

Setul de instrucțiuni x86-64

Cuprins

1	Registrii procesorului x86-64	2
2	Folosirea mediului de lucru	4
3	Sintaxa generală a instrucțiunilor în MASM	4
4	Structura unui program în limbaj de asamblare	5
5	Funcții de afișare și scriere	6
6	Declararea și inițializarea datelor	6
7	Instrucțiuni de transfer	7
8	Instrucțiuni aritmetice	8
8.1	Adunarea și scăderea	8
8.2	Înmulțirea și împărțirea	9
8.3	Incrementare și decrementare	10
9	Instrucțiuni logice	10
9.1	Instrucțiunile: AND, OR, XOR, NOT, TEST	10
9.2	Instrucțiuni de deplasare: shl, shr, sar, sal	12
9.3	Instrucțiuni de rotire: rol, rcl, ror, rcr	13
10	Moduri de adresare	13
10.1	Adresarea imediată. Adresare de tip registru	13
10.2	Adresarea indirectă	14
10.3	Adresare indirectă	14
11	Structuri condiționale și iterative. Controlul fluxului de instrucțiuni	15
11.1	Salt necondiționat	15
11.2	Salt condiționat	15
12	Instrucțiuni pentru numere reale	16
12.1	Operații aritmetice cu numere reale	17
12.2	Instrucțiuni condiționale cu numere reale	19
13	Stiva. Proceduri	19
14	Template-uri if	23
15	Template-uri while	24
16	Template procedura	26
17	Exemple	28
18	Exemple 32 bit	43

1 Regiștrii procesorului x86-64

Familia de procesoare x86-64 este cea mai comună pe piața calculatoarelor desktop și laptop-urilor (celelalte dispozitive mobile folosesc adesea arhitectura ARM). Deși, de-a lungul anilor, aceste procesoare au suferit multiple modificări arhitecturale, în designul lor s-a pus accent pe păstrarea compatibilității între versiuni.

Principalele componente ale procesorului:

- Unitatea de control - coordonează celelalte componente; este responsabilă de ceasul procesorului.
- Unitatea aritmetică și logică - este responsabilă de calculele cu numere întregi și adrese.
- regiștri - sunt unități de memorie din interiorul procesorului;
- memorie cache - este o unitate de memorie situată lângă procesor. De obicei, este organizată pe nivele L1, L2, L3 (cu capacități și viteze de acces diferite). Conține blocuri de cod accesate frecvent de procesor.
- FPU (floating point unit) - coprocesorul matematic x87, pentru lucrul cu numere reale
- unități MMX și SSE - sunt extensii SIMD (single instruction multiple data) ale procesorului pentru lucrul simultan cu multiple numere întregi și numere reale, pentru procesarea mai eficientă a datelor.

Procesorul x86-64 dispune de mai mulți regiștri printre care:

- 16 regiștri de uz general (general purpose registers) de 64 biți.
- 2 regiștri de control de 64 biți: RIP și RFLAGS.
- 8 regiștri pentru procesorul FPU x87.
- 16 regiștri XMM, pentru setul de instrucțiuni SSE. Sunt folosiți adesea pentru lucrul cu numere reale.

Regiștrii de uz general sunt folosiți de obicei ca operanzi (sursă și/sau destinație) pentru o mare parte din instrucțiuni. Este important de reținut totuși că unele instrucțiuni folosesc ca operanzi doar anumiți regiștri. Aceștia sunt:

Registru (Biții 63-0)	Denumire	Biții 31-0	Biții 15-0	Biții 15-8	Biții 7-0
RAX	Accumulator Register	EAX	AX	AH	AL
RBX	Base (Index) Register	EBX	BX	BH	BL
RCX	Counter Register	ECX	CX	CH	CL
RDY	Data Register	EDX	DX	DH	DL
RSI	Source Index	ESI	SI		SIL
RDI	Destination Index	EDI	DI		DIL
RBP	Base Pointer	EBP	BP		BPL
RSP	Stack Pointer	ESP	SP		SPL
r8		r8d	r8w		r8b
r9		r9d	r9w		r9b
r10		r10d	r10w		r10b
r11		r11d	r11w		r11b
r12		r12d	r12w		r12b
r13		r13d	r13w		r13b
r14		r14d	r14w		r14b
r15		r15d	r15w		r15b

- Regiștrii de date RAX, RBX, RCX și RDX sunt folosiți în principal pentru lucrul cu numere întregi (operații aritmetice și logice), operații cu adrese (de exemplu, parcurgeri de array-uri). De asemenea, apar și în convenții de apel a procedurilor.
- Regiștrii index RSI și RDI sunt folosiți în principal pentru lucrul cu șiruri de caractere.
- Regiștrii pointer RSP și RBP sunt folosiți pentru lucrul cu stiva procesorului.

Regiștrii de control (de 32 de biți) sunt RIP (Instruction Pointer) și RFLAGS (Flag register).

Registrul RIP (Extended Instruction Pointer), mai poartă numele de **Program Counter** și este esențial execuției instrucțiunilor.

Procesorul execută instrucțiunile secvențial. Ciclul de execuție (*fetch-decode-execute*) constă din următorii pași:

- instrucțiunea care urmează este preluată de la adresa conținută în registrul **RIP** și adăugată în registrul **IR** (instruction register).
- **RIP** este încărcat cu adresa instrucțiunii următoare.
- Instrucțiunea din **IR** (ea este doar un **opcode**, șir de biți până în acest punct) este decodificată, apoi sunt aduși în procesor operandii precizați de instrucțiune.
- Procesorul execută instrucțiunea și transferă rezultatul către operandul destinație.
- se reia ciclul.

Așadar, registrul **RIP** conține **adresa** instrucțiunii care urmează și este astfel cel responsabil de gestionarea fluxului de execuție a aplicațiilor.

Registrul **RFLAGS**, conține indicatorii (sau flag-uri) de stare, control, sistem. Acesta din urmă raportează starea execuției unui program și este folosit de sistemul de operare sau de aplicații pentru a controla fluxul de instrucțiuni.

Conținutul registrului **RFLAGS** este următorul:

Poziția	Nume	Tip Indicator	Descriere
0	CF	Stare	Carry Flag
1	-	-	Rezervat 1
2	PF	Stare	Parity Flag
3	-	-	Rezervat 0
4	AF	Stare	Auxiliary Flag
5	-	-	Rezervat
6	ZF	Stare	Zero Flag
7	SF	Stare	Sign Flag
8	TF	Sistem	Trap flag
9	IF	Sistem	Interrupt Enable Flag
10	DF	Control	Direction Flag
11	OV	Stare	Overflow Flag
12-13	IOPL	Sistem	I/O Privilege Level Flag
14	NT	Sistem	Nested Task Flag
15	-	-	Rezervat 0
16	RF	Sistem	Resume Flag
17	VM	Sistem	Virtual 8086 Mode Flag
18	AC	Sistem	Alignment check
19	VIF	Sistem	Virtual interrupt flag
20	VIP	Sistem	Virtual interrupt pending
21	ID	Sistem	ID Flag
22-63	-	-	Rezervat

Nu există o instrucțiune care să acceseze întreg registrul **RFLAGS** direct. Însă, conținutul său poate fi încărcat și descărcat pe stivă. De asemenea, pot fi accesați direct anumiți biți ai acestuia.

Indicatorii de stare pe care îi vom folosi ocupă pozițiile 0, 6, 7 și 11 ai registrului **RFLAGS** și sunt **CF**, **ZF**, **SF**, **OF**.

- Flag-ul **ZF** (zero flag) indică dacă ultima instrucțiune a avut sau nu rezultatul zero. Dacă acesta este zero, **ZF**=1; în caz contrar, **ZF**=0.
- Flag-ul **CF** (carry flag) indică dacă a fost realizat un transport în exterior de pe bitul cel mai semnificativ al rezultatului unei operații aritmetice cu numere fără semn (a existat o depășire de capacitate în cazul *unsigned*).
- Flag-ul **OF** (overflow flag) indică dacă a fost realizat un transport în exterior de pe bitul cel mai semnificativ al rezultatului unei operații cu numere cu semn (a existat o depășire de capacitate în cazul *signed*).
- Flag-ul **SF** (sign flag) indică semnul rezultatului unei operații: 1 pentru numere negative, 0 pentru pozitive.

2 Folosirea mediului de lucru

Pentru a implementa programe în limbajul de asamblare MASM folosind Visual Studio procedăm astfel:

1. Creăm un nou proiect: **File**→**New**→**Project**.
2. Alegem **Empty Project** și setăm denumirea proiectului.
3. În fereastra **Solution Explorer**, *Right Click* pe numele proiectului, apoi selectăm **Build Dependencies**. În noua fereastră, selectăm **asm**.
4. În meniul **Solution Explorer**, adăugați la **Source Files** un fișier cu extensia **.asm**.
5. La începutul sursei **.asm** includem librăriile:

```
includelib libcmt.lib  
includelib legacy_stdio_definitions.lib
```

6. În fereastra **Solution Explorer**, *Right Click* pe numele proiectului, apoi selectăm **Properties**→**Linker**→**System** și selectăm **Subsystem:Console**.
7. (Opțional) În fereastra **Solution Explorer**, *Right Click* pe numele proiectului, apoi selectăm **Properties**→**Linker**→**Right Click** pe fișierul **.asm**, apoi selectăm **Microsoft Macro Assembler**→**Listing file**. La rubrica **List All Available Information** selectăm **Yes**, iar la **Assembled Code Listing File** scriem \$(ProjectName).lst (sau orice denumire de fișier cu extensia **.lst**).

Observație. În fișierul **.lst** generat de asamblor se pot observa deplasamentele (offset-urile) tuturor variabilelor (față de începutul segmentului de date inițializat prin directiva **.DATA**), respectiv deplasamentele instrucțiunilor executabile (față de începutul segmentului de cod inițializat prin directiva **.CODE**).

3 Sintaxa generală a instrucțiunilor în MASM

Sintaxa generală a liniilor ale limbajului de asamblare MASM este:

[mnemonica] [oper1] [,oper2]

unde:

- **mnemonica** este numele instrucțiunii. La asamblare, ea (împreună cu operandii) este tradusă în cod mașină.
- operandii **oper1**, **oper2** sunt valori de intrare pentru instrucțiune.
 - Ei pot fi: constante, regiștri sau variabile din memorie.
 - Unele instrucțiuni au un operand implicit (neprecizat explicit).
 - În cazul unei instrucțiuni cu doi operandi, primul primește și rezultatul instrucțiunii.

Sintaxa MASM mai include:

- Etichetele dunt simboluri (șiruri de caractere) folosite pentru a identifica o instrucțiune în segmentul de cod. În acest caz sunt urmate de ":". Ele nu ocupă memorie în program. Instrucțiunea care apelează o etichetă o convertește într-o adresă relativă, ce reprezintă deplasamentul între instrucțiunea apelantă și poziția în cod etichetei.
- Numele variabilelor și numele procedurilor sunt utilizate similar cu etichetele (convertite la apel în deplasamente). Pentru variabile, etichetele sunt convertite în deplasamente față de începutul de segment de date.
- Directivele sunt o instrucțiuni care fac parte din sintaxa asamblorului, dar nu face parte din setul de instrucțiuni **x86-64**; ele nu sunt executate de procesor la rularea aplicației. Directivele nu ocupă (direct) memorie în program, deși unele dintre ele sunt înlocuite de asamblor cu o secvență de instrucțiuni care ocupă memorie. De exemplu, directiva **.CODE** este folosită de asamblor pentru a identifica porțiunea din program care conține instrucțiunile ce trebuie executate.
- toate comentariile sunt prefixate de caracterul ";".

- Constantele numerice sunt specificate printr-un șir de cifre și litere, ce începe obligatoriu printr-o cifră. Numerele în baza 10 se scriu doar printr-un șir de cifre. Numerele în baza 2 se scriu printr-un șir de cifre binare 0 și 1 care se termină cu litera 'B'. Numerele hexazecimale trebuie să se termine cu litera 'H' și trebuie să înceapă cu o cifră (eventual 0 dacă e necesar).
- Caracterele sau șirurile de caractere se scriu între ghilimele sau apostrofuri. Ele sunt reprezentate intern ca numere de 8 biți, folosind codul ASCII.

4 Structura unui program în limbaj de asamblare

Structura unui program în limbajul de asamblare MASM este următoarea:

```
; includem librariile
includelib libcmt.lib
includelib legacy_stdio_definitions.lib
; declaram funtiile importate
printf proto
scanf proto
.DATA
;declaram variabilele folosite
.CODE
main PROC
; instructiuni
    ret
main ENDP
END
```

Semnificația instrucțiunilor este următoarea:

1. Directiva `.DATA` marchează începutul segmentului de date, care va conține variabilele folosite.
2. Directiva `.CODE` marchează secțiunea programului care conține instrucțiunile executabile.
3. Directiva `PROC` marchează începutul unei proceduri. Toate procedurile se încheie prin directive `ENDP`. Toate programele folosite vor avea funcția `main`.
4. Instrucțiunea `ret` întoarce controlul (consolei) înapoi la sistemul de operare.
5. Directiva `END` indică finalul programului.

Un exemplu explicit este următorul program care adună două variabile.

```
.DATA
n1 SDWORD 10
n2 SDWORD 13
sum SDWORD ?
.CODE
main PROC
mov eax, n1
add eax, n2
mov sum, eax
ret
main ENDP
END
```

În segmentul de date (precedat de directiva `.DATA`) avem declarate trei variabile de tip `SDWORD` numite `n1`, `n2` și `sum`. Primele două sunt inițializate cu valorile 10, respectiv `-5` iar a treia este lăsată neinițializată.

Segmentul de cod conține patru instrucțiuni:

- `mov` este o instrucțiune de transfer a valorii celui de-al doilea operand în primul operand.
- Instrucțiunea `add` calculează suma celor doi operanzi și transferă valoarea în primul dintre ei.
- `ret` marchează ieșirea din procedura `main`.

La finalul programului, variabila `sum` va conține valoarea 23.

