

Agárdi Gábor

IBM PC

*Gyakorlati
Assembly*

Lemez melléklettel



AGÁRDY GÁBOR

IBM PC

**GYAKORLATI
ASSEMBLY**

**Nyitott rendszerű képzés
– távoktatás – oktatási segédlete**

TANKÖNYV

**LSI Oktatóközpont
A Mikroelektronika Alkalmazásának
Kultúrájáért Alapítvány**

Lektorálta: Kovács István

I S B N 963 577 117 7

Kiadó: **LSI Oktatóközpont**

Felelős vezető: **Dr. Kovács Magda**

Témafelelős: **Flier István**

LIGATURA KFT - VÁCI ÁFÉSZ NYOMDA

Munkaszám: 264-94

Tartalomjegyzék

Előszó	5
Az assembly programozás alapjai	7
Hogyan közelítsük meg a gépkódot	9
A gépkód és az assembly kapcsolata	9
Mi is az a programozás	9
A programozás eszközei	10
A regiszterek működése	11
A PC címképzése	12
Hogyan kezdünk el megírni egy programot	13
EXE programszövegek felépítése	14
COM programszövegek felépítése	14
Kilépés a DOS-ba	15
A MOV utasítás használata	16
Egy betű kiírása a képernyőre	19
Pozícionált betűkiírás	20
Pozícionált szövegkiírás	22
A logikai műveletek működése	25
A logikai műveletek vizsgálata	30
Az adatforgató műveletek vizsgálata	34
Hexadecimális számok kiírása	37
Egy szám decimális kiírása	40
Real-Time idő kiírása	42
Menükezelés	45
Memóriakezelés	52
File kezelés	53
Memória és file kezelő program	54
Szöveg kereső program	58
Paraméter átadása a DOS-ból	64
A PC hangkezelése	69
A memória rezidens programok működése	71
Hangos drive	72
Assembly Grafika	93
Grafikus üzemmódok programozása	93
A CGA üzemmód lehetőségei	93
CGA Szinbeállítás	94

Egy pont kirakása	96
Grafikus ábra kirakása	99
Egérkezelés	104
A CGA kártya belső regiszterei	116
EGA és VGA üzemmódok programozása	117
EGA színbeállítás	118
VGA színbeállítás	123
EGA VGA 0. írási mód CPU adattal	127
EGA VGA 0. írási mód SET-RESET adattal	129
EGA VGA 1. írási mód	129
EGA VGA 2. írási mód	130
VGA 3. írási mód	131
EGA VGA 0. olvasási mód	131
EGA VGA 1. olvasási mód	132
Az EGA kártya grafikus vezérlőregiszterei	132
Szövegkirakás VGA grafikus képernyőn	134
Ábrakirakás színbeállítással a 2 színű VGA üzemmódba	137
Az MCGA üzemmód működése	141
MCGA ábrakirakás színbeállítással	141
MCGA egérkezelés	146
Rajzolás a képernyőre	154
MCGA vonalrajzoló eljárás	155
MCGA Scroll rutinok	161
MCGA körrajzoló rutin	171
Képformátumok	175
Az LBM formátum felépítése	176
Egy LBM kép megjelenítése	177
Az Ssvga üzemmódok tulajdonságai	183
Ssvga üzemmódok különböző grafikus kártyáknál	183
Ssvga rajzolás közvetlen memóriakezeléssel	188
Ssvga rajzolás a ROM BIOS segítségével	192
Függelék	195
A PSP felépítése	195
A 10h megszakítás lehetőségei	197
Tárgymutató	210

ELŐSZÓ

Ebben a könyvben megpróbálom egy kicsit másképpen megközelíteni a gépi kódú programozást mint ahogyan az az eddig megjelent hasonló témájú kiadványokban történt. Nem elsősorban a programozás elméleti részével kívánok foglalkozni, hanem sokkal inkább annak gyakorlati oldalával. Céлом az, hogy mindenki, aki ezt a könyvet elolvassa, szert tehessen egy olyan alaptudásra, amit már saját magától tovább tud fejleszteni abba az irányba, amire azt fel kívánja használni. Ahhoz, hogy valaki assembly nyelven programozzon lényeges előnyt jelent valamilyen magasabb szintű (BASIC, PASCAL, stb.) nyelv alapfokú ismerete, mivel így az itt leírtak megértése sokkal egyszerűbbé válik. Annak, aki még soha nem programozott számítógépet azt tanácsolom, hogy tegye vissza a polcra ezt a könyvet, és próbálkozzon az előbb említett nyelvek valamelyikének megismerésével, így biztosan sokkal hamarabb lesz sikerélménye. Annak pedig, aki úgy dönt, hogy mégis belevág, annak sok sikert kívánok.

Agárdi Gábor

AZ ASSEMBLY PROGRAMOZÁS ALAPJAI

Az assembly az, ahol nem elég csupán a nyelv ismerete hanem a számítógép belső felépítéséről, működéséről is viszonylag részletes információval kell rendelkezünk.

Miből is áll a számítógép:

A mi számunkra legfontosabb elemek a következők:

- CPU (Central Processing Unit) Processzor
- RAM (Random Access Memory) Írható-olvasható memória

Természetesen egy számítógép jóval több részből áll, de a kezdéshez számunkra e kettő ismerete elegendő és feltétlenül szükséges. Mivel egy program nem más, mint a memóriában tárolt utasítások egymásutánja, amit a processzor értelmez és végrehajt.

A CPU a számítógépnek az a része, amelyik a memóriában tárolt adatokat kiolvassa, értelmezi és végrehajtja. A CPU-ban helyezkedik el egy másik egység, az ALU (Arithmetical Logical Unit) mely a különböző aritmetikai (összeadás, kivonás, szorzás, osztás, forgatás stb.) és logikai (és, vagy, nem, stb) műveleteket végzi. Továbbá itt kapott még helyet néhány kiemelt adattároló, melyeket regisztereknek neveznek és amiket a későbbiekben ne tévesszünk össze a változókkal. Ezek előnye, hogy a műveletvégzés velük sokkal gyorsabb, mint a memória közvetlen elemeivel, valamint az utasítások többsége ezeken keresztül kap adatokat illetve ezekben kapjuk a műveletek eredményét.

A RAM egy olyan elsődleges adattároló eszköz, melyben a programok és legtöbb esetben a feldolgozandó adatok tárolása történik. A tárolási rendszer kialakításához a matematika kettes (bináris) számrendszerét vették alapul, ugyanis itt az egyes helyi értékek mindössze kétféle értéket vehetnek fel a 0-át és az 1-et. Ez olyan, mint egy kapcsoló ki- illetve bekapcsolt állapota. A számítógép memóriája is ilyen kapcsolókból épül fel, a benne tárolt adatot pedig a kapcsolók helyzete határozza meg. Egy ilyen

kapcsolót neveznek a számítástechnikában bitnek. Minden nagyobb egység erre fog felépülni. A programozás során kevesebbet fogjuk ugyan használni a kettes számrendszert, de egyes utasítások megértéséhez feltétlen szükséges. A memória elérése szempontjából nem lenne előnyös azt bitenként kezelni, ezért a biteket csoportokba szedjük, és ilyen formában dolgozzuk fel. Minden egyes csoport 8 bitből áll, és egy ilyen egységet hívnak byte-nak. A memória ugyan bitekből épül fel, de a legkisebb elérhető (olvasható, vagy írható) része a byte, ami nem más mint 8 egymás melletti bit. Egy ilyen 8 helyértékű bináris számon a legnagyobb tárolható szám a 255. A későbbiekben szó lesz még a memória nagyobb egységeiről is. Ilyen egység a word (szó), ami két egymás melletti byte-ból (16 bit) áll, és a dword (kettősszó), ami pedig két szomszédos word-öt (32 bit) fog össze.

Tehát a memória legkisebb egysége elérés szempontjából a byte, ami azt jelenti, hogy a memóriába csak a byte-okat tudjuk írni illetve olvasni, az egyes biteket csak úgy tudjuk állítani, hogy beírunk a memóriába egy olyan byte-ot, aminek a kívánt bitje a megfelelő értékű.

A memória byte-os szervezésű, azaz egymás után álló byte-ok sorozatából áll. Az egyes memóriabyte-okat úgy érhetjük el, hogy minden byte-hoz egy cím van hozzárendelve és íráskor vagy olvasáskor ezen cím segítségével hivatkozhatunk rá. A processzor is ezen címek alapján éri el az egyes byte-okat, úgy, hogy egy címbuson és egy adatbuszon keresztül kapcsolódik a memóriához, ami valójában nem más mint néhány vezeték (8 , 16, vagy 32 db.) és még egy kis elektronika. A címbuson megjelenő érték választja ki, hogy melyik memóriabyte-ot akarjuk elérni és az így kiválasztott byte az adatbuszon fog mozogni.

Egy processzor memóriakapacitását az szabja meg, hogy milyen széles (hány bites) a címbusza. Egy IBM XT-nek például 20 bites címbusza van, ami azt jelenti, hogy maximum 2^{20} azaz 1048576 byte-ot (1Mbyte-ot) képes megcímezni. Ez egy 386-os AT gépnél, ahol 32 bites címbusz van (ez 4Gbyte). És mivel az adatbusza is 32 bites, képes az adatokat dword-ösen elérni. Ez azt jelenti, hogy egyszerre nem csak 1 byte-ot tud írni vagy

olvasni, hanem 4-et. De azért, hogy az XT-re írt programok magasabb szintű gépeken is futtathatóak legyenek, a számítógép képes az adatbuszát átállítani az XT-nek megfelelőre ekkor úgy viselkedik, mint egy XT csak egy árnyalatnyival gyorsabb.

Hogyan közelítsük meg a gépikódot:

Egy számítógépben az összes tulajdonsága mellett az az egyik legcsodálatosabb, hogy bármit megvalósíthatunk vele amit csak el tudunk képzelni. Természetesen ma még elég sok akadály állja útját szárnyaló fantáziánknak, de a számítástechnika és a programozás fejlődésével egyre kevesebb a megoldhatatlan probléma.

A gépikódú programozás a programozók legnagyobb fegyvere (már akik ismerik), ugyanis csak gépikódban lehet a számítógép minden adottságát teljes mértékben kihasználni. Mindezek mellett ennek a nyelvnek a megtanulása a legnehezebb mivel ahhoz, hogy valaki képes legyen gépikódú programokat írni, nem elég csak magát a programozási nyelvet megtanulnia, ismernie kell magát a számítógépet is.

A gépikód és az assembly kapcsolata:

A gépi kódú nyelv tulajdonképpen számokból áll. Ezek a számok tárolódnak a memóriában és ezeket értelmezi illetve hajtja végre a processzor. Azonban az utasításokat jelképező összes szám bemagolása egy kicsit száraz lenne, így kitaláltak egy jelképes nyelvet, melyet úgy alkottak meg, hogy csoportokba szedték az egyes utasításokat és ezeknek a csoportoknak könnyen tanulható neveket adtak. Ezekből az utasításokból álló nyelvet nevezték el ASSEMBLY-nek.

Mi is az a programozás:

Egy program nem más, mint a megvalósítandó feladat a gép nyelvén. Amihez úgy juthatunk el, hogy a feladatot elkezdjük felbontani egységekre, majd azokat tovább olyan apró részekre, melyek már helyettesíthetők a gép szavaival, utasításaival. Egy-egy program megírását mindig különböző szempontok

irányítják, melyek nagy mértékben függenek az adott feladattól, de vannak olyan szempontok is, melyek általánosíthatók. Ezek határozzák meg a számítógépes programozás jellegét, és ezek miatt szükséges az programozáshoz más gondolkodás, mint ami általában megszokott. Ezek az általánosítható szempontok alkotják egyben a programozás eszközeit, melyek más megvilágításba helyezik a feladatot.

A programozás eszközei:

Eljárások: ezek olyan programrészek melyek a feladatban többször előforduló, ismétlődő folyamatokat takarnak. Ilyenkor ezt csak egyszer írja meg az ember úgy, hogy bárholnan elérhető és ha szükséges, megfelelően paraméterezhető legyen. Ilyen eset például egy keretrajzolás, ahol bemenő paraméternek megadjuk a keret bal felső sarkának koordinátáit, szélességét és magasságát. Végrehajtva az eljárást, a megfelelő méretekkel kirajzolja a keretet.

Változók: Egy olyan eszközei a programozásnak, melyek segítségével adatokat tárolhatunk a memóriában. Bizonyos feladatok megoldása elképzelhetetlen lenne változók használata nélkül mivel a regiszterek száma korlátozott, nem tárolhatunk minden információt ezekben, de mivel a memória mérete nagyságrendekkel nagyobb, ezért kijelölhetünk részeket ahol a számunkra fontos információt tárolhatjuk. Ezen információ nagyon sokféle lehet (szám, szöveg, kép, stb.). Ezek, mint azt később látni fogjuk, csak felhasználásukban különböznek, tárolásukban nem.

Ciklusok: Ez egyike a legfontosabb eszközöknek, ugyanis ezek segítségével a feladatban egymás után többször ismétlődő műveleteket egyszerűsíthetünk le. Ha például az a feladat, hogy írjuk ki 10-szer egymás után a nevünket a képernyőre, akkor azt meg lehet oldani úgy is, hogy tízszer egymás után megírjuk az adott programot, de úgy is, hogy csak egyszer írjuk meg és egy ciklus segítségével tízszer egymás után lefuttatjuk. Az eredmény ugyanaz, csak a program hossza az utóbbi esetben töredéke a másiknak, emellett egyszerűbbé, áttekinthetőbbé is válik.

Elágazások, feltételek vizsgálata: Talán ez az egyik legegyszerűbb eszköze a programozásnak, ugyanis számtalan olyan eset van, amikor meg kell vizsgálni egy művelet eredményét, egy visszaérkező választ és ennek megfelelően választani kell adott lehetőségek közül.

A regiszterek működése:

A regiszterek között vannak általános rendeltetésűek illetve speciálisak. Az általános rendeltetésűek az AX, BX, CX, DX neveket viselik. Ezekbe 0-65535-ig bármilyen számot beírhatunk. Egy ilyen regiszter tulajdonképpen nem más mint egy 16 bites adat tárolására alkalmas rekesz. Ezeket használhatjuk 16 illetve 2*8 bitesként, mivel egy ilyen rekesznek van egy alsó illetve egy felső része. Ez például az AX regiszternél AL illetve AH. Az AL az alsó AH a felső rész. Ezt a következő táblázatból könnyen megérthetjük:

Decimális	Bináris		Hexadecimális
AX	AH	AL	AH AL
46	00000000	00101110 b	00 2E h
5	00000000	00000101 b	00 05 h
1649	00000110	01110001 b	06 71 h
65536	11111111	11111111 b	0F FF h

Mint azt láthatjuk, alapértelmezés szerint decimális számokat használunk. Ettől eltérő esetben jelölni kell a szám típusát, továbbá ha egy hexadecimális szám betűvel kezdődik, elé egy vezető nullát kell tenni, tehát a hexadecimális számnak mindig számjeggyel kell kezdődnie.

Egy 16 bites számmal megcímezhető legnagyobb memória cím értéke 65535. Egy számítógépben azonban nem csak 65535 byte memória lehet, hanem több. A legkevesebb ami egy XT-ben lenni szokott, az 512 Kbyte de ennél általában több (640 Kbyte, 1 Mbyte) a gép almemóriája. Hogy ne csak 64 Kbyte legyen elérhető egy gép számára, ezért találták ki a **szegmens-** és az **indexregisztereket** (az indexregisztereket esetenként **offsetregiszternek** nevezik). Ezek lényege az, hogy a rendelkezésre álló memóriából kiválaszthatunk egy 64 Kbyte

méretű szegmenst amit már képes kezelni a processzor. Ahhoz, hogy ezt megértsük, legegyszerűbb a számítógép memóriáját úgy elképzelni, mint egy nagy könyvet, amiből olvasni, illetve amibe írni szeretnénk. Mivel az egész könyvet egyszerre nem láthatjuk, ki kell nyitnunk azt valamelyik oldalon. Ez az oldal jelképezi azt a szegmenst, amivel egyidőben dolgozni tudunk. A szegmensregiszter határozza meg, hogy melyik memóriarészlettel foglalkozunk (melyik oldalon van nyitva a könyv), az indexregiszter pedig azt mutatja, hogy a kijelölt részleten belül melyik címen (az oldal melyik sorában) van a szükséges adat. A szegmensek egymástól 16 byte (paragrafus) távolságra lehetnek, így egy címet összesen 20 biten ábrázolhatunk. Ez a megoldás átfedést okoz az egyes szegmensek között. Egy cím meghatározásához szükséges 20 bites szám, a következőképpen néz ki:

Tegyük fel, hogy az 5A09Dh címet szeretnénk elérni. Ebben az esetben a regiszterek értékei így alakulnának:

Szegmensregiszter: 0101110000001001b (5A09h)
 Indexregiszter: 0000000000001101b (000Dh)

A cím: 01011100000010010000b (5A090h)
 + 0000000000001101b (000Dh)
 = 01010110000010011101b (5A09Dh)

Ugyanezt érhetjük el így is:

Szegmensregiszter: 0101011010110101b (56B5h)
 Indexregiszter: 0011010101001101b (354Dh)

A cím: 01010110101101010000b (56B50h)
 + 0011010101001101b (354Dh)
 = 01010110000010011101b (5A09Dh)

A megoldás hátránya, hogy a memóriában csak un. paragrafus határon kezdődhet egy szegmens. A hiányosság oka, hogy a cím alsó 4 bitjét nem, a szegmensregiszter határozza meg. De az így kiválasztott lapon belül már minden byte elérhető az indexregiszter segítségével. A szegmensregiszterek szerepe egy kivételével meghatározott, ettől eltérni nem célszerű:

Az Assembly programozás alapjai

CS	Code Segment	Kódszegmens
DS	Data Segment	Adatszeglens
ES	Extra Segment	Extraszegmens
SS	Stack Segment	Veremseglens

Az indexregiszterek pedig:

SI	Source Index	Forrásindex
DI	Destination Index	Célindex

Utóbbiak szerepe szintén előre meghatározott, de ha a feladat megkívánja, nyugodtan eltérhetünk tőle.

A processzornak van még további két 16 bites regisztere a **BP** BasePointer (bázismutató) illetve a **Flag** regiszter. Ez utóbbinak külön szerepe van, mivel az egyes bitjeinek jelentése meghatározott de erről később.

Hogyan kezdünk el megírni egy programot:

Az assembly programok írásához szükség van egy szövegszerkesztőre -ez bármilyen DOS formátumú szerkesztő lehet (például Norton Editor)- és egy fordító programra (például MASM, TASM stb.), illetve a hozzá tartozó linkerre, ami az előkészített állományból előállítja a futtatható programot. Tanácsolom a TASM illetve TLINK használatát mivel a könyvben szereplő mintaprogramok is ezzel készültek. A későbbiekben nem árt, ha beszerzünk még egy Debugger programot (AFD, TD stb.) is, ami sokat segíthet az esetleges hibák kijavításában. A program írásának menete a szövegszerkesztőben kezdődik, ahol elkészítjük annak forrásszövegét. A kész szövegfile-t (.ASM) első lépésben lefordítjuk egy objectfile-ra (.OBJ) és ezután EXE vagy COM típusú futtatható programot készítünk belőle a linker segítségével. Egy program felépítésének különféle formai szabályai vannak.

Egy .EXE programnál a forrás valahogy így fest:

Kód **Segment**
assume CS:Kód, DS:Adat, SS:Stack

Start:

Kód **Ends**

Adat **Segment**

.
.
.

Adat **Ends**

Stack **Segment**

.

Stack **Ends**

End Start

Egy .COM programnál pedig:

Kód **Segment**
assume CS:Kód, Ds:Kód
Org 100h

Start:

Kód **Ends**

End Start

A különbség a COM és az EXE program között, hogy míg az EXE bármilyen hosszú lehet, a COM-nak bele kell férnie egy szegmensbe tehát nem lehet 64Kbyte-nál hosszabb. A programszövegekben a **Segment** jelöli a szegmens kezdetét aminek az előtte álló címke a neve, ami tetszés szerint bármi lehet. Az adott szegmens végét az **Ends** jelzi. Az **assume** szerepe, hogy a szegmensregiszterekbe a hozzátartozó szegmenscímet töltsse. Lehetőség van egy kis egyszerűsítésre is, mivel az **SS** regiszternek nem kötelező értéket adni tehát ezt a címkét elhagyhatjuk és az **assume** sorból is törölhetjük. Ezenkívül ahogyan az a COM felépítésnél látható, **CS** értéke megegyezhet **DS** értékével. Ilyenkor ugyan az lesz a kód illetve adatszegmensünk. Az **Org** szerepe, hogy meghatározza a program kezdőcímét a szegmensen belül. Ezt célszerű 100h-nak választani, ugyanis az ez alatti memóriaterületen az operációs rendszer programunkra vonatkozó paraméterei, és adatai vannak. Az EXE programoknál van még egy fontos dolog, hogy az **assume** a **DS** regiszterbe nem azt az értéket tölti, amit mi később használni szeretnénk, így a programunkban külön be kell azt állítani, de erről majd később.

A feladatok megoldása során van még egy segítségünk, ugyanis a számítógép tartalmaz egy ROM BIOS feliratú alkatrészt amiben előre elkészített programok vannak különféle feladatok megoldására. Ezek használata az első időkben egyszerűbbé teheti a programozást, későbbiekben azonban próbáljuk meg egyre ritkábban használni.

Az első mintaprogram mindössze annyit fog csinálni, hogy ha elindítjuk, ő visszatér a DOS-hoz egy BIOS rutin segítségével.

[Program 1]

```
Pelda01      Segment                ;Szegmensdefiníció.  
              assume Cs:Pelda01, Ds:Pelda01 ;Cs és ds regiszterek beállítása  
              ;a szegmens elejére.  
  
Start:       mov  ax,Pelda01         ;A ds regiszter beállítása.  
              mov  ds,ax
```

IBM PC Gyakorlati Assembly

	mov ax,4c00h	;Kilépés a DOS-ba
	int 21h	
Pelda01	Ends	;A szegmens vége.
	End Start	;A program vége.

Amint az látható, lehetőség van a programszövegben megjegyzések elhelyezésére egy pontosvessző után. Az így elhelyezett szöveget a fordító nem értelmezi.

A **mov** utasítás segítségével adatokat mozgathatunk egy forrásból egy célba, például így lehet értéket adni az egyes regisztereknek. A **mov** utasítást két paraméter követi, először a cél majd egy vesszővel elválasztva a forrás. Ha így egy címkét írunk az adat helyére, akkor a címke szegmenscímét fogja a regiszterbe írni. A **mov ds,ax** sorra azért van szükség, mivel a **ds**-be közvetlenül nem tölthetünk adatot, csak egy másik regiszteren keresztül. A **mov** utasítás segítségével lehetőségünk van egy regiszterbe egy számot, egy másik regiszter értékét, egy címke szegmens illetve offsetcímét vagy egy memóriarekesz tartalmát tölteni vagy fordítva.

mov	ax,42527	;Ax regiszterbe tölti a 42527 számot.
mov	ax,52h	;Ax regiszterbe tölti a 52h (82) számot.
mov	al,62	;Al regiszterbe tölti a 62 számot.
mov	ax,bx	;Ax regiszterbe tölti bx értékét.
mov	al,bh	;Al regiszterbe tölti bh értékét.
mov	ax,cimke	;Ax regiszterbe tölti a címke szegmenscímét
mov	ax,word ptr [cimke]	;Ax regiszterbe tölti a címke 16 bites ;tartalmát.
mov	al,byte ptr [cimke]	;Ax regiszterbe tölti a címke 8 bites ;tartalmát.
mov	ax,[si]	;Ax regiszterbe tölti az indexregiszter ;által mutatott 16 bites értéket.
mov	al,[si]	;Al regiszterbe tölti az indexregiszter ;által mutatott 8 bites értéket.
mov	ax,[si+2]	;Az indexregiszter+2 cím által ;mutatott adat kerül ax-be.
mov	ax,[si+bx+2]	;Mint előbb, de si+bx+2 címen levő.
mov	ax,es:[si]	;Ha nem az alapértelmezés szerinti ds ;által mutatott szegmensből kívánunk ;adatot tölteni a regiszterbe, akkor azt így ;kell jelölni.

Az Assembly programozás alapjai

mov	ax,es:[si+2]	;Az es szegmens si+2 által mutatott címén ;lévő adat kerül ax-be.
mov	ax,es:[si+bx+2]	;Az es szegmens si+bx+2 által mutatott ;címén lévő adat kerül ax-be.
mov	ax,offset címke	;A címke szegmensen belüli "offset" címe ;kerül ax-be.
mov	word ptr [címke],ax	;Ax tartalmát a címke által mutatott helyre ;írja
mov	byte ptr [címke],al	;Al tartalmát a címke által mutatott helyre ;írja
mov	[si],ax	;Ax tartalmát ds:si által mutatott helyre ;írja
mov	[si],al	;Al tartalmát ds:si által mutatott helyre ;írja
mov	[si+2],ax	;Ax tartalmát ds:si+2 által mutatott helyre ;írja
mov	[si+bx+2],ax	;Ax tartalmát ds:si+bx+2 által mutatott ;helyre írja
mov	es:[si],ax	;Ax tartalmát es:si által mutatott helyre ;írja
mov	es:[si+bx],ax	;Ax tartalmát es:si+bx által mutatott helyre ;írja
mov	es:[si+bx+2],ax	;Ax tartalmát es:si+bx+2 által mutatott ;helyre írja

Mint az látható, elég sokféle variációs lehetősége van ezen egyszerű utasításnak, hogy mégis könnyebb legyen ezeket megtanulni, egy pár alapszabály: a **mov** utasítás utáni 2 operandus közül legalább az egyiknek regiszternek kell lenni. Ha memóriában levő adatra hivatkozunk, a címet mutató regiszter vagy címke mindig szögletes zárójelben van. ezenkívül ha a címet nem regiszter hanem egy címke határozza meg, elé kell írni az elérendő adat típusát (word ptr, byte ptr, dword ptr). Ezenkívül a két operandus típusának meg kell egyeznie (például ha a forrás 8 bites akkor a célnak is 8 bitesnek kell lennie). A **mov ax,52h** sornál ez egy kicsit csalóka, de a gép ezt **mov ax,0052h** alakban tárolja. Ha ezeket az alapszabályokat betartjuk, különösebb probléma nem történhet, ha mégis elkövetünk valami nagyobb bakit, azt a fordító program jelezni fogja.

Amint azt említettem, ez a program hívás után visszalép a DOS-hoz. A ROM BIOS szolgáltatásait az int utasítással lehet elérni. Az utána levő szám egy beépített rutin hivatkozási száma, melynek memóriacímét egy táblázatból olvassa ki, az ax-ben átadott érték pedig egy úgynevezett bemenő paraméter. Erre azért van szükség, mert ez a rutin sokkal többre képes annál, minthogy visszatérjen az operációs rendszerhez, de hogy melyik szolgáltatását szeretnénk igénybe venni, azt ennek segítségével kell meghatározni. Azzal, hogy az ax regiszterbe 4c00h értéket írunk, arra utasítottuk, hogy hibaüzenet nélkül lépjen ki. Ez a két sor csaknem minden programban szerepel. Ezzel elkészült első működő programunk.

Ahhoz, hogy továbbléphessünk, meg kell ismerni a karakteres képernyő felépítését mivel a következő program a képernyőre való írást mutatja be. A képernyőn látott szöveg nem más mint a memóriának egy része megjelenítve. Egy betű kiírása karakteres képernyőn úgy történik, hogy a betű kódját (ASCII kódját) és színét a képernyő-memória megfelelő helyére írjuk. Ennek a memóriának is van egy szegmenscíme, ami jelen esetben 0b800h. Ez a képernyő bal felső karakter pozíciójának címe. Ha nem ebbe a sarokba kívánunk írni, akkor a szegmenscímhez hozzá kell adni egy eltolási (offset) értéket. Ennek értékét könnyen ki lehet kiszámolni, mivel tudjuk, hogy egy karakterhez 2 byte tartozik, az első a karakter kódja, a második a színkód. Megnézzük, hogy egy sorban hány betű fér el (a mintaprogramnál 80), ezt megszorozzuk 2-vel, így megkaptuk egy sor hosszát. Ha például a 7. sor 13. oszlopába kívánunk írni, annak címe 80 karakteres módban $7 \cdot 80 \cdot 2 + 13 \cdot 2 = 1146 = 47Ah$ tehát a teljes cím: **0B800h:47Ah**. Így bármely pozíció címét ki tudjuk számolni. Ezután már csak az a dolgunk, hogy a kiszámolt címre írjuk a kívánt adatot. Itt kell figyelembe venni, hogy a 16 bites regiszternek az alsó 8 bitje tárolódik előbb és utána a felső rész. Így a színkódot az ah a karakterkódot az al regiszterbe kell tenni (vagy bármelyik másikba, de ilyen sorrendben). A színkód byte minden külön bitjének megvan a maga jelentése:

- 0.bit:** Az előtér kék színösszetevője,
- 1.bit:** Az előtér zöld színösszetevője,
- 2.bit:** Az előtér piros színösszetevője,
- 3.bit:** Az előtér intenzitása,
- 4.bit:** A háttér kék színösszetevője,
- 5.bit:** A háttér zöld színösszetevője,
- 6.bit:** A háttér piros színösszetevője,
- 7.bit:** Villogás ki- bekapcsolása (a bit 1 értéke jelenti a villogást).

A kívánt színt úgy tudjuk előállítani, hogy a megfelelő színeket összekeverjük. A 8 féle előtérszín kiegészíti egy intenzitás bit amit ha 1 állapotba állítunk, a szín fényesebb lesz. A háttér színét egy villogás bittel egészítették ki, amit ha bekapcsolunk, a betű villogni fog.

[Program 2]

Pelda02	Segment assume cs:Pelda02,ds:Pelda02	;Szegmensdefiníció. ;Cs és ds regiszterek beállítása a szegmens elejére.
Start:	mov ax,Pelda02 mov ds,ax	;A ds regiszter beállítása.
	mov ax,0b800h mov es,ax	;A képernyő-memória szegmens-címét es regiszterbe tölti.
	mov di,1146	;A di indexregiszterbe beállítja az offsetcímet.
	mov al,"A"	;Al regiszterbe az "A" betű ascii kódját tölti.
	mov ah,7	;A betű színét fekete alapon fehér színűre állítja.
	mov es:[di],ax	;Az es:di által mutatott címre írja ax tartalmat azaz a fekete alapon fehér "A" betűt.

IBM PC Gyakorlati Assembly

```
mov ax,4c00h ;Kilépés a DOS-ba.
int 21h

Pelda02      Ends ;A szegmens vége.
             End Start ;A program vége
```

A program eleje és vége azonos az előzővel, de közé lett iktatva a betű kiíratása. Első lépésben beállítjuk a képernyőmemória szegmencímét az `es` szegmensregiszterbe. Amint azt a `ds` állításánál is tapasztalhattuk, a szegmensregisztereket közvetlenül nem lehet állítani, így az `ax` regiszteren keresztül írjuk bele a megfelelő értéket. Ezután a már említett offset értéket a `di` indexregiszterbe tesszük és az így kialakuló memóriacímre kiírjuk `ax` értékét majd visszatérünk a DOS-hoz. A következő lépésben egy általunk meghatározott `xy` koordinátára fog kiírni egy letárolt karaktert. Ennek lényege, hogy az eltolási értéket a program fogja kiszámolni a megadott értékek alapján:

[Program 3]

```
Pelda03      Segment ;Szegmensdefinió.
              assume cs:Pelda03,ds:Pelda03 ;Cs és ds regiszterek beállí-
                                              ;tása a szegmens elejére.

Start:       mov ax,Pelda03 ;A ds regiszter beállítása.
              mov ds,ax

              mov ax,0b800h ;A képernyő-memória szegmens-
              mov es,ax ;címét es regiszterbe tölti.

              mov al,byte ptr [KOORD_Y] ;Al regiszterbe a KOORD_Y címke
              ;alatti értéket tölti.

              mov ah,0 ;Ah regisztert nullázza.

              mov bl,160 ;Bl regiszterbe 160-at tölt mivel
              ;egy sor hossza 160 byte.

              mul bl ;Al regiszter értékét összeszorozza
              ;bl tartalmával, az eredményt ax-ben
              ;kapjuk.
```

Az Assembly programozás alapjai

mov di,ax		;Az így kiszámolt értéket di indexregiszterbe töltjük.
mov al,byte ptr [KOORD_X]		;A vízszintes koordináta értékét ;al regiszterbe tesszük..
mov ah,0		;Nullázza ah -t
mov bl,2		;A vízszintes koordináta értékét
mul bl		;megszorozzuk kettővel
add di,ax		;és ezt hozzáadjuk az ;indexregiszterhez.
mov al,byte ptr [BETU]		;Al-be a betű kódját,
mov ah,byte ptr [SZIN]		;ah-ba a színkódot töltjük.
mov es:[di],ax		;Az es:di által mutatott ;címre írja ax tartalmát azaz ;a fekete alapon fehér "A" ;betűt.
mov ax,4c00h		;Kilépés a DOS-ba.
int 21h		
KOORD_X: db 40		
KOORD_Y: db 12		
BETU: db "A"		
SZIN: db 7		
Pelda03	Ends	;A szegmens vége.
	End Start	;A program vége

A mintaprogramban bemutatásra került a memóriaváltozók használata is. Ezeket a program egy általunk adatok számára elkülönített részén kell elhelyezni. A memóriaváltozókat egy címkével azonosíthatjuk (**KOORD_X: stb.**), amik után egy kettőspont áll. Ezután meg kell jelölni az adat típusát (**db** - byte-os, **dw** - wordös, **dd** - doublewordös adat), majd el kell helyezni a tárolandó adatot. Ha egy betűt vagy szöveget idézőjelek közé teszünk, annak ascii kódja tárolódik, így például az "A" helyén 65 lesz letárolva. A programban szerepel még egy új utasítás is. A **mul** végrehajtása során ha utána 8 bites adat áll

(bl), az al regisztert szorozza meg a megadott regiszter tartalmával és az eredményt ax-ben kapjuk. Amennyiben 16 bites adattal szorzunk (bx), az AX regiszter tartalma szorzódik és az eredményt DX és AX regiszterekben kapjuk. A magasabbik helyiértékű részt a dx-ben, az alacsonyabbat az ax-ben. Tehát ha például ax regiszter tartalma 26e5h bx tartalma pedig 76ah akkor a mul bx végrehajtása során ax regiszterben 5dd2h dx-ben pedig 0120h értéket kapnánk. A programban a szorzást a karakter memóriacímének kiszámítására használtuk a 2. program előtt leírtak szerint. A regiszter nullázására és a 2-vel való szorzásra később majd egyszerűbb módszert is láthatunk. Az utasításnál egy dologra kell vigyázni, hogy a számmal való szorzás nem megvalósítható (mul 3), de ennek a kivételével bármilyen adatot használhatunk szorzónak. Található még egy új utasítás is a szövegben, az add összeadó művelet. Ennek lényege, hogy az első operandushoz adja a másodikat. A mov utasításnál leírtak itt is érvényesek illetve itt is legalább az egyik tagnak regiszternek kell lennie.

A következő példában egy szöveg pozícionált kiíratását mutatom be. Ez annyiban különbözik az előzőtől, hogy itt nem elég egy karaktert kiolvasni, hanem egy meghatározott szöveget végig ki kell írni. Ennek két megoldása lehetséges, az egyik, hogy megszámloljuk hány betűt akarunk kiíratni és egy ciklus segítségével írjuk a képernyőre a szöveget, de ennek a módszernek hátránya, hogy ha megváltoztatjuk a szöveget, a ciklus hosszát is változtatni kell. A másik módszer, hogy a szöveg végére elhelyezünk egy olyan kódot, amit a szövegben biztos hogy nem használunk pl.: 255 és a kiíratáskor figyeljük, a kirakandó karakter kódját. Ha nem 255, akkor kитеhető, ha az, akkor vége a kiíratásnak. Ügyelni kell, hogy egy betű kirakása után a következő karaktert kell olvasni, és a következő karakterpozícióba kell tenni azt.

[Program 4]

Pelda04	Segment	;Szegmensdefiníció.
	assume cs:Pelda04,ds:Pelda04	;Cs és ds regiszterek beállítása a szegmens elejére.

Az Assembly programozás alapjai

Start:	mov ax,Pelda04 mov ds,ax	;A ds regiszter beállítása.
	mov ax,0b800h mov es,ax	;A képernyő-memória szegmens- ;címét es regiszterbe tölti.
	mov al,byte ptr [KOORD_Y]	;Al regiszterbe tölti a ;KOORD_Y címke alatt tárolt ;értéket.
	mov bl,160	;Bl-be 160-at tölt, mivel ;egy sor 160 byte.
	mul bl	;Al értékét megszorozza bl ;értékével.
	mov di,ax	;Az így kapott eredményt ;di indexregiszterbe tölti.
	mov al,byte ptr [KOORD_X] mov bl,2 mul bl	;A vízszintes pozíció értékét ;al regiszterbe tölti ;és megszorozza 2-vel
	add di,ax	;majd hozzáadja di-hez.
	mov ah,byte ptr [SZIN]	;Ah-ba a színkódot tölti.
	mov si, offset SZOVEG	;Si-be a szöveg offsetcímét ;tölti.
.1_Pelda04:	mov al,[si]	;Al regiszterbe az si által ;mutatott címen lévő adatot ;tölti.
	cmp al,255	;Al értékét összehasonlítja ;255-el.
	jz Vege	;Ha egyezik, ugrik a Vége ;címkéhez.
	mov es:[di],ax	;Az es:di által mutatott ;címlére írja ax tartalmát

IBM PC Gyakorlati Assembly

	add	di,2	:A következő képernyő-pozíció
	inc	si	:A következő betű.
	jmp	.1_Pelda04	:Ugrik a .1_Pelda04 címkéhez.
Vege:	mov	ax,4c00h	:Kilépés a DOS-ba.
	int	21h	
KOORD_X:	db	30	
KOORD_Y:	db	12	
SZOVEG:	db	"A kiíratandó szöveg",255	
SZIN:	db	7	
Pelda04	Ends		:A szegmens vége.
	End	Start	:A program vége

Ez a program már egy teljes szöveg kiíratását el tudja végezni. Innen már csak egy lépés lenne különféle vezérlőkódokat beleiktatni a szövegbe és így lehetőség lenne például a képernyő-pozíció letárolása a szövegben illetve lehetne több mondatot kiírni különböző helyekre, mindössze annyit kell tenni, hogy elhelyezni a vezérlőkódot amit a programból figyelünk, és utána az új koordinátákat amiből a program kiszámolja a szöveg címét. De ezt a feladatot már az olvasóra bízom, az eddig használtakon kívül más utasítás nem szükséges a program megírásához. Ebben azonban szerepel öt új dolog is. A legegyszerűbb a címkek használata. Ezek hasonlítanak a memóriaváltozóknál használt címkekhez, csak ezeket nem adattárolásra használjuk, hanem általában ugráshoz, mint azt például a **jmp .1_Pelda04** sor is teszi. Fontos, hogy a címkek után kettőspontot kell tenni. Kivételt képeznek a szegmens nevek illetve a memóriaváltozók, ha az adatszegmens nem egyezik meg a kódszegmessel. A **jmp** egy feltétel nélküli vezérlésátadó utasítás. Ha a program ehhez a sorhoz ér, a végrehajtást az utasítás után megadott helyen folytatja. Ez lehet egy címke mint jelen esetben, de lehet egy regiszter is (**si**, **di**, **bx**) vagy a **jmp far** utasítás segítségével egy másik szegmensbe is átugorhatunk, mivel a normál **jmp**-vel csak az adott 64K-n belül ugrándoizhatunk.

A kiírás során sor kerül a karakterkód vizsgálatára is a `cmp` utasítással, ami két érték összehasonlítására szolgál. Használata során először az az adat áll, amit össze akarunk hasonlítani és utána amivel. A dolog működése tulajdonképpen egy kivonáson alapszik, amit a gép csak magában végez el, a regiszterek tartalmát nem változtatja meg. E művelet elvégzése után a `flag` regiszter egyes bitjei az eredménytől függően állnak be. A leggyakrabban használt két jelzőbit a `carry` és a `zero`. A `carry` az úgynevezett átviteli jelzőbit értéke 0 ha a második szám nem nagyobb az elsőnél, azaz ha az első számból kivonnánk a másodikat nem negatív számot kapnánk. Ha az értéke 1, akkor az eredmény negatív tehát a második szám az összehasonlítás során nagyobb volt. A másik jelzőbit a `zero`, értéke akkor 1, ha a kivonás eredménye nulla lenne, azaz a két szám azonos. Minden más esetben értéke 0. Ezeket a jelzőbiteket különböző módon tudjuk vizsgálni. Az egyik lehetőség a feltételes elágazás. A `jz` csak akkor ugrik a megadott helyre, ha a `z` bit értéke 1. Ha a bit 0 értékét szeretnénk figyelni, akkor a `jnz` utasítást kéne használni. Ezeknél a feltételes ugrásoknál figyelembe kell venni, hogy ezek úgynevezett relatív ugrások azaz előre és visszafelé is 127 byte-ot tudnak ugrani alapesetben. Ez a legtöbbször elegendő, azonban ha mégsem, akkor más megoldáshoz kell folyamodnunk.

A programban szerepel meg egy egyszerű de új dolog, az `inc`. Ez nem csinál mást, mint növeli eggyel az utána álló operandus értékét. Ez lehet regiszter illetve memóriatartalom is. Az `inc` utasítás párja a `dec` ami csökkenti eggyel az utána álló adat értékét.

Logikai műveletek:

Elég sokat beszéltünk már a bitekről de eddig sehol sem volt rájuk különösebben szükségünk. A következőkben ismertetésre kerülő utasítások működésének megértéséhez azonban elengedhetetlen a bitek fogalmának ismerete. Ugyanis a most következő logikai műveletek hatása legegyszerűbben az egyes bitek viselkedésének vizsgálatával érthető meg.

A legegyszerűbb ilyen művelet a bitek invertálása (ellenkezőjére való fordítása). Ezt a not utasítással végezhetjük el.

A not művelet működése:

Kiinduló érték: 01001010

A not művelet utáni érték: 10110101

A példán jól látható, hogy az egyes helyiértékek tartalma az ellenkezőjére váltott. Felhasználhatjuk ezt például 255-ből való kivonás helyett, mivel az eredmény ugyan az. Ezenkívül még sok más helyen alkalmazható.

Egy az előzőhöz nagyon hasonló a neg művelet. Az eltérés csupán annyi, hogy míg not un. egyes komplementst számol addig ez a szám kettes komplementset adja eredményül, ami az eredeti érték -1-szerese, ezt legegyszerűbben úgy számolhatjuk ki, ha invertáljuk a biteket és az eredményhez hozzáadunk egyet.

A neg művelet működése:

Kiinduló érték: 01001010

A neg művelet utáni érték: 10110110

Ezt a tömörítő eljárásnál fogjuk használni, de erről majd ott. A műveletnek egyébként negatív számok kezelésénél lenne szerepe, de ezzel nem foglalkozunk.

A következő logikai műveletekhez már két adatra lesz szükség, mivel az egyes bitek találkozásától függ az eredmény értéke. Ezek az and, or illetve xor utasítások.

Az and (és) művelet során az eredményben csak ott lesz az eredmény adott bitje 1 értékű, ahol a kiinduló értékben mindkét adatban 1 az adott bit értéke.

Az and művelet működése:

1.adat: 1100

2.adat: 1010

eredmény: 1000

Az **or** (vagy) műveletnél minden bit 1 értékű az eredményben, ahol a forrás adatok közül bármelyikben 1 az adott bit értéke.

Az **or** művelet működése:

<i>1.adat:</i>	1100
<i>2.adat:</i>	1010
<i>eredmény:</i>	1110

A **xor** (kizáró vagy) utasítás hasonlít az **or** működéséhez, annyi különbséggel, hogy itt a két darab egyes találkozására is nullát ad eredményül.

Az **xor** művelet működése:

<i>1.adat:</i>	1100
<i>2.adat:</i>	1010
<i>eredmény:</i>	0110

Ezek kombinálásával bármilyen logikai művelet előállítható. Például két számot **and** kapcsolatba hozunk egymással és az eredményt invertáljuk stb. A **xor** műveletnek szokták kihasználni azt a tulajdonságát, hogy két azonos bitre mindig nullát ad. Így ha egy számot saját magával **xor**-olok, annak eredménye biztos, hogy nulla lesz. Ezt a trükköt regiszterek nullázására szokták felhasználni, mivel a **mov ax,0** utasítás sor a memóriában 3, a **xor ax,ax** csak 2 Byte-ot foglal el. Ez hosszabb programoknál jelentős lehet. Illetve az **or**, **and** utasításokat lehet például arra felhasználni, hogy egy regiszter stb. tartalmáról megtudjunk pár dolgot, mivel az **or** illetve **and** művelet végrehajtásakor a **flag** egyes bitjei az eredménynek megfelelően állnak be. Ha például **ax** regiszter értéke nulla akkor az **or ax,ax** utasítás végrehajtása után a **z** **flag** értéke 1 lesz. Ez a sor szintén rövidebb a **cmp ax,0** megoldásnál.

