CUPRINS

PREFATA

Prezenta lucrare se constituie într-o modesta contributie didactica, elaborata în scopul obtinerii unor deprinderi si cunostinte practice legate de arhitectura microprocesoarelor si a sistemelor cu microprocesoare. Astfel, pe lânga aplicatii de genul "*lucrari practice de laborator*", lucrarea contine si numeroase probleme de proiectare situate preponderent la nivelul interfetei hardware-software.

Primele 4 aplicatii practice urmaresc exploatarea unor modele si interfete de baza din cadrul clasicei arhitecturi IBM-PC. În acest context, se exerseaza arhitectura microprocesorului, a interfetelor seriale si paralele la nivelul unor transferuri de date prin interogare/întreruperi si respectiv a interfetei DMA.

Prin urmatoarele 3 lucrari se aprofundeaza arhitectura familiei de microprocesoare RISC MIPS R2000/3000 prin intermediul unui simulator software numit SPIM, conceput de catre James R. Larus de la Universitatea din Wisconsin, SUA. Am ales arhitectura MIPS datorita faptului ca pe lânga succesul sau comercial, constituie unul dintre cele mai inteligibile si elaborate microprocesoare RISC.

Cele 3 aplicatii care urmeaza au în vedere studiul celebrului microprocesor virtual, elaborat de catre profesorii John Hennessy (Univ. Stanford, SUA) si David Patterson (Univ. Berkeley, SUA), abreviat DLX. Procesorul DLX reprezinta o arhitectura RISC superscalara, didactica dar si performanta, care este investigata prin intermediul unui excelent simulator software, înzestrat cu facilitati didactice deosebite si cu o grafica atragatoare.

Ultimele 2 aplicatii practice urmaresc evaluarea si optimizarea unor elemente esentiale – memoriile cache si respectiv predictorul de ramificatii – ce caracterizeaza practic toate microprocesoarele superscalare avansate. Investigatiile propuse se realizeaza cu ajutorul a trei simulatoare parametrizabile scrise în C, elaborate în cadrul grupului de cercetare în arhitecturi avansate de la Catedra de Calculatoare si Automatica a Facultatii de Inginerie din Sibiu.

La final am propus un consistent set de peste 40 de probleme originale si fecunde, însotite de indicatii pentru solutionare. Consideram ca numai prin solutionarea sistematica de aplicatii diverse si interesante se poate aprofunda la un nivel corespunzator o asemenea disciplina tehnica de vârf, prin excelenta aplicativa.

Ideea care ne-a calauzit în elaborarea lucrarii, a constat în aceea ca domeniul arhitecturii sistemelor de calcul nu este unul preponderent descriptiv, caracterizat prin sute de biti, registri, terminale, etc. a caror functie trebuie memorata si – cel mult – înteleasa (fatalmente incomplet sau nociv). Dimpotriva, domeniul este unul viu si chiar fascinant, în care investigatia personala si creatia sunt posibile prin tehnici ca cele prezentate aici. Altfel spus, drumul de la arhitectura, vazuta ca setul tuturor implementarilor posibile, la o anumita instanta optimala a acesteia numita procesor, constituie o adevarata aventura formativa. Alegerea respectivei "instante" trebuie sa tina cont esentialmente de destinatia acelui procesor d.p.d.v. al utilizatorului sau (aplicatii în limbaje evoluate).

Aceasta lucrare se adreseaza atât studentilor de la specializarea "*Stiinta Calculatoarelor*" (anul 3) cât si celor de la specializari precum "*Electrotehnica*", "*Electronica*", "*Informatica Aplicata*", etc. De asemenea, consideram ca lucrarea poate fi utila tuturor celor interesati de arhitectura sistemelor de calcul.

Încheiem prin a exprima gratitudinea noastra profesorului Gordon B. Steven de la Universitatea din Hertfordshire, Anglia, care ne-a furnizat benchmark-urile Stanford compilate pentru procesorul HSA (Hatfield Superscalar Architecture) si a trace-urilor aferente. De asemenea, multumirile noastre d-nului dipl. Ing. Calin Brândasu pentru elaborarea lucrarii nr. 8 ("*Arhitectura microprocesoarelor DLX"*) respectiv d-nului absolvent Ciprian Cândea pentru aportul sau decisiv la elaborarea lucrarii nr. 4 ("*Controler-ul DMA*").

2 Februarie 1999 *Autorii*

Sibiu

SCRIEREA PROGRAMELOR ÎN ASAMBLARE LA CALCULATOARELE COMPATIBILE IBM – PC

1. Scopul lucrarii

Lucrarea de fata îsi propune prezentarea etapelor de realizare precum si structura unui program în limbaj de asamblare al procesorului Intel 80x86.

2. Memento teoretic

Programele în limbaj de asamblare sunt scrise sub forma de fisiere sursa, care pot fi asamblate cu ajutorul unui program asamblor (MASM, TASM), în fisiere obiect. Legaturile între mai multe module obiect asociate aceluiasi program pot fi facute folosind un program link-editor (LINK, TLINK), pentru a forma fisiere executabile. Etapele de realizare a unui program executabil sunt:

i. editarea fisierului sursa (.ASM), folosind un editor ASCII;

ii. asamblarea fisierului sursa, folosind programul asamblor, obtinându-se fisierul obiect (.OBJ);

iii. editarea legaturilor (link-editarea) pentru fisierul (fisierele) obiect, obtinându-se programul executabil (.EXE).

iv. daca programul este scris în format .COM se va folosi utilitarul EXE2BIN pentru obtinerea programului executabil .COM;

v. rularea programului, eventual depanarea acestuia folosindu-se un program depanator (debugger) ca: DEBUG, TURBO DEBUGGER, SST.

Fisierele sursa sunt formate din instructiuni în limbaj de asamblare, acestea fiind formate la rândul lor, din eticheta, mnemonica, operanzi si comentariu,

despartite prin spatii sau caractere TAB. Mnemonicele pot fi de doua feluri: comenzi date asamblorului (directive) sau instructiuni propriuzise. Directivele specifica modul în care asamblorul va genera codul obiect în momentul asamblarii.

Directive referitoare la definirea structurii de segmente.

Un segment reprezinta o colectie de instructiuni sau de date ale caror adrese sunt raportate la acelasi registru segment. Segmentele din cadrul unui program pot fi definite utilizând definitiile de segment simplificate sau cele complete.

Definirea simplificata a segmentelor utilizeaza în mod implicit conventiile limbajelor de nivel înalt (C). Pentru folosirea definitiilor de segmente simplificate se va declara un model de memorie pentru program. Acesta specifica marimea datelor si codului utilizate în program. Modelele de memorie uzuale sunt:

TINY - datele si codul programului se încadreaza într-un singur segment (64K);

SMALL - datele ocupa un singur segment si codul ocupa un singur segment;

MEDIUM - datele ocupa un singur segment, în timp ce codul poate ocupa mai multe segmente (accesarea codului se face cu referinta îndepartata);

COMPACT - codul ocupa un singur segment, datele ocupând mai multe segmente (accesarea datelor se va face cu referinta îndepartata);

LARGE - atat datele cât si codul pot ocupa mai multe segmente;

HUGE - codul si datele pot ocupa fiecare mai mult de un segment. În plus, pot exista tablouri de date mai mari de 64K.

Definirea modelului de memorie folosit se face cu directiva:

.*MODEL* <nume model>.

Începutul unui segment este indicat prin una din urmatoarele directive:

.*STACK* <marime>	- segment de stiva;
.*CODE* <nume>	- segment de cod;
.*DATA*	- segment initializat de date cu referinta apropiata;
.*DATA?*	- segment neinitializat de date cu referinta apropiata;
.*FARDATA*	- segment initializat de date cu referinta îndepartata;
.*FARDATA?*	- segment neinitializat de date cu referinta îndepartata;
.*CONST*	- segment de date constante.

Datele din segmentele definite cu directivele .*STACK*, .*CONST*, .*DATA*, .*DATA*? sunt plasate într-un grup de segmente denumit *DGROUP*.

Pentru definiri complete de segmente, începutul unui segment este definit cu directiva:

<nume> *SEGMENT* <aliniere>, <combinare>, <utilizare>, <‘clasa’>

iar sfârsitul segmentului este indicat prin folosirea directivei:

<nume> *ENDS*

Alinierea unui segment caracterizeaza adresa unde poate începe un segment. Alinierea poate fi:

BYTE - aliniere la octet;

WORD - aliniere la cuvânt;

DWORD - aliniere la dublu cuvânt;

PARA - aliniere la paragraf (16 octeti) - utilizat implicit;

PAGE - aliniere la pagina (256 octeti).

Câmpul combinare defineste modul de combinare al segmentelor ce au acelasi nume. Ea poate fi:

PUBLIC - concateneaza segmentele cu acelasi nume;

STACK - concateneaza segmentele formând un segment raportat la registrul segment SS;

COMMON - creaza segmente suprapuse prin plasarea adreselor de început ale segmentelor cu acelasi nume la aceeasi adresa;

AT adresa - etichetele si variabilele segmentului vor fi relative la adresa.

Câmpul utilizare defineste marimea cuvântului segmentului atunci când se foloseste un procesor 386/486:

USE16 - cuvântul este de 16 biti;

USE32 - cuvântul este de 32 de biti.

Tipul ‘clasa’ este un mijloc de asociere al segmentelor cu nume diferite si scopuri similare. Acest câmp se foloseste pentru a controla ordinea segmentelor si a identifica segmentele de cod (clasa ‘CODE’).

Pentru referirea cu un singur registru segment a mai multor segmente de date, definite separat în programul sursa, acestea se pot grupa folosind directiva:

nume *GROUP* segment <,segment>...

Când asamblorul necesita referirea unei adrese el trebuie sa cunoasca segmentul în care se afla adresa. Acest lucru se face folosind directiva:

ASSUME registru segment : nume...

unde nume este numele segmentului sau al grupului ce se va asocia cu registrul segment specificat. Atentie: directiva ASSUME indica segmentul asociat doar asamblorului, nu si procesorului.

Initializarea registrelor segment.

Registrele CS si IP sunt intializate prin specificarea unei adrese de început (etichete) cu directiva:

END <adresa început>

Registrul DS trebuie initializat la adresa segmentului ce va fi folosit pentru date. Daca se foloseste definirea simplificata a segmentelor, initializarea registrului DS poate fi facuta în modul urmator:

```
mov ax,@data
mov ds, ax
```

Daca se folosesc definitii complete:

```
mov ax, <nume segment>
mov ds, ax
```

Registrul SS este initializat automat la valoarea segmentului *.STACK* în cazul definirii simplificate, respectiv la valoarea ultimului segment cu tipul de combinare *STACK*.

Registrul ES nu este initializat automat, putând fi folosit pentru alte referiri.

Folosirea specificatorilor de tip.

Pentru specificarea dimensiunii unei variabile sau a unui operand se pot folosi urmatoarele cuvinte cheie:

BYTE	- variabila octet;
WORD	- variabila cuvânt;
DWORD	- variabila dublu cuvânt;
FWORD	- variabila de 6 octeti;
QWORD	- variabila de 8 octeti;
TBYTE	- variabila de 10 octeti.

Modul în care este referita o variabila poate fi specificat prin:

FAR	- referinta îndepartata (segment + offset);
NEAR	- referinta apropiata (offset);

PROC - referinta cu tipul implicit al modelului curent de memorie.

Folosirea procedurilor.

Sintaxa pentru definirea unei proceduri este:

eticheta *PROC <NEAR/FAR>*

instructiuni

eticheta *ENDP*

Etichete de cod.

Înafara de etichetele create prin definirea procedurilor, mai pot exista doua tipuri de etichete:

- etichete cu referinta apropiata, cu sintaxa:

nume:

- etichete definite cu directiva LABEL, cu sintaxa:

nume LABEL distanta

unde distanta poate fi: NEAR, FAR sau PROC.

Definirea de variabile.

Variabilele sunt formate din unul sau mai multe obiecte de date de marime specificata. Sintaxa:

<nume> directiva <initializator>

Marimea datelor este data de directiva folosita:

DB - defineste variabila octet;
DW - defineste variabila cuvânt;
DD - defineste variabila dublu cuvânt;
DF - defineste variabila de 6 octeti;
DQ - defineste variabila de 8 octeti;
DT - defineste variabila de 10 octeti.

Pentru definirea tablourilor se foloseste operatorul *DUP*:

numar *DUP* (valoare initiala)

Exemplu: *DB* 10 *DUP* (0) - aloca 10 octeti cu valoarea initiala 0.

Utilizarea structurilor si înregistrarilor.

O variabila structura reprezinta o colectie de date de dimensiuni diferite ce pot accesate prin intermediul aceleiasi variabile. Sintaxa de declarare a unei structuri:

nume structura *STRUC*
declaratii câmp
nume structura *END*

Pentru declararea unei variabile de tip structura se foloseste sintaxa:

nume nume structura <valoare initiala>...

Pentru accesarea unui anumit câmp al unei structuri se foloseste operatorul '.':

nume.câmp

O variabila de tip înregistrare este o variabila de lungime octet sau cuvânt unde anumite câmpuri de biti pot fi accesate simbolic. Declararea unei înregistrari:

nume înregistrare *RECORD* câmp,...

unde câmp reprezinta:

nume_câmp: marime <=expresie>

Exemplu: color *RECORD* blink:1, back:3, intense:1, fore:3

Declararea unei variabile de tip înregistrare se face dupa sintaxa:

nume nume înregistrare <valoare initiala>

Utilizarea macrodefinitiilor.

Macrodefinitiile permit atribuirea unui nume simbolic pentru un bloc de instructiuni în vederea folosirii numelui respectiv pentru referirea la blocul de

instructiuni. Pentru o macrodefinitie se pot defini parametri ce îi vor fi pasati. Definirea unei macrodefinitii se fece dupa sintaxa:

nume macrodefinitie *MACRO* <parametru>...
instructiuni
ENDM

Apelarea unui macro se va face:

nume macrodefinitie <argument>...

Definirea etichetelor în cadrul unei macrodefinitii trebuie sa se faca cu directiva *LOCAL*. Sintaxa:

LOCAL nume eticheta <, nume eticheta>...

Exemplu:
```
aduna MACRO ad1,ad2
    mov ax,ad1
    add ax,ad2
ENDM
```

Apelarea: aduna bx,10

Crearea programelor din module multiple.

Programele în limbaj de asamblare de dimensiuni medii si mari sunt create pornind de la mai multe fisiere sursa. Pentru folosirea globala a unor simboluri,

definite într-unul din module, în modulul de definitie simbolurile se vor declara folosind directiva:

PUBLIC nume simbol<,nume simbol>...

În modulele în care se folosesc simboluri declarate alte module, ele se vor declara cu sintaxa:

EXTERN nume simbol:tip <nume simbol:tip>...

Folosirea operatorilor.

Cei mai utilizati operatori pusi la dispozitie de macroasambloarele pentru I8086 sunt:

PTR - specifica tipul pentru o variabila sau o expresie:

tip *PTR* expresie

Exemplu: *WORD PTR* es;[si]

THIS - creaza un operand de tip specificat, ale carui valori de deplasament si segment sunt egale cu valoarea curenta a contorului de locatii. Sintaxa:

THIS tip

SEG - întoarce adresa segmentului unei expresii. Sintaxa:

SEG expresie

OFFSET - întoarce deplasamentul unei expresii. Sintaxa:

OFFSET expresie

SIZE - întoarce numarul total de octeti pentru o variabila definita cu operatorul *DUP*:

SIZE variabila

$ - reprezinta adresa instructiunii curente ce se executa

EQU - folosit pentru a atribui o constanta numerica unui simbol. Sintaxa:

nume *EQU* expresie

ORG - pozitioneaza contorul de locatii la un nou deplasament. Sintaxa:

ORG deplasament

3. Exemple de program rezolvate si probleme propuse

Problema rezolvata:

Testarea mecanismului de prefetch la microprocesorul 8086.

```
cod     segment PARA ‘CODE’
        assume cs:cod, ds:cod
        org 100h

merr    db 13,10,’8086 greseste prin mecanismul de prefetch!!!’,13,10,’$’
ok      db 13,10,’Acest 8086 nu greseste!!!’,13,10,’$’
mes     db 13,10,’Test mecanism de prefetch la microprocesor.’,13,10,’$’

begin:  mov dx,cs
        mov ds,dx
        mov dx,offset mes       ;afisarea scopului programului
        mov ah,09h
        int 21h
        jmp start               ;golire QFIFO
start:  mov si,offset et
        mov al,0aah
        mov [si+1], al          ;modificarea instructiunii urmatoare in memorie:
        mov cx,0ffaah !!
et:     mov cx,0ffffh
        sub cx,0ffaah
        jz et1
        mov dx,offset merr      ;8086 nu simte modificarea in QFIFO
        jmp et2
et1:    mov dx,offset ok                ;8086 simte modificarea
et2:    mov ah,09h
        int 21h
        mov ax,4c00h
```

```
    int 21h
cod ends
end begin
```

Sa se ruleze programul de mai sus atât prin lansare de la prompt-ul DOS, cât si pas cu pas folosind un program de depanare (DEBUG, SST, TURBO DEBUGGER, etc).

Problema propusa:

Sa se studieze comportarea microprocesorului 8086 la accesarea unui operand pe cuvânt de la o adresa de offset 0ffffh.

Bibliografie

[1] **Caprariu V., Enyedi A., Muntean M.** – *Sistemul de operare DOS*: *Ghidul Programatorului*, Editura MicroInformatica, Editia III, Cluj Napoca, 1993.

[2] **Athanasiu I., Panoiu A.** – *Calculatoare Personale*: *Microprocesoarele 8086, 286, 386*, Editura Teora, Bucuresti, 1993.

UTILIZAREA COMUNICATIEI SERIALE ÎN IBM – PC

1. Memento teoretic

Standardul RS 232C este cel mai folosit standard de cuplare seriala a doua echipamente de calcul. Cuplele de legatura seriala prezente în PC sunt cu 9, respectiv 25 de pini. Semnalele prezente în cupla cu 9 pini sunt:

pin 1: DCD (Data Carrier Detect) - intrare, indica detectarea de catre modem a purtatoarei.
pin 2: RD - intrare, receptie date.
pin 3: TD - iesire, transmisie date.
pin 4: DTR (Data Terminal Ready) - iesire, calculator pregatit pentru receptie.
pin 5: GND - masa.
pin 6: DSR (Data Send Ready) -intrare, modem cuplat la linia telefonica.
pin 7: RTS (Request To Send) - iesire, cerere de emisie din partea calculatorului.
pin 8: CTS (Clear To Send) -intrare, gata de emisie, raspuns al modemului la activarea RTS.
pin 9: RI (Ring Indicator) - intrare, indica detectia de catre modem a unei frecvente de apel.

Din punct de vedere electric la transmisia seriala prin acest standard se opereaza cu urmatoarele nivele de tensiune:

	0 logic	1 logic
emisie	-15 ÷ -5	5 ÷ 15
receptie	-15 ÷ -3	3 ÷ 15

Lungimea maxima a liniei de comunicatie pentru care schimbul de informatie este corect este de 15 m.

În PC comunicatia seriala se poate face prin intermediul a patru porturi COM. Zona de date BIOS contine la adresele 0:0400h..0:0407h o lista a celor paru adrese de baza corespunzatoare celor patru porturi.

În cursul rutinei de initializare (POST), se testeaza si se initializeaza porturile COM1 si COM2. Adresele de port corespunzatoare acestora sunt:

COM1 - 3F8h..3FFh
COM2 - 2F8h..2FFh

Asupra celor patru porturi se poate opera direct, prin instructiuni de intrare/iesire (IN, OUT), sau prin intermediul întreruperii BIOS 14h. Aceasta întrerupere va lucra cu oricare din cele patru porturi, cu conditia ca adresa de baza a portului serial solicitat sa se gaseasca în tabela din zona de date BIOS. Totodata, este necesar ca doua porturi sa nu împarta acelasi spatiu de adresare, caz în care nici unul nu va functiona corect.

Utilizarea porturilor seriale se poate face prin interogare (*polling*), sau prin întreruperi. În acest din urma caz liniile de întrerupere folosite, corespunzatoare controlorului de întreruperi 8259A sunt:

COM1 - IRQ4 - INT 0Ch
COM2 - IRQ3 - INT 0Bh

Registri asociati unui cuplor de comunicatie seriala sunt (adresele sunt date pentru COM1):

Adresa
3F8h *Scriere*: registru de date, continând cei 8 biti ai caracterului ce trebuie transmis. Când bitul DLAB=1, contine octetul inferior al divizorului frecventei

ceasului, care împreuna cu octetul superior, scris la adresa 3F9h determina rata transmisiei seriale dupa cum urmeaza:

Rata (baud)	Valoare de divizare (zecimal)
110	1040
150	768
300	384
600	192
1200	96
2400	48
4800	24
9600	12

Citire: buffer de receptie, continând cei 8 biti ai caracterului receptionat.

3F9h *Scriere*: Când bitul DLAB=1 contine octetul superior al valorii de divizare a frecventei ceasului. Când DLAB=0 reprezinta registrul de validare întreruperi:

Bit: 0 - 1=valideaza generarea unei întreruperi la receptie de date.

1 - 1=valideaza generarea unei întreruperi când bufferul de emisie este gol.

2 - 1=valideaza generarea unei întreruperi la detectarea unei erori sau oprire.

3 - 1=valideaza generarea unei întreruperi la schimbarea starii modem-ului (CTS, DSR, RI, RLSD).

4..7 - au valoarea 0.

3FAh *Citire*: registrul cauza a întreruperii, La aparitia unei întreruperi citirea acestui registru determina natura întreruperii.

Bit: 0 - 1=întrerupere activa (poate fi folosit pentru interogare).

1, 2 -00=întrerupere cauzata de o eroare (suprascriere, paritate, încadrare), sau oprire. Este resetat prin citirea registrului de stare a liniei (3fdh).

-01=date receptionate disponibile. Este resetat prin citirea bufferului de receptie (3F8h).

-10=buffer de emisie gol. Este resetat prin scrierea în bufferul de emisie (3F8h).

-11=întrerupere cauzata de schimbarea starii modem-ului. Este resetat prin citirea registrului de stare a modem-ului (3FEh).

3..7=sunt 0.

3FBh *Citire/scriere*: registrul de control al liniei.

Bit: 0, 1: lungimea cuvântului de date: 00=5, 01=6, 10=7, 11=8.

2: numarul de biti de stop: 0=1, 1=2.

3,4: paritate: x0=fara, 01=impara, 11=para.

5: nu este folosit de catre BIOS.

6: valideaza controlul opririi. 1=transmisia începe prin emiterea de 0-uri (spatii).

7: DLAB (Divisor Latch Access Bit). 1=se programeaza rata de transmisie, 0=normal.

3FCh *Scriere*: registrul de control al modem-ului.

Bit: 0: 1=activeaza DTR, 0=deactiveaza DTR.

1: 1=activeaza RTS, 0=deactiveaza RTS.

2: 1=activeaza OUT1

3: 1=activeaza OUT2 (necesar în cazul lucrului prin întreruperi).

4: 1=activeaza bucla pentru testare.

5..7: sunt 0.

3FDh *Citire*: registrul de stare a liniei. Bitii 1..4 cauzeaza generarea unei întreruperi daca aceasta este validata.

Bit: 0: 1=date receptionate. Este resetat prin citirea bufferului de receptie.

1: 1=eroare de suprascriere, caracterul precedent fiind pierdut.

2: 1=eroare de paritate. Este resetat prin citirea registrului de stare a liniei.

3: 1=eroare de încadrare; caracterul receptionat contine un bit de stop invalid.

4: 1=este indicata oprirea; se receptioneaza spatii.

5: 1=bufferul de emisie gol, este ceruta emisia urmatorului caracter.

6: 1=transmitatorul este inactiv, nici o data nefiind procesata.

3FEh *Citire*: registrul de stare al modem-ului. Bitii 0..3 determina generarea unei întreruperi daca aceasta este validata.

Bit: 0: 1=delta CTS si-a schimbat starea.

1: 1=delta DSR si-a schimbat starea.

2: 1=se detecteaza fronturi ale semnalului RI.

3: 1=delta DCD si-a schimbat starea.

4: 1=CTS este activ.

5: 1=DSR este activ.

6: 1=RI este activ.

7: 1=DCD este activ.

Întreruperea 14h pune la dispozitie urmatoarele servicii pentru interfata seriala:

AH=00h - initializarea portului serial.

La apel:

DX=numarul portului (0,1).

AL=parametri de initializare, conform urmatoarelor câmpuri:

Bit: 0,1: lungimea cuvântului de date (10=7 biti, 11=8 biti).

2: numarul bitilor de stop (0=1, 1=2).

3,4: paritatea (x0=fara, 01=impara, 11=para).

5..7: rata de transmisie: 000=110, 001=150, 010=300, 011=600, 100=1200, 101=2400, 110=4800, 111=9600.

La revenire:

AH=starea liniei seriale (vezi mai jos).

AH=01h - emisie caracter.

La apel:

DX=numarul portului (0,1).

AL=caracterul ce trebuie emis.

La revenire:

AL=caracterul emis.

Daca bitul 7 al lui AH este setat a intervenit o eroare si ceilalti 7 biti ai lui AH contin starea liniei de comunicatie (vezi mai jos).

AH=02h - receptie caracter.

La apel:

DX=numarul portului (0,1).

La revenire:

AL=caracterul emis.

AH este diferit de zero daca a aparut o eroare.

AH=03h - citirea starii portului serial.

La apel:

DX=numarul portului (0,1).

La revenire:

AX=starea portului serial

AH - starea liniei de comunicatie

bit 7: timeout.

bit 6: registrul de deplasare la transmisie gol.

bit 5: buffer de transmisie gol.

bit 4: oprire detectata.

bit 3: eroare de încadrare.

bit 2: eroare de paritate.

bit 1: eroare de suprascriere.

bit 0: date receptionate.

AL - starea modem-ului

bit 7: DCD

bit 6: RI

bit 5: DSR

bit 4: CTS

bit 3: delta DCD.

bit 2: fronturi RI.

bit 1: delta DSR.

bit 0: delta CTS.

2. Programe demonstrative.

Exemplu 1: Utilizarea interfetei seriale prin polling.

```
.model small
```

```
;tabela adreselor de port ale registrelor interfetei seriale
COM2_DATE           equ   02F8h
COM2_VAL_INTR       equ   02F9h
COM2_TIP_INTR       equ   02FAh
COM2_FORM_DATE      equ   02FBh
COM2_CTRL_MODEM     equ   02FCh
COM2_STARE          equ   02FDh
COM2_VITEZA         equ   02F8h

        .stack 100h
        .data
        .code
start:
;initializarea interfetei seriale
        mov ax,@data
        mov ds,ax
        mov dx,COM2_FORM_DATE
        mov al,80h                  ;urmeaza programarea ratei de transmisie
        out dx,al
        mov dx,COM2_VITEZA          ;programarea octetului mai putin semnificativ
        mov al,60h                  ;9600 baud
        out dx,al
        xor al,al                   ;programarea octetului mai semnificativ
        inc dx
        out dx,al
        mov al,03h                  ;setarea  formatului datelor:
        mov dx,COM2_FORM_DATE       ;8 biti, 1 bit stop, fara paritate
```

```
        out dx,al
        mov al,0Bh                  ;activare DTR si RTS
        mov dx,COM2_CTRL_MODEM
        out dx,al
        xor al,al                   ;invalidare întreruperi
        mov dx,COM2_VAL_INTR
        out dx,al
        mov dx,COM2_STARE           ;verificare daca exista un caracter
        in al,dx                    ;în bufferul de receptie
        and al,01h
        jz bucla
        mov dx,COM2_DATE            ;daca da, este citit
        in al,dx

bucla:  mov ah,0                    ;asteptarea unei taste
        int 16h
        cmp al,1Bh                  ;daca este ESC
        je iesire                   ;se face saltul la iesire
        push ax
        call print                  ;afisarea caracterului citit
        mov dx,COM2_STARE           ;asteptare buffer emisie gol
et1:    in al,dx
        and al,20h
        jz et1
        pop ax
        mov dx,COM2_DATE            ;emitere caracter citit
        out dx,al
```

```
        mov dx,COM2_STARE       ;asteptare buffer receptie plin
et2:    in al,dx
        and al,01h
        jz et2
        mov dx,COM2_DATE        ;citire caracter receptionat
        in al,dx
        call print              ;afisare caracter receptionat
        jmp bucla
iesire: mov ax,4C00h
        int 21h

print:  push ax                 ;afisare caracter
        push dx
        mov dl,al
        mov ah,02h
        int 21h
        pop dx
        pop ax
        ret

end start
```

