TÁMOP-4.1.1.F-14/1/KONV-2015-0009
„A GÉPÉSZETI ÉS INFORMATIKAI ÁGAZATOK DUÁLIS ÉS MODULÁRIS KÉPZÉSEINEK KIALAKÍTÁSA A PÉCSI TUDOMÁNYEGYETEMEN „

Gimesi László

Intel processzorok programozása assembly nyelven

Pécs
2015

A tananyag a TÁMOP-4.1.1.F-14/1/KONV-2015-0009 azonosító számú,

„A gépészeti és informatikai ágazatok duális és moduláris képzéseinek kialakítása a Pécsi Tudományegyetemen” című projekt keretében valósul meg.

PÉCSI TUDOMÁNYEGYETEM
UNIVERSITY OF PÉCS

SZÉCHENYI 2020
MAGYARORSZÁG KORMÁNYA
Európai Unió
Európai Szociális Alap
BEFEKTETÉS A JÖVŐBE

TÁMOP-4.1.1.F-14/1/KONV-2015-0009
„A GÉPÉSZETI ÉS INFORMATIKAI ÁGAZATOK DUÁLIS ÉS MODULÁRIS KÉPZÉSEINEK KIALAKÍTÁSA A PÉCSI TUDOMÁNYEGYETEMEN „

Intel processzorok programozása assembly nyelven

Gimesi László

Szakmai lektor: Markó Tamás

Nyelvi lektor: Gimesi Lászlóné

ISBN 978-963-642-831-0

Pécsi Tudományegyetem

Természettudományi Kar

Pécs, 2015.

© Gimesi László

PÉCSI TUDOMÁNYEGYETEM
UNIVERSITY OF PÉCS

Európai Unió
Európai Szociális Alap
MAGYARORSZÁG KORMÁNYA
BEFEKTETÉS A JÖVŐBE

Tartalomjegyzék

1. Bevezetés *2*

2. Hardver *3*

2.1. A rendszerbusz (sín) **4**

2.2. CPU (Central Processing Unit) **4**

2.3. Az operatív memória **8**

2.4. Az input/output egységek **10**

3. Bináris aritmetika *12*

3.1. A számrendszerek **12**

3.2. Adatábrázolás **14**

3.3. Az Intel processzorok számábrázolása **17**

3.4. Műveletvégzés **18**

4. Az Intel mikroprocesszorok *27*

4.1. Kezdetek **27**

4.2. Mikroprocesszor **29**

4.3. Az Intel processzorok regiszterkészlete **31**

4.4. A matematikai társprocesszor **35**

5. Az Intel processzorok utasításrendszere *38*

5.1. Utasítások **38**

5.2. Operandusok és címzési módok **38**

5.3. A verem (stack) **42**

5.4. Memóriaszervezés **43**

6. Az assembly programozás *45*

6.1. Szegmentálás **45**

6.2. Kezdjünk el programot írni **46**

6.3. Konstansok használata **47**

6.4. Néhány egyszerűbb szubrutin **48**

6.5. Az adatszegmens használata **58**

6.6. Lemezmeghajtó kezelése **67**

6.7. A karakteres videó-memória kezelése **70**

6.8. Mintapéldák az INT 21h interrupthoz **73**

7. Az assembly kapcsolata más nyelvekkel *78*

7.1. Assembly rutinok Turbo Pascal programokban **78**

7.2. Assembly rutinok C programokban **84**

8. Irodalomjegyzék *94*

9. Melléklet *95*

1. Bevezetés

A tárolt program elvét Neumann János[1] fogalmazta meg, miszerint a programutasítások és az adatok azonos formában (binárisan), ugyanabban a belső memóriában helyezkednek el. Így van ez a legkorszerűbb számítógépnél is. Ez azt jelenti, hogy a számítógépen bármilyen programozási nyelvet is használunk, az eredmény mindig egy bináris – az operatív memóriába tölthető – számsor (utasítássorozat).

Az első számítógépeket gépi kódban programozták, az utasításokat bináris számok formájában kellett a memóriába táplálni. 1946-ban kezdték használni a mnemonikus kódokat[2], majd a szimbolikus nyelveket, amelyekkel nagymértékben megnövelték a programírás hatékonyságát. Azonban az így elkészített utasítássorokat le kellett fordítani gépi kódra. Az első fordító (assembler) program 1949-ben készült el.

A programozás során legtöbbször egy folyamatot írunk le. Elképzeljük, hogy egy adott feladatot hogyan oldanánk meg (milyen lépések sorozatával tudnánk leírni), és azt próbáljuk a számítógéppel elvégeztetni. Egy programozási nyelvnek, és az azt lefordító fordítóprogramnak az a feladata, hogy megteremtse a kapcsolatot az emberi gondolatok és a CPU utasítás-végrehajtása között.

Minél közelebb áll egy programozási nyelv az emberi gondolkodáshoz, minél egyszerűbben lehet leírni benne a feladatot, annál magasabb színtű nyelvről (programozási nyelvről) beszélhetünk. Ekkor a fordítóprogramra hárul a nagyobb feladat, hogy a gépi kódot (futtatható állományt) előállítsa, ez a kód azonban nem lesz optimális. A működéskor felesleges utasításokat is kénytelen a processzor végrehajtani, ami megnöveli a program erőforrásigényét, de manapság – a korszerű, nagysebességű processzorok korában – a programozók nagy része ezzel nem foglalkozik.

Amikor programjaink optimalizálása, sebességének növelése a célunk, assemblyben (alacsony szinten) kell a programunkat elkészíteni. Ekkor a programozóra hárul a nagyobb feladat, hiszen neki kell összeállítani úgy az utasításokat, ahogy a CPU „gondolkodik”.

Vannak olyan esetek, amikor az assembly írásakor nem az optimalizálás a célunk. Ilyen például, amikor egy magas színtű nyelv nincs felkészítve bizonyos feladatok elvégzésére: különböző konverziók, erőforrások és portok kezelése stb.

Ahhoz, hogy az assembly-t hatékonyan tudjuk használni, meg kell ismerni a számítógépek, mikroprocesszorok felépítését, és a bináris aritmetikát. E témákkal foglalkozik a jegyzet első része. Ezt követi az Intel mikroprocesszorok működésének és programozásának ismertetése.

A jegyzet egyszerű példák segítségével, lépésről-lépésre ismerteti az utasításokat és a MASM (Microsoft Macro Assembly) nyelvet, majd foglalkozik azzal is, hogy milyen módon lehet meghívni egy assembly rutint magas színtű programozási nyelvből.

A Függelékben – többek között – összefoglalásra kerül az Intel processzorok alaputasításkészlete és a fontosabb megszakítások (interrupt-ok) kezelése.

[1] Neumann János (John von Neumann) 1903. december 28-án született Budapesten. 1925-ben a Zürich-i Egyetemen vegyészmérnöki diplomát szerzett, majd 1926-ban, Budapesten, matematikából doktorált. 1930-ban meghívták a Princeton Egyetemre, ahol nem sokkal később professzorrá nevezték ki. Fő kutatási területe a matematikai logika, a játékelmélet és a halmazelmélet volt, de foglalkozott fizikai problémák megoldásával is. A II. világháborút követően több számítógépes újítás fűződik a nevéhez.

[2] A programozásban használt, a beszélt nyelvhez hasonló, abban könnyen megjegyezhető kifejezésekket mnemoniknak nevezzük.

2. Hardver

A számítógép fejlődése során – kezdetben – minden egyes számítógép más és más architektúrával rendelkezett. Ez azt jelentette, hogy ahány típusú számítógépet terveztek és építettek, mindegyik a többitől eltérő felépítésű volt, a részegységeket más és más elvek alapján kapcsolták össze. A legtöbb számítógép-struktúra zsákutcának bizonyult, de volt néhány közülük, amely meghatározta a további fejlődést.

Neumann számára – az ENIAC-kal szerzett tapasztalatok alapján – nyilvánvalóvá vált, hogy a számítógép programozása kapcsolók és vezetékek segítségével nagyon körülményes és időigényes feladat. Ez a felismerés vezetett az első programtárolású számítógép megalkotásához, ahol a memóriában az adatokkal együtt az utasításokat is tárolták.

Neumann azt is észrevette, hogy az ENIAC-ban használt decimális aritmetika (minden számjegy ábrázolásához 10 darab elektroncsövet használtak) helyettesíthető bináris aritmetikával.[3]

Azokat a számítógépeket, amelyeket a fenti alapgondolatok alapján építettek, ma Neumann-elvű számítógépeknek nevezzük. Ez az elv még ma is alapja szinte valamennyi digitális számítógépnek. A Neumann-elvű gépnek öt elemi része volt: a memória, az aritmetikai-logikai, a vezérlő, a bemeneti és a kimeneti egység. Ezek memóriaközpontú számítógépek voltak.

Az 1960-as évek elején a DEC (Digital Equipment Corporation) bemutatta a PDP-8-at. E számítógépben alkalmazták először a BUS (BUSZ) rendszert, amely megteremtette a számítógépek bővítésének lehetőségét. Tehát a felhasználó később is – az igényeinek megfelelően – fejleszthette számítógépét, például nagyobb memóriával vagy új perifériaillesztőkkel.

A BUSZ nem más, mint párhuzamos kábelek összessége, amelyeket a számítógép részegységeinek összekötésére használnak. Ez az architektúra olyan jelentős újítás volt (a memóriaközpontú IAS géphez képest), hogy azóta is alkalmazzák majdnem minden kisszámítógépben.

Érdekességként megjegyezzük, hogy már 1964-ben is készítettek olyan számítógépet, a CDC-6600-at, ahol több processzort is használtak. Egyet az összeadáshoz, egy másikat a szorzáshoz, harmadikat az osztáshoz, és külön processzor volt az input/output műveletek elvégzéséhez.

A felsorolt néhány fejlesztés volt az, amelyek alapvetően meghatározták a mai számítógépek felépítését. Ennek megfelelően a számítógép-architektúra egy busz (más néven sín) rendszerre épül, amely biztosítja a kapcsolatot a részegységek között. A rendszerbuszhoz kapcsolódik a központi egység (CPU), a belső (operatív) memória és az input/output egységek (a perifériaillesztők).

[3] Charles Babbage (1791-1871) az 1830-as években megtervezte az „Analytical Engine” mechanikus berendezését, amely már rendelkezett szinte az összes olyan tulajdonsággal, mint egy modern számítógép. Sajnos a kor technológiai szinvonala nem tette lehetővé a gép megépítését.

2.1. A rendszerbusz (sín)

Sínnek nevezzük az azonos típusú adatátvitelre szolgáló vonalak (vezetékek) összességét. Ennek értelmében a sínrendszerű architektúráknál nem is egy, hanem három buszról beszélhetünk, úgymint címbusz (address bus), adatbusz (data bus) és vezérlőbusz (control bus). Ezek együttes neve a rendszerbusz.

A címbusz segítségével kijelölhetjük a szükséges input/output eszközt, vagy a memória tároló rekeszét. Az adatbusz biztosítja az adatáramlást a részegységek között, és a vezérlőbuszon a számítógép ütemezéséhez, vezérléséhez szükséges információk haladnak.

A buszon átvihető adat nagysága (a busz szélessége) számítógépfüggő. A busz szélessége azt jelenti, hogy egyszerre (párhuzamosan) hány bitnyi információ haladhat át rajta.

Mivel a buszon egyszerre csak egy adat tartózkodhat, előfordulhat, hogy a részegységeknek várakozniuk kell arra, hogy a buszt használhassák. Illetve az is előfordulhat, hogy a részegység foglalt és akkor a buszon lévő adatnak kell várakoznia, ami meggátolja más eszközök buszhozzáférését. A gyorsítás érdekében az egységek és az adatbusz közé egy átmeneti tárolót, adatpuffert (cache memóriát) építenek.

A számítógépben használt részegységek (processzor, memória, különböző input/output eszközök) nem egyforma sebességgel működnek, ezért az újabb számítógép-konstrukciókban nem egy sínrendszert használnak, hanem többet. Például egyet a lassúbb perifériákhoz, egy gyorsabb sínt a nagysebességű eszközökhöz (pl. winchester), és esetleg egy harmadikat a processzor és a memóriák között. Ezt a rendszert osztott sínrendszernek nevezzük. Ilyenkor a különböző sebességű buszokat szintén pufferrel (cache-sel) kell összekapcsolni.

2.2. CPU (Central Processing Unit)

A központi egység, vagy processzor (CPU) feladata az operatív (rendszer) memóriában tárolt programok végrehajtása, a számítógép részegységeinek vezérlése.

Általánosságban a processzor felépítéséről elmondható, hogy három fő egységből áll. Az aritmetikai és logikai egység (ALU), a vezérlőegység és a regiszterek. A CPU-n belüli részegységeket egy úgynevezett belső buszrendszer köti össze.

2.2.1. ALU (Arithmetic and Logic Unit)

Az ALU legfontosabb része az összeadó. Az összeadás előjeles bináris számokkal, kettes komplemens kódban történik. Ezen kívül képes az adatok bitenkénti léptetésére, és az alapvető logikai műveletek (NOT, AND, OR, XOR) végrehajtására. A műveletekhez használt adatok és eredmények általában a regisztertömbben helyezkednek el.

Az aritmetikai és logikai egység feladata lehet még a fixpontos szorzás és osztás is. A lebegőpontos aritmetikai, és a hosszú (több regiszteres) egész számokkal végzett műveleteket egy lebegőpontos aritmetikai egység végzi, ami lehet akár a processzoron kívül is (ez az úgynevezett társprocesszor vagy ko-processzor).

2.2.2. A vezérlőegység

A vezérlőegység feladata a processzor működésének ütemezése és a külső egységekkel történő kommunikáció vezérlése. Működés közben az operatív memóriából az utasításregiszterbe töltődik a programutasítás. A vezérlőegység a betöltött utasításkódnak megfelelően vezérli a processzort.

Kétféle vezérlési módot különböztetünk meg: a huzalozott (hardware) és a mikroprogramozott (firmware) vezérlést. A huzalozott vezérlés utasításkészlete kötött, az nem változtatható meg. A mikroprogramozott vezérlés azt jelenti, hogy egy processzorutasítás több mikroutasításból tevődik össze, tehát az utasításkészlet utólag megváltoztatható.

2.2.3. A regisztertömb

A processzornak a működéséhez szüksége van gyors, belső memóriára, amelyben tárolja például az utasításokat, vagy a műveletek input és output adatait. E feladatot a regiszterek látják el, amelyek korlátozott számuk és méretük miatt nagymennyiségű adat tárolására nem használhatók. (Erre való az operatív memória.)

A regisztertömb elemeit – funkciójuk szerint – három fő csoportba sorolhatjuk:

- Az ALU-hoz tartozó regiszterek, amelyek a műveletek operandusait és eredményeit, valamint a jelzőbiteket tárolják. Ezek az akkumulátor(ok) (accu) és a jelzőbit (flag) regiszterek.
- A vezérlőegységhez tartozó regiszterek, amelyek a vezérlést, a címzést és az utasítás-végrehajtást támogatják. Ezek közé tartozik az utasításregiszter, az utasításszámláló, az indexregiszter és a veremmutató is.
- Az általános célú regiszterek, amelyekben tárolhatunk adatokat, vezérlőkódokat és címeket is.

A regiszterek szóhossza (hány bit hosszúságú) általában megegyezik a processzor szóhosszával. (A legelterjedtebbek a 16, 32 és a 64 bit szóhosszúságú processzorok.)

A nagyobb számokkal végzett műveletekhez, a nagyobb memóriacímek eléréséhez a regiszterek összekapcsolhatók, úgynevezett regiszterpárokat használhatunk.

2.2.4. A processzor üzemmódjai

A normál működési állapot

A Neumann-elvű, tárolt programú számítógépek normál állapotán azt értjük, amikor a processzor egymás után végrehajtja az operatív memóriában tárolt utasításokat. Tehát ilyenkor az utasításciklusok egymás utáni végrehajtása történik.

Az utasításciklus három részből áll: az utasítás betöltése az operatív memóriából (az utasításmutató regiszterben tárolt címről), majd az utasítás hosszának megfelelően beállítja a következő utasítás címét, és végül végrehajtja azt.

Abban az esetben, ha a programunk befejeződött, vagy valamilyen oknál fogva felfüggesztettük a végrehajtását, a processzornak le kellene állnia. Azonban ez nem megengedett állapot, hiszen a processzor indít és felügyel minden folyamatot, így egy álló processzort hiába „szólítana" meg mondjuk egy periféria. Ezért – amikor a processzornak nincs feladata – egy úgynevezett várakozó ciklusban működik. (A modern mikroprocesszorok és mikrokontrollerek esetében létezik egy SLEEP üzemmód is, amikor meghatározott időre felfüggeszthetjük a processzor működését. Ilyenkor a hardvernek kell gondoskodnia az újraindításról, illetve a működés folytatásáról.)

A processzorvezérelt kiszolgáló állapot: megszakítás (interrupt)

A processzor mindaddig normál működési állapotban van, amíg valamilyen külső eszköz kiszolgálása nem válik szükségessé. Ekkor az input/output eszközök közötti kommunikációt mindig a processzor vezérli. Ez történhet a processzor kezdeményezésére (például programmal kiírunk a terminálra), illetve valamelyik eszköz kérésére.

Abban az esetben, ha valamilyen input/output eszköz kér kiszolgálást, akkor ezt jelzi a processzornak a vezérlőbuszon keresztül (interrupt kérés). Ha a processzor fogadni tudja a kérést, felfüggeszti az éppen futó program működését (megszakítás), és átadja a vezérlést az eszközkiszolgáló programnak.

Több, egyidejű kérés esetén hardveresen (vektor-interrupt), vagy programmal (operációs rendszerrel) kell eldönteni a kiszolgálás sorrendjét.

A processzortól független kiszolgáló állapot: DMA (Direct Memory Access)

Egy számítógépes rendszerben általában a processzor irányít minden műveletet. Azonban lehetnek olyan feladatok, amikor a processzor annyira leterhelt, hogy más egységeknek kell rá várakozniuk, valamint gyakran előfordul olyan tevékenység, amit a processzortól függetlenül is végre lehet hajtani.

A fentiekből következik, hogy célszerű a buszon egy processzortól független működési módot biztosítani, ilyen például a memória és a háttértár közötti adatátvitel. Ezt a módszert közvetlen adatátvitelnek, DMA-nak nevezzük. A processzor ilyenkor csak a tranzakció elindításában és a befejezés nyugtázásában vesz részt, egyébként a DMA-vezérlő átveszi a rendszerbusz irányítását.

2.2.5. Processzortípusok

CISC (Complex Instruction Set Computer)

A számítógép-alkalmazások fejlődésével megnőtt az igény az egyre többet tudó, egyre több utasítást ismerő processzorok iránt és szükség volt a régi gépekkel való kompatibilitás megőrzésére is, ezért egyre bonyolultabb gépi utasításokat vezettek be. Ezen utasításokat végrehajtó processzorokat CISC (Összetett Utasításkészletű Számítógép) processzornak nevezzük.

Az összetett utasításkészlet nagymértékben megkönnyíti az assembly programozást, valamint hatékony támogatást nyújt a magasszintű programnyelvekhez. A nagy utasításkészlet (akár 2-300 utasítása is lehet egy processzornak) bonyolultabbá és időigényesebbé teszi a mikroprogram futását, a nagy mikroprogram pedig nagy belső mikroprogram-memóriát igényel. (A mikroprogram egy interpreter, amely a gépikódú utasításokat futtatja.)

Jelentősebb CISC processzorok az Intel sorozat (286, 386, 486, Pentium), a Digital (DEC) PDP és VAX processzorai, valamint az IBM nagygépeiben használt processzorok.

RISC (Reduced Instruction Set Computer)

Az 1980-as évek elején – az alkalmazott utasítások statisztikai elemzése alapján – megtervezték a RISC (Csökkentett Utasításkészletű Számítógép) CPU-t, amelyben nem használtak interpretert. Ezek a processzorok jelentősen különböztek az akkoriban kereskedelemben lévőktől, azokkal nem voltak kompatibilisek. Ezeknél viszonylag kisszámú (kb.50) végrehajtható utasítást használtak.

A RISC processzor minden egyszerű parancsot közvetlenül végre tud hajtani, így nincsenek mikroutasítások, nincs interpreter, és nincs szükség a mikroprogram-memóriára sem. Ebből következik, hogy jóval egyszerűbb és gyorsabb a CPU.

Jelentősebb RISC processzorok a SUN szerverekben és munkaállomásokban használt processzorok, a DEC Alpha processzora és a HP munkaállomások processzorai.

Az Intel a 486-os sorozattól kezdve egy olyan processzortípust fejlesztett ki, amely ugyan CISC processzor, de tartalmaz egy RISC magot. Így a leggyakrabban használt utasításokat interpreter nélkül hajtja végre.

Párhuzamos utasításvégrehajtás (átlapolás)

A CPU sebességének növelését az utasítások párhuzamos végrehajtásával is megvalósíthatjuk. A párhuzamos működés megoldható a CPU-n belül, azaz utasításszinten, és CPU-n kívül, vagyis több processzor összekapcsolásával.

Az utasításszintű párhuzamosításra az ad lehetőséget, hogy a processzor egy ciklus alatt több részfeladatot hajt végre (az utasítás betöltése a memóriából, az utasítás dekódolása, az operandusok lekérdezése, majd az utasítás végrehajtása). A hagyományos processzorok esetében a következő utasításnak meg kell várnia azt, hogy az előző befejeződjön. A párhuzamos processzorok esetében az utasítás egy úgynevezett adatcsatornába (pipeline) kerül. Ennek működése a következő: az első utasítást beolvassuk a memóriából, majd az a csatorna második állomására, a dekódoláshoz kerül. A dekódolás alatt a csatornába beolvassuk a következő utasítást. Ezt követően a csatornában minden adat továbbhalad, az első utasítás az operandus lekérdezéshez, a második a dekódoláshoz, közben beolvassuk a harmadik utasítást,…és így tovább. A folyamat hasonló a futószalagon végzett szereléshez. Belátható, hogy ennél az egyszerűbb ciklusnál is egy időben négy utasítással foglalkozik a CPU.

Ilyen elven működnek az Intel processzorok a 486-tól kezdve (a Pentiumokban már nemcsak egy pipeline-t használnak).

A másik párhuzamos működési lehetőség az, amikor a számítógépben egyszerre több processzort használunk. Ilyenkor az operációs rendszernek kell gondoskodnia a processzorok ütemezéséről. A processzorok adatkapcsolatát általában a közös memória biztosítja. Ilyenek az array és a multiprocesszoros számítógépek.

2.3. Az operatív memória

Minden számítógép – a Neumann-elvből következik – tartalmaz egy belső (operatív vagy rendszer) memóriát az éppen futó programok (utasítások) és az adatok tárolására. A memória feladata, hogy az információt bináris formában tárolja, majd azt a megfelelő időben (pl. a processzor kérésére) rendelkezésre bocsássa.

A memória az információ tárolását egy memóriamátrixban oldja meg. Ez nem más, mint elemi (egy bites) tároló-cellák összessége, ahol a sorok byte hosszúságúak. A mátrixos szervezés megkönnyíti az adatok helyének gyors és pontos meghatározását (minden egyes sor címezhető).

A félvezető memóriák előtt a ferrittárak, azelőtt pedig a mágnesdobos tárak voltak az egyeduralkodók. Ezek úgynevezett „nem felejtő” memóriák voltak, ugyanis a tápfeszültség kikapcsolása (kimaradása) esetén megőrizték az információt. Ezeket az eszközöket a processzor írásra és olvasásra is tudta használni. Az ilyen típusú memóriákat nevezték RAM-nak (Random Access Memory). Az elnevezés arra utal, hogy a memória címfüggetlen, tetszőleges elérésű (tehát nem soros[4] elérésű).

A félvezető memóriák megjelenésével átvették ezt az elnevezést, ami nem volt szerencsés, ugyanis ezeknek az áramköröknek nem ez a fő jellemzője. Továbbá, a félvezető memóriák között megjelentek olyan típusok, amelyeket a CPU nem tud írni, csak olvasni. Ezek az úgynevezett ROM-ok (Read Only Memory), amelyek szintén tetszőleges (nem soros) elérésű memóriák.

2.3.1. RAM (Random Access Memory)

Mint már említettük, a RAM (tetszőleges vagy véletlen elérésű memória) elnevezés történeti okokból maradt fenn. Szerencsésebb lenne az írható-olvasható elnevezés használata.

A félvezető RAM-ok csak tápfeszültség jelenlétében őrzik meg az információt, ezért a program indulásakor azt be kell tölteni egy nem felejtő memóriából vagy háttértárról.

A RAM-nak két csoportját különböztetjük meg: a statikus és a dinamikus memóriákat.

A **statikus RAM**-ok (SRAM) cellánként (bitenként) egy billenő áramkört (D flip-flop) tartalmaznak. Ezen áramkörök tulajdonsága, hogy az információt (ami 0 vagy 1) mindaddig tárolják, amíg azt nem változtatjuk meg, vagy nem kapcsoljuk ki a tápfeszültséget.

Egy D flip-flop áramkör hat NAND kapuból alakítható ki, amiből nyilvánvaló, hogy egy bit tárolásához jelentős mennyiségű alkatrészből álló áramkörre van szükség. Viszont előnye a nagy sebessége, tipikus elérési ideje néhány *ns* (nanoszekundum). Ezért az SRAM-okat olyan helyen alkalmazzák, ahol követelmény a nagy sebesség, és nem túl fontos a nagy kapacitás, például a cache memóriákban.

A **dinamikus RAM**-ok (DRAM) az SRAM-mal ellentétben nem bonyolult felépítésű billenő áramköröket használnak, hanem kihasználják azt, hogy egy kondenzátor bizonyos ideig megőrzi töltését. Így minden egyes cellában elegendő egy kapacitiv félvezető elem és egy erősítő tranzisztor. A tárolók (kondenzátorok) feltöltött vagy kisütött állapota jelenti az 1 vagy 0 értéket.

A cellánkénti kis alkatrészigény miatt ezzel a technológiával olcsó és nagykapacitású tárolók építhetők.

A kapacitiv jellegű információtárolás hátránya, hogy az elektromos töltés idővel csökken, illetve megszűnik. Ez az idő ma már néhányszor 10 ms (milliszekundum). Annak érdekében, hogy

[4] Kezdetben a memóriaáramkörök fejlesztésekor csak léptetőregiszterekből, vagy még korábban (például az EDVAC számítógépben) higanyos művonalból épített soros elérésű tárolókat (Serial Access Memory, SAM) használtak.

az információ ne vesszen el, a DRAM állandó frissítést igényel, azaz a cellákat újra fel kell tölteni. Ezen kívül – ellentétben az SRAM-mal – a DRAM olvasáskor elveszíti a tartalmát, tehát azt vissza kell írni. A frissítésről és az adatvisszaírásról a memória áramkörei gondoskodnak, ami viszont jelentősen lassítja működését, hiszen a frissítés ideje alatt a DRAM másra nem használható. A DRAM-ok elérési ideje több 10 ns.

A DRAM-ot – a viszonylagos lassúsága ellenére – a számítógépek, munkaállomások operatív memóriájaként szokták felhasználni, mivel a DRAM nagy kapacitású, olcsó memóriatípus.

A memória áramkörök mindig lassúbbak voltak, mint a CPU-k, így a processzornak általában várnia kell, míg a memóriából megkap egy adatot. (Minél lassúbb a memória, annál többet kell várnia a processzornak.)

Felmerül a kérdés, hogy a sebesség növelése érdekében (drága és kis kapacitású) SRAM-okat, vagy a tárolókapacitás növelésére (olcsó, de lassú) DRAM-okat használjunk-e. Ezt a problémát úgy oldhatjuk meg, hogy a kicsi, de gyors memóriát kombináljuk a nagykapacitású, de lassú memóriával. Így egy viszonylag gyors és nagy memóriát kapunk.

A fenti eljárást „cache-elérésnek” hívják, amely a következőképpen működik: amikor a CPU-nak szüksége van egy adatra, először azt a cache-ben (SRAM) nézi meg. Csak akkor fordul a fő memóriához (DRAM), ha az adat nincs a cache-ben. Az információ ekkor az operatív memóriából a cache-be kerül, ahonnan a processzor már használhatja. Ez így még nem gyorsítja a memória elérését, de amikor az operatív memória és cache között adatot másolunk, akkor nemcsak azt az egyet írjuk át, amire éppen szüksége van a CPU-nak, hanem a következőket is. Mivel az egymás után végrehajtandó utasítások is és az egymás után feldolgozandó adatok is – az esetek döntő részében – sorban, egymás után helyezkednek el a memóriában, a processzor következő adatkérése esetén a szükséges információk – nagy valószínűséggel – már a cache-ben lesznek.

A cache-memóriák lehetnek a CPU-n kívül (Intel 386), és lehetnek belül (Intel 486). A további gyorsítások érdekében többszintű cache-t is alkalmazhatnak.

Nem tekintjük külön memória kategóriának, de az eltérő felhasználás jellege miatt mégis külön foglalkozunk a **CMOS RAM** memóriával.

A CMOS (Complementary Metal Oxide Semiconductor) egy kis áramfelvételű, kisméretű memória, melyben olyan változó adatokat tárolunk, amelyekre szükség van a számítógép kikapcsolása után is. A memória – a kis áramfelvétel következtében – egy kiskapacitású akkumulátort vagy elemet használ a folyamatos tápellátás biztosítására.

A számítógépekben a CMOS tartalmazza a rendszerdátumot és az időt, valamint a különböző eszközök alapadatait és a számítógép indításához szükséges információkat.

2.3.2. ROM (Read Only Memory)

A RAM-ok használatakor egy alapvető problémával állunk szemben. Amikor bekapcsoljuk a számítógépet, az operatív tár üres (pontosabban használhatatlan adatokkal van tele). De akkor hogyan induljon el a számítógép, hiszen a processzor csak az operatív memóriában tárolt programok futtatására képes? Két lehetőségünk van: az egyik, hogy valamilyen külső tárolóról töltjük fel a memóriát. A másik, hogy az induláshoz olyan memóriát használunk, amelyik tápfeszültség nélkül is megőrzi az információt.

A mindennapi életben számtalan olyan alkalmazás van, ahol nem használunk háttértárat. Például háztartási gépekben, játékokban, különböző vezérlő elektronikákban a programoknak és az adatok egy részének meg kell maradnia a kikapcsolás után is.

Ezekre a feladatokra fejlesztették ki a ROM (csak olvasható memória) áramköröket, amelyek tápfeszültség nélkül is megőrzik az információt. Általánosságban megjegyezhetjük, hogy a RAM és a ROM között – a CPU szempontjából – csak annyi a különbség, hogy a processzor a ROM-ba nem tud írni.

Az adat a ROM-ba a gyártás során kerül, úgynevezett maszk-programozási technikával. A tárolt információ utólag nem változtatható meg.

Annak érdekében, hogy az eszközfejlesztőknek ne a memóriagyártóktól kelljen megrendelni az egyedi programozott áramköröket, kifejlesztették a **PROM** (Programmable ROM), programozható ROM áramköröket.

A PROM egyszer írható memória. Az írás egy viszonylag magas feszültséggel történik, amivel kiégetnek egy-egy mátrixpontot. Ebből az eljárásból maradt meg a későbbiekben a „program beégetés” kifejezés.

A ROM és a PROM memóriák semmilyen körülmények között nem törölhetők, és nem írhatók újra. Ha mégis új programra van szükségünk, az egész memória-áramkört (chip-et) ki kell cserélnünk. (Az eszközök – elsősorban a mikroprogramok – fejlesztésekor, vagy ha a berendezés üzemideje alatt új programverziók várhatók, gyakran kell cserélni ezeket az alkatrészeket.) Ezért szükség volt olyan ROM típusú memóriákra, amelyeket újra fel lehet használni.

Az újraprogramozható memóriák első generációi az **EPROM** (Erasable PROM), a törölhető PROM memóriák.

Az EPROM-ok törlését erős ultraviola fény segítségével végezhetjük. Ez úgy történik, hogy a chip tokozásán található egy kis üvegablak, amelyen keresztül megvilágítható az áramköri lapka. A fény hatására mindegyik memóriacella értéke 1 lesz. A programozás – hasonlóan a PROM-hoz – külön, erre a célra kifejlesztett elektronikával történik.

Láttuk, hogy az EPROM-ok programozása elég nehézkes: azokat ki kell venni a berendezésből, egy speciális fényforrással meg kell világítani, egy beégető segítségével be kell programozni, majd vissza kell tenni az eredeti helyükre. Erre a folyamatra nincs szükség az **EEPROM** (Electrically Erasable PROM), elektromosan törölhető memóriák esetében.

Az EEPROM-ok elektromos impulzusokkal, az eredeti helyükön programozhatóak. A méretük jóval kisebb, mint az EPROM-oké, de lassúbbak és jóval drágábbak azoknál.

Az EEPROM-hoz hasonló a **flash** memória. Egyre elterjedtebben használják digitális fényképezőgépekben, kamerákban és számítógépek adathordozójaként.

A memóriaáramkörök – például a tápfeszültség-ingadozás hatására – hibákat véthetnek. Néhány memóriafajta – a hibák elleni védekezésül – hibajelző vagy hibajavító kódot használ. Hibajavító kód esetén a memória az alapadat mellé az íráskor újabb (un. hibafelismerő vagy hibajavító) biteket rak. Olvasáskor ezek segítségével ellenőrzi az adatok helyességét.

2.4. Az input/output egységek

A számítógépek felhasználója az ember, ezért szükség van olyan berendezésekre, eszközökre, amelyek biztosítják az ember és a számítógép közötti kommunikációt. Ezek az eszközök a perifériák. Természetesen találkozhatunk olyan perifériákkal is, amelyeket csak a számítógépek használnak. Ilyenek a háttértárak (winchester, CD-ROM, floppy, stb.), illetve számítógépek közötti kapcsolatot biztosító hálózati illesztők és modemek.

Az input/output eszközöket csoportosíthatjuk funkciójuk szerint, így megkülönböztetünk bemeneti (input) és kimeneti (output) perifériákat. Az input perifériák segítségével vezéreljük a számítógépet, utasításokat és adatokat táplálhatunk bele. Az output perifériákkal a számítógép ad

át szöveges, képi vagy nyomtatott információkat az ember számára. Ezen kívül lehetnek úgynevezett input-output eszközök is, amelyek képesek adatok fogadására és küldésére is. Ezek általában a háttértárak, de ilyenek például a hálózati csatolók és az audio illesztők is.

Nemcsak funkciójuk szerint csoportosíthatjuk ezeket az eszközöket. Például belső és külső perifériaként attól függően, hogy a számítógépházba beépítve, vagy azon kívül találhatók, cserélhetőek vagy fixen beépítettek, hálózati eszközök vagy csak a munkaállomás használhatja azokat és így tovább.

Egy újabb csoportosítási lehetőség az, hogy a rendszerbuszra kapcsolódó periféria-illesztő áramkör és a periféria közötti adatátvitel milyen típusú, ami például lehet soros vagy párhuzamos.

A perifériák részletesebb ismertetése nem e jegyzet célja.

3. Bináris aritmetika

3.1. A számrendszerek

A mindennapi életben a számokat a tízes (decimális) számrendszerben ábrázoljuk. Ez a történelem során alakult ki, talán azért, mert az ember a tíz ujját is felhasználta a számolás megkönnyítésére. Korábban más számrendszereket és számábrázolásokat is használtak, például: római számok, babilóniai hatvanas számrendszer. (Innen ered az idő-mértékegység (óra) felosztása is.) A mechanikus gépek is a tízes számrendszert használták, mivel az adatmegadás és az eredmény kiértékelése így volt a legegyszerűbb. Ezekben a berendezésekben általában egy tárcsa helyzete (elfordulása) jelezte, hogy melyik alaki értékről van szó, a tárcsák egymáshoz viszonyított elhelyezése pedig a helyi értéket jelentette.

Ha az elektronikus berendezések is tízes számrendszert használnának, akkor nagyon bonyolult, tíz különböző állapotot tükröző áramkört kellene építeni. Ennél sokkal egyszerűbbek és így megbízhatóbbak a két-állapotú elektronikák. (Elég azt vizsgálnunk, hogy egy vezetőben folyik áram vagy nem, egy kapcsoló be- vagy kikapcsolt állapotban van-e.) Ebből következik, hogy olyan belső kódrendszerre (számábrázolásra) van szükség, amelyben a használt állapotok (jelek) száma kettő. Így az elektronikus számítógépek a kettes (bináris) számrendszert használják. További előny, hogy a bináris adatábrázolás segítségével egységesen tudjuk kezelni az aritmetikai és logikai adatokat, műveleteket. Tehát: egy kétállapotú elektronikai elem jelölheti a bináris 1-et vagy 0-át és a logikai *igen* vagy *nem* állapotot.

A másik ok: a bináris számrendszer használatával minden – a kettes számrendszerben – végzett művelet felírható elemi logikai műveletek sorozatával. Ennek megfelelően a számítógép elektronikája is felépíthető néhány elemi logikai áramkör felhasználásával.

Természetesen a számítógép felhasználóját nem érdekli, hogy az adatokat milyen számrendszerben tárolja a gép, ő a megszokott tízes számrendszerben szeretné megadni az értékeket, és az eredményt is így várja. Ezért a bevitelnél és a kimenetnél meg kell oldani a gyors és viszonylag pontos konverziót. (Hogy mit jelent a viszonylag pontos, arra még visszatérünk!)

Megjegyezzük, hogy ha a felhasználót nem is érdekli a számítógép belső számábrázolása, azt egy programozó már nem hagyhatja figyelmen kívül.

Minden nem negatív szám felírható a következő alakban:

$$a_k \cdot 10^k + a_{k-1} \cdot 10^{k-1} + ... + a_1 \cdot 10 + a_0 + a_{-1} \cdot 10^{-1} + ... + a_{-l} \cdot 10^{-l} \qquad (3\text{-}1.)$$

ahol: k, j = 0, 1, 2, … és a_k … a_0 … a_{-l} 9-nél nem nagyobb, nem negatív, egész szám.

Ugyanígy egyértelműen felírható ez a szám kettes számrendszerben is:

$$b_m \cdot 2^m + b_{m-1} \cdot 2^{m-1} + ... + b_1 \cdot 2 + b_0 + b_{-1} \cdot 2^{-1} + ... + b_{-n} \cdot 2^{-n} \qquad (3\text{-}2.)$$

ahol: m, n = 0, 1, 2, … és b_m … b_0 … b_{-n} értéke 0 vagy 1.

Mivel a kettes számrendszerben is olyan a szimbólumokat (számjegyeket) használunk a számok írásakor, amik a tízes számrendszerbeli számokban is előfordulnak, ránézésre nem mindig

egyértelmű, hogy melyik számrendszert használjuk. A félreértések elkerülése érdekében a rendszer alapszámát indexben jelezzük. Például:

$$11_{10} = 1011_2 \text{ vagy } 1010_2 = 10_{10}$$

A továbbiakban a kettes számrendszerbeli (bináris) szám egy helyi értékét **bitnek** nevezzük. (Bit: az angol *binary digit* szavak összevonásából származik, de „pici", „falat" jelentése is van.) Ennek megfelelően a fenti, 3-2. képlettel felírt bináris szám ábrázolására $m+n+1$ bit szükséges.

A bit az informatikában használatos legkisebb információhordozó egység, az informatika alapegysége, értéke: 0 vagy 1.[5]

A tízes számrendszerben ezresével csoportosíthatjuk a számokat – az egyszerűbb kezelhetőség érdekében –, és ezeknek a nagyságrendeknek külön elnevezést adhatunk. A bináris számoknál is ezt a hagyományt követik, bár itt nem pontosan ezerszeres a váltószám.

