Gépi, nyelvi szintek

5. Probléma orientált nyelv szintje
   fordítás (fordító program)
4. **Assembly nyelv szintje**
   fordítás (assembler)
3. Operációs rendszer szintje
   részben értelmezés (operációs rendszer)
2. Gépi utasítás szintje
   ha van mikroprogram, akkor értelmezés
1. Mikroarhitektúra szintje
   hardver
0. Digitális logika szintje

Pentium 4. (T1.11. ábra)

Lapka	Dátum	MHz	Tranz.	Mem.	Megjegyzés
I-4004	1971/4	0.108	2300	640	Első egylapkás mikroproc.
I-8008	1972/4	0.108	3500	16 KB	Első 8 bites mikroroc.
I-8080	1974/4	2	6000	64 KB	Első általános célú mikroproc.
I-8086	1978/6	5-10	29000	1 MB	Első 16 bites mikroroc.
I-8088	1979/6	5-8	29000	1 MB	Az IBM PC pocesszora
I-80286	1982/6	8-12	134000	16 MB	Memória védelem
I-80386	1985/10	16-33	275000	4 GB	Első 32 bites mikroproc.
I-80486	1989/4	25-100	1.2M	4 GB	8 KB beépített gyorsítótár
Pentium	1993/5	60-233	3.1M	4 GB	Két csővezeték, MMX
P. Pro	1995/3	150-200	5.5M	4 GB	Két szintű beépített gyorsítótár
P. II	1997/5	233-400	7.5M	4 GB	Pentium Pro + MMX
P. III	1999/2	650-1400	9.5M	4 GB	SSE utasítások 3D grafikához
P. 4	2000/11	1300-3800	42M	4 GB	Hyperthreading + több SSE

Pentium 4

Nagyon sok előd (kompatibilitás!), a fontosabbak:

- **4004**: 4 bites,
- **8080**: 8 bites,
- **8086**, **8088**: 16 bites, 8 bites adat sín.
- **80286**: 24 bites (nem lineáris) címtartomány (16 K darab 64 KB-os szegmens).
- **80386**: **IA-32** architektúra, az Intel első 32 bites gépe, lényegében az összes későbbi is ezt használja.
- **Pentium II** –től **MMX** utasítások.

A Pentium 4 üzemmódjai

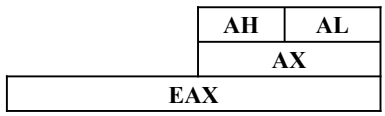
real (valós): az összes **8088** utáni fejlesztést kikapcsolja (valódi **8088**-ként viselkedik). Hibánál a gép egyszerűen összeomlik, lefagy.

virtuális 8086: a **8088**-as programok védett módban futnak (ha **WINDOWS**-ból indítjuk az **MS-DOS**-t, és ha abban hiba történik, akkor nem fagy le, hanem visszaadja a vezérlést a **WINDOWS**-nak).

védett: valódi **Pentium 4**-ként működik. Ilyenkor 4 védelmi szint lehetséges (**PSW**):
0: kernelmód (operációs r.), **1**, **2**: ritkán használt, **3**: felhasználói mód.

Általános regiszterek (32, 16 illetve 8 bitesek)

dword	word	higher byte	lower byte	
EAX	**AX**	**AH**	**AL**	Accumulátor (szorzás, osztás is)
EBX	**BX**	**BH**	**BL**	Base Register (címző)
ECX	**CX**	**CH**	**CL**	Counter Register (számláló)
EDX	**DX**	**DH**	**DL**	Data Register (szorzás, osztás, I/O)



A 8088-80286-os processzorokon a további regiszterek is 16 bitesek voltak, a 80386-os processzorral kezdődően 32 bitesre egészítették ki 16 magasabb helyértékű bittel, az így kapott regiszterek elnevezése elől egy **E** betűvel egészült ki.

A 80386-os processzorokon az általános és index regiszterek 16 bites alacsonyabb helyértékű része az eredeti névvel, a teljes regiszter az **E**-vel kiegészített névvel érhető el.

A 80386-os – Pentium processzorok tudnak 8086/8088-asként is működni (l. később).

Vezérlő regiszterek (32, eredetileg 16 bitesek)

- **EIP** (Instruction Pointer) utasítás számláló: az éppen végrehajtandó utasítás logikai címét (80286-ig **IP** az **CS** által mutatott szegmensbeli relatív címet) tartalmazza
- **ESP** (Stack Pointer) verem mutató: a stack-be (verembe) utolsónak beírt elem címét (80286-ig **SP** az **SS** által mutatott szegmensbeli relatív címet) tartalmazza
- **EFLAGS (PSW)** a processzor állapotát jelző regiszter
- **EBP** (Base Pointer) a stack indexelt címzéséhez használatos
- **ESI** (Source Index) a forrás adat terület indexelt címzéséhez használatos (**SI** az **ESI** 16 bites része)
- **EDI** (Destination Index) a cél adat terület indexelt címzéséhez használatos (**DI** az **EDI** 16 bites része)

Szegmens regiszterek (16 ill. 32 bitesek)

A szegmens regiszterek bevezetésének eredeti célja az volt, hogy nagyobb memóriát lehessen elérni.

- **CS** (Code Segment) utasítások címzéséhez
- **SS** (Stack Segment) verem címzéséhez
- **DS** (Data Segment) (automatikus) adat terület címzéséhez
- **ES** (Extra Segment) másodlagos adat terület címzéséhez
- **FS** (nincs külön neve) 80386-tól
- **GS** (nincs külön neve) 80386-tól

Memóriaszervezés:

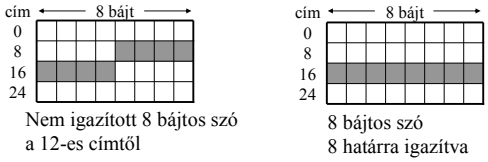
- A **CS**, **SS**, **DS**, **ES**, **FS**, **GS** regiszterek a visszafelé kompatibilitást biztosítják a régebbi gépekkel.
- **16 K db szegmens** lehetséges, de a **WINDOWS**-ok és **UNIX** is csak **1** szegmenst támogatnak, és ennek is egy részét az operációs rendszer foglalja el,
- minden szegmensen belül a címtartomány: **0 - 2^{32}-1**, 80288-ig a szegmens regiszterek 16 bitesek, a szegmensek lehetséges kezdőcíme **(0-2^{16}-1)*16**, a szegmensen belül a címtartomány: **0 - 2^{16}-1**
- **Little endian** tárolási mód: az alacsonyabb címen van az alacsonyabb helyértékű bájt.

Memória modellek

ASCII kód 7 bit + paritás → **Byte** (**bájt**)

Szó: 2 vagy 4 bájt, dupla szó 8 bájt.

Igazítás (alignment), **5.2. ábra**: hatékonyabb, de probléma a kompatibilitás (a **Pentium 4**-nek két ciklusra is szüksége lehet egy szó beolvasásához).



Az Intel 8086/8088 – Pentium társszervezése

A memória byte szervezésű.

Egy byte 8 bitből áll. word, double word.

Byte sorrend: Little Endian (LSBfirst).

A negatív számok 2-es komplemens kódban.

szegmens, szegmens cím

a szegmensen belüli „relatív” cím, logikai cím, OFFSET, Effective Address (EA) displacement, eltolás

lineáris cím, virtuális cím, fizikai cím (Address)

A 8086/8088 **FLAGS** (**STATUS**) bitjei (flag-jei)

-	-	-	-	O	D	I	T	S	Z	-	A	-	P	-	C
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- **O** (Overflow) Előjeles túlcsordulás
- **D** (Direction) A string műveletek iránya, **0**: növekvő, **1**: csökkenő
- **I** (Interrupt) **1**: Maszkolható megszakítás engedélyezése , **0**: tiltása
- **T** (Trap) **1**: „single step” (debug), **0**: Automatikus üzemmód
- **S** (Sign) Az eredmény legmagasabb helyértékű bit-je (előjel bit)
- **Z** (Zero) **1** (igaz), ha az eredmény **0**, különben **0** (hamis)
- **A** (Auxiliary Carry) Átvitel a 3. és 4. bit között (decimális aritmetika)
- **P** (Parity) Az eredmény alsó 8 bitjének paritása
- **C** (Carry) Átvitel előjel nélküli műveleteknél

A 80286 **FLAGS** (**STATUS**) bitjei kiegészültek

-	NT	IOPL		O	D	I	T	S	Z	-	A	-	P	-	C
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- **NT** (Nested Task) A jelenleg futó taszk védett módon egy másik taszkba van ágyazva
- **IOPL** (Input/Output Privilege Level) A taszk privilégium szintje CPL. A taszk csak akkor férhet hozzá az I/O porthoz, ha CPL ≤ IOPL.
  00 a legmagasabb, 11 a legalacsonyabb privilégium szint.

A 80386 **EFLAGS** magasabb helyértékű bitjei

80386-tól **FLAGS** további 16 bittel egészült ki, az eddigi bitek jelentése megmaradt.

														VM	RF
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

- **RF** (Resume) Nyomkövetésnél használatos. Ha 0, akkor minden utasítás végrehajtása után debug kivétel (#DE – debug exception) generálódik.
- **VM** (Virtual Mode) Egy vagy több 1 MB-s DOS memória partíciót engedélyez, több DOS program futhat egyidejűleg védett módban a gépen

A 80486 **EFLAGS** 18. bitje

													AC	VM	RF
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

- **AC** (Alignment Check) Ha be van állítva és word-re hivatkozunk páratlan címen vagy doubleword-re 4-gyel nem osztható címen, akkor felhasználói (3-as privilégium) szinten kivétel (exception) képződik.

A Pentium **EFLAGS** további bitje

										ID	VIP	VIF	AC	VM	RF
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

- **VIF** (Virtual Interrupt Flag) virtuális módú **I** másolata.
- **VIP** (Virtual Interrupt Pending) Ha 1, akkor egy megszakítás függőben van.
- **ID** (IDentification) Ha 1, akkor a processzor azonosítani tudja magát (támogatja a **CPUID** utasítást).

A fizikai cím meghatározása

valós (real) védett (protected)

szegmens cím

szegmens regiszter

↓

szegmens regiszter tartalma * 16

descriptor tábla elem

↓

szegmens kezdőcíme

fizikai cím

szegmens kezdőcíme + szegmensen belüli cím

Deszkriptor formátumok (8 bájt)

80286

00000000	**00000000**	**6**
Elérési jogok	**B23-B16**	**4**
Base (B15-B0)		**2**
Limit (L15-L0)		**0**

80386 – 80486 – Pentium

B31-B24	**G**	**D**	**0**	**AV**	**L19-L16**	**6**
Elérési jogok	**B23-B16**					**4**
Base (B15-B0)						**2**
Limit (L15-L0)						**0**

G = 0: limit bájtokban **G = 1:** 4 KB-os lapokban

D = 0: 16 bites mód **D = 1:** 32 bites mód

AV = 0: a szegmens nem érhető el **AV = 1:** elérhető

Az I8086/88 címzési rendszere

Operandus megadás

Adat megadás

- **Kódba épített adat** (immediate – közvetlen operandus)

```
MOV  AL, 6        ; AL új tartalma 6
MOV  AX, 0FFH     ; AX új tartalma 000FFH
```

- **Regiszter címzés:**

```
MOV  AX, BX
```

Az egyik cím mindig regiszter!

A többi adat megadás esetén az automatikus szegmens regiszter: **DS**

Direkt memória címzés: a címrészen az operandus logikai címe (eltolás, displacement)

```
MOV     AX, SZO ; AX új tartalma SZO tartalma
MOV     AL, KAR ; AL új tartalma KAR tartalma
```

Valahol a **DS** által mutatott szegmensben:

```
SZO     DW      1375H
KAR     DB      3FH
```

(DS:SZO) illetve **(DS:KAR)**

```
MOV     AX, KAR                 ; hibás
MOV     AL, SZO                 ; hibás
MOV     AX, WORD PTR KAR        ; helyes, de …
MOV     AL, BYTE PTR SZO        ; helyes, de …
```


- **Indexelt címzés:** a logikai cím: a 8 vagy 16 bites eltolás + **SI** vagy **DI (**esetleg **BX)** tartalma

```
MOV     AX, 10H[SI]
MOV     AX, -10H[SI]
MOV     AX, [SI]
```

Regiszter-indirekt címzés: eltolási érték nélküli indexelt címzés

```
MOV     AX, [BX]
MOV     AX, [SI]
```

- **Bázis relatív (bázisindex) címzés:** a logikai cím: eltolás + **BX** + **SI** vagy **DI** tartalma

```
MOV     AX, 10H[BX][SI]
MOV     AX, [BX+SI+10H]
```


Stack (verem) terület címzés

Automatikus szegmens regiszter: **SS**

Megegyezik a bázis relatív címzéssel, csak a **BX** regiszter helyett a **BP** szerepel.

Program terület címzés

Automatikus szegmens regiszter: **CS**

A végrehajtandó utasítás címe: **(CS:IP)**

Egy utasítás végrehajtásának elején:
IP = IP + az utasítás hossza.

- **IP relatív címzés:**
  IP = IP + a 8 bites előjeles közvetlen operandus
- **Direkt utasítás címzés:** Az operandus annak az utasításnak a címe, ahova a vezérlést átadni kívánjuk.

  Közeli **(NEAR):** **IP** <= a 16 bites operandus

  Távoli **(FAR):** **(CS:IP)** <= a 32 bites operandus.

```
CALL  VALAMI    ; az eljárás típusától függően
                ;     NEAR vagy FAR
```


- **Indirekt utasítás címzés:** Bármilyen adat címzési móddal megadott szóban vagy dupla szóban tárolt címre történő vezérlés átadás. Pl.:

```
JMP     AX    ; ugrás az AX-ben tárolt címre
JMP     [BX] ; ugrás a (DS:BX) által címzett
              ; szóban tárolt címre.
JMP     FAR [BX] ; ugrás a (DS:BX) által
               ; címzett dupla szóban tárolt címre.
```

Az utasítások szerkezete

prefixum	*operációs kód*	*címzési mód*	*operandus*
0 - 2 byte	1 byte	0 - 1 byte	0 - 4 byte

***Prefixum*:**
utasítás ismétlés, explicit szegmens megadás vagy **LOCK**

```
MOV     AX, CS:S      ; S nem a DS,
                      ; hanem a CS regiszterrel címzendő
```

***Operációs kód*:** szimbolikus alakját mnemonic-nak nevezzük

***Címzési mód byte*:** hogyan kell az operandust értelmezni

***Operandus*:** mivel kell a műveletet elvégezni

Címzési mód byte

A legtöbb utasítás kód után szerepel. Szerkezete:

7	6	5	4	3	2	1	0
Mód		Regiszter			Reg/Mem		

Ha a műveleti kód legalacsonyabb helyértékű bit-je
0, akkor **byte**-os művelet,
1, akkor **word**-ös (szavas) művelet.

Regiszter			Reg/Mem jelentése, ha Mód =			
	byte	word	00	01	10	11
000	AL	AX	BX + SI + DI	„00” + 8 bit displ.	„00” + 16 bit displ.	Regiszter
001	CL	CX				
010	DL	DX	BP + SI + DI			
011	BL	BX				
100	AH	SP	SI DI			
101	CH	BP				
110	DH	SI	16 bit d.	BP+8 bit d.	BP+16 bit d.	
111	BH	DI	BX	„00”+8 bit	„00”+16 bit	

Szimbolikus alakban az operandusok sorrendje, gépi utasítás formájában a gépi utasítás kód mondja meg a regiszter és a memória közti adatátvitel irányát. Pl. az alábbi két utasítás esetén a címzési mód byte megegyezik:

```
MOV     AX, 122H[SI+BX]
        ; hexadecimálisan 8B 80 0122
MOV     122H[SI+BX], AX
        ; hexadecimálisan  89 80 0122
```

7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0
Mód		Reriszter			Reg/Mem		
+16 bit d.		AX			SI+BX		

Az általános regiszterek és **SI**, **DI**, **SP**, **BP** korlátlanul használható, a többi (a szegmens regiszterek, **IP** és **STATUS**) csak speciális utasításokkal. Pl.:

```
MOV     DS, ADAT    ; hibás!

MOV     AX, ADAT    ; helyes!
MOV     DS, AX      ; helyes!
```

