Despre “ţevi”
Arhitectura Modernă a Procesoarelor (2)

Mihai Budiu — `mihaib+@cs.cmu.edu`
`http://www.cs.cmu.edu/~mihaib/`

11 noiembrie 1998

Subiect: pipelining

Cunoştinţe necesare: cunoştinţe elementare despre arhitectura calculatoarelor, programare în limbaj de asamblare

Cuvinte cheie: pipeline, stall, forward, hazard

Cuprins

Procesoarele RISC au apărut, în prima jumătate a anilor ’80; era începutul unei revoluţii în arhitectura calculatoarelor. Ideea era că în loc de a oferi o sumedenie de operaţiuni exotice, este mai eficace pentru un procesor să ofere un set restrîns de operaţiuni, pe care le poate executa cu foarte mare viteză. Primele procesoare RISC conţineau între 25 de mii şi 40 de mii de tranzistoare.

În ziua de astăzi Pentium II conţine 15 milioane de tranzistoare! Văzut dinafară Pentium II nu este un RISC, dar arhitectura internă este de acest tip.

Întrebarea este: ce s-a întîmplat cu toate tranzistoarele astea, unde au fost "înghiţite"? Setul de instrucţiuni al unui procesor modern este practic identic cu al unui procesor de acum 15 ani (exceptînd ornamente de genul MMX), deci funcţionalitatea oferită este neschimbată. Atunci la ce treabă au fost înhămaţi toţi tranzistorii? Răspunsul este: pentru a creşte performanţa.

Acest articol prezintă una dintre tehnicile cele mai simple la îndemîna unui proiectant pentru a creşte performanţa unui microprocesor; practic toate procesoarele fabricate în ziua de azi o folosesc. Este vorba de tehnica "benzii de asamblare", numită şi *pipeline*. Vom vedea că e vorba de o idee extrem de simplă şi eficace. Vom vedea apoi că proprietăţile unei benzi de asamblare ridică o grămadă de noi probleme, şi că implementarea tehnicii este extrem de complicată, cerînd un suport arhitectural substanţial. Vom mai vedea şi unele dintre metodele folosite pentru a rezolva problemele ivite.

1 Tehnologie, paralelism şi performanţă

Există două forţe motoare responsabile de creşterea spectaculoasă a puterii de calcul a microprocesoarelor. Una dintre aceste forţe este tehnologia de fabricaţie şi miniaturizarea. Miniaturizarea unui circuit integrat digital se măsoară în microni; distanţa care se indică este, grosolan vorbind, distanţa între două sîrme adiacente pe o suprafaţă a circuitului. Tehnologia curentă dominantă în acest an este undeva între 0.25 şi 0.35 microni. Pentru comparaţie, un fir de păr omenesc are cam 25 de microni, adică de 100 de ori mai mult!

Scăderea dimensiunilor înseamnă că distanţa dintre circuite scade, deci semnalul electric poate parcurge mai multe dintre ele în aceeaşi unitate de timp. De asemenea înseamnă că frecvenţa ceasului poate fi crescută. Ceasul unui microprocesor este echivalentul insului de la tobă de pe o galeră romană din antichitate: bate ritmul cu care se sincronizează toate părţile componente (vîslaşii). Frecvenţele atinse la ora actuală de procesoarele comerciale sunt de 600Mhz: 600 de milioane de operaţii pe secundă, pentru procesoarele Alpha 21264.

Atît despre tehnologie. A doua metodă de creştere a performanţei, care este oarecum o consecinţă indirectă a miniaturizării, este *paralelismul*. Dacă ai mai mulţi vîslaşi barca merge mai repede. Cu cît vîslaşii sunt mai mici, cu atît poţi pune mai mulţi în cală. (Desigur, analogia nu e perfectă, pentru că, spre deosebire de galere, pentru un microprocesor un vîslaş mai mic face la fel de multă treabă ca unul mare.)

Practic toate articolele din seria aceasta vor discuta numai despre această a doua metodă. Vom vedea că procesoarele moderne consacră o cantitate impresionantă de resurse pentru a stoarce o cît de mică îmbunătăţire a performanţei (acolo se duc cele 15 milioane de tranzistoare ale lui Pentium). În general o dublare a mărimii circuitului produce mult mai puţin decît o dublare a performanţei, dar în ziua de azi şi obţinerea cîtorva procente este o realizare meritorie.

Introducem acum cititorului prima tehnică de prelucrare paralelă, numită "tehnica benzii de asamblare".

Ora	0	1/2	1	1 1/2	2	2 1/2	3
1 maşină	intră 1		intră 2 1 terminat		intră 3 2 terminat		intră 4 3 terminat
2 maşini	1 la spălat	2 la spălat 1 la uscat	3 la spălat 2 la uscat 1 terminat	4 la spălat 3 la uscat 2 terminat	5 la spălat 4 la uscat 3 terminat	6 la spălat 5 la uscat 4 terminat	7 la spălat 6 la uscat 5 terminat

Tabelul 1: Funcţionarea celor două modele de spălătorie presupunînd că există suficient de multe rufe. Varianta cu maşini de uscat şi spălat separate produce de două ori mai multe *serii* de rufe spălate în acelaşi timp.

2 Banda de asamblare

2.1 Despre economia capitalistă

Averea considerabilă a lui Henry Ford se datora, cel puţin în parte, metodei sale inovatoare de a organiza munca la fabricile sale de automobile: lucrătorii stau aşezaţi de-a lungul unei benzi, iar maşinile neterminate merg de la unul la altul. Fiecare execută asupra maşinii o singură operaţiune, după care o pasează mai departe. Asta e banda de asamblare.

