Assembly Programozás

Rodek Lajos Diós Gábor

Tartalomjegyzék

Ábrák jegyzéke IV

Táblázatok jegyzéke V

Előszó VI

Ajánlott irodalom VII

1. Az Assembly nyelv jelentősége 1

2. A PC-k hardverének felépítése 4

3. Számrendszerek, gépi adatábrázolás 7

4. A 8086-os processzor jellemzői 13
- 4.1. Memóriakezelés . . . 13
- 4.2. Regiszterek . . . 14
- 4.3. Adattípusok . . . 17
- 4.4. Memóriahivatkozások, címzési módok . . . 18
  - 4.4.1. Közvetlen címzés . . . 18
  - 4.4.2. Báziscímzés . . . 18
  - 4.4.3. Indexcímzés . . . 18
  - 4.4.4. Bázis+relatív címzés . . . 18
  - 4.4.5. Index+relatív címzés . . . 18
  - 4.4.6. Bázis+index címzés . . . 18
  - 4.4.7. Bázis+index+relatív címzés . . . 18
- 4.5. Veremkezelés . . . 20
- 4.6. I/O, megszakítás-rendszer . . . 23

5. Az Assembly nyelv szerkezete, szintaxisa 24

6. A 8086-os processzor utasításkészlete 29
- 6.1. Prefixek . . . 29
  - 6.1.1. Szegmensfelülbíráló prefixek . . . 29
  - 6.1.2. Buszlezáró prefix . . . 30
  - 6.1.3. Sztringutasítást ismétlő prefixek . . . 30
- 6.2. Utasítások . . . 30

6.2.1. Adatmozgató utasítások 30
6.2.2. Egész számos aritmetika 31
6.2.3. Bitenkénti logikai utasítások 31
6.2.4. Bitléptető utasítások 32
6.2.5. Sztringkezelő utasítások 32
6.2.6. BCD aritmetika 32
6.2.7. Vezérlésátadó utasítások 32
6.2.8. Rendszervezérlő utasítások 33
6.2.9. Koprocesszor-vezérlő utasítások 33
6.2.10. Speciális utasítások 33

7. Assembly programok készítése **35**

8. Vezérlési szerkezetek megvalósítása **37**
8.1. Szekvenciális vezérlési szerkezet 37
8.2. Számlálásos ismétléses vezérlés 40
8.3. Szelekciós vezérlés 43
8.4. Eljárásvezérlés 48

9. A Turbo Debugger használata **52**

10. Számolás előjeles számokkal **56**
10.1. Matematikai kifejezések kiértékelése 56
10.2. BCD aritmetika 60
10.3. Bitforgató utasítások 62
10.4. Bitmanipuláló utasítások 64

11. Az Assembly és a magas szintű nyelvek **67**
11.1. Paraméterek átadása regisztereken keresztül 67
11.2. Paraméterek átadása globálisan 69
11.3. Paraméterek átadása a vermen keresztül 70
11.4. Lokális változók megvalósítása 73

12. Műveletek sztringekkel **75**

13. Az .EXE és a .COM programok, a PSP **81**
13.1. A DOS memóriakezelése 81
13.2. Általában egy programról 82
13.3. Ahogy a DOS indítja a programokat 84
13.4. .COM állományok 85
13.5. Relokáció 86
13.6. .EXE állományok 88

14. Szoftver-megszakítások **90**
14.1. Szövegkiíratás, billentyűzet-kezelés 91
14.2. Szöveges képernyő kezelése 92
14.3. Munka állományokkal 96
14.4. Grafikus funkciók használata 99

15. Hardver-megszakítások, rezidens program **102**
15.1. Szoftver-megszakítás átirányítása 103
15.2. Az időzítő (timer) programozása 107
15.3. Rezidens program készítése 111

16. Kivételek **114**

A. Átváltás különféle számrendszerek között **116**

B. Karakter kódtáblázatok **120**

Tárgymutató **125**

Irodalomjegyzék **130**

Ábrák jegyzéke

2.1. A PC-k felépítése 4

3.1. A NOT művelet igazságtáblája 8
3.2. Az AND művelet igazságtáblája 8
3.3. Az OR művelet igazságtáblája 8
3.4. A XOR művelet igazságtáblája 9

4.1. Az AX regiszter szerkezete 16
4.2. A Flags regiszter szerkezete 16

9.1. A Turbo Debugger képernyője 54

Táblázatok jegyzéke

8.1. Előjeles aritmetikai feltételes ugrások 45
8.2. Előjeltelen aritmetikai feltételes ugrások 45
8.3. Speciális feltételes ugrások 46

14.1. Gyakran használt szoftver-megszakítások 91
14.2. Állomány- és lemezkezelő DOS-szolgáltatások 98
14.3. Standard I/O eszközök handle értéke 98
14.4. Színkódok 101

16.1. A 8086-os processzoron létező kivételek 115

A.1. Átváltás a 2-es, 8-as, 10-es és 16-os számrendszerek között 116

B.1. ASCII és EBCDIC vezérlőkódok 120
B.2. ASCII és EBCDIC karakterkódok 121

Előszó

Valami . . .

Ajánlott irodalom

A következő lista tartalmazza azon művek adatait, amelyek elolvasása elősegít(het)i a jegyzet könnyebb megértését:

1. Pethő Ádám: Assembly alapismeretek 1. kötet, Számalk, Budapest, 1992
2. Peter Norton – John Socha: Az IBM PC Assembly nyelvű programozása, Novotrade, Budapest, 1991
3. Peter Norton: Az IBM PC programozása, Műszaki Könyvkiadó, Budapest, 1992
4. László József: A VGA-kártya programozása Pascal és Assembly nyelven, ComputerBooks, Budapest, 1994
5. Abonyi Zsolt: PC hardver kézikönyv
6. Dr. Kovács Magda: 32 bites mikroprocesszorok 80386/80486 I. és II. kötet, LSI, Budapest

Az (1), (2) és (3) könyvek a kezdőknek, az Assemblyvel most ismerkedőknek valók.

A (4) és (5) könyvek a hardverrel foglalkoznak, az Assemblyt ezekből nem fogjuk megtanulni.

Végül a (6) egy referenciakönyv, így ezt főleg az Assemblyben már jártas, de mélyebb ismeretekre vágyóknak ajánljuk.

1. fejezet

Az Assembly nyelv tulajdonságai, jelentősége

A számítógépes problémamegoldás során a kitűzött célt megvalósító algoritmust mindig valamilyen **programozási nyelven** (programming language) írjuk, kódoljuk le. A nyelvet sokszor az adott feladat alapján választjuk meg, míg máskor aszerint döntünk egy adott nyelv mellett, hogy az hozzánk, az emberi gondolkodáshoz mennyire áll közel. Ez utóbbi tulajdonság alapján csoportosíthatók a számítógépes programozási nyelvek: megkülönböztetünk **alacsony szintű** (low-level) és **magas szintű programozási nyelveket** (high-level programming language). Az előbbire jó példák az Assembly és részben a C, az utóbbira pedig a Pascal ill. a BASIC nyelvek. Ha a nyelvek szolgáltatásait tekintjük, akkor szembeötlő, hogy ahogy egyre fentebb haladunk az alacsony szintű nyelvektől a magas szintűek felé, úgy egyre nagyobb szabadsággal, egyre általánosabb megoldásokkal találkozunk.

Az Assembly tehát egy alacsony szintű programozási nyelv, méghozzá nagyon alacsony szintű, ebből következően pedig sokkal közelebb áll a hardverhez, mint bármely más nyelv. Főbb jellemzői:

- nagyon egyszerű, elemi műveletek
- típustalanság
- rögzített utasításkészlet
- világos, egyszerű szintaxis
- kevés vezérlési szerkezet
- nagyon kevés adattípus; ha több is van, akkor általában egymásból származtathatók valahogyan

De miért is van szükség az Assemblyre, ha egyszer ott van a többi nyelv, amikben jóval kényelmesebben programozhatunk? Erre egyik indok, hogy a magas szintű nyelvek eljárásai, függvényei sokszor általánosra lettek megírva, így teljesen feleslegesen foglalkoznak olyan dolgokkal, amikre esetleg soha sem lesz szükségünk. Erre jó példák lehetnek a Borland Pascal/C grafikus eljárásai, valamint ki-/bemeneti (I/O) szolgáltatásai. Kört rajzolhatunk a Circle eljárással is, de ennél gyorsabb megoldást kapunk, ha vesszük a fáradságot, és mi magunk írunk

egy olyan körrajzolót, ami semmi mást nem csinál, csak ami a feladata: helyesen kirajzolja a kört a képernyőre, de nem foglalkozik pl. hibaellenőrzéssel, a képernyő szélén kívülre kerülő pontok kiszűrésével stb. Hasonló a helyzet a fájlkezeléssel is. Ha nem akarunk speciális típusokat (mondjuk objektumokat, rekordokat) állományba írni, mindössze valahány bájtot szeretnénk beolvasni vagy kiírni a lemezre, akkor felesleges a fenti nyelvek rutinjait használni. Mindkét feladat megoldható Assemblyben is, méghozzá hatékonyabban, mint a másik két nyelvben.

Akkor miért használják mégis többen a C-t, mint az Assemblyt? A választ nem nehéz megadni: magasabb szintű nyelvekben a legtöbb probléma gyorsabban leírható, a forrás rövidebb, strukturáltabb, s ezáltal áttekinthetőbb lesz, könnyebb lesz a későbbiekben a program karbantartása, és ez nem csak a program szerzőjére vonatkozik. Mégsem mellőzhetjük az Assemblyt, sok dolgot ugyanis vagy nagyon nehéz, vagy egyszerűen képtelenség megcsinálni más nyelvekben, míg Assemblyben némi energia befektetése árán ezek is megoldhatók. Aki Assemblyben akar programozni, annak nagyon elszántnak, türelmesnek, kitartónak kell lennie. A hibalehetőségek ugyanis sokkal gyakoribbak itt, és egy-egy ilyen baki megkeresése sokszor van olyan nehéz, mint egy másik program megírása.

Régen, mikor még nem voltak modern programozási nyelvek, fordítóprogramok, akkor is kellett valahogy dolgozni az embereknek. Ez egy **gépi kódnak** (machine code) nevezett nyelven történt. A gépi kód a processzor saját nyelve, csak és kizárólag ezt érti meg. Ez volt ám a fárasztó dolog! A gépi kód ugyanis nem más, mint egy rakás szám egymás után írva. Akinek ebben kellett programozni, annak fejből tudnia kellett az összes utasítás összes lehetséges változatát, ismernie kellett a rendszert teljes mértékben. Ha egy kívülálló rátekintett egy gépi kódú programra, akkor annak működéséből, jelentéséből jobbára semmit sem értett meg. Nézzünk egy példát: 0B8h 34h 12h // 0F7h 26h 78h 56h // 0A3h 78h 56h, ahol a dupla törtvonal az utasításhatárt jelzi. Ez a tíz hexadecimális (tizenhatos számrendszerbeli) szám nem mond túl sokat első ránézésre. Éppen ezért kidolgoztak egy olyan jelölésrendszert, nyelvet, amiben emberileg emészthető formában leírható bármely gépi kódú program. Ebben a nyelvben a hasonló folyamatot végrehajtó gépi kódú utasítások csoportját egyetlen szóval, az ú.n. **mnemonikkal** (mnemonic) azonosítják. Természetesen van olyan mnemonik is, ami egyetlen egy utasításra vonatkozik. Ez a nyelv lett az Assembly. Ebben az új jelölésrendszerben a fenti programrészlet a következő formát ölti:

```
MOV     AX,1234h                ;0B8h 34h 12h
MUL     WORD PTR [5678h]        ;0F7h 26h 78h 56h
MOV     [5678h],AX              ;0A3h 78h 56h
```

Némi magyarázat a programhoz:

- az első sor egy számot (1234h) rak be az AX regiszterbe, amit most tekinthetünk mondjuk egy speciális változónak
- a második sor a fenti értéket megszorozza a memória egy adott címén (5678h) található értékkel
- a harmadik sor az eredményt berakja az előbbi memóriarekeszbe
- a pontosvessző utáni rész csak megjegyzés
- az első oszlop tartalmazza a mnemonikot
- a memóriahivatkozásokat szögletes zárójelek ([és]) közé írjuk

Tehát a fenti három sor egy változó tartalmát megszorozza az 1234h számmal, és az eredményt visszarakja az előbbi változóba.

Mik az Assembly előnyei?

- korlátlan hozzáférésünk van a teljes hardverhez, beleértve az összes perifériát (billentyűzet, nyomtató stb.)
- pontosan ellenőrizhetjük, hogy a gép tényleg azt teszi-e, amit elvárunk tőle
- ha szükséges, akkor minimalizálhatjuk a program méretét és/vagy sebességét is (ez az ú.n. optimalizálás)

Most lássuk a hátrányait:

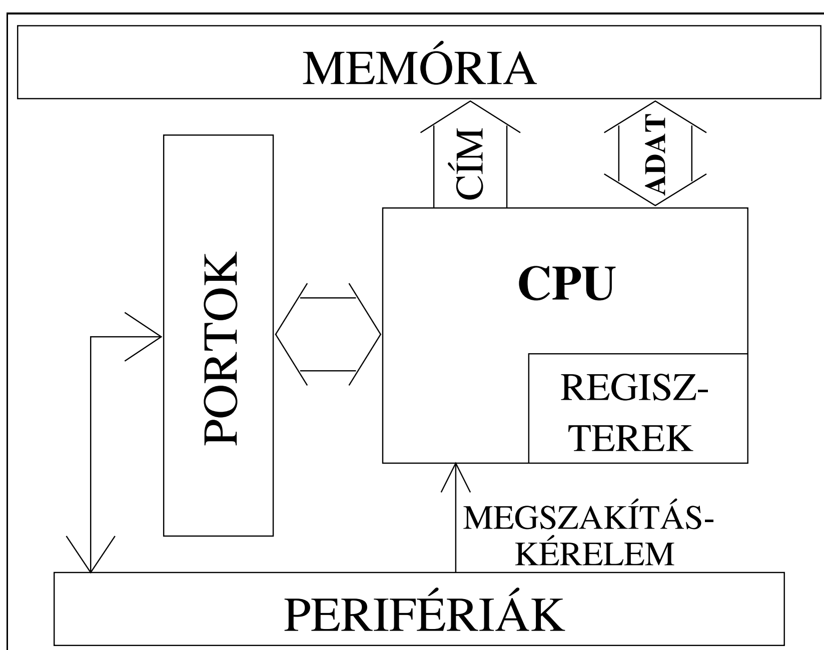
- a forrás sokszor áttekinthetetlen még a szerzőnek is
- a kódolás nagy figyelmet, türelmet, és főleg időt igényel
- sok a hibalehetőség
- a hardver alapos ismerete elengedhetetlen
- a forrás nem hordozható (nem portolható), azaz más alapokra épülő számítógépre átírás nélkül nem vihető át (ez persze igaz a gépi kódra is)

Bár úgy tűnhet, több hátránya van mint előnye, mégis érdemes alkalmazni az Assemblyt ott, ahol más eszköz nem segít. A befektetett erőfeszítések pedig meg fognak térülni.

2. fejezet

A PC-k hardverének felépítése

Az IBM PC-k felépítését szemlélteti a 2.1. ábra eléggé leegyszerűsítve:



2.1. ábra. A PC-k felépítése

Az első IBM PC az Intel 8086-os **mikroprocesszorával** jelent meg, és hamarosan követte az IBM PC XT, ami már Intel 8088-os maggal ketyegett. Később beköszöntött az AT-k időszaka, s vele jöttek újabb processzorok is: Intel 80286, 80386 (SX és DX), 80486 (SX, DX, DX2 és DX4), majd eljött az 586-os és 686-os gépek világa (Pentium, Pentium Pro, Pentium II stb.) Nem csak az Intel gyárt processzorokat PC-kbe, de az összes többi gyártó termékére jellemző, hogy (elvileg) 100%-osan kompatibilis az Intel gyártmányokkal, azaz ami fut Intel-en, az ugyanúgy elfut a másik procin is, és viszont. Az összes későbbi processzor alapja tulajdonképpen a 8086-os volt, éppen ezért mondják azt, hogy a processzorok ezen családja az Intel

80x86-os (röviden x86-os) architektúrájára épül.

A továbbiakban kizárólag az Intel 8086/8088-os processzorok által biztosított programozási környezetet vizsgáljuk. (A két proci majdnem teljesen megegyezik, a különbség mindössze az adatbuszuk szélessége: a 8086 16 bites, míg a 8088-as 8 bites külső adatbusszal rendelkezik.)

A 2.1. ábrán a **CPU** jelöli a processzort (Central Processing Unit – központi feldolgozó egység). Mint neve is mutatja, ő a gép agya, de persze gondolkodni nem tud, csak végrehajtja, amit parancsba adtak neki. A CPU-n többek között található néhány különleges, közvetlenül elérhető tárolóhely. Ezeket **regisztereknek** (register) nevezzük. A processzor működés közbeni újraindítását **resetnek** hívjuk. (Az angol „reset” szó egyik magyar jelentése az „utánállít, beállít”, de a számítástechnikában általában „újraindításként” fordítják.) Reset esetén a processzor egy jól meghatározott állapotba kerül (pl. minden regiszterbe valamilyen rögzített érték lesz betöltve). A számítógép bekapcsolásakor is lezajlik a reset. Az újraindítás egy finomabb fajtája az **inicializálás** vagy röviden **init** (initialization, init). Az init során néhány regiszter értéke megőrzésre kerül, a reset-től eltérően.

A **memória** adja a számítógép „emlékezetét”. Hasonlóan az emberi memóriához, van neki felejtő (az információt csak bizonyos ideig megőrző) és emlékező változata. Az előbbi neve **RAM** (Random Access Memory – véletlen hozzáférésű memória), míg az utóbbié **ROM** (Read Only Memory – csak olvasható memória). A ROM fontos részét képezi az ú.n. **BIOS** (Basic Input/Output System – alapvető ki-/bemeneti rendszer). A gép bekapcsolása (és reset) után a BIOS-ban levő egyik fontos program indul el először (pontosabban minden processzort úgy terveznek, hogy a BIOS-t kezdje el végrehajtani ilyenkor). Ez leellenőrzi a hardverelemeket, teszteli a memóriát, megkeresi a jelenlevő perifériákat, majd elindítja az operációs rendszert, ha lehet. A BIOS ezenkívül sok hasznos rutint tartalmaz, amikkel vezérelhetjük például a billentyűzetet, videokártyát, merevlemezt stb.

Busznak (bus) nevezzük „vezetékek” egy csoportját, amik bizonyos speciális célt szolgálnak, és a CPU-t kötik össze a számítógép többi fontos részével. Megkülönböztetünk adat-, cím- ill. vezérlőbuszt (data bus, address bus, control bus). A címbusz szélessége (amit bitekben ill. a „vezetékek” számában mérünk) határozza meg a megcímezhető memória maximális nagyságát.

A CPU a **perifériákkal** (pl. hangkártya, videovezérlő, DMA-vezérlő, nyomtató stb.) az ú.n. **portokon** keresztül kommunikál. (Tekinthetjük őket egyfajta „átjárónak” is.) Ezeket egy szám azonosítja, de ne úgy képzeljük el, hogy annyi vezeték van bekötve, ahány port van. Egyszerűen, ha egy eszközt el akar érni a CPU, akkor kiírja a címbuszára a port számát, és ha ott „van” eszköz (tehát egy eszköz arra van beállítva, hogy erre a portszámra reagáljon), akkor az válaszol neki, és a kommunikáció megkezdődik.

Azonban nem csak a CPU szólhat valamelyik eszközhöz, de azok is jelezhetik, hogy valami mondanivalójuk van. Erre szolgál a **megszakítás-rendszer** (interrupt system). Ha a CPU érzékel egy **megszakítás-kérelmet** (IRQ – Interrupt ReQuest), akkor abbahagyja az éppen aktuális munkáját, és kiszolgálja az adott eszközt. A megszakítások először a **megszakítás-vezérlőhöz** (interrupt controller) futnak be, s csak onnan mennek tovább a processzorhoz. Az XT-k 8, az AT-k 16 db. független **megszakítás-vonallal** rendelkeznek, azaz ennyi perifériának van lehetősége a megszakítás-kérésre. Ha egy periféria használ egy megszakítás-vonalat, akkor azt kizárólagosan birtokolja, tehát más eszköz nem kérhet megszakítást ugyanazon a vonalon.

A fentebb felsorolt rendszerelemek (CPU, memória, megszakítás-vezérlő, buszok stb.) és még sok minden más egyetlen áramköri egységen, az ú.n. **alaplapon** (motherboard) található.

Van még három fontos eszköz, amikről érdemes szót ejteni. Ezek az órajel-generátor, az időzítő és a DMA-vezérlő.

A CPU és a perifériák működését szabályos időközönként megjelenő elektromos impulzusok vezérlik. Ezeket nevezik **órajelnek** (clock, clocktick), másodpercenkénti darabszámuk

mértékegysége a Hertz (Hz). Így egy 4.77 MHz-es órajel másodpercenként 4770000 impulzust jelent. Az órajelet egy kvarckristályon alapuló **órajel-generátor** (clock generator) állítja elő.

A RAM memóriák minden egyes memóriarekeszét folyamatosan ki kell olvasni és vissza kell írni másodpercenként többször is, különben tényleg „felejtővé” válna. Erre a **frissítésnek** (memory refresh) nevezett műveletre felépítésük miatt van szükség. (A korrektség kedvéért: ez csak az ú.n. dinamikus RAM-okra, avagy DRAM-okra igaz.) A műveletet pontos időközönként kell végrehajtani, ezt pedig egy különleges egység, az **időzítő** (timer) intézi el. Ennek egyik dolga az, hogy kb. 15.09 μs-onként elindítsa a frissítést (ez nagyjából 66287 db. frissítést jelent másodpercenként). Ezenkívül a rendszerórát (system clock) is ez a szerkezet szinkronizálja.

A memória elérése a CPU-n keresztül igen lassú tud lenni, ráadásul erre az időre a processzor nem tud mással foglalkozni. E célból bevezették a **közvetlen memória-hozzáférés** (DMA – Direct Memory Access) módszerét. Ez úgy működik, hogy ha egy perifériának szüksége van valamilyen adatra a memóriából, vagy szeretne valamit beírni oda, akkor nem a CPU-nak szól, hanem a **DMA-vezérlőnek** (DMA controller), és az a processzort kikerülve elintézi a kérést.

3. fejezet

Számrendszerek, gépi adatábrázolás, aritmetika és logika

Az emberek általában tízes (**decimális** – decimal) számrendszerben számolnak a mindennapjaik során, hiszen ezt tanították nekik, és ez az elfogadott konvenció a világon. A processzort azonban (de a többi hardverösszetevőt, pl. a memóriát is) feleslegesen túlbonyolítaná, ha neki is ezekkel a számokkal kellene dolgoznia. Ennél jóval egyszerűbb és kézenfekvő megoldás, ha kettes alapú (**bináris** – binary) számrendszerben kezel minden adatot. Az információ alapegysége így a bináris számjegy, a **bit** (BInary digiT) lesz, ezek pedig a 0 és az 1. Bináris számok ábrázolásakor ugyanúgy helyiértékes felírást használunk, mint a decimális számok esetén. Pl. a 10011101 bináris számnak 128 + 16 + 8 + 4 + 1 = 157 az értéke. Az egyes helyiértékek jobbról balra 2-nek egymás után következő hatványai, tehát 1, 2, 4, 8, 16, 32 stb. Ha sokszor dolgozunk bináris számokkal, akkor nem árt, ha a hatványokat fejből tudjuk a 0-diktól a 16-odikig.

Egy-egy aránylag kicsi szám bináris leírásához sok 0-t és 1-et kell egymás mellé raknunk, ez pedig néha fárasztó. Ezt kiküszöbölendő a számokat sokszor írjuk tizenhatos alapú (**hexadecimális** – hexadecimal) számrendszerben. Itt a számjegyek a megszokott 10 arab számjegy, plusz az angol (latin) ábécé első hat betűje (A, B, C, D, E, F), továbbá A = 10, B = 11 stb. Mivel $16 = 2^4$, ezért négy bináris számjegy éppen egy hexadecimális (röviden hexa) számjegyet tesz ki. Az előző példa alapján 10011101b = 9Dh = 157d. Ahhoz, hogy mindig tudjuk, a leírt számot milyen alapú rendszerben kell értelmezni, a szám után írunk egy „b”, „h” vagy „d” betűt. Ha nem jelöljük külön, akkor általában a tízes számrendszert használjuk. Szokás még néha a nyolcas alapú (**oktális** – octal) felírást is alkalmazni. Ekkor a 0 – 7 számjegyeket használjuk, és 3 bináris jegy tesz ki egy oktális számjegyet. Az oktális számok végére „o” betűt írunk.

Most elevenítsük fel a legegyszerűbb, közismert logikai műveleteket. Ezek ugyanis fontos szerepet játszanak mind a programozás, mind a processzor szempontjából. A két logikai igazságértéket itt most bináris számjegyek fogják jelölni. Megszokott dolog, hogy 1 jelenti az „igaz” értéket.

A **negáció** (tagadás – negation) egyváltozós (**unáris**) művelet, eredménye a bemeneti igazságérték ellentettje. A műveletet jelölje NOT az angol tagadás mintájára. Hatása a 3.1. ábrán látható.

A **konjunkció** („ÉS”) már kétváltozós (**bináris**) művelet. Jele AND (az „és” angolul), eredményét a 3.2. ábra szemlélteti:

NOT	
0	1
1	0

3.1. ábra. A NOT művelet igazságtáblája

AND	0	1
0	0	0
1	0	1

3.2. ábra. Az AND művelet igazságtáblája

A **diszjunkció** („VAGY”) szintén bináris művelet. Jele OR (a „vagy” angolul), és a két változón MEGENGEDŐ VAGY műveletet hajt végre. Igazságtáblája a 3.3. ábrán látható.

OR	0	1
0	0	1
1	1	1

3.3. ábra. Az OR művelet igazságtáblája

Utolsó műveletünk az **antivalencia** („KIZÁRÓ VAGY”, az ekvivalencia tagadása). Jele az XOR (eXclusive OR), hatása a 3.4. ábrán követhető.

A legtöbb processzor kizárólag egész számokkal tud számolni, esetleg megenged racionális (valós) értékeket is. Az ábrázolható számok tartománya mindkét esetben véges, ezt ugyanis a processzor regisztereinek bitszélessége határozza meg. A számítástechnikában a legkisebb ábrázolható információt a bit képviseli, de mindenhol az ennél nagyobb, egészen pontosan 8 db. bitből álló **bájtot** (byte) használják az adatok alapegységeként, és pl. a regiszterek és az adatbusz szélessége is ennek többszöröse. Szokás még más, a bájt fogalmára épülő mértékegységet is használni, ilyen a **szó** (word; általában 2 vagy 4 bájt), a **duplaszó** (doubleword; a szó méretének kétszerese) és a **kvadraszó** (quadword; két duplaszó méretű). A bájt alsó ill. felső felének (4 bitjének) neve **nibble** (ez egy angol kifejezés, és nincs magyar megfelelője). Így beszélhetünk alsó és felső nibble-ről. A bájtban a biteket a leírás szerint jobbról balra 0-tól kezdve számozzák, és egy bájtot két hexadecimális számjeggyel lehet leírni.

A számítástechnikában a kilo- (k, K) és mega- (M) előtétszavak a megszokott 1000 és 1000000 helyett 1024-et ($= 2^{10}$) ill. 1048576-ot ($= 2^{20}$) jelentenek. A giga- (G), tera- (T), peta- (P) és exa- (E) hasonlóan 2^{30}-t, 2^{40}-t, 2^{50}-t ill. 2^{60}-t jelentenek.

Fontos szólni egy fogalomról, az ú.n. **endianizmusról** (endianism). Ez azt a problémát jelenti, hogy nincs egyértelműen rögzítve a több bájt hosszú adatok ábrázolása során az egyes bájtok memóriabeli sorrendje. Két logikus verzió létezik: a legkevésbé értékes bájttal kezdünk, és a memóriacím növekedésével sorban haladunk a legértékesebb bájt felé (**little-endian tárolás**), ill. ennek a fordítottja, tehát a legértékesebbtől haladunk a legkevésbé értékes bájt felé (**big-endian tárolás**). Mindkét megoldásra találhatunk példákat a különböző hardvereken.

Nézzük meg, hogy ábrázoljuk az egész számokat. Először tételezzük fel, hogy csak nemnegatív (**előjeltelen** – unsigned) számaink vannak, és 1 bájtot használunk a felíráshoz. Nyolc biten 0 és 255 ($= 2^8 - 1$) között bármilyen szám felírható, ezért az ilyen számokkal nincs gond.

XOR	0	1
0	0	1
1	1	0

3.4. ábra. A XOR művelet igazságtáblája

Ha negatív számokat is szeretnénk használni, akkor két lehetőség adódik:

- csak negatív számokat ábrázolunk
- az ábrázolási tartományt kiterjesztjük a nemnegatív számokra is

Ez utóbbi módszert szokták választani, és a tartományt praktikus módon úgy határozzák meg, hogy a pozitív és negatív számok nagyjából azonos mennyiségben legyenek. Ez esetünkben azt jelenti, hogy −128-tól +127-ig tudunk számokat felírni (beleértve a 0-t is). De hogy különböztessük meg a negatív számokat a pozitívaktól? Természetesen az **előjel** (sign) által. Mivel ennek két értéke lehet (ha a nullát pozitívnak tekintjük), tárolására elég egyetlen bit. Ez a kitüntetett bit az **előjelbit** (sign bit), és megegyezés szerint az adat legértékesebb (most significant), azaz legfelső bitjén helyezkedik el. Ha értéke 0, akkor a tekintett **előjeles** (signed) szám pozitív (vagy nulla), 1 esetén pedig negatív. A fennmaradó biteken (esetünkben az alsó 7 bit) pedig tároljuk magát a számot előjele nélkül. Megtehetnénk, hogy azonos módon kezeljük a pozitív és negatív számokat is, de kényelmi szempontok és a matematikai műveleti tulajdonságok fenntartása végett a negatív számok más alakban kerülnek leírásra. Ez az alak az ú.n. **kettes komplemens** (2's complement). Ennek kiszámítása majdnem triviális, egyszerűen ki kell vonni a számot a 0-ból. (Ehhez persze ismerni kell a kivonás szabályait, amiket alább ismertetünk.) Egy másik módszerhez vezessük be az **egyes komplemens** (1's complement) fogalmát is. Ezt úgy képezzük bármilyen értékű bájt (szó stb.) esetén, hogy annak minden egyes bitjét negáljuk (invertáljuk). Ha vesszük egy tetszőleges szám egyes komplemensét, majd ahhoz hozzáadunk 1-et (az összeadást az alábbiak szerint elvégezve), akkor pont az adott szám kettes komplemensét kapjuk meg. Egy szám kettes komplemensének kettes komplemense a kiinduló számot adja vissza. Ilyen módon a kettes komplemens képzése ekvivalens a szám −1-szeresének meghatározásával.

Egy példán illusztrálva: legyenek az ábrázolandó számok 0, 1, 2, 127, 128, 255, −1, −2, −127 és −128. A számok leírása ekkor így történik:

- 0 (előjeltelen vagy előjeles pozitív) = 00000000b
- 1 (előjeltelen vagy előjeles pozitív) = 00000001b
- 2 (előjeltelen vagy előjeles pozitív) = 00000010b
- 127 (előjeltelen vagy előjeles pozitív) = 01111111b
- 128 (előjeltelen) = 10000000b
- 255 (előjeltelen) = 11111111b
- −1 (előjeles negatív) = 11111111b
- −2 (előjeles negatív) = 11111110b
- −127 (előjeles negatív) = 10000001b

- -128 (előjeles negatív) = 10000000b

Láthatjuk, hogy az előjeltelen és előjeles ábrázolás tartományai között átfedés van ($0-127$), míg más értékek esetén ütközés áll fenn ($128-255$ ill. $-1--128$). Azaz a 11111111b számot olvashatjuk 255-nek de akár -1-nek is!

Ha nem bájton, hanem mondjuk 2 bájtos szóban akarjuk tárolni a fenti számokat, akkor ezt így tehetjük meg:

- 0 = 00000000 00000000b
- 1 = 00000000 00000001b
- 2 = 00000000 00000010b
- 127 = 00000000 01111111b
- 128 = 00000000 10000000b
- 255 = 00000000 11111111b
- -1 = 11111111 11111111b
- -2 = 11111111 11111110b
- -127 = 11111111 10000001b
- -128 = 11111111 10000000b

Ebben az esetben meg tudjuk különböztetni egymástól a -1-et és a 255-öt, de ütköző rész itt is van ($32768-65535$ ill. $-1--32768$).

Végül megnézzük, hogy végezhetők el a legegyszerűbb matematikai műveletek. A műveletek közös tulajdonsága, hogy az eredmény mindig hosszabb 1 bittel, mint a két szám közös hossza (úgy tekintjük, hogy mindkettő tag azonos bitszélességű). Így két 1 bites szám összege és különbsége egyaránt 2 bites, míg két bájté 9 bites lesz. A +1 bit tulajdonképpen az esetleges átvitelt tárolja.

Két bináris számot ugyanúgy adunk össze, mint két decimális értéket:

1. kiindulási pozíció a legalacsonyabb helyiértékű jegy, innen haladunk balra
2. az átvitel kezdetben 0
3. az aktuális pozícióban levő két számjegyet összeadjuk, majd ehhez hozzáadjuk az előző átvitelt (az eredmény két jegyű bináris szám lesz)
4. a kétbites eredmény alsó jegyét leírjuk az összeghez, az átvitel pedig felveszi a felső számjegy értékét
5. ha még nem értünk végig a két összeadandón, akkor menjünk ismét a (3)-ra
6. az átvitelt mint számjegyet írjuk hozzá az összeghez

Az összeadási szabályok pedig a következőek:

- 0 + 0 = 00 (számjegy = 0, átvitel = 0)

- $0 + 1 = 01$ (számjegy = 1, átvitel = 0)
- $1 + 0 = 01$ (számjegy = 1, átvitel = 0)
- $1 + 1 = 10$ (számjegy = 0, átvitel = 1)
- $10 + 01 = 11$ (számjegy = 1, átvitel = 1)

Ezek alapján ellenőrizhetjük, hogy a fent definiált kettes komplemens alak valóban teljesíti azt az alapvető algebrai tulajdonságot, hogy egy számnak és additív inverzének (tehát −1-szeresének) összege 0 kell legyen. És valóban:

$$1\text{d} + (-1\text{d}) = 00000001\text{b} + 11111111\text{b} = 1\ 00000000\text{b}$$

Az eredmény szintén egy bájt lesz (ami tényleg nulla), valamint keletkezik egy átvitel is. Az egyes komplemens is rendelkezik egy érdekes tulajdonsággal. Nevezetesen, ha összeadunk egy számot és annak egyes komplemensét, akkor egy csupa egyesekből álló számot, azaz −1-et (avagy a legnagyobb előjeltelen számot) fogunk kapni!

Kivonáskor hasonlóan járunk el, csak más szabályokat alkalmazunk:

- $0 - 0 = 00$ (számjegy = 0, átvitel = 0)
- $0 - 1 = 11$ (számjegy = 1, átvitel = 1)
- $1 - 0 = 01$ (számjegy = 1, átvitel = 0)
- $1 - 1 = 00$ (számjegy = 0, átvitel = 0)
- $11 - 01 = 10$ (számjegy = 0, átvitel = 1)

továbbá a fenti algoritmusban a (3) lépésben az átvitelt le kell vonni a két számjegy különbségéből. Ezek alapján ugyancsak teljesül, hogy

$$1\text{d} - 1\text{d} = 00000001\text{b} - 00000001\text{b} = 0\ 00000000\text{b}$$

azaz egy számot önmagából kivonva 0-t kapunk. Viszont lényeges eltérés az előző esettől (amikor a kettes komplemens alakot adtuk hozzá a számhoz), hogy itt sose keletkezik átvitel a művelet elvégzése után.

A kivonás szabályainak ismeretében már ellenőrizhető, hogy a kettes komplemenst valóban helyesen definiáltuk:

$$00000000\text{b} - 00000001\text{b} = 1\ 11111111\text{b}$$

A szorzás és az osztás már macerásabbak. Mindkét művelet elvégzése előtt meghatározzuk az eredmény (szorzat ill. hányados) előjelét, majd az előjeleket leválasztjuk a tagokról. Mindkét műveletet ezután ugyanúgy végezzük, mint ahogy papíron is csinálnánk. Osztás alatt itt maradékos egész osztást értünk. A szorzat hossza a kiinduló tagok hosszainak összege, a hányadosé az osztandó és osztó hosszainak különbsége, míg a maradék az osztandó hosszát örökli. (Ezt azért nem kell szigorúan venni. Egy szót egy bájttal elosztva a hányados nyudodtan hosszabb lehet 8 bitnél. Pl.: 0100h/01h = 0100.) A maradék előjele az osztandóéval egyezik meg, továbbá teljesül, hogy a maradék abszolút értékben kisebb mint az osztó abszolút értéke.

A műveletek egy speciális csoportját alkotják a 2 hatványaival való szorzás és osztás. Ezeket közös néven **shiftelésnek** (eltolásnak, léptetésnek) hívjuk.

2-vel úgy szorozhatunk meg egy számot a legegyszerűbben, ha a bináris alakban utána írunk egy 0-t. De ez megegyezik azzal az esettel, hogy minden számjegy eggyel magasabb helyiértékre csúszik át, az alsó, üresen maradó helyet pedig egy 0-val töltjük ki. Erre azt mondjuk, hogy a számot egyszer balra shifteltük. Ahányszor balra shiftelünk egy számot, mindannyiszor megszorozzuk 2-vel, végeredményben tehát 2-nek valamely hatványával szorozzuk meg. Ez a módszer minden számra alkalmazható, legyen az akár negatív, akár pozitív.

Hasonlóan definiálható a jobbra shiftelés is, csak itt a legfelső megüresedő bitpozíciót töltjük fel 0-val. Itt azonban felmerül egy bökkenő, nevezetesen negatív számokra nem fogunk helyes eredményt kapni. A probléma az előjelbitben keresendő, mivel az éppen a legfelső bit, amit pedig az előbb 0-val helyettesítettünk. A gond megszüntethető, ha bevezetünk egy előjeles és egy előjel nélküli jobbra shiftelést. Az előjeltelen változat hasonlóan működik a balra shifteléshez, az előjelesnél pedig annyi a változás, hogy a legfelső bitet változatlanul hagyjuk (vagy ami ugyanaz, az eredeti előjelbittel töltjük fel).

Megjegyezzük, hogy az angol terminológia megkülönbözteti az összeadáskor keletkező **átvitelt** a kivonásnál keletkezőtől. Az előbbit carry-nek, míg az utóbbit borrow-nak nevezik.

Túlcsordulásról (overflow) beszélünk, ha a művelet eredménye már nem tárolható a kijelölt helyen. Ez az aritmetikai műveleteknél, pl. shifteléskor, szorzáskor, osztáskor fordulhat elő. Az 1 értékű átvitel mindig túlcsordulást jelez.

Egy előjeles bájt **előjeles kiterjesztésén** (sign extension) azt a műveletet értjük, mikor a bájtot szó méretű számmá alakítjuk át úgy, hogy mindkettő ugyanazt az értéket képviselje. Ezt úgy végezzük el, hogy a cél szó felső bájtjának minden bitjét a kiinduló bájt előjelbitjével töltjük fel. Tehát pl. a −3d = 11111101b szám előjeles kiterjesztése az 11111111 11111101b szám lesz, míg a +4d = 00000100b számból 00000000 00000100b lesz. Ezt a fogalmat általánosíthatjuk is: bájt kiterjesztése duplaszóvá, szó kiterjesztése duplaszóvá, kvadraszóvá stb.

Az előzővel rokon fogalom az **előjeltelen kiterjesztés** vagy más néven **zéró-kiterjesztés** (zero extension). Egy előjeltelen bájt előjeltelen kiterjesztésekor a bájtot olyan szóvá alakítjuk át, amely a bájttal megegyező értéket tartalmaz. Ehhez a cél szó hiányzó, felső bájtjának minden bitjét 0-val kell feltölteni. Tehát pl. a 5d = 00000101b szám előjeltelen kiterjesztése az 00000000 00000101b szám, míg a 128d = 10000000b számot 00000000 10000000b alakra hozzuk. Ez a fogalom is általánosítható és értelmezhető szavakra, duplaszavakra stb. is.

4. fejezet

A 8086-os processzor jellemzői, szolgáltatásai

Ismerkedjünk most meg az Intel 8086-os mikroprocesszorral közelebbről.

4.1. Memóriakezelés

A számítógép memóriáját úgy tudjuk használni, hogy minden egyes memóriarekeszt megszámozunk. Azt a módszert, ami meghatározza, hogy hogyan és mekkora területhez férhetünk hozzá egyszerre, **memória-szervezésnek** vagy **memória-modellnek** nevezzük. Több elterjedt modell létezik, közülük a legfontosabbak a lineáris és a szegmentált modellek.

Lineáris modellen (linear memory model) azt értjük, ha a memória teljes területének valamely bájtja egyetlen számmal megcímezhető (kiválasztható). Ezt az értéket ekkor **lineáris memóriacímnek** (linear memory address) nevezzük. Az elérhető (használható) lineáris címek tartományát egy szóval **címterületnek** (address space) hívjuk.

Szegmensen (segment) a memória egy összefüggő, rögzített nagyságú darabját értjük most (létezik egy kicsit másféle szegmens-fogalom is), ennek kezdőcíme (tehát a szegmens legelső bájtjának memóriacíme) a **szegmens báziscím** avagy **szegmenscím** (segment base address). A szegmensen belüli bájtok elérésére szükség van azok szegmenscímhez képesti relatív távolságára, ez az **offszetcím** (offset address). **Szegmentált modell** (segmented memory model) esetén a **logikai memóriacím** (logical memory address) tehát két részből tevődik össze: a szegmenscímből és az offszetcímből. E kettő érték már elegendő a memória lefedéséhez. Jelölés:

SZEGMENSCÍM:OFFSZETCÍM,

tehát először leírjuk a szegmenscímet, azután egy kettőspontot, majd az offszet jön.

A 8086-os processzor 20 bites címbusszal rendelkezik, tehát a memóriacímek 20 bitesek lehetnek. Ez 1 Mbájt (= $2^{20} = 1024 \cdot 1024$ bájt) méretű memória megcímzéséhez elegendő. A processzor csak a szegmentált címzést ismeri. A szegmensek méretét 64 Kbájtban szabták meg, mivel így az offszet 16 bites lesz, ez pedig belefér bármelyik regiszterbe (ezt majd később látni fogjuk). A szegmensek kezdőcímére is tettek azonban kikötést, mégpedig azt, hogy minden szegmens csak 16-tal osztható memóriacímen kezdődhet (ezek a címek az ú.n. **paragrafus-határok**, a **paragrafus** pedig egy 16 bájt hosszú memóriaterület). Ez annyit tesz, hogy minden

szegmenscím alsó 4 bitje 0 lesz, ezeket tehát felesleges lenne eltárolni. Így marad pont 16 bit a szegmenscímből, azaz mind a szegmens-, mind az offszetcím 16 bites lett. Nézzünk egy példát:

1234h:5678h

A szegmenscím felső 16 bitje 1234h, ezért ezt balra shifteljük 4-szer, így kapjuk az 1 2340h-t. Ehhez hozzá kell adni az 5678h számot. A végeredmény a 1 79B8h lineáris cím. Észrevehetjük, hogy ugyanazt a memóriarekeszt többféleképpen is megcímezhetjük, így pl. a 179Bh:0008h, 15AFh:1EC8h, 130Dh:48E8h címek mind az előző helyre hivatkoznak. Ezek közül van egy kitüntetett, a 179Bh:0008h. Ezt a címet úgy alkottuk meg, hogy a lineáris cím alsó négy bitje (1000b = 0008h) lesz az offszetcím, a felső 16 bit pedig a szegmenscímet alkotja. Ezért az ilyen címeket nevezhetjük bizonyos szempontból „normáltnak” (normalized address), hiszen az offszet itt mindig 0000h és 000Fh között lesz.

Most tekintsük a következő logikai címet:

0FFFFh:0010h

Az előbb leírtak szerint a szegmens bázicíme 0F FFF0h lesz, ehhez hozzá kell adni a 0 0010h offszetcímet. A műveletet elvégezve az 10 0000h lineáris memóriacím lesz az eredmény. Aki eléggé szemfüles, annak rögtön feltűnhet valami: ez a cím 21 bit hosszú! Ez azért furcsa, mert azt írtuk fenntebb, hogy a 8086-os processzor 20 bites memóriacímekkel tud dolgozni. Nos, ilyen esetekben a processzor „megszabadul” a legfelső, 21. bittől, tehát azt mindig 0-nak fogja tekinteni. (Matematikai szóhasználattal úgy fogalmazhatunk, hogy a processzor a lineáris címeket modulo 10 0000h képezi.) Így hiába 21 bites a lineáris cím, a memóriához ennek a címnek csak az alsó 20 bitje jut el. Ez azt eredményezi, hogy a 10 0000h és a 00 0000h lineáris címek ugyanarra a memóriarekeszre fognak hivatkozni. Ezt a jelenséget nevezik **címterület-körbefordulásnak** (address space wrap-around). Az elnevezés onnan ered, hogy a lineáris cím „megtörik”, „átesik”, „körbefordul” az 1 Mbájtos címterület felső határán.