A következő művelet, ami ebbe a témakörbe tartozik, az a forgatás. Lehetőség van ugyanis a byte, word tartalmának forgatására többféle módon. Ami közös mindegyik megoldásnál, hogy a byte vagy word széléről kicsúszó bit mindig beíródik a carry **flag**-be.

A szó szerinti forgatást a ror (rotate right) illetve rol (rotate left) utasítások végzik. A forgatáskor az utasítás után kell írni, amit forgatni akarunk (regiszter, memóriatartalom) és egy vessző után, hogy mennyit. Ez a 8086 alapú gépeknél (XT) vagy 1 vagy cl. Utóbbi esetben cl regiszterben megadott értékkel forgat. 80286-tól fölfelé megadható nagyobb szám is, de ekkor a programszövegben jelölni kell a fordítónak, hogy a programot nem XT-re írjuk. Ez úgy történik, hogy az első sorba egy .286 sort helyezünk el. Innen a fordító tudni fogja, hogy a program 80286 utasítást is tartalmaz.

A ror utasítás működése:

forгатás előtt: 01100101 *forгатás után: 10110010*
carry értéke: 1 mivel a byte jobb szélén kicsúszó bit értéke 1.

A rol utasítás működése:

forгатás előtt: 01100101 *forгатás után: 11001010*
carry értéke: 0 mivel a byte bal szélén kicsúszó bit értéke 0.

Tehát a byte forgatása során a kiforgó bit beíródik a carry flagbe és a byte másik szélén befordul mind a rol mind a ror utasításnál.

Hasonló módon működik az rcr illetve rcl utasítások, de itt a forgatott adatot megtoldja egy bittel a carry flag. Tehát mint a byte kilencedik bitje működik ugyanis a kiforduló bit a carry flagbe kerül és a carry előző értéke fordul be a byte másik oldalán.

Az rcr utasítás működése:

carry értéke forгатás előtt: 0
a byte forгатás előtt: 01100101 *forгатás után: 00110010*
carry értéke: 1 mivel a byte jobb szélén kicsúszó bit értéke 1.

carry értéke forгатás előtt: 1
a byte forгатás előtt: 01100101 *forгатás után: 10110010*
carry értéke: 1 mivel a byte jobb szélén kicsúszó bit értéke 1.

Az rcl utasítás működése:

carry értéke forgatás előtt: 0
a byte forgatás előtt: 01100101 *forgatás után: 11001010*
carry értéke: 0 mivel a byte bal szélén kicsúszo bit értéke 0.

carry értéke forgatás előtt: 1
a byte forgatás előtt: 01100101 *forgatás után: 11001011*
carry értéke: 0 mivel a byte bal szélén kicsúszo bit értéke 0.

A harmadik forgatási lehetőség, amikor az adat kicsúszo bitje szintén a c flagbe íródik, de a másik oldalról becsúszo bit értéke minden esetben 0.

Az shr utasítás működése:

forgatás előtt: 01100101 *forgatás után: 00110010*
carry értéke: 1 mivel a byte jobb szélén kicsúszo bit értéke 1.

Az shl utasítás működése:

forgatás előtt: 01100101 *forgatás után: 11001010*
carry értéke: 0 mivel a byte bal szélén kicsúszo bit értéke 0.

Egy byte illetve word forgatására még egy lehetőség van, amikor a jobbra forgatásnál a signum (előjel) flag értéke íródik az adat bal szélére. A jobb szélső bit forgatáskor szintén c-be íródik. A művelet fordítottja azonos az shl működésével.

A sar utasítás működése:

signum értéke forgatás előtt: 0
a byte forgatás előtt: 01100101 *forgatás után: 00110010*
carry értéke forgatás után: 1 mivel a byte jobb szélén kicsúszo bit értéke 1.

signum értéke forgatás előtt: 1
a byte forgatás előtt: 01100101 *forgatás után: 10110010*
carry értéke forgatás után: 1 mivel a byte jobb szélén kicsúszo bit értéke 1.

Az sal utasítás működése:

forgatás előtt: 01100101 *forgatás után: 11001010*
carry értéke: 0 mivel a byte bal szélén kicsúszó bit értéke 0.

Mindezeket a műveleteket kipróbálhatjuk a következő két mintaprogram segítségével, ha a megfelelő helyre az általunk kipróbálni kívánt utasítást írjuk. Ennek helye a szövegben külön jelölve van.

[Program 5]

Pelda05	Segment assume cs:Pelda05,ds:Pelda05	;Szegmensdefiníció ;Cs, ds beállítása
Start:	mov ax,Pelda05 mov ds,ax	;Ds regiszter beállítása ;a kód elejére
	mov ax,0b800h mov es,ax	;A képernyő-memória szegmens- ;címét es regiszterbe tölti.
	mov ax,3 int 10h	;80*25 karakteres mód be- ;állítása, képernyőtörlés.
	xor di,di	;Di nullázása.
	mov si,offset SZOVEG1	;Si mutatja a szöveg kezdő- ;címét.
	call Kiiró1	;Meghívja a Kiiró1 eljárást.
	mov bl,byte ptr [SZAM1] call Kiiró2	;BI-ben a kiírandó szám van ;és ezt a Kiiró2 rutin ;írja ki a képernyőre.
	mov di,160	;A következő szöveg ;kezdőcíme.
	mov si,offset SZOVEG2 call Kiiró1 mov bl,byte ptr [SZAM2] call Kiiró2	;Ugyan az mint előbb.

Az Assembly programozás alapjai

	mov di,320	
	mov si,offset SZOVEG3	
	call Kiir1	
	mov bl,byte ptr [SZAM1]	;Az első számot bl
	and bl,byte ptr [SZAM2]	;regiszterbe teszi és
	call Kiir2	;vegrehajtja a kijelölt
		;műveletet a második számmal
		;amit utána kiir a
		;képernyőre. Itt kell a
		;kívánt műveletet beállítani.
	xor ax,ax	;Billentyűvárás.
	int 16h	
	mov ax,4c00h	;Kilépés a DOS-ba.
	int 21h	
Kiir1	Proc	;Kiir1 rutin kezdete.
	mov cx,16	;A szöveg 16 karakterből áll.
	mov ah,15	;Fekete alapon fehér szín.
.1_Kiir1:	mov al,[si]	;A kiíratandó betűt al
		;regiszterbe tölti, majd
	mov es:[di],ax	;kiírja es:[di] által
		;mutatott címre.
	add di,2	;A következő karakterpozíció.
	inc si	;A következő karakter.
	loop .1_Kiir1	;Csökkenti cx értéket és ugrik
		;a megadott helyre ha cx nem 0
	ret	;Visszatérés a hívó
		;programrészhez.
Kiir1	Endp	;A rutin vége.

IBM PC Gyakorlati Assembly

Kiiró2	Proc	;Kiiró2 rutin kezdete.
	add di,6	;Három karakterpozícióval ;arrébb lép.
	mov cx,8	;Az adat 8 bitből áll.
	mov ah,15	;Fekete alapon fehér szín.
.1_Kiiró2:	mov al,"0"	;Al regiszterbe a "0" ascii ;kódját tölti.
	shl bl,1	;Az adatot egyel balra ;lépteti, így a kicsorduló ;bit a carry flagbe kerül.
	jnc .2_Kiiró2	;Ha ez a bit 0, akkor ugrás ;a .2_Kiiró címkehez.
	mov al,"1"	;Ha 1, akkor az al-be az ;"1" ascii kódját töltjük.
.2_Kiiró2:	mov es:[di],ax	;A számjegyet a képernyőre ;írjuk.
	add di,2	;Egy hellyel arrébb.
	loop .1_Kiiró2	;Ismétlés cx-nek megfelelően.
	ret	;Visszatérés a rutinból.
Kiiró2	Endp	;A rutin vége.
SZOVEG1:	db "Az első byte :"	
SZOVEG2:	db "A második byte :"	
SZOVEG3:	db "Az eredmény :"	
SZAM1:	db 01011101b	
SZAM2:	db 10101011b	
Pelda05	Ends	;A szegmens vége.
	End Start	;A program vége.

Ez a program már sokkal összetettebb mint az előző négy. Ebben már megtalálhatók a ciklusok, eljárások, feltételek stb. Mint az látható is a regiszterek nullázására itt már a `xor` művelet lett használva.

A könyv elején említettem, hogy ha egy feladatra többször van szükségünk, akkor azt elég egyszer megírni, majd a programból egy `call` utasítással végrehajtatni. Ez hasonlít a `jmp` utasításra, de itt a gép megjegyzi a `call` utasítás címét a későbbi visszatéréshez. Erre mutat két példát is az 5. program. Az eljárás (procedure) kezdetét egy `Proc` szó jelzi. Természetesen ahogy a szegmenseknek, így az eljárásoknak is kell egy nevet adni, amivel később hivatkozhatunk rá. Ez a név a `Proc` előtti címke. A rutint a címkenév és az `Endp` zárja. Nagyon fontos sor a rutinunkban a `ret`. Ugyanis ez az utasítás jelenti a gépnek, hogy térjen vissza a `call` utáni sorra, ahonnan elindították a rutint. Nagyon fontos dolog, hogy ne próbáljunk meg eljárásból kilépni a DOS-ba, mert ez nagy valószínűséggel egy lefagyást fog eredményezni. Ennek oka, hogy a számítógép kezel egy úgynevezett `stacket`, ahová adatokat lehet elmenteni illetve onnan visszaolvasni. Ez a `stack` egy a memóriában visszafelé növekvő terület, ha nem adunk meg az `ss` regiszternek külön értéket, akkor a `stack` eleje a szegmensünk legvége lesz. Ha adatot mentünk ide, akkor azt beírja és csökkenti a `stack` mutató értékét, ami mindig az utoljára beírt adatra mutat. Ugyanígy kiolvasáskor is a legutoljára beírt számot kapjuk meg először és utána az előzőt stb. Nos visszatérve a lefagyás okára, a program indításakor a visszatérési cím beíródik a `stack`-be amit kilépéskor kiolvasva tudja, hova kell visszatérni. A `call` utasítás is a `stacket` használja a visszatérési cím tárolására amit a `ret`-hez érve olvas ki és ugrik a tárolt címre. Ha a rutinból próbál-nánk meg kilépni, nem a DOS-hoz való visszatérés címét olvasná ki a gép, hanem a `call` címét. Természetesen van megoldás, de ez egy kicsit bonyolultabb, ugyanis megtehetjük, hogy kiolvassuk a `stack`-ből a `call` visszatérési címét és ekkor a legutolsó tárolt adat a DOS-hoz való visszatérési címet fogja mutatni.

A másik fontos dolog ami megtalálható a programban az a ciklus. A ciklusok működése azon az elven alapszik, hogy egy regiszterbe beírjuk a végrehajtások számát, és a programrészlet végén csökkentjük a regiszter értékét és ha még nem nulla, akkor

megismételjük a programot mindaddig míg a regiszter értéke nulla nem lesz. A PC-n ezt a feladatot egyszerűen megoldhatjuk, mivel külön utasítás van erre a célra a **loop**. A ciklus lefutásának számát **cx** regiszterben kell megadni és amikor a program a **loop** utasításhoz ér, csökkenti **cx** értékét, és ha az még nem nulla, akkor ugrik a megadott címre, ami hasonlóképpen a **jmp**-hez lehet címke illetve regiszter.

A programban a kilépésen kívül két ROM BIOS funkció is használva lett. Az 10h megszakítás a képernyőt kezeli. Ha **ah**-ba 0 van, akkor a képernyő üzemmódját állítja be **al** értékének megfelelően. Jelen esetben a 80*25 karakteres módot. A 16h rutin a billentyűzet kezelést végzi. Ha **ah**-ban nulla van, akkor a gép vár egy billentyű lenyomására, és annak **ascii** kódját **al** illetve **scan** kódját **ah** regiszterben adja vissza. Itt a visszaérkező adatot nem használjuk fel, mivel a dolog szerepe csak egy billentyűvárás, hogy ne azonnal térjen vissza a DOS-hoz.

A program működését illetően a bináris számkiírás ami új. Ezt úgy oldja meg, hogy az adatot tartalmazó byte-ot eggyel balra forgatja, így abból a bal szélső bit értéke a **carry flag**-be kerül. A művelet végrehajtása előtt **al** regiszterbe a nullás számjegy kódját töltöttük be. Ha a forgatás során a **c** értéke 1 lenne, akkor **al** tartalmát az egyes számjegy kódjára változtatjuk. Ha nulla, akkor átugorjuk a változtatást. Az így kialakult számjegyet a már megszokott módon a képernyőre írjuk. Mindezt megismételjük az összes bitre (azaz 8-szor).

A következő program semmi újdonságot nem fog tartalmazni, mindössze nem két forrásadat lesz, csak egy mivel a forgatáshoz csak egy adat szükséges.

[Program 6]

Pelda06	Segment assume cs:Pelda06,ds:Pelda06	;Szegmensdefiníció ;Cs, ds beállítása
Start:	mov ax,Pelda06 mov ds,ax	;Ds regiszter beállítása ;a kód elejére
	mov ax,0b800h mov es,ax	;A képernyő-memória szegmens- ;címét es regiszterbe tölti.

Az Assembly programozás alapjai

	mov ax,3	:80*25 karakteres mód be-
	int 10h	:állítása, képernyőtörles.
	xor di,di	:Di nullázása.
	mov si,offset SZOVEG1	:Si mutatja a szöveg kezdő-
	call Kiiró1	:Meghívja a Kiiró1 eljárást.
	mov bl,byte ptr [SZAM1]	:BI-ben a kiírandó szám van
	call Kiiró2	:és ezt a Kiiró2 rutin
		:írja ki a képernyőre.
	mov di,160	:A következő szöveg
		:kezdőcíme.
	mov si,offset SZOVEG2	:Ugyan az mint előbb.
	call Kiiró1	
	mov cl,1	:A forgatás értékét egyre
		:állítja.
	mov bl,byte ptr [SZAM1]	:A számot bl regiszterbe tölti
	ror bl,cl	:és végrehajtja a kijelölt
	call Kiiró2	:műveletet, amit utána kiir a
		:képernyőre. Itt kell a
		:kivánt műveletet beállítani.
	xor ax,ax	:Billentyűvárás.
	int 16h	
	mov ax,4c00h	:Kilépés a DOS-ba.
	int 21h	
Kiiró1	Proc	:Kiiró1 rutin kezdete.
	mov cx,21	:A szöveg 21 karakterből áll.
	mov ah,15	:Fekete alapon fehér szín.
.1_Kiiró1:	mov al,[si]	:A kiíratandó betűt al
		:regiszterbe tölti, majd
	mov es:[di],ax	:kiírja es:[di] által
		:mutatott címre.

IBM PC Gyakorlati Assembly

	add	di,2	;A következő karakterpozíció.
	inc	si	;A következő karakter
	loop	.1_Kiiró1	;Csökkenti cx értékét és ugrik ;a megadott helyre ha cx nem 0
	ret		;Visszatérés a hívó ;programrészhez.
Kiiró1	Endp		;A rutin vége.
Kiiró2	Proc		;Kiiró2 rutin kezdete.
	add	di,6	;Három karakterpozícióval ;arrébb lép.
	mov	cx,8	;Az adat 8 bitből áll.
	mov	ah,15	;Fekete alapon fehér szín.
.1_Kiiró2:	mov	al,"0"	;Al regiszterbe a 0 ascii ;kódját tölti.
	shl	bl,1	;Az adatot egyel balra ;lepteti, így a kicsorduló ;bit a carry flagbe kerül.
	jnc	.2_Kiiró2	;Ha ez a bit 0, akkor ugrás ;a .2_Kiiró címkehez.
	mov	al,"1"	;Ha 1, akkor az al-be az ;1 ascii kódját töltjük.
.2_Kiiró2:	mov	es:[di],ax	;A számjegyet a képernyőre ;írjuk.
	add	di,2	;Egy hellyel arrébb.
	loop	.1_Kiiró2	;Ismétlés cx-nek megfelelően.
	ret		;Visszatérés a rutinból.
Kiiró2	Endp		;A rutin vége.

Az Assembly programozás alapjai

```
SZOVEG1: db "Az eredeti szám :"  
SZOVEG2: db "A művelet eredménye :"  
SZAM1: db 01011101b
```

```
Pelda06    Ends                ;A szegmens vége.  
           End Start          ;A program vége.
```

Az eddigi példákban megismerhettük a szövegiíratás módjait illetve a bináris számok kiíratásának egy módjával. A következőkben ismertetésre kerül a számok hexadecimális, decimális kiíratása.

A hexa számok kiíratásánál problémát jelent, hogy a számjegyeken kívül A-F-ig betűket is tartalmazhat a szám. És a kilences szám és az A betű között vannak további karakterek, melyek kiíratása nem célravezető. A megoldás, hogy egy táblázatba helyezzük a lehetséges karaktereket és a számnak megfelelő jelet olvassuk ki innen és írjuk ki a képernyőre. A PC-n az ilyen táblázatkezelésre van egy speciális utasítás, ami a beállított táblázat al-edik elemét tölti al-be, ez az xlat. A táblázat offsetcímét (kezdőcímét a szegmens elejéhez képest) bx regiszterbe kell tenni a művelet végrehajtása előtt.

[Program 7]

```
Pelda07    Segment                ;Szegmensdefiniáció  
           assume cs:Pelda07,ds:Pelda07 ;Cs, ds beállítása  
  
Start:     mov ax,Pelda07          ;Ds regiszter beállítása  
           mov ds,ax              ;a kód elejére  
  
           mov ax,0b800h          ;A képernyő-memória szegmens-  
           mov es,ax              ;címét es regiszterbe tölti.  
  
           mov ax,3                ;80*25 karakteres mód be-  
           int 10h                 ;állítás, képernyőtörles.  
  
           xor di,di              ;Di nullazása.  
  
           mov bx,offset HEXTABLE ;Bx regiszterbe a konvertáló  
           ;tábla eltölési értékét írja.
```

IBM PC Gyakorlati Assembly

```

mov dx,word ptr [HEXSZAM] ;Dx regiszterbe tölti a
                           ;kiírandó számot.

mov ah,15                  ;A számok színe fekete alapon
                           ;fényes fehér.

mov cx,4                   ;A szám négy számjegyből áll.

.1_Pelda07: push cx        ;Cx értékét a verembe menti.

mov cx,4                   ;Egy számjegyet négy bit
                           ;határoz meg.

xor al,al                  ;Törli az al regisztert.

.2_Pelda07: shl dx,1       ;A dx regiszter felső négy
rcl al,1                   ;bitjét al regiszterbe
loop .2_Pelda07           ;forgatjuk.

xlat                       ;A számjegynek megfelelő
                           ;karakterkódot tölti al
                           ;regiszterbe

mov es:[di],ax            ;és ezt kiírja a képernyőre.

add di,2                   ;A következő írási pozíció.

pop cx                     ;Cx előző értékét kiolvassuk
                           ;veremből.

loop .1_Pelda07           ;A következő számjegy
                           ;kiírása.

xor ax,ax                  ;Billentyűvárás.
int 16h

mov ax,4c00h              ;Kilépés a DOS-ba
int 21h

HEXSZAM: dw 5b2eh         ;A kiírandó szám.

HEXTABLE: db "0123456789ABCDEF" ;Konvertáló tábla.

Pelda07  Ends             ;A szegmens vége.
        End Start        ;A program vége.

```


Mint az látható, a program lelke a HEXTABLE címke alatt letárolt pár betű, ugyanis ha például a kiírandó számjegy decimális értéke 13 akkor a táblázat 13. elemét fogja kiírni a képernyőre, azaz egy D betűt. Ezt a műveletet hajtja végre az xlat utasítás. A programban egy 16 bites számot írunk ki. Egy hexadecimális számjegy 0-15-ig vehet fel értéket, amit 4 biten lehet ábrázolni. Tehát 4 egyenként 4 bites számjegyet kell kiírni a képernyőre. Ez két egymásba ágyazott ciklussal lett megoldva. Viszont ha egy regiszternek egymás után kétszer adunk értéket, (mivel a loop utasításhoz csak a cx regisztert lehet használni) akkor az a második értéket veszi fel és az elsőt elfelejti. Ellenben nekünk az első ciklusunk számolja a 4 karaktert és a második a 4 bitet. Tehát mindkettőre szükség van. A megoldás, hogy mielőtt a második ciklusnak értéket adnánk, ideiglenesen eltároljuk a cx regisztert a veremben, majd a második ciklus lefutása után kiolvassuk innen és az értékét csökkentve visszaugrunk az elmentésre mindaddig míg nem raktuk ki a teljes számot.

A stack (verem) működéséről már esett pár szó, de itt azért még egy kis részletezés, hogy mi is történik amikor a push utasítással elmentünk egy adatot vagy a pop segítségével kiolvassuk egyet. A stack, mint azt említettem az ss szegmensregiszter által meghatározott szegmens végétől kezdődően lefelé növekszik. A legutoljára beírt érték helyét az sp (stack pointer) regiszter mutatja. Ennek külső befolyásolása nem célszerű. csak feltétlenül szükséges esetben tegyük, mivel a gép adatmentéskor illetve kiolvasáskor automatikusan átírja az értékét. A továbbiakban egyébként ahol lehet kerüljük a veremműveletek használatát, mivel végrehajtása a többi utasításhoz képest meglehetősen lassú. Természetesen ez nem azt jelenti, hogy tilos használni, de ott ahol fontos a program gyorsasága, ott kerülendő.

A most következő mintaprogram a számok decimális formában való kiírását mutatja be. melynek legegyszerűbb megoldása a tízzel való osztogatás mindaddig, míg a szám nem kisebb tíznél. Az egyes számjegyeket nem az osztás eredménye, hanem a maradék adja, mivel így bármekkora számot kiírhatunk, míg ha a másik lehetséges módszert alkalmazzuk, hogy elosztjuk a számot pl 1000-el, az eredményt kiírjuk, utána 100-al és így tovább, akkor a jelen esetben a legnagyobb kiírható szám a 9999.

A programban alkalmazott módszer hátránya, hogy a számjegyeket fordított sorrendben kapjuk meg, de a probléma könnyen megoldható, ha nem egyből a képernyőre írjuk az adatot, hanem letároljuk a memóriában és közben számoljuk a számjegyek számát. Ha végeztünk, fordított sorrendben kiírhatjuk a teljes számot.

A számjegyek számkarakterre való átalakítása itt sokkal egyszerűbb, mivel ha a 0 számjegy ascii kódja 48, ezért csak hozzáadunk a számhoz 48-at és már írhatjuk is a képernyőre.

[Program 8]

```

Pelda08      Segment           ;Szegmensdefinició
                assume cs:Pelda08,ds:Pelda08 ;Cs, ds beállítása

Start:      mov  ax,Pelda08      ;Ds regiszter beállítása
                mov  ds,ax         ;a kód elejére

                mov  ax,0b800h    ;A képernyő-memória szegmens-
                mov  es,ax         ;címét es regiszterbe tölti.

                mov  ax,3          ;80*25 karakteres mód be-
                int  10h          ;állítás, képernyőtörés.

                mov  di,offset SZAMHELY ;Erre a címre fogja letárolni
                ;a számot kiírás előtt.

                mov  ax,word ptr [DECSZAM] ;A kirakandó szám.

                mov  bx,10         ;Az osztás mértéke.

                xor  cx,cx         ;A számláló nullázása.

.1_Pelda08: xor  dx,dx          ;A div utasítás a jelen
                div  bx           ;esetben dx:ax regiszter
                ;tartalmat osztja, de
                ;számunkra hasznos adat csak
                ;az ax regiszterben van,
                ;ezért a dx regisztert
                ;törölni kell.

                mov  [di],dl      ;Az osztás maradékának alsó
    
```

Az Assembly programozás alapjai

		;byte-ját a memóriába ;mentjük.
inc	cx	;A számláló növelése.
inc	di	;A következő címre írja a ;következő számot.
or	ax,ax	;Ax vizsgálata,
jnz	.1_Pelda08	;ha nem 0, ugrás vissza.
mov	si,di	;Si regiszterbe di-1-et
dec	si	;töltünk, mivel ez az utolsó ;értékes szám.
xor	di,di	;Di nullázása.
mov	ah,15	;Színbeállítás.
.2_Pelda08: mov	al,[si]	;Al-be tölti az utoljára ;letárolt számjegyet ami ;valójában az első.
cdd	al,48	;Ascii számjeggyé alakítja
mov	es:[di],ax	;és kiírja a képernyőre.
add	di,2	;Következő pozíció.
dec	si	;Előző számjegy.
loop	.2_Pelda08	;Ismétlés a számjegyek ;számának megfelelően.
xor	ax,ax	;Billentyűvárás.
int	16h	
mov	ax,4c00h	;Kilépés a DOS-ba.
int	21h	
DECSZAM: dw	34576	;Az ábrázolandó szám.
SZAMHELY: db	?	;A szám átmeneti tárolására ;szolgáló hely
Pelda08	Ends	;Szegmens vége
	End Start	;Program vége

A programban a `div` utasítás segítségével osztjuk el `ax` értékét tízzel, amit a `bx`-ben tároltunk. Azért kell 16 bites osztási műveletet használni, mert ha `bl`-t használnánk osztónak, az eredményt `al`, a maradékot `ah` regiszterben kapnánk és ha egy 2559-nél nagyobb számot osztanánk tízzel, a maradék nem férne el `ah` regiszterben. De így, hogy a `dx:ax` tartalmát osztjuk `bx`-el és `dx` értékét az osztás előtt nullára állítjuk, a maradékot `dx`-ben kapjuk, ami biztos, hogy el fog ott férni (65535-ig).

Ezzel tulajdonképpen a számok kiíratási lehetőségeit többé kevésbé letárgyaltuk, az összes többi már ezek kombinálása. Van azonban a számok tárolásának még egy nagyon gyakori fajtája az úgynevezett BCD számábrázolás. Ez nem más mint a decimális és a hexadecimális ábrázolás keveréke. Itt a hexa számban a legmagasabb használható érték a 9. Ilyen módon legfőképpen a ROM BIOS programjai kezelik a számokat. A most bemutatásra kerülő példa is erre támaszkodik, a pontos időt fogjuk a képernyőre írni. Itt például a 16 óra 42 perc 1642h számként tárolódik. Használatuk igen egyszerű, hasonlít a decimális számok kiíratására, mivel itt is hozzá kell adni a számhoz 48-at a kiírás előtt, de itt is egyszerre csak egy számjegyet teszünk ki.

A ROM program elindítása után az eredményt az egyes regiszterekben kapjuk mégpedig `ch`-ban az órát, `cl`-ben a percet és `dh`-ban pedig a másodpercet BCD formátumban.

[Program 9]

Pelda09	Segment <code>assume cs:Pelda09,ds:Pelda09</code>	;Szegmensdefiníció ;Cs, ds beállítás
Start:	<code>mov ax,Pelda09</code> <code>mov ds,ax</code> <code>mov ax,0b800h</code> <code>mov es,ax</code> <code>mov ax,3</code> <code>int 10h</code>	;Ds regiszter beállítása ;a kód elejére ;A képernyő-memória szegmens- ;címét es regiszterbe tölti. ;80*25 karakteres mód be- ;állítás, képernyőtörlés.
.1_Pelda09:	<code>mov ax,200h</code> <code>int 1ah</code>	;Az idő lekérdezése

Az Assembly programozás alapjai

	call	Kiiro	;és kiírása.
	mov	ax,100h	;Figyeli, hogy van-e
	int	16h	;lenyomott billentyű.
	jz	.1_Pelda09	;Ha nincs, ugrik az elejére.
	xor	ax,ax	;Ha van, akkor kiolvassa azt
	int	16h	
	mov	ax,4c00h	;és visszatér a DOS-hoz.
	int	21h	
Kiiro	Proc		;A rutin kezdete.
	mov	di,1670	;A képernyő közepe tája.
	mov	ah,15	;Szinbeállítás
	mov	bx,cx	;Az óra,perc értékét áttölti ;bx-be.
	call	Kiiro2	;kiírja az óra értékét.
	add	di,2	;Helyet hagy a kettőspontnak.
	call	Kiiro2	;Kiiirja a percet.
	add	di,2	;A kettőspont helye.
	mov	bh,dh	;A másodperc értékét bh-ba
	call	Kiiro2	;tölti és kiíratja.
	mov	al,";"	;Al regiszterbe a ";" jel
	mov	es:[di-6],ax	;kódját teszi és kiírja
	mov	es:[di-12],ax	;azt a két megadott helyre.
	ret		;Visszatérés a rutinból.
Kiiro	Endp		;A kiíró rutin vége.

IBM PC Gyakorlati Assembly

Kiir2	Proc	;A kiir2 rutin kezdete.
	mov cx,2	;Egyszerre 2 számjegyet
.1_Kiir2:	push cx	;irunk ki.
	mov cx,4	;Ami egyenként 4 bitből áll.
	xor al,al	;Al nullázása.
.2_Kiir2:	shl bx,1	;Bx felső 4 bitjét átforgatja
	rcl al,1	;al regiszterbe.
	loop .2_Kiir2	
	add al,48	;Átalakítja karakterkóddá
	mov es:[di],ax	;és kiírja a képernyőre.
	add di,2	;A következő képhely.
	pop cx	;A következő számjegy.
	loop .1_Kiir2	
	ret	;Visszatérés a rutinból.
Kiir2	Endp	;Az eljárás vége.
Pelda09	Ends	;A szegmens vége.
	End Start	;A program vége.

A program új utasításokat nem tartalmaz, ami mégis új az a REAL TIME óra kezelése. Ez nem más, mint egy a számítógép által kezelt óra amit software úton lekérdezhetünk, beállíthatunk stb. Ennek a kezelésére most az 1Ah BIOS program 02 funkciója lett alkalmazva. A hívás után a már említett regiszterekben kapjuk az idő értékét. A feladat csak annyi, hogy ezt kiírjuk a képernyőre. Persze ahhoz, hogy valami formája is legyen a dolognak, ne árt elválasztó kettőspontokat rakni az óra perc illetve a perc másodperc értékek közé. Ezenkívül meg kell oldani, hogy az óra járjon, de mégis ki lehessen lépni belőle. Ha az eddigi billentyűzet figyelést alkalmaztuk volna, az óra nem járna, csak kiírná az aktuális értéket és egy gombnyomásra kilépne. Ha egy jmp utasítással visszaugránánk a kiíratáshoz, akkor az óra járna, de billentyűfigyelés nélkül nem lehetne kiszállni a program futásából. A megoldás egy másik fajta figyelés kombinálása a

visszaugrással. A 16h rutinnak a 01 funkciója nem vár egy billentyű lenyomására, csak az aktuális állapotról ad információt az ax regiszteren illetve a flagen keresztül. A z jelzőbit 1 értéke jelenti, hogy nincs lenyomott gomb tehát visszaugorhatunk az óralekérdezésre. Amennyiben van, azt ki kell olvasni a billentyűzet pufferből az eddig is használt rutinnal de itt a különbség annyi, hogy nem vár mivel már van lenyomott billentyű, ha ezt a kiolvasást kihagynánk akkor kilépés után kiírná azt a betűt amit lenyomtunk. Ezután a megszokott módon visszatérünk a DOS-hoz. A program egyszerűsítése érdekében a kiírás egy eljárással lett megoldva amit egy másik eljárás kezel. A főprogram csak az időinformációk lehívását illetve a billentyűzetkezelést végzi. A megjelenítést egy rutin vezérli, hogy minden a megfelelő helyre kerüljön.

A programok másik fontos eleme a megjelenítések mellett a vezérlés. Az, hogy a felhasználó hogyan tudja irányítani a program menetét, választani adott lehetőségek közül. Az egyik legkulturáltabb megoldás a menüvezérlés, amikor a képernyőn felsorolt lehetőségek közül úgy választhatunk, hogy például egy más háttérszínű sávval mozoghatunk az egyes lehetőségek között, és az enter vagy más billentyű lenyomásával lehet kiválasztani a megfelelő funkciót. Egy másik módszer, amikor a választás egy betű lenyomásával történik, ilyen például az i/n választási lehetőség. Természetesen az kevés, hogy kiírjuk a képernyőre a menüpontokat, és közöttük mozgunk, bár ez is feladat, de a választás után el kell tudnunk dönteni, hogy melyik lehetőséget választottuk és azt kell végrehajtani. Erre láthatunk egy példát a következő programban, ami egy kicsit már hosszabb az eddigieknél és a bonyolultsága is nagyobb egy kicsit.

[Program 10]

Pelda10	Segment	
	<code>assume cs:Pelda10,ds:Pelda10</code>	;Szegmensdefiníció. ;Cs, ds beállítása.
Start:	<code>mov ax,Pelda10</code>	;Ds beállítása a kód elejére.
	<code>mov ds,ax</code>	
	<code>mov ax,0b800h</code>	;Videomemória szegmenscímét
	<code>mov es,ax</code>	;es regiszterbe tölti.

IBM PC Gyakorlati Assembly

```

mov ax,3 ;Képernyőtörlés.
int 10h

call Kiiro ;A menüpontok kiíratása.

.1_Pelda10: call Cszik ;A mutató kirakása.

.2_Pelda10: mov ax,100h ;Billentyűzet figyelés.
int 16h
jz .2_Pelda10
xor ax,ax
int 16h

cmp ah,1ch ;Enter figyelése
jz Enter ;Ugrás ha az.

cmp ah,48h ;A felfelé nyíl figyelése.
jnz .3_Pelda10 ;Ha nem az akkor a következő
;vizsgálatra ugrik.

cmp byte ptr [MUTATO],1 ;Ha a mutató értéke még nem 1
jz .2_Pelda10 ;akkor csökkenti a MUTATO
dec byte ptr [MUTATO] ;pozícióját egyel.

jmp .1_Pelda10 ;újabb billentyű figyelése.

.3_Pelda10: cmp ah,50h ;A lefelé nyíl figyelése,
jnz .2_Pelda10 ;Ha nem az, vissza a
;billentyűzet figyelésre.

cmp byte ptr [MUTATO],5 ;Ha a MUTATO még nem az
jz .2_Pelda10 ;ötödik helyen áll, növeli
inc byte ptr [MUTATO] ;az értékét.
jmp .1_Pelda10 ;Vissza a figyelés elejére.

Enter: mov cl,byte ptr [MUTATO] ;Cl ciklusváltozóba a
;MUTATO értékét tölti.

.1_Enter: mov si,offset CIMEK ;A CIMEK címkénél tárolt
mov bx,[si] ;offsetcímek közül azt
add si,2 ;tölti a bx regiszterbe,
loop .1_Enter ;amelyiken a mutató állt.

jmp bx ;Ugrik a bx által mutatott
;programrésze.

```


Az Assembly programozás alapjai

```

.2_Enter:  mov  si,offset KERDES           ;A KERDES alatti szöveg
             call Kiiro2                 ;kiíratása.

.3_Enter:  mov  ax,100h                 ;Az igen/nem választási
             int  16h                     ;lehetőségek vizsgálata.
             jz   .3_Enter
             xor  ax,ax
             int  16h
             cmp  ah,31h                 ;Ha nem, ugrás a kilépésre.
             jz   .4_Enter
             cmp  ah,17h                 ;Ha nem az igen, akkor
                                         ;újabb billentyű beolvasása,
                                         ;mert a lenyomott gomb
                                         ;érvénytelen.
             jnz  .3_Enter

             jmp  Start                 ;Ha az i lett lenyomva,
                                         ;ugrik a program elejére.

.4_Enter:  mov  ax,4c00h               ;Kilépés a DOS-hoz.
             int  21h

Menu1:    mov  si,offset SZOVEG1      ;Az egyes választások
             call Kiiro2                 ;eseten végrehajtandó
             jmp  .2_Enter               ;programok.

Menu2:    mov  si,offset SZOVEG2
             call Kiiro2
             jmp  .2_Enter

Menu3:    mov  si,offset SZOVEG3
             call Kiiro2
             jmp  .2_Enter

Menu4:    mov  si,offset SZOVEG4
             call Kiiro2
             jmp  .2_Enter

Menu5:    mov  si,offset SZOVEG5
             call Kiiro2
             jmp  .2_Enter

Csik      Proc

             mov  al,byte ptr [MUTATO2] ;A mutató régi értékét
             mov  bl,160                 ;al, egy sor hosszát a bl
                                         ;regiszterbe tölti,

```

```

xor  ah,ah
mul  bl                ;és kiszámolja a mutató
add  ax,1499          ;kezdőcímét.
mov  di,ax
mov  cx,21
mov  al,7              ;Fekete alapon fehér szín.
.1_Csik: mov es:[di],al ;A színinformációk beírása
add  di,2              ;a képernyő-memóriába,
loop .1_Csik          ;ezáltal törli az előző
                    ;csíkot.

mov  al,byte ptr [MUTATO] ;Ugyan az, mint az előbb,
mov  byte ptr [MUTATO2],al ;de most már az új pozícióval
mov  bl,160            ;és a kiemelt háttér szinnel
xor  ah,ah            ;rajzolja a csíkot.
mul  bl
add  ax,1499
mov  di,ax
mov  cx,21
mov  al,47
.2_Csik: mov es:[di],al
add  di,2
loop .2_Csik

ret

Csik    Endp

Kiiro   Proc

mov  si,offset MENUK ;A már ismert szöveg
mov  ah,7              ;kiírató rutin azzal a
mov  di,1660           ;kiegészítéssel, hogy a
                    ;0 kódra az eltárolt di
.1_Kiiro: push di      ;értékét kiolvassa, hozzáad
                    ;160-at (igy sort emel),
.2_Kiiro: mov al,[si]  ;majd ismét elmenti.
inc  si                ;A 255 kód jelenti a szöveg
cmp  al,0              ;végét és a rutinból való
jz   .3_Kiiro          ;visszatérést.
cmp  al,255
jz   .4_Kiiro
mov  es:[di],ax
add  di,2
jmp  .2_Kiiro

```

Az Assembly programozás alapjai

```
.3_Kiiro:  pop  di
          add  di,160
          jmp  .1_Kiiro
```

```
.4_Kiiro:  pop  di
          ret
```

```
Kiiro     Endp
```

```
Kiiro2    Proc
```

```
    mov  al,[si]           ;Ez a kiíró csak az első
    xor  ah,ah             ;felében különbözik az
    shl  ax,1              ;előzőtől, ugyanis itt a
    mov  di,ax             ;kiírandó szöveg első két
    mov  al,[si+1]         ;byte-ja a koordináta,
    mov  bl,160            ;amiből a program kiszámolja
    mul  bl                 ;az aktuális memóriacímet.
    add  di,ax             ;Az eljárás a továbbiakban
    add  si,2              ;azonos az előzővel, de itt
    mov  ah,12            ;nincs soremelés.
```

```
.1_Kiiro2: mov  al,[si]
          inc  si
          cmp  al,0
          jz   .2_Kiiro2
          mov  es:[di],ax
          add  di,2
          jmp  .1_Kiiro2
```

```
.2_Kiiro2: ret
```

```
Kiiro2    Endp
```

```
MUTATO:   db   1
```

```
MUTATO2:  db   1
```

```
MENUK:    db   "Az első menüpont",0
          db   "A második menüpont",0
          db   "A harmadik menüpont",0
          db   "A negyedik menüpont",0
          db   "Az ötödik menüpont",255
```

IBM PC Gyakorlati Assembly

```
SZOVEG1: db 22,5,"Az első menüpont lett kiválasztva.",0
SZOVEG2: db 21,5,"A második menüpont lett kiválasztva.",0
SZOVEG3: db 21,5,"A harmadik menüpont lett kiválasztva.",0
SZOVEG4: db 21,5,"A negyedik menüpont lett kiválasztva.",0
SZOVEG5: db 22,5,"A ötödik menüpont lett kiválasztva.",0
```

```
KERDES: db 28,20,"Akar újra választani (i/n)",0
```

```
CIMEK: dw offset Menu1
dw offset Menu2
dw offset Menu3
dw offset Menu4
dw offset Menu5
```

```
Pelda10 Ends
End Start
```

Nos a program méretétől nem szabad megijedni, mert részegységeiben vizsgálva nagyon sok minden ismerősnek tűnhet. Egyébként ez valójában egy parányi program, a gyakorlatban ilyen rövid feladattal ritkán találkozunk.

Egy program működésének elemzésekor az egyik megközelítés az, amikor a programot részegységeiben vizsgáljuk és csak az egyik programrészből a másikba átvitt paramétereket kell számon tartani a pontos működés megértéséhez. Ez a módszer itt is célravezető.

Vegyük akkor sorra ennek a programnak a részegységeit: a legelső, a képernyő beállítása a megfelelő üzemmódba, de ez már teljesen természetes. Ezután kiírjuk a képernyőre az egyes menüpontokat. Nézzük, hogyan is történik ez: Ugyebár egy `call` utasítással meghívjuk a `Kiir` rutint, ami először is `si` indexregiszterbe tölti a kirakandó szöveg kezdőcímét, `ah`-ba a szint és `di`-be a képernyőcímet, ahonnan a kiírást el kell majd kezdeni. Itt ezt a program nem számolja, mivel ez egy fix érték ami nem változik, ezért a címet közvetlenül a regiszterbe töltjük. De mivel a legritkább esetben áll egy sorból egy menü, ezért a `di` értékét elmentjük a verembe, hogy később könnyebben tudjuk kiszámolni a következő sor kezdőcímét. Ha mindezzel megvagyunk, következhet a karakterek beolvasása a memóriából, és képernyőre írása, hacsak nem ve-

zérőlköd. A rutin a 0-t és a 255-öt használja vezérlésre. A nulla hatására előveszi a veremből a di elmentett értékét ami az éppen írt sor elejére mutat. Ha ehhez hozzáadunk 160-at, éppen egy sorral lejjebb jutunk, így az egyes menüpontok pontosan egymás alatt lesznek és ugyanabban az oszlopban fognak kezdődni. Természetesen a regiszter tartalmát ismét el kell tárolni egy újabb sor címének kiszámításához. Ha nem nullával találkozik, hanem 255-el, akkor a rutin befejezésére ugrik ami abból áll, hogy a di tartalmát kiolvassa a veremből, nehogy benne maradjon a már említett lefagyasási lehetőségek miatt és egy ret utasítással visszatér oda, ahonnan elindítottuk.

A program további teendője már többször fog ismétlődni, így ide egy címke is került, amire később ugrani lehet. A legelső lépés itt, hogy a választást jelző csíkot kitesszük valamelyik menüpontra (ez alapesetben a legelső, de ezen lehet változtatni). Ezt a Csík nevű szubrutin végzi el, ami két majdnem egyforma részből áll. Az elsőben eltünteti a képernyőn lévő csíkot, majd kirakja az újat. Ez a következőképpen történik: a MUTATO2 értékét ha megszorozzuk a sor hosszával (160) és hozzáadjuk az első menüsor címét (160-at), akkor pontosan annak a menünek a helyét kapjuk, amelyiken éppen áll a mutatónk. Azért kell 160-al kevesebbet hozzáadni, mert a mutató kezdőértéke 1 és ha ezt megszorozzuk a sorhosszal, akkor a 160-at kapunk. A kezdőérték egyről való indításának oka, hogy a későbbiek során amikor ezt az értéket egy ciklusban használjuk, akkor ha $cx=0$ értékkel indul, akkor nem egyszer fog lefutni a ciklus, hanem 65536-szor (ez a loop működéséből adódik). Ha kiszámoltuk a csík kezdőcímét, elkezdhetjük a csík kirakását. Ezt a csík hosszának és színének beállításával kezdjük. Itt a szín az eredeti, fekete háttérű a törlés céljából. Ezután a már kiszámolt memóriacímre, ami most nem egy karakterhely, hanem egy színhely mert most a szöveget nem kívánjuk megváltoztatni, csak a színét, kitesszük a mutatót jelképező csíkot. Tehát egy ciklussal minden második helyre beírjuk a megadott színbyte-ot. Ha ezzel megvagyunk, megismétljük a műveletet, de most már az új pozícióval és színnel. Majd a mutató helyét a MUTATO2 változóba írjuk, hogy a következő elmozdításnál le tudjuk törölni a jelenlegit. Ezután természetesen egy ret segítségével visszatér a rutinból. A most következő BIOS rutin ellenőrzi, hogy van-e lenyomott billentyű. Ha van, akkor kiolvassa azt és megvizs-

gálja, hogy a számunkra van-e jelentése. Ha nincs, akkor vissza a figyeléshez. Ha a felfelé nyilat nyomtuk meg, akkor ellenőrizni kell, hogy a mutató nem áll-e a legfelső soron, mert innen följebb már nem léphetünk. Ha nem ott állt, akkor a mutató értékét csökkentjük eggyel. Hasonlóan a lefelé nyílnál is megvizsgáljuk a csík helyét és ha lehet, növeljük a mutató értékét. Amennyiben a menüpont kiválasztására használt **enter**-t nyomtuk le, akkor kilép a körből és az **Enter** címkénél folytatja a program futását. Itt a **cl** regiszterbe tölti a **MUTATO** értékét, **si**-be pedig azt a címet, ahonnan kezdve le lettek tárolva az esetlegesen indítható programok címei. Ez úgy történt, hogy a **CIMEK** címke után wordös formában az egyes programok offsetcímei kerültek letárolásra. Innen egy **mov** utasítással a **bx** regiszterbe töltjük először az első címet, növeljük **si** értékét kettővel, majd a **loop** utasítás a **cx**-nek megfelelően megismétli ezt a műveletet. Ha **cx**-ben 1 volt, akkor a ciklus nem kerül ismétlésre és a **bx** regiszterben marad az első cím. Ha nem egy, akkor a **cx**-től függően a megfelelő értéket kerül a **bx**-be. Ezzel megkerestük a kiválasztott menüpont által végrehajtandó program címét. A feladat már csak annyi, hogy végrehajtsuk. Ezt egy **jmp bx** utasítással tehetjük meg. Ezek a rutinok most csak annyit csinálnak, hogy **si**-be beállítják a kiíratandó szöveg kezdőcímét (természetesen a menüponttól függően) és egy másik kiíró rutin segítségével kiírják azt, majd visszaugranak a **.2_Enter** címkéhez. Az itt használt szövegkirakó eljárás csak annyiban különbözik az előzőtől, hogy a szöveg elején található 2 byte, ami a kirakandó szöveg koordinátáját tartalmazza. Ebből a program a már megszokott módon kiszámolja a képernyőcímet. A szöveg végét a 0 kód jelzi. Visszatérés után egy kérdés kerül kirakásra, mégpedig, hogy akarunk-e újból választani. Itt figyelni, ha az **n** gombot nyomtuk le, ugrik a kilépésre, ha az **i** gombot akkor előlről kezdi a programot, egyébként újabb billentyű várása következik. A programban nem a karakterek **ascii** kódja lett figyelve, hanem a **scan** kód mivel előfordulhat, hogy be van nyomva a **Caps Lock** és ekkor az **n** billentyűt hiába figyeljük mivel leütésekor **N** kerülne beolvasásra. Ellenben a **scan** kód a billentyűzet gombjait jelenti, nem pedig a rajtuk lévő betűt.

