4. PROCESOARE CU EXECUŢII MULTIPLE ALE INSTRUCŢIUNILOR [7]

4.1. CONSIDERAŢII GENERALE. PROCESOARE SUPERSCALARE ŞI VLIW (EPIC)

Un deziderat ambiţios este acela de se atinge rate medii de procesare de mai multe instrucţiuni per tact. Procesoarele care iniţiază execuţia mai multor operaţii simultan intr-un ciclu (sau tact) se numesc procesoare cu execuţii multiple ale instrucţiunilor. Un astfel de procesor aduce din cache-ul de instrucţiuni una sau mai multe instrucţiuni simultan şi le distribuie spre execuţie în mod dinamic sau static (prin reorganizatorul de program), multiplelor unităţi de execuţie.

Principiul acestor procesoare paralele numite şi "maşini cu execuţie multiplă" (MEM) constă în existenţa mai multor unităţi de execuţie paralele, care pot avea latenţe diferite. Pentru a facilita procesarea acestor instrucţiuni, acestea sunt codificate pe un singur cuvânt de 32 sau 64 de biţi uzual, pe modelul RISC anterior prezentat. Dacă decodificarea instrucţiunilor, detecţia dependenţelor de date dintre ele, rutarea şi lansarea lor în execuţie din bufferul de prefetch înspre unităţile funcţionale se fac prin hardware, aceste procesoare MEM se mai numesc şi superscalare.

Pot exista mai multe unităţi funcţionale distincte, dedicate de exemplu diverselor tipuri de instrucţiuni tip întreg sau flotant. Aşadar execuţiile instrucţiunilor întregi, se suprapun cu execuţiile instrucţiunilor flotante (FP-Flotant Point). În cazul procesoarelor MEM, paralelismul temporal determinat de procesarea pipeline se suprapune cu un paralelism spaţial determinat de existenţa mai multor unităţi de execuţie. În general structura pipeline a coprocesorului are mai multe nivele decât structura pipeline a procesorului ceea ce implică probleme de sincronizare mai dificile decât în cazul procesoarelor pipeline scalare. Acelaşi lucru este valabil şi între diferite alte tipuri de

instrucţiuni având latenţe de execuţie diferite. Caracteristic deci procesoarelor superscalare este faptul că dependenţele de date între instrucţiuni se rezolvă prin hardware, în momentul decodificării instrucţiunilor. Modelul ideal de procesare superscalară, în cazul unui procesor care poate aduce şi decodifica 2 instrucţiuni simultan este prezentat în figura 4.1.

Este evident că în cazul superscalar complexitatea logicii de control este mult mai ridicată decât în cazul pipeline scalar, întrucât detecţia şi sincronizările între structurile pipeline de execuţie cu latenţe diferite şi care lucrează în paralel devin mult mai dificile. De exemplu un procesor superscalar având posibilitatea aducerii şi execuţiei a "n" instrucţiuni maşină simultan, necesită n(n-1)/2 unităţi de detecţie a hazardurilor de date între aceste instrucţiuni (comparatoare digitale), ceea ce conduce la o complexitate ridicată a logicii de control.

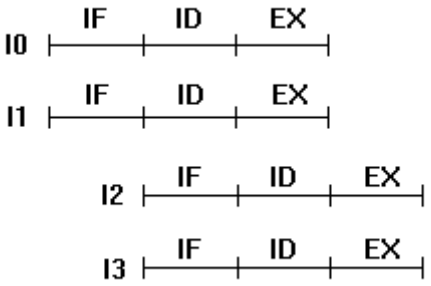

Figura 4.1. Modelul execuţiei superscalare

S-ar putea deci considera aceste procesoare MEM ca fiind arhitecturi de tip MIMD (Multiple Instructions Multiple Data) în taxonomia lui *Michael Flynn*. De remarcat totuşi că în această categorie sunt introduse cu precădere sistemele multiprocesor care exploatează paralelismul la nivelul mai multor aplicaţii (coarse grain parallelism), arhitecturile RISC ca şi cele de tip MEM exploatând paralelismul instrucţiunilor la nivelul aceleiaşi aplicaţii (fine grain parallelism). Desigur că - din punctul de vedere al acestei taxonomii - arhitecturile pipeline scalare (RISC), ar fi încadrabile în clasa SISD (Single Instruction Single Data), fiind deci incluse în aceeaşi categorie cu procesoarele

cele mai convenţionale (secvenţiale), ceea ce implică o slăbiciune a acestei sumare taxonomii.

În figura 4.2 se prezintă o structură tipică de procesor superscalar care deţine practic 2 module ce lucrează în paralel: un procesor universal şi un procesor destinat operaţiilor în virgulă mobilă. Ambele module deţin unităţi de execuţie proprii având latenţe diferite. La anumite microprocesoare superscalare regiştrii CPU sunt diferiţi de regiştrii FP, pentru a se reduce hazardurile structurale (în schimb creşteri serioase ale costurilor şi dificultăţi tehnologice) iar la altele (de ex. Motorola 88100), regiştrii CPU sunt identici cu cei ai coprocesorului. De exemplu, pentru eliminarea hazardurilor structurale, multe dintre aceste microprocesoare nu deţin "clasicul" registru al indicatorilor de condiţie. Salturile condiţionate se realizează prin compararea pe o anumită condiţie, a 2 dintre registrele codificate în instrucţiune. Hazardurile structurale la resursele hardware interne se elimină prin multiplicarea acestora şi sincronizarea adecvată a proceselor.

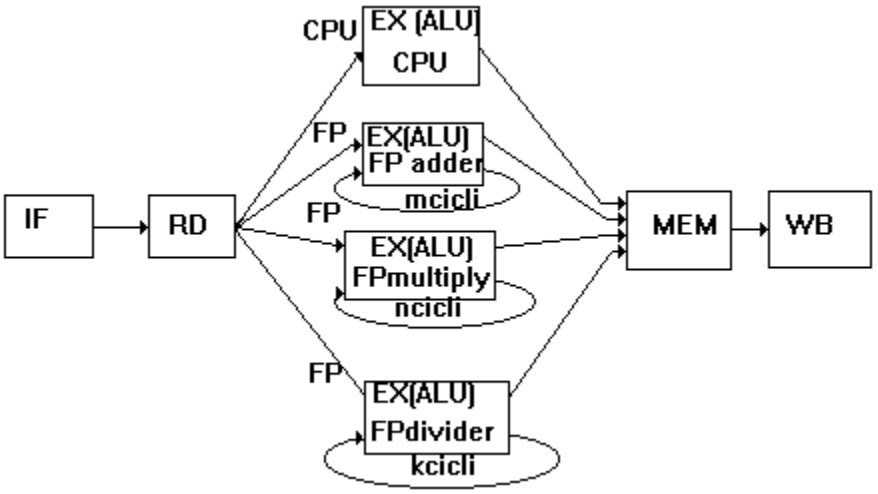

Figura 4.2. Structură de procesor superscalar pipeline

De remarcat că procesoarele superscalare, determină apropierea ratei de execuţie la una sau, în cazul în care se pot aduce mai multe instrucţiuni simultan, la mai multe instrucţiuni per ciclu. Dificultăţile de sincronizare sporite, se rezolvă prin utilizarea unor tehnici hardware bazate pe "scoreboarding" deosebit de sofisticate. Majoritatea