5 Funcții de afișare și scriere

Pentru a afișa pe ecran vom folosi funcția `printf` a limbajului C. Ea are sintaxa generală:

```
printf (const char *format , arg1 , arg2 , ... )
```

(așadar `printf` are un număr variabil de argumente).

Șirul `format` conține caractere obișnuite (afișate exact cum sunt scrise) plus specificatori de format, care încep cu `"%"`, pentru afișarea argumentelor care urmează. Câțiva dintre specificatori sunt:

Specificator	Descriere
<code>%d</code>	număr întreg (cu semn) scris în baza 10, de 32 de biți
<code>%u</code>	număr întreg (fără semn) scris în baza 10, de 32 de biți
<code>%x,%X</code>	număr întreg de 32 de biți scris în baza 16
<code>%lld</code>	număr întreg (cu semn) de 64 de biți
<code>%llu</code>	număr întreg (fără semn) de 64 de biți
<code>%f,%lf</code>	număr real, pe 32 de biți, respectiv 64 de biți
<code>%s</code>	șir de caractere
<code>%c</code>	caracter

Exemplul 1.

Comandă	Text afișat
<code>printf("Hello world");</code>	Hello world
<code>printf("Numarul a este %d",10);</code>	Numarul a este 10
<code>printf("%s","Hello world");</code>	Hello world
<code>printf("Numarul a in hex este %X",10);</code>	Numarul a in hex este A

Pentru citire de la tastatură putem folosi funcția `scanf` a limbajului C. Ea are sintaxa generală

```
scanf(const char *format , ... )
```

și are ca efect citirea de date de la tastatură conform formatului precizat de șirul `format`.

Exemplul 2.

Comandă	Descriere
<code>scanf("%d",&a);</code>	Citește de la tastatură un număr întreg, iar rezultatul este pus în variabila <i>a</i> (de tip int)
<code>scanf("%s",st);</code>	Citește de la tastatură un șir de caractere iar rezultatul este pus în variabila <i>st</i> (de tip string)

Argumentele secundare ale funcției `scanf` sunt **adrese** de memorie unde trebuie stocate valorile citite.

Pentru a utiliza aceste funcții în limbajul de asamblare, folosim directiva `PROTO`

```
printf PROTO  
scanf PROTO
```

Primii patru parametri ai acestor funcții vor fi încărcăți în regiștrii `RCX`, `RDY`, `r8`, `r9`. Apelul se face folosind instrucțiunea `call`.

6 Declararea și inițializarea datelor

Pentru a defini o constantă se poate folosi cuvântul cheie `EQU` cu sintaxa:

```
nume_const EQU valoare
```

Asamblorul va înlocui în program `nume_const` cu `valoare`.

Variabilele vor fi definite în segmentul de date, care este inițializat folosind directiva `.DATA`. Sintaxa inițializării unei variabile este:

```
nume tip valoare
```

unde

- **nume** este numele variabilei: șir de litere și cifre, care începe cu o literă. Acesta indică, de fapt, adresa variabilei (mai precis deplasamentul față de începutul segmentului de date) și nu se referă la valoarea dată.

- **tip** este o directivă care definește tipul variabilei.
- **valoare** este valoarea dorită. Pentru a inițializa un vector sau un șir de caractere putem scrie valorile dorite separate prin virgulă. Pentru șiruri de caractere, se recomandă ca ultimul caracter din șir să fie caracterul NULL (cod ASCII egal cu 0).

Pentru declararea variabilelor vom folosi următoarele tipuri de date:

- **BYTE** - număr întreg de 8 biți (fără semn); util pentru lucrul cu caractere.
- **SBYTE** - număr întreg de 8 biți (cu semn).
- **WORD** - număr întreg de 16 biți (fără semn).
- **SWORD** - număr întreg de 16 biți (cu semn).
- **DWORD** - număr întreg de 32 biți (fără semn).
- **SDWORD** - număr întreg de 32 biți (cu semn);
- **QWORD** - număr întreg de 64 de biți (fără semn);
- **SQWORD** - număr întreg de 64 de biți (cu semn); echivalentul tipului `int` din C.
- **real4**, **real8** - număr real, reprezentat în virgulă flotantă, pe 32 de biți (4 octeți), respectiv 64 de biți (8 octeți).

Exemplul 3. Prin instrucțiunea

```
car BYTE 65
```

declarăm o variabilă **car** de tip **byte**, și o inițializăm cu 65 (caracterul corespunzător este "A").

Exemplul 4. Prin instrucțiunile

```
numar SDWORD 5
rez SDWORD ?
```

declarăm o variabilă **număr** de tip **SDWORD** și o inițializăm cu valoarea 5. De asemenea, alocăm memorie pentru o variabilă **rez** de 32 de biți, care nu este însă inițializată.

Exemplul 5. Prin instrucțiunea

```
mesaj BYTE "Ana",0
```

alocăm 4 octeți, corespunzători caracterelor "A", "n", "a" și caracterul NULL. Eticheta **mesaj** va referi la deplasamentul primului caracter din șir "A".

Observație. Este indicat ca șirurile de caractere să se termine cu caracterul NULL (codul ASCII este 0).

7 Instrucțiuni de transfer

Pentru a transfera informații între locații de memorii și/sau regiștri se folosește instrucțiunea **mov**. Ea are sintaxa

```
mov dest , sursa
```

și are ca efect transferul valorii operandului **sursa** în operandul **dest**.

Cei doi operanzi pot fi: valoare imediată, registru, variabilă din memorie cu următoarele restricții:

- Operandul **dest** nu poate fi valoare imediată.
- Ambii operanzi nu pot fi variabile din memorie.
- Operanzii ar trebui să fie de același tip (**BYTE**, **SDWORD**, etc).

Exemplul 6.

Instrucțiune	Descriere
<code>mov rax,5</code>	transferă în registrul <code>rax</code> valoarea imediată 5
<code>mov ebx, num</code>	transferă în registrul <code>ebx</code> valoarea variabilei <code>num</code>
<code>mov num,esi</code>	transferă în variabila <code>num</code> , conținutul registrului <code>esi</code>
<code>mov eax,dx</code>	eroare: cei doi regiștri au lungimi diferite

Exemplul 7. Instrucțiunea C

`b=a`

are ca echivalent în limbaj de asamblare secvența de cod:

```
mov eax, a
mov b, eax
```

În cazul unor operandi de dimensiuni diferite se pot folosi instrucțiunile

```
movsxd dest,sursa; dest = 64bit, source = 32bit
movsx dest, sursa; alte dimensiuni
```

ce realizează transferul valorii din `sursa` în `dest`, realizând conversia corectă (extinzând bitul de semn). Există o restricție în privința operandilor: `dest` trebuie să fie registru, iar `sursa` nu poate fi constantă.

- Instrucțiunea

```
lea reg64, var_mem
```

transferă în registrul `reg64` (de 64 de biți) adresa efectivă a variabilei `var_mem`.

- Ea este echivalentul operatorului adresă `&` din C.
- Una dintre utilizările frecvente a acestei instrucțiuni este încărcarea într-un registru a adresei începutului unui string.

8 Instrucțiuni aritmetice

8.1 Adunarea și scăderea

Pentru a aduna două valori folosim instrucțiunea

```
add dest, sursa
```

Aceasta calculează valoarea sumei `dest+sursa` și încarcă rezultatul în `dest`.

Operandii sunt: valori imediate, regiștri sau variabile din memorie cu următoarele restricții:

- Operandul `dest` nu poate fi valoare imediată.
- Ambii operandi nu pot fi variabile din memorie.
- Operandii ar trebui să fie de același tip (BYTE, SDWORD, etc).

Exemplul 8.

Instrucțiune	Descriere
<code>add eax,5</code>	La conținutul registrului <code>eax</code> este adăugată valoarea 5
<code>add rbx, num</code>	La conținutul registrului <code>rbx</code> , este adăugată valoarea variabilei <code>num</code>
<code>add ebx,ax</code>	eroare: cele două regiștri au lungimi diferite
<code>mov num1,num2</code>	eroare: ambii operandi nu pot fi variabile din memorie

Similar, pentru scădere putem folosi instrucțiunea

```
sub dest, sursa
```

Aceasta calculează valoarea diferenței `dest-sursa` și încarcă rezultatul în `dest`.

8.2 Înmulțirea și împărțirea

Pentru calculul înmulțirilor, se pot folosi instrucțiunile `imul` (pentru întregi cu semn) și `mul` (pentru întregi fără semn). Deoarece la înmulțiri sunt posibile destul de ușor depășiri de capacitate, rezultatul înmulțirilor este păstrat pe o lungime dublă decât lungimea operandilor. De exemplu, dacă operandii au fiecare un octet, rezultatul înmulțirii va avea 16 biți. Vom descrie în detaliu modul de utilizare a instrucțiunii `imul`, cealaltă fiind similară.

Instrucțiunea `imul` are sintaxa:

`imul sursa`

unde operandul `sursa` poate fi registru sau variabilă din memorie (nu este acceptată o valoare imediată). Ea funcționează astfel:

- Dacă `sursa` are 32 de biți, `imul sursa` calculează înmulțirea $EAX \times sursa$ și stochează rezultatul în perechea de registre `EDX:EAX`.
- Dacă `sursa` are 64 de biți, `imul sursa` calculează înmulțirea $RAX \times sursa$ și stochează rezultatul în perechea de registre `RDX:RAX`.

Exemplul 9. Pentru a implementa în limbaj de asamblare instrucțiunea

`p=a*b`

din limbajul C (unde `p`, `a`, `b` sunt variabile `int`), folosim următoarea secvență de cod:

```
mov eax, a
imul b
mov p, eax
```

Pentru împărțire, putem folosi instrucțiunile `idiv` (pentru întregi cu semn) și `div` (pentru întregi fără semn). Vom descrie modul de utilizare a instrucțiunii `idiv`, cealaltă fiind similară. Sintaxa instrucțiunii este:

`idiv sursa`

unde operandul `sursa` poate fi registru sau variabilă din memorie (nu este acceptată o valoare imediată). Ea funcționează astfel:

- Dacă `sursa` are 32 biți, `idiv sursa` efectuează împărțirea $(EDX:EAX) : sursa$ și stochează câtul în registrul `EAX` și restul în registrul `EDX`.
- Dacă `sursa` are 64 biți, `idiv sursa` efectuează împărțirea $(RDX:RAX) : sursa$ și stochează câtul în registrul `RAX` și restul în registrul `RDX`.

Observație. Ambele instrucțiuni `imul` și `idiv` modifică registrele `EAX` și `EDX` (respectiv `RAX` și `RDX`).

Observație. Pentru instrucțiunea `idiv`, deîmpărțitul (operandul implicit) are lungime dublă față de împărțitor. Astfel, dacă deîmpărțitul operației curente are lungimea mai mică, bitul de semn al acestuia trebuie extins (adică extindem cu 0, dacă numărul este pozitiv și cu 1, dacă numărul este negativ). Pentru a efectua această operație putem folosi instrucțiunile:

Instrucțiune	Descriere
<code>cdq</code>	Extinde bitul de semn de la <code>eax</code> la <code>(edx:eax)</code>
<code>cqo</code>	Extinde bitul de semn de la <code>rax</code> la <code>(rdx:rax)</code>

Exemplul 10. Pentru a implementa în limbaj de asamblare secvența de instrucțiuni

`q=a/b;`
`r=a%b;`

din limbajul C (toate variabilele sunt de tip `int`) putem folosi următoarea secvență de cod:

```

mov eax , a
cdq
idiv b
mov q , eax
mov r , edx

```

8.3 Incrementare și decrementare

Instrucțiunea

```
inc sursa
```

realizează incrementarea (cu 1) a valorii operandului **sursa**. Acest operand poate fi registru sau variabilă din memorie.

Instrucțiunea

```
dec sursa
```

realizează decrementarea (cu 1) a valorii operandului **sursa**. Acest operand poate fi registru sau variabilă din memorie.

Observație. Limbajul C dispune de două variante de incrementare (respectiv decrementare). De exemplu, dată două variabile a și b de tip int, instrucțiunea

```
b=a++;
```

va atribui lui b valoarea lui a (cea originală) și apoi va incrementa a. O secvență de cod echivalentă din limbaj de asamblare va fi:

```

mov eax , a
mov b , eax
inc a

```

Pe de altă parte, instrucțiunea

```
b=++a;
```

va efectua incrementarea lui a mai întâi și apoi va atribui lui b valoarea lui a (cea incrementată). O secvență de cod echivalentă din limbaj de asamblare va fi:

```

inc a
mov eax , a
mov b , eax

```

9 Instrucțiuni logice

9.1 Instrucțiunile: AND, OR, XOR, NOT, TEST

Instrucțiunea AND implementează operația logică binară **ȘI** și are sintaxa:

```
AND dest , sursa
```

Operația **ȘI** este realizată la nivel de bit, adică se efectuează **ȘI** între fiecare bit al primului operand **dest** și bitul corespunzător al celui de-al doilea operand **sursa**. Rezultatul este stocat în primul operand. Operanzii pot fi regiștri, valori imediate sau valori din memorie cu restricțiile:

- Operandul **dest** nu poate fi valoare imediată.
- Ambii operanzi nu pot fi variabile din memorie.
- Operanzii ar trebui să fie de același tip (BYTE, SDWORD, etc).

Deoarece

0 ȘI a = 0

1 ȘI a = a

instrucțiunea **AND** poate fi folosită pentru izolarea unui anumit număr de biți din operandul **dest** sau, mai general, pentru stabilirea valorii 0 a unor biți fixați ai operandului **dest**

Exemplul 11. Instrucțiunea

AND dest , 1

va calcula ultimul bit al operandului **dest** (adică restul la împărțirea cu 2), iar

AND dest , 0Fh

va selecta ultimii patru biți ai lui **dest** (cu alte cuvinte restul la împărțirea cu 16).

Instrucțiunea OR implementează operația logică binară **SAU** și are sintaxa:

OR dest , sursa

Operația **SAU** este realizată la nivel de bit, adică se efectuează **SAU** între fiecare bit al primului operand **dest** cu bitul corespunzător al celui de-al doilea operand **sursa**. Rezultatul este stocat în primul operand. Operanzii pot fi regiștri, valori imediate sau valori din memorie cu restricțiile:

- Operandul **dest** nu poate fi valoare imediată.
- Ambii operanzi nu pot fi variabile din memorie.
- Operanzii ar trebui să fie de același tip (BYTE, SDWORD, etc).