A „többi” regiszter nem lehet aritmetikai utasítás operandusa, sőt, **IP** és **CS** csak vezérlés átadó utasításokkal módosítható, közvetlenül nem is olvasható.


```
; Assembly főprogram, amely adott szöveget ír a képernyőre
; ==========================================================
KOD  SEGMENT PARA PUBLIC 'CODE' ; Szegmens kezdet
; KOD: a szegmens neve
; align-type (igazítás típusa): BYTE, WORD, PARA, PAGE
; combine-type: PUBLIC, COMMON, AT <kifejezés>, STACK
; class: 'CODE', 'DATA', ('CONSTANT',) 'STACK', 'MEMORY'
;              ajánlott értelemszerűen
     ASSUME  CS:KOD, DS:ADAT, SS:VEREM, ES:NOTHING
; feltételezett szegmens regiszter értékek.
; A beállításról ez az utasítás nem gondoskodik!
```

```
KIIR    PROC    FAR ; A fő eljárás mindig FAR
                    ; FAR: távoli, NEAR: közeli eljárás
; Az operációs rendszer úgy hívja meg a főprogramokat, hogy
; a CS és IP a program végén lévő END utasításban megadott
; címke szegmens és OFFSET címét tartalmazza, SS és SP a
; a STACK kombinációs típusú szegmens végét mutatja,
; a visszatérés szegmens címe DS-ben van, OFFSET-je pedig 0
        PUSH    DS      ; DS-ben van a visszatérési cím
                        ; SEGMENT része
        XOR     AX, AX  ; AX⇐0, az OFFSET rész = 0
        PUSH    AX      ; Veremben a (FAR) visszatérési cím
        MOV     AX, ADAT; AX⇐ az ADAT SEGMENT címe
        MOV     DS, AX
; Most már teljesül, amit az ASSUME utasításban írtunk
; Eddig tartott a főprogram előkészületi része
```



```
        MOV     SI, OFFSET SZOVEG
                        ; SI⇐SZÖVEG OFFSET címe
        CLD             ; a SZÖVEGet növekvő címek
                        ; szerint kell olvasni
        CALL    KIIRO   ; Eljárás hívás

        RET             ; Visszatérés az op. rendszerhez
                        ; a veremből visszaolvasott
                        ; szegmens és OFFSET címre

KIIR    ENDP            ; A KIIR eljárás vége
```



```
KIIRO   PROC            ; NEAR eljárás,
                        ; megadása nem kötelező
CIKLUS: LODSB           ; AL⇐a következő karakter
        CMP     AL, 0   ; AL =? 0
        JE      VEGE    ; ugrás a VEGE címkéhez,
                        ; ha AL=0
        MOV     AH, 14  ; BIOS rutin paraméterezése
        INT     10H     ; a 10-es interrupt hívása:
                        ; az AL-ben lévő karaktert kiírja
                        ; a képernyőre
        JMP     CIKLUS  ; ugrás a CIKLUS címkéhez,
                        ; a kiírás folytatása
VEGE:   RET             ; Visszatérés a hívó programhoz
KIIRO   ENDP            ; A KIIRO eljárás vége
KOD     ENDS            ; A KOD szegmens vége
```



```
ADAT    SEGMENT     PARA    PUBLIC  'DATA'
SZOVEG  DB          'Ezt a szöveget kiírja a képernyőre'
        DB          13, 10, 0   ; 13: a kocsi vissza,
                                ; 10: a soremelés kódja,
                                ;  0: a szöveg vége jel
ADAT    ENDS                    ; Az ADAT szegmens vége
; ================================================
VEREM   SEGMENT     PARA    STACK
        DW          100 DUP (?)     ; Helyfoglalás 100 db
                                    ; inicializálatlan szó számára
VEREM   ENDS                        ; A VEREM szegmens vége
; ================================================
        END         KIIR    ; Modul vége,
                            ; a program kezdőcíme: KIIR
```


Az I8086/8088 utasítás rendszere

Jelölések

⇐ : értékadás

⇔ : felcserélés

op, op1, op2: tetszőlegesen választható operandus (közvetlen, memória vagy regiszter).

op1 és **op2** közül az egyik regiszter kell legyen!

reg: általános, bázis vagy index regiszter

mem: memória operandus

ipr: (8 bites) **IP** relatív cím

port: port cím (8 bites eltolás vagy **DX**)

[op]: az **op** által mutatott cím tartalma

Adat mozgató utasítások

Nem módosítják a flag-eket (kivéve **POPF** és **SAHF**)

```
MOV   op1, op2  ; op1 ⇐ op2 (MOVe)
XCHG  op1, op2  ; op1 ⇔ op2 (eXCHanGe), op2 sem
                ; lehet közvetlen operandus
XLAT            ; AL ⇐ [BX+AL] (trans(X)LATe), a
                ; BX által címzett maximum 256 byte-
                ; os tartomány AL-edik byte-jának
                ; tartalma lesz AL új tartalma
LDS   reg, mem  ; reg ⇐ mem, mem+1
                ; DS ⇐ mem+2, mem+3 (Load DS)
LES   reg, mem  ; reg ⇐ mem, mem+1
                ; ES ⇐ mem+2, mem+3 (Load ES)
LEA   reg, mem  ; reg ⇐ mem effektív (logikai) címe
                ; (Load Effective Address)
```

A veremmel (stack-kel) kapcsolatos adat mozgató utasítások:

```
PUSH    op   ; SP ⇐ SP-2; (SS:SP) ⇐ op
PUSHF        ; (PUSH Flags)
             ; SP ⇐ SP-2; (SS:SP) ⇐ STATUS
POP     op   ; op ⇐ (SS:SP); SP ⇐ SP+2
POPF         ; (POP Flags)
             ; STATUS ⇐ (SS:SP); SP ⇐ SP+2
```

Az Intel 8080-nal való kompatibilitást célozza az alábbi két utasítás:

```
SAHF         ; STATUS alsó 8 bitje ⇐ AH
LAHF         ; AH ⇐ STATUS alsó 8 bitje
```


Aritmetikai utasítások

```
ADD   op1, op2  ; op1 ⇐ op1 + op2   (ADD)
```

Pl.: előjeles/előjel nélküli számok összeadása

```
MOV   AX, -1    ; AX=-1 (=0FFFFH)
ADD   AX, 2     ; AX=1, C=1, O=0

ADC   op1, op2  ; op1 ⇐ op1 + op2 + C
                ; (ADD with Carry)
```

Pl.: két szavas összeadás (előjeles/előjel nélküli)

```
ADD   AX, BX
ADC   DX, CX    ; (DX:AX) = (DX:AX) + (CX:BX)

INC   op    ; op ⇐ op + 1, C változatlan! (INCrement)
```



```
SUB   op1, op2  ; op1 ⇐ op1 - op2 (SUBtraction)
CMP   op1, op2  ; flag-ek op1 - op2 szerint (CoMPare)
SBB   op1, op2  ; op1 ⇐ op1 - op2 – C:
                ; a több szavas kivonást segíti.
DEC   op        ; op ⇐ op - 1, C változatlan
                ; (DECrement)
NEG   op        ; op ⇐ -op (NEGate)
```


Az összeadástól és kivonástól eltérően a szorzás és osztás esetében különbséget kell tennünk, hogy előjeles vagy előjel nélküli számábrázolást alkalmazunk-e. További lényeges eltérés, hogy két 8 bites vagy 16 bites mennyiség szorzata ritkán fér el 8 illetve 16 biten, ezért a szorzás műveletét úgy alakították ki, hogy 8 bites tényezők szorzata 16, 16 biteseké pedig 32 biten keletkezzék:

Szorzásnál **op** nem lehet közvetlen operandus!

```
MUL   op    ; előjel nélküli szorzás (MULtiplicate),
IMUL  op    ; előjeles szorzás (Integer MULtiplicate).
```

Ha op 8 bites AX ⇐ AL * op.

Ha op 16 bites (DX:AX) ⇐ AX * op.

Osztásnál **op** nem lehet közvetlen operandus!

```
DIV   op    ; (DIVide) előjel nélküli osztás,
IDIV  op    ; (Integer DIVide) előjeles osztás,
            ; A nem 0 maradék előjele megegyezik
            ; az osztóéval.
```

Ha op 8 bites: AL ⇐ AX/op hányadosa,
AH ⇐ AX/op maradéka.

Ha op 16 bites: AX ⇐ (DX:AX)/op hányadosa,
DX ⇐ (DX:AX)/op maradéka.

Osztásnál **túlcsordulás** → azonnal elhal (abortál) a programunk!

Ha bájtot bájttal vagy szót szóval akarunk osztani, akkor:

- Előjel nélküli osztás előkészítése **AH** illetve **DX** nullázásával történik.
- Előjeles osztás előkészítésére szolgál az alábbi két előjel kiterjesztő utasítás:

```
CBW         ; (Convert Byte to Word)
            ; AX ⇐ AL előjel helyesen
CWD         ; (Convert Word to Double word)
            ; (DX:AX) ⇐ AX előjel helyesen
```

Pozitív számok esetén (az előjel **0**) az előjel kiterjesztés az **AH** illetve a **DX** regiszter nullázását, negatív számok esetén (az előjel **1**) csupa **1**-es bittel való feltöltését jelenti.

Az előjel kiterjesztés máskor is alkalmazható.

```
; Két vektor skalár szorzata. 1. változat
code    segment  para public 'code'
        assume   cs:code, ds:data, ss:stack, es:nothing

skalar  proc     far
        push     ds          ; visszatérési cím a verembe
        xor      ax,ax       ; ax ⇐ 0
        push     ax          ; visszatérés offset címe
        mov      ax,data     ; ds a data szegmensre mutasson
        mov      ds,ax       ; sajnos „mov ds,data"
                             ; nem megengedett
```



```
; A       skalár szorzat számítása
        mov      cl,n        ; cl ⇐ n, 0 ≤ n ≤ 255
        xor      ch,ch       ; cx = n szavasan
        xor      dx,dx       ; az eredmény ideiglenes helye
        JCXZ     kesz        ; ugrás a kesz címkére,
                             ; ha CX (=n) = 0
        xor      bx,bx       ; bx ⇐ 0,
                             ; bx-et használjuk indexezéshez
```



```
ism:  mov      al,a[bx]    ; al ⇐ a[0], később a[1], ...
      imul     b[bx]       ; ax ⇐ a[0]*b[0], a[1]*b[1], ...
      add      dx,ax       ; dx ⇐ részösszeg
      inc      bx          ; bx ⇐ bx+1, az index növelése

; B       ciklus vége
      dec      cx          ; cx ⇐ cx-1, (vissza)számlálás
      JCXZ     kesz        ; ugrás a kész címkére, ha cx=0
      jmp      ism         ; ugrás az ism címkére
kesz: mov      ax,dx       ; a skalár szorzat értéke ax-ben
```



```
; C       eredmények kiírása
        call     hexa        ; az eredmény kiírása
                             ; hexadecimálisan
        mov      si,offset kvse ; kocsi vissza soremelés
        call     kiiro       ; kiírása
        ret                  ; vissza az Op. rendszerhez
skalar  endp                 ; a skalár eljárás vége
; D
```



```
hexa  proc                   ; ax kiírása hexadecimálisan
      xchg     ah,al         ; ah és al felcserélése
      call     hexa_b        ; al (az eredeti ah) kiírása
      xchg     ah,al         ; ah és al visszacserélése
      call     hexa_b        ; al kiírása
      ret                    ; visszatérés
hexa  endp                   ; a hexa eljárás vége
; ------------------------------------------------------
hexa_b  proc                 ; al kiírása hexadecimálisan
      push     cx            ; mentés a verembe
      mov      cl,4          ; 4 bit-es rotálás előkészítése
      ROR      al,CL         ; az első jegy az alsó 4 biten
      call     h_jegy        ; az első jegy kiírása
      ROR      al,CL         ; a második jegy az alsó 4 biten
      call     h_jegy        ; a második jegy kiírása
      pop      cx            ; visszamentés a veremből
      ret                    ; visszatérés
hexa_b  endp                 ; a hexa_b eljárás vége
```



```
h_jegy    proc               ; hexadecimális jegy kiírása
      push     ax            ; mentés a verembe
      AND      al,0FH        ; a felső 4 bit 0 lesz,
                             ; a többi változatlan
      add      al,'0'        ; + 0 kódja
      cmp      al,'9'        ; ≤ 9 ?
      JLE      h_jegy1       ; ugrás h_jegy1 -hez, ha igen
      add      al,'A'-'0'-0AH    ; A-F hexadecimális jegyek
                             ; kialakítása
h_jegy1:  mov  ah,14         ; BIOS szolgáltatás előkészítése
      int      10H           ; BIOS hívás: karakter kiírás
      pop      ax            ; visszamentés a veremből
      ret                    ; visszatérés
h_jegy    endp               ; a hexa_b eljárás vége
```

```
kiiro  proc                ; szöveg kiírás (DS:SI)-től
       push    ax
       cld
ki1:   lodsb               ; al⇐a következő karakter
       cmp     al, 0       ; al =? 0
       je      ki2         ; ugrás a ki2 címkéhez, ha al=0
       mov     ah,14       ; BIOS rutin paraméterezése
       int     10H         ; az AL-ben lévő karaktert
                           ; kiírja a képernyőre
       jmp     ki1         ; a kiírás folytatása
ki2:   pop     ax
       ret                 ; visszatérés a hívó programhoz
kiiro  endp                ; a kiíró eljárás vége
; ---------------------------------------------------------
code   ends                ; a code szegmens vége
```



```
data    segment  para public 'data'
n       db       3
a       db       1, 2, 3
b       db       3, 2, 1
kvse    db       13, 10, 0  ; kocsi vissza, soremelés
data    ends                ; a data szegmens vége
; ===========================================
stack   segment  para stack 'stack'
        dw   100 dup (?)    ; 100 word legyen a verem
stack   ends                ; a stack szegmens vége
; ===========================================
        end  skalar         ; modul vége,
                            ; a program kezdő címe: skalar
```


Vezérlés átadó utasítások

Eljárásokkal kapcsolatos utasítások

Eljárás hívás:

CALL op ; eljárás hívás

- közeli: *push* IP, IP ⇐ op,

- távoli: *push* CS, *push* IP, (CS:IP) ⇐ op.

Visszatérés az eljárásból:

RET ; visszatérés a hívó programhoz (RETurn)

- közeli: *pop* IP,

- távoli: *pop* IP, *pop* CS.

RET op ; . . . , SP ⇐ SP+op

; op csak közvetlen adat lehet!

Feltétlen vezérlés átadás (ugrás)

JMP op ; ha op közeli: IP ⇐ op,

; ha távoli: (CS:IP) ⇐ op.

Feltételes ugrások, aritmetikai csoport

Előjeles	Reláció	Előjel nélküli
JZ ≡ JE	=	JZ ≡ JE
JNZ ≡ JNE	≠	JNZ ≡ JNE
JG ≡ JNLE	>	JA ≡ JNBE
JGE ≡ JNL	≥	JAE ≡ JNB ≡ JNC
JL ≡ JNGE	<	JB ≡ JNAE ≡ JC
JLE ≡ JNG	≤	JBE ≡ JNA

A feltételek: Zero, **E**qual, **N**o (Not),
Greater, **L**ess, **A**bove, **B**elow, **C**arry

Feltételes ugrások, logikai csoport

a flag igaz (1)	flag	a flag hamis (0)
JZ ≡ JE	Zero	JNZ ≡ JNE
JC	Carry	JNC
JS	Sign	JNS
JO	Overflow	JNO
JP ≡ JPE	Parity	JNP ≡ JPO
JCXZ	CX = 0	

A feltételek: Zero, **E**qual, **N**o (Not), **C**arry, **S**ign,
Overflow, **P**arity **E**ven, **P**arity **O**dd.

Minden feltételes vezérlés átadás **IP** relatív címzéssel (**SHORT**) valósul meg!

Pl.:

```
        JZ   MESSZE     ; Hibás, ha
                        ; MESSZE messze van
```

Megoldás:

```
        JNZ     IDE     ; Negált feltételű ugrás
        JMP     MESSZE
IDE:    . . .
```


Ciklus szervező utasítások

IP relatív címzéssel (**SHORT**) valósulnak meg.

```
LOOP    ipr  ; CX ⇐ CX – 1, ugrás ipr -re,
             ; ha CX ≠ 0
LOOPZ   ipr  ; CX ⇐ CX – 1, ugrás ipr -re,
             ; ha (CX ≠ 0 és Z=1)
LOOPE   ipr  ; ugyanaz mint LOOPZ
LOOPNZ  ipr  ; CX ⇐ CX – 1, ugrás ipr -re,
             ; ha (CX ≠ 0 és Z=0)
LOOPNE  ipr  ; ugyanaz mint LOOPNZ
```



```
; B = A n-dik hatványa,
; A és n előjel nélküli byte, B word
; Feltétel: A n-1 –dik hatványa elfér AL –ben.
        mov     cl, n     ; a ciklus előkészítése
        xor     ch, ch
        mov     al, 1     ; lehetne:      mov ax, 1
        xor     ah, ah    ; akkor ez nem kell
        JCXZ    kesz      ; ha n=0, akkor 0-szor
                          ; fut a ciklus mag
c_mag:  mul     A         ; ciklus mag
        LOOP    c_mag     ; ismétlés, ha kell
kesz:   mov     B, ax
```


Egyszerűsítési lehetőség a skalár szorzatot kiszámító programban:

```
; B
        dec     cx        ; cx ⇐ cx-1, (vissza)számlálás
        jcxz    kesz      ; ugrás a kész címkére, ha cx=0
        jmp     ism       ; ugrás az ism címkére
kesz:   mov     ax,dx     ; a skalár szorzat értéke ax-ben
```

helyett:

```
; B
        LOOP    ism       ; ugrás az ism címkére,
                          ; ha kell ismételni
kesz:   mov     ax,dx     ; a skalár szorzat értéke ax-ben
```


Annak érdekében, hogy a skalárszorzatot kiszámító program ne rontson el regisztereket, kívánatos ezek mentése:

```
; A
        PUSH    BX  ; mentés
        PUSH    CX
        PUSH    DX
```

és visszamentése:

```
        POP     DX  ; visszamentés
        POP     CX
        POP     BX
; C
```


A paraméterek szabványos helyen történő átadása

```
; Két vektor skalár szorzata. 2. változat
. . .
; A          skalár szorzat számítása
; ELJÁRÁS HÍVÁS A PARAMÉTEREK
; SZABVÁNYOS HELYEN TÖRTÉNŐ ÁTADÁSÁVAL
        CALL    SKAL        ; ELJÁRÁS HÍVÁS
                            ; eredmény az AX regiszterben
; C          eredmények kiírása
        call    hexa        ; az eredmény kiírása
        mov     si,offset kvse ; kocsi vissza, soremelés
        call    kiiro       ; kiírása
. . .
        ret                 ; vissza az Op. rendszerhez
skalar  endp                ; a skalár eljárás vége
; D
```

```
SKAL    PROC            ; KÖZELI (NEAR) ELJÁRÁS
                        ; KEZDETE
; Az A-tól C-ig tartó program rész:
        PUSH    BX      ; MENTÉSEK
        PUSH    CX
        PUSH    DX
        mov     cl,n    ; cl ⇐ n, 0 ≤ n ≤ 255
        xor     ch,ch   ; cx = n szavasan
        xor     dx,dx   ; az eredmény ideiglenes helye
        jcxz    kesz    ; ugrás a kesz címkére, ha n=0
        xor     bx,bx   ; bx ⇐ 0,
                        ; bx-et használjuk indexezéshez
```



```
ism:    mov     al,a[bx]    ; al ⇐ a[0], később a[1], ...
        imul    b[bx]       ; ax ⇐ a[0]*b[0], a[1]*b[1],...
        add     dx,ax       ; dx ⇐ részösszeg
        inc     bx          ; bx ⇐ bx+1, az index növelése
; B          ciklus vége
        LOOP    ism         ; ugrás az ism címkére,
                            ; ha kell ismételni
kesz:   mov     ax,dx       ; a skalár szorzat értéke ax-ben
        POP     DX          ; VISSZAMENTÉSEK
        POP     CX
        POP     BX
        RET                 ; VISSZATÉRÉS A HÍVÓ
                            ; PROGRAMHOZ
;           visszatérés a C ponthoz
SKAL            ENDP        ; A SKAL ELJÁRÁS VÉGE
; D
…
```

Csak az **a** és **b** vektor skalár szorzatát tudja kiszámolni!