microprocesoarelor RISC actuale sunt de tip superscalar (conţin cel puţin un coprocesor integrat în chip). Un procesor superscalar care aduce din cache-ul de instrucţiuni mai multe instrucţiuni primitive simultan, poate mări rata de procesare la 1.2-2.3 instr./ciclu măsurat pe o mare diversitate de benchmark-uri, la nivelul realizărilor practice între anii 1995-1998. Exemple remarcabile de microprocesoare superscalare comerciale de tip RISC, sunt: INTEL 960 CA, SUN SuperSPARC, MPC 601, 603, 620 (POWER PC), etc. Microprocesoarele Intel Pentium, AMD K6, etc., sunt practic procesoare având model de programare CISC dar execuţie hardware superscalară.

Procesoarele VLIW (Very Long Instruction Word) reprezintă procesoare care se bazează pe aducerea în cadrul unei instrucţiuni multiple a mai multor instrucţiuni RISC independente pe care le distribuie spre procesare unităţilor de execuţie. Aşadar, rata de execuţie ideală la acest model, este de n instrucţiuni/ciclu. Pentru a face acest model viabil, sunt necesare instrumente soft de exploatare a paralelismului programului, bazate pe gruparea instrucţiunilor simple independente şi deci executabile în paralel, în instrucţiuni multiple. Arhitecturile VLIW sunt tot de tip MEM. Principiul VLIW este sugerat în figura 4.3:

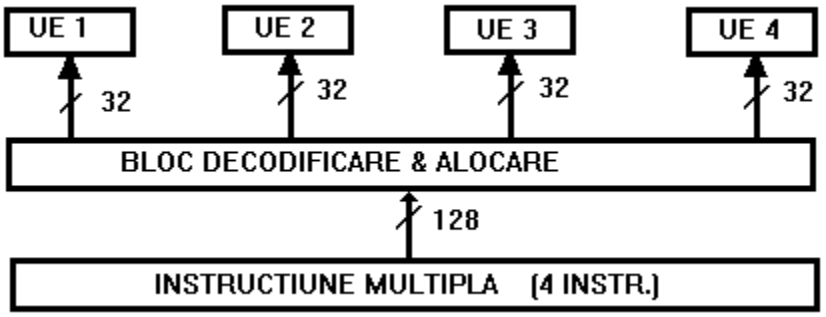

Figura 4.3. Decodificarea şi alocarea instrucţiunilor într-un procesor VLIW

În cadrul acestui model, se încearcă prin transformări ale programului, ca instrucţiunile RISC primitive din cadrul unei instrucţiuni multiple să fie independente şi deci să se evite hazardurile de date între ele, a căror gestionare ar fi deosebit de dificilă în acest caz. Performanţa procesoarelor VLIW este esenţial determinată de programele

de compilare şi reorganizare care trebuie să fie deosebit de "inteligente". De aceea acest model de arhitectură se mai numeşte uneori şi EPIC (Explicitly Parallel Instruction Computing).

Prin urmare, în cazul modelului de procesor VLIW, compilatorul trebuie să înglobeze mai multe instrucţiuni RISC primitive independente în cadrul unei instrucţiuni multiple, în timp ce în cazul modelului superscalar, rezolvarea dependenţelor între instrucţiuni se face prin hardware, începând cu momentul decodificării acestor instrucţiuni. De asemenea, poziţia instrucţiunilor primitive într-o instrucţiune multiplă determină alocarea acestor instrucţiuni primitive la unităţile de execuţie, spre deosebire de modelul superscalar unde alocarea se face dinamic prin control hardware. Acest model de procesor nu mai necesită sincronizări şi comunicaţii de date suplimentare între instrucţiunile primitive după momentul decodificării lor, fiind astfel mai simplu din punct de vedere hardware decât modelul superscalar. Un model sugestiv al principiului de procesare VLIW este prezentat în figura 4.4.

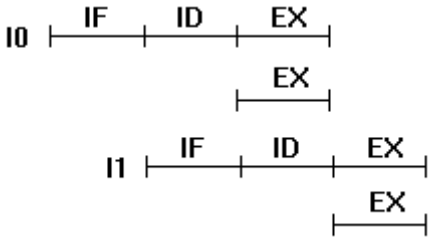

Figura 4.4. Principiul de procesare VLIW

Pentru exemplificarea principiului de procesare MEM, să considerăm secvenţa de program de mai jos :

```
LOOP:   LD    F0,0(R1)
        ADD   F4,F0,F2
        SD    0(R1),F4
        SUB   R1,R1,#8
        BNEZ  R1, LOOP
```

Se va analiza în continuare cum ar trebui reorganizată şi procesată secvenţa de program anterioară pentru a fi executată pe un procesor VLIW care poate aduce maxim 5 instrucţiuni primitive simultan şi deţine 5 unităţi de execuţie distincte şi anume: 2 unităţi LOAD / STORE (MEM1, MEM2), 2 unităţi de coprocesor flotant (FPP1, FPP2) şi o unitate de procesare a instrucţiunilor întregi şi respectiv a branch-urilor.

	MEM 1	MEM 2	FPP 1	FPP 2	CPU / BRANCH
1	Loop: LD F0, 0(R1)	LD F6, -8(R1)			
2	LD F10, -16(R1)	LD F14, -24(R1)			
3	LD F18, -32(R1)	LD F22, -40(R1)	ADD F4, F0, F2	ADD F8, F6, F2	
4	LD F26, -48(R1)		ADD F12, F10, F2	ADD F16,F14, F2	
5			ADD F20, F18, F2	ADD F24,F22, F2	
6	SD 0(R1), F4	SD -8(R1), F8	ADD F28, F26, F2		
7	SD -16(R1), F12	SD -24(R1), F16			
8	SD -32(R1), F20	SD -40(R1), F24			SUB R1, R1, #48
9	SD 0(R1), F28				BNEZ R1, Loop
10					NOP

Tabelul 4.1.

