SIMULAREA INTERFETEI PROCESOR-CACHE PENTRU O ARHITECTURA RISC SUPERSCALARA PARAMETRIZABILA

1. Scopul lucrarii

Lucrarea de fata consta în descrierea si utilizarea unui simulator parametrizabil pentru o arhitectura superscalara. 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 de scriere în cache cunoscute (*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.

2. Memento teoretic

Una din cele mai studiate zone ale cercetarii, proiectarii si implementarilor actuale în stiinta calculatoarelor o reprezinta problematica procesarii paralele la nivelul instructiunilor masina. Limitarea principala a performantei arhitecturilor RISC la o instructiune / ciclu masina, a adus începând din anul 1987 un concept si mai îndraznet: acela de masina cu executii multiple ale instructiunilor (MEM). Acestea au drept principal deziderat depasirea barierei de o instructiune / tact si atingerea unor rate de procesare de mai multe instructiuni / tact. Aceste procesoare 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 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. Desigur ca aceste arhitecturi au o complexitate hardware deosebita determinata de detectia si solutionarea hazardului între instructiuni. De exemplu, 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 [1].

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). Si acest model arhitectural are dezavantaje precum: incompatibilitati soft între variante succesive de procesoare, necesitati sporite de memorare a programelor, etc. 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 [1].

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. 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 a blocurilor în cache, acestea se clasifica în:

- **direct mapped** - daca fiecare bloc are doar un loc unde poate aparea în cache. Maparea se face dupa formula:

  (Adresa blocului) MOD (Numarul de blocuri în cache).

- **full associative** - daca blocul poate fi plasat oriunde în cache.
- **set associative** - daca blocul poate fi plasat 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) MOD (Numarul de seturi în cache) .

Daca avem *n* blocuri într-un set, cache-ul se numeste **n - way associative.**

Categorisirea cache-urilor de la *direct maped* la *full associative* este o continuitate reala de nivele *set associative*. *Direct maped* 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.* În mod alternativ, *direct maped* poate fi gândit sa aiba *m* seturi si *full associative* sa aiba doar un singur set. Majoritatea procesoarelor de astazi sunt *direct maped, two - way associative*, *four - way associative*.

Verificarea existentei blocurilor în cache se face dupa *tag*. Cache-urile au un tag 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 de la CPU. 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 validitate** 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.

Accesele la cache pot fi cu hit, la potrivirea tag-ului 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 *bloc 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 doua strategii de baza pentru selectarea blocului care trebuie înlocuit:

- **Random** (aleatoare) - pentru a întinde alocarea uniforma, blocurile candidate sunt selectate aleator. Un atu al înlocuirii aleatoare este ca e simplu de construit 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 Localizare*: 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*) este mai rea decât random sau LRU, conform literaturii [4].

Din punct de vedere al politicii de scriere în cache distingem 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 C*lean* (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. La write through nu exista evacuare de bloc la

cache-urile mapate direct, 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 sunt schimbatoare: 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.

3. Desfasurarea lucrarii

3.1. Descrierea simulatorului

3.1.1. Elementele simulatorului. Schema bloc

Arhitectura superscalara are patru nivele distincte în procesarea instructiunilor. În nivelul **IF** (fetch instructiune) - se calculeaza adresa grupului de instructiuni ce trebuiesc citite din cache-ul de instructiuni sau din memoria principala; dupa citire, blocul de instructiuni este plasat în partea superioara a Buffer-ului de Prefetch. Instructiunile din partea inferioara a buffer-ului sunt selectate si trimise celui de-al doilea nivel: **ID** (decodificare instructiune) - în care sunt decodificate instructiunile aduse, se citesc operanzii din setul de registrii generali, se calculeaza adresa de salt (pentru instructiunile de ramificatie) si respectiv se calculeaza adresa de acces la memorie (pentru instructiunile LOAD sau STORE). Instructiunile sunt apoi pasate unitatilor functionale potrivite, care folosesc operanzii sursa în timpul celei de-a treia faze de procesare a pipe-ului: **ALU/MEM.** În aceasta faza se executa operatia ALU asupra operanzilor selectati în cazul instructiunilor aritmetico-logice si se acceseaza memoria cache de date sau memoria principala (în caz de miss în cache), pentru instructiunile LOAD sau STORE. Unitatile de executie a instructiunilor LOAD si STORE sunt singurele unitati functionale care

interactioneaza în mod direct cu cache-ul de date. În final, avem cel de-al patrulea 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 copiata în registrul destinatie din setul de registrii generali [5].

