soluții candidat.

1. Generarea aleatoare a unei populații de  $n$  cromozomi (care reprezintă soluții posibile pentru problema)
2. Este evaluat fiecare membru al populației (rata de *fitness*)
3. [*Selectie*] Sunt selectați cei mai promitatori membri din populație.
4. Sunt produși urmași folosind operatorii genetici (*crossover*, *mutatie*, *inversie*).
5. Cei mai slabi candidați sunt înlocuiți cu acești urmași.
6. Se repetă de la pasul 2 un anumit număr de generații, sau până s-a ajuns la o soluție convenabilă.
7. Cel mai valoros membru al populației este ales ca soluție a problemei.

Acest algoritm trasează cadrul general al unui algoritm genetic, diferitele implementări particularizând cadrul de lucru pe problemele concrete de rezolvat orice informație euristica privind alegerea parametrilor fiind utilă. Parametrii cheie ai acestui algoritm sunt numărul de generații și dimensiunea populației. Dacă populația este prea mică nu va fi suficientă diversitate în cadrul populației pentru a obține soluția optimă. Dacă însă numărul de generații este prea mic nu vor fi destule șanse pentru a ajunge la soluția optimă. De notat că optimul nu este garantat că va fi obținut, există doar o mare probabilitate de a-l găsi. Membrilor populației li se atribuie o valoare promisiunii lor pe baza careia vor fi aleși spre reproducere cei mai promitatori. Această valoare este returnată de funcția de evaluare  $h(i)$  care măsoară starea funcție de promisiunea membrilor. Deci funcția de evaluare aleasă va influența sensibil rezultatul obținut. În loc să se

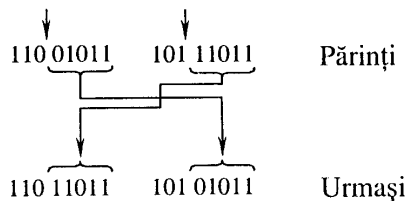
foloseasca direct functia de evaluare  $h(i)$  este mai util sa o convertim într-o functie normalizata:

$$f(i) = \frac{h(i)}{\sum_j h(j)}. \text{ Deoarece } \sum_j h(j) = 1 \text{ aceasta metoda}$$

permite folosirea valorilor de evaluare ca probabilitati.

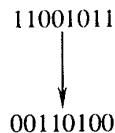
Dupa evaluarea fiecarui candidat algoritmul selecteaza perechi pentru recombinare. Pentru recombinare sunt folositi operatori genetici pentru a produce noi solutii prin combinarea elementelor parintilor. Ca si în evolutia naturala, cei mai sanatosi vor supravietui, acelor candidati care au cele mai bune evaluari li se va da o mai mare probabilitate de reproducere. Selectia este deseori probabilistica si cei mai slabi membri vor avea o probabilitate mai mica de a se reproduce dar nu sunt eliminati direct. Supravietuirea celor mai slabi este importanta deoarece pot contine înca componente esentiale al solutiei care vor putea fi extrase prin reproducere.

Exista un numar de operatori genetici folositi pentru a produce urmasi având trasaturile parintilor. Primul operator este **crossover**. Crossover primește doi candidati si îi divide, schimbând partile obtinute între ele pentru a produce alti doi membri. Punctele de interschimbare sunt selectate aleator. Sa consideram o populatie în care membrii sunt formati din numere binare de câte 5 cifre. Dupa selectia a doi membrii din populatie operatorul crossover se aplica astfel:



**Figura 7.3.** Aplicarea operatorului *crossover* asupra 2 membrii dintr-o populatie

Un alt operator important este **mutatia**. Acesta primește un singur candidat si schimba anumite aspecte din el. De exemplu ar putea schimba toti bitii din 0 în 1 si invers.

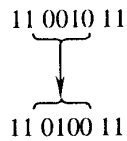


**Figura 7.4.** *Mutatia* aplicata membrilor unei populatii



Mutatia este importanta deoarece populatia initiala poate exclude o componenta esentiala a solutiei ducând astfel la un minim (optim) local. Prin operatorul de mutatie diversitatea este reintrodusa în populatie pentru a evita minimul local. Astfel este ales aleator un membru al populatiei si i se aplica operatorul de mutatie.

Alt operator genetic este *inversia* care inverseaza numai zona selectata dintr-o solutie. 01



**Figura 7.5.** *Inversia* aplicata membrilor unei populatii

### 7.3. DESFASURAREA LUCRARIII

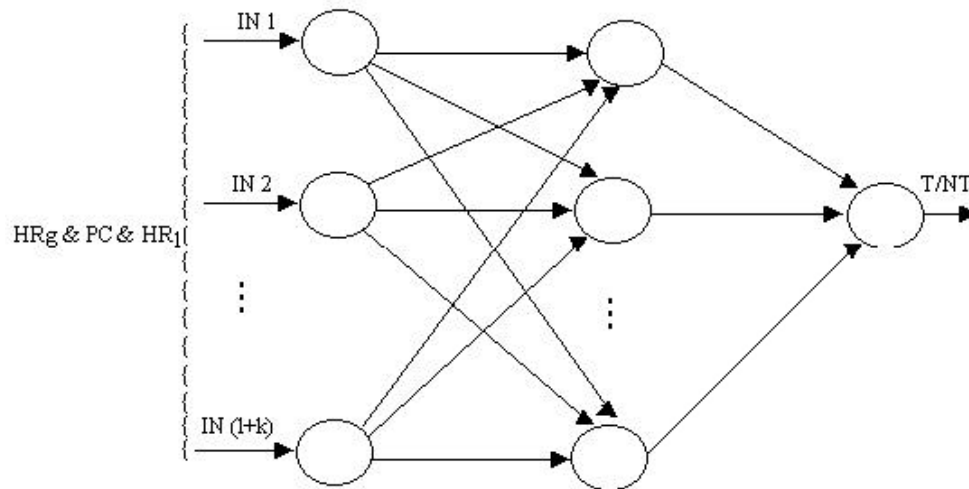
#### 7.3.1. INTERFATA CU UTILIZATORUL

Predictorul neuronal de salturi pe care l-am implementat în simulator este constituit în principal dintr-o retea neuronală, un registru de istorie global (HRG) si o tabela cu registrii de istorie locala.

**Reteaua neuronală** (pe post de predictor global al tuturor salturilor din program) este o retea de tip Multi Layer Perceptron, de tip *feedforward* având un singur nivel ascuns, un nivel de intrare si un nivel de iesire (vezi figura 7.6). Fiecare nivel este format dintr-un numar de noduri astfel [Vin99a, Vin00a]:

- Pentru primul nivel numarul de noduri este calculat dupa urmatoarea formula: *dimensiunea PC*-ului (adica 10 biti - suficient pentru benchmark-urile Stanford) + *dimensiune HRG* (KG) + *dimensiunea unui HRL* (KL), unde KG si KL sunt parametrii de intrare.
- Dimensiunea nivelului al doilea este data de *dimensiunea primului nivel* + *un delta dat* ca parametru de intrare (având ca valori discrete 2,4,6,8).

Nivelul de iesire contine un singur nod si constituie predictia saltului (*taken / not taken*).



**Figura 7.6.** Predictor neural de ramificatii program

Fiecare nod de pe nivelurile ascuns si de iesire este în legatura cu nodurile nivelului care le precede prin intermediul unor ponderi. Valorile fiecarui nod de pe aceste doua niveluri sunt calculate dupa urmatoarea formula:



unde  $n_j$  este nodul a carui valoare este calculata

$n_i$  este unul din nodurile nivelului precedent iar  $w_{i,j}$  este ponderea ce leaga nodul  $i$  de nodul  $j$

Pentru functia  $f$  (functie de *activare*) s-a folosit în cadrul simularii doar functia sigmoidala  $\frac{1}{1 + e^{-x}}$ .

Valorile nodurilor de pe nivelul de intrare vor fi cuprinse în intervalul  $[0.1, 0.9]$ .

Valorile initiale ale ponderilor se gasesc în intervalul  $[-2/N_1, 2/N_1]$  unde  $N_1$  reprezinta numarul de noduri de pe nivelul de intrare [Gal93].

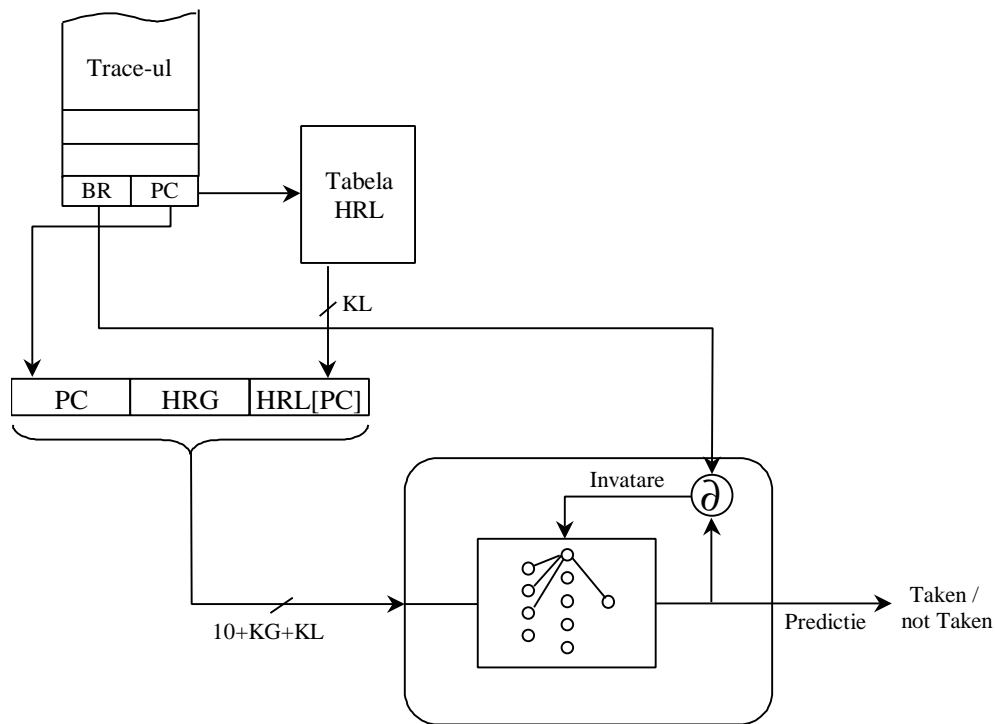
Registrul de istorie globala (HRG) este un registru de deplasare de dimensiune, parametrizabila (KG), si contine rezultatul ultimelor KG salturi, având rolul unui prim nivel de istorie a salturilor.

Tabela registrilor de istorie locala reprezinta al doilea nivel de istorie a salturilor. Spre deosebire de HRG care era un registru comun tuturor salturilor aici fiecare PC are propriul registru de istorie. In acest sens tabela HRL este implementata ca o tabela direct mapata cu  $2^{\dim(PC)}$  intrari, fiecare intrare fiind un registru de KL biti.

### 7.3.2. METODOLOGIA DE SIMULARE

Simulatorul implementat este unul bazat pe trace-uri, astfel încât principalele date de intrare ale predictorului le vor constitui trace-urile. Aceste trace-uri au fost obținute în urma trecerii benchmark-urilor Stanford printr-un simulator la nivel de execuție dedicat arhitecturii HSA (scris de cercetatorii universitatii din Hertfordshire, UK) [Col93] și a eliminării instrucțiunilor care nu constituie salturi, rezultând fișiere ce conțin intrări de forma  $\langle BR\ adr1\ adr2 \rangle$ .  $BR$  semnifică codul instrucțiunii de salt împreună cu rezultatul saltului (se face sau nu),  $adr1$  este PC-ul instrucțiunii și  $adr2$  este adresa destinație a saltului.

Privind dintr-o altă perspectivă putem considera predictorul format dintr-o bandă de intrare (trace-ul), un automat de predicție (rețeaua neuronală) și o memorie (HRG și tabela HRL) ce conține istoria salturilor.



**Figura 7.7.** Metodologia de simulare

Procesul de predicție decurge în felul următor:

1. Sunt initializate ponderile rețelei neuronale cu valori aleatoare în intervalul  $[-2/N_1, 2/N_1]$ , Registrul HRG și toate intrările din tabela HRL cu 0.
2. Este citită o intrare din fisierul trace.
3. Se construiește un vector de valori binare reprezentând imaginea binară a valorii PC-ului concatenat cu registrele HRG și HRL-ul corespunzător PC-ului curent din tabela HRL.
4. Elementele vectorului obținut sunt transformate din valori de 0 și 1 în valori din intervalul  $[0.1, 0.9]$  datorită funcției de activare folosită. Aceste valori sunt aplicate pe intrările rețelei neuronale, ele devenind valorile nodurilor de pe nivelul de intrare. Evident, dimensiunea vectorului este identică cu dimensiunea nivelului de intrare.
5. Sunt calculate valorile nodurilor de pe celelalte două nivele conform formulei de mai sus, realizând astfel etapa *forward*.
6. Valoarea nodului de pe nivelul de ieșire reprezintă rezultatul predicției, astfel:
  - Dacă valoarea este mai mică decât 0.5 rezultatul este considerat ca fiind NotTaken.
  - Similar, dacă valoarea este mai mare decât 0.5 rezultatul va fi considerat ca fiind Taken.
7. Se compară rezultatul prezis al instrucțiunii de salt cu cel real obținut din trace (determinat în urma simulării benchmark-ului la nivel de execuție), contorizându-se fie succesul fie eșecul predicției în funcție de rezultatul comparației.
8. Rezultatul real este stabilit ca *scop* la ieșirea rețelei neuronale și se calculează pentru fiecare pondere eroarea cu care a contribuit la diferența dintre scop și valoarea de pe nivelul de ieșire. Scopul ia valoarea 0.9 pentru predicție Taken și 0.1 pentru predicție NotTaken.
9. Se aplică, în cadrul etapei *backward*, fiecărei ponderi corecția calculată anterior înmulțită cu un pas de învățare. În acest fel spunem că rețeaua “învată” (se adaptează) din greselile comise.
10. Registrul de istorie globală HRG este actualizat conform execuției saltului, astfel:
  - HRG este deplasat la stânga cu o poziție pierzându-se valoarea celui mai semnificativ bit.
  - Cel mai puțin semnificativ bit este setat conform rezultatului real al saltului: valoarea 0 sau 1 (0-taken , 1-nottaken).

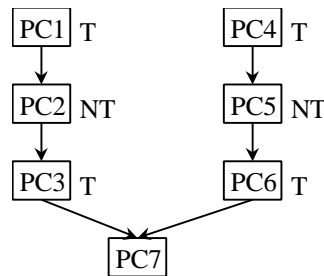
Registrul din Tabela HRL, adresat cu PC-ul curent este actualizat similar, obținându-se astfel o istorie globala a salturilor (comuna tuturor salturilor - HRG) si una locala (specifica fiecarui PC - HRL).

11. Se repeta de la pasul 2 pâna la terminarea trace-ului.

Principalii parametri ai predictorului, ce pot fi alterati de catre utilizator sunt:

- *dimensiunea registrului de istorie globala HRG (KG)*. Plaja de valori este: 2, 4, 6, 8, 10 biti.
- *dimensiunea registrului de istorie locala HRL (KL)* poate varia incremental în intervalul: 0÷4.
- *numarul de noduri de pe nivelul ascuns al rețelei* ia urmatoarele valori prin adaos fata de *numarul de noduri de pe nivelul de intrare*: +0, +2, +4, +6, +8.
- *pasul de învățare al rețelei* poate lua valorile: 0.125, 0.5 si 1.

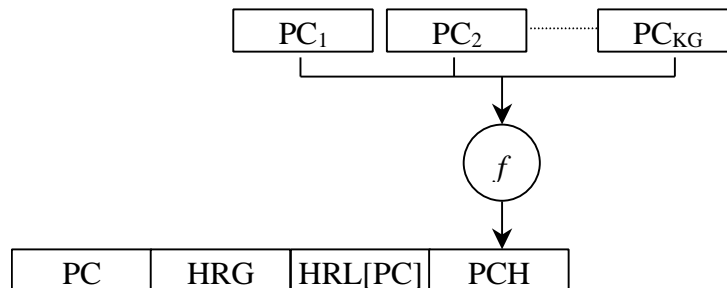
Exista situatii în executia benchmark-urilor când pentru un acelasi context global (sa presupunem T, NT, T) se ajunge la o anumita instructiune de salt pe doua cai diferite. Astfel, daca la instructiunea de salt având adresa PC7 se ajunge urmând calea de executie data de PC-urile PC1, PC2, PC3 predictia poate fi Taken iar daca se ajunge urmând cealalta cale (PC4, PC5, PC6) saltul poate fi predictionat ca Not Taken, desi are aceeasi istorie a executiei (T, NT, T).



**Figura 7.8.** Exemplificare cu context global identic (T, NT, T) dar cai distincte de executie aparute în procesare

Un pas important în vederea creșterii acuratetii de predicție îl constituie *introducerea de informatie suplimentara la intrarea rețelei neuronale*. În acest sens sunt adaugate ca si intrari în retea pentru fiecare bit din registrul de istorie globala (HRG) valoarea PC-ului aferent (adresa saltului care s-a facut). Astfel, doua “cai” de salturi distincte care converg spre aceeasi instructiune de salt si cu aceeasi istorie a executiei vor avea informatia de predicție diferita. Prin extinderea informatiei de corelatie a

salturilor are loc reducerea interferentelor de tipul anterior cu consecinte benefice asupra acuratetii de predictie [Vin99b].



**Figura 7.9.** Reducerea dimensiunii tabelelor folosind functii de dispersie

Introducerea de informatie suplimentara la intrarea predictorului constând dintr-un numar de KG registre (unde KG este dimensiunea HRG) s-a realizat prin concatenarea la vechiul vector ( $PC+HRG+HRL[PC]$ ) a unui vector ( $PCH$ ) de 10 biti obtinut prin combinarea celor KG registre de istorie a PC-ului dupa o functie de dispersie  $f$  (OR, XOR, MOD).

O alta solutie în vederea cresterii acuratetii de predictie consta în combinarea predictiei statice cu cea dinamica in trei moduri:

- ❖ Prima metoda consta în preînvatarea unor statistici, realizate asupra trace-ului, de catre retea neuronală cu o anumita eroare urmata de predictia propriu-zisa care se realizeaza dupa modelul descris mai sus. În prima faza ponderile sunt ajustate astfel încât retea “învata” o parte din trace, realizându-se o pre-predictie (se stabileste anterior executiei fiecare salt cum va fi predictionat atunci când va fi executat) înainte executiei iar apoi urmeaza predictia dinamica din timpul executiei.
- ❖ A doua metoda consta în “antrenarea” retelei folosind toate cele 8 trace-uri furnizate în mod aleator pâna când eroarea maxima a învățării va scade sub o anumita valoare. Antrenarea este realizata printr-o predictie “în gol” adica predictia unui trace fara retinerea rezultatului. Dupa ce eroarea învățării a ajuns sub o anumita valoare urmeaza predictia propriu-zisa unde de data aceasta sunt retinute rezultatele predictiei.
- ❖ A treia metoda consta în utilizarea unui algoritm genetic pentru antrenarea retelei neurale.

Revenind la prima metoda, etapa de pre-învatare corespunzatoare unui trace se realizeaza în trei pasi si anume:

- (1) Se determina o **statistica** pe baza trace-ului în care se studiaza numarul (procentul) de salturi facute si nefacute având aceleasi perechi de  $PC+HRG+HRL$ , cu alte cuvinte de câte ori a fost Taken si not Taken un anume salt în aceleasi conditii ale istoriei locale si

globala (HRG+HRL). Va rezulta un fisier (având extensia *.ssa*) cu intrari de tipul:

*PC HRG HRL nrTaken nrNotTaken %Taken %NotTaken*

- (2) Rezultatele acestei statistici sunt **filtrate** rezultând un alt fisier (având extensia *.sta*) care va constitui “*informatia*” folosita de retea neuronală pentru “*învatare*”. Filtrarea se va realiza astfel: toate intrarile din fisierul *\*.ssa* având procentul de Taken sau NotTaken mai mic de o anumita valoare parametru vor fi respinse. Intrarile care au procentul Taken mai mare decât valoarea parametru (valoare de prag stabilita apriori) se vor regasi în fisierul de iesire sub urmatoarea forma:

*PC HRG HRL 1,*

iar cele cu procentul NotTaken mai mare vor avea la iesire urmatoarea forma:

*PC HRG HRL 0.*

Prin aceasta filtrare se încearca eliminarea din statistica a salturilor indecise (nepolarizate spre taken sau not taken), de genul 50% taken si 50% not taken, care nu vor face decât sa “*ameteasca*” retea neuronală nestiind ce sa predictioneze. Daca în cazul unei intrari balanta procentului va înclina mai mult spre taken atunci retea neuronală este învatata static ca la acel PC si în acel context al istoriei sa predictioneze taken. Similar, este învatata sa predictioneze not taken în cazul înclinarii balantei în cealalta parte.

- (3) Ultima faza consta în **învatarea** propriu-zisa a statisticilor filtrate. Se aplica intrarii retelei câte un vector constând din reprezentarea binara a PC+HRG+HRL (citit din fisierul *\*.sta*) si se ajusteaza ponderile în raport cu rezultatul dorit (citit de asemenea din fisierul *\*.sta*). Procesul de învatare continua pâna când eroarea maxima (diferenta) dintre rezultatul predictiei si scop este mai mica decât o anumita valoare. Predictiei statice realizate îi urmeaza predictia dinamica dupa algoritmul prezentat la începutul subcapitolului 3.2. al prezentei lucrari (*Metodologia de simulare*) cu deosebirea ca ponderile nu mai sunt aleatoare ci sunt cele ramase în urma pre-învatarii.

În cadrul simulatorului implementat, la obtinerea rezultatelor predictiei cu pre-învatare folosind prima metoda, parametrul KL este 0 (nu exista istorie locala a salturilor) variind doar numarul de noduri de pe nivelul ascuns, istoria globala KG între valorile 0, 2, 4, 6, 8, si procentul de filtrare a statisticilor în intervalul 60÷95.

A treia metoda de pre-învatare statica o constituie utilizarea unui algoritm genetic pentru antrenarea rețelei neurale. Conform modelului teoretic s-a stabilit o populatie de cromozomi suficient de larga (50) pentru a asigura variabilitatea populatiei. Fiecare cromozom contine un numar de gene, acestea reprezentând de fapt o pondere din cadrul rețelei neurale. Genele aparținând populatiei initiale de cromozomi sunt initializate cu valori aleatoare în intervalul  $[-1,1]$ . Algoritmul genetic se desfasoara dupa scenariul descris în partea teoretica a lucrării:

- ☐ pentru un numar de generatii stabilit se repeta pasii urmasori
  - /// sunt evaluati toti cromozomii
  - /// este aplicat operatorul de încrucisare (Cross)
  - /// este aplicat operatorul de mutatie (Mutation)
- ☐ cel mai performant cromozom din generatia finala va contine prin genele sale ponderile rețelei antrenate static.

Din pacate, varianta actuala de simulator prezinta unele deficiente (referitoare doar la implementarea algoritmilor genetici) fapt ce determina întreruperea simulării în diferite faze ale procesării benchmark-urilor. În rest, oricare varianta de simulare se alege (fara antrenare sau cu pre-învatare pe baze statistice), executia decurge normal, rezultatele obtinute fiind comparabile cu cele de la schemele clasice de predictie si chiar mai bune.

### 7.3.3. DESCRIERE IMPLEMENTARE SOFTWARE

Interfata cu utilizatorul este "*user-friendly*", bazata pe meniuri, ferestre de dialog, pagini de proprietati, imagini grafice elocvente. Ea a fost conceputa în vederea unei dezvoltari ulterioare (extinderii) variantei actuale de simulator, de catre dl. ing. Marius Sbera - MSc, în cadrul unui proiect de diploma coordonat de catre unul din autorii acestei carti [Sbe00]. Bazat pe faptul ca limbajul C++ ofera un suport puternic pentru programarea orientata pe obiecte: mosteniri multiple, redefinirea operatorilor, polimorfism, functii si clase prieten etc. implementarea simulatorului s-a realizat folosind mediul de dezvoltare Microsoft Visual C++, versiunea 6.0. Înainte de a prezenta "*structura si nucleul aplicatiei*" sunt descrise câteva aspecte legate de programarea sub Windows, mai precis despre arhitectura *Document / View* a aplicatiilor scrise pentru sistemul de operare Windows '9x, NT, 2000, XP. Se poate spune ca structura interfetei utilizata este de tipul **MDI** (*multiple document interface*) *Document / View*.

Desi programele secventiale (listing-uri simple) sunt potrivite pentru explicarea unor concepte simple, cum ar fi elementele de baza ale limbajului



C++, totusi functionarea lor nu este corespunzatoare în medii multitasking gen Microsoft Windows. Într-un astfel de mediu, toate resursele se folosesc în comun: tastatura, mouse-ul, chiar si utilizatorul. Programele scrise pentru Windows trebuie sa colaboreze cu sistemul de operare si cu alte programe care pot rula în acelasi timp. Multe resurse trebuie solicitate sistemului de operare înainte de a fi folosite si, odata folosite, trebuie returnate sistemului pentru a putea fi folosite si de catre alte programe. Aceasta reprezinta o modalitate de exercitare de catre Windows a controlului asupra unor resurse ca ecranul sau alte periferice. Programul nu trebuie sa-si imagineze ca detine controlul complet asupra calculatorului pe care ruleaza ci trebuie sa ceara permisiunea preluarii controlului asupra unei resurse centrale si sa fie gata sa reactioneze la orice evenimente care îl privesc (tratarea tuturor exceptiilor).

Arhitectura Document / View este folosita de MFC (*Microsoft Foundation Class*) si AppWizard pentru organizarea programelor scrise pentru Windows. Document/View separa un document în patru clase principale [Will98]:

- ☐ O *clasa document*, derivata din **CDocument**
- ☐ O *clasa de vizualizare*, derivata din **CView**
- ☐ O *clasa cadru*, derivata din **CFrameWnd** (**CMainFrame**)
- ☐ O *clasa de aplicatie*, derivata din **CWinApp**

Fiecare dintre aceste clase are un anumit rol într-o aplicatie MFC Document/View. Clasa *de vizualizare* trateaza interactiunea dintre document si utilizator. Clasa *cadru* contine vizualizarea si alte elemente de interfata cu utilizatorul, cum ar fi meniul si barele de instrumente. Clasa *de aplicatie* este responsabila cu lansarea în executie efectiva a programului, precum si cu tratarea unor interactiuni de uz general cu Windows.

Desi numele de *Document / View* pare sa limiteze utilizarea conceptului numai la programele de procesare a textelor, aceasta arhitectura se preteaza la o mare varietate de tipuri de programe. Nu exista nici un fel de restrictii referitoare la datele gestionate de clasa CDocument: acestea pot fi un fisier creat de un procesor de texte, de un program de calcul tabelar sau un server situat la cealalta extremitate a unei retele. Similar, exista mai multe tipuri de vizualizare. O vizualizare poate fi o simpla fereastră, cum este cazul aplicatiile SDI prezentate pe parcursul acestei carti (predictorul de salturi GAg, simulatorul SATSim, predictorul de valori LastValue), sau poate fi derivata din CFormView, cu toate caracteristicile unei casete de dialog.

Exista doua tipuri principale de programe Document/View :

- ☐ SDI, sau *Single Document Interface* (Interfata uni-document)

☐ MDI, sau *Multiple Document Interface* (Interfata document multiplu)

Un program SDI accepta un singur tip de document, si aproape întotdeauna un singur tip de vizualizare. Se poate deschide un singur document la un moment dat. O aplicatie SDI se concentreaza pe o anumita sarcina (*task*), fiind în mod normal, destul de directa. Un program MDI permite folosirea mai multor tipuri de documente, fiecare document putând avea mai multe vizualizari. La un moment dat pot fi deschise mai multe documente, iar documentul deschis foloseste frecvent o bara de instrumente si meniuri personalizate, care satisfac necesitatile acelui document.

Arhitectura Document / View asigura un cadru flexibil care poate fi folosit pentru crearea de programe Windows de aproape orice tip. Unul dintre marile avantaje ale arhitecturii Document / View este acela ca împarte activitatea unui program Windows în categorii bine definite. Majoritatea claselor se pot încadra într-unul dintre urmatoarele patru categorii:

- ⇒ Controale si alte elemente de interfata cu utilizatorul, legate de o anumita vizualizare.
- ⇒ Date si clase de tratare a datelor, care apartin unui document.
- ⇒ Tratarea barelor de instrumente, a barei de stare si a meniurilor, care tin de clasa cadru.
- ⇒ Interactiunea dintre aplicatie si Windows, care survine în clasa derivata din CWinApp.

Împartirea activitatii unui program contribuie la gestionarea mult mai eficienta a concepiei acestuia. Extinderea programelor care folosesc arhitectura Document / View este relativ simpla, deoarece cele patru clase principale Document / View inter-comunica prin intermediul unor interfete bine definite. Pentru a transforma un program SDI într-un program MDI, trebuie scrisa o cantitate redusa de cod. Modificarea interfetei utilizator pentru un program Document / View are efect numai asupra clasei sau claselor de vizualizare, nefiind necesare nici un fel de schimbari pentru clasele document, cadru sau de aplicatie.

Developer Studio asigura toate instrumentele necesare crearii unei aplicatii Document / View. Instrumentele de baza pentru crearea si întretinerea unui program Document / View sunt AppWizard si ClassWizard. AppWizard se foloseste pentru crearea proiectului initial al unui program SDI sau MDI. ClassWizard se foloseste pentru adaugarea de clase, pentru tratarea mesajelor, precum si pentru adaugarea de variabile claselor incluse într-un proiect [Will98].

Revenind la simulatorul implementat se disting urmatoarele clase în componenta proiectului:

- ❖ clasa de aplicatie **CProjApp** derivata din **CWinApp**
- ❖ cadrul principal al aplicatiei **CMainFrame**, care contine **ToolBar**-ul, **StatusBar**-ul si o *bara personalizata* cu setarile retelei neurale.
- ❖ cadrul copil **CChildFrame**.
- ❖ clasa de vizualizare a interfeței **CProjView** - care permite afisarea rezultatelor simulării sub forma unui grafic.
- ❖ documentul asociat fiecarui View este reprezentat de clasa **CProjDoc**.

Pe lângă aceste clase, care constituie arhitectura Document / View a aplicatiei, mai sunt câteva clase si structuri care constituie de fapt partea functionala a simulatorului. Astfel, simulatorul propriu-zis este încapsulat în cadrul clasei **CNwSimul**, rețeaua neurala în clasa **CNetw** iar cei doi algoritmi propusi pentru antrenarea rețelei în clasele **CClassicLearn** si **CGeneticLearn**, amândoua derivate din clasa abstracta **CLearningAlg**.

### 7.3.3.1. CLASE SI STRUCTURI UTILIZATE

Clasa **CNwSimul** - reprezinta „motorul” simulatorului. Datele membre sunt constituite din registrul de istorie generala **HRG**, tabela cu registrii de istorie locala, rețeaua neurala **nNetw** (variabila de tipul **CNetw**) si parametrii necesari simulării, **m\_params**, stabiliti prin intermediul interfeței grafice, de catre utilizator.

```
long unsigned HRG;
long unsigned HRLTable[1024];
CNetw nNetw;           // rețeaua neuronală
PARAMS m_params;
```

Constructorul clasei - **CNwSimul::CNwSimul(bool\* done,PARAMS m\_params)** - primeste parametrii de simulare **m\_params** si îi stocheaza în variabila locala corespunzătoare, precum si adresa unei variabile de tip **bool**, utilizata pentru oprirea simulării.

În afara de constructor si destructor clasa mai contine o singura functie:

**void CNwSimul::Simulate(CString file,CProjView\* view)** - care realizeaza simularea propriu-zisa. Functia primeste ca si parametru numele fisierului trace (*file*) si un pointer la fereastra atasata simulatorului unde utilizatorul poate urmări desfășurarea simulării si a rezultatelor pe grafic. **Executia** simulării decurge în felul urmator:

- ☐ este initializata rețeaua neurala prin apelul functiei *InitNet*;
- ☐ daca utilizatorul a selectat ca simularea trebuie precedata de o pre-învatare a rețelei neurale, conform metodei de învățare selectate tot de

 este efectuată simularea:

a) cele trei nivele ale rețelei împreună cu dimensiunile lor:

```
double in_Layer[maxdimInLayer];    // nivelul de intrare
double mid_Layer[maxdimMidLayer];  // nivelul ascuns
double out_Layer[dimOutLayer];     // nivelul de iesire
int    dimInLayer;                  // dimensiunea nivelului de intrare
int    dimMidLayer;                 // dimensiunea nivelului ascuns
```

b) cele doua nivele de ponderi împreuna cu termenii liberi asociati

```
double pond12[[]]; // ponderile dintre nivelul de intrare si cel ascuns
double pond23[[]]; // ponderile dintre nivelul ascuns si cel de iesire
double term12[];   // termenii liberi asociati primului nivel de ponderi
double term23[];   // termenii liberi de pe al 2-lea nivel de ponderi
```

c) pasul de învățare

```
double alpha;
```

d) alte variabile locale ajutatoare

În afara de datele membre clasa *CNetw* mai contine functiile ce constituie interfata publica a clasei precum si trei functii ajutatoare private astfel:

**A) functii ajutatoare:**

- *InitPonds()* initializeaza cu valori aleatoare toate ponderile si termenii liberi rețelei. Valorile aleatoare sunt luate în intervalul  $-2/x$  si  $2/x$  unde  $x$  este dimensiunea nivelului de intrare.
- *ReInitDeltaPonds()* reinitializeaza la 0 variabilele ce contin erorile calculate
- $f(\text{double } x)$  este functia de activare a rețelei ( $\frac{1}{1 + e^{-x}}$ ).

**B) functiile publice folosite la manipularea rețelei neurale:**

- *int InitNet(int dimHRG, int dimML, double alpha)* este folosita pentru initializarea parametrilor rețelei. Astfel *dimHRG* reprezinta dimensiunea în biti a registrului de istorie globala cu ajutorul caruia se va calcula dimensiunea nivelului de intrare ( $\text{dimInLayer} = 10 + \text{dimHRG}$ ), *dimML* este diferenta dintre dimensiunile primelor doua nivele, iar *alpha* este pasul de învățare al rețelei. De asemenea, sunt initializate si ponderile rețelei prin apelul functiei ajutatoare anterior descrisa *InitPonds()*.
- *void SetIns(unsigned pc, unsigned hrg, unsigned hrl)* codifica în binar parametrii receptionati. Rezultatele codificarii sunt concatenate si plasate pe nivelul de intrare al rețelei (*in\_Layer[]*).

- *void Forward()* efectueaza pasul „înainte” al rețelei care consta în calcularea valorilor nodurilor straturilor ascuns si de iesire în functie de valorile nodurilor de pe nivelul precedent si al ponderilor dintre ele.
- *void SetIdeals(unsigned pred)* stabileste valoarea ideala care se doreste a fi învatata de retea.
- *void Backward()* foloseste valoarea setata de functia precedenta pentru a efectua pasul „înapoi” al rețelei neurale. Acest pas este efectuat în doua etape: în prima sunt calculate corectiile aferente tuturor ponderilor, care vor fi stocate în *Dpond12* si *Dpond23*, în functie de valoarea dorita la iesire; a doua etapa consta în aplicarea acestor corectii ponderilor, bineînțeles înmultite cu pasul de învatare alpha, în vederea „adaptarii” rețelei.
- *unsigned Predict()* returneaza valoarea predictionata de catre retea dupa convertirea ei în binar astfel: daca valoarea de pe nodul de iesire este mai mare decât 0.5 se întoarce 1 iar daca este mai mica decât 0.5 se întoarce 0.
- *GetPonds(double p[])* este folosita pentru extragerea tuturor ponderilor rețelei neurale în sirul *p* primit ca parametru. Sirul trebuie sa aiba o dimensiune suficient de mare pentru a stoca toate ponderile.
- *SetPonds(double p[])* este folosita pentru initializarea ponderilor rețelei neurale cu valori ale sirului *p*.

C) Tot în cadrul interfetei publice a clasei au mai fost definite doua *functii* utilizate în *antrenarea rețelei*, daca utilizatorul o solicita, cu rezultatele statisticilor efectuate asupra trace-urilor. Cele doua functii sunt:

- *void SetLearningMaterial(IIS \*p, int dim)* este folosita pentru stocarea informatiei de învatare: un sir de structuri *IIS* de dimensiune *dim*.
- *double TrainingLevel()* antreneaza propriu-zis rețeaua dupa cum urmeaza:
  - ⊕ pentru fiecare element din materialul de învatare stocat cu functia de mai sus se executa:
    - ⊕ câmpurile *pc* si *hrg* ale elementului curent sunt plasate pe intrarile rețelei cu functia *SetIns()*.
    - ⊕ sunt calculate iesirile (*forward*).
    - ⊕ este plasata iesirea dorita, extrasa din câmpul *pred* al elementului curent.
    - ⊕ este calculata eroarea dintre valoarea asteptata si cea de pe iesirea rețelei.
    - ⊕ sunt modificate ponderile (*backward*).
    - ⊕ daca *eroare* este mai mare decât *eroarea maxima* obtinuta pâna acum o retinem pe aceasta.

⊕ în final functia returneaza eroarea maxima.

Ambele clase de antrenare a rețelei, *CClassicLearn* si *CGeneticLearn*, sunt derivate din acelasi stramos comun *CLearningAlg*. În vederea unui posibil extinderi pe viitor a metodelor de antrenament si a eliminarii redundantei de stocare a variabilelor locale s-a definit prin intermediul clasei abstracte *CLearningAlg* o interfata standard formata din functiile *Init()*, *Run()* si functia constructor. Astfel, orice alt algoritm care se doreste a fi utilizat va trebui sa fie o clasa derivata din *CLearningAlg* si sa implementeze metoda virtuala *Run()*, deoarece functia *Run()* a clasei *CLearningAlg* este abstracta. Cealalta functie a clasei *CLearningAlg*, *Init()*, împreuna cu constructorul clasei sunt folosite pentru initializarea variabilelor locale:

- ❖ *m\_params*, parametrii simularii.
- ❖ *m\_nntw*, rețeaua neurala antrenata.
- ❖ *view*, view-ul asociat simularii.
- ❖ *file*, numele fisierului trace.

Clasele care încapsuleaza algoritmii de antrenare fiind derivate din *CLearningAlg* vor mosteni toate aceste date membre împreuna cu functia de initializare *Init()* nemaifiind nevoie sa fie redeclarate.

Clasa *CClassicLearn* - implementeaza algoritmul de antrenare al rețelei denumit *clasic*. Conform acestui algoritm este efectuata o statistica asupra trace-urilor, în care sunt contorizate numarul de salturi efectuate si numarul de salturi neefectuate corespunzatoare perechilor (), câte o statistica pentru fiecare pereche. Aceste statistici sunt apoi filtrate rezultând materialul folosit pentru antrenarea rețelei. În vederea obtinerii statisticilor sunt definite doua functii membre *BuildStatistics(CString file)* si *FilterStatistics()* precum si o serie de date membre în care sunt stocate rezultatele statisticii:

- ❖ *IISTNT PCs[maxdimPC]* este un sir de statistici, câte una pentru fiecare pereche (PC,HRg) întâlnita.
- ❖ *dimPC* dimensiunea vectorului *PCs*.
- ❖ *IIS finalPC[maxdimPC]* va contine statistica dupa filtrare.
- ❖ *nrfinPC* dimensiunea vectorului *finalPC*.
- ❖ functia *BuildStatistics(CString file)* construiește statistica, pe fisierul trace al carui nume este primit ca parametru, astfel:
  - *atâta timp cât nu sa întâlnit sfârșitul fisierului executa*
    - este citita din trace o intrare de forma (*instr,pc,target*)
    - daca perechea (*pc,HRG,HRLTable[pc]*) nu mai exista este adaugata o noua intrare în vectorul *PCs*.

- dacă în schimb perechea a mai existat pe poziția corespunzătoare din *PCs* se contorizează fie un salt efectuat fie unul neefectuat.
- este actualizat registrul de istorie generală *HRG*.
- este actualizat registrul de istorie locală *HRLTable[pc]*.
- ❖ funcția *FilterStatistics()* filtrează statistica obținută mai sus prin eliminarea intrărilor din *PCs* la care nici numărul de salturi efectuate și nici al celor neefectuate nu depășește un anumit procent stabilit de utilizator. Rezultatul filtrării este stocat în vectorul *finalPC*.
- ❖ funcția *Run()*, obligatoriu implementată din motivele expuse mai sus, conține doar apeluri la funcțiile *BuildStatistics* și *FilterStatistics*. O dată statistica obținută și filtrată este transmisă rețelei prin apelul funcției *SetLearningMaterial(finalPC,nrfinPC)*. Urmează antrenarea rețelei folosind funcția *TrainingLevel()* până când eroarea returnată de precedentă funcție scade sub o anumită valoare stabilită de utilizator (*m\_params.acurate*).

Clasa ***CGeneticLearn*** - conține întregul mecanism pentru antrenarea unei rețele neurale folosind un *algorithm genetic*. Conform celor prezentate în partea teoretică este nevoie de o populație de 50 de cromozomi, fiecare cromozom conținând un număr de gene. Aceste informații sunt stocate în vectorul bidimensional *gene*. De asemenea, au fost implementate funcții pentru fiecare operație specifică algoritmilor genetici:

- ❖ pentru evaluarea cromozomilor funcția *calcEvals()*. În cadrul acestei funcții genele corespunzătoare fiecărui cromozom sunt introduse ca ponderi într-o rețea neurală în vederea predicției salturilor dintr-un trace. Rezultatul, *acuratetea predicției*, va constitui evaluarea cromozomului respectiv. Valorile obținute în urma evaluărilor vor fi stocate în vectorul *Eval*.
- ❖ pentru încrucișarea a doi cromozomi funcția *Reproduce()*, pentru încrucișarea tuturor cromozomilor funcția *Cross()* și *Mutate()* pentru efectuarea de mutații.

În final este implementată și funcția *Run()* în care pentru un număr de generații, stabilit de utilizator, sunt calculate evaluările cromozomilor (*calcEvals()*), cromozomii sunt încrucișați (*Cross()*), și apoi li se aplică operatorul genetic de mutație (*Mutate()*). Înainte de a se încheia funcția *Run()* este selectat cromozomul cel mai bun și genele sale aplicate în locul ponderilor rețelei neurale (*m\_nntw->SetPonds(gene[index])*).

În continuare sunt prezentate structurile de date în care vor fi stocate informații ajutoare necesare simulării și învățării. Toate aceste structuri se găsesc definite în fișierul header *hh.h*. Structura ***PARAMS*** este definită astfel:



```

struct PARAMS
{
    int dimHRG;
    double pasInv;
    int DSecLayer;
    double acurate;
    unsigned proc;
    int ant;
    int met;
    CString files[8];
    int nr_files;
};

```

Aceasta structura este utilizata pentru stocarea si transmiterea parametrilor variabili ai simularii. Astfel, *dimHRG* reprezinta dimensiunea în biti a registrului de istorie globala, *pasInv* – este pasul de învățare al rețelei neurale, *DSecLayer* reprezinta diferenta dintre numarul de noduri al primului nivel si numarul de noduri al celui de al doilea; *acurate* este acuratetea cu care se va efectua procesul de învățare (are semnificatie doar daca *ant* si *met* iau valoarea 0); *ant*– specifica daca se realizeaza învățarea sau nu (0 – pentru învățare si 1 – pentru simulare fara învățare); *met* – reprezinta metoda de învățare utilizata (0 – metoda clasica, 1 – metoda genetica); *files* este un vector cu numele fisierelor trace iar *nr\_files* este dimensiunea acestui vector.

Structura *IISTNT* este utilizata pentru stocarea informatiilor obtinute la realizarea statisticilor din cadrul procesului de învățare clasica.

```

struct IISTNT
{
    unsigned pc;
    unsigned hrg;
    unsigned hrl;
    unsigned long taken;
    unsigned long ntaken;
};

```

Semnificatia fiecarui element din aceasta structura este urmatoarea: câmpurile *pc*, *hrg* si *hrl* reprezinta elementele identificatoare ale instructiunii de salt, adresa si context global / local; *taken* – numarul de instructiuni de salt efectuate având adresa *pc* iar *ntaken* este numarul de salturi neefectuate.

Structura *IIS* este utilizata pentru stocarea informatiilor rezultate în urma filtrarii statisticilor calculate anterior. Acestea vor constitui informatiile de intrare în cadrul procesului de antrenare a rețelei neurale.

```
struct IIS
{
    unsigned pc;
    unsigned hrg;
    unsigned hrl;
    unsigned pred;
};
```

Câmpurile *pc*, *hrg* si *hrl* au aceeași semnificație ca la structura *IISTNT* iar *pred* este rezultatul ce se dorește a fi „învătat” (*taken/not taken*) de către rețea, în condițiile reapariției instrucțiunii cu adresa *pc* și în același context de istorie globală sau locală determinat de componentele *hrg* și *hrl*.

#### 7.3.4. GHID DE UTILIZARE

Interfața cu utilizatorul permite interacționarea dintre acesta și programul de aplicație prin intermediul următoarelor resurse software: un *meniu*, o *bara de instrumente* (toolbar), o *bara de stare*, o *bara de setări* precum și *ferestra în care se afla graficul rezultatelor simulării*. O sesiune de simulări trebuie începută prin apăsarea *butonului New*, fie de pe *toolbar* fie din *meniu*. Aceasta va avea ca efect crearea unei ferestre copil în cadrul ferestrei principale. Pasul următor înainte de începerea simulărilor este stabilirea parametrilor de pe *bara cu setări*. O sesiune de simulare va consta în efectuarea simulării asupra tuturor fișierelor trace selectate. Odată stabiliți acești parametri simularea este startată prin apăsarea *butonului Play*. Oprirea prematură a simulării poate fi efectuată de pe *butonul Stop*. Simultan pot rula mai multe sesiuni de simulare, fiecare cu parametrii ei, independent de celelalte (avantajul arhitecturii MDI Document / View).

Rezultatele simulării vor fi putea fi stocate pe disc în fișiere prin apăsarea *butonului Save*. După specificarea locației și a numelui va fi creat un fișier având următoarea structură:

- o prima linie de identificare a fișierului care conține sirul de caractere „*Fisier de salvări*”
- o zonă de parametri în funcție de opțiunile alese de către utilizator care începe cu sirul „*Parametrii*”. Indiferent de opțiunile utilizatorului vor fi salvați parametrii **KG** (dimensiunea registrului de istorie globală) și

**alfa** (pasul de învățare). Apoi în funcție de parametrii aleși se va înscrie dacă simularea s-a făcut cu antrenare sau fără. Dacă a avut loc antrenarea se înscrie și metoda de antrenare. În funcție de metoda de antrenare selectată se vor înscrie și parametrii specifici acesteia astfel: pentru metoda clasică se mai înscrie acurătatea de învățare și procentajul de filtrare iar pentru metoda genetică numărul de generații. Actuala variantă de simulator prezintă din păcate un *bug*, care poate fi verificat dacă se vizualizează codul sursă al aplicației (evoluția parametrului *m\_params.met* – metoda de învățare – în fișierele *projDoc.cpp* și *NwSimul.cpp*), și anume că, la salvarea în fișierul cu rezultate, dacă s-a optat pentru antrenarea rețelei, sunt scrise gresit metodele de antrenare a rețelei: la antrenarea clasică se scrie metoda genetică și parametrii săi și invers pentru antrenarea genetică.

- zona de date propriu-zisă care conține rezultatele simularilor (*numărul de predicții corecte, incorecte și procentul de acurătate obținut*) debutează cu sirul de caractere „Date”

În continuare vor fi prezentate individual componentele interfeței.



**Figura 7.10.** Bara de instrumente cuprinde butoane de comandă

**Toolbar-ul** – aflat în partea de sus a ferestrei principale conține 6 butoane după cum urmează:

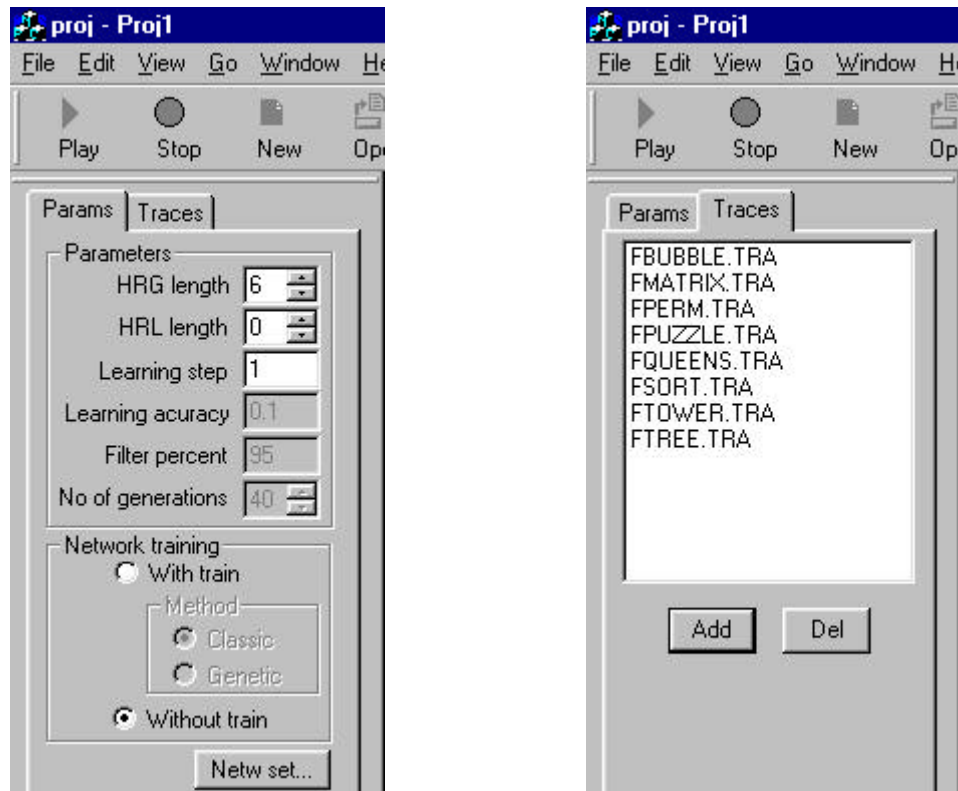
- butonul *Play* startează simularea.
- butonul *Stop* oprește o simulare în curs de desfășurare după ce utilizatorul a confirmat această acțiune.
- *New* deschide o nouă sesiune de simulare (*view*) inițializând parametrii simulării cu valorile implicite.
- *Open* deschide un fișier cu rezultate anterior salvat. La o nouă apăsare pe butonul *open*, având aceeași fereastră copil activă ca și mai înainte, noul fișier deschis va fi afișat peste cel anterior în vederea comparării rezultatelor.
- *Save* salvează rezultatele simulării într-un fișier specificat de utilizator în structura prezentată anterior.

Toate aceste butoane se pot găsi de asemenea și în meniu sub aceleași denumiri.

**Bara de setări** – oferă utilizatorului posibilitatea de a alege tipul de simulare dorit și pentru fiecare simulare parametrii specifici ai acesteia.

Bara de setari contine doua *pagini de proprietati*: pagina cu parametrii si cea cu fisierele trace.

Pagina „**Traces**” ofera posibilitatea de a selecta ce fisiere de tip *trace* vor intra în cadrul procesului de simulare. Prin intermediul butonului **Add** pot fi adaugate noi *trace*-uri iar cu ajutorul butonului **Del** pot fi sterse (nu fizic ci doar din punct de vedere a simularii).



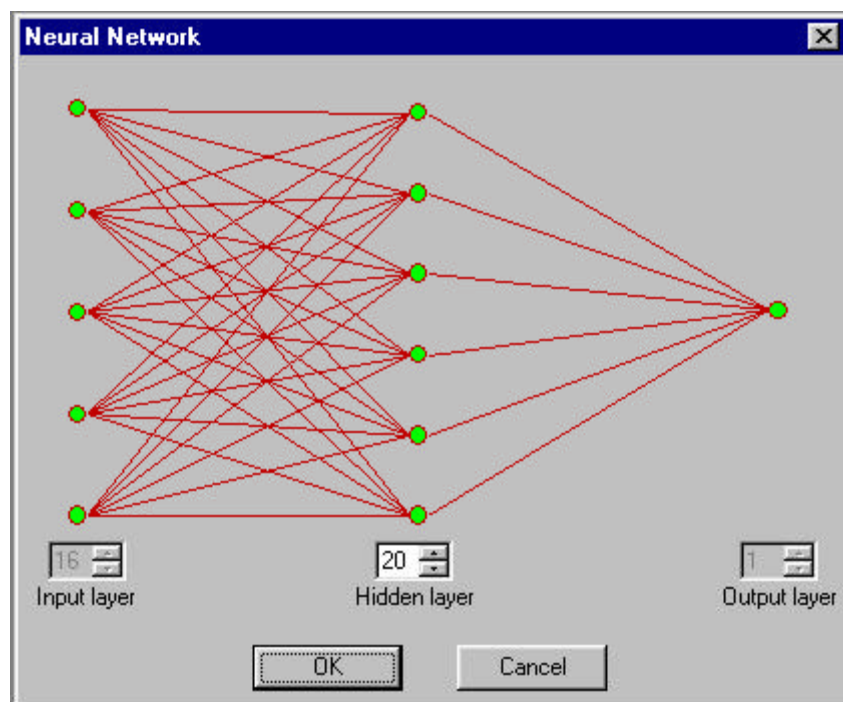
**Figura 7.11.** Configurarea predictorului neural

În pagina cu parametrii se poate alege din zona **Network Training** daca dorim sau nu ca rețeaua sa fie antrenata înainte de simularea propriu-zisa. Dacă am ales *butonul radio* „**With train**” mai trebuie sa alegem si metoda de antrenare, putând opta deocamdata între antrenare „**Classic**” si „**Genetic**”. În functie de metoda de antrenare aleasa se vor activa si parametrii specifici din zona „**Parameters**”.

Zona „**Parameters**” contine controlul *spin de incrementare/decrementare* în care se specifica dimensiunea registrului de istorie globala (HRG), un control de *editare* pentru introducerea pasului de învățare. Pe lângă acești parametri care sunt independenți de metoda de

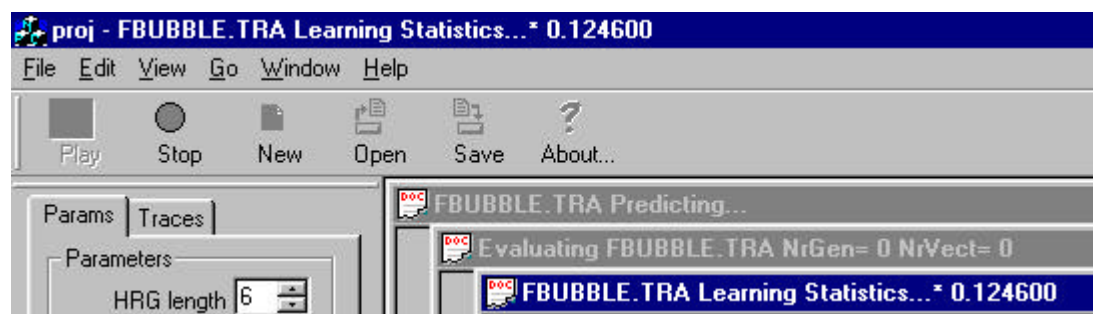
antrenare mai avem parametrii *acuratete învățare* și *filtru* care au semnificație doar în contextul selectării metodei clasice de antrenare. Acești parametri se referă la acuratetea cu care trebuie făcută antrenarea și la procentajul de filtrare a statisticilor din cadrul metodei clasice de antrenare. Parametrul „**No. of generations**” are semnificație doar în contextul selectării metodei genetice de antrenare și specifică numărul de generații parcurs la antrenarea rețelei. Valorile implicite ale parametrilor prezentați sunt fixate ca în figura precedentă, la crearea unei noi sesiuni.

Tot în cadrul primei pagini a barei de setări mai există *butonul* „**Netw Set...**” la apăsarea căruia este afișată fereastra din figura următoare, fereastra în care poate fi specificat numărul de noduri de pe nivelul ascuns. Numărul de noduri de pe nivelul de ieșire este fixat la 1 iar numărul de noduri de pe nivelul de intrare este egal cu  $10 + \text{dimensiunea HRG}$  stabilită anterior.



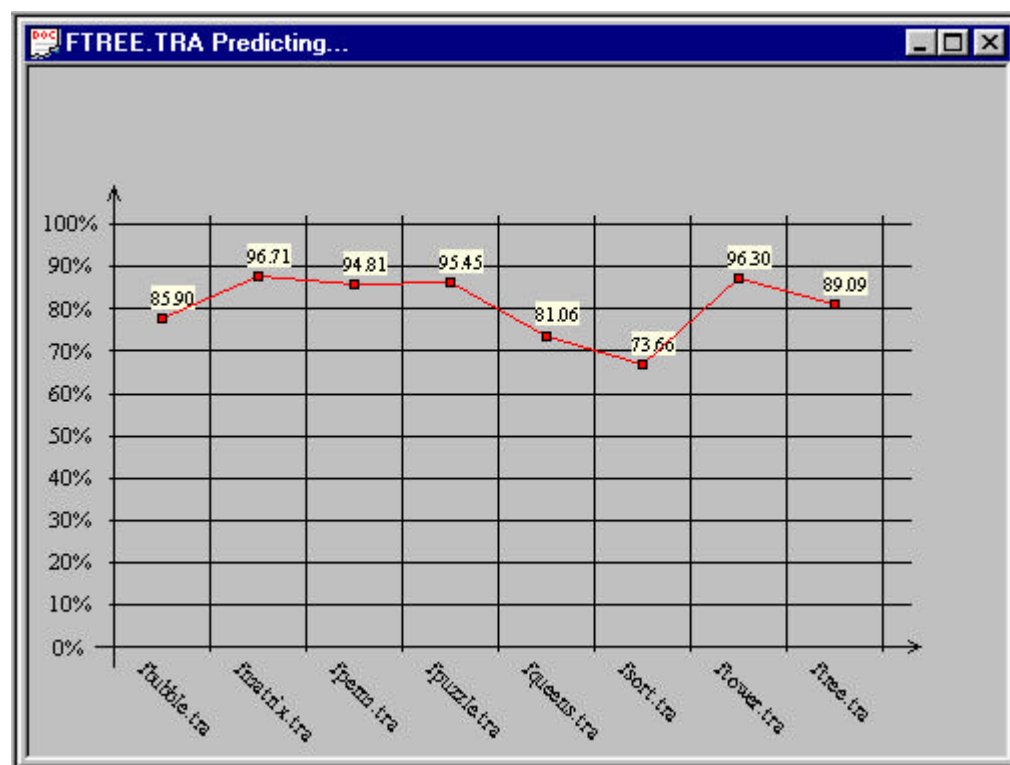
**Figura 7.12.** Stabilirea numărului de noduri de pe nivelul ascuns

Parametrii odată stabiliți și simularea începută, programul permite utilizatorului să urmărească diferitele etape (crearea de statistici, filtrare, învățare, predicție) în bara de titlu a aplicației sau a ferestrei copil.



**Figura 7.13.** Instantaneu cu simularea: predicție fara antrenare, cu preînvedere genetică și învățarea statisticilor

Astfel fiecare fereastră copil va afișa propriul *progres* iar fereastra principală va afișa progresul ferestrei copil activă curent. Rezultatele simulărilor (valorile acurateți de predicție corespunzătoare fiecărui trace (\*.tra)) vor putea fi observate pe graficul desenat în cadrul ferestrelor copil.



**Figura 7.14.** Rezultatele unei simulări sub formă grafică

## 7.4. PROBLEME PROPUSE SPRE REZOLVARE

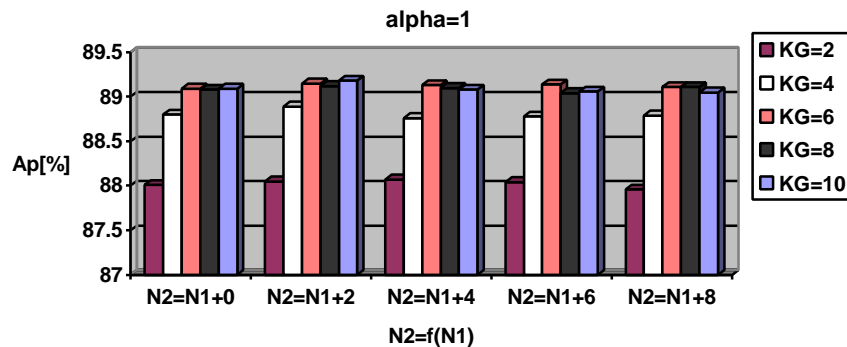
Cu ajutorul simulatorului *branchees.exe* determinati:

1. Care este numarul optim de noduri de pe nivelul intermediar în functie de istoria globala  $KG \in \{2,4,6,8,10\}$  a saltului pentru fiecare din pasii de învățare stabiliți ( $\alpha \in \{0.125, 0.5, 1.00\}$ ) și  $KL=0$ . Functia de activare utilizata în cadrul simulatorului este  $f(x) = \frac{1}{1+e^{-x}}$ . Concluzionati.
2. Cu numarul de noduri de pe nivelul ascuns stabilit determinati pasul optim de învățare în functie de istoria globala a saltului.
3. Realizati comparativ graficul acuratetii de predictie în conditiile ne-antrenari rețelei și antrenari clasice (cu preînvățare pe baza analizei statistice, urmata de predictie).
4. În conditiile determinarii numarului optim de noduri de pe nivelul ascuns ( $N2=N1+2$ ) determinati procentul optim de filtrare a statisticilor, folosit în cadrul metodei de antrenare clasica a rețelei neuronale în functie de istoria globala ( $KG \in \{2,4,6,8,10\}$ ) a saltului. Pragul de filtrare poate lua valorile: 60%, 70%, 80%, 90%, 95%.
5. Exemplificati grafic câștigul de performanta (din punct de vedere al acuratetii de predictie obtinute la punctul 1) fata de cele doua scheme de predictie studiate anterior (BTB și GAg) în conditii echivalente ( $HRg=0$  pentru comparatia cu BTB, automate de predictie pe un bit).

### 7.4.1. REPREZENTAREA GRAFICA A REZULTATELOR SIMULARILOR EFECTUATE

Se va urmări ca rezultatele simulării să respecte formatul ilustrat mai jos. De asemenea, se vor efectua simulări pe toate cele 8 benchmark-uri Stanford avute la dispoziție. Principala metrica folosită este acuratetea de predictie (**Ap**).

1. Care este numarul optim de noduri de pe nivelul intermediar în functie de istoria globala ( $KG$ ) a saltului pentru fiecare din pasii de învățare stabiliți (**a**) fara a retine o tabela cu istoria locala a salturilor.



**Figura 7.15.** Rezultatul grafic al simulării de la punctul 1)

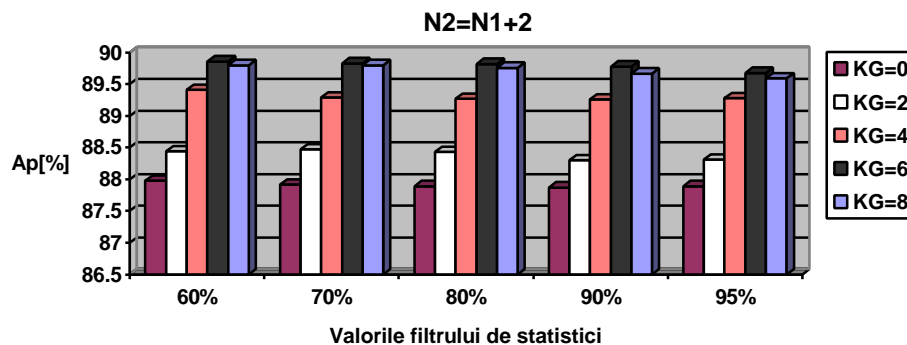
3) În condițiile determinării numărului optim de noduri de pe nivelul ascuns ( $N_2=N_1+2$ ) determinați procentul optim de filtrare a statisticilor folosit în cadrul metodei de antrenare clasică a rețelei neuronale în funcție de istoria globală (KG) a saltului.

KG	Bubble	Matrix	Perm	Puzzle	Queens	Sort	Tower	Tree	global
0	85.81	96.71	88.58	94.28	79.73	72.72	96.97	89.00	87.98
2	84.51	96.71	89.18	95.22	79.65	75.70	96.98	89.59	88.44
4	85.20	96.71	93.37	95.73	81.75	75.72	97.02	89.80	89.41
6	85.89	96.71	94.95	95.78	82.51	76.60	96.69	89.75	89.86
8	86.30	96.71	94.33	95.73	82.45	76.38	96.59	89.87	89.80

*Tabelul 7.1.*

**$A_p=f(KG)$  în condițiile  $N_2=N_1+2$ . Filtru=60%.**

Tabelul 7.1. va fi continuat cu alte patru tabele care cuprind rezultatele simulării ( $A_p$ ) pentru parametru Filtru luând valorile: 70%, 80%, 90%, 95%, funcție de dimensiunea registrului de istorie globală.



**Figura 7.16.** Rezultatul grafic al simulării de la punctul 3)



## 8. DEZVOLTAREA DE SIMULATOARE SUB MEDIUL *SIMPLESCALAR* UTILIZÂND *BENCHMARK-URILE SPEC*

---

### 8.1. SETUL DE INSTRUMENTE - "*SIMPLESCALAR 3.0*"

#### 8.1.1. CE ESTE "*SIMPLESCALAR TOOL SET*" ?

Reprezinta o colectie de instrumente software, pusa la dispozitia cercetatorilor în arhitecturi moderne de calcul, si cuprinde: compilatoare, asamblatoare, link-editoare, depanatoare (debugger-e), simulatoare si instrumente de vizualizare a unei arhitecturi (super)scalare simple, generice (Exemple: arhitecturi **PISA**, **Alpha AXP**, **ARM**) [Bur97]. Arhitectura Alpha AXP este un procesor RISC dezvoltat de firma DEC (devenita ulterior COMPAQ, actualmente HP), iar arhitectura SimpleScalar PISA (Portable ISA) se va detalia mai târziu (vezi **8.1.2**). Setul mai cuprinde un depanator la nivel de cod sursa al benchmark-ului simulat (**DLite!**) si un generator de trace-uri la nivel de structura pipeline a instructiunilor (aferent deocamdata doar celui mai detaliat simulator). Setul de instrumente "*SimpleScalar*" este distribuit gratuit (poate fi gasit si descarcat pe site-ul web "<http://www.cs.wisc.edu/~mscalar/simplescalar.html>") si ofera posibilitatea simularii de tip *execution driven*, cât mai detaliate si de înalta performanta, a celor mai moderne microprocesoare. Programele simulate reprezinta parte integranta a setului de instrumente si sunt programe de test precompilate (format cod obiect) pentru arhitecturile PISA si Alpha respectiv o parte din benchmark-urile SPEC'95. Cei interesati pot dispune de compilatorul GNU GCC (precum si de utilitarele aferente), care permite fiecarui cercetator sa compileze/asambleze/linkediteze propriile programe de test scrise pentru arhitectura SimpleScalar. Setul de instrumente

"SimpleScalar" si modulele de portare ale compilatorului GNU pe diverse platforme a fost scris de Todd Austin - începând cu anul 1994 pe când era cercetator la universitatea din Wisconsin-Madison, actualmente fiind membru al echipei de cercetatori în paralelism la nivelul instructiunilor al firmei Intel Corporation. "SimpleScalar" este sustinut în continuare de Doug Burger (autorul, în cea mai mare parte, a documentatiei) si de Todd Austin. Compilatorul GNU si instrumentele software aditionale (biblioteci, translatoare din Fortran în C) a fost scris de "Free Software Foundation".

### 8.1.2. AVANTAJELE UTILIZARII "SIMPLESCALAR TOOL SET"

⇒ **Distribuirea gratuita**, inclusiv a surselor C a tuturor modulelor componente. Printre adresele de web utile se numara:

- <ftp://ftp.cs.wisc.edu/sohi/Code/SimpleScalar/simplesim.tar>
- <http://www.cs.wisc.edu/~mscalar/simplescalar.html>

La prima adresa mentionata pot fi gasite pe lângă sursele simulatoarelor si urmatoarele fisiere în format arhiva, fiecare cuprinzând anumite elemente (necesare sau optionale) ale setului:

- ☐ **Simplesim.tar.gz** - cuprinde sursele simulatoarelor procesorului virtual SimpleScalar, scripturi si macrodefinitii ale setului de instructiuni, sursele C si cod obiect (obtinute dupa compilare/asamblare/linkeditare) ale unor mici programe de test. Este necesar pentru instalarea setului de instrumente si include o documentatie exhaustiva asupra setului SimpleScalar (**hack\_guide.pdf** sau **hack\_guide.ps**).
- ☐ **Simpleutils.tar.gz** - contine sursele unor utilitare GNU (versiunea 2.5.2) portate pe arhitectura SimpleScalar. *Nu sunt necesare pentru executia simulatoarelor dar sunt necesare la asamblarea si link-editarea propriilor benchmark-uri scrise pentru arhitectura SimpleScalar.*
- ☐ **Simpletools.tar.gz** - contine sursele si bibliotecile compilatorului GNU (*gcc* 2.6.3, *glibc* 1.0.9 si translatorul din Fortran în C *f2c*) necesare compilarii benchmark-urilor proprii pentru arhitectura SimpleScalar (codul rezultat este format mnemonica de asamblare). *Nu sunt necesare pentru executia simulatoarelor.*

- ☞ **Simplebench.big.tar.gz** - cuprinde o serie de benchmark-uri SPEC'95, în cod obiect, compilate pentru arhitectura SimpleScalar rulând pe o statie cu ordinea octetilor *big endian*.
- ☞ **Simplebench.little.tar.gz** - cuprinde aceleasi benchmark-uri SPEC'95, în cod obiect, compilate pentru arhitectura SimpleScalar rulând pe o statie cu ordinea octetilor *little endian*.

Ordinea octetilor reprezinta o conventie de numerotare de catre procesor a octetilor din interiorul unui cuvânt de 32 de biti astfel încât octetul cu numarul cel mai mic este fie cel mai din stânga fie cel mai din dreapta. Procesoarele MIPS pot opera fie cu ordinea octetilor în ambele moduri.

*big-endian*

Byte #			
0	1	2	3

*little-endian*

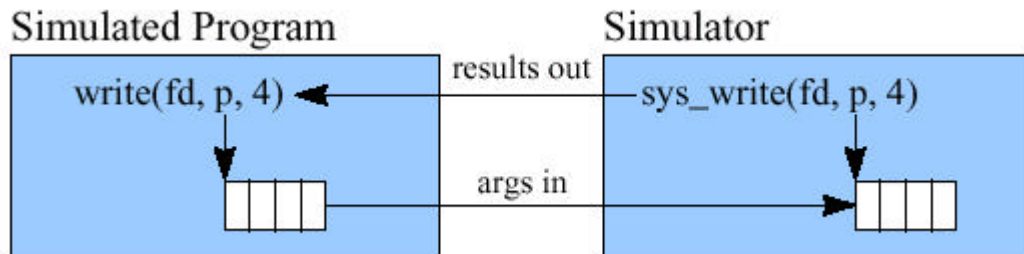
Byte #			
3	2	1	0

sau

- ⇒ **Flexibilitatea sporita** (plaja larga de valori a parametrilor simulatoarelor precum si multimea de simulatoare de arhitecturi dezvoltate, începând cu unul functional foarte rapid dar extrem de simplu, cu executie *in order* - **sim-fast** - si pâna la unul extrem de detaliat, cu executie *out of order* si *speculativa*, dotat cu un sistem ierarhizat de memorii multinivel si un predictor avansat ("*state of the art*") aferent instructiunilor de salt- **sim-outorder**).
- ⇒ **Portabilitatea:** simulatoarele ruleaza pe majoritatea platformelor UNIX pe 32 si 64 de biti si chiar pe platforma WinNT (sursele sunt compilate sub MS Visual C++), cu conditia ca uneltele software GNU sa fie instalate pe calculatorul gazda. Pentru modificarea/compilarea/link-editarea simulatoarelor/benchmark-urilor proprii poate fi folosit emulatorul Cygwin. Majoritatea utilizatorilor si dezvoltatorilor setului SimpleScalar lucreaza pe sisteme de operare Linux pe masini x86.
- ⇒ **Extensibilitatea:** pachetul SimpleScalar Tool Set cuprinde toate sursele permitând dezvoltarea ulterioara a setului - îmbunatatirea, atasarea de noi module (Exemple: simulator pentru predictia valorilor instructiunilor, reutilizarea dinamica a instructiunilor, trace cache); este bine documentat. Setul de instructiuni proiectat suporta eventuale *adnotari* (câmp suplimentar) - modificari post-compilare - în fisierele asamblare, fara a fi necesara o recompilare a codului masina.

### 8.1.3. DEZVANTAJE ÎN UTILIZAREA SETULUI DE INSTRUMENTE "SIMPLESCALAR"

- ❑ **Suporta doar seturile de instructiuni a anumitor arhitecturi:** SimpleScalar PISA si Alpha AXP, ARM7ISA.
- ❑ **SimpleScalar simuleaza doar medii uniprocessor;** în functie de benchmark-ul simulat (si numarul de instructiuni) simularea poate dura foarte mult ( $\approx 36h$  pe calculatoare Pentium II 400MHz, 64MB RAM).
- ❑ **Simularea instructiunilor se realizeaza la nivel de utilizator** (nu este simulata executia instructiunilor în interiorul sistemului de operare). Protocolul de tratare a **apelurilor sistem** (întreruperi) este urmatorul:
  - ❑ Decodificarea apelului sistem.
  - ❑ Copierea argumentelor (daca exista) în memoria simulatorului (**sim-\***).
  - ❑ Executarea apelului sistem de catre sistemul de operare (rutina de tratare a întreruperii).
  - ❑ Copierea rezultatelor daca exista în memoria programului simulat (**benchmark-ul SPEC**).



**Figura 8.1.** Protocolul de tratare a **apelurilor sistem**

- ❑ Modulul **syscall.c** implementeaza un subset de apeluri sistem specifice Unix. Pentru adaugarea oricarui nou apel sistem sau pentru portarea SimpleScalar pe un sistem de operare nou, este necesara implementarea software a rutinei de tratare în modulul **syscall.[hc]**.

### 8.1.4. CE ESTE "GNU" ? UTILITARE GNU.

GNU reprezinta abrevierea (recursiva si deci tautologica a) expresiei "GNU's Not Unix" si se pronunta în engleza "guh-NEW". Proiectul GNU a startat în anul 1984 cu scopul de a dezvolta un sistem de operare asemanator UNIX-ului dar care sa fie distribuit gratuit oricarui individ care doreste sa-l utilizeze sau sa-l dezvolte în continuare. Sistemul de operare GNU nu

reprezintă o colecție de instrumente software proprii, specifice doar GNU, ci cuprinde și alte programe existente ca "*free software*". Nucleul sistemului de operare este LINUX. Editorul de text folosit este TeX, realizat de Donald Knuth. Interfața bazată pe ferestre este identică cu cea a sistemului XWindows, realizată de Bob Scheifler. Compilatorul (*gcc*), asamblorul (*gas*) și link-editorul (*gld*) sunt proprii GNU. Pentru ca sistemul de operare să fie complet sunt necesare pe lângă instrumente de programare și un interpretor PostScript, biblioteci de C etc.