Execuţia instrucţiunilor pe un procesor MEM cu 5 unităţi de execuţie specializate

De remarcat în acest caz o rată medie de procesare de 2.4 instrucţiuni / ciclu. Altfel spus, bucla de program anterioară se execută în doar 1.42 cicli (10 cicli / 7 bucle). De remarcat printre altele, o redenumire a regiştrilor absolut necesară acestei procesări agresive. Posibilităţile hard / soft aferente unei asemenea procesări vor fi prezentate succint în continuare. Este clar că performanţa procesoarelor MEM este esenţial determinată de programele de compilare şi reorganizare care trebuie să fie deosebit de "inteligente". Cercetări realizate în comun la Universitatea Stanford, USA şi firma DEC (Digital Equipment Corporation) pe procesoare VLIW cu 4 instrucţiuni simultane, au arătat că în aplicaţii reale se ajunge la execuţia a max 2 - 3 instrucţiuni / ciclu, prin compilatoare optimizate. Deşi rare, există realizări comerciale de computere VLIW cu software de optimizare de oarecare succes pe piaţă : IBM RS / 6000 (4 instrucţiuni / ciclu , teoretic), INTEL 860 (maxim 2 instrucţiuni / ciclu), APOLLO DN 10000, etc. Aceste realizări sunt disponibile comercial începând cu anul 1991, deşi cercetările au fost iniţiate începând din 1983. Firma Intel a anunţat că noul său model de procesor

având numele de cod Merced (IA-64), ce va fi lansat în anii 1999 - 2000, va fi realizat pe principii VLIW (EPIC). Având în vedere că în cadrul acestor arhitecturi compilatorul este puternic senzitiv la orice modificare hardware, personal prevăd o legătură hardware - software mai pronunţată decât cea actuală (1998), începând cu lansarea comercială a acestei arhitecturi noi. Necesităţile de "upgrade" hardware - software, cred de asemenea vor fi mai imperioase prin această filosofie EPIC, necesitând deci mai mult decât până acum din partea utilizatorilor, serioase şi continue investiţii financiare corespunzător noilor modele. IA-64 (Intel Architecture) va fi prima arhitectură Intel pe 64 de biţi care va îngloba două caracteristici esenţiale descrise deja în capitolul 2: execuţia condiţionată prin variabile de gardă booleene a instrucţiunilor (“execuţie predicativă”) şi respectiv execuţia speculativă a instrucţiunilor – cu beneficii asupra mascării latenţei unor instrucţiuni mari consumatoare de timp şi deci asupra vitezei de procesare. Arhitectura se bazează pe explicitarea paralelismului instrucţiunilor la nivelul compilatorului într-un mod similar cu cel din arhitecturile VLIW. Intel susţine că programele optimizate pe o anumită maşină IA-64 vor funcţiona fără probleme pe oricare altă viitoare maşină întrucât latenţele unităţilor de execuţie ca şi numărul acestora sunt invizibile pentru optimizatorul de cod. Aceasta se realizează însă prin interconectarea totală a unităţilor de execuţie care se sincronizează prin tehnici de tip “scoreboarding”. Rezultă deci că un program obiect portat de la o versiune mai veche de procesor la alta mai nouă, chiar dacă va funcţiona totuşi corect, se va procesa mai lent decât dacă ar fi optimizat special pentru noua variantă de procesor.

Dificultăţile principale ale modelului VLIW sunt următoarele:

- Paralelismul limitat al aplicaţiei, ceea ce determină ca unităţile de execuţie să nu fie ocupate permanent, fapt valabil de altfel şi la modelul superscalar.

- Incompatibilitate software cu modele succesive şi compatibile de procesoare care nu pot avea în general un model VLIW identic datorită faptului că paralelismul la nivelul instrucţiunilor depinde de latenţele operaţiilor procesorului scalar, de numărul unităţilor funcţionale şi de alte caracteristici hardware ale acestuia.

- Dificultăţi deosebite în reorganizarea aplicaţiei (scheduling) în vederea determinării unor instrucţiuni primitive independente sau cu un grad scăzut de dependenţe.

- Creşterea complexităţii hardware şi a costurilor ca urmare a resurselor multiplicate, căilor de informaţie "lăţite", etc.

- Creşterea necesităţilor de memorare ale programelor datorită reorganizărilor soft şi "împachetării" instrucţiunilor primitive în cadrul unor instrucţiuni multiple care necesită introducerea unor instrucţiuni NOP (atunci când nu există instrucţiuni de un anumit tip disponibile spre a fi asamblate într-o instrucţiune multiplă).

În esenţă, prin aceste modele MEM se încearcă exploatarea paralelismului din programe secvenţiale prin excelenţă, de unde şi limitarea principală a acestui domeniu de "low level parallelism".

Actualmente, datorită faptului că aceste procesoare sunt mult mai ieftine decât procesoarele vectoriale (superprocesoare), şi totodată foarte performante, se pune problema determinării unor clase largi de aplicaţii în care modele superscalar, superpipeline şi VLIW să se comporte mai bine sau comparabil cu modelul vectorial. Se poate arăta relativ simplu, că din punct de vedere teoretic performanţa unui procesor superscalar având N unităţi funcţionale, fiecare cu o structură pipeline pe M nivele, este echivalentă cu cea a unui procesor scalar superpipeline cu o structură pipeline pe M*N nivele. Asocierea unei arhitecturi optimale unei clase de aplicaţii dată, este o problemă dificilă. Performanţa procesoarelor scalare superpipeline, superscalare şi VLIW este în strânsă legătură cu progresele compilatoarelor specifice acestor structuri, compilatoare care trebuie să extragă cât mai mult din paralelismul existent la nivelul instrucţiunilor programului.

De remarcat că modelele superscalar şi VLIW nu sunt exclusive, în implementările reale se întâlnesc adesea procesoare hibride, în încercarea de a se optimiza raportul performanţă preţ. După cum se va vedea, spre exemplu tehnicile soft de optimizare sunt comune ambelor variante de procesoare. Aceste modele arhitecturale de procesoare

paralele sunt considerate a face parte din punct de vedere arhitectural, din generaţia a III-a de microprocesoare, adică cea a anilor 1990 - 2000.

4.2. MODELE DE PROCESARE ÎN ARHITECTURILE SUPERSCALARE

În cazul procesoarelor superscalare sunt citate în literatura de specialitate 3 modalităţi distincte de procesare şi anume: In Order Issue In Order Completion (IN - IN), In Order Issue Out of Order Completion (IN - OUT) şi respectiv Out of Order Issue Out of Order Completion (OUT -OUT). Pentru exemplificarea afirmaţiei de mai sus, să considerăm o secvenţă de instrucţiuni I1 - I6 cu următoarele particularităţi: I1 necesită 2 cicli pentru execuţie, I3 şi I4 sunt în conflict structural, între I4 şi I5 există dependenţă RAW iar I5 şi I6 sunt de asemenea în conflict structural. În aceste condiţii şi considerând un procesor superscalar care poate aduce şi decodifica 2 instrucţiuni simultan şi care deţine 2 unităţi de execuţie, avem situaţiile următoare pe cele trei modele:

a) Modelul IN - IN

Este caracterizat prin faptul că procesorul nu decodifică următoarea pereche de instrucţiuni, decât în momentul în care perechea anterioară se execută. Aşadar atât execuţia cât şi înscrierea rezultatelor se face în ordinea din program ca în figură.