Például 1 kg (kilogramm) = 10^3 g (gramm), de 1 kbit (kilobit) = 2^{10} = 1024 bit. Ennek megfelelően felírható a következő táblázat:

1024 bit	= 1 kbit (kilobit)	= 2^{10} bit
1024 kbit	= 1 Mbit (megabit)	= 2^{20} bit
1024 Mbit	= 1 Gbit (gigabit)	= 2^{30} bit
1024 Gbit	= 1 Tbit (terabit)	= 2^{40} bit

A nyolcbites (nyolcjegyű) bináris számot **byte**-nak (bájt) nevezzük. A bájt a digitális információfeldolgozás alapegysége. A memóriák és a háttértárak kapacitását is bájtban mérik. Az előzőekhez hasonlóan itt is használatos a kB (kilobájt), MB (megabájt), stb. elnevezés.

Bináris számrendszerben a számok sokkal hosszabbak lesznek, mint a tízes számrendszerben. Láttuk, hogy $10^3 \approx 2^{10}$, ami azt jelenti, hogy 10 helyi érték hosszúságú bináris szám felel meg 3 helyi értékű decimális számnak. Ebből látszik, hogy a bináris számok nehezen kezelhetőek. Az áttekintés megkönnyítésére vezették be a nyolcas (oktális) és a tizenhatos (hexadecimális) számrendszereket. Azért éppen ezeket, mert a számrendszerek között egyszerű az átváltás ($8 = 2^3$; $16 = 2^4$).

Az oktális számrendszer előnye, hogy csak olyan jelöléseket (alaki értékeket) használunk, amelyek ismertek a tízes számrendszerben. Továbbá a nyolc közel áll a tízhez, így a nyolcas számrendszerben történő ábrázolás nem sokkal hosszabb, mint a tízes számrendszerbeli.

A kettes és a nyolcas számrendszerek közötti átváltás egyszerűen elvégezhető. A bináris számot jobbról bithármasokba (triádokba) csoportosítjuk, egy-egy csoport egy oktális számjegynek felel meg. A visszaalakítás is egyszerű, mivel a nyolcas számrendszerben megadott szám egy számjegye megfelel egy bináris triádnak.

Az oktális számrendszer hátránya, hogy egy bájt nem egész számú triádból áll, ami a programozásnál jelent némi gondot. Ezt a hátrányt küszöböli ki a tizenhatos számrendszer.

A hexadecimális számok és a bináris számok között is könnyű a konverzió, csak itt négyes csoportokat kell alkotni. További előny, hogy egy bájt értéke felírható két hexadecimális számjegy segítségével (8 biten $2^8 = 16^2$ különböző érték jeleníthető meg). Hátránya, hogy a tizenhatos számrendszerben 16 különböző jelre van szükség. Mivel a tízes számrendszerben tíz (0, 1, 2,…9)

[5] Claude Elwood Shannon (1916 – 2001) híradástechnikus és matematikus az információelmélet megalkotója. A legjelentősebb műve, a „Mathematical Theory of Communication", 1948-ban jelent meg, amelyben meghatározta egy rendszer entrópiáját és bevezette az információ egységét.

darab jelünk van a számok ábrázolására, így a tíztől a tizenötödik elem jelölésére új szimbólumokat kellett bevezetni:

$$10_{10} = A_{16};\ 11_{10} = B_{16};\ 12_{10} = C_{16};\ 13_{10} = D_{16};\ 14_{10} = E_{16};\ 15_{10} = F_{16};$$

3.2. Adatábrázolás

Adatábrázolás szempontjából kétféle számot különböztetünk meg: az egész típusú (fixpontos) és a lebegőpontos számokat. Az ezekkel végzett számolásokat két külön aritmetikai egység végzi.

Az egész típusú számokat tovább csoportosíthatjuk tartalmuk (jelentésük) szerint: jelenthetnek egész számot, de lehetnek kódok is (pl. utastáskód, karakterkód stb.).

3.2.1. A fixpontos számábrázolás

Legegyszerűbb számábrázolási mód az egész számok ábrázolása, tárolása. Ilyenkor például egy tetszőleges egész számot kettes számrendszerben, a megfelelő bit-hosszúságban kell tárolni. Szabvány szerint a számítógépek ezeket 1, 2, 4 vagy 8 bájt hosszúságban tárolják. (Ennek megfelelően például 1 bájton (8 bit) $2^8 = 256$ különböző értéket tudunk ábrázolni, azaz pl. 0 – 255 közötti számok tárolására alkalmas 1 bájt.)

Fixpontos (fixed point) számábrázolással lehetséges törtek tárolása is. Ebben az esetben a rendelkezésre álló adattároló elemet két részre: egészre és törtre osztják. Ennek hátránya, hogy nem rugalmas. Például egy nagyon nagy szám ábrázolásakor, amikor a törtrész elhanyagolható lenne, akkor is helyet foglalunk számára. A fixpontos ábrázolás egyik speciális esete – a már említett – egész számok esete, amikor a tizedespontot (kettedes pontot) az utolsó bit után rögzítjük.

A számok ábrázolásánál megkülönböztetünk előjeles és előjel nélküli számokat. Az előjeles abban tér el az előjel nélkülitől, hogy a szám legnagyobb helyi értékű bitjét kinevezik előjelbitnek, amely pozitív számok esetén 0, negatív számoknál pedig 1.

A fixpontos számoknál – a programozás során – tekintettel kell lenni az úgynevezett túlcsordulásra. (Törtrész esetén a lecsordulásra.) Ez azt jelenti, hogy egy művelet eredménye lehet egy olyan hosszú szám, amely a rendelkezésre álló helyen már nem fér el. A fenti példa alapján 1 bájtban 255-nél nagyobb számot nem tudunk tárolni.

A túlcsordult számjegyek elvesznek, a maradékkal való számolás hibás eredményt adhat. Ezért erre a programozás során külön figyelmet kell fordítanunk, mivel ilyen esetben – általában – sem a fordítóprogramtól, sem a futtató környezettől nem kapunk visszajelzést.

3.2.2. A lebegőpontos számábrázolás

A fixpontos ábrázolás hátrányait küszöböli ki az úgynevezett lebegőpontos számábrázolás, amely bizonyos szempontból rugalmasabban, de bonyolultabban kezeli a számokat. A lebegőpontos ábrázolás alapja az, hogy a számok hatványkitevős alakban is felírhatók:

$$\pm m \cdot p^k$$

ahol: p: a számrendszer alapszáma (esetünkben 2),
m: mantissza,
k: karakterisztika.

A hatványkitevős forma egyértelműsége érdekében elfogadtak egy közös elvet: az *m* mindig kisebb, mint 1, és a tizedesponttól (esetünkben inkább „kettedespont”) jobbra eső első számjegy nem lehet nulla.

$$\frac{1}{p} \le m < 1$$

(Például: tízes számrendszerben $0{,}1 \le m < 1$, kettes számrendszerben $0{,}5 \le m < 1$.)

A fenti feltételeket teljesítő felírási módot normalizálásnak nevezzük, amit természetesen a számítógépek automatikusan elvégeznek. Egy, a gyakorlatban is használható lebegőpontos szám tárolására minimum 32 bitre (4 bájtra) van szükség.

Ezek után nézzük meg, hogy egy lebegőpontos szám ábrázolásához milyen adatokra van szükség:

- a mantissza abszolút értéke,
- a mantissza előjele,
- a karakterisztika abszolút értéke,
- a karakterisztika előjele.

A mantissza előjelét (azaz a szám előjelét) ugyanúgy a legnagyobb helyi értékű biten jelezzük, mint a fixpontos számoknál. A következő bitek tartalmazzák az előjeles karakterisztikát, majd a jobb oldalon helyezkedik el a mantissza.

Mivel a mantissza tárolására korlátozott számú bit áll rendelkezésre, nem tudunk tetszőleges pontosságú számot ábrázolni. A lebegőpontos számok e tulajdonságát számábrázolási pontosságnak hívjuk. Szokás ezt úgy is jellemezni, hogy egy adott szám hány jegyre (helyi értékre) pontos.

A karakterisztikát és annak előjelét egyben, úgynevezett eltolásos módszerrel ábrázolják. Példának vegyünk egy 8 bites (1 byte-os) kitevőt. Ekkor a kitevő tényleges értékét úgy kapjuk meg, hogy az ábrázolt értékből kivonunk 127-et, például 1 esetén 128, -1 esetén 126 a kitevőben tárolt érték.

Megjegyezzük, hogy géptípusonként eltérhet a lebegőpontos számok bitképe.

3.2.3. Decimális számok ábrázolása

Ez a számábrázolási forma elsősorban üzleti számításoknál (pl. adatbázis-kezelőnél) fordul elő. Itt nem az aritmetikai műveleteken van a hangsúly, ezek száma kicsi. Sokkal gyakrabban fordul elő a számok beírása, illetve megjelenítése, ami a számrendszerek közötti konverziók számát megnövelné. Sok esetben – például banki műveleteknél – nem megengedett a konvertálásból származó pontatlanság. Ezért olyan ábrázolásmódot fejlesztettek ki, amely szükségtelenné teszi a számok kettes számrendszerre való konvertálását. (Természetesen a tárolás itt is binárisan történik.)

Ezzel a módszerrel – ellentétben a lebegőpontos ábrázolással – tetszőleges pontosságú számot lehet tárolni.

A decimális számábrázolás formája számítógéptől függ. Például a régebbi IBM gépeken a BCD (Binary Coded Decimal) kódolást alkalmazták. Itt a számjegyeket és az előjelet fél bájton (4 biten) ábrázolták. A PDP számítógépeknél a számjegyeket közvetlenül az ASCII kódjukkal tárolták. Ebben az esetben egy helyi érték és az előjel tárolása is egy-egy bájton történt.

3.2.4. Logikai értékek ábrázolása

A matematikai logikában logikai kifejezésekkel dolgozunk. Egy logikai kifejezés lehet egyetlen egy állítás is, amely két különböző értéket vehet fel, attól függően, hogy az állítás *igaz* (true) vagy *hamis* (false).

Mivel egy állítás (logikai adat) csak két értéket vehet fel, a tárolására elegendő egyetlen bit is. A gépközeli (alap) utasításokban általában egy bitet használnak 1: igaz, 0: hamis. A programozási gyakorlatban azonban ott, ahol a legkisebb adathossz egy bájt, a logikai információ tárolására is egy bájtot alkalmaznak, így az igaz = 1 (00000001_2) illetve a hamis = 0 (00000000_2). Elfogadott gyakorlat az is, ha a bájt értéke 0, akkor az a hamis állításnak felel meg, ha nem 0, akkor igaznak.

3.2.5. Karakterek (szöveges adatok) ábrázolása

A betűket, a számjegyeket, az írásjeleket és az egyéb jeleket, összefoglaló néven karaktereknek hívjuk. Az egy vagy több karakterből álló sorozatot szövegnek nevezzük.

A számítógépen a szöveges állományokat kódolt formában tároljuk, azaz nem a szöveg, illetve a betűk képét, hanem a karakterek numerikus kódját. Megjelenítéskor, illetve nyomtatáskor a tárolt kódhoz kapcsoljuk hozzá azt az információt, hogy milyen formában (stílusban) jelenjen meg a karakter. Az egyértelműség érdekében a karaktereket és kódjukat különböző szabványosított táblázatok tartalmazzák.

A legelterjedtebb az Amerikai Szabványügyi Hivatal (ANSI), 1977-ben megalkotott ASCII kódtáblája, amely 128 karaktert (vezérlőkarakter, betű, számjegy, írásjel) tartalmaz. A 128 karakter 7 biten (2^7) ábrázolható, így a nyolcadik bitet paritásbitként ellenőrzésre lehetett használni. Később a számítástechnika nemzetközivé válásával szükségessé vált – az angol karakterkészletben nem található – különböző nemzeti karakterek bevezetése is. Ennek érdekében 2^8-ra, azaz 256 karakterre bővítették a kódtáblát. (Legelterjedtebb 8 bites szabvány az ISO 8859 (Latin) karakterkészlet.) Mivel az összes nemzeti írásjel tárolására ez sem volt elég, több különböző, nemzeti kódtáblát alakítottak ki (például: nyugat-európai, kelet-európai, cirill, stb.). Természetesen ezek a kódtáblák az eredeti 7 bites US-ASCII kódokat változatlan jelentéssel tartalmazzák (ld. melléklet).

Egyéb (kép, hang, videó stb.) adatok ábrázolása

Mint minden adatot a számítógépen, ezeket is digitális formában tároljuk. Azonban a kódolásokra különböző szabványos formákat találtak ki, annak megfelelően, hogy mely tulajdonságait tartjuk fontosnak, illetve milyen célra használjuk azokat.

Pixelképek esetén a pixelek tulajdonságait (pl. színét), vektorgrafikus ábrázolásnál a kép létrehozásának folyamatát (utasításait) tároljuk, akár szöveges formában.

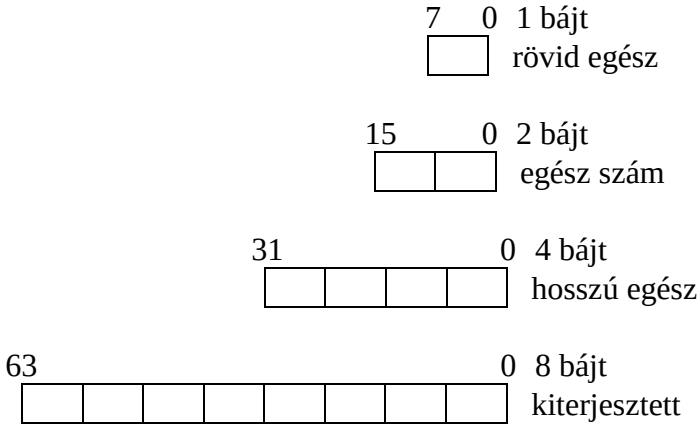
Hangok esetében mintavételezés és kvantálás után szintén digitalizálás következik.

Mind a kép-, mind a hangadatok esetében szokás az adatokon tömörítéseket is végezni, amelyek lehetnek veszteségesek vagy veszteségmentesek.

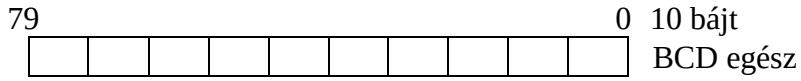
A mai modern grafikus- és hagkártyák már saját processzorral és vezérlőszoftverrel rendelkeznek, így biztosítva a gyors, valósidejű, háromdimenziós, fotorealisztikus megjelenítést.

3.3. Az Intel processzorok számábrázolása

Egész típusú számok:



Pakolt egész számok:



Valós típusú (lebegőpontos) számok:

Egyszeres pontosság (Short Real Number):

Dupla pontosság (Long Real Number):

Kiterjesztett pontosság (10-Byte Real Number):

Az összes típusú szám előjele a legnagyobb helyi értékű biten található. Ha az előjel bit 0, a szám pozitív vagy 0; ha 1, akkor a szám negatív.

A valós számok ábrázolása megegyezik a fenti, általános ismertetőben leírtakkal. E számokkal végzett műveleteket – az Intel processzorok – a 486-os sorozat óta a CPU-ba beintegrált lebegőpontos aritmetikai egységgel hajták végre. A korábbi processzorok esetében vagy matematikai társprocesszort (koprocesszort) használtak, vagy szoftveresen emulálták a művelet elvégzését.

A valós számoknál a kitevők (Exp.) határozzák meg, hogy mekkora a szám nagyságrendi tartománya. A mantissza nagysága (bitszáma) a szám pontosságára van hatással.

A binárisan kódolt (BCD) formában 18 jegyű decimális egész számot ábrázolhatunk. Ennél az adatábrázolási formánál egy bájton két decimális számjegy található. Az utolsó (tizedik) bájt legfelső bitje az előjel, a többi bitejét nem használjuk.

3.4. Műveletvégzés

A számítógépet nemcsak az adatok tárolására használjuk, hanem azokkal különböző műveleteket szeretnénk elvégezni. Mivel a felhasználó a tízes számrendszerben gondolkodik, dolgozik, a számítógép pedig – a már megtárgyalt okokból – a kettes számrendszert használja, első lépésként egy konverziót kell végrehajtanunk a számrendszerek között. Ezután végezhetők el – a kettes számrendszerben - a kívánt matematikai illetve logikai műveletek, majd az eredményt újabb, de ellentétes irányú konverzióval (tízes számrendszerben) továbbítjuk a felhasználónak.

3.4.1. Logikai műveletek

A 3.2.4. részben tárgyaltuk a logikai értékek ábrázolását, most nézzük meg, hogy milyen műveletek végezhetők velük. A logikai adatokkal, változókkal végrehajtott műveletek eredménye is logikai érték lesz. Láttuk, hogy egy logikai változó két értéket vehet fel (*igaz* vagy *hamis*), ennek megfelelően a műveletek eredménye is e két érték közül lesz valamelyik.

Gépi kódú utasítások esetén a logikai műveleteket bitenként hajtjuk végre. (Megjegyezzük, hogy a magas szintű programozási nyelvek esetében használt logikai műveleteket nem bitenként végezzük el.)

Az alábbi példákban jelöljük *A* és *B* szimbólumokkal azokat a logikai változókat, amelyekkel műveletet szeretnénk megvalósítani, és legyen *L* az eredmény:

Negálás (tagadás)

A logikai tagadás, egyváltozós művelet.

Jelölése: **NEM** (NOT), szimbóluma: ¬

	A		**L**	Állítás
¬	0	=	1	Ha A = hamis ⇒ L = igaz
¬	1	=	0	Ha A = igaz ⇒ L = hamis

A digitális technikában a jelek elnevezésénél (dokumentációkban, kapcsolási rajzokban, stb.) – a negálás jelölésére – a felülvonást használják. Például:

$$\textbf{NOT READY} = \overline{\textbf{READY}}$$

Nézzünk erre egy példát 8 biten (1 bájton):

```
NOT  1 0 1 1 0 1 0 1
     0 1 0 0 1 0 1 0
```

Logikai összeadás

Jelölése: **VAGY** (OR), szimbóluma: $\vee$

A		**B**		**L**	Állítás
0	$\vee$	0	=	0	Ha A és B = hamis $\Rightarrow$ L = hamis
0	$\vee$	1	=	1	Ha A = hamis, B = igaz $\Rightarrow$ L = igaz
1	$\vee$	0	=	1	Ha A = igaz, B = hamis $\Rightarrow$ L = igaz
1	$\vee$	1	=	1	Ha A és B = igaz $\Rightarrow$ L = igaz

Az 1 bájtos példa:

```
   1 0 1 1 1 0 1 0
OR 1 0 1 1 0 0 0 1
   1 0 1 1 1 0 1 1
```

Logikai szorzás

Jelölése: **ÉS** (AND), szimbóluma: $\wedge$

A		**B**		**L**	Állítás
0	$\wedge$	0	=	0	Ha A és B = hamis $\Rightarrow$ L = hamis
0	$\wedge$	1	=	0	Ha A = hamis, B = igaz $\Rightarrow$ L = hamis
1	$\wedge$	0	=	0	Ha A = igaz, B = hamis $\Rightarrow$ L = hamis
1	$\wedge$	1	=	1	Ha A és B = igaz $\Rightarrow$ L = igaz

Az 1 bájtos példa:

```
    1 0 1 1 1 0 1 0
AND 1 0 1 1 0 0 0 1
    1 0 1 1 0 0 0 0
```

Ha egy adatnál önmagával **ÉS** illetve **VAGY** művelet végzünk, akkor nem változik meg az értéke.

```
   1 0 1 1 1 0 1 0
OR 1 0 1 1 1 0 1 0
   1 0 1 1 1 0 1 0

   1 0 1 1 1 0 1 0
AND 1 0 1 1 1 0 1 0
   1 0 1 1 1 0 1 0
```

Ezt a tulajdonságot – programírásnál – felhasználhatjuk az adatok tesztelésére.

Kizáró VAGY

Jelölése: - (XOR), szimbóluma: ⊕

A		**B**		**L**	Állítás
0	⊕	0	=	0	Ha A és B = hamis ⇒ L = hamis
0	⊕	1	=	1	Ha A = hamis, B = igaz ⇒ L = igaz
1	⊕	0	=	1	Ha A = igaz, B = hamis ⇒ L = igaz
1	⊕	1	=	0	Ha A és B = igaz ⇒ L = hamis

Az 1 bájtos példa:

```
      1 0 1 1 1 0 1 0
XOR   1 0 1 1 0 0 0 1
      0 0 0 0 1 0 1 1
```

Adatot törölni (nullázni) tudunk, ha önmagával **kizáró VAGY** művelet végzünk.

```
      1 0 1 1 1 0 1 0
XOR   1 0 1 1 1 0 1 0
      0 0 0 0 0 0 0 0
```

Ezt az elvet fogjuk alkalmazni például a regiszterek törlésénél is.

(Több kódolási-dekódolási eljárást dolgoztak ki az $(A \oplus X) \oplus X = A$ azonosságot felhasználva.)

Mivel a processzor minden műveletet logikai műveletek sorozatából épít fel (logikai áramkörökkel hajt végre), hatékonyabb programot írhatunk, ha igyekszünk (ahol csak lehet) logikai utasításokkal programozni.

Az elektronikai alkalmazás miatt meg kell említeni még két logikai műveletet:

- **NAND** (a NEM és az ÉS műveletek egymás utáni elvégzése): $\neg (A \wedge B)$;
- **NOR** (a NEM és a VAGY műveletek egymás utáni elvégzése): $\neg (A \vee B)$;

A felsorolt logikai alapműveleteket – a számítógépben – úgynevezett kapuáramkörök végzik, így létezik AND, NAND, OR, NOR, és NOT (inverter) áramkör.

Igazolható, hogy bármely logikai művelet helyettesíthető csak NAND vagy csak NOR műveletekkel. Ennek a számítógépek felépítésénél, gyártásánál van jelentősége, hiszen gyakorlatilag egyetlen fajta logikai-áramkör segítségével kialakítható bármilyen logikai rendszer. Tehát csak egyfajta áramköri elemet kell gyártani nagy sorozatban, ami a megbízhatóságot növeli és az előállítás fajlagos költségét csökkenti.

3.4.2. Aritmetikai műveletek

Az előzőekben már láttuk, hogy a számítógépek minden műveletet a kettes számrendszerben végeznek el, és az adatokat is ebben a formában tárolják. (Ez akkor is így van, ha a felhasználó vagy a programozó – a könnyebben kezelhető forma miatt – az oktális vagy a hexadecimális számrendszert használja.)

A számítógépek felépítéséből következik, hogy fix hosszúságú (helyi értékű) számokkal dolgoznak. Ebből több olyan tulajdonság származik, amelyeket figyelembe kell vennünk a műveletek végrehajtásakor.

A **Logikai műveletek** részben már utaltunk arra, hogy a számítógép minden műveletet elemi szintre bont. Ezek után nézzük meg, hogy a bináris számrendszerben hogyan végezhetjük el az aritmetikai műveleteket.

Az alábbi példákban jelöljük *A* és *B* szimbólumokkal azokat a változókat, amelyekkel műveletet szeretnénk végezni, legyen *S* az eredmény, és vezessünk be egy *C* (carry) változót az átvitel vagy maradék jelölésére.

Összeadás

A bináris összeadást hasonlóan végezzük, mint a decimális számoknál: a számokat összeadjuk az adott helyi értéken *(A+B=S)*, és ha van maradék (átvitel) *(C)*, akkor azt hozzáadjuk a magasabb helyi értékek összegéhez.

A		**B**		**S**	**C**
0	+	0	=	0	0
0	+	1	=	1	0
1	+	0	=	1	0
1	+	1	=	0	1

Ha megvizsgáljuk az összeadás műveleti tábláját láthatjuk, hogy az eredmény nem más, mint az *A* és *B* értékkel elvégzett XOR logikai művelet, illetve a maradék a két érték AND kapcsolata. Ebből az következik, hogy egy összeadót el tudunk készíteni egy XOR és egy AND kapuáramkörből. Mivel ennél az összeadásnál még nem vettük figyelembe azt, hogy esetleg az előző (kisebb helyiértékű) két bit összeadásakor maradék is keletkezhetett, ezt az összeadó áramkört fél-összeadónak nevezzük.

Azt az áramkört, amelyik az előző átvitelt is figyelembe veszi – az eredmény kiszámításánál – teljes összeadónak nevezzük. Elkészítéséhez két fél-összeadó áramkör kell.

A számítógépek természetesen nem egybites számokkal dolgoznak, ezért az azonos helyi értéken lévő biteket egymás után össze kell adni, és a keletkezett átvitelt hozzáadni a következő helyi értékek összegéhez. Ezt elvégezhetjük sorban egymás után vagy párhuzamosan, de ekkor annyi teljes összeadó áramkörre van szükség, ahány bites az a tároló elem, amelyben a szám található.

Példaként nézzük meg két tetszőleges, 8 jegyű (1 bájt hosszú) bináris szám összeadását:

```
  1 0 1 1 1 0 1 0
+ 1 0 1 1 0 1 0 1
1   0 1 1 0 1 1 1 1
```

Mivel a példában csak 8 bit állt rendelkezésre az eredmény tárolására, a legutolsó (baloldali) átvitel elveszik (ha van). Ez a túlcsordulás, amivel a 3.2.1. részben foglalkoztunk, nem szabad figyelmen kívül hagyni, mert hibás eredményt adhat. A számítógépek – későbbi felhasználás érdekében – ezt a túlcsordult értéket ideiglenesen eltárolják.

A negatív számok előállítása

Tízes számrendszerben egy szám ellentettjét úgy kapjuk meg, hogy a számot megszorozzuk mínusz eggyel. A számítógép ezt a műveletet **kettes komplemens** képzéssel hajtja végre. Nézzük egy példa segítségével 8 bites szám esetében:

Az eredeti szám:		1 0 1 1 1 0 1 0
Negáljuk minden egyes bitjét	⇒ egyes komplemens:	0 1 0 0 0 1 0 1
Adjunk hozzá egyet	⇒ kettes komplemens:	0 1 0 0 0 1 1 0

Ha összeadjuk az eredeti számot és a kettes komplemensét, az eredmény (leszámítva az átvitelt) nulla lesz. Amit a tízes számrendszerben megszoktunk, hiszen ha egy számhoz hozzáadjuk a mínusz egyszeresét, eredményül nullát kapunk

```
  1 0 1 1 1 0 1 0
+ 0 1 0 0 0 1 1 0
  0 0 0 0 0 0 0 0
```

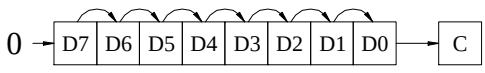
Kivonás

Digitális technikában léteznek az úgynevezett kivonó áramkörök, amelyek nagyon hasonlítanak az összeadókhoz, de a számítógépekben – az egyszerűsítés miatt – a kivonáshoz is az összeadó áramköröket használják. Természetesen ekkor a kivonandó kettes komplemensét veszik.

3.4.3. Bitenkénti léptetés (SHIFT, ROTATE)

Már többször utaltunk rá, hogy minden műveletet visszavezetnek logikai (alap) utasításokra. Ez a szorzásnál is így van. Azonban a szorzás tárgyalása előtt meg kell ismerkednünk egy olyan (nem matematikai) művelettel, amely a számítógépek működéséhez nélkülözhetetlen. Ezek a léptetés vagy shift, illetve a forgatás vagy rotálás.

Logikai shift jobbra: minden bit értékét eggyel jobbra mozgatja. A legnagyobb helyi érték 0 (nulla) lesz, a legkisebb helyi értéken kilépő bit egy erre a célra fentartott egy bites tárolóba, a Carry-be kerül.

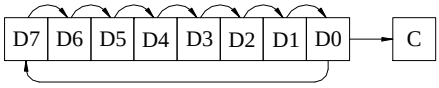


Aritmetikai shift jobbra: minden bit értékét eggyel jobbra mozgatja. A legnagyobb helyi értékre az előjel kerül (vagyis saját maga, hogy a szám előjele ne változzon meg), a legkisebb helyi értéken kilépő bit a Carry-be kerül.

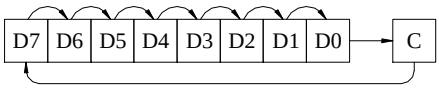
Logikai és aritmetikai shift balra: minden bit értékét eggyel balra mozgatja. A legkisebb helyi értékre a 0 (nulla), a legnagyobb helyi értéken kilépő bit a Carry-be kerül.

Figyeljük meg, hogy (a fixpontos számoknál) a balra történő léptetés kettővel történő szorzásnak, a jobbra léptetés kettővel történő osztásnak felel meg.

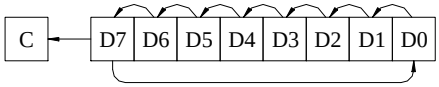
Rotálás jobbra: minden bit értékét eggyel jobbra mozgatja. A legnagyobb helyi értékre és a Carry-be a legkisebb helyi értéken kilépő bit kerül.



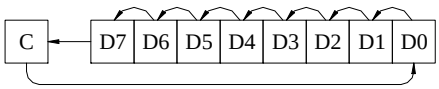
Rotálás jobbra Carry-n keresztül: minden bit értékét eggyel jobbra mozgatja. A legnagyobb helyi értékre a Carry tartalma, a legkisebb helyi értéken kilépő bit a Carry-be kerül.



Rotálás balra: minden bit értékét eggyel balra mozgatja. A legkisebb helyi értékre és a Carry-be a legnagyobb helyi értéken kilépő bit kerül.



Rotálás balra Carry-n keresztül: minden bit értékét eggyel balra mozgatja. A legkisebb helyi értékre a Carry tartalma, a legnagyobb helyi értéken kilépő bit a Carry-be kerül.



Léptető utasításokat használunk több alapfunkció (pl. soros - párhuzamos konverzió) végrehajtásán kívül a szorzás és osztás műveleténél is.

3.4.4. Szorzás

A bitenkénti szorzást hasonlóan végezzük, mint a decimális számoknál: *A*B=S*,

A		**B**		**S**
0	*	0	=	0
0	*	1	=	0
1	*	0	=	0
1	*	1	=	1

Vegyük észre, hogy az eredmény az *A* és *B* értékkel elvégzett AND logikai művelet.

A legtöbb esetben azonban nem egy bites értéket kell összeszoroznunk, hanem akár több byte hosszúságú bináris számot. Ekkor használhatjuk azt a módszert, hogy a szorzandót összeadjuk annyiszor, amennyi a szorzó. Így a szorzást visszavezettük összeadásra, de ez szorzó számú összeadást jelent. Egy hatékonyabb módszer a helyi értékenkénti összeadás (ahogy a decimális számok esetében végezzük az írásbeli szorzást), a szorzandót megszorozzuk egyesével - például balról jobbra haladva - a szorzó egyes helyi értékein álló számmal. A részszorzatokat úgy írjuk le, hogy mindig egy helyi értékkel jobbra toljuk (shift) azokat, majd az így kapott részeredményeket összeadjuk.

Nézzük egy példát két egy byte-os szám szorzására:

								·							
1	0	1	1	1	0	1	0		1	0	1	1	0	1	0	1
1	0	1	1	1	0	1	0								
	0	0	0	0	0	0	0	0							
		1	0	1	1	1	0	1	0						
			1	0	1	1	1	0	1	0					
				0	0	0	0	0	0	0	0				
					1	0	1	1	1	0	1	0			
						0	0	0	0	0	0	0	0		
							1	0	1	1	1	0	1	0	
1	0	0	0	0	0	1	1	1	0	0	0	0	0	1	0

A feladat végrehajtásakor használunk egy egybites szorzást (ÉS), a részszorzatokat eltoljuk eggyel jobbra, majd összeadjuk azokat. Mivel az összeadásnál már láttuk, hogy az is visszavezethető alap (logikai) utasításokra, a fenti folyamatból következik, hogy a szorzás is leírható alaputasítások sorozatával. (A gyakorlatban ez a folyamat valamivel egyszerűbb, mivel a nullákkal nem kell összeadást végezni, csak az eltolást végrehajtani.)

Láthatjuk, hogy az eredmény eltárolására legalább két bájt adatterület szükséges (az átvitel biten kívül). Ezt az assembly utasításoknál is figyelembe kell venni. Ennek megfelelően a bájtos szorzásnál az eredmény kétbájtos regiszterbe kerül, kétbájtos szorzásnál pedig négybájtos lesz a szorzat. Az osztás esetében fordított a helyzet, itt az eredmény fele akkora regiszterbe kerül, mint az osztandó.

3.4.5. Osztás

Az osztást például – ahogy az iskolában a bevezetésekor tanultuk – elvégezhetjük kivonások sorozatával: az osztandóból addig vonjuk ki az osztót, amíg a maradék nagyobb, mint az osztó. Az eredmény a kivonások darabszáma, a maradék az lesz, amiből már nem tudtuk kivonni az osztót.

Természetesen az osztást is elvégezhetjük jóval hatékonyabban is, felhasználva az eddig megismert bitműveleteket.

Az algoritmus működését egy mintafeladat segítségével mutatjuk be, végezzük el a 203:11 osztást:

	M (maradék)					A (osztandó)									B (osztó)					H (hányados)				
0					0	1	1	0	0	1	0	1	1	:	1	0	1	1	=	0	0	0	0	0
1					1	1	0	0	1	0	1	1	0							0	0	0	0	0
2				1	1	0	0	1	0	1	1	0	0							0	0	0	0	0
3			1	1	0	0	1	0	1	1	0	0	0							0	0	0	0	0
4	*	1	1	0	0	1	0	1	1	0	0	0	0							0	0	0	0	0
					1	1	0	1	1	0	0	0	0							0	0	0	0	1
5				1	1	0	1	1	0	0	0	0	0							0	0	0	1	0
6			1	1	0	1	1	0	0	0	0	0	0							0	0	1	0	0
7	*	1	1	0	1	1	0	0	0	0	0	0	0							0	1	0	0	0
				1	0	1	0	0	0	0	0	0	0							0	1	0	0	1
8			1	0	1	0	0	0	0	0	0	0	0							1	0	0	1	0

1. Írjuk fel a 0. sorba az elvégzendő műveletet, az osztandót (*A*), az osztót (*B*), a hányadost (*H*) és a maradékot *(M)*.
2. Léptessük *A*-t és *M*-et balra úgy, hogy az *A*-ból kilépő bit az *M*-be kerüljön. Ezzel együtt léptessük *H*-t is eggyel balra.
3. A 2. pontban leírt lépést addig végezzük, amíg az $M \geq B$ nem lesz. (*-gal jelölt sor). Ekkor módosítsuk az *M* értékét *M-B*-re és a *H* értékét *H+1*-re.
4. Folytassuk a 2. és 3. pontban leírtakat annyiszor, amilyen hosszú az osztandó. Esetünkben 8-szor. (A balra mozgatások számát a sorok elején látható szám mutatja.)

Az algoritmus alapján elkészített mintaprogram az 5.4. fejezet végén megtalálható.

3.4.6. Relációs műveletek

A számítógép, illetve a programok működéséhez – az eddig ismertetett műveleteken kívül – szükségünk lehet arra, hogy két értéket össze tudjunk hasonlítani, el tudjuk dönteni, hogy melyik a kisebb, a nagyobb, esetleg egyenlők-e. Ezeket az összehasonlító műveleteket relációknak nevezzük. Az összehasonlítás eredménye mindig egy logikai érték (igaz vagy hamis).

A relációs műveleteket a számítógép alaputasítás szinten ismeri, ami azt jelenti, hogy két értéket össze tud hasonlítani és az eredményt úgynevezett jelzőbiteken tárolja.

Amikor csak egy számról akarjuk eldönteni, hogy az negatív, nulla, vagy pozitív-e, a jelzőbitek beállítására használhatjuk az e fejezetben ismertetett módszert: önmagával AND vagy OR műveletet végzünk.

Természetesen a relációs műveletek is visszavezethetők alaputasításokra. Nézzünk egy példát. A feladat két szám összehasonlítása:

Vonjuk ki az első számból a másodikat!

- Ha az eredmény = 0 $\Rightarrow$ a két szám egyenlő;
- Ha az eredmény $\neq$ 0 és az előjel bit = 1 $\Rightarrow$ a második szám nagyobb;
- Ha az eredmény $\neq$ 0 és az előjel bit = 0 $\Rightarrow$ az első szám nagyobb.

4. Az Intel mikroprocesszorok

4.1. Kezdetek

Nézzük meg, milyen utat járt be a számítástechnika, amíg eljutott az első mikroprocesszor megalkotásáig.

Charles Babbage (1792-1871) brit matematikus és feltaláló megalkotta a modern digitális számítógép alapelveit.

Herman Hollerith (1860-1929) német származású amerikai statisztikus 1889-ben kapott szabadalmat egy lyukkártyás adatfeldolgozásra alkalmas berendezésére, amellyel az 1890-es amerikai népszámlálás adatait dolgozták fel. Hollerith 1896-ban alapította meg a Tabulating Machine Company nevű céget, amelyből aztán 1924-ben megalakult az IBM.

George Boole (1815-1864) és **Augustus de Morgan** 1847-től kezdve kidolgozta a formális logikát (a Boole-algebrát), ezzel megteremtve a modern számítógépek működéséhez szükséges egyik fontos matematikai modellt.

Leonardo Torres y Quevedo (1852-1936) 1914-ben bevezette a lebegőpontos számábrázolást a számítástechnikában. 1910 és 1920 között programvezérlésű mechanikus számológépeket épített egyedi célokra (pl. két komplex szám szorzatának kiszámítására). Tőle származnak a programozási nyelvek első kezdeményei is.

Konrad Zuse (1910-1995) 1932-ben építette Németországban az első mechanikus tárolót tetszőleges adatok, elsősorban lebegőpontos számok tárolására. A tároló 24 bites adatokat tudott fogadni. A lebegőpontos számoknál ebből 16 bit volt a mantissza, 7 bit a karakterisztika és 1 bit az előjel.

Alan Turing (1912-1954) 1936-ban az *On Computable Numbers* című művében leírta egy olyan számítógép matematikai modelljét, amely mint a lehető legegyszerűbb univerzális számítógép bármilyen véges matematikai és logikai problémát meg tud oldani.

Leslie Comrie (1893-1950) 1937-ben megalapította Londonban az első kereskedelmi jelleggel működő **számítóközpontot**. A nagyobb feladatok megoldására több számítógépet és lyukkártyás Hollerith-gépet kapcsolt össze.

Howard Aiken vezetésével fejlesztették ki az első teljesen automatikusan működő általános célú digitális számítógépet az Egyesült Államokban, a Harvard egyetemen. A tervezéshez az IBM 5 millió dollárral járult hozzá és a gép megépítését is az IBM végezte. Ez volt a ***Mark I.,*** vagy más néven **Automatic Sequence Controlled Calculator** (ASCC). Ez a gép fixpontos számokkal dolgozott (10 számjegy a tizedesvessző előtt, 13 számjegy pedig utána). Relékből épült fel, 3304 db kétállású kapcsolót tartalmazott, összesen kb. 760 000 alkatrészből állt és 500 mérföld (800 km) huzalt használtak fel hozzá. A gép kb. 15 m hosszú és 2,4 m magas volt. A memóriája a mechanikus számológépekhez hasonlóan fogaskerekekkel, tízes számrendszerben tárolta az adatokat, 72 db 23 jegyű számnak volt benne hely. Az adatbevitel lyukkártyákkal történt. A programot lyukszalag tartalmazta, ez vezérelte a gép működését. A gépnek egy összeadáshoz 0,33, egy szorzáshoz 4, egy osztáshoz 11 másodpercre volt szüksége és gyakran meghibásodott (más források szerint a szorzás ideje 6 s és a gép megbízhatóan működött). A munkát 1939-ben kezdték és 1944-ben készültek el. A tengeri tüzérség részére készítettek vele lőelem-táblázatokat. Ezt a számítógépet 1959-ig használták.

