IBM PC/XT

felhasználóknak és programozóknak

1

Pethő Ádám

Assembly alapismeretek

SZÁMALK

PROFESSZIONÁLIS SZEMÉLYI SZÁMÍTÓGÉPEK

Ádám Pethő

IBM PC/XT

FOR USERS AND PROGRAMMERS

1.

ASSEMBLY LANGUAGE FUNDAMENTALS

Second edition

Computing Applications and Service Company
Budapest, 1987

Pethő Ádám

IBM PC/XT

FELHASZNÁLÓKNAK ÉS PROGRAMOZÓKNAK

1.

ASSEMBLY ALAPISMERETEK

Második kiadás

Számítástechnika-alkalmazási Vállalat
Budapest, 1987

Lektorálta: **Herneczki István**
Supervised by **István Herneczki**

ISBN 963 553 123 0

Kiadja a Számítástechnika-alkalmazási Vállalat
Felelős kiadó: Havass Miklós vezérigazgató
Felelős szerkesztő: Lukács Erzsébet
Műszaki vezető: Molnár Zoltán
Műszaki szerkesztő: Beleznai László
A fedelet tervezte: Molnár Zoltán
Megjelent: 21 (A/5) ív terjedelemben

Készült az Astra Kisszövetkezet sokszorosító üzemében
Budapest, 1987. Felelős vezető: Pokorny Vilmos
A nyomdai megrendelés száma: 87230

A szerző előszava

Ez a könyv három kötetében az IBM PC/XT programozásához és felhasználásához szükséges alapvető ismereteket tartalmaz. A könyvek fő célja az, hogy segítséget nyújtsak az Olvasónak a gép és legelterjedtebb operációs rendszerének (MS-DOS) megismeréséhez, olyan "alacsony" szinten, ameddig egy programozó értelmesen "leereszkedhet". Ezért választottam a könyvek alapnyelveként a leghatékonyabb és (talán) legnehezebb assembly nyelvet.

Az első kötet az alapismereteket tartalmazza: az assembly programozáshoz szükséges processzorismeretet, a fordító szolgáltatásait, a fordítás, a futtatható program létrehozásának menetét. Az itt megszerzett ismeretek birtokában foghatunk majd a második és harmadik kötet olvasásának.

A második kötetben a programunk és az MS-DOS operációs rendszer közötti kapcsolatokat tárgyaljuk. Mit kap a program útravalóként, mikor elindítjuk? Milyen támogatásra számíthat az operációs rendszertől futás közben? Hogyan lép ki, mit hagyhat maga után a rendszerben, a memóriában? Hogyan használhatja ki a gép egyes perifériáit? Ezekre a kérdésekre kaphat választ az Olvasó a második kötetben. Jóllehet minden esetben assembly nyelvű példaprogramokat fogok megadni, mondanom sem kell talán, hogy mindezeket a szolgáltatásokat számos magasszintű nyelven megírt programban is igénybe vehetjük. Amennyire ez lehetséges, minden fontos funkció ismertetése mellett működő példaprogramok szerepelnek majd illusztrációként. Itt kerül sor az Intel 8087 koprocesszor vázlatos ismertetésére is.

A harmadik kötetben programozási alapfeladatokat, és fontos programozástechnikai kérdéseket (például a paraméter-átadás kérdéseit) fogunk tárgyalni. A kötet első felében egyszerű példaprogramokat találunk. Ilyenek a memóriakezelés, a különféle konverziók, a dupla pontosságú aritmetika és számos egyéb apróság. A kötet második felében nagyobb feladatok és (esetenként nem is teljes) megoldásaik találhatók. Ennek során figyelmünket nem csak a szigorúan vett programozás nehézségeire összpontosítjuk majd. Legalább ilyen fontos a komoly munkához elengedhetetlen "környezet": az operációs rendszer és egyes idevágó szolgáltatásai, a szövegszerkesztő, a fordító és a programszerkesztő alaposabb megismerése, "profi" felhasználása; a programtervezés, feljesztés, dokumentálás legfontosabb

szempontjai. Komoly munkához utasitásszintig menő tanácsokat adni úgysem lehet. Fontos azonban, hogy a precíz munkát lehetővé tevő, sőt, azt kikényszerítő eszközökkel, és a lehető legjobb felfogásban dolgozzunk. Az ilyen szempontok megvilágitását tekintem a harmadik kötet legfontosabb céljának.

Hogyan használjuk fel ezeket a könyveket?

Semmiképpen ne higyjük azt, hogy (bármilyen) könyvből meg lehet tanulni programozni. A szerzők minden igyekezete kevés e hihetetlenül bonyolult folyamat elsajátitásához. A programozás tanulása közben legfontosabb talán a gyakorlat. Eleinte minél többet kell gép mellett ülni, kisebb, de életszagú problémákat megoldani, esetleg egyszerűbb játékprogramokat irni, mert ezek, felkeltve becsvágyát, igen jól doppingolják a programozót. A kezdeti fázisban bátran kövessünk el minél több hibát, addig nem nyugodva, míg rá nem találunk a hiba igazi okára.

Ha már elég jól elsajátitottuk az alapismereteket és tudunk mindent ahhoz, hogy papíron is jól tudjunk programozni, keressünk igazi feladatokat. Ennek során igyekezzünk minél jobb programozókkal együtt dolgozni, mert nagyon sokat tanulhatunk tőlük. Ebben a fázisban érdemes inkább már elszakadni a géptől és a munka lehető legnagyobb részét a tervezés során elvégezni. Az ilyen felkészültségű programozó számára már nagyon fontos a szakirodalom olvasása, és a kollégákkal való konzultáció a lehető legszélesebb körben.

Mint az első bekezdésekből kitűnik, az Olvasók e könyveket inkább az első, mint a második fázis során használhatják. Megirásuk közben az a szempont vezetett, hogy haszonnal forgathassák olyan programozók és a programozás iránt érdeklődők, akik rendelkeznek programozási alapismeretekkel és (esetleg) bizonyos gyakorlattal magasszintű programozási nyelveken. Nem tételeztem fel assembly előismereteket (természetesen nagyon hasznos, ha az Olvasó más gépeken szerzett már bizonyos gyakorlatot e téren). Amennyiben kedves Olvasóm nem rendelkezik a szükséges alapismeretekkel, fusson át bármilyen bevezető jellegű munkát. Ilyenek most már széles választékban találhatók magyar nyelven is.

Az alapismereteket feltételezve nem részleteztem olyan számitástechnikai alapfogalmakat, mint bit, byte, utasitás, file stb.; nem indoklom meg, miért irtam a gépi címeket hexadecimálisan - némileg gyakorlott Olvasó ezzel amúgy is tisztában van. Nem törekedtem teljesen precíz tárgyalásmódra sem; tudomásul kell vennünk, hogy minden számitástechnikai

terület egy teljes világ, amelyet egyszerre kell áttekinteni. Mivel az ember erre kezdetben képtelen, el kell fogadnia, hogy számos részlet van, amit még nem ért. Ha ez nagyon zavar bennünket, lapozzunk előre a tartalomjegyzék alapján, és nézzük meg azt, ami nem világos.

A felhasznált irodalom a következő:

A processzort ismertető részek alapja:

Russel Rector és George Alexy

The Intel 8086

című könyve, valamint természetesen az Intel cég hivatalos leírása:

The 8086 and 8088 Central Processing Units

A fordítóprogram használatát és szolgáltatásait ismertető fejezetek a Microsoft

Macroassembler Reference Manual -ja

felhasználásával készültek.

A második könyv az MS-DOS, a ROM BIOS, és a gép programozható elemeinek használatával kapcsolatos ismereteket tartalmazza; ezek a részek a Microsoft

DOS Technical Reference Manual -ja

alapján készültek. Minthogy azonban (bármilyen alaposak legyenek is ezek a kézikönyvek) igen sok kiegészítő információ szükségeltetik az IBM PC/XT hatékony programozásához, melegen ajánlom az angol nyelvben járatos Olvasónak, hogy keressen további könyveket munkája során. Sajnos, rengeteg olyan apróság van, amit csak tartós utánjárással és esetleg csak

szájhagyomány útján, gyakorlott, a rendszert és a gépet alaposan ismerő szakemberek tanácsaiból lehet megtudni, hiszen minden kis részlet még egy ilyen, viszonylag egyszerű gép és operációs rendszer leírásában sem szerepelhet.

Ezért szeretném a következő művekre felhívni az Olvasó figyelmét. Forgassuk bizalommal és szorgalmasan

Adam Osborne:

ıne 16-bit Microprocessor Handbook

David J. Bradley:

Assembly Language Programming for the IBM PC

és

Peter Norton:

Programmer's Guide to the IBM PC

című munkáit, melyeket számos esetben használtam fel e könyvek megírása során is.
További hasznos információk meríthetők többek között

L. J. Scanlon

IBM PC & XT Assembly Language
A Guide for Programmers

című könyvéből.

A processzor utasításkészlete magyar nyelven is elérhető az "LSI Alkalmazás-Technikai Szolgálat" gondozásában:

i8086 utasításkészlet

címen.

Tartalom

A könyv felépítése

A fejezetek további alfejezetekre oszlanak. Ezek között az első mindig egy - az adott fejezethez tartozó - bevezetés. Ez kapta a nulladik sorszámot. Az egyes fejezetek külön tartalomjegyzéke a fejezet elején olvasható.

A lap tetején látható fejléc szöveges része a páratlan oldalakon a kötet, a páros oldalakon pedig a fejezet címe. Itt kapott helyet az oldalszám is, amely két részből áll, kötőjellel elválasztva. Az első rész a fejezet, a második pedig az oldal száma.

Terminológiai megjegyzések

A könyvben igyekeztem mindenütt olyan magyar kifejezéseket használni, amelyek a lehető legjobban fedik az adott dolog jelentését. Ezek között számos már polgárjogot nyert, széles körben alkamazott magyar szó, jónéhány viszont saját "termés". Bizonyos esetekben (pl.: byte) megmaradtam a magyarban már megszokott angol kifejezésnél. Azt hiszem, hogy néhány társával együtt ez a szó már annyira gyökeret vert a magyar szaknyelvben, hogy más kifejezés használata zavaró lenne.

Az egyes kifejezések első előfordulásánál megadjom a hivatalos (tehát Intel) terminológia eredeti angol kifejezéseit is. Ez az Olvasó hasznára válhat az eredeti angol irodalom megértésében, sikeres alkalmazásában.

Mivel nagyon szubjektív dolog annak eldöntése, hogy mely kifejezések fogadhatók el eredeti nyelven és melyeket kell lefordítani, az esetleges következetlenségekért előre is elnézést kérek az Olvasótól. Számos esetben, bár van elterjedt és kifejező magyar szó ragaszkodnom kellett az eredeti angol szóhoz, mert a magyarra fordítás kifejezetten félrevezető lenne. Például elterjedt az "átvitelbit" a Carry megnevezésére. De kénytelen vagyok az eredetinél maradni, mert a Carry szóra utaló "C" betű számos utasításkódban is szerepel. Itt tehát bizonyos "következetlenség" nem kerülhető el.

IBM PC/XT - assembly alapismeretek

Bevezetés

Az assembly programozás a számítógépek programozásának "legősibb", leghatékonyabb, de - programozástechnikai szempontból - legnehezebb, legmunkaigényesebb területe. Az assembly programozás azt jelenti, hogy a programozó lényegében gépikódban programoz. Közvetlenül írja elő minden egyes gépikódú utasítás végrehajtását, a gép összes regiszterének, memóriája minden egyes bitjének felhasználását. Ilyen módon teljesen ő vezérli a gép működését, élvezve ezáltal a legteljesebb szabadságot, de az összes felelősség is az ő vállán nyugszik.

Napjainkban a számítástudomány fejlődése odáig jutott, hogy a programozó rendkívül széles körből választhatja meg eszközeit. Ha elég képzett, akkor szinte minden gyakorlati programozási feladathoz találhat olyan magasszintű nyelvet, amelyet éppen azon probléma kezelésére fejlesztettek ki.

Ilyen körülmények között az assembly programozás szorító kényszerűség helyett olyan eszközzé vált, amelyet a programozó bármikor harcba vethet; de pontosan olyan feladatok megoldásában, ahol hatékonyan (és elfogadható idő alatt) csak az assembly nyelv segíthet. Magyarán: az assembly visszaszorul azokra a területekre, ahol igazán szükség van rá, és a legtöbb feladatot kényelmesen, könnyedén oldhatjuk meg valamely magasszintű nyelven.

A PC/XT legelterjedtebb nyelvei a C nyelv, a PASCAL és a TURBO-PASCAL, a FORTH, a FORTRAN, a COBOL (!), a PROLOG (!), az adatbázis-kezelő nyelvek közül a dBASE-II, a dBASE-III és ennek változatai. (A felkiáltójel azokat a nyelveket jelöli, amelyek természetükből adódóan vagy tervezőik szándéka szerint jellegzetesen nagy számítógépek nyelvei - de az IBM PC/XT központi tárkapacitását és teljesítményét tekintve felveszi a versenyt azokkal a gépekkel, amelyekre például a COBOL-t tervezték.)

Az Olvasó is tisztában van azzal, hogy a legtöbb feladat e nyelvek valamelyikével (bármelyikével ??) megoldható, és az assembly programozáshoz csak akkor kell nyúlnia, ha egészen

rendszerközeli programokat akar írni (bár erre fejlesztették ki a C nyelvet), ha különleges gyorsaságra van szüksége, vagy ha hardware-interruptokat akar kezelni - ez utóbbira értelmes lehetőséget még a C nyelv sem ad.

Arra is fel kell azonban hívnunk az Olvasó figyelmét, hogy ha a fenti célok elérése érdekében assembly programokat akar írni, csak akkor számíthat jó eredményre, ha nagy gyakorlata van, és nagyon jól ismeri a gépet, az operációs rendszert, valamint a gépikódú - gépi szintű - programozás sajátos követelményeit. Az assembly nem csodaszer. Hatékony fegyver, de használata nagy gyakorlatot, felkészültséget és precizitást igényel. Ha ezek nincsenek meg, akkor biztosak lehetünk abban, hogy a fegyver bennünket fog eltalálni; sokkal nagyobb erőfeszítéssel, drágábban, olyan kódot fogunk előállítani, amely hatékonyságban esetleg alatta marad egy C nyelven vagy PASCAL-ban jól megírt programnak, csak sokkalta veszélyesebb. Ha az ember assemblyben programoz, lényegében lemond minden rendszertámogatásról, saját kezébe veszi a gép vezérlését. Ez igen könnyen járhat azzal, hogy jóvátehetetlen károkat okoz a rendszerben, sőt esetleg a mágneslemezeken is.

Ezért fontos, hogy mindig nagy körültekintéssel válasszuk meg azt a programozási eszközt, amelyet használunk. Meg kell találni a megfelelő középutat a program elkészítésének, karbantartásának, valamint felhasználásának költség- és munkaráfordításai között.

Nyilvánvaló, hogy igen hatékony, tehát gyors, kis terjedelmű kód létrehozása idő- és munkaigényes folyamat, viszont a program üzemeltetését lényegesen olcsóbbá teszi. Mérlegelni kell, hogy a program életében (felhasználási ideje alatt) hányszor fog futni, milyen célt szolgál. Természetes, hogy ha csak néhány esetben fogjuk futtatni, akkor nem érdemes sokáig simogatni, javítgatni - fő, hogy egyszer jól működjön. Ha igen sokat fut (pl. egy fordítóprogram vagy egy szövegszerkesztő) akkor mindenképpen törekedni kell a hatékonyságra - nem fejthetünk ki elég nagy igyekezetet a program kompaktabbá tétele érdekében (tehát futásidejének és memóriaigényének csökkentése nagyon fontos feladat).

Az ilyen, nagyon sokszor futó programok esetén fontos szempont még a program karbantarthatósága, fejleszthetősége is. Arra kell törekednünk, hogy a kész program forrásnyelvű kódja a lehető legjobban olvasható, követhető legyen, segítve esetleges későbbi változtatások egyszerűbb, biztonságosabb végrehajtását.

Korszerű számítógépeken mind kevésbé fontos a programok memóriaigényének csökkentése, annál lényegesebb (különösen olyan viszonylag lassú processzor esetén, mint az IBM PC/XT processzora) a futásidő csökkentése (e követelmények némileg ellent is mondanak egymásnak). Mindenesetre leszögezhetjük, hogy a legfontosabb szempont a világos programstruktúra, a program jó olvashatósága. Mikor programozási nyelvet választunk, akkor az előbb részletezett szempontokat is figyelembe kell vennünk. Mivel assemblyben a legnagyobb erőfeszítéssel sem könnyű olvasható kódot írni, ez az érv is inkább valamilyen magasszintű nyelv mellett szól, ha a feladat így egyáltalán megoldható.

1. fejezet

Az Intel 8088 mikroprocesszor

Az 1. fejezet tartalomjegyzéke

1.0. Bevezetés

Ebben a fejezetben ismerkedünk meg az IBM PC/XT számítógép processzorával. Jóllehet szinte csak itt érintünk elektronikai részletkérdéseket, itt sem helyezünk túl nagy súlyt ezekre, mivel az egész gépet és a processzort is csak a programozó szemszögéből vizsgáljuk.

Az IBM-PC/XT az Intel 8088 mikroprocesszorra épült. Ez a processzor nagyon hasonlít az Intel 8086-hoz. Az e fejezetben elmondandók ritka kivételtől eltekintve mindkét processzorra igazak. Azért is említjük külün a korábbi Intel 8086 processzort, mert ez volt a később egész családdá fejlődött processzorcsapat (8086, 8088, 80186, 80286, ...) első tagja. Számos olyan gép, amely "PC-kompatibilis" megjelöléssel szerepel, e család valamelyik tagjára épül - így az IBM AT (80286), a COMMODORE PC-20, és még a végtelenségig sorolhatnánk ezeket. Ezekben a gépekben vagy eredeti Intel processzor, vagy annak valamilyen "utánérzése" játssza a központi egység szerepét. Az ilyen processzorok pedig valamennyien - valamilyen szinten, általában felülről - kompatibilisek a család első tagjával. Programozási szempontból rendkívüli hasonlóságot mutatnak ahhoz, vagy éppen teljesen megegyeznek vele. Indokolt tehát, hogy először ezzel a processzorral ismerkedjünk meg. A további bekezdésekben azt részletezzük, melyek a legfontosabb eltérések az Intel 8086 és a bennünket érdeklő Intel 8088 processzor között.

Az Intel 8086 a cég első tizenhat bites mikroprocesszora. 1979-ben jelent meg, ezzel (akár annak idején a 8008-al és a 8080-al) az Intel vezető szerepet játszott a minőségileg új, az addigiaknál sokkal hatékonyabb processzorok kialakításában.

Az Intel 8086 nem a régi, Intel 8080 típusjelű processzor továbbfejlesztése, bár sok mindent megőrzött annak sajátosságaiból. Pusztán programozástechnikai szempontból a következőket láthatjuk: e processzor regiszterkészlete bővebb, minden regisztere tizenhat bites. Rendelkezik úgynevezett indexregiszterekkel is, ami sokkal hatékonyabbá teszi a programozást. Címzési módjainak köre sokkal szélesebb, sok szempontból megközelíti még egy PDP-11-es processzor képességeit is. Utasításkészlete is "teljesebb", ami alatt nem azt értjük, hogy "minden" utasítás megvan benne, hanem azt,

hogy csaknem minden utasítása minden címzési móddal használható (az Intel 8080 processzorban csak az adatmozgatási utasítás teljes, a legtöbb műveletet csak regiszterben lehet elvégezni). Első pillantásra a régi Intel 8080 és az Intel 8086 között a fő különbség azonban az, hogy a processzor címtere tizenhat bit helyett húsz bites, ami azt jelenti, hogy összesen egy megabyte-nyi memóriát tud címezni.

Mit jelent az, hogy az Intel 8086 tizenhat bites processzor? Ahhoz, hogy válaszoljunk erre a kérdésre, meg kell ismerkednünk a "bus" fogalmával (amit a továbbiakban az egyszerűség kedvéért busznak mondunk majd). A busz lényegében párhuzamos vezetékek halmaza, amelyre "keresztben" kapcsolódnak a gép különböző áramköri egységei: a processzor, a memória, a vezérlőegységek, stb. A busz három csoportra osztható: a címbusz, amelyen a processzor a kívánt címet közli a memóriával, az adatbusz, amelyen a processzor és az egyéb egységek közötti adatforgalom zajlik és a vezérlőbusz, amelyen, mint neve is mutatja, a vezérlőjelek "mozognak" (ilyen például az órajel, amely egy azonos időközönként lüktető impulzus a vezérlőbusz egyik vezetékén, és amely a gép elemeinek belső szinkronizálását teszi lehetővé). Természetesen nem minden elem csatlakozik minden buszvonalra - amire nincs szükség, azt nem kell feltétlenül bekötni. Elvben a processzor maga is kimaradhat néhány vonalról, jóllehet ő vezérli a gép teljes működését.

A processzor busza (tehát azon kivezetések köre, amelyek segítségével a processzort az egész gép buszához kapcsolják) is a fenti három csoportra oszlik. A címvonalak száma adja meg, milyen nagy a processzor címtere, azaz legfeljebb mekkora memória kapcsolható hozzá. Az adatbusz vonalainak száma mutatja meg, hány bites a processzor, hiszen egyidejűleg ilyen bitszélességű adat mozgatására képes.

Mivel az Intel 8086 tervezői ragaszkodtak a régebbi, eléggé elterjedt negyven lábú intergált áramköri tokhoz, arra kényszerültek, hogy a cím- és adatvonalakat közös kivezetésre kössék. A processzor tokjának kivezetései közül húsz tartozik a címbuszhoz, a többi pedig a vezérlőbusz. A húsz címvonalból összesen tizenhat adatvonalként is szolgál. Ezek a kivezetések tehát a processzor különböző működési fázisaiban hol a címek, hol pedig az adatok átvitelére szolgálnak.

Megjegyzés: a fenti felosztás elég durva, miután a címvonalak egy részét vezérlési célra is használja a

processzor - ez azonban a mi szempontunkból teljesen érdektelen, hiszen csak a cím- és az adatvezetékek számára vagyunk kíváncsiak.

A processzor címvonalainak számából következik tehát, hogy a memória maximális nagysága egy megabyte, azaz kettő a huszadikon byte lehet; mivel összesen tizenhat adatvonallal rendelkezik, ezért nevezhetjük tizenhat bitesnek. Az Intel 8086 memóriakezelése azonban igen sajátos. Csak akkor mozgat valóban tizenhat bitet egyszerre, ha a megcímzett szó páros címen kezdődik; ha a szó címe páratlan, akkor két lépésben megy végbe a szó mozgatása. Ezért ezt a processzort igazából "félig" tizenhat bitesnek mondhatjuk.

Egy processzorral ismerkedve nagyon fontos kérdés az input és output lehetőségek megvizsgálása, az interrupt rendszer, és a DMA ("Direct Memory Access", a memória direkt elérése input-output tevékenység során) lehetőségének megismerése. Ezeken a területeken nem találunk szenzációs újdonságokat. Az első két kérdést (mivel a programozót nagyon is közelről érinti) az 1.3. és 1.4. alfejezetben vizsgáljuk meg részletesen. A harmadikról (mivel ez a konstruktőrre tartozik) csak annyit, hogy a processzor biztosítja a DMA beépítésének lehetőségét és gépünk tervezői többszörösen is éltek a lehetőséggel.

Ismerkedjünk meg most az Intel 8086 processzor további újításaival, amelyek lényegesen hatékonyabbá teszik minden korábbi társánál!

Egy processzor hatékonysága nagyban függ a gyorsaságától. A gyorsaságot nagyjából az órajel frekvenciájával szokták jellemezni. Az Intel 8086 maximálisan megengedett órajele nem nagyon magas. A processzor gyorsítása érdekében vezettek be a tervezők egy szellemes megoldást, az úgynevezett "utasítás előolvasási sort", eredeti nevén az "instruction queue"-t.

Egy hagyományos processzor úgy működik, hogy előbb beolvassa a memóriából a soron következő utasítást, majd értelmezi és végrehajtja azt. Működése két "lépésre" tagolható: az utasítás beolvasása (angol szóval "fetch") és végrehajtása ("execution"). A processzor a "fetch" idején "nem csinál semmit". Az utasítás végrehajtása után a processzor újabb "fetch" lépést hajt végre és így tovább.

Tervezői az Intel 8086 processzorba beépítettek egy hat byte hosszúságú memóriát. A processzor az "execution" szakaszban, az aktuális utasítás végrehajtásának idején ebbe a (ciklikusan

használt) memóriába olvassa be a következő utasítást, így annak végrehajtására nem kell sokat várni, hiszen már a processzorban van. Tehát a "fetch" és az "execution" egyidejű, egymást átlapoló tevékenység. Ezért a processzor gyorsabb, mint ahogy az órajele alapján várhatnánk. Ez a kis memória az utasítás előolvasási sor. Természetesen vezérlésátadási utasítások végrehajtása esetén a processzor törli az utasítás előolvasási sort és elölről kezdi feltöltését, így a vezérlésátadás minden formája némileg lassítja a program futását.

Az Intel 8086 processzor két üzemmódban, az ún. minimális és maximális módban dolgozhat. Az előbbi egy egyprocesszoros környezetet jelent, az utóbbi esetben pedig több Intel 8086 processzor építhető be egy berendezésbe, köthető be egy buszra. Egy ilyen többprocesszoros gép létrehozása persze mindenképpen nagyon bonyolult feladat.

A processzor nagyon fontos sajátossága még, hogy lehetővé teszi ún. koprocesszorok használatát is. A legfontosabb ilyen koprocesszor talán az Intel cég 8087 jelű lebegőpontos műveletek elvégzésére szolgáló integrált áramköre, amely sokkal gyorsabbá teszi az aritmetikai műveletek elvégzését. Fontos megjegyeznünk azt is, hogy sajátos számábrázolása rendkívüli pontosságot tesz lehetővé. Tíz byte-os munkaregiszteret használ, tehát rengeteg bit áll a rendelkezésére mind a szám karakterisztikája, mind mantisszája számára. Az Intel 8087-et igen kényelmesen "építhetjük" be az IBM PC/XT számítógépbe: egyszerűen be kell nyomni a számára fenntartott helyre a processzor mellé - a gép bekapcsolása után rögtön üzemképes. Programozása nem túlzottan egyszerű. Felhasználásának rövid ismertetését a második kötetben olvashatjuk.

Az Intel 8086 általános sajátosságai között utolsóként említjük a programozó számára talán legfontosabb újítást, a szegmensregiszterek bevezetését. A legtöbb olyan gép, amely tizenhat bites memóriafelépítésű, de a tizenhat biten címezhető hatvannégy kilobyte memóriánál nagyobb tár címzésére képes, valamilyen "bázisregisztert" használ a címtér bővítésére.

A bázisregiszter a processzor egy belső regisztere, amelynek tartalma minden egyes memóriacímzés során hozzáadódik a tizenhat bites címhez.

> Megjegyzés: ezt a műveletet természetesen beépített elektronika végzi, teljesen automatikusan, így nem lassítja lényegesen a processzor működését.

A gép címterének nagyságát végső soron a bázisregiszter nagysága, azaz felhasznált bitjeinek száma szabja meg.

Ha alaposabban utánagondolunk a bázisregiszter működésének, látjuk, hogy a bázisregiszter segítségével egy hatvannégy kilobyte hosszúságú memóriaterületet jelölünk ki. A program ebben fut, és mivel minden címzés során összeadódik a bázisregiszter és a cím tartalma, a program úgy érezheti, hogy egy olyan memóriában dolgozik, amely a 0. címmel kezdődik és hatvannégy kilobyte hosszú; fogalma sincs arról, hogy valójában egy igen nagy, pl. négy megabyte-os memóriában, valahol a középtájon helyezkedik el. Vegyük észre, hogy a bázisregiszter használata, mintegy "melléktermékként", lehetővé teszi a programok áthelyezését a tárban - ez pedig nélkülözhetetlen a multiprogramozás megvalósításához.

Az Intel 8086 bázisregiszterei, az ún. szegmensregiszterek tizenhat bitesek, de tartalmukat négy bittel balra tolva (tehát felhasznált bitjeik számát húszra bővítve) használják fel. Ilyen módon a processzor minden egyes címzés során elvégez egy ravasz összeadást.

Magában véve ez nem jelentene semmi újat - ilyet számos gépen láttunk már. Az újdonság abban áll, hogy a processzor a négy szegmensregisztert a címek "fajtái" szerint elkülönítve használja. Míg egy hagyományos processzor esetén egy adott program (mivel minden beleépített cím tizenhat bites) egyidejűleg csak hatvannégy kilobyte-nyi memóriát használhat fel, az Intel 8086 processzoron futó programok külön bázisregisztert használhatnak az utasítások címzéséhez (a "fetch"-hez), egy másikat az adatok címzése során, egy harmadikat minden stack- (verem-) művelethez és végül rendelkezésükre áll egy negyedik is, további "ablakot" nyitva a memóriaterületre. Egy Intel 8086-on futó program tehát nem "csak" hatvannégy, hanem kettőszázötvenhat kilobyte memóriát képes egyidejűleg elérni - ez pedig igen nagy előny. A szegmensregiszterek használatának alaposabb ismertetésére az 1.1. és 1.2. alfejezetben kerül sor.

Miben különbözik az Intel 8088 processzor az imént bemutatott Intel 8086-tól?

Az Intel 8088 processzor az Intel 8086 később született, de nem továbbfejlesztett, inkább egyszerűsített változata. Ugyancsak negyven lábú áramköri tokba van építve; közös cím- és adatvonalainak száma szintén húsz, de ebben csak nyolc az adatvonalak száma - így a 8088 tisztán nyolc bites processzor.

Utasítás előolvasási sora rövidebb (csak négy byte), és igen lényeges különbség még a két processzor között, hogy egészen más rendben kell a memóriát bekötni az Intel 8086, mint az Intel 8088 számára, ezért a két processzor nem cserélhető fel egymással ugyanabban a hardware-környezetben. (A memória egyszerűbb felépítése szól az Intel 8088 mellett.) Programozási szempontból azonban nincs közöttük észrevehető különbség. Regisztereik, utasításaik, címzési módjaik stb. teljesen azonosak. Megegyeznek az input-output lehetőségeik, azonos az interrupt-rendszerük. Kívülről csak annyit érezhetünk, hogy az Intel 8088 valamivel lassúbb.

> Megjegyzés: az IBM PC/XT-t tizenhat bites gépnek szokás nevezni, mert az assembly programozó minden gépi utasítást használhat nyolc és tizenhat bites változatban is - tehát a gépet teljesen tizenhat bitesnek látja. A processzor azonban minden tizenhat bites adatot a memória kétszeri megcímzésével és olvasásával/írásával ér el.

Egy processzor hatékonysága a hardware sebessége (az órajel frekvenciája vagy a memória válaszideje) mellett legalább ennyire függ utasításkészletének és címzési módjainak hatékonyságától is. Az Intel 8086 és az Intel 8088 (legalábbis 8 bites elődeihez képest) igen jól programozható. Amennyiben egy igazi jó programozó alaposan kihasználja a processzor összes előnyös tuladjonságát, szinte elképesztő eredményeket érhet el. Erre jó példaként hozhatjuk fel a dBASE-III adatbázis kezelő programot, a TURBOPASCAL fordítót. Ha átgondoljuk, hogy sok esetben milyen bonyolult feladatokat kell elvégezniük, azt mondhatjuk, hogy sebességük, hatékonyságuk félelmetesen nagy. Ha pedig megvizsgáljuk a dBASE egyes továbbfejlesztett vagy módosított változatait (dCOMP, dACCESS stb), látni fogjuk, hogy ki lehet használni a processzort talán még jobban is.

Végezetül szeretnék utalni egynéhány játékprogramra, mint a különféle repülésszimulátorok, a WAR játék, és számtalan hasonló, amelyek sokszor hihetetlenül bonyolult grafikus képeket varázsolnak az ernyőre teljesen "élvezhetően" (azaz elég gyorsan ahhoz, hogy ne unjuk el mellettük az életünket). Ez is ékesen példázza azt, hogy bármennyit szidjuk esetleg a kezünk alatt levő gépet, nincs rossz számítógép, csak gyenge programozó.

1.1. Az Intel 8086/8088 regiszterkészlete

Az alábbiakban egy téglalap egy 16 bites regiszert jelent. A középső kis függőleges bemetszés (ahol szerepel) azt jelenti, hogy a 16 bites regiszter két 8 bites regiszterként is használható.

Vezérlőregiszterek

IP (Instruction Pointer)	szabványos utasításmutató, amely mindig az éppen végrehajtandó utasítás címét tartalmazza
SP (Stack Pointer)	szabványos veremmutató, amely a verembe (stack-be) utolsóként beírt elem címét tartalmazza
STATUS (Flag-ek)	a processzor státuszbitjei, az utolsó aritmetikai művelet szerint állnak be
BP (Base Pointer)	bázismutató, a verem (stack) indexelt címzéséhez
SI (Source Index)	kiindulási terület index, amely adatterületek indexelt címzésére szolgál
DI (Destination Index)	célterület index, amely adatterületek indexelt címzésére szolgál

Általános regiszterek

AH AL AX (Accumulator)	általános regiszter minden aritmetikai stb. művelethez (v.ö. szorzás, osztás)
BH BL BX (Base Register)	általános regiszter aritmetikai műveletekhez Címzőregiszter !
CH CL CX (Counter Register)	általános regiszter aritmetikai műveletekhez Számlálóregiszter !
DH DL DX (Data Register)	általános regiszter aritmetikai műveletekhez (v.ö. szorzás, osztás, I/O)

Szegmensregiszterek (memóriakezelés)

CS (Code Segment)	a kódszegmens, az utasítások címzéséhez
SS (Stack Segment)	a stack szegmens, a verem címzéséhez
DS (Data Segment)	az adatszegmens, az adatterület címzéséhez
ES (Extra Segment)	az extra szegmens, egy másodlagos adatterület címzéséhez

A regiszterek szerepe

AX

mint általános regiszter, csaknem minden célra használható. A regiszter alsó 8 bitje az AL (Accumulator Low), a felső 8 bit az AH (Accumulator High).

Különleges szerepet játszik a szorzási és osztási utasítások végrehajtásában; az egyik operandus minden esetben AX vagy AL.

BX

az ún. "base register", a bázisregiszter. A regiszter alsó 8 bitje a BL (Base Low), a felső 8 bit a BH (Base High). Minden művelet elvégzésére használható (kivéve a szorzást és osztást).

Célja: indirekt címzés és bázisregiszteres indirekt címzés megvalósítása.

CX

"counter register", a számlálóregiszter. A regiszter alsó 8 bitje a CL (Counter Low), a felső 8 bit a CH (Counter High). Minden művelet elvégzésére használható (kivéve a szorzást és osztást).

Célja: a ciklusszervező utasítások, valamint a string-utasítások (melyek végrehajtását ismételtethetjük) számlálóregisztere.

Kiegészítés:

CL

A számlálóregiszter alsó byte-ja a bitforgató és a biteket léptető utasítások számlálóregisztere.

DX
--

"data register", az adatregiszter. A regiszter alsó 8 bitje a DL (Data Low), a felső 8 bit a DH (Data High). Minden művelet elvégzésére használható (kivéve a szorzást és osztást, melyek végrehajtásában sajátos szerepet játszik).

Célja: egyfelől az I/O utasítások során a port (lásd 1.3.) címét tartalmazza, másfelől pedig a szorzási és osztási utasítások használják a duplaszavak tárolására.

Megjegyzés: az Intel 8080 és az Intel 8086 regiszterei között a következő analógia áll fenn:

Intel 8080	Intel 8086/8088
A	AL
(B, C)	(CH, CL)
(D, E)	(DH, DL)
(H, L)	(BH, BL)
PSW	STATUS alsó 8 bitje

SI, DI

"source index", "destination index", az indexregiszterek. Minden művelet elvégzésére használhatóak (kivéve a szorzást és osztást).

Céljuk: indirekt és indexelt címzés megvalósítása. Különleges szerepet játszanak a string utasítások végrehajtásában: e regiszterek tartalmazzák a kiindulási- és/vagy céloperandusok címét.

BP
--

"Base Pointer", báziscím-mutató. Minden művelet elvégzésére használható (kivéve a szorzást és osztást).
Célja: a stack-szegmensben indirekt és indexelt címzés megvalósítása. Más célra való felhasználása ellenjavallt. Egyes nyelvek megvalósításában (pl. C nyelv) különleges szerepe van.

SP
--

"Stack Pointer", veremmutató. Automatikus regiszter. Mint neve is mutatja, a stack címzésére, kezelésére szolgál. Közvetlen utasítással, "kézzel" írása, olvasása lehetséges, de igen nagy gondosságot igényel. Belépő programunk "kap" egy stack-területet, amelyet a Stack Pointer megcímez. A Stack Pointer kiolvasása, felülírása csak különlegesen indokolt esetben tanácsos (például ha programunkból újabb programot indítunk el).

> Megjegyzés: a stack a programozás egyik legfontosabb fogalma. Egyszerűen szólva egy olyan memóriaterület, amelyet "last in, first out" (utolsóként be, elsőként ki) módon használunk címek és adatok ideiglenes tárolására.
> Magyarul: ha egy adatot be akarunk írni a stack-be, akkor az SP értéke 2-vel csökken (a stack szavas szervezésű), és az általa most címzett szóra íródik be az adat. Ha pedig ki akarunk olvasni egy adatot, akkor az SP által most címzett szó tartalmát kapjuk meg, majd az SP értéke 2-vel nő - így mindig a felhasznált terület végére írunk (last in) és az utoljára beírt elemet vesszük el először (first out). Ez a szervezés teszi lehetővé a szubrutinok végrehajtását: a visszatérési címet a stack-re tesszük (lásd CALL utasítás, 3. fejezet), majd a visszatéréskor innen vesszük el (lásd RET, u.o.). Szubrutinból újabb rutin meghívása is lehetséges, mert minden rutin mindig az utolsó elemet veszi el, tehát éppen azt, amit számára tett le a hívó. A

szubrutinok így tetszőleges mélységben hívhatók egymásból.
Szokás a stack-et még ideiglenes adattárolásra használni. Például egy "illedelmes" szubrutin elmenti és visszaállítja azon regiszterek tartalmát, amelyeket nem használ válaszadásra. Erre a célra kézenfekvő a stack-et használni.
Lássunk erre egy példát! Tegyük fel, hogy a stack pointer értéke 1206H (a stack szegmens helyét figyelmen kívül hagyjuk), és végrehajtunk egy ún. rövid szubrutinhívást, azaz meghívunk egy olyan szubrutint, amely a hívó program kódszegmensében helyezkedik el. A meghívott szubrutin elmenti az általa felhasznált SI és DI indexregisztereket [melyek tartalma legyen például 5F00H (SI) és 35C3H (DI)], majd munkája során meghív egy újabb szubrutint. Az egymást követő ábrák szemléltetik majd a stack tartalmát és az SP viselkedését. A példában utasításokat is megadunk "forgatókönyv-szerűen": bal oldalon, kissé szintezve olvashatók a végrehajtott utasítások, a jobb oldalon pedig szöveges magyarázat és az SP, valamint a stack látható.