Zárásként még egy fogalmat megemlítünk. **Fizikai memóriacím** (physical memory address) alatt azt a címet értjük, amit a memória a processzortól „megkap”. Kicsit pontosítva, a címbuszra kerülő értéket nevezzük fizikai címnek. A fizikai cím nem feltétlenül egyezik meg a lineáris címmel. Erre éppen az előbb láthattunk példát, hiszen a 10 0000h lineáris címhez a 00 0000h fizikai cím tartozik.

4.2. Regiszterek

Mint már a 2. fejezetben is említettük, a processzor (CPU) tartalmaz néhány speciális, közvetlenül elérhető tárolóhelyet, amiket **regisztereknek** nevezünk. (A „közvetlenül elérhető” jelző arra utal, hogy a regiszterek olvasásához/írásához nem kell a memóriához fordulnia a processzornak).

A 8086-os processzor összesen 14 db. 16 bites regiszterrel gazdálkodhat:

- 4 általános célú adatregiszter (AX, BX, CX, DX)
- 2 indexregiszter (SI és DI)
- 3 mutatóregiszter (SP, BP, IP)
- 1 státuszregiszter vagy Flags regiszter (SR vagy Flags)

- 4 szegmensregiszter (CS, DS, ES, SS)

Most nézzük részletesebben:

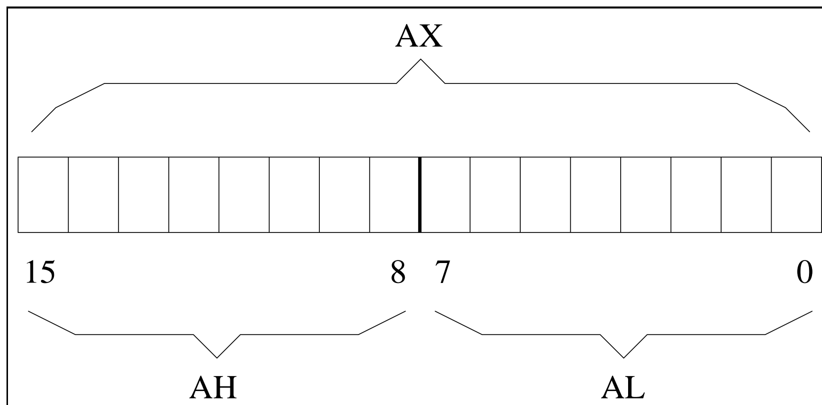
- AX (Accumulator) – Sok aritmetikai utasítás használja a forrás és/vagy cél tárolására.
- BX (Base) – Memóriacímzésnél bázisként szolgálhat.
- CX (Counter) – Sok ismétléses utasítás használja számlálóként.
- DX (Data) – I/O utasítások használják a portszám tárolására, ill. egyes aritmetikai utasítások számára is különös jelentőséggel bír.
- SI (Source Index) – Sztringkezelő utasítások használják a forrás sztring címének tárolására.
- DI (Destination Index) – A cél sztring címét tartalmazza.
- SP (Stack Pointer) – A verem tetejére mutat (azaz a verembe legutóbb berakott érték címét tartalmazza).
- BP (Base Pointer) – Általános pointer, de alapesetben a verem egy elemét jelöli ki.
- IP (Instruction Pointer) – A következő végrehajtandó utasítás memóriabeli címét tartalmazza. Közvetlenül nem érhető el, de tartalma írható és olvasható is a vezérlésátadó utasításokkal.
- SR avagy Flags (Status Register) – A processzor aktuális állapotát, az előző művelet eredményét mutató, ill. a proci működését befolyásoló biteket, ú.n. **flag-eket** (jelzőket) tartalmaz. Szintén nem érhető el közvetlenül, de manipulálható különféle utasításokkal.
- CS (Code Segment) – A végrehajtandó program kódját tartalmazó szegmens címe. Nem állítható be közvetlenül, csak a vezérlésátadó utasítások módosíthatják.
- DS (Data Segment) – Az alapértelmezett, elsődleges adatterület szegmensének címe.
- ES (Extra Segment) – Másodlagos adatszegmens címe.
- SS (Stack Segment) – A verem szegmensének címe.

A négy általános célú regiszter (AX, BX, CX és DX) alsó és felső nyolc bitje (azaz alsó és felső bájtja) külön neveken érhető el: az alsó bájtokat az AL, BL, CL, DL, míg a felső bájtokat az AH, BH, CH, DH regiszterek jelölik, amint ezt a 4.1. ábra is mutatja (a rövidítésekben L – Low, H – High). Éppen ezért a következő (Pascal-szerű) műveletek:

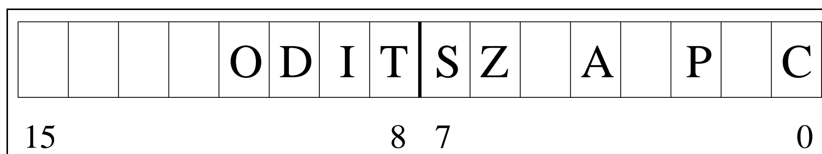
```
AX:=1234h
AL:=78h
AH:=56h
```

végrehajtása után AX tartalma 5678h lesz, AH 56h-t, AL pedig 78h-t fog tartalmazni.

A Flags regiszter felépítése a 4.2. ábrán látható. Az egyes bitek a következő információt szolgáltatják:



4.1. ábra. Az AX regiszter szerkezete



4.2. ábra. A Flags regiszter szerkezete

- C (Carry) – 1, ha volt aritmetikai átvitel az eredmény legfelső bitjénél (előjeltelen aritmetikai túlcsordulás), 0, ha nem.
- P (Parity even) – 1, ha az eredmény legalsó bájtja páros számú 1-es bitet tartalmaz, különben 0.
- A (Auxilliary carry, Adjust) – 1 jelzi, ha volt átvitel az eredmény 3-as és 4-es bitje (tehát az alsó és a felső nibble) között.
- Z (Zero) – Értéke 1, ha az eredmény zérus lett.
- S (Sign) – Értéke az eredmény legfelső bitjének, az előjelbitnek a tükörképe.
- T (Trap) – Ha 1, akkor a lépésenkénti végrehajtás (single-step execution) engedélyezve van.
- I (Interrupt enable) – Ha 1, akkor a maszkolható hardver-megszakítások engedélyezettek.
- D (Direction) – Ha 0, akkor a sztringutasítások növelik SI-t és/vagy DI-t, különben csökkentés történik.
- O (Overflow) – Ha 1, akkor előjeles aritmetikai túlcsordulás történt.

Ha hivatkozunk valamelyik flag-re, akkor a fenti betűjelekhez még hozzáírjuk az „F” betűt is. A CF, IF és DF flag-ek értékét közvetlenül is befolyásolhatjuk. Ezért pl. a CF nem csak akkor lehet 1, ha volt túlcsordulás (átvitel), hanem saját célból egyéb dolgot is jelezhet.

A Flags regiszter 1, 3, 5, 12, 13, 14 és 15 számú bitjei fenntartottak. Ezek értéke gyárilag rögzített, így nem is módosíthatjuk őket.

Aritmetikai flag-ek alatt, ha külön nem mondjuk, a CF, PF, AF, ZF, SF és OF flag-eket értjük.

Ha azt akarjuk kifejezni, hogy két vagy több regiszter tartalmát egymás után fűzve akarunk megadni egy értéket, akkor az egyes regisztereket egymástól kettősponttal elválasztva soroljuk fel, szintén a helyiértékes felírást követve. Tehát pl. a DX:AX jelölés azt jelenti, hogy a 32 bites duplaszónak az alsó szava AX-ben, felső szava DX-ben van (a leírási sorrend megfelel a bitek sorrendjének a bájtokban). Röviden: a DX:AX **regiszterpár** (register pair) egy 32 bites duplaszót tartalmaz. Ez a jelölés azonban bizonyos esetekben mást jelent, mint a memóriacímek ábrázolására használt SZEGMENS:OFFSZET felírások! Tehát a DX:AX jelölés jelenthet egy logikai memóriacímet DX szegmenssel és AX offszettel, de jelentheti azt is, hogy mi egy 32 bites értéket tárolunk a nevezett két regiszterben (ami azonban lehet egy lineáris cím is), és ennek alsó szavát AX, felső szavát pedig DX tartalmazza.

Ha **általános (célú) regiszterekről** beszélünk, akkor általában nemcsak az AX, BX, CX és DX regiszterekre gondolunk, hanem ezek 8 bites párjaira, továbbá az SI, DI, SP, BP regiszterekre is.

A következő végrehajtásra kerülő utasítás (logikai) címét a CS:IP, míg a verem tetejét az SS:SP regiszterpárok által meghatározott értékek jelölik.

A CS és IP regisztereken kívül mindegyik regiszter tartalmazhat bármilyen értéket, tehát nem csak az eredeti funkciójának megfelelő tartalommal tölthetjük fel őket. Ennek ellenére az SS, SP és Flags regiszterek más célú használata nem javasolt.

4.3. Adattípusok

A szó méretét 2 bájtban állapították meg az Intel-nél, ezért pl. a BX és DI regiszterek szavasak, BL és DH bájtosak, míg az 1234 5678h szám egy duplaszó.

A processzor csak egészekkel tud dolgozni, azon belül ismeri az előjeles és előjeltelen aritmetikát is. A több bájt hosszú adatokat little-endian módon tárolja, ezért pl. a fenti 1234 5678h szám a következő módon lesz eltárolva: a legalacsonyabb memóriacímre kerül a 78h bájt, ezt követi az 56h, majd a 34h, végül pedig a 12h. Logikai, lebegőpontos valós típusokat nem támogat a processzor!

Az egészek részhalmazát alkotják a **binárisan kódolt decimális egész számok** (Binary Coded Decimal numbers – BCD numbers). Ezek két csoportba sorolhatók: vannak **pakolt** (packed) és **pakolatlan BCD számok** (unpacked BCD number). (Használják még a „csomagolt” és „kicsomagolt” elnevezéseket is.) Pakolt esetben egy bájtban két decimális számjegyet tárolnak úgy, hogy a 4 – 7 bitek a szám magasabb helyiértékű jegyét, míg a 0 – 3 bitek az alacsonyabb helyiértékű jegyet tartalmazzák. A számjegyeket a 0h – 9h értékek valamelyike jelöli. Pakolatlan esetben a bájtnak a felső fele kihasználatlan, így csak 1 jegyet tudunk 1 bájtban tárolni.

A **sztring** (string) adattípus bájtok vagy szavak véges hosszú folytonos sorát jelenti. Ezzel a típussal bővebben egy későbbi fejezetben foglalkozunk.

Speciális egész típus a **mutató** (pointer). Mutatónak hívunk egy értéket, ha az egy memóriacímet tartalmaz, azaz ha azt a memória elérésénél felhasználjuk. Két típusa van: a **közeli** vagy **rövid mutató** (near, short pointer) egy offszetcímet jelent, míg **távoli**, **hosszú** vagy **teljes mutatón** (far, long, full pointer) egy teljes logikai memóriacímet, tehát szegmens:offszet alakú címet értünk. A közeli mutató hossza 16 bit, a távolié pedig 32 bit. Fontos, hogy mutatóként bármilyen értéket felhasználhatunk. Szintén jó, ha tudjuk, hogy a távoli mutatóknak az offszet része helyezkedik el az alacsonyabb memóriacímen a little-endian tárolásnak megfelelően, amit a szegmens követ 2-vel magasabb címen.

4.4. Memóriahivatkozások, címzési módok

Láttuk, hogy a memória szervezése szegmentált módon történik. Most lássuk, ténylegesen hogy adhatunk meg memóriahivatkozásokat.

Azokat a, regiszterek és konstans számok (kifejezések) kombinációjából álló jelöléseket, amelyek az összes lehetséges szabályos memóriacímzési esetet reprezentálják, **címzési módoknak** (addressing mode) nevezzük. Több típusuk van, és mindenhol használható mindegyik, ahol valamilyen memóriaoperandust meg lehet adni. A memóriahivatkozás jelzésére a szögletes zárójeleket ([és]) használjuk.

4.4.1. Közvetlen címzés

A címzés alakja [offs16], ahol offs16 egy 16 bites abszolút, szegmensen belüli offszetet (rövid mutatót) jelöl.

4.4.2. Báziscímzés

A címzés egy bázisregisztert használ az offszet megadására. A lehetséges alakok: [BX], [BP]. A cél bájt offszetcímét a használt bázisregiszterből fogja venni a processzor.

4.4.3. Indexcímzés

Hasonló a báziscímzéshez, működését tekintve is annak tökéletes párja. Alakjai: [SI], [DI].

4.4.4. Bázis+relatív címzés

Ide a [BX+rel8], [BX+rel16], [BP+rel8], [BP+rel16] formájú címmegadások tartoznak. A rel16 egy előjeles szó, rel8 pedig egy előjeles bájt, amit a processzor előjelesen kiterjeszt szóvá. Ezek az ú.n. **eltolások** (displacement). Bármelyik változatnál a megcímzett bájt offszetjét a bázisregiszter tartalmának és az előjeles eltolásnak az összege adja meg.

4.4.5. Index+relatív címzés

Hasonlóan a báziscímzés/indexcímzés pároshoz, ez a bázis+relatív címzési mód párja. Alakjai: [SI+rel8], [SI+rel16], [DI+rel8], [DI+rel16].

4.4.6. Bázis+index címzés

A két nevezett címzési mód keveréke. A következő formákat öltheti: [BX+SI], [BX+DI], [BP+SI] és [BP+DI]. A regiszterek sorrendje természetesen közömbös, így [BP+SI] és [SI+BP] ugyanazt jelenti. A cél bájt offszetjét a két regiszter értékének összegeként kapjuk meg.

4.4.7. Bázis+index+relatív címzés

Ez adja a legkombináltabb címzési lehetőségeket. Általános alakjuk [bázis+index+rel8] és [bázis+index+rel16] lehet, ahol bázis BX vagy BP, index SI vagy DI, rel8 és rel16 pedig előjeles bájt ill. szó lehet.

A bázis+relatív és index+relatív címzési módok másik elnevezése a **bázisrelatív** ill. **index-relatív** címzés.

A közvetlen címzésben szereplő abszolút offszetet, illetve a többi címzésnél előforduló eltolást nem csak számmal, de numerikus kifejezéssel is megadhatjuk.

Megjegyezzük, hogy az említett címzési módoknak létezik egy alternatív alakja is. Ez annyiban tér el az előzőktől, hogy a többtagú címzések tagjait külön-külön szögletes zárójelekbe írjuk, továbbá elhagyjuk a „+" jeleket. (Az eltolásban esetlegesen szereplő „-" jelre viszont szükség van.) Az eltolást nem kötelező zárójelbe tenni. A következő felsorolásban az egy sorban levő jelölések ekvivalensek:

- [BX+rel16], [BX][rel16], [BX]rel16
- [DI+rel8], [DI][rel8], rel8[DI]
- [BP+SI], [BP][SI]
- [BX+SI+rel16], [BX][SI][rel16], rel16[BX][SI]

Minden címzési módhoz tartozik egy alapértelmezett (default) szegmensregiszter-előírás. Ez a következőket jelenti:

- bármely, BP-t *nem* tartalmazó címzési mód a DS-t fogja használni
- a BP-t tartalmazó címzési módok SS-t használják

Ha az alapértelmezett beállítás nem tetszik, akkor mi is előírhatjuk, hogy melyik szegmensregisztert használja a cím meghatározásához a processzor. Ezt a **szegmensfelülbíráló prefixekkel** tehetjük meg. A prefixek alakja a következő: CS:, DS:, ES:, SS:, tehát a szegmensregiszter neve plusz egy kettőspont. Ezeket a prefixeket legjobb közvetlenül a címzési mód jelölése elé írni.

Most lássunk néhány példát szabályos címzésekre:

- [12h*34h+56h] (közvetlen címzés)
- ES:[09D3h] (közvetlen címzés, szegmensfelülbírálással)
- CS:[SI-500d] (indexrelatív címzés, szegmensfelülbírálással)
- SS:[BX+DI+1999d] (bázis+index+relatív címzés, szegmensfelülbírálással)
- DS:[BP+SI] (bázis+index címzés, szegmensfelülbírálással)

A következő jelölések viszont hibásak:

- [AX] (az AX regiszter nem használható címzésre)
- [CX+1234h] (a CX regiszter sem használható címzésre)
- [123456789ABCh] (túl nagy az érték)
- [SI+DI] (két indexregiszter nem használható egyszerre)
- [SP+BP] (az SP regiszter nem használható címzésre)
- [IP] (az IP regiszter nem érhető el)

A használt címzésmód által hivatkozott logikai memóriacímet (ill. gyakran csak annak offszet részét) **tényleges** vagy **effektív címnek** (effective address) nevezzük. Így pl. ha BX értéke 1000h, akkor a [BX+0500h] címzésmód effektív címe a DS:1500h, hiszen bázisrelatív címzéskor a bázisregiszter (BX) és az eltolás (0500h) értéke összeadódik egy offszetcímet alkotva, továbbá a korábbi megjegyzés alapján a szegmenscímet DS tartalmazza.

4.5. Veremkezelés

Vermen (stack) a következő tulajdonságokkal rendelkező adatszerkezetet értjük:

- értelmezve van rajta egy Push művelet, ami egy értéket rak be a verem tetejére
- egy másik művelet, a Pop, a verem tetején levő adatot olvassa ki és törli a veremből
- működése a **LIFO** (Last In First Out) elvet követi, azaz a legutoljára betett adatot vehetjük ki először a veremből
- verem méretén azt a számot értjük, ami meghatározza, hogy maximálisan mennyi adatot képes eltárolni
- verem magassága a benne levő adatok számát jelenti
- ha a verem magassága 0, akkor a verem üres
- ha a verem magassága eléri a verem méretét, akkor a verem tele van
- üres veremből nem vehetünk ki adatot
- teli verembe nem rakhatunk be több adatot

Az Intel 8086-os processzor esetén a verem mérete maximálisan 64 Kbájt lehet. Ez annak a következménye, hogy a verem csak egy szegmensnyi területet foglalhat el, a szegmensek mérete pedig szintén 64 Kbájt. A verem szegmensét az SS regiszter tárolja, míg a verem tetejére az SP mutat. Ez a verem **lefelé bővülő** (expand-down) verem, ami azt jelenti, hogy az újonnan betett adatok egyre alacsonyabb memóriacímen helyezkednek el, és így SP értéke is folyamatosan csökken a Push műveletek hatására. SP-nek mindig páros értéket kell tartalmaznia, és a verembe betenni ill. onnan kiolvasni szintén csak páros számú bájtot lehet. (Ez egy kis füllentés, de a legjobb, ha követjük ezt a tanácsot.)

A lefelé bővülésből következik, hogy SP aktuális értékéből nem derül ki, hány elem is van a veremben (azaz milyen magas a verem). Ugyanígy nem tudható, mekkora a verem mérete. Ehhez szükség van a verem alja kezdőcímének (azaz SP legnagyobb használható értékének) ismeretére. Az az egy biztos, hogy egészen addig pakolhatunk elemet a verembe, amíg SP el nem éri a 0000h-t.

A Push és Pop műveleteknek megfelelő utasítások mnemonikja szintén PUSH ill. POP, így megjegyzésük nem túl nehéz. A PUSH először csökkenti SP-t a betenni kívánt adat méretének megfelelő számú bájttal, majd az SS:SP által mutatott memóriacímre beírja a kérdéses értéket. A POP ezzel ellentétes működésű, azaz először kiolvas az SS:SP címről valahány számú bájtot, majd SP-hez hozzáadja a kiolvasott bájtok számát.

Nézzünk egy példát a verem működésének szemléltetésére! Legyen AX tartalma 1212h, BX tartalma 3434h, CX pedig legyen egyenlő 5656h-val. Tekintsük a következő Assembly programrészletet:

```
PUSH    AX
PUSH    BX
POP     AX
PUSH    CX
POP     BX
POP     CX
```

SP legyen 0100h, SS értéke most közömbös. A verem és a regiszterek állapotát mutatják a következő ábrák az egyes utasítások végrehajtása után:

Kezdetben:

	SS:0102h : ??
SP⇒	SS:0100h : ??
	SS:00FEh : ??
	SS:00FCh : ??
	SS:00FAh : ??

AX = 1212h, BX = 3434h, CX = 5656h

PUSH AX után:

	SS:0102h : ??
	SS:0100h : ??
SP⇒	SS:00FEh : 1212h
	SS:00FCh : ??
	SS:00FAh : ??

AX = 1212h, BX = 3434h, CX = 5656h

PUSH BX után:

	SS:0102h : ??
	SS:0100h : ??
	SS:00FEh : 1212h
SP⇒	SS:00FCh : 3434h
	SS:00FAh : ??

AX = 1212h, BX = 3434h, CX = 5656h

POP AX után:

	SS:0102h : ??
	SS:0100h : ??
SP⇒	SS:00FEh : 1212h
	SS:00FCh : 3434h
	SS:00FAh : ??

AX = 3434h, BX = 3434h, CX = 5656h

PUSH CX után:

	SS:0102h : ??
	SS:0100h : ??
	SS:00FEh : 1212h
SP⇒	SS:00FCh : 5656h
	SS:00FAh : ??

AX = 3434h, BX = 3434h, CX = 5656h

POP BX után:

	SS:0102h : ??
	SS:0100h : ??
SP⇒	SS:00FEh : 1212h
	SS:00FCh : 5656h
	SS:00FAh : ??

AX = 3434h, BX = 5656h, CX = 5656h

POP CX után:

	SS:0102h : ??
SP⇒	SS:0100h : ??
	SS:00FEh : 1212h
	SS:00FCh : 5656h
	SS:00FAh : ??

AX = 3434h, BX = 5656h, CX = 1212h

Az utasítások végrehajtása tehát azt eredményezi, hogy AX felveszi BX értékét, BX a CX értékét, CX pedig az AX eredeti értékét (olyan, mintha „körbeforgattuk” volna a regiszterek tartalmát egymás között). Mivel ugyanannyi PUSH volt, mint amennyi POP, SP értéke a végrehajtás után visszaállt a kezdetire.

Figyeljük meg, hogy a PUSH-sal berakott értékek a memóriában bentmaradnak akkor is, ha később egy POP-pal kivesszük őket, bár ekkor már nem részei a veremnek (azaz nincsenek a veremben). Ez nem túl meglepő dolog, ha figyelembe vesszük az utasítások működését.

A verem sok dologra használható a programozás során:

- regiszterek, változók értékének megcserélésére
- regiszterek ideiglenes tárolására
- lokális változók elhelyezésére
- eljáráshíváskor az argumentumok átadására, ill. a visszatérési érték tárolására

A veremregisztereknek (SS és SP) mindig érvényes értékeket kell tartalmazniuk a program futásakor, mivel azt a processzor és más programok (pl. a hardvermegszakításokat lekezelő rutinok) is használják.

4.6. I/O, megszakítás-rendszer

Ez a processzor 16 biten ábrázolja a portszámokat, így összesen 65536 db. portot érhetünk el a különféle I/O (Input/Output – ki-/bemeneti) utasításokkal. Ezek közül a 0000h és 00FFh közötti tartományt az alaplapi eszközök használják, és mivel a legfontosabb perifériák címét rögzítették, a különböző PC-s rendszerekben ezeket ugyanazon a porton érhetjük el. Bármelyik porton kezdeményezhetünk olvasást és írást egyaránt, de ezzel azért nem árt vigyázni, egy félresikeredett írással ugyanis könnyen „megadásra" késztethetjük a számítógépet.

Egy megszakítást kiválthat egy hardvereszköz (ez a **hardver-megszakítás**), de a felhasználó és a programok is elindíthatnak ilyen eseményt (ez a **szoftver-megszakítás**). A megszakítások két típusa nem csak a számukban, de tulajdonságaikban is eltér egymástól.

A hardver-megszakításokat a megszakítás-vezérlőn keresztül érzékeli a processzor. A vezérlő 8 különböző megszakítás-vonalat tart fent (AT-k esetén két vezérlőt kötöttek sorba, így lett a 8-ból 16 vonal). Ezen megszakításokat az IRQ0, IRQ1, ..., IRQ7 (továbbá IRQ8, ..., IRQ15) szimbólumokkal jelöljük. Fontos jellemzőjük, hogy a program futásával párhuzamosan aszinkron keletkeznek, és hogy **maszkolhatók**. Ez utóbbi fogalom azt takarja, hogy az egyes IRQ vonalakat egyenként engedélyezhetjük ill. tilthatjuk le. Ha egy vonal tiltva (másképpen maszkolva) van, akkor az arra vonatkozó kérések nem jutnak el a processzorhoz. Ezen kívül a processzor Flags regiszterének IF bitjével az összes hardver-megszakítás érzékelését letilthatjuk ill. újra engedélyezhetjük.

Ezzel szemben szoftver-megszakításból 256 darab van, jelölésükre az INT 00h, ..., INT 0FFh kifejezéseket használjuk. A program futása során szinkron keletkeznek (hiszen egy gépi kódú utasítás váltja ki őket), és nem maszkolhatók, nem tilthatók le. Az IF flag állásától függetlenül mindig érzékeli őket a CPU.

A két megszakítás-típus lekezelésében közös, hogy a CPU egy speciális rutint (programrészt) hajt végre a megszakítás detektálása után. Ez a rutin a **megszakítás-kezelő** (interrupt handler). Az operációs rendszer felállása után sok hardver- és szoftver-megszakításnak már van kezelője (pl. a BIOS is jópár ilyen megszakítást üzemeltet), de lehetőség van arra is, hogy bármelyik ilyen rutint lecseréljük egy saját programunkra.

A hardver-megszakítások a szoftveresek egy adott tartományára képződnek le, ez alapállapotban az INT 08h, ..., INT 0Fh (IRQ8-tól IRQ15-ig pedig az INT 70h, ..., INT 77h) megszakításokat jelenti, tehát ezek a megszakítások másra nem használhatók. Így pl. IRQ4 keletkezésekor a processzor az INT 0Ch kezelőjét hívja meg, ha IF = 1 és az IRQ4 nincs maszkolva.

A megszakítások egy speciális csoportját alkotják azok, amiket valamilyen vészhelyzet, súlyos programhiba vált ki. Ezeket összefoglaló néven **kivételnek** (exception) nevezzük. Kivételt okoz pl. ha nullával próbálunk osztani, vagy ha a verem túl- vagy alulcsordul.

5. fejezet

Az Assembly nyelv szerkezete, szintaxisa

Az Assembly nyelven megírt programunkat egy szöveges forrásfájlban tároljuk, amit aztán egy erre specializált fordítóprogrammal átalakítunk **tárgykóddá** (object code). A tárgykód még nem futtatható közvetlenül, ezért a **szerkesztő** (linker) elvégzi rajta az átalakításokat. Végül az egy vagy több tárgykód összefűzése után kialakul a futtatható program. Az említett fordítóprogramot **assemblernek** nevezzük. A forrásfájlok állománynév-kiterjesztése általában .ASM vagy .INC, a tárgykódé .OBJ, a futtatható fájloké pedig .COM vagy .EXE. Ezek az elnevezések a PC-s DOS rendszerekre vonatkoznak, más operációs rendszerek alatt (pl. UNIX, OS/2, Linux) ezek eltérhetnek. Az Assembly forrásfájloknak tiszta szöveges (text) állományoknak kell lennie, mert az assemblerek nem szeretik a mindenféle „bináris szemetet" tartalmazó dokumentumokat.

Az Assembly nem érzékeny a kis- és nagybetűkre, így tehát a PUSH, Push, push, PusH szavak ugyanazt a jelentést képviselik. Ez elég nagy szabadságot ad a különböző azonosítók elnevezésére, és a forrást is áttekinthetőbbé teszi.

Az Assembly forrás sorai a következő elemeket tartalmazhatják:

- assembler utasítások (főleg ezek alkotják a tényleges program kódját)
- pszeudo utasítások (pl. makrók, helyettesítő szimbólumok)
- assembler direktívák

A **direktívák** (directive) olyan, az assemblernek szóló utasítások, melyek közvetlenül gépi kódot nem feltétlenül generálnak, viszont a fordítás menetét, az elkészült kódot befolyásolják.

Az érvényes assembler utasítások szintaxisa a következő:

{Címke:} <Prefix> {Utasítás {Operandus<,Operandus>}} {;Megjegyzés}

A kapcsos zárójelek ({ és }) az opcionális (nem kötelező) részeket jelzik, míg a csúcsos zárójelek (< és >) 0 vagy több előfordulásra utalnak. Ezek alapján az assembler utasítások szintaxisa szavakban a következő: a sor elején állhat egy Címke, amit követhet valahány Prefix, majd írhatunk legfeljebb egy Utasítást, aminek valahány Operandusa lehet, s végül a sort egy Megjegyzés zárhatja. A megfogalmazás mutatja, hogy a csak egy címkét, megjegyzést tartalmazó sorok, sőt még az üres sor is megengedett az Assembly forrásban.

A **címke** (label) deklarációja az azonosítójából és az azt követő kettőspontból áll. Értéke az aktuális sor memóriacíme (helyesebben offszetje). Szerepe azért fontos, mert segítségével könnyen írhatjuk le a különféle vezérlési szerkezeteket. A gépi kódú utasítások ugyanis relatív vagy abszolút memóriacímekkel dolgoznak, nekünk viszont kényelmesebb, ha egy szimbólummal (a címke nevével) hivatkozunk a kérdéses sorra. A tényleges cím megállapítása a szerkesztő feladata lesz.

A **prefix** olyan utasításelem, amely csak az őt követő gépi kódú utasításra (avagy Assembly mnemonikra) hat, annak működését változtatja meg ill. egészíti ki.

Ezeket követi az **utasítás** (instruction), ami egy assembler direktíva, egy pszeudo utasítás vagy egy Assembly mnemonik lehet. Emlékeztetőül, a **mnemonik** (mnemonic) azonos műveletet végző különböző gépi kódú utasítások csoportját jelölő szimbólum (szó). Az Intel mnemonikok legalább kétbetűsek, és némelyik még számot is tartalmaz. A mnemonikok gépi kódú alakját **műveleti kódnak** (operation code, opcode) nevezzük.

A továbbiakban a „mnemonik” szó helyett inkább az „utasítást” fogjuk használni.

Az mnemonik által meghatározott utasítás valamilyen „dolgokon” hajtja végre a feladatát. Ezeket a dolgokat mutatják az **operandusok** (operand), amelyeket (ha több van) vesszővel elválasztva sorolunk fel. Operandusok építőkövei a következők lehetnek:

- regiszterek
- numerikus (szám) konstansok
- karakteres (sztring) konstansok
- szimbólumok
- operátorok (műveletek)

Regiszterekre a következő szavakkal utalhatunk: AL, AH, AX, BL, BH, BX, CL, CH, CX, DL, DH, DX, SI, DI, SP, BP, CS, DS, ES és SS.

A **numerikus konstansnak** számjeggyel kell kezdődnie, azután tartalmazhatja a tíz számjegyet ill. az A – F betűket, a végén pedig a számrendszerre utaló jelölés állhat. Alapesetben minden számot decimálisnak értelmez az assembler, kivéve ha:

- átállítottuk az alapértelmezést a RADIX direktívával
- a szám végén a „d” (decimális), „b” (bináris), „h” (hexadecimális) vagy „o” (oktális) betű áll

Fontos, hogy a betűvel kezdődő hexadecimális számok elé tegyünk legalább egy 0-t, mert különben szimbólumnak próbálná értelmezni a számunkat az assembler.

Karakteres konstansokat aposztrófok('. . . ') vagy idézőjelek (". . . ") között adhatunk meg. Ha numerikus konstans helyett állnak, akkor lehetséges hosszuk az adott direktívától (pl. a DW esetén 2 bájt) ill. a másik operandustól függ.

Szimbólumok például a konstans azonosítók, változók, címkék, szegmensek, eljárások nevei. Szimbólum azonosítója tartalmazhat számjegyet, betűket (csak az angol ábécé betűit), aláhúzásjelet (_), dollárjelet ($) és „kukacot” (@). Fontos, hogy az azonosítók *nem* kezdődhetnek számjeggyel!

Speciális szimbólum az egymagában álló dollárjel, ennek neve pozíció-számláló (location counter). Értéke az aktuális sor szegmensbeli (avagy szegmenscsoportbeli) offszetje.

Operátorból rengeteg van, a legfontosabbak talán a következők:

- kerek zárójelek
- szokásos aritmetikai műveleti jelek (+, -, *, /, MOD)
- mezőkiválasztó operátor (.)
- szegmens-előírás/-felülbírálás (:)
- memóriahivatkozás ([...])
- bitenkénti logikai műveletek (AND, OR, XOR, NOT)
- típusfelülbírálás (PTR)
- adattípus megadás (BYTE, WORD, DWORD)
- duplikáló operátor (DUP)
- relációs operátorok (EQ, NE, GT, GE, LT, LE)
- pointer típusok (NEAR, FAR)
- szegmens/offszet lekérdezés (SEG, OFFSET)
- léptető operátorok (SHL, SHR)
- méretlekérdező operátorok (LENGTH, SIZE, TYPE, WIDTH)

Az operandusok ezek alapján három csoportba sorolhatók: regiszter-, memória- és konstans (közvetlen értékű) operandusok. A különböző mnemonikok lehetnek nulla-, egy-, kettő-, három- vagy négyoperandusúak.

Fontos megjegyezni, hogy a kétoperandusú mnemonikok első operandusa a *cél*, a második pedig a *forrás*. Ezt a konvenciót egyébként csak az Intel Assembly használja, az összes többi (pl. Motorola) Assemblyben az első a forrás, a második pedig a céloperandus. Az elnevezések arra utalnak, hogy a művelet eredménye a céloperandusban tárolódik el. Ha szükséges, a céloperandus második forrásként is szolgálhat. Egy operandust használó utasításoknál az az egy operandus sokszor egyben forrás és cél is.

Végül az assembler utasítás sorának végén **megjegyzést** is írhatunk egy pontosvessző után. Ezeket a karaktereket az assembler egyszerűen átugorja, így tehát bármilyen szöveget tartalmazhat (még ékezetes betűket is). A megjegyzés a pontosvesszőtől a sor végéig tart.

Most tisztáznunk kell még néhány fogalmat. Korábban már említettük, hogy a processzor szegmentált memória-modellt használ. Hogy teljes legyen a kavarodás, az assemblernek is elő kell írnunk a memória-modellt, és szegmenseket is létre kell hoznunk. A két szegmens- és memória-modell fogalmak azonban egy picit különböznek.

Az assemblerben létrehozott szegmenseknek van nevük (azonosítójuk, ami tehát egy szimbólum), méretük legfeljebb 64 Kbájt lehet, és nem feltétlenül kell összefüggőnek lennie egy szegmens definíciójának. Ez utóbbi tulajdonság több szabadságot enged a programozónak, mivel megteheti, hogy elkezd egy szegmenst, majd definiál egy másikat, azután megint visszatér az első szegmensbe. Az azonos néven megkezdett szegmensdefiníciókat az assembler és a szerkesztő szépen összefűzi egyetlen darabbá, tehát ez a jelölésrendszer a processzor számára átlátszó lesz. Másrészt a méretre vonatkozó kijelentés mindössze annyit tesz, hogy ha nem töltünk ki teljesen (kóddal vagy adatokkal) egy memóriaszegmenst (ami viszont 64 Kbájt nagyságú),

akkor a fennmaradó bájtokkal az assembler nem foglalkozik, a program futásakor ezek valami „szemetet” fognak tartalmazni.

A memória-modell előírása akkor fontos, ha nem akarunk foglalkozni különböző szegmensek (adat-, kód-, verem- stb.) létrehozásával, és ezt a folyamatot az assemblerre akarjuk hagyni. Ehhez viszont jeleznünk kell az assemblernek, hogy előreláthatólag mekkora lesz a programunk memóriaigénye, hány és milyen típusú szegmensekre van szükségünk. Ezt a típusú szegmensmegadást **egyszerűsített szegmensdefiníciónak** (simplified segment definition) nevezzük.

A két módszert keverve is használhatjuk kedvünk és igényeink szerint, nem fogják egymást zavarni (mi is ezt fogjuk tenni). Ha magas szintű nyelvhez írunk külső Assembly rutinokat, akkor az egyszerűsített szegmensmegadás sokszor egyszerűbb megoldást szolgáltat.

Most pedig lássunk néhány fontos direktívát:

- modul/forrásfájl lezárása (END)
- szegmens definiálása (SEGMENT...ENDS)
- szegmenscsoport definiálása (GROUP)
- szegmens hozzárendelése egy szegmensregiszterhez (ASSUME)
- értékadás a $ szimbólumnak (ORG)
- memória-modell megadása (MODEL)
- egyszerűsített szegmensdefiníciók (CODESEG, DATASEG, FARDATA, UDATASEG, UFARDATA, CONST, STACK, .CODE, .DATA, .STACK)
- helyfoglalás (változó létrehozása) (DB, DW, DD, DF, DQ, DT)
- konstans/helyettesítő szimbólum létrehozása (=, EQU)
- címke létrehozása (LABEL)
- eljárás definiálása (PROC...ENDP)
- külső szimbólum definiálása (EXTRN)
- szimbólum láthatóvá tétele a külvilág számára (PUBLIC)
- feltételes fordítás előírása (IF, IFccc, ELSE, ELSEIF, ENDIF)
- külső forrásfájl beszúrása az aktuális pozícióba (INCLUDE)
- felhasználói típus definiálása (TYPEDEF)
- struktúra, unió, rekord definiálása (STRUC...ENDS, UNION...ENDS, RECORD)
- a számrendszer alapjának átállítása (RADIX)
- makródefiníció (MACRO...ENDM)
- makróműveletek (EXITM, IRP, IRPC, PURGE, REPT, WHILE)
- utasításkészlet meghatározása (P8086, P186, P286, P386 stb.; .8086, .186, .286, .386 stb.)

A direktívák használatára később még visszatérünk.

Assemblerből több is létezik PC-re, ilyenek pl. a Microsoft Macro Assembler (MASM), Turbo Assembler (TASM), Netwide Assembler (NASM), Optimizing Assembler (OPTASM). Ebben a jegyzetben a programokat a TASM jelölésmódban írjuk le, de több-kevesebb átírással másik assembler alá is átvihetők a források. A TASM specialitásait nem fogjuk kihasználni, ahol nem muszáj. A TASM saját linkert használ, ennek neve Turbo Linker (TLINK).

6. fejezet

A 8086-os processzor utasításkészlete

Itt az ideje, hogy megismerkedjünk a kiválasztott processzorunk által ismert utasításokkal (pontosabban mnemonikokkal). Az utasítások közös tulajdonsága, hogy az operandusoknak teljesíteniük kell a következő feltételeket:

- Ha egyetlen operandusunk van, akkor az általában csak az általános regiszterek közül kerülhet ki, vagy pedig memóriahivatkozás lehet (néhány esetben azonban lehet konstans kifejezés vagy szegmensregiszter is).
- Két operandus esetén bonyolultabb a helyzet: mindkét operandus egyszerre nem lehet szegmensregiszter, memóriahivatkozás és közvetlen (konstans) érték, továbbá szegmensregiszter mellett vagy általános regiszternek vagy memóriahivatkozásnak kell szerepelnie. (A sztringutasítások kivételek, ott mindkét operandus memóriahivatkozás lesz, de erről majd később.) A két operandus méretének majdnem mindig meg kell egyeznie, ez alól csak néhány speciális eset kivétel.

6.1. Prefixek

Először lássuk a prefixeket, zárójelben megadva a gépi kódjukat is (ez utóbbit persze nem kell megjegyezni):

6.1.1. Szegmensfelülbíráló prefixek

Ezek a CS: (2Eh), DS: (3Eh), ES: (26h) és SS: (36h). A TASM ezeken kívül ismeri a SEGCS, SEGDS, SEGES és SEGSS mnemonikokat is. A SCAS és a STOS utasítások kivételével bármely más utasítás előtt megadva őket az alapértelmezés szerinti szegmensregiszter helyett használandó regisztert adják meg. Természetesen ez csak a valamilyen memóriaoperandust használó utasításokra vonatkozik.

6.1.2. Buszlezáró prefix

Mnemonikja LOCK (0F0h). Hatására az adott utasítás által módosított memóriarekeszt az utasítás végrehajtásának idejére a processzor zárolja úgy, hogy a buszokhoz más hardverelem nem férhet hozzá a művelet lefolyásáig. (A teljesség kedvéért: ebben a processzor LOCK# jelű kivezetése vesz részt.) Csak a következő utasítások esetén használható: ADD, ADC, SUB, SBB, AND, OR, XOR, NOT, NEG, INC, DEC, XCHG, más esetekben hatása definiálatlan. Az említett utasítás céloperandusának kötelezően memóriahivatkozásnak kell lennie.

6.1.3. Sztringutasítást ismétlő prefixek

Lehetséges alakjaik: REP, REPE, REPZ (0F3h), REPNE, REPNZ (0F2h). Csak a sztringkezelő utasítások előtt használhatók, más esetben következményük kiszámíthatatlan. Működésükről és használatukról később szólunk.

6.2. Utasítások

Most jöjjenek az utasítások, típus szerint csoportosítva:

6.2.1. Adatmozgató utasítások

- MOV – adatok mozgatása
- XCHG – adatok cseréje
- PUSH – adat betétele a verembe
- PUSHF – Flags regiszter betétele a verembe
- POP – adat kivétele a veremből
- POPF – Flags regiszter kivétele a veremből
- IN – adat olvasása portról
- OUT – adat kiírása portra
- LEA – tényleges memóriacím betöltése
- LDS, LES – teljes pointer betöltése szegmensregiszter:általános regiszter regiszterpárba
- CBW – AL előjeles kiterjesztése AX-be
- CWD – AX előjeles kiterjesztése DX:AX-be
- XLAT, XLATB – AL lefordítása a DS:BX című fordító táblázattal
- LAHF – Flags alsó bájtjának betöltése AH-ba
- SAHF – AH betöltése Flags alsó bájtjába
- CLC – CF flag törlése
- CMC – CF flag komplementálása (invertálása)

- STC – CF flag beállítása
- CLD – DF flag törlése
- STD – DF flag beállítása
- CLI – IF flag törlése
- STI – IF flag beállítása

6.2.2. Egész számos aritmetika

- ADD – összeadás
- ADC – összeadás átvitellel (CF) együtt
- SUB – kivonás
- SBB – kivonás átvitellel (CF) együtt
- CMP – összehasonlítás (flag-ek beállítása a cél és a forrás különbségének megfelelően)
- INC – inkrementálás (növelés 1-gyel), CF nem változik
- DEC – dekrementálás (csökkentés 1-gyel), CF nem változik
- NEG – kettes komplemens képzés (szorzás −1-gyel)
- MUL – előjeltelen szorzás
- IMUL – előjeles szorzás
- DIV – előjeltelen maradékos osztás
- IDIV – előjeles maradékos osztás

6.2.3. Bitenkénti logikai utasítások (Boole-műveletek)

- AND – logikai AND
- OR – logikai OR
- XOR – logikai XOR
- NOT – logikai NOT (egyes komplemens képzés)
- TEST – logikai bit tesztelés (flag-ek beállítása a két op. logikai AND-jének megfelelően)

6.2.4. Bitléptető utasítások

- SHL – előjeltelen léptetés (shiftelés) balra
- SAL – előjeles léptetés balra (ugyanaz, mint SHL)
- SHR – előjeltelen léptetés jobbra
- SAR – előjeles (aritmetikai) léptetés jobbra
- ROL – balra forgatás (rotálás)
- RCL – balra forgatás CF-en át
- ROR – jobbra forgatás
- RCR – jobbra forgatás CF-en át

6.2.5. Sztringkezelő utasítások

- MOVS, MOVSB, MOVSW – sztring mozgatása
- CMPS, CMPSB, CMPSW – sztringek összehasonlítása
- SCAS, SCASB, SCASW – keresés sztringben
- LODS, LODSB, LODSW – betöltés sztringből
- STOS, STOSB, STOSW – tárolás sztringbe

6.2.6. Binárisan kódolt decimális (BCD) aritmetika

- AAA – ASCII igazítás összeadás után
- AAS – ASCII igazítás kivonás után
- AAM – ASCII igazítás szorzás után
- AAD – ASCII igazítás osztás előtt
- DAA – BCD rendezés összeadás után
- DAS – BCD rendezés kivonás után

6.2.7. Vezérlésátadó utasítások

- JMP – feltétel nélküli ugrás
- JCXZ – ugrás, ha CX = 0000h
- Jccc – feltételes ugrás („ccc” egy feltételt ír le)
- LOOP – CX dekrementálása és ugrás, ha CX ≠ 0000h
- LOOPE, LOOPZ – CX dekrementálása és ugrás, ha ZF = 1 és CX ≠ 0000h

- LOOPNE, LOOPNZ – CX dekrementálása és ugrás, ha ZF = 0 és CX ≠ 0000h
- CALL – eljárás (szubrutin) hívása
- RET, RETF – visszatérés szubrutinból
- INT – szoftver-megszakítás kérése
- INTO – INT 04h hívása, ha OF = 1, különben NOP-nak felel meg
- IRET – visszatérés megszakításból

6.2.8. Rendszervezérlő utasítások

- HLT – processzor leállítása, amíg megszakítás (vagy reset) nem érkezik

6.2.9. Koprocesszor-vezérlő utasítások

- WAIT – adatszinkronizálás a koprocesszorral
- ESC – utasítás küldése a koprocesszornak

6.2.10. Speciális utasítások

- NOP – üres utasítás (igazából XCHG AX,AX)

Az adatmozgató utasítások a leggyakrabban használt utasítások kategóriáját alkotják. Ez természetes, hiszen más (magasabb szintű) programozási nyelvekben is sokszor előfordul, hogy valamilyen adatot olvasunk ki vagy töltünk be egy változóba, avagy valamelyik portra. Kicsit kilógnak ebből a sorból a CBW, CWD, XLAT utasítások, de mondjuk a flag-eket állító CLC, CMC stb. utasítások is kerülhettek volna másik kategóriába.

