Designul limbajelor de asamblare

Prof.dr. Ion IVAN,
Catedra de Informatică Economică, A. S. E., Bucureşti

Limbajele de asamblare apar ca rezultat al proiectării microprocesoarelor. Limbajele evoluate de programare permit accesul la toate resursele sistemelor de calcul. Performanţa software este influenţată de modul în care a fost proiectat, de limbajul în care este elaborat şi de facilităţile oferite de limbajul de asamblare, la care se ajunge în final. Designul limbajelor de asamblare reprezintă un nou mod de abordare a construirii acestora în vederea ameliorării performanţei softwareului aplicativ.
Cuvinte cheie: *limaje de asamblare, instrucţiuni, ortogonalitate, performanţe.*

Introducere

Limbajele evoluate de programare elimină suportul învăţării programării în assembler pentru a avea acces la toate resursele sistemelor de calcul. Utilizarea corectă a funcţiilor din bibliotecile standard din programele scrise în limbajele C++ sau PASCAL oferă programatorilor posibilităţile de acces la orice nivel al resurselor.
În acest nou context, studierea unui limbaj de asamblare se justifică pentru eficientizarea unor secvenţe de program. Există posibilitatea de a introduce direct în textele sursă C/C++ secvenţe asm.
O altă motivaţie este înţelegerea exactă a unor mecanisme de manipulare a informaţiei (lucru pe stivă, lucru cu variabile pointer, definirea şi utilizarea funcţiilor virtuale).
Dacă au existat perioade în care limbajul de asamblare a fost considerat un produs natural, uneori imperfect, al procesului de proiectare a procesoarelor, acum se poate pune şi problema de a realiza microprocesoare pornind de la un limbaj de asamblare dat. Limbajul de asamblare este definit aşa fel încât să asigure eficienţa, măsurată statistic, fie la implementarea unor mecanisme noi de gestiune memorie, fie în generarea de secvenţe compacte, fie în reducerea duratei de execuţie a prelucrărilor. În continuare sunt analizate aspecte de bază ale designului limbajelor de asamblare. Programele scrise într-un limbaj de asamblare proiectat cu luarea în considerare a anumitor criterii de performanţă, vor propaga efecte pozitive în toate fazele realizării codului sursă şi ale utilizării softwareului la beneficiari. Se pune în evidenţă legătura dintre caracteristicile de ordin cantitativ şi laturile calita-tive ale unui limbaj de asamblare şi efectele de antrenare multiplă induse.

Structura instrucţiunii limbajului de asamblare

Programatorul în limbajele de asamblare, înainte de orice, trebuie să cunoască structura internă a instrucţiunilor, care diferă de la un limbaj la altul. Structura internă arată modul în care se dispun la nivel de biţi informaţiile privind: codul operaţiilor, etichetele şi expresiile de adresare ale operanzilor.
Şirul de biţi care memorează codul operaţiei are o lungime strict dependentă de numărul de operaţii de bază pe care le implementează limbajul de asamblare.
Astfel, dacă proiectantul limbajului de asamblare optează pentru un şir de opt biţi, limbajul de asamblare poate fi înzestrat cu maxim 256 de mnemonice asociate unor instrucţiuni diferite. Numărul foarte mare de mnemonice disponibile permite definiri de instrucţiuni diferite pentru aceleaşi operaţii în

cazul în care tipul operanzilor este altul. De exemplu, vor exista instrucţiuni pentru efectuarea adunării în binar, în zecimal împachetat şi în virgulă mobilă. Programatorul assembler va alege aritmetica în care lucrează strict dependent de contextul problemei pe care o rezolvă.

Dacă proiectantul limbajului de asamblare defineşte codul operaţiei ca un şir de şapte biţi, numărul mnemonicelor cu care va manipula programatorul va fi de cel mult 127. Un astfel de limbaj este mai sărac, posibilitaţile de alegere se reduc. Chiar dacă pot fi definiţi operanzi de tipuri diferite, implementarea aritmeticilor presupune apelarea de funcţii care să prelucreze datele fiecărui tip folosind operaţii dintre cele 127 implementate în limbaj. Efortul de programare este mult mai ridicat.

Este posibil ca numărul de instrucţiuni N, să fie mai mic decât 2^k unde k reprezintă lungimea şirului de biţi pe care este memorat codul instrucţiunilor. Gradul de ocupare, G este dat de relaţia:

$$G = \frac{N_1}{2^k} * 100 \qquad (1)$$

Gradul de neocupare Γ este dat de relaţia:

$$\Gamma = 100 - G \qquad (2)$$

Gradul de neocupare este cel care oferă posibilitatea designerilor să dezvolte un sistem de macrodefiniţii coerent sau să impună definirea de noi operaţii de bază ce conduc la reducerea efortului de programare.