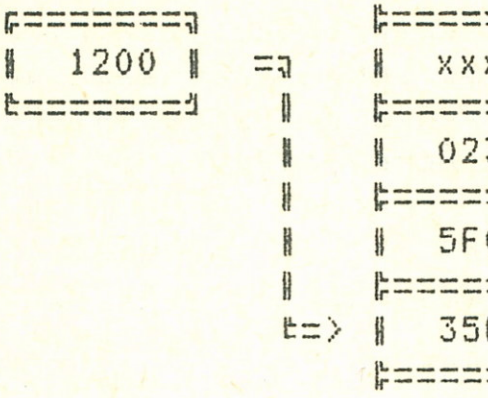
```
Utasítások          SP      Stack, magyarázatok

                  ╔═══════╗       ╠═══════╣ a stack
                  ‖  1206 ‖  ===> ‖  xxxx ‖ utolsó
                  ╚═══════╝       ╠═══════╣ eleme
    .
0234  CALL  RUTIN1          ;CALL utasítás és virt. címe
0237  .                     ;Hossza 3 byte

                  ╔═══════╗       ╠═══════╣ régi utol-
                  ‖  1204 ‖  =╗    ‖  xxxx ‖ só elem
                  ╚═══════╝   ‖    ╠═══════╣ elmentett
                              ╚=> ‖  0237 ‖ visszaté-
                                  ╠═══════╣ rési cím
```

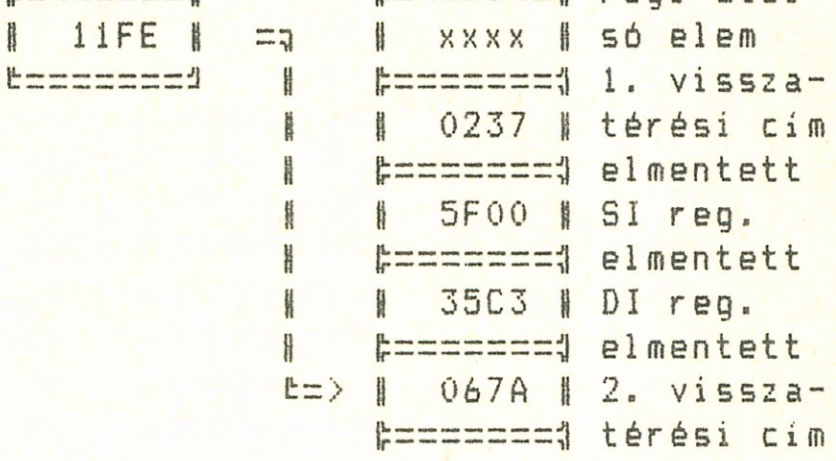
Az SP kettővel csökken, a visszatérési cím beíródik a stack-be, ezután a CALL a hívott rutin címét tölti az IP-be.

```
0675    RUTIN1:                 ;Hivott rutin és virt. cime
0675            PUSH    SI      ;SI mentése (első utasitás)
0676            PUSH    DI      ;DI mentése
```



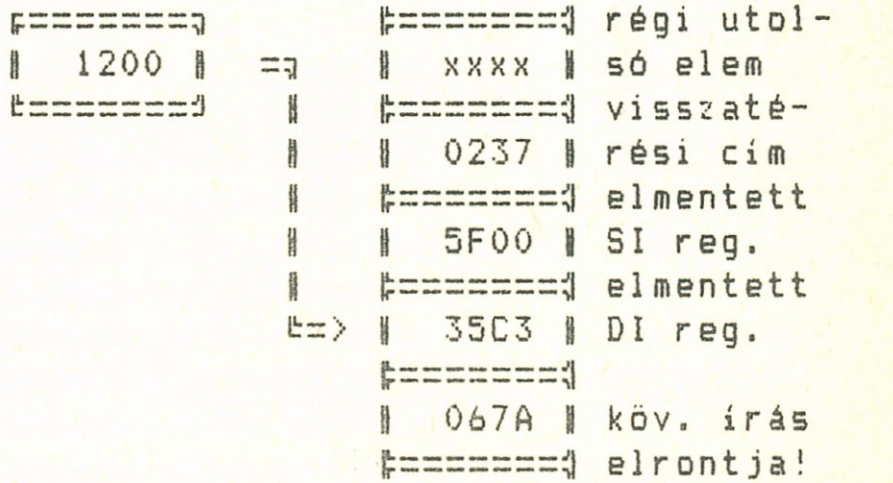
SI és DI tartalma a stack-re íródik, mialatt SP értéke kétszer kettővel csökken.

```
0677            CALL    RUTIN2  ;CALL utasitás és virt. cime
067A                            ;További utasitások
```



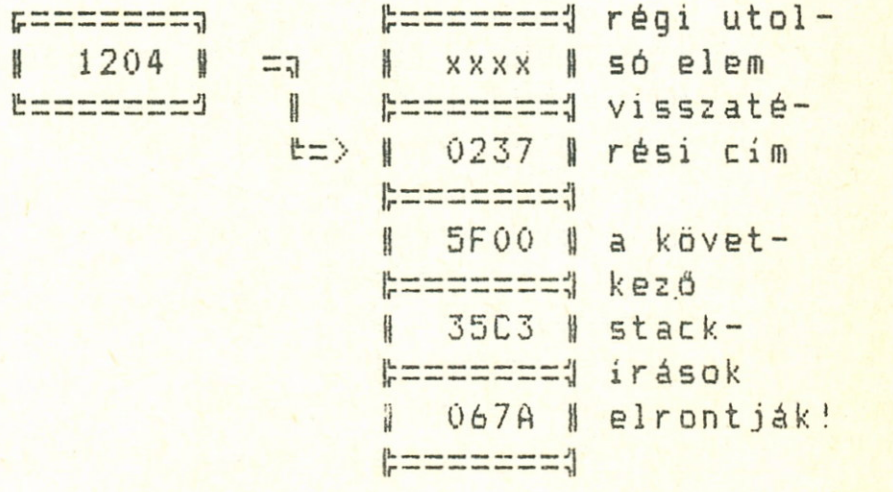
Az SP értéke kettővel csökken, a második visszatérési cím a stack-re íródik, az IP-be a második meghívott rutin címe kerül.

```
1342    RUTIN2:
                        ...     ;A meghivott második rutin
                        RET     ;Visszatérés az első rutinba
```



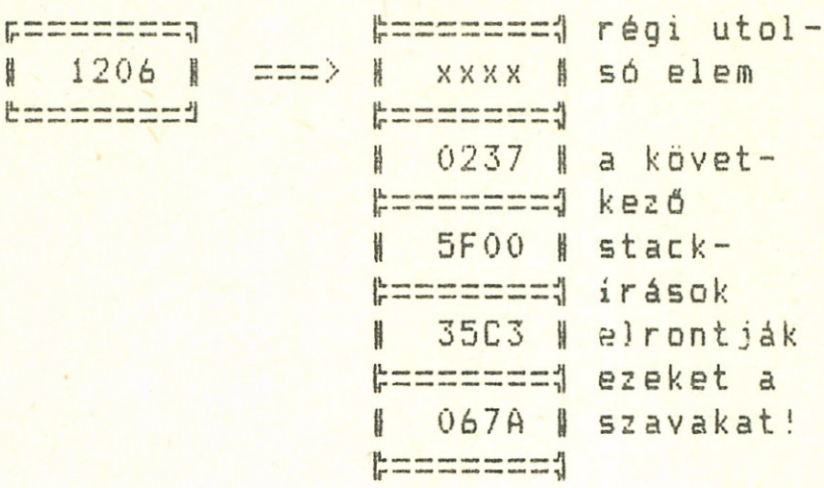
Az utolsó elem RET hatására visszairódik az IP-be, majd SP értéke kettővel megnő (így a programvégrehajtás az első hivási utasítás után folytatódik).

```
                ...             ;1. rutin utasításai
                                ;Visszatérés előkészítése
069A            POP     DI      ;DI helyreállítása
069B            POP     SI      ;SI helyreállítása
```



SI és DI tartalma fordított sorrendben visszairódik a stack-ről, eközben SP értéke kétszer kettővel nő.

```
069C            RET             ;Visszatérés első rutinból
```



Az első visszatérési cím a stack-ről IP-be töltődik, így a program végrehajtása az első hívási utasítás után folytatódik.

Megjegyzés: figyeljük meg, hogy minden írási utasításnak megvan az olvasási párja (CALL-RET, illetve PUSH-POP). Kínosan kell ügyelnünk arra, hogy a stack-et össze ne keverjük, mert ez a program hívási struktúrájának teljes összeomlását vonja maga után.

A C nyelv egyik alapvető szabálya az, hogy a hívó és a meghívott közötti paraméterátadás stack-en történik, sőt, a meghívott rutin (ott: függvény) belső változóit is a stack-en hozza létre. Ez teszi lehetővé a rekurzív szubrutinhívást. Tehát így az önmagát meghívó rutin nem teszi tönkre saját adatterületét, hiszen az a stack-en újra létrejön. Ezt a programozási gyakorlatot érdemes követni assemblyben is, ha valamilyen ok miatt reentráns (újra beléptethető) vagy rekurzív (önmagát újra meghívó) programot kell írnunk. A reentráns kód igen gyakori követelmény például vezérlési programok esetén.

IP
--

"Instruction Pointer", utasításmutató. Automatikus regiszter. Mint neve is mutatja, a soron következő utasitás címét tartalmazza. Módosítása vezérlésátadási utasításokkal lehetséges.

> Megjegyzés: a processzor működése során az IP tartalmazza a soron következő utasitás címét. Az utasítás beolvasása (fetch) közben az IP automatikusan növekszik annyival, amilyen hosszú az utasítás byte-okban. Ezért mutat IP mindig a soron következő utasításra.
> A processzor vezérlésátadást (ugrás, rutinhívás) úgy hajt végre, hogy az IP pillanatnyi tartalmát felülírja a kívánt új címmel. Az ilyen utasítás végrehajtása után a program futása a kívánt címen található utasítással folytatódik.
> Az irodalomban használt hagyományos elnevezés a PC, Program Counter. Ez elég szerencsétlen név, mert arra utal, mintha ez a regiszter "számolná" a programot. Erről természetesen szó sincs!

STATUS

A státuszregiszter a processzor vezérlő flag-jeit tartalmazza. Módosítása különleges utasításokkal lehetséges. Tartalmát azonban általában a processzor automatikusan határozza meg, hiszen a STATUS tartalma a program végrehajtása közben bekövetkező események függvénye. Vizsgálata elsősorban feltételvizsgáló utasításokkal szokásos, de tartalma közvetlenül is kiolvasható. Tartalmát szokásos menteni és visszaállítani is (ez is kiolvasás és felülírás).
A státus bitjeinek részletes ismertetését lásd az alábbiakban; míg a bitek szerepének, működésének, jelentésének részletes ismertetésére az utasításkészletet leíró 3. fejezet végén, annak 3.10. pontjában kerül sor. Indokoltnak látszik előbb az utasításokat megismerni, és csak később foglalkozni ilyen "mellékhatásokkal".

CS
--

"Code segment", kódszegmens-regiszter. A processzor címképzési eljárása során van szerepe (lásd 1.2.). Mindig az aktuális program-modul báziscímét tartalmazza. Minden utasításelérés, tehán minden "fetch" a kódszegmens-regisztert használja.

Tartalma kiolvasható, módosítása csak vezérlésátadással lehetséges.

> Megjegyzés: a CS módosítása akkor szükséges, ha programunk kódrésze nem fér el egy 64 Kb-nyi területen. Ilyenkor a kódrészt több szegmensre kell bontani. A szegmensközi vezérlésátadások során szükséges a CS módosítása. Erre a célra speciális utasítások, pontosabban a CALL és JMP utasítások speciális formái szolgálnak.

SS
--

"Stack Segment", stack-szegmens regiszter. A processzor címképzési eljárása során van szerepe. Mindig a stack-ként használt memóriaterület báziscímét tartalmazza.

E regiszter írható-olvasható. Módosítása különleges gondosságot igényel.

> Megjegyzés: mikor a program belép. akkor az SS értéke elő van készítve, az SP-hez hasonlóan. Természetesen az SS módosítása csak különösen indokolt esetben ajánlott (például akkor, ha programunkból más programot indítunk el - mint a második könyvben látni fogjuk, a rendszer nem garantálja azt, hogy az SS és SP regiszterek tartalma sértetlen marad).

DS
--

"Data Segment", adatszegmens-regiszter. A processzor címképzési eljárása során van szerepe. A program (egyik) adatterületének

címét tartalmazza - mindig az e regiszter által címzett adatterület elérése a legkényelmesebb.
E regiszter írható-olvasható. Alapértelmezésben minden adatelérés során ezt használjuk.

> Megjegyzés: e regiszter különleges szerepet játszik a string utasítások végrehajtásában. Alapértelmezés szerint a "kiindulási string-et" a DS:SI regiszterpár segítségével címezzük meg. Ettől azonban a programozó külön prefixum megadásával eltérhet.

ES
--

"Extra Segment", extraszegmens-regiszter. A processzor címképzési eljárása során van szerepe. Egy "másodlagos" adatterület báziscímét tartalmazza.
E regiszter írható-olvasható. Általában egy másodlagos adatterület eléréséhez, címzéséhez használjuk.

> Megjegyzés: az ES használatához minden esetben egy ún. prefixumot, az utasítást megelőző és azt módosító előtagot kell használnunk, amelynek helyigénye egy byte. A 4. fejezetben olvasható, hogyan lehet ezt az assemblernek előírni.
> Az ES regiszternek speciális szerepe van a string utasítások során: a "cél-string" címzése mindig az ES:DI regiszterpár segítségével megy végbe.

Mint látható, az Intel 8086/8088 processzornak számos regisztere van. Ezek a regiszterek nem egyenrangúak, mert nem lehet mindegyiket tetszőleges célre felhasználni. A leginkább fájdalmas megkötés az, hogy csak három regiszter alkalmas címzésre (BX, SI, DI - a BP ilyetén jellegű felhasználásával óvatosan bánjunk!). Kellemetlen lehet talán, hogy a szorzás és osztás csakis az AX regiszterben végezhető, bár azt hiszem, hogy a szokványos assembly programozási feladatok során ilyen jellegű aritmetikai műveletekre ritkán kerül sor. Egyebekben, mint látni fogjuk, a regiszterkészlet kényelmes programozást tesz lehetővé, bár erősen köti kezünket, hogy a regisztereket nem használhatjuk egymás helyett teljesen szabadon.

A STATUS bitjei

(az alábbiakban az "x"-el jelölt bitek nem használtak; a "t" azt jelenti, hogy az a bit vezérlésátadási utasítással tesztelhető, "s" pedig azt, hogy a bit utasítással közvetlenül állítható)

Alsó nyolc bit

t	t				t		t
S	Z	x	A	x	P	x	C
							s

s	z	a	c	p	c
i	e	u	a	a	a
g	r	x	r	r	r
n	o	i	r	i	r
		l	y	t	y
				y	

Felső nyolc bit

				t			
x	x	x	x	O	D	I	T
				s	s	s	?

O	D	I	T
v	i	n	r
e	r	t	a
r	e	e	p
f	c	r	
l	t	r	
o	i	u	
w	o	p	
	n	t	

A bitek részletes ismertetése

0. C, carry — szabványos átviteli ("carry-") bit, tulajdonképpen úgy tekinthető, mint egy operandus 9. (17.) bitje; sokszor használják indikátorként, azaz Igen/Nem feltételek jelzésére is. Utasítással tölthető, törölhető és meg is lehet fordítani (komplementálni).

2. P, parity — szabványos paritásbit; értéke 1, ha az eredmény-byte-ban (ha az eredmény szó, akkor alsó 8 bitjében) az eggyessel töltött bitek száma páros, különben 0.

4. A, auxiliarry carry — átvitel ("carry") a 3. bitről a 4.-re. Csak a pakolt BCD számokkal végzett műveletekhez használatos. Minden ilyen művelet után valamilyen kiigazítást kell végezni az eredményen, az erre szolgáló korrekciós utasítások használják e bitet (lásd pl. DAA, 3. fejezet).

6. Z, zero — szabványos zéró-bit, mely azt jelzi, hogy egy művelet eredménye 0 vagy sem; a bit 1 lesz, ha az eredmény 0 és 0 akkor, ha az eredmény 0-tól különbözik.

7. S, sign — szabványos előjelbit, mely együtt változik az eredmény előjelével: értéke 0, ha az eredmény pozitív és 1, ha negatív.

8. T, trap — beállítva "single step" mód, azaz a processzor az 1. interrupt-ot hajtja végre minden egyes utasítás után (az pedig beléptetheti egy DEBUG-program "single step"-jét). Az interrupt végrehajtása után a flag automatikusan

törlődik, a visszatérés után pedig újra egyes értéket kap.
Utasítással közvetlenül nem lehet módosítani, csak ha "kivarázsoljuk" a felső byte-ot.

Megjegyzés: ehhez a PUSHF utasítással el kell menteni a STATUS-t, visszaemelni egy általános regiszterbe, egy OR utasítással bebillenteni a "Trap"-bitet, majd újra stack-re tenni és a POPF utasítással visszatölteni a módosított STATUS-t.

9. I, interrupt — interrupt-tiltás vagy -engedélyezés; ha ez a bit 1, akkor a processzor minden hardware interrupt-ot fogad, ha pedig 0, akkor a processzor az NMI (Non Maskable Interrupt) kivételével nem fogad semmilyen hardware interrupt-ot.
Utasítással közvetlenül állítható.

10. D, direction — ciklikus műveletek iránya; a string-kezelő utasítások használják, amelyek automatikusan módosítják a pointerként használt SI és DI indexregiszterek tartalmát. Ha e bit értéke 0, akkor a módosítás növelést, ha 1, akkor csökkentést jelent.
Utasítással közvetlenül állítható.

11. O, overflow — túlcsordulás: egy "kizáró vagy" művelet eredménye a legmagasabb helyi-értékről, és -re csordulások között. Mint látható, ez egy aritmetikai flag. Ha egy művelet eredménye nem előjelhelyes (mert előjelbitre csordulás következett be), akkor ez a bit 1 lesz, ha pedig előjelhelyes, akkor 0.
Ez a bit feltételes vezérlésátadási utasítással tesztelhető, és van egy különleges "feltételes interrupt" utasítás, amely szintén e bit alapján működik (lásd INTO).

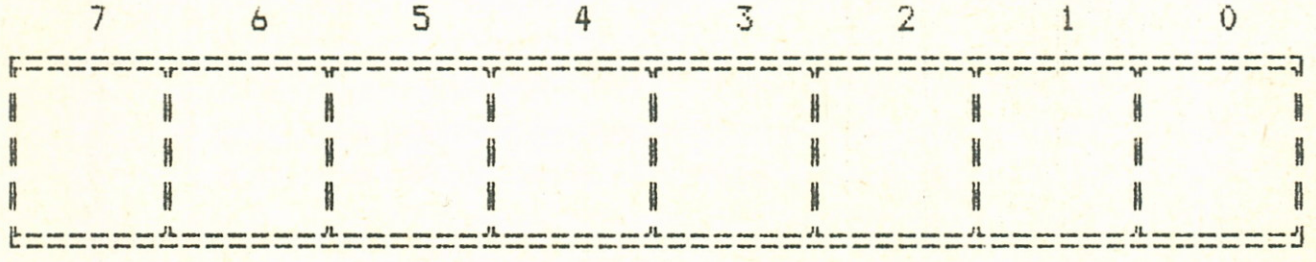
1.2. Az Intel 8086/8088 memóriakezelése

Az Intel 8088 mikroprocesszor memóriája byte-os szervezésű. Minden byte 8 bitből áll. Bármely két szomszédos byte-ot egy szóba foghatunk össze (a szavak páros vagy páratlan memóriapozíción is kezdődhetnek). Egy szó alacsonyabb helyiértékű bitjei az alacsonyabb című byte-ban vannak.

A memória nagyobb egysége a paragrafus és a lap. A paragrafus 16-al osztható címen kezdődik, hossza 16 byte. A lap (page) kezdőcíme 256-al osztható, hossza is 256 byte.

> Megjegyzés: a Microsoft Macroassembler 3.00 verzió a lapot 1024 byte hosszúnak definiálja és 1024-el osztható címen kezdi.

A memória egy byte-ját a következő módon ábrázoljuk, a biteket nagyságrendjük szerint 0-tól 7-ig (szavak esetén 0-tól 15-ig), jobbról balra haladva:



Az Intel 8088 mikroprocesszor minden regisztere 16 bites, címtere azonban 20 bitre bővített, így közvetlenül 1 Mb (megabyte) kezelésére képes. Az egyes regiszterekben (vagy az utasításokban elhelyezett operandusokban) tárolt 16 bites címet a szegmensregiszterek segítségével terjeszti ki 20 bitre a következő módon: a kiválasztott szegmensregiszter tartalmát 16-al megszorozza (ez 4 bináris helyiértékkel balra tolást jelent), ehhez adja a 16 bites címet, s az így előállított 20 bites címet továbbítja a memóriának.

Így a memóriában szegmenseket jelölünk ki, amelyek 16-al osztott kezdőcímét töltjük az egyes szegmensregiszterekbe (tehát szegmens csak 16-al osztható címen kezdődhet, azaz paragrafushatáron). Minden szegmens 64 Kb hosszú, a szegmensen belül a program által megszabott 16 bites címmel (azaz a

szegmensen belüli eltolással) érhetünk el minden byte-ot. A program által adott 16 bites címet nevezzük virtuális címnek, míg a kiválasztott szegmensregiszterhez való hozzáadás után keletkezett 20 bites cím a fizikai cím.

A szegmensből nem lehet "kicímezni", mert a processzor a 16 bites címet előjel nélküli mennyiségnek fogja fel. Amennyiben pedig az összeadás során túlcsordulás keletkezne, az elvész. Ha egy szegmensregiszter igen magasra címez a memóriában és olyan magas 16 bites címet adunk meg, hogy túlcsordulás keletkezik, akkor a valójában címzett pont a memória alsó részén lesz!

> Megjegyzés: az eredeti megnevezés az eltolásra, a szegmensen belüli címre "Effective Address", a fizikai címre egyszerűen "Address".

A szegmensregiszter kiválasztása a kovetkező: utasításcím esetén (ha IP-n át címzünk) mindig a CS, ha a stack-et címezzük (szubrutinhívás, vagy PUSH-POP utasítások) mindig az SS, adatelérés esetén alapértelmezésben DS, de itt (egy ún. "prefixum", utasításmódosító előírás felhasználásával) bármely más szegmensregiszter használható.

Ilyen módon egyidejűleg 256 Kb-ot (Kilobyte-ot) "lát" a processzor, de viszonylag egyszerűen elérhető az összes byte.

E szervezés fő előnyei a következők:

a.) a cím csak mint a szegmensregiszterben tárolt ún. báziscímhez adódó offset szerepel, tehát az egyes programszegmensek áthelyezhetőek (viszonylag egyszerűen valósítható meg a multiprogramozás);

b.) teljesen különválasztható a programok adat- és kód-területe; ez a tisztább programozás mellett lehetővé teszi a többszörösen használt progamok írását.

A hátrányok:

a.) kevésbé gyakorlott programozó nehezen tekinti át programja memóriaszervezését;

b.) bármely program "szabadon garázdálkodhat" a tárban, bármikor bármilyen területet felülírhat.

Program- és adatterületek szervezése

Mint a fentiekből következik, az Intel 8086 és 8088 processzor memóriájának teljes címzéséhez nem 16, hanem 32 bitre van szükség (a teljes 20 bites cím egy 16 bites szegmenscím és egy 16 bites virtuális cím kombinációjaként hardware-úton keletkezik). Az egyszerre látott 256 Kb (amely persze a valóságban "csak" 192, mert a stack-területet nem szoktuk közvetlenül címezni, sőt, ha szigorúan vesszük, akkor csak 128 Kb, mert az ES felhasználása külön utasítást igényel) általában sok mindenre elég.

A legegyszerűbb esetben rendelkezésünkre áll egy 64 Kb folytonos program- és egy 64 Kb hosszú adatterület, amelyeket a szegmensregiszterek egyszeri beállításával tudunk elérni. Programirás során gondosan meg kell terveznünk, hogy milyen kód- és adatcsoportokat kívánunk definiálni. Az egyes csoportok folytonosan helyezkednek majd el a memóriában, és elemeiket egy szegmensregiszter-érték segítségével változó virtuális címeken érjük el. Ezeket a folytonos memória-területeket is szegmenseknek nevezzük. A szegmensek definiálására a fordító (mint a 4. fejezetben olvasható) kényelmes és szemléletes lehetőséget biztosit.

Programtervezés közben törekedjünk a adat-, de különösen a programszegmensek számának lehetőség szerinti csökkentésére, mert a szegmensregiszterek sokszori felülirása nehézkessé, áttekinthetetlenné teszi a programot, és nem is veszélytelen: ha átírjuk pl. a DS regisztert, a fő adatszegmens helyett egy másodlagos adatterület címzéséhez, akkor a fő adatszegmens adatait a cím helyreállitásáig nem tudjuk elérni.

Ha több programszegmenst definiáltunk, ügyeljünk arra, hogy lehetőleg mindig szubrutinhivási (CALL) utasitást használjunk a szegmensközi vezérlésátadásra, és óvakodjunk a szegmensközi ugrásoktól (JMP)! Ez igen veszedelmes dolog, mivel a JMP végrehajtása után már csak a saját programozási technikánk és biztonságunk a garancia arra, hogy vissza tudunk térni a kiindulási programszegmensbe. A precíz visszatérés pedig nagyon fontos, mert a DOS szabványos kiléptető függvényei, amelyek segitségével programunk visszaadhatná a vezérlést az operációs rendszernek, feltételezik, hogy abból a programszegmensből hivtuk meg őket, amely a program belépési pontját is tartalmazta.

Ezért programunkat célszerű úgy megszervezni, hogy legyen egy főmodulunk, amely a program be- és kilépési pontját tartalmazza. Ez később részletezendő módon készítse elő a program visszatérését, majd a program logikája szerint CALL utasítással hívja meg rendre az egyes funkciókat ellátó rutinokat. A rutinok normális esetben egyszerűen visszatérnek a hívóhoz, amely végül szokásos módon visszatér a programot aktivizáló operációs rendszerhez. Ha a meghívott rutinok valamilyen okból "vészkijáraton" kényszerülnek kilépni (például észreveszik, hogy éppen most írják felül a merev lemez fő directory-ját), akkor ugorjanak vissza a főmodul ún. "error exit" pontjához, amely (esetleg valamely hibaüzenet kiírása után) kilépteti a programot.

Nagyon fontos programozástechnikai szempont, hogy minden szubrutinnak (és így az egész programnak is) ha lehetséges, egy belépési és egy kilépési pontja legyen. Ilyen módon garantálhatjuk, hogy a belépés után elvégzünk minden szükséges inicializálást, a kilépés előtt pedig megfelelően "rendet teremtünk" magunk után.

1.3. Az Intel 8086/8088 I/O lehetőségei

Az Intel 8088 processzor az I/O tevékenységet portos rendszerben végzi. A port "kapu" a processzor és a külvilág (a perifériakontrollerek) között. (A "port"-ot nem túl szerencsés elnevezéssel "porta" szóval magyarították; mi a továbbiakban egyszerűen portnak nevezzük.) Egy portot úgy foghatunk fel, mint valami regisztert. Speciális utasításokkal tudjuk írni és olvasni őket. Ha egy adatot kiírunk a portra, az a megfelelően bekötött perifériakontroller egy regiszterébe kerül és így a kontrollernek szóló információ. Ha a perifériakontroller egy porthoz kötött regiszterét a porton keresztül beolvassuk, akkor a perifériakontrollertől kaphatunk információt.

A portok a memóriához hasonlóan számokkal címzettek. "Legfeljebb" 64K darab port köthető be, amelyeket 0-tól 255-ig közvetlen címzéssel, 0-tól 65535-ig pedig a DX regiszter felhasználásával regiszteres indirekt címzéssel tudunk megcímezni.

Az Intel 8088 processzor 8 és 16 bites I/O portokat tud kezelni. Minden I/O művelet egyik operandusa az AL vagy AX regiszter, a másik pedig a megcímzett I/O port.

A perifériakontrollereket mind pollozással, mind hardware interrupt-ok segítségével vezérelhetjük.

> Megjegyzés: a "poll" (kiválasztás) azt jelenti, hogy a program sorra "kiválasztja" a használt perifériákat, azaz (I/O portjaik beolvasásával) "megkérdezi", hogy történt-e a periférián valami esemény, vagy sem. (Például egy terminál esetén leütöttek-e egy billentyűt.) Ha igen, akkor a program lekezeli az eseményt, ha nem, továbblép a következő perifériára, lekérdezi azt, és így tovább.
> Az interrupt-os perifériakezelés során a program nem "foglalkozik" a perifériával, hanem maga a perifériális egység "szól", azaz kikényszeríti az eseményt lekezelő szubrutin meghívását (lásd következő alfejezet).
>
> Megjegyezzük még, hogy a perifériakontrollerek közvetlen írása nem veszélytelen dolog. Egy-két "jól irányzott" kiírással például leformázhatjuk a floppy-disk egy sávját, vagy akár a teljes operációs rendszert is lerombolhatjuk. Mielőtt tehát ilyen feladatokra vállalkozunk, gondosan tanulmányoznunk kell a port mögött elhelyezett kontrollert!

1.4. Az Intel 8086/8088 interrupt-rendszere

Az Intel 8088 processzor 256 szintű vektoros interrupt (megszakítási) rendszerrel rendelkezik. A memória elején, a 0. címtől kezdődően minden interrupt-hoz egy 4 byte hosszú blokk tartozik. Az alsó 2 byte (1 szó) tartalmazza az interrupt-rutin offsetjét (ez kerül majd IP-be), a felső 2 byte (1 szó) pedig a rutin szegmensét (ez kerül majd CS-be).

Az interrupt-folyamat vázlatosan a következő: az interrupt-ot kérő egység "felemeli" a processzor "INT" vagy "NMI" lábán a feszültséget. Ekkor a processzor befejezi az éppen futó

utasítás végrehajtását, felemeli a feszültséget az "INTA" ("INTerrupt Acknowlegde") lábon. Ekkor az interrupt-ot kérő egység az adat/címvonalak alsó 8 bitjén beadja az interrupt sorszámát, amit a processzor beolvas.

Megjegyzés: a valóságban persze nem maga az interrupt-ot kérő egység végzi mindezen műveleteket; a gép része az ún. interrupt-vezérlő is (típusjele 8259), amely 8 egységet tud kezelni. Az egységek az interrupt-vezérlő közvetítésével állnak kapcsolatban a processzorral; az interrupt-vezérlő besorolja az esetleges "szimultán", egyszerre érkező kéréseket, továbbítja a processzornak, és ő adja be az interrupt sorszámát az interrupt nyugtázása után.
Az még megjegyzésre érdemes, hogy a 8259 lehetővé teszi az általa kezelt interruptok külön vezérlését (azaz bármelyiket külön-külön is letilthatjuk vagy beengedhetjük).

Az interrupt-vektor sorszámának beolvasása után a processzor elmenti a STATUS-t, az IP-t és végül CS-t a stack-re, majd a beolvasott interrupt-sorszámhoz rendelt blokkból kiolvassa az új IP-t és CS-t, betölti őket és automatikusan "Disable Interrupt" módba lép (törli az "I" (interrupt) flag-et, letiltva a további interrupt-okat).

Az interrupt rutin végén egy IRET (Interrupt RETurn) utasításnak kell állnia, mely visszatölti CS-t, IP-t és a STATUS-t.

Megjegyzés: az interrupt-rutinok, amelyeket sajátosan meghívott szubrutinoknak is tekinthetünk, az ún. "kernel" (mag)-szinten futnak, míg a felhasználó közönséges programjai az ún. "user"- (felhasználói) szinten. Ez a két elnevezés arra utal, hogy "rangkülönbség" van a két programfajta között. A kernel-szint mindig megszakíthatja a user-szintet, de a kernel-szint további megszakítását általában nem engedélyezik. Kernel-szinten fut az operációs rendszernek az a "magja", amely közvetlenül vezérli a perifériákat, illetve a perifériakontrollereket (lásd második könyv).

Az Intel 8088 processzor sajátossága, hogy programból is kiválthatjuk bármelyik interrupt-ot az INT utasítással. A programozási gyakorlatban ezt "software interrupt"-nak nevezzük, és élesen elkülönítjük a hardware interrupt-tól. Ez az eljárás a szubrutinhívás egy sajátos formája, melynek során a memória bármely pontjára átadhatjuk a vezérlést, és automatikusan stack-re mentődik a STATUS is.

A software interrupt-ot különböző programok, így például a mi programjaink és az operációs rendszer kommunikációjára használjuk fel; a rendszert hívó program kiadja a megfelelő INT utasítást, amely itt lényegében egy "távoli" szubrutinhívás. A kernel-szint (a rendszer) és a program között ilyenkor általában regiszterekben folyik az adatátadás.

> Megjegyzés: egy software interrupt a programozó szándéka szerint, előre megtervezett körülmények között megy végbe és valóban semmi más, mint egy szubrutinhívás (csak éppen nem azt tudjuk, hogy "hol" van a meghívandó szubrutin, hanem azt, hogy hányadik interrupt-vektorba van betéve a címe). A hardware interrupt egy kívülről kényszerített szubrutinhívás, nem tudhatjuk, hogy programunk mely két utasítása között következik majd be. Ezért hardware interrupt rutinok írásakor rendkívüli gondossággal kell eljárni, minden felhasznált regisztert menteni és helyreállítani, és nagyon pontosan meg kell terveznünk a user- és kernel-szint közötti kommunikációt. Sok esetben igen nehéz a user- és kernel-szinten, egymástól szinte függetlenül futó, mégis együttműködésre kényszerülő programok összehangolása, szinkronizálása.
> Erre egy nem teljesen kidolgozott, de a probléma nehézségeit jól illusztráló példa olvasható a második kötetben.

A rendszer csaknem minden szabványos perifériát interruptok segítségével kezel. A perifériális egységek tevékenységük megkezdését vagy befejezését a megadott hardware-interrupt aktivizálásával jelzik. Az MS-DOS összesen 7 interruptot használ fel, mint hardware-interruptot. Mint már említettük, a beépített interrupt-vezérlő legfeljebb 8 hardware interrupt kezelésére képes. A szabadon hagyott interrupt mindenesetre

lehetőséget biztosít arra, hogy (egyébként nem is kicsi) hardware-es beavatkozás segítségével újabb interrupt-vezérlőt építsünk be és így újabb hardware-interruptok kezelését tegyük lehetővé.

Megjegyzés: amennyiben egymással kommunikáló programokat kell írnunk, a programok egymásnak legkönnyebben software interruptok segítségével adhatják át a vezérlést. Minden program, amely rezidens módon "beköltözik" a memóriába, a számára előre kijelölt interrupt-vektor tartalmát elmenti, majd a saját belépési címét tölti be. Ezután a vezérlést végző főmodulnak nem konkrét címekre kell átadnia a vezérlést, hanem a megadott interruptot kell aktivizálnia.
Az MS-DOS interruptjainak kiosztását teljes részletességgel a második kötetben olvashatjuk majd. Itt is megadjuk azonban a programozó számára fenntartott interruptok sorszámát: a hexadecimális 60 az első, a 7F az utolsó az ún. user-interruptok sorában. Ezenkívül a hexadecimális F1-től FF-ig terjedő interruptokat is használhatjuk. Ez összesen 47 interrupt, ami, azt hiszem, bőségesen elegendő.

2. fejezet

Az Intel 8086/8088 címzési módjai

A 2. fejezet tartalomjegyzéke

2.0. Bevezetés

Az Intel 8086/8088 processzor címzési módjainak segítségével igen kényelmesen, hatékonyan címezhetjük meg programunk adatterületeit. A felhasznált címzési módok kiválasztása nagyban elősegítheti a hatékony, jól olvasható és tömör programok írását.

Itt tárgyaljuk a programterületek címzésének módjait, azaz a vezérlésátadás eljárásait is. Mint látni fogjuk, a processzor igen "ravasz" indirekt ugrások, szubrutinhívások végrehajtását is lehetővé teszi. Programozás közben bátran, de körültekintően éljünk ezzel a lehetőséggel.

A processzor a címszámítás során minden esetben használ egy szegmensregisztert, amely a szegmens báziscímét tartalmazza. A címzési módok tárgyalása során a példáknál általában eltekintünk ennek figyelembevételétől, a kiszámított cím csak a szegmensregiszterhez adandó offset (virtuális cím).

Jóllehet az utasításszerkezet ismertetése tematikailag nem kötődik szorosan ehhez a témához, mégis itt tárgyaljuk, mert (mint látni fogjuk) a "címzési mód byte" szerkezetének ismerete nagymértékben elősegíti a címzési módok alapos megértését.

2.1. Az Intel 8086/8088 utasításszerkezete

Az assembler utasításai, mint ismeretes, pontosan megfelelnek a processzor gépikódú utasításainak, egy-egy assembly utasítás pontosan egy gépikódú utasításra "fordul le".

A gépikódú utasítások tulajdonképpen számkódok, amelyeket be kell tölteni a gép memóriájába, és a processzor sorban egymás után elolvassa és végrehajtja őket. Minden utasítás két részből áll: az operációs kódból (mit kell tenni) és az operandus(ok)ból (mivel). Az operandus rendszerint valamilyen adatnak a címe - az adat a memória egy ismert részén helyezkedik el, és a processzor a címe (a terület kezdő byte-jának sorszáma) alapján érheti el.

Az Intel 8088 processzor gépi utasításai különböző számú byte-ot foglalnak el. A legrövidebbek 1 byte-osak, a leghosszabb utasítás az ún. prefixumokat is beszámítva 8 byte hosszú lehet. Az utasítások általában 3-4 byte-ot foglalnak le. Mint majd látni fogjuk, egyazon utasítás is különböző hosszúságu lehet, a "szovegkornyezet" függvényében.

Az általános utasításszerkezet a következő (egy téglalap itt általában egy byte-ot jelent, az "operandus" lehet akár 4 byte-os is):

Prefixum	Op. kód	Címzési mód	Operandus

Kezdjük a közepén! Az "Op. kód" az operációs kód rövidítése, ez a "mit kell tenni". Minden utasításban szerepel.

Az "Operandus", mint látható, lehet egy, két vagy négy, különleges esetben három byte-os, esetleg teljesen el is maradhat. Ez általában egy cím vagy a címképzésben szerepet játszó érték, de lehet egy adat is.

A "Címzési mód" byte csak olyan utasításokban szerepel, amelyeknek van operandusa és még akkor sem minden esetben. Azt mutatja, hogyan kell értelmezni az operandust, azaz az operandus 0, 8 vagy 16 bitje mit is jelent (éppen ezt, vagyis a

címzési módokat fogjuk részletezni e fejezetben).
Végül a "Prefixum" alkalmazása nem kötelező, és viszonylag ritkán is használjuk. Módosítja az utasítás értelmezését.

> Megjegyzés: a prefixum előírhatja az utasítás ismételtetett végrehajtását, szegmensregiszter-átdefiniálást, és a processzor adat- és címbuszának "lock"-olását, lezárását, amely műveletnek többprocesszoros környezetben van értelme.

A címzési mód byte ismertetésére csak e fejezet végén térünk ki, mikor már ismertek lesznek a címzési módok.

2.2. A programterület címzése (vezérlésátadás)

2.2.1. IP-relatív címzés

Az utasításkódban talált 8 vagy 16 bites adatot hozzáadja (előjekiterjesztéssel) az IP pillanatnyi értékéhez.

> Megjegyzés: minden feltételes vezérlésátadás ilyen IP-relatív, 8 bites eltolással - ebből következik, hogy feltételes vezérlésátadás csak közeli címekre hajtható végre. (Bővebben lásd a 3. fejezetben.)
>
> Természetesen a "megcélzott" utasítás távolságát nem nekünk kell kiszámítani; mi ilyenkor is egy címkére hivatkozunk, és a fordító számítja ki a kívánt távolságot.

2.2.2. Direkt utasításcímzés

Az utasításkódban szereplő 16 vagy 32 bites cím töltődik be az IP-be (szegmensen belüli direkt ugrás vagy szubrutinhívás), vagy IP-be és CS-be (más szegmensbe történő direkt ugrás vagy szubrutinhívás).

2.2.3. Indirekt utasításcímzés

Az utasításkódban megadott cím (bármilyen adatcímzési eljárással) egy egyszavas vagy duplaszavas blokk címe. Ez a blokk tartalmazza a kívánt címet. Egy szó (16 bit) esetén szegmensen belüli indirekt ugrás vagy szubrutinhívás, duplaszavas (32 bites) blokk esetén szegmensközi indirekt ugrás vagy szubrutinhívás.

Szegmensközi vezérlésátadás esetén mindig az alacsonyabb című szó értéke kerül IP-be, míg a magasabb című írja felül a CS szegmensregisztert.

Példák: a példák során (mivel nem az utasításokat, hanem a címzési módokat szeretnénk szemléltetni) mindig a JMP utasítást használjuk, amely feltétel nélküli ugrást ír elő.

1. Igen rövid ugrás:

```
LABEL:
        NOP
        NOP             ;üres utasítások, 1 byte
        JMP     LABEL   ;A távolság -2 byte
```

Ennek az ugrási utasításnak a kódja:

EB FC

ahol "EB" a relatív ugrás kódja, "FC" pedig -4, mivel az IP, miközben felolvasta a JMP utasítást, már megnövekedett 2-vel!

2. Hosszabb ugrás:

```
        JMP     LABEL   ;Ez az utasítás 100H-n van

LABEL:                  ;Ez 900H-ra esik
```

Ennek kódja:

E9 FD 07

ahol "E9" az operációs kód, "FD 07", helyesen olvasva "07FD", a relatív távolság: 800H-ból levonva az utasítás hosszát.

3. Direkt ugrás más szegmensre:

```
JMP     FAR  PTR  LABEL
```

Tegyük fel, hogy LABEL egy 36400H abszolút címen kezdődő szegmens 3F5H virtuális címén helyezkedik el. Ekkor az utasítás kódja:

EA F5 03 40 36

ahol "EA" az operációs kód, "03F5" a virtuális cím, amely IP-be kerül, "3640" a szegmens címe (ez kerül majd CS-be).
További példák 2.4.-ben találhatók.

2.3. Az adatterület címzése

Az adatcímzési eljárások szemléltetésére a MOV utasítást fogjuk használni a következő formában (természetesen más utasítások, más regiszterek éppen így használhatók):

```
MOV     AX, op
```

amely utasítás az "op" (operandus) értékét viszi be az akkumulátorba. Az "op" fogja szemléltetni a címzési módot; minden esetben megjegyzésként adunk egy példát is.

Az adatterület címzésére 6 (+1) eljárás használható, melyek közül 4 (felületesen szemlélve) egyazon eljárás különböző esetének tekinthető.

2.3.1. Kódba épitett adat (immediate addressing)

Ez esetben az utasitásban szereplő 8 (16) bites mennyiség maga az adat.

Példa:

```
MOV     AX, 07FFH
```

utasitás hatására az AX akkumulátorba a 07FF hexadecimális szám kerül. Az utasításkódba ekkor a 07FF 16 bites mennyiséget "épitettük be". Ebben az esetben az utasitás kódja a következő:

B8 FF 07

azaz az utasitás 3 byte-ot foglal le; ebből a "B8" az utasitás operációs kódja, "FF" a beépitett adat alsó, "07" pedig a felső byte-ja.

Ezt a címzési módot a továbbiakban "beépitett adat címzése" névvel illetjük.

2.3.2. Direkt memóriacímzés

Az utasitásban elhelyezett 16 bites szám az adat virtuális címe. Ezt a 16 bitet adja a processzor a kiválasztott szegmensregiszterhez, s az ott talált adatot használja fel.

Példa:

```
MOV     AX, [07FFH]
```

Tegyük fel, hogy DS = 1376, ekkor az AX-be a

13760H+7FFH = 13F5FH

címen található adat kerül, azaz a 1376-on kezdődő szegmens 7FF. és 800. címén fekvő szó tartalma.

Ez a szintaxis nem egyezik meg az assembler szintaxisával; a DEBUG program várja így az utasítást. Az assembler számára szimbolikusan kell megadni az operandust.
Igy a következő lenne:

```
ADAT    DW      34F2H   ;Szavas adat
                        ;(Lásd 4. fejezet)
        MOV     AX, ADAT
```

Erre az utasításra 34F2 kerülne be AX-be, tehát az "ADAT" szimbolikus nevű változó (melynek szavasnak kell lennie) tartalma, anélkül, hogy az ADAT konkrét címét ismernünk kellene.
Feltéve, hogy "ADAT" éppen a 7FF címre esik a szegmensen belül, az utasítás gépikódja:

```
A1 FF 07
```

tehát 3 byte hosszú utasítás, az első byte az operációs kód, a többi a cím.
Ez a beépitett adat címzésétől csak az operációs kódban különbözik.

A felhasznált szegmensregiszter alapértelmezésben a DS; külön prefixum segitségével bármelyik szegmesregisztert használhatjuk.

2.3.3. Indexelt címzés

A processzor az utasításban elhelyezett 8 (16) bites számot hozzáadja a kiválasztott indexregiszter tartalmához, az igy keletkezett 16 bites címet (8 bites offset esetén előjelkiterjesztést végez 16 bitre, 16 bit esetén az esetleges túlcsordulást eldobja) adja a kiválasztott szegmensregiszter tartalmához - ez az adat fizikai címe:

```
        MOV     AX, 07FH[ DI ]
```

Látható, hogy a virtuális cím kialakításában két érték játszik szerepet. A szokásos elnevezések szerint a használt indexregiszter tartalma a "báziscím", az indexregiszter elé irt szám pedig az "eltolás", vagy offset, vagy index.

Példák: e példák során (és a továbbiakban is) eltekintünk a szegmensregiszter részletezésétől, mert csak a virtuális cím képzésére kell figyelnünk; a fizikai cím mindig valamelyik szegmensregiszter felhasználásával keletkezik.