Deoarece

0 SAU $a = a$

1 SAU $a = 1$

instrucțiunea **SAU** poate fi folosită pentru stabilirea valorii 1 a unui anumit număr de biți din operandul **dest**.

Exemplul 12. Instrucțiunea

OR dest , 5

va seta valoarea 1 pentru biții de pe pozițiile 0 și 2 ai lui **dest**, pe restul lăsându-i pe loc.

Instrucțiunea XOR implementează operația logică binară **XOR (SAU EXCLUSIV)** și are sintaxa:

XOR dest , sursa

Operația **XOR** este realizată la nivel de bit, adică se efectuează **XOR** între fiecare bit al primului operand **dest** cu bitul corespunzător al celui de-al doilea operand **sursa**. Rezultatul este stocat în primul operand.

Operanzii pot fi regiștri, valori imediate sau valori din memorie cu restricțiile:

- Operandul **dest** nu poate fi valoare imediată.
- Ambii operanzi nu pot fi variabile din memorie.
- Operanzii ar trebui să fie de același tip (BYTE, SDWORD, etc).

Amintim că operația logică bit1 XOR bit2 are valoarea 0 dacă $\text{bit1}=\text{bit2}$ și valoarea 1 dacă $\text{bit1}\neq\text{bit2}$. De aceea, instrucțiunea **XOR** poate fi folosită pentru schimbarea valorii unui bit (bit toggle).

Exemplul 13. Instrucțiunea

XOR dest , 2

va schimba valoarea bitului de pe poziția 1 a lui **dest**, lăsându-i pe ceilalți pe loc.

De asemenea, **XOR** poate fi folosită pentru a inițializa un registru cu 0. Mai precis

XOR eax , eax

are ca efect încărcarea cu valoarea 0 a registrului **EAX**.

Instrucțiunea NOT implementează operația logică binară **NON (negație)** și are sintaxa:

NOT oper

Operația NON este realizată la nivel de bit, adică se efectuează negația fiecărui bit al operandului **oper** (biții 0 sunt transformați în 1 și biții 1 sunt transformați în 0). Rezultatul este stocat înapoi în operandul **oper**. Acesta poate fi registru sau valoare din memorie.

Instrucțiunea test are sintaxa

TEST oper1 , oper2

și efectuează operația de ȘI binar, fără modificarea operanzilor. Ea modifică doar indicatorii de stare din registrul EFLAGS și este urmată de obicei de un salt.

De exemplu, instrucțiunea

test EAX,1

va testa paritatea lui EAX fără distrugerea conținutului său.

9.2 Instrucțiuni de deplasare: shl, shr, sar, sal

Setul de instrucțiuni dispune de mai multe instrucțiuni de deplasare binară: shl, shr, sal, sar.

Sintaxa instrucțiunilor este

shl oper , nr_bit
shr oper , nr_bit
sal oper , nr_bit
sar oper , nr_bit

Operandul **oper** poate fi registru sau valoare din memorie, iar **nr_bit** este o valoare imediată (între 0 și (strict) dimensiunea operandului) sau registrul CL. Instrucțiunile funcționează astfel:

- Instrucțiunea **shl (shift left)** deplasează la stânga biții lui **oper** cu **nr_bit** poziții. Biții vacanți primesc valoarea 0.
- Instrucțiunea **shr (shift right)** deplasează la dreapta biții lui **oper** cu **nr_bit** poziții. Biții vacanți (de la stânga) primesc valoarea 0.
- Instrucțiunea **sal (shift arithmetic left)** efectuează deplasarea la stânga pentru numere cu semn. Biții vacanți primesc valoarea 0.
- Instrucțiunea **sar (shift arithmetic right)** efectuează deplasarea binară la dreapta pentru numere cu semn. Biții vacanți primesc valoarea semnului: 1 dacă operandul este negativ și 0 dacă este pozitiv.

În urma execuției acestor instrucțiuni, flag-ul CF se poate modifica, el reținând ultimul bit "pierdut" prin deplasare la stânga, respectiv dreapta.

Instrucțiunile **shl** și **sal** pot fi folosite pentru a înmulți rapid un operand cu puteri ale doi. Instrucțiunea **shr** are ca efect împărțirea unui operand fără semn la o putere a lui 2, iar **sar** efectuează împărțirea unui operand cu semn la o putere a lui 2 (păstrând semnul).

Exemplul 14. Presupunem că în registrul AL avem valoarea 1010 1110b, ce corespunde valorii zecimale fără semn 174d și valorii zecimale cu semn -82.

Instrucțiune	Rezultat în AL	Rezultat zecimal fără semn	Rezultat zecimal cu semn
shl AL,1	0101 1100b	92 (= 174 * 2 mod 256)	92 (= -82 * 2 mod 256)
shr AL,1	0101 0111b	87 (= 174/2)	87
sal AL,1	0101 1100b	92 (= 174 * 2 mod 256)	92 (= -82 * 2 mod 256)
sar AL,1	1101 0111b	215	-41 (= -82/2)

Observăm că **shr** nu calculează câtul corect la împărțirea cu 2 pentru valori negative, în timp ce **sar** nu calculează câtul corect la împărțirea cu 2 pentru valori fără semn (ale căror reprezentare binară începe cu 1; acestea corespund numerelor negative în reprezentarea *signed*).

9.3 Instrucțiuni de rotire: rol, rcl, ror, rcr

Setul de instrucțiuni (și asamblorul) dispune de mai multe instrucțiuni de rotire binară: **ror**, **rol**, **rcr**, **rcl**. Ele au sintaxa:

```
rol oper, nr_bits
ror oper, nr_bits
rcl oper, nr_bits
rcr oper, nr_bits
```

Operandul **oper** poate fi registru sau variabilă din memorie, iar **nr_bits** este o valoare imediată (de la 0 la 31) sau registrul **CL**. Instrucțiunile funcționează astfel:

- Instrucțiunea **rol** (rotate left) efectuează rotația la stânga a biților unui număr. Flag-ul CF va primi ultimul bit rotit în exterior de pe poziția cea mai semnificativă spre poziția 0.
- Instrucțiunea **ror** (rotate right) efectuează rotația la dreapta a biților unui număr. Flag-ul CF va primi ultimul bit rotit în exterior din poziția 0 spre cea mai semnificativă poziție.
- Instrucțiunea **rcl** (rotate through carry left) efectuează rotația la stânga a biților unui număr incluzând bitul CF în operație (pe poziția dominantă).
- Instrucțiunea **rcr** (rotate through carry right) efectuează rotația la dreapta a biților unui număr incluzând bitul CF în operație (la dreapta poziției 0).

Exemplul 15. Să presupunem că în registrul AL avem valoarea 1010 1110b și că CF=0. Atunci:

Instrucțiune	Rezultat în AL	CF
rol AL,1	0101 1101b	1
ror AL,1	0101 0111b	0
rcl AL,1	0101 1100b	1
rcr AL,1	0101 0111b	0

10 Moduri de adresare

Modul de adresare este procedeul prin care este specificat un operand al unei instrucțiuni. Acest mecanism influențează viteza de execuție a instrucțiunilor și determină lungimea acestora.

10.1 Adresarea imediată. Adresare de tip registru

În acest mod de adresare, operandul este o constantă explicită. Există totuși instrucțiuni care nu acceptă astfel de operanzi (de exemplu, **imul** sau **idiv**).

Exemplul 16. Instrucțiunea

```
mov rax, 30h
```

va transfera valoarea hexazecimală 30h (văzută ca un număr de 64 de biți) în registrul **rax**.

Adresarea de tip registru presupune ca operandul să fie conținut într-unul dintre regiștri. Foarte multe instrucțiuni cer ca unul dintre operanzi să fie registru. De aceea, este un mod de adresare folosit foarte des. Dezavantajul este totuși numărul relativ restrâns de regiștri.

Exemplul 17. Instrucțiunea

```
add rax, rbx
```

are ca operanzi regiștrii **rax** și **rbx** și returnează în **rax** suma valorilor conținute în **rax** și **rbx**.

10.2 Adresarea indirectă

Adresarea indirectă implică accesarea valorii unui operand prin intermediul adresei sale.

Un exemplu important este dat de variabilele din segmentul de date. Considerăm segmentul de date următor:

```
.data
var64  sqword 6
var32  sdword 12
```

Simbolul `var64` este interpretat de asamblor ca o adresă (mai precis, un deplasament față de începutul segmentului). Instrucțiunea

```
mov rax, var64
```

va încărca

De exemplu, dacă segmentul de date începe cu Prin acest mod de adresare, operandul este dat prin intermediul adresei sale effective. Această adresă se poate exprima printr-o valoare imediată sau un nume simbolic.

Exemplul 18. De exemplu, dacă în segmenul de date avem declarată o variabilă

```
a SDWORD 10,15
```

instrucțiunea

```
mov eax, [a]
```

va încărca registrul `eax` cu valoarea 10.

10.3 Adresare indirectă

Modul indirect de adresare presupune accesarea unei valori prin intermediul adresei sale din memorie. Schema generală de adresare este:

$$[\text{base_addr64} + \text{index_reg64} * \text{scale} + \text{const}]$$

unde

- `base_addr64` este o variabilă sau un registru de 64 de biți.
- `index_reg64` este un registru de 64 de biți.
- `scale` este o constantă de scalare cu valorile 1,2,4 sau 8.
- `const` este un număr întreg.

La întâlnirea unei astfel de expresii, asamblorul va realiza calculul de adrese corespunzător și va returna conținutul adresei finale.

Exemplul 19. Presupunem că în segmentul de date avem următoarea declarație:

```
tabel sdword 10, 15, 20
      sdword 25, 30, 35
```

- Instrucțiunile

```
mov eax, [tabel]
mov edx, [tabel+4]
```

vor încărca registrul `eax` cu valoarea primului element din `tabel` (adică 10), iar `edx` cu a doua valoare (adică 15). Aceasta se întâmplă deoarece fiecare element din tabel ocupă 32 de biți (de tip `sdword`) și astfel ocupă patru adrese de memorie.

În instrucțiuni, simbolul `tabel` este interpretat ca o adresă (mai precis, un deplasament sau o distanță în octeți față de începutul segmentului de date). Începând cu respectiva adresă sunt scrise numerele din listă.

Astfel, la adresele `tabel+0, ..., tabel+3` se află primul element, 10, la adresele `tabel+4, ..., tabel+7` se află al doilea element, 15, șamd.

Atragem atenția totuși că stocarea în memorie a unei valori se face folosind convenția little endian, adică prima adresă conține ultimul octet al valorii, a doua conține cel de-al doilea octet etc.

De exemplu, reprezentarea hexazecimală a lui 10 este 00.00.00.0a, iar reprezentarea în memorie va fi

tabel+0	tabel+1	tabel+2	tabel+3
0a	00	00	00

- Instrucțiunile

```
lea rbx, tabel
mov eax, [rbx]
mov rsi, 3
mov edx, [rbx+rsi *4]
```

vor încărca **rax** cu valoarea 10 (primul element al matricei) și **rdx** cu valoarea 25 (conținutul adresei situată la 24 de octeți de început).

- Expresia **tabel+2** nu adresează un element al matricei. Aceasta se referă la octetul situat la 2 unități distanță de începutul matricei. Primele 8 de adrese ale acestei variabile conțin

tabel:0a.00.00.000f.00.00.00

deci primii 4 octeți care încep de la **tabel+2** sunt 00.00.0f.00, care nu reprezintă un element al matricei.

11 Structuri condiționale și iterative. Controlul fluxului de instrucțiuni

Instrucțiunile de salt transferă controlul către instrucțiunea aflată la o adresă specificată. Ele sunt de două tipuri: necondiționate și condiționate și sunt folosite pentru a implementa structuri condiționale (if-then-else) și iterative (while, do-while, for).

11.1 Salt necondiționat

Instrucțiunea **JMP** realizează un salt necondiționat la o adresă specificată. Ea are sintaxa

JMP *tinta*

unde *tinta* este eticheta destinației saltului. Această etichetă este de fapt o adresa relativă, mai precis deplasamentul (offset-ul) dintre instrucțiunea țintă și instrucțiunea care urmează după salt.

11.2 Salt condiționat

Un salt condiționat are doi pași. În primul rând, este testată condiția. Apoi, dacă aceasta este îndeplinită se realizează saltul, iar dacă nu, se trece la executarea instrucțiunii care urmează saltului. Condiția testată se referă la unul sau mai mulți indicatori de stare din registrul RFLAGS. Sintaxa generală este

J<conditie> *tinta*

unde *tinta* este eticheta destinației saltului iar *<conditie>* este reprezentată prin literele:

Literă	Semnificație
G	greater than (pentru numere cu semn)
L	less than (pentru numere cu semn)
A	above (pentru numere fără semn)
B	below (pentru numere fără semn)
E	equal
N	not
O	overflow
C	carry
Z	zero
S	sign

Modificarea indicatorilor de stare se poate face ușor folosit instrucțiunea **cmp** ce are sintaxa

cmp *oper1* , *oper2*

Aceasta compară cei doi operanzi prin efectuarea scăderii **oper1-oper2** și modificând corespunzător indicatorii de stare. Operanzii pot fi valori imediate, regiștri sau variabile din memorie, însă nu pot fi ambii variabile din memorie sau ambii valori imediate.

