



Gábor Dénes  
Főiskola

ÁGOSTON GYÖRGY:

### ASSEMBLY

HALLGATÓI SEGÉDLET



2001/2002  
agoston@gdf-ri.hu

## Kettes komplemens

Képzése:

$$N_k = \begin{cases} N, & \text{ha } N \geq 0 \\ 2^{k+1} - |N|, & \text{ha } N < 0 \end{cases}$$

ahol k a felhasználható bitek száma (pl. 8).

Például a +6 kettes kompl. kódja: 0000 0110 b

(Mivel a szám pozitív, kettes számrendszerbeli alakjában tárolódik.)

## Miért tanulunk Assemblyt?

- A nyelv rálátás szintű ismerete az általános számítástechnikai képzettség elengedhetetlen része;
- A jó Assembly (ejtsd: **eszembli**) programozó kezelt és ritka;
- Nemcsak a GDF-en tanítják, a BME több karán, a volt Kandón (jelenleg BMF) és külföldön is.

Kapcsolódó téma: Bevezetés a számítástechnikába, Programozás, Mikroszámítógépek

A -6 kódja k=8 esetén, a definíció alapján:

$$2^{k+1} - |N| = 2^9 - 6 \rightarrow \begin{array}{r} 1\ 0000\ 0000\ b \\ -\ 0000\ 0110\ b \\ \hline 1111\ 1010\ b \end{array}$$

Más módszer: jobbról balra leírom az első 1-es bitig (ezt még leírom) és innentől minden bitet negálok.

Igy a -6 alakja: +6 → 0000 0110 b → 1111 1010 b

**A képzés módjából következően egy szám és kettes komplemensének összege az ábrázolási tartományon (k) belül nullát ad.**

A world leader in anti-virus software is looking for  
**virus analysts**  
with excellent 80x86 Assembler skills. You will be a graduate with at least  
two years of professional programming experience.  
We offer a very pleasant working environment next to Oxford overlooking  
the river Thames and green fields.  
The package includes profit-related pay and private medical insurance.  
Please e-mail your CV to Dr. Jan Hruska (jhr@sophos.com).  
For more information on Sophos visit <http://www.sophos.com/>  
Sophos PLC, The Pentagon, Abingdon Science Park, Abingdon, OX14 3YP,  
England. Tel 00 44 1235 559933, Fax 00 44 1235 559935

**SOPHOS**

**COBOL/ASSEMBLER**  
Erfahrene Programmierer mit guten Deutschkenntnissen für Zeitverträge  
in Deutschland oder in der Schweiz gesucht.  
Bitte senden Sie Ihre Kurzwerbung an Chiffre COBOL,  
HVG hirdetési igazgatóság, 1037 Budapest, Szépvölgyi út 35.

### Újsághirdetések, 1997-1999.

Virusanalizáló: Feltétel az alacsony szintű DOS, Windows programozási ismeret, Assembly és C gyakorlat. Előnyt jelent a Linux, SQL, PHP3, Java ismeret.

Elektronikus hirdetés (2000. nyár, [www.vbuster.hu](http://www.vbuster.hu))

**Az Intel mikroprocesszorok az egész számokat kettes komplemens alakban tárolják. Előnye: az összeadás és kivonás a negatív szám tartományban is helyes eredményt ad!**

**Példa: 8 - 6 = 2 (k=8)**

$$\begin{array}{r} 0000\ 1000\ b \\ +\ 1111\ 1010\ b \\ \hline 10000\ 0010\ b \end{array} \quad \begin{array}{l} (8) \\ (-6) \\ (2) \end{array}$$

túlcordulás, levágjuk

## I. ISMÉTLÉS

### Egyes komplemens

Más neve: inverz bináris kód.

Képzése:

- A pozitív számok a kettes számrendszerbeli alakjukban tárolódnak;
- Negatív szám esetén a pozitív alak minden bitjét negálom.

Például a -6 egyes komplemensbeli alakja:

$$+6 = 0000 0110 b \rightarrow \underline{1111 1001 b}$$

Megjegyzések:

- A szám legmagasabb helyértékű bitje az előjelet jelzi: 0 ha pozitív, 1 ha negatív;
- 1 bájton az ábrázolható tartomány:  $-128 \leq x \leq 127$ ;
- Assemblyben majd egy-egy utasítás állítja elő egy szám egyes/kettes komplemensét (NOT és NEG).

$$\begin{array}{r} -128 = 1000\ 0000\ b \\ 127 = 0111\ 1111\ b \end{array}$$



## Paritásvédelem

Az egyik legegyszerűbb adatvédelmi módszer.  
Az átvitt kódszóhoz járulékos bitet (paritásbitet) rendelnek hozzá.

Fajtái:

- páros paritásvédelem: a paritásbittel együtt összesen páros db. 1-es bit legyen  
Például a 0000101 paritásbitje 0.
- páratlan paritásvédelem: összesen prt. 1-es legyen  
Például a 0000101 paritásbitje 1.

**A programozás speciális gondolkodást igényel. A problémát a számítógép nyelvén kell tudnunk megfogalmazni.**



A PROBLÉMA



MIRE KÉPES A SZÁMÍTÓGÉP?



BONYODALMAK

*Általában a páratlan paritásvédelmet használják, itt ugyanis nem lehet csupa nullából álló kódszó (párosnál pl. 0000000 esetén a paritásbit is 0).*

A keresztrányú paritásbiteken kívül időnként hosszirányú paritáskaraktert is képeznek:

0110100	0
1011010	1
1001100	0
0101111	1
0000000	1
1110110	0

Ennél a módszernél egy bit hibája felfedezhető és kijavítható.  
(Az aláhúzott bit hibás.)

A legutolsó bit a függőleges paritásbitekre vonatkozik!

**Egy program elkészítéséhez adott az összes utasítás, csak azt kell eldönteni, melyik után mi jöjjön.**



Felhasználása: az IBM PC-kben a RAM memóriát páratlan paritásvédelemmel védi (ha védik): memóriába írásnál bájtonként generálnak egy paritásbitet, amelyet általában egy külön chip tárol. Kiolvásásnál kiolvassák a bájtot, képezik a paritásbitjét, kiolvassák az eltárolt paritásbitet is és összehasonlíják. Ha megegyezik → minden rendben. Ha nem → RAM paritás hiba (NMI generálódik, lásd később).

Az I/O adatbuszt is prt. paritással ellenőrzik (hiba → NMI). A ROM paritását nem ellenőrzik, helyette ún. ellenőrző összeget - checksum - használnak (l. később).

A soros porton keresztül történő adatátvitelnél a DOS MODE parancsával állíthatjuk be a paritásvédelem jellemzőit (páros/prt paritás, adatbitek száma).

**Milyen nyelven programozható a mikroprocesszor?**

## Mi a számítógépes program?

**Olyan, előre kidolgozott utasítások sorozata, amelyeket a mikroprocesszor megadott sorrendben végrehajt.**

**(Mindig mászt tud végrehajtani, ebben különbözik a HARD-WIRED programtól, pl. mosógép.)**

## Programozás lehetőségei

- gépi kód (első gen. nyelv, 1GL)
  - Assembly (2GL)
  - C nyelv (3GL)
  - magas szintű nyelvek (4GL, 5GL stb.)

# A gépi kód

- Számok sorozata (bájtok  $\rightarrow$  0..255, 0..FFh);  
↓  
CPU típusától függő utasításokat és adatokat jelent
  - A CPU bekapcsolástól kikapcsolásig (kizárolag) gépi kódú utasítások millióit hajtja végre;
  - Egy-egy utasítás a CISC CPU-kban ún. mikroprogramot indít el (RISC-ben huzalozott megoldás);
  - A .COM és .EXE kiterjesztésű fájlok gépi kódú futatható programok (a .BAT nem);
  - Nehéz megérteni.

## Példa gépi kódú programra:

## Írunk ki a képernyőre egy "A" betű!

A

#### A program IBM PC-n (kézikönyv alapján):

B4	02	B2	41	CD	21	CD	20	hex
180	2	178	65	205	33	205	32	dec

## **Próbáljuk ki!**

```
C:\>copy con proba.com
+^B#A!=^Z
          1 file(s) copied
C:\>proba
A
C:\>
```

A számokat bebillentyűzzük (bal Alt+3 számjegy), a végén Ctrl+Z és Enter.

## II. AZ ASSEMBLY ELEMEI

Assembly → a gépi kódú programozást megköny-  
nyítendő, az utasítást jelentő számoknak emlékez-  
tető szótörédekét, ún. *mnemonik*-ot feleltetnek  
meg.

E számok helyett a mnemonikot írjuk le, az adatok továbbra is megmaradnak számoknak. Továbbá ugrásoknál és egyéb hivatkozásoknál ún. címkével jelöljük ki a helyet, a pontos memóriacímét a gép fogja kiszámlálni. Így az Assembly nyelv a fizikai címzés fárasztó meghatározása alól is mentesít.

### Az előző program Assemblyben:

## **Az Assembly jellemzői**

- Alacsony szintű programozási nyelv (regiszter és I/O műveletek; másfelől hagyományos, soronkénti programkészítés);
  - Hardverfüggő, azaz sok CPU és gépfelépítéssel kapcsolatos ismeret szükséges hozzá;
  - Közvetlenül gépi kódra fordítható (egy-egy utasításból fordítás után 1..10-20 bájt lesz);
  - Más programnyelvekkel összehasonlítva a készített kód a legrövidebb (fájlméret) és a leggyorsabb lesz.

Az összehasonlítást végezzük el egy mintafeladataon (semmilyen következtetést nem vonhatunk le!):

*számoljunk el 0-tól 65535-ig a képernyőn!*

## Az elkészített programok futási eredményei: (Pentium II, 266 MHz CPU, Win 98, teljes képernyős DOS ablak)

<i>programnyelv:</i>	<i>fájlméret:</i>	<i>futásidő:</i>
Borland Turbo C 2.0	8 994 bájt	2,4 sec
Turbo Pascal 6.0	3 888 bájt	0,5 sec
Assembly (.COM)	80 bájt	0,15 sec

(A forráskód a példaprogramok között szerepel.)

- A gépi kódra fordító program neve: **ASSEMBLER**;
  - A visszafordító (!!!) program: **DISASSEMBLER**;
  - Használata: régebben elterjedt, az operációs rendszerek nyelve volt (újabban C-ben írják); ma már csak különleges feladatoknál (pl. ahol nagy adatátviteli sebesség, vagy kis programméret szükséges; 3D grafika, játékprogramok).

**A tantárgy keretében IBM PC kompatibilis gépeken, DOS (Windows 9x) környezetben, valós üzemmódban (Intel 8086/8088..Pentium) vesszük át az Assembly nyelv alapjait!**

## Memóriacímzés

Az Intel 80x86 processzorok valós módban és első megközelítésben 1 MB memóriát tudnak megcímezni. (Ez a memória vegyesen ROM, az alaplapi max. 640 KB RAM, videomemória RAM és UMB RAM).

1 MB eléréséhez 20 bit szükséges ( $2^{20} = 1048576$  bájt = 1 MB). Ezt egy darab 16 bites regiszterrel nem lehet lefedni, ezért két regisztert használnak (többet, mint ami minimálisan szükséges, ebből származnak a bonyolódalmak).

Bevezető példa a memóriacímzéshez:



(Danone joghurt címke)

A gyártókód első 3 számjegye jelöli, hogy az év melyik napján készült a termék (a példában a 165. napon). Az érték 1 és 366 közötti és tökéletesen rátmutat az éven belül az adott napra. Hasonló lesz Assemblyben az ún. lineáris (fizikai) címzés. (A többi jegy: az év utolsó számjegye, a töltés ideje óra-percén és a töltőgép jele, de ez most nem lényeges.)

A lejáratí idő két számából áll. Az első (07=július) kijelöl a teljes év-intervallumban egy sávot. A második (12) pedig e sávon belül jelöli ki a napot. Tehát ha a 2. szám értéke 1 → a kijelölt terület első elemére mutat, ha 31 → utolsó elemére. Hasonló lesz Assemblyben az ún. szegmens-offset címzés. Az itteni első szám (hónap) lesz a szegmens érték, a második (nap) az offset érték. A sáv (1 hónapnyi terület) neve pedig szegmens.

Visszatérve a processzorhoz, a memóriacímzés módja:

**Az első érték (regiszter) a memória egy pozíciójára mutat, és innentől NÖVEKVŐ irányban 64 KB területet tudunk elérni: írni vagy olvasni. Az ezen belüli pontos helyet jelöli ki a másik regiszter.**

Az első regiszter által mutatott memóriacím az ún. szegmens (bázis), a másik az offset\* (eltolási) cím.

A módszer neve: "szegmens-offset címzés".

\* Megjegyzés: Az elnevezés nem véletlen. A nyomdaiparban az offset nyomás esetén a nyomólemez nem ér közvetlenül a papírhoz, hanem egy gumihengerhez, mely felületére rákerül a nyomtatandó információ, majd továbbforgáskor az nyomja rá a papírra.

Szokásos jelölése: 4\_jegyű\_hexa\_szegmenscím : offsetcím

Példák: A000:0000h, B800:0015h stb.

A szegmens-offset cím átszámítása lineáris, fizikai címre:

$$\text{fizikai cím} = 16 * \text{szegmens} + \text{offset}$$

Pl. az 1234:0017h fizikai címe:  $12340h + 0017h = 12357h$

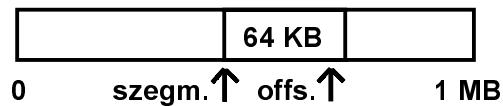
A 16-tal történő szorzás hexa-decimális számoknál egy nulla hozzáírását jelenti.

Még egy példa:

A hidegindítás kezdőcíme FFFF:0000h

Lineáris (fizikai) cím: FFFF0h + 0h = FFFF0h

A helyzetet bonyolítja, hogy négy szegmensregiszterünk van: 4 darab 64 KB-os memóriaterületen dolgozhatunk.



**Következmények:**

1. Egy szegmens mérete 64 KB (mert az offsetcím 16 bites). A .COM állományoknak bele kell férnie egy szegmensbe, így nem lehetnek ennél nagyobbak.
2. A redundancia miatt (2\*16 bittel címünk 20 bitet) ugyanaz a fizikai cím többféle szegmens-offset formában is megadható. Pl. a hidegindítás kezdőcíme F000:FFF0h is lehet. Szokás a normalizált értéket használni, ahol az offset legfeljebb 15. A memória 0..15 bájtjai viszont csak egyféléképpen címezhetők (mert ekkor a szegmens értéke nulla).
3. A szegmens kezdete nem lehet bármi, hanem a fizikai memória 0., 16., 32. stb. bájtja. Ez az ún. paragrafus (16 bájt). Tehát a szegmens paragrafushatáron kezdődik és a szegmensek átfedhetik egymást.
4. Ha a szegmensregiszter FFFFh és az offset  $\geq 10h$ , kilépünk az 1 MB memóriából! Valós módban, IBM AT gépeknél ezek szerint megcímezhető az 1 MB feletti RAM 16 bájt híján 64 KB-ja. Ez az ún. High Memory Area - HMA. Az MS-DOS 5.00-s verziótól felfelé lehetőséget nyílik programok futtatására a HMA-ban, amelyek így nem terhelik a hagyományos memóriát. A Windows elterjedésével a HMA jelentősége csökkent. Azaz a valós módban megcímez-

hető memória mérete valójában 16 bájt híján 1088 KB!

