OPTIMIZAREA PROGRAMELOR PE ARHITECTURI INTEL FOLOSIND MASM

Rocsana BUCEA-MANEA-ŢONIŞ
Academia de Studii Economice – Bucureşti

Este doctorand la Academia de Studii Economice din Bucureşti, în domeniul informaticii de gestiune, cu teza *Sistem de asistarea deciziei cu aplicaţii Business Intelligence pentru IMM-urile din România*. Activitatea ei s-a concentrat în special pe dezvoltarea de site-uri, administrarea bazelor de date Access, SqlServer2005 şi MySql, programare în PHP, ASP şi Javascript, analiza statistica în domeniul cercetărilor de marketing cu SPSS, Eview, Sphinx şi implementarea tehnicilor de web-marketing pe platforme libere şi Open source.

Radu BUCEA-MANEA-ŢONIŞ
ICPA – Bucureşti

Licenţiat al Academiei de Studii Economice, a devenit asistent în cercetare la ICPA – Bucureşti, în cadrul Laboratorului de informatică pentru mediu şi sol, unde efectuează raportări privind terenurile ecologice omogene furnizate de sisteme informatice de tip GIS şi tehnologii client-server. A participat la cursuri de perfecţionare finanţate de Uniunea Europeană şi este consultant IMM pe probleme de e-Commerce. A condus mai multe proiecte de dezvoltare site-uri, iar în prezent este doctorand cu lucrarea *Tehnologii informatice pentru realizarea soluţiilor de e-Business.*

REZUMAT. În acest articol este prezentată o viziune asupra arhitecturii calculatorului din perspectiva limbajelor de asamblare. Se prezintă elemente referitoare la arhitectura procesorului, arhitecura generală a sistemului de operare Unix şi modalităţi tehnice prin care Microsoft Assambler(MASM). asigură interacţiunea hardware - software. În acest context, studiul de caz prezintă modalităţi de optimizare a codului program generat folosind MASM.

Cuvinte chele: MASM, UCP, stack.

ABSTRACT. This paper presents an overview upon computer architecture from the perspective of assembling languages. There are presented issues about processor architecture, UNIX operating system, Microsoft Assembler (MASM) and hardware resources management. The case study shows ways of optimizing source code using MASM.

Key words: MASM, CPU, stack.

Procesoarele Intel pe 32 de biţi deţin cea mai mare cotă de piaţă IT&C, în prezent. De asemenea, sistemele de operare derivate din Unix reprezintă o tendinţă actuală atât în cazul sistemelor open sourc, cât şi al celor comerciale. În scopul obţinerii vitezelor de execuţie din ce în ce mai mari şi a gestiunii eficiente a memoriei interne, este necesară optimizarea de nivel scăzut a programelor cu MASM.

1. ARHITECTURA PROCESORULUI IA-32

Unitatea centrală de prelucrare (UCP) operează calculele şi operaţiile logice şi este formată din următoarele componente:

– entităţi de stocare locală a datelor numite registre;

– ceas intern cu frecvenţa foarte înaltă (>2GHz);

– unitate de control (UC) care coordonează succesiunea de etape aferente execuţiei instrucţiunilor ;

– unitate arimetico-logică(UAL) responsabilă cu operaţii aritmetice (+, –) şi logice (AND, OR, NOT).

UCP este conectată la placa de bază prin intermediul unui soclu (socket). Cei mai mulţi pini se leagă *la magistrala de date*, *magistrala de control* şi *magistrala de adrese.*

- Magistrala de date transportă instrucţiuni şi date între UCP si memorie.
- Magistrala de control transportă semnale binare (biţi de control) în vederea sincronizării dispozitivelor ataşate la magistrala BUS de sistem.
- Magistrala de adrese transportă adresele instrucţiunilor şi datelor în momentul transferului între CPU şi memorie.

Întrucât procesorul funcţionează la o frecvenţă superioară celorlalte componente hardware, apar frecvent stări de aşteptare de ordinul nanosecundelor.

