Curs 9 Arhitecturi RISC de uniprocesare paralelă a datelor utilizând cuvinte de instrucțiuni foarte lungi tip VLIW

1. Conceptul de paralelism

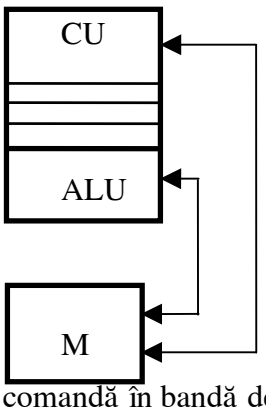
Paralelismul se referă la capacitatea de a suprapune sau executa simultan mai multe operaţii (citiri de la memorie (“memory fetch”, “load”), scrieri la memorie (“store”), operaţii aritmetice şi logice, operaţii de intrare/ieşire etc).

Paralelismul poate fi de tip:

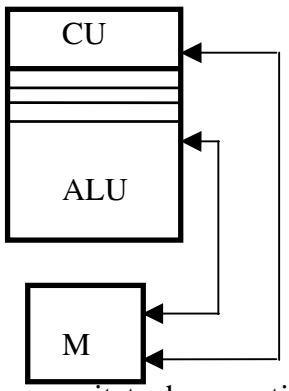
a) ***Bandă de asamblare*** (“Pipelining”): este o implementare similară liniei de asamblare, prin utilizarea unor tehnici de suprapunere a fazelor de execuţie a mai multor instrucţiuni, ceea ce conduce la creşterea performanţelor unităţilor de execuţie EU (“Execution Unit”) şi unităţii de comandă CU (“Command Unit”).

Lucrul în bandă de asamblare presupune divizarea unui “task” T în “subtask”-uri T1, ..., Tk şi asignarea “subtask”-urilor la un lanţ de staţii de procesare numite segmente de bandă de asamblare. Paralelismul se obţine prin operarea segmentelor simultan. Se pot folosi benzi de asamblare atât la nivelul fluxului de instrucţiuni, cât şi la nivelul fluxului de date, după cum se prezintă în figură:

Banda de asamblare poate fi implementată şi combinat, atât la nivelul comenzii, cât şi la nivelul execuţiei.



unitate de comandă în bandă de asamblare
- CU (“Command Unit”) -



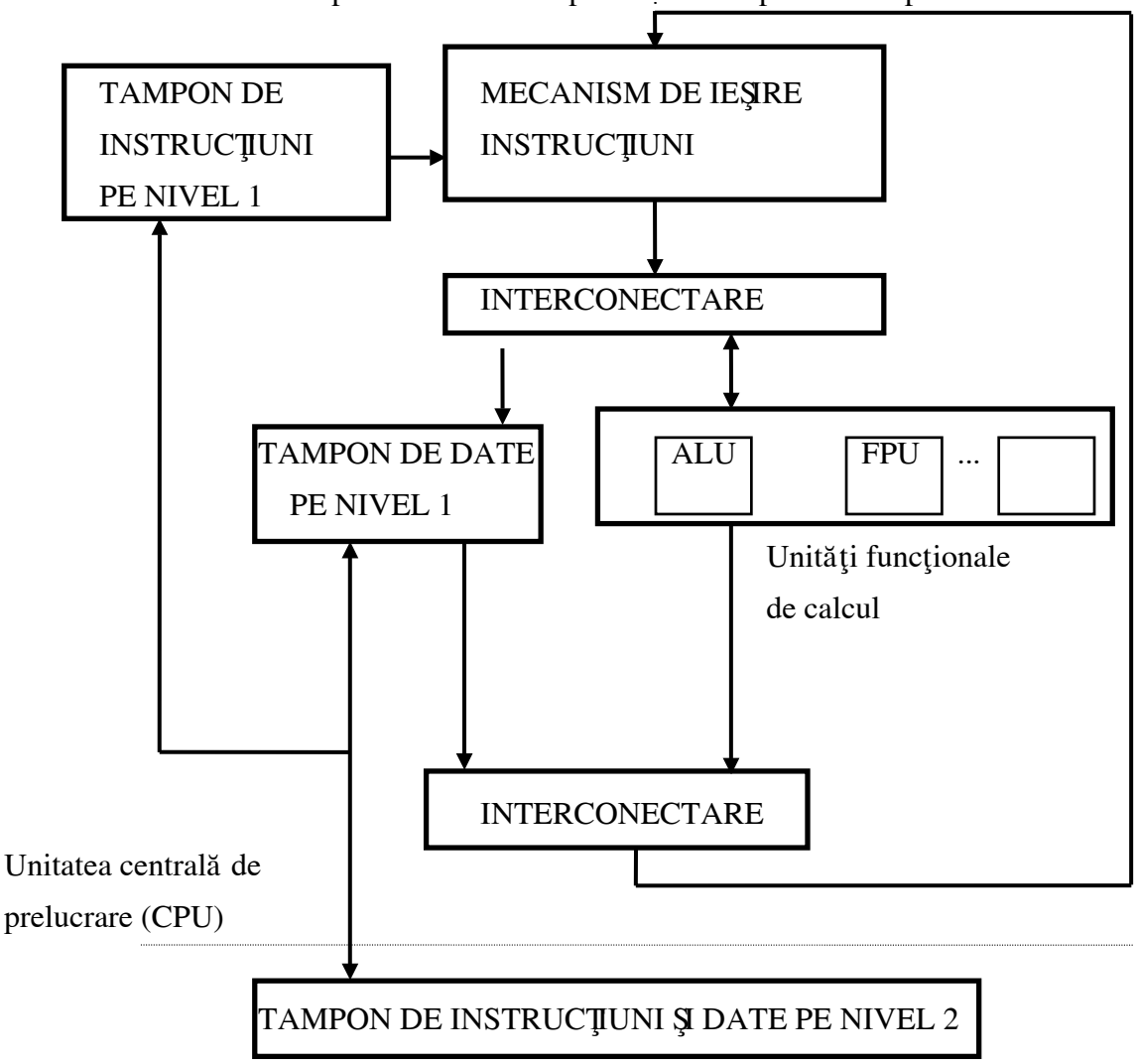
unitate de execuţie în bandă de asamblare
- ALU (“Arithmetic Logic Unit”) -

b) ***Funcţional***: prin utilizarea mai multor unităţi independente, ce realizează funcţii (logice/aritmetice) diferite, simultan cu date diferite.

Utilizarea unităţilor funcţionale în bandă de asamblare conferă procesoarelor proprietatea de scalare. Î n prezent, unităţile funcţionale în bandă de asamblare sunt utilizate, atât de RISC, cât şi de CISC, tendinţa actuală fiind de superscalare. Superscalarea [81;83;126;127] este o cale de creştere a performanţei prin exploatarea paralelismului de granularitate scăzută (“fine-grain”) şi execuţia mai multor instrucţiuni pe ciclul de ceas. Acest lucru se realizează prin implementarea

în unitatea centrală de prelucrare, a unor unităţi funcţionale multiple, fiecare dintre ele fiind organizată în bandă de asamblare.

În continuare este prezentat modelul posibil al unui procesor superscalar:



Un astfel de model permite execuţia a patru-cinci instrucţiuni într-un ciclu de ceas. Unităţile funcţionale multiple de calcul execută instrucţiunile în paralel. Mecanismul de ieşire a instrucţiunilor încarcă, într-o coadă, mai multe instrucţiuni din memoria intermediară de instrucţiuni de pe nivelul 1 şi le trimite apoi în execuţie la unităţile funcţionale disponibile. Memoria intermediară de instrucţiuni de pe nivelul 1 furnizează instrucţiuni mecanismului de ieşire a instrucţiunilor iar memoria intermediară de date de pe nivelul 1 furnizează operanzi pentru unităţile funcţionale multiple. Mecanismul de interconectare asigură transferul între componentele unităţii CPU.

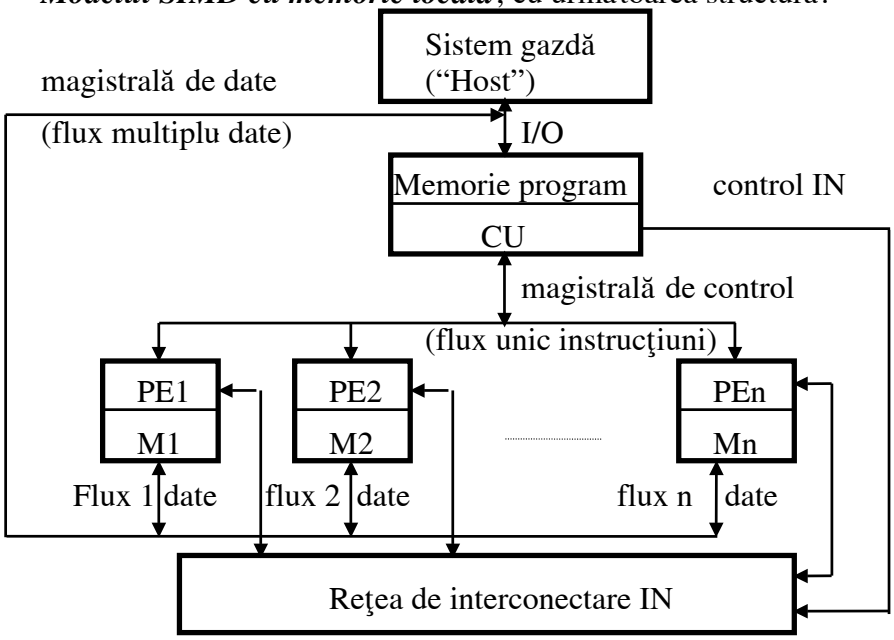
Utilizarea memoriilor intermediare pe două niveluri – nivelul 1 în acelaşi circuit cu procesorul (“on-chip”), nivelul 2 în exteriorul circuitului procesorului (“off-chip”), precum şi împărţirea memoriei intermediare incorporate în tampon de instrucţiuni şi tampon de date, conduc, de asemenea, la creşterea frecvenţei de operare a procesorului. Un calcul al influenţei memoriilor intermediare asupra creşterii performanţei procesoarelor este realizat la sfârşit.

c) ***Masiv de procesare SIMD*** ("Single Instrution flow Multiple Data flow" – flux singular de instrucţiuni şi flux multiplu de date): prin utilizarea unui masiv de elemente procesoare PE ("Processor Element") identice, care realizează simultan aceeaşi operaţie (controlate de aceeaşi comandă), asupra unor date diferite stocate în memoriile locale.

Sistemele SIMD sunt constituite într-un masiv de elemente de procesoare (PE), elemente de memorie (M), o unitate de control (CU) şi o reţea de interconectare (IN – "Interconnecting Network"). Acestea sunt ataşate unui sistem gazdă ("Host") care, din punct de vedere al utilizatorului, este un sistem de tip "front-end". Rolul lui este de a executa compilări, încărcarea programelor, operaţii de intrare/ieşire şi alte funcţii ale sistemului de operare.

În funcţie de modul de operare şi de control există:

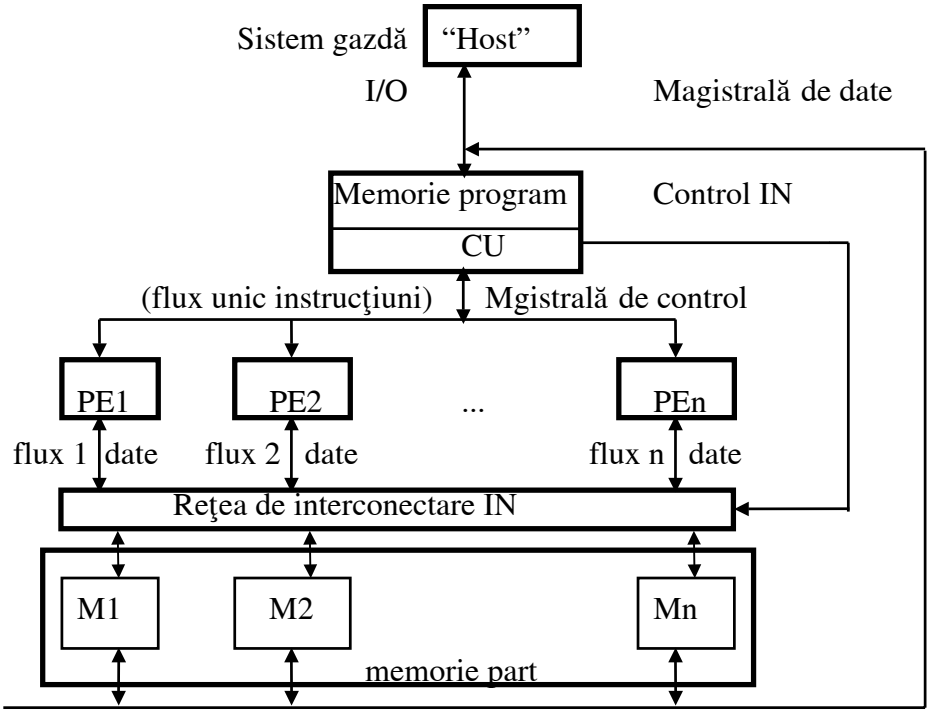
Modelul SIMD cu memorie locală, cu următoarea structură:



Conform acestui model, fiecare procesor are memoria sa locală şi execută aceeaşi instrucţiune furnizată de unitatea de control CU. Unitatea CU citeşte instrucţiuni din memoria program. Instrucţiunile de control sau cele cu operanzi scalari sunt executate direct de către unitatea CU, iar instrucţiunile cu operanzi vectoriali sunt trimise spre execuţie la procesoarele PE. De asemenea, se pot transmite cuvinte de date necesare pentru operaţiile scalare cu vectori. Când procesoarele PE îşi încarcă datele de la memoria lor locală, unitatea CU transmite adresa corespunzătoare procesoarelor PE. Instrucţiunea de control de la CU furnizează aceeaşi adresă pentru toate PE. Comunicaţia de date se realizează transferând datele între procesoare prin intermediul unei reţele de interconectare. Fiecare procesor sau grup de procesoare are porturi de comunicaţie şi registre tampon ("buffer") de date. Reţeaua IN execută câteva funcţii de mapare, depinzând de topologia reţelei şi controlul cerut de unitatea CU. Procesoarele sunt echipate cu registre de stare cu indicatori pentru testări aritmetice şi logice. Granularitatea, numărul de procesoare şi tipul reţelei variază de la o implementare la alta.

Model SIMD cu memorie partajată ("shared")

Conform acestui model, fiecare procesor comunică cu memoria comună partajată, prin intermediul unei reţele de interconectare şi execută aceeaşi instrucţiune furnizată de unitatea de control CU, la fel ca în cazul modelului anterior. Modelul are următoarea structură:



Memoriile sunt separate de procesoarele PE printr-o reţea de interconectare. Un procesor poate accesa oricare memorie iar reţeaua IN permite accesul în paralel la memoria partajată.

Un avantaj este că sunt realizate mai multe funcţii de mapare procesor-memorie, ceea ce înseamnă că pot fi mapaţi uşor în "hardware" mai mulţi algoritmi. Acest lucru implică bineînţeles creşterea volumului de comutări de date, ceea ce conduce la creşterea timpului de acces la memorie.

Unitatea CU este similară cu cea din modelul SIMD cu memorie locală, însă reţeaua IN este mai complexă.

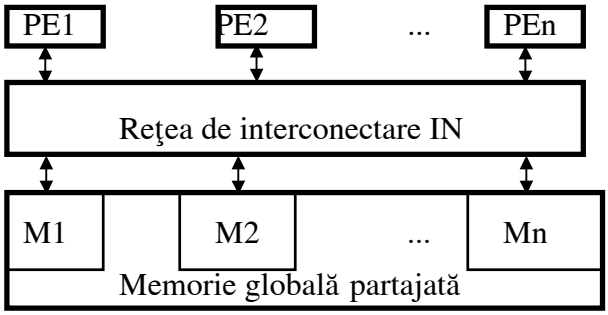
d) ***Multiprocesare MIMD*** ("Multiple Instruction flow Multiple Data flow" – fluxuri multiple de instrucţiuni şi date): prin utilizarea mai multor procesoare, fiecare executând instrucţiuni proprii şi comunicând, în general, printr-o memorie comună, sau conectate într-o reţea.