Az egész aritmetikás utasításokat szintén nagyon sokszor használjuk programozáskor. A CMP utasításnak pl. kulcsszerepe lesz a különféle elágazások (feltételes vezérlési szerkezetek) lekódolásában.

A Boole-műveletekkel mindenféle bitműveletet (beállítás, maszkolás, tesztelés) el tudunk végezni, a munkák során éppen ezért nélkülözhetetlenek.

Shiftelő műveleteket főleg a már említett, 2 hatványával való szorzás gyors megvalósítására alkalmazunk. Ezenkívül nagy precizitású, saját aritmetikai műveletek fejlesztésekor is fontos szerepet kapnak a rotáló utasításokkal együtt.

Sztringen (string) itt bájtok vagy szavak véges hosszú folytonos sorát értjük. A sztringeken operáló utasítások pl. a nagy mennyiségű adatok másolásánál, vizsgálatánál segítenek.

BCD aritmetikát csak az emberi kényelem miatt támogat a processzor. Ezek az utasítások két csoportra oszthatók: pakolatlan (AAA, AAD, AAM, AAS) és pakolt (DAA,DAS) BCD aritmetikára. Kifejezetten ritkán használjuk őket, bár ez elég szubjektív kijelentés.

A vezérlésátadó utasítások a programok egyik alappillérét alkotják, helyes használatuk a jól működő algoritmus alapfeltétele.

Rendszer- és koprocesszor-vezérlő utasításokat csak speciális esetekben, külső hardverrel való kapcsolatfenntartásra és kommunikációra szokás használni.

Különleges „csemege" a NOP (No OPeration), ami eredete szerint adatmozgató utasítás lenne (mivel az XCHG AX,AX utasítás álneve), viszont működését tekintve gyakorlatilag nem

csinál semmit. Hasznos lehet, ha a kódba olyan üres bájtokat akarunk beszúrni, amit később esetleg valami egyéb célra használunk majd, de nem akarjuk, hogy a processzor azt valamilyen egyéb utasításnak értelmezve nemkívánatos mellékhatás lépjen fel. A NOP kódja egyetlen bájt (90h).

Szegmensregiszter-operandust (ami *nem* egyenlő a szegmensfelülbíráló prefixszel) a fenti utasítások közül csak a MOV, PUSH és POP kaphat, de ezzel egyelőre ne foglalkozzunk, később még visszatérünk rájuk.

7. fejezet

Assembly programok készítése

Az Assembly nyelvű programozás folyamata 3 jól elkülöníthető fázisra bontható: forrás elkészítése, fordítás, majd szerkesztés. Nézzük meg egyenként ezeket a lépéseket!

A programok forrásállományának a könnyebb azonosítás végett (és a hagyományok betartása érdekében is) célszerű egy .ASM kiterjesztésű nevet adni. A forrást bármilyen szövegszerkesztővel (pl. a DOS-os EDIT, valamilyen commander program szerkesztője, Windows NotePad-je stb.) elkészíthetjük, csak az a fontos, hogy a kimentett állomány ne tartalmazzon mindenféle formázási információt, képeket stb., pusztán a forrás szövegét.

Ha ez megvan, akkor jöhet a fordítás. A TASM program szintaxisa a következő:

TASM {opciók} forrás{,tárgykód}{,lista}{,xref}

Az opciók különféle kapcsolók és paraméterek, amik a fordítás menetét és a generált fájlok tartalmát befolyásolják. A kimeneti állomány tárgykód (object, .OBJ kiterjesztéssel) formátumú, és alapesetben a fájl neve megegyezik a forrás nevével. Ha akarjuk, ezt megváltoztathatjuk. Kérhetjük az assemblert, hogy a fordítás végeztével készítsen listát és/vagy keresztreferencia-táblázatot (cross reference table, xref). A listában a forrás minden egyes sorához generált gépi kód fel van tüntetve, ezenkívül tartalmazza a definiált szegmensek adatai, valamint a használt szimbólumokat összegyűjtő szimbólumtáblát. A keresztreferencia-táblázat azt tartalmazza, hogy a különböző szimbólumokat hol definiálták, ill. mely helyeken hivatkoztak rájuk. Ez a fájl bináris (tehát nem szöveges), értelmes információt belőle a TCREF programmal nyerhetünk.

Legtöbbször csak a lefordítandó forrás nevét adjuk meg a TASM-nak, hacsak nincs valamilyen különleges igényünk. Néhány fontos kapcsolót azért felsorolunk:

- /a, /s – A szegmensek szerkesztési sorrendjét befolyásolják. Az első ábécé rend szerint, míg a második a definiálás sorrendje szerinti szegmens-sorrendet ír elő. Alapértelmezés a /s.
- /l, /la – A listázás formátumát befolyásolják: normál (opcio/l) ill. kibővített (/la) lista fog készülni. Alapértelmezésben nem készül lista.
- /m# – A fordítás # menetes lesz (a „#" helyére egy számot kell írni). Alapérték a 2.
- /z – Hiba esetén a forrás megfelelő sorát is kiírja.
- /zi, /zd, /zn – A tárgykódba bekerülő nyomkövetési információt befolyásolják: teljes (/zi), csak a programsorok címei (/zd) vagy semmi (/zn). Alapértelmezés a /zn.

Miután mindegyik forrást lefordítottuk tárgykódra, jöhet a szerkesztés. A TLINK is számos argumentummal indítható:

TLINK {opciók} tárgykód fájlok{, futtatható fájl}{, térkép}{, könyvtárak}

Az opciók esetleges megadása után kell felsorolni azoknak a tárgykódú állományoknak a neveit, amiket egy futtatható fájlba akarunk összeszerkeszteni. A főprogram tárgykódját kell legelsőnek megadnunk. A futtatható fájl nevét kívánságunk szerint megadhatjuk, különben a legelső tárgykódú fájl (a főprogram) nevét fogja kapni. A térképfájl (map file) az egyes modulokban definiált szegmensek, szimbólumok adatait tartalmazza, valamint külön kérésre azoknak a programsoroknak a sorszámait, amikhez kód került lefordításra. A könyvtárakban (library) több tárgykódú állomány lehet összegyűjtve, a program írásakor tehát ezekből is használhatunk rutinokat, ha akarunk.

Most lássunk néhány megadható opciót:

- /x, /m, /l, /s – A térkép (map) tartalmát befolyásolják: nincs térkép (/x), publikus szimbólumok listája lesz (/m), kódhoz tartozó sorszámok lesznek (/l), szegmensinformáció lesz (/s). Alapértelmezés a /s.
- /c – Érzékeny lesz a kisbetűkre és nagybetűkre.
- /v – Engedélyezi a nyomkövetési információk beépítését a futtatható fájlba.
- /t – Az alapértelmezett .EXE helyett .COM típusú futtatható fájlt készít.

Most lássunk egy példát! Tegyük fel, hogy elkészítettünk egy Assembly forrásfájlt PELDA.ASM néven, és hogy a programunk csak ebből az állományból áll. A fordítást ekkor a

```
TASM PELDA
```

utasítás kiadásával intézhetjük el. A fordítás során valami ilyesmi jelenik meg a képernyőn:

```
Turbo Assembler  Version 4.0  Copyright (c) 1988, 1993 Borland International

Assembling file:   pelda.asm
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  355k
```

Az „`Error messages:`” címkéjű sor a hibák, a „`Warning messages:`” címkéjű a figyelmeztetések, a „`Passes:`” címkéjű pedig a menetek számát mutatja; a „`None`” jelenti azt, ha nincs hiba.

Sikeres fordítás esetén elkészül a PELDA.OBJ állomány (ez a tárgykód). Most következik a szerkesztés. Ehhez a következő utasítást kell kiadni:

```
TLINK PELDA
```

Ennek hatására (ha nem jelenik meg semmilyen figyelmeztető- vagy hibaüzenet) elkészül a futtatható állomány, ami a PELDA.EXE nevet kapja.

Ha a szerkesztő a „`Warning: No stack`” üzenetet jeleníti meg, az azt jelenti, hogy nem hoztunk létre a programban vermet. Ez elég gyakori hiba a tapasztalatlan Assembly-programozóknál. Mivel veremre mindenképpen szüksége van a programnak, sürgősen javítsuk ki mulasztásunkat!

8. fejezet

Vezérlési szerkezetek megvalósítása

Ennyi elmélet után ideje, hogy belefogjunk az Assembly programok írásába! Először néhány példán keresztül megnézzük, hogy kódolható le néhány gyakran használt vezérlési szerkezet Assemblyben.

8.1. Szekvenciális vezérlési szerkezet

A processzor mindig szekvenciális utasítás-végrehajtást alkalmaz, hacsak nem használunk valamilyen vezérlésátadó utasítást. Lássuk hát első Assembly programunkat, ami nem sok hasznos dolgot csinál, de kezdésnek megteszi.

Pelda1.ASM:

```
MODEL SMALL

.STACK

ADAT        SEGMENT
Kar         DB        'a'
Szo         DW        "Ha"
Szam        DB        12h
Ered        DW        ?
ADAT        ENDS

KOD         SEGMENT
            ASSUME    CS:KOD,DS:ADAT
@Start:
            MOV       CX,ADAT
            MOV       DS,CX
            MOV       BL,[Kar]
            MOV       AX,[Szo]
            XOR       BH,BH
            ADD       AX,BX
            SUB       AL,[Szam]
            MOV       [Ered],AX
```

```
                MOV     AX,4C00h
                INT     21h
KOD             ENDS
                END     @Start
```

A programban három szegmenst hozunk létre: egy ADAT nevűt az adatoknak, egy KOD nevűt az utasításoknak, illetve egy vermet. Utóbbit a .STACK egyszerűsített szegmensdefiníciós direktíva készíti el, a szegmens neve STACK, mérete 0400h (1 Kbájt) lesz. Ezt a direktívát egészíti ki a MODEL, itt a memória-modellnek SMALL-t írtunk elő, ami annyit tesz, hogy a kód- és az adatterület mérete külön-külön max. 64 Kbájt lehet.

Szegmensdefiníciót a szegmens nevéből és a SEGMENT direktívából álló sorral kezdünk, ezt követi a szegmens tartalmának leírása. Lezárásként a szegmens nevét és az azt követő ENDS foglalt szót tartalmazó sort írjuk. Azaz:

```
Név             SEGMENT   {attribútumok}
definíciók
Név             ENDS
```

A SEGMENT kulcsszó után a szegmens jellemzőit meghatározó különféle dolgokat írhatunk még, ezek a szegmens illeszkedését (alignment), kombinálását (combine type), osztályát (class), operandusméretét (operand size) és hozzáférési módját (access mode) határozzák meg. Ezekkel egyelőre még ne törődjünk.

Az adatszegmensben négy változót hozunk létre. Ebből háromnak (Kar, Szo és Szam) kezdeti értéket is adunk, ezek tehát inkább hasonlítanak a Pascal típusos konstansaihoz. Változónak így foglalhatunk helyet:

```
Név             Dx              KezdetiÉrték<,KezdetiÉrték>
```

ahol Dx helyén DB, DW, DD, DF, DQ és DT állhat (jelentésük Define Byte, Word, Doubleword, Farword, Quadword és Ten byte unit). Ezek sorban 1, 2, 4, 6, 8 ill. 10 bájtnyi területet foglalnak le a Név nevű változónak. A kezdeti érték megadásánál írhatunk közönséges számokat (12h), karakter-konstansokat ('a'), sztring-konstansokat ("Ha"), ezek valamilyen kifejezését, illetve ha nem akarunk értéket adni neki, akkor egy „?"-et is. A kérdőjellel definiált változókat a legtöbb assembler 00h-val tölti fel, de az is megoldható, hogy ezeket a területeket ne tárolja el a futtatható állományban, s indításkor ezek helyén valami véletlenszerű érték legyen. Vesszővel elválasztva több értéket is felsorolhatunk, ekkor mindegyik értékhez egy újabb terület lesz lefoglalva. Így pl. a következő példa 10 bájtot foglal le, a terület elejére pedig a Valami címke fog mutatni:

```
Valami          DB              1,2,'#',"Szöveg",6+4
```

A kódszegmens megnyitása után jeleznünk kell az assemblernek, hogy ez valóban „kódszegmens", illetve meg kell adnunk, hogy melyik szegmens szerint számolja a memóriahivatkozások offszetcímét. Mindezeket az ASSUME direktívával intézzük el. Az ASSUME direktíva szintaxisa:

```
ASSUME    SzegReg:SzegNév<,SzegReg:SzegNév>
ASSUME    NOTHING
```

A SzegReg valamilyen szegmensregisztert jelent, a SzegNev helyére pedig egy definiált szegmens vagy szegmenscsoport nevét, vagy a NOTHING kulcsszót írhatjuk. Utóbbi esetben az adott szegmensregisztert egyetlen szegmenshez sem köti hozzá, és nem tételez fel semmit sem az adott regiszterrel kapcsolatban. Az ASSUME NOTHING minden előző hozzárendelést megszüntet.

Rögtön ezután egy címke következik, aminek a @Start nevet adtuk. (A „kukac" most csak arra utal, hogy ez nem változó-, hanem címkeazonosító. Ennek használata csak megszokás kérdése, de ebben a jegyzetben az összes címke nevét a „@" karakterrel fogjuk kezdeni.) Rendeltetése az, hogy a program indulásának helyét jelezze, erre az információra ugyanis az assemblernek, linkernek és az operációs rendszernek is szüksége van. A címke önmagában nem lesz elég, az indulási cím kijelölését a később leírt END direktíva fogja elvégezni.

Az első utasítást, ami végrehajtható gépi kódot generál, a MOV (MOVe data) képviseli. Ez két operandussal rendelkezik: az első a cél, a második a forrás (ez persze az összes kétoperandusú utasítást jellemzi). Működése: egyszerűen veszi a forrás tartalmát, és azt bemásolja (eltárolja) a cél op. területére. Eközben a forrás tartalmát, ill. a flag-eket békén hagyja, csak a cél tartalma módosul(hat). Operandusként általános regiszteren és memóriahivatkozáson kívül konstans értéket (kifejezést) és szegmensregisztert is megadhatunk.

Az első két sor az adatszegmens-regiszter (DS) tartalmát állítja be, hogy az általunk létrehozott ADAT-ra mutasson. Ezt nem teszi meg helyettünk az assembler annak ellenére sem, hogy az ASSUME direktívában már szerepel a DS:ADAT előírás. Látható, hogy az ADAT szegmens címét először betöltjük egy általános regiszterbe (CX), majd ezt bemozgatjuk DS-be. Viszont a MOV DS,ADAT utasítás érvénytelen lenne, mert az Intel 8086-os processzor csak memória vagy általános regiszter tartalmát tudja szegmensregiszterbe átrakni, és ez a fordított irányra is vonatkozik (mellesleg pl. a MOV 1234h,DS utasításnak nem lenne semmi értelme).

Most jegyezzünk meg egy fontos dolgot: CS tartalmát ilyen módon nem tölthetjük fel (azaz a MOV CS, ... forma érvénytelen), csak a vezérlésátadó utasítások változtathatják meg CS-t! Olvasni viszont lehet belőle, így pl. a MOV SI,CS helyes utasítás lesz.

A következő sor a Kar változó tartalmát BL-be rakja. Látható, hogy a memóriahivatkozást a szögletes zárójelpár jelzi. Ez egyébként jó példa a közvetlen címzésre, a sort ugyanis az assembler a MOV BL,[0000h] gépi utasításra fordítja le. Ha jobban megnézzük, 0000h pont a Kar változó ADAT szegmensbeli offszetje. Ezért van tehát szükség az ASSUME direktíva megadására, mert különben az assembler nem tudta volna, hogy a hivatkozás mire is vonatkozik.

AX-be hasonló módon a Szo változó értéke kerül.

A XOR BH,BH kicsit trükkös dolgot művel. Ha visszaemlékszünk a KIZÁRÓ VAGY értéktáblázatára, akkor könnyen rájöhetünk, hogy ha egy számot önmagával hozunk KIZÁRÓ VAGY kapcsolatba, akkor minden esetben nullát kapunk. Így tehát ez a sor megfelel a MOV BH,00h utasításnak, de ezt írhattuk volna még SUB BH,BH-nak is. Tehát már háromféleképpen tudunk kinullázni egy regisztert!

Hogy végre számoljunk is valamit, az ADD AX,BX összeadja a BX tartalmát az AX tartalmával, majd az eredményt az AX-be teszi. Röviden: hozzáadja BX-et az AX regiszterhez. A művelet elvégzése során beállítja az OF, SF, ZF, AF, PF és CF flag-eket.

Kitalálható, hogy a következő sorban a Szam változó tartalmát vonjuk le az AL regiszterből, a SUB (SUBtract) ugyanis a forrás tartalmát vonja ki a célból.

Mind az ADD, mind a SUB utasításra igaz, hogy operandusként általános regisztert, memóriahivatkozást és közvetlen (konstans) értéket kaphatnak.

Számolásunk eredményét el is kell tenni, ezt végzi el a MOV [Ered],AX sor.

Ha befejeztük dolgunkat, és „ki akarunk lépni a programból", akkor erről az aktuális operációs rendszert (ez most a DOS) kell értesíteni, s az majd cselekedni fog. Ezt végzi el programunk

utolsó két sora. Magyarázatképpen csak annyit, hogy a 21h szoftver-megszakítás alatt érhetjük el a DOS funkcióit, ennek 4Ch számú szolgáltatása jelenti a programból való kilépést (vagy másképpen a program befejezését, terminálását). A szolgáltatás számát AH-ban kell megadnunk, AL-be pedig egy visszatérési értéket, egyfajta hibakódot kell írni (ez lesz az ERRORLEVEL nevű változó értéke a DOS-ban). A megszakítást az egyoperandusú INT (INTerrupt request) utasítás váltja ki, s ezzel a program futása befejeződik. Az INT operandusa csak egy bájt méretű közvetlen érték lehet. A megszakítások használatát később tárgyaljuk, most egyelőre fogadjuk el, hogy így kell befejezni a programot. (Persze másképpen is lehetne, de most így csináljuk.)

Az aktuális forrásfájlt az END direktívával kell lezárni, különben az assembler morcos lesz. Ha ez a fő forrásfájl, azaz ez tartalmazza azt a programrészt, aminek el kell indulnia a program betöltése után, akkor az END után egy címke nevének kell szerepelnie. Az itt megadott címre fog adódni a vezérlés a futtatható programfájl betöltése után.

Vajon mi lett a számolás eredménye? BL-be az 'a' karaktert raktuk, ennek ASCII kódja 61h. A Szo értékének megadásához tudni kell, hogy a sztring-konstansokat is helyiértékes rendszerben értelmezi az assembler, 256-os alapú számként tekintve őket. Ezért a Szo tartalma 4861h. Az összeadás után így 4861h + 61h = 48C2h lesz AX-ben. Ebből még levonjuk a Szam tartalmát (12h), s ezzel kialakul a végeredmény: 48B0h.

8.2. Számlálásos ismétléses vezérlés

Magas szintű nyelvekben igen gyakran alkalmaznak olyan ciklusokat, ahol a ciklusváltozó egy bizonyos tartományt fut be, és a ciklus magja a tartomány minden értékére lefut egyszer. Ezt valósítják meg a különféle FOR utasítások.

Assemblyben is mód nyílik számlálásos ismétléses vezérlésre, viszont van néhány megkötés a többi nyelvvel szemben: ciklusváltozónak csak CX használható, és nem adható meg a ciklusváltozó ismétlési tartománya, csak a lefutások számát írhatjuk elő.

A következő program két vektor (egydimenziós tömb) tartalmát összegzi egy harmadik vektorba. Az egyik vektor előjeles bájtokból, míg a másik előjeles szavakból fog állni.

Pelda2.ASM:

```
MODEL SMALL

.STACK

ADAT        SEGMENT
ElemSzam    EQU       5
Vekt1       DB        6,-1,17,100,-8
Vekt2       DW        1000,1999,-32768,4,32767
Eredm       DW        ElemSzam DUP (?)
ADAT        ENDS

KOD         SEGMENT
            ASSUME    CS:KOD,DS:ADAT

@Start:
            MOV       AX,ADAT
            MOV       DS,AX
```

```
                MOV     CX,ElemSzam
                XOR     BX,BX
                MOV     SI,BX
@Ciklus:
                MOV     AL,[Vekt1+BX]
                CBW
                ADD     AX,[Vekt2+SI]
                MOV     [Eredm+SI],AX
                INC     BX
                LEA     SI,[SI+2]
                LOOP    @Ciklus
                MOV     AX,4C00h
                INT     21h
KOD             ENDS
                END     @Start
```

Az első újdonságot az adatszegmensben szereplő EQU (define numeric EQUate) kulcsszó képviseli. Ez a fordítási direktíva numerikus konstanst (pontosabban helyettesítő makrót) tud létrehozni, ezért leginkább a C nyelv #define utasításához hasonlítható. Hatására az ElemSzam szimbólum minden előfordulását az EQU jobb oldalán álló kifejezés értékével (5) fogja helyettesíteni az assembler.

Az eredmény vektor nyilván ugyanolyan hosszú lesz, mint a két kiinduló vektor, típusa szerint szavas lesz. Számára a helyet a következőképpen is lefoglalhattuk volna:

```
Eredm           DW      0,0,0,0,0
```

Ez a megoldás azonban nem túl rugalmas. Ha ugyanis megváltoztatjuk az ElemSzam értékét, akkor az Eredm definíciójához is hozzá kell nyúlni, illetve ha nagy az elemek száma, akkor sok felesleges 0-t vagy kérdőjelet kell kiírnunk. Ezeket a kényelmetlenségeket szünteti meg a duplikáló operátor. A DUP (DUPlicate) hatására az operátor bal oldalán álló számú (ElemSzam), adott típusú (DW) elemekből álló terület lesz lefoglalva, amit az assembler a DUP jobb oldalán zárójelben szereplő kifejezéssel (most „?”) inicializál (azaz ez lesz a kezdőérték). Ez a módszer tehát jóval leegyszerűsíti a nagy változóterületek létrehozását is.

A ciklusváltozó szerepét a CX regiszter tölti be, ami nem véletlen, hiszen a regiszter neve is innen származik (Counter).

A vektorelemek elérésére most nem használhatunk közvetlen címzést, hiszen azzal csak a vektorok legelső elemét tudnánk kiválasztani. Ehelyett két megoldás kínálkozik: minden adatterülethez hozzárendelünk egy-egy pointert (mutatót), vagy pedig valamilyen relatív címzést alkalmazunk.

A pointeres megoldásnak akkor van értelme, ha mondjuk egy olyan eljárást írunk, ami a célterületnek csak a címét és elemszámának méretét kapja meg. Pointer alatt itt most rövid, szegmensen belüli mutatót értünk, ami egy offszet érték lesz. Pointereket csak bázis- vagy indexregiszterekben tárolhatunk, hiszen csak ezekkel végezhető memóriahivatkozás.

A másik módszer lehetővé teszi, hogy a változóinkat ténylegesen tömbnek tekinthessük, és ennek megfelelően az egyes elemeket indexeléssel (tömbelem-hivatkozással) érhessük el. Ehhez a megoldáshoz szükség van a tömbök bázisára (kezdőcímére), illetve az elemkiválasztást megvalósító tömbindexre. A tömbök kezdőcímét most konkrétan tudjuk, így ez konstans numerikus érték lesz. A tömbindexet tetszésünk szerint tárolhatjuk bármely bázis- vagy indexregiszterben. Mivel minden vektorban elemről-elemre sorban haladunk, valamint a Vekt2

és az Eredm vektorok is szavasak, ezért most csak kettő tömbindexre van szükségünk. BX a Vekt1-et indexeli, és egyesével kell növelni. SI-t ezzel szemben Vekt2 és Eredm indexelésére is használjuk, növekménye 2 bájt lesz.

Vigyázzunk, az itt leírt tömbelem-hivatkozás csak alakilag hasonlít a magas szintű nyelvek tömbindexelésére! Míg az utóbbiaknál a hivatkozott tömbelem sorszámát kell megadni indexként, addig Assemblyben a tömbelemnek a tömb (vagy általában a terület) elejéhez képesti relatív címét használjuk.

Assemblyben a ciklus magja a ciklus kezdetét jelző címkétől (most @Cimke) a ciklusképző utasításig (ez lesz a LOOP) tart.

A Vekt1 következő elemét olvassuk ki először AL-be, bázisrelatív címzést használva. Azután ehhez kéne hozzáadnunk a Vekt2 megfelelő elemét. A két vektor elemmérete azonban eltérő, egy bájtot pedig nem adhatunk hozzá egy szóhoz. Ez megoldható, ha a bájtot átkonvertáljuk szóvá. Az operandus nélküli CBW (Convert Byte to Word) utasítás pont ezt teszi, az AL-ben levő értéket előjelesen kiterjeszti AX-be. Így már elvégezhető az összeadás, ami után az eredményt is a helyére tesszük.

Az INC (INCrement) utasítás az operandusát növeli meg 1-gyel, de természetesen nem lehet közvetlen érték ez az operandus. Működése az ADD ...,1 utasítástól csak annyiban tér el, hogy nem változtatja meg a CF értékét, és ez néha fontos tud lenni. Az INC párja a DEC (DECrement), ami eggyel csökkenti operandusát.

SI-t többféleképpen is megnövelhetnénk 2-vel:

1.

```
INC     SI
INC     SI
```

2.

```
ADD     SI,2
```

Mindkét megoldásnak van egy pici szépséghibája: az első feleslegesen ismétel meg egy utasítást, a második viszont túl nagy kódot generál (4 bájtot), és a flag-eket is elállítja. Ha kicsi a hozzáadandó érték (mondjuk előjelesen 1 bájtos), akkor érdemesebb a LEA (Load Effective Address) utasítást használni. Ez a forrás memóriaoperandus tényleges címét (effective address) tölti be a céloperandusba, ami csak egy 16 bites általános regiszter lehet. Az összes flag-et békén hagyja. Ismétlésképpen: tényleges cím alatt a használt címzési mód által jelölt, meghatározott memóriacímet (offszetet) értjük. Így a LEA SI,[SI+2] utasítás a DS:(SI+2) című memóriarekesz offszetcímét tölti be SI-be, azaz 2-vel megnöveli SI-t. Ez így csak 3 bájtos utasítást eredményez. Az utasítás jelentősége jobban látszik, ha bonyolultabb címzésmódot használunk:

```
LEA     AX,[BX+DI-100]
```

Itt AX-be egy háromtagú összeg értéke kerül, s mindezt úgy hajthatjuk végre, hogy más regiszterre nem volt szükségünk, továbbá ez az utasítás is csak három bájtot foglal el.

A tényleges ciklusképzést a LOOP utasítás végzi. Működése: csökkenti CX-et, majd ha az zérussá válik, akkor a LOOP utáni utasításra kerül a vezérlés, különben elugrik az operandus által mutatott memóriacímre. A csökkentés során egyetlen flag értéke sem változik meg. A

LOOP ú.n. relatív ugrást használ, ami azt jelenti, hogy a cél címet a LOOP után következő utasítás elejéhez képest relatívan kell megadni (ez csak a gépi kód szinten számít, Assemblyben nyugodtan használhatunk címkéket). Ezt a relatív címet 1 bájton tárolja el, így a $-128 - +127$ bájtnyi távolságra levő címekre tudunk csak ciklust szervezni. Ha az operandus által mutatott cím túl messze lenne, akkor az assembler hibát fog jelezni. Ilyenkor csak azt tehetjük, hogy más módon (pl. feltételes és feltétel nélküli ugrások kombinációjával) oldjuk meg a problémát. A szabályos ciklusokban egyébként csak visszafelé mutató címet szoktunk megadni a LOOP után (tehát a relatív cím negatív lesz). Fontos még tudni, hogy a LOOP először csökkenti CX-et, s csak ezután ellenőrzi, hogy az nullává vált-e. Így ha a ciklusba belépéskor CX zéró volt, akkor nem egyszer (és nem is nullaszor), hanem 65536-szor fog lefutni ciklusunk! A LOOP utasítással megírt ciklus tehát legalább 1-szer mindenképpen lefut.

A LOOP utasításnak még két változata létezik. A LOOPE, LOOPZ (LOOP while Equal/Zero/ZF = 1) csökkenti CX-et, majd akkor ugrik a megadott címre, ha CX $\neq$ 0000h és ZF = 1. Ha valamelyik vagy mindkét feltétel nem teljesül, a LOOP utáni utasításra kerül a vezérlés. A két mnemonik ugyanazt a műveleti kódot generálja. Hasonlóan a LOOPNE, LOOPNZ (LOOP while Not Equal/Not Zero/ZF = 0) utasítás CX csökkentése után akkor ugrik, ha CX $\neq$ 0000h és ZF = 0 is teljesül.

8.3. Egyszerű és többszörös szelekciós vezérlés

Ez a vezérlési szerkezet egy vagy több feltétel teljesülése vagy nem teljesülése szerinti elágazást tesz lehetővé a program menetében. Megfelel az IF...THEN...ELSE utasítások láncolatának.

A megoldandó probléma: konvertáljunk át egy karaktersorozatot csupa nagybetűsre. Az ékezetes betűkkel most nem törődünk, a szöveget pedig az ASCII 00h kódú karakter fogja lezárni.

Pelda3.ASM:

```
MODEL SMALL

.STACK

ADAT      SEGMENT
Szoveg    DB        "Ez egy PELDA szoveg."
          DB        "Jo proci a 8086-os!",00h
ADAT      ENDS

KOD       SEGMENT
          ASSUME    CS:KOD,DS:ADAT

@Start:
          MOV       AX,ADAT
          MOV       DS,AX
          PUSH      DS
          POP       ES
          LEA       SI,[Szoveg]
          MOV       DI,SI
          CLD
```

```
@Ciklus:
                LODSB
                OR      AL,AL
                JZ      @Vege
                CMP     AL,'a'
                JB      @Tarol
                CMP     AL,'z'
                JA      @Tarol
                SUB     AL,'a'-'A'
@Tarol:
                STOSB
                JMP     @Ciklus
@Vege:
                MOV     AX,4C00h
                INT     21h
KOD             ENDS
                END     @Start
```

A program elején két ismerős utasítással is találkozhatunk, ezek a verem tárgyalásánál már említett PUSH és POP. Mindkettő egyoperandusú, s ez az operandus nemcsak általános regiszter vagy memóriahivatkozás, de szegmensregiszter is lehet. 8 bites regisztert viszont nem fogadnak el, mivel a verem szavas szervezésű. A programban szereplő PUSH-POP pár tulajdonképpen DS tartalmát tölti ES-be, amit rövidebben írhattunk volna MOV ES,AX-nek is, mindössze szemléltetni akarjuk, hogy így is lehet értéket adni egy szegmensregiszternek.

Bár nem tartozik szorosan a kitűzött feladathoz, két probléma is felmerül ezzel a két utasítással kapcsolatban. Az egyik, hogy vajon mit csinál a PUSH SP utasítás? Nos, a 8086-os processzor esetén ez SP 2-vel csökkentése után SP új (csökkentett) értékét teszi le a verem SS:SP címére, míg a későbbi processzorok esetén a régi érték kerül eltárolásra. Furcsa egy megoldás, az biztos... Mindenesetre ezzel nem kell foglalkoznunk, csak egy érdekesség.

A másik dolog, amit viszont fontos megjegyezni, az az, hogy a POP CS utasítást a MOV CS, ... -hez hasonlóan nem szereti sem az assembler, sem a processzor. Ha belegondolunk hogy ez mit jelentene, láthatjuk, hogy feltétel nélküli vezérlésátadást valósítana meg, de úgy, hogy közben csak CS értéke változna, IP-é nem! Ez pedig veszélyes és kiszámíthatatlan eredményekkel járna. Így a CS nem lehet a POP operandusa. (Igazság szerint a 8086-os processzor értelmezni tudja ezt az utasítást, aminek gépi kódja 0Fh, viszont a későbbi processzorokon ennek a kódnak más a szerepe. Ezért és az előbb említettek miatt *ne* használjuk!)

A következő három utasítás a sztringkezelő utasításokat (LODSB és STOSB) készíti elő.

A forrássztring címét DS:SI fogja tartalmazni, így a Szoveg offszetjét betöltjük SI-be. Megjegyezzük, hogy a TASM ezt az utasítást MOV SI,OFFSET [Szoveg]-ként fogja lefordítani, talán azért, mert ez kicsit gyorsabb lesz, mint az eredeti változat.

A célsztring az ES:DI címen fog elhelyezkedni. Most helyben fogunk konvertálni, így DI-t egyenlővé tesszük SI-vel.

Az operandus nélküli CLD (CLear Direction flag) a DF flag-et törli. Ennek szükségessége rögvest kiderül.

A LODSB (LOaD String Byte) a DS:SI címen levő bájtot betölti AL-be, majd SI-t eggyel növeli, ha DF = 0, ill. csökkenti, ha DF = 1. Másik alakja, a LODSW (LOaD String Word) AX-be tölti be a DS:SI-n levő szót, és SI-t 2-vel növeli vagy csökkenti DF-től függően. Egyetlen flag-et sem változtat meg.

Az OR AL,AL utasítás ismét egy trükköt mutat be. Ha egy számot önmagával hozunk lo-

gikai VAGY kapcsolatba, akkor maga a szám nem változik meg. Cserébe viszont CF, AF és OF törlődik, SF, ZF és PF pedig az operandusnak megfelelően módosulnak. Itt most annak tesztelésére használtuk, hogy AL zérus-e, azaz elértük-e a sztringünk végét. Ezt megtehettük volna a később szereplő CMP AL,00h utasítással is, csak így elegánsabb.

Elérkeztünk a megoldás kulcsához, a **feltételes ugrásokhoz** (conditional jumps). Ezek olyan vezérlésátadó utasítások, amik valamilyen flag (vagy flag-ek) állása alapján vagy elugranak az operandus szerinti címre, vagy a következő utasításra térnek. Összesen 17 db. van belőlük, de mnemonikból ennél több van, mivel némelyik egynél több elnevezés alatt is elérhető. A táblázatban egy sorban levő mnemonikok ugyanazt az utasítást képviselik. Három csoportba oszthatók: előjeles aritmetikai, előjeltelen aritmetikai és egyéb feltételes ugrások.

Először jöjjenek az előjeles változatok (8.1. táblázat)!

8.1. táblázat. Előjeles aritmetikai feltételes ugrások

Mnemo.	*Hatás*
JL, JNGE	Ugrás, ha kisebb/nem nagyobb vagy egyenlő
JNL, JGE	Ugrás, ha nem kisebb/nagyobb vagy egyenlő
JLE, JNG	Ugrás, ha kisebb vagy egyenlő/nem nagyobb
JNLE, JG	Ugrás, ha nem kisebb vagy egyenlő/nagyobb

A mnemonikokat tehát úgy képezzük, hogy a „J” betűt (ami a Jump if... kifejezést rövidíti) egy, kettő vagy három betűs rövidítés követi. Némi angol tudással könnyebben megjegyezhetők az egyes változatok, ha tudjuk, mit jelölnek az egyes rövidítések:

- L – Less
- G – Greater
- N – Not
- LE – Less or Equal
- GE – Greater or Equal

Most nézzük az előjel nélküli ugrásokat (8.2. táblázat)!

8.2. táblázat. Előjeltelen aritmetikai feltételes ugrások

Mnemonik	*Hatás*
JB, JNAE, JC	Ugrás, ha kisebb/nem nagyobb vagy egyenlő/CF = 1
JNB, JAE, JNC	Ugrás, ha nem kisebb/nagyobb vagy egyenlő/CF = 0
JBE, JNA	Ugrás, ha kisebb vagy egyenlő/nem nagyobb
JNBE, JA	Ugrás, ha nem kisebb vagy egyenlő/nagyobb

Az alkalmazott rövidítések:

- B – Below
- C – Carry
- A – Above

- BE – Below or Equal
- AE – Above or Equal

Mint várható volt, a JB és JC ugyanazt jelentik, hiszen mindkettő azt fejezi ki, hogy a legutolsó művelet előjel nélküli aritmetikai túlcsordulást okozott.

Lássuk a megmaradt további ugrásokat (8.3. táblázat)!

8.3. táblázat. Speciális feltételes ugrások

Mnemo.	*Hatás*
JE, JZ	Ugrás, ha egyenlő/ZF = 1
JNE, JNZ	Ugrás, ha nem egyenlő/ZF = 0
JO	Ugrás, ha OF = 1
JNO	Ugrás, ha OF = 0
JS	Ugrás, ha SF = 1
JNS	Ugrás, ha SF = 0
JP, JPE	Ugrás, ha PF = 1/páros paritás
JNP, JPO	Ugrás, ha PF = 0/páratlan paritás
JCXZ	Ugrás, ha CX = 0000h

A rövidítések magyarázata pedig:

- E – Equal
- Z – Zero
- O – Overflow
- S – Sign
- P – Parity
- PE – Parity Even
- PO – Parity Odd
- CXZ – CX is Zero (vagy CX equals Zero)

Mint látható, a JCXZ kakukktojás a többi ugrás között, mivel nem egy flag, hanem a CX regiszter állása szerint cselekszik. Mindegyik feltételes ugró utasítás egyetlen operandusa a cél memóriacím, de ez is előjeles relatív címként 1 bájtban tárolódik el, a LOOP utasításhoz hasonlóan. Pontosan ezen megkötés miatt található meg mindegyik feltételnek (a JCXZ-t kivéve) a negált párja is. A következő példát ugyanis nem fogja lefordítani az assembler (a TASM-ra ez nem teljesen igaz, a JUMPS és NOJUMPS direktívákkal megadható, hogy magától kijavítsa-e az ilyen helyzeteket, avagy utasítsa vissza):

```
            XOR     AX,AX
            JZ      @Cimke
            ...
KamuAdat    DB      200 DUP (?)
@Cimke:
            ...
```

A 200 bájtos területfoglalás biztosítja, hogy a JZ utasítást követő bájt messzebb legyen a @Cimke címkétől, mint a megengedett +127 bájt. Az ugrást ráadásul végre kell hajtani, mivel a XOR AX,AX hatására ZF = 1 lesz. Ezt a hibás szituációt feloldhatjuk, ha a JZ... helyett JNZ-JMP párost használunk (a JMP leírása kicsit lentebb lesz):

```
              XOR     AX,AX
              JNZ     @KamuCimke
              JMP     @Cimke
@KamuCimke:
              ...
KamuAdat      DB      200 DUP (?)
@Cimke:
              ...
```

A példaprogramhoz visszatérve, az OR AL,AL és a JZ... hatására megoldható, hogy az algoritmus leálljon, ha elértük a sztring végét.

Most jön maga a szelekció: el kell dönteni, hogy az aktuális karaktert (ami AL-ben van) kell-e konvertálni. Konverziót kell végezni, ha 61h ≤ AL ≤ 7Ah ('a' kódja 61h, 'z' kódja 7Ah), különben változatlanul kell hagyni a karaktert. Először azt nézzük meg, hogy AL < 61h teljesül-e. Ha igen, akkor nem kell konverzió. Különben ha AL > 7Ah, akkor ugyancsak nincs konverzió, különben pedig elvégezzük az átalakítást. A leírtakat CMP-Jccc utasításpárosokkal oldjuk meg. (A „ccc" egy érvényes feltételt jelöl. A „Jccc" jelölés ezért valamilyen feltételes ugrás mnemonikja helyett áll.) A CMP (CoMPare) utasítás kiszámítja a céloperandus és a forrásoperandus különbségét, beállítja a flag-eket, az operandusokat viszont békén hagyja (és eldobja az eredményt is). Operandusként általános regisztert, memóriahivatkozást és konstans értéket adhatunk meg.

A konverziót az egyetlen SUB AL,'a'-'A' utasítás végzi el.

Akár volt konverzió, akár nem, az algoritmus végrehajtása a @Tarol címkétől folytatódik. Megfigyelhetjük, hogy ez a rész közös rész a szelekció mindkét ágában (t.i. volt-e konverzió avagy nem), így felesleges lenne mindkét esetben külön leírni. Ehelyett azt tesszük, hogy ha nem kell konvertálni, akkor elugrunk a @Tarol címkére, különben elvégezzük a szükséges átalakítást, és „rácsorgunk" erre a címkére. Az ilyen megoldások igen gyakoriak az Assembly programokban, használatuk rövidebbé, gyorsabbá teszi azokat, az algoritmusok pedig áttekinthetőbbek lesznek általuk.

Most jön a kérdéses karakter eltárolása. Ezt egy másik sztringkezelő utasítás, a STOSB (STOre String Byte) intézi el. Működése: AL-t eltárolja az ES:DI címre, majd DI-t megnöveli eggyel, ha DF = 0, illetve csökkenti, ha DF = 1. A STOSW (STOre String Word) utasítás, hasonlóan a LODSW-hez, AX-szel dolgozik, és DI-t 2-vel növeli vagy csökkenti.

Ezután vissza kell ugranunk a @Ciklus címkére, mert a többi karaktert is fel kell dolgozni a sztringünkben. Erre szolgál a JMP (JuMP) feltétel nélküli ugrás (unconditional jump). Működése: az operandusként megadott címre állítja be IP-t (van egy másik, távoli formája is, ekkor CS-t is átállítja), és onnan folytatja a végrehajtást. Érdekes, hogy a JMP is relatív címként tárolja el a cél memóriacímét, viszont a feltételes ugrásokkal és a LOOP-pal ellentétben a JMP 16 bites relatív címet használ, így a −32768 – +32767 bájt távolságban levő címekre tudunk előre-hátra ugrálni. (Ez csak a közeli, ugyanazon kódszegmensen belüli ugrásra vonatkozik, de erről majd később.)

Ha jobban megnézzük, észrevehetjük, hogy az algoritmus elején szereplő OR AL,AL // JZ @Vege és az utóbbi JMP @Ciklus utasítások egy hurok vezérlési szerkezetet írnak elő, amiben

az OR-JZ páros alkotja a kilépési feltételt, a JMP pedig maga a ciklusszervező utasítás (t.i. végtelen ciklust valósít meg). De ha jobban tetszik, ezt tekinthetjük akár előfeltételes vezérlési szerkezetnek is (WHILE... DO ciklus).

A @Vege címke elérésekor a kiinduló sztringünk már nagybetűsen virít a helyén.

8.4. Eljárásvezérlés

A program eljárásokra és függvényekre (egyszóval szubrutinokra) bontása a strukturált programozás alapját képezi. Nézzük meg, hogyan használhatunk eljárásokat Assemblyben.

A feladat: írjunk olyan eljárást, ami az AX-ben található előjel nélküli számot kiírja a képernyőre decimális alakban!

Pelda4.ASM:

```
MODEL SMALL

.STACK

KOD         SEGMENT
            ASSUME    CS:KOD,DS:NOTHING

DecKiir     PROC
            PUSH      AX
            PUSH      BX
            PUSH      CX
            PUSH      DX
            MOV       BX,10
            XOR       CX,CX
@OsztCikl:
            OR        AX,AX
            JZ        @CiklVege
            XOR       DX,DX
            DIV       BX
            PUSH      DX
            INC       CX
            JMP       @OsztCikl
@CiklVege:
            MOV       AH,02h
            JCXZ      @NullaVolt
@KiirCikl:
            POP       DX
            ADD       DL,'0'
            INT       21h
            LOOP      @KiirCikl
            JMP       @KiirVege
@NullaVolt:
            MOV       DL,'0'
            INT       21h
@KiirVege:
            POP       DX
```

```
                POP     CX
                POP     BX
                POP     AX
                RET
DecKiir         ENDP

UjSor           PROC
                PUSH    AX
                PUSH    DX
                MOV     AH,02h
                MOV     DL,0Dh
                INT     21h
                MOV     DL,0Ah
                INT     21h
                POP     DX
                POP     AX
                RET
UjSor           ENDP

@Start:
                XOR     AX,AX
                CALL    DecKiir
                CALL    UjSor
                MOV     AX,8086
                CALL    DecKiir
                CALL    UjSor
                MOV     AX,8000h
                CALL    DecKiir
                CALL    UjSor
                MOV     AX,0FFFFh
                CALL    DecKiir
                CALL    UjSor
                MOV     AX,4C00h
                INT     21h
KOD             ENDS
                END     @Start
```

Ez egy kicsit hosszúra sikeredett, de hát a probléma sem annyira egyszerű. Nézzük először a főprogramot (azaz a @Start címke utáni részt)! Itt minden ismerősnek tűnik, kivéve a CALL utasítást. Ez az egyoperandusú utasítás szolgál az eljárások végrehajtására, más szóval az **eljáráshívásra** (procedure call). Működése: a verembe eltárolja az IP aktuális értékét (tehát szimbolikusan végrehajt egy PUSH IP-t), majd IP-t beállítja az operandus által mutatott címre, és onnan folytatja az utasítások végrehajtását. A célcímet a JMP-hez hasonlóan 16 bites előjeles relatív címként tárolja. (Létezik távoli formája is, ekkor CS-t is lerakja a verembe az IP *előtt*, illetve beállítja a cél kódszegmenst is.) Az operandus helyén most egy eljárás neve áll (ami végül is egy olyan címke, ami az eljárás kezdetére mutat). Két eljárást írtunk, a DecKiir végzi el AX tartalmának kiírását, az UjSor pedig a képernyő következő sorára állítja a kurzort. Tesztelés céljából a 0, 8086, 32768 és 65535 értékeket íratjuk ki.

