Aritmetikai utasítások

Az értékadó és aritmetikai utasítások során a címzési módok különböző típusaira látunk példát. Az x86-os processzor memóriája és regiszterei a little endian tárolást követik, vagyis az alacsonyabb címen az alacsonyabb helyiértékű bájt helyezkedik el. Az assembly nyelvben a negatív számok ábrázolására a kettes komplemens számábrázolást használják. Az egyes műveletek beállítják a *STATUS* regiszter bitjeit is:

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

Az *STATUS* regiszter alábbi bitjei nem változnak az aritmetikai utasítások végrehajtása során:

Direction — A sztring műveletek iránya, 0: növekvő, 1: csökkenő.

Interrupt — 1: megszakítás engedélyezése, 0: tiltása.

Trap — 1: "single step", azaz lépésenkénti végrehajtás engedélyezve van.

Az alábbi bitek változhatnak:

Overflow — Előjeles túrcsordulás: 1, ha az eredmény nem fért el az adott típus értéktartományán, 0 különben.

Sign — Előjel, az eredmény legmagasabb helyiértékű bitje.

Zero — Jelzi, hogy nulla-e az eredmény (1 - igen, 0 - nem).

Auxiliary Carry — Másodlagos átvitel, jelzi ha volt átvitel az eredmény 3. és 4. bitje között.

Parity — Az eredmény alsó 8 bitjének paritása (a paritás akkor 1, ha az alsó 8 biten az 1-esek száma páros, különben a paritás 0. Pl. 01011101 esetén a parity értéke 0, 01011001 esetén a parity értéke 1).

Carry — Átvitel előjel nélküli műveletekhez. Akkor 1, ha volt aritmetikai átvitel az eredmény legfelső bitjénél (előjeltelen aritmetikai túlcsordulás), 0, ha nem.

Összeadás

ADD: Az **ADD** utasítás hozzá adja a forrás operandus értékét a cél operandushoz. A két operandus mérete meg kell, hogy egyezzen. Az utasítás szintaxisa:

```
ADD cél, forrás
```

A forrás tartalma változatlan marad, az cél operandus értéke pedig maga az összeg lesz. A lehetséges operandusok halmaza megegyezik a **MOV** utasítás esetén tárgyaltakkal.

Példák:

```
MOV EAX, 3h     ;     EAX = 3h
ADD EAX, 5h     ;     EAX = EAX + 5h = 3h + 5h = 8h
ADD EAX, EBX    ;     EAX = EAX + EBX
ADD AX, BX      ;     AX = AX + BX
ADD EAX, BX     ;     Helytelen! Méretbeli különbségek!
```

```
ADAT1     DD   912, 920, 928, 936, 944 ; adatszegmens tartalma
```

```
MOV ESI, offset ADAT1  ; ESI regisztert ráállítjuk az ADAT1-re
MOV EAX, 5d            ; EAX = 5h
ADD EAX, [ESI]         ; EAX = EAX + [ESI] = 5 + 912 = 917
```

Az **ADD** utasítás módosítja a **Z**, **C**, **A**, **S**, **P** és **O** flag értékét a *STATUS* regiszterben, viszont mi csak a **C**, az **O** és az **S** flag értékeivel foglalkozunk. Az összeadás során ügyelni kell a túlcsordulásokra. 8, 16 és 32 biten is különböző intervallumban tudunk tárolni előjeles számokat. Az egyszerűség kedvéért most figyeljük csak 8 biten a flag-ek viselkedését.

Ahhoz, hogy Visual Studio-ban előhozzuk a flag értékeket, Debug módban nyissuk meg a Regisztereket megjelenítő ablakot, majd az ablakon belül jobb egérgombbal kattintva, az előugró menüsorból válasszuk ki a ***Flags*** menüpontot.

