# Optimizarea programelor assembler

Prof.dr. Ion IVAN
Catedra de Informatică Economică, A.S.E. Bucureşti
Asist.ing. Cristian CODREANU
Catedra de ACT, Academia Tehnică Militară

*Optimizarea programelor reprezintă una din direcţiile spre care se concentrează producătorul de software. Există numeroase aspecte ale optimizării programelor: optimizarea timpului de execuţie, optimizarea dimensiunii codului, optimizarea textului sursă etc. Toate aceste criterii de optimizare, considerate cu diferite ponderi, contribuie la definirea conceptului de optimizare a unui program. Compilatoarele pentru limbajele evoluate (C, C++, Pascal etc) conţin facilităţi de generare de cod executabil optimizat în timp, dimensiune a codului sau combinaţii între acestea. Datorită specificaţiilor de limbaj, la nivel de asamblare nu există asemenea facilităţi încorporate în compilatoare, de aceea aspectele de optimizare revin în exclusivitate programatorului. În materialul de faţă am tratat problema optimizării timpului de execuţie, care poate surveni uneori în defavoarea dimensiunii codului. Am considerat această abordare deoarece la arhitecturile actuale, memoria nu mai este, în general, o resursa critică. Resursa critică devine timpul, mai ales la aplicaţiile de prelucrare în timp real, la prelucrări grafice, la jocuri etc.*

***Cuvinte cheie:*** *optimizare, criterii de optim, ciclu maşină, limbaj de asamblare, tipuri de date.*

## Criterii de optim

Se consideră un număr de programe destinate rezolvării aceleiaşi probleme. Dintre toate, unul singur conduce la obţinerea rezultatului în timpul cel mai scurt. În raport cu criteriul *timpul de rulare*, acest program este optim, raportat la mulţimea de programe disponibile la un moment dat.

Întrucât nu se poate vorbi de program optim în general, în continuare, prin program optim vom înţelege acel program care are comportamentul cel mai bun, chiar şi numai statistic, pe baza unui eşantion de probleme rezolvate. Introducând sau eliminând programe sau seturi de probleme de test, este posibil ca programul optim să fie de fiecare dată altul.

Criteriile după care se ierarhizează programele sunt: durata de timp necesară rezolvării unei probleme; dimensiunea problemei de rezolvat; necesarul de memorie internă; precizia rezultatelor; nivelul de generalitate al problemei ce poate fi acceptată; costul rulării programului; nivelul unei caracteristici de calitate; lungimea textului programului.

Criteriile sunt contradictorii, ceea ce determină ca la obţinerea unor avantaje din punct de vedere ale unui criteriu să se estimeze care sunt efectele negative antre-nate pentru celelalte.

Programele scrise în limbaj de asamblare au particularităţi impuse de faptul ca programatorul gestionează toate resursele şi are posibilitatea să aleagă dintre mai multe variante pentru orice construcţie pe care o realizează.

Toate criteriile sunt importante şi ideal este să fie construit programul care le îndeplineşte pe toate. Se observă că lungimea textului, dimensiunea problemei, precizia rezultatelor, calitatea şi necesarul de memorie sunt strâns legate de durata execuţiei.

Estimarea duratei de rulare a unui program scris într-un limbaj evoluat este dificilă. Optimizarea timpului de execuţie al unui program scris într-un astfel de limbaj ţine de caracteristicile limbajului şi ale compilatorului. De asemenea, compilatoarele evoluate suportă opţiuni pentru optimizarea dimensiunii codului sau a duratei de execuţie, dar aceste optimizări, sunt, deocamdată, relative, deoarece se fac la nivel local şi nu ţin cont de evoluţia în perspectivă a

programului sau de stările prin care s-a trecut anterior.
Dacă în cazul limbajelor evoluate complexitatea instrucțiunilor face dificilă estimarea duratei, pentru limbajele de asamblare durata necesară execuției unei instrucțiuni este dată cu precizie prin numărul de cicluri mașină.
Dacă programatorul face opțiuni din aproape în aproape asupra instrucțiunilor selectate, căutând să reducă numărul de cicluri, va obține un program bun în raport cu durata de execuție. Optimalitatea programului trebuie demonstrată prin compararea cu alte programe sau chiar cu alte variante de secvențe destinate respectivului program.
Optimizarea timpului de execuție se pretează la sistemele cu capacitate mare de memorie care necesită prelucrarea unei cantități de informație foarte mare în timp critic (de exemplu afișarea unor imagini dinamice pe un display grafic de rezoluție mare).
În cazul unor sisteme cu resursă de memorie de cod redusă și care procesează un volum de date mai modest, se pune problema optimizarii dimensiunii codului (de exemplu procesoare pentru urmarirea unor instalații cu parametrii lent variabili în timp).