Executarea unei singure instrucţiuni presupune parcurgea unor etape distincte care formează *ciclul*

de execuţie al instrucţiunii. Înainte de a fi executat, un program este încărcat în memorie. Instrucţiunile conţinute de acesta sunt stocate într-o coadă, iar *pointerul instrucţiune* referă următoare instrucţiune care urmează a fi executată. Următoarele etape definesc ciclul de execuţie al unei instrucţiuni (fig. 1):

– aducerea primei instrucţiuni din coada de aşteptare şi incrementarea contorului de program (IP);

– translatarea instrucţiunii din cod maşină în microcod. Operanzii instrucţiunii sunt trimişi către ALU împreună cu semnale care indică operaţia care urmează să fie calculată;

– dacă operandul este stocat în memorie, UC citeşte valoarea din memorie şi o copiază în registrele interne;

– ALU execută operaţia solicitată utilizând registrele de uz general şi registrele interne şi stochează rezultatul în registrele de uz general. ALU semnalizează starea execuţiei prin intermediul biţilor de control;

– dacă operandul de ieşire se găseşte în memorie, UC transferă rezultatul către un dispozitiv de Intrare/ Ieşire.

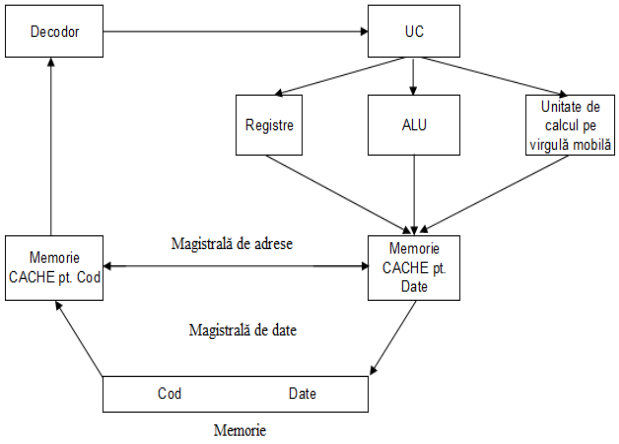

Fig. 1. Schema generală de funcţionare a procesorului IA-32.

Un procesor poate să execute în paralel mai multe etape de execuţie, procedeu denumit *pipeline.* Pentru un procesor cu un ciclu de execuţie format din şase etape, avem [Irvine, 2006]:

E1 – interfaţa BUS(BIU) accesează memoria pentru scriere/citire;

E2 – instrucţiunile citite sunt introduse într-o coadă;

E3 – instrucţiunea curentă este tradusă în microcod;

E4 – instrucţiunile în microcod sunt executate;

E5 – adresele logice pe 16 biţi sunt liniarizate pe 20 biţi ;

E6 – adresele liniarizate sunt transformate în adrese fizice (2 × 16 biţi).

Tabelul 1. **Execuţie paralelă cu şase stadii**

	1	2	3	4	5	6
1	1					
2	2	1				
3	3	2	1			
4		3	2	1		
5			3	2	1	
6				3	2	1
7					3	2
8						3

Un program în execuţie se numeşte proces (task) şi îşi administrează propriul spaţiu de memorie. Posibilitatea unui sistem de operare de a gestiona execuţia simultană a mai multor procese se numeşte *multitasking.* O UCP *singlecore* nu poate executa decât o instrucţiune la un moment dat astfel încât o componentă a sistemului de operare (scheduler) alocă fiecărui proces o secvenţă de timp în care un bloc de instrucţiuni poate fi executat. Dacă secvenţa de timp alocată este direct proporţională cu importanţa procesului, discutăm despre *multitasking preemptiv*.

Pentru a trece la executarea unui alt proces, UCP trebuie să salveze starea vechiului proces, mai precis:

– conţinutul din registre;

– valoarea contorului de program;

– valoarea biţilor de control;

– referinţe la memoria segmentată.