Principalii parametrii ai simulatorului, ce pot fi modificati de catre utilizator sunt [2, 3]:

- **FR (rata de fetch)** - defineste marimea blocului accesat din cache-ul de instructiuni, mai precis numarul de instructiuni citite simultan din cache sau memorie 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; 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 instuctiunilor cu referire la memorie, respectiv adresa instructiunii tinta în cazul instructiunilor de salt.
- **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 concurent 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.

- **Latenta** - numarul de cicli necesari executiei instructiunilor aritmetice, de salt si cele cu referire la memorie (în cazul în care accesele pentru obtinerea datei sunt cu hit în cache). Initial are valoarea 1.
- **Cache-ul de instructiuni (IC)** - este de tip *direct mapped* - organizat în locatii (fiecare locatie contine adresa unei instructiuni). **SIZE_IC** - dimensiunea cache-ului de instructiuni are plaja de valori de la 64 locatii (128, 256, ...) pâna la 8192 locatii. O locatie va avea urmatoarea structura:

INDEX:	**VA**	**TAG**

Câmpul VA (reprezentat pe un bit) reprezinta bitul de validare a informatiei si poate lua valorile 0 (initial) sau 1 (dupa prima actualizare a cache-ului).

INDEX = data *mod* SIZE_IC

TAG = data *div* SIZE_IC

- **Cache-ul de date (DC)** - este un cache mapat direct, organizat în blocuri de capacitati parametrizabile [4, 8, 16 (maxim IBS) locatii]. Încarcarea si evacuarea datelor în cache se face la nivel de bloc si nu la nivel de locatie. Însa structura locatiei este diferita de cea a cache-ului de instructiuni:

BLOC_OFFSET:	**VA**	**TAG**	**BLOC_ADR**	**INDEX**

TAG = data *div* SIZE_DC

BLOC_OFFSET = data *mod* SIZE_DC

BLOC_ADR = (data *mod* SIZE_DC) *div* BLOC_SIZE

INDEX = data *mod* BLOC_SIZE

BLOC_SIZE - dimensiunea blocului în locatii;

SIZE_DC - dimensiunea cache-ului de date în locatii; - 64, 128, ...,8192 locatii -

data - data din cache-ul de date;

INDEX - indexul în cadrul blocului;

TAG - câmp de identificare al datei;

BLOC_OFFSET - offset-ul de bloc din cache;

BLOC_ADR - adresa de bloc.

La o cautare în cache (IC sau DC) se ia tag-ul valorii cautate si se verifica daca exista la indexul sau la offset-ul de bloc respectiv. În caz afirmativ spunem ca avem acces cu HIT, altfel MISS în cache si trebuie actualizat cache-ul.

- **Memoria principala -** (care se acceseaza numai la *miss* în cache) va avea o latenta parametrizabila de **N_PEN** (10, 15, 20) tacti procesor. În cazul acceselor de date, sunt introduse penalizari numai pentru instructiunile LOAD (la STORE nu e nevoie din cauza procesorului de iesire care pipelineizeaza scrierea în memorie, facând-o transparenta pentru procesor). 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.
- Presupunem existenta unui numar suficient de mare (maxim IRmax) de **seturi de registri generali** (**NR_REG_GEN**): un set de registrii generali este necesar pentru executia unei instructiuni de tip aritmetico-logic sau cu referire la memorie.

Presupunem un *branch prediction* perfect, adica cunoasterea în permanenta a adresei corecte a urmatoarei instructiuni ce se va executa. Mecanismul de forwarding este inhibat.

Simulatorul realizat trebuie sa elimine gâtuirile care limiteaza performanta si sa investigheze posibile schimbari (arhitecturale sau tehnici de optimizare) în scopul cresterii acesteia. Prin realizarea unui model de simulare detaliat pentru

fiecare procesor, performanta obtinuta prin simulare este capabila sa asigure un rapid *feedback* în legatura cu schimbarile propuse.