A II. világháború alatt tudósok és matematikusok egy csoportja, Bletchley Parkban (Londontól északra) létrehozta az első teljesen elektronikus digitális számítógépet, a **Colossust**. A gép 1943 decemberére készült el és 1500 elektroncsövet tartalmazott. A Colossus kvarcvezérlésű volt, 5

kHz-s órajellel dolgozott, másodpercenként 25.000 karaktert tudott feldolgozni. Összesen tíz darab ilyen gép készült. Rejtjelezett német rádióüzenetek megfejtésére használta a Turing által vezetett csoport. Ezzel a géppel sikerült megfejteni a német ENIGMA rejtjelét.

Az első általános célú elektronikus digitális számítógép az ENIAC (Electronic Numerical Integrator And Computer) volt. Az ENIAC-ot a második világháború alatt kezdte el tervezni - katonai célokra - **John Presper Mauchly** és **John William Eckert**. A gépet a Pennsylvania egyetemen építették, a munkát 1946-ban fejezték be. A gép 17 468 elektroncsövet tartalmazott, több száz kW elektromos energiát fogyasztott és 450 m^2 helyet foglalt el, tömege 30 tonna volt, az elhelyezéséhez több mint 30 m hosszú termet építettek. Három nagyságrenddel gyorsabb volt, mint a relés számítógépek: az összeadást 0,2 ms, a szorzást 3 ms alatt végezte el. A programja fixen, drótozva volt, azt csak mintegy kétnapos kézi munkával, villamos csatlakozások átkötésével lehetett átkódolni. A gép memóriája 20 db tízjegyű előjeles decimális számot tudott tárolni. A számítások elvégzéséhez szükséges előkészületek tízszer annyi időt vettek igénybe, mint maga a számítás. Így szükségessé vált, hogy a program tárolására új elvet dolgozzanak ki.

Az ENIAC utóda, az EDVAC (Electronic Discrete Variable Automatic Calculator) ugyancsak Mauchly és Eckert vezetésével épült, 1944-től 1948-ig, véglegesen csak 1951-ben helyezték üzembe. Itt már felhasználták azokat az alapelveket, amelyeket **Neumann János** (1903-1957) magyar matematikus fektetett le az ENIAC működése során szerzett tapasztalatok alapján. A számítógép a programot és az adatokat memóriában tárolta. Higanyos művonal segítségével oldották meg az adattárolást, ami jelentősen csökkentette a felhasznált elektroncsövek számát.

Neumann, **Herman Goldstine**-nal együtt építette meg saját IAS gépét. Érdekessége ennek a gépnek, hogy nem volt lebegőpontos aritmetikája. Neumann úgy gondolta, hogy egy matematikus képes a tizedespont (kettedes-pont) helyét fejben nyomon követni.

A tranzisztor 1947-es feltalálásával új lendületet kapott a számítástechnika fejlődése. A tranzisztor számítógépben történő alkalmazása az 1950-es évek közepén-végén jelent meg. A tranzisztorokkal kisebb, gyorsabb és megbízhatóbb áramköröket lehetett készíteni, mint az elektroncsövekkel. Ezek a számítógépek már másodpercenként egymillió műveletet is el tudtak végezni. A tranzisztorok sokkal kevesebb energiát fogyasztanak és sokkal hosszabb életűek, megbízhatóságuk pedig mintegy ezerszerese az elektroncsövekének. A méretek csökkenése miatt sokkal olcsóbb számítógépeket lehetett készíteni, amelyek üzemeltetése is egyszerűbb és olcsóbb volt. Ekkor kezdték alkalmazni a ferritgyűrűs memóriát is.

Az első integrált áramkört (IC-t) 1958-ban készítette **Jack S. Kilby** a Texas Instrumentsnél és **Robert Noyce** a Fairchild Semiconductornál. A tömegtermelés 1962-ben indult meg, az első integrált áramköröket tartalmazó számítógépek pedig 1964-ben kerültek kereskedelmi forgalomba. Az integrált áramkörök tovább csökkentették a számítógépek árát, méretét és meghibásodási gyakoriságát. Ekkor jelent meg a bájt-szervezés és az input-output processzor, valamint jelentős fejlődés történt a táv-adatátvitelben is.

4.2. Mikroprocesszor

Az első mikroprocesszort[6] 1971-ben **Ted Hoff** (Stanford University mérnöke) tervezte, és az Intel készítette.[7] Egy 7 x 7 mm-es szilíciumlapka 2300 tranzisztort tartalmazott.

Az LSI (Large-Scale Integration) technológia tette lehetővé, hogy a számítógépek központi egysége (CPU) egyetlen félvezető lapkára (IC) kerüljön.

A mikroprocesszoroknak köszönhető, hogy a számítógépek a mindennapi élet részévé váltak. Olcsóságuk, kicsiny méretük hozzájárult a széleskörű elterjedésükhöz.

A jelentősebb Intel processzorok megjelenését és néhány jellemző paraméterét az alábbi táblázat tartalmazza:

Típus	Kiadás éve	Szóhossz [bit]	Órajel [MHz]	Technológia [µm]	Tranzisztor száma
4004	1971	4	0,108	10	$2{,}3 \times 10^3$
8008	1972	8	0,8	10	$3{,}5 \times 10^3$
8080	1974	8	2	6	$4{,}5 \times 10^3$
8088 (PC)	1978	8	5	3	$2{,}9 \times 10^4$
8086 (XT)	1978	16	5	3	$2{,}9 \times 10^4$
80286 (AT)	1982	16	6	1,5	$1{,}34 \times 10^5$
386	1985	32	16	1,5	$2{,}75 \times 10^5$
486	1989	32	25	1	$1{,}2 \times 10^6$
Pentium	1993	32	66	0,8	$3{,}1 \times 10^6$
Pentium Pro	1995	64	200	0,35	$5{,}5 \times 10^6$
Pentium II	1997	64	233	0,35	$7{,}5 \times 10^6$
Celeron	1998	64	266	0,25	$7{,}5 \times 10^6$
Pentium III	1999	64	650	0,25	$9{,}5 \times 10^6$
Pentium 4	2000	64	1500	0,18	42×10^6
Xeon	2001	64	1700	0,18	42×10^6
Pentium M	2003	64	1700	0,09	55×10^6
Core 2 Duo	2006	64	2660	0,065	291×10^6
Atom	2008	64	1860	0,045	47×10^6
Core i5	2010	64	3800	0,032	$1{,}16 \times 10^9$
Core i7	2012	64	2900	0,022	$1{,}4 \times 10^9$

Forrás: www.intel.eu

A 8080-as processzor továbbfejlesztéséből készül a Zilog Z80-as processzor, amely alapja volt például a Spectrum számítógépeknek.

A 8088-as processzor került az első IBM gyártmányú személyi számítógépbe. Ez a PC kategória még 8 bites regiszterekkel dolgozott.

Egy érdekes megfigyelést tehetünk, ha összehasonlítjuk az ENIAC központi egységét és az első mikroprocesszort. A felhasznált aktív elemek (elektroncső, tranzisztor) száma közel azonos, az órajel-frekvenciájuk is azonos nagyságrendű, a méretük viszont – alig 30 év alatt – több tonnáról és több tíz négyzetméterről, néhány milligrammra, illetve 13,5 mm^2-re csökkent.

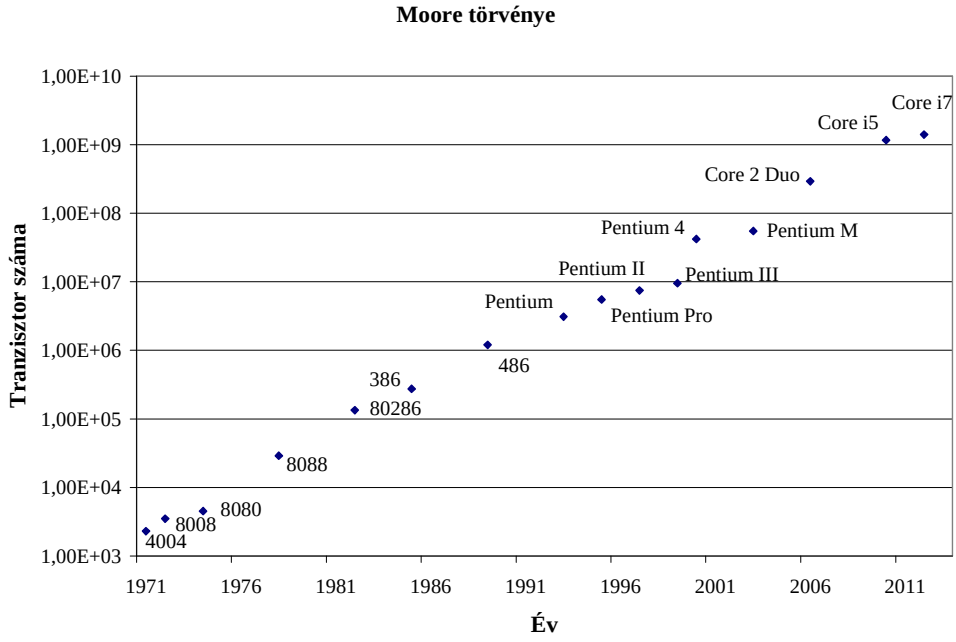
A fejlődés ütemére **Gordon Moore** (az Intel egyik alapítója) 1965-ben adott prognózist, amely szerint az egységnyi területen elhelyezhető áramköri elemek száma évente megduplázódik.[8] Ezt a megállapítását 1975-ben – az addigi tapasztalatok alapján – felülvizsgálta és két évre

[6] Mikroprocesszornak nevezzük az egyetlen félvezető elemen kialakított teljes CPU-t (Central Processing Unit=Központi vezérlő egység), amely egyetlen integrált áramkörös tokban helyezkedik el.

[7] Egyes források szerint a Texas megelőzte az Intelt, ahol ugyanebben az évben készítették el az első mikroprocesszort.

[8] Gordon E. Moore, 1965: Cramming More Components onto Integrated Circuits, Electronics, pp. 114–117.

módosította. Ez évtől kezdve nevezik e tendenciát Moore törvénynek (Moore's law). E fejlődést szemlélteti az alábbi grafikon:



Az ábrát megfigyelve feltűnik, hogy a fejlődés tendenciája lassulást mutat. Ennek két oka van: egyik az, hogy a gyártástechnológia fejlesztése nem tudott lépést tartani a törvényben megfogalmazottal. A másik pedig magából a félvezető technológiából következik, ugyanis a miniatürizálás során kezdik elérni azt a fizikai korlátot (minimális méretet), ahol a tranzisztor már nem működhet.

4.3. Az Intel processzorok regiszterkészlete

A CPU-nak része egy nagysebességű memória, amelyben ideiglenesen tárolódnak a műveletek elvégzéséhez szükséges adatok, eredmények, valamint cím- és vezérlőinformációk.

A regiszterek méretével (bithosszával) jellemezhetjük a processzort. Ennek megfelelően létezik 8, 16, 32, stb. bites központi egység. Minél nagyobb egy regiszter mérete, annál gyorsabb a processzor. Példaként adjunk össze két egész (16 bites) számot. Ha ezt 8 bites processzorral végezzük, akkor a számokat két részletben tudjuk csak összeadni. Először az alsó nyolc bitet, majd a felső nyolcat, hozzáadva az előzőből származó túlcsordulást. Amennyiben 16 bites (vagy nagyobb) processzorunk van, ez a művelet egy utasítással elvégezhető.

A regisztereket funkciójuk szerint csoportosíthatjuk. (A felsorolásban használt regiszternevek az assembly programozásban is használatosak.)

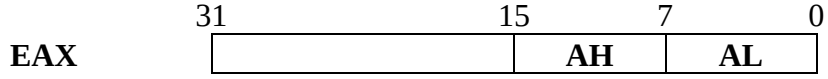
A jegyzetben tárgyalt regiszterek jelentős részét a 8088-as processzortól kezdve használják. A később bevezetett regisztereknél minden esetben megadjuk azt a processzor típust, ahol azt először alkalmazták. Látni fogjuk, hogy a regiszterek nem egyenrangúak, más-más funkciót látnak el, így egymással nem minden esetben helyettesíthetők.

4.3.1. Általános regiszterek

Több célra használható regiszterek, de egyes utasítások végrehajtásánál meghatározott szerepet töltenek be.

Akkumulátorregiszter - Accumulator

A teljes (16 bites) regiszter neve **AX**. A regiszter alsó 8 bitjére külön is lehet hivatkozni **AL** (Accumulator Low) néven, a felső 8 bitjére pedig **AH** (Accumulator High) néven. Minden művelethez használható, kitüntetett szerepe van a szorzás, az osztás és az I/O utasítások végrehajtásánál. A 32 bites változat neve **EAX**, ahol a regiszter alsó 16 bitje megfelel az **AX** regiszternek:



Bázisregiszter - Base

A regiszter alsó 8 bitje **BL** (Base Low), a felső 8 bitje **BH** (Base High). Néhány kivételtől eltekintve minden művelethez használható. Általában az adatszegmensben[9] tárolt adatok báziscímét tartalmazza, indirekt címzésnél használható. (A 32 bites változat neve: **EBX**, felépítése hasonló az **EAX**-hez**)**

[9] Az adatszegmens leírásával az 5.4 fejezet foglalkozik.

Számlálóregiszter - Counter

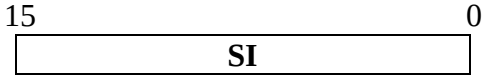
A regiszter alsó 8 bitje **CL** (Counter Low), a felső 8 bitje **CH** (Counter High). Néhány kivételtől eltekintve minden művelethez használható. Általában ciklus, léptető, forgató és sztring utasítások ciklusszámlálója. (A 32 bites változat neve: **ECX**, felépítése hasonló az **EAX**-hez**)**

Adatregiszter - Data

A regiszter alsó 8 bitje **DL** (Data Low), a felső 8 bitje **DH** (Data High). Minden művelethez használható, kitüntetett szerepe van a szorzás, az osztás és az I/O utasítások végrehajtásánál. (A 32 bites változat neve: **EDX**, felépítése hasonló az **EAX**-hez)

4.3.2. Vezérlőregiszterek

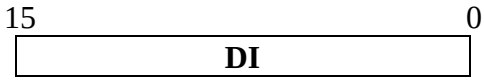
Forrás cím - Source Index



A forrásadat indexelt címzésére.

(A 32 bites változat neve: **ESI)**

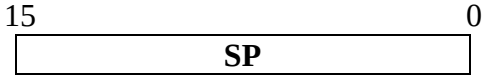
Cél cím - Destination Index



A céladat indexelt címzésére.

A két regiszter segítségével valósítható meg az indirekt és indexelt címzés. Kitüntetett szerepe van a sztring műveletek végrehajtásában. (A 32 bites változat neve: **EDI)**

Stack mutató - Stack Pointer

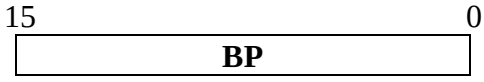


A verembe utolsóként beírt elem címe.

A veremmutató értéke automatikusan változik a stack-műveleteknek megfelelően. Közvetlen utasítással is módosítható, de ez igen nagy gondosságot igényel. (A 32 bites változat neve: **ESP)**

A verem tetején tárolt adat teljes címe az **SS:SP** regiszterpárban található.

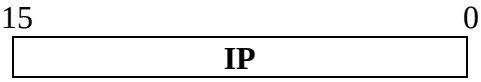
Bázis mutató - Base Pointer



A verem indexelt címzéséhez.

A regiszter használható a stack-szegmens indirekt és indexelt címzésére. Egyéb műveleteknél nem javasolt a használata. (A magasszintű nyelvek függvényhívásakor, a paraméterátadásban és a lokális változók használatakor van szerepe.) (A 32 bites változat neve: **EBP)**

Utasításmutató - Instruction Pointer



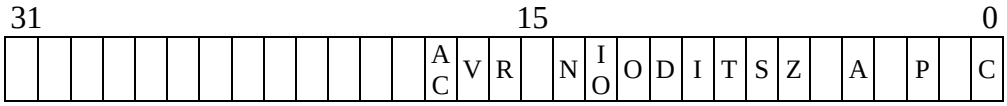
A végrehajtandó utasítás címe.

A processzor működése során az **IP** a következő utasításra mutat. Az utasítás beolvasása (fetch) után az **IP** az utasítás hosszával automatikusan növekszik, ezáltal a beolvasott utasítást követő utasításra mutat. Vezérlésátadás (ugrás, szubrutinhívás) esetén az **IP**-be az új cím íródik. (A 32 bites változat neve: **EIP)**

(Irodalmakban használják még a PC (Program Counter) elnevezést is, bár ez megtévesztő, mert nem az utasításokat számolja.)

Az utasítás teljes memóriacímét a **CS:IP** regiszterpár tartalmazza.

Státuszregiszter - Flags



A státuszregiszter bitjei az utolsó művelet eredményének megfelelően állnak be. Tartalmuk utasítással módosítható és lekérdezhető. A feltételes vezérlőátadó utasítások a bitek állásától függően működnek. (A 386-os processzortól 32 bites!)

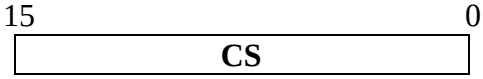
A jelzőbitek (flags) elnevezései:

C – Carry (átvitelbit)	Például bitléptesnél a regiszterből kikerülő bit értéke.
P – Parity (paritásbit)	P=1, ha az eredmény alsó 8 bitjében lévő egyesek száma páros, egyébként P=0.
A – Auxiliary Carry	fél átvitel, átvitel a 3. és a 4. bit között. BCD számokkal végzett műveleteknél használatos.
Z – Zero (zéró bit)	Z=1, ha az eredmény nulla, egyébként Z=0.
S – Sign (előjel bit)	S=0, ha az eredmény pozitív, egyébként S=1.
T – Trap (léptetés)	Utasításonkénti megszakítás engedélyezése nyomkövetéshez.
I – Interrupt	Hardvermegszakítás engedélyezése (I=1) – tiltása (I=0).
D – Direction	Ciklus és sztringkezelő index - nő, ha D=0 és csökken, ha D=1.
O – Overflow	O=1, ha az eredmény nem fér el a célregiszterben.
Kiegészítés 80286-tól:	
IO – I/O Protection Level	2 bites jelző, a privilegizálási szint értéke (0-3). Csak védett módban.
N – Nested Task	Beágyazott taszk jelző. Csak védett módban.
Kiegészítés 80386-tól:	
R – Resume	Hibajelző megszakítás engedélyezése.
V – Virtual 8086	Mode 8086 virtuális üzemmód engedélyezése, védett módban.
Kiegészítés 80486-tól:	
AC – Aligment Check	Illeszkedő memóriahívás engedélyezése.

4.3.3. Szegmensregiszterek (memóriakezelés)

A szegmensregiszterek tárolják a különböző funkciókhoz használt memóriaszegmens-címeket. (A memóriacímzéshez mindig regiszterpárt használunk!)

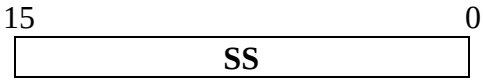
Kódszegmens – Code Segment



Kódszegmens, az utasítások címzéséhez.

Az éppen futó programmodul báziscímét tartalmazza. Minden utasításbetöltés (fetch) a kódszegmens-regisztert használja. Tartalma kiolvasható, de tartalma csak vezérlésátadással módosulhat.

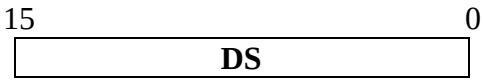
Veremszegmens - Stack Segment



Stack szegmens, a verem címzéséhez.

A stack-ként használt memóriaterület báziscímét tartalmazza. Írható-olvasható, de módosítása nagy figyelmet igényel.

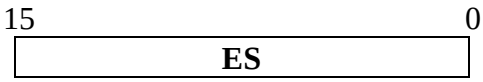
Adatszegmens - Data Segment



Adatszegmens, az adatterület címzéséhez.

Az adatszegmens báziscímét tartalmazza, írható-olvasható regiszter. Sztring műveleteknél – alapértelmezés szerint – a forrás-sztring címét a **DS:SI** regiszterpár tartalmazza.

Extra szegmens - Extra Segment



Extra szegmens, második adatterület címzéséhez.

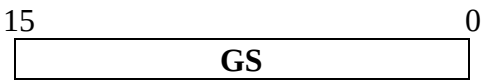
Másodlagos adatszegmens báziscímét tartalmazza, írható-olvasható regiszter. Az **ES** alkalmazásakor egy prefixumot (módosító előtagot) kell használnunk az utasítás előtt. Sztring műveleteknél – alapértelmezés szerint – a cél-sztring címét az **ES:SI** regiszterpár tartalmazza.

A 386-os processzortól további szegmensregisztereket vezettek be:

Extra szegmens - Extra Segment

Extra szegmens, további adatterület címzéséhez.

Extra szegmens - Extra Segment



Extra szegmens, további adatterület címzéséhez.

A 386-os processzoroktól lehetőség van a szegmensregiszterek nélküli, úgynevezett lineáris címzésre is.

4.4. A matematikai társprocesszor

A numerikus adatfeldolgozó processzort (NDP Numeric Data Processor) a 286-os processzor mellett használták először, társprocesszorként (287-es koprocesszor). (A 386-os CPU társprocesszora a 387-es.) Ezt még külön tokba építve lehetett a számítógépben elhelyezni. A 486-os processzor az első olyan Intel processzor, amely lebegőpontos egységet tartalmaz.

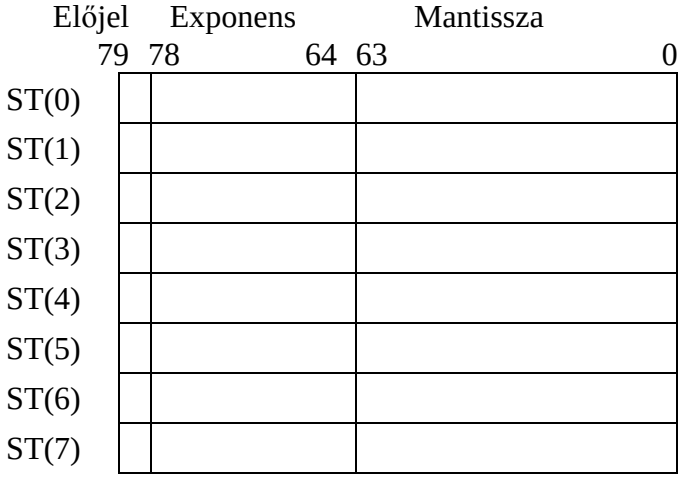
A matematikai társprocesszor alkalmazásával egy újabb lehetőség nyílt a párhuzamos művelet-végrehajtásra. Amikor a programban lebegőpontos utasítás van, akkor a CPU átadja azt az NDP-nek, majd folytatja a programvégrehajtást. Természetesen, ha az NDP által szolgáltatott eredményre szükség van, akkor azt meg kell várnia.

A matematikai processzor a lebegőpontos műveletek végrehajtásához általában saját regisztereit használja. E regisztereket – eltérően a már bemutatott regiszterektől – veremként címezzük. A verem tetejére (aktuális elemére) az állapot regiszter hivatkozik. A regiszterekben 8 darab 10 bájtos lebegőpontos számot tudunk tárolni.

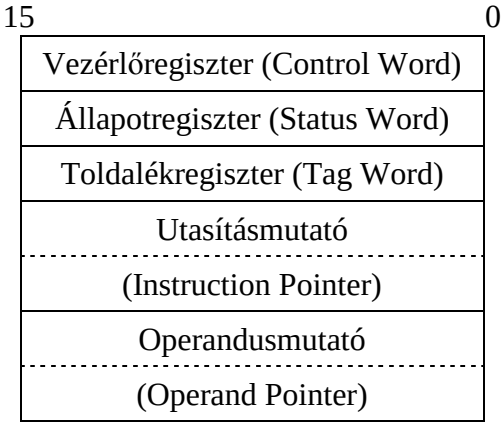
Ezen kívül a koprocesszor úgynevezett 16 bites kontrolregisztereket is tartalmaz.

4.4.1. A koprocesszor regiszterkészlete

Adatregiszterek:



Kontrol regiszterek:



Az utasításmutató-regiszterbe kerül az EIP regiszter tartalma. Operandusmutató tartalmazza a lebegőpontos utasítás által használt – memóriában lévő – operandus eltolás (offset) címét.

Vezérlőregiszter:

A vezérlő regiszter segítségével befolyásolhatjuk a koprocesszor működését.

X	X	X	IC	RC	PC	X	X	PM	UM	OM	ZM	DM	IM

IM – Invalid Operation Mask (Érvénytelen művelet)

DM – Denormal Operand Mask (Nem normalizált operandus)

ZM – Zero Divide Mask (Nullával történő osztás)

OM – Owerflow Mask (Túlcsordulás)

UM – Underflow Mask (Alulcsordulás)

PB – Precision Mask (Pontosság)

PC – Precision Control (Pontosság meghatározása.) A PC határozza meg, hogy az NDP milyen pontosságú számformát használjon a művelethez.

00 – egyszeres pontosság (alapértelmezett)

01 – nem használt

10 – dupla pontosság

11 – kiterjesztett pontosság

RC – Rounding Control (Kerekítés meghatározása) Az RC határozza meg, hogy az NDP hogy illessze az eredményt a megadott pontosságú eredményregiszterbe.

00 – legközelebbi értékre kerekít (alapértelmezett)

01 – lefelé kerekít

10 – felfelé kerekít

11 – 0-ra kerekítés

IC – Infinity Control (Végtelen értelmezése) Csak a 287-es processzorban használták.

1 – pozitív és negatív végtelen is létezik
(A többi NDP csak ezt az üzemmódot használja.)

0 – előjel nélküli (közös) végtelenhez tartanak a számok

Állapotregiszter:

B	C3	ST	C2	C1	C0	ES	SF	PE	UE	OE	ZE	DE	IE

IE – Invalid Operation Exception (megszakítás érvénytelen művelet miatt): egyéb hibák jelzése (pl. aritmetikai, hivatkozási hibák).

DE – Denormal Exception (nem normalizált kivétel[10]): a művelet operandusa nem normalizált.

ZE – Zero Devide Exception (nullával való osztás kivétel): nullával osztás jelzése.

OE – Owerflow Exception (túlcsordulás kivétel): az eredmény nagyobb abszolút értékű, mint amit az NDP ábrázolni tud.

UE – Underflow Exception (alulcsordulás kivétel): az eredmény kisebb abszolút értékű, mint amit az NDP nullától különbözően ábrázolni tud.

PE – Precision Exception (pontosság megszakítás): Ha az NDP az adott pontosságban nem tudja ábrázolni az eredményt. Ez a kivétel fordul elő akkor is, ha kisebb pontosságú adatra konvertálunk.

SF – Stack Fault (veremhiba): verem-túlcsordulás jelzése (287-ben nincs).

ES – Error Summary (hibaösszegzés): értéke 1, ha bármelyik lebegőpontos utasítás hibát talált.

C0 – Status, Carry (túlcsordulás jelzőbit).

C1 – Status, Undefinied (nem definiált jelzőbit): a feltételes vezérlésátadásnál nem használt.

C2 – Status, Parity (parítás jelzőbit).

C3 – Status, Zero (zéró jelzőbit).

ST – Stack pointer (regisztermutató): a legutoljára feltöltött lebegőpontos regiszterre mutat. Ha új szám kerül a verembe, értéke eggyel csökken. Ha kiveszünk egy adatot, akkor eggyel nő.

B – Busy (foglaltságbit): értéke 1, ha az NDP foglalt, vagy ha kivétel történt.

Toldalékregiszter

A regiszter 8 db kétbites információt tartalmaz az ST regiszterekről.

00 – a regiszter érvényes lebegőpontos értéket tárol

01 – a regiszter tartalma 0.0

10 – a regiszterben végtelen, normalizálatlan vagy érvénytelen szám van

11 – a regiszter üres (nincs használatban)

[10] Kivételek (exception): a CPU belső programjának (firmware) megszakítását okozó események.

5. Az Intel processzorok utasításrendszere

Minden számítógép – a Neumann-elvből következik – tartalmaz egy belső (operatív vagy rendszer) memóriát az éppen futó programok (utasítások) és az adatok tárolására. A memória feladata, hogy az információt bináris formában tárolja, és azt – például a processzor kérésére – rendelkezésre bocsássa.

Az Intel mikroprocesszor memóriája bájt szervezésű. Ez azt jelenti, hogy a memória bájtonként címezhető, vagyis minden egyes bájtnak van egy memória címe. Bármely két szomszédos bájt egy 16 bites szót alkot (a szó páros illetve páratlan memóriacímen is kezdődhet). Az Intel processzoroknál a növekvő bájtsorrendet (little endian) alkalmazzák, vagyis az alacsonyabb helyiérték van az alacsonyabb memóriacímen.

Mivel a programutasítások is a memóriában tárolódnak, így ezek tulajdonképpen bináris számok, amelyeket a processzor sorban egymás után beolvas és végrehajt.

A gépi kódú utasítások négy részből állnak, méretük függ az utasítástól és a címzési módtól.

Az általános utasításszerkezet a következő:

Prefixum	Operáció kód	Címzési mód	Operandus

1. A prefixum módosítja az utasítás értelmezését. (Használata nem kötelező.) Használatával előírhatjuk például az ismétlések számát, vagy letilthatjuk a megszakításkérést.
2. Az operáció kód adja meg, hogy a processzornak milyen műveletet kell végrehajtani. Minden utasításban szerepelnie kell!
3. A címzési mód az operandusok – a későbbiekben részletesen tárgyalt – értelmezését adja meg. (Csak operandust tartalmazó utasításokban található.)
4. Az operandus lehet konstans, cím vagy címzéshez használt érték. Hossza változó, de el is maradhat.

5.1. Utasítások

Az assembly utasítások pontosan megfelelnek egy gépi kódú utasításnak, azaz a fordító program (assembler) minden utasítást egy gépi kódra fordít le.

A jegyzetben az Intel processzorok alap utasításkészletével foglalkozunk. (Nem térünk ki a lebegőpontos utasításokra.) Az utasítások részletes leírását „Az Intel processzorok utasításai” mellékletben tárgyaljuk.

5.2. Operandusok és címzési módok

Az Intel processzorok úgynevezett egycímes processzorok, ami azt jelenti, hogy egy utasítással csak egy memóriahelyet tudunk megcímezni. Azaz ha az utasítás két operandusú (például: *MOV*), akkor csak az egyik lehet memóriacím, a másiknak regiszternek kell lennie.

Néhány, elsősorban a *MOV* utasítás segítségével ismerkedjünk meg a különböző operandusokkal és címzési módokkal. (Szintaktikája: *MOV cél, forrás* - A *forrás* tartalmát a *cél* operandusba másolja, a *cél* eredeti értéke elvész, a *forrás* tartalma megmarad.)

Vannak olyan utasítások, amelyeknek nincs operandusuk, vagy maga az utasítás egyértelműen (implicit) meghatározza azokat:

```
RET         ;Visszatérés szubrutinból. nincs operandus
CBW         ;AL-ben lévő értéket AX-be konvertálja (implicit)
```

5.2.1. Regiszteroperandus

Az utasítás paraméterei regiszterek, amelyekben tároljuk a művelet elvégzéséhez szükséges adatokat. (Az operandusokat a regiszter nevével adjuk meg.) 8, 16 vagy 32 bites regiszterekkel végezhető művelet, ennek megfelelően változik az utasítás hossza.

```
MOV AH, AL      ;AL tartalmát AH-ba másolja (8 bites adatmozgatás)
MOV AX, BX      ;BX tartalmát AX-be másolja (16 bites adatmozgatás)
MOV EAX, EBX    ;EAX tartalmát EBX-be másolja (32 bites adatmozgatás)
```

(Az utasítások egy részénél csak meghatározott regiszterek használhatók.)

5.2.2. Közvetlen (immediate) operandus

Ilyenkor az operandust az utasítás kódja tartalmazza.

```
MOV AX, 2       ; AX-be kettőt teszi
```

Ez az utasítás gépi kódra lefordítva B80200h, három egymás utáni bájtot foglal el a memóriában. A B8h (a szám utáni „h” jelzi, hogy az hexadecimális) az utasításkód, ez van a legkisebb című bájton. Ezt közvetlenül a kétbájtnyi adat (02h és 00h) követi. Azért 0200h és nem 0002h, mert az Intel processzorok az alsó bájtot (02h-t) tárolják előbb a memóriában (little endian).

5.2.3. Direkt memóriacímzés

Ebben az esetben az operandus egy előre (a DATA szegmensben) deklarált adat. Ennek a memóriacíme szerepel az utasításban.

```
.DATA
   Adat DB 10h, 11h, 12h, 13h       ;DB jelentése: az adategységünk bájtos
.CODE
      .
      .
   MOV AX, Adat      ;AX-be tölti az Adat címen lévő értéket (10h)
   MOV AX, [Adat]    ;Ua. mint az előző
   MOV AX, Adat[2]   ;AX-be tölti az Adat+2 címen lévő értéket (12h)
   MOV AX, Adat+2    ;Ua. mint az előző
```

Az utasítás gépi kódra lefordítva A10002. Az A1h az utasításkód, amelyet közvetlenül a kétbájtos adatcím követ. (Feltételezve, hogy az adat címe 0200h.) Látható, hogy az előző (immediate operandus) példától csak az operációs kódban különbözik.

Az adat szegmenscímének (alapértelmezésben) a **DS**-ben kell lennie, ezért ezt használat előtt be kell állítani:

```
;A DS esetében csak regiszteroperandus címzési módot választhatunk, így a beállítását
két lépésben kell elvégezni
MOV AX, DGROUP    ;Adatszegmens címe AX-be. A DGROUP a program
                  ;betöltésekor kap értéket az operációs rendszertől.
MOV DS, AX        ;Adatszegmens címe a DS-be.
```

5.2.4. Indirekt memóriacímzés

Itt az operandussal megadott helyen nem az adatot, hanem annak címét találjuk.

5.2.4.1. Regiszter indirekt címzés

```
MOV AX, [BX]        ;AX-be tölti a BX által megcímzett memória tartalmát.
```

Az alapértelmezett adatszegmens címe **DS**. Más szegmensre történő hivatkozás:

```
MOV AX, ES:[BX]   ;AX-be tölti az ES:BX által megcímzett memória tartalmát.
```

Ilyen esetben csak szegmensregiszter használható prefixumként.

5.2.4.2. Indexelt (bázis-relatív) címzés

```
.DATA
   Adat DB 10h, 11h, 12h, 13h
.CODE
        .
        .
   MOV AX, Adat[BX] ;AX-be tölti az Adat+BX címen lévő adatot
```

Ez a címzési mód hasonlít egy tömb elemére történő hivatkozáshoz.

5.2.4.3. Bázis plusz index címzés

```
MOV AX, [BX][DI] ; AX-be tölti a BX+DI címen lévő adatot
```

5.2.4.4. Bázis plusz relatív címzés

```
MOV AX, Adat[BX][DI]     ; AX-be tölti az Adat+BX+DI címen lévő adatot
```

Például többdimenziós tömbök kezelésére használható. Figyelem: kétdimenziós tömb használata esetén a tömb elemeinek egymás utáni elérésekor az egyik regisztert nem egyesével (az adat hosszával) kell változtatni, hanem figyelembe kell venni a tömb (pl. sor) méretét!

5.2.5. A stack (verem) címzése

A stack címzésére a bázisregiszteres címzési mód használható. A szegmenscímet az **SS** regiszter, a báziscímet pedig a **BP** regiszter tartalmazza. (E regisztereket – bár megengedett – nem célszerű másra használni.)

A stack-et ideiglenes adattárolásra és szubrutin híváskor paraméter átadásra használhatjuk. Itt tárolódik – automatikusan – szubrutin hívás esetén a visszatérési cím is.[11] Ezért is fontos, hogy a stack használatakor körültekintően járjunk el.

Példa: Tegyük fel, hogy két egyszavas paramétert adunk át egy szubrutinnak a vermen keresztül. Mivel a szubrutin hívásakor a visszatérési cím (közeli hívás esetén 2, távoli hívás esetén 4

[11] Használatával részletesen *„Az assembly kapcsolata más nyelvekkel”* fejezetben foglalkozunk.

bájt) is a verem tetejére kerül, ezt is figyelembe kell venni. (Most tételezzük fel, hogy közeli hívás történt.) Az átadott paraméterek elérése a hívott szubrutinban:

```
PUSH BP         ;BP eredeti tartalmának mentése
MOV BP, SP      ;A stack tetejének címe BP-be
MOV AX, [BP+4]  ;Az első paraméter betöltése AX-be
MOV BX, [BP+6]  ;A második paraméter betöltése BX-be
POP BP          ;BP eredeti értékének visszaállítása
```

Látható, hogy címzésre a BP regisztert használtuk, és használat után az eredeti értéket visszaállítottuk.

5.2.6. A programterület címzése

Példákban a *JMP cím* feltétel nélküli vezérlőátadó utasítást fogjuk használni. Működése: a program a „*cím*” által megadott helyen folytatódik. (A **CS:IP** utasításregiszterbe íródik a *cím*)

5.2.6.1. IP-relatív címzés

```
JMP SHORT cím    ;Közeli (rövid) ugrás a címre
```

A „SHORT” előtag használata a programozó szándékát fejezi ki a közeli (127 bájton belüli) vezérlésátadásra. Amennyiben az ugrás 127 bájtnál nagyobb, a fordító hibaüzenetet küld.

A feltételes vezérlőátadó utasítások (*JZ, JNZ, JGE,* stb.) mindig IP-relatív címzéssel működnek.

5.2.6.2. Direkt címzés

```
JMP cím          ;Direkt ugrás a szegmensen belüli címre
```

A fordító program mindig IP-relatív hívást alkalmaz a direkt hívás helyett, ha a *cím* távolsága 127 bájton belül van.

```
JMP FAR PTR cím  ;Direkt ugrás a távoli címre (4 bájtos)
```

5.2.6.3. Indirekt címzés

```
JMP BX           ;Ugrás a BX által címzett utasításra
JMP [BX]         ;Ugrás a DS:BX által megadott
                 ;memóriaszóval címzett utasításra
JMP DWORD PTR [BX][SI]
                 ;Ugrás a BX+SI címen lévő 4 bájttal megcímzett
                 ; (távoli) utasításra
```

5.3. A verem (stack)

A verem egy olyan – az operációs rendszer által kijelölt – memóriaterület, ahova közvetlen utasításokkal tudunk adatot elmenteni, és onnan letölteni.

A stack területén tárolhatjuk – átmenetileg – adatainkat, itt tárolódik – szubrutin hívásakor – a visszatérési cím, vagy – magas színtű nyelvek esetében – az eljárások és függvények paraméterei és lokális változói is. Az operációs rendszer is használja ezt a területet, például megszakításokkor ide menti a futtatáshoz szükséges regiszterek tartalmát.

A stack egy **last in – first out** (utolsóként be, elsőként ki) szervezésű memória. Azaz, amit utoljára beletettünk, azt vehetjük ki elsőként. Amikor beteszünk egy adatot a verembe, a veremmutató[12] értéke kettővel[13] csökken, ha kiemelünk egyet, akkor kettővel nő. Így mindig a pointer által mutatott címre írunk (last in), és onnan is olvasunk (first out).