5. A szegmens-offset címzés viszonylag bonyolult, más processzorokban (pl. Z80) nem használják.
6. A fenti leírás a valós módot tartalmazza. Védekt módban a címzés bonyolultabb. Továbbá 80386-tól van az ún. „Unreal mode” (Real Flat Mode, Real Big Mode), amikor valós módban 4 GB címezhető meg.

## Regiszterek

A regiszterek a mikroprocesszorban helyezkednek el, adattárolásra (mint változók) és műveletvégzésre használhatók. A 80x86 regiszterei:

**1. Általános regiszterek:** AX, BX, CX, DX

- accumulator / base / counter / data register
- 16 bitesek ( $\rightarrow$  szó, word)

↓

az ábrázolható szám tartomány:

előjel nélküli: 0..65535 (0000..FFFFh)

előjeles: -32768..32767 (8000..7FFFh)

- A regiszterek felső és alsó 8 bitje külön-külön is megcímízhető, mint két 8 bites regiszter.  
Ezek neve: AH, AL, BH, BL, CH, CL, DH, DL.  
(High - magas, Low - alacsony helyérték)

Tárolható szám tartomány: 0..255 (00..FFh)  
előjelesen -128..127 (80..7Fh)

Pl. ha  $AX=1234h \rightarrow AH=12h$   $AL=34h (\rightarrow$  bájt)  
(Ezért is jó a 16-os számrendszer használata! Ugyanez decimálisan:  $AX=4660$ ,  $AH=18$ ,  $AL=52$ ;  $256 \cdot 18 + 52 = 4660$ )

- Intel 80386-tól: 32 bites regiszterek (EAX, EBX stb., az alsó 16 bites részei AX, BX stb.)

**2. Indexregiszterek:**

SI (source index, forrásindex)

DI (destination index, célindex)

16 bites regiszterek, NEM oszthatók 8-8 bitre.

A memoriában offset cím kijelölésére használhatók.

**3. Veremregiszterek:**

SP (stack pointer, verem tetejét mutatja)

BP (base pointer, vermen belüli műveletekhez)

**4. Szegmensregiszterek:**

CS (code segment, kódszegmens - programunk)

SS (stack segment, veremszegmens)

DS (data segment, adatszegmens - adataink)

ES (extra segment - bármi, pl. videomemória)

Programból az ES segítségével címezhetjük térszölegesen a memóriát, a többi regiszter értékét nem állítjuk közvetlenül.

80386+ : FS és GS (másodlagos adatszegmens regisztere) és sok újabb regiszter.

**5. Utasításmutató:**

IP (instruction pointer)

A futó program a CS által kijelölt szegmensben van. A következő végrehajtandó utasításra az IP mutat (offset cím). Így, ha a program nem tartalmaz ugró utasításokat, IP értéke folyamatosan nő (de nem minden egyesével).

A következő végrehajtandó utasítás címe CS:IP.

**6. A Flag regiszter (állapot szó, állapotregiszter, FLAGS)**

- Különálló jelzőbitek (flagek) halmaza, mint regiszter NEM használható (adattárolásra, vagy műveletvégzésre)!
- A flagek utasításuktól függően állítódnak, illetve néhány flag-et külön művelettel módosíthatunk;
- A legfontosabb jelzőbitek:

ZERO - zérus jelzőbit (ZF)  
CARRY - átvitel jelzőbit (CF)

- Zérus jelzőbit: ha aritmetikai vagy logikai művelet eredménye nulla, ZF=1 lesz. A FLAGS 6. bitje.

pl.  $AH:=1$   
 $AH:=AH-1 \rightarrow ZF=1$

a XOR igazságátablázata miatt

pl.  $AH:=$ tetszőleges szám  
 $AH:=$ bitenkénti XOR (AH,AH)  $\rightarrow AH=0$ ,  $ZF=1$

- Átvitel jelzőbit: a legmagasabb helyen keletkező átvitelt tárolja, CF=1 ha előjelváltás történik. A FLAGS legalsó, 0. bitje.

pl.  $AH:=1$  és  $AH:=AH-2 \rightarrow CF=1$

**További jelzőbitek:**

- Túlcordulás (Overflow flag) pl.  $127+1 \rightarrow OF=1!$   
Előjeles műveletek esetén OF=1 jelzi a túlcordulást, vagyis azt, hogy az eredmény nem előjelhelyes.
- Félátvitel vagy segédátvitel (Half carry, Auxiliary carry flag):  
Átvitel az eredmény 3. bitjéről a 4. bitre (bitszámozás: 76543210; BCD korrekciót jelöl) pl.  $00001111b + 1 \rightarrow AF=1$
- Előjel (Sign flag):  
Megegyezik az eredmény legmagasabb helyiértékű bitjével (SF=1 kettes komplement alakban negatív számot jelöl).  
pl.  $AL:=5$  és  $AL:=AL-10 \rightarrow SF=1$  (átfordul negatívba)

- Paritás (Parity flag):  
PF=1 ha az eredmény alsó bájtjában az 1-es bitek száma páros (prt. paritásjelzés). pl. AL:=2 és AL:=AL+1 → PF=1
- Megszakítás (Interrupt flag)  
IF=0 esetén a HW által generált megszakításokat nem szolgálja ki a mikroprocesszor.
- Egylépéses üzemmód (Trap flag)  
TF=1 esetén minden utasítás végrehajtása után az INT 1 megszakítás következik be. Nyomkövetésnél használják.
- Irányjelző (Direction flag)  
Stringműveletekben SI / DI léptetése (DF=0 → növekvő).

**A regiszterek számát és feldatolását a mikroprocesszor tervezése során határozzák meg. Számuk végleges, Assembly utasításokkal NEM hozhatunk létre újakat és nem szüntethetjük meg őket.**



## 2. Regiszter címzés:

MOV AL, BL → AL felveszi BL értékét, BL változatlan marad

MOV AX, DL → hibás, mert AX 16 és DL 8 bites

A szegmensregiszterekbe közvetlenül nem írhatunk, csak két lépésen keresztül:

pl. MOV AX,<érték> és MOV DS,AX

A MOV IP,<érték> ugyancsak nem létezik, IP értéke az ugrások során változik.

## 3. Kapcsolattartás a memóriával:

MOV AX, [SI]                    **OLVASÁS A MEMÓRIÁBÓL!**  
MOV AX, [0FF00h]

MOV [SI], AX                    **ÍRÁS A MEMÓRIÁBA!**

[SI] jelentése: a memória DS:SI szegmens-offset címen kezdődő 2 bájtos érték. A [szögletes zárójel] tehát NEM a regisztert jelenti, hanem a regiszter által kijelölt memóriarekesz(ek) értékét!

Az írás/olvasás (számok tárolása) ún. **BÁJT FORDÍTOTT** alakban történik: pl. íráskor először AL kerül a memóriába, majd a következő rekeszbe AH.

## A MOV utasítás

Az egyik legfontosabb, legsokoldalúbb Assembly parancs. Az angol move - mozgat (ejtsd: műv) szóból képzett szótöredék - mnemonik. Szintaktikája:

MOV cél, érték

### Fajtái:

#### 1. Közvetlen érték (immediate) címzés

Pl. a MOV AH,5 hatására az AH regiszter értéke 5 lesz (=Pascal AH:=5) → ÉRTÉKADÓ UTASÍTÁS!

A megcímímezhető memória nagysága 1 MB (+HMA).

Az előző példákban nem szerepelt a szegmens, csak az offset cím. Ilyenkor alapértelmezett DS. Ez azonban egy ún. szegmensfelülről prefix segítségevel módosítható:

MOV ES:[SI],AX vagy MOV CS:[DI],AL

A []-en belül felhasználható regiszterek: SI, DI, BX.

### További példák az értékadásra:

MOV AX,200 → AX értéke 200 lesz (AH=0, AL=200)

MOV AX,-5 → AX értéke -5 (0FFFh-kettes kompl.)

Az, hogy egy regiszterben előjeles (kettes komplement,  $-128 \leq x \leq 127$ ), vagy pozitív ( $0 \leq x \leq 255$ ) szám van, semmi nem mutatja, az dönti el, ahogyan használjuk!

Emllett pl. mov ah,-2 ≡ mov ah,254 stb.

MOV AX,FFh → hibás, a helyes: MOV AX,0FFh  
(mert egy "FFh" nevű címkét keres)

MOV AX,70000 → hibás, 0 és 65535 közé eshet!

MOV 5,AL → hibás (számba regisztert tölteni ???)

#### 4. Egyéb lehetőségek:

MOV AX,[DI+4]                    - indexelt címzés (SI vagy DI)

MOV AX,[BX+4]                    - bázis relatív címzés (BX)

MOV AX,[BX+DI+4]                - bázis relatív indexelt címzés

##### • ASCII kód helyett írható a karakter is:

MOV AL,82 helyett írható: MOV AL,'R'

##### • Adattípus ideiglenes átdefiniálása:

A MOV [SI],'R' utasítást elfogadja az Assembler egy warninggal (figyelmeztetéssel), de MOV WORD PTR [SI],'R'-ként fordítja le, azaz 2 bájtot ír a memóriába! Megoldás: MOV BYTE PTR [SI],'R'.

## Első Assembly programunk

Feladat a "Hello világ!" kiíratása. A forrásprogram:

```

title elso programunk ; a program neve
.model small ; kis memóriamodell (.EXE, 64 KB kód, 64 KB adat+verem)
.stack ; 1 KB veremszegmens definíálása (.stack N)
.data ; adatszegmens, benne a kiíratandó szöveg
    szoveg db 'Hello világ !$'
.code ; címkék
.start: ; címke
    mov ax,_data ; DS ráállítása az adatszegmensre
    mov ds,ax ; (csak két lépéshoz lehetséges)
    mov ah,9 ; szöveg kiíratása megszakítás segítségével
    mov dx,offset szoveg ; (DS:DX-tól ír ki '$' jelig)
    int 21h
    mov ax,4C00h ; kilépés DOS-ba hibajelzés nélkül
    int 21h
end start ; program vége, a belépési pont a "start"

```

Az INT utasításokat kezeljük egyelőre fekete dobozként: nem ismerjük működésüket, csak azt, hogyha kiadjuk a megfelelő MOV-ok után a megfelelő INT-et, a gép végrehajt valamit: kiír egy szöveget, kilép DOS-ba stb.

## Egy Assembly parancsor felépítése

címke: utasítás paraméter(ek) ; megjegyzés  
label: mnemonik operandus(ok) ; comment

Például: ide: mov ah,9 ; értékkadás

Ez a legbővebb alak. A megjegyzés a program visszafejtését segíti, címkét ugrás vagy más hivatkozás esetén használunk, a paraméterek száma a parancstól függ. Azaz szélsőséges esetben egy sorban csak egy utasítás is szerepelhet pl. NOP. A kis és nagybőyük használata egyaránt megengedett (nincs különbség).

## A programkészítés lépései

### 1. Forrásprogram megírása

Egyesű szövegszerkesztővel → .ASM állomány  
(E résznél kell gondolkodni.)



### 2. Fordítás

C:\> TASM fájlnév

Microsoft Macro Assembler, Borland Turbo Assembler vagy más fordító segítségével.

makró: programrészlet, tartalma fordításkor beszerkesztők a forrásszövegbe (kifejtés). A generált tárgykód mérete emiatt nem csökken.

A fordítás általában két lépésből áll:

- Szimbólumtáblázat elkészítése (név, érték, típus)  
Pl. ha ugrunk a program egy későbbi, címkevel jelölt pontjára, a címke értékét nem tudhatjuk előre, így nem kerülhet be a tárgykódba.
- Utasításkódok fordítása és a szimbólumértékek kitöltése.

A fordítás végeredménye az .OBJ (object-tárgykód) fájl: szabványos felépítésű, a lefordított programot tartalmazza, más programokban felhasználható (Pascal, C++, Delphi stb.), viszont még nem futtható.

Kiegészítés: TLIB - gyakran használt .OBJ fájlokat .LIB "könyvtár" állományba helyezhetünk el (< C/C++ fordítók!).

### 3. Szerkesztés

A szabványos .OBJ-(k)ből futtatható állapotra:

C:\>TLINK fájlnév (Enter)

A linker elkészíti az .EXE állományt. Viszont .COM fájl: TLINK/T vagy .EXE-ből az EXE2BIN programmal.

### 4. Futtatás

5. Hibakeresés → vissza az 1. ponthoz

Azaz jelen esetben:

C:\TEMP>edit elso.asm

C:\TEMP>tasm elso  
Turbo Assembler Version ...

C:\TEMP>tlink elso  
Turbo Link Version ...

C:\TEMP>elso  
Helló világ!  
C:\TEMP>\_

## Az első időszakban leggyakoribb hibák



1. Az első sorba ".model small" vagy "model small"-t írunk (két L-lel, vagy a kezdő pont nélkül; ez utóbbit MASM-nál számít, TASM-nál nem probléma).

Eredmény: fordításkor minden sorra hibát kapunk és nem keletkezik .OBJ állomány.

A régi assembler fordítók nem ismerik a memóriamodelleket, a hagyományos programszerkezetet használhatjuk (lásd később).

2. "int21h"-t vagy "int 21 h"-t írunk, azaz a paramétert egybeírjuk a mnemonikkal, vagy a h-t külön írjuk.  
Vagy az '(aposztróf)' helyett a ` jelet használjuk.  
Vagy a "start:" címkénél kihagyjuk a kettőspontot.  
Eredmény: a fordító az adott sorra szintaktikai hibát jelez és nem fordítja le a programot.

3. A program végén nem lépünk ki DOS-ba, azaz kifejejtjük a "mov ax,4C00h" és "int 21h" utasításokat.  
Ez már nem szintaktikai, hanem elvi hiba, a fordítás sikrül, futtatáskor viszont programunk "megszalad" (a gép lefagy vagy újraindul; Windows alatt a taszk rendszerszinten bezáródik - általános védelmi hiba).

## Adatdefiniáló direktívák

**Direktíva:** a fordítónak szóló, a fordítás körülményeit meghatározó utasítás, **NEM Assembly mnemonik** (mint pl. a ".model small")

A regiszterek száma véges, változóink tárolására néha nem elegendő. Megoldás: a memoriában is elhelyezhetjük, pl. az adatszegmensben.

Az erre szolgáló direktívák: DB (define byte), DW (define word), DD (define doubleword - 4 bájt) stb.