Procesoarele IA-32 operează în două moduri principale: *real* şi *protejat*. În modul real, caracteristic sistemelor de operare începând cu Ms.DOS şi până la Windows 98, programele beneficiază de acces direct la resursele de memorie şi dispozitivele hardware, acest fapt cauzând blocări frecvente ale sistemului de operare. În modul protejat, programele gestionează separat segmente de memorie, iar procesorul împiedică accesarea zonelor de memorie din afara segmentelor asociate. În modul *virtual*, un program poate accesa o zonă de memorie rezervată sistemului de operare fără a periclita funcţionarea acestuia.

În modul protejat, un procesor poate adresa până la 4 GB de memorie, în modul real până la 1 MB, iar în modul virtual fiecare program poate accesa 1 MB de memorie.

Un procesor are opt registre de uz general pe 32 de biţi. Le amintim pe cele mai importante:

EAX – denumit şi registru acumulator extins [Grosso, 2007], folosit implicit în operaţiile de multiplicare şi împărţire;

ECX – utilizat în mod implicit ca şi contor pentru instrucţiunile repetitive;

ESP – referă datele din structura de memorie de tip FILO (*stack),* decrementându-şi valoarea de la adrese superioare la adrese inferioare asociate primului byte al ultimului element introdus în stack.

În stack sunt memorate variabilele locale, parametrii funcţiilor şi adresele de memorie asociate rezultatelor întoarse. Procesul care apelează funcţia adresează zona superioară din stack pentru argumentele de intrare şi parametrii de ieşire. Funcţia apelată adresează zona inferioară setând registrul *frame stack* (EBP) ca marcator pentru referirea indirectă a zonei alocate variabilelor locale. Zona de memorie alocată de funcţie este eliberată apoi, incrementând valoarea registrului ESP cu valoarea registrului de frame – EBP [Blunden, 2003]:

– DS, CS, SS – registre pe 16 biţi care în modul real adresează zona de date, zona de cod şi zona de stack alocate unui program, iar în modul protejat referă tabelele descriptoare de segment;

– EIP – registrul referinţă de instrucţiune având rolul de contor de program;

– EFLAGS – controlează funcţionarea UCP prin intermediul biţilor de control.

2. ARHITECTURA SISTEMULUI DE OPERARE

Relaţia hardware-software care se stabileşte într-un calculator personal este cel mai bine explicată în relaţie cu conceptul de maşină virtuală introdus de Andrew Tanenbaum. Din această perspectivă, un calculator poate fi văzut ca o superpoziţie de maşini virtuale de diferite nivele. De exemplu o maşină virtuală de nivel 1 poate executa instrucţiuni scrise doar într-un limbaj ipotetic L1, în timp ce o maşină virtuală de nivel 0 execută numai comenzi scrise într-un limbaj L0. Din acest motiv, limbajele de programare de nivel înalt trebuie interpretate sau translatate în limbaje de programare de nivel redus pentru a accesa direct dispozitive hardware. Dar dacă, totuşi, limbajul ipotetic L1 este mult prea complex pentru a fi învăţat şi utilizat de un programator obişnuit? În acest caz se impune proiectarea unei maşini virtuale de nivel 2 care să poată fi gestionată cu un limbaj de programare L3 de nivel foarte înalt, mai puternic şi mai intuitiv (fig. 2).

Sistemul de operare este acea componentă software care gestionează resursele hardware ale calculatorului şi asigură un mediu de execuţie pentru programele rezidente. Arhitectura unui sistem cuprinde *kernelul* sau nucleul sistemului de operare, responsabil cu gestionarea dispozitivelor de Intrare/Ieşire, sistemul de funcţii primitive, bibliotecile software standard, componenta *shell* care interfaţează alte aplicaţii utile sistemului de operare (fig. 3) [Stevens, 2005].

nivelul 5	Limbaj de programare de nivel înalt
nivelul 4	Limbaj de asamblare
nivelul 3	Sistem de operare
nivelul 2	CISC/RISC
nivelul 1	Circuite logice
nivelul 0	Formalism logico-digital

Fig. 2. Sistemul de calcul ca o stivă formată din diferite nivele de abstractizare, după A.S. Tanenbaum.

