Redenumirea regiştrilor
Seria: arhitectura modernă a calculatoarelor

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

18 iunie 1999

Subiect: tehnici de creştere a performanţei procesoarelor moderne: redenumirea regiştrilor (register renaming)

Cunoştinţe necesare: cunoştinţe elementare despre arhitectura procesoarelor

Cuvinte cheie: registru, redenumire, dependenţă

Contents

Cu cîtăva vreme în urmă am publicat în PC Report două articole despre arhitectura procesoarelor moderne, care se doreau a fi parte dintr-o suită despre acest subiect foarte generos. Din păcate (sau din fericire, depinzînd de perspectivă), m-am luat cu vorba despre Internet şi alte lucruri, şi nu am mai continuat pe acest subiect, deşi unul dintre articole promitea "demistificarea" unei liste impresionante de termeni.

Articolul de faţă este o continuare a celor două anterioare. Pentru o tratare matură a subiectului rămîne valabilă referinţa oferită cu acea ocazie, Hennessy and Patterson "Computer Architecture — a Quantitative Approach", Morgan Kaufmann, ediţia II, 1995.

1 Performanţă şi compatibilitate

Intel este cea mai mare companie din domeniul hardware, cu un venit anual de 26 de miliarde de dolari în 1998. Într-un fel ar fi de aşteptat ca cei care au inventat microprocesorul să fie şi liderii din punct de vedere economic. Pe de altă parte, adesea, tehnologic vorbind, compania a fost depăşită de altele în ceea ce priveşte performanţa microprocesoarelor realizate. De fapt, multă vreme procesoarele 8086, 80386, Pentium, etc. au fost cu mult mai slabe în performanţe decît procesoare contemporane lor ale unor firme concurente. Atunci cum se explică succesul economic nemaipomenit al lui Intel? Probabil că nu există un răspuns simplu, dar cel puţin o fărîmă de răspuns stă în *producţia de masă*. PC-urile sunt de departe cele mai răspîndite calculatoare acum, aşa că, chiar în condiţiile unor produse inferioare calitativ (dar cu preţuri mici) Intel a putut să cîştige mult mai mult decît celelalte companii. Cîştiguri care apoi au fost investite în cercetare şi dezvoltare, care au dus la crearea lui Pentium II, care este un chip cu adevărat extraordinar.

De fapt avantajele şi dezavantajele lui Intel provin din aceeaşi sursă, producţia de masă. Producţia de masă necesită compatibilitate între produse, pentru ca utilizatorii să poată beneficia de software-ul care a fost deja scris. Asta este un avantaj nemaipomenit, şi este cert că relaţia strînsă cu Microsoft a însemnat enorm în succesul lui Intel (Microsoft fiind acum compania cea mai mare din lume, socotind cotaţia la bursă). Dar compatibilitatea a fost şi blestemul lui Intel.

La începutul anilor '80, din laboratoarele de cercetare de la universităţile Stanford şi Berkeley ieşea un concept complet nou de arhitectură a procesoarelor: procesorul RISC, încarnat în procesorul MIPS (actualmente în posesiunea lui Silicon Graphics). RISC-urile sunt procesoare care implementează instrucţiuni extrem de simple, dar care profită de această simplitate pentru a rula la viteze extreme, folosind un hardware foarte eficace. Aparent RISC-ul era sortit să fie învingător, şi o sumă întreagă de companii au început să fabrice RISC-uri. Performanţele lor erau într-adevăr spectaculoase, comparat cu procesoarele de tip CISC, tradiţionale.

Aceasta era şi problema lui Intel: 8088 şi toţi descendenţii lui sunt de la început compatibile unul cu altul, deci trebuie să implementeze acelaşi set de instrucţiuni. Dar setul de instrucţiuni x86 (cum este abreviată familia) a fost proiectat înaintea revoluţiei RISC (mai exact în 1978), deci nu putea beneficia de toate avantajele tehnologice care pot fi aplicate în cazul acestora. Intel era sortită să rămînă în urmă.

Salvarea a venit însă dintr-o direcţie oarecum neaşteptată: din tehnologie. Intel a reuşit în ultimii ani să recupereze toate diferenţele faţă de competitorii săi, şi să livreze procesoare extrem de performante. Cum se explică acest lucru?

Diferenţa RISC–CISC este o diferenţă *relativă*; relativă la tehnologia curentă. Dimensiunea tranzistoarelor dintr-un circuit integrat în 1986, şi deci numărul de tranzistoare care se puteau construi, era limitat la o valoare în jurul a 100 000. Cu atîtea tranzistoare puteai construi o maşinărie RISC eficace, dar nu şi una CISC; puteai face CISC-uri doar lente, folosind microcod, pentru că sarcina decodificării şi executării unui set de instrucţiuni complex cerea mai multe resurse. Dar avansul implacabil al tehnologiei şi-a spus cuvîntul, dimensiunea şi viteza circuitelor se dublează la fiecare 18 luni, deci în 1995 Intel a avut la dispoziţie suficiente resurse pe pilula de siliciu pentru a lupta cot-la-cot cu RISC-urile, folosind propriile lor idei, cu Pentium II. Şi, cel puţin deocamdată, Intel a cîştigat, ajutată fiind şi de formidabila economie de masă a PC-ului.