Prezentăm în continuare o listă a celor mai frecvente instrucțiuni de salt bazate pe comparații:

- instrucțiuni de salt ca urmare a unei comparații de variabile fără semn (de exemplu, caractere)

Instrucțiune	Condiție de salt	Flag-uri testate
JE	=	$ZF = 1$
JNE	$\neq$	$ZF = 0$
JA sau JNBE	>	$ZF = 0$ și $CF = 0$
JB sau JNAE	<	$CF = 1$
JAЕ sau JNB	$\geq$	$CF = 0$
JBE sau JNA	$\leq$	$ZF = 1$ sau $CF = 1$

- instrucțiuni de salt ca urmare a unei comparații de variabile cu semn (de exemplu, numere întregi)

Instrucțiune	Condiție de salt	Flag-uri testate
JE	=	$ZF = 1$
JNE	$\neq$	$ZF = 0$
JG sau JNLE	>	$ZF = 0$ și $SF = OF$
JL sau JNGE	<	$SF \neq OF$
JGE sau JNL	$\geq$	$SF = OF$
JLE sau JNG	$\leq$	$ZF = 1$ sau $SF \neq OF$

- instrucțiuni de salt bazate direct pe flag-uri

Instrucțiune	Condiție de salt
JO	$OF = 1$
JNO	$OF = 0$
JC	$CF = 1$
JNC	$CF = 0$
JZ	$ZF = 1$
JNZ	$ZF = 0$
JS	$SF = 1$
JNS	$SF = 0$
JRCXZ	$RCX = 0$

(ultima instrucțiune nu testează un indicator de condiție, ci registrul RCX).

Pe lângă salturi, există și instrucțiuni iterative directe. Principala instrucțiune de ciclare este `loop` cu sintaxa

`loop eticheta`

Aceasta decrementează mai întâi conținutul registrului RCX, apoi testează dacă acesta este 0. Saltul la `eticheta` se efectuează numai când conținutul este nenul.

Așadar, `loop eticheta` poate fi înlocuită cu

`dec RCX`
`jnz eticheta`

Surprinzător, instrucțiunea `loop` este mai înceată decât combinația `dec/jnz`. Însă, `loop` face codul mai ușor de citit.

12 Instrucțiuni pentru numere reale

Numerele reale pot fi declarate folosind directivele `real4` și `real8`. De exemplu, în codul următor

```
.data
num_f real4 6.4
num_d real8 -125.64
```

sunt declarate două variabile:

- `num_f`, un număr în virgulă flotantă cu precizie simplă (pe 32 de biți = 4 octeți) cu valoarea 6.4.
- `num_d`, un număr în virgulă flotantă cu precizie dublă (pe 64 de biți = 8 octeți) cu valoarea -125.64.

Operațiile cu numere reale vor folosi regiștrii `xmm`. Aceștia sunt în număr de 16: `xmm0`, `xmm1`, ..., `xmm15` și au fiecare o dimensiune de 128 de biți.

Un registru `xmm` poate conține patru numere de 32 de biți (sau două de 64 de biți) și există instrucțiuni care acționează pe toate aceste patru componente. Pentru simplitate, vom lucra doar cu câte un număr inserat în acești regiștri.

Transferul de memorie cu numere reale se face folosind instrucțiunile

```
movss dest, sursa; operandi de 32 de biți
movsd dest, sursa; operandi de 64 de biți
```

Operandii au următoarele restricții:

- Operandii trebuie să aibă aceeași dimensiune.
- Operandii nu sunt valori imediate.
- Unul dintre operandi trebuie să fie un registru `xmm`.

Exemplul 20. `movss xmm0, x`
`movsd xmm1, xmm2`
`movsd x, xmm0`

Procesorul dispune și de instrucțiuni de transfer de la regiștrii `xmm` la regiștrii generali:

```
movd reg32, xmm_n
movd xmm_n, reg32
movq reg64, xmm_n
movq xmm_n, reg64
```

Acestea sunt utile pentru apelul unor funcții de tipul `printf` care cer ca argumentele să fie în regiștri generali.

12.1 Operații aritmetice cu numere reale

1. Instrucțiunile pentru adunare au sintaxa:

```
addss xmm_n, sursa; sursa are 32b
addsd xmm_n, sursa; sursa are 64b
```

și realizează operația:

$$\text{xmm_n} \leftarrow \text{xmm_n} + \text{sursa}$$

unde `xmm_n` este un registru `xmm`, iar `sursa` este un registru `xmm` sau o variabilă din memorie (nu poate fi valoare imediată).

Exemplul 21. `.data`
`fdnum1 real8 1.5`
`fdnum2 real8 3.5`
`fdsum real8 ?`
`.code`
`movsd xmm0, fdnum1`
`addsd xmm0, fdnum2`
`movsd fdsum, xmm0 ; fdsum = 1.5 + 3.5 = 5.0`

2. Instrucțiunile pentru scădere au sintaxa:

```
subss xmm_n, sursa; sursa are 32b
subsd xmm_n, sursa; sursa are 64b
```

și realizează operația:

$$\text{xmm_n} \leftarrow \text{xmm_n} - \text{sursa}$$

unde `xmm_n` este un registru `xmm`, iar `sursa` este un registru `xmm` sau o variabilă din memorie (nu poate fi valoare imediată).

Exemplul 22. .data

```

fdnum1 real8 1.5
fdnum2 real8 3.5
fddif real8 ?
.code
movsd xmm0, fdnum1
subsd xmm0, fdnum2
movsd fddif, xmm0 ; fddif = 1.5 - 3.5 = -2.0

```

3. Instrucțiunile pentru multiplicare au sintaxa:

```

mulss xmm_n, sursa ; sursa are 32b
mulsd xmm_n, sursa ; sursa are 64b

```

și realizează operația:

$$\text{xmm_n} \leftarrow \text{xmm_n} * \text{sursa}$$

unde `xmm_n` este un registru `xmm`, iar `sursa` este un registru `xmm` sau o variabilă din memorie (nu poate fi valoare imediată).

Exemplul 23. .data

```

fdnum1 real8 1.5
fdnum2 real8 3.5
fdprod real8 ?
.code
movsd xmm0, fdnum1
mulsd xmm0, fdnum2
movsd fdprod, xmm0 ; fdprod = 1.5 * 3.5 = 5.25

```

4. Instrucțiunile pentru împărțire au sintaxa:

```

divss xmm_n, sursa ; sursa are 32b
divsd xmm_n, sursa ; sursa are 64b

```

și realizează operația:

$$\text{xmm_n} \leftarrow \text{xmm_n} / \text{sursa}$$

unde `xmm_n` este un registru `xmm`, iar `sursa` este un registru `xmm` sau o variabilă din memorie (nu poate fi valoare imediată).

Exemplul 24. .data

```

fdnum1 real8 1.75
fdnum2 real8 3.5
fddiv real8 ?
.code
movsd xmm0, fdnum1
divsd xmm0, fdnum2
movsd fddiv, xmm0 ; fddiv = 1.75 / 3.5 = 0.5

```

5. Instrucțiunile pentru radical au sintaxa:

```

sqrtss xmm_n, sursa ; sursa are 32b
sqrtsd xmm_n, sursa ; sursa are 64b

```

și realizează operația:

$$\text{xmm_n} \leftarrow \sqrt{\text{sursa}}$$

unde `xmm_n` este un registru `xmm`, iar `sursa` este un registru `xmm` sau o variabilă din memorie (nu poate fi valoare imediată).

Exemplul 25. .data

```

fdnum real8 0.16
.code
sqrtsd xmm0, fdnum ; xmm0 = sqrt(0.16) = 0.4

```

12.2 Instrucțiuni condiționale cu numere reale

Pentru salturi condiționale folosind variabile reale se pot folosi instrucțiunile:

```
ucomiss xmm_n, oper ; (operanzi 32 biti)
ucomisd xmm_n, oper ; (operanzi 64 de biti)
```

unde `xmm_n` este un registru `xmm`, iar `oper` este un registru `xmm` sau din memorie.

Aceste instrucțiuni calculează diferența celor doi operanzi (fără să le modifice valorile) și modifică flaguri corespunzătoare în `RFLAGS` (similar cu instrucțiunea `cmp`).

Instrucțiunile de salt condiționat sunt

Nume	Descriere	Nume	Descriere
ja	jump if above ($>$)	jae	jump if above or equal ($\geq$)
jb	jump if below ($<$)	jbe	jump if ($\leq$)
je	jump if equal ($==$)	jne	jump if not equal ($\neq$)

Acestea funcționează similar cu cele pentru variabile întregi.

13 Stiva. Proceduri

Stiva este o structură de tip LIFO (last in, first out) ce permite două operații inserare (push) și ștergere (pop). Singurul element direct accesibil dintr-o stivă este vârful stivei. Stiva este folosită de procesor pentru a urmări apelurile funcțiilor (procedurilor), pentru transmiterea parametrilor și reținerea variabilelor locale ale acestora.

Pentru a implementa stiva, procesorul `x86-64` folosește regiștrii `RSP` și `RBP`. Principalele caracteristici ale stivei procesorului sunt:

- pot fi salvate pe stivă doar valori `WORD` (16 biți) sau `QWORD` (64 de biți). Nu pot fi adăugate valori `DWORD` sau `BYTE`. (procesorul dispune de două moduri de operare: modul real în care adresele au 16 biți și modul protejat, cu adrese de 64 de biți; stiva reflectă acest lucru).
- stiva crește spre adrese mai mici, adică adăugarea pe stivă se face dinspre adrese mari spre adrese mici.
- Registrul `RSP` conține permanent adresa vârfului stivei (ultimul element inserat), mai precis cel mai puțin semnificativ octet al acestuia (datorită convenției little endian).

Pentru simplitate, pe stivă vom plasa doar operanzi de 64 de biți.

Principalele instrucțiuni de lucru cu stiva sunt instrucțiunile `push` și `pop`.

Instrucțiunea `push` are sintaxa

```
push sursa
```

unde operandul `sursa` este număr întreg de 64 de biți (registru, memorie sau valoare imediată). Efectul acesteia este inserarea pe stivă a operandului `sursa`. Aceasta înseamnă că:

- `RSP` este decrementat cu 8 (lungimea operandului `sursa`).
- La adresa conținută în `RSP` este încărcat operandul `sursa`.

Instrucțiunea `pop` are sintaxa

```
pop dest
```

unde operandul `dest` este registru sau variabilă din memorie. Prin această instrucțiune este extras vârful stivei și încărcat în operandul `dest`. Această înseamnă că:

- Operandul `dest` primește valoarea de la adresa `RSP` (8 octeți).
- `RSP` este incrementat cu 8 (lungimea `dest`).

Există și câteva instrucțiuni specializate pentru stivă:

- `pushfq` salvează pe stivă conținutul registrului `RFLAGS`.
- `popfq` extrage de pe stivă conținutul registrului `RFLAGS`.

Prima aplicație simplă a utilizării stivei este pentru stocarea temporară a datelor.

Exemplul 26. Instrucțiunea `imul` distruge conținutul registrului `RDX`. Pentru a îi păstra valoarea putem folosi secvența de cod:

```
push rdx
imul ecx
pop rdx
```

La final, `RDX` va avea aceeași valoare ca la început (deși instrucțiunea `imul` modifică registrul `EDX`).

Exemplul 27. Similar, `printf` distruge regiștrii `RAX`, `RCX`, `RDX`. Dacă vrem să le salvăm putem folosi o secvența de cod de forma:

```
push rax
push rcx
push rdx
— apel printf —
pop rdx
pop rcx
pop rax
```

(extragerea trebuie făcută în ordinea inversă inserării).

Exemplul 28. Putem folosi stiva pentru a inversa două valori din memorie. De exemplu dacă vrem să schimbăm între ele valorile variabilelor `val1` și `val2` putem proceda astfel:

```
push val1
push val2
pop val1
pop val2
```

Una din cele mai importante utilizări ale stivei este în transferul controlului către proceduri sau funcții, precum și în transmiterea a parametrilor unei funcții sau proceduri. Sintaxa declarării unei proceduri este

```
nume PROC
<instrucțiuni procedura>
ret
nume ENDP
```

unde:

- Eticheta `nume` indică numele procedurii.
- Directivele `PROC` și `ENDP` marchează începutul și finalul codului procedurii.
- Instrucțiunea `ret` transferă controlul înapoi la funcția care apelează procedura (întoarcere la apelant).

Apelul unei proceduri în funcția apelantă se face prin instrucțiunea `call` cu sintaxa:

```
call nume
```

Asamblorul înlocuiește numele procedurii `nume` nu cu adresa efectivă a începutului procedurii (prima instrucțiune după antet), ci cu deplasamentul relativ (în octeți) fața de poziția instrucțiunii `call` în segmentul de cod. Acesta poate fi număr pozitiv sau negativ.

Instrucțiunea `call` execută următorii pași:

- Este adăugată pe stivă valoarea din registrul `RIP` (ce conține adresa instrucțiunii de după `call`, în funcția apelantă):

$$RSP = RSP - 8$$

$$RSP = RIP$$

- `RIP` primește adresa primei instrucțiuni din funcția apelată. Mai precis: se adună la `RIP` deplasamentul relativ până la eticheta procedurii:

$$ERP := RIP + \text{deplasamentrelativ}$$

Astfel, instrucțiunea care va fi executată după `call` este prima instrucțiune a procedurii apelate.

Procesul invers, de ieșirea din apelat și întoarcere la apelant, se face folosind instrucțiunea **ret**. Acesta extrage de pe stivă conținutul RIP salvat prin **call**. Mai precis, **ret** execută următorii pași:

```
RIP = [RSP]
RSP = RSP + 8
```

De obicei, procedurile au parametri (argumente). Cea mai simplă metodă de a transmite parametri este prin intermediul regiștrilor generali, ce pot fi văzuți ca variabile globale disponibile tuturor programelor. Această metodă este utilă pentru transmiterea unui număr mic de parametri și este foarte rapidă.

Pe de altă parte, există destul de puțini regiștri generali, reducând numărul de parametri ce pot fi utilizați. De asemenea, acești regiștri pot fi folosiți și de procedură însăși (de exemplu în operații aritmetice), sau poate avem un lanț de proceduri ce se apelează una pe alta. Aceasta face "contabilitatea" regiștrilor să fie anevoioasă.

Exemplul 29. Următorul program calculează suma a două numere folosind o procedură cu argumente transmise prin regiștri. Rezultatul este returnat în **eax**.

```
...
main PROC
    mov ecx, 5
    mov ebx, 7
    call sum
    ret
main ENDP
sum PROC
    mov eax, ebx
    add eax, ecx
    ret
sum ENDP
...
```

A doua metodă de transmitere a parametrilor este realizată folosind stiva. Mai precis, argumentele necesare sunt inserate pe stivă înaintea apelului și sunt luate direct de pe stivă în cadrul procedurii. La final, stiva trebuie curățată: toate argumentele inserate trebuie eliminate. Vârful stivei este mutat înainte de inserarea parametrilor.