Memóriakezelés:

A programozásnak egy másik fontos eleme a memória kezelése. Gyakran előfordulhat ugyanis, hogy egy program

működése közben például a lemezről egy hosszabb szöveget akar beolvasni, ami az adott programszegmensbe már nem férne bele, vagy más egyéb művelethez memóriára lenne szüksége. Ekkor jönnek a bonyodalmak, ugyanis ki tudja miért, de amikor a DOS betölt a memóriába egy programot, akkor az összes szabad memóriát lefoglalja, majd betölti a programot, de a nem használt memóriaterületet nem szabadítja fel. Ez azt jelenti, hogy ott van az üres memóriaterület, de használhatatlan. Ahhoz tehát, hogy a memória egy részét saját céljainkra le tudjuk foglalni, előbb fel kell szabadítani a nem használt részeket. Ez egy egyszerű BIOS művelet, ahol megadjuk az általunk használt program szegmenscímét és hosszát, a többit már elvégzi a gép. Ezután a maradék memória a rendelkezésünkre áll. Ügyelni kell, hogy a program hosszát nem byte-ban kell megadni, mert így maximum 64K területet használhatnánk, hanem paragrafusban. Egy paragrafus nem más mint 16 byte, tehát a programunk méretét, el kell osztani 16-al és megkapjuk a hosszát. Nem árt azonban egy kicsivel többet lefoglalni, főleg ha egy szegmensben belül van a kód, az adat és a stack, mivel így egy pár veremművelet után felülírhatjuk a programunkat. Tehát mindig annyival többet célszerű megtartani a saját programunk, amennyit maximálisan használhat. A megmaradt területtel ezután már szabadon garázdálkodhatunk. Az első lépés a felszabadítás után, hogy lefoglalunk egy általunk meghatározott méretű részt a szabad memóriából, amit azután majd felhasználhatunk céljainkra. Ez szintén egy BIOS hívással történik, meg kell adni, hogy mennyi memóriát akarunk a géptől, és ő ezt kiutalja nekünk ha van annyi. Ha nincs, akkor a legnagyobb lefoglalható memóriablokk méretét kapjuk válaszként. Egyébként pedig a lefoglalt blokk méretét a szegmenscímével együtt. Ha ez is megvan, miénk a lehetőség, hogy azt kezdjük vele amit akarunk. A továbbiakban ezt majd arra fogjuk felhasználni, hogy a lemezről betöltünk egy szöveget, amit majd kiírunk a képernyőre. De ahhoz, hogy ezt meg tudjuk tenni, meg kell még ismerni a file kezelést.

File kezelés:

A lemezen lévő file-okat többféleképpen kezelhetjük. Ezek a lehetőségek közül most a legegyszerűbbel fogunk ismerkedni. Ezt szintén a BIOS végzi három lépésben. Az első a file megnyitása, a

második az írás vagy olvasás és a harmadik a file lezárása. A művelet végzése során fontos információt tartalmaz a file azonosító (handle), amit a file megnyitása után kapunk a géptől, és a másik két művelet számára feltétlenül szükséges. A megnyitáshoz nem szükséges más, mint a file neve és egy megnyitási mód, ami meghatározza, hogy milyen műveletet akarunk végezni a file-al. A rutinból való visszatéréskor `ax` regiszterbe adja a file azonosító számot, amit majd használni fognak a további rutinok. Ezután következik az írás vagy olvasás ahol meg kell adni a mozgatandó byte-ok számát és helyét. Ha elvégeztük a műveletet, le kell zárni a file-t!

A most következő program lefoglal a memóriából egy 64K hosszú területet, ahová a lemezről be fog tölteni egy nevet tartalmazó adatfile-t, amit egy rövid program segítségével kiíratunk a képernyőre.

[Program 11]

```

Pelda11      Segment                ;Szegmensdefiníció.
                assume cs:Pelda11,ds:Pelda11 ;Cs és ds regiszterek
                ;Hozzárendelése a címkekéhez.

Start:      mov  ax,Pelda11          ;Ds regiszter beállítása a
                mov  ds,ax              ;kód elejére.

                mov  ax,3               ;Képernyőtörles.
                int  10h

                mov  ax,4a00h           ;A program által lefoglalt
                mov  bx,1024            ;memóriaterületet 16K
                int  21h                ;mértűre változtatjuk.
                mov  ax,4800h           ;Kérünk a DOS-tól egy
                mov  bx,4096            ;64K méretű memórialapot.
                int  21h
                mov  word ptr [MEMOCIM],ax;Ennek szegmenscímét a
                ;memóriaváltozóba tesszük.

                mov  ax,3d00h           ;A FILENEV alatt tárolt nevű
                mov  dx,offset FILENEV  ;file megnyitása olvasásra.
                int  21h
                mov  word ptr [FILESZAM],ax;A visszakapott file azonosító
                ;információt a memóriába mentjük.
    
```


Az Assembly programozás alapjai

	push ds	;A ds regiszter értékét
	mov bx,ax	;elmentjük, mivel mi nem
	mov ax,word ptr [MEMOCIM]	;a ds által mutatott
	mov ds,ax	;szegmensre kívánunk betölteni
	mov ax,3f00h	;hanem az általunk lefoglalt-
	xor dx,dx	;ra. A DOS hátránya viszont,
	mov cx,0ffff	;hogya a betöltendő file
	int 21h	;helyet a ds:dx regiszterben
		;kell megadni a rutinnak.
	 pop ds	 ;A ds visszaállítása.
	 mov ax,3e00h	 ;A file lezárása.
	mov bx,word ptr [FILESZAM]	
	int 21h	
	 xor di,di	 ;A kiírás címét es:di,
	xor si,si	;az olvasását a ds:si
	mov ax,0b800h	;fogja tartalmazni, de ehhez
	mov es,ax	;ds értékét ismét el kell
	push ds	;menteni.
	mov ax,word ptr [MEMOCIM]	
	mov ds,ax	
	mov ah,14	;Szinbeállítás.
	xor bx,bx	
 .1_Pelda11:	mov al,ds:[si]	 ;A kiírató rutin.
	inc si	
	cmp al,13	;Ha enter kóddal találkozik,
	jz .2_Pelda11	;ugrik a soremelő program-
		;részhez.
	cmp al,255	;A 255 a szöveg végét jelenti.
	jz Kilepes	;Ekkor kilépés.
	 mov es:[di+bx],ax	 ;A betű kiírása és a
	add bx,2	;pozíciók állítása.
	jmp .1_Pelda11	;Új betű.
 .2_Pelda11:	xor bx,bx	 ;A 13 utáni 0ah kódot átlépve
	inc si	;a következő sor elejére
	add di,160	;állítja az írás címét.
	jmp .1_Pelda11	
 Kilepes:	pop ds	 ;A ds eredeti értékének
		;visszaállítása.

IBM PC Gyakorlati Assembly

xor	ax,ax	:Billentyűvárás.
int	16h	
mov	ax,4c00h	:Kilépés a DOS-hoz
int	21h	
FILESZAM:	dw	?
MEMOCIM:	dw	?
FILENEV:	db	"nevek.txt",0 :A betöltendő file neve.
Pelda11	Ends	:A szegmens vége.
	End	Start :A program vége.

Mint az látható, a program hasonlóképpen kezdődik, mint az eddigiek. A képernyőtörítés után azonban meghívásra kerül a 21h rutin, ami tulajdonképpen nem más, mint a DOS-nak egy része. Ugyanis ez a megszakítás kezeli a legtöbb DOS funkciót. Ezek közül mi most a 4ah alfunkciót használjuk, ami a lefoglalt memória méretének megváltoztatására szolgál. Bx regiszterben kell megadni az új hosszt (paragrafusokban) és es regiszterben a szegmenscímet. Nos ezt hiába keresnénk, nem lett megadva mivel a program induláskor ds és es regiszterekbe az úgynevezett PSP (majd később) -ami a program előtt található- szegmenscímét rakja. Ezért nem kell külön beállítani ezt az értéket, és a ds-t ezért állítjuk be .exe programoknál külön a kód elejére. A program által használható memória hosszát 1024 paragrafusra azaz 16K-ra állítottuk be, Ennyi biztos elég lesz.

Ezután a file betöltése céljára kérünk a géptől egy kis memóriát, méghozzá 64K-t. És ha megkaptuk, eltároljuk a szegmenscímét, hogy később tudjuk használni.

Most következik a használandó szövegfile megnyitása. Ezt a műveletet a 3dh DOS funkció végzi. Az al regiszterbe nullát töltünk, ami azt jelenti, hogy a file-t csak olvasásra nyitjuk meg. Ha itt 1 lenne az a csak írást, a 2 az írás-olvasást jelenti. Az al regiszter többi bitjével nem foglalkozok, mivel az más könyvekben sőt is megtalálható. A dx regiszterbe pedig azt a címet kell rakni, ahol megtalálható a betöltendő file elérési útja, neve a kiterjesztéssel aminek a végét egy nulla jelzi. A rutinból való visszatérés-

kor **ax** regiszter fogja tartalmazni a file azonosító számot amit elmentünk, mivel később is szükség lesz rá. A következő lépés a szöveg betöltése. Ehhez **ds:dx** regiszterbe be kell állítani annak a helynek a címét, ahová be akarjuk tölteni az adatot és **cx**-be az adatblokk hosszát. Ez itt most nem lett pontosan beállítva, de ha többet írunk be ide, nem történik nagy katasztrófa, legfeljebb nem tölt be annyit, csak amilyen hosszú a file. Egyébként a betöltött byte-ok számát megadja a rutin visszatéréskor, de erre most semmi szükség. Ami viszont fontos, hogy ha **ds** értékét megváltoztattuk, semmi olyan műveletet nem végezhetünk, ami a memóriaváltozókat és egyéb a régi adatszegmensben lévő adatot befolyásolna, mivel a **ds** most máshová mutat. Ezért a szegmensregiszter értékét is a verembe kell lementenünk mivel ezt a változtatás nem befolyásolja. A file betöltése után visszaállítjuk az eredeti **ds** értéket mivel a fileszám egy memóriaváltozóban lett eltárolva, és ezt csak így tudjuk kiolvasni a file lezárásához. Ezzel a file betöltése megtörtént, most már a rendelkezésünkre áll szöveg a lefoglalt memóriaterületen.

További tennivalónk, hogy a beolvasott neveket a képernyőn megjelenítsük sorban egymás alatt. Ehhez nem árt, ha ismerjük a DOS alapú szövegszerkesztők által készített szövegformátumokat. Tudni kell róluk, hogy minden sor végére ahol enter nyomtunk, egy 13 és egy 10 kódot rak. Ebből a 13 az enter a 10 pedig a nyomtatók számára az új sort jelentik. A szöveg végére egy 255 kódot helyeztem, amit a programból figyelve jelzi a felsorolás végét.

A kiírató rutin hasonlóképpen működik, mint az már régebben történt, azzal a különbséggel, hogy a sorokon belüli címet a **bx** regiszter növelésével változtatjuk, ez megkönnyíti a soremelést mivel így csak nullázni kell a **bx** regisztert és hozzáadni **di**-hez a sorhosszt. Az egyetlen dolog, amire oda kell figyelni hasonló esetekben, hogy a **ds** regiszter értékét megváltoztattuk, így memóriaváltozók használata nem ajánlott.

Adódhat olyan esemény, amikor egy szó, mondat helyét kell meghatározni egy hosszabb szövegben, adatbázisban stb. Illetve vizsgálnunk kell, hogy egyáltalán megtalálható-e az adott szöveg. Ilyenkor a megoldás, hogy letároljuk a keresendő

kifejezést és sorban összehasonlítjuk a dokumentum minden részével mindaddig, míg rá nem lelünk a kívánt szövegre. Ezen feladat megoldásánál nagyon hasznos a **cmpsb / cmpsw** illetve a **scasb / scasw** utasítások a **rep** kiegészítéssel. A jelentésükre majd a mintaprogram ismertetése után térek ki. A most következő program egy eltárolt mondatban keres egy szót, és kiírja, hogy a mondat hányadik karakterétől kezdődik.

[Program 12]

Pelda12	Segment	;Szegmensdefiníció
	assume cs:Pelda12,ds:Pelda12	;Cs,ds kijelölése
Start:	mov ax,Pelda12	;Ds,es regiszter beállítása
	mov ds,ax	;a kód elejére.
	mov es,ax	
	cld	;A d jelzőbitet töri.
	mov cx,47	;A vizsgált szöveg hossza.
	mov di,offset SZOVEG	;A szöveg és a keresett
	mov si,offset SZO	;szó címének beállítása.
	mov al,[si]	;Al regiszterbe a keresett
		;szó kezdőbetűjét tölti.
.1_Pelda12: repnz scasb		;Es:di címen lévő karaktert
		;összehasonlítja al értékével
		;és ha nem egyezik, növeli
		;di-t,csökkenti cx-et és
		;ismétli mindezt, míg nem
		;talál egyező betűt, vagy cx
		;értéke nem 0.
	jnz Nincs	;Ha a vizsgálatból való
		;kilépéskor z flag értéke
		;0, akkor végignézte a
		;szöveget és nem talált
		;egyező betűt.
	dec di	;Egy betűvel visszább lép
		;mivel a scasb növelte a
		;talált betű után is eggyel.
	push cx di si	;A regiszterek elmentése,
		;mivel a következő rutin
		;megváltoztatja azokat.

Az Assembly programozás alapjai

	mov cx,6	;A keresett szó 6 betű hosszú.
	repz cmpsb	;Es:di és ds:si által ;mutatott területek összeha- ;sonlítása egymással.
	pop si di cx	;A regiszterek visszatöltése.
	jz Talalt	;Ha a rep cmpsb végrehajtása ;után zero flag 1, az azt ;jelenti, hogy a két szöveg ;meg egyezik egymással.
	inc di	;Ha nem, akkor a következő
	jmp .1_Pelda12	;betűtől folytatni a keresést.
Nincs:	mov si,offset UZENET1	;UZENET1 szöveg kiírása
	call Kiiro	
	jmp Kilepes	;és kilépés.
Talalt:	mov si,offset UZENET2	;UZENET2 kiírása,
	mov ax,di	;a megtalálási pozícióból
	sub ax,offset SZÖVEG	;kivonjuk a szöveg offsetcímét
	push ax	;és ezt elmentjük a számkíró
	call Kiiro	;rutin számára.
	pop ax	;Ax előhívása,
	call Kiiro2	;a szám kiírása.
Kilepes:	xor ax,ax	;Billentyűvárás.
	int 16h	
	mov ax,4c00h	;Kilépés a DOS-ba.
	int 21h	
Kiiro	Proc	
	mov ax,0b800h	;Képernyő-beállítás.
	mov es,ax	
	mov ax,3	
	int 10h	
	xor di,di	
	mov ah,15	
.1_Kiiro:	mov al,[si]	;Az Si címen lévő szöveg
	inc si	;kiírása.

IBM PC Gyakorlati Assembly

```

    cmp  al,0
    jz   .2_Kiiro
    mov  es:[di],ax
    add  di,2
    jmp  .1_Kiiro

.2_Kiiro:  ret

Kiiro     Endp

Kiiro2    Proc

    mov  di,offset SZAMHELY ;Erre a címre fogja letárolni
                                ;a számot a kiírás előtt.

    mov  bx,10                ;Az osztás mértéke.

    xor  cx,cx                ;A számláló nullázása.

.1_Kiiro2: xor  dx,dx          ;A div utasítás a jelen
                                ;esetben dx:ax regiszter
                                ;tartalmat osztja, de
                                ;számunkra hasznos adat csak
                                ;az ax regiszterben van,
                                ;ezért a dx regisztert
                                ;torolni kell.
    mov  [di],dl              ;Az osztás maradékának alsó
                                ;byte-ját a memóriába
                                ;mentjük.

    inc  cx                   ;A számláló növelése.

    inc  di                   ;A következő címre írja a
                                ;következő számot.

    or   ax,ax                ;Ax vizsgálata,
    jnz  .1_Kiiro2            ;ha nem 0, ugrás vissza.

    mov  si,di                ;Si regiszterbe di-1-et
    dec  si                   ;tölünk, mivel ez az utolsó
                                ;értékes szám.

    mov  di,44                ;Kiírási pozíció beállítása.

    mov  ah,15                ;Szinbeállítás.

```

Az Assembly programozás alapjai

```
.2_Kiir2:  mov  al,[si]                ;Al-be tölti az utoljára
                                ;letárolt számjegyet ami
                                ;valójában az első.

          add  al,48           ;Ascii számjeggyé alakítja
          mov  es:[di],ax      ;és kiírja a képernyőre.

          add  di,2            ;Következő pozíció.

          dec  si              ;Előző számjegy.

          loop .2_Kiir2       ;Ismétlés a számjegyek
                                ;számának megfelelően.

          xor  ax,ax           ;Billentyűvárás.
          int  16h

          mov  ax,4c00h        ;Kilépés a DOS-ba.
          int  21h

          ret

Kiir2     Endp

SZOVEG:  db  "Ez az a szöveg, amiben keresni fogunk egy szót."

SZO:     db  "amiben"         ;A keresett szó.

UZENET1: db  "Nincs ilyen szó a vizsgált szövegben.",0

UZENET2: db  "A keresett szó helye:",0

SZAMHELY:db  5 dup (0)        ;A szám átmeneti tárolására
                                ;szolgáló hely

Pelda12  Ends
          End  Start
```

Korábbi programokban már használtuk a `cmp` utasítást, innel két adatot hasonlíthattunk össze. Az iménti példa is megoldható lett volna ezzel, de sokkal hosszabb, bonyolultabb lett volna a program. A PC rendelkezik egy olyan utasítás csoporttal ami egy kicsit összetettebben működik az eddig megismerteknél. Nos nézzük, hogy is működnek ezek: amikor a gép a `scasb`

utasítással találkozik, összehasonlítja az **es:di** címen tárolt adatot (jelen esetben byte-ot) **al** regiszter tartalmával és a jelzőbitek a művelet eredményétől függően állítja be. Továbbá a végrehajtás után növeli (csökkenti) **di** regisztert a **d** flag értékétől függően (**d=0** esetén növeli). Így ha ismételtetjük a **scasb** utasítást, megkereshetjük az adott szövegben az **al** tartalmának előfordulási helyét. Ezt az ismételtetést oldja meg a **rep** kiegészítés ami a végrehajtás során csökkenti **cx** értékét és mindaddig ismétli a mögé írt műveletet, míg **cx** el nem éri a nullát. A mi esetünkben ez nem elegendő, mivel figyelni kell, hogy egyező-e a betű vagy sem. Ezt oldja meg a **repnz** ami mindaddig ismételi, amíg a **z** bit nulla illetve a **cx** regiszter nagyobb nullánál. Tehát ha egyező betűt talál a **scasb**, akkor a zero flag 1 értékre vált, minek hatására a **repnz** abbahagyja az ismétlést. A **cx** regiszterbe a vizsgálandó szöveg hosszát állítjuk, mivel a keresést a szöveg végéhez érve be kell fejezni. A keresési ciklusból való kilépéskor a **z** bit értéke mutatja a kilépés okát. Ha 1 akkor egyező betűt talált, ha 0 akkor a **cx** értéke érte el a nullát.

A másik utasítás amit a keresett szó összehasonlítására használtunk, az a **cmpsb**, ami a **ds:si** és az **es:di** által címzett két byte-ot hasonlítja össze, majd növeli **si** és **di** értékét. A flag itt is a művelet eredményétől függően áll be. Az itt használt kiegészítés, a **repz** csak addig ismételi, amíg a művelet eredménye egyenlő azaz zero flag értéke 1. Illetve míg **cx** nem nulla.

Ezen két utasítás segítségével nagyon egyszerű egy bármilyen hosszú szöveg keresése.

A mintaprogram tartalmaz még egy új utasítást is, ami befolyásolja az előző kettő működését. Ugyanis az, hogy a **rep** hatására a **cx** értéke csökken vagy nő, azt a **d** jelzőbit határozza meg. Ha értéke 1, akkor csökkenteni fogja. Ezért a **d** bitet nullába kell állítani. Ezt a **cld** utasítás végzi el. Ezután a programban beállítjuk a szöveg hosszát, címét és a keresett szó címét. Ezután a szó első betűjét **al** regiszterbe töltjük. Ezzel a kiindulási paramétereket beállítottuk, következhet a keresés. Jelen esetben a keresett szó az "amiben" ezért a **repnz scasb** megkeresi a szövegben előforduló első "a" betűt. A kereső ciklusból való kilépéskor ha **z=0** akkor a szöveg végéhez értünk, ami azt jelenti,

hogy a keresett szó nincs a szövegben. Amennyiben talált a program al-ben tárolt betűt, akkor következik az összehasonlítás. Első lépésben csökkentjük di értékét, hogy arra a helyre mutasson, ahol egyező betűt talált, majd elmentjük a használt regisztereket, mivel a következő programrész is használja ezeket és nekünk a keresés esetleges folytatásához a jelenlegi értékek szükségesek.

A vizsgált szó összehasonlítását a szóhossz beállításával kezdjük, majd a repz cmpsb összehasonlítja a talált betű címétől kezdődő részt az általunk keresett szóval. Amennyiben a 6 karakter bármelyikénél eltérést talál, kilép a repz ciklusból. A vizsgálat után visszatöltjük az elmentett regisztereket és megvizsgáljuk a kilépés okát. Ha $z=1$ akkor a vizsgált szó megegyezik az általunk megadottal. Ha nem, akkor a következő karaktertől (inc di) folytatódik a keresés egy újabb "a" betűig.

Amennyiben a ciklus végigfutott úgy, hogy nem találta meg a kívánt adatot, a Nincs címkére ugrik, ahol a Kiíró rutin segítségével közli ezt velünk.

Hasonlóképpen a Talalt esetében, de itt a szó helyét is kiírja nekünk a már ismert szöveg és szákkíró rutinok segítségével.

Természetesen, ha egy gigantikus méretű adathalmazban akarunk megkeresni valamit, az időbe telhet. Ezért például adatbázis szerű adatok között sokkal könnyebben kereshetünk, ha az adatainkat fix hosszúságú területeken tároljuk. Mit is jelent ez? Vegyünk példának egy telefon regiszter programot. Ha név szerint akarunk kereshetünk, akkor teljesen fölösleges azt a telefonszámok és a címek között is megnézni. Ezért azt szokták csinálni, hogy meghatározzák, hogy a név maximálisan pl.: 30 karaktert, a telefonszám 15-öt, a cím pedig 100 karakter helyet foglalhat el, ezért minden adat számára egy ekkora területet foglalnak le még ha rövidebb is ennél. Ezekután a keresés annyiból áll, hogy megnézzük a név első betűjét és ha nem stimmel, akkor 145 karakterrel arrébb lépünk, ami a következő név első betűjét tartalmazza. Ugyanígy a telefonszám és a cím esetében is. Előnye a dolognak, hogy a fölösleges információt nem vizsgáltuk meg, és ezzel rengeteg időt megspóroltunk.

Ha már szó esett az adatbázisokról, egy kis elméleti információként a rendezésről: az adatok valamilyen szempont szerinti sorbarendezésének alapvetően két típusa van. A valós és a látszólagos. A valós rendezésnél az adatokat fizikailag sorbarendezzük a megadott szempont szerint, míg a látszólagos esetében csak egy sorrendet készítünk, ahol a helyes sorrendet tároljuk le és az adatbázis sorrendjét nem változtatjuk meg. Előnye az utóbbinak, hogy gyorsabb rendezést lehet elérni vele. Az adatok sorbarakására egy ötlet, ami ugyan valószínűleg nem a leggyorsabb, de kipróbáltan jól működik. A teendők, hogy összehasonlítjuk az első és a második bejegyzést egymással. Amennyiben nem megfelelő a sorrendjük, egyszerűen megcseréljük a kettőt. Ezután megtesszük ezt a 2. és a 3. adattal és így tovább egészen az utolsóig. Ezzel azonban még nem lesznek sorban az adatok, mivel egy lépésben nem lehet ezzel a módszerrel az utolsó bejegyzést az első helyre varázsolni, ezért a teljes műveletet meg kell ismételni a bejegyzések száma-1-szer ahhoz, hogy biztosan a megfelelő sorrendet kapjuk.

Előfordulhat, hogy a programunknak a DOS-ból akarunk valamilyen paramétert átadni. Lásd például DIR /P stb. esetében a / jel után valamilyen paramétert lehet megadni. Az elválasztást szolgáló jel feladata csupán annyi, hogy a programból könnyebben meg tudjuk keresni a paramétert, mivel az utasítás és a paraméter között nem biztos, hogy csak egy szóköz van. A példaprogram mindössze annyit csinál, hogy a paraméterként írt részt kiírja a képernyőre a megadott koordinátákra. A program használata: **pelda13 /10,12,kiírandó szöveg**. fontos, hogy fölösleges szüneteket ne rakjunk a / után mivel a program ott a koordinátát fogja keresni.

[Program 13]

```
Pelda13    Segment
            assume cs:Pelda13,ds:nothing

Start:     mov    si,80h                ;A DTA kezdőcímét si-be
            ;teszi.

            mov    cl,[si]             ;A paraméterrész hosszát cx
            inc    si                  ;regiszterbe teszi és a
            ;mutatót eggyel növeli.
```

Az Assembly programozás alapjai

	mov ax,0b800h	;A videomemória kezdőcímet
	mov es,ax	;es regiszterbe tölti.
.1_Pelda13:	mov ah,"/"	;A / jel keresése.
	call Keres	
	mov dx,si	;Dx regiszterbe teszi a ;rutin által megadott címet.
	mov ah,""	;A , karakter keresése.
	call Keres	
	push cx	;Cx ideiglenes elmentése.
	mov cx,si	;Cx regiszterben kiszámoljuk
	mov di,si	;a koordináta számjegyeinek
	sub di,2	;számát, di-ben pedig az
	sub cx,dx	;utolsó számjegy címe lesz.
	dec cx	;Majd kiszámoljuk a koordináta
	call Szamito	;számértékét.
	pop cx	;Hasonlóképpen mint az előbb
	push dx	;tettük, megismételjük a
	mov dx,si	;műveletet a függőleges
	mov ah,""	;koordináta értékkel is.
	call Keres	
	push cx	
	mov cx,si	
	mov di,si	
	sub di,2	
	sub cx,dx	
	dec cx	
	call Szamito	
	pop cx	
	pop di	;Di-be az előzőleg elmentett
		;dx értékét töltjük és
	shl di,1	;megszorozzuk kettővel. Ezzel
		;kiszámoltuk a koordináta
		;vízszintes címét.
	mov ax,dx	;A függőlegest a szokott
	mov bl,160	;módon megszorozzuk 160-al
	mul bl	;és hozzáadjuk di-hez.
	add di,ax	

IBM PC Gyakorlati Assembly

	mov ax,3	;Képernyőtörlés.
	int 10h	
.2_Pelda13:	mov ah,15	;Fekete alapon fényes fehér
	mov al,[si]	;színnel cx által mutatott
	es:[di],ax	;számú karaktert írunk
	inc si	;es:[di] címtől kezdődően.
	add di,2	
	loop .2_Pelda13	
	xor ax,ax	;Billentyűvárás.
	int 16h	
	mov ax,4c00h	;Kilépés.
	int 21h	
Keres	Proc	
.1_Keres:	mov al,[si]	;Ds:[si] címen lévő karaktert
	inc si	;összehasonlítja ah értékével.
	dec cl	;növeli si-t és csökkenti a
	cmp al,ah	;még hátralévő karakterek
	jnz .1_Keres	;számát. Ha nem egyezett a
		;két karakter, akkor ismétli
		;a ciklust mindaddig míg meg
		;nem találja a keresett
		;karaktert.
	ret	
Keres	Endp	
Szamito	Proc	
	mov bl,1	;A legkisebb helyiérték.
	xor dx,dx	;Dx nullázása.
.1_Szamito:	mov al,[di]	;A di által mutatott számjegyet
		;al regiszterbe tölti.
	dec di	;Csökkenti di-t
	sub al,48	;Mivel al-ben nem egy számjegy
		;van, hanem annak ascii kódja,
		;ezért abból ki kell vonni
		;48-at, hogy a számjegyet

Az Assembly programozás alapjai

```
                                ;kapjuk eredményül.

mul    bl                        ;Ezt az értéket megszorozzuk
                                ;az aktuális helyiértéknek
                                ;megfelelő szorzószámmal
add    dx,ax                     ;és hozzáadjuk dx-hez.

mov    ax,10                     ;A következő helyiértéket
mul    bl                        ;úgy kapjuk, hogy a jelenlegit
mov    bx,ax                     ;megszorozzuk 10-el.

loop   .1_Szamito               ;Ha van még számjegy, ugrás
                                ;vissza.

ret

Szamito    Endp

Pelda13    Ends
            End    Start
```

Amint arról a memóriakezelésnél már szó esett, a program betöltése után a ds és es regiszterekbe nem a program címe, hanem egy úgynevezett PSP szegmenscíme kerül. A PSP teljes leírása megtalálható a függelékben, itt csak az un. DTA érdekes a számunkra, ami a PSP 128. byte-jától kezdődik és hosszát a legelső byte-on tárolt érték mutatja. Ez a terület azt a célt szolgálja, hogy a DOS parancs után begépelte paramétereket eltárolja. A most következő programban a DOS parancs utáni első karakter a 81h címen lesz. Ezt kihasználva ci regiszterbe betöltjük a paraméterrész hosszát és megkeressük az elválasztó "/" jelet ami a paraméterek kezdetét jelzi. Ha ezt megtaláltuk, akkor meg kell keresni az eső "," karaktert, és a két cím közötti karakterekből kell egy számot készíteni. Mivel megkerestük a szám elejét is és a végét is, meg tudjuk határozni, hogy hány karakterből áll. Amennyiben egynél több, akkor az egyes helyiértékeket a megfelelő szorzószámmal meg kell szorozni és ezeket összeadni, hogy a valós számértéket kapjuk. Ezt természetesen meg kell tenni a második számmal is és ha már mindkettőt ismerjük, ebből ki kell számolni a kiírási címet. Ezután a második vessző utántól kezdődően ki kell írni a szöveget a kiszámolt címre. A szöveg hosszát a cx értéke mutatja, amit az egyes kereséseknel termé-

szetesen csökkenteni kellett, hogy mindig az aktuális értéket mutassa.

Ilyen módon írhatunk akár decimális-hexadecimális szám konvertert vagy bármilyen külső paraméterrel működő programot.

Programjainkban nem árt, ha például egy hibára nem csak egy szöveggel, hanem egy hangjelzéssel is felhívjuk a felhasználó figyelmét. Ugyan a PC beépített hangszórója túlzott zenélgetésre nem ad lehetőséget, azért egy kis ügyességgel egészen szép hatások csikarhatók ki belőle.

A számítógépben található egy 8253 jelzésű integrált áramkör, ami három 16 bites osztót foglal magába. Az első csatornája a 8259-es megszakítás kezelőre lett kötve, ami így másodpercenként 18.2-szer aktivizálódik.

A második csatorna a 8237-es DMA vezérlőre van kötve. Ezt módosítani nem célszerű, mivel a memóriafrissítést is ez végzi és ha megváltoztatjuk, elveszhet az összes adatunk.

A számunkra most a harmadik csatornája fontos, ami a beépített hangszóróra lett kötve. Tehát ha beállítunk egy adott osztási arányt és bekapcsoljuk a hangszórót, akkor ott az oszcillátor frekvencia (1.19318 MHz) megfelelően leosztott értéke jelenik meg.

A hangszóró bekapcsolását, az osztási arány beállítását stb. úgynevezett portműveletek segítségével érhetjük el. Egy portra való adat írás hasonlít a memóriába való íráshoz, csak itt nem egy RAM van a vezeték végén, hanem egy áramkörü egység, ami figyelni az adott portcímet, és ha az neki szól, akkor az adatvezetéről kiolvassa a küldött értéket.

Így a 8253-as programozására a 43h port szolgál. Az ide írt adat egyes biteinek jelentése:

- 0.bit: 0 esetén bináris, 1 esetén BCD számlálás.
- 1-3. bitek: A kívánt üzemmód száma.
- 4-5. bitek: 00 - A számláló értékét regiszterbe írja.

Az Assembly programozás alapjai

- 01 - Az osztó alsó byte-jának írása/olvasása.
- 10 - Az osztó felső byte-jának írása/olvasása.
- 11 - Az osztó alsó majd felső byte-jának írása/olvasása

6-7. bitek: A kiválasztott csatorna száma (0-2)

A számláló értékét akkor érdemes regiszterbe mentetni kiolvasás előtt, ha mind az alsó, felső byte értékét ki kívánjuk olvasni úgy, hogy azok közben ne változhassanak meg. Ilyenkor a gép az aktuális értéket egy belső regiszterbe menti, amit azután kiolvashatunk. De a számláló programozása részletesebben megtalálható Az IBM PC-k BELSŐ FELÉPÍTÉSE című könyvben és még sok másiban. Az egyes csatornák osztási arányát a 40h-42h portokon keresztül állíthatjuk.

A következő feladatunk a beállítás után, hogy bekapcsoljuk a hangszórót. Ezt a 8255 típusú 24 bites programozható I/O kontroller megfelelő programozásával tehetjük meg. A PC-ben ez az áramkör 3 darab 8 bites részre van bontva, amiből a másodiknak az alsó két bítje befolyásolja a hangszórót. A másik 6 a billentyűzetet, RAM paritást stb. Ezért ügyelni kell arra, hogy ezen biteket ne változtassuk meg. Ezt úgy érhetjük el, hogy írás előtt kiolvassuk a port értékét, az alsó két bitet AND vagy OR művelettel megfelelően beállítjuk és ezután írjuk vissza az értéket. Így csak a hangszórót vezérlő biteket befolyásoljuk. A 8255 leírása szintén megtalálható az említett könyvben vagy más szakirodalomban.

Nos ezek után a program, ami nem tesz mást, mint csippan egyet kb. 1 kHz-es frekvenciával.

[Program 14]

```
Pelda14      Segment
              assume cs:Pelda14,ds:nothing

Prg_8255     equ    61h                ;A 8255 portcíme
Prg_timer    equ    43h                ;Az osztó programozásának
Timer        equ    42h                ;és beállításának portcíme.
```

IBM PC Gyakorlati Assembly

```
Start:  mov  al,10110110b      ;A Timer2 kijelölése osztónak.
        out  Prg_timer,al

        mov  ax,1193         ;Az osztási arány beállítása
        out  Timer,al       ;kb 1000 Hz-re.
        mov  al,ah
        out  Timer,al

        in   al,Prg_8255     ;A hangszóró bekapcsolása
        or   al,00000011b
        out  Prg_8255,al

        mov  cx,0ffffh      ;0ffffh értékű holtciklus
.1_Pelda14: loop .1_Pelda14 ;ami a csippanás hosszát
                               ;határozza meg.

        in   al,Prg_8255     ;A hangszóró kikapcsolása.
        and  al,11111100b
        out  Prg_8255,al
        mov  ax,4c00h       ;Kilépés a DOS-ba.
        int  21h

Pelda14  Ends
        End  Start
```

A lista elején mindjárt egy újdonság, amit eddig még nem használtunk, az `equ` egy címkével. Ez nem csinál mást, mint a magasabb szintű nyelvekben a konstans. Tehát fordításkor a program, ha a szövegben az `equ` előtt álló címkével találkozik valahol, akkor az `equ` után írt számot fogja oda befördíteni. Előnye csak az áttekinthetőség szempontjából van, mivel így mindig tudjuk, hogy az adott portcím mihez tartozik. Továbbá van két új assembly utasítás is, az `in` segítségével egy portról olvashatunk be egy értéket, az `out` pedig ennek az ellenkezője, vele egy adatot küldhetünk a kiválasztott porton keresztül.

A programban semmi különlegesség nincs ezeken kívül. Első lépésben felprogramozza az osztót, majd beállítja az osztási arányt és bekapcsolja a hangszórót. Ezután egy holtciklus segítségével vár egy kicsit majd kikapcsolja és kilép.

A memória rezidens programok működése:

A memória rezidens programok annyiban különböznek az eddig megismertektől, hogy a programból való kilépéskor nem törlődik ki a memóriából, hanem ott marad. Kombinálni szokták a dolgot valamelyik int rutin átirányításával. Mint arról már felületesen szó esett, az int rutinok címei egy táblázatban vannak letárolva. Minden rutinhoz egy szegmens és egy offsetcím, azaz összesen 4 byte tartozik. Ha ezt a címet mi megváltoztatjuk, akkor nem az adott funkció hajtódik végre, hanem az a program, aminek a címét megadtuk. Ezen módon átirányíthatjuk a saját programunkra is. Ilyenkor gondoskodni kell arról, hogy a saját programunk végezze el az adott rutin feladatát, vagy pedig a programunk végrehajtása előtt vagy után indítsa el az eredeti BIOS rutint is. Egy int híváskor első lépésben a verembe mentődik az állapotregiszter (flag) és a hívó program int utáni sorának címe a visszatérés céljából, majd a táblázatból kiolvassa a rutin címét és egy távoli ugrással átadja a vezérlést neki. A BIOS rutin végén egy iret utasítással tér vissza a hívó programhoz, minek hatására visszatöltődik a flag is.

A memória rezidens programoknál alapvető szempont a rövideg. Ezért érdemes a programot .com formátumúra írni. A programból való kilépés is egy kicsit másként történik akkor, ha azt szeretnénk, hogy továbbra is maradjon a memóriában a kód lényeges része. Ilyenkor meg kell adni a programunk szegmenscímét és az első olyan byte offsetcímét, amire már nincs szükségünk, ezután nem egy int21h hanem egy int27h ROM BIOS hívással lépünk ki, melynek hatására a megadott rész a memóriában marad.

A vezérlésátadásra gyakran használják az 1ch BIOS rutin átirányítását, amit a gép másodpercenként 18.2-szer végrehajt. Ez alkalmazható például akkor is, ha azt szeretnénk, hogy a pontos idő mindig ki legyen írva a képernyő valamely részére. Vigyázni kell azonban, hogy egy int rutinból egy másik int rutin nem hívható, ezért ha rezidens programot írunk és mégis szükség lenne egy BIOS hívásra akkor vagy megírjuk azt a műveletet, amit a BIOS rutin végezne, vagy hasonló módon hajtjuk végre, mint az eredeti rutint, azaz szimulálunk egy int hívást, ami abból áll, hogy

egy **pushf** utasítással elmentjük a **flaget** és egy távoli **call** utasítással indítjuk el a kívánt rutint.

A 15. programban egy gyakori problémára adok egy megoldási lehetőséget, mivel velem is gyakran előfordul az, hogy figyelem a winchester működését visszajelző ledet, hogy tudjam, él-e még a rendszer, vagy lefagyott. Egyes vírusok tevékenységét is ki lehet szűzni így. A megoldás, amit ez a rövidke rutin kínál az, hogy minden lemezműveletnél csippan egy rövidet a gép. Ez egy kicsit furcsán hat, amikor a gép keres egy programot, és egy rakás tartalomjegyzéket végignézi, vagy egyéb olyan műveletnél, ahol egymás után sűrűn fordul a lemezegységhez, ekkor ugyanis egy sorozatos csipogást, esetenként recsegést produkál, de a programot kedvünkre megváltoztathatjuk, csipogás helyett akár a képernyőre írhatunk egy jelet vagy kinek mi tetszik.

[Program 15]

```

Pelda15   Segment                               ;Szegmensdefiníció
assume cs:Pelda15, ds:Pelda15                   ;Cs,ds hozzárendelése.
org 100h                                       ;A program kezdőcíme.

Start:    jmp Init                               ;Az installáló program
                                                ;indítása.

Drive_Int dd ?                                   ;Az eredeti ROM rutin címének
                                                ;kihagyott 4 byte-os rész.

Rutin     Proc Far                               ;Távoli rutin kezdete.
assume cs:Pelda15,ds:Nothing                   ;A rutinban nem használunk
                                                ;adatszegmenst.

pushf                                           ;A flag és a programunkban
push ax cx                                       ;használt regiszterek mentése.

mov al,10110110b                               ;A Timer2 kijelölése osztónak.
out 43h,al

mov ax,500                                       ;Az osztási arány beállítása
out 42h,al
mov al,ah
out 42h,al
    
```

Az Assembly programozás alapjai

```

in    al,61h                ;A hangszóró bekapcsolása
or    al,00000011b
out   61h,al

mov   cx,0fffh              ;0fffh értékű holtciklus
;ami a csippanás hosszát
;határozza meg.
.1_Pelda15: loop .1_Pelda15

in    al,61h                ;A hangszóró kikapcsolása.
and   al,11111100b
out   61h,al

pop   cx ax                 ;Az általunk használt
;regiszterek eredeti értékének
;visszatöltése

call  Drive_Int             ;Az eredeti BIOS rutin
;indítása
iret                                     ;Visszatérés a megszakításból.

Rutin: Endp                 ;A távoli rutin vége.

Init: cli                   ;Külső megszakításkérések
;engedélyezésének tiltása.

mov   ax,3513h              ;Az Int13 rutin címét
int   21h                   ;ES:BX-be tölti
mov   word ptr [Drive_Int],bx ;és a Drive_Int címre írja.
mov   word ptr [Drive_Int+2],es

mov   ax,2513h              ;Az int13h rutin címét állítsa
mov   dx,offset Rutin       ;át a Rutin címére.
int   21h

sti                                     ;Megszakítást engedélyezi

mov   dx,offset Init        ;Az első fölösleges byte címe.
int   27h                   ;A program befejezése úgy,
;hogy maradjon rezidens.

Pelda15: Ends               ;A szegmens vége
End   Start                 ;A program vége

```

A programban a **13h** rutint csapoltuk meg egy kicsit, amiről tudni kell, hogy az összes lemezművelet végrehajtása ezen keresztül hajtódik történik. Tehát ha a rutin ugrási címét ideiglenesen átirányítjuk a csippanó rutinra, akkor minden lemezművelet végrehajtásakor elindul az általunk írt kód is.

A program egy **jmp** utasítással kezdődik aminek hatására átugorja a rezidens csipogó részt, ugyanis az első feladat az, hogy a program rezidenssé válása előtt kiolvassuk az eredeti rutin ugrási címét ahhoz, hogy egy **call** utasítással el tudjuk majd indítani azt. Továbbá át kell írunk azt a címet a mi rutinunkra. Vigyázni kell azonban arra, hogy a cím átírása közben nem érkezhessen semmilyen megszakításkérés a kiválasztott funkcióhoz, mivel ekkor problémás dolgok történhetnek. A megoldás az, hogy egy **cli** utasítással letiltjuk a megszakításkérések elfogadását majd az átírás befejeztével egy **sti** segítségével engedélyezzük azt.

Ha az átírás megtörtént, cs regiszterben megadjuk a kód szegmenscímét (azaz ezt már beállítottuk a program elején) és **dx** regiszterben pedig az első olyan byte címét, ami már nem szükséges, tehát az installáló rész kezdőcímét. Ezután a **27h** DOS hívással úgy lépünk ki a programból, hogy az az elejétől a kijelölt részig a memóriában marad. Természetesen ezután visszatér a DOS-hoz. Mint látható, a program aktív részét nem indítottuk el. Ennek oka, hogy ezt a gép végzi el ha valamilyen lemezműveletet végzünk, ekkor ugyanis meghívásra kerül a **13h** rutin, aminek címét az installáló programrész átírta, ennek hatására a megadott címtől elindul a rutin.