## Aritmetikai műveletek

**A négy alapművelet:**

ADD - addition, ÖSSZEADÁS: cél = cél + forrás  
 ADC - add with carry: cél = cél + forrás + CF  
 SUB - subtraction, KIVONÁS: cél = cél - forrás  
 SBB - substr. with borrow: cél = cél - forrás - CF  
 MUL - multiplication, SZORZÁS (IMUL - előjeles)  
 DIV - division, OSZTÁS (IDIV - előjeles)

**Kitérő:** egy szám -1 szerese: NEG (kettes kompl.)

### Példák:

```
.model small
.data
    x db 25           ; adatszegmens következik
    y db 'A'          ; DB = define byte (x←25)
    z db ?            ; a karakter ASCII kódja
    nullak db 10 dup (0) ; kezdőérték = ami a mem-ban van!
    tablazat db 5,6,7 ; tiz darab nulla
    nagyszam dw 8000  ; táblázatot is lehet!
    escape equ 27     ; DW = define word
    uzenet db 'Esc-vége!$' ; konstans (≈Pascal const)
    uzenet_hossza db $-uzenet ; konstans (=konstans - uzenet)
    code stb.
```

### A változókkal végzett műveletek a programból:

```
mov x,25           ; értékkopírozás (x legyen 25)
mov bh,y           ; érték lekérdezése
mov bx,y           ; hibás, mert 8/16 bitesek
mov al,escape      ; konstans használata
mov al,tablazat[2] ; FONTOS! A táblázat első
                    ; eleme a NULLADIK sorszáma!
                    ; mű! A []-en belüli szám
                    ; bájtokban léptet!
```

### Példa:

```
mov ah,5           ; AH = 5 = 0000 0101 b
not ah             ; AH = 1111 1010 b = FAh
add ah,8           ; továbbfordul és 2 lesz (!)
neg ah             ; AH = -2 (FEh)
stc                ; set carry flag (CF:=1)
adc ah,10          ; AH = -2+10+CF = 9
sub ah,4            ; AH = 9-4 = 5
mov al,-4
stc
sbb ah,al          ; AH = 5-(-4)-CF = 8
```

Assemblyben nincs "Range checking error", nem történik semmi különös, a regiszter "továbbfordul" és CF=1 lesz.

```
mov ax,2           ; új példa, szorzás
mov bh,8
mul bh             ; AX=16 (az AX-re vonatk!)
;-----
mov ax,2           ; szorzás 16 bites számmal
mov bx,8
mul bx             ; DX:AX=16 (4 bájton)
;-----
mov ax,16          ; osztás
mov bh,3
div bh             ; AL=5 hányados, AH=1 mar.
```

Az assembler a címkeket, hivatkozásokat a fordítás során számokra (offset címekre) alakítja, a gépi kódban konkrét memóriacímek szerepelnek.

Példa: a „Hello világ!” forráskódjában 3 címke szerepelt: \_data, szöveg és start. A gépi kódot disassemblerrel visszafordítva: (2 címke látható, aláhúzással jelölve)

memóriacím: gépi kód: mnemonik:

17DD:0000	B8 DE 17	MOV AX,17DE
17DD:0003	8E D8	MOV DS,AX
17DD:0005	B4 09	MOV AH,09
17DD:0007	BA 02 00	MOV DX,0002
17DD:000A	CD 21	INT 21
17DD:000C	B8 00 4C	MOV AX,4C00
17DD:000F	CD 21	INT 21

bájt fordított tárolás!

Amennyiben 1-et kívánunk hozzáadni/kivonni:

INC - inkrementálás, pl. inc al ≡ add al,1  
 DEC - dekrementálás, pl. dec al ≡ sub al,1

(A Turbo Pascalban is van INC/DEC eljárás, amely hasonló hatású.)

### Példa:

```
mov al,255
inc al             ; AL=0 lesz és ZF=1, AF=1
```

## Aritmetikai műveletek BCD kódban:

Ha két BCD kódban tárolt számmal műveletet végzünk, átvitel esetén az eredmény helytelen lesz.

**Megoldás:** a normál ADD, SUB, MUL és DIV utasítások kiadásakor még egy korrigáló utasítást is kell adni!

A végeredmény BCD kódban jelentkezik és helyes lesz. (így támogatja a CPU a BCD aritmetikát.)

A korrigáló utasítások: tömörített BCD esetén:

DAA - decimal adjust AL after addition  
DAS - decimal adjust AL after subtraction

Zónázott BCD:

AAA - ASCII adjust after addition  
AAS - ASCII adjust after subtraction  
AAM - ASCII adjust after multiplication  
AAD - ASCII adjust before division

A BCD korrigáló utasítások AL (AX) -re vonatkoznak és az aritmetikai művelet után kell kiadni (osztásnál előtte).

## Példák:

```

mov ax,0506h ; zónázott BCD, példa összeadásra
add al,08h
aaa           ; AX=0604h (56+8=64)

mov ax,0207h ; zónázott BCD, osztás
aad
mov bl,6
div bl       ; 27/6, AL=hányados=4, AH=maradék=3
mov al,72h   ; tömörített BCD, összeadás
add al,19h
daa          ; AL=91h (72+19=91)
mov al,72h   ; tömörített BCD, kivonás
sub al,19h
das          ; AL=53h (72-19=53)

```

## Példa:

Az alábbi ciklus magja hányszor fut le?

```
mov cx,3
ide:
    ;<ciklusmag>
loop ide
```

A megoldáshoz ismerni kell a LOOP működését: először eggyel csökkenti CX-et ( $CX \leftarrow CX - 1$ ) és ha nem nulla, ugrik a címkelvel jelölt helyre.

**Megoldás:** háromszor, azaz annyiszor, amennyit a CX-be elhelyeztünk ( $\approx$ Pascal FOR ciklus).

## Új példa:

```
ide:
mov cx,3
    ;<ciklusmag>
loop ide
```

Most hányszor fut le?

Az első két sort felcserélük!

**Megoldás:** a ciklus magjában CX-nek minden állandó értéket adunk - végtelen ciklust köszítettünk. VIGYÁZAT!

## Még egy példa:

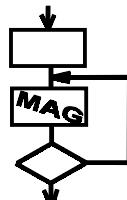
```
mov cx,0
ide:
    ;<ciklusmag>
loop ide
```

Hányszor fut le?

A megoldás alapja, hogy először csökken CX és utána történik a vizsgálat.

## Ciklusok

- Bizonyos programrészlet többszöri végrehajtása;
- A ciklusváltozó a CX regiszter;
- Hátróltesztelős, egyszer mindenképp lefut; (a Pascal FOR és REPEAT-UNTIL keveréke)
- A ciklus végének jelzése: LOOP <címke>



Emiatt az első LOOP végrehajtásakor CX átfordul 0-ból FFFFh-ra, így összesen 65536-szor hajtódik végre a ciklus magja.

További lehetőségek ( $\approx$ Pascal REPEAT-UNTIL ciklus):

LOOPZ, LOOPE - kilépés CX=0 vagy ZF=0 esetén  
LOOPNZ, LOOPNE - kilépés CX=0 vagy ZF=1 esetén

## Egymásba ágyazott ciklusok készítése:

Megoldható, csak a belső ciklusba való belépés előtt CX-et menteni kell (a verembe, más regiszterbe, vagy memória-változóba) és a végén visszaállítani. Például:

```

mov cx,5
ide1:
    ;<cx elmentése, pl. push cx>
    mov cx,3
ide2:
    ;<ciklusmag>
loop ide2
    ;<cx visszaállítása, pl. pop cx>
loop ide1

```

## Ciklus készítése elemi utasításokból:

```

...
mov al,5
vissza:
    ;<ciklus magja>
    dec al
jnz vissza

```

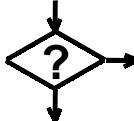
Az elemi lépésekben el-készíthetjük az ún. "fordított ciklust" is, amely-nél a ciklusváltozók változása a különleges.

A fordított ciklus készítését a régi BASIC nyelv támogatta (FOR-NEXT utasítások).

JNZ - "jump if not zero", feltételes ugrás, működését minden tárgyaljuk.

A LOOP és JNZ-nél az ugrás helye egy előjeles bájtban tárolódik, ezért max. 128 bájtot lehet hátraugrani, vagy 127-et előre (rövid - short - ugrás).

## Elágazás és ugrások



Az "if-then-else" szerkezet, megvalósítása két lépésben történik:

1. Először kiadunk egy olyan utasítást, amely valamelyik jelzőbitet (flag-et) állítja:

- aritmetikai művelet (pl. ha eredménye nulla, ZF=1)
  - regiszter léptetés (a kilépő bit CF-be kerül, később)
  - két értéket összehasonlító utasítás (CMP - compare)
- A CMP-nél nem lehet minden operandus memória változó!

2. Utána valamely jelzőbit értékétől függően ugrás történik, vagy a program a köv. utasításnál folytatódik. Azaz **Assemblyben az elágazás mindig kétirányú (case, switch)!**

A feltételes ugrások leggyakrabban használt utasításai (ugrani max. 128 bájtot hátra, vagy 127-et előre lehet, különben az "\*\*\*Error\*\* ID.ASM(6) Relative jump out of range by 004Ch bytes" hibaüzenetet kapjuk):

JZ - "jump if zero", ugrás ha a zérus jelzőbit értéke 1 (HA ARITMETIKAI VAGY LOGIKAI MŰVELET EREDMÉNYE 0, ZF=1!)

JNZ - "jump if no zero", ugrás ha ZF=0

JC - "jump if carry", ugrás ha az átvitel jelzőbit értéke 1

JNC - "jump if no carry", ugrás ha CF=0

Két operandus összehasonlítása (CMP op1, op2) után:

JA, JAE, JE, JNE, JB, JBE

"jump above / below / equal" - ugrás, ha az 1. operandus a 2.-nál nagyobb,  $\geq$ ,  $=$ ,  $\neq$ ,  $<$  vagy  $\leq$ . Valójában itt is flaget vizsgál!

Előjeles mennyiségeknél: JG, JL (greater, less), JGE stb.

## Ugrás CX=0 esetén: JCXZ <címke>

Feltétel nélküli ugrás: JMP ("jump"), pl: JMP ide

Ha a célpont más szegmensben található:

JMP <4 bájtos cím>; szegmens-offset

Ugrás egy regiszter tartalma szerinti címre: JMP BX

A ugró utasítások tulajdonképpen (CS:) IP-t állítják.

**A strukturált programnyelvekben az ugró utasítások (pl. Pascalban vagy C-ben a GOTO) ellenkeznek a nyelv filozófiájával. Assemblyben használatuk természetes, megkerülhetetlen, a nyelv szerves része!**

**Példa:** ha nem nyomunk billentyűt, a program fussen

A főprogram végtelenített hurokban fut, amelyből Esc (ASCII kódja 27) ütésére lépünk ki. Két feltételezés és egy kötelező ugrással oldjuk meg:

```

vissza:           ; végtelenített hurok kezdete
    mov ax,100h   ; van-e leütött billentyű? Ha igen, ZF=0
    int 16h
    jz tovabb     ; ha nincs, átugorjuk a beolvasást
    mov ah,8       ; bill. beolvasása (ASCII kód AL-ben)
    int 21h
    cmp al,27     ; Esc ütése esetén kiugrunk a hurokból
    je kilepes    ; "jump if equal" (vagy: jz kilepes)
tovabb:          ; ha nincs bill., vagy van, de nem az Esc
    ;<a program további része>
    jmp vissza    ; kötelező visszaugrás a hurok elejére
kilepes:         ; kilepés: mov ax,4c00h ; kilépés DOS-ba
    int 21h

```

## Regiszter léptető műveletek

Regiszterek bitjeit mozgathatjuk. Az egyik oldalon egy bit kicsúszik, a másik oldalon egy belép.

7654 3210

Aszerint, hogy a kilépő bit hová kerül (CF-be, vagy a túloldalon visszajön), a másik oldalról mi jön be (CF, a kilépő bit vagy 0) és milyen irányú a léptetés, a lehetőségek:

ROL, ROR - a kilépő bit CF-be kerül + a túloldalt visszajön

RCL, RCR - körforgás, a CF is benne van a körben

SHL, SHR - a kilépő bit CF-be kerül és 0 bit lép be

SAR - a kilépő bit CF-be kerül és nem lép be semmi (a két legnagyobb bit megegyezik - előjell!)

rotate left/right shift left/right
rotate through carry left/right shift arithmetic right (SAL=SHL)

## Felhasználása:

Ilyen van a C-ben és a Pascalban is!

Pl. C-ben: int i,j; i=9;  
 j=i<<2; /\* j=36 \*/  
 j=i>>2; /\* j=2 \*/

• A balra léptetés 2 hatványaival való szorzást jelent, a jobbra léptetés osztást (ha a szám páratlan - a kilépő bit miatt CF=1 lesz; gyorsabb, mint a MUL/DIV);

• Egy regiszter tetszőleges bitjének értékét lekérdezhetjük (a CF-be elforgatással és feltételes ugrással).

A lépésszám lehet 1 vagy CL, 80286-tól 0..31 is.

Például: mov cl,5 és rol ah,cl

A 80386+ CPU-tól közvetlen bitmanipulációs utasítások is vannak (bit lekérdezése, állítása 0/1-be, ellentétesre).

### Példa: AL=?

```

mov al,5      ; kezdőérték
rol al,1      ; *2, AL=10 és CF=0
mov cl,2
shr al,cl     ; osztás 4-gyel, AL=2, CF=1
shl al,1      ; nulla lép be, AL=4, CF=0
stc           ; "set carry flag", CF=1
rcl al,1      ; AL=9 (CF=1 volt), CF=0
neg al        ; kettes kompl, AL=-9=F7h
sub al,2      ; AL=-11=F5h
sar al,1      ; ELŐJELHELYES, AL=-6, CF=1

```

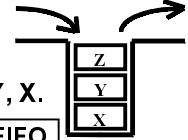
### A verem (stack) és a szubrutinok

A verem a RAM memóriában, a stack szegmensben ( $\rightarrow$  SS regiszter) található különleges szervezésű tár.

Elméleti működése: LIFO ("Last In First Out"):

Példa: a verembe 3 adatot helyezek  
el: X, Y, Z.

A kivétel sorrendje fordított: Z, Y, X.



A LIFO mellett gyakran használatos a FIFO (cső) szervezésű tár, pl. az Intel CPU-knál a memóriatartalom beolvasásának gyorsítására.

### Logikai műveletek

Az utasítások bitenkénti logikai műveletet hajtanak végre, igazságtáblázatuk alapján.

Az AND utasítással egy regiszter tetszőleges bitjeit nullázhatjuk. Más neve MASZKOLÁS. Például:

```

mov ah,eredmeny ; mem-beli változó
and ah,00001111b ; maszkolás

```

Hatása: AH felső 4 bitje törlődik (0 lesz), alsó 4 bitje változatlan marad (mert ott a maszk 1 volt).

Az OR UTASÍTÁSSAL egy regiszter tetszőleges bitjeit 1-be állíthatjuk. Például:

```

mov ah,eredmeny
or ah,00001111b

```

A felső 4 bit változatlan marad, az alsó 4 bit 1 lesz.

### A XOR UTASÍTÁS

XOR= eXclusive OR - kizáró vagy (hasonló az OR-hoz, csak 1 és 1 esetén 0-t ad eredményül).

Leggyakoribb használata:

#### • Regiszter nullázása, pl:

```
xor ax,ax
```

Egy bájttal rövidebb a 3 bájtos `mov ax,0`-nál.