Responsabilitatea curățării stivei revine sau apelantului sau apelatului. Vom folosi convenția **cdecl** de apel a funcțiilor. Argumentele se transmit pe stivă în ordine inversă (de la ultimul, la primul), iar responsabilitatea curățării stivei revine **apelantului** (funcția care execută **call**). Registrul **RAX** va conține, de obicei, valoarea returnată.

Astfel, o procedură **FCT** cu doi parametri **SQWORD** va fi apelată cu o secvență de forma:

```
push param2
push param1
call FCT
add rsp, 16
```

Stiva după **call** va avea forma:

RSP_original +0	param2	
-8	param1	
-16	RIP_for_ret	←RSP_curent

Procedura **FCT** va adresa cei doi parametri **param1** și **param2** prin **[RSP+8]**, respectiv **[RSP+16]**.

Acest mod de transmitere a parametrilor ridică totuși o problemă. Pointerul **RSP** se modifică la fiecare instrucțiune **push/pop/call**. Dacă o procedură conține astfel de instrucțiuni, urmărirea parametrilor prin intermediul **RSP** poate deveni anevoioasă (de exemplu, un argument adresat prin **[RSP+8]** poate deveni mai târziu **[RSP+16]**).

În această situație se folosește registrul **RBP**. El poartă numele de **Stack Frame Pointer** și rămâne constant pe parcursul procedurii. Argumentele și variabilele locale sunt adresate în funcție de **RBP**.

Astfel, vom începe procedurile cu secvența de cod:

```
push rbp
mov rbp, rsp
```

(salvăm valoarea originală a lui RBP), iar la final, înainte de `ret`, restaurăm valoarea RBP:

```
mov rsp, rbp
pop rbp
```

Argumentele pot fi găsite la adresele `rbp+16`, `rbp+24`, `rbp+32`,...

Variabilele locale pot fi alocate tot pe stivă. De exemplu, dacă vrem să alocăm o variabilă locală de tip `SQWORD`, începem procedura cu:

```
push rbp
mov rbp, rsp
sub rsp, 8
```

Variabila locală poate fi adresată prin `[RBP-8]`. La finalul procedurii adăugăm:

```
mov rsp, rbp
push rbp
```

Exemplul 30. Următorul program calculează suma a două numere și returnează rezultatul în `RAX` transmițând argumentele pe stivă:

```
main PROC
    push num2
    push num1
    call sum
    add rsp, 16
    ret
main ENDP
sum PROC
    push rbp
    mov rbp, rsp
    mov rax, [rsp+16]
    add rax, [rsp+24]
    mov rsp, rbp
    pop rbp
    ret
sum ENDP
END
```

Exemplul 31. Următorul program calculează suma $1 + 2 + \dots + n$ folosind o procedură cu argumentul transmis pe stivă și o variabilă locală.

```
main PROC
    push n
    call sum
    add rsp, 8
    ret
main ENDP
sum PROC
    push rbp
    mov rbp, rsp
    sub rsp, 8
    mov rax, 0
    mov rcx, [rbp+16]
    mov QWORD PTR [rbp-8], 1
loop01:
    add rax, [rbp-8]
    inc QWORD PTR [rbp-8]
    loop loop01
    mov rsp, rbp
    pop rbp
    ret
sum ENDP
END
```

14 Template-uri if

Această secțiune conține câteva template-uri pentru structuri if si while. Condiția testată într-un if este de obicei o comparație între două valori (a=b, a<b, etc.)

Vom presupune pentru simplitate ca valorile comparate se afla în `eax` si `ebx`. Implementarea if se va realiza după schemele următoare:

1. Structura if fara ramura `else`

```
;if fara else
cmp eax,ebx
<jump if (conditia este falsa)> la adresa end_if01
<instructiuni din if>
end_if01:
<instructiuni dupa if>
```

2. Structura if cu ramura `else`.

```
;if-else
cmp eax,ebx
<jump if (conditia este falsa)> la adresa else01
<instructiuni din if>
jmp endif01
else01:
<instructiuni din ramura else>
endif01:
<instructiuni dupa if>
```

Exemplul 32. Pseudocod:

```
if    (eax=ebx)    executa    ebx=ebx+1
```

Cod asamblare:

```
cmp eax,ebx
jne endif01    ;poate fi folosit si jnz
inc ebx
endif01:    <instructiuni_dupa if>
```

Exemplul 33. Pseudocod:

```
if    (eax!=ebx)    executa    ebx=ebx+1
```

Cod asamblare:

```
cmp eax,ebx
je    endif02
inc ebx
endif02:    <instructiuni_dupa if>
```

Exemplul 34. Pseudocod:

```
if    (eax>ebx)    executa    ebx=ebx+1
```

Cod asamblare:

```
cmp eax,ebx
jle    endif03
inc ebx
endif03:    <instructiuni_dupa if>
```

Exemplul 35. Pseudocod:

```
if    (eax<=ebx)    executa    ebx=ebx+1
```

Cod asamblare:

```
        cmp  eax,ebx
        jg   endif04
        inc  ebx
endif04:    <instructiuni_dupa if>
```

Exemplul 36. Pseudocod:

```
if (eax=ebx)    executa    ebx=ebx+1
else           executa    eax=eax+1
```

Cod asamblare:

```
        cmp  eax,ebx
        jnz  else05
        inc  ebx
        jmp  endif05
else05:    inc  eax
endif05:    <instructiuni_dupa if>
```

Alternativ, putem inversa ramurile if-else:

```
        cmp  eax,ebx
        jz   then05
        inc  eax
        jmp  endif05
then05:    inc  eax
endif05:    <instructiuni_dupa if>
```

Exemplul 37. Pseudocod:

```
if    (eax<ebx)    executa    ebx=ebx+1
else           executa    eax=eax+1
```

Cod asamblare:

```
        cmp  eax,ebx
        jge  else06
        inc  ebx
        jmp  endif06
else06:    inc  eax
endif06:    <instructiuni_dupa if>
```

15 Template-uri while

Această secțiune conține câteva template-uri pentru structuri **while**. Condiția de ieșire din buclă este de obicei o comparație a două valori ($a=b$, $a<b$, etc.). Vom presupune, pentru simplitate, ca valorile sunt una conținută în **eax** și cealaltă o constantă **const** (**eax** este contorul buclei).

Atentie: Sunt valabile aceleași observații ca și pentru instrucțiunile **while** și **do-while** din C:

- **eax** trebuie inițializat înainte de intrarea în buclă cu prima valoare ce trebuie testată.
- printre ultimele instrucțiuni din buclă ar trebui să se afle o modificare a lui **eax** (incrementare, dublare, etc.).
- într-o structură **while** verificarea condiției se face la început; dacă ea nu este satisfăcută buclă este sarită.
- într-o structură **do-while** verificarea condiției se face la final; dacă ea nu este satisfăcută programul "cade" din buclă.

Schema generala a unei structuri while va fi:

1. **while** (conditie) executa <instructiuni din while>

```
<initializarea lui eax>
while01:
cmp eax,const
<jump if (conditia este falsa)> endwhile01
<instructiuni din while>
jmp while01
endwhile01:
<instructiuni dupa while>
```

2. **do** <instructiuni din do-while> **while** (conditie)

```
<initializarea lui eax>
do-while01:
<instructiuni din do-while>
cmp eax,ebx
<jump if (conditia este adevarata)> do-while01
<instructiuni dupa do-while>
```

Instrucțiunea 'for' din C este de obicei implementata printr-o structura **while** sau **do-while** (codul este mai eficient). Exista insa instrucțiunea **loop** ce este recomandata pentru executia unui numar fix de bucle.

Exemplul 38. Pseudocod:

```
eax=1;
while    (eax<3)    executa    eax=eax+1
```

Cod asamblare:

```
while01:    mov eax,1
            cmp eax,3
            jge end-while01
            <instructiuni>
            inc eax
            jmp while01
end-while01: <instructiuni dupa while>
```

Exemplul 39. Pseudocod:

```
ebx=val_initiala;
while    (ebx!=0)    executa    ebx=ebx-1
```

Cod asamblare:

```
while02:    mov ebx,val_initiala
            cmp ebx,0
            jz  end_while02
            dec eax
            jmp while02
end_while02: <instructiuni dupa while>
```

Exemplul 40. Pseudocod:

```
eax=val_initiala
do    eax=eax-1    while(eax>0)
```

Cod asamblare:

```

do_while03:      mov eax, val_initiala
                  <instructiuni>
                  dec eax
                  cmp eax, 0
                  jle do_while03
                  <instructiuni dupa do-while>

```

Exemplul 41. Pseudocod:

```

ebx=val_initiala
do    ebx=ebx+4    while([ebx]!=0)

```

Cod asamblare:

```

do_while04:      mov ebx, val_initiala
                  add ebx, 4
                  mov eax, [ebx]
                  cmp eax, 0
                  jnz do_while04
                  <instructiuni dupa do-while>

```

Exemplul 42. Pseudocod:

```

pentru ebx=1,...,n    executa    <instructiuni>

```

Cod asamblare:

```

loop01:          mov ecx, n
                  mov ebx, 1
                  <instructiuni>
                  inc ebx
                  loop loop01

```

Exemplul 43. Pseudocod:

```

pentru ecx=n,...,1    executa    <instructiuni>

```

Cod asamblare:

```

loop02:          mov ecx, n
                  <instructiuni>
                  loop loop02

```

16 Template procedura

- Toate procedurile vor avea forma:

```

functie PROC
push rbp
mov rbp, rsp
...
<instructiuni procedura>
...
mov rsp, rbp
pop rbp
ret
functie ENDP

```

îl distrugem).

- Apelul din funcția apelantă (presupunem că funcția are doi parametri de tip `sqword`) va avea forma:

```
...  
push arg2  
push arg1  
call functie  
add rsp,16 ; 8*(numar argumente adaugate pe stiva)  
...
```

- Argumentele (inseate pe stivă de funcția apelantă) vor fi adresate prin [rbp+16], [rbp+24],...
- Variabilele locale sunt alocate tot pe stivă, prin decrementarea RSP cu lungimea totală a tuturor noilor variabile.

17 Exemple

Exemplul 44. Următorul program calculează diverse operații aritmetice cu numere întregi.

```
1  ;programul calculeaza suma, diferenta ,
2  ;produsul si catul a doua numere intregi
3  ;initializate ca variabile globale
4  includelib libcmtd.lib
5  includelib legacy_stdio_definitions.lib
6  printf proto
7  scanf  proto
8  .data
9  fmt BYTE "sum=%lld",0Ah,"dif=%lld",0Ah, "prod=%lld",0Ah,"quot=%lld",0Ah,"rem=%lld",0Ah,0
10 x SQWORD 25 ;x:= 16
11 y SQWORD 10 ;y:=10
12 sum SQWORD ? ; sum, difference,... sunt alocate dar neinitializate
13 difference SQWORD ?
14 product SQWORD ?
15 quotient SQWORD ?
16 remainder SQWORD ?
17 .code
18 main proc
19 ; add op1,op2 efectueaza op1:=op1+op2
20 ; sub op1,op2 efectueaza op1:=op1-op2
21
22 mov rax,x ;rax := x
23 add rax,y ;rax := rax+y = x+y
24 mov sum,rax ;sum := rax
25
26 mov rax,x
27 sub rax,y
28 mov difference,rax; difference = x-y
29
30 ; imul op efectueaza (rdx:rax) := rax * op
31 ; rezultatul inmultirii este calculat pe 128 de biti
32 ; ultimii 64 de biti sunt stocati in rax
33 ; rdx primeste bitii din rezultat care 'depasesc' 64 de biti
34 ; daca inmultirea nu genereaza overflow, rdx=0.
35 mov rax,x ;rax := x
36 imul y ;(rdx:rax) := rax * y = x*y
37 mov product,rax ;product := rax
38
39
40 mov rax,x
41 cqo
42 ;cqo extinde bitul de semn din rax in (rdx:rax)
43 ;se efectueaza conversia lui rax (64 bit) in (rdx:rax) (128 biti)
44 ;idiv y efectueaza (rdx:rax) / y (impartim 128 biti la 64 de biti )
45 ;catul este plasat in rax
46 ;restul este plasat in rdx
47 idiv y
48 mov quotient,rax
49 mov remainder,rdx
50
51 ;printf("sum=%lld \n dif=%lld \n prod= %lld \n quot=%lld rem=%lld ",
52 ; sum,difference,product,quotient,remainder);
53 ;se afiseaza pe cate o linie valorile sum, difference,...
54 ;argumentele printf se transmit in ordine in rcx,rdx,r8 si r9
55 ;urmatoarele argumente sunt transmise pe stiva in ordine inversa
56 ;conventia Windows ABI cere ca pt primele 4 argumente, sa se aloce spatiu pe stiva
57 ;stiva trebuie aliniata la un multiplu de 16.
```

```

58 sub rsp,8
59 push remainder      ;plasam pe stiva ultimele doua argumente
60 push quotient
61 sub rsp,32
62 lea rcx,fmt      ;rcx := adresa stringului de format
63 mov rdx, sum      ;rdx := al doilea arg. al printf
64 mov r8, difference
65 mov r9, product
66 call printf
67 add rsp,56
68 ret
69 main endp
70 END

```

Operațiile aritmetice se efectuează așa cum au fost descrise anterior. O dificultate apare însă când vrem să folosim `printf` pentru afișare.

Convenția Windows ABI are următoarele cerințe:

- Primele patru argumente sunt (în ordine) transmise prin `RCX`, `RDY`, `R8`, `R9`.
- Începând cu al cincilea, argumentele se transmit funcției prin stivă (în ordine inversă).
- Înainte de `call`, este necesară alocarea unui `shadow space` de (cel puțin) 32 de octeți pe stivă (pentru primele 4 argumente).
- Stiva trebuie aliniată la 16 octeți, i.e. toate alocările pe stivă (`shadow space`, argumente și `call`) trebuie să însumeze un multiplu de 16.
- De exemplu, dacă `printf` are două argumente, ele trebuie transmise în `RCX` și `RDY`, iar pe stivă trebuie alocați 40 de octeți (astfel încât împreună cu instrucțiunea `call` stiva să fie ocupată cu 48 de octeți, multiplu de 16).