A főprogram legfontosabb lépése a használt regiszterek elmentése, ugyanis a megszakítás kezelő rutinnak nem szabad észrevennie, hogy mi beépültünk elé. Ezt úgy tehetjük meg, hogy azokat a regisztereket, amiket a hang kiadása során használunk, eltesszük a verembe és a hang kiadása után kiolvassuk onnan, a **13h** rutin úgy fut le, mintha semmi sem történt volna előtte. A hang kiadása úgy történik, mint az előző programban. Fontos, hogy a program elején elmentjük a **flaget** és két használt regisztert. A hang kiadása után csak a regisztereket olvassuk vissza. Ennek lényege, hogy így csapjuk be a gépet, mivel egy int végrehajtásakor is elmentődik a **flag** az ugrás előtt, ami történhet egy **jmp**

utasítással is, de ekkor az utána következő íret nem szükséges mivel a program egyből a DOS-hoz tér vissza. Formai követelmény, hogy a végrehajtandó rutinnál jelölni kell, hogy az egy távoli rutin (nem abban a szegmensben fog elhelyezkedni, ahol az `int13h`-t hívó program). Ezt jelölhetjük egy `label far` címkével vagy ahogyan az a programban is megtörtént.

Assembly utasítások összefoglalása:

Az eddig ismertetett példaprogramokból elég sok utasítást meg lehetett ismerni, azonban ez még koránt sem az összes. Azért, hogy ne legyen olyan, ami ebben a könyvben nem szerepel, a most következő utasítás ismertetőbe a 8086-os processzor teljes készlete bemutatásra kerül és egy-két 286-os is, melyekhez nem szükséges a védett mód ismerete. Ezen ismertetőbe megtalálható az utasítások leírása, végrehajtásának közelítő időtartama illetve az általa befolyásolt flag bitek.

AAA "Ascii Adjust after Addition" azaz két ascii számjegy összeadása után az eredményből az AAA utasítás `al` ah regiszterekben két pakolatlan BCD számot hoz létre. Ha tehát például mi a "8"-hoz adtuk a "7"-et, akkor az az összeadás a következőképpen történik: $38h+37h=6Fh$, az AAA művelet végrehajtása után `ah` regiszterben 1 `al`-ben pedig 5 lesz. Azaz `ax` regiszterben az 15 pakolatlan BCD számot kapjuk.

Módosított flagek: `cf af` (a többbit nem meghatározható módon befolyásolja)

Végrehajtási idő (órajel ciklus): 4

AAS "Ascii Adjust after Subtraction" Két ascii számjegy kivonása után az eredményt az előző utasításhoz hasonlóan BCD formátumra hozza.

Módosított flagek: `cf af` (a többbit nem meghatározható módon befolyásolja)

Végrehajtási idő (órajel ciklus): 4

AAD "Ascii Adjust before Division" azaz BCD számok előkészítése osztáshoz oly módon, hogy az ah al regiszterekben lévő pakolatlan BCD szám nagyobbik helyiértékét (ah) megszorozza 0ah-val és hozzáadja al-hez. Így a szám hexadecimális alakját kaptuk.

Módosított flagek: **sf zf pf**

Végrehajtási idő (órajel ciklus): 60

AAM "Ascii Adjust after Multiplication" az AAD utasítás fordítottja, ez a szorzás után kapott hexa számot alakítja át pakolatlan BCD számmá.

Módosított flagek: **sf zf pf**

Végrehajtási idő (órajel ciklus): 80

DAA "Decimal Adjust after Addition" Két BCD szám összeadása során is előfordulhat, hogy az eredmény nem felel meg a valóságnak, mivel a számítógép a két számot nem BCD hanem hexa számnak kezeli, így például a 28h és a 39h számok összeadása során a kapott eredmény 61h. Ha ezután végrehajtjuk a DAA műveletet, akkor a gép kiigazítja ezt és 67h-t csinál belőle.

Módosított flagek: **sf zf af pf cf**

Végrehajtási idő (órajel ciklus): 4

DAS "Decimal Adjust after Subtraction" két BCD szám kivonása után az eredményt BCD alakra igazítja mint a DAA.

Módosított flagek: **sf zf af pf cf**

Végrehajtási idő (órajel ciklus): 4

- DEC** "Decrement" operandus értékének csökkentése eggyel.
Módosított flagek: of sf zf af pf
Végrehajtási idő (órajel ciklus): 2-13
- INC** "Increment" operandus értékének növelése eggyel.
Módosított flagek: of sf zf af pf
Végrehajtási idő (órajel ciklus): 2-23
- ADD** "Addition" a második operandus értékét az elsőhöz adja.
Módosított flagek: of sf zf af pf cf
Végrehajtási idő (órajel ciklus): 3-25
- SUB** "Subtraction" a második operandus értékét kivonja az elsőből.
Módosított flagek: of sf zf af pf cf
Végrehajtási idő (órajel ciklus): 3-37
- ADC** "Add with Carry" az első operandushoz hozzáadja a másodikat és a carry flag tartalmát.
Módosított flagek: of sf zf af pf cf
Végrehajtási idő (órajel ciklus): 3-25
- SBB** "Subtraction with Borrow" op1 értékéből kivonja op2 értékét és a carry bit tartalmát.
Módosított flagek: of sf zf af pf cf
Végrehajtási idő (órajel ciklus): 3-37

DIV "unsigned Division" előjel nélküli osztás. 8 bites művelet esetén az osztandó számot ax regiszter tartalmazza az osztót pedig bl és az eredményt al-be a maradékot pedig ah-ba kapjuk, 16 bites művelet esetén az osztandót dx:ax tartalmazza az osztót pedig a bx regiszter, az eredményt az ax, a maradékot pedig dx regiszterben kapjuk.

Módosított flagek: nem meghatározott.

Végrehajtási idő (órajel ciklus): 0-162

MUL "unsigned Multiplication" előjel nélküli szorzás, hasonlóan az osztáshoz 8 bites szám esetén a szorzót bl a szorzandót al tartalmazza, az eredményt ax regiszterben kapjuk.

Módosított flagek: of cf

Végrehajtási idő (órajel ciklus): 70-143

IDIV "integer signed Division" előjeles osztás azonosan az előjel nélkülihez, de az előjelek figyelembevételével. (az előjelet az adat legmagasabb értékű bitje jelzi)

Módosított flagek: nem meghatározott.

Végrehajtási idő (órajel ciklus): 101-194

IMUL "integer signed multiplication" előjeles szorzás mint az előjel nélküli, csak az előjelek figyelembevételével.

Módosított flagek: of cf

Végrehajtási idő (órajel ciklus): 80-164

MOV "Move" adatmozgatás, a második operandus tartalmát az elsőbe írja. Részletesebb ismertetése megtalálható a könyvben.

Módosított flagek: nem befolyásolja a flagek állását.

Végrehajtási idő (órajel ciklus): 2-26

MOVS

MOVSB

MOVSW "Move String Byte/Word" DS:SI által címzett byte-ot illetve wordöt ES:DI címre írja majd Direction flag értékétől függően növeli (d=0) illetve csökkenti (d=1) di és si regisztereket byte-os műveletnél eggyel wordösnél pedig kettővel.

Módosított flagek: nem befolyásolja a flagek állását.

Végrehajtási idő (órajel ciklus): 9+17 (25)/rep

STOS

STOSB

STOSW "Store String Byte/Word" al illetve ax tartalmát ES:DI címre írja, majd d flagtól függően növeli illetve csökkenti di értékét.

Módosított flagek: nem befolyásolja a flagek állását.

Végrehajtási idő (órajel ciklus): 9+10 (14)/rep

LODS

LODSB

LODSW "LOaD String Byte/Word DS:SI által címzett byte vagy word tartalmát al illetve ah regiszterbe tölti és az si értékét a d flagtól függően változtatja.

Módosított flagek: nem befolyásolja a flagek állását.

Végrehajtási idő (órajel ciklus): 9+13 (17)/rep

LEA

"Load Effectiv Adres" a kijelölt cél operandusba tölti a forrásoperandus offsetcímét, működése megegyezik a MOV reg, offset cím utasítással.

Módosított flagek: nem befolyásolja a flagek állását.

Végrehajtási idő (órajel ciklus): 8-14

LDS "Load pointer using DS" egy négy byte-os memória operandus első és másodi byte-ja kerül a megadott regiszterbe, 3. és 4. byte-ja pedig a ds regiszterbe.

Módosított flagek: nem befolyásolja a flagek állását.

Végrehajtási idő (órajel ciklus): 30-36

LES "Load pointer usign ES" egy négy byte-os memória operandus első és másodi byte-ja kerül a megadott regiszterbe, 3. és 4. byte-ja pedig az es regiszterbe.

Módosított flagek: nem befolyásolja a flagek állását.

Végrehajtási idő (órajel ciklus): 30-36

XLAT "trans(X)LATe byte" a bx+al címen lévő adatot tölti al regiszterbe. Segítségével könnyedén olvashatunk ki egy maximum 256 byte hosszú táblázatból egy adatot mivel csak annyit kell tenni, hogy bx regiszterbe beállítjuk a táblázat kezdőcímét, majd al-be a kiolvasandó elem sorszámát és az utasítás végrehajtása után a kiválasztott elem értéke al-ben lesz.

Módosított flagek: nem befolyásolja a flagek állását.

Végrehajtási idő (órajel ciklus): 11

XCHG "eXCHanGe" megcseréli a forrás és a céloperandus értékét.

Módosított flagek: nem befolyásolja a flagek állását.

Végrehajtási idő (órajel ciklus): 3-37

- LAHF** "Load AH from Flags" a flag regisztert az ah regiszterbe másolja.
- Módosított flagek: nem befolyásolja a flagek állását.
- Végrehajtási idő (órajel ciklus): 4
- SAHF** "Store AH in Flags" ah regiszter értékét a flag regiszterbe másolja.
- Módosított flagek: **zf sf af pf cf**
- Végrehajtási idő (órajel ciklus): 4
- CBW** "Convert BYte into Word" az ah regisztert feltölti az al regiszter felső bitjével. Ez a művelet főként előjeles műveletek végzésénél használatos.
- Módosított flagek: nem befolyásolja a flagek állását.
- Végrehajtási idő (órajel ciklus): 2
- CWD** "Convert Word to Doubleword" hasonlóan az előzőhöz, de itt egy wordöt alakít dupla worddé DX:AX-be.
- Módosított flagek: nem befolyásolja a flagek állását.
- Végrehajtási idő (órajel ciklus): 5
- CMP** "CoMPare integers" két operandus összehasonlítása egy látszólagosan elvégzett kivonási eljárás segítségével és a flag regiszter egyes bitjeit ennek megfelelően állítja be.
- Módosított flagek: **of sf zf af pf cf**
- Végrehajtási idő (órajel ciklus): 3-14

CMPS

CMPSB

CMPSW "CoMPare String Byte/Word" DS:SI és ES:DI által mutatott memória rekesz tartalmának összehasonlítása majd si és di regisztereket d flagtól függően növeli vagy csökkenti.

Módosított flagek: cf af pf of sf zf

Végrehajtási idő (órajel ciklus): 9+22 (30)/rep

SCAS

SCASB

SCASW "SCAn String Byte/Word" al illetve ax tartalmát összehasonlítja ES:DI által mutatott memóriarekesz tartalmával és a jelzőbiteket ennek megfelelően állítja be. Az utasítás segítségével könnyen megtalálható egy adat előfordulási helye.

Módosított flagek: of zf cf pf sf af

Végrehajtási idő (órajel ciklus): 9+15 (19)/rep

TEST

"TEST" hatása azonos az AND művelettel, de a regiszterek értékét nem változtatja meg, csak a flaget.

Módosított flagek: of cf sf zf pf

Végrehajtási idő (órajel ciklus): 3-25

CLC

"CLear Carry flag" az átviteli jelzőbitet nullára állítja.

Módosított flagek: cf

Végrehajtási idő (órajel ciklus): 2

STC

"SeT Carry flag" az átviteli jelzőbitet 1-be állítja.

Módosított flagek: cf

Végrehajtási idő (órajel ciklus): 2

- CMC** "CoMplement Carry" komplementálja a carry flaget.
Módosított flagek: **cf**
Végrehajtási idő (órajel ciklus): 2
- CLD** "CLear Direction flag" az irányjelző flaget nullára állítja.
Módosított flagek: **df**
Végrehajtási idő (órajel ciklus): 2
- STD** "SeT Direction flag" az irányjelző flag 1-be állítása.
Módosított flagek: **df**
Végrehajtási idő (órajel ciklus): 2
- CLI** "CLear Interrupt flag" a hardware megszakításkérések fogadásának tiltása (kivéétel az NMI).
Módosított flagek: **if**
Végrehajtási idő (órajel ciklus): 2
- STI** "SeT Interrupt flag" hardware megszakítások engedélyezése.
Módosított flagek: **if**
Végrehajtási idő (órajel ciklus): 2
- IN** "INput from i/o port" adott porton lévő adat beolvasása. Az olvasás történhet **al** vagy **ax** regiszterbe, továbbá **8** illetve 16 bites portcímről. Utóbbi esetén a címet **dx** regiszterbe kell tölteni. Egyébként számmal kiírható.
Módosított flagek: nem befolyásolja a flagek állását.
Végrehajtási idő (órajel ciklus): 8-14

INS

INSB

INSW

"INput String Byte/Word from i/o port" byte vagy word beolvasása dx által mutatott portról ES:DI címre, majd növeli vagy csökkenti di értékét d flagtól függően.

Módosított flagek: nem befolyásolja a flagek állását.

Végrehajtási idő (órajel ciklus): 9+ 14(17)/rep

OUT

"OUTput to i/o port" hasonlóan mint az IN utasításnál, csak itt nem adat beolvasása történik a portról, hanem egy al vagy ax állata tartalmazott adat küldése a kijelölt portra.

Módosított flagek: nem befolyásolja a flagek állását.

Végrehajtási idő (órajel ciklus): 8-14

OUTS

OUTSB

OUTSW

"OUTput String Byte-Word to i/o port" DS:SI által címzett adat küldése a kijelölt porton keresztül, majd si regiszter növelése vagy csökkentése.

Módosított flagek: nem befolyásolja a flagek állását.

Végrehajtási idő (órajel ciklus): 9+14 (17)/rep

PUSH

"PUSH" csökkenti sp értékét kettővel és a kiválasztott 16 bites regisztert a verem tetejére írja (sp mindig az utoljára beírt adatra mutat).

Módosított flagek: nem befolyásolja a flagek állását.

Végrehajtási idő (órajel ciklus): 15-36

PUSHF

"PUSH Flag" a flag regiszter tartalmát írja a stackre.

Módosított flagek: nem befolyásolja a flagek állását.

Végrehajtási idő (órajel ciklus): 14

Az Assembly programozás alapjai

- POP** "POP" kiolvassa az **sp** által mutatott értéket és a kiválasztott regiszterbe tölti, majd növeli **sp** értékét kettővel.
- Módosított flagek: nem befolyásolja a flagek állását.
- Végrehajtási idő (órajel ciklus): 12-37
- POPF** "POP Flag" a veremről leemelt értéket a **flag** regiszterbe tölti majd növeli **sp** értékét kettővel.
- Módosított flagek: **of df if tf sf zf af pf cf**
- JMP** "JuMP" feltétel nélküli vezérlésátadás a megadott címre (közele -near illetve távoli -far).
- Módosított flagek: nem befolyásolja a flagek állását.
- Végrehajtási idő (órajel ciklus): 11-24
- JA** "Jump if Above" ugrás, ha előjel nélkül nagyobb.
- Módosított flagek: nem befolyásolja a flagek állását.
- Végrehajtási idő (órajel ciklus): 4-16
- JNA** "Jump if Not Above" ugrás, ha előjel nélkül kisebb vagy egyenlő.
- Módosított flagek: nem befolyásolja a flagek állását.
- Végrehajtási idő (órajel ciklus): 4-16
- JNC** "Jump if No Carry" ugrik, ha előjel nélkül nagyobb vagy egyenlő.
- Módosított flagek: nem befolyásolja a flagek állását.
- Végrehajtási idő (órajel ciklus): 4-16

- JC "Jump if Carry" Ugrik ha előjel nélkül kisebb.
Módosított flagek: nem befolyásolja a flagek állását.
Végrehajtási idő (órajel ciklus): 4-16
- JZ "Jump if Zero" ugrás ha egyenlő.
Módosított flagek: nem befolyásolja a flagek állását.
Végrehajtási idő (órajel ciklus): 4-16
- JNZ "Jump if Not Zero" ugrik ha nem egyenlő.
Módosított flagek: nem befolyásolja a flagek állását.
Végrehajtási idő (órajel ciklus): 4-16
- JG "Jump if Greater" ugrik, ha előjelesen nagyobb.
Módosított flagek: nem befolyásolja a flagek állását.
Végrehajtási idő (órajel ciklus): 4-16
- JNG "Jump if Not Greater" ugrik, ha előjelesen kisebb vagy egyenlő.
Módosított flagek: nem befolyásolja a flagek állását.
Végrehajtási idő (órajel ciklus): 4-16
- JGE "Jump if Greater or Equal" ugrik, ha előjelesen nagyobb vagy egyenlő.
Módosított flagek: nem befolyásolja a flagek állását.
Végrehajtási idő (órajel ciklus): 4-16

Az Assembly programozás alapjai

- JNGE** "Jump if Not Greater or Equal" ugrik, ha előjelesen kisebb.
- Módosított flagek: nem befolyásolja a flagek állását.
- Végrehajtási idő (órajel ciklus): 4-16
- LOOP** "LOOP" csökkenti **cx** értékét és ha még nem érte el a nullát, ugrik a megadott címre. Segítségével **cx** hosszúságú ciklust szervezhetünk.
- Módosított flagek: nem befolyásolja a flagek állását.
- Végrehajtási idő (órajel ciklus): 5-17
- LOOPZ** "LOOP while Zero" a ciklus addig tart, míg **cx** el nem éri a nullát, vagy a **z** bit nullára nem vált. (**z=0** esetén kilép a ciklusból)
- Módosított flagek: nem befolyásolja a flagek állását.
- Végrehajtási idő (órajel ciklus): 6-18
- LOOPNZ** "LOOP while Not Zero" az előző utasítás ellentéte, a ciklus addig tart, míg **z=0** és **cx** nem 0 egyébként vége.
- Módosított flagek: nem befolyásolja a flagek állását.
- Végrehajtási idő (órajel ciklus): 6-18
- REP** "REPeat string prefix" a karakterlánc műveletek ismétlése amíg **cx** nem nulla. Továbbá csökkenti **cx** értékét.
- Módosított flagek: nem befolyásolja a flagek állását.
- Végrehajtási idő (órajel ciklus): 2

REPZ

REPZ "REPeat string prefix while Zero/Not Zero" a kiegészítés hatása azonos a LOOP utasításnál említettekkel.

Módosított flagek: nem befolyásolja a flagek állását.

Végrehajtási idő (órajel ciklus): 2

CALL

"CALL" szubrutin hívása úgy, hogy ip jelenlegi értékét a verembe menti (rövid ugrásnál) illetve távoli ugrásnál a cs szegmensregiszter értékét is menti.

Módosított flagek: nem befolyásolja a flagek állását.

Végrehajtási idő (órajel ciklus): 16-57

INT

"softwarte INTerrupt" software megszakítási eljárás indítása.

Módosított flagek: nem befolyásolja a flagek állását.

Végrehajtási idő (órajel ciklus): 51-52

INTO

"INTerrupt if Overflow" megszakítási rutin indítása ha az overflow flag értéke 1.

Módosított flagek: nem befolyásolja a flagek állását.

Végrehajtási idő (órajel ciklus): 53-54

RET

RETF

"near/far RETurn from subroutine" kiolvassa a veremből a rutin előzőleg elmentett visszatérési címét és beírják a megfelelő regiszterekbe, ezzel átadják a vezérlést a rutint hívó utasítás utáni sorra.

Módosított flagek: nem befolyásolja a flagek állását.

Végrehajtási idő (órajel ciklus): 20-32

Az Assembly programozás alapjai

- IRET** "Interrupt RETurn" visszatérés megszakítási rutinból.
Módosított flagek: nem befolyásolja a flagek állását.
Végrehajtási idő (órajel ciklus): 32
- NOT** "Not" logikai nem művelet, az operandus bitjeit ellenkező értékűre állítja.
Módosított flagek: nem befolyásolja a flagek állását.
Végrehajtási idő (órajel ciklus): 3-24
- NEG** "Negate" az operandus kettes komplementjét képezi azaz invertálja az op. bitjeit és hozzáad egyet.
Módosított flagek: of sf zf af cf pf
Végrehajtási idő (órajel ciklus): 3-24
- AND** "AND" logikai és művelet, eredményei:
- | | |
|-----|---|
| 0 0 | 0 |
| 0 1 | 0 |
| 1 0 | 0 |
| 1 1 | 1 |
- Módosított flagek: of sf zf pf cf
Végrehajtási idő (órajel ciklus): 3-25
- OR** "OR" logikai vagy művelet, eredményei:
- | | |
|-----|---|
| 0 0 | 0 |
| 0 1 | 1 |
| 1 0 | 1 |
| 1 1 | 1 |
- Módosított flagek: of sf zf pf cf
Végrehajtási idő (órajel ciklus): 3-27

XOR	"eXclusive OR" logikai kizáró vagy:	0 0 0
		0 1 1
		1 0 1
		1 1 0

Módosított flagek: **of sf zf pf cf**

Végrehajtási idő (órajel ciklus): 3-37

ROL "ROtate Left" operandus forgatása balra, a kiforduló bit az adat másik szélére és a c flagbe íródik.

Módosított flagek: **cf of**

Végrehajtási idő (órajel ciklus): 2-36

ROR "ROtate Right" az előző művelet ellenkezője, itt a forgatás jobbra történik.

Módosított flagek: **cf of**

Végrehajtási idő (órajel ciklus): 2-36

RCL

RCR "Rotate through Carry Left/Right" az adat forgatása jobbra vagy balra a carry biten keresztül. A kicsorduló bit c-be kerül és az operandus másik szélére innen fordul be.

Módosított flagek: **of cf**

Végrehajtási idő (órajel ciklus): 2-36

SHL

SHR "SHift Left/Right" az adat balra illetve jobbra tolása úgy, hogy a másik oldalról beforduló bit értéke 0. Természetesen a kicsorduló bit itt is a carry-be kerül.

Módosított flagek: **cf zf of pf sf**

Végrehajtási idő (órajel ciklus): 2-36

Az Assembly programozás alapjai

SAL	"Shift Arithmetic Left" működése megegyezik az SHL utasításéval. Módosított flagek: cf zf of pf sf Végrehajtási idő (órajel ciklus): 2-36
SAR	"Shift Arithmetical Right" az adatot jobbra tolja és bal oldalra a signum bit értéke íródik. Módosított flagek: cf zf of pf sf Végrehajtási idő (órajel ciklus): 2-36
HLT	"HaLt" leállítja a processzor működését, innen csak egy reset vagy egy hardware interrupt billentheti ki. Az utóbbi esetben a program a megszakítás lekezelése után a HLT utasítást követő sortól folytatódik. Módosított flagek: nem befolyásolja a flagek állását. Végrehajtási idő (órajel ciklus): 2
LOCK	"LOCK" egy tetszőleges utasítás előtt használva a művelet végrehajtásának idejére lezárja a processzor buszait. Ez főként többprocesszoros rendszereknél használatos, amikor nem kívánatos, hogy egy művelet végrehajtásának ideje alatt a processzor más (külső) utasítást is fogadjon. Módosított flagek: nem befolyásolja a flagek állását. Végrehajtási idő (órajel ciklus): 2
WAIT	"WAIT" a processzort várakozó állásba helyezi, míg a TEST kivezetésen alacsony szintet nem kap. Ezt akkor szokták alkalmazni, amikor a gépben eltérő sebességű részcsoportok találhatók illetve fontos szerepe van a matematikai processzorok kezelésénél. Módosított flagek: nem befolyásolja a flagek állását. Végrehajtási idő (órajel ciklus): 3

NOP "No OPeration" nincs műveletvégzés, főként ciklusok pontos időzítésére használják stb.

Módosított flagek: nem befolyásolja a flagek állását.

Végrehajtási idő (órajel ciklus): 3

GRAFIKA

Az IBM PC programok manapság egyre ritkábban használnak szöveges üzemmódot, helyette inkább kis grafikus ábrákkal, ikonokkal segítik a program használatát és teszik esztétikusabbá. Ezek programozásához ismerni kell a grafikus képernyő kezelését, és egyéb grafikus eljárásokat. Ebben a fejezetben erről lesz szó.

A PC-k grafikai képessége alapvetően hardware függő. Ezt meghatározza, hogy milyen grafikus kártya van a számítógépben no és persze, hogy milyen a monitor. Ezek a kártyák elég sok paraméterben különböznek egymástól, ezek közül a fontosabbak: felbontás, színmegjelenítő képesség, memória és egyéb hardware programozási paraméterek. A legegyszerűbb ami még használatos, azaz úgynevezett *HERCULES* típusú, melynek felbontása 720*348 képpont. A monitor színe általában borostyán vagy zöld szokott lenni. Tulajdonságaira külön nem térünk ki. A könyvben részletesen tárgyalt monitor kártya típusok a *CGA*, *EGA*, *VGA*, *SVGA*.

GRAFIKUS ÜZEMMÓDOK PROGRAMOZÁSA

A grafikus kártyák programozása több lépésből áll és több módszerrel is elvégezhető. Az egyszerűbb megoldás a ROM BIOS által biztosított lehetőségek kihasználása. A nehezebbik, de a gyorsabb és hatékonyabb, a direkt programozás portokon keresztül OUT utasítással.

Valamilyen alakzat vagy pont kirakása úgy történik, hogy a megfelelő memóriaterületekre a megfelelő adatot írjuk. Hogy hova és mit, az a képernyő-memória helyétől és felépítésétől is függ.

A CGA ÜZEMMÓD

A legelőször ismertetésre kerülő kártya a CGA melynek grafikus felbontása 320*200 képpont egyszerre maximum négy

színnel, amit 16 szín közül lehet kiválasztani. Illetve van a kártyának egy nagyobb felbontású üzemmódja is a 640*200 képpont de itt csak egy háttér illetve egy előtér színt használhatunk. A kártya sajátossága, hogy a képernyőt két egyforma részre bontja. Az első felében a páros sorok, a másodikban a páratlan sorok tartalmát tárolja.

A képernyő-memória szegmenscíme 0B800h. Az egyes lapok hossza könnyen kiszámolható, mivel 4 féle szín kiválasztásához legalább 2 bit szükséges, így 1 byte-on 4 képpontot tárolhatunk. Egy lapon 320*100 képpont van, aminek memóriaigénye 32000/4 Byte azaz 8000. Mivel a 16000 a számítástechnikában nem egy kerek szám, ezért a 16384-hez fennmaradó 384-et két egyenlő részre osztották, így a páros sorok 0B800:0000h címen, a páratlanok pedig a 0B800:2000h címen kezdődnek. Mindez 640*200-as felbontásnál annyiban változik, hogy egy képpontot nem 2 hanem 1 bit határoz meg, de mivel vízszintesen kétszerezse a felbontás, ezért semmi más nem történik.

Sajnos a színkiosztás előre meghatározott, 2 fix paletta közül lehet választani.

Az első paletta színei:

0 - Háttérszín 1 - Zöld 2 - Piros 3 - Sárga/Barna

A második kiosztás:

0 - Háttérszín 1 - Türkiz 2 - Lila 3 - Fehér/Szürke

A 16 szín úgy alakul ki, hogy egy úgynevezett intenzitás beállítására is sor kerül (Így lesz a barnából sárga és a szürkéből fehér).

[Program 16]

<code>cga1</code>	<code>segment</code>	<code>:Szegmensdefiníció.</code>
	<code>assume cs:cga1,ds:cga1</code>	<code>:A cs illetve ds regiszterek</code>
		<code>beállítás.</code>

Grafika programozása Assembly nyelven

```
start:  mov  ax,cga1      ;Ds beállítása a szegmens elejére.
        mov  ds,ax

        mov  ax,0b800h ;A CGA képernyő memóri-
        mov  es,ax     ;ájának kezdőcímét es
                        ;szegmensregiszterbe tölti.

        mov  ax,4      ;A 10h BIOS rutin segítségével
        int  10h      ;beállítja a 320*200/4-es
                        ;grafikus üzemmódot.
                        ;ah=0 jelenti, hogy a képer-
                        ;nyő üzemmódot akarjuk
                        ;állítani és al=4 jelenti a
                        ;CGA módot.

        mov  ah,0bh    ;Szintén a 10h rutin
        mov  bh,1      ;segítségével, de most annak
        mov  bl,1      ;a 0bh funkciójával beállítjuk,
        int  10h      ;hogy az 1-es palettakiosztást
                        ;használja. bh=1 jelenti,
                        ;hogy palettaállítás
                        ;következik, bl-be pedig a
                        ;paletta sorszámát kell írni
                        ;(0 vagy 1).

        mov  al,01010101b ;A képernyő bal felső sarkába
        mov  es:[0],al    ;írunk egymás mellé nyolc 1-es ,
        mov  al,10101010b ;2-es és 3-as szinkódot,
        mov  es:[1],al    ; így mindhárom palettaszín
        mov  al,11111111b ; és a háttérszín is látszani fog.
        mov  es:[2],al

        xor  ax,ax      ;Billentyűvárás
        int  16

        mov  ah,0bh    ; A 0. paletta beállítása.
        mov  bh,1
        mov  bl,0
        int  10h
        xor  ax,ax      ;Újabb billentyű várás.
        int  16

        mov  ah,0bh    ;Ha bh-ba 0 van és így
        mov  bh,0      ;hijuk meg a 0bh funkciót,
        mov  bl,1      ;akkor a bl regiszterben a
        int  10h      ;háttér színét lehet megadni.
                        ;Jelen esetben 1 azaz sötétkék.
```

	xor	ax,ax	;Billentyűvárás.
	int	16	
	mov	ah,0bh	;A háttérszín 4. bitje az előtér
	mov	bh,0	;szín intenzitása. Itt ezt
	mov	bl,1+16	;állítja aktív állapotba (így
	int	10h	;lesz a barnából sárga és a
			;szürkéből fehér).
	xor	ax,ax	;Billentyűvárás.
	int	16	
	mov	ax,3	;80*25 karakteres szöveges
	int	10h	;mód visszaállítása.
	mov	ax,4c00h	;A 4ch DOS funkció hívása
	int	21h	;(visszatérés a DOS-hoz).
cga1	ends		;Szegmens vége.
	end	start	;Program vége.

A program indításkor alapértékeknek megfelelő háttérszín mellett kirak a bal felső sarokba egymás mellé egy kék, egy lila egy fehér vonalat, és vár egy billentyű lenyomására. Ezután átállítja a palettát, így a színek zöldre, pirosra, sárgára változnak. Újabb billentyűnyomás után a színek sötétre váltanak, és a háttér kék színű lesz. Utolsó lépésben ismét nagyobb intenzitással jelennek meg a vonalak, majd visszatér a DOS-hoz.

Egy grafikus pont megjelenítéséhez az első lépés, hogy meghatározzuk a helyét és színét. Erre láthatunk példát a következő programban.

[Program 17]

cga2	segment		;Szegmensdefiníció.
	assume	cs:cga2,ds:cga2	;A cs, ds regiszterek
			;beállítása.
start:	mov	ax,cga2	;A ds regiszter beállítása
	mov	ds,ax	;a szegmens elejére.

Grafika programozása Assembly nyelven

```
mov ax,0b800h      ;A videomemória kezdőcímét
mov es,ax          ;az es szegmensregiszterbe
                  ;tölti.

xor di,di          ;Indexregisztert nullázza.

mov ax,4           ;A 320*200-as grafikus
int 10h           ;felbontás beállítása.

mov ah,0bh        ;A paletta címke alatt tárolt
mov bh,1          ;számú palettát választja ki.
mov bl,byte ptr [PALETTA]
int 10h

mov ah,0bh        ;A háttér címkénél lévő szint
mov bh,0          ;állítja be papírszínnek.
mov bl,byte ptr [HATTER]
mov dl,byte ptr [FENYESSEG]
mov cl,4          ;Az intenzitás a háttérszín 4. bitje
shl dl,cl         ;ezért azt 4 bittel balra
add bl,dl         ;kell léptetni, hogy utána
int 10h          ;hozzá lehessen adni a színkódhoz.

mov bl,byte ptr [KOORD_Y] ;A bl regiszterbe tölti a függőleges
test bl,1         ;koordinátát és megnézi, hogy
jz .i_cga2        ;páros illetve páratlan sorról van-e
mov di,2000h     ;szó és ennek megfelelően

.i_cga2:
shr bl,1         ;beállítja a megfelelő kezdőcímet.
mov ax,80        ;Innentől már csak fél képpel
mul bl           ;kell számolni, ezért osztja 2-vel.
add di,ax        ;és mivel 320 pont 80 byte-on fér
                ;el, megszorozzuk 80-nal, és ezt
                ;adjuk a di-hez.

mov bx,word ptr [KOORD_X] ;A vízszintes koordináta előállítá-
mov cl,bl         ;sánál figyelembe kell venni,
and cl,00000011b ;hogy egy byte 4 pont adatait
shl cl,1         ;hordozza. Ezért a címkiszámítás-
shr bx,1         ;nál ez az alsó 2 bit nem kell.
shr bx,1         ;Annak a 2 bitnek a pontkirakásnál
add di,bx        ;van szerepe, ugyanis az határozza
                ;meg a pont helyét a byte-ban
                ;cl forgatására azért van szükség,
                ;mert egy képpont 2 bit helyet
                ;foglal.
```

```

sorszámú      mov  al,byte ptr [SZIN]      ;Mivel egy byte-ban a bal szélső
ror           ror  al,1          ;2 bit, tartalmazza a kisebb

ror           ror  al,1          ;pontot, ezért a színbyte alsó 2
ror           ror  al,cl        ;bitjét felülre forgatjuk, ezután
mov          es:[di],al        ;cl-ben kiszámolt értékkel jobbra
;toljuk és kiírjuk a már kiszámolt
;memória címre.

xor          ax,ax            ;Billentyűvárás.
int         16h

mov          ax,3             ;80*25 soros karakteres mód
int         10h              ;visszaállítása.

mov          ax,4c00h        ;Visszatérés a DOS-hoz.
int         21h

KOORD_X: dw  160             ;Vízszintes koordináta (0-319)
KOORD_Y: db  100            ;Függőleges koord. (0-199)
SZIN:      db  3             ;Színkód (0-3)
PALETTA:  db  0             ;Palettakód (0,1)
HATTER:   db  0             ;Háttérszín (0-7)
FENYESSEG:db  1             ;Intenzitás (0,1)

cga2        ends           ;Szegmensvég
end         start          ;Programvég

```

Bármilyen képpel, ábrával is dolgozunk, az mindig különböző színű pontokból tevődik össze. Ahhoz, hogy a kívánt képet lássuk, minden pontnak a helyén kell lennie. Egy egyedülálló pont kirakására mutat példát a 17. program. Itt be lehet állítani a pont koordinátáit és a színét.

A képernyőn való tájékozódásnál segítséget nyújt egy képzeletbeli koordináta rendszer, melynek [0;0] pontja a bal felső sarok. A memóriában egy sor tárolása lineárisan történik, tehát képernyőn egymás mellett lévő pontok a memóriában egymás után helyezkednek el. A páros és páratlan sorokat a CGA üzemmódban külön tárolják. A képernyő-memória kezdőcímén helyezkedik el a 0. sor 0. byte-ja. A 0. sor jobb szélső pontja után a 2. sor első pontja található és így tovább. Az 1. sor a kezdőcím+2000h címtől kezdődik és a sor utolsó pontját a 3. sor el-

ső pontja követi. A képpont címének kiszámításakor az első lépés meghatározni, hogy a pont páros vagy páratlan sorban van-e. Ezt el lehet végezni egy 2-vel való osztással (jobbra léptetéssel) és ha a c flag 1, akkor a sor páratlan, a kezdőcím **0B800:2000h**. A továbbiakban a képernyő felével kell csak számolni. Ezt meg lehet oldani úgy, hogy a függőleges koordinátát osztjuk 2-vel, vagy a sor hosszát vesszük 160 pontra. Az eredmény ugyan az, de az első módszer annyiban egyszerűbb, hogy a függőleges osztás már egyszer megtörtént amikor meghatároztuk, hogy a sor páros-e vagy páratlan és ennek az eredményét szorozzuk meg 80-nal, mivel ennyi byte szükséges egy képsor tárolásához. Ehhez még hozzá kell adni a vízszintes koordinátának megfelelő értéket. A pont vízszintes pozícióját osztjuk négygel és ezt adjuk az előbb kiszámolt értékhez. Az osztásnál kapott maradék határozza meg, hogy a byte-on belül hányadik pontot kell kitenni.

Egy grafika viszont általában nem egy pontból áll, hanem sokból. Ennek kirakása már egy kicsit lassú lenne, ha minden pontnak külön meg kéne adni a koordinátáját. Helyette azt szokták csinálni, hogy egy téglalap alakú befoglaló formába helyezik az alakzatot, és annak minden pontját eltárolják. így egy kicsit több pont kerül ugyan tárolásra, de nem kell megadni a pontok koordinátáját, csak az alakzat szélességét és magasságát. Ezekből a paraméterekből a kirakó program automatikusan kiszámolja a pontok helyét. Ilyen elven vannak tárolva az úgynevezett ikonok, sprite-ok, brush-ok. A következő példában egy klasszikus esetről lesz szó, mégpedig az egérkurzor kirakásáról és mozgatásáról. Első lépésben csak egy alakzatot fogunk kitenni a képernyőre. Ez az alapja az egérkezelésnek. Ha már van egy program, ami egy adott koordinátára ki tud rakni egy valamekkora alakzatot, akkor már csak egy lépés a mozgatás. Bonyolítja a dolgot, hogy ha kirakunk egy ábrát, az alatta lévő képet el kell tárolni, hogy a későbbiekben vissza tudjuk rakni, ha az egér már máshol van. A megoldás, hogy az egér kirakása előtt eltároljuk az alatta lévő képet, és amikor elmozdítjuk az ábrát, az alakzat törlése helyett ezt a képet rakjuk vissza.

[Program 18]

cga3	segment assume cs:cga3,ds:cga3	;Szegmensdefiníció. ;A cs, ds regiszterek ;beállítása.
start:	mov ax,cga3 mov ds,ax	;A ds regiszter beállítása ;a szegmens elejére.
	mov ax,0b800h mov es,ax	;A videomeória kezdőcíme ;az es szegmensregiszterbe ;tölti.
	xor di,di	;Indexregiszter törlése.
	mov ax,4 int 10h	;A 320*200-as grafikus ;felbontás beállítása.
	mov ah,0bh mov bh,1 mov bl,byte ptr [PALETTA] int 10h	;A paletta címke alatt tárolt ;számú palettát választja ki.
	mov ah,0bh mov bh,0 mov bl,byte ptr [HATTER] mov dl,byte ptr [FENYESSEG] mov cl,4 shl dl,cl add bl,dl int 10h	;A háttér címkénél lévő szint ;állítja be papírszínnek. ;Intenzitás beállítása.
	mov bl,byte ptr [KOORD_Y] test bl,1 jz .1_cga3 mov di,2000h	;Az ikon bal felső pontjának ;kiszámítása.
.1_cga3:	shr bl,1 mov ax,80 mul bl add di,ax mov bx,word ptr [KOORD_X] mov cl,bl and cl,11b shl cl,1 shr bx,1	

	shr	bx,1	
	add	di,bx	
	mov	di,10000000b	;Beállítomaszk.
	mov	dh,01111111b	;Törölmaszk.
	shr	di,cl	;A byte-on belüli bitcím
	shr	dh,cl	;beállítása.
	mov	si,offset [IKON]	;Az ikon táblázatcímét si -be
			;rakja.
	mov	cx,8	;Az ábra 8 képsor magas.
.2_cga3:	push	di dx cx	;A regiszterek elmentése a
			;későbbi felhasználás
			;céljából.
	mov	cx,16	;8*2 bit széles az ikon.
	mov	ax,[si]	;Ax regiszterbe tölt egy
			;sort az alakzattól.
.3_cga3:	push	cx	
	shl	ax,1	;Ax balra léptetésével
	jc	.4_cga3	;megvizsgáljuk, hogy kirakni
	and	es:[di],dh	;kell-e a pontot, vagy törölni.
	jmp	.5_cga3	;Az AND művelet hatására mivel
.4_cga3:	or	es:[di],dl	;a dh regiszter 1 bitje 0 ezért az a
bit			;törődök. Az OR műveletnél a dl
			;regiszter 1 bitje segítségével
			; 1-be állítjuk es:[di] címen
			; lévő byte egyik bitjét.
.5_cga3:	ror	dh,1	;A soronkövetkező bit figyelése.
	ror	dl,1	
	jnc	.6_cga3	;A ROR utasításnál a kicsúszo
	inc	di	;bit c flagbe kerül. Ha dl
			;forgatásakor c flag 1-be áll,
			;az azt jelenti, hogy dl és dh
			;körbefordult. Ilyenkor di -t
			;növelni kell eggyel.
.6_cga3:	pop	cx	
	loop	.3_cga3	
	pop	cx dx di	

IBM PC Gyakorlati Assembly

```

        cmp    di,2000h           ;A program megvizsgálja, hogy
        jnc    .7_cga3           ;páros avagy páratlan sor
        add    di,2000h         ;következik, és ettől függően
        jmp    .8_cga3           ;kivon vagy hozzáad di-hez

.7_cga3:  sub    di,2000h-80       ;2000h-t. Illetve kivonásnál
                                           ;80-nal kevesebbet, hogy a
                                           ;következő sorra ugorjon.

.8_cga3:  add    si,2             ;A következő ikonsor olvasása.
        loop   .2_cga3

        xor    ax,ax             ;Billentyűvárás.
        int    16h

        mov    ax,3              ;80*25 soros karakteres mód
        int    10h              ;visszaállítás.

        mov    ax,4c00h          ;Visszatérés a DOS-hoz.
        int    21h

KOORD_X: dw    160               ;Vízszintes koordináta (0-319)
KOORD_Y: db    100               ;Függőleges koord. (0-199)
PALETTA: db    0                 ;Palettakód (0,1)
HATTER:  db    0                 ;Háttérszín (0-7)
FENYESSEG:db  1                 ;Intenzitás (0,1)
IKON:    dw    10101010100000b   ;Ez egy 8*8 képpont méretű
        dw    1011111110000000b ;2 színű nyíl.
        dw    1011111000000000b ;A nyíl alakját azért nem
        dw    1011101110000000b ;lehet tökéletesen kivenni,
        dw    1010001011100000b ;mert 1 pontot 2 szám határoz
        dw    1000000010111000b  ;meg.
        dw    0000000000101110b
        dw    0000000000001000b

cga3     ends                    ;Szegmensvég
        end    start             ;Programvég

```

A program működését illetően a színek beállításáig teljesen megegyezik az előző példával. Az eltérés ott kezdődik, hogy a koordináta számításnál a kapott memória cím az ikon bal felső sarkának memóriacíme lesz. Innentől következik a 8*8 képpont méretű ikon kirakása (ez lehetne más méretű is). Mivel PC-n a memória egyes bitjeit egyenként nem lehet állítani, ezért erre egy trükköt alkalmazunk, mégpedig az OR és az AND művelet

tulajdonságait használjuk ki. Ha két egyenként nyolc bites számot OR kapcsolatba hozunk egymással, az eredményben azon bitek lesznek 1-es értékűek, ahol a két byte valamelyikébe 1 értéke volt az adott bitnek. Ezt a programban úgy használjuk fel, hogy az egyik byte a művelet során egy a képernyőn lévő adat, a másik pedig egy előre elkészített érték, melyben csak az egyik bit 1-es értékű, a többi 0. Ha két byte így találkozik, a következő eredményt kapjuk:

	00100110b	(Képernyő adat)
OR	00001000b	(Segéd Byte)

	00101110b	(Eredmény)

Ezen módszerrel egy byte bármely bitje 1-be állítható. A bitek törlése igen hasonló, csak ott az AND műveletet használjuk. Ugyanis itt az eredmény csak akkor lesz 1, ha mindkét bit 1:

	00100110b	(Képernyő adat)
AND	11111011b	(Segéd Byte)

	00100010b	(Eredmény)

Tehát most már egy byte tetszőleges bitjét 1-be illetve 0-ba tudjuk állítani. Ennek segítségével bitenként ki lehet rakni a kívánt ábrát. Mindössze annyit teszünk, hogy vesszük az ikon egy sorát, megvizsgáljuk az egyes bitek értékét és ezeket beállítjuk a képernyőn. Figyelni kell arra, hogy a megfelelő memória biteit állítsuk, mert az ikon nem biztos, hogy éppen egy byte első bitjétől kezdődik és biztos, hogy nem fér ki egy byte-on. Ezért a megfelelő időben a következő byte-ra kell ugrani. Ennek megoldása úgy történt, hogy a segéd byte forogtásakor ha az 1-es bit a byte szélén volt, és innen továbbforgattuk, akkor carry jelzőbit 1-es szintje jelzi ezt és egy *jc/jnc* utasítással ettől függően irányíthatjuk a program további futását. Amennyiben végzett a program az ikon egy sorának kirakásával, kiszámolja a következő sor kezdőcímét és megismétli a már említetteket összesen nyolcszor, mivel 8 képpont magas az ábra. Ezekután egy billentyűvárás és visszatérés a DOS-hoz.