Multiprocesoarele se referă la execuţia simultană de "task"-uri pe un sistem de calcul paralel asincron format din mai multe procesoare, fiecare cu CU şi ALU, şi module de memorie şi I/O. Procesoarele cooperează strâns, dar independent.

Există două modele de bază:

Modelul MIMD cu memorie partajată ("shared-memory")

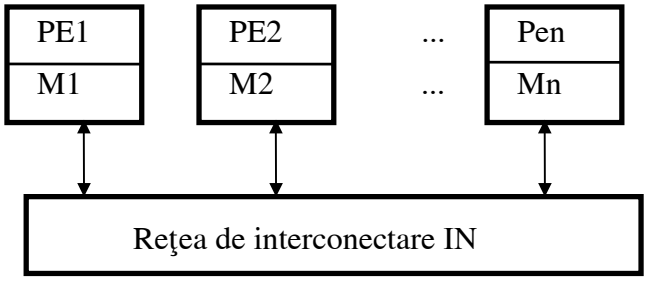
Sunt sisteme strâns cuplate, în care există o conectivitate completă între procesoare şi modulele de memorie; au următoarea structură:



PE sunt procesoare de execuţie nu neapărat identice, cu sau fără memorie locală. Memoria globală poate fi accesată de către toate procesoarele; creşterea lăţimii de bandă la memoria globală se realizează prin implementarea local la procesoare a unor memorii intermediare ("cache").

Modelul MIMD cu trecere de mesaje ("message-parsing")

Structura de sistem este următoarea:

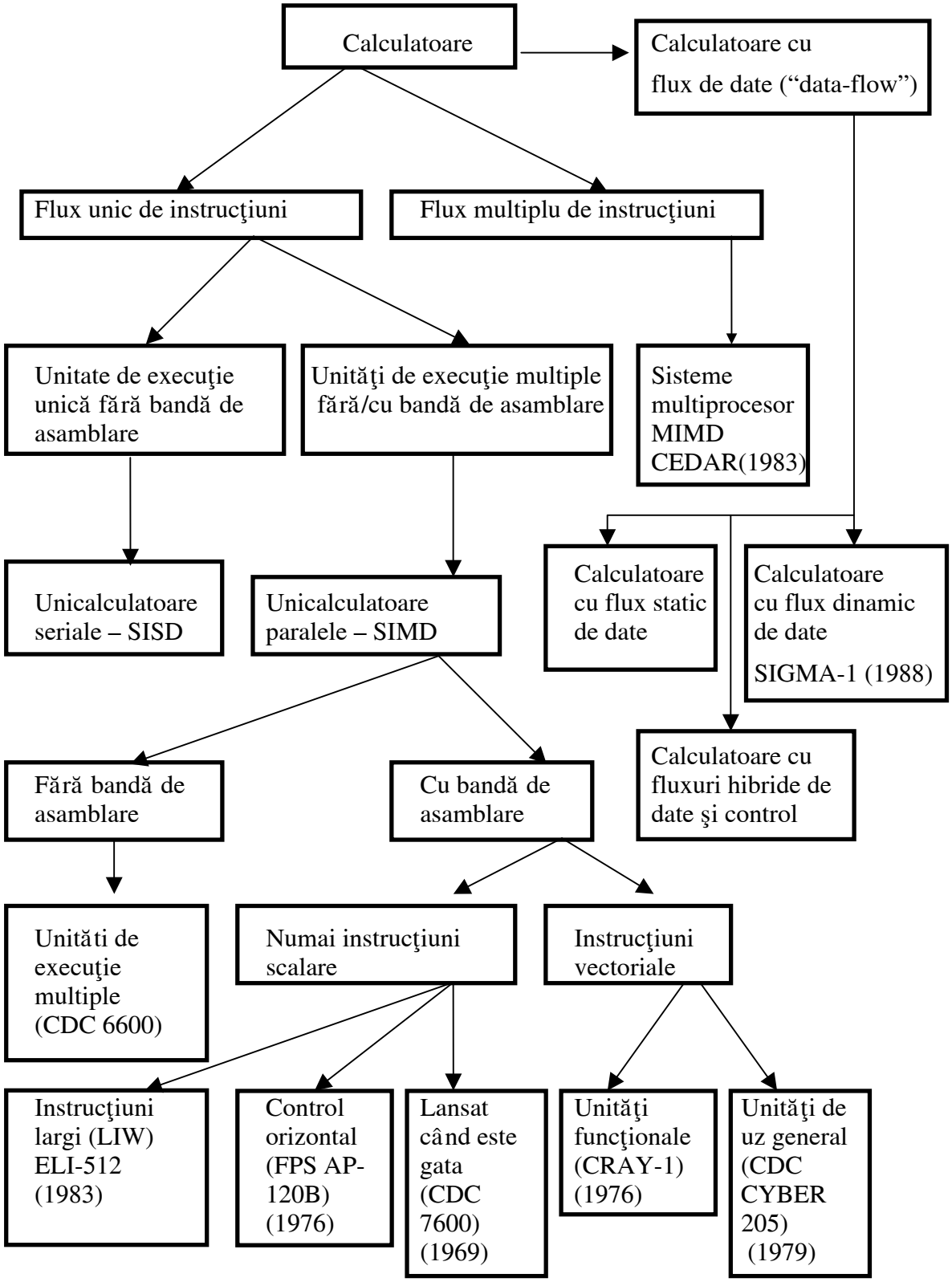


Fiecare modul sistem are un PE, memorie şi interfaţă de I/O. Comunicaţia de date este dată prin mesaje şi nu prin intermediul variabilelor partajate ca la modelul anterior şi urmează un protocol de comunicaţie predeterminat. Gradul de cuplare este mai mic decât la modelul anterior.

Paralelismul este implementat într-un sistem de calcul pe mai multe niveluri, îmbinând mai multe tipuri de paralelism (cum sunt cele de mai sus). Astfel, la nivel de sistem se pot folosi mai multe procesoare, organizate într-un masiv de procesoare sau într-un multiprocesor, iar la nivelul unităţilor centrale de prelucrare se pot implementa mai multe unităţi funcţionale independente, fiecare organizate în bandă de asamblare.

În acest mod s-a ajuns la obţinerea unor performanţe din ce în ce mai ridicate pe sistemele de calcul, cu un cost relativ scăzut.

O clasificare structurală a sistemelor, care reflectă şi gradul de paralelizare a prelucrării datelor, este următoarea:



2. Arhitectura VLIW

Prezentare generală

Primele proiecte VLIW ("Very Long Instruction Word") au fost inspirate din arhitecturile CDC 6600, CRAY-1, IBM 360/91 şi MIPS. În anii '70, multe dintre sistemele masive de procesoare şi dintre procesoarele de prelucrare dedicată de semnale foloseau instrucţiuni lungi în memoria ROM, pentru a calcula transformata Fourier rapidă (FFT) şi alţi algoritmi.

Precursoarele adevărate ale procesoarelor VLIW pot fi considerate maşinile RISC ("Reduced Instruction Set Computer"), al căror principiu a fost expus pentru prima oară de către prof. David A. Patterson din University of California de la Barkeley în 1980. Într-un procesor RISC banda de asamblare permite execuţia în paralel a fazelor unei instrucţiuni obţinând execuţia unei instrucţiuni pe ciclu de ceas. Începând cu proiectele RISC - I de la Universitatea din Berkeley în 1980 şi MIPS de la Universitatea din Stanford în 1981 şi continuând cu procesoarele SPARC de la SUN Microsystems, Transputer T800 de la Inmos, MC 88000 de la Motorola, Am 29000 de la AMD, i860 de la Intel etc., procesoarele RISC au condus la o altă idee, conform căreia să se realizeze citirea simultană a mai multor instrucţiuni şi atribuirea lor mai multor unităţi de execuţie, prin intermediul căilor multiple de date, obţinând astfel o arhitectură LIW ("Long Instruction Word" – cuvânt lung de instrucţiune) sau VLIW.

Conceptul RISC a evoluat în multe direcţii: sunt cunoscute proiectele PIPE ("Parallel Instruction and Pipelined Execution" – instrucţiuni paralele şi execuţie în bandă de asamblare) de la Universitatea din Wisconsin, RIMMS ("Reduced Instruction Set Architecture for Multi-Microprocessor" – arhitectură cu set redus de instrucţiuni pentru multiprocesoare) de la Universitatea din Reading, SIC ("Single Instruction Computer" – calculatoare cu un singur flux de instrucţiuni), WISC ("Writable Instruction Set Computer" – calculatoare cu set de instrucţiuni inscriptibile). De peste 10 ani conceptul RISC a cucerit piaţa procesoarelor aducând un mare câştig de performanţă pentru companiile ce au investit în el (Hewlett-Packard, IBM, Mips, Motorola, Sun, etc). Una din noutăţile din Silicon Valley este proiectarea, de către Intel şi HP, a unui nou procesor de tip VLIW, compatibil atât Intel x86, cât şi HP Precision Architecture.

Legătura cu maşinile VLIW au realizat-o maşinile RISC superscalare. Acestea sunt maşini care execută instrucţiuni multiple pe un ciclu de ceas. Totuşi numai un număr mic de instrucţiuni independente (2 până la 5) pot fi executate într-o singură perioadă de ceas, deoarece ele au puţine unităţi independente de execuţie. De exemplu, un astfel de procesor poate avea o unitate de bandă de asamblare pentru prelucrare în virgulă mobilă, două unităţi de bandă de asamblare pentru prelucrare întregi, o unitate de bandă de asamblare pentru încărcare/memorare şi o initate de procesare ramificaţii. Iniţial, procesoarele VLIW au fost concepute ca o alternativă la procesoarele superscalare RISC. Procesoarele VLIW utilizează, asemenea acestora, principiul microcodificării orizontale pentru a decodifica o instrucţiune VLIW. Procesorul VLIW reprezintă o extensie logică a procesorului RISC. Ca şi un procesor RISC superscalar, o maşină

VLIW execută câteva operaţii simple la un moment dat. Diferenţa constă în modul de rezolvare a dependenţelor la ieşire, care apar atunci când se execută mai multe operaţii în paralel. În cazul sistemelor VLIW, rezolvarea provine de la compilator, care este responsabil cu împachetarea mai multor instrucţiuni simple într-un cuvânt de instrucţiune lungă. Compilatoarele de VLIW sunt responsabile pentru a determina care instrucţiuni depind de altele.

Maşinile VLIW necesită construirea de unităţi logice multiple pentru a păstra instrucţiunile împachetate într-o instrucţiune largă. Acest lucru necesită spaţiu suplimentar pe "chip". Tehnologia RISC permite minimizarea acestui spaţiu. Una din primele maşini cu instrucţiuni lungi, conţinând operaţii multiple pe instrucţiune, a fost AP - 120B, de la Floating Point Systems din 1981. O altă maşină cu instrucţiuni lungi a fost Intel iWarp din 1988. Primul proiect VLIW a fost proiectul ELI - 512 de la Yale din 1983. Primele maşini adevărate VLIW au fost minisupercalculatoarele din anii '80 de la trei companii: Multiflow, Culler şi Cydrome. Ele nu au reprezentat un succes comercial deoarece tehnologia existentă atunci nu a permis acest lucru. Totuşi, experienţa de a scrie compilatoare pentru VLIW este utilizată astăzi în cadrul proiectelor VLIW actuale (în acest sens trebuie subliniate eforturile companiei HP în realizarea unor compilatoare VLIW performante).

Intrarea arhitecturii VLIW pe piaţa comercială s-a realizat mai întâi în zona procesoarelor incorporate ("embedded processors"), cu funcţionalitate integrată (cum sunt procesoarele TriMedia şi TMS320C62x). În prezent, arhitectura VLIW a pătruns şi pe piaţa comercială a calculatoarelor "desktop" (prin procesoarele IA-64 sau Itanium de la Intel şi Crusoe de la Transmeta).

Procesoarele VLIW au mai multe caracteristici:

În primul rând, ele au unităţi funcţionale independente multiple şi, de aceea, pot executa simultan multe operaţii pe aceste unităţi. Operaţiile sunt împachetate într-o instrucţiune foarte lungă. O astfel de instrucţiune poate include operaţii aritmetice (cum ar fi operaţii cu întregi şi în virgulă mobilă), referiri la memorie, operaţii de control, deplasări de date şi ramificaţii. O instrucţiune foarte lungă trebuie să posede un set de câmpuri pentru fiecare unitate funcţională; aceasta poate avea între 256 şi 1024 biţi sau chiar mai mult.

Aceste procesoare posedă numai o unitate de control, ce generează un cuvânt de comandă lung, care controlează explicit unităţile funcţionale prin câmpuri independente. Astfel, ele pot executa numai un singur flux de cod, deoarece controlorul global selectează pentru execuţie o singură instrucţiune foarte lungă, pe fiecare perioadă de ceas. În plus, fiecare operaţie poate fi executată în bandă de asamblare şi solicită pentru execuţie un număr mic de perioade de ceas.

Procesoare utilizează, pentru execuţia în paralel a operaţiilor, un "hardware" simplu şi fără interblocare. Dependenţele de date şi resurse sunt rezolvate înainte de execuţie şi controlate

explicit de către instrucţiuni. De asemenea, procesoarele utilizează căi paralele de date pentru unităţile funcţionale independente multiple şi un volum mic de logică de control şi sincronizare.

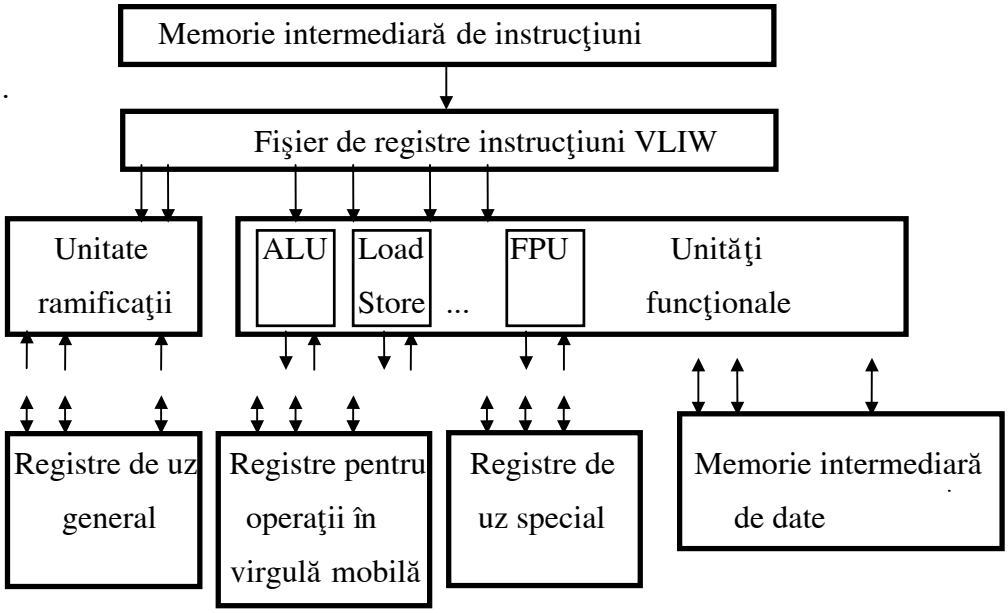
Prezentare functională

Arhitectura VLIW reprezintă una dintre cele mai eficiente soluţii în proiectarea microprocesoarelor. Pentru ca un procesor să lucreze mai repede există două posibilităţi: creşterea frecvenţei ceasului şi execuţia mai multor operaţii pe fiecare ciclu de ceas. Creşterea frecvenţei ceasului necesită inventarea unor procese de fabricare şi adoptarea unor arhitecturi (cum sunt benzile de asamblare mai lungi), care să menţină circuitul cât mai ocupat cu execuţia "task"-urilor. Execuţia mai multor operaţii pe un ciclu de ceas necesită integrarea de unităţi funcţionale multiple pe acelaşi "chip", care să permită execuţia în paralel a „task"-urilor. Problema planificării "task"-urilor este astfel crucială în proiectarea procesoarelor moderne. Procesoarele superscalare actuale realizează acest lucru prin "hardware" pentru rezolvarea dependenţelor de instrucţiuni. "Hardware"-ul de planificare creşte însă geometric cu numărul de unităţi funcţionale şi conduce la limitări de implementare.