1. Példa:

```
53 + 18 = 71

       00110101
     + 00010010
   0 0 01000111     Rendben. Előjelhelyes és nincs se átvitel, se
                    túlcsordulás.
```

Túlcsordulás(**O**), Átvitel(**C**), Előjel(**S**)

2. Példa:

```
53 + 83 = 136

       00110101
     + 01010011
   1 0 10001000     Ez -120! Előjel rossz és túlcsordulás is van!
```

A fenti példában az előjeles számábrázolás miatt nem a valódi értéket kapjuk vissza 8 biten. A 2-es komplemenses számábrázolásban az első bit az előjel bit, amely ha 1, akkor negatív számról van szó.

3. Példa:

```
-53 + (-83) = -136

       11001011
     + 10101101
   1 1 01111000     Ez +120! Előjel is rossz, túlcsordulás és
                    átvitel is van!
```

Ha egy 8 bites és egy 16 bites, vagy egy 16 bites és 32 bites számot szeretnénk összeadni előjelhelyesen, akkor assemblyben a 8 és 16 bites pár esetén a 8 bites operandust 16 bitesre, a 16 és 32 bites pár esetében pedig a 16 bites operandust 32 bitesre kell konvertálnunk.

Kiterjesztési műveletek

CBW: A **CBW** (Convert Byte to Word) utasítás az AL 8 bites regiszterben tárolt értéket 16 bitesre konvertálja (előjelhelyesen). Azaz kiterjeszti az **AL** regiszter előjel bitjét az **AH** regiszterbe, megőrizve az előjel értékét.

Példa:

```
MOV AL, -10d   ;    AL = 0F6h
MOV BX, 2525d  ;    BX = 09DDh
CBW            ;    AX = FFF6h
ADD AX, BX     ;    AX = 09D3h
```

CWD: A **CWD** (Convert Word to Doubleword) utasítás az **AX** 16 bites regiszterben tárolt értéket 32 bitesre konvertálja (előjelhelyesen). Azaz kiterjeszti az **AX** regiszter előjel bitjét a **DX** regiszterbe.

Példa:

```
MOV AX, -101d  ;    AX = FF9Bh
CWD            ;    DX:AX = FFFFFF9Bh
```

Tehát felhívnánk még egyszer a figyelmet, hogy, míg a **CBW** esetében az **AL** 8 bites regiszter 16 bitesre konvertálása után a 16 bites előjeles érték elérhetővé vált az **AX** regiszterben, a **CWD** esetében az **AX** 16 bites regiszter 32 bitesre konvertálása után a 32 bites előjeles érték nem az **EAX** regiszterben érhető el, hanem a **DX:AX**-ben.

CWDE: A **CWDE** (Convert Word to Extended Doubleword) utasítás is az **AX** 16 bites regiszterben tárol értéket konvertálja 32 bitesre, viszont a **CWD** utasítással ellentétben az **EAX** regiszterre egészíti ki előjelhelyesen, nem pedig a **DX**-re.

Példa:

```
MOV AX, -101d  ;    AX = FF9Bh
CWDE           ;    EAX = FFFFFF9Bh
```

CDQ: A **CDQ** (Convert Doubleword to Quadword) utasítás is az **EAX** 32 bites regiszterben tárol értéket konvertálja 64 bitesre az EDX:EAX regiszterpárba. Tehát az EAX regiszter felső bitjével lesz kitöltve az EDX regiszter.

Hasznos osztások előtt az EAX regiszter kiterjesztésére 32 bites osztásnál, mivel ekkor az osztandó az EDX:EAX regiszterpár.