1.1 Compatibilitatea şi regiştrii

Desigur, asta este o poveste interesantă, dar ce are a face cu arhitectura modernă a calculatoarelor? Ei bine, deşi partea economică este cu certitudine incitantă, cuvîntul cheie asupra căruia o să mă aplec pentru a construi acest articol este "compatibilitatea".

Pentium III poate încă executa cod scris pentru procesorul 8086. De fapt, majoritatea codului executat în lume pe procesoare Pentium a fost scris cu procesoare mai slabe decît 80286 în minte (la ora actuală cel mai rulat sistem de operare din lume este încă Windows 3.1). În plus, foarte multă lume scrie încă software pentru platforme vechi, pentru că baza instalată este uriaşă, şi altfel ar însemna să dai cu piciorul unei mulţimi de potenţiali clienţi. În plus, metoda obişnuită de distribuţie a software-ului este în formă de programe executabile, gata compilate. Asta înseamnă că o mulţime de programe folosesc numai facilităţile vechi ale procesorului, chiar dacă acesta are acum cu mult mai multe resurse.

De pildă, în mod esenţial numărul de regiştri de bază la Pentium este în continuare 4 (EAX, EBX, ECX şi EDX), deşi costul unui registru în hardware este nesemnificativ, iar performanţa obţinută din folosirea unui număr mare de regiştri este substanţială. Din motive de compatibilitate însă, Intel nu poate schimba radical setul de instrucţiuni, introducînd noi regiştri. Prin comparaţie, procesoarele moderne RISC au cel puţin cîte 32 de regiştri.

Regiştrii sunt foarte importanţi pentru performanţă pentru că accesul la datele din regiştri este foarte rapid (de fapt, ce sunt altceva regiştrii, decît o foarte mică memorie cache aflată chiar pe procesor; un cache al cărui management este făcut de compilator?). Odată cu miniaturizarea şi creşterea vitezelor de ceas, diferenţa de durată între accesele la regiştri şi cele la memorie creşte îngrijorător (de exemplu am văzut într-un articol din PC Report nişte măsurători pentru un sistem Pentium 266Mhz, la care accesul la memorie putea dura de 13 ori mai mult decît cel al un registru!). Diferenţa aceasta este de sute de cicli pentru cazul multi-procesoarelor, care au nevoie de mecanisme complicate de arbitrare a accesului la memorie.

Desigur, aici contrastăm viteza de acces la memoria principală, dar foarte adesea datele se vor afla de fapt în cache-ul microprocesorului. Chiar şi aşa, cache-urile moderne L1 nu sunt capabile să ţină pasul cu viteza procesoarelor, oferind timpi de acces de ordinul a 2–4 cicli, timpi care vor creşte în viitor (pentru că ciclul scade)[1].

Dacă are mulţi regiştri la dispoziţie, un compilator poate aplica o serie mai largă de optimizări, şi are mai multă libertate în plasamentul valorilor, putînd optimiza mai eficace programele. Un singur registru în plus poate însemna foarte mult pentru eficienţa codului compilat. Vom vedea că un număr redus de regiştri forţează compilatorul să reutilizeze aceiaşi regiştri pentru lucruri diferite. Acesta reutilizare înseamnă dependenţe între instrucţiuni, care la rîndul lor cauzează imposibilitatea de execuţie a instrucţiunilor simultan.

Vom vedea în acest articol o soluţie foarte ingenioasă a acestei probleme; vom vedea cum, dînd iluzia programelor că au la dispoziţie numai un număr foarte redus de regiştri, procesoarele sunt capabile să utilizeze intern mai multe, obţinînd majoritatea beneficiilor descrise mai sus.

[1]Despre cache-uri am scris cu mai mult timp în urmă în PC Report mai multe articole ample.

2 Paralelism şi dependenţe

Dacă 50% din creşterea în performanţă a procesoarelor contemporane provine cu certitudine din aportul tehnologiei, care permite folosirea unor ceasuri din ce în ce mai rapide, cealaltă jumătate trebuie sa fie atribuită inovaţiilor hardware, şi mai cu seama, paralelismului exploatat.

Paralelism înseamnă că mai multe activităţi independente se desfăşoară simultan. Nu prea este clar despre ce fel de paralelism poate fi vorba în cazul procesoarelor: acestea primesc doar un singur program, care este o secvenţă de instrucţiuni, pe care trebuie să-l execute. Ce se poate face în paralel atunci?

Există mai multe feluri de paralelism, de obicei categorisit după granularitatea sarcinilor executate în paralel. De pildă, un sistem de operare este capabil să execute simultan mai multe aplicaţii; acesta este paralelismul la nivel de aplicaţie. Procesoarele însă acţionează la un nivel microscopic, privind doar la cîte o instrucţiune din program. Ele manipulează paralelism la o granularitate infimă, comparat cu paralelismul proceselor.

Chiar dacă programele scrise de noi denotă o suită de acţiuni care trebuie efectuate într-o anumită secvenţă, există o cantitate oarecare de libertate în ordinea în care acestea sunt îndeplinite. Dacă, de pildă, iniţializăm mai multe variabile, adesea aceste operaţii pot fi executate în orice ordine, obţinînd aceleaşi efecte.