A paraméterek regiszterekben történő átadása

```
; Két vektor skalár szorzata. 3. változat
. . .
; A
; ELJÁRÁS HÍVÁS A PARAMÉTEREK
; REGISZTEREKBEN TÖRTÉNŐ ÁTADÁSÁVAL
        MOV     CL, n       ; PARAMÉTER BEÁLLÍTÁSOK
        XOR     CH, CH      ; CX = n, ÉRTÉK
        MOV     SI,OFFSET a ; SI ⇐ a OFFSET CÍME
        MOV     DI,OFFSET b ; DI ⇐ b OFFSET CÍME
        call    skal        ; eljárás hívás
                            ; eredmény az ax regiszterben
        call    hexa        ; az eredmény kiírása
        mov     si,offset kvse ; kocsi vissza, soremelés
        call    kiiro       ; kiírása
. . .
        ret                 ; visszatérés az Op. rendszerhez
skalar  endp                ; a skalár eljárás vége
```



```
skal    proc                ; Közeli (NEAR) eljárás kezdete
        push    bx          ; mentések
        push    cx
        push    dx
        xor     dx,dx       ; az eredmény ideiglenes helye
        jcxz    kesz        ; ugrás a kesz címkére, ha n=0
        xor     bx,bx       ; bx ⇐ 0,
                            ; bx-et használjuk indexezéshez
```



```
ism:    mov     al,[SI+BX]  ; FÜGGETLEN a-tól
        imul    BYTE PTR [DI+BX]; FÜGGETLEN b-től
; csak „BYTE PTR”-ből derül ki, hogy 8 bites a szorzás
        add     dx,ax       ; dx ⇐ részösszeg
        inc     bx          ; bx ⇐ bx+1, az index növelése
        loop    ism         ; ugrás az ism címkére,
                            ; ha kell ismételni
kesz:   mov     ax,dx       ; a skalár szorzat értéke ax-ben
        pop     dx          ; visszamentések
        pop     cx
        pop     bx
        ret                 ; visszatérés a hívó programhoz
skal    endp                ; a skal eljárás vége
; D
. . .
```

Így csak kevés paraméter adható át!


```
; Két vektor skalár szorzata. 4. változat
. . .
; A
; ELJÁRÁS HÍVÁS A PARAMÉTEREK
; VEREMBEN TÖRTÉNŐ ÁTADÁSÁVAL
        MOV     AL,n        ; AL-t nem kell menteni, mert
        XOR     AH,AH       ; AX-ben kapjuk az eredményt
        PUSH    AX          ; AX=n a verembe
        MOV     AX,OFFSET a ; AX ⇐ a OFFSET címe
        PUSH    AX          ; a verembe
        MOV     AX,OFFSET b ; AX ⇐ b OFFSET címe
        PUSH    AX          ; a verembe
```

A verembe került:

n értéke, a címe, b címe *paraméterek*

```
        call    skal        ; eljárás hívás
                            ; eredmény az ax regiszterben
        ADD     SP,6        ; paraméterek ürítése a veremből
...
        ret                 ; visszatérés az Op. rendszerhez
skalar  endp                ; a skalár eljárás vége


        call    skal
```

Hatására a verembe került a **visszatérési cím**


```
skal    proc                ; Közeli (near) eljárás kezdete
        PUSH    BP          ; BP értékét mentenünk kell
        MOV     BP,SP       ; BP ⇐ SP,
                            ; a stack relatív címzéshez
        PUSH    SI          ; mentések
        PUSH    DI
        push    bx
        push    cx
        push    dx
```

A verem tartalma:

n értéke, a címe, b címe	*paraméterek*
visszatérési cím,	
bp, si, di, bx, cx, dx	*mentett regiszterek*

A verem tartalma:

n értéke, a címe, b címe	*paraméterek*
visszatérési cím,	
bp, si, di, bx, cx, dx	*mentett regiszterek*

```
(SS:SP)  dx   PUSH  BP    ; BP értékét mentenünk kell   - 10
  + 2    cx   MOV   BP,SP ; BP ⇐ SP,                    - 8
  + 4    bx                                             - 6
  + 6    di                                             - 4
  + 8    si                                             - 2
  +10    bp  ----------------------------------- (SS:BP)
  +12    visszatérési cím                               + 2
  +14    b címe                                         + 4
  +16    a címe                                         + 6
  +18    n értéke                                       + 8
  ...    korábbi mentések                               ...
```



```
  +10    bp  ----------------------------------- (SS:BP)
  +12    visszatérési cím                               + 2
  +14    b címe                                         + 4
  +16    a címe                                         + 6
  +18    n értéke                                       + 8
  ...    korábbi mentések                               ...

        MOV     SI,6[BP]    ; SI ⇐ az egyik vektor címe
        MOV     DI,4[BP]    ; DI ⇐ a másik vektor címe
        MOV     CX,8[BP]    ; CX ⇐ a dimenzió értéke
        xor     dx,dx       ; az eredmény ideiglenes helye
        jcxz    kesz        ; ugrás a kesz címkére, ha n=0
        xor     bx,bx       ; bx ⇐ 0, indexezéshez
```



```
ism:    mov     al,[si+bx]  ; független a-tól
        imul    byte ptr [di+bx]  ; független b-től
; csak „byte ptr”-ből derül ki, hogy 8 bites a szorzás
        add     dx,ax       ; dx ⇐ részösszeg
        inc     bx          ; bx ⇐ bx+1, az index növelése
        loop    ism         ; ugrás az ism címkére,
                            ; ha kell ismételni
kesz:   mov     ax,dx       ; a skalár szorzat értéke ax-ben
```



```
        pop     dx          ; visszamentések
        pop     cx
        pop     bx
        POP     DI
        POP     SI
        POP     BP
        ret                 ; visszatérés a hívó programhoz
skal    endp                ; a skal eljárás vége
; D
...

        ADD     SP,6        ; paraméterek ürítése a veremből
```

helyett más megoldás:

```
        RET     6           ; visszatérés a hívó programhoz
                            ; verem ürítéssel: ... SP = SP + 6
```

C konvenció

Hogy egy eljárás különböző számú paraméterrel legyen hívható, azt úgy lehet elérni, hogy a paramétereket fordított sorrendben tesszük a verembe, mert ilyenkor a visszatérési cím alatt lesz az első, alatta a második, stb. paraméter, és általában a korábbi paraméterek döntik el, hogy hogyan folytatódik a paramétersor.

```
f(x,y);          push  y
                 push  x
                 call  f
```


Lokális adat terület, rekurzív és re-entrant eljárások

Ha egy eljárás működéséhez lokális adat területre, munkaterületre van szükség, és a működés befejeztével a munkaterület tartalma fölösleges, akkor a munkaterületet célszerűen a veremben alakíthatjuk ki. A munkaterület lefoglalásának ajánlott módja:

```
...   proc    ...
      PUSH    BP        ; BP értékének mentése
      MOV     BP,SP     ; BP ⇐ SP,
                        ; a stack relatív címzéshez
      SUB     SP,n      ; n byte-os munkaterület lefoglalása
      ...               ; további regiszter mentések
```


Lokális adat terület (**NEAR** eljárás esetén)

```
(SS:SP)   lokális adat terület        ...
  + 2                                 ...
  ...                                 ...
  ...                                 – 2
          bp  - - - - - - - - - - - - - - - - - - - - (SS:BP)
          visszatérési cím            + 2
          paraméterek                 ...
          korábbi mentések            ...
```

A munkaterület negatív displacement érték mellett stack relatív címzéssel érhető el. (A veremben átadott paraméterek ugyancsak stack relatív címzéssel, de pozitív displacement érték mellett érhetők el.)

A munkaterület felszabadítása visszatéréskor a

```
...                 ; visszamentések
MOV     SP,BP       ; a munkaterület felszabadítása
POP     BP          ; BP értékének visszamentése
ret     ...         ; visszatérés
```

utasításokkal történhet.

Rekurzív és re-entrant eljárások

Egy eljárás **rekurzív**, ha önmagát hívja közvetlenül, vagy más eljárásokon keresztül.

Egy eljárás **re-entrant**, ha többszöri belépést tesz lehetővé, ami azt jelenti, hogy az eljárás még nem fejeződött be, amikor újra felhívható. A rekurzív eljárással szemben a különbség az, hogy a rekurzív eljárásban „programozott”, hogy mikor történik az eljárás újra hívása, re-entrant eljárás esetén az esetleges újra hívás ideje a véletlentől függ. Ez utóbbi esetben azt, hogy a munkaterületek ne keveredjenek össze, az biztosítja, hogy újabb belépés csak másik processzusból képzelhető el, és minden processzus saját vermet használ.

Rekurzív és re-entrant eljárások

Ha egy eljárásunk készítésekor betartjuk, hogy az eljárás a paramétereit a vermen keresztül kapja, kilépéskor visszaállítja a belépéskori regiszter tartalmakat – az esetleg eredményt tartalmazó regiszterek kivételével –, továbbá a fenti módon kialakított munkaterületet használ, akkor az eljárásunk rekurzív is lehet, és a többszöri belépést is lehetővé teszi (re-entrant).

String kezelő utasítások

Az **s** forrás területet (**DS:SI**),
a **d** cél területet pedig (**ES:DI**) címzi.

A mnemonik végződése (**B** / **W**) vagy az operandus jelzi, hogy bájtos vagy szavas a művelet.

A címzésben résztvevő indexregiszterek értéke
1-gyel módosul bájtos,
2-vel szavas művelet esetén.

Ha a **D** (**Direction**) flag értéke
0, akkor az indexregiszterek értéke növekszik,
1, akkor csökken.

CLD ; D ⇐ 0
STD ; D ⇐ 1

Az alábbi utasítások
– mint általában az adat mozgató utasítások –
érintetlenül hagyják a flag-eket

Átvitel az (**ES:DI**) által mutatott címre
a (**DS:SI**) által mutatott címről:

MOVSB ; MOVe String Byte
MOVSW ; MOVe String Word
MOVS d,s ; MOVe String (byte vagy word)

d és **s** csak azt mondja meg,
hogy bájtos vagy szavas az átvitel!

Betöltés **AL**-be illetve **AX**-be a (**DS:SI**) által mutatott címről (csak **SI** módosul):

LODSB ; LOaD String Byte
LODSW ; LOaD String Word
LODS s ; LOaD String (byte vagy word)

Tárolás az (**ES:DI**) által mutatott címre **AL**-ből illetve **AX**-ből (csak **DI** módosul):

STOSB ; STOre String Byte
STOSW ; STOre String Word
STOS d ; STOre String (byte vagy word)

Az alábbi utasítások beállítják a flag-eket

Az (**ES:DI**) és a (**DS:SI**) által mutatott címen lévő byte illetve szó összehasonlítása, a flag-ek **s – d (!!!)** értékének megfelelően állnak be.

CMPSB ; CoMPare String Byte
CMPSW ; CoMPare String Word
CMPS d,s ; CoMPare String (byte vagy word)

Az (**ES:DI**) által mutatott címen lévő byte (word) összehasonlítása **AL**-lel (**AX**-szel), a flag-ek **AL – d** illetve **AX – d (!!!)** értékének megfelelően állnak be.

SCASB ; SCAn String Byte
SCASW ; SCAn String Word
SCAS d ; SCAn String (byte vagy word)

Ismétlő prefixumok

REP ≡ REPZ ≡ REPE és **REPNZ ≡ REPNE**
A **Z**, **E**, **NZ** és **NE** végződésnek hasonló szerepe van, mint a **LOOP** utasítás esetén.

Ismétlő prefixum használata esetén a string kezelő utasítás **CX**-szer kerül(het) végrehajtásra:
- ha **CX = 0**, akkor egyszer sem (**!!!**),
- különben minden végrehajtást követően **1**-gyel csökken a **CX** regiszter tartalma. Amennyiben **CX** csökkentett értéke **0**, akkor nem történik további ismétlés.

A flag beállító string kezelő utasítás ismétlésének további feltétele, hogy a flag állapota megegyezzen a prefixum végződésében előírttal.

Ismétlő prefixumok

REP ≡ REPZ ≡ REPE és **REPNZ ≡ REPNE**

A program jobb olvashatósága érdekében

- flag-et nem állító utasítások előtt mindig **REP**-et használjunk,
- flag-et beállító utasítás előtt pedig sohase **REP**-et, hanem helyette a vele egyenértékű **REPE**-t vagy **REPZ**-t!

```
; A 100 elemű  array  nevű tömbnek van-e
;      3-tól különböző eleme?
       mov     cx, 100
       mov     di, -1      ; előbb lehessen inc,  mint cmp
       mov     al, 3
NEXT:  inc     di
       cmp     array[di], al ; array di-edik eleme = 3?
       LOOPE   NEXT        ; ugrás NEXT-re,
                           ; ha CX≠0 és a di-edik elem=3
       JNE     NEM3        ; CX = 0  vagy  array[di] ≠ 3
       . . .               ; array ≡ 3
       . . .
NEM3:  . . .               ; di az első 3-tól különböző
       . . .               ; elem indexe
```


Ugyanennek a feladatnak a megoldása string kezelő utasítás segítségével:

```
; A 100 elemű  array  nevű tömbnek van-e
; 3-tól különböző eleme?
        mov     cx,100
        mov     di,offset array
        mov     AL,3
 REPE   SCAS    array       ; array 0., 1., ... eleme = 3?
        JNE     NEM3
        . . .               ; array ≡ 3,
        . . .
NEM3:   DEC     DI          ; DI az első ≠ 3 elemre mutat
        . . .
```


A 100 elemű array nevű tömbnek van-e 3-tól különböző eleme?

```
       mov     cx, 100
       mov     di, -1
       mov     al, 3
NEXT:  inc     di
       cmp     array[di], al
       LOOPE   NEXT

       JNE     NEM3
       . . .
       . . .
NEM3:  . . .
       . . .
```

```
        mov     cx,100
        mov     di,offset array
        mov     AL,3
 REPE   SCAS    array

        JNE     NEM3
        . . .
        . . .
NEM3:   DEC     DI
        . . .
```

Használja **ES**-t. Sokkal gyorsabb

Nem minden eltérés lényeges!

Egyszerűsített lexikális elemző

Feladata, hogy azonosító, szám, speciális jelek és a program vége jel előfordulásakor rendre **A**, **0**, **,** és **.** karaktert írjon a képernyőre. Az esetleges hibákat **?** jelezze.

