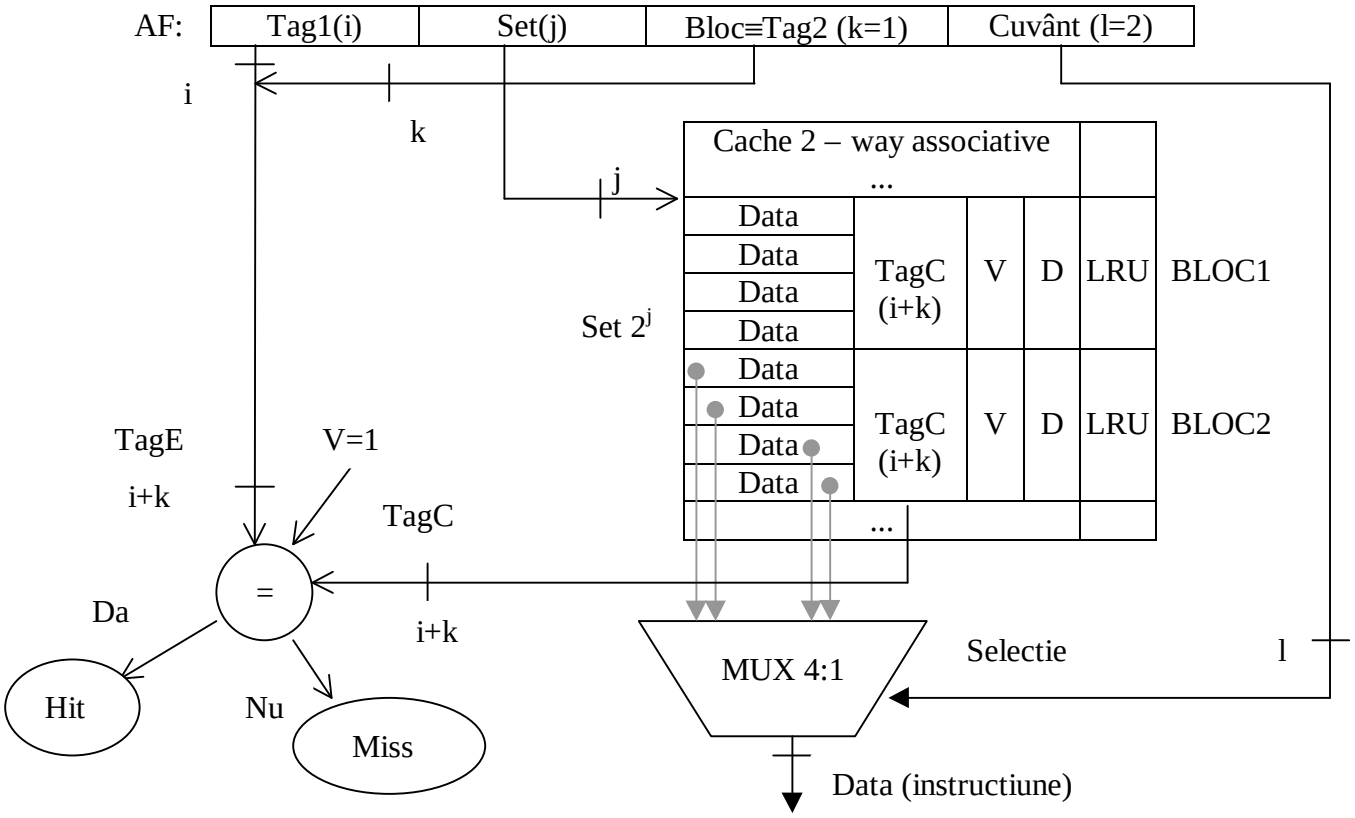
INDICATII DE SOLUTIONARE PENTRU PROBLEME PROPUSE

1. a.