Pentru a face mai evidente beneficiile tehnicii, îmi permit să o ilustrez cu încă un exemplu. Să presupunem că vreţi să investiţi într-o spălătorie/uscătorie (Nufărul?). Ce i-aţi recomanda proprietarului: să cumpere 10 maşini de spălat care centrifughează şi usucă, sau, cu aceiaşi bani, 9 maşini care spală şi 9 care usucă? Presupunem că un spălatul şi uscatul durează la fel de mult (oricare maşină am folosi), o jumătate de oră.

Răspunsul este: cu cele 9 perechi de maşini eficienţa este cu 80% mai mare! Iată de ce: să presupunem că avem un şuvoi constant de consumatori. Atunci cu maşinile de spălat putem face 10 încărcături de rufe în fiecare oră. Cu maşinile perechi însă putem face 9 încărcături la fiecare *jumătate de oră*, aşa cum arată tabelul 1:

2.2 Cum rămîne cu calculatoarele?

Ei bine, exact aceeaşi idee poate fi aplicată în cazul construcţiei microprocesoarelor! În acest caz banda de asamblare se numeşte *pipeline*, sau conductă. De aici şi titlul articolului de faţă. Am oarecare dificultăţi în a alege o traducere rezonabilă a termenului, aşa că pe parcursul articolului voi folosi variaţi termeni, incluzînd pe cel de "ţeavă".

Cum se aplică deci conceptul în cazul microprocesoarelor? Care e treaba unui microprocesor? Să execute, una cîte una, instrucţiunile programelor scrise de utilizatori. Dar execuţia unei instrucţiuni se poate descrie ca o serie de paşi succesivi; ceva de genul: adu instrucţiunea din memorie, uită-te ce fel de instrucţiune este (o operaţie aritmetică/logică, un apel de procedură, un salt, etc.), decide care date trebuie procesate (care sunt regiştrii[1] care conţin acele date), extrage datele din regiştri, efectuează operaţia asupra datelor, pune rezultatul la loc unde trebuie, şi o ia

[1]Regiştrii microprocesorului sunt o minusculă memorie aflată în interiorul procesorului, unde el păstrează datele pe care intenţionează să le proceseze în viitorul foarte apropiat.

de la capăt cu instrucţiunea următoare. În figura 1 avem structura unui procesor ipotetic pe care indicăm cinci stagii prin care o instrucţiune trece în prelucrare.

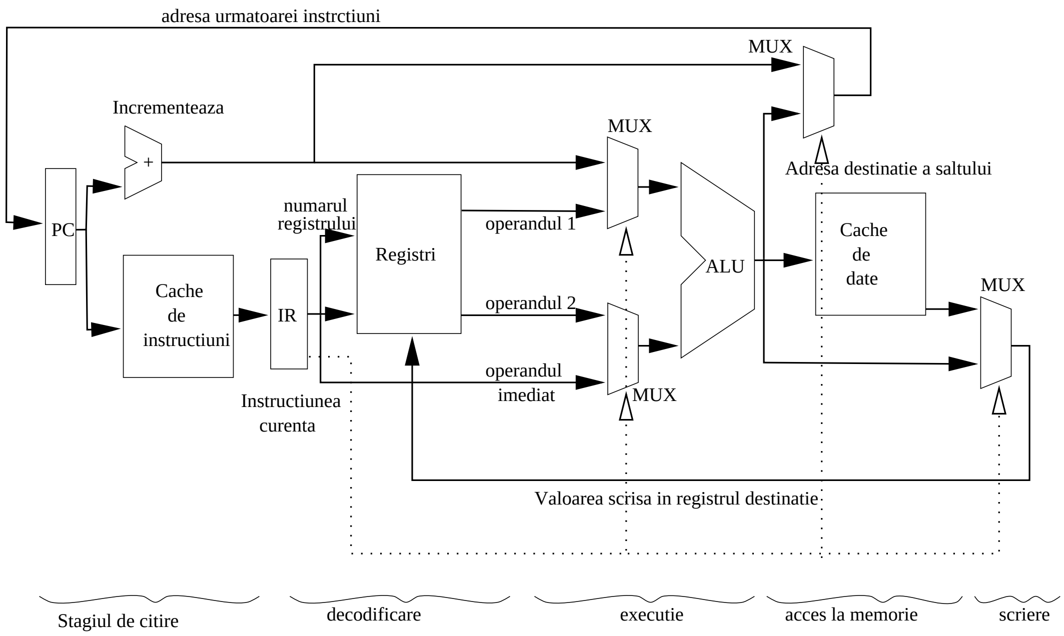

Figura 1: Arhitectura internă a unui microprocesor ipotetic. **PC** este Program Counter, adresa instrucţiunii în curs de execuţie din program. **IR** este Instruction Register, registrul în care instrucţiunea curentă este decodificată pentru a extrage comenzile asupra celorlalte unităţi ale microprocesorului. **Cache-urile**, în număr de două, sunt memoriile în care procesorul ţine datele, respectiv instrucţiunile. **MUX** sunt multiplexoare: circuite care au mai multe intrări şi o singură ieşire; ele aleg una dintre intrări şi o trimit la ieşire. Informaţia despre care dintre intrări este selectată depinde de instrucţiunea curentă, şi este indicată prin linia punctată. De exemplu, ultimul multiplexor din dreapta, va alege linia de sus pentru instrucţiuni care citesc din memorie şi pun rezultatul într-un registru, şi va alege linia de jos pentru instrucţiuni care scriu o valoare calculată. **ALU** este unitatea aritmetică-logică (Arithmetic-Logical Unit). Am etichetat unele dintre conexiuni cu informaţii despre valoarea pe care o poartă. Pentru fiecare instrucţiune, unele dintre aceste sîrme vor purta informaţii utile iar altele nu. Încercaţi să identificaţi pentru fiecare tip de instrucţiune de fapt ce se întîmplă.