Exemplu 2: Utilizarea interfetei seriale prin întreruperi cu monitorizare pe display.

```
        .model tiny

;tabela cu deplasamentele registrelor fata de baza
rxdat           equ    0        ;registru date receptie
```

```
txdat       equ   0      ;registru date emisie
lowsp       equ   0      ;registru divizor viteza low
higsp       equ   1      ;registru divizor viteza high
itenb       equ   1      ;registru validare întreruperi
intid       equ   2      ;registru identificare întreruperi
lcont       equ   3      ;registru control linie
mcont       equ   4      ;registru control modem
txint       equ   2      ;masca indicator întrerupere emisie

;lungime barete de monitorizare a functionarii pe display
lbe         equ   1ceh   ;bareta solicitari emisie
lbr         equ   30eh   ;bareta solicitari receptie

;masti
iniv4       equ   0efh   ;masca nivelului de întrerupere 4

cseg   segment
       org 100h
       assume cs:cseg, ds:cseg, es:cseg,ss:cseg

exmp   proc far
       mov ax,40h
       mov es.ax
       mov dx,es:[0]      ;adresa pentru COM1 (0:0400)
       mov comx,dx        ;se memoreaza în segmentul program

;initializare UART si întreruperi

       mov ax,lcont
       mov bl,10000000b ;DLAB=1
```

```
        call putb           ;trimite octetul din bl în registrul dat de ax
        mov ax,higsp
        mov bl,3            ;3 în divizor high
        call putb
        mov ax, lowsp
        xorbx,bx            ;0 în divizor low
        call putb           ;viteza va fi 150 baud
        mov ax,lcont
        mov bl,1            ;8 biti, fara paritate, 1 bit stop
        call putb
        mov ax,mcont
        mov bl,0bh          ;activeaza DTR,RTS
        call putb

;pregatirea vectorului de întrerupere

        mov dx,offset ioint ;deplasament rutina în segment
        mov ax,250ch        ;functia DOS 25h, vectorul 0c (COM1)
        int 21h             ;apel DOS

;activarea întrruperilor în interfata

        mov ax,itenb
        mov bl,3            ;validare întreruperi emisie si receptie
        call putb
        in al,21h           ;citeste registrul de mascare al 8259A
        and al,iniv4
        nop
        nop
```

```
        nop
        nop
        out 21h,al          ;se demascheaza nivelul 4 de întreruperi

;din acest moment UC si-a terminat sarcina de initializare a interfetei. În continuare
;controlul UC este predat sistemului de operare care de regula aloca UC altor
;programe. Prin sistemul de întreruperi însa, de câte ori interfata termina de emis sau
;de receptionat un caracter, UC este întrerupta si obligata sa  execute rutina de
;servire a interfetei seriale, numita aici IOINT. Durata de executie a rutinei este mult
;inferioara intervalului la care este ea solicitata. În restul timpului UC este la
;dispozitia altor utilizatori. Controlul este dat aici sistemului de operare cu optiunea
;ca zona de memorie în care se gaseste acest program, deci si IONIT sa nu mai fie
;alocata altor programe, astfel ca rutina de servire sa ramâna nealterata.

        lea dx,rezrm        ;limita rezervarii de memorie
        int 27h             ;revenire în DOS cu ramânere rezidenta

;urmeaza rutina de servire a întreruperii propriuzisa. Spre deosebire de partea
;anterioara, rutina se executa la fiecare solicitare de asistenta a UC din partea
;interfetei seriale.
;
ioint:  sti                 ;validarea înteruperilor
        push ds             ;salvarea starii curente
        push es
        push di
        push ax
        push bx
```

```
        push dx
        mov ax,cs
        mov ds,ax           ;initializarea segmentului de date
reia:   mov ax,intid
        mov bl,txint        ;identificarea sursei întreruperii (emisie sau receptie)
        call gets
        jz recep

;întrerupere la emisie

        mov ax,txdat
        mov bl,txb
        call putb           ;emite octetul din txb
        inc txb             ;trece la codul urmator

        mov ax,lbe          ;secventa de monitorizare pe display
        les di,spote
        call flash
        mov word ptr spote,di     ;actualizarea pozitiei pe display pentru emisie
        jmp alte            ;daca între timp s-a terminat o receptie

;întrerupere la receptie

recep:  mov ax,rxdat
        call getb           ;citeste octetul receptionat
        mov ah,rxb          ;octetul referinta
        inc rxb             ;trece la codul urmator
        cmp ah,al           ;compara receptia cu referinta
        nop                 ;eroarea nu este tratata în acest exemplu
```

```
        mov ax,lbr          ;secventa de monitorizare pe display
        les di,spotr
        call flash
        mov word ptr spotr,di       ;actualizarea pozitiei pe display pentru receptie

alte:   mov ax,intid
        mov bl,1            ;se verifica daca mai sunt solicitari active (când emisia si
                            ;receptia apar simultan, sau în timpul tratarii uneia survine
                            cealata).
        call gets
        jz reia             ;se reia secventa de identificare si tratare

        mov al,20h          ;EOI pentr 8259A
        out 20h,al
        pop dx              ;restaurari registre
        pop bx
        pop ax
        pop di
        pop es
        pop ds
        iret                ;revenirea în programul întrerupt

;subrutine si rezervari de memorie

gets: mov dx,comx           ;testeaza starea bitilor mascati în bl din registrul indicat de
                            ;ax
        add dx,ax
        in al,dx
        test al,bl
```

```
        retn

putb:   mov dx, comx        ;trimite octetul din bl în registrul indicat de ax
        add dx,ax
        mov al,bl
        out dx,al
        retn

getb:   mov dx,comx         ;aduce în al octetul din registrul indicat de ax
        add dx,ax
        in al,dx
        retn

flash:  mov byte ptr es:[di],20h
        cmp di,ax
        jne et1
        sub di,40h
et1:    inc di
        inc di
        mov byte ptr es:[di],0dbh
        retn
exmp endp

;rezervari si definitii

comx        dw    0       ;adresa  interfetei COM1
txb         db    0       ;coduri emise
rxb         db    0       ;coduri receptionate
spote       dd    0b0000190h
```

```
spotr       dd    0b00002d0h
rezrm       equ   $
cseg ends
end   exmp
```

O problema obisnuita existenta în transmisiile seriale este aceea a protocolului de dialog între emitator si receptor. La emitator/receptor exista în mod obisnuit câte un buffer de dimensiune finita pentru datele ce urmeaza a fi emise/receptionate. Când acest buffer este plin calculatorul ignora datele noi pâna ce bufferul se goleste suficient. Pentru evitarea fenomenului mai sus amintit se implementeaza protocoale hard sau soft, prin intermediul carora emitatorul si receptorul îsi semnaleaza unul altuia umplerea bufferului propriu. Protocolul XON/XOFF este un protocol soft care functioneaza pe urmatorul principiu: când bufferul de receptie este plin, receptorul trimite un caracter XOFF (în zecimal 19) pentru a comunica emitatorului sa opreasca emisia datelor. Când bufferul de receptie devine suficient de gol, receptorul trimite un caracter XON (în zecimal 17) pentru a indica faptul ca transmisia datelor poate reîncepe. Când este folosit protocolul XON/XOFF codurile XON si XOFF vor fi folosite întotdeauna ca si coduri de control si nu ca date, deci acest protocol nu este recomandat în cazul transmisiilor binare.

Exemplu 3: Program de folosire a interfetei seriale prin întreruperi cu implementarea protocolului XON/XOFF.

```
        .model small
COM1  equ  03F8h
        .code
```

```
start:
jmp begin

caracter        db      30h

handleCOM1      PROC FAR  ;rutina de tratare int 0Ch
        push ax                 ;salvarea registrilor folositi în rutina
        push dx
        push ds
        push bx
        mov ax,COM1+2
        in al,dx
        and al,00000111b
        cmp al,03h              ;este intrerupere la receptie?
        jnz alfa                ;nu - salt la alfa
        mov dx,COM1
        in al,dx
        cmp al,13h              ;caracterul XOFF?
        jz beta
        mov dx,COM1+1           ;tratarea  caracterului XON
        mov al,03h
        out dx,al               ;validare întreruperi emisie/receptie
        jmp alfa

beta:   mov dx,COM1+1           ;tratarea caracterului XOFF
        mov al,01h              ;invalidare întrerupere emisie
        out dx,al
        jmp exit1
```

```
alfa:   mov ax,0B000h          ;adresa de inceput a memoriei video
        mov ds,ax
        mov dl,cs:caracter     ;încarcarea caracterului curent
        mov bl,dl
        xor bl,3Ah             ;trece de ‘9’?
        jn et1
        mov dl,30h             ;caracterul ‘0’
        mov cs:caracter,dl
et1:    mov ds:[20],dl         ;scriere caracter curent
        inc cs:caracter
        mov dx,COM1            ;emitere caracter ‘A’
        mov al,'A'
        out dx,al

exit1:  mov al,20h             ;EOI pentru 8259A
        out 20h,al
        pop bx                 ;refacere stare registri
        pop ds
        pop dx
        pop ax
        iret
handleCOM1      ENDP

sfirsit         db 0

;rutina de initializare port serial
initCOM1    PROC NEAR
```

```
        mov ah,00h              ;setarea parametrilor de comunicatie folosind serviciul
BIOS
        mov al,67h              ;600 baud, 8 biti, 2 biti stop
        mov dx,0                ;fara paritate
        int 14h
        mov ax,250Ch
        mov dx,offset handleCOM1
        int 21h

        mov dx,COM1+4           ;activare DTR,RTS
        mov al,0Bh
        out dx,al
        mov dx,COM1+1
        mov al,3
        out dx,al               ;validare întreruperi emisie/receptie

        in al,21h               ;activare IRQ4 în 8259A
        and al,11101111b
        out 21,al
        ret
initCOM1    ENDP

begin:
        mov ax,@CODE
        mov ds,ax
        call initCOM1
        mov dx,offset sfirsit
        mov cl,4                ;calcularea lungimii codului rezident în paragrafe
```

```
        shr dx,cl
        inc dx                  ;rotunjire superioara
        mov ax,3100h
        int 21h                 ;ramânere rezidenta
END start
```

3. Probleme propuse.

1. Sa se modifice exemplul 2 de folosire a interfetei seriale prin întreruperi pentru emiterea seriala a unor caractere introduse de la tastatura si afisarea lor în rutina de tratare a întreruperii de receptie. În acest scop se va folosi o cupla cu bucla locala.

2. Sa se modifice exemplul 1 pentru citirea si interpretarea erorilor ce pot aparea la transmisia seriala.

3. Sa se testeze exemplul 3 de folosire a protocolului XON/XOFF pentru comunicarea cu o consola programabila conectata serial la PC.

UTILIZAREA COMUNICATIEI PARALELE ÎN IBM – PC

1. Memento teoretic.

În PC comunicatia paralela se poate face prin intermediul a trei porturi de comunicatie, denumite LPT1, LPT2 si LPT3. Zona de date BIOS contine, începând de la adresa 0:0408h o lista a adreselor de baza corespunzatoare cuploarelor paralele. Adresele pentru LPT1 si LPT2 sunt:

LPT1 - 378h..37Fh
LPT2 - 278h..27Fh

Asupra acestor porturi se poate opera direct, prin instructiuni de intrare/iesire (IN, OUT), sau prin intermediul întreruperii BIOS 17h.

Utilizarea comunicatiei paralele se poate face prin interogare (*polling*), sau prin întreruperi. Liniile de întrerupere folosite, corespunzatoare controlorului de întreruperi 8259A, respectiv tipurile corespunzatoare ale întreruperilor în PC, sunt:

LPT1 - IRQ7 - INT 0Fh
LPT2 - IRQ5 - INT 0Dh

Registri asociati unui cuplor de comunicatie paralela (adresele fiind date pentru LPT1), sunt:

Adresa

378h Registrul de date al cuplorului paralel.

Scriere: contine caracterul ASCII trimis imprimantei.

Citire: Contine ultimul caracter trimis.

37Ah *Citire/scriere*: Registrul de control al imprimantei.

Bit: 0 - strob, este 1 la transmiterea unui octet;

1 - AUTO LineFeed, 1 cauzeaza trimiterea automata a unui LF dupa un CR;

2 - INIT, initializare, 0 reseteaza imprimanta;

3 - SLCT IN, selectare, 1 selecteaza imprimanta;

4 - IRQ Enable, validare întreruperi, 1 - permite generarea unei întreruperi *hardware* când ACK este 0;

5, 6, 7 - sunt 0.

379h *Citire*: Registrul de stare al imprimantei.

Bit: 0,1,2 - sunt 0

3 - ERROR, cand e 0 imprimanta semnalizeaza o eroare;

4 - SLCT, 1: imprimanta e selectata;

5 - PE, 1: lipsa hârtie în imprimanta;

6 - ACK, 0: este permisa trimiterea urmatorului caracter;

7 - BUSY, 0: *buffer*-ul imprimantei este plin, imprimanta este decuplata, sau a aparut o alta eroare;

Întreruperea BIOS 17h pune la dispozitie urmatoarele servicii pentru interfata paralela:

AH=00h - Imprimarea unui caracter.

La apel:

AL - codul ASCII al caracterului;

DX - numarul imprimantei (0, 1 sau 2);

La revenire:

AH - starea imprimantei, semnificatiile bitilor fiind date la functia 02h;

AH=01h - Initializarea portului paralel.

La apel:

DX - numarul imprimantei;

La revenire:

AH - starea imprimantei, semnificatiile bitilor fiind date la functia 02h;

AH=02h - Citirea starii imprimantei.

La apel:

DX - numarul imprimantei.

La revenire:

AH - starea imprimantei (returnata si de functiile 0 si 1)

bit 0 - 1: *timeout*;

bit 1, 2 - nefolositi;

bit 3 - 1: eroare I/O;

bit 4 - 0: imprimanta decuplata, 1: imprimanta selectata;

bit 5 - 1: lipsa hârtie în imprimanta;

bit 6 - 1: ACK;

bit 7 - 1: imprimanta *ready*.

2. Programe demonstrative.

Exemplul 1: Imprimarea unui sir de caractere prin interogare (*polling*).

```
        .model small

        buff    equ     278h            ;registrul date LPT2
        stat    equ     279h            ;registrul stare LPT2
        cont    equ     27Ah            ;registrul control LPT2
        ack     equ     01000000b       ;bitul ACK

        .data

        text    db      'Test imprimanta',13,10
        lung    equ     $-text          ;lungimea textului

        .code
start:  mov ax,@data                    ;încarcarea registrului ds
        mov ds,ax                       ;cu baza segmentului de date
        mov dx,cont                     ;registrul de control
        mov al,08h                      ;selectare imprimanta
        out dx,al
        mov si,offset text
        mov cx,lung                     ;lungimea textului
et1:    lodsb                           ;al <- ds:[si], si++
        mov dx,buff                     ;registrul de date
        out dx,al                       ;imprimare caracter
        mov dx,cont                     ;registrul de control
        mov al,0Fh                      ;strob = 1
        out dx,al
        mov al,0Eh                      ;strob = 0
        out dx,al
```

```
        mov dx,stat                 ;registrul de stare
et2:    in al,dx                    ;citire stare
        test al,ack                 ;gata pentru urmatorul caracter?
        jnz et2
        loop et1
        mov ax,4C00h                ;iesire în DOS
        int 21h
end start
```

Exemplul 2: Imprimare prin folosirea întreruperilor.

```
        .model small

        .stack 100h

        .data

        int0foff    dw 0            ;deplasamentul "vechii" rutine de tratare
        int0fseg    dw 0            ;segmentul "vechii" rutine de tratare
        mes         db 13,10,'Pentru terminare apasati o tasta!$'
        oldmask     db 0            ;masca întreruperilor în controlorul 8259A

        .code

        contor      db 10           ;contorul caracterelor imprimate pe o linie

;Rutina de întîrziere
del:    push cx
        mov cx,0FFh
e:      dec cx
        or cx,cx
```

```
        jnz e
        pop cx
        ret

;Rutina de imprimare a unui caracter
printch:
        push ax                 ;salvarea registrilor folositi în rutina
        push dx
        mov dx,378h
        out dx,al               ;imprimarea caracterului
        mov al,0FFh             ;strob on
        mov dx,37Ah
        out dx,al
        mov al,0FEh             ;strob off
        out dx,al
        pop dx                  ;refacerea registrilor la valorile anterioare
        pop ax
        ret

;Rutina de tratare a întreruperii LPT1 (int 0Fh)
new_int0f:
        push ax                 ;salvarea registrilor folositi în rutina
        push dx
        mov al,'A'              ;imprimarea a 10 caractere ‘A’ pe o linie
        dec cs:contor
        jnz et1
        mov al,10               ;daca s-a ajuns la al 10-lea
        mov cs:contor,al        ;initializare contor
```

```
        mov al,13                   ;imprimare CR
        call printch
        mov al,10                   ;imprimare LF
et1:    call printch
        mov al,20h                  ;EOI pentru 8259A
        out 20h,al
        pop dx                      ;refacerea registrilor implicati în rutina
        pop ax
        iret

;Programul principal
start:  mov ax,@data                ;initializarea registrului DS
        mov ds,ax                   ;cu baza segmentului de date

        mov ax,350Fh                ;citirea vectorului întreruperii 0Fh
        int 21h
        mov int0foff,bx             ;salvarea offset-ului
        mov int0fseg,es             ;salvarea segmentului

        mov ax,250Fh                ;capturarea întreruperii 0Fh
        push ds                     ;salvarea ds
        push cs
        pop ds                      ;ds=cs
        mov dx,offset new_int0f     ;offset-ul rutinei proprii
        int 21h
        pop ds                      ;refacerea registrului ds

        in al,21h                   ;citirea mastii din registrul 8259A
        mov oldmask,al              ;salvarea mastii
```

```
and al,7Fh                ;demascarea întreruperii LPT1 (0Fh)
out 21h,al                ;setarea noii masti

mov dx,37Ah               ;scrierea registrului de control cu INIT=0
mov al,1Ah
out dx,al
call del                  ;asteptare
mov al,1Eh                ;scrierea registrului de control cu INIT=1
out dx,al

mov ah,09h                ;afisare mesaj
mov dx,offset mesaj
int 21h

mov ah,1                  ;asteptare o tasta
int 21h

mov al,oldmask
out 21h,al                ;refacerea vechii masti

mov dx,37Ah
mov al,0Ah                ;invalidare întrerupere LPT1+ INIT=0
out dx,al
call del                  ;asteptare
mov al,0Eh                ;INIT=1
out dx,al

mov ax,250Fh              ;refacerea vechiului vector de întrerupere
mov dx,int0fseg
mov ds,dx
```

```
        mov dx,int0foff
        int 21h

        mov ax,4C00h                    ;revenirea în DOS
        int 21h
end start
```

3. Probleme propuse.

1. Sa se încerce programele de mai sus, adaugându-se tratarea erorilor ce pot aparea (imprimanta decuplata, lipsa hârtie, etc.).

2. Sa se scrie un program de testare a registrilor cuploarelor LPT1 si LPT2, prin scrierea a câte unui bit pe 0, respectiv 1, urmata de verificarea bitului respectiv prin citire si comparare. În cazul aparitiei unei erori sa se afiseze un mesaj corespunzator.

3. Sa se scrie un program rezident care indirecteaza întreruperea serviciilor pentru imprimanta (int 17h), cu scopul salvarii într-un fisier a tuturor datelor trimise imprimantei.

DISPOZITIVE CU ACCES LA MEMORIE

1. Memento teoretic

Deoarece microprocesorul ar consuma foarte mult timp pentru transferurile de date dintre periferice (datele stocate pe hard disc, flopy disc, într-un proces de achizitie de date etc.) s-a recurs la folosirea de circuite specializate care sa preia acesta sarcina de transfer intre periferic si memorie, cunoscând adresa sursa, adresa destinatie si numarul de octeti de transferat.

2. Controler-ul DMA în arhitectura IBM – PC

Firma IBM a folosit controler-ul DMA 8237A fabricat de firma Intel, controler care se foloseste si în ziua de astazi, în structura PC. Se va studia acest circuit în continuare.

Urmasii lui 8237A sunt 8237A-4 si 8237A-5 care este cel mai recent contoler si totodata cel care respecta compatibilitatea cu cele mai vechi. Acestea au urmatoarele caracteristici:

- Posibilitate de E/D pentru fiecare canal a cereri de DMA (DMA Request)
- Patru canale independente / chip
- Posibilitatea de transfer memorie - memorie
- Posibilitate de incrementare sau decrementare a adresei sursa - destinatie
- Posibilitate de transfer cu o viteza de maxim 1.6 MB/s cu 5 MHz 8237A-5
- Posibilitate de de expandare directa la orice numar de canale
- Posibilitate de de cerere soft pentru DMA Request
- Posibilitate de de control independent pentru DREQ sI DACK
- Poate accesa maxim un segment de memorie (64 Ko).

Schema bloc a controlerului DMA 8237A-5 este prezentata în anexa 1.

Descrierea pinilor :

RESET: intrare activa în "1" logic si care reseteaza pe 0 registrii de comanda, stare, cereri, temporar si masca. Dupa RESET circuitul este inactiv un ciclu. Tip: I.

CLK : linie de ceas cu frecventa de 5MHz pentru 8237A-5 (3MHz pentru 8237A).

CS : (Chip Select) este activ in "0" logic si este utilizat pentru selectia circuitului în mod I/O, permitând programarea circuitului în timpul ciclurilor inactive. Tip: I.

REDY : este utilizata pentru sincronizarea circuitului cu memoriile si cu dispozitivele periferice lente. Tip: I.

AEN : (Adress Enable) iesire folosita pentru selectarea circuitului latch. Cind AEN=1 bitii A_8 - A_{15} sunt trecuti pe magistrala de adrese. Se poate folosi pentru dezactivarea altor periferice ce utilizeaza magistrala in mod DMA. Este activ in "1" logic. Tip: O.

EOP : (End Of Process) este un semnal bidirectional care indica terminarea unui serviciu DMA. Acest semnal se poate activa intern sau extern. Totodata este în strânsa legatura cu TC (Terminal Count) care indica ca un canal a terminat transferul. Tip: I/O.

ADSTB : (Adress Strobe) este activ în "1" logic si este utilizat pentru a selecta bitii A_8 - A_{15} într-un latch extern. Tip: O.

MEMR : (Memory Read) iesire TS activa în "0" logic folosita pentru a citi din memorie de la adresa selectata într-un ciclu de DMA Read sau transfer memorie - memorie. Tip: O.

MEMW : (Memory Write) iesire TS activa în "0" logic folosita pentru a scrie în memorie de la adresa selectata într-un ciclu de DMA Write sau transfer memorie - memorie. Tip: O.

IOR : (I/O Read) linie bidirectionala TS prin intermediul careia CPU poate citi registrii de control în timpul ciclurilor inactive. În timpul ciclurilor active este folosita de DMA pentru a accesa date de la periferice care utilizeaza DMA Write transfer. Tip: I/O.

DRQ0-DRQ3 : (DMA Request) intrari pe care perifericele transmit asincron semnale cu semnificatia ca este necesar un transfer DMA pentru canalul specificat. Aceste cereri pot fi servite în mod cu prioritati fixe unde DRQ0 este cea mai prioritara sau cu prioritati ciclice. Linia DRQ corespunzatoare cereri trebuie mentinuta în stare activa pâna când linia DACK corespunzatoare devine activa (pâna când cererea devine recunoscuta de CPU). Tip: I.

DACK0-DACK3: (DMA Acknowledge) iesire utilizata pentru a semnaliza perifericului ce a cerut transfer DMA ca i s-a alocat un ciclu DMA. Nivelul activ poate fi programat. Dupa RESET toate liniile DACK sunt active în "0" logic. Tip: O.

HRQ : (Hold Request) linie utilizata de 8237A pentu a cere CPU controlul asupra magistralelor. Acest semnal se poate valida sau nu printr-o masca. Tip: O.

HLDA: (Hold Acknowledge) linie de raspuns ce vine de la CPU ca magistralele au fost eliberate in vederea unui transfer DMA. Tip: I.

A_0 ... A_3: linii de adrese bidirectionale TS. Ele sunt iesiri în ciclul activ si contin cei mai semnificativi patru biti ai adresei. În ciclul pasiv sunt folosite de CPU pentru a adresa registrii interni. Tip: O.

A_4 ... A_7 : linii de adresa care sunt valabile doar în timpul transferului DMA si furnizeaza bitii A_4 ... A_7 ai adresei. Tip: O.

D_0 ... D_7 : linii bidirectionale TS conectate la magistrala de date. În "starea program" pe aceste linii se citesc sau se scriu registrii de adresa, stare, numaratori de catre CPU. În ciclul inactiv aceste linii au semnificatia de biti de adresa A_8 ... A_{15}. Tip: I/O.

Descrierea functionala

În schema bloc a circuitului se observa urmatoarele module :

I) Blocul de control al comenzilor : decodificarea diferitelor comenzi transmise circuitului de catre CPU

II) Blocul de determinare a prioritatiilor : rezolva problema ordinii în care se achita cererile de transfer DMA pentru mai multe canale care o cer simultan

III) Canalele de transfer DMA : dispune de 4 canale independente programabile. În cadrul fiecarui canal exista cinci registrii (adresa de baza, numarator cuvinte, adresa curenta, numarator cuvinte curente, registru de mod).

Se poate observa ca circuitul 8237A dispune de un spatiu de memorie interna de 344 biti.

Vom prezenta, în continuare, registrii interni ai circuitului 8237A

Nume registru	Lungime	Numar
Base Adress Registers	16 biti	4
Base Word Count Registers	16 biti	4
Curent Address Registers	16 biti	4
Curent Word Count Registers	16 biti	4
Temporary Address Register	16 biti	1
Temporary Word Count Register	16 biti	1
Status Register	8 biti	1
Command Register	8 biti	1
Temporary Register	8 biti	1
Mode Register	6 biti	4
Mask Register	8 biti	1
Request Register	8 biti	1

Functionarea DMA

Circuitul 8237A functioneaza în doua moduri de ciclii. Primul dinte ei este ciclul inactiv când DMA-ul nu executa nici o cerere DMA si al doilea mod cel în care DMA (cel putin un canal) executa o cerere si se numeste ciclu activ. În ciclu pasiv CPU poate inspecta si modifica valorile din registrii interni ai DMA-ului.

Ciclul pasiv

În acest ciclu controler-ul verifica daca nu apare o cerere de întrerupere. Daca linia CS si HLDA sunt în "0" logic atunci cotrolerul intra în starea de Progam Condition în care asteapta ca CPU sa inspecteze, citeasca sau sa scrie registrii interni. Registrul intern flip-flop este utilizat pentru a determina care este octetul

“low” si care este cel “hi” când se programeaza adresa si numarul de octeti care sunt reprezentati pe 16 biti. Bitul flip-flop este resetat de catre comenzile Reset si “ master clear”.

Ciclul activ

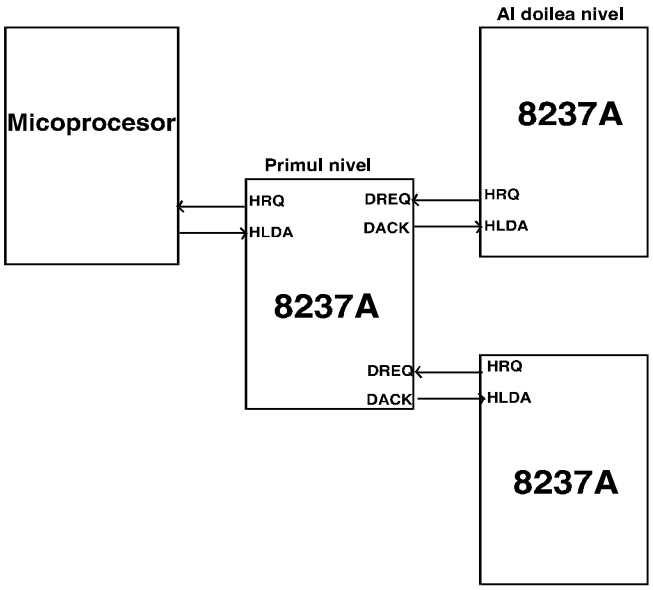
Când circuitul 8237A este în ciclul pasiv si pe una din liniile de cerere DMA nemascata vine o cerere atunci va activa linia HRQ pentru CPU si va intra în ciclul activ. În acest ciclu poate intra în unul din cele patru moduri :

Single Transfer Mode: în acest mod este programat sa faca doar un singur transfer. Numaratorul (Word count) va fi decrementat sau incrementat depinde cum a fost programat iar când ajunge la zero sau la FFFFH, un TC va fi transmis pentru canalul respectiv si va cauza procesul de autointializare daca canalul a fost programat pentru asa ceva.

Block Transfer Mode : în acest mod circuitul DMA este trebuie activat de DREQ dupa ce a fost un TC sau de un EOP extern. Din nou procesul de initializare va fi apare daca a fost programat.

Demand Transfer Mode: este programat sa continue sa transfere dupa ce un semnal TC a aparut sau un EOP extern, sau dupa ce DREQ a devenit inactiv.

Cascade Mode : În acest mod se pot conecta în cascada mai multe circuite 8237A. Acestea se pot conecta cum este aratat în figura.



Tipuri de transfer

Exista trei moduri de transfer care la rândul lor pot activa în trei moduri distincte: de citire, scriere si verificare. Transferul cu scriere muta date de la un port de I/O în memorie activând liniile MEMW si IOR. Transferul cu citire muta date din memorie la un port de I/O activând liniile MEMR sI IOW. Transferul cu verificare este un proces de pseudo transfer.

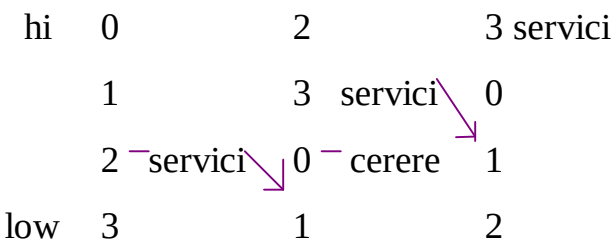
Memorie-Memorie - pentru a muta un bloc de date de la o adresa la o alta adresa cu un minim de efort si timp se poate utiliza facilitatea circuitului 8237A de a transfera blocuri de date memorie în memorie. Pentru acest lucru este necesar setarea unui bit din registrul de comanda pentru a selecta canalul 0 si 1 de a lucra in mod de transfer memorie în memorie. Acest proces este initializat de catre o cerere DERQ soft trimisa pentru canalul 0. În registrul care contine adresa curenta pentru canalul 0 este sursa care este incrementata sau decrementata depinde de cum a fost programata. Octetii care trebuie transferati se vor citi din memorie în registrul temporar din 8237A dupa care se va activa un ciclu de scriere în memorie pe canalul

1. Când în registrul “Word count” se va afla FFFFH se va genera un TC care va cauza un EOP.

Canalul 0 poate fi programat astfel încât sa retina aceeasi valoare pentru a putea umple o zona de memorie cu aceasi valoare.

Autoinitializare - Programând un bit în registrul de mod se va activa acesta optiune. În acest mod se va restaura automat adresa curenta (Curent Adress) si numarul de octeti de transferat (Curent Word Count) când un EOP este activ. Astfel DMA-ul este gata sa înceapa un nou transfer când un semnal DERQ este detectat.

Prioritati - 8237A suporta doua tipuri de prioritati selectabile soft. Prima dinte ele este prioritatea fixa care fixeaza fiecarui canal o prioritate de la 0 (cea mai prioritara) la 3(cea mai putin prioritara). Al doilea tip de prioritati este cel cu prioritati rotative. Aceasta presupune urmatorul algoritm: ultimul canal servit va primii cea mai mica prioritate. În acest fel se va putea evita monopolizarea serviciului DMA de catre un singur canal.



Compress Timing - în traducere ar însemna ca 8237A poate suprima un tact în timpul de transfer al datelor.