Designul structural al limbajului de asamblare are la bază ipoteza conform căreia toate elementele limbajului au aceiaşi importanţă. Această viziune conduce la abordarea separată a componentelor instrucţiunii.

În designul structural resursele sunt privite de sinestătător. Operanzii pot fi stocaţi în registre (R), în memorie (M) sau pot fi definiţi în corpul instrucţiunii, imediat (I). Complexitatea limbajului de asamblare se modifică radical în cazul opţiunilor designerului pentru instrucţiunile cu doi operanzi.

Dacă se optează pentru intreaga gamă de structuri de operanzi, vor fi definite instrucţiuni de tip R-R, R-M, R-I, M-M, M-I. Structurile interne ale acestor tipuri vor ocupa zone de memorie de lungime variabilă.

În cazul în care există 16 registre de lucru codul operaţiei ocupă 8 biţi, o instrucţiune de tip R-R va fi definită pe cel puţin 16 biţi. Pentru operaţii R-R se asociază coduri distincte.

În cazul în care codul operaţiei este unic, nedepinzând de operanzi, sunt necesari încă 3 biţi pentru a indica tipul (000 pentru tipul R-R, 001 pentru tipul R-M, 010 pentru tipul R-I etc.).

Limbajele de asamblare ale generaţiilor de microprocesoare actuale evită tipul de instrucţiuni M-M, având definite mnemonice pentru lucrul cu şiruri de caractere, suficient de flexibile.

Expresiile de adresare se evaluează în timpul asamblării, semnificaţiile fiind date de coduri memorate pe anumite poziţii din corpul structurii interne a instrucţiunii.

Folosind o zonă de 2 biţi se pot obţine toate combinaţiile ce corespund atributelor direct / indirect şi indexat / neindexat. Pentru tipologii identificate şi limitate ca număr de expresii de adresare se stabileşte o zonă care memorează coduri asociate care prin interpretare în timpul execuţiei permit utilizarea corectă a informaţiilor din zonele "operanzi" din structura internă a instrucţiunii. Codurile conduc la posibilitatea realizării unei variabilităţi a lungimii zonei de memorie ocupată de informaţiile despre operanzi.

Designul limbajului de asamblare pentru microprocesoarele 80x86 a condus la o structură de instrucţiune internă specifică tipurilor de instrucţiuni R-R, R-M, R-I, codul operaţiei are încorporate informaţii privind formatul

datelor (bait, cuvânt, cuvânt dublu) şi direcţia de parcurgere (adresare sau traversare) a memoriei, figura 1.

biţii

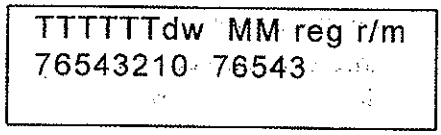

Fig. 1 Structura internă a instrucţiunii

unde:
T - biţi pentru codul operaţiei;
d - direcţie de parcurgere;
w - tip operand (bait/cuvânt);
MM - interpretarea deplasării;
reg - registre (semiregistre) codificate;
r/m - tipologii expresii de adresare.

Limbajele de asamblare mai vechi, datorită numărului mare de registre şi a aritmeticilor implementate, au regrupat expresiile de adresare pe un număr mai restrâns de biţi. Gradul de dependenţă a câmpurilor din structura instrucţiunilor era foarte slabă .

Limbajul de asamblare a microprocesorului convenţional x86 are definit un grad de dependentă ridicat între elmentele din structura instrucţiunii. Astfel, deplasarea este interpretată funcţie de câmpul w (câmpul MM depinde de câmpul w), iar câmpul reg depinde ca interpretare tot de câmpul w. Câmpul r/m depinde de câmpul MM. Dacă valoarea MM este 11, câmpul r/m este interpretat reg.

Designul corectiv pentru limbajele de asamblare

Pentru limbaje precum Fortran, Cobol, PL/1 în timp au fost efectuate analize statistice, toate evidenţiind neuniformitatea cu care programatorii utilizează in-strucţiunile şi tipurile de date şi de structuri oferite.

În mod normal, experienţa oferită de utilizarea facilităţilor ar trebui să conducă la perfecţionarea limbajelor, aspect realizat prin trecerea de la FORTRAN IV la FORTRAN '77 şi acum deja există FORTRAN'90. Mai dinamic, limbajul C a înregistrat evoluţiile C++ şi VISUAL C++.