1. Legyen DI tartalma 12E0, az eltolás pedig 7F. Ekkor a

```
MOV     AX, 7FH[ DI ]
```

utasítás a DS által címzett szegmensben a 12E0-tól számított 7F. és 80. byte-ot olvassa be AX-be, azaz a 135F és 1360 byte-ok tartalmát. Gépikódja:

8B 45 7F

ahol 8B az operációs kód, 45 az ún. címzési mód byte és 7F az eltolás (csak egy byte !!).

2. Legyen most az eltolás 7FF, amely nem fér el egy byte-on. Ekkor a gépikód:

8B 85 FF 07

ahol 8B és 85 az operációs kód és címzési mód byte, FF és 07pedig a 16 bites eltolás byte-jai.

3. Tegyük fel, hogy DI = FF00, az eltolás 7FFH. Ekkor a keletkezett cím 06FF (mert a processzor eldobja a túlcsordulást), s ez a virtuális cím.

4. Ha az eltolás negatív, akkor a címszámitás a következőképpen folyik:

```
MOV     AX, 0F8H[ DI ]
```

Legyen DI tartalma 3265H. Ekkor

3265H + F8H -> 3265H + FFF8H = (1) 325DH
előjelkiterjesztés !

azaz mintha 8-at kivontunk volna a címből, mert a zárójelbe tett túlcsordulás elvész.

A felhasznált szegmensregiszter alapértelmezésben a DS; külön prefixum segítségével bármelyik szegmesregisztert használhatjuk.

Megjegyzés: az indexelt címzésben a két szereplő érték összeadódik, s az összeadandók egyenrangúak - ebből következik, hogy kétféle fontos dologra használhatjuk fel ezt a címzési módot.
Az első esetben az indexelt címzést adatstruktúra címzésére használjuk (lásd 4. fejezet). Ekkor a struktúra címe helyezkedik el az indexregiszterben és az eltolással adjuk meg, hogy a struktúra melyik elemére hivatkozunk (ti. a struktúra kezdetétől számított távolságot adjuk meg az eltolással).
A második eset a "buffercímzés" esete. A buffert felfoghatjuk úgy is, mint homogén (egyenrangú) elemek folytonos halmazát a memóriában. Ekkor a buffer kezdőcíme lesz az eltolás (és így átveszi a "bázis" szerepét), a bufferen belüli címet az indexregiszter fogja tartalmazni, hiszen könnyen módosítható.

2.3.4. Regiszter-indirekt címzés (implied addressing)

Ez a címzési mód úgy tekinthető, mint az indexelt címzésnek az az esete, mikor az offset 0, azaz nincs eltolás az indexregiszterhez képest. A kiválasztott indexregiszterben levő 16 bit a virtuális cím.

A felhasznált regiszter lehet a BX regiszter is, az is használható a regiszter-indirekt címzésben címregiszterként.

Prefixum segítségével a címzéshez más szegmensregisztert is használhatunk.

Példák:

```
MOV     AX, [ SI ]
MOV     AX, [ BX ]
```

Ekkor az adatszegmensben az SI illetve a BX által címzett memóriaszó kerül be az akkumulátorba.

```
MOV     AX, CS:[ SI ]
MOV     AX, ES:[ BX ]
MOV     AX, SS:[ DI ]
```

Itt példát láttunk arra, hogyan kell más szegmensregisztert kiválasztani a címzésre.

A gépikódok rendre:

```
8B 04                   MOV     AX, [ SI ]
8B 07                   MOV     AX, [ BX ]

2E 8B 04                MOV     AX, CS:[ SI ]
36 8B 07                MOV     AX, ES:[ BX ]
26 8B 05                MOV     AX, SS:[ DI ]
```

Mint látható, ebben az esetben a gépikódból elmarad az eltolás, így rövidebb lesz az utasítás - csak egy címzési mód byte szabja meg, melyik regisztert használjuk.

2.3.5. Bázisrelatív címzés

Ez a címzési mód lényegében azonos az indexelt direkt címzéssel, de indexregiszter helyett a BX regisztert használjuk. Bármely fentebb leírt offset használható: 8 bites offset esetén előjelkiterjesztés 16 bitre, 16 bit esetén a túlcsordulás elvész.

E címzési módnak az ad külön jelentőséget, hogy kombinálható az indexelt címzéssel. Ekkor az offset értéke összegződik az indexregiszter, valamint a BX regiszter tartalmával, s ez a virtuális cím.

A felhasznált szegmensregiszter alapértelmezésben a DS; külön prefixum segítségével bármelyik szegmesregisztert használhatjuk.

Példák:

```
MOV     AX, 057FH[ BX ]
```

nagyon hasonlít egy indexelt címzéshez;

```
MOV     AX, [ BX + 057FH ]
```

az előző utasítással megegyező.

A következő példa világít rá a bázisrelatív címzés leglényegesebb sajátosságára:

```
MOV     AX, 057FH[ SI ][ BX ]
```

amelynek olvastán a processzor összegzi BX és SI tartalmát, majd hozzáadja a 057FH eltolást (a túlcsordulásokat eldobva) és az így kapott számot használja virtuális címként. Ugyanez az utasítás így is írható:

```
MOV     AX, [ 057FH + SI + BX ]
```

vagy pedig

```
MOV     AX, [ 057FH + SI ][ BX ]
```

ahogyan az adott helyen a leglogikusabbnak látszik.

Ez a címzési mód, mint látható, három tényezőből állítja elő a virtuális címet. Ezek közül egy (az offset) fix, a másik kettő csak "futásidőben dől el", hiszen a bázisregiszter és az indexregiszter tartalma csak futás közben alakul ki. Célszerűen használható többdimenziós tömbök, struktúratömbök vagy bufferek tömbjeinek címzéséhez - a címképzés két futás közben változó eleme két "dimenzió" kényelmes használatát teszi lehetővé. Egyedül az jelenthet némi kényelmetlenséget, hogy kétdimenziós címzés esetén az egyik regisztert nem egyesével kell

módosítanunk, hanem figyelembe kell vennünk a "sor" (az egyik dimenzió sugallta memóriaterület) hosszát.

Megjegyzés: bázisregiszterként dolgozhatunk a BX és a BP regiszterrel is. Ha a BX-et használjuk, az alapértelmezett szegmensregiszter a DS lesz; mint fentebb olvasható, prefixum segítségével más szegmensregisztert is használhatunk. A BP esetén az alapértelmezett szegmensregiszter az SS. Másik szegmensregiszterre prefixum használatával térhetünk át. A BP-t azonban csak különlegesen indokolt esetben használjuk másra, mint amire való (lásd a következő alfejezetet).

Fontos emlékeznünk arra, hogy bármely indexregiszter használata kombinálható bármely bázisregiszterrel, de nem használhatunk együtt két bázis- vagy két indexregisztert.

2.3.6. A stack-terület címzése

A stack-terület címzése során a bázisregiszteres címzési mód használható, de mindig a BP regiszter szolgál bázisként. Használható valamelyik indexregiszter is (lásd bázisrelatív címzés). Az alapértelemzett szegmensregiszter az SS. Egy prefixum segítségével más szegmensregiszter is használható. Azonban a BP-t (mivel "hardware-módon" kötődik a stack-hez) csak akkor használjuk másra, ha a stack-et nem akarjuk sem paraméterátadásra, sem ideiglenes változók tárolására használni.

Példák:

Tegyük fel, hogy két egyszavas paramétert adtunk át egy szubrutinnak a stack-en, ráadásul a rutin egy belső változóját szintén a stack-en helyezi el. Ekkor a rutin célszerűen menti a BP-t, majd SP-t átmásolja a BP-be. Rövid hívást feltételezve a paraméterek távolsága a BP által ekkor címzett byte-tól 4 (vagy hosszú hívás esetén, mikor a visszatérési cím két

szó, 6). A változó offsetje, amelynek kezdeti értékét lenyomjuk a stack-re, -2:

```
PUSH    BP              ;Régi BP mentése
MOV     BP, SP          ;BP előkészítés
MOV     AX, 3465H       ;Kezdeti érték AX-be
PUSH    AX              ;Belső változó

MOV     AX, 4[ BP ]     ;Első par. címzése
MOV     BX, 6[ BP ]     ;Második par. "
MOV     CX, -2[ BP ]    ;Belső változó "

POP     BP              ;Belső vált. eldob
POP     BP              ;BP visszaállítása
```

Látható, hogy ilyenkor a BP "szent és sérthetetlen"; amennyiben menet közben másra is akarjuk használni, akkor mentés és visszaállítás szükséges, ez azonban nehezen olvashatóvá teszi a programot és óriási kockázatot jelent. Hiszen ha csak egyszer nem állítjuk vissza a regiszter tartalmát, amikor szükséges volna, akkor a "csillagos égre" címzünk a paraméterek vagy a belső változó helyett.

2.3.7. Regisztercímzés

Ez a címzési mód félig megjegyzésként szerepel, mert nem "hivatalos" címzési mód.

Azt jelenti, hogy az utasítás egyetlen vagy mindkét operandusa valamelyik regiszter. Ilyen esetben a címzett adat(ok) már valamelyik regiszterben van(nak).

Példák:

```
MOV     AX, BX          ;BX tartalma AX-be

MOV     DS, AX          ;DS felülírása

MOV     DL, DH
MOV     CL, BH
```

Figyelem: a regiszterek közül nem mindegyik írható vagy olvasható közvetlenül! A négy általános regiszter, a két indexregiszter és a bázisregiszter korlátlanul használható.

Az SP olvasása gyakori lépés:

```
        MOV     BP, SP          ;BP előkészítése
```

írása vagy közvetlen utasítással történő módosítása lehetséges:

```
        MOV     SP, 300     vagy

        ADD     SP, 4           ;Elfelejtjük a
                                ;paramétereket
```

de igen óvatosan végzendő.

Az IP írása csak vezérlésátadó utasításokkal lehetséges, közvetlenül olvasni nem lehet.

Végül a szegmensregiszterek közül a DS, ES és SS korlátlanul írható-olvasható (de nem írható felül beépített adattal). Lássunk egy példát a DS előkészítésére:

Rossz:

```
        MOV     DS, DATA_SEG    ;Szintaktikus hiba
```

Helyes:

```
        MOV     AX, DATA_SEG    ;Beépített adat
        MOV     DS, AX
```

Az SS írása különös óvatosságot igényel, ha felülírjuk, elvész a stack!

Megjegyezzük még, hogy szegmensregiszter (sajnos) nem szerepelhet aritmetikai utasítások operandusaiként. Ez is a processzor "önvédelméhez" tartozik.

A CS írása (az IP-hez hasonlóan) vezérlésátadással lehetséges csak.

A DEBUG szó nélkül elfogadja a

```
MOV     CS, AX
```

utasítást, azonban ennek végrehajtása esetén "lefagy" az egész rendszer.
Az így előállított kód futásra képes lehetne ugyan, de a következmények életveszélyesek. Gondoljuk csak el, hogy az IP tartalmaz valamilyen értéket; ha CS-t felülírjuk, akkor az így címzett szegmens IP-edik pozíciójára kerül a vezérlés (tehát nem tudjuk, hogy melyik memóriapozícióra). Ennek következményei pedig beláthatatlanok.

2.3.8. Összefoglalás

A programozó szempontjából lényegében háromféle címzési mód létezik: a regisztercímzés (7.), a direkt adatmegadás (1.), és a bázisregiszteres indexelt memóriacímzés, melynek elfajuló esetei adják a 2-6 pontban részletezett címzési módokat.

Fontos megszorítás

Az Intel 8086/8088 ún. "egycímes" processzor, ami azt jelenti, hogy egy utasítás a legjobb akarattal is csak egy memóriapozíciót tud megcímezni. Tehát a kétoperandusú utasítások (mint MOV, ADD, stb.) egyik operandusának minden esetben regiszternek kell lennie, csak a másik lehet a memória valamelyik byte-ja (szava).

2.4. A címzési módok az assemblerben

Mint már a példákban is láttuk, a különböző címzése módok igen természetesen, célszerű formában írhatók le az assembler fordító számára. Vegyük először a programterület címzését!

IP-relatív és direkt címzés:

```
        JMP     SHORT   cím     ;A SHORT operátor jelzi,
                                ;hogy relatív (rövid) ugrás
        JMP     cím
```

A fordító, hacsak teheti, IP-relatív ugrást alkalmaz a direkt ugrás helyett. Ha az áthidalandó távolság nagyobb, mint 127 byte (tehát nem érhető el egy egybyte-nyi offset-tel), akkor direkt ugrás keletkezik. A "SHORT" kulcsszó kiemeli a programozó szándékát a rövid ugrásra. Használata esetén a fordító hibaüzenetet küld, amennyiben az ugrás 127 byte-nyi távolságnál nagyobb.

Felhívjuk az Olvasó figyelmét, hogy a feltételes vezérlésátadási utasítások mindegyike IP-relatív, és ezt megváltoztatni nem lehet.

Direkt címzés:

```
        JMP     cím             ;Direkt ugrás a címre
                                ;( szegmensen belül! )

        JMP     FAR PTR cím     ;Direkt ugrás a címre
                                ;(4 byte-os beépített érték)
                                ;Lásd 4. fejezet !
```

Indirekt címzés:

```
        JMP     BX              ;Ugrás a BX által címzett
                                ;utasításra
        JMP     [ BX ]          ;Ugrás a DS:BX-el megadott
                                ;memóriaszóval címzett
                                ;utasításra
```

```
        JMP     DWORD PTR [BX][SI]
                                ;Ugrás a BX + SI címen ta-
                                ;lálható 4 byte-os blokk ál-
                                ;tal címzett távoli pontra
                                ;Lásd 4. fejezet !
```

Adatcímzési módok leírása:

Beépített adat elérése (immediate címzés):

```
        MOV     AL,7FH          ;AL-be 7F hexa
                                ;8 bit AL miatt
        MOV     DX,7FH          ;DX-be 7F hexa
                                ;16 bit DX miatt

KONST   EQU     0FCH            ;Konstans definíció

        MOV     DI,KONST        ;Immediate a definíció miatt
                                ;EQU - lásd 4. fejezet !
```

Direkt memóriacímzés:

```
BYTE    DB      0               ;Adatdefiníció
                                ;DB - lásd 4. fejezet
        MOV     AH,BYTE         ;Memóriacímzés BYTE
                                ;definíciója miatt
                                ;Nem következetes jelölés!
```

Egyéb címzési módok:

```
        offset[ SI ][ BX ]
        offset[ DI ][ BX ]
        offset[ BP ]
        stb.
```

ahol bármely összetevő szabadon elhagyható. A "["-t és a "]"-t mindig ki kell írni (itt nem opcionális értéket jelent, hanem címek összeadását). A fenti címeket így is írhatnánk:

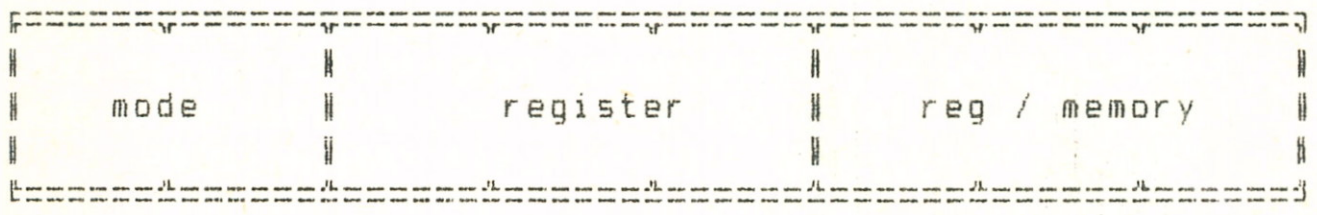
```
        [ offset + SI + BX ]
        [ offset + DI + BX ]
        [ offset + BP ]
```

2.5. A címzési mód byte

A címzési mód byte általában kiegészíti az utasításkódot. Nem minden esetben használatos, mert egyfelől vannak olyan utasítások, melyeknek egyáltalán nincs operandusuk, másfelől pedig olyanok, melyekhez nem szükséges a címzési mód byte (pl. ha egy beépített adatot töltünk be valamelyik regiszterbe).

A címzési mód byte ismerete nem szükséges a programozáshoz; azért vázolom fel mégis, mert elmélyült tanulmányozása közelebb vihet a címzési módok alapos megértéséhez.

Egy-egy kocka egy-egy bitet jelent.



ahol

mode — a második két hárombites csoport értelmezését szabja meg (2 bit);

register — azt szabja meg (a mode-tól függően), hogy melyik regisztert kell használni mint operandust (3 bit)

reg / memory — azt szabja meg (a mode-tól függően), hogy melyik címzési módot használjuk az operandus elérésére (3 bit).

A "mode" jelentése:

00 — "reg/memory" egy regiszter- vagy memóriaoperandust jelent, nincs további (ún. displacement) byte, vagyis az adat vagy címe valamelyik regiszterben van

01	reg/memory egy regiszter- vagy memóriaoperandust jelent, 1 displacement byte-al, vagyis az adat címe 8 bites eltolással képződik
10	reg/memory egy regiszter- vagy memóriaoperandust jelent, 2 displacement byte-al, vagyis az adat címe be van épitve a kódba, vagy 16 bites eltolással képződik
11	reg/memory egy második regiszter-operandust jelent az alábbi táblázat szerint.

Megjegyzés: a "displacement" kifejezés magyarul nehezen adható vissza, mert angolul is eléggé elvonatkoztatott értelemben használják e helyen. Egyszerűen a kódba épitett 8 vagy 16 bites adatot jelenti, amely lehet virtuális cím vagy eltolás.

A "register" jelentése:

	byte-os	szavas
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

utasítás esetén

Az, hogy az utasítás szót vagy byte-ot mozgat, az operációs kódba van beépítve, ez az operációs kód legalsó bitje.

Végül a "reg/mem" jelentése:

	Mode = 00	Mode = 01	Mode = 10 esetén
000	BX + SI	BX + SI + 8 bit	BX + SI + 16 bit
001	BX + DI	BX + DI + 8 bit	BX + DI + 16 bit
010	BP + SI	BP + SI + 8 bit	BP + SI + 16 bit
011	BP + DI	BP + DI + 8 bit	BP + DI + 16 bit
100	SI	SI + 8 bit	SI + 16 bit
101	DI	DI + 8 bit	DI + 16 bit
110	Direkt cím		
111	BX	BX + 8 bit	BX + 16 bit

ahol "8 bit" és "16 bit" az ún displacement (index, eltolás) bitekben kifejezett hosszúságát jelenti.

A "mode = 11" eset megegyezik a "register" esettel - az utolsó hárombites csoport egy regisztert határoz meg.

Az adatmozgás iránya az operációs kódból derül ki. Vizsgáljuk meg például az alábbi két utasítás címzési mód byte-ját!

```
MOV     CX, 325[ SI ][ BX ]
MOV     325[ SI ][ BX ], CX
```

Az utasítások címzési mód byte-ja megegyezik (értéke 88H), gépikódjuk jelzi, hogy a memóriából CX-be, vagy a CX-ből a memóriába visszük-e az adatot.

> Megjegyzés: beépitett adat címzése esetén nincs címzési mód byte; e címzési módra minden egyes utasítás esetén külön gépikód utal.
>
> Megjegyezzük még, hogy azért nem magyar kifejezéseket használunk a címzési mód byte egyes mezőire, mert e könyv(ek)ben többé soha, sehol nem hivatkozunk egyikre sem; lefordításuk felesleges volna. Az angol elnevezések használata viszont segithet az Olvasónak az eredeti szakirodalom olvasásában.

Szegmensregiszter-választás

Ha a szegmensregiszterek használata során el akarunk térni az automatikus kiválasztástól (a gyakorlatban csaknem minden adatcímzéssel a DS regisztert használjuk), akkor ezt külön jelezni kell az assemblernek:

```
MOV     AX, ES:[BX]
```

Ez az utasítás az ES szegmensregiszter által címzett szegmens BX-ben levő virtuális című szavát olvassa be az AX-be.

A szegmenszregiszter átdefiniálása mindig külön byte segítségével történik. Ez a byte az utasítás operációs kódja előtt helyezkedik el, és felépítése a következő:

0	0	1	r	r	1	1	0

ahol "rr" jelentése a következő:

00 ES regiszter kiválasztása

01 CS regiszter kiválasztása

10 SS regiszter kiválasztása

11 DS regiszter kiválasztása

Megjegyzés: ez utóbbi prefixumot használjuk, ha a BP regisztert az adatszegmensben akarjuk bázisregiszteres címzésre használni.

3. fejezet

Az Intel 8086/8088 utasításkészlete

A 3. fejezet tartalomjegyzéke

3.0. Bevezetés

Az utasításokat röviden, utasításcsoportokba szedve tekintjük át. A csoportokon belül a sorrendet az utasítások fontossága, alkalmazásuk gyakorisága döntötte el (bár ez eléggé szubjektív dolog). Csak egy külön alfejezetben foglaljuk össze, hogy melyik utasítás hogyan állítja a STATUS flag-jeit. Az utasítások ismertetése során elégedjünk meg annyival, hogy a flag-ek működése teljesen logikus, és ebben nincs lényeges eltérés a többi gépikódhoz viszonyítva. (Jellemző, hogy az adatmozgató utasítások nem állítják a flag-eket - számoljunk ennek előnyeivel és hátrányaival.)

Nem adjuk meg az utasítások gépikódját, és különböző variánsaikat gyakran "egy kalap alá vesszük". A MASM macroassembler ugyanis a felhasználni kívánt variánst szintén teljesen logikusan a megadott operandusokból határozza meg - ezzel nem kell törődnünk.

> Megjegyzés: tekintsük például a MOV utasítást. Ennek is igen sok változata van. Lehet szavas, byte-os; használhatunk beépített adatot, direkt vagy egyéb címzést, az adatmozgás irányulhat regiszterből memóriába és viszont - minden esetben más az utasítás operációs kódja, más címzési mód-byte tartozik hozzá (a lényegileg eltérő operációs kódok száma hét, ahol figyelmen kívül hagytuk például azt, hogy ugyanaz az utasítás szavas és byte-os is lehet), hosszúsága 2 byte-tól (prefixum nélkül) 6 byte-ig terjedhet, assembly nyelvi megjelenése mégis egyforma.

Az utasítások részletezése során nem törekedtem a teljes precizitásra. Igyekeztem a lényegre összpontosítani, hogy az Olvasó ne vesszen el a részletekben, hanem legyen lehetősége áttekinteni az utasításkészletet. Ha a részletekre van szükségünk, akkor olvassuk el az adott utasítást az LSI-féle könyvben vagy az eredeti leírásban. Hivatalosnak csak ez utóbbi tekinthető, és nyugodtan feltehetjük, hogy ez gyakorlatilag hibátlan. A legbiztosabb azonban az, ha behívjuk a DEBUG programot és a gyakorlatban kipróbáljuk az utasítást (ne felejtsük el, hogy van némi eltérés a fordító és a DEBUG által

megkövetelt szintaxis között; ezekre az utasítások ismertetése során utalni fogunk).

A fejezetben először mindig az utasítás assembler formáját olvashatjuk, majd hatásának vázlatos leírását.

Mint közismert, az assemblerek a gépikódok számértéke helyett egy ún. "mnemonic"-ot, emlékeztető szócskát használnak. A továbbiakban is ezt a kifejezést használjuk, igazán találó magyar szó híján. (Nevezhetnénk például mnemonikus jelképnek, avagy emlékeztető jelképnek.) A mnemonic általában 3-4 betűs szimbólum, az utasítás angol "leírásának" többé-kevésbé értelmes rövidítése (például az AAA - "Ascii Adjust after an Addition", az akkumulátor tartalmának ASCII-konverziója összeadás után).

Sokszor fordul majd elő olyan eset, hogy több mnemonic-hoz tartozik egy magyarázat. Ekkor a különböző mnemonic-ok azonos utasítást jelentenek, csak a tisztább kódolás kedvéért van több különböző mnemonic. Például:

```
        JNZ     cimke
        JNE     cimke
```

(Jump on NonZero és Jump on Not Equal) azonosak, de az első inkább logikai jellegű, a második inkább numerikus. Az elsőt ciklusszervezésben használjuk inkább ("csökkentjük DX-et és ha még nem 0, akkor folytatjuk ..."), a másodikat pedig számok összehasonlítása során ("összehasonlítjuk a-t b-vel és ha nem egyenlőek ...").

Az alkalmazott rövidítések:

[...]	a ... tartalmát jelenti
op	tetszőlegesen választott memóriaoperandus vagy regiszter
reg	regiszter
mem	tetszőleges memóriaoperandus

3.1. Adatmozgatási utasítások

MOV op1, op2

"MOVe" - az "op2"-t az "op1"-be írja, felülírva az eredeti tartalmat.
Változatai: 8 és 16 bites adatok mozgatása, tetszőleges címzési mód segítségével.

XCHG op1, op2

"eXCHanGe" - az "op1" és "op2" értékeinek felcserélése.
Változatai: 8 és 16 bites adatok mozgatása, minden címzési mód felhasználásával - kivéve természetesen a beépített adat címzését.

Megjegyzés: ez egy igen hasznos utasítás, lehetővé teszi változók értékének cseréjét segédváltozók felhasználása nélkül.

XLAT

"trans(X)LATe" - AL <-- [[AL] + [BX]] ; a BX által címzett, legfeljebb 256 byte-os adattáblázat AL-edik elemét kiolvassa és beírja AL-be - ez egy kódkonverziót végző utasítás.

Megjegyzés: az XLAT egy jellegzetes felhasználása a következő:

```
        (adatszegmensben)
HEXTB   DB      '0123456789ABCDEF
                        ;Hexa számjegyek
        (kódszegmensben)
HEXCONV PROC
        PUSH    BX          BX mentése
        AND     AL, 0FH     Biztos ami biztos
        MOV     BX, OFFSET HEXTB
        XLAT
        POP     BX
        RET
HEXCONV ENDP
```

Ez a kis program AL-ben egy 0 és 15 közé eső bináris számot kap bemenetként, ami közvetlenül konvertálható egy hexadecimális számjeggyé. Letettük a szóbajöhető hexa számjegyek ASCII-kódját az adatszegmensbe (az ismeretlen jelöléseket lásd a 4. fejezetben), majd meghivtuk a szubrutint. A rutin először (biztos ami biztos) levágja a számot úgy, hogy biztosan ne legyen nagyobb 15-nél, betölti a HEXTB tábla kezdőcímét a BX-be, majd végrehajtja az XLAT utasítást. Az XLAT összeadja a BX és az AL tartalmát, az összeget címként használva beolvas egy byte-ot AL-be. Legyen például AL tartalma a bemenetkor 12 (dec.). Ekkor a cím BX tartalmánál éppen 12-vel nagyobb lesz (nem kell tudni, HEXTB pontosan hol van), és ezen a byte-on pontosan a 'C' betű kódja van, ez kerül be AL-be - éppen a 12 decimális érték hexadecimális megfelelője.

LDS reg, mem

"Load pointer using DS" - a "mem" (4 byte-os operandus) első és második byte-ja kerül a megcímzett regiszterbe, harmadik-negyedik byte-ja pedig DS-be.

LES reg, mem

"Load pointer using ES" - a "mem" (4 byte-os operandus) első és második byte-ja kerül a megcímzett regiszterbe, harmadik és negyedik byte-ja pedig ES-be.

Megjegyzés: az LDS és LES utasítások egy mozdulattal töltik be egy más adatszegmensben levő változó címét az egyik szegmensregiszterbe (szegmens) és az egyik (célszerűen címzésre alkalmas) regiszterbe. Ezt az utasítást követően azonnal címezhető a kívánt memóriapozíció.

LEA reg, mem

"Load Effective Address" - a "mem" virtuális címét (szegmensen belüli offset-jét) tölti "reg"-be.

Megjegyzés: ez az utasítás látszólag felesleges, a gyakorlatban a címek betöltésére az OFFSET operátort használjuk (4. fejezet). Mégis nagyon fontos ez az utasítás, mert a "mem" operandus tetszőleges címzési móddal állhat. Példa:

LEA DI, 322[BX][DI]

amelynek végrehajtása után DI-ben azt a virtuális címet találjuk, amely megegyezik a 322[BX][DI] címzéssel elérhető pozíció virtuális címével. Ez tehát egy "run-time", azaz futás közben elvégzett címképzés, és (mint a fenti példa mutatja) eredménye lehet regisztertartalmak függvénye!

PUSH op

"PUSH" - a két byte-os operandust stack-re írja (SP 2-vel csökken, az általa most címzett szóra másolódik az "op").

PUSHF

"PUSH Flags" - a STATUS stack-re íródik. Ez az utasítás a flag-ek mentésére szolgál.

POP op

"POP" - "op" (két byte-os operandus) felülíródik a stack tetején levő szóval (az SP által címzett szó átmásolódik, majd SP 2-vel megnő).

POPF

"POP Flags" - a stack tetején levő szó felülírja a STATUS-t. Ezzel az utasítással tudjuk a flag-ek tartalmát helyreállítani (vagy pl. a TRAP flag-et bebillenteni).

Megjegyzés: ezen utasítások fő haszna az, hogy a szubrutinok (beleértve az interrupt-rutinokat is) az általuk használt regisztereket elmenthetik, majd futásuk befejezésekor visszaállíthatják anélkül, hogy erre statikus memóriaterületet kellene igénybevenniük.
Használatuk során nagyon ügyeljünk arra, hogy minden PUSH-nak meglegyen a POP "párja", nehogy kevesebb (vagy több) elemet vegyünk el a stack-ről, mint amennyit letettünk.
Arra is vigyázzunk, nehogy rossz sorrendben állítsuk helyre a mentett értékeket! Ha regisztertartalmakat mentünk későbbi helyreállítás céljából, akkor a PUSH-ok fordított sorrendjében kell következniük a POP-oknak. Javaslom, hogy szokjunk meg egy regiszter-sorrendet (pl. AX, BX, CX, DX, SI, DI, STATUS), és a mentés során mindig ehhez tartsuk magunkat.
Ha programunk "megőrül" és látszólag értelmetlenül ugrál a szubrutinok között, mindig vizsgáljuk át a PUSH és POP utasításokat - majdnem biztos, hogy a stack kavarodik össze.
A PUSH-POP utasítások használatára példát

olvashatunk az első fejezetben, az SP ismertetése alatt.
Figyeljük meg, hogy a PUSH és POP utasítások operandusa "op" és nem "reg" - ez azt jelenti, hogy tetszőleges memóriacímzéssel elért adatot is menthetünk a stack-re, és a stack tetején levő értéket közvetlenül írhatjuk be egy memóriaszóba. Csak utasításba épített adatot nem menthetünk közvetlenül a stack-re.

LAHF

"Load into AH Flags" - a STATUS alsó 8 bitjét (az "Intel 8080" biteket) AH-ba másolja.

SAHF

"Store from AH Flags" - az AH tartalma töltődik a STATUS alsó 8 bitjére.

Megjegyzés: e két utasítás az Intel 8080-al való kompatibilitás biztosítására szolgál.
Az irodalomban olvasható, hogy az Intel 8086/8088 assembly szinten "felülről" kompatibilis az Intel 8080-al. Ez nehezen definiálható dolog, mert olyan fordítóprogramot írni, amely tetszőleges programozási nyelvet gépikódra fordít, nem lehetetlen (lásd például ADA, ami kicsit bonyolultabb a 8080 assemblynél). Ez a kompatibilitás bizonyos értelemben fennállna, ha minden 8080 utasításnak megfelelne egy 8086 utasítás, de, mint látni fogjuk, a 8080 feltételes szubrutinhívási és visszatérési utasításai nem kódolhatók egy 8086-os utasítássá. Ezzel pedig a kérdés el is dőlt, a kompatibilitás nem assembly szintű.

3.2. Aritmetikai utasítások

Addíciós, összeadási műveletek

ADD op1, op2

"ADDition" - "op1"-hez adja "op2"-t, és az eredményt "op1" helyére írja.
Változatok: 8 vagy 16 bites számokkal végzett műveletek, tetszőleges címzési móddal.

ADC op1, op2

"ADd with Carry" - "op1"-hez adja "op2"-t és a "C" (carry bit) értékét, az eredményt "op1" helyére írja.
Változatok: 8 vagy 16 bites számokkal végzett műveletek, tetszőleges címzési móddal.

INC op

"INCrement" - "op" értékét 1-el növeli.
Változatok: 8 vagy 16 bites számokkal végzett műveletek, tetszőleges címzési móddal.

Megjegyzés: nem módosítja a "C" (carry bit) értékét. Lásd még: DEC utasítás.

AAA

"Ascii Adjust after an Addition", azaz ha két ASCII-számjegykódot adtunk össze, akkor az összegből AH-ban és AL-ben két pakolatlan BCD számot hoz létre decimális átszámítással.

Megjegyzés: a BCD (Binary Coded Decimal) számábrázolás ismertetése: második kötet.

Példa: tegyük fel, hogy a '6' és '8' ASCII-számjegyeket adtuk össze, összegük pedig AL-ben van. AL = 36H + 38H = 6EH, amelynek "semmi köze" a számértékek összegéhez. Ha ezután kiadunk egy AAA utasítást, akkor AH-ba 01H, AL-be pedig 04H kerül, azaz AX-ben a 14 pakolatlan BCD számpár lesz.

DAA

"Decimal Adjust after an Addition", azaz két BCD szám összeadása után az eredményt BCD-alakra hozza.

Példa: tegyük fel, hogy a két összeadott pakolt BCD számérték 28H és 68H volt. Ekkor AL-ben persze 90H lesz, amit DAA úgy igazít ki pakolt BCD formátumra, hogy az alsó 4 bithez 6-ot ad, a felsőt érintetlenül hagyja - így lesz AL-ben 96H, ami a két BCD-szám helyes összege BCD-ben.

Kivonási műveletek

SUB op1, op2

"SUBtraction" - "op1"-ből kivonja "op2"-t, az eredményt "op1" helyére írja. A Carry bitet komplementálja.
Változatok: 8 vagy 16 bites számokkal végzett műveletek, tetszőleges címzési móddal.

SBB op1, op2

"SuBtraction with Borrow" - "op1"-ből kivonja "op2"-t és "C"-t (carry bit), az eredményt "op1" helyére írja. A keletkező Carry bitet komplementálja.
Változatok: 8 vagy 16 bites számokkal végzett műveletek, tetszőleges címzési móddal.

DEC op

"DECrement" - "op" tartalmát 1-el csökkenti.
Változatok: 8 vagy 16 bites számokkal végzett műveletek, tetszőleges címzési móddal.

Megjegyzés: nem módosítja a "C" (carry bit) értékét.
Az INC és DEC utasítások eme tulajdonsága igen finom megfontolás eredménye. Mint az 1. fejezetben olvasható, a Carry bit gyakran használatos indikátorként, azaz Igen/Nem feltételek jelzésére. Nos, ha egy ciklus belsejében nagy fáradsággal előállítottuk a kívánt értéket, majd DEC vagy INC utasítással módosítjuk a ciklusváltozót, akkor biztosan megőrződik a Carry értéke, az előállított logikai érték.
Az aritmetika szabályai szerint a Carry például akkor töltődne 1-re, ha 0-t csökkentettünk (gyakori eset, hogy nem 0-ig, hanem -1-ig akarjuk futtatni a ciklusszámlálót).

AAS

"Ascii Adjust after a Subtraction", azaz ASCII számjegykódokkal végzett kivonás után az eredményt pakolatlan BCD formára hozza (AH:AL)-ben (azaz az AX regiszterben).

Megjegyzés: ez a művelet az AAD párja, mindent fordítva végez.

DAS

"Decimal Adjust after a Subtraction", azaz BCD számokkal végzett kivonás után az eredményt BCD formára hozza.

Megjegyzés: ez a művelet a DAA párja, mindent fordítva végez.

NEG op

"NEGate" - "op" értéket negálja, azaz a -1-szeresével írja felül.

Megjegyzés: az úgynevezett kettes komplemens képzésére szolgál. Minden bit megfordul, és az eredmény eggyel nő.

Szorzási műveletek

MUL op

"MULtiplication" - AX (AL) tartalmát összeszorozza "op"-al (16, illetve 8 bites művelet). 8 bites művelet eredménye AX-be, 16 bites esetén (DX:AX)-be (felső 16 bit a DX-be) kerül. Az operandusokat előjel nélküli számoknak tekinti.
Változatok: 8 vagy 16 bites számokkal végzett műveletek, tetszőleges címzési móddal, kivéve a beépített adat címzését.

IMUL op

"Integer MULtiplication" - pontosan úgy működik, mint a MUL, de előjeles szorzást végez.

AAM

"Ascii Adjust after a Multiplication", azaz BCD számok kiigazítása szorzás után.

Példa: tegyük fel, hogy két pakolatlan BCD számot szoroztunk össze: 07H-t és 09H-t. Ekkor a szorzás után AX-ben 3FH lesz, ezt fogja AAM a 0603H értékre kiigazítani úgy, hogy elosztja 0AH-val (ami a tízes számrendszerben tíz), a maradékot AL-be, a hányadost pedig AH-ba teszi.
Vegyük észre, hogy ez az utasítás alkalmas bináris-decimális konverzió végrehajtására! Nem

használja ki ugyanis, hogy a szám hogyan keletkezett (és ezért nem nagyobb, mint 51H), bármilyen 8 bites érték módosítását elvégzi. Az AL-ben maradó szám mindig a soron következő számjegy bináris értéke, AH-t pedig AL-be másolva ismételjük a műveletet, míg végül 0-t nem kapunk hányadosként.

Osztási műveletek

DIV op

"DIVision" - AX vagy (DX:AX) tartalmát maradékosan osztja "op" értékével (8, illetve 16 bites művelet). 8 bites művelet esetén AL-be kerül a hányados, AH-ba a maradék. 16 bites művelet esetén AX-ben lesz a hányados és DX-ben a maradék.
Változatok: 8 vagy 16 bites számokkal végzett műveletek, tetszőleges címzési móddal, kivéve a beépített adat címzését.

IDIV op

"Integer Division" - pontosan úgy működik, mint a DIV, de előjeles műveletet végez.

Megjegyzés: az osztás során előfordulhat, hogy a hányados nem fér el a számára kijelölt helyen. Osszuk el például a 3D52H értéket 2H-val (16 bit és 8 bit)! Az eredmény 1EA6H, ami nem fér el 8 biten. Ez az "osztási túlcsordulás", a "divide overflow" esete, amely, mint a második kötetben olvasható, a 0. interrupt-ot aktivizálja. Az interrupt mögött egy rendszerrutin van, amely a kiváltó programot azonnal abortálja.
Ha ezt el akarjuk kerulni, akkor (1) vizsgáljuk le az értékeket, és 8 bites művelet helyett végezzünk 16 bitest; (2) irányítsuk át a 0. interrupt-ot saját rutinunkra, amely lekezeli a hibát.

AAD

"Ascii Adjust for a Division", azaz BCD-számok előkészítése osztáshoz.

Megjegyzés: ez megfordítottja AAM-nek. Ha AH-ban és AL-ben két pakolatlan BCD szám van, akkor használható: megszorozza AH-t 0AH-val, és hozzáadja AL-hez.
Vegyük észre, hogy ez az utasítás alkalmas rövid digitális stringek (pl. '35') konverziójára. Töltsük a "tizeseket" AH-ba, az "egyeseket" AL-be, majd adjuk ki az AAD utasítást. AL-ben a szám bináris megfelelőjét találjuk.

CBW

"Convert Byte to Word", AL előjelének kiterjesztése AX-re, azaz 8 bites bináris szám előkészítése előjeles osztáshoz.

CWD

"Convert Word to Doubleword", AX előjelének kiterjesztése (DX:AX)-re, azaz 16 bites bináris szám előkészítése előjeles osztáshoz.

Összehasonlítási műveletek

CMP op1, op2

"CoMPare" - összehasonlítja "op1"-et "op2"-vel, azaz úgy állítja a flag-eket, mintha "op1"-"op2" kivonást végzett volna el.
Változatok: 8 vagy 16 bites számokkal végzett műveletek, tetszőleges címzési móddal.

3.3. String-kezelő utasítások

A string-kezelő utasítások a memóriablokkokkal (stringekkel) végzett műveleteket könnyítik meg. Általános konvenció, hogy DS:SI címzi meg a kiindulási (forrás-) stringet, ES:DI pedig a cél-string-et. Bármely ilyen utasítás végrehajtása során SI és DI a "D" (direction) flag állásától függően növekszik vagy csökken 1-el byte-os, illetve 2-vel szavas művelet esetén. Ha a D flag 0-t tartalmaz, akkor növelés, ha 1-et, akkor csökkentés következik be. Ennek részletesebb ismertetése a 3.3. alfejezet végén olvasható.
Megjegyezzük még, hogy a kiinduló operandus szegmense átdefiniálható, a céloperandusé azonban nem, az kötelezően ES. Lássuk most az utasításokat!

```
MOVSB
MOVSW
```

"MOVe String Byte"/"MOVe String Word" - a (DS:SI) által címzett byte (szó) átmásolása (ES:DI) címre; az SI és DI regiszterek a fent leírt módon változnak.

A fenti utasítások szokásos alakja még

```
MOVS            destination, source
```

"MOVe String" - ahol "source" és "destination" a kiindulási és a cél-string szimbolikus, vagy valamilyen, a címzésre egyértelműen utaló formájú megadása.

Ez az alak jelentheti a byte-os vagy szavas utasítást is; a fordító az operandusok ismeretében választja ki a megfelelő gépikódot.

Tegyük fel például, hogy a két alább deklarált string egy szegmensben van, és az egyszerűség kedvéért ES és DS egyaránt ezt a szegmenst címzi.

```
SOURCE  DB      30      DUP     (' ')

DEST    DB      40      DUP     (' ')


        MOV     DI, OFFSET DEST
        MOV     SI, OFFSET SOURCE
        MOV     CX, 30          ;Source hossza
COPY_LOOP:
        MOVS    DEST, SOURCE    ;Másolás
        DEC     CX              ;Ismétlés, míg
        JNZ     COPY_LOOP       ;kész nincs
```

Ha a fenti példában "DB" helyett "DW" operátorral deklaráltuk volna a stringeket, akkor a fordító automatikusan szavas operációs kódot használna.

Felhívjuk az Olvasó figyelmét, hogy a MOVS utasítás nem készíti elő a felhasznált regisztereket. A megadott operandusok lehetővé teszik a fordító számára, hogy ellenőrizze típusukat. A másolási utasítást így is írhtatnánk:

```
        MOVS    ES:BYTE PTR [DI], DS:[SI]
```

ahol "BYTE PTR" (lásd 4. fejezet) mutatja meg, hogy byte-os műveletet akarunk végezni, nem szavasat. Szavas művelet esetén természetesen a "WORD PTR" operátort kellene használni.
Véleményem szerint nem helyes a MOVS utasítás használata; sokkal jobban szeretem, ha expliciten kiírjuk, melyik formát kívánjuk alkalmazni. Egy memóriablokk kezelése éppen elég fontos feladat ahhoz, hogy ne hagyatkozzunk automatikus kódolásra. (A szerző.)

```
        CMPSB
        CMPSW
```

"CoMPare String Byte"/"CoMPare String Word" - a (DS:SI) által címzett byte (szó) összehasonlítása az (ES:DI) által címzettel; az SI és DI regiszterek a fent leírt módon változnak.
Az utasítás után a flag-ek ugyanúgy állnak, ahogy a CMP utasítás állítaná őket.

