3. PROCESOARE PIPELINE SCALARE CU SET OPTIMIZAT DE INSTRUCŢIUNI

3.1. MODELUL RISC. CARACTERISTICI GENERALE.

Microprocesoarele RISC (**Reduced Instruction Set Computer)** – replică la **lipsa de eficienţă a modelului convenţional de procesor de tip CISC** (**Complex Instruction Set Computer).** Multe dintre instrucţiunile maşină ale procesoarelor CISC sunt foarte rar folosite în softul de bază, cel care implementează sistemele de operare, utilitarele, translatoarele, etc. Modelele CISC sunt caracterizate de un set foarte bogat de instrucţiuni - maşină, formate de instrucţiuni de lungime variabilă, numeroase moduri de adresare deosebit de sofisticate, etc. Evident că această **complexitate arhitecturală are repercursiuni negative asupra performanţei µP**.

Primele microprocesoare RISC s-au proiectat la Universităţile din Stanford (prof. *J. Hennessy*) şi respectiv Berkeley (prof. *D. Patterson*), din California, USA (1981).

Caracteristicile de bază ale modelului RISC sunt următoarele:

- **Timp de proiectare şi erori de construcţie mai reduse** decât la variantele CISC.
- Unitate de comandă hardware cablată ⇒ rata de execuţie a instrucţiunilor .
- Utilizarea tehnicilor de **procesare pipeline a instrucţiunilor ⇒ rată teoretică de execuţie de o instrucţiu2ne / ciclu**, pe modelele de procesoare care pot lansa în execuţie la un moment dat o singură instrucţiune (procesoare **scalare**).
- Implementarea unor arhitecturi avansate de memorie cache şi MMU (Memory Management Unit). De asemenea conţin mecanisme hardware de memorie virtuală bazate în special pe paginare ca şi sistemele CISC de altfel.
- **Set relativ redus de instrucţiuni simple**[1], majoritatea fără referire la memorie şi cu **puţine moduri de adresare – arhitectură tip LOAD / STORE**.
- **Format fix al instrucţiunilor**, codificate în general pe un singur cuvânt de 32

[1] Anumite instrucțiuni (INC, DEC, MOV) pot fi emulate cu ADD, instrucțiunea de calcul a restului REM poate fi implementată folosind teorema împărțirii cu rest iar instrucțiunile de salt condiționat (BNE, BEQ, BGT, …) le emulez folosind instrucțiunile de setare a condiției SNE, SEQ, SGT și BNEZ sau BEQZ.

biţi, mai recent pe 64 biţi (Alpha 21164, Power PC-620, etc.) ⇔ "***Simplicity favors regularity***" (Regularity makes implementation simpler and simplicity enables higher performance at lower cost).

- **Numărul mare de registre generale** este util şi pentru **mărirea spaţiului intern de procesare[2], model ortogonal de programare**, etc. Registrul R0 este cablat la masă.

- **"*Smaller is faster*".** Main memory: millions of locations, but slower. Registers (small size – 32 regs x 4B, or 32 regs x 7B) are faster to access than memory

---

- **"*Make the common case fast and the uncommon correct*".** Execuția cazurilor frecvente trebuie să se realizeze cu viteză ridicată pe când cazurile mai puțin frecvente trebuie executate corect (vezi și principiul Trace Scheduling și introducerea codurilor compensatoare**).**

3.2. SET DE INSTRUCŢIUNI.

Proiectarea setului de instrucţiuni aferent microprocesoarelor RISC presupune:

- **Compatibilitatea cu seturile de instrucţiuni ale altor procesoare pe care s-au dezvoltat produse software consacrate –** cerinţă **contradictorie cu creşterea de performanţă a sistemului[3]**.
- **Setul de instrucţiuni este în strânsă dependenţă cu tehnologia folosită**, care de obicei **limitează sever performanţele** (constrângeri legate de **aria de integrare**, **numărul de pini**, cerinţe restrictive particulare ale tehnologiei).
- **Minimizarea complexităţii unităţii de comandă precum şi minimizarea traficului procesor - memorie.**
- Setul de instrucţiuni maşină – ales ca suport pentru implementarea limbajelor HLL.

---

[2] Compilatorul identifică variabilele *hot* (frecvent folosite) pentru stocarea acestora în registre. Problema alocării registrelor este rezolvată prin analogie cu problema colorării hărților.

[3] Aplicațiile vechi (pe 32 de biți) nu rulează toate pe sisteme de operare pe 64 de biți. Aplicațiile grafice realizate în BorlandC 3.1 nu pot fi rulate pe Windows 7 sau Windows Vista, iar anumite aplicații care rulau cu succes pe Windows 7 și MinGW pe Windows 8 rulează doar scrise în CodeBlocks.

RISC I Berkeley

- set de 31 instrucţiuni: **aritmetico-logice**, **de acces la memorie**, **de salt / apel subrutine** şi **speciale** ⇒ simplificare a logicii de decodificare şi a unităţii de comandă.
- set logic de 32 regiştri generali pe 32 biţi.

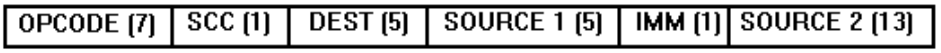

Figura 3.1. Formatul instrucţiunii la Berkeley I RISC.

Câmpul **IMM = 0** arată că cei mai puţini semnificativi (c.m.p.s.) 5 biţi ai câmpului SOURCE2 codifică al 2-lea registru operand.

Câmpul **IMM = 1** arată că SOURCE2 semnifică o constantă pe 13 biţi cu extensie semn pe 32 biţi.

Câmpul SCC (Store Condition Codes) semnifică validare / invalidare a activării indicatorilor de condiţie, corespunzător operaţiilor aritmetico-logice executate.

Mod de adresare	**Vax 11/780**	**Echivalent RISC 1**
Registru	Ri	Ri
Imediat	#Literal	S2 (literal pe 13 biti), IMM =1
Indexat	Ri + offset	Ri+S2(offset pe 13 biti)
Indirect-registru	(Ri)	Ri+S2, S2=0
Absolut	@#Adresa	R0+S2, R0=0

Tabelul 3.1. Modurile de adresare la Berkeley I RISC.

Următoarele 3 formate de instrucţiuni sunt întâlnite practic în toate arhitecturile RISC studiate (DLX, Intel 860, MIPS 88000, SPARC, APLHA, Power PC, HP PA):

instrucţiuni registru – registru (RS1, RS2, RDEST), de tip aritmetico – logice cu 2 operanzi

instrucţiuni registru – constantă imediată (RS1, constantă, RDEST) sunt instrucţiuni de transfer, aritmetico – logice, etc.

instrucţiuni de ramificaţie în program (JUMP / CALL, BRANCH, etc.)