Chiar dacă nu există nici un avantaj tehnic al GNU asupra Unix, există totuși un avantaj social, permițând utilizatorilor să coopereze la utilizarea și dezvoltarea continuă a sistemului de operare și unul etic prin respectarea libertății individului. Creșterea în fiabilitate și viteză a unor componente (programe) GNU față de cele similare Unix s-a obținut prin tratarea diferită a fișierelor. Astfel, fișierele de dimensiuni foarte mari sunt încărcate în întregime în memoria principală, fără a mai avea probleme din punct de vedere al interfetelor de intrare/ieșire la citirea (parcursarea) secvențială a conținutului.

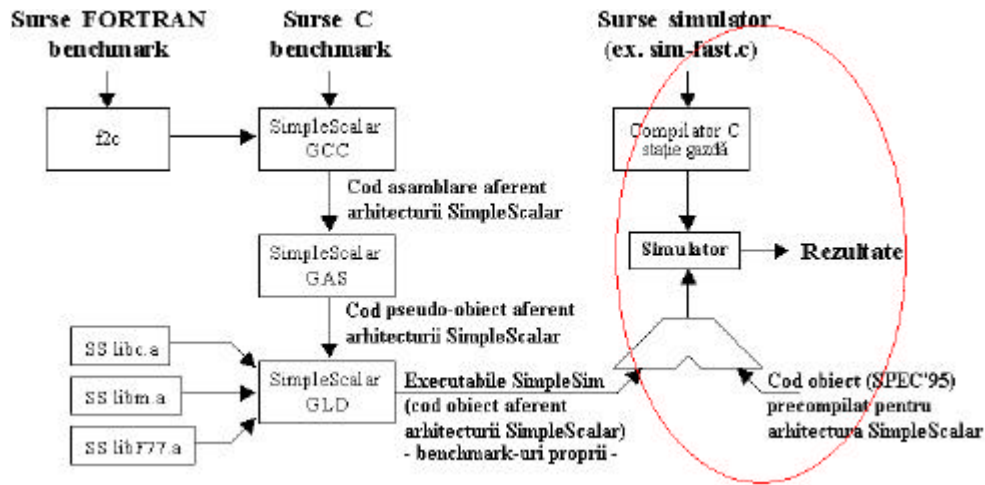
**GNOME** (GNU Network Object Model Environment) reprezintă varianta desktop a proiectului GNU. Început de Miguel de Icaza în 1997 și continuat cu sprijinul companiei Red Hat Software, GNOME stabilește o serie de facilități la nivel desktop, dar folosind programe software exclusiv disponibile gratuit. Prezintă avantaje tehnice precum suportul unei varietăți de limbaje (nu doar C++).

Dintre utilitățile GNU amintim câteva și rolul îndeplinit de fiecare din ele:

- **ld** - link-editor GNU.
- **as** - asamblorul portabil GNU.
- **ar** - utilitar folosit la crearea, modificarea și extragerea din arhive.
- **objcopy** - copiază și translatează fișiere obiect.
- **objdump** - afișează informații din fișiere obiect.
- **size** - listează dimensiunea unei secțiuni a unui fișier obiect sau fișier arhivă.
- **strings** - listează sirurile de caractere care pot fi tipărite din fișiere.
- **windres** - compilator pentru fișiere de resurse Windows.
- **gprof** - afișează informații de profil.

Majoritatea acestor programe utilizează **BFD** ("*Binary File Descriptor*" - bibliotecă binară a descriptorilor de fișiere), pentru a realiza o manipulare a fișierelor "*low-level*". Utilitățile GNU pot fi portate pe majoritatea variantelor de UNIX precum și pe sistemele Windows având procesoare Intel.

Etapale parcurse pornind de la sursa HLL (*High Level Languages*) a programelor de test si pâna la simularea de tip *execution driven* sunt evidentiata în figura urmatoare.



**Figura 8.2.** Interactiunea instrumentelor în cadrul setului SimpleScalar

Sursele FORTRAN ale programelor de test proprii sunt convertite în C folosind translatorul `f2c`. Atât benchmark-urile C cât si cele convertite din FORTRAN sunt compilate cu ajutorul compilatorului GNU `gcc` (dedicat arhitecturii SimpleScalar) rezultând cod în format de asamblare pentru respectiva arhitectura. Codul rezultat este trecut prin asamblorul SimpleScalar `gas` care genereaza un format foarte asemanator cu codul obiect dar nu identic (cuprinde simboluri + cod obiect). Formatul final de cod obiect se obtine cu ajutorul utilitarului SimpleScalar `gld` care link-editeaza codul rezultat la pasul anterior (*pseudo-obiect*) cu bibliotecile proprii arhitecturii SimpleScalar (`SS libc.a`, `SS libm.a`, `SS libF77.a`). Codul obiect se constituie ca parametru de intrare pentru simulatoarele arhitecturii SimpleScalar. Setul de instrumente SimpleScalar pune la dispozitie si câteva dintre benchmark-urile SPEC '95 în acelasi format cod obiect (precompilate pentru arhitectura SimpleScalar). Daca nu se dispune de simulatorul dorit în format executabil, ci doar de sursa C a acestuia, atunci acesta trebuie compilat cu ajutorul compilatorului disponibil pe statia gazda (orice compilator de ANSI C). Simularea se face foarte detaliat, la nivel de ciclu de executie al procesorului, rezultatele generate fiind continutul resurselor, gradul de încărcare al acestora, rate de procesare, de hit, acurateti de predictie, informatii de profil etc.

### 8.1.5. INSTALAREA SETULUI DE INSTRUMENTE "SIMPLESCALAR". GHID DE UTILIZARE.

Pentru început se prezintă pașii efectuați în instalarea setului de instrumente *SimpleScalar 3.0* pornind de la arhiva **simplesim-3.0b.tar.gz** disponibilă (pe CD-ROM) sau la adresa <http://www.cs.wisc.edu/~mscalar/simplescalar.html> și de la kit-ul de instalare **Cygwin** (aferent emulatorului de Linux pe sistemul de operare WinNT). Pașii descriși în continuare sunt valabili doar pe sistemele de operare Windows 2000 Profesional și Windows NT 4.0.

- ❏ Se descarcă de la adresa amintită anterior fișierul arhivă în format **tar.gz** "*simplesim-3.0b.tar.gz*". Se extrage din acesta conținutul său: directorul **SIMPLESIM-3.0/\*.\*** cu fișierele componente, dintre care și directorul **target-pisa**. Fișierele acestui director (care descriu funcționarea unei arhitecturi **pisa**) sunt copiate în directorul părinte(SIMPLESIM-3.0).
- ❏ Se instalează emulatorul de linux CYGWIN. După instalarea corectă se execută dublu-click pe iconița aplicației CYGWIN (sau *Start -> Programs -> Cygnus Solutions -> Cygwin Bash Shell*) și se intră în prompter-ul de Linux. În acest moment (la primul acces după instalare) se generează automat un director **home** și un subdirector cu numele **user**-ului de pe stația respectivă (fie **Administrator**) în directorul CYGWIN. În directorul **home/Administrator** se copiază SIMPLESIM-3.0 cu tot conținutul său.
- ❏ Revenind în CYGWIN se tastează comanda **\$cd SIMPLESIM-3.0** și se ajunge în directorul tocmai copiat. Se tastează în continuare succesiunea de comenzi:

**\$make config-pisa**

și

**\$make**

Prima comandă determină instalarea componentelor (în vederea compilării instrumentelor setului pentru arhitectura *pisa*). Executia comenzii **make** poate genera o eroare la compilare deoarece tipul returnat de funcția *myrand()* din fișierul **misc.c** care apelează la rândul ei funcția *random()* diferă de cel returnat de funcția standard definită în **stdlib.h** (din **Cygwin\usr\include**). Eroarea se corectează modificând tipul funcției *random()* în **misc.c** (din **long** în **int**). Executia din nou a comenzii **make** se va încheia cu mesajul "**My work is done here...**" ceea ce demonstrează compilarea / asamblarea / link-editarea cu succes a simulatoarelor setului *SimpleScalar 3.0*.

- ☐ Înainte de simularea propriu-zisa (lansarea în executie a unuia din fisierele **sim-\*.exe**) trebuie copiat fisierul **cygwin1.dll** din **Cygwin\bin** în directorul **Cygwin\home\Administrator\Simplesim-3.0**. În acelasi director se vor copia si benchmark-urile SPEC (executabilele si intrarile corespunzatoare). În caz contrar în momentul simulării trebuie specificata calea spre programele de test folosite în simulare.

Pentru rularea unui program de test (se considera directorul curent ca fiind cel ce contine simulatorul - executabilul) se tasteaza secventa:

- ☐ din **prompter de DOS** sau **fereastră RUN din WinNT**:

`<Nume_simulator> <Cale_benchmark> [optiuni]`

- ☐ din **Linux** (sau **Cygwin**):

`(sim_path/sim-bin bmark_path/benchmark_bin input_set_path/input_set > output_file) > err_out_file`

*Exemplu:*

```
./sim-bpred -redir:sim apsi_simout.res -redir:prog apsi_progout.res
-max:inst 5000000 apsi.ss < apsi.in
```

**Obs:**

1. În loc de **sim-bpred** poate fi pus orice simulator.
2. Benchmark-ul **apsi.ss** poate fi înlocuit cu oricare altul din suita SPEC dar cu intrarea aferenta (**input\_set\_path/input\_set**).

Optiunile sunt specifice fiecarui simulator si optionale. Ulterior, se vor detalia separat la fiecare din simulatoare. Exista totusi un numar de 6 optiuni disponibile tuturor simulatoarelor:

Optiune	Comanda realizata
<b>-h</b>	Afiseaza mesaje ajutatoare în functionarea simulatorului
<b>-d</b>	Starteaza afisorul de mesaje specifice depanatorului
<b>-i</b>	Starteaza executia interactiva în mediul depanare
<b>-q</b>	Încheie imediat executia
<b>-dumpconfig</b> <file>	Genereaza un fisier de configurare salvând parametrii din linia de comanda. În acest fisier sunt permise comentarii; acestea starteaza cu simbolul #.
<b>-config</b> <file>	Citeste si încarca parametrii simulatorului dintr-un fisier de configurare. Fisierul de configurare poate referi (apela) alte fisiere de configurare.

*Tabelul 8.1.*

**Optiuni disponibile tuturor simulatoarelor**



### 8.1.6. SIMULATOARELE SETULUI DE INSTRUMENTE "SIMPLESCALAR 3.0".

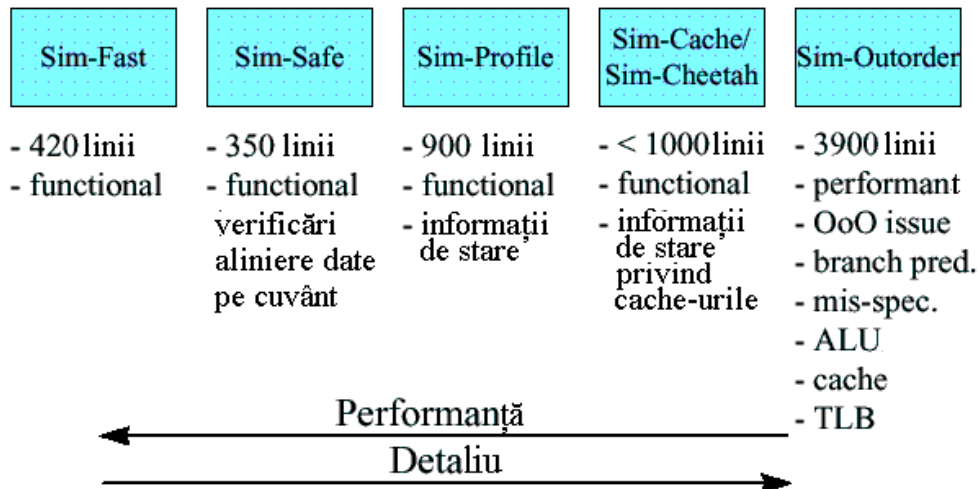


Figura 8.3. Studiu comparativ asupra simulatoarelor setului SimpleScalar

1. **Simulatorul functional** (*sim-fast*) - este cel mai rapid, elementar și puțin detaliat simulator. Simulează execuția **secvențială** a instrucțiunilor, nu presupune un sistem ierarhic de memorie, nu verifică dependențele dintre instrucțiuni. O versiune similară a acestui simulator este *sim-safe* care îndeplinește sarcinile simulatorului *sim-fast*, iar suplimentar verifică alinierea corectă a cuvintelor (instrucțiuni/date) și drepturile de acces pentru fiecare referință la memorie. Nici unul din cele două simulatoare nu acceptă parametrii suplimentari în linia de comandă. Ambele versiuni sunt foarte simple (codul sursă având mai puțin de 300 de linii) constituindu-se într-un excelent punct de start pentru înțelegerea funcționării interne a simulatorului.
2. **Simulatorul de cache-uri** (*sim-cache*) - reprezintă un simulator functional ideal pentru simularea rapidă a arhitecturilor care utilizează un sistem ierarhic de memorie, considerându-se suplimentar că timpul de acces la cache nu este relevant în ce privește performanța obținută, neimplicând deci așteptări suplimentare. *Sim-cache* acceptă argumente în linia de comandă referitoare la: dimensiunea primului, respectiv celui de-al doilea nivel de cache de instrucțiuni/date, dimensiunea buffer-ului de traducere a adreselor datelor/instrucțiunilor. Există opțiuni care permit golirea cache-urilor în cazul apelurilor sistem sau care determină

remaparea instructiunilor pe 64 de biti la instructiuni echivalente pe 32 de biti.

Optiune	Comanda realizata
<b>-cache:dl1</b> <confi>	Configureaza primul nivel al cache-ului de date
<b>-cache:dl2</b> <config>	Configureaza al doilea nivel al cache-ului de date
<b>-cache:il1</b> <config>	Configureaza primul nivel al cache-ului de instructiuni
<b>-cache:il2</b> <config>	Configureaza al doilea nivel al cache-ului de instructiuni
<b>-tlb:dtlb</b> <config>	Configureaza buferul de translatare a adreselor datelor (TLB).
<b>-tlb:itlb</b> <config>	Configureaza buferul de translatare a adreselor instructiunilor (TLB).
<b>-flush</b> <boolean>	Pe conditie <i>true</i> goleste cache-urile la un apel sistem. <boolean> = 0   1   TRUE   true   FALSE   false
<b>-icompress</b>	Remapeaza instructiunile pe 64 de biti aferente simulatoarelor setului <i>SimpleScalar</i> la instructiuni echivalente pe 32 de biti.
<b>-pcstat</b> <stat>	Genereaza informatii de profil într-un fisier text.

Tabelul 8.2.

### Optiuni specifice *sim-cache*

Configuratia cache-ului (<config>) are urmatorul format:

**<name> : <nsets>: <bsize>: <assoc> : <repl>**

Semnificatia câmpurilor este urmatoarea:

- ☐ **name** - numele cache-ului, trebuie sa fie unic.
- ☐ **nsets** - numarul de seturi din cache.
- ☐ **bsize** - dimensiunea blocului (pentru bufferele TLB - dimensiunea paginii).
- ☐ **assoc** - asociativitatea cache-ului (putere a lui 2).
- ☐ **repl** - politica de înlocuire a blocurilor din cache (l | f | r), unde l = *LRU*, f = *FIFO*, r = *random*.

Dimensiunea cache-ului se obtine ca produs între **nsets** x **bsize** x **assoc**. Pentru a avea un nivel de cache unificat în ierarhie (Princeton), "*se pointeaza*" la cache-ul de instructiuni cu numele cache-ului de date, pe nivelul corespunzator.

*Exemplu:*

1. **sim-cache –redir:sim applu\_simout.res –max:inst 5000000 –cache:il1 il1:128:64:1:l applu.ss<applu.in**
2. **sim-cache –redir:sim applu\_simout.res –max:inst 5000000 –cache:il2 dl2 applu.ss<applu.in** (cache unificat pe nivelul 2)

Valorile implicite pentru fiecare din cache-uri, în cadrul acestui simulator sunt:

L1 Cache de instructiuni:	<b>il1:256:32:1:l</b>	(8 Ko)
L1 Cache de date:	<b>dl1:256:32:1:l</b>	(8 Ko)
L2 Cache unificat:	<b>ul2:1024:64:4:l</b>	(256 Ko)
TLB pentru instructiuni:	<b>itlb:16:4096:4:l</b>	(64 intrari)
TLB pentru date:	<b>dtlb:32:4096:4:l</b>	(128 intrari)

3. **Simulatorul de cache-uri cheetah** (*sim-cheetah*) - permite generarea rezultatelor simulării pentru configuratii de cache multiple, printr-o singura simulare.

Optiune	Comanda realizata
<b>-refs [inst   data   unified]</b>	Specifica tipul fluxului (date/instructiuni) analizat.
<b>-C [fa   sa   dm]</b>	Tipul cacheului: complet asociat, set asociativ, mapat direct.
<b>-R [lru   opt]</b>	Politica de înlocuire a blocurilor conflictuale din cache.
<b>-a &lt;sets&gt;</b>	$\log_2$ din limita inferioara a numarului de seturi simulate simultan.
<b>-b &lt;sets&gt;</b>	$\log_2$ din limita superioara a numarului de seturi.
<b>-n &lt;assoc&gt;</b>	Asociativitatea maxima de analizat.
<b>-in &lt;interval&gt;</b>	Intervalul, exprimat în dimensiunea cache-ului, la care se afiseaza rezultate, în cazul în care cache-ul este complet asociativ.
<b>-M &lt;size&gt;</b>	Dimensiunea maxima a cache-ului care este supusa atentiei.
<b>-C &lt;size&gt;</b>	Dimensiunea maxima a cache-ului mapat direct care este analizata.

*Tabelul 8.3.*

### Optiuni specifice *sim-cheetah*

Nucleul *Cheetah* simuleaza eficient cache-uri complet asociative, precum si o politica de înlocuire (uneori) optima (algoritmul MIN al lui *Belady* - blocul cel mai târziu referit în viitor va fi selectat spre înlocuire).

Politica se dovedeste optima pentru fluxuri de instructiuni (fisiere) *read-only*. Pentru cache-urile cu modalitate de scriere *write-back* algoritmul de înlocuire nu este întotdeauna optim (spre exemplu poate fi mult mai costisitor sa se înlocuiasca blocul cel mai târziu referit în viitor daca blocul trebuie scris în memoria principala ("*murdar*"), fata de un bloc "*curat*" referit în viitor putin mai devreme decât blocul "*murdar*" anterior). Nucleul *Cheetah* a fost conceput ca o biblioteca de sine statatoare, rezidenta în directorul **libcheetah/**. *Sim-cheetah* accepta urmatoorii parametri în linia de comanda, suplimentari celor generali, disponibili tuturor simulatoarelor din setul *SimpleScalar*:

Ambele simulatoare sunt ideale pentru studiul "*high - level*" al cache-urilor, fara a tine cont de timpul de acces la cache (doar rata de miss fiind analizata). Pentru a masura însa efectul organizarii cache-ului asupra timpului de executie al programelor de calcul trebuie utilizat simulatorul *sim-outorder*, mult mai complex, la nivel de executie.

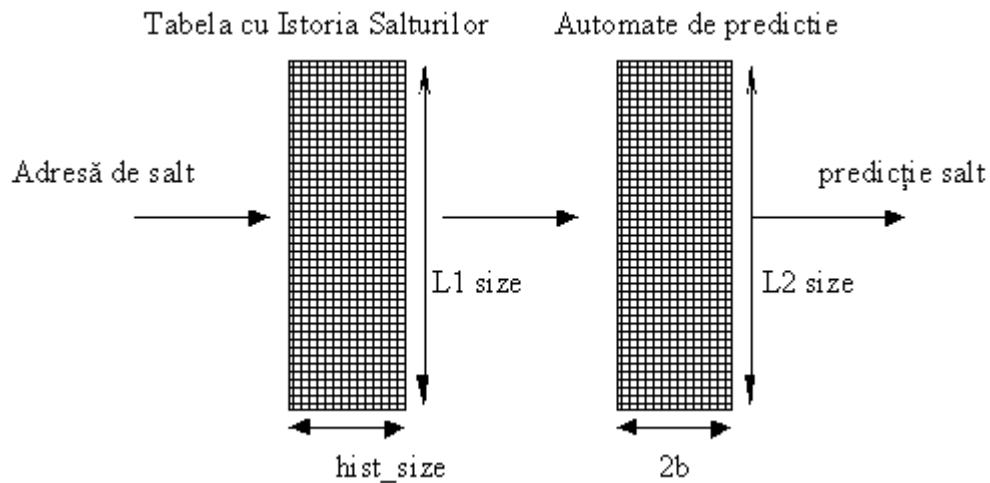
**4. Predictorul de salturi** (*sim-bpred*) – genereaza informatii statistice privitoare la acuratetea de predictie, numarul si caracteristici ale instructiunilor de salt din programele de aplicatie. Predictia salturilor este specificata prin alegerea flag-ului – **bpred** urmat de unul din urmatoarele 6 argumente:

- *nottaken* – saltul va fi prezis întotdeauna **not taken**;
- *taken* – saltul va fi prezis întotdeauna **taken**;
- *perfect* – predictorul va fi perfect din punct de vedere a acuratetii de predictie ;
- *bimod* – predictor bimodal, folosind un BTB ('*branch target buffer*') având automate de predictie pe 2 biti;
  - *2lev* – predictor adaptiv pe 2 nivele;
  - *comb* – predictor combinat (bimodal si adaptiv pe 2 nivele);

În functie de tipul predictorului argumentele specifice sunt prezentate mai jos:

- ☐ **-bpred:bimod <size>** ⇔ stabileste dimensiunea tabeli predictorului bimodal la <size> intrari.
- ☐ **-bpred:2lev <l1size><l2size><hist\_size><xor>** ⇔ precizeaza un predictor adaptiv pe doua nivele. Modelul de organizare este descris în figura 8.4.
  - ☐ **<l1size>** - specifica numarul de intrari în tabela aflata pe primul nivel al predictorului (**Ex:** N=1 – implica un registru de istorie globala, specific predictoarelor GAg sau GAp).

- ❑ **<l2size>** - reprezinta numarul de intrari în tabela aflata pe cel de-al doilea nivel al predictorului.
- ❑ **<hist\_size>** - identifica numarul de biti de istorie (globala) utilizati în procesul de predicție.
- ❑ **<xor>** - permite folosirea functiei de dispersie *xor* (dintre *adresa de salt* si *istoria salturilor*) pentru determinarea indexului de adresare în tabela de predicție aflata pe nivelul al doilea (selectarea *automatului de predicție* corespunzator).



**Figura 8.4.** Structura de predictor adaptiv pe 2 nivele

Tabelul 8.4 prezintă parametrii corespunzători unor scheme de predicție moderne. Valorile implicite pentru cei patru parametri anterior amintiți sunt **1** (**<l1size>**), **1024** (**<l2size>**), **8** (**<hist\_size>**) și **0** (**<xor>**), respectiv.

PREDICTOR	L1_size	Hist_size	L2_size	Xor
GAg	1	W	$2^W$	0
GAp	1	W	$>2^W$	0
PAg	N	W	$2^W$	0
PAP	N	W	$2^{N+W}$	0
Gshare	1	W	$2^W$	1

*Tabelul 8.4*

#### Scheme moderne de predicție corelate pe două nivele

- ☞ **-bpred:btb** <sets> <assoc> ⇔ configureaza un predictor de tip BTB având <sets> seturi si gradul de asociativitate <assoc>. Valorile implicite sunt **512** seturi si asociativitate **4-way**.
- ☞ **-bpred:spec\_update** <stage> ⇔ permite actualizarea speculativa a schemelor de predictie în fazele de decodificare sau de scriere rezultat în setul de registri generali (<stage>=[ID/WB]). Prin ne-setarea acestui parametru actualizarea predictorului se face nespeculativ în faza **Commit** (retragerea registrilor din buffer-ul de reordonare).

*Exemplu:*

- 1) **sim-bpred -redir:sim applu\_simout.res -max:inst 5000000 -bpred 2lev -bpred:2lev 1 1024 10 0 applu.ss<applu.in**
  - 2) **sim-bpred -redir:sim applu\_simout.res -max:inst 5000000 -bpred bimod -bpred:btb 512 4 applu.ss<applu.in**
5. **Simulatorul de informatii de profil** (*sim-profile*) - genereaza detaliat informatii despre adrese si clase de instructiuni, simboluri (adrese de date/instructiuni), accese la memorie, instructiuni de salt, etc.

Optiune	Comanda realizata
<b>-iclass</b>	Tipul claselor de instructiuni (ALU, Branch, etc).
<b>-iprof</b>	Tipul instructiunilor (bnez, addi etc).
<b>-brprof</b>	Tipul clasei instructiunilor de salt (salturi directe/indirecte, apeluri de subrutina, salturi conditionate).
<b>-amprof</b>	Tipul modului de adresare (directa/indirecta/indexata).
<b>-segprof</b>	Zona de date accesata (statica, dinamica).
<b>-tsymprof</b>	Informatii privind executia (functii utilizate, depanarea în interiorul acestora).
<b>-dsymprof</b>	Informatii privind zona de date.
<b>-taddrprof</b>	Informatii privind executia la o anumita adresa.
<b>-all</b>	Seteaza pe <i>True</i> toate optiunile.

*Tabelul 8.5.*

### Optiuni specifice *sim-profile*

Generarea unor statistici a informatiilor de profil, obtinute deci în urma rularii programului, se poate realiza prin comanda:

***sim-profile* -pcstat sim\_num\_insn test-math >&! test-math.out**

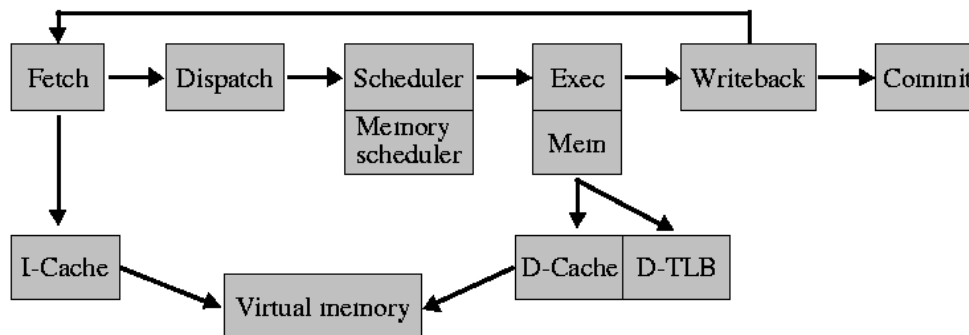
Trei dintre simulatoarele setului simplescalar (*sim-profile*, *sim-cache* si *sim-outorder*) suporta exprimarea statistica a informatiilor de profil

(descrierea detaliata a rezultatelor rularii în cadrul unui segment de text) aferenta diferitelor variabile contor de numere întregi (numarul de instructiuni din program, numarul de referinte la memorie, numarul miss-urilor pe primul nivel al cache-ului de instructiuni, acuratetea predictiei instructiunilor de salt).

*Exemplu:*

- pcstat** *sim\_num\_insn* - caracteristici ale executiei: numarul instructiunilor executate.
- pcstat** *sim\_num\_refs* - profilul referintelor la memorie (numarul instructiunilor load/store).
- pcstat** *ill.misses* - profilul acceselor cu miss în primul nivel al cache-ului de instructiuni.
- pcstat** *bpred\_bimod.misses* - profilul predictiei instructiunilor de salt.

## 6. Simulatorul super-speculativ de complexitate ridicata out of order (*sim-outorder*)



**Figura 8.5.** Structura pipeline a simulatorului Out of Order

*Sim-outorder* suporta expediere și execuție "out of order" bazat pe unitatea RUU (*register update unit*). Schema RUU utilizează un buffer de reordonare necesar în redenumirea automată a registrilor și reținerea rezultatelor pentru instructiunile aflate în așteptare. În fiecare ciclu, buffer-ul de reordonare retrage instructiunile încheiate în ordinea inițială a programului și le depune și în setul de registri generali.

Sistemul de memorie cuprinde și un buffer aferent instructiunilor Load/Store (coada FIFO). Valorile de memorat sunt plasate în coada dacă instructiunea Store este speculativă. Instructiunile Load sunt expediate spre unitățile de execuție cu referire la memorie doar când adresele tuturor instructiunilor Store anterioare sunt cunoscute (evitarea aliasurilor). Instructiunile Load pot fi satisfăcute fie prin execuție propriu-zisă fie printr-

un mecanism de bypassing realizat prin hardware, preluând valoarea de înscris a unei instructiuni Store direct din coada de asteptare, în cazul unei potriviri de adresa. Instructiunile Load executate speculativ pot genera accese cu miss în cache, dar "miss-urile" speculative în TLB blocheaza structura pipeline pâna la cunoasterea conditiei de salt.

Nucleul de executie al simulatorului este structurat astfel:

```

ruu_init();
for ( ; ; ) {
    ruu_commit();
    ruu_writeback();
    lsq_refresh();
    ruu_issue();
    ruu_dispatch();
    ruu_fetch();
}

```

unde fiecare rutina descrie o faza de procesare a instructiunii.

Bucula de program este executata odata pentru fiecare ciclu masina (A nu se înțelege perioada de tact). La încheierea fiecarui program - cu apelul sistem **exit()** - simulatorul executa un apel **longjmp()** la functia principala **main()** pentru generarea informatiilor statistice. Latentele tuturor unitatilor functionale se regasesc în structura *fu\_config[]* din modulul *sim-outorder.c*.

Faza **fetch instructiune** a structurii pipeline este implementata de catre functia *ruu\_fetch()*. Unitatea de fetch modeleaza largimea de banda a fluxului de instructiuni si primeste ca parametri de intrare urmatoarele: *PC-ul instructiunii curente* (adresa de la care se va face procesul de fetch), *starea predictorului* si *eventuale predictii gresite* rezultate din unitatea de executie a instructiunilor de salt. În fiecare ciclu procesor, sunt citite si aduse instructiuni dintr-o singura linie a cache-ului de instructiuni (un acces cu miss blocheaza eventualele urmatoare citiri, pâna la rezolvarea respectivului *miss*). Instructiunile citite sunt depuse într-un buffer de prefetch, în asteptare, de unde vor fi expediate spre decodificare si executie. De asemenea, se calculeaza cu ajutorul predictorului noua linie din cache-ul de instructiuni de unde va avea loc urmatorul proces de fetch.

Implementarea software a nivelului pipeline de **decodificare** si **clasificare/expediere** a instructiunilor spre unitatile functionale de executie se face în rutina *ruu\_dispatch()*. Pe lânga decodificare are loc si *redenumirea registrilor* în vederea eliminarii dependentelor WAR si WAW. Functia foloseste instructiunile din bufferul de prefetch ca pointeri spre unitatea RUU activa si spre unitatea (tabloul) de redenumire. O data per ciclu procesor, sunt preluate din buffer-ul de prefetch un numar de instructiuni (cel mult numarul maxim de instructiuni ce pot fi executate



simultan) și depuse într-o coadă cu instrucțiuni ce vor fi reorganizate de către scheduler. În această fază de procesare sunt efectuate predicțiile instrucțiunilor de salt. La apariția unei predicții gresite, simulatorul folosește speculativ buffer-e de stare, care sunt tratate printr-o politică **copy on write**. Rutina *ruu\_dispatch()* introduce și conectează instrucțiuni la unitatea RUU și la coada de instrucțiuni Load/Store; deoarece separa operațiile aferente instrucțiunilor cu referire la memorie în două (adunare/scadere pentru calcul efectiv de adresa și accesul efectiv la memorie).

Nivelul **issue** al structurii pipeline, de rutare efectivă (atașarea fiecărei instrucțiuni la câte o unitate funcțională de execuție) este implementat software prin procedurile *ruu\_issue()* și *lsq\_refresh()*. Rutinele pastrează de asemenea dependentele de date la resurse (registrii sau locații de memorie). În fiecare ciclu procesor, rutinele de scheduling localizează instrucțiunile pentru care registrii sursa sunt disponibili. Asignarea instrucțiunilor Load disponibile din punct de vedere al registrilor de intrare este stagnată dacă există o instrucțiune Store anterioară în coada Load/Store cu adresa efectivă necunoscută (nerezolvată). Dacă adresa instrucțiunii Store este chiar adresa de la care instrucțiunea Load, aflată în așteptare, urmează să citească atunci valoarea de memorat este înaintată spre utilizare acesteia. Altfel, instrucțiunea Load trebuie executată normal.

Faza de **execuție** a structurii pipeline este tratată, de asemenea, în rutina *ruu\_issue()*. În fiecare ciclu masină, rutina extrage cât mai multe (maxim  $IR_{max}$ ) instrucțiuni disponibile (și independente) pentru execuție din coada cu instrucțiuni reorganizate. Dacă unitățile funcționale de execuție sunt disponibile, se startează execuția instrucțiunilor (fiecărei unități funcționale libere i se asociază, dacă există, o instrucțiune de același tip cu unitatea). În final, rutina organizează evenimentele de scriere a rezultatelor (resursele folosite - registrii, memorie) în funcție de latența unităților funcționale (se va ține cont de operațiile de acces la cache-ul de date/memorie) într-o coadă de priorități. Accesele cu miss la buferul de date TLB întârzie execuția operațiilor cu memoria până în faza *Commit* a structurii pipeline. Acestor accese li se alocă în mod curent o latență fixă. Latentele tuturor unităților funcționale se regăsesc în structura *fu\_config[]* din modulul *sim-outorder.c*.

Scrierea efectivă a rezultatelor se realizează în faza **writeback**, implementată software în rutina *ruu\_writeback()*. În fiecare ciclu masină, rutina scanează coada de evenimente aferente instrucțiunilor care au încheiat faza de execuție. La găsirea unei instrucțiuni în această coadă, se parcurge lanțul dependentelor de date aferent instrucțiunii respective pentru a marca instrucțiunile dependente aflate în așteptare. Dacă o instrucțiune dependentă se află în așteptare pentru execuție, rutina o marchează ca fiind gata de

execuție. În această fază se cunoaște cu exactitate dacă saltul a fost corect predictionat sau nu. Dacă a avut loc o predicție greșită, starea procesorului este reluată de la ultimul punct de verificare, eliminând ultimele instrucțiuni executate eronat.

Procedura *ruu\_commit()* tratează instrucțiunile din faza *writeback* care sunt gata să actualizeze corect (*in-order*) structurile hardware folosite (buffer reordonare, set de registri, coada de instrucțiuni load/store, unitatea RUU). De asemenea, se actualizează și cache-ul de date (sau memoria) și sunt tratate accesele cu miss în bufferul de date TLB.

Datorită complexității sale *sim-outorder* rulează cu un ordin de mărime mai lent decât *sim-fast*. Parametrii din linia de comandă se pot clasifica în funcție de modulul component (nucleul de execuție, interfata procesor-cache, ierarhia de memorie, predictorul de salturi). Parametrii specifici nucleului de execuție al procesorului stabilesc numărul de instrucțiuni care sunt extrase simultan din cache/decodificate/transmise spre unitățile funcționale de execuție, modul de procesare (*in/out order*), capacitățile diferitelor buffer-e (coada Load/Store, unitatea RUU, unitățile funcționale ALU pentru întregi/flotante).

➤ *Parametrii specifici nucleului de execuție al procesorului:*

- ❑ **-fetch:ifqsize** <size> - setează numărul de instrucțiuni ce vor fi extrase simultan din cache / memorie principală. Trebuie să fie o putere a lui 2. Implicit are valoarea 4.
- ❑ **-fetch:mplat** <cycles> - setează latența unei predicții eronate a unei instrucțiuni de salt (procesul de "recovery"). Implicit are valoarea 3 cicluri.
- ❑ **-decode:width** <insts> - setează numărul de instrucțiuni ce vor fi simultan decodificate. Trebuie să fie o putere a lui 2. Implicit are valoarea 4.
- ❑ **-issue:width** <insts> - setează numărul maxim de instrucțiuni ce vor fi simultan decodificate într-un ciclu. Trebuie să fie o putere a lui 2. Implicit are valoarea 4.
- ❑ **-issue:inorder** – forțează simulatorul să lucreze *in-order*. Implicit are valoarea "false".
- ❑ **-issue:wrongpath** – permite expedierea spre execuție a instrucțiunilor după o speculație greșită. Implicit are valoarea *true*.
- ❑ **-ruu:size** <insts> - capacitatea unității RUU (în instrucțiuni). Implicit are valoarea 16.
- ❑ **-lsq:size** <insts> - capacitatea bufferului Load/Store (în instrucțiuni). Implicit are valoarea 8.

- ❑ **-res:ialu** <num> - specifica numarul unitatilor ALU de numere întregi. Implicit ia valoarea 4.
- ❑ **-res:imult** <num> - specifica numarul unitatilor de înmulțire/împartire întregi. Implicit ia valoarea 1.
- ❑ **-res:mempports** <num> - specifica numarul porturilor cache-ului de pe nivelul L1. Implicit ia valoarea 2.
- ❑ **-res:fpalu** <num> - specifica numarul unitatilor ALU în virgula mobila. Implicit ia valoarea 4.
- ❑ **-res:fpmult** <num> - specifica numarul unitatilor de înmulțire/împartire în virgula mobila. Implicit ia valoarea 1.

Parametrii ce caracterizeaza interfata procesor - sistemul ierarhizat de memorie sunt constituiti din toti parametrii cache-ului inclusiv formatul acestora, utilizati în **sim-cache** si folositi de catre **sim-outorder** cu urmatoarele completari: specifica latentă de hit în primul nivel al cache-ului de date, specifica dimensiunea în octeti a magistralei de memorie, reprezinta latentă pentru deservirea unui miss în tabela TLB.

➤ *Parametrii suplimentari* (gasiti doar la **sim-outorder** nu si la **sim-cache**) *ce caracterizeaza interfata procesor - sistemul ierarhizat de memorie:*

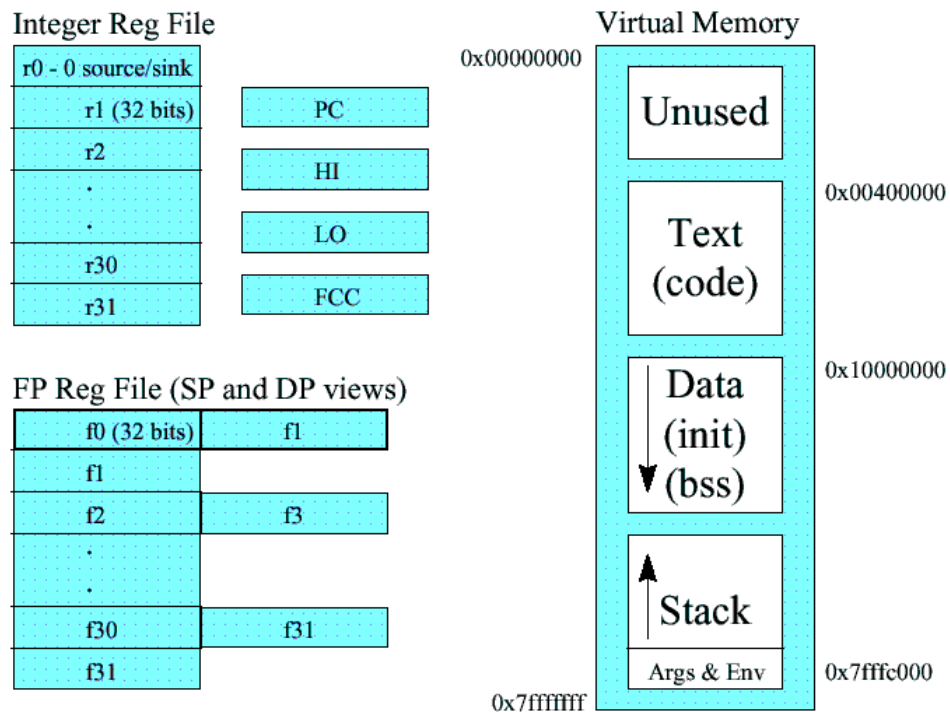
- ❑ **-cache:d1lat** <cycles> - specifica latentă de hit în primul nivel al cache-ului de date. Implicit ia valoarea 1. Exista optiunea corespondenta si pentru cache-ul de instructiuni (pentru ambele nivele **il1, il2**).
- ❑ **-mem:width** <bytes> - specifica dimensiunea în octeti a magistralei de memorie.
- ❑ **-tlb:lat** <cycles> - reprezinta latentă pentru deservirea unui miss în tabela TLB.

➤ *Parametrii specifici predictorului de salt:*

La fel ca simulatorul de cache-uri care constituie atât parte integranta a simulatorului *out of order* cât si simulator de sine statator, predictorul de salturi poate rula si individual, stabilind aceleasi informatii statistice: acuratetea de predictie, numarul de instructiuni de salt, numarul de interfete în tabelele de predictie, numarul de accese cu hit în tabele etc. Pot fi simulate o serie de predictoare, pornind de la tabela BTB la scheme de predictie adaptiva corelate pe 2 nivele: GAg, GAp, PAg, PAp.

## 8.2. ARHITECTURA SIMPLESCALAR

Arhitectura SimpleScalar este derivata din arhitectura procesorului MIPS - IV ISA. Organizarea memoriei în sistemele bazate pe arhitectura SimpleScalar este conventionala. Spatiul de adrese utilizator (pe 31 de biti) este compus din trei parti: cod, date si stiva program. Prima zona din spatiul de adrese (începe la 0x400000) este segmentul text, care memoreaza instructiunile programului. Deasupra segmentului de text este segmentul de date (începând de la adresa 0x1000000) si este împartita în doua parti. Zona de date statica contine obiecte a caror marime si adresa sunt cunoscute de catre compilator si link-editor. Imediat, deasupra acestei zone se afla datele dinamice. La alocarea spatiului dinamic de memorie pentru un program (prin **malloc** si apelul sistem **sbrk**) marginea superioara a segmentului de date se deplaseaza în sus. La marginea superioara a spatiului de adrese (0x7ffc000) este stiva de program, care creste în jos, spre segmentul de date.

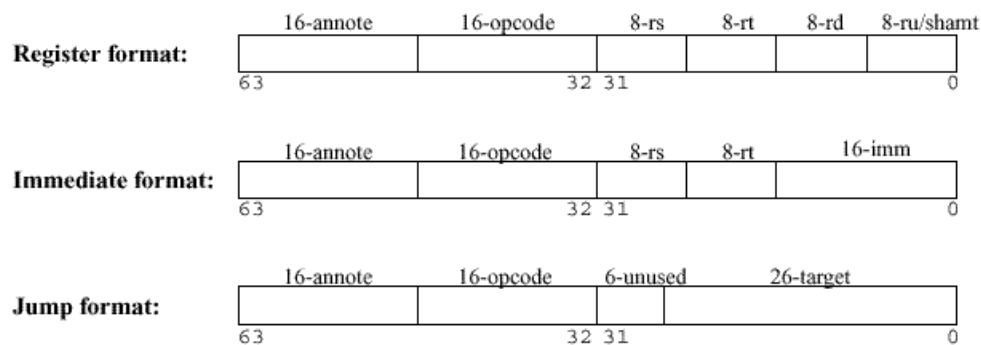


**Figura 8.6.** Arhitectura virtuala "SimpleScalar" - registrii/memorie. Orientare inversa.

Setul de instructiuni SimpleScalar PISA (Portable ISA) este o extensie a setului de instructiuni al procesorului DLX conceput de *Hennessy* si *Patterson*, incluzând de asemenea un numar de instructiuni si moduri de adresare specifice procesoarelor MIPS IV si RS/6000. Exista totusi câteva diferente notabile dar si caracteristici suplimentare din care amintim:

- ◆ Codificarea instructiunilor este pe 8 octeti.
- ◆ Cuprinde o instructiune noua, inexistentă la procesoarele amintite mai sus, de extragere a radacinii patrute atât în simpla cât si în dubla precizie.
- ◆ Instructiunile cu referire la memorie (load/store) suporta doua moduri de adresare - pentru toate tipurile de date - suplimentare celor existente la procesorul MIPS IV: *indexat* (*registru + registru*) si *auto - incrementare/decrementare*.
- ◆ Nu exista întârzieri arhitecturale (*delay slot*-uri). Instructiunile load/store/branch care în mod normal sunt caracterizate de un "*delay slot*" [în momentul finalizării instructiunii cauzatoare a întârzierii s-au executat si instructiunile succesoare din "*delay slot*"], în cazul de fata nu executa si instructiunile succesoare din banda de asamblare.

Ca si în cazul arhitecturii MIPS exista trei formate de instructiuni: *registru* (R-tip), *imediat* (I-tip) si de *salt* (J-tip).



**Figura 8.7.** Formatul instructiunilor pentru arhitectura SimpleScalar

Formatul registru este specific instructiunilor de calcul (aritmetico - logice). Formatul imediat este folosit de catre instructiunile de salt conditionat si de transfer. Cuprinde un câmp constantă imediată pe 16 biti. Al treilea format se numeste J-tip, si cuprinde printre altele si un câmp de adresa de 24 de biti. Câmpurile alocate fiecarui registru sunt pe 8 biti în scopul suportării unei extensii ulterioare a arhitecturii de registre de 256 registri întregi si 256 flotanti. Câmpul *opcode* este pe 16 biti facilitând o

decodificare rapida a instructiunii. Desi, multiplele formate complica hardware-ul, prin pastrarea unor formate similare se poate reduce aceasta complexitate. Astfel, primele doua câmpuri ale tuturor formatelor prezentate mai sus sunt identice.

Setul de instructiuni proiectat suporta eventuale *adnotari* (câmp suplimentar pe 16 biti) - modificari, sintetizari a instructiunilor post-compilare - în fisierele asamblare, fara a fi necesara o recompilare a codului masina. Exista doua tipuri de adnotari: pe bit sau la nivel de câmp. O adnotare pe bit este urmatoarea:

**lw /a \$r6, 4(\$r7)**

Adnotarea /a urmareste setarea primului bit al câmpului adnotare. Adnotarea pe bit de la /a la /p stabileste setarea bitilor de la 0 la 15 din respectivul câmp. O adnotare pe bit are urmatoarea forma:

**lw /6:4(7) \$r6, 4(\$r7)**

Adnotarea de mai sus seteaza câmpul de 3 biti (de la 4 la 6) în interiorul câmpului de 16 biti la valoarea 7. Aceste adnotari ar putea contine informatii utile hardului, furnizate de catre compilator. Prin intermediul lor, s-ar putea proiecta o interfata hardware – software extrem de utila, care sa faciliteze executia *run-time* a instructiunilor.

Setul de instructiuni precum si apelurile sistem utilizate în cadrul setului de instrumente *SimpleScalar 3.0* se regasesc în anexe 3 respectiv 4 ale prezentei lucrari.

### 8.3. BENCHMARK-URI FOLOSITE ÎN SIMULARE: SPEC, MEDIABENCH.

Benchmark-urile SPEC (*Standard Performance and Evaluation Corporation*) au fost dezvoltate de “SPEC’s Open Systems Group” (OSG), care înglobeaza peste 30 de producatori de calculatoare, integratori de sisteme, autori si consultanti din întreaga lume. Noile benchmark-uri reprezinta seturi de aplicatii necesare pentru evaluarea performantelor calculatorului si pot fi folosite pe diverse versiuni de UNIX, Linux si Microsoft [SPEC].

SPEC CPU2000 este a 4-a versiune majora a seturilor de benchmark-uri SPEC CPU, care, în 1989 a devenit primul standard acceptat la scara

larga pentru compararea performanțelor la calcul intensiv pe o varietate de arhitecturi. Performanțele rezultate cu CPU2000 nu pot fi comparate cu cele de la CPU95, pentru ca au fost adăugate noi *benchmark*-uri, iar cele vechi au fost schimbate.

Din motive financiare (lipsa licenței *benchmark*-urilor SPEC2000) am realizat simulările pe *benchmark*-urile SPEC95 (disponibile gratuit odată cu pachetul *SimpleScalar Tool Set*). Caracteristicile *benchmark*-urilor SPEC precum și intrările lor aferente sunt descrise în următorul tabel:

Programele de test SPEC '95			
<i>Benchmarks</i>	<i>Intrari folosite în simulare</i>	<i>Caracteristici</i>	<i>Numar instructiui executate</i>
Applu	Applu.in	Rezolva sisteme de matrici prin metode de pivotare	5000000
Apsi	Apsi.in	Calculeaza statistici asupra temperaturilor si gradelor de poluare	5000000
Cc1	1stmt.i	Compileaza surse pre-procesate în cod optimizat pt. procesorul SPARC	5000000
Compress95	Bigtest.in	Comprima un fisier text utilizând algoritmul adaptiv <i>Lempel-Ziv</i>	5000000
Fpppp	Natoms.in	Determina derivate multi-electron.	5000000
Go	9stone21.in	Joc de <i>Go</i> având înglobate metode strategice rafinate de IA	5000000
Hydro2d	Hydro2d.in	Calculul jeturilor galactice utilizând ecuațiile hidrodinamice ale lui <i>Navier - Stokes</i>	5000000
Ijpeg	Vigo.ppm	Comprimare prin algoritmi JPEG a unor fisiere tip imagine	5000000
Perl	Scrabbl.pl	Manipulari de texte si numere (anagrame, factorizari de numere prime)	5000000
Wave5	Wave5.in	Rezolva ecuațiile lui Maxwell..	5000000
Su2cor	Su2cor.in	Calculul masei unor particule elementare utilizând teoria <i>Quark-Gluon</i>	5000000
Swim	Swim.in	Rezolva ecuațiile lichidelor subtiri utilizând ecuații cu diferente finite (singurul bench în simpla precizie)	5000000
Tomcatv	Tomcatv.in	Rezolva probleme de geometrie computationala	5000000
Li	*.lsp	Interpretor de <i>Lisp</i>	5000000
Turb3d	Turb3d.in	Simuleaza turbulenta într-un zona cubica.	5000000
Vortex	Vortex.lit	Construieste si manipuleaza trei baze de date relationale.	5000000
Mgrid	Mgrid.in	Determina potentialul unui câmp electromagnetic.	5000000

Tabelul 8.6.

### Caracteristicile *benchmak*-urilor SPEC'95

*“Tehnologia sistemelor de calcul se dezvoltă așa de repede, încât trebuie să oferim noi pachete de benchmark-uri, pentru a asigura un mediu de testare adecvat. SPEC CPU95 a fost un mare succes, dar este timpul să facem trecerea la benchmark-uri standardizate, care reflectă îmbunătățirile tehnologice aferente microprocesoarelor, noi compilatoare, aplicații multimedia și transmisii de semnal audio/video/GSM, care s-a făcut în ultimii 5 ani; aceste benchmark-uri formează SPEC 2000.”*(Kaivalya M. Dixit, președinte SPEC)

SPEC CPU2000 cuprinde 2 seturi de benchmark-uri: CINT2000 pentru măsurarea performanțelor în cazul calculului intensiv cu numere întregi și CFP2000 pentru performanțele în cazul calculului intensiv în virgula flotantă. Cele 2 seturi măsoară performanțele procesorului, arhitecturii memoriei și compilatorului unui calculator. Îmbunătățirile aduse seturilor noi includ timp mai mare de execuție și probleme mai ample pentru benchmark-uri, o varietate mai mare a aplicațiilor, o ușurință mai mare de utilizare și platforme standard de dezvoltare care vor permite SPEC să producă versiuni adiționale pentru alte sisteme.

Setul CINT2000 cuprinde 12 benchmark-uri bazate pe aplicații, scrise în limbajele C și C++, iar CFP2000 cuprinde 14 benchmark-uri scrise în FORTRAN (77 și 90) sau C care realizează operații în virgula mobilă.

În ultima decada, substanțiale îmbunătățiri au avut loc în tehnologia compilatoarelor pentru extragerea și valorificarea paralelismului la nivelul instrucțiunilor (ILP). Majoritatea cercetărilor în acest domeniu s-au bazat pe calculul de uz general, mai exact pe suita de benchmarkuri SPEC dezvoltate pentru asistarea în evaluarea comercială și marketing-ul variantelor desktop a sistemelor de calcul. În timp ce aceste aplicații au constituit un bun mijloc pentru direcționarea cercetării existente, ele nu cuprind toate elementele esențiale ale aplicațiilor multimedia și de comunicație.

În același timp, o multitudine de arhitecturi de microprocesor au apărut având structuri VLIW, EPIC și SIMD care se potrivesc perfect necesităților compilatoarelor moderne. Majoritatea acestor procesoare sunt destinate suportului aplicațiilor dedicate (*“embedded applications”*) cum sunt cele multimedia sau comunicații, mai degrabă decât pentru sisteme de uz general. Din nefericire, există un decalaj între “comunitatea compilatoristilor” și dezvoltatorii de aplicații dedicate. *Media Bench* constituie o suita de benchmark-uri, instrumente software pentru evaluarea și sintetizarea sistemelor multimedia și de comunicații, dedicate umplerii acestui “gap” [Lee97]. Majoritatea acestor aplicații implică optimizări manuale a rutinelor scrise în limbaje de asamblare, în special pentru cele care cuprind bucle imbricate. Ideea de bază în implementarea acestor



optimizari o constituie identificarea codului din afara buclor, a dependentelor de date, a buclor vectorizabile si aplicarea tehnicilor de *trace scheduling* sau *software pipelining*.

Principalele scopuri urmarite de *Media Bench* sunt:

- ☐ reprezinta cu acuratete o baza de simulare a noilor sisteme multimedia si de comunicatie (un instrument soft de evaluare a sistemelor).
- ☐ se focalizeaza pe aplicatii portabile scrise in limbaje de nivel înalt întrucât arhitecturile de procesoare si dezvoltatorii de software migreaza înspre respectiva directie.
- ☐ stabilirea cu precizie a avantajelor MediaBench comparativ cu alternativele existente (SPEC-int, SoftFloat).

Noile clase de arhitecturi de procesoare destinate suportului aplicatiilor multimedia combina o parte din caracteristicile procesoarelor de semnal (“*DSP devices*”) - posibilitatea de a executa concurent mai multe operatii, cu caracteristicile procesoarelor de uz general care constituie “*good targets*” pentru compilatoare (numar mare de seturi de registri generali, suport pentru executia conditionata si speculativa).

MediaBench 1.0 reprezinta o suita de 19 aplicatii complete, disponibile pe Internet oricarui utilizator, scrise în limbaje de nivel înalt si compilate de catre multiple compilatoare independente pentru arhitecturi de procesare diferite. Plaja de aplicatii cuprinde procesare de imagini, compresii de sunet si imagine, comunicatii, procesare de semnal. Componentele MediaBench sunt:

- ☐ **JPEG:** JPEG este o metoda de compresie standardizata pentru imagini color sau în scala de gri. Compresia se realizeaza “*cu pierderi*” întrucât imaginea rezultata (dupa comprimare) nu este chiar identica cu cea initiala. Doua aplicatii diferite sunt derivate din codul sursa al JPEG: **cjpeg** – care realizeaza compresia si **djpeg** care realizeaza decompresia. De retinut ca benchmark-ul JPEG este comun (identic ca si cod sursa) atât suitei MediaBench cât si SPECInt’95.
- ☐ **MPEG:** MPEG2 reprezinta standardul actual predominant pentru transmisii video digitale de înalta calitate. Nucleul de calcul îl constituie o transformata cosinus pentru **codare** (*mpeg2enc*) respectiv transformata inversa pentru **decodare** (*mpeg2dec*).
- ☐ **GSM:** Se bazeaza pe standardul european GSM 06.10 pentru transcodare la rata maxima a vorbirii (sunetelor - *speech*), prI-ETS 300 036, care foloseste semnalul rezidual de excitatie pentru codificarea predictiei pe termen lung la 13kbit/s. GSM 06.10 comprima cadre de 160 (pe 13 biti) în 260 de biti.

- ☞ **G.721 Voice Compression:** Se refera la implementarile CCITT (*International Telegraph and Telephone Consultative Comitee*) G.711, G.721 si G.723 privitor la compresiile de sunet.
- ☞ **PGP:** PGP este folosit la realizarea semnaturilor electronice. Se bazeaza pe criptarea mesajelor cu ajutorul functiilor de dispersie.
- ☞ **PEGWIT:** Constituie un program de criptare cu cheie publica si autentificare, utilizând pentru aceasta curbe eliptice peste GF ( $2^{255}$ ), functia de dispersie SHA1.
- ☞ **Ghostscript:** Reprezinta un interpretor de limbaj PostScript.
- ☞ **Mesa:** Constituie o librerie grafica 3-D, similara cu OpenGL. Pachetul include programe demonstrative si aplicatii precum *mipmap*, *osdemo* sau *textgen* care efectueaza transformari / generari rapide ale texturii cu ajutorul filtrelor.
- ☞ **RASTA:** Reprezinta un program pentru recunoasterea vorbirii care suporta urmatoarele tehnici: PLP, RASTA si Jah-RASTA. Tehnicile trateaza simultan “zgomotul suplimentar” si “distorsiunea spectrala” prin filtrarea traiectoriilor temporale a unor spectre de banda critice transformate neliniar.
- ☞ **EPIC:** Este un utilitar de compresie de imagini experimental. Filtrele au fost proiectate pentru a permite o extrem de rapida decodificare, fara a fi necesara utilizarea registrilor flotanti (în virgula mobila).
- ☞ **ADPCM:** Reprezinta una din cele mai simple si mai vechi forme de codificare audio si se bazeaza pe modulatia adaptiva a semnalului audio.

Una din problemele care se pun este daca programele de test *MediaBench* sunt cantitativ diferite de SPECInt pentru un numar de caracteristici de performanta pe care arhitectii le considera importante. În urma unei analize statistice comparative a rezultatelor simularilor efectuate pe cele doua suite de benchmark-uri, exista o diferenta semnificativa în ce priveste patru parametri de performanta, în favoarea primei suite. **SPECInt necesita aproape cu 300% mai multa largime de banda decât MediaBench.** Doar programul de test *pegwitdec* surclaseaza media benchmark-urilor SPECInt. Acest trafic poate fi o consecinta directa a unor rate de hit relativ scazute la cache-ul de instructiuni si apare în cazul procesoarelor dedicate caracterizate de busuri limitate si capacitate redusa a memoriei. Ceilalti parametri care exprima o performanta superioara în cazul benchmark-urilor Media sunt: **rata de procesare** (masurabila în instructiuni/ciclu), **rata de hit în cache-ul de instructiuni** si **rata de hit pentru citirile din cache-ul de date** (ambele exprimate procentual).

Se stie ca unul din cele mai stringente scopuri urmarite de proiectantii de sisteme dedicate este de a reduce costul, în mare parte realizat prin reducerea dimensiunii modulelor componente. Daca arhitectii de procesoare

ar dori să realizeze cipuri (unități centrale) pentru sisteme de comunicații și multimedia pe baza rezultatelor simularilor efectuate pe benchmark-urile SPECInt, atunci cache-ul de instrucțiuni ar fi cu 18% mai mare decât dacă s-ar urma indicațiile de proiectare generate de simularile pe MediaBench. În ceea ce privește **acuratetea predicției și influența numărului de unități aritmetico-logice asupra ratei de procesare**, rezultatele simularilor obținute sunt aproximativ identice, indiferent de benchmark-urile utilizate.

## 8.4. APLICATII PROPUSE SPRE REZOLVARE

Scopul acestui paragraf îl constituie familiarizarea cititorilor cu mediul de simulare aferent arhitecturilor SimpleScalar 3.0. Pentru început nu este necesară modificarea codului sursă. Folosind instrumentele de simulare (preponderent simulatoarele *sim-\**, *programele de test* – în cod executabil și *intrările aferente*) se va verifica operativitatea componentelor setului. Invocarea simulării se poate face fie manual din linie de comandă fie cu ajutorul fișierelor de configurare (*batch file*). În funcție de benchmark și de setul sau de date de intrare, execuția poate dura un timp îndelungat (≈36h pe calculatoare Pentium II 400MHz, 64MB RAM).

1. *Să se ruleze pe fiecare din cele 5 simulatoare 25.000.000 instr. din benchmark-ul applu.ss și să se evalueze rezultatele simulării (timp de execuție, diverși parametri funcție de benchmark). Cele 5 simulatoare sunt: fast, bpred, cache, cheetah, outorder. Rezultatele simulării vor fi indirectate spre fișierele applu\_simout.res iar ieșirea programului spre applu\_progout.res. Să se înlocuiască apoi applu.ss cu apsi.ss și să se reia testele.*
2. *Studiati influența dimensiunii cache-ului de instrucțiuni / date (ratei de hit în cache-uri).*
  - a) Se va varia **numărul de seturi**:

Cache:	il1	il1	: 256	:64	:1	:1	(16Ko)
			: 128				(8 Ko)
			: 64				(4Ko)
			: 32				(2Ko)
			: 16				(1Ko)

Linia de comanda:

```
sim-cache.exe -redir:sim applu_16_simout.res -
redir:prog applu_16_progout.res -max:inst
5000000 -cache:il1 il1:16:64:1:l applu.ss <
applu.in
```

Se va simula variatia numarului de seturi si pentru cache-ul de date. Parametrii de simulare vor fi identici.

b) Se va varia **asociativitatea**:

Cache:	il1	il1	: 256	: 64	: 1	: l	(mapat direct)
			: 128	: 64	: 2	: l	
			: 64	: 64	: 4	: l	
			: 32	: 64	: 8	: l	
			: 16	: 64	: 16	: l	

c). Se va varia **modul de evacuare**:

Cache:	il1	il1	: 256	: 64	: 1	: l	sau : r	sau : f
--------	-----	-----	-------	------	-----	-----	---------	---------

3. Sa se studieze acuratetea de predictie in ipostazele:

Linia de comanda:

```
sim-bpred.exe -redir:sim applu_simout.res -
redir:prog applu_progout.res -max:inst 5000000 -
bpred 2lev -bpred:2lev 1 1024 10 0 applu.ss <
applu.in
```

		L1	L2	w	xor		
-bpred:	2 lev	1	256	8	0	GAg	(L2=2 <sup>w</sup> ,L1=1)
		1	1024	8	0	GAp	(L2>2 <sup>w</sup> ,L1=1)
		2	256	8	0	PAg	(L2=2 <sup>w</sup> ,L1 <sup>1</sup> 1)
		4	256	8	0	PAg	
		1	512	8	0	PAp	(L2=2 <sup>w+L1</sup> ,L1 <sup>1</sup> 1)
-bpred	: btb	<sets>		<asoc>			
		512		4			
		1024		4			
		2048		4			

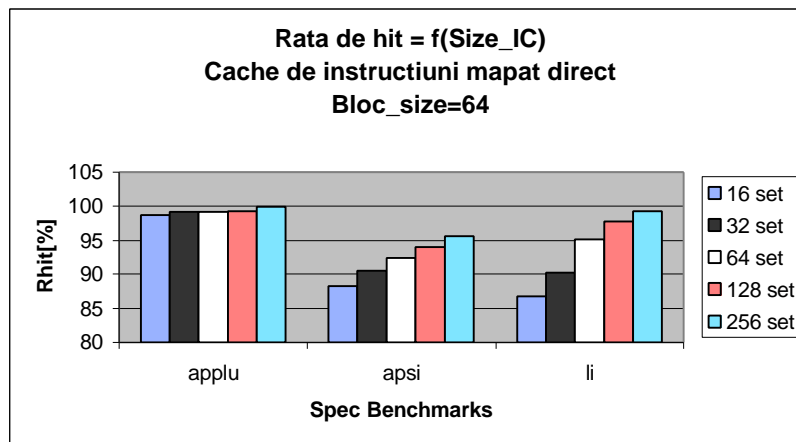
4. Comparati rezultatele simularilor obtinute folosind acelasi simulator (se compara configuratiile de cache simulate, configuratiile de scheme de predictie a salturilor). Ordonati performanta configuratiilor si formulati o ipoteza încercând sa justificati rezultatele.

5. a) Descrieti diferenta dintre multiplele instrumente de simulare (sim-\*) si de ce considerati necesara existenta a mai mult de un simulator ?  
 b) Studiati comparativ fisierele cu rezultatele statistice în urma simulării cu trei simulatoare distincte (sim-cache, sim-cheetah și sim-outorder).  
 c) Comparati viteza de executie a celor trei simulatoare (sim\_inst\_rate). De ce unul din ele are o viteza mult mai scazuta ? Daca a-ti scrie un simulator propriu care sa implementeze conceptul de Selective Victim Cache ce simulator din cadrul setului SimpleScalar 3.0 l-ati modifica ? Dar pentru implementarea unui predictor de valori, sau a unei scheme de predictie bazata pe retele neuronale ? Justificati.  
 d) Calculati necesarul de memorie (în octeti) pentru fiecare din schemele de predictie simulate la problema 3.
6. Sa se studieze influenta variatiei numarului de unitati functionale asupra ratei de procesare în cadrul simulatorului sim-outorder.

#### 8.4.1. REPREZENTAREA GRAFICA A REZULTATELOR SIMULARILOR EFECTUATE

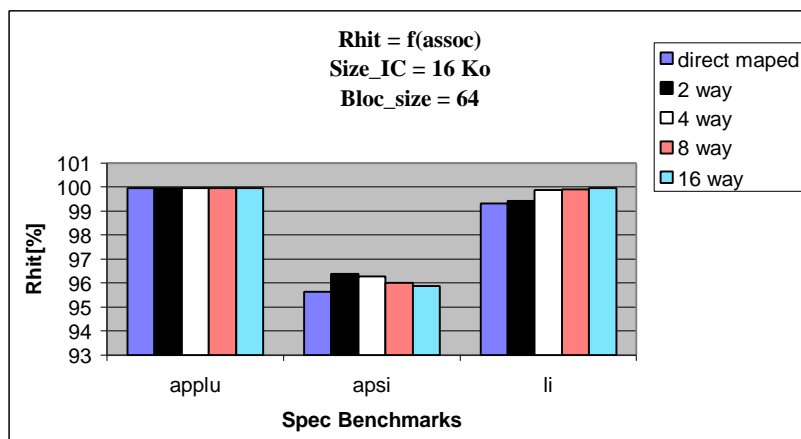
Se va urmări ca rezultatele simulării sa respecte formatul ilustrat mai jos. De asemenea, se vor efectua simulări pe celelalte benchmark-uri SPEC avute la dispozitie (din tabelul anterior prezentat).

2a)



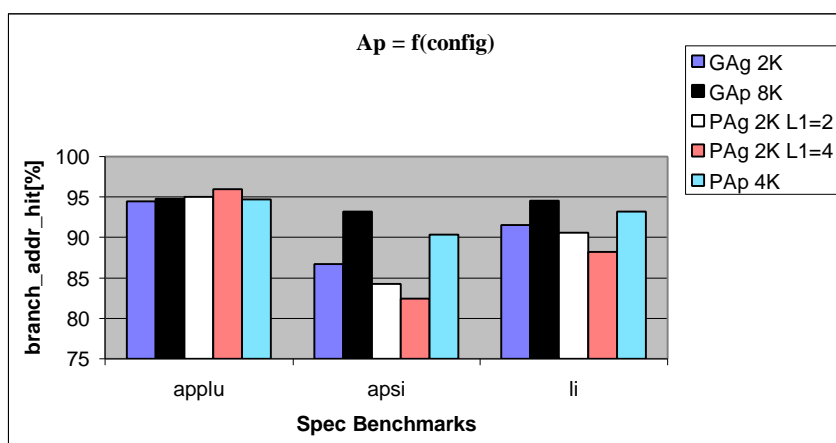
**Figura 8.8.** Influenta dimensiunii cache-ului de instructiuni asupra ratei de hit in cache.

2b)



**Figura 8.9.** Influenta gradului de asociativitate in cache-ul de instructiuni asupra ratei de hit in cache.

3)



**Figura 8.10.** Studiul diverselor configuratii de predictoare asupra acuratetii de predictie.

## 9. EXTINDEREA MEDIULUI *SIMPLESCALAR* CU MODULUL DE MASURARE A GRADELOR DE LOCALITATE

---

### 9.1. METODOLOGIA DE REALIZARE A MODULULUI *VALUE PREDICTION*

#### 9.1.1. DESCRIEREA SUMARA A CODULUI SURSA AFERENT SIMULATOARELOR *SIMPLESCALAR* EXISTENTE SI A RUTINELOR COMPONENTE

Setul de instructiuni aferent arhitecturii SimpleScalar este definit prin intermediul unor macrouri în modulul **ss.def** [Bur97]. Fiecare macro definește codul operatiei, numele instructiunii, diverse flag-uri (indicatori de conditie), operanzii sursa si destinatie, precum si actiunea ce trebuie îndeplinita pentru fiecare instructiune în particular. Actiunile instructiunilor comune tuturor simulatoarelor sunt definite în fisierul **ss.h**. Acele actiuni care necesita implementari distincte pentru simulatoare diferite, sunt definite în mod specific în codul sursa al fiecarui simulator.

La rularea unui simulator, rutina *main()* (definita în modulul **main.c**) realizeaza toate initializarile (ceasul simulatorului, biblioteca BFD, numarul de instructiuni procesate, unitatea de decodificare a instructiunilor etc.), procesarea (identificarea) optiunilor generale de simulare, iar apoi încarca în memorie codul obiect (executabil) al benchmark-ului care va fi simulat, verifica valorile parametrilor. În final, *main()* apeleaza subrutina *sim\_main()* specifica si descrisa distinct în fiecare simulator în parte. *Sim\_main()* predecodifica întregul segment de text (codul si datele benchmark-ului) pentru o mai rapida simulare, si apoi starteaza simularea din punctul specificat (instructiunea 0, sau dupa simularea unui anumit numar de

instructiuni etc.). Pentru o mai buna înțelegere a functionarii interne a simulatorului setului de instrumente SimpleScalar 3.0. se prezinta în continuare codul sursa al fisierelor componente. Toate aceste fisiere se regasesc si la adresa de web: <ftp://ftp.cs.wisc.edu/sohi/Code/Simplescalar/simplesim.tar>.

- ❏ **bpred.[c,h]:** Verifica instantierea, functionarea si actualizarea predictoarelor de salturi. Cele mai importante functii de interfata sunt *bpred\_create()*, *bpred\_loockup()*, *bpred\_update()*.
- ❏ **cache.[c,h]:** Contine functiile necesare configurarii unor multiple tipuri de cache (TLB-uri, cache-uri de date si instructiuni). Structurile de date folosite sunt dinamice (liste simplu înlantuite) care ajuta la compararea tag-urilor în cache-urile cu asociativitate scazuta(cele mult patru), si tabele de dispersie pentru comparatiile de tag-uri în cache-urile de cu grad ridicat de asociativitate. Functiile cheie folosite sunt: *cache\_create()*, *cache\_acces()*, *cache\_probe()*, *cache\_flush()*, si *cache\_flush\_adr()*.
- ❏ **endian.[c,h]:** Defineste câteva functii simple pentru a determina ordinea la nivel de octet si cuvânt de date(byte si word) pe platformele sursa (unde au fost compilate) dar si pe cele tinta (unde sunt lansate în executie).
- ❏ **eventq[c,h]:** Defineste functiile si macro-urile pentru administrarea cozilor ordonate de evenimente (folosite pentru ordonarea si arbitrarea scrierilor în faza “write-back”). Metodele mai importante sunt: *eventq\_queue()* si *eventq\_servive\_events()*.
- ❏ **loader.[c,h]:** Încarca programul tinta (applu.ss) în memoria simulatorului, seteaza marimea segmentelor de date, cod si stiva, initializeaza stiva de program, si stabileste “*punctul de intrare*” în programul de simulat. Functia cheie care realizeaza acest lucru este *ld\_load\_prog()*.
- ❏ **main.c:** Executa toate initializarile si functiile principale ale simulatorului. Metodele cheie sunt: *sim\_options()*, *sim\_config()*, *sim\_main()*, si *sim\_stats()*.
- ❏ **memory.[c,h]:** Cuprinde functiile pentru citirea din sursa, scrierea în destinatie, initializare, si afisare a continutului memoriei principale. Memoria este implementata ca si un spatiu vast compartimentat, fiecare fractiune a ei fiind alocata la cerere. Functia de baza este *mem\_acces()*.
- ❏ **misc.[c,h]:** Contine numeroase functii foarte utile, ca de exemplu: *fatal()*, *panic()*, *warn()*, *info()*, *debug()*, *getcore()*, si *elapsed\_time()*.
- ❏ **options.[c,h]:** Contine codul responsabil cu optiunile simulatorului, folosit pentru procesarea argumentelor din linia de comanda si/sau optiunile specificate în fisierele de configurare. Optiunile sunt stocate într-o baza de date (vezi functiile *opt\_reg\_\**()). *opt\_print\_help()*



genereaza o lista de mesaje de ajutor, iar *opt\_print\_options()* afiseaza starea curenta a fiecărei optiuni.

- ❏ **ptrace.[c,h]:** Furnizeaza codul necesar generarii de *trace*-uri la nivel de faze *pipeline* de procesare din simulatorul *sim-outorder*.
- ❏ **regs.[c,h]:** Evidentiaza functiile privind initializarea si afisarea continutului seturilor de registri generali: întregi, flotanti.
- ❏ **sim.h:** Contine câteva declaratii de variabile externe si prototipuri de functii.
- ❏ **stats.[c,h]:** Cuprinde rutinele necesare manipularii rezultatelor statistice ale simulării. Functii cheie folosite sunt: *stat\_reg\_\*()*, *stat\_reg\_formula()*, *stat\_print\_stats()*, *stat\_reg\_dist()* si *stat\_add\_sample()*.
- ❏ **syscall.[c,h]:** Contine codul care actioneaza ca o interfata între *apelurile sistem* aferente arhitecturii *SimpleScalar* si *apelurile sistem* caracteristice sistemului de operare care ruleaza pe calculatorul gazda.
- ❏ **sysprobe.c:** Determina ordinea la nivel de octet si cuvânt (*byte* and *word order*) existenta pe calculatorul gazda, si genereaza flagurile de compilare corespunzatoare.

### 9.1.2. EXTINDEREA SETULUI "*SIMPLESCALAR*" CU MODULUL *VALUEPREDICTION*.

- ❏ Pentru studierea unei anumite idei/concept/metode si raportare în concordanta cu majoritatea cercetatorilor în domeniul arhitecturilor de calcul trebuie ca simularea sa fie realizata pe benchmark-urile SPEC, SoftFloat sau *MediaBench*. Spre exemplu, în anul 2000, mai mult de o treime din lucrarile publicate în conferintele de top dedicate arhitecturii calculatoarelor au folosit setul *SimpleScalar* pentru simularea / evaluarea propriilor idei de proiectare.
- ❏ Setul de instrumente "*SimpleScalar*" realizeaza o standardizare a procesului de simulare a unei microarhitecturi.
- ❏ Drept consecinta, este dificil de realizat un **simulator independent** care sa proceseze benchmark-urile SPEC.
- ❏ **Solutia** consta în: se alege unul din simulatoarele sursa existente (cel care s-ar potrivi cel mai mult cu "*background-ul*" necesar cercetării noastre - de exemplu *sim-fast*, *sim-bpred*) si se urmeaza pasii descrisi în continuare:

- A)** Pe scheletul simulatorului existent (fie acesta **sim-bpred.c**), redenumindu-l, se construiește un fișier sursă nou, numit **sim-VPred.c**.

În cadrul acestui fișier se vor adauga (sau elimina) noi informații.