Desigur, două instrucţiuni se pot executa simultan numai dacă nu depind una de alta. Două instrucţiuni ca "`f=1; g=f+2`" nu se pot executa simultan, pentru că a doua are nevoie de rezultatul primeia. Spunem atunci că a doua instrucţiune depinde de prima, sau că între ele există o *dependenţă*.

Atunci cînd programele sunt traduse în cod-maşină, între micile instrucţiuni rezultate adesea se găsesc unele care sunt *independente*, deci care se pot executa în orice ordine. Aceste instrucţiuni pot fi executate deci şi în paralel, dacă avem resursele necesare la dispoziţie. Acest gen de paralelism este extrem de important, şi are propriul său nume: "paralelism la nivel de instrucţiune", sau Instruction Level Parallelism, ILP.

3 Procesoare superscalare şi LIW

Majoritatea procesoarelor moderne exploatează ILP într-un mod foarte natural: au mai multe unităţi din fiecare fel, care le permit să execute mai multe instrucţiuni simultan. Astfel, ele trebuie să aibă mai multe unităţi care aduc instrucţiuni din memorie, care le decodifică, care le execută şi care stochează rezultatele. (Articolul citat despre pipelines discută puţin mai în detaliu despre feluritele unităţi dintr-un microprocesor.)

Există două categorii mari de procesoare care exploatează ILP executînd instrucţiuni în paralel; ele se deosebesc după felul în care se decide *care* instrucţiuni se pot simultaneiza.

LIW: Prima mare categorie de procesoare lasă decizia în cîrca compilatorului; acesta are misiunea să indice care instrucţiuni sunt independente. Hardware-ul doar ia instrucţiunile indicate şi le trimite unor unităţi separate. Hardware-ul este simplu, iar compilatorul este complicat. De fapt putem vedea compilatorul ca oferind hardware-ului nişte "super-instrucţiuni", formate din mai multe instrucţiuni elementare; din cauza asta astfel de procesoare se numesc "Long Instruction Word" (LIW), sau cîteodată VLIW (Very LIW). De exemplu procesorul de semnal Texas

Instruments TMS320C62x primeşte simultan pînă la 8 instrucţiuni.

Superscalare: A doua clasă de procesoare calculează dependenţele în întregime în hardware, şi decide la faţa locului care instrucţiuni pot fi lansate în paralel[2]. Pentium II de pildă citeşte, decodifică şi încearcă să execute cîte două instrucţiuni la fiecare bătaie de ceas. Aceste procesoare se numesc *superscalare*.

4 Pipelines (conducte)

Pe lîngă această metodă de a exploata ILP, există o alta foarte ingenioasă, numită "banda de asamblare", sau "conductă". Am scris un articol întreg despre acest subiect, dar iată ideile esenţiale: dacă avem o suită de acţiuni de efectuat şi mai multă forţă de muncă, şi dacă putem descompune fiecare acţiune în mai multe bucăţele, atunci putem construi o bandă de asamblare, în care fiecare porţiune a benzii face o singură parte. La fel ca zidarii: unul ia cărămidă, unul o întinde, unul pune mortar şi unul o înfige în perete.

La fel stau lucrurile şi în cazul procesoarelor: citirea, decodificarea, execuţia, stocarea rezultatelor, sunt tot atîtea micro-acţiuni, care pot fi executate simultan pentru instrucţiuni diferite: cînd o instrucţiune tocmai se termină, cea de după ea stochează rezultatele, următoarea tocmai şi le calculează în timp ce a patra este decodificată, etc.

Acest gen de paralelism se numeşte paralelism de pipeline. La o vedere superficială paralelismul de pipeline nu este afectat de dependenţe; în realitate acestea sunt la fel de importante ca şi în cazul superscalarelor. Să ne gîndim un pic: dacă avem cele două instrucţiuni de mai sus, dependente, "`f=1; g=f+2`", atunci cînd `f+2` vrea să citească valoarea lui `f` pentru a face calcule cu ea, valoarea de fapt încă nu a fost calculată, şi nici stocată unde trebuie, pentru că instrucţiunea `f=1` se află încă în stagiul de execuţie. În analogia cu zidarii, este ca şi cum un zidar fabrică chiar mortarul de care are nevoie unul dinaintea lui; cel care pune mortar nu are cum să acţioneze înainte ca mortarul să existe.

5 Dependenţele sub lupă

Dacă sunteţi nerăbdători să aflaţi ce are asta a face cu redenumirea regiştrilor, o să fac aici o scurtă avanpremieră. Vom vedea că anumite dependenţe sunt de fapt artificial introduse, din cauză că unii regiştri trebuie refolosiţi pentru a stoca variabile complet diferite; dacă am avea mai mulţi regiştri, ca să punem o variabilă în fiecare, aceste dependenţe ar dispărea. Ei bine, redenumirea regiştrilor tocmai asta va face: va folosi o găleată cu regiştri ascunşi, pe care-i va folosi în astfel de cazuri, eliminînd anumite dependenţe şi mărind gradul de paralelism de care procesorul poate profita.