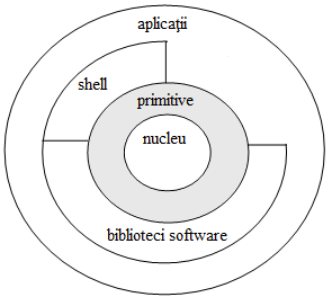

Fig. 3. Arhitectura sistemului de operare.

Shell este un interpretor de comenzi care preia intrarea de la utilizator şi execută instrucţiunea cu eventualele argumente. Un program poate fi executat după ce numele său a fost introdus în linia de comandă a terminalului după *prompt* sau prin parcurgerea secvenţială a unui fişier de *script.* Există mai multe tipuri de shell, cum ar fi DOS Shell şi Power Shell pentru Windows sau Bourne Shell, Korn Shell şi TENEX Shell pentru UNIX (fig. 4).

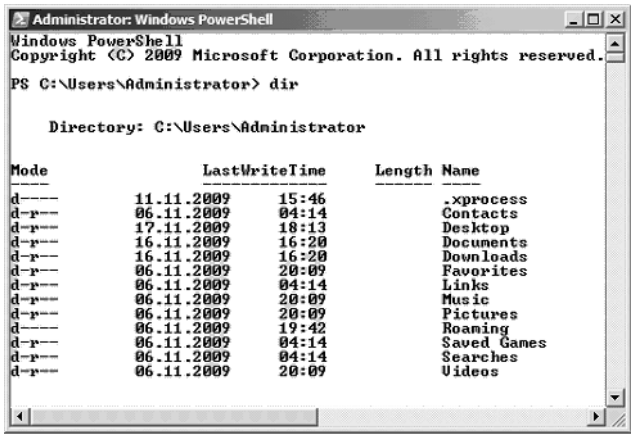

Fig. 4. Power Shell pentru Windows Server 2008.

Sistemul de primitive asigură accesul programelor la kernel pentru a executa operaţii de citire/scriere. Sistemul de primitive este accesat fie în limbajul de asamblare, fie în limbajul C, reglementat în acest caz de standardul

POSIX. Bibliotecile software standard asigura suportul necesar întocmai pentru efectuarea acestor operații. De exemplu pentru metoda standard de scriere *printf* din biblioteca *stdio* este accesata primitiva *write* care este şi ea supra-definită în biblioteca standard *unistd.*

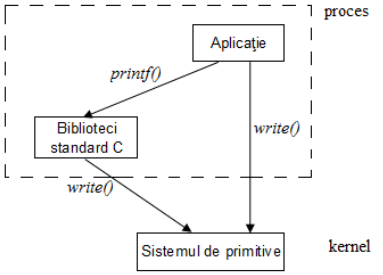

Fig. 5. Accesarea kernel de către sistemul de primitive şi via C.

Pentru a efectua scrierea într-un fişier utilizând primitiva *write()*, se va utiliza descriptorul de fişier, cu altă valoare decât 0, 1 sau 2 corespunzătoare descriptoarelor standard STDIN, STDOUT şi STDERR (fig. 5).

Sistemul de fişiere dintr-un sistem de operare este organizat ierarhic plecând de la directorul rădăcină denumit *root* şi simbolizat prin caracterul "/" pe sisteme UNIX şi caracterul "C" pe sisteme Windows. Apelarea fişierelor pe disc se face direct, precizând calea întreagă pornind de la directorul rădăcină sau relativ, specificând calea în raport cu directorul curent. Denumirea unui fişier poate fi limitată la 255 de caractere în funcție de sistemul de operare utilizat.

În cazul fişierelor executabile, sistemul de operare caută în tabela de alocare a fişierelor pe disc (*ext3* sau *NTFAT*) dimensiunea şi calea fişierului. Apoi sistemul de operare identifică următoarea locație de memorie liberă şi adaugă informațiile din tabela de alocare în tabela de descriptori. După această etapă, este creat un proces copil cu identificatorul de proces *PID* setat la 0, specificând în acest fel prioritatea execuției față de procesul părinte, şi este executată prima instrucțiune din program. Procesul rulează până când cedează controlul procesului părinte ori sistemului de operare.