**Cicluri mașină**

În general, mecanismele au o componentă principală care dezvoltă o operație de bază, esențială pentru funcția pentru care au fost realizate. Strungul are un ax a cărui rotație determină antrenarea celorlalte subansamble. Operația de bază este rotatia completă. În cazul altor mașini, operația de bază este efectuarea completă a unei deplasări du-tevino. Ciclul mașină este fie o rotație completă, fie o mișcare rectilinie dus-întors.
Calculatoarele sunt înzestrate cu circuite de tact care definesc coordonatele în timp ale operațiilor ce se pot efectua. Astfel, un calculator are un ciclu mașină de durată dată prin împarțirea unei microsecunde la numărul de MHz ai ceasului său.
Un calculator 80286/10 MHz are un ciclu de 100 nanosecunde; un calculator 80286/ 5 MHz are un ciclu de 200 nanosecunde; un calculator 80486/ 50MHz are un ciclu de 20 nanosecunde.
Când se proiectează un limbaj de asamblare se stabilesc implementările fizice ale instrucțiunilor, precizându-se cu exactitate numărul de cicluri necesar efectuării lor.
Astfel, când operanzii sunt specificați, numărul de cicluri e fix. De exemplu, instrucțiunea *mov ax,100* se execută în 4 cicluri (8088), iar instrucțiunea *mov ax,cx* se execută în 2 cicluri (8088).
Când operanzii sunt variabile a căror poziție se calculează prin evaluarea unei expresii, iar poziția lor efectivă este segmentul precizat sau adresa este un număr impar deși se lucrează la nivel de cuvânt, numărul de cicluri se majorează.
Instrucțiunea *mov ax,SUMA* se execută în 8+6 cicluri, expresia offset-ului necesită pentru calculul adresei 6 cicluri. Când adresarea operandului este indirectă folosind fie registrul index, fie registrul de bază, sunt necesare încă 5 cicluri. Astfel, instrucțiunea *mov ax,[bx]* se execută în 8+5 cicluri.
Dacă expresia pentru adresarea operandului conține deplasare și registru index sau registru de bază, sunt necesare încă 9 cicluri (instrucțiunea *mov ax,VECTOR[bx]* se execută în 8+9 cicluri).
Expresia de calcul al adresei în care apar registrul de bază și registrul index necesită încă 7 cicluri, iar expresia care determină adresa pe bază de deplasare, registru index și registru de bază necesită încă 11 cicluri.
Instrucțiunile:
*mov ax,bx[si]*
*mov ax,MATRICE[bx][di]*
necesită 8+7, respectiv, 8+11 cicluri. Când sunt definite instrucțiunile se specifică toți factorii care influențează numărul de cicluri.
La prezentarea instrucțiunilor, în tabele există o coloană destinată numărului de cicluri și a variabilității.
Dacă se consideră factorii $f_1, f_2, ..., f_K$ care influențează numărul de cicluri pentru operații, după ce se determină exact contribuția fiecărui factor, stabilind coeficienții $c_1$, $c_2$, ..., $c_K$, modelul de calcul al numărului de cicluri pentru instrucțiunea Ij este dat de ecuația:

$NCI_j = c_{0j} + Sc_i \bullet x_i$, unde $c_{0j}$ reprezintă numărul de cicluri fixat pentru instrucţiunea $I_j$; $x_i$ reprezintă variabila booleană cu valoarea 1 dacă factorul $f_i$ este prezent şi 0 în caz contrar.

Dacă un factor vizează încărcarea de segment, coeficientul sau asociat este cu valoare 2. Dacă factorul de aliniere a adresei la nivel de cuvânt este prezent, când operandul are adresa efectivă un număr impar, "forţarea" la baitul următor cu adresa număr par presupune adaugarea a 4 cicluri. Instrucţiunea *mov SUMA,ax* se execută în 20 cicluri, dacă 8 cicluri se datoresc faptului că un operand este în registrul AX iar celălalt este în memorie, 6 cicluri rezultă din calculul adresei de 16 biţi ai offset-ului, 2 cicluri provin din încărcarea segmentului unde se află variabila sumă, aşa cum rezultă din definirea ei şi din punerea în corespondenţă a segmentelor cu registrele de segment în directiva ASSUME şi 4 cicluri sunt necesare pentru a forţa încărcarea unei adrese pare, pentru că variabila SUMA este definită la o adresăa impară.

Numărul de cicluri este influenţat de distanţa la care se face saltul necondiţionat, dacă se lucrează în mod protejat sau nu, cât de lungă este zona de memorie care se transferă, dacă se lucrează în cadrul aceluiaşi segment sau între segmente diferite. Rezultatele pot fi scurte, în apropiere sau la distanţă.

Modelul de calcul al numărului de cicluri se obţine din tabelele de descriere a instrucţiunilor cât şi din prezentarea fiecărei instrucţiuni în parte.

Menţinând clasificarea instrucţiunilor după numărul de cicluri, se preferă acele instrucţiuni care nu depind de amplasamentul operanzilor sau se preferă acele moduri de adresare care generează sistematic un număr redus de cicluri.

Instrucţiunea de salt necondiţionat din secvenţa:

```
        .....
alfa:   mov ax, 5
        .....
        jmp alfa
```

având distanţa de tip scurt necesită 1 ciclu, în timp ce un salt necondiţionat intersegmente, folosind adresarea indirectă ca în instrucţiunea: *jmp beta [bx][di]* necesită 16 + 7 + 2 cicluri ($c_{0j} = 16, c_1 = 7, c_2 = 2$).

