SUPORT SEMINAR 1 - ASSEMBLER

Arhitectura memoriei [LUNG00]
Una dintre cele mai importante caracteristici ale arhitecturii unui calculator este modul de organizare a memoriei şi modul în care informaţia din memorie este accesată.
Memoria principală este organizată ca un set de locaţii de memorare, numerotate consecutiv, începând de la 0. Numerele asociate locaţiilor fizice reprezintă adresa fizică, iar mulţimea totală a adreselor fizice constituie spaţiul de adrese fizice.
O adresă logică este o adresă utilizată într-o instrucţiune de către programator. Adresele care pot fi utilizate de un program constituie spaţiul de adrese logice. Organizarea acestui spaţiu defineşte arhitectura memoriei.
Organizarea spaţiului de adrese fizice este determinată de tehnologia utilizată pentru memorie şi de costul ei, dar spaţiul de adrese logice nu ar trebui să fie condiţiona de oricare dintre aceste consideraţii. Organizarea memoriei logice este determinată de structura programelor ce vor rula în memorie. Iniţial, spaţiul de adrese logice era identic cu cel de adrese fizice.

Memoria liniară
Este cea mai obişnuită organizare a spaţiului de adrese logice şi cea mai obişnuită arhitectură pentru memorie: un spaţiu liniar, continuu de adrese. Adresele pornesc de la zero şi continuă în mod liniar, fără goluri sau întreruperi, până la limita superioară, impusă de numărul toţi biţi dintr-o adresă logică.
Spaţiul rezervat de un program pentru procedurile şi datele sale reprezintă o zonă continuă ce este accesată în funcţie de adresa de început a acesteia prin intermediul deplasării (offset-ului).
Nemaparea corectă a zonelor de adrese logice cu cele fizice conduce în aceste situaţii la utilizarea incorectă a aceluiaşi spaţiu de către mai multe programe.
Maparea este, în esenţă, procesul de translatare a adreselor logice în adrese fizice. In cazul memoriei liniare adresele logice sunt puse în corespondenţă cu adresele fizice. O adresă logică este asignată la o adresă fizică aleasă în funcţie de capacitatea memoriei şi de spaţiul rămas liber. Maparea reprezintă un mecanism pentru realocarea spaţiului de adrese logice peste spaţiul de adrese fizice.

Memoria paginată
În locul mapării întregului spaţiu logic de adrese ca o zonă continuă de adrese fizice, se mapează pagini de dimensiune fixă, mai mici, ale spaţiului de adrese logice, în pagini de memorie fizică. Astfel, un program mare nu trebuie să fie realocat într-o zonă (porţiune) continuă de memorie, care poate fi greu de găsit într-un cadru cu multiprograme, ci mai degrabă în mai multe secţiuni de memorie, mai mici, care sunt mult mai uşor de găsit şi sunt disponibile. Sunt mai uşor de găsit 35 de pagini de 1 Kb, decât un bloc de 35 Kb.
Mecanismul de paginare reprezintă baza pentru protecţia memoriei într-un spaţiu logic de adrese. Fiecare pagină poată avea asociate atribute (drepturi de acces) care indică modul de accesare a paginii.

Memoria segmentată

Altă formă de organizare a memoriei logice este memoria segmentată. Motivația o reprezintă faptul că programele nu sunt scrise ca o secvență liniară de instrucțiuni şi date, ci mai degrabă ca bucăți (secvențe) de cod şi bucăți de date. De exemplu un program are o secvență principală de cod şi mai multe proceduri separate. Aceste module de cod şi date sunt de diferite dimensiuni Spațiul logic de adrese este despărțit în spații liniare de adrese, fiecare cu o anumită dimensiune. Fiecare dintre aceste spații de adrese liniare este denumit segment. Fiecare element dintr-un segment este accesat prin intermediul:

- adresei de început a segmentului;
- deplasamentului, adresa relativă față de începutul segmentului a elementului căutat.