Ezen utasítások általános alakja a

CMPS dest, source

("CoMPare String") utasítás, amely ugyanazt a szerepet játssza a CMPSB/CMPSW utasításokhoz képest, mint a MOVS a MOVSB/MOVSW-hez képest.

LODSB
LODSW

"LOaD String Byte"/"LOaD String Word" - AL-be (AX-be) tölti a (DS:SI) által címzett byte-ot (szót); az SI regiszter a fent leírt módon változik.
Ezen utasítások általános alakja a

LODS source

("LOad String") utasítás, amely ugyanazt a szerepet játssza a LODB/LODSW utasításokhoz képest, mint a MOVS a MOVSB/MOVSW-hez képest.

STOSB
STOSW

"STOre String Byte"/"STOre string Word" - AL (AX) tartalmát az (ES:DI) címre írja ki; a DI regiszter a fent leírt módon változik.
Ezen utasítások általános alakja a

STOS dest

("STOre String") utasítás, amely ugyanazt a szerepet játssza a STOSB/STOSW utasításokhoz képest, mint a MOVS a MOVSB/MOVSW-hez képest.

SCASB
SCASW

"SCAn String Byte"/"SCAn String Word" - AL (AX) tartalmát összehasonlítja az (ES:DI) által címzett byte (szó) értékével: a flag-eket az [ES:DI] - AL (AX) különbség szerint állítja be. A DI regiszter a fent leírt módon változik.
Ezen utasítások általános alakja a

SCAS dest

("SCAn String") utasítás, amely ugyanazt a szerepet játssza a SCASB/SCASW utasításokhoz képest, mint a MOVS a MOVSB/MOVSW-hez képest.

Elemezzük a string-kezelő utasítások működését a MOVS utasítás segítségével! Tegyük fel, hogy DS, SI, ES és DI a kívánt értékeket tartalmazzák. Ekkor, ha a STATUS D flag-je 0, azaz "up", "felfelé", akkor a

MOVSB

utasítás elvégzi egy byte másolását, majd SI-t és DI-t eggyel megnöveli, míg a

MOVSW

a (szavas) másolás végrehajtása után kettővel.
Ha a D flag 1, azaz "down", "lefelé", akkor ugyanezek az utasítások előbb végrehajtják a másolást, azután csökkentik SI és DI értékét eggyel, illetve kettővel.
Tehát a MOVSB/MOVSW utasítás ciklikus ismételtetésével igen kényelmesen tudunk egész memóriablokkokat másolni; a címzőregiszterek tartalmának módosítása automatikus.

Figyelem: az utasítások nem egészen logikusak! Ha felfelé haladva előbb másolunk, azután módosítunk, akkor lefelé haladva előbb csökkenteni kellene, azután másolni. Tegyük fel, hogy egy blokkot akarunk átmásolni egy memóriaterületre, amelynek kezdőpoziciója valahol az átmásolandó blokk belsejében van, és a kezdőcímek és a hosszak ismertek. Ekkor kézenfekvő, hogy visszafelé másoljunk, mert így nem vesztünk adatot, ehhez

azonban az utolsó byte címére van szükségünk. A kezdőcím és a hossz összege mint cím az utolsó byte mögé mutat; ha az utasítások "logikusan" működnének, azonnal hozzáláthatnánk a másoláshoz. Tudván az utasítások e következetlenségét, az összeadás után még csökkenteni kell a címeket (esetleg tudomásul venni, hogy eggyel több elemet fogunk átmásolni).

Befejezésül arra kell még felhívnunk az Olvasó figyelmét, hogy a string-utasítások nem alkalmasak egy szegmensnél hosszabb (azaz 64 Kb-ot meghaladó) memóriablokk kezelésére. Ha ugyanis a DI folytonos növelés közben eléri az FFFFH maximális értéket, az újabb növelés a 0H-t állítja elő, amely a szegmens első byte-jára mutat (mint ismert, a címszámítások során fellépő túlcsordulások elvesznek). A processzor pedig nem módosítja a szegmensregiszter(ek) tartalmát, ha az indexregiszterek átlépik az FFFFH "küszöböt".

REP prefixum

"REPeat" - a REP kódot string-utasítások elé írhatjuk. Előzetesen CX-be kell töltenünk a string (memóriablokk) hosszát, és ekkor a REP prefixummal ellátott string utasítás addig ismétlődik CX műveletenkénti csökkentésével, míg CX nulla nem lesz.

> Megjegyzés: a REP prefixum és változatai lehetővé teszik, hogy egy legfeljebb 64 Kb-os memóriablokkot egyetlen utasítással új helyre másoljunk (MOVS), adott értékkel feltöltsünk (STOS), hogy megkeressük benne egy byte-nyi vagy szónyi érték első előfordulását (SCAS), hogy összehasonlítsunk két memóriablokot (CMPS).
> Megjegyezzük még, hogy a REP prefixum a SCAS és CMPS utasítások előtt REPZ-vel vagy REPE-vel egyenértékű!
> Felhívjuk a figyelmet, hogy ezek az utasítások nem tudnak "kibújni" a szegmensregiszterek által címzett szegmensekből, ezért szavas műveletek esetén nincs értelme 7FFFH-nál nagyobb számláló megadásának.

REPZ
REPE

"REPeat while Zero"/"REPeat while Equal" - a REPZ vagy REPE prefixum hatására a processzor CX folytonos csökkentése mellett megvizsgálja a "Z" (Zero flag) állását is, és ha a Z = 0 (vagyis a művelet eredménye nem 0), akkor abbahagyja az ismétlést.

Megjegyzés: a kilépés oka a Z-bitből olvasható ki. Ha az ismétlés azért ért véget, mert valamelyik elvégzett művelet eredménye nullától különbözött, akkor Z = 0, ha pedig a CX (tehát a blokk) fogyott el, akkor Z = 1.

Ez a prefixum operációs kódját tekintve megegyezik REP-el. Az adatmozgató utasítások előtt (MOVS, (LODS), STOS) mindig REP-et "érnek", az összehasonlító utasítások (CPMS, SCAS) előtt pedig mindig REPZ-t. A különböző mnemonic-ok csupán a tisztább kódolást segítik elő. (LODS azért szerepel külön zárójelben, mert használatának ismételtetett formában nincs semmi értelme - minden AL-be (AX-be) töltött adattal kell valamit csinálni.)
Gondoljuk meg azt is, hogy a MOVS, LODS és STOS utasítások nem módosítják a flag-eket, tehát nem baj, hogy REP itt REPZ-vel ekvivalens; azonban a CMPS és SCAS utasítások használata során ügyelnünk kell a prefixum helyes megválasztására. Gyakorlatilag a CMPS és SCAS utasításokat amúgyis blokkok összehasonlítására, illetve egy adott érték első (következő) előfordulásának megkeresésére használjuk - így a prefixumok ilyen egybeesése teljesen logikus.

REPNZ
REPNE

"REPeat while NonZero"/"REPeat while NotEqual" - a REPNZ vagy REPNE prefixum hatására a processzor CX folytonos csokkentése

mellett megvizsgálja a "Z" (Zero flag) állását is, és amennyiben Z = 1 (azaz az elvégzett művelet eredménye 0, akkor a CX tartalmától függetlenül abbahagyja az ismétlést.

Megjegyzés: a kilépés okát itt is a Z flag mutatja: ha azért lépett ki, mert a művelet eredménye 0 volt, akkor Z = 1, ha pedig CX fogyott el, akkor Z = 0.

Példák:

1. Másoljunk egy hexadecimális 100 elemű blokkot az adatszegmens 325. byte-tól a 652. byte-jával kezdődő területre!

```
        MOV     AX, DS
        MOV     ES, AX   ;Szegmensreg.

                MOV     SI, 325H
                MOV     DI, 653H ;Címek be
                MOV     CX, 100H ;Blokkhosszúság
        REP     MOVSB            ;Blokkmásolás
```

2. Hasonlítsunk össze két memóriablokkot, és adjuk vissza AL-ben a kiindulási, AH-ban pedig a célblokk első eltérő elemét! Tegyük fel, hogy CX, DS:SI és ES:DI elő vannak készítve.

```
COMPARE PROC    NEAR

        REPZ    CMPSB            ;Összehasonlítás
                JZ      COMP_EQU
                MOV     AL, [ SI ]
                MOV     AH, [ DI ]
                JMP     COMP_EX
COMP_EQU:
                XOR     AX, AX   ;AH és AL = 0
COMP_EX:
                RET

COMPARE ENDP
```

(Elnézést kérek az Olvasótól, hogy megint nem ismertetett jelöléseket használok. Véleményem szerint hasznosabb néha kész példákat is megadni, mint programtöredékeket. Ha PROC és ENDP zavarja, lapozzon előre a 4. fejezetre. A szerző)

3. Töltsünk fel egy memóriablokkot szóközökkel. Legyen a blokk címe BLOKK, hossza SIZE!
Lássuk előbb a gyalogos megoldást, utána egy trükkösebbet. A b) példa itt jóval hosszabb, de lehet olyan eset, amikor ezt a trükköt jól ki lehet használni - például ha egy byte tartalmát kell (előzetes kiírás nélkül) megismételni bizonyos számú következő byte-on.

a)

```
                MOV     AL, ' '  ;Kiírandó érték

                MOV     CX, SIZE ;Konstans
                MOV     DI, OFFSET BLOKK
                                 ;BLOKK címe
        REP     STOSB
```

b)

```
                MOV     AX, DS
                MOV     ES, AX   ;Szegmensreg.

                MOV     CX, SIZE ;Konstans
                MOV     DI, OFFSET BLOKK
                MOV     SI, DI

                MOV     [ DI ], ' ' ;Első elem ki
                INC     DI

                DEC     CX       ;Egy már ott van
        REP     MOVSB            ;Az elsőt ismétli
```

Megjegyzés: azért kell "nagyon bent" elhelyezni az utasításokat, hogy a REP (stb.) prefixumok szépen kiemelkedjenek a kódból, de mégse kelljen

őket az első oszlopba írni. Bármelyik prefixum elég fontos ahhoz, hogy erősen felhívjuk rá a figyelmet.

3.4. Vezérlésátadási utasitások

A vezérlésátadási utasitások a legfontosabb utasitások közé tartoznak, nélkülük programozni lehetetlen volna, míg például a legtöbb aritmetikai utasítás nélkülözhető.

A vezérlésátadási utasitásokat három fő csoportra osztva mutatjuk be. A csoportok: a szubrutinhívási és visszatérési, az ugró, végül a feltételes vezérlésátadási utasitások.

A szubrutinhívó utasitások segítségével tisztábban, logikusabban, biztonságosabban kódolhatunk. Használjuk őket, ahol csak lehet, minden apró önálló tevékenységet oldjunk meg szubrutin segitségével. Befektetett munkánk sokszorosan meg fog térülni a program biztonságos "belövésében", módosithatóságában, valamint helytakarékosságban.

A feltételes vezérlésátadó utasitások alapvető fontosságúak, semmiképpen nem nélkülözhetők.

A programozás elméleti szakemberei legszívesebben megszüntetnék az ugró utasitásokat. Valóban, ezek gyakran teljesen áttekinthetetlenné teszik a kódot; semmiképpen nem szabad ezeket sűrűn használni. Ahol csak lehet, dolgozzunk szubrutinhívásokkal. Egy bizonyos helyen azonban a legszigorúbb elvek szerint sem nélkülözhetjük ezeket. Ha egy programunk "elágazik" úgy, hogy egy feltétel függvényében az "a" vagy a "b" programrészt kell végrehajtani, majd a végrehajtás a "c" részen folytatódik mindkét esetben, akkor úgy szokás kódolni, hogy az "a" után helyezkedik el "b", ez után pedig "c". A feltételes ugrás elugrat a "b"-re, vagy "ráereszkedünk" az "a"-ra. A "b" befejezése után persze "rácsoroghatunk" "c"-re, de az "a" után mindenképpen el kell ugrani "c"-re, különben "ráesnénk" a "b"-re, ez pedig nem kivánatos.

Megjegyzendő még, hogy a feltételes vezérlésátadások mind rövidek, legfeljebb -128 és 127 byte-nyi távolság áthidalására képesek. Amennyiben nagyobb távolságra kell elugrani (ami bizony előfordul) csak a feltétel nélküli ugrások segithetnek. Ez már az Intel 8086/8088 sajátossága.

Szubrutinhivási és visszatérési utasítások

CALL op

"CALL" - az IP pillanatnyi értéke (a soron következő utasítás virtuális címe) stack-re kerül, majd IP felülíródik "op"-al, mely lehet direkt cím (immediate érték) vagy tetszőleges címzési móddal elért regiszter- vagy memóriaoperandus.
Lehet "rövid" (szegmensen belüli) vagy "hosszú" (szegmensközi) szubrutinhívást végrehajtani.
A hívás módját (rövid/hosszú) a hívott eljárás típusa határozza meg, bár utasításonként külön is előírható.
Megjegyezzük, hogy távoli hívás esetén először IP, majd CS kerül a stack-re.

RET

"RETurn" - a stack tetején levő értékkel felülírja IP-t (és távoli hívás esetén) CS-t. A visszatérés módját (rövid/hosszú) a hívott eljárás típusa határozza meg, bár utasításonként külön is előírható. A visszatérés módjának mindig meg kell egyeznie a tényleges hívás módjával!
Megjegyezzük, hogy távoli visszatérés esetén először CS, majd IP töltődik vissza.

RET konstans

ugyanaz, mint a RET, de a visszatérési cím felovasása után az SP-t "konstans"-sal megnöveli. Igy lehet egyetlen utasítással "elfelejteni" (azaz az SP visszaállításával törölni) a rutinnak a stack-en átadott paramétereket, amelyeket a rutin hívása előtt helyeztek el a stack-en. (lásd a C nyelvet !! ??)

> Megjegyzés: a felkiáltójel arra utal, hogy a C nyelvben a hívó a stack-en adja át a paramétereket a meghivottnak, ezért a meghivott "függvény" (szubrutin) visszatérésekor "takaritani" kell. Ez azt jelenti, hogy a hívó

által a stack-en elhelyezett paramétereket törölni kell. A kérdőjelek azt mutatják, hogy a C nyelv megvalósításaiban erre (bár lehetségesnek és kézenfekvőnek látszik) mégsem a RET utasítás ezen változatát használják fel. Ennek oka pedig a következő: nem a meghívott rutin tudja, hogy valójában hány paramétert kapott a stack-en, hanem a hívó, hogy valójában hány és milyen típusú paramétert készített elő (azaz hány PUSH utasítást hajtott végre). Ha tehát egy rutint nem mindig ugyanannyi paraméterrel hívunk meg (ami nem támogatott, de megengedett gyakorlat), és a meghívott takarítana, akkor végzetesen összezavarná a stack-et.

Ugrási utasítások

JMP op

"JuMP" - IP-t "op"-al felülírjuk. Az operandus lehet regiszter (16 bites cím, rövid ugrás), beépített adat (16 vagy 32 bites cím, direkt - rövid vagy hosszú - ugrás), vagy pedig memóriablokk (16 vagy 32 bites cím, indirekt - rövid vagy hosszú - ugrás).

Mint már a bevezetőben említettük, a JMP utasítás használata csak akkor ajánlható, ha kénytelenek vagyunk használni. Ha lehet, ilyen esetekben is kerüljük a "körmönfont" címzésű ugrásokat. A program olvashatósága szinte fontosabb, mint az, hogy jó legyen - olvasható programot még mindig ki lehet javítani, olvashatatlan, kusza, zagyva kód viszont kockázatos, javíthatatlan és csaknem biztos, hogy nem pontosan a programozó szándéka szerint működik.

Feltételes vezérlésátadó utasítások

A feltételes vezérlésátadási utasítások a programozás során (talán a MOV mellett) a leggyakrabban használt utasítások. Mind elméleti, mind gyakorlati szempontból rendkívül fontosak.

Minden feltételes vezérlésátadás esetén egy IP-relatív címzés megy végbe 8 bites értékkel, azonban az utasítások különbségét MASM (a macroassembler) számítja ki; nekünk csak arra kell ügyelnünk, hogy ne kíséreljünk meg "túlságosan" hosszú (azaz 127 byte-nál hosszabb) feltételes ugrást, mert ez lehetetlen, fordításkor hibaüzenethez vezet!

Természetesen ugorhatunk előre és hátra, tehát a 8 bites adat előjelkiterjesztéssel adódik IP-hez.

Mit kell tennünk, ha egy feltételes vezérlésátadással mégis nagyon messzire kell elugrani? Lássunk erre egy példát!

```
LABEL:
        ... kód ...     ;Hosszabb, mint 200 byte

        JNZ     LABEL   ;kellene.
```

(vagyis a JNZ utasítással azt vizsgáljuk, hogy az utolsó aritmetikai utasítás eredménye 0 volt-e vagy sem, s ha 0 volt, szeretnénk visszaugrani a LABEL-el címkézett utasításra.)

```
Ekkor:
LABEL:
        ... kód ...     ;Hosszabb, mint 200 byte
        JZ      LABELZ  ;
        JMP     LABEL   ;JNZ LABEL
LABELZ:
```

a követendő eljárás. Látni fogjuk, hogy minden feltételnek létezik az ellenkezője is, tehát a fenti trükk minden esetben alkalmazható. Ilyen utasítássorozatokat érdemes macroként is definiálni (lásd 4. fejezet).

Ciklusszervező utasítások

```
LOOP    cimke
```

"LOOP" - hasonlóan a REP prefixumhoz, eggyel csökkenti CX-et és ha az nem nulla, akkor IP-be a "cimke" értékét írja (azaz annyit ad hozzá IP-hez, hogy az éppen "cimke" legyen). Ez az utasítás a ciklusszervezést nagyon kényelmessé teszi.

```
LOOPNZ  cimke
LOOPNE  cimke
```

"LOOP while NonZero"/"LOOP while NotEqual" - hasonlóan a REPNE/REPNZ prefixumhoz, megvizsgálja Z értékét. Ha ez 0, akkor csökkenti CX-t, és ha ez még nem 0, végrehajtja az ugrást, különben nem.

```
LOOPZ   cimke
LOOPE   cimke
```

"LOOP while Zero"/"LOOP while Equal" - hasonlóan a REPE/REPZ prefixumhoz, megvizsgálja Z értékét. Ha ez 1, akkor csökkenti CX-t és ha ez még nem 0, végrehajtja az ugrást, különben nem.

> Megjegyzés: Z mutatja, miért fejeződött be a ciklus. Ha CX fogyott el, akkor Z = 1, ha pedig az utolsó flag-töltő művelet eredménye volt 0-tól különböző, akkor Z = 0.

Feltételes ugrások

A feltételes vezérlésátadó utasítások a STATUS egyes flag-jeit (vagy flag-csoportját) vizsgálják, és azok állásától függően hajtanak végre vezérlésátadást. A flag-eket előzőleg végrehajtott (főként aritmetikai) utasítások állítják be

automatikusan az eredmény függvényében (lásd 3.10.). A kiadott feltételes vezérlésátadás megvizsgálja a szükséges flag-eket és ha úgy találja, hogy a feltétel teljesült, akkor módosítja IP tartalmát (azaz végrehajtja a vezerlésátadást), ha pedig a feltétel nem teljesül, akkor nem csinál semmit.

Felfogásom szerint a feltételes ugrások két nagy csoportra oszthatók, a "logikai" és az "aritmetikai" csoportra. A csoportokban vannak közös elemek. A felosztást a következő elv szerint végezzük:

a logikai csoport elemei egyetlen STATUS-beli bitet vizsgálnak meg, és a feltétel teljesülése esetén ugranak vagy nem ugranak;

az aritmetikai csoport utasításai (ha szükséges) több flag-et is vizsgálnak egyszerre, hogy eldöntsék, az utolsó művelet eredménye például nagyobb vagy egyenlő-e nullával, s ha igen, ugranak. Ezek az utasítások tehát egy aritmetikus feltételt vizsgálnak.

A feltételes ugrásokat fejből megtanulni elég nehéz és felesleges is. Azt tanácsolom, higyjük el: minden értelmes feltételhez van egy ugrás, ami éppen azt valósítja meg, és mindennek az ellenkezője is megvan. Ha egyszer találkozunk egy megvalósítandó feltétellel, akkor nézzünk utána bármelyik könyvben - biztosan meg fogjuk találni a megfelelőt.

Aritmetikai csoport

a) Előjel nélküli műveletek eredményének vizsgálata
(a résztvevő adatokat előjel nélkülinek tekintjük):

```
        JA      cimke
        JNBE    cimke
```

"Jump if Above"/"Jump if Not Below or Equal" - ugrik, ha előjel nélkül "nagyobb" vagy "nem kisebb-egyenlő".

```
JAE     cimke
JNB     cimke
JNC     cimke
```

"Jump if Above or Equal"/"Jump if Not Below"/ "Jump if No Carry" - ugrik, ha előjel nélkül "nagyobb-egyenlő" vagy "nem kisebb".

```
JB      cimke
JNAE    cimke
JC      cimke
```

"Jump if Below"/"Jump if Not Above or Equal"/"Jump if Carry" - ugrik, ha előjel nélkül "kisebb" vagy "nem nagyobb-egyenlő".

```
JBE     cimke
JNA     cimke
```

"Jump if Below or Equal"/"Jump if Not Above" - ugrik, ha előjel nélkül "kisebb-egyenlő" vagy "nem nagyobb".

```
JZ      cimke
JE      cimke
```

"Jump if Zero"/"Jump if Equal" - akkor ugrik, ha "egyenlő", azaz Z = 1.

```
JNZ     cimke
JNE     cimke
```

"Jump if Not Zero"/"Jump if Not Equal" - ugrik, ha "nem egyenlő", azaz Z = 0.

Megjegyzés: a JZ/JE és a JNZ/JNE utasítások minden csoportnak elemei!

b) Előjeles műveletek eredményének vizsgálata
(kettes komplemensű előjeles számokkal végzett műveletek eredményének tesztelésére)

```
JG      cimke
JNLE    cimke
```

"Jump if Greater"/"Jump if Not Less or Equal" - ugrik, ha előjelesen "nagyobb" vagy "nem kisebb-egyenlő".

```
JGE     cimke
JNL     cimke
```

"Jump if Greater or Equal"/"Jump if Not Less" - ugrik, ha előjelesen "nagyobb-egyenlő" vagy "nem kisebb".

```
JL      cimke
JNGE    cimke
```

"Jump if Less"/"Jump if Not Greater or Equal" - ugrik, ha előjelesen "kisebb" vagy "nem nagyobb-egyenlő".

```
JLE     cimke
JNG     cimke
```

"Jump if Less or Equal"/"Jump if Not Greater" - ugrik, ha előjelesen "kisebb-egyenlő" vagy "nem nagyobb".

Megjegyzés: a JZ/JE és a JNZ/JNE utasítások e csoportnak is elemei.

Az aritmetikai csoport feltételtáblázata

(az egy sorban álló utasítások gépikódja azonos, tehát csak a mnemonic különböző)

Feltétel	Előjeles	Előjel nélküli
=	JE, JZ	JE, JZ
nem =	JNE, JNZ	JNE, JNZ
>	JG, JNLE	JA, JNBE
>=	JGE, JNL	JAE, JNB
<	JL, JNGE	JB, JNAE
<=	JLE, JNG	JBE, JNA

Logikai csoport

```
        JNO     cimke
```

"Jump if No Overflow" - ugrik, ha "nincs túlcsordulás", O = 0.

```
        JO      cimke
```

"Jump if Overflow" - ugrik, ha "van túlcsordulás", O = 1.

```
        JNP     cimke
        JPO     cimke
```

"Jump if No Parity"/"Jump if Parity Odd" - ugrik, ha P = 0, a paritás páratlan, "parity odd".

```
        JP      cimke
        JPE     cimke
```

"Jump if Parity"/"Jump if Parity Even" - ugrik, ha P = 1, a paritás páros, "parity even".

```
        JS      cimke
```

"Jump if Sign" - ugrik, ha "van előjel", S = 1. (Az eredmény negatív.)

```
        JNS     cimke
```

"Jump if No Sign" - ugrik, ha "nincs előjel", S = 0. (Az eredmény pozitív.)

```
        JC      cimke
```

"Jump if Carry (set)" - ugrik, ha a Carry 1, C = 1.

```
        JNC     cimke
```

"Jump if No Carry" - ugrik, ha a Carry 0, C = 0.

Megjegyzés: e két utasítás ("álruhában", azaz más mnemonikus kóddal) szerepelt az aritmetikai csoportban is.

```
        JZ      cimke
        JE      cimke
```

"Jump if Zero"/"Jump if Equal" - akkor ugrik, ha "egyenlő", azaz Z = 1.

```
JNZ     cimke
JNE     cimke
```

"Jump if Not Zero"/"Jump if Not Equal" - ugrik, ha "nem egyenlő", azaz Z = 0.

Megjegyzés: a JZ/JE és a JNZ/JNE utasítások szerepeltek az aritmetikai utasítások között is.

```
JCXZ    cimke
```

"Jump if CX is Zero" - ugrik, ha CX = 0.

Megjegyzés: ez az utasítás, mint látható, a LOOP "ellentéteként" is használható. A LOOP addig ismétli a ciklus végrehajtását, míg a CX regiszter tartalma (folytonos csökkentéssel) nem nulla. A JCXZ utasítás pedig addig ismételtetheti a ciklus végrehajtását, míg a CX regiszter 0.

3.5. Logikai utasítások

Minden logikai utasítás operandusai lehetnek 8 vagy 16 bitesek, és az operandus eléréséhez tetszőleges legális címzési módot alkalmazhatunk - a logikai utasítások cél-operandusa lehet regiszter és memóriában elhelyezett operandus is.

```
AND     op1, op2
```

"AND" - bitről bitre logikai "és" műveletet végez "op1" és "op2" között, és az eredményt "op1" helyére írja.

Az AND utasítás bitenkénti igazságtáblázata:

a/b	igaz	hamis
igaz	igaz	hamis
hamis	hamis	hamis

ahol "igaz" az 1-el töltött, "hamis" pedig a 0 értékű bitet jelenti.

OR op1, op2

"OR" - bitről bitre logikai "vagy" műveletet végez "op1" és "op2" között, és az eredményt "op1" helyére írja.

Az OR utasítás bitenkénti igazságtáblázata:

a/b	igaz	hamis
igaz	igaz	igaz
hamis	igaz	hamis

ahol "igaz" az 1-el töltött, "hamis" pedig a 0 értékű bitet jelenti.

XOR op1, op2

"eXclusive OR" - bitről bitre logikai "kizáró vagy" műveletet végez "op1" és "op2" között, és az eredményt "op1" helyére írja.

Az XOR utasítás bitenkénti igazságtáblázata:

a/b	igaz	hamis
igaz	hamis	igaz
hamis	igaz	hamis

ahol "igaz" az 1-el töltött, "hamis" pedig a 0 értékű bitet jelenti.

Megjegyzés: ez a három alapvető logikai utasítás bitek törlésére (AND), 1-el való töltésére (OR) és megfordítására (XOR) is szolgál. Sok esetben ugyanis nem két egyenrangú értéket használunk fel, hanem az egyik az operandus (ez rendszerint valamelyik regiszter tartalma), a másik pedig az ún. maszk (ami általában egy beépített adat). A maszk egy olyan érték, amelyben a kívánt bitek 1-es értékűek, a többi 0. Tegyük fel például, hogy az AX-ben levő operandusnak balról számítva a 3., 6. és 13. bitjét szeretnénk kezelni. Ekkor a maszk hexadecimális értéke 2048 (H). Először töröljük a kívánt biteket:

```
        AND     AX, NOT 2048H   ;Egyes komplemens
                                ;(Lásd 4. fej.)
        AND     AX, 0DFB7H      ;Egyenértékű
```

Amely biteken a felhasznált maszk (az eredeti egyes komplemense) 1, ott az eredmény az AX megfelelő bitje lesz. Ugyanis ha az eredeti bit 0, akkor 1 és 0 kerül szembe, ez pedig 0. Ha az eredeti bit 1, akkor 1 és 1 kerül szembe, ami 1. Ahol a maszk 0, ott az eredmény biztosan 0 lesz. Itt látható, miért hívják maszknak a maszkot: amely biteken a maszk 1, ott "át lehet látni" rajta, ott látjuk az eredeti értéket, míg a többi ponton a maszk "takar", tehát maszkolás után az eredeti értékek nem látszanak.

Töltsük most a biteket 1-el:

```
OR      AX, 2048H
```

Ahol a maszk 0, ott az igazságtábla értelmében az eredeti érték marad; ahol a maszk 1, az eredmény biztosan 1 lesz, tehát a "megcélzott" három bitet 1-el töltöttük.

Fordítsuk most meg a biteket!

```
XOR     AX, 2048H
```

Ahol a maszk 0, ott az eredmény az eredeti érték, ahol pedig a maszk 1, ott megfordul, hiszen 0 kizáró vagy 1 az 1, és 1 kizáró vagy 1 az 0.

```
NOT     op
```

"NOT" - "op" tartalmát negálja (egyes komplemens képzése). Minden bit helyére egyszerűen az ellenkező értéket írja.

```
TEST    op1, op2
```

"TEST" - bitről bitre logikai "és" műveletet végez "op1" és "op2" között, és az eredmény szerint állítja be a flag-eket. Az operandusok nem változnak.

3.6. Processzorvezérlő utasítások

```
CLC
```

"CLear Carry" - törli a "C" (carry) bitet.

STC

"SeT Carry" - 1-el tölti a "C" (carry) bitet.

CMC

"CoMplement Carry" - megfordítja (komplementálja) a "C" (carry) bitet .

Megjegyzés: ez a három utasítás teszi különösen alkalmassá a Carry bitet Igen/Nem információk tárolására.

CLD

"CLear Direction" - törli a "D" (direction) bitet. A string-kezelő utasítások során ekkor SI és/vagy DI növekszenek majd.

STD

"SeT Direction" - 1-el tölti a "D" (direction) bitet. A string-kezelő utasítások során ekkor SI és/vagy DI csökkenni fognak.

CLI

"CLear Interrupt" - törli az "I" (interrupt) bitet. Ez az utasítás letiltja a hardware-interrupt-ok fogadását (kivéve természetesen az NMI-t (lásd második kötet).

STI

"SeT Interrup - 1-el tölti az "I" (interrupt) bitet. Ez az utasítás engedélyezi hardware-interrupt-ok fogadását.

NOP

"No OPeration" - nincs tevékenység.

WAIT

"WAIT" - a processzor "wait" (várakozó) állapotba lép mindaddig, míg a TEST kivezetésen "low" szintet nem érzékel (külső szinkronizáláshoz). A "wait" állapot ad lehetőséget a DMA eszközök beépitésére, esetleg lassúbb memóriák kezelésére. Fontos szerepe van a külső processzorokkal (Intel 8087) való szinkron fenntartásában is.

LOCK

"LOCK" - A busz lezárása (prefixum). Ha egy tetszőleges utasítás előtt megadjuk, akkor az utasítás végrehajtása idejére a processzor LOCK ("lezárás") vezérlővonalával lezárja az adat- és címbuszt. Többprocesszoros környezetben azon utasitások előtt kell használni, mikor valamilyen tevékenység közben nem engedhető meg kívülről adott utasítások fogadása.

ESC op.kód, mem

"ESCape" - először a legfeljebb 6 bites "op.kód"-ot, majd a "mem" operandus címét kiirja ("rákényszeriti") a busz adat/címvonalaira. Ez az utasítás többprocesszoros környezetben más processzoroknak szóló utasitások kiadására szolgál.

> Megjegyzés: jellegzetes felhasználási területe az Intel 8087 koprocesszor "meghajtása". Ez a segédprocesszor állandóan a 8088 buszán "lóg", és mikor érzékeli az ESC-el kiküldött operációs kódot, akkor gyorsan "elkapja" a címvonalakra kiadott címet. Az operációs kód ekkor persze valamilyen 8087-nek szóló utasitás, a cím pedig az operandus címe.
> ESC-el kiküldött utasításokkal programozhatjuk a 8087-et, miközben saját memóriában tárolhatjuk az egyes operandusokat, és a 8088 hatékony címzési módjai segitségével érhetjük el őket. Bővebben lásd a második kötetben.

HLT

"HaLT" - a processzort "HALT" (leállított) állapotba lépteti. Innen csak a hardware reset billenti ki, vagy hardware interrupt. Ha a processzor a HLT utasítás kiadása előtt képes volt interruptot fogadni, akkor az interrupt lekezelése után a processzor a HLT utáni utasítással folytatja majd a munkát. Ha az utasítás előtt a processzor "disable interrupt" módban volt, csak az NMI interrupt (ez a 2. interrupt) (és persze a RESET) lépteti ki a processzort a "HALT" állapotból.

Megjegyzés: a hardware reset a processzor teljes újraindítását jelenti. Az újraindításhoz a processzor RESET lábán kell a feszültséget fel kell emelni. Ennek hatására egyébként a vezérlés az F000:FFF0 (vagy FFFF:0000) szegmens-offset címre adódik. Itt indul újra a végrehajtás.
A RESET a gyakorlatban ki/be kapcsolás, mert az IBM PC sajnálatosan nélkülözi a RESET gombot.
A programozási gyakorlatban a HLT utasítást soha nem adjuk ki; ha várni kell valamire, akkor írunk egy egyszerű várakozási ciklust.

3.7. Input-output utasítások

IN AL/AX, cím

"INput" - a (8 bites) "cím" által címzett 8 (16) bites I/O port beolvasása az AL (AX) regiszterbe.

IN AL/AX, DX

A DX által címzett 8 (16) bites I/O port beolvasása az AL (AX) regiszterbe.

OUT cím, AL/AX

"OUTput" - kiírás a (8 bites) "cím" által címzett 8 (16) bites I/O portra az AL (AX) regiszterből.

OUT DX, AL/AX

Kiírás a DX által címzett 8 (16) bites I/O portra az AL (AX) regiszterből.

Megjegyzés: az output műveletek elvégzése igen veszedelmes. Egyetlen "szerencsésen" kiadott OUT utasítás lerombolhatja az operációs rendszert, két-három ügyes OUT-tal pedig leformázhatjuk a lemezünk egy sávját. Ezért csak nagy óvatossággal adjunk ki OUT utasításokat, ahol lehet, ott bízzuk ezt a feladatot a rendszerfüggvényekre (második kötet).

3.8. Interrupt utasítások

INT sorszám

"INTerrupt" - software interrupt kiváltása. Hatására stack-re kerül a STATUS, majd CS, végül IP. Ezután a (8 bites) "sorszám" által megszabott sorszámú interrupt-vektorból CS-be kerül a vektor szegmens- és IP-be az offset-része (tehát aktivizálódik a kért sorszámú interrupt).

INTO

"INTerrupt Overflow" - hatására a processzor megvizsgálja az "O" (overflow) flag tartalmát. Ha ez 1, aktivizálja az overflow-interruptot, majd az interrupt rutin befejeződése után az INTO-t követő utasításon folytatja a program végrehajtását. Ha pedig az "O" bit 0, rögtön a következő utasításra tér át.

INT

"INTerrupt" - operandus nélkül ez az utasítás a 3. interruptot aktivizálja. Ez, mint a második kötetben olvasható, az ún. "breakpoint", töréspont interrupt. Ezen interrupt mögött valamilyen futáskövető program töréspont funkciója van. Az interrupt bekövetkezésekor az kezd majd futni, célszerűen lehetővé téve a program állapotának megvizsgálását.
Ez az utasítás 1 byte hosszú, így a futáskövető (DEBUG) program bármely utasításkód helyére "be tudja lopni". Tehát a RAM (Random Access Memory, tetszőleges elérésű memória) bármely pontjára lehetséges egy töréspontot tenni.

IRET

"Interrupt RETurn" - visszatérés interrupt-rutinból - a processzor megfelelő sorrendben felveszi a stack-ről az interrupt aktivizálása során elmentett értékeket (ti. a CS és az IP, valamint a STATUS értékét), visszaállítva a processzor eredeti állapotát.

> Megjegyzés: interrupt-rutinból csak IRET utasítással szabad visszatérni, és IRET-el lezárt rutint csakis INT utasítással szabad meghívni.

3.9. Bitforgató és bitléptető utasítások

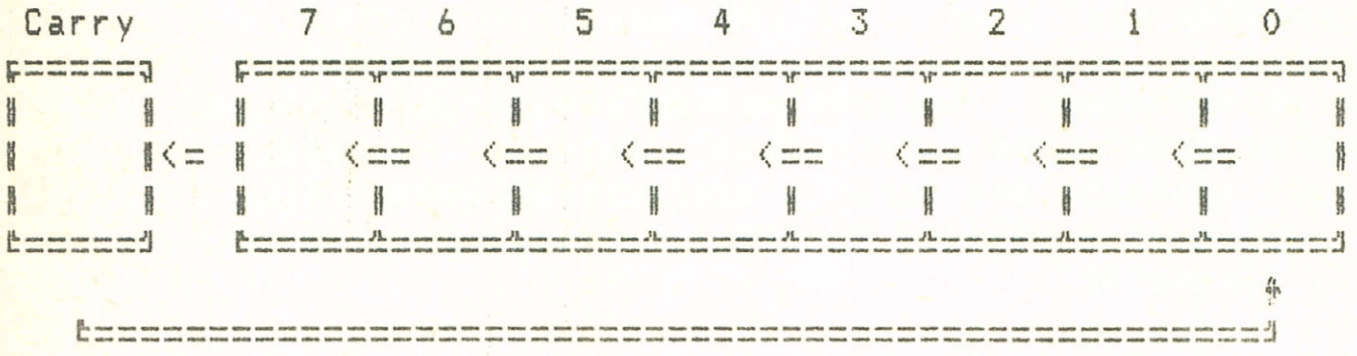
Az alábbi utasításokban az "op" lehet 8 vagy 16 bites, és tetszőleges címzési módot alkalmazhatunk. Általános alapelv még, hogy az operandus "kilépő" bitje minden esetben megjelenik a Carry flag-ben is. Bár az itt felsorolt utasítások közeli "rokonai" egymásnak, mégis jól elkülönithetően két csoportra oszthatók.

Bitforgató (rotálási) utasítások

RCL op, 1/CL

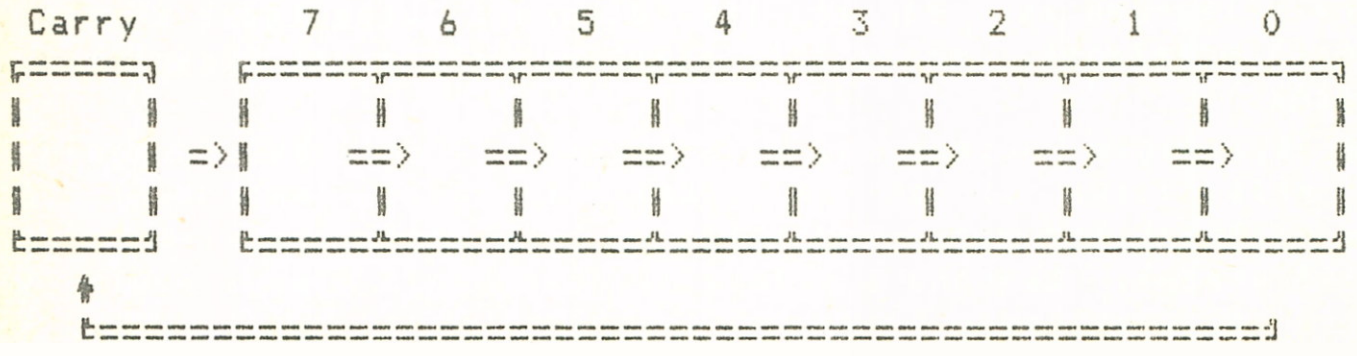
"Rotate through Carry Left" - ha 1-et írtunk, "op" tartalmát egyszer balra rotálja "carry-n át". A kicsorgó bit kerül Carry-be, Carry eredeti tartalma lép be jobbról. Ha az 1 konstans helyett CL-t adtunk meg, annyiszor ismétli a fenti eljárást, amennyi a CL regiszter tartalma. Az utasítás végrehajtása során CL tartalma változatlan marad.