Descrierea Registrilor

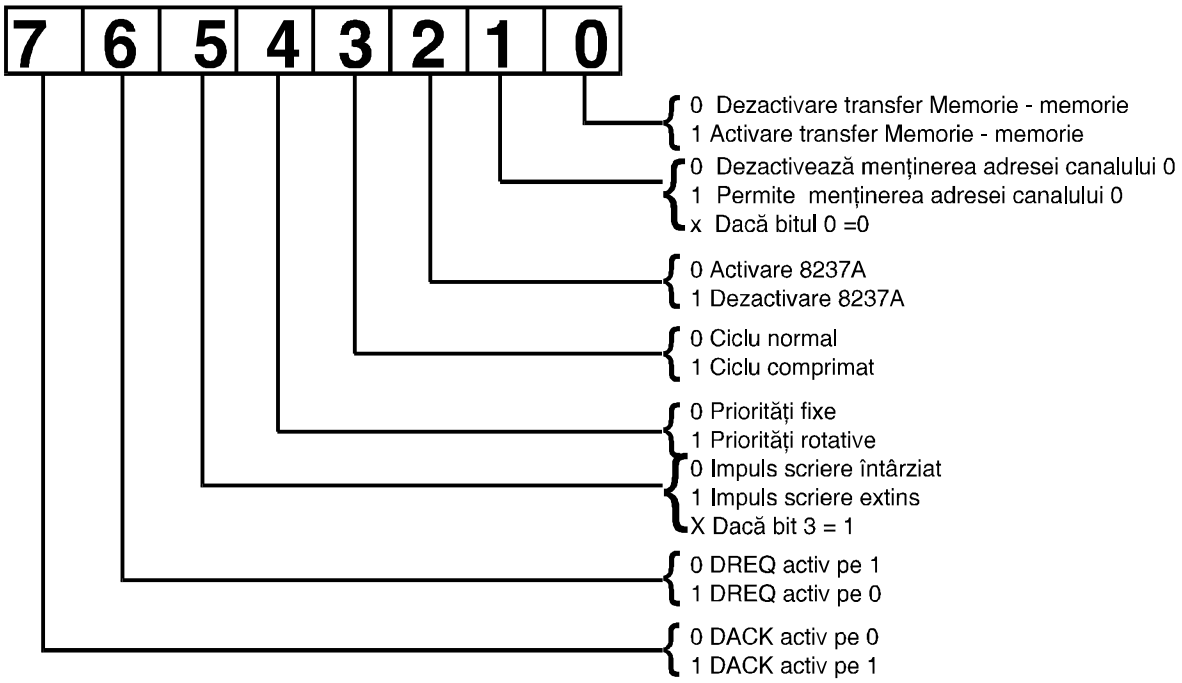
Registrul cu adresa curenta (Curent Adress Register): este un registru pe 16 biti care contine adresa de unde se vor transfera date. Aceasta adresa se va incrementa sau decremnta automat dupa fiecare transfer. Daca canalul este

programat cu autoinitializare adresa curenta va fi recopiata în registru doar dupa un EOP.

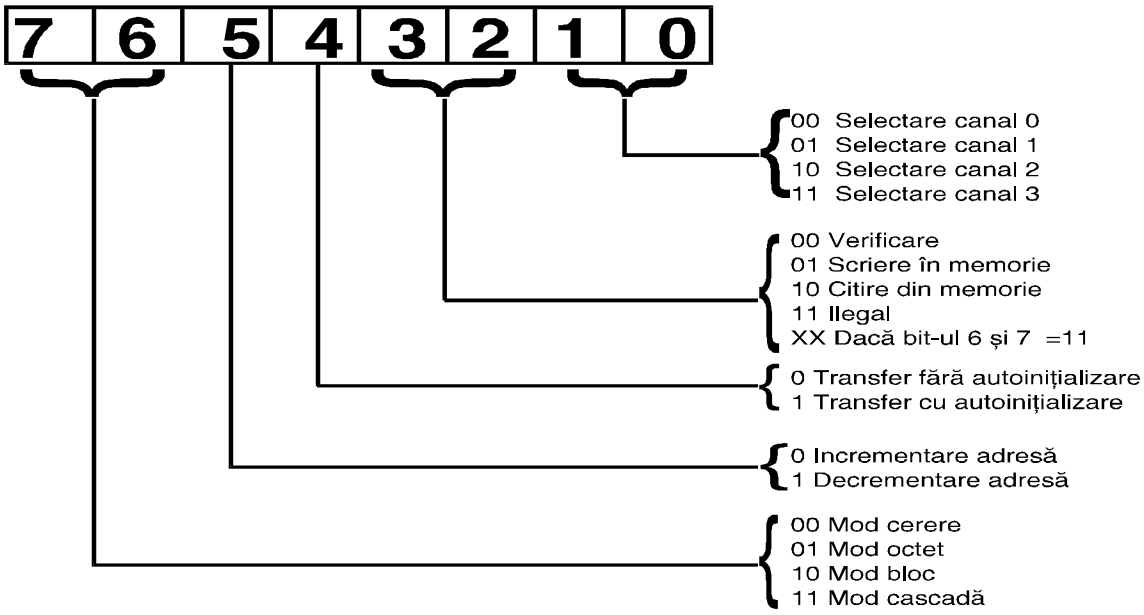
Registru numarator de cuvinte (Curent Word Register): este un registru pe 16 biti care stabileste câte transferuri DMA au mai ramas de efectuat. **Atentie**: numarul de transferuri efectuate va fi cu unu mai mult decât numarul înscris (Ex. daca se înscrie 100 se vor transfera 101 octeti) La trecerea din 0 în FFFFh se va genera TC. Daca canalul a fost programat cu autoinitializare valoarea originala va fi inscrisa în momentul aparitiei unui EOP (se initiaza procesul de autoinitializare)

Registrul cu adresa de baza si cu numarul de octeti de transferat (Base Adress and Base Word Count Registers): fiecare canal are asociat acest registru care este utilizat intern pentru procesul de autoinitializare de unde se vor copia informatiile originale. Acest registru se înscrie simultan cu scrierea primilor doi registrii descrisi mai sus, si nu poate fi citit de catre microprocesor.

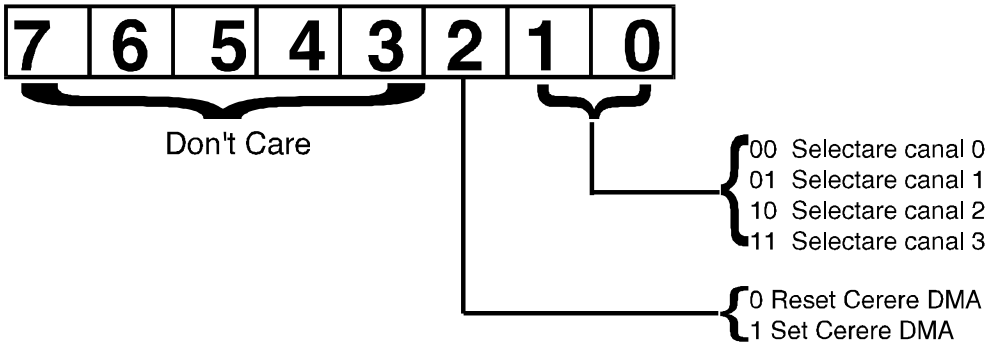
Registru de comanda (Command Register): Este un registru pe 8 biti care contine informatii referitoare la operatiile efectuate de 8237A. Este perogramat de CPU în timpul starii de Program Condition si este sters de RESET sau MASTER Clear. În tabelul alaturat se specifica functiile bitilor din acest octet.



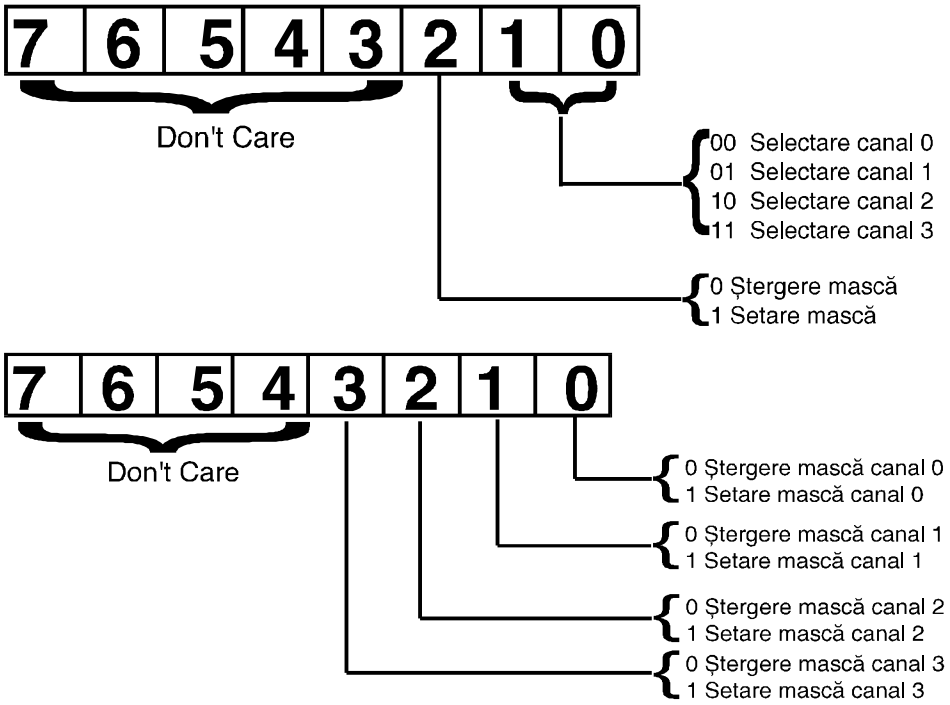
Registrul de mod (Mode Register): fiecarui canal îi corespunde un registru de 6 biti în care se specifica felul transferului. A se vede figura alaturata.



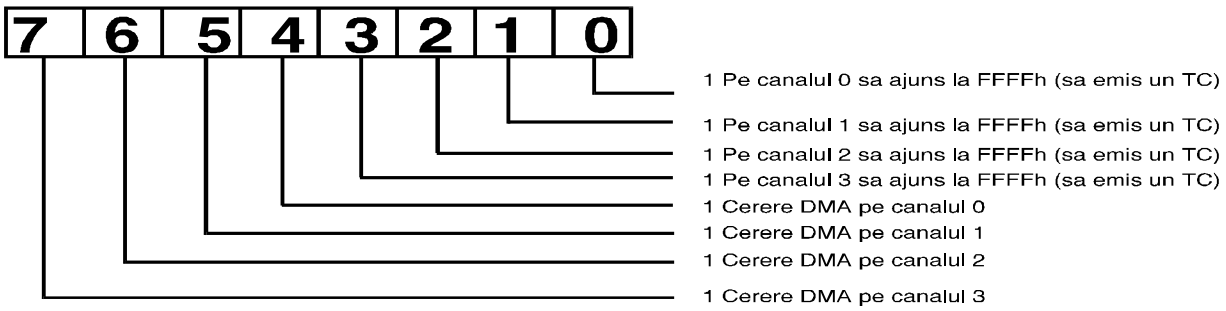
Registrul de cerere servici DMA (Request Register): 8237A poate raspunde unei cereri de transfer DMA care este initiata prin soft. Fiecare canal are asignata o pozitie (un bit) în registrul de cerere care arata daca se cere servici DMA pentru canalul respectiv sau nu. Acest registru nu se poate masca si respecta regula stabilita de logica de prioritati. Pentru a se vedea cum se poate seta - reseta bitul asociat uniu canal a se vedea figura alaturata. Pentru a putea un canal sa execute o cerere DMA soft acest canal trebuie sa fie programat in **mod bloc** (**Block Mode**)



Registrul masca (Mask Register): fiecarui canal îi este atribuit un bit cu ajutorul caruia se specifica daca acest canal este inactiv (disable) sau activ în momentul activarii liniei DREQ. Se pot stabili mastile pentru fiecare canal separat sau pentru mai multe canale în acelasi timp. A se vedea figura.



Registrul de stare (Status Register): este un registru care poate fi citit de CPU si care furnizeaza informatii despre fiecare canal în parte în acel moment. Aceste informatii ne spun daca, canalul respectiv este solicitat sau daca s-a emis un TC.



Registru temporar (Temporary Register): este un registru folosit pentru a memora octetul curent în procesul de transfer memorie - memorie. Dupa un transfer complet CPU poate citi ultimul octet transferat. Acest registru este sters dupa un RESET.

Comenzi soft (Software Commands): Mai exista câteva comenzi soft care pot fi executate în starea de Program Condition si care nu se reflecta direct asupra unui bit anume. Aceste comenzi sunt:

Clear First/Last Flip Flop: Aceasta comanda este necesara în momentul în care se doreste scrierea unui registru de 16 biti. Dupa executarea acestei comenzi, 8237A respecta urmatoarea regula: primul octet înscris va fi octetul "low" urmatorul fiind octetul "hi" s.a.m.d.

Master Clear: Aceasta comanda are acelasi efect cu un reset hard. Dupa executarea acestei comenzi registrii de comanda, stare, cerere, temporar, si Flip/Flop vor fi resetati si registrul masca va fi setat, dupa care 8237A va intra în *ciclul pasiv*.

Clear Mask Register: Acesta comanda sterge toate mastile de la cele *patru* canale si din acel moment vor putea accepta cererile de transfer.

Programarea

Vom prezenta în continuare modul de programare a lui 8237A doar pe 8 biti; pe 16 biti fiind absolut identic, doar porturile fizice asociate canalelor DMA fiind diferite (vezi anexa). În tabelul urmator se vor indica porturile prin intermediul caruia se poate programa 8237A.

Port	Sens	Actiune
00h	R/W	Adresa canal 0 DMA

01h	R/W	Contor canal 0 DMA
02h	R/W	Adresa canal 1 DMA
03h	R/W	Contor canal 1 DMA
04h	R/W	Adresa canal 2 DMA
05h	R/W	Contor canal 2 DMA
06h	R/W	Adresa canal 3 DMA
07h	R/W	Contor canal 3 DMA
08h	R	Registru de stare
08h	W	Registru de comanda
09h	W	Registru cereri soft
0Ah	W	Registru masti (invidual)
0Bh	W	Registru mod
0Ch	W	Stergere Flip-Flop intern
0Dh	R	Registru temporar
0Eh	W	Reset 8237A
0Fh	W	Registru masti (toate canalele)

Pentru a putea accesa un spatiu de adrese de 1MB sunt necesare 20 linii de adresa, dar fiindca 8237A nu dispune doar de 16 linii de adresa s-a gasit solutia urmatoare. Din adresa pe 20 de biti se separa primii 4 care se vor înscrie în asa numitul registru pagina (registru intern). O pagina este un bloc de memorie de 64 Ko (analog cu adresa de segment). Astfel s-a împartit primul megaoctet de memorie dupa cum se poate vedea în urmatorul tabel:

Pagina	Segment : Ofset
0	0000:0000 – 0000:FFFF
1	1000:0000 – 1000:FFFF
2	2000:0000 – 2000:FFFF
3	3000:0000 – 3000:FFFF
4	4000:0000 – 4000:FFFF
5	5000:0000 – 5000:FFFF
6	6000:0000 – 6000:FFFF
7	7000:0000 – 7000:FFFF
8	8000:0000 – 8000:FFFF
9	9000:0000 – 9000:FFFF
A	A000:0000 - A000:FFFF
B	B000:0000 - B000:FFFF
C	C000:0000 - C000:FFFF
D	D000:0000 - D000:FFFF
E	E000:0000 - E000:FFFF
F	F000:0000 - F000:FFFF

Porturile de pagina asociate canalelor DMA sunt:

Canal DMA	Port asociat
0	87h
1	83h
2	81h
3	82h
4	8Fh

5	8Bh
6	89H
7	8Ah

Ei bine daca cunoastem pagina si segmentul de unde dorim sa transferam date, numarul de octeti de transferat putem trece la programarea propriu-zisa. Pasii necesari programarii cu succes sunt urmatorii :

1. Dezactivare canal
2. Reset octet F/F
3. Set mode
4. Set page
5. Set offset
6. Set lungime bloc
7. Enable channel
8. Programarea perifericului de a permite transfer DMA

În continuare se vor prezenta doua exemple de programare a lui 8237A. Primul arata cum s-ar putea testa daca un canal DMA functioneaza corect sau nu prin soft. Cel de-al doile exemplu arata cum se poate initia un transfer DMA memorie - memorie.

Exemplul 1 :

```
Prog        segment at 0FFF0h
            assume cs:prog

Start       Label       FAR
```

```
DMA     EQU             0
        Mov al,0
        Out 83h,al      ;Initializare registru pagina
        Mov al,4
        Out DMA+8,al    ;Dezactivare DMA
        Out DMA+0Dh,al  ;Reset DMA
        Mov al,0FFh
R1:     Mov bl,al
        Mov bh,al
        Mov cx,8
        Xor dx,dx
R2:     Out dx,al       ;Scrie octet Low
        nop
        Out dx,al       ;Scrie octet High
        Mov ax,0101h
        In al,dx        ;citire High
        Mov ah,al
        In al,dx        ;Citire Low
        Cmp bx,ax       ;Verificare înscriere
        Jz C1
        Jmp Eroare
C1:     Inc dx          ;urmatorul registru DMA
        Loop R2
        Inc al          ;Configuartie de test 0000h
        Jz R1           ;Test O.K.
R3:     In al,0A0h      ;Seexecuta o citire de la portul 0A0h
        Mov cx,10h
```

```
C2:         Loop c2          ;Bucla de întârziere
            Jmp R3           ;Bucla în caz de eroare
Eroare:     In al,0a0h       ;se executa doua citiri de la portul 0a0h
            nop
            In al,0a0h
            Mov cx,10h
C3:         Loop C3          ;Bucla de întârziere
            Jmp Eroare

            Org 0F0h
            Jmp Start
Prog        ends
end start
```

Exemplul 2 **:**

```
            .model small

            .data

mesage      db      10,13,'11111111111111111111111111','$'

clear_fl    equ     0ch
 mask01     equ     0ah
 un_mask    equ     0ah
 ch0        equ     00h
 ch1        equ     02h
```

```
ch1_cnt     equ     03h
ch0_cnt     equ     01h
page0       equ     087h
page1       equ     083h
mode        equ     0bh
cmmd        equ     08h
dma_request equ     09h

page_0      db      '0'
page_1      db      '0'
off0        dw      '0'
off1        dw      '0'

message1    db      10,13,'22222222222222222222222','$'

            .code
pause_      proc
            push ax
            push cx
            push dx
;Motivul acestui macro este acela de a lasa un timp minim între accesul la porturile
;hard.
;Desi pe unele masini foarte rapide s-ar putea sa nu existe nici o diferenta importanta
;de timp
            mov  cx,500
            mov  dx,61h
IAR:        in   al,dx      ;in   al,dx
```

```
                loop IAR
                pop dx
                pop cx
                pop ax
endp

; pt. intrare dx:ax=segment:offset adress
; out  DH = Page(0-F)
; AX = Offset

MakePage proc                   ;procedura care calculeaza pagina si ofsetul
                push bx
                mov bl,dh
                shr bl,1
                shr bl,1
                shr bl,1
                shr bl,1
                shl dx,1
                shl dx,1
                shl dx,1
                shl dx,1
                add ax,dx
                adc bl,0
                mov dh,bl
                pop bx
                ret
endp MakePage
```

```
start:
        mov ax,@data
        mov ds,ax
        mov dx,ax           ;sursa segmentul de date curent
        mov ax,offset mesage
        call MakePage
        mov byte ptr ds:[page_0],dh
        mov word ptr ds:[off0],ax
        mov dx,@data        ; destinatia
        mov ax,offset message1
        call MakePage
        mov byte ptr ds:[page_1],dh
        mov word ptr ds:[off1],ax

;Scriu pe ecran sirurile initiale
        mov ah,09h
        xor al,al
        mov dx,offset mesage
        int 21h
        mov dx,offset message1
        int 21h

;Pregatesc DMA
;se fac canalele Disable pt. a putea fi programate
        mov al,0100b
        out mask01,al       ;setez maska pt canalul 0 deci îl fac disable =>
                            ;acum îl pot programa
```

```
 pause_
            mov al,05h
            out mask01,al              ;acelasi lucru pt. canalul 1
pause_
            mov al,0;
            out clear_fl,al            ;clear FL primul pas care trebe facut
pause_
            mov al,10001000b ;modul pt. canal 0
            out mode,al
pause_
            mov al,10000101b ;modul pt. canal 1
            out mode,al
pause_
            mov byte ptr al,ds:[page_1] ;setez pagina pt. canalul 1
            out page1,al
pause_
            mov byte ptr al,ds:[page_0] ; setez pagina pt. primul canal 0
            out page0,al
pause_
            mov word ptr ax,ds:[off0]
            out ch0,al
pause_
            mov al,ah
            out ch0,al          ;programez adresa pt. primul canal 0
pause_
            mov word ptr ax,ds:[off1þ
            out ch1,al
```

```
pause_
        mov al,ah
        out ch1,al          ;acelasi lucru pt. canalul 1
pause_
        mov ax,27
        out ch0_cnt,al
pause_
        mov al,ah
        out ch0_cnt,al      ;programez numarul de octeti de mutat se time
                            seama de ;canalul 1 si nu de canalul 0
pause_
        mov ax,27
        out ch1_cnt,al
pause_
        mov al,ah
        out ch1_cnt,al      ;programez numarul de octeti de mutat se time
                            ;seama de canalul
                            ;1 si nu de canalul 0
pause_
        mov al,00001001b ;enable memory to memory transfer
        out cmmd,al
pause_
        mov al,001b
        out un_mask,al      ;demaschez calalele DMA 0 si 1
pause_
        mov al,000b
        out un_mask,al
```

```
pause_
        mov al,100b          ;pt canalul 0
        out dma_request,al ;cerere DMA
pause_
        mov ah,09h           ;Scriere mesaj dupa transfer
        xor al,al
        mov dx,offset mesage
        int 21h
        mov dx,offset message1
        int 21h
        mov ax,4c00h
        int 21h
end start
end code
```

Bibliografie

[1] **Dobrota V. si altii** - *Aplicatii în sisteme cu microprocesoare din familia intel 80x86*, vol.2.

[2] **Athanasiu I., Panoiu A.** – *Calculatoare Personale*: *Microprocesoarele 8086, 286, 386*, Editura Teora, Bucuresti, 1993.

ARHITECTURA MICROPROCESOARELOR MIPS R2000/R3000

1. Scopul lucrarii

Lucrarea de fata urmareste familiarizarea cu o arhitectura de procesor scalar RISC. Astfel, vom studia modurile de adresare, sintaxa asamblor, setul de instructiuni, partajarea memoriei s.a. la procesorul MIPS R2000 - un exemplu mai recent de masina load/store, acesta putând fi considerat reprezentantul unei masini RISC complete.

2. Memento teoretic

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

Arhitectura calculatoarelor MIPS este simpla si obisnuita, putând fi usor înteleasa si învatata. Procesorul contine 32 registrii de uz general si un set de instructiuni bine proiectat care-l face o tinta favorabila pentru generarea codului într-un compilator. Pe lânga 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.

Arhitectura MIPS, ca majoritatea calculatoarelor RISC, este dificil de programat direct datorita întârzierilor introduse de branch-uri (instructiuni de ramificatie) si load-uri (instructiuni cu referire la memorie) precum si datorita restrictiilor introduse de modurile de adresare. Impedimentul este acceptabil întrucât aceste calculatoare sunt destinate a fi programate în limbaje de nivel înalt, si astfel, ele prezinta o interfata potrivita pentru programatorii de compilatoare

mai degraba decât pentru cei ce scriu în limbaje de asamblare. Pentru a atinge o performanta ridicata, compilatoarele MIPS trebuie sa foloseasca registrii în mod eficient.

O buna parte a complexitatii programelor rezulta din instructiunile care implica întârzieri. 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).

Proiectantii MIPS au furnizat doua moduri de acces al operanzilor. Primul este de a face mai rapid accesul la constante mici si al doilea de a face branch-urile mai eficiente. Compromisul realizat de proiectanti este de a pastra toate instructiunile de aceeasi lungime, necesitând diferite tipuri de formate de instructiuni pentru instructiuni diverse.

3. Desfasurarea lucrarii

3.1. Schema bloc si registrii procesorului MIPS R2000

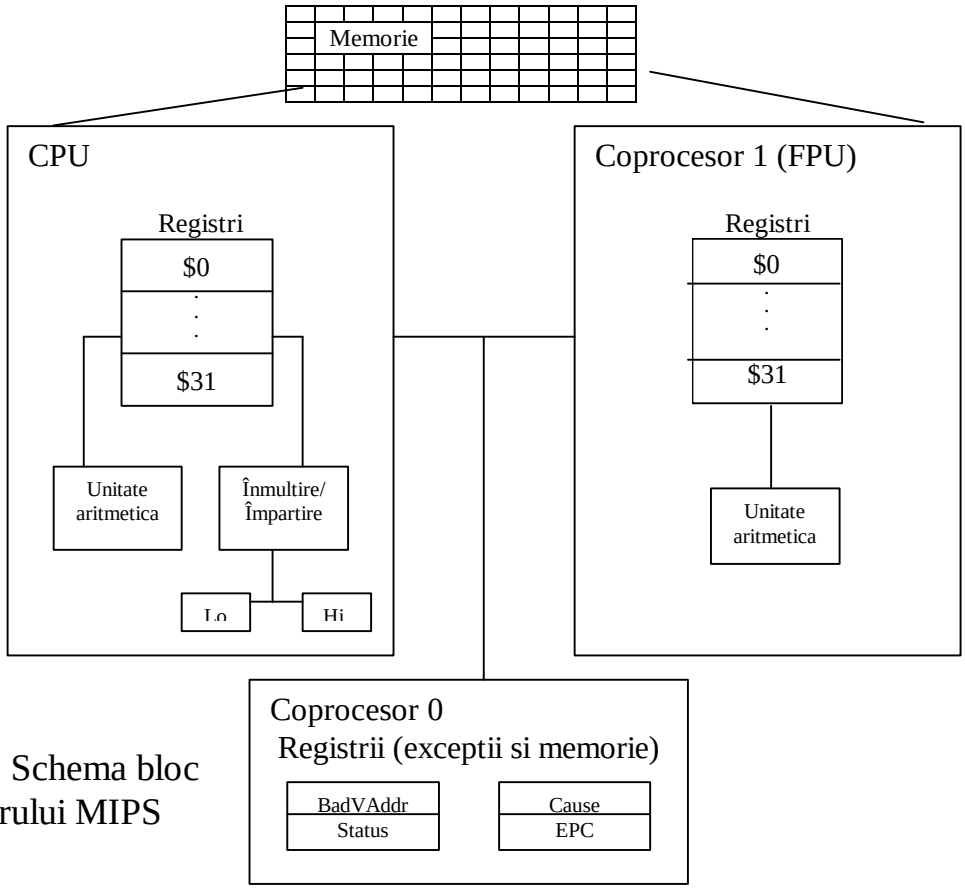


Figura 1. Schema bloc a procesorului MIPS R2000

Unitatea centrala de procesare a MIPS contine 32 registrii generali numerotati 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 conventii despre cum ar trebui folositi registrii. Aceste conventii sunt principii calauzitoare, care nu sunt fortate de catre hardware. Totusi, un program care violeaza aceste principii nu va functiona perfect cu alt software.

Numele Registrilor	Numarul Registrilor	Utilizarea Registrilor
Zero	0	Constanta 0.
at, k_0, k_1	1, 26, 27	Rezervati sistemului de operare; nu trebuie folositi de catre programele utilizator sau compilatoare.

v_0, v_1	2, 3	Pastreaza rezultatele returnate de functii în urma apelurilor sistem sau (în v_0) - codul apelurilor sistem.
$a_0 \div a_3$	4 ÷ 7	Transmiterea primelor patru argumente rutinei apelate (restul argumentelor sunt puse pe stiva).
$t_0 \div t_9$	8 ÷ 15, 24, 25	Registrii de salvare ai rutinei apelante, memoreaza temporar valori care nu trebuie pastrate de-a lungul apelului.
$s_0 \div s_9$	16 ÷ 23	Registrii 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. Registrii MIPS si conventii de utilizare

În plus, coprocesorul 0 contine registrii care sunt folositi pentru tratarea exceptiilor, pe care îi prezentam în tabelul 2.

Numele Registrului	**Numãr**	**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 2. Registrii coprocesorului 0

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

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 2).

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 3. Modurile de Adresare la MIPS R2000/R3000

Majoritatea instructiunilor load si store opereaza doar cu date aliniate. O cantitate este aliniata daca adresa sa de memorie este un multiplu a marimii sale în octeti. De aceea, un obiect de jumatate 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*).

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, 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 le aplice operatiile potrivite.