3.3. ARHITECTURA SISTEMULUI DE MEMORIE LA PROCESOARELE RISC

Microprocesoarele RISC deţin un management intern de memorie (MMU-Memory Management Unit) în majoritatea implementărilor. MMU-ul are **rolul** de a **translata adresa virtuală emisă de către microprocesor într-o adresă fizică de acces la memoria principală** şi respectiv **de a asigura un mecanism de control şi protecţie - prin paginare sau / şi segmentare a memoriei - a accesului la memorie**. Sistemele RISC deţin **memorii cache interne şi externe cu spaţii în general separate pentru instrucţiuni şi date** (arhitecturi Harvard de memorie) şi **structuri de tip DWB (Data Write Buffer) cu rol de gestionare a scrierii efective a datelor în memoria sistem, prin degrevarea totală a procesorului** propriu - zis. DWB **capturează data şi adresa emise de către microprocesor şi execută efectiv scrierea** în memoria cache sau în cea principală, permiţând astfel microprocesorului să-şi continue activitatea fără să mai aştepte până când data a fost efectiv scrisă in memoria principală. Aşadar, **DWB se comportă ca un mic procesor de ieşire lucrând în paralel cu microprocesorul**.

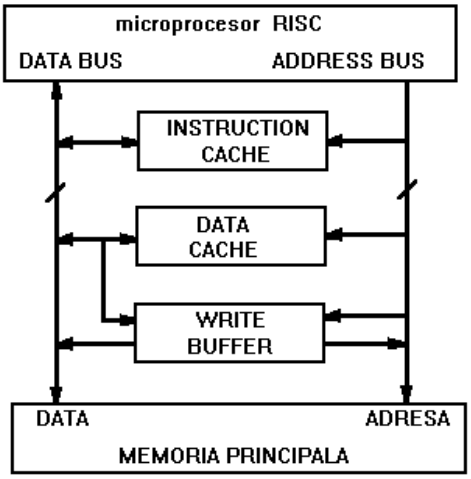

Figura 3.9. Arhitectura de memorie tip Harvard la procesoarele RISC

Problema majoră a sistemelor de memorie pentru ambele variante constă în decalajul tot mai accentuat între performanţa microprocesoarelor care creşte anual cu 50 -75% şi respectiv performanţa tehnologică a memoriilor (latenţa) care are o rată de creştere de doar 7% pe an. Prin urmare acest **"semantic gap" microprocesor - memorie reprezintă o provocare esenţială pentru îmbunătăţirea performanţelor arhitecturilor de memorie** aferente arhitecturilor actuale de

procesoare.

3.4. PROCESAREA PIPELINE ÎN CADRUL PROCESOARELOR SCALARE

Cea mai importantă caracteristică arhitecturală a acestor **microprocesoare RISC scalare** o constituie **procesarea pipeline** a instrucţiunilor şi datelor. Din punct de vedere istoric **primul calculator pipeline viabil** a fost supersistemul **CDC 6600 (firma Control Data Company, şef proiect *Seymour Cray,*1964)**.

Tehnica de procesare pipeline reprezintă o tehnică de **procesare paralelă** a informaţiei prin care un **proces secvenţial este divizat în subprocese**, fiecare subproces fiind executat într-un segment special dedicat şi care operează în paralel cu celelalte segmente. Fiecare segment execută o procesare parţială a informaţiei. Rezultatul obţinut în segmentul i este transmis în tactul următor spre procesare segmentului (i+1). Rezultatul final este obţinut numai după ce informaţia a parcurs toate segmentele, la ieşirea ultimului segment. Denumirea de **pipeline** provine de la **analogia cu o bandă industrială de asamblare**. **Diversele procese se pot afla în diferite faze de prelucrare în cadrul diverselor segmente, simultan**. **Suprapunerea procesărilor** este posibilă prin asocierea unui **registru de încărcare (latch – Ti)** fiecărui segment din pipeline. **Registrele Ti** produc o separare între segmente (**nivelele de prelucrare – Ni** – combinaţionale sau secvenţiale) a. Î. fiecare segment să poată prelucra date separate.

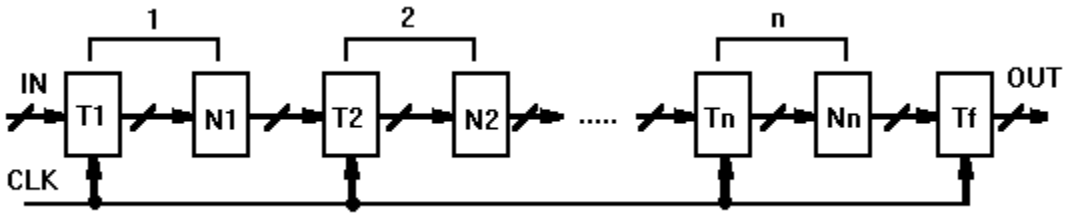

Figura 3.10. Structură de procesare tip pipeline.

Este evident că **nivelul cel mai lent de prelucrare va stabili viteza de lucru a benzii**. Aşadar, se impune în acest caz **partiţionarea unui eventual proces mai lent în subprocese cu timpi de procesare cvasiegali şi interdependenţe minime**. O soluţie

principială, de a compensa întârzierile diferite pe diversele nivele de prelucrare:

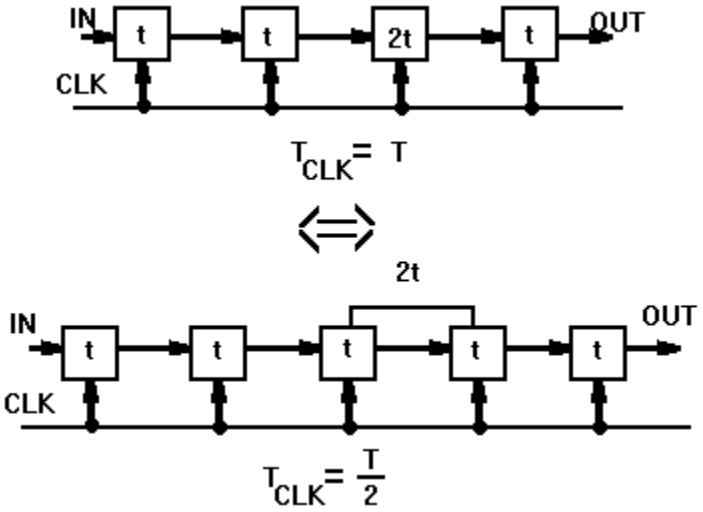

Figura 3.11. Balansarea unei structuri pipeline prin divizarea nivelului lent

Se defineşte **rata de lucru** R a unei benzi de asamblare ca fiind **numărul de procese executate în unitatea de timp T** (ciclu maşină sau tact). Considerând *m* posturi de lucru (**nivele pipeline**) prin care trec *n* procese (**instrucţiuni**), rezultă că banda le va executa într-un interval de timp **Dt = (m + n -1) * T**. Normal, având în vedere că **m*T** este timpul de "setup" al benzii, adică timpul necesar terminării primului proces. Rata de execuţie a unei benzi prin care trec **n** procese este:

$$\mathbf{Rn = n/ (m+n-1)T} \qquad (3.1)$$

Rata ideală este de un **proces (instrucţiune) per ciclu**, întrucât:

$$R = \lim_{n\to\infty} Rn = 1/T \qquad (3.2)$$

Această **rată teoretică nu se va putea atinge în practică**, nu numai datorită faptului că se prelucrează un **număr finit de procese** şi datorită **timpului de "setup"**, ci mai ales datorită unor **blocaje în funcţionarea normală a benzii** numite **hazarduri**.