Ei bine, instrucţiunile joacă exact rolul rufelor: dacă avem în procesor cîte un circuit independent pentru fiecare din aceste numeroase sarcini, atunci putem pune aceste circuite să lucreze simultan pe instrucţiuni succesive. Astfel, în timp ce instrucţiunea 1 pune rezultatul la loc (stagiul de acces la memorie), instrucţiunea 2 operează asupra propriilor date (stagiul de execuţie), instrucţiunea 3 tocmai extrage datele (stagiul de decodificare), iar instrucţiunea 4 tocmai este adusă din memorie (stagiul de citire).

Observaţi că — aparent — cîştigul pe care l-am obţine transformînd procesorul într-un pipeline, este *pe gratis*: şi un procesor obişnuit are nevoie de toate aceste circuite, însă atunci cînd folosea unul dintre ele, celelalte erau inutile. Că lucrurile nu stau chiar aşa vom vedea în secţiunile următoare.

Cartea “canonică” pentru studiul arhitecturii calculatoarelor moderne este Hennessy and Patterson “Computer Architecture — a Quantitative Approach”, publicată de editura Morgan Kaufmann în ediţia a doua în 1995. Această carte este de fapt versiunea pentru cursuri doctorale a unei alte cărţi de aceiaşi autori. Cartea este excelent scrisă, iar capitolele 3 şi 4 sunt dedicate în întregime tehnicii de pipelining, în total 250 de pagini. Dacă subiectul vă interesează cu adevărat, vă recomand să obţineţi această carte; dacă nu, poate că articolul acesta este suficient de ilustrativ. Acest articol va discuta doar despre tehnicile elementare folosite în pipelining; despre tehnicile avansate (multiple-issue, out-of-order execution, etc.) voi scrie probabil un altul. Desenele din acest text, şi diagrama procesorului ipotetic, sunt bazate pe prezentarea din această carte.

2.3 Regiştrii de separaţie

Dacă un procesor este implementat ca un pipeline, atunci între diferitele *stagii* ale ţevii se află nişte “tampoane”, care izolează stagiile unul de altul. Arhitectural vorbind, tampoanele sunt de fapt tot nişte regiştri, numiţi *pipeline registers*. Fiecare din aceşti regiştri este comandat de ceasul microprocesorului, şi încarcă în interior toate rezultatele procesării obţinute din stagiul anterior: instrucţiunea, rezultatele parţiale, informaţii de stare, etc. Figura 2 arată segmentarea procesorului de mai sus.

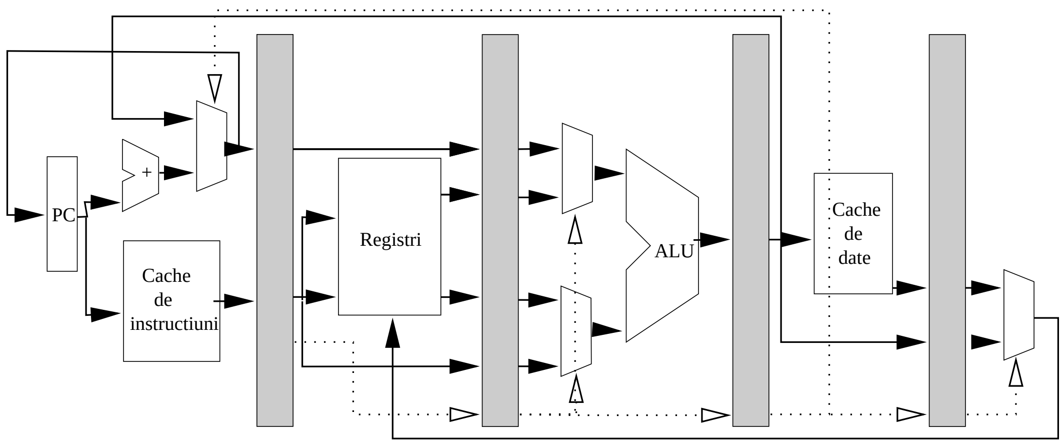

Figura 2: Procesorul de mai sus împărţit în stagii. Barele haşurate sunt regiştrii de separaţie. Observaţi că am mutat un multiplexor din stagiul de acces la memorie în stagiul de citire, pentru că altfel în cazul unui salt am fi avut două stagii care încearcă simultan să scrie în registrul PC. Cu toate acestea, linia care controlează multiplexorul vine în continuare din stagiul de acces la memorie, unde instrucţiunile de salt îşi termină de calculat destinaţia.

Vom vedea că regiştrii de separaţie au un rol important pentru blocajul ţevii în anumite circumstanţe.

2.4 Influenţa asupra ceasului

Ce cîştigăm cu ajutorul ţevilor?

Păi, în primul rînd, fiecare din stagii este mai scurt decît întregul, deci se poate executa mai repede decît dacă am executa toate stagiile unul după altul. Putem deci mări frecvenţa ceasului;

este exact acelaşi fenomen ca la maşinile de spălat de mai sus (încărcături la jumătate de oră în loc de o oră).

În al doilea rînd, aşa cum am văzut, acum toate stagiile sunt folosite simultan, fiecare pentru altă instrucţiune. Aceasta este o sursă de paralelism, care implică o altă creştere a performanţei.