În general între două instrucţiuni există o dependenţă dacă folosesc acelaşi registru sau aceeaşi adresă de memorie. Acesta însă nu este un criteriu suficient; contează şi *cum* folosesc acest registru comun. De pildă dacă două instrucţiuni citesc dintr-un acelaşi registru, între ele nu există nici o dependenţă, pentru că acţiunea de citire lasă registrul neschimbat, deci ordinea instrucţiunilor nu

[2]Pentru o comparaţie sumară între expresivitatea metodelor statice (compilarea) şi a celor dinamice vedeţi şi articolul meu din PC Report din luna iunie 1999, despre debugger-e.

influenţează rezultatul. Dacă una din instrucţiuni însă *scrie* în registru, atunci avem o dependenţă. Dependenţele se denotează cu 3 litere: Acţiune-După-Acţiune, de pildă Scrie-După-Citire. Denumirile tradiţionale sunt în engleză:

RAW Read after write. Avem o astfel de dependenţă în exemplul anterior, reprodus şi în figura 1.

```
1: f = 1
2: g = f + 2
```

Figura 1: O dependenţă RAW: `f` este citit de instrucţiunea 2 după ce ce este scris de către instrucţiunea 1.

A doua instrucţiune citeşte valoarea lui `g` după ce prima o scrie. Această dependenţă se mai numeşte şi "dependenţă adevărată" (true dependence).

WAR apare cînd o instrucţiune scrie într-un registru care tocmai a fost citit, cum ilustrează şi figura 2.

```
1: a = b + c
2: b = c + 1

...

x: mem[10] = b  ← Instructiunea care-l "foloseste" pe b
```

Figura 2: O dependenţă WAR: `b` este scris de instrucţiunea 2, după ce a fost citit de instrucţiunea 1.

Din nou, nu putem schimba ordinea instrucţiunilor, sau nu putem risca să le executăm în paralel, pentru că, dacă a doua se termină înainte ca prima să citească valoarea, rezultatul primeia va fi greşit. O astfel de dependenţă se mai numeşte *anti-dependenţă*.

WAW este ultimul tip, în care o două instrucţiuni scriu în aceeaşi valoare; vedeţi figura 3.

```
1: f = 1
2: f = g + 2

...

x: h = f * 3  ← Instructiunea care-l "foloseste" pe f
```

Figura 3: O dependenţă WAW apare cînd două instrucţiuni au aceeaşi destinaţie.

Aceste dependenţe se mai numesc şi *output dependences*.

Aparent dependenţele WAW nu trebuie să apară în programe: de ce compilatorul ar indica o instrucţiune al cărei efect să fie imediat distrus? De ce şi-ar bate proiectantul procesorului capul cu astfel de instrucţiuni? În realitate dependenţele WAW pot apărea în mai multe contexte:

1. Între instrucţiuni nu neapărat consecutive, cum este ilustrat în figura 4. Aici avem o dependenţă WAW între prima şi a treia instrucţiune;

```
1: y = a
2: x = y + 2
3: y = b
```

Figura 4: O dependenţă WAW şi una RAW la care participă o singură valoare.

2. Pe de altă parte, nu e treaba procesorului să decidă dacă un program are sau nu sens; poate cineva scrie de mînă un astfel de program; procesorul tot trebuie să dea rezultatele corecte;
3. Un alt caz de acest gen poate apărea dacă apare o întrerupere; deşi nu există nicăieri în program două scrieri consecutive în acelaşi registru, saltul la rutina de întrerupere poate cauza apariţia unei astfel de secvenţe;
4. În fine, procesoarele moderne folosesc o altă tehnică, numită *execuţie speculativă*: execută cod chiar ne-necesar, pentru că nu sunt sigure ce trebuie să facă, deci mai bine să facă ceva, orice, care s-ar putea dovedi util, decît să stea degeaba (intenţionez să consacru un articol întreg acestui subiect). Ei bine, dacă ai executat cod speculativ şi ai observat că de fapt nu trebuia, atunci poate vei dori să ştergi efectele codului speculativ; în acest caz pot apărea din nou dependenţe WAW.

Încă o dată: studiul dependenţelor este important, pentru că existenţa lor reduce posibilitatea de execuţie paralelă a mai multor instrucţiuni din program (fie prin paralelism superscalar, VLIW ori pipelined). Din fericire, în anumite, cazuri putem face ceva pentru a ameliora situaţia.

6 Redenumirea regiştrilor

În realitate numai dependenţele "adevărate" (RAW) sunt de ne-evitat. De celelalte putem scăpa redenumind regiştri. Observaţia cheie este că programele în final vor stoca toate rezultatele în memorie; pentru utilizator conţinutul regiştrilor nu este important. Dacă programul foloseşte registrul `x` sau `y`, nu are nici o importanţă atîta vreme cît obţinem acelaşi rezultat.

Figurile 5 şi 6 arată cum putem rescrie dependenţele WAR şi WAW de mai sus, obţinînd acelaşi efect. Vom presupune că avem la dispoziţie în fiecare caz cîte un registru nefolosit.

```
1: a = b + c
2: f = c + 1

...

x: mem[10] = f    <-- Instructiunea care-l "foloseste" pe b
```

Figura 5: Dependenţa WAR este eliminată înlocuind apariţiile lui `b` cu un registru nou (`f`).