Ha mozgatni is akarjuk az alakzatot, akkor némileg megváltozik a program felépítése. Ugyanis itt már nem szabad a 0 biteket kirakni, mivel ezzel törölnénk a képernyőn lévő képet. Addig, amíg csak a nyíl alatti rész törlődik, addig nincs semmi probléma, mivel azt előtte eltárolja a program, de egy ikon kirakása során nem csak a hasznos részt rakja ki, (jelen esetben a nyíl) hanem a befoglaló téglalap többi pontját is, ami nem tartalmaz értékes adatot. Ezáltal ha azokat a 0-kat is kirakná a program akkor a nyíl körüli téglalapot is megjelenítené. Ebből már látszik, hogy csakis OR művelettel szabad majd adatot kitenni, de így nem biztos, hogy a nyíl mindenhol a megfelelő színű lesz. Ez az OR művelet előbbi ismertetéséből könnyen belátható. A megoldás, hogy a nyíl alatti részt előbb ki kell törölni (minden bitjét 0-ba kell állítani) és ezután már nyugodtan ki lehet tenni az ikont OR művelettel. Ezt a törlést egy egérmaszkk segítségével végezzük el, aminek minden bitje 1 kivéve a nyíl helyét. Ha ezt AND művelettel tesszük ki, akkor az 1-es bitek alatti memóriatartalmak nem változnak, de a 0 alattiak kinullázódnak. Így a teljes művelet sor: képtartalom lementése ; egérmaszkk kirakása ; egér kirakása ; elmozdulás figyelése, ha van akkor az eredeti képtartalom kirakása ; koordináták megváltoztatása ; vissza az elejére. A programból kilépni valamelyik egérgomb megnyomásával lehet.

[Program 19]

```

cga4      segment                ;Szegmensdefiníció.
          assume cs:cga4,ds:cga4 ;A cs, ds regiszterek
                                               ;beállítása.

start:    mov  ax,cga4            ;A ds regiszter beállítása
          mov  ds,ax              ;a szegmens elejére.

          mov  ax,0b800h          ;A videomeória kezdőcímét
          mov  es,ax              ;az es szegmensregiszterbe
                                               ;tölti.

          call egerinstall        ;Az egér installálása
          jnz  .1_cga4            ;Ha nincs hiba, akkor
                                               ;elkezdődhet a program.

          mov  ax,3                ;80*25 karakteres kép
          int  10h                 ;beállítása.

```

Grafika programozása Assembly nyelven

```
mov di,52 ;0. sor 26. karaktere.
mov ax,7 ;fekete alapon fehér
;betűk lesznek.

mov si,offset HIBATEXT
.1_hiba: mov al,[si] ;Az egrerhiba szöveg kiiratasa.
cmp al,0 ;Ha a szövegbe 0 kód van,
jz .2_hiba ;az a szöveg végét jelenti.
mov es:[di],ax ;Ha nem, akkor kiirható a
add di,2 ;karakter.
inc si ;Következő karakter mindaddig,
jmp .1_hiba ;mig a 0 kodhoz nem ér a program.

.2_hiba: xor ax,ax ;billentyű várás, majd kilépés.
int 16h
jmp kilepes

.1_cga4: mov ax,4 ;A 320*200-as grafikus
int 10h ;felbontás beállítása.
mov ah,0bh ;A paletta címke alatt tárolt
mov bh,1 ;számú palettát választja ki.
mov bl,byte ptr [PALETTA]
int 10h

mov ah,0bh ;A háttér címkénél lévő szint
mov bh,0 ;állítja be papírszínnek.
mov bl,byte ptr [HATTER]
mov dl,byte ptr [FENYESSEG]
mov cl,4 ;Intenzitás beállítása
shl dl,cl
add bl,dl
int 10h

mov di,2025 ;A képernyő közepe tájára kirakunk
mov al,01010101b ;egy 100*100 képpont méretű 1-es
mov cx,50 ;színkódú négyzetet.

.2_cga4: push cx
mov cx,25
rep stosb ;es:[di] címre írja al-t és növeli
;di-t annyiszor, amennyi cx értéke.
add di,2000h-25 ;A következő páratlan sorra ugrik.
mov cx,25 ;A -25 azért kell, mert a STOSB
rep stosb ;utasítás megnövelte di értékét.
sub di,2000h+25-80 ;A következő páros sor.
pop cx
```

	loop .2_cga4	;Cx értékének megfelelően ;ismétli a ciklust.
.3_cga4:	call mentes call mask call eger	;Vezérlőrutinok hívása a ;megfelelő sorrendben.
.4_cga4:	call gombtest jnz kilepes call mozgás	
	jz .4_cga4 call kirakas mov ax,3 int 33h	;Ha nincs mozgás, vissza. ;Az egér koordinátáinak ;lekérdezése.
	cmp cx,624 jc .5_cga4	;Az egerkezelő az aktuális ;üzemmód legnagyobb felbontását ;(640*200) veszi alapul. És mivel ;az ikon 8 képpont széles ezért, ;hogya ne lóghasson ki a képből ;640-16-nál le kell tiltani.
	mov ax,4 mov cx,624 int 33h	;Ha ennél nagyobb lenne, akkor ;ide állítja vissza.
.5_cga4:	cmp dx,192 jc .6_cga4 mov ax,4 mov dx,192 int 33h	;Ugyan az, mint a vízszintes ;koordinátánál, de itt nem ;kell a pozíciót duplán ;számolni.
.6_cga4:	shr cx,1 mov word ptr [koord_x],cx mov byte ptr [koord_y],dl jmp .3_cga4	;A megváltozott értéket ;beírjuk a memória változóba.
kilepes:	mov ax,3 int 10h	;80*25 soros karakteres mód ;visszaállítás.
	mov ax,4c00h int 21h	;Visszatérés a DOS-hoz.
mentes	proc	

Grafika programozása Assembly nyelven

	xor	si,si	;Forrásindex regiszter ;nullázása
	mov	bl,byte ptr [koord_y]	
	test	bl,1	;Ha az y koordináta páratlan,
	jz	.1_kepmentes	;akkor si-hez hozzá kell
	add	si,2000h	;adni 2000h-t.
.1_mentes:	mov	ax,80	;Kiszámoljuk a függőleges
	shr	bl,1	;memóriacímet.
	mul	bl	
	add	si,ax	;És ezt si-hez adjuk.
	mov	bx,word ptr [koord_x]	
	mov	cl,bl	;A vízszintes pozíció alsó
	shr	bx,1	;2 bitje a byte-on belüli
	shr	bx,1	;cím.
	and	cl,3	
	shl	cl,1	
	xor	bh,bh	
	add	si,bx	;A vízszintes pozíciót is ;si-hez adjuk.
	mov	bl,cl	;Cl tartalmát, azaz a byte-on ;belüli pozíció értékét a bl ;regiszterben tároljuk, mivel cx ;regisztert több más helyen is ;használni fogjuk.
	mov	cx,8	;Az ikon 8 képsor magas
	mov	di,offset kephely	;A kephely memóriacímtől ;kezdődően lesz letarolva ;a kép
.2_mentes:	push	si cx	;cx és si verembe mentése.
	mov	dh,es:[si]	;Az es:[si] címen levő byte ;tartalmát, ami a képernyő ;egy része dh-ba teszi.
	mov	cl,bl	
	mov	dl,128	;DI regiszter 7. bit-jét ;1-be állítom. Ezt forgatva ;fogom tudni, hogy mikor kell ;a következő címen levő byte-ot ;beolvasni.

IBM PC Gyakorlati Assembly

	shl	dh,cl	:A beolvasott memóriatartalom ;és a segédbyte forgatása úgy, ;hog az értékes bitek kerülnek ;a byte szélére.
	ror	dl,cl	
	mov	cx,16	:A tárolandó rész 16 bit széles.
.3_mentes:	shl	dh,1	:A kiolvasott byte értékét balra ;toljuk eggyel, így a kicsúszó ;bit a carry flagbe kerül.
	rcl	ax,1	:Az rcl hatására az ax regisztert ;balra forgatja, es a 0. bit ;helyére a c flag értékét ;forgatja be.
	ror	dl,1	:A segédbyte forgatása. Ha
	jnc	.4_mentes	;a kiforduló bit 0 akkor a ;következő bit átírása ;következik, ha 1 akkor a ;byte végére ért az olvasás ;és egy új byte-ot kell ;beolvasni.
.4_mentes:	loop	.3_kepmentes	
	mov	[di],ax	:Ha ax mind a 16 bit-jébe ;betöltődött a megfelelő ;adat, akkor azt a memóriába ;írjuk (képhely).
	pop	cx si	
	cmp	si,2000h	:A következő sor címének
	jc	.5_mentes	;kiszámítása (páros után
	sub	si,2000h-80	;páratlan és fordítva)
	jmp	.6_mentes	
.5_mentes:	add	si,2000h	
.6_mentes:	add	di,2	:Ax következő értékét 2 ;byte-al arrébb kell ;letárolni.
	loop	.2_mentes	:A következő képsor olvasása ;ha van meg. Ha nincs.
	ret		;akkor visszatérés a hívó ;programhoz.
mentes	endp		

kirakas	proc	
	xor di,di	
	mov bl,byte ptr [koord_y]	:Az ikon bal felső pontjának
	test bl,1	:kiszámítása.
	jz .1_kirakas	
	mov di,2000h	
.1_kirakas:	shr bl,1	
	mov ax,80	
	mul bl	
	add di,ax	
	mov bx,word ptr [koord_x]	
	mov cl,bl	
	and cl,11b	
	shl cl,1	
	shr bx,1	
	shr bx,1	
	add di,bx	
	mov dl,10000000b	:Beállítomaszk.
	mov dh,01111111b	:Törölmaszk.
	ror dl,cl	:A byte-on belüli bitcím
	ror dh,cl	:beállítása.
	mov si,offset kephely	
		:Az ikon táblázatcímét si-be
		:rakjuk.
	mov cx,8	:Az ábra 8 képsor magas.
.2_kirakas:	push di dx cx	:A regiszterek elmentése a
		:későbbi felhasználás
		:celjából.
	mov cx,16	:2*8 bit széles az ikon.
	mov ax,[si]	:Ax regiszterbe töltünk egy
		:sort az alakzatból.
.3_kirakas:	push cx	
	shl ax,1	:Ax balra léptetésével
	jc .4_kirakas	:megvizsgálja, hogy kirakni
	and es:[di],dh	:kell-e a pontot, vagy törölni.
	jmp .5_kirakas	:az AND művelet hatására mivel
.4_kirakas:	or es:[di],dl	:a dh regiszter 1 bitje 0 ezért az
		:a bit törlődik. Az OR műveletnél a
		:dl regiszter 1 bitje segítségével

```

;1-be állítja es:[di] címen lévő
;byte értékét.

.5_kirakas:  ror    dh,1
              ror    dl,1
              jnc   .6_kirakas
              inc   di
              ;A következő bit figyelése.
              ;A ROR utasításnál a kicsüsző
              ;bit c flagbe kerül. Ha dl
              ;forgatásakor c flag 1-be áll,
              ;az azt jelenti, hogy dl és dh
              ;körbefordult. Ilyenkor di-t
              ;növelni kell egyel.

.6_kirakas:  pop    cx
              loop  .3_kirakas
              pop   cx dx di
              cmp   di,2000h
              jnc   .7_kirakas
              add   di,2000h
              jmp   .8_kirakas
.7_kirakas:  sub    di,2000h-80
              ;A program megvizsgálja, hogy
              ;páros avagy paratlan sor
              ;következik, es ettől függően
              ;kivon vagy hozzáad di-hez
              ;2000h-t. Illetve kivonásnál
              ;80-nal kevesebbet, hogy a
              ;következő sora ugorjon

.8_kirakas:  add   si,2
              loop  .2_kirakas
              ;A következő ikonsor olvasása
              ;következik.

              ret

kirakas     endp

mask       proc

              mov   si,offset IKONMASK
              xor   di,di
              mov   bl,byte ptr [KOORD_Y]
              mov   ax,80
              test  bl,1
              jz    .1_mask
              add   di,2000h
              ;Az ikon bal felső sarkának
              ;kiszámítása

.1_mask:    shr   bl,1
              mul  bl
              add  di,ax
              mov  bx,word ptr [KOORD_X]
              mov  cl,bl
              shr  bx,1
              shr  bx,1
    
```



```

and    cl,3
shl    cl,1
add    di,bx
mov    bl,cl
mov    cx,8

.2_mask:  push  di cx
          mov   ax,[si]
          mov   dl,01111111b           ;Törlőmaszk
          mov   cl,bl
          ror   dl,cl
          mov   cx,16

.3_mask:  shl   ax,1                   ;Ahol az egérmaszkban
          jc   .4_mask                 ;0 van, ott törli a képet
          and  es:[di],dl              ;így helyet csinál a nyílnak.
                                          ;Ahol 1, ott az eredeti
                                          ;képernyőtartalmat hagyja.

.4_mask:  ror   dl,1
          jc   .5_mask
          inc  di

.5_mask:  loop  .3_mask
          pop  cx di
          cmp  di,2000h
          jnc  .6_mask
          add  di,2000h
          jmp  .7_mask

.6_mask:  sub   di,2000h-80

.7_mask:  add   si,2
          loop .2_mask
          ret

mask      endp

eger      proc

          mov  si,offset IKON
          xor  di,di
          mov  bl,byte ptr [KOORD_Y]
          mov  ax,80
          test bl,1

```

```

        jz     .1_eger
        add   di,2000h

.1_eger:  shr   bl,1
         mul   bl
         add   di,ax
         mov   bx,word ptr [KOORD_X]
         mov   cl,bl
         shr   bx,1
         shr   bx,1
         and   cl,3
         shl   cl,1
         add   di,bx
         mov   bl,cl
         mov   cx,8

.2_eger:  push  di cx
         mov   ax,[si]
         mov   dl,10000000b
         mov   cl,bl
         ror   dl,cl
         mov   cx,16

.3_eger:  shl   ax,1
         jnc  .4_eger
         or   es:[di],dl
         ;Hasonló módon mint az előbb
         ;csak most pont fordítva,
         ;ahol az ikonban l van azt OR
         ;művelettel kiteszi. Ha 0-val
         ;találkozik, akkor továbblép.

.4_eger:  ror   dl,1
         jnc  .5_eger
         inc  di

.5_eger:  loop  .3_eger
         pop  cx di
         cmp  di,2000h
         jnc  .6_eger
         add  di,2000h
         jmp  .7_eger

.6_eger:  sub   di,2000h-80

.7_eger:  add   si,2
         loop .2_eger

        ret
    
```

```

eger                endp

egerinstall        proc

                    xor    ax,ax                ;A 33h megszakítás kezeli az
                    ;egér összes funkcióját.
                    ;ebből a 0. az egér installálása.
                    int    33h                ;A rutinból való visszatéréskor
                    ;Ax=0 jelenti, hogy az egér
                    ;nem inicializálható.
                    ;255 jelenti a sikeres
                    ;installálást.

                    cmp    ax,0

                    ret

egerinstall        endp

mozgas             proc

                    mov    ax,3                ;Ha Ax=3 paraméterrel hívom
                    ;meg az egérkezelő rutint,
                    int    33h                ;visszatéréskor BX regiszter
                    ;0. bitje a bal, 1. bitje a jobb
                    ;2. bitje a középső gomb állapotát,
                    ;CX a vízszintes koordinátát
                    ;DX a függőleges koordinátát
                    ;tartalmazza.

                    shr    cx,1
                    cmp    cx,word ptr [KOORD_X] ;Ha eltérés van az eddigi
                    jnz    .1_mozgas            ;koordináta és a mostani között
                    cmp    dl,byte ptr [KOORD_Y] ;Zero flag értéke 0 lesz.

.1_mozgas:        ret

mozgas            endp

gombtest          proc

                    mov    ax,3
                    int    33h
                    test   bl,11b                ;Ha le van nyomva a jobb
                    ;vagy bal gomb, a teszt után

```

```

;Z flag 0 lesz. Ha mind a 0. és
;az 1. bit is 0, azaz nincs
;lenyomott billentyű, a Z flag
;1-be áll.

```

```

                ret

gombtest      endp

KOORD_X: dw    160                ;Vízszintes koordináta
KOORD_Y: db    100                ;Függőleges koord.
PALETTA: db    0                  ;Palettakód (0,1)
HATTER:  db    0                  ;Háttérszín (0-7)
FENYESSEG:db   1                  ;Intenzitás (0,1)
IKON:         dw    1010101010100000b
                dw    1011111110000000b
                dw    1011111000000000b
                dw    1011101110000000b
                dw    1010001011100000b
                dw    1000000010111000b
                dw    0000000000101110b
                dw    000000000001000b

IKONMASK:dw   0000000000011111b
                dw    0000000001111111b
                dw    0000000011111111b
                dw    0000000001111111b
                dw    0000110000001111b
                dw    0011111100000011b
                dw    111111111000000b
                dw    111111111110011b

KEPHELY: dw   0,0,0,0,0,0,0
HIBATEXT: db   "Az egér nem installálható !".0

cga4         ends                ;Szegmensvég
end          start                ;Programvég

```

A későbbiek folyamán lesz még szó hasonló eljárásokról, rutinokról. Saját program íráskor azonban nem tanácsolom ezek közvetlen felhasználását, mivel ezek a példák programozás technikai ötleteket nem nagyon tartalmaznak, ezáltal ennél gyorsabbat, rövidebbet is lehet írni. Ezek célja csupán, hogy ismer-

tessem az üzemmód sajátosságait és egy-két ötletet adjak a felhasználáshoz. A másik ok, amiért nem érdemes ezeket a rutintokat programba ágyazni, mert CGA monitort már elég kevés helyen használnak. Van még további szépséghibája is a dolognak, mint például az utolsó programnál az egeret mozgatva néha vibrál a nyíl. Ennek oka a monitorok működéséből adódik. Mégpedig, hogy a monitor a kép kirajzolását a bal felső sarokban kezdi, végigmegegy azon a soron és rajzolás nélkül a következő sor elejére tér vissza. A kép jobb alsó sarkát elérve szintén rajzolás nélkül ismét a bal felső sarokba tér vissza az elektronsugár. Probléma akkor adódik, ha kinn van a képernyőn egy ábra, az elektronsugár a kép közepén tart és ekkor mi megváltoztatjuk a kinnlevő képet. Az eredmény egy rövid időre a régi kép első fele és az új kép második fele. Sajnos erre nem minden programban figyelnek oda. A megoldás, hogy azalatt a bizonyos függőleges visszatérési idő alatt kell az ábrát kirakni. Egyébként ezt a visszatérést hívják Vertical Blanknek és másodpercenként monitortípustól függően 50 60 70-szer történik. Ezt az IBM PC gépeknél figyelemmel lehet kísérni, mivel van egy portcím, aminek egyik bitje a vízszintes, egy másik pedig a függőleges visszatérést jelzi. Erről majd nemsokára.

A fejezet elején szó esett arról, hogy a grafikus üzemmódokat közvetlenül is lehet programozni. Az eredmény ugyan az, mivel a BIOS is ezt teszi, csak előtte még sok olyan dolgot is elvégez, amire nem biztos, hogy az adott helyen szükség van. Ez a direkt programozás alapvetően két módon történik. Vagy egy megfelelő porton keresztül kiküldünk egy adatot, amit a képernyővezérlő értelmez, illetve két lépésből úgy, hogy egy portcímen egy belső regisztert címezünk meg és utána egy másik csatornán küldjük az adatot. Vannak ugyanis olyan portok, ahol többféle dolgot is lehet állítani és ezek közül ki kell választani azt amelyikre éppen szükség van. Ellenben ennek is van egy hátránya ami főképpen akkor jelentkezik, ha például egy CGA kártyára irt programot VGA kártyával üzemelő gépen futtatunk. Ugyanis itt azoknak a regisztereknek, amiket a CGA kártyánál használnánk lehet, hogy egészen más jelentésük van. Ezért ezt csak akkor érdemes használni, ha azonos kártya típusal lesz használva a program. Vannak természetesen kivételek, amik minden típusnál megegyeznek. Ilyen például a vertical blank.

A most következő kártya regiszter ismertetőben azok a címek, ahol külön cím és adatregiszter szerepel, ott a változtatás úgy történik, hogy a címregiszterre az elérendő regiszter sorszámát írjuk, és ezután az adatregiszter címén küldjük az adatot:

OUT címregiszter,regiszterszám ; OUT adatregiszter,adat

<u>Portcím</u>	<u>Típus</u>	<u>Funkció</u>
3d4h	Csak írható	MC6845 képernyővezérlő címregisztere
3d5h	Írható, olvasható	A képernyővezérlő adatregisztere
00h	Csak írható	Vízszintes felbontás karakteregységben
01h	Csak írható	Egy sor hossza karakteregységben
02h	Csak írható	Vízszintes kioltás kezdetének karakterpozíciója
03h	Csak írható	Vízszintes kioltás vége
04h	Csak írható	Karakter sorok száma
05h	Csak írható	Függőleges igazítás
06h	Csak írható	megjelenített sorok száma
07h	Csak írható	Függőleges kioltás kezdete
08h	Csak írható	Interlace mód
09h	Csak írható	Karaktert alkotó pontsorok száma
0ah	Csak írható	Kurzor kezdő pontsora
0bh	Csak írható	kurzor utolsó pontsora
0ch	Csak írható	A megjelenítés kezdeti memóriacímének magas helyiértékű byte-ja
0dh	Csak írható	A megjelenítés kezdeti memóriacímének alacsony helyiértékű byte-ja
0eh	Írható, olvasható	A kurzor pozíciójának magas helyiértékű byte-ja
0fh	Írható, olvasható	A kurzor pozíciójának alacsony helyiértékű byte-ja
10h	Csak olvasható	Fényceruza helyének magas helyiértékű byte-ja
11h	Csak olvasható	Fényceruza helyének alacsony helyiértékű byte-ja
3d8h	Csak írható	Üzem mód regiszter 7-6: Nem használt 5: Villogás engedélyezése 0 - Az attribútum 7. bitje a háttérszín intenzitását engedélyezi 1 - Az attribútum 7. bitje a

		villogást engedélyezi
		4: Nagyfelbontású grafikus mód beállítása
		3: Megjelenítés engedélyezése
		2: Fekete fehér mód beállítása
		0 - Fekete - fehér mód
		1 - Színes mód
		1: Grafikus mód beállítása
		0: 80*25 karakteres üzemmód beállítása
3d9h	Csak írható	Színkiválasztás regiszter
		7-6: Nem használt
		5: Paletta sorszám
		4: Színkészlet kiválasztása
		3: Intenzitás engedélyezése
		2-1-0: Karakteres módban a keret színe 320*200 módnál a háttér színe 640*200 módnál az előtér színét határozza meg (RGB)
3dah	Csak olvasható	Státuszregiszter
		7-4: Nem használt
		3: Független visszatérés alatt 1 az értéke
		2: Fényceruza kapcsolója zárva van
		1: Fényceruza engedélyezett
		0: 6845 nem használja a képernyőt (vízszintes vagy függőleges visszatérés)
3dbh	Csak írható	Fényceruza latch törlése
3dch	Csak írható	Fényceruza regisztereinek beállítása

EGA ÉS VGA ÜZEMMÓDOK PROGRAMOZÁSA

A továbbiakban ismertetésre kerülő monitor kártya típusok az EGA és a VGA. Az EGA kártyával részletesen foglalkozni nem fogok, mivel a működése kis eltéréssel azonos a VGA kártya működésével. Az eltéréseket természetesen megemlítem. Ezek közül a legalapvetőbb a grafikai felbontásban jelentkezik. Az EGA kártya maximális felbontása 640*350 képpont 16 színnel, amit egy 64 elemű palettából lehet kiválasztani. Eltérés

a CGA kártyához képest, hogy itt a kártyába már van beépítve külső RAM memória. Hogy mekkora az változó, általában 128K-tól 2M-ig szokott előfordulni. De az alap VGA üzemmódokhoz maximálisan 256K elég. Ennél több csak az SVGA üzemmódokhoz kell. Ezenkívül az EGA, VGA kártyáknak van saját karakterkészletük.

Az EGA kártya szolgáltatása a már említett üzemmódon és a CGA módjain kívül még további három lehetőség, mégpedig a 320*200/16, 640*200/16, 640*350/2 módok.

A videomemória kezdőcíme itt nem 0B800h hanem 0A000h. Egyszerre 64K RAMterülethez lehet hozzáférni, ha egy másik video lapra szeretnék írni, akkor azt lapozni kell.

A CGA kártya azon hátrányát, hogy a páros és páratlan sorok külön vannak tárolva, azt itt már kiküszöbölték, az adatok folytonosan vannak tárolva a memóriában. Ellenben a 16 színű üzemmódnál bejön egy újdonság. A 16 szín kiválasztásához 4 bitre van szükség. Ez a négy bit azonban nem egymás mellett van tárolva ahogyan a CGA kártyánál a 2 bit, hanem egymás mögött. Ez elsőre elég furcsának tűnhet a helyzet azonban az, hogy négy különálló lapra van bontva a képernyő. Ezeket hívják **plane**-eknek. Így egy képernyő négy egymás mögött elhelyezkedő plane-ből áll, melyekre egyenként vagy egyszerre lehet adatot írni, illetve róluk olvasni. Egy plane egy byte-ja nyolc egymás melletti képpont valamely bitjét tartalmazza.

A legegyszerűbb módszer egy pontot kirakására a ROM BIOS használata. Ugyanígy a háttérszín megváltoztatására stb. Ennek hátránya, hogy esetenként lassu lehet. Természetesen, amikor nem követelmény a nagy sebesség tökéletesen megfelelnek ezek a rutinok. Használatuk könnyű és egyszerű.

Az első példaprogramban itt is az üzemmód színpalettáját mutatom be. Először az alapbeállítású kiosztást majd az EGA 64 színlehetőségét.

[Program 20]

ega1	segment	:Szegmensdefiníció.
	assume cs:ega1,ds:ega1	:Cs és ds beállítása.
start:	mov ax,ega1	:Ds regiszter beállítása
	mov ds,ax	:a szegmens elejére.
	mov ax,10h	:640*350/16 <i>EGA</i> mód
	int 10h	:beallitasa.
	mov cx,16	:16 szinpaletta van.
.1_ega1:	push cx	
	mov cx,8	:Egy csik 8 pont magas
.2_ega1:	push cx	
	mov cx,140	:és 140 pont széles.
.3_ega1:	push cx	
	mov cx,word ptr [KOORD_X]	:A kirakandó pont X
		:koordinátája.
	mov dx,word ptr [KOORD_Y]	:Az Y koordináta.
	mov al,byte ptr [SZIN]	:A kirakandó pont szine.
	mov bh,byte ptr [LAP]	:A videolap sorszáma.
	mov ah,0ch	:Pontkirakás rutin
	int 10h	:hivása.
	inc word ptr [KOORD_X]	:Egy 140*8 pont méretű
	pop cx	:téglalapot rajzolunk a
	loop .3_ega1	:megadott palettaszinnel.
	mov ax,140	
	sub word ptr [KOORD_X],ax	
	inc word ptr [KOORD_Y]	
	pop cx	
	loop .2_ega1	
	pop cx	
	inc byte ptr [SZIN]	:Ezt az összes paletta-
	loop .1_ega1	:szinnel is megismételjük.
	xor ax,ax	:billentyűvárás
	int 16h	

```

.4_egal:  mov  cx,15                ;A 16 palettából a 0.
                                                ;a háttér, ezért azt nem
                                                ;állítjuk, csak a többi.

                mov  bl,1                ;Az első állítandó paletta.
                mov  bh,byte ptr [ELSO]   ;Az első színkód

.5_egal:  push cx bx
                mov  ax,1000h
                int  10h                ;Az adott paletta beállítása
                                                ;a megfelelő színűre

                pop  bx cx
                inc  bl                ;A paletta sorszám és a
                inc  bh                ;színkód növelése 15-ször
                loop .5_egal

.6_egal:  xor   ax,ax                ;billentyűfigyelés
                int  16h

                cmp  ah,48h            ;A felfelé nyíl volt?
                jnz  .7_egal

                mov  al,byte ptr [ELSO]   ;Ha igen, és az első
                cmp  al,48              ;palettaszín még nem
                jnc  .6_egal            ;érte el a 48-at, akkor
                inc  byte ptr [ELSO]     ;növeljük a kezdőszint
                jmp  .4_egal            ;és vissza a paletta-
                                                ;beállításhoz.

.7_egal:  cmp  ah,50h                ;Ha a lefelé nyíl volt,
                jnz  .8_egal            ;hasonlóan mint az előbb
                mov  al,byte ptr [ELSO]   ;megvizsgálja a kezdő
                cmp  al,0                ;értéket és ha nem 0,
                jz   .6_egal            ;csökkenti egyel, majd
                dec  byte ptr [ELSO]     ;vissza a paletta-beállításhoz
                jmp  .4_egal

.8_egal:  cmp  ah,1ch                ;Az ENTER lett lenyomva?
                jnz  .6_egal            ;Ha nem, újabb betű olvasása.

                mov  ax,3                ;Ha igen, 80*25 text képernyő
                int  10h

                mov  ax,4c00h           ;és visszatérés a DOS-hoz
                int  21h

```

Grafika programozása Assembly nyelven

```
KOORD_X: dw 250 ;A kirakandó pont X
KOORD_Y: dw 100 ;illetve Y koordinátája
SZIN: db 0 ;és a pont színe.
ELSO: db 0 ;Paletta-beállításnál az első
;csík színe.
LAP: db 0 ;A grafikus videolap sorszáma.

ega1 ends ;A szegmens vége.
end start ;A program vége.
```

Első lépésben kirak a program 16 darab 8*140 képpont méretű különböző színű csíkot egymás alá. Ez az EGA illetve VGA kártyák 16 színű felbontásainak alap paletta-beállítása. Ezek a színek sorrendben:

0	Fekete	8	Sötétszürke
1	Kék	9	Világoskék
2	Zöld	10	Világoszöld
3	Türkiz	11	Világos türkiz
4	Vörös	12	Piros2
5	Lila	13	Világos lila
6	Narancs	14	Sárga
7	Világos szürke	15	Fehér

Ellentétben a CGA móddal, itt lehetőség nyílik a palettaszínek megváltoztatására egyenként illetve az egészet egyszerre. A beállításnál az egyes bitek jelentése a következő:

7	-	Nem használt
6	-	Nem használt
5	-	Másodlagos vörös
4	-	Másodlagos zöld
3	-	Másodlagos kék
2	-	Vörös
1	-	Zöld
0	-	Kék

A példaprogramban az egyenkénti beállítás lett alkalmazva, mégpedig úgy, hogy egy **ENTER** lenyomása után görgetni lehet a színeket a föl-le nyilakkal. A paletta módosításánál a BIOS 10h megszakításának 10. programjának 0. funkcióját használjuk (ah=10; al=0), itt a belépésnél bl regiszterbe kell tenni a módosítandó paletta sorszámát bh-ba pedig a színkódot. Az egyszerre

re történő változtatásnál **al**-be kettőt kell írni, és **es:dx** címen le kell tárolni a 16 palettaszínt és egy keretszínt. A 0. paletta a háttér színét határozza meg. Ha a keret színét külön akarjuk állítani, ahhoz **al**-be egyet **bh**-ba pedig a keret színét kell írni.

Látható, hogy a program elég lassú és darabos. Valamennyire lehetne gyorsítani a dolgot, ha a palettaszíneket nem egyenként állítanánk, hanem egyszerre, de ez a sorok kirajzolását nem gyorsítja. A megoldás a közvetlen programozás. Ezt VGA felbontás mellett mutatom be egy kis változtatással. A változtatás oka, hogy míg az EGA felbontás mellett a 16 színt egy 64 elemű színskálából lehet kiválasztani, addig a VGA kártyáknál 64 féle piros zöld és kék színt lehet beállítani. Ezek variációit egy kicsit sokáig tartana végiggörgetni, mivel 262144 féle színt lehet beállítani. Ahhoz, hogy mégis be tudjam mutatni az összes színlehetőséget, a színeket egyenként lehet állítani. A piros összetevőt a **q a**, a zöldet a **w s**, a kéket pedig a **e d** billentyűkkel pozitív illetve negatív irányba. És mindez a háttérszín változásában jelentkezik. A kilépés az **ENTER** billentyűvel történik. A VGA 16 színű felbontásánál van egy kis furcsaság ami itt a programnál ugyan nem észrevehető, de a programozás során esetleg gondot okozhatna. Ugyanis itt egy 256 elemű palettában helyezkedik el az a 16 szín amit használhatunk, de ez nem az első 16, hanem a következő sorrendben helyezkedik el:

0	-	0	8	-	56
1	-	1	9	-	57
2	-	2	10	-	58
3	-	3	11	-	59
4	-	4	12	-	60
5	-	5	13	-	61
6	-	20	14	-	62
7	-	7	15	-	63

Tehát ha például a 12. palettakódot akarjuk állítani, akkor a 60. palettaszínt kell megváltoztatni. A továbbiakban minden program VGA képernyőre íródik, de kisebb változtatásokkal (színbeállítás, felbontás stb.) EGA üzemmódra is áttehetők.

A direkt programozásnál lehetőség nyílik a kártya összes lehetőségének a kihasználására. Itt a legegyszerűbb alkalmazást mutatom be, amikor a kiválasztott plane-ekre a megfelelő adatokat írjuk. Az első lépésben a VGA üzemmódot állítjuk be. Ezt még a BIOS segítségével. Ezután kiválasztjuk a legegyszerűbb írási módot: SET-RESET funkció tiltása; adatkirakási mód beállítása felülírásra; adatváltoztatás engedélyezése az összes bitre; a csíkok kirakásához kiválasztjuk minden csíknál a megfelelő bitsíkot. Ezzel előállítottuk ugyanazt a képet, amit az előző program a BIOS pontkirakó rutinja segítségével. Azt hiszem, a sebesség megváltozása magáért beszél. A palettaszínek változtatása direkt úton úgy történik, hogy a 3C8h portcímen kiküldjük a változtatni kívánt paletta sorszámát és ezután a 3C9h porton a piros, zöld, kék összetevőket. Ügyelni kell arra, hogy 3*6 bitnek kezeli a gép a színeket, ezért ha egy összetevőnél 63-nál nagyobb számot küldünk ki, annak is csak az első 6 bitjét fogja értelmezni. A program működése a továbbiakban hasonlít az előzőre, csak itt több billentyűt figyelünk.

[Program 21]

vga1	segment	;Szegmensdefiníció.
	assume cs:vga1,ds:vga1	;Cs és ds beállítása.
start:	mov ax,vga1	;Ds regiszter beállítása
	mov ds,ax	;a szegmens elejére.
	 mov ax,12h	 ;640*480/16 VGA mód
	int 10h	;beállítása.
	 mov dx,3ceh	 ;SET-RESET funkció tiltása.
	mov al,1	
	out dx,al	
	inc dx	
	mov al,0	
	out dx,al	
	 dec dx	 ;A 0. írási mód kiválasztása.
	mov al,5	
	out dx,al	
	inc dx	
	xor ax,ax	
	out dx,al	

```

dec dx ;Felülírási mód kiválasztása.
mov al,3
out dx,al
inc dx
mov al,0
out dx,al

dec dx ;Adatváltóztatás engedélyezése
mov al,8 ;az összes bitre.
out dx,al
inc dx
mov al,11111111b
out dx,al

mov ax,0a000h ;Videomemória kezdőcíme.
mov es,ax
mov di,8031

mov cx,15
.1_vga1: push cx
mov dx,3c4h ;A megfelelő plane-ek
mov al,2 ;kiválasztása.
out dx,al
inc dx
mov al,byte ptr [SZIN]
out dx,al
mov cx,8

.2_vga1: push cx ;A színcsíkok kirajzolása.
mov cx,17
mov al,255
rep stosb
sub di,17
pop cx
add di,80
loop .2_vga1

pop cx
inc byte ptr [SZIN] ;A következő palettaszín
loop .1_vga1 ;kiválasztása.

.3_vga1: mov dx,3c8h
mov al,0 ;A háttérszín állítása a
out dx,al ;megfelelő színűre.
inc dx
mov al,byte ptr [PIROS]

```

```

out    dx,al
mov    al,byte ptr [ZOLD]
out    dx,al
mov    al,byte ptr [KEK]
out    dx,al

.4_vga1:  xor    ax,ax                ;Billentyűfigyelés.
          int    16h

          cmp    al,"q"            ;A q billentyű figyelése.
          jnz    .5_vga1
          mov    al,63
          cmp    byte ptr [PIROS],al
          jnc    .4_vga1
          inc    byte ptr [PIROS]   ;Piros összetevő növelése
          jmp    .3_vga1           ;ha még nem 63.

.5_vga1:  cmp    al,"a"            ;Az a billentyű figyelése.
          jnz    .6_vga1
          mov    al,0
          cmp    byte ptr [PIROS],al ;Csökkentése ha nem 0.
          jz     .4_vga1
          dec    byte ptr [PIROS]
          jmp    .3_vga1

.6_vga1:  cmp    al,"w"            ;A w billentyű figyelése.
          jnz    .7_vga1
          mov    al,63
          cmp    byte ptr [ZOLD],al
          jnc    .4_vga1
          inc    byte ptr [ZOLD]   ;Zöld összetevő növelése
          jmp    .3_vga1           ;ha még nem 63.

.7_vga1:  cmp    al,"s"            ;A s billentyű figyelése.
          jnz    .8_vga1
          mov    al,0
          cmp    byte ptr [ZOLD],al ;Csökkentése ha nem 0.
          jz     .4_vga1
          dec    byte ptr [ZOLD]
          jmp    .3_vga1

.8_vga1:  cmp    al,"e"            ;A e billentyű figyelése.
          jnz    .9_vga1
          mov    al,63
          cmp    byte ptr [KEK],al

```

```

        jnc  .4_vga1
        inc  byte ptr [KEK]           ;Kék összetevő növelése
        jmp  .3_vga1                 ;ha még nem 63.

.9_vga1:  cmp  al,"d"                   ;A d billentyű figyelése.
        jnz  .10_vga1
        mov  al,0
        cmp  byte ptr [KEK],al       ;Csökkentése ha nem 0.

        jz   .4_vga1
        dec  byte ptr [KEK]
        jmp  .3_vga1

.10_vga1: cmp  al,13                   ;ENTER billentyű figyelése.
        jnz  .4_vga1
        mov  ax,3                     ;Ha igen, 80*25 text képernyő
        int  10h

        mov  ax,4c00h                 ;Visszatérés a DOS-hoz
        int  21h

SZIN:    db  1
PIROS:   db  0                       ;Az egyes színösszetevők
ZOLD:    db  0                       ;kiinduló értéke.
KEK:     db  0

vga1     ends                          ;A szegmens vége.
end      start                          ;A program vége.

```

Amint azt már említettem, ebben az üzemmódban többféle lehetőségünk van a képernyőre való íráskor. A legegyszerűbb, amit itt is használtunk, a következőképpen működik:

Műveleti kód reg.
 XXXX00000 (Felülírás)

	<i>Eredeti adat</i>	<i>Memória lapozó reg.</i>	<i>Új adat</i>
<i>Plane3</i>	XXXXXXXXXX	0	XXXXXXXXXX
<i>Plane2</i>	XXXXXXXXXX	1	XXXX1011
<i>Plane1</i>	XXXXXXXXXX	1	XXXX1011
<i>Plane0</i>	XXXXXXXXXX	0	XXXXXXXXXX
			00001111
			<i>Adatválttatás engedélyezés</i>

XXXX1011
Felhasználói
adat

EGA - VGA kártyák 0. írási módja a felhasználói adat használatával

A számítógépen az írási módot a grafikus vezérlő 5. regiszterének 0-1 bitje határozza meg. Használata:

OUT 3deh,5; OUT 3dfh,XXXXXX00b (a 0. módhoz).

A továbbiakban ki kell választani, hogy melyik plane-re kívánunk adatot írni, ettől függ a leendő pont(ok) színe. Ez hasonlóképpen működik, mint az előző, de ezt nem a grafikus vezérlő regisztereivel lehet állítani, hanem egy un. Sequencer segítségével aminek szintén van adat és címregisztere. Nekünk most a 2. belső regisztere van szükségünk, annak is az alsó 4 bitjére, ugyanis ezek értéke határozza meg, hogy egy plane-re lehet-e írni vagy sem. A 0 érték jelenti a tiltott állapotot.

OUT 3c4h,2; OUT 3c5h,XXXX0110b
(az 1. és a 2. plane lett engedélyezve.)

Továbbá, mivel az általunk beirt adatot kívánjuk, hogy megjelenjen a képernyőn, a SET-RESET funkciót le kell tiltani. Ezt a grafikus vezérlő 1. regiszterére irt 0-val lehet elérni:

OUT 3deh,1; OUT 3dfh,XXXX0000b
(ezzel az összes bitsíkon tiltottuk a SET-RESET funkciót)

Van még egy-két dolog amit be kell állítani, köztük az is, hogy milyen módon kívánjuk azt az adatot kitenni a képernyőre, lehetőség van ugyanis felülírási, AND, OR és XOR módok kiválasztására a vezérlő 3. regiszterén keresztül. Itt lehet még beállítani azt is, hogy ha az adatot nem egy az egybe akarjuk kirakni, hanem mondjuk el kívánjuk forgatni valahány bittel. A regiszter alsó 3 bitje a rotálás értékét a 4-5. bit a kirakási módot határozza meg. Mégpedig a következő módon:

00 - felülírás
01 - AND kapcsolat
10 - OR kapcsolat
11 - XOR kapcsolat

Ellenben ezen funkció miatt célszerű a MOV utasítást ke-
rülni a képernyőre való íráskor, helyette inkább használjuk az
OR műveletet a 16 színű üzemmódokban, mivel ha a műveleti
kód regisztert felülírásra állítottuk, akkor az biztos, hogy felül-
írást fog végezni, bármilyen művelettel tettük is ki az adott ada-
tot. Ha azonban például OR vagy más logikai műveletre állítot-
tuk és MOV utasítással teszünk ki adatot a képernyőre, annak
eredménye nehezen kiszámítható. Ez egy kicsit furcsa ugyan,
de ha nem a MOV utasítást használjuk, akkor mindig a műveleti
kód regiszterben meghatározott műveletet fogja végrehajtani.

Lehetőség nyílik továbbá arra is, hogy ne csak azt határoz-
zuk meg, hogy melyik plane-re lehessen írni, hanem azt is, hogy
egy byte-on belül mely bitek legyenek írhatók és melyek értékét
ne lehessen megváltoztatni. Ezt az adatváltoztatás engedélye-
zése nevű regiszter állítja, mely a vezérlő 8. regisztere.

OUT 3deh,8; OUT 3dfh,00001111b

Ezek után már nincs is más dolgunk, mint a megfelelő cím-
re a kívánt adatot kirakni. Ez a módszer lett alkalmazva a 6.
programnál is.

A 0. írási módban lehetőség van egy belső adat segítségével
írni a képernyőre. Ekkor az általunk beírt adaton kívül egy
előre beállított adatot is a képernyőre ír a következő módon:

Műveleti kód reg.
XXX00000 (*Felülírás*)

		<i>Set-Reset engedélyező</i>			
	<i>Eredeti adat</i>	<i>Memória lapozó reg.</i>	<i>Set-Reset adat</i>		<i>Új adat</i>
<i>Plane3</i>	XXXXXXXX	0	X	X	XXXXXXXX
<i>Plane2</i>	XXXXXXXX	1	1	1	XXXX1111
<i>Plane1</i>	XXXXXXXX	1	1	0	XXXX0000
<i>Plane0</i>	XXXXXXXX	1	0	X	XXXX1011
					00001111
					<i>Adatváltoztatás engedélyezés</i>

EGA - VGA kártyák 0. írási módja a SET-RESET használatával

Ekkor tulajdonképpen az történik, hogy ha műveletet végzünk a képernyő valamely részén, a művelet végrehajtódik és a SET-RESET engedélyező regiszter által kijelölt plane-ek közül azok ahol az írás engedélyezett, a SET-RESET adatregiszterében beállított értéket veszik fel, ahogyan ez az előző ábrán is jól látható.

A következő írási módot leginkább kiolvasáskor érdemes használni, vagy pedig ha azt szeretnénk, hogy a képernyő tartalmát véletlenül se lehessen megváltoztatni. Az 1-es írási mód lényege ugyanis az, hogy azt az adatot írja a képernyőre, ami előzőleg is ott volt, ezáltal semmi változás nem történik.