- ⇒ **Eliminarea** se referă la directive de compilare etc având ca "target" o altă platformă (Exemplu: *Alpha*). **Atentie**, acest lucru trebuie făcut cu multă grijă ! Ramânerea informațiilor ne-necesare în codul sursă al simulatorului nu va afecta funcționarea corectă a acestuia pe platforma pe care lucrăm (*PISA*).
- ⇒ **Adaugarea** constă în *opțiuni noi* (parametrii de intrare ai simulatorului), *statistici generale* precum și codul aferent **simularii efective** (rutina **sim\_main()** - specifică fiecărui simulator, și apelată din modulul extern **main.c**, după initializări prealabile) a ideii/tehnicii propuse spre investigare.
- B)** Se construiește un fișier sursă nou **VPred.c** care să conțină definirea simbolurilor/parametrilor, structurilor (spre exemplu: **LVPT**), implementarea funcțiilor apelate în modulul **sim-VPred.c** (spre exemplu: **CVU** etc). Se poate păstra scheletul existent (fie **bpred.c**), redenumindu-l.

- C)** Completarea modulului **Makefile** (în cadrul căruia are loc compilarea/asamblarea/linkeditarea) cu noul modul oriunde este nevoie (fișiere sursă **.c**, fișiere header **.h**, fișiere obiect **.o**, biblioteci suplimentare, fișiere executabile - ce vor fi generate). Apariția mesajului "*My work is done here !*" exprimă încheierea cu succes a operațiunii de compilare/asamblare/linkeditare a simulatorului implementat.

**Obs:**

- i) Pasul **B)** poate lipsi dar în acest caz modulul **sim-VPred.c** ar deveni prea complex cuprinzând și definițiile, implementările funcțiilor apelate în interiorul său. Este deci nerecomandabil.
- ii) Programarea se realizează în limbajul **C standard** sub sistemul de operare **Linux**, neexistând situații de gen "*Not Enough Memory*" la compilare în ciuda complexității aplicației.

Înainte de a începe simularea în sine simulatorul face uz de câteva funcții predefinite, astfel:

- ❖ înregistrează opțiunile de intrare în funcția *sim\_reg\_options* (struct *opt\_odb\_t \*odb*) folosind funcțiile *opt\_reg\_uint*, *opt\_reg\_string*
- ❖ verifică valorile argumentelor de intrare în funcția *sim\_check\_options* (struct *opt\_odb \*odb*, int argc, char \*argv)

- ❖ înregistrează informațiile statistice de ieșire în *sim\_reg\_stats* (struct\_stat\_sdb\_t \*odb) folosind *stat\_reg\_counter*, *stat\_reg\_int*, *stat\_reg\_formula*
- ❖ initializează simulatorul cu *sim\_init(void)*
- ❖ încarcă programul (benchmark-ul selectat) în starea simulată cu *sim\_load\_prog(char \*fname, int argc, char \*\*argv, char \*\*envp)*.

Funcțiile descrise sunt apelate în multe etape ale simulării și pot fi folosite în structura simulatorului (dar nu este necesar). Folosind funcțiile de ajutor furnizate, simulatorul nou creat este integrat în structura SimpleScalar și va obține interfața standard pentru opțiunile de intrare, și una pentru furnizarea rezultatelor. Cu aceste noi funcții de ajutor avem să definim câteva macrouri pentru a accesa registrii pe care îi folosește simulatorul (SET\_NPC, SET\_TPC, etc), pentru decodarea dependentelor registrilor generali (DGPR, DFPR\_L, DFPR\_F, etc.), funcții de ajutor pentru starea precisă a arhitecturii de memorie (READ\_BYTE, WRITE\_BYTE etc).

Funcția *sim\_main()* conține nucleul simulatorului. Ea buclează într-un ciclu infinit unde se aduc, se decodifică și execută instrucțiunile din program. Când programul simulat se încheie sau ajunge la un număr maxim de instrucțiuni ciclul se încheie ca de altfel și simularea.

Operația de aducere și decodificare se implementează prin două macrouri:

- MD\_FETCH\_INST(inst, mem, regs.reg\_PC)
- MD\_SET\_OPCODE

Execuția instrucțiunii se încheie folosind o întrerupere unde este inclusă definiția mașinii (**machine.def** care pointează la **pisa.def** ori **alpha.def**). Execuția instrucțiunii actuale este încheiată în **machine.def** folosind macrourele definite anterior (SET\_GPR, SET\_NPC, SET\_FPR etc). Mai mult, în timpul fazei de execuție a instrucțiunii se realizează simularea proprie (determinarea localității pe tipuri de instrucțiune, pe registrii procesorului MIPS și eventual predicția cu schema "**Last Value**").

Fisierele **vpred.h** și **vpred.c** conțin declarațiile de structuri și de date, definițiile funcțiilor folosite în implementare în vederea cuantizării localității valorilor: **addrList**, **valueList**, **pushAddress**, **pushValue**, **foundAddress**, **foundValue**.

Blocul de registre generale folosit în simulatorul SimpleScalar este implementat ca o structură definită în **regs.h**. Flaguri folosite sunt:

- F\_ICOMP - identifică operații aritmetico - logice
- F\_FCOMP - identifică operații în virgula mobilă (Floating Point)
- F\_CTRL - instrucțiuni de control (jump)
- F\_UNCOND - salt necondiționat
- F\_COND - salt condiționat

F\_MEM - instructiune de acces la memorie (cuprind atât instructiuni Load cât si instructiuni Store)  
 F\_LOAD - încarca instructiunea Load (citire din memorie)  
 Alte flaguri sunt definite în fisierul **pisa.h**.

## 9.2. DESCRIEREA SIMULATORULUI "LAST VALUE PREDICTOR"

Pentru extinderea setului de instrumente SimpleScalar cu modulul **VPred.c (Last Value Predictor** - care exploateaza gradul de localitate existent în programele de calcul - pe tipuri de instructiuni, pe registri etc. si implementeaza un modul minimal de predictor) s-a pornit de la scheletul **bpred.c** cu toate ca poate cel mai potrivit ar fi fost, datorita simplitatii sale, simulatorul *sim-fast*. Dupa descarcarea de pe Internet de la adresa "<http://www.cs.wisc.edu/~mscalar/simplescalar.html>" a fisierului *simplesim-3.0b.tar.gz*, functia *myrand()* din *misc.c* a fost modificata pentru eliminarea erorii aparute la compilare; astfel apelul functiei *random()* a fost conditionat si de *defined(\_CYGWIN32\_)*. Pentru compilarea surselor am folosit programul **Cygwin** (comanda *make*), acesta fiind un emulator de Linux, care permite generarea executabilului pentru sistemul Windows. Pentru implementarea noului simulator **VPred (Last Value Predictor)**, a fost necesara modificarea fisierului *Makefile* astfel încât la comanda *make* sa fie compilat si acesta. Structurile utilizate, precum si declaratiile functiilor se afla în fisierul *vpred.h*, iar definitiile functiilor pot fi gasite în *vpred.c*. Motorul simulatorului îl reprezinta functia *sim\_main()* din *sim-vpred.c*, aici sunt folosite atât structurile cât si functiile amintite.

Fata de arhitectura clasica a unui procesor superscalar implementata în simulatorul *sim-bpred*, se introduc structurile de date necesare pentru determinarea localitatii valorilor existenta la nivelul instructiunilor Load/ALU sau a registrilor din benchmark-urile SPEC. **Localitatea (vecinatatea) valorilor descrie probabilitatea statistica de referire a unei valori anterior folosite si stocata în aceeasi locatie de memorie.** De asemenea se introduc si structurile necesare pentru implementarea unor tehnici de predictie a valorilor. Utilizând aceasta tehnica hardware speculativa [Lip96] se urmareste exploatarea redundantei existente în programe prin colapsarea dinamica a dependentelor de date. Tehnica Load Value Prediction predictioneaza rezultatele instructiunilor Load la

expedierea spre unitatile functionale de executie exploatând corelatia dintre adresele respectivelor instructiuni (sau date) si valorile citite din memorie de catre acestea, permitând deci instructiunilor Load/ALU etc. sa se execute înainte de calculul adresei si îmbunatatind astfel performanta. Ca si consecinta a predictiei valorilor se reduc efectele defavorabile ale hazardurilor RAW, prin reducerea asteptarilor instructiunilor dependente ulterioare.

Deoarece simulatoarele de tip *execution driven* permit executia unui numar foarte mare de instructiuni, pentru determinarea localitatii valorilor sunt folosite structuri de date dinamice (fara limitari - *liste simplu înlantuite*). Acest lucru este necesar deoarece **localitatea valorii pentru un benchmark este calculata ca raport dintre numarul de instructiuni Load dinamice care regasesc o aceeași valoare în memorie ca si precedentele k accese si numarul de instructiuni Load dinamice existente în benchmarkul respectiv**, iar acest lucru presupune memorarea adresei fiecarei instructiuni Load dinamice din acel benchmark. Pentru determinarea localitatii/predictiei valorilor sunt folosite urmatoarele structuri:

<pre>// pentru extragerea gradului de localitate typedef struct element *valueList; struct element {     sword_t value;     valueList nextValue; };  typedef struct location *addrList; struct location {     md_addr_t addr;     addrList nextAddress;     valueList values; };</pre>	<pre>// pentru Last Value Predictor  typedef struct LVPTlocation *LVPTaddrList; struct LVPTlocation {     md_addr_t addr;     LVPTaddrList nextAddress;     LVPTvalueList values;     int automat;     // sword_t stride[2]; - va fi folosit     pentru dezvoltari ulterioare     (predictorul incremental)};  typedef struct LVPTelement *LVPTvalueList; struct LVPTelement{     sword_t value;     LVPTvalueList nextValue;     // int count; - va fi folosit pentru     dezvoltari ulterioare (predictorul     contextual) };</pre>
--	--

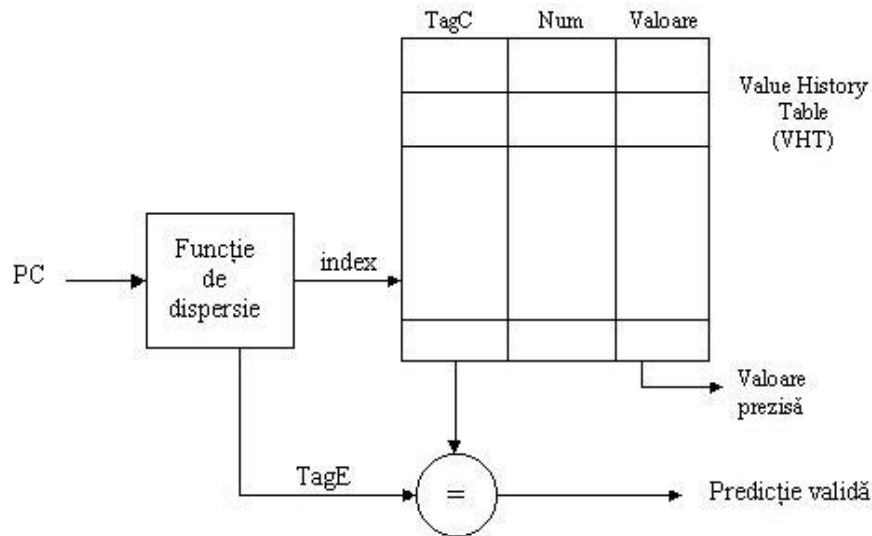
Fiecare locatie din LVPT contine urmatoarele câmpuri:

- addr* – adresa instructiunii sau adresa datei;
- values* – reprezinta lista valorilor pentru Load-ul respectiv;
- automat* – este un automat cu patru stari (*numarator saturat pe doi biti*). Un Load este nepredictibil daca automatul acestuia se afla în starea 0 sau 1 si este predictibil daca automatul se afla în starea 2 sau 3;

Presupunând ca pe lângă corelatia dintre adresele instructiunilor Load si valorile citite din memorie de catre acestea, exista o corelatie si între adresele datelor aferente instructiunilor Load si valorile de la acele adrese, simulatorul a fost proiectat în asa fel încât sa permita utilizarea în predictie atât a adresei instructiunii cât si a adresei datei. Alti parametri importanti ai simulatorului care pot fi modificati de catre utilizator, sunt: *tipul simularii* (**determinarea localitatii** - pe tipuri de instructiuni/registrii procesorului MIPS - sau **predictia valorilor**), *gradul de localizare* (istorie folosita), *tipul tabelii de predictie* (mapata direct sau asociativa), *dimensiunea tabelii de predictie*. Predictorul utilizat (a fost doar *Last Value Predictor* - aplicat pentru început doar instructiunilor Load). O idee interesanta pe care am implementat-o o constituie evaluarea gradului de localitate exprimat de registrii procesorului (în cazul nostru arhitectura MIPS IV – vezi Anexa V) [FlorVin02]. În functie de rezultatele obtinute, extinderea conceptului de localitate asupra registrilor determina implementarea unei structuri de predictie mult mai simple (cel mult 32 sau 64 de locatii), reducând astfel costul hardware. Un alt deziderat al proiectului l-a constituit exploatarea gradului de localitate exprimat si de alte tipuri de instructiuni (pentru început cele aritmetico-logice - ALU, urmând ca într-o cercetare viitoare si pentru instructiunile Store). Avantajele predictiei valorii pentru instructiunile ALU (rezultatele acestora) constau atât în reducerea timpului de executie pentru instructiunile consumatoare de timp (DIV, MULT) dar si în reducerea efectelor defavorabile ale hazardurilor RAW, prin reducerea asteptarilor instructiunilor dependente ulterioare.

Predictoarele de tip “*last value*”, fac parte din categoria predictoarelor computationale si sunt caracterizate prin faptul ca predictioneaza noua valoare ca fiind aceeaasi cu ultima valoare produsa de catre instructiunea respectiva [Vin02]. Exista si variante care schimba strategia de modificare a valorii bazat pe histerezis. Un exemplu de mecanism de histerezis consta într-un numarator saturat asociat fiecărei intrari în tabela de predictie. Acesta este incrementat/decrementat atunci când predictia este corecta/incorecta, respectiv. Valoarea memorata în tabela este evacuata numai când valoarea indicata de catre numaratorul asociat este sub un anumit prag prestabilit. Un alt mecanism de histerezis nu modifica valoarea

predictionata din tabela pâna când noua valoare nu a aparut repetitiv de un anumit numar de ori. Figura urmatoare (figura 9.1) prezinta schema de principiu a unui predictor de tip *LastValue*.



**Figura 9.1.** Schema bloc pentru un predictor de tip “*last value*”

În schema predictorului de tip “*last value*”, câmpul “*Num*” implementeaza mecanismul de histeresis printr-un numator, în concordanta cu algoritmul de modificare a valorii care a fost implementat. Functia de dispersie determina un index rezonabil de accesare a tablei VHT, pe post de adresa. Desigur ca VHT are o capacitate limitata, în principal functie de tehnologia de integrare folosita.

La citirea unei instructiuni Load din cache sau din memoria centrala, în cazul unei table **LVPT** (*Last Value Predictor Table*) mapata direct, cu cei mai puțin semnificativi biti ai adresei instructiunii Load ( $PC_{LOW}$ ) se adreseaza tabela LVPT. Maparea se face dupa urmatoarea formula:

$$\text{index} = \text{addr} \bmod \text{LVPTdim},$$

unde **addr** reprezinta adresa instructiunii sau adresa datei, iar **LVPTdim** reprezinta dimensiunea tablei de predictie. Se verifica daca *addr* este egala cu adresa de la indexul respectiv, caz în care avem hit. În cazul în care valorile nu sunt egale vom avea miss în LVPT, nu se poate face predictie, iar locatia corespunzatoare indexului calculat va fi actualizata

cu noua adresa si cu valoarea adusa din memorie de catre instructiunea Load.

În cazul în care se foloseste o tabela **LVPT asociativa**, *addr* este comparata cu adresa din fiecare locatie a tabelii si vom avea hit în cazul în care adresa este gasita. Daca avem miss în LVPT, pe baza algoritmului LRU (Least Recently Used) implementat, se evacueaza din tabela cel mai putin recent referit Load si se introduce noua adresa si valoarea citita din memorie.

Indiferent de tipul tabelii utilizate (mapata direct sau asociativa), în cazul în care avem hit în LVPT se face predictie. În functia *predictValue* din *vpred.c* este implementat predictorul de tip “*last value*” descris în [Lip96]. În cazul în care automatul din locatia LVPT se afla în starea *predictibil*, se înainteaza valoarea prezisa si aceasta este preluata prin **bypassing** de catre instructiunile dependente aflate în asteptare în statiile de rezervare. La returnarea datei reale din memorie, aceasta este comparata cu valoarea prezisa, si instructiunile dependente executate speculativ, fie urmeaza parcursul normal - nivelul *Write Back* al structurii *pipe* - fie sunt retrimise spre executie. Tabela LVPT este actualizata prin incrementarea automatului în cazul unei predictii corecte sau decrementarea acestuia în cazul unei predictii incorecte si introducerea datei citite din memorie, evacuând din locatia respectiva cea mai veche valoare printr-un algoritm de tip LRU (*Least Recently Used*). În cazul predictorului *Last Value*, istoria fiind egala cu 1, se evacueaza de fapt ultima valoare.

Exemplificam în continuare prezentând principalele functii ce intervin în cadrul predictorului Last Value: **predictValue**, **foundAssociativeLVPTAddress** si **insertLVPTValue**.

```
sword_t predictValue(LVPTaddrList p, int history)
```

```
{
    int i,j;
    LVPTvalueList q = p->values;
    if(history == 1 && q != NULL)
        return q->value;
    return 0;
}
```

```
/* declaration of list */
```

```
LVPTaddrList lvpt;
```



```
int foundAssociativeLVPTAddress(md_addr_t addr, sword_t value, int
history)
{
    LVPTaddrList p = lvpt;
    LVPTaddrList q;
    int i;
    int found = 0;
    if(lvpt == NULL)    // nu exista nici o instructiune adaugata tabeli de
                        // predictie
        return found;
    if(p->addr == addr) // instructiunea a fost adaugata anterior în tabela de
                        // predictie
    {
        found = 1;
        if(p->values != NULL)    // la adresa respectiva a existat cel putin
                                // o valoare
        {
            if(value == predictValue(p, history))
            {
                if(p->automat == 2 || p->automat == 3) /* automatul se afla în
                                                        starea predictibil */
                {
                    valuePrediction++;
                    classifiedPred++;
                    predictable++;
                }
                if(p->automat == 0 || p->automat == 1) /* nepredictibil */
                    classifiedUnpred++;
                if(p->automat < 3)
                    p->automat++;
            }
        }
        else
        {
            if(p->automat == 2 || p->automat == 3) /* predictibil */
                classifiedPred++;
            if(p->automat == 0 || p->automat == 1) /* nepredictibil */
            {
                classifiedUnpred++;
                unpredictable++;
            }
            if(p->automat > 0)
```

```

        p->automat--;
    }
}
else
{
    if(p->automat > 0)
        p->automat--;
}
insertLVPTValue(p, value, history); /* daca a existat o valoare la
                                     aceea locatie atunci ea este
                                     înlocuita cu noua valoare,
                                     pastrându-se tot timpul ultima
                                     valoare accesata; daca nu
                                     atunci se alocă spațiu și se
                                     inserează noua valoare */
    return found;
}

void insertLVPTValue(LVPTAddrList ad, sword_t value, int history)
{
    int i;
    LVPTvalueList q = ad->values;
    LVPTvalueList p;
    LVPTvalueList newValue;

    if(history == 1)
    {
        if(ad->values != NULL)
        {
            q->value = value;
            return;
        }
    }
    newValue=(LVPTvalueList)malloc(sizeof(struct LVPTelement));
    newValue->value=value;
    newValue->nextValue=ad->values;
    ad->values=newValue;
}

```

Rezultatele simulării împreună cu parametrii arhitecturii vor fi scrise într-un fișier (simout.res) în directorul curent.

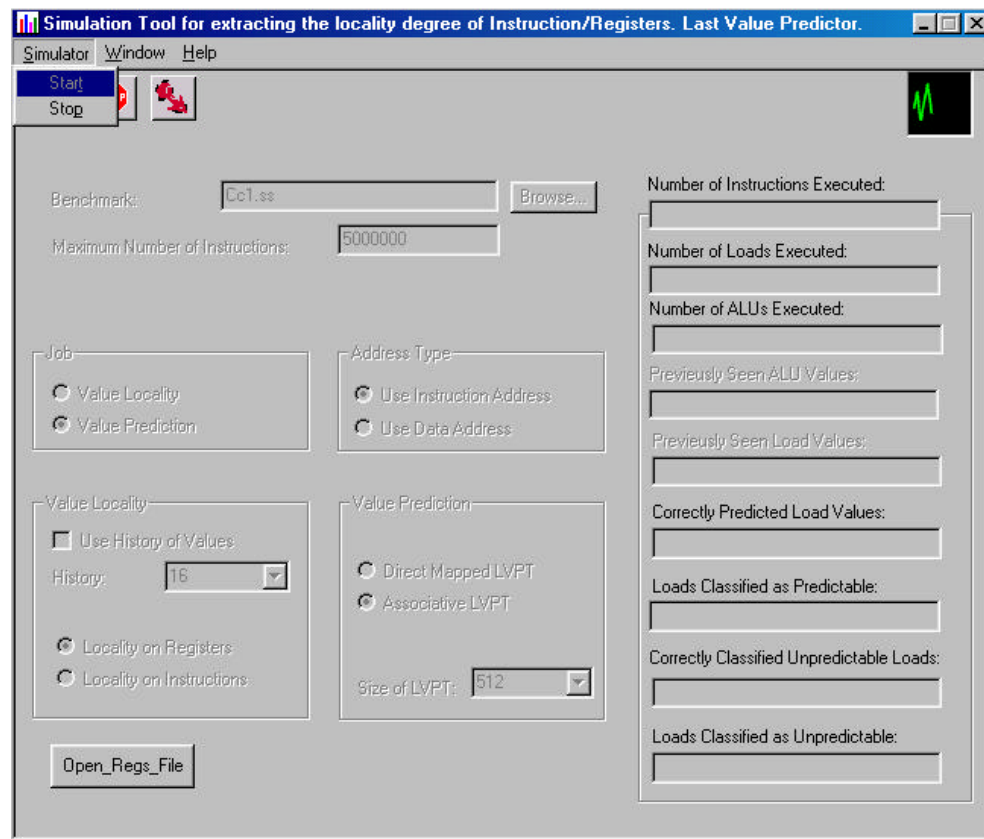
### 9.3. LAST VALUE PREDICTOR. INTERFATA GRAFICA. GHID DE UTILIZARE.

Privind din punctul de vedere al utilizatorului se considera ca fiind imperios necesara o interfata vizuala prietenoasa, bazata pe meniuri, ferestre de dialog, imagini grafice edificatoare etc. Meniurile sunt o componenta esentiala a majoritatii programelor Windows, cu exceptia unor aplicatii simple, orientate pe casete de dialog, toate programele Windows oferind un tip de meniu sau altul. O cerinta impusa interfetei cu utilizatorul o reprezinta *mnemonica* (litera subliniata), care trebuie sa apara în fiecare articol de meniu si la apasarea careia sa se selecteze articolul de meniu corespunzator. Împreuna cu tastele fierbinti (*hot keys*) – Ctrl + litera; Alt + litera; etc. – mnemonicile își dovedesc utilitatea atunci când sistemul de calcul dispune doar de tastatura nu si de mouse. Avantajul utilizarii imaginilor si a butoanelor grafice, consta în caracterul international al imaginilor, spatiu redus de stocare decât echivalentele textuale. Principalul scop al utilizarii imaginilor grafice este sa ajute utilizatorul sa recunoasca un program sau o functie mai rapid decât prin parcurgerea unui text descriptiv. Interfata trebuie sa fie simplu de utilizat, sa permita utilizatorului manevrarea usoara a simulatorului, interpretarea si prelucrarea eficienta a rezultatelor, extinderea ulterioara cu noi optiuni de meniu sau salvarea diverselor rezultate sub diferite forme.

Implementarea interfetei simulatorului în mediul Developer Studio din Visual C++ (versiunea 6.0) s-a facut datorita faptului ca limbajul C++ ofera un suport puternic pentru programarea orientata pe obiecte: încapsulare, mosteniri multiple, redefinirea operatorilor, functii si clase prieten etc. Conceptele de mostenire si polimorfism creeaza premisele dezvoltarii ulterioare (extinderii) a variantei actuale de simulator. De asemenea, implementarea trebuie realizata în asa maniera încât, orice modificare (adaugare) în hardware sau software sa fie facuta cu minim de efort. Pe lângă compilator, pachetul Visual C++ contine biblioteci, exemple si documentatia necesara pentru crearea aplicatiilor în sistemele de operare Windows 9x, Windows 2000 sau Windows NT.

Pentru a porni simulatorul este nevoie de un sistem pe care sa fie instalat Windows 9x sau Windows 2000 si sa existe benchmark-urile SPEC (programe de test). Din motive ce tin de sistemul de operare Windows 9x (preluarea identicatorului unui proces parinte - lucru care se face diferit în Windows NT) aplicatia nu poate fi rulata pe Windows NT. Aceasta se datoreaza functiilor API folosite (*CreateToolhelp32Snapshot* - care creaza o

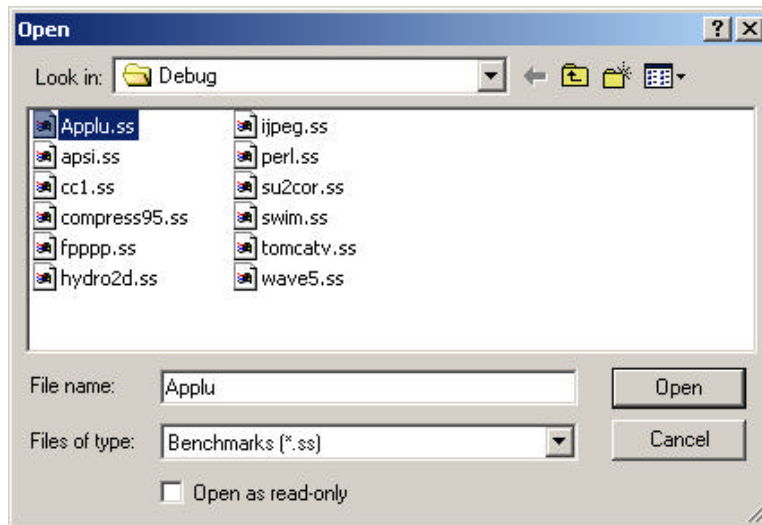
lista cu toate procesele aflate în executie în momentul respectiv, *Process32First* - care determina primul proces din lista anterior creata si *Process32Next* - care determina urmatorul proces din lista ce respecta o anumita conditie), functii care nu au corespondent în Windows NT. Pentru functionarea corecta a aplicatiei "*Last Value Predictor*" procesele care stau în spatele aplicatiei (**sim-inst.exe**, **sim-regs.exe** si **sim-vpred.exe**) si sunt vitale ei, pot fi compilate/asamblate/linkeditate atât sub sistemul de operare Windows NT/Windows 2000 cât si sub Windows 9x. La lansarea în executie a aplicatiei, pe ecran apare o fereastră înzestrata cu un meniu principal (figura 9.2) si se poate trece la introducerea parametrilor simulării.



**Figura 9.2.** Fereastră principală de configurare a simulatorului

Unul din cei mai importanti parametri este benchmark-ul simulat, deoarece fara acesta simularea nu poate fi pornita. Prin apasarea butonului *Browse*, se poate deschide fereastra (figura 9.3) care permite selectia si deschiderea unui program de test (benchmark). Limitarea duratei simulării sau a numarului de instructiuni care sa fie executate se poate face prin

introducerea unui numar maxim de instructiuni. În cazul în care aceasta limitare nu este facuta, simularea poate dura mai multe ore, sau chiar zile. Utilizatorul poate sa aleaga tipul de simulare (determinarea localitatii valorilor sau predictia valorilor), adresa care sa fie utilizata (adresa instructiunii sau adresa datei), gradul de localizare (istoria folosita), tipul tabelii de predictie (mapata direct sau asociativa) si dimensiunea acesteia. Predictorul simulat este de tip *last value* (se face predictie fara istorie).



**Figura 9.3.** Fereastra “Open” pentru selectia benchmark-urilor

Dupa introducerea parametrilor arhitecturii, simularea poate fi pornita fie prin comanda *Start* din meniul *Simulator*, Alt+S (figura 9.2), fie prin apasarea primului buton grafic. Oprirea simularii se face fie prin comanda *Stop* din meniul *Simulator* (figura 9.5), fie prin apasarea butonului grafic *Stop*. Iesirea din aplicatie se poate face prin comanda *Exit* din meniul *Window* (figura 9.5), prin apasarea butonului grafic de iesire sau prin apasarea butonului de închidere aflat în coltul din stânga sus al ferestrei.

În vederea obtinerii unor lamuriri suplimentare sau pentru dezvoltarea ulterioara a simulatorului trebuie contactata autoarea acestui simulator, apelând rubrica de meniu *Help*, care genereaza urmatoarea fereastra *About* (figura 9.4).

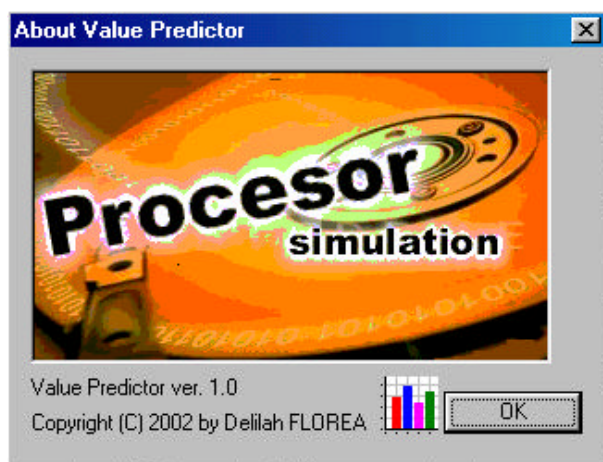


Figura 9.4. "Ecran de prezentare" a aplicatiei.

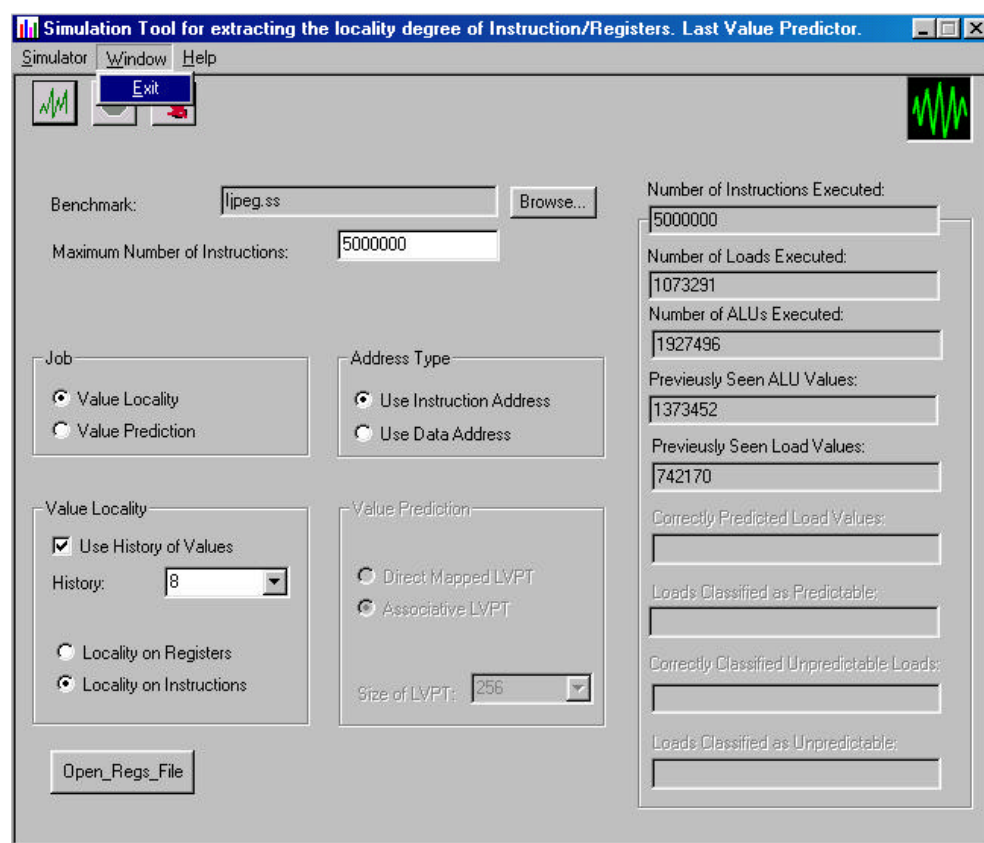
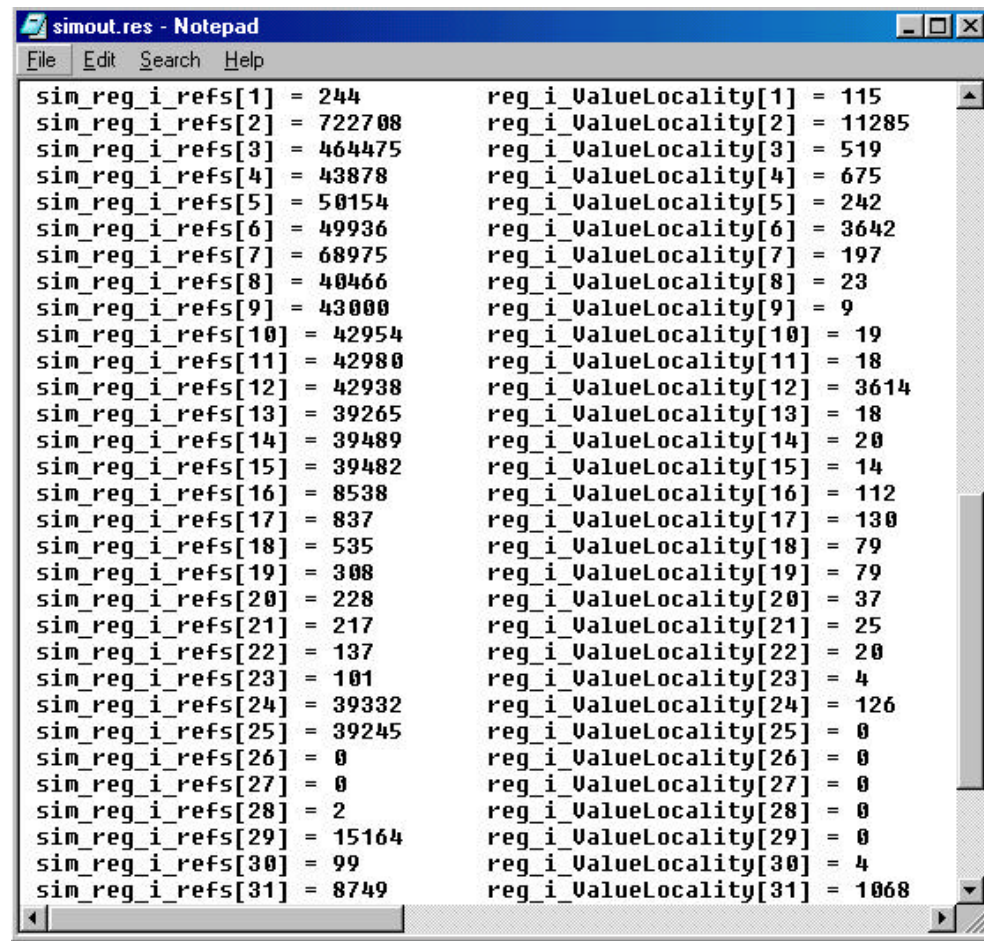


Figura 9.5. Rezultatele simulării "Localitatea valorii pe tipuri de instructiuni".

La sfârșitul simulării, în partea dreapta a ferestrei (figura 9.5), apar rezultatele: numărul de instrucțiuni executate și numărul de instrucțiuni Load/ALU executate. În cazul în care s-a dorit determinarea localității valorilor, este afișat numărul de instrucțiuni Load a caror valoare citită din memorie se regăsește printre ultimele  $k$  valori aduse de aceeași instrucțiune (dacă s-a folosit adresa instrucțiunii) sau de la aceeași adresă (dacă s-a folosit adresa datei), unde  $k$  reprezintă istoria, parametru introdus de utilizator. De asemenea, se poate observa, dacă s-a selectat butonul corespunzător, câte instrucțiuni ALU regăsesc în urma execuției aceeași valoare în registrul destinație din ultimele  $k$  valori anterior memorate.



```

simout.res - Notepad
File Edit Search Help

sim_reg_i_refs[1] = 244      reg_i_ValueLocality[1] = 115
sim_reg_i_refs[2] = 722708  reg_i_ValueLocality[2] = 11285
sim_reg_i_refs[3] = 464475  reg_i_ValueLocality[3] = 519
sim_reg_i_refs[4] = 43878   reg_i_ValueLocality[4] = 675
sim_reg_i_refs[5] = 50154   reg_i_ValueLocality[5] = 242
sim_reg_i_refs[6] = 49936   reg_i_ValueLocality[6] = 3642
sim_reg_i_refs[7] = 68975   reg_i_ValueLocality[7] = 197
sim_reg_i_refs[8] = 40466   reg_i_ValueLocality[8] = 23
sim_reg_i_refs[9] = 43000   reg_i_ValueLocality[9] = 9
sim_reg_i_refs[10] = 42954  reg_i_ValueLocality[10] = 19
sim_reg_i_refs[11] = 42980  reg_i_ValueLocality[11] = 18
sim_reg_i_refs[12] = 42938  reg_i_ValueLocality[12] = 3614
sim_reg_i_refs[13] = 39265  reg_i_ValueLocality[13] = 18
sim_reg_i_refs[14] = 39489  reg_i_ValueLocality[14] = 20
sim_reg_i_refs[15] = 39482  reg_i_ValueLocality[15] = 14
sim_reg_i_refs[16] = 8538   reg_i_ValueLocality[16] = 112
sim_reg_i_refs[17] = 837    reg_i_ValueLocality[17] = 130
sim_reg_i_refs[18] = 535    reg_i_ValueLocality[18] = 79
sim_reg_i_refs[19] = 308    reg_i_ValueLocality[19] = 79
sim_reg_i_refs[20] = 228    reg_i_ValueLocality[20] = 37
sim_reg_i_refs[21] = 217    reg_i_ValueLocality[21] = 25
sim_reg_i_refs[22] = 137    reg_i_ValueLocality[22] = 20
sim_reg_i_refs[23] = 101    reg_i_ValueLocality[23] = 4
sim_reg_i_refs[24] = 39332  reg_i_ValueLocality[24] = 126
sim_reg_i_refs[25] = 39245  reg_i_ValueLocality[25] = 0
sim_reg_i_refs[26] = 0      reg_i_ValueLocality[26] = 0
sim_reg_i_refs[27] = 0      reg_i_ValueLocality[27] = 0
sim_reg_i_refs[28] = 2      reg_i_ValueLocality[28] = 0
sim_reg_i_refs[29] = 15164  reg_i_ValueLocality[29] = 0
sim_reg_i_refs[30] = 99     reg_i_ValueLocality[30] = 4
sim_reg_i_refs[31] = 8749   reg_i_ValueLocality[31] = 1068

```

Figura 9.6. Rezultate ale simulării "Localitatea valorii pe registri".

Dacă s-a selectat **localitatea pe registri** atunci la finalul simulării trebuie apasat butonul **Open\_Regs\_File** pentru a observa localitatea

obtinuta pe fiecare din cei 32 de registrii ai microprocesorului MIPS (mai puțin R0 care are tot timpul valoarea 0). Figura 9.6 ilustreaza un fragment din fisierul *simout.res* cuprinzând aceste rezultate.

Având aceste date, se poate determina acuratetea predictiei pentru arhitecturile simulate. În cazul simulării unui predictor de valori este afisat *numarul Load-urilor a caror valoare a fost prezisa corect*. Alte date care ne arata cât de eficient a fost automatul implementat sunt: *numarul Load-urilor clasificate predictibile* - generate de automatul de predictie, *numarul Load-urilor clasificate nepredictibile* - generate de automatul de predictie, *numarul Load-urilor predictibile* care au fost **prezise corect** (automatul a clasificat valoarea predictibila iar valoarea propusa de automat a fost exact cea care se aducea din memorie). Analog pentru cele **nepredictibile prezise corect** (în sensul ca automatul a clasificat valoarea ca nepredictibila iar valoarea propusa a fost într-adevar diferita de cea adusa din memorie).

#### 9.4. APLICATII PROPUSE SPRE REZOLVARE

Obiectivul primordial al acestui paragraf este de a aprofunda cunostintele avute despre mediul de simulare SimpleScalar 3.0 si de a-l folosi într-un mod eficient la evaluarea multiplelor arhitecturi de procesare si de a încerca sa modificam codul sursa al unuia din simulatoare, aducându-ne propria contributie (simularea unor arhitecturi sau tehnici noi de procesare). Pentru simulare vor fi create *fisiere batch* care lanseaza secvential simularea tuturor benchmark-urilor avute la dispozitie. Înainte de a modifica codul sursa al fisierelor (de ex: *sim-bpred.c*) se va crea un director de lucru în care se vor copia fisierele necesare si apoi se va trece la editarea codului propriu. Se vor modifica cu precadere functiile *sim\_reg\_options()*, *sim\_reg\_stats()*.

1. Studiati influenta “istoriei” (parametrul **history**) asupra localitatii valorii:
  - a) pentru instructiunile de tip Load folosind adresa instructiunii pentru indexarea tabeli LVPT.
  - b) pentru instructiunile de tip aritmetico-logic (ALU).
2. În conditiile obtinerii valorii optime pentru parametrul **history** determinati localitatea valorii pentru instructiunile de tip Load:
  - a) centrat pe producator (“*using instruction address*”)
  - b) centrat pe memorie (“*using data address*”)

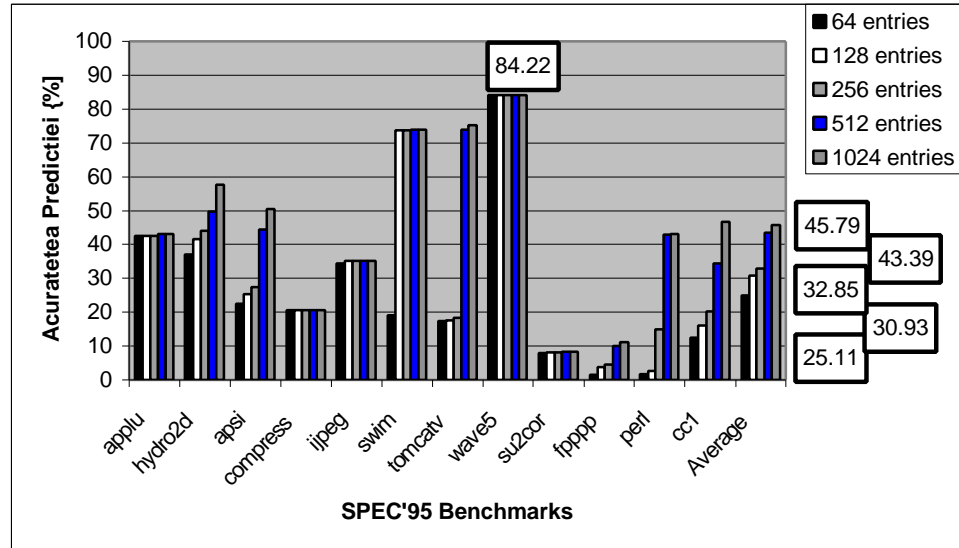


3. Sa se determine influenta parametrului **history** asupra localitatii valorii pe registrii procesorului MIPS aferent arhitecturii SimpleScalar. Determinati care registrii sunt caracterizati de o localitate considerabila (peste 70%).
4. Determinati acuratetea predictorului LastValue (**Correctly predicted load values / Number of Loads Executed**) în functie de arhitectura schemei de predictie: *mapata direct* versus *asociativa*.
5. Determinati influenta dimensiunii tablei LVPT asupra acuratetii predictiei valorii în cele doua cazuri:
  - a) LVPT – mapata direct
  - b) LVPT – asociativa
6. Stabiliti procentajul de instructiuni Load predictibile si nepredictibile identificate corect de catre predictorul LastValue, în urmatoarele 4 cazuri:
  - a) LVPT asociativa, indexata cu PC-ul instructiunii
  - b) LVPT mapata direct, indexata cu PC-ul instructiunii
  - c) LVPT asociativa, indexata cu adresa datei
  - d) LVPT mapata direct, indexata cu adresa datei
7. Adaugati un parametru nou de simulare si o statistica noua în cadrul simulatorului folosit (Ex: transformam **sim-bpred** în **sim-vpred** si adaugam *LoadValueLocality*, *ALUValueLocality*).
8.
  - a) Adaugati o optiune de simulare noua (fie **-memaddr**) care daca este **0** simuleaza localitatea valorii centrata pe producator (indexarea tablei LVPT se face cu adresa instructiunii) iar daca este **1** simuleaza localitatea valorii centrata pe memorie (indexarea LVPT se face cu adresa datei).
  - b) Verificati ca dupa modificarile efectuate, simulatoarele setului *SimpleScalar* (în principiu doar unul a fost modificat), înca functioneaza perfect. Rulati sursa modificata (simulatorul) pe un singur fisier de configurare si sesizati modificarile în fisierul cu rezultate statistice. Comparati rezultatele obtinute cu cele generate de simulatorul **LastValuePredictor**.

#### 9.4.1. REPREZENTAREA GRAFICA A REZULTATELOR SIMULARILOR EFECTUATE

Se va urmări ca rezultatele simulării să respecte formatul ilustrat mai jos. Simularile vor fi efectuate pe benchmark-urile SPEC avute la dispoziție; extrageți concluzii aferent fiecărui grafic în parte.

5b)



**Figura 9.7.** Acuratetea predictiei utilizând adresa instructiunii (PC) si o tabela asociativa în functie de dimensiunea tablei LVPT.

6b)

SPEC'95 Benchs	64		128		256		512		1024	
	Pred	Unpr	Pred	Unpr	Pred	Unpr	Pred	Unpr	Pred	Unpr
Applu	99.93	97.38	79.22	73.33	81.99	82.53	85.59	80.69	87.47	83.18
Apsi	96.01	82.97	96.66	83.92	96.65	84.7	97.16	82.86	96.96	81.74
cc1	91.29	86.70	91.87	82.21	92.04	77.19	95.16	75.84	94.76	75.55
Compress	94.04	94.8	94.04	94.79	94.06	94.77	89.06	93.17	89.07	93.16
Hydro	96.52	81.35	98.25	79.35	98.56	74.29	98.6	72.24	97.54	69.57
Ijpeg	81.17	98.63	81.17	98.57	86.86	99.24	86.94	99.23	87.03	99.33
Perl	85.56	88.5	83.10	84.6	82.35	81.51	86.98	78.08	91.13	81.46
su2cor	99.47	99.74	99.18	99.69	99.04	99.78	97.43	99.62	97.37	99.59
Swim	96.07	61.39	98.56	37.72	98.67	26.78	99.03	39.54	98.3	53.07
Tomcatv	97.64	58.91	96.26	66.94	94.96	62.61	95.38	61.60	95.77	50.70
HM	94.4	73.6	92.5	74.4	93.2	68.8	93.8	74.5	94.2	75.8

*Tabelul 9.1.*

**Procentajul de instructiuni Load predictibile si nepredictibile identificate de catre predictorul de tip "Last Value". LVPT - mapata direct si este indexata cu PC-ul instructiunii.**

## 10. IMPLEMENTAREA UNOR TEHNICI DE PREDICTIE DINAMICA A VALORILOR

---

### 10.1. SCOPUL LUCRARIII

Scopul lucrării consta în implementarea câtorva din tehnicile de predicție dinamică a valorilor. În acest sens este indicată verificarea fezabilității a noi scheme de predicție a valorii: predictoare *computationale* (generează predicția pe baza istoriei valorilor memorate, prelucrându-le după un algoritm stabilit) respectiv *contextuale* (lucrează la nivel de o singură instrucțiune și încearcă să prezică viitoarea valoare care va fi produsă de instrucțiune bazându-se pe repetarea unor contexte anterior generate). Întrucât respectivele scheme încearcă să înmagazineze o cât mai mare istorie de predicție, aceasta implică tabele hardware de mari dimensiuni și costuri ridicate de implementare. Optimul cost / performanță poate fi determinat doar prin simulare.

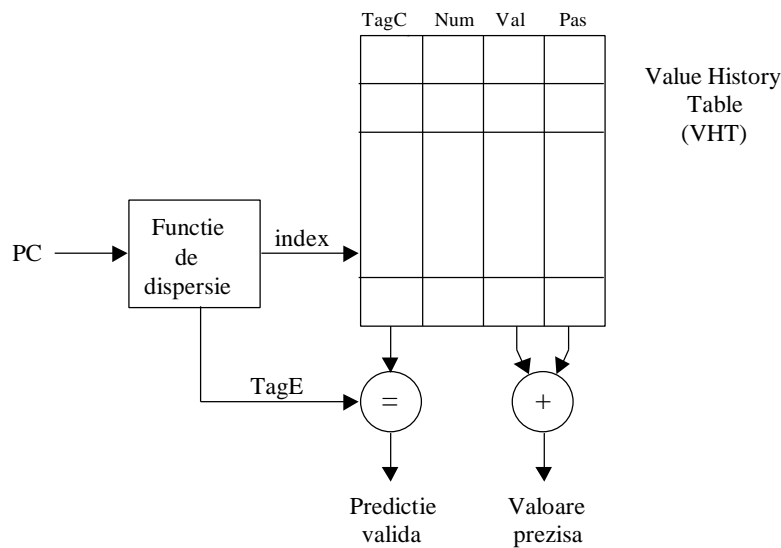
### 10.2. MEMENTO TEORETIC

#### 10.2.1. PREDICTOARE INCREMENTALE

Lipasti [Lip96] arată că acurătatea predicției obținută printr-o schemă de tip “*last value*” este în medie de 49%, măsurat pe o serie de benchmark-uri SPEC. Considerând că se memorează ultimele 4 valori produse de către o anumită instrucțiune și că abilitatea predictorului de a alege valoarea corectă este perfectă, s-a obținut o acurătate de predicție medie de 61%.

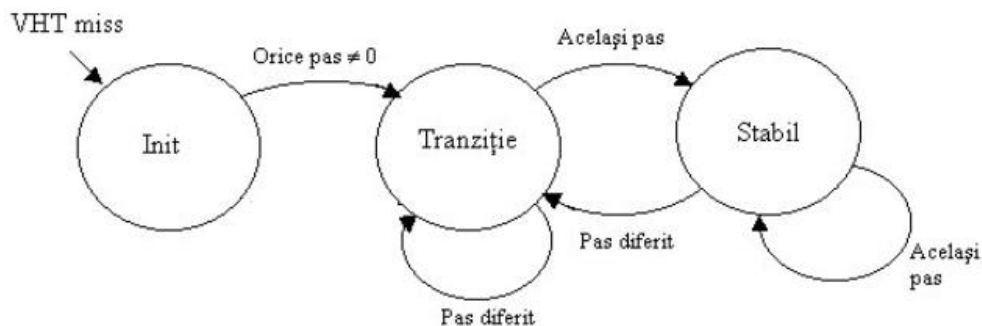
Predictoarele incrementale fac parte din categoria celor computationale si reprezinta o generalizare a predictorului de tip *Last Value* (echivalent cu un predictor incremental cu pas 0). În acest caz, considerând ca  $v_{n-1}$  si  $v_{n-2}$  sunt cele mai recente valori produse, noua valoare  $v_n$  va fi calculata dupa formula de recurenta:  $v_n = v_{n-1} + (v_{n-1} - v_{n-2})$ , unde  $(v_{n-1} - v_{n-2})$  este pasul secventei. Pasul nu este obligatoriu sa fie constant în timpul executiei programului, ar putea fi si variabil, însa pentru obtinerea unei acurateti ridicate, pasul trebuie sa fie constant pe perioade de timp suficiente pentru predictie. În aceasta idee se propun scheme de actualizare a pasului bazate pe histerezis. Astfel, pasul memorat în tabelele de predictie este modificat numai atunci când numaratorul saturat asociat, memoreaza o valoare situata peste un prag stabilit. Fireste, acest numarator este incrementat în cazul unei predictii corecte, respectiv decrementat în cazul unei predictii incorecte. O alta strategie cu histerezis este numita “2 – delta”. În cadrul acesteia sunt memorati doi pasi ( $s_1$  si  $s_2$ ),  $s_1 = v_n - v_{n-1}$ . Pasul  $s_2$  este cel folosit în procesul de predictie, numai atunci când aceeași valoare  $s_1$  a aparut de doua ori consecutiv. Ambele metode cu histerezis reduc numarul predictiilor eronate de la doua, la doar una, în cazul secventelor incrementale repetitive.

#### 10.2.1.1. PREDICTOARE INCREMENTALE CU PAS CONSTANT



**Figura 10.1.** Structura de predictor incremental

În tabela de predicție a unui predictor incremental, pe lângă valoarea adusă de instrucțiunea Load din memorie, trebuie introdus și pasul. În figura anterioară (figura 10.1) se prezintă o structură tipică de predictor incremental (pas constant). O acțiune importantă în cadrul unui predictor incremental este constituită de detectia pasului. Prima dată când o instrucțiune va fi procesată, va rezulta un *miss* în tabela de predicție VHT și este evident că nu se va face nici o predicție.

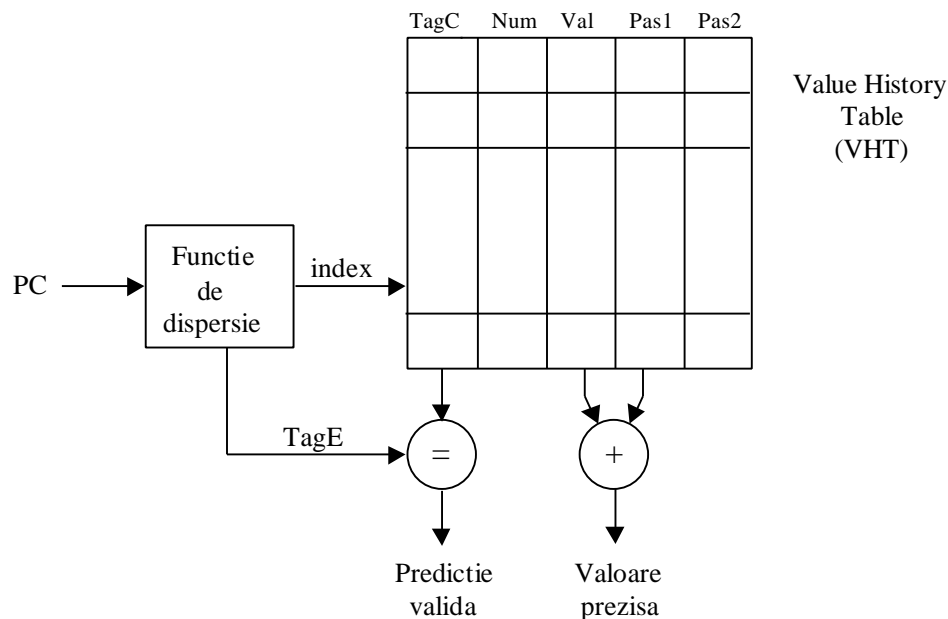


**Figura 10.2.** Automatul de stare asociat

Când o instrucțiune produce un rezultat atunci rezultatul este memorat în câmpul "Val" din VHT și automatul trece în starea inițială "Init" (vezi figura 10.2). Atât timp cât eventuale următoare instanțe succesive ale aceleiași instrucțiuni produc același rezultat ( $\text{pas}=0$ ), automatul va rămâne în starea inițială. Dacă însă o instanță următoare a instrucțiunii produce un rezultat ( $R_1$ ) diferit de precedentul, atunci se calculează pasul  $S_1 = R_1 - \text{Val}$ , iar  $R_1$  și  $S_1$  se introduc în câmpurile "Val" și "Pas" din VHT. Totodată automatul trece în starea intermediară "Tranzitie". Dacă în această stare apare o nouă instanță dinamică a instrucțiunii, nu se va genera nici o predicție, dar atunci când aceasta produce un rezultat ( $R_2$ ), se calculează pasul  $S_2 = R_2 - \text{Val}$ , iar  $R_2$  se introduce în câmpul "Val" din VHT. Dacă  $S_1 = S_2$  atunci automatul trece în starea "Stabil", altfel rămâne în starea "Tranzitie" și  $S_2$  se introduce în câmpul "Pas" din VHT. În starea "Stabil", valoarea prezisă este calculată prin adunarea valorii memorate în tabela VHT cu pasul  $S_2$ . Cu alte cuvinte, acest automat de stare implementează un anumit "grad de încredere" asociat predicției. Aceasta nu se face decât în cazul în care pragul de încredere (*Confidence Threshold*) depășește o anumită valoare, apriori determinată.

### 10.2.1.2. PREDICTOARE INCREMENTALE DE TIP “2-DELTA”

Spre deosebire de predictorile incrementale cu pas constant, predictorile incrementale de tip “2-delta” memorează doi pași în intrările tabelului VHT. În figura următoare (figura 10.3) se prezintă structura acestui predictor.



**Figura 10.3.** Structura predictorului incremental de tip “2-delta”

Prima dată când o instrucțiune va fi procesată, va rezulta un *miss* în tabela de predicție VHT și evident că nu se va face nici o predicție. Când o instrucțiune produce un rezultat atunci rezultatul este memorat în câmpul “Val” din VHT și automatul trece în starea inițială. La următoarea instanță a aceleiași instrucțiuni nu se va genera nici o predicție, dar se calculează pasul  $S_1 = R_1 - \text{Val}$ , iar  $R_1$  și  $S_1$  se introduc în câmpurile “Val” și “Pas1” din VHT. Dacă apare o nouă instanță dinamică a instrucțiunii, din nou nu se generează nici o predicție, în schimb se calculează pasul  $S_2 = R_2 - \text{Val}$ , valoarea din câmpul “Pas1” se introduce în câmpul “Pas2”,  $R_2$  și  $S_2$  se introduc în câmpurile “Val” și “Pas1” din VHT. Dacă la o instanță dinamică a instrucțiunii  $S_1 = S_2$ , se calculează valoarea prezisă adunând valoarea memorată în tabela VHT cu pasul  $S_2$ . În cazul în care automatul se află în starea *predictibil* se generează predicția. Numaratorul este incrementat

atunci când predicția este corectă, respectiv decrementat atunci când predicția este greșită.

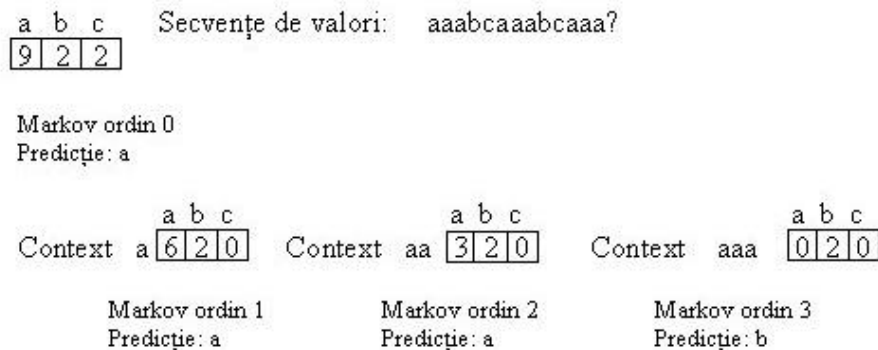
Cele două scheme anterior prezentate nu sunt singurele predictoare computaționale existente. Pot fi implementate și alte variații pe această temă, alegerea soluției optime fiind determinată prin simulare. Pentru o acuratețe ridicată a predicției trebuie anticipat tipul calculului pe care programul îl execută, în caz contrar rolul predictoarelor ar fi total inefficient. Realizarea acestui lucru se poate face doar în urma unei analize a caracteristicilor de profil a programului sursă din limbajul de nivel înalt [Vin02].

### 10.2.2. PREDICTOARE CONTEXTUALE

Predictoarele bazate pe context lucrează la nivel de o singură instrucțiune și încearcă să prezică viitoarea valoare care va fi produsă de instrucțiune bazându-se pe valorile anterior generate. În acest caz predicția valorii următoare se face pe baza unui context anterior înregistrat și a valorilor ce au urmat imediat acestui context în istoria memorată. Printr-un context se înțelege o secvență finită de valori cu apariție repetată. Aceste predictoare bazate pe context permit predicția oricărei secvențe repetitive de valori, incrementale sau non – incrementale. Un predictor contextual este de ordinul  $k$  dacă informația sa de context include ultimele  $k$  valori iar căutările se fac cu acest *pattern* de  $k$  valori. Principala limitare a acestor predictoare constă în faptul că pentru a predicționa corect o anumită valoare, aceasta trebuie să fi urmat același context cel puțin o dată.

#### 10.2.2.1. PREDICTOARE CONTEXTUALE DE TIP PPM

O clasă importantă a predictoarelor contextuale sunt cele care implementează algoritmul “*Prediction by Partial Matching*” (PPM), care conține un set de predictoare markoviene (număratoare de frecvențe de apariție ale valorilor succesoare contextului) ca în figura următoare (figura 10.4). Ideea originală aparține lui Trevor Mudge [Mud96] și a fost utilizată în recunoașterea formelor și algoritmi de compresie.

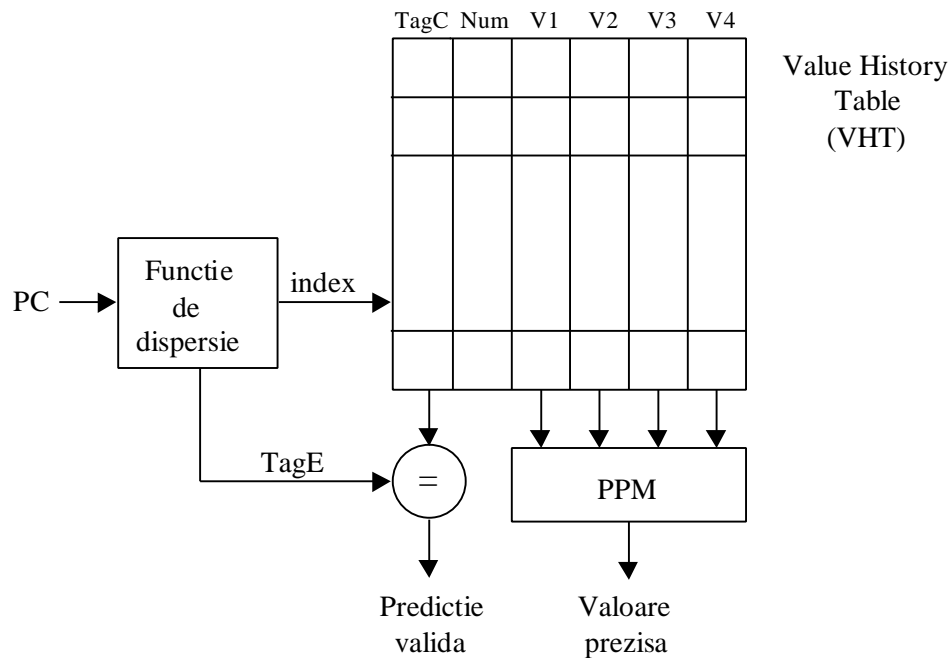


**Figura 10.4.** Predictor contextual markovian

Valoarea predictionata va fi aceea care a urmat cu cea mai mare frecventa contextului considerat. Dupa cum se observa în figura 10.4, ea este functie si de contextul considerat, un context mai “bogat” (mai multe valori care formeaza contextul) conducând adeseori la o acuratete mai ridicata a predictiei (nu întotdeauna: câteodata se poate comporta ca “zgomot”!). Valoarea optima a contextului se determina doar prin simulare. În exemplul considerat, abia predictorul Markov de ordinul 3 genereaza o predictie “corecta”. Ca si în cazul predictiei branch-urilor si aici un predictor PPM complet ar trebui sa contina N predictoare *Markov*, de la ordinul 0 pâna la (N-1). Daca predictorul *Markov* de ordinul (N-1) produce o predictie (context gasit în secventa de valori) procesul se termina, daca nu atunci se activeaza predictorul *Markov* de ordinul (N-2) s.a.m.d. Deoarece predictorul PPM complet este greu de implementat în hardware (s-a demonstrat ca un PPM complet de ordinul  $k$  consuma aproximativ dublu costului hardware necesar implementarii unui predictor corelat adaptiv pe doua niveluri de acelasi ordin) se poate alege o varianta simplificata de predictor contextual care sa contina doar doua predictoare *Markov*. Astfel, daca predictorul *Markov* de ordinul (N-1) nu produce o predictie (contextul nu este gasit în secventa de valori memorate - cele *history*) se activeaza predictorul *Markov* de ordinul 0. S-a considerat N-1 ca fiind dimensiunea contextului (vezi figura 10.4).

Secventele periodice de valori sunt predictionate cu o acuratete de 100% prin predictoarele contextuale de tip PPM, în schimb, perioada de învățare a acestora (numarul de valori din secventa de intrare generate înainte primei predictii corecte) este mai mare decât în cazul predictoarelor incrementale.





**Figura 10.5.** Structura predictorului contextual de tip PPM

În figura 10.5 se prezintă structura predictorului contextual de tip PPM. Fiecare intrare din tabela de predicție VHT are asociat câte un numărător saturat. Acesta este incrementat atunci când predicția este corectă, respectiv decrementat atunci când predicția este incorectă. În câmpurile  $V_1, V_2, \dots, V_4$  se memorează ultimele patru valori pentru fiecare instrucțiune din tabela VHT. Dacă automatul se află în starea *predictibil*, pe baza secvenței de valori memorate se generează predicția, valoarea prezisă fiind aceea care a urmat cu cea mai mare frecvență contextului (figura 4). Blocul PPM determină care valoare s-a repetat de cele mai multe ori după contextul format din ultimele cele mai recente patru valori ( $V_1, V_2, V_3, V_4$ ), în istoria de valori avute la dispoziție.

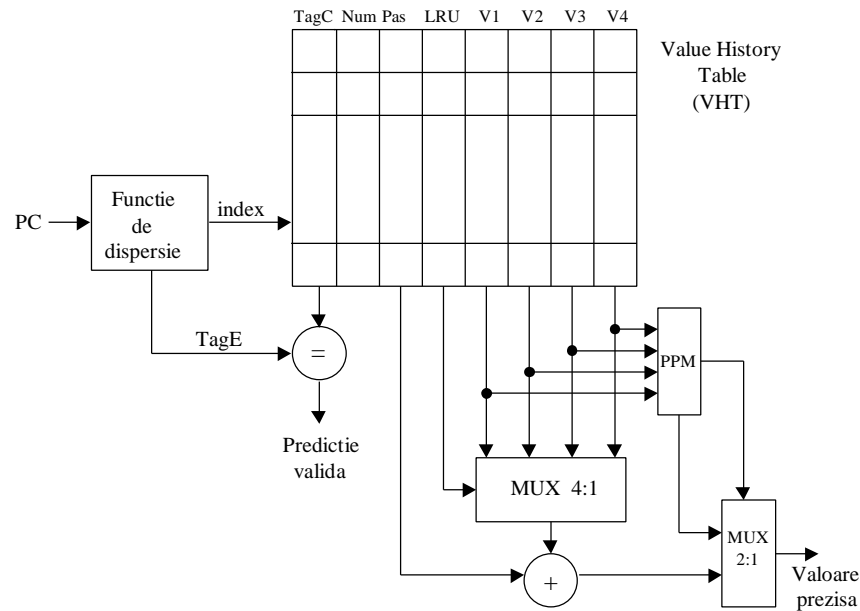
O implementare simplificată și fezabilă tehnologică a predictorului contextual de tip PPM o constituie predictorul adaptiv pe două niveluri. Întrucât această schemă de predicție nu a fost implementată în cadrul simulatorului de față nu se va insista asupra ei, detalii suplimentare găsindu-se în [Vin02]. Foarte pe scurt, principiul de funcționare al schemei este următorul: fiecărei instrucțiuni memorate în VHT i se asociază un context reprezentând practic secvența ultimelor  $p$  valori produse de către aceasta. În cadrul acestui context se calculează prin intermediul a patru numărătoare ( $C_0 - C_3$ ), valoarea cea mai frecvent generată de către respectiva instrucțiune

în istoria ultimelor sale  $p$  instante. Dacă aceasta valoare a aparut suficient de frecvent, ea este predictionata ca fiind valoarea urmatoare. Faptul ca tabela VHT memoreaza doar ultimele 4 valori produse se bazeaza pe urmatoarea justificare statistica: între 15 - 45% dintre instructiuni produc o singura valoare în ultimele lor 16 instante succesive si respectiv între 28 - 67% dintre instructiuni produc maximum 4 valori distincte în ultimele lor 16 instante dinamice de aparitie [Lip96, Vin02].

### 10.2.3. PREDICTOARE HIBRIDE

Limitarea fundamentala a procesului de predictie a valorilor prin predictoare computationale poate fi depasita cu ajutorul unui predictor hibrid. Statisticile cercetatorilor arata ca un singur tip de predictor (computational, contextual etc.) nu da în general rezultatele cele mai bune. Este evident ca anumite tipuri de secvente de valori generate prin program sunt prezise mai bine de un anumit predictor, iar altele, de catre un alt tip de predictor particular. Din acest motiv s-a nascut ideea predictiei hibride, adica doua sau mai multe predictoare de valori sa conlucreze în mod dinamic în procesul de predictie. Spre exemplificare, în figura urmatoare (figura 10.6) se prezinta un predictor hibrid compus dintr-un predictor contextual de tip PPM si respectiv un predictor incremental. Predictorul contextual are întotdeauna prioritate, astfel valoarea generata de catre predictorul incremental este folosita doar în cazul în care predictorul contextual nu genereaza o predictie. Solutia nu este cea mai performanta: schema hibrida sufera datorita neadaptivitatii sale. Pentru obtinerea unei acurateti de predictie optime este recomandabila implementarea unor grade de încredere (numaratoare saturate) fiecarui tip de predictor, în locul acestei prioritizari simpliste [Vin02].

Exista si implementari interesante de predictoare hibrid compuse dintr-un predictor adaptiv pe 2 niveluri si respectiv un predictor incremental [Wan97]. Ideea de baza este simpla: dacă predictorul pe 2 niveluri genereaza o predictie, aceasta este cea generata de catre predictorul hibrid, în caz contrar, se selecteaza valoarea generata de catre predictorul incremental (dacă acesta genereaza vreuna).



**Figura 10.6.** Schema bloc a unui predictor hibrid (PPM, incremental)

Scopul noilor generații de microprocesoare avansate: execuția superspeculativă a instrucțiunilor și viteze ridicate de procesare poate fi atins prin implementarea schemelor de predicție valorilor anterior enunțate. Principalele avantaje obținute vor fi reducerea latentei caii critice de program, reducerea timpului mediu de acces la memorie – pentru instrucțiunea Load, diminuarea efectelor defavorabile ale hazardurilor RAW în urma deblocării spre execuție a instrucțiunii dependente următoare.

În această lucrare au fost realizate câteva investigații cantitative privind conceptul de predicție dinamică a valorilor. Evaluările arhitecturilor propuse s-au făcut folosind un simulator execution-driven dezvoltat din setul de instrumente SimpleScalar-3.0 și pe benchmark-urile SPEC'95.

### 10.3. DESFĂȘURAREA LUCRĂRII

Interfața aplicației este asemănătoare cu cea a simulatorului *LastValuePredictor* din capitolul anterior "Extinderea mediului *SimpleScalar* cu modulul de măsurare a gradelor de localitate", ambele având ca trunchi comun modulul de măsurare al gradului de localitate

pentru instructiunile Load si predictorul de tip *LastValue*. Suplimentar, lucrarea de fata implementeaza trei scheme noi cu aplicabilitate în predictia valorilor: predictorul *incremental*, *contextual* si unul *hibrid* compus din cele doua. Fata de arhitectura clasica a unui procesor superscalar implementata în simulatorul *sim-fast*, se introduc structurile necesare descrierii tabelii de predictie **VHT** (*value history table*) - pentru predictorul incremental si respectiv cel contextual.

```
typedef struct VHTelement *VHTvalueList;
struct VHTelement
{
    sword_t value;
    VHTvalueList nextValue;
    int count;    // este folosit doar de catre predictorul contextual
};

typedef struct VHTlocation *VHTaddrList;
struct VHTlocation
{
    md_addr_t addr;
    VHTaddrList nextAddress;
    VHTvalueList values;
    int automat;
    sword_t stride[2]; // este folosit doar de catre predictorul incremental
};
```

Fiecare locatie din **VHT** contine urmatoarele câmpuri:

- ☞ **addr** – adresa instructiunii sau adresa datei;
- ☞ **values** – reprezinta lista valorilor pentru Load-ul respectiv;
- ☞ **automat** – este un automat cu patru stari. Un Load este nepredictibil daca automatul acestuia se afla în starea 0 sau 1 si este predictibil daca automatul se afla în starea 2 sau 3;
- ☞ **stride** – reprezinta doua câmpuri în care sunt memorati cei doi pasi utilizati în predictia incrementală.

Aceste câmpuri sunt folosite doar de predictorul incremental.

Presupunând ca pe lângă corelatia dintre adresele instructiunilor Load si valorile citite din memorie de catre acestea, exista o corelatie si între adresele datelor aferente instructiunilor Load si valorile de la acele adrese, simulatorul a fost proiectat în asa fel încât sa permita utilizarea în predictie atât a adresei instructiunii cât si a adresei datei. Alti parametri importanti ai simulatorului care pot fi modificati de catre utilizator, sunt: tipul simulării

(determinarea localității sau predicția valorilor), “adâncimea” istoriei, tipul tabelii de predicție (mapată direct sau asociativă), dimensiunea tabelii de predicție (numărul de locații), tipul predictorului utilizat (*last value*, incremental, contextual sau hibrid) și dimensiunea contextului (*pattern*) în cazul predictorilor contextuale și hibride.