În documentaţii se fac referiri detaliate asupra expresiilor de adresă care intervin în calculul ciclurilor ce se asociază fiecarei variabile de instrucţiune.

Esta important să se cunoască efectele contradictorii ale opţiunilor la scrierea de programe în limbaj de asamblare. De regulă se obişnuieşte obţinerea unui text compact prin definirea şi apelarea de proceduri. Se minimizează lungimea textului dar fiecare apel de procedură înseamnă o instrucţiune CALL. Dacă procedura este referită prin adresă în cadrul aceluiaşi segment, numărul de cicluri necesar execuţiei este 11. Dacă se lucrează în cadrul aceluiaşi segment dar adresele sunt pe 4 baiţi, pentru un apel de procedura sunt necesare 26 cicluri. Gestionarea prin apeluri de proceduri a task-urilor sau lucrul în mod privilegiat conduce la un necesar de cicluri de 177, respectiv, 90+4*x cicluri (x reprezintă numărul de parametri). Orice procedură presupune revenirea în funcţia apelatoare folosind instrucţiunea RET, care consumă între 11 şi 55 cicluri, de asemenea, depinzând de tipul formulei de calcul a adresei instrucţiunii ce urmează lui CALL. Dacă această adresă este stocată cu instrucţiunea POP/PUSH înainte de apel şi procedura este în acelaşi segment în care se afla funcţia apelatoare, sunt necesare 11 cicluri. Dacă tipul de pointer este FAR sunt necesari 55 cicluri. Deşi lucrând cu proceduri s-a obţinut reducerea lungimii textului, numărul de cicluri adăugat este de cel puţin 7 + 11 sau de cel mult 185 + 55 cicluri.

**Volumul de operaţii**

Un program scris în limbaj de asamblare conţine linii sursă care la asamblare genereaza instrucţiuni executabile (masina) şi linii sursă ce definesc contextul de alocare resurse şi de iniţializare. Volumul de operaţii se referă la instrucţiunile executabile.

Procedura:

```
aduna: PROC
        push   bp
```

```
        mov     bp,sp
        mov     ax,[bp+4]
        add     ax,[bp+6]
        add     ax,[bp+8]
        pop     bp
        ret
aduna   ENDP
```

conține șapte instrucțiuni executabile, fiecare se execută o singura dată. Volumul de operații, ca număr de instrucțiuni ce se execută, în acest caz este chiar 7.

Procedura:

```
generare:       PROC FAR
        push    bp
        mov     bp,sp
        push    cx
        push    bx
        mov     cx,[bp+6]
        mov     bx,[bp+8]
        mov     ax,0
ciclu:  add     al,[bx]
        adc     ah,0
        inc     bx
        loop    ciclu
        pop     cx
        pop     bx
        pop     bp
        ret
generare        ENDP
```

conține 14 instrucțiuni executabile, dintre care:

```
ciclu:  add     al,[bx]
        adc     ah,0
        inc     bx
        loop    ciclu
```

se execută de un număr de ori dependent de conținutul registrului *cx*. Presupunând ca registrul *cx* conține o valoare notată generic **n**, volumul de operații executate la apelul procedurii generale va fi dat de relația: $V = k_0 + n_1 \bullet k_1 + \ldots + n_p \bullet k_p$, unde $k_0$ reprezintă numărul instrucțiunilor executate o singura dată; $n_i$ reprezintă numărul de repetări ale buclei de program i; $k_i$ reprezintă numărul de instrucțiuni cuprinse în bucla de program i. Valoarea sa este V=11+4*n.

Există proceduri în care apar comparații și se efectuează selecții ale secvențelor. Se notează $P_j$ probabilitatea ca o condiție $C_j$ să fie îndeplinită, caz în care se execută o secvență având volumul $V_f$. Numărul repetărilor testului pentru condiția $C_j$ este $n_j$.

Volumul operațiilor este dat de relația:

$$V = \left(P_j \bullet V_t + \left(1 - P_j\right) \bullet V_f\right) \bullet n_j.$$

Volumul de operații apare ca un număr mediu de operații care se vor executa în timp. Procedura:

```
Trans   PROC NEAR
        mov     al,[si]
        cmp     al,0
        jz      final
        cmp     al,'a'
        jb      urmat
        cmp     al,'z'
        ja      urmat
        and     al,NOT 20h
        mov     [si],al
urmat:  inc     si
        jmp     trans
final:  ret
trans   ENDP
```

convertește un șir de lungine M caractere, terminat cu zero, într-un șir format numai din litere mari. Volumul de operații depinde de structura șirului de caractere. În structura textelor din limba română 6% dintre caractere sunt litere mari. Procedura trans va efectua transformarea *and al,NOT 20h* în 94% din cazuri. La stânga literelor mici se află 3,3 % dintre caracterele unui text, iar la dreapta restul de 0,7 %, asa cum rezultă din observații parțiale.

Pentru textul de lungime M caractere, delimitat prin zero, instrucțiunile procedurii trans se execută în medie de un număr de ori indicat în tabelul 1.