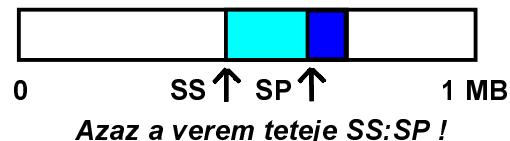
- Az igazságtáblázata miatt kétszeri XOR után viszszakapjuk az eredeti értéket (pl. képernyő "felett" alakzat mozgatása, vagy titkosítás: a bájtot egy adott számmal XOR-olva tárolom el, a visszakódolás egy újabb XOR-ral történik).

A TEST utasítás megegyezik az AND-dal, de csak a jelzőbiteket állítja. Végül NOT-egyes komplement.

Gyakorlati megvalósítása: a verem vége rögzített (általában megegyezik az SS szegmens végével), teteje viszont állandóan változik, egy regiszter

SP (Stack Pointer - veremmutató)

jelzi. A verem visszafelé nő, azaz adat behelyezésekor SP csökken, kivételkor pedig növekszik.



Azaz a verem teteje SS:SP !

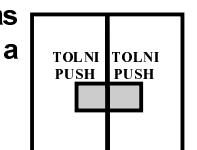
A verem felhasználása:

1. adatok ideiglenes tárolására
2. szubrutin visszatérési cím tárolására

#### 1. Adatok ideiglenes tárolása:

Ha egy regiszter értékét ideiglenesen felülírjuk, az eredeti tartalmat szokás a verembe menteni (más megoldás a memória kijelölt részében tárolás).

A veremre vonatkozó utasítások a



PUSH és a POP.

#### Példa:

```

PUSH AX          ; 16 bites regiszter
PUSH BX          ; mentése a verembe
; <program további része>
POP BX           ; visszatöltés
POP AX

```

Nem árt, ha annyi POP van, amennyi PUSH (azaz ne hagyunk a veremben felesleges adatot és ne vegyük ki többet, mint amennyit betettünk!).

Visszatöltésnél fontos a sorrend, különben a regiszterek értékei felcserélődnek (néha ez is cél!).

Egy PUSH-nál először SP 2-vel csökken és utána tárolódik a 2 bájtos érték. POP-nál fordított: az érték kivétele után nő SP 2-vel.

Megjegyzés: Az Intel CPU-k visszafelé kompatibilitása nem 100%-os, mert pl. PUSH SP, POP AX után AX-ben más érték lesz Intel 8086/8088-as, és 80286-os (vagy jobb) CPU-val felszerelt gépen. A PUSH SP lecsökkenti SP-t és a verembe helyezi (SP-t), de 80286-ostól ez az - egyébként ritkán használt - utasítás mégis SP eredeti értékét helyezi a verembe. Az eltérés felhasználható számítógépünk típusának megállapítására.

### Az előző példa Assembly nyelven:

```
call szub      ; szubrutinhívás
...
call szub      ; még egyszer meghívom
mov ax,4c00h   ; kilépés DOS-ba
int 21h

szub proc     ; szubrutin kezdete
...
ret           ; visszatérés
szub endp    ; szubrutin vége
```

Más lehetőség az általános regiszterek mentése a verembe (SP értéke 16-tal változik):

PUSHA - AX, BX, CX, DX, SI, DI, BP, SP mentése  
POPA - ezek visszatöltése CSAK 80286-tól! (.286)

A Flag regiszter mentése: PUSHF, POPF

Assembly direktíva: a fordításkor várható 80286-os utasítás is.

Hol tárolódik a visszatérési cím? A veremben!  
A CALL működése: PUSH IP és IP ← cél offset.  
Ha szegmensen túli az ugrás: CALL FAR, ekkor mindenekelőtt PUSH CS és CS ← célszegmens.

### A PUSH/POP és CALL/RET UGYANAZT A VERMET HASZNÁLJA!

Emiatt így ne adjunk át adatot a push bx  
szubrutinnak (a program lefagy,  
mert a POP kiveszi a visszatérési  
címét)! Megoldás: regiszter, me-  
moriaváltozó, vagy BP használata.

```
-----  
push bx  
call szub  
-----  
pop bx  
...  
ret
```

A veremnek csak a tetejéhez férhetünk hozzá. Ha egy belső elem értékére van szükségünk (ki nem vehetjük), használhatjuk a

#### BP (Base Pointer - bázismutató)

regisztert. Ilyenkor a MOV utasítás nem a szokásos DS, hanem az SS szegmensre vonatkozik.

Például:

```
mov ax,[bp]    ; SS:BP-től 2 bájt!
mov al,[bp-4]
```

Példa: adatátvétel a verem keresztül a BP reg-rel:

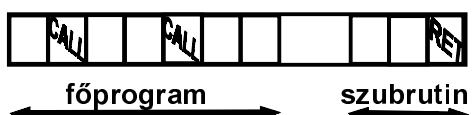
```
mov dl,'A'      ; --- FŐPROGRAM ---
push dx         ; adatátadás
call szub       ; szubrutin hívása
pop dx          ; a feleslegessé vált adat törlése
                ; (vagy visszajelző adat átvétele)
int 20h         ; kilépés DOS-ba

szub proc near ; --- SZUBRUTIN ---
mov bp,sp        ; adatátvétel 2 lépésekben
mov dx,[bp+2]    ; +2 csak NEAR szubrutin esetében!
; <műveletek az átvett adattal>
ret              ; visszatérés a CALL utáni utas-ra
szub endp        ; szubrutin vége
```

### 2. Szubrutinhívásnál:

Szubrutin: a program futása során egy programrészletet többször, különböző helyről meghívunk (akkor paraméterátadással is). Hasonlít a Pascal procedure - eljárás - ra.

A szubrutint a program tetszőleges részén a CALL (hívás) utasítással hívjuk meg. A szubrutin végén RET (return, visszatérés) áll. Példa: ugyanazt a szubrutint 2-szer hívjuk:

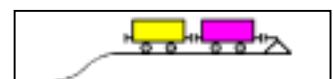


A CALL hatására a verembe a CALL utáni utasítás offset címe kerül, a RET ezt veszi vissza, ide ugrat.

A visszatérési cím veremben tárolása következtében többszörös mélységen is hívhatunk szubrutinokat (az egyetlen korlát a verem mérete).

Példa a verem használatára a számítástechnikán kívül:

vasúti holtvágány



Jól gondoljuk meg előre a verem használatát, mert nincs ele-nőrzés és ha túlcordul ("Stack overflow" - különösen .COM fájl esetén könnyen megeshet), felülírhatja programunkat!

## Megszakítások (Interrupt)

Megszakítás: az aktuális folyamatot a CPU felfüggeszti, új tevékenység elvégzése (ugrás egy szubrutinra, "a megszakítás kiszolgálása"), majd visszatérés és a program folytatása

- A hardver eszközök és a mikroprocesszor megszakításokon keresztül kommunikálnak egymással (pl. billentyűzet, egér, HDD, FDD stb.);
- Más, mint a CALL, mert annak helyét és idejét mi határozuk meg a programban, nem váratlanul jön;
- Vannak HW / SW, külső / belső megszakítások.

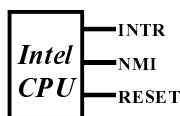
### A megszakítások fajtái prioritás szerint:

#### "1." RESET (nyomógomb)

Az utasítás végrehajtása közben is félbeszakítja a processzor működését és ugrik (hidegindítás). A CPU alaphelyzetbe áll.

#### 2. NMI

*Non Maskable Interrupt*, nem maszkolható (nem tiltható) megszakítás: utasítás befejezése, regiszterek mentése és ugrás. Általában kritikus eseménynél használják (pl. áramkimaradásnál a RAM mentése háttértárra). Az IBM



### Egy megszakítás feldolgozása:

```

PUSHF      ; Flag reg. mentése a verembe
IF ← 0     ; további hw megsz. tiltása (!)
            ; Azaz az Intel CPU-knál egyszintű!
TF ← 0     ; egylépéses üzemmód tiltása
PUSH CS
PUSH IP
IP ← megfelelő érték, láasd később
CS ← megfelelő érték
  
```

Variációk: INTO (ha OF=1 akkor INT 4); INT (=INT 3)

A megszakítás végén IRET áll. Hatása:

```
POP IP, POP CS, POPF.
```

PC-ken RAM, I/O paritáshiba, vagy a coprocesszor hatására keletkezik NMI.

Az alaplapok egy része nincs RAM paritásvédelemmel elátva (vagy paritásgenerátor tartalmaz) → nem lesz NMI.

#### Az NMI letiltása: (érdekesség)

```

in al,70h          ; olvasás CMOS portról
or al,10000000b   ; a 7. bit 1-be állítása
out 70h,al         ; visszairás
in al,70h          ; zavar elkerülése
  
```

NMI bekövetkezésének egyik oka lehet, ha a gépben RAM-ot cserélünk, amely *lassabb* a szükségesnél.

NMI bekövetkeztekor a gép kiírja:

"RAM parity error at xxxx:yyyy" és leáll.

#### 3. IRQ (INTR bemenet)

*Interrupt ReQuest*, megszakítás kérés: hardver megszakítás, külső eszközök kérésére. Csak akkor érvényesül, ha a megszakítás jelzőbit engedélyezi (IF=1; ez a jelzőbit nem vonatkozik az NMI-re).

#### 4. INT

Szoftver megszakítás: Assembly utasítás (INT 0..255), amellyel a program tetszőleges helyén rátugorhatunk a megszakítást kiszolgáló rutinra (amit a hardver egység hív IRQ-val, vagy más által megírt programra, pl. a ROM-BIOS-ban).

Az IBM PC gépeken több tíz megszakítást előre megírtak és a ROM-ban (BIOS), valamint MS-DOS esetén az IO.SYS és MSDOS.SYS állományokban helyeztek el. Ezek a megszakítások bármely DOS-os gép esetén rendelkezésre állnak. Így az Assembly programokban nem kell megírni egy-egy bonyolultabb műveletet, elég meghívni a megfelelő megszakítást.

Csak a legalsó szintű megszakítások vannak a ROM-ban, a többi cserélhető, op. rendszer függő (Windows, UNIX stb.)!

A megszakítások: billentyű-, képernyő-, egér- és fájlkezelés, időzítés stb. Egyszóval szinte minden.

#### Példák: (néhány DOS hívás, DOS függvény ← INT 21h)

##### • karakter kiírása a képernyőre:

```

mov ah,2
mov dl,'W'      ; kiíratandó karakter
int 21h
  
```

##### • szöveg kiírása a képernyőre: Pascalban egyszerűbb:

```

mov ah,9
mov dx,offset szoveg
int 21h          ; DS:DX-től '$'-ig tart
  
```

##### • karakter bekérése és kiírása:

```

mov ah,1
int 21h          ; ASCII kód AL-be kerül
  
```

##### • csendes bekérés (nem kerül kiírásra):

```

mov ah,8
int 21h          ; ASCII kód AL-be kerül
  
```

##### • van-e lenyomott billentyű? (ROM-BIOS hívás!)

```

mov ax,100h
int 16h          ; ha van, ZF=0 lesz
  
```

Pascal: READKEY

Pascal: KEYPRESSED

## • kilépés DOS-ba:

```
mov ah,4ch
mov al,kilépés_kódja
int 21h
```

DOS batch fájlból  
lekérdezhető az  
errorlevel-lel!

Csak .COM esetén, még egy "elavultabb" lehetőség: int 20h.

## • képernyőtörles (és 80\*25 szöveges üzemmód)

```
mov ax,3
int 10h ; ROM-BIOS hívás
```

DOS: MODE co80

A megszakítások leírását kézikönyvek tartalmazzák. Vannak ún. nem publikált hívások is.

INT 0Eh - IRQ 6, floppy disk kontroller, parancs befejezését jelzi  
 INT 10h - video szolgáltatások (felbontás beállítása, kiírás stb.)  
 INT 13h - disk I/O: szektor írás/olvasás/ellenőrzés, sáv formázása  
 INT 15h - AT kiterjesztett szolgáltatások (védett módba átkapcsolás stb.)  
 INT 16h - keyboard I/O  
 INT 19h - rendszer újraindítása  
 INT 1Bh - Ctrl+Break leütése esetén  
 INT 1Ch - user timer interrupt: másodpercenként átlagosan 18,2-szer hajtódik végre az órajelre (55 ms-onként), kezdetben egy IRET-re mutat. Az INT 8 hívja meg.  
 INT 20h-2Fh - DOS megszakítások: a kiszolgáló rutin nem a ROM-BIOSban található, hanem a RAM memoriában és bootoláskor az MSDOS.SYS és az IO.SYS tartalmazza (e fájlok a ROM bővítésének tekintetében)  
 INT 33h - egérkezelő függvények: MOUSE.COM stb. tartalmazza a kiszolgáló programot  
 INT 67h - Expanded Memory kezelő függvények

## Megjegyzések:

- A kész megszakításokat nem kötelező felhasználni (írhatunk helyette újat), viszont ha mégis, akkor alkalmazkodnunk kell hozzájuk. Pl. a karakter kiíró rutin pont a DL-ből ír ki és nem másból stb;
- Egy-egy INT megszakítás több ún. "alszolgáltatást" tartalmaz, hogy melyiket hívjuk meg, AH-ban kell a hívás előtt megadni;
- A megszakítások el/méletileg minden regiszter eredeti értékét megőrzik (ha mégsem, a hívás előtt mentsük - PUSH, PUSHA);

? Honnan tudja a processzor egy-egy megszakításnál,  
hogy melyik memóriacímen található a kiszolgáló rutin?

A választ az ún. megszakítási vektortáblázat adja:

A RAM memória legelső 1 KB-ja tartalmazza azokat a 4 bájtos szegmens-offset címeket, amelyek a megszakítást kiszolgáló rutin kezdőcíméit mutatják:  
0-1-2-3. bájt = INT 0, ... egészen INT 255-ig.

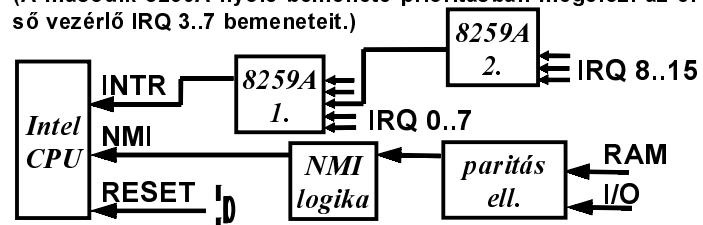
Ezek az értékek átírhatók, így saját megszakításokat is készíthetünk. A rezidens programok (segédprogramok, driverek stb.) és a vírusok működésének alapelve az egyes megszakítások átirányítása.

A PC-kben a HW és SW megszakítások "keverednek" egymással, pl. az NMI-t feldolgozó rutin hívása: INT 2.

- A megszakítások megértésekor lehet látni jól, hogy mi hardver és mi operációs rendszer függő. Az IBM PC-kben a megszakítások (kiszolgálás vektortáblázat alapján, NMI stb.) hardver megoldás, bármely szoftver esetén ugyanúgy működnek;
- Ugyanezen ok miatt nem lehet pl. UNIX alatt futtatni DOS-os programokat - a megszakításokat más rendszer másképp dolgozza fel! A DOS szimuláló környezetekben (pl. Windows NT Command prompt) csak néhány megszakítás kiszolgálása van megírva;
- A ROM tartalma gyártótól függ, a megszakításokat feldolgozó program más és más lehet. Emiatt ajánlatos a ROM tartalmát csak a megszakításokon keresztül elérni, ezek biztosítják a szabványos kapcsolódási felületet;
- A melegindítás (Ctrl+Alt+Delete) nem törli a vektortáblázatot (→ a vírusok megmaradhatnak)!