Alternativa este de a lăsa "software"-ul să facă planificarea "task"-urilor, ceea ce chiar realizează arhitectura VLIW. Astfel, un compilator optimizat poate examina programul, poate găsi toate instrucţiunile fără dependenţe şi le poate împacheta într-o instrucţiune foarte lungă, pentru ca apoi să le execute concurent pe un număr egal de unităţi funcţionale. O astfel de instrucţiune lungă (meta-instrucţiune) conţine multe câmpuri mici, fiecare codificând direct o operaţie pentru o unitate funcţională particulară.

O arhitectura generală VLIW ar fi:

Figura 2.1.



Un astfel de procesor VLIW constă într-o colecţie de unităţi funcţionale (sumatoare, multiplicatoare, anticipare ramificaţii etc), conectate printr-o magistrală, plus registre şi memorii intermediare; acest lucru este foarte bun pentru că se câştigă "hardware" iar durata execuţiei este limitată doar de unităţile funcţionale proprii.

O funcţie importantă a procesoarelor VLIW este aceea că pot implementa vechile seturi de instrucţiuni CISC mai bine chiar decât procesoarele RISC. Aceasta pentru că programarea procesorului VLIW este foarte asemănătoare cu scrierea de microcod, putându-se conserva astfel programul, utilizându-se instrucţiunile CISC complexe (cum ar fi LODS şi STOS de la Intel x86) implementate de procesoarele CISC prin microprograme de microcod ROM pe circuitul procesor.

Microcodul este cel mai scăzut nivel de limbaj, sincronizând porţi şi magistrale şi transferând datele între unităţile funcţionale. Procesorul RISC a eliminat microcodul în favoarea logicii cablate mai rapide, iar procesorul VLIW a scos microcodul în afara chipului procesor şi l-a plasat la compilator; astfel, emularea de cod CISC se execută mai rapid pe VLIW decât pe RISC.

În schimb, compilatorul este mai complex decât la sistemele RISC; de aceea, se poate realiza o îmbinare "hardware-software" a implementării VLIW (în capitolele următoare sunt prezentate astfel de tehnici). Există o varietate de arhitecturi, care demonstrează multe sau toate caracteristicile descrise mai sus. Un model tipic pentru o arhitectură VLIW ar putea fi:

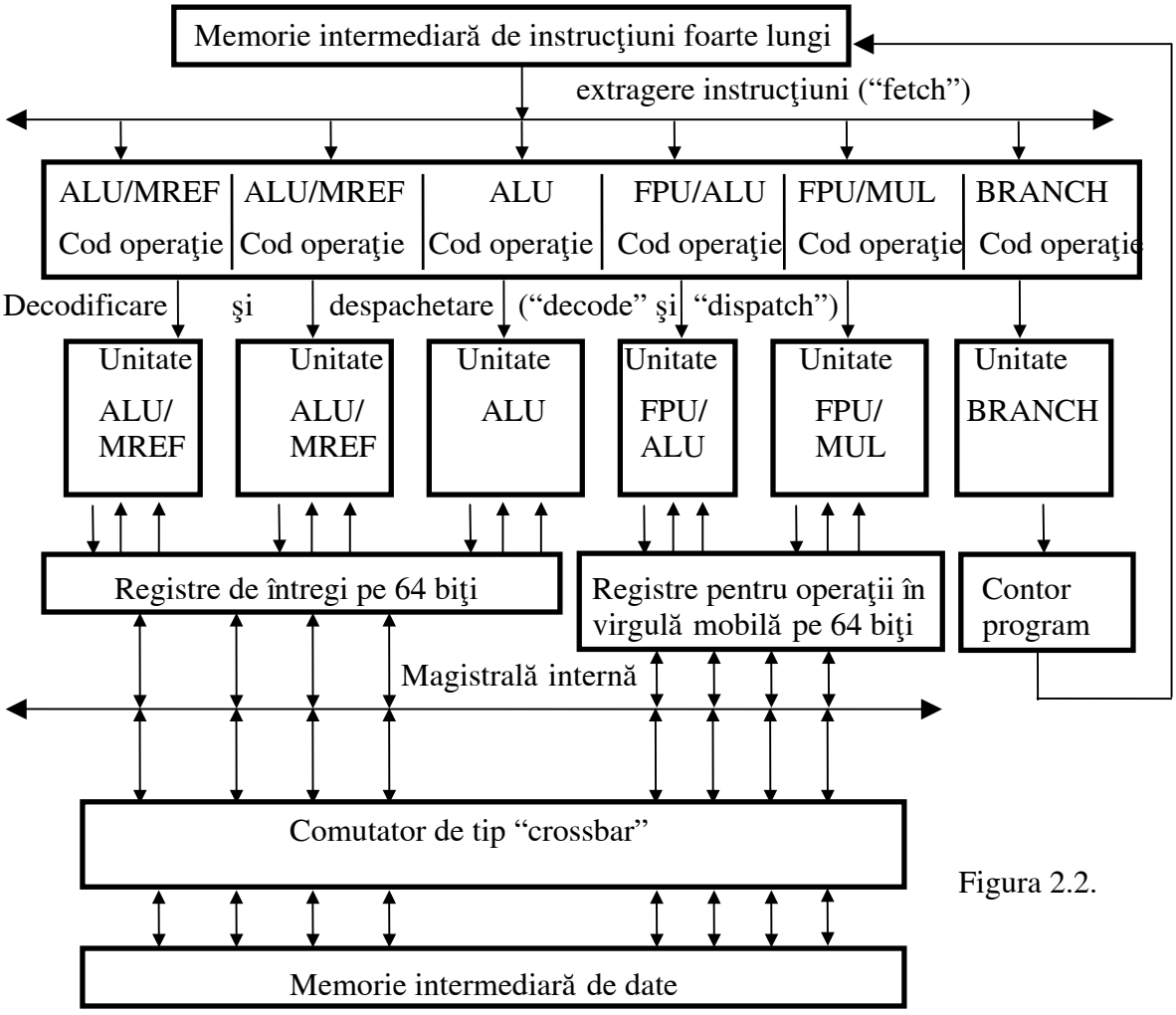


Figura 2.2.

Legendă:

ALU/MREF – operaţie aritmetică-logică/ referire la memorie (“load/store”)
FPU – operaţie în virgulă mobilă
FPU/ALU – operaţie de adunare/scădere în virgulă mobilă
FPU/MUL – operaţie de multiplicare în virgulă mobilă
BRANCH – ramificaţie
- codul operaţiei conţine şi biţii de control ai operaţiilor

Căile de date constau într-un număr de unităţi funcţionale (pentru întregi, virgulă mobilă şi de salt), conectate prin registre globale şi speciale multiport şi prin intermediul unei reţele de comutare de tip “crossbar” la o memorie intermediară de date. Fiecare unitate funcţională este controlată de un set independent de câmpuri ale instrucţiunii, extrasă din memoria intermediară de instrucţiuni VLIW în registrul instrucţiunii, şi de registre specifice de adresă pentru operanzi şi rezultate. Există de asemenea o unitate de salt controlată tot de instrucţiune. Ea generează adresa instrucţiunii următoare bazându-se pe câmpurile instrucţiunii curente şi condiţiile de cod memorate.

Un procesor VLIW poate avea un singur fişier de registre globale (ca procesorul TRACE/7) sau fişiere multiple. De asemenea poate avea unităţi omogene (ca la IBM) sau heterogene (ca la TRACE/7). Procesoarele superscalare au unităţi funcţionale multiple şi fişiere mari de registre, dar ele utilizează decodificatoare de instrucţiuni complexe pentru extragerea operanzilor şi procesarea lor în paralel. Există procesoare care au câteva dintre caracteristicile procesoarelor VLIW. Un exemplu este procesorul Intel i860 care are un mod dual de prelucrare instrucţiuni, în care acţionează ca un mic procesor VLIW.

Datorită lungimii foarte mari a cuvîntului de instrucţiune şi pentru a elimina creşterea complexităţii “hardware”-ului, procesoarele VLIW utilizează un compilator complex care preia instrucţiunile şi le pune într-o secvenţă de cod liniară, astfel încât ele să formeze un cuvânt de instrucţiune foarte lung, din care instrucţiunile pot fi executate în paralel, evitându-se conflictele de date. Memoriile intermediare se utilizează pentru a creşte lăţimea de bandă (“bandwidth”), permiţând accesul rapid la instrucţiuni şi operanzi. Instrucţiunile din cuvântul instrucţiune foarte lung sunt procesate folosind benzi de asamblare multiple.

Pentru a forma şi executa un cuvânt de instrucţiune foarte lung, compilatorul utilizează anumite tehnici, cum ar fi “*trace scheduling*“, “*software pipelining*” şi altele, care sunt prezentate într-un capitol următor. Un compilator VLIW împachetează grupuri de operaţii independente într-un cuvânt de instrucţiune foarte lung, într-un mod care utilizează cât mai eficient unităţile funcţionale pe timpul fiecărui ciclu de ceas. Compilatorul descoperă toate dependenţele de date şi apoi determină cum să rezolve aceste dependenţe, reordonând întregul program prin deplasări ale blocurilor de cod; acest lucru este realizat la procesoarele RISC superscalare printr-un “hardware” adecvat de reordonare şi anticipare. Pentru procesoarele

VLIW, ca şi pentru cele RISC, ramificaţiile sunt inamicul cel mai important al execuţiei. Dacă procesoarele RISC rezolvă ramificaţiile printr-o logică "hardware" de anticipare, procesoarele VLIW le lasă pentru compilator, care le planifică prin intermediul blocurilor de cod folosite pentru a genera instrucţiunea foarte lungă; compilatoarele de VLIW execută o analiză sofisticată a codului şi pot deplasa cât mai mult cod deasupra ramificaţiei anticipate, cod ce poate fi însă oricând refăcut, dacă se cere acest lucru.

Un "hardware" adecvat pe procesorul VLIW poate oferi un anumit suport pentru compilator. De exemplu, o operaţie de salt cu căi multiple permite câtorva ramificaţii să fie împachetate într-o singură instrucţiune lungă şi să fie executate în acelaşi ciclu de ceas.

Deoarece compilatorul trebuie să cunoască microarhitectura procesorului VLIW pentru care compilează codul program, orice cod maşină pe care îl produce rulează bine numai pe procesorul respectiv; orice rulare pe o nouă generaţie de procesor VLIW presupune o recompilare a codului. Compatibilitatea cu noile generaţii poate fi totuşi păstrată prin folosirea unor tehnici "hardware" sau "software" adecvate. Datorită acestor facilităţi, performanţa procesoarelor VLIW este de 10 până la 30 de ori mai bună decât un procesor convenţional.

Modelele maş inii de stare

Un procesor este reprezentat ca o uniune dintre calea de control şi calea de date. Calea de control poate fi modelată cu o maşină de stări finite, care generează semnale de control pentru calea de date şi reacţionează la semnalele de stare primite de la calea de date. Modificând structura căii de control se pot diferenţia clase de arhitecturi diferite.

Un prim model este modelul maşinii Moore, utilizată într-un model clasic de uniprocesor SISD microprogramat:

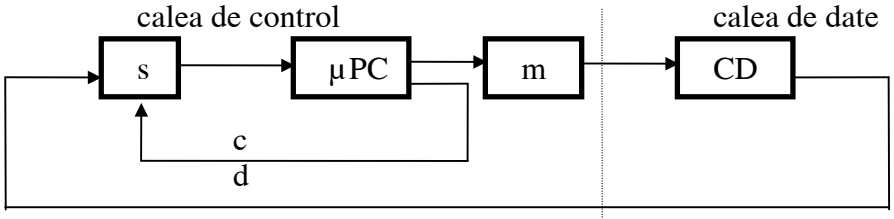


Figura 2.3.

µPC - contorul de microprogram

m - microcodul

s - secvenţiatorul şi câmpurile de control pentru calea de date

c - starea căii de control

d - starea căii de date (condiţia de cod)

Funcţia m (de ieşire) depinde numai de starea variabilei µPC care va implica execuţia unei instrucţiuni pe calea de date. Funcţia s (de stare următoare) depinde de starea căii de date, a căii de control şi intrările externe.

Un alt model este modelul VLIW:

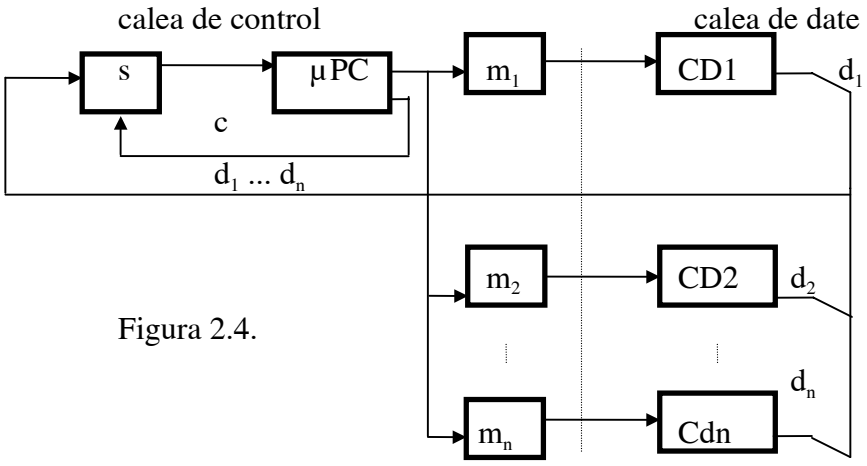


Figura 2.4.

Procesorul VLIW are unităţi funcţionale multiple, fiecare corespunzând căii de date a unui procesor SISD. Modelul căii de control conţine o ieşire separată, mapând funcţiile m_1, ..., m_n pentru fiecare unitate funcţională din calea de date. Funcţia de stare următoare s va depinde de starea fiecărei unităţi funcţionale, d_1, ..., d_n, starea căii de control şi intrările externe.

Modelul tradiţional SIMD este o simplificare a modelului VLIW. Astfel, ieşirea unei funcţii singulare m va fi distribuită la fiecare unitate funcţională. Dacă funcţiile m_1, ..., m_n ale unui model VLIW sunt identice şi egale cu funcţia m a unui model SIMD, atunci cele două maşini au o funcţionare echivalentă; aceasta implică faptul că se poate programa un procesor VLIW să emuleze funcţionarea unui procesor SIMD. De aceea se poate considera maşina VLIW ca un superset funcţional al maşinii SIMD.