Példa:

```
MOV EAX, -123456789159   ;    EAX = -123456789159
CDQ                      ;    EDX:EAX = -123456789159
MOV EBX, 1000000         ;    EBX = 10000000
IDIV EBX
```

Kivonás

SUB: A **SUB** (SUBtract) utasítás két operandusú kivonás művelet. Kivonja a forrás operandust a cél operandusból. A lehetséges operandusok halmaza megegyezik az **ADD** és a **MOV** utasításnál tárgyaltakkal. Az **SUB** utasítás módosítja a **Z**, **C**, **A**, **S**, **P** és **O** flag értékét a *STATUS* regiszterben. A szintaxis:

```
SUB cél, forrás
```

Példa:

```
MOV EAX, 40h   ;    EAX = 00000040h
SUB EAX, 27h   ;    EAX = 00000019h     S = 0, O = 0, C = 0
```

```
MOV EAX, 40h   ;    EAX = 00000040h
SUB EAX, 57h   ;    EAX = FFFFFFE9h     S = 1, O = 1, C = 0
```

```
ADAT1     DD   912, 920, 928, 936, 944 ; adatszegmens tartalma
```

```
MOV ESI, offset ADAT1 ; ESI regisztert ráállítjuk az ADAT1-re
```

```
MOV EAX, 2000d          ; EAX = 7D0h
SUB EAX, [ESI]          ; EAX = EAX - [ESI] = 2000d-912d = 1088d
```

```
MOV BX, 5d              ; BX = 5h
MOV AX, 9d              ; AX = 9h
SUB AX, BX              ; AX = AX - BX = 9h - 5h = 4h
```

Szorzás

32 bites módban az egész értékek összeszorzása elvégezhető, mint egy 32, 16 vagy 8 bites művelet.

MUL: 32 bites módban a **MUL** (unsigned MULtiply - előjelnélküli szorzás) utasításnak három verziója van: az első verzió egy 8 bites operandust szoroz össze az **AL** regiszterrel. A második verzió egy 16 bites operandust szoroz össze az **AX** regiszterrel. A harmadik verzió egy 32 bites operandust szoroz össze az **EAX** regiszterrel. A szorzó és a szorzandó méretének mindig meg kell egyeznie és a szorzat mindig ennek a méretnek a kétszerese. A három formátum elfogad regisztert és memória címet is operandusként, viszont sosem közvetlen operandust (konstanst):

```
MUL reg/mem8

MUL reg/mem16

MUL reg/mem32
```

A **MUL** utasításnak tehát **egy** operandusa van, ami a szorzó. Az alábbi táblázat mutatja be az alapértelmezett szorzandót és a szorzatot, attól függően, hogy mekkora méretű szorzót adtunk meg:

Szorzandó	Szorzó	Szorzat
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

A táblázatban leírtak értelmezése: ha az **AL** regiszter tartalmát megszorozzuk egy **8** bites operandussal, akkor az eredmény az **AX** regiszterbe kerül. Az **AX** megszorozva egy **16** bites operandussal, az eredmény a **DX** és **AX** regiszterekbe kerül, ami azt jelenti, hogy a szorzat felső 16 bitje a **DX** regiszterbe, míg az alsó 16 bit az **AX** regiszterbe kerül tárolásra. Ugyanezen az elven működik az **EAX** regiszter megszorzása egy **32** bites operandussal, mely során az eredmény az **EDX** és **EAX** regiszterekbe kerül. Mivel a cél operandus mérete kétszer akkora, mint a szorzandó és a szorzó mérete, ezért túlcsordulás nem fordulhat elő. A **Carry** flag 1 értéke jelzi, ha történt átvitel 8 bites szorzás esetén az **AH**-ba, 16 bites szorzás esetén az **DX**-be, illetve 32 bites szorzás esetén az **EDX**-be.