## A megszakításokat feldolgozó áramkör (elvi rajz):

A két kaszkádba kötött interrupt vezérlő összesen 16 hardver megszakítást tud fogadni. A második kimenete az első IRQ 2 bemenetére van kötve (és IRQ 9 szoftveresen át van irányítva IRQ 2-re). A legnagyobb prioritású az NMI, a legkisebb az IRQ 7. (A második 8259A nyolc bemenete prioritásban megelőzi az előző vezérlő IRQ 3.7 bemeneteit.)



## Az Assembly program gyorsasága

A mikroprocesszor leggyorsabban a flip-flopokból (R-S tároló) álló regisztereivel képes műveleteket végezni.

A RAM memória elérése (változók vagy veremműveletek esetén) jóval több időt igényel.

Az állandó háttértár (merev., floppy) még többet.

Tehát melyik a gyorsabb: MOV AX,BX vagy PUSH AX ?

## Néhány megszakítás:

- INT 0 - osztási túlcsordulás DIV vagy IDIV utasításnál (pl. ha nullával osztunk) - ún. "belső megszakítás" v. eltérülés (a CPU okozza)
- INT 1 - ha TF=1, minden utasítás után egy INT 1 is következik. Mogszakításoknál TF=0 miatt nem működik. Nyomkövetésre használható.
- INT 2 - NMI
- INT 3 - töréspont elhelyezése programban (egy bájtos utasítás)
- INT 4 - az INTO utasításnál OF=1 esetén INT 4 generálódik
- INT 5 - PrintScreen megnyomására nyomat (INT 9 hívja meg), indexhatár túllépés (80286+)
- INT 6 - illegális utasítás végrehajtási kísérlete (80286+)
- INT 7 - ha nincs matematikai coprocesszor, egy NPU utasításnál ide ugrik (80286+)
- INT 8 - IRQ 0, időzítő: másodpercenként 18,2-szer aktivizálódik
- INT 9 - IRQ 1, billentyűzet: lenyomáskor / felengedéskor keletkezik (bill. kódot állít elő, kezeli a Ctrl+Alt+Del, SYSRQ és PAUSE gombokat)

## Egy példaprogram

A program kiír a képernyőre egy "A" betűt:

```
title egy betu kiirasa
.model small      ; kis memóriamodell (.EXE fájl)
.stack           ; 1 KB veremszegmens létrehozása
.data            ; adatszegmens (most üres)
.code             ; kódszegmens
.start:          ; --- KEZDÖDIK A PROGRAM ---
    mov ah,2        ; kar. kiírása DOS megszak-sal
    mov dl,'A'
    int 21h
    mov ax,4C00h    ; kilépés DOS-ba hibaj. nélkül
    int 21h
.end start       ; prg vége, belépési pont "start"
```

Bővítés: az "A" után egy "B"-t is írjon ki!

```
... start:
    mov ah,2
    mov dl,'A'
    int 21h
    mov dl,'B'      ; két új sor: a 2. kar. kiírása
    int 21h ...     ; az AH=2 -t már megadtuk!
```

Újabb bővítés: 5-ször írjuk ki az "AB"-t!

```
... start:
    mov ah,2          ; értékadás a cikluson kívül!
    mov cx,5          ; a ciklus 5-ször hajtódjon végre
.ide:
    mov dl,'A'
    int 21h
    mov dl,'B'
    int 21h
.loop ide ...      ; ciklus vége
```

- Polimorf vírusok: a kódba véletlenszerűen NOP-okat generál, kódösszehasonlító víruskereső nem ismeri fel.

### HLT - HaLT

A processzor megáll, csak Reset vagy külső hardver megszakításra indul tovább (a megszakítás lekezelése után).

### WAIT

Várakozás a TEST (BUSY) vonalak aktív állapotára, a co-processzor miatt használják.

### LOCK

Prefixum. A következő utasítás idejére lezárja az adatbuszt. A processzor nem fogadhat addig külső utasításokat. Többprocesszoros esetben használható.

Pl. lock mov [di], al

### LAHF / SAHF - a flag regiszter AH-ba tölti / fordítva

### Táblázatok használatánál előnyös:

XLAT - jelentése: MOV AL, DS:[BX+AL]

### Szám "kiterjesztése" az előjel megtartásával:

CBW - convert byte to word AX ← AL (pl. FEh → FFFEh)

CWD - convert word to doubleword DX:AX ← AX (FFFF:FFFEh)

### Adatcsere portokon keresztül:

```
out dx,al      ; 1 bájt kiküldése
out dx,ax      ; 2 bájt
in al,dx       ; 1 bájt beolvasása
in ax,dx       ; 2 bájt
```

Az IN/OUT utasításokkal tartja a kapcsolatot a CPU más "intelligens" egységekkel, mint pl. a DMA, időzítő, billentyűzet, vagy a CMOS stb.

DMA stb. olvasásakor a cím ki OUT-olása után egy kis időt várni kell, amíg az érték beolvasható IN-nel!

## Egyéb utasítások

XCHG: két regiszter értékének felcserélése, pl:

```
mov cx,1234h
mov dx,5678h
xchg cx,dx
```

(Más megoldás: a verem segítségével.)

### LEA - Load Effective Address

Hatása megegyezik a speciális MOV-val:

Pl. lea dx,szoveg ≡ mov dx,offset szoveg

### Stringkezelő utasítások:

A memóriában egy utasítással blokkokat mozgathatunk, vagy egy bájtot kereshetünk meg. Összetett, tipikusan CISC regiszterekre jellemző utasítások.

MOVSB - másolás ES:[DI]←DS:[SI] és SI / DI vált.

LODSB - betöltés AL←DS:[SI] és SI változtatása

STOSB - tárolás ES:[DI]←AL és DI változtatása

CMPSB - összehas. DS:[SI] és ES:[DI] (flag-ek!)

SCASB - összehas. AL-t és ES:[DI] -t (flag-ek!)

### NOP - No Operation

Üres utasítás, nem történik semmi (csak az idő telik).

Felhasználása:

- Időzítés\* (LOOP-pal együtt, a CPU órajelétől függ!);
- Lefordított program módosítása: a nem kívánt részeket NOP-pal felülírjuk. Egy NOP 1 bájtot foglal el, így könnyű programrészleteket törölni.

(A futtatható fájlból nem lehet csak úgy kitörölni a felesleges bájtokat, mert a gépi kód konkret memóriacímeket tartalmaz → törlés után eltölődnak!)

\*Az időhúzásra még egy trükk: JMP \$+2 !!!

Ha B helyett W áll - 2 bájt mozog és AL helyett AX áll (MOVSW, LODSW stb.).

REP - prefix, a köv. utasítást CX-szer hajtja végre

Pl. REP MOVSB

REPE/REPZ - ugyanez, csak kilép ZF=0 esetén

Az adatmozgás irányát az irányjelző bit (direction flag) jelzi: DF=0 → SI / DI nő, DF=1 → csökken.

### A flagek közvetlen állítása:

- carry flag: STC, CLC, CMC (set, clear, complement)
- interrupt flag: STI, CLI
- irányjelző flag: STD, CLD

**Fontos:** a MOV utasítás nem állítja a flageket!

A Windows 3.x/9x végleges lefagyásztára 3 bájtos .COM fájlal:

```
250          cli
235 254     ide: jmp ide
A Windows NT-nél nem működik.
```

### Példa a Macro/Turbo ASM-nél használható direktívákra:

```
ter dw 100 dup (?) ; 100 szó foglalása
Ekkor length ter=100, type ter=2, size ter=200
```

### Intel Pentium MMX (MultiMedia eXtension): 57 új utasítás

### 3. lépés:

A lényeg kivételével könnyű megírni programunkat:

```
title kiir program
.model small
.stack
.data
.uzenet db 'Üssön le billentyuket, Esc-vege!',10,13,'$'
.code
start:
    mov ax,3           ; képernyőtörölés
    int 10h
    mov ax,_data       ; DS ráállítása az adatszegmensre
    mov ds,ax
    mov ah,9           ; üzenet kiírása
    mov dx,offset uzenet
    int 21h
    ; --- IDE KERÜL A PROGRAM LÉNYEGE! ---
    kilepes: mov ax,4c00h      ; kilépés DOS-ba hibaj. nélkül
              int 21h
end start
```

10, 13 = soraelés (CR, LF)



## III. ASSEMBLY PROGRAMOZÁS

### Egyszerű feladat megoldási lépései

1. Programozásilag hogy oldom meg, eszközök ki-választása.
2. A feladat részekre bontása.
3. Az egyes részeknek Assembly programrészleteket feleltetünk meg, változók és regiszterek ki-választása.
4. Program megírása (.ASM), fordítás, futtatás, hibakeresés.

### Példa:

Írunk Assembly programot, amely letörli a képernyőt, kiírja az "Üssön le billentyűt, Esc-vége!" üzenetet és a leütött billentyűket úgy jeleníti meg, hogy "a" helyett "b"-t ír ki, "b" helyett "c"-t stb. A program Esc leütésére érjen véget!

```
Üssön le bill, Esc-vége!
BCD
C:>_
```

### 4. lépés (+ 1. lépés is):

Bárhogyan húzzuk, halasztjuk, most érünk el ahhoz a ponthoz, amikor el kell gondolkoznunk azon, hogy ezt a problémát ténylegesen hogyan oldjuk meg???



A probléma programnyelv-független algoritmuskészítés, nem Assembly jellegű.

Megoldás: van egy megszakítás, amely beolvashat a billentyűt és van egy másik, amely kiír egy karaktert. A billentyű ASCII kódban jelenik meg, EZT KELL MEGNÖVELNI! Azaz a kiírás nélküli karakterbekérést kell használnunk.

Az előzőleg megírt programunknál a kihagyott "lényeg"-es rész (most éppen Assembly nyelven megírva):

```
vissza:      ; végtelenített hurok
    mov ah,8   ; bill. csendes bekérése (AL-be)
    int 21h
    cmp al,27 ; Esc? Ha igen => kilépés DOS-ba
    je kilepes ; "jz kilepes" is helyes
    mov ah,2   ; karakter kiírása következik
    mov dl,al  ; mert a kiírás DL-ból történik
    inc dl    ; EZ A LEGFONTOSABB SOR !!!
    int 21h
    jmp vissza ; kötelező visszaugrás
```

### Megoldás:

1. lépés: egyelőre hagyjuk későbbre!
2. lépés: a program négy, jól elkülöníthető részből áll:

- képernyőtörölés
- üzenet kiírása
- billentyű bekérése és kiírása, A LÉNYEG!
- kilépés DOS-ba

### Bonyolítás:

A program a 10. billentyű, vagy Esc ütésére érjen véget (amelyik előbb bekövetkezik).

### Mi lehet a megoldás?

Végtelenített hurok helyett egy 10-szer lefutó ciklust készítünk. Így a 10. gomb leütésére a ciklus ér véget, Esc esetén pedig kiugrunk a belséjéből!

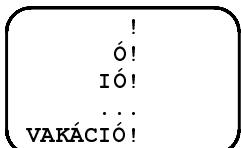
## A program "lényeg"-e átírva:

```
mov cx,10      ; ciklusváltozónak értékkadás  
vissza:       ; visszaugrás hely  
    mov ah,8  
    int 21h  
    cmp al,27  
    je kilepes  
    mov ah,2  
    mov dl,al  
    inc dl  
    int 21h  
loop vissza ; ciklus vége
```

Egy egyszerűbb Assembly programot érdemes teljesen megértenünk. Ez akkor sikerült, ha minden soráról tudjuk, mi a feladata, miért szerepel, hogy működik és mi történne, ha nem szerepelne!

## Példa:

Írunk Assembly programot, amely letörlí a képernyőt és a "VAKÁCIÓ!" szót a következőképpen írja ki a képernyőre:



A szöveg csak egyszer legyen eltárolva a programban, amit úgy készítünk el, hogy egyszerűen lehessen tetszőleges hosszúságú szöveget kiíratni!

**1. lépés:** megoldás módjának meghatározása. Ez a legfontosabb pont! Szintén programnyelv-független probléma a kiírás megszervezése.

**Első gondolat:** minden sort külön eltárolok (szöveg db ...) és egymás után kiírom a képernyőre.  
Így lehet, de nem túlzottan szép! Ne így csináljuk!

**Második ötlet:** a kiíratandó sort egyszer tárolom le, többször kiírom és utána más-más darabszámú szóközt (space) írok a szövegre rá (a képernyőn).

Így már jobb, de a szóközök kiírása bonyolult.

Ne így csináljuk!

**Harmadik ötlet:** legyen még egy string, amely kezdetben üres és hátulról 1-1 betűt átmásolok:



A "kiírni" stringet fogjuk nyolcszor kiírni a képernyőre, a "szöveg"-ből csak a betűket másolom át!

Valósítsuk meg ezt a megoldást!

## 2. lépés: részekre bontás

A program négy jól elkülöníthető részre tagolható:

1. rész: regiszterek beállítása (pl. adatszegmens)
2. rész: képernyőtörles
3. rész: kiíratás CIKLUS segítségével, a LÉNYEG!
4. rész: kilépés DOS-ba

## 3. lépés: Assembly programrészletek

A regiszterek kiválasztása jön: melyiket milyen célra használjuk fel?

Így mutasson a "szöveg"-re, DI "kiírni"-ra! Kezdőértekként mindenki a legutolsó betűre mutasson, s majd csökkentjük!

Tehát: 

```
lea si,szoveg  
add si,hossz-1  
lea di,kiirni  
add di,hossz-1
```

DX mutasson minden a "kiírni" elejére, mert az INT 21h megszakítás DX offset címtől kezdi a kiírást.