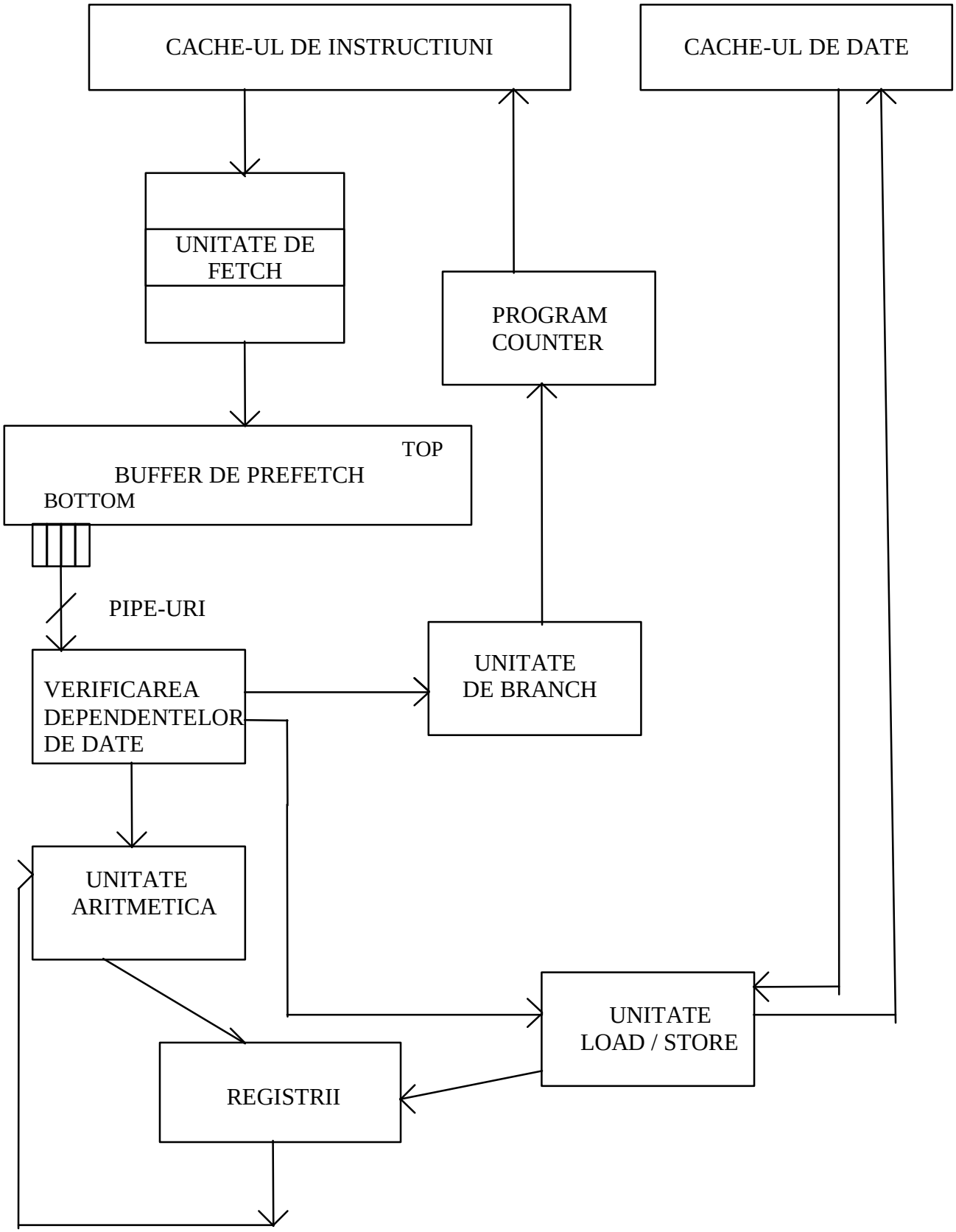

Figura 1. Schema bloc a arhitecturii superscalare simulate

3.1.2. Programele de test Stanford

Evaluarea performantelor arhitecturii se face prin simulare, fara simulare fiind foarte dificil de estimat. Metoda folosita este *trace driven simulation*. Programele benchmark de numere întregi Stanford, sunt o suita de opt programe care necesita putina initializare de date si care sunt gândite sa manifeste comportamente similare cu scopul general al programelor de calculator, desi unele au o natura destul de recursiva. Programele implica executia a 100 pâna la 900 de mii de instructiuni. Codul original C este întâi trecut printr-un compilator "*gnu C"* care produce formatul corect al codului în mnemonica de asamblare (fisiere *.ins), precum si directive de asamblare, comenzi de alocare a datelor. Codul este apoi executat pe un simulator la nivel de instructiuni, care produce la iesire un fisier trace de instructiuni (fisiere *.trc). În final aceste trace-uri constiuie programe de test în simularea unui sir de arhitecturi de cache, pentru determinarea optimului de performanta în anumite conditii de intrare. De remarcat ca, la rularea repetata a aceluiasi program C aferent oricaruia din cele opt benchmark-uri se obtine acelasi fisier Trace.