Tabelul 1

| Instrucțiunea | Număr mediu de repetări |
|---|---|
| mov al,[si] | M |
| cmp al,0 | M |
| jz final | M |
| cmp al,'a' | M |
| jb urmat | 0.33*M |
| cmp al,'z' | 0.967*M |
| ja urmat | 0.967*M |
| and al,NOT 20h | 0.96*M |
| mov [si],al | 0.96*M |
| inc si | M |
| jmp trans | M |
| ret | 1 |

Volumul mediu de operații executabile la un apel al procedurii trans este
V = M*(6+0.967+0.967+0.96+0.96+0.33)+1 operații.
Conceptul de operație este general și se observă de la început ca posibilitatea de a compara operațiile este dificilă datorită diferenței de complexitate pe care fiecare operație o induce. Un transfer de date, intuitiv este mai simplu decât o înmulțire, iar apelul unei proceduri este mai complex decât o implementare.
Se acceptă ipoteza conform căreia complexitatea operațiilor este strâns legată de numărul de cicluri mașină asociate. Diversității de instrucțiuni îi corespunde o multitudine de numere de cicluri. Mai mult, tipurile de adresare modifică numărul de cicluri pentru fiecare instrucțiune. Instrucțiunile care manipulează un volum redus de informație sau au operanzi prefixați sunt puse în corespondență cu un număr redus de cicluri. Instrucțiunile care au adrese ce sunt calculate după formule complexe, care impun regăsiri, necesită un număr de cicluri mașină superior.
Pentru a reflecta mai exact efortul de execuție, volumul programului se va exprima ca număr de cicluri mașină. Astfel, secvența:

```
.....
mov   ds,ax        ; 2 cicluri mașină
xor   ax,ax        ; 3 cicluri mașină
inc   ax           ; 2 cicluri mașină
cmp   ax,20        ; 3 cicluri mașină
.....
```

are un volum V = 10 cicluri, rezultat ca sumă a ciclurilor asociate instrucțiunilor care o alcătuiesc.
Pentru procedura:

```
strlen  PROC NEAR
        cld                ; 2 cicluri
        push   cx          ; 10 cicluri
        mov    cx,0ffffh   ; 4 cicluri
        mov    al,0        ; 4 cicluri
        repne  scasb       ; n*(6+15) cicluri
        jne    eroare ; 8 cicluri pentru salt, 4 cicluri fara salt
        mov    ax,0ffffh   ; 4 cicluri
        sub    ax,cx       ; 3 cicluri
        dec    ax          ; 2 cicluri
        dec    di          ; 2 cicluri
        dec    di          ; 2 cicluri
        jmp    SHORT final ; 1 ciclu
eroare: mov    di,0        ; 4 cicluri
        mov    es,di       ; 2 cicluri
final:  pop    cx          ; 8 cicluri
        ret                ; 8 cicluri
strlen ENDP
```

volumul de operații exprimat în cicluri este $V_1 = 50 + 21 \cdot n$ în caz de eroare sau $V_2 = 24 + 21 \cdot n + 30$ cicluri.

**Secvențe echivalente**

Dacă se urmărește optimizarea programelor scrise în limbaj de asamblare reducând volumul operațiilor, în primul rând se caută folosirea de secvențe echivalente care se execută într-un număr mai redus de cicluri.

Inițializarea unui registru se realizează în moduri diferite. Din secvența:

```
mov   ax,0         ; 4 cicluri
sub   ax,ax        ; 3 cicluri
xor   ax,ax        ; 3 cicluri
```

rezultă necesitatea de a utiliza una din ultimele două variante, deși prima instrucțiune este mai sugestivă.
La tipul de adresare indexat este necesară incrementarea unui registru cu o rație egală cu lungimea zonei de memorie care este

referită. Dacă raţia este o unitate; din secvenţa:

```
add    si,1          ; 4 cicluri
inc    si            ; 2 cicluri
```

rezultă că este avantajoasă utilizarea instrucţiunii inc şi, efectul fiind major mai ales pentru faptul că referirea este proprie unei secvenţe executate repetitiv.

Dacă raţia cu care se modifică registrul index este un număr oarecare, repetarea instrucţiunii inc registru este ineficientă. Se optează spre una din variantele din secvenţa:

```
add    si,57   ; ratia este 54, 4 cicluri
add    si,bx   ; 3 cicluri, registrul bx a fost
initializat mov bx,57
add    si,ratia; 16+6 cicluri
```

De cele mai multe ori nu este posibilă alocarea unui registru pentru memorarea raţiei şi se defineşte o constantă simbolică (RATIA EQU 57) dupa care incrementarea este realizată prin *add si,RATIA*.

Lucrurile devin mult mai simple dacă se pune problema alegerii modalităţii de a înmulţi un număr cu 2k sau de a-l impărţi prin 2k. Pentru înmulţirea numărului aflat în registrul ax cu 32 se alege secvenţa:

```
mov    cl,5
mov    ax,79
sal    ax,cl   ; 8+4*5 cicluri
```

întrucat secvenţa:

```
mov    bl,32
mov    al,79
cbw    b       ; 2 cicluri
mul bl         ; 71 cicluri
```

necesită mai multe cicluri.

În cazul în care o procedură este apelată de un număr mare de ori, este preferabil să se includă textul ei în locul instrucţiunii *call*. Chiar dacă lungimea textului sursă creşte mult se obţine o reducere a numărului de cicluri generate de fiecare apel şi de fiecare revenire în secvenţa apelatoare.