Tehát: 

```
lea dx,kiirni
```

A képernyőtörles: `mov ax,3` és `int 10h`

**A ciklusváltozó CX legyen (ahány betűs "szöveg").**

**A ciklus:**

```
mov cx,hossz  
vissza:  
    mov al,[si] ; betű másolása  
    mov [di],al  
    int 21h      ; (AH=9 előzőleg)  
    dec si        ; léptetés balra  
    dec di  
loop vissza
```

Kilépés DOS-ba: `mov ax,4c00h` és `int 21h`

## 4. lépés: program megírása

```
TITLE vakacio  
.MODEL SMALL  
.STACK  
.DATA  
    szoveg db 'VAKÁCIÓ!'  
    hossz equ $-szoveg ; azaz a programban hossz=8  
    kiirni db hossz dup(' '),10,13,'$' ; sörémelessel  
.CODE  
start:  
    mov ax,_data  
    mov ds,ax  
    mov si,offset szoveg  
    add si,hossz-1  
    mov di,offset kiirni  
    add di,hossz-1  
    lea dx,kiirni ; megegyezik: "mov si,offset kiirni"
```

```

mov ax,3      ; képernyőtörles
int 10h
mov ah,9      ; szöveg kiíratásának előkész.
mov cx,hossz ; ciklus
vissza:
  mov al,[si]
  mov [di],al
  int 21h      ; kiíratás
  dec si
  dec di
loop vissza   ; ciklus vége
mov ax,4c00h   ; kilépés DOS-ba
int 21h
end start     ; program vége

```

### Példa: BX tartalmát írassuk ki hexában (16 biten)!

Mint előbb, csak 16 bit=4 hexa számjegy → 2 ciklust készítünk. A belső ciklus BX-ból 4 bitet mozgat DL-be, kiíratjuk. Ha DL>9, 'A'..'F'-et kell kiírni, ezért DL-t növeljük 7-tel (ASCII kódok: '0'=48→'9' utáni=58 és 'A'=65; 65-58=7!).

```

mov ah,2      | loop ciklus2
mov cx,4      |   pop cx
ciklus1:      |   add dl,'0'
               |   xor dl,dl
               |   push cx
               |   mov cx,4
               |   ciklus2:
               |   rcl bx,1
               |   rcl dl,1
               |   tovabb:
               |   int 21h
               |   loop ciklus1

```

### Ugyanez a feladat, de a végeredmény .COM fájl:

```

title vakacio    ; hagyományos programszerkezettel!
progi SEGMENT   ; szegmens definiálása
ASSUME cs:progi, ds:progi, ss:progi, es:progi
ORG 100h         ; a 100h offset címtől fordítson
start:
  lea si,szoveg  ; megegyezik mov si,offset szoveg-gel
  add si,hossz-1
<stb. ezek a sorok változatlanok, "loop vissza"-ig>
  int 20h        ; kilépés DOS-ba .COM fájl esetén
  ; (mov ax,4c00h és int 21h is használható)
szoveg db 'NYÁRI SZÜNET!' ; minden egyéb a program végén
hossz equ $-szoveg
kiirni db hossz dup(' '),10,13,'$'
progi ENDS       ; szegmens vége
END start        ; program vége

```

### Programok felépítése

A futtatható fájlok kiterjesztése .COM, .EXE és .BAT lehet ("CEB" - azonos nevek esetén a futtatási sorrend). A .BAT-tól most tekintsünk el, mert nem gépi kód.

A szegmentálási technika miatt, valamint mivel a JMP és CALL utasítások relatív címet tartalmaznak, a futtatható program a memória bármelyik szegmensébe betölthető. Azaz a program készítésekor nem feltételezhetünk semmilyen betöltési címet, a program oda töltődik, ahol van hely (a rezidens programok fölé) - kivétel pl. a bootolás.

### Még egy megoldás:

```

TITLE vakacio    MEMÓRIAMODELL .COM FÁJLOK LÉTREHOZÁSÁRA!
.MODEL TINY
.CODE
.org 100h
start:
  lea si,szoveg
  add si,hossz-1
  mov di,offset kiirni
  add di,hossz-1
  lea dx,kiirni
  mov ax,3
  int 10h
  mov ah,9
  mov cx,hossz
  vissza:
    mov al,[si]
    mov [di],al
    int 21h
    dec si
    dec di
  loop vissza
  int 20h
  szoveg db 'Gábor Dénes Főiskola'
  hossz equ $-szoveg
  kiirni db hossz dup(' '),10,13,'$'
END start

```

A memóriamodellekörön (Assembly, C nyelv stb.)  
 TINY - a kód+adatok+verem elfér 64 KB-ban, minden mutató near típusú. Az így elkészített program .COM fájlba szerkeszthető.  
 SMALL - a kód max. 64 KB lehet, az adatok+verem másik 64 KB. minden mutató near típusú, .EXE (speciálisan .COM) állomány készíthető.  
 MEDIUM, COMPACT, LARGE, HUGE - 64 KB helyett 1 MB, .EXE fájl készíthető.

### A .COM állományokról

- Csak a gépi kódot tartalmazza, semmi mást;
- Egy szegmensnyi, tehát mérete max. 64 KB lehet (kód+adat+verem);
- A szabad szegmens első 100h bájtja az ún. *Program Szegmens Prefix* - PSP, itt tárolódik pl. a program indításakor megadott paraméterek (DTA, I. később), ennek tartalmát elkezíti a DOS, és az ezutáni területre tölti be a fájlt;
- CS, DS, ES és SS a szegmensre mutat, SP=FFF Eh a szegmens végére áll és az IP=100h címre adódik át a vezérlés (a veremben az FFFEh és FFFFh helyekre 0 kerül; ha kevesebb az igénybe vehető memória, SP más értéket vesz fel!);

Kivételek minden vannak,  
pl. a Windows 95/98/Me  
COMMAND.COM-ja 93 KB-os,  
de ez valójában egy .EXE fájl.

### Példa: BX tartalmát írassuk ki binárisan (16 biten)!

A megoldás lényege: két RCL utasítással BX legfelső bitjét DL-be forgatjuk, a számóból karaktert készítünk és kiíratjuk. Mivel AX 16 bites → ciklusba foglaljuk az egészet. Tehát:

```

mov ah,2      ; karakter kiíratás előkészít.
mov cx,16      ; ciklus, 16-szor
ciklus:
  xor dl,dl    ; DL nullázása
  rcl bx,1      ; BX legfelső bitjét a CF se-
  rcl dl,1      ; gítségével átvisszük DL-be
  add dl,'0'    ; 0..9 → '0'..'9'
  int 21h      ; a karakter kiíratása
loop ciklus   ; ciklus vége

```

(Ez és a következő „igazi” Assembly jellegű feladat.)

### .COM állomány készítése Turbo Assembler esetén:

C:\>tasm <prgnév> és C:\>tlink/t <prgnév>

(Vagy: lefordítjuk .EXE-re és az EXE2BIN.EXE segítséggel továbbfordítjuk .COM-ra. Az EXE2BIN az MS-DOS negyedik, kiegészítő lemezén található, a Windows 9x/Me már nem tartalmazza. Ebben az esetben a LINK (TLINK) szerkesztő az átmeneti .EXE fájl készítésénél figyelmeztet a veremszegmens hiányára: "Warning: no stack segment." Ha nem lenne ez az üzenet, az lenne a baj!)

- Adataink elhelyezkedése: adatszegmensben (.data), a program legvégén (int 20h után), vagy a legelején (akkor egy "jmp tovabb" is szükséges).

## .COM programok írásakor elkövethető fatális hibák:

- Veremszegmenst definiálunk: .stack  
(Vermet használhatunk, csak nem definiálhatjuk; természetesen .data használható)
- DS-nek értéket adunk, pl: mov ds, ax
- Az org 100h hiányzik, vagy más értéke van.

A fordító nem jelez hibát, a linker viszont nem fordít .COM-ra ("Cannot generate COM file"), illetve az EXE2BIN a "File cannot be converted" üzenetet adja.

## Az .EXE fájlok ról

A fájl egy néhány száz bájtos fejléc (header) kezdődik. A fejléc tartalmazza az ún. relokációs táblázatot, amelynek tartalma alapján betöltés után a programot módosítja a gép. A módosítás oka a több szegmens használata, a szegmens-relatív utasításokat (pl. MOV AX, \_DATA, CALL FAR) a DOS az aktuális értékre állítja.

A relokáció után DS és ES a PSP elejére áll, CS, IP, SS, SP a fejléc alapján értéket kap, azaz a program elindul.

A felépítésből következően egy .COM fájl kevesebb helyett foglal el és a relokáció hiánya miatt hamarabb töltődik be, mint az .EXE, viszont elavultabbnak minősíthető.

A .COM és .EXE állományok felépítése meglehetősen eltér egymástól, az .EXE szerkezet a modernebb és bonyolultabb. Emiatt egyes vírusok is "szakosodnak".

## Példa:

### Képernyőtörlés (gyorsabb a megszakításnál):

```
mov ax,0b800h ; képernyőmemória
mov es,ax      ; kezdőcíme ES-be
clc            ; hogy DI növekedjen
mov al,' '     ; a karakter szóköz
mov ah,00000111b ; attr: fekete-fehér
xor di,di      ; bal felső saroktól
mov cx,80*25    ; 25 sor, 80 oszlop
rep stosw      ; MOV ES:[DI],AX és
                 ; DI=DI+2, CX-szer
```

## Grafikus üzemmód:

- Első megjelenítő: teletype (≈írógép)
- Első monitor (MDA): szöveges üzemmód, 2 szín
- Később fejlesztik ki a grafikus megjelenítőket:
  - CGA - Color Graphic Adapter (mára muzeális)
  - EGA - Enhanced Graphic Adapter
  - VGA - Video Graphic Array, felülről kompat. (Hercules - mellékvágány)
  - idáig szabványosak az üzemmódok ---
  - SVGA, XGA - ahány gyártó, annyi működés
  - SVGA monitorokra: VESA szabvány kidolgoz.



## Közvetlen képernyőmemória-használat

A RAM bizonyos része a képernyőhöz van rendelve (videomemória, a kártyán). Ha itt egy értéket megváltoztatunk - pl. MOV utasítással -, a képernyón azonnal megjelenik a változás (karakter, vagy képpont).

Az IBM PC-knél a szöveges / grafikus üzemmód elkülönül egymástól: át kell váltani, más a memória szerkezete stb.

Szöveges üzemmódban a képernyőmemória kezdete B8000h (régi Hercules kártyánál B0000h). A sorok lineárisan, egymás után tárolódnak. Egy pozícióhoz két bájt tartozik: egy bájt ASCII kar. kódot egy bájt színkód (ún. attribútum) követ. (Több képernyőlap van, mindegyikre írhatunk, váltás 1 utas-ra.)

## A VGA-MCGA üzemmód

- Az egyik legkönnyebben programozható;
- A képernyő 320x200 pontból áll (kisfelbontású);
- Egyidőben 256 szín lehet a képernyőn (de hogy melyik milyen legyen, külön be lehet állítani, azaz nem 256 színből választhatunk! ← paletta);
- A képernyőmemória kezdőcíme A0000h;
- Egy képpont egy bájt (0-háttérszín, 1..255-előtér);
- A sorok lineárisan, egymás után tárolódnak.

## Példa:

Színes karakter kiírása csak MOV utasításokkal:

```
mov ax,0b800h ; képernyőmem. kezdőcíme
mov es,ax      ; ES-be
mov di,160      ; 2. sor 1. oszlop
mov al,'W'      ; karakter
mov ah,11011010b ; attribútum
mov es:[di],ax ; MAGA A KIÍRÁS!
                 ; először AL íródik ki
```

Az attribútum bájt felépítése:  
1 bit villogás, 3 bit háttérszín (RGB), 1 bit előtér intenzitás, 3 bit előtérszín (RGB).  
Azaz most lila háttér, zöld előtér és villog.

A grafikus üzemmódba (a Pascalhoz hasonlóan) át kell váltani:

- Váltás VGA-MCGA üzemmódba:

```
mov ax,13h
int 10h
```

- Visszaváltás 80\*25-ös szöveges módba + képernyőtörlés (a DOS "mode co80" utasításának felel meg):

```
mov ax,3
int 10h
```

- Egy-egy szín beállítása (újradefiniálás):

```

mov dx,3c8h          ; 0=háttér, 1..255=előtér
mov al,átállítandó_szín sorsz.
out dx,al             ; kommunikálás a videokártyával
inc dx
mov al,piros_összetevő
out dx,al             ; A színösszetevők értéke 0..63
mov al,zöld_összetevő
out dx,al             ; lehet, 6 bites értékek
mov al,kék_összetevő
out dx,al             ; (RGB = Red-Green-Blue).
out dx,al
Példák:
0-0-63 = tiszta (nem kevert)
legerősebb kék
0-0-50 = tiszta kék, de sötétebb
0-0-0 = fekete
63-63-63 = max. fehér stb.

```

- A képernyő tetszőleges pontjának színezése:

```

mov ax,0a000h      ; kezdőcím ES-be*
mov es,ax
mov al,szín_sorszáma ; 0..255
mov di,kiiratás_helye
mov es:[di],al        ; kiiratás
0 - bal felső sarok
319 - első sor utolsó kép-pontja
320 - 2. sor első képpontja stb.

```

\* mert az A0000h szegmens-offset alakban A000:0000!

### Megjegyzések:

- A vízszintes egyenes rajzolása egyszerűbben:

```

cld                      ; DI növekedjen
mov di,320*10+60         ; kezdőpont
mov cx,50                 ; hossz
rep stosb                ; jelentése: CX-szer
                           ; MOV ES:[DI],AL és
                           ; INC DI

```

- A színeknek van kezdőértéke (pl. a 0., azaz a háttér fekete), beállításuk induláskor nem kötelező, viszont össze-vissza értékeket;

- Ha rajzolás *után* egy palettaszín értékét megváltatjuk (az OUT utasításokkal) → az összes ilyen színnel rajzolt képpont azonnal követi a változást! Ilyen módon nagyon egyszerűen készíthető pl. egy **kék** rajzból szempillantás alatt **zöld**.

**A rajzolás (szövegkiírás) elve:** olyan matematikai függvényt készítünk, amelynek során DI egymás után a megfelelő értékeket veszi fel.

Pl. vízszintes egyenes megrajzolása:

```

mov di,320*10+60 ; kezdőpont
mov cx,50          ; hossz (pixel)
ciklus:
  mov es:[di],al
  inc di            ; ez a matem. fv!
loop ciklus

```



- függőleges egyenes:

```
add di,320
```

- 45 fokban balra lefelé dőlő:

```
add di,319
```

- jobbra dőlő:

```
(add di, 321)
```

- a rajzolás alulról felfelé történjen:

```
(sub di, 320)
```

- négyszet, téglalap, háromszög, trapéz:

```
(szakaszokból)
```

- Mozgóképek esetén érdemes a rajzolást / palettaállítást az elektronsugár visszafutása alatt elvégezni (ún. vertical blank):

```

mov dx,3dah
var: in al,dx           ; CGA status reg.
      test al,1000b       ; csak 1 bit kell
      jz var               ; várunk a VB-ra

```

- A képernyőmemória mérete itt  $320 \times 200 \times 1 = 64\ 000$  bájt, teljes egészében elfér az alsó 1 MB RAM-ban. SVGA, XGA esetén lapozás szükséges.

- A színek száma:  $64^3 = 262.144$ , ebből látható egyidejűleg max. 256 különböző a képernyőn.

- A színek állítása az OUT utasításokkal nemcsak grafikus üzemmódban működik, hanem szöveges módban is! Ekkor a 0. szín a háttér, a 7. szín a normál szövegszínt (általában szürke, lásd Start - Programok - MS-DOS Parancssor) állítja át, a többi szín (1..255) a szöveges üzemmód egyéb színeit.

Azaz a szöveges üzemmód színeit tetszőleges finomsággal állíthatjuk be! (←Más programnyelven megoldható???)