Observaţi că ambele aceste programe produc exact acelaşi rezultat ca programele iniţiale.
Cred că acum începe să devină clar de fapt ce se întîmplă în miezul procesorului:

```
1: f = 1
2: a = g + 2
...
x: h = a * 3   <--   Instructiunea care-l "foloseste" pe f
```

Figura 6: Dependenţa WAW este eliminată înlocuind apariţiile lui `f` cu un registru nou (`a`).

1. Microprocesorul are intern foarte mulţi regiştri, dar expune programatorului numai cîţiva;
2. Cînd procesorul execută cod, detectează dependenţele care se pot rezolva (anti- şi output-). Atunci cînd găseşte astfel de dependenţe, în cazul în care posedă un registru intern liber, foloseşte acest registru în locul celui care provoca dependenţa.
3. În felul acesta procesorul transformă instrucţiuni dependente în instrucţiuni independente, care se pot apoi executa în paralel, mărind performanţa execuţiei.
4. Procesorul ţine minte regiştrii redenumiţi, şi apariţiile lor ulterioare sunt de asemenea redenumite.

6.1 Un exemplu

În loc de a prezenta algoritmul detaliat folosit pentru a redenumi regiştri, voi ilustra funcţionarea sa cu un exemplu. Figurile 7–14 arată “filmul” execuţiei unui progrămel mic de 5 instrucţiuni pe un procesor superscalar care poate executa simultan adunări şi înmulţiri. Presupunem că operaţia de adunare se poate efectua în 1 ciclu de ceas, iar cea de înmulţire în 2. Ilustrăm şi structurile de date menţinute de procesor pentru a ţine cont de redenumiri. Acest exemplu este adaptat după un exemplu al domnului Randy Bryant, prezentat la un curs de arhitectura calculatoarelor în toamna anului 1998 la Carnegie Mellon.

Vom presupune că procesorul nostru poate executa instrucţiuni în orice ordine, dacă sunt independente.

Dacă urmăriţi filmul cu atenţie, veţi observa că numai dependenţele adevărate apar, celelalte fiind eliminate de redenumire. Veţi vedea marcate instrucţiuni care aşteaptă una după alta datorită dependenţelor.

Desigur, exemplul acesta este pedagogic, pentru că de fapt şirul instrucţiunilor nu se termină niciodată; acest algoritm incîlcit este executat fără osteneală de procesor, de 500 de milioane de ori pe secundă!

7 De ce apar dependenţe inutile

Din moment ce procesorul are mulţi regiştri, şi din moment ce unele dependenţe pot fi evitate, de ce apar ele totuşi în program? Răspunsul este acelaşi: din motive de compatibilitate.

Setul de instrucţiuni al unui procesor trebuie să ofere vederi compatibile asupra procesorului, chiar dacă măruntaiele acestuia se schimbă foarte mult în timp.

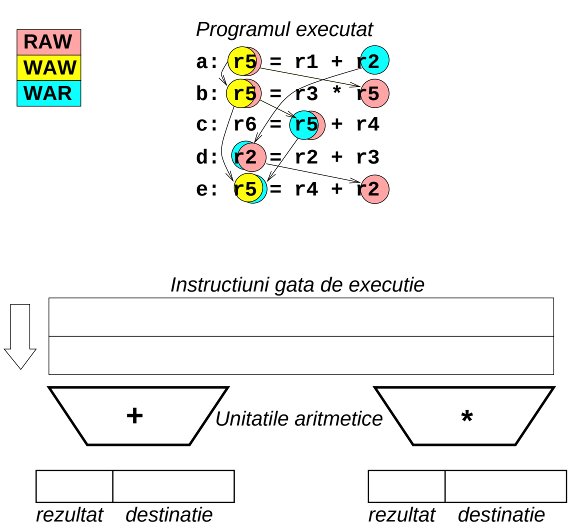

Figura 7: Programul de executat, valorile iniţiale ale regiştrilor, regiştrii ascunşi şi cozile de instrucţiuni gata de execuţie. Observaţi că săgeţile pentru dependenţe merg mereu în acelaşi "sens": WAW pe verticală, RAW de la stînga spre dreapta, WAR de la dreapta spre stînga. Cîmpul *valid* indică dacă acest registru ascuns are o valoare corectă în interior.

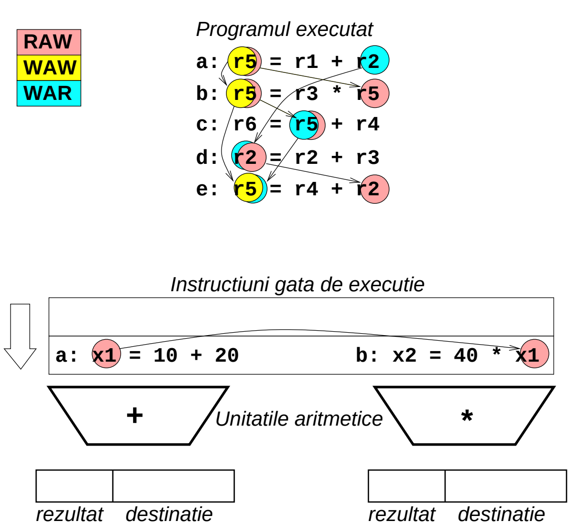

Figura 8: Ciclul 1: instrucţiunea `a` îşi redenumeşte destinaţia din `r5` în `x1`, iar instrucţiunea `b` îşi redenumeşte destinaţia `x2`. Instrucţiunea de înmulţire aşteaptă rezultatul din registrul `x1` (fostul `r5`, acum redenumit), pentru a putea începe: avem aici o dependenţă RAW, care nu poate fi eliminată. Observaţi că `r5` a fost deja redenumit de 2 ori. Numele de `x2` este stocat pentru `r5`, pentru ca următoarele instrucţiuni care au nevoie de `r5` să ia de fapt de aici valorile lor.