DECODIFICARE	EXECUŢIE		WB	
	UE1	UE2		
I1, I2				1
I3, I4	I1	I2		2
				3
		I3	I1, I2	4
I5, I6		I4		5
	I5		I3, I4	6
	I6			7
			I5, I6	8

Tabelul 4.2.

Exemplu de procesare IN-IN

b) Modelul IN - OUT

Este caracterizat de faptul că execuţia propriu-zisă se face în ordine, în schimb înscrierea rezultatelor se face de îndată ce o instrucţiune s-a terminat de executat. Modelul este mai eficient decât cel precedent însă poate crea probleme de genul întreruperilor imprecise care trebuiesc evitate prin tehnici deja amintite în capitolul 3.

DECODIFICARE	EXECUŢIE		WB	
	UE1	UE2		
I1, I2				1
I3, I4	I1	I2		2
		I3	I2	3
I5, I6		I4	I1, I3	4
	I5		I4	5
	I6		I5	6
			I6	7

Tabelul 4.3.

Exemplu de procesare IN-OUT

c) Modelul OUT - OUT

Este cel mai agresiv şi performant model de procesare a instrucţiunilor într-un procesor superscalar. Instrucţiunile sunt aduse şi decodificate sincron, presupunând deci existenţa unui buffer între nivelul de decodificare şi execuţie (instructions window). Astfel creşte capacitatea de anticipare a instrucţiunilor independente dintr-un program. Modelul permite o exploatare mai bună a paralelismului instrucţiunilor la nivelul unui program dat, prin creşterea probabilităţii de determinare a unor instrucţiuni independente, stocate în buffer.

DECODIFICARE	EXECUŢIE		WB	
	UE1	UE2		
I1, I2				1
I3, I4	I1	I2		2
I5, I6		I3	I2	3
	I6	I4	I1, I3	4
	I5		I4, I6	5
			I5	6

Tabelul 4.4.

Exemplu de procesare OUT-OUT

Desigur că execuţia Out of Order este posibilă numai atunci când dependenţele de date între instrucţiuni o permit. Cade în sarcina hardului eliminarea dependenţelor şi alocarea instrucţiunilor din buffer la diversele unităţi de execuţie (rutarea).

4.3. ARHITECTURA LUI R. TOMASULO

A fost proiectată şi implementată pentru prima dată în cadrul unităţii de calcul în virgulă mobilă din cadrul sistemului IBM - 360 / 91 şi este atribuită lui *Roberto Tomasulo*, considerat a fi fost pionierul procesării superscalare şi pe acest motiv, laureat al prestigiosului premiu *Eckert Mauchly Award* pe anul 1996, acordat celor mai performanţi constructori şi proiectanţi de calculatoare. Arhitectura este una de tip superscalar având deci mai multe unităţi de execuţie, iar algoritmul de control al acestei structuri stabileşte relativ la o instrucţiune adusă, momentul în care aceasta poate fi lansată în execuţie şi respectiv unitatea de execuţie care va procesa instrucţiunea. Arhitectura permite execuţia multiplă şi Out of Order a instrucţiunilor şi constituie modelul de referinţă în reorganizarea dinamică a instrucţiunilor într-un procesor superscalar. De asemenea, algoritmul de gestiune aferent arhitecturii permite anularea hazardurilor WAR şi WAW printr-un ingenios mecanism hardware de redenumire a regiştrilor, fiind deci posibilă execuţia Out of Order a instrucţiunilor şi în aceste cazuri. Aşadar, singurele hazarduri care impun execuţia In Order sunt cele de tip RAW.

În cadrul acestei arhitecturi, detecţia hazardurilor şi controlul execuţiei instrucţiunilor sunt distribuite iar rezultatele instrucţiunilor sunt "pasate anticipat" direct unităţilor de execuţie prin intermediul unei magistrale comune numită CDB (Common Data Bus). Arhitectura de principiu este prezentată în figura 4.5. Ea a fost implementată prima dată în unitatea de virgulă mobila FPP a calculatorului IBM 360/91, pe baza

căreia se va prezenta în continuare. Staţiile de rezervare (SR) memorează din SIF (Stiva Instrucţiuni Flotante - pe post de buffer de prefetch aici) instrucţiunea ce urmează a fi lansată spre execuţie. Execuţia unei instrucţiuni începe dacă există o unitate de execuţie neocupată momentan şi dacă operanzii aferenţi sunt disponibili în SR aferentă. Fiecare unitate de execuţie (ADD, MUL) are asociată o SR proprie. Precizăm că unităţile ADD execută operaţii de adunare / scădere iar unitaţile MUL operaţii de înmulţire / împărţire. Modulele LB şi SB memorează datele încărcate din memoria de date respectiv datele care urmează a fi memorate. Toate rezultatele provenite de la unităţile de execuţie şi de la bufferul LB sunt trimise pe magistrala CDB. Bufferele LB, SB precum şi SR deţin câmpuri de TAG necesare în controlul hazardurilor de date între instrucţiuni.

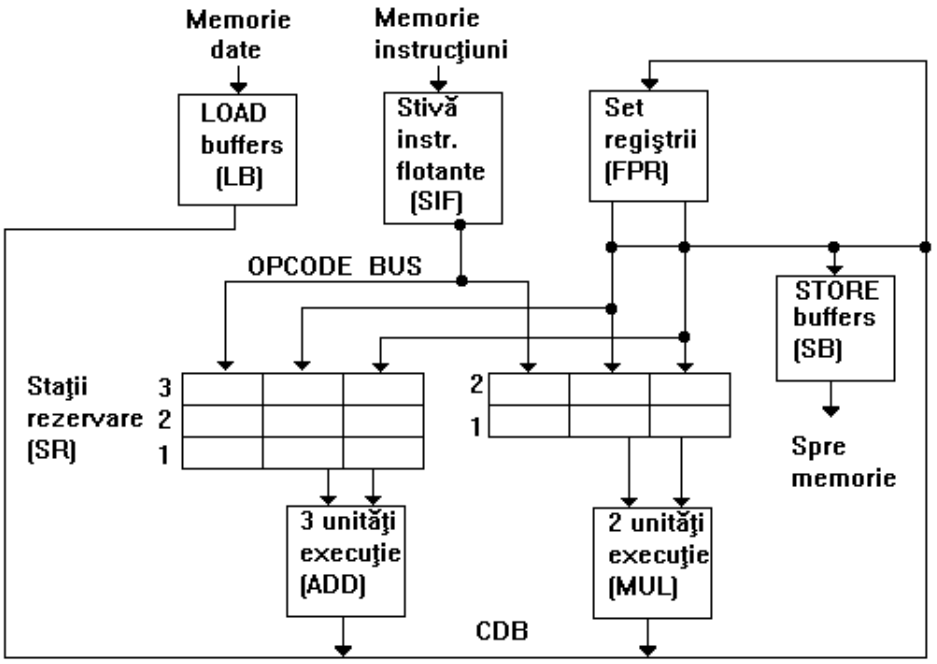

Figura 4.5. Arhitectura lui Tomasulo

Există în cadrul acestei unităţi de calcul în virgulă mobilă şi deci în cadrul mai general al procesării superscalare, 3 stagii de procesare a instrucţiunilor şi anume:

1) *Startare* - aducerea unei instrucţiuni din SIF (bufferul de prefetch) într-o staţie de rezervare. Aducerea se va face numai dacă există o SR disponibilă. Dacă operanzii aferenţi se află în FPR (setul de regiştri generali), vor fi aduşi în SR aferentă. Dacă

instrucţiunea este de tip LOAD / STORE, va fi încărcată într-o SR numai dacă există un buffer (LB sau SB) disponibil. Dacă nu există disponibilă o SR sau un buffer, rezultă că avem un hazard structural şi instrucţiunea va aştepta până când aceste resurse se eliberează.