Fiecare segment poate fi asociat unui modul de date sau de program. Programul poate avea procedura principală într-un segment, fiecare altă procedură în propriul ei segment şi fiecare structură importantă de date în segmentul propriu. Astfel, structura adreselor logice reflectă organizarea logică a programului.
Maparea, în acest caz, este implementată printr-o tabelă de segment, care păstrează i descriptor de segment pentru fiecare segment. Descriptorul de segment conține adrese de început a segmentului şi lungimea acestuia. Componenta selectorului de segment a unei adrese logice este utilizată ca un index pentru a selecta descriptorul de segment în tabela (descriptorilor) de segment. Apoi deplasamentul este adunat la adresa de segmentului, furnizată de descriptor, pentru a calcula adresa fizică a operandului referit. Deplasamentul este verificat de hardware pentru a se asigura că referința nu depăşeşte lungimea segmentului.
Descriptorul de segment mai conține, pe lângă adresa de bază a segmentului şi limita sa, şi atribute referitoare la tipul segmentului. Drepturile de acces sunt deci asociate, în particular, fiecărui segment, în ciuda faptului că modulele programului referă acele segmente.

Memoria virtuală
Spațiul de adrese logice este mult mai mare decât memoria fizică. Memoria virtuală este un mecanism utilizat pentru a extinde limitele memoriei fizice. Într-un sistem cu memorie virtuală, aceasta apare utilizatorului ca şi cum întregul spațiu logic de adrese este disponibil pentru memorare. Dar, de fapt, în orice moment doar câteva pagini din spațiul logic de adrese sunt mapate peste spațiul fizic. Alte pagini nu sunt prezente în memoria principală; în schimb, informația din aceste pagini este memorată într-o memorie secundară, cum ar fi discul al cărui cost/bit este mult mai scăzut.
Orii câte ori este accesată o pagină care lipseşte, sistemul de operare încarcă pagina respectivă de pe disc şi memorează pe disc o altă pagină, care nu a fost recent referită.

Procesorul **8086** are 14 registre pe 16-biți, utilizați în diferite scopuri.

Registre de Segment		
CS	Code	Număr pe 16-biți ce reprezintă adresa segmentului de cod

	Segment	activ.
DS	Data Segment	Număr pe 16-biți ce reprezintă adresa segmentului de date activ.
SS	Stack Segment	Număr pe 16-biți ce reprezintă adresa segmentului de stivă activ.
ES	Extra Segment	Număr pe 16-biți ce reprezintă adresa segmentului suplimentar activ.
Registre index		
IP	Instruction Pointer	Număr pe 16-biți ce reprezintă offeset-ul instrucțiunii următoare de executat în cadrul segmentului de cod.
SP	Stack Pointer	Număr pe 16-biți ce reprezintă offeset-ul din cadrul stivei.
BP	Base Pointer	Utilizat pentru a transfera în/din stivă date.
Registre de bază		
AX	Accumulator Register	Calcule şi operații de intrare/ieşire
BX	Base Register	Utilizat ca index
CX	Count Register	Utilizat in instrucțiunile de ciclare
DX	Data Register	Operații de intrare/ieşire, înmulțire/împărțire.
Registre index		
SI	Source Index	Utilizat in operațiile cu şiruri pentru a parcurge sursa.
DI	Destination Index	Utilizat in operațiile cu şiruri pentru a parcurge sursa.

Registrele de bază pot fi accesate si din punctul de vedere a primilor 8 biți, respectiv a ultimilor 8 biți.

AX	
AH	AL
BX	
BH	BL
CX	
CH	CL
DX	
DH	DL

De exemplu, dacă registrul AX conține valoarea 1234h atunci AH are valoarea 12h şi AL are 34h.

Un registru pe 16 biţi numit registru FLAG este utilizat pentru a controla execuţia instrucţiunilor. Flag-urile reprezintă biţi ce pot avea valoarea 1 (**SET**) sau 0 (**NOT SET**).

Registru de FLAG-uri		
Abr.	**Nume**	**Nr. bit**
OF	Overflow Flag	11
DF	Direction Flag	10
IF	Interrupt Flag	9
TF	Trap Flag	8
SF	Sign Flag	7
ZF	Zero Flag	6
AF	Auxiliary Carry	4
PF	Parity Flag	2
CF	Carry Flag	0

Semnificaţia indicatorilor de condiţie este următoarea [IVAN02]:

- CF (Carry Flag, indicator de transport) semnifică un transport sau un împrumut în, respectiv din bitul cel mai semnificativ al rezultatului.
- PF (Parity Flag, indicator de paritate) este poziţionat după cum biţii din cel mai puţin semnificativ octet al rezultatului sunt în număr par (poziţionare pe 1) sau impar (poziţionare pe 0).
- AF (Adjust Flag, indicator de ajustare) este folosit în aritmetica zecimală şi semnifică un transport sau un împrumut în, respectiv din bitul 4 (cel mai semnificativ bit al jumătăţii celei mai puţin semnificative) al rezultatului.
- ZF (Zero Flag, indicator de zero) indică dacă rezultatul unei operaţii a fost sau nu zero.
- SF (Sign Flag, indicator de semn) are aceeaşi valoare ca bitul cel mai semnificativ al rezultatului (bitul de semn): 0 - pozitiv, 1 - negativ.
- TF (Trap Flag, indicator de urmărire a execuţiei) este folosit la depanarea programelor prin execuţia lor pas cu pas - dacă este setat, procesorul forţează automat o excepţie după execuţia fiecărei instrucţiuni.
- IF (Interrupt Flag, indicator de întreruperi) precizează dacă procesorul ia în considerare sau nu întreruperile externe.
- DF (Direction Flag, indicator de direcţie) precizează sensul (0 - crescător sau 1 - descrescător) în care este modificat contorul de adrese la operaţiile cu şiruri.
- OF (Overflow Flag, indicator de depăşire) semnifică depăşirea domeniului admisibil la reprezentarea rezultatului unei operaţii aritmetice cu sau fără semn. Practic, este poziţionat pe 1 dacă apare un transport înspre bitul cel mai semnificativ al rezultatului din bitul vecin, dar nu şi din bitul cel mai semnificativ spre CF sau invers, dinspre bitul cel mai semnificativ spre CF, dar nu şi spre bitul cel mai semnificativ din bitul vecin. Similar, la împrumut, este poziţionat pe 1

dacă apare un transport de la bitul cel mai semnificativ la bitul vecin, dar nu şi înspre bitul cel mai semnificativ dinspre CF sau invers, dinspre CF spre b.c.m.s., dar nu şi dinspre bitul cel mai semnificativ spre bitul vecin.

TIPURI DE DATE

1. Bit
2. Nibble – grup de 4 biți; utilizat în codificarea numerelor în format BCD (Binary Coded Decima)
3. Byte (octet) – grup de 8 biți; definire cu **db**;
4. Word (cuvânt) – grup de 16 biți sau 2 octeți; definire cu **dw**;
5. Double (dublu cuvânt) – grup de 32 biți sau 4 octeți; definire cu **dd**;
6. Quad – grup de 64 biți sau 8 octeți; definire cu **dq**.

ETAPE IN REALIZAREA UNUI PROGRAM ASSEMBLER

1. Editare fişier sursă numit **fişier.ASM** cu un editor de text simplu; NOTEPAD în Windows sau editorul de DOS apelat din linia de comandă: *C:\>edit fisier.asm*
2. construirea fişierului obiect şi verificarea codului din sursă: *C:\> tasm fisier.asm*
3. dacă nu există erori de asamblare se construieşte **fisier.OBJ**
4. obținerea executabilului: *C:\> tlink fisier.OBJ*
5. rularea executabilului în Debugger: *C:\> td fisier*

Exemplu 1:

```
; programul determină expresia e=a+b
;-----------------------------------------------------------------------
.model small
.286
.stack 100h
.data
        a db 10
        b db 35

        e db ?
.code
        mov AX,@data
        mov DS,AX

        mov AL,a
        add AL,b
        mov e,AL

        mov AX,4c00h
        int 21h
end
```

Descriere program:

.model small : Liniile care încep cu „.”reprezintă instrucțiuni speciale care indică programului assembler anumite informații, descrise de cuvintele cheie ce urmează, cu privire la programul de construit. În această situație **model** indică faptul că urmează a se indica modelul de memorie (şi cantitatea de memorie) utilizat de program. Acest program necesită un spațiu redus de memorie, fapt indicat prin **small**. Un model de memorie specifică modul în care codul şi datele sunt adresate, ori sunt în acelaşi segment fizic ori în mai multe segmente. Când toate datele (sau tot codul) se află în acelaşi segment atunci elementele sunt adresate prin adrese *near* date de deplasarea lor (offset) față de adresa de început a segmentului. Dacă se utilizează mai multe segmente atunci elementele sunt adresate prin adrese *far* date de adresa segmentului şi offset. Tipuri standard de modele sunt: SMALL, MEDIUM, COMPACT şi LARGE.