3.4.2. PRINCIPIUL DE PROCESARE ÎNTR-UN PROCESOR PIPELINE

Procesarea pipeline a instrucţiunilor reprezintă o tehnică de procesare prin intermediul căreia **fazele (ciclii) aferente multiplelor instrucţiuni sunt suprapuse în**

timp. Se înţelege printr-o **fază aferentă unei instrucţiuni maşină o prelucrare atomică a informaţiei** care se desfăşoară după un algoritm implementat în hardware şi care durează unul sau mai mulţi tacţi. Microprocesoarele RISC uzuale deţin o structură pipeline de instrucţiuni întregi pe 4 - 6 nivele (microprocesoarele MIPS au 5 nivele):

1. Nivelul IF (instruction fetch) - se calculează adresa instrucţiunii ce trebuie citită din cache-ul de instrucţiuni sau din memoria principală şi se aduce instrucţiunea;

2. Nivelul RD (ID) - se decodifică instrucţiunea adusă şi se citesc operanzii din setul de regiştri generali. În cazul instrucţiunilor de salt, pe parcursul acestei faze se calculează adresa de salt;

3. Nivelul ALU - se execută operaţia ALU asupra operanzilor selectaţi în cazul instrucţiunilor aritmetico-logice; se calculează adresa de acces la memoria de date pentru instrucţiunile LOAD / STORE;

4. Nivelul MEM - se accesează memoria cache de date sau memoria principală, însă numai pentru instrucţiunile LOAD / STORE. Desigur că un ciclu cu MISS în cache pe acest nivel (ca şi pe nivelul IF de altfel), va determina stagnarea temporară a acceselor la memorie sau chiar a procesării interne. La scriere, problema aceasta nu se pune datorită procesorului de ieşire specializat DWB care lucrează în paralel cu procesorul central după cum deja am arătat.

5. Nivelul WB (write buffer) - se scrie rezultatul ALU sau data citită din memorie (în cazul unei instrucţiuni LOAD) în registrul destinaţie din setul de regiştri generali ai microprocesorului.

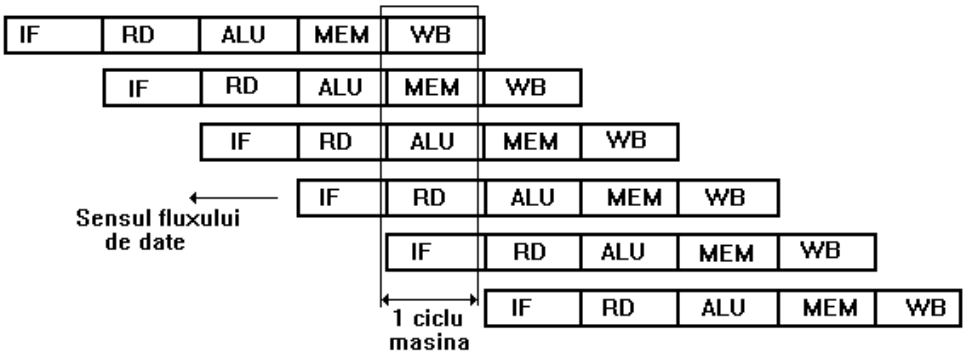

Figura 3.13. Principiul procesării pipeline într-un procesor RISC

Se observă imediat **necesitatea suprapunerii a 2 nivele concurenţiale: nivelul IF şi respectiv nivelul MEM, ambele cu referire la memorie.** În cazul microprocesoarelor RISC această situaţie se **rezolvă** deseori prin legături **(busuri) separate între procesor şi memoria de date respectiv de instrucţiuni (arhitectură Harvard).**

Deşi această împărţire pe 5 nivele este caracteristică mai multor microprocesoare RISC, ea are o deficienţă importantă şi anume că **nu orice instrucţiune trece prin toate cele 5 nivele de procesare**. **Proiectanţii anumitor microprocesore RISC** (de ex. microprocesorul HARP - Hatfield Advanced RISC Processor proiectat la Universitatea din Hertfordshire, Marea Britanie), **au comprimat nivelele ALU şi MEM într-unul singur**. In acest caz calculul adreselor de memorie se face în nivelul RD prin mecanisme care reduc acest timp de calcul.

Tipuri de parallelism in sistemele de calcul

A. Paralelism la nivel de bit

De la apariţia tehnologiei **VLSI** (Very Large Scale Integration) de fabricaţie a chipurilor, în anii **1970, până în aproximativ 1986**, arhitecturile de calculatoare au evoluat prin **dublarea dimensiunii cuvântului** (câtă informaţie poate un procesor să execute pe ciclu). **Creşterea dimensiunii cuvântului reduce numărul de instrucţiuni maşină care trebuie executate pentru a efectua operaţii pe variabile ale căror dimensiuni sunt mai mari decât lungimea cuvântului**. Istoric, microprocesoarele pe 4 biţi au fost înlocuite cu cele pe 8 biţi, apoi pe 16 biţi, apoi pe 32 de biţi sau chiar **procesoare vectoriale**. Această tendinţă a fost marcată de introducerea de procesoare pe 32 de biţi, care a fost un standard pentru arhitecturile de calcul, pentru două decenii. În anii 2003 - 2004, odată cu arhitecturile x86-64, procesoarele pe 64 de biţi au devenit un lucru obişnuit. Tendinţa (vezi IBM Cell pe cu regiştri pe 128 de biţi folosiţi ca vectori) este ca procesoarele să extindă setul de instrucţiuni (ISA) cu instrucţiuni vectoriale.

B. Paralelism la nivel de faze / instrucţiuni

O maşină RISC cu cinci faze pipeline (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Write back) poate fi considerată reprezentativă pentru **paralelismul la nivel de instrucţiuni (ILP** – Instruction Level Parallelism). **Un program este, în esenţă, un flux de instrucţiuni ce trebuie executate de către un procesor**. Aceste instrucţiuni pot fi **reordonate şi combinate în grupuri care apoi sunt executate în paralel**, fără a schimba rezultatul programului. Acest lucru este cunoscut sub denumirea de paralelism la nivel de instrucţiune. Progresul în domeniul ILP a dominat arhitecturile de calculator de la mijlocul anilor 1980 până la mijlocul anilor 1990.

Toate procesoarele moderne au mai multe faze de procesare pipeline. Fiecare fază pipeline corespunde unei acţiuni diferite pe care procesorul o efectuează asupra unei instrucţiuni. Cu alte cuvinte, un procesor pipeline cu N faze poate avea până la N instrucţiuni aflate în diferite faze de execuţie (**parelelism temporal la nivel de fază *pipeline***). Aşa cum am precizat mai sus, procesorul pipeline canonic este cu 5 faze de execuţie dar acest lucru nu este fix: Pentium 4 avea 31 de faze de execuţie.