2) Execuţie - dacă un operand nu este disponibil, prin monitorizarea magistralei CDB de către SR ("snooping" - spionaj), se aşteaptă respectivul operand. În această fază se testează existenţa hazardurilor de tip RAW între instrucţiuni. Când ambii operanzi devin disponibili, se execută instrucţiunea în unitatea de execuţie corespunzătoare.

3) Scriere rezultat (WB) - când rezultatul este disponibil se înscrie pe CDB şi de aici în FPR sau într-o SR care aşteaptă acest rezultat ("forwarding").

De observat că nu există pe parcursul acestor faze testări pentru hazarduri de tip WAR sau WAW, acestea fiind eliminate prin însăşi natura algoritmului de comandă după cum se va vedea imediat. De asemenea, operanzii sursă vor fi preluaţi de către SR direct de pe CDB prin "forwarding" când acest lucru este posibil. Evident că ei pot fi preluaţi şi din FPR în cazurile în care nu vor fi produşi de instrucţiunile din staţiile de rezervare sau din unităţile de execuţie.

O SR deţine 6 câmpuri cu următoarea semnificaţie:

OP - codul operaţiei (opcode) instrucţiunii din SR.

Qj, Qk - codifică pe un număr de biţi unitatea de execuţie (ADD, MUL, etc.) sau numărul bufferului LB, care urmează să genereze operandul sursă aferent instrucţiunii din SR. Dacă acest câmp este zero, rezultă că operandul sursă este deja disponibil într-un câmp Vi sau Vj al SR sau pur şi simplu nu este necesar. Câmpurile Qj, Qk sunt pe post de TAG, adică atunci când o unitate de execuţie sau un buffer LB "pasează" rezultatul pe CDB, acest rezultat se înscrie în câmpul Vi sau Vj al acelei SR al cărei TAG coincide cu numărul sau numele unităţii de execuţie sau bufferului LB care a generat rezultatul.

Vj, Vk - conţin valorile operanzilor sursă aferenţi instrucţiunii din SR. Remarcăm că doar unul dintre câmpurile Q respectiv V sunt valide pentru un anumit operand.

BUSY - indică atunci când este setat că SR şi unitatea de execuţie aferentă sunt ocupate momentan.

Regiştrii generali FPR şi bufferele SB deţin fiecare de asemenea câte un câmp Qi, care codifică numărul unităţii de execuţie care va genera data ce va fi încărcată în respectivul registru general respectiv care va fi stocată în memoria de date. De asemenea, deţin câte un bit de BUSY. Bufferele SB deţin în plus un câmp care conţine adresa de acces precum şi un câmp care conţine data de înscris. Bufferele LB conţin doar un bit BUSY şi un câmp de adresă.

Spre a exemplifica funcţionarea algoritmului să considerăm în continuare o secvenţă simplă de program maşină:

	Start	Execuţie	WB
1. LF F6, 27(R1)	x	x	x
2. LF F2, 45(R2)	x	x	
3. MULTF F0, F2, F4	x		
4. SUBF F8, F6, F2	x		
5. DIVF F10, F0, F6	x		
6. ADDF F6, F8, F2	x		

În continuare prezentăm starea SR şi a FPR în momentul definit mai sus, adică prima instrucţiune încheiată, a 2-a în faza de execuţie iar celelalte aflate în faza de startare.

Staţiile de rezervare (SR)

Nume SR	BUSY	OP	Vj	Vk	Qj	Qk
ADD1	DA	SUB	(LOAD1)			LOAD2
ADD2	DA	ADD			ADD1	LOAD2
ADD3	NU					
MUL1	DA	MUL		(F4)	LOAD2	
MUL2	DA	DIV		(LOAD1)	MUL1	

Tabelul 4.5.

Situaţia staţiilor de rezervare în prima instanţă

Regiştrii generali FPR

CÂMP	F0	F2	F4	F6	F8	F10
Qi	MUL1	LOAD2		ADD2	ADD1	MUL1
BUSY	DA	DA	NU	DA	DA	DA

Tabelul 4.6.

Starea regiştrilor generali în prima instanţă

Din aceste structuri de date implementate în hardware, rezultă de exemplu că SR ADD1 urmează să lanseze în execuţie instrucţiunea SUBF F8, F6, F2. Valoarea primului operand (F6) se află deja în câmpul Vj unde a fost memorată de pe magistrala CDB ca urmare a terminării execuţiei primei instrucţiuni. Evident că rezultatul acestei instrucţiuni a fost preluat de pe CDB în registrul F6 dar şi în bufferul LB1. Al 2-lea operand al instrucţiunii SUBF nu este încă disponibil. Câmpul de TAG Qk arată că acest operand va fi generat pe CDB cu "adresa" LOAD2 (LB2) şi deci această SR va prelua operandul în câmpul Vk de îndată ce acest lucru devine posibil. Preluarea acestui operand se va face de către toate SR care au un câmp de TAG identic cu LOAD2 (LB2).

Să considerăm de exemplu că latenţa unităţilor ADD este de 2 impulsuri de tact, latenţa unităţilor MUL este de 10 impulsuri de tact pentru o înmulţire şi respectiv 40 impulsuri de tact pentru o operaţie de împărţire. "Starea" secvenţei anterioare în tactul premergător celui în care instrucţiunea MULTF va intra în faza WB va fi următoarea:

	Start	Execuţie	WB
1. LF F6, 27(R1)	x	x	x
2. LF F2, 45(R2)	x	x	x
3. MULTF F0, F2, F4	x	x	

4. SUBF F8, F6, F2 x x x

5. DIVF F10, F0, F6 x

6. ADDF F6, F8, F2 x x x

În acest moment, starea staţiilor de rezervare şi a setului de regiştri generali va fi cea prezentată în tabelele 4.7 respectiv 4.8:

Staţiile de rezervare (SR)

Nume SR	BUSY	OP	Vj	Vk	Qj	Qk
ADD1	NU					
ADD2	NU					
ADD3	NU					
MUL1	DA	MUL	(LOAD2)	(F4)		
MUL2	DA	DIV		(LOAD1)	MUL1	

Tabelul 4.7.