	Model de memorie			
Tip adrese	**SMALL**	**MEDIUM**	**COMPACT**	**LARGE**
Adrese COD	**near**	**near**	**far**	**far**
Adrese DATE	**near**	**far**	**near**	**far**

.stack : Altă linie care descrie programul. Instrucțiunea indică locul în care începe segmentul de stivă. Acesta este utilizat ca zonă temporară de stocare a rezultatelor intermediare sau pentru a conserva starea sistemului (descrisă de valorile din registre) înainte de a efectua operații care să altereze datele existente. De asemenea stiva este utilizată şi în transferul parametrilor către şi din proceduri. Indiferent dacă este utilizată sau nu este obligatorie definirea segmentului de stivă deoarece un program .EXE trebuie să aibă o stivă. Opțional se poate defini şi dimensiunea stivei ca număr de octeți. Dacă nu se specifică dimensiunea se iau implicit 1024 de bytes.

.data : indică faptul că începe segmentul de date. Implicit, reprezintă terminarea segmentului de stivă. In segmentul de date sunt definite variabilele cu care se lucrează în program.

a db 10: se defineşte o variabilă de tip byte cu valoarea zecimală 10.

.code : indică începerea segmentului de cod şi implicit, terminarea segmentului de date. Acest segment conține instrucțiunile programului.

mov AX, @data : Instrucțiunea încarcă în registrul AX adresa segmentului de date. Alte simboluri predefinite @code – adresa segmentului de cod.

mov DS,AX : Instrucțiunea încarcă în registrul segment DS adresa segmentului de date din registrul AX. Operația este necesară deoarece nu este permisă încărcarea registrului DS cu o valoare constantă (adresa).

mov AL,a: se pune în registrul AL valoarea lui a.

add AL,b: se adună la valoarea din registrul AL, valoarea lui b.

mov e,AL: variabila e este inițializată cu valoarea din registrul AL

mov AX, 4c00h : Încarcă în registrul AX valoarea hexazecimală 4c00h. Acest lucru este necesar apelării ulterioare a rutinei 21h.

int 21h : Apelul întreruperii 21h. In registrul AH se găseşte valoarea 4ch deoarece AX are valoarea 4c00h. Pentru rutina DOS acest lucru semnifică ieşire din program. Valoarea din AL, adică 00h, reprezintă codul de ieşire din program ce indică terminare execuție fără eroare.

.end **:** marchează sfârşitul fişierului sursă. **ATENTIE:** Dacă se specifică numele unei etichete definită anterior (de obicei înainte de prima instrucțiune) atunci aceasta reprezintă adresa din CS la care porneşte execuția programului. De exemplu programul

```
...
.code
        mov AX,@data
        mov DS,AX
Start:
        mov AL,a
        add AL,b
        mov e,AL
      ...
end Start
```

rulează începând cu instrucțiunea *mov AL,a*.

Exemplu 2:

```
; programul afişează un mesaj
;-----------------------------------------------------------------------
.model small
.stack
.data
mesaj   db "Afisare mesaj !!!","$"

.code

main   proc
   mov   AX,seg mesaj
   mov   DS,AX

   mov   AH,09
   lea   DX,mesaj
   int   21h

   mov   AX,4c00h
   int   21h
main   endp
```

```
end main
```

Descriere program:

main proc : codul poate fi structurat prin intermediul procedurilor. Pentru a face analogie cu programul C care trebuie să conțină obligatoriu o procedură numită *main,* se defineşte procedura şi în programul assembler. Instrucțiunea indică începutul subprogramului numit *main*. Sfârşitul subprogramului este indicat de instrucțiunea *end main*.

mesaj db " Afisare mesaj !!!" : defineşte o variabilă şir de caractere numită *message* ce conține textul **Afisare mesaj !!!**.

mov AX, seg mesaj : Instrucțiunea încarcă în registrul AX adresa segmentului de date. Reprezintă o alternativă la instrucțiunea *mov AX, @data* pentru că instrucțiunea *seg* întoarce adresa segmentului în care se află definită variabila respectivă. Situația este posibilă pentru că fiecare variabilă este identificată printr-o adresa de forma *segment:offset*. Fără a încărca registrul DS cu adresa segmentului activ de date nu se încarcă corect adresa variabilei *mesaj*.

mov AH, 09 : încarcă în registrul AH valoarea constantă 09. Este necesar pentru a afişa un mesaj pe ecran utilizând întreruperea DOS 21h.

lea DX, message : Instrucțiunea LEA (Load Efective Address) încarcă în registrul DX offset-ul din cadrul segmentului de date la care se găseşte variabila *mesaj* . Acest lucru este necesar pentru a afişa mesajul pe ecran utilizând întreruperea 21h.

int 21h : Instrucțiunea generează o întrerupere DOS. Procesorul apelează rutina indicată de numărul întreruperii, în acest caz o rutină DOS. Procedura verifică registrul AH pentru a vedea ce trebuie să facă. In acest caz, valoarea 09 din AH indică faptul că trebuie să scrie pe ecran un şir de biți.