XLAT utasítás alkalmazásának tervezése:

Karakter típusok	karakterek	kód
Betű	A … Z a … z	2
Számjegy	0 … 9	4
Speciális jel	, . tabulátor ; + - ( ) cr lf	6
Vége jel	$	8
Hibás karakter	a többi	0


```
data    segment     para  public  'data'

; ugró táblák a szintaktikus helyzetnek megfelelően:

; kezdetben, speciális és hibás karakter után
;                              következő karakter
t_s     dw   hiba    ; hibás kar.: ?   → spec. jel szint
        dw   lev_a   ; betű: A    → azonosító szint
        dw   lev_n   ; számjegy: 0    → szám szint
        dw   lev_s   ; spec. jel: ,    → spec. jel szint
        dw   vege    ; $: .    → program vége
```



```
; azonosító szint
t_a     dw   hiba    ; hibás kar.: ?   → spec. jel szint
        dw   ok      ; betű: nincs teendő
        dw   ok      ; számjegy: nincs teendő
        dw   lev_s   ; speciális jel: ,    → spec. jel szint
        dw   vege    ; $: .    → program vége
; szám szint
t_n     dw   hiba    ; hibás kar.: ?   → spec. jel szint
        dw   hiba    ; betű: hiba: ?   → spec. jel szint
        dw   ok      ; számjegy: nincs teendő
        dw   lev_s   ; speciális jel: ,    → spec. jel szint
        dw   vege    ; $: .    → program vége
```

```
level     dw   ?                ; az aktuális ugrótábla címe
c_h       db   0                ; hibás karakter kódja
c_b       db   2                ; betű kódja
c_n       db   4                ; számjegy kódja
c_s       db   6                ; speciális jel kódja
c_v       db   8                ; végjel kódja
specjel   db   ' ,.   ;+-()', 13, 10    ; a speciális jelek
vegjel    db   '$'              ; vége jel, kihasználjuk,
                                ; hogy itt van!
table     db   256 dup (?)      ; átkódoló tábla (256 byte)
text      db   'a,tz.fe&a 21 a12; 12a $' ; elemzendő szöveg
data      ends
```



```
code      segment  para public 'code'
          assume   cs:code, ds:data, es:data, ss:stack
lex       proc     far
          push     ds
          xor      ax,ax
          push     ax          ; visszatérési cím a veremben
          mov      ax,data
          mov      ds,ax
          mov      es,ax       ; assume miatt
          call     prepare     ; átkódoló tábla elkészítése
          mov      si, offset text   ; az elemzendő szöveg
                               ; kezdőcíme
          call     parsing     ; elemzés
          ret                  ; vissza az Op. rendszerhez
lex       endp
```



```
prepare   proc            ; az átkódoló tábla elkészítése
; az eljárás rontja ax, bx, cx, di, si tartalmát
          cld             ; a string műveletek iránya pozitív
          mov     bx, offset table
          mov     di,bx
          mov     al,c_h    ; hibás karakter kódja
          mov     cx,256    ; a tábla hossza
  rep     stos    table; table ⇐ minden karakter hibás
```



```
          mov     al,c_b    ; betű kódja
          mov     di,'A'    ; A ASCII kódja
          add     di,bx     ; A helyének offset címe
          mov     cx,'Z'-'A'+1   ; a nagybetűk száma
                            ; a betűk ASCII kódja folyamatos!
  rep     stosb
          mov     di,'a'    ; a ASCII kódja
          add     di,bx     ; a helyének offset címe
          mov     cx,'z'-'a'+1   ; a kisbetűk száma
  rep     stosb
```



```
          mov     al,c_n    ; számjegy kódja
          mov     di,'0'    ; 0 ASCII kódja
          add     di,bx     ; 0 helyének offset címe
          mov     cx,'9'-'0'+1   ; a számjegyek száma
                ; a számjegyek ASCII kódja folyamatos!
  rep     stosb
```



```
          mov     si,offset specjel; speciális jelek
                                    ; feldolgozása
          xor     ah,ah     ; hogy ax=al legyen
pr1:      lods    specjel   ; speciális jel ASCII kódja
          mov     di,ax     ; ah=0 miatt ax = a jel kódja
          cmp     al,vegjel ; vegjel közvetlenül a
                            ; speciális jelek után van!
          je      pr2       ; ez már a vegjel
          mov     al,c_s    ; speciális karakter kódja
          mov     [bx+di],al; elhelyezés a táblában
          jmp     pr1       ; ciklus vége
```

```
pr2:    mov     al,c_v      ; a végjel kódja
        mov     [bx+di],al  ; elhelyezés a táblában
        ret                 ; vissza a hívó eljáráshoz
prepare endp
```



```
parsing proc            ; elemzés
; az eljárás rontja ax, bx, cx, di, si tartalmát
        cld     ; a string műveletek iránya pozitív
        mov     bx, offset table
        mov     di,offset t_s ; spec. jel szint
lv1:    mov     level,di    ; szint beállítás
        xor     ah,ah       ; hogy ax=al legyen
ok:     lods    text        ; a következő karakter
        xlat                ; al ⇐ 0, 2, 4, 6 vagy 8
        mov     di, level   ; di ⇐ az akt. ugrót. címe
        add     di, ax      ; az ugrótáblán belüli cím
        jmp     [di]        ; kapcsoló utasítás
```



```
hiba:   mov     di,offset t_s    ; hibás karakter,
                                 ; spec. jel szint következik
        mov     al,'?'
lv2:    mov     ah,14    ; BIOS hívás előkészítése
        int     10h      ; BIOS hívás:
                         ; karakter írás a képernyőre
        jmp     lv1

lev_a:  mov     di,offset t_a    ; azonosító kezdődik
        mov     al,'A'
        jmp     lv2
```



```
lev_n:  mov     di,offset t_n    ; szám kezdődik
        mov     al,'0'
        jmp     lv2
lev_s:  mov     di,offset t_s    ; speciális jel
        mov     al,','
        jmp     lv2
vege:   mov     al,'.'           ; szöveg vége
        mov     ah,14    ; BIOS hívás előkészítése
        int     10h      ; BIOS hívás:
                         ; karakter írás a képernyőre
        ret              ; elemzés vége, vissza a hívóhoz
parsing endp
code    ends
```



```
stack   segment  para     stack     'stack'
        dw   100 dup (?)  ; 100 word legyen a verem
stack   ends
        end      lex   ; modul vége, start cím: lex
```


Logikai utasítások

Bitenkénti logikai műveleteket végeznek.

1 az igaz, **0** a hamis logikai érték.

```
AND   op1,op2   ; op1 ⇐ op1 & op2, bitenkénti és
TEST  op1,op2   ; flag-ek op1 & op2 szerint
OR    op1,op2   ; op1 ⇐ op1 | op2, bitenkénti vagy
XOR   op1,op2   ; op1 ⇐ op1 ^ op2 (eXclusive OR),
                ; bitenkénti kizáró vagy
NOT   op        ; op ⇐ ~op, bitenkénti negáció,
                ; nem módosítja STATUS tartalmát!
```

Bit forgató (Rotate) és léptető (Shift) utasítások

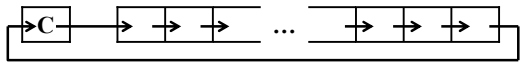
Forgatják (Rotate) illetve léptetik (Shift) **op** tartalmát. A forgatás/léptetés történhet

- 1 bittel,
- vagy byte illetve word esetén a **CL** regiszter alsó 3 illetve 4 bit-jén megadott bittel jobbra (Right) vagy balra (Left).

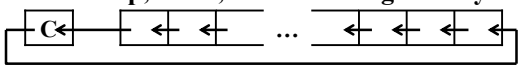
Az utoljára kilépő bit lesz a **Carry** új tartalma.

A rotálás történhet a **Carry**-n keresztül, ilyenkor a belépő bit a **Carry**-ből kapja az értékét:

RCR op,1/CL ; Rotate through Carry Right

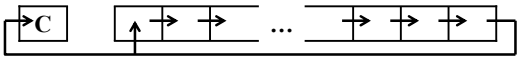


RCL op,1/CL ; Rotate through Carry Left

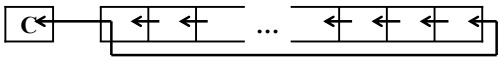


A rotálás történhet úgy, hogy **Carry** csak a kilépő bitet fogadja, a belépő bit értékét a kilépő bit szolgáltatja:

ROR op,1/CL ; ROtate Right

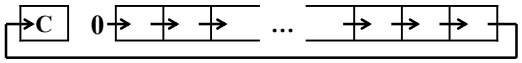


ROL op,1/CL ; ROtate Left



Logikai léptetés jobbra: A belépő bit **0**:

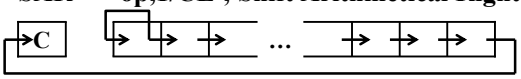
SHR op,1/CL ; SHift Right



Előjel nélküli egész számok **2 hatványával** történő osztására alkalmas.

Aritmetikai léptetés jobbra: A belépő bit **op** előjele:

SAR op,1/CL ; Shift Arithmetical Right



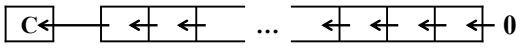
Előjeles egész számok **2 hatványával** történő osztására alkalmas. Negatív számok esetén csal!

Balra léptetéskor a belépő bit mindig **0**:

SHL op,1/CL ; SHift Left

SAL op,1/CL ; Shift Arithmetical Left

SAL ≡ SHL



Előjel nélküli vagy előjeles egész számok **2 hatványával** történő szorzására alkalmas.


```
hexa    proc                ; ax kiírása hexadecimálisan
; legyen a példa kedvéért:  ax = 1234H
        xchg    ah,al       ; ah és al felcserélése
; most:                     ax = 3412H, al = 12H
        call    hexa_b      ; al (az eredeti ah) kiírása
; kiírtuk, hogy 12
        xchg    ah,al       ; ah és al visszacserélése
; most újra:                ax = 1234H, al = 34H
        call    hexa_b      ; al kiírása
; most kiírtuk, hogy 34, tehát eddig kiírtuk, hogy 1234
        ret                 ; visszatérés
hexa    endp                ; a hexa eljárás vége
```

```
hexa_b  proc            ; al kiírása hexadecimálisan
; az első híváskor:     al = 12H
        push    cx      ; mentés a verembe
        mov     cl,4    ; 4 bit-es rotálás előkészítése
        ROR     al,CL   ; az első jegy az alsó 4 biten
; most:                 al = 21H
        call    h_jegy  ; az első jegy kiírása
; kiírtuk, hogy 1
        ROR     al,CL   ; a 2. jegy az alsó 4 biten
; most újra:            al = 12H
        call    h_jegy  ; a második jegy kiírása
; most kiírtuk, hogy 2, tehát eddig kiírtuk, hogy 12
        pop     cx      ; visszamentés a veremből
        ret             ; visszatérés
hexa_b  endp            ; a hexa_b eljárás vége
```



```
h_jegy  proc            ; hexadecimális jegy kiírása
        push    ax      ; mentés a verembe
        AND     al,0FH  ; a felső 4 bit 0 lesz,
                        ; a többi változatlan
        add     al,'0'  ; + 0 kódja
        cmp     al,'9'  ; ≤ 9 ?
        jle     h_jegy1 ; ugrás h_jegy1 -hez, ha igen
        add     al,'A'-0AH-'0' ; A…F hexadecimális
                        ; jegyek kialakítása
h_jegy1: mov    ah,14   ; BIOS hívás:
        int     10H     ; karakter kiírás
        pop     ax      ; visszamentés a veremből
        ret             ; visszatérés
h_jegy  endp            ; a h_jegy eljárás vége
```


Processzor vezérlő utasítások I.

A processzor állapotát módosít(hat)ják.

A **STATUS** bitjeinek módosítása			
Flag	**CL**ear	**S**e**T**	**C**o**M**plement
C	**CLC**	**STC**	**CMC**
D	**CLD**	**STD**	
I	**CLI**	**STI**	

Processzor vezérlő utasítások II.

NOP ; **NO** o**P**eration, üres utasítás,
; nem végez műveletet.

WAIT ; A processzor várakozik, amíg más
; processzortól (pl. lebegőpontos
; segédprocesszortól) kész jelzést nem kap.

HLT ; **HaLT**, leállítja a processzort.
; A processzor külső megszakításig
; várakozik.

Input, output (I/O) utasítások (I8086/88)

A külvilággal történő információ csere **port**-okon (kapukon) keresztül zajlik. A kapu egy memória cím, az információ csere erre a címre történő írással, vagy erről a címről való olvasással történik. Egy-egy cím vagy cím csoport egy-egy perifériához kötődik. A központi egység oldaláról a folyamat egységesen az **IN** (input) és az **OUT** (output) utasítással történik.

Input, output (I/O) utasítások (I8086/88)

A perifériától függ, hogy a hozzá tartozó port 8 vagy 16 bites. A központi egységnek az **AL** illetve **AX** regisztere vesz részt a kommunikációban. A port címzése 8 bites közvetlen adattal vagy a **DX** regiszterrel történik.

```
IN   AL/AX,port
        ; AL/AX ⇐ egy byte/word a port-ról
OUT     port,AL/AX
        ; port ⇐ egy byte/word AL/AX-ből
```

A periféria oldaláról a helyzet nem ilyen egyszerű. Az input információ „**csomag**okban” érkezik, az output információt „csomagolva” kell küldeni. A csomagolás (vezérlő információ) mondja meg, hogy hogyan kell kezelni a csomagba rejtett információt (adatot). Éppen ezért az operációs rendszerek olyan egyszerűen használható eljárásokat tartalmaznak (**BIOS** – Basic Input/Output System – rutinok, stb.), amelyek elvégzik a ki- és becsomagolás munkáját, és ezáltal lényegesen megkönnyítik a külvilággal való kommunikációt.

Megszakítás rendszer, interrupt utasítások

- Az **I/O** utasítás lassú ↔ a **CPU** gyors, a **CPU** várakozni kényszerül
- **I/O** regiszter (**port**): a **port** és a központi egység közötti információ átadás gyors, a periféria autonóm módon elvégzi a feladatát. Újabb perifériához fordulás esetén a **CPU** várakozni kényszerülhet.
- **Pollozásos technika** (~tevékeny várakozás)**:** a futó program időről időre megkérdezi a periféria állapotát, és csak akkor ad ki újabb **I/O** utasítást, amikor a periféria már fogadni tudja. A hatékonyság az éppen futó programtól függ.

Megszakítás

A (program) megszakítás azt jelenti, hogy az éppen futó program végrehajtása átmenetileg megszakad – a processzor állapota megőrződik, hogy a program egy későbbi időpontban folytatódhassék – és a processzor egy másik program, az úgynevezett **megszakítás kezelő** végrehajtását kezdi meg.

Miután a megszakítás kezelő elvégezte munkáját, gondoskodik a processzor megszakításkori állapotának visszaállításáról, és visszaadja a vezérlést a megszakított programnak: **Átlátszóság.**

Pl.: nyomtatás

- Nyomtatás pufferbe, később a tényleges nyomtatást vezérlő program indítása.
- Nyomtatás előkészítése (a nyomtató megnyitása), **HLT**.
- A továbbiak során a nyomtató megszakítást okoz, ha kész újabb adat nyomtatására. Ilyenkor a **HLT** utasítást követő címre adódik a vezérlés. A következő karakter előkészítése nyomtatásra, **HLT**. A bekövetkező megszakítás hatására a megszakító rutin mindig a következő adatot nyomtatja. Ha nincs további nyomtatandó anyag, akkor a nyomtatást vezérlő program lezárja a nyomtatót (nem következik be újabb megszakítás a nyomtató miatt), és befejezi a működését.

A **HLT** utasítás csak akkor szükséges, ha a nyomtatást kérő program befejezte a munkáját. Ellenkező esetben visszakaphatja a vezérlést. Ilyenkor az ő feladata az esetleg szükséges várakozásról gondoskodni a program végén.

Bevitel esetén olyankor is várakozni kell, ha még a beolvasás nem történt meg, és a további futáshoz szükség van ezekre az adatokra.

Jobb megoldás, ha a **HLT** utasítás helyett az operációs rendszer fölfüggeszti a program működését, és elindítja egy másik program futását.

Ez vezetett a **multiprogramozás** kialakulásához.

A megszakítás kezelő (megszakító rutin) megszakítható-e? Gyors periféria kiszolgálása közben megszakítás kérés, …

„Alap” állapot – „megszakítási” állapot, megszakítási állapotban nem lehet újabb megszakítás.

Hierarchia: megszakítási állapotban csak magasabb szintű ok eredményezhet megszakítást.

Bizonyos utasítások csak a központi egység bizonyos kitüntetett állapotában hajthatók végre, alap állapotban nem → csapda, szoftver megszakítás.

Megoldható az operációs rendszer védelme, a tár védelem stb.

A megoldás nem tökéletes: **vírus**.

I8086/88

Megszakítás kiszolgálásakor a **STATUS**, **CS** és **IP** a verembe kerül, az **I** és a **T** flag **0** értéket kap (az úgynevezett maszkolható megszakítások tiltása és folyamatos üzemmód beállítása), majd **(CS:IP)** felveszi a megszakítás kezelő kezdőcímét.

Átlátszóság: Amikor bekövetkezik egy megszakítás, akkor bizonyos utasítások végrehajtódnak, de amikor ennek vége, a **CPU** ugyanolyan állapotba kerül, mint amilyenben a megszakítás bekövetkezése előtt volt.

Megszakítás és csapda

Megszakítás (interrupt): Olyan automatikus eljárás hívás, amit általában nem a futó program, hanem valamilyen **B/K** eszköz idéz elő, pl. a program utasítja a lemezegységet, hogy kezdje el az adatátvitelt, és annak végeztével megszakítást küldjön. **Megszakítás kezelő**.

Csapda (trap): A program által előidézett feltétel (pl. túlcsordulás) hatására automatikus eljárás hívás. **Csapda kezelő**.

A csapda a **programmal szinkronizált**,
a megszakítás nem.

Interrupt (csapda) utasítások

Szoftver megszakítást (csapdát) eredményeznek.

INT i ; 0 ≤ i ≤ 255,
; megszakítás az **i.** ok szerint.

Az **INT 3** utasítás kódja csak egy byte (a többi 2 byte), így különösen alkalmas nyomkövető (**DEBUG**) programokban történő alkalmazásra.

A **DEBUG** program saját magához irányítja a **3**-as megszakítást. Az ellenőrzendő program megadott pontján (törés pont, **break point**) lévő utasítást (annak 1. bájtját) átmenetileg az **INT 3** utasításra cseréli, és átadhatja a vezérlést az ellenőrzendő programnak. Amikor a program az **INT 3** utasításhoz ér, a megszakítás hatására a **DEBUG** kapja meg a vezérlést. Kiírja a regiszterek tartalmát, és további információt kérhetünk a program állapotáról.
Később visszaírja azt a tartalmat, amit **INT 3** -ra cserélt, elhelyezi az újabb törés pontra az **INT 3** utasítást és visszaadja a vezérlést az ellenőrzendő programnak.

Ezek alapján érthetővé válik a **DEBUG** program néhány „furcsasága”:

- Miért „felejti el” a töréspontot? Ha ugyanis nem felejtené el – azaz nem cserélné vissza a töréspontra elhelyezett utasítást az eredeti utasításra – akkor a program nyomkövetésében nem tudna továbblépni.
- Miért nem lehet egy ciklus futásait egyetlen töréspont elhelyezésével figyeltetni, stb?