Példa:

```
MOV AL, 150d    ; AL = 96h
MOV BL, 10d     ; BL = 0Ah
MUL BL          ; AX = AL * BL = 96h * 0Ah = 05DCh

_____________________________________________________________

MOV AX, 8000d   ; AX = 1F40h
MOV BX, 5000d   ; BX = 1388h
MUL BX          ; DX:AX = AX * BX = 1F40h * 1388h = 0262 5A00h

_____________________________________________________________

MOV EAX, 1000000d   ; EAX = 000F 4240h
MOV EBX, 5000000d   ; EBX = 004C 4B40h
MUL EBX         ; EDX:EAX = EAX * EBX = 0000 048C 2739 5000h

_____________________________________________________________

MUL WORD PTR [SI]   ; DX:AX = AX * (SI helyen a memóriában
                    ; található szó)
; Kell a WORD PTR, mert a fordító nem tudja, hány bites
; művelet! Természetesen a WORD helyett használhatunk BYTE,
; illetve DWORD típust is
```

IMUL: Előjeles szorzás. Működése hasonló a **MUL** utasításhoz, azzal a különbséggel, hogy az **IMUL** megőrzi a szorzat előjelét. Az x86-os utasítás halmaz három különböző formátumát támogatja az **IMUL** utasításnak: egy-, kettő-, illetve három operandusos verzió.

Az **egy** operandusú verzió használata megegyezik a **MUL** utasításéval:

Példa:

```
MOV AL, -2d     ;     AL = 0FEh
MOV CL, 3d      ;     CL = 03h
IMUL CL         ;     AX = 0FFFAh
```

IMUL két operandussal (80286-tól): az első operandus egy regiszter (16 vagy 32 bites), a második operandus lehet regiszter, memóriahivatkozás vagy közvetlen érték (konstans), az elsővel egyező bitszélességű. Az eredmény az első operandusban keletkezik. Nagyon fontos, hogy itt nincs bitszélesség kiterjesztés, tehát nincs átvitel 16 bites szorzás esetén **DX**-be, illetve 32 bites szorzás esetén **EDX**-be. A **C** és **O** flag mutatják, ha van elveszett átvitel.

Példa:

```
MOV AX, -2d     ;     AX = 0FFFEh
MOV BX, 5d      ;     BX = 5h
IMUL AX, BX     ;     AX = AX * BX = 0FFF6h
IMUL AX, 3h     ;     AX = AX * 3h = 0FFE2h
```

```
ADAT1     DD   912, 920, 928, 936, 944 ; adatszegmens tartalma
```

```
MOV EDX, offset ADAT1
IMUL AX, [EDX] ;     AX = AX * [EDX] = 9520h
               ;     (nincs átvitel, elvesszük az előjelet!)
               ;     átvitellel, helyesen FFFF 9520h lenne
```

IMUL három operandussal (80386-tól): az első operandus egy regiszter (16 vagy 32 bites), a második operandus lehet regiszter vagy memóriahivatkozás, a harmadik operandus egy közvetlen érték (konstans). A második és a harmadik operandus kerül összeszorzásra és az eredmény az első operandusban keletkezik. A két operandusos verzióhoz hasonlóan itt sincs bitszélesség kiterjesztés, viszont az **O** és **C** flag jelzik az elveszett átvitelt.

Példa:

```
MOV BX, 3d          ;     BX = 3h
IMUL AX, BX, 6d     ;     AX = BX * 6h = 3h * 6h = 0Ch
```

Osztás

DIV: A **DIV** (előjelnélküli osztás) utasítás használható 8, 16 és 32 bites verzióban is. Egyetlen operandusa van, az osztó, ami lehet egy regiszter vagy egy memória cím. A szintaxisa:

```
DIV reg/mem8

DIV reg/mem16

DIV reg/mem32
```

A következő táblázat mutatja be a kapcsolatot az osztandó, az osztó, a hányados és a maradék között:

Osztandó	Osztó (operandus)	Hányados	Maradék
AX	reg/mem8	AL	AH
DX:AX	reg/mem16	AX	DX
EDX:EAX	reg/mem32	EAX	EDX

A táblázatban leírtak értelmezése: Ha **8** bites az operandus, akkor az eredmény úgy áll elő, hogy az **AL** tartalmazza az **AX/op** hányadosát, a **AH**-ba pedig az **AX/op** maradéka kerül. Ha az operandus **16** bites, akkor **AX**-be a **(DX:AX)/op** hányadosa, **DX**-be pedig a (**DX:AX**)/**op** maradéka kerül. Ha az operandus **32** bites, akkor **EAX**-be az (**EDX:EAX**)/**op** hányadosa, míg az **EDX**-be az (**EDX:EAX**)/**op** maradéka kerül.