Situaţia staţiilor de rezervare în a doua instanţă

Regiştrii generali FPR

CÂMP	F0	F2	F4	F6	F8	F10
Qi	MUL1					MUL2
BUSY	DA	NU	NU	NU	NU	DA

Tabelul 4.8.

Starea regiştrilor generali în a doua instanţă

De remarcat că algoritmul a eliminat hazardul WAR prin registrul F6 între instrucţiunile DIVF şi ADDF şi a permis execuţia Out of Order a acestor instrucţiuni, în vederea creşterii ratei de procesare. Cum prima instrucţiune s-a încheiat, câmpul Vk aferent SR MUL2 va conţine valoarea operandului instrucţiunii DIVF, permiţând deci

ca instrucţiunea ADDF să se încheie înaintea instrucţiunii DIVF. Chiar dacă prima instrucţiune nu s-ar fi încheiat, cîmpul Qk aferent SR MUL2 ar fi pointat la LOAD1 şi deci instrucţiunea DIVF ar fi fost independentă de ADDF. Aşadar, algoritmul prin "pasarea" rezultatelor în SR de îndată ce acestea sunt disponibile, evită hazardurile WAR. Pentru a pune în evidenţă întreaga "forţă" a algoritmului în eliminarea hazardurilor WAR şi WAW prin redenumire dinamică a resurselor, să considerăm bucla următoare:

```
LOOP:   LF F0, 0 (R1)
        MULTF F4, F0, F4
        SD 0 (R1), F4
        SUB R1, R1, #4
        BNEZ R1, LOOP
```

Considerând o unitate de predicţie a branchurilor de tip "branch-taken", 2 iteraţii succesive ale buclei se vor procesa ca mai jos (tabelele 4.9):

	Start	Execuţie	WB
LF F0, 0 (R1)		x	x
MULTF F4, F0, F2	x		
SD 0 (R1), F4		x	
LF F0, 0 (R1)		x	x
MULTF F4, F0, F2	x		
SD 0 (R1), F4		x	

Se observă o procesare de tip "loop unrolling" ("netezirea buclei") prin hardware (tabelele 4.9). Instrucţiunea LOAD din a 2-a iteraţie se poate executa înaintea instrucţiunii STORE din prima iteraţie întrucât adresele de acces sunt diferite în câmpurile din buffere. Ulterior şi instrucţiunile MULTF se vor putea suprapune în execuţie. De remarcat deci hazardul de tip WAW prin F0 între instrucţiunile de LOAD s-a eliminat cu ajutorul SR şi a bufferelor SB şi LB.

Staţiile de rezervare (SR)

Nume SR	BUSY	OP	Vj	Vk	Qj	Qk
ADD1	NU					
ADD2	NU					
ADD3	NU					
MUL1	DA	MUL		(F2)	LOAD1	
MUL2	DA	MUL		(F2)	LOAD2	

Buffere SB

CÂMP	SB 1	SB 2	SB 3
Qi	MUL1	MUL2	
BUSY	DA	DA	NU
ADR	(R1)	(R1)-8	
V			

Buffere LB

CÂMP	LB 1	LB 2	LB 3
ADR	(R1)	(R1)-8	
BUSY	DA	DA	NU
V			

Tabelul 4.9.

Contextul procesorului aferent buclei de program

Arhitectura Tomasulo are deci avantajele de a avea logica de detecţie a hazardurilor distribuită şi prin redenumire dinamică a resurselor, elimină hazardurile WAW şi WAR. Acest lucru este posibil pentru că resursele tip sursă folosite şi aflate în starea "BUSY", nu se adresează ca nume de regiştri ci ca nume de unităţi de execuţie ce vor produce aceste surse. În schimb, arhitectura este complexă, necesitând deci costuri ridicate. Este necesară o logică de control complexă, capabilă să execute căutări / memorări asociative cu viteză ridicată. Având în vedere progresele mari ale tehnologiilor VLSI, variante uşor îmbunătăţite ale acestei arhitecturi se aplică practic în toate procesoarele superscalare actuale (pentru reducerea conflictelor, se folosesc mai multe busuri de tip CDB).

Acest mecanism de forwarding din arhitectura lui Tomasulo, are meritul de a reduce semnificativ din presiunea la "citire" asupra setului general de regiştri logici, speculând dependenţele RAW între instrucţiuni.

4.4. O ARHITECTURĂ REPREZENTATIVĂ DE PROCESOR SUPERSCALAR

Având în vedere ideile de implementare a execuţiilor multiple din arhitectura lui Tomasulo, o arhitectură superscalară reprezentativă este prezentată în figura 4.6. Prin SR am notat staţiile de rezervare aferente unităţilor de execuţie ale procesorului. Acestea implementează printre altele bufferul "instruction window" necesar procesoarelor superscalare cu execuţie Out of Order. Numărul optim de locaţii al fiecărei SR se determină pe bază de simulare.

Deşi performanţa maximă a unei asemenea arhitecturi ar fi de 6 instrucţiuni/ciclu, în realitate, bazat pe simulări ample, s-a stabilit că rata medie de execuţie este situată între 1-2 instrucţiuni / ciclu. În sub 1% din cazuri, măsurat pe benchmark-uri nenumerice, există un potenţial de paralelism mai mare de 6 instrucţiuni / ciclu în cazul unei arhitecturi superscalare "pure". Aceasta se datorează în primul rând capacităţii limitate a bufferului de prefetch care constituie o limitare principială a oricărui procesor, exploatarea paralelismului între instrucţiuni fiind limitată de capacitatea acestui buffer. În tehnologia actuală acesta poate memora între 8 - 64 instrucţiuni, capacităţi mai mari ale acestuia complicând mult logica de detecţie a hazardurilor RAW după cum am arătat (vezi paragraful 3.1). Prezentăm pe scurt rolul modulelor componente din această schemă tipică.

Decodificatorul plasează instrucţiunile multiple în SR- urile corespunzătoare. O unitate funcţională poate starta execuţia unei instrucţiuni din SR imediat după decodificare dacă instrucţiunea nu implică dependenţe, operanzii îi sunt diponibili şi dacă unitatea de execuţie este liberă. În caz contrar, instrucţiunea aşteaptă în SR până când aceste condiţii vor fi îndeplinite. Dacă mai multe instrucţiuni dintr-o SR sunt simultan disponibile spre a fi executate, procesorul o va selecta pe prima din secvenţa de instrucţiuni.