Ez a fajta memóriaszervezés teszi lehetővé a szubrutinok hívását tetszőleges mélységben és a rekurziót is.

A *CALL szubrutin* hívás elmenti a stack tetejére a következő (*CALL* utáni) utasítás címét. Ezt a címet tölti be az **IP**-be (utasításpointerbe) a *RET* utasítás.

A következő fejezetben látni fogjuk, a programunk a memóriába az alacsonyabb memóriacímtől kezdődően töltődik be. A stack kezdete viszont a rendelkezésre álló memória végén helyezkedik el, tehát a legmagasabb memóriacímen.

A processzor regiszterkészletének ismertetésénél láttuk, hogy a verem tetején tárolt (aktuális) adat teljes címe az **SS:SP** regiszterpárban található.

Assembly nyelven a *PUSH forrás* utasítást használjuk adattárolásra, és a *POP cél* utasítást az adatok visszatöltésére a veremből.

A *PUSH* utasítás az **SP** (veremmutató) értékét kettővel csökkenti, majd az operandusban (*forrásban*) tárolt kétbájtos adatot elhelyezi a verem tetején.

A *POP* utasítás kiveszi a verem tetején lévő (kétbájtos adatot), amit a *cél*-lal megadott címre tölt. Ezután az **SP** értékét kettővel növeli.

[12] Veremmutató egy regiszter (változó), amely a verem tetejére, – a verembe - utoljára betett adat memóriacímére mutat.
[13] Az Intel processzorok esetében, a verembe mindig szavas (16 bites) számokat tudunk eltárolni. Mivel ez 2 bájtnyi adatot jelent, így a veremmutató értéke kettővel változik.

5.4. Memóriaszervezés

A számítógépekben a CPU és az operatív memória külön helyezkedik el. Közöttük a kapcsolatot a BUS biztosítja (memóriainterfész). A címbuszon *n* számú címvonal található, így összesen 2^n memóriaelem címezhető meg. A memória lineárisan viselkedik, azaz folyamatos elérést biztosít a *0* kezdőcímtől 2^n-1 címig. Egy lineáris memória esetén a felhasználói program elméletileg a teljes memóriát eléri, annak bármelyik címére adatot írhat, és azt kiolvashatja.

A korai Intel processzorok memóriakezelése eltér ettől a memóriamodelltől. Ezek a processzorok a szegmentált memóriacímzést használják. A 386-os processzortól kezdve alkalmasak a szegmensnélküli címzésre is. Ebben az estben felvetődik az operációs rendszer kompatibilitásának kérdése is.

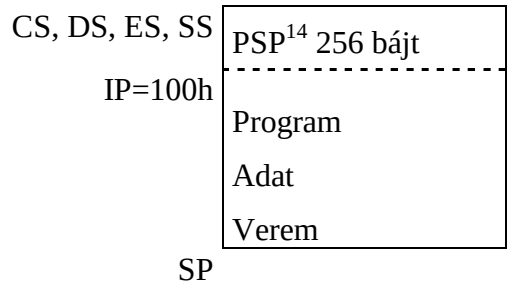
A korábbi mikroprocesszorokban a memória címzésére szolgáló regiszterek 16 bit hosszúak voltak, amelyekkel maximálisan 64 kbájt memória címzésére volt lehetőség. Ezért – a címzéshez – egy újabb, úgynevezett szegmensregisztert is felhasználtak, így a cím két részből, egy szegmens- és egy báziscímből áll. A szegmensen belül a címzés lineáris. Egy szegmens mérete 64 kbájt. A programozó egy szegmensen belül elérhet bármilyen adatot.

A program betöltésekor dől el, hogy a program és az adatok ténylegesen hová kerülnek a memóriába. Ezeket a címeket – később látni fogjuk – programmal le is kérdezhetjük.

A COM állomány felépítése

COM állományokat a felhasználó csak ritkán készít. Általában a memóriarezidens programok ilyenek. Ezeket – a memória jobb kihasználása és a sebesség növelése érdekében – csaknem minden esetben assemblyben írják. E programokkal a jegyzetünkben nem foglalkozunk, csak az összehasonlítás érdekében mutatjuk be.

Memóriaelrendezés:



Látható, hogy a PSP, az utasítások, az adatok és a verem számára összesen egy szegmensnyi hely áll rendelkezésre. Ebből következik, hogy a CS, DS, ES, SS regiszterek értékei megegyeznek.

Mivel a PSP 256 (100h) bájtnyi helyet elfoglal, ezért a programunk a 100h címen fog kezdődni.

[14] PSP: (Program Segment Prefix) az operációs rendszer számára tartalmaz információkat. Ez a terület más célra nem használható.

Amikor az operációs rendszer betölti a COM állományt a memóriába, a következő lépéseket végzi el:

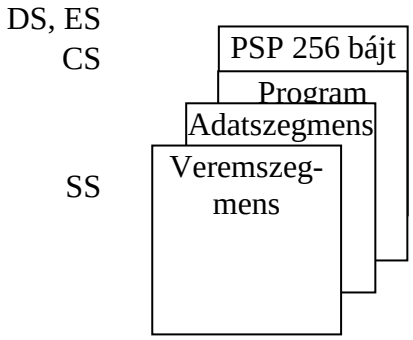
- Létrehozza a 256 bájtos előtétet (PSP). Ide kerül – többek között – a programot indító parancssor is.
- Bemásolja az egész fájlt a háttérről a memóriába, a PSP után.
- Beállítja a DS, ES, SS szegmensregisztereket a PSP elejére.
- Beállítja az SP regisztert a szegmens végére.
- Beállítja a CS regisztert a PSP elejére és az IP regisztert a 100h címre.

Az EXE állomány felépítése

A fentiekből látható, hogy a háttértáron tárolt COM állományt úgy is fel lehet fogni, mint a memóriába betöltött fájl (folyamat) tükörképe, ezért ezeket a programokat (bizonyos eseteket kivéve) egyetlen szegmensbe kell megírni.

Az EXE állományok ezzel szemben sokkal rugalmasabbak, több szegmenst használhatnak, ezáltal nagyobb programok írhatók benne.

Memóriaelrendezés:



Látható, hogy ebben az esetben a program, az adat és a stack külön szegmensen helyezkedik el.

Amikor az operációs rendszer betölti az EXE állományt a memóriába, a következő lépéseket végzi el:

- Létrehozza a 256 bájtos előtétet (PSP). Hasonlóan a COM állományéhoz.
- A fejléc kivételével betölti a PSP utáni memóriába a programot.
- A fejléc alapján megkeresi – a programban – az összes olyan hivatkozást, amelyet át kell helyezni.
- Beállítja a DS, ES szegmensregisztereket a PSP elejére. (Amennyiben a programban használunk adatszegmenst, akkor ezeket programban kell beállítanunk, hogy az adatszegmensre mutassanak.)
- Beállítja az SS-t és az SP-t a fejléc információi alapján.
- Beállítja a CS regisztert a kódszegmens elejére, az IP regisztert pedig a fejlécben megadott ofszet címre.

6. Az assembly programozás

Az Assembly program írása hasonlóan történik a magas szintű nyelvekéhez. Először szükséges egy szövegszerkesztő, amellyel elkészíthetjük a forrásnyelvű (szöveges) változatot, majd ebből a megfelelő fordító- és LINK (TASK) program segítségével a már önállóan is működő, futtatható állományt.

Az Assembly programok fordítására több fordító (assembler) programot fejlesztettek ki (pl. MASM, TASM). Mi a Microsoft által készített Macro Assembler[15] (MASM) programmal ismerkedünk meg.

A programírást a PWB (Microsoft Programmer's WorkBench) keretrendszer segítségével végezzük. E program tartalmaz egy szövegszerkesztő, fordító, linkelő és futtató programrészt, valamint a hibakeresést segítő nyomkövető (debugger) alkalmazást.

Meg kell jegyeznünk, hogy az assembly programok processzor- és operációsrendszer-függőek, így a megírt és lefordított állományunk nem biztos, hogy másik környezetben is működni fog. Erre figyeljünk, amikor olyan programot írunk, amelyet más számítógépeken is szeretnénk használni.

6.1. Szegmentálás

Korábban láttuk, hogy a memóriakezelésnél a szegmentált modellt használjuk, így a cím két részből, egy szegmens- és egy báziscímből áll. A szegmensen belül a címzés lineáris. Egy szegmens mérete 64 kbájt. A programozó egy szegmensen belül elérhet bármilyen adatot.

A program betöltésekor dől el, hogy a program és az adatok ténylegesen hová kerülnek a memóriába. Ezeket a címeket – később látni fogjuk – programmal le is kérdezhetjük.

A szegmentálást az Assembly programunkban is definiálnunk kell.

```
DOSSEG
.MODEL SMALL
.STACK          ;Veremszegmens kijelölése
.DATA           ;Adatszegmens, csak akkor kell megadni
                ;ha tartalmaz adatot
.CODE           ;Kódszegmens, ezután írhatjuk a programutasításokat.
.
.
.
END
```

[15] A MASM bizonyos programozást könnyítő eljárásokat tartalmaz, amelyeket makróknak nevezünk. Makrókat a programozó is definiálhat.

A DOSSEG direktíva jelzi a fordítónak, hogy programszegmenseket egy szigorúan meghatározott rendben akarjuk betölteni, függetlenül a programban megadott sorrendtől.

A .MODEL direktívát annak megadására használjuk, hogy az eljáráshívások NEAR (közeli) vagy FAR (távoli) jellegűek-e. A .MODEL SMALL direktíva megadja az assemblernek, hogy csak egy kódszegmensünk van, és az összes eljáráshívás NEAR. A .MODEL MEDIUM esetén az eljáráshívások – ha nincs külön NEAR-nek definiálva – FAR módúak.

A .STACK direktívát COM állományoknál nem, de EXE állományok esetén meg kell adni.

A .DATA direktívára akkor van szükség, ha adatszegmenst akarunk használni. Például, ha egy magas szintű nyelvben írt programban nem használunk globális változót, akkor nincs szükség az adatszegmensre.

A .CODE direktíva a kódszegmens kezdetét jelzi. (Ezután írhatjuk a programunkat.)

A fordító programnak az END direktíva jelzi a program végét.

A programírás során tetszőlegesen használhatunk kis- és nagybetűket. (Köztük különbség csak a szövegkonstansokban van.)

Az Assembly programozásban a ”;” jelzi a magyarázó (comment) szöveget, ami a sor végéig (ENTER-ig) tart.

6.2. Kezdjünk el programot írni

A program írásának csak úgy van értelme, ha annak valamilyen eredménye van, és azt meg is tudjuk jeleníteni. Amikor tanulunk egy programozási nyelvet, általában az elsők között szerepel az a feladat, hogy írjunk ki valamit a képernyőre. Mivel az Assemblyen nincsenek előre megírt függvények, eljárások, a képernyő kezelését is nekünk kellene megírni. Ez azonban még egy jól felkészült programozót is próbára tenne, hiszen egy komplett monitorkezelőt (drivert) kellene elkészíteni.

Segítséget jelent, hogy az operációs rendszer részét képezik azok a hardverkezelő szoftverek (driverek), amelyeket már telepítettünk a számítógépünkre, így lehetőségünk van az operációs rendszer monitorkezelő programjának használatára.

Az operációs rendszer rutinjainak (nemcsak driver) hívására a szoftvermegszakításokat (interruptokat) használhatjuk. (A megszakítások használatát külön fejezetben tárgyaljuk.)

Elsőnek írjunk ki egy tetszőleges karaktert a képernyőre (használjuk a 21h DOS megszakítást):

```
.MODEL SMALL
.STACK
.CODE

    MOV DL, "A"     ;A kiírandó karakter kódja DL-be
    MOV AH, 2       ;AH-ba a képernyőre írás funkciókódja
    INT 21h         ;Kiírás
    MOV AH, 4Ch     ;AH-ba a visszatérés funkciókódja
    INT 21h         ;Visszatérés az operációs rendszerbe

END
```

Minden programunkat az operációs rendszerbe való visszatérést biztosító utasításokkal kell befejezni!

```
MOV AH, 4Ch
INT 21h
```

Az assembler (fordító) nem tesz különbséget a kis és nagybetű között. Tehát a programunkat írhatjuk kis és nagy betűk használatával, de vegyesen is használhatjuk azokat. A jobb áttekinthetőség érdekében az utasításokat, a regiszterneveket és a makrókat nagybetűvel, a változókat és címeket, pedig kisbetűvel írjuk.

Bővítsük a programunkat: kérjünk be a billentyűzetről egy karaktert, majd írjuk ki a képernyőre azt.

```
.MODEL SMALL
.STACK
.CODE
    MOV AH, 1       ;AH-ba a beolvasás funkciókód
    INT 21h         ;Egy karakter beolvasása, a karakter ASCII kódja AL-be kerül

    MOV DL, AL      ;A kiírandó karakter kódja DL-be
    MOV AH, 2       ; AH-ba a képernyőre írás funkciókódja
    INT 21h         ; Karakter kiírása

    MOV AH, 4Ch     ;Kilépés a programból
    INT 21h

END
```

6.3. Konstansok használata

A fenti programjainkban kétféle szám konstanst használtunk:

MOV AH, ***1*** és a *MOV AH,* ***4Ch.***

Az első esetben decimálisan, a másodikban hexadecimálisan adtuk meg az értékeket.

A számrendszer alapszámát (radix) mindig a szám után írt betűvel jelezzük:

- **Y** vagy **B** – bináris szám,
- **O** vagy **Q** – oktális szám,
- **T** vagy **D** – decimális szám,
- **H** – hexadecimális szám.

(A betűvel kezdődő hexadecimális számoknál a fordítóprogram nem tudja megkülönböztetni azokat a nevektől, ezért ilyenkor a szám elé egy „vezető” nullát írunk. Például: ***0FFh***.)

A fordítóprogram alapértelmezett számrendszere a 10-es, így a decimális számok esetében elhagyhatjuk a betűjelet. Az alapértelmezett számrendszert a ***.RADIX n*** (n: a számrendszer alapja, 2 és 16 közötti érték) direktívával állíthatjuk be.

Ha például, a *.RADIX 16* direktívát kiadjuk, akkor a tizenhatos számrendszer lesz az alapértelmezett, így a hexadecimális számok esetén a **h** jelölést elhagyhatjuk.

Amennyiben a számrendszert 10-nél nagyobbra állítjuk, a decimális és a bináris számoknál nem használhatjuk a **D** illetve **B** betűjelet.

A karakter-konstansok megadásánál használhatjuk a karakter kódját, de a szimbólumát is.

```
MOV AL, 65
MOV AL, ”A”
MOV AL, ’A’
```

Mindhárom esetben az AL regiszterbe a 65 kerül (az „A” ASCII kódja 65).

6.4. Néhány egyszerűbb szubrutin

Annak érdekében, hogy a gyakran használt eljárásokat ne kelljen újra leírni, tegyük azokat alprogramba (szubrutinba). Készítsünk egy karakter beolvasó és egy karakter kiíró szubrutint, majd ezeket egy főprogram segítségével hívjuk meg. Az alprogramok közötti adatátadásra használjuk a **DL** regisztert.

Eljárások elválasztására használnunk kell a PROC és ENDP direktívákat, amelyek közé írjuk a szubrutin utasításait. Alprogramok használatakor a főprogramot is el kell neveznünk, és a két direktíva közé kell tennünk! Az alprogramokat tetszőleges sorrendben írhatjuk.

```
.MODEL SMALL
.STACK
.CODE
main proc               ;Főprogram
    CALL read_char      ;Karakter beolvasása
    CALL write_char     ;Karakter kiírása
    MOV AH,4Ch          ;Kilépés
    INT 21h
main endp
read_char proc          ;Karakter beolvasása. A beolvasott karakter DL-be kerül
    PUSH AX             ;AX mentése a verembe
    MOV AH, 1           ;AH-ba a beolvasás funkciókód
    INT 21h             ;Egy karakter beolvasása, a kód AL-be kerül
    MOV DL, AL          ;DL-be a karakter kódja
    POP AX              ;AX visszaállítása
    RET                 ;Visszatérés a hívó rutinba
read_char endp
write_char proc         ;A DL-ben lévő karakter kiírása a képernyőre
    PUSH AX             ;AX mentése a verembe
    MOV AH, 2           ; AH-ba a képernyőre írás funkciókódja
    INT 21h             ; Karakter kiírása
    POP AX              ;AX visszaállítása
    RET                 ;Visszatérés a hívó rutinba
write_char endp
END main                ;END + a főprogram neve
```

Az eljárásokat mindig „*név + proc*”-cal kezdjük és „*név + endp*”-vel fejezzük be. Amikor több eljárás van a programunkban, meg kell adnunk a betöltőnek (loader-nek), hogy melyik a főprogram. Ezt az END direktíva után írt (példánkban *main*) névvel tehetjük meg.

(A következőkben a már ismertetett szubrutinokat újra nem írjuk le, csak hivatkozunk rájuk.)

Egészítsük ki programunkat egy soremeléssel:

```
.MODEL SMALL
.STACK
.CODE
main proc               ;Főprogram
        CALL read_char  ;Karakter beolvasása
        CALL cr_lf      ;Soremelés
        CALL write_char ;Karakter kiírása

        MOV AH,4Ch      ;Kilépés
        INT 21H
main endp
CR EQU 13               ;CR-be a kurzor a sor elejére kód
LF EQU 10               ;LF-be a kurzor új sorba kód
cr_lf proc
        PUSH DX         ;DX mentése a verembe
        MOV DL, CR
        CALL write_char ;kurzor a sor elejére
        MOV DL, LF
        CALL write_char ;Kurzor egy sorral lejjebb
        POP DX          ;DX visszaállítása
        RET             ;Visszatérés a hívó rutinba
cr_lf endp
END main
```

Itt egy újabb direktívával, az EQU-val ismerkedtünk meg. Segítségével a számokat nevekkel (konstansokkal) helyettesíthetjük. Helyette használhatjuk az „=” jelet is. (A *CR EQU 13* megfelel a *CR = 13* kifejezésnek.) A szubrutinban használt *MOV DL, CR* egyenértékű a *MOV DL, 13* utasítással.

A konstansok fordítási időben kerülnek behelyettesítésre, ellentétben a magas szintű nyelvekben használt változókkal, amelyek futási időben kapnak értéket.

Most bővítsük szubrutinkészletünket egy bináris, egy decimális és egy hexadecimális konvertáló és kiíró elemmel.

Feladat: kérjünk be egy karaktert a billentyűzetről, majd írjuk ki a karakter kódját binárisan, decimálisan és hexadecimálisan.

```
.MODEL SMALL
.STACK
.CODE
main proc                ;Főprogram
        CALL read_char   ;Karakter beolvasása
        XOR DH, DH       ;DH törlése
        CALL cr_lf       ;Soremelés
        CALL write_binary ;Karakterkód konvertálása bináris számmá és kiírása a képernyőre
        CALL cr_lf       ;Soremelés
        CALL write_decimal ; Karakterkód konvertálása decimális számmá és kiírása a képernyőre
        CALL cr_lf       ;Soremelés
        CALL write_hexa  ;Konvertálás hexadecimális számmá és kiírása a képernyőre
        MOV AH,4Ch       ;Kilépés
        INT 21h
main endp
END main
```

A DH regiszter törlésével egészítettük ki főprogramunkat. Korábban már láttuk, hogy a számítógép minden műveletet logikai műveletekké alakít át. Ezért – ahol csak lehet – mi is inkább logikai utasításokat használjunk, amelyeket nemcsak gyorsabban hajt végre a processzor, de kisebb az utasítás helyfoglalása is. A törlést elvégezhettük volna a *MOV DH, 0* utasítással is, ez azonban három bájtot foglal el, míg az *XOR DH, DH* csak kettőt.

A bináris kiíráskor a regiszter bitjeit kell kiírni, sorban egymás után, balról jobbra. Ehhez kihasználhatjuk a bitforgató (rotáló) utasítások azon tulajdonságát, hogy a regiszterből kilépő bit a CF-be (carry-bit) kerül. Tehát a regiszter tartalmát nyolcszor egymás után rotáljuk balra, és a CF tartalmának megfelelően 1-et vagy 0-át írunk ki. Azaz, ha a CF-ben 0 van, akkor a "0" (ASCII 48) karaktert, 1 esetén a „1” (ASCII 49) karaktert kell megjeleníteni.

Használjuk az *ADC DL, "0"* utasítást, amely összeadja a DL tartalmát, 48-at (a "0" kódját) és a CF bitet, az eredmény a DL-be kerül. Ha eredetileg DL = 0, akkor az összeadás után 48 lesz az értéke, ha CF = 0, és 49 lesz, ha CF = 1.

A ciklus megírásához használjuk a *LOOP cím* számlálóciklust. A processzor a LOOP hatására, az irányjelző bit (DF) értékének megfelelően csökkenti vagy növeli a CX regiszter tartalmát. Amennyiben a CX értéke nem nulla, a program a *cím*-en folytatódik, ellenkező esetben a következő utasításon.

```
write_binary proc       ;kiírandó adat a DL-ben
    PUSH BX             ;BX mentése a verembe
    PUSH CX             ;CX mentése a verembe
    PUSH DX             ;DX mentése a verembe
    MOV BL, DL          ;DL másolása BL-be
    MOV CX, 8           ;Ciklusváltozó (CX) beállítása
binary_digit:
    XOR DL, DL          ;DL törlése
    RCL BL, 1           ;Rotálás balra eggyel, kilépő bit a CF-be
    ADC DL, "0"         ;DL = DL + 48 + CF
    CALL write_char     ;Bináris digit kiírása
    LOOP binary_digit   ;Vissza a ciklus elejére
    POP DX              ;DX visszaállítása
    POP CX              ;CX visszaállítása
    POP BX              ;BX visszaállítása
    RET                 ;Visszatérés a hívó rutinba
write_binary endp
```

Figyeljük meg, hogy a szubrutinokban elmentjük azon regiszterek tartalmát, amelyeket a rutinban használunk. Ez azért célszerű, mert így megőrizzük az eredeti tartalmat, és a híváskor nem kell arra figyelnünk, hogy a regiszter esetleg megváltozik.

Mivel a STACK-re mentünk, a visszatöltést fordított sorrendben kell elvégezni. (Amit utoljára elmentettünk, azt vehetjük ki először.)

A decimális konvertáló elkészítése előtt írjuk meg a hexadecimális átalakítót, mert annak egy részét használni fogjuk a decimális konvertáláskor.

Egy hexadecimális helyi érték (digit) négybites bináris számnak felel meg, tehát egy bájt két hexadecimális digittel írható le. Első lépésként válasszuk ketté a DL tartalmát: alsó illetve felső négy bitre, majd az így kapott értékekkel külön-külön végezzük el a konvertálást. Ehhez egy új (*write_hexa_digit*) alprogramot készítünk.

Először a felső négy bitre van szükségünk (eltolva az alsó négy bit helyére), amit úgy kapunk meg, hogy a DL tartalmát négy bittel jobbra mozgatjuk (osztás 16-tal). Az *SHR DL, CL* utasítással a DL tartalmát CL (4) szer jobbra shift-eljük, így a felső négy bit alulra csúszik, helyükbe pedig nulla kerül.

A második digit konvertálásánál a szám már jó helyen (alsó négy biten) van, csak a felső bitek tartalmát kell kitörölni. Ezt végezhetjük el az *AND DL, 0Fh* utasítással (maszkolás).

```
write_hexa proc          ;A DL-ben lévő két hexa számjegy kiírása
    PUSH CX              ;CX mentése a verembe
    PUSH DX              ;DX mentése a verembe
    MOV DH, DL           ;DL mentése
    MOV CL, 4            ;Shift-elés száma CX-be
    SHR DL, CL           ;DL shift-elése 4 hellyel jobbra
    CALL write_hexa_digit ;Hexadecimális digit kiírása
    MOV DL, DH           ;Az eredeti érték visszatöltése DL-be
    AND DL, 0Fh          ;A felső négy bit törlése
    CALL write_hexa_digit ;Hexadecimális digit kiírása
    POP DX               ;DX visszaállítása
    POP CX               ;CX visszaállítása
    RET                  ;Visszatérés a hívó rutinba
write_hexa endp
```

A hexadecimális digit konvertálásánál meg kell vizsgálnunk, hogy a szám 10 alatti-e, azaz decimális számjeggyel le tudjuk-e írni. Ekkor – ahogy a bináris átalakítónál már láttuk – a "0" kódjához hozzáadjuk a bináris értéket. Amennyiben a szám nagyobb vagy egyenlő, mint tíz, akkor ehhez még hozzá kell adni az "A" és a "0" kódja közötti különbséget és le kell vonni 10-et.

```
write_hexa_digit proc
    PUSH DX              ;DX mentése a verembe
    CMP DL, 10           ;DL összehasonlítása 10-zel
    JB non_hexa_letter   ;Ugrás, ha kisebb 10-nél
    ADD DL, "A"-"0"-10   ;A – F betűt kell kiírni
non_hexa_letter:
    ADD DL, "0"          ;Az ASCII kód megadása
    CALL write_char      ;A karakter kiírása
    POP DX               ;DX visszaállítása
    RET                  ;Visszatérés a hívó rutinba
write_hexa_digit endp
```

A decimális konverziót úgy végezzük, mint ahogy a számrendszerek közötti váltást. A számot elosztjuk tízzel (a számrendszer alapszámával), majd az eredménnyel addig végezzük az osztást, amíg az eredmény nulla nem lesz. A maradékokat tároljuk, amelyek megadják az új (tízes) számrendszerbeli számjegyeket (a legkisebb helyiértéktől kezdve).

Az eljárásban a verembe gyűjtjük a maradékokat, majd egy ciklus segítségével visszaolvassuk (fordított sorrenben kell kiírni őket, mint ahogy keletkeznek), illetve kiírjuk azokat a már megírt

write_hexa_digit szubrutinnal. Az osztások számlálásához a CX regisztert használjuk, így a visszaolvasásnál nem kell külön törődnünk a ciklusváltozó kezdőértékével.

```
write_decimal proc
        PUSH AX             ;AX mentése a verembe
        PUSH CX             ;CX mentése a verembe
        PUSH DX             ;DX mentése a verembe
        PUSH SI             ;SI mentése a verembe
        XOR DH, DH          ;DH törlése
        MOV AX, DX          ;AX-be a szám
        MOV SI, 10          ;SI-ba az osztó
        XOR CX, CX          ;CX-be kerül az osztások száma
decimal_non_zero:
        XOR DX, DX          ;DX törlése
        DIV SI              ;DX:AX 32 bites szám osztása SI-vel, az eredmény AX-
                              be, a maradék DX-be kerül
        PUSH DX             ;DX mentése a verembe
        INC CX              ;Számláló növelése
        OR AX, AX           ;Státuszbitek beállítása AX-nek megfelelően
        JNE decimal_non_zero ;Vissza, ha az eredmény még nem nulla
decimal_loop:
        POP DX              ;Az elmentett maradék visszahívása
        CALL write_hexa_digit ;Egy decimális digit kiírása
        LOOP decimal_loop
        POP SI              ;SI visszaállítása
        POP DX              ;DX visszaállítása
        POP CX              ;CX visszaállítása
        POP AX              ;AX visszaállítása
        RET                 ;Visszatérés a hívó rutinba
write_decimal endp
```

Bővítsük a szubrutin gyűjteményünket – különböző számrendszerekben történő – adatbeírás lehetőségével. Írjuk meg a fenti kiíró rutinok beolvasó párjait: a decimális, a hexadecimális és a bináris beolvasást.

A rutinokban nem végzünk ellenőrzést, tehát nem vizsgáljuk, hogy adatmegadáskor csak szám karaktert adunk-e meg, vagy a hexadecimális számoknál csak a megfelelő betűt használjuk-e. A konverziót 1 bájtra végezzük, tehát az adatmegadás során erre is tekintettel kell lennünk!

A beolvasó rutinok működésének alapelve a következő: a billentyűzetről karaktereket olvasunk be sorban, egymás után. Először a magasabb, majd az egyre alacsonyabb helyi értékűeket, az ENTER-ig.

A számkarakter átalakításánál kihasználhatjuk, hogy az ASCII kódtáblában a számok kódjai egymás után következnek. A „0” kódja 48, az „1” kódja 49 …. a „9” kódja 57. Ha a kódokból kivonunk 48-at („0” kódját), akkor a szimbólumnak megfelelő numerikus értéket kapjuk: ”0” - ”0” = 0; ”1” - ”0” = 1 …. ”9”- ”0” = 9. Ezután a számokat helyi értéküknek megfelelően kell eltárolni. Vesszük a legmagasabb helyen lévő számjegyet, amit eggyel magasabb helyi értékre tolunk (megszorozzuk a számrendszer alapszámával), majd hozzáadjuk a következő értéket. Ezt addig folytatjuk, amíg van alacsonyabb helyi értékű számjegy.

Nézzük ezt egy példa segítségével: legyenek a beírt karakterek 2, 1, 3.

Szorozzuk meg a legmagasabb helyi értékű számjegyet (2) tízzel, majd adjuk hozzá a következő értéket.

$$2 * 10 + 1 = 21$$

Az így kapott értéket (ha van még új szám) újra megszorozzuk tízzel, majd hozzáadjuk a következő értéket.

$$21 * 10 + 3 = \mathbf{213}$$

Ezután már könnyen megírhatjuk rutinjainkat.

```
read_decimal proc
    PUSH AX             ;AX mentése a verembe
    PUSH BX             ;BX mentése a verembe
    MOV BL, 10          ;BX-be a számrendszer alapszáma, ezzel szorzunk
    XOR AX, AX          ;AX törlése
read_decimal_new:
    CALL read_char      ;Egy karakter beolvasása
    CMP DL, CR          ;ENTER ellenőrzése
    JE read_decimal_end ;Vége, ha ENTER volt az utolsó karakter
    SUB DL, "0"         ;Karakterkód minusz ”0” kódja
    MUL BL              ;AX szorzása 10-zel
    ADD AL, DL          ;A következő helyi érték hozzáadása
    JMP read_decimal_new ;A következő karakter beolvasása
read_decimal_end:
    MOV DL, AL          ;DL-be a beírt szám
    POP BX              ;AB visszaállítása
    POP AX              ;AX visszaállítása
    RET                 ;Visszatérés a hívó rutinba
read_decimal endp
```

Ennek alapján elkészíthetjük a hexadecimális beolvasó rutinunkat.

Itt az alapszám (szorzó) 16 (10h) lesz. A konvertálásnál figyelembe kell venni, hogy a betűkarakterek ASCII kódja nem közvetlenül a számoké után következik, ezért külön kell választanunk a számjegy és a betű karaktereket. Számjegy esetén a szubrutin ugyanúgy működik, mint a decimális beolvasásnál. Betű karakternél a kódból – a ”0” kódján kívül – még 7-et le kell vonni. (7 a különbség a ”9” és az ”A” kódja között.)

```
read_hexa proc
    PUSH AX              ;AX mentése a verembe
    PUSH BX              ;BX mentése a verembe
    MOV BL, 10h          ;BX-be a számrendszer alapszáma, ezzel szorzunk
    XOR AX, AX           ;AX törlése

read_hexa_new:
    CALL read_char       ;Egy karakter beolvasása
    CMP DL, CR           ;ENTER ellenőrzése
    JE read_hexa_end     ;Vége, ha ENTER volt az utolsó karakter

    CALL upcase          ;Kisbetű átalakítása naggyá
    SUB DL, "0"          ;Karakterkód minusz ”0” kódja
    CMP DL, 9            ;Számjegy karakter?
    JBE read_hexa_decimal ;Ugrás, ha decimális számjegy
    SUB DL,7             ;Betű esetén még 7-et levonunk
read_hexa_decimal:
    MUL BL               ;AX szorzása az alappal
    ADD AL, DL           ;A következő helyi érték hozzáadása
    JMP read_hexa_new    ;A következő karakter beolvasása

read_hexa_end:
    MOV DL, AL           ;DL-be a beírt szám
    POP BX               ;BX visszaállítása
    POP AX               ;AX visszaállítása
    RET                  ;Visszatérés a hívó rutinba

read_hexa endp
```

A hexadecimális adatok megadásánál azt akarjuk, hogy ne kelljen megkülönböztetni a kis- és nagybetűket, azaz az értékeket adhassuk meg mindkét formában. Ennek érdekében készítünk egy kisbetű - nagybetű konvertáló eljárást.

Először vizsgáljuk meg, hogy a karakter kisbetű-e. Ha nem, akkor nem csinálunk vele semmit, ha igen, akkor a kódjából kivonjuk a kisbetűk és a nagybetűk kódjai közötti különbséget (”a” - ”A”).

```
upcase proc             ;DL-ben lévő kisbetű átalakítása nagybetűvé
    CMP DL, "a"         ;A karakterkód és ”a” kódjának összehasonlítása
    JB upcase_end       ;A kód kisebb, mint ”a”, nem kisbetű
    CMP DL, "z"         ;A karakterkód és ”z” kódjának összehasonlítása
    JA upcase_end       ;A kód nagyobb, mint ”z”, nem kisbetű
    SUB DL, "a"-"A"     ;DL-ből a kódok különbségét
upcase_end:
    RET                 ;Visszatérés a hívó rutinba
upcase endp
```

Figyeljük meg, hogy a rutinok nevei utalnak annak funkciójára (beszédes nevek). Ez a módszer megkönnyíti a programlisták olvasását, értelmezését.

Az alprogramokban használ címkék (ugrási helyek) nevei nemcsak a működésre, de az azt tartalmazó szubrutin nevére is utalnak. Ezzel a módszerrel elkerülhetjük, hogy állandóan arra kelljen figyelnünk, hogy van-e már olyan címke, amit létre akarunk hozni. **Két azonos nevű címke nem lehet egy programban belül!**

Készítsük el a bináris beolvasót is.

Már korábban láttuk, hogy a bináris számok kettővel (alapszámmal) való szorzása úgy is elvégezhető, hogy eggyel balra léptetünk (shiftelünk).

```
read_binary proc
    PUSH AX             ;AX mentése a verembe
    XOR AX, AX          ;AX törlése
read_binary_new:
    CALL read_char      ;Egy karakter beolvasása
    CMP DL, CR          ;ENTER ellenőrzése
    JE read_binary_end  ;Vége, ha ENTER volt az utolsó karakter
    SUB DL, "0"         ;Karakterkód minusz ”0” kódja
    SAL AL, 1           ;Szorzás 2-vel, shift eggyel balra
    ADD AL, DL          ;A következő helyi érték hozzáadása
    JMP read_binary_new ;A következő karakter beolvasása
read_binary_end:
    MOV DL, AL          ;DL-be a beírt szám
    POP AX              ;AX visszaállítása
    RET                 ;Visszatérés a hívó rutinba
read_binary endp
```

Készítsük el a 3.4.5. fejezetben ismertetett osztás algoritmusa alapján az osztó programunkat. Használjuk az eddig elkészített szubrutinokat!

Legyen az osztandó az **AL**, az osztó a **BL** regiszter, a maradékot a **DH**, míg a hányadost a **DL** regiszterben tároljuk.

```
.MODEL SMALL
.STACK
.CODE
main proc                ;Főprogram
    CALL read_decimal    ;Osztandó beolvasása
    MOV AL, DL
    CALL cr_lf
    CALL read_decimal    ;Osztó beolvasása
    MOV BL, DL
    CALL cr_lf
    XOR DX, DX           ;DH (maradék), DL (hányados) törlése
    MOV CX, 8            ;Ciklusszám
Cycle:
    SHL AL,1             ;Osztandó eggyel balra, CR-be a kilépő bit
    RCL DH,1             ;Maradék eggyel balra, belép a CR tartalma
    SHL DL,1             ;Hányados eggyel balra
    CMP DH, BL
    JB Next              ;A maradék kiseb,mint az osztó
    SUB DH, BL           ;Az osztó kivonása a maradékból
    INC DL               ;Hányados növelése
Next:
    LOOP Cycle
Stop:
    CALL write_decimal   ;Hányados (DL) kiírása
    CALL cr_lf           ;Soremelés
    MOV DL, DH
    CALL write_decimal   ;Maradék (CL) kiírása
    MOV AH,4Ch           ;Kilépés
    INT 21h
main endp
END main
```

6.5. Az adatszegmens használata

Tegyünk egy karaktert az adatszegmensbe, majd írjuk ki azt a képernyőre:

```
.MODEL SMALL
.STACK
.DATA
adat DB 65              ;Egy bájt („A” kódjának) elhelyezése az adatszegmensbe
.CODE
main proc
    MOV AX, DGROUP      ;Adatszegmens helyének lekérdezése
    MOV DS, AX          ;DS beállítása, hogy az adatszegmensre mutasson
    MOV DL, adat        ;Az adat betöltése DL-be
    CALL write_char     ;Karakter kiírása
    MOV AH,4Ch          ;Visszatérés az operációs rendszerbe
    INT 21h
main endp
```

Az adatszegmensben elhelyezett adatunkat a címe alapján tudjuk megkeresni, illetve azonosítani. Esetünkben e cím neve *adat*. (Mivel ez az első definiált adat, így a cím értéke 0.)

A DB az adattípust definiáló direktíva, amit később részletesen tárgyalunk.

A program írásakor előre nem tudjuk megmondani, hogy a memóriában – betöltéskor – hová kerül az adatszegmens. Ez függ attól, hogy milyen programok futnak már a számítógépen, illetve hol van szabad hely a memóriában. Mivel az adatszegmens helye csak a betöltéskor válik véglegessé, szükségünk van egy DGROUP mutatóra, ahová az operációs rendszer beírja az adatszegmens címét. Ezt az adatszegmens használatakor mindig be kell tölteni a DS regiszterbe.

Mivel a memóriából közvetlenül nem tölthetünk be adatot a DS-be, így ezt két lépésben kell megtennünk: először AX-be tesszük a DGROUP-un lévő értékét, majd ezt írjuk a DS-be.

Egy másik megoldás:

```
.MODEL SMALL
.STACK
.DATA
adat DB "A"             ;Egy bájt („A” kódjának) elhelyezése az adatszegmensre
.CODE
main proc
    MOV AX, DGROUP      ;Adatszegmens helyének lekérdezése
    MOV DS, AX          ;DS beállítása, hogy az adatszegmensre mutasson
    LEA BX, adat        ;Az adat offset címének betöltése BX-be
    MOV DL, [BX]        ;DL-be tölti a BX-el címzett memória tartalmát
    CALL write_char     ;Karakter kiírása
    MOV AH,4Ch          ;Visszatérés az operációs rendszerbe
    INT 21h
main endp
```

Itt a BX regiszteren keresztül címeztük meg az adatot (indirekt címzés). Ezt a megoldást akkor célszerű alkalmazni, amikor egy változó névhez nemcsak egy érték kapcsolódik, például tömbök vagy sztringek használatakor. Nézzünk erre is egy példát.

A következő feladatban írjunk ki a képernyőre egy tetszőleges szöveget.