La citirea unei instrucțiuni Load din cache sau memoria centrală, în cazul unei tabelii VHT (Value History Table) mapată direct, cu cei mai puțin semnificativi biți ai adresei instrucțiunii Load ( $PC_{LOW}$ ) se adresează tabela VHT (Value History Table). Maparea se face după următoarea formulă:

$$\text{index} = \text{addr} \bmod \text{VHTdim},$$

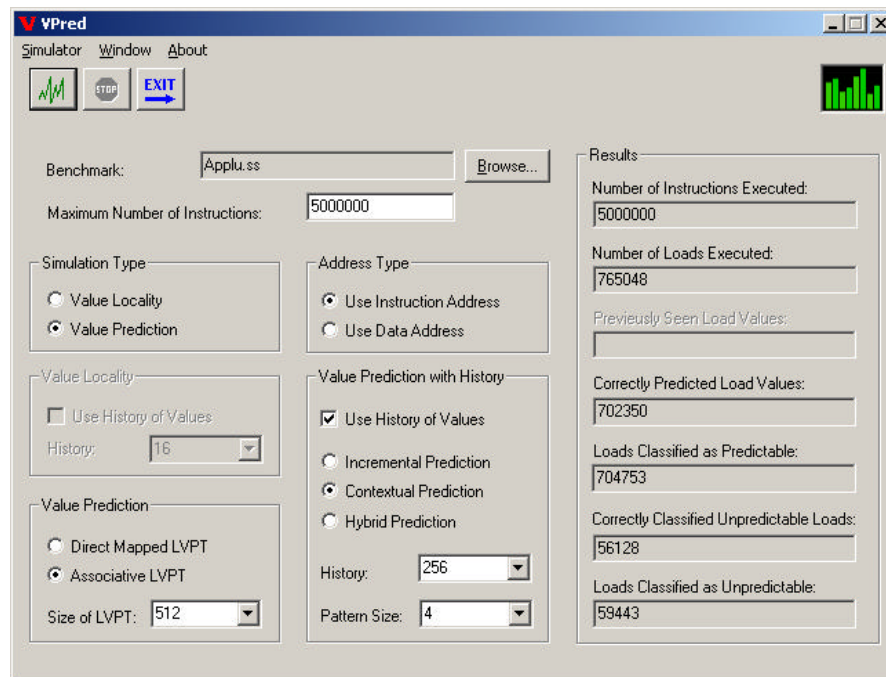
unde *addr* reprezintă adresa instrucțiunii sau adresa datei, iar *VHTdim* reprezintă dimensiunea tabelii de predicție. Se verifică dacă *addr* este egal cu adresa de la indexul respectiv, caz în care avem hit. În cazul în care valorile nu sunt egale vom avea miss în VHT, nu se poate face predicție, iar locația corespunzătoare indexului calculat va fi actualizată cu noua adresă și cu valoarea adusă din memorie de instrucțiunea Load.

În cazul în care se folosește o tabelă VHT asociativă, *addr* este comparat cu adresa din fiecare locație a tabelii și vom avea hit în cazul în care adresa este găsită. Dacă avem miss în VHT, pe baza algoritmului LRU (Least Recently Used) implementat, se evacuează din tabelă cel mai puțin recent accesat Load și se introduce noua adresă și valoarea citită din memorie. Am dorit să evaluăm performanțele diferitelor predictoare în condițiile utilizării unui algoritm LRU “perfect”, care e mai greu de implementat în hardware. Tabela VHT poate fi privită ca o listă în care cele mai recent accesate Load-uri se află la început, iar cele mai puțin recent accesate Load-uri se află la sfârșit. În cazul unui hit în VHT Load-ul găsit trece pe prima poziție din listă. În cazul unui miss în VHT, se evacuează Load-ul aflat pe ultima poziție, iar noul Load se înserează la începutul listei.

Indiferent de tipul tabelii utilizate (mapată direct sau asociativă), în cazul în care avem hit în VHT se face predicție. În funcția *predictValue* din *vpred.c* sunt implementate: predictorul de tip “last value”, predictorul incremental de tip “2-delta”, predictorul contextual de tip PPM (Prediction by Partial Matching) și predictorul hibrid (incremental, contextual). În această funcție este folosit predictorul corespunzător parametrilor introduși de către utilizator. În cazul în care automatul din locația VHT se află în starea *predictibil*, se înaintea valoarea prezisă și aceasta este preluată prin bypassing de către instrucțiunile dependente aflate în așteptare în stadiile de rezervare. La returnarea datei reale din memorie, aceasta este comparată cu valoarea prezisă, și instrucțiunile dependente executate speculativ fie urmează parcursul normal - nivelul Write Back al structurii pipe - fie sunt retrimise spre execuție. Tabela VHT este actualizată prin incrementarea

automatului în cazul unei predicții corecte sau decrementarea acestuia în cazul unei predicții gresite și introducerea datei citite din memorie, evacuând din locația respectivă cea mai veche valoare. Lista valorilor implementează o structură de tip coadă (pentru reținerea celor mai recente 4 sau mai multe valori).

Rezultatele simulării împreună cu parametrii arhitecturii vor fi scrise într-un fișier (*simout.res*) în directorul curent (unde se afla programele de test și simulatorul *sim-vpred.exe*).



**Figura 10.7.** Fereastra principală a simulatorului

Limitarea duratei simulării sau a numărului de instrucțiuni care să fie executate se poate face prin introducerea unui număr maxim de instrucțiuni. În cazul în care această limitare nu este făcută, simularea poate dura mai multe ore, sau chiar zile. Utilizatorul poate să aleagă tipul de simulare (determinarea localității valorilor sau predicția valorilor), adresa care să fie utilizată (adresa instrucțiunii sau adresa datei), “adâncimea” istoriei, tipul tabelii de predicție (mapata direct sau asociativă) și dimensiunea acesteia. În cazul în care se face predicție fără istorie, predictorul simulat este de tip *last value*. Dacă predicția se face cu istoria valorilor utilizatorul trebuie să aleagă tipul predictorului care poate fi incremental contextual sau hibrid. În cazul unei predicții contextuale sau hibride, trebuie aleasă dimensiunea

contextului (*pattern*). Trebuie precizat că predictorul incremental implementat este de tip “2-delta”, iar cel contextual este de tip PPM (Prediction by Partial Matching). Trebuie observat că rezultatele obținute prin predicție sunt mai slabe decât cele generate prin localitate și datorită faptului că tabelele de predicție sunt limitate dar și datorită modului de implementare al arhitecturilor de predicție (mapate direct, asociative).

#### 10.4. APLICATII PRACTICE PROPUSE SPRE REZOLVARE

1. Folosind predictorul incremental (și indexarea tabelului de predicție cu adresa instrucțiunii) stabiliți influența dimensiunii tabelului VHT asupra acurăției predicției valorii, în cele două cazuri:
  - a) VHT – mapata direct
  - b) VHT - asociativa

Repetati simulările în cazul în care indexarea tabelului de predicție se face cu adresa datei.

2. Determinați influența contextului (*lungimea paternului*) asupra acurăției de predicție în cazul unui predictor contextual: dimensiunea tabelului 512 intrări. Plaja de valori a patern-ului cuprinde valorile 1,2,3,4,6 și 8.
3. Determinați acurătatea de predicție utilizând predictorul hibrid, variind contextul (pattern-ul) în plaja de valori (1,2,3,4,,6,8). Indexarea tabelului de predicție se face întâi cu adresa instrucțiunii apoi cu adresa datei.
4. a) Cunoșcând rezultatele obținute la punctul 5 (vezi capitolul - “Extinderea mediului SimpleScalar cu modulul de măsurare a gradelor de localitate”), să se prezinte comparativ acurătatea de predicție în funcție de schema de predicție folosită: LastValue, incremental, contextual, hibrid. Se vor prezenta rezultatele simulărilor în ambele situații: indexarea tabelului de predicție cu adresa instrucțiunii respectiv adresa datei.  
 b) Cuantificați procentual creșterea în acurătate a predicției adusă de predictorul *incremental*, respectiv *contextual* și *hibrid* față de predictorul de tip “*last value*”.
5. a) La simulatorul din lucrarea anterioară (“*Extinderea mediului SimpleScalar cu modulul de măsurare a gradelor de localitate*”) “LastValue” adăugați o opțiune de simulare nouă (fie **-incremental**) care dacă este 0 simulează predictorul LastValue iar dacă este 1 pe cel incremental. Pentru realizarea cu succes a acestei cerințe se va ține cont în primul rând de structurile de date și funcțiile implementate :

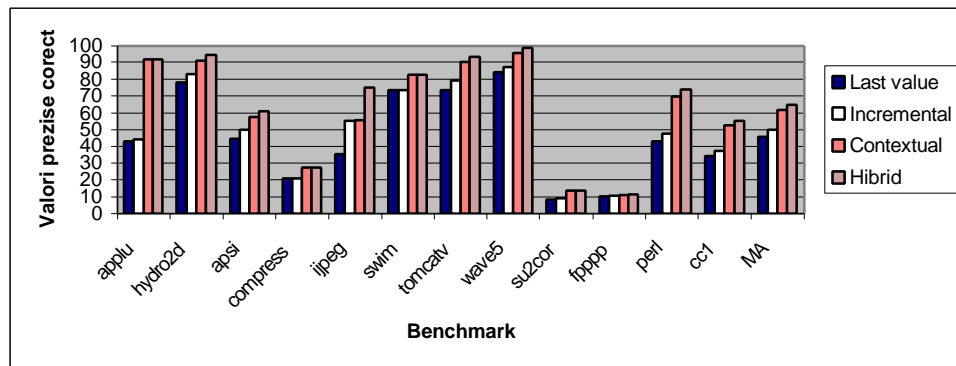
LVPTlocation, found AssociativeLVPTAddress (adresa, valoare) din lucrarea anterioara.

b) Verificati ca dupa modificarile efectuate, simulatoarele setului SimpleScalar (în principiu doar unul a fost modificat), înca functioneaza perfect. Rulati sursa modificata (simulatorul) pe un singur fisier de configurare si sesizati modificarile în fisierul cu rezultate statistice. Comparati rezultatele obtinute cu cele generate de simulatorul **LastValuePredictor** modificat.

#### 10.4.1. REPREZENTAREA GRAFICA A REZULTATELOR SIMULARILOR EFECTUATE

Se va urmări ca rezultatele simulării să respecte formatul ilustrat mai jos. Simularile vor fi efectuate pe benchmark-urile SPEC avute la dispoziție; extrageți concluzii aferent fiecărui grafic în parte.

4a)



**Figura 10.8.** Compararea celor patru tehnici de predictie pentru adresa instructiunii

4b)

Predictor	Predictie cu adresa datei
Incremental	3,89 %
Contextual	12,36 %
Hibrid	15,61 %

*Tabelul 10.1.*

**Cresterea de performanta obtinuta cu cele trei predictoare comparativ cu predictorul de tip “last value”**



## 11. PROBLEME PROPUSE SPRE REZOLVARE

---

1. Consideram un procesor scalar pipeline cu 5 nivele (IF, ID, ALU, MEM, WB) si o secventa de 2 instructiuni succesive si dependente RAW, în doua ipostaze:

A.            i1: LOAD R<sub>1</sub>, 9(R<sub>5</sub>)  
              i2: ADD R<sub>6</sub>, R<sub>1</sub>, R<sub>3</sub>

B.            i1: ADD R<sub>1</sub>, R<sub>6</sub>, R<sub>7</sub>  
              i2: LOAD R<sub>5</sub>, 9(R<sub>1</sub>)

- a. Stabiliti cu ce întârziere (*Delay Slot*) startea a doua instructiune ?  
b. Dar daca se aplica mecanismul de *forwarding* ?  
În acest caz, pentru secventa **B**, cât ar fi fost întârzierea daca în cazul celei de a doua instructiuni, modul de adresare nu ar fi fost *indexat* ci doar *indirect registru* ? Comentati.  
c. Verificati rezultatele obtinute pe procesorul DLX. Determinati cresterea de performanta obtinuta aplicând mecanismul de *forwarding*  $[(IR_{Cu\ forwarding} - IR_{Fara\ forwarding}) / IR_{Fara\ forwarding}]$ .

**Obs.** Se considera ca operanzii instructiunilor sunt necesari la finele fazei ID, iar rezultatele sunt disponibile în setul de registri generali la finele fazei WB.

2. Scrieti o secventa de program asamblare RISC care sa reprezinte translatarea corecta a programului scris în limbaj C ? Initial, registrii R<sub>i</sub>, R<sub>k</sub>, R<sub>l</sub>, R<sub>j</sub>, R<sub>m</sub> contin respectiv variabilele i, k, l, j si m.

$K = X[i-4] + 12;$

$L = Y[j+5] \text{ XOR } K;$

$$M = K \text{ AND } L;$$

Se considera programul executat pe un procesor scalar pipeline RISC cu 4 nivele (IF, ID, ALU/MEM, WB) și operații instrucțiunilor sunt necesari la finele fazei ID, iar rezultatele sunt disponibile în setul de registre generali la finele fazei WB. Se cere:

- a. Reprezentați graful dependentelor de date (numai dependentele de tip RAW).
- b. În câte impulsuri de tact se execută secvența de program asamblare ?
- c. Reorganizați această secvență în vederea minimizării timpului de execuție (se considera că procesorul detine o infinitate de registre generali).
- d. Aplicând tehnica de *forwarding*, în câte impulsuri de tact se execută secvența reorganizată ?

3. Se considera o arhitectura superscalară caracterizată de următorii parametri (vezi **Figura 7.8: Schema bloc arhitecturii superscalare Harvard simulate** – L. Vintan, A. Florea – “Microarhitecturi de procesarea a informației”, Editura Tehnica, București, 2000, pag. 199):

**FR** = 4 instr. / ciclu; – nr. instrucțiuni citite simultan din cache-ul de instrucțiuni sau memoria de instrucțiuni în caz de miss în cache.

**IRmax** = 2 (respectiv 4) instr. / ciclu; – nr. maxim de instrucțiuni independente lansate simultan în execuție.

**N\_PEN** = 10 impulsuri de tact; – nr. impulsuri de tact penalizare necesari accesului la memoria de instrucțiuni în caz de miss în cache.

**Latenta** = 2 impulsuri de tact; – nr. impulsuri de tact necesari execuției pentru orice tip de instrucțiune, lansată din buffer-ul de prefetch (unități de execuție nepipeline-izate).

**IBS** = 8 locații (instrucțiuni); – dimensiunea buffer-ului de prefetch.

**RmissIC** = 40%; – rata de miss în cache-ul de instrucțiuni (din 5 citiri din IC, primele 2 sunt cu miss).

Considerăm următoarea secvență succesivă de 20 instrucțiuni, caracterizată de hazarduri RAW aferente, executată pe arhitectura dată.

i1 – i2 – **i3 – RAW – i4** – i5 – **i6 – RAW – i7** – i8 – i9 – i10 – **i11 – RAW – i12** – i13 – i14 – i15 – i16 – **i17 – RAW – i18** – i19 – i20. (în continuare “nu mai sunt instructiuni de executat”)

**Obs.** În cadrul unui ciclu de executie se realizeaza urmatoarele: din partea inferioara a buffer-ului de prefetch sunt lansate maxim IRmax instructiuni independente, iar simultan în partea superioara a buffer-ului sunt aduse, daca mai e spatiu disponibil, FR instructiuni din cache-ul sau memoria de instructiuni.

Determinati cresterea de performanta (studiu asupra ratei medii de procesare) prin varierea parametrului IRmax de la 2 la 4 instructiuni / ciclu. Prezentati în fiecare ciclu de executie continutul buffer-ului de prefetch.

4. Consideram un procesor RISC scalar pipeline caracterizat printre altele de urmatoarele instructiuni:

ADD	Ri, Rj, Rk – al doilea operand poate fi si valoare imediata
LD	Ri, adresa
ST	adresa, Ri
MOV	Ri, Rj
BEQ	Ri, Rj, label
BNE	Ri, Rj, label
J	label

a. Acest procesor executa urmatoarea secventa:

ST	(R9), R6
LD	R10, (R9)

Rescrieti aceasta secventa folosindu-va de instructiunile cunoscute pentru a elimina **ambiguitatea referintelor la memorie** aparute în secventa originala data si a le executa mai rapid.

b. Se da secventa de instructiuni de mai jos:

ST	4(R5), R8
LD	R9, 8(R6)

Realizati o noua secventa cât mai rapida care sa înlocuiasca în mod corect pe cea de mai sus si care sa elimine posibila **ambiguitatea a referintelor la memorie**, favorizând executia instructiunii LD înaintea lui ST.

5. Dându-se urmatoarele secvente de instructiuni care implica dependente reale de date (RAW) sa se rescrie aceste secvente, cu un numar minim de instructiuni si folosind doar aceiasi registri (eventual R0 = 0 suplimentar), dar eliminând dependentele respective.

**Obs.** Unele instructiuni pot ramâne nemodificate.

- a)    MOV   R6, R7  
       ADD   R3, R6, R5
- b)    MOV   R6, #4  
       ADD   R7, R10, R6  
       LD     R9, (R7)
- c)    MOV   R6, #0  
       ST     9(R1), R6
- d)    MOV   R5, #4  
       BNE   R5, R3, Label
- e)    ADD   R3, R4, R5  
       MOV   R6, R3

6. Se considera o arhitectura superscalara caracterizata de urmatoarii parametri (vezi **Figura 7.8: Schema bloc arhitecturii superscalare Harvard simulate** – L. Vintan, A. Florea – “Microarhitecturi de procesarea a informatiei”, Editura Tehnica, Bucuresti, 2000, pag. 199):

**FR** = 4 (respectiv 8) instr. / ciclu; – nr. instructiuni citite simultan din cache-ul de instructiuni sau memoria de instructiuni în caz de miss în cache.

**IRmax** = 4 instr. / ciclu; – nr. maxim de instructiuni independente lansate simultan în executie.

**N\_PEN** = 10 impulsuri de tact; – nr. impulsuri de tact penalizare necesari accesului la memoria de instructiuni în caz de miss în cache.

**Latenta** = 2 impulsuri de tact; – nr. impulsuri de tact necesari executiei pentru orice tip de instructiune, lansata din buffer-ul de prefetch (unitati de executie nepipeline-izate).

**IBS** = 8 locatii (instructiuni); – dimensiunea buffer-ului de prefetch.

**RmissIC** = 50% (initial); – rata de miss în cache-ul de instructiuni (din 2 citiri din IC, prima se va considera cu miss).

Pe arhitectura data se proceseaza urmatoarea secventa succesiva de 32 instructiuni, caracterizata de hazarduri RAW aferente.

$i1 - i2 - i3 - \text{RAW} - i4 - i5 - i6 - \text{RAW} - i7 - i8 - i9 - i10 - i11 - \text{RAW} - i12 - i13 - i14 - i15 - i16 - i17 - \text{RAW} - i18 - i19 - i20 - i21 - i22 - \text{RAW} - i23 - i24 - i25 - i26 - i27 - \text{RAW} - i28 - i29 - i30 - i31 - \text{RAW} - i32.$

**Obs.** În cadrul unui ciclu de executie se realizeaza urmatoarele: din partea inferioara a buffer-ului de prefetch sunt lansate maxim IR<sub>max</sub> instructiuni independente, iar simultan în partea superioara a buffer-ului sunt aduse, daca mai e spatiu disponibil, FR instructiuni din cache-ul sau memoria de instructiuni.

Determinati cresterea de performanta (studiu asupra ratei medii de procesare) prin varierea parametrului FR de la 4 la 8 instructiuni / ciclu. De mentionat ca aceasta variatie a FR de la 4 la 8 implica diminuarea ratei de miss în cache-ul de instructiuni cu 50%. Prezentați în fiecare ciclu de executie continutul bufferului de prefetch. În ce consta limitarea performantei în acest caz ?

7. Se considera urmatoarea secventa de instructiuni executata pe un procesor pipeline cu 4 nivele (IF, ID, ALU/MEM, WB), fiecare faza necesitând un tact, cu urmatoarea semnificatie:

**IF** = citirea instructiunii din cache-ul de instructiuni sau din memorie

**ID** = decodificarea instructiunii si citirea operanzilor din setul de registri generali

**ALU / MEM** = executie instructiuni aritmetice sau accesare memorie

**WB** = înscriere rezultat în registrul destinatie

Operanzii instructiunilor sunt necesari la finele fazei ID, iar rezultatele sunt disponibile în setul de registri generali la finele fazei WB.

```

i1:  ADD  Ri, R0, #i
i2:  ADD  Rj, Ri, #4
i3:  LOAD  R1, (Ri)
i4:  LOAD  R2, (Rj)
i5:  ADD  R3, R1, R2
i6:  SUB   R4, R1, R2
i7:  ABS   R4, R4
i8:  ADD  R1, R13, R14
i9:  DIV   R1, R1, #2
i10: STORE (Ri), R1

```

Se cere:

- Reprezentati graful dependentelor de date (RAW, WAR, WAW) si precizati în câte impulsuri de tact se executa secventa? Initial, structura “*pipe*” de procesare este goala, registri initializati cu 0.
- Reorganizati aceasta secventa în vederea minimizarii timpului de executie (se considera ca procesorul detine o infinitate de registri generali disponibili). În câte impulsuri de tact s-ar procesa în acest caz secventa ?
- Ce simuleaza secventa initiala daca în instructiunea i8 în locul registrilor R<sub>13</sub> am avea R<sub>3</sub> si în locul lui R<sub>14</sub> am avea R<sub>4</sub>. Precizati formula matematica obtinuta.

**8. Se considera secventa de program RISC:**

```

i1:  ADD  R1, R2, #15
i2:  ADD  R3, R4, #17
i3:  ADD  R5, R3, R1
i4:  ADD  R6, R5, #12
i5:  ADD  R3, R7, #3
i6:  ADD  R8, R3, #2
i7:  ADD  R9, R8, #14

```

Se cere:

- Sa se construiasca graful dependentelor de date (RAW, WAR, WAW) aferent acestei secvente si precizati în câte impulsuri de tact se executa secventa, stiind ca latenta de executie a instructiunii ADD este de 1 ciclu ?

- b. Sa se determine modul optim de executie al acestei secvente reorganizate, pe un procesor RISC superscalar cu 6 seturi de registri generali si 3 unitati ALU.
9. a) Are sens implemetarea unui bit de validare (V) pentru fiecare bloc din cache în cadrul unui sistem de calcul uniprocessor (fara DMA) ? Justificati.  
b) O exceptie *Page Fault* implica întotdeauna una de tip TLB miss ? Dar reciproc ?
10. a) Ce limitare principiala exista asupra ratei de fetch a instructiunilor ? Cum ar putea fi depasita aceasta limitare ?  
b) Descrieti succint tehnicile de eliminare a dependentelor RAW între instructiuni.  
c) Daca într-o arhitectura tip Tomasulo ar lipsi câmpurile de “tag” ( $Q_j, Q_i$ ) din cadrul statiilor de rezervare, considerati ca ar mai functiona schema ? Justificati punctual.
11. Se considera un microprocesor RISC cu o structura «pipe» de procesare a instructiunilor, având vectorul de coliziune atasat 101011. Sa se determine rata teoretica optima de procesare a instructiunilor pentru acest procesor [instr/ciclu] si sa se expliciteze algoritmul dupa care trebuie introduse instructiunile în structura.
12. Fie vectorul de coliziune 1001011 atasat unui microprocesor RISC cu o structura «pipe» de procesare a instructiunilor. Sa se determine rata teoretica optima de procesare a instructiunilor pentru acest procesor [instr/ciclu] si explicitând algoritmul dupa care trebuie introduse instructiunile în structura. Determinati cresterea de performanta comparativa cu situatia în care s-ar procesa pe o structura CISC conventionala  $[(IR_{RISC} - IR_{CISC}) / IR_{CISC}]$ .
- 13.a) În cadrul unui procesor vectorial se considera urmatoarea secventa de program:
- ```
x = 0
for i = 1 to 100 do
    x = x + A[i]*B[i];
endfor
```

În câți cicli de tact se execută secvența ? Este bucla vectorizabilă ? În caz negativ, scrieți o nouă secvență de program care să aibă același efect dar care să fie executată mult mai rapid ? Determinați noul timp de execuție al secvenței ? Concret din ce motive se câștigă timp de procesare ?

- b) Să se proiecteze un cache de instrucțiuni cuplat la un procesor superscalar (VLIW). Lungimea blocului din cache se consideră egală cu rata de fetch a procesorului, în acest caz 8 instrucțiuni / bloc. Cache-ul va fi de tipul **4 way set associative** (cu 4 blocuri / set). Explicați semnificația fiecărui câmp utilizat (necesar) în proiectare.
  - c) Descrieți avantajelor introduse de Tomasulo în cadrul arhitecturii care îi poartă numele.
- 14.a) Prezentați pașii succesivi aferenți drumului de la o arhitectura parametrizabilă dată, la instanța sa optimă, numită procesor.
- b) De ce nu este suficientă doar optimizarea unităților secvențiale de program (“basic-blocks”), fiind necesară optimizarea globală a întregului program?
  - c) Care sunt în opinia dvs. motivele pentru care microprocesoarele superscalare au un succes comercial net superior celor cu paralelism exploatat prin optimizări de program (“Scheduling” static) – de exemplu microprocesoare tip VLIW, EPIC, etc. ?
15. Relativ la gestiunea memoriei în sistemele moderne, tratați într-o manieră concisă următoarea problemă:
- a. În ce rezidă necesitatea unui sistem de memorie virtuală (MV) ?
  - b. Explicați principiile protecției în sistemele cu MV ?
  - c. Explicați succint rolul TLB (Translation Lookaside Buffer) în traducerea adreselor.
  - d. În ce constă dificultățile implementării unui cache virtual ? Care ar fi avantajele acestuia ?
- 16.a) Caracteristici ale structurilor de program care limitează accelerarea unui sistem paralel de calcul în raport cu unul secvențial;
- b) Tipuri arhitecturale de memorii cache. Avantajele/dezavantajele fiecărei organizări. Explicați succint noțiunea de coerență a cache-urilor în sistemele multiprocesor;



- c) Limitari arhitecturale ale paradigmei superscalare si independente de masina, referitoare la:
- maximizarea *ratei de aducere a instructiunilor din memorie*
  - maximizarea *ratei de executie a instructiunilor*.

17.a) Care este semnificatia termenilor:

- ❖ Arhitectura RISC (Reduced Instruction Set Computer)
- ❖ Arhitectura CISC (Complex Instruction Set Computer)

Care arhitectura este mai rapida si de ce ?

- b) Explicati semnificatia nivelelor pipeline aferente unui microprocesor (IF, ID, ALU, MEM, WB). Explicati avantajele introduse de conceptul de procesare "pipeline".
- c) Ce este o arhitectura VLIW (Very Large Instruction Word) ? Descrieti asemanarile si deosebirile dintre o arhitectura VLIW si una superscalara. Subliniati avantajele arhitecturii VLIW fata de o arhitectura conventionala de procesor.
- d) Un microprocesor cu 4 nivele pipeline (IF, ID, ALU/MEM, WB) executa urmatoarea secventa de cod:

```

ADD          ; Adunare
BR SUB1      ; Salt neconditionat - apel subrutina
SUB          ; Scadere
MUL          ; Înmultire
.
.

```

**SUB1:** ; Intrarea în subrutina - adresa primei  
; instructiuni din subrutina

Ce se întâmpla când procesorul întâlnește instructiunea de salt (situatia instructiunilor în "pipe")? Considerând ca procesorul primește o întrerupere dupa ce executa instructiunea de salt, explicati clar secventa de evenimente petrecute dupa receptia întreruperii.

18. Consideram un procesor pipeline cu 5 nivele (IF, ID, ALU, MEM, WB) în care conditia de salt este verificata pe nivelul de decodificare, operandii instructiunilor sunt necesari la finele fazei ID, iar rezultatele sunt disponibile în setul de registri generali la finele fazei WB. Trasati diagrama ciclului de tact a procesorului incluzând dependentele reale de date (RAW), reprezentati graful dependentelor de date, la executia urmatoarei secvente de instructiuni. Stabiliti cu ce întârziere (*Delay Slot*) startea a doua instructiune în cazul existentei unui hazard RAW între

două instrucțiuni; care este timpul total de procesare al secvenței. Explicit subliniați toate tipurile de hazard (de date, de ramificație) și dacă și unde poate fi aplicat mecanismul de forwarding. Presupunând că instrucțiunea de salt este corect predictionată ca *non-taken* în câți cicli de tact se va procesa secvența ?

```
ADD R1, R2, R3
LD  R2, 0(R1)
BNE R2, R1, dest
SUB R5, R1, R2
LD  R4, 0(R5)
SW  R4, 0(R6)
ADD R9, R5, R4
```

- 19.a) În general un microprocesor consumă multe impulsuri de tact pentru a citi o locație de memorie DRAM, datorită latentei relativ mari a acesteia. Implementarea caror concepte arhitecturale micșorează timpul mediu de acces al microprocesorului la memoria DRAM ?
- b) Care este principalul concept arhitectural prin care se micșorează timpul mediu de acces al procesorului la discul magnetic ?

- 20.a) Care sunt principalele cauze ce limitează performanța unei structuri "pipeline" de procesare a instrucțiunilor masina ?
- b) De ce timpul de execuție al unui program pe un multiprocesor cu N procesoare nu este în general de N ori mai mic decât timpul de execuție al aceluiași program pe un sistem uniprocessor ?
- c) La ce se referă necesitatea coerenței memoriilor cache dintr-un sistem multiprocesor ? Ce strategii principale de menținere a coerenței memoriilor cache cunoașteți ?

21. Să se proiecteze un cache de instrucțiuni cuplat la un procesor superscalar (VLIW). Lungimea blocului din cache se consideră egală cu rata de fetch a procesorului, în acest caz 4 instrucțiuni / bloc. Cache-ul va fi de tipul:

- a) semiasociativ, cu 2 blocuri / set (2 – way set associative)
- b) complet asociativ (full - associative)
- c) cu mapare directă (direct mapped)

Ce se întâmplă dacă în locul adresării cu adrese fizice se consideră adresare cu adresă virtuală?

22. a) Ce diferente exista între doua procesoare superscalare, unul cu executie "*in order*" si altul cu executie "*out of order*" ? Cum reuseste fiecare dintre ele sa extraga si sa creasca paralelismul existent la nivelul instructiunilor ? Care sunt principalele elemente structurale care compun modelul de executie Out of Order. Ce functii realizeaza aceste elemente ?

b) De ce este dificila procesarea out-of-order a instructiunilor cu referire la memorie ? Ce presupune analiza anti-alias a referintelor la memorie? Cum este implementata în cazul simulatorului SATSim ? Care dintre cele doua secvente de program s-ar putea procesa mai rapid pe un procesor superscalar cu executie «Out of Order» a instructiunilor? Justificati.

**B1.**

```
for i=1 to 100
  a[2i]=x[i];
  y[i]=a[i+1]+5;
```

**B2.**

```
a[2]=x[1];
y[1]=a[2]+5;
for i=2 to 100
  a[2i]=x[i];
  y[i]=a[i+1]+5;
```

c) Descrieti pe scurt rolul buffer-elor de redenumire si reordonare în cazul simulatorului SATSim. Cum sunt implementate cele doua structuri ? Cunoasteti vreo arhitectura în care rolul buffer-ului de "*renaming*" sa fie preluat de cel de reordonare ? Considerati ca toate instructiunile au nevoie de o intrare în buffer-ul de redenumire si cel de reordonare? Justificati.

23. Se considera un procesor scalar pipeline, în 3 faze diferite de procesare (IF,EX,WR), fiecare faza necesitând un tact, cu urmatoarea semnificatie:

**IF** = aducere si decodificare a instructiunii

**EX**=selectie operanzi din setul de registri si executie

**WR**=înscrisere rezultat în registrul destinatie

Se considera secventa de program:

```
1: R1<-- (R11)+(R12)
2: R1<-- (R1)-(R13)
3: R2 <-- (R3)+4
4: R2 <-- (R1)*(R2)
5: R1<-- (R14)+(R15)
6: R1<-- (R1)+(R16)
```

- a) În câte impulsuri se executa secventa? (initial, structura «pipe» de procesare este «goala») Reorganizati aceasta secventa de program în vederea minimizarii timpului de executie (procesorul detine o infinitate de registri generali disponibili). În câte impulsuri de tact s-ar procesa în acest caz secventa ?
  - b) În câte tacte (minimum) s-ar procesa secventa daca procesorul ar putea executa simultan un numar nelimitat de instructiuni independente? Se considera ca procesorul poate aduce în acest caz, simultan, 6 instructiuni din memorie. Justificati.
24. Se considera o structura «pipe» de procesare a instructiunilor având un nivel de citire a operanzilor din setul de registri (RD), situat anterior unui nivel de scriere a rezultatului în setul de registri (WR). Careia dintre cele doua operatii (RD, WR) i se da prioritate în caz de conflict si în ce scop ?
25. Se considera ca 20% dintre instructiunile unui program determina ramificarea acestuia (salt efectiv). Care ar fi în acest caz rata de fetch (FR) posibila pentru un procesor superscalar (VLIW – Very Long Instruction Word) având resurse hardware nelimitate si o predictie perfecta a branch-urilor (cunoastere anticipata a adresei de salt) ? Este posibila o depasire a acestei limitari fundamentale ? Daca da, care ar fi noua limitare impusa parametrului FR prin solutia Dvs. ?
26. Un procesor superscalar poate lansa în executie simultan maxim N instructiuni ALU independente. Logica de detectie a posibilelor hazarduri RAW (Read After Write) între instructiunile ALU are costul «C» (\$). Cât va costa logica de detectie daca s-ar dori ca sa se poata lansa simultan în executie maxim (N+1) instructiuni ALU independente ? (Se vor considera costurile ca fiind direct proportionale cu «complexitatea» logicii de detectie a hazardurilor RAW).
27. Relativ la o memorie cache cu mecanism de adresare tip «mapare directa», precizati valoarea de adevar a afirmatiilor de mai jos, cu justificarile de rigoare.
- a) Rata de hit creste daca capacitatea memoriei creste;
  - b) Data de la o anumita locatie din memoria principala poate fi actualizata la orice adresa din cache;

- c) Scrieri în cache au loc numai în ciclurile de scriere cu miss în cache;
- d) Are o rata de hit net mai mare decât cea a unei memorii complet asociative si de aceeași capacitate.

**28.** Se considera secventa de program RISC:

```
1:  ADD  R1, R11, R12
2:  MUL  R1, R1, R13
3:  ADD  R2, R3, R9
4:  SUB  R2, R1, R2
5:  ADD  R1, R14, R15
```

- a) Reprezentați graful dependentelor de date (numai dependentele de tip RAW)
- b) Știind că între 2 instrucțiuni dependente RAW și succesive e nevoie de o întârziere de 2 cicli, în câți cicli s-ar executa secvența ?
- c) Reorganizați secvența în vederea unui timp minim de execuție (nu se considera alte dependente decât cele de tip RAW).

- 29.a)** Considerând un procesor RISC pe 5 nivele pipe (IF, ID, ALU, MEM, WB), fiecare durând un ciclu de tact, precizați câți cicli de întârziere («branch delay slot») impune o instrucțiune de salt care determină adresa de salt la finele nivelului ALU ?
- b) De ce se preferă implementarea unor busuri și memorii cache separate pe instrucțiuni, respectiv date în cazul majorității procesoarelor RISC (pipeline) ?
  - c) De ce sunt considerate instrucțiunile CALL / RET mari consumatoare de timp în cazul procesoarelor CISC (ex. I-8086) ? Cum se evită acest consum de timp în cazul microprocesoarelor RISC ?

- 30.** Considerând un microprocesor virtual pe 8 biți, având 16 biți de adrese, un registru A pe 8 biți, un registru PC și un registru index X, ambele pe 16 biți și ca opcode-ul oricărei instrucțiuni e codificat pe 1 octet, să se determine numărul impulsurilor de tact necesare aducerii și execuției instrucțiunii «memorează A la adresa dată de (X+deplasament)». Se considera că instrucțiunea e codificată pe 3 octeți și că orice procesare (operație internă) consumă 2 tacte. Un ciclu de fetch opcode durează 6 tacte și orice alt ciclu extern durează 4 tacte.

**31.** Relativ la o arhitectura de memorie cache cu mapare directă se considera afirmațiile:

- a) Nu permite accesul simultan la câmpul de date și respectiv «tag» al unui cuvânt accesat.
- b) La un acces de scriere cu hit, se scrie în cache atât data de înscris cât și «tag-ul» aferent.
- c) Rata de hit crește ușor dacă 2 sau mai multe blocuri din memoria principală - accesate alternativ de către microprocesor - sunt mapate în același bloc din cache.

Stabiliți valoarea de adevăr a acestor afirmații și justificați pe scurt răspunsul.

**32.** Ce corectie (doar una!) trebuie făcută în secvența de program asamblare pentru ca translatarea de mai jos să fie corectă și de ce? Inițial, registrele  $R_i$ ,  $R_k$ ,  $R_l$ ,  $R_j$  conțin respectiv variabilele  $i$ ,  $k$ ,  $l$ ,  $j$ . Primul registru după mnemonica este destinație.  $(R_j + \text{offset})$  semnifică operand în memorie la adresa dată de  $(R_j + \text{offset})$ .

|                   |                            |
|-------------------|----------------------------|
| $k = a[i+2] + 5;$ | i1:    ADD $R_k, \#2, R_i$ |
| $l = c[j+9] - k;$ | i2:    LOAD $R_k, (R_k+0)$ |
|                   | i3:    ADD $R_k, R_k, \#5$ |
|                   | i4:    ADD $R_l, \#9, R_j$ |
|                   | i5:    LOAD $R_l, (R_j+0)$ |
|                   | i6:    SUB $R_l, R_l, R_k$ |

**33.** Se considera un microsistem realizat în jurul unui microprocesor care ar accepta o frecvență maximă a tactului de 20 MHz. Regenerarea memoriei DRAM se face în mod transparent pentru microprocesor. Procesul de regenerare durează 250 ns. Orice ciclu extern al procesorului durează 3 perioade de tact. Poate funcționa în aceste condiții microprocesorul la frecvența maximă admisă? Justificați.

**34.** Explicați concret rolul fiecăreia dintre fazele de procesare (ALU, MEM, WB) în cazul instrucțiunilor:

- a) STORE  $R_5$ ,  $(R_9)06h;$   
sursa

b) LOAD R7, (R8)F3h;  
dest

c) AND R5, R7, R8.  
dest

**35.** Se considera un procesor scalar pipeline, în 3 faze diferite de procesare (IF, EX, WR), fiecare faza necesitând un tact, astfel:

**IF** = fetch instructiune si decodificare;

**EX** = selectie operanzi din setul de registri si executie;

**WB** = înscriere rezultat în registrul destinatie.

- a) În câte impulsuri de tact se executa secventa de program de mai jos ?  
b) Reorganizati aceasta secventa în vederea minimizarii timpului de executie.

```
1:  ADD  R3, R2, #2
2:  ADD  R1, R9, R10
3:  ADD  R1, R1, R3
4:  ADD  R2, R3, #4
5:  ADD  R2, R1, R2
6:  STORE R3, (R1)2
```

**36.** Un procesor pe 32 biti la 50 MHZ, lucreaza cu 3 dispozitive periferice prin interogare. Operatia de interogare a starii unui dispozitiv periferic necesita 100 de tacte. Se mai stie ca:

- a) interfata cu mouse-ul trebuie interogata de 30 de ori / s pentru a fi siguri ca nu se pierde nici o «miscare» a utilizatorului.  
b) floppy - discul transfera date spre procesor în unitati de 16 biti si are o rata de transfer de 50 ko / s.  
c) hard - discul transfera date spre procesor în unitati de 32 biti si are o rata de transfer de 2 Mo / s.

Determinati în [%], fractiunea din timpul total al procesorului, necesara interogarii starii fiecarui periferic. Comentati.

37. a) Care este rolul fundamental al conceptului de "**Selective Victim Cache**" într-un sistem ierarhizat de memorie (în care cache-ul principal este de *cu mapare directa*) ? Descrieti principial si sistematic (cazuri) algoritmul respectiv. Explicitati rolul bitilor de **hit**, respectiv **sticky** si al blocului tranzitoriu în cadrul algoritmului de predictie.  
 b) Prin ce este superioara arhitectura SVC fata de o arhitectura VC echivalenta (rata de hit, numar de interschimbari, timp mediu de acces). Explicitati.  
 c) Care este tendinta din punct de vedere al ratei de miss pentru cache-ul principal respectiv al ratei de utilizare pentru victim cache în conditiile cresterii capacitatii cache-ului principal.
38. a) Descrieti pe scurt metodologia de simulare în cadrul schemei de predictie BTB (GAg) având la dispozitie trace-urile Stanford.  
 b) Se considera un automat de predictie pe doi biti (numarator saturat cu 4 stari - similar celui prezentat în capitolul 6.2). Descrieti **clasa Automat** cu attributele si metodele componente, asociata functionarii respectivului automat, punctând necesitatea fiecarui membru.  
 c) Se considera un simulator al schemei de predictie de tip GAg (similar cu cel descris în capitolul 6.3.2). Simulatorul prezinta trei câmpuri de editare cu urmatoarele semnificatii:
- ☐ **Rata de hit** - exprima numarul de accese cu hit în tabela PHT (cazurile când TAG-ul instructiunii respective de salt a fost gasit în tabela)
  - ☐ **Salturi facute** - contorizeaza numarul de salturi prezise ca "taken" (predictie - *se face*) de catre automatul de predictie în momentul determinarii predictiei.
  - ☐ **Salturi nefacute** - retine numarul de salturi prezise ca "not taken" (predictie - *nu se face*) de catre automat.
- Ce relatie exista între cele trei câmpuri descrise anterior ?
39. a) Descrieti succint metodologia de simulare a interfetei *procesor - cache* pentru o arhitectura RISC superscalara parametrizabila, folosind trace-urile Stanford.  
 b) Ce limitari trebuie impuse asupra unora dintre parametrii de simulare **FR** (rata de aducere a instructiunilor din cache sau memorie), **IBS** (capacitatea buffer-ului de prefetch), **IRmax** (numarul maxim de instructiuni ce pot fi lansate simultan în executie), **SIZEIC** (dimensiunea cache-ului de instructiuni)) si în ce relatie trebuie sa se afle acestia pentru obtinerea performantei optime ?



c) Care este influenta numarului de seturi de registri generali asupra ratei de procesare într-un procesor superscalar generic ? Care este valoarea maxima utila ? De ce aceasta valoare maxima nu conduce neaparat si la performanta maxima ? Se considera ca un set de registri generali este necesar pentru executia unei instructiuni de tip aritmetico-logic sau cu referire la memorie.

**40.** Consideram 3 memorii cache care contin 4 blocuri a câte un cuvânt / bloc. Una este complet asociativa, alta semiasociativa cu 2 seturi a câte 2 cuvinte si ultima cu mapare directa. Stiind ca se foloseste un algoritm de evacuare de tip LRU, determinati numarul de accese cu HIT pentru fiecare dintre cele 3 memorii, considerând ca procesorul citește succesiv de la adresele 0, 8, 0, 6, 8, 10, 8 (primul acces la o anumita adresa va fi cu MISS).

**41.** Se considera secventa de program RISC:

```
1:  ADD  R3, R2, #2
2:  ADD  R1, R9, R10
3:  SUB  R1, R1, R3
4:  ADD  R2, R3, #4
5:  ADD  R2, R1, R2
```

Între doua instructiuni dependente RAW si succesive în procesare, e nevoie de o întârziere de 1 ciclu de tact.

a) În câți cicli de tact se executa secventa initiala ?

b) În câți cicli de tact se executa secventa reorganizata aceasta secventa în vederea unui timp minim de procesare ?

**42.** Se considera o unitate de disc având rata de transfer de  $25 \times 10^4$  biti/s, cuplata la un microsystem. Considerând ca transferul între dispozitivul periferic si CPU se face prin întrerupere la fiecare octet, în mod sincron, ca timpul scurs între aparitia întreruperii si intrarea în rutina de tratare este de  $2 \mu s$  si ca rutina de tratare dureaza  $10 \mu s$ , sa se calculeze timpul pe care CPU îl are disponibil între 2 transferuri succesive de octeti.

**43.** a. Daca rata de hit în cache ar fi de 100%, o instructiune s-ar procesa în 8.5 cicli de tact. Sa se exprime în [%] scaderea medie de performanta daca rata de hit devine 89%, orice acces la memoria principala se

desfasoara pe 6 tacte si ca orice instructiune face 3 referinte la memorie.

- b. De ce e avantajoasa implementarea unei pagini de capacitate «mare» într-un sistem de memorie virtuala ? De ce e dezavantajoasa aceasta implementare ? Pe ce baza ar trebui facuta alegerea capacitatii paginii ?

**44.** Se considera un procesor scalar pipeline, în 3 faze diferite de procesare (IF, EX, WR), fiecare faza necesitând un tact, astfel:

**IF** = fetch instructiune si decodificare;

**EX** = selectie operanzi din setul de registri si executie;

**WB** = înscriere rezultat în registrul destinatie.

- a) În câte impulsuri de tact se executa secventa de program de mai jos ?  
 b) Reorganizati aceasta secventa în vederea minimizarii timpului de executie (se considera ca procesorul detine o infinitate de registri generali).

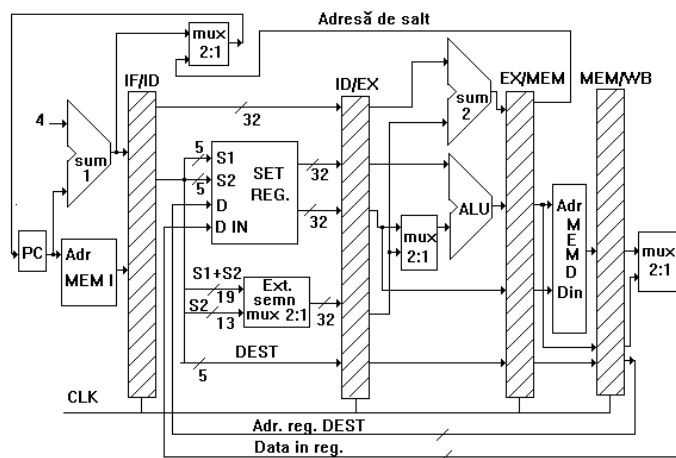
```

1:  R1 ← (R11) + (R12)
2:  R1 ← (R1) + (R13)
3:  R2 ← (R3) + 4
4:  R2 ← (R1) + (R2)
5:  R1 ← (R14) + (R15)
6:  R1 ← (R1) + (R16)
  
```

**45.** Se considera un microprocesor RISC cu o structura «pipe» de procesare a instructiunii, având vectorul de coliziune atasat 01011. Sa se determine rata teoretica optima de procesare a instructiunii pentru acest procesor [instr/ciclu].

**46.** De ce implementarea algoritmului lui R. TOMASULO într-o arhitectura superscalara ar putea reduce din «presiunea» la citire asupra seturilor de registri generali ? Gasiti vreo similitudine în acest sens, între un CPU superscalar având implementat acest algoritm si un CPU de tip TTA (Transport Triggered Architecture) ?

47. De ce considerati o instructiune de tip RETURN este mai dificil de predictionat printr-un predictor hardware ? Puteti sugera vreo solutie în vederea eliminarii acestei dificultati ? În ce consta noutatea «principiala» a predictoarelor corelate pe doua nivele ?
48. Cum credeti ca s-ar putea masura printr-un simulator de tip «trace driven», câstigul de performanta introdus de tehnicile de paralelizare a buclelor de program (ex. «Loop Unrolling», «Software Pipelining», etc.)
49. Cum explicati posibilitatea interblocarii proceselor în cadrul limbajului OCCAM ? Ce înțelegeti prin «sectiune critica de program» în cadrul unui sistem multimicro ? Care este «mesajul» transmis de «legea lui AMDAHL» pentru sistemele paralele de calcul ?
50. Se considera structura hardware a unui microprocesor RISC, precum în figura de mai jos.



Raspundeti la urmatoarele întrebări.

- Ce tip de instructiuni activeaza sumatorul «sum 2» si în ce scop ?
- Într-un tact, la setul de registri pot fi necesare 2 operatii simultane: citire (nivelul RD din pipe), respectiv scriere (nivelul WB din pipe). Carei operatii i se da prioritate si în ce scop ?
- Ce rol are unitatea ALU în cazul unei instructiuni de tip LOAD ?
- Ce informatie se memoreaza în latch-ul EX/MEM în cazul instructiunii: ST (R7)05, R2 si de unde provine fiecare informatie ?

51. Se considera secventa de program RISC:

```
1:    SUB   R7, R2, R12
2:    ADD   R1, R9, R10
3:    ADD   R1, R1, R7
4:    SUB   R2, R7, R12
5:    ADD   R2, R1, R2
6:    ADD   R1, R6, R8
7:    ADD   R1, R1, R7
8:    SUB   R1, R1, R12
9:    LD     R1, (R1)2
10:   LD     R4, (R4)6
11:   ADD   R1, R4, R1
12:   ADD   R1, R1, R2
13:   ST     R1, (R4)16
14:   ST     R7, (R1)16
```

- a) Sa se construiasca graful dependentelor / precedentelor de date aferent acestei secvente. Cu exceptia LOAD-urilor care au latenta de 2 cicli, restul instructiunilor au latenta de 1 ciclu.
- b) În baza algoritmului LIST SCHEDULING, sa se determine modul optim de executie al acestei secvente (nr. cicli), pentru un procesor superscalar având 2 unitati ADD, 1 unitate SUB si 1 unitate LOAD/STORE. Unitatea pentru LOAD este nepipeline-izata.

52. Pe durata executiei unui program principal o subrutina este apelata **exclusiv** printr-o instructiune CALL situata la o anumita adresa în acest program, de 611 de ori. De ce este dificil de predictionat în acest caz instructiunea RETURN de la finele subrutinei apelate?

În limbaj de asamblare MIPS prezentati solutiile urmatoarelor probleme:

53. Scrieti un program folosind recursivitatea, care citeste de la tastatura doua numere întregi pozitive si afiseaza cel mai mare divizor comun si cel mai mic multiplu comun al celor doua numere.

54. Scrieti un program recursiv care rezolva problema Turnurilor din Hanoi pentru  $n$  discuri ( $n$  – parametru citit de la tastatura). Enuntul problemei este urmatorul:

*Se dau trei tije simbolizate prin A, B si C. Pe tija A se gasesc  $n$  discuri de diametre diferite, asezate în ordine descrescatoare a diametrelor privite de jos în sus. Se cere sa se mute discurile de pe tija A pe tija B, folosind tija C ca tija de manevra, respectându-se următoarele reguli:*

- ❖ La fiecare pas se muta un singur disc.
- ❖ Nu este permis sa se aseze un disc cu diametrul mai mare peste un disc cu diametrul mai mic.

55. Realizati un program care citeste de la tastatura doua numere naturale  $n$  si  $k$  ( $n > k$ ) si calculeaza si afiseaza pe consola valorile urmatoare:

$$C_n^k \text{ si } A_n^k$$

56. Se citeste un vector cu componente numere naturale de la tastatura. Dimensiunea sa ( $n$ ) este citita tot de la tastatura. Sa se tipareasca numarul cifrelor de 0 cu care se termina numarul format din produsul celor  $n$  componente fara a calcula însa produsul.

57. Scrieti un program care afiseaza primele  $n$  perechi de numere prime impare consecutive ( $n$  - numar impar citit de la tastatura). Exemplu: (3,5), (5,7), etc.

58. (**Conjectura lui Goldbach**) Scrieti un program în limbaj de asamblare DLX, care citeste de la tastatura, prin intermediul modulului Input.s, un numar  $n$  par,  $n > 6$ . Sa se determine toate reprezentările lui  $n$  ca suma de numere prime, suma cu numar minim de termeni. Afisarea se face pe consola pe fiecare linie câte o solutie.

59. Un procesor pipeline ideal (rata de hit în cache-uri = 100% si timpul de acces la cache egal cu 1 ciclu de tact) are rata de procesare  $R_i = 1$  instr/ciclu. Cât este rata de procesare a unui procesor echivalent, dar care are latentă la cache-ul de date de 2 cicli, stiind ca 30% din instructiuni acceseaza acest cache (cu hit).

- 60.** Proiectați automatul de control cache, într-un sistem multimicroprocesor simetric pe bus comun, având în vedere că un bloc din cache se poate afla într-una din stările: PARTAJAT, EXCLUSIV sau INVALID.
- 61.** Se considera un sistem multimicroprocesor (SMM) cu  $N$  microprocesoare legate printr-o rețea de interconectare (RIC) la  $N$  module fizice de memorie. În ce ar consta și ce ar permite o RIC cu lățime de bandă maximă? Dar una cu lățime de bandă minimă?
- 62.** Într-un SMM cu memorie centrală partajată și în care fiecare procesor deține un cache propriu, un procesor inițiază o scriere cu MISS într-un anumit bloc aflat în starea “partajat” (“shared”). Precizați procesele succesive care au loc în urma acestei operații.
- 63.** Într-un SMM un procesor inițiază o citire cu MISS la un bloc invalid în cache-ul propriu, blocul aflându-se în copia exclusivă într-alt procesor. Precizați concret procesele succesive care au loc în urma acestei operații.
- 64.** Se considera un procesor cu un cache conectat la o memorie principală printr-o magistrală (bus de date) de 32 de biți; un acces cu hit în cache durează un ciclu de tact. La un acces cu miss în cache întregul bloc trebuie extras din memoria principală prin intermediul magistralei. O tranzacție pe bus constă dintr-un ciclu de tact pentru trimiterea adresei pe 32 de biți spre memorie, 4 cicluri de tact în care are loc accesarea memoriei (strobarea datelor pe bus) și 1 ciclu pentru transferarea fiecărui cuvânt de date (32 octeți) în blocul din cache. Se presupune că procesorul continuă execuția doar după ce ultimul cuvânt a fost adus în cache. Următorul tabel exprimă rata medie de miss într-un cache de 1Mbyte pentru diverse dimensiuni a blocului de date.

| Dimensiunea blocului (B), în cuvinte | Rata de miss (m), în % |
|--------------------------------------|------------------------|
| 1                                    | 4,5                    |
| 4                                    | 2,4                    |
| 8                                    | 1,6                    |
| 16                                   | 1,0                    |
| 32                                   | 0,75                   |

Se cere:

- a) Pentru care din blocuri (pentru ce dimensiune) se obtine cel mai bun timp mediu de acces la memorie ?
- b) Daca accesul la magistrala adauga doi cicli suplimentari la timpul mediu de acces la memoria principala (disputa pentru ocuparea bus-ului), care din blocuri determina optima valoare pentru timpul mediu de acces la memorie ?
- c) Daca latimea magistralei este dublata la 64 de biti, care este dimensiunea optima a blocului de date din punct de vedere al timpului mediu de acces la memorie ?

65. Un procesor pipeline ideal are rata de procesare ideala  $R_{ideal} = 1$  instr/ciclu. Care este rata de procesare a unui procesor echivalent, dar care nu detine mecanism de predictie a salturilor. Se stie ca 25% dintre instructiunile programului sunt instructiuni de salt si ca “*branch delay slot-ul*” aferent fiecarei instructiuni de salt este de 2 cicli.

66. Considerând un microprocesor virtual pe 16 biti, având 32 biti de adrese, setul de registri întregi pe 16 de biti ( $PC, R_x$  cu  $x \in [0,31]$ ). Se stie ca opcode-ul oricarei instructiuni e codificat pe 2 octeti, instructiunea e codificata pe 4 octeti si ca orice procesare (operatie interna) consuma un timp neglijabil. Se considera ca orice ciclu extern dureaza 5 tacte. Se considera ca un ciclu extern se încheie în momentul în care fie datele din memorie s-au încarcat cu succes în registrul destinatie, fie datele s-au scris cu succes în memorie. Sa se determine numarul impulsurilor de tact necesare aducerii si executiei instructiunii:

**LH  $R_2, (R_9 + \text{deplasament});$**  «Încarca un semicuvânt **halfword** (2 octeti) din memorie de la adresa data de  $(R_9 + \text{deplasament})$  în registrul  $R_2$ ».

67. a) Prezentați comparativ avantaje/dezavantaje ale memoriei cache asociative fata de memoria cache cu mapare directa.  
b) Descrieti succint metodele de predictie ale branch-urilor.

68. a) Ce reprezinta si ce rol are un simulator dedicat unei arhitecturi de calcul ? Metodologii de simulare cunoscute. Care dintre acestea este implementata de simulatoarele setului SimpleScalar 3.0 ?

- b) Avantajele setului SimpleScalar 3.0 ? Ce componente contine setul de instrumente SimpleScalar si ce rol au ? Descrieti succint fiecare din simulatoarele setului (parametrii, rezultate) ?
- c) Care sunt pasii succesivi aferenti drumului de la o arhitectura parametrizabila data, la instanta sa optimala, numita procesor. Rolul benchmark-ului în cadrul acestui proces. De ce credeti ca la fiecare aproximativ 3 ani este necesara scrierea de noi programe de test pentru arhitecturile de calcul ?
69. a) Care sunt parametrii de simulare, statisticile necesare, si pasii efectuati pentru extinderea mediului SimpleScalar 3.0 cu un modul de masurare a gradelor de localitate. Justificati necesitatea fiecarui parametru folosit.
- b) Definiti conceptul de localitate a valorii si structurile de date care sunt necesare în implementarea acestuia. Justificati fiecare câmp / structura / functie utilizata.
- c) Descrieti functionarea schemei de predictie de tip “*Last Value*”.
70. a) Ce exprima afirmatia “*cu o “adâncime” a istoriei de 1 (regasirea aceleiasi valori în resursa asignata ca si în cazul precedentului acces), programele de test exprima o localitate a valorii de peste 50% în timp ce extinzând verificarea în spatiul ultimelor 8 accese la memorie se obtine o localitate de peste 70%*”. Cum poate fi exploatata practic concluzia afirmatiei.
- b) În cadrul predictorului implementat de tip *Last Value* printre statistici se afla procentul de instructiuni Load clasificate predictibile de catre automatul de predictie si pentru care predictia a fost corecta si respectiv procentul de instructiuni Load clasificate nepredictibile de catre automat si pentru care predictia a fost gresita. În medie aceste statistici au fost de 90% respectiv 50%. Cum puteti interpreta aceste rezultate ? Ce ar trebui îmbunatatit în cadrul acestui predictor ?
71. a) Care sunt principalele avantaje ale tehnicii de reutilizare dinamica a instructiunilor ?
- b) De ce este dificila detectarea reutilizabilitatii unei functii C prin schemele de reutilizare dinamica a instructiunilor ?
- c) Care sunt principalele avantaje ale tehnicii de predictie dinamica a valorilor instructiunilor ?



72. a) Prezentați o clasificare a principalelor tipuri de predictoare de valori. Ce tipuri de secvențe de valori poate predictiona fiecare predictor? Ce tipuri de secvențe de valori sunt impredictibile ?  
 b) Descrieți succint modul de lucru al unui predictor *incremental*.  
 c) Prezentați pe scurt modul de lucru al unui predictor *contextual*.
73. a) Asupra carei limitări fundamentale acționează o arhitectură cu predictor dinamic de valori ?  
 b) Asemănări și deosebiri între tehnicile de reutilizare dinamică a instrucțiunilor respectiv de predicție a valorilor instrucțiunilor.
74. Se considera următoarea configurație a arhitecturii pipeline DLX (având 5 nivele pipeline IF, ID, EX, MEM, WB) și un program pentru respectiva arhitectură:

| Tip unitate de execuție  | Numar | Latenta (cicli) |
|--------------------------|-------|-----------------|
| Adunare/Scadere Flotant  | 1     | 2               |
| Înmulțire Flotant/Întreg | 1     | 5               |
| Împartire Flotant/Întreg | 1     | 12              |
| Adunare/Scadere Întreg   | 1     | 1               |

```

.data 0x2000
tab_A:
.word 1, 2, 5, 7

.data 0x3000
tab_B:
.word 2, 3, 4, 8

.text
loop:
lw    R6, 0x2000(R4);      încarca din memorie de la adresa
                           ;(R4+2000) și depune în R6

lw    R7, 0x3000(R4)
add   R3, R6, R7;          R3 ← R6 + R7
addd  f10, f12, f14;       f10 ← f12 + f14 (în dubla precizie)
movfp2i r8, f10;          R8 ← f10
slt    R2, R6, R7;         dacă R6 < R7 atunci R2 ← 1; altfel
                           ;R2 ← 0;

```

```

add      R4, R4, 4
bnez     R2, loop;      daca R2<>0 atunci se executa salt la
                        ;loop
add      R6, R7, R8
trap 0;                apel sistem de încheiere program

```

Se considera ca o instructiune este decodificata abia în momentul în care instructiunea anterioara de care ea depinde starteaza faza WB (scriere rezultat).

- Identificati hazardurile din program si clasificati-le dupa tipul lor: structurale, de date (RAW, WAW, WAR), control si datorate întreruperilor software ale programului.
- Care hazarduri pot fi eliminate de catre compilator ? Pentru fiecare dintre acestea, descrieti pe scurt cum s-ar putea realiza acest lucru. Respectiv hazardurile pot fi eliminate si prin metode hardware ?
- Determinati rata de procesare (instructiuni/ciclu), justificând pe diagrama ciclor procesorului. Refaceti diagrama ciclor procesorului considerând ca instructiunile 5 (*movfp2i*) si 6 (*slt ...*) sunt interschimbate.

**75.** Sa se optimizeze prin tehnica *trace scheduling* urmatoarea secventa de program. Prima coloana reprezinta secventa de program C (extras dintr-o bucla de interclasare a doi vectori) iar a doua coloana semnifica secventa de cod obiect obtinuta prin compilare. Executia secventei se face pe un procesor superscalar care decodifica 4 instructiuni simultan si detine 5 unitati de executie (ALU1, ALU2, SHF, LS, BRN). Doar instructiunile Load consuma 2 cicli de tact (Store - 1 tact), restul unitatilor executa operatiile aferente într-un timp unitar. Se considera ca saltul conditionat **se face preponderent** iar procesarea instructiunilor este **IN-ORDER**. Determinati cresterea de performanta obtinuta dupa reorganizarea prin aceasta tehnica (graful de control aferent secventei de optimizat si graful dependentelor pentru trace-ul considerat - *modelele de executie*). Identificati codurile compensatorii introduse. Ar avea vreun efect favorabil asupra ratei de procesare, migrarea instructiunii 13 în basic block-ul anterior ?

Se cunoaste ca registrul:

- ☐ R1 retine adresa elementului  $a[i]$  din tabloul a.
- ☐ R2 pastreaza adresa elementului  $b[j]$  din tabloul b.

- ☐ R3 retine adresa elementului  $c[k]$  din tabloul  $c$ .
- ☐ R4 pastreaza elementul curent  $a[i]$ , din tabloul  $a$ .
- ☐ R5 retine elementul curent  $b[j]$ , din tabloul  $b$ .

```

i=0; j=0; k=0;          1:      ADD  R1, R0, base_a
if (a[i]<b[j])           2:      ADD  R2, R0, base_b
{                        3:      ADD  R3, R0, base_c
    c[k]=a[i];          4:      LD    R4, (R1)
    i++;                5:      LD    R5, (R2)
}                        6:      SGT   R7, R4, R5
else                    7:      BEQZ  R7, B_MARE
{                        8:      ST    R5, (R3)
    c[k]=b[j];          9:      ADD  R2, R2, 4
    j++;               10:     JMP    CRESTE_K
}                        B_MARE:
k++;                   11:     ST    R4, (R3)
                       12:     ADD  R1, R1, 4
                       CRESTE_K:
                       13:     ADD  R3, R3, 4

```

**76.** Se considera un procesor scalar pipeline, având un "branch delay slot" (BDS) de 2 cicli si secventa de program de mai jos:

```

1.      A ← B
2.      B ← B - 1
3.      if (A=Q) then goto 7
        NOP
        NOP
4.      Q ← Q +1
5.      D ← E
6.      E ← F
7.      X ← Q
8.

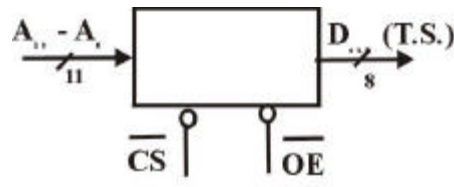
```

} BDS = 2 cicli

- a) Determinati rata de procesare (instr.utile/ciclu) a secventei de program în cazul în care branch-ul (3) nu se face, respectiv se face.
- b) Optimizati secventa de program prin mascarea (umplerea) BDS-ului cu instructiuni utile. Determinati performanta conform cerintelor punctului a) si în cazul programului optimizat astfel.

77. Un microcontroller 8051 are 4ko memorie EPROM interna, amplasati începând de la adresa 0000H. Sa se extinda memoria de program cu înca 12ko memorie EPROM externa, amplasati în continuarea spatiului de adresare aferent celei interne. Se vor folosi circuite EPROM de 2ko/chip ca în figura: Concret se cere:

- harta de memorie a EPROM-ului
- schema de interfata între 8051 si cele 6 circuite de EPROM extern.



$\overline{EA} \equiv$  legat la  $V_{CC}$  neaparat!!

**Obs:** Se poate adauga o extensie de 16ko RAM externa, pe acelasi spatiu de adresare cu EPROM-ul ?

$$\overline{PSEN} \leftrightarrow \begin{cases} \overline{RD} (\overline{OE}) \\ \overline{WR} (\overline{WE}) \end{cases} \text{ Harvard}$$

78. Optimizati urmatoarea bucla de program aplicând tehnica "*software pipelining*". Considerând un procesor superscalar cu o infinitate de unitati de executie, predictie perfecta a branch-urilor si un "*load delay slot*" de 1 ciclu, precizati în câti cicli se executa bucla initiala, respectiv cea optimizata.

```

Loop:  ld    R4, (R5)0
       xor   R7, R4, R9
       st    (R5)0, R7
       add   R5, R5, #4    // incrementare index
       sub   R6, R6, #1    // decrementare contor iteratii
       brnz  R6, Loop

```

79. Considerând un microprocesor virtual pe 16 biti, având 32 biti de adrese, setul de registri întregi pe 32 de biti ( $PC$ ,  $R_x$  cu  $x \in [0,31]$ ) si setul de registri flotanti (simpla precizie) pe 32 de biti ( $F_x$  cu  $x \in [0,31]$ ). Un registru flotant dubla precizie este realizat prin concatenarea registrilor  $F_x:F_{x+1}$ , unde  $x$  este numar par între 0 si 30. Se stie ca opcode-ul oricarei instructiuni e codificat pe 2 octeti, instructiunea e codificata pe 4 octeti

si ca orice procesare (operatie interna) consuma 2 tacte. Un ciclu de fetch opcode dureaza 6 tacte si orice alt ciclu extern dureaza 4 tacte. Se considera ca un ciclu extern se încheie în momentul în care fie datele din memorie s-au încarcat cu succes în registrul destinatie, fie datele s-au scris cu succes în memorie. Sa se determine numarul impulsurilor de tact necesare aducerii si executiei instructiunii

**LD F<sub>2</sub>, (R<sub>9</sub>+deplasament);** «Încarca o valoare **double (8 octeti)** din memorie de la adresa data de (R<sub>9</sub>+deplasament) în registrul dubla precizie F<sub>2</sub>».

80. a) Considerând un microprocesor scalar (*non-pipeline*) care aduce si executa secvential câte o instructiune (Aduce instructiunea 1, Executa instructiunea 1, Aduce instructiunea 2, Executa instructiunea 2). Se stie ca busul de date este de 16 biti, cel de adresa de 32 biti si setul de registri întregi pe 32 de biti (PC, R<sub>x</sub> cu x∈[0,31]). Opcode-ul oricarei instructiuni e codificat pe 2 octeti. Instructiunile cu referire la memorie sunt codificate pe 4 octeti iar cele aritmetice doar pe 2 octeti, si orice procesare (operatie interna) consuma 2 tacte. Un ciclu de fetch opcode dureaza 6 tacte si orice alt ciclu extern dureaza 4 tacte. Se considera ca un ciclu extern se încheie în momentul în care fie datele din memorie s-au încarcat cu succes în registrul destinatie, fie datele s-au scris cu succes în memorie. Sa se determine numarul impulsurilor de tact necesare aducerii si executiei secventei de instructiuni:

(1) **LH R<sub>7</sub>, (R<sub>9</sub>+deplasament);** încarca 2 octeti din memorie în registrul R<sub>7</sub> de la adresa data de R<sub>9</sub> + deplasament

(2) **MUL R<sub>5</sub>, R<sub>7</sub>, R<sub>3</sub>;**  $R_5 \leftarrow R_7 \times R_3$

b) Dar daca procesorul este pipeline [poate suprapune fazele de executie în timp - (Aduce instructiunea 1), (Executa instructiunea 1, Aduce instructiunea 2), (Executa instructiunea 2, Aduce instructiunea 3), etc]

81. a) De ce este necesara implementarea conceptului de **Data Write Buffer** într-un procesor superscalar ? Ce este un DWB ? Avantajele utilizarii sale.
- b) Care sunt avantajele implementarii tehnicii "**Load Multiplu**" (executia simultana a mai multor instructiuni Load care citesc din

acelasi bloc, fara sa existe un Store intercalat) asupra ratei de procesare ? Ce parametrii (resurse) și în ce masura influenteaza rata de procesare într-un procesor superscalar generic când este implementata aceasta tehnica ? (vezi **Figura 7.8: Schema bloc arhitecturii superscalare Harvard simulate** – L. Vintan, A. Florea – “Microarhitecturi de procesarea a informatiei”, Editura Tehnica, Bucuresti, 2000, pag. 199)

82. Considerând un microprocesor virtual pe 16 biti, având 32 biti de adrese, setul de registri întregi pe 32 de biti ( $PC, R_x$  cu  $x \in [0,31]$ ) și setul de registri flotanti (simpla precizie) pe 32 de biti ( $F_x$  cu  $x \in [0,31]$ ). Un registru flotant dubla precizie este realizat prin concatenarea registrilor  $F_x:F_{x+1}$ , unde  $x$  este numar par între 0 și 30. Se stie ca opcode-ul oricarei instructiuni e codificat pe 2 octeti, instructiunea e codificata pe 4 octeti și ca orice procesare (operatie interna) consuma 2 tacte. Un ciclu de fetch opcode dureaza 6 tacte și orice alt ciclu extern dureaza 4 tacte. Se considera ca un ciclu extern se încheie în momentul în care fie datele din memorie s-au încarcat cu succes în registrul destinatie, fie datele s-au scris cu succes în memorie. Sa se determine numarul impulsurilor de tact necesare aducerii și executiei instructiunii

**SD F<sub>6</sub>, deplasament(R<sub>4</sub>);** «Scrie o valoare **double (8 octeti)** - registrul dubla precizie  $F_6$  - în memorie la adresa data de  $(deplasament+R_4)$ ».

83. Se considera un procesor scalar pipeline, având un "branch delay slot" (BDS) de 2 cicli și secventa de program de mai jos. De asemenea, se cunoaste ca între doua instructiuni dependente RAW exista o întârziere de un ciclu de tact.

|    |                      |                 |
|----|----------------------|-----------------|
| 1. | $A \leftarrow B$     |                 |
| 2. | $C \leftarrow A - 1$ |                 |
| 3. | If (D=Q) then goto 7 |                 |
|    | NOP                  | } BDS = 2 cicli |
|    | NOP                  |                 |
| 4. | $Q \leftarrow Q + 1$ |                 |
| 5. | $D \leftarrow E$     |                 |
| 6. | $F \leftarrow D * A$ |                 |
| 7. | $X \leftarrow Q$     |                 |
| 8. |                      |                 |

- a) Determinati rata de procesare (instr.utile/ciclu) a secventei de program în cazul în care branch-ul (3) nu se face, respectiv se face.
- b) Optimizati secventa de program prin mascarea (umplerea) BDS-ului cu instructiuni utile. Determinati performanta conform cerintelor punctului a) si în cazul programului optimizat astfel.

**84.** Sa se optimizeze prin tehnica *trace scheduling* secventa urmatoare de program. Prima coloana reprezinta secventa de program C (extras dintr-o bucla de determinare a numarului de elemente pozitive / negative dintr-un vector) iar a doua coloana semnifica secventa de cod obiect obtinuta prin compilare. Executia secventei se face pe un procesor superscalar care decodifica 4 instructiuni simultan si detine 5 unitati de executie (ALU1, ALU2, SHF, LS, BRN). Doar instructiunile Load consuma 2 cicli de tact (Store - 1 tact), restul unitatilor executa operatiile aferente într-un timp unitar. Se considera ca saltul conditionat **se face preponderent** iar procesarea instructiunilor este **IN-ORDER**. Determinati cresterea de performanta obtinuta dupa reorganizarea prin aceasta tehnica (graful de control aferent secventei de optimizat si graful dependentelor pentru trace-ul considerat). Determinati rata de procesare si introduceti codurile compensatoare necesare.

|                                   |    |                                   |
|-----------------------------------|----|-----------------------------------|
| i=0;contor_plus=0;contor_minus=0; | 1: | ADD R1, R0, base_a                |
| if (a[i]<0)                       | 2: | LD R4, (R1)                       |
| contor_minus++;                   | 3: | SGE R7, R4, R0                    |
| else                              | 4: | BEQZ R7, Minus                    |
| contor_plus++;                    | 5: | ADD contor_plus, contor_plus, 1   |
| i++;                              | 6: | JMP Creste_I                      |
|                                   |    | Minus:                            |
|                                   | 7: | ADD contor_minus, contor_minus, 1 |
|                                   |    | Creste_I:                         |
|                                   | 8: | ADD R1, R1, 4                     |

Se cunoaste ca registrul:

- ☐ R1 retine adresa elementului a[i] din tabloul a.
- ☐ R4 retine elementul curent a[i], din tabloul a.

**85.** Scrieti o secventa de program asamblare RISC care sa reprezinte translatarea corecta a programului scris în limbaj C ? Initial, registrii R<sub>i</sub>, R<sub>k</sub>, R<sub>l</sub>, R<sub>j</sub>, R<sub>m</sub> contin respectiv variabilele i, k, l, j si m.

$$K = A[2*I+3]-6;$$

$$L = C[J-9] \text{ AND } K;$$

$$B[K+3] = 4 * L - 1;$$

Se considera programul executat pe un procesor scalar pipeline RISC cu 5 nivele (IF, ID, ALU, MEM, WB) si operanzii instructiunilor sunt necesari la finele fazei ID, iar rezultatele sunt disponibile în setul de registri generali la finele fazei WB. Se cere:

- Reprezentati graful dependentelor de date (dependentele de tip RAW, WAR si WAW).
- În câte impulsuri de tact se executa secventa de program asamblare ?
- Reorganizati aceasta secventa în vederea minimizarii timpului de executie (se considera ca procesorul detine o infinitate de registri generali).
- Aplicând tehnica de *forwarding*, în câte impulsuri de tact se executa secventa reorganizata ?

86. Se considera o arhitectura superscalara caracterizata de urmatoorii parametri (vezi **Figura 7.8: Schema bloc arhitecturii superscalare Harvard simulate** – L. Vintan, A. Florea – “Microarhitecturi de procesarea a informatiei”, Editura Tehnica, Bucuresti, 2000, pag. 199):

**FR** = 4 instr. / ciclu; – nr. instructiuni citite simultan din cache-ul de instructiuni sau memoria de instructiuni în caz de miss în cache.

**IRmax** = 2 (respectiv 4) instr. / ciclu; – nr. maxim de instructiuni independente lansate simultan în executie.

**N\_PEN** = 10 impulsuri de tact; – nr. impulsuri de tact penalizare necesari accesului la memoria de instructiuni/date în caz de miss în cache-ul corespunzator.

**Latenta** = 2 impulsuri de tact; – nr. impulsuri de tact necesari executiei pentru orice tip de instructiune mai putin cele de inmultire si impartire, lansata din buffer-ul de prefetch (unitati de executie nepipeline-izate).

**Latenta\_MULT** = 6 impulsuri de tact; – nr. impulsuri de tact necesari executiei pentru instructiunile de inmultire lansate din buffer-ul de prefetch.



**Latenta\_DIV** = 14 impulsuri de tact; – nr. impulsuri de tact necesari executiei pentru instructiunile de impartire lansate din buffer-ul de prefetch.

**IBS** = 8 locatii (instructiuni); – dimensiunea buffer-ului de prefetch.

**RmissIC** = 50%; – rata de miss în cache-ul de instructiuni (se presupune ca primele accese in cache sunt cu miss).

Consideram urmatoarea secventa succesiva de 24 instructiuni, caracterizata de hazarduri RAW aferente, executata pe arhitectura data.