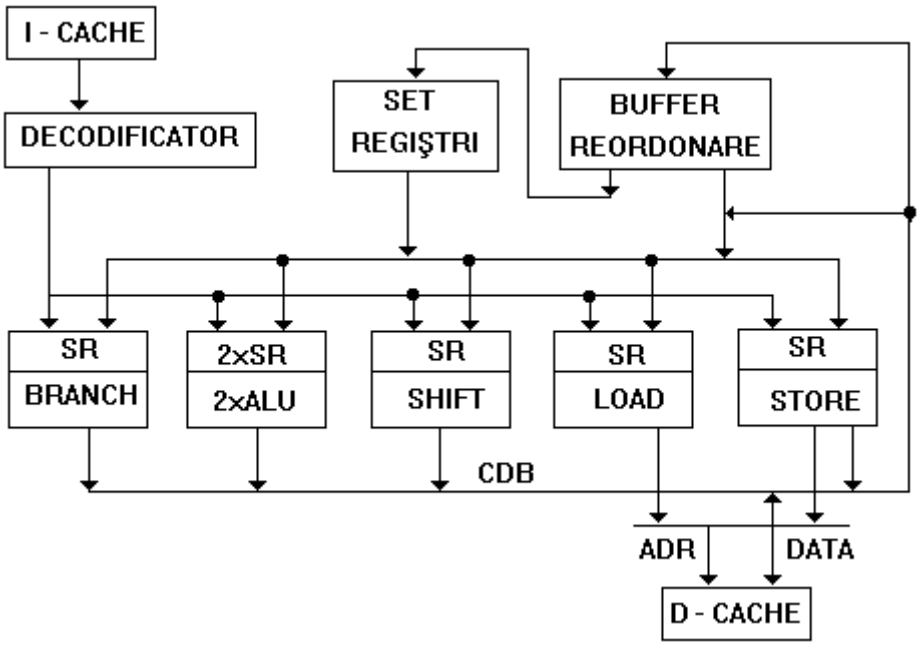

Figura 4.6. Arhitectura tipică a unui procesor superscalar

Desigur că este necesar un mecanism de arbitrare în vederea accesării CDB de către diversele unităţi de execuţie (UE). În vederea creşterii eficienţei, deseori magistralele interne sunt multiplicate. Prezentăm în figura 4.7 circulaţia informaţiei într-o structură superscalară puternică, similară cu cea implementată la microprocesorul Motorola MC 88110. Setul de regiştri generali (FILE) este multiplicat fizic, conţinutul acestor seturi fizice este identic însă în orice moment. Am considerat că UE- urile conţin şi staţiile de rezervare aferente. Din acest motiv, având în vedere mecanismul de "forwarding" implementat, comunicaţia între UE şi CDB s-a considerat bidirecţională.

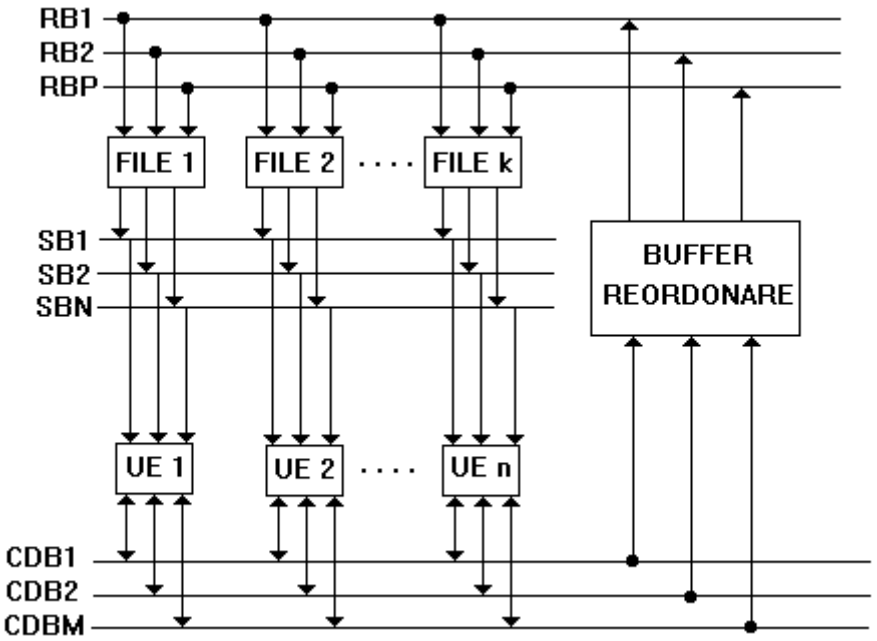

Figura 4.7. Multiplicarea magistralelor şi a seturilor de regiştri

Există 3 categorii de busuri comune şi anume: busuri rezultat (RB), busuri sursă (SB) si busuri destinaţie (CDB). N corespunde numărului maxim de instrucţiuni care pot fi lansate simultan în execuţie. Min (M, P) reprezintă numărul maxim de instrucţiuni care pot fi terminate simultan. Uzual se alege M = P. Există implementate mecanisme de arbitrare distribuite în vederea rezolvării tuturor hazardurilor structurale posibile pe parcursul procesărilor.

Pe bază de simulare se încearcă stabilirea unei arhitecturi optimale. Astfel se arată că pentru o rată de fetch şi de execuţie de 4 instrucţiuni, procesarea optimă din punct de vedere performanţă/cost impune 7 busuri destinaţie, 4 unităţi de execuţie întregi şi 8 staţii de rezervare pentru unităţile LOAD / STORE. Pentru o asemenea arhitectură s-ar obţine o rată de procesare de 2.88 instrucţiuni / tact, măsurat însă pe benchmark-uri cu un puternic caracter numeric, favorizante deci (Livermore Loops). Ideea de bază este însă că hazardurile structurale se elimină şi aici prin multiplicarea resurselor hardware, deci fără pierderi de performanţă. Gradul de multiplicare trebuie însă stabilit prin simulări ample ori prin metode teoretice.

Bufferul de reordonare

Bufferul de reordonare (RB - *Reorder Buffer*) este în legătură cu mecanismul de redenumire dinamică a regiştrilor în vederea execuţiei Out of Order precum şi cu necesitatea implementării unui mecanism precis de tratare a evenimentelor de excepţie(derute, devieri, întreruperi hard-soft, etc.). Acesta conţine un număr de locaţii care sunt alocate în mod dinamic rezultatelor instrucţiunilor.

În urma decodificării unei instrucţiuni, rezultatul acesteia este asignat unei locaţii din RB, iar numărul registrului destinaţie este asociat acestei locaţii. În acest mod, registrul destinaţie este practic redenumit printr-o locaţie din RB. În urma decodificării se crează 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 RB. 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 RB sau, dacă valoarea nu a fost încă procesată, tag-ul aferent locaţiei. Dacă mai multe locaţii din RB conţin acelaşi număr de registru (mai multe instrucţiuni în curs au avut acelaşi registru destinaţie), se va genera locaţia cea mai recent înscrisă (tag sau valoare).