ES=A000h  
értéke fix



ES:[DI]  
DI értéke  
állandóan  
változik

- Video interrupt: int 10h. AH=0 → videomód beállítása, AH=0Fh → aktuális video üzemmód lekérdezése. Ez egy ROM-BIOS megszakítás. Az AL értékei:

◆ 0-3 = szöveges üzemmódok	
◆ 4-6 = grafikus	↑ CGA
◆ 7 = szöveges (Hercules)	
(◆ 8-0Ch = PCjr vagy foglalt)	
◆ 0Dh-10h = grafikus	↑ EGA
◆ 11h-13h = grafikus	↑ VGA (IBM defin.)
◆ 18h-62h = szöveges ill. grafikus	↑ SVGA

Az aktuális video üzemmód megtalálható a 0:0449h címen is (ROM-BIOS rendszerváltozók területe).

## Hanggenerálás (PC Speaker)

```
Prg_8255 equ 61h ; A 8255 portcime
Prg_timer equ 43h ; Az osztó programozásának
Timer equ 42h ; és beállításának portcíme
frekv dw 1000 ; hang frekvenciája (Hz)
start: mov bx,frekv ; A Timer részére szükséges
        mov ax,34DDh ; osztó meghatározása
        mov dx,12h
        div bx
        mov ax,bx
        8253 IC
        • Három db. 16 bites számítáció
        1. órajel - másodpercenként 18,2-szer
        2. DMA - memóriafrissítés
        3. hangszóró - ezt módosíthatjuk!
        • Kezdőértékek beállítása: 40-42h port
        • Működésmódban beállítása: 43h port
        • Be/kikapcs: 61h port 0. és 1. bitje
```

```
mov al,0B6h ; A Timer2 felprogramozása
out Prg_timer,al
mov al,bl
out Timer,al
mov al,bh
out Timer,al
in al,Prg_8255 ; A hangszóró bekapcsolása
or al,00000011b ; két bit 1-be állításával
out Prg_8255,al
xor cx,cx ; várakozás
ido: loop ido ; (csippanás hossza)
in al,Prg_8255 ; A hangszóró kikapcsolása
and al,11111100b ; (0. és 1. bit törlése)
out Prg_8255,al
```

## Példák: .COM

```
.model tiny
.code

org 80h
hossz db ?
db ? ; felesl.
elso_kar db ?

org 100h
start:
;<stb.>
```

## .EXE

```
.model small
.stack
.data
.code
start:
MOV AX, _DATA
MOV DS, AX
mov si,80h
mov al,[si] ; hossz
mov ch,[si+2] ; első
mov cl,[si+3] <stb.>
```

nem kellene!

## Példa: a parancssor tartalmát kiírjuk, kétféleképpen

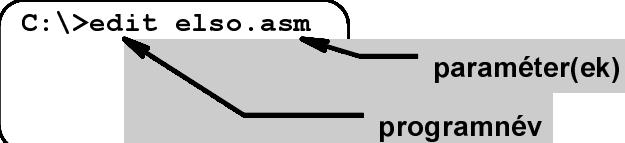
```
.model tiny
.data
szoveg1 db 'A paramétersor hossza: $'
szoveg2 db ' bájt',10,13,'Tartalma: $'
hiba db 'Írjon a paramétersorba!$'
.code

org 80h ; hasonlít Pascal "absolute"-ra
hossz db ? ; hossz változó mutatja a hosszt!

org 100h
start:
mov ax,3
int 10h
```

```
cmp hossz,0
jne tovabb
mov ah,9
mov dx,offset hiba
int 21h
int 20h
tovabb:
mov ah,9
mov dx,offset szoveg1
int 21h
mov ah,2
mov dl,hossz
add dl,'0'
int 21h
mov ah,9
mov dx,offset szoveg2
int 21h
; --- 1. megoldás ---
xor ch,ch
mov cl,hossz
mov ah,2
mov si,81h
ciklus:
mov dl,[si]
int 21h
inc si
loop ciklus
; --- 2. megoldás ---
mov si,81h
xor bh,bh
mov bl,hossz
add si,bx
mov [si].byte ptr '$'
mov ah,9
mov dx,81h
int 21h
int 20h
end start
```

## Paraméterátvétel parancssorból

Pl.  paraméter(ek) programnév

DTA (Disk Transfer Area): a paraméterek a PSP 80h címétől tárolódnak (mind .EXE, mind .COM esetén): 80h=hossz, 81h=" " vagy "/" és 82h..FFh között a paraméter(ek). Az .EXE állományoknál DS a PSP elejére mutat - nem kell átállítani (ahogy eddig tettük)!

## Egérkezelés



A megszakítás működéséhez egy egér driver jelenléte szükséges: DOS alatt egy memóriarezidens program (pl. mouse.com) valósítja meg, Windows - "MS-DOS parancssor"-nál a rendszer biztosítja a meghajtót.

A meghajtó program egyrészt (általában) a soros port hardver megszakítását kezeli (IRQ 4, int 0Ch), ezáltal fogadja az egér saját protokolljával küldött adatokat, másrészről a 33h megszakítást is átveszi, amelyen keresztül a felhasználói programok lekérdezhetik az egér aktuális állapotát.

Az egér pozíciójának és gombjainak lekérdezése:

```
mov ax,3  
int 33h
```

A megszakítás nem vár semmire, rögtön visszatér. A regisztek értéke:

BX=0 → nincs lenyomva egérgomb  
BX=1 → bal gomb lenyomva  
BX=2 → jobb gomb lenyomva  
BX=3 → minden gomb lenyomva (látszik: a bitek váltanak!)  
CX → x koordináta  
DX → y koordináta

Ha az egér a bal felső sarokban áll: CX=DX=0.

**Szöveges képernyő esetén:** az egeret jobbra mozgatva CX NYOLCASÁVAL nő, balra nyolcasával csökken. Lefelé mozgatva DX nyolcasával nő, felfelé nyolcasával csökken.  
Azaz  $0 \leq CX \leq 632, 0 \leq DX \leq 192$ .

Windows alatt, ha az egeret látni szeretnénk, váltsunk át teljes képernyőről ablakra (bal Alt+Enter).

Az int 33h további lehetőségei: egér driver létezésének lekérdezése, egér gombjainak száma, egérkursor megjelenítése és eltüntetése, egér mozgatása adott pontra stb. Windows alatt néha problémák adódnak működésével.

**Példa:** az egeret mozgatjuk és a jobb gombbal kattintunk

```
.model tiny  
.data  
    szoveg1 db 'Mozgassa az egeret a '  
    db 'bal felső sarokba!',10,13,'$'  
    szoveg2 db 'Nyomja le a jobb '  
    db 'egérgombot!',10,13,'$'  
    szoveg3 db 'Engedje fel!',10,13,'$'  
.code  
org 100h  
  
start:  
    mov ax,3  
    int 10h
```

## Memóriarezidens programok

Más néven TSR (*Terminate and Stay Resident*).

A DOS egyfelhasználós-egyfeladatos (single user - single tasking) rendszer: az elindított program megkapja a teljes billentyűzet, képernyő és processzor kezelését, és amíg nem ér véget, nem futathatunk más programot. (A Windows egyfelhasználós-többfeladatos, a UNIX multiuser-multitasking rendszer.) Igen ám, de most jön a TSR.

Egy rezidens program két részből áll: inicializációs és rezidens rész.

Első indításakor beállít bizonyos dolgokat, majd visszaadja a vezérlést az op. rendszernek úgy, hogy a program egy része a memóriában marad, a DOS ezt a területet foglaltnak fogja tekinteni (INT 27h, vagy INT 21h/AH=31h).

 *Most, hogy a program a memóriában van, még semmit nem tud csinálni. Hogyan kerül rá a vezérlés?*

*Úgy, hogy az inicializációs rész a kilépés előtt egy vagy több megszakítást átirányít a memóriában maradó programrészre!*

A megszakítási vektortáblázat átírására külön INT 21h függvények szolgálnak. Az átírás alatt a megszakításokat tiltani kell (CLI), a végén engedélyezni (STI). Így a megszakítás hívásakor a rezidens rész kapja meg a vezérlést. Elvégzi a teendőit és utána általában meghívja az eredeti megszakítást (a vírusok pláne).

**Megjegyzés:** ettől a DOS még egyfeladatos rendszer (bár az INT 1Ch érdekes lehetőségeket nyújt).

Megszakításból megszakítás nem hívható, ezért:

```
pushf      ; INT elemi utasításokkal  
call eredeti_megszakítás  
iret       ; rezidens program vége
```

Ezen a módon működnek a különböző rendszerprogramok:

- billentyűzetátdefiniáló (KEYB, MULTIKEY)
- egérkezelő (MOUSE.COM, MOUSE.SYS)
- CD lejátszó (MSCDEX)
- lemezgyorsító (SMARTDRV)
- hálózati programok stb.

```
mov ah,9  
mov dx,offset szoveg1  
int 21h  
  
vissza1:  
    mov ax,3  
    int 33h  
    cmp bx,2 ; lenyomva?  
    jne vissza2  
    mov ah,9  
    mov dx,offset szoveg3  
    int 21h  
  
vissza2:  
    mov ax,3  
    int 33h  
    cmp bx,0 ; feleng?  
    jne vissza3  
    int 20h  
end start
```

A memóriarezidens programokkal a MEM külső DOS parancs foglalkozik:

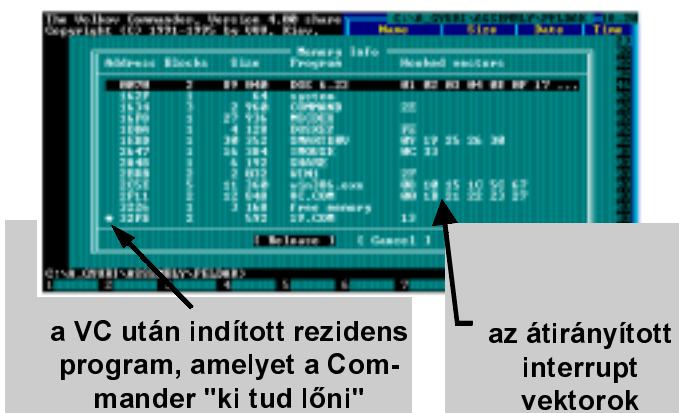
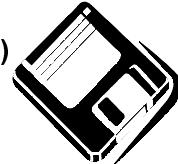
- mem - memória típusok és méretei
- mem/c | more - rezidens prg-k felsorolása
- mem/d - a hagyományos memória tartalma
- mem/f - a szabad memória
- mem/m - megadott programot részletez

Rezidens program megszüntetése: legjobb az "uninstall" funkciójával - ha van -, vagy külső segédprogrammal (pl. Volkov Commanderrel, Alt+F5):

## A fájlkezelésről röviden

Számos megszakítás-függvény segíti a háttértár manipulációját:

- szektorfületek: írás/olvasás (int 13h - ROM !)
- fájlműveletek: létrehozás, nyitás / záras, írás / olvasás (int 21h)
- könyvtár és fájlműveletek (int 21h)



## Az IBM PC memóriatérképe

Cím:	Leírás:
0000:0000	Megszakítási vektortáblázat (256x4 bájt)
0040:0000	ROM-BIOS rendszerváltozók területe (256 bájt)
0050:0000	DOS adatterület
xxxx:0000	BIOS kiegészítés (IO.SYS), DOS megszakításkezelők (MSDOS.SYS - INT 21h), DOS pufferek, adatterületek és installált prg-k (.DRV, .SYS)
xxxx:0000	A COMMAND.COM rezidens része - kb. 4 KB -, amely tartalmazza az INT 22h/23h/24h kezelőit
xxxx:0000	TSR programok
xxxx:0000	Futó felhasználói programok (.COM vagy .EXE)

xxxx:0000	A COMMAND.COM tranziens része, amely a belső parancsok értelmezőjét tartalmazza (újratöltődik). A felhaszn. RAM terület vége 9FFFFh (640 KB).
A000:0000	EGA-VGA-SVGA videomemória (VRAM) - 64 KB! (B000:0000 - monokróm (Hercules) adaptornál B800:0000 - szöveges üzemmód esetén - 32 KB)
C000:0000	Installálható külső ROM (videokártyán stb.)
C800:0000	alaplapi járulékos ROM területe
E000:0000	Az alaplap ROM területe (2 x 64 KB) vagy UMB RAM, vagy expanded memória (F000:0000 - ROM-BIOS (64 KB) - INT 0..1Ch kisz. rutinok FFFF:0000 - hidegindítás belépési pontja FFFF:0005 - ROM-BIOS dátuma ASCII formában)
10000..0000	AT extended memória (elöljén a 4MMA)

## Az Assembly utasítások felépítése

Egy-egy Assembly utasítást a fordító 1..10-20 bájtnyi számra fordít le (gépi kód).

- 1 bájtos utasítások pl: NOP, HLT, INT 3 stb. A gépi kód számok sorozata (nehéz benne programozni), a CISC CPU-kban ún. mikroprogramot indít el.
- 7 bájtos: mov cs:tomb[di],1

A gépi kódot jelentő számok közül egy vagy több bájt az *utasításkód*, a többi a *paraméter(ek)*.

Pl. az INT 20h: CDh 20h (CDh=utkód, 20h=paraméter)

Az utasításkód felépítése: néhány bit tartalmazza az operációs kódot, a többi bit jelöli ki a regisztert.

Egybájtos utasításnál: regiszter:

Pl. INC AX = 1000 000 b	000 - AX
INC BX = 1000 011 b	011 - BX stb.

opkód: INC

A több bájtos utasítások is hasonló módon épülnek fel, az operandust meghatározó összetevők neve "w", "mod" és "r/m" (w=0 esetén 8 bites az op, w=1-nél 16).

(Érdekesség: egyes PIC mikrokontrollerekben az utasítások 14 bitesek, az adatok 8 bitesek.)

## Az Assembly lehetőségei

Assemblyben - jellegénél fogva - lehetőségünk van olyan trükkökre is, amelyeket a magas szintű programnyelvek "nem támogatnak", pl:

- HDD és FDD kezelése szektoronként (pl. formattálás!);
- A váltóbillentyűk (Shift, Ctrl, Alt) érzékelése (a BIOS az INT 9 segítségével tartja nyilván, a 40:17h és 40:18h címeiken);
- A billentyűzet LED-jeinek vezérlése (villogtatás stb.);
- Futás közbeni kódátírás. a program módosítja önmagát (!)

VIGYÁZAT! A gépi kódú programokkal nagy kárt lehet okozni a szoftverben, sőt a hardverben is!

## A váltóbillentyűk kezelése

A billentyűzet interrupt (int 16h) 2-es szolgáltatása adja vissza a váltóbillentyűk állapotát: AL-ben 0. bit=jobb Shift, 1. bit=bal Shift. Ugyanakkor a 0:0417h és 0:0418h memóriacímen (ROM-BIOS rendszerváltozók) is megtalálhatók.