Most térjünk vissza a program elejére, ahol az eljárásokat helyeztük el (egyébként bárhol lehetnének, ez csak megszokás kérdése). Eljárást a következő módon definiálhatunk:

```
Név             PROC        {opciók}
eljárástörzs
Név             ENDP
```

Az eljárás törzsét (azaz annak működését leíró utasításokat) a PROC ... ENDP direktívák (nem pedig mnemonikok!) fogják közre. Van néhány olyan rész, ami a legtöbb eljárás törzsében közös. Az eljárás elején szokásos az eljárásban később módosított regiszterek tartalmát a verembe elmenteni, hogy azokat később visszaállíthassuk, és a hívó program működését ne zavarjuk be azzal, hogy a regiszterekbe zagyvaságokat teszünk. Természetesen ha egy regiszterben akarunk visszaadni valamilyen értéket, akkor azt nem fogjuk elmenteni. Az eljárásból a hívóhoz visszatérés előtt a verembe berakott regiszterek tartalmát szépen helyreállítjuk, méghozzá pontosan a berakás fordított sorrendjében.

A DecKiir eljárásban a 4 általános adatregisztert fogjuk megváltoztatni, ezért ezeket szépen PUSH-sal berakjuk a verembe az elején, majd az eljárás végén fordított sorrendben POP-pal kivesszük.

Bináris számot úgy konvertálunk decimálisra, hogy a kiinduló számot elosztjuk maradékosan 10-zel, és a maradékot szépen eltároljuk. Ha a hányados nulla, akkor készen vagyunk, különben a hányadost tekintve az új számnak, azon folytatjuk a 10-zel való osztogatást. Ha ez megvolt, akkor nincs más dolgunk, mint hogy az osztások során kapott maradékokat a megkapás fordított sorrendjében egymás mellé írjuk mint számjegyeket. Így tehát az első maradék lesz a decimális szám utolsó jegye.

Az osztót most BX-ben tároljuk, CX pedig az eltárolt maradékokat (számjegyeket) fogja számolni, ezért kezdetben kinullázzuk.

Az osztásokat egy előfeltételes ciklusba szervezve végezzük el. A ciklusnak akkor kell megállnia, ha AX (ami kezdetben a kiírandó szám, utána pedig a hányados) nullává válik. Ha ez teljesül, akkor kiugrunk a ciklusból, és a @CiklVege címkére ugrunk.

Ha AX $\neq$ 0000h, akkor osztanunk kell. A DIV (unsigned DIVision) utasítás szolgál az előjeltelen osztásra. Egyetlen operandusa van, ami 8 vagy 16 bites regiszter vagy memóriahivatkozás lehet. Ha az operandus bájt méretű, akkor AX-et osztja el az adott operandussal, a hányadost AL-be, a maradékot pedig AH-ba teszi. Ha szavas volt az operandus, akkor DX:AX-et osztja el az operandussal, a hányados AX-be, a maradék DX-be kerül. A 6 aritmetikai flag értékét elrontja. Nekünk most a második esetet kell választanunk, mert előfordulhat, hogy az első néhány osztás hányadosa nem fog elférni egy bájtban. Így tehát BX-szel fogjuk elosztani DX:AX-et. Mivel a kiinduló számunk csak 16 bites volt, a DIV viszont 32 bites számot követel meg, DX-et az osztás elvégzése előtt törölnünk kell.

A maradékokat a veremben fogjuk tárolni az egyszerűbb kiíratás kedvéért, ezért DX-et berakjuk oda, majd a számlálót megnöveljük. Végül visszaugrunk az osztó ciklus elejére.

A számjegyek kiírására a legegyszerűbb módszert választjuk. Már említettük, hogy az INT 21h-n keresztül a DOS szolgáltatásait érhetjük el. A 02h számú szolgáltatás (azaz ha AH = 02h) a DL-ben levő karaktert kiírja a képernyőre az aktuális kurzorpozícióba. AH-ba ezért most berakjuk a szolgáltatás számát.

Ha az eljárás hívásakor AX nulla volt, akkor az osztó ciklus egyszer sem futott le, hanem rögtön az elején kiugrott. Ekkor viszont CX = 0, hiszen így állítottuk be. Ezek hatására a JCXZ utasítás segítségével a @NullaVolt címkén folytatjuk az eljárás végrehajtását, ahol a 21h-s szoftver-megszakítás segítségével kiírjuk azt az egy számjegyet, majd „rácsorgunk" a @KiirVege címkére.

Ha nemzéró értékkel hívtuk meg eljárásunkat, akkor itt az ideje, hogy kiírjuk a CX db. jegyet a képernyőre. Ezt egy egyszerű számlálásos ismétléses vezérléssel (LOOP ciklussal)

elintézhetjük. Mivel a verem LIFO működésű, így a legutoljára betett maradékot vehetjük ki legelőször, és ezt is kell első számjegyként kiírnunk. Az érvényes decimális jeggyé konvertálást az ADD DL,'0' utasítás végzi el, majd a karaktert megszakítás-hívással kiküldjük a monitorra. A kiírás végeztével szintén a @KiirVege címkére megyünk.

Miután visszaállítottuk a módosított eredeti regiszterek tartalmát, vissza kell térnünk arra a helyre, ahonnan meghívták az eljárást. Erre szolgál a RET (RETurn) utasítás. Két változata van: ha nem írunk mellé operandust, akkor a veremből kiszedi IP-t (amit előzőleg a CALL rakott oda), és onnan folytatja a végrehajtást. De írhatunk egy 16-bites közvetlen értéket (numerikus kifejezést) is operandusként, ekkor az IP kiszedése *után* ezt a számot hozzáadja SP-hez (olyan, mintha Szám/2 db. POP-ot hajtana végre), majd az új CS:IP címre adja a vezérlést. Mi most nem akarunk változtatni a veremmutatón, így az egyszerű RET-tel visszatérünk a főprogramba.

Az UjSor eljárás nagyon egyszerű és rövid. A regiszterek elmentésén ill. visszaállításán kívül csak két megszakítás-hívást tartalmaz, amik a 0Dh és 0Ah ASCII kódú karaktereket írják ki a képernyőre. Az első karakter az ú.n. kocsivissza (CR – Carriage Return), a másik pedig a soremelés (LF – Line Feed). Ezek alkotják az **új sor** (new line) karakterpárt.

Ha sok regisztert kell elmentenünk és/vagy visszaállítanunk, akkor fárasztó és felesleges minden egyes regiszterhez külön PUSH vagy POP utasítást írni. Ezt megkönnyítendő, a TASM lehetővé teszi, hogy egynél több operandust írjunk ezen utasítások után, az egyes operandusokat egymástól *szóközzel* elválasztva. Így a PUSH AX BX CX DX // POP SI DI ES utasítások ekvivalensek a következő sorozattal:

```
PUSH    AX
PUSH    BX
PUSH    CX
PUSH    DX
POP     SI
POP     DI
POP     ES
```

9. fejezet

A Turbo Debugger használata

Az eddig megírt programjaink működését nem tudtuk ellenőrizni, úgy pedig elég kényelmetlen programozni, hogy nem látjuk, tényleg azt csinálja-e a program, amit elvárunk tőle. Ha valami rosszul működik, netán a számítógép semmire sem reagál (ezt úgy mondjuk, hogy „kiakadt” vagy „lefagyott”), a hiba megkeresése egyszerűen reménytelen feladat segítség nélkül. Ezt a természetes igényt elégítik ki a különböző debugger (nyomkövető, de szó szerint „bogártalanító”) szoftverek ill. hardverek. (Az elnevezés még a számítástechnika hőskorszakából származik, amikor is az egyik akkori számítógép működését valamilyen rovar zavarta meg. Azóta hívják a hibavadászatot „bogárirtásnak”.)

A Turbo Assemblert és Linkert készítő Borland cég is kínál egy szoftveres nyomkövető eszközt, Turbo Debugger (TD) néven. Most ennek használatával fogunk megismerkedni.

A Turbo Debugger fő tulajdonsága, hogy képes egy futtatható állományt (.EXE vagy .COM kiterjesztéssel) betölteni, majd annak gépi kódú tartalmát Assembly forrásra visszafejteni. Ezt hívjuk **disassemblálásnak** (disassembly) vagy szebb magyar szóval a forrás **visszafejtésének**. A legszebb a dologban az, hogy a programban szereplő kód- és memóriahivatkozásokat konkrét számok helyett képes az eredeti forrásban szereplő szimbólumokkal megjeleníteni. Ezenkívül minden utasítást egyenként hajthatunk végre, ha akarjuk, többször is, sőt megadhatjuk, hogy a program végrehajtása egy bizonyos feltétel teljesülése esetén szakadjon meg. Figyelhetjük a memória tetszőleges területének tartalmát, a regiszterek és flag-ek értékeit, s mindezeket meg is változtathatjuk. A program futását az eredeti forrásokon is képes követni. Szóval csupa-csupa hasznos szolgáltatással bír, amikkel pont olyan kényelmesen figyelhetjük programunk tevékenységét, mintha mondjuk a Borland C fejlesztő rendszerében (IDE) dolgoznánk.

Ahhoz hogy mindezt a kényelmet élvezhessük, nem kell mást tenni, mint:

- minden forrást a /zi vagy /zd kapcsolóval lefordítani
- a tárgykódokat a /v kapcsolóval összeszerkeszteni

Ha ezt a két műveletet elvégeztük, akkor a .EXE állományunk (remélhetőleg) tartalmazza az összes szimbolikus információt, amire a nyomkövetés során szükség lehet.

A dolognak két hátránya van: egyrészt .COM programokba nem kerülhet nyomkövetési info, másrészt ez az adathalmaz az .EXE fájl méretét igencsak megnövelheti (előfordulhat, hogy az eredeti többszöröse lesz a kimenet mérete).

A legelső példaprogramot (Pelda1.ASM) begépeltük és elmentettük PELDA.ASM néven, majd lefordítottuk és linkeltük, az összes debug információt belerakva. A

```
TD PELDA.EXE
```

parancs kiadása utáni állapotot tükrözi a 9.1. kép.

A képernyő tetején ill. alján a már megszokott menüsor és státuszsor találhatók.

A kép nagy részét elfoglaló munkaasztalon (desktop) helyezkednek el a különféle célt szolgáló ablakok.

A legfelső ablak (CPU ablak) több részre osztható. A bal felső sarok az aktuális kód disassemblált változatát mutatja, és most éppen úgy van beállítva, hogy a forrás eredeti sorait is megjeleníti. A következő végrehajtandó sor a bal oldalon egy kis jobbra mutató háromszöggel van megjelölve (ez most a CS:0009h című sor).

Emellett az egyes regiszterek és flag-ek tartalma látható. Az előző állapothoz képest megváltozott dolgokat fehér színnel jelöli meg.

A bal alsó sarok a memória egy adott területének tartalmát mutatja hexadecimális és ASCII alakban (ez az ú.n. memory dump).

Végül a jobb alsó sarokban a verem aktuális állapotát szemlélhetjük meg. A veremmutatót (azaz a verem tetejét) szintén egy jobbra mutató sötét háromszög jelzi.

A kép közepén levő nagyobb ablakban a forrásban követhetjük nyomon a program futását. Itt mindig azt a fájlt látjuk, amihez az éppen végrehajtott kód tartozik. A következő végrehajtandó sort a bal oldalon kis fehér háromszög mutatja.

Az alatta látható, Stack címkéjű ablak a különböző eljárás- és függvényhívások során a verembe rakott argumentumokat mutatja.

A legalsó, keskeny ablak a Watches címkét viseli. Ennek megfelelően az általunk óhajtott változók, memóriaterületek aktuális értékét követhetjük itt figyelemmel.

A felső sorban található menürendszeren kívül minden ablakban előhívható egy helyi menü (local menu) az <ALT>+<F10> billentyűkombinációval, ahol az aktuális ablakban (vagy részablakban) rendelkezésre álló plusz szolgáltatásokat érhetjük el. Ilyenek pl. az utasítás assemblálása, regiszter tartalmának megváltoztatása, flag törlése, megfigyelendő változó felvétele a Watches listába stb.

Az ablakok között az <F6> és <Shift>+<F6> billentyűkkel mozoghatunk, míg az aktuális ablakon belül a <Tab> és a <Shift>+<Tab> kombinációkkal lépkedhetünk a részablakok között.

Most áttekintjük az egyes menük funkcióit. A File menü a szokásos állományműveleteket tartalmazza, de itt kaphatunk információt a betöltött programról is.

Az Edit menü szintén a gyakori másolás-beszúrás típusú funkciókat rejti.

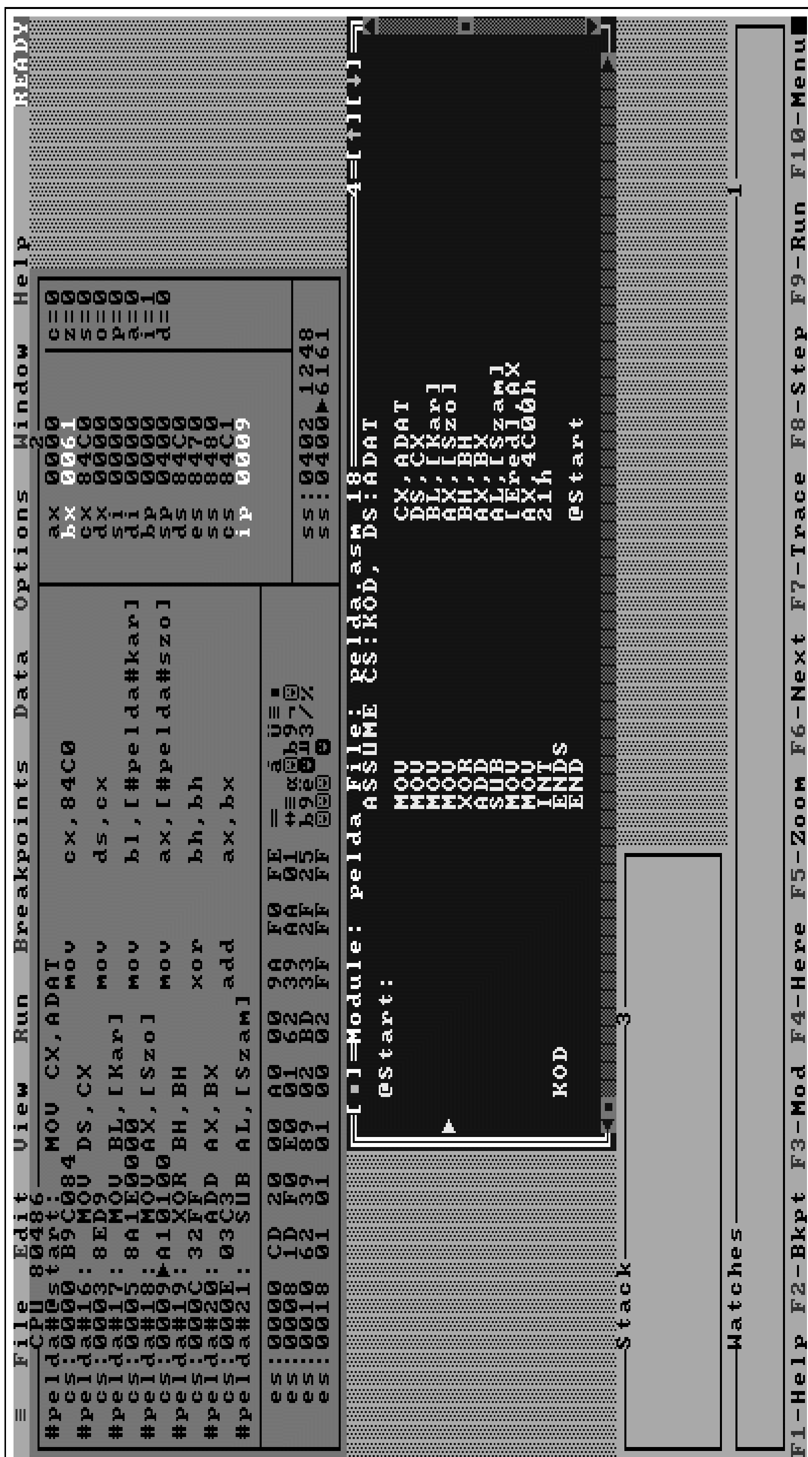
A View menü készlete igen gazdag. Innen nézhetjük meg a változókat (Variables), a CPU ablakot, másik programmodult, tetszőleges állományt és még sok minden mást.

A Run menüben, mint neve is sejteti, a futtatással kapcsolatos tevékenységek vannak összegyűjtve, de itt állíthatók be a program futtatási (parancssoros) argumentumai is. Szinte az összes itteni szolgáltatáshoz tartozik valamilyen gyorsbillentyű (hot key) is. Néhány ezek közül: futtatás <F9>, újraindítás <Ctrl>+<F2>, adott sorig végrehajtás <F4>, lépésenkénti végrehajtás <F7>, CALL és INT utasítások átugrása <F8>.

A Breakpoints menüvel a töréspontokat tarthatjuk karban. A **töréspont** egy olyan hely a kódban, ahol a program végrehajtásának valamilyen feltétel teljesülése esetén (vagy mindenképpen) meg kell szakadnia. Ilyenkor a vezérlést ismét visszakapjuk a TD képernyőjével együtt, és kedvünk szerint beavatkozhatunk a program menetébe.

A Data menü az adatok, változók manipulálását segíti.

Az Options menüben találhatók a TD beállításai. Ilyenek pl. a forrás nyelve (Language), helye (Path for source), képernyővel kapcsolatos dolgok (Display options).



9.1. ábra. A Turbo Debugger képernyője

A Window menü az ablakokkal való bűvészkedésre jó, a Help menü pedig a szokásos súgót tartalmazza.

A TD egyébként nemcsak Assembly, de Pascal és C nyelvű programot is képes nyomon követni, és az ezekben a nyelvekben meglevő összes adattípust is képes kezelni.

10. fejezet

Számolás előjeles számokkal, bitműveletek

Ebben a fejezetben a gyakorlati problémák megoldása során gyakran előforduló feladatokkal foglalkozunk. Teljes példaprogramokat most nem közlünk, a megoldás módszerét egy-egy rövid programrészleten fogjuk bemutatni.

10.1. Matematikai kifejezések kiértékelése

Az első problémakört a különböző előjelű számokat tartalmazó matematikai kifejezések kiértékelése alkotja. Például tegyük fel, hogy AL-ben van egy előjeles, míg BX-ben egy előjel nélküli érték, és mi ezt a két számot szeretnénk összeadni, majd az eredményt a DX:AX regiszterpárban tárolni. Az ehhez hasonló feladatoknál mindig az a megoldás, hogy a két összeadandót azonos méretűre kell hozni. Ez egész pontosan két dolgot jelent: az előjeles számokat előjelesen, az előjelteleneket zéró-kiterjesztésnek (előjeltelen-kiterjesztésnek) kell alávetni. Ismétlésként: zéró-kiterjesztésen (zero extension) azt értjük, amikor az adott érték felső, hiányzó bitjeit csupa 0-val töltjük fel, ellentétben az előjeles kiterjesztéssel, ahol az előjelbitet használjuk kitöltő értékként. Azt is vegyük figyelembe, hogy az eredmény (összeg) mindig hosszabb lesz 1 bittel mint a kiinduló tagok hosszának maximuma. Ha ez megvan, akkor jöhet a tényleges összeadás. Itt figyelnünk kell arra, mit és milyen sorrendben adunk össze. Először az alsó bájtokon/szavakon végezzük el a műveletet. Itt kapunk egy részeredményt, valamint egy esetleges átvitelt, amit a felső bájtok/szavak összeadásakor is figyelembe kell venni.

Ezek alapján nézzünk egy lehetséges megoldást:

```
CBW
CWD
ADD     AX,BX
ADC     DX,0000h
```

Először tisztázzuk az eredmény méretét. Az előjeles szám 8 bites, az előjeltelen 16 bites, ebből 17 bit jön ki. Az alsó 16 bit meghatározása nem nagy kunszt, a CBW utasítás szépen kiterjeszti előjelesen AL-t AX-be, amihez aztán hozzáadhatjuk BX tartalmát. Az eredmény alsó 16 bitje ezzel már megvan. A legfelső bit azonban, mint gondolhatnánk, nem maga az átvitel lesz.

Lássuk mondjuk, mi lesz, ha AL = −1, BX = 65535. AX-ben előjeles kiterjesztés után 0FFFFh lesz, de ugyanezt fogja BX is tartalmazni. A két számot összeadva 0FFFEh-t kapunk, és CF = 1 lesz. Látható, hogy a helyes eredmény is 0 FFFEh lesz, nem pedig 1 FFFEh. A megoldás kulcsa, hogy az eredményt 24 vagy 32 bitesnek tekintjük. Most az utóbbit választjuk, hiszen DX:AX-ben várjuk az összeget. Innen már következik a módszer: mindkét kiinduló számot 32 bitesnek képzeljük el, s az összeadást is eszerint végezzük el. Az AX-ben levő előjeles számot az CWD (Convert Word to Doubleword) operandus nélküli utasítás DX:AX-be előjelesen kiterjeszti, az összes flag-et békén hagyva. A másik, BX-ben levő tagnak zéró-kiterjesztésen kellene átesnie, amit mi most kihagyunk, de az összeadás során figyelembe fogunk venni. Miután AX-ben képeztük az eredmény alsó szavát, itt az ideje, hogy a felső szavakat is összeadjuk az átvitellel együtt. A kétoperandusú ADC (ADd with Carry) utasítás annyiban tér el az ADD-től, hogy a célhoz CF zéró-kiterjesztett értékét is hozzáadja. Az előjeles tag felső szava már DX-ben van, a másik tag felső szava azonban 0000h. Ezért az utolsó utasítással DX-hez hozzáadjuk a közvetlen adatként szereplő nullát és CF-et is. Ha mindenképpen zéró kiterjesztést akarunk, akkor így kell módosítani a programot:

```
CBW
CWD
XOR     CX,CX
ADD     AX,BX
ADC     DX,CX
```

CX helyett persze használhatunk más regisztert is a nulla tárolására. A példát befejezve, DX 0FFFFh-t fog tartalmazni a CWD hatására, amihez a 0000h-t és az átvitelt hozzáadva DX is nullává válik. DX:AX így a helyes eredményt fogja tartalmazni, ami 0000 FFFEh.

A feladatban helyettesítsük most az összeadást kivonással, tehát szeretnénk az AL-ben levő előjeles számból kivonni a BX-ben levő előjel nélküli számot, az eredményt ugyancsak DX:AX-ben várjuk. A megoldás a következő lehet:

```
CBW
CWD
SUB     AX,BX
SBB     DX,0000h
```

Teljesen világos, hogy az összeadásokat a programban is kivonásra kicserélve célhoz érünk. Az ADC párja az SBB (SuBtract with Borrow), ami a SUB-tól csak annyiban tér el, hogy a célból CF zéró-kiterjesztett értékét is kivonja.

Mind a 4 additív utasítás az összes aritmetikai flag-et, tehát a CF, PF, AF, ZF, SF és OF flag-eket módosítja. Operandusként általános regiszteren és memóriahivatkozáson kívül konstans értéket is kaphatnak.

Térjünk most rá a szorzásra és osztásra. A gépi aritmetika tárgyalásánál már említettük, hogy szorozni és osztani sokkal körülményesebb, mint összeadni és kivonni. A gondot az előjeles és előjeltelen számok csak tovább bonyolítják. Azt is említettük, hogy előjeles esetben a tagok előjelét le kell választani a műveletek elvégzéséhez, miután megállapítottuk az eredmény előjelét. (Ez persze nem a mi dolgunk, a processzor elvégzi helyettünk.) Ezen okból mind szorzásból mind osztásból létezik előjeles és előjel nélküli változat is. Mindegyik utasításban közös, hogy az egyik forrás tag és az eredmény helye is rögzítve van, továbbá mindegyik uta-

sítás egyetlen operandust kap, ami csak egy általános regiszter vagy memóriahivatkozás lehet. Közvetlen értékkel tehát *nem* szorozhatunk, de *nem* is oszthatunk!

Nézzük meg először az előjel nélküli változatokat, ezek az egyszerűbbek.

Szorozni a MUL (unsigned MULtiplication) utasítással tudunk. Ennek egyetlen operandusa az egyik szorzó tagot (multiplier) tartalmazza. Az operandus mérete határozza meg a továbbiakat: ha 8 bites az operandus, akkor AL-t szorozza meg azzal, az eredmény pedig AX-be kerül. Ha szó méretű volt az operandus, akkor másik tagként AX-et használja, az eredmény pedig DX:AX-ben keletkezik.

Osztásra a DIV (unsigned DIVision) utasítás szolgál, ezzel már korábban találkoztunk, de azért felelevenítjük használatát. Az egyetlen operandus jelöli ki az osztót (divisor). Ha ez bájt méretű, akkor AX lesz az osztandó (dividend), és a hányados (quotient) AL-be, a maradék (remainder) pedig AH-ba fog kerülni. Ha az operandus szavas volt, akkor DX:AX-et osztja el vele, majd a hányados AX-be, a maradék pedig DX-be kerül.

Az előjeles esetben mindkét utasítás ugyanazokat a regisztereket használja a forrás illetve a cél tárolására, mint az előjel nélküli változatok.

Előjeles szorzást az IMUL (Integer signed MULtiplication) utasítással hajthatunk végre. Az eredmény előjele a szokásos szabály szerint lesz meghatározva, tehát a két forrás előjelbitjét logikai KIZÁRÓ VAGY kapcsolatba hozva kapjuk meg.

Végül előjelesen osztani az IDIV (Integer signed DIVision) utasítás alkalmazásával tudunk. A hányados előjele ugyanúgy lesz meghatározva, mint az IMUL esetén, míg a maradék az osztandó előjelét örökli.

Ezek az utasítások elég „rendetlenül” módosítják a flag-eket: a szorzások a CF és OF flag-et változtatják meg, míg a PF, AF, ZF és SF flag-ek értéke meghatározatlan, az osztó utasítások viszont az összes előbb említett flag-et definiálatlan állapotban hagyják (azaz nem lehet tudni, megváltozik-e egy adott flag értéke, s ha igen, mire és miért).

Ha a szorzat nagyobb lenne, mint a források mérete, akkor a MUL és IMUL utasítások mind CF-et, mind OF-et 1-re állítják. Különben mindkét flag törlődni fog. Ez egész pontosan azt jelenti, hogy ha bájtos esetben AH, szavas esetben pedig DX értékes biteket tartalmaz a szorzás elvégzése után, akkor állítják be a két említett flag-et a szorzó utasítások.

Az utasítások használatának szemléltetésére nézzünk meg egy kicsit összetett példát! Tegyük fel, hogy a következő kifejezés értékét akarjuk meghatározni:

$$\frac{A \cdot B + C \cdot D}{(E - F)/G}$$

A betűk hét számot jelölnek, ezek közül A és B előjel nélküli bájtok, C és D előjeles bájtok, E és F előjel nélküli szavak, és végül G előjel nélküli bájt. Feltesszük, hogy E nagyobb vagy egyenlő F-nél. Az eredmény egy előjeles szó lesz, amit műveletek elvégzése után AX-ben kapunk meg. Az osztásoknál a maradékkal nem törődünk, így az eredmény csak közelítő pontosságú lesz. Hasonlóan nem foglalkozunk az esetleges túlcsordulásokkal sem. Lássuk hát a megoldást:

```
MOV     AX,[E]
SUB     AX,[F]
DIV     [G]
MOV     CL,AL
XOR     CH,CH
MOV     AL,[A]
MUL     [B]
MOV     BX,AX
```

```
MOV     AL,[C]
IMUL    [D]
CWD
ADD     AX,BX
ADC     DX,0000h
IDIV    CX
```

Érdemes átgondolni, hogy a 6 műveletet milyen sorrendben célszerű elvégezni. Először most az E-F kivonást hajtjuk végre, amit rögvest elosztunk G-vel, az eredményt berakjuk CL-be, s ezt rögtön zéró-kiterjesztésnek vetjük alá, azaz CH-t kinullázzuk. Ezután, hogy minél kevesebb regiszter-művelet legyen, az előjeltelen szorzást végezzük el előbb, az eredményt BX-ben tároljuk. Most jön a másik, előjeles szorzat kiszámolása, aminek az eredményét rögtön előjelesen kiterjesztjük DX:AX-be, hogy az összeadást végre tudjuk hajtani. Az összeg meghatározása után elvégezzük a nagy előjeles osztást, s ezzel AX-ben kialakul a végleges eredmény.

A teljes korrektség kedvéért megjegyezzük, hogy ez a példa két helyen „sántít”. Nevezetesen az osztásoknál nem biztos, hogy a hányados el fog férni a számára kijelölt helyen, és ez bizony súlyos hibához vezethet (a kíváncsiaknak eláruljuk, hogy ilyen esetben egy osztási kivétel keletkezik). Ennek ellenőrzésétől, kikerülésétől azonban most eltekintünk.

Utolsó esetként megnézzük, hogyan képezhetjük egy szám additív inverzét, vagy magyarosabban a -1-szeresét. (Szintén helyes, ha azt mondjuk, hogy képezzük a szám kettes komplemensét.) Erre külön utasítás szolgál. Az egyoperandusú NEG (NEGate) utasítás az operandusát kivonja 0-ból, az eredményt pedig visszaírja az operandusba. Operandusként általános regisztert vagy memóriahivatkozást adhatunk meg. Nem véletlen, hogy „kivonást” írtunk, ugyanis ha az operandus egyenlő 0-val, akkor CF törlődni fog. Ha az operandus nemzéró, CF értéke *minden esetben* 1 lesz. Az utasítás különben mind a 6 aritmetikai flag-et az eredménynek megfelelően megváltoztatja.

Első példaként tegyük fel, hogy a AX-ben levő értéket szeretnénk kivonni a 0-ból. A megoldás igen egyszerű:

```
NEG     AX
```

Kicsit bonyolítsunk a helyzeten. Most a cél a DX:AX regiszterpárban levő szám inverzének meghatározása legyen. Első gondolatunk a következő:

```
NEG     AX
NEG     DX
```

Ez azonban *rossz* megoldás! Miért? Próbáljuk ki mondjuk a -32768-ra. Ennek a számnak a 0FFFF 8000h felel meg, tehát DX = 0FFFFh és AX = 8000h. Tudjuk, hogy az eredménynek +32768 = 0000 8000h-nak kell lennie. Ha a fenti két utasítást elvégezzük, rossz eredményként 0001 8000h-t kapunk. A hiba abban van, hogy a kiinduló szám 32 bites, amit mi két különálló 16 bites számként kezeltünk.

Ha visszaemlékszünk arra, hogyan is definiáltuk a kettes komplemens fogalmát, rálelhetünk egy jó megoldásra:

```
NOT     AX
NOT     DX
```

```
ADD     AX,0001h
ADC     DX,0000h
```

Először tehát képezzük a kiinduló érték egyes komplemensét, majd ezt megnöveljük eggyel. Ez egész biztosan helyes eredményt szolgáltat.

Egy másik megoldást is bemutatunk:

```
NEG     AX
ADC     DX,0000h
NEG     DX
```

Ha nem világos, miért is működik ez, akkor gondolkozzunk el rajta, hogyan végezzük el a kivonást binárisan. A feladatban a kiinduló számot (DX:AX) ki kell vonni 0-ból. Ezt csak két lépésben tehetjük meg, de arra vigyázni kell, hogy a szám alsó és felső szava közti kapcsolatot fenntartsuk. Ez jelen esetben annyit jelent, hogy az alsó szó kivonása (azaz NEG AX elvégzése) után az esetleges átvitelt le kell vonnunk a felső szóból. Mi most hozzáadtunk, de ez így helyes, hiszen $-(\mathrm{DX} + \mathrm{CF}) = -\mathrm{DX} - \mathrm{CF}$, tehát mégiscsak levontuk az átvitelt.

10.2. BCD aritmetika

Mivel a hétköznapokban általában a decimális (10-es alapú) számrendszert használjuk, kényelmesebb lenne, ha a számítógépet is rá tudnánk venni, hogy ezt alkalmazza a bináris helyett. Nos, a BCD (Binary Coded Decimal) aritmetika pont ezt támogatja. A kifejezés jelentése egyértelmű: binárisan kódolt decimális. Azaz arról van szó, hogy bár a processzor bináris regisztereit és utasításait használjuk, de az adatokat (számokat) decimális alakban adjuk meg.

Ennek használatához két új adattípus áll rendelkezésre. Lehetőségünk van 8 biten egy vagy két decimális jegy tárolására is. Ennek alapján beszélhetünk **pakolt** (csomagolt) vagy **pakolatlan** (csomagolatlan) BCD számokról. Pakolt esetben két decimális jegyet tárolunk egyetlen bájtban, a bájt alsó felén a kisebb helyiértékű, felső felén (4 bitjén) pedig a magasabb helyiértékű jegyet. Decimális jegy alatt egy 0h – 9h közötti értéket értünk, de ez *nem azonos* a „0” – „9” karakterekkel! (Ez azért van, mert az ASCII kódtáblázat szerint a „0” karakter kódja 48d, avagy 30h.) Pakolatlan esetben a bájt felső 4 bitjének nincs szerepe, ott nem tárolunk semmit sem.

Fontos, hogy *csak nemnegatív* értékeket tárolhatunk ilyen módon!

A tényleges utasítások ismertetése előtt kezdjük egy példával. Vegyünk két számot, ezek legyenek mondjuk a 28 és a 13. Pakolatlan alakban tároljuk el őket mondjuk az AX és BX regiszterekben. Ezt annyit jelent az előzőek alapján, hogy AX = 0208h és BX = 0103h. Adjuk össze a két számot az ADD AH,BH // ADD AL,BL utasításokkal! Ennek hatására AX tartalma 030Bh lesz a bináris összeadás szabályai miatt. A helyes eredmény a 41 lenne, de nem ezt kaptuk. Nem véletlenül, hiszen az összeg alsó bájtja érvénytelen, mivel nem decimális jegyet tartalmaz. Ennek korrigálását végzi el az AAA (ASCII Adjust after Addition) operandus nélküli utasítás. Működése röviden: ha volt decimális átvitel (azaz AF = 1) vagy AL alsó bitnégyese érvénytelen (azaz (AL AND 0Fh) > 9), akkor AL-hez hozzáad 6-ot, megnöveli AH-t, majd AF-be és CF-be 1-et tölt. Különben a két flag értéke 0 lesz. A végén törli AL felső 4 bitjét. Esetünkben AL alsó fele érvénytelen, ezért hozzáad ahhoz 6-ot (0311h), megnöveli AH-t (0411h), kitörli AL felső felét (0401h), AF-et és CF-et pedig 1-re állítja. Közben AX-ban kialakul az immár helyes végeredmény.

Ismét vegyünk két számot, tekintsük mondjuk a 45-öt és a 17-et. Szintén pakolatlan alakban tároljuk el őket AX-ben és BX-ben. Most tehát AX = 0405h, BX = 0107h. Vonjuk ki a kisebbet a nagyobból a SUB AH,BH // SUB AL,BL utasítással! Hatására AX tartalma 03FEh lesz, ami igen furcsa eredmény. Ezt a szintén operandus nélküli AAS (ASCII Adjust after Subtraction) utasítással hozhatjuk helyre. Működése: ha AL alsó négy bitje érvénytelen jegyet tartalmaz, vagy AF = 1, akkor AL-ből kivon 6-ot, csökkenti AH-t, AF-et és CF-et pedig 1-re állítja. Különben a két flag tartalma 0 lesz. A végén törli AL felső 4 bitjét. A példánknál maradva, itt mindkét feltétel teljesül, ezért kivon AL-ből 6-ot (03F8h), csökkenti AH-t (02F8h), törli AL felső felét (0208h), AF-et és CF-et pedig 1-re állítja. AX-ban így kialakul a helyes eredmény.

Kérdezhetné valaki, hogy miért nem a SUB AX,BX utasítással végeztük el a kivonást. A válasz egyszerű: azért, mert akkor eredményül 02FEh-t kaptunk volna, amit az AAS 0108h-ra alakítana, és mi nem ezt az eredményt várjuk. A kulcs itt az átvitelben van, amit mi most nem akarunk a szám felső bájtjába átvinni. Másrészt pedig az alsó jegyeket kell *később* kivonni, mivel az AAS működése az AF értékétől is függ. Ha a felső jegyeket vonnánk ki később, akkor AF tartalma ez utóbbi kivonás szerint állna be, ami helytelenné tenné a végső eredményt. Ezt szemléltetendő hajtsuk végre a SUB AL,BL // SUB AH,BH utasításokat az AX = 0200h, BX = 0009h értékekre! Hatásukra AX = 02F7h és AF = 0 lesz. Ha erre hajtjuk végre az AAS utasítást, akkor annyit változik a helyzet, hogy AX felső 4 bitje törlődik, 0207h-t eredményezve. Ha a két utasítást fordított sorrendben végezzük el, akkor AX változatlanul 02F7h-t fog tartalmazni, de most AF értéke 1 lesz. Ekkor alkalmazva az AAS-t, az korrigálni fogja AX-et, a helyes 0101h eredményt előállítva benne.

Most legyen AL = 04h, BL = 08h. Szorozzuk össze a két számot a MUL BL utasítással! Ennek eredményeként AX tartalma 0020h lesz, ami a 32 bináris alakja. Ezt az értéket szeretnénk pakolatlan BCD alakra hozni. Az operandus nélküli AAM (ASCII Adjust after Multiplication) utasítás AL tartalmát maradékosan elosztja 10-zel, a hányadost AH-ba, a maradékot pedig AL-be téve. (Vigyázat, ez nem elírás, itt az osztásoktól eltérően tényleg fordítva helyezkedik el a hányados és a maradék!) A példánkra alkalmazva AX tartalma 0302h lesz.

Most osztani fogunk. Legyen AX = 0205h, BL = 05h. El akarjuk osztani az első számot a másodikkal. Osztani csak bináris alakban tudunk, ehhez viszont mindkét tagot erre az alakra kell hozni. Az 5-tel nincs is gond, a 25-öt viszont át kell alakítani. Erre kényelmes megoldás az ismét csak operandus nélküli AAD (ASCII Adjust before Division) utasítás. Hatására AL tartalma az AH · 10 + AL kifejezés értéke lesz, AH pedig kinullázódik. Az AAD után már elvégezhetjük az osztást a DIV BL utasítással. Ennek eredményeként AX tartalma 0005h lesz, ez a helyes eredmény.

A szorzásra és osztásra itt most nagyon egyszerű példákat írtunk csak. A bonyolultabb esetekkel nem foglalkozunk, ezt az Olvasóra hagyjuk. Nagyszerű lehetőség a gondolkodásra és gyakorlásra egyaránt.

Az AAA, AAS és AAM utasításokat mindig az adott művelet *utáni*, az AAD utasítást pedig a művelet *előtti* korrekcióra használjuk! Az AAA és AAS csak a CF és AF flag-eket változtatja „logikusan”, a maradék 4 aritmetikai flag értéke meghatározatlan. Ezzel szemben az AAM és AAD utasítások PF, SF és ZF tartalmát változtatják definiáltan, a maradék 3 aritmetikai flag értéke itt is definiálatlan lesz a végrehajtás után.

Eddig a pakolatlan BCD számok használatát mutattuk be. Most lássuk, mit művelhetünk a pakolt BCD alakkal! Ismét tekintsük első példánkat, azaz a 28-as és 13-as számokat! Tároljuk őket AL-ben és BL-ben, tehát legyen AL = 28h, BL = 13h. Az ADD AL,BL utasítással elvégezve az összeadást AL tartalma 3Bh lesz, ami persze nem helyes eredmény. Ezt most az operandus nélküli DAA (Decimal Adjust after Addition) utasítással hozhatjuk helyre. Ennek működése

már kicsit bonyolultabb: ha AL alsó fele érvénytelen jegyet tartalmaz, vagy AF = 1, akkor AL-hez hozzáad 6-ot, AF-be pedig 1-et tölt. Különben AF értéke 0 lesz. Ezután ha AL felső 4 bitje érvénytelen jegyet tartalmaz, vagy CF = 1, akkor AL-hez hozzáad 60h-t, és CF-et 1-re állítja. Különben CF értéke 0 lesz. Nem nehéz, egyszerűen szükség szerint korrigálja mindkét jegyet. Fontos, hogy működése közben az „összeadásokat" úgy végzi el, hogy a flag-ek tartalma nem módosul. Példánkhoz visszatérve, itt most CF = AF = 0, és az alsó jegy érvénytelen. Ezért a DAA hozzáad AL-hez 6-ot (41h), AF-et 1-re állítja, és kész.

A kivonáshoz tekintsük a második példát, tehát a 45-ös és 17-es számokat. Ehhez legyen AL = 45h, BL = 17h. Vonjuk ki a kisebbet a nagyobból a SUB AL,BL utasítással. Eredményként AL tartalma 2Eh lesz. Ezt ismét korrigálni kell az operandus nélküli DAS (Decimal Adjust after Subtraction) utasítás segítségével. Működése teljesen megfelel a DAA működésének, annyi eltéréssel, hogy 6-ot illetve 60h-t von ki az alsó és a felső jegyből szükség esetén. A példában CF = 0 és AF = 1, így a DAS kivon AL-ből 6-ot (28h), AF és CF értéke változatlan marad.

Mindkét utasítás módosítja a CF, PF, AF, ZF és SF flag-eket, OF tartalma meghatározatlan lesz végrehajtásuk után.

Szorzás és osztás esetére nincs ilyen korrigáló utasítása a processzornak, így ezekkel a műveletekkel nehezebb dolgunk lesz.

10.3. Bitforgató utasítások

Következő témánk a bitforgató utasításokat tekinti át. Ezek alapvetően két csoportra oszthatók: shiftelő és rotáló utasításokra. Az összes ilyen utasítás közös jellemzője, hogy céloperandusuk általános regiszter vagy memóriahivatkozás lehet, míg a második operandus a léptetés/forgatás számát adja meg bitekben. Ez az operandus vagy a közvetlen 1-es érték, vagy a CL regiszter lehet. (A TASM megenged 1-nél nagyobb közvetlen értéket is, ekkor a megfelelő utasítás annyiszor lesz lekódolva. Így pl. az SHL AX,3 hatására 3 db. SHL AX,1 utasítást fog az assembler generálni.) CL-nek minden bitjét figyelembe veszi a 8086-os processzor, így pl. akár 200-szor is léptethetünk. A legutoljára kicsorgó bit értékét minden esetben CF tartalmazza.

A shiftelés fogalmát már definiáltuk a gépi aritmetika elemzése közben (3. fejezet), ezért itt csak annyit említünk meg, hogy shiftelésből megkülönböztetünk előjeles (ez az ú.n. **aritmetikai**) és előjeltelen (ez a **logikai**) változatot, és mindkettőből van balra és jobbra irányuló utasítás is. Mindegyik shiftelő utasítás módosítja a CF, PF, SF, ZF és OF flag-eket, AF értéke meghatározatlan lesz. Ha a lépésszám 1-nél nagyobb, akkor OF értéke is meghatározatlan lesz.

Balra shiftelni az SHL (SHift logical Left) és az SAL (Shift Arithmetical Left) utasításokkal tudunk, közülük a második szolgál az előjeles léptetésre. Balra shiftelésnél azonban mindegy, hogy a céloperandus előjeles vagy előjeltelen érték-e, ezért, bár két mnemonik létezik, ténylegesen csak 1 utasítás van a gépi kód szintjén. Ezért ne csodálkozzunk, ha mondjuk a Turbo Debugger nem ismeri az SAL mnemonikot (de például a JC-t sem ismeri . . .).

Jobbra shiftelésnél már valóban meg kell különböztetni az előjeles változatot az előjel nélkülitől. Az SHR (SHift logical Right) előjel nélküli, míg az SAR (Shift Arithmetical Right) előjeles léptetést hajt végre a céloperanduson.

Nézzünk most egy egyszerű példát a shiftelés használatára. Tegyük fel, hogy AL-ben egy előjeles bájt van, amit szeretnénk megszorozni 6-tal, az eredményt pedig AX-ben várjuk. Ezt igazán sokféle módszerrel meg lehet oldani (LEA, IMUL, ADD, SUB stb.), de mi most léptetések segítségével tesszük meg. A megoldás majdnem triviális: a 6-tal való szorzás ugyanazt jelenti, mintha az eredeti szám dupláját és négyszeresét adnánk össze.

```
CBW
SHL     AX,1
MOV     BX,AX
SHL     AX,1
ADD     AX,BX
```

Ennél szebb megoldás is lehetséges, de a lényeg ezen is látszik. A program működése remélhetőleg mindenkinek világos.

Hasonló elgondolással megoldható az is, ha mondjuk az előjeles AL tartalmát meg akarjuk szorozni 3/2-del, az eredményt itt is AX-ben várva.

```
CBW
MOV     BX,AX
SAR     AX,1
ADD     AX,BX
```

Rotáláson (forgatáson) azt a, léptetéshez igen hasonló műveletet értjük, amikor az operandus bitjei szépen egymás után balra vagy jobbra lépnek, miközben a kicsorgó bit a túloldalon visszalép, „körbefordul”. Ez a fajta rotálás az operandus értékét „megőrzi”, legalábbis olyan értelemben, hogy ha mondjuk egy bájtot 8-szor balra vagy jobbra rotálunk, az eredeti változatlan bájtot kapjuk vissza. Rotálni nyilván csak előjeltelen értékeket lehet (tehát az előjelbitnek itt nincs speciális szerepe). Balra a ROL (ROtate Left), jobbra pedig a ROR (ROtate Right) mnemonikokkal lehet forgatni. Bármelyik forgató utasítás csak a CF és OF értékét módosítja. Ha 1-nél többször forgatunk CL-t használva, akkor OF tartalma meghatározatlan.