Comentariile în fisiere de asamblare încep cu simbolul #. Orice urmeaza acestui caracter pâna la sfârsitul 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         # Must be global
main:   lw $t0, item
```

Numerele sunt implicit în baza 10. Daca sunt precedate de caracterul 0x, ele sunt interpretate ca 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 - alineaza datele urmatoare într-un domeniu de 2^n octeti. De exemplu,

.align 2 alineaza 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*.

.ascii str - memoreaza sirul *str* în memorie, dar nu si terminatorul null.

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

.byte b_1, ..., b_n - 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, ..., d_n - memoreaza cele n numere în flotant dubla precizie în locatii succesive de memorie.

.extern sym size - declara data memorata la *sym* de dimensiunea *size* si este un simbol global. Aceasta directiva valideaza asamblorul sa memoreze datele într-o fractiune a segmentului de date care este accesata eficient prin registrul $gp.

.float f_1, ..., 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, ...,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 revalideaza avertismentele. Întrucât instructiunile false expandeaza în cod care foloseste registrul $1, programatorii trebuie sa fie foarte atenti când parasesc valorile din registru.

.space n - aloca 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, ..., w_n - memoreaza cuvinte de 32 de biti în locatii succesive de memorie.

3.4. Formatul instructiunilor si setul de instructiuni 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

0	R_s	R_t	R_d	0	0x20
6	5	5	5	5	6

adunare cu depasire (overflow)

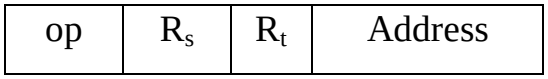
Instructiunea *add* consta din 6 câmpuri. Dimensiunea fiecarui câmp în biti este indicata de numerele de sub câmp. Aceasta instructiune începe cu 6 biti de 0. Urmeaza codificarea registrului sursa pe 5 biti, registru tampon si registrul destinatie tot pe 5 biti. Un alt câmp general este Imm_{16}, care 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_s – primul registru sursa, operand al instructiunii
- R_t – al doilea registru sursa
- R_d – registrul destinatie; pastreaza rezultatul operatiei
- *Shamt* – cantitate de 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:



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.

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.

3.4.1. Instructiuni aritmetice si logice

abs R_{dest}, R_{src} Valoare absoluta

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

add $\mathbf{R_{dest}}$, $\mathbf{R_{src1}}$, $\mathbf{S_{rc2}}$ 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}.

div **R_{src1}, RS_{rc2}**	Împartire cu depasire
divu **R_{src1}, RS_{rc2}**	Împartire fara depasire

Împarte continutul celor doua registre. Câtul 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 câtul împartirii întregi dintre R_{src1} si S_{rc2} în registrul destinatie R_{dest}.

mul R_{dest}, R_{src1},S_{rc2}	Înmultire fara depasire
mulo R_{dest}, R_{src1},S_{rc2}	Înmultire cu depasire
mulou R_{dest}, R_{src1},S_{rc2}	Înmultire fara semn cu depasire

Pune în registrul R_{dest} rezultatul obtinut în urma înmultirii dintre întregii din registrii R_{src1} si S_{rc2}.

***mult* $\mathbf{R_{src1}}$, $\mathbf{R_{src2}}$** Înmultire

***multu* $\mathbf{R_{src1}}$, $\mathbf{R_{src2}}$** Înmultire fara semn

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

neg R_{dest}, R_{source} Valoarea negativa cu depasire

negu R_{dest}, R_{source} Valoarea negativa fara depasire

Valoarea negativa a întregului din registrul R_{source} este depusa în registrul R_{dest}.

nor R_{dest}, R_{src1}, S_{rc2} NOR

Rezultatul operatiei NOR logic a întregilor din registrele R_{src1} si S_{rc2}, este depus în registrul R_{dest}.

***not* $\mathbf{R_{dest}}$, $\mathbf{R_{source}}$** NOT

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

***or* $\mathbf{R_{dest}}$, $\mathbf{R_{src1}}$, $\mathbf{S_{rc2}}$** OR

***ori* $\mathbf{R_{dest}}$, $\mathbf{R_{src1}}$, Imm** OR imediat

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

rem R_{dest}, R_{src1}, S_{rc2}	Rest
remu 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 si depinde de conventiile masinii pe care ruleaza programul.

rol R_{dest}, R_{src1}, S_{rc2}	Roteste la stânga
ror 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} si pune rezultatul în registrul destinatie.

sll R_{dest}, R_{src1}, S_{rc2}	Deplaseaza logic la stânga
sllv R_{dest}, R_{src1}, S_{rc2}	Deplaseaza logic la stânga variabil
sra R_{dest}, R_{src1}, S_{rc2}	Deplaseaza aritmetic la dreapta
srav R_{dest}, R_{src1}, S_{rc2}	Deplaseaza aritmetic la dreapta variabil
srl R_{dest}, R_{src1}, S_{rc2}	Deplaseaza logic la dreapta
srlv 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 $\mathbf{R_{dest}}$, $\mathbf{R_{src1}}$, $\mathbf{S_{rc2}}$	Scadere cu depasire
subu $\mathbf{R_{dest}}$, $\mathbf{R_{src1}}$, $\mathbf{S_{rc2}}$	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}.

3.4.2. Instructiuni 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 destinatie precum si extensia de semn.

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

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

lwcz R_{dest}, address Încarcare cuvânt în coprocesor

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

lwl R_{dest}, address Încarcare din stânga a cuvântului
lwr R_{dest}, address Încarcare din dreapta a cuvântului

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

sb R_{source}, address Memorare octet

Memoreaza octetul mai putin semnificativ al registrului sursa la adresa specificata de *address*.

sd R_{source}, address Memorare dublu cuvânt

Memoreaza 8 octeti (continutul registrilor R_{source} si R_{source} +1) la adresa indicata de *address*.

sh R_{source}, address Memorare semicuvânt

Memoreaza primii doi octeti mai putin semnificativi al registrului sursa la adresa specificata de *address*.

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

Memoreaza cuvântul din registrul sursa la adresa *address*.

swcz R_{source}, address Memorare cuvânt din coprocesor

Memoreaza cuvântul din R_{source} al coprocesorului z la adresa *address*.

swl R_{source}, address Memorare portiune stânga din cuvânt

swr R_{source}, address Memorare portiune dreapta din cuvânt

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

ulh R_{dest}, address Încarcare semicuvânt de la adresa nealiniata

ulhu R_{dest}, address Încarcare semicuvânt fara semn de la adresa nealiniata

Încarcare semicuvânt si extensie semn de la adrese posibil nealiniate în registrul destinatie.

ulw R_{dest}, address Încarcare cuvânt de la adresa nealiniata

Încarcare cuvânt de la adrese posibil nealiniate în registrul destinatie.

ush R_{source}, address — Memorare semicuvânt la adresa nealiniata

Memoreaza semicuvântul mai putin semnificativ din registrul sursa la o adresa posibil nealiniata.

usw R_{source}, address — Memorare cuvânt la adresa nealiniata

Memoreaza cuvântul din registrul sursa la o adresa posibil nealiniata.

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

3.4.3. Instructiuni de întreruperi si exceptii

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.

3.4.4. Instructiuni de comparatie

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

seq R_{dest}, R_{src1}, S_{rc2} Seteaza flag în caz de egalitate

sge R_{dest}, R_{src1}, S_{rc2} Seteaza flag pe relatia de mai mare

sgeu R_{dest}, R_{src1}, S_{rc2}	Seteaza flag pe relatia de mai mare fara semn
sgt R_{dest}, R_{src1}, S_{rc2}	Seteaza flag pe relatia de mai mare strict
sgtu R_{dest}, R_{src1}, S_{rc2}	Seteaza flag pe relatia de mai mare strict fara semn
sle R_{dest}, R_{src1}, S_{rc2}	Seteaza flag pe relatia de mai mic sau egal
sleu R_{dest}, R_{src1}, S_{rc2}	Seteaza flag pe relatia de mai mic sau egal fara semn
slt R_{dest}, R_{src1}, S_{rc2}	Seteaza flag pe relatia de mai mic strict
slti R_{dest}, R_{src1}, Imm	Seteaza flag pe relatia de mai mic strict decât valoarea imediata
sltu R_{dest}, R_{src1}, S_{rc2}	Seteaza flag pe relatia de mai mic decât valoare fara semn
sltiu R_{dest}, R_{src1}, Imm	Seteaza flag pe relatia de mai mic decât valoare imediata fara semn
sne 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.

3.4.5. Instructiuni 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}-1instructiuni 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*.

bczt label	Branch pe conditie adevarata în coprocesor
bczf label	Branch pe conditie falsa în coprocesor

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

beq R_{src1}, S_{rc2}, label	Branch pe egalitate
bge R_{src1}, S_{rc2}, label	Branch pe conditie de mai mare sau egal
bgeu R_{src1}, S_{rc2}, label	Branch pe conditie de mai mare sau egal, comparatie fara semn
bgt R_{src1}, S_{rc2}, label	Branch pe conditie de mai mare strict
bgtu R_{src1}, S_{rc2}, label	Branch pe conditie de mai mare strict, comparatie fara semn
ble R_{src1}, S_{rc2}, label	Branch pe conditie de mai mic sau egal
bleu R_{src1}, S_{rc2}, label	Branch pe conditie de mai mic sau egal, comparatie fara semn
blt R_{src1}, S_{rc2}, label	Branch pe conditie de mai mic strict

bltu R_{src1}, S_{rc2}, label	Branch pe conditie de mai mic, comparatie fara semn
bne R_{src1}, S_{rc2}, 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.

***beqz* $\mathbf{R_{source}}$, label**	Branch pe egalitate cu zero
bgez R_{source}, label	Branch pe conditie de mai mare sau egal cu zero
bgezal R_{source}, label	Branch pe conditie de mai mare sau egal cu zero si salvare legatura
bltzal R_{source}, label	Branch pe conditie de mai mic strict decât zero si salvare legatura
bgtz R_{source}, label	Branch pe conditie de mai mare strict decât zero
***blez* $\mathbf{R_{source}}$, label**	Branch pe mai mic sau egal decât zero
bltz R_{source}, label	Branch pe conditie de mai mic strict decât zero
bnez R_{source}, label	Branch pe neegalitate cu zero

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

***j* label**	Jump

Salt neconditionat la eticheta.

***jal* label** Jump si salvare legatura

***jalr* R_{source}** 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.

***jr* R_{source}** Jump la adresa din registru

Salt neconditionat la adresa specificata de registrul sursa.

3.4.6. Instructiuni de transfer

***move* R_{dest}, R_{source}** 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 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.

***mfhi* R_{dest}** Transfera din HI

***mflo* R_{dest}** Transfera din LO

Muta continutul registrului HI sau LO în registrul destinatie.

mthi $\mathbf{R_{dest}}$	Transfera în HI
mtlo $\mathbf{R_{dest}}$	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.

mfcz R_{dest}, $C_{opsource}$	Transfer din coprocesor z

Muta continutul registrului $C_{opsource}$ al coprocesorului z în registrul destinatie R_{dest}.

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, mai ales în lucrarile de laborator "*Utilizarea simulatorului SPIM*" si "*Investigatii arhitecturale utilizând simulatorul SPIM*".

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 registrii, care sunt numerotati $\$f_0$ - $\$f_{31}$. Deoarece aceste registre sunt doar pe 32 de biti, sunt necesare doua din 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 dubla precizie
abs.s FR_{dest}, FR_{source}	Valoare absoluta registru flotant simpla 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 registrii sursa în simpla sau dubla precizie si pune rezultatul în registrul destinatie.

c.eq.d FR_{src1}, FR_{src2}	Comparatie de egalitate în dubla precizie
c.eq.s FR_{src1}, FR_{src2}	Comparatie de egalitate în simpla precizie

Se seteaza flagul conditie în flotant pe true daca continutul registrilor sursa este identic.

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}.

c.lt.d FR_{src1}, FR_{src2}	Comparatie pe relatie de strict mai mic în dubla precizie
c.lt.s FR_{src1}, FR_{src2}	Comparatie pe relatie de strict mai mic în simpla precizie

Se seteaza flagul conditie în flotant pe true daca FR_{src1} este mai mic strict 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 registrii 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 FR_{dest}, FR_{src1}, FR_{src2}	Înmultire în dubla precizie
mul.s FR_{dest}, FR_{src1}, FR_{src2}	Înmultire în simpla precizie

Calculeaza produsul celor doi registrii FR_{src1} si FR_{src2} si pune rezultatul în registrul FR_{dest}.

neg.d FR_{dest}, FR_{source}	Neaga valoare dubla precizie
neg.s FR_{dest}, FR_{source}	Neaga valoare simpla precizie

Neaga valoarea registrului sursa si o depune în registrul destinatie.

s.d FR_{dest}, address	Memorare valoare dubla precizie la adresa
s.s FR_{dest}, address	Memorare valoare simpla precizie la adresa

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

sub.d FR_{dest}, FR_{src1}, FR_{src2}	Scadere în dubla precizie
sub.s FR_{dest}, FR_{src1}, FR_{src2}	Scadere în simpla precizie

Calculeaza diferenta celor doi registrii FR_{src1} si FR_{src2} si pune rezultatul în registrul FR_{dest}.

3.5. Utilizarea memoriei si conventii de apel

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.

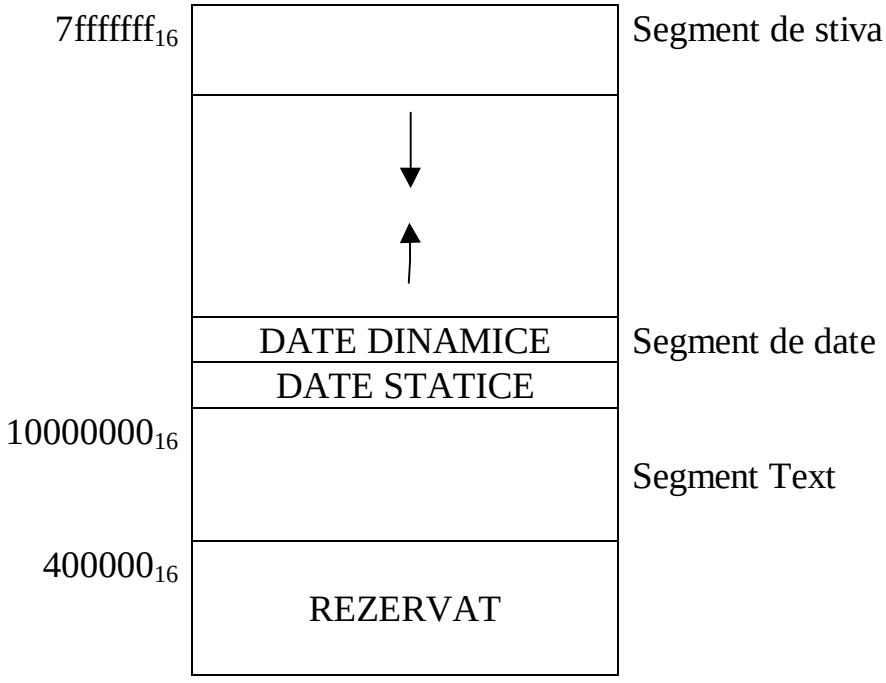


Figura 2. Harta de memorie a microprocesorului MIPS R2000

Conventiile de apel descrise folosesc **gcc**, nu compilatorul nativ al MIPS, care foloseste o conventie mai complexa si usor mai rapida. Diferenta dintre cele doua compilatoare este ca, de obicei, compilatorul MIPS nu foloseste un indicator de cadru, astfel încât acest registru este disponibil ca un alt registru de salvare $\$s_8$.

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 urmatorii pasi:

a. Transmiterea argumentelor. Prin conventie, primele patru argumente sunt transferate în registrii $\$a_0$ - $\$a_3$ (chiar unele compilatoare mai simple simplifica aceasta conventie si transfera toate argumentele pe stiva). Argumentele ramase sunt puse pe stiva.
b. Salveaza registrii $\$t_0$ - $\$t_9$.
c. Executa instructiunea *jal*.

În subrutina apelata sunt necesari urmatorii pasi:

a. Se calculeaza cadrul de stiva prin scaderea marimii cadrului din indicatorul de stiva.
b. Salveaza registrii $\$s_0$ - $\$s_7$ 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 $\$v_0$ si se executa urmatorii 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.

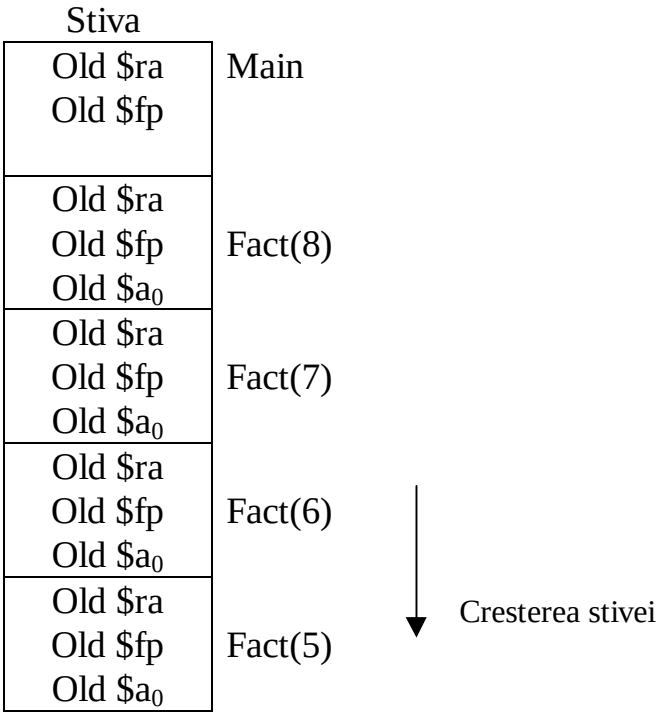


Figura 3. Imaginea stivei pentru calculul valorii fact(8)

Bibliografie

[1] **Hennessy J., Patterson D.** - *Computer Organization and Design*: *The Hardware / Software Interface*, Morgan Kaufmann Publishers, San Francisco, California, 1994

[2] **Kane G., Heinrich J.** – *MIPS RISC Architecture*, Prentice Hall, 1992

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

UTILIZAREA SIMULATORULUI SPIM

1. Scopul lucrarii

Lucrarea de fata prezinta o descriere detaliata a simulatorului SPIM. Volumul mare de informatie poate ascunde faptul ca SPIM este simplu si usor de folosit ca program. Este o lucrare de laborator rapida despre SPIM, cu scopul de a încarca, depana si rula programe MIPS simple.

2. Memento teoretic

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: una simpla care se numeste *spim*. Ea ruleaza pe orice tip de terminal. Se tasteaza comanda la terminal iar *spim* o executa. Versiunea superioara se numeste *xpim* si ruleaza pe un sistem X - window. *Xpim* este mult mai usor de învatat si folosit deoarece comenzile sunt întotdeauna vizibile pe ecran si afiseaza permanent continutul registrilor, dar necesita un display mapat pe bit.

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 si larga aplicabilitate nu poate fi egalata de hardware-ul adevarat.

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 folositoare. Un altul ar fi progresul spre calculatoare noi si rapide, facând 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 înteles 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 folositoare 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 harware-ului existent. Pentru aceasta, vom urma conventiile programatorilor în limbaje de asamblare MIPS, care este folosit în mod obisnuit de catre masina extinsa.

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 latenta 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" (vezi "*Arhitectura microprocesoarelor MIPS R2000/R3000*").

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, SPIM foloseste "little-endian", în timp ce pe un MacIntosh, HP Bobcat sau Sun SPARC, ordinea folosita este "big-endian".

3. Desfasurarea lucrarii

3.1. Optiuni în linia de comanda

Ambele versiuni de SPIM (*spim* si *xspim*) accepta urmatoarele optiuni în linia de comanda.

- *bare* –
  simuleaza o masina MIPS simpla, fara pseudoinstructiuni sau moduri de adresare suplimentare asigurate de asamblor. Implica *quiet*.
- *asm* –
  simuleaza masina virtuala MIPS furnizata de asamblor. Optiunea *asm* este implicita.
- *notrap* –
  nu se încarca handler-ul de exceptie standard si codul de start-up. Acest handler trateaza exceptiile. Când are loc o exceptie, SPIM executa salt la locatia 80000080H, care contine codul de deservire a exceptiei. Codul de start-up apeleaza rutina *main*. Fara o rutina de start-up, SPIM începe executia de la instructiunea etichetata cu *__start*.
- *trap* –

încarca handler-ul standard de exceptie si codul de start-up. Aceasta optiune este implicita.

- *noquiet –*

  afiseaza un mesaj când apare o exceptie. Este implicita.

- *quiet –*

  nu afiseaza mesaj la exceptie.

- *nomapped_IO –*

  dezactiveaza maparea memoriei.

- *mapped_IO –*

  valideaza facilitatea de mapare a memoriei într-un dispozitiv de intrare/ iesire. Programele care folosesc apeluri sistem SPIM pentru a citi de la un terminal nu pot, de asemenea sa foloseasca maparea memoriei.

- *file –*

  încarca si executa codul scris în asamblare al fisierului.

- *execute –*

  încarca si executa codul în fisiere executabile MIPS *a.out.* Aceasta comanda este disponibila doar când SPIM ruleaza pe un sistem cu procesor MIPS. Programul nu poate invoca nici un serviciu al sistemului de operare (în cazul unei intrari sau iesiri), întrucât SPIM nu simuleaza trap-urile (exceptiile) nucleului MIPS.

- *s seg size –*

  seteaza dimensiunea initiala a segmentului de memorie *seg* la valoarea de *size* octeti. Segmentele de memorie se numesc: **text, data, stack, ktext si kdata.** Segmentul **text** contine instructiunile din program. Segmentul **data** pastreaza datele programului. **Stack** este segmentul de stiva din timpul rularii programului. Pe lânga executia programului, SPIM executa si cod sistem care trateaza întreruperile si exceptiile. Acest cod se afla într-o zona separata a spatiului de adresa numita kernel (nucleu). Segmentul **ktext** pastreaza instructiunile acestui cod

iar **kdata** datele sale. Nu exista **kstack** întrucât codul sistem foloseste aceeasi stiva ca si programul utilizator. De exemplu, linia de comanda urmatoare:

- s data 2000000 - semnifica faptul ca segmentul de date utilizator începe la adresa 0x10010000 si are dimensiunea initiala de 2000000 octeti.

- *l seg size* –

  stabileste limita superioara a segmentului de memorie *seg*, care poate fi de *size* octeti. Segmentele de memorie care pot creste sunt: data, stack si kdata.

3.2. Interfata cu terminalul

Versiunea simpla a SPIM este apelata cu *spim*. Nu este necesar un display mapat pe bit si poate fi rulat de pe orice terminal. Desi *spim* poate fi mai dificil de învatat, el opereaza exact ca *xspim* si asigura aceeasi functionalitate. Interfata cu terminalul se bazeaza pe comenzile:

- *exit* –

  opreste simulatorul.

- *read "file"* –

  citeste fisierul "file" scris în limbaj de asamblare MIPS. Daca fisierul a fost deja citit în SPIM, sistemul trebuie reinitializat sau simbolurile globale vor trebui multiplu definite.

- *load "file"* –

  comanda sinonima cu read.

- *execute "a.out"* –

  citeste executabile MIPS în simulator. Comanda este disponibila numai daca SPIM ruleaza pe un sistem ce contine doar procesor MIPS.

- *run <addr> –*

  porneste rularea programului. Daca argumentul optional <addr> este prezent, programul începe de la aceasta adresa, altfel de la simbolul global __start, care este codul de start implicit.

- *step <N> –*

  ruleaza programul în grupuri de N instructiuni (implicit 1). Afiseaza instructiunile care se executa.

- *continue –*

  continua executia programului fara oprire.

- *print $N –*

  afiseaza continutul registrului N.

- *print $fN –*

  afiseaza continutul registrului flotant N.

- *print addr –*

  afiseaza continutul adresei de memorie addr.

- *print _sym –*

  afiseaza tabela de simboluri, adresa de simboluri globale (nu locale).

- *reinitialize –*

  sterge (anuleaza) continutul registrilor si memoria.

- *breakpoint addr –*

  stabileste un punct de întrerupere (breakpoint) la adresa *addr*. Aceasta poate fi fie o adresa de memorie fie o eticheta.

- *delete addr –*

  sterge toate punctele de întrerupere de la adresa *addr*.

- *list –*

  listeaza toate punctele de întrerupere.

- **.** –

  restul liniei este o instructiune în asamblare care este stocata în memorie.

- <nl> –

linie noua; care determina reexecutarea comenzii anterioare.

- ? –

afiseaza o cerere de help.

Majoritatea comenzilor pot fi abreviate la prefixul: ex, re, l, ru, s, p. Comenzile de genul reinitialize necesita un prefix mai lung.

3.3. 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_0 si argumentele în registrele a_0 ... a_3 (sau f_{12} pentru valori în flotant). Apelurile sistem care returneaza valori pun rezultatele în registrele v_0 sau f_0 pentru operatii în flotant.

Serviciul	Codul Apelului Sistem	Argumentele	Rezultatul
Print_int	1	a_0 = integer	
Print_float	2	f_{12} = float	
Print_double	3	f_{12} = double	
Print_string	4	a_0 = string	
Read_int	5		Integer (în v_0)
Read_float	6		Float (în f_0)
Read_double	7		Double (în f_0)
Read_string	8	a_0 = buffer, a_1 = lungimea	
Sbrk	9	a_0 = cantitate	Adresa (în v_0)
Exit	10		

Tabelul 1. Servicii Sistem

Sbrk întoarce un pointer la un bloc de memorie ce contine n octeti, iar *exit* opreste rularea programului.

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

Exemplul 1.

Pentru exemplificare vom scrie codul necesar afisarii 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 secventei de instructiuni vom observa ca instructiunea *li $v0, 4* – care este de fapt o pseudoinstructiune, este înlocuita de instructiunea *ori $2, $0, 4*; instructiunea de încarcare imediata în registru se obtine dintr-o operatie de SAU între registrul 0, care este întotdeauna 0, si valoarea respectiva.

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 "Dati 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
```

3.4. Pseudoinstructiuni folosite în limbaj de asamblare MIPS

Prezentam în continuare câteva din pseudoinstructiunile pe care le vom întâlni în lucrarile 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

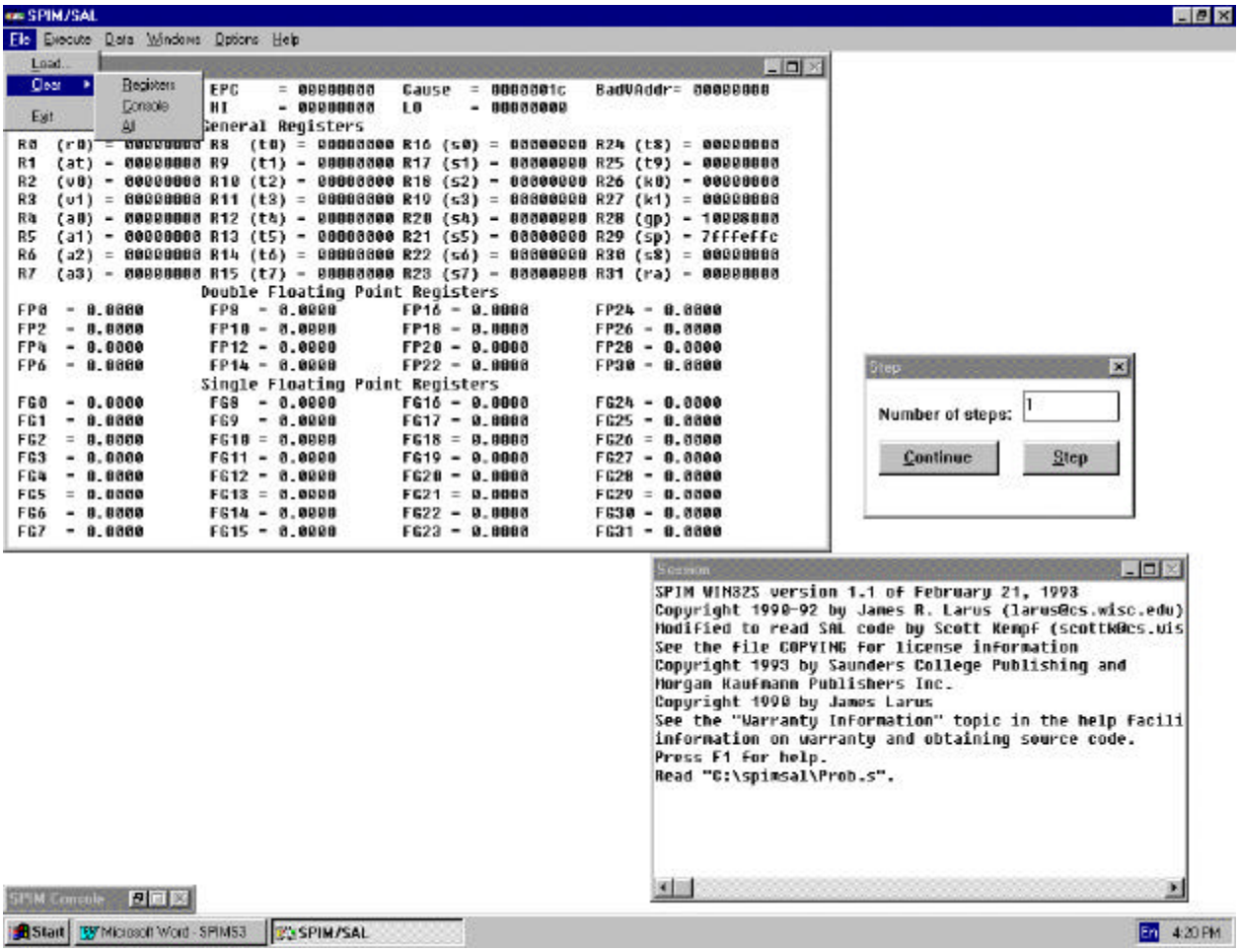
Apel sistem:

syscall - nu are argumente si determina un apel sistem de citire de la tastatura/ afisare pe consola sau terminare program

Celelalte pseudoinstructiuni aritmetico-logice, cu referire la memorie, de salt (conditionat sau nu), de transfer au fost prezentate în lucrarea "*Arhitectura microprocesoarelor MIPS R2000/R3000*".

3.5. Ghid de utilizare al simulatorului

Când pornim simulatorul pe ecran apare fereastra urmatoare.



Pasul urmator ar fi încarcarea 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. Încarcarea fisierelor se poate face totodata si din linia de comanda.

Pentru reluarea unui program sau încarcarea 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.

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 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 aparea 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 atribute 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.

Bibliografie

[1] **Hennessy J., Patterson D.** - *Computer Organization and Design*: *The Hardware / Software Interface,* Morgan Kaufmann Publishers, San Francisco, California, 1994

[2] **Kane G., Heinrich J.** – *MIPS RISC Architecture*, Prentice Hall, 1992

INVESTIGATII ARHITECTURALE UTILIZÂND SIMULATORUL SPIM

1. Scopul lucrarii

Lucrarea de fata urmareste studiul benchmark-urilor existente, deprinderi practice privind programarea în asamblare MIPS.

2. Desfasurarea lucrarii

2.1. Note explicative referitoare la benchmark-ul Input.s

Benchmark-ul *Input.s* este un program de test simplu, care utilizeaza pseudoinstructiunile ce expandeaza în apeluri sistem, prezentate în lucrarea de laborator anterioara, 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 registrii, continutul locatiilor de memorie, instructiunile efectiv executate în urma rularii pas cu pas sau în mod continuu a programului.

```
.data
d1:    .asciiz "gimme an int: "
d2:    .asciiz "gimme a char: "
d3:    .asciiz "gimme a string: "
dog:   .space 80
```

Zona de date utilizator începe la adresa 10010000h. Sirul de la eticheta d1 este memorat de la aceasta adresa pe 15 octeti (14 caractere plus un octet pentru terminatorul NULL).

d1: 10010000h → 1001000Eh (15 octeti)

d2: 1001000Fh → 1001001Dh (15 octeti)

d3: 1001001Eh → 1001002Fh (17 octeti)

dog: 10010030h → 10010080h (80 octeti de spatiu)

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.

Comentariu asupra secventei de instructiuni:

Operatia efectuata	Opcode-ul Pseudoinstructiunii	Nr. crt. Instr. Simpla	Expandare în instructiuni MIPS simple	Semnificatie instructiune simpla
# afisare string si citire întreg de la tastatura	puts d1	1.	lui $4, 4097	încarcare în registrul a_0 a adresei de unde se va afisa stringul.
		2.	ori $2, $0, 4	încarcare în v_0 a codului de afisare string pentru apelul sistem.
		3.	syscall	afisare pe consola a stringului de la

				eticheta d1.
# citire de la tastatura în registrul a_0	geti $\$a_0$	1.	ori $2, $0, 5	încarcare în v_0 a codului operatiei de citire întreg.
		2.	syscall	citire de la tastatura a unui întreg; rezultatul este depus în v_0.
		3.	add $4, $2, $0	transfer valoare din v_0 în a_0.
# incrementare cu 1	addi $\$a_0$, $\$a_0$, 1	1.	addi $4, $4, 1	incrementare cu 1 a registrului a_0.
# afisare continut registru a_0	puti $\$a_0$	1.	ori $2, $0, 1	încarcare în v_0 a codului de afisare întreg
		2.	addi $4, $4, $0	pregatire în a_0 argument pentru apel sistem
		3.	syscall	afisare pe consola a întregului din $\$a_0$.
# afisare caracter ‘\n’	putc ‘\n’	1.	ori $4, $0, 10	pune în a_0 codul ASCII al caracterului ‘\n’ (0x0A)
		2.	ori $2, $0, 11	încarcare în v_0 a codului de afisare

				caracter
		3.	syscall	afisare caracter pe consola
# afiseaza mesaj de la eticheta d2	puts d2	1.	lui $1, 4097	încarcare octet superior de adresa în registrul at.
		2.	ori $4, $1, 15	încarcare în a_0 adresa etichetei d2.
		3.	ori $2, $0, 4	încarcare în v_0 a codului de afisare string pentru apelul sistem.
		4.	syscall	afisare pe consola mesaj de la eticheta d2
# citeste un caracter în registrul a_0	getc $$a_0$$	1.	ori $2, $0, 12	încarcare în v_0 a codului operatiei de citire caracter.
		2.	syscall	citire de la tastatura a unui caracter; codul sau ASCII este depus în v_0.
		3.	add $4, $2, $0	transfer valoare din v_0 în a_0.
# incrementeaza	addi $$a_0$$, $$a_0$$, 1	1.	addi $4, $4, 1	incrementare cu 1 a registrului a_0

cu 1 codul ASCII al caracterului citit anterior				
# afisare continut registru a_0	puti a_0	1.	ori $2, $0, 11	încarca în v_0 codul de afisare caracter.
		2.	addi $4, $4, $0	pregatire în a_0 argument pentru apel sistem
		3.	syscall	afisare pe consola a codului ASCII din a_0.
# afisare caracter '\n'	putc '\n'	1.	ori $4, $0, 10	pune în a_0 codul ASCII al caracterului '\n' (0x0A)
		2.	ori $2, $0, 11	încarcare în v_0 a codului de afisare caracter
		3.	syscall	afisare caracter pe consola
# încarcare adresa eticheta d3	la a_0, d3	1.	lui $1, 4097	încarcare octet superior de adresa în registrul at.
		2.	ori $4, $1, 30	încarcare în a_0 adresa etichetei d3.