FORMA ASSEBLER A UNUI PROGRAM .EXE

.model
.stack
.data
    ….
.code
    …
end

Instrucțiuni utilizate:

INSTRUCȚIUNEA MOV

- realizează principalele operații de transfer a valorilor;
- este atât de utilizată încât este imposibil să scrii un program assembler fără această instrucțiune;
- forma analitică a instrucțiunii este:

MOV *destinație*, *sursă*

şi realizează copierea valorii de la *sursă* la *destinație* fără a modifica sursa.

- permite multiple combinații de tipuri ale sursei şi destinației (registru, valoare imediată, locație de memorie) însă exclude situațiile:
    - destinația nu poate fi segmentul CS; cum segmentul CS conține întotdeauna adresa codului de executat nu este indicat a se modifica această valoarea în timpul prelucrării.; singurul mod de a modifica CS este prin intermediul instrucțiunilor *int, jmp* sau *call*;
    - destinația şi sursa nu pot fi în acelaşi timp operanzi din memorie (variabile definite în segmentul de date); această restricție se aplică tuturor instrucțiunilor procesoarelor 80x86 care necesită doi operanzi; exemplu:

```
.data
        vb1 DW 300
        vb2 DW ?
….
mov vb2,vb1   ;EROARE
```

pentru a evita situația se utilizează un registru asemenea unei zone temporare; de exemplu:

```
.data
        vb1 DW 300
        vb2 DW ?
….
mov AX, vb1
mov vb2,AX
```

    - dacă sursa este o valoare imediată (constantă valorică), destinația nu poate fi unul dintre registrele de segment (CS, DS, ES, SS); pentru a evita situația se utilizează un registru;
    - destinația nu poate fi un operand imediat (constantă valorică); exemplu:

```
mov 5, AX      ;EROARE
```

- cea mai rapidă combinație este MOV registru, registru;

Instrucțiunea XCHG

- interschimbă valorile din *sursă* şi *destinaţie*;
- forma analitică:

XCHG *destinaţie*, *sursă*

- nu afectează nici un Flag bit;

Instrucţiunile LDS şi LES

- sunt printre puţinele instrucţiuni care prelucrează un dublu cuvânt (32 de biţi); transferă dublu-cuvântul din memorie către 2 registre de 16 biţi; valoarea din cei doi octeţi superiori ai dublu-cuvântului este copiată într-unul din registrele de segment (fie DS sau ES în funcţie de instrucţiune LDS sau LES); cuvântul (16 biţi) mai puţin semnificativ este copiat într-un registru general indicat ca operand destinaţie în instrucţiune; de obicei dublu-cuvântul este un pointer ce conţine adresa unei variabile dată de segment:offset (adresa segmentului pe 16 biţi şi offestul în cadrul segmentului tot pe 16 biţi);
- încarcă adresă în segment;
- forma analitică:

LDS *registru*, *sursă*
LES *registru*, *sursă*

  unde sursa este reprezentată de o variabilă definită pe 32 de biţi;
- registrul FLAG este neafectat;
- restricţie: *registru* nu poate fi unul din registrele de segment;
- exemplu:

```
mesaj db "Afisare mesaj !!!","$"
pointer_mesaj dd mesaj
…
lds DX,pointer_mesaj ; încarcă în DS adresa segmentului în care se află mesaj şi în DX
                     ; offest-ul
```

sau dacă DS este deja încărcat cu adresa segmentului atunci

```
les DX, mesaj        ;încarcă în ES adresa segmentului şi în DX offset-ul
```

ultima instrucţiune poate fi înlocuită cu:

```
mov DX, OFFSET mesaj ; OFFSET returnează offset-ul variabilei mesaj
```

Instrucţiunea ADD

- adună la operandul destinaţie valoarea operandului sursă;

- adunarea se realizează pe 16 biți;
- forma analitică:

ADD *destinație*, *sursă*

- permite multiple combinații de tipuri ale sursei şi destinației (registru, valoare imediată, locație de memorie) însă exclude situațiile:
  - destinația nu poate fi segmentul CS; cum segmentul CS conține întotdeauna adresa codului de executat nu este indicat a se modifica această valoarea în timpul prelucrării.; singurul mod de a modifica CS este prin intermediul instrucțiunilor *int, jmp* sau *call*;
  - destinația şi sursa nu pot fi în acelaşi timp operanzi din memorie (variabile definite în segmentul de date); această restricție se aplică tuturor instrucțiunilor procesoarelor 80x86 care necesită doi operanzi; exemplu:

```
.data
        vb1 DW 300
        vb2 DW ?
….
add vb2,vb1     ;EROARE
```

pentru a evita situația se utilizează un registru asemenea unei zone temporare; de exemplu:

```
.data
        vb1 DW 300
        vb2 DW ?
….
mov AX, vb1
add vb2,AX
```

  - dacă sursa este o valoare imediată (constantă valorică), destinația nu poate fi unul dintre registrele de segment (CS, DS, ES, SS); pentru a evita situația se utilizează un registru;
  - destinația nu poate fi un operand imediat (constantă valorică); exemplu:

```
add 5, AX       ;EROARE
```

- registru FLAG modificat: OF, CF, PF, SF, AF, ZF;

Instrucțiunea SUB (SUBstract)

- scade din operandul destinație valoarea operandului sursă;
- forma analitică:

SUB *destinație*, *sursă*

- restricții identice ca la instrucțiunea ADD;
- exemple:

```
.data
        vb1 DW 300
        vb2 DW 50
….
mov AX, vb1
mov CX, vb2
sub vb1, 50
sub AX, CX
sub AX, vb1
sub AX,10
```

Instrucțiunea NEG

- utilizat pentru a nega operandul, scăzând-ul din valoarea 0;
- forma analitică:

NEG *operand*

- registru FLAG afectat: OF,SF, ZF, AF, PF, CF;
- operandul trebuie să fie registru sau variabilă;
- când operandul conține valoarea minimă posibilă aceasta nu este afectată;

Instrucțiunea MUL (MULtiply)

- multiplică o valoare unsigned din AL (când operandul este de tip byte) sau din AX (când operandul este de tip word) cu valoarea operandului specificat; rezultatul este returnat în AX când operandul este de tip byte şi în DX:AX (cei 2 octeți superiori în DX) când acesta este de tip word;
- forma analitică:

MUL *operand*

- registru FLAG modificat: OF şi CF sunt setați dacă partea superioară a rezultatului (DX dacă operandul este de tip word, sau AH este de tip byte) este diferită de 0;alte flag-uri SF, ZF, AF, PF;
- operandul poate fi alt registru sau o variabilă;
- dacă valoarea înmulțită are semn atunci se utilizează IMUL;

Instrucțiunea DIV (DIVide)

- împarte o valoare unsigned din AX (când operandul este de tip byte; în acest caz

DX trebuie să conțină valoarea 0) sau din DX:AX (când operandul este de tip word) cu valoarea operandului specificat; câtul este returnat în AL iar restul în AH când operandul este de tip byte şi în DX:AX (DX conține restul şi AX câtul) când acesta este de tip word;

- forma analitică:

DIV *operand*

- registru FLAG modificat: nedefiniți;
- operandul poate fi alt registru sau o variabilă;
- în cazul operandului de tip word memorarea deîmpărțitului în DX:AX se face cu instrucțiunea **cwd** (convert word to double) ce extinde valoarea din AX în zona DX:AX; exemplu:

```
vb1 dw 55
vb2 dw 15
cat dw ?
rest dw ?
...
mov AX,vb1
cwd
div vb2
mov cat,AX
mov rest,DX
```

Instrucțiunea ROR (ROtate Right)

- utilizată pentru a roti la dreapta biții destinației de atâtea ori cât este specificat; cum fiecare bit este rotit la dreapta, cel mai puțin semnificativ bit al destinației este copiat în locul celui mai semnificativ bit şi în CF (carry flag);
- forma analitică:

ROR *destinație*, *contor*

- destinația poate fi registru sau variabilă; contorul poate fi constantă sau registrul CL; exemple:

```
vb dw 1234h
…
mov CL, 4
ror AX, CL
ror AX,4
ror vb, CL
ror vb, 4
```