Limbajele de asamblare sunt cele chemate să permită implementarea noilor mecanisme. Avantajele lu-crului pe stivă a condus la imple-mentarea de instrucţiuni (PUSH, POP) absente la primele limbaje. Frecvenţa redusă de lucru în aritmetica zecimal împachetată a condus la excluderea din lista mnemonicelor a elementelor corespunzătoare operaţiilor acestei aritmetici.

Se presupune existenţa unui program, simplu de altfel, care citeşte fişiere cu texte sursă assembler şi contorizează instrucţiunile. Se observă clar că frecvenţa cu care apar instrucţiuni precum **mov, add, inc, dec** diferă radical de frecvenţa cu care apar instrucţiuni ca **hlt, aaa, aad, das, lock, les, xchg**. De asemenea, dacă alături de frecvenţele cu care sunt utilizate resursele (registre, zone de memorie) se vor putea defini asocieri operaţii-resurse care trebuie tratate distinct.

Existenţa metodelor statistice moderne oferă un instrument eficient de grupare a instrucţiunilor legate de operanzi ca un tot si tratarea distinctă a instrucţiunilor din punct de vedere al utilizării.

Toate clasificările instrucţiunilor (după tipul operaţiei, natura operanzilor, tipul operanzilor) sunt strict legate de latura semantică a limbajului. Metodele de clasificare statistice oferă posibilitatea de a structura limbajul de asamblare după rigorile utilizatorilor.

Se consideră o matrice a frecvenţelor F cu elemente f_{ij} care au semnificaţia: număr de apariţii ale instrucţiunii A_i cu utilizarea resursei S_j. Prin definirea unui sistem de ponderi adecvat, se procedează la agregarea informaţiilor, obţinându-se o măsură a intensităţii întrebuinţării unei instrucţiuni într-un context dat (operatori utilizaţi). Un algoritm de clasificare conduce la identificarea unei noi tipologii de instrucţiuni.

Structura internă a instrucţiunii este rezultatul natural al utilizării. Se por-

neşte la proiectarea limbajului de asamblare de la cerinţele reale ale programatorilor. Se spune că se utilizează un design corectiv, întrucât clasele se obţin pornind de la programe scrise într-un limbaj existent. Noul limbaj de asamblare este rezultatul introducerii de corecţii la un limbaj existent. Pentru ca rezultatele să fie semnificative este necesar ca eşantionul de programe cu care s-a efectuat constituirea matricei F să fie suficient de mare şi să cuprindă o diversitate de programe care să acopere multitudinea de aplicaţii care se dezvoltă în limbaje de asamblare.

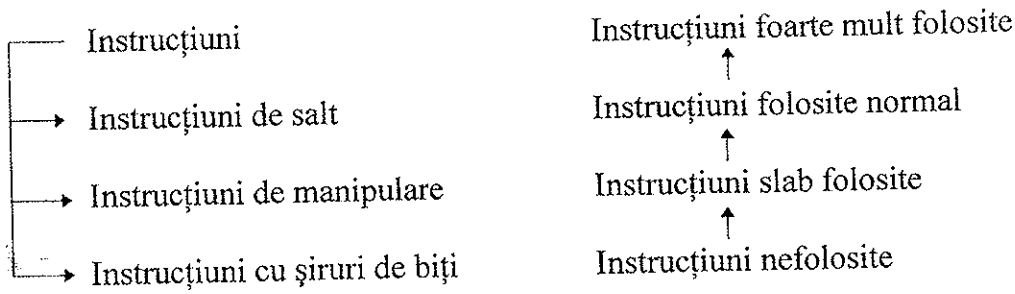

Fig. 2. Migrarea instrucţiunilor

Clasele de instrucţiuni obţinute după frecvenţele de întrebuinţare în programe vor avea asociate coduri ale operaţiilor care din structură să permită interpretări diferite.
Dacă de exemplu, instrucţiunea **inc ax** are frecvenţa de aparitie cea mai mare, i se va asocia codul de operaţie format numai din zerouri.
Dacă se optează pentru un limbaj de asamblare cu codul operaţiei format din 8 biţi, zerourile nesemnificative ca număr permit crearea de grupe de instrucţiuni, fără o acoperire consecutivă a submulţimilor formate din codurile consecutive.
În ipoteza că în clasa de instruc-ţiuni foarte frecvent întâlnite se află 18 elemente, reprezentabile pe cinci poziţii binare, codurile acestei clase vor avea trei zerouri în faţă (000xxxxx). Din cele 32 de combinaţii binare cu cinci poziţii sunt utilizate numai 18.
Într-o primă etapă se realizează punerea în corespondenţă a cla-selor cu codurile. În cazul în care rămân instrucţiuni cărora nu li se asociază coduri datorită epuizării celor 256 de combinaţii binare, se trece la rafinarea claselor.
Rafinarea este un procedeu complex care vizează fie reducerea numărului de clase, fie translaţia de instrucţiuni de la o clasă la alta, obţinându-se o altă repartizare a instrucţiunilor în clase, ca în figura 3.