Observaţi că succesul acestei metode se bazează pe faptul că avem de procesat un şir de instrucţiuni extrem de lung, continuu. Dacă la spălătorie vin rufe din două în două ore, atunci nu cîştigăm nimic din faptul că putem scoate o nouă serie la fiecare jumătate de oră.

Observaţi că durează o vreme de cînd prima instrucţiune intră în ţeavă pînă cînd termină execuţia: atîţia cicli de ceas cîte stagii avem. Latenţa (*latency*), sau durata propagării unei instrucţiuni prin pipeline, este mai mare decît în cazul unui procesor fără pipeline, pentru că am adăugat durata stocării datelor în regiştrii de separaţie. Pe de altă parte, observaţi că după ce prima instrucţiune iese din ţeavă, a doua se termină în ciclul imediat următor. Deci rata de execuţie (*throughput*) este de o instrucţiune pe ciclu de ceas! Distincţia între *latency* şi *throughput* este extrem de importantă.

Acest fenomen apare într-o formă exacerbată în cazul reţelelor de calculatoare, în care există doi parametri independenţi: durata propagării datelor între două calculatoare, şi viteza de transmisiune a datelor. Putem avea linii cu durată de propagare mică (de exemplu cu latenţa de 2ms), dar cu viteză mică, cum ar fi un modem de 14.4Kbps. Putem avea însă o linie cu durată de propagare extrem de mare (500ms), dar cu o viteza foarte mare, cum ar fi un canal de transmisiune prin satelit, de 2Mbps. Interesant este că în cazul reţelelor de calculatoare produsul acestor cantităţi (latenţa * rata de transmisie) este cel mai important; cu cît produsul este mai mare, cu atît susţinerea performanţei reţelei este mai greu de obţinut. Dar am divagat; sper să revin asupra acestei teme într-un alt articol. Înapoi la ţevile noastre.

Să observăm că viteza la care putem pune ceasul este limitată de **cel mai lent stagiu din ţeavă**. Asta pentru că toţi lucrătorii trebuie să lucreze cu ritmul celui mai încet dintre ei. Din această cauză, o împărţire a sarcinilor la 5 circuite nu garantează o creştere a vitezei de 5 ori. Să presupunem că cele 5 stagii durează 3, 3, 3, 5, respectiv 3 nanosecunde, şi că întîrzierea unui registru de separaţie este de 2 nanosecunde. Atunci circuitul fără ţeavă execută o instrucţiune la fiecare 3+3+3+5+3=17 nanosecunde, şi asta dă şi viteza ceasului. Pe de altă parte, circuitul din "felii" execută o instrucţiune la 5+2 nanosecunde (5 pentru stagiul cel mai lent, plus două ns pentru propagarea prin registru). Iată deci cum, deşi am împărţit sarcina la 5, din cauza imbalansului creşterea de viteză obţinută este de numai $17/7 = 2.42$ ori.

3 Dependenţe

Trebuie să vă temperez şi mai tare entuziasmul în ceea ce priveşte pipeline-urile: mai există o grămadă de probleme pe care le-am trecut cu vederea, dar care devin evidente de îndată ce ne aplecăm puţin asupra construcţiei.

Problema este că, foarte adesea, nu putem executa mai multe instrucţiuni consecutive chiar una după alta, pentru că anumite constrîngeri fac acest lucru imposibil. Acest gen de interferenţă între instrucţiuni consecutive se numeşte în engleză *hazard*. Voi folosi în româna termenul "dependenţe", deşi acesta nu este tocmai exact[2]. Despre ce fel de dependenţe este vorba?

[2]Pentru a fi precis, hazardurile sunt de mai multe tipuri, iar unele hazarduri se numesc *dependencies*. Oricum, aproximarea mea nu este complet nerezonabilă.

3.1 Dependenţe structurale

Prima problemă care poate apărea provine din faptul că una dintre aserţiunile mele de mai sus poate fi falsă; anume aceasta: "cînd folosea unul dintre [stagii], celelalte erau inutile". Iată un exemplu în care acest lucru nu este adevărat: un procesor trebuie după fiecare instrucţiune să incrementeze adresa de unde se ia următoarea instrucţiune (adresa este aflată în registrul numit "*Program Counter*"). Pentru că incrementarea este o operaţiune aritmetică, procesorul ar putea folosi pentru acest scop unitatea aritmetică-logică (în figura noastră am fi avut în loc de ALU şi circuitul de incrementare un singur circuit). Aici avem deci un conflict: o altă instrucţiune, aflată în stagiul de calcul ar putea dori să folosească acea unitate în acelaşi timp pentru că trebuie să adune două numere.

Astfel de "hazards" sunt numite "structurale", pentru că structura procesorului nu permite executarea anumitor tipuri de instrucţiuni simultan în stagii diferite. În exemplul nostru, instrucţiunile care nu folosesc unitatea aritmetică (de pildă o instrucţiune de salt absolut) nu cauzează nici un fel de conflicte.

Putem da şi alte exemple de dependenţe structurale: mai multe instrucţiuni vor să acceseze simultan acelaşi registru, mai multe instrucţiuni vor să acceseze memoria (de pildă o instrucţiune care vrea să-şi adune operanzii şi tocmai îi citeşte şi una care a terminat şi vrea să scrie rezultatul), sau instrucţiuni a căror execuţie durează mai mulţi cicli de ceas.

Un exemplu de ultima speţă sunt de pildă operaţiile în virgulă flotantă (adică cu numere "reale", nu întregi) care durează uneori zeci de cicli de ceas, iar procesorul de obicei are o singură unitate de calcul în virgulă flotantă.