În cazul nostru, `printf` are 6 argumente. Primele patru sunt transmise prin regiștri (`rcx`, `rdx`, `r8`, `r9`), iar ultimele două prin stivă.

Înainte de `call` (liniar 61), trebuie păstrați 32 de octeți pe stivă (`shadow space`). Astfel, pe stivă avem alocați $32 + 16 = 48$ octeți. La apelul `call` se mai adaugă 8, ceea ce înseamnă că pe stivă avem alocați 56 de octeți.

Dar 56 nu este multiplu de 16, deci stiva este nealiniată. Pentru a preveni aceasta înainte de începutul apelului `printf` (linia 59), alocăm încă 8 octeți, pentru aliniere.

Exemplul 45. Următorul program citește trei numere și calculează o operație aritmetică cu ele, afișând un mesaj de eroare în caz de împărțire la 0.

```

1 ;citim de la tastatura x,y,z
2 ;calculam rez= (3z+2) / (x-y)
3 ;calculam rez2 = (3z+2) mod (x-y)
4 ;afisam mesaj de eroare daca nu putem efectua calculul.
5 includelib libcmt.lib
6 includelib legacy_stdio_definitions.lib
7 printf proto
8 scanf proto
9 .data
10 msg_x byte "Dati-x:",0
11 msg_y byte "Dati-y:",0
12 msg_z byte "Dati-z:",0
13 fmt_read byte "%lld",0
14
15 fmt_afisare BYTE "cat=%lld , -rest=%lld",0ah,0
16 error_msg byte "Impartire - la -0",0ah,0
17
18 x SQWORD ?
19 y SQWORD ?
20 z SQWORD ?
21 quot SQWORD ?

```

```

22 rem SQWORD ?
23 .code
24 main proc
25 ;citim cu prompt x
26 ;printf("Dati x:")
27 ;arg. printf este transmis prin rcx (pointer la caracter)
28 sub rsp,40
29 lea rcx,msg_x
30 call printf
31
32 ;scanf("%lld",&x)
33 ;primul argument este transmis in rcx (pointer la caracter)
34 ;al doilea este adresa lui x, in care vom depune rezultatul citirii
35 lea rcx,fmt_read
36 lea rdx,x
37 call scanf
38
39 ;citim cu prompt y
40 lea rcx,msg_y
41 call printf
42
43 lea rcx,fmt_read
44 lea rdx,y
45 call scanf
46
47 ;citim cu prompt z
48 lea rcx,msg_z
49 call printf
50
51 lea rcx,fmt_read
52 lea rdx,z
53 call scanf
54 add rsp,40
55
56 mov rax,3
57 imul z ; rax=3*z
58 sub rax,2 ; rax=3*z - 2 ->deimpartitul
59
60 mov rbx,x
61 sub rbx,y ;rbx = x-y ->impartitorul
62
63 ;testam daca impartitorul este zero
64 ;daca da afisam mesaj de eroare
65
66 cmp rbx,0
67 jne calc_expr
68 ;daca rbx=0, afisam mesaj de eroare
69 sub rsp,40
70 lea rcx,error_msg
71 call printf
72 add rsp,40
73 jmp final
74 calc_expr:
75 cqo ; pregatim deimpartitul; extindem bitul de semn din rax
76 idiv rbx ; rax = rax/rbx si rdx = rax mod rbx
77
78 ;afisam rezultatele
79 mov quot,rax
80 mov rem, rdx
81
82 ;printf("cat=%lld , rest= %lld",quot,rest)

```

```

83 ;rcx primeste adresa stringului format
84 ;rdx si r8 preiau celelalte argumente
85
86 sub rsp,40
87 lea rcx,fmt_afisare
88 mov rdx,quot
89 mov r8,rem
90 call printf
91 add rsp,40
92
93 final:
94 ret
95 main endp
96 END

```

Exemplul 46. Următorul program citește un număr n și calculează $n!$, afișând un mesaj de eroare în cazul n negativ, precum și în caz de overflow.

```

1 ;citim n si calculam n!
2 ;daca n este negativ afisam mesaj de eroare
3 ;daca n! genereaza overflow afisam mesaj de eroare
4 includelib libcmt.lib
5 includelib legacy_stdio_definitions.lib
6 printf proto
7 scanf proto
8 .DATA
9 n          SQWORD      ?
10 read_prompt BYTE "Dati -numarul -n:" ,0
11 read_fmt    BYTE "%lld" ,0
12 error_neg   BYTE "Numarul -este -negativ!" ,0ah,0
13 error_overflow BYTE "Overflow!" ,0ah,0
14 print_result BYTE "%lld!=%lld" ,0ah,0
15 .CODE
16 main PROC
17 ;citim cu prompt n
18 sub rsp,40
19 lea rcx,read_prompt
20 call printf
21
22 lea rcx,read_fmt
23 lea rdx,n
24 call scanf
25 ;daca n<0 afisam mesaj de eroare
26 cmp n,0
27 jge efect_calc
28 ;daca nu se efectueaza saltul , afisam mesaj de eroare
29     lea rcx,error_neg
30     call printf
31     jmp final
32 efect_calc:
33 ;calculam n! in rax
34 mov rax,1
35 mov rcx,1
36 ;do { rax = rax * rcx; rcx++; } while(rcx<=n)
37 beg_while:
38     imul rcx
39     ;testam daca inmultirea a generat overflow
40     jno cont_calcul
41     ;daca s-a generat overflow , afisam mesaj si iesim
42     lea rcx,error_overflow
43     call printf

```

```

44         jmp final
45 cont_calcul:
46     inc rcx
47     cmp rcx,n
48     jle beg_while
49 ;n! se va afla in rax
50 ;printf("%lld!=%lld",n,rax)
51 lea rcx,print_result
52 mov rdx,n
53 mov r8,rax
54 call printf
55
56 final:
57 add rsp,40
58 ret
59 main ENDP
60 END

```

Exemplul 47. Următorul program implementează o buclă pentru calculul sumei $1^2 + 2^2 + \dots + n^2$, cu n o variabilă globală inițializată în segmentul de date.

```

1 ;calculam suma 1^2 + ... + n^2
2 ;n este initializat ca variabila globala
3 ;daca n<=0 afisam mesaj de eroare
4 includelib libcmnt.lib
5 includelib legacy_stdio_definitions.lib
6 printf proto
7 scanf proto
8 .DATA
9 n          SQWORD 5
10 error_msg BYTE "%lld <-0-!" ,0ah,0
11 print_sum  BYTE "1^2+...+%lld^2=%lld" ,0ah,0
12 .CODE
13 main PROC
14 cmp n,0
15 jnle calcul_suma
16 ;daca n<=0 afisam mesaj de eroare
17 sub rsp,40
18 lea rcx,error_msg
19 mov rdx,n
20 call printf
21 add rsp,40
22 jmp final
23
24 calcul_suma:
25 ;instruciunea loop decrementeaza rcx si
26 ;executa saltul daca rcx!=0
27 ;vom calcula suma n^2 + ... + 1^2
28 mov rcx,n
29 mov rbx,0
30 ;suma se afla in rbx
31 ;rax va fi folosit pt a calcula rcx^2
32 bucla:
33     mov rax,rcx
34     imul rcx
35     add rbx,rax
36 loop bucla
37
38 ;afisam rezultatul
39 sub rsp,40
40 lea rcx,print_sum

```

```

41 mov rdx,n
42 mov r8,rbx
43 call printf
44 add rsp,40
45 final:
46 ret
47 main ENDP
48 END

```

Exemplul 48. Următorul program este o implementare simplă a unei structuri **if-then-else**: citim un număr de la tastatură și afișăm un mesaj dacă numărul este sau nu egal cu 0.

```

1 ;citim num de la tastatura
2 ;afisam mesaj corespunzator daca num=0 sau nu
3 includelib libcmt.lib
4 includelib legacy_stdio_definitions.lib
5 printf proto
6 scanf proto
7 .data
8 read_prompt byte "Dati-numarul:",0
9 read_fmt byte "%lld",0
10 num sqword ?
11 mesaj byte "Numarul-este-0",0
12 mesajno byte "Numarul-nu-este-0"
13 .code
14 main proc
15
16 ;printf("Dati numarul:")
17 ;arg. printf este transmis prin rcx
18 sub rsp,40
19 lea rcx,read_prompt
20 call printf
21
22 ;scanf("%lld",&num)
23 ;formatul este transmis prin rcx
24 ;rdx primeste adresa lui num
25
26 lea rcx,read_fmt
27 lea rdx,num
28 call scanf
29 add rsp,40
30
31 mov rax,num
32 cmp rax,0
33 jne notzero
34 ;daca rax !=0 se executa saltul la eticheta
35 ;daca nu se continua cu instr. urmatoare
36         sub rsp,40
37         lea rcx,mesaj
38         call printf
39         add rsp,40
40         jmp end_if
41 notzero:
42         sub rsp,40
43         lea rcx,mesajno
44         call printf
45         add rsp,40
46 end_if:
47 ret
48 main endp
49 end

```

Exemplul 49. Următorul program descrie cum se poate citi un array de la tastatură.

```
1  ;citim de la tastatura un array
2  includelib libcmtd.lib
3  includelib legacy_stdio_definitions.lib
4  printf proto
5  scanf proto
6  .data
7  read_prompt_dim    byte "Dati dimensiunea:",0
8  read_fmt_dim       byte "%d",0
9  read_fmt           byte "%lld",0
10 read_prompt_arr    byte "array[%d]=-",0
11 dim sdword ?
12 array sqword 100 dup(?)
13 .code
14 main proc
15 ;printf("Dati dimensiunea:")
16 sub rsp,40
17 lea rcx, read_prompt_dim
18 call printf
19 ;scanf("%d",&num)
20 lea rcx, read_fmt_dim
21 lea rdx, dim
22 call scanf
23 add rsp,40
24
25 ;rbx va contine adresa elementului curent al array
26 ;esi va contine indexul curent
27 lea rbx, array
28 mov esi,0
29
30 begin_while:
31 ;if(esi>=dim) salt la final
32 ;else citim element
33     cmp esi, dim
34     jge end_while
35
36 ;printf("array[%d]", esi)
37     sub rsp,40
38     lea rcx, read_prompt_arr
39     movsxd rdx, esi
40     call printf
41 ;scanf("%lld",&array[esi])
42 ;adresa array[esi] se afla in rbx
43     lea rcx, read_fmt
44     mov rdx, rbx
45     call scanf
46     add rsp,40
47
48 ;incrementam rbx cu 8 (adresa urmatorului element din array)
49 ;incrementam esi
50     add rbx,8
51     inc esi
52 jmp begin_while
53 end_while:
54
55 ret
56 main endp
57 end
```

Exemplul 50. Următorul program calculează suma alternantă a termenilor unui array dat.

```

1 ;calculam ar[0]-ar[1]+ar[2]-...
2 ;unde ar este un array dat ca variabila globala
3 includelib libcmtd.lib
4 includelib legacy_stdio_definitions.lib
5 printf proto
6 scanf proto
7 .data
8 ar sqword 1,2,3,4,5,6,7,8,9,10
9 dim sqword 10
10 sum sqword ?
11 print-format byte "sum=%lld",0ah,0
12 .code
13 main proc
14 ;rax va contine suma alternata
15 ;rcx va fi indicele curent
16 ;rdx va contine paritatea indicelui
17 lea rbx,ar
18 mov rax,0
19 mov rcx,0
20 ;while (rcx<dim) { rax = rax +/- ar[rcx], dupa paritatea rcx}
21 begin_loop:
22 cmp rcx,dim
23 je exit_loop; cand rcx=dim iesim din bucla
24 ;testam paritatea lui rcx
25 mov rdx,rcx
26 and rdx,1; rdx:=rcx & 1 =rcx mod 2
27 ;rax = rax+ ar[rcx] daca rdx=0
28 ;rax = rax- ar[rcx] daca rdx=1
29 cmp rdx,0
30 jne branch_else
31 add rax,[rbx+rcx*8]
32 jmp end_branch
33 branch_else:
34 sub rax,[rbx+rcx*8]
35 end_branch:
36 ;incrementam rcx si reluam ciclul
37 inc rcx
38 jmp begin_loop
39 exit_loop:
40 ;afisam suma
41 sub rsp,40
42 lea rcx,print-format
43 mov rdx,rax
44 call printf
45 add rsp,40
46 ret
47 main endp
48 end

```

Exemplul 51. Următorul program testează dacă adunarea a trei numere reale date este asociativă.

```

1 ;testam daca adunarea este asociativa
2 ;pentru trei numere date.
3 includelib libcmtd.lib
4 includelib legacy_stdio_definitions.lib
5 printf proto
6 .data
7 print_fmt byte "rezultat=%lf",0ah,0
8 mesaj byte "sunt -egale", 0ah,0
9 mesajno byte "nu-sunt -egale",0ah,0
10 nol real8 0.1

```

```

11 no2 real8 0.2
12 no3 real8 0.3
13 .code
14 main proc
15 ;incarcarea registrilor xmm se face cu instructiunea movsd
16 ;operatia de adunare se realizeaza cu addsd
17 movsd xmm0,no1
18 addsd xmm0,no2
19 addsd xmm0,no3; xmm0=(no1+no2)+no3
20 movsd xmm1,no2
21 addsd xmm1,no3
22 addsd xmm1,no1; xmm1=no1+(no2+no3)
23 ;comparatia se realizeaza cu ucomisd
24 ucomisd xmm0,xmm1
25 jne noteq
26     sub rsp,40
27     lea rcx,mesaj
28     call printf
29     add rsp,40
30     jmp final
31 noteq:
32     sub rsp,40
33     lea rcx,mesajno
34     call printf
35     add rsp,40
36 final:
37 ret
38 main endp
39 end