Az utasítás hatása (byte-os változatban) a következő:



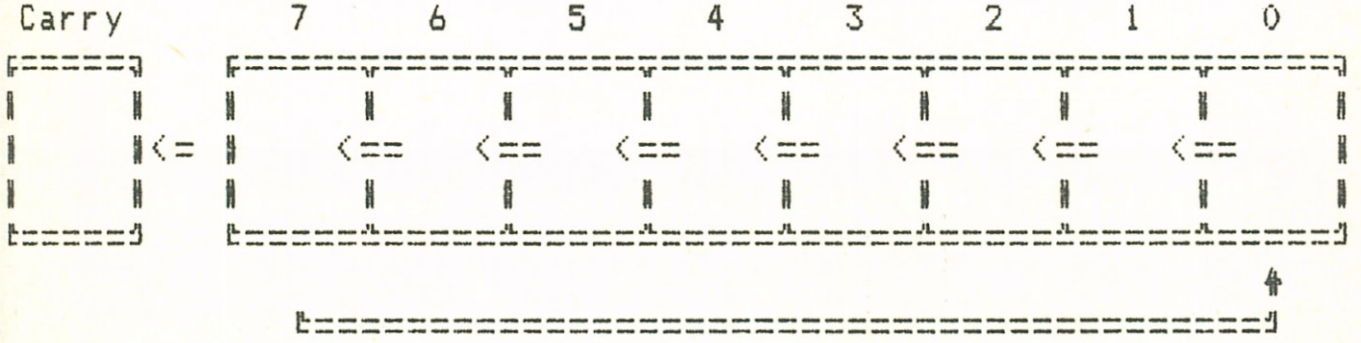
RCR op, 1/CL

"Rotate through Carry Right" - ha 1-et írtunk, "op" tartalmát egyszer jobbra rotálja "carry-n át". A kicsorgó bit kerül Carry-be, Carry eredeti tartalma lép be balról. Ha az 1 konstans helyett CL-t adtunk meg, annyiszor ismétli a fenti eljárást, amennyi a CL regiszter tartalma. Az utasítás végrehajtása során CL tartalma változatlan marad.
Az utasítás hatása (byte-os változatban) a következő:



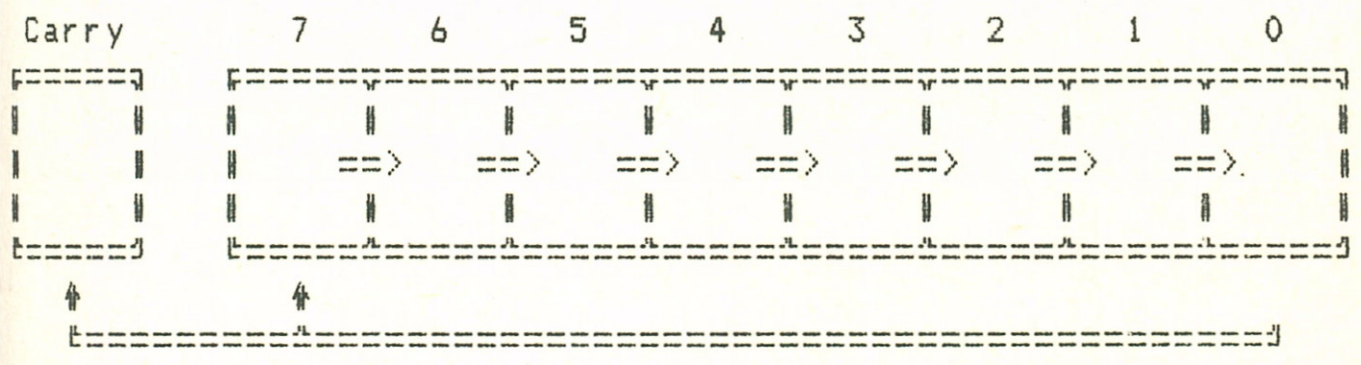
ROL op, 1/CL

"ROtate Left" - ha 1-et írtunk, "op" tartalmát egyszer balra rotálja "önmagán". A kicsorgó bit lép be jobbról és bekerül Carry-be is. Ha az 1 konstans helyett CL-t adtunk meg, annyiszor ismétli a fenti eljárást, amennyi a CL regiszter tartalma. Az utasítás végrehajtása során CL tartalma változatlan marad.
Az utasítás hatása (byte-os változatban) a következő:



ROR op, 1/CL

"ROtate Right" - ha 1-et írtunk, "op" tartalmát egyszer jobbra rotálja "önmagán". A kicsorgó bit lép be balról és bekerül Carry-be is. Ha az 1 konstans helyett CL-t adtunk meg, annyiszor ismétli a fenti eljárást, amennyi a CL regiszter tartalma. Az utasítás végrehajtása során CL tartalma változatlan marad.
Az utasítás hatása (byte-os változatban) a következő:



Megjegyzés: a két utasítástípus felhasználási területe elsősorban az ún. rekordok kezelése. A rekordok (lásd 4. fejezet) egyes mezőit

adatvesztés nélkül a ROL és ROR utasításokkal hozhatjuk olyan helyzetbe, ahol aritmetikai műveleteket végezhetünk velük.
Az RCL és RCR utasítások a rekordok kezelésével kapcsolatban akkor játszhatnak fontos szerepet, ha valamilyen okból egy duplaszót kívánunk rekordként kezelni. (A programozási példákat lásd a harmadik kötetben.)

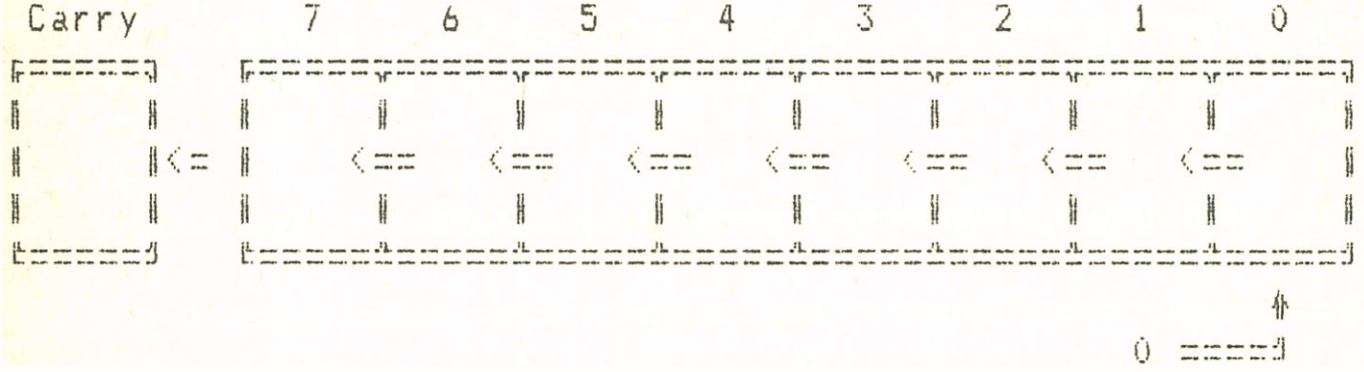
Bitléptető (shiftelési) utasítások

```
SAL     op, 1/CL
SHL     op, 1/CL
```

"Shift Arithmetical Left"/"SHift Left" - ha 1-et írtunk, "op" tartalmát egyszer balra lépteti. A kicsorgó bit bekerül Carry-be, jobbról 0 lép be. Ha az 1 konstans helyett CL-t adtunk meg, annyiszor ismétli a fenti eljárást, amennyi a CL regiszter tartalma. Az utasítás végrehajtása során CL tartalma változatlan marad.

Megjegyzés: ez a művelet (ha egyszer hajtjuk végre) megfelel egy előjel nélküli 2-vel való szorzásnak. Az esetleges szorzási túlcsordulás jelenik meg Carry-ben.

Az utasítás hatása (byte-os változatban) a következő:

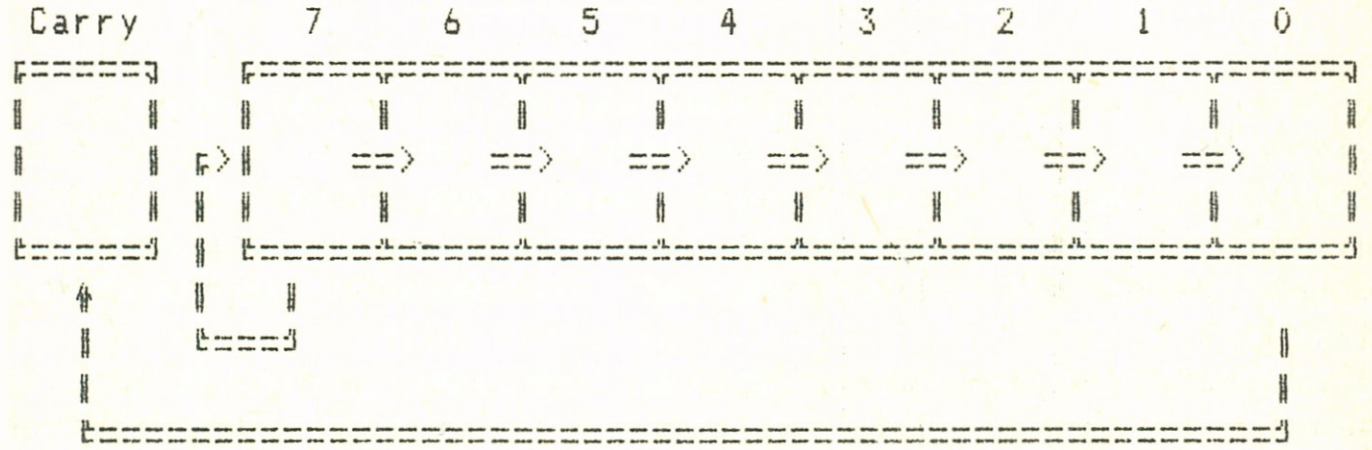


SAR op, 1/CL

"Shift Arithmetical Right" - ha 1-et írtunk, "op" tartalmát egyszer "aritmetikusan" jobbra lépteti. A kicsorgó bit bekerül Carry-be, balról pedig az előjelbit ismétlődik. Ha az 1 konstans helyett CL-t adtunk meg, annyiszor ismétli a fenti eljárást, amennyi a CL regiszter tartalma. Az utasítás végrehajtása során CL tartalma változatlan marad.

Megjegyzés: ez a művelet (ha egyszer hajtjuk végre) megfelel egy 2-vel való előjeles osztásnak. A hányados "op"-ban marad, a maradék Carry-be kerül.

Az utasítás hatása (byte-os változatban) a következő:

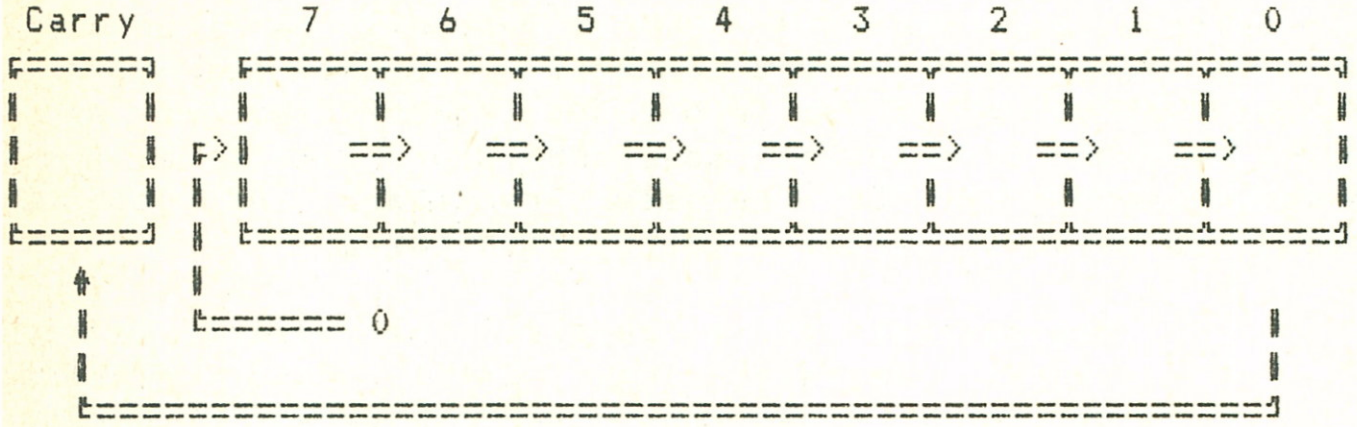


SHR op, 1/CL

"SHift Right" - ha 1-et írtunk, "op" tartalmát egyszer jobbra lépteti. A kicsorgó bit bekerül Carry-be, balról 0 lép be. Ha az 1 konstans helyett CL-t adtunk meg, annyiszor ismétli a fenti eljárást, amennyi a CL regiszter tartalma. Az utasítás végrehajtása során CL tartalma változatlan marad.

Megjegyzés: ez a művelet (ha egyszer hajtjuk végre) megfelel egy 2-vel való előjel nélküli osztásnak, melynek eredménye kerül a léptetett operandus helyére, a maradék pedig a Carry-be.

Az utasítás hatása (byte-os változatban) a következő:



3.10. A flag-ek viselkedése

A flag-ek között, mint láthattuk, az I, D és T flag nem aritmetikus, nem automatikus működésű. Ebben az alfejezetben azt részletezzük, hogy az egyes utasítások hogyan állítják a többi flag-et. Ehhez egy újabb csoportosítást használunk fel, aszerint soroljuk csoportokba az utasításokat, hogy milyen flag-eket állítanak.

Első lépésként azonban vizsgáljuk meg, hogy pontosan milyen szerepet játszanak a 2. fejezetben könnyedén "szabványos átvitel" stb. elnevezéssel illetett flag-ek az aritmetikai műveletek eredménye "minőségének" regisztrálásában! Tekintsünk egy igen egyszerű esetet: vizsgáljuk meg az összeadás műveletét és lehetséges eseteit. Az egyszerűség kedvéért 8 biten végezzük a műveleteket. A számábrázolás természetesen kettes komplemenső előjeles számábrázolás (lásd második kötet).

Végezzünk először egy egyszerű összeadást két pozitív bináris számmal, amelyek nem túl nagyok, tehát nem közelítik meg az ábrázolási tartomány felső határát. Később pedig megvizsgáljuk, mi történik, ha elég nagy vagy elég kicsi számokkal végzünk műveleteket.

1. példa:

```
     00110101          53
   + 00010010        + 18
-------------      ------
   0 01000111          71
```

Ez esetben nincs semmi meglepetés - az eredmény szépen megfér 8 biten.

Változtassuk meg most a második operandust, emeljük fel 19-ről ("kis pozitív") például 83-ra!

2. példa:

```
     00110101          53
   + 01010011        + 83
-------------      ------
   0 10001000         136    (ez  -120 !)
```

Itt az első meglepetés. Az eredmény, ha előjel nélkül vizsgáljuk, helyes. Mivel azonban kettes komplemenső számábrázolást alkalmazunk, negatív lett, hiszen az eredmény nem fér el az értékes hét biten.

Kísérelјük meg most két kis abszolút értékű negatív szám összeadását!

3. példa:

```
     11001011         -53
   + 11101110       + -18
-------------      ------
   1 10111001         -71
```

Az eredmény (8 bitre csonkítva) helyes, de mivel mindkét szám negatív volt, keletkezett egy kilencedik bit is - ezt a bitet fogja majd tartalmazni a Carry.

Adjunk most össze két abszolút értékben nagyobb negatív számot!

4. példa:

```
     11001011          -53
   + 10101101        + -83
-------------      ------
   1 01111000        -136     (ez   120 !)
```

Ez ismét "meglepő" eredmény - két negatív szám összege pozitív lett.

Végül vizsgáljuk meg a különböző előjelű számokkal végzett műveleteket!

5. példa:

```
     11001011          -53
   + 00010010        +  18
-------------      ------
   0 11011101          -35
```

6. példa:

```
     00110101           53
   + 11101110        + -18
-------------      ------
   1 00100011          35
```

Mint láthatjuk, az eredmények előjelhelyesek, de az egyik összeadás ismét nem-nulla kilencedik bitet eredményezett.

Utolsóként tekintsünk egy olyan esetet, mikor egy számot (mintegy véletlenül) éppen az ellentettjével adunk össze!

7. példa:

```
     11101110          -18
   + 00010010        +  18
-------------      ------
   1 00000000            0
```

Az eredmény 8 bitje valóban 0 lett, de ismét nullától eltérő kilencedik bit keletkezett!

Vizsgáljuk most meg, hogyan működik az öt aritmetikai flag (amely a következő négy: Zero, Sign, Carry, Overflow - az

Auxiliarry Carry szerepét a második kötetben ismertetjük) valamint a Parity flag e műveletek hatására! (Ismét felhívjuk a figyelmet, hogy a többi flag - I, D, és T - nem automatikus flag-ek, ezekre nem hatnak az aritmetikai utasítások.)

a) Zero flag: viselkedése roppant egyszerű. Csak a 7. példa esetén kapott egyes értéket, mert az eredmény 8 bitje ekkor 0. Az összes többi esetben (jelezve, hogy az eredmény 0-tól eltérő) a Zero flag tartalma 0 lesz.

b) Sign flag: mint említettük, ez a flag mindig együtt változik az eredmény előjelével. Tehát az egyes példákban mindig a legfelső bittel megegyező értéket kap. Figyeljük meg azonban, hogy a 2. és 4. példában a Sign flag értéke nem adja meg a valódi előjelet, mivel ekkor a belső átvitelek elrontották az eredmény előjelének értékét!

c) Carry flag: a Carry mindig megegyezik az eredmény előtt külön megadott számjeggyel, mivel a Carry nem más, mint az összeadás során a legfelső biten keletkezett túlcsordulás. Figyeljük meg, hogy "értéke", logikus tartalma csak az azonos előjelű számokkal végzett műveletek során van, ekkor az eredmény előjele a Sign flag helyett a Carry flag-ből olvasható ki. Különböző előjelű számokkal végzett műveletek után (5-7. példa) a Carry tartalma teljesen érdektelen.

A fenti eszmefuttatásból az következne, hogy az eredmény helyes értékeléséhez előzetesen le kell vizsgálnunk a műveletben résztvevő számokat előjelük szerint. Hagyományos (tehát Overflow flag-gel nem rendelkező) processzor esetében ez így is van. Azonban az Intel 8086/8088 Overflow flag-je kisegít bennünket:

d) Overflow flag: mint a második fejezetben olvasható, ez a flag az utolsó bitről, illetve -re csordulások "kizáró vagy"-a. Vizsgáljuk most meg példáinkat! Kettős felkiáltójel emeli ki a kritikus eseteket (ahol a Sign flag állása nem helyes). A flag viselkedését szemléltető táblázatot a következő oldalon olvashatjuk:

	előjelre keletkező	előjelről túlcsord.	Overflow flag
1. példa:	0	0	0
2. példa: (!!)	1	0	1
3. példa:	1	1	0
4. példa: (!!)	0	1	1
5. példa:	0	0	0
6. példa:	1	1	0
7. példa:	1	1	0

Láthatjuk tehát, hogy az Overflow flag pontosan a kritikus esetekben lesz 1 - összeadás után az Overflow flag vizsgálatával lehet eldönteni, hogy az eredmény előjelhelyes-e vagy sem.

Megjegyzés: ha két "n"-bites számot összeadunk, akkor az eredmény biztosan "n+1" bitet igényel, tehát mindenképpen előjelkiterjesztést kell végeznünk. Ez persze a szám helyigényét növeli. Byte-os változók összege szóban, szavasaké értelmesen duplaszóban fér csak el. Az Overflow flag állása adja meg, hogy az előjelkiterjesztést a Sign vagy a Carry flag alapján kell-e elvégezni.

e) Parity flag: ez a flag mindig az eredmény alsó 8 bitjének (tehát szavas művelet esetén sem a teljes 16 bit!) paritását tartalmazza. Jóllehet minden aritmetikai és logikai utasítás értelmesen módosítja a Parity flag-et, ennek a flag-nek gyakorlati jelentősége csak a különféle adatátviteli ellenőrzések során van (gyakran használják az adatátvitel helyességének ellenőrzésére az ún. hossz- és keresztparitást).

	Eredmény	Parity flag
1. példa:	01000111	páros - 1
2. példa:	10001000	páros - 1
3. példa:	10111001	páratlan - 0
4. példa:	01111000	páros - 1
5. példa:	11011101	páros - 1
6. példa:	00100011	páratlan - 0
7. példa:	00000000	páros - 1

Flag-et nem módosító utasítások

MOV, XCHG, XLAT, PUSH, PUSHF, POP, LDS, LES, LEA, MOVS, LODS, STOS, a REP prefixum
: az összes adatmozgatás, kivéve POPF-t, LAHF-t és SAHT-t;

CBW és CWD
: az aritmetikai utasítások közül;

CALL, RET, JMP, a feltételes vezérlésátadási, a LOOP utasítások és a JCXZ
: tehát az összes vezérlésátadás, kivéve az interrupt utasításokat;

HLT, WAIT, ESC, LOCK, valamint IN, OUT.

Az összes aritmetikai flag-et módosító utasítások

ADD, ADC, SUB, SBB, CMP, NEG, SCAS és CMPS
: a legtöbb aritmetikai, addíciós jelleqű utasítás.

A Carry-t nem módosító utasítások

INC és DEC
: (lásd 3.2.).

Auxiliarry Carry-t és a Carry-t módosító utasítások

AAA és AAS
: minden egyéb aritmetikai flag-et is módosítanak, de nem definiált módon.

Megjegyzés: a "nem definiált módon" kitétel arra utal, hogy ezen flag-ek értéke nem a bemeneti adatoktól függ, hanem a processzor belső

folyamataitól. Nem számíthatunk a flag-ek előző értékének megőrződésére sem.

Az Overflow flag-et nem módosító utasítások

DAA és DAS minden egyéb aritmetikai flag-et módosítanak.

Carry-t és Overflow-t módosító utasítások

MUL és IMUL minden egyéb aritmetikai flag-et is módosítanak, de nem definiált módon.

Aux. Carry-t definiálatlanul módosító utasítások

(Carry és Overflow értéke 0 lesz)

AND, OR, XOR, TEST, SHR, SHL és SAR

Ezek az utasítások alkalmasak arra is, hogy a Carry flag-et 0-ra töltsék úgy, hogy egyúttal valami mást is elérhetünk.
Például "mellékhatásként" AX-et törölhetjük a

```
XOR     AX, AX
```

utasítással (ekkor persze Zero 1 lesz!). A többi flag: Carry - 0, Parity - 1, Sign - 0, Overflow - 0.

Csak Carry-t és Overflow-t módosító utasítások

RCL, RCR, ROL és ROR.

Specifikus flag-eket módosító utasítások

CLC, STC, CMC, CLD, STD, CLI, STI, SAHF
Ez utóbbi csak a STATUS alsó 8 bitjét érinti.

A Parity, Sign és Zero-t érintő utasítások

AAD és AAM.

Aritmetikai flag-eket definiálatlanul hagyók

DIV és IDIV.

Minden flag-et érintő utasítások

POPF és IRET ezek az utasítások stack-ről töltik vissza a flag-ek értékét.

Az Interrupt és Trap flag-et érintő utasítások

INT és INTO törlik az Interrupt és Trap flag-eket, melyeket majd az IRET fog helyreállítani. (Az INTO utasítás csak akkor, ha végrehajtásakor az Overflow flag 1 volt, mivel ez feltételes utasítás.) A

DIV és IDIV utasítások csak osztási túlcsordulás esetén érintik e flag-eket.

4. fejezet

A MASM macroassembler szolgáltatásai

A 4. fejezet tartalomjegyzéke

4.0. Bevezetés

A MASM (Macro ASseMbler) fordító sokrétű szolgáltatásokat nyújt. Ezek közül a legfontosabbak:

szegmensek definiálása és kezelése
címzési módok, adatszélesség automatikus felismerése
relatív címek kiszámítása.
megjegyzések kezelése
macro-írási lehetőségek
adatstruktúra-definíciók
field (mező) -definíciók
feltételes fordítási lehetőségek
RADIX (számrendszer) kontroll

A MASM alkalmazkodott a processzor sajátosságaihoz. Ezért a programírás kezdetén az adat- és programterületek számára szegmenseket kell definiálnunk, amelyek megkönnyítik a furcsa címszámításban való tájékozódást, valamint kód- és adatterületeink megfelelő csoportosítását, tagolását.

Programjaink általában egy programszegmenst és több adatszegmenst tartalmaznak (64 Kb-nál hosszabb kódterületű programok írása nem javallt, sőt!). A MASM (és a LINK) lehetővé teszi, hogy a programkódot tetszőlegesen sok modulra bontva írjuk meg, és minden modulban a logikailag az adott modulhoz tartozó adatterületeket helyezzük el. A linkelés során a különböző helyekről származó, de azonos nevű és típusú szegmensben elhelyezett program- vagy adatterületek szépen egymás mellé szerkesztődnek majd.

Célszerű lehet úgynevezett "include"-file-okat írni, amelyek a legfontosabb konstans-, macro- stb. definíciókat tartalmazzák. E file-ok egy része "abszolút", tehát minden program számára szükséges vagy hasznos információt tartalmaz (közhasznú macro-k, az operációs rendszer hívásait segítő konstansok stb), más része pedig egy adott programhoz (feladathoz) kötődik. Segítségükkel egyfelől sok munkát, másfelől pedig bizonytalanságokat küszöbölhetünk ki.

4.1. Az assembly program elemei

Az assembly program sorokból áll, egy sor pedig a következő négy elemből:

[név] műveleti kód operandus(ok) [;megjegyzés]

Ezek közül a zárójelezettek ([...]) opcionálisak. Az egyes részeket egymástól szóközzel vagy tabulátorral kell elválasztani. Az első oszlopban csak a "név" kezdődhet, és annak mindig ott kell kezdődnie.

A "név" egy szimbólum; lehet macro neve, lehet változó, vagy egy program-belépési pont neve (tehát általában egy memóriapozíciót szimbolizál). Az adott sorra ennek megadásával hivatkozhatunk.

Egy név a következő karakterekből állhat: az angol ABC betűi A-tól Z-ig (a kisbetűket a fordító nagybetűvé konvertálja), a számjegyek 0-tól 9-ig, valamint a "?" (kérdőjel), "." (pont), "@", "_" és "$" speciális szimbólumok. A név első karaktere nem lehet számjegy. A "." (pont) karakter csak az első pozíción használható (lásd STRUC pszeudo-operátor). A név első 31 karaktere értelmezett.

A "műveleti kód" minden nem-üres programsor legfontosabb eleme; lehet egy mnemonic (lásd 2. fejezet), vagy egy pszeudo-operátor, egy a fordítónak szóló utasítás (lásd lejjebb).

Az "operandus" a "műveleti kód"-hoz tartozó, s annak megfelelő szintaxisú operandust, a művelet "tárgyát" jelenti (számos műveleti kódhoz tartozik több operandus). Lényegében véve egy operandus lehet egy szimbólum, egy adat (konstans), vagy egy ezekből képzett kifejezés (lásd konstansok).

A "megjegyzés" (comment) egy tetszőleges szöveg. Mindig ";"-vel kell elválasztani a sortól. A megjegyzés a ";"-től a sor végéig tart. A fordító figyelmen kívül hagyja.

Megengedett üres sorok beírása, és az egyes elemek között tetszőlegesen sok szóköz vagy tabulátor felhasználása. Ezeket a lehetőségeket használjuk ki a program olvashatóbbá tételére.

4.2. Konstansok megadása

Számkonstansok megadása

Számkonstansokat a programszövegben a szokásos alakban adhatunk meg. Minden számnak a 0 - 9 számjegyek valamelyikével kell kezdődnie.

Példa:

```
        MOV     AX, 27          ;AX-be (dec.) 27-et tölt

        OR      SI, 32          ;SI 5. bitjét 1-el tölti

        ADD     AL, 0FFH        ;AL-hez 255-öt ad
```

Számrendszermegadás

a) Alapértelmezés megadása:

.RADIX i

pszeudo-operátor segítségével átállíthatjuk az alapértelmezett számrendszert "i"-re. Ha nem adjuk ki .RADIX-t, akkor az alapértelmezés decimális.

b) Eltérés az alapértelmezéstől:

szám	alapértelmezett számrendszer
számB	bináris
számO	oktális
számD	decimális
számH	hexadecimális

számként értelmeződik.

Karakterkonstansok, stringek

Karakterkonstansok megadása idézőjelek közé zárva lehetséges:

```
'Ez egy string'
"EZ IS EGY STRING."
```

Megjegyezzük, hogy kettőnél több karakter megadása csak a DB pszeudo-operátor után megengedett.

4.3. Szimbólumok

A szimbólumok definícióját minden esetben egy szegmensen (a memória egy folytonos területén) belül kell elhelyezni. A szegmensdefiniálást lásd 4.5.1.-ben.

A szimbólumok memóriapozíciókat jelentenek. A MASM kétféle szimbólumot ismer: a címkéket és a változókat. A címkék a programterület egyes pozíciói (amelyek utasításokat tartalmaznak), a változók pedig területfoglaló operátorokkal kijelölt memóriapozíciók.

Minden szimbólumnak három attributuma, jellemző sajátossága van: a szegmens (amelyen belül definiáltuk), az offset (a szegmensen belüli virtuális cím) és a típus.

4.3.1. Címkék

A címkéket (label) csak programterületen definiálhatjuk, és JMP vagy CALL utasítások operandusai lehetnek.

A címkék attributumai:

A "szegmens" annak a paragrafusnak a címe, amelyen a címkét tartalmazó szegmens kezdődik. Amikor a címkére hivatkozunk, valamelyik szegmensregiszterben (címke esetén általában a CS-ben) benne kell lennie ennek az értéknek (hacsak nem éppen ezzel a vezérlésátadási utasítással írjuk be). Megjegyezzük, hogy a "szegmens" konkrét számértéke csak futás közben dől el;

itt szegmens alatt a fordító számára definiált szegmens nevét értjük. Lásd még a SEG pszeudo-operátort.

Az "offset" nem más, mint a cimkének a szegmens kezdetétől vett távolsága byte-okban. (Lásd még az OFFSET pszeudo-operátort, 4.4.)

A "típus" azt szabja meg, hogyan hivatkozhatunk a cimkére. A cimkének kétféle típusa lehet: NEAR és FAR. Egy NEAR típusú cimkére "közeli", szegmensen belüli, míg FAR típusúra "távoli", azaz szegmensközi vezérlésátadással hivatkozhatunk.

A cimkéket a sor elején kell megadni. Ha ":"-al zárjuk le, akkor NEAR típusúként definiáljuk a cimkét:

```
CIMKE:
```

Az ilyen cimkére a FAR PTR pszeudo-operátor segítségével lehet "távolról" hivatkozni (lásd 4.4.).

FAR típusú cimkét vagy a PROC pszeudo-operátor, vagy a LABEL segítségével definiálhatunk:

```
ROUT    PROC    FAR             ;Szubrutin kezdete,
        ...                     ;
        ...                     ; utasitásai
        ...                     ;
ROUT    ENDP                    ;és lezárása
```

vagy

```
ENTRY   LABEL   FAR             ;ENTRY egy távoli cimke
```

Előbbire egy

```
        CALL    ROUT            ;Szubrutin hívása
```

utóbbira pedig

```
        JMP     ENTRY           ;Távoli ugrás!
```

automatikusan távoli ugrásként értelmeződik. Ilyen esetekben a MASM, tudván, hogy a cimkék FAR típusúak, automatikusan szegmens:offset címzést épít be az utasítás kódjába.

4.3.2. Változók

A változók definíciója a területfoglaló pszeudo-operátorok segítségével lehetséges (lásd 4.5.2.).

A változóknak (adatcimkéknek) szintén három attributumuk van: a szegmens, az offset és a típus. A szegmens és az offset pontosan az, mint a cimkék esetében, azzal a különbséggel, hogy itt bármely szegmensregiszter segítségével címezhetünk (bár SS nem javallott).

A változók típusa tulajdonképpen egy konstans érték, amelyet automatikusan a lefoglalt terület típusa szab meg. A típusok a következők lehetnek: BYTE (1), WORD (2), DWORD (4), QWORD (8), STRUC (felhasználó szabja meg) és RECORD (2) (lásd 4.5.2.).

Ha ettől el akarunk térni, akkor a LABEL pszeudo-operátor segítségével definiálhatunk egy kívánt típusú szimbólumot. Legyen pl.

```
DAT     DB      0FH, 07FH       ;Lefoglal két byte-ot
```

Ekkor a

```
        MOV     AL, DAT         ;0F-et tölt AL-be
```

de a

```
        MOV     AX, DAT
```

utasítás fordítási hibát okoz, mert DAT típusa BYTE (az ehhez tartozó számérték 1), AX pedig szavas - a típusok nem egyeznek. Ilyenkor a következőképpen járhatunk el:

```
DATW    LABEL   WORD            ;DATW  WORD típusú szimb.
DAT     DB      0FH, 07FH       ;Két byte-ot foglalunk le

        MOV     AL, DAT         ;Byte-ot mozgat
        MOV     AX, DATW        ;Szót mozgat
```

Tehát a BYTE típusú DAT változóval megegyező offsetre definiálunk egy DATW nevű, WORD típusú változót.

4.4. Kifejezések

Kifejezéseket szimbólumok és konstansok, valamint különböző műveleti jelek és pszeudo-operátorok segítségével adhatunk meg. Természetesen egy változó használatakor nem hivatkozhatunk annak tartalmára, csak a címére - olyan értékekre, amelyek már fordítási időben ismertek.

Az alkalmazható operátorok a precedencia szerinti sorrendben (a magasabb precedenciájúakkal kezdve):

1.)

zárójelezett elemek - (...) vagy [...]

a zárójel természetesen minden más operátornál magasabb precedeniájú

. (pont) - struktúranév.változó

a "." (pont) operátor struktúranevek és struktúratagok összekötésére szolgál.

LENGTH - egy terület elemeinek száma
SIZE - egy terület hossza byte-okban
WIDTH - egy rekord field-jének szélessége

Definiáljunk például egy szavakból álló tömböt az alábbi programsorral:

```
EXP     DW      100     DUP     ( 1 )
```

ez 100 szót folgal le, kezdeti értékként 1-et ír mindegyikbe. Ekkor

```
LENGTH  EXP     = 100
TYPE    EXP     =   2
SIZE    EXP     = 200
```

Megjegyzés: DW, DUP - lásd 4.5.2.

2.)
sregnév: - szegmens-átdefiniálás

Például:

```
MOV     AX, ES:[ BX ]
```

amely az ES által címzett szegmens BX által címzett szavát olvassa be AX-be.

3.)
PTR - egy változó típusának átdefiniálása

Például:

```
DAT     DB      0, 0    ;DAT  BYTE-típusú

MOV     AX, WORD PTR DAT        ;DAT-ot szavassá
```

A fenti példa azt mutatja, hogyan lehet két egymást követő byte-ot (ha ez indokolt) egy szóként egy utasítással kezelni. A MASM elég "intelligens" ahhoz, hogy felismerje a típusok különbözőségét. A WORD PTR operátor mindössze arra szolgál, hogy "közöljük" a MASM-al: tudjuk, hogy eltérnek a típusok, de éppen ezt és a soron következő byte-ot akarjuk olvasni.
Ez az operátor tehát "csak" a hibaüzenet elkerülésére szolgál.

A PTR mellett használható típusnevek:

BYTE, WORD, DWORD, QWORD, TBYTE, valamint
NEAR és FAR.

Példa: tegyük fel, hogy VAR egy szavas változó, amelynek alsó byte-ját akarjuk AL-be olvasni. Ekkor

```
MOV     AL, BYTE PTR    VAR
```

a helyes utasítás.
Megjegyezzük még, hogy a QWORD és TBYTE kulcsszavakat eléggé ritkán, szinte csak a 8087 koprocesszor programozása során használjuk.

OFFSET - egy szimbólum offsetjét adja meg
SEG - egy szimbólum szegmensét jelenti
TYPE - egy terület típusa, elemeinek hossza

Példa: legyen ARR a következő terület neve:

```
ARR     DQ      10      DUP     ( ? )
```

Ekkor

```
        MOV     BX, OFFSET ARR
```

az ARR terület offsetjét (a szegmens kezdetétől vett távolságát) tölti be BX-be;

```
        MOV     AX, SEG ARR
        MOV     ES, AX
```

az ARR szegmensének címét írja be AX-be, majd onnan az ES szegmensregiszterbe; végül pedig

```
        MOV     CX, TYPE ARR
```

után CX-be 8 kerül.

Megjegyzés: ha valamely területet a DUP operátor segítségével, többszörösen felhasznált területfoglaló operátorokkal definiálunk, akkor a terület voltaképpen az alapelemek tömbje (lásd 4.5.2.); típusa viszont az alapelem típusa, azaz annak hossza byte-okban.

THIS - adott offset-ű vagy szegmensű és típusú operandus létrehozására szolgál. Például:

```
BT_ARR  EQU     THIS    BYTE
WD_ARR  DW      100     DUP     (?)
```

Milyen területek és milyen szimbólumok keletkeznek ezen utasítások hatására? A fordító lefoglal egy 100 szóból álló tömböt, melynek kezdőcíme WD_ARR (word

array) lesz. BT_ARR egy olyan szimbólum, melynek offsetje és szegmense megegyezik WD_ARR-al, de típusa BYTE. Segítségével a lefoglalt memóriaterületet elérhetjük mint szavak 100 elemű tömbjét (ennek neve lesz WD_ARR), de hivatkozhatunk rá, mint byte-ok 200 elemű tömbjére is (neve BT_ARR). Kiadhatunk tehát ilyen utasításokat:

```
        MOV     AX, WD_ARR[ 23 ]
```

és

```
        MOV     BL, BT_ARR[ 30 ]
```

Megjegyzés: a THIS nagyon hasonlít a LABEL operátorhoz (lásd 4.6.2.). Főleg azért hasznos, mert mintegy "rámutat" az adott pontra: itt és éppen itt definiáljuk a másik szimbólumot. Haszna tehát abban rejlik, hogy a programot némileg olvashatóbbá teszi.

4.)

HIGH – egy szó felső felére címez
LOW – egy szó alsó felére címez

Például:

```
DAT     DW      0

MOV     AH, HIGH DAT    ;DAT+1-ről olvas byte-ot
```

5.)
Szorzás, osztás és "modulo" operátor

Például:

```
MOV     CX, (TYPE EXP)*(LENGTH EXP)

MOV     AX, 100  MOD 17 ;AX-be 15-t ír
```

6.)
Összeadás, kivonás és negatív előjel

Például:

```
MOV     AX, KONST1 + KONST2     ;Összeg AX-be
MOV     SI, -100                ;SI-be -100
```

7.)
EQ, NE, LT, LE, GT, GE - relációs operátorok

(feltételes blokkokban használatosak, lásd 4.9.)

8.)
NOT - logikai nem

Például:

```
AND     AX, NOT 0401H   ;AX 0. és 10. bitjét törli
```

9.)
AND - logikai és

Például:

```
MOV     AX, 0F2EH AND 0753H     ;AX-be 0702H kerül
```

10.)
OR - logikai "vagy"
XOR - "kizáró vagy"

Például:

```
BT_ATT  DB      01H OR 02H OR 04H

        OR      AL, 0AH OR 0DH          ;AL OR 0FH !
```

Megjegyzés: ügyeljünk arra, hogy ezeket a logikai operátorokat precízen és kommentálva használjuk, mert

(elég szerencsétlen módon) megegyeznek a megfelelő assembly-mnemonic-okkal, és ez tréfás félreértésekre adhat alkalmat!

11.)

SHORT - rövid ugrás (IP-relatív)

Például:

```
JMP     cimke           ;Direkt ugrás
JMP     SHORT cimke     ;IP-relatív ugrás
```

4.5. Pszeudo-operátorok

A pszeudo-operátorok a fordítónak szóló utasítások. Egyesek közülük területfoglalásra, mások macrodefiniálásra, és az ezzel rokon struktúra- és rekorddefiniálásra, vagy blokk-ismételtetésre szolgálnak. Igen fontosak még a konstansokat, valamint az eljárásokat definiáló pszeudo-operátorok. A többi pszeudo-operátor lényegében a fordítás menetét vezérli.

4.5.1. Szegmensek és szegmenscsoportok definiálása

a) Szegmensdefiniálás

Már az előző fejezetekben olvashattuk, hogy a szegmens az IBM-PC/XT programozásának alapvetően fontos fogalma. Minden területfoglalással járó assembler utasítást vagy pszeudo-operátort egy szegmensen belül kell elhelyezni. A szegmenst neve és osztálya azonosítja. A programozó szabadon választhatja meg mindkettőt.

Minden megnyitott szegmenst le kell zárni. A lezárt szegmensek újra megnyithatók akár az adott modulon belül, akár más modulban. A többször megnyitott szegmensek az "összekötési" paramétertől (szegmenstípus) függően egymás mögé vagy egymásra

szerkesztődnek majd. Az első esetben egyszerűen folytatjuk a korábban elkezdett szegmens feltöltését, a második eset pedig overlay ágak definiálására szolgál.

A többször elkezdett szegmenseket a programszerkesztő fogja összeszerkeszteni. A szerkesztés stratégiája vázlatosan a következő: első lépésben egymáshoz függeszti az azonos nevű és osztályú szegmenseket, hogy ezek folytonosan helyezkedjenek el. Ezután egymás mellé helyezi az azonos osztályú szegmenseket. A szegmensek sorrendje ezen belül az előfordulás sorrendjét követi. Fontos ismeret, hogy a LINK minden MEMORY típusú szegmenst a program legvégére szerkeszt - ez mindig a legmagasabb fizikai címre kerül. Ezért a MEMORY szegmens utolsó byte-ja egyúttal a program utolsó byte-ja is - efölött már szabad memóriaterület van.

Milyen szegmenseket definiálunk általában? Minden program értelemszerűen tartalmaz egy kódszegmenst, egy adatszegmenst és egy stack-szegmenst. Elmondható, hogy ezek nélkül a program gyakorlatilag futásra képtelen. Egy program (akárhány modulból álljon is) pontosan egy stack-szegmenst tartalmazzon. Természetesen akárhány kód- és adatszegmenst definiálhatunk.

Lássuk most magát a szegmensdefiniáló operátort!

SEGMENT - szegmens megnyitása

ENDS - szegmens lezárása

Használata:

```
szg-név SEGMENT [align-type]    [combine-type] ['class']
                (igazítás)      (szegmenstípus) (osztály)
        .
szg-név ENDS
```

ahol:

"szg-név"

a programozó által adott szegmensnév. A szegmensnév tetszőlegesen választható. Amennyiben magasszintű nyelvhez akarunk assemblyben írt részeket csatolni, ügyeljünk a compiler által adott nevekre!

Alapértelmezés nincs.

"align-type"
(igazítás) a szegmens elhelyezésére utal, lehet

PARA - paragrafushatárra (16-al osztható cím)
BYTE - byte-határra (látszólag)
WORD - szóhatárra (látszólag)
PAGE - laphatárra (256-el osztható cím)

Alapértelmezés: PARA

Megjegyzés: a BYTE és WORD igazítás természetesen csak "látszólag" valósulhat meg akkor, ha az adott nevű és osztályú szegmenst csak egyszer nyitjuk meg. Ekkor a kész szegmens mindenképpen egy paragrafushatárra fog kerülni.
Ha azonban egy szegmenst többször megnyitunk és lezárunk, mindig folytatva az egyszer már definiált szegmens feltöltését, a további szegmensek természetesen igazodhatnak byte- vagy szóhatárra.
Pontosan ez a helyzet akkor, ha assemblyben írt részeket fűzünk egy más nyelven írott programhoz. Az adott fordító a program kódrésze számára definiál egy szegmenst. Az assembly részletet ehhez a szegmenshez kell fűznünk; ekkor a BYTE vagy WORD igazítás nagyon fontos, hogy az assembly részlet folytonosan illeszkedjen a másik program (egyik) kódszegmenséhez. Nem is annyira a folytonosságot kell biztosítanunk, mint azt elkerülni, hogy egy visszafelé igazítás miatt átfedés keletkezzen.

"combine-type"
(szó szerint kombinálási típus, tehát összekötési típus, értelem szerint inkább szegmenstípus) szabja meg, hogy a többször elkezdett azonos nevű szegmensek milyen viszonyban vannak egymással. A szegmensek e paramétere azonban más jellemzők megadására is szolgál, melyek inkább a szegmens típusának tekinthetők. Emiatt neveztem ezt a paramétert inkább szegmenstípusnak, mint kombinálási típusnak vagy másegyébnek. A szegmenstípus megadására az alábbi kulcsszavakból választhatunk:

PUBLIC - egymás után szerkesztendők
COMMON - egymásra szerkesztendők (overlay)

Megjegyzés: ezek a kulcsszavak definiálják a szegmens "összekötését": mi a teendő, ha a fordító vagy a programszerkesztő több azonos nevű szegmens definiciójával találkozik, akkor ezeket egymásra fordítsa (szerkessze) vagy egymás mögé.