Példa:

A következő 8 bites előjelnélküli osztás esetén (83h/2h), a hányados értéke 41h, a maradék pedig 1h:

```
MOV AX, 0083h  ; AX = 0083h (osztandó)

MOV BL, 2h     ; BL = 2h (osztó)

DIV BL         ; AL = 41h (hányados), AH = 01h (maradék)
```

A következő 16 bites előjelnélküli osztás esetén (8003h/100h), a hányados értéke 80h, a maradék pedig 3h. **DX** tartalmazza a felső 16 bitjét az osztandónak, ezért mindenképp ki kell nullázni még a **DIV** utasítás végrehajtása előtt:

```
MOV DX, 0      ; DX = 0000, osztandó nullázása (felső 16 bit)
MOV AX, 8003h  ; AX = 8003h, osztandó (alsó 16 bit)
MOV CX, 100h   ; CX = 100h, osztó
DIV CX         ; AX = 0080h (hányados), DX = 0003h (maradék)
```

A következő 32 bites előjelnélküli osztás memória operandust használ, mint osztó:

```
.data
osztando  QWORD 0000000800300020h  ;    64 bites érték
oszto     DWORD 00000100h
.code
MOV EDX, DWORD PRT osztando + 4    ;    felső 32 bit
MOV EAX, DWORD PTR osztando        ;    alsó 32 bit
DIV oszto ; EAX = 08003000h(hányados),EDX = 00000020h(maradék)
```

IDIV: Az előjeles osztás közel megegyezik az előjelnélkülivel, egy fontos különbséggel: az osztandó előjel kiterjesztett kell, hogy legyen még az osztás elvégzése előtt. Az előjel kiterjesztés azt jelenti, hogy átmásoljuk a szám legfelső bitjét az összes felsőbb bit helyére az adott változóban vagy regiszterben. Ahhoz, hogy bemutassuk ennek a fontosságát, próbáljuk ki mi történik, ha kihagyjuk ezt a lépést. A következő kódban a **MOV** utasítással hozzá rendeljük a -101 értéket az **AX** regiszterhez, ami az **EAX** regiszter alsó 16 bitje:

```
.data
szoValtozo     SWORD     -101
.code
MOV AX, szoValtozo       ;    AX = FF9Bh
MOV BX, 2                ;    BX = 2 (osztó)
IDIV BX                  ;    DX:AX/BX (előjeles művelet)
```

Igaz, hogy az **AX** regiszterben benne van az FF9Bh, viszont az osztáskor a **DX** értékét nem állítottuk be, így az bármi lehet. Így teljesen kérdéses, hogy milyen eredményt kapunk, de az biztos, hogy nem helyeset. Tehát az út a helyes megoldáshoz, hogy használjuk a **CWD** utasítást, ami az **AX** regisztert előjelhelyesen kiterjeszti **DX:AX**-re még az osztás végrehajtása előtt:

```
.data
szoValtozo      SWORD      -101 ;     009Bh
.code
MOV AX, szoValtozo         ;     AX = FF9Bh
CWD                        ;     DX:AX = FFFF FF9Bh
MOV BX, 2                  ;     EBX = 2 (osztó)
IDIV BX                    ;     AX = DX:AX/BX = FFFFFF9B/2 =FFCD
```

A már tárgyalt **MOVSX** parancs használata hasznos lehet ilyen műveletek esetén.

```
MOVSX EAX, AX        ;     EAX = FFFF FF9Bh
                     ;     AX előjelhelyes kiterjesztése EAX-re
```

Osztásnál előfordulhat, hogy a hányados nem fér el az adott regiszterben amiben tárolásra kerülne, ilyenkor azonnal abortál a program. Célszerű megelőző ellenőrzéseket végezni:

Nem nulla-e az osztó?