Următorul model prezentat este modelul XIMD:

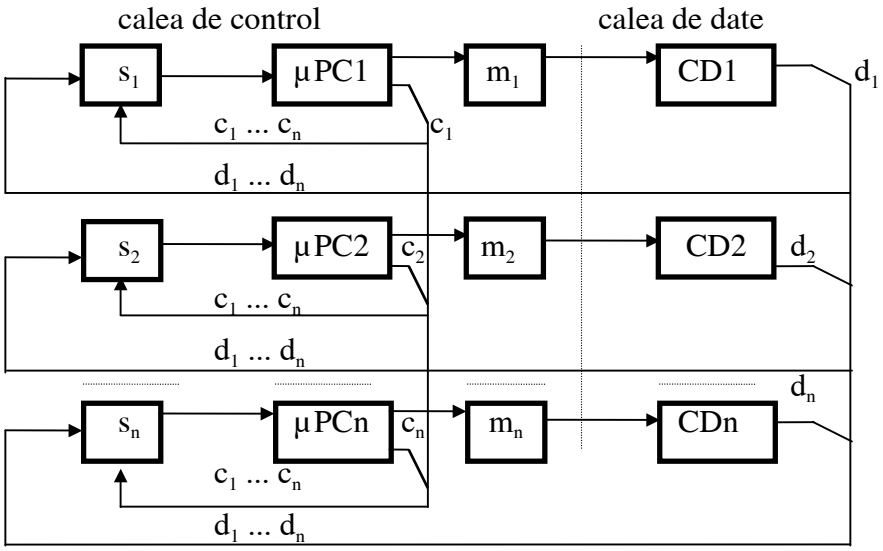


Figura 2.5.

Apare duplicarea secvenţiatorului din calea de control (care şi ea se duplică). Se realizează o mapare a căilor de date la fiecare secvenţiator.

Ca şi VLIW, modelul XIMD poate executa operaţii unice pe fiecare unitate funcţională; în plus, poate executa câte o operaţie unică de control pentru fiecare unitate funcţională. Se poate considera maşina XIMD ca un superset funcţional de VLIW; astfel, dacă funcţiile m_1, ..., m_n sunt identice şi μPC_1, ..., μPc_n au valori iniţiale identice, atunci maşina XIMD va fi echivalentă funcţional cu maşina VLIW.

Complexitatea controlului operaţiilor, care pot fi executate pe un model XIMD, este afectată de trei factori:

- numărul de valori ale fiecărei variabile de stare a căilor de control;

- numărul de valori ale fiecărei variabile de stare a căilor de date;

- complexitatea şi programabilitatea fiecărei funcţii de stare următoare.

Primii doi factori afectează volumul de informaţie care poate fi partajat de-a lungul fluxurilor de instrucţiuni, în timp ce al treilea factor determină cum un flux de instrucţiuni poate diferenţia între diferite situaţii apărute în alte unităţi funcţionale.

Un ultim model reliefat este modelul MIMD:

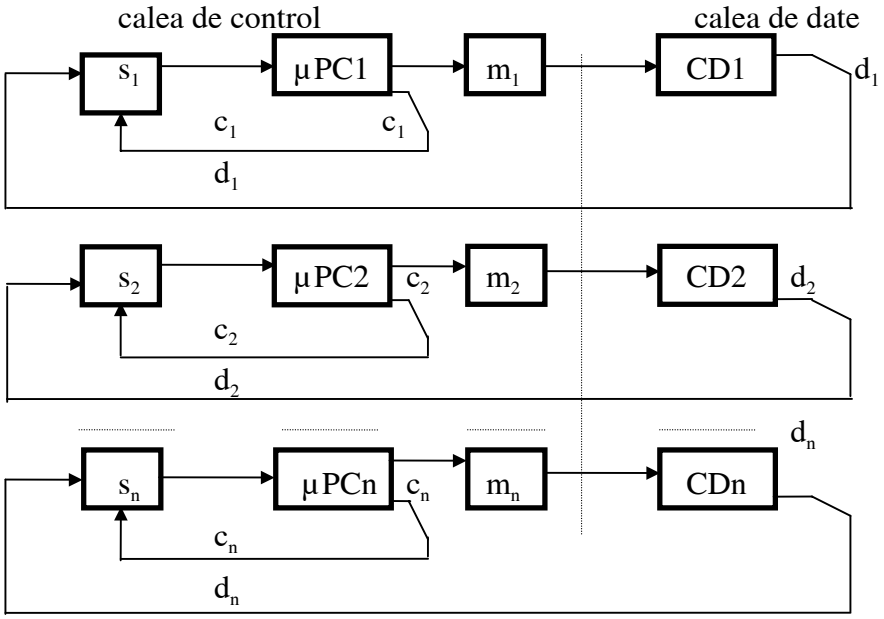


Figura 2.6.

Selectând ca funcţiile m_i să nu depindă de starea altor unităţi funcţionale, modelul XIMD poate fi funcţional echivalent cu un model MIMD.

3. Tehnici "hardware" de creştere a paralelismului pentru procesoarele VLIW

Utilizarea unui "hardware", care să monitorizeze fluxul de instrucţiuni şi să grupeze mai multe instrucţiuni din flux într-o instrucţiune VLIW prin procesorul însuşi, constituie o opţiune posibilă pentru furnizarea instrucţiunilor multiple. Aceasta ar permite ca fluxul secvenţial de instrucţiuni să fie preluat direct la procesor, cu execuţia unor părţi de cod accelerate atunci când este posibil, păstrându-se în acelaşi timp compatibilitatea.

În 1988, Melvin, Shebanov şi Patt au propus un "hardware" pentru compactarea microoperaţiilor, generate prin citirea instrucţiunilor, într-un tampon de instrucţiuni decodificate. Implementarea a fost numită unitate de umplere ("fill unit" – FU) şi funcţionează după cum se descrie în continuare.

Un registru tampon, de preîncărcare instrucţiuni ("prefetch"), citeşte instrucţiunile din memoria principală sau memoria intermediară de instrucţiuni internă şi trimite microinstrucţiunile corespunzătoare la unitatea FU. Aceasta le colectează împreună şi le plasează ca un cuvânt multinod într-un memorie intermediară de instrucţiuni decodificate DIC ("Decoded Instruction Cache"). Referinţele microoperaţiilor au fost redenumite într-un spaţiu de adrese al DIC şi microoperaţiile de la o singură instrucţiune pot fi separate în două cuvinte multinod. Scrierea liniilor din DIC se termină (adică un cuvânt multinod a fost finalizat) ori de câte ori a fost înregistrată o ramificaţie sau nu a mai rămas în cuvântul multinod nici un slot (fantă de întârziere) de microoperaţie gol. Microoperaţiile din cuvâtul multinod pot avea dependenţe de date, care pot fi rezolvate mai târziu, printr-o planificare dinamică "hardware" întreţesută. Scopul FU este de a se obţine un cuvânt atomic larg la planificarea dinamică. Pentru recuperarea stării se utilizează un punct de test ("checkpoint") de sistem, astfel că maşina poate executa revenire ("back up") la punctul de test adecvat şi poate rula în mod secvenţial ori de câte ori apare o excepţie. Implementarea conţine şi posibilitatea redenumirii registrelor, pentru a codifica cererile de accelerare ("forwarding") de la cuvintele multinod, iar registrele sunt redenumite la nivelul arhitectural al setului de instrucţiuni ISA ("Instruction Set Architecture"), pentru a îmbunătăţi performanţa.

4. Tehnici "software" de creştere a paralelismului pentru procesoarele VLIW

Maşinile VLIW pun mare accent pe optimizarea compilatorului, degrevând "hardware"-ul intern de mai multe funcţii importante, cum ar fi:

- evitarea dependenţelor dintre operaţii;
- compactarea mai multor operaţii independente într-o singură instrucţiune lungă, capabilă să se execute complet în paralel (lucru întărit şi de planificarea "software" a resurselor necesare pentru executarea în paralel a tuturor operaţiilor din instrucţiunea lungă);
- anticiparea cel puţin parţială a ramificaţiilor condiţionale, extrăgându-se astfel un paralelism ridicat la nivel de instrucţiune.

O instrucţiune VLIW ("Very Long Instruction Word") poate include mai multe operaţii, cum ar fi, de exemplu: două operaţii întregi, două operaţii în virgulă mobilă, două referiri la memorie şi o ramificaţie; aceste operaţii se execută în paralel pe resursele procesorului, care sunt planificate de către un compilator optimizat.

Pentru a creşte paralelismul procesoarelor superscalare în general, VLIW în special, au fost dezvoltate şi se folosesc diferite tehnici de compilare. Fiecare nouă proiectare a unui procesor necesită şi îmbunătăţirea tehnicilor de compilare.

Tehnicile dezvoltate în acest capitol ajută la creşterea vitezei de execuţie ("speedup") a arhitecturilor superscalare, în general, a celor VLIW, în special.

Scăderea performanţelor procesoarelor superscalare se datorează mai ales următoarelor probleme: limitarea "hardware" a paralelismului; neanticiparea ramificaţiilor şi neevitarea blocărilor benzilor de asamblare; dificultăţi în menţinerea unei viteze susţinute de execuţie în cazul apariţiei mai multor ramificaţii condiţionale pe ciclul de ceas.

Tehnicile de planificare "software", împreună cu un "hardware" adecvat, rezolvă aceste neajunsuri şi măresc viteza de execuţie.

Din cadrul tehnicilor simple de compilare se pot enumera: asamblarea prin program ("software pipelining"); planificarea urmelor ("trace scheduling"); planificarea prin filtrare ("percolation scheduling") etc.

În cadrul tehnicilor complexe de compilare se disting: deplasarea speculativă a operaţiilor de încărcare/memorare în afara buclelor ("speculative load/store motion"); nespecularea ("unspeculation"); planificarea globală prin descompunerea buclelor şi redenumirea registrelor ("global scheduling"); planificarea lărgită ("enhanced pipeline scheduling"); planificare cu gărzi ("sentinel scheduling"); combinarea limitată a operaţiilor ("limited combining"); expandarea blocurilor de bază ("basic block expansion"); planificarea cu reacţie inversă ("profiling directed feedback") etc.

Înaintea aplicării acestor tehnici trebuie determinate dependenţele programului, prin construirea grafurile de dependenţe pentru programele compilate.

În continuare se prezintă pe scurt aceste tehnici.

Tehnica de asamblare prin program ("software pipelining")

"Software pipelining" este o tehnică utilizată atât de sistemele uniprocesor, cât şi de sistemele multiprocesor. Ea constă în distribuţia unei bucle pe mai multe procesoare sau unităţi funcţionale (în cazul procesoarelor VLIW sau alte procesoare superscalare), ceea ce se realizează prin atribuirea stărilor independente la procesoare sau unităţi funcţionale independente şi, apoi, înlănţuirea continuă a operaţiilor.

Tehnica "software pipelining" reprezintă o metodă pentru optimizarea buclelor, prin suprapunerea operaţiilor provenind de la diferite iteraţii. Acest lucru este realizat prin partiţionarea corpului unui program în segmente, care pot fi procesate în mod bandă de asamblare ("pipeline"); buclele sunt reorganizate astfel încât fiecare iteraţie din codul optimizat prin "software pipelining" să fie compusă din secvenţe de instrucţiuni alese din diferite iteraţii

ale segmentului original de cod. Există un anumit cod de de activare, care este necesar înaintea începerii buclei, ca şi un cod pentru terminare după ce bucla este compilată.

Planificatorul de "task"-uri întreţese instrucţiuni de la diferite iteraţii ale buclei, asamblând operaţiile de încărcare de la memorie, apoi toate operaţiile aritmetice-logice, iar, în final, toate memorările. O buclă asamblată întreţese instrucţiuni provenind din diferite iteraţii, fără a se executa desfăşurarea ei. Metoda reprezintă o traducere prin program a algoritmului Tomasulo, executat prin "hardware". Bucla asamblată prin program poate conţine, de exemplu, o încărcare ("load"), o adunare ("add") şi o memorare ("store"), fiecare dintr-o iteraţie diferită.

Diagrama spaţiu-timp pentru transformarea "pipelining" constă în următoarele faze: toate stările din buclă sunt executate simultan, pe mai multe procesoare sau unităţi funcţionale, această fază corespunde distribuirii stărilor în spaţiu (la procesoare sau unităţi funcţionale); apoi, iteraţiile sunt asamblate în timp. Timpul de execuţie depinde de numărul de iteraţii ale buclei şi numărul de stări independente pentru fiecare iteraţie, care furnizează adâncimea maximă a unităţilor în bandă de asamblare.

Tehnica "software pipelining" îmbunătăţeşte performanţa din două motive:

- se îmbunătăţeşte execuţia în bandă de asamblare prin planificarea operaţiilor cu dependenţe mai îndepărtate;

- creşte paralelismul datorită segmentelor multiple ale buclei, care pot fi executate simultan pe o maşină superscalară.

Desfăş urarea buclelor ("loop unrolling")

"Desfăşurarea buclelor" este o metodă utilizată în general pentru bucle fără dependenţe de date între iteraţii. Prin această tehnică se expandează iteraţiile unei bucle de multiple ori şi se replanifică instrucţiunile din bucla desfăşurată, crescând astfel distanţa lexicală dintre instrucţiunile cu dependenţe.

Această optimizare este disponibilă datorită compilatorului. Desfăşurarea buclei este valabilă numai dacă instrucţiunile desfăşurate sunt planificate în vederea minimizării dependenţelor. O problemă majoră a desfăşurării buclei este faptul că numărul de registre cerute creşte. Pentru a evita conflictele de utilizare a registrelor la execuţia concurentă a iteraţiilor, se utilizează registre diferite pentru fiecare iteraţie.

În programele reale nu se cunoaşte, de regulă, marginea superioară a unei bucle; se presupune că este **n** şi se doreşte desfăşurarea unei bucle de **k**-ori; in loc de o singură buclă desfăşurată, se va genera o pereche de bucle. Prima buclă execută de (**n mod k**)-ori şi are corpul ca al buclei originale. Această versiunea desfăşurată a buclei este înconjurată de o a doua buclă, exterioară ei, care iterează de (**n div k**)-ori. Numărul de registre poate limita desfăşurarea printr-un factor de desfăşurare UF ("unrolling factor"); aceasta se întâmplă mai ales la operaţii în

virgulă mobilă. Majoritatea programelor sintetice de test pot fi desfăşurate de mai multe ori, deoarece corpul buclei conţine un număr mic de operaţii.

Un avantaj al desfăşurării buclei este faptul că fiecare iteraţie desfăşurată necesită numai o buclă de control. Câştigul de planificare pe bucla desfăşurată este chiar mai mare decât pe bucla originală. Aceasta pentru că desfăşurarea buclei oferă posibilitatea planificării mai multor calcule. Acest mod de planificare a buclei implică faptul că citirile şi scrierile de la/la memorie sunt independente şi pot fi interschimbate. Tehnica de planificare prin bucle desfăşurate este simplă şi utilă, datorită creşterii mărimii fragmentelor de cod, care pot fi planificate efectiv. Această transformare, realizată pe timpul compilării, este similară algoritmului Tomasulo (cu redenumirea registrelor şi execuţie speculativă"out-of-order").

Dezavantajul metodei: este dificil de obţinut o planificare optimă, pentru că bucla trebuie să fie desfăşurată de un număr suficient de ori pentru a ţine banda de asamblare complet utilizată.

Acest lucru determină ca metoda să nu fie atât de efectivă ca metoda "software pipelining".

Tehnica planificării urmelor ("trace scheduling")

Prin această tehnică se generează un paralelism suplimentar. Tehnica a fost dezvoltată iniţial special pentru procesoarele VLIW şi reprezintă o combinaţie de două procese separate. Primul proces, numit *selectarea urmelor* ("trace selection"), încearcă să găsească cea mai potrivită secvenţă de operaţii care vor fi asamblate împreună într-un număr mic de instrucţiuni; această secvenţă se numeşte *urmă* ("trace"). Pentru a genera urme lungi, se foloseşte tehnica desfăşurării buclei ("loop unrolling") în timp ce ramificaţiile buclei sunt executate. O dată ce o urmă este selectată, următorul proces, numit *compactarea urmelor* ("trace compaction"), încearcă să comprime urma într-un număr mic de instrucţiuni lungi. Operaţia de compactare a urmelor deplasează operaţiile dintr-o secvenţă, împachetându-le, pe cât este posibil, în câteva instrucţiuni lungi.