AT kif - a szegmensnek a "kif"-edik paragrafusra kell kerülnie

Megjegyzés: ezt az opciót egyfelől ROM (Read Only Memory) területre szánt programok esetén használjuk. Leggyakrabban azonban olyan szegmensek paramétere, amelyeknek célja, hogy más programok (horribile dictu: egyenesen az operációs rendszer) adatterületeinek offsetjeit definiálják. Ilyen esetben elkészítjük a kívánt adatterület "mását" területfoglalások nélkül, pusztán az ORG operátor használatával. Az AT operátor segítségével a megfelelő helyre pozícionáljuk a szegmenst, majd futás közben a szegmens kezdőcímét valamelyik szegmensregiszterbe töltve közvetlenül érhető el az kívánt adatterület.

STACK - a stack része lesz - ebből csak egy lehet

Megjegyzés: a stack szegmens minden assembly program integráns része. Amennyiben a szabályok szerint definiáljuk, a kész program betöltése után a stackterület és a kezeléséhez szükséges regiszterek előkészítve várnak bennünket. Lásd 4.13. - assembly alapfile.

MEMORY - memóriaterület lesz; az összes szegmens fölé fog kerülni. Ha több ilyen típusú van, ezek COMMON viszonyban vannak! Vigyázzunk azonban, hogy osztálya is MEMORY legyen!

A szegmenstípus alapértelmzése PUBLIC.

"'class'"
a szegmens osztálya (a '-k használata kötelező; a LINK erről ismeri fel, hogy ez a szó adja meg a szegmens osztályát). Az osztály nevét a programozó adja meg tetszése szerint. A LINK az azonos osztályú szegmenseket folytonosan helyezi el. Az osztályon belül abban a sorrendben helyezkednek majd el a szegmensek, ahogy az object-file-okban megtalálja őket a LINK.
A javasolt szegmensosztályok a következők: 'CODE', 'DATA', 'CONSTANT', 'MEMORY', 'STACK', melyek rendre a kódot, az adatokat (változókat), a memóriában elhelyezett konstansokat (vagyis az állandó értéket tartalmazó változókat), a dinamikusan használt memóriaterületet és végül a stack területet jelentik.

Megjegyzés: amennyiben más nyelvhez assemblyben írunk bizonyos részleteket, alkalmazkodnunk kell az adott fordító által használt nevekhez, osztályokhoz stb. Ellenkező esetben sohasem fognak "találkozni" a más nyelven és az assemblyben írott részek.

A MASM mindezen operandusokat továbbítja a LINK-nek, amely majd végleges helyükre szerkeszti az egyes szegmenseket.

b) Szegmenscsoportok definiálása

A programozó több különböző nevű, típusú és osztályú szegmenst foghat össze egy csoportba. Ezek a szegmensek egy fizikai szegmensre kerülnek majd, az őket megcímző szegmensregiszterek a memória ugyanazon pontjára mutatnak. Ezáltal a program "rendezettebben" helyezkedik el a memóriában, és programozás közben nem kell olyan gondosan ügyelni arra, hogy a szegmensregiszterek éppen hová mutatnak. A csoportba szervezés feltétele azonban, hogy az egybefogott szegmensek hossza összesen se lépje túl a 64 Kb-ot. Erre ügyelni a programozó feladata, mert a MASM (nem tudván, nincs-e több source file) ezt nem ellenőrizheti.
Amennyiben programunk minden szegmensét összefogjuk egy csoportba, az alább ismertetett ASSUME operátorral nem a

szegmensek, hanem a csoport nevét asszociáljuk minden szegmensregiszterhez. A szegmensregiszterek előkészítését EXE típusú programnál természetesen nekünk kell elvégezni! A GROUP operátor használata megkönnyíti programunk .COM típusú file-á alakítását (lásd 4.13., 5.4. és második könyv), bár nem szükséges feltétel.

A GROUP operátor használata:

GROUP - szegmenscsoport definició

cs-név GROUP szg-név1[, szg-név2, ...]

A "cs-név" nevű szegmenscsoport a GROUP operátor mögött felsorolt "szg-név1", "szg-név2", ... nevű szegmenseket összefogja egy szegmensbe, így azok fizikailag egy paragrafusra kerülnek, ugyanarra a szegmens-báziscímre hivatkoznak.

c) Aktív szegmensek kijelölése

A MASM egyik leghasznosabb szolgáltatása annak ellenőrzése, hogy a megcímezni kívánt adatok és utasítások elérhetőek-e az éppen fennálló szegmensregiszter-tartalmak mellett. Azonban ahhoz, hogy ezt a szolgáltatást igénybe vegyük, meg kell mondanunk, hogy egy szegmensregiszterbe éppen melyik szegmens címét töltöttük. Erre szolgál a most ismertetendő

ASSUME

operátor. Használatának szintaxisa:

ASSUME szg-reg:szg-név[, szg-reg:szg-név]

Ezzel az operátorral adja meg a programozó, hogy melyik szegmenst éppen melyik szegmensregiszterhez asszociálja. Maga az operátor nem készíti elő a szegmensregisztert az adott szegmens megcímzésére. Arra szolgál csupán, hogy a programozó "közölje" a fordítóval, hogy (újabb ASSUME kiadásáig) melyik szegmensben kell keresni azokat a szimbólumokat, amelyekre

hivatkozik. A fordító pedig ellenőrzi, hogy a hivatkozott szimbólumok valóban a megnevezett szegmensben vannak-e, és ha nem, akkor hibaüzenetet küld.

Az ASSUME megadása nem kötelező, de ha elhagyjuk, minden szimbólumhivatkozásnál "szg-reg:" operátorral specifikálni kell azt a szegmensregisztert, melyben (terveink szerint) a szegmens kezdőcíme van. Használata tehát ajánlott!

Minden adatcímzés, ha nem használjuk a szegmensregiszter megadására a prefixumot, a DS szegmensregisztert használja. Tehát a DS-hez mindig azt a szegmenst kell asszociálnunk, amelynek kezdőcíme benne van.

Ha például több adatszegmensünk van, melyek neve DAT1, DAT2 és DAT3, és egy ASSUME-al megadtuk, hogy a DS regiszterrel ezentúl a DAT3 szegmenst akarjuk megcímezni, akkor a fordító hibaüzenetet küld minden olyan adatcímzésre, amely a DAT1-ben vagy DAT2-ban elhelyezett adatokra irányul és nem írtuk elő valamely szegmensregiszter használatát az adat neve előtt.

Lássuk a példát:

```
DAT1    SEGMENT PARA    PUBLIC  'DATA'

V_BYTE1         DB      0
V_WORD1         DW      0

DAT1    ENDS

DAT2    SEGMENT PARA    PUBLIC  'DATA'

V_BYTE2         DB      0
V_WORD2         DW      0

DAT2    ENDS

DAT3    SEGMENT PARA    PUBLIC  'DATA'

V_BYTE3         DB      0
V_WORD3         DW      0

DAT3    ENDS
```

Lássunk most ehhez egy kódszegmens részletet, amelyben minden jó és rossz példát, amennyire csak lehet, felsorolunk. Fel kell

tételeznünk, hogy az első itt megadott utasítás előtt a DS regisztert nem készítettük elő, tartalma ugyanaz, mint a program belépésekor volt.

```
CODE    SEGMENT PARA    PUBLIC  'CODE'


        MOV     AX, V_WORD3     ;Hibás, nincs aktív szg. DS-hez

        MOV     AX, DS:V_WORD3  ;Rendben, a programozó tudja !
                                ;Valójában nem jó, mert nem
                                ;tudjuk DS hova mutat !
        MOV     AX, DAT3
        MOV     DS, AX          ;DS előkészítése

        MOV     AX, V_WORD3     ;Szintaktikusan hibás, nincs
                                ;aktív szegmens DS-hez
                                ;Jó lenne, mert DS jó !
    ASSUME      DS:DAT3

        MOV     AX, V_WORD3     ;Rendben !
        MOV     AX, V_WORD2     ;Szintaktikusan is hibás, mert
                                ;nem aktív szegmensben van
                                ;Nem is jó, mert DS nem jó
        MOV     AX, DAT2
        MOV     DS, AX          ;DAT2 címe DS-ben

        MOV     AX, V_WORD3     ;Szintaktikusan helyes, de nem
                                ;jó - DS már nem oda címez !
        MOV     AX, V_WORD2     ;Hibás, még mindig nem aktív
                                ;a DAT2 szegmens !

    ASSUME      DS:DAT2

        MOV     AX, V_WORD2     ;Szintaktikusan és címzésben jó

CODE    ENDS
```

Tanulságképpen leszögezhetjük, hogy helyes lenne a DS előkészítését mindjárt a kódszegmens elején elvégezni, majd azonnal az előkészítés után (vagy közvetlenül előtte - ebben is legyünk következetesek) kiadjuk a megfelelő ASSUME-t. Mikor

pedig módosítjuk a DS tartalmát, akkor ezt azonnal jelezzük a fordítónak egy újabb ASSUME kiadásával. A fenti példa jól illusztrálja, hogy ha nem adjuk meg a fordítónak ezt a jelzést, akkor vagy félre fog bennünket vezetni, mert elérhetetlen adat címzésére nem ad hibaüzenetet, vagy olyan esetben is hiba lép fel, amikor fizikailag jó lenne a címzés, pusztán az ASSUME kiadásával késtünk egy kicsit. A helyes kód az alábbi volna:

```
CODE    SEGMENT PARA    PUBLIC  'CODE'


        MOV     AX, DAT3
        MOV     DS, AX          ;DS előkészítése
    ASSUME      DS:DAT3         ;Jelzés a fordítónak

        MOV     AX, V_WORD3     ;Rendben !
                                ;El akarjuk érni V_WORD2-t!
        MOV     AX, DAT2
        MOV     ES, AX          ;ES előkészítése
    ASSUME      ES:DAT2         ;Jelzés a fordítónak

        MOV     BX, ES:V_WORD2  ;Rendben !

        MOV     AX, DAT2
        MOV     DS, AX          ;DAT2 címe DS-ben
    ASSUME      DS:DAT2         ;DAT2 lesz aktív DS-ben

        MOV     AX, V_WORD2     ;Rendben !

CODE    ENDS
```

A CS szegmensregisztert használjuk a kódterület címzéséhez. Minden cimkedeklaráció hibaüzenetet okoz, ha a deklaráció előtt nem adtunk meg egy ASSUME-t a CS regiszterre:

```
CODE    SEGMENT PARA    PUBLIC  'CODE'

START:                          ;Hibás, nincs aktív szegmens !
    ASSUME      CS:CODE
ENTRY:                          ;Helyes

CODE    ENDS
```

Nagyon figyeljünk arra, hogy a szegmensregiszterek beállitását helyesen végezzük el - az ASSUME nem a szegmensregiszter betöltését, csak az éppen érvényesnek tartott szegmensregiszter-szegmens hozzárendelést definiálja!

A kódszegmens elején minden szegmensregiszterhez meg szokás adni, melyik szegmenshez asszociáljuk. Például:

```
ASSUME  CS:CODE, DS:DATA, SS:STACK, ES:NOTHING
```

Az ES: után megadott

NOTHING - semmi

kulcsszót akkor használjuk, ha az adott szegmensregiszterhez (éppen) nem akarunk szegmenst asszociálni.

Egyébként, mint a fentiekből kiviláglik, explicit szegmensregiszter-megadás esetén a fordító nem ellenőrzi az adatok elérhetőségét, hiszen amúgy sem tudhatja, hogy az adott pillanatban a futás közben milyen érték lesz valamely szegmensregiszterben. Mint már hangsúlyoztuk, az ASSUME arra szolgál, hogy a programozó szándékáról informálja a fordítóprogramot. Használata semmiféle garanciát nem jelent arra nézve, hogy a szintaktikusan helyes programok valóban jól fognak címezni, pusztán segíti a program tisztább és valamelyest biztonságosabb megírását.

4.5.2. Adatterület foglalás

Az adatterületeket általában külön szegmensben, az adatszegmensben foglaljuk le. Indokolt esetben azonban a kódszegmensben, az utasítások között is elhelyezhetünk bizonyos adatokat - egyes vezérlési információk logikus helye itt van.

Olyan adatokat szokás kódterületen elhelyezni, amelyek vagy állandóan kellenek (pl. a PSP címe - lásd második kötet) vagy pedig el akarjuk őket "rejteni" mások szeme elől, tehát meg akarjuk akadályozni, hogy egy egyszerű (prefixum nélküli) címzéssel elérhetőek legyenek. Ha ravaszul elhelyezünk néhány (elsősorban ellenőrzésre szolgáló) változót a programkódban, akkor megnehezíthetjük a program megfejtését, disassemblálását.

A program védelmére számos trükk ismeretes; mindenesetre néhány, a programkódban szétszórt gonosz kis változó alaposan megkeseritheti a védelmet feltörni szándékozó életét.

Mint látni fogjuk, a MASM széleskörű lehetőségeket biztosít az adatterületek definiálására. Egyszerű változók mellett definiálhatunk struktúrákat, ún. rekordokat és "tömbváltozókat" is. Ez utóbbi azért szerepel idézőjelben, mert nem valódi tömbkezelésről, hanem tömbszerűen kezelt változóhalmazról van szó; a címzés során használhatunk ugyan indexeket, de indexként csakis konstans értékeket használhatunk.

Egyszerű területfoglalás

DB (Define Byte)

[név] DB kif

"név" néven hivatkozhatunk a lefoglalt byte-ra; "kif" (kifejezés) értékét irja be előkészitésként. ","-vel elválasztva több kifejezést is megadhatunk. Ezek sorban egymás után következő byte-okat foglalnak majd le.

A DB után irhatunk tetszőleges hosszúságú string-et idézőjelek közé zárva. Igy definiálhatunk szövegeket a program számára.

Például:

```
DAT     DB      45D         ;Lefoglal egy byte-ot,
                            ;kezdeti értéke 45 (10)
DATW    DB      10, 20      ;Két byte lefoglalása

CHARS   DB      'a','b','c' ;Három byte, a,b,c
                            ;ASCII-kódjával töltve
TEXT    DB      'This is a text', 13, 0AH, 24H
                ;Szöveg számára helyfoglalás
                ;összesen 17 byte, a vége:
                ; CR, LF és egy dollárjel
```

Megjegyzés: általában nem tartom szerencsésnek a kezdeti értékadást (kivétel persze a szövegek

elhelyezése, ami értelmesen csak így valósítható meg), mert ezek az értékek csak a betöltéskor állnak így be. Ha debug-olás közben az ember javított a programon, felülírva a kódot és, kipróbálandó a változtatást, elölről le akarja futtatni, akkor az előző (és amúgyis hibás) futás során a fordító-inicializálta változók elromlottak. Az újabb futás így biztosan nem lesz sikeres, mert minden változót kézzel inicializálni képtelenség.
Ha minden változót expliciten, futás közben programból inicializálunk, akkor a debug-olás során a programot újraindíthatjuk.
Ha lehet, egyébként ne DEBUG-al javítsunk hibákat, mert a source-ban problematikus minden változtatást precízen keresztülvezetni. Egy-két hibás utasítást azért ki szabad javítani, de célszerű a sikeres próba után azonnal javítani a source-t és újra fordítani. Ellenkező esetben a kódban javított hibákról könnyen elfeledkezünk, hiszen már "lebirkóztuk" őket. A program újabb fordítása után pedig ezek az ősrégi hibák, mint megannyi kísértet, újra felbukkanak, a teljes megtébolyodás szélére sodorva a programozót.

DW (Define Word)

Szó lefoglalására szolgál. Egyebekben pontosan egyezik DB-vel. Szövegdefiniálásra nem alkalmas. Segítségével kényelmesen lehet azonban szimbólumok offsetjét vagy szegmenscímeket elhelyezni.

Például:

```
S1      SEGMENT PARA    PUBLIC  'DATA'

VAR     DB      0       ;Nem igazi inicializálás
AD_VAR  DW      VAR     ;Helyes inicializálás !!
SG_VAR  DW      S1      ;Ez már kétes !

S1      ENDS
```

Megjegyzés: kétes helyességűnek bélyegeztem a második értékadást, mert az alább ismertetett DD operátor biztonságosabban oldja meg ezt a feladatot.

DD (Define Doubleword)

Duplaszó lefoglalására szolgál. Egyebekben pontosan megegyezik DB-vel. Szövegdefiniálásra nem alkalmas. Segítségével könnyen definiálhatunk olyan blokkot, amely egy távoli címet tartalmaz (azaz valamely objektum szegmens:offset alakú címét).

Például:

```
AD_VAR  DD      VAR     ;Leteszi egy 4 byte-os
                        ;blokkba VAR címét
```

Ezt most egy LDS vagy LES utasítás operandusaként használhatjuk:

```
        LDS     SI, AD_VAR
        MOV     AL, [ SI ] ;VAR-t olvassa be !
```

Megjegyzés: a DD operátor alkalmas még kétszeres pontosságú egész és egyszeres pontosságú lebegőpontos változók definiálására is.

DQ (Define Quadword)

Négy szó lefoglalására szolgál. Egyebekben pontosan egyezik a DB-vel. Szövegdefiniálásra nem alkalmas.
A DQ speciális felhasználása: helyfoglalás duplapontosságú lebegőpontos változók számára.

DT (Define Tenbytes)

Tíz byte lefoglalására szolgál. Egyebekben pontosan egyezik a DB-vel. Szövegdefiniálásra nem alkalmas.
A DT speciális felhasználása: helyfoglalás 10 (20) jegyből álló pakolatlan (pakolt) BCD számok számára.

Megjegyzés: az Intel 8087 koprocesszor ismeri a pakolt decimális formátumú változókat. Ezeket lehet a DT operátorral definiálni. A szám persze 19 jegyet tartalmaz, a 20 jegy helyén (a legalacsonyabb című fél byte-on) helyezkedik el a szám előjele.

DUP

("duplicate") - ez az operátor arra szolgál, hogy egy területfoglalást, a kezdeti érték megadásával együtt, megtöbbszörözhessünk:

név op. ism.sz DUP (k.ért.)

A "név" nevű, "op."-al (lehet DB, DW stb.) megadott területfoglalást "ism.sz"-szor megismétli, a "k.ért" kezdeti értéket adva.

Például:

```
ARRAY   DB      20      DUP     ( 16 )
```

egy 20 byte-os "tömböt" foglal le és minden elembe 16-ot ír. A tömb persze sokféleképpen címezhető, de az elemek direkt címzése így is lehetséges:

```
        MOV     AL, ARRAY[ 11 ]
```

amely az ARRAY + 11. byte-ot, azaz a tömb 12. elemét olvassa be AL-be. Az index egy kifejezés lehet, de e kifejezés minden elemének fordítási időben ismertnek kell lennie - ez tehát nem igazi tömbkezelés.
Ha nem akarjuk a lefoglalt tömb elemeit fordítási időben inicializálni, akkor a DUP után "(?)"-et kell írni. A zárójeles kifejezés használata kötelező.

Struktúrák és rekordok

Struktúrák definiálására a következő operátorpár szolgál:

```
STRUC                   - struktúradefiníció kezdete
ENDS                    - struktúradefiníció vége
```

A STRUC pszeudo-operátor jellegét tekintve hasonlít a macro-hoz. Egy adatterület belső felépítésének, struktúrájának

leírására szolgál, de tényleges területfoglalást nem végez; ez a definíció felhasználásakor, "hívásakor" történik meg. Egy struktúra leírható volna macro segítségével is; azonban, mint látni fogjuk, speciális területfoglaló operátor lévén több szolgáltatást nyújt.

```
str-név STRUC
        ...
        ...
v-név   def-op  kif
        ...
        ...
str-név ENDS
```

ahol:

"str-név" a struktúra neve,

"..." jelzi, hogy tetszőleges sok változó definiálható a struktúrán belül, "v-név" a struktúra egy belső változójának, ún. "field"-jének, elemének neve,

"v-név" a struktúra elemének neve, amely a struktúra többi elemétől különbözteti meg az adott elemet (az elemeket az eredeti irodalom "field"-nek, mezőnek is nevezi),

"def-op" lehet DB, DW, DD, DQ, DT, tehát minden "elemi" területfoglaló operátor. Struktúra belsejében nem használhatunk más struktúrát vagy rekordot,

"kif" megszabja az elem kezdeti értékét (csak területfoglaláskor jut érvényre)

"ENDS" mutatja meg a struktúra végét.

Az így definiált struktúrát a fordító megjegyzi, és ha leírjuk a struktúra nevét ("meghívjuk a definiciót"), lefoglalja a kellő hosszúságú memóriát és a definiálásnál és/vagy hívásnál megadott kezdőértékekkel fel is tölti.

A struktúra hívása:

```
név     str-név          < k.ért1, k.ért2,.. >
```

ahol:
"név" a lefoglalt terület neve,
"str-név" a struktúra neve,
"k.ért" a hívásnál megadott kezdőérték.

Egy struktúrát tetszőlegesen sokszor "meghívhatunk". A híváskor adott kezdeti értékek a megadás sorrendjében töltődnek be az egyes elemekbe.

Megjegyezzük, hogy azon elemeknek adhatunk értéket híváskor, melyek külön név alatt, egyedi területfoglalási pszeudo-operátorral definiáltak. Azokat az elemeket, melyeket a DUP pszeudo-operátor segítségével vagy egy direktíva többszöri alkalmazásával foglaltunk le, nem inicializálhatjuk a struktúra hívásakor.

Megengedett egy struktúra hívása a DUP ismételtető operátor alkalmazásával is.

Példa:

A következő struktúra egy directory bejegyzés befogadására szolgál (a directory bejegyzés ismertetését lásd a második kötetben):

```
DIRENT          STRUC

F_NAM   DB      8  DUP ( ? )     ;File név
F_TYP   DB      3  DUP ( ? )     ;File típus
F_ATTR  DB      ?                ;Attributum
F_R1    DB      10 DUP ( ? )     ;Foglalt
TIM_LW  DW      0                ;Ut.írás ideje
DAT_LW  DW      0                ;Ut.írás dátuma
F_R2    DB      2  DUP ( ? )     ;Foglalt
F_SIZE  DB      4  DUP ( ? )     ;File hossz

DIRENT          ENDS
```

Mint látható, minden elemnek külön neve van a struktúrán belül. Az elemek különböző típusúak és

viszonylag könnyen tudunk majd megcímezni minden változót.

Hívás (területfoglalás a struktúra számára):

```
DIR1    DIRENT  < , , 020H >
```

Ez egy olyan directory bejegyzés, amelyben az "attributum-elemet" előre "archive"-ra állitottuk be (lásd második könyv). A DUP-pal foglalt elemek híváskor történő inicializálása nem lehetséges.

```
DIRA    DIRENT  20      DUP     <>
```

húszelemű struktúratömböt foglal le, amelynek minden eleme egy directory bejegyzés befogadására képes. A "<>" mutatja, hogy a tömb elemeit egyáltalán nem inicializáltuk.

Hivatkozás a struktúra field-jeire (elemeire):

```
név.elem
```

ahol:

"név" a struktúraterület lefoglaláskor adott neve,
"elem" az elem struktúradefiniáláskor adott neve.

Példa: felidézve az előző struktúrát, tegyük fel, hogy beolvastuk a lemez directory-jának egy bejegyzését a DIR1 néven lefoglalt memóriaterületre. Olvassuk be a file hosszát a (DX:AX) duplaszóba (például egy osztási utasitás előkészitéseként):

```
        MOV     DX, DIR1.F_SIZ + 2      ;Felső és
        MOV     AX, DIR1.F_SIZ          ;alsó szó
```

Tegyük fel most, hogy a directory 20 bejegyzését olvastuk be a DIRA néven lefoglalt struktúratömbbe. Vigyük most be a DIRA tömb 12. elemének "idő" elemét AX-be!

```
        MOV     AX, DIRA[ 12 ].TIM_LW
```

Olvassuk ki a 6. tömbelemből a file típusának zárókarakterét AL-be!

```
MOV     AL, DIRA[ 6 ].F_TYP[ 2 ]
```

Figyelem: a tömbök kezdőelemének indexe 0!

Megjegyzés: figyeljük meg a C nyelvvel való hasonlóságot mind a struktúra-, mind pedig a (látszólagos) tömbkezelés szintaxisában!

A hivatkozás során a struktúra kezdőcímét betölthetjük valamely címzésre alkalmas regiszterbe (pl. BX-be), s ekkor a következőképpen hivatkozhatunk rá:

```
[ BX ].elem
```

vagy

```
elem.[ BX ]
```

Innen látható, hogy az elem neve nem más, mint a struktúra kezdetétől vett távolsága byte-okban, azaz egy offset, eltolás. Ezt használjuk ki az indexelt címzés során.

Példa:

```
S       STRUC
FIELD1  DB      1, 2    ;Nem módosítható
FIELD2  DB  10  DUP (?) ;Nem módosítható
FIELD3  DB      5       ;Módosítható
FIELD4  DB      'ZSOFI' ;Módosítható
S       ENDS
```

E példában elsősorban a híváskor történő módosítás lehetőségét figyelhetjük.
Az első elem (kétszeresen használt DB) híváskor automatikusan kap kezdeti értéket, melyet < ... > -vel azért nem módosíthatunk, mert többszörösen használtuk fel a DB operátort. A második elem tartalma határozatlan, de híváskor sem adhatunk kezdeti értéket, mivel a DUP operátort használtuk. A

harmadik és negyedik elem híváskor automatikusan kap kezdeti értéket, de < ... > szerkezet segítségével ezt módosíthatjuk is.

Hívás:

```
DBAREA  S    < , , 7, 'KATA'>   ;3. és 4. elem
                                ;felülírva!
```

Hivatkozás:

```
        MOV     SI, OFFSET DBAREA
        MOV     AL, [SI].FIELD3 ;AL <- 3. elem

        MOV     AL, FIELD3.[SI] ;Egyenértékű

        MOV     AL, DBAREA.FIELD2
```

Figyeljük meg, hogy ".elem" vagy "elem." az elem struktúrán belüli offsetjét, azaz a struktúra kezdetétől vett távolságát jelenti!

RECORD

A RECORD arra szolgál, hogy egy egybyte-os vagy egyszavas "struktúrát" definiáljunk, melynek elemei egy vagy néhány szomszédos bitből állnak. Ezeket az elemeket az eredeti terminológia szigorúan (és értelemszerűen!) field-nek nevezi, ennek megfelelően a továbbiakban mi mezőnek hívjuk. A rekord különösen hasznos driverek (perifériát, illetve perifériakontrollert vezérlő programok) írásakor, ha definiálni akarjuk a vezérlő portra kiírandó utasításokat, ahol minden bitnek (bitcsoportnak) külön jelentése van.

Megjegyzés: sok helyen azt is a rekord előnyének mondják, hogy "helyet takarít meg", mert több független információt adhatunk meg egy szóban. Ez az állítás eléggé meglepő, ha arra gondolunk, milyen sok utasításra van szükség ahhoz, hogy egy rekord valamelyik mezőjét "kibányásszuk" és elkülönítsük a többitől!

Lássuk most a REKORD operátor szintaxisát!

```
rk-név  RECORD  mező:szélesség[=kif][,...]
```

ahol:

rk-név	a rekord-struktúra neve
mező	a mező neve
szélesség	a mező szélessége bitekben
kif	kezdeti érték

A rekord definíciója nem jelent területfoglalást, az csak híváskor valósul meg. A hívás során a struktúra hívásához hasonlóan adhatunk értéket az egyes mezőknek.

A mezőnév konstansként szerepel a MASM szimbólumtáblájában, és értéke megegyezik a mezőnek a szó jobb szélső pozíciójától vett távolságával. Ha ezt az értéket CL-be töltjük, akkor egy jobbra rotálási utasítás a mezőt a jobb szélső pozícióba tolja.

Ha a mező neve elé írjuk a MASK operátort, ez a szimbólum azt a bit-mintát jelenti, amelynek segítségével a mezőt (eredeti helyén) egy AND utasítás kiadásával "elkülöníthetjük" a byte (szó) egyéb bitjeitől, azaz a többit nullázhatjuk.

Példa:

```
AREA    RECORD  X:3, Y:4, Z:9   ;Deklaráció

TABLE   AREA    10 DUP (< 0, 2, 255 >)  ;Definíció
                        ;Lefoglaltunk 10 rekordot
                        ;értékeik: 04FF hexa

        MOV     DX, TABLE[ 2 ]  ;Felolv. a 3.-at
        AND     DX, MASK Y      ;kimaszkoljuk Y-t
                                ;MASK Y = 1E00
                                ;DX értéke 0400
        MOV     CL, Y           ;Rotálási szám
        SHR     DX, CL          ;Y a jobb szélre
                                ;DX = 2
```

4.6. A szimbólumdefiníció egyéb módjai

4.6.1. Konstansok definiálása

A konstansok definíciója két operátorral lehetséges, az "="-vel és az EQU pszeudo-operátorral.

```
EQU             - konstansdeklaráló operátor

név     EQU     kif

ahol:
név             a konstans neve (ezzel hivatkozhatunk rá),
kif             a konstans értéke.
```

Az EQU-val definiált konstansok értéke nem változtatható meg.

```
=               - konstansdelkaráló operátor

név     =       kif

ahol:
név             a konstans neve,
kif             a konstans értéke.
```

Az "="-vel definiált konstansok újabb "="-vel újradefiniálhatók. A konstansnév így más-más számot "ér" a source file különböző részein (lásd még: REPT operátor).

Példák:

1. példa:

```
ABC     EQU     25

        MOV     AX, ABC         ;Egyenértékű a
        MOV     AX, 25          ;utasítással

ABC     EQU     30              ;Hibás
```

Az első konstansdefiníció egyszer és mindenkorra lefoglalja az ABC szimbólumot és 25-öt rendel hozzá. A második definiálási kísérlet ezért hibaüzenetet okoz.

2. példa:

```
X       =       0
        DB      X       ;DB   0 -val azonos
X       =       65H     ;X újradefiniálása
        DB      X       ;DB   65H -val azonos
X       =       X + 2
        DB      X       ;DB   67H -el azonos
```

3. példa:

```
X       =       100
                ;Itt X megváltoztatható
X       EQU     200     ;Legális utasítás
                ;X a továbbiakban nem módosítható:
X       =       300     ;Illegális utasítás
```

4.6.2. Címkedeklaráció

LABEL

A LABEL-el definiálhatunk egy címkét, és mindhárom attributumát megszabhatjuk:

```
név     LABEL   típus
```

ahol:

név	a definiálandó címke
típus	a címke típusa. Ez lehet BYTE, WORD, DWORD, QWORD, TBYTE, struktúranév és rekordnév (a STRUC kulcsszó előtt, a struktúra definiálásakor adott nevet kell használni), vagy NEAR vagy FAR, mint egy belépési pont neve).

Az így definiált cimke szegmense az, amelyben definálják, offsetje pedig megegyezik a legelső, utána következő területfoglaló utasítás offsetjével, vagyis a helyszámláló (lásd 4.7.) pillanatnyi értékével.

A LABEL nem használható struktúradefiníció belsejében, mivel az nem foglal le területet.

Példák:

```
1.
ENTRY   LABEL   FAR
ENTRY1:

        JMP     ENTRY   ;Távoli ugrás lesz
        JMP     ENTRY1  ;Rövid ugrás lesz

2.

VARW    LABEL   WORD    ;Szavas típusú változónév
VAR     DB      0, 0    ;Két byte lefoglalása


        MOV     AX, VAR     ;Hibás, eltérő típusok
        MOV     AX, VARW    ;Legális, mindkét
                            ;byte-ot mozgatja
        MOV     AL, VAR         ;E két utasítás
        MOV     AH, VAR + 1     ;hatása megegyezik
                                ;az előzővel

        LDS     BX, VAR     ;Hibás, eltérő típusok
        LDS     BX, VARW    ;Hibás, eltérő típusok
                            ;LDS duplaszót vár
```

4.6.3. Eljárásdeklaráció

Az eljárások (angol nevük "procedure") megfelelnek a szubrutinoknak. Minden szubrutint az alább ismertetett két operátor (PROC és ENDP) között kell megadni. Az eljárás a programok kódolásának egyik legfontosabb eszköze. Ennek

ellenére az eljárásra vagy szubrutinra nem tudok jobb (egyszerű) definiciót adni, mint a következőt: egy eljárás olyan utasítások sorozata, amelyet a RET utasítás zár le, s amelyet ebből adódóan a CALL utasítással aktivizálunk.

```
PROC                    - eljárás kezdete
ENDP                    - eljárás vége
```

Az eljárásdeklaráció tulajdonképpen szubrutinok típusának definiálására szolgál. Azt szabja meg, hogy milyen típusú a rutinban elhelyezett (esetleg több) RET utasítás. A rutin hívásának ezzel egyezőnek kell lennie!

```
név     PROC    {[NEAR], FAR}
        ...
        ...                     ;Az eljárás utasításai
név     ENDP
```

ahol:

név — az eljárás neve (tulajdonképpen cimke)

A NEAR operátor alkalmazása nem kötelező, ezt jelzi [...], a kapcsos zárójel azt érzékelteti, hogy a felsorolt kulcsszavak valamelyikét használhatjuk.

Példa: az alábbi eljárás a (DX:AX) duplaszóhoz adja a (CX:BX) duplaszó tartalmát. Az előbbi egy szorzás eredménye (lásd 3. fejezet, MUL utasítás), az utóbbi pedig egy másik kétszeres pontosságú érték. Visszatéréskor az összeg (DX:AX)-ben lesz. A szubrutint, feladatára utalva DBADD-nak, azaz "DouBle precision ADDition"-nak nevezzük el:

```
DBADD   PROC    NEAR

        ADD     AX, BX          ;Alsó szavak
        ADC     DX, CX          ;Felső + átvitel
        RET                     ;Kész, vissza!

DBADD   ENDP
```

Ez a rutin a

```
CALL    DBADD
```

utasítással hivható meg ugyanebből a szegmensből. Más programszegmensből ezt a rutint (lévén típusa NEAR) nem hívhatjuk meg.

Az eljárásdeklaráció magában véve nem hajt végre semmilyen vezérlésátadást, az eljárásokból való visszatérést nekünk kell biztosítanunk a RET assembly utasítás kiadásával.

Megjegyzés: a processzor utasításkészlete nem teszi lehetővé, hogy egy eljárást rövid és hosszú CALL-al is meghívjunk, hiszen a RET utasítás "be van építve" az eljárásba.
A MASM természetesen "vigyáz" erre. Ha egy eljárást egyszer "közelinek" deklaráltunk, akkor a hívások mind közeliek lesznek, és más programszegmensekből való (kényszerűen távoli) hívási kísérlet hibajelzést vált ki. Ha az eljárást "távoliként" deklaráltuk, akkor minden hívás automatikusan távoli lesz.
Ez az utóbbi lehetőség mégse csábítson senkit arra, hogy minden eljárását automatikusan távoliként deklarálja. A szegmensek rendszere (minden bonyodalmassága mellett) arra kényszerít bennünket, hogy a programunkat (jól) moduláljuk, azaz vágjuk szét több logikai egységbe. Célszerűen ezek lesznek az egyes programszegmensek. A szegmensen belüli vezérlésátadás így modulon belüli is lesz, tehát "belügy", a szegmensközi vezérlésátadás pedig egyúttal modulok közötti is, tehát "globális" eljárás meghívása. Ezért a szegmentálás arra is alkalmas, hogy az egyes modulok "belügyeit" elrejtse a többi modul elől, megakadályozva az illetéktelen hívásokat. Ha minden eljárásunk távoli, azaz "globális" lesz, akkor kényelmesen hívhatjuk meg valamennyit, de elesünk a logikai védelem lehetőségétől. Azt javaslom tehát, hogy kerüljük ezt a nem éppen szerencsés gyakorlatot és mindig vállaljuk a gondos programtervezéssel járó látszólagos pluszmunkát. Ha nem öncélúan igyekszünk szebbé tenni a programot,

hanem gondosan követjük annak saját belő logikáját, akkor a végeredmény sokkal biztonságosabb és kissé gyorsabban futó program lesz – ne felejtsük el ugyanis, hogy a távoli szubrutinhivás még lassúbb is, tekintve, hogy egy helyett két szót kell a stack-re, tehát memóriába irnia, a visszatéréskor pedig olvasnia.

Az eljárások definíciói egymásba ágyazhatók. Ezzel azt érhetjük el, hogy a "külső" eljárásnak több belépési pontja is lehet.

Példa: bővitsük az előző szubrutint a (CX:BX) duplaszó betöltésével, de úgy, hogy megtartjuk a már előkészített duplaszavak összeadását is! Az előkészítéssel kombinált rutin neve DBADDIN (az angol DouBle precision ADDition & INitialization elnevezésből). Bemeneti adatai a (DX:AX) duplaszó (amely feltevésünk szerint egy szorzás eredménye), valamint a másik összeadandó címe az SI regiszterben. A beágyazott rutin neve DBADD mint fent, és specifikációja változatlan.

```
DBADDIN PROC    NEAR

        MOV     BX, 0[ SI ]     ;Alsó szó be
        MOV     CX, 2[ SI ]     ;Felső szó be
                                ;Előkészítés kész
DBADD   PROC    NEAR
        ADD     AX, BX          ;Alsó szavak
        ADC     DX, CX          ;Felső + átvitel
        RET                     ;Kész, vissza!
DBADD   ENDP

DBADDIN ENDP
```

Most tehát meghivhatjuk a DBADDIN rutint, ha előzőleg SI-be beolvastuk a másik összeadandó címét, de meg tudjuk hívni közvetlenül DBADD-ot is, ha a másik tényező már rendelkezésünkre áll a (CX:BX) regiszterpárban.
Figyeljük meg, hogy a két rutin típusa megegyezik. Vigyáznunk kell a helyes beágyazásra is. A "belső"

rutint mindenképpen le kell zárni, mielőtt a "külsőt" lezárnánk. Ha felcseréljük ezt a két sort, a fordító goromba hibaüzeneteket küld.
Mint már megemlítettem, a leghelyesebb gyakorlat, ha egy blokknak egy belépési és egy kilépési pontja van. Ezt a tanácsot most könnyen alátámaszthatjuk a következőkkel. A DBADDIN rutin elrontja a BX és CX regisztereket, holott ehhez nincs joga, hiszen azok nem tartalmaznak sem bemenő, sem pedig válaszparamétereket. Semmi baj, a PUSH utasítással elmenthetjük a regiszterek tartalmát, majd visszatérés előtt két POP-al... igen, de ekkor a DBADD rutin többé nem hívható meg önállóan, mert a végén (a két rutinnak közös RET utasítása van) két felesleges POP utasítás van!
A fenti problémát kulturált módon az alábbiak szerint lehet megoldani:

```
DBADDIN PROC    NEAR

        PUSH    CX
        PUSH    BX              ;Mentés

        MOV     BX, 0[ SI ]     ;Alsó szó be
        MOV     CX, 2[ SI ]     ;Felső szó be
                                ;Előkészítés kész
        CALL    DBADD           ;összeadás

        POP     BX
        POP     CX              ;Helyreállítás
        RET

DBADDIN ENDP
```

Természetesen lassúbb lesz a végrehajtás, a kód valamivel hosszabb (van benne egy pótlólagos CALL és egy RET, ami összesen 4 byte) viszont a rutinok teljesen "tisztességesek" - illetéktelenül nem rontanak el egyetlen regisztert sem. Ez pedig nem lebecsülendő előny!

4.6.4. Globális szimbólumok

Ha egy programot több file-ba (modulba) osztva írunk meg, akkor bizonyos szimbólumokat "láthatóvá" kell tennünk minden modul számára. Erre szolgál a PUBLIC pszeudo-operátor, melyet azon file elején kell kiadni, melyben a szimbólumot definiáltuk:

```
PUBLIC  szimb1[, szimb2, ... ]
```

Hatására a felsorolt szimbólumok "láthatóvá válnak" más modulok számára is, azaz a fordító továbbadja ezeket a szimbólumokat a szerkesztőnek, hogy az kitölthesse azokat a helyeket, amelyeken más modulokban, más file-okban hivatkozunk ezekre a szimbólumokra.

Abban a modulban, amelyben más modulban definiált szimbólumokra kívánunk hivatkozni, az EXTRN pszeudo-operátort kell használnunk. A fordító minden egyes szimbólumhivatkozást ki szeretne tölteni; az EXTRN-el külső szimbólumnak deklaráltakra való hivatkozásokat definiálatlanul hagyja. Ezeket majd a LINK fogja feloldani, kitölteni más file-okban PUBLIC-ként megadott szimbólumokkal.

```
EXTRN   szimb1:típus[, szimb2:típus, ... ]
```

ahol

"szimb1", stb. a külső szimbólumként deklarált szimbólum neve,

"típus" a szimbólum típusa. A típus lehet BYTE, WORD, DWORD stb, vagy rutinok esetén NEAR és FAR.

Megjegyzés: ha nagyobb lélegzetű programot kívánunk írni, akkor célszerű több modulra (forrásnyelvű file-ra) felbontani, hogy lehetőleg rövidebb file-okat kelljen editálni, listázni stb. Viszont a szétvágást alapos megfontolással úgy célszerű megtervezni, hogy logikailag is különböző részekre vágjuk fel a programot, amelyek egymással a lehető legkevesebb ponton találkoznak (egy belépési pont, egy paraméterblokk címe és ... semmi több), tehát a

lehető legkevesebb legyen a globális szimbólumok száma. Figyeljünk arra is, hogy EXTRN-el az adott file-ban csak azokat a szimbólumokat definiáljuk külsőként, amelyekre valóban hivatkozunk (s amelyekre jogosan hivatkozunk).
Lásd még az eljárásdeklaráció idevágó megjegyzéseit.

4.7. A helyszámláló és módosítása

A fordító a programszöveg olvasása közben meghatározza az egyes programsorok (utasítások) helyigényét, és egy változóban tárolja a pillanatnyi offset-et, vagyis az aktuális szegmens kezdetétől vett távolságot byte-okban számolva.

Megjegyzés: természetesen minden szimbólum offsetje e változó pillanatnyi értéke.

Ezt a változót hívjuk helyszámlálónak, és a programban a "$" (dollár) jellel hivatkozhatunk rá. A helyszámlálót az ORG pszeodo-operátor segítségével állíthatjuk át:

```
ORG     kif
```

hatására a programkód további részét a "kif" értékével megegyező címtől kezdve helyezi el a fordítóprogram, azaz áthelyezi a helyszámlálót (ez persze általában területátlépéssel jár). A CP/M operációs rendszerből származó programozói gyakorlat szerint a programkód kezdetét általában a 100H címre teszik.