Nem túl nagy-e az osztó?

További műveletek

INC: Az **INC** (INCrement) utasításnak egy operandusa van. Az operandusában megadott értéket 1-gyel növeli. Nincs hatással az átvitelre.

Példa:

```
MOV AX, 5h      ;     AX = 5h
INC AX          ;     AX = 6h
```

DEC: A **DEC** (DECrement) utasításnak egy operandusa van. Az operandusában megadott értéket 1-gyel csökkenti. Nincs hatással az átvitelre.

Példa:

```
MOV AX, 5h      ;     AX = 5h
DEC AX          ;     AX = 4h
```

NEG: A **NEG** (NEGate) utasításnak egy operandusa van. Az operandusában megadott érték előjelét megfordítja, azzal, hogy az adott számot átkonvertálja a kettes komplemensére.

Példa:

```
MOV AX, 5h      ;     AX = 5h
NEG AX          ;     AX = FFFFFFFBh
```

ADC: Az **ADC** (ADd with Carry) utasítás hozzáadja a forrás operandust és a **Carry** flag tartalmát is a cél operandushoz. Az utasítás formátuma megegyezik az **ADD** utasításéval és mind két operandus méretének meg kell egyeznie. 64 bites összeadás esetén a következő módon használható fel az **ADC** parancs.

Példa:

```
MOV EAX, 0FFFFFFFFh      ; EAX = 0FFFFFFFFh
MOV EBX, 0FFFFFFFFh      ; EBX = 0FFFFFFFFh
```

```
MOV ECX, 11h            ; ECX = 11h
MOV EDX, 11h            ; EDX = 11h
ADD EAX, EBX            ; EAX = EAX + EBX = FFFFFFFEh, C = 1
ADC ECX, EDX            ; ECX = ECX + EDX + C = 23h
```

SBB: Az **SBB** (SuBtract with Borrow) utasítás kivonja a cél operandusból a forrás operandust és a **Carry** flag tartalmát is. A lehetséges operandusok listája megegyezik az **ADC** utasításéval. A következő példa 64 bites kivonás használatának lehetőségét mutatja be 32 bites regiszterek segítségével. Először beállítjuk az **EDX**:**EAX** regiszter tartalmát a 0000000700000001h értékre, majd kivonunk belőle 5-öt. Az alsó 32 bitek kerülnek kivonásra először, beállítódik a **Carry** flag értéke 1-re, majd a felső 32 bitből kivonódik a **Carry** flag értéke (és az **SBB** második operandusának tartalma):

Példa:

```
MOV EDX, 7      ; EDX = 00000007h
MOV EAX, 1      ; EAX = 00000001h
SUB EAX, 5      ; EAX = EAX - 5 = FFFFFFFCh, C = 1
SBB EDX, 0      ; EDX = EDX - C = 00000006h
```

CMP: A **CMP** (CoMPare) utasítás két operandusú művelet. Szintaxisa:

```
CMP op1, op2
```

A megadott két operandus értékét hasonlítja össze oly módon, hogy veszi az *op1-op2* művelet eredményét, és az eredménynek megfelelően állítja be a flag-eket. A későbbiekben feltételes vezérlésátadáshoz nagyon jó felhasználható lesz.

SHL: Az **SHL** (SHift Left) utasítás logikai léptetést végez balra a cél operanduson belül, az alsó biteket feltöltve nullával. A legmagasabb helyiértéken lévő bit bekerül a Carry flag-be, és ami addig a Carry flag-ben volt eldobódik. A forrás operandusban adhatjuk meg, hogy hány bittel szeretnénk arrább léptetni.