3. STUDIU DE CAZ: FOLOSIREA LIMBAJULUI DE ASAMBLARE PENTRU OPTIMIZAREA PROGRAMELOR C/C++

Optimizarea vitezei de execuție poate viza modificări asupra fişierului intermediar de assembler generat de compilatorul de C++. În general, compilatoarele nu pot egala performanțele dezvoltatorului uman în scrierea de cod assembler optimizat. Este prezentată în continuare o aplicație de prelucrare grafică pentru imagini color de tip hartă de biți (BMP), cu o adâncime a culorii mai mare de 8 biți. Un pixel este format din trei serii de biți, una pentru fiecare culoare de baza. În funcție de adâncimea de culoare n, un pixel va ocupa n/8 bytes, prin urmare spațiul ocupat de un rând din imagine se calculează după formula 1, iar dimensiunea unui fişier, după formula 2, pentru o aliniere la un cuvânt (WORD) de memorie.

$$RandDim = \frac{4 \times Adancime_{culoare} \times Latime_{imagine}}{32} \quad (1)$$

$$BMPDim = Dimensiune_{Antet} + RandDim \times Inaltime_{imagine} \quad (2)$$

În general, un fişier BMP este structurat pe 4 niveluri, după cum rezultă din figura 6.

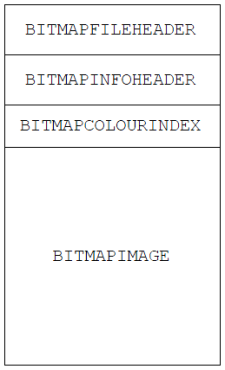

Fig. 6. Structura de date BITMAP.

Headerul cu informații generale despre fişier conține următoarele structuri de date, reprezentate în limbajul de programare C/C++:

```
typedef struct tagBITMAPFILEHEADER
{
   WORD bfType; // precizeza tipul fisierului, ex. BM
   DWORD bfSize; //dimensiunea în bytes a fisierului
   BMP
   WORD bfReserved1; //rezervat, trebuie sa fie 0
   WORD bfReserved2; //rezervat, trebuie sa fie 0
   DWORD bfOffBits; //numarul de bytes de la care
   incepe imaginea propriuzisa
}BITMAPFILEHEADER;
```

Headerul Device IndependentBitmap(DIB), o componentă esențială pentru interfața programabilă (API) Windows GDI, conține următoarea structură:

```
typedef struct tagBITMAPINFOHEADER
{
   DWORD biSize; //dimensiunea structurii în bytes
   DWORD biWidth; //latimea imaginii în pixeli
   DWORD biHeight; //inaltimea imaginii în pixeli
   WORD biPlanes; //numarul paletelor de culoare,
   trebuie sa fie 1
   WORD biBitCount; //numarul de biți pentru un
   pixe
   DWORD biCompression;//tipul compresiei de date,
   0 pentru BMP
   BITMAPFILEHEADER
   BITMAPINFOHEADER
   BITMAPCOLOURINDEX
   BITMAPIMAGE
   DWORD biSizeImage; //dimensiunea imaginii în
   bytes
```

```
   DWORD biXPelsPerMeter; //pixeli/metru aferenti
   abscisei(X)
   DWORD biYPelsPerMeter; //pixeli/metru aferenti
   ordonatei(Y)
   DWORD biClrUsed; //numarul de culori utilizate
   DWORD biClrImportant; //numarul culorilor
   relevate
}BITMAPINFOHEADER;
```

Paleta de culori referă cele trei culori fundamentale pe 8 biți, adică 256 de combinații pentru fiecare culoare organizate în rastere de la stânga la dreapta, după forma (X,Y) = Roşu (R) 255 Verde (G) 255 Albastru (B) 255:

```
typedef struct tagBITMAPCOLOURINDEX
{
   BYTE red;//rosu
   BYTE green;//verde
   BYTE blue;//albastru
   BYTE junk;//zgomot
}
BITMAPCOLOURINDEX;
```

Imaginea scanată dintr-un fişier de tip BMP va fi încărcată în memorie pentru efectuarea normalizării culorii, deoarece paleta de culori poate veni în format BGR. În acesta ordine de idei vor fi citite headerele fişierului pentru a obține OFFSET-ul sau poziția absolută a primului pixel din imagine şi dimensiunea acesteia.