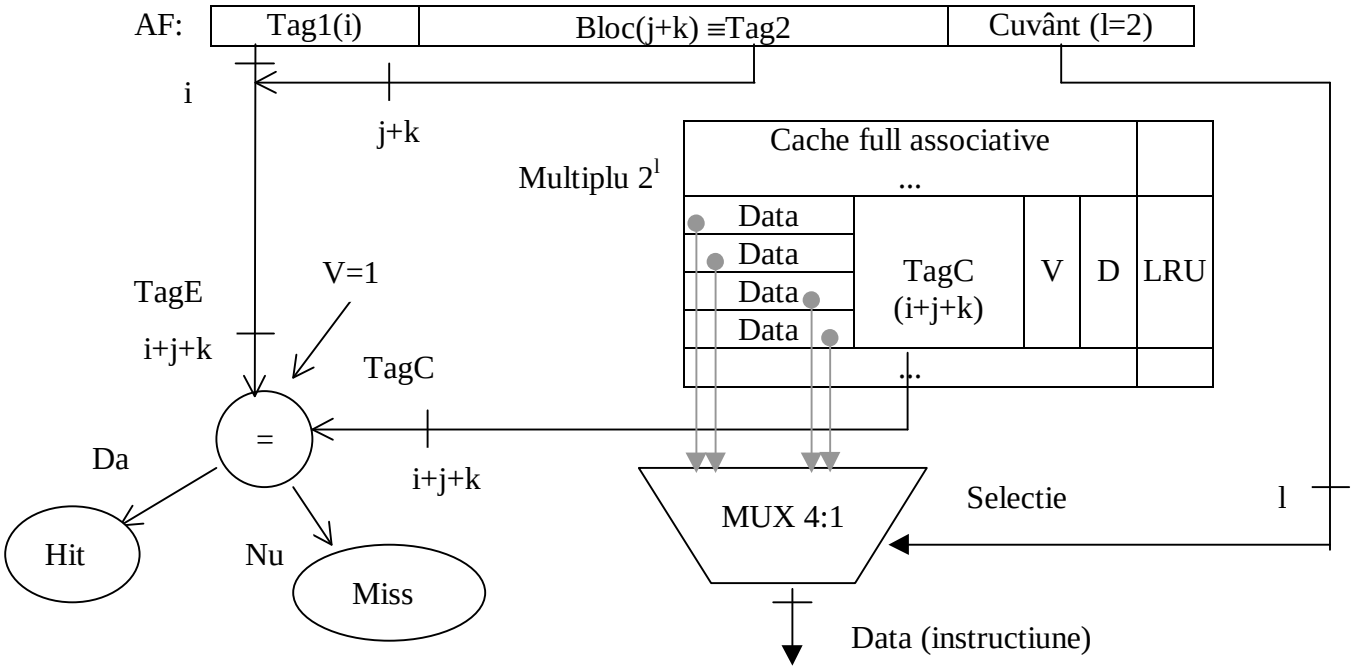


Cache-ul semiasociativ contine 2^j seturi, fiecare set contine 2 blocuri.

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.

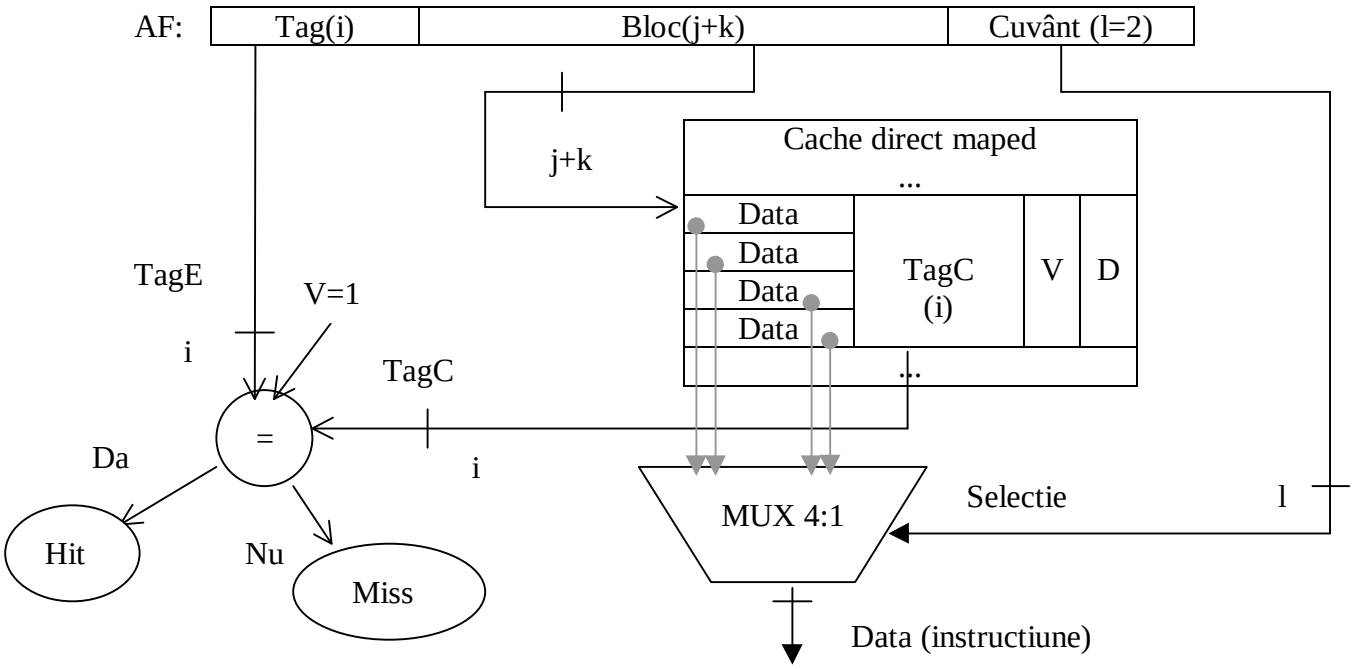
D – bit Dirty. Este necesar la scrierea în cache-ul de date.