$i1 - i2 - i3 - RAW - i4 - i5 - i6 - RAW - i7 - i8 - i9 - i10 - i11 - RAW - i12 - i13 - i14 - i15 - i16 - i17 - RAW - i18 - i19 - i20 - i21 - RAW - i22 - i23 - i24$ . (în continuare “nu mai sunt instructiuni de executat”)

De asemenea, se stie ca instructiunea  $i2$  este instructiune de împartire,  $i8$  si  $i20$  sunt instructiuni de înmultire iar instructiunile  $i3$ ,  $i7$  si  $i21$  sunt cu referire la memorie si determina accese cu miss la cache-ul de date.

**Obs.** În cadrul unui ciclu de executie se realizeaza urmatoarele: din partea inferioara a buffer-ului de prefetch sunt lansate maxim  $IR_{max}$  instructiuni independente, iar simultan în partea superioara a buffer-ului sunt aduse, daca mai e spatiu disponibil,  $FR$  instructiuni din cache-ul sau memoria de instructiuni.

Determinati cresterea de performanta (studiu asupra ratei medii de procesare) prin varierea parametrului  $IR_{max}$  de la 2 la 4 instructiuni / ciclu. Prezentați în fiecare ciclu de executie continutul buffer-ului de prefetch.

**87.** Se considera o arhitectura superscalara caracterizata de urmatorii parametri (vezi **Figura 7.8: Schema bloc arhitecturii superscalare Harvard simulate** – L. Vintan, A. Florea – “Microarhitecturi de procesarea a informatiei”, Editura Tehnica, Bucuresti, 2000, pag. 199):

**FR** = 4 (respectiv 8) instr. / ciclu; – nr. instructiuni citite simultan din cache-ul de instructiuni sau memoria de instructiuni în caz de miss în cache.

**IRmax** = 4 instr. / ciclu; – nr. maxim de instructiuni independente lansate simultan în executie.

**N\_PEN** = 10 impulsuri de tact; – nr. impulsuri de tact penalizare necesari accesului la memoria de instructiuni/date în caz de miss în cache-ul corespunzator.

**Latenta** = 2 impulsuri de tact; – nr. impulsuri de tact necesari executiei pentru orice tip de instructiune, lansata din buffer-ul de prefetch (unitati de executie nepipeline-izate).

**IBS** = 8 (respectiv 16 locatii (instructiuni)); – dimensiunea buffer-ului de prefetch.

**RmissIC** = 50%; – rata de miss în cache-ul de instructiuni (se presupune ca primele accese în cache sunt cu miss).

Pe arhitectura data se proceseaza urmatoarea secventa succesiva de 32 instructiuni, caracterizata de hazarduri RAW aferente.

*i1 – i2 – RAW – i3 – i4 – i5 – i6 – RAW – i7 – i8 – i9 – i10 – i11 – i12 – RAW – i13 – i14 – i15 – i16 – i17 – RAW – i18 – i19 – i20 – i21 – i22 – RAW – i23 – i24 – i25 – i26 – i27 – RAW – i28 – i29 – i30 – i31 – RAW – i32.*

De asemenea, se stie ca instructiunile *i10* si *i17* sunt instructiuni cu referire la memorie si determina accese cu miss la cache-ul de date.

**Obs.** În cadrul unui ciclu de executie se realizeaza urmatoarele: din partea inferioara a buffer-ului de prefetch sunt lansate maxim  $IR_{max}$  instructiuni independente, iar simultan în partea superioara a buffer-ului sunt aduse, daca mai e spatiu disponibil,  $FR$  instructiuni din cache-ul sau memoria de instructiuni.

Sa se determine care configuratie arhitecturala din urmatoarele 4 cazuri: (**A1.**  $FR=4$ ;  $IBS=8$ , **A2.**  $FR=4$ ;  $IBS=16$ , **B1.**  $FR=8$ ;  $IBS=8$  si **B2.**  $FR=8$ ;  $IBS=16$ ) impune performanta (rata medie de procesare) optima. Prezantati în fiecare ciclu de executie continutul bufferului de prefetch. În ce consta limitarea performantei în acest caz ?

- 88.** a) Ce deficiente functionale ar avea o schema de predictie de tip GAg (**Figura 2.28 - "Arhitecturi de procesoare cu paralelism la nivelul instructiunilor"**, L. Vintan, Editura Academiei, 2000). Cu  $i=0$  ( $PC_{low}$ ) si fara câmpul TAG în tabela de predictie ?  
b) Considerând o schema GAg cu  $k=3$ , câte automate de predictie distincte ar putea accesa un branch ? Ar putea accesa chiar mai putine ?

- c) Ce particularizare ar trebui facuta într-o schema GAg pentru a se transforma într-una de tip BTB?
- d) Care dintre schemele de predicție GAg respectiv PAg (**figura 2.29 - "Arhitecturi de procesoare cu paralelism la nivelul instructiunilor"**, L. Vintan, Editura Academiei, 2000) este mai rapida ? Justificare.
- e) De ce schema PAp ar putea fi mai performanta (**figura 2.30 - "Arhitecturi de procesoare cu paralelism la nivelul instructiunilor"**, L. Vintan, Editura Academiei, 2000) decât schema PAg ?
- f) Considerând o schema PAp cu  $i+k=3$  intrari în tabela de istorie a branch-urilor (PBHT) si  $HR_{local}=2$ , câte automate de predicție distincte ar putea accesa un branch ?

**89.** Construiti un automat cu 4 stari (într-o stare el poate predictiona DA sau NU) care sa predictioneze 100% corect un branch având urmatorul comportamentS

- a) NT, NT, T, T, NT, NT, T, T, ...
- b) T, T, T, NT, T, T, T, NT, ...

Câte astfel de automate distinct îndeplinesc conditiile impuse anterior ?

**Obs:** Se va considera ca procesele de predicție starteara dupa initializarea automatului.

**90.** a) Se considera schema de predicție de tip GAg (fig. 2.28 - "Arhitecturi de procesoare cu paralelism la nivelul instructiunilor", L. Vintan, Editura Academiei, 2000), cu mapare directa si cu un singur bit de predicție (PRED). Se stie ca  $i=11$  (PClow),  $k=1$  (informatia de corelatie). Se mai stie ca branch-ul de la  $PC=2DF4h$ , codificat pe 4 octeti, are urmatorul comportament: daca saltul anterior nu s-a facut atunci el se va face la adresa  $4FC0h$ , iar daca saltul anterior s-a facut atunci el nu se face. Sa se precizeze, daca este posibil, continutul câmpurilor TAG, TARGET, PRED aferente urmatoarelor adrese ale tablei de predicție (PHT): BE6h, BE7h, BE8h, BE9h.

b) Care dintre cele 3 branch-uri de mai jos considerati ca este dificil de predictionat ? Comentati.

|           | HRg | T    | NT   |
|-----------|-----|------|------|
| <b>B1</b> | 10  | 3    | 1047 |
| <b>B2</b> | 10  | 4900 | 5044 |
| <b>B3</b> | 11  | 1    | 1    |

91. Se considera urmatoarea secventa de 14 instructiuni de salt din cadrul fisierului trace **sort** (care implementeaza algoritmul “*quicksort*”), component al suitei de benchmark-uri *Stanford*.

|     |          |
|-----|----------|
| 1.  | BS 27 9  |
| 2.  | BM 17 28 |
| 3.  | BT 35 38 |
| 4.  | NT 40 41 |
| 5.  | BT 45 27 |
| 6.  | BS 27 9  |
| 7.  | BM 17 28 |
| 8.  | BT 35 38 |
| 9.  | BT 40 42 |
| 10. | BT 45 27 |
| 11. | BS 27 9  |
| 12. | BM 17 28 |
| 13. | BT 35 38 |
| 14. | BT 40 42 |

Se stie ca instructiunile de salt respective sunt memorate într-o tabela de salturi de tip BTB cu 8 locatii, vida (initial) – câmpurile tabelii sunt initializate cu 0. Se reaminteste ca instructiunile de salt se introduc în tabela cu prima ocazie în care se face (**taken**) iar cele (**not taken**) nu se adauga în tabela întrucât nu îmbunatatesc acuratetea de predictie. Formatul instructiunilor de salt este urmatorul:

|    |     |    |
|----|-----|----|
| BT | 12  | 30 |
| BS | 32  | 98 |
| BM | 100 | 33 |
| NT | 36  | 37 |

Reprezentarea salturilor se face sub forma unor triplete <**TipBr** **AdrCrt** **AdrDest**>, unde **TipBr** se prezinta sub forma unei codificari pe doua caractere, primul dintre ele indica daca saltul se face (‘B’ – saltul se face, ‘N’ – saltul nu se face), iar al doilea caracter indica tipul saltului: ‘T’ sau ‘F’ – salturi conditionate, ‘S’ – apeluri de tip Call, ‘M’ – apeluri de tip Return, ‘R’ – salturi neconditionate. Alegerea acestei codificari a fost inspirata de mnemonicile întâlnite în sursa în limbaj de asamblare (BT, BF – salt conditionat, BSR – instructiune de tip Call).

Se cere sa se determine acuratetea de predictie în doua cazuri distincte:

- BTB – mapata direct si BTB – complet asociativa
- Sa se rezolve problema similara având un BTB de 4 locatii.

## 12. INDICATII DE SOLUTIONARE

---

1.

A. i1: LOAD R<sub>1</sub>, 9(R<sub>5</sub>)  
i2: ADD R<sub>6</sub>, R<sub>1</sub>, R<sub>3</sub>

a)

|     |    |    |     |     |     |     |     |    |  |
|-----|----|----|-----|-----|-----|-----|-----|----|--|
| i1: | IF | ID | ALU | MEM | WB  |     |     |    |  |
|     |    | IF | ID  | ALU | MEM | WB  |     |    |  |
|     |    |    | IF  | ID  | ALU | MEM | WB  |    |  |
| i2: |    |    |     | IF  | ID  | ALU | MEM | WB |  |

**Delay = 2 cicli.**

b) aplicând *forwarding*

|     |    |    |     |     |     |     |    |  |
|-----|----|----|-----|-----|-----|-----|----|--|
| i1: | IF | ID | ALU | MEM | WB  |     |    |  |
|     |    | IF | ID  | ALU | MEM | WB  |    |  |
| i2: |    |    | IF  | ID  | ALU | MEM | WB |  |

**Delay = 1 ciclu.**

B. i1: ADD R<sub>1</sub>, R<sub>6</sub>, R<sub>7</sub>  
i2: LOAD R<sub>5</sub>, 9(R<sub>1</sub>)

a)

|     |    |    |     |     |     |     |     |    |
|-----|----|----|-----|-----|-----|-----|-----|----|
| i1: | IF | ID | ALU | MEM | WB  |     |     |    |
|     |    | IF | ID  | ALU | MEM | WB  |     |    |
|     |    |    | IF  | ID  | ALU | MEM | WB  |    |
| i2: |    |    |     | IF  | ID  | ALU | MEM | WB |

**Delay = 2 cicli.**

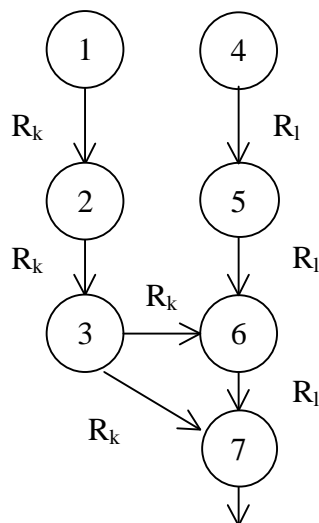
b) aplicând forwarding

|     |    |    |     |     |     |    |  |
|-----|----|----|-----|-----|-----|----|--|
| i1: | IF | ID | ALU | MEM | WB  |    |  |
| i2: |    | IF | ID  | ALU | MEM | WB |  |

**Delay = 0 cicli.** (necesar  $R_1$  la începutul fazei ALU a i2 pentru calcularea adresei de memorie de unde va fi citita data; dacă modul de adresare ar fi indirect registru,  $R_1$  ar fi necesar doar la începutul fazei MEM, deci între cele doua instructiuni dependente RAW mai poate exista încă o a treia fara a implica penalizari)

- 2.
- i1: SUB  $R_k, R_i, \#4$
  - i2: LOAD  $R_k, (R_k+0)$
  - i3: ADD  $R_k, R_k, \#12$
  - i4: ADD  $R_l, \#5, R_j$
  - i5: LOAD  $R_l, (R_l+0)$
  - i6: XOR  $R_l, R_l, R_k$
  - i7: AND  $R_m, R_k, R_l$

a)



b).

|     |    |    |           |           |           |    |
|-----|----|----|-----------|-----------|-----------|----|
| I1: | IF | ID | ALU / MEM | WB        |           |    |
|     |    | IF | ID        | ALU / MEM | WB        |    |
| I2: |    |    | IF        | ID        | ALU / MEM | WB |

**Delay = 1 ciclu.**

1 – nop – 2 – nop – 3 – 4 – nop – 5 – nop – 6 – nop – 7  $\Rightarrow T_{ex} = 12$  cicli executie.

c) 1 – 4 – 2 – 5 – 3 – nop – 6 – nop – 7  $\Rightarrow T_{ex} = 9$  cicli executie.

d) Aplicând forwarding-ul rezulta **Delay = 0 cicli**  $\Rightarrow$  Secventa se executa:

1 – 4 – 2 – 5 – 3 – 6 – 7  $\Rightarrow T_{ex} = 7$  cicli executie.

3.

a) **IRmax** = 2 instr. / ciclu.

**FR** = 4 instr. / ciclu  $\Rightarrow$  fiind 20 instructiuni de executat  $\Rightarrow$  vor fi 5 accese (citiri) la IC.

**RmissIC** = 40%  $\Rightarrow$  2 accese la IC vor fi cu miss. Consideram primele 2 accese la cache.

Configuratia buffer-ului de prefetch în fiecare ciclu de executie:

| C1 | C2 | C3 | C4  | C5  | C6  | C7  | C8  | C9  | C10 | C11 | C12 | C13 |
|----|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|    |    |    |     |     | i16 |     |     |     |     |     |     |     |
|    |    |    | i12 |     | i15 |     |     | i20 |     |     |     |     |
|    | i8 |    | i11 | i12 | i14 | i16 |     | i19 |     |     |     |     |
|    | i7 | i8 | i10 | i11 | i13 | i15 | i16 | i18 | i20 |     |     |     |
| i4 | i6 | i7 | i9  | i10 | i12 | i14 | i15 | i17 | i19 |     |     |     |
| i3 | i5 | i6 | i8  | i9  | i11 | i13 | i14 | i16 | i18 | i20 |     |     |
| i2 | i4 | i5 | i7  | i8  | i10 | i12 | i13 | i15 | i17 | i19 |     |     |
| i1 | i3 | i4 | i6  | i7  | i9  | i11 | i12 | i14 | i16 | i18 | i20 |     |

C1 – se executa în 10 cicli (primul miss în IC – aducere instructiuni din memoria principala); în buffer nefiind nici o instructiune nu se executa nimic.

C2 – se executa în 10 cicli (al doilea miss în IC – aducere instructiuni din memoria principala); în buffer sunt instructiuni, se executa i1 si i2.

- C3 – se executa în 2 cicli (nu se face fetch instructiune nefiind spatiu disponibil în buffer); în buffer sunt instructiuni, se executa i3 datorita dependentelor RAW dintre i3 si i4.
- C4 – se executa în 2 cicli (hit în IC – aducere instructiuni din cache); se executa i4 si i5.
- C5 – se executa în 2 cicli (nu se face fetch instructiune nefiind spatiu disponibil în buffer); se executa i6 datorita dependentelor RAW dintre i6 si i7.
- C6 – se executa în 2 cicli (hit în IC – aducere instructiuni din cache); se executa i7 si i8.
- C7 – se executa în 2 cicli (nu se face fetch instructiune); se executa i9 si i10.
- C8 – se executa în 2 cicli (nu se face fetch instructiune); se executa i11 datorita dependentelor RAW dintre i11 si i12.
- C9 – se executa în 2 cicli (hit în IC – aducere instructiuni din cache); se executa i12 si i13.
- C10 – se executa în 2 cicli (nu se mai face fetch instructiune – s-au adus toate instructiunile); se executa i14 si i15.
- C11 – se executa în 2 cicli; se executa i16 si i17.
- C12 – se executa în 2 cicli; se executa i18 si i19.
- C13 – se executa în 2 cicli; se executa i20 – s-a golit buffer-ul nu mai sunt instructiuni de executat.

$$T_{ex} = 10 (C1) + 10 (C2) + 2 (C3) + 2 (C4) + 2 (C5) + 2 (C6) + 2 (C7) + 2 (C8) + 2 (C9) + 2 (C10) + 2 (C11) + 2 (C12) + 2 (C13) = 42 \text{ impulsuri de tact}$$

$$IR = 20 \text{ instr.} / 42 \text{ imp.} = 0,476 \text{ instr.} / \text{tact}$$

b) **IR<sub>max</sub>** = 4 instr. / ciclu.

Configuratia buffer-ului de prefetch în fiecare ciclu de executie:

| C1 | C2 | C3  | C4  | C5  | C6  | C7  | C8 |
|----|----|-----|-----|-----|-----|-----|----|
|    |    |     |     |     |     |     |    |
|    |    | i12 | i16 |     |     |     |    |
|    | i8 | i11 | i15 | i16 | i20 |     |    |
| i4 | i7 | i10 | i14 | i15 | i19 |     |    |
| i3 | i6 | i9  | i13 | i14 | i18 | i20 |    |
| i2 | i5 | i8  | i12 | i13 | i17 | i19 |    |
| i1 | i4 | i7  | i11 | i12 | i16 | i18 |    |



$$T_{\text{ex}} = 10 (C1) + 10 (C2) + 2 (C3) + 2 (C4) + 2 (C5) + 2 (C6) + 2 (C7) + 2$$

$$(C8) = 32 \text{ impulsuri de tact}$$

$$IR = 20 \text{ instr.} / 32 \text{ imp.} = 0,625 \text{ instr.} / \text{tact}$$

$$\text{Cresterea performantei} = (IR(4) - IR(2)) / IR(2) = 31\%$$

## 4.

```

a) MOV R10, R6
   ST   (R9), R6

b) ADD R3, R5, #4 /* Calculeaza adresa pentru ST */
   ADD R4, R6, #8 /* Calculeaza adresa pentru LD */
   LD   R9, 8(R6) /* Daca adresele difera efectuam anticipat LD */
   BEQ  R3, R4, et /* Daca adresele sunt egale, salt pentru a încarca
                   în R9 valoarea corecta (R8)*/

J      end
et:    MOV R9, R8
end:    ST 4(R5), R8 /* Daca adresele coincid sau nu se memoreaza R8
                   la adresa ceruta */

```

## 5.

|                                                           |                                                              |
|-----------------------------------------------------------|--------------------------------------------------------------|
| a)    MOV   R6, R7<br>ADD   R3, R6, R5                    | Ra)    MOV   R6, R7<br>ADD   R3, R7, R5                      |
| b)    MOV   R6, #4<br>ADD   R7, R10, R6<br>LD    R9, (R7) | Rb)    MOV   R6, #4<br>ADD   R7, R10, #4<br>LD    R9, 4(R10) |
| c)    MOV   R6, #0<br>ST    9(R1), R6                     | Rc)    MOV   R6, #0<br>ST    9(R1), R0                       |
| d)    MOV   R5, #4<br>BNE   R5, R3, Label                 | Rd)    MOV   R5, #4<br>BNE   R5, #4, Label                   |
| e)    ADD   R3, R4, R5<br>MOV   R6, R3                    | Re)    ADD   R3, R4, R5<br>ADD   R6, R4, R5                  |

## 6.

a)  $IR_{max} = 4$  instr. / ciclu.

$FR = 4$  instr. / ciclu  $\Rightarrow$  fiind 32 instrucțiuni de executat  $\Rightarrow$  vor fi 8 accese (citiri) la IC.

$R_{missIC_1} = 50\% \Rightarrow 4$  accese la IC vor fi cu miss. Considerăm primele 4 accese la cache.

Configurația buffer-ului de prefetch în fiecare ciclu de execuție:

| C1 (F1) | C2 (F2) | C3 (F3) | C4 (F4) | C5  | C6  | C7  | C8  | C9  | C10 | C11 | C12 | C13 |
|---------|---------|---------|---------|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|         |         |         |         |     |     |     |     |     |     |     |     |     |
|         |         |         |         |     |     | i24 | i28 |     |     |     |     |     |
|         |         | i12     | i16     |     |     | i23 | i27 | i28 | i32 |     |     |     |
|         | i8      | i11     | i15     | i16 | i20 | i22 | i26 | i27 | i31 | i32 |     |     |
| i4      | i7      | i10     | i14     | i15 | i19 | i21 | i25 | i26 | i30 | i31 |     |     |
| i3      | i6      | i9      | i13     | i14 | i18 | i20 | i24 | i25 | i29 | i30 |     |     |
| i2      | i5      | i8      | i12     | i13 | i17 | i19 | i23 | i24 | i28 | i29 |     |     |
| i1      | i4      | i7      | i11     | i12 | i16 | i18 | i22 | i23 | i27 | i28 | i32 |     |

C1 – se execută în 10 cicli (primul miss în IC – aducere instrucțiuni din memoria principală); în buffer nefiind nici o instrucțiune nu se execută nimic.

C2 – se execută în 10 cicli (al doilea miss în IC – aducere instrucțiuni din memoria principală); în buffer sunt instrucțiuni, se execută i1, i2 și i3.

C3 – se execută în 10 cicli (al treilea miss în IC – aducere instrucțiuni din memoria principală); în buffer sunt instrucțiuni, se execută i4, i5 și i6 datorită dependențelor RAW dintre i6 și i7.

C4 – se execută în 10 cicli (al patrulea miss în IC – aducere instrucțiuni din memoria principală); se execută i7, i8, i9 și i10.

C5 – se execută în 2 cicli (nu se face fetch instrucțiune nefiind spațiu disponibil în buffer); se execută i11 datorită dependențelor RAW dintre i11 și i12.

C6 – se execută în 2 cicli (hit în IC – aducere instrucțiuni din cache); se execută i12, i13, i14 și i15.

C7 – se execută în 2 cicli (hit în IC); se execută i16 și i17 datorită dependențelor RAW dintre i17 și i18.

C8 – se execută în 2 cicli (hit în IC); se execută i18, i19, i20 și i21.

C9 – se execută în 2 cicli (nu se face fetch instrucțiune nefiind spațiu disponibil în buffer); se execută i22 datorită dependențelor RAW dintre i22 și i23.

C10 – se execută în 2 cicli (hit în IC); se execută i23, i24, i25 și i26.

C11 – se executa în 2 cicli; (nu se mai face fetch instructiune – s-au adus toate instructiunile); se executa i27 datorita dependentelor RAW dintre i27 si i28.

C12 – se executa în 2 cicli; se executa i28, i29, i30 si i31.

C13 – se executa în 2 cicli; se executa i32 – s-a golit buffer-ul nu mai sunt instructiuni de executat.

$$T_{ex} = 10 (C1) + 10 (C2) + 10 (C3) + 10 (C4) + 2 (C5) + 2 (C6) + 2 (C7) + 2 (C8) + 2 (C9) + 2 (C10) + 2 (C11) + 2 (C12) + 2 (C13) = 58 \text{ impulsuri de tact}$$

$$IR = 32 \text{ instr.} / 58 \text{ imp.} = 0,55 \text{ instr.} / \text{tact}$$

b) **FR** = 8 instr. / ciclu  $\Rightarrow$  fiind 32 instructiuni de executat  $\Rightarrow$  vor fi 4 accese (citiri) la IC.

**RmissIC<sub>2</sub>** = 50%  $\times$  **RmissIC<sub>1</sub>**  $\Rightarrow$  **RmissIC<sub>2</sub>** = 25%  $\Rightarrow$  1 acces la IC va fi cu miss. Consideram primul acces la cache.

Configuratia buffer-ului de prefetch în fiecare ciclu de executie:

| C1 (F1) | C2 | C3 | C4  | C5  | C6  | C7  | C8  | C9  | C10 | C11 | C12 | C13 | C14 |
|---------|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| i8      |    |    | i16 |     |     | i24 |     |     |     | i32 |     |     |     |
| i7      |    |    | i15 |     |     | i23 | i24 |     |     | i31 |     |     |     |
| i6      |    |    | i14 |     |     | i22 | i23 |     |     | i30 |     |     |     |
| i5      | i8 |    | i13 | i16 |     | i21 | i22 |     |     | i29 | i32 |     |     |
| i4      | i7 |    | i12 | i15 |     | i20 | i21 |     |     | i28 | i31 |     |     |
| i3      | i6 |    | i11 | i14 |     | i19 | i20 | i24 |     | i27 | i30 |     |     |
| i2      | i5 | i8 | i10 | i13 |     | i18 | i19 | i23 | i24 | i26 | i29 |     |     |
| i1      | i4 | i7 | i9  | i12 | i16 | i17 | i18 | i22 | i23 | i25 | i28 | i32 |     |

$$T_{ex} = 10 (C1) + 2 (C2) + 2 (C3) + 2 (C4) + 2 (C5) + 2 (C6) + 2 (C7) + 2 (C8) + 2 (C9) + 2 (C10) + 2 (C11) + 2 (C12) + 2 (C13) + 2 (C14) = 36 \text{ impulsuri de tact}$$

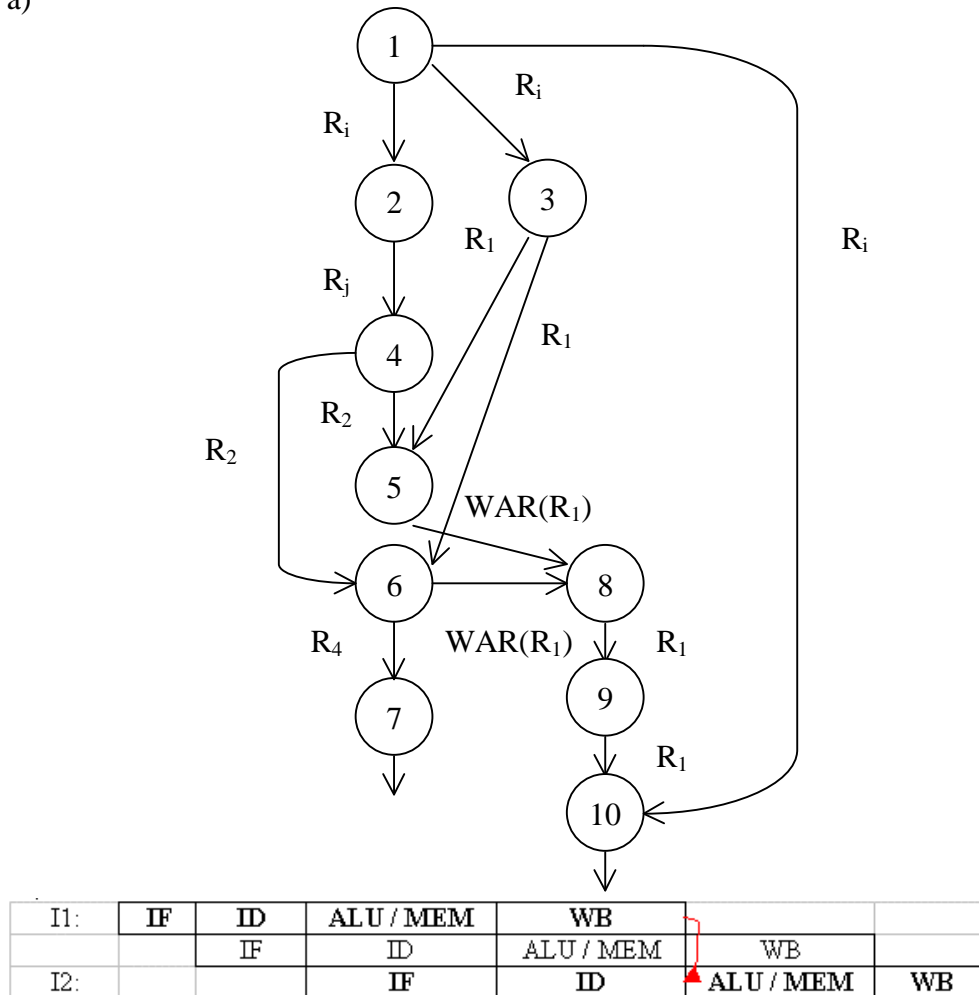
$$IR = 32 \text{ instr.} / 36 \text{ imp.} = 0,89 \text{ instr.} / \text{tact}$$

Cresterea performantei =  $(IR(FR=8) - IR(FR=4)) / IR(FR=4) = 61,8\%$  (substantiala).

Limitarea performantei consta în capacitatea buffer-ului de prefetch egala cu rata de fetch (**IBS=FR**).

7.

a)



**Delay = 1 ciclu.**

1 – nop – 2 – 3 – 4 – nop – 5 – 6 – nop – 7 – 8 – nop – 9 – nop – 10  $\Rightarrow T_{ex} = 15$  cicli executie.

b) După aplicarea tehnicii de *renaming* la registrul  $R_1$  în instrucțiunea i8 obținem:

```

i8:  ADD  R1', R13, R14
i9:  DIV  R1', R1', #2
i10: STORE (Ri), R1'

```

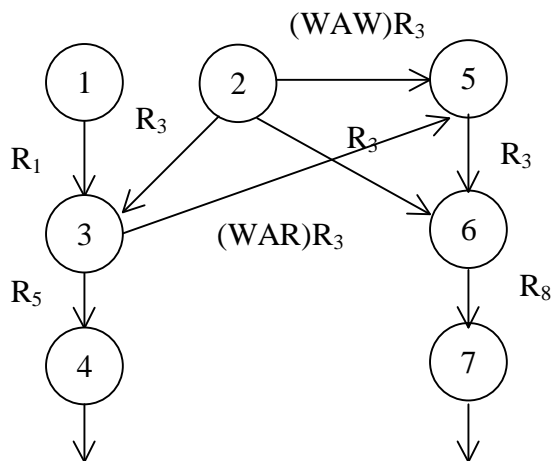
$1 - 8 - 2 - 3 - 4 - 9 - 5 - 6 - 10 - 7 \Rightarrow T_{\text{ex}} = 10$  cicli executie.

c) Prin înlocuirea registrilor  $R_{13}$  cu  $R_3$  si  $R_{14}$  cu  $R_4$ , secventa realizeaza determinarea maximului dintre elementele  $x[i]$  si  $x[i+4]$  ale unui sir de date stocat în memorie. Se obtine formula:

$$\max(x[i], x[i+4]) = \frac{(x[i+4] + x[i]) + |x[i+4] - x[i]|}{2}$$

8.

a)



Secventa se executa astfel:

$1 - 2 - \text{nop} - 3 - \text{nop} - 4 - 5 - \text{nop} - 6 - \text{nop} - 7 \Rightarrow T_{\text{ex}} = 11$  cicli executie.

b) Se aplica tehnica de *renaming* pe ramura  $5 - 6 - 7 \Rightarrow R_3 = R_3'$ .

Astfel având 3 unitati ALU la dispozitie executia secventei ar fi urmatoarea:

| Nr. Tact | Instructiuni executate în paralel |
|----------|-----------------------------------|
|          | 1 - 2 - 5                         |
|          | 3 - 6                             |
|          | 4 - 7                             |

$T_{\text{ex}} = 3$  cicli executie.

## 9.

- a) Da. La startarea sistemului se încarcă programul *încarcator* care verifică perifericele, citește informații din EPROM. E posibil să existe hit la citire din cache care nefiind inițializat conține “*proști*” și dacă nu ar exista bitul de validare (V) sistemul ar prelua valoarea din cache eronată.
- b) O excepție *Page Fault* implică întotdeauna o excepție TLB miss. Reciproc, nu e obligatoriu.

10. a) Rezultate statistice bazate pe simulări laborioase pe benchmark-uri reprezentative (SPEC '95, '00) arată că un basic-block conține doar 4-5 instrucțiuni în medie, ceea ce înseamnă că rata de fetch (FR) a instrucțiunilor e limitată la cca. 5, aducerea simultană a mai multor instrucțiuni fiind inutilă (fetch bottleneck). Această limitare fundamentală are consecințe defavorabile și asupra ratei medii de execuție a instrucțiunilor (IR - Issue Rate) întrucât  $IR \leq FR$ . Cercetările actuale insistă pe **îmbunătățirea mecanismelor de aducere a instrucțiunilor** prin următoarele tehnici:

- *predicția simultană a mai multor ramificații / tact* rezultând deci rate IR sporite.
- *posibilitatea accesării și aducerii simultane a mai multor basic-block-uri din cache, chiar dacă acestea sunt nealinierte, prin utilizarea unor cache-uri multiport.*
- *pastrarea unei latente reduse a procesului de aducere a instrucțiunilor, în contradicție cu cele 2 cerințe anterioare.*

Alți factori care determină limitarea ratei de fetch a instrucțiunilor sunt: lățimea de bandă limitată a interfeței procesor - cache, missurile în cache, predicțiile eronate ale ramificațiilor.

O soluție interesantă față de limitările mai sus menționate, o constituie **trace-procesorul**, adică un procesor superscalar având o **memorie trace-cache** (TC). Ca și cache-urile de instrucțiuni (IC), TC este accesată cu adresa de început a noului bloc de instrucțiuni ce trebuie executat, în paralel cu IC. În caz de miss în TC, instrucțiunea va fi adusă din IC sau - în caz de miss și aici - din memoria principală. Spre deosebire însă de IC, TC memorează instrucțiuni contigue din punct de vedere al secvenței lor de execuție, în locații contigue de memorie. O linie din TC memorează un segment de instrucțiuni executate dinamic și secvențial în program (trace-segment). Evident, un trace poate conține mai multe basic-block-uri (unități secvențiale de program). Asadar, o linie TC poate conține N instrucțiuni sau M basic-block-uri,  $N > M$ , înscrise pe parcursul execuției lor.

b) Hazardurile constituie acele situatii care pot sa apara în procesarea pipeline si care pot determina blocarea (stagnarea) procesarii, având deci o influenta negativa asupra ratei de executie a instructiunilor. Conform unei clasificari consacrate hazardurile sunt de 3 categorii : hazarduri structurale, de date (RAW, WAR si WAW) si de ramificatie.

Considerând instructiunile **i** si **j** succesive, **hazardul RAW** (Read After Write) apare atunci când instructiunea **j** încearca sa citeasca o sursa înainte ca instructiunea **i** sa scrie în aceasta. Pentru o procesare corecta, instructiunea **j** trebuie stagnata cu un ciclu masina. Solutia software consta în inserarea unei instructiuni NOP între **i** si **j** (umplerea "delay slot"-ului) sau a unei alte instructiuni utile, independenta de instructiunea **i**. O alta modalitate de rezolvare consta în stagnarea hardware a instructiunii **i**, stagnare determinata de detectia hazardului RAW de catre unitatea de control. Realizarea întârzierii hardware se poate baza pe tehnica "**scoreboarding**" propusa pentru prima data de catre *Seymour Cray* (1964 - CDC 6600). Se impune ca fiecare registru al procesorului din setul de registri sa aiba un "*bit de scor*" asociat. Daca bitul este zero, registrul respectiv e disponibil, daca bitul este 1, registrul respectiv este ocupat. Daca pe un anumit nivel al procesarii este necesar accesul la un anumit registru având bitul de scor asociat pe 1, respectivul nivel va fi întârziat, permitându-i-se accesul doar când bitul respectiv a fost sters de catre procesul care l-a setat. De remarcat ca **ambele solutii bazate pe stagnarea fluxului** (software - NOP sau hardware - semafoare) au acelasi efect defavorabil asupra performantei.

Exista situatii în care hazardul RAW se rezolva prin hardware fara sa cauzeze stagnari ale fluxului de procesare. Aceste tehnici de rezolvare se numesc **tehnici forwarding** (bypassing) bazate pe "pasarea anticipata" a rezultatului instructiunii **i**, nivelului de procesare aferent instructiunii **j** care are nevoie de acest rezultat. În implementarea controlului mecanismelor de forwarding, pot apare si situatii conflictuale care trebuiesc rezolvate pe baza de *arbitrare-priorizare*.

Scheduler-ul HSS [VinFlor00] (dezvoltat la universitatea din Hertfordshire) foloseste tehnica "*merging*" pentru a depasi limitarile introduse prin dependentele reale de date. Aceasta implica combinarea a doua instructiuni într-una singura. Exista trei categorii de astfel de instructiuni. Prima categorie, numita **MOV Merging** implica o pereche de instructiuni în care prima din ele este MOV. A doua categorie numita **Immediate Merging** se caracterizeaza prin faptul ca ambele instructiuni au ca operanzi sursa valori imediate. A treia categorie se numeste **MOV Reabsorption** si are ca a doua instructiune o instructiune MOV sau instructiunea ce se va infiltra va fi convertita la tipul primei instructiuni.

**MOV Merging**

Secventa initiala:

MOV R6, R7  
ADD R3, R6, R5

Secventa modificata:

MOV R6, R7; ADD R3, R6, R5

**Immediate Merging**

Secventa initiala:

SUB R3, R6, #3  
ADD R4, R3, #1

Secventa modificata:

SUB R3, R6, #3; ADD R4, R6, #-2

**MOV Reabsorption**

Secventa initiala:

ADD R3, R4, R5  
MOV R6, R3

Secventa combinata:

ADD R3, R4, R5; ADD R6, R4, R5

c) În cadrul *arhitecturii lui Tomasulo*, detectia hazardurilor si controlul executiei instructiunilor sunt distribuite iar rezultatele instructiunilor sunt "pasate anticipat" direct unitatilor de executie prin intermediul unei magistrale comune numita CDB (Common Data Bus). Statiile de Rezervare (SR) detin câmpuri de TAG (de identificare) necesare în controlul hazardurilor de date între instructiuni. **Qi, Qk** - codifica pe un numar de biti unitatea de executie (ADD, MUL, etc.) sau numarul bufferului LB (aferent instructiunilor Load), care urmeaza sa genereze operandul sursa aferent instructiunii din SR. Daca acest câmp este zero, rezulta ca operandul sursa este deja disponibil într-un câmp **Vi** sau **Vj** al SR sau pur si simplu nu este necesar. Câmpurile Qj, Qk sunt pe post de TAG, adica atunci când o unitate de executie sau un buffer LB "paseaza" rezultatul pe CDB, acest rezultat se înscrie în câmpul Vi sau Vj al acelei SR al carei TAG coincide cu numarul sau numele unitatii de executie sau bufferului LB care a generat rezultatul. Arhitectura lui Tomasulo nu ar functiona corect fara câmpurile de tag Qj, Qk deoarece în cazul unei dependente RAW între instructiuni acestea ar stagna pâna când registrii (operanzii necesari) sunt înscriși în setul de registri generali (la finele fazei WB si nu la finele EX/MEM cum ar fi fost daca existau cele doua câmpuri de TAG).

**11.** Daca alegem strategia Greedy obtinem rata de procesare

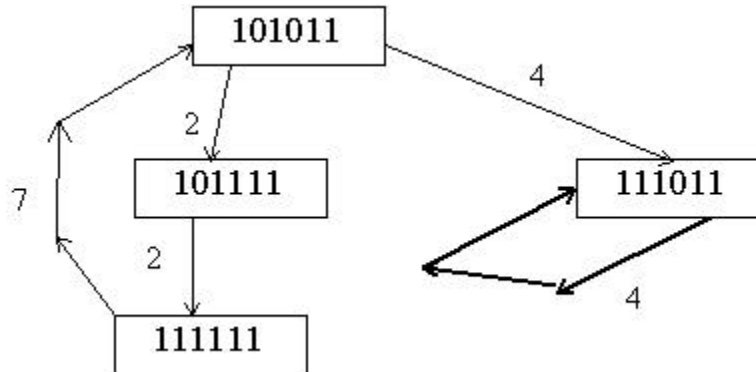
$$IR = \frac{3}{11} = 0,27 \text{ instr/ciclu}$$

Daca alegem strategia non - Greedy rata de procesare obtinuta este:

$$IR = \frac{1}{4} = 0,25 \text{ instr/ciclu}$$



În acest caz e mai avantajos sa alegem strategia Greedy.



13. a) Procesarea vectoriala reprezinta prelucrarea informatiei numerice sub forma de vectori. Ea urmareste în principiu cresterea ratei de procesare la nivelul programului, fiind deci o tehnica de **exploatare a paralelismului la nivelul datelor din program**. De multe ori arhitecturile vectoriale se întâlnesc în literatura sub numele de sisteme SIMD.

Datorita dependentelor de date existente (rezultatul  $x$  depinde si de valoarea sa de la pasul anterior), secventa nu este vectorizabila. Timpul de executie este  $T=201$  cicli (O atribuire, 100 de înmultiri si 100 de adunari).

Secventa modificata implica folosirea unui registru vectorial suplimentar dar care permite vectorizarea partiala a buclei.

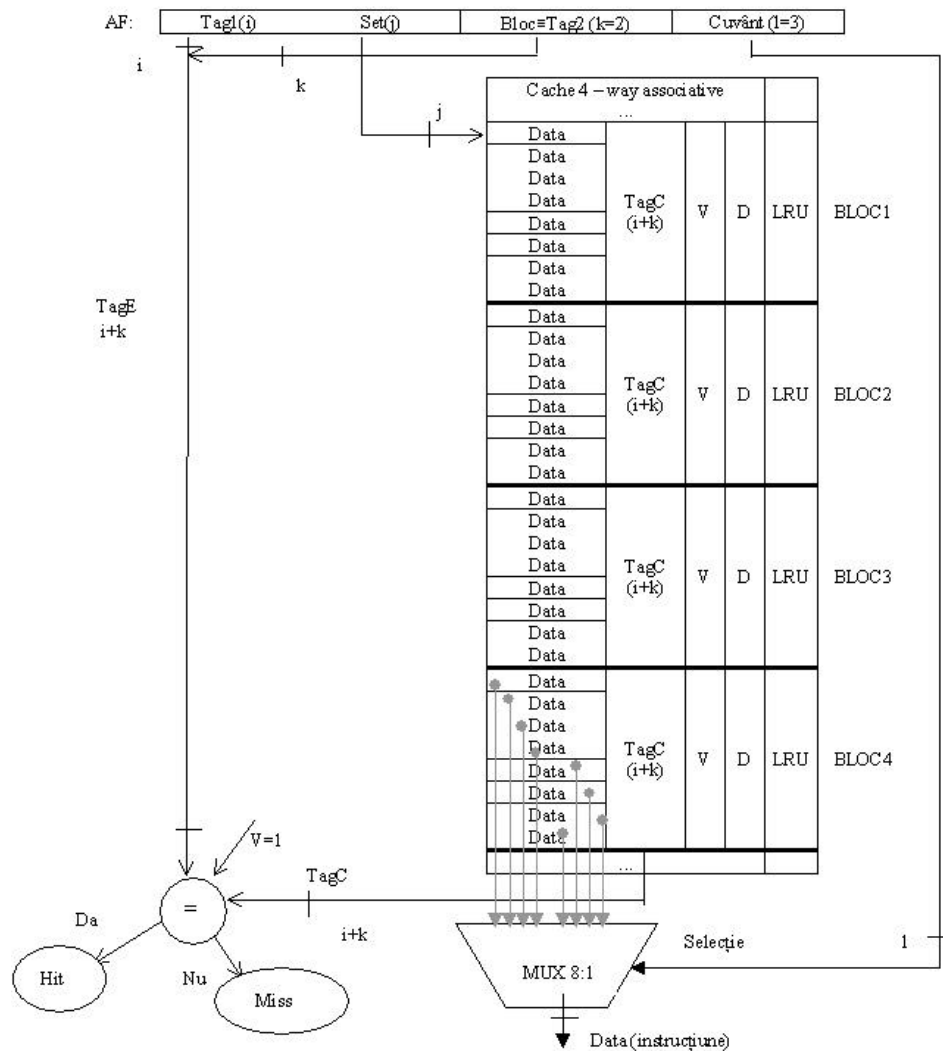
```

x = 0
for i = 1 to 100 do           //vectorizabila
    C[i] = A[i]*B[i]
endfor

for i = 1 to 100 do         //nevectorizabila
    x = x + C[i];
  
```

Timpul de executie este substantial ameliorat  $T=102$  cicli (O atribuire, 1 înmultire vectoriala si 100 de adunari). Timpul de procesare se îmbunatateste substantial prin vectorizarea partiala a buclei

b)



Cache-ul 4-way asociativ conține  $2^j$  seturi, fiecare set conține 4 blocuri.

V – bit de validare (0 – nu e validă data; 1 – validă). Inițial are valoarea 0. Este necesar numai pentru programe automodificabile la cache-urile de instrucțiuni.

D – bit Dirty. Este necesar la scrierea în cache-ul de date.

c) Arhitectura lui Tomasulo a fost proiectată și implementată pentru prima dată în cadrul unității de calcul în virgula mobilă din cadrul sistemului IBM - 360/91. Arhitectura este una de tip superscalar având mai multe unități de execuție, iar algoritmul de control al acestei structuri stabilește relativ la o instrucțiune adusă, momentul în care aceasta poate fi lansată în execuție și respectiv unitatea de execuție care va procesa instrucțiunea. Logica de

detectie a hazardurilor este distribuita. Arhitectura permite executia multipla si Out of Order a instructiunilor si constituie modelul de referinta în reorganizarea dinamica a instructiunilor într-un procesor superscalar. Algoritmul de gestiune aferent arhitecturii permite anulara hazardurilor WAR si WAW printr-un ingenios mecanism hardware de redenumire a registrilor. Acest lucru este posibil pentru ca resursele tip sursa folosite si aflate în starea "BUSY", nu se adreseaza ca nume de registri ci ca nume de unitati de executie ce vor produce aceste surse. Mecanismul de forwarding din arhitectura lui Tomasulo, are meritul de a reduce semnificativ din presiunea la "citire" asupra setului general de registri logici, speculând dependentele RAW între instructiuni.

**14. a)** În procesul de proiectare al procesoarelor, aferent generatiilor viitoare, accentul principal nu se mai pune pe implementarea hardware, ci pe proiectarea arhitecturii în strânsa legatura cu aplicatiile potentiale. Se porneste de la o arhitectura de baza (generica), puternic parametrizata, care este modificata si îmbunatatita dinamic, prin simulari laborioase pe programe de test (benchmark-uri) reprezentative. **Figura 0.1. "Etapetele de simulare, comparare si determinare efectiva ale unei arhitecturi optime, pornind de la sursa HLL (High Level Languages) a programelor de test si pâna la implementarea hardware a arhitecturii"** din cadrul prezentei lucrari evidentiaza tocmai pasii ceruti.

Codul original sursa (în limbajul C/Fortran) al benchmarkului este trecut întâi printr-un **cross-compiler** (de exemplu generatorul de cod obiect "gnuCC" sub Unix) care produce formatul corect al codului în mnemonica de asamblare, precum si directive de asamblare, comenzi de alocare a memoriei etc. Optional, codul rezultat în acest punct poate fi "*rearanjat*" prin intermediul unui **scheduler**, care împacheteaza instructiunile în grupuri de instructiuni independente. Codul obiect rezultat este trecut printr-un **simulator execution driven** puternic parametrizabil, care produce la iesire un fisier trace de instructiuni. Acesta reprezinta totalitatea instantelor dinamice, aferente instructiunilor statice (codul masina al benchmark-urilor în mnemonica de asamblare) scrise în ordinea executiei lor. În final, aceste trace-uri constituie intrari pentru **simulatorul trace driven**, de asemenea parametrizabil, care genereaza parametrii completi aferenti procesarii, pentru determinarea optimului de performanta în anumite conditii de intrare.

b) Problema optimizarii programelor în vederea procesarii lor pe arhitecturi superscalare si VLIW este una de mare interes în cercetarea actuala. Optimizarea se refera în general la 2 aspecte: optimizarea locala, adica în cadrul unitatilor secventiale de program (basic-block-uri) si respectiv

optimizarea globala, adica a întregului program. Optimizarea locala a basic-block-urilor se realizeaza folosind algoritmi cvasioptimali într-o singura trecere, de tip "List Scheduling" (LS). Optimizarea globala a programului se bazeaza pe algoritmi deterministi de tip "Trace Scheduling" sau pe algoritmi euristici de tip "Enhanced Percolation". Paralelismul la nivelul basic-block-urilor, este relativ scazut (2-3 instructiuni). Deoarece majoritatea programelor HLL sunt scrise în limbaje imperative si pentru masini secventiale cu un numar limitat de registre în vederea stocarii temporare a variabilelor, este de asteptat ca gradul de dependente între instructiunile adiacente sa fie ridicat. Asadar pentru marirea nivelului de paralelism este necesara suprapunerea executiei unor instructiuni situate în basic-block-uri diferite (*software pipelining*, *loop unrolling*), ceea ce conduce la ideea optimizarii globale. Problema optimizarii globale este de natura NP - completa.

c) Arhitecturile MEM sunt implementate în doua variante distincte: procesoare *superscalare* si respectiv procesoare *VLIW* (very large instruction word). În varianta superscalara cade în sarcina hardului sa verifice independenta instructiunilor aduse în buffer-ul de prefetch si sa le lanseze în executii multiple spre unitatile de executie pipeline-izate. Caracteristic procesoarelor superscalare este faptul ca dependentele de date între instructiuni se rezolva prin hardware, în momentul decodificarii instructiunilor. Desigur ca aceste arhitecturi au o complexitate hardware deosebita determinata de detectia si solutionarea hazardului între instructiuni. Complexitatea logicii de detectie a independentei instructiunilor ce se doresc a fi lansate în executie simultan, creste proportional cu patratul numarului de instructiuni. De asemenea, ele sunt limitate în posibilitatea de a exploata paralelismul la nivelul instructiunilor, de capacitatea limitata a buffer-ului de prefetch. La ora actuala, pe plan comercial, piata este dominata de microprocesoarele superscalare, în principal datorita dezideratelor de compatibilitate între versiunile din cadrul aceleiasi familii. La procesoarele VLIW, mai rare decât cele superscalare, în implementari comerciale, cade în sarcina compilatorului de a reorganiza programul original în scopul "*împachetarii*" într-o singura instructiune multipla a mai multor instructiuni RISC primitive si independente, care vor fi alocate unitatilor de executie în conformitate stricta cu pozitia lor în instructiunea multipla. Un procesor VLIW aduce mai multe instructiuni primitive simultan si le lanseaza în executie concurent spre unitatile functionale deja alese de compilator (scheduler). Acest model de procesor nu mai necesita sincronizari si comunicatii de date suplimentare între instructiunile primitive dupa momentul decodificarii lor, fiind astfel mai simplu din punct de vedere hardware decât modelul superscalar.

Dificultatile principale ale modelului VLIW sunt urmatoarele [VinFlor00]:

- ❑ Paralelismul limitat al aplicatiei, ceea ce determina ca unitatile de executie sa nu fie ocupate permanent, fapt valabil de altfel si la modelul superscalar.
- ❑ Incompatibilitate software cu modele succesive si compatibile de procesoare care nu pot avea în general un model VLIW identic datorita faptului ca paralelismul la nivelul instructiunilor depinde de latentele operatiilor procesorului scalar, de numarul unitatilor functionale si de alte caracteristici hardware ale acestuia.
- ❑ Dificultati deosebite în reorganizarea aplicatiei (scheduling) în vederea determinarii unor instructiuni primitive independente sau cu un grad scazut de dependente.
- ❑ Cresterea complexitatii hardware si a costurilor ca urmare a resurselor multiplicat, cailor de informatie "latite".
- ❑ Cresterea necesitatilor de memorare ale programelor datorita reorganizarilor soft si "împachetarii" instructiunilor primitive în cadrul unor instructiuni multiple care necesita introducerea unor instructiuni NOP (atunci când nu exista instructiuni de un anumit tip disponibile spre a fi asamblate într-o instructiune multipla).

De remarcat ca modelele superscalar si VLIW nu sunt exclusive, în implementarile reale se întâlnesc adesea procesoare hibride, în încercarea de a se optimiza raportul performanta pret (HSA, IA64).

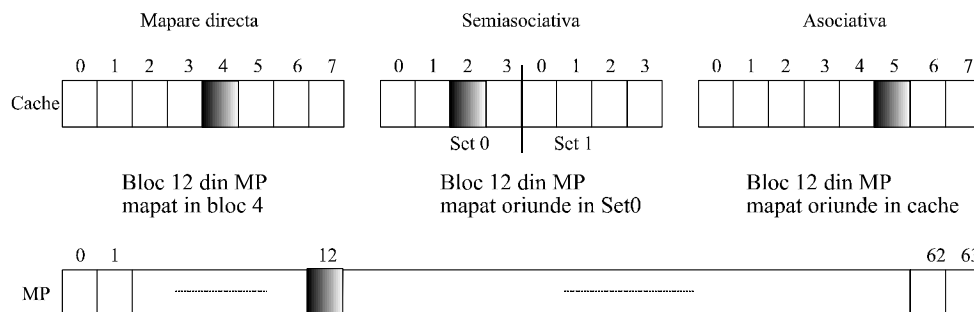
Caracteristica principala a arhitecturii HSA (Hatfield Superscalar Architecture) o constituie exploatarea paralelismului la nivelul instructiunii, într-un mediu superscalar, prin scheduling agresiv aplicat codului sursa în momentul compilarii. Scheduler-ul dezvoltat la Universitatea din Hertfordshire, UK (HSS - Hatfield Superscalar Scheduler) [VinFlor00] a fost implementat ca parte integranta a proiectului HSA (Hatfield Superscalar Architecture) – o arhitectura superscalara minima care combina cele mai bune caracteristici ale conceptelor VLIW si superscalare. HSS rearanjeaza codul HSA scris în limbaj de asamblare pentru a forma grupuri de instructiuni care pot fi expediate în paralel unitatilor functionale în momentul executiei.

**16. a)** Ideal ar fi ca un sistem paralel dotat cu N procesoare sa proceseze un program de N ori mai rapid decât un sistem monoprocesor, cerinta numita *scalabilitate completa*. În realitate, acest deziderat de scalabilitate nu se realizeaza din multiple motive. În privinta scalabilitatii, aceasta este mai usor de realizat pe un sistem cu resurse distribuite decât pe unul având resurse centralizate, întrucât acestea din urma constituie un factor de

“strangulare” a activitatii. Dintre cauzele care stau în calea unei scalabilitati ideale, se amintesc [VinFlor00]:

- Gradul de secventialitate intrinsec al algoritmului executat. Exista în cadrul unui algoritm operatii secventiale dependente de date si deci imposibil de partajat în vederea procesarii lor paralele pe mai multe procesoare. Legea lui Amdahl sugereaza ca un procentaj ( $f \times 100\%$ ) oricât de scazut de calcule secventiale impune o limita superioara a accelerarii ( $1/f$ ) care poate fi obtinuta pentru un anumit algoritm paralel pe un sistem paralel de calcul, indiferent de numarul  $N$  al procesoarelor din sistem si topologia de interconectare a acestora.
- Timpul consumat cu sincronizarea si comunicarea între procesele rezidente pe diversele ( $\mu$ )procesoare din sistem.
- Imposibilitatea balansarii optimale a activitatii procesoarelor din sistem, adica frecvent nu se poate evita situatia în care anumite procesoare sa fie practic inactive sau cu un grad scazut de utilizare.
- Operatiile de I/O, în cazul nesuprapunerii lor peste activitatea de executie a task-ului de catre procesor.
- Planificarea suboptimala a proceselor din punct de vedere software (activare proces, punere în asteptare a unui proces, schimbarea contextului în comutarea proceselor)

b) Memoria cache este o memorie situata din punct de vedere logic între CPU si memoria principala (uzual DRAM), mai mica, mai rapida si mai scumpa (per byte) decât aceasta si gestionata – în general prin hardware – astfel încât sa existe o cât mai mare probabilitate statistica de gasire a datei accesate de catre CPU, în cache. Din punct de vedere arhitectural, exista 3 tipuri distincte de memorii cache în conformitate cu gradul de asociativitate: cu *mapare directa*, *semiasociative* si *complet asociative* [VinFlor00].



La cache-urile cu **mapare directa**, ideea principala consta în faptul ca un bloc din MP poate fi gasit în cache (hit) într-un bloc unic determinat. În acest caz regula de mapare a unui bloc din MP în cache este:

(Adresa bloc MP) modulo (Nr. blocuri din cache)

Strictetea regulii de mapare conduce la o simplitate constructiva a acestor memorii dar si la fenomenul de interferenta al blocurilor din MP în cache.

La **cache-urile semiasociative** exista mai multe seturi, fiecare set având mai multe blocuri componente. Aici, regula de mapare precizeaza strict doar setul în care se poate afla blocul dorit, astfel:

(Adresa bloc MP) modulo (Nr. seturi din cache)

Blocul dorit se poate mapa oriunde în setul respectiv. La un miss în cache, înainte de încărcarea noului bloc din MP, trebuie evacuat un anumit bloc din setul respectiv. În principiu exista implementate doua-trei tipuri de algoritmi de evacuare: pseudorandom (cvasialeator), FIFO si LRU ("Least Recently Used"). Algoritmul LRU evacueaza blocul din cache cel mai demult neaccesat, în baza principiului de localitate temporala. Daca un set din cache-ul semiasociativ contine N blocuri atunci cache-ul se mai numeste "tip *N-way set associative*".

Într-un cache semiasociativ rata de interferenta se reduce odata cu cresterea gradului de asociativitate (N "mare"). Prin reducerea posibilelor interferente ale blocurilor, cresterea gradului de asociativitate determina îmbunatatirea ratei de hit si deci a performantei globale. Pe de alta parte însa, asociativitatea impune cautarea dupa continut ceea ce conduce la complicatii structurale si deci la cresterea timpului de acces la cache si implicit la diminuarea performantei globale. Optimizarea gradului de asociativitate, a capacitatii cache, a lungimii blocului din cache nu se poate face decât prin laborioase simulari software, variind toti acesti parametrii în vederea minimizarii ratei globale de procesare a instructiunilor [instr./cicli].

Memoriile cache **complet asociative**, implementeaza practic un singur set permitând maparea blocului practic oriunde în cache. Ele nu se implementeaza deocamdata în siliciu datorita complexitatii deosebite si a timpului prohibit de cautare. Reduc însa total interferentele blocurilor la aceeasi locatie cache si constituie o metrica superioara utila în evaluarea ratei de hit pentru celelalte tipuri de cache-uri (prin comparatie).

Într-un Sistem Multiprocesor o informatie poate fi **privata** (locala) - daca ea este utilizata de catre un singur CPU, sau **partajata** - daca se impune a fi utilizata de catre mai multe procesoare [VinFlor00]. În cazul informatiilor partajate, de obicei acestea sunt memorate în cache-ul local al unui anumit CPU, reducându-se astfel latenta necesara accesarii lor de catre un respectivul procesor. Totodata, se reduc coliziunile implicate de accesul

simultan (citiri) al mai multor CPU-uri la respectiva informatie (informatia necesara mai multor procesoare coincide) si respectiv necesitatile de largime de banda a busului comun. Din pacate, pe lânga aceste avantaje, partajarea informatiilor introduce un dezavantaj major, constând în problema **coerentei cache-urilor**. În principiu, *un sistem de memorie aferent unui Sistem Multiprocesor este coerent daca orice citire a unei informatii returneaza informatia scrisa cel mai recent*. Astfel, problema coerentei cache-urilor în SMM se refera la necesitatea ca un bloc din memoria globala - memorat în mai multe memorii cache locale - sa fie actualizat în toate aceste memorii, la orice acces de scriere al unui anumit CPU.

În vederea mentinerii coerentei cache-urilor cu precadere în sistemele multiprocesor – exista 2 posibilitati în functie de ce se întâmpla la o scriere:

- ❑ **Write invalidate** – prin care CPU care scrie determina ca toate copiile din celelalte memorii cache sa fie invalidate înainte ca el sa-si modifice blocul din cache-ul propriu.
- ❑ **Write Broadcast** – CPU care scrie pune data de scris pe busul comun spre a fi actualizate toate copiile din celelalte cache-uri.

Ambele strategii de mentinere a coerentei pot fi asociate cu oricare dintre protocoalele de scriere (Write Through, Write Back) dar de cele mai multe ori se prefera WB cu invalidate.

c) În ce priveste limitările arhitecturale ale paradigmei superscalare referitoare la maximizarea *ratei de aducere a instructiunilor din memorie* la aceasta întrebare s-a raspunsul prin intermediul problemei 10 pct.a.

Limitările de tip "*issue bottleneck*" sunt determinate fundamental de gradul de secventialitate intrinsec programelor (hazarduri structurale si de date), dar si de mecanismul de aducere a instructiunilor din memorie ( $IR \leq FR$ ). În ultimii ani s-au dezvoltat doua tehnici hardware menite sa exploateze redundanta existenta în programe, reducând timpului de executie al acestora prin colapsarea dinamica a dependentelor de date: **reutilizarea dinamica a instructiunilor** si **predictia valorilor** [VinFlor00].

VP (*value prediction*) predictioneaza rezultatele instructiunilor bazat pe rezultatele cunoscute anterior, efectueaza operatiile folosind valorile prezise iar executia speculativa este confirmata la un moment ulterior - *late validation* - când valorile corecte devin disponibile, deci dupa executia instructiunii. În cazul unei predictii corecte, calea critica este redusa întrucât instructiunile care s-ar executa secvential în mod normal, pot fi executate în paralel. În caz contrar, instructiunile executate cu intrari eronate trebuiesc reexecutate.

IR (*instruction reuse*), recunoaste un lant de instructiuni dependente executat anterior si nu-l mai executa din nou - *early validation* - actualizând



doar diferite date în tabelele hardware aferente. Astfel, IR *comprima* un lant de instructiuni din calea critica de executie a programului.

Alti factori care determina limitarea ratei de issue a instructiunilor sunt: largimea de banda limitata a interfetei procesor - cache, missurile în cache-ul de date, alias-urile de memorie. Pentru a contracara efectul acestor factori se poate utiliza cu succes mecanismul **Data Write Buffer** (DWB). DWB reprezinta un mic procesor de iesire care lucreaza în paralel cu CPU degrevându-l pe acesta de sarcina scrierii în cache. DWB ofera porturi de scriere virtuale multiple spre deosebire de DataCache care contine un singur port de citire (LOAD) si un singur port de scriere (STORE). DWB rezolva prin "bypassing" elegant hazardurile de tip "LOAD after STORE" cu adrese identice, nemaifiind deci necesara accesarea sistemului de memorie de catre instructiunea LOAD.

**17. a)** Modelele CISC sunt caracterizate de un set foarte bogat de instructiuni - masina, formate de instructiuni de lungime variabila, numeroase moduri de adresare deosebit de sofisticate, lipsa unei interfete hardware – software optimizate etc. Evident ca aceasta complexitate arhitecturala are o repercursiune negativa asupra performantei masinii.

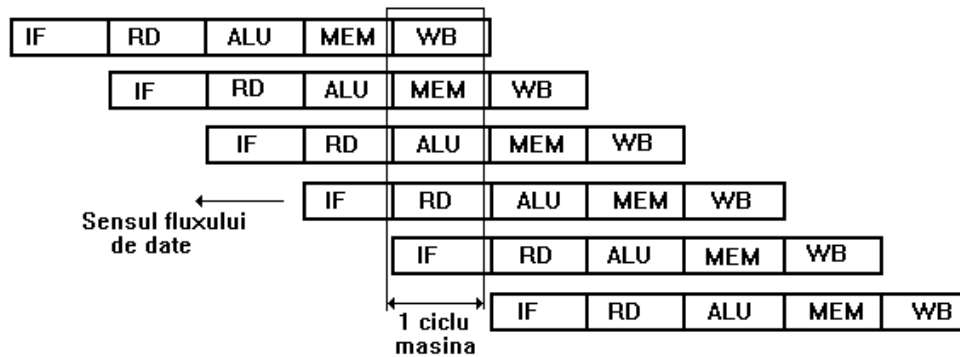
Microprocesoarele RISC au aparut ca o replica la lipsa de eficienta a modelului conventional de procesor de tip CISC. Caracteristicile de baza ale modelului RISC sunt:

- Unitate de comanda hardware în general cablata, cu firmware redus sau deloc, ceea ce maresta rata de executie a instructiunilor.
- Utilizarea tehnicilor de procesare pipeline a instructiunilor, ceea ce implica o **rata teoretica de executie de o instructiune / ciclu**, pe modelele de procesoare care pot lansa în executie la un moment dat o singura instructiune (procesoare scalare). Sunt mai rapide decât modelele CISC.
- Memorie sistem de înalta performanta, prin implementarea unor arhitecturi avansate de memorie cache si MMU.
- Set relativ redus de instructiuni simple, majoritatea fara referire la memorie si cu putine moduri de adresare. În general, doar instructiunile LOAD / STORE sunt cu referire la memorie (arhitectura tip LOAD / STORE). Caracteristica de "set redus de instructiuni" are sensul de set optimizat de instructiuni în vederea implementarii aplicatiilor propuse (în special implementarii limbajelor de nivel înalt - C, C++, Pascal).
- Format fix al instructiunilor, codificate în general pe un singur cuvânt de 32 biti, mai recent pe 64 biti (Alpha 21264, IA-64 Merced).
- Set de registre generale substantial mai mare decât la CISC-uri, în vederea lucrului "în ferestre" (register windows), util în optimizarea

instructiunilor CALL / RET. Numarul mare de registre generale este util si pentru marirea spatiului intern de procesare, tratarii optimizate a evenimentelor de exceptie, modelului ortogonal de programare. Registrul R0 este cablat la zero în majoritatea implementarilor, pentru optimizarea modurilor de adresare si a instructiunilor.

b) Procesarea pipeline a instructiunilor reprezinta o tehnica de procesare prin intermediul careia fazele (ciclii) aferente multiplelor instructiuni sunt suprapuse în timp. Se înțelege printr-o faza aferenta unei instructiuni masina o prelucrare atomica a informatiei care se desfasoara dupa un algoritm implementat în hardware si care dureaza unul sau mai multi tacti. Arhitectura microprocesoarelor RISC este mai bine adaptata la procesarea pipeline decât cea a sistemelor conventionale CISC, datorita instructiunilor de lungime fixa, a modurilor de adresare specifice, a structurii interne bazate pe registre generale. Microprocesoarele RISC uzuale detin o structura pipeline de instructiuni întregi pe 4 - 6 nivele (microprocesoarele MIPS au 5 nivele tipice):

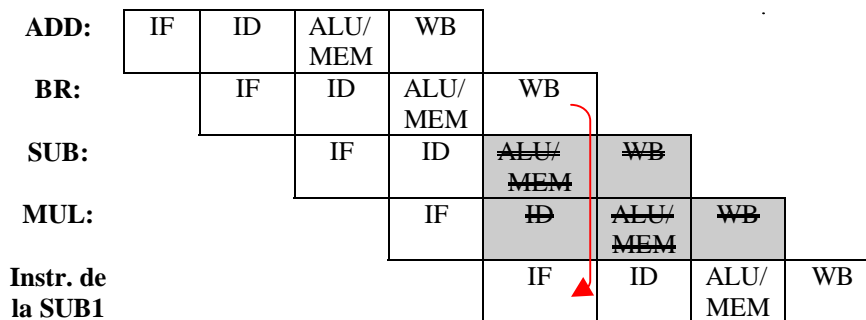
1. Nivelul **IF** (instruction fetch) - se calculeaza adresa instructiunii ce trebuie citita din cache-ul de instructiuni sau din memoria principala si se aduce instructiunea;
2. Nivelul **RD (ID)** - se decodifica instructiunea adusa si se citesc operanzii din setul de registri generali. În cazul instructiunilor de salt, pe parcursul acestei faze se calculeaza adresa de salt;
3. Nivelul **ALU** - se executa operatia ALU asupra operanzilor selectati în cazul instructiunilor aritmetico-logice; se calculeaza adresa de acces la memoria de date pentru instructiunile LOAD / STORE;
4. Nivelul **MEM** - se acceseaza memoria cache de date sau memoria principala, însa numai pentru instructiunile LOAD / STORE. Acest nivel pe functia de citire poate pune probleme datorate neconcordanței între rata de procesare si timpul de acces la memoria principala. Rezulta deci ca într-o structura pipeline cu N nivele, memoria trebuie sa fie în principiu de N ori mai rapida decât într-o structura de calcul conventionala. Acest lucru se realizeaza prin implementarea de arhitecturi de memorie rapide (cache, memorii cu acces întretesut, etc.). Desigur ca un ciclu cu MISS în cache pe acest nivel (ca si pe nivelul IF de altfel), va determina stagnarea temporara a acceselor la memorie sau chiar a procesarii interne. La scriere, problema aceasta nu se pune datorita procesorului de iesire specializat DWB care lucreaza în paralel cu procesorul central dupa cum deja am aratat.
5. Nivelul **WB (write buffer)** - se scrie rezultatul ALU sau data citita din memorie (în cazul unei instructiuni LOAD) în registrul destinatie din setul de registri generali ai microprocesorului.



Se observa necesitatea suprapunerii a 2 nivele concurentiale: nivelul IF si respectiv nivelul MEM, ambele cu referire la memorie. În cazul microprocesoarelor RISC aceasta situatie se rezolva prin busuri separate între procesor si memoria de date respectiv de instructiuni (arhitectura Harvard).

c) Vezi problema 14c).

d) Procesarea pipeline a secventei de instructiuni este urmatoarea:



Instructiunea **SUB** este extrasa din cache fara a sti daca saltul se va face sau nu. Abia dupa faza ALU/MEM aferenta instructiunii **BR** se cunoaste rezultatul saltului (*taken*). Între timp inclusiv instructiunea **MUL** a fost extrasa din cache iar instructiunea anterioara (SUB) a fost decodificata. Din cauza ca saltul este taken cele doua instructiuni SUB si MUL s-au executat eronat. În acest moment se va executa faza IF aferenta instructiunii de la adresa SUB1 iar executia instructiunilor executate eronat se întrerupe. Valoarea registrilor pipeline este resetata. **Delay slot**-ul este de 2 cicli de tact.

**19. a)** Implementarea memoriei **cache** are rolul de a micsora timpul mediu de acces al microprocesorului la memoria DRAM Cache-ul este adresat de catre CPU în paralel cu memoria principala (MP): daca data dorita a fi

accesata se gaseste în cache, accesul la MP se aborteaza, daca nu, se acceseaza MP cu penalizarile de timp impuse de latentă mai mare a acesteia, relativ ridicată în comparatie cu frecvența de tact a CPU. Oricum, data accesata din MP se va introduce si în cache. Memoriile cache îmbunătătesc performanța îndeosebi pe citirile cu hit iar în cazul utilizării scrierii tip “Write Back” si pe scrierile cu hit.

Pentru a reduce rata de miss a cache-urilor mapate direct (fara sa se afecteze însă timpul de hit sau penalitatea în caz de miss), cercetatorul Norman Jouppi a propus conceptul de “**victim cache**” - o memorie mica complet asociativă, plasată între primul nivel de cache mapat direct si memoria principală. Blocurile înlocuite din cache-ul principal datorită unui miss sunt temporar stocate în victim cache. Dacă sunt referite din nou înainte de a fi înlocuite din victim cache, ele pot fi extrase direct din victim cache cu o penalitate mai mica decât cea a memoriei principale. Algoritmul de victim cache încearcă sa izoleze blocurile conflictuale si sa le memoreze doar unul în cache-ul principal restul în victim cache. Dacă numărul blocurilor conflictuale este suficient de mic sa se potrivească în victim cache, atât rata de miss în nivelul următor de memorie cât si timpul mediu de acces va fi îmbunătătit datorită penalității reduse implicate de prezenta blocurilor în victim cache.

Pentru a reduce numărul de interschimbari dintre cache-ul principal si victim cache, Stiliadis si Varma au introdus un nou concept numit **selective victim cache**(SVC). Cu SVC, blocurile aduse din memoria principală sunt plasate selectiv fie în cache-ul principal cu mapare directă fie în selective victim cache, folosind un algoritm de predicție euristic bazat pe istoria folosirii sale. Blocurile care sunt mai puțin probabil sa fie accesate în viitor sunt plasate în SVC si nu în cache-ul principal. Predicția este de asemenea folosită în cazul unui miss în cache-ul principal pentru a determina dacă este necesară o schimbare a blocurilor conflictuale. Algoritmul obiectiv este de a plasa blocurile, care sunt mai probabil a fi referite din nou, în cache-ul principal si altele în victim cache.

b) **Memoria virtuală** (MV) reprezintă o tehnică de organizare a memoriei prin intermediul careia programatorul “vede” un spațiu virtual de adresare foarte mare si care, fara ca programatorul sa “simtă”, este mapat în memoria fizic disponibilă. În cazul MV, memoria principală este analoagă memoriei cache între CPU (*Central Processing Unit*) si memoria principală, numai ca de această dată ea se situează între CPU si discul hard. Deci memoria principală (MP) se comportă oarecum ca un cache între CPU si discul hard. Prin mecanismele de MV se mărește probabilitatea ca informația ce se dorește a fi accesată de către CPU din spațiul virtual (disc), sa se afle în MP,

reducându-se astfel dramatic timpul de acces de la  $8 \div 15$  ms la  $45 \div 70$  ns în tehnologiile anilor 1999!

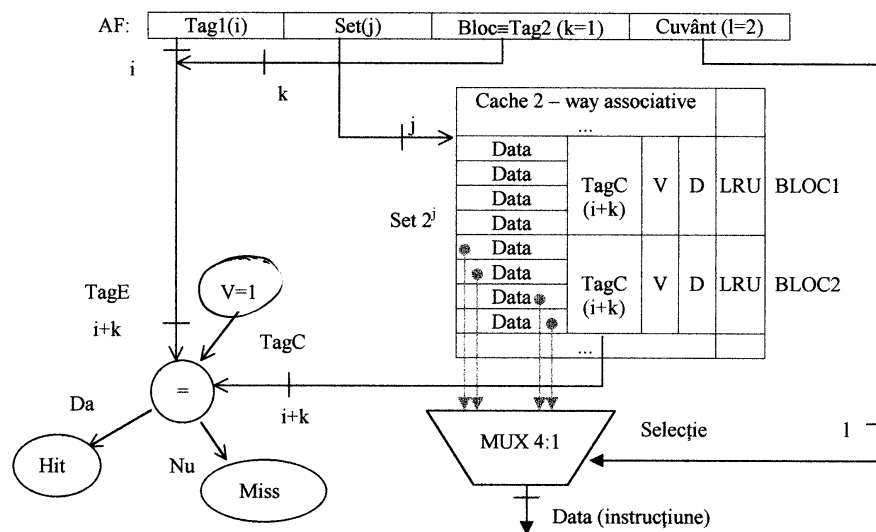
21. a.

Cache-ul **semiasociativ** conține  $2^j$  seturi, fiecare set conține 2 blocuri.