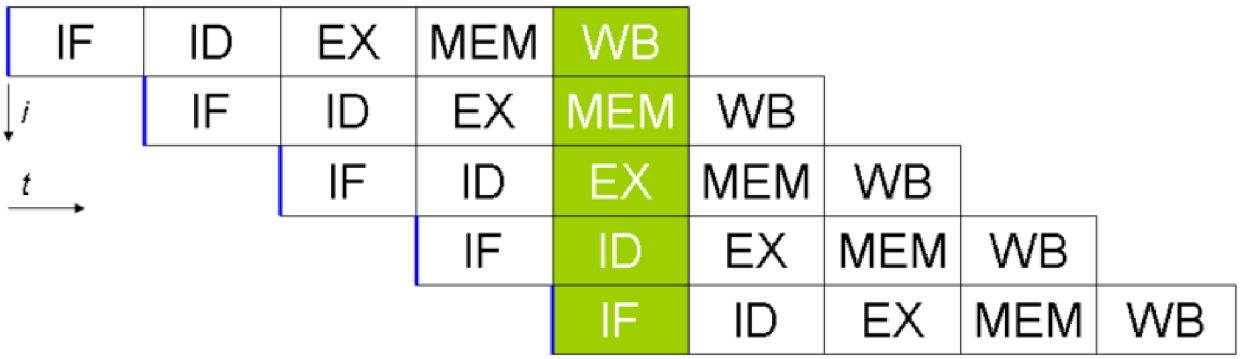

Fig. 2: Pipeline cu 5 faze

În plus faţă de paralelismul la nivel de instrucţiune adus de structurile pipeline, unele procesoare pot trimite spre execuţie mai mult de o instrucţiune la un moment dat. Acestea sunt cunoscute ca **procesoare superscalare** (**extind paralelismul temporal de tip pipeline la unul spaţial – presupune existenţa mai multor unităţi de execuţie care să lucreze simultan**). Instrucţiunile pot fi grupate împreună numai dacă nu există nici o dependenţă de date între ele. Algoritmul lui **Tomasulo** este unul dintre cei mai cunoscuţi şi utilizaţi algoritmi care permit exploatarea ILP şi execuţia out-of-order.

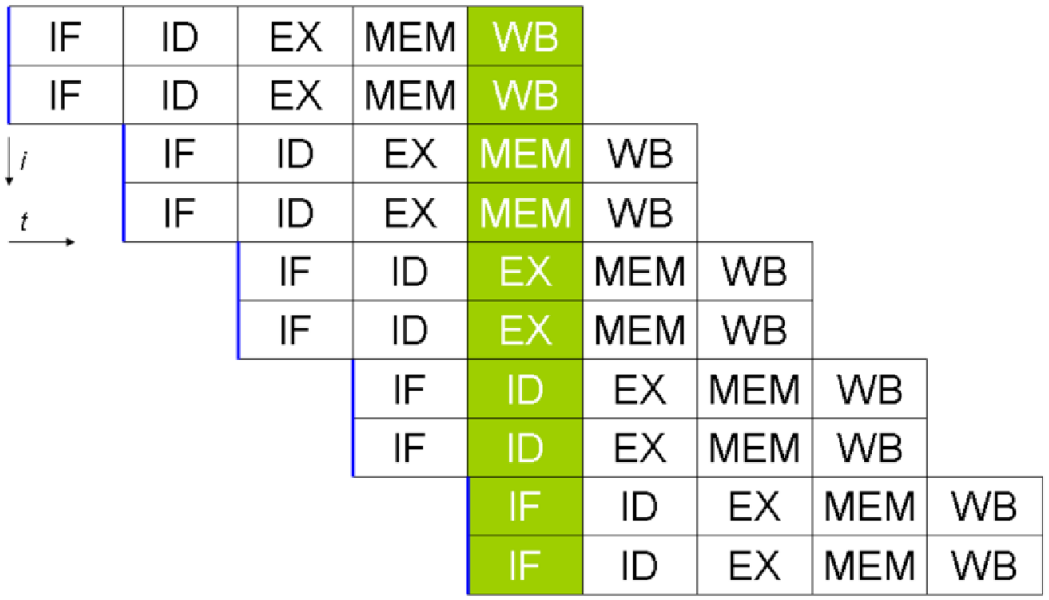

*Fig. 3: Pipeline cu 5 faze al unui procesor superscalar, capabil de a emite **două instrucţiuni pe ciclu**. Poate avea două instrucţiuni în fiecare etapă a pipeline, pentru un total de până la 10 instrucţiuni executate simultan.*

C. Paralelism la nivel de date

Paralelismul la nivel de date este paralelismul inerent din buclele de program, care se axează pe distribuirea datelor în diferite noduri de calcul pentru a fi prelucrate în paralel. **Aceleaşi operaţii se fac pe aceleaşi sau pe diferite seturi de date.**

„Parallelizing loops often leads to similar (not necessarily identical) operation sequences or functions being performed on elements of a large data structure." Multe aplicaţii ştiinţifice şi de inginerie prezintă paralelism la nivel de date. **O dependenţă la nivelul unei bucle de program** este dependenţa unei iteraţii a buclei de ieşirea produsă de una sau mai multe iteraţii precedente ale acelei bucle. Astfel de dependenţe **previn paralelizarea buclelor de program**. Cu cât dimensiunea problemei devine mai mare, cu atât cantitatea de paralelism de date disponibil, de obicei, creşte (**optimizare locală** vs. **optimizare globală**).

D. Paralelism la nivel de fire de execuţie (thread) / taskuri

Un "*procesor multithread*"" (PMT) deţine abilitatea de a procesa instrucţiuni provenite din thread-uri ("fire de execuţie") diferite, facilitând astfel execuţia programelor "multifir". **Un *thread* reprezintă o secvenţă atomică de program, concurentă, recunoscută de către sistemele de operare** (SO). Aceste **SO permit mai multor fire de execuţie să ruleze, alocându-le resursele necesare** în acest scop. **Coexistenţa mai multor thread-uri active permite exploatarea unui tip de paralelism numit "*Thread Level Parallelism*"** (TLP). Instrucţiunile din thread-uri diferite, fiind independente între ele,

se pot executa în paralel ceea ce implică grade superioare de utilizare ale resurselor precum şi mascarea latenţelor unor instrucţiuni. În acest ultim sens, de asemenea, gestiunea branch-urilor este simplificată, latenţa acestora putând fi (măcar parţial) acoperită de instrucţiuni aparţinând unor thread-uri diferite şi deci independente de condiţia de salt. De asemenea, efectul defavorabil al miss-urilor în cache-uri poate fi contracarat prin acest multithreading (dacă un thread generează un miss în cache de ex., CPU-ul - *Central Processing Unit* - poate continua procesele de aducere ale instrucţiunilor din cadrul celorlalte thread-uri). Aşadar "groapa" semantică între conceptele de procesare multithreading a aplicaţiilor HLL şi procesorul hardware convenţional este umplută tocmai de PMT.

Single-core superscalar processors cannot fully exploit TLP (fine grain, course grain, SMT – permits multiple independent threads of control to execute SIMULTANEOUSLY on the SAME core; e.g. if one thread is waiting for a long-latency operation to complete, another thread can use the integer units). In the SMT mode, some processor structures (i.e. issue queue, physical register files, functional units, caches) are shared among the threads, and others (rename tables, ROBs, Load/Store Queues, branch predictors) are private to each thread.

Paralelismul la nivel de taskuri este caracteristica unui program paralel de a permite **realizarea de operaţii complet diferite fie pe aceleaşi seturi de date sau pe seturi diferite**. Acest lucru este opus cu paralelismul la nivel de date, unde acelaşi operaţii se fac pe aceleaşi sau diferite seturi de de date. Paralelismul la nivel de taskuri nu este scalabil, de obicei, cu dimensiunea problemei [Cul99].