Példa: a jobb Shift lenyomását és felengedését vizsgáljuk

```
.model tiny
.data
    szoveg1 db 'Nyomja le a jobb Shiftet!',10,13,'$'
    szoveg2 db 'Engedje fel!',10,13,'$'
.code
org 100h
start:
```

```
        jz ki
            mov ah,8
            int 21h
            cmp al,27
            jne ki
            mov bl,1
        ki: ret
bill endp

c1:
        push cx
        xor cx,cx
        c2:
            nop
            loop c2
            pop cx
        loop c1
        ret

idohuzas proc
        mov cx,200
    idohuzas endp
end start
```

Tehát a LED-eket két OUT utasítással vezérelhetjük, mindenkor után rövid időhúzás szükséges (LOOP+NOP, 65536-szor)!

```
        mov ax,3
        int 10h
        mov ah,9
        mov dx,offset szoveg1
        int 21h
vissza1:
        mov ah,2
        int 16h
        ror al,1
        jnc vissza1 ; lenyomva?
        mov ah,9
        mov dx,offset szoveg2
        int 21h

        vissza2:
            mov ah,2
            int 16h
            ror al,1
            jc vissza2 ; feleng?
            int 20h
        end start
```

## A billentyűzet LED-jeinek kezelése



A 60h porton keresztül vezéreljük a billentyűzethez tartozó chip-et (billentyűzet vezérlő 8042). Az adatok kézikönyvekből kereshetők ki. A LED-ek állítása nem befolyásolja a Lock billentyűk állapotát.

Példa: a NumLock LED-et villogtatjuk, Esc-kilépés

```
.model tiny
.data
    szoveg db 'Kilepes '
    db Esc-re:$'
.code
org 100h
start:
        mov ax,3
        int 10h
        mov ah,9
        mov dx,offset szoveg
```

Itt egy 5 bájtos JMP FAR található - "távoli" ugrás a ROM-BIOS belépési címére (értéke gyártótól függ).

- Processzor és CMOS ellenőrzése;
- Időzítők, megszakítás vezérlők és DMA felprogramozása;
- Billentyűzet illesztő, alsó 64 KB RAM ellenőrzése;
- Monitor illesztő típusának megállapítása, alaphelyzetbe állítása, memóriamérétének meghatározása;
- Alapértelmezett interrupt vektor címek kitöltése;
- ROM-scan: alaplapon járulékos ROM keresése, utána ROM keresése I/O egységek kártyáin

A következő tartományban keres:

- alaplap bővített ROM: E0000h-tól 64 KB méretben (régen a ROM-BASIC-et tartalmazta. Ha nincs → a helyén lehet UMB RAM, vagy expanded memória)
- kártyán: C0000h-tól DFFFFh-ig 2 KB-os blokkonként

```
        int 21h
vissza:
        mov bl,010b
        call led
        call bill
        cmp bl,1
        je kilepes
        call idohuzas
        xor bl,bl
        call led
        call bill
        cmp bl,1
        je kilepes
        call idohuzas
        jmp vissza
kilepes:
```

```
        int 20h
led proc
        mov al,0edh
        out 60h,al ; előkész.
        xor cx,cx ; időhúzás
        ciklus:
            nop
            loop ciklus
            mov al,bl
            out 60h,al ; LED ki!
            ret
        led endp
bill proc ; Esc-re BL=1
        xor bl,bl
        mov ax,100h
        int 16h
```

**A ROM akkor érvényes, ha felépítése:**

0. bájt = 55h
  1. bájt = AAh
  2. bájt = 512 bájtos blokkok száma (modul hossza), az alaplap járul. ROM-nál nem használják
  3. bájt = program belépési pontja
- ...
- modul utolsó bájtja = checksum-hoz szükséges nullázó bájt (az alaplapi járulékos ROM-nál ez a 65535. bájt)**

**Pascal mintaprogram a checksum kisz.-ra (járulékos ROM; helyes eredményt csak valós módban futva ad!):**

```
program romcheck;
var i,j:word;
x:array[0..65534] of byte absolute $E000:$0;
y:byte absolute $E000:$FFFF;
begin
  i:=0;
  for j:=0 to 65534 do
    i:=i+x[j];
  i:=i+y;
  writeln('checksum=', i mod 256);
end.
```

A ROM érvényes, ha checksum=0. Ekkor a BIOS egy CALL FAR SEGMENT:0003 utasítással átadja a vezérlést, így a külfölféle hardver eszközök hozzá tudnak kapcsolódni a rendszerhez (interrupt vektorok beállítása stb.). A visszatérés RET FAR utasítással történik.

- CMOS-ban tárolt adatok kiolvasása, memória méretének és a háttértárok típusának megállapítása, memória ellenőrzése ( minden bájtba ír ), billentyűzet alapállapotba állítása, párhuzamos és soros portok keresése és az eredmény összehasonlítása a CMOS-ban tároltal (lényeges eltérés esetén hibaüzenet);
- Megpróbálja olvasni az A: jelű meghajtó rendszerbetöltső programját (BOOT szektor - head 0, track 0, sector 1), ha sikeres a betöltés, a vezérlést átadja. Ha nem, a C: egységgel próbálkozik (Master Boot Record, head 0, cylinder 0, sector 1, a 0:7C00h helyre tölti be és végrehajtja; egyes alaplapokon az A: C: sorrend megfordítható). Ha ez sem sikerül, hibaüzenetet ír ki ("Non system disk or disk error..."; a nagyon régi eredeti IBM PC-ken ehelyett INT 18h utasítással elindul egy BASIC értelmező).

Egyes gépeken a ROM tartalmát átmásolják a RAM bizonos területére és innen futtatók - gyorsabb!

**Példa Assemblyben megírt Pascal függvényre:**

```
function jobbshift:
  boolean;
label tovabb;
var seged:boolean;
begin
  asm
    mov ah,2
    int 16h
  end;
  jobbshift:=segéd
end;
```

A függvény hívása: if jobbshift then write ('lenyomva') else write('felengedve');

**Példa .OBJ fájl beillesztésére. Az .ASM állomány:**

```
public _kkiir    ↗ kiigazítás (para, page stb.)
_text segment byte public 'code'
assume cs:_text
_kkiir proc near
  ;<ide jön a program>
  ret
_kkiir endp
_text ends
end
```

"public" - más program hívja, mint szubrutint!

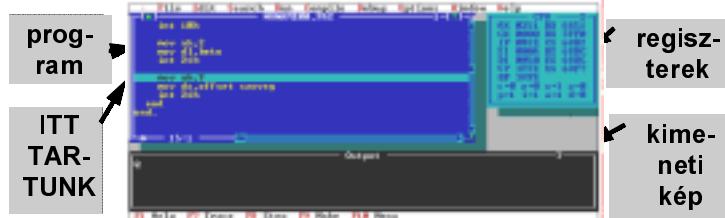
A Pascal programban: ↗ külső (public) modul

```
procedure _kkiir; external; {$L asmprg.obj}
```

Meghívása: \_kkiir;

## Kapcsolat a Turbo Pascallal

Lehetőség nyílik Assembly programbetétek használatára (80286/287-es utasításokig), változók átvételére, .OBJ fájlok beszerkesztésére. A regisztereket megtekinthetjük, a programot lépésenként futtathatjuk:



**Példa a megszakítások átirányítására (INT 1Ch):**

```
PROGRAM ora_atiranyitasa;
USES crt,dos;
VAR eredetiora: pointer;
  i:byte;
PROCEDURE oramegsz;
  interrupt; {Az új megsz.}
begin
  textColor(red);
  write('B');
  textColor(lightgray)
end;
BEGIN
  clrscr;
  {A megsz.átiirányítása}
  getintvec($1c,eredetiora);
  setintvec($1c,@oramegsz);
  for i:=0 to 255 do {Fóprg}
  begin
    delay(50);
    write('A')
  end;
  {A megsz.visszaállítása}
  setintvec($1c,eredetiora)
END.
```

(A képernyőn megjelenik: AAAAABAAAAAAAB... ; a „B” betűk száma a gép sebességétől is függ.)

**Példa Pascalban deklarált változó használatára:**

```
PROGRAM pas2;
VAR betu:char;
  szoveg:string;
BEGIN
  betu:='Q';
  szoveg:='Üdv$';
  asm
    mov ax,3
    int 10h
  END.
```

```
        mov ah,2
        mov dl,betu
        int 21h
        mov ah,9
        mov dx,offset
                      szoveg+1
        int 21h
      end
END.
```

A program kiírja a képernyőre: "QÜdv".

## Kapcsolat a C nyelvvel

```
/* A program letörli a képernyöt és kiírja: "C nyelv".
   Fordítása: C:\>tcc -B prg_neve (Enter) */
main()
{
  char betu='C';
  char *szoveg=" nyelv$";
  asm mov ah,2
  asm mov dl,betu
  asm int 21h
  asm mov ah,9
  asm mov dx,offset szoveg /* lea dx,szoveg NEM JO! */
  asm int 21h
}
```

## Nyomkövetés (Debugging)

Ha a forrásprogramban szintaktikai hibát követünk el → az assembler fordító (TASM) jelez és nem fordítja le a programunkat.

Ha súlyos elvi hibát vétünk (pl. .COM formátumnál verem-szeg-menst definiálunk, vagy az org 100h hiányzik stb.) → a linker (TLINK) jelez és nem keletkezik futtatható fájl.

Az egyéb programozási hibák viszont nem derülnek ki, csak indítás után láthatjuk a következményeket.

Nincs IDE, mint a Turbo Pascalban. Különféle külső nyomkövető (debugger) programokat használhatunk fel a hibás rész(ek) megtalálására.

### 1. A DEBUG nyomkövető

- Külső DOS program (C:\DOS\DEBUG.EXE vagy C:\WINDOWS\COMMAND\DEBUG.EXE, ~20 KB-os);
- Egykarakteres parancs üzemmódban működik;
- Lehetőségei (zárójelben a parancsok):
  - hexadecimális számok összeadása és kivonása (h), *assembler* (a), *disassembler* (u), memória tartalmának megtekintése (d) és átírása (f), fájlműveletek (l, w, n), a regiszterek tartalmának megtekintése (r), *lépésekénti* programvégrehajtás (p, t) stb.

```
C:\>debug
-a
38AA:0100 mov ah,2
38AA:0102 mov dl,41
38AA:0104 int 21
38AA:0106 int 20
38AA:0108 ^C
-g
A
Program terminated normally
-q
C:\>_
```

A DEBUG segítségével Assembly programot írunk és futtatunk!

## Egyéb segédprogramok

### 1. Sourcer

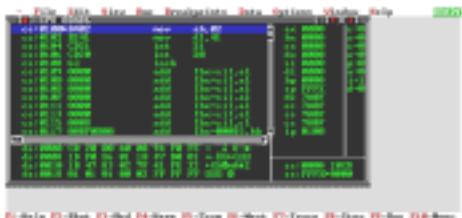
Régi (copyright-os) disas-sembler, amely az értékkedásokat és ugrásokat címkékkel jelzi, felismeri az adatterületeket, commentekkel látja el a leggyakoribb megszakításokat stb. Használata jóval kényelmesebb, mint a DEBUG/Turbo Debugger.

```
6310:0005 B4 09      mov ah,9
6310:0007 8D 16 0002  lea dx,
             data_1 ; (6311:0002='Assembly')
6310:000B CD 21      int 21h
             ; DOS Services ah=function 09h
             ; display char string at ds:dx
```



### 2. Turbo Debugger

- A TASM programcsomag része (a MASM-nál CodeView);
- Lehetőségei: lépésekénti programvégrehajtás, regiszterek és memória figyelése stb.



Tulajdonképpen ezen programokkal minden viszszafejthető (pl. a **COMMAND.COM** is), ha van elég időnk hozzá (és nem vagyunk törvénytisztelők).

Monitorprogram: a futás körülményeit szimulálja.

### 2. Tech Help

Az IBM PC hardver és szoftver adatait (megszakítások, ROM változók, fájlok felépítése stb.) tartalmazó hypertext jellegű adatbázis program, szintén copyright-os.

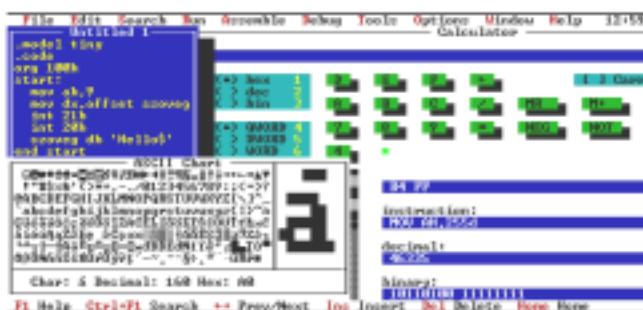


### 3. Assembly keretprogramok

Ezek a shareware programok a Turbo Pascal IDE környezetéhez hasonló kényelmes lehetőséget nyújtanak Assembly programok írás-fordítás-futtatás-nyomkövetsésére, valamint egyebeket (ASCII táblázat, számológép stb.) is biztosítanak.



ASMLAB



ASMEDIT

Ralf Brown  
About...  
Ralf Brown is a Systems Scientist (Research Faculty) at Carnegie Mellon University's (pioneering) Language Technologies Institute (now the Center for Machine Translation in Pittsburgh, Pennsylvania). Currently working on the AT&T/CERN 2, Z8000, Zilog Z80 and Zilog Z180 and Zilog Z1800 interrupt projects, and recently on Zilog's Z80C32X, and Generation 3 interrupt-based machine translations. He is well-known in cyberspace for maintaining the Interrupt List and various other programs, and has co-authored a number of books.

Ralf's Informational Pages

- [Biography of the author](#)
- [Latest News](#)
- [Documentation Information Regarding Interrupts](#)
- [Locating References on the Web](#)
- [Additional Information](#)

Ralf Brown's Interrupt List  
Indexed HTML Version - Release 61

A Gift to DOS Programmers

HTML version of the famous Ralf Brown Interrupt List with over 9000 linked pages and 350 indexes making the process of searching much easier. This list contains every documented and undocumented interrupt call known. Ralf Brown is a Postdoctoral Fellow at Carnegie Mellon University's Center for Machine Translation in Pittsburgh, Pennsylvania. He is well-known in cyberspace for maintaining the Interrupt List. We all appreciate his continued support.

• [Table of Contents](#) - Access the Interrupt List by Table of Contents

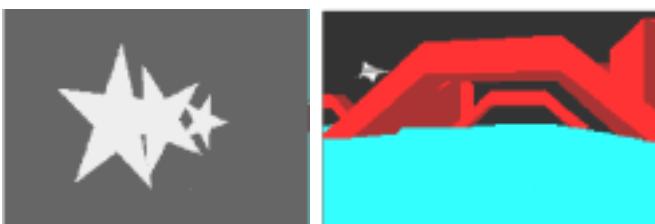
• [Categories](#) - Access the Interrupt List by Command Category

• [Interrupt](#) - Access the Interrupt List by Interrupt Number

Search:

### 4. Assembly demók

A találkozókon a feladat minél látványosabb - általában grafikus - program készítése, a fájl maximális mérete kötött (256 bájt, 4 KB stb.) .



### 5. Internet címek

Assembly programok:

<http://www.simtel.iif.hu/simtel.net/msdos/asmutl.html>

Ralf Brown féle megszakítási lista:

[www.pobox.com/~ralf](http://www.pobox.com/~ralf)

A megszakítási lista HTML formátumban:

[www.ctyme.com/rbrown.htm](http://www.ctyme.com/rbrown.htm)

és

[www.delorie.com/djgpp/doc/rbinter](http://www.delorie.com/djgpp/doc/rbinter)