A forgatásnak van egy másik változata is, ami egy igen hasznos tulajdonsággal bír. A forgatást ugyanis úgy is el lehet végezni, hogy nem a kilépő bit fog belépni, hanem CF forgatás előtti értéke. Szemléletesen ez annyit jelent, mintha a CF bit az operandus egy plusz bitje lenne: balra forgatás esetén a legfelső utáni, míg jobbra forgatáskor a legalsó bit előtti bit szerepét tölti be. Ekkor azt mondjuk, hogy carry-n (CF-en) keresztül forgatunk. Ilyen módon balra az RCL (Rotate through Carry Left), jobbra az RCR (Rotate through Carry Right) utasítás forgat.

Forgatást például olyankor használunk, ha mondjuk szeretnénk az operandus minden egyes bitjét egyenként végigvizsgálni. A következő program például a BL-ben levő érték bináris alakját írja ki a képernyőre.

```
            MOV     AH,02h
            MOV     CX,8
@Ciklus:
            MOV     DL,'0'
            ROL     BL,1
            ADC     DL,00h
            INT     21h
            LOOP    @Ciklus
```

Az algoritmus működése azon alapul, hogy a ROL a kiforgó bitet CF-be is betölti, amit azután szépen hozzáadunk a „0” karakter kódjához. A forgatást nyolcszor elvégezve BL-ben újra a kiinduló érték lesz, s közben már a kiírás is helyesen megtörtént.

Szintén jó szolgálatot tehet valamelyik normál rotáló utasítás, ha egy regiszter alsó és felső bájtját, vagy alsó és felső bitnégyesét akarjuk felcserélni. Például az

```
ROL     AL,4
ROR     SI,8
```

utasítások hatására AL alsó és felső fele megcserélődik, majd SI alsó és felső bájtjával történik meg ugyanez.

Ha saját magunk akarunk megvalósítani nagy pontosságú aritmetikát, akkor is sokszor nyúlunk a shiftelő és rotáló utasításokhoz. Tegyük fel mondjuk, hogy a shiftelést ki akarjuk terjeszteni duplaszavakra is. A következő két sor a DX:AX-ben levő számot lépteti egyszer balra:

```
SHL     AX,1
RCL     DX,1
```

Az alapmódszer a következő: balra léptetésnél az első utasítás az SHL vagy SAL legyen, a többi pedig az RCL, és a legalsó bájttól haladunk a legfelső, legértékesebb bájt felé. Jobbra léptetés esetén a helyzet a fordítottjára változik: a legfelső bájttól haladunk lefelé, az első utasítás a szám típusától függően SHR vagy SAR legyen, a többi pedig RCR. Egyszerre csak 1 bittel tudjuk ilyen módon léptetni a számot, de így is legfeljebb 7 léptetésre lesz szükség, mivel a LepesSzam db. bittel való léptetést megoldhatjuk, ha a számot LepesSzam mod 8-szor léptetjük, majd az egész területet LepesSzam/8 bájttal magasabb (avagy alacsonyabb) címre másoljuk.

Utolsó példánkban az SI-ben levő számot előjelesen kiterjesztjük DI:SI-be. (A regiszterek ilyen „vad” választása természetesen szándékos.) Ezt megtehetnénk a hagyományos módon is, azaz a CWD utasítást használva, de ehhez SI-t AX-be kellene mozgatni, az eredményt pedig visszaírni a helyére. Így ráadásul AX és DX értéke is elromlana. Trükkösebb megoldás a következő:

```
PUSH    CX
MOV     DI,SI
MOV     CL,15
SAR     DI,CL
POP     CX
```

A program működése abban rejlik, hogy az SAR utasítás a DI (és így végső soron SI) előjelbitjével tölti fel DI-t, és az előjeles kiterjesztéshez pont ez kell. Igaz, így CL tartalma romlik el, de nyugodtan el lehet hinni, hogy a 8086-osnál újabb processzorokon ez jobban megtérül. Akit érdekel, miért, eláruljuk, hogy a 80186-os processzortól kezdve 1-től eltérő közvetlen értékkel (azaz konstanssal) is léptethetünk, nemcsak CL-t használva.

10.4. Bitmanipuláló utasítások

Utolsó témakörünk a bitmanipulációk (beállítás, törlés, negálás és tesztelés) megvalósításával foglalkozik.

Elsőként a logikai utasítások ilyen célú felhasználását nézzük meg. Ebbe a csoportba 5 olyan utasítás tartozik, amik az operandusok között ill. az operanduson valamilyen logikai műveletet hajtanak végre bitenként. 3 utasítás (AND, OR és XOR) kétoperandusú, a művelet eredménye minden esetben a céloperandusba kerül. A cél általános regiszter vagy memóriahivatkozás lehet, forrásként pedig ezeken kívül közvetlen értéket is írhatunk. A TEST szintén kétoperandusú, viszont az eredményt nem tárolja el. A NOT utasítás egyoperandusú, ez az operandus egyben

forrás és cél is, típusa szerint általános regiszter vagy memóriahivatkozás lehet. Az utasítások elnevezése utal az általuk végrehajtott logikai műveletre is, ezért bővebben nem tárgyaljuk őket, a gépi logika leírásában megtalálható a szükséges információ.

Az AND, OR, XOR és TEST utasítások a flag-eket egyformán kezelik: CF-et és OF-et 0-ra állítják, AF meghatározatlan lesz, PF, ZF és SF pedig módosulhatnak. A NOT utasítás nem változtat egyetlen flag-et sem.

A TEST különleges logikai utasítás, mivel a két operandus között bitenkénti logikai ÉS műveletet végez, a flag-eket ennek megfelelően beállítja, viszont a két forrást nem bántja, és a művelet eredményét is eldobja.

Egyszerű logikai azonosságok alkalmazása által ezekkel az utasításokkal képesek vagyunk különféle bitmanipulációk elvégzésére. Az alapprobléma a következő: adott egy bájt, amiben szeretnénk a 2-es bitet 0-ra állítani, a 4-es bitet 1-be billenteni, a 6-os bit értékét negálni, illetve kíváncsiak vagyunk, hogy az előjelbit (7-es bit) milyen állapotú. A bájtot tartalmazza most mondjuk az AL regiszter. Minden kérdés egyetlen utasítással megoldható:

```
AND     AL,0FBh
OR      AL,10h
XOR     AL,40h
TEST    AL,80h
```

A bit törléséhez a logikai ÉS művelet két jól ismert tulajdonságát használjuk fel: Bit AND 1 = Bit, illetve Bit AND 0 = 0. Ezért ha AL-t olyan értékkel (**bitmaszkkal**) hozzuk logikai ÉS kapcsolatba, amiben a törlendő bit(ek) helyén 0, a többi helyen pedig csupa 1-es áll, akkor készen is vagyunk.

Bitek beállításához a logikai VAGY műveletet és annak két tulajdonságát hívjuk segítségül: Bit OR 0 = Bit, valamint Bit OR 1 = 1. AL-t ezért olyan maszkkal kell VAGY kapcsolatba hozni, amiben a beállítandó bit(ek) 1-es, a többiek pedig 0-ás értéket tartalmaznak.

Ha egy bit állapotát kell negálni (más szóval **invertálni** vagy komplementálni), a logikai KIZÁRÓ VAGY művelet jöhet szóba. Ennek két tulajdonságát használjuk most ki: Bit XOR 0 = Bit, és Bit XOR 1 = NOT Bit. A használt maszk ezek alapján ugyanúgy fog felépülni, mint az előbbi esetben.

Ha az összes bitet negálni akarjuk, azt megtehetjük az XOR operandus,11. . . 1b utasítással, de ennél sokkal egyszerűbb a NOT operandus utasítás használata, ez ráadásul a flag-eket sem piszkálja meg. (Az 11. . . 1b jelölés egy csupa 1-esekből álló bináris számot jelent.)

Ha valamely bit vagy bitek állapotát kell megvizsgálnunk (más szóval **tesztelnünk**), a célravezető megoldás az lehet, hogy az adott operandust olyan maszkkal hozzuk ÉS kapcsolatba, ami a tesztelendő bitek helyén 1-es, a többi helyen 0-ás értékű. Ha a kapott eredmény nulla (ezt ZF = 1 is jelezni fogja), akkor a kérdéses bitek biztosan nem voltak beállítva. Különben két eset lehetséges: ha egy bitet vizsgáltunk, akkor az a bit tutira be volt állítva, ha pedig több bitre voltunk kíváncsiak, akkor a bitek közül valamennyi (de legalább egy) darab 1-es értékű volt az operandusban. Ha ez utóbbi dologra vagyunk csak kíváncsiak (tehát a bitek értéke külön-külön nem érdekel bennünket, csak az a fontos, hogy legalább egy be legyen állítva), akkor felesleges az AND utasítást használni. A TEST nem rontja el egyik operandusát sem, és ráadásul a flag-eket az ÉS művelet eredményének megfelelően beállítja.

A bitmanipuláció speciális esetét jelenti a Flags regiszter egyes bitjeinek, azaz a flag-eknek a beállítása, törlése stb. Az aritmetikai flag-eket (CF, PF, AF, ZF, SF, OF) elég sok utasítás képes megváltoztatni, ezek eredményét részben mi is irányíthatjuk megfelelő operandusok választásával. A CF, DF és IF flag értékét külön-külön állíthatjuk bizonyos utasításokkal: törlésre

a CLC, CLD és CLI (CLear Carry/Direction/Interrupt flag), míg beállításra az STC, STD és STI (SeT Carry/Direction/Interrupt flag) utasítások szolgálnak. Ezenkívül CF-et negálhatjuk a CMC (CoMplement Carry flag) utasítással. Mindegyik említett utasítás operandus nélküli, és más flag-ek értékét nem befolyásolják.

Ha egy olyan flag értékét akarjuk beállítani, amit egyéb módon nehézkes lenne (pl. AF), vagy egy olyan flag értékére vagyunk kíváncsiak, amit eddig ismert módon nem tudunk írni/olvasni (pl. TF), akkor más megoldást kell találni. Két lehetőség is van: vagy a PUSHF-POPF, vagy a LAHF-SAHF utasítások valamelyikét használjuk. Mindegyik utasítás operandus nélküli.

A PUSHF (PUSH Flags) utasítás a Flags regiszter teljes tartalmát (mind a 16 bitet) letárolja a verembe, egyébként működése a szokásos PUSH művelettel egyezik meg. A POPF (POP Flags) ezzel szemben a verem tetején levő szót leemeli ugyanúgy, mint a POP, majd a megfelelő biteken szereplő értékeket sorban betölti a flag-ekbe. (Ezt azért fogalmaztuk így, mivel a Flags néhány bitje kihasználatlan, így azokat nem módosíthatjuk.)

A másik két utasítás kicsit másképp dolgozik. A LAHF (Load AH from Flags) a Flags regiszter alsó bájtját az AH regiszterbe másolja, míg a SAHF (Store AH into Flags) AH 0, 2, 4, 6 és 7 számú bitjeit sorban a CF, PF, AF, ZF és SF flag-ekbe tölti be. Más flag-ek ill. a verem/veremmutató értékét nem módosítják.

A szemléltetés kedvéért most AF-et mind a kétféle módon negáljuk:

1.

```
PUSHF
POP     AX
XOR     AL,10h
PUSH    AX
POPF
```

2.

```
LAHF
XOR     AH,10h
SAHF
```

11. fejezet

Az Assembly nyelv kapcsolata magas szintű nyelvekkel, a paraméterátadás formái

Ebben a fejezetben azt vizsgáljuk, hogy egy önálló Assembly program esetén egy adott eljárásnak/függvénynek milyen módon adhatunk át paramétereket, függvények esetében hogyan kaphatjuk meg a függvény értékét, illetve hogyan tudunk lokális változókat kezelni, azaz hogyan valósíthatjuk meg mindazt, ami a magas szintű nyelvekben biztosítva van. Ezután megnézzük, hogy a Pascal és a C pontosan hogyan végzi ezt el, de előbb tisztázzunk valamit:

Ha a meghívott szubrutinnak (alprogramnak) nincs visszatérési értéke, akkor **eljárásnak** (procedure), egyébként pedig **függvénynek** (function) nevezzük. Ezt csak emlékeztetőül mondjuk, no meg azért is, mert Assemblyben formálisan nem tudunk különbséget tenni eljárás és függvény között, mivel mindkettőt a PROC-ENDP párral deklaráljuk, viszont ebben a deklarációban nyoma sincs sem paramétereknek, sem visszatérési értéknek, ellentétben pl. a Pascallal (PROCEDURE, FUNCTION), vagy C-vel, ahol végülis minden szubrutin függvény, de ha visszatérési értéke VOID, akkor eljárásnak minősül, s a fordítók is ekként kezelik őket.

11.1. Paraméterek átadása regisztereken keresztül

Ez azt jelenti, hogy az eljárásokat/függvényeket (a továbbiakban csak eljárásoknak nevezzük őket) úgy írjuk meg, hogy azok bizonyos regiszterekben kérik a paramétereket. Hogy pontosan mely regiszterekben, azt mi dönthetjük el tetszőlegesen (persze ésszerű keretek között, hiszen pl. a CS-t nem használjuk paraméterátadásra), de nem árt figyelembe vennünk azt sem, hogy hogy a legérdemesebb ezek megválasztása. A visszatérési értékeket (igen, mivel nem egy regiszter van, ezért több dolgot is visszaadhatunk) is ezeken keresztül adhatjuk vissza. Erre rögtön egy példa: van egy egyszerű eljárásunk, amely egy bájtokból álló vektor elemeit összegzi. Bemeneti paraméterei a vektor címe, hossza. Visszatérési értéke a vektor elemeinek előjeles összege. Az eljárás feltételezi, hogy az összeg elfér 16 biten.

Pelda5.ASM:

```
Osszegez        PROC
```

```
                PUSH    AX
                PUSHF
                CLD                             ;SI-nek növekednie kell
                                                ;majd
                XOR     BX,BX                   ;Kezdeti összeg = 0
@AddKov:
                LODSB                           ;Következő vektor-
                                                ;komponens AL-be
                CBW                             ;előjelesen kiterjesztjük
                                                ;szóvá
                ADD     BX,AX                   ;hozzáadjuk a meglevő
                                                ;összeghez
                LOOP    @AddKov                 ;Ismételd, amíg CX > 0
                POPF
                POP     AX
                RET
Osszegez        ENDP

.
.
.
                                                ;Példa az eljárás
                                                ;meghívására
                MOV     CX,13                   ;13 bájt hosszú a vektor
                MOV     SI,OFFSET Egyikvektor   ;cím offszetrésze
                                                ;feltesszük,hogy DS az
                                                ;adatszegmensre mutat
                CALL    Osszegez
                MOV     DX,BX                   ;Pl. a DX-be töltjük az
                                                ;eredményt
.
.
.
                                                ;valahol az
                                                ;adatszegmensben
                                                ;deklarálva vannak
                                                ;a változók:
Hossz           DW      ?
Cim             DD      ?
Eredmeny        DW      ?
Probavektor     DB      13 DUP (?)
```

Ezt az eljárást úgy terveztük, hogy a vektor címét a DS:SI regiszterpárban, a vektor hosszát a CX regiszterben kapja meg, ugyanis az eljárás szempontjából ez a legelőnyösebb (a LODSB és LOOP utasításokat majdnem minden előkészület után használhattuk). Az összeget az eljárás a BX regiszterben adja vissza.

Érdemes egy pillantást vetni a PUSHF-POPF párra: az irányjelzőt töröljük, mert nem tudhatjuk, hogy az eljárás hívásakor mire van állítva, de éppen e miatt fontos, hogy meg is őrizzük azt. Ezért mentettük el a flageket.

Az AX regiszter is csak átmeneti „változó”, jobb ha ennek értékét sem rontjuk el.

A példában egy olyan vektor komponenseit összegeztük, amely globális változóként az adatszegmensben helyezkedik el.

11.2. Paraméterek átadása globális változókon keresztül

Ez a módszer analóg az előzővel, annyi a különbség, hogy ilyenkor a paramétereket bizonyos globális változókba teszünk, s az eljárás majd onnan olvassa ki őket. Magas szintű nyelvekben is megtehetjük ezt, azaz nem paraméterezzük fel az eljárást, de az felhasznál bizonyos globális változókat bemenetként (persze ez nem túl szép megoldás). Ennek mintájára a visszatérési értékeket is átadhatjuk globális változókban. Írjuk át az előző példát:

Pelda6.ASM:

```
Osszegez        PROC
                PUSH    AX
                PUSH    BX
                PUSHF
                CLD                             ;SI-nek növekednie kell
                                                ;majd
                XOR     BX,BX                   ;Kezdeti összeg = 0
                MOV     CX,[Hossz]
                LDS     SI,[Cim]
@AddKov:
                LODSB                           ;Következő vektor-
                                                ;komponens AL-be
                CBW                             ;előjelesen kiterjesztjük
                                                ;szóvá
                ADD     AX,BX                   ;hozzáadjuk a meglevő
                                                ;összeghez
                LOOP    @AddKov                 ;Ismételd, amíg CX > 0
                MOV     [Eredmeny],BX
                POPF
                POP     BX
                POP     AX
                RET
Osszegez        ENDP
.
.
.
                                                ;Példa az eljárás
                                                ;meghívására
                MOV     [Hossz],13              ;13 bájt hosszú a vektor
                MOV     WORD PTR Cim[2],DS      ;ami az
                                                ;adatszegmensben
                                                ;van, így a cím
                                                ;szegmensrésze az DS
                MOV     WORD PTR [Cim],OFFSET Egyikvektor ;cím offszetrésze
                                                ;feltesszük,hogy DS az
```

```
                                                          ;adatszegmensre mutat
                                                          ;*
                CALL      Osszegez
                MOV       DX,[Eredmeny]                   ;Pl. a DX-be töltjük az
                                                          ;eredményt

.
.
.
                                                          ;valahol az
                                                          ;adatszegmensben
                                                          ;deklarálva vannak
                                                          ;a változók:
Hossz           DW        ?
Cim             DD        ?
Eredmeny        DW        ?
Egyikvektor     DB        13 DUP (?)
Masikvektor     DB        34 DUP (?)
```

A példához nem kívánunk magyarázatot fűzni, talán csak annyit, hogy mivel itt már a BX is egy átmeneti változóvá lett (mert nem rajta keresztül adjuk vissza az összeget), ezért jobb, ha ennek értékét megőrzi az eljárás.

Van egy dolog, amire azonban nagyon vigyázni kell az effajta paraméterátadással. Képzeljük el, hogy már beírtuk a paramétereket a változókba, de még mielőtt meghívnánk az eljárást (a „*”-gal jelzett helyen), bekövetkezik egy megszakítás. Tegyük fel, hogy a megszakításban egy saját megszakítási rutinunk hajtódik végre, ami szintén meghívja az Osszegez eljárást. Mi történik akkor, ha ő a Masikvektor komponenseit összegezteti? Ő is beírja a paramétereit ugyanezekbe a globális változókba, meghívja az eljárást, stb., végetér a megszakítási rutin, és a program ugyanott folytatódik tovább, ahol megszakadt. Esetünkben ez azt jelenti, hogy meghívjuk az eljárást, de nem azokkal a paraméterekkel, amiket beállítottunk, mert a megszakítási rutin felülírta azokat a sajátjaival. Ez természetesen a program hibás működését okozza. Az előző esetben, ahol regisztereken keresztül adtuk át a cuccokat, ott ez nem fordulhatott volna elő, hiszen egy megszakításnak kötelessége elmenteni azokat a regisztereket, amelyeket felhasznál. A probléma megoldása tehát az, hogy ha megszakításban hívjuk ezt az eljárást, akkor el kell mentenünk, majd vissza kell állítanunk a globális változók értékeit, mintha azok regiszterek lennének.

11.3. Paraméterek átadása a vermen keresztül

Ennek lényege, hogy az eljárás meghívása előtt a verembe beletesszük az összes paramétert, az eljárás pedig kiolvassa őket onnan, de nem a POP művelettel. Mivel a paraméterek sorrendje rögzített, és a utánuk a verembe csak a visszatérési cím kerül, így azok az eljáráson belül az SP-hez relatívan elérhetőek. Ezek szerint olyan címzésmódot kellene használni, amelyben SP közvetlenül szerepel, pl. MOV AX,[SP+6], csakhogy ilyen nem létezik. A másik probléma az SP-vel az lenne, hogy ha valamit ideiglenesen elmentünk a verembe, akkor a paraméterek relatív helye is megváltozik, így nehézkesebb lenne kezelni őket. Épp erre „találták ki” viszont a BP regisztert, emiatt van, hogy a vele használt címzésmódokban az alapértelmezett szegmens nem a DS, hanem az SS (de már a neve is: Base Pointer – Bázismutató, a vermen belül). A módszert ismét az előző példa átírásával mutatjuk be:

Pelda7.ASM:

```
Osszegez    PROC
            PUSH    BP
            MOV     BP,SP
            PUSH    AX
            PUSHF
            CLD                                 ;SI-nek növekednie kell
                                                ;majd
            XOR     BX,BX                       ;Kezdeti összeg = 0
            MOV     CX,[BP+8]
            LDS     SI,[BP+4]
@AddKov:
            LODSB                               ;Következő vektor-
                                                ;komponens AL-be
            CBW                                 ;előjelesen kiterjesztjük
                                                ;szóvá
            ADD     BX,AX                       ;hozzáadjuk a meglevő
                                                ;összeghez
            LOOP    @AddKov                     ;Ismételd, amíg CX > 0
            POPF
            POP     AX
            POP     BP
            RET     6
Osszegez    ENDP

.
.
.

                                                ;Példa az eljárás
                                                ;meghívására
            MOV     AX,13
            PUSH    AX                          ;13 bájt hosszú a vektor
            PUSH    DS
            MOV     AX,OFFSET Egyikvektor       ;cím offszetrésze
            PUSH    AX
            CALL    Osszegez
            MOV     DX,BX                       ;Pl. a DX-be töltjük az
                                                ;eredményt

.
.
.

                                                ;valahol az
                                                ;adatszegmensben
                                                ;deklarálva vannak
                                                ;a változók:
Hossz       DW      ?
Cim         DD      ?
Eredmeny    DW      ?
```

```
Probavektor     DB      13 DUP (?)
```

Az eljárásban BP értékét is megőriztük a biztonság kedvéért. A RET utasításban szereplő operandus azt mondja meg a processzornak, hogy olvassa ki a visszatérési címet, aztán az operandus értékét adja hozzá SP-hez. Ez azért kell, hogy az eljárás a paramétereit kitakarítsa a veremből. Hogy jobban megértsük a folyamatot, ábrákkal illusztráljuk a verem alakulását:

A BP-t tehát ráállítjuk az utolsó paraméterre (pontosabban itt most BP elmentett értékére), ez a bázis, s ehhez képest +4 bájtra található a vektor teljes címe, amit persze úgy helyeztünk a verembe, hogy az alacsonyabb címen szerepeljen az offszetrész, aztán a szegmens. BP-hez képest pedig +8 bájtra található a vektor hossza. Az eljárásból való visszatérés után persze minden eltűnik a veremből, amit előtte belepakoltunk a hívásához.

A visszatérési értéket most BX-ben adja vissza, erről még lesz szó.

A magas szintű nyelvek is vermen keresztüli paraméterátadást valósítanak meg. Ennek a módszernek kétféle változata is létezik, aszerint, hogy a paraméterek törlése a veremből kinek a feladata:

- Pascal-féle változat: a verem törlése az eljárás dolga, amint azt az előbb is láttuk
- C-féle változat: a verem törlése a hívó fél feladata, erre mindjárt nézünk példát

A Pascal a paramétereket olyan sorrendben teszi be a verembe, amilyen sorrendben megadtuk őket, tehát a legutolsó paraméter kerül bele a verembe utoljára. A C ezzel szemben fordított sorrendben teszi ezt, így a legelső paraméter kerül be utoljára. Ennek megvan az az előnye is, hogy könnyen megvalósítható a változó számú paraméterezés, hiszen ilyenkor az első paraméter relatív helye a bázishoz képest állandó, ugyanígy a másodiké is, stb., csak azt kell tudnia az eljárásnak, hogy hány paraméterrel hívták meg. Most pedig nézzük meg a példát, amikor a hívó törli a vermet:

Pelda8.ASM:

```
Osszegez        PROC
                PUSH    BP
                MOV     BP,SP
                PUSH    AX
                PUSHF
                CLD                     ;SI-nek növekednie kell
                                        ;majd
                XOR     BX,BX           ;Kezdeti összeg = 0
                MOV     CX,[BP+8]
                LDS     SI,[BP+4]
@AddKov:
                LODSB                   ;Következő vektor-
                                        ;komponens AL-be
                CBW                     ;előjelesen kiterjesztjük
                                        ;szóvá
                ADD     BX,AX           ;hozzáadjuk a meglevő
                                        ;összeghez
                LOOP    @AddKov         ;Ismételd, amíg CX > 0
                POPF
                POP     AX
                POP     BP
```

```
                RET
Osszegez        ENDP

.
.
.

                                                ;Példa az eljárás
                                                ;meghívására
                MOV     AX,13
                PUSH    AX                      ;13 bájt hosszú a vektor
                PUSH    DS                      ;cím szegmensrésze
                MOV     AX,OFFSET Egyikvektor   ;cím offszetrésze
                PUSH    AX
                CALL    Osszegez
                ADD     SP,6
                MOV     DX,BX                   ;Pl. a DX-be töltjük az
                                                ;eredményt
```

Látható, hogy ilyenkor a visszatérési értéket is át lehetne adni a vermen keresztül úgy, hogy pl. az egyik paraméter helyére beírjuk azt, s a hívó fél onnan olvassa ki, mielőtt törölné a vermet (a Pascal-szerű verzió esetén ez persze nem lehetséges). Ezt azonban nem szokás alkalmazni, a C is mindig regiszterekben adja ezt vissza, ez a könnyebben kezelhető megoldás, mi is tegyünk így.

11.4. Lokális változók megvalósítása

A lokális változókat kétféleképpen valósíthatjuk meg: az egyik, hogy elmentjük valamely regisztert, s felhasználjuk azt változóként, ugyanúgy, ahogy idáig is tettük az Osszegez eljárásban az AX-szel. A másik megoldás, amit a magas szintű nyelvek is alkalmaznak, hogy maga az eljárás kezdetben foglal a veremben a lokális változóknak helyet, ami annyit tesz, hogy SP értékét lecsökkenti valamennyivel. A változókat továbbra is a bázis-technikával (a BP-n keresztül) éri el. Írjuk át ismét az Osszegez-t úgy, hogy ezt a módszert alkalmazza (most a lokális változóban tároljuk ideiglenesen az összeget):

Pelda9.ASM:

```
Osszegez        PROC
                PUSH    BP
                MOV     BP,SP
                SUB     SP,2                    ;Helyet foglalunk a
                                                ;lokális változónak
                PUSH    AX
                PUSHF
                CLD                             ;SI-nek növekednie kell
                                                ;majd
                MOV     WORD PTR [BP-2],0h      ;Kezdeti összeg = 0
                MOV     CX,[BP+8]
                LDS     SI,[BP+4]
@AddKov:
```

```
                LODSB                            ;Következő vektor-
                                                 ;komponens AL-be
                CBW                              ;előjelesen kiterjesztjük
                                                 ;szóvá
                ADD     [BP-2],AX                ;hozzáadjuk a meglevő
                                                 ;összeghez
                LOOP    @AddKov                  ;Ismételd, amíg CX > 0
                POPF
                POP     AX
                MOV     BX,[BP-2]                ;BX = lok. változó
                ADD     SP,2                     ;lok. vált. helyének
                                                 ;felszabadítása
                POP     BP
                RET
Osszegez        ENDP

.
.
.

                                                 ;Példa az eljárás
                                                 ;meghívására
                MOV     AX,13
                PUSH    AX                       ;13 bájt hosszú a vektor
                PUSH    DS                       ;cím szegmensrésze
                MOV     AX,OFFSET Egyikvektor    ;cím offszetrésze
                PUSH    AX
                CALL    Osszegez
                ADD     SP,6
                MOV     DX,BX                    ;Pl. a DX-be töltjük az
                                                 ;eredményt
```

Hogyan is néz ki a verem az eljárás futásakor?

12. fejezet

Műveletek sztringekkel

Sztringen bájtok vagy szavak véges hosszú folyamát értjük. A magas szintű nyelvekben is találkozhatunk ezzel az adattípussal, de ott általában van valamilyen megkötés, mint pl. a rögzített maximális hossz, kötött elhelyezkedés a memóriában stb. Assemblyben sztringen nem csak a szegmensekben definiált karakteres sztring-konstansokat értjük, hanem a memória tetszőleges címén kezdődő összefüggő területet. Ez azt jelenti, hogy mondjuk egy 10 bájt méretű változót tekinthetünk 10 elemű tömbnek (vektornak), de kedvünk szerint akár sztringnek is; vagy például a verem tartalmát is kezelhetjük sztringként.

Mivel minden regiszter 16 bites, valamint a memória is szegmentált szervezésű, ezeknek természetes következménye, hogy a sztringek maximális hossza egy szegmensnyi, tehát 64 Kbájt. Persze ha ennél hosszabb folyamokra van szükség, akkor megfelelő feldarabolással ez is megoldható.

A 8086-os mikroprocesszor 5 utasítást kínál a sztringekkel végzett munka megkönnyítésére. Ezek közül kettővel már találkoztunk eddig is. Az utasítások közös jellemzője, hogy a forrássztring címét a DS:SI, míg a cél címét az ES:DI regiszterpárból olvassák ki. Ezek közül a DS szegmenst felülbírálhatjuk, minden más viszont rögzített. Az utasítások csak bájtos vagy szavas elemű sztringekkel tudnak dolgozni. Szintén közös tulajdonság, hogy a megfelelő művelet elvégzése után mindegyik utasítás a sztring(ek) következő elemére állítja SI-t és/vagy DI-t. A következő elemet most kétféleképpen értelmezhetjük: lehet a megszokott, növekvő irányú, illetve lehet fordított, csökkenő irányú is. A használni kívánt irányt a DF flag állása választja ki: DF = 0 esetén növekvő, míg DF = 1 esetén csökkenő lesz.

Ugyancsak jellemző mindegyik utasításra, hogy az Assembly szintjén több mnemonikkal és különféle operandusszámmal érhetők el. Pontosabban ez azt takarja, hogy mindegyiknek létezik 2, operandus nélküli rövid változata, illetve egy olyan változat, ahol „áloperandusokat” adhatunk meg. **Áloperandus** (pseudo operand) alatt most egy olyan operandust értünk, ami csak szimbolikus, és nem adja meg a tényleges operandus címét/helyét, csak annak méretét, szegmensét jelöli ki. Ezekre a már említett megkötések (DS:SI és ES:DI) miatt van szükség. Az áloperandussal rendelkező mnemonikok a CMPS, LODS, MOVS, SCAS és STOS, míg az egyszerű alakok a CMPSB, CMPSW, LODSB, LODSW, MOVSB, MOVSW, SCASB, SCASW, STOSB és STOSW. Minden rögtön világosabb lesz, ha nézünk rájuk egy példát:

Pelda10.ASM:

```
MODEL SMALL
```

```
.STACK

ADAT            SEGMENT
Szoveg1         DB      "Ez az első szöveg.",00h
Szoveg2         DB      "Ez a második szöveg.",00h
Kozos           DB      10 DUP (?)
Hossz           DW      ?
Vektor          DW      100 DUP (?)
ADAT            ENDS

KOD             SEGMENT
                ASSUME  CS:KOD,DS:ADAT

@Start:
                MOV     AX,ADAT
                MOV     DS,AX
                MOV     ES,AX
                LEA     SI,[Szoveg1]
                LEA     DI,[Szoveg2]
                CLD
                XOR     CX,CX
@Ciklus1:
                CMPSB
                JNE     @Vege
                INC     CX
                JMP     @Ciklus1
@Vege:
                LEA     SI,[SI-2]
                LEA     DI,[Kozos]
                ADD     DI,CX
                DEC     DI
                STD
        REP     MOVS    ES:[Kozos],DS:[SI]
                LEA     DI,[Szoveg1]
                MOV     CX,100
                XOR     AL,AL
                CLD
        REPNE   SCASB
                STC
                SBB     DI,OFFSET Szoveg1
                MOV     [Hossz],DI
                PUSH    CS
                POP     DS
                XOR     SI,SI
                LEA     DI,[Vektor]
                MOV     CX,100
@Ciklus2:
                LODSB
                CBW
                NOT     AX
                STOSW
```

```
                LOOP    @Ciklus2
                MOV     AX,4C00h
                INT     21h
KOD             ENDS
                END     @Start
```

A program nem sok értelmeset csinál, de szemléltetésnek megteszi.

Először meg akarjuk tudni, hogy a Szoveg1 és a Szoveg2 00h kódú karakterrel terminált sztringek elején hány darab megegyező karakter van. Ezt „hagyományos” módon úgy csinálnánk, hogy egy ciklusban kiolvasnánk az egyik sztring következő karakterét, azt összehasonlítanánk a másik sztring megfelelő karakterével, majd az eredmény függvényében döntenénk a továbbiakról. Számláláshoz CX-et használjuk. A ciklus most is marad csakúgy, mint a JNE-JMP páros. Az összehasonlítást viszont másképp végezzük el.

Első sztringkezelő utasításunk mnemonikja igen hasonlít az egész aritmetikás összehasonlításra. A CMPS (CoMPare String) mnemonik kétoperandusú utasítás. Mindkét operandus memóriahivatkozást kifejező áloperandus, és rendhagyó módon az első operandus a *forrássztring*, míg a második a *célsztring*. Az utasítás szintaxisa tehát a következő:

```
CMPS    forrássztring,célsztring
```

Az utasítás a forrássztringet összehasonlítja a célsztringgel, beállítja a flag-eket, de az operandusokat nem bántja. Egészen pontosan a célsztring egy elemét vonja ki a forrássztring egy eleméből, és a flag-eket ennek megfelelően módosítja. A forrássztring alapértelmezésben a DS:SI, a célsztring pedig az ES:DI címen található, mint már említettük. Az első operandus szegmensregisztere bármi lehet, a másodiké kötelezően ES. Ha egyik operandusból sem derül ki azok mérete (bájt vagy szó), akkor a méretet nekünk kell megadni a BYTE PTR vagy a WORD PTR kifejezés valamelyik operandus elé írásával. A PTR operátor az ő jobb oldalán álló kifejezés típusát a bal oldalán álló típusra változtatja meg, ezt más nyelvekben hasonló módon típusfelülbírálásnak (type casting/overloading) hívják. Ha nem kívánunk az operandusok megadásával bajlódni, akkor használhatunk két másik rövidítést. A CMPSB és CMPSW (CoMPare String Byte/Word) operandus nélküli mnemonikok rendre bájt ill. szó elemű sztringeket írnak elő a DS:SI és ES:DI címeken, tehát formálisan a CMPS BYTE/WORD PTR DS:[SI],ES:[DI] utasításnak felelnek meg.

A CMPSB utasítás tehát a sztringek következő karaktereit hasonlítja össze, majd SI-t és DI-t is megnöveli 1-gyel, hiszen DF = 0 volt. Ha ZF = 0, akkor vége a ciklusnak, különben növeljük CX-et, és visszaugrunk a ciklus elejére.

A @Vege címkére eljutva már mindenképpen megtaláltuk a közös rész hosszát, amit CX tartalmaz. SI és DI a két legutóbb összehasonlított karaktert követő bájtra mutat, így SI-t kettővel csökkentve az a közös rész utolsó karakterére fog mutatni.

Ezt követően a közös karakterláncot a Kozos változó területére fogjuk másolni. Mivel SI most a sztring végére mutat, célszerű, ha a másolást fordított irányban végezzük el. Ennek megfelelően állítjuk be DI értékét is. DF-et 1-re állítva készen állunk a másolásra.

Következő új sztringkezelő utasításunk a MOVS, MOVSB, MOVSW (MOVe String Byte/Word). Az utasítás teljes alakja a következő:

```
MOVS    cél,forrás
```

Az utasítás a forrás sztringet átmásolja a cél helyére, flag-et persze nem módosít. A példában

jól látszik, hogy a megadott operandusok tényleg nem valódi címre hivatkoznak: célként nem az ES:[Kozos] kifejezés tényleges címe lesz használva, hanem az ES:DI által mutatott érték, de forrásként is adhattunk volna meg bármit SI helyett. Ezek az operandusok csak a program dokumentálását, megértését segítik, belőlük az assembler csak a forrás szegmenst (DS) és a sztring elemméretét (ami most a Kozos elemmérete, tehát bájt) használja fel, a többit figyelmen kívül hagyja. A MOVSB, MOVSW mnemonikok a CMPS-hez hasonlóan formálisan a MOVS BYTE/WORD PTR ES:[DI],DS:[SI] utasítást rövidítik.

Ha egy adott sztring minden elemére akarunk egy konkrét sztringműveletet végrehajtani, akkor nem kell azt feltétlenül ciklusba rejtenünk. A sztringutasítást ismétlő prefixek minden sztringkezelő utasítás mnemonikja előtt megadhatók. Három fajtájuk van: REP; REPE, REPZ és REPNE, REPNZ (REPeat, REPeat while Equal/Zero/ZF = 1, REPeat while Not Equal/Not Zero/ZF = 0).

A REP prefix működése a következő:

1. ha CX = 0000h, akkor kilépés, az utasítás végrehajtása befejeződik
2. a prefixet követő sztringutasítás egyszeri végrehajtása
3. CX csökkentése 1-gyel, a flag-ek nem változnak
4. menjünk az (1)-re

Ha tehát kezdetben CX = 0000h, akkor nem történik egyetlen művelet sem, hatását tekintve gyakorlatilag egy NOP-nak fog megfelelni az utasítás. Különben a sztringkezelő utasítás 1-szer végrehajtódik, CX csökken, és ez mindaddig folytatódik, amíg CX zérus nem lesz. Ez végül is azt eredményezi, hogy a megadott utasítás pontosan annyiszor kerül végrehajtásra, amennyit CX kezdetben tartalmazott. REP prefixet a LODS, MOVS és STOS utasításokkal használhatunk.

A REPE, REPZ prefix a következő módon működik:

1. ha CX = 0000h, akkor kilépés, az utasítás végrehajtása befejeződik
2. a prefixet követő sztringutasítás egyszeri végrehajtása
3. CX csökkentése 1-gyel, a flag-ek nem változnak
4. ha ZF = 0, akkor kilépés, az utasítás végrehajtása befejeződik
5. menjünk az (1)-re

A REPE, REPZ tehát mindaddig ismétli a megadott utasítást, amíg CX ≠ 0000h és ZF = 1 is fennállnak. Ha valamelyik vagy mindkét feltétel hamis, az utasítás végrehajtása befejeződik.

A REPNE, REPNZ prefix hasonló módon akkor fejezi be az utasítás ismételt végrehajtását, ha a CX = 0000h, ZF = 1 feltételek közül legalább az egyik teljesül (tehát a fenti pszeudokódnak a (4) lépése változik meg).

A REPE, REPZ és REPNE, REPNZ prefixeket a CMPS és a SCAS utasításoknál illik használni (hiszen ez a két utasítás állítja a flag-eket is), bár a másik három utasítás esetében működésük a REP-nek felel meg.

Még egyszer hangsúlyozzuk, hogy ezek a prefixek kizárólag egyetlen utasítás ismételt végrehajtására használhatóak, s az az utasítás csak egy sztringkezelő mnemonik lehet.

Gépi kódú szinten csak két prefix létezik: REP, REPE, REPZ és REPNE, REPNZ, az első prefix tehát különböző módon működik az őt követő utasítástól függően.

A programhoz visszatérve, a REP MOVS ... utasítás hatására megtörténik a közös karakterlánc átmásolása. Mivel a REP prefix CX-től függ, nem volt véletlen, hogy mi is ebben tároltuk a másolandó rész hosszát. Ezt a sort persze írhattuk volna az egyszerűbb REP MOVSB alakban is, de be akartuk mutatni az áloperandusokat is.

Ha ez megvolt, akkor meg fogjuk mérni a Szoveg1 sztring hosszát, ami ekvivalens a 00h kódú karaktert megelőző bájtok számának meghatározásával. Pontosan ezt fogjuk tenni mi is: megkeressük, hol van a lezáró karakter, annak offszetjéből már meghatározható a sztring hossza.

A SCAS, SCASB, SCASW (SCAn String Byte/Word) utasítás teljes alakja a következő:

```
SCAS        célsztring
```

Az utasítás AL illetve AX tartalmát hasonlítja össze a cél ES:[DI] bájttal/szóval, beállítja a flag-eket, de egyik operandust sem bántja. Az összehasonlítást úgy kell érteni, hogy a célsztring elemét kivonja AL-ből vagy AX-ből, s az eredmény alapján módosítja a szükséges flag-eket. A SCASB, SCASW mnemonikok a SCAS BYTE/WORD PTR ES:[DI] utasítást rövidítik.

Esetünkben egészen addig akarjuk átvizsgálni a Szoveg1 bájtjait, amíg meg nem találjuk a 00h kódú karaktert. Mivel nem tudjuk, mennyi a sztring hossza, így azt sem tudjuk, hányszor fog lezajlani a ciklus. CX-et tehát olyan értékre kell beállítani, ami biztosan nagyobb vagy egyenlő a sztring hosszánál. Most feltesszük, hogy a sztring rövidebb 100 bájtnál. DF-et törölni kell, mivel a sztring elejétől növekvő sorrendben vizsgálódunk. A REPNE prefix hatására egészen mindaddig ismétlődik a SCAS, amíg ZF = 1 lesz, ami pedig azt jelenti, hogy a legutolsó vizsgált karakter a keresett bájt volt.

Ha az utasítás végzett, DI a 00h-s karakter utáni bájtra fog mutatni, ezért DI-t 1-gyel csökkentjük, majd levonjuk belőle a Szoveg1 kezdőoffszetjét. Ezzel DI-ben kialakult a hossz, amit rögvest el is tárolunk. A kivonást kicsit cselesen oldjuk meg. Az STC utasítás CF-et állítja 1-re, amit a célból a forrással együtt kivonunk az SBB-vel. Az OFFSET operátor nevéből adódóan a tőle jobbra álló szimbólum offszetcímét adja vissza.

A maradék programrész az előzőekhez képest semmi hasznosat nem csinál. A 100 db. szóból álló Vektor területet fogjuk feltölteni a következő módon: bármelyik szót úgy kapjuk meg, hogy a kódszegmens egy adott bájtját előjelesen kiterjesztjük szóvá, majd ennek vesszük az egyes komplemensét. Semmi értelme, de azért jó. :)

Mivel a kódszegmens lesz a forrás, CS-t berakjuk DS-be, SI-t a szegmens elejére állítjuk.

A ciklusban majdnem minden ismerős. A LODS, LODSB, LODSW és STOS, STOSB, STOSW utasításokkal már találkoztunk korábban, de azért ismétlésképpen felidézzük működésüket. Teljes alakjuk a következő:

```
LODS        forrás
STOS        cél
```

Forrás a szokott módon DS:SI, ebből DS felülbírálható, a cél pedig ES:DI, ami rögzített. Az utasítások AL-t vagy AX-et használják másik operandusként, a flag-eket nem módosítják. A LODS a forrást tölti be AL-be/AX-be, a STOS pedig AL-t/AX-et írja a cél területére. A LODS-nél a forrás, míg a STOS-nál AL/AX marad változatlan. SI-t ill. DI-t a szokott módon frissítik. A LODSB, LODSW, valamint a STOSB, STOSW utasítások formálisan sorban a LODS BYTE/WORD PTR DS:[SI], ill. a STOS BYTE/WORD PTR ES:[DI] utasításokat rövidítik.

Az AL-be beolvasott bájtot a CBW terjeszti ki előjelesen AX-be, majd ezt az egyoperandusú NOT utasítás bitenként logikailag negálja (azaz képezi az egyes komplemensét), végül a STOS utasítás a helyére teszi a kész szót. A ciklus lefutása után a program futása befejeződik.

Az utasítások teljes alakjának használatát csak a forrás szegmens megváltoztatása indokolhatná, minden más esetben a rövid alakok célravezetőbbek és jobban átláthatóak. Azonban az egyszerűbb változatok esetén is lehetőségünk van a forrás szegmens kiválasztására, mégpedig kétféle módon. Az egyik, hogy az utasítás előtt alkalmazzuk valamelyik SEGxx prefixet, a másik, hogy az utasítás előtti sorban kézzel berakjuk az adott prefix gépi kódját. Az első módszer MASM esetén nem működik (esetleg más assemblernél sem), a másodikban meg annyi a nehézség, hogy tudni kell a kívánt prefix gépi kódját. Ezek alapján a LODS SS:[SI] utasítást az alábbi két módon helyettesíthetjük:

1.

```
SEGSS     LODSB
```

2.

```
DB        36h
LODSB
```

Az SS: prefix gépi kódja 36h, a LODSB-é 0ACh, így mindkettő megoldás a 36h 0ACh bájtsorozatot generálja.

13. fejezet

Az .EXE és a .COM programok közötti különbségek, a PSP

Az, hogy a futtatható állományoknak milyen a formátumuk, az adott operációs rendszertől függ. A .COM és .EXE fájlok a DOS operációs rendszer állományai, csak ő képes ezeket futtatni.