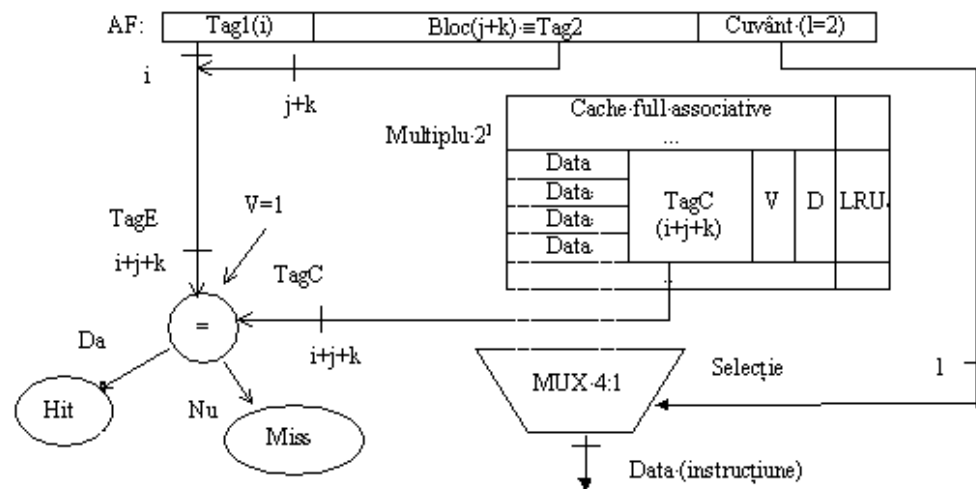
V – bit de validare (0 – nu e validă data; 1 – validă;). Inițial are valoarea 0.

Este necesar numai pentru programe automodificabile la cache-urile de instrucțiuni.

D – bit Dirty. Este necesar la scrierea în cache-ul de date.

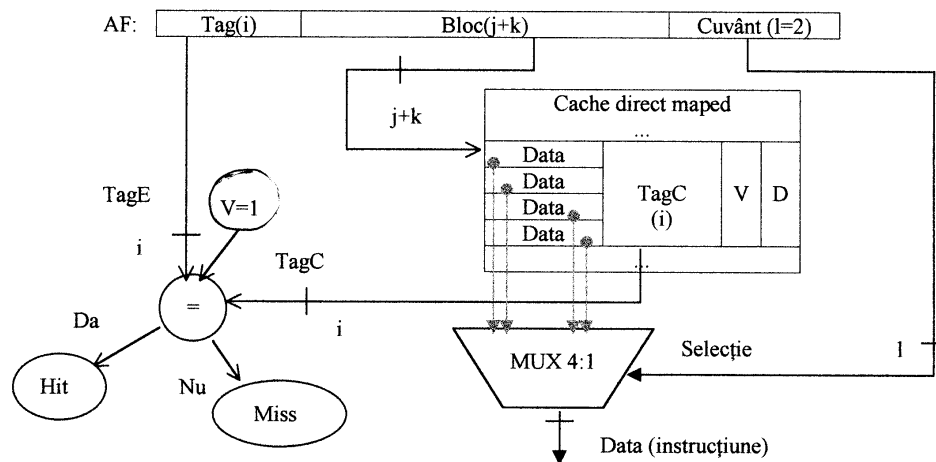


b.



În cazul cache-urilor **full asociative** dispăre câmpul set. Practic există un singur set. Datele pot fi memorate oriunde (în orice bloc) în cache.

c. În cazul cache-urilor **mapate direct**, datele vor fi memorate în același loc de fiecare dată când sunt accesate. Din acest motiv vom ști la fiecare acces ce dată va fi evacuată din cache, nemaifiind necesar câmpul LRU (evacuare implicită).



22. a)

| In Order                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | Out of Order                                                                                                                                                                                                                                                                                                                                                                                   |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>❑ <i>Logica de trimitere în execuție.</i> O instrucțiune va fi executată abia după ce toate celelalte instrucțiuni anterioare, de care ea depinde, au fost executate.</li> </ul>                                                                                                                                                                                                                                                                                      | <ul style="list-style-type: none"> <li>❑ <i>Logica de trimitere în execuție.</i> Instrucțiunile pot fi executate în altă ordine decât cea inițială cu condiția de a păstra tot timpul valid contextul procesorului (consistența și integritatea registrilor).</li> </ul>                                                                                                                       |
| <ul style="list-style-type: none"> <li>❑ <i>Extrage paralelismul la nivel software</i> cu ajutorul unui scheduler.</li> <li>• În momentul compilării rearanjează instrucțiunile și redenumeste registre.</li> <li>• Ascunde întârzierile datorate instrucțiunilor de salt prin umplerea branch delay slotului cu instrucțiuni valide sau branch prediction static (întârziind procesarea instrucțiunilor următoare cu un număr de cicli egal cu BDS până când adresa de salt devine disponibilă).</li> </ul> | <ul style="list-style-type: none"> <li>❑ <i>Extrage paralelismul la nivel hardware.</i> Trebuie tratate următoarele patru procese: <ul style="list-style-type: none"> <li>• <i>Branch Prediction Dinamic</i></li> <li>• <i>Redenumirea registrilor</i></li> <li>• <i>Expedierea instrucțiunilor spre execuție</i></li> <li>• <i>Mentinerea precisă a întreruperilor</i></li> </ul> </li> </ul> |

b) Procesarea «Out of Order» a instructiunilor cu referire la memorie într-un program este dificila datorita accesarii aceleiasi adrese de memorie de catre o instructiune Load respectiv una Store. Exemplificam pe secventa de instructiuni urmatoare:

Presupunem ca la adresa 2000h avem memorata valoarea 100.

```
LOAD  R1, 2000h
ADD    R1, R1, #12
STORE  R1, 2000h
```

În urma executiei în registrul R1 si implicit la adresa 2000h vom avea valoarea 112.

Prin *Out of Order* aplicat instructiunilor cu referire la memorie valoarea din registrul R1 precum si cea din memorie de la adresa 2000h ar fi alterata.

Secventele de program în limbajul unui procesor RISC ar deveni:

```
R1 ← x[i];
(R1) → a[2i];  STORE Adr1
R2 ← a[i+1];  LOAD  Adr2
R6 ← (R2) + 5;
(R6) → y[i];
```

Cele doua instructiuni cu referire la memorie ar fi paralelizabile (executabile Out of Order) daca  $i \neq 1$  ( $\text{Adr1} \neq \text{Adr2}$ ).

Benchmark-ul **B2** s-ar procesa mai repede pe un procesor superscalar cu executie *Out of Order* a instructiunilor, pentru ca cele doua aliasuri ( $i=1$ ) au fost scoase în afara buclei, prin urmare în cadrul buclei **B2**, paralelizarea Load / Store e posibila.

**23.** a. Vezi pr. 44 a), b).

Prin «renaming» aplicat registrului R1 putem elimina hazardul WAW dintre instructiunile (1 si 5) si (2 si 6), deci le putem trimite în executie în acelasi ciclu.

**Executia:** tact 1 - instructiunile: 1, 5, 3.  
tact 2 - instructiunile: 2, 6.  
tact 3 - instructiunea: 4.

**24.** Nivelul **WR** este mai prioritar datorita hazardurilor RAW între doua instructiuni succesive (vezi si **50.b**).

25.  $FR_{med} = 5$ ; Da. Predictia a  $N$  PC-uri simultan implica  $FR_{med} \cong N \times 5$ .

26.  $C \rightarrow (N-1) + (N-2) + \dots + 2 + 1 = N(N-1) / 2$  comparatoare RAW.

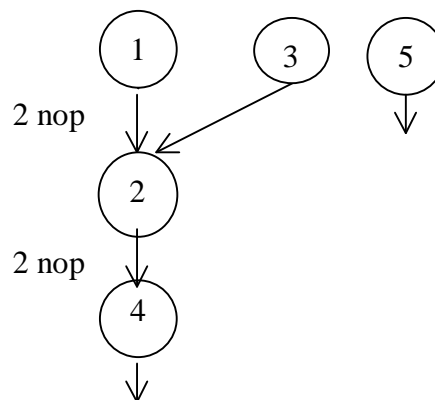
Pentru  $(N+1)$  instructiuni  $\Rightarrow N(N+1) / 2$  comparatoare  $\rightarrow C'$

$C' / C = N(N+1) / N(N-1) \Rightarrow C' = C(N+1) / (N-1)$

27.

- a) **A.** Strictetea regulii de mapare conduce la o simplitate constructiva a memoriilor cache dar si la fenomenul de interferenta al blocurilor din MP în cache. Presupunând o MP de 64 de blocuri si un cache de 8 locatii  $\Rightarrow$  rata de interferenta este de **8** (8 blocuri din MP se mapeaza în acelasi bloc din cache). Crescând dimensiunea cache-ului la 16 blocuri  $\Rightarrow$  rata de interferenta devine **4** (se reduce la jumătate numărul blocurilor conflictuale).
- b) **F.** Justificarea o constituie chiar definitia maparii directe:  
(**Adresa bloc MP**) modulo (**Nr. blocuri din cache**)
- c) **F.** Scrieri în cache au loc si în cazul scrierilor cu **hit** în cache (se scrie numai câmpul de date nu si TAG-ul).
- d) **F.** Într-un cache semiasociativ rata de interferenta se reduce odata cu cresterea gradului de asociativitate. Prin reducerea posibilelor interferente ale blocurilor, cresterea gradului de asociativitate determina îmbunatatirea ratei de hit si deci a performantei globale.

28. a)



b) 5 cicli pentru instr. + 4 cicli nop = 9 cicli executie

c)  $1 - 3 - 5 - 2 - \text{nop} - \text{nop} - 4 \Rightarrow 7$  cicli executie



29. a.)

Instrucțiunea de salt:

|    |    |     |     |     |     |     |    |
|----|----|-----|-----|-----|-----|-----|----|
| IF | ID | ALU | MEM | WB  |     |     |    |
|    | IF | ID  | ALU | MEM | WB  |     |    |
|    |    | IF  | ID  | ALU | MEM | WB  |    |
|    |    |     | IF  | ID  | ALU | MEM | WB |

*Branch delay slot* = 2 cicli.

b) Motivul implementarii de busuri si memorii cache separate pe instructiuni, respectiv date în cazul majoritatii procesoarelor RISC (pipeline) consta în faptul ca nu exista coliziuni la memorie de tipul (IF, MEM).

c) Instructiunile CALL / RET sunt mari consumatoare de timp în cazul procesoarelor CISC datorita salvarilor si restaurarilor de registri (registrul stare program, registrul de flaguri, PC) pe care acestea le executa de fiecare data când sunt apelate. Evitarea consumului de timp în cazul microprocesoarelor RISC se face prin implementarea *ferestrelor de registre* sau prin inlining-ul procedurilor (utilizarea de macroinstructiuni).

30. Codificarea instructiunii este urmatoarea:

|                 |
|-----------------|
| 7.....0         |
| Opcode          |
| Deplasament (1) |
| Deplasament (2) |

Se efectueaza urmatoarele operatii:

| Operatia executata        | Durata executie (cicli) |
|---------------------------|-------------------------|
| Fetch Instructiune        | 6                       |
| Fetch Deplasament (1)     | 4                       |
| Fetch Deplasament (2)     | 4                       |
| Calcul adresa de memorare | 2                       |
| Sciere A în memorie       | 4                       |

Total cicli executie = 20.

31. a) **Fals!** În caz de hit, pe baza compararii tag-urilor, se citește si data respectiva => acces simultan la tag si date.

b) **Fals!** Fiind hit tag-ul nu mai are sens sa-si modifice valoarea.

c) **Fals!** Datorita interferentelor alternative, rata de hit este egala cu 0.

32. i5: LOAD R<sub>1</sub>, (R<sub>1</sub>+0)

33. **Nu!** O regenerare transparentă (durează 250ns) trebuie să “încapă” între 2 accese la DRAM ale microprocesorului. Cum un ciclu extern al procesorului durează 3 tacte (150ns) regenerarea transparentă este imposibilă la frecvența maximă a procesorului.

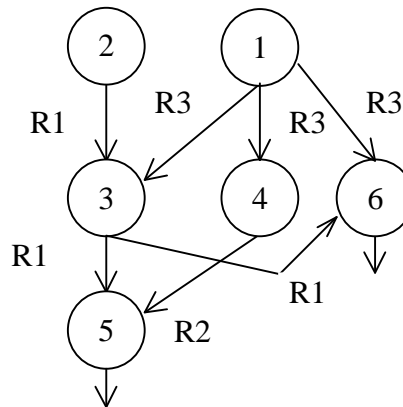
a. ALU: (R9) + 06; MEM: scriere R5 la adresa (R9) + 06; WB: nimic

b. ALU: (R8) + F3; MEM: citire de la adresa (R8) + F3; WB: scriere în R7;

c. ALU: (R7) + (R8); MEM: nimic; WB: scriere în R5.

35.

a. 1 - 2 - nop - 3 - 4 - nop - 5 - 6 ⇒ 8 cicluri.



b. 1 - 2 - 4 - 3 - 6 - 5 ⇒ 6 cicluri.

36.

a. Nr.tacte /s consumate pentru interogare mouse:  $30 \times 100 = 3000$  tacte/ s

$$f = \frac{3000}{50 \times 10^6} = 0,006\%$$

$$b. \frac{\text{Nr.interogari}}{s} = \frac{50 \frac{\text{ko}}{s}}{2 \frac{\text{o}}{\text{acces interogare}}} = 25 \times 2^{10} \frac{\text{acces interogare}}{s}$$

Nr. tacte necesar pentru Nr.interogari/ s =  $25 \times 2^{10} \times 100$  tacte

$$f = \frac{25,6 \times 10^5}{50 \times 10^6} = 5,12\%$$

$$c. \frac{\text{Nr.interogari}}{s} = \frac{2 \frac{\text{Mo}}{s}}{4 \frac{o}{\text{acces interogare}}} = 0,5 \times 2^{20} \frac{\text{acces interogare}}{s}$$

Nr. tacte necesar pentru Nr.interogari/ s =  $0,5 \times 2^{20} \times 100$  tacte

$$f = \frac{0,5 \times 2^{20} \times 100}{50 \times 10^6} > 100\%$$

În cazul hard-disc-ului este imposibila comunicatia dintre procesor si periferic prin interogare. (Într-o secunda procesorul realizeaza  $50 \times 10^6$  tacte, iar pentru un transfer cu o rata de 2 Mo/ s sunt necesare într-o secunda  $50 \times 2^{20}$  tacte, imposibil).

37.a)

- Minimizarea numarului de interschimbari dintre cache-ul principal si victim cache.
- Mecanismul de predictie bazat pe istoria anterioara a blocurilor existente în cache-ul principal (conflictual) si cel din victim cache, stabileste daca blocul adus din memoria principala va fi plasat fie în cache-ul principal fie în selective victim cache. Algoritmul obiectiv este de a plasa blocurile, care sunt mai probabil a fi referite din nou, în cache-ul principal si celelalte în selective victim cache. Predictia este de asemenea folosita în cazul unui miss în cache-ul principal pentru a determina daca este necesara o schimbare a blocurilor conflictuale.

39. a) Vezi capitolul 4.3.1. al prezentei lucrari.

c) Val. minima = 1 (procesor scalar)

Val. maxima = IRmax

40. a – cazul memoriei **mapate direct** cu 4 locatii.

Se acceseaza pe rând locatiile:

| Locația<br>accesată | 0       | 8       | 0       | 6       | 8       | 10      | 8      |
|---------------------|---------|---------|---------|---------|---------|---------|--------|
|                     | Tag     | Tag     | Tag     | Tag     | Tag     | Tag     | Tag    |
| 0                   | 0(miss) | 2(miss) | 0(miss) | 0       | 2(miss) | 2       | 2(hit) |
| 1                   | X       | X       | X       | X       | X       | X       | X      |
| 2                   | X       | X       | X       | 1(miss) | 1       | 2(miss) | 2      |
| 3                   | X       | X       | X       | X       | X       | X       | X      |

Rezultă în cazul memoriei mapate direct un singur acces cu hit  $R_{\text{miss}} = 6 / 7 = 85.71\%$

b – cazul memoriilor semiasociative cu 2 seturi a câte 2 cuvinte.

**Întrucât toate adresele sunt pare se accesează doar blocurile din setul 0.**

| Locația accesată | Setul 0 |    |                       |
|------------------|---------|----|-----------------------|
| 0                | 0       | X  | Miss                  |
| 8                | 0       | 8  | Miss                  |
| 0                | 0       | 8  | Hit                   |
| 6                | 0       | 6  | Miss. Se evacuează 8. |
| 8                | 8       | 6  | Miss. Se evacuează 0. |
| 10               | 8       | 10 | Miss. Se evacuează 6. |
| 8                | 8       | 10 | Hit.                  |

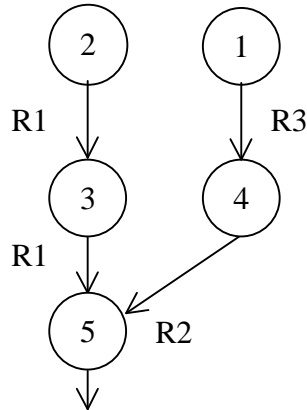
În cazul memoriei two-way asociative sunt 2 accese cu hit  $R_{\text{miss}} = 5 / 7 = 71.42\%$

c – cazul memoriilor complet asociative.

| Locația<br>accesată | 0       | 8       | 0      | 6       | 8      | 10       | 8      |
|---------------------|---------|---------|--------|---------|--------|----------|--------|
|                     | Tag     | Tag     | Tag    | Tag     | Tag    | Tag      | Tag    |
| 0                   | 0(miss) | 0       | 0(hit) | 0       | 0      | 0        | 0      |
| 1                   | X       | 8(miss) | 8      | 8       | 8(hit) | 8        | 8(hit) |
| 2                   | X       | X       | X      | 6(miss) | 6      | 6        | 6      |
| 3                   | X       | X       | X      | X       | X      | 10(miss) | 10     |

În cazul memoriei complet asociative sunt 3 accese cu hit  $R_{\text{miss}} = 4 / 7 = 57.14\%$

41. a. 1 - 2 - nop - 3 - 4 - nop - 5  $\Rightarrow$  7 cicli.



b. 1 - 2 - 4 - 3 - nop - 5  $\Rightarrow$  6 cicli.

42. În 1 s se transfera  $25 \times 10^4$  biti.

În x s se transfera 1 octet.

Rezulta  $x = 8 / (25 \times 10^4) = 32 \mu s$

Fie  $t_r$  = timpul disponibil dintre 2 transferuri succesive de octeti.

$t_1$  = timpul scurs între aparitia întreruperii și intrarea în rutina de tratare;

$t_1 = 2 \mu s$ ;

$t_2$  = timpul în care este tratată rutina de întrerupere;  $t_2 = 10 \mu s$ ;

$t_r = x - t_1 - t_2 = (32 - 2 - 10) \mu s \Rightarrow t_r = 20 \mu s$ .

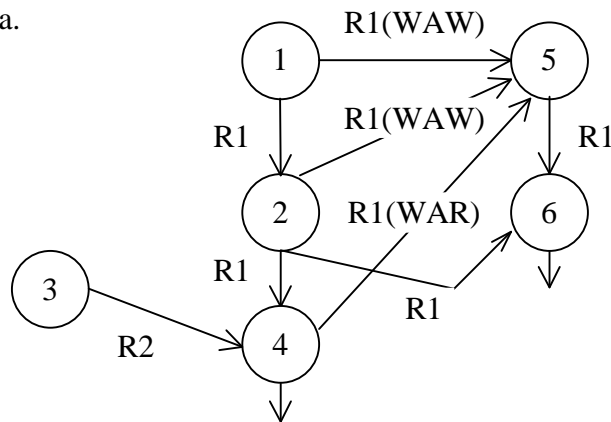
43. a.  $\frac{(1 - 0,11) \times 8,5 + 0,11 \times 6 \times 3}{8,5} = 1,1229 \Rightarrow$  diminuare a performanței cu

$\approx 13\%$ .

b. **Avantajul** implementării unei pagini de capacitate «mare» într-un sistem de memorie virtuală constă în principiul localizării acceselor, determinând o optimizare a acceselor la disc (se reduce numărul acestora).

**Dezavantajul** îl reprezintă aducerea inutilă de informație de pe disc (pierdere de timp), care trebuie apoi evacuată în cazul unei erori de tipul *PageFault*. Dimensiunea paginii se alege în urma unor simulări laborioase.

44. a.



1 - nop - 2 - 3 - nop - 4 - 5 - nop - 6  $\Rightarrow$  9 cicli executie.

b.

1:  $R1 \leftarrow (R11) + (R12)$

5:  $R1' \leftarrow (R14) + (R15)$

3:  $R2 \leftarrow (R3) + 4$

2:  $R1 \leftarrow (R1) + (R13)$

6:  $R1' \leftarrow (R1') + (R16)$

4:  $R2 \leftarrow (R1) + (R2)$

1 - 5 - 3 - 2 - 6 - 4  $\Rightarrow$  6 cicli executie

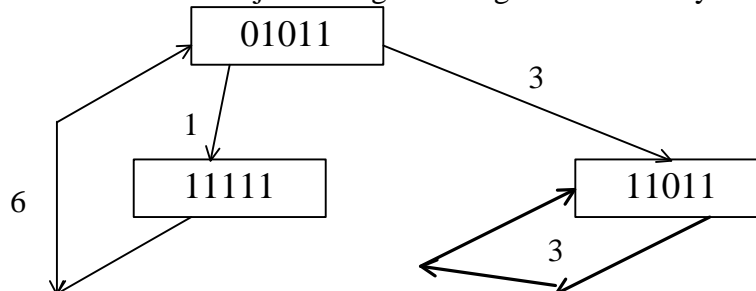
45. Daca alegem strategia Greedy obtinem rata de procesare

$$IR = \frac{2}{7} \text{ instr/ciclu}$$

Daca alegem strategia non - Greedy rata de procesare obtinuta este:

$$IR = \frac{1}{3} \text{ instr/ciclu}$$

În acest caz e mai avantajos sa alegem strategia non - Greedy.



**46.** Algoritmul lui Tomasulo permite anularea hazardurilor WAR si WAW printr-un mecanism hardware de redenumire a registrilor, favorizând executia multipla si *Out of Order* a instructiunilor. Mecanismul de «forwarding» implementat prin arhitectura Tomasulo (statii de rezervare) determina reducerea semnificativa a presiunii la «citire» asupra setului de registri logici, înlăturând o mare parte din dependentele RAW dintre instructiuni [33].

În cazul unei arhitecturi TTA, numarul de registri generali poate fi redus semnificativ datorita faptului ca trebuie stocate mai putine date temporare, acestea circulând direct între unitatile de executie (FU - unitati functionale), nemaifiind necesara memorarea lor în registri. «Forwarding-ul» datelor este realizat software prin program, spre deosebire de procesoarele superscalare care realizeaza acest proces prin hardware folosind algoritmul lui Tomasulo [33].

**47.** O instructiune de tip RETURN este dificil de predictionat printr-un predictor hardware datorita **fenomenului de interferenta a salturilor**. Acesta apare în cazul unei predictii incorecte datorate exclusiv adresei de salt incorecte din tabela de predictii, care a fost modificata de catre un alt salt anterior. Instructiunile de tip RETURN reprezinta salturi care-si modifica dinamic adresa tinta, favorizând dese aparitii ale fenomenului de interferenta. Analog se întâmpla si în cazul salturilor în mod de adresare indirect [33].

Noutatea «principiala» a predictoarelor corelate pe doua nivele consta în faptul ca predictia unei instructiuni tine cont de predictia ultimelor  $n$  instructiuni de salt anterioare; se foloseste un registru de predictie (registru binar de deplasare) care memoreaza istoria ultimelor  $n$  instructiuni de salt. Valoarea acestui registru concatenata cu cei mai putini semnificativi biti ai PC-ului instructiunii de salt curente realizeaza adresarea cuvântului de predictie din tabela de predictie [33, 45].

**50.** a. Sumatorul “sum 2” este activat de o instructiune de branch, pentru calculul adresei de salt.

b. Nivelul **WR** este mai prioritar datorita hazardurilor RAW. Operatia de citire ar putea avea nevoie de un registru în care nu s-a înscris încă rezultatul final, rezulta prioritatea scrierii fata de citire.

c. În cazul unei instructiuni de tip LOAD, unitatea ALU are rol de calcul adresa.

d. În latch-ul EX/MEM se memoreaza valoarea (R7+05).

**52.** Se citesc caractere de la intrare pâna se întâlnește conditia de iesire, si se salveaza acestea în stiva. Conditia de iesire o constituie tastarea caracterului '0'. La întâlnirea sa se afiseaza caracterul curent ('0' – primul caracter afisat) si se va apela functia de afisare. În cadrul acestei functii vom prelua din stiva caracterele memorate si le vom tipari.

**Obs.** E necesar un parametru pentru contorizarea caracterelor scrise în stiva.

Programul modificat pentru a nu afisa si caracterul '0', difera prin faptul ca la întâlnirea conditiei de iesire se apeleaza direct functia de afisare.

**53.** Este esential sa calculam  $cmmdc$  (cel mai mare divizor comun) dintre cele doua numere,  $cmmmc$  (cel mai mic multiplu comun) putându-se calcula apoi din formula:

$$cmmdc(a,b) \cdot cmmmc(a,b) = a \cdot b$$

Pentru calcularea  $cmmdc(a,b)$  folosim recursivitatea:

$$Cmmdc(a,b) = \begin{cases} a, & \text{daca } a = b \\ Cmmdc(a,b-a), & \text{daca } a < b \\ Cmmdc(b,a-b), & \text{daca } a > b \end{cases}$$

Se apeleaza recursiv functia având ca noi parametri minimul dintre cele doua numere si modulul diferentei dintre cele doua valori, pâna la întâlnirea conditiei de iesire ( $a = b$ ), în a (si b) aflându-se chiar cel mai mare divizor comun.

\*\*\*\*\*Sursa în limbaj de asamblare MIPS \*\*\*\*\*  
#calculeaza recursiv  $cmmdc$  si  $cmmmc$  dintre  $n$  numere naturale (de preferat  $< 1000$ )

```

                .data
n:              .space 4
date:          .space 100
mesaj:         .asciiz "Introduceti nr. de numere:"
mesaj1:        .asciiz "Cel mai mic multiplu comun este:"
mesaj2:        .asciiz "\nCel mai mare divizor comun este:"

```



```

        .text
b      program      #sare peste proceduri

#functie 2 intrari, 2 iesiri – cmmdc local (de 2 numere)
mare_mic:
    add    $sp,$sp,-4
    lw     $t0,0($sp)    #citeste numar 1
    add    $sp,$sp,-4
    lw     $t1,0($sp)    #citeste numar 2
    mul    $t3,$t0,$t1    #cel mai mic temporar
    bge    $t0,$t1,cont   #daca t0>=t1 sare
    add    $t2,$0,$t1     #daca nu interschimba
    add    $t1,$0,$t0
    add    $t0,$0,$t2

cont:
    div    $t0,$t1        #imparte numerele
    add    $t0,$0,$t1     #impartitor devine deimpartit
    mfhi   $t1            #rest devine impartitor
    bnez   $t1,cont       #daca rest nu=0 reia
    sw     $t0,0($sp)     #pune mare pe stiva
    add    $sp,$sp,4
    div    $t2,$t3,$t0
    sw     $t2,0($sp)     #pune mic pe stiva
    add    $sp,$sp,4
    jr     $31            #revine din functie

#sfarsit functie

#functia recursiva
recursiva:
    sw     $31,0($sp)     #pune adresa de revenire pe stiva
    add    $sp,$sp,4
    lw     $t0,-8($sp)    #citeste stanga de pe stiva si pune in t0
    lw     $t1,-12($sp)   #citeste dreapta de pe stiva si pune in t1
    sub    $t2,$t1,$t0    #face diferenta intre dreapta si stanga si pune
                        #in t2
    bnez   $t2,loc1       #daca nu stanga=dreapta
    lw     $t6,date($t0)  #daca stanga=dreapta citeste din memorie si
                        # pune pe stiva
    lw     $31,-4($sp)    #citeste de pe stiva adresa de revenire
    add    $sp,$sp,-12    #reface stiva folosita de aceasta functie
    sw     $t6,0($sp)     #scrie primul rezultat pe stiva

```

```

        add    $sp,$sp,4
        sw     $t6,0($sp)    #scrie al doilea rezultat pe stiva
        add    $sp,$sp,4
        jr     $31           #revine din functie
loc1:
        bgt    $t2,4,loc2    #daca diferenta>4 adica mai mult de 2 numere
        lw     $t6,date($t0)  #citeste primul numar
        lw     $t7,date($t1)  #citeste al doilea numar
        sw     $t6,0($sp)    #pune numar 1 pe stiva
        add    $sp,$sp,4
        sw     $t7,0($sp)    #pune numar 2 pe stiva
        add    $sp,$sp,4
        jal    mare_mic      #apeleaza functie
        add    $sp,$sp,-4     #citeste rezultat 1
        lw     $t6,0($sp)
        add    $sp,$sp,-4     #citeste rezultat 2
        lw     $t7,0($sp)
        lw     $31,-4($sp)    #citeste de pe stiva adresa de revenire
        add    $sp,$sp,-12    #reface stiva folosita de aceasta functie
        sw     $t7,0($sp)    #scrie primul rezultat pe stiva
        add    $sp,$sp,4
        sw     $t6,0($sp)    #scrie al doilea rezultat pe stiva
        add    $sp,$sp,4
        jr     $31           #revine din functie
loc2:
        #aici se apeleaza recursiv
        lw     $t0,-8($sp)    #citeste stanga
        lw     $t1,-12($sp)   #citeste dreapta
        add    $t2,$t1,$t0     #imparte (st+dr)/2
        add    $t3,$0,2
        div    $t4,$t2,$t3
        div    $t4,$t4,4       #face mod 4
        mul    $t4,$t4,4
        sw     $t4,0($sp)     #pune (st+dr)/2 pe stiva
        add    $sp,$sp,4
        sw     $t0,0($sp)     #pune st pe stiva
        add    $sp,$sp,4
        jal    recursiva      #apeleaza recursiva
        lw     $t0,-16($sp)   #citeste stanga
        lw     $t1,-20($sp)   #citeste dreapta
        sw     $t1,0($sp)     #pune dr pe stiva
        add    $sp,$sp,4

```

```

add    $t2,$t1,$t0    #imparte (st+dr)/2+4
add    $t3,$0,2
div    $t4,$t2,$t3
div    $t4,$t4,4      #face mod 4
mul    $t4,$t4,4
add    $t4,$t4,4
sw     $t4,0($sp)     #pune (st+dr)/2+4 pe stiva
add    $sp,$sp,4
jal    recursiva      #apeleaza recursiva
lw     $t0,-8($sp)    #citeste mare 1
lw     $t1,-16($sp)   #citeste mare 2
sw     $t0,0($sp)     #scrie mare 1 pe stiva
add    $sp,$sp,4
sw     $t1,0($sp)     #scrie mare 2 pe stiva
add    $sp,$sp,4
jal    mare_mic       #apeleaza mare_mic
lw     $t0,-12($sp)   #citeste mic 1
lw     $t1,-20($sp)   #citeste mic 2
sw     $t0,0($sp)     #pune mic 1 pe stiva
add    $sp,$sp,4
sw     $t1,0($sp)     #pune mic 2 pe stiva
add    $sp,$sp,4
jal    mare_mic       #apeleaza mare_mic
lw     $t0,-16($sp)   #t0=mare final
lw     $t1,-8($sp)    #impartitor
lw     $t2,-20($sp)   #pt immultit
lw     $t3,-28($sp)   #pt immultit
mul    $t4,$t3,$t2    #mic*mic/mare
div    $t5,$t4,$t1    #t5=mic final
add    $sp,$sp,-32    #reface stiva
lw     $31,-4($sp)    #citeste de pe stiva adresa de revenire
add    $sp,$sp,-12    #reface stiva folosita de aceasta functie
sw     $t0,0($sp)     #scrie primul rezultat pe stiva
add    $sp,$sp,4
sw     $t5,0($sp)     #scrie al doilea rezultat pe stiva
add    $sp,$sp,4
jr     $31            #revine din functie
#sfarsit functie
program:
li     $v0,4           #program principal
la     $a0,mesaj       #afisaza mesaj

```

```

        syscall
        li      $v0,5
        syscall
        add     $v0,$v0,-1    #n este defapt n-1
        mul     $a3,$v0,4     #a3=n*4
        sw      $v0,n($0)     #pune n in memorie
        add     $a2,$0,$0     #a2=contor
loc:
        li      $v0,5         #citeste cate un numar
        syscall
        sw      $v0,date($a2) #scrie in memorie
        add     $a2,$a2,4     #incrementeaza contor
        bge     $a3,$a2,loc   #daca nu s-au citit destule date se reia
        sw      $a3,0($sp)    #pune n pe stiva
        add     $sp,$sp,4
        sw      $0,0($sp)     #pune 0 pe stiva
        add     $sp,$sp,4
        jal     recursiva     #apeleaza functia
        li      $v0,4         #afisaza mesaj
        la      $a0,mesaj1
        syscall
        add     $sp,$sp,-4    #citeste rezultat1
        lw      $a0,0($sp)
        li      $v0,1         #afisaza rezultat
        syscall
        li      $v0,4         #afisaza mesaj
        la      $a0,mesaj2
        syscall
        add     $sp,$sp,-4    #citeste rezultat2
        lw      $a0,0($sp)
        li      $v0,1         #afisaza rezultat
        syscall
        li      $v0,10        #terminare program
        syscall

```

\*\*\*\*\*

**54.** Semnificatia celor 3 tije este urmatoarea:

- A – sursa
- B – destinatie
- C – manevra

Problema pentru  $n$  discuri se rezolva usor daca putem rezolva pentru  $(n-1)$  discuri, deoarece rezolvând-o pe aceasta, vom putea muta primele  $(n-1)$  discuri de pe tija A(sursa) pe C(manevra), apoi discul  $n$  (cu diametrul cel mai mare) de pe tija A(sursa) pe tija B(destinatie) si din nou cele  $(n-1)$  discuri de pe tija C(manevra) pe tija B(destinatie).

Conditia de iesire din subrutina o constituie problema transferarii unui singur disc ( $n=1$ ) de pe sursa pe destinatie. Pasii algoritmului sunt:

Se citesc de la tastatura numarul de discuri ( $n$ ), si identificatorii (caractere) tijelor sursa, destinatie si manevra si se apeleaza subrutina *hanoi* având ca parametrii efectivi cele patru valori citite anterior. În cadrul functiei *hanoi* se executa:

Se salveaza în stiva adresa de revenire si cadrul de stiva. Se testeaza daca se îndeplineste conditia de iesire din subrutina.

Daca da, se afiseaza transferul (tija sursa si tija destinatie), se reface continutul registrilor PC si fp din stiva si se actualizeaza SP. Se executa instructiunea de la adresa data de PC.

Daca nu, se executa secventa:

- Se salveaza în stiva registrii corespunzatori parametrilor (tijelor).
- Se actualizeaza numarul de discuri,  $n \leftarrow (n-1)$  si rolul fiecărei tije (noua destinatie va fi tija C, tija A va fi sursa iar tija B va fi manevra). Se reapeleaza *hanoi*. [Se muta cele  $(n-1)$  discuri de pe tija sursa pe tija de manevra].
- Se refac parametrii din stiva (tije). Se afiseaza transferul [cel de-al  $n$  - lea disc de pe tija sursa(A) pe tija destinatie(B)].
- Se salveaza în stiva registrii corespunzatori parametrilor (tijelor).
- Se actualizeaza numarul de discuri,  $n \leftarrow (n-1)$  si rolul fiecărei tije (noua destinatie va fi tija B, tija C este sursa iar tija A va fi manevra). Se reapeleaza *hanoi*. [Se muta cele  $(n-1)$  discuri de pe tija de manevra pe cea destinatie].

**55.** Se modifica programul de calcul al factorialului unui numar, prezentat în limbaj de asamblare MIPS (vezi lucrarea “*Investigatii Arhitecturale Utilizând Simulatorul SPIM*”), calculând în paralel cu factorialul si aranjamentele, folosind formulele:

$$C_n^k = \frac{n!}{(n-k)!k!}; \quad A_n^k = \frac{n!}{(n-k)!}$$

Parametrii  $n$  si  $k$  se citesc de la tastatura si sunt salvati în stiva. Se intra în subrutina unde se executa:

Se verifica daca  $k = 1$  (conditia de iesire din subrutina).

Daca **da**, se seteaza în registrii rezultat valoarea 1, punctul de plecare în calcularea produselor  $1 \times 2 \times \dots \times k$  si  $n \times (n-1) \times \dots \times [n-(k-1)]$ . Se reface continutul registrilor PC si fp din stiva si se actualizeaza SP. Se executa instructiunea de la adresa data de PC.

Daca **nu**, se executa secventa:

- Se actualizeaza parametrii  $n \leftarrow (n-1)$  si  $k \leftarrow (k-1)$ . Se reapeleaza subrutina.
- Se preiau din stiva termenii salvati si se înmultesc cu rezultatele calculate pâna la acest pas.
- Se reface continutul registrilor PC si fp din stiva si se actualizeaza SP. Se executa instructiunea de la adresa data de PC.

În încheierea programului se calculeaza combinarile cu formula raportului dintre aranjamente si permutari si afiseaza valorile aranjamentelor, permutarilor si combinarilor.

**57.** Se parcurge sirul numerelor naturale din doi în doi (ne intereseaza doar numerele impare) începând cu numarul 3 (primul numar impar prim) si se determina daca este prim sau nu (vezi lucrarea "*Investigatii Arhitecturale Utilizând Simulatorul SPIM*"). În cazul în care numarul este prim (fie acesta  $k$ ) se verifica daca si urmatorul numar impar ( $k+2$ ) este prim si daca **da** se afiseaza perechea de numere. Se testeaza daca am ajuns la numarul de perechi de numere prime cerut si daca nu se continua algoritmul cu numarul prim mai mic. În momentul în care un numar se dovedeste a nu fi prim se trece la urmatorul numar impar.

### **58. Implementare în limbaj de asamblare DLX**

/\*\*\*\*\* Programul principal \*\*\*\*\*/

```

.data
Prompt:      .asciiz "Introduceti n:"
PrintfFormat1: .asciiz "Primul numar din suma %d\n\n"
               .align 2
PrintfFormat2: .asciiz "Al doilea numar din suma %d\n\n"
               .align 2
PrintfFormat3: .asciiz "Introduceti va rog un numar par!!\n"
               .align 2
PrintfFormat4: .asciiz "Introduceti va rog un numar mai mare de
                   6!!\n"
```

```
PrintfPar1:      .align 2
                  .word PrintfFormat1
PrintfValue1:    .space 8
PrintfPar2:      .word PrintfFormat2
PrintfValue2:    .space 8
PrintfPar3:      .word PrintfFormat3
PrintfPar4:      .word PrintfFormat4

                  .text
                  .global main
main:
                  addi r1,r0,Prompt
                  jal InputUnsigned
                  sle r2,r1,6
                  bnez r2,eticheta1
                  add r25,r1,r0
                  addi r15,r0,2
                  div r3,r1,r15
                  mult r4,r3,r15
                  sub r2,r1,r4
                  bnez r2,eticheta2
                  j eticheta3
eticheta1:
                  add r14,r0,PrintfPar4
                  trap 5
                  j main
eticheta2:
                  add r14,r0,PrintfPar3
                  trap 5
                  j main
eticheta3:
                  addi r13,r0,1
eticheta4:
                  sgt r11,r13,r3
                  bnez r11,end
vprim:
                  addi r24,r24,1
                  sub r12,r13,1
                  bnez r12,vp1
                  j r13prim
vp1:
```

```

                                addi r20,r0,2
eticheta5:
                                sub r20,r20,r13
                                beqz r20,nr_e_prim
                                add r20,r20,r13
                                div r21,r13,r20
                                mult r22,r21,r20
                                sub r23,r13,r22
                                beqz r23,nueprim
                                addi r20,r20,1
                                j eticheta5
nr_e_prim:
                                sub r24,r24,2
                                beqz r24,afisare
                                add r24,r24,2
r13prim:
                                add r16,r13,r0
                                sub r13,r25,r13
                                j vprim
afisare:
                                sw PrintfValue1,r16
                                addi r14,r0,PrintfPar1
                                trap 5
                                sw PrintfValue2,r13
                                addi r14,r0,PrintfPar2
                                trap 5
nueprim:
                                add r24,r0,r0
                                add r13,r16,2
                                j eticheta4
end:
                                trap 0

```

```

/***** Rutina Input.s *****/
;***** Citeste un numar întreg pozitiv *****/
;asteapta în registrul R1 adresa unui sir de caractere terminat cu NULL.
;returneaza valuarea citita în R1
;modifica continutul registrilor R1,R13,R14

```



```

.data
ReadBuffer: .space 80
ReadPar:    .word 0,ReadBuffer,80
PrintfPar:  .space 4
SaveR2:     .space 4
SaveR3:     .space 4
SaveR4:     .space 4
SaveR5:     .space 4
.text
.global InputUnsigned
InputUnsigned:
;*** salveaza continutul registrilor
sw      SaveR2,r2
sw      SaveR3,r3
sw      SaveR4,r4
sw      SaveR5,r5
sw      PrintfPar,r1
addi    r14,r0,PrintfPar
trap    5
addi    r14,r0,ReadPar
trap    3
;*** determina valoarea citita
addi    r2,r0,ReadBuffer
addi    r1,r0,0
addi    r4,r0,10      ;Decimal system
Loop:   ;*** reads digits to end of line
lbu     r3,0(r2)
seqi    r5,r3,10      ;LF -> Exit
bnez    r5,Finish
subi    r3,r3,48      ;`0`
multu   r1,r1,r4;Shift decimal
add     r1,r1,r3
addi    r2,r2,1 ;increment pointer
j       Loop
Finish: ;*** restore old register contents
lw      r2,SaveR2
lw      r3,SaveR3
lw      r4,SaveR4
lw      r5,SaveR5
jr      r31          ; Return

```

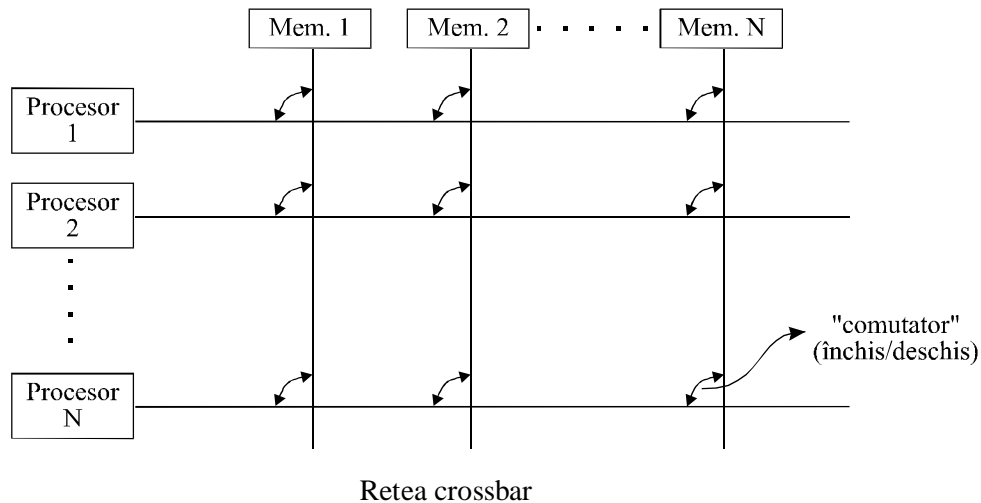
```

/*****

```

**61. RIC cu lărgime de bandă maximă** - rețea de tip crossbar, permite implementarea oricărei bijecții procesoare-module memorie;

**SMM** cu interconectare “**crossbar**” este o rețea dinamică deținând complexitatea cea mai ridicată dintre arhitecturile de SMM; în schimb conflictele procesoarelor la resursele de memorie comună partajată sunt minime [VinFlor00]. Comunicatia între orice pereche procesor – memorie este întârziată în nodul de conexiune aferent. De remarcat că pot avea loc simultan până la  $N$  accese ale procesoarelor la memorie, în ipoteza în care nu există două procesoare care să acceseze același modul de memorie. Pentru evitarea conflictelor de acest gen, se încearcă atunci când este posibil “împrăstieri” favorabile ale informației în modulele de memorie globale. De exemplu în aplicațiile pe vectori, dacă aceștia au pasul 1 atunci scalarii succesivi sunt situați în blocuri succesive de memorie, rezultând minimizări ale conflictelor.



Deși cea mai performantă, arhitectura devine practic greu realizabilă pentru un număr  $N$  ridicat de procesoare, din cauza costurilor ridicate ( $N^2$  noduri).

**RIC cu lărgime de bandă minimă** - rețea unibus, un singur procesor master la un moment dat.

**SMM pe bus comun** constituie o rețea statică caracterizată de faptul că RIC este un simplu bus comun partajat în timp de către  $\mu$ procesoare (UMA) [VinFlor00]. Este cea mai simplă arhitectură, dar conflictele potențiale ale procesoarelor (masterilor) pe busul comun, pot fi ridicate. Desigur, există un

singur master activ pe bus la un moment dat. Busul comun si memoria globala (slave) sunt partajate în timp de catre masteri. Resursele locale ale masterilor - memorii cache locale - au rolul de a diminua traficul pe busul comun. Accesul pe bus se face prin intermediul unui arbitru de prioritati, centralizat sau distribuit.

Arhitectura implica dificultati tehnologice legate de numarul maxim de masteri cuplabili (în practica pâna la 32), reflexii si diafonii ale semnalelor pe bus. Cum capacitatile si inductantele parazite cresc proportional cu lungimea busului, rezulta ca acesta trebuie sa fie relativ scurt. Exista arhitecturi standardizate de SMM pe bus comun (VME – dezvoltat de Motorola pe  $\mu$ p MC680X0, MULTIBUS – Intel pentru I - 80X86).

- 62.**
1. Operatie “*write miss*” pe busul comun.
  2. Citire bloc de la unul din cache-urile care îl detine (“*snooping*”).
  3. Toate procesoarele care au detinut respectivul bloc în cache-uri, îl trec în starea “*invalid*” ( $V=0$ ).
  4. Procesorul care l-a citit în pasul 2, îl scrie si îl pune în cache în starea “*exclusiv*”.

**64.**

**Timp mediu de acces instructiune =  $T_{\text{acces\_cache}} \cdot R_{\text{hit}} + T_{\text{acces\_DRAM}} \cdot (1 - R_{\text{hit}})$**

a)  $T_{\text{acces\_DRAM}} = 1 + 4 + \text{dim\_bloc} \cdot 1$

| $T_{\text{acces\_DRAM}}$ | $\text{dim\_bloc}$ | $\text{rata\_miss}$ | $\text{Rezultat\_partial}$ | $T_{\text{acces\_cache}}$ | $\text{rata\_hit}$ | $\text{Rezultat\_final}$ |
|--------------------------|--------------------|---------------------|----------------------------|---------------------------|--------------------|--------------------------|
| 6                        | 1                  | 4,5                 | 0.2700                     | 1                         | 0.955              | 1.225                    |
| 9                        | 4                  | 2,4                 | 0.2160                     | 1                         | 0.976              | <b>1.192</b>             |
| 13                       | 8                  | 1,6                 | 0.2080                     | 1                         | 0.984              | <b>1.192</b>             |
| 21                       | 16                 | 1,0                 | 0.2100                     | 1                         | 0.99               | 1.2                      |
| 37                       | 32                 | 0,75                | 0.2775                     | 1                         | 0.9925             | 1.27                     |

b)  $T_{\text{acces\_DRAM}} = 2 + 1 + 4 + \text{dim\_bloc} \cdot 1$

| $T_{\text{acces\_DRAM}}$ | $\text{Dim\_bloc}$ | $\text{rata\_miss}$ | $\text{Rezultat\_partial}$ | $T_{\text{acces\_cache}}$ | $\text{rata\_hit}$ | $\text{Rezultat\_final}$ |
|--------------------------|--------------------|---------------------|----------------------------|---------------------------|--------------------|--------------------------|
| 8                        | 1                  | 4,5                 | 0.3600                     | 1                         | 0.955              | 1.315                    |
| 11                       | 4                  | 2,4                 | 0.2640                     | 1                         | 0.976              | 1.24                     |
| 15                       | 8                  | 1,6                 | 0.2400                     | 1                         | 0.984              | 1.224                    |
| 23                       | 16                 | 1,0                 | 0.2300                     | 1                         | 0.99               | <b>1.22</b>              |
| 39                       | 32                 | 0,75                | 0.2925                     | 1                         | 0.9925             | 1.285                    |

c)  $T_{acces\_DRAM} = 1 + 4 + dim\_bloc * 0.5$  (se pot aduce doua blocuri de 32 de octeti simultan)

| T_acces<br>_DRAM | Dim_<br>bloc | rata_<br>miss | Rezultat_<br>partial | T_acces<br>_cache | rata_hit | Rezultat<br>_final |
|------------------|--------------|---------------|----------------------|-------------------|----------|--------------------|
| 5.5              | 1            | 4,5           | 0.2475               | 1                 | 0.955    | 1.2025             |
| 7                | 4            | 2,4           | 0.1680               | 1                 | 0.976    | 1.144              |
| 9                | 8            | 1,6           | 0.1440               | 1                 | 0.984    | 1.128              |
| 13               | 16           | 1,0           | 0.1300               | 1                 | 0.99     | <b>1.12</b>        |
| 21               | 32           | 0,75          | 0.1575               | 1                 | 0.9925   | 1.15               |

75.

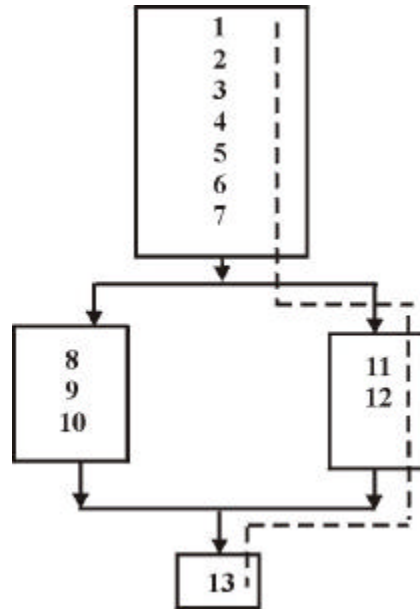
Executia trace-ului neoptimizat

| DECODIFICARE |    |    |    | EXECUTIE |      |     |    |     | CICLU |
|--------------|----|----|----|----------|------|-----|----|-----|-------|
| I0           | I1 | I2 | I3 | ALU1     | ALU2 | SHF | LS | BRN |       |
| 1            | 2  | 3  | 4  |          |      |     |    |     | 1     |
|              |    |    |    | 1        | 2    |     |    |     | 2     |
| 5            | 6  | 7  | 8  | 3        |      |     | 4  |     | 3     |
|              |    |    |    |          |      |     |    |     | 4     |
|              |    |    |    |          |      |     | 5  |     | 5     |
|              |    |    |    |          |      |     |    |     | 6     |
|              |    |    |    | 6        |      |     |    |     | 7     |
| 9            | 10 | 11 | 12 |          |      |     |    | 7   | 8     |
| 13           | X  | X  | X  | 12       |      |     | 11 |     | 9     |
|              |    |    |    | 13       |      |     |    |     | 10    |

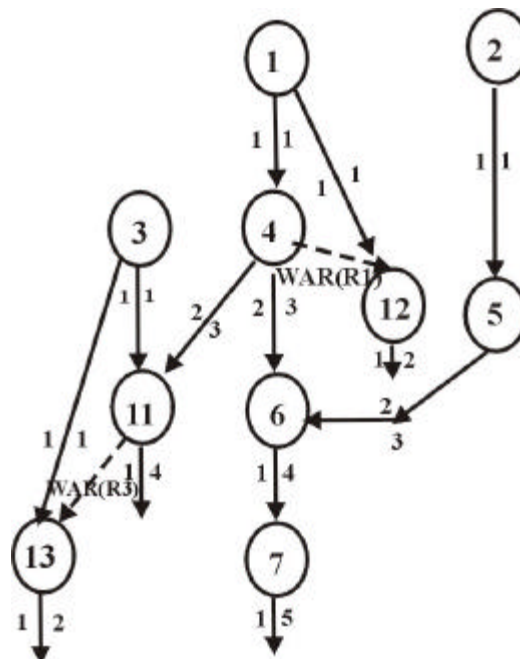
Datorita paralelismului limitat al acestei secvente se obtine o rata medie de executie de doar **1,3** instr. / ciclu. Graful de control al acestei secvente este complus din 3 basic block-uri ca în figura 1.

De remarcat ca decodificarea unui nou grup de instructiuni poate avea loc si în tactul imediat urmator lansarii în executie a tuturor instructiunilor anterior decodificate.

Graful de control aferent secvenței de optimizat



Graful dependentelor pentru trace-ul considerat



Ordinea de execuție în urma optimizării

| Nod / Prioritate |
|------------------|
| 7/5; 13/2        |
| 6/4; 11/4; 12/2  |
| 5/3              |
| 4/3; 3/1         |
| 1/1; 2/1         |

Execuția secvenței optimizate se face doar în 7 cicli și datorită hazardurilor structurale la unitatea Load/Store ( $IR = 13/7 \approx 1,86$  instr./ciclu):

| EXECUTIE |      |     |    |     | CICLU |
|----------|------|-----|----|-----|-------|
| ALU1     | ALU2 | SHF | LS | BRN |       |
| 1        | 2    |     |    |     | 1     |
| 3        |      |     | 4  |     | 2     |
|          |      |     |    |     | 3     |
|          |      |     | 5  |     | 4     |
|          |      |     |    |     | 5     |
| 6        | 12   |     | 11 |     | 6     |
| 13       |      |     |    | 7   | 7     |

Întrucât basic block-ul care conține instrucțiunile 11 și 12 se execută preponderent se obține un avantaj prin migrarea instrucțiunilor 11, 12, 13 în basic block-ul anterior fiind necesare însă coduri compensatoare. Rezultatul obținut este optim deoarece unitățile ALU sunt pline în tactul 6 și în plus instrucțiunea 13 nu poate percola dincolo de instrucțiunea 11 (s-ar altera contextul programului). Codul reorganizat arată astfel:

|     |      |                  |
|-----|------|------------------|
| 1:  | ADD  | R1, R0, base_a   |
| 2:  | ADD  | R2, R0, base_b * |
| 3:  | ADD  | R3, R0, base_c   |
| 4:  | LD   | R4, (R1) *       |
| 5:  | LD   | R5, (R2) *       |
| 6:  | SGT  | R7, R4, R5       |
| 11: | ST   | R4, (R3)         |
| 12: | ADD  | R1, R1, 4 *      |
| 13: | ADD  | R3, R3, 4        |
| 7:  | BNEZ | R7, C2 *         |

C1:            **JMP**    *Gata*  
 C2:            **ST**     *R5, -4(R3)*  
 C3:            **ADD**   *R2, R2, 4*  
 C4:            **SUB**   *R1, R1, 4 (compensare 12)*  
*Gata:*

Pentru a pastra semantica programului, unele instructiuni au fost înlocuite cu altele noi (s-au doar opcode-ul acestora). Pentru instructiunea 13 nu mai este nevoie de cod compensator daca fac scrierea corecta în memorie la adresa **-4(R3)** R3 pointând spre locatia curenta (libera - nescrisa) din tabloul rezultat.

76. a)

$$IR_{NT} = 7/9 \text{ [instr./ciclu]}$$

$$IR_T = 4/6 \text{ [instr./ciclu]}$$

b)

- |    |                      |                                          |
|----|----------------------|------------------------------------------|
| 1. | $A \leftarrow B$     |                                          |
| 3. | if (A=Q) then goto 8 |                                          |
| 2. | $B \leftarrow B - 1$ | } BDS umplut!                            |
| 7. | $X \leftarrow Q$     |                                          |
|    |                      | 2 creste performanta întotdeauna         |
|    |                      | 7 creste performanta când saltul se face |
| 4. | $Q \leftarrow Q + 1$ |                                          |
| 5. | $D \leftarrow E$     |                                          |
| 6. | $E \leftarrow F$     |                                          |
| 7. | $X \leftarrow Q$     |                                          |
| 8. |                      |                                          |

Rezulta:

$$IR_{NT} = 7/8 \text{ [instr./ciclu]}$$

$$IR_T = 4/4 \text{ [instr./ciclu]}$$

**Obs1:** Evident, optimizarea trebuie sa pastreze logica secventei initiale.

**Obs2:** Când saltul nu se face, executia aditionala în BDS a instructiunii 7 nu altereaza semantica.

**Obs3:** Performanta secventei optimizate creste în ambele variante (NT/T).

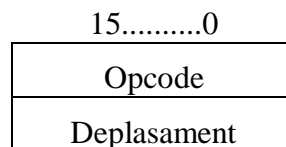
78.

**Bucula optimizata:**

|       |      |            |               |              |
|-------|------|------------|---------------|--------------|
|       | ld   | R4, (R5)0  | → load 1      |              |
|       | xor  | R7, R4, R9 | → xor 1       | Prolog       |
|       | ld   | R4, (R5)4  | → load 2      |              |
| Loop: | st   | (R5)0, R7  | → store (k)   |              |
|       | xor  | R7, R4, R9 | → xor (k+1)   | Corp         |
|       | ld   | R4, (R5)8  | → load (k+2)  | Bucula       |
|       | add  | R5, R5, #4 |               | Pipelinizata |
|       | sub  | R6, R6, #1 |               |              |
|       | brnz | R6, Loop   |               |              |
|       | st   | (R5)0, R7  | → store (n-1) |              |
|       | xor  | R7, R4, R9 | → xor (n)     | Epilog       |
|       | st   | (R5)4, R7  | → store (n)   |              |

**Bucula initiala:** Hazarduri RAW prin R4, R7 ⇒ 4 cicli (nop ⇔ LDS)**Bucula optimizata:** Elimina hazardurile RAW din corpul buclei ⇒ 1 ciclu.**Obs:** Orice variatiuni coerente de calcul sunt considerate (load pipelinizat sau nu, etc).

79. Codificarea instructiunii este urmatoarea:



Se efectueaza urmatoarele operatii:

| Operatia executata                                      | Durata executie (cicli) |
|---------------------------------------------------------|-------------------------|
| Fetch Instructiune (Opcode) - 2o                        | 6                       |
| Fetch Deplasament - 2o                                  | 4                       |
| Calcul adresa de memorare (R <sub>9</sub> +deplasament) | 2                       |
| Aducere octetii (0 si 1) din memorie                    | 4                       |
| Aducere octetii (2 si 3) din memorie                    | 4                       |
| Aducere octetii (4 si 5) din memorie                    | 4                       |
| Aducere octetii (6 si 7) din memorie                    | 4                       |

Total cicli executie = 28.



80. a) Codificarea instructiunii este urmatoarea:

15.....0

| Opcode                                |
|---------------------------------------|
| Deplasament                           |
| doar pentru instructiunile Load/Store |

Se efectueaza urmatoarele operatii:

| Operatia executata                                           | Durata executie (cicli) |
|--------------------------------------------------------------|-------------------------|
| Fetch Instructiune (Opcode) - 2o (1)                         | 6                       |
| Fetch Deplasament - 2o (1)                                   | 4                       |
| Calcul adresa de memorare ( $R_9 + \text{deplasament}$ ) (1) | 2                       |
| Aducere octetii (0 si 1) din memorie (1)                     | 4                       |
| Fetch Instructiune (Opcode) - 2o (2)                         | 6                       |
| Executa operatia de înmultire si scriere rezultat (2)        | 2                       |

Total cicli executie = 24

b) Codificarea instructiunii se pastreaza;

Se efectueaza urmatoarele operatii:

| Operatia executata                                           | Durata executie (cicli) |
|--------------------------------------------------------------|-------------------------|
| Fetch Instructiune (Opcode) - 2o (1)                         | 6                       |
| Fetch Deplasament - 2o (1)                                   | 4                       |
| Calcul adresa de memorare ( $R_9 + \text{deplasament}$ ) (1) | 2                       |
| Fetch Instructiune (Opcode) - 2o - 2 cicli de tact (2)       |                         |
| Aducere octetii (0 si 1) din memorie (1)                     | 4                       |
| Fetch Instructiune (Opcode) - 2o - 4 cicli de tact (2)       |                         |
| Executa operatia de înmultire si scriere rezultat (2)        | 2                       |

Total cicli executie = 18

81. a)

- DWB determina prevenirea stagnarilor procesorului acolo unde nu exista suficiente porturi de scriere în cache-ul de date pentru a asigura toate cererile de scriere existente.

- Eliminarea hazardurilor RAW prin colapsarea dependentelor de date. Rezolvarea prin **bypassing** a hazardurilor de tip “Load after Store” cu adrese identice.
  - DWB = mic procesor de iesire, implementat ca o coada FIFO de lungime parametrizabila.
  - Lucreaza în paralel cu procesorul eliberându-l pe acesta de sarcina scrierii în cache-ul de date.
- b)
- Posibilitatea de a executa simultan mai multe instructiuni Load fara a fi necesara multiplicarea porturilor de citire ale cache-ului de date.
  - *Dimensiunea Cache-ului de date*: Un cache mic determina putine blocuri rezulta posibilitatea ca instructiunile Load sa citeasca date din acelasi bloc.
  - *Capacitatea bufferului de prefetch*: Un buffer de instructiuni mare faciliteaza prezenta simultana a mai multor instructiuni Load în buffer care pot citi din acelasi bloc.

82. Codificarea instructiunii este urmatoarea:

|             |
|-------------|
| 15.....0    |
| Opcode      |
| Deplasament |

Se efectueaza urmatoarele operatii:

| Operatia executata                                       | Durata executie (cicli) |
|----------------------------------------------------------|-------------------------|
| Fetch Instructiune (Opcode) - 2o                         | 6                       |
| Fetch Deplasament - 2o                                   | 4                       |
| Calcul adresa de memorare ( $R_4 + \text{deplasament}$ ) | 2                       |
| Scriere octetii (0 si 1) în memorie                      | 4                       |
| Scriere octetii (2 si 3) în memorie                      | 4                       |
| Scriere octetii (4 si 5) în memorie                      | 4                       |
| Scriere octetii (6 si 7) în memorie                      | 4                       |

Total cicli executie = 28.

83. a)

$$IR_{NT} = 7/11 \text{ [instr./ciclu]}$$

$$IR_T = 4/7 \text{ [instr./ciclu]}$$

b)

1.  $A \leftarrow B$
3. If (D=Q) then goto 8
7.  $X \leftarrow Q$  } BDS umplut!
2.  $C \leftarrow A - 1$  2 creste performanta întotdeauna
4.  $Q \leftarrow Q + 1$  7 creste performanta când saltul se face
5.  $D \leftarrow E$
7.  $X \leftarrow Q$
6.  $F \leftarrow D * A$
- 8.

Rezulta:

$$IR_{NT} = 8/8 \text{ [instr./ciclu]}$$

$$IR_T = 4/4 \text{ [instr./ciclu]}$$

**Obs1:** Evident, optimizarea trebuie sa pastreze logica secventei initiale.**Obs2:** Când saltul nu se face, executia aditionala în BDS a instructiunii 7 nu altereaza semantica.**Obs3:** Performanta secventei optimizate creste în ambele variante (NT/T).

84.

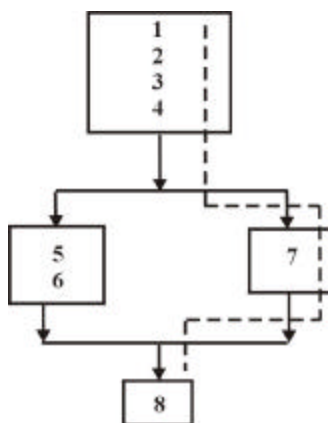
Executia trace-ului neoptimizat

| DECODIFICARE |    |    |    | EXECUTIE |      |     |    |     | CICLU |
|--------------|----|----|----|----------|------|-----|----|-----|-------|
| I0           | I1 | I2 | I3 | ALU1     | ALU2 | SHF | LS | BRN |       |
| 1            | 2  | 3  | 4  |          |      |     |    |     | 1     |
|              |    |    |    | 1        |      |     |    |     | 2     |
|              |    |    |    |          |      |     | 2  |     | 3     |
|              |    |    |    |          |      |     |    |     | 4     |
|              |    |    |    |          | 3    |     |    |     | 5     |
| 5            | 6  | 7  | 8  |          |      |     |    | 4   | 6     |
|              |    |    |    | 7        | 8    |     |    |     | 7     |

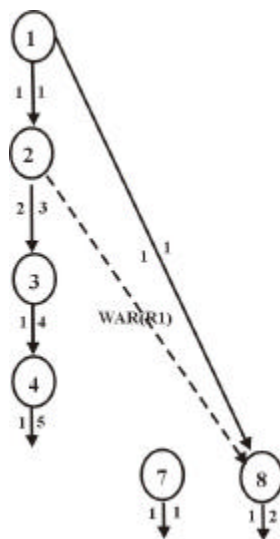
Datorita paralelismului limitat al acestei secvente se obtine o rata medie de executie de doar **1,14** instr. / ciclu. Graful de control al acestei secvente este complus din 3 basic block-uri ca în figura 1.

De remarcat ca decodificarea unui nou grup de instructiuni poate avea loc si în tactul imediat urmator lansarii în executie a tuturor instructiunilor anterior decodificate.

Graful de control aferent secventei de optimizat



Graful dependentelor pentru trace-ul considerat



Ordinea de executie în urma optimizarii

| Nod / Prioritate |
|------------------|
| 4/5;             |
| 3/4; 8/2;        |
| 2/3; 7/1;        |
| 1/1;             |

Executia secventei optimizate se face doar în 5 cicli si datorita hazardurilor reale de date ( $IR = 8/5 \approx 1,6$  instr./ciclu):

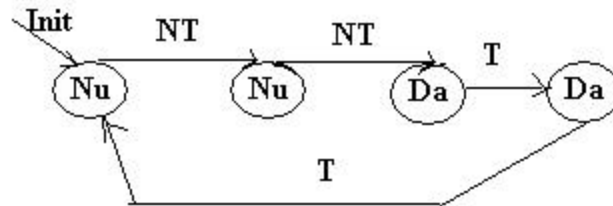
| EXECUTIE |      |     |    |     | CICLU |
|----------|------|-----|----|-----|-------|
| ALU1     | ALU2 | SHF | LS | BRN |       |
| 1        |      |     |    |     | 1     |
| 7        |      |     | 2  |     | 2     |
|          |      |     |    |     | 3     |
| 8        | 3    |     |    |     | 4     |
|          |      |     |    | 4   | 5     |

Întrucât basic block-ul care contine instructiunea 7 se executa preponderent se obtine un avantaj prin migrarea instructiunilor 7 si 8 în basic block-ul anterior fiind necesare însa coduri compensatoare. Codul reorganizat arata astfel:

|       |             |                                 |
|-------|-------------|---------------------------------|
| 1:    | ADD         | R1, R0, base_a *                |
| 2:    | LD          | R4, (R1)                        |
| 7:    | ADD         | contor_minus, contor_minus, 1 * |
| 3:    | SGT         | R7, R4, R0                      |
| 8:    | ADD         | R1, R1, 4 *                     |
| 4:    | <b>BNEZ</b> | R7, C2 *                        |
| C1:   | JMP         | Gata                            |
| C2:   | ADD         | contor_plus, contor_plus, 1     |
| C3:   | SUB         | contor_minus, contor_minus, 1   |
| Gata: |             |                                 |

Pentru a pastra semantica programului, unele instructiuni au fost înlocuite cu altele noi (s-au doar opcode-ul acestora). Pentru instructiunea 8 nu mai este nevoie de cod compensator, ea executându-se indiferent de ramura pe care se continua procesarea.

89. a)



b) Analog (3 Da, 1 Nu)  
 $4^4 = 256$  automate distincte!

90.

a) Din ipoteza problemei rezulta ca:

Daca  $PC = 2DF4h$  si  $HRg=0 \Rightarrow$  Adresa destinatie (TARGET) este  $4FC0h$  si  
 bitul  $PRED=1$

Daca  $PC = 2DF4h$  si  $HRg=1 \Rightarrow$  Adresa destinatie (TARGET) este  $2DF8h$  si  
 bitul  $PRED=0$

Preluând informatiile de corelatie(k) si gradul de localizare (i) rezulta:

$PC = 2DF4h = 10.1101.1111.0100_2$ . Concatenând doar cei 11 biti (i) ai PC-ului cu  $HRg$  (1 bit) rezulta

i)  $HRg=0 \Rightarrow 10.1 \mid 101.1111.0100_2 + 0_2 \Rightarrow 1011.1110.1000_2 = BE8h \Rightarrow$   
 TARGET= $4FC0h$  si  $PRED=1$

ii)  $HRg=1 \Rightarrow 10.1 \mid 101.1111.0100_2 + 1_2 \Rightarrow 1011.1110.1001_2 = BE9h \Rightarrow$   
 TARGET= $2DF8h$  si  $PRED=0$

Adresele  $BE6h$  si  $BE7h$  sunt neprecizabile nedispunând de alte informatii.

b) Datorita procentajului  $\approx 50\%$  T si tot la fel NT, saltul 2 este mai greu predictibil.

91. Pentru a veni în sprijinul solutiei se reamintesc primele 5 instructiuni din secventa de procesat:

|            |                 |
|------------|-----------------|
| <b>15.</b> | <b>BS 27 9</b>  |
| <b>16.</b> | <b>BM 17 28</b> |
| <b>17.</b> | <b>BT 35 38</b> |
| <b>18.</b> | <b>NT 40 41</b> |
| <b>19.</b> | <b>BT 45 27</b> |

Se va prezenta doar cazul tabelii de predictie - mapata direct de capacitate 8 locatii, pentru tabela asociativa problema se rezolva similar din punct de vedere metodologic, doar parametrii difera: INDEX, TAG, TARGET.

Notam  $w=0$  – *numarul de predictii gresite.*

$g=0$  – *numarul de predictii corecte.*

Predictia instructiunilor urmeaza cursul:

**1 - (BS 27 9)** –  $PCert=27 \Rightarrow$  locatia accesata din BTB este data de :

**INDEX** =  $PCert \bmod 8 = 3$  si **TAG** =  $PCert \div 8$

**TARGET** = 9

Predictia anterioara a fost 0 dar primul caracter **B** implica faptul ca saltul se face  $\Rightarrow$  predictie incorecta  $\Rightarrow w=1$  si predictia viitoare (câmpul **PRED**) devine 1 (bineînțele pe respectiva locatie).

**2 – (BM 17 28)** –  $PCert=17 \Rightarrow$  locatia accesata din BTB este data de

**INDEX** =  $PCert \bmod 8 = 1$  si **TAG** =  $PCert \div 8 = 2$

**TARGET** = 28

Predictia anterioara a fost 0 dar primul caracter **B** implica faptul ca saltul se face  $\Rightarrow$  predictie incorecta  $\Rightarrow w=2$  si predictia viitoare (câmpul **PRED**) devine 1 (pe locatie 1).

**3 – (BT 35 38)** –  $PCert=35 \Rightarrow$  locatia accesata din BTB este data de

**INDEX** =  $PCert \bmod 8 = 3$  si **TAG** =  $PCert \div 8 = 4$

**TARGET** = 38

Predictia anterioară a fost 1, primul caracter **B** implica faptul că saltul se face  $\Rightarrow$  predicție corectă (pe direcție) dar Target-ul diferă  $\Rightarrow w=3$ , predicția viitoare (câmpul **PRED**) rămânând 1 (pe locația 3).

**4** – (NT 40 41) – PCert=40  $\Rightarrow$  locația accesată din BTB este data de

**INDEX** = PCert **modulo** 8 = 0 și **TAG** = PCert **div** 8 = 5

**TARGET** = 0 (nu se va introduce în BTB)

Predictia anterioară a fost 0, primul caracter **N** implica faptul că saltul nu se face  $\Rightarrow$  predicție corectă dar saltul nu se va introduce în tabelă  $\Rightarrow g=1$ , predicția viitoare (câmpul **PRED**) rămânând 0 (pe locația 5).

**5** – (BS 27 9) – PCert=27  $\Rightarrow$  locația accesată din BTB este data de

**INDEX** = PCert **modulo** 8 = 3 și **TAG** = PCert **div** 8 = 4

**TARGET** = 9

Predictia anterioară a fost 1, primul caracter **B** implica faptul că saltul se face  $\Rightarrow$  predicție corectă (pe direcție) dar Target-ul diferă  $\Rightarrow w=3$ , predicția viitoare (câmpul **PRED**) rămânând 1 (pe locația 3).



## 13. CE CONTINE CD-UL ?

---

CD-ul ce însoțeste această carte cuprinde o serie de aplicații software, documente și alte fișiere care încearcă să sprijine, la un cost rezonabil necesarul zilnic de cunoaștere al studenților din anii terminali, masteranți și doctoranți din domeniul ingineriei calculatoarelor și domenii conexe; al inginerilor și cercetătorilor.

Cea mai mare parte a programelor și fișierelor existente pe acest CD sunt în limba română (dezvoltate în cadrul *colectivului de cercetare în microarhitecturi* din cadrul catedrei de Calculatoare și Automatizări a Facultății de Inginerie, Sibiu și condus de către dl. prof.univ.dr.ing. Lucian Vintan, unul dintre autorii acestei lucrări). Celelalte aplicații (simulatoare, benchmark-uri și alte instrumente software) au fost descărcate în mod *free* de la adrese web pe care le vom aminti la momentul potrivit. Pe baza acestor simulatoare, autorii au dezvoltat o colecție de lucrări practice care descriu și îmbunătățesc funcționarea microarhitecturilor simulate, dar și simulatoare și benchmark-uri proprii care implementează cele mai recente tehnici din domeniul ***procesoarelor cu paralelism la nivelul instructiunilor***. Materialul este organizat pe mai multe directoare, fiecare dintre ele corespunzând, pe cât posibil, unui simulator (arhitectura) distinct. Mai sunt prezente, de asemenea, și benchmark-urile cu intrările aferente necesare simulării. Acele programe sau fișiere care nu au putut fi incluse în nici unul dintre domeniile prezentate mai jos sunt grupate sub denumirea comună de **Diverse**.

În plus, dacă doriți să citiți această carte în format electronic, va punem la dispoziție textul cărții în format *.pdf*. Pentru a putea citi acest format aveți la dispoziție programul *Adobe Acrobat Reader* pe care trebuie să-l instalați pe calculatorul dumneavoastră.

### 13.1. SIMULAREA UNOR ARHITECTURI CU PARALELISM LA NIVELUL INSTRUCTIUNILOR

Această secțiune realizează o descriere sumară a simulatoarelor, (structura, implementare, resurse necesare, ghid de utilizare, platforma de

executie), prezente pe suportul CD-ROM care însoteste aceasta carte, dar si pe Internet la adresa <http://www.ulbsibiu.ro/ing/vintan/simulatoare.html>, <http://vectra.ulbsibiu.ro/~adrian.florea>

Evaluarea performantelor arhitecturilor de calcul se face prin simulare, fara simulare fiind foarte dificil, practic imposibil, de estimat. Metodele de investigare folosite sunt cunoscute sub denumirile de *execution* sau *trace driven simulation*.

Simulatoarele sunt dedicate arhitecturilor RISC scalare (MIPS) si superscalare (DLX, SATSim - un simulator hibrid din punct de vedere al metodologiei de simulare, vizual si extrem de animat, care exploateaza conceptele avansate ale arhitecturii superscalare: executia 'out-of-order' a instructiunilor, 'branch prediction', studiul interfetei procesor-cache). De asemenea, sunt implementate simulatoare de cache-uri, scheme clasice si moderne de predictie a salturilor (BTB, GAg, predictor neuronal), simulatoare la nivel de executie a instructiunilor *in order* respectiv *out of order* într-o arhitectura superscalara parametrizabila tipica - HSA (*Hatfield Superscalar Architecture*), scheduler - optimizator de cod - pentru arhitectura HSA). O noutate în plus o constituie setul de instrumente "SimpleScalar 3.0", o colectie de instrumente software, pusa la dispozitia cercetatorilor în arhitecturi moderne de calcul, si care cuprinde: compilatoare, asamblatoare, link-editoare, simulatoare si instrumente de vizualizare a unei arhitecturi (super)scalare simple, generice si care ofera posibilitatea simulării de tip *execution driven*, cât mai detaliate si de înalta performanta a celor mai moderne microprocesoare.