Odată citită imaginea, se poate trece la inter schimbarea culorilor de baza roşu (R) şi albastru (B), după următorul algoritm:

```
for (imageIdx = 0;imageIdx <
bitmapInfoHeader.biSizeImage;imageIdx+=3)
{
   tempRGB = bitmapImage[imageIdx];
   bitmapImage[imageIdx] = bitmapImage[imageIdx + 2];
   bitmapImage[imageIdx + 2] = tempRGB;
}
```

După generarea codului de assembler, se impune o optimizare de nivel scăzut care cuprinde următoarele etape:

1) Utilizarea instrucțiunii mov pentru copierea matricii de pixeli:

```
mov eax,dword ptr [filePtr]
push eax
push 1
mov ecx,dword ptr [ebp-2Ch]
push ecx
mov edx,dword ptr [bitmapImage]
push edx
call dword ptr [__imp__fread (0EC7254h)]
add esp,10h
```

2) Desfăşurarea structurii repetitive responsabile cu modificarea imaginii şi care alocă cel mai lung timp în vederea execuției cu un factor de 3:

```
mov dword ptr [imageIdx],0
jmp LoadBitmapFile+0FBh (0EC139Bh)
mov eax,dword ptr [imageIdx]
add eax,3
mov dword ptr [imageIdx],eax
mov eax,dword ptr [imageIdx]
cmp eax,dword ptr [ebp-2Ch]
jae LoadBitmapFile+12Dh (0EC13CDh)
mov eax,dword ptr [bitmapImage]
add eax,dword ptr [imageIdx]
mov cl,byte ptr [eax]
mov byte ptr [tempRGB],cl
mov eax,dword ptr [bitmapImage]
add eax,dword ptr [imageIdx]
mov ecx,dword ptr [bitmapImage]
add ecx,dword ptr [imageIdx]
mov dl,byte ptr [ecx+2]
mov byte ptr [eax],dl
mov eax,dword ptr [bitmapImage]
add eax,dword ptr [imageIdx]
mov cl,byte ptr [tempRGB]
mov byte ptr [eax+2],cl
jmp LoadBitmapFile+0F2h (0EC1392h)
```

3) Mutarea de la început a conținutului variabilelor în regişştri. Atunci când nu este stocat într-un registru, o cantitate semnificativă din timpul de execuție al unei structuri repetitive este dedicat preluării variabilei din memorie, atribuirii unei valori noi pentru ea şi stocarea ei înapoi în memorie iar şi iar. Stocarea acestora într-un registru poate îmbunătăți în mod semnificativ performanța:

```
mov dword ptr [imageIdx],0
```

Rezultatul optimizat al prelucrării este prezentat în figura 7.

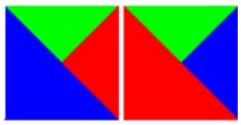

Fig. 7. Imaginea BMP pe 24 de biți înainte şi după prelucrarea culorilor de bază.

4. CONCLUZII

Limbajele de asamblare se deosebesc deosebit de utile atunci când se ridica problema compatibilității unui program software cu arhitectura hardware, facilitând optimizări de nivel scăzut care permit execuția rapida şi eficienta pe o platforma dedicata. Chiar dacă în prezent MASM este un prețios instrument didactic pentru facultățile de profil, se impune accentuarea importantei învățării acestui limbaj în etapele incipiente de diseminare a conținutului didactic, prin rolul sau fundamental, stand la baza oricărui limbaj de programare sau compilator modern.

BIBLIOGRAFIE

[5] **Blunden, B.**, *Software Exorcism*, Apress, 2003.
[6] **Grosso, E., Bicego M.**, *Fondamenti di Informatica per l'universita*, G. Giappichelli Editore, 2007.
[7] **Irvine, Kip R.**, Assembly Language for Intel-based Computers, Prentice Hall, 2006.
[8] **Stevens, W., R., Rago, St.,A.**, *Advanced Programming in the UNIX Environment*, Addison-Wesley, 2005.