Példa:

```
MOV AL, 8Fh     ; AL = 10001111b
SHL AL, 1       ; AL = 00011110b mivel 1 bittel balra tolódott
                ; C = 1
```

SHR: Az **SHR** (SHift Right) utasítás logikai léptetést végez jobbra a cél operanduson belül, a felső biteket feltöltve nullával. A legalacsonyabb helyiértéken lévő bit bekerül a Carry flag-be, és ami addig a Carry flag-ben volt eldobódik. A forrás operandusban adhatjuk meg, hogy hány bittel szeretnénk arrább léptetni.

Példa:

```
MOV AL, 0D0h    ; AL = 11010000b
SHR AL, 1       ; AL = 01101000b mivel 1 bittel jobbra tolódott
                ; C = 0
```

ROL: A **ROL** utasítás hasonlít az **SHL** utasításhoz, azzal a különbséggel, hogy itt körkörösen mozognak a bitek balra. A legmagasabb helyiértéken lévő bit beleíródik a Carry flag-be is és a legalacsonyabb helyiértékű bit helyére is.

Példa:

```
MOV AL, 8Fh     ; AL = 10001111b
ROL AL, 1       ; AL = 00011111b mivel 1 bittel balra tolódott
                ; körbe. C = 1
```

ROR: A **ROR** utasítás hasonlít az **SHR** utasításhoz, azzal a különbséggel, hogy itt körkörösen mozognak a bitek jobbra. A legalacsonyabb helyiértéken lévő bit beleíródik a Carry flag-be is és a legmagasabb helyiértékű bit helyére is.

Példa:

```
MOV AL, 0D1h    ; AL = 11010001b
ROR AL, 1       ; AL = 11101000b mivel 1 bittel jobbra tolódott
                ; körbe. C = 1
```

Egy gyakorlatiasabb példa az **SHL** és a **ROL** utasítás használatára:

```
MOV AL, -80h    ;    AL = 80h
CBW             ;    AX = FF80h
MOV BX, 1234h   ;    BX = 1234h
IMUL BX         ;    DX:AX = FFF6 E600h
```

A szorzás elvégzése után a szorzat a **DX**:**AX** regiszterpárosba kerül. Ha kiszeretnénk íratni a szorzatot, akkor mindenképp jó lenne elhelyezni előbb az **EAX** regiszterben. Erre több lehetőség is van. A következő példa az **SHL** utasítás használatát mutatja be erre a célra:

```
PUSH AX         ;    AX tartalma a verembe
MOV AX, DX      ;    AX = FFF6h
SHL EAX, 16     ;    EAX = FFF6 0000h
POP AX          ;    EAX = FFF6 E600h
```

Majd ugyancsak a **DX**:**AX** tartalmának **EAX**-be való áthelyezése a **ROL** utasítás segítségével:

```
ROL EAX, 16     ;    EAX = E600 0000h
MOV AX, DX      ;    EAX = E600 FFF6h
ROL EAX, 16     ;    EAX = FFF6 E600h
```

Példa adatszegmens gyakorláshoz

A alábbi adatszegmensben található változókat felhasználva hajtsunk végre tetszőleges műveletek kombinációiból alkotott utasítássorozatokat.

Pl.: `A+B, D-A, E*F, H+E*F, (H+I)/D, (L-I)/A, H*E-J, ...`

```
.data ; adatszegmens
     TestString     BYTE "Aritmetika!", 0ah, 0

     A          BYTE      100d
     B          BYTE      -80h
     D          BYTE      200d
     E          BYTE      5d

     F          WORD      1234h
     G          WORD      -4h
     H          WORD      30000d
     I          WORD      40000d

     J          DWORD     100000d
     K          DWORD     1000h
     L          DWORD     4000000d
     M          DWORD     15h

     ADAT1      WORD      912, 920, 928, 936, 944
     ADAT2      BYTE      11, 15, 20
```