# încarcare cod afisare string de la eticheta d3	li v_0$, 4	1.	ori $2, $0, 4	încarcare în v_0 a codului de afisare string pentru apelul sistem.
# apel sistem	syscall	1.	syscall	afisare mesaj de la eticheta d3
# încarcare adresa eticheta dog	la a_0$, dog	1.	lui $1, 4097	încarcare octet superior de adresa în registrul at.
		2.	ori $4, $1, 47	încarcare în a_0 adresa etichetei d3.
# încarcare în registru a_1 a lungimii sirului	li a_1$, 80	1.	ori $5, $0, 80	încarcare în a_1 a lungimi sirului de la eticheta dog
# încarcare cod citire string de la tastatura	li v_0$, 8	1.	ori $2, $0, 8	încarcare în v_0 a codului de citire string pentru apelul sistem.
# apel sistem	syscall	1.	syscall	citire string de la tastatura în octetii de la adresa specificata de eticheta dog.
# afiseaza string citit anterior de la	puts dog	1.	lui $1, 4097	încarcare octet superior de adresa în registrul at.

eticheta dog				
		2.	ori $4, $1, 47	încarcare în a_0 adresa etichetei dog.
		3.	ori $2, $0, 4	încarcare în v_0 a codului de afisare string pentru apelul sistem.
		4.	syscall	afisare pe consola mesaj de la eticheta dog
# iesire din program	done	1.	ori $2, $0, 10	încarcare în v_0 a codului de iesire din program pentru apelul sistem.
		2.	syscall	apel sistem pentru iesirea din program

2.2. 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 vom implementa 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 vom scrie un program care parcurge numerele naturale si testeaza daca fiecare este prim. Se vor afisa primele astfel de 24 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 *testprime* 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=24×25 numere prime.

Subrutina *test_prime*

```
        .data

d1:     .asciiz "Numarul este prim! "
d2:     .asciiz "Numarul nu este prim!"

        .text
        .globl main
main:
        li      $a0, 137            # Numarul pe care-l testez (n)
        li      $v0, 2              # Primul numar prim (initial i=2)
ET:     sub     $v0, $v0, $a0       # i = i-n
        beqz    $v0, $L1            # Testez i<0
        add     $v0, $v0, $a0       # Refac i; i = i + n;
        divu    $a0, $v0            # Calculez restul împartirii lui n la i
        mfhi    $a1                 # Pun restul în a1
        beqz    $a1, $L2            # Daca restul e 0 salt la eticheta - numarul nu e
                                    # prim
        addi $v0, $v0, 1             # Cresc i-ul - urmatorul numar cu care-l împart
```

```
                                # pe n
        j       ET

$L1:
        li      $a0, d1         # Numarul nu se împarte exact cu nici unul din
                                # numerele mai mici decât el, numarul este prim
        li      $v0, 4
        syscall                 # Afisare mesaj de la eticheta d1
        j       END             # Salt la sfârsitul programului

$L2:
        li      $a0, d2         # Numarul nu este prim
        li      $v0, 4
        syscall                 # Afisare mesaj de la eticheta d2
END:
        done
```

Solutia problemei 1

```
        .text
        .globl main
main:
        li      $a2, 24         # Numarul de numere prime
        li      $a0, 2          # Primul numar prim (numarul prim pe care-l
                                # testez)
        li      $a1, 0          # Contorizarea numerelor prime
ET1:
        sub     $a1, $a1, $a2   # Mai sunt numere prime de afisat ?
        beqz    $a1, END
```

```
        add     $a1, $a1, $a2       # Refacere contor
        jal     test_prime          # Salt la testare numar prim

RESTART:
        add     $a0, $a0, 1         # Urmatorul numar de testat
        j       ET1

test_prime:
        li      $v0, 2              # Primul numar cu care se imparte numarul testat
ET:
        sub     $v0, $v0, $a0       # Am testat toate numerele mai mici decat n ?
        beqz    $v0, $L1
        add     $v0, $v0, $a0       # Refacere $v0
        divu    $a0, $v0            # Calculeaza restul impartirii lui n la valorile
                                    # anterioare lui n mod i
        mfhi    $a3                 # Restul impartirii este depus in a3
        beqz    $a3, END_PROC       # Daca restul este 0 numarul n nu este prim
        addi    $v0, $v0, 1         # Crestem i-ul urmator numar cu care-l impartim
                                    # pe n
        j       ET

$L1:
        add     $a0, $a0, $0        # Afisare numar prim pe ecran
        li      $v0, 1
        syscall
        add     $a1, $a1, 1         # Creste numarul numerelor prime afisate cu 1

        subu    $sp, $sp, 32        # Pregatire stiva pentru scrierea numarului
        sw      $ra, 20($sp)        # urmator care se va testa daca este prim
```

```
        sw      $fp, 16($sp)
        addu    $fp, $sp, 32
        sw      $a0, 0($fp)         # Memoreaza a0

        putc '\n'                   # Afiseaza un retur de car pentru ca urmatorul
                                    # numar prim sa apara pe rand nou
        lw      $a0, 0($fp)         # Refacere a0 pentru a testa daca este nr. prim
        lw      $ra, 20($sp)        # Reface adresa de revenire
        lw      $fp, 16($sp)        # Reface pointer de cadru
        addu    $sp, $sp, 32        # Reface pointer-ul de stiva

END_PROC:
        j       RESTART

END:
        done
```

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 acestei rutine ilustreaza cum programul afecteaza stiva, registrii $sp, $fp, $ra. Programul principal *main* creeaza cadrul de stiva si salveaza registrii $fp si $ra (pointer-ul de cadru si adresa de revenire din procedura). Vom considera dimensiunea minima a stivei de 32 octeti, mai mare decât ar fi necesar pentru memorarea celor doua registre. Sa se ruleze programul pas cu pas, sa se urmareasca continutul stivei si al registrilor, actualizarea registrilor la revenirea din subrutina si corectitudinea rezultatelor.

Solutia problemei 2

```
        .text
        .globl main
main:
        subu   $sp, $sp, 32        # Cadrul de stiva de 32 bytes
        sw     $ra, 20($sp)        # Salveaza adresa de revenire
        sw     $fp, 16($sp)        # Salveaza vechiul pointer de cadru
        addu   $fp, $sp, 32        # Seteaza noul pointer de cadru
```

Programul principal, care începe la eticheta *main*, apeleaza rutina *fact* si paseaza acesteia singurul sau argument n (initial 10). La revenirea din subrutina se va afisat mesaj plus valoarea factorialului.

```
        li     $a0, 10             # Pune argumentul în $a0
        jal    fact                # Apeleaza rutina de calcul a factorialului

        la     $a0, $LC            # Pune adresa string-ului de afisat în $a0
        li     $v0, 4              # Pune codul operatiei în v0
        syscall                    # Afiseaza mesaj
```

La sfârsit, dupa afisarea valorii factorialului, programul se încheie, dar nu înainte de a restaura registrii salvati în stiva si refacerea vechiului cadru de stiva.

```
        mflo   $a0                 # Salveaza rezultatul în a0
        li     $v0, 1              # Pune codul operatiei în v0
        syscall                    # Afiseaza valoarea factorialului

        lw     $ra, 20($sp)        # Reface adresa de revenire
```

```
        lw      $fp, 16($sp)        # Reface pointer-ul de cadru
        addu    $sp, $sp, 32        # Reface cadrul de stiva

        done                        # Încheiere program

        .data
$LC:
        .ascii "The factorial of 10 is = "
```

Rutina factorial are stuctura similara cu programul principal. Întâi se creeaza cadrul de stiva se salveaza registrii care se folosesc $a0, $fp si $ra.

```
        .text
fact:
        subu    $sp, $sp, 32        # Cadrul de stiva este de 32 bytes
        sw      $ra, 20($sp)        # Salveaza adresa de revenire
        sw      $fp, 16($sp)        # Salveaza vechiul pointer de cadru
        addu    $fp, $sp, 32        # Seteaza noul pointer de cadru

        sw      $a0, 0($fp)         # Salveaza argumentul (n)
```

Nucleul acestei rutine realizeaza calculul efectiv n! = n × (n-1)!. Se testeaza daca argumentul este mai mare decât 0. Daca nu, rutina returneaza valoarea 1. Daca este mai mare ca 0, se reapeleaza recursiv rutina *fact*, pentru a calcula (n-1)! si se înmulteste cu n.

```
        lw      $2, 0($fp)          # Load n
        bgtz    $2, $L2             # Branch if n>0
        li      $2, 1               # Returneaza 1
        j       $L1                 # Salt la codul de revenire
```

```
$L2:
        lw      $3, 0($fp)          # Load n
        subu    $2, $3, 1           # Calculeaza n-1
        move    $a0, $2             # Muta valoarea în $a0
        jal     fact                # Reapeleaza rutina de calcul a factorialului

        lw      $3, 0($fp)          # Load n
        mul     $2, $2, $3          # Calculeaza fact(n-1)*n
```

În final, rutina factorial restaureaza registrii si returneaza valoarea factorialului în registrul $v0.

```
$L1:                                # Rezultatul în $2
        lw      $ra, 20($sp)        # Restaureaza $ra
        lw      $fp, 16($sp)        # Restaureaza $fp
        addu    $sp, $sp, 32        # Reface cadrul de stiva
        j       $ra                 # Revenire în programul apelant
```

3. Scrieti un program care sa sorteze un sir de *n* (initial n=5) numere întregi prin metoda *bubblesort*. Rulati programul pentru diverse siruri de numere de dimensiuni diferite.

Solutia problemei 3

```
        .data
sir:
        .word 2,1,7,9,5
        .text
```

```
        .globl main
main:
        li $a0, 0
tsgata:
        bnez   $a0, term
        li     $a0,1
        li     $a1, 5          # a1=n-1
        li     $a2, 0          # a2=j, indicele în sir
for:
        sub    $a2, $a2, $a1
        beqz   $a2, tsgata
        add    $a2, $a1, $a2
        la     $t0, sir
        li     $t1, 4
        mul    $a3, $a2, $t1
        add    $a3, $t0, $a3   # adr=sir+4*j
        lw     $t2, ($a3)      # sir[j]
        add    $t4, $a2, 1
        mul    $a3, $t4, $t1
        add    $a3, $t0, $a3
        lw     $t3, ($a3)      # sir[j+1]
        sub    $t2, $t2, $t3
        bgtz   $t2, schimba
        add    $a2, $a2, 1
        j      for
schimba:
        add    $t2, $t2, $t3
        add    $t5, $t2, $0
        add    $t2, $t3, $0    # t2=sir[j+1]
```

```
        add     $t3, $t5, $0            # t3=sir[j]
        sw      $t3, ($a3)
        sub     $a3, $a3, $t1
        sw      $t2, ($a3)
        li      $a0, 0
        j       for
term:
        done
```

Observatii:

1. O deficienta a simulatorului SPIM consta în imposibilitatea de a varia diferiti parametrii 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.

Bibliografie

[1] **Larus J.** - *SPIM S20: A MIPS R2000 Simulator*, Morgan Kaufmann Publishers, 1993

ARHITECTURA MICROPROCESOARELOR DLX

1. Scopul lucrarii

Lucrarea de fata prezinta arhitectura procesorului didactic virtual DLX, setul de instructiuni si modurile de adresare ale acestuia.

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 asemanatoare cu cea a DLX-ului.(de ex. MIPS, SPARC, etc.)

Setul de instructiuni al acestui microprocesor ca si al precursorilor lui se bazeaza pe o serie de consideratii generale privitoare la caracteristicile microprocesoarelor moderne actuale RISC cum ar fi:

- registrii de uz general cu o arhitectura LOAD-STORE
- suport pentru modurile de adresare :
    - indexat (cu un offset de adresa de 12 sau 16 biti)
    - imediat (operandul este o constanta pe 12 sau 16 biti)
    - registru (operandul se afla într-un regsitru general)
- suport pentru instructiuni simple de tip: load, store, add, substract, move registru, and, shift, compare equal, comapre not equal, branch (cu o adresa relativa la PC de dimensiune cel putin 8 biti), jump, call, return.
- suport pentru tipuri si date pe 8, 16, 32 pentru întregi si 64 biti conforme cu standardul IEEE 754 pentru virgula flotanta.

- în functie de optiune suport pentru codificare fixa a instructiunilor în caz ca se opteaza pentru performanta si codificare variabila daca se opteaza pentru cod redus.
- punerea la dispozitie a cel putin 16 registri de uz general (în general se folosesc pe 32 de biti) pe lînga o serie de registri separati pentru operatii în virgula flotanta precum si asigurarea ca toate modurile de adresare se pot aplica tuturor tipurilor de transfer de date
- urmarirea obtinerii unui set de instructiuni minimal.

În concordanta cu aceste criterii-cerinte, DLX-ul le adopta în felul urmator:

- set de instructiuni simplu de tip LOAD-STORE
- proiectat pentru o structura PIPELINE eficienta incluzând o codificare fixa a setului de instructiuni
- eficienta în rularea compilatoarelor

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

3. Desfasurarea lucrarii

Registrii microprocesorului DLX

DLX-ul are un set de 32 de registri generali (GPR) pe 32 de biti denumiti R0, R1,…R31. Aditional 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.

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 au 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. Operanzi 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.

Moduri de adresare la DLX

Singurile 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.

Formatul instructiunii la DLX

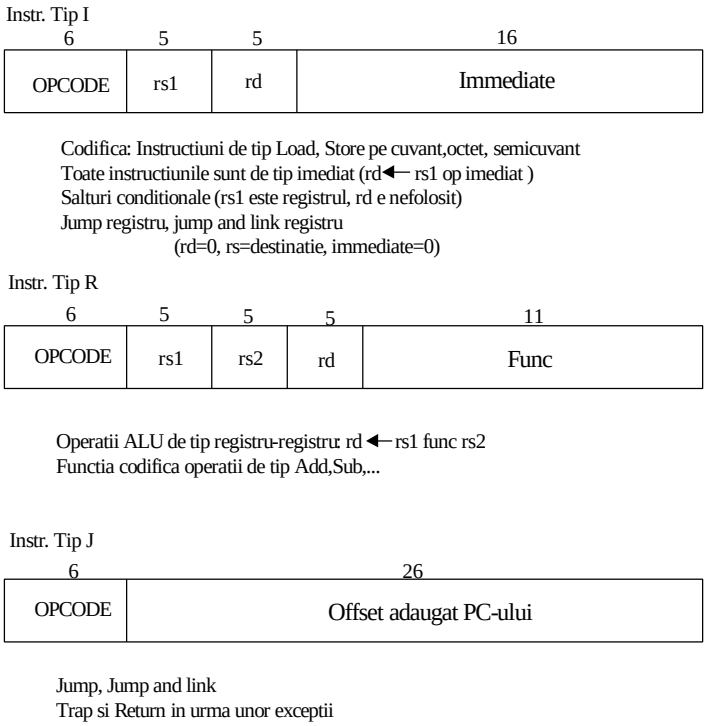


Figura 1. Formatul instructiunii

Formatul instructiunii este proiectat în asa fel încât sa asigure o decodificare optima si o functionare rapida a structurii pipeline. Setul de instructiuni este ortogonal pe 32 de biti cu un OPCODE primar de 6 biti. Acest format permite un

deplasament de 16 biti folosit ca index, constanta imediata sau adresa relativa (la PC) de salt.

Instructiunile DLX

DLX-ul suporta o lista de instructiuni simple mentionate anterior si în plus alte câteva. Exista patru clase de baza în care se pot împartii instructiunile: LOAD-STORE, ALU, salturi si instructiuni de comparare si instructiuni în virgula flotanta. Pentru toate aceste operatii oricare dintre cei 32 de registrii generali poate fi folosit cu specificarea faptului ca încarcarea lui R0 cu orice valoare ramâne fara efect. Valorile în virgula flotanta simpla 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	Regs[R1]$\leftarrow_{32}$ Mem[30+Regs[R2]]
LW R1,1000(R0)	Load word	Regs[R1]$\leftarrow_{32}$ Mem[1000+0]
LB R1,40(R3)	Load byte	Regs[R1]$\leftarrow_{32}$ (Mem[40+Regs[R3]]$_0$)^{24} ## Mem[40+Regs[R3]]
LBU R1,40(R3)	Load byte unsigned	Regs[R1]$\leftarrow_{32}$ 0^{24} ## Mem[40+Regs[R3]]
LH R1,40(R3)	Load half word	Regs[R1]$\leftarrow_{32}$ (Mem[40+Regs[R3]]$_0$)^{16} ## Mem[40+Regs[R3]]##Mem[41+Regs[R3]]
LF F0,50(R3)	Load float	Regs[F0]$\leftarrow_{32}$ Mem[50+Regs[R3]]
LD F0,50(R2)	Load double	Regs[F0]##Regs[F1]$\leftarrow_{64}$ Mem[50+Regs[R2]]
SW 500(R4),R3	Store word	Mem[500+Regs[R4]]$\leftarrow_{32}$ Regs[R3]
SF 40(R3),F0	Store float	Mem[40+Regs[R3]]$\leftarrow_{32}$ Regs[F0]
SD 40(R3),F0	Store double	Mem[40+Regs[R3]]$\leftarrow_{32}$ Regs[F0]; Mem[44+Regs[R3]]$\leftarrow_{32}$ Regs[F1]
SH 502(R2),R3	Store half	Mem[502+Regs[R2]]$\leftarrow_{16}$ Regs[R31]$_{16..31}$
SB 41(R3),R2	Store byte	Mem[41+Regs[R3]]$\leftarrow_{8}$ Regs[R2]$_{24..31}$

Figura 2. Instructiunile de tip Load-Store la DLX

Pentru o întelegere mai buna a notatiilor din tabelul anterior sa consideram exemplul urmator:

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

Semnificatia acestuia este: octetul de la adresa de memorie specificata de continutul lui R8 cu o extensie de semn la 16 biti si este mai apoi încarcat în jumatatea inferioara a lui R10 (jumatatea 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 jumatatea 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 plaseaza 0.

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

Figura 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 de 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 plaseaza 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 4 se exemplifica acest tip de instructiuni.

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

Figura 4. Instructiuni de salt si ramificatie

Instruction type/opcode	Instruction meaning
Data transfers	**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**
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, MOVS2I	Move from/to GPR to/from a special register
MOVF, MOVD	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
Arithmetic/logical	**Operations on integer or logical data in GPRs; signed arithmetic trap on overflow**
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
Control	**Conditional branches and jumps; PC-relative or through register**
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
Floating point	**FP operations on DP and SP formats**
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 5. Setul complet de instructiuni DLX

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 întelegem 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 ciclii. 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.

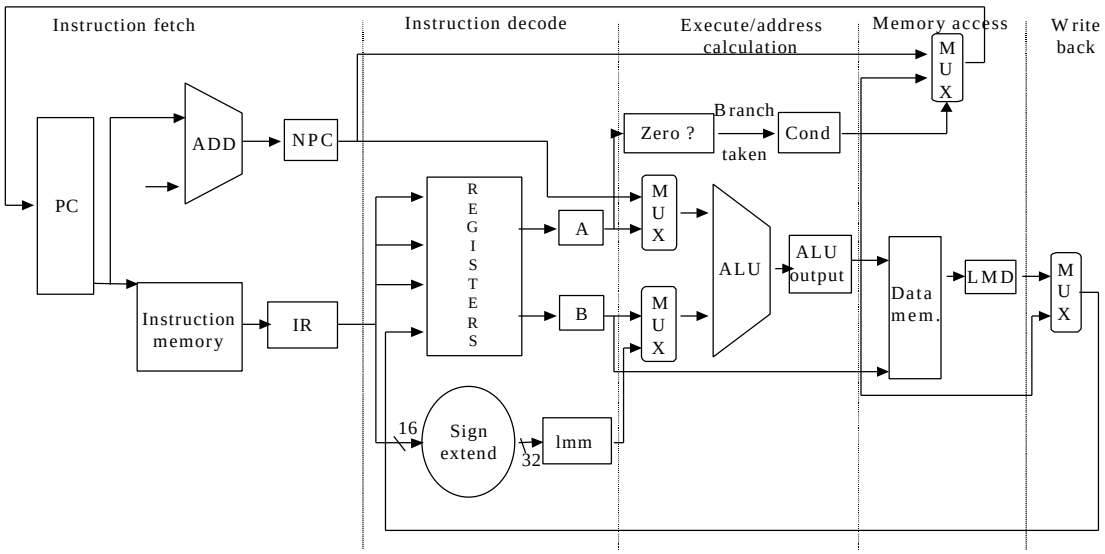


Figura 6. Implementarea simpla DLX

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

- **Instruction Fetch (IF)**

IR ← Mem[PC]

NextPC ← PC+4

Obtine PC-ul si aduce instructiunea respectiva din memorie în registrul IR; incrementeaza PC-ul cu 4, acesta continând astfel adresa urmatoarei instructiuni din secventa de program. Registrul IR contine instructiunea curenta iar NPC contine PC-ul urmator.

- **Instruction Decode (ID)**

A ← Regs[$IR_{6..10}$];

B ← Regs[$IR_{11..15}$];

Imm ← ((IR_{16})^{16}## $IR_{16..31}$)

Decodifica instructiunea si acceseaza setul de registri pentru a citi continutul acestora. Acestia sunt cititi temporar într-un grup de doi registrii temporari A si B pentru a putea fi folositi ulterior în cadrul ciclilor ulteriori. Celor mai putin semnificativi 16 biti ai registrului IR li se extinde semnul si 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:

ALUOutput ← A+Imm;

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

- Instructiune ALU de tip Registru-Registru:

ALUOutput ← A 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:

ALUOutput ← A op Imm;

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

- Branch:

ALUOuput ← NPC+Imm;
Cond ← (A op 0)

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 ciclii 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 aceeasi categorie sunt incluse si instructiunile de tip Jump.

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

În acest caz avem trei tipuri de instructiuni:

- Referinte la memorie:

LMD ← Mem[ALUOutput] sau
Mem[ALUOutput] ← B;

Daca instructiunea este un load data este citita din memorie în registrul LMD (Load Memory Register) iar daca este o instructiune de tip store atunci data din registrul B este scrisa în memorie.

- Branch:

If (cond)
    PC ← ALUOutput
else
    PC ← NPC

Daca saltul se face PC-ul este înlocuit cu adresa de salt aflata în registrul ALUOutput iar daca saltul nu se face el este înlocuit cu PC-ul incrementat aflat în registrul NPC.

- **Write Back (WB)**

- Instructiune ALU registru-registru:

$Regs[IR_{16..20}] \leftarrow ALUOutput;$

- Instructiune ALU registru –imediat:

$Regs[IR_{11.15}] \leftarrow ALUOutput;$

- Instructiuni de tip Load:

$$Regs[IR_{11.15}] \leftarrow LMD;$$

Scrie rezultatul în setul de registri fie ca acesta provine din memorie sau din iesirile ALU. Se observa din figura 6 ca la sfârsitul fiecarui ciclu, orice valoare calculata si necesara într-un ciclu ulterior, se memoreaza într-o locatie care poate fii 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 ciclii, celelalte desfasurându-se de-a lungul acinci ciclii masina.

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

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. 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 instructiuni) 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:

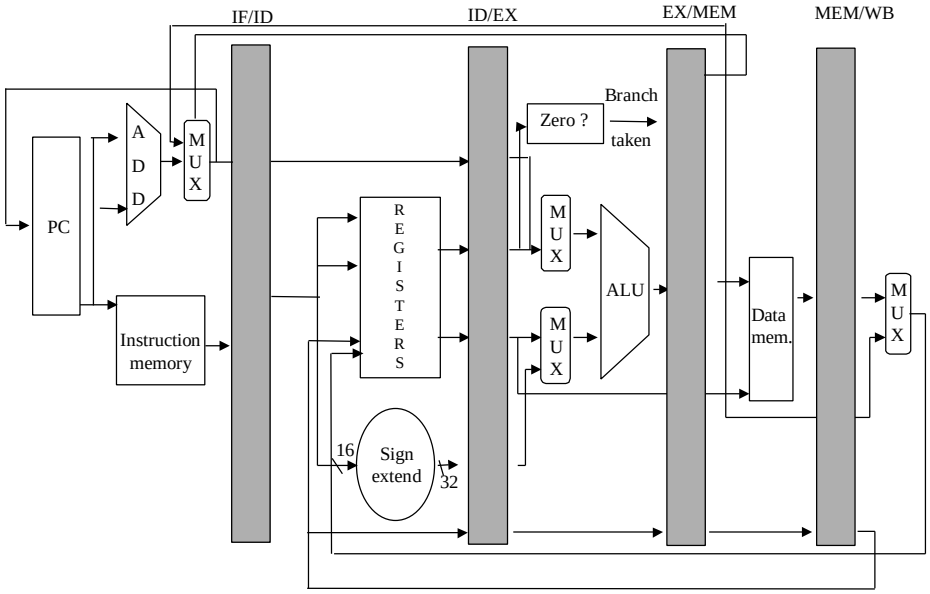


Figura 7. Implementarea DLX cu structura pipeline

Prin registrii pipeline trec atât date cât si informatii de control al fluxului structurii pipeline. Aceste informatii trebuie sa fie transferate de la un nivel la altul al structurii pipeline pâna în momentul în care vor fi folositi. Nu se pot folosi doar registrii temporari datorita faptului ca se pot suprascrie valori care nu au fost folosite înca. De exemplu, câmpul unui registru necesar unei scrieri pe parcursul unei operatii LOAD sau ALU, este luat din latch-ul MEM/WB si nu din IF/ID. Aceasta deoarece se doreste ca operatia respectiva sa scrie registrul destinatie desemnat de operatia respectiva si nu unul din registrii care tranziteaza de la IF la ID. Acesta din urma e copiat dintr-un latch în altul pâna în faza WB unde e folosit. De remarcat este faptul ca primele doua etape din structura pipeline sunt independente de tipul de instructiune deoarece orice instructiune este decodificata doar la finele ciclului ID. Activitatea ciclului IF depinde daca instructiunea este un salt care se face sau care nu se face. Daca saltul se face atunci adresa de salt este folosita ca PC si pentru a calcula adresa urmatoare. Pentru a controla fluxul datelor în schema de mai sus e necesar sa cunoastem functionarea celor patru MUX-uri din figura 7.

Cele doua MUX-uri de la intrarile ALU sunt utilizate în functie de tipul instructiunii care este dat de câmpul IR din registrul ID/EX. MUX-ul de la iesirea sumatorului, înaintea registrului IF/ID este utilizat daca instructiunea este un salt. Astfel daca instructiunea este un salt, PC-ul este încarcat cu noua valoare a adresei de salt în cazul ca acesta se face. Noua adresa se obtine pe baza fazei EX/MEM. Multiplexorul alege daca sa foloseasca PC-ul curent sau valoarea din EX/MEM NPC(vezi registrul **NPC** figura 6). Acest multiplexor este controlat de catre registrul EX/MEM cond (registrul ***Cond*** în figura 6). Cel de-al patrulea multiplexor este setat în functie de tipul instructiunii din ciclul WB: ALU sau LOAD.

Se observa de asemenea necesitatea unui al cincilea multiplexor care sa aleaga portiunea corecta a registrului IR în latch-ul MEM/WB pentru a specifica

registrul destinatie. Acesta poate avea doua variante: registru-registru ALU respectiv ALU-imediat sau LOAD.

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

Figura 8. Nivelele pipeline DLX

Bibliografie

[1] **Hennessy J., Patterson D.** - *Computer Organization and Design*: *The Hardware / Software Interface*, Morgan Kaufmann Publishers, San Francisco, California, 1994

[2] **Gründbacher H.** - *Windows Deluxe Simulator - Tutorial*, University of Technology, Viena, 1992

UTILIZAREA SIMULATORULUI DLX

1. Scopul lucrarii

Scopul lucrarii este de a ajuta la întelegerea conceptelor legate de procesarea pipeline a instructiunilor precum si a aspectelor arhitecturale specifice procesoarelor RISC. Lucrarea de fata prezinta o descriere, nu foarte detaliata, a simulatorului DLX, un simulator scris pentru sistemul Windows. Pentru a utiliza simulatorul sunt necesare cunostinte minime de lucru în Windows (minim Windows 3.0), cum ar fi: startarea aplicatiei, lucrul cu ferestre (minimizare, maximizare), defilarea într-o fereastra, activarea sau aducerea unei ferestre în "*foreground*", etc.

2. Memento teoretic

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. 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 instrutiune) 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 copiata în registrul destinatie corespunzator.

Modul de lucru al simulatorului este urmatorul: dupa încarcarea 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 sunt programe de calcul cu numere întregi, scrise în asamblare. 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.

3. Desfasurarea lucrarii

3.1. Pornirea si configurarea WinDLX

Pornirea WinDLX se face, ca orice aplicatie Windows, printr-un dublu click pe iconita fisierului executabil WinDLX. Pe ecran vor aparea o fereastra principala si sase ferestre copil minimizate. Pentru reinitializarea procesorului se alege optiunea *Reset all* din meniul *File*. Pe ecran va aparea fereastra "ResetDLX" si intentia de resetare a procesorului trebuie cofirmata printr-un click pe butonul OK.