3.4.3. STRUCTURA PRINCIPIALĂ A UNUI PROCESOR RISC. Implementare pipeline

Stagiile de interconectare ale nivelelor structurii (IF / ID, ID / EX, EX / MEM, MEM / WB) sunt implementate sub forma unor regiştri de încărcare, actualizaţi sincron cu fiecare nou ciclu de procesare. Memorarea anumitor informaţii de la un nivel la altul este absolut necesară, pentru ca informaţiile conţinute în formatul instrucţiunii curente să nu se piardă prin suprapunerea fazelor de procesare.

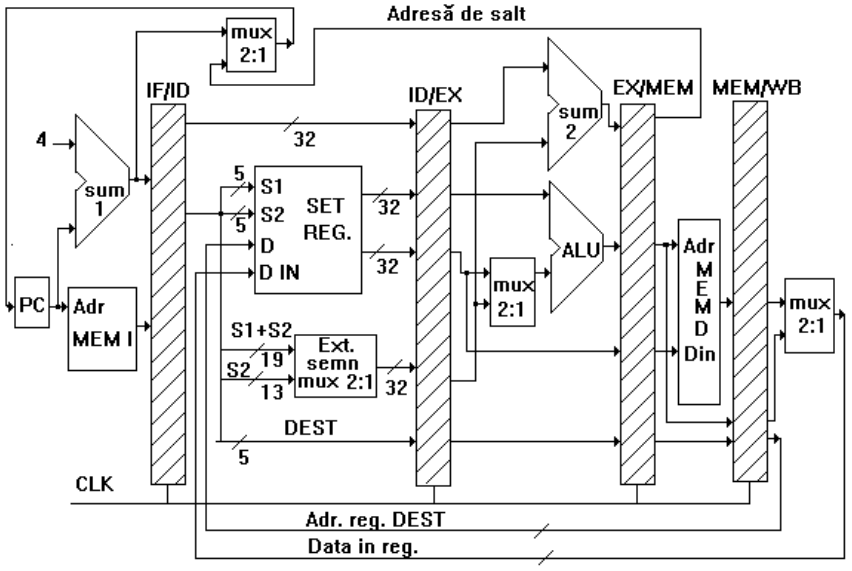

Figura 3.14. Schema de principiu a unui procesor RISC

Exemplu: ce s-ar întâmpla dacă **câmpul DEST** nu ar fi memorat succesiv din nivel în nivel şi rebuclat la intrarea setului de regiştri generali? **Setul de regiştri generali trebuie să permită simultan două citiri şi o scriere**.

3.4.4. PROBLEMA HAZARDURILOR ÎN PROCESOARELE RISC

Hazardurile constituie acele situaţii care pot să apară în procesarea pipeline şi care pot determina blocarea (stagnarea) procesării, având deci o influenţă negativă asupra ratei de execuţie a instrucţiunilor. Conform unei clasificări consacrate aceste hazarduri sunt de 3 categorii : **hazarduri structurale, de date şi de ramificaţie.**

3.4.4.1. HAZARDURI STRUCTURALE (HS): PROBLEME IMPLICATE ŞI SOLUŢII

Sunt determinate de **conflictele la resurse comune. Eliminarea prin hardware presupune multiplicarea acestor resurse**.

Exemplu: un **procesor care are un set de regiştri generali de tip uniport şi în anumite situaţii există posibilitatea ca 2 procese să dorească să scrie în acest set simultan**. Figura 3.15 descrie o structură ALU implementată la **microprocesorul RISC**

superscalar HARP care permite 4 operaţii ALU simultane. Prin **partiţionarea timpului afectat nivelului WB în două**, în cadrul acestui nivel se pot face **două scrieri (în prima jumătate a nivelului WB se înscrie în setul de regiştri conţinutul căii A, iar în a doua parte a nivelului WB se înscrie conţinutul căii B**).

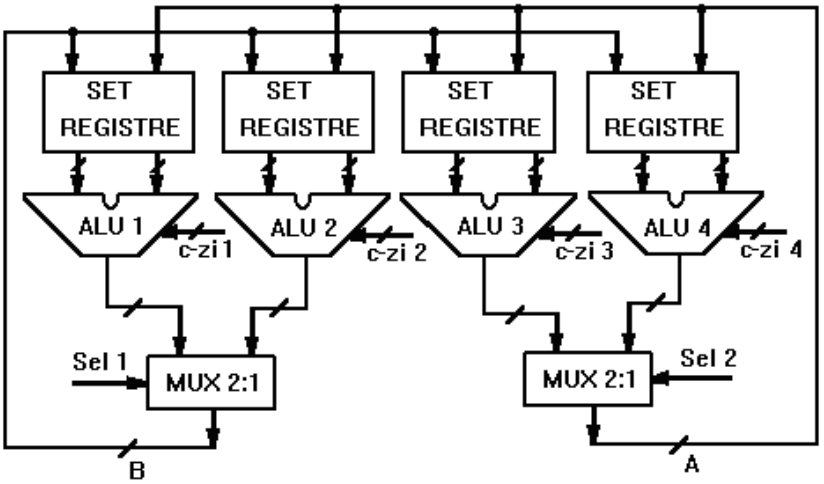

Figura 3.15. Structură de procesare multiplă.

Exemplu: **accesul simultan la memorie a 2 procese distincte:** unul de aducere a instrucţiunii **(IF),** iar celălalt de aducere a operandului sau scriere a rezultatului în cazul unei instrucţiuni LOAD / STORE **(nivelul MEM).** Se rezolvă **printr-o arhitectură Harvard a busurilor şi cache-urilor**. Totuşi, există **microprocesoare care deţin busuri şi cache-uri unificate pe instrucţiuni şi date** (Power PC 601).

Structural Hazards - Solutions

o Stall

- Low Cost, Simple (+)
- Increases CPI (–)
- Try to use for rare events in high-performance CPUs

o Duplicate Resources

- Decreases CPI (+)
- Increases cost (area), possibly cycle time (–)
- Use for cheap resources, frequent cases
- Separate I-, D-caches, Separate ALU/PC adders, Reg File Ports

o Pipeline Resources

- High performance (+)
- Control is simpler than duplication (+)
- Tough to pipeline some things (RAMs) (–)
- Use when frequency makes it worthwhile
- Ex: Fully pipelined FP add/multiplies critical for scientific

3.4.4.2. HAZARDURI DE DATE: DEFINIRE, CLASIFICARE, SOLUŢII DE EVITARE A EFECTELOR DEFAVORABILE

Apar când **o instrucţiune depinde de rezultatele unei instrucţiuni anterioare** în bandă. Clasificare în **3 categorii**, **dependent de ordinea acceselor de citire respectiv scriere**, în cadrul instrucţiunilor.

Considerând instrucţiunile **i** şi **j** succesive, **hazardul RAW** (Read After Write) apare atunci când instrucţiunea **j** încearcă să citească o sursă înainte ca instrucţiunea **i** să scrie în aceasta. Apare deosebit de **frecvent** în implementările actuale de procesoare pipeline. Să considerăm secvenţa de instrucţiuni de mai jos procesată într-o structură pe 5 nivele, ca în figura următoare. Se observă că data ce urmează a fi incărcată în R5 este disponibilă doar la finele nivelului MEM aferent instrucţiunii I1, prea târziu pentru procesarea corectă a instrucţiunii I2 care ar avea nevoie de această dată cel târziu la începutul nivelului său ALU. Aşadar, pentru o procesare corectă, I2 trebuie stagnată cu un ciclu maşină.