Spre exemplificare prezentam primele doua linii de cod din programul de test **fsort.trc** [5].

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 [2, 3].

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 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 [2, 3, 5].

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

Cele opt benchmark-uri - *bubble, sort, perm, puzzle, queens, matrix, tree* si *tower* - desi relativ scurte, sunt caracterizate de calcul intensiv si au o dinamica ridicata a numarului de instructiuni. Un numar de benchmark-uri folosesc recursivitatea: *perm, tower* 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 [2, 3, 5].

Benchmark-ul *tower* reprezinta programul de rezolvare a problemei turnurilor din Hanoi pentru sapte discuri. Benchmark-urile *bubble, tree* si *sort* reprezinta trei programe bazate pe tehnici de sortare diferite. Benchmark-ul *matrix* presupune calcularea produsului a doua matrici. Benchmark-ul *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 nu se extinde si la aplicatiile grafice si multimedia, rutine critice ale sistemelor de operare.

Observatii

Pentru simulare, pe lânga cele 8 fisiere *trace* (*.trc) sunt necesare 8 fisiere identice ca nume având extensia (*.txt). Aceste fisiere (menite sa pastreze doar instructiunile), sunt o prelucrare proprie a programelor scrise în mnemonica de asamblare (*.ins), cu scopul de a ajuta la determinarea dependetelor de date reale existente între instructiuni (RAW). [2, 3]

3.2. 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 *blcache.exe* generati [2]:

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. Studiati influenta capacitatii cache-ului de instructiuni asupra ratei de procesare IR(SIZE_IC) si asupra ratei de miss la cache-ul de instructiuni R_{missIC}(SIZE_IC).
3. Studiati influenta capacitatii cache-ului de date asupra ratei de procesare IR(SIZE_DC) si asupra ratei de miss la cache-ul de date R_{missDC}(SIZE_DC).
4. Determinati influenta numarului maxim de instructiuni ce pot fi trimise simultan în executie asupra ratei de procesare IR(IRmax).
5. La acest punct nu se va mai considera, ca pâna acum, numar nelimitat de seturi de registri generali. În simularea efectuata la punctele 1÷4 valoarea parametrului NR_REG_GEN este identica cu cea a IRmax. Se va determina numarul optim de seturi de registrii (2, 3, 4, ...IRmax) - IR(NR_REG_GEN) în variantele cu cache de date uniport (o singura instructiune cu referire la

memorie se poate executa) sau biport (doua instructiuni cu referire la memorie se pot executa: L+L sau L+S).

6. Pentru valoarea optima determinata la punctul 5 a numarului de seturi de registrii generali, studiati comparativ performanta (rata de procesare) pe doua tipuri de cache de date (uniport sau biport) - IR(NR_UNIT_LS).

Simulatorul *blcache.exe* nu simuleaza procesul de scriere în cache. Acest lucru va fi realizat cu simulatoarele *blwbcach.exe* (tehnica de scriere în cache este write back) si *blwtcach.exe* (tehnica de scriere în cache este write through).

7. Se vor genera graficele IR(BLOC_SIZE) si R_{missDC}(BLOC_SIZE) în cele doua ipostaze: scriere în cache prin write back si scriere în cache prin write through. Se va studia comparativ realismul, prin rata de procesare, introdus prin cele doua tehnici de scriere fata de situatia când nu se foloseste nici una din aceste tehnici IR(tehnica de scriere în cache).

Bibliografie

[1] **Vintan L.** – *Metode de evaluare si optimizare în arhitecturile paralele de tip I.L.P.*, Editura Universitatii "Lucian Blaga" Sibiu, Sibiu, 1997.

[2] **Florea A.** – *Optimizarea proceselor de scriere într-o arhitectura RISC superscalara de tip Harvard,* Teza de Masterat, Sibiu, 1998 (îndrumator L. Vintan).

[3] **Florea A.** – *Simulator pentru o arhitectura superscalara parametrizabila de tip Princeton*, Lucrare de Licenta, Sibiu, 1997 (îndrumator L. Vintan).

[4] **Hennessy J., Patterson D.** – *Computer Architecture*: *A quantitative Approach*, Morgan Kaufmann Publishers, Second Edition, 1996.

[5] **Collins R.** – *The HSA Simulator*, University of Hertfordshire, UK, Technical Report, 1993.