Există două limitări în compactarea urmelor:

- dependenţele de date, care forţează o ordine parţială de operare;

- punctele de salt, care creează locuri de-a lungul cărora codul nu poate fi deplasat uşor; în esenţă, codul trebuie să fie compactat în cea mai scurtă secvenţă posibilă, care conservă dependenţele de date, iar ramificaţiile sunt impedimentul principal al acestui proces.

În această tehnică, compilatorul rezolvă conflictele de resurse şi utilizează planificarea pe urme pentru ca aplicaţiile program să fie partiţionate în căi libere de bucle (aşa-numitele urme); utilizând algoritmi specifici, compilatorul determină care este cea mai bună urmă de executat.

Această planificare poate include o deplasare de cod, care ar putea cauza inconsistenţe de date când se execută ramificaţii şi, pentru a compensa aceste lucruri, compilatorul adaugă

operaţii suplimentare de la alte urme. Procesul este repetat până când toate urmele sunt planificate. Compilatorul verifică de asemenea dacă există referiri simultane la memorie, pentru a nu le executa în paralel.

Marele avantaj al tehnicii "trace scheduling" faţă de tehnica "software pipelining" este faptul că tehnica "trace scheduling" include o metodă care deplasează codul de-a lungul ramificaţiilor.

Tehnica de planificare prin filtrarea fluxului de instrucţiuni ("percolation scheduling")

"Percolation scheduling" reprezintă o tehnică folosită efectiv pentru planificarea operaţiilor multiple provenite din codurile scalare. Ca şi tehnica "trace scheduling", această tehnică extrage operaţii independente din codul scalar al programului, pe care le asamblează într-o instrucţiune VLIW de lungime corespunzând cu numărul de unităţi funcţionale ale procesorului VLIW. Prin tehnica "percolation scheduling" deplasarea codului este filtrată la fiecare moment, pentru a furniza o instrucţiune lungă, care se poate executa fără întârziere.

Tehnica "percolation scheduling", ca şi cele prezentate anterior, încearcă să crească volumul de paralelism local al instrucţiunii, care poate fi exploatat de către o maşină care execută mai mult de o instrucţiune pe fiecare ciclu maşină.

Deplasarea speculativă de "load/store"-uri în afara buclelor

Această tehnică urmăreşte evitarea acceselor la memorie din bucle, în special încărcări de la memorie ("load"). Instrucţiunile de încărcare tind să introducă întârzieri într-o operaţie a procesorului, în special dacă datele cerute nu rezidă în memoria intermediară („cache").

În esenţă, aceasta reprezintă o generalizare a ideii de a deplasa instrucţiuni invariante de buclă ("loop-invariant instructions") în afara buclelor, cu capabilităţi suplimentare de deplasare a încărcărilor şi memorărilor, care sunt executate condiţional (adică fac parte dintr-un "if").

Această strategie poate fi extinsă la proceduri generale, utilizând un program de analiză inter-procedurală (mai ales că nu se degradează semnificativ încărcarea ("overhead"-ul)).

Tehnica nespeculării ("unspecullation scheduling")

Operaţiile speculative (operaţii ale căror rezultate nu contribuie întotdeauna la rezultatul final al programului) pot apărea în codul intemediar datorită tehnicilor variate de manipulare a codului, care au fost aplicate, sau datorită faptului că astfel de operaţii au fost prezente în programul original. O secvenţă de operaţii devine speculativă dacă este deplasată înaintea unei ramificaţii condiţionate, care impune ca operaţia să fie executată pentru a obţine rezultatul dorit al programului. Astfel, în acele cazuri când ramificaţia condiţionată conduce prin calea unde au fost localizate instrucţiunile, execuţiile instrucţiunilor pot cauza degradarea performanţelor. Obiectivul nespeculării este de a descoperi şi reduce numărul operaţiilor speculative, aceasta

evitând degradarea potenţială a performanţelor, prin gruparea operaţiilor speculative în jos pe una sau două destinaţii ale ramificaţiei condiţionate, transformându-le în operaţii nespeculative. Gruparea instrucţiunilor conduce la posibilitatea existenţei unui număr de blocuri de bază cu o singură intrare şi o singură ieşire, care buclează; stările imbricate "if-then-else-endif" sunt exemple de asemenea grupuri.

Tehnicile de planificare globală ("global scheduling") şi de planificare extinsă ("enhanced pipeline scheduling")

Regiunile din programe sunt compactate prin combinarea planificării globale ("global scheduling") şi a planificării extinse ("enhanced pipeline scheduling"), începând de la regiunile cele mai interioare (bucle) şi sfârşind cu cele mai exterioare (întreaga procedură).

Aceste două tehnici au fost folosite cu succes în compilatorul xl_c pentru sistemul IBM RS/6000.

Prin planificarea globală este paralelizat codul secvenţial eliminându-se antidependenţele şi dependenţele de ieşire prin redenumirea registrelor, aşa cum se cere în timpul planificării. Un atribut al fluxului de date, la începutul fiecărui bloc de bază, indică ce operaţii sunt disponibile pentru deplasarea în sus prin acest bloc de bază. Planificarea globală constă în alegerea celei mai puţin utilizate operaţii din setul de operaţii, care se poate deplasa la un moment dat, deplasând toate operaţiile în acel punct, contabilizând copiile pentru marginile care unesc căile deplasării de cod, dar nu sunt pe ele şi actualizând atributele fluxului de date ale blocurilor de bază numai pe căile care au fost traversate de către operaţiile deplasate. Deplasările de cod sunt globale, fără trecerea prin transformările atomice ale planificării prin filtrare ("percolation scheduling"), aceasta pentru o mai bună eficienţă. Pentru executarea deplasărilor de cod, se utilizează o reprezentare intermediară care este direct executabilă mai degrabă ca un cod secvenţial RISC, decât ca un cod VLIW. Algoritmul poate fi utilizat şi pentru a genera cod paralelizat pentru procesoare superscalare cu unităţi ALU multiple.

Pentru a compacta buclele, planificarea extinsă plasează o barieră (gardă) la punctul curent de planificare, şi lasă planificarea globală să caute cea mai bună operaţie pe toate căile care pot traversa marginile din spate ale buclei, dar nu bariera. Buclele sunt desfăcute anterior planificării şi este efectuată redenumirea domeniului de lucru pentru a creşte oportunităţile de planificare.

În continuare este analizată tehnica de planificare extinsă a benzii de asamblare ("enhanced pipeline scheduling").

Această tehnică este utilizată pentru planificarea buclelor conţinând constructori condiţionali. Are avantajul că corpul buclei rămâne intact în timpul planificării şi astfel banda de asamblare poate fi oprită după orice pas terminat.

Modelul maşini de stare este următorul:

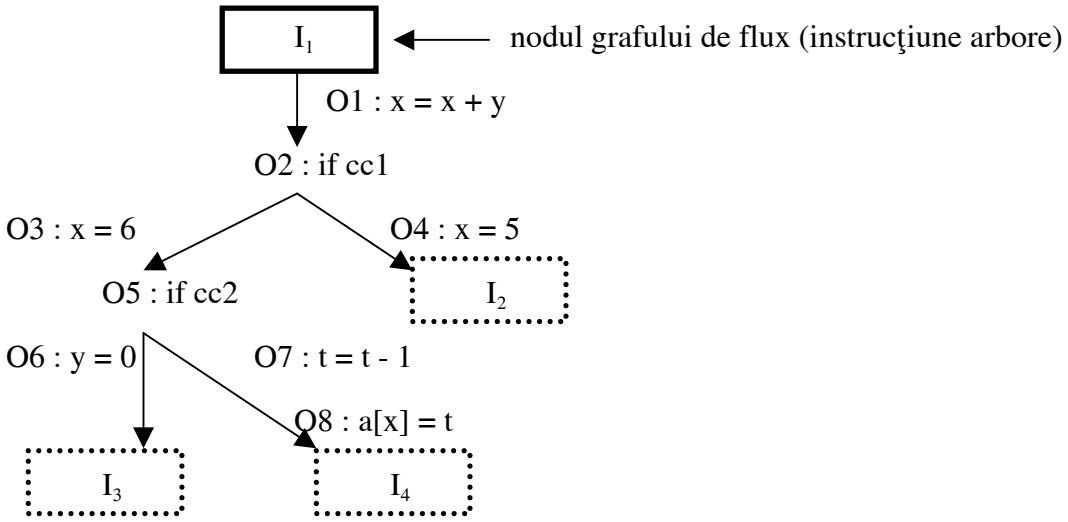


Figura 4.1. Instrucţiune arbore ("tree instruction") utilizată în algoritmul planificării extinse

$I_2 – I_4$: etichetele altor noduri din graful de flux

cc1, cc2: coduri condiţie ce sunt calculate înainte ca nodul să fi intrat în execuţie şi numai în acele stări, de-a lungul unei căi de la rădăcină la o frunză, care sunt executate.

Toate operaţiile pe căile selectate sunt executate concurent, utilizând vechile valori pentru toţi operanzii. De exemplu, în figura 4.1, atribuirea lui t prin O7 nu afectează utilizarea lui t prin O8. Dupa ce toate operaţiile au fost executate, rezultatele sunt scrise la registre sau memorie, şi controlul este transferat de la instrucţiunea corespunzătoare etichetei la frunză.

Tehnica planificării extinse creşte paralelismul pe o arhitectură cu căi mulptiple de salt, permiţând ca mai multe operaţii de salt condiţionat să se deplaseze într-un nod al grafului de flux. La execuţia condiţionată, operaţiile pot fi plasate pe orice ramificaţie din nodul grafului de flux, spre deosebire de modelul tradiţional de maşină, în care toate operaţiile trebuie să preceadă orice operaţie de salt condiţionat.

Acest lucru permite migrarea codului pe arhitecturi mai performante. Migrarea aplicaţiilor este asistată automat de calculator prin translatarea codului de pe o maşină pe alta. Procedeul poate fi o cale de migrare şi între diferite tipuri de arhitecturi (de la CISC la RISC, la arhitecturi superscalare sau la arhitecturi VLIW). Migrarea are loc la timpul de execuţie.

Tehnica de planificare cu gărzi ("sentinel scheduling")

Tehnica a fost dezvoltată de S. A. Mahlke et al. [98] şi funcţionează astfel:

Funcţia "**reducere_dependenţe_graf**" deplasează dependenţele de control de la un graf de dependenţe pentru a permite deplasarea speculativă de cod.

Instrucţiunile pot fi executate speculativ, dacă ele sunt permise ca speculative în arhitectură: de exemplu, ramificaţiile, apelările de subrutine şi instrucţiunile de intrare/ieşire (i/o) nu pot fi executate speculativ, la fel şi instrucţiunile de memorare fără suport de memorie.

Funcţia "**sentinel_scheduling**" utilizează graful redus de dependenţe pentru a executa planificarea listei tuturor instrucţiunilor dintr-un "superblock".

Tehnica de combinare limitată a operaţiilor ("limited combining")

Această tehnică scanează codul scalar, căutând oportunităţi pentru a reduce lungimea căii de cod prin combinarea operaţiilor colapsabile (cum ar fi: A r4 = r5; B r6 = r4@r7, unde r4 este utilizat în noua instrucţiune C r6=r5@r7). Acest lucru este optimizat, când ambele operaţii sunt înăuntrul aceluiaşi bloc de bază. Tehnica de **combinare limitată** poate acoperi un număr de blocuri de bază căutând operaţii colapsabile, incluzând eventual puncte de intersecţie, prin duplicarea codului. Această transformare va **căuta** o secvenţă de instrucţiuni, începând de la o instrucţiune încarcă imediat la registru ("load immediate to register") sau copiază registru ("register copy") (unde registru înseamnă un registru în virgulă fixă, unul în virgulă mobilă, sau o condiţie de cod), prin ramificaţii necondiţionate, până la ultima utilizare a registrului destinaţie pentru acele instrucţiuni. Dacă reuşeşte **căutarea**, întreaga secvenţă de instrucţiuni (începând cu instrucţiunea următoare primeia şi sfârşind cu ultima instrucţiune utilă) înlocuieşte instrucţiunea primită iniţial. Toate apariţiile registrului destinaţie, pentru instrucţiunea începută în aceeaşi secvenţă, sunt înlocuite prin sursa (literal sau registru) instrucţiunii primite, şi este adăugat un salt necondiţionat la instrucţiunea următoare ultimei utilizări. Orice cod inaccesibil lăsat de transformare poate fi şters prin tehnici cum ar fi *eliminarea codului inaccesibil* ("unreachable code elimination").

Tehnica expandării blocurilor de bază ("basic block expansion")

Blocurile de bază pot fi create de optimizări de programe sau chiar programul original; ele sfârşesc cu un salt necondiţionat. Aceste ramificaţii nu sunt luate în considerare de către procesoarele VLIW, dar ele pot încetini procesoarele superscalare pentru că ele consumă resurse importante (de exemplu, dacă un salt condiţionat nexecutat este urmat imediat de un salt necondiţionat). Salturile condiţionate nu pot fi evitate, dar tehnicile de expansiune a blocurilor de bază încearcă să minimizeze apariţia salturilor necondiţionate la execuţie, copiind codul de la destinaţia ramificaţiei. Pentru a se evita blocarea benzii de asamblare, această tehnică determină numărul de instrucţiuni de neramificaţie care pot fi deplasate înaintea saltului necondiţionat, examinând codul precedent al ramificaţiei. Căutarea se opreşte când s-au înlocuit suficiente instrucţiuni de neramificaţie, când o instrucţiune (dintr-o buclă) este revizitată, sau când se execută o revenire dintr-o procedură. Există o limită a ferestrei de lucru şi depăşirea ei opreşte căutarea; în acest caz, punctul de oprire este cel care determină blocarea minimă. Odată oprită căutarea, codul este copiat, începând de la destinaţia ramificaţiei necondiţionate până la punctul de oprire. Ramificaţia necondiţionată originală este ştearsă şi este inserat un nou salt necondiţionat după codul copiat, pointând către instrucţiunea imediat următoare punctului de oprire.

Tehnica de planificare cu reacţie inversă ("feedback") – PDF ("Profiling Directed Feedback")

Informaţia de "profile" este utilizată pentru a minimiza numărul de execuţii ale blocurilor de bază. Din setul de blocuri de bază se vor număra execuţiile unui subset care acoperă toate blocurile (scade astfel "overhead"-ul). Pentru a genera şi utiliza codul de "profile", se compilează un program de două ori şi se rulează fiecare versiune separat. În timpul primei treceri a PDF, compilatorul inserează un cod de numărare la timpul de execuţie dintr-un subset de blocuri de bază. Când programul astfel compilat este executat, se creează un fişier care indică de câte ori au fost executate blocurile de bază conţinând codul de numărare. Numărarea rulărilor multiple ale aceluiaşi program poate fi acumulată. În timpul celei de-a doua treceri a PDF, compilatorul citeşte înapoi ("feedback") din fişierul creat anterior, din acelaşi loc în care compilatorul a inserat la prima trecere codul de numărare, calculează setul complet de numărări de blocuri de bază din setul de blocuri de bază de numărare citit din fişier şi-l utilizează pentru optimizări.

Nu toate blocurile de bază au nevoie însă de cod de numărare. Este posibil să se micşoreze volumul de instrucţiuni numărate într-o buclă (reducând astfel şi "overhead"-ul PDF) prin deplasarea încărcărilor şi memorărilor invariante în afara buclei.