Bármilyen programozási nyelven is dolgozunk, amikor sztringet (szövegsort) használunk, meg kell adnunk a szöveg kezdetét és végét. A szöveg kezdete minden esetben egy memóriacím, esetünkben ez az „*adat*” nevű mutató. A szöveg végét a programozási nyelvek eltérően kezelhetik, az Assembly-ben mi dönthetjük el, hogy a sztringeket hogyan zárjuk le. Azonban szokás – a C nyelvhez hasonlóan – a szöveget egy bináris nullával befejezni. Ezt figyelembe véve írjuk meg programunkat.

```
.MODEL SMALL
.STACK
.DATA
adat DB "Ez egy tetszőleges szöveg,",10,13,"ami több soros is lehet.",10,13,0
.CODE
main proc
    MOV AX, DGROUP      ;Adatszegmens beállítása
    MOV DS, AX
    LEA BX, adat        ;Az ADAT címe BX-be
new:
    MOV DL, [BX]        ;DL-be egy karakter betöltése
    OR DL, DL           ;Státuszbitek beállítása DL-nek megfelelően
    JZ stop             ;Kilépés a ciklusból, ha DL=0
    CALL WRITE_CHAR     ;Egy karakter kiírása
    INC BX              ;BX növelése, BX a következő karakterre mutat
    JMP new             ;Vissza a ciklus elejére
stop:
    MOV AH,4Ch          ;Kilépés
    INT 21h
main endp
```

Az *adat* változónév után szereplő *DB* direktíva azt jelzi, hogy adatunk bájtos.

Az adattípus megadásához a következő direktívák használhatók:

BYTE, DB	előjel nélküli bájt	0 – 255
SBYTE	előjeles bájt	-128 – 127
WORD, DW	előjel nélküli szavas (2 bájtos)	0 – 65535 (64k)
SWORD	előjeles szavas	-32768 (32k) – 32767 (32k)
DWORD, DD	előjel nélküli duplaszavas	0 – 4G
SDWORD	előjeles duplaszavas	-2G – 2G
FWORD, DF 6 bájtos	csak 386-os processzortól	
QWORD, DQ	8 bájtos	koprocesszor utasítás
TBYTE, DT	10 bájtos	

Látható, hogy az adatmegadásnál nincs szükség az adat hosszának definiálására. Amennyiben karaktersorozatot adunk meg, azt megtehetjük idézőjelek között egyszerre, vagy karakterenként, vesszővel elválasztva. Amikor numerikus értékekkel töltjük fel az adatterületet, azokat vesszővel választjuk el egymástól. A programunkban a szöveget folyamatosan, a soremeléseket és a zárókódot numerikusan adtuk meg.

A következő példánkban többféleképpen megadott szöveget írjunk ki.

Sztring kiírására írjunk szubrutint, amely „bemenő paramétere” a BX regiszter, azaz a BX-ben adjuk át a kiírandó karaktersorozat kezdőcímét.

```
.MODEL SMALL
.STACK
.DATA
adat_1 DB       "Ez az első sor",10,13,0
adat_2 DB       "E","z"," ","a"," ","m","á","s","o","d","i","k",10,13,0
adat_3 DB       69,122,32,97,32,104,97,114,109,97,100,105,107,10,13,0
.CODE
main proc
    MOV AX, DGROUP      ;Adatszegmens beállítása
    MOV DS, AX
    LEA BX, adat_1      ;Az adat_1 címe BX-be
    CALL write_string   ;Kiírás
    LEA BX, adat_2      ;Az adat_2 címe BX-be
    CALL write_string   ;Kiírás
    LEA BX, adat_3      ;Az adat_3 címe BX-be
    CALL write_string   ;Kiírás
    MOV AH,4Ch          ;Kilépés
    INT 21h
main endp
```

```
write_string proc          ;BX-ben címzett karaktersorozat kiírása 0 kódig.
    PUSH DX                ;DX mentése a verembe
    PUSH BX                ;BX mentése a verembe
write_string_new:
    MOV DL, [BX]           ;DL-be egy karakter betöltése
    OR DL, DL              ;DL vizsgálata
    JZ write_string_end    ;0 esetén kilépés
    CALL write_char        ;Karakter kiírása
    INC BX                 ;BX a következő karakterre mutat
    JMP write_string_new   ;A következő karakter betöltése
write_string_end:
    POP BX                 ;BX visszaállítása
    POP DX                 ;DX visszaállítása
    RET                    ;Visszatérés
write_string endp
```

A következő példánkban az adatszegmens megadásakor több újdonságot is megfigyelhetünk.

Az *n DUP (val)* operátor segítségével *n* darab *val* értéket helyezünk el az adatszegmensre.

A .DATA utáni ? azt jelenti, hogy olyan változókat akarunk definiálni, amelyeknek nincs kezdeti értéke. Ezért nem is kell, hogy a programállományban (pl. az EXE fájlban) helyet foglaljon. Az adatterületre csak a program betöltése után van szükségünk. Ha a .DATA? után valamilyen konkrét értéket (pl. *adat DB ”A”*) definiálunk, akkor az assembler az összes változónak helyet foglal az EXE állományban. Ezért a kezdeti értékkel rendelkező változókat a .DATA, a többit pedig a .DATA? részben kell definiálni DUP (?) segítségével.

Írjunk egy karaktersorozat-beolvasó alprogramot. A beolvasott sztring visszaírására használjuk a már elkészített szubrutint. A tárolóhely kezdőcímének megadására itt is a BX regisztert használjuk.

```
.MODEL SMALL
.STACK
.DATA?
adat DB 100 DUP (?)
.CODE
main proc
        MOV AX, DGROUP      ;Adatszegmens beállítása
        MOV DS, AX
        LEA BX, adat        ;Az adatterület címe BX-be

        CALL read_string    ;Karaktersorozat beolvasása
        CALL cr_lf          ;Soremelés
        CALL write_string   ;Karaktersorozat visszaírása

        MOV AH,4Ch          ;Kilépés
        INT 21h
main endp
read_string proc
        PUSH DX             ;DX mentése a verembe
        PUSH BX             ;BX mentése a verembe

read_string_new:
        CALL read_char      ;Egy karakter beolvasása
        CMP DL, CR          ;ENTER ellenőrzése
        JE read_string_end  ;Vége, ha ENTER volt az utolsó karakter

        MOV [BX], DL        ;Mentés az adatszegmensre
        INC BX              ;Következő adatcím
        JMP read_string_new ;Következő karakter beolvasása

read_string_end:
        XOR DL, DL
        MOV [BX], DL        ;Sztring lezárása 0-val

        POP BX              ;BX visszaállítása
        POP DX              ;DX visszaállítása
        RET                 ;Visszatérés

read_string endp
```

TYPEDEF direktíva

E direktíva segítségével definiálhatunk pointerváltozót.

*typename **TYPEDEF** distance **PTR** qualifiedtype*

typename: típusnév,
distance: FAR vagy NEAR, de el is maradhat.
qualifiedtype: adattípus (BYTE, WORD, stb.)

Ezután lehet a .DATA részben a pointer deklarálását elvégezni.

buf DB 100 DUP (?)
adat typename buf

PTR direktíva

Eddigi feladatainkban csak egynemű (bájtos) adatokkal foglalkoztunk. Amikor két adattal végeztünk valamilyen műveletet, akkor a két adat típusának (hosszának) meg kellett egyeznie. Például:

MOV [BX], DL vagy CMP [BX], 0

A *MOV* esetében a DL regiszter bájtos, így a fordító program tudja, hogy ez bájt hosszúságú adat mozgatását jelenti. Amikor regiszter is szerepel az utasításban, az assembler - a regiszter nevéből - tudja, hogy milyen típusú adattal kell a műveletet elvégezni.

A második utasításból viszont nem derül ki, hogy szavakat vagy bájtokat akarunk összehasonlítani. Írjuk le ezt az utasítást másképp:

CMP BYTE PTR [BX], 0

Így megmondjuk az assemblernek, hogy a BX egy bájt hosszúságú adatra mutat, tehát két bájtot akarunk összehasonlítani.

Nézzünk egy másik esetet, amikor adatszegmensből töltünk be karaktereket:

LEA BX, Adat

MOV DL, [BX]

BX-be került az adatok kezdő címe, majd a BX inkrementálásával töltjük be sorban, egymás után a karaktereket. Ezt megtehetjük másképpen is:

XOR BX, BX

MOV DL, adat[BX]

Itt a BX nem az adat címét tartalmazza, hanem a kezdőponthoz (*adat*) képest elfoglalt helyét (ofszetjét). BX inkrementálásával érhetjük el a következő karaktereket ebben az esetben is.

Ha két karaktert szeretnénk egy memóriaszóba írni, akkor a

MOV DX, adat[BX]

nem megengedett utasítás, mivel az *adat* egy bájtos címke, a DX pedig egy szavas regiszter. Ebben az esetben – ha az *adat*-ot valóban szó címként akarjuk használni – a direktívát alkalmazva, ezt megadhatjuk a fordítóprogramnak.

MOV DX, WORD PTR Adat[BX]

A bemutatott direktívák felhasználásával írjuk meg az előző alprogramunkat a processzor sztringkezelő utasításainak segítségével.

```
.MODEL SMALL
FPBYTE TYPEDEF FAR PTR BYTE
.STACK
.DATA?
buf DB 100 DUP (?)
adat FPBYTE buf
.CODE
main proc
        MOV AX, DGROUP      ;Adatszegmens beállítása
        MOV DS, AX
        LES DI, adat        ;A sztring-pointer betöltése a ES:DI-be
        CALL read_string    ;Karaktersorozat beolvasása
        CALL cr_lf          ;Soremelés
        LDS SI, adat        ;A sztring-pointer betöltése a DS:SI-be
        CALL write_string   ;Karaktersorozat visszaírása
        MOV AH,4Ch          ;Kilépés
        INT 21h
main endp
write_string proc
        PUSH AX             ;AX mentése
        PUSH DX             ;DX mentése
        CLD                 ;Irányjelző beállítása
write_string_new:
        LODSB               ;DS:SI tartalma AL-be, SI-t eggyel megnöveli
        OR AL, AL           ;Sztring végének ellenőrzése
        JZ write_string_end
        MOV DL, AL          ;DL-be a kiírandó karakter
        CALL write_char     ;Karakter kiírása
        JMP write_string_new ;Új karakter
write_string_end:
        POP DX              ;DX visszaállítása
        POP AX              ;AX visszaállítása
        RET
write_string endp
```

```
read_string proc
    PUSH DX             ;DX mentése a verembe
    PUSH AX             ;BX mentése a verembe
    CLD                 ;Irányjelző beállítása
read_string_new:
    CALL read_char      ;Egy karakter beolvasása
    CMP DL, CR          ;ENTER ellenőrzése
    JE read_string_end  ;Vége, ha ENTER volt az utolsó karakter
    MOV AL, DL          ;AL-be a karakter
    STOSB               ;AL tartalma ES:DI címre, DI-t eggyel megnöveli
    JMP read_string_new ;Következő karakter beolvasása
read_string_end:
    XOR AL,AL
    STOSB               ; Sztring lezárása 0-val
    POP AX              ;BX visszaállítása
    POP DX              ;DX visszaállítása
    RET                 ;Visszatérés
read_string endp
```

A **LES** és **LDS** utasítások mutatót töltenek be a regiszterbe, ezért használatuk előtt definiálni kellett azt a mutatót, amely az adatterületre mutat. A mutató megadásánál újabb direktívákkal találkozunk.

A sztringek beolvasását és kiírását elvégezhetjük az INT 21h megszakítás segítségével is.

Az interrupt-ok használatakor a következőkre kell figyelni:

- Az írás és olvasás adatterületének címét a DS:DX regiszter-párba kell megadni.
- Kiíráskor a zárókarakter a „$”.
- Beolvasás esetén az adatterület első bájtja a puffer hosszát kell, hogy tartalmazza (az ENTER-rel együtt). A másodikba pedig a ténylegesen beolvasott karakterek száma kerül.

Beolvasáskor, ha a megadott puffer méreténél több karaktert akarunk megadni, azt nem engedi és sípolással figyelmeztet. Előnye ennek az eljárásnak, hogy használható a karaktertörlés (Backspace). A pufferbe csak az ENTER leütése után kerül a karaktersor. Tehát a beolvasáshoz szükséges adatterület mérete a két első (hossz) bájtból, a beolvasandó karakterek számából és az ENTER-ből tevődik össze.

Az egyszerűség kedvéért most nem külön szubrutinokba tesszük az eljárásokat.

Mivel a kiíráshoz „$”-al kell lezárni a sztringet – hogy ezzel ne kelljen foglalkozni – a puffert előre feltöltjük ezzel a karakterrel.

```
.MODEL SMALL
.STACK
.DATA
Attr DB 10,0              ;Hossz adatok: puffer méret, karakterszám
Adat DB 11 DUP ('$')      ;Adatterület, feltöltve ”$” karakterrel
.CODE
main proc
    MOV AX, DGROUP        ;Adatszegmens beállítása
    MOV DS, AX
    LEA DX, Attr          ;Input puffer, az első két bájt a hosszakhoz
    MOV AH, 0Ah           ;Funkciókód
    INT 21h               ;Sztring beolvasása
    CALL CR_LF            ;Soremelés
    LEA DX, Adat          ;Output puffer
    MOV AH, 9h            ;Funkciókód
    INT 21h               ;Sztring kiírás
    MOV AH,4Ch            ;Visszatérés az operációs rendszerbe
    INT 21h
main endp
```

A programunkban 10 karakter méretű puffert foglaltunk le, de ebbe csak 9 karaktert tudunk beírni, mivel az ENTER (Dh) karakternek is kell egy hely.

Vegyük észre, hogy az assemblyben megadott adatnevek csak az adatterület kezdetét jelentik. Nyugodtan hivatkozhatunk egyik címről egy másik alatt definiált értékre. Ez abból adódik, hogy a memóriában sorban, egymás után hézagmentesen tárolódnak az adatokat, és azokra bármelyik báziscímhez képest hivatkozhatunk.

OFFSET operátor

Egy adat címét megadhatjuk az OFFSET operátor segítségével is. Írjuk át az előző programot úgy, hogy a *LEA DX, Adat* helyett a *MOV DX, OFFSET Adat* utasítást használjuk. Mindkét esetben a DX regiszterbe az *Adat* címe töltődik be.

6.6. Lemezmeghajtó kezelése

Egyszerű program segítségével mutatjuk be egy lemezmeghajtó olvasását. A programban nem végzünk hibafigyelést, például nem vizsgáljuk, hogy van-e lemez a meghajtóban.

A programban az INT 25h megszakítást használjuk, amely segítségével egy blokkot (szektort) olvasunk be a lemezről. A beolvasás nem karakterenként történik, mint például a billentyűzetről, hanem blokkosan. Ilyenkor csak a forrás és cél címet, valamint a másolandó bájtok számát kell megadnunk.

A floppy lemezen egy blokk mérete 512 bájt, így az adatok tárolására 512 bájtnyi helyet foglalunk le az adatszegmensben. (A korszerű háttértárak már nagyobb szektorérettel rendelkeznek. Ha nem floppyt akarunk olvasni, ügyeljünk arra, hogy a tárterület megfelelően nagy legyen.

Programunk segítségével a beolvasott adatokat kétféleképpen írjuk ki: hexadecimális és karakteres formában is. Egy 80 karakter/soros képernyőn egy sorban 16 bájtnyi adat fér el. Így egy blokk kiírásához 32 sorra lesz szükség. (Ha 25 soros monitorunk van, akkor célszerű a megjelenítést két fél blokkra osztani.)

A képernyőelrendezés az alábbi ábrán látható:

```
EB 34 90 49 42 4D 20 20 33 2E 33 00 02 02 01 00  Ű4ÉIBM..3.3.....
02 70 00 A0 05 F9 03 00 09 00 02 00 00 00 00 00  .p.á. ..........
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 12  ................
00 00 00 00 01 00 FA 33 C0 8E D0 BC 00 7C 16 07  ......ú3└Ä╨╝.|..
BB 78 00 36 C5 37 1E 56 16 53 BF 2B 7C B9 0B 00  ╗x.6┼7.V.S┐+|╣..
FC AC 26 80 3D 00 74 03 26 8A 05 AA 8A C4 E2 F1  ü¼&Ç=.t.&Ő.¬Ő─Ô˜
06 1F 89 47 02 C7 07 2B 7C FB CD 13 72 67 A0 10  ..ëG.ă.+|ű=.rgá.
7C 98 F7 26 16 7C 03 06 1C 7C 03 06 0E 7C A3 3F  |ś¸&.|...|...|ú?
7C A3 37 7C B8 20 00 F7 26 11 7C 8B 1E 0B 7C 03  |ú7|Ş.. ¸&.|ő..|.
C3 48 F7 F3 01 06 37 7C BB 00 05 A1 3F 7C E8 9F  ├H¸ ..7|╗..í?|Ŕč
00 B8 01 02 E8 B3 00 72 19 8B FB B9 0B 00 BE D6  .Ş..Ŕ|.r.őű╣..ŻÍ
7D F3 A6 75 0D 8D 7F 20 BE E1 7D B9 0B 00 F3 A6  }¸Žu.Ża.żß}╣..¸Ž
74 18 BE 77 7D E8 6A 00 32 E4 CD 16 5E 1F 8F 04  t.żw}Ŕj.2ń=.Ä.Ć.
8F 44 02 CD 19 BE C0 7D EB EB A1 1C 05 33 D2 F7  ĆD.=.ż└}ŰŰí..3Ď¸
36 0B 7C FE C0 A2 3C 7C A1 37 7C A3 3D 7C BB 00  6.|■└ó<|í7|ú=|╗.
07 A1 37 7C E8 49 00 A1 18 7C 2A 06 3B 7C 40 38  .í7|ŔI.í.|*.;|@8
06 3C 7C 73 03 A0 3C 7C 50 E8 4E 00 58 72 C6 28  .<|s.á<|PŘN.XrĂ(
06 3C 7C 74 0C 01 06 37 7C F7 26 0B 7C 03 D8 EB  .<|t...7|¸&.|.ěŮ
D0 8A 2E 15 7C 8A 16 FD 7D 8B 1E 3D 7C EA 00 00  đŐ.¸|Ő.ř}ő.=|ŕ..
70 00 AC 0A C0 74 22 B4 0E BB 07 00 CD 10 EB F2  p.Č.└t"┤.╗..=.Ű¸
33 D2 F7 36 18 7C FE C2 88 16 3B 7C 33 D2 F7 36  3Ď¸6.|■┬ł.;|3Ď¸6
1A 7C 88 16 2A 7C A3 39 7C C3 B4 02 8B 16 39 7C  .|ł.*|ú9|├┤.ő.9|
B1 06 D2 E6 0A 36 3B 7C 8B CA 86 E9 8A 16 FD 7D  ▒.ĎŠ.6;|ő╧ĆÚŐ.ř}
8A 36 2A 7C CD 13 C3 0D 0A 4E 6F 6E 2D 53 79 73  Ö6*|=.├..Non-Sys
74 65 6D 20 64 69 73 6B 20 6F 72 20 64 69 73 6B  tem.disk.or.disk
20 65 72 72 6F 72 0D 0A 52 65 70 6C 61 63 65 20  .error..Replace.
61 6E 64 20 73 74 72 69 6B 65 20 61 6E 79 20 6B  and.strike.any.k
65 79 20 77 68 65 6E 20 72 65 61 64 79 0D 0A 00  ey.when.ready...
0D 0A 44 69 73 6B 20 42 6F 6F 74 20 66 61 69 6C  ..Disk.Boot.fail
75 72 65 0D 0A 00 49 42 4D 42 49 4F 20 20 43 4F  ure...IBMBIO..CO
4D 49 42 4D 44 4F 53 20 20 43 4F 4D 00 00 00 00  MIBMDOS..COM....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA  ..............U¬
```

A programban felhasználjuk a már megírt szubrutinokat.

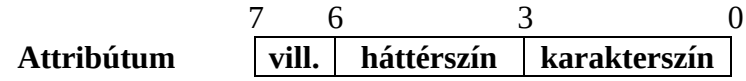
```
.MODEL SMALL
        Space EQU " "           ;Szóköz karakter
.STACK
.DATA?
        block DB 512 DUP (?)    ;1 blokknyi terület kijelölése
.CODE
main proc
        MOV AX, Dgroup          ;DS beállítása
        MOV DS, AX
        LEA BX, block           ;DS:BX memóriacímre tölti a blokkot
        MOV AL, 0               ;Lemezmeghajtó száma (A:0, B:1, C:2, stb.)
        MOV CX, 1               ;Egyszerre beolvasott blokkok száma
        MOV DX, 0               ;Lemezolvasás kezdőblokkja
        INT 25h                 ;Olvasás
        POPF                    ;A veremben tárolt jelzőbitek törlése
        XOR DX, DX              ;Kiírandó adatok kezdőcíme DS:DX
        CALL write_block        ;Egy blokk kiírása
        MOV AH,4Ch              ;Kilépés a programból
        INT 21h
main endp
write_block proc                ;Egy blokk kiírása a képernyőre
        PUSH CX                 ;CX mentése
        PUSH DX                 ;DX mentése
        MOV CX, 32              ;Kiírandó sorok száma CX-be
write_block_new:
        CALL out_line           ;Egy sor kiírása
        CALL cr_lf              ;Soremelés
        ADD DX, 16              ;Következő sor adatainak kezdőcíme;
        LOOP write_block_new    ;Új sor
        POP DX                  ;DX visszaállítása
        POP CX                  ;CX visszaállítása
        RET
write_block endp
```

```
out_line proc
        PUSH BX          ;BX mentése
        PUSH CX          ;CX mentése
        PUSH DX          ;DX mentése
        MOV BX,DX        ;Sor adatainak kezdőcíme BX-be
        PUSH BX          ;Mentés a karakteres kiíráshoz
        MOV CX, 16       ;Egy sorban 16 hexadecimális karakter
hexa_out:
        MOV DL, Block[BX] ;Egy bájt betöltése
        CALL write_hexa  ;Kiírás hexadecimális formában
        MOV DL,Space     ;Szóköz kiírása a hexa kódok között
        CALL write_char
        INC BX           ;Következő adatbájt címe
        LOOP hexa_out    ;Következő bájt
        MOV DL, Space    ;Szóköz kiírása a kétféle mód között
        CALL write_char
        MOV CX, 16       ;Egy sorban 16 karakter
        POP BX           ;Adatok kezdőcímének beállítása
ascii_out:
        MOV DL, Block[BX] ;Egy bájt betöltése
        CMP DL, Space    ;Vezérlőkarakterek kiszűrése
        JA visible       ;Ugrás, ha látható karakter
        MOV DL, Space    ;Nem látható karakterek cseréje szóközre
visible:
        CALL write_char  ;Karakter kiírása
        INC BX           ;Következő adatbájt címe
        LOOP ascii_out   ;Következő bájt
        POP DX           ;DX visszaállítása
        POP CX           ;CX visszaállítása
        POP BX           ;BX visszaállítása
        RET              ;Vissza a hívó programba
out_line endp
```

6.7. A karakteres videó-memória kezelése

A képernyőn megjelenő kép nem más, mint a képernyő-memória leképezése. Ha a képernyő memóriába beírunk egy karaktert, akkor az azonnal (a frissítési frekvenciának megfelelő sebességgel) megjelenik a monitoron is.

A videó-memória szegmenscíme 0B800h, ami megfelel a képernyő első karakterpozíciójának (bal felső sarok) memóriacímének. Minden karakterpozíciót egy 16 bites szóval jellemezhetünk. Az alsó bájt tartalmazza a karakterkódot, a felső a megjelenítő attribútumot.



Az attribútum bitjei:

0. – a karakter kék színösszetevője,
1. – a karakter zöld színösszetevője,
2. – a karakter piros színösszetevője,
3. – a karakterszín intenzitása (világosságbit),
4. – a háttér kék színösszetevője,
5. – a háttér zöld színösszetevője,
6. – a háttér piros színösszetevője,
7. – a karakter villogtatása.

Az attribútum értékének kiszámítása:

*attribútum = 128 * villogás + 16 * háttérszín + karakterszín*

(Ha a memória írására az AX regisztert használjuk, akkor AL-be írjuk a karakterkódot és AH-ba az attribútumot.)

Színkódok:

0	fekete	8	sötétszürke
1	kék	9	világoskék
2	zöld	10	világoszöld
3	cián	11	világoscián
4	piros	12	világospiros
5	ibolya (lila)	13	világoslila
6	barna	14	sárga
7	szürke	15	fehér

Írjunk programot, amely a képernyő közepére kiír egy piros „*” karaktert, szürke háttérrel.

A képernyő-memóriában a képernyő egy karakter pozíciójának ofszetcímét a következőképpen számítjuk ki:

2((sorszám-1)*sorhossz + karakterpozíció-1)*

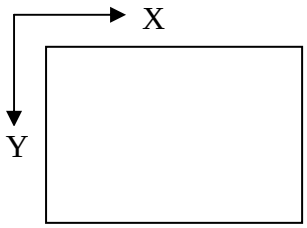
Kettővel azért kell megszorozni a képletet, mert egy karakter 2 bájtot foglal el.

Így egy 80 karakter/sor és 50 soros képernyő közepének ofszetcíme: 3918, a 25 sorosé pedig 1838.

```
.MODEL SMALL
.STACK
.CODE
main proc
        MOV AX, 0B800h          ;Képernyő-memória szegmenscíme ES-be
        MOV ES, AX
;Állítsuk be a képernyőt 80x25-ös felbontásra.
        MOV AH, 0h              ;Képernyőüzemmód
        MOV AL, 3h              ;80x25-ös felbontás, színes üzemmód
        INT 10H

        MOV DI, 1838            ;Képernyő közepének ofszetcíme
        MOV AL, "*"             ;Kiírandó karakter
        MOV AH, 128+16*7+4      ;Színkód: szürke háttér, piros karakter, villog
        MOV ES:[DI], AX         ;Karakter beírása a képernyő-memóriába
        MOV AH, 4Ch             ;Kilépés
        INT 21h
main endp
        END main
```

Ne feledkezzünk meg arról, hogy a képernyő kezdő karaktere a bal felső sarokban van. Ha ezt koordináta-rendszerben képzeljük el, akkor X jobbra, Y pedig lefelé növekszik.



Az első karakter pozíciója (1,1), ofszetcíme a memóriában viszont 0, így ezt is figyelembe kell vennünk. Tehát a karakter címzése (a képernyőn) 1-el, míg a memória címzése 0-val kezdődik.

Bővítsük a programunkat úgy, hogy tetszőleges képernyőcímen jeleníthessük meg a kívánt karaktert. A koordinátákat, a karakter kódját és az attribútumot adatszegmensben adjuk meg.

```
.MODEL SMALL
.STACK
.DATA
    x DB 10             ;X koordináta, oszlopszám
    y DB 10             ;Y koordináta, sorszám
    kar DB "A"          ;Kiírandó karakter
    att DB 4            ;Kiírás attribútuma
.CODE
main proc
    MOV AX, dgroup      ;Adatszegmens beállítása
    MOV DS, AX
    MOV AX, 0B800h      ;Képernyő-memória szegmenscíme ES-be
    MOV ES, AX
    XOR AX, AX          ;AX törlése
    MOV BL, 160         ;Szorzó betöltése BL-be
    MOV AL, y           ;Y koordináta betöltése AL-be
    DEC AL              ;AL-1, az 1. karakter a memória 0. címén van
    MUL BL              ;AL szorzása 160-nal
    MOV DI, AX          ;DI-be a sorszámból számított memóriahely
    XOR AX, AX          ;AX törlése
    MOV AL, x           ;X koordináta betöltése AL-be
    DEC AL              ;AL-1, az 1. karakter a memória 0. címén van
    SHL AL, 1           ;AL szorzása 2-vel (1-el balra shift)
    ADD DI, AX          ;DI-hez hozzáadjuk az oszlopszámból
                        ;számított memóriahelyet
    MOV AL, kar         ;AL-be a karakterkód
    MOV AH, att         ;AH-ba a karakter attribútuma
    MOV ES:[DI], AX     ;Betöltés a képernyő-memória kiszámított címére
    MOV AH, 4Ch         ;Program befejezése
    INT 21h
main endp
END main
```

Az eddigi mintafeladatok alapján írjunk programot, amely billentyűzetről kéri be a koordinátákat, a karaktert és a színkódokat. Használjuk a decimális beolvasó eljárásunkat is.

6.8. Mintapéldák az INT 21h interrupthoz

Az alábbiakban néhány mintaprogramot mutatunk be (a teljesség igénye nélkül) az INT 21h interrupt használatára.

A szöveges üzenetek kiírására – az egyszerűség kedvéért – az INT 21h megszakítást használjuk (ld. 6.5. fejezet).

A rendszerdátum lekérdezése:

```
.MODEL SMALL
.STACK
.CODE
main proc
    MOV AH, 2Ah
    INT 21h
    CALL cr_lf
    CALL write_decimal      ;A nap kiírása
    CALL cr_lf
    MOV DL,DH
    CALL write_decimal      ;A hónap kiírása
    CALL cr_lf
    MOV DL, CH
    CALL write_binary       ;Az év felső bájtjának kiírása
    CALL cr_lf
    MOV DL, CL
    CALL write_binary       ;Az év alsó bájtjának kiírása
    CALL cr_lf
    MOV DL, AL
    CALL write_decimal      ;A hét napja sorszámának kiírása
    CALL cr_lf
    MOV AH,4Ch              ;Kilépés
    INT 21h
main endp
end main
```

Mivel az évszám a DX regiszterbe (2 bájt) kerül, a write_decimal szubrutinnal csak egybájtos értéket tudunk kiíratni, ezért az évszám bájtjait külön-külön binárisan íratjuk ki.

A rendszeridő lekérdezése:

```
.MODEL SMALL
.STACK
.CODE
main proc
        MOV AH, 2Ch
        INT 21h
        CALL cr_lf
        CALL write_decimal      ;A századmásodperc kiírása
        CALL cr_lf
        MOV DL,DH
        CALL write_decimal      ;A másodperc kiírása
        CALL cr_lf
        MOV DL, CL
        CALL write_decimal      ;A perc kiírása
        CALL cr_lf
        MOV DL, CH
        CALL write_decimal      ;Az óra kiírása
        CALL cr_lf
        MOV AH,4Ch              ;Kilépés
        INT 21h
main endp
end main
```

Könyvtár (mappa) létrehozása:

```
.MODEL SMALL
.STACK
.CODE
.DATA
        Library         DB  'teszt',0
        Success         DB  'Sikeres megnyitás$'
        Error           DB  'Sikertelen megnyitás$'
.CODE
main proc
        MOV AX,DGROUP   ;Adatszegmens beállítása
        MOV DS,AX
        LEA DX, Library ;Könyvtár nevének kezdőcíme
        MOV AH,39h      ;A könyvtár létrehozása
        INT 21h
```

```
        JNC exist           ;Sikeres megnyitás
        LEA DX, Error       ;"Sikertelen megnyitás" szöveg kezdőcíme
        JMP stop
exist:
        LEA DX, Success     ;"Sikeres megnyitás" szöveg kezdőcíme
stop:
        MOV AH,9            ;Szövegkiírás
        INT 21h
        MOV AH,4Ch          ;Kilépés
        INT 21h
main endp
end main
```

Könyvtár (mappa) törlése:

```
.MODEL SMALL
.STACK
.CODE
.DATA
        Library             DB  'teszt',0
        Success             DB  'Sikeres törlés$'
        Error               DB  'Sikertelen törlés$'
.CODE
main proc
        MOV AX,DGROUP       ;Adatszegmens beállítása
        MOV DS,AX
        LEA DX, Library     ;Könyvtár nevének kezdőcíme
        MOV AH,3Ah          ;A könyvtár törlése
        INT 21h
        JNC exist           ;Sikeres törlés
        LEA DX, Error       ;"Sikertelen törlés" szöveg kezdőcíme
        JMP stop
exist:
        LEA DX, Success     ;"Sikeres törlés" szöveg kezdőcíme
stop:
        MOV AH,9            ;Szövegkiírás
        INT 21h
        MOV AH,4Ch          ;Kilépés
        INT 21h
main endp
end main
```

Fájl törlése:

```
.MODEL SMALL
.STACK
.DATA
        File_name       DB  'teszt.txt',0
        Success         DB  'Sikeres törlés$'
        Nofile          DB  'A fájl nem létezik$'
        Error           DB  'Törlési hiba$'
.CODE
main proc
        MOV AX,DGROUP       ;Adatszegmens beállítása
        MOV DS,AX
        LEA DX, File_name   ;Fájl nevének kezdőcíme
        MOV CX,3Fh          ;Keresés minden attribútumra
        MOV AH,4Eh          ;A fájl első előfordulásának keresése
        INT 21h
        JNC file_exist      ;A fájl létezik
        LEA DX, Nofile      ;"A fájl nem létezik" szöveg kezdőcíme
        JMP stop
file_exist:
        LEA DX, File_name   ;Fájl nevének kezdőcíme
        MOV AH,41h          ;Fájl törlés
        INT 21h
        JNC deleting        ;Sikeres törlés
        LEA DX, Error       ;"Törlési hiba" szöveg kezdőcíme
        JMP stop
deleting:
        LEA DX, Success     ;"Sikeres törlés" szöveg kezdőcíme
stop:
        MOV AH,9            ;Szövegkiírás
        INT 21h
        MOV AH,4Ch          ;Kilépés
        INT 21h
main endp
end main
```

Olvasás fájlból:

```
.MODEL SMALL
.STACK
.DATA
    File        DB   'Text.txt',0
    Error       DB   'Fájl nyitási hiba$'
    Read_error  DB   'Olvasási hiba$'
    Data        DB 1024 DUP ('$')       ;Adatterület
.CODE
main proc
    MOV AX, Dgroup      ;DS beállítása
    MOV DS, AX
    LEA DX, File        ;Féjl nevének kezdőcíme
    XOR AL, AL          ;Fájl megnyitása olvasásra
    MOV AH, 3Dh         ;Fájl megnyitása
    INT 21h
    JNC Opened          ;Sikeres fájlnyitás, AX=Handle
    LEA DX, Error       ;"Fájlnyitási hiba" szöveg kezdőcíme
    JMP Stop
Opened:
    LEA DX, Data        ;Adatterület kezdőcíme
    MOV CX, 1024        ;Beolvasott bájtok száma (puffer mérete)
    MOV BX, AX          ;Handle BX-be
    MOV AH, 3Fh         ;Olvasás a fájlból
    INT 21h
    JNC Success         ;Sikeres olvasás
    LEA DX, Read_error  ;"Olvasási hiba" szöveg kezdőcíme
    JMP Stop
Success:
    MOV AH, 3Eh         ;Fájl lezárás
    INT 21h
    LEA DX, Data        ;Adatterület kezdőcíme a kiíráshoz
Stop:
    MOV AH,9            ;Szövegkiírás
    INT 21h
    MOV AH,4Ch          ;Kilépés
    INT 21h
main endp
end main
```

7. Az assembly kapcsolata más nyelvekkel

Ebben a fejezetben írt mintapéldákat akkor is érdemes áttanulmányozni, ha az adott programozási nyelvet nem használjuk, ugyanis új eljárásokkal fogunk megismerkedni.

7.1. Assembly rutinok Turbo Pascal programokban

Az assembly utasítások használatát - Pascal nyelven - néhány mintaprogram segítségével mutatjuk be.

Az assembly programmodulok futtatásakor nem szükséges tárolnunk (verembe menteni) az AX, BX, CX, és DX regisztereket. Azonban az SI, DI, BP, SP, CS, DS, és SS regisztereket el kell menteni, ha használjuk azokat.

7.1.1. Alapműveletek

Írjunk rutinokat a négy alapművelet elvégzésére.

```
Program Feladat1
    {Bájt típusú adatok összeadása, eredmény bájtos}
Function Osszeg (A, B: Byte): Byte ; Assembler;
    ASM
        MOV AL, A       {Az első paraméter AL-be}
        ADD AL, B       {A második paraméter  összeadása AL-lel}
    END;                {A függvény értéke AL-ben van}
    {Bájt típusú adatok kivonása, eredmény is bájtos}
Function Kulonbseg (A, B: Byte): Byte ; Assembler;
    ASM
        MOV AL, A       {Az első paraméter AL-be}
        SUB AL, B       {A második paraméter kivonása AL-ből}
    END;                {A függvény értéke AL-ben van}
    {Word típusú adatok szorzása, eredmény LongInt}
Function WordSzorzas (A, B: Word): LongInt ; Assembler;
    ASM
        MOV AX, A       {Az első paraméter AX-be}
        MUL B           {AX szorzása B-vel, eredmény DX:AX-ben}
    END;                {A függvény értéke DX:AX-ben van}
```

```
    {Előjeles szorzás}
Function IntSzorzas (A, B: ShortInt): Integer ; Assembler;
    ASM
        MOV AL, A       {Az első paraméter AL-be}
        IMUL B          {Szorzása B-vel}
    END;                {A függvény értéke AX-ben van}
    {Főprogram}
Begin
    Writeln('Összeg: 25 + 40 = ', Osszeg(25, 40));
    Writeln('Különbség: 123 – 34 = ', Kulonbseg(123, 34));
    Writeln('Szorzás: 2345 * 1234 = ', WordSzorzas(2345, 1234));
    Writeln('Előjeles szorzás: -12 * 25 = ', IntSzorzas(-12, 25));
End.
```

A függvények visszatérési értéke:

- Byte, Shortint (1 bájtos) a függvény típusa, akkor az AL regiszterbe kell tölteni a visszatérési értéket.
- Word, Integer (2 bájtos) a függvény típusa, akkor az AX regiszterbe kell tölteni a visszatérési értéket.
- Longint, Pointer (4 bájtos) a függvény típusa, akkor a DX:AX regiszterpárba kell tölteni a visszatérési értéket.
- Real (6 bájtos) a függvény típusa, akkor az DX:BX:AX regiszterekbe kell tölteni a visszatérési értéket. (Figyelembe kell venni a valós számok ábrázolásának szabályát.)
- String típusú függvény esetén a visszatérési értéket a @Result változó által meghatározott memóriaterületre kell pakolni. Pascal-ban a sztring kezelése eltér az Assemblyétől. Ezért a sztring összeállításánál figyelembe kell venni azt, hogy az első byte tartalmazza a sztring hosszát.
- Azoknál a típusoknál (Single, Double, Extended) ahol a koprocesszort is használjuk, a visszatérési értéket a matematikai processzor ST(0) regiszterén keresztül adjuk vissza.