Teoretic un hazard structural se poate oricînd evita duplicînd unităţile funcţionale care sunt în conflict; această soluţie nu este însă întotdeauna fezabilă. De pildă, dacă o împărţire[3] durează 10 cicli, atunci ar trebui să avem 10 împărţitoare pentru a permite execuţia a 10 împărţiri succesive.

Tot pentru a evita dependenţele structurale procesoarele moderne au, aşa cum arătam în figură, două cache-uri L1[4] separate: unul pentru instrucţiuni (I-cache) şi unul pentru date (D-cache): în acest fel se poate citi o instrucţiune simultan cu scrierea rezultatelor alteia.

Vom vedea un pic mai jos cum rezolvă un procesor astfel de dependenţe.

3.2 Dependenţe ale datelor

O dependenţă mult mai subtilă este cea a datelor. Să presupunem că avem un program cu două instrucţiuni consecutive: una care scrie numărul 2 în registrul 3, iar următoarea care adaugă valoarea 5 acelui registru.

În tabelul 2 vedem cum progresează aceste instrucţiuni într-o ţeavă ipotetică care seamănă cu cea descrisă mai sus. Am mai pus nişte instrucţiuni noop, care nu fac nimic, în jur, pentru a ilustra mai bine

Diagrama din tabelul 2 este tipică pentru a descrie evoluţia datelor într-un pipeline. Programul este scris pe verticală, ceasul este marcat pe orizontală. Starea ţevii poate fi citită pe verticală, de sus în jos. Căsuţa de la instrucţiunea 1 şi ceasul 3 arată în care dintre fazele ţevii se află acea instrucţiune la momentul 3, în cazul nostru în stagiul de acces la memorie (M).

[3]Presupunem că circuitul care face împărţirea nu poate le însuşi fi implementat ca un pipeline.

[4]Am scris cel puţin trei articole despre cache-uri în PC Report; le puteţi accesa din pagina mea de web.

Instrucţiunea	Ceasul 0	1	2	3	4	5	6	7	8
noop	C	D	E	M	S				
scrie 2 în R3		C	D	E	M	S^+			
aduna 5 la R3			C	D^+	E	M	S		
noop				C	D	E	M	S	
noop					C	D	E	M	S

Tabelul 2: Avansul instrucţiunilor într-un pipeline ideal. Am prescurtat stagiile cu **C**: citire, **D**: decodificare, **E**: execuţie, **M**: memorie, **S**: scriere. Această execuţie nu este posibilă (fără ajutor suplimentar), pentru că la momentul de timp 3 instrucţiunea de adunare vrea valoarea din R3, care va fi scrisă în R3 de instrucţiunea de scriere abia la momentul 5. Dependenţa este marcată cu două semne + în tabel.

Care e problema? Problema este că instrucţiunea de scriere pune datele în registrul 3 abia cînd atinge ultimul stagiu din ţeavă. Pe de altă parte, instrucţiunea de acumulare citeşte valoarea registrului 3 atunci cînd este în faza de decodificare. Dar datorită suprapunerilor, decodificarea instrucţiunii de acumulare se face la momentul 3, iar scrierea la momentul 5! Din cauza suprapunerii, în execuţie am inversat ordinea în timp în care se petrec două operaţiuni. Dacă nu facem nimic, rezultatul final va fi desigur greşit, pentru că acumularea nu vede efectul scrierii.

Înainte de a vedea ce e de făcut să inspectăm o altă dificultate care poate apărea.

3.3 Dependenţe ale controlului

Un tip special de dependenţe este cauzat de instrucţiunile de salt. O instrucţiune de salt indică o întrerupere în fluxul normal al programului. Neplăcerea apare din faptul că execuţia instrucţiunii de salt se termină destul de tîrziu, abia cînd instrucţiunea a calculat adresa finală de destinaţie. Dar între timp în ţeavă au intrat o grămadă de instrucţiuni, toate cele care urmau imediat. Evident, acestea nu trebuie executate (sau vor fi executate dacă saltul este condiţionat de o condiţie care este falsă).

Astfel de dependenţe se numesc "dependenţe de control", din cauză că sunt produse de modificări în "controlul" (ordinea de execuţie) a programului.

Voi consacra un articol întreg acestui tip de dependenţe, pentru că efectul lor este catastrofal asupra performanţei, şi pentru că sunt extrem de greu de reparat fără a pierde toate avantajele unui pipeline. Impactul salturilor este extrem de important din două motive:

- Din raţiuni de performanţă, procesoarele moderne au ţevi foarte foarte lungi (peste zece stagii); astfel de procesoare se numesc pompos "superpipelined". Ori, cu cît ţeava este mai lungă, cu atît o golire şi reîncărcare a ei costă mai mult. Ţeava nu mai este capabilă în acest caz să producă un rezultat la fiecare ciclu de ceas, pentru că, după o golire, timpul pînă la următorul rezultat este din nou egal cu latenţa, care este tot una cu numărul de stagii.

- Mai grav este faptul că, statistic s-a observat că, în medie, *una din 7 instrucţiuni executate este un salt.* Această densitate enormă de salturi în codul maşină este cu adevărat îngrijorătoare: dacă la fiecare 7 instrucţiuni trebuie să golim ţeava, atunci performanţa se duce cu totul de rîpă.