INTO ; megszakítás csak **O=1 (Overflow)**
; esetén a **4.** ok szerint

Visszatérés a megszakító rutinból

IRET ; IP, CS, STATUS feltöltése a
; veremből

Szemafor

Legyen az **S** szemafor egy olyan word típusú változó, amely mindegyik program számára elérhető. Jelentse **S=0** azt, hogy az erőforrás szabad, és **S≠0** azt, hogy az erőforrás foglalt. Próbáljuk meg a szemafor kezelését!

; 1. kísérlet

```
ujra:    mov     cx,S
         jcxz    szabad
         ...             ; foglalt az erőforrás, várakozás
         jmp     ujra
szabad:  mov     cx,0FFFFh
         mov     S,cx    ; a program lefoglalta az erőforrást
```


; 2. kísérlet

```
ujra:    MOV     CX,0FFFFH
         XCHG    CX,S    ; már foglaltat jelez a
                         ; szemafor!
         JCXZ    szabad  ; ellenőrzés S korábbi tartalma
                         ; szerint.
         ...             ; foglalt az erőforrás,
                         ; várakozás
         jmp     ujra
szabad:  ...             ; szabad volt az erőforrás,
                         ; de a szemafor már foglalt
```


Az **XCHG** utasítás mikroprogram szinten:

Segéd regiszter ⇐ S; S ⇐ CX; CX ⇐ Segéd regiszter

olvasás – módosítás – visszaírás

```
ujra:    mov     cx,0FFFFh
  LOCK   xchg    cx,S    ; S már foglaltat jelez
         jcxz    szabad  ; ellenőrzés S korábbi
                         ; tartalma szerint
         ...             ; foglalt, várakozás
         jmp     ujra
szabad:  ...             ; használható az erőforrás,
                         ; de a szemafor már foglalt
         ...
         MOV     S,0     ; a szemafor szabadra állítása
```


Assembly programozás
Pszeudo utasítások

A pszeudo utasításokat a fordítóprogram hajtja végre. Ez a végrehajtás fordítás közbeni tevékenységet vagy a fordításhoz szükséges információ gyűjtést jelenthet.

Adat definíciós utasítások

Az adatokat általában külön szegmensben szokás és javasolt definiálni iniciálással vagy anélkül.

Az adat definíciós utasítások elé általában azonosítót (változó név) írunk, hogy hivatkozhassunk az illető adatra. Egy-egy adat definíciós utasítással – vesszővel elválasztva – több azonos típusú adatot is definiálhatunk. A kezdőérték – megfelelő típusú – tetszőleges konstans (szám, szöveg, cím, ...) és **`kifejezés`** lehet. Ha nem akarunk kezdőértéket adni, akkor **`?`** -et kell írnunk.

DUP operátor

```
kifejezés DUP (adat)
```


Egyszerű adat definíciós utasítások

Define Byte (**DB**):

```
Adat1   db  25      ; 1 byte, kezdőértéke decimális 25
Adat2   db  25H     ; 1 byte, kezdőértéke hexadec. 25
Adat3   db  1,2     ; 2 byte (nem egy szó!)
Adat4   db  5 dup (?); 5 inicializálatlan byte
Kar     db  'a','b','c'    ; 3 ASCII kódú karakter
Szoveg  db  "Ez egy szöveg",13,0AH
                    ; ACSII kódú szöveg és 2 szám
Szov1   db  'Ez is "szöveg"'
Szov2   db  "és ez is 'szöveg'"
```

Define Word (**DW**):

```
Szo         dw  0742H,452
Szo_címe    dw  Szo      ; Szo offset címe
```

Define Double (**DD**):

```
Szo_f       dd  Szo      ; Szo távoli
                         ; (segment + offset) címe
```

Define Quadword (**DQ**)

Define Ten bytes (**DT**)

Összetett adat definíciós utasítások

Struktúra és a rekord.

Először a **típus**t kell definiálni. A típus **definíció** nem jelent helyfoglalást. A struktúra illetve rekord konkrét példányai struktúra illetve rekord **hívás**sal definiálhatók. A struktúra illetve rekord elemi részeit **mező**knek (**field**) nevezzük.

A hardver nem ismeri ezeket az adat típusokat, a kezelésükről szoftveresen kell gondoskodni!

Struktúra

Struktúra definíció: a struktúra típusát definiálja a későbbi struktúra hívások számára, ezért a memóriában nem jár helyfoglalással.

```
Str_típus  STRUC   ; struktúra (típus) definíció
...                ; mező (field) definíciók:
...                ;     egyszerű adat definíciós
...                ;     utasítások
Str_típus  ENDS    ; struktúra definíció vége
```

A **mező** (**field**) definíció csak egyszerű adat definíciós utasítással történhet, ezért struktúra mező nem lehet másik struktúra vagy rekord.

A mezők definiálásakor megadott értékek kezdőértékül szolgálnak a későbbiekben történő struktúra hívásokhoz. A definícióban megadott kezdőértékek közül azoknak a mezőknek a kezdőértéke híváskor felülbírálható, amelyek csak egyetlen adatot tartalmaznak (ilyen értelemben a szöveg konstans egyetlen adatnak minősül). Pl.:

```
S       STRUC            ; struktúra (típus) definíció
F1      db  1,2          ; híváskor nem lehet felülírni
F2      db  10 dup (?)   ;     nem lehet felülírni
F3      db  5            ;     felülírható
F4      db  'a','b','c'  ;     nem lehet felülírni, de
F5      db  'abc'        ;     felülírható
S       ENDS
```


Struktúra hívás: A struktúra definíciójánál megadott **Str_típus** névnek a műveleti kód részen történő szerepeltetésével hozhatunk létre a definíciónak megfelelő típusú struktúra változókat. A kezdőértékek fölülbírása a kívánt értékek **< >** közötti felsorolásával történik

```
S1    S                ; kezdőértékek a definícióból
S2    S  <,,7,,'FG'>   ; F3 kezdőértéke 7,
                       ; F5-é 'FG '
S3    S  <,,'A'>       ; F3 kezdőértéke 'A' ,
                       ; a többi a definícióból
```

Struktúrából vektort is előállíthatunk, pl.:

```
S_v   S  8 dup (<,,'A'>)
                       ; 8 elemű struktúra vektor
```


Struktúra mezőre hivatkozás: A struktúra változó nevéhez tartozó **OFFSET** cím a struktúra **OFFSET** címét, míg a mező neve a struktúrán belüli címet jelenti. A struktúra adott mezejére úgy hivatkozhatunk, hogy a struktúra és mező név közé **.**-ot írunk, pl.:

```
        MOV     AL,S1.F3
```

A **.** bármely oldalán lehet másfajta cím is, pl.

```
        MOV     BX, OFFSET S1
```

után az alábbi utasítások mind ekvivalensek az előzővel:

```
        MOV     AL,[BX].F3
        MOV     AL,[BX]+F3
        MOV     AL,F3.[BX]
        MOV     AL,F3[BX]
```

A fentiekből az is következik, hogy a mező és struktúra név – ellentétben a magasabb szintű programozási nyelvekkel – szükségképpen **egyedi név**, tehát sem másik struktúra definícióban, sem közönséges változóként nem szerepelhet.

A struktúra vektorokat a hagyományos módon még akkor sem indexezhetjük, ha az index konstans. Pl.

```
MOV     AL,S_v[5].F3
        ; szintaktikusan helyes, de
```

`[5]` nem a vektor ötödik elemére mutató címet fogja eredményezni, csupán `5` byte-tal magasabb címet, mint `S_v.F3`. Ha `i` változó, akkor

```
MOV     AL,S_v[i].F3
        ; szintaktikusan is HIBÁS!
```


Mindkét esetben programmal kell kiszámíttatni az elem offset-jét, pl. ha `i` word:

```
MOV     AX,TYPE S   ; S hossza byte-okban
                    ;     (l. később)
MUL     i           ; Az indexet 0-tól számoljuk!
MOV     BX,AX       ; az adat nem „lóghat ki” a
                    ; szegmensből (DX=0)
MOV     AL,S_v.F3[BX]
        ; AL ⇐ az i-dik elem F3 mezeje.
```


Rekord

Rekord definíció: Csak a rekord típusát definiálja a későbbi rekord hívások számára.

```
Rec_típus  RECORD mező_specifikációk
```

Az egyes mező specifikációkat `,`-vel választjuk el egymástól.

Mező specifikáció:

```
mező_név:szélesség=kezdőérték
```

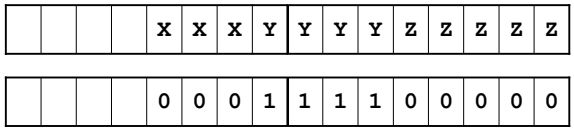
`szélesség` a mező bit-jeinek száma.

Az **`=kezdőérték`** el is maradhat, ha elmarad, az a mező `0`-val való inicializálását írja elő.

Pl.:

```
R   RECORD X:3,Y:4=15,Z:5
```

Az `R` rekord szavas (12 bit), a következőképpen helyezkedik el egy szóban:



Rekord hívás: A rekord definíciójánál megadott névnek a műveleti kód részen történő szerepeltetésével hozhatunk létre a definíciónak megfelelő típusú rekord változókat. A kezdőértékek fölülbírálása a kívánt értékek `< >` közötti felsorolásával történik.

```
R1      R   < >      ; 01E0H, kezdőértékek a
                     ;     definícióból
R2      R   <,,7>    ; 01E7H, X, Y kezdőértéke a
                     ;     definícióból, Z-é 7
R3      R   <1,2>    ; 0240H, X kezdőértéke 1, Y-é 2,
                     ;     Z-é a definícióból
```

Rekordból vektort is előállíthatunk, pl.:

```
R_v     R   5 dup (<1,2,3>) ; 0243H,
                    ; 5 elemű rekord vektor
```


Rekord mezőre hivatkozás

A mező név olyan konstansként használható, amely azt mondja meg, hány bittel kell jobbra léptetnünk a rekordot, hogy a kérdéses mező az 1-es helyértékre kerüljön.

`MASK` és **`NOT MASK`** operátor

```
; AX ⇐ R3 Y mezeje a legalacsonyabb helyértéken
    MOV     AX,R3       ; R3 szavas rekord!
    AND     AX,MASK Y   ; Y mezőhöz tartozó bitek
                        ;     maszkolása
    MOV     CL,Y        ; léptetés előkészítése
    SHR     AX,CL       ; kész vagyunk.
```

`SAR` nem lenne korrekt: nem biztos, hogy az Y mező nem tartalmazza az előjel bitet.

Kifejezés

Egy művelet operandusa lehet konstans, szimbólum vagy kifejezés.

Konstans

A konstans lehet numerikus vagy szöveg konstans.

A numerikus konstansok decimális, hexadecimális, oktális és bináris számrendszerben adhatók meg. A számrendszert a szám végére írt **D**, **H**, **O** illetve **B** betűvel választhatjuk ki.

.RADIX n ; 2 ≤ n ≤ 16 , n decimális

A szöveg konstansokat a **DB** utasításban ″ vagy ′ jelek között adhatjuk meg.

Szimbólum

A szimbólum lehet szimbolikus konstans, változó név vagy címke.

Szimbolikus konstans: Az **=** vagy az **EQU** pszeudo utasítással definiálható. Szimbolikus szöveg konstans csak **EQU**-val definiálható. A szimbolikus konstans a program szövegnek a definíciót követő részében használható, értékét a használat helyét megelőző utolsó definíciója határozza meg.

Ha egy szimbólumot **EQU**-val definiálunk, akkor ezt a szimbólumot a modulban másutt nem definiálhatjuk!


```
S       =       1       ; S értéke 1
N       EQU     14      ; N értéke 14
        MOV     CX,N    ; CX ⇐ 14
ISM:
S       =       S+1 ; S értéke ezután 2, függetlenül
                    ; attól, hogy hányadszor fut a ciklus
        MOV     AX,S    ; AX ⇐ 2
        LOOP    ISM
N       =       5       ; hibás
N       EQU     5       ; hibás
S       =       5       ; helyes
S       EQU     5       ; hibás
```


Szimbolikus konstansként használhatjuk a **$** jelet (helyszámláló), melynek az értéke mindenkor a program adott sorának megfelelő **OFFSET** cím. A helyszámláló értékének módosítására az **ORG** utasítás szolgál, pl.:

```
ORG     $+100H ; 100H byte kihagyása
               ; a memóriában
```


Címke: Leggyakoribb definíciója, hogy valamelyik utasítás előtt a sor első pozíciójától **:** -tal lezárt azonosítót írunk. Az így definiált címke **NEAR** típusú. Címke definícióra további lehetőséget nyújt a **LABEL** és a **PROC** pszeudo utasítás:

```
ALFA:   ...     ; NEAR típusú

BETA    LABEL FAR; FAR típusú
GAMMA:  ...     ; BETA is ezt az utasítást
                ; címkézi, de GAMMA NEAR típusú
```


Az eljárás deklarációt a **PROC** pszeudo utasítással nyitjuk meg. A címke rovatba írt azonosító az eljárás neve és egyben a belépési pontjának címkéje. Az eljárás végén az eljárás végét jelző **ENDP** pszeudo utasítás előtt meg kell ismételnünk ezt az azonosítót, de az ismétlés nem minősül címkének. Az eljárás címkéje aszerint **NEAR** vagy **FAR** típusú, hogy maga az eljárás **NEAR** vagy **FAR**. Pl.:

```
A       PROC            ; NEAR típusú
        ...
B       PROC    NEAR    ; NEAR típusú
        ...
C       PROC    FAR     ; FAR típusú
        ...
```

Címkére vezérlés átadó utasítással hivatkozhatunk, **NEAR** típusúra csak az adott szegmensből, **FAR** típusúra más szegmensekből is.

Változó: Definíciója adat definíciós utasításokkal történik. Néha (adat) címkének is nevezik.

Kifejezés

A kifejezés szimbólumokból és konstansokból épül fel az alább ismertetendő műveletek segítségével. Kifejezés az operátorok, pszeudo operátorok operandus részére írható.

Értékét a fordítóprogram határozza meg, és a kiszámított értéket alkalmazza operandusként. Szimbólumok értékén konstansok esetében természetesen a konstans értékét, címkék, változók esetében a hozzájuk tartozó címet – és nem a tartalmat – értjük.

Kifejezés

A kifejezés értéke nemcsak számérték lehet, hanem minden, ami az utasításokban megengedett címzési módok valamelyikének megfelel. Pl. **[BX]** is kifejezés és értéke a **BX** regiszterrel történő indirekt hivatkozás, és ehhez természetesen a fordító programnak nem kell ismernie **BX** értékét.

Természetesen előfordulhat, hogy egy kifejezés egyik szintaktikus helyzetben megengedett, a másikban nem, pl.:

```
mov     ax,[BX] ; [BX] megengedett
mul     [BX]    ; [BX] hibás, de
mul     WORD PTR [BX] ; megengedett
```

Egy kifejezés akkor megengedett, ha az értéke **fordítási időben meghatározható** és az adott szintaktikus helyzetben alkalmazható, pl. az adott utasítás lehetséges címzési módja megengedi. A megengedett kifejezés értékeket az egyes utasítások ismertetése során megadtuk.

A műveletek csökkenő precedencia szerinti sorrendben:

1. **( )** és **[ ]** (zárójelek) továbbá **< >**: míg a **( )** zárójel pár a kifejezés kiértékelésében csupán a műveletek sorrendjét befolyásolja, addig a **[ ]** az indirekció előírására is szolgál. Ha a **[ ]** -en belüli kifejezésre nem alkalmazható indirekció, akkor a **( )** -lel egyenértékű;
   - **LENGTH változó**: a **változó**-hoz tartozó adat terület elemeinek száma;
   - **SIZE változó**: a **változó**-hoz tartozó adat terület hossza byte-okban;
   - **WIDTH R/F**: az **R** rekord vagy az **F** (rekord) mező szélessége bitekben;
   - **MASK F**: az **F** (rekord) mező bitjein **1**, másutt **0**;

Pl.:

```
v       dw      20 dup (?)
rec     record  x:3,y:4
table   dw      10 dup (1,3 dup (?))
str     db      "12345"
```

esetén:

```
        mov     ax,LENGTH v       ; ax ⇐ 20
        mov     ax,LENGTH rec     ; ax ⇐ 1
        mov     ax,LENGTH table   ; ax ⇐ 10
                          ; a belső DUP ignorálva!
        mov     ax,LENGTH str     ; ax ⇐ 1
                          ; str egy elem
```

```
v       dw      20 dup (?)
rec     record  x:3,y:4
table   dw      10 dup (1,3 dup (?))
str     db      "12345"
```

esetén:

```
        mov     ax,SIZE v           ; ax ⇐ 40
        mov     ax,SIZE rec         ; ax ⇐ 1
        mov     ax,SIZE table       ; ax ⇐ 20
                                    ; a belső DUP ignorálva
        mov     ax,SIZE str         ; ax ⇐ 1
                                    ; str bájtos
```



```
v       dw      20 dup (?)
rec     record  x:3,y:4
table   dw      10 dup (1,3 dup (?))
str     db      "12345"
```

esetén:

```
        mov     ax,WIDTH rec    ; ax ⇐ 7
        mov     ax,WIDTH x      ; ax ⇐ 3
        mov     ax,MASK x       ; ax ⇐ 70H
```


2. **.** (pont): struktúra mezőre hivatkozásnál használatos;

3. **:** mező szélesség (rekord definícióban) és explicit szegmens megadás (segment override prefix). Az explicit szegmens megadás az automatikus szegmens regiszter helyett más szegmens regiszter használatát írja elő, pl.:

```
mov     ax, ES:[BX]
                ; ax ⇐ (ES:BX) címen lévő szó
```

Nem írható felül az automatikus szegmens regiszter az alábbi esetekben:

- **CS** program memória címzésnél,
- **SS** stack referens utasításokban (**PUSH**, **POP**, **...**),
- **ES** string kezelő utasításban **DI** mellett, de az **SI**-hez tartozó **DS** átírható.