13.1. A DOS memóriakezelése

Hamarosan megnézzük, hogy hogyan történik általában egy program elindítása, de hogy világosan lássunk, tudnunk kell egyet s mást a DOS-ról, azon belül is annak memóriakezeléséről. Ugyebár 1 Mbájt memóriát tudunk megcímezni (így a DOS is), viszont ennyit mégsem tudunk kihasználni, ugyanis az 1 Mbájt memória felső területén (azaz a magas címtartományban) helyezkedik el pl. a BIOS programkódja (ráadásul az ROM memória) és a képernyőmemória, tehát pl. ezeket nem tudjuk általános tárolásra használni. A memóriából mindenesetre összesen 384 Kbájtot tartanak fenn, így az 1024 Kbájtból csak 640 Kbájtot tud a DOS ténylegesen használni (ez a 640 ismerős szám kell, hogy legyen). Ezen a(z alsó) 640 Kbájton belül a DOS memóriablokkokat (memóriaszeleteket) kezel, amelyeknek négy fő jellemzőjük van:

- memóriablokk helye (kezdőcíme)
- memóriablokk mérete
- memóriablokk állapota (foglalt/szabad)
- memóriablokk (tulajdonosának) neve

Ezekről az adatokról a DOS nem vezet külön adminisztrációt egy saját, operációs rendszeri memóriaterületen, ahol pl. különböző listákban tárolná őket (amihez természetesen senkinek semmi köze), hanem egyszerűen az adott memóriablokk elején helyezi el őket. Ezek az adatok szintén egy blokkot alkotnak, amelynek mérete 16 bájt, neve pedig **memóriavezérlő blokk** (MCB – Memory Control Block). Ezzel megengedi a felhasználói programoknak is azt, hogy elérhessék ezeket az információkat, ami a DOS szemszögéből elég nagy hiba, ugyanis bárki „turkálhat” bennük, ami miatt tönkremehet a memória nyilvántartása. Ezalatt azt értjük, hogy adott esetben egy program a memóriablokkokra vonatkozó műveleteket nem a DOS szabványos

eljárásain keresztül végzi, hanem saját maga, mégpedig úgy, hogy egyszerűen átírja az MCB-t, és/vagy új(ak)at hoz létre, ha újabb memóriablokk(ok) keletkeztek. Egyébként három műveletet értelmez a DOS a memóriablokkjain, mindegyik eljárásnak vannak hibakódjaik (visszatérési értékeik):

Adott méretű memóriablokk lefoglalása (memória foglalása):

Ha sikerült lefoglalni a kívánt méretű memóriát, akkor visszakapjuk a lefoglalt terület címét. Hibakódok: nincs elég memória, ekkor visszakapjuk a legnagyobb szabad blokk méretét is.

Adott kezdőcímű blokk felszabadítása:

Hibakódok: a kezdőcím nem egy blokk kezdetére mutat.

Adott kezdőcímű foglalt blokk méretének megváltoztatása:

Hibakódok: nincs elég memória (ha növelni akarjuk), érvénytelen blokkcím.

Ha memóriát foglalunk le, akkor a DOS létrehoz egy új blokkot a kívánt mérettel, a blokk legelején az MCB-vel, s a blokk tényleges kezdőcíménél egy 16 bájttal nagyobb címet ad vissza mint a lefoglalt memória kezdetét. A felhasználói program (aki a memóriát kérte) szempontjából az MCB tehát nem tartozik a blokkhoz, gondoljunk csak a magas szintű programozási nyelvekre: ott is használunk ilyen függvényeket, s bennünket egyáltalán nem érdekel, hogy a lefoglalt memóriát „ki” és milyen módon tartja nyilván. A DOS tehát eleve egy 16 bájttal nagyobb blokkot foglal le, hogy az MCB-t is el tudja helyezni benne.

Lényeges, hogy soha egyetlen memóriablokk MCB-je se sérüljön meg, mindig helyes adatokat tartalmazzon, különben az említett memóriakezelő eljárások a következő hibakóddal térnek vissza: az MCB-k tönkrementek. Ilyenkor természetesen az adott művelet sem hajtódik végre. Erre példa: ha az adott blokk kezdőcíméhez hozzáadjuk annak méretét, akkor megkapjuk a következő blokk kezdőcímét (pontosabban annak MCB-jének kezdőcímét), és így tovább, azt kell kapnunk, hogy a legutolsó blokk utáni terület kezdőcíme a 640 Kbájt. Ha pl. ez nem teljesül, az azt jelenti, hogy valamelyik (de lehet, hogy több) MCB tönkrement, s attól kezdve talán már olyan területeket vizsgáltunk MCB-ként, amik valójában nem is azok. Mindesetre ilyenkor használhatatlanná válik a nyilvántartás. A DOS is az említett (saját) eljárásokat használja. Ha az előbbi hibával találkozik, akkor azt úgy kezeli le, hogy a „Memory Allocation Error – memóriakiosztási hiba” üzenettel szépen meghal.

Bootoláskor a DOS lefoglal magának egy blokkot (a memória elején), oda rakja adott esetben a saját programkódját, a többi memóriát pedig bejelöli egyetlen nagy szabad blokknak, amelyet aztán a programok szabadon használhatnak.

13.2. Általában egy programról

Tekintsük át *elméleti* szempontból, hogy általában hogyan épül fel egy program, mire van szüksége a futáshoz és futás közben, aztán megnézzük, hogy mindezeket hogyan biztosították DOS alatt.

Egy program három fő részből tevődik össze:

Kód:

Ez alatt magát az alacsony szintű programkódot értjük, a fordítóprogramok feladata, hogy a magas szintű nyelven leírt algoritmust a processzor számára feldolgozható kódra alakítsák át.

Adat:

Ebben a részben helyezkednek el a program *globális* változói, azaz a statikusan lefoglalt memória.

Verem:

Magas szintű nyelvekben nem ismeretes ez a fogalom (úgy értve, hogy nem a nyelv része), viszont nélkülözhetetlen az alacsony (gépi) szinten. A fordítók pl. ennek segítségével valósítják meg a *lokális* változókat (amiről már szó volt).

A gyakorlatban legtöbbször a programkódot nem egyetlen összefüggő egységként kezeljük (és ez az adatokra is vonatkozik), hanem „csoportosítjuk" a program részeit, s ennek alapján külön szegmensekbe (logikailag különálló részekbe) pakoljuk őket. Pontosan ezt csináljuk akkor, amikor az assemblernek szegmenseket definiálunk (ez tehát az a másfajta szegmensfogalom, amikor a szegmenseket logikailag különválasztható programrészeknek tekintjük). Meg lehet még említeni a Pascalt is: ő pl. a unitokat tekinti különálló programrészeknek, így minden unit programkódja külön kódszegmensbe kerül. Adatszegmens viszont csak egy globális van.

Nézzük meg, hogy mely részeket kell eltárolnunk a programállományban!

A programkódot mindenképpen, hiszen azt nem tudjuk a „semmiből" előállítani. A vermet viszont semmiképpen nem kell, hiszen a program indulásakor a verem üres (ezért mit is tárolnánk el?), ezért ezzel csak annyit kell csinálni, hogy a memóriában kijelölünk egy bizonyos nagyságú területet a verem számára, a veremmutatót a veremterület végére állítjuk (ott lesz a verem teteje). Szándékosan hagytuk a végére az adatokat, mert ennek elég csak valamely részét eltárolni. Elég csak azokat a globális változókat eltárolnunk, amelyeknek van valami kezdeti értéke a program indulásakor. Ezt úgy szokás megoldani, hogy a kezdőértékkel nem rendelkező változókat egy külön adatszegmensbe rakjuk, s előírjuk (ezt meg lehet mondani az assemblernek), hogy ez a szegmens ne kerüljön be az állományba (azaz a linker ne szerkessze be), viszont a program indításakor az operációs rendszer „tudjon" eme szegmensről, s biztosítson neki memóriát, azaz hozza létre azt. E megoldás azért kényelmetlen, mert ilyenkor két adatszegmensünk is van, de azt mi egyben szeretnénk látni, ezért a programnak futás közben „váltogatnia" kellene a 2 között. Kényelmesebb megoldás, ha egyetlen adatszegmenst deklarálunk, az egyik felében a kezdőértékkel rendelkező változókat, a másik felében pedig az azzal nem rendelkező változókat helyezzük el, s azt írjuk elő, hogy a szegmens másik fele ne kerüljön be az állományba (ez is megoldható, hiszen a szegmenseket szakaszokban is megadhatjuk, így megmondhatjuk azt is az assemblernek, hogy a szegmens adott darabja ne foglaljon feleslegesen helyet az állományban. Ez persze csak akkor teljesülhet, ha abban a szegmensben ténylegesen úgy deklaráljuk a változókat, hogy ne legyen kezdőértéke (pl. `Nev DB ?`), ha ez a követelmény valahol nem teljesül, akkor mindenképpen bekerül az egész szegmensdarab az állományba).

Egy program indításakor a programállományból tehát meg kell tudnia az operációs rendszernek (mivel ő indítja a programokat), hogy hol található(ak) az állományban a kódot tartalmazó részek. Ha ez megvan, lefoglalja nekik a megfelelő mennyiségű memóriá(ka)t (ott lesznek elhelyezve a program *kódszegmensei*), majd oda betölti a programkódot. Ezután következik az adat. Hasonlóan jár el a program adataszegmenseivel (az állományban le nem tárolt szegmenseket is beleértve, ezen memóriabeli szegmensekbe nyilván nem tölt be semmit az állományból, így az itt található változóknak meghatározatlan lesz a kezdőértéke), az adott szegmensek tartalmát szintén bemásolja. A memóriában meglesznek tehát a program *adatszegmensei* is. Azután foglal memóriát a veremnek is (*veremszegmens*). A veremmutatót ráállítja a veremszegmens végére, az adatszegmens-regisztert pedig valamely adatszegmens elejére (ezzel a program elő van készítve a futásra), majd átadja a vezérlést (ráugrik) a program első utasítására valamelyik kódszegmensben.

Mindez, amit leírtunk, csak elméleti dolog, azaz hogyan végzi el egy operációs rendszer általában egy program elindítását. Azonban, mint azt látni fogjuk, a DOS ilyen nagy fokú szabadságot nem enged meg számunkra.

13.3. Ahogy a DOS indítja a programokat

A DOS-nak kétféle futtatható állománya van (hűha!), a .COM és a .EXE formátumú. Amit ebben a fejezetben általánosságban leírunk, az mind a kettőre vonatkozik. A kettő közti részletes különbséget ezután tárgyaljuk.

Mint az látható, az, hogy a program mely szegmense tulajdonképpen mit tartalmaz, azaz hogy az kód- vagy adatszegmens-e, az teljesen lényegtelen. A DOS viszont ennél is továbbmegy: a programállományaiban letárolt szegmensekről semmit sem tud! Nem tudja, hogy hány szegmensről van szó, így pl. azt sem, hogy melyik hol kezdődik, mekkora, stb., egyetlen egy valamit tud róluk, mégpedig azt, hogy azok összmérete mennyi. Éppen ezért nem is foglal mindegyiknek külön-külön memóriát, hanem megnézi, hogy ezek mindösszesen mekkora memóriát igényelnek, s egy ekkora méretű memóriablokkot foglal le, ezen belül kap majd helyet minden szegmens. Annak, hogy a szükséges memóriát egyben foglalja le, annyi hátránya van, hogy ha nincs ekkora méretű szabad memóriablokk, akkor a programot nem tudja futtatni (ekkor kiköpi a „Program too big to fit in memory – a program túl nagy ahhoz, hogy a memóriába férjen" hibaüzenetet), holott lehet, hogy a szükséges memória „darabokban" rendelkezésre állna. Igazából azonban a gyakorlatban ez a probléma szinte egyáltalán nem jelentkezik, mivel ez csak a memória felaprózódása esetén jelenhet meg, azaz akkor, ha egyszerre több program is fut, aztán valamely(ek) végetér(nek), az általuk lefoglalt memóriablokk(ok) szabaddá válik/válnak, így a végén egy „lyukacsos" memóriakép keletkezik. Mivel a DOS csak egytaszkos oprendszer, azaz egyszerre csak egy program képes futni, mégpedig amelyiket a legutoljára indították, így nyilván az őt indító program nem tud azelőtt befejeződni, mielőtt ez befejeződne. Ebből az következik, hogy a szabad memória mindig egy blokkot alkot, a teljes memória foglalt részének mérete pedig a veremelv szerint változik: amennyit hozzávettünk a foglalt memória végéhez, azt fogjuk legelőször felszabadítani. Persze azért a DOS esetében is felaprózódhat a memória, ha egy program dinamikusan allokál magának memóriát, aztán bizonyos lefoglalt területeket tetszőleges sorrendben (és nem a lefoglalás fordított sorrendjében) szabadít fel. Ha ez a program ebben az állapotban saját maga kéri az DOS-t egy újabb program indítására, akkor már fennáll(hat) az előbbi probléma. Elkalandoztunk egy kicsit, ez már inkább az operációs rendszerek tárgy része, csak elmélkedésnek szántuk.

A DOS tehát egyetlenegy memóriablokkot foglal le minden szegmens számára, ezt az egész memóriablokkot nevezik **programszegmensnek** (Program Segment) (már megint egy újabb szegmensfogalom, de ez abból adódik, hogy a szegmens szó darabot, szeletet jelent, s ezt ugyebár sok mindenre lehet érteni). Fontos, hogy ezekkel a fogalmakkal tisztában legyünk.

Mint arról már szó volt, a programszegmens akkora méretű, amekkora a szegmensek által összesen igényelt memória. Nos, ez nem teljesen igaz, ugyanis a programszegmens *legalább* ekkora (+256 bájt, ld. később), ugyanis ha a programindításkor a DOS egy olyan szabad memóriablokkot talál, ami még nagyobb is a szükségesnél, akkor is „odaadja" az egészet a programnak. Hogy ez mire jó, azt nem tudjuk, mindenesetre így van. Ebből az a kellemetlenség adódik, hogy pl. ha a program a futás során dinamikusan szeretne memóriát allokálni, akkor azt a hibaüzenetet kaphatja, hogy nincs szabad memória. Valóban nincs, mert az összes szabad memória a programszegmenshez tartozhat, a programszegmens meg ugyebár nem más, mint egy foglalt memóriablokk, s a DOS memóriafoglaláskor nem talál más szabad blokkot. Ezért ilyenkor a

programnak induláskor le kell csökkentenie a programszegmensének méretét a minimálisra a már említett funkcióval!

Még egy fontos dolog, amit a DOS minden program indításakor elvégez: a programszegmens elején létrehozza a **programszegmens-prefixet** (Program Segment Prefix – PSP), ami egy 256 bájtból álló információs tömb. A +256 bájt tehát a PSP miatt kell. Mire kell a PSP és mit tartalmaz?

Sok mindent. Ezen 256 bájtos terület második fele pl. azt a sztringet tartalmazza, amit a promptnál a program neve után írunk paraméternek. Azt, hogy az első 128 bájt miket tárol, azt nagyon hosszú lenne itt felsorolni (inkább nézzük meg egy könyvben), számunkra nem is igazán fontos, csak egy-két érdekesebb dolog: az első két bájt az INT 20h kódja (0CDh 20h), vagy pl. a környezeti sztringek (amelyeket a DOS SET parancsával állíthatunk be) számára is le van foglalva egy memóriablokk, ennek a szegmenscíme (ami azt jelenti, hogy a cím offszetrésze nulla, így azt nem kell letárolni, csak a cím szegmensrészét, de ez igaz minden más memóriablokk címére is) benne van a PSP-ben. Így a program kiolvashatja ezt, s hozzáférhet a környezeti sztringekhez, módosíthatja azokat, stb. A DOS saját maga számára is tárolgat információt: pl. megtalálható az adott programot elindító program PSP-jének szegmenscíme is (**visszaláncolás** – backlink), ami általában a parancsértelmezőé (COMMAND.COM), hiszen legtöbbször onnan indítjuk a programokat. Mire kell ez? Arra, hogy ha a program befejeződik, akkor a DOS tudja, hogy „kinek” kell visszaadnia a vezérlést, és ez nyilván a programot betöltő program.

Ezek után nézzük meg a különbséget .COM és .EXE között.

13.4. .COM állományok

Ezek a fájlok voltak a DOS első futtatható állományai, nagyon egyszerű felépítésűek, ugyanis olyan programok számára tervezték, amelyek egyetlen szegmensből állnak. Erről ugyan nem volt szó, de egy szegmensnek természetesen lehet vegyes jellege is, tartalmazhat kódot is és adatot is egyszerre. Nos, .COM-ok esetében még a veremterület is ebben a szegmensben kell, hogy elhelyezkedjen. Egy programnak illene minimum 3 szegmensének lennie a három programrész számára, de itt szó sincs erről. Mint tudjuk, egy szegmens max. 64 Kbájt lehet, így egy .COM program is. Ráteszünk azonban még egy lapáttal: a PSP is a szegmens része, indításkor a DOS a PSP mögé másolja be a .COM programot! (Ezért van az, hogy .COM programok írásakor a szegmens elejére be kell raknunk azt a bizonyos ORG 100h direktívát, ezzel jelezve az assemblernek, hogy minden szegmensbeli offszetcímhez „adjon hozzá” 100h-t, azaz az offszeteket tolja el 256 bájttal, mivel a PSP nyilván nem tárolódik el a programállományban, de mindig „oda kell képzelnünk” a szegmens elejére). Induláskor a DOS a verembe berak egy 0000h-t, ami a PSP-ben levő INT 20h utasítás offszetcíme. A kilépés tehát megoldható egy RET utasítással is, mivel a DOS a programot egy közeli eljárásnak tekinti. (Ez csak .COM-ok esetén alkalmazható, mert ehhez a CS-nek a PSP-re kell mutatnia).

Ezután programszegmensen a programszegmens csak azon részét értjük, amelyet a program ténylegesen használ, nem vesszük bele tehát a felesleges részt, mivel annak semmi szerepe. A .COM programok programszegmense mindig 64 Kbájt, eltekintve attól az (egyébként nagyon ritka) esettől, amikor már 64 Kbájt szabad memória sincs a program számára, de ezzel most nem foglalkozunk. Miért van az, hogy a szegmens mindig kibővül 64 Kbájtra? Azért, hogy minél több hely legyen a verem számára. Ugyanis a .COM fájlokban a programról semmi kísérőinformáció nem található, a fájl csak magát az egyetlen szegmenst tárolja le, így nem tudjuk megadni a DOS-nak, hogy a szegmensen belül hol a veremterület, azaz hogy mi legyen

az SP kezdeti értéke. Emiatt a DOS úgy gondolkodik, hogy automatikusan maximális vermet biztosít, ezért növeli meg a szegmensméretet 64 Kbájtra, s annak végére állítja a veremmutatót. Ebből az is következik, hogy a program írásakor nekünk sem kell törődnünk a veremmel.
Ezek szerint azt sem tudjuk megadni, hogy a szegmensben hol található az első végrehajtandó utasítás. Így van, ennek mindig a .COM fájl elején kell lennie, azaz a memóriabeli szegmens 256. bájtján. Indításkor a DOS tehát a fenti ábra szerint állítja be a regisztereket.

13.5. Relokáció

Ha egy program több szegmenst is tartalmaz, akkor ahhoz, hogy el tudja érni azokat, meg kell határoznia minden egyes szegmens tényleges szegmenscímét. Azért nevezik ezt a folyamatot **relokációnak** (áthelyezésnek), mert a szegmenseknek csak a program elejéhez képesti *relatív* elhelyezkedését ismerhetjük (vehetjük úgy is, hogy a program legeleje a 0-n, azaz az 1 Mbájtos memória elején kezdődik, ekkor a szegmensek relatív címei egyben *abszolút* címek is, a program futóképes lenne), azonban a program a memória szinte tetszőleges részére betöltődhet, s hogy pontosan hova, az csak indításkor derül ki. A program csak akkor lesz futóképes, ha a relatív szegmenscímeket átírjuk, azaz azokhoz hozzáadjuk a program elejének szegmenscímét. Ezzel a programot mintegy áthelyeztük az adott memóriaterületre. Lássuk ezt egy példán keresztül:

```
Kod       SEGMENT  PARA             ;paragrafushatáron kez-
                                    ;dődjön a szegmens
          ASSUME   CS:Kod,DS:Adat   ;mondjuk meg az assem-
                                    ;blernek, hogy mely szeg-
                                    ;mensregiszter mely
                                    ;szegmensre mutat,
                                    ;ő eszerint generálja
                                    ;majd az
                                    ;offszetcímeket

          MOV      AX,Adat          ;Az „Adat” szegmens
                                    ;szegmenscíme
          MOV      DS,AX            ;A feltételezésnek
                                    ;tegyünk is eleget,
                                    ;ne vágjuk át szegény
                                    ;assembler bácsit

.
.
.

          MOV      AX,4C00h
          INT      21h
Kod       ENDS

Adat      SEGMENT  PARA             ;paragrafushatáron kez-
                                    ;dődjön a szegmens
Valami    DB       2
Akarmi    DB       14
Adat      ENDS
```

```
        END
```

Ez egy szokványos program, de a lényeg most azon van, hogy amikor az elején a DS-t ráállítjuk az Adat szegmensre, akkor a MOV AX,Adat utasítással mit kezd az assembler, hiszen az Adat értéke (ami egy szegmenscím) attól függ, hogy a program a memóriába hová töltődik majd be. A válasz az, hogy az Adat egy relatív érték, a program első szegmenséhez képest. Hogy melyik lesz az első szegmens, az csak a linkelésnél válik el, amikor a linker az egyes szegmensdarabokat összefűzi egyetlen szegmenssé, valamint a szegmenseket egyetlen programmá, de hogy a szegmensek milyen sorrendben kövessék egymást a programban, arról ő dönt. Az assembler tehát csak bejelöli az .OBJ fájlban, hogy ezen a helyen egy relatív szegmenscím kelletik, ezt a linker észreveszi, s beírja oda azt. A fenti programnak csak két szegmense van, tegyük fel, hogy a Kod lesz az első, s hogy a Kod szegmens mérete pl. 36 bájt (mivel ez nem lenne 16-tal osztható, ezért a méret linkeléskor 16-tal oszthatóra nő, hogy az utána következő szegmens paragrafushatárra essen, így most 48 bájt lesz). Ekkor a lefordított program a következő lesz:

```
        MOV     AX,0003h    ;48/16=3
        MOV     DS,AX

.
.
.

        MOV     AX,4C00h
        INT     21h         ;Ezután szükség szerint
                            ;„szemét” következik,
                            ;hogy a „DB 2”
                            ;a program
.                           ;elejéhez képest 16-tal
.                           ;osztható címre essen,
.                           ; hiszen azt a szegmenst
                            ;paragrafushatáron
                            ;kezdtük

        DB      2
        DB      14
```

Most a program egy relatív szegmenscímet tartalmaz (0003), de még mielőtt elindíthatnánk, azt át kell javítani a tényleges címre. Ha pl. a Kod szegmens a memória 5678h:0000h területére kerül majd, akkor a 0003h-hoz hozzá kell adni az 5678h-t, hogy helyes értéket kapjunk.

Mivel ezt a program indításakor kell elvégezni, így ez a feladat a DOS-ra hárul, de ő ezt csak az .EXE fájlok esetében képes elvégezni, mivel ott el van tárolva a relokációhoz szükséges információ. Ebből világos, hogy egy .COM program nem tartalmazhat ilyen módon relatív szegmenscímeket, igy pl. a fenti programot sem lehet .COM-má fordítani, a linker is idegeskedik, ha ilyennel találkozik. A másik megoldás az, ha a program maga végzi el induláskor a szükséges relokációt. Ezzel elérhetjük azt is, hogy egy .COM program több szegmenst is tartalmazzon (amelyek összmérete nem haladhatja meg persze a 64K-t), csak a relokációt el kell végeznünk. A fenti példát átírva:

```
DOSSEG

Kod             SEGMENT  PARA
                ASSUME   CS:Kod,DS:Adat
                ORG      100h              ;PSP helye

@Start:
                MOV      AX,Kod:Adat_eleje ;a „Kod” szegmens elejé-
                                           ;hez képest az
                                           ;„Adat_eleje” címke,
                                           ;azaz az adat szegmens
                                           ;eleje hány bájtra helyez-
                                           ;kedik el
                MOV      CL,4h
                SHR      AX,CL             ;osztva 16-tal (az előző
                                           ;példában közvetlenül ez
                                           ;az érték volt benne az
                                           ;utasításban)
                MOV      BX,DS             ;DS ekkor még a PSP
                                           ;elejére, azaz a „Kod”
                                           ;szegmens elejére
                                           ;mutat
                ADD      AX,BX
                MOV      DS,AX             ;ráállítjuk az adatszeg-
                                           ;mensre
.                                          ;innen DS már az adat
.                                          ;szegmensre mutat
.
                MOV      AX,4C00h
                INT      21h
Kod             ENDS

Adat            SEGMENT  PARA
                ORG      0h                ;Ezen a szegmensen belül
                                           ;nincs eltolás
Adat_eleje      LABEL

Valami          DB       2
Akarmi          DB       14
Adat            ENDS
                END      @Start
```

A program elején levő DOSSEG direktívával biztosítjuk azt, hogy a szegmensek a deklarálás sorrendjében kövessék egymást az összeszerkesztett programban is, hiszen itt fontos, hogy a Kod legyen az első szegmens.

13.6. .EXE állományok

Az .EXE állományokat a bonyolultabb programok számára találták ki, amelyek tetszőleges számú szegmensből állnak. Magában az állományban ezek egymást követően, folyamatosan vannak eltárolva, ahogyan a linker egymás után fűzte őket. Fontos azonban megjegyezni, hogy

nem biztos, hogy az egész állomány magát a programot tárolja, mivel az állományban a program után még tetszőleges hosszúságban tárolhatunk pl. adatokat, mint egy közönséges adatfájlban. Ezek az adatok nem tartoznak a programhoz, még egyszer kihangsúlyozzuk. Az .EXE fájlok elején mindig található egy **fejléc** (header), azaz egy információs tömb, amely a DOS számára nélkülözhetetlen a program elindításához.

A betöltési folyamat hasonló a .COM-okéhoz: a fejlécből a DOS megnézi, hogy a fejléc után eltárolt program milyen hosszú, azaz hol a határ a program és az előbb említett adatok között. Itt persze már nem él az a megkötés, hogy a program (azaz a szegmensek összmérete) nem haladhatja meg a 64 Kbájtot. Ezután szintén a fejlécből megnézi, hogy a programnak mennyi többletmemóriára van szüksége (ez a bejegyzés a „minimálisan szükséges többletmemória"). Ez biztositja a programban el nem tárolt szegmensek létrehozását a memóriában, amiről már beszéltünk. A programszegmens így legalább az összméret + többletmemória méretű lesz. Van azonban egy „maximálisan szükséges többletmemória"-bejegyzés is, elméletileg ennél nagyobb soha nem lehet nagyobb a programszegmens, ezzel tehát el lehetne kerülni, hogy az feleslegesen nagy legyen, de a gyakorlatban egyáltalán nem élnek ezzel a lehetőséggel, szinte mindig maximumra állitják (a linkerek, merthogy ők hozzák létre a futtatható állományokat). A DOS tehát lefoglalja a programszegmenst, bemásolja az .EXE-ből a programot, s elvégzi a program relokációját. Hogy a programban mely helyeken kell átírni a relatív szegmenscímeket, azt a **relokációs táblából** (relocation table) (ami szintén a fejléc része) tudja az oprendszer. Mivel programunk több szegmensből áll, kijelölhetjük, hogy melyik a veremszegmens (szintén relatívan), indításkor a DOS az SS:SP-t ennek megfelelően állítja be. Ugyanígy relatívan adhatjuk meg, hogy hol van a program **belépési pontja** (entry point), azaz mi lesz a CS:IP kezdeti értéke. Fontos megjegyezni, hogy .EXE-k esetében a PSP nem tartozik semelyik szegmenshez sem, ezért nincs szükség ilyen programok írásakor sehol sem az ORG 100h direktíva megadására. Kezdeti adatszegmenst nem tudunk kijelölni a program számára, a DOS úgy van vele, hogy ha egy programnak több szegmense is van, akkor azok között valószínűleg egynél több adatszegmens is van, így a programnak futás közben úgyis állítgatnia kell a DS-t, akkor miért ne tegye ezt meg pl. már rögtön induláskor, ahogy az első példában már láttuk? Ez amúgy sem okoz a programnak gondot, a relokációt elvégzik helyette, nem neki kell vele szenvednie. Éppen ezért kezdetben a DS, ES a PSP elejére mutat.

14. fejezet

Szoftver-megszakítások

Mint említettük, szoftver-megszakításnak egy program által kiváltott megszakítást nevezünk. Hogy miért van erre szükség? Nos, néhány szoftver-megszakítás jócskán megkönnyíti a különféle hardvereszközök elérését, míg mások az operációs rendszer egyes funkcióival teremtenek kapcsolatot, de számos egyéb felhasználásuk is lehetséges.

Összesen 256 db. szoftver-megszakítást kezel a 8086-os mikroprocesszor. Ebből 8 (vagy 16) db. az IRQ-k kezelésére van fenntartva, néhány pedig speciális célt szolgál, de a többi mind szabadon rendelkezésünkre áll. Azt azért tudni kell, hogy a gép bekapcsolása, ill. az operációs rendszer felállása után néhány megszakítás már foglalt lesz, azokon keresztül használhatjuk a munkánkat segítő szolgáltatásokat.

Megszakítást az egyoperandusú INT utasítással kezdeményezhetünk. Az operandus kizárólag egy bájt méretű közvetlen adat lehet, ami a kért szoftver-megszakítást azonosítja. Az utasítás pontos működésével a következő fejezetben foglalkozunk majd. Érdemes megemlíteni még egy utasítást. Az INTO (INTerrupt on Overflow) egy 04h-s megszakítást kér, ha OF = 1, különben pedig működése megfelel egy NOP-nak (azaz nem csinál semmit sem, csak IP-t növeli). Ennek akkor vehetjük hasznát, ha mondjuk az előjeles számokkal végzett műveletek során keletkező túlcsordulás számunkra nemkívánatos, és annak bekövetkeztekor szeretnénk lekezelni a szituációt.

Egy adott számú megszakítás sokszor sok-sok szolgáltatás kapuját jelenti. A kívánt szolgáltatást és annak paramétereit ilyenkor különböző regiszterekben kell közölni, és a visszatérő értékeket is általában valamilyen regiszterben kapjuk meg. Hogy melyik megszakítás mely szolgáltatása hányas számú és az milyen regiszterekben várja az adatokat, lehetetlen fejben tartani. Ezeknek sok szakkönyvben is utánanézhetünk. Ha viszont a könyv nincs a közelben, akkor sem kell csüggedni. Számos ingyenes segédprogramot kifejezetten az ilyen gondok megoldására készítettek el. A legismertebbek: HelpPC, Tech Help!, Norton Guide, Ralf Brown's Interrupt List. A programok mindegyike hozzáférhető, és tudtunkkal magáncélra ingyenesen használható. Mindegyik tulajdonképpen egy hatalmas adatbázison alapul, amiben hardveres és szoftveres témákat is találhatunk kedvünk szerint.

A 14.1. táblázatban felsorolunk néhány gyakoribb, fontos megszakítást a teljesség igénye nélkül.

A következőkben néhány példán keresztül bemutatjuk a fontosabb szolgáltatások használatát.

14.1. táblázat. Gyakran használt szoftver-megszakítások

10h	Képernyővel kapcsolatos szolgáltatások (Video services)
13h	Lemezműveletek (Disk operation)
14h	Soros ki-/bemenet (Serial I/O)
16h	Billentyűzet (Keyboard)
1Ch	Időzítő (Timer tick)
20h	Program befejezés (Program terminate – DOS)
21h	DOS szolgáltatások (DOS services)
25h	Közvetlen lemezolvasás (Absolute disk read – DOS)
26h	Közvetlen lemezírás (Absolute disk write – DOS)
27h	Rezidenssé tétel (Terminate and stay resident – DOS)
2Fh	Vegyes szolgáltatások (Multiplex)
33h	Egér támogatás (Mouse support)

14.1. Szövegkiíratás, billentyűzet-kezelés

Legyen a feladat a következő: a program írja ki a képernyőre a „Tetszik az Assembly?” szöveget, majd várjon egy billentyű lenyomására. Ha az „i” vagy „I” gombot nyomják le, akkor írja ki az „Igen.” választ, „n” és „N” hatására pedig a „Nem.” szöveget. Mindkét esetben ezután fejezze be működését. Egyéb billentyűre ne reagáljon, hanem várakozzon valamelyik helyes válaszra. Lássuk a programot:

Pelda11.ASM:

```
MODEL SMALL

.STACK

ADAT        SEGMENT
Kerdes      DB        "Tetszik az Assembly? $"
Igen        DB        "Igen.",0Dh,0Ah,'$'
Nem         DB        "Nem.",0Dh,0Ah,'$'
ADAT        ENDS

KOD         SEGMENT
            ASSUME    CS:KOD,DS:ADAT

@Start:
            MOV       AX,ADAT
            MOV       DS,AX
            MOV       AH,09h
            LEA       DX,[Kerdes]
            INT       21h
@Ciklus:
            XOR       AH,AH
            INT       16h
            CMP       AL,'i'
            JE        @Igen
            CMP       AL,'I'
```

```
                JE      @Igen
                CMP     AL,'n'
                JE      @Nem
                CMP     AL,'N'
                JNE     @Ciklus
@Nem:
                LEA     DX,[Nem]
                JMP     @Vege
@Igen:
                LEA     DX,[Igen]
@Vege:
                MOV     AH,09h
                INT     21h
                MOV     AX,4C00h
                INT     21h
KOD             ENDS
                END     @Start
```

A program elég rövid, és kevés újdonságot tartalmaz, így megértése nem lesz nehéz.

Szöveget sokféleképpen ki lehet írni a képernyőre. Mi most az egyik DOS-funkciót fogjuk használni erre. Már láttuk, hogy a DOS-t az INT 21h-n keresztül lehet segítségül hívni, a kért szolgáltatás számát AH-ban kell megadnunk. A 4Ch sorszámú funkciót már eddig is használtuk a programból való kilépésre. A 09h szolgáltatás egy dollárjellel („$") lezárt sztringet ír ki az aktuális kurzorpozícióba. A szöveg offszetcímét DX-ben kell megadni, szegmensként DS-t használja.

A billentyűzet kezelését az INT 16h-n keresztül tehetjük meg. A szolgáltatás számát itt is AH-ba kell rakni. Ennek 00h-s funkciója egészen addig várakozik, míg le nem nyomunk egy gombot a billentyűzeten, majd a gomb ASCII-kódját visszaadja AL-ben. Hátránya, hogy néhány billentyű lenyomását nem tudjuk így megfigyelni (pl. Ctrl, Shift), míg más billentyűk AL-ben 00h-t adnak vissza, s közben AH tartalmazza a billentyűt azonosító számot (ez az ú.n. **scan code**). Most nekünk csak a kicsi és nagy „I" ill. „N" betűkre kell figyelnünk, így nincs más dolgunk, mint a megszakításból visszatérés után AL-t megvizsgálnunk. Ha AL-ben 'i' vagy 'I' van, akkor a @Igen címkére megyünk. Ha AL = 'n' vagy AL = 'N', akkor a @Nem címkét választjuk célként. Különben visszamegyünk a @Ciklus címkére, és várjuk a következő lenyomandó billentyűt. Figyeljük meg, hogy a szelekció utolsó feltételében akkor megyünk vissza a ciklusba, ha AL ≠ 'N', máskülönben „rácsorgunk" a @Nem címkére, ahová amúgy is mennünk kéne. Ezzel a módszerrel megspóroltunk egy JMP-t.

Akár a @Igen, akár a @Nem címkét választottuk, a szöveg offszetjének DX-be betöltése után a @Vege címkénél kötünk ki, ahol kiírjuk a választ, majd kilépünk a programból.

14.2. Szöveges képernyő kezelése, számok hexadecimális alakban kiírása

Következő problémánk már rafináltabb: a program indításkor törölje le a képernyőt, majd a bal felső sarokba folyamatosan írja ki egy szó méretű változó tartalmát hexadecimálisan, miközben figyeli, volt-e lenyomva billentyű. A változó értékét minden kiírást követően eggyel növelje meg, kezdetben pedig a 0000h-ról induljon. Ha volt lenyomva billentyű, akkor annak

ASCII kódja szerinti karaktert írja ki a második sorba. A szóköz (space) megnyomására lépjen ki.

Pelda12.ASM:

```
MODEL SMALL

.STACK

ADAT        SEGMENT
Szamlalo    DW        0000h
HexaJegy    DB        "0123456789ABCDEF"
ADAT        ENDS

KOD         SEGMENT
            ASSUME    CS:KOD,DS:ADAT

HexKiir     PROC
            PUSH      AX BX CX DX
            LEA       BX,[HexaJegy]
            MOV       CL,4
            MOV       DL,AL
            SHR       AL,CL
            XLAT
            XCHG      AL,DL
            AND       AL,0Fh
            XLAT
            MOV       DH,AL
            PUSH      DX
            XCHG      AL,AH
            MOV       DL,AL
            SHR       AL,CL
            XLAT
            XCHG      AL,DL
            AND       AL,0Fh
            XLAT
            MOV       DH,AL
            MOV       AH,02h
            INT       21h
            MOV       DL,DH
            INT       21h
            POP       DX
            INT       21h
            MOV       DL,DH
            INT       21h
            POP       DX CX BX AX
            RET
HexKiir     ENDP

Torol       PROC
            PUSH      AX BX CX DX
            MOV       AX,0600h
```

```
                MOV     BH,07h
                XOR     CX,CX
                MOV     DX,184Fh
                INT     10h
                POP     DX CX BX AX
                RET
Torol           ENDP

GotoXY          PROC
                PUSH    AX BX
                MOV     AH,02h
                XOR     BH,BH
                INT     10h
                POP     BX AX
                RET
GotoXY          ENDP

@Start:
                MOV     AX,ADAT
                MOV     DS,AX
                CALL    Torol
@Ciklus:
                XOR     DX,DX
                CALL    GotoXY
                MOV     AX,[Szamlalo]
                CALL    HexKiir
                INC     AX
                MOV     [Szamlalo],AX
                MOV     AH,01h
                INT     16h
                JZ      @Ciklus
                CMP     AL,20h
                JE      @Vege
                INC     DH
                CALL    GotoXY
                MOV     AH,02h
                MOV     DL,AL
                INT     21h
                XOR     AH,AH
                INT     16h
                JMP     @Ciklus
@Vege:
                XOR     AH,AH
                INT     16h
                MOV     AX,4C00h
                INT     21h
KOD             ENDS
                END     @Start
```

Az adatszegmensben nincs semmi ismeretlen, bár a HexaJegy tartalma kicsit furcsának tűnhet. A homályt rövidesen el fogja oszlatni, ha megnézzük, hogyan írjuk ki a számot.

A HexKiir az AX-ben levő előjeltelen számot írja ki négy karakteres hexadecimális alakban

(tehát a vezető nullákkal együtt). Az eljárásban a szó kiírását visszavezetjük két bájt hexadecimális kiírására, azt pedig az alsó és felső bitnégyesek számjeggyé alakítására. Mivel a képernyőre először a felső bájt értékét kell kiírni, az alsó bájtot dolgozzuk fel, amit aztán berakunk a verembe, majd a felső bájt hasonló átalakításai után azt kiírjuk a képernyőre, végül a veremből kivéve az alsó bájt hexadecimális alakja is megjelenik a helyén. DX-ben fogjuk tárolni a már elkészült számjegyeket. BX és CL szerepére hamarosan fény derül.

Először a felső bitnégyest kell számjeggyé alakítanunk. Ezért AL-t elrakjuk DL-be, majd AL felveszi a felső bitnégyes értékét (ami 00h és 0Fh között lesz). Ezt az SHR AL,CL utasítással érjük el. Most CL = 4 (erre állítottuk be), s így AL felső négy bitje lekerül az alsó négy helyére, miközben a felső bitnégyes kinullázódik (ez a shiftelés tulajdonsága). Ez pedig azt fogja jelenteni, hogy AL-ben ugyanakkora szám lesz, mint amekkora eredetileg a felső bitnégyesében volt.

Az operandus nélküli XLAT (transLATe byte; X – trans) utasítás az AL-ben levő bájtot kicseréli a DS:BX című fordítótáblázat AL-edik elemére (nullától számozva az elemeket), tehát szimbolikusan megfelel a MOV AL,[BX+AL] utasításnak (ami ugyebár így illegális), de AL-t előjeltelenül értelmezi. Egyetlen flag tartalmát sem módosítja. Az alapértelmezett DS szegmens felülbírálható prefixszel. Esetünkben ez most azt jelenti, hogy az AL-ben levő számot az azt szimbolizáló hexadecimális számjegyre cseréli le, hiszen BX a HexaJegy offszetcímét tartalmazza. Így már világos a HexaJegy definíciója. Az XLAT utasítás egyébként kaphat egy „áloperandust" is, ennek szerepe hasonló a sztringkezelő utasítások operandusaihoz.

Érdekesség, hogy az XLAT utasítás az XLATB mnemonik alatt is elérhető, tehát mindkettő ugyanazt jelenti a gépi kód szintjén. Hogy miért kell egy ilyen egyértelmű dolgot végző utasításnak 2 mnemonik, azon lehetne filozofálni. Az SHL-SAL párosra még rá lehet fogni, hogy segítik a program dokumentálását (t.i. az operandus típusát jelzi a mnemonik), viszont a mostani esetben érthetetlen a dupla mnemonik létezése.

Most, hogy megvan az első jegy, AL eredeti alsó bitnégyesét is át kell alakítani számjeggyé. AL-t DL-ben tároltuk el, és az elején azt mondtuk, hogy a kész számjegyeket DX-ben fogjuk gyűjteni. Most tehát célszerű lenne, ha AL és DL tartalmát fel tudnánk cserélni. Erre jó a kétoperandusú XCHG (eXCHanGe) utasítás, ami a két operandusát felcseréli. Az operandusok vagy 8, vagy 16 bites regiszterek, vagy egy regiszter és egy memóriahivatkozás lehetnek. A flag-eket persze nem bántja. Speciális esetnek számít, ha AX-et cseréljük fel valamelyik másik 16 bites általános regiszterrel, ez ugyanis csak 1 bájtos műveleti kódot eredményez. Az XCHG AX,AX utasítás a NOP (No OPeration) operandus nélküli mnemonikkal is elérhető.

AL tehát ismét az eredeti értékét tartalmazza. Most a felső bitnégyest kellene valahogy leválasztani, törölni AL-ben, hogy ismét alkalmazhassuk az XLAT-ot. Ezt most az AND utasítás végzi el, ami ugyebár a céloperandust a forrásoperandussal logikai ÉS kapcsolatba hozza, majd az eredményt a cél helyén tárolja. Most AL-t a 0Fh értékkel hozza ÉS kapcsolatba, ami annyit fog jelenteni, hogy a felső bitnégyes törlődik (hiszen Valami AND 00. . . 0b = 0h), az alsó pedig változatlan marad (mivel Valami AND 11. . . 1b = Valami).

AL-t most már átalakíthatjuk számjeggyé (XLAT), majd miután DH-ban eltároltuk, DX-et berakjuk a verembe. AH még mindig az eredeti értékét tartalmazza, ideje hát őt is feldolgozni. Az XCHG AL,AH után (ami most MOV AL,AH is lehetne, hiszen a szám alsó felére többé már nincs szükségünk) az előzőkhöz hasonlóan előállítjuk a két számjegyet DX-ben. Ha ez is megvan, akkor nincs más hátra, mint a négy számjegyet kiírni. Ezt a már jól ismert 02h számú INT 21h szolgáltatással tesszük meg.

A 14.1. táblázatban látható, hogy az INT 10h felelős a megjelenítéssel kapcsolatos szolgáltatásokért. Így logikus, hogy ehhez a megszakításhoz fordulunk segítségért a képernyő letörléséhez. A Torol eljárás nagyon rövid, ami azért van, mert a törlést egyetlen INT 10h hívással

elintézhetjük. A 06h-s szolgáltatás egy szöveges képernyőablak felfelé görgetésére (scrolling) szolgál, és úgy működik, hogy az AL-ben megadott számú sorral felfelé csúsztatja a képernyő tartalmát, az alul megüresedő sorokat pedig a BH-ban megadott színű (attribútumú) szóközökkel tölti fel. Ha AL = 0, akkor az egész képernyőt felgörgeti, ami végülis a kép törléséhez vezet. Az ablak bal felső sarkának koordinátáit CX, míg a jobb alsó sarokét DX tartalmazza. A felső bájtok (CH és DH) a sort, az alsók (CL és DL) az oszlopot jelentik, a számozás 0-tól kezdődik. Most úgy tekintjük, hogy a képernyő 80 oszlopos és 25 soros képet jelenít meg, ezért DH-ba 24-et (18h), DL-be pedig 79-et (4Fh) töltünk. A törlés után a kurzor a képernyő legalsó sorának elejére kerül.

A GotoXY eljárás a kurzor pozícionálását teszi meg. A 02h számú video-szolgáltatás a DH-adik sor DL-edik oszlopába rakja a kurzort, a számozás itt is 0 bázisú. BH-ba 0-t rakunk, de ezzel most ne törődjünk. (Akit érdekel, BH-ban a használt képernyőlap sorszámát kell megadni.)