Să estimăm costul unei astfel de situaţii pentru procesorul nostru de mai sus, cu întîrzieri de 3, 3, 3, 5, 3 nanosecunde. Atunci procesorul va executa 7 instrucţiuni fiecare la cîte un ciclu, după care timp de 4 cicli va umple din nou ţeava golită de un salt. Asta înseamnă 7*6 + 4*6 = 66 cicli pentru 7 instrucţiuni, adică 66/7 = 9.42ns/instrucţiune în medie. Creşterea de performanţă a procesorului faţă de modelul fără ţeavă este acum de 17/9.42 = 1.8 ori. Şi încă am presupus că celelalte feluri de dependenţe nu cauzează nici o întîrziere!

3.4 Excepţii

Mai avem de-a face cu o ultimă neplăcere, care este mult mai greu de rezolvat decît cele indicate anterior, şi despre care nici nu vom vorbi prea mult în acest articol.

Cînd anumite condiţii excepţionale se ivesc, procesorul trebuie să întrerupă complet fluxul execuţiei, să execute o rutină specială, iar apoi uneori să reia programul întrerupt din exact acelaşi punct. Evenimentele care cauzează această întrerupere intempestivă se numesc excepţii. Există multe feluri de excepţii, iar tratamentul lor depinde de tip. Exemple de excepţii: împărţirea prin zero, accesul la o pagină de memorie virtuală care nu se află în memoria fizică, indicaţia terminării unui transfer de către un dispozitiv periferic (ex. discul), întîlnirea unui punct de oprire (*breakpoint*) pus de un program de depanare, etc.

Problema cea mai mare nu este în asemenea cazuri oprirea programului şi saltul (care seamănă teribil cu o instrucţiune obişnuită de salt), ci repornirea. În momentul apariţiei unei excepţii, în ţeavă se pot afla o sumedenie de instrucţiuni, cine ştie de unde de prin memorie (poate la una dintre ele s-a ajuns printr-un salt sau chiar o altă excepţie), etc.

Dacă procesorul vrea să poată relua execuţia după o excepţie, atunci trebuie să posede o grămadă de circuite care menţin foarte multă informaţie despre întreaga stare a ţevii, pentru a permite repornirea. Un astfel de pipeline se numeşte *restartable*, şi este extrem de complicat.

4 Soluţii

Concluzia care se desprinde este că e mai uşor de zis decît de făcut un pipeline. Dar lupta pentru supremaţie în performanţa se dă pe viaţă şi pe moarte între marile companii (nu e o metaforă: cei mai mari concurenţi ai lui Intel, Cyrix şi AMD au trebuit să fie cumpăraţi de alte mari companii, IBM, respectiv National Semiconductors, pentru a supravieţui).

Ca atare trebuie găsite soluţii. În restul articolului vom investiga nişte soluţii pentru problema dependenţelor (dar nu şi pentru cea a excepţiilor restartabile).

4.1 Compilatoare grijulii

O posibilă soluţie (care însă nu este folosită decît parţial) este ca în software să garantăm că astfel de lucruri nu se pot întîmpla. Compilatorul care generează cod pentru un microprocesor ar trebui să ne asigure că două instrucţiuni care se vor afla simultan în ţeavă nu vor interfera una cu alta. Compilatorul poate obţine acest efect umplînd spaţiul dintre două astfel de instrucţiuni cu instrucţiuni care nu fac nimic (no-op: no operation).

Instrucţiunea	Ceasul 0	1	2	3	4	5	6	7	8
noop	C	D	E	M	S				
noop		C	D	E	M	S			
scrie 2 în R3			C	D	E	M	S^+		
aduna 5 la R3				C	C*	C*	D^+	E	M
noop							C	D	E

Tabelul 3: Avansul instrucţiunilor într-un pipeline cu blocaj. Steluţa indică un blocaj: la momentul respectiv instrucţiunea indicată rămîne în acelaşi stagiu. Pentru că starea ţevii se poate citi pe fiecare coloană, observaţi că la momentul de timp 4 numai stagiile **C, E, M** şi **S** apar; asta înseamnă că stagiul **D** conţine o bulă. La momentul următor bula se propagă în stagiul **E**, şi o nouă bulă apare în stagiul **D**. De îndată ce instrucţiunea care scrie în R3 ajunge în stagiul S, instrucţiunea următoare poate continua, pentru că a obţinut rezultatul. Observaţi că acum dependenţa este rezolvată, pentru că semnele + sunt în aceeaşi coloană, deci rezultatul este deja disponibil cînd este cerut.

Din păcate soluţia aceasta nu este viabilă. Un motiv este că ar trebui ca compilatorul să aibă cunoştinţe intime despre arhitectura internă a ţevii (ca să ştie ce depinde de cine). Dar Pentium, Pentium Pro şi Pentium II au arhitecturi interne complet diferite, deşi implementează acelaşi set de instrucţiuni; ne-ar trebui deci un compilator diferit pentru fiecare maşina; mai mult decît atît, programele de pe una n-ar mai merge pe alta, din cauză ca altele ar fi dependenţele care trebuie evitate.

Pe de altă parte, compilatoarele moderne încearcă din răsputeri să ajute hardware-ul, fără a garanta neapărat generarea unui cod lipsit complet de dependenţe. Operaţiunea numită *code scheduling* (ordonarea codului) este extrem de importantă pentru a mări performanţa programelor. Practic compilatoarele încearcă să aranjeze instrucţiunile codului în aşa fel încît instrucţiuni care depind una de alta (cum sunt cele două de mai sus) sunt cît mai departe una de alta. De exemplu, dacă după cele două instrucţiuni de mai sus vine o instrucţiune care incrementează registrul 2, atunci ultimele două instrucţiuni pot fi schimbate între ele fără a modifica rezultatul programului, tocmai pentru că sunt independente. Îmi propun să discut într-un articol separat despre tehnicile de creştere a performanţei folosite de compilatoare, aşa că trec acum la prezentarea primei soluţii reale folosite pentru a rezolva problema dependenţelor.