Este evident deja că RB se implementează sub forma unei memorii asociative, căutarea făcându-se după numărul registrului destinaţie la scriere, respectiv sursă la citire. Dacă accesarea RB se soldează cu miss, atunci operandul sursă va fi citit din setul de regiştri. În caz de hit, valoarea sau tag-ul citite din RB sunt memorate în SR corespunzătoare. Când o unitate de execuţie generează un rezultat, acesta se va înscrie în SR şi în locaţia din RB 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 RB, instrucţiunile următoare vor continua să-l citească din RB ca operand sursă până când va fi evacuat şi scris în setul de regiştri. Evacuarea (faza WB) se va face în ordinea secvenţei originale de instrucţiuni pentru a se putea evita

excepţiile imprecise. Aşadar, redenumirea unui registru cu o locaţie din RB se termină în momentul evacuării acestei locaţii.

Bufferul RB poate fi gestionat ca o memorie FIFO (First In First Out). În momentul decodificării unei instrucţiuni, rezultatul acesteia este alocat în coada RB. Rezultatul instrucţiunii este înscris în momentul în care unitatea de execuţie corespunzătoare îl generează. Când acest rezultat ajunge în prima poziţie a RB, dacă între timp nu au apărut excepţii, este înscris în setul de regiştri. Dacă instrucţiunea nu s-a încheiat atunci când locaţia alocată în RB a ajuns prima, bufferul RB nu va mai avansa până când această instrucţiune nu se va încheia. Decodificarea instrucţiunilor poate însă continua atât timp cât mai există locaţii disponibile în RB.

Dacă apare o excepţie, bufferul RB este golit, procesorul bazându-se pe contextul memorat In Order în setul general de regiştri. Astfel, deşi procesează Out of Order, procesorul superscalar implementează un mecanism de excepţii precise. Mecanismul este similar cu cel numit "history buffer" şi prezentat în cadrul procesoarelor pipeline scalare. În general, capacitatea RB se stabileşte pe baza simulării unei arhitecturi superscalare, pe diverse programe de test reprezentative (benchmark- uri). Se apreciază bazat pe măsurări şi simulări laborioase pe o multitudine de benchmark-uri că performanţa unui procesor superscalar "pur" nu poate depăşi în medie 2-3 instr. / tact.

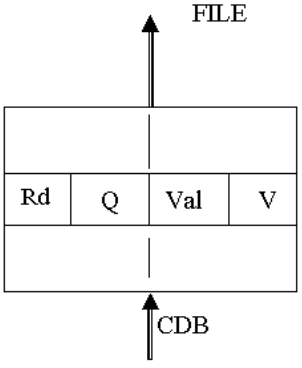

Figura 4.7.bis. Formatul intrării RB

O linie din RB conţine următoarele câmpuri:

Rd = registrul destinaţie care este referit de instrucţiune

Q = numărul unităţii de execuţie care generează rezultatul (asignat lui Rd)

Val = valoarea (rezultatul) asigurată registrului Rd

V = intrare validă / invalidă

În RB se caută asociativ astfel:

după Rd, în momentul scrierii rezultatului instrucţiunii în registrul destinaţie (faza *"Write Back"*)

după găsirea unei identităţi de tip $R_{sursă1}=R_d$ sau $R_{sursă2}=R_d$, în faza de "*issue*", când este necesară asignarea valorii sau a tag-ului regiştrilor sursă, în staţia de rezervare aferentă.

O alternativă la utilizarea unui RB constă în utilizarea **tehnicii de redenumire dinamică a regiştrilor.** Aceasta se bazează pe exploatarea unui set extins de regiştri fizici care înlocuieşte din punct de vedere funcţional atât RB-ul cât şi staţiile de rezervare. Pe timpul fazei de lansare în execuţie a instrucţiunilor, registrul destinaţie este redenumit cu un registru liber (nealocat) din acest set extins. Astfel, dependenţele de tip WAR şi WAW sunt eliminate. Acest registru fizic alocat destinaţiei instrucţiunii, devine "*registru logic*" abia în momentul în care instrucţiunea se încheie din punct de vedere al procesării *in-order* (aşadar faza "*commit*", ulterioară fazei "*write-back*"). Conversia logic-fizic se face printr-o simplă tabelă de mapare.

Un avantaj al redenumirii dinamice faţă de tehnica RB constă în faptul că procesele aferente terminării *in-order* a instrucţiunilor sunt simplificate. Această terminare implică faptul că maparea respectivului registru logic în registrul fizic corespondent nu mai este activă, şi, desigur, registrul fizic redevine disponibil. În cazul RB, staţia de rezervare se elibera în momentul încheierii execuţiei instrucţiunii în unitatea de execuţie iar locaţia din RB se elibera la finele execuţiei *in-order* a instrucţiunii (faza "*commit*"). Totuşi, în cazul dealocării unui registru fizic trebuie verificat să nu mai existe instrucţiuni în curs care să-l utilizeze pe post de registru-sursă. Ca alternativă pentru dealocarea unui registru fizic, procesorul poate aştepta ca o altă instrucţiune să modifice respectivul registru. În acest caz, un registru poate fi ţinut alocat mai mult decât ar fi stricat necesar

(cazul MIPS R10000). O altă simplificare a tehnicii de redenumire faţă de cea a utilizării unui RB constă în faptul că operanzii sursă ai instrucţiunii se află în setul extins de regiştrii (în cazul filozofiei RB, sursele puteau fi găsite în RB sau în regiştrii logici). Redenumirea dinamică a regiştrilor este implementată la microprocesoarele MIPS R10000/120000, Alpha 21264, Pentium III, IV).

În esenţă, un procesor cu execuţii multiple ale instrucţiunilor este compus din 2 mecanisme decuplate: mecanismul de aducere (fetch) a instrucţiunilor pe post de producător şi respectiv mecanismul de execuţie a instrucţiunilor pe post de consumator. Separarea între cele 2 mecanisme (arhitectură decuplată) se face prin bufferele de instrucţiuni şi staţiile de rezervare, ca în figura 4.18. Instrucţiunile de ramificaţie şi predictoarele hardware aferente acţionează printr-un mecanism de reacţie între consumator şi producător. Astfel, în cazul unei predicţii eronate, bufferul de prefetch trebuie să fie golit măcar parţial iar adresa de acces la cache-ul de instrucţiuni trebuie şi ea modificată în concordanţă cu adresa la care se face saltul.

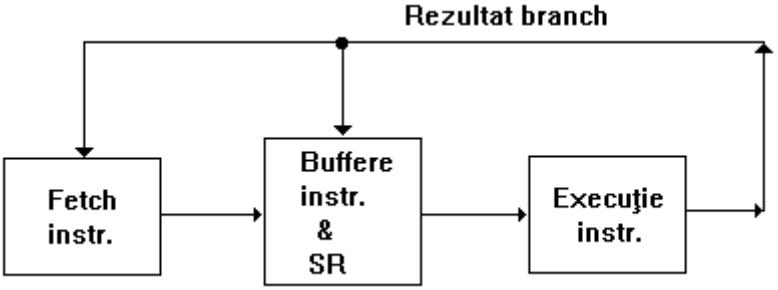

Figura 4.18. Arhitectură superscalară decuplată