Megjegyzés: ez szükséges is abban az esetben, ha .COM típusú file-á kívánjuk átalakitani az elkészült .EXE típusú file-t (a .COM alak egy CM/P-szerű programformátumot jelent, melyben a program teljes egészében egy szegmensen belül helyezkedik el, és az alsó 100 pozíción egy CP/M-hez hasonló struktúra definiált - pl. a virtuális 0. pozíción egy INT 20 utasítás van - kilépés a programból, a virtuális 5.

pozició egy, a DOS-függvények belépési pontjára ugró hosszú CALL található, és 5CH-tól rendelkezésünkre áll egy File Control Block), lásd második könyv: a DOS táblázatai - Program Segment Prefix.
Megjegyezzük még, hogy az így átlépett terület nem foglal helyet az object file-on, eltérően a területfoglaló operátorok által lefoglalt területtől.

Példa:

```
ORG     120H    ;A program kódolása a 120. byte-
                ;tól folytatódik

ORG     $+2     ;Két byte kimarad

CSEG    SEGMENT PAGE

BEGIN   = $     ;A BEGIN szimbólum értéke
                ;$ lesz


IF ($-BEGIN) MOD 256    ;Ha nincs laphatáron

        ORG     ($-BEGIN)+256-((($-BEGIN) MOD 256)
                        ;Kiigazitja BEGIN-t
ENDIF
```

(Lásd még 4.9, feltételes fordítás)

Visszafelé is léptethetjük a helyszámlálót. Ez lehetőséget nyújt arra, hogy ugyanazt az adatterületet többféleképpen felosztva is definiálhassuk.

Példa:

```
BYTBL   DB      100     DUP     ( ? )

$ = $ - 100

WRDBL   DW       50     DUP     ( ? )
```

Ekkor BYTBL és WRDBL offsetje megegyezik, ugyanazt a területet jelentik mindketten, de BYTBL elemeit byte-os, WRDBL elemeit pedig szavas műveletekkel érhetjük el.

4.8. Formátumvezérlő operátorok

Rendkívül fontos, hogy ha a kezünkbe veszünk egy listát, azonnal lássuk, mi az, hamar tudjunk benne tájékozódni és a lista lehetőleg úgy nézzen ki, hogy ne érezzünk iránta mindjárt leküzdhetetlen ellenszenvet. Ez az alfejezet azokat a fontos formátumvezérlő operátorokat ismerteti, amelyek segítségünkre lehetnek e cél elérésében.

Ezek az operátorok általában csak a file elején használhatók, és általában a listafile formátumát, külső megjelenését szabályozzák, "programozási" hasznuk nincs.

NAME modulnév

Nevet ad az adott modulnak (file-nak). Ez egy belső név, amelyet azonban a fordító átad a LINK-nek, és szerepet játszhat egy esetleges overlay-struktúra megadásában. A NAME operátor hiányában a TITLE operátorral megadott nevet, ha az sincs, akkor a file nevének első 6 karakterét használja fel modul-névnek a rendszer.

(TITLE - lásd 4.11.)
(SUBTTL - lásd 4.11.)

.RADIX kif

Meghatározza az alapértelmezett számrendszert. A RADIX használata esetén is eltérhetünk az alapértelmezett számrendszertől egy konstans megadása során a konstans után írt B, O, D, vagy H betűkkel (lásd 4.2.)

COMMENT határolójel ... szöveg ... határolójel

Arra szolgál, hogy egy hosszabb megjegyzésblokkot pontosvesszők használata nélkül iktathassunk be a program elején.

határolójel — tetszőleges karakter (a szövegben nem használható!)

szöveg — a megjegyzés tetszőlegesen hosszú szövege

INCLUDE file-spec

Szövegfile "beemelésére" szolgál (nem csak file elején). Ahol kiadjuk, beolvassa a megadott file-t és hozzáfordítja a szövegünkhöz.

Ezzel tulajdonképpen forrásnyelvű könyvtárakat tudunk definiálni és felhasználni. Egy include file bármit (tehát konstansdeklarációkat, macro-kat, de kész szubrutinokat) is tartalmazhat.

Megjegyzés: az 1.00 verzió "nem szereti" az include file-okat, mivel a programozó szívesen zsúfolja tele ezeket mindenféle konstans- és macrodefinícióval. Ekkor pedig a MASM igen hamar azt mondja, hogy "Out of Memory" - nincs elég memória. Ez az üzenet legalábbis meglepő egy 640 Kb-os gépen futó programtól. Jómagam olyan file fordítása közben szaladtam rá erre a hibára, amely "csatolt részeivel" (tehát az include file-okkal, a teljes object és lista file-al) együtt sem volt hosszabb 50 Kb-nál. Minden kísérletem kudarcba fulladt. Megpróbáltam a konstansdefiníciókat macrokba csoportosítani, majd a felhasználás után a PURGE operátorral törölni a macro-kat. Ez egy darabig segített, de egy picivel hosszabb program fordítása közben azonnal újra előjött a hiba - és én, feladva a reménytelen küzdelmet, megszüntettem az include file-okat, bemásolva a szükséges deklarációkat a

file-ba.
Mi okozza ezt a hibát? Az igazat megvallva nem tudom. Az alap-ok a fordító szimbólumtáblájának végzetes betelése. Azt hiszem, hogy ez azért van, mert a fordító a minimális memóriakiépítésű gépre számít és nem vizsgálja meg, mekkora memória áll a rendelkezésére a valóságban.
A végső megoldást a 3.00 verziójú fordítóra való átállás hozta meg, azóta bátran használom a leghosszabb include file-okat is.

4.9. Feltételes fordítási operátorok

A feltételes fordítási operátorok arra szolgálnak, hogy különböző szövegblokkokat figyelmen kívül hagyhassunk. Például, ha egy programot úgy alakítunk ki, hogy többféle rendszerben is futhasson, a rendszerfüggő szakaszokat feltételes blokkba helyezzük el, és mikor egyik rendszerből átvisszük a másikba, más feltételeket állítunk be. Pl.:

```
IFDEF   MSDOS           ;Ha MSDOS szimbólum definiált
        ...             ;MS-DOS specifikus kód
ENDIF

IFDEF   CPM             ;Ha CPM szimbólum definiált
        ...             ;CP/M specifikus kód
ENDIF
```

és ha MS-DOS alá szánjuk a programot, az MSDOS szimbólumot definiáljuk és CPM-et nem, így csak az MS-DOS specifikus kód fordul. Amikor pedig CP/M alatt szeretnénk lefordítani, CPM-et definiáljuk és MSDOS-t nem, ekkor csak a CP/M specifikus kód fordul, a másik nem.

Ezzel mindjárt meg is adtuk egy feltételes blokk formáját: egy feltétellel kezdődik, szerepelhet benne egy opcionális ELSE ág is (az a blokk akkor fordul, ha a feltétel nem teljesül) és egy ENDIF zárja le:

```
IFxx    feltétel
        ...             ;Fordul, ha "feltétel" teljesül
[ELSE]
        ...             ;Fordul, ha nem teljesül
ENDIF
```

Feltételes fordítási operátorok

IF kif

Igaz, ha "kif" értéke nem 0.

IFE kif

Igaz, ha "kif" értéke 0.

IF1

Igaz a fordítás első menete alatt.

IF2

Igaz a fordítás második menete alatt.

IFDEF szimbólum

Igaz, ha "szimbólum" definiált.

IFNDEF szimbólum

Igaz, ha "szimbólum" nem definiált.

IFB <argumentum>

Igaz, ha "argumentum" blank.

IFNB <argumentum>

Igaz, ha "argumentum" nem blank.

IFIDN <arg-1>,<arg-2>

Igaz, ha az "arg-1" string megegyezik az "arg-2" stringgel.

4.10. Macrodefiniálás és blokkismétlés

A macro olyan előre definiált utasítássorozatot jelent, amelyet a program tetszőleges helyére beiktathatunk a macro nevének és argumentumainak leírásával. Ezt nevezzük a macro meghivásának. A fordító ekkor egyszerűen bemásolja az utasítássorozatot, a macrodefinicióban megadott paraméterek helyére behelyettesítve a hívási argumentumokat.

A macro definiálása a MACRO és az ENDM pszeudo-operátorok segítségével történik. A macro törzsét azok a sorok alkotják, melyek a két operátor között vannak. A macro nevét és paraméterlistáját a MACRO operátor sorában adjuk meg.

név MACRO [par1][, par2, ...]

Macro definiálása. "név" a macro neve, "par1",... a paraméterek. A MACRO sorától az ENDM soráig leirt sorok alkotják a macro törzsét. Ebben használhatjuk a paramétereket, melyek csak a macro definicióban érvényes szimbólumok. A macro meghivása során a fordító bemásolja a kódba a macro törzsét, a paraméterek helyére pedig behelyettesiti (betűről betűre) a híváskor megadott argumentumokat.

ENDM Lezárja a macro törzsét.

EXITM Ha a fordító ezzel az operátorral találkozik, befejeződik a macro másolása. Ezt az operátort egy feltételes blokkban célszerű elhelyezni.

&

Lehetővé teszi, hogy az argumentumot hozzáfűzzük egy, a macro törzsében definiált, nem paraméterjellegű szöveghez (lásd a 2. példát).

;;

Megjegyzés elhelyezésére szolgál a macro-ban. Ez a megjegyzés nem fog megjelenni a listázás során.

!

A felkiáltójel utáni karaktert literálisan értelmezi a fordító. Ezt az operátort arra használhatjuk, hogy valamilyen speciális macro karaktert, például az "&" jelet helyezzük el valahol a macro szövegében.

%

Ez az operátor a következő paramétert számmá konvertálja (lásd a 2. példát).

PURGE macro-név

Törli a macro-k közül a megadott nevűt. Megadása nem szükséges, ha újra akarunk definiálni egy macro-t; az újabb definíció egyszerűen felülírja a régit. Tehát ez látszólag egy luxus operátor. Jelentősége akkor van, ha attól tartunk, hogy a fordító szimbólumtáblája túlzottan betelik.

LOCAL par1[, par2, ...]

A LOCAL pszeudo-operátor arra szolgál, hogy a felsorolt paramétereket a macro-n belül lokálissá tegye (lásd 3. példát).

REPT kifejezés

A REPT operátor a MACRO-hoz hasonlóan egy blokk definiálására szolgál, de a definiált blokkot annyiszor másolja be a programkódba, ahányszor "kifejezés" előírja (lásd a 4. példát). A REPT blokkot ENDM-el kell lezárni.

IRP par, <argumentumlista>

Az IRP operátor az általa definiált blokkot annyiszor ismétli meg, ahány eleme van az argumentumlistának. Az első másolásnál az argumetumlista első, a másodiknál a lista második, sít. elemét írja be "par" minden előfordulási helyére. Az IRP blokkját ENDM-el kell lezárni. Lásd az 5. példát!

IRPC par, string

Az IRPC operátor az általa definiált blokkot annyiszor ismétli meg, ahány karaktere van a stringnek. Az első másolásnál a string első, a másodiknál a második, sít. karakterét írja be "par" minden előfordulási helyére. Az IRPC blokkját ENDM-val kell lezárni. Lásd a 6. példát!

Példák

1. példa:

```
GEN     MACRO   XX, YY, ZZ
        MOV     AX, XX
        ADD     AX, YY
        MOV     ZZ, AX
        ENDM

Hivás:
        GEN     ELSO, MASODIK, OSSZEG
                                ;ELSO-t bemásolja AX-be,
                                ;MASODIK-at hozzáadja és
                                ;OSSZEG címre kiírja
                                ;(direkt címzés)
        GEN     DX, 35, [BX]    ;DX-t AX-be, hozzáad 35-öt
                                ;és kiírja a BX által meg-
                                ;címzett szóba
```

2. példa:

```
MAKERR  MACRO   X
LB      =       0
        REPT    X
LB      =       LB+1
        MAKLIB  %LB
        ENDM            ;REPT vége
        ENDM            ;MACRO vége

MAKLIB  MACRO   Y
ERR&Y   DB      'ERROR &Y', 0
        ENDM
```

Hívás:

```
        MAKERR  2
```

Létrehozott kód:

```
ERR1    DB      'ERROR 1', 0
ERR2    DB      'ERROR 2', 0
```

Ez jó példa a macro-k egymásba skatulyázására, a macro-nak macro-ból való hívására és a "&" és "%" operátorok használatára.

3. példa:

A BX által címzett string többszöri kiírása

```
MULMSG  MACRO   TT
        LOCAL   AGAIN           ;A cimke lokális!
        MOV     CX, TT
        MOV     AH, 9           ;A kért DOS-függvény száma
AGAIN:  MOV     DX, BX
        INT     21              ;A DOS hívása
        LOOP    AGAIN           ;Ism. CX-szer
        ENDM
```

Ezzel az AGAIN szimbólumot minden híváskor definiálnánk. Ez többszörösen definiált szimbólum lenne, ha nem használtuk volna

a LOCAL operátort, amely jelzi a fordítónak, hogy ez a szimbólum csak a maro belsejében él. Hívás:

```
        MULMSG  10              ;10-szer irja ki az üzenetet
```

4. példa:

```
X       =       0
        REPT    10
X       =       X+1
        DB      X
        ENDM
```

Ez a kód a

```
        DB      1
        DB      2
        .
        DB      10
```

sorozatot generálja.

5.példa:

```
        IRP     X, < 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 >
        DB      X
        ENDM
```

Ez a kód a 4. példában látott sorozatot generálja.

6. példa:

```
        IRPC    X, 01234567
        DB      X
        ENDM
```

Ez a kód a

```
        DB      0
        DB      1
        .
        DB      7
```

sorozatot generálja.

4.11. Listavezérlési operátorok

.CREF

Ez az operátor megengedi a cross-reference (kereszthivatkozási lista) készítését (alapértelmezés).

.XCREF

Ez az operátor letiltja a cross-reference (kereszthivatkozási lista) készítését.

.LALL

A teljes macroszöveg listázása - hatására minden egyes macrohívás kifejtését sorról sorra olvashatjuk a listában.

.SALL

A macro-k által generált szöveg és object-kód listázásának letiltása

.XALL

Csak valóban kódot generáló részek listázása (az utóbbi három közül alapértelmezés).

.LIST

Listázás engedélyezése (alapértelmezés).

.XLIST

Listázás letiltása.

Megjegyzés: akkor hasznos operátor, ha a program számos, már teljes biztonsággal belőtt részt tartalmaz. Ilyenkor felesleges a jó részeket mindig újra listázni, hiszen az nagyon lemez-, papír- és időigényes. Nem mindig szokás listázni az adatszegmens azon részeit, ahol a program üzeneteit definiáltuk - ez a listán roppant sok helyet igényel és általában felesleges - a kiírt üzeneteknek úgyis beszélniük kell magukért!

%OUT szöveg

Fordítás közben "szöveg"-et kiírja a terminálra.

Megjegyzés: ez hosszabb fordításnál a felhasználó "megnyugtatására" szolgál.

PAGE [op-1] [op-2]

Operandus nélkül listázás közben lapot dob a fordító.
Az első operandus az egy oldalra kerülő sorok számát adja meg; értéke 10 és 255 között lehet. Alapértelmezése: 66.
Ha az első operandus egy "+" jel, akkor a fejezet sorszámát megnöveli, a lapszámot 1-re állítja vissza. A "+" operandusnak egyedül kell állnia.

A második operandus a lap szélességét szabja meg karakterekben; értéke 60-tól 132-ig terjedhet.

TITLE név

Hatására a "név" nevet minden lista-oldalra kinyomtatja. Egy file-nak csak egy TITLE-ja lehet.

SUBTTL név

Hatására minden lapra kinyomtatja a "SUBTTL név" nevet is a TITLE által megszabott név alá. Egy file-ban tetszőlegesen sok SUBTTL használható.

Megjegyzés: célszerű a nagyobb logikai egységeket SUBTTL-el is ellátni, ez egy kicsit olvashatóbbá teszi a programot.

.LFCOND

Ha kiadjuk, minden feltételes blokkot listáz a fordító.

.SFCOND

A nem teljesülő feltételek alatti blokkokat nem listázza a fordító.

.TFCOND

Vált a fenti két feltétel között.

4.12. A source file végét jelző operátor

END [kif]

jelzi a source file végét a fordító számára. A file további részében elhelyezett sorokat a fordító már figyelmen kívül hagyja.

kif a program kezdőutasításának címe.
Csak abban a modulban kell használni, amelyben a program törzse és így belépési pontja helyezkedik el. Egy programban legfeljebb egy kezdőcím adható meg.

4.13. Javaslat assembly alapfile-okra

Nagyon fontos, hogy a forrásnyelvű file külső megjelenése, formátuma valamilyen ésszerű szabványhoz igazodjon. Ezenkívül, mint az eddig elmondottakból kiviláglik, minden egyes assembly programban számos dolgot definiálnunk kell ahhoz, hogy egyáltalán lehetséges legyen a program lefordítása (pl. szegmensdefiníciók; a sikeres szerkesztéshez egy STACK szegmens stb.).

Az eddigiekben csak arról olvashattunk, hogy a programunk, ha már egyszer elindult, mit hogyan csináljon. Ez valóban nagyon érdekes terület. Joggal merülhet fel azonban az

Olvasóban az is, hogy programja hogyan fejezheti be a futást, azaz hogyan adhatja vissza a vezérlést az operációs rendszernek. A második kötetben, amely az MS-DOS és programunk kapcsolatát taglalja, választ kapunk erre az izgalmas kérdésre is. Szükséges azonban, hogy az Olvasó az újabb kötet kézbevétele előtt is tudjon egyszerű programokat írni, majd a programok futtatása során ki is léptethesse őket. A legegyszerűbb ezt egy rövid utasításssorozattal végezni.

Mindezen szempontok azt sugallják, hogy készítsünk jóelőre egy olyan file-t, amely tartalmaz minden, a fordítás és szerkesztés munkájához szükséges alapinformációt, és a programok írása során nekünk "csak" meg kell töltenünk a szegmenseket a szükséges tartalommal. Ez a kis file tartalmazza majd a szabályos kilépést előkészítő utasításokat is.

Ez az alfejezet első felében (igencsak szubjektív) tanácsokat tartalmaz a program formájára, külső megjelenésére nézve. A második része pedig egy jól bevált alapfile szövege. Ezt felhasználhatjuk kiindulásként.

4.13.1. A source file javasolt fomátuma

Az alább felsorolt szempontok a szerző (meglehet, kifacsarodott) személyes ízlését tükrözik. Rendkívül fontosnak tartom azt, hogy esztétikus, szép legyen a programunk megjelenése source file-ként és listában egyaránt. Fontosnak tartom, hogy minden programozó alakítson ki magának olyan konvenciókat file-jai formálására, a szükséges dolgok logikus megadására, és ezekhez tartsa magát az ésszerűség határain belül. Mindenképpen megéri a fáradságot, hogy szép, áttekinthető és logikus formát adjunk a programnak, mert ha később módosítani vagy javítani kell, hasznos lesz, ha tudjuk, hova kell lapozni a macro-k, a konstansok stb. definíciójához.

Minden fontos pontot lássunk el elegendő és precíz magyarázattal a megjegyzés-mezőben, az eljárások legyenek viszonylag rövidek, áttekinthetőek és szabatosan fogalmazottak.

Javaslom, hogy (ha tudjuk) angol nyelven kommentáljuk a programokat. Ez azért fontos, mert magyar nyelvű ékezetes szövegeket kissé körülményes létrehozni (bár nem lehetetlen) és még körülményesebb kinyomtatni. Az ékezetek nélküli magyar

szöveg pedig könnyen lehet félrevezető (pl. a "szöveg" és "szóvég" szavak string-kezelés közben együtt is előfordulhatnak). Azonkívül minden programozási szabvány is melegen ajánlja az angol nyelvű kommentálást. El kell ismernünk azt, hogy angolul rendkívül tömören és precízen lehet fogalmazni, ez pedig a megjegyzésekre meghagyott pár karakterpozíción elengedhetetlenül szükséges.

Ezen elvek követése látszólag igen munkaigényes, de tapasztalni fogjuk, hogy a befektetett munka sokszorosan fog megtérülni.

> Megjegyzés: felhívom az Olvasó figyelmét, hogy az alábbi bekezdések javaslatokat és nem szabályokat tartalmaznak. Fogadja ezeket kellő kritikával, gondolja át, mi miért van így és logikus-e; mi tetszik és mi nem tetszik benne, mivel kellene bővíteni és mi hagyható el belőle. Szó sincsen tehát "tízparancsolatról"; mindenkit amúgyis saját keserű tapasztalatai tanítanak meg majd logikus, szabatos és szép programok írására.

A source file elején a TITLE és PAGE operátorok után célszerű egy hosszabb COMMENT segítségével körvonalazni a file tartalmát. Ezután következzen:

a "GLOBAL SECTION", amelyben felsoroljuk a PUBLIC és az EXTRN szimbólumokat,

a "CONSTANT SECTION", amely a konstansdefiníciókat tartalmazza,

a "MACRO SECTION", amelyben a felhasznált macro-kat, struktúrákat és rekordokat definiáljuk. Ekkor következhetnek az egyes szegmensek. Adjuk meg először a STACK SEGMENT-et, amely több file-os programok esetén csak a főmodulban folgal helyet.

> Megjegyzés: konstans és macro definícióinkat célszerű lehet külön file-ban rögzíteni és az INCLUDE operátor segítségével beolvasni. Ez a gyakorlat különösen akkor hasznos, ha a program több modulból áll - még hasznosabb akkor, ha az egyes modulokat más-más programozó dolgozza ki - ilyenkor a közösen használt konstansoknak és macro definícióknak csak egy forrása van, tehát nem fordulhat elő, hogy a csoport tagjai közül egyesek félrehallás vagy tévedés folytán

helytelen definíciókkal dolgoznak.
Vigyáznunk kell azonban arra, hogy az e kötetben alapul vett 1.00 változatú macroassembler szimbólumtáblázata igen hamar betelik. Amennyiben a harmadik kötetben ismertetendő 3.00 verziót használjuk, ez a veszély már nem fenyeget.

A "DATA SECTION" tartalmazza az adatszegmens(ek) definícióját, majd
a "CODE SECTION"-ben helyezkednek el a kódszegmensek (amelyek a program logikája szerint persze tartalmazhatnak adat- és konstansdefiníciókat is).
Végül, ha szükségünk van dinamikus memóriafelhasználásra, definiáljunk egy MEMORY nevű, 'MEMORY' típusú és osztályú szegmenst, amelyet a LINK feltétlenül a program végére helyez el, ez lesz fizikailag az utolsó szegmens - felfelé korlátlanul bővíthető (azaz, ha szükséges és kellő óvatossággal járunk el, itt létrehozhatunk nagyméretű tömböket, buffereket - de vigyázni kell, hogy ne lépjük túl a memória határait (lásd második kötet, Program Segment Prefix).

Megjegyzés: a "SECTION" szó, a fejezet persze nem kulcsszó: egyszerűen megjegyzésként használjuk, hogy a definíciók rendben helyezkedjenek el.
Megjegyezzük még, hogy amennyiben több file-ba írjuk meg a programot, egy-egy adat- és kód-szegmens szétoszthatható több file-ba is. Ezzel a lehetőséggel azonban óvatosan éljünk, mert nagyon zavarossá teheti a programunkat.

A konstansdefiníciók során, mint a példákban is láthattuk, a sor elején kezdjük a konstans szimbolikus nevét (amelyre hivatkozhatunk majd a programban), egy tabulátor után következzen az EQU vagy az "=" jel, újabb tabulátor után pedig a konstans értékét definiáló kifejezés. Ha elég ügyesen használjuk fel a tabulátort, akkor mind a source file-ban, mind a listán szépen oszlopokba sorakoznak a definíciók.
Egyébként ahol egy szóköz állhat, ott tetszőlegesen sok szóköz és/vagy tabulátor használható egymás után és az mindig egy szóközzel egyenértékű a fordító számára.

Megjegyzés: az "="-vel definiált konstansok sajátos szerepük miatt gyakran a kód- vagy adatszegmensekbe kerülnek; lásd az ismétlési blokkokat.

Lehetőleg minden fontos konstanst lássunk el külön megjegyzéssel.

Az adatszegmensekben is törekedjünk arra, hogy szépen egymás alá sorakozzanak az egyes elemek. Ez csakis esztétikai, áttekinthetőségi szempontból lényeges.

A kódszegmensekben az utasításokat célszerű két tabulátor után írni. Ekkor ugyanis nemcsak a sor elejére írandó címkék, de az esetleges prefixumok (melyeket célszerűen egy tabulátor után írunk) is "kiugranak" a kódból. A fordítónak szóló operátorokat is egy tabulátor után szoktam írni, hogy azok is kiemelkedjenek.

A kétoperandusú utasítások operandusait vesszővel választjuk el. Célszerű, ha még egy szóközt (esetleg tabulátort) is beiktatunk, ez a fordítót nem zavarja, a programozónak kényelmesebb.

Végül néhány szót a szimbólumokról. A fordító igen hosszú neveket is megenged (31 karakterig). Javasolható, hogy a legtöbb szimbólumot két (vagy több) részből tegyük össze egy-egy "_" (underscore) jellel elválasztva. Az egyik utaljon a szimbólum jellegére, a másik pedig konkrét jelentésére. Például az egy csoportba tartozó adatok viselhetnek egy "nemesi előnevet" (FL_), amely felhasználásuk fő területére utal, és egy "tulajdonnevet", amely a konkrét jelentést hordozza (RECSIZ); együtt:

```
FL_RECNUM       DW      0       ;Current record number
FL_RECSIZ       DB      0       ;Record size of the file
```

Az idevonatkozó szabványok szerint egyébként angolul nevezzük el a szimbólumokat, és nevüket a magánhangzók és a nem "nagyon fontos" mássalhangzók elhagyásával rövidítjük, hogy lehetőleg 6-8 karakternél ne legyenek hosszabbak. Ez a gyakorlat különösen akkor hasznos, ha programunkat esetleg más gépen vagy más operációs rendszer alatt is akarjuk fordítani és futtatni - egyáltalán nem biztos, hogy a másik assembler (vagy programszerkesztő) is képes 31 karakteres nevek kezelésére, megkülönböztetésére.

Minden szimbólumnevet csupa nagybetűből építsünk fel, mert

alapértelmezésben a MASM és a LINK is minden kisbetűt nagybetűvé konvertál (fordító- és rendszerfüggetlenség !!).

Nagyon fontos még, különösen, ha magasszintű nyelven írt programokhoz írunk assembly részeket, hogy a szimbólumok neve sohasem kezdődhet a "_" (underscore) karakterrel, mert ez foglalt a szabványos könyvtári rutinok és azok belső változói számára.

A programcimkék megadásánál a következő gyakorlat látszik célszerűnek: az eljárásnak rövid, de velős nevet adunk, majd az eljárás belső cimkéit két részből állítjuk össze: az első rész az eljárás neve, esetleg rövidítve, a második utaljon a konkrét programrészlet rendeltetésére. (A MASM fájdalmasan nélkülözi a lokális cimkéket.) Lássunk egy példát!

```
; BLKCPY        - copies a memory block into an other block
;                       The blocks are in the same segment
;                       and their size does not exceeds the
;                       64 Kb.
;       Inputs:
;               DS/SI   - source block segment/offset
;               DS/DI   - destination block segment/offset
;                               DS and ES are equal !
;               CX      - block size in bytes
;       Outputs:
;               none
;

BLKCPY  PROC    NEAR

                PUSHF                   ;Save flags

                TEST    CX, CX          ;Is it a zero-block?
                JZ      BCPY_QUIT       ;On zero return
                CLD                     ;Init D flag "up"
                CMP     SI, DI          ;Choose higher addr.
                JE      BCPY_QUIT       ;Identical blocks
                JA      BCPY_FORW       ;Jump to copy forw.
```

```
BCPY_BCKW:                              ;Copy backward
            STD                         ;Init D flag "down"
            ADD     SI, CX
            DEC     SI                  ;End of source block
            ADD     DI, CX
            DEC     DI                  ;End of dest. block
BPCY_FORW:

      REP   MOVSB                       ;Repeat until CX = 0

BCPY_QUIT:
            POPF                        ;Retrieve flags
            RET
BLKCPY  ENDP
```

A fenti példa, úgy érzem, jól körvonalazza a főbb szempontokat. Az eljárás rövid, világos - egy memóriablokk másolását végzi el "intelligensen", azaz megvizsgálja, nem 0 hosszúságú-e a blokk, nem azonosak-e a kezdőcímek - ilyenkor nem csinál semmit. Végül megvizsgálja, melyik blokk van alacsonyabb címen. Ha a célblokk, akkor a másolást a legalacsonyabb című byte-al kezdi, és emelkedő címekkel folytatja, ha pedig a kiindulási blokk címe alacsonyabb, akkor az utolsó byte-ot kell előbb másolni és csökkenő címeken folytatni a másolást. Jól elkülönül a mentés és visszatöltés (magukat a bemenő adatokat nem feltéltenül kell menteni, ezekről a hívó amúgyis lemondott már).

Felhívom a figyelmet az eljárást bevezető megjegyzésre, amely a szubrutin "szinopszisát", rövid leírását tartalmazza. Ez nagyon fontos. A rutin hívásakor csak azt kell megnéznünk, a szinopszisnak megfelelően adtuk-e meg a paramétereket, a rutin írása során pedig csak azt, hogy a jónak feltételezett paramétereket jól alkalmaztuk-e. Ez garantálhatja, hogy a blokkmásolás, valahányszor szükséges, jól működjön, s ha mégse, akkor a szinopszis alapján könnyű kijavítani.

Nagyobb programok írása során kódolás előtt érdemes efféle kis eljárásokra bontani a programot, minden eljárás szinopszisát gondosan elkészíteni. Ezután meg kell győződni arról, hogy a meghívandó eljárások igényei kielégíthetők-e a hívás pillanatában (azaz rendelkezésre állanak-e egyáltalán a szükséges adatok - ha nem, akkor át kell fogalmaznunk az algoritmust vagy annak egy részét). Ha az adatstruktúrákat, s

az eljárásokat elég gondosan terveztük meg, akkor a kódolás meglepően gyors lesz, a program pedig könnyen tesztelhető.

Megjegyzés: a programozási gyakorlatban szomorúan tapasztalhatjuk, hogy a lefektetett elveknek még az ellenkezője sem igaz. Mikor apró rutinokra bontjuk a feladatot, akkor észre fogjuk venni, hogy egy jól definiált rutin igen sok (és néha meglepő) helyeken használható. Ezen felbuzdulva úgy "hajlítjuk" a rutin specifikációját, hogy több célra is alkalmas legyen. Azután a belövés során tapasztaljuk, hogy mégsem egészen jó a program: a kritikus rutin bizonyos szituációkban nem egészen jól működik. Kijavítjuk - erre más szituációkban fog hibázni. Végül (szerencsés esetben pár hét alatt) felismerjük, hogy nem azonos, csak hasonló funkciókról van szó. Nosza, írunk egy hasonló rutint. Ekkor azonban alaposan meg kell fontolni, mikor melyiket kell meghívnunk. Itt megint biztosan hibázunk. "... s megint elölről ...", ez a folyamat nem elég gondosan tervezett program esetén az örökkévalóságon túl tarthat.
A túlzott strukturáltság bizony könnyen vezethet oda, hogy egy apró részlet megváltoztatása magával rántja az egész struktúrát - esetleg könnyebb elölről kezdeni a munkát, mint kisebb módosítást elvégezni egy programon.

4.13.2. Alapfile .EXE programhoz

Az elsőként megadott file semmilyen kontrét utasítást nem tartalmaz, pusztán azokat a szükséges "fejezeteket" sorolja fel, amelyek vagy nélkülözhetetlenek (stack szegmens, ASSUME operátor, valamint a szabályos kilépést előkészítő néhány utasítás), vagy szép formát adhatnak programunknak. Tehát (különösen egyszerűbb programok esetén) egy részét el is hagyhatjuk.

A zárójeles magyarázatok természetesen feleslegesek, ezek csak az egyes részek rendeltetését írják le.

```
PAGE    60,132
TITLE   programcím

COMMENT *
                ( Bevezető megjegyzések: a program
                  szerzőjének neve, keletkezésének,
                  módosításának dátuma, a módosítást
                  végző személyek neve;
                  a program célja, esetleges egyéb moduljai,
                  fordítási utasítások, valamint a megértést
                  könnyítő általános megjegyzések )
*
PAGE
;-----------------------------------------------;
;
; Extern section
;
;-----------------------------------------------;

                ( Külső változók deklarációja )

;-----------------------------------------------;
;
; Public section
;
;-----------------------------------------------;

                ( E file-ban definiált
                  globálisok felsorolása )
PAGE
;-----------------------------------------------;
;
; Constant section
;
;-----------------------------------------------;

                ( Konstansok definíciója, a
                  konstansdefiníciókat beolvasó
                  INCLUDE utasításokkal együtt )
                ( Az INCLUDE-ok előtt célszerű egy
                  .XLIST, utánuk pedig egy .LIST
                  utasítás kiadása )
```

```
PAGE
;-------------------------------------------------;
;
; Macro section
;
;-------------------------------------------------;

                ( Macro, struktúra-, rekord definíciók,
                  a macro stb. definíciókat beolvasó
                  INCLUDE utasításokkal együtt )
PAGE
;-------------------------------------------------;
;
; Data section
;
;-------------------------------------------------;

DATA    SEGMENT PARA    PUBLIC  'DATA'

                ( Az adatterületek definíciója )

DATA    ENDS

                ( További adatszegmensek )

;-------------------------------------------------;
;
; Stack section
;
;-------------------------------------------------;

STACK   SEGMENT PARA    STACK   'STACK'

        DW      STACK_SIZE      DUP  (?)   ;Stack's
                                           ;size 256 words

STACK_START     LABEL   WORD               ;Top of the stack

STACK   ENDS
PAGE
```

```
;-----------------------------------------------;
;
; Code section
;
;-----------------------------------------------;

CODE    SEGMENT PARA    PUBLIC  'CODE'

START   PROC    FAR

        ASSUME  CS:CODE, DS:DATA, SS:STACK, ES:NOTHING


                PUSH    DS
                XOR     AX, AX
                PUSH    AX              ;Init for return

                MOV     AX, DATA
                MOV     DS, AX          ;Init DS register

;-----------------------------------------------;
;
; Main prngram
;
;-----------------------------------------------;

                ( A főprogram utasitásai )

;-----------------------------------------------;
;
; Exit to the caller process
;
;-----------------------------------------------;

                RET                     ;Far return to DOS

START   ENDP
PAGE
```

```
;-----------------------------------------------;
;
;Procedures
;
;-----------------------------------------------;

                ( További eljárások a főmodulban )

CODE    ENDS

                ( További kódszegmensek )

MEMORY  SEGMENT PARA    MEMORY  'MEMORY'

                                ;Top of program's memory

MEMORY  ENDS

        END     START
```

A fenti kis "program" (egyébként - a zárójelben elhelyezett magyarázatok elhagyása után - lefordítva és összeszerkesztve "futásra" képes, azonnal kilép) minden szükségeset tartalmaz.

Ennek a kis üres váznak a kitöltésével viszonylag kényelmesen írhatunk assembly programokat. Javasolható, hogy a file (vagy némileg egyszerűsített változata) mindig legyen az assembly programozáshoz használt aldirectory-ban. Hasznos lehet, ha írásvédetté tesszük, hogy ne tudjuk kitörölni vagy felülírni. Egy ilyenféle alapfile hiánya - különösen, ha megszoktuk és számítunk rá - eléggé fájdalmas lehet.

Súlyos kérdésként merülhet fel az előbbi file tanulmányozása után: ahhoz, hogy egy ici-pici kis assembly programot meg tudjuk írni, valóban egy ilyen mammutnyi nagy állatot kell-e kiindulásnak felhasználni? Nincs-e valamilyen mód arra, hogy egy ennél sokkal egyszerűbb szerkezetű file-t töltsünk ki? De, természetesen van. Először is, apró programok esetén a komoly dokumentációhoz szükséges sok "section" elhagyható, hiszen ezek nem részei a programnak. Ha pedig a teljes szabadságot biztosító .EXE típusú program helyett beérjük a kötöttebb formátumú, de sokkal kényelmesebb .COM típusúval is, akkor sokkal egyszerűbb alapfile-ból is kiindulhatunk!

4.13.3. Alapfile .COM programhoz

Miben különbözik egymástól egy .EXE és egy .COM típusú program? A teljes pontosságú választ a második kötetben kapjuk mejd meg. Egyelőre érjük be annyival, hogy a .COM típusú program csak egyetlen szegmensből állhat, amely magábafoglalja a teljes kód-, adat- és stack-területet, valamint a rendszer által futáskor előkészitett 256 byte hosszú Program Segment Prefix területet (második kötet). Programunk ez esetben nem tartalmazhat egyetlen távoli vezérlésátadást sem. A fordítás és programszerkesztés folyamata után (lásd 5. fejezet) az elkészült .EXE típusú program futásra képtelen, mert nem tud szabályosan kilépni. Futtatása előtt még egy programot kell "rászabadítani", az EXE2BIN.EXE nevű konvertáló programot. Ez alakítja át (ha lehet) a .EXE típusú programot .COM típusúvá, amely ezután már futásra kész. Lássunk most egy csupasz alapfile-t (melyet komoly program esetén célszerű lenne bővíteni a fenti "section"-okkal). Felhívjuk a figyelmet arra, hogy a stack szegmens deklarációja nem véletlenül hiányzik, hanem teljesen felesleges.

Amennyiben jobban szeretnénk külön adat- és programszegmenst definiálni, akkor se hagyjuk el a kódszegmensben az ORG operátort, mert arra feltétlenül szükség van, hogy helyet biztosítsunk a Program Segment Prefix 256 byte-ja számára. A külön adatszegmens deklarációja kövesse a kódszegmensét DATA néven, és a programszöveg elején fogjuk össze a két szegmenst egy csoportba a GROUP operátor segítségével.

Aki ismeri a CP/M operációs rendszert, emlékezhet a memória ottani felosztására és a programok ottani felépítésére. Ha mindezt felidézi, megállapíthatja, hogy a .COM formátum szinte teljesen analóg a CP/M programformátumával.

Mint fentebb említettük, a .COM formátum sokkal kényelmesebb és egyszerűbb, mint a .EXE típus. Elmondhatjuk azt is, hogy számos DOS-funkció csak igen ravasz trükkökkel használható .EXE típusú programokból, míg ugyanezek a .COM formátumban magától értetődő könnyedséggel hívhatók.

```
PAGE    60,132
TITLE   programcim

COMMENT *
                ( Bevezető megjegyzés )
*

CODE    SEGMENT PARA    PUBLIC  'CODE'

        ASSUME  CS:CODE, DS:CODE, ES:NOTHING, SS:NOTHING

        ORG     100H

START:
        JMP     ENTRY

                ( Adatterület definíciója )

ENTRY:
                ( A program kódja )

                MOV     AH, 0           ;EXIT to DOS function
                INT     21H             ;Call DOS

CODE    ENDS

        END     START
```

A fenti file-ok közül az elsőt nevezzük el B-EXE.ASM -nek, míg a másodikat B-COM.ASM -nek, arra utalva, hogy az első .EXE típusú programok, a másik pedig .COM típusú programok létrehozására szolgál.

5. fejezet

Az assembler és a LINK használata

Futtatható program létrehozása

Az 5. fejezet tartalomjegyzéke

5.0. Bevezetés

Ebben a fejezetben a futtatható program létrehozásának menetét attól a ponttól kezdve ismertetjük, hogy a programozó megtervezte, megírta és egy szövegszerkesztő program segítségével source file-ba (azaz forrásnyelvű file-ba) rögzítette a program szövegét.

A munka lépései a következők:

a) fordítás - a fordítóprogram segítségével a már létrehozott source file-t lefordítjuk, azaz létrehozunk belőle egy object file-t. (Az object file-t "tárgykód" néven is szokták említeni.)

b) Az object file(ok)-ból a programszerkesztő, angolul "linker" vagy (más operációs rendszerekben megszokott kifejezéssel) "linkage editor", felhasználásával létrehozunk egy végrehajtható programot.
Azt a folyamatot, melyben létrejön a futtatható program, "összeszerkesztésnek" fogjuk nevezni.
Arra kérem az Olvasót, hogy nagyon ügyeljen a szövegszerkesztés és összeszerkesztés közötti különbségre. Az első a forrásnyelvű szöveg létrehozását jelenti valamilyen szövegszerkesztő ("editor") segítségével, a második pedig az object file-ok végrehajtható file-á való konverzióját. Előbbi tehát a program létrehozásának első, utóbbi pedig az utolsó lépése.

c) A belövés munkájában alkalmazunk valamilyen futáskövető programot (DEBUG, SYMDEB, FSD, stb.).

d) Legfontosabb: a kész programot gondosan dokumentáljuk.

Az IBM-PC-XT standard assemblere a Microsoft Corporation által kifejlesztett macroassembler, amely két menetben állítja elő a szabványos "object" formátumú bináris file-t. Ha a

programunk több source file-ból állna, természetesen minden file-t külön kell lefordítani.

Ezután az összeszerkesztés munkáját az MS-DOS operációs rendszer szabványos programszerkesztője, a LINK program végzi.

Megjegyzésre méltó érdekesség, hogy a Microsoft nemcsak az MS-DOS operációs rendszer egyes változataihoz, hanem minden fordítóprogramjához is mellékel egy programszerkesztőt. Ezért a LINK program számos változata létezik - egy külön LINK magához az MS-DOS operációs rendszerhez (ez lehet talán a "szabványos"), egy külön az assemblerhez, egy újabb a C nyelvhez és így tovább. Ráadásul az egyes nyelvek különböző verzióihoz is más és más, egyre fejletteb, "okosabb" LINK-et adnak.

Azonban a LINK változatai nem mindig tudják egymást helyettesíteni még akkor sem, ha különleges szolgáltatásaikról le is mondanánk. Például az assemblerhez és a C fordítóhoz mellékelt LINK programok között igen fontos különbségek vannak. Ha egy C nyelvű programhoz akarunk assemblyben megírt részeket szerkeszteni, mindig a C nyelv programszerkesztőjét használjuk!

Erről bővebbet a második könyvben talál az Olvasó.

5.1. Az assembler működése

Az assemlber a source file-ból (azt változatlanul meghagyva) létrehoz egy object file-t, külön kívánság esetén lista-, valamint cross-reference (hivatkozási) file-t is.

Az object file már gépikódú formában tartalmazza az utasításokat, de futásra még nem képes, mivel (többek között) számos címhivatkozás még nem végleges formában szerepel benne. Ezek teljes előkészítése a LINK feladata. Az object file alapértelmezett típusa ".OBJ".