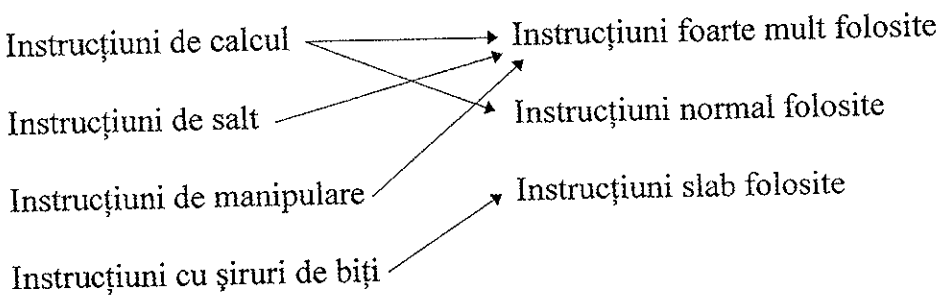

Fig. 3. Restrângerea resurselor

Dacă prin restrângerea claselor sau prin translaţie s-a obţinut creşterea numărului de elemente de la 18 la 31 în clasa instrucţiunilor cel mai frecvent folosite şi dacă s-a obţinut încadrarea instrucţiunilor prin punere în corespondenţă cu

coduri binare şi de opt poziţii, se poate trece la dezvoltarea în continuare a structurii interne a instrucţiunilor limbajului de asamblare.
Un astfel de limbaj prezintă particularitatea ca structurarea instrucţiunilor începe chiar cu codul operatiei. Este mai corect să se vorbească de codul operaţiei-operand, întrucât frecvenţele de apariţie ţin seama de operaţie şi de operanzii utilizaţi.
Aparent un astfel de limbaj nu mai prezintă regularităţile întâlnite la celelalte limbaje de asamblare.
Structurarea în continuare a instrucţiunilor se efectuează depinzând strict de alcătuirea claselor. Tipurile de adresare, modul de parcurgere (stânga/ dreapta), expresiile de adresare, se vor regăsi sau nu la fiecare clasă, după cum rezultă din elementele care o alcătuiesc. Cifrele semnificative de pe poziţiile zero, unu sau doi vor marca fiecare tip structural. Chiar dacă se înregistrează "întreruperi" în secvenţele de atribuire a codurilor, clasele neavând exact 32 de elemente, codificarea aceasta a instrucţiunilor ţine seama de particularităţile de utilizare ale limbajului.

Stabilirea numărului de registre

Limbajele de asamblare consideră registrele ca resurse. Se definesc instrucţiuni în care se utilizează numai anumite registre, ceea ce restrictionează foarte mult manipularea datelor. Înseamnă că registrele nu au funcţiuni identice, unele dintre ele fiind "dedicate".
Astfel, în cazul limbajului de asamblare definit pentru microprocesoarele x86, registrul **AX** este "dedicat pentru numeroase operaţii (ajustări, înmulţiri, împărţiri) şi este destinaţie pentru multe operaţii. Problematica pe care o are designerul este de a stabili numărul optim de registre pe care să le gestioneze eficient (statistic) programatorii. Acest "optim" este rezultatul unui proces de definire "la birou" a secvenţelor în diferite ipoteze cu luarea în considerare a numărului de cicluri pe care le determină fiecare soluţie dată.
Se consideră un lot de probleme frecvent rezolvate in limbaj de asamblare P_1, P_2, ... P_m. Se vor scrie programe in ipoteza în care limbajul este definit cu un singur registru de lucru. Se va observa abundenţa de instrucţiuni **mov** pentru stocare rezultate intermediare şi pentru iniţializari. Scriind un program de analiză a celor m programe care oferă soluţii pentru problemele P_i, i=1,2,..,m, se evaluează volumul de prelucrări exprimat ca număr de cicluri maşină.
Tot astfel se procedează şi pentru definirile de limbaje de asamblare în care există două, trei sau mai multe registre. Numărul de registre determină şi implementările de structuri fundamentale prin intermediul expresiilor de adresare. De exemplu, registrul index permite atât implementarea structurii de dată masiv, cât şi definirea structurii repetitive.
Utilizarea indicatorului număr de cicluri maşină, conduce la omogenizarea de rezultate, chiar dacă se pierd anumite informaţii. El permite agregări şi manipularea cu medii aritmetice şi dispersii.
Dacă se consideră mai multe loturi de probleme, este posibilă analiza stabilităţii limbajului de asamblare, tot din punct de vedere statistic. În cazul în care s-a obţinut un grad de stabilitate corespunzător, se poate trece la alegerea numărului eficient de registre în raport cu criteriul reducerii numărului de cicluri maşină.
Cu cât lotul problemelor este mai vast, există posibilitatea de a acoperi o gamă mare de aplicaţii şi de a face un studiu mai complet asupra comportamentului limbajului de asamblare. Indiferent de mărimea lotului, este riscant să se vorbească despre optimizarea limbajului sau de stabilirea numărului op-