Működése:

	<i>Eredeti adat</i>	<i>Új adat</i>
<i>Plane3</i>	11001001	11001001
<i>Plane2</i>	00011001	00011001
<i>Plane1</i>	01011010	01011010
<i>Plane0</i>	10000011	10000011

EGA - VGA kártyák 1. írási módja

A következő írási mód hasonló a 0. mód SET-RESET módjához, de itt az adat, ami a bitsíkokba beíródik, az a felhasználó által beírt adat, nem pedig egy előre beállított. Működése során az engedélyezett plane-ek írható bitjeit a használt értékekkel tölti fel. Működése:

	<i>Műveleti kód reg.</i> XXX00000 (<i>Felülírás</i>)			
<i>Eredeti adat</i>	<i>Memória lapozó reg.</i>	<i>Felhasználói adat alsó 4 bitje</i>		<i>Új adat</i>

IBM PC Gyakorlati Assembly

Plane3	XXXXXXXX	0	X	XXXXXXXX
Plane2	XXXXXXXX	1	1	XXXX1111
Plane1	XXXXXXXX	1	0	XXXX0000
Plane0	XXXXXXXX	1	1	XXXX1111

00001111

*Adatváltoztatás
engedélyezése*

EGA - VGA kártyák 2. írási módja

A VGA kártyáknak van még további egy írási módja, amikor az adatváltoztatás engedélyezését nem csak a regiszter határozza meg, hanem a felhasználó által beírt érték és a regiszter együttesen határozza meg. A plane-ekre beíródó adatot pedig a SET-RESET adat határozza meg a következő módon:

Műveleti kód reg.

XXX 00 010 (*Felülírási mód 2 bitnyi forgatással*)

		<i>Set-Reset engedélyező</i>			
	<i>Eredeti adat</i>	<i>Memória lapozó reg.</i>	<i>Set-Reset adat</i>	<i>Új adat</i>	
Plane3	XXXXXXXX	1	1	1	XXX11X11
Plane2	XXXXXXXX	1	1	1	XXX11X11
Plane1	XXXXXXXX	1	1	0	XXX00X00
Plane0	XXXXXXXX	1	0	X	XXXXXXXX
					00011111
					<i>Adatváltoztatás engedélyezés</i>
					AND
					01101111
					<i>Felhasználói adat</i>
					11011011
					<i>A felhasználói adat forgatás után</i>

VGA kártyák 3. írási módja

Látható, hogy az adatváltoztatást engedélyező regisztert AND kapcsolatba hoztuk az általunk beírt adat elforgatott értékével, és a SET-RESET adat csak arra a plane-re íródott be, amely biteken a művelet eredménye 1 volt, a többit változatlanul

hagyja. A forgatás mértékét a műveleti kód regiszter alsó 3 bitje határozza meg. Jelen esetben 2 bit.

A 16 színű módokban nem csak az írás történik a megszo-
kottól eltérően, hanem az olvasás is. Ez következik abból, hogy
itt egy byte-hoz nem 8 bit információ tartozik, hanem 32. Az egy-
szerűbbik mód, ahogyan ezeket olvasni lehet, hogy kiválasztunk
egy plane-t amelyik tartalmára szükségünk van, és egy egyszer-
ű MOV utasítással egy regiszterbe töltjük. Van azonban egy
másik lehetőség is, mégpedig egyszerre olvasni mind a 32 bitet.
No igen, ez nem férne bele egy byte-ba, de erre találták ki, hogy
nem a bitek tartalmát kapjuk meg információnak, hanem egy
előre beállított szín előfordulásának helyét. Tehát ha nekünk egy
piros színt kell keresni, egyszerűen beállítjuk a piros színt a
színfigyelő regiszterben és a képernyőtartalom olvasásakor nem
az ott levő adatokat kapjuk eredménynek, hanem azt, hogy az
olvasott képpontok közül melyik piros. Ezenkívül lehetőségünk
nyílik lapokat kizárni a figyelésből. Tehát lehet, hogy a képer-
nyőn van világos és sötét piros is amik például egy bitjükben kü-
lönböznek. Ennek figyelését letilthatjuk, és mind a sötét, mind a
világos piros színre egyezést fog mutatni a gép olvasáskor.
Ezen két olvasási mód működése:

	<i>Képernyő tartalom</i>	<i>Kiolvasási lapozó regiszter XXXXXXXX 11</i>	<i>A kiolvasás eredménye</i>
<i>Plane3</i>	00101101	+	00101101
<i>Plane2</i>	01101010	-	
<i>Plane1</i>	00101010	-	
<i>Plane0</i>	11010011	-	

EGA - VGA kártyák 0. kiolvasási módja

	<i>Képernyő tartalom</i>	<i>Színfigyelés maszkregiszter</i>	<i>Színfigyelő regiszter</i>
<i>Plane3</i>	00101101	0	X
<i>Plane2</i>	01111110	1	1
<i>Plane1</i>	00101010	1	0
<i>Plane0</i>	11010111	1	1

01010100

*A kiolvasás
eredménye*

EGA - VGA kártyák 1. kiolvasási módja

Amint az látható, a 3. bitsík ki lett tiltva a figyelésből így mindazon helyekre egyező értéket adott, ahol a 101 kombinációt találta.

A következőkben ismertetésre kerül az EGA üzemmód grafikus vezérlőjének regiszterkiosztása:

<u>Portcím</u>	<u>Típus</u>	<u>Funkció</u>
3CEh	Csak írható	Grafikus vezérlő címregisztere
3CFh	Írható, olvasható	Grafikus vezérlő adatregisztere
00h	csak írható	SET-RESET adat regiszter 7-4: Nem használt 3: A 3. plane adata 2: A 2. plane adata 1: Az 1. plane adata 0: A 0. plane adata
01h	csak írható	SET-RESET engedélyező regiszter 7-4: Nem használt 3: A 3. plane engedélyezése 2: A 2. plane engedélyezése 1: Az 1. plane engedélyezése 0: A 0. plane engedélyezése
02h	csak írható	Színfigyelő regiszter 7-4: Nem használt 3: A 3. plane adata 2: A 2. plane adata 1: Az 1. plane adata 0: A 0. plane adata
03h	csak írható	Műveleti kód regiszter 7-5: Nem használt 4-3: Írási művelet kiválasztása 00: Felülírás

		01: AND kapcsolat 10: OR kapcsolat 11: XOR kapcsolat 2-0: Rotálás értéke										
04h	Csak írható	Kiolvasási lapozó regiszter 7-2: Nem használt 1-0: Olvasni kívánt bitsik száma										
05h	Csak írható	Módregiszter 7-6: Nem használt 5: A video shiftregiszter vezérlése 0 - Folytonos adattárolás 1 - Páros; páratlan adattárolás 4: Címzési mód beállítása 0 - Folyamatos memória címzés 1 - Páros; páratlan címzés 3: Kiolvasási mód beállítása 0 - Közvetlen kiolvasás 1 - A szinfigyelés eredménye 2: Grafikus vezérlő kimeneteinek tiltása 1 - 0: Írési mód beállítása 00: 0. Írési mód 01: 1. Írési mód 10: 2. Írési mód 11: 3. Írési mód (Csak VGA kártyán)										
06h	Csak írható	Grafikus parancsregiszter 7 - 4: Nem használt 3 - 2: Memóriatartomány beállítása <table style="margin-left: 20px; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left;">Kezdőcím</th> <th style="text-align: left;">Hossz</th> </tr> </thead> <tbody> <tr> <td>00 - 0A000:0000</td> <td>128K</td> </tr> <tr> <td>01 - 0A000:0000</td> <td>64K</td> </tr> <tr> <td>10 - 0B000:0000</td> <td>32K</td> </tr> <tr> <td>11 - 0B800:0000</td> <td>32K</td> </tr> </tbody> </table> 1: Páros; páratlan címzésű kezelés 0: Grafikus címzési mód	Kezdőcím	Hossz	00 - 0A000:0000	128K	01 - 0A000:0000	64K	10 - 0B000:0000	32K	11 - 0B800:0000	32K
Kezdőcím	Hossz											
00 - 0A000:0000	128K											
01 - 0A000:0000	64K											
10 - 0B000:0000	32K											
11 - 0B800:0000	32K											
07h	Csak írható	Szinfigyelés maszkregiszter 7-4: Nem használt 3: A 3. plane engedélyezése 2: A 2. plane engedélyezése 1: Az 1. plane engedélyezése 0: A 0. plane engedélyezése										

08h Csak írható

Adatváltoztatás engedélyezése

Az adatváltoztatás engedélyezett azon helyeken, ahol az engedélyező regiszter értéke 1

A VGA kártyákon ezen regiszterek annyiban módosulnak, hogy nem csak írhatók, hanem értéküket ki is lehet olvasni.

Szó esett már arról, hogy az EGA illetve VGA kártyáknak van saját karakterkészletük. A grafikus képernyőre történő írás azonban egy kicsit eltér a szöveges üzemmódtól, mivel itt nem elég a betű kódját kiírni a képernyőre. Egy karakter megjelenítése hasonló módon történik, mint ahogyan egy ábra kirakása történ, valahol le van tárolva a betűk képe, és a betű kódjától függően kiválasztjuk, hogy melyik alak felel meg az adott betűnek és azt kitesszük a képernyőre. Így bármilyen formájú karakterkészletet előállíthatunk, de használhatjuk a videokártyában lévőket is. Ezek címének meghatározására segítségül szolgál a BIOS 10h rutinjának egyik funkciója. Egy szöveg kirakására van egy egyszerűbb, kevésbé igényes, rövidebb módszer is. Ez természetesen a ROM BIOS használata. Erre láthatunk példát a következő példaprogramban:

[Program 22]

```

vga2      segment                ;Szegmensdefiníció.
          assume cs:vga2,ds:vga2 ;Cs és ds beállítása.

start:    mov  ax,vga2            ;Ds regiszter beállítása
          mov  ds,ax              ;a szegmens elejére.

          mov  ax,12h              ;640*480/16 VGA mód
          int  10h                 ;beállítása.
          mov  ax,1123h            ;8*8 pontos grafikus
          int  10h                 ;karakterkészlet betöltése
          ;a ROM-ból

          mov  ax,ds               ;Ds regiszter értéket
          mov  es,ax               ;es-be is beírjuk
          mov  bh,0                 ;A 0. lapra
          mov  bl,12                ;8 pontos betűmérettel
          mov  dl,10                ;piros színnel
          mov  dh,10                ;a 10,10 koordinátába
    
```


Grafika programozása Assembly nyelven

```
mov cx,59 ;kiradjuk a TEXT1 címke
mov bp,offset TEXT1 ;alatt tárolt 59 betűs
mov ax,1300h ;szöveget.
int 10h

mov ax,1122h ;8*14 pontos grafikus
int 10h ;karakterkészlet betöltése
;a ROM-ból

mov ax,1300h
mov bh,0 ;14 pontos betűmérettel
mov bl,12 ;piros színnel
mov dl,10 ;a 10,10 koordinátába
mov dh,10 ;kiradjuk a TEXT2 címke
mov cx,60 ;alatt tárolt 60 betűs
mov bp,offset TEXT2 ;szöveget.
int 10h

mov ax,1124h ;8*16 pontos grafikus
int 10h ;karakterkészlet betöltése
;a ROM-ból

mov ax,1300h
mov bh,0 ;16 pontos betűmérettel
mov bl,12 ;piros színnel
mov dl,10 ;a 10,13 koordinátába
mov dh,13 ;kiradjuk a TEXT3 címke
mov cx,60 ;alatt tárolt 60 betűs
mov bp,offset TEXT3 ;szöveget.
int 10h

xor ax,ax ;billentyűvárás
int 16h
mov ax,3 ;80*25 text képernyő.
int 10h

mov ax,4c00h ;Visszatérés a DOS-hoz
int 21h

TEXT1: db "Ez egy szöveg a VGA grafikus "
db "képernyőn 8*8 pontos betűkkel."
TEXT2: db "Ez egy szöveg a VGA grafikus "
db "képernyőn 8*14 pontos betűkkel."
TEXT3: db "Ez egy szöveg a VGA grafikus "
db "képernyőn 8*16 pontos betűkkel."
```

```

vga2      ends                ;A szegmens vége.
          end  start          ;A program vége.
    
```

A program működése során a 10h BIOS rutin 2 alfunkcióját használja a szöveg megjelenítésére. Az egyikkel betölti a használni kívánt karakterkészletet, a másikkal pedig kirakja a megadott szöveget a kiválasztott lapra (ez a lap nem azonos a plane-nel) a megfelelő koordinátába a megfelelő színnel. A program futtatása során megfigyelhető, hogy a szövegeket nagyjából azonos távolságra teszi egymástól pedig az első két szöveg koordinátája megegyezik, csak a harmadik függőleges pozíciója lett kettővel megnövelve. Ennek oka, hogy a koordináta kiszámításnál az adott betűtípus méretét veszi alapul a gép. És így az első szöveg 8 pont magas, a 2. már 14 képpont magas betűkből áll. Egy szöveg ilyen módon történő kirakásánál az első lépés, hogy betöltsük a használni kívánt karakterkészletet. Ezt a 10h BIOS rutin 11h programja végzi. **AI** regiszterben megadjuk, hogy melyik készletet töltsse be:

AI = 22h - 8*14 pontos grafikus karakterkészlet betöltése

AI = 22h - 8*8 pontos grafikus karakterkészlet betöltése

**AI = 23h - 8*16 pontos grafikus karakterkészlet betöltése
(csak VGA kártyákon)**

Ezután jöhet a kiíratás. Az **es** szegmensregiszterben kell megadni a kiírandó szöveg szegmencímét **bp** regiszterben pedig az offsetcímét. A kiíratást a 10h megszakítás 13h rutinja végzi. **AI** regiszterben különféle alfunkciókat lehet beállítani:

AI = 00h - A szöveg csak a karakterek kódjait tartalmazza, a szint **bl regiszterben kell beállítani, a kurzort nem mozgatja.**

AI = 01h - A szöveg csak a karakterek kódjait tartalmazza, a szint **bl regiszterben kell beállítani, A kurzort mozgatja.**

AI = 10h - A szöveg a karakter és színekódokat felváltva tartalmazza, a kurzort nem mozgatja.

AI = 11h - A szöveg a karakter és színekódokat felváltva tartalmazza, a kurzort mozgatja.

A továbbiakban bármilyen 16 színű üzemmódot használunk is, az itt említett információk többé-kevésbé igazak, minimális módosítással alkalmazhatók.

Most a VGA kártyák egy újabb lehetőségéről, a 640*480/2 üzemmódjáról lesz szó. Ha már tudjuk kezelni a 16 színű módot, akkor ezzel biztos, hogy nem lesznek problémáink, mivel a felépítése azonos a 16 színű mód egy plane-jével. Egy byte 8 képpontot tartalmaz, tehát az adatok bittérképesen tárolódnak, 0 a háttérszín, 1 a kiválasztott tintaszín. A memóriában a pontok tárolása lineárisan egymás után történik és az egyik sor utolsó pontját a következő sor első pontja követi. Az írás és olvasás teljesen hétköznapi módon történik, egy egyszerű MOV utasítással. Előnye, a gyors és egyszerű kezelhetőség. A példaprogramban egy ábra kirakását mutatom be a 16 színű felbontásnál használt színállítással kombinálva:

[Program 23]

```

vga3      segment                ;Szegmensdefiníció.
          assume cs:vga3,ds:vga3 ;Cs és ds beállítása.

start:    mov  ax,vga3            ;Ds regiszter beállítása
          mov  ds,ax              ;szegmens elejére.

          mov  ax,11h              ;640*480/2 mód beállítása.
          int  10h

          mov  ax,0a000h           ;A videomemória kezdőcímét es
          mov  es,ax              ;szegmensregiszterbe töltjük.

          mov  bx,word ptr [KOORD_Y] ;Az ábra bal felső sarkának
          mov  ax,80               ;kiszámítása
          mul  bx
          mov  di,ax
          mov  bl,8
    
```

```

mov ax,word ptr [KOORD_X]
div bl
mov cl,ah
xor ah,ah
add di,ax
mov dl,10000000b ;Beállítomaszk
mov dh,01111111b ;Törölmaszk
ror dl,cl
ror dh,cl
mov si,offset IKON
mov cx,16

.1_vga3: push di cx
mov cx,16 ;A CGA ábrakirakó rutinhoz
mov ax,[si] ;hasonlóan ax regiszter
.2_vga3: shl ax,1 ;forgatásával 1-be illetve
jc .3_vga3 ;0-ba állítom a képernyő-
and es:[di],dh ;memória megfelelő bitjeit.
jmp .4_vga3

.3_vga3: or es:[di],dl

.4_vga3: ror dl,1
ror dh,1
jc .5_vga3
inc di

.5_vga3: loop .2_vga3
pop cx di
add si,2
add di,80 ;A műveletet mind a 16 soron
loop .1_vga3 ;elvégzem.

.6_vga3: mov dx,3c8h
mov al,63 ;Az ábra beállítása a
out dx,al ;megfelelő színüre.
inc dx
mov al,byte ptr [PIROS]
out dx,al
mov al,byte ptr [ZOLD]
out dx,al
mov al,byte ptr [KEK]
out dx,al

```

.7_vga3:	<pre>xor ax,ax int 16h cmp al,"q" jnz .8_vga3 mov al,63 cmp byte ptr [PIROS],al jnc .7_vga3 inc byte ptr [PIROS] jmp .6_vga3</pre>	<pre>;A q billentyű figyelése. ;Piros összetevő növelése ;ha még nem 63.</pre>
.8_vga3:	<pre>cmp al,"a" jnz .9_vga3 mov al,0 cmp byte ptr [PIROS],al jz .7_vga3 dec byte ptr [PIROS] jmp .6_vga3</pre>	<pre>;Az a billentyű figyelése. ;Csökkentése ha nem 0.</pre>
.9_vga3:	<pre>cmp al,"w" jnz .10_vga3 mov al,63 cmp byte ptr [ZOLD],al jnc .7_vga3 inc byte ptr [ZOLD] jmp .6_vga3</pre>	<pre>;A w billentyű figyelése. ;Zöld összetevő növelése ;ha még nem 63.</pre>
.10_vga3:	<pre>cmp al,"s" jnz .11_vga3 mov al,0 cmp byte ptr [ZOLD],al jz .7_vga3 dec byte ptr [ZOLD] jmp .6_vga3</pre>	<pre>;Az s billentyű figyelése. ;Csökkentése ha nem 0.</pre>
.11_vga3:	<pre>cmp al,"e" jnz .12_vga3 mov al,63 cmp byte ptr [KEK],al jnc .7_vga3 inc byte ptr [KEK] jmp .6_vga3</pre>	<pre>;Az e billentyű figyelése. ;Kék összetevő növelése ;ha még nem 63.</pre>
.12_vga3:	<pre>cmp al,"d" jnz .13_vga3 mov al,0</pre>	<pre>;A d billentyű figyelése.</pre>

IBM PC Gyakorlati Assembly

```

        cmp     byte ptr [KEK],al           ;Csökkentése ha nem 0.
        jz      .7_vga3
        dec     byte ptr [KEK]
        jmp     .6_vga3

.13_vga3:  cmp     al,13                     ;ENTER billentyű figyelése.
        jnz     .7_vga3

        mov     ax,3                         ;Ha igen, 80*25 text képernyő.
        int     10h

        mov     ax,4c00h                    ;Visszatérés a DOS-hoz.
        int     21h

PIROS:    db     31                         ;Az egyes színösszetevők
ZOLD:    db     31                         ;kiinduló értéke.
KEK:     db     31

IKON:    dw     000011111100000b
        dw     0001100000011000b
        dw     001000000000100b
        dw     0100110000110010b
        dw     0101011001011010b
        dw     1000110000110001b
        dw     1000000010000001b
        dw     1000000101000001b
        dw     1000000101000001b
        dw     1000001001000001b
        dw     1001000110001001b
        dw     0100110000110010b
        dw     0100011111100010b
        dw     0010001111000100b
        dw     0001100000011000b
        dw     0000011111100000b

KOORD_X: dw     312                         ;Az ábra X és Y koordinátája
KOORD_Y: dw     232

vga3     ends                                 ;A szegmens vége
        end     start                       ;A program vége

```

Amint az látható, a program egy kicsit egyszerűbb mint a CGA üzemmódnál bemutatott kirakó rutin. Működése szinte azonos vele, csak a címkiszámítás sokkal egyszerűbb. Itt a képernyőcímet úgy kapjuk meg, hogy a függőleges koordináta értékét

megszorozzuk 80-nal, a vízszintes értéket elosztjuk 8-al és a kettőt összeadjuk. A byte-on belüli pozíciót pedig az osztásnál kapott maradék határozza meg. Itt ezt betöltjük `cl` regiszterbe, és a maszkregisztereket `cl` értékének megfelelően forgatjuk. A háttérszint itt is a 0. paletta határozza meg, az előtér színét azonban a 63. paletta változtatásával tudjuk befolyásolni.

Az MCGA üzemmód működése:

A következőkben számomra és sok programozó számára is az egyik legszimpatikusabb VGA mód ismertetése következik. Ez a 320*200-as felbontás 256 féle színnel. Hogy miért tetszik olyan sok embernek ez az üzemmód? Nos eddig a címkiszámításoknál a vízszintes koordinátát osztani kellett, azután a maradékkal forgatni, figyelni, hogy az ábrakirakásnál mikor érünk a byte szélére stb. Nos itt ez nincs. Itt egy képpontot egy byte határoz meg. Ugyanis csak így lehet 256 féle színt kirakni egy képernyőre. A színek sorszáma itt megegyezik a paletta sorszámaival. A 0. a háttérszín, az összes többi pedig tintaszínként alkalmazható. A felbontás ugyan nem a legjobb, de a 256 féle színlehetőség valamelyest kárpótol minket. A képernyő-memória kezdőcíme itt is 0A000h és a sorok lineárisan egymás után vannak letárolva a memóriában. A következő mintaprogram egy színes ábra kirakását mutatja be a hozzá tartozó paletta beállítással. A későbbiekben ezt a rutint fogjuk továbbfejleszteni egy komplett egérkezelő rutinná:

[Program 24]

vga4	segment assume cs:vga4,ds:vga4	;Szegmensdefiníció. ;Cs és ds beállítása.
start:	mov ax,vga4 mov ds,ax	;Ds regiszter beállítása a ;szegmens elejére.
	mov ax,0a000h mov es,ax	;A videomemória kezdőcímet ;es regiszterbe írjuk.
	mov ax,13h int 10h	;A 320*200/256 üzemmód ;beállítása

```

mov  dx,3c8h           ;Fekete háttérszín
xor   ax,ax           ;beállítás
out   dx,al
inc   dx
out   dx,al

mov   cx,13           ;A használt 13 előtér szín
mov   al,1            ;beállítás a letárolt
mov   si,offset PALETTA ;értékeknek megfelelően.

.1_vga4:  push  ax
          dec   dx
          out   dx,al
          inc   dx
          mov   al,[si]
          out   dx,al
          mov   al,1[si]
          out   dx,al
          mov   al,2[si]
          out   dx,al
          add   si,3
          pop   ax
          inc   al
          loop  .1_vga4

          mov   bx,word ptr [KOORD_Y]
          mov   ax,320           ;A bal felső sarok címének
          mul   bx              ;kiszámítása.
          mov   di,ax
          add   di,word ptr [KOORD_X]
          mov   si,offset IKON  ;Az ábra kirakása.
          mov   cx,8

.2_vga4:  push  cx di
          mov   cx,8

.3_vga4:  mov   al,[si]
          mov   es:[di],al
          inc   di
          inc   si
          loop  .3_vga4
          pop   di cx
          add   di,320
          loop  .2_vga4

```


Grafika programozása Assembly nyelven

	xor ax,ax	;billentyűvárás.
	int 16h	
	xor bx,bx	;A paletták görgetése.
	mov si,offset PALETTA	;Si+bx az első palettaszínré
.4_vga4:	mov ax,1	;mutat.
	mov cx,13	;13 palettaszint kell
		;állítani.
	push bx	;A kezdőértéket elmentjük,
		;hogy tudjuk majd változ-
		;tatni az értékét
.5_vga4:	push ax	;Állítandó paletta sorszá-
		;mának elmentése, hogy később
		;innen tudjuk folytatni.
	mov dx,3c8h	;A megfelelő palettaszín
	out dx,al	;váltóztatása.
	inc dx	
	mov al,[si+bx]	
	out dx,al	
	mov al,1[si+bx]	
	out dx,al	
	mov al,2[si+bx]	
	out dx,al	
	pop ax	
	inc ax	
	add bx,3	;A következő 3 színkomponens.
	cmp bx,54	;Ha a végére ért, akkor újra
	jc .6_vga4	;előről.
	xor bx,bx	
.6_vga4:	loop .5_vga4	
	pop bx	;A kezdőértéket eggyel
	add bx,3	;eltoljuk, ettől futnak a
	cmp bx,54	;színek. Ha elérte a véget,
	jc .7_vga4	;ismét előről.
	xor bx,bx	
.7_vga4:	mov ax,100h	;Ha van lenyomott billentyű,
	int 16h	;a befejezésre ugrik.
	jnz .10_vga4	
	mov cx,02ffff	;Egy holtciklus beiktatásával

IBM PC Gyakorlati Assembly

```

.8_vga4:  loop  .8_vga4           ;lassítjuk a program futását.
             mov   dx,3dah       ;A program futása csak akkor

.9_vga4:  in    al,dx           ;folytatódik, ha az elektron-
             test  al,1000b       ;sugár éppen a függőleges
             jz   .9_vga4       ;visszatérítést végzi.
             jmp  .4_vga4

.10_vga4: xor   ax,ax         ;A lenyomott billentyű
             int  16h           ;kiolvasása

             mov  ax,3          ;80*25 szöveges képernyő
             int  10h

             mov  ax,4c00h     ;Kilépés a DOS-ba
             int  21h

IKON:    db   1,2,3,4,5,6,7,8
             db   2,3,4,5,6,7,8,0
             db   3,4,5,6,7,8,0,0
             db   4,5,6,7,8,9,0,0
             db   5,6,7,8,9,10,11,0
             db   6,7,8,9,10,11,12,13
             db   7,8,0,0,11,12,13,0
             db   8,0,0,0,0,13,0,0

PALETTA: db  40,10,10,35,15,10,30,20,10,25,25,10,20,30,10,15,35,10
             db  10,40,10,10,35,15,10,30,20,10,25,25,10,20,30,10,15,35
             db  10,10,40,15,10,35,20,10,30,25,10,25,30,10,20,35,10,15

KOORD_X: dw  156           ;Az ábra X és Y koordinátája
KOORD_Y: dw  96

vga4     ends           ;Szegmens vége
             end   start   ;Program vége

```

A program egy paletta-beállítással kezdődik, feketére állítja a háttérrel és az előre letárolt értékeknek megfelelően beállítja az első 13 előtér szint. Ezután jön az ikon koordinátáinak kiszámítása egy szorzással és egy összeadással. Az így kapott címre egy 8*8-as ciklus segítségével kirakjuk a tárolt ábrát. Hogy ne a már megszokott látvány táruljon elénk, egy gombnyomás után egy látványos ám igen egyszerű rutin lép működésbe. A trükk lényege, hogy görgeti az első 13 palettát és így egy érdekes ef-

fektust hoz létre. A görgetés itt úgy lett megoldva, hogy le van tárolva egy 18 elemű paletta és ebből kiválasztunk 13-at. Először a 1-13-ig utána 2-14-ig. Az 5-18 után mivel nincsen 19. szín letárolva, a 6-18,1 palettaszíneket fogja az első 13 színhelyre beírni. A lényeg, hogy mindig 13 egymás utáni színt választunk ki, és a kezdőpontot egy színnel mindig arrébb toljuk. Ebben a programban a billentyűzetfigyelés is másképpen lett megoldva, mivel az eddiginél addig várt a program, míg le nem nyomtunk egy gombot. Itt ez nem megoldható, mivel a program folyamatosan fut, csak egy billentyű lenyomásakor kell hogy megálljon, ezért a 16h rutin egy másik lehetősége lett alkalmazva, mégpedig a billentyűzet státusz figyelése. A rutinból való visszatéréskor a jelzőbit 1-es értéke mutatja ha nincs lenyomott billentyű, 0 ha van, ekkor kilépés a programból, de előbb ki kell olvasni a lenyomott billentyű kódját, mert különben ez benne maradna a billentyűzet pufferben. Hogy a színek futása ne legyen darabos, töredezett be lett építve egy vertical blank figyelés, így biztos, hogy akkor fogja a gép megváltoztatni az ábra színeit, amikor az elektronsugár éppen visszafelé halad a jobb alsó sarokból a bal felsőbe.

Ezt a színvariálást az egérkezelő rutinban nem fogjuk alkalmazni, de aki úgy gondolja, beleteheti. A program a koordinátaszámításig azonos az előzővel. Az egérkezelést azonban egy kicsit másként fogjuk megoldani mint a 19. programban, mivel itt elég ha arra figyelünk, hogy ne rakjunk nullákat a képernyőre. Ha egy pontot kirakunk a képernyőre, az biztos hogy olyan színű lesz amelyet akartunk, mivel itt lehet a MOV utasítást használni az ábra kirakásához mert egy képpont egy teljes byte-ot foglal és így nem változtatja a körülötte levő pontokat. Természetesen az alatta levő képet itt is el kell tárolnunk mert az egér mozgása következtében elveszne. Egy további változtatás teszi teljesebbé a rutint, mégpedig, hogy a képernyőn kinn lesz négy másik ábra is, ezek valamelyikére rákattintva lehet a programból kilépni. Ezzel a módszerrel akárhány helyet lehet figyelni, mivel az ellenőrizni kívánt mezők bal felső és jobb alsó koordinátáit kell csak megadni felsorolásszerűen egy táblázatban. A programba a vertical blank-hez időzítés már szintén be van építve:

[Program 25]

vga5	segment	;Szegmensdefinicio.
	assume cs:vga5,ds:vga5	;Cs és ds beállítása.
start:	mov ax,vga5	;Ds regiszter beállítása a
	mov ds,ax	;szegmens elejére.
	xor ax,ax	;Az egér installálása.
	int 33h	
	cmp ax,0	
	jnz .1_vga5	;Ha nincs hiba, akkor ;elkezdődhet a program.
	mov ax,3	;80*25 karakteres kép
	int 10h	;beállítása.
	mov ax,0b800h	;A szöveges képernyő
	mov es,ax	;kezdőcíme
	mov di,52	;0. sor 26. karaktere
	mov ah,7	;fekete alapon fehér ;betűk lesznek.
.1_hiba:	mov si,offset HIBATEXT	
	mov al,[si]	;Az egérhiba szöveg kiíratása.
	cmp al,0	;Ha a szövegbe 0 kód van,
	jz .2_hiba	;az a szöveg végét jelenti.
	mov es:[di],ax	;Ha nem, akkor kiírható a
	add di,2	;karakter.
	inc si	;Következő karakter mindaddig,
	jmp .1_hiba	;míg a 0 kódhoz nem ;ér a program
.2_hiba:	xor ax,ax	;billentyűvárás
	int 16h	
	jmp kilepes	
.1_vga5:	mov ax,0a000h	;A videomemória kezdőcímet
	mov es,ax	;es regiszterbe írjuk.
	mov ax,13h	;A 320*200/256 üzemmód
	int 10h	;beállítása
	mov dx,3c8h	;Fekete háttérszín

Grafika programozása Assembly nyelven

```
xor ax,ax ;beállítása
out dx,al
inc dx
out dx,al

mov cx,14 ;A használt 14 előtérszín
mov al,1 ;beállítása a letárolt
mov si,offset PALETTA ;értékeknek megfelelően.

.2_vga5: push ax
dec dx
out dx,al
inc dx
mov al,[si]
out dx,al
mov al,1[si]
out dx,al
mov al,2[si]
out dx,al
add si,3
pop ax
inc al

loop .2_vga5

.3_vga5: mov si,offset ITABLA ;Az ITABLA címke alatt
mov ax,[si] ;tárolt koordinátákra
cmp ax,0ffffh ;kirakja a kilépő ikonokat

jz .4_vga5 ;A 0ffffh kód jelzi a
mov bx,2[si] ;koordináta felsorolás végét.
mov word ptr [KOORD_X],ax
mov word ptr [KOORD_Y],bx
push si
mov si,offset IKON2
call cimki ;Az ikon memóriacímének
call kirako ;kiszámítása és kirakása.
pop si
add si,4
jmp .3_vga5

.4_vga5: mov ax,3 ;Az egér kezdő koordinátáinak
int 33h ;beállítása
mov word ptr [KOORD_X],cx
mov word ptr [KOORD_Y],dx
```

IBM PC Gyakorlati Assembly

.5_vga5:	call	cimki	;Az egér memóriacímének ;kiszámítása.
	call	elrako	;Az ikon alatti kép ;eltárolása
	call	eger	;Az egér kirakása
.6_vga5:	mov	ax,3	;Figyeli, hogy van-e
	int	33h	;lenyomott gomb
	test	bx,3	
	jz	.7_vga5	
	call	gombok	;Ha van, ugrik a vizsgalatra.
	jnc	kilepes	;Ha a rutinból való vissza- ;téréskor c flag 0 értékű, ;az azt jelenti, hogy egy ;ábra területén nyomtuk ;le az egér valamelyik ;gombját és ekkor kilépés.
.7_vga5:	call	mozgas	;Az elmozdulás vizsgálata.
	jz	.6_vga5	;Ha van elmozdulás, z=0
	cmp	cx,625	;A képernyőből való
	jc	.8_vga5	;kicsúszást is figyelni kell,
	mov	cx,624	;ha nagyobb a maximumnál
.8_vga5:	cmp	dx,193	;akkor a maximumértéket
	jc	.9_vga5	;kell visszaállítani.
	mov	dx,192	
.9_vga5:	mov	word ptr [KOORD_X],cx	;Az új egérkoordinátákat
	mov	word ptr [KOORD_Y],dx	;a memóriaváltozókbá írjuk.
	mov	ax,4	
	int	33h	
	call	vertbl	;Időzítés a vertical bl-hez.
	mov	si,offset HELY	;Az eltárolt képet vissza-
	call	kirako	;rakjuk a helyére
	jmp	.5_vga5	
kilepes:	mov	ax,3	;80*25 szöveges képernyő
	int	10h	

Grafika programozása Assembly nyelven

```
mov ax,4c00h           ;Kilépés a DOS-ba
int 21h

eger proc

mov di,word ptr [CIM]
mov si,offset IKON
mov cx,8               ;Az ikon 8 képsor magas

.1_eger: push cx
mov cx,8               ;és 8 pont széles.

.2_eger: mov al,[si]    ;Megvizsgáljuk, hogy az
cmp al,0               ;ikon adott pontját ki kell-e
jz .3_eger             ;rakni. Mert ha a pont értéke
                       ;0, akkor azt nem szabad
                       ;kirakni, mert az törölné
                       ;az eger körülötti képet.

mov es:[di],al        ;A hasznos pont kirakása.

.3_eger: inc di        ;A következő pont.
inc si
loop .2_eger
pop cx
add di,312            ;A következő sor.
loop .1_eger

ret

eger endp

elrako proc

mov si,word ptr [CIM]
mov di,offset HELY
mov cx,8               ;Egy 8*8-as ciklus

.1_elrako: push cx     ;segítségével a di által
mov cx,4               ;meghatározott helyről

.2_elrako: mov ax,es:[si] ;a HELY címre tároljuk a
mov [di],ax           ;képernyőn lévő képet.
add di,2
add si,2
loop .2_elrako
```

```

    pop    cx
    add    si,312
    loop   .1_elrako

    ret

elrako    endp

```

kirakoproc

```

    mov    di,word ptr [CIM]
    mov    cx,8                ;A forrás helyét előre
.1_kirako: push cx            ;beállítottuk si-be.
    mov    cx,4                ;Így 8*4 word-öt ds:si-ról
    rep    movsw                ;es:di-re másol.
    pop    cx
    add    di,312
    loop   .1_kirako

    ret

```

kirakoendp

vertbl proc

```

.1_vb:    mov    dx,3dah
    in     al,dx
    test   al,1000b
    jz     .1_vb

```

```

.2_vb:    in     al,dx
    test   al,1000b
    jnz    .2_vb

```

ret

vertbl endp

mozgas proc

```

    mov    ax,3                ;Az egér koordinátáit
    int    33h                ;összehasonlítja az előző
    cmp    cx,word ptr [KOORD_X] ;letárolt értékkel, és ha

```


Grafika programozása Assembly nyelven

```
jnz .1_mozgas ;eltérést talál (az egér  
cmp dx,word ptr [KOORD_Y] elmozdult) akkor z=0  
;értékkel tér vissza a hívó  
;programhoz.  
  
.1_mozgas: ret  
  
mozgas endp  
  
cimki proc  
  
mov bx,word ptr [KOORD_Y] ;A bal felső sarok címének  
mov ax,320 ;kiszámítása.  
mul bx  
mov di,ax  
mov ax,word ptr [KOORD_X]  
shr ax,1 ;A címkiszámításnál figyelembe  
add di,ax ;kell venni, hogy a vízszintes  
mov word ptr [CIM],di ;koordináta duplája lett  
;letárolva az egér sajátossága  
;miatt.  
  
ret  
  
cimki endp  
  
gombok proc  
  
mov si,offset GTABLA ;A GTABLA címke alatt tárolt  
;területeket fogja  
;ellenőrizni a program,  
.1_gombok: cmp [si],0ffffh ;melynek a végét egy 0ffffh  
;kód jelzi.  
  
jz .3_gombok ;Ha ezt elérte, nem volt  
;a vizsgálandó helyek  
;semelyikén az egér, c=1  
;paraméterrel tér vissza.  
  
mov ax,[si] ;A bal felső koordináta  
mov bx,2[si] ;vizsgálata. Ha az egér  
cmp word ptr [KOORD_X],ax ;valamelyik koordinátája  
jc .2_gombok ;kisebb ezeknél biztos, hogy  
cmp word ptr [KOORD_Y],bx ;nem esik az adott helyre.  
jc .2_gombok
```

```

mov ax,4[si]           ;A jobb alsó sarok vizsgálá-
mov bx,6[si]           ;lata. Ha az egér bármely
cmp ax,word ptr [KOORD_X] ;koordinátája nagyobb
jc .2_gombok           ;ezeknél, tehát ha a vizsgált
cmp bx,word ptr [KOORD_Y] ;ponttól akár jobbra akár
jc .2_gombok           ;lejjebb van biztos, hogy
                       ;az egér nem egy ábrára
                       ;mutat.

ret                   ;Ha idáig elért a program,
                       ;akkor az egér egy ábra
                       ;valamely pontjára mutat.
                       ;Ez esetben visszatéréskor
                       ;c jelzőbit 0 értékű.

.2_gombok: add si,8    ;A következő mező vizsgálata.
             jmp .1_gombok

.3_gombok: stc        ;Az egér az egyik vizsgált
             ret       ;mező fölött sincs, vissza-
                       ;téréskor c=1.

gombok      endp

IKON:       db 1,2,3,4,5,6,7,8           ;Az egér ikonja.
             db 2,3,4,5,6,7,8,0
             db 3,4,5,6,7,8,0,0
             db 4,5,6,7,8,9,0,0
             db 5,6,7,8,9,10,11,0
             db 6,7,8,9,10,11,12,13
             db 7,8,0,0,11,12,13,0
             db 8,0,0,0,0,13,0,0

IKON2:      db 14,14,14,14,14,14,14,14 ;A kilépés ikonja
             db 14,0,0,0,0,0,0,14
             db 14,0,14,0,0,14,0,14
             db 14,0,0,14,14,0,0,14
             db 14,0,0,14,14,0,0,14
             db 14,0,14,0,0,14,0,14
             db 14,0,0,0,0,0,0,14
             db 14,14,14,14,14,14,14,14

HELY        db 64 dup (0)              ;Ide tárolja le a program
                                               ;az egér alatti képet.

```

Grafika programozása Assembly nyelven

```
PALETTA: db 40,10,10,35,15,10,30,20,10,25,25,10,20,30,10,15,35,10
          db 10,40,10,10,35,15,10,30,20,10,25,25,10,20,30,10,15,35
          db 10,10,40,63,63,63

ITABLA:  dw 60,30,240,50,24,170,600,140,0ffffh

GTABLA:  dw 60,30,74,37,240,50,254,57,24,170,38,177
          dw 600,140,614,147,0ffffh

KOORD_X: dw 156 ;Az ábra X és Y koordinátája
KOORD_Y: dw 96

CIM:     dw 0

HIBATEXT: db "Az egér nem installálható !",0

vga5    ends ;Szegmens vége
end      start ;Program vége
```

A program szokásos módon egy egérinstallálás után egy paletta beállítással kezdődik, majd az ITABLA táblázatban tárolt koordinátákra kirakja a kilépést szimbolizáló ábrákat. Az ábrák memóriacímének kiszámítását már külön rutin végzi, mivel ha minden kirakáshoz odaraktunk volna egy címkiszámító rutint, a program hossza a kétszeresére nőtt volna. És van még egy előnye a dolognak, mivel így nyugodtan számolhatunk 640*200 felbontással, mert a címkiszámító rutin ezt figyelembe véve számolja a memóriacímet. Egyébként az egérkezelő eljárások lettek volna sokkal bonyolultabbak. Ezután beállítjuk az egér kezdő pozícióját, és a már ismert módon, csak egérmaszknél kezeljük az egér mozgását mindaddig, míg nincs lenyomva az egérnek az egyik gombja sem. Amint lenyomtuk valamely gombot, elindul egy ellenőrző rutin, ami az aktuális egérpozíciót összehasonlítja a GTABLA címkénél letárolt mezők koordinátaival. Amennyiben a mező bal felső sarkától balra illetve feljebb, vagy a jobb alsó saroktól jobbra illetve lejjebb van az egér akkor biztos, hogy nem mutat a vizsgált mezőre. Ebben az esetben a következő területet kell vizsgálni. Ha nincs több mező, akkor egyik fölött sem volt egerünk ezért folytatódhat a program ott ahol éppen tartott. Ha egy mező fölött volt, akkor kilép a programból. Egy kicsit változtatva lett a vertical blank rutinon is mivel előfordulhat azaz eset is amikor a képernyő felső részén van az egér

és éppen a visszafutás legvégén kezdjük el kirakni az ábrát. Ilyenkor szintén furcsaságok történnek. A legbiztonságosabb a visszafutás legelejét elcsipni. Ezt úgy lehet elérni, hogy a vertical blank rutin megvárja míg az elektronsugár rajzolni fog, ezt végigvárja és ezután rögtön következik a képkirakás. Így biztos, hogy a vb. legelején kezdjük el kirakni az ábránkat. Figyelni kell arra is, hogy korlátlan méretű ábra kirakását ezzel a módszerrel nem időzíthetjük mivel csak egy bizonyos számú műveletet képes a processzor ezen rövid visszafutási idő alatt elvégezni. Tehát nagyobb ábrák időzített kirakását más trükkökkel kell megoldani.

Rajzolás a képernyőre:

A továbbiakban ismét egy klasszikus problémával fogunk foglalkozni mégpedig a vonalrajzolással. Aki egy kicsit ért a matematikához, ismeri az egyenes egyenletét amivel egy vonal egyes pontjainak koordinátáját könnyedén kiszámíthatjuk. Adódik azonban egy probléma, ami megnehezíti a feladat ilyen módon történő megoldását, ugyanis a számítógép nem ismeri a tört számokat. Sajnos csak egész számokkal hajlandó műveleteket végezni. Megtehetjük, hogy írunk egy lebegőpontos aritmetikát melynek segítségével használhatunk tört számokat is. Ez is egy megoldás azonban ehhez nem árt, ha egy ágyat is magunk mellé készítünk (no nem azért, mert belefáradunk a kód megírásába, ami ugyan nem könnyű) mivel az eredményt kívárni elég sok időbe telne. Ugyanis így a program minden egyes pont kiszámításánál az általunk megírt számoló rutint használná, ami sokkal tovább tart, mint az egész számokkal való művelet. Tehát a megoldás egy másik algoritmus, amivel legelőször a ZX Spectrum ROM-jában találkoztam. Ez nem jelenti azt, hogy ez egy elavult módszer hiszen mindmáig több helyen is láttam alkalmazását. A művelet lényege, hogy először kiválasztjuk a kezdőpontot. Ezután megnézzük, hogy a végpont milyen irányban és milyen távolságra van ettől, mindez x és y összetevőkre lebontva. Alapvetően két vonaltípust különböztetünk meg és ezeket már a program elején külön kell választani egymástól. A különbség az irányukból adódik, ugyanis az algoritmus lényege, hogy az egyik összetevőt minden lépésben növeli eggyel és kiszámolja az ehhez tartozó másik összetevőt. Ha nem bontanánk

külön a két típust, akkor előfordulhatna olyan eset, amikor például a 0,0;10,100 vonalat kéne kirajzolni és mi az x koordinátát növeljük egyesével, az y-t számítjuk. Ekkor a vonalból mindössze 10 pont látszana. Tehát az egyenes kirakása előtt el kell döntenünk, hogy a x illetve az y összetevője nagyobb a vonalnak. Ezt egy kivonással és összehasonlítással elvégezhetjük. Az előbb említett példánál az x összetevő 10-0 azaz 10 az y pedig 100-0 tehát 100, ebből adódóan mindig a nagyobbikat választjuk ki a lépkedésre ugyanis így biztos, hogy az egyenes minden pontja a képernyőre kerül. Fontos, még az is, hogy az egyenes végpontja milyen irányban helyezkedik el a kezdőponthoz képest, mivel, ha annak valamely koordinátája kisebb, akkor azt csökkenteni kell. Ezt a programban úgy oldottam meg, hogy nem kivonok belőle, hanem hozzáadok -1-et (65535-öt) az eredmény ugyanaz mivel a regiszter körbefordul. A vonal pontjait kiszámító módszer igen egyszerű, mindössze annyit kell tenni, hogy egy regiszterhez minden lépésben hozzáadom a kisebb koordináta összetevőt, és összehasonlítom a nagyobbikkal. Ha kisebb, nincs oldalirányú lépés, ha nagyobb akkor kivonjuk a regiszterből azt és teszünk egy lépést oldalirányba is. Így mire a vonal végére érünk, amit egy ciklussal állítunk a nagyobbik méret +1-re, pontosan a kijelölt ponthoz érünk.