WinDLX este capabil sa lucreze cu câteva configuratii. Salvarea configuratiei poate fi facuta la sfârsitul 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 simularii. Pentru alegerea configuratiei standard avem de parcurs urmatorii pasi: din meniul *Configuration* selectam optiunea *Floating Point Stages* si verificam existenta urmatoarelor setari:

	Numar	Cicli Întârziere
Unitate de adunare	1	2
Unitate de înmultire	1	5
Unitate de împartire	1	19

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, de exemplu, ciclu 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, rebuclând la o intrare, iesirea unei unitati functionale sau, la procesoarele superscalare folosind statii de rezervare. [1]

3.2. Încarcarea programelor de test

Pentru a putea starta simularea, cel putin un program trebuie încarcat în memoria principala. Astfel, se selecteaza optiunea *Load Code or Data* din meniul *File*. Va aparea o fereastra de dialog din care se pot selecta un numar arbitrar de module cu extensia "*.s*", reprezentând coduri sursa DLX. Daca este confirmata selectia prin apasarea butonului *Load*, toate modulele selectate vor fi asamblate si codul este scris în memoria DLX. Erorile raportate vor apare într-o fereastra de dialog separata iar datele scrise în memorie vor fi invalide. Dupa o încarcare cu succes putem reseta procesorul DLX prin intermediul unei ferestre de dialog.

Daca s-a selectat un fisier nedorit, se insereaza respectivul fisier si se apasa butonul *Delete*. Daca nu se selecteaza nimic, atunci nici un fisier nu va fi încarcat în memorie.

Posibilitatea de încarcare multipla a modulelor în acelasi timp permite definirea simbolurilor globale care pot fi folosite în mai multe module. Ulterior, va fi posibil sa încarcam module fara a tine cont de ordinea de încarcare.

Pentru exemplificare vom considera benchmark-ul *fact.s* si rutina necesara *input.s*.

- se executa click pe **fact.s**.
- se apasa butonul *Select*.
- se executa click pe **input.s**.
- se apasa butonul *Select*.
- se apasa butonul *Load*.

Dupa executia acestor pasi simularea poate începe.

3.3. Simularea propriu-zisa

3.3.1. Fereastra Pipeline

În aceasta fereastra sunt descrise cele 5 nivele pipeline ale executiei instructiunilor.

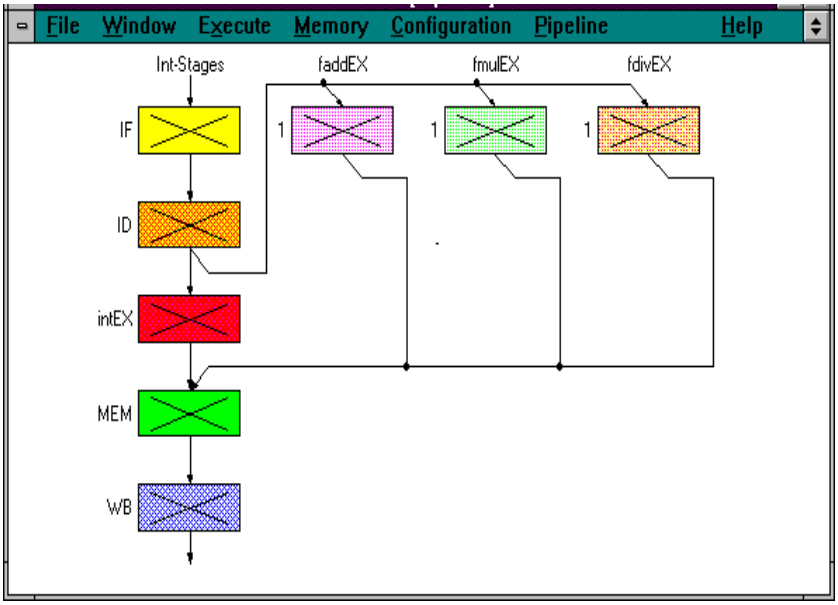


Figura 1. Fereastra pipeline

La nivelul **IF** în fiecare ciclu de tact o noua instructiune este extrasa din memorie si depusa în registrul instructiunii. PC este incrementat pentru a pointa la urmatoarea instructiune. La nivelul **ID** are loc executia branch-urilor pentru a reduce stagnarile. La nivelul **IntEX** au loc operatii aritmetice (mai putin înmultire si împartire) si calcul efectiv al adresei pentru instructiunile cu referire la memorie. În **FaddEX** se executa operatii de adunare si scadere în simpla sau dubla precizie. Unitatile **FmulEX** si **FdivEX** executa operatii de înmultire si împartire atât asupra întregilor (cu semn si fara semn) cât si asupra flotantilor în simpla si dubla precizie. Singurele instructiuni ale DLX active la nivelul **MEM** sunt LOAD si STORE. Calculul adresei de citire sau scriere s-a efectuat în nivelul anterior. Rezultatul este scris în registrii în nivelul **WB**. El provine fie

din memorie fie dintr-o operatie ALU. Operatia de scriere se face în prima jumatate a ciclului de ceas, astfel încât o instructiune aflata pe nivelul ID sa poata citi rezultatul în a doua jumatate a ciclului de ceas, eliminând necesitatea de bypass-ing a rezultatului instructiunii.

3.3.2. Fereastra Cod

Instructiunile DLX încarcate în memorie si adresele lor sunt afisate (în hexazecimal) si 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 unuia 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 ferestra cu adresa $TEXT este colorata galben. Apasând F7 vom avansa în simulare cu un pas. Aceasta va determina schimbarea culorii primei linii în orange 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.

3.3.3. Diagrama ciclurilor 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 instructiune selectata vom obtine mai multe detalii despre instructiunea respectiva, în fiecare ciclu de tact (instructiunea dezasamblata, adresa ei, codificarea în hexazecimal, starea de executie - nivelul pipeline, modul în care s-a încheiat - normal, cu eroare, întrerupt, numarul ciclului când a început executia, numarul de cicli de executie, etc).

Meniul Diagrama ciclului de tact contine urmatoarele optiuni:

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

În continuare vom prezenta detalii privind stagnari sau forwarding în executie. Stagnarile sunt reprezentate prin chenare colorate în culoarea corespondenta nivelului pipeline în care se stagneaza. Distingem urmatoarele tipuri de stagnari:

- **R-Stall** - stagnare datorata unui hazard RAW. O sageata rosie închis va indica instructiunea care cauzeaza stagnarea (dupa al carei operand se asteapta).
- **T-Stall** - stagnare datorata unei exceptii Trap. O instructiune Trap ramâne în nivelul IF, pâna când nici o alta instructiune nu se mai afla în pipe.
- **W-Stall** - stagnare datorata unui hazard WAW. Dependenta dintre instructiunile implicate este sugerata tot printr-o sageata de culoare rosu închis.
- **S-Stall** - hazard structural. Nu exista suficiente resurse pentru deservirea instructiunilor.
- **Stall** - daca o instructiune în flotant se afla în nivelul MEM, urmatoarea instructiune va fi oprita în nivelul intEX, etichetat cu "Stall", pâna la extragerea datelor necesare din memorie.

Desi este posibila aparitia oricarui tip de hazard mentionat 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 executie pentru fiecare nivel al pipe-ului. Totusi, pentru programe de test care utilizeaza succesiv mai multe instructiuni de (adunare/scadere, înmultire sau împartire) în flotant pot apare hazarduri structurale pentru anumite configuratii arhitecturale.

Exemplu:

Consideram urmatoarea configuratie arhitecturala, fara mecanism de forwarding:

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

si urmatoarea secventa de instructiuni:

*i*1: *multd f2, f2, f0*
*i*2: *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 înmultire.

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:

*i*1: *ins_float R2, R2, R0 se executa în 30 ciclii*
*i*2: *ins_integer R2, R4, R6 se executa în 5 ciclii*

Instructiunea pe întregi (*i*1) se încheie mai repede decât cea în flotant (*i*2) 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 întelegere mai buna, vom exemplifica în continuare pe diagrama ciclului de tact de mai jos.

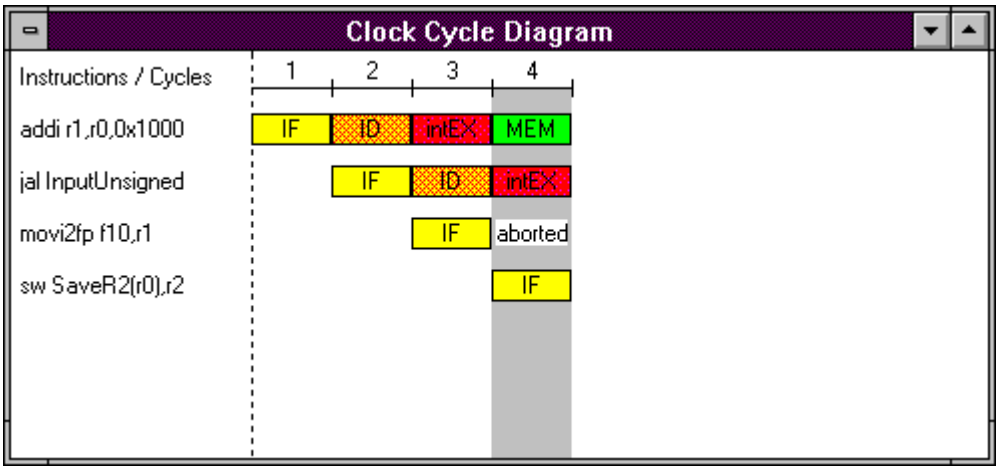


Figura 2. Diagrama ciclului de tact

În diagrama alaturata observam ca simularea este în al patru-lea 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 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.

3.3.4. 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:

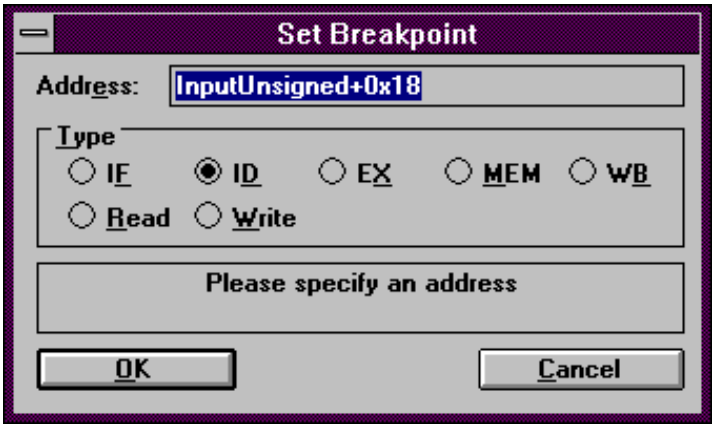


Figura 3. Fereastra Breakpoint

Când examinam fereastra cod observam instructiuni similare care se repeta. Astfel, parcurgerea pas cu pas a codului sursa al benchmark-ului devine plictisitor si ineficient. 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 3. Tipul implicit al breakpoint-ului este ID. Se confirma cu butonul OK.

În consecinta, pe linia din fereastra de cod pe care se afla instructiunea **trap 0x5**, va aparea expresia "BID", semnificând ca programul se întrerupe când instructiunea respectiva este în faza de decodificare. Pentru examinarea punctelor de întrerupere deja definite se executa click pe iconita *Breakpoint*. Declansarea simularii se face selectând optiunea *Run* din meniul *Execute* sau tasta F5. O fereastra ne va informa ca ne aflam pe nivelul ID si s-a atins primul

punct de întrerupere. Daca studiem diagrama ciclului de ceas vom observa ca simularea este în ciclul 14, iar instructiunea **trap 0x5** se afla în stagnare.

Motivatia consta în faptul ca nivelele pipeline ale DLX sunt golite de fiecare data când se întâlneste o instructiune trap, pentru a evita toate posibilitatile problemei. Daca vom activa fereastra *Display DLX-IO* din meniul *Execute*, în fereastra va aparea mesajul: " An integer value:".

3.3.5. Fereastra Registru

În aceasta fereastra sunt afisate valorile tuturor registrilor. Este posibila totodata si modificarea continutului registrilor generali - GPR, a registrilor flotanti - FPR, a PC-ului si registrului de stare - FPSR, dar nu si continutul registrilor speciali. Distingem mai multe tipuri de afisare si reprezentare a continutului registrilor: hexadecimal, decimal, flotant simpla si dubla precizie. Selectia unui registru se face prin dublu click pe numele acestuia.

Procesorul DLX are urmatorii registri vizibili în programe:

- GPR (R0..R31) - 32 registri de uz general pe 32 de biti; R0 (ca si la procesorul MIPS R2000) este întotdeauna cablat la 0.
- FPR - set de registri flotanti care pot fi folositi ca 32 de registri în simpla precizie (F0 ÷ F31), sau perechi de registri par-impar în dubla precizie (D0 to D30).
- FPSR - registru de stare folosit atât pentru comparatii cât si pentru exceptii în virgula flotanta. Toate instructiunile de transfer în/din registrul de stare lucreaza cu registrii de uz generali. Instructiunile de salt conditionat testeaza bitul de comparatie din registrul FPSR.

- PC – program counter-ul contine întotdeauna adresa urmatoarei instructiuni ce urmeaza a fi extrasa din memorie. Instructiunile de ramificatie (salturi conditionate sau neconditionate) pot altera continutul PC.

De asemenea, DLX contine si registrii 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 registrii sunt cititi si trimisi unitatii arimetico-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 tintei 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, pseudoregistrul 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←(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 activand 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 instructiuni subd sa încheie faza MEM.

3.3.6. 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", aratând ca instructiunea **trap 0** a fost executata. Instructiunea **trap 0** nu este definita, ea fiind folosita 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**.

Concluzii

Lucrarea prezinta caracteristicile importante ale procesorului DLX, urmarind întelegerea 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.

Bibliografie

[1] **Hennessy J., Patterson D.** - *Computer Organization and Design*: *The Hardware / Software Interface*, Morgan Kaufmann Publishers, San Francisco, California, 1994

[2] **Gründbacher H. -** *Windows Deluxe Simulator - Tutorial*, University of Technology, Viena, 1992

INVESTIGATII ARHITECTURALE UTILIZÂND SIMULATORUL DLX

1. Scopul lucrarii

Lucrarea de fata urmareste studierea programelor de test, deprinderi practice privind programarea în limbajul de asamblare al procesorului DLX, precum si o analiza cantitativa si calitativa a rezultatelor obtinute în urma simularii efectuate pe benchmark-urile existente.

2. Desfasurarea lucrarii

2.1. 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**).

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 prelucrarii 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 facând 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 [1].

Serviciul	Codul 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 de executie	Deschiderea unui fisier pentru citire sau scriere. Fisierele deschise sunt automat închise la resetarea DLX sau oprirea WinDLX. Descriptorul de fisier este 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 citeste blocul.	Un bloc dintr-un fisier sau o linie de la intrare (stdin)

		2. Adresa zonei unde se va stoca blocul citit. 3. Dimensiunea blocului ce va fi citit (în octeti)	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.

Detalii suplimentare legate de modurile de deschidere a fisierelor, drepturilor de acces si executie asupra acestora pot fi gasite studiind functiile de biblioteca C. [2]

2.2. Note explicative referitoare la benchmark-ul Fact.s

Benchmark-ul *Fact.s* este un program de test simplu, care calculeaza si afiseaza pe ecran factorialul unui numar, introdus de la tastatura cu ajutorul modulului *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 simularii.

```
;**** WINDLX : Calculul Factorialului unui numar ****
;-------------------------------------------------------------------------------------------
;Programul necesita subrutina INPUT.s
;Se citeste un numar de la tastatura si calculeaza factorialul sau
;(tipul rezultatului: double)
; Rezultatul este afisat pe consola
;-------------------------------------------------------------------------------------------

            .data
                        ;incepe segmentul de date - valoare implicita 0x1000
Prompt:     .asciiz     "An integer value >1 : "
                        ;0x1000 - adresa de inceput a segmentului de date
                        ;memoreaza un string de 0x16 octeti

PrintfFormat: .asciiz   "Factorial = %g\n\n"
                        ;0x1017 - adresa la care se memoreaza un sir de 0x10
                        ;octeti
            .align 2
                        ;datele vor fi memorate pe cuvant
PrintfPar:  .word PrintfFormat
                        ;0x1028 - adresa la care se memoreaza un cuvant pe 4
                        ;octeti (adresa 0x1017)
PrintfValue: .space8
                        ;0x102C - se aloca 8 octeti necesari stocarii rezultatului
```

```
        .text
                ;segmentul de cod incepe la simbolul main - valoare
                ;implicita 0x100
        .global main
main:
;*** Se apeleaza procedura de intrare pentru citirea numarului de la tastatura ***

        addi   r1,r0,Prompt     ;r1<-adresa mesaj de la eticheta <Prompt>
        jal InputUnsigned       ;apel subrutina Input.s unde va avea loc
                                ;citirea efectiva a numarului de la tastatura

;*** Initializare valori ***
;*** La iesirea din procedura Input.s numarul citit se afla in registrul r1 ***

        movi2fp f10,r1          ;r1 -> d0 d0 - folosit pe post de registru contor
        cvti2d f0,f10           ;converteste valoarea intreaga din f10 in dubla
                                ;precizie si o depune in f0
        addi r2,r0,1            ;1 -> d2 d2 - pastreaza rezultatul. Initializat cu 1.
        movi2fp f11,r2          ;r2 -> f11
        cvti2d f2,f11           ;converteste valoare intreaga din f11 in dubla
                                ;precizie in f2
        movd  f4,f2             ;1  -> d4 d4 - constanta 1

;*** Iesirea din bucla se face daca d0 = 1 ***

Loop:   led f0,f4               ;se testeaza daca d0<=1
        bfpt Finish             ;salt la eticheta Finish daca d0<=1

;*** Inmultire si reapelare bucla ***
        multd f2,f2,f0          ;rezultat <- rezultat*contor [f2 <- (f2*f0)]
```

```
            subd f0,f0,f4       ;decrementare contor cu constanta 1 [f0<- (f0-1)]
            j Loop              ;rebuclare

Finish:     ;*** Afisare rezultat pe consola ***
            sd PrintfValue,f2   ;memoreaza rezultat (dubla precizie) la adresa
                                ;PrintfValue
            addi r14,r0,PrintfPar   ;pregatire adresa pentru afisare mesaj +
                                    ;rezultat
            trap 5              ;intrerupere software ce are ca efect afisarea pe
                                ;ecran a continutului locatiei de memorie de la
                                ;adresa data de registrul r14

;*** Incheiere program ***
            trap 0              ;intrerupere fara efect asupra DLX (procesor +
                                ;memorie) care anunta incheierea executiei
                                ;programului
```

Comentariul subrutinei Input.s

```
;*************** WINDLX: Citirea unui numar întreg pozitiv ***************
;-------------------------------------------------------------------------------------------
;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: .space80
                        ;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:  .space4
                        ;0x1090
SaveR2:     .space4
                        ;0x1094 – 4 octeti necesari pentru memorarea lui R2
SaveR3:     .space4
                        ;0x1098 – 4 octeti necesari pentru memorarea lui R3
SaveR4:     .space4
                        ;0x109C – 4 octeti necesari pentru memorarea lui R4
SaveR5:     .space4
                        ;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
```

3. Probleme propuse spre rezolvare

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

II. Scrieti un program care citeste *n* numere de la tastatura prin intermediul modulului **Input.s** si calculeaza suma numerelor si le depune succesiv în memoria DLX la adresa 0x1500.

III. Numim CONFIGURATIE DE BAZA a procesorului RISC DLX, urmatoarea configuratie arhitecturala:

1 unitate executie numere întregi, latenta 1 ciclu;
1 unitate executie adunare flotant (FPP ADD), latenta 2 cicli;
1 unitate executie înmultire flotant (FPP MUL), latenta 5 cicli;
1 unitate executie împartire flotant (FPP DIV), latenta 19 cicli;
Nu e implementata tehnica "forwarding".

În acest context, în continuare, se cere:

1. Rata de procesare (IR, în instructiuni/ciclu), masurata pe benchmark-ul dat, pentru configuratia de baza DLX.

2. Cât devine rata de procesare (comparativ cu cazul precedent), daca se considera 2 respectiv 8 unitati de executie FPP ADD (în rest, sunt pastrate caracteristicile configuratiei de baza).

3. IR, daca se considera 2 respectiv 8 unitati de executie FPP MUL (în rest, configuratia de baza).

4. IR, daca se considera 2 respectiv 8 unitati de executie FPP DIV (în rest, configuratia de baza).

5. IR, daca fata de configuratia de baza se activeaza optiunea "forwarding".

6. IR, pentru variatia latentei unitatii de executie FPP ADD de la 1 la 5 cicli.

7. IR, pentru latenta unitatii FPP MUL de 1 respectiv 20 cicli.

8. IR, pentru latenta unitatii FPP DIV de 1 respectiv 32 cicli.

Observatii

1. Rezultatele obtinute la punctele III.2, III.3, III.4 sunt identice cu cel obtinut la punctul III.1. Explicatia o regasim daca privim în fereastra Statistica, si observam ca nu exista stagnari 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).

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.

Bibliografie

[1] **Gründbacher H. -** *Windows Deluxe Simulator - Tutorial*, University of Technology, Viena, 1992.

[2] **Negrescu L.** – *Introducere în limbajul C* - Editura MicroInformatica, Cluj Napoca., 1993

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.

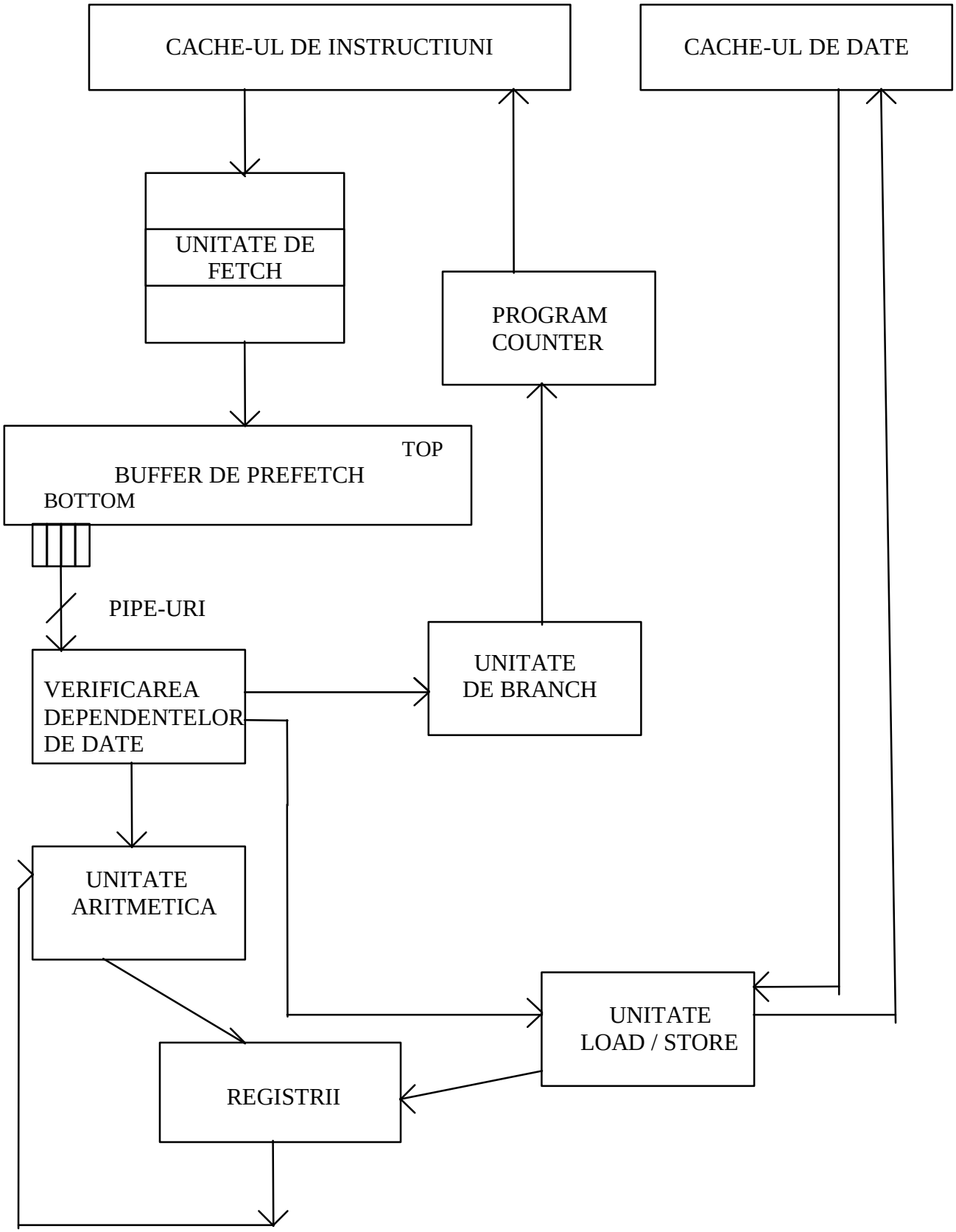


Figura 1. Schema bloc a arhitecturii superscalare simulate

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.

OPTIMIZAREA SCHEMELOR DE PREDICTIE PENTRU RAMIFICATIILE DE PROGRAM ÎN PROCESOARELE SUPERSCALARE AVANSATE

1. Scopul lucrarii

Lucrarea de fata insista pe investigarea unor arhitecturi moderne de predictoare a ramificatiilor de program (**branch**), în vederea reducerii penalitatilor introduse de instructiunile de ramificatie în procesoarele pipeline superscalare. Vor fi explorate în mod critic metodologiile de predictie existente, vor fi îmbunatatite, optimizate si stabilite limitarile fundamentale. Se vor stabili schemele de predictie optimale asociate diferitelor tipuri de ramificatii din program.

2. Memento teoretic

Predictia prin hardware reprezinta una din cele mai performante strategii actuale de gestionare a ramificatiilor de program. Aceste strategii 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. Cercetari recente insista pe aceasta problema, întrucât s-ar elimina necesitatea reorganizarilor 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, superscalare) este imperios necesara. Notând cu BP (Branch Penalty) numarul mediu de cicli de asteptare pentru fiecare instructiune din program, introdusi de salturile fals predictionate, se poate scrie relatia:

$$BP = C\,(1-Ap)\,b\,IR \qquad (1)$$

unde s-au notat prin:

C= Numarul de cicli de penalizare introdusi de un salt prost predictionat

Ap= Acuratetea predictiei

b= Procentajul instructiunilor de salt, din totalul instructiunilor, procesate în program

IR= Rata medie de lansare în executie a instructiunilor

Se observa ca BP(Ap=0)=C b IR, iar BP(Ap=1)=0 (normal, predictia este ideala aici). Impunând un BP=0.1 si considerând valorile tipice: C=5, IR=4, b=22.5%, rezulta ca fiind necesara o acuratete a predictiei de peste 97.7% ! Cu alte cuvinte, la o acuratete de 97.7%, IR=4/1.4=2.8 instr./ciclu, fata de IR=4 instr./ciclu, la o predictie perfecta (Ap=100%). 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 si va declansa aducerea instructiunii urmatoare de pe ramura prezisa. Daca predictia se dovedeste a fi fost falsa structura pipeline se evacueaza si se va initia procesarea celeilale ramuri de program. Totodata, valoarea bitului de predictie din BPB se inverseaza.

Dezavantajul schemei BPB cu un singur bit se manifesta îndeosebi în cazul buclelor de program. Bazat pe tehnica BPB în acest caz vom avea uzual 2 predictii false: una la intrarea în bucla (prima parcurgere) si alta la iesirea din bucla (ultima parcurgere a buclei). Astfel, considerând o bucla de program care va fi executata de

N ori, acuratetea predictiei va fi de (N - 2) * 100 / N %, iar saltul se face în proportie de (N - 1) * 100 / N %.

Pentru a elimina acest dezavantaj se utilizeaza 2 biti de predictie modificabili conform grafului de tranzitie de mai jos (numarator saturat). În acest caz acuratetea predictiei unei bucle care se face de (N - 1) ori va fi (N - 1) * 100 / N %.

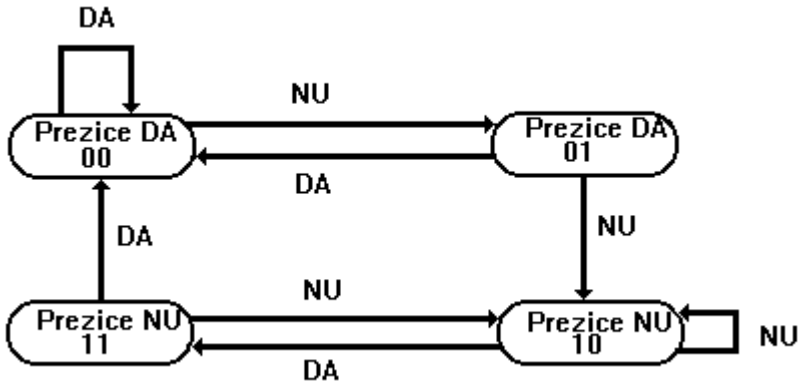


Figura 1. Automat de predictie de tip numarator saturat pe 2 biti

Prin urmare, în cazul în care se prezice ca branch-ul se va face, aducerea noii instructiuni se face de îndata ce continutul noului PC e cunoscut. În cazul unei predictii incorecte, se evacueaza structura pipeline si se ataca cealalta ramura a instructiunii de salt. Totodata, bitii de predictie se modifica în conformitate cu graful din figura numit si numarator saturat (vezi Fig.1).

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.

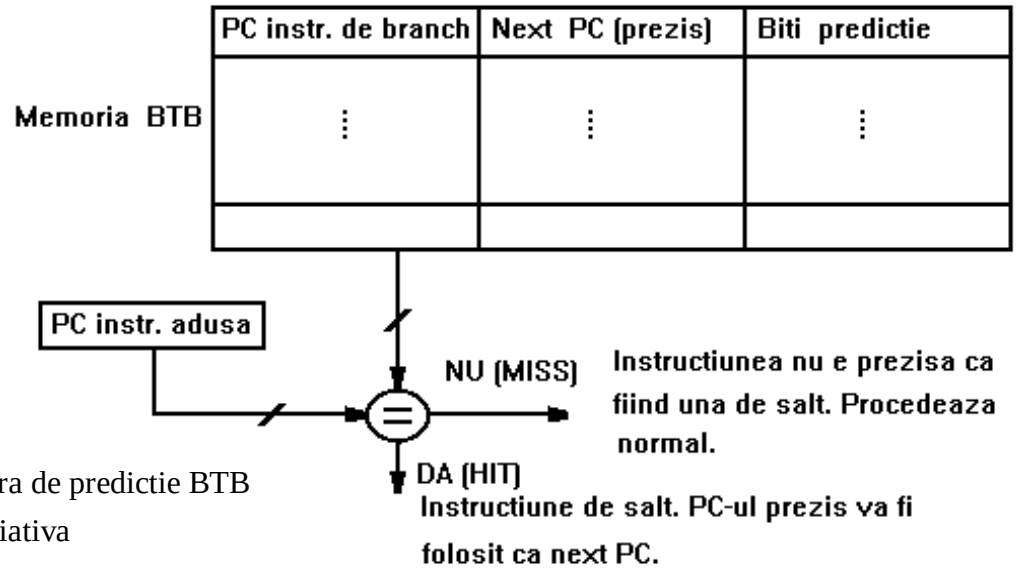


Figura 2. Structura de predictie BTB asociativa

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) Daca instructiunea adusa e o instructiune de branch, se trece în pasul EX1, altfel se continua procesarea normala.