### Alegerea tipului de dată

În limbajele evoluate lucrul cu diferite tipuri de date este netransparent, programatorul neavând la dispoziţie resursele antrenate. În programul scris în limbaj de asamblare lucrul cu date codificate binar înseamnă a defini variabile la nivel de bait sau pe cuvânt. Odată definită o variabilă la nivel de bait, se vor utiliza registrele al, bl, cl, dl, ah, bh, ch, dh. Lucrul la nivel de cuvânt înseamna lucrul cu registrele ax, bx, cx, dx. În ambele cazuri se are în vedere testarea indicatorului de condiţie CF pentru a gestiona corect rezultatele. Se pot folosi instrucţiunile setului, ca şi cum limbajul de asamblare este proiectat preponderent pentru a lucra cu numere codificate binar.

Dacă se doreşte să se lucreze în aritmetică zecimală, mai întâi se vor construi toate procedurile care operează în această aritmetică: transferuri între operanzi, adunări, scăderi, înmulţiri, impărţiri, deplasări, alinieri. Procedurile vor fi cu un grad ridicat de generalitate. Evaluarea unei expresii va consta mai întâi în demontarea ei în paşi elementari; se pregătesc parametrii şi se vor apela funcţiile pentru efectuarea operaţiilor. Programatorul va gestiona şi rezultatele elementare şi pe cele intermediare.

Dacă se lucrează în virgula mobilă, fiecare limbaj de asamblare admite un set de instrucţiuni specifice acestui tip de date şi chiar registre specializate.

În procesul de optimizare a programelor scrise în limbaj de asamblare, alegerea tipului de date este esenţială pentru efortul de programare în primul rând. Lucrul cu date de tip real presupune utilizarea resurselor unui coprocesor, iar aritmetica zecimală e folosită când operanzii au un număr foarte mare de cifre şi sunt întregi.

Neomogenitatea operanzilor este aproape exclusă în programele scrise în limbaje de asamblare.

Dacă un operand este definit pe un bait şi participă la evaluarea unei expresii în care un rezultat intermediar este în registrul ax, la nivel de cuvânt, este necesară o conversie de la bait la cuvânt pe care o asigură programatorul. În programele scrise în limbaj de asamblare construcţiile implicite sunt foarte rare. Secvenţa:

```
mov    bx,ax
mov    al,operand_bait
cbw
add    ax,bx
```

ilustrează faptul că programatorul gestionează şi rezultatele intermediare. Problema

devine mai complicată când un operand este întreg iar altul este zecimal împachetat. Mai întâi se ia decizia cum se va lucra în continuare. Dacă se va continua lucrul în zecimal împachetat, operandul binar va fi convertit apelând la o procedură la forma de reprezentare zecimal împachetată. În continuare se vor apela proceduri de lucru pentru zecimal împachetat.

Dacă opțiunea este de a lucra în binar, operandul zecimal împachetat va fi convertit în binar și se vor folosi în continuare instrucțiunile setului definit pentru limbajul de asamblare considerat.

În cazul în care este necesar să se lucreze în virgulă mobilă cei doi operanzi vor fi convertiți folosind proceduri speciale și cu instrucțiunile de virgulă mobilă se va continua lucrul.

Lucrul cu operanzi neomogeni impune existența unei multitudini de proceduri de conversie care să acopere totalitatea cerințelor de omogenizare a tipurilor. După omogenizare, programatorul va folosi numai instrucțiunile sau procedurile specifice tipului pentru care s-a hotărât să continue lucrul.

Optimizarea programului este obținută în această fază prin numărul de proceduri de conversie ce se apelează și prin procedurile de operații. Un programator cu experiență va ști când să nu folosească aritmetica binară, problematica alegerii tipului cel mai potrivit fiind destul de rară. Programatorul își definește din start operanzi omogeni și exclude efectuarea de conversii tocmai pentru ca neomogenitatea este transparenta în programele assembler prin creșterea lungimii datorate secvențelor suplimentare specifice omogenizarii.

Chiar dacă procedurile de conversie sunt rezultatul unui proces de optimizare, adăugarea lor la un program determină creșterea volumului de operații.

## Eliminarea subexpresiilor comune

Limbajele evoluate încearcă să redea forma algebrică a expresiilor. De aceea programatorul își pune distinct problema eliminarii subexpresiilor comune. Expresia:

$E = (a+b+c) * (a+b+c-d) + (c^2+a+b+c) / (a+b+c)$

va fi scrisă de cele mai multe ori direct cum apare și în rare cazuri se va calcula E1=a+b+c după care se va calcula: $E = E1 * (E1-d) + (c^2+E1) / E1$.