- registru FLAG: OF este setat doar dacă contorul are valoarea 1 şi bitul cel mai

semnificativ al destinației este diferit de bitul vecin; pentru alte valori nu există regulă de modificare a lui OF; se modifică şi CF;

Instrucțiunea ROL (ROtate Left)

- utilizată pentru a roti la stânga biții destinației de atâtea ori cât este specificat; cum fiecare bit este rotit la stânga, cel mai semnificativ bit al destinației este copiat în locul celui mai puțin semnificativ bit şi în CF (carry flag);
- analogie cu ROR;

Instrucțiunea SAL sau SHL (Shift Arithmetic Left şi SHift Left)

- se utilizează pentru a muta la stânga biții din destinație cu atâtea poziții câte sunt specificate în contor; cu fiecare poziție se adaugă la stânga în bitul cel mai puțin semnificativ valoarea 0;
- forma analitică:

SAL *destinație*, *contor*

- registrul FLAG: SF, ZF şi PF; cel mai semnificativ bit al destinației este mutat în CF; OF este setat dacă contorul are valoarea 1 iar bitul de semn îşi păstrează valoarea;
- utilizat pentru a înmulți destinația o putere a lui 2;
- destinația poate fi registru sau variabilă; contorul poate fi constantă sau registrul CL;

Instrucțiunea SAR (Shift Arithmetic Right)

- se utilizează pentru a muta la dreapta biții din destinație cu atâtea poziții câte sunt specificate în contor; cu fiecare poziție se adaugă la dreapta în vecinul bitului cel mai semnificativ valoarea 0; bitul cel semnificativ (bitul de semn) îşi păstrează valoarea;
- forma analitică:

SAR *destinație*, *contor*

- registrul FLAG: SF, ZF şi PF; cel mai puțin semnificativ bit al destinației este mutat în CF; OF este setat dacă contorul are valoarea 1 iar bitul de semn şi vecinul său sunt diferiți;
- utilizat pentru a împărți un număr negativ (destinația) la o putere a lui 2;
- destinația poate fi registru sau variabilă; contorul poate fi constantă sau registrul CL;

Instrucțiunea SHR (Shift Rigth)

- se utilizează pentru a muta la dreapta biții din destinație cu atâtea poziții câte sunt specificate în contor; cu fiecare poziție se adaugă în vecinul bitului cel mai semnificativ valoarea 0;
- forma analitică:

SHR *destinație*, *contor*

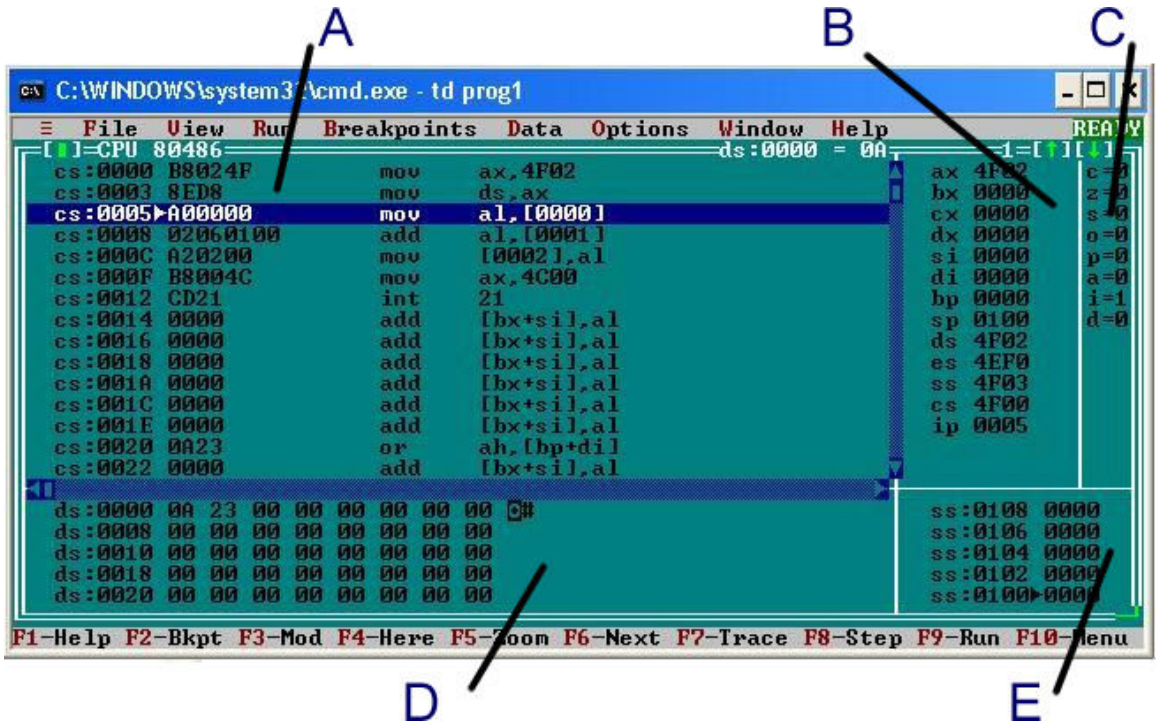
- registrul FLAG: SF, ZF şi PF; cel mai puțin semnificativ bit al destinației este mutat în CF; OF este setat dacă contorul are valoarea 1 iar bitul de semn şi vecinul său sunt diferiți;
- utilizat pentru a împărți un număr pozitiv (destinația) la o putere a lui 2;
- destinația poate fi registru sau variabilă; contorul poate fi constantă sau registrul CL;