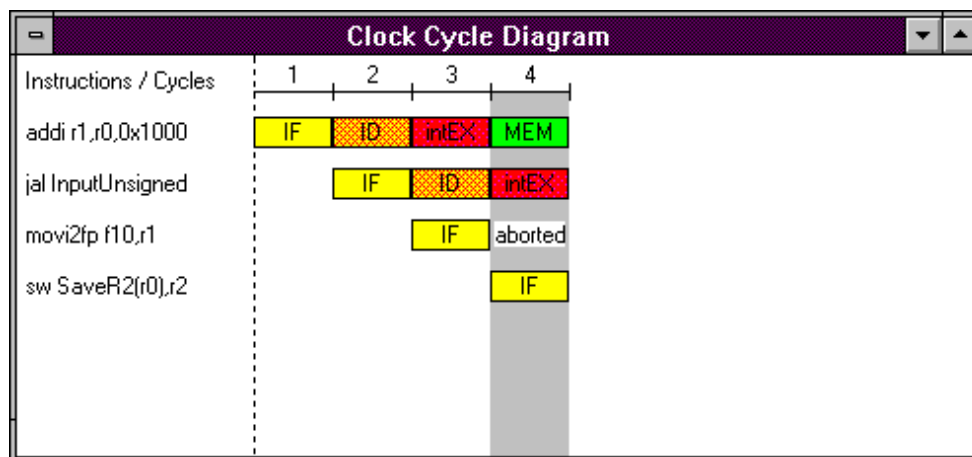
Arhitectura familiei de microprocesoare RISC MIPS R2000/3000 este aprofundata prin intermediul simulatorului software numit SPIM, conceput de catre *James R. Larus*. Arhitectura MIPS, pe lângă succesul sau comercial, constituie unul dintre cele mai inteligibile si elaborate arhitecturi RISC.

SPIM/SAL este o portare pe WIN32S a SPIM, simulator scris pentru uz didactic în Facultatea de Calculatoare a Universitatii din Wisconsin, SUA. WIN32S este o extensie pe 32 de biti a sistemului Windows 3.1. Interfata programabila cu aplicatia a WIN32S este un subset al WIN32, un API suportat de sistemul de operare Windows NT. Aplicatii ce folosesc WIN32S API, cum sunt SPIM/SAL, sunt compilate într-un format executabil numit *executabil portabil* (PE) care ruleaza fie pe Microsoft Windows 3.1. cu extensie WIN32S fie pe Microsoft Windows NT.

SPIM poate citi si executa imediat fisiere scrise în limbaj de asamblare sau executabile MIPS. El contine un debugger si asigura câteva servicii specifice sistemelor de operare. SPIM este mult mai lent decât un

calculator real (aproximativ de 100 de ori). Totusi, costul scazut si larga aplicabilitate nu poate fi egalata de hardware-ul adevarat.

Microprocesor virtual, abreviat DLX, conceput de catre profesorii *John Hennessy* (Univ. Stanford, SUA) si *David Patterson* (Univ. Berkeley, SUA), reprezinta o arhitectura RISC superscalara, didactica dar si performanta, cu cinci nivele pipeline de procesare, care este investigata prin intermediul unui excelent simulator software, înzestrat cu facilitati didactice deosebite si cu o grafica atragatoare. Simulatorul a fost implementat software de catre prof. Herbert Grünbacher de la *Institut für Technische Informatik, University of Technology, Vienna*, 1992 si este disponibil gratuit pe Internet la adresa <ftp://ftp.mkp.com/pub/dlx>. Scopul este de a ajuta la înțelegerea conceptelor legate de procesarea pipeline a instructiunilor precum si a aspectelor arhitecturale specifice procesoarelor RISC. Pentru o înțelegere mai buna dar si pentru a stârni interesul cititorului, în continuare se exemplifica sumar, pe diagrama ciclor procesorului, desfasurarea procesarii pipeline a instructiunilor, aferente unui program de test.



**Figura 13.1.** Diagrama ciclului de tact

În diagrama alaturata se observa ca simularea este în al patru-lea ciclu, prima instructiune este în nivelul MEM (acces la memorie), a doua în intEX (executie operatie aritmetico-logica) si a patra în IF (extragere instructiune din memorie). A treia instructiune este întrerupta. Motivatia consta în faptul ca a doua instructiune **jal**, este un apel neconditionat. Acest lucru e cunoscut abia dupa cel de-al treilea ciclu de tact, dupa ce instructiunea de apel a fost decodificata. În acest timp, instructiunea imediat urmatoare celei de salt (**movi2fp**) a trecut de faza IF, însa instructiunea care se va executa efectiv dupa instructiunea de salt se afla la alta adresa. Astfel, executia instructiunii

**movi2f** trebuie întrerupta, lasând loc gol în pipe. Saltul se va face la eticheta *InputUnsigned*. Pentru detalii suplimentare accesati simulatorul DLX de pe CD.

SATSim (*'Superscalar Architecture Trace Simulator'*) reprezinta un simulator hibrid, adica simularea se face pe trace-uri (*"trace-driven"*), dar sunt prezentate în fiecare ciclu de tact continutul fiecărei unitati de executie, a statiilor de rezervare, a buffer-ului de reordonare si de redenumire, a structurii pipeline(*'execution driven'*), o unealta de animatie interactiva care exploateaza conceptele avansate ale arhitecturii superscalare: executia *'out-of-order'* a instructiunilor, *'branch prediction'*, studiul interfetei procesor-cache.

SATSim este puternic parametrizabil, permite utilizatorilor sa modifice interactiv parametrii configuratiilor hardware si sa vizualizeze efectele lor într-o maniera mult mai accesibila decât este posibil cu ajutorul altor simulatoare existente (de ex. simulatoarele setului SimpleScalar, simulatorul Out of Order aferent arhitecturii HSA).

Implementarea SATSim s-a realizat în mediul Developer Studio din Visual C++, versiunea 6.0, oferind utilizatorului o interfata prietenoasa, bazata pe meniuri, ferestre de dialog, imagini grafice edificatoare. Programul ruleaza pe platforme Windows 9x sau NT. SATSim a fost realizat pentru uzul studentilor de catre *Mark Wolf* si *Linda Wills* si este folosit la "Georgia Institute of Technology" într-un curs de arhitectura calculatoarelor. Actuala versiune a simulatorului poate fi descarcata gratuit de pe Internet de la adresa: <http://www.ece.gatech.edu/research/pica/SATSim/satsim.html>.

Pentru a utiliza si înțelege cât mai clar modul de lucru al simulatorului sunt necesare cunostinte minime de operare în Windows dar si cunostinte bogate de microarhitectura calculatoarelor, modele de procesare existente, superscalaritate, simulare de tip *execution* respectiv *trace driven*. Înțelegerea acestor concepte fundamentale trebuie sa stea la baza viitoarelor teme de cercetare gen: *trace cache*, *localitatea* si *predictia valorilor*, *reutilizarea dinamica a instructiunilor*, *multiprocesare* si *multithreading*.

Din grupul simulatoarelor de cache-uri amintim pe cel ce realizeaza simularea interfetei procesor - cache (SIMCACHE) într-o arhitectura superscalara parametrizabila. Scopul principal al simularii îl constituie evaluarea si optimizarea unor elemente esentiale – memoriile cache – ce caracterizeaza practic toate microprocesoarele superscalare avansate. Accentul este pus pe problematica cache-urilor, mai precis pe relatia dintre nucleul de executie si o arhitectura de memorie de tip Harvard, într-un context de procesor paralel, folosind tehnicile cunoscute de scriere în cache (*write back* si *write through*). Reamintim ca, o arhitectura Harvard se

caracterizeaza atât prin spatii separate pentru instructiuni si date cât si prin busuri separate între procesor si memoria cache de date respectiv de instructiuni.

Al doilea simulator de cache-uri dezvoltat implementeaza conceptul de (*selective*) *victim cache* (HSVICTP2). Scopul urmarit este îmbunatatirea performantelor memoriilor cache cu mapare directa, integrate într-un procesor paralel, utilizând conceptul de victime cache (simplu si selectiv). Pentru a reduce rata de miss a cache-urilor mapate direct (fara sa se afecteze timpul de hit sau penalitatea în caz de miss), *Norman Jouppi* de la compania DEC (actualmente *Compaq*) a propus în 1990 conceptul de *victim cache* (o memorie mica complet asociativa, plasata între primul nivel de cache mapat direct si memoria principala; blocurile înlocuite din cache-ul principal datorita unui miss sunt temporar memorate în victim cache; daca sunt referite din nou înainte de a fi înlocuite din victim cache, ele pot fi extrase direct din victim cache cu o penalitate mai mica). Pentru reducerea numarului de interschimbari dintre cache-ul principal si victim cache, *Stiliadis* si *Varma*, au introdus în 1994 un nou concept numit *selective victim cache* (SVC). Investigatiile propuse plus aspecte legate de implementarea ideala (în memoria principala) vs. folosind tabele de dispersie a bitilor de hit memorate în primul nivel de cache, respectiv folosirea vectorilor binari (*hit* si *sticky*) pentru o pastrare graduala a istoriei folosirii blocurilor, s-au realizat cu ajutorul simulatorului amintit mai sus, parametrizabil, scris în limbajul Pascal sub mediul Borland Delphi 3.0.

Dintre cele trei predictoare de branch-uri prezentate si simulate, primele doua (BTB si GAg) constituie o implementare detaliata a schemelor clasice de predictie cunoscute în literatura de specialitate sub numele de BTB (Branch Target Buffer) respectiv GAg (Global History Register, Global Prediction History Table). Scopul simularii îl reprezinta investigarea unor arhitecturi moderne de predictie a ramificatiilor de program (**branch**), în vederea reducerii penalitatilor introduse de instructiunile de ramificatie în procesoarele pipeline superscalare. În carte, au fost explorate în mod critic metodologiile de predictie existente, iar apoi îmbunatatite, optimizate si stabilite limitările fundamentale. S-au stabilit, prin simulari complexe, schemele de predictie optimale asociate diferitelor tipuri de ramificatii din program.

Cel de-al treilea simulator de acest gen reprezinta o varianta neuronală de predictor, idee novatoare ce apartine unuia dintre autorii acestei carti. Predictorul neuronal de salturi este constituit în principal dintr-o retea neuronală, un registru de istorie globală (HRG) si o tabela cu registrul de istorie locală. Reteaua neuronală simulata este o retea de tip Multi Layer Perceptron având un singur nivel ascuns, un nivel de intrare si un nivel de

iesire. Registrul de istorie globala (HRG) este un registru de deplasare de dimensiune parametrizabila si contine rezultatul (Taken/Not Taken) ultimelor  $k$  salturi, având rolul unui prim nivel de istorie a salturilor. Tabela registrilor de istorie locala reprezinta al doilea nivel de istorie a salturilor. Spre deosebire de HRG, care era un registru comun tuturor salturilor, în acest caz, fiecare PC are teoretic propriul registru de istorie. Privind dintr-o alta perspectiva putem considera predictorul format dintr-o banda de intrare (trace-ul ce va fi simulat), un automat de predictie (retea neuronală) si o memorie (HRG si tabela HRL) ce contine istoria salturilor. Implementarea simulatorului este realizata în limbajul C++, iar ca mediu de dezvoltare Microsoft Visual C++ v.6.0. Fara a intra în amanunte legate de programarea sub Windows se poate spune ca structura interfetei utilizata este de tipul MDI, putând simula mai multe trace-uri simultan.

Daca simulatoarele anterior prezentate au fost de tipul *trace driven*, cu exceptia celui *hibrid* - SATSim, urmatoarele simulatoare sunt de tipul *execution driven*. Dintre acestea, primele fac parte din setul de instrumente SimpleScalar 3.0, o colectie de instrumente software, pusa la dispozitia cercetatorilor în arhitecturi moderne de calcul si cuprinde: compilatoare, asamblatoare, link-editoare, simulatoare si instrumente de vizualizare a unei arhitecturi (super)scalare simple, generice (Exemple: arhitecturi **PISA**, **Alpha AXP**, **ARM**). Setul mai cuprinde un depanator la nivel de cod sursa al benchmark-ului simulat (**DLite!**) si un generator de trace-uri la nivel de pipe a instructiunilor (aferent deocamdata doar celui mai detaliat simulator). Setul de instrumente "SimpleScalar" este distribuit gratuit (poate fi gasit si descarcat pe site-ul web "<http://www.cs.wisc.edu/~mscalar/simplescalar.html>") si ofera posibilitatea simulării de tip *execution driven*, cât mai detaliate si de înalta performanta a celor mai moderne microprocesoare. Programele simulate reprezinta parte integranta a setului de instrumente si sunt programe de test precompilate (format cod obiect) pentru arhitecturile PISA si Alpha, si o parte din benchmark-urile SPEC'95. Cei interesati pot dispune de compilatorul GNU GCC (precum si de utilitarele aferente), care permite fiecarui cercetator sa compileze/asambleze/linkediteze propriile programe de test scrise pentru arhitectura SimpleScalar. Setul de instrumente "SimpleScalar" si modulele de portare a compilatorului GNU pe diverse platforme a fost scris de Todd Austin - începând cu anul 1994 pe când era cercetator la universitatea din Wisconsin-Madison, actualmente fiind membru al echipei de cercetatori în paralelism la nivelul instructiunilor al firmei Intel Corporation. "SimpleScalar" este sustinut în continuare de Doug Burger (autorul, în cea mai mare parte, a documentatiei) si de Todd Austin. Compilatorul GNU si

instrumentele software aditionale (biblioteci, translatoare din Fortran în C) a fost scris de "*Free Software Foundation*".

Pornind de la scheletul predictorului de salturi *sim-bpred.c* autorii acestei lucrari au dezvoltat un nou modul destinat masurarii gradelor de localitate a valorii pe diverse resurse (instructiuni aritmetico-logice / cu referire la memorie, registrii procesorului MIPS - arhitectura virtuala care sta la baza setului de instrumente SimpleScalar). De asemenea, simulatorul permite exploatarea conceptului de localitate cu ajutorul tehnicii de predictie a valorii, mai precis, schema de tip *Last Value*. Dupa descarcarea de pe Internet de la adresa "<http://www.cs.wisc.edu/~mscalar/simplescalar.html>" a fisierului *simplesim-3.0b.tar.gz*, functia *myrand()* din *misc.c* a fost modificata pentru eliminarea erorii aparute la compilare; astfel apelul functiei *random()* a fost conditionat si de *defined(\_CYGWIN32\_)*. Pentru compilarea surselor am folosit aplicatia **Cygwin** (comanda *make*), acesta fiind un emulator de Linux, care permite generarea executabilului pentru sistemul Windows. Pentru implementarea noului simulator **VPred (Last Value Predictor)**, a fost necesara modificarea fisierului *Makefile* astfel încât la comanda *make* sa fie compilat si acesta. Structurile utilizate, precum si declaratiile functiilor se afla în fisierul *vpred.h*, iar definitiile functiilor pot fi gasite în *vpred.c*. Motorul simulatorului îl reprezinta functia *sim\_main()* din *sim-vpred.c*.

Implementarea interfetei simulatorului în mediul Developer Studio din Visual C++ (versiunea 6.0) s-a facut datorita faptului ca limbajul C++ ofera un suport puternic pentru programarea orientata pe obiecte: încapsulare, mosteniri multiple, redefinirea operatorilor, functii si clase prieten etc. Conceptele de mostenire si polimorfism creaza premisele dezvoltarii ulterioare (extinderii) a variantei actuale de simulator. De asemenea, implementarea s-a realizat în asa maniera încât, orice modificare (adaugare) în hardware sau software sa fie facuta cu minim de efort.

Pentru a porni simulatorul este nevoie de un sistem pe care sa fie instalat sistemul Windows 9x sau Windows 2000 si sa existe benchmark-urile SPEC (programe de test). Din motive ce tin de sistemul de operare Windows 9x (preluarea identicatorului unui proces parinte - lucru care se face diferit în Windows NT) aplicatia nu poate fi rulata pe Windows NT. Aceasta se datoreaza functiilor API folosite (*CreateToolhelp32Snapshot* - care creaza o lista cu toate procesele aflate în executie în momentul respectiv, *Process32First* - care determina primul proces din lista anterior creata si *Process32Next* - care determina urmatorul proces din lista ce respecta o anumita conditie), functii care nu au corespondent în Windows NT. Pentru functionarea corecta a aplicatiei "*Last Value Predictor*" procesele care stau în spatele aplicatiei si sunt vitale ei (**sim-inst.exe**, **sim-**

**regs.exe** si **sim-vpred.exe**), pot fi compilate/asamblate/linkeditate atât sub sistemul de operare Windows NT/Windows 2000 cât si sub Windows 9x.

Urmatorul simulator, denumit *Incremental&&Contextual* se înscrie pe aceeași linie a simulatoarelor componente ale setului SimpleScalar 3.0. Interfata este realizată în Visual C++ v.6.0, iar aplicația rulează fără probleme în Windows'9x sau Win2000, dar nu și în WinNT, pentru aceleași motive ca și simulatorul *LastValue*. Noul simulator înglobează noi tehnici de predicție dinamică a valorilor, și anume: predicțiile incrementale, contextuale și hibride. Simulatorul este exploatat obținându-se rezultate cantitative deosebit de sugestive în înțelegerea și optimizarea acestor microarhitecturi agresive. Accentul este pus pe interpretarea formativă a acestor rezultate și nu pe obținerea unor performanțe deosebite.

Fisierele incluse în cadrul setului SimpleScalar au următoarele extensii:

- ✓ **.C** - codul sursă original al simulatoarelor sau al programelor de test.
- ✓ **.ss** - formatul executabil al benchmark-urilor compilate pentru arhitectura SimpleScalar.
- ✓ **.in** - fișierele aplicate ca intrări pentru executabilele SPEC'95.
- ✓ **.res** - fișiere care cuprind rezultatele simulărilor: rate de procesare, rate de miss, acurateți de predicție etc.

CD-ul însoțitor cuprinde, de asemenea, două simulatoare la nivel de execuție a instrucțiunii, corespunzătoare arhitecturii superscalare concepute la Hatfield, Anglia. Primul, numit RES\_SIM (simulator de resurse), realizat într-o formă primară, dar puternic parametrizabil, nu implementează conceptul de cache însă deține diverse alte facilități precum: procesare *in order*, principalul rezultat generat fiind rata de procesare, vizualizarea gradului de utilizare a resurselor, generarea trace-ului de instrucțiuni în urma simulării etc.

Al doilea simulator aferent arhitecturii HSA (OUT\_SIM) este mult mai complex și îl completează pe primul. Este implementat conceptul de cache, împreună cu mecanisme auxiliare gen DWB (*data write buffer*) sau *Outstanding Buffer* (simulatorul de *victim cache*). De asemenea, noul simulator implementează mecanismul de execuție *out of order* cu procesele componente corespunzătoare (*branch prediction*, *out of order instruction dispatch*, *renaming* aplicat registrilor prin algoritmi tip *Tomasulo*, buffer de reordonare pentru menținerea precisă a întreruperilor și coerenței procesorului etc.).

Pe arhitectura superscalară HSA cu procesare *in order* s-a dezvoltat un *scheduler* (HSS – optimizator de cod obiect în vederea execuției cât mai rapide) în scopul reorganizării instrucțiunilor pentru eliminarea dependențelor de date existente, eliminarea software a BDS-urilor (*branch*



*delay slot*) si cresterea potentialului de paralelism. Scheduler-ul de instructiuni dezvolta o serie de tehnici: *software pipelining*, *combining*, *merging* (*move*, *immediate*), *inlining* aplicat procedurilor, analiza *anti-alias* etc., în scopul optimizarii performantelor buclelor (basic-block-urilor) pentru cresterea globala a nivelului de paralelism disponibil. Daca pentru toate simulatoarele anterioare platforma de rulare era Windows '9x sau NT, platforma de executie pentru scheduler-ul de instructiuni HSS este Linux sau Unix.

Procesorul superscalar HSA încearca sa creasca paralelismul la nivelul instructiunilor atât **static** - în momentul compilarii - prin scheduling, cât si **dinamic** - în timpul executiei - prin diverse mecanisme (procesare *out of order*, utilizare DWB, *branch prediction*). Ambele simulatoare (RES\_SIM si OUT\_SIM) pot primi ca parametri de intrare fisiere cod masina optimizate sau nu.

Un “dosar” (folder) va cuprinde programele de test (benchmark-urile) existente si necesare pentru simulare. Programele tip benchmark numite Stanford, sunt o suita de opt programe care necesita putina initializare. Programele implica executia a 100 pâna la 900 de mii de instructiuni dinamice. Codul original C este întâi trecut printr-un compilator “*gnu CC*” care produce formatul corect al codului în mnemonica de asamblare, precum si directive de asamblare, comenzi de alocare a datelor etc. Codul este apoi executat pe un simulator la nivel de instructiuni, care produce la iesire un fisier tip *trace* de instructiuni. De remarcat ca, la rularea repetata a aceluasi program C aferent oricaruia din cele opt benchmark-uri se obtine acelasi fisier *trace*. Fisierele prezente au urmatoarele extensii:

- ✓ **.C** - codul sursa original al programelor de test.
- ✓ **.ins** - varianta cod masina a benchmark-urilor necesare simulatorului de cache (BLWBCACH) si pentru cele doua simulatoare *execution driven* aferente arhitecturii HSA (RES\_SIM si OUT\_SIM).
- ✓ **.inp** - similar ca format cu cele *.ins* (practic o versiune mai noua), necesare ca parametri de intrare celor doua simulatoare *execution driven* aferente arhitecturii HSA (RES\_SIM si OUT\_SIM).
- ✓ **.out** - varianta optimizata a benchmark-urilor cu extensia *.ins*. Sunt utilizate de catre simulatorul RES\_SIM pentru cuantificarea gradului de paralelism obtinut într-o procesare *in order* prin scheduling.
- ✓ **.trc** - fisiere trace reprezentând o înlantuire de triplete *<TipInstr AdrCrt AdrDest>*, unde *TipInstr* poate lua una din valorile ‘B’ – branch, ‘L’ – load, ‘S’ – store; *AdrCrt* reprezinta valoarea registrului PC – adresa instructiunii curente, iar *AdrDest* reprezinta adresa de memorie a datei accesate – în cazul instructiunilor cu referire la memorie (‘L’ / ‘S’) sau adresa destinatie a saltului – în cazul

instructiunilor de salt si ramificatie ('B'). Sunt generate de simulatorul execution driven RES\_SIM având ca parametri de intrare benchmark-urile .ins, si necesare simulatoarelor *trace driven* de cache-uri: Exista totusi o deficiente a acestor trace-uri: nu evidentiaza salturile care nu se fac. Din acest motiv s-au generat noi trace-uri, care prind si aceste salturi (fisierele \*.tra).

- ✓ **.tra** - fisiere *trace* proprii, utilizate în cadrul predictoarelor de branch-uri implementate (PAP, NEURONAL). Sunt o prelucrare a programelor scrise în mnemonica de asamblare (\*.ins) si a trace-urilor originale (\*.trc), cu scopul de a evidentia toate salturile (inclusiv cele care nu se fac). Contin doar branch-urile (atât cele care se fac cât si cele care nu se fac) si exclud instructiunile Load / Store.
- ✓ **.lib, .dll** - biblioteci statice si dinamice necesare în simularea arhitecturilor HSA.
- ✓ **.shd** - varianta optimizata a benchmark-urilor Stanford, utilizata de simulatorul OUT\_SIM. Rezulta un fenomen foarte interesant si contrastant, si anume: performanta obtinuta (rata de procesare) printr-o executie *out of order* a fisierelor optimizate este inferioara celei obtinute la procesarea *in order* a acelorasi fisiere (si chiar celei obtinute prin procesare *out of order* a fisierelor optimizate).
- ✓ **.use** - fisiere rezultat, de tip text, generate în urma simularii la nivel de executie a benchmark-urilor Stanford, efectuata pe arhitectura HSA.

## 13.2. DOCUMENTE

Trei dintre fisierele aparținând secțiunii curente reprezintă cartea de față în format .pdf, .doc si zip.

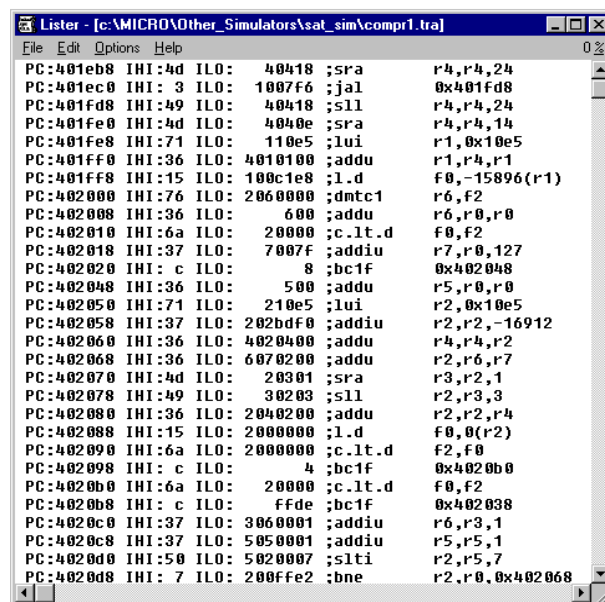
- ◆ Cartea de față în format .pdf sau .doc si arhivata cu utilitarul Winzip (format .zip).
- ◆ Alte fisiere în diverse formate .html si .txt, .bmp.

**Nota:** Programele de simulare cuprinse pe CD-ul atasat sunt concepute de grupul de cercetare al autorilor sau sunt de tip *shareware* / *freeware*, fiind în acest caz incluse fara modificari. Autorii acestei carti si editura nu-si asuma nici o responsabilitate pentru utilizarea acestor fisier si programe.

## ANEXA I

### SATSIM: GHID DE UTILIZARE

---



```
PC:401eb8 IHI:4d ILO: 40418 ;sra r4,r4,24
PC:401ec0 IHI: 3 ILO: 1007f6 ;jal 0x401fd8
PC:401fd8 IHI:49 ILO: 40418 ;sll r4,r4,24
PC:401fe0 IHI:4d ILO: 4040e ;sra r4,r4,14
PC:401fe8 IHI:71 ILO: 110e5 ;lui r1,0x10e5
PC:401ff0 IHI:36 ILO: 4010100 ;addu r1,r4,r1
PC:401ff8 IHI:15 ILO: 100c1e8 ;l.d f0,-15896(r1)
PC:402000 IHI:76 ILO: 2060000 ;dmtc1 r6,f2
PC:402008 IHI:36 ILO: 600 ;addu r6,r0,r0
PC:402010 IHI:6a ILO: 20000 ;c.lt.d f0,f2
PC:402018 IHI:37 ILO: 7007f ;addiu r7,r0,127
PC:402020 IHI: c ILO: 8 ;bc1f 0x402048
PC:402048 IHI:36 ILO: 500 ;addu r5,r0,r0
PC:402050 IHI:71 ILO: 210e5 ;lui r2,0x10e5
PC:402058 IHI:37 ILO: 202bdf0 ;addiu r2,r2,-16912
PC:402060 IHI:36 ILO: 4020400 ;addu r4,r4,r2
PC:402068 IHI:36 ILO: 6070200 ;addu r2,r6,r7
PC:402070 IHI:4d ILO: 20301 ;sra r3,r2,1
PC:402078 IHI:49 ILO: 30203 ;sll r2,r3,3
PC:402080 IHI:36 ILO: 2040200 ;addu r2,r2,r4
PC:402088 IHI:15 ILO: 2000000 ;l.d f0,0(r2)
PC:402090 IHI:6a ILO: 2000000 ;c.lt.d f2,f0
PC:402098 IHI: c ILO: 4 ;bc1f 0x4020b0
PC:4020b0 IHI:6a ILO: 20000 ;c.lt.d f0,f2
PC:4020b8 IHI: c ILO: ffd0 ;bc1f 0x402038
PC:4020c0 IHI:37 ILO: 3060001 ;addiu r6,r3,1
PC:4020c8 IHI:37 ILO: 5050001 ;addiu r5,r5,1
PC:4020d0 IHI:50 ILO: 5020007 ;slli r2,r5,7
PC:4020d8 IHI: 7 ILO: 200ffe2 ;bne r2,r0,0x402068
```

Figura I.1. Secventa de instructiuni din fisierul trace (*compr1.tra*)

Procesarea secventei de instructiuni starteaza în **ciclul 0** când se realizeaza faza *fetch instructiune*. Se aduce din cache-ul de instructiuni de la adresa specificata de PC (0x401EB8) doar prima instructiune (din motive de aliniere).

#### CICLUL 1.

În **ciclul 1** instructiunea de index 0 (cea aflata la adresa 0x401EB8) se va afla în faza de decodificare, în acelasi timp facându-se *fetch* asupra urmatorului grup de cel mult doua instructiuni (*factorul superscalar al arhitecturii* este 2 în cazul de fata si specifica numarul maxim de

instrucțiuni care pot fi citite din cache / decodificate / expediate spre stațiile de rezervare și unități de execuție), respectiv numărul maxim de instrucțiuni care pot scrie rezultatul în setul de registre generali (din locațiile *buffer*-ului de redenumire).

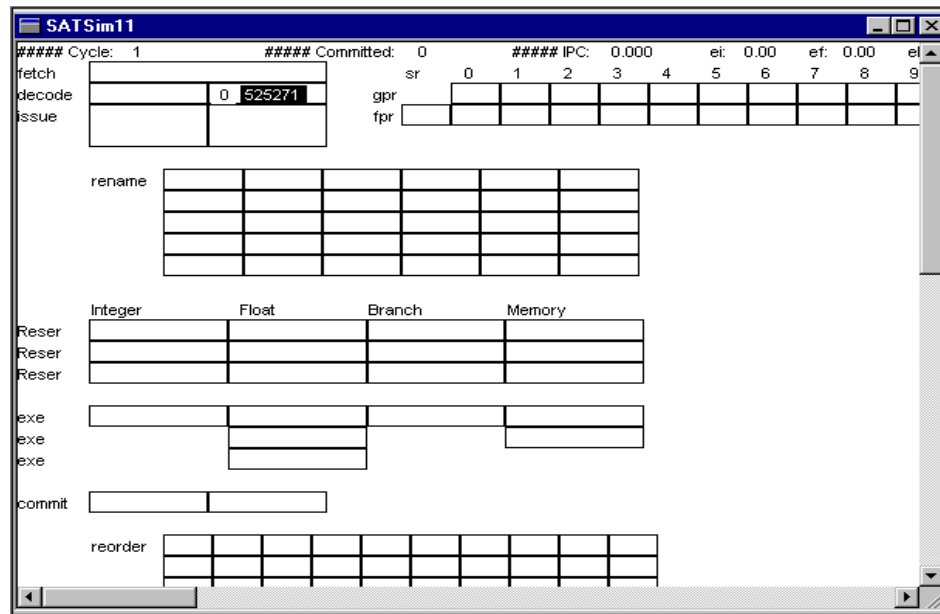
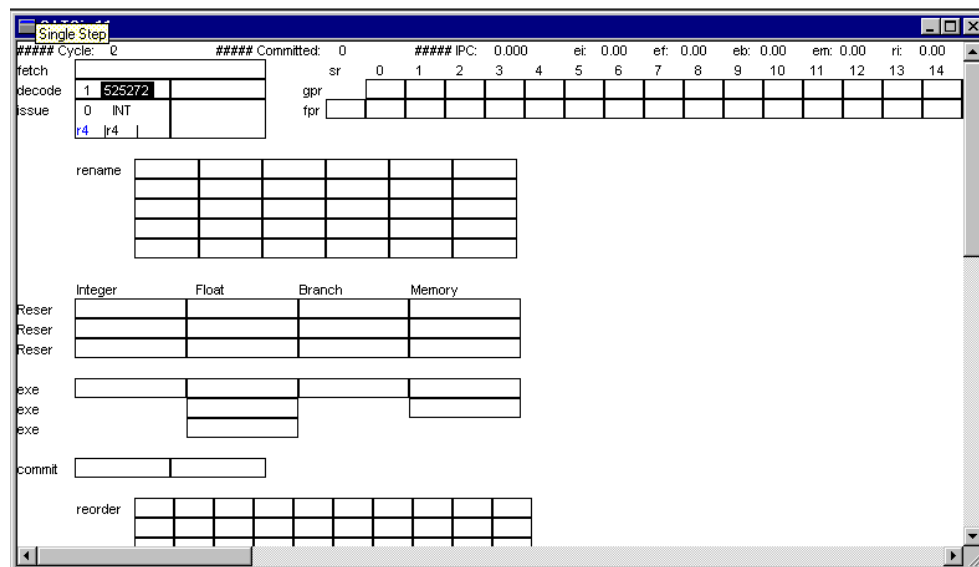


Figura I.2. Continutul resurselor arhitecturale în ciclul 1 de procesare

## CICLUL 2.

În **ciclul 2** instrucțiunea 0 se afla în faza de *dispatch* (*issue* 1<sup>st</sup>). După decodificare se cunoaște că ea este de tip întreg, iar registrul r4 are rol atât de operand sursă (casuta din mijloc, rândul de jos), cât și de registru destinație (casuta din stânga, rândul de jos). Al doilea operand sursă este de fapt o valoare imediată. În sprijinul afirmațiilor de mai sus sta chiar definiția primei instrucțiuni: **sra r4, r4, 24** care se poate vedea în figura I.1. Doar instrucțiunea 1 este decodificată, unul din motive, după cum se va vedea ulterior, fiind faptul că aceasta reprezintă un *salt taken* (care se face).

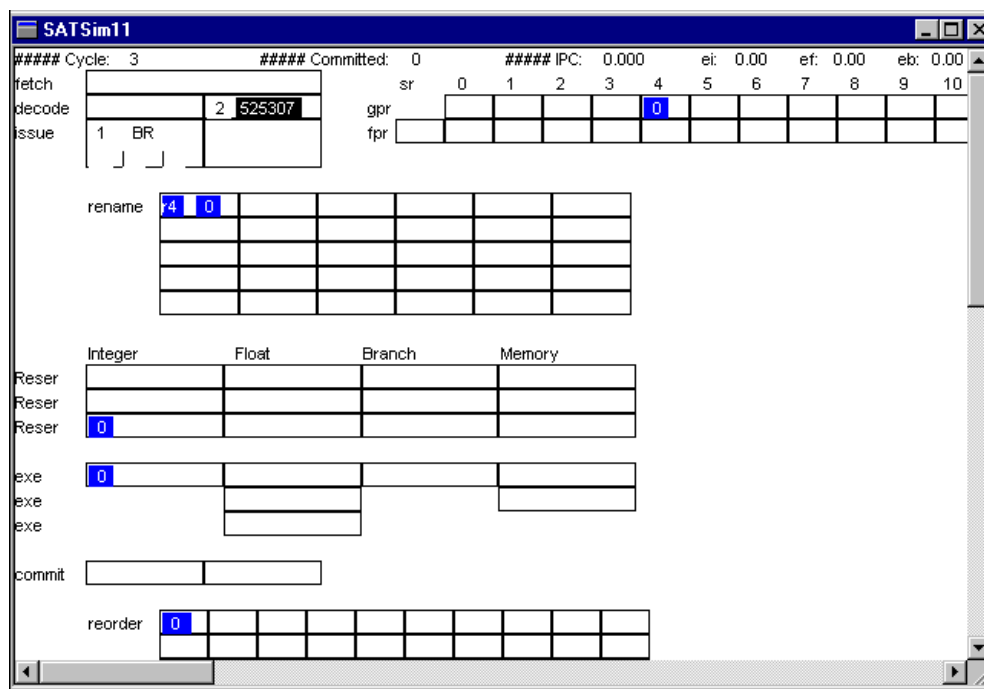


**Figura I.3.** Continutul resurselor arhitecturale în ciclul 2 de procesare

### CICLUL 3.

Acest ciclu (3) de procesare introduce o animozitate sporita fata de ceea ce s-a întâmplat pâna în acest moment, mai multe resurse arhitecturale (*buffer-ul de redenumire* si cel de *reordonare*, *statiile de rezervare* si *unitatile functionale de executie*) devenind operabile (au un rol activ). Întrucât instructiunea 0 modifica registrul destinatie r4, atunci se alocă o intrare în *buffer-ul de redenumire*, sugerându-se faptul ca instructiunea 0 va fi cea care genereaza valoarea lui r4. De asemenea, în *setul de registri arhitecturali* (generali) pentru valori întregi – **gpr** – la indexul 4 se va scrie indexul ultimei instructiuni care calculeaza valoarea lui r4. Ciclul 3 constituie cea de-a doua faza de *issue* (*issue 2<sup>nd</sup>*) pentru instructiunea 0, în care aceasta este transmisa statiei de rezervare dar si unitatii functionale de executie cu numere întregi, fiind disponibila, pentru efectuarea operatiei aritmetice specificata în opcode-ul instructiunii. În urma fazei de expediere (*issue 2<sup>nd</sup>*) se creeaza prin hard un “**tag**” care reprezinta numele unitatii de executie care va procesa rezultatul instructiunii respective. Din punct de vedere al interfetei simulatorului, acest tag este ilustrat prin numarul care scrie registrul destinatie (indexul instructiunii). Astfel ca, instructiunile viitoare care-l vor avea pe r4 ca registru sursa vor depinde de rezultatul instructiunii 0 (si vor astepta calcularea sa pentru a putea fi procesate în unitatile de executie corespunzatoare). De asemenea, instructiunea 0 va fi înscrisa în *buffer-ul de reordonare* pentru a ajuta la retragerea instructiunilor în ordinea initiala a programului, pentru asigurarea unui mecanism precis de

tratare a exceptiilor. În acelasi ciclu de tact, instructiunea 1 care s-a dovedit a fi de salt se afla în prima faza de *issue* (*issue 1<sup>st</sup>*), instructiunea 2 care reprezinta *target-ul* branch-ului se decodifica si se executa *fetch* asupra unui nou grup de instructiuni.



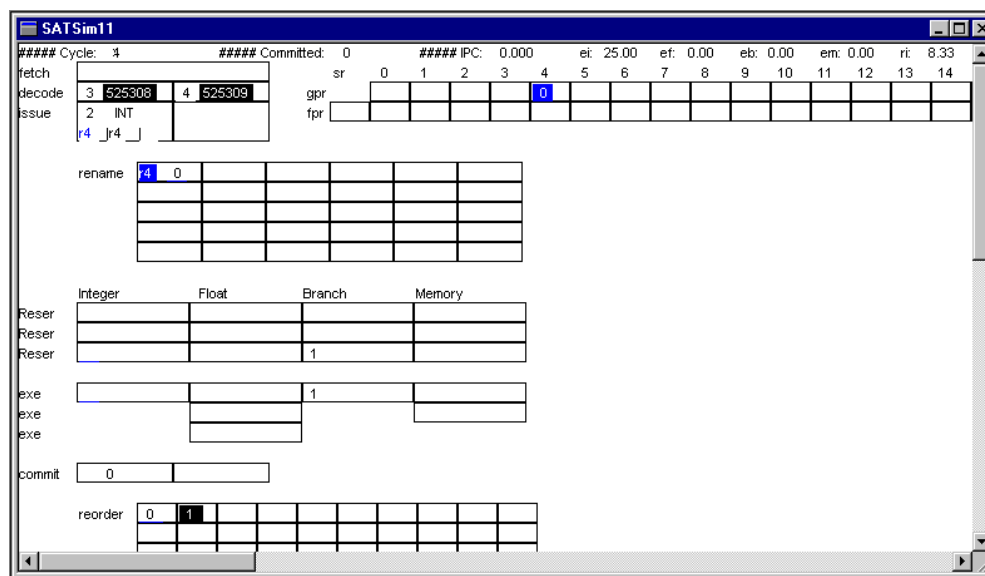
**Figura I.4.** Instantaneu realizat în ciclul 3 de executie

#### CICLUL 4.

Deoarece unitatea functionala de executie pentru numere întregi, pipeline-izata, efectueaza operatia aritmetico-logica într-un ciclu de tact, rezulta ca în **ciclul 4** de procesare rezultatul operatiei aferenta instructiunii 0 este disponibil în intrarea corespunzatoare buffer-ului de redenumire. Acest lucru se remarca prin pierderea culorii din casutele corespunzatoare a buffer-ului de redenumire (RenB) si reordonare (RB). Se executa practic faza *commit* în care rezultatul instructiunii din RenB (din setul de registri **fizici**) este transferat în setul de registri **arhitecturali** (generali). La finele acestui ciclu, instructiunea va fi evacuata din RenB si RB, iar locatia 4 a setului *gpr* va contine valoarea corecta.

În acelasi timp, instructiunea 1 este înscrisa doar în RB fiind o instructiune de salt; neavând registru destinatie nu se va realiza nici o redenumire. Întrucât aceasta instructiune se afla în executie propriu-zisa, în

buffer-ul de reordonare aceasta intrare este colorata. În faza de *dispatch* (*issue 1<sup>st</sup>*) se afla o noua instructiune (cu indexul 2), de tip întreg, având r4 pe post de sursa dar si de destinatie. În faza de decodificare se afla instructiunile 3 si 4.



**Figura I.5.** Instructiunea 0 efectueaza faza **commit** în ciclul 4 de executie

## CICLUL 5.

În **ciclul 5**, registrul r4 este redenumit cu “tag-ul” 2 (instructiunea care calculeaza noua valoare a lui r4); instructiunea 2 este înscrisa în *ReorderBuffer* iar unitatea functionala pentru întregi va trece la efectuarea operatiei. Întrucât operandul sursa este (numit) tot r4, în casuta corespunzatoare acestuia din statia de rezervare si unitatea de executie se va gasi tag-ul 0 – ultima instructiune care l-a scris pe r4. Instructiunea 1 efectueaza faza *commit*, instructiunea 3 (ALU) si 4 (LD) sunt în faza *issue 1<sup>st</sup>*, iar instructiunile 5 si 6 sunt decodificate. În setul *gpr* la locatia 4 se va scrie tag-ul 2.

În fiecare ciclu de procesare se pot vizualiza diverse informatii statice precum: numarul ciclului curent (*cycle*), câte instructiuni au fost în întregime procesate (*committed*), rata de procesare a instructiunilor ( $IPC = Committed/cycle$ ), ratele de utilizare a unitatilor functionale de executie si a statiilor de rezervare pentru fiecare tip de instructiune (ei, ri - pentru operatii cu numere întregi), (ef, rf – pentru operatii cu numere în virgula mobila),

(eb, rb – pentru instructiuni de salt), (em, rm – pentru instructiunile cu referire la memorie), rate de utilizare ale celor doua buffer-e de redenumire si reordonare.

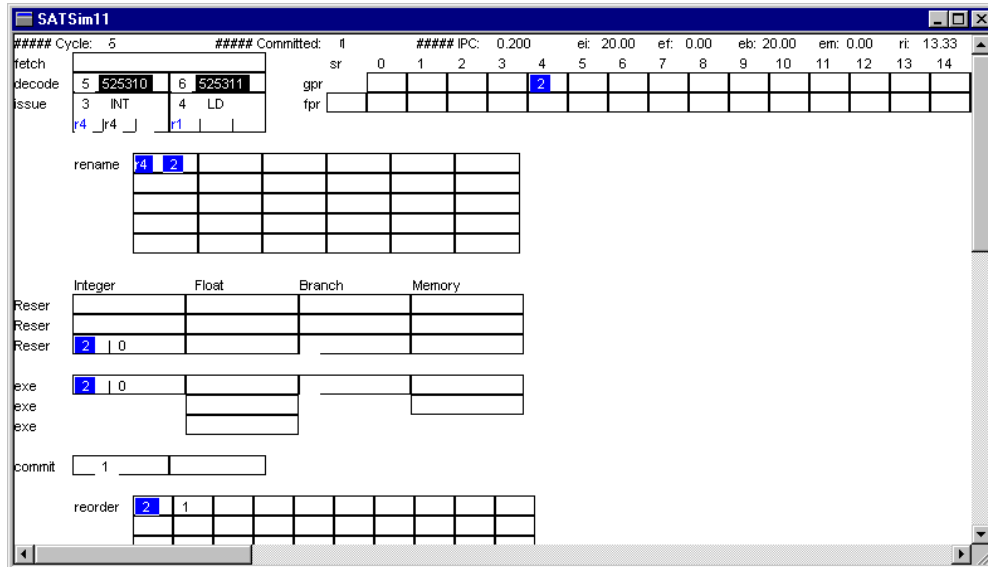


Figura I.6. Ciclul 5 de procesare

## CICLUL 6.

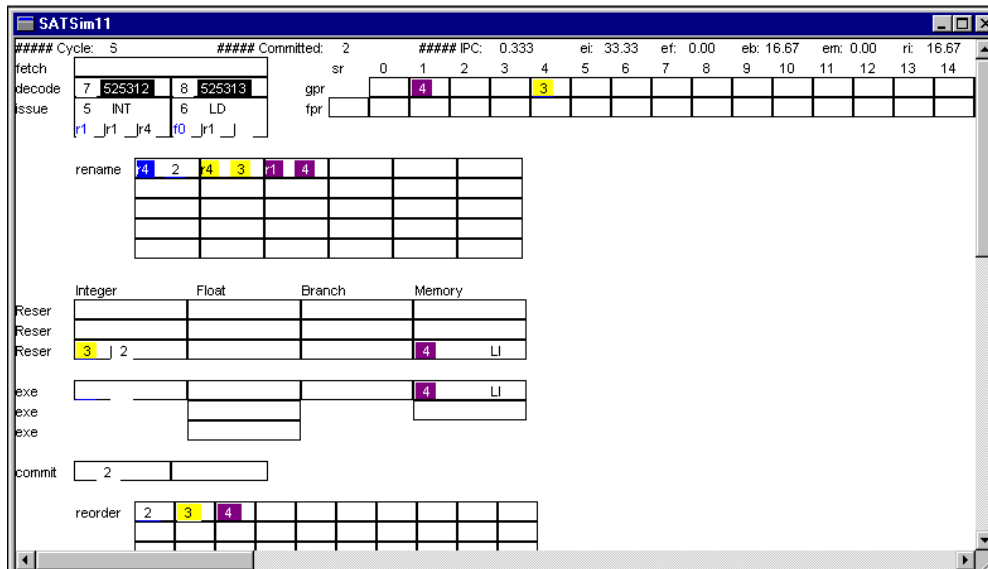


Figura I.7. Ciclul 6 de procesare: evidentiarea mecanismului de *renaming*



$$\begin{aligned}\text{Astfel, în ciclul 6, IPC} &= \frac{2 (\text{instr. 0 si instr. 1})}{6 (\text{cycle})} = 0.333 \text{ instr./tact} \\ e_i &= \frac{2 (\text{instr. 0 si instr. 1})}{6 (\text{cycle})} \cdot 100 [\%] = 33.33 [\%] \\ e_i &= \frac{1 (\text{instr. 1})}{6 (\text{cycle})} \cdot 100 [\%] = 16.67 [\%], \text{ etc.}\end{aligned}$$

Se reaminteste ca instructiunea de index 3 (sra r4, r4, 14) are registru destinatie tot pe r4. Aceasta implica redenumirea registrului r4 cu o noua locatie a buffer-ului de “renaming” – practic cu un nou registru fizic, iar în setul *gpr* la locatia 4 va fi înscris tag-ul 3 – ultima instructiune care-l calculeaza pe r4. Locatia din RenB care detine valoarea anterioara a lui r4 (cea generata de instructiunea 2) ramâne disponibila pâna dupa efectuarea fazei commit întrucât chiar daca aceasta valoare a lui r4 nu trebuie înscrisa în *gpr*, ea trebuie pasata prin “forwarding” statiilor de rezervare care asteapta aceasta valoare pentru a putea lansa instructiunea dependenta (în acest caz 3) în executie propriu-zisa. Observam ca întrucât instructiunea 2 a trecut de faza “exe”, ea a pierdut culoarea din resursele arhitecturii RenB, RB, SR, sau faza commit.

Instructiunea 4 (*Load* cu mod de adresare imediat –**lui r1, 0x10e5**) va genera valoarea viitoare a lui r1, deci acest registru va fi redenumit, indexul instructiunii va fi scris, atât în ReorderBuffer (colorat), cât si în setul *gpr* la locatia 1. Instructiunea 4 va fi expediată în statia de rezervare aferenta instructiunilor cu referire la memorie dar si în prima faza *pipeline* de procesare a unitatii de executie respective. Se reaminteste ca unitatile de executie pentru numere flotante prezinta 3 stagii *pipeline* de procesare iar unitatile cu referire la memorie, 2 stagii. Faptul ca instructiunea 4 este un *Load* cu adresare imediata se observa si din caseta *Memory* a statiilor de rezervare, unde nu exista operand sursa si mnemonica LI pentru diferentierea de un *Load* cu adresare indexata. În faza de *dispatch (issue – 1<sup>st</sup>)* se afla instructiunea 5 cu operanzi întregi (**addu r1,r4,r1**) respectiv instructiunea 6 cu referire la memorie (**l.d f0, -15896(r1)** - încărcare din memorie a unei valori dubla precizie (8o)). De asemenea, are loc decodificarea instructiunilor 7 si 8.

## CICLUL 7.

În **ciclul 7** de procesare se observa ca din buffer-ul de redenumire a disparut (a fost evacuata) instructiunea 2 si au fost alocate alte doua intrari aferente registrilor destinatie pentru instructiunile 5 si 6.

Analog cu cazul registrului r4 în ciclul 6, si pentru registrul r1 are loc o noua redenumire cu instructiunea 5. Acest fapt se observa si în setul de registri generali *gpr* la locatia 1 (instructiunea 5 fiind ultima care calculeaza valoarea lui r1). Totusi instructiunea 4 nu este eliminata din RenB deoarece valoarea registrului r1 care se calculeaza si în acest ciclu de tact este necesara instructiunilor dependente aflate în statiile de rezervare – instructiunea 5 în acest caz. De asemenea, si registrul f0 din *fpr* este redenumit cu instructiunea 6. Cele doua noi instructiuni sunt introduse si în coada FIFO reprezentata de buffer-ul de reordonare.

Întrucât instructiunea 2 a fost complet procesata (a parcurs toate fazele din structura pipeline) instructiunea 3 poate fi deblocata (procesata de unitatea de executie corespunzatoare). Totodata, instructiunea 4 continua executia pe cel de-al doilea nivel pipeline de procesare. Nici o instructiune nu se afla în faza *commit* (toate instructiunile din RB, dar si din RenB “*au culoare*”) deoarece nu sunt valori de înscris în seturile *fpr* si *gpr*.

Instructiunea 5 asteapta în statia de rezervare pâna la disponibilitatea operanzilor (calculati de catre instructiunile 3 si 4). De asemenea, instructiunea 6 asteapta si ea disponibilitatea registrului r1 (calculat de instructiunea 5).

Instructiunile 7 (**dmtc1 r6, f2** cu operanzi flotanti si întregi) si 8 (**addu r6, r0, r0**) sunt pregatite pentru a putea fi expediate spre executie, iar instructiunile 9 si 10 sunt decodificate în tactul curent.

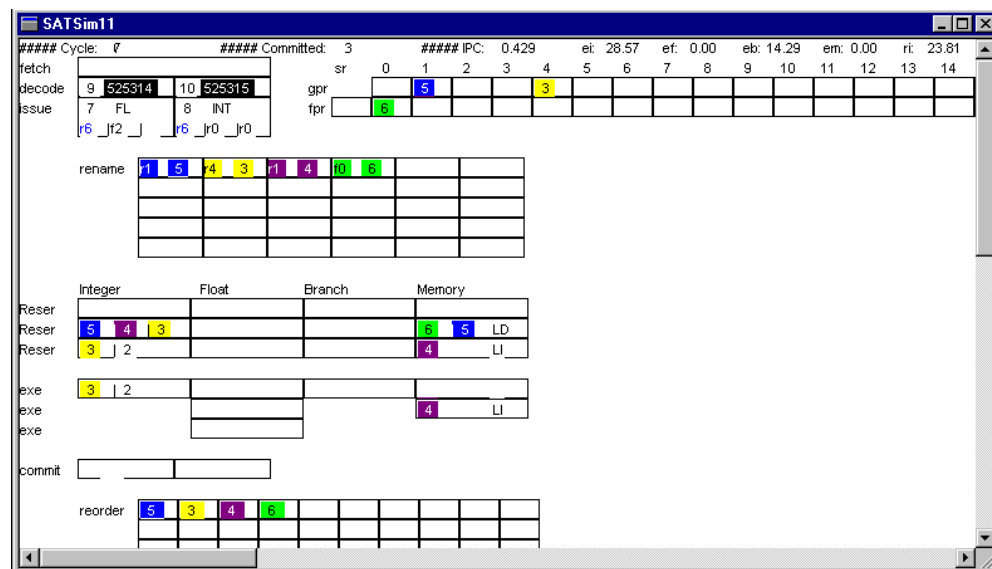


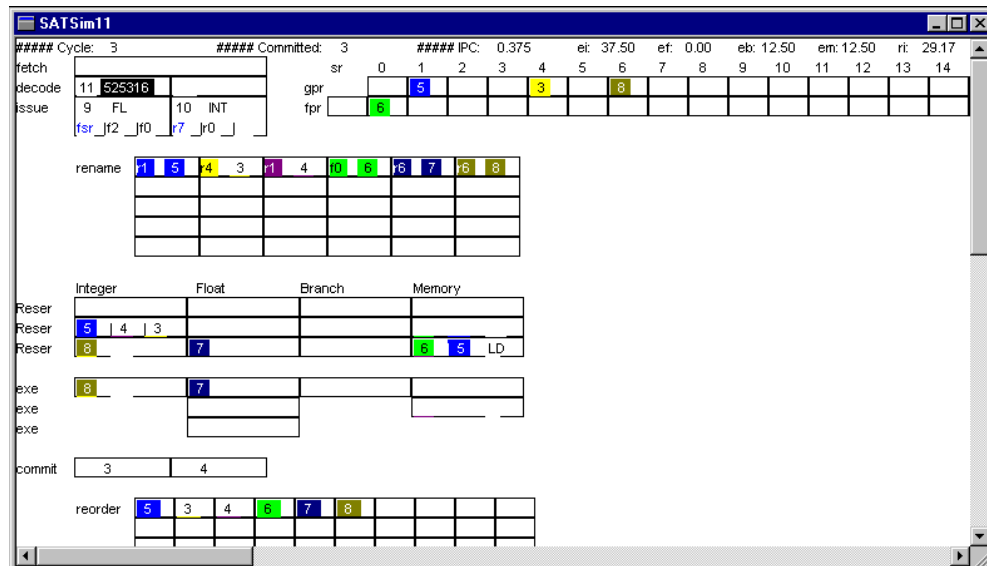
Figura I.8. Ciclul 7 de executie

## CICLUL 8.

**Ciclul 8** de procesare este caracterizat de lansarea în execuție a instrucțiunilor 7 și 8 având operanții disponibili. Chiar și instrucțiunea 5 ar putea fi executată dar din cauza *hazardului structural* existent, aceasta stagnează în SR (o singură unitate funcțională de execuție cu operanți întregi nu poate satisface simultan mai multe instrucțiuni).

Registrul r6 este redenumit de două ori în acest ciclu de tact (i se alocă două intrări în RenB), favorizând eliminarea hazardului WAW dintre instrucțiunile 7 și 8. Întrucât instrucțiunea 8 este cea care scrie ultimul rezultat, în setul de registre generali *gpr* la locația 6 se va afla indexul acestei instrucțiuni. Cele două instrucțiuni sunt înscrise **in-order** și în buffer-ul de reordonare. Instrucțiunile 3 și 4 ajungând în faza *commit* rezulta “*pierderea culorii*” în RenB și RB. Astfel în tactul următor intrările corespunzătoare din cele două buffer-e vor fi evacuate (*factorul superscalar* = 2), iar în *gpr* valoarea registrului r4 va fi disponibilă (corectă).

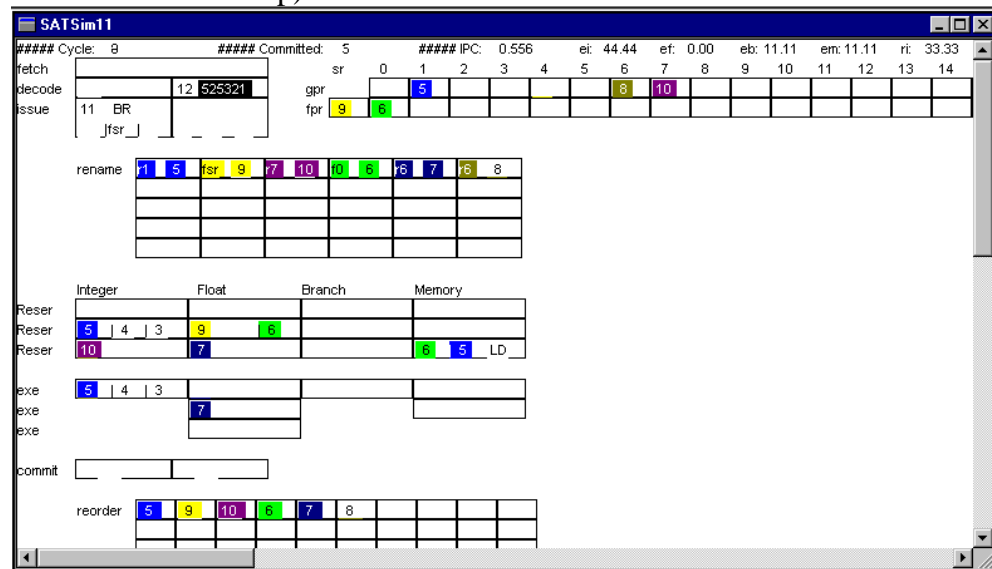
Tot în ciclul 8 de procesare are loc *dispatch*-ul (*issue* - 1<sup>st</sup>) instrucțiunilor 9 (**c.lt.d f0, f2** - cu operanți flotanti și 10 (**addiu r7, r0, 127** - cu operanți întregi și decodificarea instrucțiunii 11 (doar 11 - din cauza că aceasta reprezintă o instrucțiune de salt, iar predicția fiind perfectă nu are sens procesarea instrucțiunilor dincolo de *branch*, ci de la adresa *target* a saltului).



**Figura I.9.** Continutul resurselor arhitecturale în ciclul 8 de procesare

## CICLUL 9.

Pornind oarecum diferit fata de expunerile aferente ciclilor anteriori, se observa instructiunea 11 (de salt) aflata în faza de *dispatch*, iar instructiunea 12 (cea care se decodifica în tactul curent) reprezinta instructiunea destinatie a branch-ului. Întrucât instructiunile 9 si 10 au ca rezultat modificarea registrului de stare flotant (*fsr*) si a registrului întreg *r7*, are loc redenumirea celor doi registri cu indexul instructiunilor în cauza. Sunt alocate astfel doua intrari în buffer-ul de *renaming*. Instructiunea 8 care a fost executata dar nu a fost complet procesata (nu a efectuat si nici nu va efectua faza *commit* pâna când instructiunile din fata sa (5, 6, 7) nu au parcurs faza *commit* si nu au fost evacuate din buffer-ul de reordonare), va astepta în ReorderBuffer si valoarea sa va ramâne (disponibila însa) în registrul de *renaming*, "fara culoare" în RenB si RB dar "cu culoare" în gpr. Registrii care au fost redenumiti vor capata culoare: locatia 7 a *gpr* va fi înscrisa cu instructiunea 10 iar locatia registrului de stare flotant va fi înscrisa cu instructiunea 9. Instructiunea 5, care are operanzii disponibili este lansata în executie în fata instructiunii 10 (si ea cu operanzii disponibili) pentru a permite si altor instructiuni dependente sau nu (din SR) sa poata fi executate, respectiv evacuate din RB daca au fost complet procesate pâna în acest moment. Totodata instructiunea 7 se afla în al doilea stadiu pipeline de procesare (se cunoaste faptul ca instructiunile cu operanzi flotanti sunt consumatoare de timp).



**Figura I.10.** Continutul resurselor arhitecturale în ciclul 9 de procesare

## CICLUL 10.

În ciclul 10 se decodifica instrucțiunile 13 și 14, iar instrucțiunea 12 (**addu r5, r0, r0**) se afla în faza de *issue* -  $1^{st}$ . Instrucțiunea 5, care se afla în faza *commit*, ceea ce înseamnă ca operația aritmetico - logică s-a executat în ciclul anterior, fapt observat prin deblocarea instrucțiunii 6 care va putea fi executată începând cu ciclul de tact următor, dar și prin "*pierderea culorii*" în RenB pentru instrucțiunea 5. Abia în ciclul 11, se va pierde culoarea în locația 1 a setului *gpr* (după efectuarea fazei *commit* de către instrucțiunea 5). Tot atunci se vor evacua intrările din RenB și RB aferente instrucțiunii 5. Instrucțiunea 7 a ajuns în ultimul stadiu pipeline al unității de execuție flotantă, iar în stadiile de rezervare aferente acestui tip de instrucțiune așteaptă și instrucțiunea 9 (dependentă și de rezultatul instrucțiunii 6 încă neprocesate). Instrucțiunea de salt 11 va aștepta în SR corespunzătoare până când operandul sau sursa devine disponibil (deci până după execuția instrucțiunii 9).

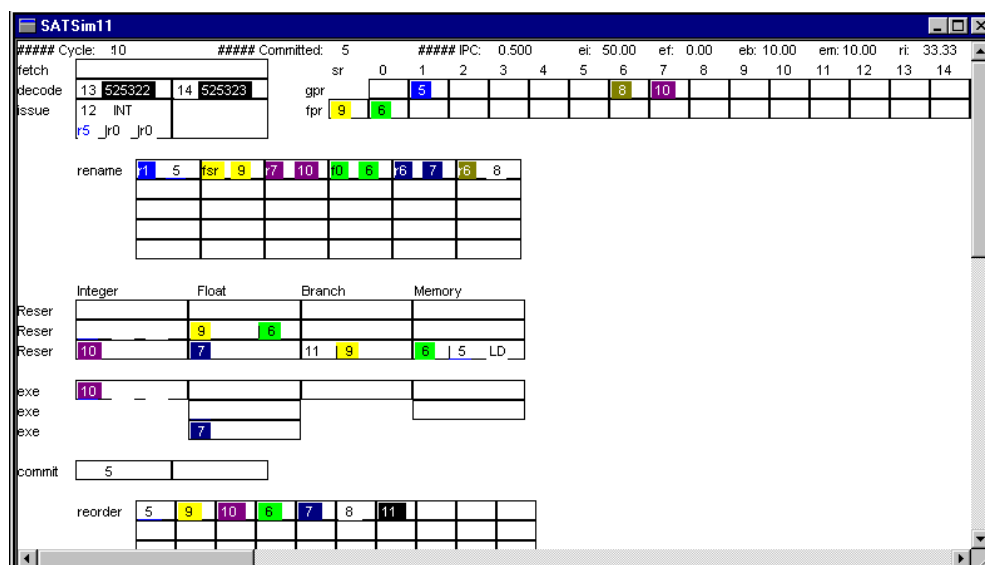


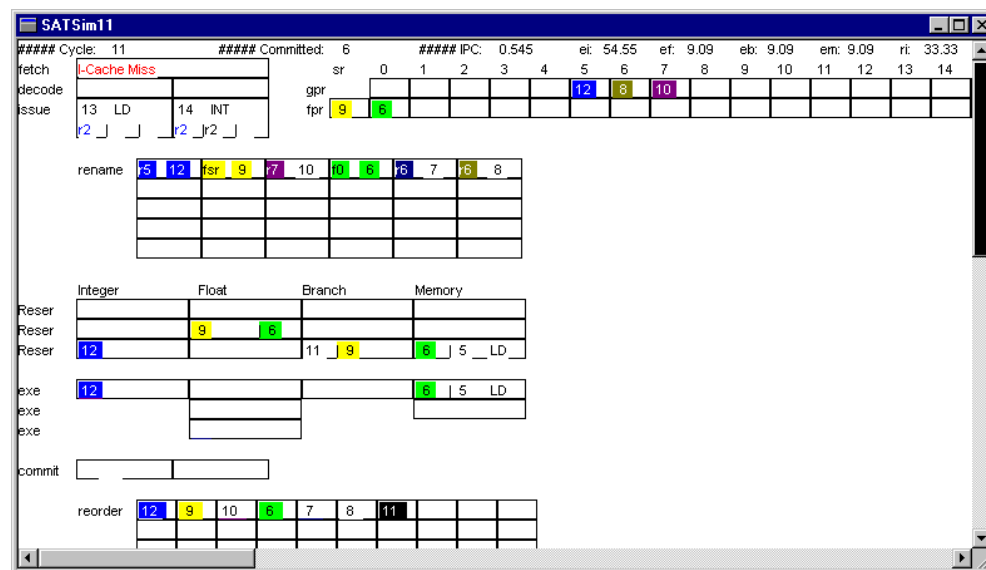
Figura I.11. Ciclul 10 de execuție

## CICLUL 11.

**Ciclul 11** este caracterizat de un eveniment nou, apariția unui miss în cache-ul de instrucțiuni în cadrul fazei de *fetch*. Acest lucru se reflectă în *timing* prin întârzierea cu 4 cicli (*I-Cache Miss Penalty*) a procesului de *fetch* (respectiv *decode*). Totuși, datorită superscalarității și a procesării

pipeline, aceasta întârziere este mascata prin încheierea fazelor de executie pentru instructiunile aflate în diverse resurse arhitecturale (statii de rezervare, unitati functionale de executie, buffer-e de redenumire si reordonare).

Instructiunile 13 (**lui r2, 0x10e5**) si 14 (**addiu r2,r2,-16912**) continua procesarea fiind pregatite pentru rutarea spre statiile de rezervare respectiv unitatile functionale de executie. Instructiunii 12 - întrucât modifica continutul registrului r5 - i se va aloca o intrare în buffer-ul de redenumire iar în setul *gpr* la locatia 5 se va înscrie indexul acesteia (valoarea 12). De asemenea, se aloca o intrare în buffer-ul de reordonare pt. instructiunea 12, care este si lansata automat în executie (având operanzii disponibili). Starea instructiunii 11 nu se modifica ea așteptând în SR disponibilitatea operandului sau (calculat de instructiunea 9). În acest ciclu de tact instructiunea 7 a încheiat faza de executie, însa valoarea registrului 6 ramâne în locatia buffer-ului de redenumire (nu efectueaza înca *commit* decât dupa instructiunea 6 si dupa retragerea acesteia din buffer-ul de reordonare). Afirmatiile anterioare sunt justificate prin "pierderea culorii" în RenB si RB pentru instructiunea 7 si "pastrarea culorii" pentru instructiunea 6, care în acest ciclu se afla pe primul nivel pipeline al fazei de executie.



**Figura I.12.** Ciclul 11 de procesare: aparitia primului miss în Cache-ul de Instructiuni

## ANEXA II

### PSEUDO-TRANSLATOR PENTRU SATSIM

---

#### II.1. NOTIUNI GENERALE

Acest pseudo translator/simulator (**PseudoExSAT**) este util în realizarea trace-urilor necesare simulatorului SATSim. Se recomanda ca în toate aceste trace-uri PC-ul primei instructiuni sa fie egal cu **401EB8h**, lucru realizabil cu ajutorul translatorului PseudoExSAT. **Pseudo translatorul/simulatorul** a fost realizat în limbajul C++ Builder versiunea 4.0.

#### II.2. MOD DE UTILIZARE

Interfata cu utilizatorul se bazeaza în principal pe urmatorul meniu:

- **File** având urmatoarele submeniuri:
  - **New (F1)** - permite editare de cod asamblare pentru translatore spre fisierul trace
  - **Open** - permite deschiderea de fisiere asamblare (doar cu extensia **.s**)
  - **Exit** - determina închiderea aplicatiei.
- **Edit** cuprinde urmatoarele optiuni cu semnificatie evidenta:
  - **Cut**
  - **Copy**
  - **Paste**
  - **Delete**
  - **Select all**

- **Compileaza** – realizeaza translatarea efectiva a codului din limbaj de asamblare MIPS în fisier de tip trace (**.tra**). Translatarea se bazeaza pe executia efectiva a codului cu valorificarea continutului registrilor.
  - **Compileaza**
  - **Citeste programul**
- **Registri**
  - **Initializeaza registrii**
  - **Reseteaza registrii**
  - **Initializeaza registrii cu valori aleatoare**
- **Clear** (curata fereastra din dreapta în care se afla codul desfasurat - fisierul trace - si fereastra de jos în care sunt prezentati pasii procesului de translatare).
- **Find Text** (cauta un text în cadrul interfetei – procedura standard a sistemului de operare).



**Figura II.1.** Fereastra principala a translatorului: meniul si zonele de editare

Interfata cu utilizatorul este împartita în 4 ferestre (zone):

1. Prima fereastra, situata stânga sus, initial este goala. Fereastra prezinta continutul fiecarui registru din setul de registri generali la încărcarea unui fisier de asamblare.
2. A doua fereastra, situata la mijloc sus, initial este goala. În ea este afisat codul sursa încărcat din fisierul asamblare MIPS cu extensia **.s**.
3. A treia fereastra, situata în dreapta sus, initial este si ea goala. Aici se va înscrie dupa executia codului (la translatare) continutul fisierului trace.



4. A patra fereastră situată în partea de jos, este inițial goală. În ea se afișează la traducare pași și instrucțiunile decodificate.

Un instantaneu cu Pseudo-Translatorul după deschiderea unui fișier sursă și executia codului acestuia arată ca în figura următoare.

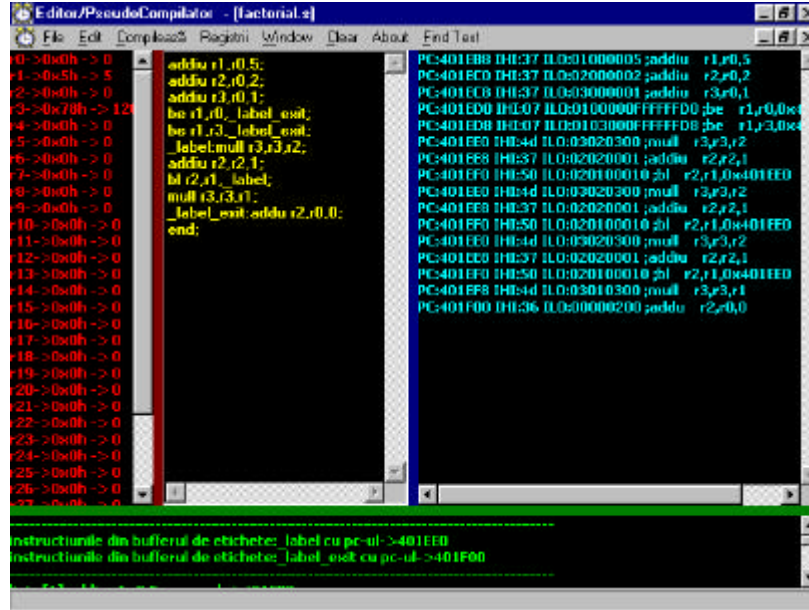


Figura II.2. Continutul zonelor de editare la executia programului de calcul al factorialului de 5

### II.3.MOD DE REALIZARE

Translatorul poate interpreta în faza actuală următoarele instrucțiuni: **addu, addiu, mull, div, be, bne, bl, bg, ble, bge, j, subu, subiu, and, or, xor, sll, srr, mulli, divi, andi, ori, xori.**

- Instrucțiunile *addu, addiu, mull, div* etc au fost implementate prin operațiile de adunare, înmulțire, împărțire, etc a registrelor sursă situate într-un tablou de 32 de elemente.
- Instrucțiunile de ramificație *be, bne, bl, bg, ble, bge, j* sunt tratate separat față de celelalte instrucțiuni. La prima parcurgere a codului este calculat indexul etichetei (indexul unde se va face saltul) și reținut într-o variabilă (gen *history*) astfel ca la următoarele treceri prin buclă indexul de salt este preluat din variabila *history*.

La compilare se citește din fișierul \*.s codul sursă și se decodifică fiecare linie, se determină registrele sursă și destinație precum și tipul instrucțiunii. Toate acestea se memorează în tablourile de registre, respectiv în tabloul de instrucțiuni aflate în *clasa MDIEdit*.

Decodificarea instrucțiunilor se face la compilare după ce s-au introdus într-un tablou toate instrucțiunile din codul sursă cu ajutorul funcției *trans()* care are implementată toate tipurile de operații (de tip **RRR** (atât sursa cât și destinația sunt registre), **RRV** (o sursă este registru iar cealaltă este valoare), **RRE** (unde operanzii sursă sunt în registre iar **E** reprezintă eticheta la care se va face saltul). Codificarea nu este standardizată ci proprie implementării PseudoExSAT.

Funcția *trans()* apelează funcțiile *executa\_instr*, *executa\_instrimm* care realizează efectiv operația de adunare, scădere, înmulțire etc.

După decodificare (apelul funcției *trans()*) se testează dacă instrucțiunea este de ramificație. În caz afirmativ se determină dacă se face saltul, cu excepția instrucțiunii *Jump* în cazul căreia saltul se face necondiționat. Dacă saltul se face, se calculează adresa de salt și se memorează într-un tablou de tip *history* la indexul instrucțiunii care reprezintă numărul instrucțiunii din codul sursă (numărul rândului pe care se află instrucțiunea de salt) doar la prima trecere. Dacă saltul nu se face se scrie instrucțiunea în fișierul *trace* și se incrementează variabila care reține numărul instrucțiunii.

La a doua trecere printr-o buclă, dacă condiția instrucțiunii de salt rămâne nemodificată se consideră că acea buclă este o buclă infinită iar utilizatorul va fi avertizat cu un mesaj de eroare care va opri generarea trace-ului.

*Condiția necesară* pentru ca trace-ul să fie corect făcut este că după fiecare instrucțiune să se pună semnul punct-virgulă iar la sfârșitul programului scris în asamblare să apară cuvântul cheie **end** urmat de semnul punct-virgulă. O altă condiție esențială pentru ca simulatorul *SATSim* să funcționeze corect este că fișierul *.trace* să conțină ca delimitator între 2 instrucțiuni codul ASCII în hexazecimal al caracterului **newline (0Ah)**.

În ultima fază a translării - după decodificare și detectia instrucțiunilor de ramificație - se face scrierea în fișierul *.tra* a conținutului *ferestrei dreapta sus*. Scrierea instrucțiunilor desfășurate în fișierul trace se face prin deschiderea fișierului la nivel binar, pentru a elimina în cazul delimitatorului **newline** a octetului **0Dh**.

Fișierul *.tra* va avea același nume cu fișierul sursă *.s*.

Alături de aplicație se găsesc fișierele sursă *factorial.s* (calculează factorialul de 5 și depune rezultatul în registrul R3), *min\_3\_elm.s* (minimul dintre 3 elemente și depune rezultatul în registrul r4).