4.

- **típus PTR cím:** (típus átdefiniálás) ahol **típus** lehet **BYTE**, **WORD**, **DWORD**, **QWORD**, **TBYTE**, illetve **NEAR**, **FAR** (előre hivatkozás esetén fontos) és **PROC**. Pl.:

```
        MUL     BYTE PTR [BX]  ; a [BX] címet
                               ; byte-osan kell kezelni
```

- **OFFSET kifejezés:** a **kifejezés OFFSET** címe (a szegmens kezdetétől számított távolsága byte-okban);
- **SEG kifejezés:** a **kifejezés** szegmens címe (abban az értelemben, ahogy a szegmens regiszterben szokásos tárolni, tehát valós üzemmódban a szegmens tényleges kezdőcímének **16**-oda);

- **TYPE változó:** az elemek hossza byte-okban, ha változó, de

```
TYPE string = 1,
TYPE konstans = 0,
TYPE NEAR címke = -1,
TYPE FAR címke = -2

JMP    (TYPE cím) PTR [BX]
    ; NEAR vagy FAR ugrás attól függően, hogy cím
    ; közeli vagy távoli címke
```

- **... THIS típus:** a program szöveg adott pontján adott típusú szimbólum létrehozása;

Pl.:

```
ADATB       EQU     THIS BYTE
            ; BYTE típusú változó, helyfoglalás nélkül
ADATW       dw      1234H
            ; ez az adat ADATB-vel byte-osan érhető el
. . .
  mov   al,BYTE PTR ADATW ; al ⇐ 34H, helyes
  mov   al,ADATB          ; al ⇐ 34H, helyes
  mov   ah,ADATB+1        ; ah ⇐ 12H, helyes
```

Emlékeztetünk arra, hogy szavak tárolásakor az alacsonyabb helyértékű byte kerül az alacsonyabb címre!

5.
- **LOW kifejezés:** egy szó alsó (alacsonyabb helyértékű) byte-ja;
- **HIGH kifejezés:** egy szó felső (magasabb helyértékű) byte-ja;

Pl.:

```
ADATW   dw      1234H
        mov     al,LOW ADATW   ; al ⇐ 34H
        mov     ah,HIGH ADATW  ; ah ⇐ 12H
```

6. Előjelek:
- **+ :** pozitív előjel;
- **– :** negatív előjel;

7. Multiplikatív műveletek:
- *** :** szorzás;
- **/ :** osztás;
- **MOD:** (modulo) a legkisebb nem negatív maradék, pl.:
  ```
  mov     al,20 MOD 16   ; al ⇐ 4
  ```
- **kifejezés SHL lépés:**
  kifejezés léptetése balra **lépés** bittel;
- **kifejezés SHR lépés:**
  kifejezés léptetése jobbra **lépés** bittel;

  lépés is lehet kifejezés!

A kifejezésben előforduló műveleti jelek (**SHL**, **SHR**, és a később előforduló **NOT**, **AND**, **OR**, és **XOR**) nem tévesztendők össze a velük azonos alakú műveleti kódokkal: az előbbiket a fordító program, az utóbbikat a futó program hajtja végre!

8. Additív műveletek:
- **+ :** összeadás;
- **– :** kivonás;

9. Relációs operátorok (igaz=-1, hamis=0): általában feltételes fordítással kapcsolatban fordulnak elő
- **EQ : =**
- **NE : ≠**
- **LT : <**
- **LE : ≤**
- **GT : >**
- **GE : ≥**

```
// -1 EQ 0FFFFFFFFH   igaz
// -1 NE 0FFFFFFFFH   hamis
33 bites argumentumok!
1 GT -1            igaz
1 GT 0FFFFFFFFH    hamis
```


10. **NOT :** bitenkénti negálás;

11. **AND :** bitenkénti és művelet;

12. Bitenkénti vagy és kizáró vagy művelet:
- **OR :** bitenkénti vagy művelet;
- **XOR :** bitenkénti kizáró vagy művelet;

Feltételes fordítás

A fordító programok általában – így az assembler is – feltételes fordítási lehetőséget biztosít. Ez azt jelenti, hogy a program bizonyos részeit csak abban az esetben fordítja le, ha – a fordítóprogram számára ellenőrizhető – feltétel igaz illetve hamis.

```
IFxx        feltétel
...         ; lefordul, ha a feltétel igaz
ELSE        ; el is maradhat
...         ; lefordul, ha a feltétel hamis
ENDIF
```



```
IF      kifejezés  ; igaz, ha
                   ; kifejezés≠0
IFE     kifejezés  ; igaz, ha
                   ; kifejezés=0
```

Pl.:

```
IF      debug GT 20
        call   debug1
ELSE
        call   debug2
ENDIF
```

```
IF1                 ; igaz a fordítás
                    ; első menetében
IF2                 ; igaz a fordítás
                    ; második menetében

IFDEF  Szimbólum    ; igaz, ha Szimbólum
                    ; definiált
IFNDEF Szimbólum    ; igaz, ha Szimbólum
                    ; nem definiált
```

Pl. Csak akkor definiáljuk **buff**-t, ha a hossza ismert:

```
IFDEF  buff_len
buff   db  buff_len dup (?)
ENDIF
```



```
IFB    <arg>          ; igaz, ha
                      ; arg üres (blank)
IFNB   <arg>          ; igaz, ha
                      ; arg nem üres

IFIDN  <arg1>,<arg2>  ; igaz, ha
                      ; arg1≡arg2 teljesül
IFDIF  <arg1>,<arg2>  ; igaz, ha
              ; arg1≡arg2 nem teljesül
```


Makró és blokk ismétlés

Makró definíció:

```
M_név  MACRO  [fpar1[,fpar2...]]
                      ; makró fej (kezdet)
...                   ; makró törzs
       ENDM           ; makró vége
```

fpar1,fpar2... formális paraméterek vagy egyszerűen paraméterek.

A makró definíció nem lesz része a lefordított programnak, csupán azt határozza meg, hogy később mit kell a makró hívás helyére beírni (makró kifejtés, helyettesítés).

A makró törzsön belül előfordulhat makró hívás és másik makró definíció is.

Makró hívás:

M_név [apar1[,apar2...]]

apar1,apar2... aktuális paraméterek/argumentumok.

A műveleti kód helyére írt **M_név** hatására a korábban megadott definíció szerint megtörténik a makró helyettesítés, más néven makró kifejtés. Ez a makró törzs bemásolását jelenti, miközben az összes paraméter összes előfordulása a megfelelő argumentummal helyettesítődik. A helyettesítés szövegesen történik, azaz minden paraméter – mint szöveg – helyére a megfelelő argumentum – mint szöveg – kerül.

A helyettesítés nem rekurzív.

Makró hívás argumentuma nem lehet makró hívás.

Az argumentumnak megfelelő formális paraméternek lehet olyan előfordulása, amely a későbbiek során makró hívást eredményez.

Dupla szavas összeadás: **(DX:AX)⇐(DX:AX)+(CX:BX)**

Eljárás deklaráció:

```
EDADD   PROC    NEAR
        ADD     AX,BX
        ADC     DX,CX
        RET
EDADD   ENDP
```

Makró definíció:

```
MDADD   MACRO
        ADD     AX,BX
        ADC     DX,CX
        ENDM
```


Ha a programban valahol dupla szavas összeadást kell végezzünk, akkor hívnunk kell az eljárást illetve a makrót:

Eljárás hívás:

```
CALL    EDADD
```

Makró hívás:

```
MDADD
```

Futás közben felhívásra kerül az **EDADD** eljárás

Fordítás közben megtörténik a makró helyettesítés:

```
ADD     AX,BX
ADC     DX,CX
```

Futás közben ez a két utasítás kerül csak végrehajtásra.

Látható, hogy eljárás esetén kettővel több utasítást kell végrehajtanunk, mint makró esetében (**CALL EDADD** és **RET**).

Még nagyobb különbséget tapasztalunk, ha **(CX:BX)** helyett paraméterként kívánjuk megadni az egyik összeadandót:

Eljárás deklaráció:

```
EDADD2 PROC   NEAR
    PUSH    BP
    MOV     BP,SP
    ADD     AX,4[BP]
    ADC     DX,6[BP]
    POP     BP
    RET     4
EDADD  ENDP
```

Eljárás hívás:

Ha **SI** az összeadandónk címét tartalmazza, akkor a felhívás:

```
    PUSH    2[SI]
    PUSH    [SI]
    CALL    EDADD2
```

Futás közben végrehajtásra kerül a paraméter átadás, az eljárás hívás, az eljárás: összesen 9 utasítás

Makró definíció:

```
MDADD2  MACRO   P
IFB     <P>
        ADD     AX,BX
        ADC     DX,CX
ELSE
        ADD     AX,P
        ADC     DX,P+2
ENDIF
        ENDM
```

Makró hívás:

```
    MDADD2  [SI]
```

Fordítás közben a hívás az

```
    ADD     AX,[SI]
    ADC     DX,[SI]+2
```

utasításokra cserélődik, futás közben csak ez a két utasítás kerül végrehajtásra.

```
    MDADD2
```

hatása:

```
        ADD     AX,BX
        ADC     DX,CX
```

Most sem része a makró definíció a lefordított programnak.

A fenti példában rövid volt az eljárás törzs, és ehhez képest viszonylag hosszú volt a paraméter átadás és átvétel. Ilyenkor célszerű a makró alkalmazása.

De ha a program sok helyéről kell meghívnunk egy hosszabb végrehajtandó programrészt, akkor általában célszerűbb eljárást alkalmazni.

Paraméter másutt is előfordulhat a makró törzsben, nemcsak az operandus részen, pl.:

```
PL      macro p1,P2
        mov   ax,p1
        P2    p1
        endm

        PL    Adat, INC
```

hatása:

```
        mov   ax,Adat
        INC   Adat
```


A **&**, **%**, **!** karakterek továbbá a **<>** és **;;** speciális szerepet töltenek be makró kifejtéskor.

& (helyettesítés operátor):

- ha a paraméter – helyettesített – értéke része egy szónak;
- idézeten belüli helyettesítés:

```
errgen      macro  y, x
err&y       db     'Error &y: &x'
            endm

            errgen 5, <Unreadable disk>
```

hatása:

```
err5        db  'Error 5: Unreadable disk'
```

<> (literál szöveg operátor): Ha aktuális paraméter szóközt vagy **,**-t is tartalmaz. Az előző példa **<>** nélkül:

```
        errgen 5, Unreadable disk
```

kifejtve:

```
err5    db      'Error 5: Unreadable'

adat    macro   p
        db      p
        endm

        adat    <'abc',13,10,0>
        adat    'abc',13,10,0
```

kifejtve:

```
        db      'abc',13,10,0
        db      'abc'
```


! (literál karakter operátor): Az utána következő karaktert makró kifejtéskor közönséges karakterként kell kezelni. Pl.: a korábbi **errgen** makró

```
        errgen 103, <Expression !> 255>
```

hívásának hatása:

```
err103 db  'Error 103: Expression > 255'
```

de

```
        errgen 103, <Expression > 255>
```

hívásának hatása:

```
err103 db  'Error 103: Expression '
```


% (kifejezés operátor): Az utána lévő argumentum (kifejezés is lehet) értéke – és nem a szövege – lesz az aktuális paraméter. Pl.:

```
sym1    equ     100
sym2    equ     200
txt     equ     'Ez egy szöveg'

kif     macro   exp, val
        db      "&exp = &val"
        endm

        kif     <sym1+sym2>, %(sym1+sym2)
        kif     txt, %txt

        db      "sym1+sym2 = 300"
        db      "txt = 'Ez egy szöveg'"
```


Az alábbi példa a **%** használatán kívül a makró törzsön belüli makró hívást is bemutatja:

```
s       =       0
ErrMsg  MACRO   text
s       =       s+1
        Msg     %s,text
        ENDM

Msg     MACRO   sz,str
msg&sz  db      str
        ENDM
```



```
s       =       0
ErrMsg  MACRO   text
s       =       s+1
        Msg     %s,text
        ENDM

Msg     MACRO   sz,str
msg&sz  db      str
        ENDM
```

```
        ErrMsg 'syntax error'
```

makró hívás hatására bemásolásra kerül (**.LALL** hatására látszik a listán) az

```
s       =       s+1
        Msg     %s,'syntax error'
```

szöveg. **s** értéke itt **1**-re változik. Újabb makró hívás (**Msg**). A **%s** paraméter az **s** értékére (**1**) cserélődik, majd kifejtésre kerül ez a makró is, ebből kialakul:

```
msg1    db      'syntax error'
```



```
s       =       0
ErrMsg  MACRO   text
s       =       s+1
        Msg     %s,text
        ENDM

Msg     MACRO   sz,str
msg&sz  db      str
        ENDM
```

Egy újabb hívás és hatása:

```
        ErrMsg 'invalid operand'
msg2    db      'invalid operand'
```

`;;` (makró kommentár): A makró definíció megjegyzéseinek kezdetét jelzi. A `;;` utáni megjegyzés a makró kifejtés listájában nem jelenik meg.

`LOCAL c1[,c2...]`

`c1`, **`c2`**, **`...`** minden makró híváskor más, **`??xxxx`** alakú szimbólumra cserélődik, ahol **`xxxx`** a makró generátor által meghatározott hexadecimális szám. A **`LOCAL`** operátort közvetlenül a makró fej utáni sorba kell írni.

```
KOPOG   macro   n
        LOCAL   ujra
        mov     cx,n
ujra:   KOPP
        loop    ujra
        endm
```

Ha a programban többször hívnánk a **`KOPOG`** makrót, akkor a **`LOCAL`** operátor nélkül az **`ujra`** címke többször lenne definiálva.

Makró definíció belsejében lehet másik makró definíció is. A belső makró definíció csak a külső makró meghívása után jut érvényre, válik láthatóvá. Pl.:

```
shifts      macro       OPNAME ; makrót
                            ; definiáló makró
OPNAME&S    MACRO       OPERANDUS,N
            mov         cl, N
            OPNAME      OPERANDUS,cl
            ENDM
            endm
```



```
shifts      macro       OPNAME ; makrót
                            ; definiáló makró
OPNAME&S    MACRO       OPERANDUS,N
            mov         cl, N
            OPNAME      OPERANDUS,cl
            ENDM
            endm
```

Ha ezt a makrót felhívjuk pl.:

```
            shifts ROR
```

akkor a

```
RORS        MACRO   OPERANDUS,N
            mov     cl, N
            ROR     OPERANDUS,cl
            ENDM
```

makró definíció generálódik.


```
RORS        MACRO   OPERANDUS,N
            mov     cl, N
            ROR     OPERANDUS,cl
            ENDM
```

Mostantól meghívható a **`RORS`** makró is, pl.:

```
            RORS    AX, 5
```

aminek a hatása:

```
            mov     cl, 5
            ROR     AX,cl
```


Makró definíció belsejében meghívható az éppen definiálás alatt lévő makró is (a makró hívás ezáltal rekurzívvá válik).

```
PUSHALL macro   reg1,reg2,reg3,reg4,reg5
IFNB    <reg1>          ;; ha a paraméter nem üres
        push    reg1    ;; az első regiszter mentése
        PUSHALL reg2,reg3,reg4,reg5  ;; rekurzió
ENDIF
        ENDM
```

Most pl. a

```
        PUSHALL ax, bx, cx
```

makró hívás hatása:

```
        push    ax
        push    bx
        push    cx
```

```
PUSHALL  macro    reg1,reg2,reg3,reg4,reg5
IFNB     <reg1>           ;; ha a paraméter nem üres
         push     reg1    ;; az első regiszter mentése
         PUSHALL  reg2,reg3,reg4,reg5  ;; rekurzió
ENDIF
         ENDM

         PUSHALL  ax, bx, cx
```

makró hívás hatása:

```
         push     ax
         PUSHALL  bx, cx
```

az újabb hívás hatása:

```
         push     bx
         PUSHALL  cx
```

az újabb hívás hatása:

```
         push     cx
         PUSHALL
```

ennek hatására nem generálódik semmi.


```
FL_CALLELJ     =   0
CALLELJ    macro ;; Eljárást beépítő és felhívó makró
         LOCAL   FIRST    ;; nem lenne fontos
IF       FL_CALLELJ       ;; a 2. hívástól igaz
         call    Elj      ;; elég felhívni az eljárást
         EXITM            ;; makró helyettesítés vége
ENDIF
FL_CALLELJ     =   1     ;; csak az első híváskor
         JMP     FIRST    ;; jut érvényre
Elj      proc             ;; eljárás deklaráció
         ...
         ret
Elj      endp
FIRST:   call    Elj      ;; az eljárás felhívása
         endm
```


Az első **CALLELJ** hívás hatására az

```
FL_CALLELJ =   1
               JMP         ??0000
Elj            proc
               ...
               ret
Elj            endp
??0000:        call        Elj
```

utasítások generálódnak (**??0000** a **FIRST**-ből keletkezett).

A további **CALLELJ** hívások esetén csak egyetlen utasítás, a

```
call      Elj
```

utasítás generálódik.

A megoldás előnye, hogy az eljárás akkor és csak akkor része a programnak, ha a program tartalmazza az eljárás felhívását is, és mégsem kell törődjünk azzal, hogy hozzá kell-e szerkesztenünk a programhoz vagy se.

Megváltoztathatunk egy makró definíciót azáltal, hogy újra definiáljuk.

Makró definíción belül előfordulhat másik makró definíció.

E két lehetőség kombinációjából adódik, hogy a makró definíción belül megadhatunk ugyanarra a makró névre egy másik definíciót, ezáltal készíthető olyan makró, amely „átdefiniálja” önmagát.