RD2) Se trimite PC-ul prezis din BTB spre memoria de instructiuni. În cazul în care conditiile de salt sunt satisfacute, se trece în pasul EX 3, altfel în pasul EX2.

EX1) Se introduce PC-ul instructiunii de salt precum si PC-ul prezis în BTB. De obicei aceasta alocare se face în locatia cea mai de demult neaccesata (Least Recently Used- LRU).

EX2) Predictia s-a dovedit eronata. Trebuie reluata faza IF de pe cealalta ramura cu penalizarile de rigoare datorate evacuarii structurilor pipeline.

EX3) Predictia a fost corecta, însa numai daca si PC-ul predictionat este într-adevar corect, adica neschimbat.În acest caz, se continua executia normala.

În tabelul urmator (tabelul 1) sunt rezumate avantajele si dezavantajele tehnicii BTB, anterior descrise.

Tabelul 1. Penalizarea într-o structura de predictie tip BTB

Instr. în BTB ?	Predictie	Realitate	Cicli penalizare
Da	Da	Da	0(Ctt)
Da	Da	Nu	Ctn
Da	Nu	Nu	0
Da	Nu	Da	Cnt
Nu	-	Da	Ct
Nu	-	Nu	0

În baza celor prezentate anterior, rezulta ca numarul de cicli de penalizare CP este dat de urmatoarea relatie:

$$CP = P_{BTB}\,(P_{tn}*C_{tn} + P_{nt}*C_{nt}) + (1-P_{BTB})*P*C_t \qquad (2)$$

unde s-a notat:

P_{BTB} - probabilitatea ca instructiunea de salt sa se afle în BTB;

$\mathbf{P_{tn}}$ - probabilitatea ca saltul sa fie prezis ca se face si în realitate nu se va face;

$\mathbf{C_{tn}}$ - reprezinta ciclii de penalizare corespunzator situatiei.

$\mathbf{P_{nt}}$ - probabilitatea ca saltul sa fie prezis ca nu se face si în realitate se va face;

$\mathbf{C_{nt}}$ - reprezinta ciclii de penalizare în aceasta situatie.

P - probabilitatea ca respectiva instructiune de salt sa se faca. $\mathbf{C_t}$ – reprezinta ciclii de penalizare daca saltul se face si acesta nu este în BTB.

În acest caz, rata de procesare a instructiunilor ar fi data de relatia:

$$\mathbf{IR = 1/(1 + P_b * CP), [instr./tact]} \qquad \mathbf{(3)}$$

unde $\mathbf{P_b}$ = probabilitatea ca instructiunea curenta sa fie una de ramificatie.

Observatii:

1) BTB nu îmbunatateste performanta pentru o predictie corecta de tipul "saltul nu se face" (P_{nn}= 0), întrucât în acest caz structura se comporta în mod implict la fel ca si o structura fara BTB.

2) Un branch trebuie introdus în BTB, **cu prima ocazie când el se va face**. Un salt care ar fi prezis ca nu se va face nu trebuie introdus în BTB pentru ca nu are potentialul de a îmbunatati performanta. Din acest motiv, exista strategii care atunci când trebuie evacuat un branch din BTB îl evacueaza pe cel cu potentialul de performanta minim, care nu coincide neaparat cu cel mai putin folosit (vezi [Dub91, Per93]). Astfel în [Per93] se construieste câte o variabila MPP (Minimum Performance Potential), implementata în hardware, asociata fiecarui cuvânt din BTB. Evacuarea din BTB se face pe baza MPP-ului minim. Acesta se calculeaza ca un produs între probabilitatea ca un branch din BTB sa fie din nou accesat (LRU) si respectiv probabilitatea ca saltul sa se faca. Aceasta din urma se

obtine pe baza unei istorii a respectivului salt (taken / not taken). Minimizarea ambilor factori duce la minimizarea MPP-ului si deci la evacuarea respectivului branch din BTB, pe motiv ca potentialul sau de performanta este minim.

În literatura [Hen96, Dub91, Per93], bazat pe testari laborioase, se arata ca se obtin predictii corecte în cca. 88% din cazuri folosind un bit de predictie si respectiv în cca.93% din cazuri folosind 16 biti de predictie. Microprocesorul Intel Pentium avea un predictor de ramificatii bazat pe un BTB cu 256 de intrari.

O problema dificila este determinata de **instructiunile de tip RETURN** întrucât o aceeasi instructiune, 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 bazau 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]. Sa consideram pentru o prima exemplificare a acestei idei o secventa de program C extrasa din benchmark-ul Eqntott din cadrul grupului de benchmark-uri SPECint '92:

```
(b1)  if (x = = 2)
          x = 0;
(b2)  if (y = = 2)
          y = 0;
(b3)  if (x ! = y) {
```

Se observa imediat ca în acest caz daca salturile b1 si b2 nu se vor face atunci saltul b3 se va face în mod sigur (x = y = 0). Asadar saltul b3 nu depinde de comportamentul sau anterior ci de comportamentul anterior al salturilor b1 si b2, fiind deci **corelat** cu acestea. Evident ca în acest caz schemele de predictie anterior prezentate nu vor da randament. 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. Sa consideram acum pentru analiza o secventa de program C simplificata împreuna cu secventa obtinuta în urma compilarii (s-a presupus ca variabila x este asignata registrului R1).

```
if      (x = = 0)          (b1)   BNEZ  R1, L1
          x = 1;                   ADD  R1, R0, #1
if      (x = = 1)           L1:   SUB  R3, R1, #1
                           (b2)   BNEZ  R3, L2
```

Se poate observa ca daca saltul conditionat b1 nu se va face, atunci nici b2 nu se va face, cele 2 salturi fiind deci corelate. Vom particulariza secventa anterioara, considerând iteratii succesive ale acesteia pe parcursul carora x variaza de exemplu între 0 si 5. Un BPB clasic, initializat pe predictie NU, având un singur bit de predictie, s-ar comporta ca în tabelul 2. Asadar o astfel de schema ar predictiona în acest caz, întotdeauna gresit!

Tabelul 2. Modul de predictionare al unui BPB clasic

x	Predictie b1	Actiune b1	Predictie noua b1	Predictie b2	Actiune b2	Predictie noua b2
5	NU	DA	DA	NU	DA	DA
0	DA	NU	NU	DA	NU	NU
5	NU	DA	DA	NU	DA	DA
0	DA	NU	NU	DA	NU	NU

Sa analizam acum comportarea unui predictor corelat având un singur bit de corelatie (se coreleaza deci doar cu instructiunea de salt anterior executata) si un singur bit de predictie. Acesta se mai numeste si predictor corelat de tip (1, 1). Acest predictor va avea 2 biti de predictie pentru fiecare instructiune de salt: primul bit predictioneaza daca instructiunea de salt actuala se va face sau nu, în cazul în care instructiunea anterior executata nu s-a facut iar al doilea analog, în cazul în care instructiunea de salt anterior executata s-a facut. Exista deci urmatoarele 4 posibilitati (tabelul 3).

Tabelul 3. Semnificatia bitilor de predictie pentru o schema corelata

Biti predictie	Predictie daca precedentul salt nu s-a facut	Predictie daca precedentul salt s-a facut
NU / NU	NU	NU
NU / DA	NU	DA
DA / NU	DA	NU
DA / DA	DA	DA

Ca si în cazul BPB-ului clasic cu un bit, în cazul unei predictii care se dovedeste a fi eronata bitul de predictie indicat se va complementa. Comportarea predictorului (1,1) pe secventa anterioara de program este prezentata în continuare (s-a considerat ca bitii de predictie asociati salturilor b1 si b2 sunt initializati pe NU / NU).

Tabelul 4. Modul de predictionare al unei scheme corelate

x	Predictie b1	Actiune b1	Predictie noua b1	Predictie b2	Actiune b2	Predictie noua b2
5	**NU**/NU	DA	DA/NU	NU/**NU**	DA	NU/DA
0	DA/**NU**	NU	DA/NU	**NU**/DA	NU	NU/DA
5	**DA**/NU	DA	DA/NU	NU/**DA**	DA	NU/DA
0	DA/**NU**	NU	DA/NU	**NU**/DA	NU	NU/DA

Dupa cum se observa în tabelul 4, singurele doua predictii incorecte sunt când x = 5 în prima iteratie. În rest, predictiile vor fi întotdeauna corecte, schema comportându-se deci foarte bine spre deosebire de schema BPB clasica. În cazul

general, un predictor corelat de tip (m,n) utilizeaza comportarea precedentelor m instructiuni de salt executate, alegând deci o anumita predictie de tip Da sau Nu din 2^m posibile iar n reprezinta numarul bitilor utilizati în predictia fiecarui salt. O comportare alternativa a unui salt este simplu de predictionat printr-o schema corelata, în schimb printr-o schema clasica este foarte dificil. Asadar schemele corelate sunt eficiente atunci când predictia depinde si de un anumit pattern al istoriei saltului de predictionat, corelatia fiind în acest caz particular cu istoria pe m biti chiar a acelui salt si nu cu istoria anterioarelor m salturi.

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 predictie poate corespunde la mai multe salturi. Exista în implementare 2 nivele deci: un **registru de predictie** al carui continut concatenat cu PC- ul c.m.p.s. al instructiunii de salt pointeaza la un cuvânt din **tabela de predictii** (aceasta contine bitii de predictie, adresa destinatie, etc.). În [Yeh92], nu se face concatenarea PC - registru de predictie si în consecinta se obtin rezultate nesatisfacatoare datorita **interferentei diverselor salturi** la aceeasi locatie din tabela de predictii.

În [Pan92], o lucrare de referinta în acest plan, se analizeaza calitativ si cantitativ într-un mod foarte atent, rolul informatiei de corelatie, pe exemple concrete extrase din benchmark-urile SPECint '92. Se arata ca bazat pe predictoare de tip numaratoare saturate pe 2 biti, schemele corelate (5-8 biti de corelatie utilizati) ating acurateti ale predictiilor de pâna la 11% în plus fata de cele clasice.

De remarcat ca un BPB clasic reprezinta un predictor de tip (0,n), unde n este numarul bitilor de predictie utilizati. Numarul total de biti utilizati în implementarea unui predictor corelat de tip (m,n) este:

$$N = 2^m * n * NI \quad (4)$$

unde NI reprezinta numarul de intrari al BPB-ului utilizat.

Exista citate în literatura mai multe implementari de scheme de predictie a ramificatiilor, prima implementare comerciala a unei astfel de scheme facându-se în microprocesorul Intel Pentium Pro. Astfel, implementarea tipica a unui predictor corelat de tip **GAg** (Global History Register, Global Prediction History Table) este prezentata în figura 3. Tabela de predictii PHT (Prediction History Table) este adresata cu un index rezultat din concatenarea a **doua informatii ortogonale:** PClow (i biti), semnificând gradul de localizare al saltului, respectiv registrul de predictie (HR- History Register pe k biti), semnificând "contextul" în care se situeaza saltul în program. Ambele contributii s-au facut cu scopul eliminarii interferentelor branch-urilor în tabela de predictie. Adresarea PHT exclusiv cu HR ca în articolul [Yeh92], ducea la serioase interferente (mai multe salturi puteau accesa aceelasi automat de predictie din PHT), cu influente evident defavorabile asupra performantelor. Desigur, PHT poate avea diferite grade de asociativitate. Un cuvânt din aceasta tabela are un format similar cu cel al cuvântului dintr-un BTB. Se pare ca se poate evita concatenarea HR si PClow în adresarea PHT, cu rezultate foarte bune, printr-o functie de dispersie tip SAU EXCLUSIV între acestea, care sa adreseze tabela PHT. Aceasta are o influenta benefica asupra capacitatii tabelei PHT.

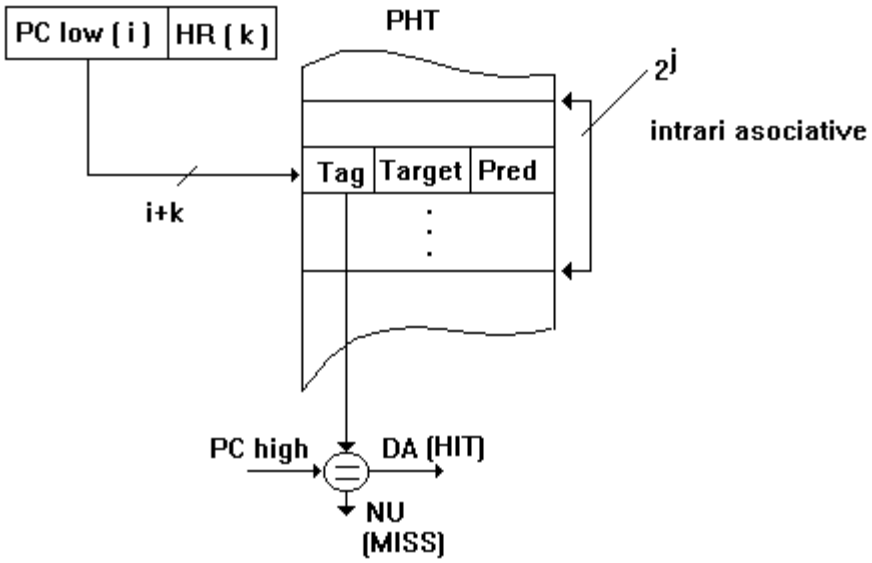


Figura 3. Structura de predictie de tip GAg

În scopul reducerii interferentelor diverselor salturi în tabela de predictii, în [Yeh92] se prezinta o schema numita **PAg-** Per Address History Table, Global PHT, a carei structura este oarecum asemanatoare cu cea a schemei GAg. Componenta HR*(k) a fost introdusa în [Vin97], având semnificatia componentei HR de la varianta GAg, adica un registru global care memoreaza comportarea ultimelor k salturi. Fara aceasta componenta, schema PAg si-ar pierde din capacitatea de adaptare la contextul programului în sensul în care schema GAg o face. *Yeh* si *Patt* renunta la informatia de corelatie globala (HRg) în trecerea de la schemele de tip GAg la cele de tip PAg, în favoarea exclusiva a informatiei de corelatie locala (HRl). În schimb, componenta HR din structura History Table, contine "istoria locala" (taken / not taken) a saltului curent, ce trebuie predictionat. Dupa cum se va arata mai departe, performanta schemei PAg este superioara celei obtinute printr-o schema de tip GAg, cu tributul de rigoare platit complexitatii hardware.

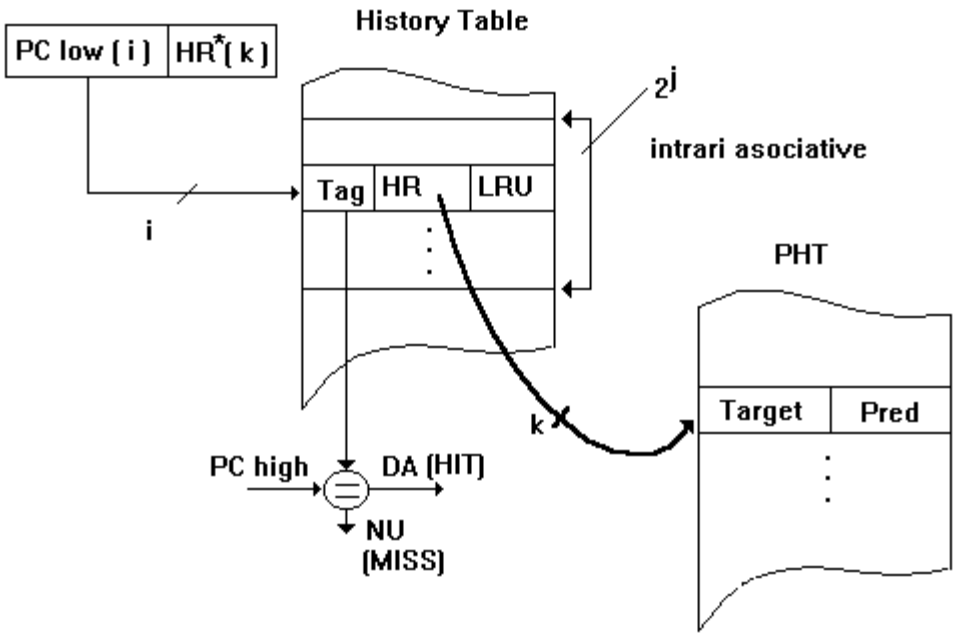


Figura 4. Structura de predictie de tip PAg

Asadar aceasta schema de tip PAg predictioneaza pe baza a **3 informatii ortogonale,** toate disponibile pe chiar timpul fazei IF: istoria HRg a anterioarelor salturi corelate (taken / not taken), istoria saltului curent HRl si PC-ul acestui salt. Daca adresarea tabelei 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 mai jos (figura 5). Complexitatea acestei scheme o face practic neimplementabila în siliciu la ora actuala, fiind doar un model utilizat în cercetare.

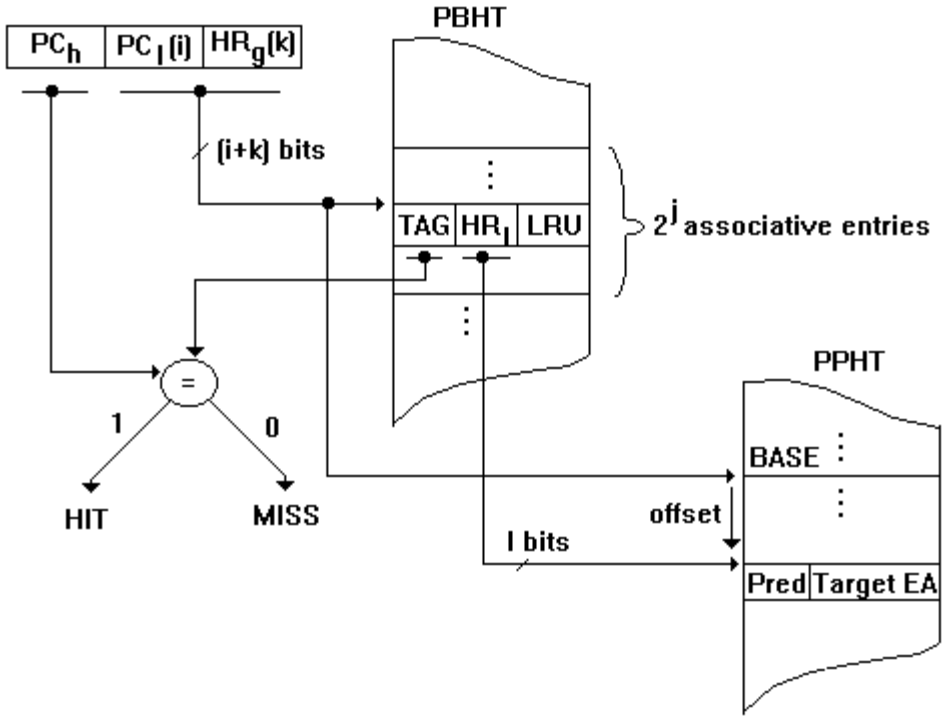


Figura 5. Structura de predictie de tip PAp

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].

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. O linie din acest cache contine ultimele N adrese tinta ale saltului împreuna cu tag-ul aferent.

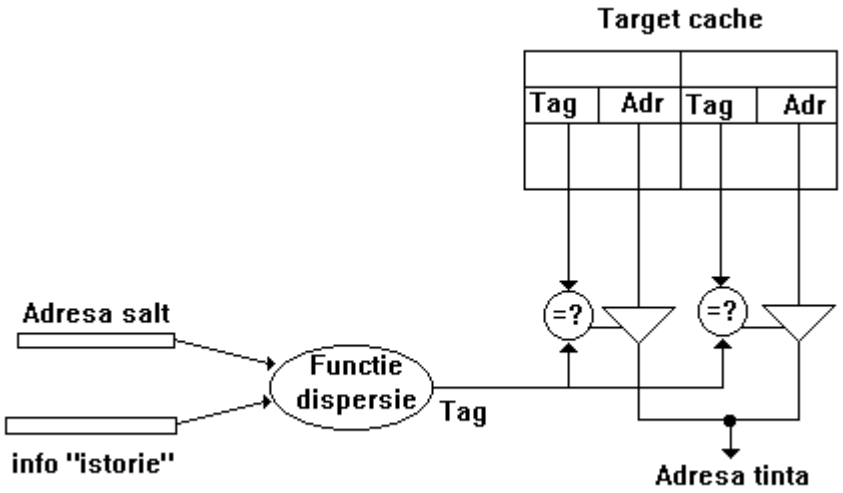


Figura 6. Predictia adresei în cazul salturilor indirecte

Informatia "istorie" provine din doua surse: istoria saltului indirect sau a anterioarelor salturi si respectiv ultimele N adrese tinta, înscrise în linia corespunzatoare din cache. Aceste doua surse de informatie binara sunt prelucrate prin intermediul unei functii de dispersie (SAU EXCLUSIV), rezultând indexul de adresare în cache si tag-ul aferent. Dupa ce adresa tinta a saltului devine efectiv cunoscuta, se va introduce în linia corespunzatoare din cache. Schema actioneaza "în mod disperat", mizând pe faptul ca la acelasi context de aparitie a unui salt indirect se va asocia o aceeasi adresa tinta. Si în opinia mea, aceasta abordare principiala pare singura posibila în cazul acestor salturi greu predictibile. Prin astfel de scheme, masurat pe benchmark-urile SPECint '95 acuratetea predictiei salturilor

indirecte creste si ca urmare, câstigul global asupra timpului de executie este de cca 4.3% - 9% [Cha97].

3. Desfasurarea lucrarii

3.1. Prezentarea simulatorului

Partea practica a lucrarii consta în simulare efectuata pe benchmark-urile 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 deficienta 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 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, BRA – salt neconditionat si MOV PC, RA – instructiune de tip Return).

Automatul de predictie 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 fiecarei stari. Pentru o mai buna întelegere a functionarii automatului exemplificam pe doua situatii diferite:

1. automatul **ABAB:2** - pe un bit
2. automatul **BCBAADCD:12** - pe 2 biti

Tabelul care descrie functionarea primului automat este urmatorul:

Stare Curenta	Stare urmatoare		Predictie
	Pentru intrare = 0	Pentru intrare = 1	
A	A	B	0
B	A	B	1

Prima parte a sirului pâna la caracterul ‘:’ reprezinta tranzitiile pentru starea ‘A’ cu intrare 0, apoi cu intrare 1, apoi tranzitiile din ‘B’ pentru aceleasi intrari. Numarul din a doua parte este “vazut” în binar sub forma “0010” si reprezinta

iesirile asociate fiecarei stari în parte (starii ‘A’ îi este asociat cel mai putin semnificativ bit, în acest caz bitul ‘0’, urmatorul bit lui ‘B’, bitul ‘1’, etc).

Tabelul care descrie functionarea celui de-al doilea automat arata astfel:

Stare Curenta	Stare urmatoare		Predictie
	Pentru intrare = 0	Pentru intrare = 1	
A	B	C	0
B	B	A	0
C	A	D	1
D	C	D	1

Cache-ul de predictie simulat este cel din figura 5. Acesta e alcatuit din doua tabele:

- prima cu 2^{i+k+j} intrari asociative care contine istoria locala a saltului, tag-ul de verificat si un câmp LRU asociat fiecarei intrari. Adresarea în aceasta tabela se face cu cei mai putini semnificativi biti ai PC-ului (i) concatenati cu istoria globala a saltului (k).
- a doua tabela are 2^{i+k} intrari. La ofset-ul l(istoria locala anterior citita) în aceasta tabela se afla bitul de predictie si adresa noii instructiuni tinta.

Parametrii de intrare ai simulatorului sunt în ordine:

- *Nume trace (.tra)* – oricare din cele 8 trace-uri cu extensia *.tra*. Fie fsort.tra.
- *Tip automat* – sir de caractere ce descrie automatul. Fie BCBAADCD:12
- *Numar seturi asociative* – se alege o putere de a lui 2. Fie 4 seturi $\Rightarrow$ i=2
- *Dimensiune set asociativ (nr. intrari)* – Presupunem 8 intrari $\Rightarrow$ j=3
- *Dimensiune registru istorie locala* – în biti. Presupunem l=2

- *Dimensiune registru istorie globala* – în biti. Presupunem k=3

Simulatorul genereaza urmatoarele rezultate:

- Numarul de branch-uri predictionate corect.
- Numarul de branch-uri predictionate gresit.
- Numarul de branch-uri negasite în prima tabela (salturi care au generat MISS în procesul de cautare).
- Numarul de adrese predictionate incorect (salturi care sunt în prima tabela, predictionate corect dar adresa tinta predictionata nu este cea corecta – fiind modificata între timp).

3.2. Probleme propuse spre rezolvare

Cu ajutorul simulatorului *pap.exe* generati:

1. Rata de predictie corecta pe fiecare benchmark în parte într-o configuratie data.

2. Stabiliti influenta parametrilor i, j, k si l (câte un singur parametru trebuie variat succesiv) asupra

ratei de predictie corecte a branch-urilor si stabiliti valoarea optima a acestora.

3. Stabiliti influenta corelata a celor trei parametri i, j si k asupra ratei de predictie.

4. Cuantificati câstigul introdus printr-o implementare ideala, în care nu ar exista adresa tinta modificata în tabele.

Bibliografie

[1] [Cha97] **Chang P.Y., Hao E., Patt Y.N**. - *Target Prediction for Indirect Jumps,* ISCA '97 (http://www.eecs.umich.edu/HPS)

[2][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

[3][Koh95] **Kohonen T., et al.** - *Learning Vector Quantization (LVQ). Program Package Ver. 3.1*, Helsinki University of Technology, SF-02150 Espoo, Finland, 1995

[4][Per93] **Perleberg C., Smith A. J**. - *Branch Target Buffer Design and Optimisation,* IEEE Trans. Computers, No. 4, 1993.

[5][Vin97] **Vintan L., Steven G.B. -** *Memory Hierarchy Limitations in Multiple Instruction- Issue Processor Design*, Proceedings of Euromicro '97 Conference (Short Papers), Budapest 1-4 Sept. 1997, IEEE Computer Society Press, California, USA, 1997

[6][Yeh92] **Yeh T., Patt Y.** - *Alternative Implementations of Two Level Adaptive Branch Prediction,* 19 th Ann. Int.'L Symp. Computer Architecture, 1992

[7][Hen96] **Hennessy J., Patterson D.** - *Computer Architecture*: *A quantitative Approach*, Morgan Kaufmann Publishers, Second Edition, 1996.

[8][Dub91] **Dubey P., Flynn M.** - *Branch Strategies*: *Modeling and Optimization*, IEEE Transaction on Computer, No 10, 1991.

[9][Mud96] **Mudge T.N., et al.** - *Limits of Branch prediction*, Technical Report, Electrical Engineering and Computer Science Department, The University of Michigan, Ann Arbor, Michigan, USA, 1996

[10][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

[11][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.

PROBLEME PROPUSE SPRE REZOLVARE

1. Sa se proiecteze un cache de instructiuni cuplat la un procesor superscalar (VLIW). Lungimea blocului din cache se considera egala cu rata de fetch a procesorului, în acest caz 4 instructiuni / 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 directa** (direct mapped)

Ce se întâmpla daca în locul adresarii cu adrese fizice se considera adresare cu adresa virtuala?

2. a. De ce este dificila procesarea «Out of Order» a instructiunilor Load respectiv Store într-un program si de ce ar putea fi ea benefica?

b. 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];
```

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;
```

3. 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=înscriere 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.

4. 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 ?

5. 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 harware 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. ?

6. 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).

7. 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. O 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 aceeasi capacitate.

8. Se considera secventa de program RISC:

```
1:    ADD  R1, R11, R12
2:    ADD  R1, R1, R13
3:    ADD  R2, R3, R9
4:    ADD  R2, R1, R2
5:    ADD  R1, R14, R15
```

a. Reprezentati graful dependentelor de date (numai dependentele de tip RAW)

b. Stiind ca între 2 instructiuni dependente RAW si succesive e nevoie de o întârziere de 2 cicli, în câti cicli s-ar executa secventa ?

c. Reorganizati secventa în vederea unui timp minim de executie (nu se considera alte dependente decât cele de tip RAW).

9. a. Considerând un procesor RISC pe 5 nivele pipe (IF, ID, ALU, MEM, WB), fiecare durând un ciclu de tact, precizati câti cicli de întârziere («*branch delay slot*») impune o instructiune de salt care determina adresa de salt la finele nivelului ALU ?

b. De ce se prefera implementarea unor busuri si memorii cache separate pe instructiuni, respectiv date în cazul majoritatii procesoarelor RISC (pipeline) ?

c. De ce sunt considerate instructiunile CALL / RET mari consumatoare de timp în cazul procesoarelor CISC (ex. I-8086) ? Cum se evita acest consum de timp în cazul microprocesoarelor RISC ?

10. Considerând un microprocesor virtual pe 8 biti, având 16 biti de adrese, un registru A pe 8 biti, un registru PC si un registru index X, ambele pe 16 biti si ca opcode-ul oricarei instructiuni e codificat pe 1 octet, sa se determine numarul impulsurilor de tact necesare aducerii si executiei instructiunii «memoreaza A la adresa data de (X+deplasament)». Se considera ca instructiunea e codificata pe 3 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.

11. Relativ la o arhitectura de memorie cache cu mapare directa se considera afirmatiile:

a. Nu permite accesul simultan la câmpul de date si respectiv «tag» al unui cuvânt accesat.

b. La un acces de scriere cu hit, se scrie în cache atât data de înscris cât si «tag-ul» aferent.

c. Rata de hit creste usor daca 2 sau mai multe blocuri din memoria principala - accesate alternativ de catre microprocesor - sunt mapate în acelasi bloc din cache.

Stabiliti valoarea de adevar a acestor afirmatii si justificati pe scurt raspunsul.

12. Ce corectie (doar una!) trebuie facuta în secventa de program asamblare pentru ca translatarea de mai jos sa fie corecta si de ce ? Initial, registrii R_i, R_k, R_l, R_j contin respectiv variabilele i, k, l, j. Primul registru dupa mnemonica este destinatie. (R_j+offset) semnifica operand în memorie la adresa data de (R_j+offset).

k = a[i+2]+5;
l = c[j+9] - k;

i1: ADD R_k, #2, R_i
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

13. Se considera un microsistem realizat în jurul unui microprocesor care ar accepta o frecventa maxima a tactului de 20 MHZ. Regenerarea memoriei DRAM se face <u>în mod transparent</u> pentru microprocesor. Procesul de regenerare dureaza 250 ns. Orice ciclu extern al procesorului dureaza 3 perioade de tact. Poate functiona în aceste conditii microprocesorul la frecventa maxima admisa? Justificati.