Când se scrie programul în assembler, demontarea expresiei și reașezarea rezultatelor intermediare pentru a respecta prioritățile operatorilor îl determină pe programator să urmărească simultan și modalități de a reduce lungimea secvenței. Eliminarea subexpresiilor comune apare ca o necesitate firească pentru programator. Mai mult, el va căuta să gestioneze cu grijă rezultatele intermediare, lungimile zonelor de memorie asociate lor, pentru a nu deteriora omogenitatea operanzilor. Secvența:

```
mov   ax,a        ;5
add   ax,b        ;7
add   ax,c        ;7
mov   E1,ax       ;3
sub   ax,d        ;7
mul   E1          ;24
mov   prod1,ax    ;3
mov   prod1+2,dx  ;3
mov   ax,c        ;5
mul   c           ;24
add   ax,E1       ;7
adc   dx,0        ;7
div   E1          ;25
xor   dx,dx       ;2
add   ax,prod1    ;7
adc   dx,prod1+2  ;7
mov   E,ax        ;3
mov   E+2,dx;3 total 149 cicluri (80286)
```

ține seama de ceea ce conțin registrele dupa efectuarea operațiilor și utilizează acest conținut. Este puțin probabil ca programatorul să repete de patru ori secvența:

```
mov   ax,a
add   ax,b
add   ax,c
```

în programul său, fără cel puțin să se gândească la scrierea unei macrodefiniții pentru a-și ușura efortul de a scrie textuil sursă.

## Gestionarea corectă a invarianților

În secvențele care se execută repetitiv din eroare sunt introduse inițializări ale variabi-

lelor care alterează rezultatele finale. Secvența:

```
        mov   cx,22
ciclu:  mov   ax,0
        mov   si,0
        add   ax,x[si]
        inc   si
        loop  ciclu
        mov   total,ax
```

va conduce nu la însumarea elementelor unui vector, ci la însumarea primei componente a vectorului, numai. Cele două inițializări de registre, invarianții din secvență, sunt scoși în afară, obținând variabilitatea cerută de orice structură repetitivă. Secvența:

```
        mov   ax,0
        mov   si,0
        mov   cx,22
ciclu:  add   al,xîsiş
        inc   si
        loop  ciclu
```

realizează obiectivul propus.

**Regruparea ciclurilor**

În cazul secvențelor repetitive în care rezultatele sunt independente, numărul de repetări este identic, secvențele se pot regrupa reducând numărul operațiilor induse de testul variabilei de control și de modificarea acesteia. Secvența:

```
mov   cx,37          ; 4 cicluri
xor   ax,ax          ; 3 cicluri
xor   si,si          ; 3 cicluri
ciclux:add   al,xîsiş ;18 cicluri * 37
inc   si             ; 2 cicluri * 37
loop  ciclux         ; 37 * 9 cicluri sau 5
cicluri
mov   suma,ax        ; 14 cicluri
mov   cx,37          ; 4 cicluri
xor   ax,ax          ; 3 cicluri
xor   si,si          ; 3 cicluri
cicluy:add   al,y[si]; 18*37 cicluri
inc   si             ; 2*37 cicluri
loop cicluy          ; 9*37 sau 5 cicluri
mov sumy,ax          ; 14 cicluri
```

se execută într-un volum V=10+29*37+24+10+29*37+14=2204 cicluri. Secvența în care se regrupează ciclurile:

```
mov   cx,37          ; 4 cicluri
xor   si,si          ; 3 cicluri
xor   ax,ax          ; 3 cicluri
xor   bx,bx          ; 3 cicluri
ciclu: add   al,x[si]; 18*37 cicluri
add   bl,x[si]       ; 18*37 cicluri
inc   si             ; 2*37 cicluri
loop ciclu           ;9*37 cicluri 5 cicluri
mov   sumax,ax       ; 14 cicluri
mov   sumay,bx       ; 14 cicluri
```

necesită un volum V=13+37*47+5+28=1785 cicluri (8088).

Dacă numărul de elemente ale unui șir este par se poate înjumătăți numărul de repetări prin calculul a două sume (suma elementelor cu poziție pară și suma elementelor cu poziție impară). La ieșirea din ciclu printr-o însumare se obține rezultatul dorit. Secvența:

```
mov   cx,2*N         ;4 cicluri, N
                     ;constanta simbolica
xor   ax,ax          ; 3 cicluri
xor   si,si          ; 3 cicluri
ciclu: add   ax,x[si]; 18*2*N cicluri
inc   si             ; 2*2*N cicluri .
inc   si             ; 2*2*N cicluri
loop ciclu
                     ; 9*2*N cicluri sau 5
                     ;cicluri
mov   sum,ax         ;14 cicluri
```

necesită un volum de operații V= =29+2*N*31 cicluri.

Prin însumarea separată a elementelor cu poziții pare, respectiv impare, secvența:

```
        mov   cx,N
        xor   ax,ax
        xor   bx,bx
        xor   si,si
        ciclu: add   ax,x[si]
        add   si,2
        add   bx,xîsiş
        add   si,2
        loop ciclu
        add   ax,bx
```

necesită un număr de operații V=13+ 2*N*26+N+8. Comparând cele două volume, rezultă o diferență D=8+9*N cicluri, ceea ce justifică o astfel de regrupare a termenilor din structurile de date omogene.