Az önmagát átdefiniáló makrók esetében a belső és külső definíciót lezáró **ENDM** utasítások között egyetlen utasítás sem szerepelhet – még kommentár sem!

Önmagát „átdefiniáló” makró (az előző feladat másik megoldása):

```
CALLELJ2   macro             ; külső makró definíció
           jmp     FIRST
Elj2       proc              ; eljárás deklaráció
           ...
           ret
Elj2       endp
FIRST:     call    Elj2      ; eljárás hívás
CALLELJ2   MACRO             ; belső makró definíció
           call    Elj2      ; eljárás hívás
           ENDM              ; belső makró definíció vége
           endm              ; külső makró definíció vége
```

CALLELJ2 első hívásakor a kifejtés eredménye:

```
            jmp     FIRST
Elj2        proc            ; eljárás deklaráció
            ...
            ret
Elj2        endp
FIRST:      call    Elj2    ; eljárás hívás
CALLELJ2    MACRO           ; belső makró definíció
            call    Elj2    ; eljárás hívás
            ENDM        ; belső makró definíció vége
```


A kifejtés **CALLELJ2** újabb definícióját tartalmazza, ez felülírja az eredeti definíciót, és a továbbiak során ez a definíció érvényes. Ez alapján a későbbi **CALLELJ2** hívások esetén

```
call    Elj2
```

a kifejtés eredménye.

Megjegyezzük, hogy most is szerencsésebb lett volna a **FIRST** címkét lokálissá tenni. Igaz, hogy csak egyszer generálódik, de így a **CALLELJ2** makró használójának tudnia kell, hogy a **FIRST** címke már „foglalt”!

Ha egy **M_név** makró definíciójára nincs szükség a továbbiak során, akkor a

```
PURGE    M_név
```

pszeudo utasítással kitörölhetjük.

Blokk ismétlés

Nemcsak a blokk definíciójának kezdetét jelölik ki, hanem a kifejtést (hívást) is előírják. A program más részéről nem is hívhatók.

Blokk ismétlés **kifejezés**-szer:

```
REPT    kifejezés
...     ; ez a rész ismétlődik
ENDM
```


A korábban ismertetett kopogást így is megoldhattuk volna:

A korábbi megoldás:

```
KOPOG   macro   n
        LOCAL   ujra
        mov     cx,n
ujra:   KOPP
        loop    ujra
        endm
```

```
        REPT    N
        KOPP
        ENDM
```

Ha pl. **N=3**, akkor ennek a hatására a

```
        KOPP
        KOPP
        KOPP
```

makró hívások generálódnak.

Megjegyzés: Most **N** nem lehet változó – fordítási időben ismert kell legyen az értéke!

Blokk ismétlés argumentum lista szerint:

```
IRP     par, <arg1[,arg2...]>
... ;   ez a rész többször bemásolásra
... ;   kerül úgy, hogy par rendre
... ;   fölveszi az arg1,arg2... értéket
ENDM

IRP     x, <1,2,3>
db      x
ENDM

db      1
db      2
db      3
```

Blokk ismétlés string alapján:

```
IRPC    par,string
...     ; ez a rész kerül többször bemásolásra úgy,
...     ; hogy par rendre fölveszi
...     ; a string karaktereit
ENDM
```

Ezt a **string**-et nem kell idézőjelek közé tenni (újabb ismétlés jelentene). Ha a **string**-en belül pl. szóköz vagy , is előfordul, akkor **<>** jelek közé kell tenni.

Az előző feladatot így is megoldhattuk volna:

```
IRPC    x,123
db      x
ENDM
```


Másik példa:

```
IRPC    x,ABCDEFGHIJKLMNOPQRSTUVWXYZ
db      '&x'        ;; nagy betűk
db      '&x'+20h    ;; kis betűk
ENDM
```

Hatása:

```
db      'A'
db      'A'+20h             a kódja
. . .
db      'Z'
db      'Z'+20h             z kódja
```

Fontos az **&** jel, nélküle **'x'**-ben **x** nem paraméter, hanem string lenne!

Makró definíció tartalmazhat blokk ismétlést, és blokk ismétlés is tartalmazhat makró definíciót vagy makró hívást.

Pl.: A bit léptető és forgató utasítás kiterjesztésnek egy újabb megoldása:

```
; makrót definiáló blokkismétlés
        IRP     OP,<RCR,RCL,ROR,ROL,SAR,SAL>
OP&S    MACRO   OPERANDUS,N
        mov     cl, N
        OP      OPERANDUS,cl
        ENDM
        ENDM
```

Ennek a megoldásnak előnye, hogy nem kell külön meghívnunk a külső makrót az egyes utasításokkal, mert ezt elvégzi helyettünk az **IRP** blokk ismétlés.


```
; makrót definiáló blokkismétlés
        IRP     OP,<RCR,RCL,ROR,ROL,SAR,SAL>
OP&S    MACRO   OPERANDUS,N
        mov     cl, N
        OP      OPERANDUS,cl
        ENDM
        ENDM
```

hatása:

```
RCRS    MACRO   OPERANDUS,N
        mov     cl, N
        RCR     OPERANDUS,cl
        ENDM
RCLS    MACRO   OPERANDUS,N
        ...
```


Szegmens, szegmens csoport

sz_név **SEGMENT** *aling_type combine_type 'osztály'*

. . . szegmens

sz_név **ENDS**

sz_név a szegmens (szelet) neve.

A fordító az azonos nevű szegmens szeleteket úgy tekinti, mintha folyamatosan, egyetlen szegmens szeletbe írtuk volna.

Az azonos nevű szegmens szeletek paraméterei egy modulon belül nem változhatnak.

A szerkesztő egy memória szegmensbe szerkeszti az azonos nevű szegmenseket.

aling_type (illesztés típusa): a szerkesztőnek szóló információ. Azt mondja meg, hogy a szegmens szelet milyen címen kezdődjön:

BYTE 1-gyel,

WORD 2-vel,

DWORD 4-gyel,

PARA 16-tal,

PAGE 256-tal osztható címen.

Akkor van jelentősége, ha a szegmens szelet egy másik modulban lévő ugyanilyen nevű szegmens szelet folytatása.

combine_type (kombinációs típus): a szerkesztőnek szóló üzenet. Lehet:

PUBLIC: (alapértelmezés) az azonos nevű szegmens szeletek egymás folytatásaként szerkesztendők.

COMMON: az azonos nevű szegmens szeletek azonos címre szerkesztendők. Az így keletkező terület hossza megegyezik a leghosszabb ilyen szegmens szelet hosszával. A **COMMON** hatása csak különböző modulokban megírt szegmens szeletekre érvényesül.

MEMORY: a szerkesztő ezt a szegmenst az összes többi szegmens fölé fogja szerkeszteni, mindig a program legmagasabb címre kerülő része (a Microsoft **LINK** programja ezt nem támogatja).

STACK: a stack részeként szerkesztendő a szegmens szelet, egyebekben megegyezik a **PUBLIC**-kal. Amennyiben van **STACK** kombinációs típusú szegmens a programban, akkor **SS** és **SP** úgy inicializálódik, hogy **SS** az utolsó **STACK** kombinációs típusú szegmensre mutat, **SP** értéke pedig ennek a szegmensnek a hossza.

AT kif: a **kif** sorszámú paragrafusra kerül a szegmens szelet. Alkalmas lehet pl. a port-okhoz kapcsolódó memória címek szimbolikus definiálására.

A szegmens osztály legtöbbször **CODE**, **DATA**, **CONSTANT**, **STACK**, **MEMORY**.

Az **ASSUME** utasítás az assembler-t informálja arról, hogy a címzésekhez a szegmens regisztereket milyen tartalommal használhatja, más szóval, hogy melyik szegmens regiszter melyik szegmensnek a szegmens címét tartalmazza (melyik szegmensre mutat):

ASSUME sz_reg1:sz_név1[, sz_reg2:sz_név2 ...]

ASSUME sz_reg1:sz_név1[, sz_reg2:sz_név2 ...]

Az **ASSUME** utasításban felsorolt szegmenseket „aktív”-nak nevezzük.

Az **ASSUME** utasítás nem gondoskodik a szegmens regiszterek megfelelő tartalommal történő feltöltéséről! Ez a programozó feladata!

Az **ASSUME** utasítás hatása egy-egy szegmens regiszterre vonatkozóan mindaddig érvényes, amíg egy másik **ASSUME** utasítással mást nem mondunk az illető regiszterről.

A **GROUP** utasítással csoportosíthatjuk a szegmenseinket:

```
G_nev   GROUP S_név1[, S_név2...]
```

Az egy csoportba sorolt szegmenseket a szerkesztő a memória egy szegmensébe helyezi. Ha ilyenkor az **ASSUME** utasításban a csoport nevét adjuk meg, és ennek megfelelően állítjuk be a bázis regisztert, akkor a csoport minden szegmensének minden elemére tudunk hivatkozni. Ilyenkor egy változó **OFFSET**-je és effektív címe (**EA**) nem feltétlenül egyezik meg.


```
GRP     GROUP       ADAT1,ADAT2
ADAT1   SEGMENT     para public 'data'
A       dw          1111h
...
ADAT1   ENDS

ADAT2   SEGMENT     para public 'data'
W       dw          2222h
...
ADAT2   ENDS

code  segment     para public 'code'
      ASSUME      CS:code, DS:GRP
      ASSUME      SS:stack, ES:nothing
...
```

```
GRP   GROUP  ADAT1,ADAT2
ADAT1 SEGMENT ...
A     dw     1111h
...
ADAT1 ENDS
ADAT2 SEGMENT ...
W     dw     2222h
...
ADAT2 ENDS
```

```
code  segment ...
      ASSUME CS:code, DS:GRP
      ASSUME SS:stack, ...
      ...
```

```
      MOV SI,OFFSET W ; SI ⇐ W offset-je: 0
; az ADAT2 szegmens elejétől mért távolság
      MOV     AX,[SI]  ; AX ⇐ 1111h,
; de!!!
      LEA     SI, W    ; SI ⇐ effektív címe:
; a GRP szegmens csoport elejétől mért távolság
      MOV     AX,[SI]  ; AX ⇐ 2222h.
```


Globális szimbólumok

A több modulból is elérhető szimbólumok.

A globális szimbólumok teszik lehetővé, hogy a programjainkat modulokra bontva készítsük el. Az egyes modulok közötti kapcsolatot a globális szimbólumok jelentik.

Globális szimbólumok

Ha egy szimbólumot globálissá kívánunk tenni, akkor **PUBLIC**-ká kell nyilvánítanunk annak a modulnak az elején, amelyben a szimbólumot definiáljuk:

```
PUBLIC     sz1[, sz2...]
```

Azokban a modulokban, amelyekben más modulban definiált szimbólumokat is használni szeretnénk, az ilyen szimbólumokat **EXTRN**-né kell nyilvánítanunk:

```
EXTRN sz1:típus1[, sz2:típus2...]
```


INCLUDE utasítás

```
INCLUDE     File_Specifikáció
```

hatására az assembler az **INCLUDE** utasítás helyére bemásolja az utasítás paraméterében specifikált file szövegét.

Az **INCLUDE**-olt file-ok is tartalmazhatnak **INCLUDE** utasítást.

Ha makró definícióinkat a **MyMacros.i** file-ba kívánjuk összegyűjteni, akkor célszerű ezt a file-t így elkészítenünk:

```
IFNDEF   MyMacros_i
MyMacros_i   = 1
...      ;; makró, struktúra definíciók
...      ;; EXTRN szimbólumok
ENDIF
```

Ekkor a **MyMacros.i** file legfeljebb egyszer kerül bemásolásra, mert az összes további esetben a feltételes fordítás feltétele már nem teljesül.

A **.**-ot **_**-sal helyettesítettük!

A legtöbb include file-ban ezt a konvenciót alkalmazzák.

Lista vezérlési utasítások

```
TITLE       cím
```

A fordítás során keletkező lista minden oldalán megjelenik ez a **cím**. Egy modulon belül csak egyszer alkalmazható.

```
SUBTITLE    alcím
```

Többször is előfordulhat egy modulon belül. A program lista minden oldalán – a cím alatt – megjelenik az utolsó **SUBTITLE** utasításban megadott **alcím**.

PAGE [op1][,op2]

Paraméter nélkül lapdobást jelent.

Ha egyetlen paramétere van és az egy **+** jel, akkor a fejezet sorszámát növeli eggyel, és a lapszámot **1**-re állítja.

Ettől eltérő esetekben **op1** az egy lapra írható sorok (**10≤op1≤255**), **op2** az egy sorba írható karakterek számát jelenti (**60≤op2≤132**).

Ha valamelyik paramétert nem adjuk meg, akkor természetesen nem változik a korábban beállított értéke.

A sorok száma kezdetben **66**, a karaktereké **80**.

A **TITLE**, a **SUBTITLE** és a **PAGE** egy elkészült programcsoport végső papír-dokumentációjának jól olvashatóságát segíti.

A programfejlesztés során ritkán készítünk program listákat.

NAME név

A modul nevét definiálhatjuk vele. A szerkesztő ezt a nevet fogja használni. Ha nem szerepel **NAME** utasítás a modulban, akkor a **TITLE** utasítással megadott cím a modul neve. Ha ez sincs, akkor a file nevének első **6** karaktere lesz a modul neve.

COMMENT határoló_jel szöveg határoló_jel

Segítségével több soros kommentárokat írhatunk. Az assembler a **COMMENT** utáni első látható karaktert tekinti **határoló_jel**-nek, és ennek a jelnek az újabb előfordulásáig minden kommentár. Nyilvánvaló, hogy a kommentár belsejében nem szerepelhet **határoló_jel**.

%OUT szöveg

Amikor ehhez az utasításhoz ér a fordítóprogram, akkor a paraméterként megadott **szöveg**-et kiírja a képernyőre.

.RADIX számrendszer_alapja

Ha programban egy szám nem tartalmaz számrendszer jelölést, akkor az illető számot ebben a számrendszerben kell érteni (alapértelmezésben decimális).

.LIST

Engedélyezi a forrás- és tárgykódú sorok bekerülését a lista file-ba (alapértelmezés).

.XLIST

Tiltja a forrás- és tárgykódú sorok bekerülését a lista file-ba. Jól használható arra, hogy **INCLUDE** előtt tiltsuk a listázást, utána **.LIST** -el újra engedélyezzük, és ezzel az **INCLUDE** file-ok többszöri listázását elkerüljük.

.LFCOND

Minden feltételes blokk kerüljön listára.

.SFCOND

Csak a teljesülő feltételes blokkok kerüljenek listára (alapértelmezés).

.TFCOND

Vált a két előző listázási mód között.

.CREF

Készüljön kereszthivatkozási (cross-reference) tábla (alapértelmezés). Ez a tábla azt a célt szolgálja, hogy könnyen megtaláljuk az egyes változókra történő hivatkozásokat a programban.

.XCREF

Ne készüljön kereszthivatkozási tábla.

.LALL

Kerüljön listára a makró hívások kifejtése.

.SALL

Ne kerüljön listára a makró hívások kifejtése.

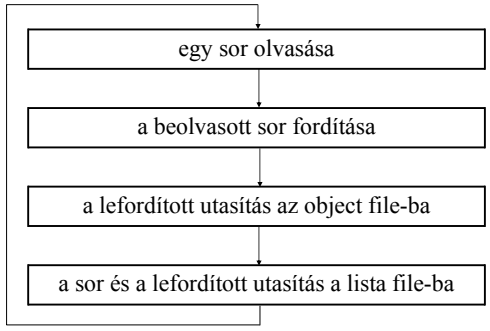
.XALL

A makró hívások kifejtéséből csak a kódot generáló rész kerüljön listára (alapértelmezés).

END kifejezés

A modul végét jelzi, **kifejezés** a program indítási címe. Ha a programunk több modulból áll, akkor természetesen csak egy modul végén adhatunk meg kezdő címet.

Assembler



előre hivatkozási probléma

Megoldási lehetőség:

Az assembler kétszer olvassa a program szövegét (két menet). Az első menet célja összegyűjteni, táblázatba foglalni a szimbólum definíciókat, így a második menet idején már minden (a programban definiált) szimbólum ismert, tehát a második menetben már nem jelentkezik az előre hivatkozási probléma.

Valahogy megpróbálni a fordítást egy menetben. Késleltetni a fordítást ott, ahol előre hivatkozás van, pl. táblázatba tenni a még le nem fordított részeket. A menet végén már minden szimbólum ismert, ekkor feldolgozni a táblázatot. Esetleg minden szimbólum definíciót követően azonnal feldolgozni a szimbólumra vonatkozó korábbi hivatkozásokat.

Mindkét esetben szükség van szimbólum tábla készítésére, de az utóbbi megoldásban a még le nem fordított utasítások miatt is szükség van táblázatra. További nehézséget jelent, hogy nem sorban készülnek el a tárgy kód (object code) utasításai, ezért ezeket pl. listába kell helyezni, majd rendezni a listát, és csak ezután történhet meg az object és a lista file elkészítése.

Manapság a legtöbb assembler két menetben működik.

Két menetes assembler, első menet

Legfontosabb feladata a szimbólum tábla felépítése.

A szimbólum tábla:

A szimbólum neve	értéke	egyéb információk
. . .	. . .	. . .

érték: – címke címe,
– változó címe,
– szimbolikus konstans értéke.

A szimbólum neve	értéke	egyéb információk
. . .	. . .	. . .

egyéb információk:

- típus,
- méret,
- szegmens neve, amelyben a szimbólum definiálva van,
- relokációs flag,
- . . .