14. Explicati concret rolul fiecareia dintre fazele de procesare (ALU, MEM, WB) în cazul instructiunilor:

a. STORE <u>R5</u>, (R9)06h;
sursa

b. LOAD R7, (R8)F3h;
dest

c. AND R5, R7, R8.
dest

15. 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

16. 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.

17. Se considera instructiunea (I-8086):

3000h: MOV [BX]0F3h, AX

EA dest sursa

a. La ce adresa fizica se aduce opcode-ul instructiunii ?
b. La ce adrese fizice se scriu registrii AL, respectivAH ?

Înainte de executia instructiunii avem: CS = 1D00h
BX = 1B00h
SS = 2000h
DS = DF00h.

18. Se considera instructiunea (I-8086):

2000h: PUSHAX

EA

a. De la ce adresa se aduce instructiunea ?
b. La ce adrese fizice se scriu registrii AL, respectiv AH ?

Se stie ca înainte de executia instructiunii PUSH avem: CS = AE00h
SS = 1FF0h
SP = 001Eh
DS = 1F20h.

19. Un automat de regenerare al memoriilor DRAM declanseaza efectiv procesul de regenerare daca sunt simultan îndeplinite conditiile:

a. activare semnal cerere refresh (CREF).

b. microprocesorul nu lucreaza momentan cu memoria. Având în vedere dezideratul regenerarii «transparente» (sa nu fie simtita de catre microprocesor), ar functiona corect automatul ? Comentati si sugerati o eventuala corectie.

20. 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 citeste succesiv de la adresele 0, 8, 0, 6, 8, 10, 8 (primul acces la o anumita adresa va fi cu MISS).

21. Se considera secventa de program RISC:

```
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
```

Între doua instructiuni dependente RAW si succesive în procesare, e nevoie de o întârziere de 1 ciclu de tact.

a. În câti cicli de tact se executa secventa initiala ?

b. În câti cicli de tact se executa secventa reorganizata aceasta secventa în vederea unui timp minim de procesare ?

22. Se considera o unitate de disc având rata de transfer de 25×10^4 biti/s, cuplata la un microsistem. 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 2μs si ca rutina de tratare dureaza 10μs, sa se calculeze timpul pe care CPU îl are disponibil între 2 transferuri succesive de octeti.

23. 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 ?

24. 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)

25. 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].

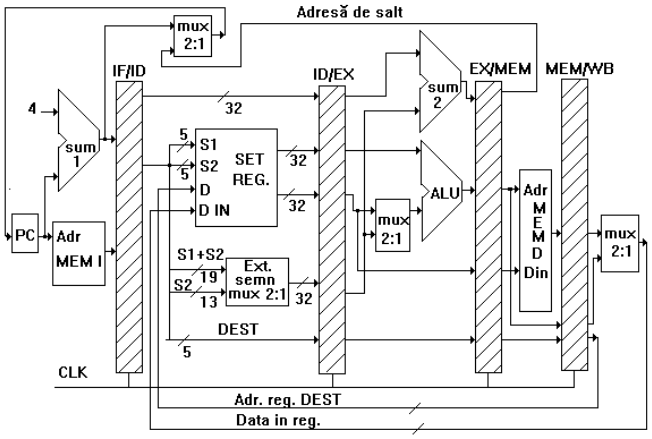
26. De ce implementarea algoritmului lui R. TOMASULO într-o arhitectura superscalara ar putea reduce din «presiunea» la citire asupra seturilor de registri generali ? Gasiti vreo similitudine în acest sens, între un CPU superscalar având implementat acest algoritm si un CPU de tip TTA (Transport Triggered Architecture) ?

27. 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 ?

28. 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.)

29. Cum explicati posibilitatea interblocarii proceselor în cadrul limbajului OCCAM ? Ce întelegeti prin «sectiune critica de program» în cadrul unui sistem multimicro ? Care este «mesajul» transmis de «legea lui AMDAHL» pentru sistemele paralele de calcul ?

30. Se considera structura hardware a unui microprocesor RISC, precum în figura de mai jos.



Raspundeti la urmatoarele întrebari.

a. Ce tip de instructiuni activeaza sumatorul «sum 2» si în ce scop ?

b. 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 ?

c. Ce rol are unitatea ALU în cazul unei instructiuni de tip LOAD ?

d. Ce informatie se memoreaza în latch-ul EX/MEM în cazul instructiunii: ST (R7)05, R2 si de unde provine fiecare informatie ?

31. 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.

În limbaj de asamblare MIPS prezentati solutiile urmatoarelor probleme:

32. 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, sfârsitul sirului va fi dat de citirea caracterului ‘0’.
Modificati programul astfel încât ‘0’ – care marcheaza sfarsitul sirului, sa nu fie tiparit.

33. 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.

34. 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 urmatoarele 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.*

35. Realizati un program care citeste de la tastatura doua numere naturale *n* si *k* (n>k) si calculeaza si afiseaza pe consola valorile urmatoare:

$$C_n^k \text{ si } A_n^k$$

36. Sa se citeasca un sir de numere întregi de la tastatura, a carui dimensiune este citita tot de la tastatura. Sortati sirul prin metoda "*bubblesort*", memorati succesiv datele la adresa 0x10012000 si afisati sirul sortat pe consola.

37. 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.

38. Scrieti un program, în limbaj de asamblare DLX, care citeste *n* numere întregi de la tastatura prin intermediul modulului **Input.s** (vezi lucrarea *Investigatii Arhitecturale Utilizând Simulatorul DLX*) si calculeaza maximul, minimul si suma numerelor si le depune succesiv în memoria DLX la adresa 0x1500.

39. Proiectati automatul de control cache, într-un sistem multimicroprocesor simetric pe bus comun, având în vedere ca un bloc din cache se poate afla într-una din starile: PARTAJAT, EXCLUSIV sau INVALID.

40. Se considera un sistem multimicroprocesor (SMM) cu N microprocesoare legate printr-o retea de interconectare (RIC) la N module fizice de memorie. În

ce ar consta si ce ar permite o RIC cu largime de banda maxima ? Dar una cu largime de banda minima ?

41. Într-un SMM cu memorie centrala partajata si în care fiecare procesor detine un cache propriu, un procesor initiaza o scriere cu MISS într-un anumit bloc aflat în starea "partajat" ("shared"). Precizati procesele succesive care au loc în urma acestei operatii.

42. Într-un SMM un procesor initiaza o citire cu MISS la un bloc invalid în cache-ul propriu, blocul aflându-se în copia exclusiva într-alt procesor. Precizati concret procesele succesive care au loc în urma acestei operatii.

Bibliografie

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

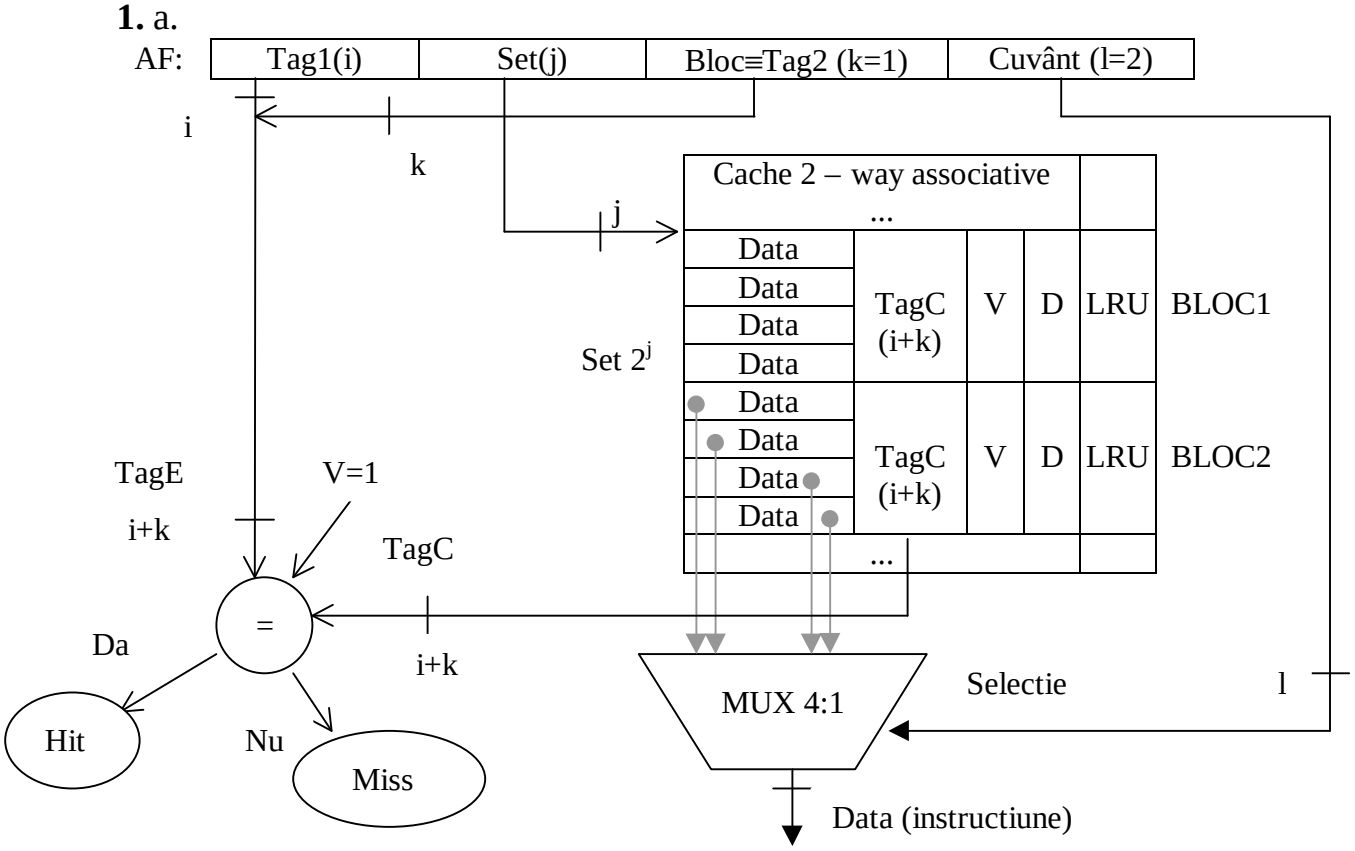
[2] **Yeh T. Y., Patt Y. N.** – *Alternative Implementation of Two-Level Branch Prediction*, Department of EECS, The University of Michigan, 1992.

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

[4] **Knuth D. E.** – *Tratat de programarea calculatoarelor*, vol. I, IV, Editura Tehnica, Bucuresti, 1974.

INDICATII DE SOLUTIONARE PENTRU PROBLEME PROPUSE

1. a.

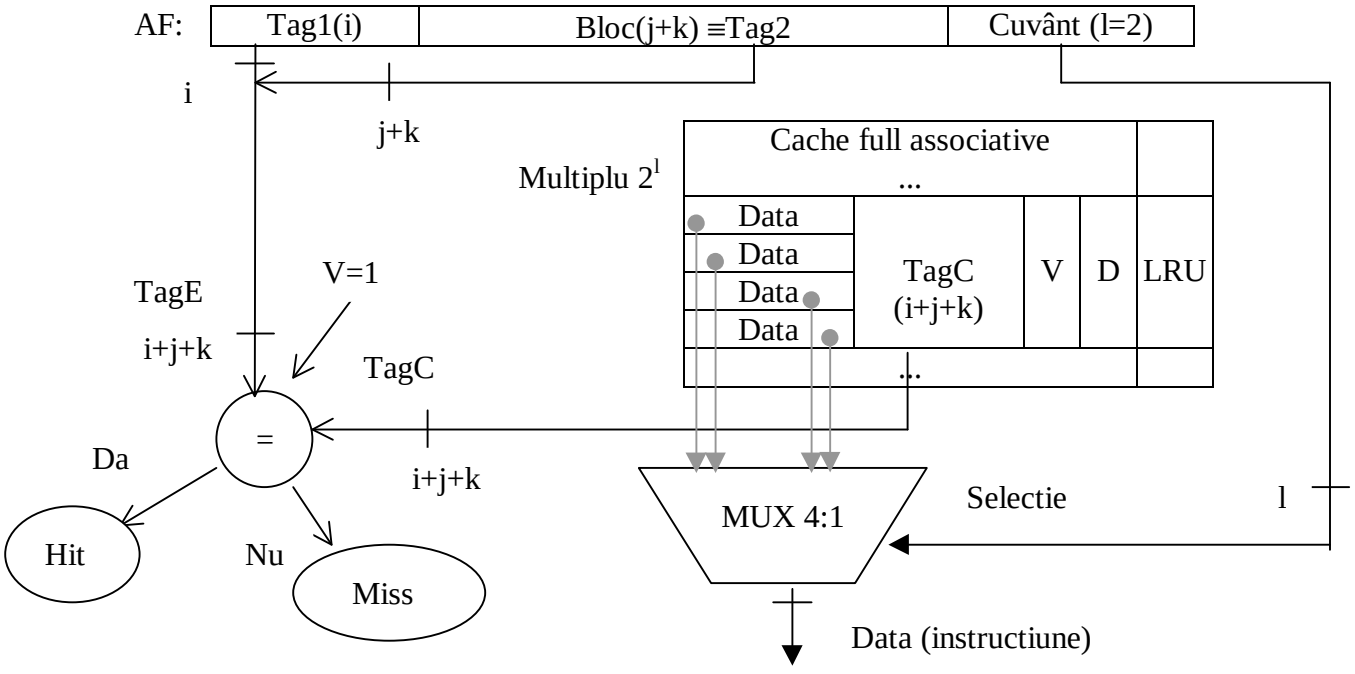


Cache-ul semiasociativ contine 2^j seturi, fiecare set contine 2 blocuri.

V – bit de validare (0 – nu e valida data; 1 – valida;). Initial are valoarea 0. Este necesar numai pentru programe automodificabile la cache-urile de instructiuni.

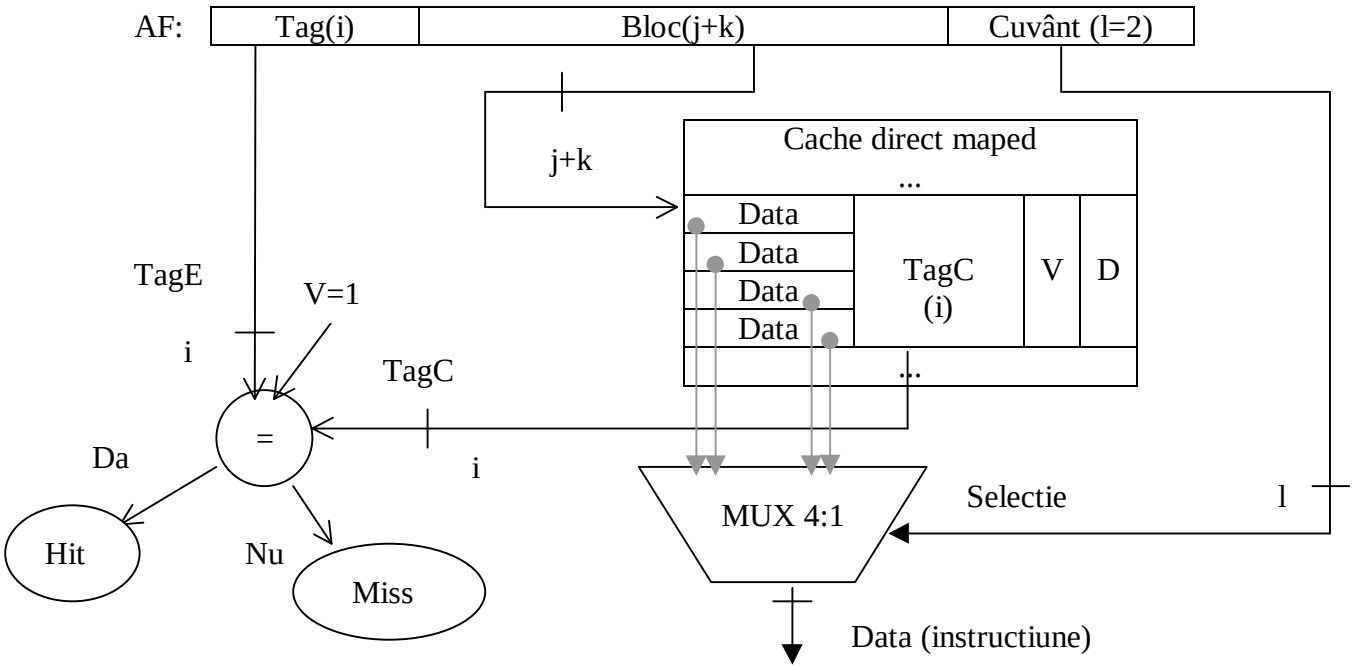
D – bit Dirty. Este necesar la scrierea în cache-ul de date.

b.



În cazul cache-urilor full asociative dispare câmpul set. Practic exista 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 acelasi loc de fiecare data când sunt accesate. Din acest motiv vom sti la fiecare acces ce data va fi evacuata din cache, nemaifiind necesar câmpul LRU (evacuare implicita).

2. a. 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.

b. Secventele de program în limbajul unui procesor RISC ar deveni:

$R1 \leftarrow x[i];$
$(R1) \rightarrow a[2i];$ STORE Adr1
$R2 \leftarrow a[i+1];$ LOAD Adr2
$R6 \leftarrow (R2) + 5;$
$(R6) \rightarrow y[i];$

Cele doua instructiuni cu referire la memorie ar fi paralelizabile (executabile Out of Order) daca $i \neq 1$ (Adr1 ≠ 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.

3. a. Vezi pr. 24 a), b).

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.

4. Nivelul **WR** este mai prioritar datorita hazardurilor RAW între doua instructiuni succesive (vezi si **30.b**).

5. $FR_{med} = 5$; Da. Predictia a N PC-uri simultan implica $FR_{med} \cong N \times 5$.

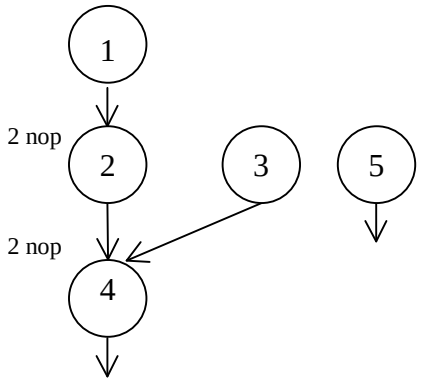
6. $C \rightarrow (N-1) + (N-2) + ... + 2 + 1 = N(N-1) / 2$ comparatoare RAW.

Pentru (N+1) instructiuni => $N(N+1) / 2$ comparatoare $\rightarrow C'$

$C' / C = N(N+1) / N(N-1) => C' = C(N+1) / (N-1)$

7. a) A. b) F. c) F. d) F.

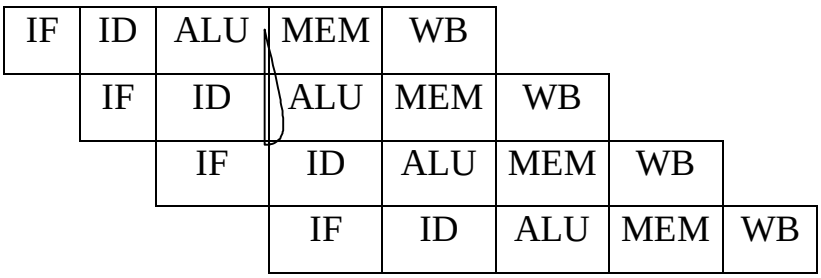
8. a)



b) 5 cicli pentru instr. + 4 cicli nop = 9 cicli executie

c) 1 – 3 – 5 – 2 – nop – nop – 4 => 7 cicli executie

9. a) Instructiunea de salt:



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).

10. 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
Scriere A în memorie	4

Total cicli executie = 20.

11. a) **Fals**! În caz de hit, pe baza compararii tag-urilor, se citeste 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 scade.

12. i5: LOAD R_l, ($\underline{R_l}$+0)

13. **Nu**! O regenerare transparenta (dureaza 250ns) trebuie sa "încapa" între 2 accese la DRAM ale microprocesorului. Cum un ciclu extern al procesorului

dureaza 3 tacte (150ns) regenerarea transparenta este imposibila la frecventa maxima a procesorului.

14.

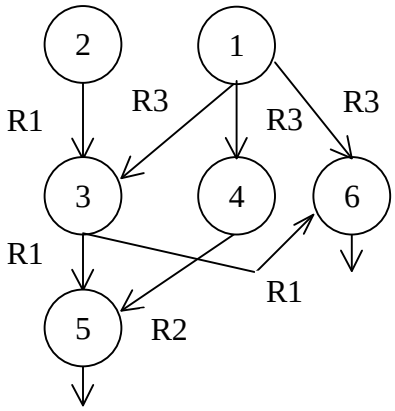
a. ALU: (R9) + 06; MEM: scriere R5 la adresa (R9) + 06; WB: nimic

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.

15.

a. 1 - 2 - nop - 3 - 4 - nop - 5 - 6 => 8 cicli.



b. 1 - 2 - 4 - 3 - 6 - 5 => 6 cicli.

16.

a. Nr.tacte /s consumate pentru interogare mouse: 30×100 = 3000 tacte/ s

$$f = \frac{3000}{50 \times 10^6} = 0{,}006\%$$

b. $$\frac{\text{Nr.interogari}}{s} = \frac{50 \frac{ko}{s}}{2 \frac{o}{\text{acces interogare}}} = 25 \times 2^{10} \frac{\text{acces interogare}}{s}$$

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 50×10^6 tacte, iar pentru un transfer cu o rata de 2 Mo/ s sunt necesare într-o secunda 50×2^{20} tacte, imposibil).

17.

a. AF = 20000h;
b. AL se memoreaza la adresa: E0BF4h (adresa para)

AH se memoreaza la adresa: E0BF3h (adresa impara) [scriere pe cuvânt la adresa impara]

SS = 2000h constituie informatie redundanta.

18.

a. AF = B0000h;
b. AL se memoreaza la adresa: 1FF1Ch (adresa para)

AH se memoreaza la adresa: 1FF1Dh (adresa impara) [scriere pe cuvânt la adresa impara]

DS = 1F20h constituie informatie redundanta.

19. Nu! Daca se activeaza CREF si microprocesorul vrea sa acceseze apoi memoria, acesta trebuie pus în stare «WAIT». La activarea CREF în conditiile în care microprocesorul nu lucreaza cu memoria, regenerarea va astepta ca microprocesorul sa “atace” memoria, pentru ca dupa aceea sa se “strecoare”.

20. a – cazul memoriei mapate direct cu 4 locatii.

Se acceseaza pe rând locatiile:

Locatia accesata	**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

Rezulta în cazul memoriei mapate direct un singur acces cu hit R_{miss} = 6 / 7 = 85.71%

b – cazul memoriilor semiasociative cu 2 seturi a câte 2 cuvinte.

Întrucât toate adresele sunt pare se acceseaza doar blocurile din **setul 0**.

Locatia accesata	**Setul 0**		
0	0	X	Miss
8	0	8	Miss
0	0	8	Hit
6	0	6	Miss. Se evacueaza 8.
8	8	6	Miss. Se evacueaza 0.
10	8	10	Miss. Se evacueaza 6.
8	8	10	Hit.

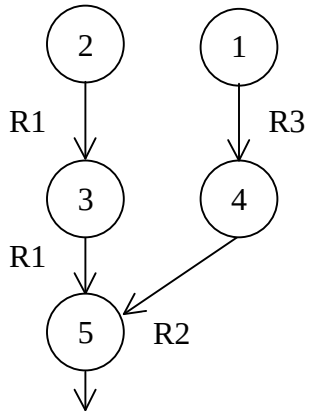
În cazul memoriei two-way asociative sunt 2 accese cu hit R_{miss} = 5 / 7 = 71.42%

c – cazul memoriilor complet asociative.

Locatia accesata	**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_{miss} = 4 / 7 = 57.14%

21. a. 1 - 2 - nop - 3 - 4 - nop - 5 => 7 cicli.



b. 1 - 2 - 4 - 3 - nop - 5 => 6 cicli.

22. În 1 s se transfera 25×10^4 biti.

În x s se transfera 1 octet.

Rezulta x = 8 / (25×10^4) = 32 μs

Fie t_r = timpul disponibil dintre 2 transferuri succesive de octeti.

t_1 = timpul scurs între aparitia întreruperii si intrarea în rutina de tratare;

t_1 = 2 μs;

t_2 = timpul în care este tratata rutina de întrerupere; t_2 = 10μs;

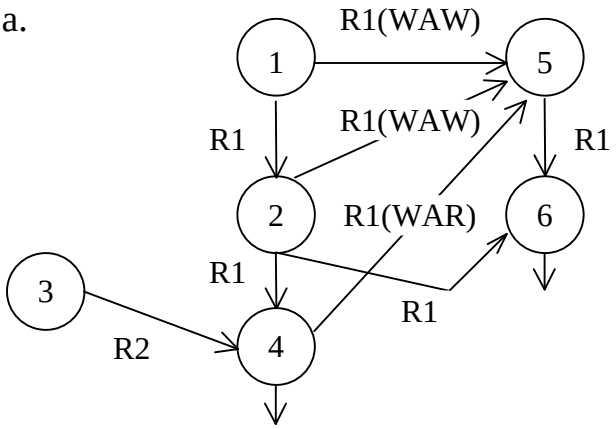
$t_r = x - t_1 - t_2 = (32 – 2 – 10)$ μs => t_r = 20 μs.

23. a. $\frac{8,5+0,11\times6\times3}{8,5}=1,2329$ => diminuare a performantei cu ≈ 24%.

b. **Avantajul** implementarii unei pagini de capacitate «mare» într-un sistem de memorie virtuala consta în principiul localizarii acceselor, determinând o optimizare a acceselor la disc (se reduce numarul acestora).

Dezavantajul îl reprezinta aducerea inutila de informatie de pe disc (pierdere de timp), care trebuie apoi evacuata în cazul unei erori de tipul *PageFault*. Dimensiunea paginii se alege în urma unor simulari laborioase.

24. a.



1 - nop - 2 - 3 - nop - 4 - 5 - nop - 6 => 9 cicli executie.

b.

1: R1 ← (R11) + (R12)

5: R1' ← (R14) + (R15)

3: R2 ← (R3) + 4

2: R1 ← (R1) + (R13)

6: R1' ← (R1') + (R16)

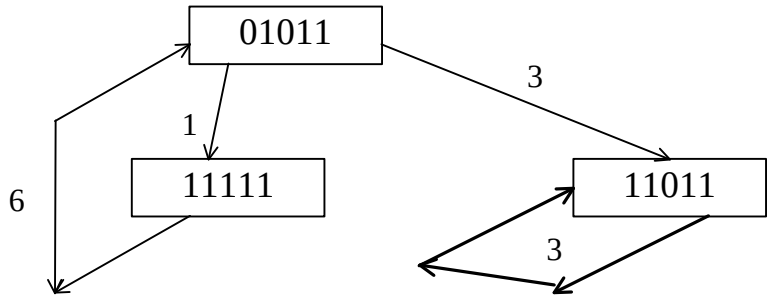
4: R2 ← (R1) + (R2)

1 - 5 - 3 - 2 - 6 - 4 => 6 cicli executie

25. Daca alegem strategia Greedy obtinem rata de procesare $I=\frac{2}{7}\,\mathbf{i}\text{nstr}/\text{ciclu}$

Daca alegem strategia non - Greedy rata de procesare obtinuta este: $I = \frac{1}{3}$ **i**$^{instr}/_{ciclu}$

În acest caz e mai avantajos sa alegem stategia non - Greedy.



26. 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, înlaturând o mare parte din dependentele RAW dintre instructiuni [1].

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 [1].

27. 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 [1].

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 [1, 2].

30. 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 înca 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).

32. Se citesc caractere de la intrare pâna se întâlneste 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.

33. 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:

$$cmmdc(a,b) \times cmmmc(a,b) = a \times b$$

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.

34. 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:

a. Se salveaza în stiva registrii corespunzatori parametrilor (tijelor).
b. Se actualizeaza numarul de discuri, n ← (n-1) si rolul fiecarei 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].
c. Se refac parametrii din stiva (tijele). Se afiseaza transferul [cel de-al *n* - lea disc de pe tija sursa(A) pe tija destinatie(B)].
d. Se salveaza în stiva registrii corespunzatori parametrilor (tijelor).
e. Se actualizeaza numarul de discuri, n ← (n-1) si rolul fiecarei 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].

35. 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:

$$C_n^k = \frac{n!}{(n-k)!k!}; \qquad A_n^k = \frac{n!}{(n-k)!}$$

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 1×2× ... ×k si n×(n-1)× ... ×[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:

a. Se actualizeaza parametrii n ← (n-1) si k ← (k-1). Se reapeleaza subrutina.
b. Se preiau din stiva termenii salvati si se înmultesc cu rezultatele calculate pâna la acest pas.
c. 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.

36. Se citeste dimensiunea sirului, si elementele care se vor memora la adresa ceruta. Este necesara o variabila booleana, initial având valoarea 1 (true), care specifica daca au fost facute interschimbari între elementele sirului. Se apeleaza rutina *bubble* în care au loc urmatoarele:

Variabila booleana este resetata (ia valoarea 0).

Se porneste de la primul element al sirului si se compara elementele învecinate. Daca doua dintre acestea nu îndeplinesc relatia de ordine ceruta (impusa), se interschimba (fiecare element se memoreaza la adresa celuilalt) si se seteaza variabila booleana. Se parcurge tot sirul, astfel ca dupa o prima parcurgere elementul maxim (sau minim) se va afla pe ultima pozitie din sir.

Daca a avut loc cel putin o interschimbare se reia algoritmul. Altfel, sirul se gaseste în memorie sortat si va fi afisat pe consola.

37. 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.

38. Se citesc prin intermediul modulului **Input.s** usor modificat, dimensiunea unui sir si elementele sale ca numere întregi pozitive. Se stocheaza în memoria DLX la adresa specificata. Se seteaza suma si elementul maxim pe 0, iar minimul la o valoare maxima admisa (fie 0x7fff). Se parcurge sirul si se executa operatiile:

a. Se însumeaza în registrul suma - (fie r15) – suma numerelor anterior citite cu cea a elementului curent.
b. Daca elementul curent (citit din memorie), este mai mare decât maximul, atunci maximul devine elementul curent.
c. Daca elementul curent este mai mic decât minimul, atunci minimul devine elementul curent.
d. Se reia punctul a. pâna am parcurs tot sirul.

Se afiseaza suma, maximul si minimul în fereastra DLX-IO.

40. Retea de tip crossbar, permite implementarea oricarei bijectii procesoare-module memorie; retea unibus, un singur procesor master la un moment dat.

41.
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".

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] **Yeh T. Y., Patt Y. N.** – *Alternative Implementation of Two-Level Branch Prediction*, Department of EECS, The University of Michigan, 1992.

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

[4] **Knuth D. E.** – *Tratat de programarea calculatoarelor*, vol. I, IV, Editura Tehnica, Bucuresti, 1974.