5. Compatibilitatea arhitecturală VLIW

În cazul arhitecturilor VLIW pot apărea probleme de compatibilitate a programelor executate pe maşini de generaţii diferite. Problema compatibilităţii arhitecturilor VLIW poate fi rezolvată utilizând diferite scheme "hardware" sau "software" pentru păstrarea compatibilităţii codului compilat. Astfel de scheme utilizează tehnici "hardware", cum ar fi *divizarea execuţiei* ("split-issue"), *unitatea de umplere* ("fill unit"), sau tehnici "software" de recompilare a codului obiect al unui program VLIW dintr-o generaţie în alta. Înainte de a se prezenta câteva dintre aceste tehnici, este realizată o scurtă caracterizare a maşinilor VLIW din punctul de vedere al problemelor de compatibilitate. Astfel, arhitecturile VLIW sunt maşini orizontale cu cuvânt lung de instrucţiune (format dintr-o multioperaţie) constând în câteva operaţii executate simultan, în acelaşi ciclu de ceas.

Programele VLIW sunt "latency-cognizant", adică se cunoaşte latenţa (întârzierea) tuturor unităţilor funcţionale. Arhitecturile VLIW se mai numesc arhitecturi cu latenţă cunoscută UAL ("Unit Assumed Latency"), spre deosebire de arhitecturile cu latenţă necunoscută NUAL ("Non-Unit Assumed Latency"), care atribuie întârzieri pentru toate unităţile funcţionale; majotitatea arhitecturilor superscalare sunt UAL.

Există două modele de planificare pentru programele cu latenţă cunoscută:

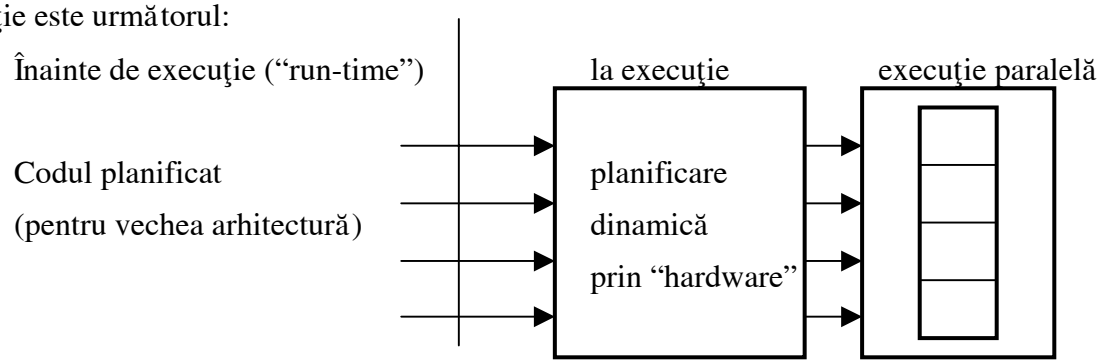
a) modelul "Equals" (E – egal): oricare operaţie are exact latenţa de execuţie specificată pentru ea;

b) modelul "Less-Than-or-Equals" (LTE – mai mic sau egal): latenţa poate fi mai mică sau egală cu cea specificată.

Modelul E produce o planificare puţin mai scurtă decât LTE în principal datorită reutilizării registrelor. Totuşi, LTE simplifică implementarea întreruperilor şi furnizează compatibilităţi liniare atunci când latenţele sunt reduse.

Tehnici "hardware"

Aceste tehnici folosesc o planificare dinamică "hardware" a codului. Un model de execuţie este următorul:



a) Metoda "split-issue"

Tehnica de planificare dinamică "hardware" prin divizarea execuţiei ("split-issue"), elaborată de B. Rau, se caracterizează prin următoarele:

Fiecare operaţie este divizată într-o pereche de operaţii de tipul "read_and_execute" (RE – citeşte şi execută), "destination_writeback" (DW – scrie destinaţie). RE utilizează ca destinaţie un registru anonim iar DW scrie la destinaţia specificată în operaţia originală. RE este executată în următorul ciclu de ceas disponibil, în care nu sunt dependenţe sau constrângeri de resurse, pe când DW este executată în ultimul ciclu de ceas aflat după întârzierea dată de (ciclul de ieşire + originea – operaţia – latenţa – 1). Pentru a se asigura că DW nu se execută mai devreme sunt prevăzuţi indicatori.

b) Metoda "fill-unit"

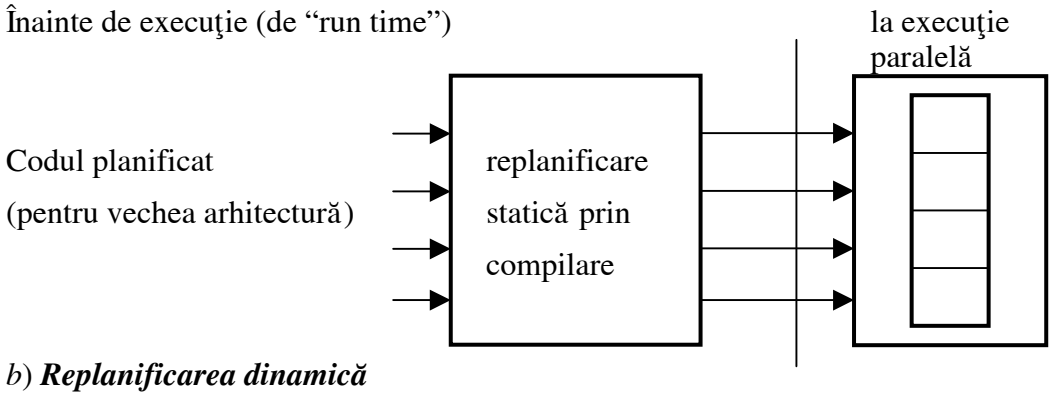
Această metodă a fost descrisă anterior, de aceea se va prezenta doar pe scurt modul de lucru:

- Procesorul execută un flux de operaţii UAL sau NUAL.
- Concurent cu execuţia, unităţile de umplere compactează operaţiile într-o multioperaţie, care se memorează într-o linie de memorie intermediară ascunsă ("shadow instruction cache").
- Formarea unei noi multioperaţii de către unitatea de umplere este terminată când se înregistrează o instrucţiune de ramificaţie.

Tehnici "software"

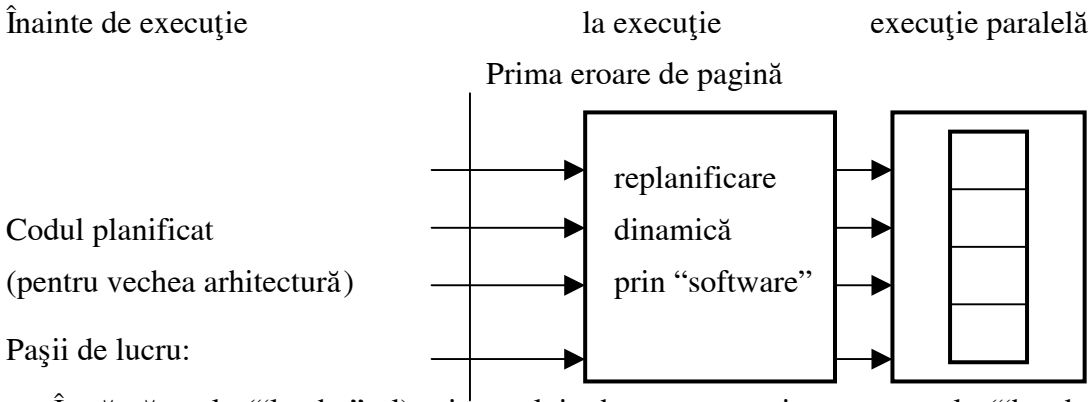
a) ***Recompilarea statică***

Această tehnică recompilează întregul program în mod static ("off-line") şi poate lua avantajele compilatoarelor optimizate. Recompilarea completă poate fi evitată prin menţinerea copiilor multiple de programe pentru diferitele arhitecturi destinaţie într-un fişier obiect partiţionat; un modul adecvat acestei cerinţe poate fi planificat la instalare. Această metodă introduce devierea de la procesul normal pentru dezvoltare şi de la procesul rutinei de instalare pentru utilizator, deoarece copiile multiple pot conduce la inconsistenţe de date şi pentru că spaţiul de memorare cerut de copii poate deveni excesiv de mare. Modelul de execuţie este următorul:



b) ***Replanificarea dinamică***

Tehnica de replanificare dinamică recurge la o versiune limitată de planificare prin program ("software scheduling") fără a necesita resurse "hardware" suplimentare. Modelul de execuţie este următorul:



- Încărcătorul ("loader"-ul) sistemului de operare citeşte antetul ("header"-ul) programului binar şi detectează versiunea generaţiei arhitecturii.
- După ce prima pagină a programului este încărcată pentru execuţie, agentul de gestiune a erorii de pagină invocă modulul de replanificare dinamică, care replanifică execuţia pe maşina curentă; procesul este repetat pentru fiecare nouă eroare de pagină.
- Paginile translatate sunt salvate pe un spaţiu de înlocuire ("swap"); sunt replanificate numai paginile care sunt executate în timpul duratei de viaţă a programului.

6. Modele arhitecturale VLIW

În acest capitol sunt prezentate diferite modele arhitecturale VLIW, implementate în cadrul unor maşini de uniprocesare paralelă performantă. În primul rând este descris un model VLIW pentru un procesor de semnal. Principiul VLIW este foarte potrivit pentru prelucrări multimedia, acestea presupunând un număr mare de operaţii necesare a se efectua în paralel. Ca realizare practică sunt cunoscute procesoarele TriMedia, realizat de firma Philips şi TMS320C62x, realizat de firma Texas Instruments. Capitolul se încheie cu prezentarea procesoarelor VLIW actuale, IA-64 sau Itanium de la firma Intel şi Crusoe (compatibil cu un procesor Mobile Intel Pentium) de la firma Transmeta.

Modelul VLIW pentru un procesor de semnal

Un procesor de semnale video poate fi proiectat conţinând mai multe elemente de procesare (PE- "Processing Elements") lucrând în paralel. Semnalul de prelucrat este digitalizat şi introdus la procesare prin intermediul unui comutator de tip "crossbar" (cu bare încrucişate). Elementele de procesare execută operaţiile în paralel pe baza unor grafuri de flux de semnal SFG ("Signal Flow Graph").

Arhitectura unui astfel de procesor, numit VSP ("Video Signal Processor"), este prezentată în figura următoare:

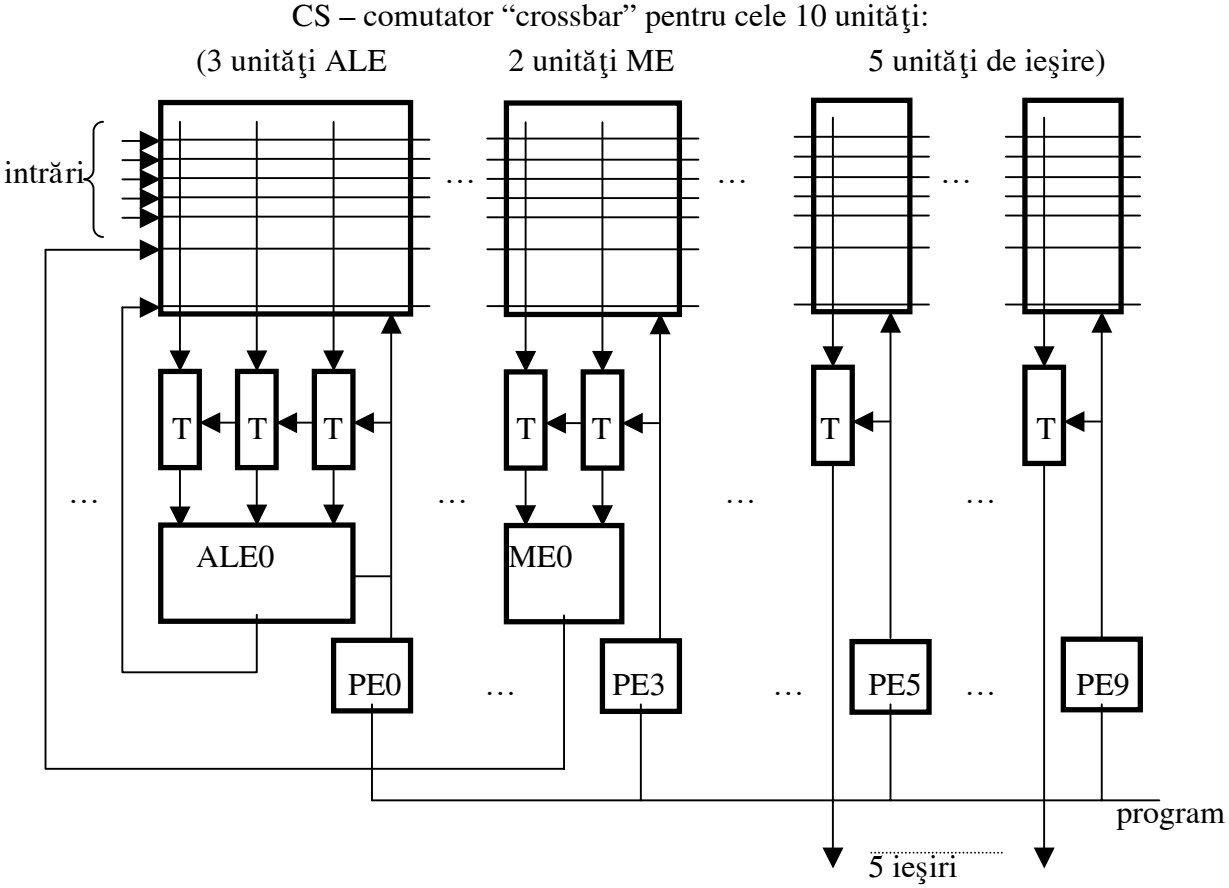


Figura 6.1.

T – tampoane; CS – comutator cu bare încrucişate ("Crossbar Switch")

Cele 10 PE corespund următoarelor unităţi:

- 3 elemente logice aritmetice (ALE – "Arithmetic Logic Element) în bandă de asamblare;
- 2 elemente de memorare (ME – "Memory Element);
- 5 elemente de ieşire.

Fiecare dintre ALE, ME şi elementele de ieşire are programul său individual de lungime până la 16 instrucţiuni. Un PE execută programul periodic iar frecvenţa cea mai scăzută este:

$$\frac{27 \text{ MHz}}{16} = 1{,}6875 \text{ MHz}$$

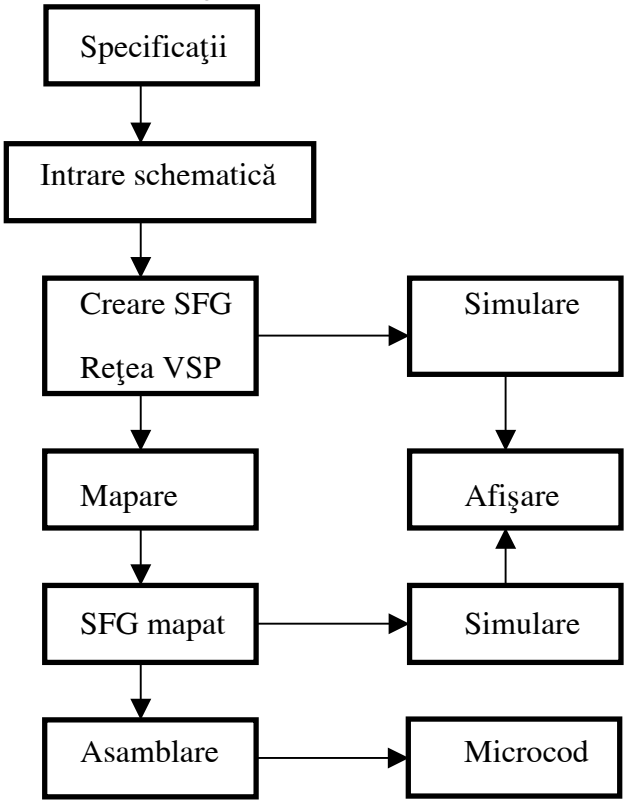
Instrucţiunile pentru ALE au 56 biţi, cele pentru ME au 36 biţi iar cele pentru elementele de ieşire au 12 biţi. O instrucţiune VLIW va avea: 3 x 56 + 2 x 36 + 5 x 12 = 300 biţi

Programele PE sunt încărcate serial printr-o interfaţă. Sistemele pot avea 1, 3, 5, 8 sau 16 procesoare de semnale video (VSP).