7.1.2. Input – Output műveletek

Írjunk programokat billentyűzetről történő olvasás és a képernyőre írás szemléltetésére.

```
Program Feladat2   {Standard input – output kezelése}
Var
    Kar: Char;
    {Egy karakter kiírása monitorra}
Procedure Kiir(C: Char); Assembler;
    ASM
        MOV DL, C       {DL-be a karakter kódja}
        MOV AH, 2       {AH-ba a képernyőre írás funkciókódja}
        INT 21h         {Karakter kiírása}
    END;
    {Egy karakter beolvasása a billentyűről}
Function Beolvas: Char; Assembler;
    ASM
        MOV AH,1        {AH-ba a beolvasás kódja}
        INT 21h         {Karakter beolvasása, a karakter kód AL-ben}
    END;
Begin
    Kar:=Beolvas;
    Writeln;
    Kiir(Kar);
End.
```

Töltsünk fel egy tömböt karakterekkel: a-tól z-ig, majd írassuk ki. A kiírás végére tegyünk egy soremelést.

```
Program Feladat3   {Tömb feltöltése és kiírása}
Const
    MAX = 26;
    CR_LF: Array[1..3] of Char = #$0D#$0A'$'; {0Dh, 0Ah, és a '$' karakterek}
Type
    TombType = Array[1.. MAX]of Char;
Var
    Tomb: TombType;
    {Tömb feltöltése kisbetűkkel}
```

```
Procedure Feltolt(Var T: Tombtype); Assembler;
    ASM
        PUSH DI          {DI mentése}
        LES DI, T        {A tömb ofszet címének beállítása}
        MOV AL, 'a'      {Első karakter}
        MOV CX, MAX      {Ciklusváltozó beállítása}
        CLD              {Irányjelző bit törlése, ES:DI automatikusan nő a
                          sztringművelet után}
@Uj_karakter:            {Ha nincs előre deklarálva a cím '@'-al kell kezdeni}
        STOSB            {AL tartalmát az ES:DI címre teszi, majd DI-t eggyel
                          megnöveli.}
        INC AL           {AL-be a következő karakter.}
        LOOP @Uj_karakter  {Következő karakter}
        POP DI           {DI visszaállítása}
    END;
    {Soremelés kiírása}
Procedure Soremeles; Assembler;
    ASM
        LEA DX, CR_LF    {CR_LF ofszet címének betöltése DX-ben
        MOV AH, 9        {Sztring kiírása a '$' karakterig}
        INT 21h          {DS:DX kezdőcímű sztring kiírása}
    END;
    {Várakozás egy karakter leütésére}
Procedure Wait; Asembler;
    ASM
        MOV AH, 1        (Várakozás egy karakter leütésére}
        INT 21h          {DS:DX kezdőcímű sztring kiírása}
    END;
Begin
    Feltolt(Tomb);
    For I:=1 to MAX do
        Kiir(Tomb[I]);   {A Kiir eljárás az előző példában szerepel!}
    Soremeles;
    Wait;
End.
```

Írjunk programot, amellyel egy sztringet feltöltünk a billentyűzetről, majd kiírjuk a képernyőre.

```
Program Feladat4   {Sztring kiírása}
Var
    ST: string;
    {Sztring kiírás a képernyűre}
Procedure StringWrite(Var S: String); Assembler;
    ASM
        PUSH SI          {SI mentése}
        LDS SI, S        {A sztring-pointer betöltése a DS:SI-be}
        CLD              {Irányjelző beállítása}
        LODSB            {DS:SI által megcímzett bájt betöltése AL-be. Ez a
                          sztring hossza}
        XOR CX, CX       {CX törlése}
        MOV CL, AL       {Szting hossza CL-be (számláló beállítása)}
@Uj_karakter:
        LODSB            {AL-be DS:SI tartalma, SI-t eggyel megnöveli.}
        MOV DL, AL       {DL-be a kiírandó karakter}
        MOV AH, 2        {Kiírás funkció}
        INT 21h          {Karakter kiírása}
        LOOP @Uj_karakter   {Következő karakter}
        POP SI           {SI visszaállítása}
    END;
Procedure StringRead(Var S: String); Assembler;{Sztring beolvasás}
Const
    CR=#13;
    ASM
        PUSH DI          {DI mentése
        LES DI, S        {A sztring-pointer betöltése a ES:DI-be}
        CLD              {Irányjelző beállítása}
        XOR CX, CX       {CX törlése}
        PUSH DI          {Sztring hosszának címét elmentjük}
        INC DI           {Az első hely kihagyása, ide a hossz kerül}
```

```
@Uj_karakter:
        MOV AH, 1       {Karakter beolvasás funkció}
        INT 21h         {Karakter beolvasása AL-be}
        CMP AL, CR      {ENTER ellenőrzése}
        JE @Vege        {Vége, ha ENTER volt az utolsó karakter}
        STOSB           {AL tartalmát ES:DI címre}
        INC CL          {Karakterszámlálás}
        JMP @Uj_karakter    {Következő karakter}
@Vege:
        MOV AL, CL      {AL-be a sztring hossza}
        POP DI          {Sztring kezdetének visszaállítása}
        STOSB           {A hossz kiírása ES:DI címre}
        POP DI          {DI visszaállítása}
    END;
Begin
    Readln(ST);         {Sztring feltöltése Readln eljárással}
    Writeln;            {Soremelés}
    StringWrite(ST);    {Sztring kiírása saját eljárással}
    ST:='';             {Sztring törlése}
    StringRead(ST);     {Sztring feltöltése saját eljárással}
    Writeln;            {Soremelés}
    Writeln(ST);        {Sztring kiírása Writeln eljárással}
    Readln;             {Vár egy ENTER-re}
End.
```

A főprogramban elkülönítettük a beolvasó és kiíró rutinok hívását, és külön-külön leteszteltük azokat a Pascal saját eljárásaival.

A beolvasó rutinnál ügyeljünk a sztring hosszának helyes beállítására, ugyanis a Pascal a hosszból dönti el, hogy hány karakter kell kiírnia. Így ha például nulla kerül a hossz-bájtba, akkor a *Writeln* nem ír ki semmit, annak ellenére, hogy a sztring betöltését helyesen végeztük el.

7.2. Assembly rutinok C programokban

C nyelvből hívható assembly eljárások hívásának kétféle módját mutatjuk be. Az első esetben – hasonlóan a Pascalhoz – a programba beépített, a C fordító (compiler) által lefordított modulokat vizsgáljuk meg, majd a 7.2.2 fejezetben, assemblyben írt, assemblerrel fordított programok alkalmazásához készítünk mintafeladatokat.

A mintaprogramban egérkezelő rutinok működését mutatjuk be.

Az egér vezérlését a 33h szoftveres megszakításon keresztül végezhetjük el. Itt nincs lehetőségünk az összes egérfunkció bemutatására, erről a melléklet Megszakítások fejezetében bővebb információt szerezhetünk.

7.2.1. C programba írt assembly utasítások

A programunkban bemutatunk néhány grafikus egérkurzor definiálást is.

```
/* Egérkezelő alaprutinok */

typedef struct _PTRSHAPE            /* Egérkurzor struktúra */
{
        unsigned xHot, yHot;        /* Egérkurzor bázispontja */
        unsigned afsPtr[32];        /* Egérkurzor képe */
} PTRSHAPE;

/* Egérkurzor láthatósága 1: látható, 2: nem látható*/
typedef enum _PTRVIS { SHOW = 1, HIDE = 2 } PTRVIS;

/* Public egérfunkciók */
int MouseInit( void );              /* Egérkurzor struktúra */
int GetMouseEvent( void );          /* Egérkurzor struktúra */

/* Ablak definiálása. Ezen belül mozoghat a kurzor.*/
void SetMouseWin( short xmin, short ymin, short xmax, short ymax void SetMousePos(
short x, short y );                 /* Egérkurzor pozicionálása */
void SetMouseVis( PTRVIS pv );      /* Kurzor ki-be kapcsolása */
void SetMouseShape( PTRSHAPE _far *ps );      /* Kurzor beállítás */

struct MOUINFO                      /* Egér adatstruktúra*/
{
        short Mousex, Mousey;       /* Egér pozició*/
        unsigned MouseBtn;          /* Egér gomb*/
} static mi = { 0, 0, 0 };
```

```
/* Néhány egérkurzor definiálás */

/* Nyíl alakú kurzor */
 PTRSHAPE s_arrow = {0,0,
 0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,
 0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,
 0XC000,0XF000,0X7C00,0X7F00,0X3FC0,0X3FF0,0X1FFC,0X1FFF,
 0X0F00,0X0F00,0X0700,0X0700,0X0300,0X0300,0X0100,0X0100};

 /* Kör alakú kurzor */
PTRSHAPE s_circle = {8,8,
 0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,
 0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,
 0X0000,0X0000,0X0000,0X0000,0X0180,0X07E0,0X07E0,0X0FF0,
 0X0FF0,0X07E0,0X07E0,0X0180,0X0000,0X0000,0X0000,0X0000};

/* X alakú kurzor */
PTRSHAPE s_x = {8,8,
 0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,
 0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,0XFFFF,
 0X0000,0X2004,0X700E,0X381C,0X1C38,0X0E70,0X07E0,0X03C0,
 0X03C0,0X07E0,0X0E70,0X1C38,0X381C,0X700E,0X2004,0X0000};

/* MouseInit - Egér inicializálása.
* Visszatérési érték 0 ha az egeret nem sikerült inicializálni */
      int MouseInit()
      {
          register flag, btn;
          _asm
          {
                xor ax, ax              ;Egérfunkció 0
                int 33h                 ;Reset mouse
                mov flag, ax            ;Funkciókód mentése
                mov btn, bx             ;Egérgomb mentése
          }
          if( !flag ) return 0;
          _asm
          {
                mov ax, 1               ;Egérkurzor látható
                int 33h
                mov ax, 3               ;Egérkurzor-pozíció lekérdezése
                int 33h
                mov mi.Mousex, cx       ;Oszlop koordináta
                mov mi.Mousey, dx       ;Sor koordináta
                mov mi.MouseBtn, bx     ;Egérgomb
          }
          return btn;  /* Visszatérés az egérgomb kódjával */
      }
```

```
/* GetMouseEvent – Egér státusz lekérdezése
 * Visszatérés: 1 ha van változás, és 0 ha nincs */

    int GetMouseEvent( void )
    {
        int rtn;
        _asm
        {
            mov ax, 3           ;Egér koordináta és gomb lekérdezése
            int 33h
            xor ax, ax          ;Ha nincs változás
            cmp bx, mi.MouseBtn     ;Másik gomb, mint a legutolsó?
            je noevent
            cmp cx, mi.Mousex   ;Változott az oszlop koordináta?
            jne event
            cmp dx, mi.Mousey   ;Változott a sor koordináta?
            je noevent

        event:
            mov ax, 1           ;Ha bármi is változott
            mov mi.Mousex, cx   ;Új oszlopkoordináta
            mov mi.Mousey, dx   ;Új sorkoordináta
            mov mi.MouseBtn, bx     ;Új egérgomb

        noevent:
            mov rtn, ax         ;Visszatérési érték
        }
        return rtn;
    }

/* SetMouseWin – Egér mozgási területének beállítása */

    void SetMouseWin(short xmin, short ymin, short xmax, short ymax)
    {
        _asm
        {
            mov cx, xmin        ;Bal alsó sarok X koordinátája
            mov dx, xmax        ;Jobb felső sarok X koordinátája
            mov ax, 7           ;Vízszintes mozgásintervallum beállítása
            int 33h

            mov cx, ymin        ;Bal alsó sarok Y koordinátája
            mov dx, ymax        ;Jobb felső sarok koordinátája
            mov ax, 8           ;Függőleges mozgásintervallum beállítása
            int 33h
        }
}
```

```
/* SetMouseVis – Láthatóság beállítása.
/* Paraméterek: látható ( SHOW) vagy nem látható (HIDE) */
    void SetMouseVis( PTRVIS pv )       /*pv=1: látható, pv=2: nem látható*/
    {
        _asm
        {
            mov ax, pv          ;Láthatóság beállítása
            int 33h
        }
    }

/* SetMousePos – Egérkurzor pozíciójának beállítása */
    void SetMousePos( short x, short y )
    {
        _asm
        {
            mov ax, 4           ;Egérpozíció beállítás funkció
            mov cx, x           ;Oszlop koordináta
            mov dx, y           ;Sor koordináta
            int 33h
        }
    }

/* SetMouseShape – Egérkurzor képének beállítása */
    void SetMouseShape( PTRSHAPE _far *ps )
    {
        _asm
        {
            les di, ps          ;Ábra leírás memóriacíme ES:DI-be
            mov bx, es:[di].xHot ;X bázispont
            mov cx, es:[di].yHot ;Y bázispont
            mov dx, di
            add dx, 4           ;Kezdet + 4-en kezdődik a kép
            mov ax, 9           ;Grafikus kurzor beállítása
            int 33h
        }
}
```

7.2.2. *Assembly rutinok C-hez*

A következő mintaprogramok segítségével megmutatjuk, hogyan lehet olyan assembly eljárásokat írni, amelyeket hívhatunk C nyelvű programokból.

Ahhoz, hogy megadjuk a MASM fordítónak, hogy C-ből hívható eljárást írunk, a *.MODEL* direktívát ki kell egészítenünk egy „C” betűvel:

```
.MODEL SMALL, C
```

Ezen kívül használnunk kell a *PUBLIC* direktívát is, amely megmondja az assemblernek, hogy az utána megnevezett eljárást hozzáférhetővé (nyilvánossá) kell tenni más állományokban (például C-ben) lévő programok számára.

A *PUBLIC* direktíva hatására az assembler olyan egyéb információkat generál a LINK (szerkesztő) számára, amelyek lehetővé teszik más programokból történő hozzáférést.

A bemutatást egyszerű programmal kezdjük: töröljük le a képernyőt.

A képernyő törléséhez használjuk a *INT 10h* megszakítás *sorgörgetés felfelé* funkcióját.

```
.MODEL SMALL, C
.CODE
PUBLIC clear_screen
clear_screen PROC
    XOR AL, AL      ;Ablak törlése
    XOR CX, CX      ;Bal felső sarok (CL = 0, CH = 0)
    MOV DH, 49      ;50 soros képernyő alsó sora (25 sorosnál ide 24 kell)
    MOV DL, 79      ;Jobb oldali oszlop sorszáma
    MOV BH, 7       ;Törölt helyek attribútuma: fekete háttér szürke karakter
    MOV AH, 6       ;Sorgörgetés felfelé (Scroll-up) funkció
    INT 10h         ;Képernyő törlése
    RET
clear_screen ENDP
```

Látható, hogy a C nyelvű programhoz írt assembly programmodulok esetében nem szükséges elmentenünk az AX, BX, CX, és DX regiszterek tartalmát.

Az SI, DI, BP, SP, CS, DS, és SS regiszterek eredeti tartalmát viszont el kell menteni és vissza kell állítani, ha használjuk azokat!

Paraméterátadás

Az eddigi példaprogramjainkban a paraméterek (adatok) átadására a regisztereket használtuk. A C programok azonban a vermet (stack-et) használják a paraméterek átadására.

Azzal, hogy megadjuk az assemblernek, hogy C-hez írunk alprogramot, néhány beépített automatizmus egyszerűsíti a program írását. Erre láthatunk példát a következő mintaprogramban, ahol ismertetjük a forrásnyelvű listát, és az assembler által kiegészített állományt is.

Írjunk ki egy karaktersort a képernyőre. Használjuk a már megírt képernyőtörlőt is.

(Figyeljünk arra, hogy a C nyelvben különbség van a kis- és nagybetűk között.)

A C programunk:

```
Main()
    {
       celar_screen();       /*Képernyő törlése*/
       write_string("Ez egy szöveg");
    }
```

Az általunk megírt forrásnyelvű állomány:

```
PUBLIC write_string
write_string PROC USES SI, string: PTR BYTE
    PUSHF               ;Státuszregiszter tárolása
    CLD                 ;Irányjelző beállítása (nőni fog)
    MOV SI, string      ;Cím betöltése az SI-be
new_char:
    LODSB               ;DS:SI címről egy karakter betöltése AL-be
    OR AL,AL            ;Sztring végének ellenőrzése
    JZ end_string       ;Ugrás, ha AL=0
    MOV AH, 2           ;Képernyőre írás funkció
    INT 21h             ;Egy karakter kiírása a képernyőre
    JMP new_char
end_string:
    POPF                ;Jelzőbitek visszaállítása
    RET
write_string ENDP
```

A programban két új direktívát is használtunk.

Az első a USES SI, megmondja a fordítónak, hogy az SI regisztert fogjuk használni, tehát el kell menteni a verembe, majd vissza kell tölteni azt.

A második direktívával egy string nevű mutatót definiálunk, amely a sztring első karakterére mutat, és bájt típusú:

```
string: PTR BYTE,
```

Írjuk meg – a fenti direktívák nélkül – a programot:

```
PUBLIC write_string

write_string PROC
        PUSH BP         ;BP mentése
        MOV BP, SP      ;BP beállítása a paraméterekre
        PUSH SI         ;SI mentése
        PUSHF           ;Státuszregiszter tárolása
        CLD             ;Irányjelző beállítása (nőni fog)
        MOV SI, [BP+4]  ;Cím betöltése az SI-be
new_char:
        LODSB           ;DS:SI címről egy karakter betöltése AL-be
        OR AL,AL        ;Sztring végének ellenőrzése
        JZ end_string   ;Ugrás, ha AL=0
        MOV AH, 2       ;Képernyőre írás funkció
        INT 21h         ;Egy karakter kiírása a képernyőre
        JMP new_char    ;Új karakter beolvasása
end_string:
        POPF            ;Jelzőbitek visszaállítása
        POP SI          ;SI visszaállítása
        POP BP          ;BP visszaállítása
        RET 2
write_string ENDP
```

A BP regiszter egy különleges célú regiszter, mivel az alapértelmezésben az SS regiszterrel együtt használjuk. Tehát a stack területét címezhetjük vele.

A stack aktuális címét a MOV BP, SP utasítással töltöttük a BP-be. A sztring címének betöltésénél (MOV SI, [BP+4]) 4-et hozzá kell adnunk. Ez azért szükséges, mert a függvény paraméterei 4 bájttal feljebb helyezkednek el, mint az a cím, ami a veremmutató elmentésekor volt.

A paraméterátadás jobb megértéséhez nézzünk egy egyszerű példát:

Legyen a C programunk:

```
Main()
        {
           int param_1, param_2, param_3;
           /* Meghívjuk a subroutine függvényt 3 paraméterrel*/
           subroutine (param_1, param_2, param_3);
        }
```

A *subroutine* függvény hívásakor a következő történik:

- A jobb szélső paramétertől kezdve betölti a paramétereket a verembe. (Legutoljára az első paramétert menti.)
- Menti a visszatérési címet a verembe.
- Átadja a vezérlést a függvénynek.

Vizsgáljuk meg a verem képét:

Cím	Verem tartalma	
[BP+8]	3. paraméter	
[BP+6]	2. paraméter	
[BP+4]	1. paraméter	
	Visszatérési cím	**IP** mentése. Szubrutin hívásakor ez a verem teteje.
BP	**BP** mentése	*PUSH BP*, **BP** mentése, *MOV BP, SP* az aktuális cím **BP**-ben
	SI mentése	*PUSH SI*, **SI** mentése.

Figyeljük meg, hogy a verem az alacsonyabb memóriacím felé (ábrán lefelé) növekszik.

Amennyiben a BP mentése után azonnal betöltjük a veremmutatót (SP) BP-be, utána már szabadon menthetünk bármennyi adatot a stackre, az már nem befolyásolja a BP-ben elmentett cím és a paraméter távolságát. Azaz az első paraméter mindig azonos ofszetre lesz **BP**-ben tárolt címtől, *MODEL SMALL* esetében 4. Távoli (*FAR*) hívás esetén, mivel a **CS** (kódszegmens) is bekerül a stack-be, az ofszet 6 lesz.

A paraméterek által elfoglalt stack nagysága függ a paraméter típusától, amit a szubrutinban is figyelembe kell vennünk. Vagyis a paraméterek címe (az első kivételével) függ attól, hogy milyen típusúak (hány bájtosak).

Mivel a MASM generálja – a paraméterátadáshoz szükséges összes – utasítást, a fent említett direktívák használatakor nem kell törődnünk azzal, hogy az utasításokat megfelelő sorrendben írjuk, és azzal sem, hogy a paraméterek melyik veremcímen találhatók.

Érdekességként megjegyezzük, hogy a paraméterek fordított sorrendbe történő tárolása lehetővé teszi, hogy kevesebb paramétert használjunk a függvényben, mint amennyivel azt meghívtuk. (Az utolsó paramétereket hagyhatjuk el.)

Nézzünk kétparaméteres adatátadásra példaprogramot. Írjunk egy képernyő-kurzort mozgató rutint. Hivatkozzunk a már elkészített függvényekre.

A C főprogram:

```
Main()
    {
      int x, y;                        /*Kurzor pozíció */
      x = 40;                          /*Képernyő közepe 80x50-es felbontásnál*/
      y = 25;
      clear_screen();                  /*Képernyő törlése*/
      goto_xy(x, y);                   /*Kurzor pozicionálás*/
      write_string("Ez egy szöveg");   /*Szöveg kiírása*/
    }
```

Az assembly szubrutin:

```
PUBLIC goto_xy
goto_xy PROC x: WORD, y: WORD
        MOV DH, BYTE PTR (y)     ;Kurzor oszlop koordinátája
        MOV DL, BYTE PTR (x)     ;Kurzor sor koordinátája
        MOV BH, 0                ;Képernyő oldalszámának beállítása
        MOV AH, 2                ;Kurzorpozíció funkció
        INT 10h                  ;Kurzorpozíció beállítása
        RET
goto_xy ENDP
```

Figyeljük meg, hogy a paraméterek sorrendje megegyezik a hívási sorrenddel. Tehát először az **x**-et, majd az **y**-t deklaráltuk.

Programunkban egy újabb különlegességgel találkozunk: *BYTE PTR (y).*

Híváskor az x és y változókat egészként, paraméter átvételkor pedig word-ként deklaráltuk, ami két - két bájtot jelent. A kurzor címzéséhez pedig két egy bájtos adatra van szükség. Így a fordításkor

MOV DH, WORD PTR [PB+4]

utasítást kapnánk. Ez azonban az assemblyben nem megengedett. Nem mozgathatunk közvetlenül két bájtos (szavas) adatot egy bájtos helyre. Használnunk kell a BYTE PTR direktívát. Ebben az esetben az utasítás:

MOV DH, BYTE PTE WORD PTR [PB+4] lenne.

Ezt viszont az assembler nem tudja kezelni. Ezt a problémát úgy tudjuk kiküszöbölni, hogy zárójelezünk. Tehát:

MOV DH, BYTE PTE (WORD PTR [PB+4])

Ez pedig, visszaírva az eredeti programlistánkba, megfelel:

MOV DH, BYTE PTE (y) utasításnak.

Függvényérték visszaadása

C-ben a függvény érték visszaadására a következő regisztereket használjuk:

- AL – bájt hosszúságú adatnál,
- AX – szó esetén,
- DX:AX – duplaszavas adatnál.

Szemléltetésül írjunk egy rutint, amely segítségével bekérünk egy karaktert a billentyűzetről.

A C főprogram:

```
Main()
    {
        clear_screen();                 /*Képernyő törlése*/
        goto_xy(40, 25);                /*Kurzor a képernyő közepére*/
        write_string(”Ez egy szöveg”);  /*Szöveg kiírása*/
        while (read_key() != ’ ’);      /*Karakterek olvasása szóközig*/
    }
```

Az assembly szubrutin:

```
    PUBLIC read_key
    read_key PROC
        XOR AH, AH      ;Beolvasás funkció
        INT 16h         ;Karakter kód AL-be, scan kód AH-ba
        OR AL, AL       ;Kiterjesztett kód, ha AL=0
        JZ ext_kod      ;Ugrás, ha kiterjesztett kód
        XOR AH, AH      ;AH =0, ha  ASCII kódot adunk vissza
        RET
    ext_kod:
        MOV AL,AH       ;Kiterjesztett (scan) kód AL-be
        MOV AH, 1       ; AH =1, ha  kiterjesztett kódot adunk vissza
        RET
    read_key ENDP
```

8. Irodalomjegyzék

Goldstine, Herman H.: A számítógép Pascaltól Neumannig, Műszaki könyvkiadó, 2003.

Kovács Magda: 32 bites mikroprocesszorok 80386/80486; LSI oktatóközpont.

Marschik Iván: Mikrogéprendszerek tervezése, SZÁMALK, 1984.

Microsoft: MASM Programmer's Guide; Microsoft Corporation.

Pethő Ádám: IBMPC/XT felhasználóknak és programozóknak 1 – 3 kötet; SZÁMALK, 1990.

Peter Norton – John Socha: Az IBM PC assembly nyelvű programozása; Novotrade, 1991.

Pirkó József: Turbo Pascal 5.5; LSI kiadó, 1990.

Texas: Bevezetés a mikroprocesszor technikába, Műszaki kiadó, 1982.

Texas: TTL receptek, Műszaki kiadó, 1976.

9. Melléklet

10. Intel processzorok utasításkészlete *96*

11. Lebegőpontos utasításkészlet *123*

12. Megszakítások *134*

12.1. INT 10h funkció: képernyő driver hívása **134**

12.2. INT 16h funkció: klaviatúra driver hívása **136**

12.3. INT 21h funkció: DOS funkciók hívása **137**

12.4. INT 25h funkció: lemezszektor olvasása **141**

12.5. INT 26h funkció: lemezszektor írása **141**

12.6. INT 33h funkció: egér driver hívása **142**

13. A 7 bites ASCII kódtábla *145*

13.1. Alapkódok **145**

13.2. Billentyűkódok **146**

10. Az Intel processzorok utasításkészlete

AAA - ASCII Adjust for Addition (ASCII számok összeadása)

AAA

Két ASCII szám összeadása után használható utasítás. Hatására az AH és AL regiszterbe két pakolatlan BCD (Binary Coded Decimal) számjegy jön létre.

Módosított flagek: AF, CF (OF, PF, SF, ZF bizonytalan)

Például: Adjuk össze a “6” és a “9” ASCII számjegyeket AL = 36h + 39h = 6Fh (ez nem egyenlő a számjegyek összegével). Az AAA utasítás hatására a 6Fh ból az AH = 01h, az AL = 05h lesz, azaz AX = 15h (BCD számpár).

AAD - ASCII Adjust for Division (BCD számok előkészítése osztáshoz)

AAD

Az **AH** és **AL**-ben lévő BCD szám átalakítása bináris (hexadecimális) számmá.

Módosított flagek: SF ZF PF (AF,CF,OF bizonytalan)

AAM - ASCII Adjust for Multiplication (BCD számmá alakítás)

AAM

A szorzás eredményeként kapott értéket alakítja át pakolatlan BCD számmá.

Módosított flagek: PF, SF, ZF (AF, CF, OF bizonytalan)

AAS - ASCII Adjust for Subtraction

AAS

Két ASCII szám kivonása után használható utasítás. Hatására az AH és AL regiszterben két pakolatlan BCD (Binary Coded Decimal) számjegy jön létre.

Módosított flagek: AF, CF (OF, PF, SF, ZF bizonytalan)

Működése hasonló az AAA utasításéval.

ADC - Add With Carry (összeadás)

ADC cél, forrás

A *forrrás* operandust és a **CF**-et hozzáadja a *cél* operandushoz. Az eredmény a *cél* helyére kerül.

Módosított flagek: AF, CF, OF, SF, PF, ZF

ADD - Arithmetic Addition (összeadás)

ADD cél, forrás

A *forrás* operandust hozzáadja a *cél* operandushoz. Az eredmény a *cél* helyére kerül.

Módosított flagek: AF, CF, OF, PF, SF, ZF

A művelet 8, 16, 32 bites számokkal, tetszőleges címzési módban elvégezhető.

AND - Logical And (bitenkénti ÉS)

AND cél, forrás

A *forrás* **ÉS** *cél* logikai művelet végrehajtása. Az eredmény a *cél* helyére kerül.

Módosított flagek: CF, OF, PF, SF, ZF (AF bizonytalan)

ARPL - Adjusted Requested Privilege Level of Selector (szelektor RPL beállítása)

ARPL cél, forrás

Vezérlésátadás letiltása, ha a cél alacsonyabb prioritási szinten van, mint a forrás. Ha a cél RPL (alsó két bit) értéke kisebb, mint a forrás RPL értéke, akkor cél=cél+1 és ZF=1, egyébként cél változatlan és ZF=0. (286-tól használható védett (protected) módú utasítás.)

Módosított flagek: ZF

BOUND - Array Index Bound Check (tömbindex ellenőrzés)

BOUND cél, forrás

Az 5-ös megszakítás hívása, ha a *cél* operandus értéke kisebb a *forrás* által meghatározott memóriacímen tárolt értéknél, és nagyobb a következő memóriacímen lévőnél. Azaz, ha a *cél* értéke nem egy megadott intervallumon belül van. (80188-tól használható utasítás 16 illetve 32 bites operandusokkal.)

Módosított flagek: -

BSF - Bit Scan Forward (bitkeresés előre)

BSF cél, forrás

A *forrás*-ban megkeresi (jobbról) az első 1-be állított bitet, az indexét a *cél*-ba írja és **ZF**=0. Ha a *forrás* nem tartalmaz 1-et, a *cél* bizonytalan és **ZF**=1 lesz. (386-os processzortól használható.)

Módosított flagek: ZF

BSR - Bit Scan Reverse (bitkeresés vissza)

BSR cél, forrás

A *forrás*-ban megkeresi (balról) az első 1-be állított bitet, az indexét a *cél*-ba írja és **ZF**=0. Ha a *forrás* nem tartalmaz 1-et, a *cél* bizonytalan és **ZF**=1 lesz. (386-os processzortól használható.)

Módosított flagek: ZF

BSWAP - Byte Swap (bájt csere)

BSWAP regiszter

A *regiszter* [0 – 7] és [24 – 31] valamint a [8 – 15] és [16 – 23] bitjeinek cseréje. Például konvertálás 32 és 16 bites tárolási mód között. (486-os processzortól használható 32 bites regiszterekre.)

Módosított flagek: -

BT - Bit Test (bit teszt)

BT cél, forrás

A *cél* adat *forrás*-adik bitjének tárolása a **CF**-be. (Csak a 386-os processzortól használható.)

Módosított flagek: CF

BTC - Bit Test with Compliment (bit teszt)

BTC cél, forrás

A *cél* adat *forrás*-adik bitjének tárolása a **CF**-be, és az eredeti bit negálása. (Csak a 386-os processzortól használható.)

Módosított flagek: CF

BTR - Bit Test with Reset (bit teszt)

BTR cél, forrás

A *cél* adat *forrás*-adik bitjének tárolása a **CF**-be, és az eredeti bit törlése. (Csak a 386-os processzortól használható.)

Módosított flagek: CF

BTS - Bit Test and Set (bit teszt)

BTS cél, forrás

A *cél* adat *forrás*-adik bitjének tárolása a **CF**-be, és ez eredeti bit beállítása 1-re. (Csak a 386-os processzortól használható.)

Módosított flagek: CF

CALL - Procedure Call (szubrutinhívás)

CALL cím

A program működése a *cím* címkénél folytatódik. A **CS:IP** (utasításpointer) értékét a stack-re menti, majd a **CS:IP**-be a *cím* címe kerül.
(Közeli hívás esetén csak az **IP**-t módosítja.)

Módosított flagek: -

CBW - Convert Byte to Word (konvertálás bájtból szóba)

CBW

Az **AL**-ben lévő értéket **AX**-be konvertálja úgy, hogy a 8. biten lévő előjelet átmásolja a 16. bitre.

Módosított flagek: -

CDQ - Convert Double to Quad (konvertálás duplaszóból négybájtosba)

CDQ

EAX tartalmának konvertálása **EDX:EAX**-be. Csak 386-os processzortól.

Módosított flagek: -

CLC - Clear Carry (CF törlése)

CLC

Törli a **CF** jelzőbitet.

Módosított flagek: CF

CLD - Clear Direction Flag (DF törlése)

CLD

Törli a **DF** (direction) jelzőbitet. A sztringkezelő utasítások esetén az **SI** illetve **DI** indexregiszterek növekedni fognak.

Módosított flagek: DF

CLI - Clear Interrupt Flag (interrupt letiltás)

CLI

Törli az **IF** interrupt jelzőbitet. Letiltja a hardver megszakításkérések fogadását.

Módosított flagek: IF

CLTS - Clear Task Switched Flag (taskváltás-bit törlése)

CLTS

Törli a taszkváltás-bitet a gép állapot regiszterében. (286- as processzortól, privilegizált módban használható)

Módosított flagek: -

CMC - Complement Carry Flag (CF komplementálása)

CMC

Komplementálja (ellenkezőjére változtatja) a **CF**-et.

Módosított flagek: CF

CMP – Compare (összehasonlítás)

CMP cél, forrás

A *cél* és *forrás* összehasonlítása. A művelet során úgy állítja be a flag regisztert, mintha *cél – forrás* kivonást végzett volna.

Módosított flagek: AF, CF, OF, PF, SF, ZF

CMPS - Compare String (Byte, Word or Doubleword) (két sztring összehasonlítása)

CMPS cél, forrás

A *cél* címen és a *forrás* címen kezdődő sztringek összehasonlítása.

CMPSB

CMPSW

CMPSD (386-tól)

A **DS:SI** és az **ES:DI** regiszterpárokkal megadott kezdőcímű sztringek összehasonlítása.

Az SI és a DI, az irányflag (Direction Flag) állástól függően csökken vagy nő, CMPS (az operandus típusától függően) automatikusan, CMPB (bájt) eggyel, CMPW (word) kettővel és CMPD (double) néggyel. A REP utasítással együtt használható.

Módosított flagek: AF, CF, OF, PF, SF, ZF

CMPXCHG - Compare and Exchange (összehasonlítás és csere)

CMPXCHG cél, forrás

A *cél*-t összehasonlítja az akkumulátorral (**AL**, **AX**, **EAX**). Ha egyenlő, a *forrás* értéke a *cél*-ba kerül, egyébként a *cél* az akkumulátorba íródik. (Csak 486-os processzortól)

Módosított flagek: AF, CF, OF, PF, SF, ZF

CWD - Convert Word to Doubleword (konvertálás szóból duplaszóba)

CWD

AX tartalmának **DX:AX**-be konvertálása. Az előjel a **CX** legnagyobb helyi értékű bitje lesz.

Módosított flagek: -

CWDE - Convert Word to Extended Doubleword (konvertálás szóból duplaszóba)

CWDE

AX tartalmának **EAX**-be konvertálása. Csak 386-os processzortól.

Módosított flagek: -

DAA - Decímal Adjust for Addition (BCD számok összeadása)

DAA

Két BCD szám összeadása után az eredményt BCD alakra konvertálja.

Módosított flagek: AF, CF, PF, SF, ZF (OF bizonytalan)

Példa: Adjuk össze a 25h és a 69h BCD számokat, az eredmény 8Eh lesz. Alkalmazva a DAA utasítást kapjuk a helyes 94h eredményt.

DAS - Decimal Adjust for Subtraction (BCD számok kivonása)

DAS

Két BCD szám kivonása után az eredményt BCD alakra konvertálja.

Módosított flagek: AF, CF, PF, SF, ZF (OF bizonytalan)

Működése hasonló a DAA utasítással.

DEC – Decrement (csökkentés 1-el)

DEC cél

A *cél* operandus értékéből 1-et levon.

Módosított flagek: AF OF PF SF ZF

DIV – Divide (előjel nélküli osztás)

DIV forrás

16bit esetén: **AX** osztása a *forrás*-sal, az eredmény **AL**-be, a maradék **AH**-ba kerül.
32bit esetén: **DX:AX** osztása a *forrás*-sal, az eredmény **AX**-be, a maradék **DX**-be kerül.
64bit esetén: **EDX:EAX** osztása a *forrás*-sal, az eredmény **EAX**-be, a maradék **EDX**-be kerül.

Módosított flagek: (AF,CF,OF,PF,SF,ZF bizonytalan)

ENTER - Make Stack Frame (veremkeret előkészítése)

ENTER memória szint

A magasszintű nyelvek által használt veremkeret előkészítése. A *memória* adja meg, hogy mennyi dinamikus hely kell a hívott rutinnak, a *szint* a rutinok egymásba ágyazási szintjét adja. (80188-os processzortól.)

Módosított flagek: -

ESC – Escape (társprocesszor vezérlése)

ESC kód, forrás

A *kód* operációs kódot és a *forrás* memóriacímet kiírja a busz adat és címvonalára. Többprocesszoros rendszereknél a másik processzor vezérlésére szolgál. Ekkor a *kód* a processzornak szóló utasítás, a *forrás* pedig az operandus címe.

Módosított flagek: -

HLT - Halt CPU (CPU leállítása)

HLT

A processzor HALT (leállított) állapotba kerül. Folytatás csak hardver interrupt-al (ha az engedélyezett) vagy újraindítással (a RESET vonal 1-be állításával) lehetséges. Hardver interrupt esetén folytatás az aktuális **CS:IP** címtől.

Módosított flagek: -

IDIV - Signed Integer Division (előjeles osztás)

IDIV for

Hasonló a DIV utasításhoz. A műveletet előjelhelyesen hajtja végre.

Módosított flagek: (AF, CF, OF, PF, SF, ZF bizonytalan)

IMUL - Signed Multiply (előjeles szorzás)

IMUL forrás

8 bit esetén: **AL** tartalmát megszorozza a *forrás*-sal, az eredmény **AX**-be kerül.
16 bit esetén: **AX** tartalmát megszorozza a *forrás*-sal, az eredmény **DX:AX**-be kerül.
32 bit esetén: **EAX** tartalmát megszorozza a *forrás*-sal, az eredmény **EDX:EAX**-be kerül.

IMUL regiszter, konstans (286-tól)

A *regiszter*-t megszorozza a *konstans*-sal, az eredmény a *regiszter-be* kerül. (16 vagy 32 bites szorzás.)

IMUL regiszter, forrás, konstans (286-tól)

A *konstans*-t megszorozza a *forrás*-sal (regiszter vagy memóriacím), az eredmény a *regiszter*-be (16 bites!) kerül.

IMUL regiszter, forrás (386-tól)

A *regiszter*-t megszorozza a *forrás*-sal (regiszter vagy memóriacím), az eredmény a *regiszter-be* kerül. (16 vagy 32 bites szorzás.)

Módosított flagek: CF, OF (AF, PF, SF, ZF bizonytalan)

```
Példa:  imul    dx,514      ; DX szorzása 514-el, eredmény DX-be
        imul    ax, [bx], 12; BX címen lévő érték szorzása 12-vel, eredmény AX-be
        imul    dx, ax      ; DX és AX szorzása, eredmény DX-be
        imul    ax, [bx]    ; BX címen lévő érték szorzása AX-el, eredmény AX-ben
```

IN - Input Byte or Word From Port (portról adat beolvasás)

IN cél, port

A *cél* (**AL**, **AX** vagy **EAX**) feltöltése a *port* (8 bit) által címzett I/O portról.
A *port* helyett a **DX** regiszterben is megadható a száma, ekkor 16 bites lehet az értéke.
Megjegyzés: IBM PC-n 10 bites (0-1023) lehet a portszám.

Módosított flagek: -

INC – Increment (növelés 1-el)

INC cél

A *cél* operandus értékéhez 1-et hozzáad.

Módosított flagek: AF, OF, PF, SF, ZF

INS - Input String from Port (sztring beolvasás)

INS cél, port

Az ES:DI (ES:EDI) kezdőcímű memória feltöltése a *port*-ról. (80188-tól)

INSB

INSW

INSD (386-tól)

Az ES:DI (ES:EDI) kezdőcímű memória feltöltése a *port*-ról.
A DI, az irányflag (Direction Flag) állástól függően csökken vagy nő, INS (az operandus típusától függően) automatikusan, INSB (bájt) eggyel, INSW (word) kettővel és INSD (double) néggyel.
A *port* helyett itt is használható a **DX** regiszter (16 bit).

Módosított flagek: -

INT – Interrupt (megszakítás)

INT konstans

A *konstans* sorszámú szoftverinterrupt indítása. Hatására a stack-re iródik a **CS** és az **IP** majd a **CS:IP**-be kerül az interrupt-vektor táblából – a *konstans*-nak megfelelő – interrupt szoftver címe.

Módosított flagek: TF, IF

INTO - Interrupt on Overflow (megszakítás)

INTO

OF = 1 esetén indítja az INT 4 (0000:0010 című) interrupt-ot.

Módosított flagek: IF, TF

INVD - Invalidate Cache (belső cache törlése)

INVD

Törli a CPU belső cache-ét. Egy speciális buszciklus segítségével a külső cache is törölhető. (Csak 486-os processzortól.)

Módosított flagek: -

INVLPG - Invalidate Translation Look-Aside Buffer Entry (TLB belépés érvénytelenítése)

INVLPG

Törli a TLB (Translation Look-Aside Buffer) belépését. (Például a virtuális memória lapozásánál használható.) (Csak 486-os processzortól.)