A főprogram nagyon egyszerűre sikeredett, hiszen a legtöbb dolgot eljáráshívással intézi el. A képernyő letörlése (CALL Torol) után belépünk a @Ciklus kezdetű hurokba. Itt a kurzort felrakjuk a bal felső sarokba, ahová kiírjuk a számláló aktuális értékét, majd megnöveljük a változót. Most jön az, hogy meg kell nézni, nyomtak-e le billentyűt. Erre a nemrég látott 00h-s INT 16 szolgáltatás nem jó, hiszen az nekiáll várakozni, ha nincs lenyomva egyetlen gomb sem. Ehelyett a 01h funkciót használjuk fel, ami a ZF-ben jelzi, volt-e billentyű lenyomva: ha ZF = 1, akkor nem, különben AX tartalma megfelel a 00h-s szolgáltatás által visszaadottnak (tehát AL-ben az ASCII kód, AH-ban a scan kód). Így ha ZF = 1, akkor visszamegyünk a ciklus elejére. Máskülönben megnézzük, hogy a szóközt nyomták-e meg, ennek ASCII kódja 32 (20h), és ha igen, akkor vége a bulinak, a @Vege címkén át befejezzük a ténykedést. Egyébként a kurzort a következő (második) sorba állítjuk (erre azért jó most az INC DH, mert DX végig nulla marad a @Ciklus utáni sortól kezdve), majd a megszokott INT 21h 02h számú szolgáltatással kirakjuk a lenyomott billentyű ASCII kódjának megfelelő karaktert, ami AL-ben van. Az utána következő két sor (XOR AH,AH // INT 16h) feleslegesnek tűnhet, de enélkül a következő lenyomott karaktert nem fogja érzékelni a program. (Kicsit precízebben: Az INT 16h egy pufferből olvassa ki a következő lenyomott gomb kódjait. A 01h-s szolgáltatás a puffer mutatóját nem állítja át, így a következő hívás ismét ezt a billentyűt jelezné lenyomottnak, ami nem lenne okés. A 00h hívása viszont módosítja a mutatót, és ezzel minden rendben lesz.) Dolgunk végeztével visszatérünk a ciklusba.

14.3. Munka állományokkal

Az állomány-kezelés nem tartozik a könnyű dolgok közé, de egyszer érdemes vele foglalkozni, sokszor ugyanis egyszerűbben célhoz érhetünk Assemblyben, mint valamelyik magas szintű nyelvben.

A feladat nem túl bonyolult: hozzunk létre egy állományt, majd figyeljük a billentyűzetet. Minden beírt karaktert írjon ki a program a képernyőre és az állományba is folyamatosan és azonnal. Az Esc megnyomására zárja le az állományt és lépjen ki a programból. Az Esc-hez tartozó karaktert már nem kell az állományba írni. Az állomány nevét a programban konstans módon tároljuk, legyen mondjuk TESZT.OUT. Ha nem sikerült létrehozni az állományt, akkor írjon ki egy hibaüzenetet, majd azonnal fejezze be a működést.

Pelda13.ASM:

```
MODEL SMALL
```

```
.STACK

ADAT        SEGMENT
FNev        DB        "TESZT.OUT",00h
Hiba        DB        "Hiba a fájl "
            DB        "létrehozásakor!$"
Karakter    DB        ?
ADAT        ENDS

KOD         SEGMENT
            ASSUME    CS:KOD,DS:ADAT

@Start:
            MOV       AX,ADAT
            MOV       DS,AX
            MOV       AH,3Ch
            XOR       CX,CX
            LEA       DX,[FNev]
            INT       21h
            JC        @Hiba
            MOV       BX,AX
            MOV       CX,0001h
@Ciklus:
            XOR       AH,AH
            INT       16h
            CMP       AL,1Bh
            JE        @Lezar
            MOV       [Karakter],AL
            MOV       AH,02h
            MOV       DL,AL
            INT       21h
            MOV       AH,40h
            LEA       DX,[Karakter]
            INT       21h
            JMP       @Ciklus
@Lezar:
            MOV       AH,3Eh
            INT       21h
            JMP       @Vege
@Hiba:
            MOV       AH,09h
            LEA       DX,[Hiba]
            INT       21h
@Vege:
            MOV       AX,4C00h
            INT       21h
KOD         ENDS
            END       @Start
```

Ha állományokkal, könyvtárakkal vagy lemezekkel kell dolgoznunk, azt mindenképpen a DOS szolgáltatásain keresztül érdemes tenni, hacsak valami egyéb indok (pl. a sebesség kriti-

kus) nem indokol mást. A DOS az összes megszokott tevékenység végrehajtását lehetővé teszi az INT 21h megszakítás szolgáltatásain keresztül, amint ezt a 14.2. táblázat mutatja.

14.2. táblázat. Állomány- és lemezkezelő DOS-szolgáltatások

39h	Könyvtár létrehozás (MKDIR)
3Ah	Könyvtár törlés (RMDIR)
3Bh	Könyvtárváltás (CHDIR)
3Ch	Állomány létrehozás/csonkítás (Create)
3Dh	Állomány megnyitás (Open)
3Eh	Állomány lezárás (Close)
3Fh	Olvasás megnyitott állományból (Read)
40h	Írás megnyitott állományba (Write)
41h	Lezárt állomány törlése (Delete/Unlink)
42h	Fájlmutató pozícionálása (Seek)
43h	Attribútumok beállítása/olvasása (CHMOD)

Ezeknek az állománykezelő függvényeknek közös tulajdonsága, hogy az állományra a megnyitás és/vagy létrehozás után nem az állomány nevével, hanem egy speciális, egyedi azonosítószámmal, az ú.n. **file handle**-lel hivatkoznak. Ez egy 16 bites érték. A szabványos ki-/bemeneti eszközök (standard input és output) a 0000h – 0004h értékeken érhetők el, ezekre is ugyanúgy írhatunk, ill. olvashatunk róluk, mintha közönséges fájlok lennének. A 14.3. táblázat mutatja az ilyen eszközök handle számát.

14.3. táblázat. Standard I/O eszközök handle értéke

0	Standard input (STDIN)
1	Standard output (STDOUT)
2	Standard hiba (STDERR)
3	Standard soros porti eszköz (STDAUX)
4	Standard nyomtató (STDPRN)

Új állomány létrehozására a 3Ch számú szolgáltatás való. Hívásakor CX-ben kell megadni a leendő állomány attribútumait a következő módon: az alsó három bit egyes bitjei kapcsolóknak felelnek meg, 1 jelenti a csak írható (Read Only) hozzáférést, 2 a rejtett fájlt (Hidden), 4 pedig a rendszerfájlt (System). Ezek különböző kombinációja határozza meg a végleges attribútumot. Most nekünk egy normál attribútumú fájlra van szükségünk, így CX nulla lesz. DS:DX tartalmazza az állomány nevének címét, amely név egy **ASCIIZ sztring**, azaz a 00h kódú karakterrel kell lezárni. A szolgáltatás visszatéréskor CF-ben jelzi, hogy volt-e hiba. Ha CF = 1, akkor volt, ilyenkor AX tartalmazza a hiba jellegét leíró hibakódot, különben pedig AX-ben a file handle található. Hiba esetén kiírjuk a hibaüzenetet a 09h-s szolgáltatással, majd befejezzük a programot. Állomány létrehozásánál figyelni kell arra, hogy ha az adott nevű állomány már létezik, akkor a DOS nem tér vissza hibával, hanem az állomány méretét 0-ra állítja, majd megnyitja azt.

Ha a létrehozás sikerült, akkor a handle-t átrakjuk BX-be, mivel a további szolgáltatások már ott fogják keresni.

A billentyűzetről való olvasásra a már említett 00h-s funkciót használjuk. Az Esc billentyű a 17 (1Bh) kódú karaktert generálja, így ha ezt kaptuk meg AL-ben, akkor elugrunk a @Lezar címkére. Megnyitott (ez fontos!) állomány lezárására a 3Eh szolgáltatást kell használni, ami

BX-ben várja a handle-t, visszatéréskor pedig CF ill. AX mutatja az esetleges hibát és annak okát.

Ha nem az Esc-et nyomták le, akkor AL-t berakjuk a Karakter nevű változóba, majd kiíratjuk a képernyőre a már jól ismert 02h DOS-funkcióval. Ezután a 40h számú szolgáltatást vesszük igénybe a fájlba íráshoz. A handle-t itt is BX-ben kell közölni, CX tartalmazza a kiírandó bájtok számát (ez most 1 lesz), DS:DX pedig annak a memóriaterületnek a címe, ahonnan a kiírást kérjük. Visszatéréskor AX a ténylegesen kiírt bájtok számát mutatja, ill. a hibakódot, ha CF = 1. Ezek után visszatérünk a @Ciklus címkére.

14.4. Grafikus funkciók használata

Most ismét egy kicsit nehezebb feladatot oldunk meg, de a fáradozás meg fogja érni. A program először átvált a $640 \cdot 480$ felbontású grafikus videomódba, majd egy pattogó fehér pontot jelenít meg. A pont mindegyik falon (a képernyő szélein) rendesen vissza fog pattanni. Billentyű lenyomására pedig vissza fogja állítani a képernyőmódot szövegesre, majd ki fog lépni.

Pelda14.ASM:

```
MODEL SMALL

.STACK

KOD             SEGMENT
                ASSUME    CS:KOD,DS:NOTHING

@Start:
                MOV       AX,0012h
                INT       10h
                XOR       CX,CX
                MOV       DX,CX
                MOV       SI,3
                MOV       DI,2
                XOR       BH,BH
@Ciklus:
                MOV       AX,0C00h
                INT       10h
                ADD       CX,SI
                JNS       @V1
                NEG       SI
                ADD       CX,SI
@V1:
                CMP       CX,640
                JB        @V2
                NEG       SI
                ADD       CX,SI
@V2:
                ADD       DX,DI
                JNS       @F1
                NEG       DI
```

```
                ADD     DX,DI
@F1:
                CMP     DX,480
                JB      @F2
                NEG     DI
                ADD     DX,DI
@F2:
                MOV     AL,0Fh
                INT     10h
                                        ;Várakozás ...
                PUSH    CX DX
                MOV     DL,10
@Var1:
                XOR     CX,CX
@Var2:
                LOOP    @Var2
                DEC     DL
                JNZ     @Var1
                POP     DX CX
                                        ;
                MOV     AH,01h
                INT     16h
                JZ      @Ciklus
                XOR     AH,AH
                INT     16h
                MOV     AX,0003h
                INT     10h
                MOV     AX,4C00h
                INT     21h
KOD             ENDS
                END     @Start
```

Adatszegmensre most nincs szükségünk, úgyhogy rögtön a videomód beállításával kezdünk. Az INT 10h 00h-s szolgáltatásával állítható be a képernyőn használt videomód, az igényelt mód számát AL-ben kell közölnünk. Sok szabványos videomódnak van előre rögzített száma, de nekünk most csak a $80 \cdot 25$ felbontású színes szöveges (ez a 03h számú mód), ill. a $640 \cdot 480$ felbontású grafikus (12h a száma) VGA-módokra lesz szükségünk. A VGA (Video Graphics Array) a videovezérlő típusára utal, azaz ennél régebbi típusú (EGA, CGA, HGC stb.) videokártyával nem tudjuk kipróbálni a programot, de ez manapság már nem túl nagy akadály.

A pattogó pont koordinátáit a DX és CX regiszter tárolja majd, DX mutatja a sort, CX pedig az oszlopot. Mindkettő 0 bázisú, az origó tehát a (0,0). BH a használt képernyőlap sorszámát tartalmazza, de ez most lényegtelen. SI lesz a vízszintes, DI pedig a függőleges lépésköz, azaz ennyi képponttal fog arrébb ugrani a pont.

A fő ciklusban először törölni kell a pont előző képét, majd ki kell számolni az új koordinátákat, ki kell rakni az új helyére a pontot, végül ellenőrizni kell a billentyűzetet.

A pont törlése annyit jelent, hogy a legutóbbi (CX,DX) koordinátákra fekete színnel kirakunk egy pontot. Pont rajzolására a 0Ch szolgáltatás lesz jó. AL-ben a rajzolt pont színét kell megadni, ehhez tekintsük a 14.4. táblázatot. Látjuk, hogy a fekete kódja a 00h. BH-ban a már említett képernyőlap-sorszám van, a kirakandó pont koordinátáit pedig éppen a CX, DX regiszterekben várja.

Miután töröltük a pont előző példányát, ideje, hogy az új hely koordinátáit meghatározzuk.

14.4. táblázat. Színkódok

00h	Fekete
01h	Kék
02h	Zöld
03h	Ciánkék
04h	Vörös
05h	Bíborlila
06h	Barna
07h	Világosszürke
08h	Sötétszürke
09h	Világoskék
0Ah	Világoszöld
0Bh	Világoscián
0Ch	Világospiros
0Dh	Világosbíbor
0Eh	Sárga
0Fh	Fehér

A vízszintes és függőleges koordinátákat hasonlóan dolgozzuk fel, ezért csak az egyiket nézzük most meg. Tekintsük mondjuk az abszcissza kiszámítását. Először természetesen a lépésközt (SI) adjuk hozzá CX-hez. Ha a kapott érték negatív, akkor SF be fog állni. Eszerint végrehajtjuk a szükséges korrekciót és rátérünk a @V1 címkére, vagy pedig rögtön erre a címkére jövünk. A korrekció abban áll, hogy a lépésköz előjelét megfordítjuk (tehát negáljuk a lépésközt), majd ezt az új lépésközt ismét hozzáadjuk CX-hez. Ha ezen az ellenőrzésen túljutottunk, akkor még meg kell nézni, hogy a képernyő jobb oldalán nem mentünk-e túl. Ha igen (azaz CX > 639), akkor ugyanúgy járunk el, mint az előbb, tehát negáljuk a lépésközt és hozzáadjuk CX-hez. DX módosítása teljesen hasonló módon történik.

Most már kirakhatjuk a pontot új helyére. Mivel fehér pontot szeretnénk, AL-be 0Fh-t rakunk.

Az ez után következő néhány sor nem tartozik az eredeti célkitűzéshez, de a működés ellenőrzéséhez elengedhetetlen. A két pontosvesszővel közrefogott sorok csak a ciklus lassítását szolgálják, és nem csinálnak mást, mint 655360-szor végrehajtják az üres ciklusmagot. Erre a mostani számítógépek gyorsasága miatt van szükség, különben a pont nemhogy pattogna, de valósággal száguldana a képernyőn.

Miután letelt a késleltetés, a 01h-s INT 16h szolgáltatás hívásával megnézzük, hogy nyomtak-e le billentyűt, s ha nem, akkor irány vissza a ciklus elejére.

Kilépés előtt még egyszer kiolvassuk a billentyűzetet, majd visszaállítjuk a videomódot szövegesre.

15. fejezet

Megszakítás-átdefiniálás, hardver-megszakítások, rezidens program, kapcsolat a perifériákkal, hardver-programozás

Ideje, hogy kicsit mélyebben elmerüljünk a megszakítások világában. Nézzük meg először, hogy is hajtja végre a processzor az INT utasítást.

A 0000h szegmens első 1024 bájtján (a 0000h és a 03FFh offszetek közti területen) található a megszakítás-vektor tábla. Ez nem más, mint egy **ugrótábla**, mivel mindegyik bejegyzése egy 4 bájtos távoli pointer (azaz szegmens:offszet alakú memóriacím), nevük **megszakítás-vektor** (interrupt vector). Kicsit utánaszámolva láthatjuk, hogy $256 \cdot 4 = 1024$. Ebben a táblában található tehát mind a 256 db. szoftver- (részben hardver- is) megszakítás végrehajtó programjának kezdőcíme. Ezeket a programokat **megszakítás-kezelőnek** (interrupt handler) nevezzük. Minden megszakításnak pontosan egy kezelőprogramja van, de ugyanaz a kezelő több megszakításhoz is tartozhat.

Ha a programban az INT utasítás kódjával találkozik a processzor (ami 0CDh), akkor kiolvassa az utána levő bájtot is, ez a kért megszakítás száma. Felismerve, hogy megszakítás következik, a verembe berakja sorban a Flags, CS és IP regiszterek aktuális értékét (CS:IP az INT utasítást követő utasítás címét tartalmazza), majd az IF és TF flag-eket törli, ezzel biztosítva, hogy a megszakítás lefolyását nem fogja semmi megakadályozni. (Szimbolikusan: PUSHF // PUSH CS // PUSH IP) Jelölje a kért megszakítás számát N. Ezek után a processzor betölti a CS:IP regiszterpárba a memória 0000h:$(N \cdot 4)$ címén levő duplaszót, azaz CS-be a 0000h:$(N \cdot 4 + 2)$ címen levő, míg IP-be a 0000h:$(N \cdot 4)$ címen található szó kerül betöltésre (emlékezzünk vissza, a processzor little-endian tárolásmódot használ). Ennek hatására az N-edik megszakítás-vektor által mutatott címen folytatódik a végrehajtás.

Ha a kezelő elvégezte dolgát, akkor egy különleges utasítás segítségével visszatér az őt hívó programhoz. Erre az IRET (Interrupt RETurn) operandus nélküli utasítás szolgál. Az IRET kiadásakor a processzor a visszatérési címet betölti a CS:IP-be (azaz leemeli először az IP-t, majd CS-t a veremből), a Flags regiszter tartalmát szintén visszaállítja a veremből, majd a végrehajtást az új címen folytatja. (Szimbolikusan: POP IP // POP CS // POPF)

Nézzünk egy példát! Tegyük fel, hogy az INT 21h utasítást adjuk ki, és legyen Flags =

0F283h (azaz IF = SF = CF = 1, a többi flag mind 0), CS = 7000h, IP = 1000h, SS lényegtelen, SP = 0400h. Az INT 21h megszakítás vektora a 0000h:0084h címen található, ennek értéke most legyen mondjuk 2000h:3000h. Az INT 21h utasítás végrehajtásának hatására a következők történnek: CS:IP értéke 2000h:3000h lesz, Flags-ben törlődik az IF flag, így az a 0F083h értéket fogja tartalmazni, valamint a verem a következő képet fogja mutatni:

	SS:0400h : ??
	SS:03FEh : 0F283h (Flags)
	SS:03FCh : 7000h (CS)
SP⇒	SS:03FAh : 1000h (IP)
	SS:03F8h : ??

Mivel tudjuk, hogy hol találhatók az egyes megszakítások belépési pontjai, megtehetjük, hogy bármelyik vektort kedvünkre átírjuk. Ezzel a lehetőséggel rengeteg program él is, elég, ha a DOS-t, BIOS-t, vagy mondjuk az egeret kezelő programot (eszközmeghajtót) említjük. És mivel a hardver-megszakítások is bizonyos szoftver-megszakításokon keresztül lesznek lekezelve, akár ezeket is átirányíthatjuk saját programunkra. Mindkét esetre mutatunk most példákat.

15.1. Szoftver-megszakítás átirányítása

Első programunk a következőt fogja tenni: indításkor át fogja venni az INT 21h kezelését. Az új kezelő a régit fogja meghívni, de ha az igényelt funkció a 09h-s lesz (az a bizonyos sztringkiíró rutin), akkor DS:DX-et egy előre rögzített szöveg címére fogja beállítani, és arra fogja meghívni az eredeti szövegkiíró funkciót. A program billentyű lenyomása után vissza fogja állítani az eredeti megszakítás-kezelőt, majd be fog fejeződni.

Pelda15.ASM:

```
MODEL SMALL

.STACK

ADAT        SEGMENT
Helyes      DB       "Ha ezt látod, akkor"
            DB       " működik a dolog.$"
Teszt       DB       "Ez nem fog megjelenni!$"
ADAT        ENDS

KOD         SEGMENT
            ASSUME   CS:KOD,DS:ADAT

RegiCim     DW       ?,?

UjKezelo    PROC
            PUSHF
            CMP      AH,09h
            JNE      @Nem09h
            POPF
            PUSH     DS
            PUSH     DX
```

```
                MOV     DX,ADAT
                MOV     DS,DX
                LEA     DX,[Helyes]
                PUSHF
                CALL    DWORD PTR CS:[RegiCim]
                POP     DX
                POP     DS
                IRET
@Nem09h:
                POPF
                JMP     DWORD PTR CS:[RegiCim]
UjKezelo        ENDP

@Start:
                MOV     AX,ADAT
                MOV     DS,AX
                XOR     AX,AX
                MOV     ES,AX
                CLI
                LEA     AX,[UjKezelo]
                XCHG    AX,ES:[21h*4]
                MOV     CS:[RegiCim],AX
                MOV     AX,CS
                XCHG    AX,ES:[21h*4+2]
                MOV     CS:[RegiCim+2],AX
                STI
                MOV     AH,09h
                LEA     DX,[Teszt]
                INT     21h
                MOV     AH,02h
                MOV     DL,0Dh
                INT     21h
                MOV     DL,0Ah
                INT     21h
                XOR     AH,AH
                INT     16h
                CLI
                MOV     AX,CS:[RegiCim]
                MOV     ES:[21h*4],AX
                MOV     AX,CS:[RegiCim+2]
                MOV     ES:[21h*4+2],AX
                STI
                MOV     AX,4C00h
                INT     21h
KOD             ENDS
                END     @Start
```

Az első szembetűnő dolog az, hogy a kódszegmensben van definiálva a RegiCim nevű változó, ami majd az eredeti INT 21h kiszolgáló címét fogja tárolni. Ennek magyarázata a megszakítás-kezelő rutin működésében rejlik.

Azt már említettük, hogy minden eljárás elején érdemes és illendő elmenteni a használt regisztereket a verembe, hogy azokat az eljárás végén gond nélkül visszaállíthassuk eredeti

értékükre. Ez a szabály a megszakítás-kezelő rutinokra kötelezőre változik. A regiszterekbe a szegmensregisztereket és a Flags regisztert is bele kell érteni. Ismétlésként, az operandus nélküli PUSHF utasítás a verembe berakja a Flags regisztert, míg a POPF a verem tetején levő szót a Flags-be tölti be.

A saját kezelő rutinunk működése két ágra fog szakadni aszerint, hogy AH egyenlő-e 09h-val avagy nem. A feltétel tesztelését a hagyományos CMP-JNE párossal oldjuk meg, de mivel ez az utasítás megváltoztat(hat) néhány flag-et, ezért a tesztelés előtt PUSHF áll, valamint mindkét ág a POPF-fel indul. Ez garantálja, hogy az összes flag érintetlenül marad. Erre kérdezhetné valaki, hogy miért őrizzük meg a flag-ek értékét, ha a Flags úgyis el van mentve a veremben. Nos, már láttuk, hogy néhány szolgáltatás pl. a CF-ben jelzi, hogy volt-e hiba. Ezek a szolgáltatások visszatérés során a veremben levő Flags regiszter-példányt egyszerűen eldobják (később látni fogjuk, hogyan), így módosításunk a hívó programra is visszahatna, ami kellemetlen lehetne.

Ha AH $\neq$ 09h, akkor a @Nem09h talányos nevű címkén folytatódik a megszakítás kiszolgálása. Azt mondtuk, hogy minden egyéb esetben a régi funkciót fogjuk végrehajtatni. Ennek megfelelően cselekszik a programban szereplő JMP utasítás. Ezt a mnemonikot már ismerjük és használtuk is. Itt viszont két újdonságot is láthatunk: az első, hogy az operandus nem egy eljárás neve vagy címkéje mint eddig, hanem egy memóriahivatkozás (szegmensprefixszel együtt). Az ugrás eme formáját **közvetett** avagy **indirekt ugrásnak** (indirect jump) nevezzük, míg a régebbi alakot **közvetlen** vagy **direkt ugrásnak** (direct jump). Az elnevezés arra utal, hogy a cél címét nem az operandus, hanem az operandus által mutatott memóriacímen levő változóban levő pointer határozza meg. Magyarán: a JMP-t úgy hajtja végre a processzor, hogy kiszámítja az operandus (esetünkben a CS:RegiCim változó) tényleges címét, majd az azon a címen levő pointert tölti be a megfelelő regiszterekbe (IP-be vagy a CS:IP párba). A másik újdonságot a DWORD PTR képviseli, ennek hatására az operandus típusa most DWORD lesz, ami duplaszót jelent ugyebár. Ha helyette WORD PTR állna, akkor a JMP a szokásos **szegmensen belüli** avagy **közeli ugrást** (intrasegment vagy near jump) tenné meg, azaz csak IP-t változtatná meg. Nekünk viszont CS:IP-t kell új értékekkel feltöltenünk, és ez már **szegmensközi** avagy **távoli ugrást** jelent (intersegment vagy far jump). A DWORD helyett állhatna még a FAR is, az ugyancsak távoli pointert írna elő. Az utasítás most tehát CS-be a CS:(RegiCim + 2) címen levő, míg IP-be a CS:RegiCim címen levő szót tölti be, majd onnan folytatja a végrehajtást. A szemfülesebbek rögtön keresni kezdik az említett IRET-et. Erre most nekünk nincs szükségünk, hiszen SP a régi (híváskori) állapotában van, a Flags is érintetlen, és az eredeti kiszolgálóból való visszatérés után nem akarunk már semmit sem csinálni. Ezért a legegyszerűbb módszert választjuk: a feltétlen vezérlésátadást a régi rutinra. Ha az a rutin befejezteténykedését, a veremben az eredeti Flags, CS és IP értékeket fogja találni, és így közvetlenül a hívó programba (nem az UjKezelo eljárásba) fog visszatérni.

Ha AH = 09h, akkor el kell végeznünk DS:DX módosítását tervünknek megfelelően. Mivel nem szép dolog, ha a változás visszahat a hívóra, mindkét regisztert elmentjük a verembe. Ez a művelet azonban meggátol bennünket abban, hogy a másik esethez hasonlóan ráugorjunk a régi kezelő címére. Ha ugyanis a JMP DWORD PTR CS:[RegiCim] utasítást alkalmaznánk itt is, akkor a hívó programba visszatérés nem igazán sikerülne, mondhatni csődöt mondana. Hogy miért? A kulcs a verem. A legutolsó két veremművelettel a verem tetején a DS és DX regiszterek híváskori értéke lesz, amit a régi kezelőrutin a CS:IP regiszterekbe fog betölteni, és ez valószínűleg katasztrofális lesz (nem beszélve arról, hogy SP is meg fog változni a híváskori helyzethez képest). Ezért most ugrás helyett hívást kell alkalmaznunk. Ez a hívás is kicsit más, mint az eddig megismert. A CALL utasítás ilyen alakját a fenti példához hasonlóan **távoli indirekt eljáráshívásnak** nevezzük. (A CALL-nak is létezik közeli és távoli alakja, és mindkettőből van direkt és indirekt változat is.) A CALL hatására CS és IP bekerül a verembe, a Flags

viszont nem, miközben a régi kiszolgálórutin arra számít, hogy az SS:(SP + 4) címen (ez nem szabályos címzésmód!) a Flags tükörképe van. Ezért a CALL előtt még kiadunk egy PUSHF-et. Aki kicsit jobban elgondolkodik, annak feltűnhet, hogy ez az utasításpáros tulajdonképpen egy INT utasítást szimulál. Így ha a régi kiszolgáló végez, akkor visszatér a saját kezelőnkbe, ahol mi kitakarítjuk a veremből a DS és DX értékeit, majd visszatérünk a hívó programba. Ezt most IRET-tel tesszük meg annak ellenére, hogy említettük, némelyik szolgáltatás esetleg valamelyik flag-ben adhatna vissza eredményt. Mi most viszont csak a 09h számú szolgáltatás működésébe avatkozunk be, ami nem módosítja egyik flag-et sem.

Ha egészen korrekt megoldást akarunk, akkor az IRET utasítást a RETF 0002h utasítással kell helyettesíteni. A RETF (Far RETurn) az eddig használt RET utasítástól abban tér el, hogy visszatéréskor nemcsak IP-t, de CS-t is visszaállítja a veremből (tehát működése szimbolikusan POP IP // POP CS). Az opcionális szó méretű közvetlen operandus azt az értéket jelöli, amit azután (t.i. IP és CS POP-olása után) SP-hez hozzá kell adnia. A visszatérés eme formája annyiban tér el az IRET-től, hogy a Flags eredeti értékét nem állítja vissza a veremből, a megadott 0002h érték hatására viszont SP-t úgy módosítja, mintha egy POPF-et is végrehajtottunk volna. Az eredmény: a hívóhoz gond nélkül visszatérhetünk a megszakításból úgy, hogy esetleg valamelyik flag eredményt tartalmaz.

Nem válaszoltuk még meg, hogy miért a kódszegmensben definiáltuk a RegiCim változót. A válasz sejthető: ha az adatszegmensben lenne, akkor annak eléréséhez először be kellene állítanunk DS-t. Ez viszont ellentmond annak a követelménynek, hogy DS értékét nem (sem) szabad megváltoztatnunk abban az esetben, ha AH ≠ 09h. Ezt csak valamilyen csellel tudnánk biztosítani, pl. így:

```
PUSH    DS DX
MOV     DX,ADAT
MOV     DS,DX
PUSH    WORD PTR DS:[RegiCim+2]
PUSH    WORD PTR DS:[RegiCim]
POP     DX DS
RETF
```

Hangsúlyozzuk, ezt csak akkor kellene így csinálni, ha az ADAT nevű szegmensben definiáltuk volna a RegiCim változót. Mivel azonban a kódszegmensbe raktuk a definíciót, a kezelő rutin közvetlenül el tudja érni a változót. (Ehhez azért az is kell, hogy az INT 21h kiadása után a saját kezelő rutinunk CS szegmense megegyezzen a RegiCim szegmensével, de ez most fennáll.)

ES-t eddig még nem sokszor használtuk, most viszont kapóra jön a megszakítás-vektor táblázat szegmensének tárolásánál.

Mielőtt nekifognánk átírni az INT 21h vektorát, a CLI utasítással letiltjuk a hardver-megszakításokat (IF = 0 lesz). Erre azért van szükség, mert ha az átírás közepén bejönne egy megszakítás, akkor azt a processzor minden további nélkül kiszolgálná. Ha annak kezelőjében szerepelne egy INT 21h utasítás, akkor a rossz belépési cím miatt nem a megfelelő rutin hívódna meg, és ez valószínűleg álomba küldené a gépet. A vektor módosítása után ismét engedélyezzük a bejövő megszakításokat az IF 1-be állításával, amit az STI utasítás végez el.

A vektor módosításánál az XCHG egy csapásra megoldja mind a régi érték kiolvasását, mind az új érték beírását.

Miután sikeresen magunkra irányítottuk ezt a megszakítást, rögtön ki is próbáljuk. A helyes működés abban áll, hogy nem a Teszt címkéjű szöveg fog megjelenni, hanem a Helyes. Annak ellenőrzésére, hogy a többi szolgáltatást nem bántottuk, a 02h funkcióval egy új sor karakterpárt

írunk ki.

Végül gombnyomás után visszaállítjuk az eredeti kezelőt, és kilépünk a programból.

15.2. Az időzítő (timer) programozása

A következő program már a hardver-programozás tárgykörébe tartozik. A feladat az, hogy írjunk egy olyan eljárást, ami ezredmásodperc (millisecundum) pontossággal képes meghatározott ideig várakozni. Ezzel például stoppert is megvalósíthatunk. Ez a rész kicsit magasabb szintű a szokásosnál, így ha valaki esetleg nem értené, akkor nyugodtan ugorja át. :)

Pelda16.ASM:

```
MODEL SMALL

.STACK

KOD             SEGMENT
                ASSUME      CS:KOD,DS:NOTHING

Orajel          DW          ?
Oszto           DW          ?
MSec            DW          ?
RegiCim         DW          ?,?

UjKezelo        PROC
                PUSH        AX
                INC         CS:[MSec]
                MOV         AX,CS:[Oszto]
                ADD         CS:[Orajel],AX
                JC          @Regi
                MOV         AL,20h
                OUT         20h,AL
                POP         AX
                IRET
@Regi:
                POP         AX
                JMP         DWORD PTR CS:[RegiCim]
UjKezelo        ENDP

@Start:
                XOR         AX,AX
                MOV         DS,AX
                CLI
                LEA         AX,[UjKezelo]
                XCHG        AX,DS:[08h*4]
                MOV         CS:[RegiCim],AX
                MOV         AX,CS
                XCHG        AX,DS:[08h*4+2]
                MOV         CS:[RegiCim+2],AX
                MOV         DX,0012h
                MOV         AX,34DCh
```

```
                MOV     BX,1000
                DIV     BX
                MOV     CS:[Oszto],AX
                MOV     CS:[Orajel],0000h
                MOV     BL,AL
                MOV     AL,36h
                OUT     43h,AL
                JMP     $+2
                MOV     AL,BL
                MOV     DX,0040h
                OUT     DX,AL
                JMP     $+2
                MOV     AL,AH
                OUT     DX,AL
                JMP     $+2
                STI
                MOV     AH,02h
                MOV     DL,'1'
                MOV     CX,9
                MOV     BX,1000
@Ciklus:
                INT     21h
                INC     DL
                MOV     CS:[MSec],0000h
@Var:
                CMP     CS:[MSec],BX
                JB      @Var
                LOOP    @Ciklus
                MOV     DL,0Dh
                INT     21h
                MOV     DL,0Ah
                INT     21h
                CLI
                MOV     AX,CS:[RegiCim]
                MOV     DS:[08h*4],AX
                MOV     AX,CS:[RegiCim+2]
                MOV     DS:[08h*4+2],AX
                MOV     AL,36h
                OUT     43h,AL
                JMP     $+2
                XOR     AL,AL
                OUT     40h,AL
                JMP     $+2
                OUT     40h,AL
                JMP     $+2
                STI
                MOV     AX,4C00h
                INT     21h
KOD             ENDS
                END     @Start
```

Adatszegmensre nem lesz szükségünk, a változókat a könnyebb elérhetőség miatt a kód-

szegmensben definiáljuk.

A PC hardverének áttekintésekor már megemlítettük az időzítőt (timer). Ez az egység három független számlálóval (counter) rendelkezik (szemléletesen három stopperrel), és mindegyik egy időzítő-csatornához tartozik. Ebből egyik a már szintén említett memória-frissítéshez kell, egy pedig a belső hangszóróra van rákötve. A harmadik felelős a belső rendszeróra karbantartásáért, valamint a floppy meghajtó motorjának kikapcsolásáért. Bár elsőre nem látszik, a megoldást ez utóbbi fogja jelenteni. Ha ez a számláló lejár, az időzítő egy megszakítást kér, ez a megszakítás-vezérlő IRQ0-ás vonalán keresztül valósul meg. Végül a hardver-megszakítást a processzor érzékeli, és ha lehet, kiszolgálja. Nos, ha az eredeti kiszolgálót mi lecseréljük a saját rutinunkra, ami a speciális feladatok elvégzése után meghívja a régi kiszolgálót, akkor nyert ügyünk van.

Az időzítő egy kvarckristályon keresztül állandó frekvenciával kapja az áramimpulzusokat. Ez a frekvencia az 12 34DCh Hz (1193180 Hz). Az időzítő mindhárom számlálójához tartozik egy 16 bites osztóérték. A dolog úgy működik, hogy az adott számláló másodpercenként 12 34DCh/Osztó-szor jár le. A rendszerórához tartozó számláló (ez a 0-ás csatorna) osztója 65536, míg a memória-frissítésért felelős (ez az 1-es csatorna) számlálóé 18 alapállapotban. Ha elvégezzük az osztásokat, azt látjuk, hogy a memóriát másodpercenként kb. 66288-szor frissítik, míg a rendszerórát másodpercenként kb. 18.2-szer állítja át a kiszolgálórutin.

Tervünket úgy fogjuk megvalósítani, hogy az időzítő 0-ás csatornájának osztóját olyan értékre állítjuk be, hogy másodpercenként 1000-szer generáljon megszakítást. A megszakítást az IRQ0 vonalról a megszakítás-vezérlő átirányítja az INT 08h szoftver-megszakításra, de ettől ez még hardver-megszakítás marad, aminek fontosságát később látjuk majd. Ha IF = 1, a processzor érzékeli a megszakítást, majd meghívja az INT 08h kezelőjét, ami elvégzi a teendőket. Ezt a kezelőt cseréljük le egy saját rutinra, ennek neve UjKezelo lesz.

Nézzük meg először a változók szerepét. Az eredeti kezelő címét ismét a RegiCim tárolja. Oszto tartalmazza a számlálóhoz tartozó kiszámolt osztóértéket. MSec tartalma minden megszakítás-kéréskor eggyel fog nőni, ezt használjuk fel majd a pontos várakozás megvalósítására. Az Orajel változó szerepének megértéséhez szükséges egy dolgot tisztázni. Mikor is kell meghívni a régi kiszolgálót? Ha minden megszakítás esetén meghívnánk, akkor az óra az eredetinél sokkal gyorsabban járna, és ezt nem szeretnénk. Pontosan úgy kell működnie, mint előtte. Ehhez nekünk is üzemeltetnünk kell egy számlálót. Ha ez lejár, akkor kell meghívni a régi kiszolgálót. Számláló helyett most osztót fogunk bevezetni, ennek mikéntje mindjárt kiderül.

Az UjKezelo eljárásban csak az AX regisztert használjuk fel, ezt tehát rendesen elmentjük a verembe. Ezt követően megnöveljük eggyel az MSec értékét célunk elérése érdekében. Most következik annak eldöntése, meg kell-e hívni a régi kiszolgálót. Ehhez az osztóértéket (Oszto tartalmát) hozzáadjuk az Orajel változóhoz. Ha átvitel keletkezik, az azt jelenti, hogy Orajel értéke nagyobb vagy egyenlő lett volna 65536-nál. Mivel azt mondtuk, hogy Orajel egy osztó szerepét játssza, innen következik, hogy a régi kezelőrutint pontosan akkor kell meghívni, amikor CF = 1 lesz. Gyakorlatilag az időzítő működését utánozzuk: ha az osztóértékek összege eléri vagy meghaladja a 65536-ot, és ekkor hívjuk meg a régi rutint, ez pontosan azt csinálja, mint amikor az időzítő az eredeti 65536-os osztó szerinti időközönként vált ki megszakítást. Némi számolással és gondolkozással ezt magunk is beláthatjuk.

Ha CF = 1, akkor szépen ráugrunk a régi kiszolgáló címére, s az majd dolga végeztével visszatér a megszakításból.

Ha viszont CF = 0, akkor nekünk magunknak kell kiadni a parancsot a visszatéréshez. Előtte azonban meg kell tenni egy fontos dolgot, és itt lesz szerepe annak a ténynek, hogy ez a rutin mégiscsak hardver-megszakítás miatt hajtódik végre. Dolgunk végeztével jelezni kell a megszakítás-vezérlőnek, hogy minden oké, lekezeltük az adott hardver-megszakítást. Ha ezt

a **nyugtázásnak** (acknowledgement) nevezett műveletet elfelejtjük megtenni, akkor az esetlegesen beérkező többi megszakítást nem fogja továbbítani a processzor felé a vezérlő. A megszakítás-vezérlőt a 0020h és 0021h számú portokon keresztül érhetjük el (AT-k esetében a második vezérlő a 00A0h és 00A1h portokon csücsül). Az OUT (OUTput to port) utasítás a céloperandus által megadott portra kiírja a forrásoperandus tartalmát. A port számát vagy 8 bites közvetlen adatként (0000h – 00FFh portok esetén), vagy a DX regiszterben kell közölni, míg a forrás csak AL vagy AX lehet. Ha a forrás 16 bites, akkor az alsó bájt a megadott portra, míg a felső bájt a megadott után következő portra lesz kiírva. A nyugtázás csak annyit jelent, hogy a 0020h számú portra ki kell írni a 20h értéket (könnyű megjegyezni:). Ezután IRET-tel befejezzük ténykedésünket.

Megjegyezzük, hogy az OUT párja az IN (INput from port) utasítás. Működése: a forrásoperandus által megadott portról beolvasott adatot a céloperandusába írja. A portszámot szintén vagy bájt méretű közvetlen adatként, vagy a DX regiszterrel kell megadni, célként pedig ismét AL-t vagy AX-et írhatunk. Ha a cél 16 bites, az alsó bájt a megadott portról, a felső bájt pedig az utána következő, eggyel nagyobb számúról lesz beolvasva.

Mivel nincs adatszegmens, most DS-t használjuk fel a megszakítás-vektorok szegmensének tárolására. A megszakítások letiltása után a már szokott módon eltároljuk a régi INT 08h kezelő címét, ill. beállítjuk a saját eljárásunkat. Ezt követően kiszámoljuk az Oszto értékét. Mint említettük, az időzítő alapfrekvenciája 12 34DCh, és minden ezredmásodpercben akarunk majd megszakítást, így az előző értéket 1000-rel elosztva a hányados a kívánt értéket fogja adni. Aki nem értené, ez miért van így, annak egy kis emlékeztető:

$$12\ 34DCh/osztó = frekvencia$$

Ezt az egyenletet átrendezve már következik a módszer helyessége. Az Orajel változót kinullázzuk a megfelelő működéshez. Ezután az osztót még az időzítővel is közölni kell, a három OUT utasítás erre szolgál. Ezekkel most nem foglalkozunk, akit érdekel, az nyugodtan nézzen utána valamelyik említett adatbázisban. A JMP utasítások csak arra szolgálnak, hogy kicsit késleltessék a további bájtok kiküldését a portokra, a perifériák ugyanis lassabban reagálnak, mint a processzor. Megfigyelhetjük a JMP operandusában a cél megjelölésekor a $ szimbólum használatát. Emlékeztetőül: a $ szimbólum az adott szegmensben az aktuális sor offszetjét tartalmazza. Ez a sor egyéb érdekességet is tartogat, a használt ugrás ugyanis sem nem közeli (near), sem nem távoli (far). Ezt az ugrásfajtát **rövid ugrásnak** (short jump) nevezzük, és ismertetőjele, hogy a relatív cím a feltételes ugrásokhoz és a LOOP-hoz hasonlóan csak 8 bites. A 2-es szám onnan jön, hogy a rövid ugrás utasításhossza 2 bájt (egy bájt a műveleti kód, egy pedig a relatív cím). A rövid ugrást még kétféleképpen is kikényszeríthetjük: az egyik, hogy a céloperandus elé odaírjuk a SHORT operátort, a másik, hogy JMP helyett a JMPS (JuMP Short) mnemonikot használjuk.

A megszakítás működésének tesztelésére egy rövid ciklus szolgál, ami annyit tesz, hogy kiírja a decimális számjegyeket 1-től 9-ig, mindegyik jegy után pontosan 1 másodpercet várakozva. A várakozást az MSec változó figyelésével tesszük annak nullázása után. MSec-et a megszakítások bekövetkeztekor növeli egyesével az UjKezelo eljárás, ez pedig ezredmásodpercenként történik meg. Ha tehát MSec = 1000, akkor a nullázás óta pontosan egy másodperc telt el.

A ciklus lejárta után egy új sor karakterpárt írunk még ki, majd végezetül visszaállítjuk az eredeti INT 08h kezelőt, valamint az időzítő eredeti osztóját. Ezután befejezzük a program működését. Az időzítővel kapcsolatban még annyit, hogy a 0000h érték jelöli a 65536-os osztót.

A program történetéhez hozzátartozik, hogy az első verzióban az UjKezelo eljárás legutolsó sorában a DWORD helyén FAR állt. A TASM 4.0-ás verziója azonban furcsa módon nem jól

fordítja le ezt az utasítást ebben a formában, csak akkor, ha DWORD-öt írunk típusként. Erre későbbi programjainkban nem árt odafigyelni, ugyanis lehet, hogy a forrás szemantikailag jó, csak az assembler hibája (tévedése) miatt nem megfelelő kód szerepel a futtatható állományban. Erről általában a debugger segítségével győződünk meg.

15.3. Rezidens program (TSR) készítése, a szöveges képernyő közvetlen elérése

Rezidens programon (resident program) egy olyan alkalmazást értünk, amihez tartozó memóriaterületet vagy annak egy részét a DOS nem szabadítja fel a program befejeződésekor. A program kódja ilyenkor a memóriában bentmarad. Az inaktív programot sokszor egy megszakítás vagy valamilyen hardveresemény éleszti fel. A programok másik gyakran használt elnevezése a **TSR** (Terminate and Stay Resident). Néhány jól ismert segédprogram is valójában egy TSR, mint pl. a DOS CD-meghajtókat kezelő interfésze (MSCDEX.EXE), egérmeghajtók (MOUSE.COM, GMOUSE.COM stb.), különféle képernyőlopók, commanderek (Norton Commander, Volkov Commander, DOS Navigator) stb.

TSR-t .COM programként egyszerűbb írni, így most mi is ezt tesszük. A feladat az, hogy a program rezidensen maradjon a memóriában az indítás után, és a képernyő bal felső sarkában levő karakter kódját és annak színinformációit folyamatosan növelje eggyel. (Csak szöveges módban!) Nem túl hasznos, de legalább látványos.

Pelda17.ASM:

```
MODEL TINY

KOD             SEGMENT
                ASSUME   CS:KOD,DS:KOD
                ORG      0100h

@Start1:
                JMP      @Start

RegiCim         DW       ?,?

UjKezelo        PROC
                PUSHF
                CALL     DWORD PTR CS:[RegiCim]
                PUSH     DI ES
                MOV      DI,0B800h
                MOV      ES,DI
                XOR      DI,DI
                INC      BYTE PTR ES:[DI]
                INC      DI
                INC      BYTE PTR ES:[DI]
                POP      ES DI
                IRET
UjKezelo        ENDP

@Start:
```

```
                XOR     AX,AX
                MOV     ES,AX
                CLI
                LDS     SI,ES:[1Ch*4]
                MOV     CS:[RegiCim],SI
                MOV     CS:[RegiCim+2],DS
                LEA     AX,[UjKezelo]
                MOV     ES:[1Ch*4],AX
                MOV     ES:[1Ch*4+2],CS
                STI
                LEA     DX,@Start
                INT     27h
KOD             ENDS
                END     @Start1
```

A feladat megoldásához egy olyan megszakításra kell „ráakaszkodni”, ami elég sokszor hívódik meg. Ez lehetne az eddig megismertek közül INT 08h, INT 16h, esetleg az INT 21h. Az időzítő által kiváltott INT 08h eredeti kezelője dolga végeztével egy INT 1Ch utasítást ad ki, aminek a kezelője alapesetben csak egy IRET-ből áll. Ezt a megszakítást bárki szabadon átirányíthatja saját magára, feltéve, hogy meghívja az eredeti (pontosabban a megszakítás-vektor szerinti) kezelőt, illetve hogy a megszakításban nem tölt el túl sok időt. Az időkorlát betartása azért fontos, mert az INT 08h ugyebár egy hardver-megszakítás, és ha a megszakítás-vezérlőt csak az INT 1Ch lefutása után nyugtázza a kezelő, akkor az eltelt időtartam hossza kritikussá válhat.