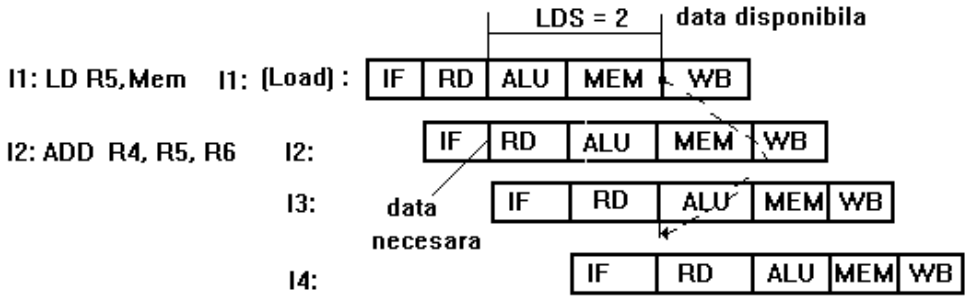

Figura 3.20. Hazard RAW în cazul unei instrucţiuni LOAD

Soluţie software – presupune **inserarea unei instrucţiuni NOP între I1 şi I2**

(umplerea "load delay slot"-ului - LDS) sau o altă instrucţiune utilă independentă de instrucţiunea I1. **Stagnarea hardware** a instrucţiunii I2 datorată **detecţiei hazardului RAW de către unitatea de control –** tehnica "**scoreboarding**" propusă de către ***Seymour Cray*** **(1964 - CDC 6600)**. Se impune ca **fiecare registru al procesorului** din setul de regiştri să aibă un "**bit de scor**" asociat. Dacă bitul este zero, registrul respectiv e disponibil, dacă bitul este 1, registrul respectiv este ocupat. Dacă pe un anumit nivel al procesării este necesar accesul la un anumit registru având bitul de scor asociat pe 1, respectivul nivel va fi întârziat, permiţându-i-se accesul doar când bitul respectiv a fost şters de către procesul care l-a setat. **Soluţiile bazate pe stagnarea fluxului (software - NOP sau hardware - semafoare) au acelaşi efect defavorabil asupra performanţei.**

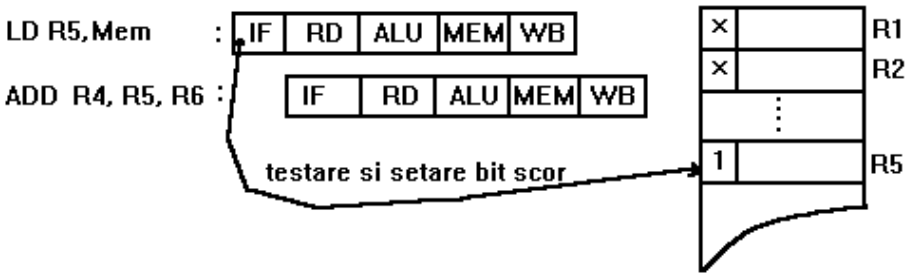

Figura 3.21a. Detecţie hazard pe baza bitului de scor

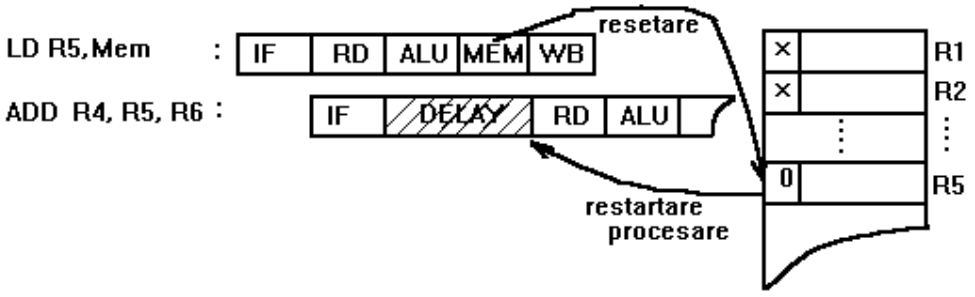

Figura 3.21b. Restartarea procesării instrucţiunilor

Rezolvare prin hardware a hazardului RAW fără a cauza stagnări ale fluxului de procesare – **forwarding** bazate pe "**pasarea anticipată" a rezultatului instrucţiunii i, nivelului de procesare aferent instrucţiunii j care are nevoie de acest rezultat**.

ADD R1, R2, R3 ; R1 <------ (R2) + (R3)

SUB R4, R1, R5 ; R4 <------ (R1) - (R5)

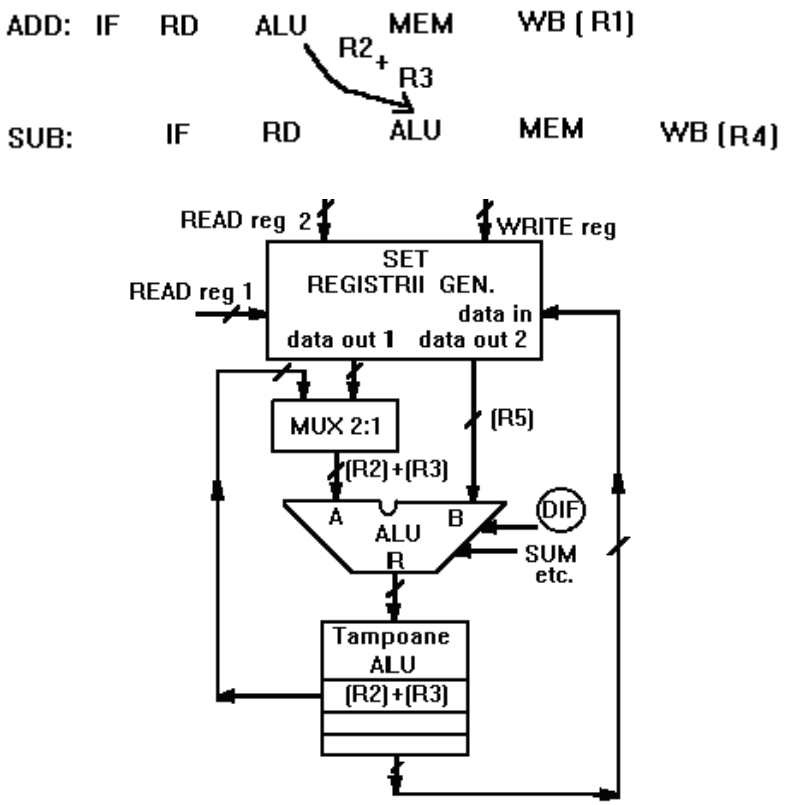

Figura 3.22. Implementarea "forwarding"-ului

Rezultatul ALU aferent primei instrucţii (R2 + R3) este memorat în tampoanele ALU la finele fazei ALU a instrucţiunii ADD. **Dacă unitatea de control va detecta hazardul RAW, va selecta pe parcursul fazei ALU aferente instrucţiunii SUB la intrarea A a ALU, tamponul care conţine (R2) + (R3)**, evitând astfel hazardul RAW. Este necesară i**mplementarea proceselor de forwarding nu numai de la ieşirile ALU spre intrări ci şi din cadrul nivelului următor (MEM) spre intrările ALU**.