Módosított flagek: -

IRET/IRETD - Interrupt Return (visszatérés megszakításból)

IRET

IRETD (386-tól)

Visszatérés interrupt rutinból. A stack-re elmentett visszatérési címet betölti a **CS:IP** regiszterpárba.

Módosított flagek: AF, CF, DF, IF, PF, SF, TF, ZF

JA/JNBE - Jump Above / Jump Not Below or Equal (ugrás, ha nagyobb)

JA cím

JNBE cím

Ugrás a *cím*-re, ha **CF = 0** és **ZF = 0**. Előjel nélküli *nagyobb* illetve *nem kisebb-egyenlő*.

Módosított flagek: -

JAE/JNB/JNC - Jump Above or Equal / Jump on Not Below / Jump Not Carry (ugrás, ha nem kisebb)

JAE cím

JNB cím

JNC cím

Ugrás a *cím*-re, ha **CF = 0**. Előjel nélküli *nagyobb-egyenlő* illetve *nem kisebb*.

Módosított flagek: -

JB/JNAE/JC - Jump Below / Jump Not Above or Equal / Jump on Carry (ugrás, ha kisebb)

JB cím

JNAE cím

JC cím

Ugrás a *cím*-re, ha **CF = 1**. Előjel nélküli *kisebb* illetve *nem nagyobb-egyenlő*.

Módosított flagek: -

JBE/JNA - Jump Below or Equal / Jump Not Above (ugrás, ha nem nagyobb)

JBE cím

JNA cím

Ugrás a *cím*-re, ha **CF = 1** vagy **ZF = 1**. Előjel nélküli *kisebb-egyenlő* illetve *nem nagyobb*.

Módosított flagek: -

JCXZ/JECXZ - Jump if Register (E)CX is Zero (ugrás, ha CX nulla)

JCXZ cím

JECXZ cím (386-tól)

Ugrás a *cím*-re, ha **CX = 0** illetve **ECX = 0**. Előjel nélküli összehasonlítás.

Módosított flagek: -

JE/JZ - Jump Equal / Jump Zero (ugrás, ha egyenlő)

JE cím

JZ cím

Ugrás a *cím*-re, ha **ZF = 1**. Előjel nélküli *egyenlő*.

Módosított flagek: -

JG/JNLE - Jump Greater / Jump Not Less or Equal (ugrás, ha nagyobb)

JG cím

JNLE cím

Ugrás a *cím*-re, ha **ZF = 0** és **SF = OF**. Előjeles *nagyobb* illetve *nem kisebb-egyenlő*.

Módosított flagek: -

JGE/JNL - Jump Greater or Equal / Jump Not Less (ugrás, ha nem kisebb)

JGE cím

JNL cím

Ugrás a *cím*-re, ha **SF = OF**. Előjeles *nagyobb-egyenlő* illetve *nem kisebb*.

Módosított flagek: -

JL/JNGE - Jump Less / Jump Not Greater or Equal (ugrás, ha kisebb)

JL cím

JNGE cím

Ugrás a *cím*-re, ha **SF ≠ OF**. Előjeles *kisebb* illetve *nem nagyobb-egyenlő*.

Módosított flagek: -

JLE/JNG - Jump Less or Equal / Jump Not Greater (ugrás, ha nem nagyobb)

JLE cím

JNG cím

Ugrás a *cím*-re, ha **ZF = 1** or **SF ≠ OF**. Előjeles *kisebb-egyenlő* illetve *nem nagyobb*.

Módosított flagek: -

JMP - Unconditional Jump (feltétel nélküli vezérlésátadás)

JMP cím

Vezérlésátadás a *cím* címre. A **CS:IP** utasításregiszterbe beírja a *cím*-et. Közeli hívás esetén csak az **IP** változik.

Módosított flagek: -

JNE/JNZ - Jump Not Equal / Jump Not Zero (ugrás, ha nem egyenlő)

JNE cím

JNZ cím

Ugrás a *cím*-re, ha **ZF = 0**. Előjel nélküli *nem egyenlő*.

Módosított flagek: -

JNO - Jump Not Overflow (ugrás, ha nincs túlcsordulás)

JNO cím

Ugrás a *cím*-re, ha **OF = 0**. Előjeles összehasonlítás.

Módosított flagek: -

JNS - Jump Not Signed (ugrás, ha nincs előjel)

JNS cím

Ugrás a *cím*-re, ha **SF = 0**. Előjeles összehasonlítás.

Módosított flagek: -

JNP/JPO - Jump Not Parity / Jump Parity Odd (ugrás, ha a paritás páratlan)

JNP cím

JPO cím

Ugrás a *cím*-re, ha **PF = 0**.

Módosított flagek: -

JO - Jump on Overflow (ugrás túlcsordulás esetén)

JO cím

Ugrás a *cím*-re, ha **OF = 1**.

Módosított flagek: -

JP/JPE - Jump on Parity / Jump on Parity Even (ugrás, páros paritás esetén)

JP cím

JPE cím

Ugrás a *cím*-re, ha **PF = 1**.

Módosított flagek: -

JS - Jump Signed (ugrás, ha az eredmény negatív)

JS cím

Ugrás a *cím*-re, ha **SF = 1**. Előjeles összehasonlítás.

Módosított flagek: -

LAHF - Load Register AH From Flags (flag regiszter másolása AH-ba)

LAHF

A *flag* regiszter alsó 8 bitjét **AH**-ba másolja.

Módosított flagek: -

LAR - Load Access Rights (hozzáférési jog betöltése)

LAR cél, forrás

A *forrás* által leírt szelektor tartalmát a cél felső bájtjába írja, az alsó bájtba nulla kerül. **ZF** értéke 1, ha a betöltés sikeres. (286-os processzortól, védett módban)

Módosított flagek: ZF

LDS - Load Pointer Using DS (mutató betöltése)

LDS cél, forrás

A *forrás* (32 bites) pointer betöltése a **DS:cél** regiszterpárba. A *forrás* első és második bájtja a *cél* regiszterbe, a harmadik és negyedik bájtja a **DS**-be kerül. Távoli (far) pointerek meghatározásánál használható. Segítségével címezhetünk másik szegmensen lévő memóriát.

Módosított flagek: -

LEA - Load Effective Address (szegmensen belüli memóriacím betöltése)

LEA cél, forrás

A *forrás* offset címének betöltése a *cél* operandusba.

Módosított flagek: -

(Hatása megegyezik a *MOV cél, OFFSET forrás* utasítással.)

LEAVE - Restore Stack for Procedure Exit (kilépés az aktuális veremkeretből)

LEAVE

Lokális változók felszabadítása (ENTER utasítás fordítottja). Visszaállítja az eredeti (ENTER előtti) **SP** és **BP** regiszterek tartalmát. Így a következő RET utasítással visszatérhetünk a hívó eljáráshoz.

Módosított flagek: -

LES - Load Pointer Using ES (mutató betöltése)

LES cél, forrás

A *forrás* (32 bites) pointer betöltése az **ES:cél** regiszterpárba. A *forrás* első és második bájtja a *cél* regiszterbe, a harmadik és negyedik bájtja az **ES**-be kerül. Távoli (far) pointerek meghatározásánál használható. Segítségével címezhetünk másik szegmensen lévő memóriát.

Módosított flagek: -

LFS - Load Pointer Using FS (mutató betöltése)

LFS cél, forrás

A *forrás* (32 bites) pointer betöltése az **ES:cél** regiszterpárba. A *forrás* első és második bájtja a *cél* regiszterbe, a harmadik és negyedik bájtja az **FS**-be kerül. Távoli (far) pointerek meghatározásánál használható. Segítségével címezhetünk másik szegmensen lévő memóriát. (386-os processzortól.)

Módosított flagek: -

LGDT - Load Global Descriptor Table (adatbetöltés a GDT táblába)

LGDT forrás

A *forrás* címen lévő adat betöltése a GDT-be (Global Descriptor Table). (286-os processzortól, csak védett módban.)

Módosított flagek: -

LIDT - Load Interrupt Descriptor Table (adatbetöltés az IDT táblába)

LIDT forrás

A *forrás* címen lévő adat betöltése az IDT-be (Interrupt Descriptor Table). (286-os processzortól, csak védett módban.)

Módosított flagek: -

LGS - Load Pointer Using GS (mutató betöltése)

LGS cél, forrás

A *forrás* (32 bites) pointer betöltése az **ES:cél** regiszterpárba. A *forrás* első és második bájtja a *cél* regiszterbe, a harmadik és negyedik bájtja az **GS**-be kerül. Távoli (far) pointerek meghatározásánál használható. Segítségével címezhetünk másik szegmensen lévő memóriát. (386-os processzortól.)

Módosított flagek: -

LLDT - Load Local Descriptor Table (adatbetöltés az LDTR táblába)

LLDT forrás

A *forrás* betöltése az LDTR-be (Local Descriptor Table Register). (286-os processzortól, csak védett módban.)

Módosított flagek: -

LMSW - Load Machine Status Word (adatbetöltés az MSW-be)

LMSW forrás

A *forrás* betöltése az MSW-be (Machine Status Word). (286-os processzortól, csak védett módban.)

Módosított flagek: -

LOCK - Lock Bus (CPU buszainak lezárása)

LOCK

Utasítás előtt (prefix) kell megadni. Használatakor – az utasítás végrehajtásának idejére – lezárja a CPU buszait, hogy a végrehajtás során (például több processzor esetén) a processzor ne fogadhasson újabb utasítást.

Módosított flagek: -

LODS - Load String (Byte, Word or Double)

LODSB

LODSW

LODSD (386-tól)

A **DS:SI** regiszterpárokkal megadott kezdőcímű sztring másolása **AL, AX** vagy **EAX** regiszterbe.

Az SI az irányflag (Direction Flag) állástól függően csökken vagy nő, LODS (az operandus típusától függően) automatikusan, LODSB (bájt) eggyel, LODSW (word) kettővel és LODSD (double) néggyel. A LOOP utasítással együtt használható.

Módosított flagek: -

LOOP - Decrement CX and Loop if CX Not Zero (ciklus)

LOOP cím

A LOOP utasítás és a *cím* közötti utasításokat a **CX**-ben tárolt számszor hajtja végre. A LOOP a **CX** értékét eggyel csökkenti és a *cím*-et az **IP**-be írja, ezt a **CX=0** értékig folytatja. A LOOP és a feltételes vezérlésátadásnál csak 8 bites relatív címzés lehetséges. Azaz -128 vagy 127 bájtnál nem lehet hosszabb az ugrás.

Módosított flagek: -

LOOPE/LOOPZ - Loop While Equal / Loop While Zero (ciklus)

LOOPE cím

LOOPZ cím

A ciklus addig tartat, amíg **CX** vagy a **ZF** nem nulla.
Működése hasonló a LOOP-hoz.

Módosított flagek: -

LOOPNZ/LOOPNE - Loop While Not Zero / Loop While Not Equal

LOOPNZ cím

LOOPNE cím

A ciklus addig tartat, amíg **CX** nem nulla és a **ZF** nulla.
Működése hasonló a LOOP-hoz.

Módosított flagek: -

LSL - Load Segment Limit (szegmens-határérték betöltése)

LSL cél, forrás

A szegmens-határérték (*forrás*) betöltése a *cél* regiszterbe. Sikeres betöltés esetén **ZF** egybe áll, egyébként nulla. (286-os processzortól, védett módban.)

Módosított flagek: ZF

LSS - Load Pointer Using SS (mutató betöltése)

LSS cél, forrás

A *forrás* (32 bites) pointer betöltése az **ES:cél** regiszterpárba. A *forrás* első és második bájtja a *cél* regiszterbe, a harmadik és negyedik bájtja az **SS**-be kerül. Távoli (far) pointerek meghatározásánál használható. Segítségével címezhetünk másik szegmensen lévő memóriát. (386-os processzortól.)

Módosított flagek: -

LTR - Load Task Register (taszkregiszter betöltése)

LTR forrás

A *forrás* betöltése a taszk regiszterbe.

Módosított flagek: -

MOV - Move Byte or Word (bájt vagy szó mozgatása)

MOV cél, forrás

A *forrás* tartalmát a *cél* operandusba írja. (A *cél* eredeti értéke elvész.)

Módosított flagek: -

MOVS - Move String (sztring mozgatás)

MOVS cél, forrás

A *forrás* címen kezdődő sztring másolása, a *cél* címre.

MOVSB

MOVSW

MOVSD (386-tól)

A **DS:SI** regiszterpárokkal megadott kezdőcímű sztring másolása az **ES:DI** címre.

Az SI és a DI, az irányflag (Direction Flag) állástól függően csökken vagy nő, MOVS (az operandus típusától függően) automatikusan, MOVSB (bájt) eggyel, MOVSW (word) kettővel és MOVSD (double) néggyel. A REP utasítással együtt használható.

Módosított flagek: -

MOVSX - Move with Sign Extend (előjeles adatmozgatás)

MOVSX cél, forrás

A *forrás* operandus értékének másolása a *cél* regiszterbe előjelhelyesen. (Például 8 bites *forrás* és 16 bites *cél* esetén előjelhelyes lesz az átvitel.) (Csak 386-os processzortól.)

Módosított flagek: -

MOVZX - Move with Zero Extend (adatmozgatás)

MOVZX cél, forrás

A *forrás* operandus értékének előjel nélküli másolása a *cél* regiszterbe. Különböző hosszúságú operandusok (rövidebb *cél*) esetén a plusz biteket nullával tölti fel. (Csak 386-os processzortól.)

Módosított flagek: -

MUL - Unsigned Multiply (előjel nélküli szorzás)

MUL forrás

8 bit esetén: **AL** tartalmát megszorozza a *forrás*-sal, az eredmény **AX**-be kerül.
16 bit esetén: **AX** tartalmát megszorozza a *forrás*-sal, az eredmény **DX:AX**-be kerül.
32 bit esetén: **EAX** tartalmát megszorozza a *forrás*-sal, az eredmény **EDX:EAX**-be kerül.

Módosított flagek: CF, OF (AF, PF, SF, ZF bizonytalan)

NEG - Two's Complement Negation (negálás)

NEG cél

A *cél* értékét negálja, a kettes komplemensét képzi.

Módosított flagek: AF, CF, OF, PF, SF, ZF

NOP - No Operation (nincs művelet)

NOP

Üres utasítás.

Módosított flagek: -

NOT - One's Compliment Negation (logikai NEM)

NOT cél

A *cél* tartalmát bitenként negálja (egyes komplemens).

Módosított flagek: -

OR - Inclusive Logical OR (bitenkénti VAGY)

OR cél, forrás

A *forrás* **OR** *cél* logikai művelet végrehajtása. Az eredmény a *cél* helyére kerül.

Módosított flagek: CF, OF, PF, SF, ZF (AF bizonytalan)

OUT - Output Data to Port

OUT port, forrás

A *forrás* (**AL**, **AX** vagy **EAX**) tartalmának kiírása a *port* (8 bit) által címzett I/O portra.
A *port* helyett a **DX** regiszterben is megadható a száma, ekkor 16 bites lehet az értéke.
Megjegyzés: IBM PC-n 10 bites (0-1023) lehet a portszám.

Módosított flagek: -

OUTS - Output String to Port (sztring kiírása)

OUTS port, forrás

Az ES:DI (ES:EDI) címen kezdődő sztring kiírása a *port*-ra. (80188-tól)

OUTSB

OUTSW

OUTSD (386-tól)

Az ES:DI (ES:EDI) címen kezdődő sztring kiírása a *port*-ra.

A DI, az irányflag (Direction Flag) állástól függően csökken vagy nő, OUTS (az operandus típusától függően) automatikusan, OUTSB (bájt) eggyel, OUTSW (word) kettővel és OUTSD (double) néggyel.
A *port* helyett itt is használható a **DX** regiszter (16 bit).

Módosított flagek: -

POP - Pop Word off Stack (adat visszatöltés stack-ből)

POP cél

A *stack* tetején (**SS:SP**) lévő adat beírása a *cél* operandusba. Az **SP** regiszter értékét 2-vel (16 bites szóhossz) vagy 4-el (32 bites szóhossz) növeli.

Módosított flagek: -

POPA/POPAD - Pop All Registers onto Stack (adat visszatöltés stack-ből)

POPA (80188-tól)

POPAD (386-tól)

A *stack* tetején lévő 8 adat beírása az **(E)DI, (E)SI, (E)BP, (E)SP, (E)BX, (E)DX, (E)CX, (E)AX** regiszterekbe. (A megadott sorrendben!) Az **SP** értékét aktualizálja.

Módosított flagek: -

POPF/POPFD - Pop Flags off Stack (flag regiszter feltöltése a stack-ből)

POPF

A *stack* tetején (**SS:SP**) lévő adat beírása a *flag* regiszterbe. Az **SP** regiszter értékét 2-vel (16 bites szóhossz) vagy 4-el (32 bites szóhossz) növeli.

Módosított flagek: összes flag

386-os (32 bites) processzortól használható a POPFD utasítás.

PUSH - Push Word onto Stack (adat mentése a stack-be)

PUSH forrás

PUSH immed (80188-tól)

A *forrás* (1 szó hosszú) tárolása a stack-ben. Az **SP** regiszter értékét 2-vel (16 bites szóhossz) vagy 4-el (32 bites szóhossz) csökkenti.

Módosított flagek: -

A stack memória címét az **SS:SP** regiszterpár tartalmazza.

PUSHA/PUSHAD - Push All Registers onto Stack (adat mentése a stack-be)

PUSHA (80188-tól)

PUSHAD (386-tól)

Az **(E)AX, (E)CX, (E)DX, (E)BX, (E)SP, (E)BP, (E)SI, (E)DI** regiszterek tartalmának másolása a stack-re. (A megadott sorrendben!) Az **SP** értékét aktualizálja.

Módosított flagek: -

PUSHF/PUSHFD - Push Flags onto Stack (flag regiszter mentése a stack-be)

PUSHF

A *flag* regiszter (1 szó hosszú) tárolása a stack-ben. Az **SP** regiszter értékét 2-vel (16 bites szóhossz) vagy 4-el (32 bites szóhossz) csökkenti.

Módosított flagek: -

386-os (32 bites) processzortól használható a PUSHFD utasítás.

RCL - Rotate Through Carry Left (rotálás balra carry-n keresztül)

RCL cél, szám

A *cél* bitenkénti rotálása balra, a **CF**-en keresztül, *szám*-szor. A baloldalon kikerülő bit a **CF**-be iródik, a **CF** régi értéke a legkisebb helyi értékre kerül.

Módosított flagek: CF, OF

RCR - Rotate Through Carry Right (rotálás jobbra carry-n keresztül)

RCR cél, szám

A *cél* bitenkénti rotálása jobbra, a **CF**-en keresztül, *szám*-szor. A jobboldalon kikerülő bit a **CF**-be íródik, a **CF** régi értéke a legmagasabb helyi értékre kerül.

Módosított flagek: CF, OF

REP - Repeat String Operation (sztringkezelő utasítások ismétlése)

REP utasítás

Az *utasítás* többszöri végrehajtása, amíg a **CX** nem nulla. Az ismétlések számát **CX**-be kell tölteni. A REP utasítás a **CX** értékét – műveletenként – eggyel csökkenti. (Prefix utasítás)

Módosított flagek: -

REPE/REPZ - Repeat Equal / Repeat Zero (sztringkezelő utasítások ismétlése)

REPE utasítás

REPZ utasítás

Az *utasítás* többszöri végrehajtása, amíg a **CX** vagy a **ZF** nem nulla (addig, amíg a karakterek megegyeznek). Az ismétlések számát **CX**-be kell tölteni. Az utasítás a **CX** értékét – műveletenként – eggyel csökkenti. (Prefix utasítás)

Módosított flagek: -

REPNE/REPNZ - Repeat Not Equal / Repeat Not Zero (sztringkezelő utasítások ismétlése)

REPNE utasítás

REPNZ utasítás

Az *utasítás* többszöri végrehajtása, amíg a **CX** nem nulla és a **ZF** nulla (addig, amíg a karakterek különbözőek). Az ismétlések számát **CX**-be kell tölteni. Az utasítás a **CX** értékét – műveletenként – eggyel csökkenti. (Prefix utasítás)

Módosított flagek: -

RET/RETF - Return From Procedure (visszatérés szubrutinból)

RET konstans

RETF konstans

RETN konstans

A stack tetején lévő értéket betölti a **CS:IP** regiszterpárba, így a következő utasítást az új **CS:IP** címről veszi. Amennyiben a *konstans*-t (bájtos érték) megadjuk, akkor a visszatérési cím beállítása után az SP-t *konstans* számú bájtal megnöveli (adatátadás stack-en keresztül).
RETF (far) távoli, RETN (near) közeli híváskor használható. Közeli híváskor csak az **IP** változik.

Módosított flagek: -

ROL - Rotate Left (rotálás balra)

ROL cél, szám

A *cél* bitenkénti rotálása balra, *szám*-szor. A baloldalon kikerülő bit a legkisebb helyi értékre és a **CF**-be íródik.

Módosított flagek: CF, OF

ROR - Rotate Right (rotálás jobbra)

ROR cél, szám

A *cél* bitenkénti rotálása jobbra, *szám*-szor. A jobboldalon kikerülő bit a legmagasabb helyi értékre és a **CF**-be íródik.

Módosított flagek: CF, OF

SAHF - Store AH Register into FLAGS (AH-t a flag regiszterbe másol)

SAHF

Az AH regiszter tartalmát a flag regiszter alsó 8 bitjére másolja.

Módosított flagek: AF, CF, PF, SF, ZF

SAL - Shift Arithmetic Left (aritmetikai shift balra)

SAL cél, szám

A *cél* bitenkénti shiftelése balra *szám*-szor. A baloldalon kikerülő bit a **CF**-be íródik, a legkisebb helyi értékre **0** kerül.

Módosított flagek: CF, OF, PF, SF, ZF (AF bizonytalan)

SAR - Shift Arithmetic Right (aritmetikai shift jobbra)

SAR cél, szám

A *cél* bitenkénti shiftelése jobbra *szám*-szor. A jobboldalon kikerülő bit a **CF**-be íródik, a legmagasabb helyi értékre az utolsó bit értéke (az előjel) kerül.

Módosított flagek: CF, OF, PF, SF, ZF (AF bizonytalan)

SBB - Subtract with Borrow (kivonás)

SBB cél, forrás

A *cél*-ből kivonja a *forrás* és a **CF** értékét az eredmény a *cél*-ba kerül.

Módosított flagek: AF, CF, OF, PF, SF, ZF

SCAS - Scan String (Byte, Word or Doubleword) (sztring karakterenkéti vizsgálata)

SCAS cél

Az **AL**, **AX**, vagy **EAX** regiszter tartalmát összehasonlítja a *cél* cím tartalmával.

SCASB

SCASW

SCASD (386-tól)

Az **AL, AX** vagy **EAX** regiszter tartalmát összehasonlítja **ES:DI** regiszterpárokkal megcímzett értékkel.

A **DI**, az irányflag (Direction Flag) állástól függően csökken vagy nő, SCAS (az operandus típusától függően) automatikusan, SCASB (bájt) eggyel, SCASW (word) kettővel és SCASD (double) néggyel. A LOOP vagy REP utasítással együtt használható.

Módosított flagek: AF, CF, OF, PF, SF, ZF

SETAE/SETNB - Set if Above or Equal / Set if Not Below (bájt beállítás, ha nem kisebb)

SETAE cél

SETNB cél

A *cél* = 1, ha **CF** = 0 (nem kisebb), egyébként: *cél* = 0. (Előjel nélküli utasítás 386-os processzortól.)

Módosított flagek: -

SETB/SETNAE - Set if Below / Set if Not Above or Equal (bájt beállítás, ha kisebb)

SETB cél

SETNAE cél

A *cél* = 1, ha **CF** = 1 (kisebb), egyébként: *cél* = 0. (Előjel nélküli utasítás 386-os processzortól.)

Módosított flagek: -

SETBE/SETNA - Set if Below or Equal / Set if Not Above (bájt beállítás, nem nagyobb)

SETBE cél

SETNA cél

A *cél* = 1, ha **CF** = 1 vagy **ZF** = 1 (nem nagyobb), egyébként: *cél* = 0. (Előjel nélküli utasítás 386-os processzortól.)

Módosított flagek: -

SETNBE/SETA - Set if Below or Equal / Set if Not Above (bájt beállítás, nagyobb)

SETNBE cél

SETA cél

A *cél* = 1, ha **CF** = 0 és **ZF** = 0 (nagyobb), egyébként: *cél* = 0. (Előjel nélküli utasítás 386-os processzortól.)

Módosított flagek: -

SETE/SETZ - Set if Equal / Set if Zero (bájt beállítás, ha egyenlő)

SETE cél

SETZ cél

A *cél* = 1, ha **ZF** = 1 (egyenlő), egyébként: *cél* = 0. (386-os processzortól.)

Módosított flagek: -

SETNE/SETNZ - Set if Not Equal / Set if Not Zero (bájt beállítás, ha nem egyenlő)

SETNE cél

SETNZ cél

A *cél* = 1, ha **ZF** = 0 (nem egyenlő), egyébként: *cél* = 0. (386-os processzortól.)

Módosított flagek: -

SETL/SETNGE - Set if Less / Set if Not Greater or Equal (bájt beállítás, ha kisebb)

SETL cél

SETNGE cél

A *cél* = 1, ha **SF** ? **OF** (kisebb), egyébként: *cél* = 0. (Előjeles utasítás 386-os processzortól.)

Módosított flagek: -

SETGE/SETNL - Set if Greater or Equal / Set if Not Less (bájt beállítás, kisebb)

SETGE cél

SETNL cél

A *cél* = 1, ha **CS** = **OF** (kisebb), egyébként: *cél* = 0. (Előjeles utasítás 386-os processzortól.)

Módosított flagek: -

SETLE/SETNG - Set if Less or Equal / Set if Not greater or Equal (bájt beállítás, ha nem nagyobb)

SETLE cél

SETNG cél

A *cél* = 1, ha **ZF** = 1 vagy **SF** ? **OF** (nem nagyobb), egyébként: *cél* = 0. (Előjeles utasítás 386-os processzortól.)

Módosított flagek: -

SETG/SETNLE - Set if Greater / Set if Not Less or Equal (bájt beállítás, nagyobb)

SETG cél

SETNLE cél

A *cél* = 1, ha **ZF** = 0 vagy **SF** = **OF** (nagyobb), egyébként: *cél* = 0. (Előjeles utasítás 386-os processzortól.)

Módosított flagek: -

SETS - Set if Signed (bájt beállítás, ha van előjel)

SETS cél

A *cél* = 1, ha **SF** = 1, egyébként: *cél* = 0. (386-os processzortól.)

Módosított flagek: -

SETNS - Set if Not Signed (bájt beállítás, ha nincs előjel)

SETNS cél

A *cél* = 1, ha **SF** = 0 (nem kisebb), egyébként: *cél* = 0. (386-os processzortól.)

Módosított flagek: -

SETC - Set if Carry (bájt beállítás, ha van átvitel)

SETC cél

A *cél* = 1, ha **CF** = 1, egyébként: *cél* = 0. (386-os processzortól.)

Módosított flagek: -

SETNC - Set if Not Carry (bájt beállítás, ha nincs átvitel)

SETNC cél

A *cél* = 1, ha **CF** = 0, egyébként: *cél* = 0. (386-os processzortól.)

Módosított flagek: -

SETO - Set if Overflow (386-tól) (bájt beállítás, ha van túlcsordulás)

SETO cél

A *cél* = 1, ha **OF** = 1, egyébként: *cél* = 0. (386-os processzortól.)

Módosított flagek: -

SETNO - Set if Not Overflow (bájt beállítás, ha nincs túlcsordulás)

SETNO cél

A *cél* = 1, ha **OF** = 0, egyébként: *cél* = 0. (386-os processzortól.)

Módosított flagek: -

SETP/SETPE - Set if Parity / Set if Parity Even (bájt beállítás, páros paritás esetén)

SETP cél

SETPE cél

A *cél* = 1, ha **PF** = 1, egyébként: *cél* = 0. (386-os processzortól.)

Módosított flagek: -

SETNP/SETPO - Set if No Parity / Set if Parity Odd (bájt beállítás, páratlan paritás esetén)

SETNP cél

SETPO cél

A *cél* = 1, ha **PF** = 0, egyébként: *cél* = 0. (386-os processzortól.)

Módosított flagek: -

SGDT - Store Global Descriptor Table (GDT regiszter tárolása)

SGDT cél

A GDR (Global Descriptor Table) regiszterének tárolása a *cél*-ban megadott memóriába. (286-os processzortól, védett módban.)

Módosított flagek: -

SHL - Shift Logical Left (logikai shift balra)

SHL cél, szám

A *cél* bitenkénti shiftelése balra, *szám*-szor. A baloldalon kikerülő bit a **CF**-be íródik, a legkisebb helyi értékre **0** kerül.

Módosított flagek: CF, OF, PF, SF, ZF

SHR - Shift Logical Right (logikai shift jobbra)

SHR cél, szám

A *cél* bitenkénti shiftelése jobbra, *szám*-szor. A jobboldalon kikerülő bit a **CF**-be íródik, a legmagasabb helyi értékre **0** kerül.

Módosított flagek: CF, OF, PF, SF, ZF (AF bizonytalan)

SHLD/SHRD - Double Precision Shift (duplapontos léptetés)

SHLD cél, forrás, szám

SHRD cél, forrás, szám

A *cél* bitenkénti shiftelése balra (SHLD) vagy jobbra (SHRD), *szám*-szor (a *forrás* regiszterrel együtt). A belépő bitek megegyeznek a *forrás*-ból kilépővel. (386-os processzortól.)

Módosított flagek: CF, PF, SF, ZF, (OF, AF bizonytalan)

SIDT - Store Interrupt Descriptor Table (IDT regiszter tárolása)

SIDT cél

Az IDT (Interrupt Descriptor Table) regiszterének betöltése a *cél* által megadott memóriába. (286-os processzortól, védett üzemmódban.)

Módosított flagek: -

SLDT - Store Local Descriptor Table (LDT regiszter tárolása)

SLDT cél

Az LDT (Local Descriptor Table) regiszterének betöltése a *cél* által megadott memóriába. (286-os processzortól, védett üzemmódban.)

Módosított flagek: -

SMSW - Store Machine Status Word (MSW tárolása)

SMSW cél

Az MSW (Machine Status Word) betöltése a *cél*-ba. (286-os processzortól, védett üzemmódban.)

Módosított flagek: -

STC - Set Carry (CF egyre állítása)

STC

A **CF** (carry) bitet egyre állítja.

Módosított flagek: CF

STD - Set Direction Flag (DF egyre állítása)

STD

A **DF** (direction) jelzőbitet egyre állítja. A sztringkezelő utasítások esetén az **SI** illetve **DI** indexregiszterek csökkeni fognak.

Módosított flagek:DF

STI - Set Interrupt Flag (interrupt engedélyezése)

STI

Az **IF** interrupt jelzőbitet egybe állítja. Engedélyezi a hardver megszakítások fogadását.

Módosított flagek:IF

STOS - Store String (Byte, Word or Doubleword) (sztring tárolása)

STOSB

STOSW

STOSD

Az **AL, AX** vagy **EAX** regiszter tartalmát **ES:DI** regiszterpárokkal megadott kezdőcímű sztringbe másolja.

A **DI**, az irányflag (Direction Flag) állástól függően csökken vagy nő, STOS (az operandus típusától függően) automatikusan, STOSB (bájt) eggyel, STOSW (word) kettővel és STOSD (double) néggyel. A LOOP vagy REP utasítással együtt használható.

Módosított flagek: -

STR - Store Task Register (taszk-regiszter tárolása)

STR cél

A taszk-regisztert a *cél*-ban tárolja. (286-os processzortól, védett üzemmódban.)

Módosított flagek: -

SUB – Subtract (kivonás)

SUB cél, forrás

A *cél*-ből kivonja a *forrás* értékét az eredmény a *cél*-ba kerül.

Módosított flagek: AF, CF, OF, PF, SF, ZF

TEST - Test For Bit Pattern (ÉS teszt)

TEST cél, forrás

Megegyezik a logikai AND művelettel (AND *cél*, *forrás*), csak a jelzőbiteket állítja, a *cél* értéke nem változik.

Módosított flagek: CF, OF, PF, SF, ZF (AF bizonytalan)

VERR/VERW - Verify Read/Write (szegmens ellenőrzése)

VERR forrás

Ha a *forrás*-ban megadott szegmens olvasható/írható, akkor **ZF** = 1, egyébként **ZF** = 0. (286-os processzortól, védett üzemmódban.)

Módosított flagek: ZF

WAIT/FWAIT - Event Wait (eseményre várás)

WAIT

FWAIT

A processzor várakozó állapotba kerül mindaddig, amíg a TEST kivezetésen 0 (low) szintet nem kap. A wait állapot biztosítja a lassúbb egységekkel (DMA, memória, coprocesszor) történő szinkronizálást.

Módosított flagek: -

WBINVD - Write-Back and Invalidate Cache (cache visszaírás)

WBINVD

A CPU belső cache-ét kiírja a memóriába, majd törli azt. (Csak 486-os processzortól.)

Módosított flagek: -

XCHG – Exchange (csere)

XCHG cél, forrás

A cél és a forrás értékeinek felcserélése.

Módosított flagek: -

XLAT – Translate (áttöltés)

XLAT

Az **AL** regiszterbe tölti a **BX:AL** (segment:offset) memóriacím tartalmát. Segítségével egy maximum 256 bájt hosszúságú adattábla **AL**-edik elemét olvashatjuk ki. A tábla kezdőcíme **BX**-ben van.

Módosított flagek: -

A MASM 5.x-től használható helyette az **XLATB** utasítás.

XOR - Exclusive OR (bitenkénti KIZÁRÓ VAGY)

XOR cél, forrás

A *forrás* **XOR** *cél* logikai művelet végrehajtása. Az eredmény a *cél* helyére kerül.

Módosított flagek: CF, OF, PF, SF, ZF (AF bizonytalan)

11. Lebegőpontos utasításkészlet

Az utasítások ismertetése után felsoroljuk azokat a matematikai processzor típusokat, amelyek használják az adott utasítást. (A 486-os processzortól már mindegyik lebegőpontos utasítás használható, így ezek nem szerepelnek a felsorolásban.)

Az ST(0) lebegőpontos regisztert szokás ST-ként is nevezni.

F2XM1 – 2^x-1 (Y = 2^x – 1 kiszámítása)

F2XM1

Paraméter nélküli utasítás. X-et az ST(0)-ról veszi és Y értékét ST(0)-ra teszi.

FABS – ABSolute value (Valós számok abszolút értéke)

FABS

ST(0) ban lévő érték abszolút értéke.

FADD/FADDP/FIADD – Add (Valós számok összeadása)

FADD

Paraméter nélküli utasítás. ST(1)-be másolja ST(1) és ST(0) összegét.
ST(1) lesz a stack teteje (ST).

FADDP *reg, ST* *(pl. FADDP ST(6), ST)*

ST(0) és egy tetszőleges lebegőpontos *regiszter* összeadása, eredmény a *regiszterbe*.
Egy elemet leemel a veremből.

FADD *ST, reg*

ST(0) és egy tetszőleges lebegőpontos *regiszter* összeadása, eredmény ST(0)-ba.

FADD *reg, ST*

ST(0) és egy tetszőleges lebegőpontos *regiszter* összeadása, eredmény a *regiszterbe*.
Egy elemet leemel a veremből.

FADD *forrás* *(pl. FADD QWORD PTR [BX])*

ST(0)-ba másolja ST(0) és a *forrás* összegét. (A forrás 32 és 64 bites lehet.)

FIADD *forrás* *(pl. FADD data[DI])*

ST(0)-ba másolja ST(0) és a *forrás* összegét. (A forrás egész típusú.)

FADD *ST(n)*

ST(0)-ba másolja ST(0) és ST(n) összegét.

FCHS – Change Sign (Előjelváltás)

FCHS

ST(0) előjelét megváltoztatja.

FCLEX/FNCLEX – Clear Exception (Kivételek törlése)

FCLEX

Törli az állapotszóban a kivétel, az **IR** és a **B** jelzőbiteket.

FNCLEX

Törli az állapotszóban a kivétel, az **IR** és a **B** jelzőbiteket. Nem végez ellenőrzést.

FCOM/FCOMP/FCOMPP/FICOM/FICOMP – Compare (Összehasonlítás)

FCOM

ST(0) és ST(1) összehasonlítása.

FCOM *reg* *(pl. FCOM ST(3))*

ST(0) és ST(n) összehasonlítása.

FCOM *forrás* *(pl. FCOM data[DI])*

ST(0) és a *forrás* összehasonlítása. (A *forrás* 32 és 64 bites adat lehet)

FCOMP *reg*

ST(0) és ST(n) összehasonlítása. Veremről leemeli az ST(0)-t.

FCOMP *forrás*

ST(0) és a *forrás* összehasonlítása. (A *forrás* 32 és 64 bites adat lehet)
Veremről leemeli az ST(0)-t.

FCOMPP

ST(0) és ST(1) összehasonlítása. Veremről leemeli az ST(0)-t és ST(1)-et.

FICOM *forrás* *(pl. FICOM WORD PTR [BP+6])*

ST(0) és a *forrás* összehasonlítása. (A *forrás* egész típusú)

FICOMP *forrás* *(pl. FICOM WORD PTR [BP+6])*

ST(0) és a *forrás* összehasonlítása. Veremről leemeli az ST(0)-t. (A *forrás* egész típusú)

Státuszbitek állása:

C3	***C2***	***C1***	***C0***	***Reláció***
0	*0*	*?*	*0*	*ST > forrás*
0	*0*	*?*	*1*	*ST < forrás*
1	*0*	*?*	*0*	*ST = forrás*
1	*1*	*?*	*1*	*ST nem összehasonlítható*

FCOS – Cosine (Koszinusz számítása)

FCOS

ST(0)-be ST(0) koszinusza

FDECSTP – Decrement Stack Pointer (Veremmutató csökkentése)

FDECSTP

A koprocesszor veremmutatóját eggyel csökkenti. Ha az 0, akkor 7-re állítja.

FDISI/FNDISI – Disable Interrupts (Megszakítások letiltása)

FDISI, FNDISI

Megakadályozza a koprocesszor megszakítást

FDIV/FDIVP/FIDV – Divide (Valós osztás)

FDIV

ST(1)-be ST(1) és ST(0) hányadosa. Veremről leemeli az ST(0)-t.

FDIV *ST, reg* *(pl. FDIV ST, ST(n))*

ST(0)-ba ST(0) és ST(n) hányadosa.

FDIV *reg, ST* *(pl. FDIV ST(n), ST)*

ST(n)-ba ST(n) és ST(0) hányadosa.

FDIV *forrás* *(pl. FDIV DWORD PTR [BX])*

ST(0) osztása a *forrás*-al, az eredmény ST(0)-ba (A *forrás* 32 és 64 bites adat lehet)

FDIVP *ST, reg* *(pl. FDIVP ST, ST(n))*

ST(0)-ba ST(0) és ST(n) hányadosa. Veremről leemeli az ST(0)-t.

FDIVP *reg, ST* *(pl. FDIVP ST(n), ST)*

ST(n)-ba ST(n) és ST(0) hányadosa. Veremről leemeli az ST(0)-t.

FIDIV *forrás* *(pl. FDIV Data[DI])*

ST(0) osztása a *forrás*-al, eredmény ST(0)-ba (A *forrás* egész típusú)

FDIVR/FDIVRP/FIDIVR – Divide Reserved (Valós osztás)

FDIVR

ST(1)-be ST(0) és ST(1) hányadosa. Veremről leemeli az ST(0)-t.