## ANEXA III

### DEFINIREA SETULUI DE INSTRUCȚIUNI

Aceasta anexa ilustrează toate instrucțiunile aferente arhitecturii *SimpleScalar* (opcode, formatul în asamblare și semantica). Semantica instrucțiunilor este exprimată ca o expresie C, care folosește operatorii și operanzii descriși în tabelul III.1 al anexei de față. Operanzii care nu apar în tabel se regăsesc într-unul din câmpurile descrise în formatul instrucțiunii (figura 8.7 din capitolul 8 al prezentei lucrări). Pentru fiecare instrucțiune adresa următoarei instrucțiuni NPC ("next PC") se obține din valoarea curentă (CPC) la care se adaugă 8 (CPC+8), dacă nu este altfel specificat.

#### III.1. INSTRUCȚIUNILE DE CONTROL.

| Mne-monica              | Opcode | Format     | Semantica                                                          | Descriere                                                                                |
|-------------------------|--------|------------|--------------------------------------------------------------------|------------------------------------------------------------------------------------------|
| <b>J</b>                | 0x01   | J target   | SET_NPC ((CPC & 0xf0000000)   (TARGET << 2))                       | Salt neconditionat la o adresa absolută                                                  |
| <b>JAL</b> <sup>1</sup> | 0x02   | JAL target | SET_NPC ((CPC & 0xf0000000)   (TARGET << 2))<br>SET_GPR(31, CPC+8) | Salt neconditionat la o adresa absolută cu salvarea adresei de revenire (Apel subrutină) |
| <b>JR</b>               | 0x03   | JR rs      | TALIGN(GPR(RS))<br>SET_NPC(GPR(RS))                                | Salt neconditionat la adresa specificată de registrul sursă                              |
| <b>JALR</b>             | 0x04   | JALR rs    | TALIGN(GPR(RS))<br>SET_GPR(RD, CPC+8)<br>SET_NPC(GPR(RS))          | Salt neconditionat la adresa specificată de registrul sursă cu                           |

<sup>1</sup> Registrul ra (\$31) păstrează adresa de revenire dintr-o procedură după apelul instrucțiunii *jal*. \$ra e necesar a fi salvat pe stivă doar când subrutina apelează alte subrutine.

|             |      |                    |                                                                             |                                              |
|-------------|------|--------------------|-----------------------------------------------------------------------------|----------------------------------------------|
|             |      |                    |                                                                             | salvarea legaturii prin registru             |
| <b>BEQ</b>  | 0x05 | BEQ rs, rt, offset | If (GPR(RS)==GPR(RT))<br>SET_NPC(CPC+8+(OFFS ET<<2))<br>else SET_NPC(CPC+8) | Salt la offset daca rs=rt                    |
| <b>BNE</b>  | 0x06 | BNE rs, rt, offset | If (GPR(RS)!=GPR(RT))<br>SET_NPC(CPC+8+(OFFS ET<<2))<br>else SET_NPC(CPC+8) | Salt la offset daca rs nu este egal cu rt    |
| <b>BLEZ</b> | 0x07 | BLEZ rs, offset    | If (GPR(RS)<=0)<br>SET_NPC(CPC+8+(OFFS ET<<2))<br>else SET_NPC(CPC+8)       | Salt la offset daca rs <= 0                  |
| <b>BGTZ</b> | 0x08 | BGTZ rs, offset    | If (GPR(RS)>0)<br>SET_NPC(CPC+8+(OFFS ET<<2))<br>else SET_NPC(CPC+8)        | Salt la offset daca rs > 0                   |
| <b>BLTZ</b> | 0x09 | BLTZ rs, offset    | If (GPR(RS)<0)<br>SET_NPC(CPC+8+(OFFS ET<<2))<br>else SET_NPC(CPC+8)        | Salt la offset daca rs < 0                   |
| <b>BGEZ</b> | 0x0a | BLTZ rs, offset    | If (GPR(RS)>=0)<br>SET_NPC(CPC+8+(OFFS ET<<2))<br>else SET_NPC(CPC+8)       | Salt la offset daca rs >=0                   |
| <b>BC1F</b> | 0x0b | BC1F offset        | If (!FCC)<br>SET_NPC(PC+8+(OFFSE T<<2))<br>else SET_NPC(CPC+8)              | Salt in virgula mobila pe conditie falsa     |
| <b>BC1T</b> | 0x0c | BC1T offset        | If (FCC)<br>SET_NPC(PC+8+(OFFSE T<<2))<br>else SET_NPC(CPC+8)               | Salt in virgula mobila pe conditie adevarata |

### III.2.INSTRUCTIUNILE LOAD/ STORE

| Mne-monica | Opcode | Format                       | Semantica                                     | Descriere                                      |
|------------|--------|------------------------------|-----------------------------------------------|------------------------------------------------|
| <b>LB</b>  | 0x20   | LB rt, offset(rs)<br>inc_dec | SET_GPR(RT, READ_SIGNED_BYTE(GPR(RS)+OFFSET)) | Incarca un octet cu semn de la adresa relativa |
| <b>LB</b>  | 0xc0   | LB rt,                       | SET_GPR(RT,                                   | Incarca un octet cu semn                       |

|            |      |                                  |                                                                                                      |                                                                                       |
|------------|------|----------------------------------|------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
|            |      | (rs+rd)<br>inc_dec               | READ_SIGNED_BYTE(<br>GPR(RS)+GPR(RD)))                                                               | de la adresa din rs<br>indexata cu o valoare din<br>rd                                |
| <b>LBU</b> | 0x22 | LBU rt,<br>offset(rs)<br>inc_dec | SET_GPR(RT,<br>READ_UNSIGNED_BYTE(<br>GPR(RS)+OFFSET))                                               | Incarca un octet fara<br>semn de la adresa<br>relativa                                |
| <b>LBU</b> | 0xc1 | LB rt,<br>(rs+rd)<br>inc_dec     | SET_GPR(RT,<br>READ_UNSIGNED_BYTE(<br>GPR(RS)+GPR(RD)))                                              | Incarca un octet fara<br>semn de la adresa din rs<br>indexata cu o valoare din<br>rd  |
| <b>LH</b>  | 0x24 | LH rt,<br>offset(rs)<br>inc_dec  | SET_GPR(RT,<br>READ_SIGNED_HALF(<br>GPR(RS)+OFFSET))                                                 | Incarca un intreg cu<br>semn de la adresa<br>relativa                                 |
| <b>LH</b>  | 0xc2 | LH rt,<br>(rs+rd)<br>inc_dec     | SET_GPR(RT,<br>READ_SIGNED_HALF(<br>GPR(RS)+GPR(RD)))                                                | Incarca un intreg cu<br>semn de la adresa din rs<br>indexata cu o valoare din<br>rd   |
| <b>LHU</b> | 0x26 | LHU rt,<br>offset(rs)<br>inc_dec | SET_GPR(RT,<br>READ_UNSIGNED_HALF(<br>GPR(RS)+OFFSET))                                               | Incarca un intreg fara<br>semn de la adresa<br>relativa                               |
| <b>LHU</b> | 0xc3 | LHU rt,<br>(rs+rd)<br>inc_dec    | SET_GPR(RT,<br>READ_UNSIGNED_HALF(<br>GPR(RS)+GPR(RD)))                                              | Incarca un intreg fara<br>semn de la adresa din rs<br>indexata cu o valoare din<br>rd |
| <b>LW</b>  | 0x28 | LW rt,<br>offset(rs)<br>inc_dec  | SET_GPR(RT,<br>READ_WORD(GPR(RS)<br>+OFFSET))                                                        | Incarca un cuvant de la<br>adresa relativa                                            |
| <b>LW</b>  | 0xc4 | LW rt,<br>(rs+rd)<br>inc_dec     | SET_GPR(RT,<br>READ_WORD(GPR(RS)<br>+GPR(RD)))                                                       | Incarca un cuvant de la<br>adresa din rs indexata cu<br>o valoare din rd              |
| <b>DLW</b> | 0x29 | DLW rt,<br>offset(rs)<br>inc_dec | SET_GPR(RT,<br>READ_WORD(GPR(RS)<br>+OFFSET))<br>SET_GPR(RT+1,<br>READ_WORD(GPR(RS)<br>+OFFSET+4))   | Incarca un dublu cuvant<br>de la adresa relativa                                      |
| <b>DLW</b> | 0xce | DLW rt,<br>(rs+rd)<br>inc_dec    | SET_GPR(RT,<br>READ_WORD(GPR(RS)<br>+GPR(RD)))<br>SET_GPR(RT+1,<br>READ_WORD(GPR(RS)<br>+GPR(RD)+4)) | Incarca un dublu cuvant<br>de la adresa din rs<br>indexata cu o valoare din<br>rd     |
| <b>L.S</b> | 0x2a | L.S ft,<br>offset(rs)<br>inc_dec | SET_FPR_L(FT,<br>READ_WORDS(GPR(RS)<br>+OFFSET))                                                     | Incarca un cuvant intr-<br>un registru in virgula<br>mobila de la adresa<br>relativa  |
| <b>L.S</b> | 0xc5 | L.S ft,<br>(rs+rd)               | SET_FPR_L(FT,<br>READ_WORD(GPR(RS)                                                                   | Incarca un cuvant intr-un<br>registru in virgula mobila                               |

|            |      |                            |                                                                                                                |                                                                                                             |
|------------|------|----------------------------|----------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|
|            |      | inc_dec                    | +GPR(RD)))                                                                                                     | de la adresa din rs indexata cu o valoare din rd                                                            |
| <b>L.D</b> | 0x2b | L.D ft, offset(rs) inc_dec | SET_FPR_L(FT, READ_WORD(GPR(RS)+OFFSET))<br>SET_FPR_L(FT+1, READ_WORD(GPR(RS)+OFFSET+4))                       | Incarca un dublu cuvant intr-un registru in virgula mobila de la adresa relativa                            |
| <b>L.D</b> | 0xcf | L.D ft, (rs+rd) inc_dec    | SET_FPR_L(FT, READ_WORD(GPR(RS)+GPR(RD)))<br>SET_FPR_L(RT+1, READ_WORD(GPR(RS)+GPR(RD)+4))                     | Incarca un dublu cuvant intr-un registru in virgula mobila de la adresa din rs indexata cu o valoare din rd |
| <b>LWL</b> | 0x2c | LWL rt, offset(rs)         | Vezi ss.def pentru o descriere detaliata a acestei semantici<br><b>Obs:</b> LWR nu suporta pre-/post, inc/dec. | Incarca cuvant la stanga, adresare relativa                                                                 |
| <b>LWR</b> | 0x2d | LWR rt, offset(rs)         | Vezi ss.def pentru o descriere detaliata a acestei semantici<br><b>Obs:</b> DWR nu suporta pre-/post, inc/dec. | Incarca cuvant la dreapta, adresare relativa                                                                |
| <b>SB</b>  | 0x30 | SB rt, offset(rs) inc_dec  | WRITE_BYTE(GPR(RT), GPR(RS)+OFFSET)                                                                            | Memoreaza un octet la adresa relativa                                                                       |
| <b>SB</b>  | 0xc6 | SB rt, (rs+rd) inc_dec     | WRITE_BYTE(GPR(RT), GPR(RS)+GPR(RD))                                                                           | Memoreaza un octet la adresa din rs indexata cu o valoare din rd                                            |
| <b>SH</b>  | 0x32 | SH rt, offset(rs) inc_dec  | WRITE_HALF(GPR(RT), GPR(RS)+OFFSET)                                                                            | Memoreaza un intreg la adresa relativa                                                                      |
| <b>SH</b>  | 0xc7 | SH rt, (rs+rd) inc_dec     | WRITE_HALF(GPR(RT), GPR(RS)+GPR(RD))                                                                           | Memoreaza un intreg la adresa din rs indexata cu o valoare din rd                                           |
| <b>SW</b>  | 0x34 | SW rt, offset(rs) inc_dec  | WRITE_WORD(GPR(RT), GPR(RS)+OFFSET)                                                                            | Memoreaza un cuvant la adresa relativa                                                                      |
| <b>SW</b>  | 0xc8 | SW rt, (rs+rd) inc_dec     | WRITE_WORD(GPR(RT), GPR(RS)+GPR(RD))                                                                           | Memoreaza un cuvant la adresa din rs indexata cu o valoare din rd                                           |
| <b>DSW</b> | 0x35 | DSW rt, offset(rs) inc_dec | WRITE_WORD(GPR(RT), GPR(RS)+OFFSET)<br>WRITE_WORD(GPR(RT)+1, GPR(RS)+OFFSET+4)                                 | Memoreaza un dublu cuvant la adresa relativa                                                                |

|            |      |                                  |                                                                                                                          |                                                                                                                         |
|------------|------|----------------------------------|--------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------|
| <b>DSW</b> | 0xd0 | DLW rt,<br>(rs+rd)<br>inc_dec    | WRITE_WORD(GPR(RT)<br>, GPR(RS)+GPR(RD))<br>WRITE_WORD(GPR(RT<br>+1),<br>GPR(RS)+GPR(RD)+4)                              | Memoreaza un dublu<br>cuvant la adresa din rs<br>indexata cu o valoare din<br>rd                                        |
| <b>DSZ</b> | 0x38 | DSW rt,<br>offset(rs)<br>inc_dec | WRITE_WORD(0,<br>GPR(RS)+GPR(RD))<br>WRITE_WORD(0,<br>GPR(RS)+GPR(RD)+4)                                                 | Memoreaza un dublu 0,<br>adresare relativa                                                                              |
| <b>DSZ</b> | 0xd1 | DLW rt,<br>(rs+rd)<br>inc_dec    | WRITE_WORD(0,<br>GPR(RS)+GPR(RD))<br>WRITE_WORD(0,<br>GPR(RS)+GPR(RD)+4)                                                 | Memoreaza un dublu 0,<br>adresare indexata                                                                              |
| <b>S.S</b> | 0x36 | S.S ft,<br>offset(rs)<br>inc_dec | WRITE_WORD(FPR_L(F<br>T),GPR(RS)+OFFSET)                                                                                 | Memoreaza un cuvant<br>dintr-un registru in<br>virgula mobila la adresa<br>relativa                                     |
| <b>S.S</b> | 0xc9 | S.D ft,<br>(rs+rd)<br>inc_dec    | WRITE_WORD(FPR_L(F<br>T),GPR(RS)+GPR(RD))                                                                                | Memoreaza un cuvant<br>dintr-un registru in<br>virgula mobila la adresa<br>din rs indexata cu o<br>valoare din rd       |
| <b>S.D</b> | 0x37 | S.D ft,<br>offset(rs)<br>inc_dec | WRITE_WORD(FPR_L(F<br>T),GPR(RS)+OFFSET))<br>WRITE_WORD(FPR_L(F<br>T+1),GPR(RS)+OFFSET+<br>4))                           | Memoreaza un dublu<br>cuvant dintr-un registru<br>in virgula mobila la<br>adresa relativa                               |
| <b>S.D</b> | 0xd2 | S.D ft,<br>(rs+rd)<br>inc_dec    | SET_FPR_L(FT,<br>READ_WORD(GPR(RS)<br>+GPR(RD)))<br>SET_FPR_L(RT+1,<br>READ_WORD(GPR(RS)<br>+GPR(RD)+4))                 | Memoreaza un dublu<br>cuvant dintr-un registru<br>in virgula mobila la<br>adresa din rs indexata cu<br>o valoare din rd |
| <b>SWL</b> | 0x39 | LWL rt,<br>offset(rs)            | Vezi ss.def pentru o<br>descriere detaliata a acestei<br>semantici<br><b>Obs:</b> SWR nu suporta pre-<br>/post, inc/dec. | Memoreaza cuvant la<br>stanga, adresare relativa                                                                        |
| <b>SWR</b> | 0x3a | SWR rt,<br>offset(rs)            | Vezi ss.def pentru o<br>descriere detaliata a acestei<br>semantici<br><b>Obs:</b> SWR nu suporta pre-<br>/post, inc/dec. | Incarca cuvant la<br>dreapta, adresare relativa                                                                         |

## III.3. INSTRUCȚIUNILE PENTRU ÎNTREGI

| Mne-monica   | Opcode | Format            | Semantica                                                                                                   | Descriere                                              |
|--------------|--------|-------------------|-------------------------------------------------------------------------------------------------------------|--------------------------------------------------------|
| <b>ADD</b>   | 0x40   | ADD<br>rd,rs,rt   | OVER(GPR(RS),GPR(RT))<br>SET_GPR(RD,GPR(RS)+GPR(RT))                                                        | Adunare cu semn cu verificarea depasirii               |
| <b>ADDI</b>  | 0x41   | ADDI<br>rd,rs,IMM | OVER(GPR(RS),IMM)<br>SET_GPR(RD,GPR(RS)+IMM)                                                                | Adunare imediata cu verificarea depasirii              |
| <b>ADDU</b>  | 0x42   | ADD<br>rd,rs,rt   | SET_GPR(RD,GPR(RS)+GPR(RT))                                                                                 | Adunare fara semn fara verificarea depasirii           |
| <b>ADDIU</b> | 0x43   |                   | SET_GPR(RD,GPR(RS)+IMM)                                                                                     | Adunare imediata fara verificarea depasirii, fara semn |
| <b>SUB</b>   | 0x44   | SUB<br>rd,rs,rt   | OVER(GPR(RS),GPR(RT))<br>SET_GPR(RD,GPR(RS)-GPR(RT))                                                        | Scadere cu semn cu verificarea depasirii               |
| <b>SUBU</b>  | 0x45   | SUBU<br>rd,rs,rt  | SET_GPR(RD,GPR(RS)-GPR(RT))                                                                                 | Scadere fara semn fara verificarea depasirii           |
| <b>MULT</b>  | 0x46   | MULT<br>rs,rt     | SET_HI((RS*RT)/(1<<32))<br>SET_LO((RS*RT)%(1<<32))                                                          | Inmultire cu semn                                      |
| <b>MULTU</b> | 0x47   | MULTU<br>rs,rt    | SET_HI(((unsigned)RS*(unsigned)RT)/(1<<32))<br>SET_LO(((unsigned)RS*(unsigned)RT)%(1<<32))                  | Inmultire fara semn                                    |
| <b>DIV</b>   | 0x48   | DIV rs,rt         | DIV0(GPR(RT))<br>SET_LO(GPR(RS)/GPR(RT))<br>SET_HI(GPR(RS)%GPR(RT))                                         | Impartire cu semn                                      |
| <b>DIVU</b>  | 0x49   | DIVU<br>rs,rt     | DIV0(GPR(RT))<br>SET_LO((unsigned)GPR(RS)/(unsigned)GPR(RT))<br>SET_HI((unsigned)GPR(RS)%(unsigned)GPR(RT)) | Impartire fara semn                                    |
| <b>MFHI</b>  | 0x4a   | MFHI rd           | SET_GPR(RD,HI)                                                                                              | Muta din registrul HI in registrul rd                  |
| <b>MTHI</b>  | 0x4b   | MTHI rd           | SET_HI(GPR(RS))                                                                                             | Muta in registrul HI                                   |
| <b>MFLO</b>  | 0x4c   | MFLO rd           | SET_GPR(RD,LO)                                                                                              | Muta din registrul LO                                  |
| <b>MTLO</b>  | 0x4d   | MTLO<br>rd        | SET_LO(GPR(RS))                                                                                             | Muta in registrul LO                                   |
| <b>AND</b>   | 0x4e   | AND               | SET_GPR(RD,GPR(RS)&GPR(RT))                                                                                 | Si logic                                               |



|              |      |                        |                                                         |                                            |
|--------------|------|------------------------|---------------------------------------------------------|--------------------------------------------|
|              |      | rd,rs,rt               | PR(RT))                                                 |                                            |
| <b>ANDI</b>  | 0x4f | ANDI<br>rd,rs,<br>UIMM | SET_GPR(RD,GPR(RS)&UIMM)                                | Si logic                                   |
| <b>OR</b>    | 0x50 | OR<br>rd,rs,rt         | SET_GPR(RD,GPR(RS) GPR(RT))                             | Sau logic                                  |
| <b>ORI</b>   | 0x51 | ORI<br>rd,rs,<br>UIMM  | SET_GPR(RD,GPR(RS) UIMM)                                | Sau logic imediat                          |
| <b>XOR</b>   | 0x52 | XOR<br>rd,rs,rt        | SET_GPR(RD,GPR(RS)^GPR(RT))                             | Sau                                        |
| <b>XORI</b>  | 0x53 | XORI<br>rd,rt,<br>UIMM | SET_GPR(RD,GPR(RT) UIMM)                                | Sau logic imediat                          |
| <b>NOR</b>   | 0x54 | NOR<br>rd,rs,rt        | SET_GPR(RD,-(GPR(RS) GPR(RD)))                          | Sau negat logic                            |
| <b>SLL</b>   | 0x55 | SLL<br>rd,rt,<br>SHAMT | SET_GPR(RD,GPR(RT)<<(SHAMT))                            | Deplasare logica la stanga                 |
| <b>SLLV</b>  | 0x56 | SLLV<br>rd,rt,rs       | SET_GPR(RD,GPR(RT)<<(GPR(RS)&0x1f))                     | Deplasare logica la stanga variabila       |
| <b>SRL</b>   | 0x57 | SRL<br>rd,rt,Shamt     | SET_GPR(RD,GPR(RT)>>(SHAMT))                            | Deplasare logica la dreapta                |
| <b>SRLV</b>  | 0x58 | SRLV<br>rd,rt,rs       | SET_GPR(RD,GPR(RT)>>(GPR(RS)&0x1f))                     | Deplasare logica la dreapta variabila      |
| <b>SRA</b>   | 0x59 | SRA<br>rd,rt,Shamt     | SET_GPR(RD,SEX(GPR(RT))>>SHAMT,31-SHAMT)                | Deplasare aritmetica la dreapta            |
| <b>SRAV</b>  | 0x5a | SRAV<br>rd, rt, rs     | SET_GPR(RD,SEX(GPR(RT))>>GPR(RS),31-(GPR(RD)&0x1f))     | Deplasare aritmetica la dreapta variabila  |
| <b>SLT</b>   | 0x5b | SLT<br>rd,rs,rt        | SET_GPR(RD,(GPR(RS)<GPR(RT))?1:0)                       | Seteaza rd daca rs< rt                     |
| <b>SLTI</b>  | 0x5c | SLTI<br>rd,rs,imm      | SET_GPR(RD,(GPR(RS)<IMM)?1:0)                           | Seteaza rd daca rs< imm                    |
| <b>SLTU</b>  | 0x5d | SLTU<br>rd,rs,rt       | SET_GPR(RD,(((unsigned)GPR(RS)<(unsigned)GPR(RT))?1:0)) | Seteaza registrul rd daca rs<rt fara semn  |
| <b>SLTIU</b> | 0x5e | SLTIU<br>rd,<br>rs,imm | SET_GPR(RD,(((unsigned)GPR(RS)<(unsigned)IMM)?1:0))     | Seteaza registrul rd daca rs<imm fara semn |

## III.4. INSTRUCȚIUNILE ÎN VIRGULA MOBILA

| Mne-monica   | Opcode | Format            | Semantica                                                                                             | Descriere                                        |
|--------------|--------|-------------------|-------------------------------------------------------------------------------------------------------|--------------------------------------------------|
| <b>ADD.S</b> | 0x70   | ADD.S<br>fd,fs,ft | FPALING(FD)<br>FPALING(FS)<br>FPALING(FT)<br>SET_FPR_F(FD,FPR_F(FS)+<br>FPR_F(FT))                    | Adunare in<br>virgula mobila<br>simpla preizie   |
| <b>ADD.D</b> | 0x71   | ADD.D<br>fd,fs,ft | FPALING(FD)<br>FPALING(FS)<br>FPALING(FT)<br>SET_FPR_D(FD,FPR_D(FS)<br>+FPR_D(FT))                    | Adunare in<br>virgula mobila<br>dubla preizie    |
| <b>SUB.S</b> | 0x72   | SUB.S<br>fd,fs,ft | FPALING(FD)<br>FPALING(FS)<br>FPALING(FT)<br>SET_FPR_F(FD,FPR_F(FS)-<br>FPR_F(FT))                    | Scadere in virgula<br>mobila simpla<br>preizie   |
| <b>SUB.D</b> | 0x73   | SUB.D<br>fd,fs,ft | FPALIGN(FD)<br>FPALIGN(FS)<br>FPALIGN(FT)<br>SET_FPR_D(FD,FPR_D(FS)-<br>FPR_D(FT))                    | Scadere in virgula<br>mobila dubla<br>preizie    |
| <b>MUL.S</b> | 0x74   | MUL.S<br>fd,fs,ft | FPALIGN(FD)<br>FPALIGN(FS)<br>FPALIGN(FT)<br>SET_FPR_F(FD,FPR_F(FS)*<br>FPR_F(FT))                    | Inmultire in<br>virgula mobila<br>simpla preizie |
| <b>MUL.D</b> | 0x75   | MUL.D<br>fd,fs,ft | FPALIGN(FD)<br>FPALIGN(FS)<br>FPALIGN(FT)<br>SET_FPR_D(FD,FPR_D(FS)<br>*FPR_D(FT))                    | Inmultire in<br>virgula mobila<br>dubla preizie  |
| <b>DIV.S</b> | 0x76   | DIV.S<br>fd,fs,ft | FPALIGN(FD)<br>FPALIGN(FS)<br>FPALIGN(FT)<br>DIV0(FPR_F(FT))<br>SET_FPR_F(FD,FPR_F(FS)/<br>FPR_F(FT)) | Impartire in<br>virgula mobila<br>simpla preizie |
| <b>DIV.D</b> | 0x77   | DIV.D<br>fd,fs,ft | FPALIGN(FD)<br>FPALIGN(FS)<br>FPALIGN(FT)<br>DIV0(FPR_D(FT))<br>SET_FPR_D(FD,FPR_D(FS)/<br>FPR_D(FT)) | Impartire in<br>virgula mobila<br>dubla preizie  |

|                      |      |                   |                                                                         |                                                                           |
|----------------------|------|-------------------|-------------------------------------------------------------------------|---------------------------------------------------------------------------|
| <b>ABS.S</b>         | 0x78 | ABS.S<br>fd,fs    | FPALIGN(FD)<br>FPALIGN(FS)<br>SET_FPR_F(FD,fabs((double)<br>FPR_F(FS))) | Valoarea absoluta<br>in simpla precizie                                   |
| <b>ABS.D</b>         | 0x79 | ABS.D<br>fd,fs    | FPALIGN(FD)<br>FPALIGN(FS)<br>SET_FPR_D(FD,fabs((double)<br>FPR_D(FS))) | Valoarea absoluta<br>in dubla precizie                                    |
| <b>MOV.S</b>         | 0x7a | MOV.S<br>fd,fs    | FPALIGN(FD)<br>FPALIGN(FS)<br>SET_FPR_F(FD,FPR_F(FS))                   | Muta valoarea in<br>virgula mobila in<br>simpla precizie                  |
| <b>MOV.D</b>         | 0x7b | MOV.D<br>fd,fs    | FPALIGN(FD)<br>FPALIGN(FS)<br>SET_FPR_D(FD,FPR_D(FS))                   | Muta valoarea in<br>virgula mobila in<br>dubla precizie                   |
| <b>NEG.S</b>         | 0x7c | NEG.S<br>fd,fs    | FPALIGN(FD)<br>FPALIGN(FS)<br>SET_FPR_F(FD,-FPR_F(FS))                  | neaga valoarea in<br>virgula mobila in<br>simpla precizie<br>din fs in fd |
| <b>NEG.D</b>         | 0x7d | MOV.D<br>fd,fs    | FPALIGN(FD)<br>FPALIGN(FS)<br>SET_FPR_D(FD,-<br>FPR_D(FS))              | Neaga valoarea<br>in virgula mobila<br>in dubla precizie                  |
| <b>CVT.S.<br/>D</b>  | 0x80 | CVT.S.D<br>fd,fs, | FPALIGN(FD)<br>FPALIGN(FS)<br>SET_FPR_F(FD,(float)FPR_<br>D(FS))        | converteste<br>valoarea dublu<br>din fs in simpla<br>precizie in fd       |
| <b>CVT.S.<br/>W.</b> | 0x81 | CVT.S.W<br>fd,fs  | FPALIGN(FD)<br>FPALIGN(FS)<br>SET_FPR_D(FD,(float)FPR_<br>L(FS))        | Converteste<br>intregul in simpla<br>precizie                             |
| <b>CVT.D.<br/>S.</b> | 0x82 | CVT.D.S<br>fd,fs  | FPALIGN(FD)<br>FPALIGN(FS)<br>SET_FPR_D(FD,(double)FPR_<br>F(FS))       | Converteste<br>simpla precizie in<br>dubla precizie                       |
| <b>CVT.D.<br/>W.</b> | 0x83 | CVT.D.W<br>fd,fs  | FPALIGN(FD)<br>FPALIGN(FS)<br>SET_FPR_D(FD,(double)FPR_<br>L(FS))       | Converteste<br>intregul in simpla<br>precizie                             |
| <b>CVT.W.<br/>S.</b> | 0x84 | CVT.W.S<br>fd,fs  | FPALIGN(FD)<br>FPALIGN(FS)<br>SET_FPR_D(FD,(long)FPR_<br>F(FS))         | Converteste<br>simpla precizie in<br>integer                              |
| <b>CVT.W.<br/>D.</b> | 0x85 | CVT.W.D<br>fd,fs  | FPALIGN(FD)<br>FPALIGN(FS)<br>SET_FPR_D(FD,(long)FPR_<br>D(FS))         | Converteste<br>dublul in intreg                                           |
| <b>C.EQ.S</b>        | 0x90 | C.EQ.S.<br>fs,ft  | FPALIGN(FS)<br>FPALIGN(FT)                                              | Testeaza daca<br>fs=ft in simpla                                          |

|               |      |                  |                                                                         |                                               |
|---------------|------|------------------|-------------------------------------------------------------------------|-----------------------------------------------|
|               |      |                  | SET_FCC(FPR_F(FS))==FPR_F(FT))                                          | precizie                                      |
| <b>C.EQ.D</b> | 0x91 | C.EQ.D.<br>fs,ft | FPALIGN(FS)<br>FPALIGN(FT)<br>SET_FCC(FPR_D(FS))==FPR_D(FT))            | Testeaza daca<br>fs=ft in dubla<br>Precizie   |
| <b>C.LT.S</b> | 0x92 | C.LT.S.<br>fs,ft | FPALIGN(FS)<br>FPALIGN(FT)<br>SET_FCC(FPR_F(FS)<FPR_F(FT))              | Testeaza daca<br>fs<ft in spl.<br>Precizie    |
| <b>C.LT.D</b> | 0x93 | C.LT.D.<br>fs,ft | FPALIGN(FS)<br>FPALIGN(FT)<br>SET_FCC(FPR_D(FS)<FPR_D(FT))              | Testeaza daca<br>fs<ft in dubla<br>precizie   |
| <b>C.LE.S</b> | 0x94 | C.LE.S.<br>fs,ft | FPALIGN(FS)<br>FPALIGN(FT)<br>SET_FCC(FPR_F(FS)<=FPR_F(FT))             | Testeaza daca<br>fs<=ft in spl.<br>precizie   |
| <b>C.LE.D</b> | 0x95 | C.LE.D.<br>fs,ft | FPALIGN(FS)<br>FPALIGN(FT)<br>SET_FCC(FPR_D(FS)<=FPR_D(FT))             | Testeaza daca<br>fs<=ft in dubla.<br>Precizie |
| <b>SQRT.S</b> | 0x96 | SQRT.S<br>fd, fs | FPALIGN(FD)<br>FPALIGN(FS)<br>SET_FPR_F(FD,sqrt((double)<br>FPR_F(FS))) | Radacina patrata<br>in simpla precizie        |
| <b>SQRT.D</b> | 0x97 | SQRT.D<br>fd, fs | FPALIGN(FD)<br>FPALIGN(FS)<br>SET_FPR_D(FD,sqrt((double)<br>FPR_D(FS))) | Radacina patrata<br>in dubla precizie         |

### III.5. ALTE INSTRUCȚIUNI.

| Mne-monica     | Opcode | Format        | Semantica                                                                                       | Descriere                    |
|----------------|--------|---------------|-------------------------------------------------------------------------------------------------|------------------------------|
| <b>NOP</b>     | 0x00   | NOP           | Nu are                                                                                          | Nici o operatie              |
| <b>SYSCALL</b> | 0xa0   | SYSCALL       | Vezi anexa IV pentru detalii                                                                    | Apel de sistem               |
| <b>BREAK</b>   | 0xa1   | BREAK<br>uimm | Actiunea depinde de simulator. De obicei este afisat un mesaj de eroare si este apelata abort() | Declara o eroare de program  |
| <b>LUI</b>     | 0xa2   | LUI uimm      | SET_GPR(RT,UIMM<<16)                                                                            | Înarcă o valoare imediata în |

|             |      |                   |                       |                                                                                         |
|-------------|------|-------------------|-----------------------|-----------------------------------------------------------------------------------------|
|             |      |                   |                       | partea <i>high</i> a registrului destinatie. Partea <i>low</i> va fi initializata cu 0. |
| <b>MFC1</b> | 0xa3 | MFC1 <i>rt,fs</i> | SET_GPR(RT,FPR_L(FS)) | Muta dintr-un registru in virgula mobila intr-unul de intregi                           |
| <b>MTC1</b> | 0xa4 | MTC1 <i>rt,fs</i> | SET_FPR_L(FS,GPR(RT)) | Muta dintr-un registru de intregi in unul in virgula mobila s                           |

### III.6. OPERATORII SI OPERANZII CARE APAR ÎN SEMANTICI

| Operatori/operanzi    | Descriere                                                                            |
|-----------------------|--------------------------------------------------------------------------------------|
| <b>FS</b>             | Identice cu <i>rs</i>                                                                |
| <b>FT</b>             | Asemenea cu <i>rt</i>                                                                |
| <b>FD</b>             | Asemenea cu <i>rd</i>                                                                |
| <b>UIMM</b>           | Campul <i>imm</i> fara semn extins la valoarea cuvântului                            |
| <b>IMM</b>            | Campul <i>imm</i> cu semn extins la valoare de cuvânt                                |
| <b>OFFSET</b>         | Campul <i>imm</i> cu semn extins la cuvânt                                           |
| <b>CPC</b>            | PC-ul curent la executia instructiunii                                               |
| <b>NPC</b>            | Valoarea PC-ului urmator                                                             |
| <b>SET_NPC(V)</b>     | Seteaza urmatorul PC la valoarea <i>V</i>                                            |
| <b>GPR(N)</b>         | Registrul general <i>N</i> folosit                                                   |
| <b>SET_GPR(N,V)</b>   | Seteaza registrul <i>N</i> la valoarea <i>V</i> .                                    |
| <b>FPR_F(N)</b>       | Valoarea in simpla precizie a registrului in virgula mobila <i>N</i>                 |
| <b>SET_FPR_F(N,V)</b> | Seteaza registrul in virgula mobila <i>N</i> la valoarea in simpla precizie <i>V</i> |
| <b>FPR_D(N)</b>       | Valoarea in dubla precizie a registrului in virgula mobila <i>N</i>                  |
| <b>SET_FPR_D(N,V)</b> | Seteaza registrul in virgula mobila <i>N</i> la registrul dublu <i>V</i>             |
| <b>FPR_L(N)</b>       | Valoarea literală a cuvântului registrului in virgula mobila <i>N</i>                |
| <b>SET_FPR_L(N,V)</b> | Seteaza registrul in virgula mobila <i>N</i> la cuvântul <i>V</i>                    |
| <b>HI</b>             | Valoarea registrului <i>HI</i>                                                       |

|                              |                                                                        |
|------------------------------|------------------------------------------------------------------------|
| <b>SET_HI(V)</b>             | Seteaza <i>HI</i> la valoarea <i>V</i>                                 |
| <b>LO</b>                    | Valoarea registrului <i>LO</i>                                         |
| <b>SET_LO(V)</b>             | Seteaza registrul <i>LO</i> la valoarea <i>V</i>                       |
| <b>READ_SIGNED_BYTE(A)</b>   | Citeste un byte cu semn de la adresa <i>A</i>                          |
| <b>READ_UNSIGNED_BYTE(A)</b> | Citeste un byte fara semn de la adresa <i>A</i>                        |
| <b>WRITE_BYTE(V,A)</b>       | Scrie byte-ul <i>v</i> la adresa <i>A</i>                              |
| <b>READ_SIGNED_HALF(A)</b>   | Citeste intregul cu semn de la adresa <i>A</i>                         |
| <b>READ_UNSIGNED_HALF(A)</b> | Citeste de la adresa <i>A</i> intregul fara semn                       |
| <b>WRITE_HALF(V,A)</b>       | Scrie valoarea intreaga a lui <i>V</i> la adresa <i>A</i>              |
| <b>READ_WORD(A)</b>          | Citeste un cuvânt de la adresa <i>A</i>                                |
| <b>WRITE_WORD(V,A)</b>       | Scrie <i>V</i> la adresa lui <i>A</i>                                  |
| <b>TALIGN(T)</b>             | Verifica daca adresa <i>T</i> este aliniata la limita de 8 biti        |
| <b>FPALIGN(N)</b>            | Verifica daca registrul <i>N</i> este complet divizibil cu 2           |
| <b>OVER(X,Y)</b>             | Verifica pentru depasire cand fac $X+Y$                                |
| <b>UNDER(X,Y)</b>            | Verifica pentru depasire cand fac $X-Y$                                |
| <b>DIV0(V)</b>               | Verifica pentru eroarea de impartire cu 0 folosind <i>V</i> ca divizor |

Tabelul III.1.

### Semantica operatorilor si operanzilor folositi în instructiuni

## ANEXA IV

### DEFINIREA APELURILOR DE SISTEM

---

Anexa de fata ilustreaza toate apelurile sistem (întreruperi software) suportate de simulatoarele setului de instrumente *SimpleScalar*, cu codurile si semnificatia aferenta. Apelurile sistem reprezinta un mic set de servicii ale sistemului de operare prin instructiuni si sunt initiate de instructiunea SYSCALL. Pentru a apela un serviciu, programul încarca codul apelului în registrul \$v<sub>0</sub> si argumentele în registrele \$a<sub>0</sub> ... \$a<sub>3</sub> (sau \$f<sub>12</sub> pentru valori în flotant). Apelurile sistem care returneaza valori pun rezultatele în registrele \$v<sub>0</sub> sau \$f<sub>0</sub> pentru operatii în flotant. (De exemplu, citirea datelor dintr-un fisier si depunerea lor într-o variabila de memorie se realizeaza prin instructiunea **read**(int *fd*, char \**buf*, int *nbyte*). În acest caz, în \$v<sub>0</sub> depunem 0x03 - codul apelului, în \$a<sub>0</sub> *fd* - descriptorul de fisier din care se va efectua citirea, în \$a<sub>1</sub> *buf* - adresa de memorie care va retine datele citite - iar în \$a<sub>2</sub> *nbyte* - numarul de octeti cititi efectiv).

Simularea instructiunilor benchmark-urilor se realizeaza la nivel de utilizator (nu este simulata executia instructiunilor în interiorul sistemului de operare). Protocolul de tratare a **apelurilor sistem** este urmatorul (vezi figura IV.1):

- ❏ Decodificarea apelului sistem.
- ❏ Copierea argumentelor (daca exista) în memoria simulatorului (**sim-\***).
- ❏ Executarea apelului sistem de catre sistemul de operare (rutina de tratare a întreruperii).
- ❏ Copierea rezultatelor daca exista în memoria programului simulat (**benchmark-ul SPEC**).
- ❏ Modulul **syscall.c** implementeaza un subset de apeluri sistem specifice Unix. Pentru adaugarea oricarui nou apel sistem sau pentru portarea SimpleScalar pe un sistem de operare nou, este necesara implementarea software a rutinei de tratare în modulul **syscall.[hc]**.

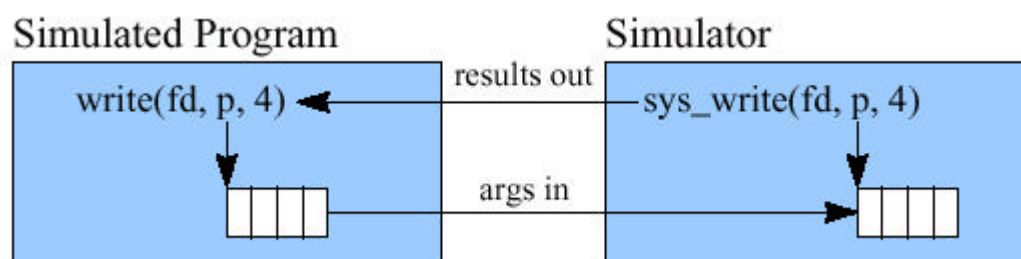


Figura IV.1. Protocolul de tratare a apelurilor sistem

| Mnemonica     | Cod apel | Interfata                                                                                       | Descriere                                       |
|---------------|----------|-------------------------------------------------------------------------------------------------|-------------------------------------------------|
| <b>EXIT</b>   | 0x01     | Void exit (int status);                                                                         | Încheie procesul curent.                        |
| <b>READ</b>   | 0x03     | Int read (int fd, char *buf, int nbyte)                                                         | Citeste din fisier într-o variabila de memorie. |
| <b>WRITE</b>  | 0x04     | Int write (int fd, char*buf, int nbyte)                                                         | Scrie din bufer in fisier                       |
| <b>OPEN</b>   | 0x05     | Int open (char *fname, int flags, int mode)                                                     | Deschide un fisier                              |
| <b>CLOSE</b>  | 0x06     | Int close (int fd)                                                                              | Inchide un fisier                               |
| <b>CREAT</b>  | 0x08     | Int creat (char *fname, int mode)                                                               | Creaza un fisier                                |
| <b>UNLINK</b> | 0x0a     | Int unlink (char *fname)                                                                        | Sterge un fisier                                |
| <b>CHDIR</b>  | 0x0c     | Int chdir (char *path)                                                                          | Schimba directorul procesului                   |
| <b>CHMOD</b>  | 0x0f     | Int chmod (int *fname, int mode)                                                                | Schimba modul de acces la fisier                |
| <b>CHOWN</b>  | 0x10     | Int chown (char *fname, int owner, int group)                                                   | Scimba utilizatorul fisierului si grupul        |
| <b>BRK</b>    | 0x11     | Int brk (long addr)                                                                             | Schimba adresa de intrerupere a procesului      |
| <b>LSEEK</b>  | 0x13     | Long lseek (int fd, long offset, int whence)                                                    | Muta pointerul de fisier                        |
| <b>GETPID</b> | 0x14     | Int getpid (void)                                                                               | Gaseste identificatorul procesului              |
| <b>GETUID</b> | 0x18     | Int getuid (void)                                                                               | Gaseste identificatorul utilizatorului          |
| <b>ACCESS</b> | 0x21     | Int access (char *fname, int mode)                                                              | Determina accesibilitatea unui fisier           |
| <b>STAT</b>   | 0x26     | Struct stat<br>{<br>short st_dev;<br>long st_ino;<br>unsigned short st_mode;<br>short st_nlink; | Gaseste situatia fisierului                     |



|                      |      |                                                                                                                                                                                                                                                                         |                                                       |
|----------------------|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------|
|                      |      | <pre> short st_uid; short st_gid; short st_rdev; int st_size; int st_atime; int st_spare1; int st_mtime; int st_spare2; int st_ctime; int st_spare3; long st_bksize; long st_blocks; long st_gennum; long st_spare4; }; int stat (char *fname, struct stat *buf) </pre> |                                                       |
| <b>LSTAT</b>         | 0x28 | Int lstat (char *fname, struct stat *buf)                                                                                                                                                                                                                               | Obține starea fisierului (fara dereferirea legaturii) |
| <b>DUP</b>           | 0x29 | Int dup (int fd)                                                                                                                                                                                                                                                        | Dubleaza descriptorul de fisier                       |
| <b>PIPE</b>          | 0x2a | Int pipe (int fd[2])                                                                                                                                                                                                                                                    | Creaza un interproces comm.channel                    |
| <b>GETGID</b>        | 0x2f | Int getgid (void)                                                                                                                                                                                                                                                       | Obține identificatorii de grup                        |
| <b>IOCTL</b>         | 0x36 | Int ioctl (int fd, int request, char *arg)                                                                                                                                                                                                                              | Instrumentul de control al interfetei                 |
| <b>FSTAT</b>         | 0x3e | Int fstat (int fd, struct stat *buf)                                                                                                                                                                                                                                    | Obține starea descriptorului de fisier                |
| <b>GETPAGESIZE</b>   | 0x40 | Int getpagesize(void)                                                                                                                                                                                                                                                   | Obține dimensiunea paginii                            |
| <b>GETDTABLESIZE</b> | 0x59 | Int getdtablesize(void)                                                                                                                                                                                                                                                 | Obține dimensiunea tabelii de descriptori             |
| <b>DUP2</b>          | 0x5a | Int dup2(int fd1, int fd2)                                                                                                                                                                                                                                              | Dubleaza un descriptor                                |
| <b>FCNTL</b>         | 0x5c | Int fcntl(int fd, int cmd, int arg)                                                                                                                                                                                                                                     | Control fisier                                        |
| <b>SELECT</b>        | 0x5d | Int select (int width, fd_set*readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)                                                                                                                                                                    | Multiplexeaza I/O                                     |
| <b>GETTIMEOFDAY</b>  | 0x74 | <pre> Struct timeval {     long tv_sec;     long tv_usec; }; struct int {     timezone tz_minuteswest; </pre>                                                                                                                                                           | Obține data si ora                                    |

|                  |      |                                                                                           |                                      |
|------------------|------|-------------------------------------------------------------------------------------------|--------------------------------------|
|                  |      | int tz_dstime;<br>};<br>int gettimeofday(struct<br>timeval *tp, struct<br>timezoone *tzp) |                                      |
| <b>WRITEV</b>    | 0x79 | Int writew(int fd, struct<br>iovec*iov, int cnt)                                          | .....                                |
| <b>UTIMES</b>    | 0x8a | Int utimes (char *file,<br>struct timeval *tvp)                                           | Seteaza ora pt un fisier             |
| <b>GETRLIMIT</b> | 0x90 | Int getrlimit (int res, struct<br>rlimit *rlp)                                            | Obtine consumul maxim<br>de resurse  |
| <b>SETRLIMIT</b> | 0x91 | Int setrlimit (int res, struct<br>rlimit *rlp)                                            | Seteaza consumul maxim<br>de resurse |

Tabelul IV.1.

Apeluri sistem suportate de simulatoarele setului de instrumente *SimpleScalar*

## ANEXA V

---

Secventa examinata din punct de vedere al predictiei valorilor registrilor MIPS reprezinta partea de afisare a unui sir de numere prime dintr-un program care calculeaza, stocheaza în memorie si în final afiseaza primele 600 de numere prime. Dupa obtinerea celor 600 de numere, acestea se citesc din memorie în registrii, urmând a fi afisate câte 25 pe rând (cu spatiu între ele). Testarea aplicatiei s-a facut pe simulatorul SPIM, scris de catre James Larus în 1993 pentru uz didactic la Facultatea de Calculatoare a Universitatii din Wisconsin.

|          |                                             |                                                               |
|----------|---------------------------------------------|---------------------------------------------------------------|
| PC = 0   | la \$t <sub>2</sub> , sir                   | // sir retine adresa de început a<br>// celor 600 de elemente |
| RESTART: |                                             |                                                               |
| PC = 4   | lw \$a <sub>0</sub> , (\$t <sub>2</sub> )   |                                                               |
| PC = 8   | li \$v <sub>0</sub> , 1                     | // afisare numar prim curent                                  |
| PC = 12  | syscall                                     |                                                               |
| PC = 16  | sub \$a <sub>1</sub> , \$a <sub>1</sub> , 1 | // decrementare contor numere<br>// prime ramase de afisat    |
| PC = 20  | add \$t <sub>2</sub> , \$t <sub>2</sub> , 4 | //incrementare adresa numar<br>// prim curent                 |
| PC = 24  | sub \$sp, \$sp, 4                           |                                                               |
| PC = 28  | sw \$a <sub>0</sub> , (\$sp)                |                                                               |
| PC = 32  | li \$v <sub>0</sub> , 11                    |                                                               |
| PC = 36  | li \$a <sub>0</sub> , 0x20                  | // afisare spatiu între numere<br>// prime                    |
| PC = 40  | syscall                                     |                                                               |
| PC = 44  | li \$t <sub>0</sub> , 25                    |                                                               |
| PC = 48  | div \$a <sub>1</sub> , \$t <sub>0</sub>     | // am ajuns la cel de-al 25-lea<br>// numar prim ?            |
| PC = 52  | mfhi \$t <sub>1</sub>                       |                                                               |

```

PC = 56      bnez $t0, nu_sare
PC = 60      li $v0, 11
PC = 64      li $a0, 0x0a          // afisare pe rând nou
PC = 68      syscall
nu_sare:
PC = 72      lw $a0, ($sp)          //refacere numar prim
PC = 76      add $sp, $sp, 4        //refacere stiva
PC = 80      bnez $a1, RESTART

```

Cu toate ca secventa aleasa nu este poate cel mai sugestiv exemplu si cu toate ca instructiunile cu stiva puteau fi înlocuite de altele de transfer, s-a urmarit înțelegerea gradului ridicat de localitate existent pe registrii procesorului MIPS, si cum un predictor simplu de tip "*Last Value*" modificat (care retine 2 sau 4 valori ale fiecarui registru) si cu mecanism "*perfect*" de selectie a valorii prezise poate exploata conceptul de localitate.

Metodologia de determinare a gradului de localitate pe registri consta în verificarea în fiecare faza *Write Back* daca valoarea scrisa în respectivul registru se regaseste printre ultimele  $k$  valori anterior scrise. Predictia se realizeaza abia în faza ID (*decode*) dupa citirea valorii operanzilor din setul de registri generali.

Din exemplul prezentat se observa ca în timpul executiei programului registrul  $v_0$  ia doar doua valori - coduri de apel sistem (1, respectiv 11), registrul  $t_0$  pastreaza valoarea constanta 25,  $sp$  ia una din valorile  $0x7ffc000$  respectiv  $0x7ffbffc$ . Ceilalti registri își modifica valorile datorita instructiunilor incrementale/decrementale (reprezinta indecsi de adresa, elemente distincte stocate în memorie - numere prime). Pe lângă valorile variabile pe care le ia din memorie registrul  $a_0$  retine si doua valori (20, respectiv 10) - parametri ai apelului sistem de afisare caracter.



## BIBLIOGRAFIE

---

[Bar02] **Barbat B.** - *Sisteme inteligente orientate spre agent*, Editura Academiei Române, Bucuresti, ISBN 973-27-0940-5, 2002.

[Bel66] **Belady, L.** - *A study of replacement algorithms for a virtual-storage Computer*, IBM Systems Journal, 1966.

[Bhan96] **Bhandarkar D.** - *Alpha Implementations and Architecture: Complete Reference and Guide*, Digital Press, 1996.

[Bur97] **Burger D., Austin T.** - *The SimpleScalar Tool Set, Version 2.0*, University of Wisconsin Madison, USA, CSD TR #1342, June, 1997

[Burj01] **Burja L.** - *Perceptronul cu un singur strat* - Gazeta de Informatica GInfo nr. 7, noiembrie 2001, Editura Agora Media, Tg. Mures.

[Cha97] **Chang P.Y., Hao E., Patt Y.N.** - *Target Prediction for Indirect Jumps*, ISCA '97 (<http://www.eecs.umich.edu/HPS>).

[Col93] **Collins, R.** - *Developing A Simulator for the Hatfield Superscalar Processor*, Division of Computer Science, Technical Report No. 172, University of Hertfordshire, December 1993.

[Colc98] **Colceriu R.** - *Imbunatatirea performantelor memoriilor cache cu mapare directa, integrate intr-un procesor paralel* - Teza de Masterat, Sibiu, 1998 (îndrumator L. Vintan).

[Dub91] **Dubey P., Flynn M.** - *Branch Strategies: Modeling and Optimization*, IEEE Transaction on Computer, No 10, 1991.

- [Eve96] **Evers M., Chang P.Y., Patt Y.N.** - *Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches*, ISCA '96.
- [Flor98] **Florea A.** – *Optimizarea proceselor de scriere într-o arhitectura RISC superscalara de tip Harvard*, Teza de Masterat, Sibiu, 1998 (îndrumator L. Vintan).
- [Flor00] **Florea A., Egan C.** – *Reducing the Technological Gap Between an Advanced Processor and the Memory Hierarchy System* –the 4<sup>th</sup> International Conference on Technical Informatics (CONTI '00) Timisoara, 2000.
- [FlorVin02] **Florea A., Vintan L., Sima D.** - *Understanding Value Prediction through Complex Simulations*, Proceedings of The 5<sup>th</sup> International Conf. on Technical Informatics (CONTI '2002), University "Politehnica" of Timisoara, Romania, octombrie 2002.
- [Gal93] **Gallant S.I.** - *Neural Networks and Expert Systems*, MIT Press. 1993.
- [Grü92] **Gründbacher H.** - *Windows Deluxe Simulator - Tutorial*, University of Technology, Viena, 1992.
- [Hen94] **Hennessy J., Patterson D.** - *Computer Organization and Design: The Hardware / Software Interface*, Morgan Kaufmann Publishers, San Francisco, California, 1994.
- [Hen96] **Hennessy J., Patterson D.** - *Computer Architecture: A quantitative Approach*, Morgan Kaufmann Publishers, Second Edition, 1996.
- [Hill88] **Hill, M.** – *A Case for Direct-Mapped Caches*, IEEE Computer, December 1988.
- [Host] **Hostetler L.B., Mirtich B.** - *DLXsim - A Simulator for DLX*.
- [Joup90] **Jouppi, N.** – *Improving Direct-Mapped Cache Performance by the addition of a Small Fully Associative Cache and Prefetch Buffers*, Proc. 17<sup>th</sup> International Symposium On Computer Architecture, 1990.

- [Kae91] **Kaeli D., Emme P.** – *Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns*, 18-th Int.'L Conf. On Computer Architecture, Toronto, May 1991.
- [Kan92] **Kane G., Heinrich J.** – *MIPS RISC Architecture*, Prentice Hall, 1992.
- [Lar93] **Larus J.** - *SPIM S20: A MIPS R2000 Simulator*, Morgan Kaufmann Publishers, 1993.
- [Lee97] **Lee C., Potkonjak M. and Mangione-Smith W.** - *MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems*, 1997.
- [Lip96] **Lipasti M., Wilkerson C., Shen P.** – *Value Locality and Load Value Prediction*, 17<sup>th</sup> ASPLOS International Conference VII, pg. 138-147, MA, SUA, October 1996.
- [McFar92] **McFarling, S.** – *Cache replacement with dynamic exclusion*, In Proc. 19<sup>th</sup> Int'l. Symposium on Computer Architecture, 1992, pg. 192÷200.
- [Mih99] **Mihu I.Z.** - *Arhitectura sistemelor de calcul. Concepte avansate de proiectare*, Editura Casa Cartii de Stiinta, Cluj-Napoca, 1999
- [Mou] **Moura, C.** - *SuperDLX A Generic Superscalar Simulator*, ACAPS Technical Memo 64, McGill University School of Computer Science.
- [Mud96] **Mudge, T.N., Chen, I., Coffey, J.** - *Limits of Branch Prediction*, Technical Report, Electrical Engineering and Computer Science Department, The University of Michigan, Ann Arbor, Michigan, USA, January 1996.
- [Neg93] **Negrescu L.** – *Introducere în limbajul C* - Editura MicroInformatica, Cluj Napoca., 1993.
- [Pan92] **Pan S.T., So K., Rahmeh J.T.** - *Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation*, ASPLOS V Conference, Boston, October, 1992.
- [Per93] **Perleberg C., Smith A. J.** - *Branch Target Buffer Design and Optimisation*, IEEE Trans. Computers, No. 4, 1993.

- [Sbe00] **Sbera M.** – *Investigatii cantitative privind conceptul de predictor neuronal de ramificatii de program*, BA Diploma, July, 2000.
- [Sim97] **Sima D., Fountain T., Kacsuk P.** – *Advanced Computer Architecture: A Design Space Approach*, Addison-Wesley, 1997.
- [Song94] **Song, S.P., Denman, M., Chang, J.**, "The PowerPC 604 RISC Microprocessor", IEEE Micro, October 1994, p.8-17.
- [SPEC] – *The SPEC benchmark programs*, <http://www.spec.org>
- [Ste96] **Steven, G. B.** – *A Superscalar Architecture to Exploit ILP*, Proceedings of the 22<sup>nd</sup> Euromicro Conference, Prague, 2-5 September 1996.
- [Sti94] **Stiliadis, D., Varma, A.** – *Selective Victim Caching: A Method to Improve the Performance of Direct-Mapped Caches*, TR UCSC-CRL-93-41, University of California, 1994 (republished in a shorter version in IEEE Trans. on Computers, May 1997).
- [Sug93] **Sugumar, R., Abraham, S.** – *Efficient simulation of caches under optimal replacement with applications to miss characterization*, Performance Evaluation Review, 21(1):24-35, 1993.
- [Vin98] **Vintan L.**, Armat C.- *Some Investigations about Selective Victim Caching into a Superscalar Environment*, Transactions on Automatic Control and Computer Science, Special Issue Dedicated to 3<sup>rd</sup> International Conf. on Technical Informatics (CONTI '98), Volume 43 (57), No 4 of 4, ISSN 1224-600X, University "Politehnica" of Timisoara, Romania, 1998.
- [Vin99] **Vintan L., Florea A.** - *Investigating New Branch Prediction Through Quantitative Approach – Beyond 2000: Engineering Research Strategies*, November 25-27, Sibiu, 1999.
- [Vin99a] **Vintan L.** - *Predicting Branches through Neural Networks: A LVQ and a MLP Approach*, University "L. Blaga" of Sibiu, Faculty of Engineering, Dept. of Comp. Sc., Technical Report, February 1999.
- [Vin99b] **Vintan L., Egan, C.** - *Extending Correlation in Branch Prediction Schemes*, International Euromicro'99 Conference, Milano, Italy, September 1999.




- [Vin00] **Vintan L.** - *Arhitecturi de procesoare cu paralelism la nivelul instructiunilor*, Editura Academiei Române, Bucuresti, ISBN 973-27-0734-8, 2000 (264 pg.).
- [Vin00a] **Vintan L.** - *Towards a Powerful Dynamic Branch Predictor*, in Romanian Journal of Information Science and Technology, Romanian Academy Publishing House, Bucharest, 2000.
- [VinFlor99] **Vintan L., Florea A.** – *Sisteme cu microprocesoare. Aplicatii*, Editura Universitatii "Lucian Blaga" din SIBIU, 1999 (250 pg.).
- [VinFlor00] **Vintan N. L., Florea A.** – *Microarhitecturi de procesare a informatiei*, Editura Tehnica, Bucuresti, ISBN 973-31-1551-7, 2000 (312 pg.)
- [Vin02] **Vintan L.** - *Predictie si speculatie în microprocesoarele avansate*, Editura Matrix Rom, Bucuresti, ISBN 973-685-497-3, 2002 (89 pg.).
- [Ung99] **Ungherer T. et all** – *Processor Architecture: From DataFlow to Superscalar and Beyond*, Berlin, 1999.
- [Yeag96] **Yeager, K.C.** - *The Mips R10000 Superscalar Microprocessor*, IEEE Micro, April 1996, p.28- 40.
- [Yeh92] **Yeh T., Patt Y.** - *Alternative Implementations of Two Level Adaptive Branch Prediction*, 19<sup>th</sup> Ann. Int.'L Symp. Computer Architecture, 1992.
- [Wan97] **Wang K, Franklin M.** - *Highly Accurate Data Value Prediction using Hybrid Predictors*, Proceedings of the 30<sup>th</sup> Annual ACM/IEEE International Symposium on Microarchitecture, December 1997.
- [Will98] **Williams, M.** – *Bazele Visual C++ 4*, Editura Teora, Bucuresti, Octombrie 1998.
- [Wol00] **Wolff M., Wills L.** - *A Superscalar Architecture Trace Simulator using interactive animation*.

# CUPRINS

|                                                                                                                                         |           |
|-----------------------------------------------------------------------------------------------------------------------------------------|-----------|
| <b><u>PREFATA</u></b>                                                                                                                   | <b>1</b>  |
| <b><u>CUPRINS</u></b>                                                                                                                   | <b>4</b>  |
| <b><u>0. SIMULATORUL. CE ESTE? ESTE NECESARA SIMULAREA?</u></b>                                                                         | <b>7</b>  |
| <b><u>METODOLOGII DE SIMULARE.</u></b>                                                                                                  | <b>7</b>  |
| <b><u>1. ARHITECTURA MICROPROCESOARELOR MIPS R2000/R3000</u></b>                                                                        | <b>9</b>  |
| 1.1. SCOPUL LUCRARIII                                                                                                                   | 9         |
| 1.2. MEMENTO TEORETIC                                                                                                                   | 9         |
| 1.3. DESFASURAREA LUCRARIII                                                                                                             | 12        |
| 1.3.1. SCHEMA BLOC SI REGISTRII PROCESORULUI MIPS R2000                                                                                 | 12        |
| 1.3.2. ORDINEA OCTETILOR SI MODURILE DE ADRESARE                                                                                        | 14        |
| 1.3.3. SINTAXA ASAMBLOR                                                                                                                 | 15        |
| 1.3.4. FORMATUL INSTRUCIUNILOR SI SETUL DE INSTRUCIUNI AL PROCESORULUI MIPS R2000                                                       | 16        |
| 1.3.5. UTILIZAREA MEMORIEI SI CONVENTII DE APEL                                                                                         | 28        |
| 1.3.6. APELURI SISTEM                                                                                                                   | 30        |
| 1.3.7. PSEUDOINSTRUCIUNI FOLOSITE ÎN LIMBAJ DE ASAMBLARE MIPS                                                                           | 32        |
| 1.3.8. GHID DE UTILIZARE AL SIMULATORULUI                                                                                               | 33        |
| 1.4. PROBLEME DE LABORATOR PROPUSE SPRE REZOLVARE                                                                                       | 36        |
| 1.4.1. NOTE EXPLICATIVE REFERITOARE LA BENCHMARK-URILE IMPLEMENTATE                                                                     | 36        |
| 1.4.2. SOLUTII LA PROBLEME PROPUSE                                                                                                      | 36        |
| 1.4.3. PROBLEME PROPUSE SPRE REZOLVARE                                                                                                  | 41        |
| <b><u>2. INVESTIGATII ARHITECTURALE UTILIZÂND SIMULATORUL DLX</u></b>                                                                   | <b>43</b> |
| 2.1. SCOPUL LUCRARIII                                                                                                                   | 43        |
| 2.2. MEMENTO TEORETIC                                                                                                                   | 43        |
| 2.3. DESFASURAREA LUCRARIII                                                                                                             | 45        |
| 2.3.1. ARHITECTURA MICROPROCESOARELOR DLX                                                                                               | 45        |
| 2.3.2. PORNIREA SI CONFIGURAREA WINDLX [Grü92]                                                                                          | 56        |
| 2.3.3. ÎNCARCAREA PROGRAMELOR DE TEST                                                                                                   | 58        |
| 2.3.4. SIMULAREA PROPRIU-ZISA                                                                                                           | 58        |
| 2.3.5. APELURI SISTEM                                                                                                                   | 70        |
| 2.4. PROBLEME DE LABORATOR PROPUSE SPRE REZOLVARE                                                                                       | 71        |
| 2.4.1. NOTE EXPLICATIVE REFERITOARE LA BENCHMARK-URILE MIN SUMA MAX.S RESPECTIVE INVERSAREA UNUI NUMAR CITIT DE LA TASTATURA            | 71        |
| 2.4.2. PROBLEME PROPUSE SPRE REZOLVARE                                                                                                  | 78        |
| SCURTE CONCLUZII                                                                                                                        | 80        |
| <b><u>3. SATSIM: SIMULAREA ARHITECTURILOR SUPERSICALARE FOLOSIND ANIMATIE INTERACTIVA</u></b>                                           | <b>81</b> |
| 3.1. SCOPUL LUCRARIII                                                                                                                   | 81        |
| 3.2. INTRODUCERE                                                                                                                        | 81        |
| 3.3. INTERFATA CU UTILIZATORUL. DESCRIERE DETALIATA.                                                                                    | 83        |
| 3.3.1. PARAMETRII SIMULATORULUI                                                                                                         | 83        |
| 3.3.2. ALTE RESURSE SOFTWARE UTILIZATE. MENIUL SIMULATORULUI SI BARA DE INSTRUMENTE                                                     | 93        |
| 3.4. DESCRIEREA FUNCTIONALITATII PROCESORULUI                                                                                           | 96        |
| 3.4.1 FAZA FETCH-INSTRUCIUNE                                                                                                            | 99        |
| 3.4.2 FAZA DE DECODIFICARE A INSTRUCIUNII                                                                                               | 100       |
| 3.4.3 NIVELUL "DISPATCH-ISSUE" DE PROCESARE. RUTAREA INSTRUCIUNILOR SPRE STATIILE DE REZERVARE SI UNITATILE DE EXECUTIE CORESPUNZATOARE | 100       |
| 3.4.4 SETUL DE REGISTRII GENERALI                                                                                                       | 102       |
| 3.4.5 BUFFER-UL DE REORDONARE (RB)                                                                                                      | 102       |
| 3.4.6 BUFFER-UL DE REDENUMIRE (RENB)                                                                                                    | 103       |
| 3.4.7. STATIILE DE REZERVARE (SR)                                                                                                       | 104       |

|                                                                                                           |            |
|-----------------------------------------------------------------------------------------------------------|------------|
| 3.4.8. UNITATILE FUNCTIONALE DE EXECUTIE IMPLEMENTATE PIPELINE                                            | 106        |
| 3.4.9. FAZA COMMIT (WRITE BACK)                                                                           | 107        |
| 3.5. DISFUNCTIONALITATI RECUNOSCUTE ÎN ACEASTA VERSIUNE A SIMULATORULUI                                   | 108        |
| 3.6. CONCLUZII SI PROIECTE DE VIITOR                                                                      | 109        |
| 3.7. PROBLEME PROPUSE SPRE REZOLVARE                                                                      | 110        |
| <b>4. SIMULAREA INTERFETEI PROCESOR - CACHE PENTRU O ARHITECTURA RISC SUPERSCLARA PARAMETRIZABILA</b>     | <b>111</b> |
| 4.1. SCOPUL LUCRARI                                                                                       | 111        |
| 4.2. MEMENTO TEORETIC                                                                                     | 111        |
| 4.3. DESFASURAREA LUCRARI                                                                                 | 116        |
| 4.3.1. DESCRIEREA SIMULATORULUI                                                                           | 116        |
| 4.3.2. PROGRAMELE DE TEST STANFORD                                                                        | 124        |
| 4.4. PROBLEME PROPUSE SPRE REZOLVARE                                                                      | 127        |
| <b>5. ÎMBUNATATIREA SISTEMULUI IERARHIZAT DE MEMORIE PRIN ARHITECTURI DE TIP "SELECTIVE VICTIM CACHE"</b> | <b>129</b> |
| 5.1. SCOPUL LUCRARI                                                                                       | 129        |
| 5.2. MEMENTO TEORETIC                                                                                     | 129        |
| 5.2.1. ALGORITMUL DE SELECTIVE VICTIM CACHE                                                               | 135        |
| 5.2.2. ALGORITMUL DE PREDICTIE                                                                            | 136        |
| 5.2.3. METRICI DE PERFORMANTA                                                                             | 139        |
| 5.3. DESFASURAREA LUCRARI                                                                                 | 148        |
| 5.3.1. DESCRIEREA SIMULATORULUI                                                                           | 148        |
| 5.4. PROBLEME PROPUSE SPRE REZOLVARE                                                                      | 154        |
| 5.4.1. INDICATII DE REZOLVARE                                                                             | 155        |
| <b>6. IMPLEMENTAREA SCHEMELOR CLASICE DE PREDICTIE ÎN PROCESOARELE SUPERSCLARE AVANSATE.</b>              | <b>157</b> |
| 6.1. SCOPUL LUCRARI                                                                                       | 157        |
| 6.2. LUNG MEMENTO TEORETIC                                                                                | 157        |
| 6.3. DESFASURAREA LUCRARI                                                                                 | 171        |
| 6.3.1. PREDICTOR DE SALTURI DE TIP BTB                                                                    | 171        |
| 6.3.2. PREDICTOR CORELAT DE SALTURI, ADAPTIV PE DOUA NIVELE - GAG                                         | 187        |
| <b>7. PREDICTOR NEURONAL DE RAMIFICATII PROGRAM</b>                                                       | <b>199</b> |
| 7.1. SCOPUL LUCRARI                                                                                       | 199        |
| 7.2. INTRODUCERE ÎN RETELE NEURALE SI ALGORITMI GENETICI                                                  | 199        |
| 7.2.1. RETELE NEURALE. GENERALITATI.                                                                      | 199        |
| 7.2.2. MODELE DE RETELE NEURALE                                                                           | 203        |
| 7.2.3. PREDICTIA DE SALTURI PRIN METODE NEURALE                                                           | 208        |
| 7.2.4. ALGORITMI GENETICI (EVOLUTIONARY COMPUTING)                                                        | 209        |
| 7.3. DESFASURAREA LUCRARI                                                                                 | 211        |
| 7.3.1. INTERFATA CU UTILIZATORUL                                                                          | 211        |
| 7.3.2. METODOLOGIA DE SIMULARE                                                                            | 213        |
| 7.3.3. DESCRIERE IMPLEMENTARE SOFTWARE                                                                    | 218        |
| 7.3.4. GHID DE UTILIZARE                                                                                  | 228        |
| 7.4. PROBLEME PROPUSE SPRE REZOLVARE                                                                      | 233        |
| 7.4.1. REPREZENTAREA GRAFICA A REZULTATELOR SIMULARILOR EFECTUATE                                         | 233        |
| <b>8. DEZVOLTAREA DE SIMULATOARE SUB MEDIUL SIMPLESCALAR UTILIZÂND BENCHMARK-URILE SPEC</b>               | <b>235</b> |
| 8.1. SETUL DE INSTRUMENTE - "SIMPLESCALAR 3.0"                                                            | 235        |
| 8.1.1. CE ESTE "SIMPLESCALAR TOOL SET" ?                                                                  | 235        |
| 8.1.2. AVANTAJELE UTILIZARII "SIMPLESCALAR TOOL SET"                                                      | 236        |
| 8.1.3. DEZVANTAJE ÎN UTILIZAREA SETULUI DE INSTRUMENTE "SIMPLESCALAR"                                     | 238        |
| 8.1.4. CE ESTE "GNU" ? UTILITARE GNU.                                                                     | 238        |
| 8.1.5. INSTALAREA SETULUI DE INSTRUMENTE "SIMPLESCALAR". GHID DE UTILIZARE.                               | 241        |
| 8.1.6. SIMULATOARELE SETULUI DE INSTRUMENTE "SIMPLESCALAR 3.0".                                           | 243        |
| 8.2. ARHITECTURA SIMPLESCALAR                                                                             | 254        |

|                                                                                                                  |            |
|------------------------------------------------------------------------------------------------------------------|------------|
| 8.3. BENCHMARK-URI FOLOSITE ÎN SIMULARE: SPEC, MEDIABENCH.                                                       | 256        |
| 8.4. APLICATII PROPUSE SPRE REZOLVARE                                                                            | 261        |
| 8.4.1. REPREZENTAREA GRAFICA A REZULTATELOR SIMULARILOR EFECTUATE                                                | 263        |
| <b>9. EXTINDEREA MEDIULUI SIMPLESCALAR CU MODULUL DE MASURARE A GRADELOR DE LOCALITATE</b>                       | <b>265</b> |
| 9.1. METODOLOGIA DE REALIZARE A MODULULUI VALUE PREDICTION                                                       | 265        |
| 9.1.1. DESCRIEREA SUMARA A CODULUI SURSA AFERENT SIMULATOARELOR SIMPLESCALAR EXISTENTE SI A RUTINELOR COMPONENTE | 265        |
| 9.1.2. EXTINDEREA SETULUI "SIMPLESCALAR" CU MODULUL VALUEPREDICTION.                                             | 267        |
| 9.2. DESCRIEREA SIMULATORULUI "LAST VALUE PREDICTOR"                                                             | 270        |
| 9.3. LAST VALUE PREDICTOR. INTERFATA GRAFICA. GHID DE UTILIZARE.                                                 | 277        |
| 9.4. APLICATII PROPUSE SPRE REZOLVARE                                                                            | 282        |
| 9.4.1. REPREZENTAREA GRAFICA A REZULTATELOR SIMULARILOR EFECTUATE                                                | 283        |
| <b>10. IMPLEMENTAREA UNOR TEHNICI DE PREDICTIE DINAMICA A VALORILOR</b>                                          | <b>285</b> |
| 10.1. SCOPUL LUCRARI                                                                                             | 285        |
| 10.2. MEMENTO TEORETIC                                                                                           | 285        |
| 10.2.1. PREDICTOARE INCREMENTALE                                                                                 | 285        |
| 10.2.2. PREDICTOARE CONTEXTUALE                                                                                  | 289        |
| 10.2.3. PREDICTOARE HIBRIDE                                                                                      | 292        |
| 10.3. DESFASURAREA LUCRARI                                                                                       | 293        |
| 10.4. APLICATII PRACTICE PROPUSE SPRE REZOLVARE                                                                  | 297        |
| 10.4.1. REPREZENTAREA GRAFICA A REZULTATELOR SIMULARILOR EFECTUATE                                               | 298        |
| <b>11. PROBLEME PROPUSE SPRE REZOLVARE</b>                                                                       | <b>299</b> |
| <b>12. INDICATII DE SOLUTIONARE</b>                                                                              | <b>335</b> |
| <b>13. CE CONTINE CD-UL ?</b>                                                                                    | <b>395</b> |
| 13.1. SIMULAREA UNOR ARHITECTURI CU PARALELISM LA NIVELUL INSTRUCIUNILOR                                         | 395        |
| 13.2. DOCUMENTE                                                                                                  | 404        |
| <b>ANEXA I</b>                                                                                                   | <b>405</b> |
| <b>SATSIM: GHID DE UTILIZARE</b>                                                                                 | <b>405</b> |
| <b>ANEXA II</b>                                                                                                  | <b>417</b> |
| <b>PSEUDO-TRANSLATOR PENTRU SATSIM</b>                                                                           | <b>417</b> |
| II.1. NOTIUNI GENERALE                                                                                           | 417        |
| II.2. MOD DE UTILIZARE                                                                                           | 417        |
| II.3. MOD DE REALIZARE                                                                                           | 419        |
| <b>ANEXA III</b>                                                                                                 | <b>421</b> |
| <b>DEFINIREA SETULUI DE INSTRUCIUNI</b>                                                                          | <b>421</b> |
| III.1. INSTRUCIUNILE DE CONTROL.                                                                                 | 421        |
| III.2. INSTRUCIUNILE LOAD/ STORE                                                                                 | 422        |
| III.3. INSTRUCIUNILE PENTRU ÎNTREGI                                                                              | 426        |
| III.4. INSTRUCIUNILE ÎN VIRGULA MOBILA                                                                           | 428        |
| III.5. ALTE INSTRUCIUNI.                                                                                         | 430        |
| III.6. OPERATORII SI OPERANZII CARE APAR ÎN SEMANTICI                                                            | 431        |
| <b>ANEXA IV</b>                                                                                                  | <b>433</b> |
| <b>DEFINIREA APELURILOR DE SISTEM</b>                                                                            | <b>433</b> |
| <b>ANEXA V</b>                                                                                                   | <b>437</b> |
|  <b>BIBLIOGRAFIE</b>          | <b>439</b> |