De pildă, x86 are în continuare doar 4 regiştri fundamentali; compilatorul nu are cum să folosească regiştri diferiţi pentru valori diferite, pentru că nu are destui regiştri la dispoziţie. Din cauza asta va re-folosi regiştri, cauzînd apariţia unor dependenţe.

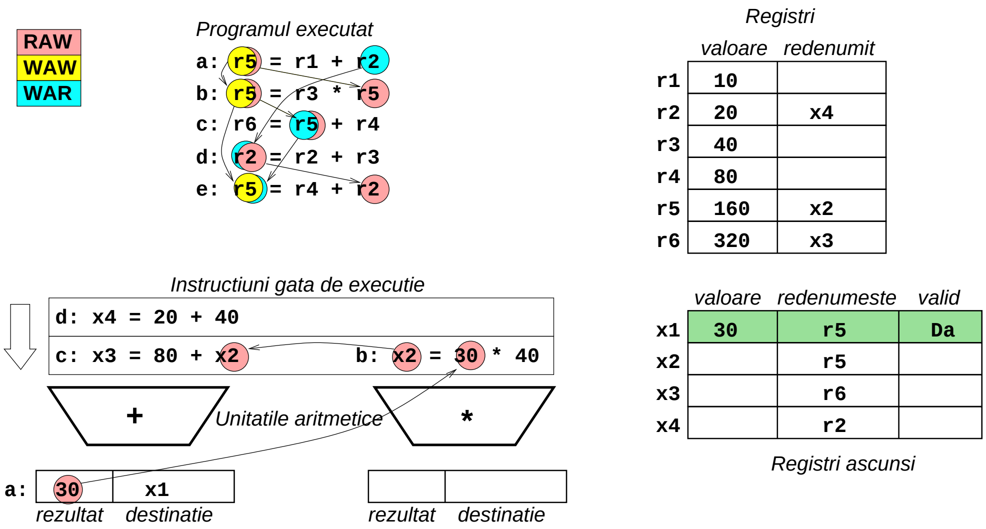

Figura 9: Ciclul 2: instrucţiunea `a` s-a terminat, şi a scris rezultatul în destinaţia ei, `x1`. Instrucţiunea de înmulţire poate începe, pentru că acum registrul `x1` este valid. Instrucţiunea `d` este gata de execuţie, dar instrucţiunea `c` nu a primit încă operandul din registrul `x2`, care este calculat de înmulţire. Instrucţiunea `c` şi-a redenumit destinaţia în `x3`, iar instrucţiunea `d` în `x4`.

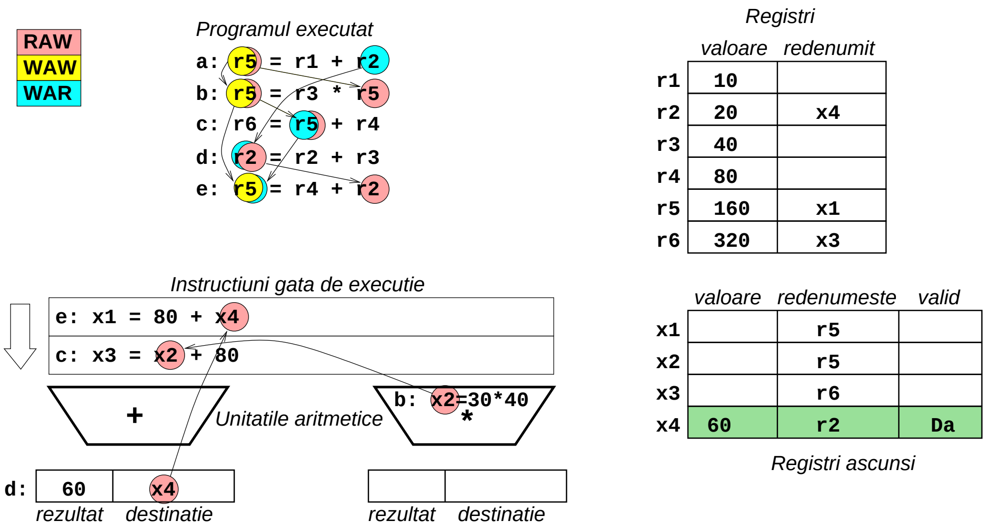

Figura 10: Ciclul 3: instrucţiunea `a` s-a terminat complet; cu toate acestea nu a schimbat registrul `r5`, pentru că acum acesta este redenumit în `x2`. Instrucţiunea `b` îşi începe execuţia pentru că are toate valorile necesare, dar termină numai primul din cei doi cicli necesari pentru execuţie. Instrucţiunea `d` o ia înaintea lui `c`. Instrucţiunea `e` este pregătită pentru execuţie şi îşi redenumeşte destinaţia din `r5` în `x1`. Observaţi ca valoarea din `x1` a fost distrusă; ea nu mai era necesară, datorită output-dependenţei. Puteam observa acest lucru, deoarece nici o instrucţiune nu mai avea nevoie de registrul `x1` ca sursă, şi nici un registru nu mai era redenumit în `x1`.