4.2 Blocajul (stall)

Dacă o instrucţiune nu poate progresa în ţeavă din cauză că-i lipsesc anumite resurse (în exemplul de mai sus, registrul 3 încă nu are valoarea necesară), atunci aceste instrucţiuni sunt pur şi simplu ţinute pe loc în ţeavă în aceleaşi stagii, în timp ce cele care le preced sunt lăsate să continue. Oprirea unei instrucţiuni se numeşte blocaj, sau *stall*. În spatele instrucţiunii care continuă se formează un gol, numit "bulă" (cu b mic) (*bubble*). Bula este de fapt o instrucţiune noop: no operation, care nu are nici un efect.

Tabelul 3 prezintă evoluţia programului de mai sus atunci cînd apare o bulă.

De îndată ce instrucţiunea care avea resursele cerute îşi termină execuţia, instrucţiunile de după ele, care aveau nevoie de resurse, îşi pot continua execuţia în mod obişnuit. Blocarea unei instrucţiuni în ţeavă este relativ uşor de produs: registrul de separaţie de dinaintea acelui stagiu nu mai citeşte valorile produse de stagiul anterior, ci păstrează vechiul său conţinut.

Soluţia prin care compilatorul inserează instrucţiuni inutile se numeşte *statică*, pentru că programul apoi rămîne neschimbat. Prin contrast, atunci cînd bulele sunt create de microprocesorul însuşi atunci cînd programul se execută, tehnica se numeşte *dinamică*. Observaţi că şi în acest caz programul din memorie este neschimbat: bula apare doar în ţeavă, şi apoi dispare.

Soluţia asta pare acceptabilă. Care este costul pe care trebuie să-l plătim?

În primul rînd trebuie hardware special pentru a detecta dependenţele. Asta înseamnă practic o serie de comparatoare: un comparator compară registrul în care scrie instrucţiunea care se află în stagiul 4 cu regiştrii citiţi de instrucţiunile din stagiile 3, 2. Un alt comparator se uită să vadă dacă tipurile de instrucţiuni din aceste stagii într-adevăr folosesc regiştri (de pildă o instrucţiune de salt imediat nu foloseşte nici un registru. Dacă, de pildă, instrucţiunea din stagiul 4 va scrie în registrul consumat de instrucţiunea din stagiul 2, atunci stagiul 2 este blocat, stagiile 3 şi 4 avansează, iar în stagiul 3 se formează o bulă.

Un al doilea preţ pe care-l plătim pentru blocaj este scăderea performanţei. Din cauză că instrucţiunile nu avansează şi ţeava procesează nimicuri, rata efectivă cu care instrucţiunile sunt executate scade sub una pe ciclu. Cît de mult scade, depinde de o sumedenie de factori, începînd cu calitatea compilatorului şi terminînd cu capacitatea ţevii de a evita blocajele prin următorul mijloc pe care-l analizăm, înaintarea.

4.3 Înaintarea (forwarding)

În lupta lor acerbă cu nanosecundele, proiectanţii de microprocesoare au găsit încă o soluţie pentru a rezolva dependenţele. În exemplul de mai sus, o instrucţiune aflată în stagiul de decodificare avea nevoie de nişte date pe care instrucţiunea aflată în stagiul de execuţie tocmai le calculase. Cea din stagiul de decodificare însă trebuia să aştepte ca rezultatul calculelor să fie pus într-un registru, ceea ce se va întîmpla abia mai tîrziu. Ideea este atunci simplă: din moment ce tot am deja rezultatul, de ce să mai aştept să fie scris în registru şi apoi să-l iau din nou? Ce-ar fi dacă producătorul ar trimite rezultatul pe o scurtătură direct la consumator?

Această tehnică se numeşte "înaintare" (*forwarding*), deşi strict vorbind este o "înapoiere", pentru că datele sunt trimise înapoi în ţeavă, de la producător la consumator. Asta permite instrucţiunii din stagiul de decodificare să-şi continue execuţia netulburată, fără a mai fi nevoie de introducerea unei bule. Înaintarea este ilustrată în figura 3

Soluţia este clar preferabilă, pentru că viteza de execuţie rămîne aceeaşi, o instrucţiune pe ciclu. Costul plătit este însă în hardware: pe lîngă circuitele prezente în cazul blocării, înaintarea are nevoie de o mulţime de trasee speciale şi circuite de selecţie (multiplexoare) pentru a trimite datele pe scurtături. Dar budgetul în tranzistoare al proiectanţilor este atît de mare încît nu se dau în lături de la aşa ceva.

Procesoarele reale folosesc deci o mixtură a celor trei tehnici prezentate anterior; nu orice se poate rezolva cu înaintare, aşa că blocajul este practic întotdeauna necesar. (Un exemplu ar fi o instrucţiune care termină de calculat un rezultat abia în stagiul 4, dar care este consumat de instrucţiunea următoare în stagiul 2. În momentul în care a doua instrucţiune vrea valoarea, ea nici nu există, deci înaintarea nu este de nici un folos., deci trebuie folosit un blocaj.)