Grafurile de flux de semnal (SFG) descriu algoritmii programelor de prelucrare a semnalelor. Paşii de mapare a SFG pe reţeaua VSP sunt:

- asignare PE: atribuie fiecărei operaţii elementul PE pe care se execută
- asignare timp: atribuie fiecărei operaţii timpul de start al primei execuţii (în cicli de ceas)
- asignare cale: atribuie fiecărei precedenţe etichetate de date calea de urmat prin VSP

Traiectoria programelor urmăreşte organigrama:



Exemple de implementări actuale de procesoare de semnal, având nucleul de procesare de tip VLIW:

Procesorul Texas Instruments TMS320C62x

Acest procesor conţine 8 unităţi de execuţie independente împărţite pe două căi de date, fiecare având aceleaşi patru tipuri de unităţi funcţionale: L (adunare întreagă, operaţii logice, numărare pe biţi, adunare în virgulă mobilă, conversie din virgulă mobilă); S (adunare întreagă, operaţii logice, operaţii pe biţi, deplasări, constante, ramificaţii/control, comparări în virgulă mobilă, conversii din virgulă mobilă, împărţiri în virgulă mobilă); D (adunare întreagă, încărcare – memorare); M (înnmulţire întreagă şi în virgulă mobilă).

Frecvenţa de ceas a procesorului este de 200 MHz, suficientă pentru procesări de semnale.

Fiecare cale de date are un fişier de 16 registre pe 32 biţi cu 10 porturi de citire şi 6 porturi de scriere; există posibilitatea de a transfera valori între cele două fişiere de registre de pe cele două căi de date, prin intermediul unei magistrale interne globale.

Schema de execuţie este următoarea:

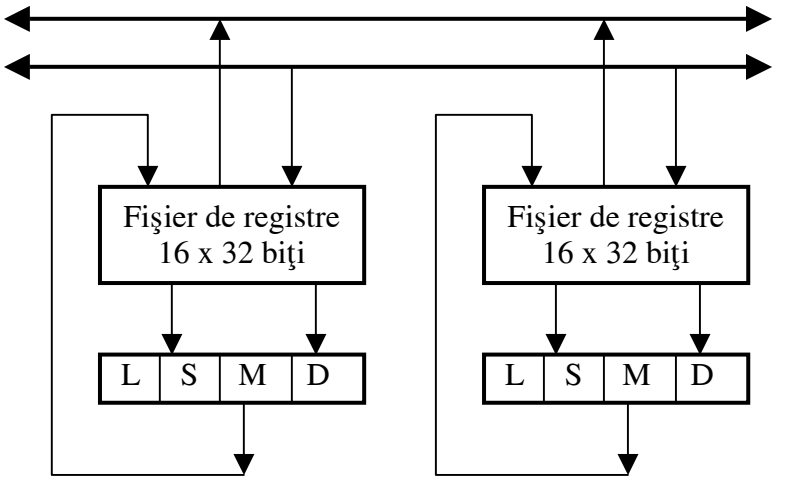


Fig. 6.2.

Calea internă de execuţie are 256 biţi şi se pot executa 8 operaţii pe fiecare perioadă de ceas, cu operanzi pe 32 biţi.

Din memoria internă de instrucţiuni se citesc pachete conţinând până la 8 instrucţiuni, care pot fi citite simultan. Fiecare pachet citit se poate expanda în până la 8 pachete de execuţie, funcţie de biţii de paralelizare asociaţi. Un pachet de execuţie are 8 instrucţiuni care se pot executa în paralel, cum s-a specificat mai sus.

Schematic, un pachet citit se poate extinde în pachete de execuţie ca în exemplul următor:

Fie următorul pachetul citit:

A	B	C	D	E	F	G	H
1	1	1	0	0	1	1	0

Se obţine o secvenţă de 3 pachete de execuţie A || B || C || D, E, F || G || H, în care s-a ţinut cont de faptul că un şir de biţi 1 urmat de un 0 conduce la o execuţie paralelă, în timp ce un şir de biţi începând cu un 0 indică o execuţie secvenţială a instrucţiunilor.

Cele 3 pachete de execuţie sunt:

x	A	B	x	x	C	x	D
E	x	x	x	x	x	x	x
x	F	x	G	x	x	H	x

rezultă 24 de instrucţiuni

Execuţia instrucţiunilor se realizează în trei faze, fiecare în bandă de asamblare, conform următoarelor trei etape:

- Citire: are patru etaje (PG – generarea adresi program; PS – trimiterea adresei program; PW – accesarea programului din memoria intermediară; PR – primirea pachetului de program citit);
- Decodificare: are două etaje (DP – despachetarea instrucţiunilor, conversia pachetului citit în pachete de execuţie şi rutarea lor către unităţile funcţionale; DC – decodificarea instrucţiunilor);
- Execuţie: are maxim 10 etaje; 90% din instrucţiuni utilizează doar primele cinci etaje, doar operaţiile de adunare/înmulţirre în virgulă mobilă în dublă precizie utilizează şi celelalte cinci etaje.

Procesorul Phillips Trimedia TM 1000

TM 1000 este un procesor de semnale multimedia, având nucleul de procesare CPU în arhitectură VLIW. Procesorul posedă cinci unităţi multifuncţionale de execuţie, având un total de 27 de funcţii şi poate executa 5 operaţii pe perioada de ceas; are de asemenea un fişier de 128 de registre cu 15 porturi de citire (deoarece o operaţie necesită doi operanzi şi o informaţie pentru anticiparea ramificaţiilor) şi 5 porturi de scriere.

Se pot citi 64 de octeţi simultan din memoria intermediară de instrucţiuni, împachetaţi într-o instrucţiune VLIW.

Execuţia se realizează pe cinci etaje: citire instrucţiuni (FI), decompresie (DC), citire registre (RR), execuţie (EX) şi scriere rezultat (WB).

Schema microarhitecturală internă a procesorului este următoarea:

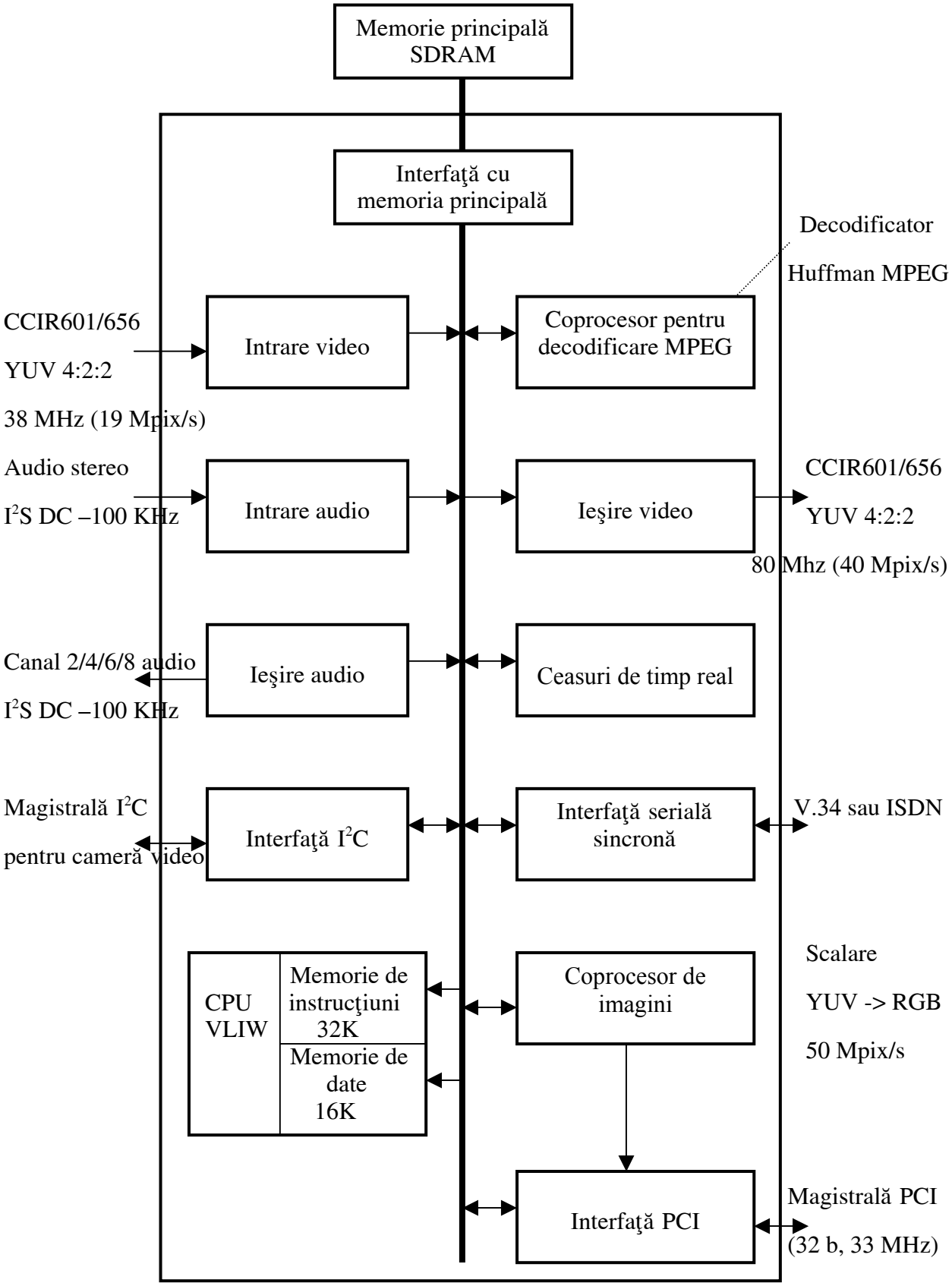

Memorie principală
SDRAM
Interfaţă cu
memoria principală
Decodificator
Huffman MPEG
CCIR601/656
YUV 4:2:2
38 MHz (19 Mpix/s)
Intrare video
Coprocesor pentru
decodificare MPEG
Audio stereo
I2S DC –100 KHz
Intrare audio
Ieşire video
CCIR601/656
YUV 4:2:2
80 Mhz (40 Mpix/s)
Canal 2/4/6/8 audio
I2S DC –100 KHz
Ieşire audio
Ceasuri de timp real
Magistrală I2C
pentru cameră video
Interfaţă I2C
Interfaţă serială
sincronă
V.34 sau ISDN
CPU
VLIW
Memorie de
instrucţiuni
32K
Memorie de
date
16K
Coprocesor de
imagini
Scalare
YUV -> RGB
50 Mpix/s
Interfaţă PCI
Magistrală PCI
(32 b, 33 MHz)

Fig. 6.3.

Implementări actuale de procesoare VLIW

Procesorul Transmeta Crusoe

În mod tradiţional, procesoarele VLIW au fost proiectate cu scopul de a maximiza performanţa prin execuţia în paralel a cât mai multor instrucţiuni posibile. Firma Transmeta şi-a propus, şi a reuşit, sa realizeze un procesor cu performanţe moderate, comparativ cu procesoarele curente, dar cu un consum foarte mic de tensiune şi fiind astfel potrivit pentru emularea procesoarelor din calculatoarele mobile, din familia Mobile Intel Pentium, sau a maşinilor virtuale Java.

Procesorul Crusoe are instrucţiuni pe 128 biţi, numite molecule, fiecare moleculă codificând 4 operaţii, numite atomi. Formatul moleculei determină direct cum sunt rutate operaţiile la cele 5 unităţi funcţionale (2 unităţi de prelucrare întregi, o unitate de prelucrare în virgulă mobilă, o unitate de încărcare/memorare şi o unitate de procesare a ramificaţiilor).

Emularea codului pentru execuţie se realizează cu ajutorul unui translator binar ce rulează la timpul de execuţie. Astfel, numai acest ”software” trebuie modificat atunci când se schimbă arhitectura procesorului. În urma execuţiei codului emulat se colectează informaţii care conduc la auto-optimizări ale translatorului.

Figura următoare prezintă schema bloc a procesorului Crusoe:

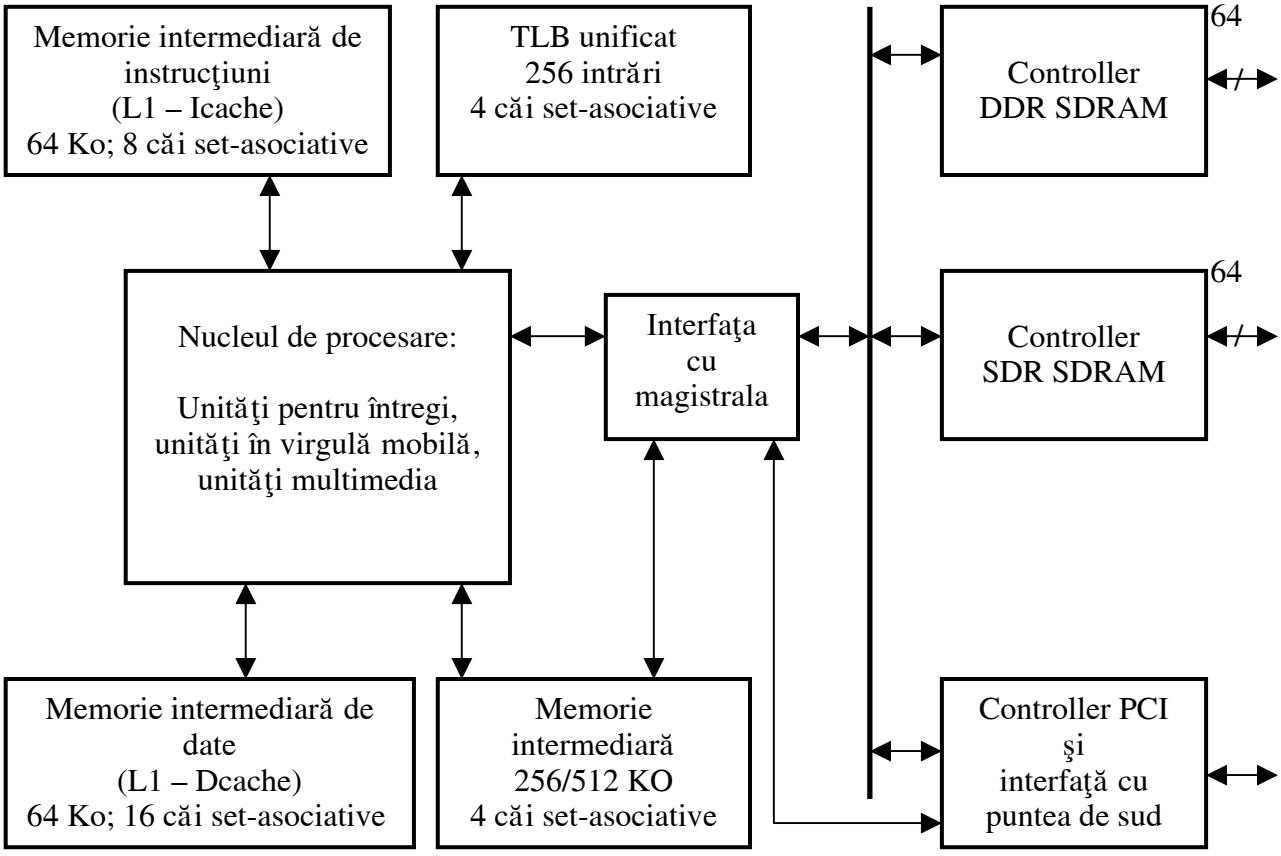


Fig. 6.4.