tim de registre de lucru, "dedicate" sau nu. Oricum atributul de optim se va referi la lotul de programe rezultat, orice extensie fiind riscantă dacă nu s-a studiat suficient reprezentativitatea acestui lot.
Cercetările efectuate până în pre-zent pe un lot de 60 de programe cu acelaşi grad de complexitate pune in evidenţa eficienţa unui limbaj de asamblare cu două registre acumulator (împerecheate corespunzător pentru a se face identificarea corectă a operanzilor şi pentru eliminarea ambiguităţii în cazul ajustărilor). De asemenea, tipurile de expresii de adresă ce intervin în registre pot fi dezvoltate pentru a introduce nivele de indirectare de ordin superior.

Concluzii

Designul limbajului de programare cu luarea în considerare a finalităţii, scrierea de programe, determină proiectarea unui limbaj pentru utilizator. Numeroase dificultăţi care apar în asamblare, la generarea formei interne a instrucţiunilor, sunt probleme independente de programator. Odată rezolvate corect, programul asamblor va opera asupra unei mulţimi de programe în creştere, propagând efectele pozitive ale limbajului.
Dacă la proiectare sunt luate în considerare şi elemente de compresie a programelor in cod maşină, limbajul de asamblare oferă o trăsătură benefică tot la nivelul utilizatorilor finali.
Preocupările de design pentru limbaje de asamblare capătă acum o nouă caracteristică, aceea de a fi deschis spre utilizatori. Opţiunea spre neomogenitate de tratare a instrucţiunilor nu determină complicaţii la nivelul utilizatorilor. La nivelul celor care implementează limbajul de asamblare, fiecare neomogenitate se traduce în modalitate distinctă de tratare. Diversităţii de tipuri de instrucţiuni îi va corespunde o creştere a complexităţii programului asamblor.
Pentru a obţine rezultate cu nivel de stabilitate ridicat este necesară crearea unei baze de programe scrise în limbaj de asamblare care să includă cât mai multe stiluri de programe şi cât mai multe tipuri de probleme rezolvate.
Ca şi în cazul altor limbaje, designul limbajelor de asamblare ia în considerare menţinerea unui nivel ridicat al ortogonalităţii instrucţiunilor. Studiul efectuat acum a presupus ortogonalitatea deja existentă a instrucţiunilor din limbajul de asamblare asociat microprocesoarelor x86, fără a se proceda la definirea de noi instrucţiuni sau noi tipuri de expresii de adresare. Dacă se vor efectua în viitor şi aceste modificări, designul limbajului de asamblare capătă un nivel de profunzime mult mai accentuat, influenţând portabilitatea limbajului, în sensul reducerii.

Bibliografie

1. Irina Atanasiu, Alexandru Pănoiu - *Microprocesoarele 8086, 80286, 80386. Programarea în limbaj de asamblare*, Editura TEORA, Bucureşti 1992.
2. Penn Brumm, Don Brumm - *80386 Assembly Language*, TAB Professional and Reference Books, Blue Ridge Summit, 1988.
3. Patrick Cohen - *Le Microprocesseur PENTIUM. Architec-ture et programation,* Armand Colin, Paris, 1994.
4. Len Dorfman - *Structured Assembly Language*, Windcrest Books,Blue Ridge Summit, 1990.
5. Julio Sanchez, Maria P. Canton - *Numerical Programming the 386, 486 & PENTIUM*, McGraw-Hill Inc., New York, 1995.
6. Sen Cuo Ro, Sheau Chuen - *i386 / i486 Advanced Program-ming*, Van Nostrand Reinhold, San Jose, 1993.