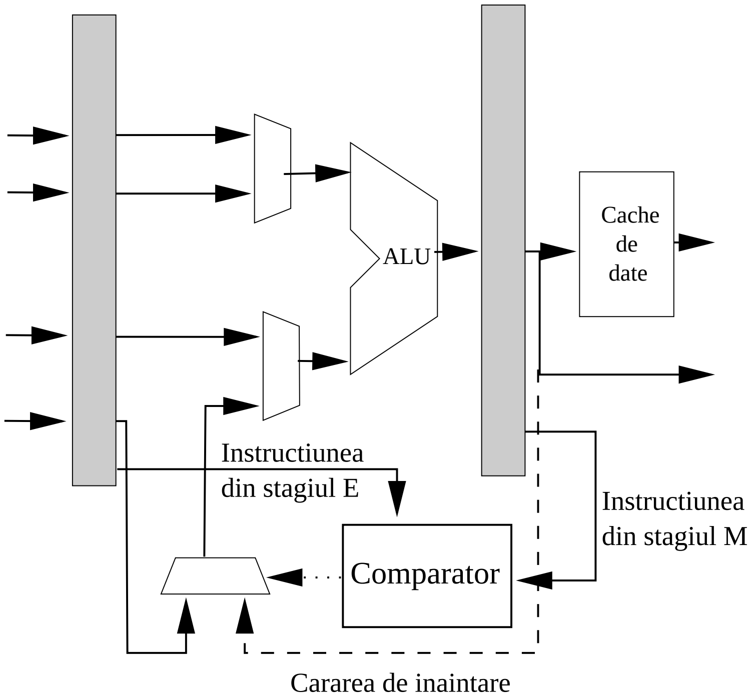

Figura 3: Stagiul de execuţie împreună cu cărarea de înaintare a datelor. Stagiul de execuţie poate prelua o valoare fie de la stagiul de decodificare, fie de la propria sa ieşire. Noul multiplexor introdus, în partea de jos a figurii, face această alegere. Multiplexorul este comandat de un comparator care se uită să vadă dacă una dintre instrucţiuni produce valoarea unui registru de care cealaltă are nevoie.

5 Variaţiuni şi concluzii

Încheiem aici incursiunea noastră în arhitectura procesoarelor-ţeavă. În comparaţie cu alte tehnici, pe care sperăm că le vom putea prezenta în articole ulterioare, pipelining-ul este pînă la urmă o metodă destul de simplă şi cu o eficacitate imediată. În procesoarele moderne nu numai fazele de execuţie ale unei instrucţiuni sunt "pipelined", ci şi operaţiile aritmetice; de pildă unele unităţi aritmetice calculează o înmulţire pe 32 de biţi în 5-6 cicli de ceas, dar pentru că înmulţitorul este pipelined, el poate accepta o nouă pereche de operanzi la fiecare ciclu, şi poate produce rezultate cu aceeaşi frecvenţă.

Procesoarele moderne folosesc un întreg arsenal de alte tehnici, extrem de sofisticate, dintre care enumerăm aici:

Superpipeline: am definit acest termen deja: indică un pipeline extrem de adînc, cîteodată de peste 10 stagii.

Superscalar: procesoarele superscalare au la dispoziţie mai multe pipelines paralele, şi pot lansa în execuţie mai multe instrucţiuni simultan. Pentium II are de pildă două ţevi, numite U şi V. Un astfel de procesor poate termina mai multe instrucţiuni în fiecare ciclu de ceas!

VLIW: este un acronim de la *Very Long Instruction Word*, adică “instrucţiuni foarte lungi”. Astfel de procesoare au instrucţiuni formate din mai multe instrucţiuni elementare (ex. o adunare şi o scădere simultan). Compilatorul are sarcina să grupeze instrucţiunile laolaltă în aşa fel încît să poată fi executate simultan. Diferenţa faţă de un superscalar este că superscalarul decide dinamic care instrucţiuni merg împreună, pe cînd la VLIW acest lucru se face la compilare.

Reordonare: procesoarele moderne sunt în stare să execute instrucţiunile în altă ordine decît cea în care sunt prezente în program; asta se numeşte *out of order execution*. Practic ele realizează echivalentul operaţiei de *scheduling* pe care o fac compilatoarele, dar o fac în mod dinamic. Avantajul este că dacă trebuie să blochezi o instrucţiune în ţeavă dar cele de după ea au toate resursele disponibile, poţi să le laşi s-o ia înainte, ţinînd ţeava ocupată.

Execuţia speculativă: este o tehnică folosită pentru a micşora impactul “hazardurilor” de control (adică dependenţele produse de salturi). Procesorul presupune că saltul se face într-o anumită direcţie şi încarcă în ţeavă instrucţiunile de la “destinaţie” (asta se numeşte speculaţie — *speculation*). Dacă se dovedeşte că a greşit în ghicirea adresei de destinaţie a saltului, atunci pur şi simplu şterge tot ce a încărcat greşit în ţeavă şi începe execuţia de la locul unde mergea în realitate saltul.

Prezicerea salturilor: am văzut că dependenţele de control sunt foarte costisitoare, şi forwarding nu ajută în cazul lor (pentru că adresa destinaţie este calculată relativ tîrziu, dar este necesară foarte devreme, în chiar primul stagiu, care aduce instrucţiuni din memorie). Pentru asta procesoarele moderne încearcă să “ghicească” în mod inteligent direcţia unui salt, bazîndu-se pe execuţiile trecute ale acelui salt. O cantitate enormă de hardware este dedicată acestui scop.

Dacă vrem să rezumăm învăţămintele acestui text într-o frază putem spune aşa: atunci cînd ai destulă mînă de lucru şi multe activităţi de făcut, e foarte economic să organizezi munca într-o bandă de asamblare; specializarea fiecărui lucrător garantează eficacitate sporită în muncă, iar durata medie de producere a unui rezultat este egală cu lungimea activităţii celui mai lent dintre lucrători.

Gata cu distracţia, acum la treabă. Fiecare să-şi reia locul în ţeavă!