Adatszegmensünk most sem lesz, az egy szem RegiCim változót a kódszegmens elején definiáljuk, és mellesleg a rezidens részben foglal helyet.

Az UjKezelo eljárás elején hívjuk meg az előző kezelőt, s ez után végezzük el a képernyőn a módosításokat.

A képernyőre nem csak a DOS (INT 21h) vagy a video BIOS (INT 10h) segítségével lehet írni. A megjelenítendő képernyőtartalmat a videovezérlő kártyán levő videomemóriában tárolják el, majd annak tartalmát kiolvasva készül el a végleges kép a monitoron. Ennek a memóriának egy része a fizikai memória egy rögzített címtartományában elérhető. Magyarra fordítva ez azt jelenti, hogy grafikus módok esetén a 000A 0000h, míg szöveges módoknál a 000B 0000h (fekete-fehér) vagy a 000B 8000h (színes) fizikai címeken kezdődő terület a videokártyán levő memória aktuális állapotát tükrözi, és oda beírva valamit az igazi videomemória is módosulni fog. Grafikus módoknál a 0A000h szegmens teljes terjedelmében felhasználható, míg szöveges mód esetén a 0B000h és 0B800h szegmenseknek csak az első fele (tehát a 0000h – 7FFFh offszetek közötti terület). Bennünket most csak a szöveges módok érdekelnek, azokon belül is a színes képernyők esetén használatosak. Fekete-fehér (monokróm) képet előállító videokártya esetében a 0B000h szegmenset kell használni, egyéb változtatásra nincs szükség.

A videomemóriában a képernyőn látható karakterek sorfolytonosan helyezkednek el, és minden karaktert egy újabb bájt követ, ami a színinformációt (attribútumot) tartalmazza. Ez pontosan azt jelenti, hogy a karakterek a páros, míg az attribútum-bájtok a páratlan offszetcímeken találhatók. Az attribútumot a következő módon kódolják: a 0 – 3 bitek adják az előtérszínt, a 4 – 6 bitek a hátteret, míg a 7-es bit villogást ír elő, de úgy is beállítható a videokártya, hogy a háttérszín 4. bitjét jelentse. A színkódok megegyeznek a már korábban (14.4. táblázat) leírtakkal.

Az UjKezelo rutinhoz visszatérve, a bal felső sarokban levő karakter a 0B800h:0000h címen helyezkedik el, amit az attribútum követ. Miután mindkét bájtot külön-külön inkrementáltuk,

IRET-tel visszatérünk a hívóhoz.

A főprogram szokatlanul rövidre sikerült, hiszen nincs sok feladata. Mindössze a régi megszakítás-vektort menti el, majd beállítja az újat. Az egyetlen újdonságot az LDS (Load full pointer into DS and a general purpose register) utasítás képviseli, ami a DS:céloperandus regiszterpárba a forrásoperandus által mutatott memóriaterületen levő távoli mutatót (azaz szegmenst és offszetet) tölti be. Ez a mostani példában azt jelenti, hogy SI-be az ES:$(1\text{Ch} \cdot 4)$, míg DS-be az ES:$(1\text{Ch} \cdot 4 + 2)$ címeken levő szavakat tölti be. Hasonló műveletet végez az LES utasítás, ami DS helyett ES-t használja. A céloperandus csak egy 16 bites általános célú regiszter, a forrás pedig csak memóriahivatkozás lehet mindkét utasításnál.

Ezek után a főprogram be is fejezi működését, de nem a hagyományos módon. Az INT 27h egy olyan DOS-megszakítás, ami a program működését úgy fejezi be, hogy annak egy része a memóriában marad (azaz TSR-ré válik). A rezidens rész a CS:DX címig tart, kezdete pedig a PSP szegmense. A PSP-re és a .COM programok szerkezetére a 13. fejezet világít rá. Mi most csak a RegiCim változót és az UjKezelo eljárást hagyjuk a memóriában (meg a PSP-t, de ez most nem lényeges).

Az INT 27h-val legfeljebb 64 Kbájtnyi terület tehető rezidenssé, és kilépéskor nem adhatunk vissza hibakódot (exit code), amit az INT 21h 4Ch funkciójánál AL-ben közölhettünk. Ezeket a kényelmetlenségeket küszöböli ki az INT 21h 31h számú szolgáltatása. Ennek AL-ben megadhatjuk a szokásos visszatérési hibakódot, DX-ben pedig a rezidenssé teendő terület méretét kell megadnunk paragrafusokban (azaz 16 bájtos egységekben), a PSP szegmensétől számítva.

Megszakítás-vektorok beállítására és olvasására a DOS is kínál lehetőséget. Az INT 21h 25h számú szolgáltatása az AL-ben megadott számú megszakítás vektorát a DS:DX által leírt címre állítja be, míg a 35h szolgáltatás az AL számú megszakítás-vektor értékét ES:BX-ben adja vissza.

16. fejezet

Kivételek

A **kivétel** (exception) egy olyan megszakítás, amit a processzor vált ki, ha egy olyan hibát észlel, ami lehetetlenné teszi a következő utasítás végrehajtását, vagy egy olyan esemény történt, amiről a programoknak és a felhasználónak is értesülniük kell. Minden kivételt egy decimális sorszámmal és egy névvel azonosítanak, ezenkívül minden kivételhez tartozik egy kettőskeresztből (#) és két betűből álló rövidítés, amit szintén mnemoniknak hívnak (ez viszont nem egy utasításra utal).

A 8086-os mikroprocesszor összesen 4 féle kivételt képes generálni. A kivétel kiváltásához szükséges feltételek teljesülése esetén a processzor a verembe berakja a Flags, CS és IP regiszterek tartalmát (szimbolikusan PUSHF // PUSH CS // PUSH IP), majd törli az IF és TF flageket. Ezután sor kerül a kivételt lekezelő programrész (exception handler) végrehajtására. Az Intel az INT 00h – INT 1Fh szoftver-megszakításokat a kivétel-kezelő programok számára tartja fenn. Egy adott N sorszámú kivétel esetén az INT N megszakítás kezelője lesz végrehajtva. A teendők elvégzése után a kezelő IRET utasítással visszatérhet abba a programba, amely a kivételt okozta. Bár a későbbi processzorokon megjelentek olyan, nagyon súlyos rendszerhibát jelző kivételek, amik után a kivételt okozó program már nem indítható újra (vagy nem folytatható), a 8086-os processzoron mindegyik kivétel megfelelő lekezelése után a hibázó program futása folytatható.

A kivételek 3 típusba sorolhatók: vannak hibák (**fault**) és csapdák (**trap**). (A 80286-os processzoron egy harmadik kategória is megjelent, ezek az ú.n. **abort**-ok.) A két fajta kivétel között az a különbség, hogy míg a fault-ok bekövetkeztekor a verembe mentett CS:IP érték a hibázó utasításra mutat, addig a trap-ek esetén a hibát kiváltó utasítást követő utasítás címét tartalmazza CS:IP veremben levő másolata. (Az abort kivételek esetén a CS:IP-másolat értéke általában meghatározatlan.) Ezenkívül a trap-ek (mint nevük is mutatja) a program hibamentesítését, debuggolását támogatják, míg a fault és abort típusú kivételek a kritikus hibákat jelzik, és azonnali cselekvésre szólítanak fel.

Most pedig lássuk, milyen kivételek is vannak, és ezek mely feltételek hatására jönnek létre (16.1. táblázat)!

A 0-ás kivétel az osztáskor megtörténő hibák során keletkezik. A DIV és az IDIV utasítások válthatják ki, és alapvetően két oka van: az egyik, hogy nullával próbáltunk meg osztani, ez az eset általában a hibás algoritmusokban fordul elő. A másik ok viszont gyakoribb: akkor is ezt a kivételt kapjuk, ha az osztás hányadosa nem fér el a célban. Például ha AX = 0100h, BL = 01h, és kiadjuk a DIV BL / IDIV BL utasítások valamelyikét, akkor a hányados is 0100h lenne, ez viszont nem fér el AL-ben (hiszen a hányadost ott kapnánk meg, a maradékot pedig AH-ban).

16.1. táblázat. A 8086-os processzoron létező kivételek

Szám	*Mnemo*	*Név*	*Típus*	*Kiváltó feltételek*
0	#DE	Divide Error	Trap	DIV és IDIV utasítások
1	#DB	Debug	Trap	INT 01h utasítás/TF = 1
3	#BP	Breakpoint	Trap	INT 3 utasítás
4	#OF	Overflow	Trap	INTO utasítás, ha OF = 1

Ennek eredménye a 0-ás kivétel lesz. Ezt a kivételt alapesetben a DOS kezeli le, egy hibaüzenet kiírása után egyszerűen terminálja az éppen futó programot, majd visszaadja a vezérlést a szülő alkalmazásnak (COMMAND.COM, valamilyen shell program, commander stb.).

Megjegyezzük, hogy a 0-ás kivételt a 80286-os processzortól kezdődően hibaként (fault) kezelik.

Az 1-es kivétel keletkezésének két oka lehet: vagy kiadtunk egy INT 01h utasítást (kódja 0CDh 01h), vagy bekapcsoltuk a TF flag-et.

3-ast kivételt az INT 3 utasítás tud okozni. Az INT utasításnak erre a megszakítás-számra két alakja is van: az egyik a megszokott kétbájtos forma (0CDh 03h), a másik pedig csak egy bájtot foglal el (0CCh). A két kódolás nem teljesen azonos működést vált ki, ez azonban csak a 80386-os vagy annál újabb processzorokon létező virtuális-8086 üzemmódban nyilvánul meg.

4-es kivétel akkor keletkezik, ha OF = 1, és kiadunk egy INTO utasítást.

Bár nem kivétel, említést érdemel egy korábban még be nem mutatott megszakítás. Az **NMI** (NonMaskable Interrupt) olyan hardver-megszakítás, ami a megszakítás-vezérlőt kikerülve közvetlenül a processzor egyik lábán (érintkezőjén) keresztül jut el a központi egységhez, és amint neve is mutatja, nem tiltható le. A CPU az IF flag állásától függetlenül mindig ki fogja szolgálni ezt a megszakítást, mégpedig az INT 02h kezelőt meghívva. Mivel NMI csak valamilyen hardverhiba folytán keletkezik (pl. memória-paritáshiba, nem megfelelő tápfeszültség stb.), lekezelése után a számítógép nincs biztonságos állapotban a folytatáshoz, ezért az NMI-kezelőben általában hibaüzenet kiírása (esetleg hangjelzés adása) után leállítják a működést. Ez utóbbit pl. úgy érik el, hogy kiadnak egy CLI utasítást, majd egy végtelen ciklusba engedik a processzort (ez a legegyszerűbben egy önmagára mutató JMP utasítást jelent). Másik megoldás a CLI után a HLT (HaLT) operandus nélküli utasítás kiadása lehet, aminek hatására „álomba szenderül” a processzor, és onnan csak egy hardver-megszakítás, az NMI vagy a reset „ébreszti fel.”

A dologban az a vicces, hogy az NMI keletkezését le lehet tiltani az alaplap egy bizonyos portjára írva. (XT-k és PC esetén a 00A0h porton a legfelső bit 0-ás értéke, míg AT-knál a 0070h port ugyanazon bitjének 1-es értéke tiltja le az NMI-t.) Ez elég kockázatos dolog, hiszen ha egyszer NMI-t észlel az alaplap, és nem képes azt a CPU tudtára hozni, akkor a felhasználó mit sem sejtve folytatja munkáját, közben pedig lehet, hogy több adatot veszít el így, mintha az NMI-t látva resetelné vagy kikapcsolná a gépet.

Megjegyezzük, hogy a matematikai koprocesszor (FPU) is generálhat NMI-t. Ebben az esetben az NMI keletkezése nem hardverhibát, hanem a számítások közben előforduló numerikus hibákat (kivételeket) jelzi. Az NMI-kezelőnek a dolga, hogy eldöntse, le kell-e állítani a számítógépet avagy nem.

A. Függelék

Átváltás különféle számrendszerek között

A 0 – 255 értékek bináris, oktális, decimális és hexadecimális számrendszerek közötti átváltását könnyíti meg az A.1. táblázat.

A.1. táblázat. Átváltás a 2-es, 8-as, 10-es és 16-os számrendszerek között

Dec	*Bin*	*Oct*	*Hex*	*Dec*	*Bin*	*Oct*	*Hex*
0	0000 0000b	000o	00h	1	0000 0001b	001o	01h
2	0000 0010b	002o	02h	3	0000 0011b	003o	03h
4	0000 0100b	004o	04h	5	0000 0101b	005o	05h
6	0000 0110b	006o	06h	7	0000 0111b	007o	07h
8	0000 1000b	010o	08h	9	0000 1001b	011o	09h
10	0000 1010b	012o	0Ah	11	0000 1011b	013o	0Bh
12	0000 1100b	014o	0Ch	13	0000 1101b	015o	0Dh
14	0000 1110b	016o	0Eh	15	0000 1111b	017o	0Fh
16	0001 0000b	020o	10h	17	0001 0001b	021o	11h
18	0001 0010b	022o	12h	19	0001 0011b	023o	13h
20	0001 0100b	024o	14h	21	0001 0101b	025o	15h
22	0001 0110b	026o	16h	23	0001 0111b	027o	17h
24	0001 1000b	030o	18h	25	0001 1001b	031o	19h
26	0001 1010b	032o	1Ah	27	0001 1011b	033o	1Bh
28	0001 1100b	034o	1Ch	29	0001 1101b	035o	1Dh
30	0001 1110b	036o	1Eh	31	0001 1111b	037o	1Fh
32	0010 0000b	040o	20h	33	0010 0001b	041o	21h
34	0010 0010b	042o	22h	35	0010 0011b	043o	23h
36	0010 0100b	044o	24h	37	0010 0101b	045o	25h
38	0010 0110b	046o	26h	39	0010 0111b	047o	27h
40	0010 1000b	050o	28h	41	0010 1001b	051o	29h
42	0010 1010b	052o	2Ah	43	0010 1011b	053o	2Bh
44	0010 1100b	054o	2Ch	45	0010 1101b	055o	2Dh
Dec	*Bin*	*Oct*	*Hex*	*Dec*	*Bin*	*Oct*	*Hex*

A.1. táblázat. (folytatás)

Dec	*Bin*	*Oct*	*Hex*	*Dec*	*Bin*	*Oct*	*Hex*
46	0010 1110b	056o	2Eh	47	0010 1111b	057o	2Fh
48	0011 0000b	060o	30h	49	0011 0001b	061o	31h
50	0011 0010b	062o	32h	51	0011 0011b	063o	33h
52	0011 0100b	064o	34h	53	0011 0101b	065o	35h
54	0011 0110b	066o	36h	55	0011 0111b	067o	37h
56	0011 1000b	070o	38h	57	0011 1001b	071o	39h
58	0011 1010b	072o	3Ah	59	0011 1011b	073o	3Bh
60	0011 1100b	074o	3Ch	61	0011 1101b	075o	3Dh
62	0011 1110b	076o	3Eh	63	0011 1111b	077o	3Fh
64	0100 0000b	100o	40h	65	0100 0001b	101o	41h
66	0100 0010b	102o	42h	67	0100 0011b	103o	43h
68	0100 0100b	104o	44h	69	0100 0101b	105o	45h
70	0100 0110b	106o	46h	71	0100 0111b	107o	47h
72	0100 1000b	110o	48h	73	0100 1001b	111o	49h
74	0100 1010b	112o	4Ah	75	0100 1011b	113o	4Bh
76	0100 1100b	114o	4Ch	77	0100 1101b	115o	4Dh
78	0100 1110b	116o	4Eh	79	0100 1111b	117o	4Fh
80	0101 0000b	120o	50h	81	0101 0001b	121o	51h
82	0101 0010b	122o	52h	83	0101 0011b	123o	53h
84	0101 0100b	124o	54h	85	0101 0101b	125o	55h
86	0101 0110b	126o	56h	87	0101 0111b	127o	57h
88	0101 1000b	130o	58h	89	0101 1001b	131o	59h
90	0101 1010b	132o	5Ah	91	0101 1011b	133o	5Bh
92	0101 1100b	134o	5Ch	93	0101 1101b	135o	5Dh
94	0101 1110b	136o	5Eh	95	0101 1111b	137o	5Fh
96	0110 0000b	140o	60h	97	0110 0001b	141o	61h
98	0110 0010b	142o	62h	99	0110 0011b	143o	63h
100	0110 0100b	144o	64h	101	0110 0101b	145o	65h
102	0110 0110b	146o	66h	103	0110 0111b	147o	67h
104	0110 1000b	150o	68h	105	0110 1001b	151o	69h
106	0110 1010b	152o	6Ah	107	0110 1011b	153o	6Bh
108	0110 1100b	154o	6Ch	109	0110 1101b	155o	6Dh
110	0110 1110b	156o	6Eh	111	0110 1111b	157o	6Fh
112	0111 0000b	160o	70h	113	0111 0001b	161o	71h
114	0111 0010b	162o	72h	115	0111 0011b	163o	73h
116	0111 0100b	164o	74h	117	0111 0101b	165o	75h
118	0111 0110b	166o	76h	119	0111 0111b	167o	77h
120	0111 1000b	170o	78h	121	0111 1001b	171o	79h
122	0111 1010b	172o	7Ah	123	0111 1011b	173o	7Bh
124	0111 1100b	174o	7Ch	125	0111 1101b	175o	7Dh
126	0111 1110b	176o	7Eh	127	0111 1111b	177o	7Fh
128	1000 0000b	200o	80h	129	1000 0001b	201o	81h
130	1000 0010b	202o	82h	131	1000 0011b	203o	83h
132	1000 0100b	204o	84h	133	1000 0101b	205o	85h
Dec	*Bin*	*Oct*	*Hex*	*Dec*	*Bin*	*Oct*	*Hex*

A.1. táblázat. (folytatás)

Dec	*Bin*	*Oct*	*Hex*	*Dec*	*Bin*	*Oct*	*Hex*
134	1000 0110b	206o	86h	135	1000 0111b	207o	87h
136	1000 1000b	210o	88h	137	1000 1001b	211o	89h
138	1000 1010b	212o	8Ah	139	1000 1011b	213o	8Bh
140	1000 1100b	214o	8Ch	141	1000 1101b	215o	8Dh
142	1000 1110b	216o	8Eh	143	1000 1111b	217o	8Fh
144	1001 0000b	220o	90h	145	1001 0001b	221o	91h
146	1001 0010b	222o	92h	147	1001 0011b	223o	93h
148	1001 0100b	224o	94h	149	1001 0101b	225o	95h
150	1001 0110b	226o	96h	151	1001 0111b	227o	97h
152	1001 1000b	230o	98h	153	1001 1001b	231o	99h
154	1001 1010b	232o	9Ah	155	1001 1011b	233o	9Bh
156	1001 1100b	234o	9Ch	157	1001 1101b	235o	9Dh
158	1001 1110b	236o	9Eh	159	1001 1111b	237o	9Fh
160	1010 0000b	240o	0A0h	161	1010 0001b	241o	0A1h
162	1010 0010b	242o	0A2h	163	1010 0011b	243o	0A3h
164	1010 0100b	244o	0A4h	165	1010 0101b	245o	0A5h
166	1010 0110b	246o	0A6h	167	1010 0111b	247o	0A7h
168	1010 1000b	250o	0A8h	169	1010 1001b	251o	0A9h
170	1010 1010b	252o	0AAh	171	1010 1011b	253o	0ABh
172	1010 1100b	254o	0ACh	173	1010 1101b	255o	0ADh
174	1010 1110b	256o	0AEh	175	1010 1111b	257o	0AFh
176	1011 0000b	260o	0B0h	177	1011 0001b	261o	0B1h
178	1011 0010b	262o	0B2h	179	1011 0011b	263o	0B3h
180	1011 0100b	264o	0B4h	181	1011 0101b	265o	0B5h
182	1011 0110b	266o	0B6h	183	1011 0111b	267o	0B7h
184	1011 1000b	270o	0B8h	185	1011 1001b	271o	0B9h
186	1011 1010b	272o	0BAh	187	1011 1011b	273o	0BBh
188	1011 1100b	274o	0BCh	189	1011 1101b	275o	0BDh
190	1011 1110b	276o	0BEh	191	1011 1111b	277o	0BFh
192	1100 0000b	300o	0C0h	193	1100 0001b	301o	0C1h
194	1100 0010b	302o	0C2h	195	1100 0011b	303o	0C3h
196	1100 0100b	304o	0C4h	197	1100 0101b	305o	0C5h
198	1100 0110b	306o	0C6h	199	1100 0111b	307o	0C7h
200	1100 1000b	310o	0C8h	201	1100 1001b	311o	0C9h
202	1100 1010b	312o	0CAh	203	1100 1011b	313o	0CBh
204	1100 1100b	314o	0CCh	205	1100 1101b	315o	0CDh
206	1100 1110b	316o	0CEh	207	1100 1111b	317o	0CFh
208	1101 0000b	320o	0D0h	209	1101 0001b	321o	0D1h
210	1101 0010b	322o	0D2h	211	1101 0011b	323o	0D3h
212	1101 0100b	324o	0D4h	213	1101 0101b	325o	0D5h
214	1101 0110b	326o	0D6h	215	1101 0111b	327o	0D7h
216	1101 1000b	330o	0D8h	217	1101 1001b	331o	0D9h
218	1101 1010b	332o	0DAh	219	1101 1011b	333o	0DBh
220	1101 1100b	334o	0DCh	221	1101 1101b	335o	0DDh
Dec	*Bin*	*Oct*	*Hex*	*Dec*	*Bin*	*Oct*	*Hex*

A.1. táblázat. (folytatás)

Dec	*Bin*	*Oct*	*Hex*	*Dec*	*Bin*	*Oct*	*Hex*
222	1101 1110b	336o	0DEh	223	1101 1111b	337o	0DFh
224	1110 0000b	340o	0E0h	225	1110 0001b	341o	0E1h
226	1110 0010b	342o	0E2h	227	1110 0011b	343o	0E3h
228	1110 0100b	344o	0E4h	229	1110 0101b	345o	0E5h
230	1110 0110b	346o	0E6h	231	1110 0111b	347o	0E7h
232	1110 1000b	350o	0E8h	233	1110 1001b	351o	0E9h
234	1110 1010b	352o	0EAh	235	1110 1011b	353o	0EBh
236	1110 1100b	354o	0ECh	237	1110 1101b	355o	0EDh
238	1110 1110b	356o	0EEh	239	1110 1111b	357o	0EFh
240	1111 0000b	360o	0F0h	241	1111 0001b	361o	0F1h
242	1111 0010b	362o	0F2h	243	1111 0011b	363o	0F3h
244	1111 0100b	364o	0F4h	245	1111 0101b	365o	0F5h
246	1111 0110b	366o	0F6h	247	1111 0111b	367o	0F7h
248	1111 1000b	370o	0F8h	249	1111 1001b	371o	0F9h
250	1111 1010b	372o	0FAh	251	1111 1011b	373o	0FBh
252	1111 1100b	374o	0FCh	253	1111 1101b	375o	0FDh
254	1111 1110b	376o	0FEh	255	1111 1111b	377o	0FFh
Dec	*Bin*	*Oct*	*Hex*	*Dec*	*Bin*	*Oct*	*Hex*

B. Függelék

Karakter kódtáblázatok

A karakterek kódolására két szabvány terjedt el. Ezek az **ASCII** (American Standard Code for Information Interchange) és az **EBCDIC** (Extended Binary-Coded Decimal Interchange Code) nevet kapták. Az ASCII eredetileg 7 bites volt (tehát csak 128 karaktert tartalmazott), később bővítették csak ki 8 bitesre. A felső 128 pozícióban általában különféle nyelvek betűit és speciális jeleket tárolnak. Mi itt az ú.n. Latin-2 (más néven ISO 8859-2, 1250-es kódlap) kiosztást mutatjuk be, ez tartalmazza ugyanis az összes közép európai nyelv (köztük a magyar) betűit is.

Mindkét szabvány esetén néhány karakternek különleges funkciója van. Ezeket számos periféria (pl. modem, nyomtató, terminál) vezérlési célokra használja. Az ilyen karaktereknek általában nincs nyomtatható (megjeleníthető) képe. Ehelyett egy néhány betűs név utal szerepükre. Az ismert rövidítések jelentését mutatja a B.1. táblázat.

B.1. táblázat. ASCII és EBCDIC vezérlőkódok

Név	*Jelentés*
ACK	ACKnowledge
BEL	BELl
BS	BackSpace
CAN	CANcel
CR	Carriage Return
DC1	Device Control 1 (X-ON)
DC2	Device Control 2
DC3	Device Control 3 (X-OFF)
DC4	Device Control 4
DEL	DELete
DLE	Data Line Escape
EM	End of Medium
ENQ	ENQuiry
EOT	End Of Transmission
ESC	ESCape
ETB	End of Transmission Block
ETX	End of TeXt
FS	File Separator
FF	Form Feed

B.1. táblázat. (folytatás)

Név	*Jelentés*
GS	Group Separator
HT	Horizontal Tabulator
LF	Line Feed
NAK	Negative AcKnowledge
NUL	NULl (end of string)
RS	Record Separator
SI	Shift In
SO	Shift Out
SOH	Start Of Heading
SP	SPace
STX	Start of TeXt
SUB	SUBstitute
SYN	SYNchronous idle
US	Unit Separator
VT	Vertical Tabulator

A B.2. táblázat tartalmazza az említett két szabvány beosztását. Az üresen hagyott helyek funkciója ismeretlen.

B.2. táblázat. ASCII és EBCDIC karakterkódok

Kód	*ASCII*	*EBCDIC*	*Kód*	*ASCII*	*EBCDIC*
0	NUL	NUL	1	SOH	SOH
2	STX	STX	3	ETX	ETX
4	EOT	PF	5	ENQ	HT
6	ACK	LC	7	BEL	DEL
8	BS		9	HT	
10	LF	SMM	11	VT	VT
12	FF	FF	13	CR	CR
14	SO	SO	15	SI	SI
16	DLE	DLE	17	DC1	DC1
18	DC2	DC2	19	DC3	TM
20	DC4	RES	21	NAK	NL
22	SYN	BS	23	ETB	IL
24	CAN	CAN	25	EM	EM
26	SUB	CC	27	ESC	CU1
28	FS	IFS	29	GS	IGS
30	RS	IRS	31	US	IUS
32	SP [1]	DS	33	!	SOS
34	"	FS	35	#	
36	$	BYP	37	%	LF
38	&	ETB	39	’	ESC
Kód	*ASCII*	*EBCDIC*	*Kód*	*ASCII*	*EBCDIC*

[1] Látható szóköz.

B.2. táblázat. (folytatás)

Kód	*ASCII*	*EBCDIC*	*Kód*	*ASCII*	*EBCDIC*
40	(		41	)	
42	*	SM	43	+	CU2
44	,		45	–	ENQ
46	.	ACK	47	/	BEL
48	0		49	1	
50	2	SYN	51	3	
52	4	PN	53	5	RS
54	6	UC	55	7	EOT
56	8		57	9	
58	:		59	;	CU3
60	<	DC4	61	=	NAK
62	>		63	?	SUB
64	@	SP	65	A	
66	B		67	C	
68	D		69	E	
70	F		71	G	
72	H		73	I	
74	J	¢	75	K	.
76	L	<	77	M	{
78	N	+	79	O	¦
80	P	&	81	Q	
82	R		83	S	
84	T		85	U	
86	V		87	W	
88	X		89	Y	
90	Z	!	91	[	$
92	\	*	93	]	)
94	ˆ	;	95	_	
96	‘		97	a	/
98	b		99	c	
100	d		101	e	
102	f		103	g	
104	h		105	i	
106	j		107	k	,
108	l	%	109	m	_
110	n	>	111	o	?
112	p		113	q	
114	r		115	s	
116	t		117	u	
118	v		119	w	
120	x		121	y	
122	z	:	123	{	#
124	¦	@	125	}	’
126	˜	=	127	DEL	"
Kód	*ASCII*	*EBCDIC*	*Kód*	*ASCII*	*EBCDIC*

B.2. táblázat. (folytatás)

Kód	*ASCII*	*EBCDIC*	*Kód*	*ASCII*	*EBCDIC*
128			129		a
130		b	131		c
132		d	133		e
134		f	135		g
136		h	137		i
138			139		
140			141		
142			143		
144	ı[2]		145	` [3]	j
146		k	147	ˆ [4]	l
148	˜ [5]	m	149		n
150	˘ [6]	o	151	˙ [7]	p
152		q	153		r
154	˚ [8]		155		
156			157	˝ [9]	
158	[10]		159	ˇ [11]	
160	[12]		161	Ą	
162	˘ [13]	s	163	Ł	t
164	¤	u	165	Ľ	v
166	Ś	w	167	§	x
168	¨ [14]	y	169	Š	z
170	Ş		171	Ť	
172	Ź		173	[15]	
174	Ž		175	Ż	
176	° [16]		177	ą	
178	[17]		179	ł	
180	´ [18]		181	ľ	
182	ś		183	ˇ [19]	
Kód	*ASCII*	*EBCDIC*	*Kód*	*ASCII*	*EBCDIC*

[2] Pontnélküli „i”.
[3] „Grave” ékezet.
[4] „Circumflex” vagy „hat” ékezet.
[5] „Tilde” ékezet.
[6] „Breve” ékezet.
[7] „Dot” ékezet.
[8] „Ring” ékezet.
[9] „Double acute” vagy „hungarian umlaut” ékezet.
[10] „Ogonek” ékezet.
[11] „Haček” vagy „caron” ékezet.
[12] Nemtörhető szóköz.
[13] „Breve” ékezet.
[14] „Dieresis” vagy „umlaut” ékezet.
[15] Feltételes kötőjel (soft hyphen).
[16] Fokjel.
[17] „Ogonek” ékezet.
[18] „Acute” ékezet.
[19] „Haček” vagy „caron” ékezet.

B.2. táblázat. (folytatás)

Kód	*ASCII*	*EBCDIC*	*Kód*	*ASCII*	*EBCDIC*
184	¸[20]		185	š	
186	ş		187	ť	
188	ź		189	˝[21]	
190	ž		191	ż	
192	Ŕ		193	Á	A
194	Â	B	195	Ă	C
196	Ä	D	197	Ĺ	E
198	Ć	F	199	Ç	G
200	Č	H	201	É	I
202	Ę		203	Ë	
204	Ě		205	Í	
206	Î		207	Ď	
208	Đ		209	Ń	J
210	Ň	K	211	Ó	L
212	Ô	M	213	Ő	N
214	Ö	O	215	×	P
216	Ř	Q	217	Ů	R
218	Ú		219	Ű	
220	Ü		221	Ý	
222	Ţ		223	ß	
224	ŕ		225	á	
226	â	S	227	ă	T
228	ä	U	229	ĺ	V
230	ć	W	231	ç	X
232	č	Y	233	é	Z
234	ę		235	ë	
236	ě		237	í	
238	î		239	ď	
240	đ	0	241	ń	1
242	ň	2	243	ó	3
244	ô	4	245	ő	5
246	ö	6	247	÷	7
248	ř	8	249	ů	9
250	ú		251	ű	
252	ü		253	ý	
254	ţ		255	˙[22]	
Kód	*ASCII*	*EBCDIC*	*Kód*	*ASCII*	*EBCDIC*

[20]„Cedilla” ékezet.
[21]„Double acute” vagy „hungarian umlaut” ékezet.
[22]„Dot” ékezet.

Tárgymutató

A, Á

abort-class exception 114
acknowledgement . . . . . *lásd* nyugtázás
adatmozgató utasítások 30
address space . . . . . *lásd* címterület
    wrap-around . . . . . *lásd* címterület-körbefordulás
addressing mode . . . . . *lásd* címzési mód
alaplap 5
áloperandus 75
általános célú adatregiszter 14
általános célú regiszterek 17
antivalencia 8
aritmetikai flag 17
aritmetikai shiftelés 62
ASCII kódtábla 120
ASCIIZ sztring 98
assembler 24
átvitel 12
    borrow 12
    carry 12

B

backlink . . . . . *lásd* visszaláncolás
bájt 8
BCD aritmetikai utasítások 32
BCD szám . . . . . 17, *lásd* binárisan kódolt decimális egész számok
    pakolatlan 17
    pakolt 17
belépési pont 89
big-endian tárolásmód 8
bináris
    művelet 7
    számjegy . . . . . *lásd* bit
    számrendszer 7
binárisan kódolt decimális egész számok 17
BIOS 5
bit 7
    beállítása 65
    invertálása 65
    tesztelése 65
    törlése 65
bitléptető utasítások 32
bitmaszk 65
borrow 12
busz 5
buszlezáró prefix 30
byte . . . . . *lásd* bájt

C

carry 12
céloperandus 26
címke 25
címterület 13
    -körbefordulás 14
címzés
    bázis 18
    bázis+index 18
    bázis+index+relatív 18
    bázis+relatív, bázisrelatív 18
    index 18
    index+relatív, indexrelatív 18
    közvetlen 18
címzési mód 18
CPU . . . . . 5, *lásd* mikroprocesszor

D

decimális számrendszer 7
direkt ugrás 105
direktíva 24
    .STACK 38
    ASSUME 38
    DB 38
    DD 38
    DF 38
    DOSSEG 88
    DQ 38

DT . . . 38
DW . . . 38
END . . . 40
ENDP . . . 50
ENDS . . . 38
EQU . . . 41
JUMPS . . . 46
LABEL . . . 88
MODEL . . . 38
NOJUMPS . . . 46
ORG . . . 85
PROC . . . 50
SEGMENT . . . 38
disassemblálás . . . 52
displacement . . . *lásd* eltolás
diszjunkció . . . 8
DMA . . . 6
-vezérlő . . . 6
dollárjel . . . 25
doubleword . . . *lásd* duplaszó
duplaszó . . . 8

E, É

EBCDIC kódtábla . . . 120
effektív cím . . . *lásd* tényleges cím
egész aritmetikai utasítások . . . 31
egyes komplemens alak . . . 9
egyszerűsített szegmensdefiníció . . . 27
eljárás . . . 67
eljáráshívás . . . 49
előjel . . . 9
-bit . . . 9
előjeles kiterjesztés . . . 12
előjeles számok . . . 9
előjeltelen kiterjesztés . . . 12
előjeltelen számok . . . 8
eltolás . . . 18
endianizmus . . . 8
big-endian tárolásmód . . . 8
little-endian tárolásmód . . . 8
entry point . . . *lásd* belépési pont
exception . . . *lásd* kivétel
abort . . . 114
fault . . . 114
trap . . . 114

F

fault-class exception . . . 114
fejléc . . . 89
feltételes ugrás . . . 45
file handle . . . 98
fizikai memóriacím . . . 14
flag . . . 15
aritmetikai . . . 17
Flags regiszter . . . 14
forgatás . . . *lásd* rotálás
forrásoperandus . . . 26
frissítés . . . 6
function . . . *lásd* függvény
függvény . . . 67

G

gépi kód . . . 2
giga- . . . 8

H

hardver-megszakítás . . . 23
header . . . *lásd* fejléc
hexadecimális számrendszer . . . 7

I, Í

időzítő . . . 6
indexregiszter . . . 14
indirekt ugrás . . . 105
init . . . 5
instruction . . . *lásd* utasítás
interrupt . . . *lásd* megszakítás

K

karakteres konstans . . . 25
kettes komplemens alak . . . 9
kilo- . . . 8
kivétel . . . 23, 114
konjunkció . . . 7
koprocesszor-vezérlő utasítások . . . 33
közeli ugrás . . . 105
közvetlen memória-hozzáférés . *lásd* DMA
kvadraszó . . . 8

L

lefelé bővülő verem . . . 20
léptetés . . . *lásd* shiftelés
LIFO elv . . . 20
lineáris memóriacím . . . 13
lineáris memória-modell . . . 13
linker . . . *lásd* szerkesztő
little-endian tárolásmód . . . 8

logikai memóriacím 13
logikai művelet 7
    ÉS . . . . *lásd* konjunkció
    KIZÁRÓ VAGY . . . . *lásd* antivalencia
    MEGENGEDŐ VAGY . . . . *lásd* diszjunkció
    tagadás . . . . *lásd* negáció
logikai shiftelés 62
logikai utasítások 31

M

machine code 2
MASM . . *lásd* Microsoft Macro Assembler
MCB . . . . *lásd* memóriavezérlő blokk
mega- 8
megjegyzés 26
megszakítás . . . . *lásd* megszakítás-kérelem
    hardver- 23
    -kérelem 5
    -kezelő 23, 102
    -maszkolás 23
    -rendszer 5
    szoftver- 23
    -vektor 102
    -vezérlő 5
    -vonal 5
memória 5
    RAM 5
    ROM 5
memóriacím
    fizikai 14
    lineáris 13
    logikai 13
    normált 14
memória-modell 13, 27
    lineáris 13
    szegmentált 13
memória-szervezés . . *lásd* memória-modell
memóriavezérlő blokk 81
Microsoft Macro Assembler 28
mikroprocesszor 4
mnemonik 2, 25
mutató 17
    közeli-, rövid- 17
    távoli-, hosszú-, teljes- 17
mutatóregiszter 14
műveleti kód 25

N

negáció 7
nibble 8
NMI 115
normált memóriacím 14
numerikus konstans 25

Ny

nyugtázás 110

O, Ó

object code . . . . *lásd* tárgykód
offszetcím 13
oktális számrendszer 7
opcode . . . . *lásd* műveleti kód
operandus 25
operátor 25
    DUP 41
    OFFSET 79
    PTR 77
    SHORT 110
órajel 5
    -generátor 6

P

paragrafus 13
paragrafus-határ 13
periféria 5
pointer . . . . *lásd* mutató
    far, long, full . . . *lásd* távoli-, hosszú-, teljes mutató
    near, short . *lásd* közeli-, rövid mutató
port 5
pozíció-számláló . . . . *lásd* dollárjel
prefix 25
    buszlezáró 30
    CS: 29
    DS: 29
    ES: 29
    LOCK 30
    REP, REPE, REPZ 30, 78
    REPNE, REPNZ 30, 78
    SEGCS 29
    SEGDS 29
    SEGES 29
    SEGSS 29
    SS: 29
    szegmensfelülbíráló 19, 29
    sztringutasítást ismétlő 30
procedure . . . . *lásd* eljárás

programming language 1
high-level 1
low-level 1
programozási nyelv 1
alacsony szintű 1
magas szintű 1
programszegmens 84
programszegmens-prefix 85
PSP . . . . *lásd* programszegmens-prefix

Q
quadword . . . . *lásd* kvadraszó

R
RAM 5
regiszter 5, 14
általános célú 14, 17
Flags 14
index- 14
mutató- 14
státusz- 14
szegmens- 15
regiszterpár 17
relokáció 86
relokációs tábla 89
rendszervezérlő utasítások 33
reset 5
rezidens program 111
ROM 5
rotálás 63
rövid ugrás 110

S
scan code 92
shiftelés 12
aritmetikai . 62, *lásd* előjeles shiftelés
előjeles 12
előjeltelen 12
logikai . . 62, *lásd* előjeltelen shiftelés
sign . . . . *lásd* előjel
sign extension . . . . *lásd* előjeles kiterjesztés
speciális utasítások 33
stack . . . . *lásd* verem
státuszregiszter 14

Sz
szám
binárisan kódolt decimális egész . . 17
előjeles 9
előjeltelen 8
pakolatlan BCD 17
pakolt BCD 17
számrendszer
bináris 7
decimális 7
hexadecimális 7
oktális 7
szegmens 13, 26
szegmens báziscím 13
szegmenscím . . . . . *lásd* szegmens báziscím
szegmensen belüli ugrás . *lásd* közeli ugrás
szegmensfelülbíráló prefix 19, 29
szegmensközi ugrás . . . . . *lásd* távoli ugrás
szegmensregiszter 15
szegmentált memória-modell 13
szerkesztő 24
szimbólum 25
szó 8
szoftver-megszakítás 23
sztring 17
sztringkezelő utasítások 32
sztringutasítást ismétlő prefixek 30

T
tárgykód 24
TASM . . . . *lásd* Turbo Assembler
távoli ugrás 105
TD . . . . *lásd* Turbo Debugger
tényleges cím 20
TLINK . . . . *lásd* Turbo Linker
töréspont 53
trap-class exception 114
TSR . . . . *lásd* rezidens program
túlcsordulás 12
Turbo Assembler 28
Turbo Debugger 52
Turbo Linker 28

U, Ú
ugrás
közeli 105
rövid 110
távoli 105
ugrótábla 102
új sor karakterpár 51
unáris művelet 7
utasítás 25

AAA . . . 60
AAD . . . 61
AAM . . . 61
AAS . . . 61
adatmozgató . . . 30
ADC . . . 57
ADD . . . 39
AND . . . 65
BCD aritmetikai . . . 32
bitléptető . . . 32
CALL . . . 49, 105
CBW . . . 42
CLC . . . 66
CLD . . . 44, 66
CLI . . . 66
CMC . . . 66
CMP . . . 47
CMPS, CMPSB, CMPSW . . . 77
CWD . . . 57
DAA . . . 61
DAS . . . 62
DEC . . . 42
DIV . . . 50, 58
egész aritmetikai . . . 31
HLT . . . 115
IDIV . . . 58
IMUL . . . 58
IN . . . 110
INC . . . 42
INT . . . 40
Jccc (feltételes ugrások) . . . 45
JMP . . . 47, 105
JMPS . . . 110
koprocesszor-vezérlő . . . 33
LAHF . . . 66
LDS . . . 113
LEA . . . 42
LES . . . 113
LODS, LODSB, LODSW . . . 44, 79
logikai . . . 31
LOOP . . . 42
LOOPE, LOOPZ . . . 43
LOOPNE, LOOPNZ . . . 43
MOV . . . 39
MOVS, MOVSB, MOVSW . . . 77
MUL . . . 58
NEG . . . 59
NOP . . . 95
NOT . . . 65
OR . . . 44, 65
OUT . . . 110
POP . . . 20, 44
POPF . . . 66
PUSH . . . 20, 44
PUSHF . . . 66
RCL . . . 63
RCR . . . 63
rendszervezérlő . . . 33
RET . . . 51
ROL . . . 63
ROR . . . 63
SAHF . . . 66
SAL . . . 62
SAR . . . 62
SBB . . . 57
SCAS, SCASB, SCASW . . . 79
SHL . . . 62
SHR . . . 62
speciális . . . 33
STC . . . 66
STD . . . 66
STI . . . 66
STOS, STOSB, STOSW . . . 47, 79
SUB . . . 39
sztringkezelő . . . 32
TEST . . . 65
vezérlésátadó . . . 32
XCHG . . . 95
XLAT, XLATB . . . 95
XOR . . . 39, 65

V

verem . . . 20
vezérlésátadó utasítások . . . 32
visszaláncolás . . . 85

W

word . . . *lásd* szó

Z

zero extension . . . *lásd* zéró-kiterjesztés
zéró-kiterjesztés . . . *lásd* előjeltelen kiterjesztés

Irodalomjegyzék

[1] Dr. Kovács Magda, *32 Bites Mikroprocesszorok 80386/80486 I./II.*, LSI, Budapest, **[könyv]**

[2] Abonyi Zsolt, *PC Hardver Kézikönyv*, **[könyv]**

[3] Dan Rollins, *Tech Help! v4.0*, Copyright © Flambeaux Software, Inc., 1985, 1990, **[program]**

[4] David Jurgens, *HelpPC v2.10*, Copyright © 1991, **[program]**

[5] *Expert Help Hypertext System v1.09*, Copyright © SofSolutions, 1990, 1992, **[program]**

[6] Ralf Brown, *Ralf Brown's Interrupt List Release 61*, Copyright © 1989, 2000, **[program, txt]**

[7] Nobody, *The Interrupts and Ports database v1.01*, Copyright © 1988, **[ng]**

[8] Morten Elling, *Borland's Turbo Assembler v4.0 Ideal mode syntax*, Copyright © 1995, **[ng]**

[9] *The Assembly Language Database*, Copyright © Peter Norton Computing, Inc., 1987, **[ng]**

[10] P. H. Rankin Hansen (Ping), *The Programmers Reference v0.02b*, **[ng]**

[11] *Intel iAPx86 Instruction Set*, 1996, **[ng]**

[12] *http://www.pobox.com/~ralf/*, Ralf Brown's Homepage, **[www]**

Ebben a gyűjteményben mind nyomtatott, mind elektronikus formájú művek, illetve segédprogramok megtalálhatók. A bejegyzések utolsó tagja utal a forrás típusára:

- **[könyv]** – Nyomtatásban megjelent könyv
- **[program]** – Segédprogram
- **[ng]** – Norton Guide formátumú (.ng kiterjesztésű) adatbázis
- **[txt]** – Tisztán szöveges formátumú dokumentum
- **[www]** – World Wide Web (Internet) honlap címe