Eljáráshívás: **CALL** `op`
Működése:

- A visszatérési címet a verem tetejére teszi
- ugrik op címre (általában címkével adjuk meg)

Visszatérés az eljárásból: **RET**

Visszatérés eljárásból: **RET**
Működése:

- Kiveszi a verem tetejéről a visszatérési címet
- Odaugrik

Eljárás meghívása: **CALL**

String/számsorozat következő (8 bites) elemének betöltése: **LODSB**
Működése:

- AL ← DS:[SI]
- SI ←
  - ha D flag 0: SI + 1
  - ha D flag 1: SI – 1

String/számsorozat következő (16 bites) elemének betöltése: **LODSW**
Működése:

- AX ← DS:[SI]
- SI ←
  - ha D flag 0: SI + 2
  - ha D flag 1: SI – 2

---

8 bites érték tárolása stringbe/számsorozatba: **STOSB**
Működése:

- ES:[DI] ← AL
- DI ←
  - ha D flag 0: DI + 1
  - ha D flag 1: DI – 1

---

16 bites érték tárolása stringbe/számsorozatba: **STOSW**
Működése:

- ES:[DI] ← AX
- DI ←
  - ha D flag 0: DI + 2
  - ha D flag 1: DI – 2

Verembe helyezés: **PUSH** `op16`
Működése:

- SP ← SP – 2 (tehát behelyezéskor SP fix 2-vel csökken)
- SS:[SP] ← op16

A fix 2 miatt csak 16 bites (wordös) operandussal működik!

Veremből kivétel: **POP** `op16`
Működése:

- op16 ← SS:[SP]
- SP ← SP + 2 (tehát kivételkor SP fix 2-vel nő)

A fix 2 miatt csak 16 bites (wordös) operandussal működik!

D flag értéke alapértelmezésben 0. 1-be az **STD**, 0-ba a **CLD** utasítással állíthatjuk.

Szoftveres megszakítás kérése (DOS/BIOS rutin hívása): **INT** `konst8`
Működése: konst8 számú megszakítást kezdeményezi.

Eljárás szintaxisa:

```
eljarasneve PROC
     ; többi kód
     ; mindenképp fusson rá egy RET utasításra!
eljarasneve ENDP
```

Regiszter nullázása másképpen: **XOR** `reg, reg.` Ez a módszer gyorsabb, és 16 bites regiszter esetén rövidebb gépi kódot eredményez.

Regiszter nulla értékre tesztelése (triviálisan **CMP** reg, 0) másképpen: **OR** reg, reg. A Z flag és a regiszter értéke szempontjából ugyanaz történik, de az **OR** reg, reg gyorsabb, és többnyire (AL-től különböző regiszter esetén) rövidebb gépi kódot eredményez.

AL 8 → 16 bitre bővítése AX-be előjeltelen érték esetén („előjeltelen CBW”):
XOR AH, AH
AX 16 → 32 bitre bővítése DX:AX-be előjeltelen érték esetén („előjeltelen CWD”): **XOR** DX, DX
A CBW-vel, CWD-vel ellentétben nem csak ezekkel a regiszterekkel működik.

DS:SI által mutatott 0-terminált string végigjárása (pl. kiírás céljából):

```
kov: LODSB
     CMP        AL,  0
     JE         vege
     ; AL-ben van a beolvasott karakter,
     ; itt végezhetjük el a kívánt műveletet
     JMP        kov
vege:     ; ide írhatunk mindent, amit a bejárás
          ; után szeretnénk csinálni
```

Műveletsorozat CX-ben tárolt darabszámszor való végrehajtása:

```
     ; kód, ami előállítja a kívánt CX tartalmat
kov: ; a többször végrehajtandó műveletsorozat
     ; (ciklusmag), ami nem módosítja CX-et
     LOOP kov
```

Kettesével léptetést, és hasonlókat a CX ciklusmagban történő módosításával érhetünk el. (Ekkor nem biztos, hogy az eredeti CX tartalomszor fut le a ciklusmag.)

Ha MASM-ban hexadecimális konstanst adunk meg, annak mindenképpen számmal kell kezdődnie. Ha nem így van (pl. EF53h), akkor írjunk elé egy 0-t (Pl. 0EF53h). Ha egy sorban egy hexadecimális konstansra Symbol not defined hibát kapunk, akkor ezt a vezető 0-t felejtettük el.

AX-ben lévő szám számjegyeinek meghatározása (kiíráshoz, összeszámoláshoz, …): az általános számrendszerátváltó algoritmussal, 32/16 bites osztással. Az AX == 0 speciális esetet külön kell kezelni. 10-nél nagyobb alapú számrendszerben történő kiírásnál figyelni kell a 9-nél nagyobb alaki értékű számjegyekre. (A betűk nem a '9' után közvetlenül következnek az ASCII táblában.) A számjegyek fordított sorrendben állnak elő az algoritmus végrehajtása során, ezért kiíráshoz meg kell fordítani a sorrendet. (Pl. verem segítségével.)

AL-ben lévő ASCII karakter képernyőre írása (és a kurzor léptetése):

```
     MOV        AH,  0Eh  ; decimálisan 14
     INT        10h
```

Karakterkonstansok a C-hez hasonlóan használhatók, például:

```
     MOV        AL,  'p'
```

Stringet szintén a C-hez hasonlóan definiálhatunk:

```
hello     DB    ”Hello Assembly!”, 0
```

Figyeljünk oda arra, hogy a C-vel ellentétben **nem** kerül automatikusan a string végére lezáró 0 karakter!

Sortörést is tehetünk stringbe a 13, 10 kódú karakterekkel (CR, LF):

```
sortor    DB    ”Es most jon a sortores:”, 13, 10, 0
```

Számsorozat definiálása: vesszővel elválasztva. Példa:
8 bites elemekkel:

```
bajtok    DB        7, 8, 39, 42
```

16 bites elemekkel:

```
wordok    DW        10, 1423, 99, 15
```

Definiált érték offsetjének megállapítása: offset kulcsszó. Jelentése nagyon hasonlít a C nyelv address-of (&) operátorára. Példa:

```
     MOV        SI,  offset hello
```

ahol hello definíciója:

```
hello     DB        „Hello Assembly!”, 0
```

Az Assembly nyelvben sokszor könnyebb feltételesen átugrani egy kódrészletet, mint feltételesen végrehajtani, ezért segíthet, ha úgy gondolkodunk egy probléma megoldásakor, hogy egy műveletet vagy műveletsort mikor *nem* kell végrehajtani.

Módszer előjeles számok kiírására AX-ből: ha negatív, akkor kiírunk egy mínusz jelet, negáljuk a számot, és kiírjuk előjeltelenként. Ha nemnegatív, kihagyjuk a mínusz jelet és a negálást.

```
     CMP        AX,  0
     JGE        elojelkesz
     PUSH       AX
     MOV        AL,  '-'
     MOV        AH,  0Eh
     INT        10h
     POP        AX
     NEG        AX
elojelkesz:
     ; Kiírjuk AX tartalmát, mintha előjeltelen lenne
```