[Program 26]

Vga6	Segment assume cs:Vga6,ds:Vga6	;Szegmensdefiníció ;Cs, ds hozzárendelése ;a szegmensregiszterekhez.
Start:	mov ax,Vga6 mov ds,ax	;Ds beállítása a kód elejére.
	mov ax,0a000h mov es,ax	;Es beállítása a képernyő- ;memória szegmenscímére.
	mov ax,13h int 10h	;320*200/256 felbontás ;beállítása.
	call Line	;A vonalhúzó rutin futtatása.
	xor ax,ax int 16h	;Billentyűvárás.

IBM PC Gyakorlati Assembly

	mov ax,3	;Vissza a Text képernyőre.
	int 10h	
	mov ax,4c00h	;Vissza a DOS-hoz.
	int 21h	
Line	Proc	;A vonalhúzó rutin kezdete.
	mov bx,word ptr [Xa]	;A vonal kezdőpontjának és
	mov ax,word ptr [Xb]	;végének x koordinátáját
		;beírja ax,bx regiszterekbe.
	mov dx,1	;A vízszintes lépésköz.
	sub ax,bx	;Kiszámítja a két pont közötti
		;távolságot.
	jnc .1_Line	;Ha ez nem negatív, átugorja
		;a következő részt.
	neg ax	;A távolság valódi értékének a
		;kiszámítása.
	mov dx,65535	;A vízszintes lépésközt -1-re
		;állítja.
.1_Line:	mov word ptr [DELTA_X],ax	;A regisztereket beírja a
	mov word ptr [PONT_X],bx	;memóriaváltozóba.
	mov word ptr [SGN_X],dx	
	mov bx,word ptr [Ya]	;Ugyanaz, ami a vízszintes
	mov ax,word ptr [Yb]	;értékeknél történt.
	mov dx,1	
	sub ax,bx	
	jnc .2_Line	
	neg ax	
	mov dx,65535	
.2_Line:	mov word ptr [DELTA_Y],ax	
	mov word ptr [PONT_Y],bx	
	mov word ptr [SGN_Y],dx	
	cmp ax,word ptr [DELTA_X]	;Eldönti, hogy a vonal milyen
		;irányú.

```

jc    .1_Vizsz                ;Ha a vízszintes összetevője
                                ;a nagyobb, arra az eljárásra
                                ;ugrik.

mov   dx,ax
shr   dx,1                    ;Egyébként az algoritmus
                                ;növekményét számláló
                                ;regiszterbe beírja a vonal
                                ;vízszintes hosszának a felét.

;Fuggoleges

mov   cx,ax                    ;A ciklusszámlálóba pedig a
inc   cx                       ;hossznál eggyel többet.

.1_Fuggo: push dx                ;Számológó elmentése.

call  Pont                     ;Az aktuális pont kirakása.

pop   dx
mov   ax,word ptr [SGN_Y]      ;Az algoritmus szerinti
add   word ptr [PONT_Y],ax     ;következő pont koordinátainak
add   dx,word ptr [DELTA_X]    ;kiszámítása.
cmp   dx,word ptr [DELTA_Y]
jc    .2_Fuggo

sub   dx,word ptr [DELTA_Y]
mov   ax,word ptr [SGN_X]
add   word ptr [PONT_X],ax
.2_Fuggo: loop .1_Fuggo

ret

;Vízszintes

.1_Vizsz: mov cx,word ptr [DELTA_X] ;Hasonlóan mint a függőleges
mov    dx,cx                       ;esetben történt, itt is
inc    cx                           ;azok a műveletek hajtódnak
shr    dx,1                           ;végre, csak az ellenkező

.2_Vizsz: push dx                    ;koordinátákkal.
call   Pont
pop    dx
mov    ax,word ptr [SGN_X]
add    word ptr [PONT_X],ax
add    dx,word ptr [DELTA_Y]
cmp    dx,word ptr [DELTA_X]
jc     .3_Vizsz

```

IBM PC Gyakorlati Assembly

```

sub dx,word ptr [DELTA_X]
mov ax,word ptr [SGN_Y]
add word ptr [PONT_Y],ax
.3_Vizsz: loop .2_Vizsz

```

```
ret
```

Line **Endp**

Pont **Proc**

```

mov di,word ptr [PONT_X] ;A pont koordinátáiból ki
mov ax,word ptr [PONT_Y] ;kell számolni annak címét
mov bx,320                ;a képernyőn, amit a szokott
mul bx                    ;módon egy szorzással és egy
add di,ax                 ;összeadással teszünk.
mov al,byte ptr [COLOR]  ;Majd a megadott színnel
mov es:[di],al           ;kiradjuk a pontot a megfelelő
ret                       ;helyre.

```

Pont **Endp**

```

PONT_X: dw ? ;A kirakandó pont xy
PONT_Y: dw ? ;koordinátája.

Xa:     dw 50 ;A vonal kezdőpontjának
Ya:     dw 50 ;koordinátái.

Xb:     dw 200 ;A vonal végpontjának
Yb:     dw 100 ;koordinátái.

DELTA_X: dw ? ;A vonal vízszintes mérete.
DELTA_Y: dw ? ;A vonal függőleges mérete.

SGN_X:  dw ? ;A vízszintes lépésköz és
           ;iránya.

SGN_Y:  dw ? ;A függőleges lépésköz és
           ;iránya.

COLOR:  db 7 ;A vonal színe.

Vga6    Ends ;A szegmens vége.
        End Start ;A program vége.

```


A program a szokásos beállításokkal kezdődik. Ezután következik a vonalrajzoló rutin indítása, majd billentyűvárás, paraméterek visszaállítása és visszatérés a DOS-hoz. A vonalrajzoló azért lett eljárásba téve, hogy könnyedén felhasználható legyen más rutinokban is. Azonban a saját programban való felhasználás előtt nem árt némi változtatás. Célszerű kihagyni a pontkirakó rutint, és a számító rutinoknál nem a koordinátát hanem a memóriacímet változtatni. Továbbá érdemes a lehetőségekhez képest minél kevesebb memóriaváltozót használni és amit csak lehet regiszterekkel megoldani ugyanis így nagy mértékben gyorsíthatjuk a programunk működését.

A rutin első lépésben a vonal kezdő- és végpontjának vízszintes koordinátái vonja ki egymásból. Ezzel megtudva a két pont távolságát és egymáshoz képesti elhelyezkedését. Amennyiben a kivonás elvégzése során carry flag értéke 1 állapotba áll, az azt jelenti, hogy a végpont x koordinátája kisebb mint a kezdőponté tehát majdan nem növelni kell értékét hanem csökkenteni, ezt úgy oldottam meg, hogy a koordinátához hozzáadok 65535-öt, aminek ugyan az a hatása, mintha kivontam volna értékéből egyet. Ezért a dx regiszter előzőleg 1-re állított értékét megváltoztatom a már említettre. Ezenkívül el kell végezni egy kettes komplement képzést is mivel a kivonás eredménye nem a két pont abszolút távolsága, hanem annak komplemente ezért azt negálni kell. Ezután már a valódi távolságot fogja mutatni a regiszter. Ha elvégeztük a kívánt műveleteket, az eredményüket memóriaváltozóba írjuk majd megismételjük a leírtakat a függőleges koordinátákkal is. Ezzel a vonal kirajzolásához minden szükséges előkészítő műveletet elvégeztünk. A továbbiakban már csak az a dolgunk, hogy a vonal x illetve y összetevőjétől függően eldöntsük, hogy melyik rutint indítsuk el.

Az algoritmus első lépése, hogy beállítja a számlálóját a nagyobbik xy érték felére. Ezt azért teszi mert a rutin működéséből adódóan ha nulláról indítanánk azt, akkor a húzott vonalban a töréspontok nem a megfelelő helyen lesznek, pl, ha a 0,0 pontból a 100,0 pontba húznánk egy vonalat, a törés nem az 50. pontnál lenne hanem valahol a 99. környékén, de ha a számláló kezdőértéke nem nulla, hanem az említett érték, akkor a prob-

léma nem jelentkezik. Második lépésben meghatározza a ciklus hosszát. Ezt úgy teszi, hogy veszi a nagyobbik összetevőt, és megnöveli eggyel, amit azért tesz mert ha a 0,0 pontból az 1,1 pontba húzunk egy vonalat, akkor a két pont közötti távolság 1, de nekünk két pontot kell kitenni.

Ezt követi a pont kirakása, ami a már ismert módon történik, ezért itt külön nem térek rá ki.

Ha ez megtörtént, veszi a hosszirányú lépésközt (1; 65535) amit a hosszirányú koordinátához ad. Ezt minden esetben meg kell tenni. Majd a számlálóhoz hozzáadjuk a kisebbik összetevő értékét, és összehasonlítjuk a nagyobbik összetevő értékével. Ha annál kisebb vagy egyenlő, átugorja a következő pár utasítást és újabb hosszirányú lépést tesz. Amennyiben értéke már nagyobb, akkor a számlálóból ki kell vonni a nagyobbik xy értéket, hogy újból növelni lehessen, majd a oldalirányú lépésköz értékét a oldalirányú koordinátához adjuk és amíg a ciklus el nem fogy, ismétli ezeket a műveleteket. Aki úgy gondolja, papíron kipróbálhatja az algoritmus működését.

A következő hasznos téma, amivel foglalkozni kívánok, az a képernyő egy ablakának mozgatása valamely irányban. Az ilyen programokat hívják **scroll** rutinoknak. Ezek általában egy pixelponttal jobbra, balra, fölfelé vagy lefelé mozgatják az ablakon belüli képet. Ennek többféle megoldása létezik, az egyik, amikor az ablak szélén kicsúszó adat elvész és a másik oldalról üres képpontok jönnek be, vagy a kép "végtelenítve van" azaz, ami kiment az a másik oldalról bejön, vagy lehet még például olyan, amit főként látványosabb képernyőtörlésre szoktak használni, hogy több részletben mint egy redőny magába fordul a kinnlévő kép. Nos ezek mindegyikére láthatunk majd egy oldal illetve egy függőleges irányú megoldást, de előbb a működésükről egy pár szó: tulajdonképpen mindegyik azon alapszik, hogy a képernyő-memória egy részét máshová másoljuk át. Hogy hová, az a scroll rutintól függ. például egy vízszintes scrollnál amikor nincs szükségünk a kicsorduló részre, akkor egyszerűen egy képponttal arrébb másoljuk a kije-

lött területet. Vigyázni kell azonban arra, hogy nem mindegy hogyan másolunk. Célszerű a onnan kezdeni, ahol kiesnek a pontok, és első lépésben a legszélső helyére másolni az eggyel beljebbit majd a kettővel beljebbit az előző helyére és így tovább. Fellép itt is az a probléma, hogy mekkora területet tudunk egy képváltás alatt a képernyőn arrébb mozgatni, ugyanis itt már nagyon csúnyán előjönnek a törések, ha nem időzítjük megfelelően a mozgatást. A függőleges irányú rutinoknál azonos a helyzet. Akkor, ha azt szeretnénk, hogy körbe forogjon az ábra, akkor két lehetőségünk van, az egyik, hogy eltávolítjuk a kieső képpontokat valahová a memóriába és a mozgatás végén kirakjuk az első pont helyére, vagy ami egy kicsit lassabb, de nem szükséges hozzá háttérmemória, az a képpont cserélgetős eljárás, amikor szintén a végéről indulva megcseréljük a két utolsó pontot egymással majd egy ponttal beljebb lépünk mindaddig, míg el nem érjük a kijelölt terület végét. Így a cserélgetések hatására az utolsó pont az első helyére vándorol. A redőny vagy más néven fodros scrolloknál az első rutinhoz hasonlóan járunk el, mindössze annyi különbséggel, hogy nem az egész képet mozgatjuk egyszerre, hanem sok különálló részre bontva például 8 pixelenként mozgatjuk, így érjük el azt a hatást mintha egy redőny mögé csúszna a kép.

[Program 27]

Vga7	Segment assume cs:Vga7,ds:Vga7	;Szegmensdefiníció ;Cs,ds hozzárendelése
Start:	mov ax,Vga7 mov ds,ax	;Ds regiszter beállítása a ;program elejére.
	mov ax,13h int 10h	;320*200/256 üzemmód ;bekapcsolása
	mov ax,0a000h mov es,ax	;A képernyő-memória kezdőcímet ;es regiszterbe tölti
	xor ax,ax	;A színek kezdőértéke.
	mov cx,16	;Függőleges sorok száma/2
	mov di,92*320+144	;Az ábra kezdőcíme.

IBM PC Gyakorlati Assembly

.1_Vga7:	push cx mov cx,16	;Az egymásba ágyazott ciklus ;miatt el kell menteni cx ;értékét. ;A vízszintes pontok száma/4
.2_Vga7:	mov es:[di],ax mov es:320[di],ax mov es:2[di],ax mov es:322[di],ax inc al inc ah add di,4 loop .2_Vga7 add di,576 pop cx loop .1_Vga7 mov cx,128	;A színek kirakása. ;Következő szín ;Következő képpont. ;Egy új sorra ugrik. ;A mozgatások száma.
.3_Vga7:	push cx call Korbe_Fel call Korbe_Bal call Szunet pop cx loop .3_Vga7 xor ax,ax int 16h mov ax,3 int 10h mov ax,4c00h int 21h	;A scroll rutin(ok) hívása ;A sebesség csökkentése. ;A ciklus ismétlése, míg cx ;nem nulla. ;Billentyűvárás. ;80*25 karakteres mód ;beállítása. ;Visszatérés az operációs ;rendszerhez.
Korbe_Bal	Proc call Vert_Bl	;Az eljárás kezdete. ;Időzítés hívása.

Grafika programozása Assembly nyelven

```
mov di,92*320+144      ;Az ablak kezdőcíme.

mov cx,32              ;A sorok száma.

.1_KB:  push cx
        mov cx,63      ;A vízszintes méret-1

.2_KB:  mov ax,es:[di]  ;Az adott pozíciókban levő
        xchg al,ah     ;képpontok felcserélése.
        mov es:[di],ax

        inc di         ;Következő képpont.
        loop .2_KB

        pop cx         ;A következő sorra lép.
        add di,257
        loop .1_KB

        ret           ;Visszatérés a hívó programhoz

Korbe_Bal Endp        ;Az eljárás vége.

Korbe_Fel Proc

        call Vert_BI

        mov di,92*320+144
        mov cx,31

.1_KF:  push cx
        mov cx,32

.2_KF:  mov ax,es:[di]  ;Mivel függőlegesen mozgatjuk
        mov bx,es:320[di] ;az alakzatot és páros számú
        mov es:[di],bx  ;a szélessége, lehet egyszerre
        mov es:320[di],ax ;egy wordöt mozgatni.

        add di,2
        loop .2_KF

        pop cx
        add di,256
        loop .1_KF

        ret
```

Korbe_Fel Endp

Mozgat_Bal Proc

```

        call  Vert_BI           ;Időzítés.

        mov  di,92*320+144     ;Az ablak címe.

        mov  cx,32             ;A sorok száma.

.1_MB:  push  cx
        mov  cx,63             ;A vízszintes képpontok-1

.2_MB:  mov  al,es:1[di]        ;Az adott képpont átmásolása
        mov  es:[di],al        ;a mellette levő pozícióba.

        inc  di                 ;Következő képpont.
        loop .2_MB

        xor  al,al              ;Az utoljára mozgatott pont
        mov  es:[di],al        ;helyere nullat írva töröljük azt.

        pop  cx                 ;Egy új sor mozgatása.
        add  di,257
        loop .1_MB

        ret                     ;Vissza a hívó programhoz.
    
```

Mozgat_Bal Endp

Mozgat_Fel Proc

```

        call  Vert_BI

        mov  di,92*320+144
        mov  cx,64

.1_MF:  push  cx
        mov  cx,31

.2_MF:  mov  ax,es:320[di]     ;Hasonlóan mint a forgatónál,
        mov  es:[di],ax       ;itt is lehet wordöt mozgatni.

        add  di,320
    
```

```

loop .2_MF

xor ax,ax
mov es:[di],ax

pop cx
sub di,31*320-2
loop .1_MF

ret

```

Mozgat_Fel Endp

Fodor_Bal Proc

```

call Vert_BI ;Időzítés

mov di,92*320+144 ;Az ablak memóriacíme.

mov cx,32 ;A sorok száma.
.1_FB: push cx

mov cx,8 ;A blokkok száma egy sorban.
.2_FB: push cx

mov cx,7 ;Egy blokk szélessége-1

.3_FB: mov al,es:1[di] ;Hasonlóan mint a mozgatonál,
mov es:[di],al ;a megadott szélességű részt
inc di ;arrébb tolja egy ponttal úgy,
loop .3_FB ;hogyan a szélén levő pont
xor al,al ;törlődik.
mov es:[di],al

inc di ;Ezt megteszi az egész soron.
pop cx
loop .2_FB

pop cx ;Egy új sor mozgatása.
add di,256
loop .1_FB

ret

```

Fodor_Bal Endp

```

Fodor_Fel Proc

    call Vert_Bl
    mov  di,92*320+144
    mov  cx,64

.1_FF:  push  cx
        mov  cx,4

.2_FF:  push  cx
        mov  cx,7

.3_FF:  mov  ax,es:320[di]      ;Hasonlóan a többihez, wordös
        mov  es:[di],ax     ;kezelés.
        add  di,320
        loop .3_FF

        xor  ax,ax
        mov  es:[di],ax
        add  di,320
        pop  cx
        loop .2_FF

        pop  cx
        sub  di,32*320-2
        loop .1_FF

    ret

Fodor_Fel Endp

Szünet Proc

        push  cx              ;Egy 1fffh hosszú ciklus.
        mov  cx,1fffh        ;anélkül, hogy a cx értéke
.1_SzUNET: loop .1_SzUNET    ;megváltozna a rutinból
        pop  cx              ;való visszatéréskor.

    ret

SzUNET Endp

Vert_Bl Proc

```



```

mov dx,3dah ;A rutin addig vár, míg egy
.1_VB: in al,dx ;előlről kezdődő visszafutást
test al,8 ;nem érzékel.
jnz .1_VB
.2_VB: in al,dx
test al,8
jz .2_VB

ret

Vert_Bl Endp

Vga7 Ends ;A szegmens vége
End Start ;A program vége.

```

A programok működése könnyedén megérthető, ha elemeire bontva vizsgáljuk. A program elején semmi új dolog nincs, beállítja a szokott dolgokat. Ezután csinál egy 16*16-os ciklust, ami segítségével kirakunk egy színpalettát ami később az ábrát fogja helyettesíteni. Ha ezzel megvagyunk, jöhet a mozgatás. Az egyes rutinok úgy lettek megírva, hogy egy hívásra egy pontnyit mozgatnak, ezért ha ennél többet kívánunk mozgatni, akkor egy ciklusba kell szervezni azt. Vigyázni kell azonban arra, hogy a programok használják a cx regisztert, úgyhogy a call utasítás előtt el kell menteni a verembe az értékét. A minta alapbeállítása kétféle körbeforgatást végez, ezáltal a szöveg balra-fel mozog. De ezt természetesen meg lehet változtatni, hogy a többi rutint is ki tudjuk próbálni. Az eljárások hívása utáni sor egy várakozást iktat be, mégpedig egy 1fffh hosszú holtciklus segítségével. Az ismétlési ciklus leteltével egy billentyűvárás és egy képernyő visszaállítás következik majd visszatérés az operációs rendszerhez. Nos akkor nézzük az egyes rutinok működését:

Minden forgatás előtt egy Vertical Blank időzítés lett beiktatva. Ezután következik a mozgatandó ablak bal felső sarkának meghatározása. Ezek az összes rutinban megtalálhatók.

Ahhoz, hogy egy 64*32 képpont méretű alakzatot mozgatni tudjunk, egy függőleges és egy ebbe ágyazott vízszintes ciklusra van szükségünk (vagy fordítva, de akkor megváltozik a

műveletek sorrendje). A vízszintes érték megadásánál a szélesség mínusz egyet kell megadni (vízszintes scrollnál) mert különben eggyel több pontot mozgatna. Az első rutinnál a már említett cserélgető eljárás lett alkalmazva úgy, hogy a képernyő-memóriából kiolvas egy wordöt, megcseréli a regiszter alsó-felső byte-ját majd visszairja. Ezzel két egymás melletti pontot cserélünk meg. Ahhoz, hogy az egész ábra mozogjon, növelni kell eggyel a vízszintes koordináta értékét és végrehajtani az említett műveletet egészen a sor végéig majd megismételni az összes sorra, ezért van szükség a két egymásba ágyazott ciklusra.

• Ha a forgatás iránya nem vízszintes, hanem függőleges, akkor már nem célravezető a regiszterek cserélgetése mivel a visszairás helyét is meg tudjuk változtatni. Továbbá, hogy egy kicsivel gyorsabb legyen a rutinunk működése, érdemes nem byte-os adatokkal dolgozni hanem wordössel.

Itt a két egymásba ágyazott ciklusnál a külső a magasság-1 és a belső ciklus a szélesség/2 a word miatt. A mozgatás úgy történik, hogy betölti **ax** regiszterbe az **index** regiszter által mutatott értéket és **bx**-be az alatta lévő két pontot (**di+320**) majd az első címre írja vissza a **bx**-et és a másodikra az **ax**-et. Ezzel megcserélődött a két képpont. Hasonlóan mint az előbb, végrehajtuk a műveletet az egész sorra majd eggyel lejjebb lépve megismételjük azt mindaddig, míg el nem érjük a ciklus végét.

Akkor, ha nem akarjuk, hogy az adatok körbeforduljanak, egy egyszerű másolást kell végeznünk. Ezt valósítja meg a **Mozgat** nevű rutin. Természetesen hasonlóképpen kezdődik mint elődei, a különbség ott kezdődik, hogy a mozgatás egy olvasásból és egy új helyre történő írásból áll. Majd ha lefutott a belső ciklus, az éppen aktuális helyről törli a pontot (ez mindig az utolsó pont helye). Így valósítja meg, hogy üres legyen az a rész, ahonnan lecsúszik az alakzat. Amire vigyázni kell ilyen rutinoknál az az, hogy pontosan a szélétől induljunk és vigyázzunk arra, hogy ne írjunk az alakzaton kívülre (ne tévesszük el a határait). A másik fontos dolog, hogy a másolást mindig úgy végezzük, hogy választunk egy haladási irányt ami a mozgatás irányával mindig ellentétes, ezután vesszük az adott pontot és hátra

felé másoljuk. Ennek lényege, hogy ha előre felé másolnánk, akkor a következő képpont amit mozgatnunk kéne, az az általunk odamásolt pont lenne. Így a legelső pont színével teleírnánk az adott sort. Ugyanez igaz a függőleges irányú mozgatásra is. A különbség ott annyi, hogy mint az előbbi függőleges rutinnál, itt is megengedhetjük a wordös adatok használatát.

A következő scroll rutinok tulajdonképpen nem hoznak semmi újdonságot, mindössze látványosak. Ugyanis itt csak arról van szó, hogy az előbbi mozgató eljárást nem az egész alakzatra alkalmazzuk, hanem csak egy 8×8 -as négyzetre. Így ez a kis alakzat fog ugyan úgy viselkedni mint az előbb a nagy. Ha nem csak egy ilyen objektumra hajtjuk végre a műveletet, hanem az egész alakzatot felbontjuk kis négyzetekre és mindegyiken elvégezzük azt, akkor lesz ez az érdekes jelenség.

A rutint egyébként sokkal hatékonyabban lehet használni bittérképes (2 színű) üzemmódokban mivel ott elegendő kiolvasni egy byte-ot, egy `shl shr` utasítással eltolni majd visszaírni. Így a rutin lefutása is sokkal gyorsabb lenne.

Az egyes rutinoknál más és más beállítás lenne szükséges, úgyhogy célszerű egy új eljárás előtt a várakozási időt és az ismétlések számát beállítani mivel például egy Fodor rutin lefutásához 8 lépés szükséges mindössze és viszonylag nagyobb várakozási idő, hogy látni is lehessen valamit.

A következő elég gyakran előforduló grafikai feladat a kör rajzolása. Itt hasonló nehézségek lépnek fel, mint az egyenes húzásánál. A feladatot megoldhatnánk a köregyenlettel, vagy abból kiindulva, hogy a szinusz és a koszinusz függvények a körből lettek alkotva, így a kör pontjait úgy is megkaphatjuk, hogy egy $0-360$ -ig tartó ciklus (x) segítségével a körhöz tartozó pontok egyik koordinátája $r \cdot \sin(x)$ a másik pedig $r \cdot \cos(x)$ lesz. Az eddig említett megoldások hátránya, hogy alapvetően nem ártana hozzájuk egy lebegőpontos aritmetika amivel a tört számokat is tudjuk kezelni, valamint az, hogy egy 10 sugarú kört is 360 pontból és egy 100 sugarú kört is 360 pontból épít fel. Így adódhat olyan eset, amikor a kör fala $1-2$ pont vastag illetve olyan is, amikor a pontok között rés van. A megoldás

hasonló mint az egyenes rajzoló rutinnál. Egy olyan algoritmus, ami nem számol csak egész számokkal valamint mindig a megfelelő számú pontot teszi ki a képernyőre. Az itt használt rutin kihasználja a kör azon tulajdonságát, hogy könnyedén felbontható nyolc egyforma részre. Így tulajdonképpen elegendő csak egy nyolcad kört rajzolni és a maradékot tükrözéssel ábrázolni. Az algoritmus működése a következőképpen néz ki.

C_x = a középpont X koordinátája

C_y = a középpont Y koordinátája

r = a kör sugara

D_x és D_y a Delta X illetve a Delta Y a középponttól való távolság, amit a program számol ki.

$D_x=r$

$D_y=0$

Számító: Ha $r > D_y^2$ akkor ugrás a pontkirakásra.

Különben: $D_x=D_x-1$

$r=2 * D_x+1$

Pontkirakó: Pont C_x+D_x, C_y+D_y

Pont C_x-D_x, C_y+D_y

Pont C_x+D_x, C_y-D_y

Pont C_x-D_x, C_y-D_y

Pont C_x+D_y, C_y+D_x

Pont C_x-D_y, C_y+D_x

Pont C_x+D_y, C_y-D_x

Pont C_x-D_y, C_y-D_x

$D_y=D_y+1$

Ha $D_y < D_x$ akkor vissza a számításhoz.

Különben vége.

Aki járatos a magasabb szintű nyelvekben, az könnyedén megérti az előbb említetteket. Ami talán a leginkább feltűnő, az a nyolc pontkirakás. Ez végzi a tükrözést, hogy egy teljes kört

kapjunk. Aki úgy gondolja, szintén elemezheti a rutin működését papíron. Nos ezek után a program:

[Program 28]

Vga8	Segment assume cs:Vga8,ds:Vga8	;Szegmensdefiníció ;Cs, ds hozzárendelése ;a szegmensregiszterekhez.
Start:	mov ax,Vga8 mov ds,ax mov ax,0a000h mov es,ax mov ax,13h int 10h mov dx,word ptr [SUGAR] xor cx,cx	;Ds beállítása a kód elejére. ;Es beállítása a képernyő- ;memória szegmenscímére. ;320*200/256 felbontás ;beállítása. ;Beállítjuk a szükséges kezdő ;értékeket (Dx=r , Dy=0).
.1_Vga8:	mov ax,cx mov bx,cx push dx mul bx pop dx cmp word ptr [SUGAR],ax ja .2_Vga8 dec dx mov bx,dx shl bx,1 inc bx add word ptr [SUGAR],bx	;Előállítjuk a Dy*Dy értékét. ;És ezt összehasonlítjuk a ;sugár értékével. ;Amennyiben a sugár értéke ;kisebb vagy egyenlő az előbbi ;értékkel, akkor atugorja ;a következő részt és a ;pontkirakással folytatja a ;rutin végrehajtását. ;A Dx=Dx-1 és az r=r+2*Dx+1 ;műveletek végrehajtása.
.2_Vga8:	mov ax,word ptr [KOORD_X] add ax,dx mov word ptr [PONT_X],ax	;1. pont kirakása.

```

mov ax,word ptr [KOORD_Y]
add ax,cx
mov word ptr [PONT_Y],ax
call Pont

```

```

mov ax,word ptr [KOORD_X] ;2. pont kirakása.
sub ax,dx
mov word ptr [PONT_X],ax
mov ax,word ptr [KOORD_Y]
add ax,cx
mov word ptr [PONT_Y],ax
call Pont

```

```

mov ax,word ptr [KOORD_X] ;3. pont kirakása.
add ax,dx
mov word ptr [PONT_X],ax
mov ax,word ptr [KOORD_Y]
sub ax,cx
mov word ptr [PONT_Y],ax
call Pont

```

```

mov ax,word ptr [KOORD_X] ;4. pont kirakása.
sub ax,dx
mov word ptr [PONT_X],ax
mov ax,word ptr [KOORD_Y]
sub ax,cx
mov word ptr [PONT_Y],ax
call Pont

```

```

mov ax,word ptr [KOORD_X] ;5. pont kirakása.
add ax,cx
mov word ptr [PONT_X],ax
mov ax,word ptr [KOORD_Y]
add ax,dx
mov word ptr [PONT_Y],ax
call Pont

```

```

mov ax,word ptr [KOORD_X] ;6. pont kirakása.
sub ax,cx
mov word ptr [PONT_X],ax
mov ax,word ptr [KOORD_Y]
add ax,dx
mov word ptr [PONT_Y],ax
call Pont

```

```

mov ax,word ptr [KOORD_X] ;7. pont kirakása.
add ax,cx
mov word ptr [PONT_X],ax
mov ax,word ptr [KOORD_Y]
sub ax,dx
mov word ptr [PONT_Y],ax
call Pont

mov ax,word ptr [KOORD_X] ;8. pont kirakása.
sub ax,cx
mov word ptr [PONT_X],ax
mov ax,word ptr [KOORD_Y]
sub ax,dx
mov word ptr [PONT_Y],ax
call Pont

inc cx ;A Dy=Dy-1 végrehajtása.
cmp cx,dx ;Ha Dx<Dy akkor vége.
jnc .3_Vga8
jmp .1_Vga8 ;Különben újabb pont.

.3_Vga8: xor ax,ax ;Billentyűvárás.
int 16h

mov ax,3 ;Szöveges képernyő.
int 10h

mov ax,4c00h ;Visszatérés az operációs
int 21h ;rendszerhez.

Pont Proc

push bx ax dx
mov di,word ptr [PONT_X] ;A pont koordinátáiból ki
mov ax,word ptr [PONT_Y] ;kell számolni annak címét
mov bx,320 ;a képernyőn, amit a szokott
mul bx ;módon egy szorzással és egy
add di,ax ;összeadással teszünk.
mov al,byte ptr [COLOR] ;Majd a megadott színnel
mov es:[di],al ;kiradjuk a pontot a megfelelő
;helyre.

pop dx ax bx
ret

```

Pont	Endp	
PONT_X:	dw	?
PONT_Y:	dw	?
COLOR:	db	7
SUGAR:	dw	50
KOORD_X:	dw	160
KOORD_Y:	dw	100
Vga8	Ends	
	End	Start

;A kirakandó pont xy
;koordinátája.
;A vonal színe.
;A kor sugara,
;X koordinátája,
;Y koordinátája.

;A szegmens vége.
;A program vége.

A programban a szükséges üzemmód, stb. beállítások után következik Dx és Dy kiinduló értékének beállítása. A következő lépés az összehasonlítási művelet. Ahhoz, hogy ezt el tudjuk végezni, előbb elő kell állítani a $Dy \cdot Dy$ értékét egy regiszterbe és csak ezután tudjuk összehasonlítani a sugár értékével. A négyzetre emelést úgy végezzük el, hogy ax és bx regiszterbe is a Dy értékét tesszük, majd összeszorozzuk őket. Ha az összehasonlítás során a sugár kisebb-egyenlőnek bizonyult, csökkentjük Dx értékét majd a sugárhoz adjuk a $sugár + 2 \cdot dx + 1$ értéket. Lényegében ez az egész eljárás lelke, ugyanis ez határozza meg, hogy mikor kell oldalirányú lépést tenni a rajzolás során. A pontok kirakását a vonalrajzoló rutinnál használt eljárás végzi. Hívása előtt beállítjuk a megfelelő koordinátákat attól függően, hogy melyik körcikket rajzoljuk éppen. Ezután következik egy függőleges lépés és ha a $Dy > Dx$ akkor megismétli a rutint. Hátránya a programnak, hogy megváltoztatja a sugár kiinduló értékét, de ezen könnyedén lehet segíteni egy átmeneti tároló beiktatásával. További szépséghibája a dolognak, hogy a $320 \cdot 200$ -as felbontásnál nem kört hanem ellipszist rajzol, aminek az az oka, hogy ebben a felbontásban torzít a megjelenítés. Ezen úgy lehet segíteni, hogy például $640 \cdot 480$ -as felbontáson működtetjük a rutint ami már pontos kört eredményez.

Képfarmátumok:

A grafikus képek tárolása a lemezen többféleképpen történhet. A legegyszerűbb, amit a **Paint Brush** nevű windows alatti rajzoló program is támogat, a bittérképes tárolás, ami azt jelenti, hogy a kép minden pontja egyenként le van mentve. Ha figyelembe vesszük, hogy például egy 1024*768/256 színű grafika ilyen módon történő tárolásához 768Kbyte plusz a fejléc és a színpaletta hossza szükséges, ami már nem csekély méret. Ezért ezt csak egészen apró ábrák tárolására érdemes használni vagy egy jó nagy winchestert kell beépíteni a gépbe (de ez egy kicsit költséges). Erre a problémára lettek kitalálva a különféle tömörítési eljárások, minek hatására a file mérete a töredékére változik. Nos elég sok tömörítési eljárást kidolgoztak az évek során, az egyik jobb mint a másik. Az egyik legtömörebb talán a .GIF formátum, de ennek kitömörítése nem a legegyszerűbb. Ezért egy kis kompromisszum árán arra jutottam, hogy a bizonyára sokak számára ismerős **De Luxe Paint** rajzolóprogram képfarmátumát fogom ismertetni. Ennek oka, hogy aránylag jó tömörítési arányt lehet vele elérni és a visszafejtése sem okoz nagy fejtörést.

Az eljárás lényege, hogy az egymás mellett lévő egyforma byte-okat letömöríti oly módon, hogy tárolja, hogy hány darab és hogy milyen értékű adat volt az adott helyen. Természetesen meg kell különböztetni a tömörített és a tömörítetlen adatokat egymástól, amit úgy oldottak meg, hogy a mennyiséget tároló byte-ot negálták, így 7. bitje mindig 1 értékű. Ezen módszerrel egy csomagban maximum 128 Byte-ot tárolhatunk mivel a vezérlő byte 0-127-ig tömörítetlen adatokat jelent, ez fölött pedig tömörítettet. A pontos darabszámot úgy kapjuk, ha ezt a byte-ot negáljuk (kettes komplementjét vesszük) majd eggyel megnöveljük az értékét.

Ha olyan byte-ot találunk, amelyeknek a 7. bitje 0, az azt jelenti, hogy utána az adott byte értéke+1 darab tömörítetlen adatot találunk. Ezt egyszerűen kimásoljuk innen, és a blokk utáni byte-ról ismét eldöntjük, hogy tömörített vagy normál adatok jönnek utána. De hogy egy kicsit egyszerűbb legyen megérteni a működését, egy példa:

A memóriában lévő adat: ... 81 23 04 45 F2 A6 75 EA FC
00 02 36 AF 98 E4 FF ...

Ezen adat kitömörítése a következőképpen történne: Mivel az első adat 81h aminek a 7. bitje 1, ezért ezt negálni kell és utána megnövelni eggyel. Így eredményként 80h-t kapunk. Ezután szükségünk van a következő adatra ami a 23h és a megfelelő helyre (ahová ki akarjuk tömöríteni a képet) beírunk 80h darab 23h-t. Ezután vesszük a következő értéket, ami itt a 04h. Ez azt jelenti, hogy a soron következő 04h+1 azaz 5 darab byte nincs tömörítve. Tehát a 128 darab 35 után letároljuk egymás után a 45h, F2h, A6h, 75h, EAh értékeket. A következő adat az FCh jelentése: **neg FCh+1** darabot kell a következő byte-ból írni a memóriába, tehát 5 darab 0-át kell írni az előző adatok után. Ezután következik egy 02 ami mutatja, hogy a 36h, AFh, 98h adatokat kimásolhatjuk innen mivel nem tömörített adatok. És végül az E4 FF jelentése, hogy 29 darab 255-öt kell írni az előző adatok után.

Ezzel a módszerrel képeknél viszonylag jó tömörítési arányt lehet elérni. Azonban az eljárás ismerete nem elegendő egy kép kirakásához. Ugyanis a rajzoló program tárol még egy pár hasznos információt a file-ban. Köztük a színpalettát, az adott rajz méretét ,hogy milyen felbontásban készült és egyéb más adatokat.

Az minden adatsoportot egy címke jelöl, ami után egy négy byte-os méret található. Ez határozza meg az adott címke hosszát. Különbség a PC-s adattároláshoz képest, hogy a hossz egyes helyiértékei nem fordított sorrendben helyezkednek el, hanem úgy, ahogyan leírjuk azt. Az egyes címek jelentését és tartalmukat a részletesség igénye nélkül a következő táblázat tartalmazza:

Cím	Hossz	Tartalom/Jelentés
00	4	<i>FORM</i> formátum címke.
04	4	A file hossza ezt követően.
08	4	A formátum neve.

0C	4	<i>BMHD</i> bittérkép fejléc címke.
10	4	<i>BMHD</i> blokk hossza.
14	2	A kép szélessége.
16	2	A kép magassága.
20	2	A színek száma az üzemmódban.
22	2	A képtorzítás aránya.
24	4	A készítésnél használt üzemmód X és Y irányú felbontása.
28	4	<i>CMAP</i> címke, a palettaszínek tárolási helyét jelöli.
2C	4	A <i>CMAP</i> blokk hossza.
30	?	R-G-B színösszetevők.
?	4	<i>DPPS</i> címke. A perspektívát tárolja.
DPPS+4	4	<i>DPPS</i> blokk hossza.
?	4	CRNG címke, a DP-nél használt színátmeneteket tárolja itt.
CRNG+4	4	A CRNG blokk hossza.
?	4	<i>TINY</i> címke, a DP-be történő kép betöltése előtt kirajzolódó kicsinyített képet tárolja itt.
TINY+4	4	A <i>TINY</i> blokk hossza.
?	4	<i>BODY</i> címke, a kép kezdetét jelöli.
BODY+4	4	A <i>BODY</i> blokk hossza, azaz a kép tömörített mérete.

A kitömörítés során figyelembe kell venni, hogy a színeket 24 bites színpalettát feltételezve tárolja a gép, de mivel az alsó két bitet nem használja, egyszerűen arrébb kell tolni az értéket két bittel és máris megkapjuk a helyes értéket.

A most következő programban a felsorolt adatok közül nem sokat fogunk használni, mivel tudjuk, hogy 320*200/256 üzemmódban készült képet szeretnénk kirakni azonos módban és méretben, nincs szükségünk az adatok többségére.

[Program 29]

```
Vga9      Segment      ;Szegmensdefiníció.
          assume cs:Vga9,ds:Vga9      ;Cs,ds beállítása.
```

IBM PC Gyakorlati Assembly

```

Start:   mov  ax,Vga9           ;Ds regiszter beállítása a
        mov  ds,ax           ;kód elejére.

        mov  ax,4a00h        ;A program által lefoglalt
        mov  bx,1024         ;memóriaterületet 16K
        int  21h             ;méretüre változtatjuk.

        mov  ax,4800h        ;Kérünk a DOS-tól egy
        mov  bx,4096         ;64K méretű memórialapot.
        int  21h

        mov  word ptr [MEMOCIM],ax;Ennek szegmenscímét a
                                ;memóriaváltozóba tesszük.

        mov  ax,3d00h        ;A FILENEV alatt tárolt nevű
        mov  dx,offset FILENEV ;file megnyitása olvasásra.
        int  21h
        mov  word ptr [FILESZAM],ax;A visszakapott file azonosító
                                ;információt a memóriába
                                ;mentjük.

        jnc  Tovabb
        jmp  Hiba

Tovabb: push  ds              ;A ds regiszter értékét
        mov  bx,ax            ;elmentjük, mivel mi nem
        mov  ax,word ptr [MEMOCIM];a ds által mutatott
        mov  ds,ax            ;szegmensre kívánunk betölteni
        mov  ax,3f00h         ;hanem az általunk lefoglalt-
        xor  dx,dx            ;ra. A DOS hátránya viszont,
        mov  cx,0ffffh        ;hogy a betöltendő file
        int  21h             ;helyét a ds:dx regiszterben
                                ;kell megadni a rutinnak.

        pop  ds              ;A ds visszaállítása.

        mov  ax,3e00h         ;A file lezárása.
        mov  bx,word ptr [FILESZAM]
        int  21h

        mov  ax,13h          ;320*200/256 grafikus üzemmód
        int  10h             ;beállítása.

        mov  dx,3c8h         ;A színpaletta kinullázása
        mov  cx,256
        xor  ax,ax
    
```

```

.1_Torles:  out  dx,al
              inc  dx
              push ax
              xor  ax,ax

              out  dx,al           ;R-G-B nullázása.
              out  dx,al
              out  dx,al

              pop  ax
              dec  dx
              inc  al
              loop .1_Torles

              mov  al,byte ptr [CIMKE] ;A BODY címke megkeresése.
              mov  es,word ptr [MEMOCIM]
              mov  di,6
              mov  cx,es:[di]
              xchg cl,ch

.1_Vga9:   repnz scasb
              jnz  Hiba

              mov  si,offset CIMKE
              push di cx
              dec  di
              mov  cx,4
              repz cmpsb
              pop  cx di
              jnz  .1_Vga9

              mov  dx,es:[di+5]      ;A tömörített kép hosszát
              xchg dl,dh              ;a dx regiszterbe teszi.

              add  di,7               ;Az első adatbyte-ra állítja
              ;a mutatót.

              push ds                 ;Az adatszemens elmentése.

              mov  ds,word ptr [MEMOCIM];Ds-be a tömörített kép
              ;kezdőcíme lesz.
              mov  ax,0a000h         ;Es-be a videomemória
              ;szegmenscímét tölti.
              mov  es,ax
              ;Si-be a tömörített kép
              ;kezdőcímét teszi.
    
```

	xor	di,di	;Regiszterek nullázása.
	xor	ch,ch	
.2_Vga9:	and	dx,dx	;Ha dx=0 azaz a kép kirakása
	jz	Szin	;a végére ért, ugrik a színek
			;beállítására.
	mov	cl,[si]	;Cl regiszterbe teszi a kép
	inc	si	;egy byte-ját, növeli si-t
	dec	dx	;és csökkenti a számlálót.
	test	cl,128	;Ha cl 7. bitje 0 akkor
	jz	Normal	;tömörítetlen adat következik.
;Tomoritett			;Különbön tömörített.
	neg	cl	;Ez esetben negálni kell
	inc	cl	;az értékét és növelni eggyel.
	mov	al,[si]	;Al regiszterbe tölti a
			;kiírandó byte-ot.
	inc	si	;A következő byte.
	dec	dx	;A számláló csökkentése.
	rep	stosb	;Cl darab al kirakása a képre,
	jmp	.2_Vga9	;majd vissza.
Normal:	inc	cl	;Cl+1 adat átmásolása ds:si
	sub	dx,cx	;címről es:di címre és a
	rep	movsb	;számláló ennek megfelelően
	jmp	.2_Vga9	;történi csökkentése majd
			;vissza.
Szin:	mov	si,30h	;A színek most a 30h címtől
			;kezdődően vannak letárolva.
	mov	cx,256	;256*3 byte hosszon.
	mov	al,0	;Az első állítandó paletta
			;a 0. sorszámú.
.1_Szin:	mov	dx,3c8h	;A színpaletta módosítása
	out	dx,al	;a memóriában tárolt adatoknak
	inc	al	;megfelelően.
	push	ax	
	inc	dx	

```

mov  al,[si]                ;A piros összetevő beállítása.
shr  al,1
shr  al,1
out  dx,al

mov  al,[si+1]              ;A zöld összetevő beállítása.
shr  al,1
shr  al,1
out  dx,al

mov  al,[si+2]              ;A kék összetevő beállítása.
shr  al,1
shr  al,1
out  dx,al

add  si,3                   ;A következő szín.
pop  ax
loop .1_Szin

xor  ax,ax                  ;Billentyűvárás.
int  16h

Hiba: mov  ax,3              ;Karakteres mód beállítása.
      int  10h

      mov  ax,4c00h         ;Visszatérés az operációs
      int  21h              ;rendszerhez.

CIMKE: db  "BODY"
FILESZAM:dw ?
MEMOCIM:dw ?
FILENEV: db  "demokep.lbm",0

Vga9  Ends                 ;A szegmens vége.
      End  Start           ;A program vége.