Literál:

pl. az **IBM 370**-es gépcsaládon:

```
L       14,=F'5'  ; Load register 14 az 5-ös
                  ; Full Word konstanssal
```

Többek között a literálok gyakori használata vezetett a közvetlen operandus megadás kialakulásához és elterjedéséhez.

Egy lehetséges operációs kód tábla részlete:

mnemonic	op1	op2	kód	hossz	osztály
AAA	-	-	37	1	6
ADD	reg8	reg8	02	2	10
ADD	reg16	reg16	03	2	11
...	...	...	...	...	...
AND	reg8	reg8	22	2	10
AND	reg16	reg16	23	2	11
...	...	...	...	...	...


```
procedure ElsőMenet;  {1. menet, vázlat}
const méret = 8; EndUtasítás = 99;
var
  HelySzámláló, osztály, hossz, kód:
    integer;
  VanInput: boolean;
  szimbólum, literál, mnemo:
     array[1..méret] of char;
  sor: array[1..80] of char;
begin
  Előkészítés;
  TáblákIinicializálása;
  HelySzámláló := 0;
  VanInput = true;
```



```
while VanInput do begin      {sorok feldolgozása}
  SorOlvasás(sor);
  Megőrzés(sor);
  if NemKomment(sor) then begin  {nem kommentár}
    SzimbólumDef(sor, szimbólum);
    if szimbólum[1] <> ' ' then
                          {szimbólum definíció}
      ÚjSzimbólum(sor,szimbólum, HelySzámláló);
    LiterálKeresés(sor, literál);
    if literál[1] <> ' ' then
      ÚjLiterál(literál);
    hossz := 0;
    OpKódKeresés(sor, mnemo);
    OpKódTáblában(sor, mnemo, osztály, kód);
```



```
      if osztály < 0 then   {nem létező utasítás}
        PszeudoTáblában(sor,mnemo, osztály, kód);
      if osztály < 0 then HibásOpKód;
      hossz := típus(osztály);  {utasítás hossza}
      HelySzámláló := HelySzámláló + hossz;
      if osztály = EndUtasítás then begin
        VanInput := false;
        LiterálTáblaRendezés;
        DuplikátumokKiszűrése;
        Lezárások;
      end;   {if osztály = }
    end;     {nem kommentár}
  end;       { while VanInput }
end;         {1. menet}
```


OpKódKeresés eljárás triviális, mindössze az a feladata, hogy a ***sor***-ban az első szóköz után a látható karaktereket a következő szóközig terjedően ***mnemo***-ba másolja.

OpKódTáblában eljárás meglehetősen bonyolult, az operandusok elemzésével kell megállapítania, hogy az utasítás melyik ***osztály***-ba tartozik. Látszólag feleslegesen határozza meg a ***kód***-ot, de a többi feladata mellett ez már nagyon egyszerű, és így ez a függvény a második menetben változtatás nélkül alkalmazható.

Az ***OpKódTáblában*** eljárás nem alkalmas pl. az **ORG** pszeudo utasítás feldolgozására! Nem ismeri a ***HelySzámláló***-t.

A ***SorOlvasás(sor); Megőrzés(sor);*** arra utal, hogy a második menetben olvashatjuk az első menet eredményét. Pl. az első menet folyamán szokás elvégezni az **INCLUDE** utasításokhoz, a makró definíciókhoz és makró hívásokhoz tartozó feladatokat.

Az ***Előkészítés*** valami ilyesmi lehet:

```
 Push(NIL);     {sehova mutató pointer a verembe}
  InputFileNyitás;
  p = ProgramSzövegKezdete;
...
```

A továbbiak során ***p*** mutatja a következő feldolgozandó karaktert.

A ***SorOlvasás*** eljárás:

```
begin
  while p↑ = EOF do begin
    Pop(p);
    if p = NIL then ENDHiba; {nincs END utasítás}
  end;
  EgySorOlvasás(sor);
end;
```

Ha ***sor*** történetesen **INCLUDE** utasítás, akkor az ***EgySorOlvasás*** eljárás ezt a következőképpen dolgozhatja fel:

```
Push(p);
IncludeFileNyitás;
p = IncludeSzövegKezdete;
```



```
procedure MásodikMenet; {2. menet, vázlat}
const méret = 8; EndUtasítás = 99;
var
  HelySzámláló, osztály, hossz, kód: integer;
  VanInput: boolean;
  szimbólum, mnemo: array[1..méret] of char;
  sor: array[1..80] of char;
  operandusok[1..3] of integer;
    {op1, op2, címzési mód byte}
  object: [1..10] of byte;
begin
  Előkészítés2;
  {nincs TáblákInicializálása;}
  HelySzámláló := 0;
  VanInput = true;
```



```
while VanInput do begin      {sorok feldolgozása}
  SorOlvasás2(sor);
  {nincs Megőrzés(sor);}
  if NemKomment(sor) then begin  {nem kommentár}
    SzimbólumDef(sor, szimbólum);
    if szimbólum[1] <> ' ' then
                          {szimbólum definíció}
       SzimbólumEllenőrzés
        (sor, szimbólum, HelySzámláló);
    {nincs LiterálKeresés(sor, literál);
    hossz := 0;
    OpKódKeresés(sor, mnemo);
    OpKódTáblában(sor, mnemo, osztály, kód);
```



```
    if osztály < 0 then   {nem létező utasítás}
       PszeudoTáblában(sor,mnemo, osztály, kód);
    if osztály < 0 then HibásOpKód2;
       {Most készül a lista!}
     case osztály of
       0: hossz := fordít0(sor, operandusok);
       1: hossz := fordít1(sor, operandusok);
       ...
     end;
     Összeállítás
       (kód, osztály, operandusok, object);
     ObjectKiírás(object);
     Listázás(HelySzámláló, object, sor);
     HelySzámláló := HelySzámláló + hossz;
```



```
      if osztály = EndUtasítás then begin
        VanInput := false;
        {nincs LiterálTáblaRendezés;
        DuplikátumokKiszűrése;}
        Lezárások2;
      end;    {if osztály = }
    end;      {nem kommentár}
  end;        {while VanInput }
end;          {2. menet}
```


Összeállítás = assemble

Az assembler számos hibát ismerhet fel:

- használt szimbólum nincs definiálva,
- egy szimbólum többször van definiálva,
- nem létező operációs kód,
- nincs elegendő operandus,
- túl sok operandus van,
- hibás kifejezés (pl. 9 egy oktális számban),
- illegális regiszter használat,
- típus hiba,
- nincs **END** utasítás,
- . . .

Számos olyan hibát azonban, melyet a magasabb szintű nyelvek fordítói könnyen felismernek – vagy egyáltalán elő se fordulhatnak – az assembler nem tud felderíteni:

- az eljárás hívás paramétereinek típusa nem megfelelő,
- a regiszter mentések és helyreállítások nem állnak „párban”,
- hibás vagy hiányzik a paraméter vagy a lokális változó terület ürítése a veremből,
- a hívás és a hívott eljárás helyén érvényes **ASSUME**-ok ellentmondásosak (nem feltétlenül hiba, de az lehet),
- . . .

Az object file nemcsak a lefordított utasításokat tartalmazza, hanem további – a szerkesztőnek szóló – információt is.

Makró generátor

Feladata a makró definíciók megjegyzése (pl. makró táblába helyezése) és a makró hívások kifejtése. Általában az assembler első menetében működik.

Makró definíciók felismerése

Amennyiben az assembler a forrás szöveg olvasása közben makró definíciót talál (ezt könnyű felismerni a műveleti kód részen lévő **MACRO** szó alapján), akkor a makró definíció teljes szövegét – az **ENDM** műveleti kódot tartalmazó sor végéig – elhelyezi a makró táblában. A felismerést és a tárolást a ***SorOlvasás*** vagy a ***PszeudoTáblában*** eljárás végezheti.

Makró hívások kifejtése

Az

```
OpKódTáblában(sor,mnemo, osztály, kód);
if osztály < 0 then       {nem létező utasítás}
    PszeudoTáblában(sor, mnemo, osztály, kód);
```

programrész után be kell szúrni az

```
if osztály < 0 then
    MakróTáblában(sor, mnemo, osztály, kód);
```

sorokat. A eljárás feladata a makró hívás felismerése és a makró helyettesítés is. A kifejtett makró egy pufferbe kerül, a puffer tartalma az **INCLUDE** utasításnál látottakhoz hasonlóan illeszthető a program szövegébe.

A makró kifejtés egy ciklusban:

```
EgySzóOlvasásaAMakróTörzsből;
  if FormálisParaméter then
    AMegfelelőAktuálisParaméterÁtmásolása;
  else
    ASzóÁtmásolása;
  ElválasztójelFeldolgozása;
```

Az ***ElválasztójelFeldolgozása*** legtöbbször az elválasztójel másolását jelenti, de a makró definícióban különleges szerepet játszó karakterek esetén ettől eltérő – magától értetődő – speciális feladatot kell végrehajtani.

A **LOCAL** utasítás feldolgozásához a makró generátor egy **0** kezdeti értékű változót használ. Makró híváskor a **LOCAL** utasításban szereplő szimbólumot, és az összes előfordulását a makró törzsben **??xxxx** alakú azonosítóval helyettesíti, ahol **xxxx** a változó aktuális értéke hexadecimális számrendszerben. A változó értékét minden a **LOCAL** utasításban szereplő szimbólum feldolgozása után **1**–gyel növeli.

Legegyszerűbb, ha a lokális szimbólumot formális paraméternek tekinti, és a generált **??xxxx** alakú azonosítót a megfelelő argumentumnak.

Szerkesztő

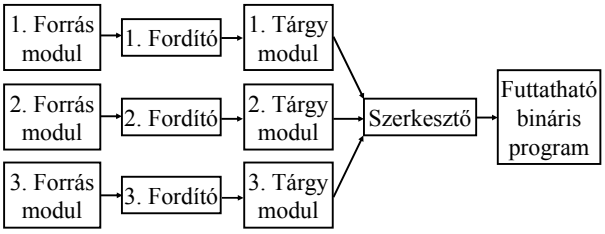
A következő feladatokat kell megoldania:

- az azonos nevű és osztályú szegmens szeletek egymáshoz illesztése a szegmens szeletek definíciójában megadott módon,
- a **`GROUP`** pszeudo utasítással egy csoportba sorolt szegmensek egymás után helyezése,
- a relokáció elvégzése,
- a külső hivatkozások (**`EXTRN`**) feloldása.

Az object file nemcsak a lefordított utasításokat tartalmazza, hanem további – a szerkesztőnek szóló – információt is.

Szerkesztő (~7.13. ábra)

Lehetővé teszi a program akár különböző nyelveken készített részleteinek összeillesztését.



Két menetben dolgozik:

- Az első menetben táblázatokat készít (térkép – map és globális szimbólum tábla).
- A második menetben a táblázatokban elhelyezett információk alapján elvégzi a szerkesztést.

A térkép (map) szegmens szeletenként a következő információt tartalmazza:

- modul név,
- szegmens név,
- osztály,
- illesztés típusa,
- kombinációs típus,
- hossz,
- kezdőcím,
- relokációs konstans.

Az első menet végén a térképet átrendezi, majd kitölti kezdőcímeket és a relokációs konstansokat.

A globális szimbólum tábla a **`PUBLIC`** változókból:

modul név	szegmens név	szimbólum	típus	cím
...	...	...	...	...

A **`PUBLIC`** utasítás nem tartalmazza a típust és a szegmens nevét, de az assembler ismeri, és el tudja helyezni a tárgy (object) modulban.
A cím a táblázat összeállításakor még relokálatlan címet jelent. Az első menet végén ebben a táblázatban is elvégezhető a relokáció.

Az assembler az **`EXTRN`** utasítás alapján a következő információt adja át:

		hivatkozás1		hivatkozás2		...
szimbólum	**típus**	**szegmens név**	**offszet cím**	**szegmens név**	**offszet cím**	...
...	...	...	...	...	...	...

Definiálatlan külső hivatkozások.

Moduláris programozás.

Object könyvtárak.

Dinamikus szerkesztés

Nagyméretű programokban bizonyos eljárások csak nagyon ritkán szükségesek. Ezeket nem kell statikusan a programhoz szerkeszteni.

Csatoló táblázat (Linkage Segment): minden esetleg szükséges eljáráshoz egy csatoló blokkot (struktúrát) tartalmaz.

```
CSAT_STR  STRUCT
CIM       DD     FAR PTR CSATOLO
NEV       DB     '         '; 8 szóköz
CSAT_STR  ENDS
```



```
CSAT_STR   STRUCT
CIM        DD     FAR PTR CSATOLO
NEV        DB     '         '; 8 szóköz
CSAT_STR   ENDS
```

Pl. a dinamikusan szerkesztendő **ALFA** eljáráshoz az:

```
ALFA       CSAT_STR     <, 'ALFA'>
```

csatoló blokk tartozhat. Az eljárás csatolása, hívása:

```
           MOV    BX, OFFSET ALFA
           CALL   [BX].CIM
```

Első híváskor a **CSATOLO** kapja meg a vezérlést. A csatolandó eljárás neve a **[BX].NEV** címen található. A név alapján betölti és a programhoz szerkeszti a megfelelő eljárást.


```
CSAT_STR   STRUCT
CIM        DD     FAR PTR CSATOLO
NEV        DB     '         '; 8 szóköz
CSAT_STR   ENDS
```

A szerkesztés végeztével **CIM** -et a most a programhoz szerkesztett eljárás kezdőcímére változtatja, és erre a címre adja a vezérlést:

```
           JMP    [BX].CIM
```

JMP, és nem **CALL**, hogy a veremben a **CSATOLO** -t hívó programhoz való visszatérés címe maradjon.

További hívások esetén a **CSATOLO** közbeiktatása nélkül azonnal végrehajtásra kerül az imént csatolt eljárás.

CSATOLO használhatja és módosíthatja a program szerkesztésekor készült térképet (map) és a globális szimbólumok táblázatát.

Szokásos megszorítás: a csatolandó eljárás nem tartalmazhat **EXTRN** utasítást, és egyetlen, a memóriába bárhová betölthető modulból áll. Ekkor a szerkesztés magára a betöltésre, és ennek a tényét rögzítő adminisztrációra egyszerűsödik.

A dinamikusan szerkesztett eljárásokat könyvtárakba szokás foglalni (pl.: **.dll**), az eljárások általában többszöri belépést tesznek lehetővé (re-entrant).

Továbbfejlesztés:

A csatoló paraméterként kapja meg a felhívandó eljárás nevét.

A csatoló program hoz létre egy csatoló táblázatot.

A csatoló a táblázatban ellenőrzi, hogy csatolva van-e a kívánt eljárás. Ha nincs, akkor elvégzi a csatolást, ha pedig csatolva van, akkor közvetlenül meghívja az eljárást.

Menü vezérelt rendszer esetében, ha a kiválasztott menü elemhez tartozó szövegből generálni lehet az – esetleg csatolandó, majd – végrehajtandó eljárás nevét és az eljárást tartalmazó file nevét, akkor a program bővítését, javítását a megfelelő file-ok hozzáadásával vagy cseréjével és a menü szöveg file-jának cseréjével akár üzemelés közben is elvégezhetjük.

A csatolt program törlése: a törlendő eljárás csatoló blokkjában a **CIM** visszaállítása – illetve a csatoló tábla törlése – után az eljárás törölhető.

Szokásos, hogy egy dinamikusan szerkesztendő eljáráshoz tartozik egy számláló, melynek értéke a csatolás előtt **0**.
A hívó program először „bejelenti” az igényét az eljárásra. Ha a számláló **0**, akkor megtörténik a csatolás, és mindenképpen: számláló **++**.
Ha a továbbiakban már nem igényli az eljárást, akkor „elengedi”: számláló **--**.
Ha a számláló **=0**, akkor senki sem igényli az eljárást, tehát törölhető.

Programok hangolása

Két operációs rendszer	MULTIX	TSS/67
Alkalmazott programozási nyelv	95%-ban PL/I	assembly
Program lista	3.000 oldal	30.000 oldal
Programozók száma	50	300
Költség	10 millió $	50 millió $

Egy programozó néhány évig egy nagyobb feladaton dolgozva havi átlagban csak kb. 100-200 (!!) ellenőrzött utasítást ír, függetlenül az alkalmazott programozási nyelvtől, és egy **PL/I** utasítás 5-10 assembly utasításnak felel meg.

Sokkal gyorsabb **TSS/67** ?

Irodalmi adatok alapján azt lehet mondani, hogy (hangolás előtti) nagyobb programok 1%-a felelős a program futási idejének kb. 50%-áért, 10%-a a 90%-áért.

Program hangoláson azt a folyamatot értjük, amikor megállapítjuk a kritikus részeket, és ezek gyorsításával az egész program futását felgyorsítjuk.

A kritikus részek felderítése.

Tételezzük fel, hogy ugyanannak a feladatnak a megoldásához assemblyben 5-ször annyi utasításra (és időre) van szükség, mint probléma orientált nyelv esetén, és az elkészült program 3-szor olyan gyors. A probléma orientált nyelven készült változatának kritikus 10%-át assemblyben újra programozzuk.

A költségek és futási idők alakulása:

	programozó év	futási idő
Assembly nyelv	**50**	**333**
Probléma orientált nyelv	**10**	**1000**
Hangolás előtt		
a kritikus 10%	**1**	**900**
a többi 90%	**9**	**100**
Összesen	**10**	**1000**
Hangolás után		
a kritikus 10%	**1+5**	**300**
a többi 90%	**9**	**100**
Összesen	**15**	**400**

- A program probléma orientált nyelven történő elkészítésének és hangolásának ideje (és költsége) kb. harmada (15 programozó év) annak, mintha az egészet assemblyben készítenénk (50 programozó év),
- a sebessége csak 20%-kal gyengébb (333 helyett 400).