Procesorul Intel Itanium IA-64

Acest procesor este prima implementare pe 64 biţî pentru a procesa instrucţiuni de tip VLIW, dezvoltat împreună de firmele Intel şi Hewlett Packard (HP).

Procesorul conţine 4 unităţi de prelucrare întregi, 4 unităţi multimedia, 2 unităţi de încărcare/memorare, 2 unităţi de prelucrare în virgulă mobilă cu dublă precizie şi 2 unităţi de prelucrare în virgulă mobilă cu precizie simplă.

Acesta lucrează la o frecvenţă de 800 de MHz, este realizat într-o tehnologie de 0,18 microni şi are banda de asamblare cu 10 etaje.

Arhitectura IA-64 utilizează un format fix de instrucţiuni împachetate pe 128 biţi. Fiecare instrucţiune VLIW, numită multioperaţie, constă într-unul sau mai multe pachete de 128 biţi, fiecare pachet având 3 operaţii şi un câmp de control. IA-64 utilizează un tampon de decuplare şi suport speculativ pentru a creşte viteza de execuţie.

Itanium posedă un set de 128 de registre generale şi un alt set de 128 de registre în virgulă mobilă pe 82 biţi.

Arhitectura IA-64 implementează suport pentru algoritmul de planificare pe bucle (“software pipelining”), dezvoltat de Bob Rau de la HP în cadrul unui proiect anterior (Cydra-5); pentru aceasta s-a implementat posibilitatea de rotaţie a registrelor astfel încât nu mai este necesară desfacerea buclei şi redenumirea registrelor utilizate în iteraţii succesive.

IA-64 suportă execuţie şi control speculativ utilizând un mecanism ”software/hardware” propriu. Ea suportă de asemenea planificarea statică şi dinamică a ramificaţiilor.

Setul de instrucţiuni conţine şi instrucţiuni SIMD potrivite pentru procesare multimedia.

Procesorul Itanium este probabil cea mai complexă arhitectură VLIW implementată comercial, ceea ce justifică preocupările multor alte firme de a include în proiectele lor modelul arhitectural VLIW.

Schema bloc a procesorului Intel Itanium este următoarea:

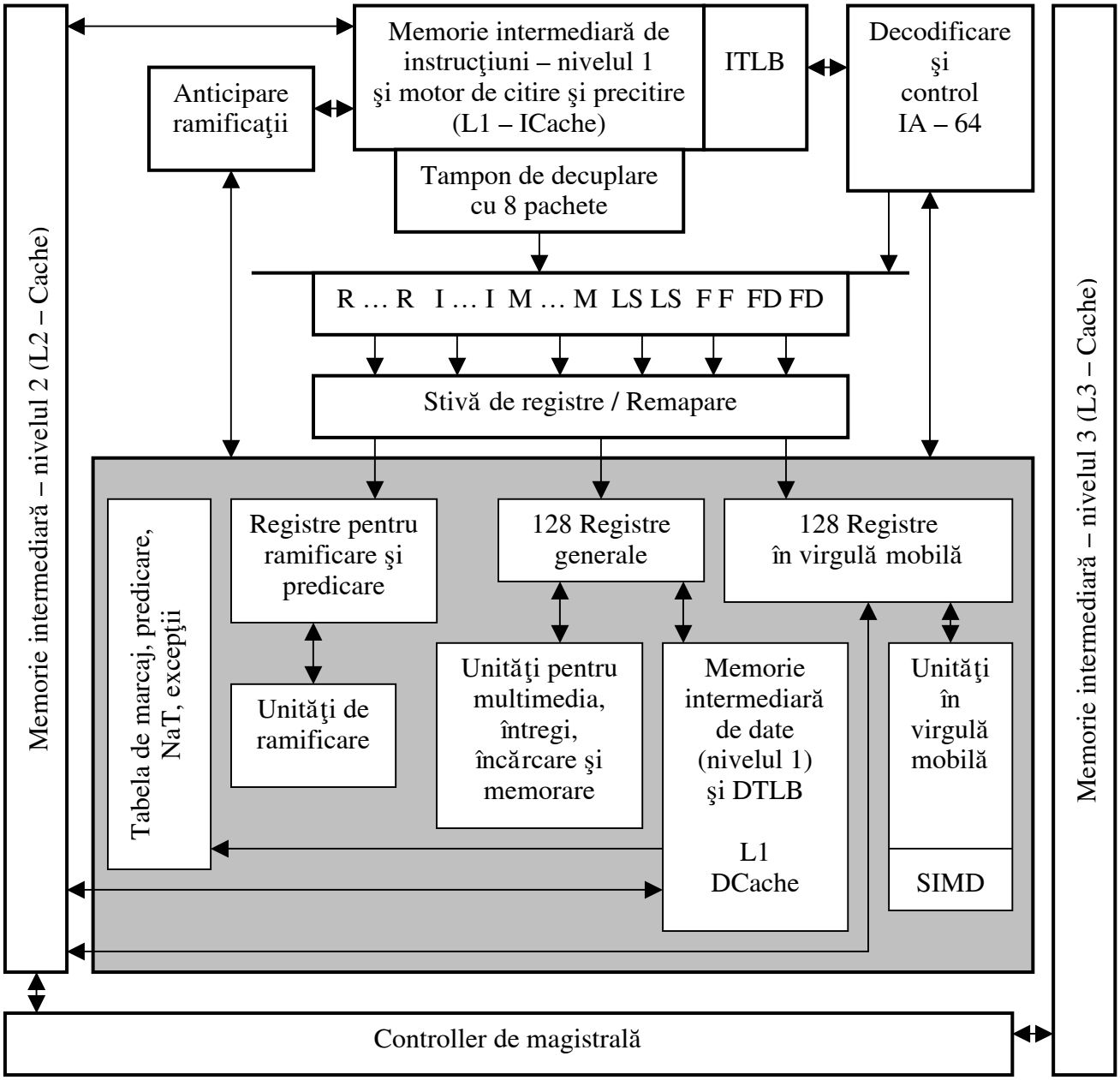
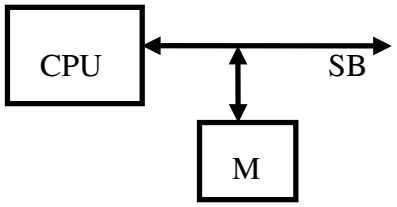

Memorie intermediară – nivelul 2 (L2 – Cache)
Memorie intermediară de instrucţiuni – nivelul 1 şi motor de citire şi precitire (L1 – ICache)
ITLB
Decodificare şi control IA – 64
Anticipare ramificaţii
Tampon de decuplare cu 8 pachete
R … R I … I M … M LS LS F F FD FD
Stivă de registre / Remapare
Tabela de marcaj, predicare, NaT, excepţii
Registre pentru ramificare şi predicare
Unităţi de ramificare
128 Registre generale
Unităţi pentru multimedia, întregi, încărcare şi memorare
128 Registre în virgulă mobilă
Memorie intermediară de date (nivelul 1) şi DTLB
L1 DCache
Unităţi în virgulă mobilă
SIMD
Memorie intermediară – nivelul 3 (L3 – Cache)
Controller de magistrală

Fig. 6.5.

7. Creşterea performanţei procesoarelor superscalare utilizând memorii intermediare

Utilizarea memoriilor intermediare (numite şi memorii tampon sau "cache") înăuntrul şi în afara procesorului a condus la o creştere semnificativă a vitezei de execuţie, atât în cazul procesoarelor CISC, cât şi în cazul celor RISC, deci şi pentru procesoarele VLIW.

Se consideră un sistem tipic constând într-un procesor (CPU – "Central Processing Unit"), o magistrală sistem (SB – "System Bus") şi o ierarhie de memorii M:



Se poate calcula rata de execuţie a instrucţiunii (IR – "Instruction Rate") în MIPS utilizând lăţimea de bandă B ("Bus Bandwidth"). Lăţimea de bandă în Mocteţi/sec este definită prin volumul de lucru transferat la şi de la memorie în unitatea de timp.

Se poate calcula B prin relaţia:

$$B = u_b \times \frac{n_b}{n_c \times t_c} \qquad (1)$$

unde: u_b este utilizarea magistralei, n_b este numărul de octeţi transferat pe un acces la memorie, n_c este numărul de cicli de ceas per transfer cu memoria şi t_c este durata ciclului de ceas.

Se consideră că n_c este format din numărul de cicli de ceas pentru magistrala sistem şi numărul de cicli de ceas pentru acces la memoria sistem. De asemenea, se consideră că un grad de utilizare a magistralei de minim 75% este întâlnit în majoritatea sistemelor de calcul.

Acum, IR poate fi calculată prin relaţia:

$$IR = \frac{B}{n_t} = \frac{B}{n_i + n_d} \qquad (2)$$

unde: n_t este numărul total mediu de octeţi per instrucţiune executată, n_i este numărul de octeţi instrucţiune şi n_d este numărul de octeţi de date.

Există patru căi diferite de a utiliza o memorie intermediară într-un sistem de calcul:

<u>Sistem cu memorie intermediară de instrucţiuni/date în afara procesorului</u>

Când apare un "cache miss" (adică o eroare de pagina lipsă în memoria intermediară), atunci se vor adăuga un număr suplimentar de cicli de ceas (n_{ca}) necesari pentru a accesa memoria sistem. Deci, se poate calcula noul n_c* prin relaţia:

$$n_c^* = n_c + n_{ca} \times miss\ ratio\ extern \qquad (3)$$

Sistem cu memorie intermediară de instrucţiuni în interiorul procesorului

Când apare o eroare de pagină, se modifică atât n_c, cât şi n_i.

Se pot calcula astfel:

$$n_c^* = n_{c1} + n_c \times miss\ ratio \qquad (4)$$

unde: n_{c1} este n_c pentru a accesa memoria intermediară internă

$$n_i^* = miss\ ratio\ intern \times \frac{n_i \times \text{numărul de octeţi de instrucţiune pe miss}}{n_b} \qquad (5)$$

Sistem cu memorii intermediare separate pentru instrucţiuni şi pentru date în interiorul procesorului

Când apare o eroare de pagină, se modifică atât n_c, cât şi n_i, dar şi n_d.

n_c şi n_i se calculează ca mai sus, iar n_d astfel:

$$n_d^* = miss\ ratio \times \frac{n_d \times \text{numărul de octeţi de date pe miss}}{n_b} \qquad (6)$$

Sistem cu memorii intermediare interne separate pentru instrucţiuni şi pentru date în conjuncţie cu o memorie intermediară externă rapidă pentru instrucţiuni/date

Î n acest caz :

$$n_c^* = n_{c1i} + (n_{c2} + n_{ca} \times miss\ ratio\ extern) \times miss\ ratio\ intern \qquad (7)$$

unde: n_{c1i} este n_c pentru memoria intermediară de instrucţiuni internă şi

n_{c2} este n_c pentru accesul la memoria intermediară

n_i^* şi n_d^* se calculează ca mai sus.

Evaluarea performanţei unui procesor VLIW

Se vor compara performanţele obţinute la un procesor superscalar performant (fie acesta procesorul PowerPC 620) şi la un procesor VLIW.

Parametrii comuni cunoscuţi sunt:

Frecvenţa ceasului: 500 MHz; gradul de utilizare a magistralelor: 75% (rezonabil de altfel).

Memoria intermediară de date şi instrucţiuni externă = 128 Kocteţi; miss ratio = 2.5%; linii de 128 octeţi.

Memoria intermediară de date internă = 64 Kcuvinte; miss ratio = 2.5%; linii de 32 octeţi.

Memoria intermediară de instrucţiuni internă = 64 Kcuvinte; miss ratio = 2%; linii de 64 octeţi.

Magistrala externă de date este pe 128 biţi iar magistrala internă de date este pe 64 biţi.

n_b = 128/8 =16 octeţi; n_{c1i} = 8 cicli de ceas pentru un acces la memoria intermediară de instrucţiuni internă; n_{c2} = 10 cicli de ceas pentru un acces la memoria intermediară externă; n_{ca} = 5 cicli de ceas pentru un acces la memoria sistem.

t_c = 1/500 MHz = 0.2×10^{-8} secunde;

Se aplică formulele (7), (1), (5), (6) şi (2) din paragraful anterior.

Se obţine:

$n_c^* = 8 + (10 + 5 \times 0.025) \times 0.02 \cong 8.203$ cicli de ceas.

$B = 0.75 \times [16/(8.203 \times 0.2 \times 10^{-8})] = 731.44$ Mocteţi/secundă.

Pentru procesorul RISC, având magistrala internă de instrucţiuni pe 256 biţi, rezultă:

$n_i = 256/(8 \times 4$ instrucţiuni$) = 8$ octeţi; $n_d = 64/8 = 8$ octeţi

$n_i^* = 0.02 \times [(8 \times 32)/16] = 0.32$ octeţi $n_d^* = 0.025 \times [(8 \times 32)/16] = 0.4$ octeţi

Se obţine o frecvenţă de execuţie pentru procesorul RISC:

$I_{RPowerPC} = 731.44/(0.32 + 0.4) \cong 1015.9$ MIPS.

Se va considera acum un procesor VLIW cu aceiaşi parametri de lucru ca şi procesorul PowerPC, mai puţin magistrala internă de instrucţiuni, care este pe 432 biţi.

Se obţine:

$n_i = 432/(8 \times 13$ instrucţiuni$) = 4.154$ octeţi; $n_d = 64/(8 \times 12$ operaţii$) = 0.667$ octeţi.

Rezultă:

$n_i^* = 0.02 \times [(4.154 \times 32)/16] = 0.166$ octeţi

$n_d^* = 0.025 \times [(0.667 \times 32)/16] = 0.034$ octeţi

Se obţine o frecvenţă de execuţie pentru procesorul VLIW:

$I_{RVLIW} = 731.44/(0.166 + 0.034) \cong 3657.2$ MIPS.

Cum $I_{RVLIW} / I_{RpowerPC} \cong 3.6$, rezultă o creştere teoretică de 3.6 ori a ratei de execuţie a instrucţiunilor pe un procesor VLIW faţă de execuţia pe un procesor superscalar RISC.

Prin eliminarea dependenţelor dintre instrucţiuni încă din faza de compilare a programului, înainte de execuţia lui pe procesor, se obţine pentru procesorul VLIW o creştere suplimentară a performanţei de aproximativ 25%.

Se obţine $I^*_{RVLIW} = 1.25 \times I_{RVLIW} = 4571.5$ MIPS.

Rezultă o creştere de performanţă de $I^*_{RVLIW} / I_{RpowerPC} \cong 4.5$ ori faţă de cea a procesorului RISC.

ÎNTREBĂRI

1. Să se prezinte tipurile de paralelism implementate în sistemele de calcul numeric.
2. Pe baza unei scheme bloc de funcționare, să se prezinte caracteristicile şi modul de operare ale unui procesor VLIW.
3. Să se definească o arkitectură VLIW pentru procesorul DLX (MIPS) prezentat la curs.
4. Să se prezinte o modalitate “hardware” de creştere a performanței unui procesor VLIW.
5. Să se prezinte o modalitate “software” de creştere a performanței unui procesor VLIW.
6. Cum se poate păstra compatibilitatea între diferite generații de arhitecturi VLIW?
7. Evaluați performanța procesorului DLX (MIPS sau VLIW) cu şi fără utilizarea de memorii intermediare interne separate pentru date şi instrucțiuni şi memorie intermediară externă pentru date/instrucțiuni; se va calcula frecvența de execuție a instrucțiunilor (în MIPS), utilizând unele din formulele prezentate în curs.