```

Exemplul 52. Următorul program ilustrează alocarea unui tablou bidimensional, calculând determinantul unei matrice 2×2 de numere întregi de 32 de biți, precum și determinantul unei matrice 2×2 de numere reale de 64 de biți.

```

1 ;calculam determinantul unei matrice (2,2)
2 ;cu intrari numere intregi/reale
3 ;afisam un mesaj daca det este pozitiv sau nu
4 includelib libcmnt.lib
5 includelib legacy_stdio_definitions.lib
6 printf proto
7 .data
8 printfmt byte "det_int=%d",0ah,"det_real=%.3lf",0ah,0
9 mat_int sdword 1,5
10         sdword 2,11
11 mat_re real8 1.5, -1.0
12         real8 2.0, 4.0
13 .code
14 main proc
15 ;mat_int are coeficienti sdword (32 biti)
16 ;fiecare coef. ocupa 4 adrese
17 ;folosim registrii eax,ebx pentru calcule
18 mov eax,[mat_int]
19 imul [mat_int+12]
20 mov ebx,eax
21 mov eax,[mat_int+4]
22 imul [mat_int+8]
23 sub ebx,eax
24 ; det mat_int se afla in ebx.
25
26 ;mat_re are coef. real8 (64 biti)
27 ;fiecare coef. ocupa 8 adrese

```

```

28 movsd xmm0,[mat_re]
29 mulsd xmm0,[mat_re+24]
30 movsd xmm1,[mat_re+8]
31 mulsd xmm1,[mat_re+16]
32 subsd xmm0,xmm1
33 ;det mat_re se afla in xmm0
34
35 sub rsp,40
36 lea rcx,print_fmt
37 mov rdx,rbx
38 movq r8,xmm0
39 call printf
40 add rsp,40
41 main endp
42 ret
43 end

```

Exemplul 53. Următorul program folosește o funcție cu un argument transmis pe stivă pentru calculul unei sume.

```

1 ;calculam suma 1+2+...+n printr-o procedura
2 ;n este transmis pe stiva
3 ;folosim o variabila locala pentru index
4 ;returnam suma in rax
5 includelib libcmtd.lib
6 includelib legacy_stdio_definitions.lib
7 printf proto
8 scanf proto
9 .DATA
10 n QWORD 10
11 fmt BYTE "%lld",0
12
13 .CODE
14 main PROC
15 ;sum are un singur argument transmis pe stiva
16 push n ;plasam n pe stiva
17 call sum ;apelam functia
18 add rsp,8 ;curatam stiva
19 ;afisam suma
20 sub rsp,40
21 lea rcx,fmt
22 mov rdx,rax
23 call printf
24 add rsp,40
25 ret
26 main ENDP
27 sum PROC
28 sub rsp,8;alocam pe stiva spatiu pentru contorul sumei
29 mov rcx,[rsp+16]; rcx = [rsp+16] = n
30 mov rax,0;rax contine valoarea curenta a sumei
31 mov QWORD PTR [rsp],0; contor=0
32 ;do { rax+=[rsp]; [rsp]++;} while(rcx<=[rsp+16])
33 tag:
34 add rax,[rsp]
35 inc qword ptr [rsp]
36 cmp [rsp],rcx
37 jle tag
38 add rsp,8
39 ret
40 sum ENDP
41 END

```

Exemplul 54. Următorul program calculează minimul a trei numere întregi, folosind instrucțiuni de salt.

```
1 ;citim 3 nr intregi
2 ;det minimul dintre ele, folosind o functie
3 includelib libcmt.lib
4 includelib legacy_stdio_definitions.lib
5 printf proto
6 scanf proto
7 .data
8 read_prompt1 byte "Dati-num1: ",0
9 read_prompt2 byte "Dati-num2: ",0
10 read_prompt3 byte "Dati-num3: ",0
11 read_fmt byte "%lld",0
12 num1 sqword ?
13 num2 sqword ?
14 num3 sqword ?
15 mesaj byte "Minimul este -%lld",0ah,0
16 .code
17 main proc
18 ;printf("Dati num1:");
19 ;scanf("%lld",num1);
20 sub rsp,40
21 lea rcx,read_prompt1
22 call printf
23 lea rcx,read_fmt
24 lea rdx,num1
25 call scanf
26 add rsp,40
27
28 ;printf("Dati num2:");
29 ;scanf("%lld",num2);
30 sub rsp,40
31 lea rcx,read_prompt2
32 call printf
33 lea rcx,read_fmt
34 lea rdx,num2
35 call scanf
36 add rsp,40
37
38 ;printf("Dati num3:");
39 ;scanf("%lld",num3);
40 sub rsp,40
41 lea rcx,read_prompt3
42 call printf
43 lea rcx,read_fmt
44 lea rdx,num3
45 call scanf
46 add rsp,40
47
48 ;apelam minim3(num1,num2,num3)
49 ;arg. sunt plasate pe stiva
50 ;valoarea returnata se afla in rax
51 push num3
52 push num2
53 push num1
54 call minim3
55 add rsp,24
56
57 ;afisam minimul aflat in rax
58 sub rsp,40
59 lea rcx,mesaj
```

```

60 mov rdx, rax
61 call printf
62 add rsp, 40
63 ret
64 main endp
65
66 ; minim3(long long arg1, long long arg2, long long arg3)
67 ; [rsp+8]= arg1, [rsp+16]=arg2, [rsp+24]=arg3
68 minim3 proc
69 mov r8, [rsp+8]
70 mov r9, [rsp+16]
71 mov r10, [rsp+24]
72 ; cele trei arg. se afla in r8, r9, r10
73 ; daca r8 <= r9 comparam r8 cu r10 pentru minim
74 ; altfel comparam r9 cu r10 pentru minim
75 ; minimul se va afla in rax
76 cmp r8, r9
77 jge comp2cu3
78     cmp r8, r10
79     jge min_arg3
80     mov rax, r8
81     jmp final
82 min_arg3:
83     mov rax, r10
84     jmp final
85 comp2cu3:
86     cmp r9, r10
87     jge min_arg3
88     mov rax, r9
89 final:
90 ret
91 minim3 endp
92 end

```

Exemplul 55. Următorul program calculează maximul a trei numere reale, folosind instrucțiuni de salt.

```

1 ; citim 3 nr reale
2 ; determinam maximul folosind o procedura
3 includelib libcmt.lib
4 includelib legacy_stdio_definitions.lib
5 printf proto
6 scanf proto
7 .data
8 fmt_write byte "Dati numerele:" , 0
9 fmt_read byte "%lf%lf%lf" , 0
10 num1 real8 ?
11 num2 real8 ?
12 num3 real8 ?
13 print_mes byte "Numerele sunt -%lf , -%lf , -%lf." , 0ah, "Maximul este : -%lf" , 0ah, 0
14 .code
15 main proc
16 sub rsp, 40
17 lea rcx, fmt_write
18 call printf
19 add rsp, 40
20
21 sub rsp, 40
22 lea rcx, fmt_read
23 lea rdx, num1
24 lea r8, num2
25 lea r9, num3

```

```

26  call  scanf
27  add  rsp,40
28
29  ;apelam maxim3(num1,num2,num3)
30  push num3
31  push num2
32  push num1
33  call maxim3
34  add  rsp,24
35
36  push rax
37  sub  rsp,32
38  lea  rcx, print_mes
39  mov  rdx,num1
40  mov  r8,num2
41  mov  r9,num3
42
43  call printf
44  add  rsp,40
45  ret
46  main  endp
47
48  maxim3  proc
49  push  rbp
50  mov  rbp, rsp
51  movsd xmm0, real8 ptr [rbp+16]
52  movsd xmm1, real8 ptr [rbp+24]
53  movsd xmm2, real8 ptr [rbp+32]
54  ;arg. sunt in xmm0,xmm1,xmm2
55  ;daca xmm0<xmm1 comparam xmm1 cu xmm2 pentru maxim
56  ;daca xmm0>=xmm1 comparam xmm0 cu xmm2 pentru maxim
57
58  ucomisd xmm0,xmm1
59  jb  comp2cu3
60      ucomisd xmm0,xmm2
61      jb  max_arg3
62      movq rax,xmm0
63      jmp final
64  max_arg3:  movq rax,xmm2
65              jmp final
66  comp2cu3:
67      ucomiss xmm1,xmm2
68      jb  max_arg3
69      movq rax,xmm1
70  final:
71  mov  rsp,rbp
72  pop  rbp
73  ret
74  maxim3  endp
75  end

```

Exemplul 56. ;citim un numar natural

```

2  ;calculam suma cifrelor
3  ;calculam rasturnatul
4  includelib libcmtd.lib
5  includelib legacy_stdio_definitions.lib
6  printf proto
7  scanf proto
8  .data
9  fmt_write byte "Dati-numarul-natural:-",0
10  fmt_write_err byte "Numarul-nu-e-pozitiv!-",0

```

```

11  fmt_read_dim    byte "%lld",0
12  message byte "suma cifrelor este: %lld",0ah,0
13  message2 byte "rasturnatul este: %lld",0ah,0
14  num sqword ?
15  .code
16  main proc
17  sub rsp,40
18  lea rcx,fmt_write
19  call printf
20  add rsp,40
21
22  sub rsp,40
23  lea rcx,fmt_read_dim
24  lea rdx,num
25  call scanf
26  add rsp,40
27
28  mov rax,num
29  cmp rax,0
30  jl @F
31  push num
32  call sum_digit
33  add rsp,8
34
35  sub rsp,40
36  lea rcx,message
37  mov rdx,rax
38  call printf
39  add rsp,40
40
41
42  push num
43  call rasturnat
44  add rsp,8
45
46  sub rsp,40
47  lea rcx,message2
48  mov rdx,rax
49  call printf
50  add rsp,40
51
52  jmp end_prog
53 @@: sub rsp,40
54     lea rcx,fmt_write_err
55     call printf
56     add rsp,40
57
58  end_prog:
59  ret
60  main endp
61
62  sum_digit proc
63  push rbp
64  mov rbp,rsp
65  mov rcx,10
66  mov rbx,0
67  mov rax,[rbp+16]
68  begin_while: cmp rax,0
69               je  end_while
70               cqo
71               idiv rcx

```

```

72         add rbx,rdx
73     jmp begin_while
74 end_while:
75 mov rax,rbx
76 mov rsp,rbp
77 pop rbp
78 ret
79 sum_digit endp
80
81 rasturnat proc
82 push rbp
83 mov rbp,rsp
84 mov rcx,10
85 mov rax,0
86 mov rbx,[rbp+16]
87 begin_while: cmp rbx,0
88             je end_while
89
90             push rax
91             mov rax,rbx
92             cqo
93             idiv rcx
94             mov rbx,rax
95             pop rax
96
97             push rdx
98             imul rcx
99             pop rdx
100            add rax,rdx
101            jmp begin_while
102 end_while:
103 mov rsp,rbp
104 pop rbp
105 ret
106 rasturnat endp
107
108 end

```

Exemplul 57. ;intr-o matrice, calculam suma elementelor

```

2 ;de pe o coloana data
3 includelib libcmt.lib
4 includelib legacy_stdio_definitions.lib
5 printf proto
6 scanf proto
7 .data
8
9 nr_lin sqword 3
10 nr_col sqword 4
11 array sqword 7, 6, -1, 4
12         sqword 1, 4, 4, 2
13         sqword 3, 2, 4, 3
14 col sqword 2
15 fmt_read byte "%lld",0
16 mesaj byte "Suma este %lld",0ah,0
17 .code
18 main proc
19 ;apelam sum(array, nr_lin, nr_col, col)
20 ;suma se afla in rax
21 push col
22 push nr_col
23 push nr_lin

```

```

24 lea rcx, array
25 push rcx; adresa array este transmis ca arg
26 call sum
27 add rsp, 32
28
29 sub rsp, 40
30 lea rcx, mesaj
31 mov rdx, rax
32 call printf
33 add rsp, 40
34 ret
35 main endp
36
37 ;sum(array, nr_lin, nr_col, col)
38 sum proc
39 push rbp
40 mov rbp, rsp
41 ;[rbp+16]=adresa array
42 ;[rbp+24,32,40] = nr_lin, nr_col, col
43 mov rbx, [rbp+16]; rbx=adresa array
44
45 mov rcx, [rbp+40]; rcx=col
46 sal rcx, 3 ; rcx= col>>3 = col * 8
47 add rbx, rcx; rbx = adr. array[0][col]
48
49 ;calc. lung unei linii
50 mov rcx, [rbp+32]; rcx=nr_col
51 sal rcx, 3 ; rcx = rcx<<3 = rcx *8
52
53 mov rsi, 0; rsi = contor pt coloana
54 mov rax, 0; rax = suma curenta
55 bucla:
56     cmp rsi, [rbp+24]
57     je final
58     add rax, [rbx]
59     add rbx, rcx
60     inc rsi
61     jmp bucla
62 final:
63 mov rsp, rbp
64 pop rbp
65 ret
66 sum endp
67
68 end

```

18 Exemple 32 bit

Exemplul 58. Următorul program calculează expresia $(v_1 - v_2)/(v_3 \cdot 2 + 4)$ pentru niște valori date v_1, v_2, v_3 date

```

1 ;calculeaza expresii aritmetice
2 ; (v1-v2)/(v3*2+4)
3 includelib msvcrt.lib
4 includelib legacy_stdio_definitions.lib
5 .386
6 .MODEL FLAT,C
7 .STACK 100h
8 printf PROTO arg1:ptr BYTE, printlist:VARARG
9 scanf PROTO arg2:ptr BYTE, inputlist:VARARG