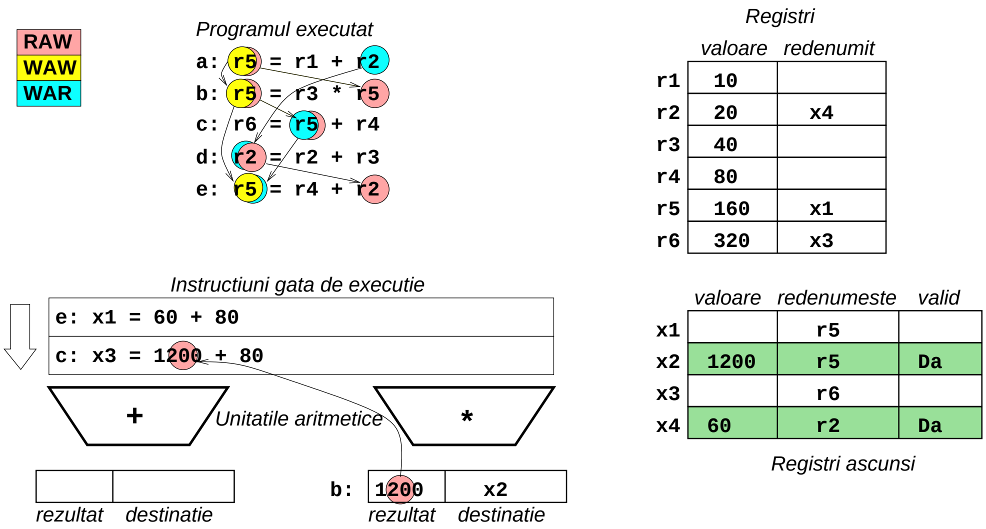

Figura 11: Ciclul 4: instrucţiunea de înmulţire `b` se termină şi oferă valoarea necesară executării instrucţiunii `c`. Instrucţiunea `c` începe execuţia. Instrucţiunea `d` nu poate fi considerată terminată, pentru că este posibil ca instrucţiunea dinaintea ei, `x` să genereze o excepţie, deci nu avem voie să stocăm încă rezultatele lui `d`.

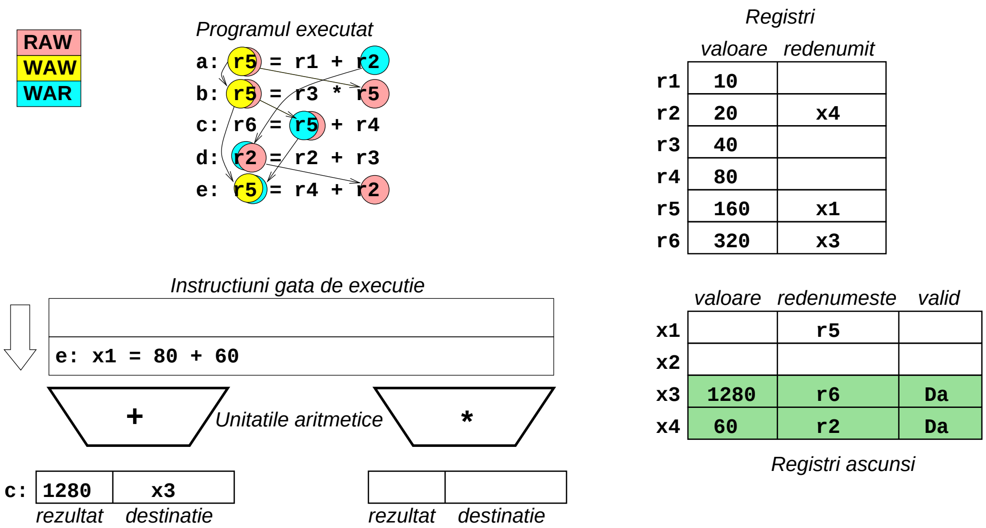

Figura 12: Ciclul 5: Instrucţiunea `b` s-a terminat, dar din nou, `r5` nu este schimbat. Datorită celei de-a doua dependenţe WAW, valoarea calculată este nu mai trebuie stocată. Încă nu putem termina `d`, pentru că `c` nu e gata. `c` tocmai se termină.