```

A program mindjárt egy már ismertetett rutinnal kezdődik, lefoglal egy 64Kbyte hosszú részt és betölti ide a lemezen található mintaképet. A grafikus üzemmód bekapcsolása után első lépésben az összes palettaszínt feketére állítja. Ezt azért teszi, mert egy kép kitömörítése több időt vesz igénybe, mint egy palettaállítás és ha fekete színekkel kirakjuk a képet, akkor abból semmi nem fog látszani, azonban amikor a színeket is beállítjuk, ami már nagyon rövid idő alatt megoldható, hirtelen megjeleni a

kép. Ezzel azt a hatást érhetjük el, mintha ilyen gyorsan lett volna kicsomagolva a kép. Ezután egy már szintén ismert eljárással megkeressük a *BODY* címkét, ami a kép elejét jelöli. *Dx* regiszterbe tesszük a címkét követő hosszt, amit számlálóléként használva fogjuk tudni, hogy mikor érünk a kép végére.

A kitömörítés két részre lett bontva. Az egyik a tömörített adatokat teszi a képernyőre, a másik pedig a tömörítetlent. Az eljárások a már említett módon működnek, figyelve, hogy a *dx* regiszter értékét mindig megfelelő mértékben csökkentjük.

Elérve a kép végét jöhet a színek beállítása. Itt megtehetnénk, hogy megkeressük a *CMAP* címkét, de mivel ismert a kezdőcíme, ettől eltekintünk és a hosszát sem fogjuk megnézni mivel egy 256 színű képnél ez mindig 300h tehát a jól bevált out utasításos módszerrel beállítjuk mind a 256 szín RGB értékét. Ezután nincs más dolgunk mint várni egy billentyű lenyomásáig és utána visszaállítani az eredeti paramétereket és kilépni a programból.

Egy védelem be lett építve a rutinba, mivel ha a kép betöltése sikertelen lett volna és a program mégis kinullázza a színeket és elkezdi keresni a *BODY* címkét, abból legjobb esetben is egy fekete kép kerülhet ki eredményül, de a rendszer lefagyása sem elképzelhetetlen. Ezért a file megnyitásakor figyeljük a *carry* flag értékét és ha ez 1 akkor billentyűvárás nélkül kilépünk a programból, mivel valamilyen oknál fogva a gép nem tudta megnyitni a *demokep.ibm* nevű file-t.

Ez a rutin csak 256 színű képek kirakására alkalmas, mivel a 16 színű képek úgy tárolódnak, hogy a 0. sor 0. plane-je, az 0. sor 1. pane-je, ..., 0. sor 3. plane-je, 1. sor 0. plane-je, ..., utolsó sor 3. plane-je. Ezért a kitömörítő rutint ennek megfelelően át kell alakítani.

Amennyiben a DP programból nem egy teljes képet mentettünk le, hanem egy brush-t (.BBM) akkor az ugyanígy tárolódik, de a kép méreténél a brush méretei találhatóak. A 16 színű módokra említett dolog itt is igaz.

AZ SVGA ÜZEMMÓDOK TULAJDONSÁGAI

Az IBM gépeknél el lett fogadva egy szabványos képernyő kezelés, amihez minden gyártó cégnek tartania kell magát. Ez a szabvány azonban csak a standard módokra vonatkozik, azaz a 13h üzemmódig. Az ettől följebb eső résszel szabadon garázdálkodhatnak. Így az SVGA kártyák sokban eltérnek egymástól. Ezek első és legfontosabb tulajdonsága, hogy nincsenek szabványosítva, így ahány grafikus kártya, annyi féle lehetőség és tulajdonság. Azért, hogy mégis lehessen valamit hasznosítani ezen szolgáltatásokból, megpróbálok az általam eddig fellelt ismeretanyagot belevinni ebbe a fejezetbe, ami ugyan nem túl sok, de remélem mindenki talál benne hasznosat. Az ismeretek hiányának oka, hogy a gyártó cég nem mellékel a kártya mellé egy részletes leírást, amiben szerepelnének az adott eszköz programozási lehetőségei. Ez részben érthető (üzletpolitika) de ezzel megnehezítik a programozók feladatát.

Az egyik különbség a kártyák között az egyes (azonos) üzemmódok száma (amit al-be kell írni a 10h int rutinnál). Ezt ugyan megadják a kártyához tartozó kis füzetkében, de ha esetleg olyan programot akarunk írni, ami nem csak egy féle kártyatípuson kellene hogy működjön, akkor a többiét is ismerni kell. Nos a most következő ismertető talán segít egy kicsit ebben.

ACUMOS

Üzem mód:	Felbontás:	Színek száma:
59h	800*600	2
58h	800*600	16
5Ch	800*600	256
5Dh	1024*768	16
5Eh	640*400	256
5Fh	640*480	256

AHEAD

Üzem mód:	Felbontás:	Színek száma:
60h	640*400	256
61h	640*480	256
62h	800*600	256
63h	1024*768	256
6Ah	800*600	16
71h	800*600	16
74h	1024*768	16

ATI

Üzem mód:	Felbontás:	Színek száma:
54h	800*600	16
61h	640*400	256
62h	640*480	256
63h	800*600	256
64h	1024*768	256
65h	1024*768	16

CHIPS & TECH

Üzem mód:	Felbontás:	Színek száma:
70h	800*600	16
72h	1024*768	16
78h	640*400	256
79h	640*480	256
7Bh	800*600	256

EVEREX

!! ax=0x70h bl=Üzem mód !!

Üzem mód:	Felbontás:	Színek száma:
-----------	------------	---------------

Grafika programozása Assembly nyelven

01h	752*410	16
02h	800*600	16
11h	1280*350	4
12h	1280*600	4
13h	640*350	256
14h	640*400	256
15h	512*480	256
20h	1024*768	16
30h	640*480	256
31h	800*600	256
32h	1024*768	256

GENOA GVGA

Üzem mód:	Felbontás:	Színek száma:
59h	720*512	16
5Bh	640*350	256
5Ch	640*480	256
5Dh	720*512	256
5Eh vagy 6Ch	800*600	256
5Fh	1024*768	16
6Ah vagy 79h	800*600	16
7Ch	512*512	16
7Dh	512*512	256
7Eh	640*400	256
7Fh	1024*768	4

NCR

Üzem mód:	Felbontás:	Színek száma:
58h	800*600	16
59h	800*600	2
5Ah	1024*768	2
5Ch	800*600	256
5Dh	1024*768	16
5Eh	640*400	256
5Fh	640*480	256

OAK TECH

Üzem mód:	Felbontás:	Színek száma:
52h	800*600	16
53h	640*480	256
54h	800*600	256
56h	1024*768	16
58h	1280*1024	16
59h	1024*768	256

PARADISE

Üzem mód:	Felbontás:	Színek száma:
58h	800*600	16
59h	800*600	2
5Ch	800*600	256
5Dh	1024*768	16
5Eh	640*400	256
5Fh	640*480	256

TRIDENT 8900CL

Üzem mód:	Felbontás:	Színek száma:
5Bh	800*600	16
5Ch	640*400	256
5Dh	640*480	256
5Eh	800*600	256
5Fh	1024*768	16
60h	1024*768	4
61h	768*1024	16
62h	1024*768	256
6Ch	640*480	16M
74h	640*480	32K
75h	640*480	64K
76h	800*600	32K
77h	800*600	64K

TSENG (Genoa, Orchid, Willow)

Üzem mód:	Felbontás:	Színek száma:
29h	800*600	16
2Dh	640*350	256
2Eh	640*480	256
2Fh	720*512	256
30h	800*600	256
37h	1024*768	16

TSENG 4000

Üzem mód:	Felbontás:	Színek száma:
29h	800*600	16
2Dh	640*350	256
2Eh	640*480	256
2Fh	640*400	256
30h	800*600	256
37h	1024*768	16
38h	1024*768	256

VIDEO7

!! ax=6F05h bl=Üzem mód !!

Üzem mód:	Felbontás:	Színek száma:
60h	752*410	16
61h	720*540	16
62h	800*600	16
63h	1024*768	2
64h	1024*768	4
65h	1024*768	16
66h	640*400	256
67h	640*480	256
68h	720*540	256
69h	800*600	256
6Ah	1024*768	256

Az itt feltüntetett üzemmódokat kell al regiszterbe írni az üzemmód váltásakor. Kivételt képez az a két eset, ahol az külön jelölve is van, itt az ax regiszterbe a megadott értéket, bl-be pedig az üzemmódot kell írni.

A következő probléma akkor adódik, ha olyan üzemmódot használunk, ahol a kép mérete túllépi a 64KByte-os méretet (16 színű módoznál a 4*64KByte-ot). Ekkor ugyanis lapozni kell. Azaz a videokártyán található memóriából egy másik szegmenst lapoz be a gép RAMjába (általában 0A000h-tól kezdődően). Nos itt jelentkezik a dokumentáció hiánya. Ugyanis azt, hogy az adott kártyát mely portcímen keresztül és mi módon lehet lapozni, azt általában elfelejtik közölni a felhasználóval.

A következő program arra hivatott, hogy egy 256 színű SVGA módban pontokból kirajzoljon egy vonalat a képernyő valamely részére. Erre egyébként a ROM BIOS is képes, így ez is bemutatásra kerül. A különbség szemmel látható lesz annak ellenére, hogy a közvetlen memóriakezelő rutint lehetett volna még gyorsítani azzal, ha megvizsgáljuk, hogy kell-e lapozni, vagy pedig ugyan arra a szegmensre esik a pont, ahová az előzőt tettük. Ami fontos, hogy a programban ki lehet választani pár gyakori kártyatípus közül a megfelelőt és a gép ettől függően állítja be a video üzemmódot, és a kártyának megfelelő lapozó eljárást fogja használni.

[Program 30]

Svga1	Segment		;Szegmensdefiníció.
	assume cs:Svga1,ds:Svga1 ;Cs,ds		hozzárendelése.
Szeleseg	equ 1024		;Az üzemmód X irányú felbon- tása a címkszámításhoz.
Start:	mov ax,Svga1		;Ds regiszter beállítása a kód
	mov ds,ax		;elejére.
	mov ax,0a000h		;A videomemória kezdőcímet
	mov es,ax		;es regiszterbe teszi.
	call Trident_Ini		;Az üzemmód bekapcsolása.

Grafika programozása Assembly nyelven

```
.1_Svga1:  mov  cx,1024                ;A vonal hossza.
           push  cx

           mov  ax,word ptr [Koord_Y] ;Az Y koordináta értékét
           mov  bx,Szeleseg        ;megszorozzuk a képernyő
           mul  bx                  ;szélességével, és az így
           mov  di,ax               ;kapott eredmény kisebbik
           add  di,word ptr [Koord_X] ;helyiértékű részét di-be
           adc  dx,0                ;tesszük, majd hozzáadjuk
           ;a vízszintes koordináta
           ;értékét. A lapozáshoz a
           ;magasabb helyiértékű word
           ;kerül felhasználásra mivel
           ;a szorzáskor itt alakul ki,
           ;hogyanyszor nagyobb a cím
           ;a szegmensméretnél.
           ;A számításnál hozzá kell meg
           ;adni az összeadáskor esetleg
           ;képződött carry flaget mert
           ;ez is azt jelenti, hogy nem
           ;fért bele egy wordbe a szám.

           call Trident_Lapoz      ;A kiszámított lap beállítása.

           mov  al,byte ptr [Szin]  ;A számolt helyre a megadott
           mov  es:[di],al          ;színnel kitesz egy pontot.

           pop  cx
           inc  word ptr [Koord_X] ;Az X koordináta növelése
           loop .1_Svga1           ;míg cx nem nulla.

           xor  ax,ax               ;Billentyűvárás.
           int  16h

           mov  ax,3                ;A szöveges mód beállítása.
           int  10h
           mov  ax,4c00h            ;Visszatérés az operációs
           int  21h                ;rendszerhez.

Ahead_Ini Proc

           mov  ax,63h
           int  10h
           ret

Ahead_Ini Endp
```

Oak_Ini Proc

```
mov ax,59h
int 10h
ret
```

Oak_Ini Endp

Trident_Ini Proc

```
mov ax,62h
int 10h
ret
```

Trident_Ini Endp

Tseng_Ini Proc

```
mov ax,38h
int 10h
ret
```

Tseng_Ini Endp

Ahd_Lapoz Proc

```
mov ah,dl
mov cl,4
shl dl,cl
or ah,dl
mov al,0dh
mov dx,3ceh
out dx,ax
ret
```

Ahd_Lapoz Endp

Oak_Lapoz Proc

```
mov dh,dl
mov cl,4
shl dh,cl
```



```
or    dl,dh
mov   ah,dl
mov   al,11h
mov   dx,3deh
out   dx,al
inc   dx
in    al,dx
mov   al,ah
out   dx,al
ret
```

Oak_Lapoz Endp

Trid_Lapoz Proc

```
mov   ah,dl
mov   dx,3c4h
mov   al,0eh
out   dx,al
inc   dx
in    al,dx
and   ax,0ff0h
xor   ah,2
or    al,ah
out   dx,al
ret
```

Trid_Lapoz Endp

Tsg_Lapoz Proc

```
mov   al,dl
mov   ah,al
mov   cl,3
shl   al,cl
or    al,40h
or    al,ah
mov   dx,3cdh
out   dx,al
ret
```

Tsg_Lapoz Endp

IBM PC Gyakorlati Assembly

```
Koord_X: dw 0
Koord_Y: dw 384
Szin: db 15
```

```
Svga1      Ends                ;A szegmens vége.
           End  Start          ;A program vége.
```

[Program 31]

```
Svga2      Segment            ;Szegmensdefiníció.
           assume cs:Svga2,ds:Svga2 ;Cs,ds hozzárendelése.

Start:     mov  ax,Svga2       ;Ds regiszter beállítása a kód
           mov  ds,ax          ;elejére.

           mov  ax,0a000h      ;A videomemória kezdőcímet
           mov  es,ax          ;es regiszterbe teszi.

           call Trident_Ini    ;Az üzemmód bekapcsolása.

           mov  cx,1024        ;A vonal hossza.
.1_Svga2:  push cx

           mov  al,byte ptr [Szin] ;Al regiszterbe a pont színét,
           mov  cx,word ptr [Koord_X] ;cx-be az X, dx-be az Y
           mov  dx,Word ptr [Koord_Y] ;koordinátát kell helyezni,
           xor  bh,bh          ;bh tartalmazza a grafikus lap
           mov  ah,0ch         ;számát (itt csak 0 lehet) és
           int  10h           ;az ah=0ch jelenti a rutinnak,
                               ;hogy egy pontot kell kitenni
                               ;a megadott paraméterekkel.

           pop  cx
           inc  word ptr [Koord_X] ;Az X koordináta növelése
           loop .1_Svga2        ;mig cx nem nulla.

           xor  ax,ax          ;Billentyűvárás.
           int  16h

           mov  ax,3           ;A szöveges mód beállítása.
           int  10h

           mov  ax,4c00h       ;Visszatérés az operációs
           int  21h            ;rendszerhez.
```

Ahead_Ini Proc

```
mov ax,63h
int 10h
ret
```

Ahead_Ini Endp

Oak_Ini Proc

```
mov ax,59h
int 10h
ret
```

Oak_Ini Endp

Trident_Ini Proc

```
mov ax,62h
int 10h
ret
```

Trident_Ini Endp

Tseng_Ini Proc

```
mov ax,38h
int 10h
ret
```

Tseng_Ini Endp

Koord_X: dw 0

Koord_Y: dw 384

Szin: db 15

Svga2 Ends ;A szegmens vége.

End Start ;A program vége.

A programok működése igen egyszerű, az egyetlen olyan dolog, ami eddig nem nagyon volt használva, az a cím kiszámításnál alkalmazott megoldás, mégpedig az, hogy 16 bites szor-

zásnál az eredményt `dx:ax`-ben kapjuk meg és ezzel meg lehet határozni, hogy a pont hányadik videoszegmensre esik mivel `dx` regiszter azt fogja mutatni, hogy az eredmény hányszor nagyobb egy szegmensnél. Az `adc` utasításra azért van szükség, mert előfordulhat, hogy a szorzás után `di` regiszterben egy 65535-höz közeli szám alakul ki, és a vízszintes koordináta hozzáadása során körbefordulna a regiszter és így egy 0-hoz közeli értéket kapnánk, ami tulajdonképpen jó is, de nem azon a szegmensen, ahol az előbb voltunk, hanem a következőn. Ezért kell hozzáadni a `dx` regiszterhez a `0+carry` értéket.

Amikor a pontkirakást a BIOS végzi el, akkor annyi csupán a dolgunk, hogy a megfelelő regiszterekbe beállítsuk a kívánt paramétereket és elindítsuk a rutint. A gép automatikusan kiszámolja a képpont címét és a megfelelő színűre állítja azt. Az egyetlen különbség a sebesség, az utóbbi sokszor lassabb a direkt rutinnál.

FÜGGELÉK

A PSP felépítése:

Cím byte:	Tartalom:
0-1	A két byte tartalma: CD 20, ami egy int 20 utasításnak felel meg. Erre a címre ugorva a program befejezheti a futását.
2-3	A memória felső határ, az első nem használható memóriacím paragrafusokban kifejezve.
4	Fenntartott.
5-9	A DOS funkciókkal kapcsolatos távoli call utasítás.
0Ah-0B	A program befejezési offsetcíme.
0Ch-0Dh	A program befejezési szegmenscíme.
0Eh-0Fh	A Ctrl Break kilépés offsetcíme.
10h-11h	A Ctrl Break kilépés szegmenscíme.
12h-13h	A hibakezelő rutin offsetcíme.
14h-15h	A hibakezelő rutin szegmenscíme.
16h-2B	A DOS részére fenntartott terület.
2Ch-2Dh	A programkörnyezet szegmenscíme
2Eh-4Fh	A DOS részére fenntartott terület.
50h-52h	Ez a 3 byte egy int 21h távoli hívást tartalmaz.
53h-5Bh	A DOS részére fenntartott terület.

5Ch-6Bh	Normál (nem megnyitott) file leíró blokk (FCB1)
6Ch-7Fh	Második normál nem megnyitott FCB2. Ezt a két file leíró blokkot a DOS készíti el és az adott utasításra vonatkozó paramétereket tartalmazza.
80h-FFh	Lemez átviteli terület (DTA). Itt tárolódnak azok a paraméterek, amiket az utasítást követően megadtunk mint paramétert.

A 10h megszakítás lehetőségei:

Funkció:	Működése:
ah=0	A megjelenítési mód beállítása, az üzemmód sorszámát. EGA,VGA (MCGA) kártyák esetén ha az üzemmód 7. bitje 1 értékű, akkor nem törli a képernyőt, különben igen.
ah=1	A kurzor típusának beállítása. ch regiszter alsó 4 bitje a kurzor kezdő, cl alsó 4 bitje az utolsó sorának számát határozza meg. Csak szöveges képernyőn alkalmazható.
ah=2	A kurzor pozíciójának beállítása. Dh a sor, dl az oszlop, bh a lap sorszámát határozza meg.
ah=3	A kurzor pozíciójának lekérdezése. A bh regiszterben kell megadni a lap sorszámát, ahol kíváncsiak vagyunk a kurzor helyzetére és az előbb említett regiszterekben kapjuk meg a szükséges információkat. Grafikus üzemmódban a lap sorszáma általában 0.
ah=4	A fényceruza regisztereinek olvasása. Amikor ah regiszter értéke 1, akkor a fényceruzával kijelölt karakter sorát a dh, oszlopát a dl regiszterben kapjuk meg. Grafikus módban a ch illetve EGA kártyáknál a cx regiszter tartalmazza a fényceruza Y bx pedig az X koordinátáját.

- ah=5** Az aktív lap kiválasztása, a lap sorszámát **al** regiszterben kell megadni.
- ah=6** Az aktuális lap egy részének felfelé mozgatása. A sorok számát **al** regiszterben kell megadni (0 esetén törli a képet). A mozgatandó terület bal felső sarkát **cl ch**, jobb alsó sarkát pedig **dl dh** regiszterek határozzák meg.
- ah=7** Az aktuális lap egy részének lefelé mozgatása. A megadandó paraméterek megegyeznek a felfelé mozgatásnál leírtakkal.
- ah=8** A **bh** regiszter által mutatott lapon a kurzor pozíciójában lévő karakter **ascii** kódjának és színének lekérdezése. A kódot **ah**, a szint **al** regiszterekben kapjuk meg.
- ah=9** Karakterek írása a kurzor pozíciójától kezdődően. A kiírandó karakter kódját **al**, az ismétlések számát **cx** regiszterben kell megadni. **Bh** regiszter jelentése 256 színű grafikus módoknál a háttérszín, egyébként a lap sorszáma, **bl** regiszter karakteres képernyőn a teljes szint, grafikus módoknál az előtér színét határozza meg. Ha **bl** regiszter 7. bitje 1 értékű, akkor az adatot **xor** művelettel teszi ki a képernyőre.
- ah=0Ah** A funkció nagyban megegyezik az előzővel, annyi különbséggel, hogy szöveges módban nincs lehetőség attribútum megadására, ekkor az éppen aktuális szint fogja használni.
- ah=0Bh** A színpaletta állítása. Ha **bh** regiszter értéke nulla, akkor 320*200 CGA módban a háttér, karakteres módban a keret, 640*200 CGA módban az előtér, 640*200 EGA módban a háttér színét lehet megadni **bl** regiszterben. Ha **bh** regiszter tartalma 1 akkor a 320*200-as CGA képernyőhöz tartozó két színpaletta közül lehet választani **bl** segítségével (0 vagy 1).

- ah=0Ch** Pont kirakása a **dx** által megadott sorban, **cx** által megadott oszlopban **al** színnel, **bh** által mutatott lapra. 256 színű grafikus módban **al** csak a szint határozza meg, különben ha a 7. bitje 1 értékű, akkor az aktuális helyen lévő színnel **xor** kapcsolatba hozza az általunk megadottat, és az eredményt fogjuk látni a képernyőn.
- ah=0Dh** Egy pont visszaolvasása. Az eredményt az előbb említett regiszterekben kapjuk.
- ah=0Eh** Karakter kirakása a kurzor pozíciójába annak mozgatásával. **Al** regiszterbe kell tenni a kiírandó karaktert, grafikus módban **bl** regiszterbe az előtér színét, szöveges módban **bh**-ba a lap számát.
- ah=0Fh** Az aktuális video mód lekérdezése. **Al** regiszterben kapjuk az üzemmód sorszámát, **ah**-ban az egy sorban lévő karakterek számát, **bh** regiszterben pedig az aktív lap számát.
- ah=10h** EGA, MCGA, VGA kártyák színpalettáinak beállítása a megfelelő értékűre.
- al=0** A **bl** regiszterben megadott számú paletta regiszter beállítása **bh**-ban megadott színűre.
- al=1** A keret színének beállítása **bh** regiszterben megadott színűre. (kivételel MCGA)
- al=2** A paletta regiszterek és a keret állítása **es:dx** által mutatott adatterület értékei alapján. Az első 16 byte a palettaszint, a 17. byte a keret színét határozza meg.
- al=3** A szín byte 3. bitjének értelmezése. Ha **bl=0** akkor az intenzitás, **bl=1** esetén pedig villogás vezérlése.

- al=7** A VGA kártyák paletta regiszterek kiolvasása. A regiszterek jelentése megegyezik a 0. funkcióval.
- al=8** A keret színének visszaolvasása. A szín értékét **bh** regiszterben kapjuk meg.
- ah=9** A VGA kártyák színpalettáinak értékét és a keret színét egy **es:dx** által mutatott 17 byte-os adat-területre másolja. Az egyes byte-ok értelmezése megegyezik a 2. funkciónál leírtakkal.
- al=10h** Az MCGA és VGA kártyák paletta regisztereinek beállítása. **Bx**-ben megadott sorszámú palettát a **dh ch cl** regiszterek által meghatározott RGB összetevőknek megfelelő színűre állítja. Az egyes komponensek maximális értéke 256 színű módoknál 63 mivel ilyenkor a gép teljes színpalettája 18 bites.
- al=12h** Az MCGA és a VGA kártyák színpalettáinak beállítása az **es:dx** által mutatott táblázat adatai alapján. **Cx** mutatja a beállítandó színregiszterek számát. A színeket RGB sorrendben kell tárolni a memóriában.
- al=13h** A VGA szinkiválasztó regiszter beállítása. Ha **bl** értéke 0, akkor az üzemmód regiszter 7. bitjét a **bh** regiszter 7. bitje határozza meg. Ha **bl=1** akkor a **bh**-ban megadott adatot a szinkiválasztó regiszterbe írja.
- al=15h** Az MCGA és a VGA kártyák színregisztereinek olvasása. A regiszterek működére megegyezik a 10h funkciónál említettekkel.
- al=17h** Az MCGA és a VGA kártyák színregisztereinek kiolvasása a 12h funkciónál leírtakkal megegyező módon.

- al=18h** Az MCGA és a CGA kártyák digitál-analóg átalakítója maszkregiszterének beállítása **bl** regiszterben megadott értékre.
- al=19h** A digitál-analóg átalakító maszkregiszterének kiolvasása a 18h funkciónak megfelelően.
- al=1Ah** A VGA kártyák szinkiválasztó regiszterének kiolvasása a **bh** regiszterbe. **Bl** regisztert az üzemmód regiszter 7. bitje alapján állítja be. Ha **bl=0**, akkor **bh** regiszterbe a szinkiválasztó regiszter 2. 3. bitjét, ha **bl=1** akkor a 0-3. bitjét teszi.
- al=1Bh** Szürke skála előállítása VGA illetve MCGA kártyáknál. **Bx** regiszterben kell megadni az első szinkód számát, **cx**-ben pedig a színskála hosszát.
- ah=11h** Az EGA, MCGA, VGA üzemmódok karakter generátorának programozása az **al** regiszterben meghatározott alfunkció szerint. **Al=0-4** funkciók alaphelyzetbe állítják a megjelenítőt, de nem törlik a képernyőt, **al=10h-14h** funkciók megegyeznek a 0-4 funkciókkal, de csak a 0. lap aktivizálása után használhatók. **Al=20-24** a grafikus karakterkészletet tölti be a memóriába.
- al=0** A felhasználó által definiált karakterkészlet betöltése. **Es:bp** regiszterekbe kell beállítani a karaktertábla kezdőcímét, **cx**-be a karakterek számát, **dx**-be az első karakter kódját, **bl** regiszterbe a karaktergenerátor sorszámát, ahová betöltse a készletet és **bh** regiszterbe az egy karaktert leíró byte-ok számát.
- al=1** A 8*14 pont méretű betűkészlet betöltése a ROM-ból. **Bl** regiszterben kell megadni a karakter generátor sorszámát.

- al=2** A 8*8-as betűkészlet betöltése, **bl** regiszter jelentése megegyezik az előzővel.
- al=3** Az aktív karakter generátor kijelölése. A **bl** regiszterben kell megadni a készlet sorszámát.
- al=4** Csak MCGA és VGA kártyáknál a 8*16 pontos betűkészlet betöltése.
- al=10h-14h** Id.: 1-4.
- al=20h** 8*8 képpont méretű grafikus felhasználói készlet betöltése. A kezdőcímet **es:bp** regiszterekben kell megadni.
- al=21h** Felhasználói grafikus karakterkészlet betöltése **es:bp** címtől. Az egy karakterhez tartozó byte-ok számát **cx** regiszterben kell megadni. A megjelenítendő sorok számát **bl** értéke határozza meg. Ha **bl=0**, akkor a megjelenítendő sorok számát **dl** regiszter tartalmazza, **bl=1** esetén 14, **bl=2** esetén 25, **bl=3** esetén pedig 43 soros megjelenítés.
- al=22h** 8*14 pont méretű grafikus készlet betöltése a memóriába. **Bl** és **dl** szerepe megegyezik az előzővel.
- al=23h** 8*8 pontos grafikus karakterkészlet betöltése a memóriába. **Bl**, **dl** szerepe azonos az előzőekkel.
- al=24h** VGA kártyák 8*16 pontos készletének betöltése. **Bl**, **dl** szerepe mint előbb.
- al=30h** A karakterkészlet címének lekérdezése **bh** regiszterben megadott szempont szerint. A karakterenkénti sorok számát **cx** a kijelzett sorok számát pedig **dl** regiszterben kapjuk. Az **es:bp** értéke **bh** tartalmától függően áll be:

- bh=0** **Es:bp** az int 1Fh vektort adja,
- bh=1** **Es:bp** az int 44h vektort adja,
- bh=2** **Es:bp** a ROM 8*14 pontos karakter készletének címét adja,
- bh=3** **Es:bp** a 8*8-as készlet 0-127 ascii kódokhoz tartozó címét adja,
- bh=4** **Es:bp** a 8*8-as készlet második felének címét mutatja,
- bh=5** **Es:bp** a ROM 9*14 pontos karakter készletének címét adja,
- bh=6** **Es:bp** a ROM 8*16 pontos karakter készletének címét adja,
- bh=7** **Es:bp** a ROM 9*16 pontos karakter készletének címét adja,

ah=12h EGA, MCGA, VGA konfigurációs funkciók:

bl=10h Konfigurációs információk lekérdezése. Hívás után ha **bh** regiszter értéke nulla, a beállított üzemmód színes, 1 esetén fekete-fehér. **Ch** regiszter mutatja a bemérő csatlakozón lévő jelet, **cl** a kártya csatlakozójának állását. **Bl** regiszterből tudhatjuk meg a memória kapacitás mértékét.

- bl=0** : 64Kbyte,
- bl=1** : 128 Kbyte,
- bl=2** : 196 Kbyte,
- bl=3** : 256 Kbyte vagy több.

bl=20h A 43 soros EGA, VGA képernyőtartalmat kinyomtatni képes rutin kiválasztása.

bl=30h Csak VGA kártyáknál a karakteres képernyő rastersorainak száma:

al=0 : 200 sor

al=1 : 350 sor

al=2 : 400 sor

bl=31h VGA kártya esetén a standard paletta értékek betöltésének engedélyezése:

al=0 A betöltés engedélyezve,

al=1 A betöltés tiltva.

bl=32h A VGA kártyák periféria-, memória regiszterei elérésének engedélyezése, tiltása:

al=0 Az elérés megengedett,

al=1 Az elérés tiltott.

bl=33h Fekete-fehér VGA kimenet előállításának engedélyezése:

al=0 A konverzió megengedett,

al=1 A konverzió tiltott.

bl=34h A kurzor emuláció engedélyezése VGA kártyán:

al=0 Az emuláció megengedett,

al=1 Az emuláció tiltott.

bl=36h VGA kártya kimeneteinek vezérlése:

al=0 A kimenetek engedélyezettek,

al=1 A kimenetek tiltottak.

- ah=13h** Az **es:bp** címen kezdődő **cx** hosszúságú szöveg kiíratása a **bh** regiszter által meghatározott lapra. A szöveg koordinátáját **dx** regiszterben kell megadni, **dh**-ba a sor-, **dl** regiszterbe pedig az oszlop koordinátáját. **Al** regiszter segítségével több alfunkció közül választhatunk:
- al=0** A szöveg csak a karakterek kódjait tartalmazza, az attribútum információt a **bl** regiszter határozza meg. A kurzort nem mozgatja.
- al=1** A szöveg csak a karakterek kódjait tartalmazza, az attribútum információt a **bl** regiszter határozza meg. A kurzort mozgatja.
- al=2** A szöveg a karakter és színkódokat felváltva tartalmazza, a kurzort nem mozgatja.
- al=3** A szöveg a karakter és színkódokat felváltva tartalmazza, a kurzort mozgatja.
- ah=1Ah** A megjelenítő adatainak lekérdezése.
- al=0** A megjelenítő adatainak lekérdezése, **bl**-ben az aktív, **bh** regiszterben az inaktív megjelenítő kódját kapjuk.

Kód	Megjelenítő
0	Nincs beépítve
1	MDA fekete-fehér
2	CGA színes
3	Fenntartva
4	EGA színes
5	EGA fekete-fehér
6	PGA
7	VGA analóg fekete-fehér
8	VGA analóg színes
9	Fenntartva
0Ah	MCGA digitális színes
0Bh	MCGA analóg fekete-fehér

0Ch MCGA analóg színes
0FFh Ismeretlen megjelenítő típus.

al=1 Megjelenítő kombinációk beállítása. A regiszterek használata megegyezik az előbb használtakal.

ah=1Bh Az MCGA, VGA adottságainak és állapotának lekérdezése. A rutin hívása előtt **bh** regiszterbe nullát, **es:di** regiszterekbe pedig egy 64 byte méretű adattárolásra használható memóriarekesz címét kell megadni, ahová a gép az adott megjelenítővel kapcsolatos információkat tárolni fogja. Ennek felépítése:

Cím	Hossz	Funkció
0	4	A BIOS statikus adattáblázatára mutató cím.
4	1	Az aktuális üzemmód kódja.
5	2	A kijelzett karakteroszlopok száma.
7	2	A képernyő frissítésére használt memória terület hossza.
9	2	A memória terület kezdőcíme.
0Bh	16	Az egyes lapokon lévő kurzorok címei.
1Bh	1	A kurzor utolsó sorának száma.
1Ch	1	A kurzor kezdősorának száma.
1Dh	1	Az aktív lap sorszáma.
1Eh	2	A kártya periféria címe.
20h	1	Üzemmód regiszter értéke.
21h	1	A szinkiválasztó regiszter értéke.

22h	1	A megjelenített karaktorsorok száma.
23h	2	Egy karakter rasztersorainak száma.
25h	1	Az aktív megjelenítő kódja.
26h	1	Az inaktív megjelenítő kódja.
27h	2	A kijelzett színek száma.
29h	1	Az elérhető lapok száma.
2Ah	1	A kijelzett resztersorok száma: 0 - 200 1 - 350 2 - 400 3 - 480 4 - fenntartva 5 - 600 6 - 768
2Bh	1	A karaktergenerátor sorszám, amit a színbyte 3. bitjének 0 értéke mellett használ VGA kártyán.
2Ch	1	A karaktergenerátor sorszám, amit az előbb említett bit 1 értéke mellett használ.
2Dh	1	Állapotkód. Az egyes bitek jelentése: 7-6 Fenntartva. 5 A villogás engedélyezett. 4 Kurzor emuláció engedett. 3 Az alap palettaértékek betöltése engedélyezett. 2 Fekete-fehér monitor használata. 1 A szürke árnyalatok előállítás engedélyezett. 0 Minden mód beállítható.
2Eh	3	Fenntartva.

31h	1	<p>A kártya memóriakapacitása:</p> <p style="margin-left: 40px;">0 - 64K 1 - 128K</p> <p style="margin-left: 40px;">2 - 196K 3 - 256K vagy több.</p>																														
32h	1	<p>Paraméter mentés lehetőségei:</p> <p>7-6 Fenntartva.</p> <p>5 Kiegészítő táblázat használata.</p> <p>4 A paletta regiszterek felülírhatók.</p> <p>3 A grafikus karakterek felülírhatók.</p> <p>2 A normál karakterek felülírhatók.</p> <p>1 Dinamikus jellemzők táblázata aktív.</p> <p>0 Két karakterkészlet használata VGA kártyánál.</p>																														
33h	13	<p>Fenntartva</p> <p>A statikus paraméterek táblázata:</p> <table border="0" style="width: 100%;"> <thead> <tr> <th style="text-align: left;">Cím</th> <th style="text-align: left;">Hossz</th> <th style="text-align: left;">Funkció</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>3</td> <td>A 0 - 13h üzemmódok beállíthatósága, az engedélyezést az egyes bitek értéke határozza meg.</td> </tr> <tr> <td>3</td> <td>4</td> <td>Fenntartva.</td> </tr> <tr> <td>7</td> <td>1</td> <td>A beállítható pontsorok száma karakteres módban:</td> </tr> <tr> <td></td> <td></td> <td>7-3 Fenntartva.</td> </tr> <tr> <td></td> <td></td> <td>2 400 sor.</td> </tr> <tr> <td></td> <td></td> <td>1 350 sor.</td> </tr> <tr> <td></td> <td></td> <td>0 200 sor.</td> </tr> <tr> <td>8</td> <td>1</td> <td>Az egyszerre megjeleníthető karakter készletek száma.</td> </tr> <tr> <td>9</td> <td>1</td> <td>A betölthető készletek száma.</td> </tr> </tbody> </table>	Cím	Hossz	Funkció	0	3	A 0 - 13h üzemmódok beállíthatósága, az engedélyezést az egyes bitek értéke határozza meg.	3	4	Fenntartva.	7	1	A beállítható pontsorok száma karakteres módban:			7-3 Fenntartva.			2 400 sor.			1 350 sor.			0 200 sor.	8	1	Az egyszerre megjeleníthető karakter készletek száma.	9	1	A betölthető készletek száma.
Cím	Hossz	Funkció																														
0	3	A 0 - 13h üzemmódok beállíthatósága, az engedélyezést az egyes bitek értéke határozza meg.																														
3	4	Fenntartva.																														
7	1	A beállítható pontsorok száma karakteres módban:																														
		7-3 Fenntartva.																														
		2 400 sor.																														
		1 350 sor.																														
		0 200 sor.																														
8	1	Az egyszerre megjeleníthető karakter készletek száma.																														
9	1	A betölthető készletek száma.																														

0Ah	2	Egyéb BIOS tulajdonságok:
		15 A D/A átalakító a színekiválasztó regiszterrel vezérelhető.
		14 A színekódok betölthetők.
		13 64 palettaregiszter használható.
		12 Kurzor emuláció használható.
		11 Paletta alapértékeinek betöltése használható.
		10 Karakter készlet betölthető.
		9 A szürkeárnyalatok beállíthatók.
		8 Minden mód használható.
		7-4 Fenntartva.
		3 Kombinációs lehetőségek.
		2 Villogás / háttérintenzitás.
		1 Az állapot mentés használható.
		0 Fényceruza engedélyezett.
0Ch	2	Fenntartva.
0Eh	1	Mentési paraméterek:
		7-6 Fenntartva.
		5 Kiegészítő táblázat használható.
		4 Paletta regiszterek menthetők.
		3 Grafikus készlet menthető.
		2 Normál karakterek menthetők.
		1 Dinamikus jellemzők táblázata menthető.
		0 Több karakterkészlet engedett.
0Fh	1	Fenntartva.
ah=1Ch		A VGA kártya állapotának mentése, visszatöltése.
al=0		A mentéshez szükséges memóriaterület méretének lekérdezése. A cx regiszter bitjeivel választható ki a menteni kívánt tulajdonságok csoportja és bx regiszterben kapjuk a szükséges memória terület méretét 64Byte-os egységben.

Bit	Paramétercsoport
2	Színkódok adatai
1	BIOS adatmező
0	Hardware paraméterek.

al=1 A paraméterek mentése az **es:bx** által címzett memória területre. **Cx** szerepe az lekérdezésnél leírtakkal megegyezik.

al=2 A lementett paraméterek visszaállítása az **es:bx** által mutatott memória terület adatai alapján. **Cx** szerepe hasonló az előbb említettekkel.

Tárgymutató

A

AAA, 74
AAD, 75
AAM, 75
AAS, 74
Adatbusz, 7
ADC, 76
ADD, 76
ALU, 6
AND, 25; 88; 101
Ascii kód, 17
Assembly, 8

B

BCD, 41
Belső regiszter, 114
Betűtípus, 135
Bináris, 36
BIT, 7; 25; 93; 130
Bitsík, 122
Bittérkép, 136; 174
Brush, 182
Byte, 7; 16; 20

C

CALL, 32; 87
Carry, 102
CBW, 80
CGA, 92
Ciklusok, 9; 32
Cím, 7
Címbusz, 7
Címke, 20; 23
CLC, 81
CLD, 61; 82
CLL, 82
CMC, 82
CMP, 80

CMPS, 80

CMPSB, 57; 81
CMPSW, 57; 81
CPU, 6
CWD, 80

D

DAA, 75
DAS, 75
DEC, 76
Decimális, 38
DIV, 40; 77
DMA, 67
DTA, 66
Dword, 7; 16; 20

E

EGA, 116
Egérkurzor, 98
Eljárások, 9
EQU, 69

F

FAR, 23
Felbontás, 92; 116; 140
Feltételek, 10
File, 52
Flag, 12; 24; 70

G

Gépikód, 8

H

Handle, 53
Háttér, 121
Hercules, 92
Hexadecimális, 36
HLT, 90
Holtciklus, 70

I

I/O, 68
IDIV, 77
Ikon, 92; 98; 143
IMUL, 77
IN, 69; 82
INC, 76
Indexregiszter, 12
INS, 83
INSB, 83
INSW, 83
INT, 17; 87
Int, 70
Intenzitás, 93
INTO, 87
IRET, 70; 87

J

JA, 84
JC, 85
JG, 85
JGE, 85
JMP, 23; 84
JNA, 84
JNC, 84
JNG, 85
JNGE, 86
JNZ, 85
JZ, 85

K

Karakterkészlet, 117; 133
Kártya, 92
Képernyő-memória, 92; 93; 140
Képernyővezérlő, 114
Képfarmátum, 174
Keret, 120
Komplemens, 25
Kontroller, 68
Koordináta, 97

L

LAHF, 80
LDS, 79
LEA, 78
Lebegőpontos, 153
LES, 79
LOCK, 90
LODS, 78
LODSB, 78
LODSW, 78
LOOP, 32; 86
LOOPNZ, 86
LOOPZ, 86

M

Megszakítás, 67
Memória, 51
Memóriaváltozó, 20
Menü, 44
Monitor, 92
MOV, 15; 77
MOVS, 78
MOVSB, 78
MOVSW, 78
MUL, 77

N

NEG, 25; 88
NOP, 90
NOT, 25; 88

O

Offset, 10; 17
OR, 26; 88; 101
Org, 14
OUT, 69; 83; 92
OUTS, 83
OUTSB, 83
OUTSW, 83

P

Paletta, 93; 117; 120; 140; 176
Paragrafus, 11
Paritás, 68
Plane, 117; 121
Pont, 95
POP, 38; 84
POPF, 84
Port, 67; 92; 114
PSP, 55; 66
PUSH, 38; 83
PUSHF, 83

R

RAM, 6
RCL, 28; 89
RCR, 27; 89
Real-Time, 41
Regiszter, 10
REP, 57; 86
REPNZ, 86
REPZ, 86
RET, 32; 87
RETF, 87
Rezidens, 70
ROL, 27; 89
ROM BIOS, 14; 17; 33; 92;
 117
ROR, 27; 89

S

SAHF, 80
SAL, 29; 89
SAR, 28; 90
SBB, 76
Scan kód, 51
SCAS, 81
SCASB, 57; 81
SCASW, 57; 81
Scroll, 159
Sequencer, 126

SET-RESET, 122

SHL, 28; 89
SHR, 28; 89
Stack, 38
Standard, 183
STC, 81
STD, 82
STL, 82
STOS, 78
STOSB, 78
STOSW, 78
SUB, 76
SVGA, 183
Szegmens, 10; 14; 23; 55
Szegmenscím, 93
Szegmensregiszter, 12
Színbyte, 17
Szinkiosztás, 93

T

TEST, 81
Tömörítés, 174

Ü

Üzem mód, 92

V

Változók, 9
Verem, 38
Vertical Blank, 114
VGA, 116
Video lap, 117
Videomemória, 117

W

WAIT, 90
Word, 7; 16; 20

X

XCHG, 79
XLAT, 36; 79
XOR, 26; 88

Ha *elektronikai alkatrészek* beszerzéséről van szó, Magyarországon még ma is az **ELEKTROMODUL** a legnagyobb szakvállalat.

- * készletezés
- * *bizományi beszerzés*
- * *export - import*
- * *nagykereskedelem*
- * *kiskereskedelmi árusítás, bizományban is*
- * *katalógustár*
- * *alkalmazástechnikai tanácsadás.*

De *kereskedőház*-ként nemcsak alkatrészekkel foglalkozunk!

- * *alapanyagok*
- * *alkatrészek*
- * *szereleési anyagok* * *részegységek*
- * *gépek*
- * *műszerek*
- * *komplett technológiák*

beszerzését vállaljuk elsősorban az elektronikai ipar területén, de esetenként más ágazatok részére is.

Raktárbázisunk szolgáltatásai:

- * *vámudvarként behozatali és kiviteli vámkezelés*
- * *vámkiszérvényezés*
- * *bérraktározás.*



ELEKTROMODUL Vállalat Elektronikai Kereskedőház
1132 Budapest, Victor Hugo u. 11-15.
Tel.: 270-2899, 270-1477 Fax: 129-1650 Tx.: 22-5154

Raktárbázis:
1158 Budapest, Késmárk u. 92.
Tel.: 251-8700 Fax: 163-3087 Tx: 22-6622