Hazardul WAR (Write After Read) poate să apară atunci când instrucţiunea **j** scrie o destinaţie înainte ca aceasta să fie citită pe post de sursă de către o instrucţiune anterioară **i**. **Hazardurile WAR** pot apare şi datorită execuţiei instrucţiunilor în afara ordinii lor normale, din program (execuţie **Out of Order**). Această **procesare Out of Order este impusă de creşterea performanţei** si se poate realiza atât prin mijloace

hardware cât şi software, legat de optimizarea programelor pe arhitecturile pipeline.

Ex.: I1: MULF Ri, **Rj**, Rk; Ri <--- (**Rj**) * (Rk)

I2: ADD **Rj**, Rp, Rm; **Rj** <--- (Rp) + (Rm)

Figura 3.23. Apariţia unui hazard WAR

Instrucţiunea I1 (de coprocesor flotant) se va încheia în execuţie după I2 care este o instrucţiune de procesor cu operanzi întregi, întrucât numărul de nivele aferent structurii pipeline a coprocesorului este mai mare decât numărul de nivele aferent procesorului. ⇒ I1, I2 se termină "Out of Order" (I2 înaintea lui I1).

Secvenţa reorganizată prin software, care elimină hazardul WAR este :

MULF Ri, Rj, Rk

ADD Rx, Rp, Rm

. . . . .

MOV Rj, Rx : Rj <- Rx (după Ri <- (Rj) x (Rk))

Hazardul WAW (Write After Write **– slow operation followed by fast operation**), apare atunci când instrucţiunea **j** scrie un operand înainte ca acesta să fie scris de către instrucţiunea **i –** scrierile făcându-se într-o ordine eronată. **Hazardul WAW poate apărea în structurile care permit unei instrucţiuni să fie procesată chiar dacă o instrucţiune anterioară este blocată, sau in cazul execuţiei Out of Order a instrucţiunilor care au aceeaşi destinaţie**.

Ex.: I1: DIVF **Rj**, Ri, Rk; **Rj** <--- (Ri) / (Rk)

I2: ADD **Rj**, Rp, Rm; **Rj** <--- (Rp) + (Rm)

Figura 3.24 Apariţia unui unui hazard WAW

Hazardurile de tip **WAW sau WAR nu reprezintă hazarduri reale**, ci mai degrabă conflicte de nume. Ele **pot fi eliminate de către compilator** (scheduler) **prin redenumirea resurselor utilizate de program**. Se mai numesc **dependenţe de ieşire**

respectiv **antidependenţe**.

BUFFER-UL DE REDENUMIRE (*RENB*)

În urma decodificării unei instrucţiuni, rezultatul acesteia este asignat unei locaţii din buffer-ul de redenumire [Ung99], iar numărul registrului destinaţie este asociat acestei locaţii. În acest mod, registrul destinaţie este practic redenumit printr-o locaţie din RenB. Rezultatul instrucţiunii este disponibil abia în momentul în care unitatea de execuţie corespunzătoare îl generează. În urma decodificării se creează prin hard un "tag" care reprezintă numele unităţii de execuţie care va procesa rezultatul instrucţiunii respective. Acest tag va fi scris în aceeaşi locaţie din RenB. Din acest moment, când o instrucţiune următoare face referire la respectivul registru pe post de operand sursă, ea va apela în locul acestuia valoarea înscrisă în RenB sau, dacă valoarea nu a fost încă procesată, tag-ul aferent locaţiei. Dacă mai multe locaţii din RenB conţin acelaşi număr de registru (mai multe instrucţiuni în curs au avut acelaşi registru destinaţie, într-o secvenţă), se va înscrie (în *fpr* sau *gpr*) locaţia din RenB cea mai recent înscrisă (tag sau valoare) – eliminându-se astfel hazardurile WAR sau WAW.

RenB se implementează sub forma unei memorii asociative, căutarea făcându-se după numărul registrului destinaţie la scriere, respectiv sursă la citire. Dacă accesarea RenB se soldează cu miss, atunci operandul sursă va fi citit din setul de regiştri. În caz de hit, valoarea sau tag-ul citite din RenB sunt memorate în SR corespunzătoare. **Când o unitate de execuţie generează un rezultat, acesta se va înscrie în SR şi în locaţia din RenB care au tag-ul identic cu cel emis de către respectiva unitate**. Rezultatul înscris într-o SR poate debloca anumite instrucţiuni aflate în aşteptare. După ce rezultatul a fost scris în RenB, instrucţiunile următoare vor continua să-l citească din RenB ca operand sursă până când va fi evacuat şi scris în setul de regiştri (în faza **commit/write-back** de procesare). Redenumirea unui registru cu o locaţie din RenB se termină prin evacuarea acestei locaţii.

3.4.4.3. HAZARDURI DE RAMIFICAŢIE (HR): PROBLEME IMPLICATE ŞI SOLUŢII

Sunt **generate de către instrucţiunile de ramificaţie** (branch). Cauzează pierderi de perfomanţă în general mai importante decât hazardurile structurale şi de date. **Efectele defavorabile** ale instrucţiunilor de ramificaţie pot fi **reduse** prin **metode soft** (**reorganizarea programului sursă**), sau prin **metode hard** care determină în avans dacă saltul se va face sau nu (**branch prediction**) şi calculează în avans noul PC.

Predicţiile eronate:

- Cauzează **consum suplimentar de putere** a procesorului prin execuţia instrucţiunilor de pe calea greşit predicţionată şi reluarea procesării de pe calea corectă (ciclu suplimentar presupune modificarea unor valori în registre, structuri pipeline, buffer-e, cache realizate prin comutarea tranzistorilor dintr-o stare în alta => 0→1 sau 1→0 => consum suplimentar de putere).
- **Cresc timpul de execuţie** diminuând performanţa globală a arhitecturii, conducând şi la un **consum de energie suplimentar. t ↑ => E ↑**

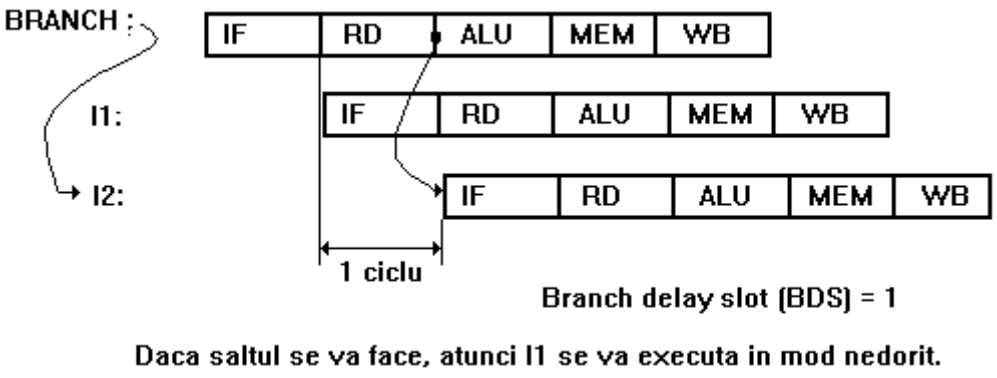

Figura 3.26. Hazardul de ramificaţie pe un procesor scalar

Soluţie pentru o procesare corectă, ar fi aceea de a **dezvolta unitatea de control hardware în vederea detectării prezenţei saltului şi de a întârzia procesarea**

instrucţiunilor următoare cu un număr de cicli egal cu latenţa BDS-ului, până când adresa de salt devine disponibilă. **Soluţia implică desigur reducerea performanţelor**. Acelaşi efect l-ar avea şi "umplerea" BDS-ului de către scheduler cu instrucţiuni NOP în vederea întârzierii necesare. **Soluţionarea prin software a ramificaţiilor de program se bazează pe reorganizarea secvenţei de instrucţiuni maşină, astfel încât efectul defavorabil al salturilor să fie eliminat.**