FDIVR *ST, reg* *(pl. FDIVR ST, ST(n))*

ST(0)-ba ST(n) és ST(0) hányadosa.

FDIVR *reg, ST* *(pl. FDIVR ST(n), ST)*

ST(n)-ba ST(0) és ST(n) hányadosa.

FDIVR *forrás* *(pl. FDIVR DWORD PTR [BX])*

A *forrás* osztása ST(0)-val, az eredmény ST(0)-ba (A *forrás* 32 és 64 bites adat lehet)

FDIVRP *ST, reg* *(pl. FDIVRP ST, ST(n))*

ST(0)-ba ST(n) és ST(0) hányadosa. Veremről leemeli az ST(0)-t.

FDIVRP *reg, ST* *(pl. FDIVRP ST(n), ST)*

ST(n)-ba ST(0) és ST(n) hányadosa. Veremről leemeli az ST(0)-t.

FIDIVR *forrás* *(pl. FDIVR Data[DI])*

A *forrás* osztása ST(0)-val, az eredmény ST(0)-ba (A *forrás* egész típusú)

FENI/FNENI - Enable Interrupts (Megszakítások engedélyezése)

FENI, FNENI

Az utasítás lehetővé teszi a megszakítások fogadását.

FFREE Free Register (Regiszter felszabadítása)

FFREE *reg* *(pl. FFREE ST(n))*

Törli az ST(n) regiszterhez tartozó toldalékbiteket. A regiszter tartalma nem változik.

FINCSTP – Increment Stack Pointer (Veremmutató növelése)

FINCSTP

A koprocesszor veremmutatóját eggyel növeli. Ha az 7, akkor 0-ra állítja.

FINIT/FNINIT – Initialize Coprocessor (Koprocesszor inicializálása)

FINIT, FNINIT

A processzor alaphelyzetbe állítása. A kontrolregisztert 3FFh-ra állítja, törli a kivételjelzőket és felszabadítja a lebegőpontos stack-et.

FLD/FILD/FBLD – Load (Betöltés a stack tetejére)

FLD *reg* *(pl. FLD ST(n))*

ST(n) regiszter tartalmának másolása a verem tetejére.

FLD *forrás*

A *forrás* másolása a verem tetejére. (A *forrás* valós)
A betöltés előtt konverziót végez.

FILD *forrás*

A *forrás* másolása a verem tetejére. (A *forrás* egész)
A betöltés előtt konverziót végez.

FBLD *forrás*

A *forrás* másolása a verem tetejére. (A *forrás* BCD)
A betöltés előtt konverziót végez.

FLD1/FLDZ/FLDPI/FLDL2E/FLDL2T/FLDLG2/FLDLN2 – Laod Constant (Konstans betöltése)

A verem tetejére betölti a következő konstansokat:

Utasítás	***Konstans***
FLD1	*+ 1.0*
FLDZ	*+ 0.0*
FLDPI	π
FLDL2E	$log_2(e)$
FLDL2T	$log_2(10)$
FLDLG2	$log_{10}(2)$
FLDLN2	$log_e(10)$

FLDCW – Load Control Word (Vezérlőszó betöltése)

FLDCW *forrás*

A 16 bites *forrás* betöltése a koprocesszor vezérlőszavába.

FLDENV/FLDENVW/FLDENVD – Load Environment State (Környezeti állapot betöltése)

FLDENV *forrás*, FLDENVW *forrás*, FLDENVD *forrás*

A környezeti változó tartalmazza a kontrolszót, a státuszszót, a toldalékregisztert, az utasítás- és az operandusmutatót.

387-es processzortól 28 bájtos, előtte 14 bájtos volt a környezeti változó.

FMUL/FMULP/FIMUL – Multiply (Szorzás)

FMUL

ST(1)-be ST(0) és ST(1) szorzata. Veremről leemeli az ST(0)-t.

FMUL *ST, reg* *(pl. FMUL ST, ST(n))*

ST(0)-ba ST(n) és ST(0) szorzata.

FMUL *reg, ST* *(pl FMUL ST(n), ST)*

ST(n)-ba ST(0) és ST(n) szorzata.

FMUL *forrás* *(pl FMUL DWORD PTR [BX])*

A *forrás* szorzása ST(0)-val, az eredmény ST(0)-ba. (A *forrás* 32 és 64 bites adat lehet)

FMULP *ST, reg* *(pl. FMULP ST, ST(n))*

ST(0)-ba ST(n) és ST(0) szorzata. Veremről leemeli az ST(0)-t.

FMULP *reg, ST* *(pl. FMULP ST(n), ST)*

ST(n)-ba ST(0) és ST(n) szorzata. Veremről leemeli az ST(0)-t.

FIMUL fo*rrás* *(pl. FIMUL Data[DI])*

A *forrás* szorzása ST(0)-val, az eredmény ST(0)-ba (A *forrás* egész típusú)

FNOP – No Operation (Üres utasítás)

FNOP

Az utasítás várakozásra használható.

FPATAN – Partial Arctangent (Arcus-tanges számítása)

FPATAN

Z = arctan(Y/X) értékének kiszámítása. X értéke ST(0)-ban, Y értéke ST(1)-ben van.
Az eredmény ST(1)-be kerül, majd a veremről leemeli az ST(0)-t.

FPREM – Partial Remainder (Maradékos osztás)

FPREM

ST(0) osztása ST(1)-el. A maradék ST(0)-ra kerül.

FPREM1 – IEEE Partial Remainder (Parciális maradékos osztás)

FPREM1

ST(0) osztása ST(1)-el. A maradék ST(0)-ra kerül.

FPTAN – Partial Tangent (Parciális tangens számítása)

FPTAN

Y/X = tan(Z) értékének kiszámítása. Z értéke ST(0)-ban, Y ST(0)-ba (Z helyére), majd X a stack tetejére ST(0)-ra kerül.

FRNDINT - Round to Integer (Kerekítés egészre)

FRNDINT

ST(0) értékét egészre kerekíti. A kontrolszó RC értékétől függően négyféle kerekítési mód lehetséges.

FRSTOR/FRSTORW/FRSTORD – Restore Saved State (Állapot adatok visszaállítása)

FRSTOR *forrás*, FRSTORW *forrás*, FRSTORD *forrás*

A *forrás*-sal megadott 94 bájtos (387-től 108 bájtos) memóriaterületről visszaállítja a koprocesszor állapotát.

FSAVE/FSAVEW/FSAVED/FNSAVE/FNSAVEW/FNSAVED – Save Status (A kooprocesszor állapotának mentése)

FSAVE cél, FSAVEW cél, FSAVED cél
FNSAVE cél, FNSAVEW cél, FNSAVED cél

A *cél*-lal megadott 94 bájtos (387-től 108 bájtos) memóriaterületre elmenti a koprocesszor állapotát. Ezután inicializálja a koprocesszort.

FSCALE – Scale (Karakterisztika módosítása)

FSCALE

$Y = Y * 2^X$ értékének kiszámítása. X értéke ST(1)-en, Y ST(0)-on található.

FSETPM – Set Protected Mode (Védett mód beállítása)

FSETPM

A koprocesszort védett módba helyezi.

FSIN – Sine (Szinusz számítás)

FSIN

ST(0) értékének szinusza, eredmény ST(0)-ba.

FSINCOS – Sine and Cosine (Szinusz és koszinusz számítása)

FSINCOS

ST(0) értékének szinuszát és koszinuszát kiszámítja, az ST(0)-ra először a szinusz, majd a koszinusz értékét teszi.

FSQRT – Square Root (Négyzetgyökvonás)

FSQRT

ST(0) értékének négyzetgyökét veszi, az eredményt ST(0)-ra teszi.

FST/FSTP/FIST/FISTP/FBSTP – Store (Adat tárolás)

FST *reg*

ST(0) tárolása a megadott *reg*-ben (ST(n)).

FSTP *reg*

ST(0) tárolása a megadott *reg*-ben (ST(n)). A veremről leemeli az ST(0)-t.

FST *cél* *(pl. FST longs[BX])*

ST(0) tárolása a megadott memóriahelyre. (A *cél* 32 vagy 64 bites hely lehet.)

FSTP *cél* *(pl. FST tempgreal[BX])*

ST(0) tárolása a megadott memóriahelyre. A veremről leemeli az ST(0)-t.
(A *cél* 32 és 64 bites hely lehet.)

FIST *cél* *(pl. FIST doubles[BX])*

ST(0) tárolása a megadott memóriahelyre. (A *cél* egész típusú.)

FISTP *cél* *(pl. FISTP doubles[BX])*

ST(0) tárolása a megadott memóriahelyre. A veremről leemeli az ST(0)-t.
(A *cél* egész típusú.)

FBSTP *cél* *(pl. FBST bcds[BX])*

ST(0) tárolása a megadott memóriahelyre. A veremről leemeli az ST(0)-t.
(A *cél* 80 bites (BCD) hely lehet.)

FSTCW/FNSTCW - Store Control Word (Vezérlőszó tárolása)

FSTCW *cél*, FNSTCW *cél*

A koprocesszor vezérlőszavának aktuális értékét tárolja a 16 bites *cél* helyen.

FSTENV/FSTENVW/FSTENVD/FNSTENV/FNSTENVW/FNSTENVD – Store Enviroment State (A környezet tárolása)

FSTENV *cél*, FSTENVW *cél*, FSTENVD *cél*
FNSTENV *cél*, FNSTENVW *cél*, FNSTENVD *cél*

A környezeti változó, az állapot- a vezérlő-, és a toldalékregiszter, valamint a kivételmutató mentése a megadott *cél* memóriahelyre.
(387-es processzortól 28 bájt, előtte 14 bájt.)

FSTSW/FNSTSW – Store Status Word (Állapotregiszter mentése)

FSTSW *cél*, FNSTSW *cél*

Az állapotszó aktuális értékét elmenti a *cél* helyre.

Processzor: 87, 287, 387

FSTSW AX, FNSTSW AX

Az állapotszó aktuális értékét elmenti az **AX** regiszterbe.

FSUB/FSUBP/FISUB – Subtract (Kivonás)

FSUB

ST(1)-be tölti ST(0) és ST(1) különbségét, majd a veremről leemeli az ST(0)-t.

FSUB ST, *reg* *(lp. FSUB ST, ST(n))*

ST(0)-ba tölti ST(0) és ST(n) különbségét.

FSUBP ST, *reg* *(lp. FSUBP ST, ST(n))*

ST(0)-ba tölti ST(0) és ST(n) különbségét, majd a veremről leemeli az ST(0)-t.

FSUB *reg*, ST *(lp. FSUB ST(n), ST)*

ST(n)-ba tölti ST(n) és ST(0) különbségét.

FSUBP *reg*, ST *(lp. FSUBP ST(n), ST)*

ST(n)-ba tölti ST(n) és ST(0) különbségét, majd a veremről leemeli az ST(0)-t.

FSUB *forrás* *(lp. FSUB shortreals[DI])*

ST(0)-ba tölti ST(0) és a *forrás* különbségét. (A *forrás* 32 vagy 64 bites valós lehet.)

FISUB *forrás* *(lp. FSUB warray[DI])*

ST(0)-ba tölti ST(0) és a *forrás* különbségét. (A *forrás* 16 vagy 32 bites egész lehet.)

FSUBR/FSUBRP/FISUBR – Subtract (Kivonás)

FSUBR

ST(1)-be tölti ST(1) és ST(0) különbségét, majd a veremről leemeli az ST(0)-t.

FSUBR ST, *reg* *(lp. FSUBR ST, ST(n))*

ST(0)-ba tölti ST(n) és ST(0) különbségét.

FSUBRP ST, *reg* *(lp. FSUBRP ST, ST(n))*

ST(0)-ba tölti ST(n) és ST(0) különbségét, majd a veremről leemeli az ST(0)-t.

FSUBR *reg*, ST *(lp. FSUBR ST(n), ST)*

ST(n)-ba tölti ST(0) és ST(n) különbségét.

FSUBRP *reg*, ST *(lp. FSUBRP ST(n), ST)*

ST(n)-ba tölti ST(0) és ST(n) különbségét, majd a veremről leemeli az ST(0)-t.

FSUBR *forrás* *(lp. FSUBR shortreals[DI])*

ST(0)-ba tölti a *forrás* és ST(0) különbségét. (A *forrás* 32 vagy 64 bites valós lehet.)

FISUBR *forrás* *(lp. FISUBR warray[DI])*

ST(0)-ba tölti a *forrás* és ST(0) különbségét. (A *forrás* 16 vagy 32 bites egész lehet.)

FTEST - Test for Zero (Nulla teszt)

FTEST

A TS(0) tartalmát összehasonlítja +0.0-val.)

Státuszbitek állása:

C3	***C2***	***C1***	***C0***	***Reláció***
0	*0*	*?*	*0*	*ST > 0.0*
0	*0*	*?*	*1*	*ST < 0.0*
1	*0*	*?*	*0*	*ST = 0.0*
1	*1*	*?*	*1*	*ST nem összehasonlítható*

FUCOM/FUCOMP/FUCOMPP – Unordered Compare (Rendezetlen összehasonlítás)

FUCOM

ST(0) és ST(1) összehasonlítása.

FUCOM *reg*

ST(0) és ST(n) összehasonlítása.

FUCOM *forrás*

ST(0) és *forrás* összehasonlítása. (A *forrás* 32 vagy 64 bites valós lehet.)

FUCOMP

ST(0) és ST(1) összehasonlítása, majd a veremről leemeli az ST(0)-t.

FUCOMP *reg*

ST(0) és ST(n) összehasonlítása, majd a veremről leemeli az ST(0)-t.

FUCOMP *forrás*

ST(0) és *forrás* összehasonlítása, majd a veremről leemeli az ST(0)-t.
(A *forrás* 32 vagy 64 bites valós lehet.)

FUCOMPP

ST(0) és ST(1) összehasonlítása, majd a veremről leemeli az ST(0)-t és ST(1)-et.

Státuszbitek állása:

C3	***C2***	***C1***	***C0***	***Reláció***
0	*0*	*?*	*0*	*ST > forrás*
0	*0*	*?*	*1*	*ST < forrás*
1	*0*	*?*	*0*	*ST = forrás*
1	*1*	*?*	*1*	*ST nem összehasonlítható*

FWAIT – Wait (CPU várakozás)

FWAIT

A parancs felfüggeszti a CPU működését mindaddig, amíg a koprocesszor befejezi az utasítás végrehajtását.

FXAM – Examine (Verem tetejének vizsgálata)

ST(0) állapotának vizsgálata.

Státuszbitek állása:

C3	*C2*	*C1*	*C0*	*Eredmény*
0	*0*	*0*	*0*	*+Nem szabályos*
0	*0*	*0*	*1*	*+NAN*
0	*0*	*1*	*0*	*-Nem szabályos*
0	*0*	*1*	*1*	*-NAN*
0	*1*	*0*	*0*	*+Normalizált*
0	*1*	*0*	*1*	*+∞*
0	*1*	*1*	*0*	*-Normalizált*
0	*1*	*1*	*1*	*-∞*
1	*0*	*0*	*0*	*+0*
1	*0*	*0*	*1*	*Üres*
1	*0*	*1*	*0*	*-0*
1	*0*	*1*	*1*	*Üres*
1	*1*	*0*	*0*	*+Nem normalizált*
1	*1*	*0*	*1*	*Üres*
1	*1*	*1*	*0*	*-Nem normalizált*
1	*1*	*1*	*1*	*Üres*

FXCH Exchange Registers (Verem elemeinek felcserélése)

FXCH

ST(0) és ST(1) tartalmának kicserélése.

FXCH *reg*

ST(0) és *reg* (ST(n)) tartalmának kicserélése.

FXTRACT – Extract Exponent and Significand (Kitevő és mantissza kiemelése)

FXTRACT

ST(0)-ban tárolt adat kitevőjét ST(0)-ba teszi, majd a verem tetejére tölti a mantisszát

FYL2X- Ylog2(X) (Ylog2(X) értékének kiszámítása)

FYL2X

Z = $Y\log_2(X)$ értékének kiszámítása. X értéke ST(0)-on, Y értéke ST(1)-en található. Leemeli a verem tetején lévő értéket, majd az ST(0)-ra helyezi Z-t.

FYL2X- Ylog2(X+1) (Ylog2(X+1) értékének kiszámítása)

FYL2X

Z = $Y\log_2(X+1)$ értékének kiszámítása. X értéke ST(0)-on, Y értéke ST(1)-en található. Leemeli a verem tetején lévő értéket, majd az ST(0)-ra helyezi Z-t.

12. Megszakítások

12.1. INT 10h funkció: képernyő driver hívása

AH = 00h Képernyőüzemmód beállítása.

Szöveg üzemmódban:

AL=0 40 x 25, mono
AL=1 40 x 25, színes
AL=2 80 x 25, mono
AL=3 80 x 25, színes
AL=7 80 x 25, monokróm képernyő adapter

Grafikus üzzemmód

AL=4 320-szor 200, színes
AL=5 320-szor 200, mono
AL=6 640-szer 200, mono

AH = 01h Kurzorméret beállítása.

A kurzort színes adapter esetén 8 (mono adapternél 14) pontsorból álló téglalap definiálja. Értéke 0-7 illetve 0-13 közötti lehet.

CH a kurzor kezdő sora
CL a kurzor utolsó sora

Ha a kezdősor száma kisebb, mint az utolsóé, a kurzor lyukas lesz. Ha a kezdősor száma nagyobb a maximumnál, akkor a kurzor nem látszik.

AH = 02h Kurzorpozíció beállítása.

DH kurzorpozíció sor koordinátája (0-24)
DL kurzorpozíció oszlop koordinátája (0-79)
BH oldalszám

AH = 03h Kurzorpozíció olvasása.

BH oldalszám

Válasz:

DH kurzor sorszáma
DL kurzor oszlopszáma
CH, CL kurzor mérete. (ld. Kurzorméret-beállításnál)

AH = 04h Fényceruzapozíció olvasása.

Válasz:

AH bekapcsolás jelzése
Ha AH = 1, a fényceruza be van kapcsolva, és az alábbi regiszterek korrekt adatokat tartalmaznak.
DH, DL a mutatott sor és oszlop száma (karakteres üzemmódban)
CH, BX a mutatott sor és oszlop száma (grafikus üzemmódban)

AH = 05h Aktív képernyőoldal kiválasztása.

AL oldalszám (0 – 7, alfanumerikus üzemmódban.)

AH = 06h ***Sorgörgetés felfelé ablakban.***

AL léptetések száma (0 esetén törli az ablakot.)
CH, CL ablak bal felső sarkának sor, oszlop koordinátái
DH, DL ablak jobb alsó sarkának koordinátái
BH törölt sorok attribútuma
(Az attribútum használatát és típusait a programozás fejezetben részletesen ismertettük.)

AH = 07h ***Sorgörgetés lefelé ablakban.***

Hasonlóan működik, mint a felfelé görgetés.

AH = 08h ***Kurzor alatti karakter és attribútumának olvasása.***

BH képernyőoldal (szöveges üzemmódban)

Válasz:

AL olvasott karakter
AH karakter attribútuma (szöveges üzemmódban)

AH = 09h ***Karakter kiírása az aktuális kurzorpozícióba.***

AL karakter kód
BL karakter attribútum
BH képernyőoldal (szöveges üzemmódban)
CX ismétlések száma

AH = 0Ah ***Karakter kiírása az aktuális kurzorpozícióba (attributum nélkül).***

AL karakter kód
BH képernyőoldal (szöveges üzemmódban)
CX ismétlések száma

AH = 0Bh ***Színpaletta vagy háttérszín beállítása***

BH 0 - háttérszín megadás
1 – palettaszín megadás
BL színkód

AH = 0Ch ***Raszterpont megjelenítése.***

AL színkód
DL raszter sorszám (y koordináta)
CX raszter oszlopszám (x koordináta)

AH = 0Dh ***Raszterpont lekérdezés.***

DL raszter sorszám (y koordináta)
CX raszter oszlopszám (x koordináta)

Válasz:

AL színkód

AH = 0Eh ***Karakter kiírás, majd a kurzor léptetése.***

AL karakterkód
BL karakter színe (grafikus üzemmódban)
BH oldalszám (szöveges üzemmódban)

AH = 0Fh ***Aktuális üzemmód lekérdezése.***

Válasz:

AL aktuális képernyő üzemmód
AH karakterek száma egy sorban
BH aktív képernyőoldalak száma

12.2. INT 16h funkció: klaviatúra driver hívása

AH = 00h ***Billentyűzet olvasása.***

Válasz:

AL billentyű ASCII kódja (funkció billentyűknél 0)
AH funkció billentyű letapogató kódja (scan-kód)

AH = 01h ***Billentyűzet pufferének lekérdezése.***

Válasz:

ZF 1 – a puffer üres
0 – a pufferben van karakter
AL ASCII kód
AH letapogató kód

AH = 02h ***Billentyűzet státuszának lekérdezése.***

Válasz:

AL státusz billentyűk állapota (a megfelelő helyi értékű bit értéke 1, ha az adott billentyű aktív)
0 – jobb Shift lenyomva
1 – bal Shift lenyomva
2 – Control lenyomva
3 – Alt lenyomva
4 – Scroll Lock be
5 – Num Lock be
6 – Caps Lock be
7 – Insert be

12.3. INT 21h funkció: DOS funkciók hívása

AH = 01h Olvasás a standard inputról (billentyűbeolvasás és echo).

Válasz:

AL a lenyomott billentyű ASCII kódja

Funkcióbillentyű esetén, első olvasásra **AL**-ben 0 értéket kapunk, majd a második olvasáskor kapjuk meg scan-kódot.

AH = 02h Kiírás a standard outputra (karakterkiírás a képernyőre).

DL ASCII kód (nemcsak megjeleníthető karakterek)

A következő hat funkcióhívás paraméterei megegyeznek az előző kettőével.

AH = 03h Olvasás a standard kiegészítő inputról.

AH = 04h Kiírás a standard kiegészítő outputra.

AH = 05h Kiírás a standard nyomtatóra.

AH = 06h Olvasás vagy kiírás a standard inputról vagy outputra.

AH = 07h Olvasás a standard inputról, nincs CTRL-BREAK ellenőrzés.

AH = 08h Olvasás a standard inputról (azonos a 01h funkcióval, echo nélkül).

AH = 09h Karakterlánc kiírása a standard outputra.

DS:DX karakterlánc kezdetének memória címe (mutató)
A sztring a $ karakterig tart.

AH = 0Ah Karakterlánc olvasása a standard inputról.

Válasz:

DS:DX puffer kezdetének memória címe (mutató)
A puffer 1. bájtja a puffer hosszát (az ENTER-rel együtt), a 2. pedig a ténylegesen beolvasott karakterek számát tartalmazza. A sztring tényleges beolvasása az ENTER (0Dh) karakter leütése után történik meg.

AH = 0Bh Standard input (billentyű) állapotának lekérdezése.

AL állapotkód (FF-et tartalmaz, van karakter az eszköz pufferében, 0-át, ha nincs, vagy fájlvég.)

AH = 0Ch Klaviatúra pufferének törlése és funkcióhívás.

Törli a billentyűzet pufferét, majd meghívja az **AL**-ben megadott (01h, 06h, 07h, 08h, 0Ah) funkciókódú interruptot.

AH = 20h Kilépés a DOS-ba.

Visszatérés az operációs rendszerhez csak COM fájloknál.

Válasz:

AL visszatérési kód

AH = 25h Megszakításvektor beállítása.

AL megszakítás száma
DS:DX az új megszakításkezelő címe.

AH = 35h Megszakításvektor olvasása.

AL megszakítás száma

Válasz:
ES:BX a megszakításkezelő címe

AH = 2Ah Rendszerdátum lekérdezése

Válasz:
CX év (1980-2099)
DH hónap
DL nap
AL a hét napja (sorszám)

AH = 2Bh Rendszerdátum beállítása

CX év (1980-2099)
DH hónap
DL nap

Válasz:
AL=0 sikeres beállítás
AL=0FFh hibás dátum, a rendszerdátum nem változott

AH = 2Ch Rendszeridő lekérdezése

Válasz:
CH óra
CL perc
DH másodperc
DL századmásodperc

AH = 2Dh Rendszeridő beállítása

CH óra
CL perc
DH másodperc
DL századmásodperc

Válasz:
AL=0 sikeres beállítás
AL=0FFh hibás idő, a rendszeridő nem változott

AH = 39h Könyvtár létrehozása (MKDIR)

DS:DX a könyvtár (mappa) neve karakterlánc kezdetének memória címe (teljes elérési út). A karakterlánc végét 00h jelzi!

Válasz:

CF=0 sikeres létrehozás

CF=1 sikertelen létrehozás, AX-ben a hibakód

Megjegyzés:

- Az elérési útvonalnak (az utolsó könyvtár kivételével) léteznie kell.
- Sikertelen a megnyitás, ha már létezik a megadott mappa, ha megtelt a szülő könyvtár, vagy az útvonal név hosszabb az adott operációs rendszerben megengedettnél.

AH = 3Ah Könyvtár törlése (RMDIR)

DS:DX a könyvtár (mappa) neve karakterlánc kezdetének memória címe (teljes elérési út). A karakterlánc végét 0 jelzi!

Válasz:

CF=0 sikeres törlés

CF=1 sikertelen törlés, AX-ben a hibakód

Megjegyzés:

- A könyvtárnak üresnek kell lennie.

AH = 3Bh Az aktuális könyvtár beállítása (CHDIR)

DS:DX a könyvtár (mappa) neve karakterlánc kezdetének memória címe (teljes elérési út). A karakterlánc végét 0 jelzi!

Válasz:

CF=0 sikeres átállítás

CF=1 sikertelen átállítás, AX-ben a hibakód

AH = 3Dh Fájl megnyitás

DS:DX a fájl neve karakterlánc kezdetének memória címe (teljes elérési út). A karakterlánc végét 0 jelzi!

AL mód, a bitek jelentése:

0-2	Access mód
	000: csak olvasás (Read-only)
	001: csak írás (Write-only)
	010: írás-olvasás (Read-Write)
3	foglalt
4-6	Sharing mód
7	Öröklésjelző

Válasz:

CF=0 sikeres átállítás, AX-ben a fájlkezelő kód

CF=1 sikertelen átállítás, AX-ben a hibakód

AH = 3Eh Fájl lezárás

BX a fájlkezelő kódja

Válasz:

CF=0 sikeres átállítás

CF=1 sikertelen átállítás, AX-ben a hibakód

AH = 3Fh ***Olvasás fájlból***

BX a fájlkezelő kódja
CX a beolvasott bájtok száma
DS:DX a cél adatterület kezdőcíme

Válasz:
CF=0 sikeres átállítás, AX-ben a ténylegesen beolvasott bájtok száma
CF=1 sikertelen átállítás, AX-ben a hibakód

AH = 40h ***Írás fájlba***

BX a fájlkezelő kódja
CX a beolvasott bájtok száma
DS:DX a forrás adatterület kezdőcíme

Válasz:
CF=0 sikeres átállítás, AX-ben a ténylegesen beolvasott bájtok száma
CF=1 sikertelen átállítás, AX-ben a hibakód

AH = 41h ***Fájl törlés***

DS:DX a fájl neve karakterlánc kezdetének memória címe (teljes elérési út). A karakterlánc végét 0 jelzi!

Válasz:
CF=0 sikeres törlés
CF=1 sikertelen törlés, AX-ben a hibakód

AH = 4Ch ***Kilépés*** *(Visszatérés az operációs rendszerbe COM és EXE fájloknál.)*

AL visszatérési kód

AH = 4Eh ***Fájl első előfordulásának keresése***

DS:DX a fájl neve karakterlánc kezdetének memória címe (teljes elérési út). A karakterlánc végét 0 jelzi!
CX keresési attribútum, a bitek jelentése:
0 Read-Only,
1 Hidden,
2 System,
3 Volume Label,
4 Directory,
5 Archive,
6-15 foglalt.

Válasz:
CF=0 találat
CF=1 nincs találat, AX-ben a hibakód

12.4. INT 25h funkció: lemezszektor olvasása

AL meghajtó száma (0=A, 1=B, 2=C, stb.)

CX olvasandó szektorok száma

DX kezdő szektor száma (az első szektor 0)

DS:BX cél (memória) kezdőcíme.

Válasz:

AX hibakód

12.5. INT 26h funkció: lemezszektor írása

AL meghajtó száma

CX írandó szektorok száma

DX kezdő szektor száma

DS:BX forrás (memória) kezdő címe

Válasz:

AX hibakód

12.6. INT 33h funkció: egér driver hívása

AX = 00h Inicializálás

Válasz:

AX 0 esetén nem inicializálható, egyébként megtörtént.

AX = 01h Egérkurzor láthatóvá tétele

Ha a kurzor nem látható, láthatóvá teszi, egyébként hatástalan.

AX = 02h Egérkurzor láthatatlanná tétele

Ha a kurzor látható, láthatatlanná teszi, egyébként hatástalan.

AX = 03h Egérkurzor koordinátáinak lekérdezése

Válasz:

BX A hívás pillanatában lenyomott egérgomb kódja
0 – bal gomb
1 – jobb gomb
2 – középső gomb

CX Oszlop koordináta

DX Sor koordináta

AX = 04h Egérkurzor mozgatása adott pontba

CX Oszlop koordináta

DX Sor koordináta

AX = 05h Gombok lenyomásának száma

BX Melyik gombról kérünk információt

Válasz:

AX A ténylegesen lenyomott gomb

BX A gomb lenyomásának száma (ld. 03h funkciót)

CX Oszlop koordináta

DX Sor koordináta

AX = 06h Gombok felengedéseinek száma

BX Melyik gombról kérünk információt

Válasz:

AX A lenyomott gomb (ld. 03h funkciót)

BX A gomb felengedéseinek száma

CX Oszlop koordináta

DX Sor koordináta

A következő két funkcióval egy téglalap alakú területet jelölhetünk ki, amelyiken belül mozoghat az egérkurzor.

AX = 07h ***Egér vízszintes mozgásterületének megadása***

CX A terület első oszlopának sorszáma
DX A terület utolsó oszlopának sorszáma

AX = 08h ***Egér függőleges mozgásterületének megadása***

CX A terület felső sorának sorszáma
DX A terület alsó sorának sorszáma

AX = 09h ***Grafikus egérkurzor definiálása***

BX A kurzor X irányú bázispontja
CX A kurzor Y irányú bázispontja
DX A kurzor bitképének memóriahelye

AX = 0Ah ***Szöveges egérkurzor definiálása***

BX 0: szoftveres, 1: hardveres definíció
CX, DX A kurzor első és utolsó sora hardveres definíciónál
CX, DX A kurzor bitképének címe, szoftveres definíciónál

AX = 0Bh ***Információ az egér mozgásáról***

Válasz:
CX Vízszintes elmozdulás (balra: negatív, jobbra pozitív)
DX Függőleges elmozdulás (felfelé: negatív, lefelé: pozitív)

AX = 0Dh ***Fényceruza emuláció bekapcsolása***

AX = 0Eh ***Fényceruza emuláció kikapcsolása***

AX = 0Fh ***Lépésköz állítása***

A mozgás finomságát szabályozhatjuk.
CX Vízszintes lépésköz
DX Függőleges lépésköz

AX = 10h ***Letiltott tartomány megadása.***

Az egér számára tiltott téglalap alakú terület megadása.
CX, DX Bal felső sarok
SI, DI Jobb alsó sarok

AX = 13h ***Maximális sebesség beállítása***

DX Lépésköz/másodperc

AX = 1Ah ***Érzékenység beállítása.***

BX Vízszintes Mickey-szám[16]
CX Függőleges Mickey-szám
DX Sebesség Mickey/másodperc

AX = 1Bh ***Érzékenység lekérdezése***

Válasz:
BX Vízszintes Mickey-szám
CX Függőleges Mickey-szám
DX Sebesség Mickey/másodperc

AX = 1Ch ***Megszakítási arány magadása***

Az megszakítások számának megadása, az egérpozíció lekérdezésének gyakorisága
BX Megszakítások száma:
0 – nincs megszakítás
1 – 30 megszakítás/másodperc
2 – 50 megszakítás/másodperc
3 – 100 megszakítás/másodperc
4 – 200 megszakítás/másodperc

AX = 1Dh ***Képernyő kiválasztása***

BX Képernyő lapszáma, ezen jelenik meg a kurzor

AX = 1Eh ***Képernyő lekérdezése***

Válasz:
BX Képernyő lapszáma

AX = 24h ***Egér-driver lekérdezése***

Válasz:
BX Driver verziószáma (BH.BL)
CH Egér típusa
1 – busz egér
2 – soros egér
3 – InPort egér
4 – PS-2 egér
5 – HP egér
CL IRQ szám

[16] Egy Mickey az egér egységnyi elmozdulása (az e legkisebb elmozdulás, amit az egér érzékel).

13. A 7 bites ASCII kódtábla

13.1. Alapkódok

dec.	hexa	kar.	dec.	hexa	kar.	dec.	hexa	kar.	dec.	hexa	kar.
0	0		32	20	**space**	64	40	**@**	96	60	**`**
1	1		33	21	**!**	65	41	**A**	97	61	**a**
2	2		34	22	**"**	66	42	**B**	98	62	**b**
3	3		35	23	**#**	67	43	**C**	99	63	**c**
4	4		36	24	**$**	68	44	**D**	100	64	**d**
5	5		37	25	**%**	69	45	**E**	101	65	**e**
6	6		38	26	**&**	70	46	**F**	102	66	**f**
7	7	**bel**	39	27	**'**	71	47	**G**	103	67	**g**
8	8	**BS**	40	28	**(**	72	48	**H**	104	68	**h**
9	9	**HT**	41	29	**)**	73	49	**I**	105	69	**i**
10	a	**LF**	42	2a	*****	74	4A	**J**	106	6a	**j**
11	b	**VT**	43	2b	+	75	4B	**K**	107	6b	**k**
12	c	**FF**	44	2c	**,**	76	4C	**L**	108	6c	**l**
13	d	**CR**	45	2d	**-**	77	4D	**M**	109	6d	**m**
14	e		46	2e	**.**	78	4E	**N**	110	6e	**n**
15	f		47	2f	/	79	4F	**O**	111	6f	**o**
16	10		48	30	**0**	80	50	**P**	112	70	**p**
17	11		49	31	**1**	81	51	**Q**	113	71	**q**
18	12		50	32	**2**	82	52	**R**	114	72	**r**
19	13		51	33	**3**	83	53	**S**	115	73	**s**
20	14		52	34	**4**	84	54	**T**	116	74	**t**
21	15		53	35	**5**	85	55	**U**	117	75	**u**
22	16		54	36	**6**	86	56	**V**	118	76	**v**
23	17		55	37	**7**	87	57	**W**	119	77	**w**
24	18		56	38	**8**	88	58	**X**	120	78	**x**
25	19		57	39	**9**	89	59	**Y**	121	79	**y**
26	1a		58	3a	**:**	90	5A	**Z**	122	7a	**z**
27	1b	**ESC**	59	3b	**;**	91	5B	**[**	123	7b	**{**
28	1c		60	3c	<	92	5C	\	124	7c	**|**
29	1d		61	3d	=	93	5D	**]**	125	7d	**}**
30	1e		62	3e	>	94	5E	^	126	7e	**~**
31	1f		63	3f	**?**	95	5F	**_**	127	7f	**DEL**

13.2. Billentyűkódok

kar.	kód	shift	ctrl	alt	alt gr	kar.	kód	shift	ctrl	alt	alt gr	kar.	kód	shift	ctrl	alt	alt gr
f1	0;3b	0;54	0;5e	0;68	0;68	**cr**	0d	0d	0a	-	0;1c	**a**	61	41	1	0;1e	0;3b
f2	0;3c	0;55	0;5f	0;69	0;69	**esc**	1b	1b	-	-	-	**b**	62	42	2	0;30	0;7b
f3	0;3d	0;56	0;60	0;6a	0;6a	**sp**	20	20	20	-	20	**c**	63	43	3	0;2e	0;26
f4	0;3e	0;57	0;61	0;6b	0;6b	**'**	27	22				**d**	64	44	4	0;20	0;d1
f5	0;3f	0;58	0;62	0;6c	0;6c	**,**	2c	3c				**e**	65	45	5	0;12	0;f6
f6	0;40	0;59	0;63	0;6d	0;6d	**-**	2d	5f				**f**	66	46	6	0;21	0;5b
f7	0;41	0;5a	0;64	0;6e	0;6e	**.**	2e	3e				**g**	67	47	7	0;22	0;7d
f8	0;42	0;5b	0;65	0;6f	0;6f	**/**	2f	3f				**h**	68	48	8	0;23	0;26
f9	0;43	0;5c	0;66	0;70	0;70	**0**	30	29	ff	0;29	0;29	**i**	69	49	9	0;17	0;d6
f10	0;44	0;5d	0;67	0;71	0;71	**1**	31	21	ff	0;78	0;7e	**j**	6a	4a	a	0;24	0;a1
f11	0;85	0;87	0;89	0;8b	0;8b	**2**	32	40	0	0;79	-	**k**	6b	4b	b	0;25	0;88
f12	0;86	0;88	0;8a	0;8c	0;8c	**3**	33	23	ff	0;7a	-	**l**	6c	4c	c	0;26	0;9d
↑	0;48	0;48	0;8d	0;98	0;98	**4**	34	24	ff	0;7b	-	**m**	6d	4d	d	0;32	0;3c
↓	0;50	0;50	0;91	0;a0	0;a0	**5**	35	25	ff	0;7c	-	**n**	6e	4e	e	0;31	0;7d
→	0;4d	0;4d	0;74	0;9d	0;9d	**6**	36	5e	1e	0;7d	-	**o**	6f	4f	f	0;18	0;3e
←	0;4b	0;4b	0;73	0;9b	0;9b	**7**	37	26	ff	0;7e	0;60	**p**	70	50	10	0;19	0;2a
ins	0;52	0;52	0;92	0;a2	0;a2	**8**	38	2a	ff	0;7f	-	**q**	71	51	11	0;10	0;56
del	0;53	0;53	0;93	0;a3	0;a3	**9**	39	28	ff	0;80	-	**r**	72	52	12	0;13	0;9e
home	0;47	0;47	0;77	0;97	0;97	**;**	3b	3a				**s**	73	53	13	0;1f	0;d0
end	0;4f	0;4f	0;75	0;9f	0;9f	**=**	3d	2b				**t**	74	54	14	0;14	0;24
p up	0;49	0;49	0;84	0;99	0;99	**@**	40					**u**	75	55	15	0;16	0;cf
P dn	0;51	0;51	0;76	0;a1	0;a1	**[**	5b	7b	18	0;17		**v**	76	56	16	0;2f	0;40
bs	8	8	7f	0;0e	0;0e	\	5c	7c	19	0;2b		**w**	77	57	17	0;11	0;7c
ht	9	0;0f	0;94	0;a5	0;a5	**]**	5d	7d	20	0;18		**x**	78	58	18	0;2d	0;23
						`	60	7e	-	0;29		**y**	79	59	19	0;3e	0;3e
												z	7a	5a	1a	0;e1	0;e1

Figyeljük meg, hogy betűk esetében a módosító billentyűk használatával egy bitet törlünk a karakter kódjából. A shift billentyű az 5-ös, a ctrl billentyű pedig a 6-os számú bitet törli.

A táblázatból is látható, hogy több mint 256 (1 bájton ábrázolható) billentyűkombináció létezik, ezért úgynevezett kettős billentyűkódokat használunk (elsősorban a funkcióbillentyűknél). Ilyenkor a karakterkód mindig 0, a második érték a billentyű letapogató (scan) kódja.