Fişier batch pentru editare şi realizare programe assembler:

```
REM Fisier pentru dezvoltarea unui program in limbaj de asamblare
@ECHO off
ECHO Lansare editor pentru fisier sursa
:AAA
EDIT %1.asm
PAUSE
ECHO Lansare asamblor pentru celelalte fisiere: .obj,.lst,.crf
TASM %1
PAUSE
ECHO Daca in urma asamblarii apar erori, se reia editarea
IF errorlevel 1 GOTO AAA
PAUSE
ECHO Obtin executabilele
TLINK %1.obj;
ECHO Lansare in executie a programului prin TD
td %1.exe
```

Se salvează fişierul cu numele executa.bat Din linia de comandă se tastează *execută nume_program* şi automat se urmăresc paşii de realizare a unui program assembler.

Descriere FEREASTRA TURBO DEBUGGER



Fereastra A

- **descrie segmentul de cod al programului;**
- **în partea dreaptă adresele CS:0005 indică adresa în segmentul de cod a instrucțiunii respective;**
- **simbolul ►indică instrucțiunea curentă;**
- **instrucțiunile sunt executate pas cu pas utilizând tasta F8;**
- **valoarea hexa alăturată adresei CS:0005 *A00000* este reprezentarea internă a instrucțiunii *mov al, [0000]*.**

Fereastra B

- **descrie registrele şi valorile pe care acestea le au la momentul respectiv;**
- **valorile din registre sunt în hexa.**

Fereastra C

- **descrie registrul FLAG.**

Fereastra D

- **descrie segmentul de date;**
- **imediat după încărcarea adresei segmentului de date în DS, pentru a vizualiza această zonă din memorie se utilizează secvența de instrucțiuni Ctrl+G şi apoi valoarea DS:0000 în căsuța de dialog (ATENȚIE fereastra activă trebuie să fie D);**

- **utilizată pentru verificarea rezultatelor;**
- **pe coloana din dreapta sunt descrise adresele din segmentul de date (de ex. DS:0008)**

Fereastra E

- **descrie segmentul de stivă;**
- **valorile SS:0100 descriu adresele din segmentul de stivă;**
- **simbolul ►indică locația curentă în stivă;**

Bibliografie

[SOMN92] Dan SOMNEA, Vlăduț TEODOR - *Programarea în Assembler*, Ed. Tehnică, Bucureşti, 1992.

[MUSC97] Gheorghe MUSCĂ - *Programare în limbaj de Asamblare*, Ed. Teora, Bucureşti 1997.

[LUNG00] Vasile LUNGU – *Procesoare INTEL* Programare în Limbaj de asamblare, Editura TEORA, Bucureşti, 2000.

[IVAN02] Ion IVAN, Paul POCATIUL, Doru CAZAN – *Limbaje de asamblare*, Academia de Studii Economice Bucureşti, Centrul de Învățământ Economic deschis la Distanță, Bucureşti, 2002.

[IVAN97] Laur IVAN – *Programarea în limbaj de asamblare, Culegere de probleme*, Editura ASE, Bucureşti 1997.

[BARK90] Nabajyoti BARKAKATI – The Waite Group's Microsoft Macro Assembler Bible, SAMS Publishing, New York, 1990.