### Eliminarea secvențelor inconsistente

Uneori în programe se introduc instrucțiuni care anulează efectele operațiilor precedente. Programatorul va elimina acele instrucțiuni care nu corespund cerințelor și distrug rezultate create anterior. Eliminarea de instrucțiuni atrage reducerea volumului de operații.
In secvența urmatoare:

```
mov bx,0        ;1
xor ax,ax       ;2
add ax,2        ;3
mov bx,ax       ;4
```

instrucțiunea 1 este inconsistentă, deoarece registrul bx este apoi modificat (instrucțiunea 4) fară ca valoarea stocată în el anterior să fie folosită. Pentru limbajele evoluate (e.g. C, Pascal) la compilare există posibilitatea ca utilizatorul să fie avertizat asupra variabilelor nefolosite. În asamblare o asemenea analiză la compilare este cvasi-imposibilă, de aceea programatorul trebuie să depisteze asemenea secvențe încă din faza de scriere a codului sursă.

### Eliminarea secvențelor inactive

Programele scrise în limbaje evoluate pot conține secvențe ce nu se activează indiferent de contextul în care se rulează programul. În programele scrise în limbaj de asamblare, pentru a reduce deplasarea operanzilor, la definire aceștia sunt incluși în segmentul program, ca în secvența:

```
.CODE
start:  jmp   alfa
x       dw    10
y       dw    20
z       dw    ?
alfa:   mov   ax,x
add     ax,b
mov     z,ax
mov     ah,4ch
int     21h
END     start
```

Astfel de instrucțiuni sunt frecvente și corecte, poate chiar eficiente, și de aceea este dificil a se identifica secvențele care nu se activează niciodată în execuție.

Există și situații când se construiesc teste în mod eronat, fără a asigura încheierea unui ciclu sau existența cel puțin a unei situații în care se traversează și o altă ramură a structurii alternative.
Expresiile booleene construite în limbajele evoluate pot conduce la evaluare la valori constante indiferent de variațiile operanzilor. Complexitatea expresiilor și manipularea eronată a operatorilor generează secvențe numite cod mort, adică secvențe ce nu se execută niciodată. În programele scrise în limbaj de asamblare astfel de construcții apar ca incorecte relativ ușor, întrucât se identifică invariabilitatea operanzilor.
Secvența:

```
mov   ax,5
cmp   ax,0
jz    alfa
.....
jmp   beta
alfa: .....
beta: nop
```

este interpretată ca generatoare de cod mort dacă este inclusă chiar într-o structură repetitivă, pentru ca atât timp cât ax va conține 5 și se va compara cu valoarea zero, secvența etichetată cu alfa nu se va executa. Volumul de operații nu este influențat dacă se ia în considerare coeficientul zero al probabilității acestei secvențe inactive. Codul mort afectează numai lungimea programului ca număr da baiți ocupați de codul obiect asociat unui text sursă.

### Reacoperirea segmentelor

Segmentele se gestionează de către programator. Directivele de punere în corespondență a segmentelor cu registre de segmente, încărcarea adreselor de segment, sunt elemente la dispoziția programatorului. Instrucțiunile de salt necondiționat se diferențiază după cum destinația este în același segment sau este în alt segment, adresarea fiind directă sau indirectă. Apelul de procedură din același segment are 7 sau 11 cicluri, în timp ce pentru proceduri din alte segmente numărul de cicluri poate fi 13 sau 26 cicluri.

Optimizarea reacoperirii vizează programe complexe, care operează cu structuri de date ce necesită mai multe segmente de date care se încarcă alternativ. Pentru proceduri se va asocia o arborescență pentru a se identifica ce componente se afla încărcate de pe fiecare ramură.

**Alocarea optimă a regiştrilor**

Alocarea regiştrilor este o problemă de construire a compilatoarelor. Aceeaşi secvență se va utiliza în moduri diferite, obținîndu-se de fiecare dată alt număr de cicluri maşină.

Alocarea regiştrilor are ca obiectiv minimizarea numărului de cicluri. Se vor construi compilatoare care realizează alocări de registre şi tipuri de adresări care să conducă la atingerea acestui obiectiv. De exemplu, pentru evaluarea expresiei e=a+b+c se construieşte secvența:

```
mov ax,a        ;5 cicluri
mov bx,b        ;5 cicluri
mov cx,c        ;5 cicluri
add ax,bx       ;2 cicluri
add ax,cx       ;2 cicluri
mov e,ax        ;3 cicluri
```

căreia îi corespund 22 cicluri maşină. Secvența echivalentă:

```
mov ax,a        ;5 cicluri
add ax,b        ;7 cicluri
add ax,c        ;7 cicluri
mov e,ax        ;3 cicluri
```

conține instrucțiuni care totalizează tot 22 cicluri.

Problematica alocării este importantă pentru operanzii reutilizabili din expresii. Astfel, expresia e = ( a + b - c ) * ( a - b + c ) se calculează în secvența:

```
xor dx,dx       ;2 cicluri
mov ax,a        ;5 cicluri
sub ax,b        ;7 cicluri
add ax,c        ;7 cicluri
mov bx,ax       ;2 cicluri
mov ax,a        ;5 cicluri
add ax,b        ;7 cicluri
sub ax,c        ;7 cicluri
mul bx          ;21 cicluri
```

careia îi corespund 63 cicluri. În secvența echivalentă:

```
xor dx,dx       ;2 cicluri
mov bx,b        ;5 cicluri
mov cx,c        ;5 cicluri
mov si,a        ;5 cicluri
mov ax,si       ;2 cicluri
sub ax,bx       ;2 cicluri
add ax,cx       ;2 cicluri
mov bx,ax       ;2 cicluri
mov ax,si       ;2 cicluri
add ax,bx       ;2 cicluri
sub ax,cx       ;2 cicluri
mul bx          ;21 cicluri
```

se obțin 52 cicluri.