O problemă şi mai mare a arhitecturii x86 este faptul că regiştrii ei sunt *asimetrici*! La x86, majoritatea operaţiilor aritmetice trebuie să aibă registrul `eax` ca destinaţie, chiar dacă ceilalţi regiştri sunt liberi. Din cauza asta compilatorul este forţat să facă şi mai multe jonglerii cu regiştrii; probabil ca intern Pentium este implementat ca un RISC simetric, în care toţii regiştrii pot fi destinaţia unei valori, şi ca foloseşte din plin redenumirea regiştrilor pentru a lansa în execuţie

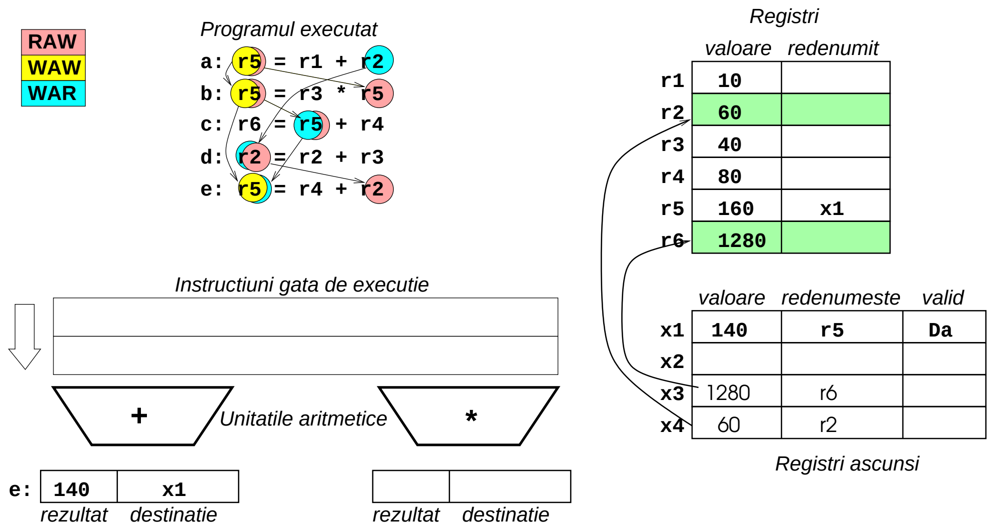

Figura 13: Ciclul 6: În fine, instrucţiunea `c` se termină, şi ca atare se poate termina şi `d`. Regiştrii ascunşi îşi copiază valorile înapoi în regiştrii reali, iar ultima instrucţiune intră în execuţie.

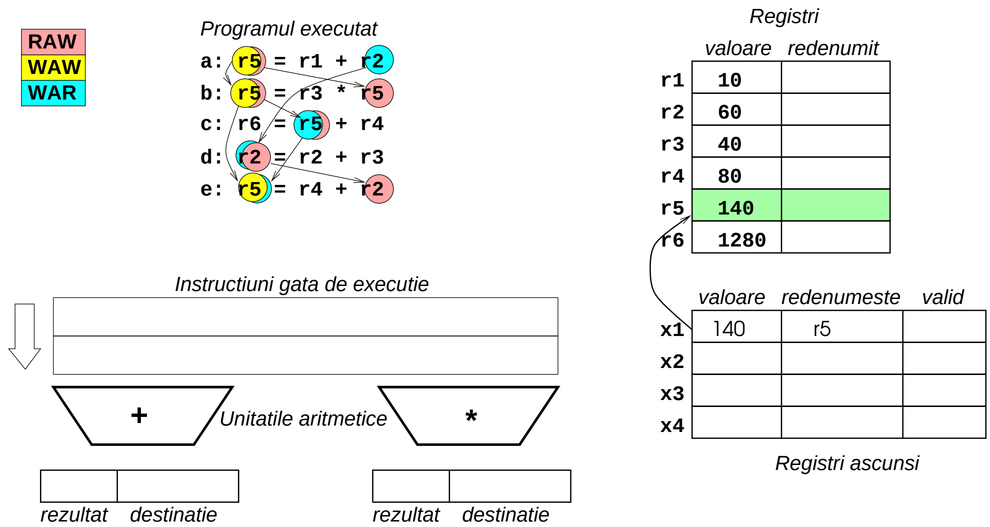

Figura 14: Ciclul 7: Ultima instrucţiune s-a terminat, iar rezultatul ei este preluat dintr-un registru ascuns în cel real.

mai multe instrucţiuni aritmetice simultan.

7.1 Un panaceu?

Trebuie să înţelegem că, deşi salvează din performanţă în mod substanţial, redenumirea regiştrilor nu este un panaceu pentru a rezolva problema lipsei de regiştri. Ce se întîmplă: să zicem că la un moment dat compilatorul are de manipulat 5 valori pentru un sistem x86, care are doar 4 regiştri fundamentali.

Ei bine, atunci compilatorul nu are altceva de făcut decît să “verse” (*spill*) o variabilă în memorie, de unde să o încarce într-un registru cînd e necesar. Pentru astfel de cazuri, mecanismul de redenumire este inutil.

8 Concluzii

În acest text am văzut cum tensiunea dintre compatibilitate şi tehnologie împinge arhitecţii microprocesoarelor la soluţii disperate, cum ar fi redenumirea regiştrilor. Această tehnică permite procesorului să menţină valori diferite în regiştrii săi ascunşi disponibili, chiar dacă compilatorului aceştia îi sunt invizibili. Făcînd acest lucru, procesorul reduce numărul dependenţelor dintre instrucţiuni, şi face execuţia paralelă a instrucţiunilor fezabilă, mărind productivitatea.

S-ar zice că Intel s-a săturat de problemele lui x86; în viitoarea arhitectură anunţată (de foarte, foarte multă vreme), IA64, Intel va implementa un procesor RISC cu 128 de regiştri. Să-i ajungă.