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 ← x[i];
(R1) → a[2i];        STORE Adr1
R2 ← a[i+1];         LOAD  Adr2
R6 ← (R2) + 5;
(R6) → y[i];
```

Cele doua instructiuni cu referire la memorie ar fi paralelizabile (executabile Out of Order) daca $i \neq 1$ ($Adr1 \neq Adr2$).

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 $FR_{med} \cong N \times 5$.

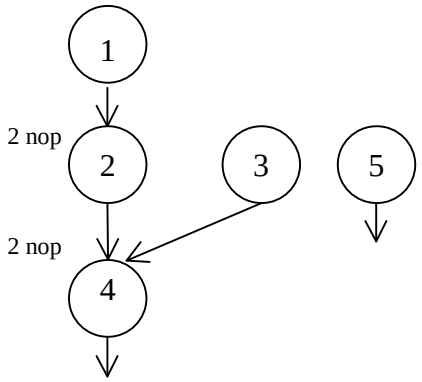
6. C $\rightarrow$ (N-1) + (N-2) + ... + 2 + 1 = N(N-1) / 2 comparatoare RAW.

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.

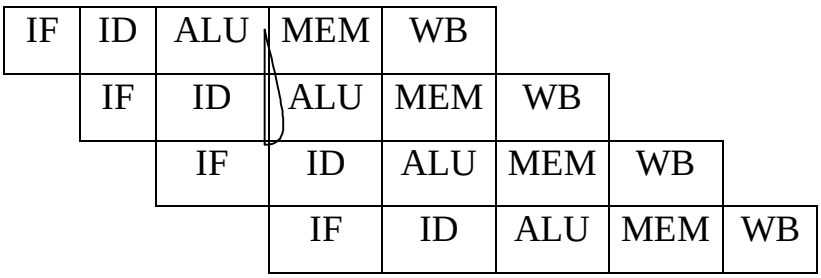
8. a)



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.

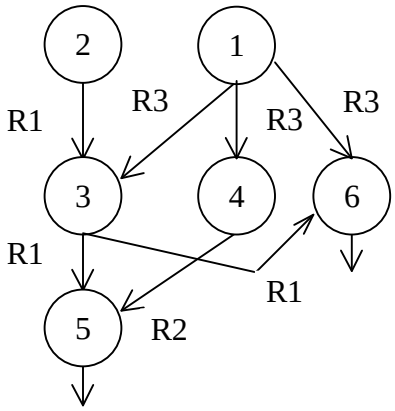
a. ALU: (R9) + 06; MEM: scriere R5 la adresa (R9) + 06; WB: nimic

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.

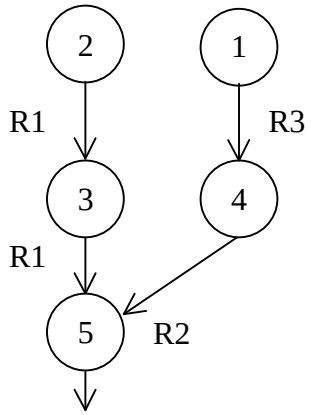
În cazul memoriei two-way asociative sunt 2 accese cu hit R_{miss} = 5 / 7 = 71.42%

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;

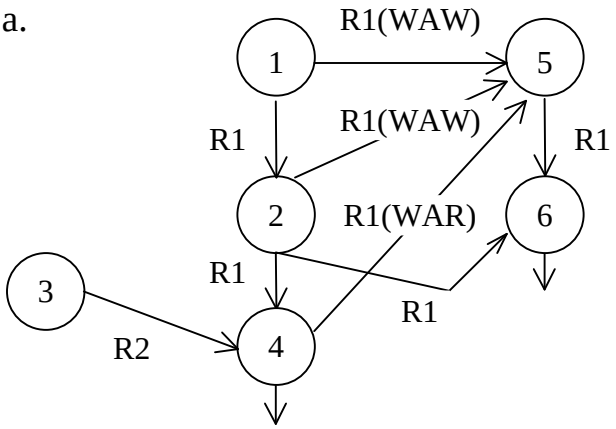
$t_r = x - t_1 - t_2 = (32 – 2 – 10)$ μs => t_r = 20 μs.

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)

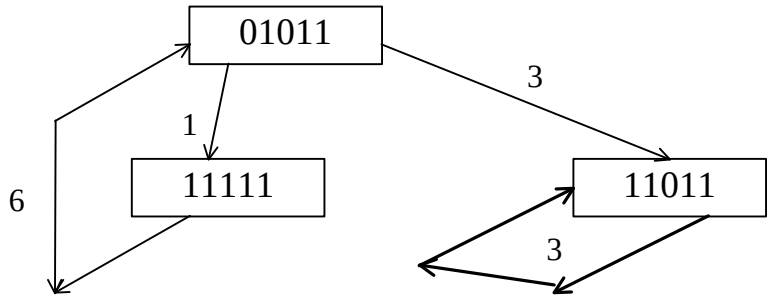
4: R2 ← (R1) + (R2)

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**nstr/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.