Numărul de cicluri necesare pentru execuția unei instrucțiuni cu operanzi din memorie este mai mare decât în cazul în care operanzii s-ar afla în regiştrii. Pentru reducerea timpului de execuție, se va urmări păstrarea rezultatelor intermediare în regiştrii liberi, printr-o alocare optimă a acestora.

Conform acestui principiu, secvența pentru calculul expresiei :

$E=(a+b+c)*(a+b+c-d)+(c^2+a+b+c)/(a+b+c)$

devine, prin utilizarea regiştrilor *bx*, şi si *di*, următoarea:

```
mov   ax,a    ;5 cicluri
add   ax,b    ;7 cicluri
add   ax,c    ;7 cicluri
mov   bx,ax   ;2 cicluri      expresia E1
sub   ax,d    ;7 cicluri
mul   bx      ;21 cicluri
mov   di,ax   ;2 cicluri
mov   si,dx   ;2 cicluri
mov   ax,c    ;5 cicluri
mul   c       ;24 cicluri
add   ax,bx   ;2 cicluri
adc   dx,0    ;7 cicluri
div   bx      ;14 cicluri
xor   dx,dx   ;2 cicluri
add   ax,di   ;2 cicluri
adc   dx,si   ;2 cicluri
mov   E,ax    ;3 cicluri
mov   E+2,dx;3 cicluri      total      117 cicluri
```

Se remarcă utilizarea regiştilor *bx, şi* si *di* pentru păstrarea unor rezultate intermediare. Este clar ca după această secvență, valorile inițiale din aceşti regiştri vor fi modificate. Dacă acestea sunt necesare pentru prelucrări ulterioare, se pot salva, fie în stivă cu

instrucţiunea *push*, fie în variabile din memorie.
Utilizarea regiştrilor se face dupa o analiză a codului pe baza statistică, astfel încât costul salvării şi refacerii regiştrilor folosiţi să fie mai mic decât costul utilizării exclusiv a memoriei pentru salvarea rezultatelor intermediare.

**Concluzii**

Particularităţile limbajelor de programare se regăsesc la optimizare. Se observă că optimizarea programelor scrise în limbaj de asamblare conţine acele elemente ce impun simplitate secvenţelor de program. Criteriile de optim, numeroase la celelalte limbaje de programare, se restrâng, atenţia fiind îndreptată spre minimizarea volumului de operaţii.
Datorită faptului că programele scrise în limbaj de asamblare nu înlocuiesc aplicaţii scrise în limbaje de tip ***C***, ci le presupun, prin dimensiune, prin complexitate, dacă minimizează numărul de cicluri maşină, înseamnă ca s-a realizat optimizare. Programatorul în assembler are multe restricţii de utilizare a registrelor, a instrucţiunilor. S-a făcut deosebire între optimizarea sistemelor de programe şi optimizarea pe textul sursă. De aceea, aplicaţiile în assembler sunt de regulă părţi ce se încorporează în construcţii mult mai complexe. Optimizarea va fi orientată spre viteza de calcul în principal, reducerea lungimii programului, creşterea intensităţii de utilizare a operanzilor trec pe un plan secundar.
Microprocesoarele evoluate sunt proiectate astfel încât ins-trucţiunile sunt executate în *pipeline*. Aceasta presupune suprapunerea unor stări disjuncte din execuţia unor instrucţiuni. În paralel cu extragerea codului unei instrucţiuni şi trecerea la execuţia ei, se extrage şi codul următoarei instrucţiuni. Aceste particularităţi ţin exclusiv de arhitectura microprocesorului pe care va rula programul. Se poate rafina optimizarea timpului de execuţie luând în calcul aceste partic-ularităţi, dar cu referire strictă la un tip de procesor.

**Bibliografie**

1. I. Ivan, S. Coman, Al. Balog, R. Arhire, *Tehnici de evaluare a efectelor optimizarii de programe*, Revista de statistică, nr 3, 7 - 1983
2. Sen-Cuo Ro, Shean-Chuen Her, *i386/i486 Advanced Programming*, Van Nostrand Reinhold, New York, 1993
3. Marvin Schaefer, *A Mathematical Theory of Global Program Optimization*, Prentice Hall, înc., Enghrood Cliffs, New Jersey, 1973
4. Julio Sanchez, Maria P. Canton, *Numerical Programming the 387, 486 & Pentium*, McGraw-Hill, înc., New York, 1995
5. Patrick Cohen, *Le microprocesseur PENTIUM. Architecture et programation*, Armand Colin, Paris, 1994
6. Michael L. Schmit, *Pentium Processor. Optimization Tools*, A. P. Professional, London, 1995
7. Holgher Schakel, *Programmer en Assembleur sur PC*, Editions Micro Application, Paris, 1993