```

```

10 .DATA
11 v1      SDWORD  -50
12 v2      SDWORD  -10
13 v3      SDWORD  2
14 rez     SDWORD  ?
15 fmt     BYTE    0Ah," Rezultatul este:%d" ,0Ah,0
16 .CODE
17 main PROC
18 ;add dest, sursa —> dest:=dest+sursa
19 ;sub dest, sursa —> dest:=dest-sursa
20 ;imul oper —> (edx:eax):=eax*oper; oper nu poate fi val imediat
21 ;idiv oper —> eax,edx primesc catul si restul impartirii lui (edx:eax) la oper.
22 ; daca vrem sa imp eax la operand, edx ar trebui sa preia semnul lui eax.
23 mov ebx,v1
24 sub ebx,v2      ;ebx=v1-v2
25 mov eax,2       ;eax=2
26 imul v3         ;eax=2*v3; edx=0
27 add eax,4       ;eax=2*v3+4
28 mov ecx,eax     ;eax preia ebx si val veche eax e pastrata in ecx
29 mov eax,ebx
30 cdq             ;extinde bitul de semn de la eax la (edx:eax)
31 idiv ecx
32 mov rez,eax
33 INVOKE printf,ADDR fmt,rez
34 ret
35 main ENDP
36 END

```

Exemplul 59. Următorul program calculează suma $1 + 2 + \dots + n$ pentru un n citit de la tastatură.

```

1 ;calculam suma 1+2+...+n, folosind o structura while
2 includelib msvcrt.lib
3 includelib legacy_stdio_definitions.lib
4 .386
5 .MODEL FLAT,C
6 .STACK 100h
7 printf PROTO arg1:ptr BYTE, printlist:VARARG
8 scanf PROTO arg2:ptr BYTE, inputlist:VARARG
9 .DATA
10 n      SDWORD ?
11 mesaj1 BYTE  "Dati-n:" ,0
12 mesaj2 BYTE  "Eroare: -n<=0" ,0Ah,0
13 mesaj3 BYTE  "Suma este:%3d" ,0Ah,0
14 fmt     BYTE  "%d"
15 sum     SDWORD ?
16 .CODE
17 main PROC
18     INVOKE printf,ADDR mesaj1
19     INVOKE scanf,ADDR fmt,ADDR n
20     cmp n,0                                ;daca n<0 afisam mesaj de eroare
21     jge bucla
22     INVOKE printf,ADDR mesaj2
23     jmp final
24 bucla:
25     ;sum=0
26     ; int i=1; while(i<=n){sum+=i; i++;}
27     ; i->eax, sum ->ebx
28     mov eax,1                                ;eax =1
29     mov ebx,0                                ;ebx=0
30 while1: cmp eax,n
31     jg endwhile1                            ;daca eax>n, iesim din bucla

```

```

32 dol:    add ebx,eax          ;ebx = ebx + eax
33        inc eax              ;eax=eax + 1
34        jmp while1
35 endwhile1:
36        mov sum,ebx
37        INVOKE printf,ADDR mesaj3,sum
38 final:  ret
39 main    ENDP
40        END

```

Exemplul 60. Următorul **program** citește de la tastatură un vector și determină maximul său.

- pointerul la primul element al vectorului **sir** este încărcat în **ebx**.
- Citirea este făcută folosind adresare indexat-scalată. Elementul **sir[i]** (notație C) este accesat prin **adresa** sa **ebx+4*esi** (adunăm la adresa primului element din șir, stocat în **ebx**, **4*esi** octeți).
- registrul **esi** este folosit ca index de parcurgere a vectorului, luând valori de la 0 la **dim-1**, dimensiunea vectorului, cu pas de incrementare 1.
- Determinarea maximului, pe de altă parte, se face folosind adresare bazat-indexat. Elementul **sir[i]** este accesat prin **adresa** sa **ebx+esi**.
- registrul **esi** este folosit ca index de parcurgere a vectorului, luând valori de la 0 la **4*(dim -1)**, din 4 în 4.

```

1  ;citim un vector de la tastatura si determinam cel mai mare element
2  includelib msvcrt.lib
3  includelib legacy_stdio_definitions.lib
4  .386
5  .MODEL FLAT,C
6  .STACK 100h
7  printf PROTO arg1:ptr BYTE, printlist:VARARG
8  scanf  PROTO arg2:ptr BYTE, inputlist:VARARG
9  .DATA
10 sir          SDWORD 100 dup(?)
11 format  byte "Dati-vectorul:",0ah,0
12 format2 byte "Dati-dimensiunea:",0
13 format3 byte "sir[%d]=" ,0
14 format4 byte "%d" ,0
15 dim      dword ?
16 format5 byte "Maximul-este:%d" ,0ah,0
17 .CODE
18 main PROC
19 invoke printf,addr format2
20 invoke scanf, addr format4,addr dim
21 invoke printf,addr format
22 ;citim vectorul
23 mov ebx, offset sir      ; in ebx punem pointerul la primul element al sir.
24 mov esi,0                ; esi va fi indexul de parcurgere al vectorului
25 beg_while: cmp esi,dim ; citim valori pana cand esi=dim
26 je end_while
27 invoke printf, addr format3,esi
28 invoke scanf, addr format4,addr [ebx+esi*4]
29 ;ebx+esi*4 este adresa situata la distanta 4*esi de ebx
30 ;ebx+esi*4 este deci adresa elementului de pe pozitia esi in vectorul (de numere intregi)
31 inc esi;incrementam esi, pentru a trece la elementul urmator
32 jmp beg_while
33 end_while:
34 ;determinam maximul
35 mov ebx, offset sir      ; in ebx punem pointerul la primul element al sir.
36 mov eax,[ebx]            ; eax va fi maximul curent

```

```

37 ; eax = sir[0] (primul element al vectorului indicat de ebx)
38 ;vom parcurge sir (din 4 in 4 adrese) comparand fiecare element cu eax, modificand maximul
39 ;elementul din sir va fi identificat prin [ebx+esi] = continutul adresei ebx, deplasat cu
40 ;esi va lua valori de la 0 la 4*dim-4, din 4 in 4
41 mov ecx,dim
42 sal ecx,2 ;ecx=4*dim
43 mov esi,0
44 beg_while1: cmp esi,ecx ;parcurgem vectorul pana cand esi = ecx =4*dim
45 je end_while1
46 cmp [ebx+esi],eax ;daca [ebx+esi] (termenul curent al sirului) este <= eax, nu facem nimic
47 jle no_change ;altfel, eax devine [ebx+esi]
48 mov eax,[ebx+esi]
49 no_change: add esi,4 ;trecem la elementul urmator
50 jmp beg_while1
51 end_while1:
52 invoke printf, addr format5,eax
53 ret
54 main ENDP
55 END

```

Exemplul 61. Următorul program verifică dacă un număr citit este palindrom.

```

1 ;verificam daca un numar dat este palindrom
2 includelib msvcrt.lib
3 includelib legacy_stdio_definitions.lib
4 .386
5 .MODEL FLAT,C
6 .STACK 100h
7 printf PROTO arg1:ptr BYTE, printlist:VARARG
8 scanf PROTO arg2:ptr BYTE, inputlist:VARARG
9 .DATA
10 numar SDWORD ?
11 mesaj1 BYTE "Dati -numar:" ,0
12 mesaj3 BYTE "Numarul -este -palindrom" ,0ah,0
13 mesaj4 BYTE "Numarul -nu -este -palindrom" ,0ah,0
14 fmt BYTE "%d"
15 .CODE
16 main PROC
17 INVOKE printf,ADDR mesaj1 ;citim numar
18 INVOKE scanf,ADDR fmt,ADDR numar
19 bucla:
20 ;sum=0
21 ; while(numar) {sum+=numar%10; numar=numar/10;}
22 ; numar->eax, sum ->ebx
23 mov eax,numar
24 mov ebx,0
25 mov ecx,10
26 while1: cmp eax,0
27 je endwhile1 ;daca eax=0, iesim din bucla
28 dol: cdq
29 idiv ecx ;eax=eax/10, edx=eax%10
30 push eax
31 push edx
32 mov eax,ebx ;eax=ebx
33 imul ecx ;eax =eax*10, edx=0
34 mov ebx,eax
35 pop edx
36 pop eax
37 add ebx,edx
38 jmp while1
39 endwhile1:

```

```

40 cmp numar,ebx
41 je pal
42 invoke printf, addr mesaj4
43 jmp final
44 pal: invoke printf, addr mesaj3
45 final: ret
46 main ENDP
47 END

```

Exemplul 62. Următorul program citește de la tastatură trei numere naturale `baza`, `exp` și `num` și calculează expresia $baza^{exp} \bmod num$, folosind ridicări succesive la pătrat.

```

1 ;calculam baza^exp mod num
2 includelib msvcrt.lib
3 includelib legacy_stdio_definitions.lib
4 .386
5 .MODEL FLAT,C
6 .STACK 100h
7 printf PROTO arg1:ptr BYTE, printlist:VARARG
8 scanf PROTO arg2:ptr BYTE, inputlist:VARARG
9 .DATA
10 baza SDWORD ?
11 exp SDWORD ?
12 num SDWORD ?
13 mesaj1 BYTE "Dati-baza:",0
14 mesaj2 BYTE "Dati-exp:",0
15 mesaj3 BYTE "Dati-num:", 0
16 fmt BYTE "%d"
17 mesaj4 BYTE "baza^exp mod num=%d",0ah,0
18 putere SDWORD ?
19 .CODE
20 main PROC
21 INVOKE printf,ADDR mesaj1
22 INVOKE scanf,ADDR fmt,ADDR baza
23
24 INVOKE printf,ADDR mesaj2
25 INVOKE scanf,ADDR fmt,ADDR exp
26
27 INVOKE printf,ADDR mesaj3
28 INVOKE scanf,ADDR fmt,ADDR num
29
30
31 bucla:
32 mov eax,1
33 mov putere,eax ;putere=1
34 mov ecx,exp ;ecx=exp
35 mov ebx,baza ;ebx=baza
36 beg_while:
37 cmp ecx,0
38 je final
39 push ecx
40 and ecx,1
41 cmp ecx,0
42 je end_if1
43
44 mov eax,putere ;eax=putere
45 imul ebx ;eax=putere*ebx
46 cdq
47 idiv num ;edx=eax/num
48 mov putere,edx ;putere=edx
49

```

```

50 end_if1: pop ecx
51
52 mov eax, ebx      ;ebx=eax
53 imul ebx          ;eax=ebx^2
54 cdq
55 idiv num          ;ebx= eax%edx
56 mov ebx,edx
57
58 sar ecx,1
59 jmp beg_while
60 final:
61 invoke printf,addr mesaj4,putere
62 ret
63 main      ENDP
64 END

```

Exemplul 63. Următorul program citește de la tastatură un vector și îi determină minimul, folosind două proceduri (pentru citire, respectiv determinarea minimului).

```

1 ;citim un vector de la tastatura si determinam cel mai mic element folosind proceduri
2 includelib msvcrt.lib
3 includelib legacy_stdio_definitions.lib
4 .386
5 .MODEL FLAT,C
6 .STACK 100h
7 printf PROTO arg1:ptr BYTE, printlist:VARARG
8 scanf  PROTO arg2:ptr BYTE, inputlist:VARARG
9 .DATA
10 sir          SDWORD 100 dup(?)
11 format byte "Dati-vectorul:",0ah,0
12 format2 byte "Dati-dimensiunea:",0
13 format3 byte "sir[%d]=" ,0
14 format4 byte "%d" ,0
15 dim        dword ?
16 format5 byte "minimul-este-%d" ,0ah,0
17 .CODE
18 main PROC
19 invoke printf,addr format2
20 invoke scanf, addr format4,addr dim
21 invoke printf,addr format
22 ;citim vectorul
23 push dim
24 push offset sir
25 call citire
26 add esp,8
27 push dim
28 push offset sir
29 call minim
30 add esp,8
31 invoke printf, addr format5,eax
32 ret
33 main ENDP
34 ;citire (int *vector, int dim)
35 citire proc
36 push ebp
37 mov ebp,esp
38 mov ebx, [ebp+8] ; in ebx punem pointerul la primul element al sir.
39 mov esi,0 ; esi va fi indexul de parcurgere al vectorului
40 beg_while: cmp esi,[ebp+12] ; citim valori pana cand esi=dim
41 je end_while
42 invoke printf, addr format3,esi

```

```

43 lea eax,[ebx+esi*4]
44 push eax
45 push offset format4
46 call scanf
47 add esp,8
48 ;ebx+esi*4 este adresa situata la distanta 4*esi de ebx
49 ;ebx+esi*4 este deci adresa elementului de pe pozitia esi in vectorul (de numere intregi)
50 inc esi;incrementam esi, pentru a trece la elementul urmator
51 jmp beg_while
52 end_while:
53 mov esp,ebp
54 pop ebp
55 ret
56 citire endp
57 minim proc
58 push ebp
59 mov ebp,esp
60 mov ebx,[ebp+8]
61 mov eax,[ebx]
62 mov esi,1
63 beg_while2: cmp esi,[ebp+12]
64 je end_while2
65 cmp eax,[ebx+4*esi]
66 jle end_if
67 mov eax,[ebx+4*esi]
68 end_if: inc esi
69 jmp beg_while2
70 end_while2:
71 mov esp,ebp
72 pop ebp
73 ret
74 minim endp
75 END

```

Exemplul 64. Următorul program calculează suma de la 1 la n prin intermediul unei proceduri.

```

1 ;suma de la 1 la n intr-o procedura
2 includelib msvcrt.lib
3 includelib legacy_stdio_definitions.lib
4 .386
5 .MODEL FLAT,C
6 .STACK 100h
7 extern printf:proc
8 extern scanf:proc
9 .DATA
10 n sdword ?
11 fmt byte "%d",0
12 msj byte "Dati-n:",0
13 msj2 byte "Suma este:%d",0ah,0
14 .code
15 main PROC
16 push offset msj
17 call printf
18 add esp,4
19 push offset n
20 push offset fmt
21 call scanf
22 add esp,8
23 push n
24 call sum
25 add esp,4

```

```

26 push eax
27 push offset msj2
28 call printf
29 add esp,8
30 ret
31 main      ENDP
32 sum      PROC
33 push ebp
34 mov ebp,esp
35 sub esp,4
36 mov eax,0
37 mov ecx,[ebp+8]
38 mov DWORD PTR [ebp-4],1
39 loop01:
40 add eax,[ebp-4]
41 inc DWORD PTR [ebp-4]
42 loop loop01
43 mov esp,ebp
44 pop ebp
45 ret
46 sum      ENDP
47 END

```