Exemplu de reorganizare în vederea eliminării efectului BDS- ului, considerând că BDS- ul instrucţiunii de salt este doar de 1 tact. **Rata de execuţie se îmbunătăţeşte întotdeauna, indiferent dacă saltul se face ori nu se face.**

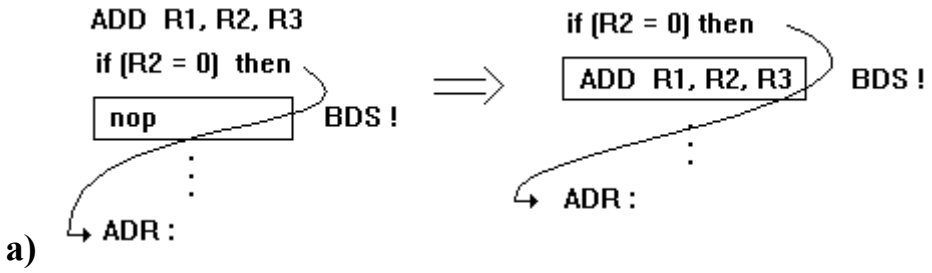

Figura 3.28. Soluţionarea optimă

Dacă această reorganizare nu e posibilă, se încearcă una dintre următoarele 2 variante.

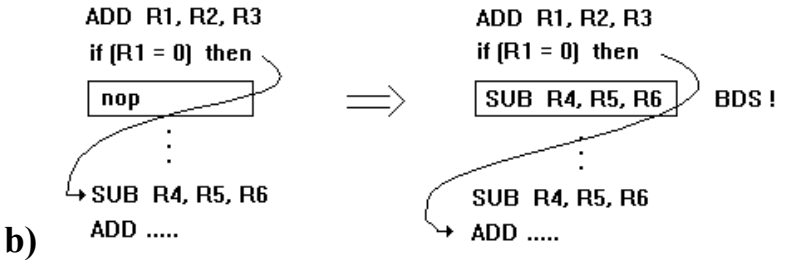

Figura 3.29. Soluţie utilă când saltul se face preponderent

În acest caz, **rata de execuţie creşte doar atunci cînd saltul condiţionat se face**. Dacă saltul nu se face, trebuie ca instrucţiunea introdusă în BDS (SUB R4, R5, R6), să nu provoace execuţia eronată a ramurii de program respective. **Dezavantaj**: **zona de cod** şi necesită **timp de execuţie sporit** cu o instrucţiune adiţională când saltul **nu** se face.

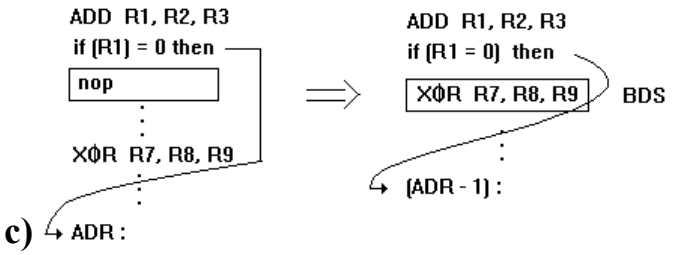

Figura 3.30. Soluţie utilă când saltul nu se face preponderent

Acest ultim caz, creşte **performanţa doar atunci când saltul condiţionat nu se face**. Este necesar ca instrucţiunea introdusă în BDS, să nu provoace execuţia eronată a ramurii de program în cazul în care saltul se face. Se urmăreşte **"umplerea" BDS-ului cu instrucţiuni utile şi care să nu afecteze programul din punct de vedere logic**.

Strategiile hardware de predicţie a branch-urilor au la bază un proces de predicţie "run - time" a ramurii de salt condiţionat precum şi determinarea în avans a noului PC. Ele sunt comune procesoarelor scalare dar şi superscalare. **Avantaj: se elimină necesitatea reorganizărilor soft ale programului sursă şi se obţine o independenţă faţă de maşină.**

Necesitatea predicţiei, mai ales în cazul procesoarelor cu execuţii multiple ale instrucţiunilor este imperios necesară. Notând cu **BP (Branch Penalty) numărul mediu de cicli de aşteptare pentru fiecare instrucţiune din program**, **introduşi de salturile fals predicţionate**, se poate scrie relaţia:

BP= C (1-Ap) b IR **(3.5)**

C= Numărul de cicli de penalizare introduşi de un salt prost predicţionat
Ap= Acurateţea predicţiei
b= Procentajul instrucţiunilor de salt, din totalul instrucţiunilor
IR= Rata medie de lansare în execuţie a instrucţiunilor

Se observă că BP(Ap=0)=C b IR, iar BP(Ap=1)=0 (**predicţie ideală**). Impunând un BP=0.1 şi considerând valorile tipice: C=5, IR=4, b=22.5%, ⇒ ca fiind necesară o acurateţe a predicţiei de peste **97.7%** ! ⇒ **IR=4/1.4=2.8 instr./ciclu, faţă de IR=4 instr./ciclu**, la o predicţie perfectă (Ap=100%).

1. **SCHEME CLASICE DE PREDICŢIE (IMPLEMENTATE) ÎN PROCESOARELE SUPERSCALARE AVANSATE**
   - PREDICTOR DE SALTURI DE TIP BTB
   - PREDICTOR DE SALTURI ADAPTIV, CORELAT PE DOUĂ NIVELURI (GAg)
2. **OPTIMIZAREA SCHEMELOR DE PREDICŢIE PENTRU RAMIFICAŢIILE DE PROGRAM ÎN PROCESOARELE SUPERSCALARE AVANSATE (stadiu de SIMULATOARE)**
   - PREDICTOR *MARKOVIAN* DE RAMIFICAŢII PROGRAM (Predictor contextual de tip PPM complet)
   - DETECTIA SI IZOLAREA SALTURILOR *DIFICIL DE PREZIS INTR-UN ANUMIT CONTEXT*. PREDICŢIA ACESTORA PRIN INTERMEDIUL UNOR **PREDICTOARE NEURONALE** DE RAMIFICAŢII PROGRAM (*perceptronul multistrat, perceptron simplu, perceptron fast path-based*)

[Cul99] Culler D., Singh J., Gupta A., *Parallel Computer Architecture - A Hardware/Software Approach*, Morgan Kaufmann Publishers, 1999

[Pat04] Patt Y., *The Microprocessor Ten Years From Now: What Are The Challenges, How Do We Meet Them?* Distinguished Lecturer talk at Carnegie Mellon University, Aprilie 2004