A listafile egy kinyomtatható szövegfile, mely egyrészt tartalmazza a gépikódú változatot, másrészt a program teljes szövegét, az összes megjegyzéssel együtt. A listafile minden sora egy forrásnyelvű sort tartalmaz, a hozzátartozó lefordított kóddal együtt. A sorok a sorszámmal kezdődnek. Ezután következnek a lefordított kódok hexadecimálisan, tehát a memória tartalma. Ezután olvasható a source-sor. A programlista a programbelövés, futáskövetés nélkülözhetetlen eszköze.

A cross-reference file, a kereszthivatkozási file a MASM másik, a programbelövést segítő terméke. Ez egy ember által fogyaszthatatlan bináris file, amelyből értelmes szövegfile csak a CREF program közbejöttével keletkezik.

A MASM assembler kétmenetes, azaz a forrásszöveget kétszer olvassa végig, hogy object kódot hozzon létre.

Az első menetben a MASM összeszedi a használt szimbólumokat, megállapítja az utasítások helyigényét, előkészíti a kódgenerálást. A hibaüzenetek egy része ekkor keletkezik, lista (hacsak nem kérjük külön) nem készül.

A MASM a második menetben hozza létre a kódot és a listát, minden általa megismert szimbólum értékét behelyettesítve. Ha olyan szimbólumdefiníciót talál az assembler, mely a második menetben (is) kap értéket, akkor ezt összeveti az első menetben kapott értékkel. Ha nem egyezik, "phase error" (fázishiba) következik be. Ezt a rettegett hibát azért kell külön kiemelni, mert minőségileg különbözik a szokásos hibáktól. Nem szintaktikus hiba, és felderítése sokszor igen nehéz.

5.2. Fordítás

Fordítás (I)

Gépeljük be a

```
MASM
```

parancsot (ENTER-el terminálva). Ekkor elindul a macroassembler és megkérdezi a source-, az object-, a lista- és a cross-reference file nevét (a file specifikálása során hivatkozhatunk tetszőleges device-ra):

```
Source filename [.ASM] : file-név
Object filename [file-név.OBJ] : { file-név }
Source listing [NUL.LST]: { file-név }
Cross-reference [NUL.CRF]: { file-név }
```

ahol a [...] az alapértelmezett nevet jelenti. "file-név" mellett nem kell megadni az alapértelmezett ".ASM" tipust. Az object file nevét csak a forrásnyelvű file nevétől való eltérés esetén kell kiírni (egyébként nem szokás eltérni az eredetitől). Alapértelmezésben lista- és cross-reference file nem készül; ha kérjük, akkor ki kell írni a nevét. Ezt jelzi a "{ file-név }". Ezeknek szokásos neve szintén megegyezik a forrásnyelvű file nevével.

Fordítás (II)

```
MASM sfile, ofile, lfile, cfile/parms
```

ahol sfile, ofile, lfile, cfile rendre a source-, object-, lista- és cross-reference file neve. A "/" után opcionálisan adhatjuk meg a fordítási paramétereket (parms). Ha a parancssor végéről valamit elhagyunk, arra MASM kulon kérdez. Ha ezt sem akarjuk, akkor ";"-vel zárjuk le a parancsot.

Például adjuk ki a

MASM file-név;

parancsot. Ekkor sem lista-, sem hivatkozási file nem készül. Ez a szokásos fordítás, ha kevés változtatást végeztünk a programban és nem indokolt újabb lista készítése. A

MASM file-név,,,;

parancs hatására "file-név" néven készül el mindhárom output file. Ez a szokásos fordítási parancs, ha szükségünk van új lista- és hivatkozási file-ra.

Fordítás (III)

Van lehetőség arra, hogy batch file-al vezéreljük a fordítást. Ekkor a batch file-ba kell beiktatni a 2. formátumú MASM-hivásokat.

Lásd a 5.6. alatti példát!

Az MASM paraméterei (opciói)

/D hatására lista készül mindkét fordítási fázisban. Ez megkönnyíti a "phase error"-ok kijavítását.

/O a lista numerikus részét MASM oktálisan konvertálja

/X elnyomja a feltételesen fordítandó blokkok "else" (nem fordított) ágának listázását.

5.3. Cross-reference, kereszthivatkozás

Kereszthivatkozási listát csak akkor kaphatunk, ha az assemblerrel elkészíttettünk egy .CRF típusú file-t. Erre kell "ráereszteni" a CREF programot, amely egy olyan listát készít, melyben ABC-sorrendben felsorolja az összes globális szimbólumot, és azokat a programsorokat, ahol hivatkoztunk az adott változóra. Ez igen hasznos lehet a debug-olás és programmódosítás során.

A CREF program pontosan úgy aktivizálható, mint az assmbler. Az 1. módon külön rákérdez a .CRF file nevére, majd megerősítteti velünk az elkészítendő .REF típusú lista nevét. A 2. módon a CREF sorában adhatjuk meg az input és az output file nevét. A CREF aktivizálható batch file-ból is.

Példa:

```
MASM program,,,;
CREF program,;
```

Ennek hatására a MASM lefordítja a PROGRAM.ASM nevű file-t, létrehozva a PROGRAM.OBJ, a PROGRAM.LST és a PROGRAM.CRF file-okat. Ezután CREF a PROGRAM.CRF-ből előállítja a PROGRAM.REF file-t. Ezek közül PROGRAM.CRF "haszontalan", PROGRAM.LST és PROGRAM.REF kinyomtatandó, míg PROGRAM.OBJ szolgál inputként a LINK számára.

5.4. A LINK szerepe és használata

A LINK programszerkesztő arra szolgál, hogy az object file-(ok)ból létrehozza a futtatható kódot. Azért választják el a fordítóktól, mert

a) így több különálló file-ba rögzíthetjük a programot, a kisebb file-ok (modulok) editálása, fordítása, mentése könnyebb, a file-ok áttekinthetőbbek;

b) így (minthogy a fordítók szabványos object file-okat hoznak létre), lehetővé válik az, hogy a program különböző részeit különböző programozási nyelven írjuk meg.

A LINK szerepe tehát a következő: a különböző object file-okban szereplő feloldatlan hivatkozásokat (ti. azokat, amelyek egy másik file-ban definiált szimbólumra vonatkoznak) kitölti, az azonos nevű és osztályú szegmenseket összehozza, készít egy memória-felhasználási tervet (a programok nem fix memóriaterületen futnak majd, hanem a rendszer dönti el, hogy hová töltődnek be; lásd bővebben második kötet: az EXE file-ok felépítése).

A LINK a MASM-al analóg módon hívható. Ha a file-ok nevét nem specifikáljuk, mindegyikre rákérdez, de van lehetőség még a LINK sorában megadni minden file-t, vagy pedig LINK-et batch file-ból aktivizálni.

A LINK object és library (könyvtár) file-okat vár inputként. Több object megadása egymástól szóközzel vagy "+" jellel elválasztva lehetséges. Ez az első paraméter. Ettől ","-vel elválasztva adandó meg a tulajdonképpeni program neve (melynek típusa .EXE), a map ("térkép"), mely az egyes modulok tárigényét, valamint a globális szimbólumok nevét és típusát tartalmazza (alapértelmezett típusa .MAP), és a library (object könyvtár, melynek alapértelmezett típusa .LIB).

Megjegyzés: a könyvtár igen hasznos programozási segédeszköz. Előre megírt és lefordított rutinokat tartalmaz. Ezek közül bármelyiket szabadon hozzászerkeszthetjük elkészítendő programunkhoz. Célszerű egy könyvtárat létrehozni a programozás során leggyakrabban felmerülő problémák kezeléséhez - ezzel munkát, helyet és időt takarítunk meg. Könyvtárak létrehozására, kezelésére az MS-DOS operációs rendszerben a LIB program szolgál. Ennek használatáról a harmadik könyvben olvashatunk.

Tegyük fel például, hogy PROG1.OBJ, RUT1.OBJ és RUT2.OBJ file-ok összeszerkesztését szeretnénk elvégezni. Az elkészítendő programfile neve legyen PROGRAM.EXE. Ugyancsak PROGRAM néven kérünk egy map file-t is; library-t nem használunk.

(I)

LINK (ENTER)

A LINK lekérdezi a szükséges file-ok nevét:

```
Object Modules [.OBJ] : PROG1 + RUT1 + RUT2
Run File [PROG1.EXE]: PROGRAM
List File [.MAP]: PROGRAM/M
Libraries [.LIB]: (ENTER)
```

Ennek hatására LINK elvégzi a modulok összeszerkesztését. A map file után ajánlatos használni a /M opciót, mert a globálisok listáját csak ekkor kapjuk meg. Enélkül pedig a map nagyon szegényes.

(II)
Ugyanezt az összeszerkesztést elindíthatjuk az alábbi parancssorral:

```
LINK PROG1 + RUT1 + RUT2, PROGRAM, PROGRAM/M;
```

Ha programunk több modulból áll, akkor mindenképpen tanácsos egy batch file-t írni az összeszerkesztéshez. Ennek (attól eltekintve, hogy kevesebbet kell gépelnünk) az a jelentősége, hogy nem tévesztjük el a LINK-nek adandó parancsokat.

Ugyanilyen szituációban lehet segítségünkre a LINK hívásának negyedik módja. Elő kell készítenünk egy speciális LINK parancsfile-t, amelyben (akár több soron át) egymástól "+" jelekkel elválasztva adjuk meg az object file-ok neveit, majd vesszővel elválasztva a .EXE, a .MAP file-ok specifikációját és végül a .LIB-ek felsorolását, szintén a "+" jellel leválasztva. Magát a LINK-et ekkor a következő parancssorral indítjuk:

```
LINK @filenév
```

Ez azz egyetlen mód arra, hogy olyan sok object file nevét adjuk meg, amelyek már nem férnek el egy parancssorban.

A LINK számos opcióját azért nem részletezzük e helyen, mert minden MS-DOS (és MASM) verzióhoz más LINK tartozik, más-más opciókkal. Ezeket a LINK adott verziója leírásában kell megnéznünk. Mindenesetre soroljuk fel a LINK legfontosabb opcióit: a stack méretének megadása (/STACK=n), a kisbetűk

nagybetűvé konvertálásának letiltása (/NOIGNORECASE) és végül az elkészült program elhelyezésének megadása. Ha valamilyen okból nem az elérhető legalacsonyabb, hanem éppen a legmagasabb címre kívánjuk betölteni a programot, akkor a LINK-et a /HIGH opcióval kell meghívnunk.

5.5. .COM típusú program létrehozása

Mint az előző alfejezetben irtakból kiviláglik, a LINK program minden esetben .EXE típusú programot hoz létre, amelyet az erre szolgáló EXE2BIN.EXE nevű programmal konvertálhatunk át .COM típusúvá. A konverzió szükséges akkor, ha úgy foglamaztuk meg a programot, hogy .COM típusú lesz majd; tehát már a program írásakor el kell döntenünk azt a kérdést, hogy milyen típusú program lesz a végeredmény. Természetesen nem minden .EXE típusú program alakítható át .COM típusúvá, de amelyik igen, azt át is kell alakítani, ellenkező esetben (hacsak nem nagyon trükkösen fogalmaztuk meg a programot) a kész program nem fog helyesen futni és nem tud majd szabályosan kilépni.

Amennyiben .EXE típusú programot akarunk írni, akkor a 4.13 alfejezetben megadott alapfile-ok közül az elsőt (vagy ahhoz hasonlót) használjunk fel kiindulásként. Ha pedig .COM típusú program létrehozása a cél, akkor természetesen a másodikat töltsük ki utasításainkkal.

Az EXE2BIN program használata:

```
EXE2BIN  programnév.EXE programnév.COM
```

Mint látható, a LINK által létrehozott .EXE típusú programot adjuk meg először, utána pedig a létrehozandó .COM program specifikációját. A file-ok specifikációja során a .EXE elhagyható. Ha a második file típusát hagyjuk el, akkor annak típusa .BIN lesz.

Amennyiben a konverzió nem sikeres, az EXE2BIN hibaüzenetet küld:

```
FILE CANNOT BE CONVERTED
```

és a kívánt .COM típusú file-t nem jön létre.

5.6. Példák fordító és szerkesztő batch file-okra

Mint tudjuk, a batch file egy olyan szövegfile, amely részint MS-DOS parancsokat, programhívásokat, részint pedig sajátos nyelvének parancsait tartalmazza. (Bővebben lásd a DOS Reference Manual-t.)
Legyen az első batch file neve COMP.BAT, tartalma pedig a következő (feltesszük, hogy listát, kereszthivatkozási listát és map-et szeretnénk kérni minden egyes alkalommal):

```
MASM %1,,,;
IF  ERRORLEVEL 1 GOTO CERR
PRINT %1.LST
CREF %1,;
PRINT %1.REF
LINK %1,,/M,NUL;
IF ERRORLEVEL 1 GOTO LERR
PRINT %1.MAP
%1
GOTO END
:CERR
PRINT %1.LST
ECHO Fordítási hiba !
GOTO END
:LERR
PRINT %1.MAP
ECHO Szerkesztési hiba !
:END
```

A "%1" szimbólum a híváskor megadott első paramétert jelenti. Mit csinál ez a batch file? Először lefordítja az első paraméterként adott nevű, .ASM típusú source file-t, lista- és hivatkozási file-t is készítve. Megvizsgálja, hogy sikeres volt-e a fordítás. Ha nem, akkor MASM 1-es hibakóddal lépett ki. Ekkor elugrik a CERR címkére, kinyomtatja a listafile-t (ekkor szükséges csak igazán), majd egy figyelmeztető üzenet után kilép.
Ha a fordítás sikeres volt, akkor elkészíti a .REF típusú kereszthivatkozási listát, majd megkísérli a program összeszerkesztését. Ha a LINK 1-es hibakóddal tért vissza, azaz

a szerkesztés sikertelen volt, akkor kinyomtatja a .MAP típusú file-t, majd a figyelmeztető üzenet után kilép.

Ha a szerkesztés sikeres volt, akkor rögtön el is indítja a programot, mindjárt ki is próbálhatjuk. Ez egy veszedelmes lépés, mert a nyomtatás még folyik (valószínű, hogy a lassú printer még a lista tizedéig sem jutott, mikor a programunk futni kezd), és ha a még kellően be nem lőtt program "elszáll", akkor oda a printelés is.

A DEBUG elindítása sem lehetetlen a batch file-ból. Be nem lőtt programok esetén (ha egyáltalán beepítjük a programindítást) a program közvetlen elindításánál ("%1") ez a parancssor talán szerencsésebb :

```
DEBUG %1.EXE [paraméterek]
```

Ez a parancssor ugyanis elindítja az MS-DOS futáskövető programját, amelynek segítségével utasításonként hajthatjuk végre a programot. Tehát a rendszer teljes "legyilkolásának" veszélye sokkal kisebb.

A második példa sokkal egyszerűbb, mert nem készül sem lista-, sem pedig hivatkozási file, viszont ebbe beépítjük az elkészült .EXE program konverzióját .COM típusúra. Feltettük, hogy a fordítás és a szerkesztés minden esetben sikeres. Ilyenféle parancsfile-t adhatunk például kész programjainkkal. Mindenképpen szerencsés dolog a kész termékeket a fordítást és szerkesztést, tehát a program előállítását vezérlő parancsfile-okkal ellátni. Ez éppúgy része a rendszernek, mint a teljes dokumentáció.

```
MASM %1;
LINK %1;                    (lehetne pl. LINK @%1.LNK is!)
EXE2BIN %1.EXE %1.COM
DEL %1.EXE
```

6. fejezet

A DEBUG program használata

A 6. fejezet tartalomjegyzéke

6.0. Bevezetés

Az MS-DOS DEBUG nevű programja tetszőleges program futásának tesztelésére, nyomonkövetésére szolgál. Ez a futáskövető a hasonló programoktól elvárható négy fő funkció mindegyikével rendelkezik: memória- és regisztertartalmak kiírása, azok megváltoztatása, programvégrehajtás lépésenként, végül a program végrehajtása egy töréspontig, azaz egy meghatározott utasításig.

Számos egyéb szolgáltatása is van. Képes visszafejteni egy adott ponttól a memóriában talált programot (sajnos nem pontosan a MASM formátuma szerint) és lehetővé teszi programok írását assembly mnemonic-okkal. Fontos lehet még, hogy bárhova átmásolhatunk, vagy tetszőleges értékekkel feltölthetünk egy memóriablokkot.

> Megjegyzés: a legvadabb programokat éppen a DEBUG segítségével lehet egyszerűen létrehozni, a MASM ugyanis egy olyan kötött formát szab meg a felhasználónak, amely általánosságban megkönnyíti a munkát, de "aljasságok" elkövetését megakadályozza.

A DEBUG képes arra is, hogy a lemezterület meghatározott pontjáról olvasson (így "belenézhetünk" a lemez bármelyik szektorába, sőt hibás vagy nem formázott lemezeket is megkísérelhetünk olvasni), és segítségével a memória egy adott részének pillanatnyi tartalmát lemezfile-ra menthetjük.

6.1. A DEBUG leírásában használt jelölések

cím	egy- vagy kétrészes címleírás: seg:offs pl. 0044:0F57 segreg:offs pl. ES:0F30 offs pl. 400 (ez utóbbi a DS:re vonatkozik)
byte	egy byte-nyi információ (egy vagy két hexadecimális számjegy)
dr	egy számjegy, amely azonosítja a kívánt lemezmeghajtót (eredeti neve "drive", innen származik): 0 - A:, 1 - B:, stb.
filespec	file specifikáció (DOS formátumban)
lista	egy vagy több byte-nyi vagy string érték
portcím	1-4 hexa számjegyből álló szám, amely azonosít egy I/O portot
tartomány	egy memóriatartomány kijelölése, formátuma lehet 1. "cím cím" vagy "cím,cím" 2. "cím L hossz" ahol "hossz" egy hexa szám. Megjegyezzük, hogy a tartomány legfeljebb 10000 (hexa) hosszú lehet.
sect sect	1-3 hexa számjegyből álló számok, melyek azonosítják a beolvasandó lemezterület első szektorát, valamint hosszát szektorokban. A "sect" az eredeti "sector" megnevezés rövidítése.
string	idézőjelek közé zárt karaktersorozat
érték	1-4 jegyű hexadecimális szám

6.2. A DEBUG indítása

Mint a DOS-ban szokásos, a DEBUG is egyszerűen nevének leírásával indítható:

C>DEBUG <ENTER>

Ezután belép a DEBUG és a "-" (mínuszjel) prompt-tal jelzi, hogy kész a felhasználó parancsainak fogadására. Lehetséges az indítás a tesztelni kívánt program specifikálásával is:

C>DEBUG filespec

Ekkor a DEBUG rögtön be is tölti a specifikált programot, melynek tesztelését azonnal elkezdhetjük. Természetesen ekkor is meg kell várnunk a prompt-ot, a mínuszjelet. Hasznos szolgáltatás még, hogy lehetséges a paraméterátadás is:

C>DEBUG filespec par1 [par2 ...]

A DEBUG betölti a specifikált programfile-t, és annak "Program Segment Prefix"-ét (lásd második könyv) úgy készíti elő, mintha a programunkat a

C>filespec par1 [par2 ...]

paranccsal indítottuk volna el.

A DEBUG prompt-ja a "-" (mínuszjel). Akkor adhatunk a DEBUG-nak új parancsot, ha ezt látjuk a sor elején. A parancsokat általában az ENTER-el kell terminálni, lezárni, azaz "elküldeni". A CTRL-BREAK leállítja az elindított tevékenységet.

6.3. A DEBUG parancsai

A DEBUG parancsai egykarakteresek, nagy- és kisbetűkkel is megadhatók. A parancsok és paraméterek, ahol szükséges az elválasztás, szóközzel vagy határolójellel (általában ",") is elválaszthatóak.

6.3.1. File- és lemezkezelési parancsok

N [dr:][path]file-név[.típus]

Name - a tesztelni (debug-olni) kívánt file megnevezése. Mint látható, teljes file-specifikációt használhatunk, azaz eltérhetünk az aktuális lemezmeghajtótól (drive) és tartalomjegyzéktől (directory), a lemez bármely pontjára tudunk hivatkozni. Amennyiben eltérünk az aktuális meghajtótól, akkor a kívánt lemezt betűvel és kettősponttal kell megneveznünk!

Megjegyzés: bármilyen file-t beolvashatunk, nem kell okvetlenül "EXE" vagy "COM" típusúnak lennie. Ha kíváncsiak vagyunk egy file tartalmára, a DEBUG segítségével byte-ról byte-ra megnézhetjük, sőt bele is javíthatunk.

L[cím] [dr sect sect]

Load - beolvassa az "N", a "Name" paranccsal specifikált file-t vagy az itt megnevezett meghajtóról a megadott szektorokat (egyszerre legfeljebb 80 szektort).
Ha megneveztünk már egy file-t, a "cím" megadása nem szükséges, a programfile "jó helyre" fog betöltődni. Ez azt jelenti, hogy a DEBUG pontosan olyan elvek szerint olvassa be a programot a memóriába, ahogyan az a rendszer tenné (lásd második könyv).

W[cím] [dr sect sect]

Write - módosított adatok (programkód) kiírása lemezre "cím"-től kezdődően. Felülírhatunk vele bármely lemezterületet (így pl. a File Access Table területét is!), így nem lehetünk eléggé óvatosak.
Ha nem adunk meg szektor-specifikációt, akkor a "Name" paranccsal specifikált file-t fogja felülírni a program.

6.3.2. Memória- és regisztertartalmak kezelése

D[cím] vagy D[tartomány]

Dump - memóriaterület tartalmának kiírása a képernyőre: "cím"-től kezdve hexadecimális 80 (azaz decimálisan 128) byte, vagy a "tartomány" tartalmát.

E cím [lista]

Enter - értékek bevitele a memóriába. A memória felülírása a "cím"-től kezdődik.
Ha nem adunk "listá"-t, akkor vár a beviendő értékre. Ez "byte"-nyi érték lehet. Ha "space"-t ütünk terminátorként, akkor E áttér a következő byte-ra; ha "-"-t, akkor az előzőre. Az ENTER billentyűre befejezi az adatok bevitelét.

Példák:
1. Adjuk meg a következő parancsot:

eb800:500 <ENTER>, mire a DEBUG a

b800:0500 20. sorral válaszol és várja a választ, ahol "20" a byte eredeti tartalma (ez lehet más is, természetesen).

A cursor most a pont után villog. Adjunk be ezután "41 <SPACE>"-t (a szóköz lényeges). Ekkor a képernyő egy pontján (a 8. sor elején) megjelenik egy "A" betű, és a DEBUG a következő pozíciót nyitja meg:

b800:0500 20.41 07.

Ezt írjuk felül "14 <ENTER>"-el. Ekkor az előző "A" attributuma megváltozik; eddig fekete alapon szürke betűt láttunk, ezután sötétkék alapon piros lesz a betű (lásd második könyv).

Megjegyzés: a szóköz azért kellett, mert ENTER-re a DEBUG végrehajtana egy soremelést és a következő lépésben megnyitott (b800:501) pozíció (a képernyő feltolása miatt) már nem az "A", hanem a korábban alatta levő karakter attributuma lenne.

2.
Ha a "cím" után megadunk "listá"-t is, akkor a DEBUG a lista hosszának megfelelő számú byte-ot ír felül:

E ds:100 F3 "xyz" 8d

Hatására 5 byte-ot ír felül, az elsőbe d3-at (hexa), aztán "x", "y" és "z" kódját, végül pedig hexa 8d-t.

M tartomány cím

Move - "tartomány" tartalmának átmásolása "cím"-től kezdve. A másolás "intelligens", átlapoló blokkokat is adatvesztés nélkül másol át.

C tartomány cím

Compare - összehasonlít két memóriaterületet ("tartományt"-t a "cím"-től kezdődő, azonos hosszúságú területtel) és listázza a különböző pozíciókat, először címüket, majd tartalmukat kiírva.

F tartomány lista

Fill - "tartomány" feltöltése "listá"-val, annyiszor ismételve, ahányszor szükséges.

S tartomány lista

Search - "tartomány"-ban megkeresi a "lista" valamennyi előfordulását (byte-ról byte-ra egyező területet) és (kezdőcímükkel) kilistázza őket.

R[regiszternév]

Register - a processzor regisztereinek tartalmát kiírja és lehetővé teszi megváltoztatásukat.
Ha nem adunk meg regiszternevet, akkor minden regiszter tartalmát kiírja. Ilyen esetben azonban semelyik regiszter tartalmát nem változtathatjuk meg.
Ha megnevezünk egy regisztert (AX, BX, CX, DX, BP, SI, DI, SP, IP, DS, ES, SS, CS, F valamelyikét) akkor csak annak a tartalmát írja ki, és lehetővé teszi megváltoztatását. Az "F" tartalmát bitenként lehet módosítani.
Az egyes flag-állások "fantázianeve" a következő:

Carry	- CY/NC
Parity	- PO/PE
Aux. carry	- AU/NA
Zero	- ZR/NZ
Sign	- PL/NG
Trap	- nem állítható
Direction	- UP/DW
Interrupt	- EI/DI
Overflow	- OV/NV

Az "RF" parancsra a DEBUG kiírja az összes flag-et, és megengedi egyikük módosítását, ekkor be kell adni a kívánt "fantázianevet" a fentiek közül.

Programterületek kilistázása és módosítása

U[cím] vagy U[tartomány]

Unassemble - a kurrens utasítástól vagy "cím"-től (vagy "tartomány" elejétől) megkísérli assemblyre visszafordítani az utasításokat. Ha a kezdőcímet nem helyesen adjuk meg (s igy a visszafejtés nem utasításkódon, hanem operanduson kezdődik), teljesen értelmetlen eredményt kaphatunk.

A[cím]

Assemble - az opcionálisan megadott címtől kezdve, mnemonikus formában assembly parancsokat vihetünk be. Ha címet nem specifikálunk, első izben a CS:0100H-tól, minden további "A" utasítás kiadása esetén az utolsó utáni címtől kezdődik a "fordítás".
Nagyon érdekes, hogy a DEBUG megengedi megjegyzések "elhelyezését" egy pontosvessző után. Természetesen ezeket soha többé nem látjuk viszont, tehát megadásuk feleslegesnek mondható.

Fő különbségek a MASM és a DEBUG szintaxisa között:

1.) Címzési módok: ha számot irunk operandusként, az beépitett címzést jelent, a direkt memóriacímzéshez [...]-be kell tenni az operandust.

2.) A távoli visszatérést a RETF mnemonic-al kell megadni.

3.) Néhány utasítás nem adható meg, pl. a

```
MOV     [256F], 0F4C
```

utasítás - ki tudja, miért, míg a MASM megengedi, hogy beépitett adatot direkt címzéssel egyenesen egy memóriabyte-ba irjunk, a DEBUG ezt nem fogadja el.

4.) A szegmens-átdefiniáló prefixumot az utasítás elé kell írni:

ES: MOV AX, 34F[SI][BX]

6.3.3. Programkövetés

G[=cím] [cím [cím...]]

Go - a programvégrehajtás folytatása. A folytatás címét az első opcionális operandussal adhatjuk meg. A második (a többi) opcionális operandus töréspont(ok) specifikálására szolgál (ez lehetővé teszi, hogy feltételes vezérlésátadó utasítások mindkét "ágára" tehessünk töréspontot).
Ha az első operandust elhagyjuk, a végrehajtás az IP által címzett utasítástól folytatódik.
Ha töréspontо(ka)t adunk meg, a programvégrehajtás az első töréspont elérésekor megszakad, de a DEBUG "elfelejti" a töréspontо(ka)t, mindig újra meg kell adnunk ezeket. Ha töréspontot nem adtunk meg, akkor a program a kilépésig vagy (gyakrabban) az első fatális hibáig fut.
Töréspontról csak másik töréspontra tudunk G paranccsal menni. Nem hajthatunk úgy végre ciklusokat, hogy a visszaléptető utasításra teszünk egy töréspontot, majd annak elérésekor ugyanaddig a pontig szóló újabb G-t adunk. Ekkor a DEBUG nem hajt végre egyetlen utasítást sem! Előbb egy T parancsot, majd egy újabb teljesen specifikált G parancsot kell kiadni.

T[=cím] [érték]

Trace - nyomkövetés. Végrehajt "érték" számú lépést (utasítást) a pillanatnyi utasítástól (vagy a "cím"-en levő utasítástól) kezdve, majd egy R parancsot, azaz kiírja a regiszterek pillanatyi állását. Ha "érték"-et nem adunk meg, a DEBUG egy utasítást hajt végre.

H érték érték

Hexarithmetic - kiírja a megadott két érték összegét és különbségét.

Megjegyzés: ez a parancs azért itt szerepel, mert leginkább olyan esetekben alkalmazzuk, ha ki akarjuk számítani két utasítás egymástól vett távolságát. A távolság fontos lehet akkor, ha el kell döntenünk: elérhető-e egy utasítás egy feltételes vezérlésátadással, vagy sem.

6.3.4. Input-output utasítások

I portcím

Input - input-ot hajt végre "portcím"-ról.

O portcím byte

Output - output-olja "byte"-ot a megadott port-ra.

Megjegyzés: a manuálisan végrehajtott I/O utasítások igen veszélyesek. Egy-két jól irányzott output kiadásával könnyedén lerombolhatjuk a rendszert, sőt, ami még nagyobb baj, a lemezterületen is jóvátehetetlen károkat okozhatunk.

6.3.5. Kilépés DEBUG-ból

Q

Quit - befejezi a DEBUG futtatását (kilép).

Az IBM PC/XT - felhasználóknak és programozóknak

A teljes háromkötetes munka azok számára szolgál hasznos olvasmányként, akik programozási gyakorlattal, de legalábbis számítástechnikai alapismeretek birtokában szeretnék felhasználni, programozni az IBM PC/XT személyi számítógépet.

A könyv koncepciójában körülbelül félúton helyezkedik el a tankönyv és a kézikönyv között. Egyfelől teljességre törekszik, célja, hogy a lehetőségek határain belül mindent elmondjon erről a gépről és az operációs rendszerről. Másfelől mélyebben igyekszik megvilágítani minden olyan részletet, amely "kényesebb" kérdésnek látszik és számos olyan elméleti és gyakorlati jótanácsot ad, amelyek hasznosak lehetnek minden kezdő programozó számára.

A könyv alapnyelve az assembly, sőt az első kötet a gép assembly programozásának ismertetésére szorítkozik. A később ismertetett rendszerfunkciók és hardware leírások természetesen nem kötődnek az assemblerhez. Aki figyelmesen végigolvassa a három kötetet, lehetőleg a gyakorlatban is kipróbálva minden ismertetett lehetőséget, teljes képet kap arról, hogy mit tud az MS-DOS operációs rendszer, és az e rendszer vezérelte hardware.

A három kötet a hardware elemeinek ismertetése során azonban csak addig megy el, ameddig az operációs rendszer is "engedi" a programozót. Nem ismertet tehát olyan lehetőségeket, amelyek meghaladják a szokott programozási kereteket. Ezen ismeretek birtokában nem "feszíthetjük szét" az operációs rendszert, nem kerülhetjük meg annak szolgáltatási és védelmi rendszerét. Ez már jóval meg is haladná egy átlagos programozó és felhasználó igényeit.

I. kötet

IBM PC/XT assembly alapismeretek

Ez a kötet az IBM-PC-XT assembler programozásához szükséges alapvető ismereteket tartalmazza. Megértéséhez csak számítástechnikai alapismeretek szükségesek.

A kötet fejezeteiben a következő témaköröket ismerteti: az Intel 8086/8088 processzor legfontosabb szolgáltatásai, újdonságai; a processzor gépi szintű programozásához szükséges ismeretek: címzési módok és utasításkészlet; a MASM 1.0 verziója szolgáltatásainak teljes ismertetése, beleértve a MASM összes pszeudo operátorát, más néven direktíváit; a fordítás és programszerkesztés vázlatos menete, valamint a MASM és a LINK használata, végül pedig a DEBUG futáskövető program használata.

A fenti tárgykörök tényszerű ismertetésén túl számos kisebb programpélda és programtöredék segíti az Olvasót a kényes pontokon.

A kötetben olyan assembler alapfile-ok is helyet kaptak, melyeket különféle programok írása során kiindulásként használhat a programozó.

II. kötet

A program és az MS-DOS operációs rendszer

A második kötet célja, hogy ismertesse az MS-DOS operációs rendszer és egy futó program kapcsolatainak teljes rendszerét. Milyen útravalóval indítja le az MS-DOS a programunkat? Futás közben milyen MS-DOS szolgáltatásokat vehetünk igénybe? Hogyan léphet ki a programunk, mit hagyhat maga után?

Ez kötet első részében tehát leírja, hogyan készíti elő futásra az MS-DOS a programunkat, az elindított program milyen paramétereket vehet át.

Részletesen és minden egyes függvényhíváshoz (esetenként több) példát adva ismerteti az MS-DOS függvényeit, valamint az ANSI driver szolgáltatásait.

A kötet második része a ROM BIOS funkcióit ismerteti, szintén számos programpéldával. Segítségével közvetlenül fordulhatunk a képernyő-, klaviatúra-, diskette-, printer- és aszinkron vonalkezelő driverekhez, tehát sokkal szabadabban mozoghatunk a gép hardware-támogatta szolgáltatásai között, mint pusztán az MS-DOS segítségével. A második kötet leírja a hang-generálás legegyszerűbb módszereit is.

A kötet harmadik részében ismerteti az Intel 8087 lebegőpontos koprocesszor felhasználását, programozását. Ennek során kitér a számítógépes számábrázolás legelterjedtebb módszereire is.

III. kötet

IBM PC/XT programozási példatár

A harmadik kötet, mint címe is mutatja, egy programozási példatár. E kötet alapnyelve is az assembly. Első részében számos igen egyszerű programot és ezeknek különféle megvalósításait tartalmazza. Ennek során részletesen megvizsgálja az egyes megoldások előnyeit, hátrányait. Számos igen fontos kérdés (mint a paraméterátadás, a vezérlési szerkezetek, adatszerkezetek stb) is terítékre kerül.

A kötet második része a programozás gyakorlati problémáival foglalkozik: a "profi" programozáshoz ajánlható környezet megteremtése (directory-rendszer, az assembler programozásra "kihegyezett" szövegszerkesztő stb.). Ennek során kitér a MASM 3.0 verziójának számos újdonságára, mint a SYMDEB, a MAKE segédprogram és a többi.

A kötet harmadik része nagyobb programozási feladatokat tartalmaz. Nem mindig törekszik a feladatok teljes megoldására (bár néhány hasznos kis segédprogram is olvasható benne). Az eddig szerzett ismeretek birtokában bárki viszonylag csekély (de nagyon tanulságos és hasznos) erőfeszítés árán részleteiben is befejezheti az itt felvetett és kényesebb pontjaikon megoldott programozási feladatokat.

The IBM PC/XT for users and programmers

The three volumes of this book may be useful for those who, having some practice in programming or at least a general understanding of computer fundamentals, would like to use and program the IBM PC/XT.

As to its concepts, the book lies somewhere halfway between a textbook and a manual. On the one hand, it strives to be as complete as possible in its endeavours to tell its readers everything about this computer and operating system. On the other hand, it attempts to give an insight into "delicate" details and provide the novice programmer with useful theoretical and practical advice.

The basic programming language of the book is the assembly language. In fact, the first volume of the book confines itself to the discussion of the assembly language programming of the IBM PC/XT. The system functions and hardware descriptions discussed are not connected to the assembler only. Those who study the three volumes carefully and also practice with the facilities described will have a complete view on what the MS-DOS operating system, and the hardware under its control, can do.

In describing the elements of the hardware, the book proceeds only to the point where the operating system "lets" the programmer go. Thus, it does not describe facilities which lie beyond the scope of general programming. With the knowledge given by the book, the reader cannot "burst open" the operating system, cannot bypass its services and system of protection. These attempts would be beyond the needs of the average user or programmer.

Volume 1

IBM PC/XT assembly language fundamentals

This volume contains basic information on the assembly language programming of the IBM PC/XT. The reader of the book is only required to be familiar with the basic principles of computing.

The chapters of the book discuss the following topics: the most important services and novelties of the Intel 8088/8086 processor; information required for the machine-level programming of the processor: the addressing modes and the instruction set; the complete description of the facilities of the MASM macroassmbler; the process of assembling and linking, the use of the MASM and the LINK programs and the use of the DEBUG program.

In addition to the factual description of the above topics, several sample programs and program fragments are included to assist the reader.

The volume includes basic assembler source files that can be used as a starting frame in writing various assembler programs.

Volume 2

The program and the MS-DOS operating system

The aim of Volume 2 is to describe the entire system of connections between the MS-DOS operating system and the programs running under it. How does the MS-DOS start a program? What run-time services are provided by the MS-DOS for the program? How can a program exit and what can it leave behind?

The first part of this volume describes how the MS-DOS prepares a program for running and what parameters can be received by the program started.

It contains a detailed description, along with appropriate examples, of each and every MS-DOS function and of the services of the ANSI driver.

The second part of the volume deals with the functions of the ROM BIOS with several sample programs. With the help of the ROM BIOS a direct link is provided to the video, keyboard, diskette, printer and asynchronous line drivers, allowing the programmer a greater freedom of movement among the hardware supported services of the computer than would be possible by means of the MS-DOS alone. Volume 2 also includes the simplest methods of sound generation.

The third part of the volume describes the use and programming of the Intel 8087 mathematical coprocessor. It also covers the most common forms of computer number representation.

Volume 3

Sample programs for the IBM PC/XT

As shown by the title, Volume 3 is a collection of sample programs. The programming language of this volume is also the assembly language. The first part of the volume contains a number of very simple programs and their various implementations. The advantages and disadvantages of the implementations are discussed in detail. This part also dwells on a number of important issues, such as parameter passing, control structures, data structures, etc.

The second part deals with the practical problems of programming: the creation of an environment recommended for professional programming by setting up an appropriate directory structure and using a "custom tailored" editor for writing assembler source files. Several new features of the MASM version 3.00 including the SYMDEB and MAKE utility programs are also discussed here.

The third part of the volume contains more complex programming problems. The problems are not always completely solved, although several little useful utility programs are actually written. With the knowledge obtained from the book, the reader is urged to make an effort to complete these programs (whose delicate details have already been solved), to gain programming experience.

Данное издание в трех томах является полезным чтением для того, кто желает использовать, программировать персональную ЭВМ IBM PC/XT имея практику в области программирования, но в крайней мере основные знания по ВТ.

Эта книга является промежуточной от учебника до руководства. С одной стороны старается к полноте, имеет цель — в рамках возможности — рассказать все о машине и об операциой системе. С другой стороны она освещает ряд подробностей, оказывающие особую трудность и дает теоретические и практические советы, полезные для каждого начинающего программиста.

Основным языком является ассемблер, и первый том включает в себе изложение этого языка. Системные функции и описания технических средств конечно не ограничены на ассемблер. Внимательно прочитав три тома, и по мере возможности проверив в практике изложенные, получим полную картину о возможностях операционной системы MS—DOS и комплекса технических средств, работающего под ее управлением.

Элементы технического обеспечения описываются только в мере допускаемые программисту операционной системой возможностей. Таким образом не излагаются возможности, выходящие из обычных рамок программирования. Следовательно программист не приобретает знания, с помощью которых он может разброить рамки операционной системы, обгонять ее услуги и систему защиты. Все это далеко выше потребностей обыкновенного программиста или пользователя.

Том I.

Основы ассемблер IBM PC/XT

В этом тому содержатся основные знания, необходимые к программированию в ассемблере. К пониманию необходимы лишь общие знания по вычислительной технике.

В отдельных заглавиях описываются: основные услуги процессора Интел 8086/8088, его новости; знания к программированию процессора на машинном уровне: способы адрессации и набор каманд; полное изложение услуг версии 1.0 MASM, включая все псевдооператоры (директивы) MASM-а, эскизный ход трансляции и редактирования программ; использование MASM и LINK; использование программы слежения выполнения DEBUG.

Кроме фактического описания высшей тематики Читателю обеспечивает помощь на трудных пунктах множество примеров и отрывок программ.

В тому находятся и основные файлы ассемблера, используемые в качестве исхода при сочетании программ разного рода.

Том II.

Программа и операционная система МС–ДОС

Целью второго тома является описать полную систему связей между операционной системы MS–DOS и выполняемой программы. Чем отправляет MS–DOS программу? Какие услуги MS–DOS допустимы при прогоне? Как может программа выходить, и что остается за ней?

В первой части описано, как подготавливает MS–DOS программу к прогону, какие параметры переносимы в отпущенную программу.

Подробно и с (несколькими) примерами изложены функции MS–DOS по вызову, а также услуги драйвера ANSI.

В второй части тома изложены функции ROM BIOS , также проиллюстрировано примерами. С его помощью имеется возможность непосредственного обращения к фрайверам управления дисилеем, клавиатурой, НГМД, АЦПУ и асинхронной линией, то есть имеется более свободное движение по услугам, поддержанным техническими средствами, чем с помощью MS–DOS. Описаны также простейшие возможности генерирования звуков.

В третьей части изложены применение и программирование копроцессора с плавающей точкой INTEL 8087.

При этом представлены наиболее распространенные способы изображения цифр в ЭВМ.

Том III.

Сборник примеров по программированию IBM PC/XT

Том 3, как покажет и его заглавие содержит практические примеры. Основной язык тома представляет собой ассемблер. В первой части находятся простые примеры с разными решениями. При этом исследуются выгоды, недостатки отдельных решений. Просвещается много важных вопросов, в частности передача параметров, структуры управления, структуры данных и.т.д.

Вторая часть занимается такими практическими проблемами программирования, как создание среды, необходимой к профессиональным программированиям (система каталогов, редактор текста для программирования в ассемблере и т. д.). При этом изложены некоторые новости версии 3.0 MASM-а, как вспомогательные программы SYMDEB и MAKE.

В третьей части тома находятся более крупные примеры программирования. Нет стремления к полному решению примеров, несмотря на то, что можно там прочитать ряд полезных вспомогательных программных отрывок. С помощью приобретенных в предыдущих частях книги знаний, читатель может сам — сравнительно маленьким (но очень учетливым и полезным) усилием закончить подставленные, на своих критических пунктах авторам решенные задания.

139.–Ft

A háromkötetesre tervezett mű első része az IBM PC/XT assembly nyelvű programozáshoz szükséges legfontosabb ismereteket foglalja össze. Nyolc fejezetben ismerteti a gép processzorát, utasításkészletét és címzési módjait; a fordítás és szerkesztés folyamatát, a fordító szolgáltatásait, valamint a leghasznosabb rendszerismereteket: az operációs rendszer szolgáltatásait, a gép belső drivereinek használatát.

A tervbe vett további kötetekkel együtt (melyek számos példa mellett az újabban elérhető assembler fordítók és a hozzájuk tartozó újabb programok ismertetését is tartalmazzák majd) ez a könyv azoknak nyújt segítséget, akik bizonyos programozási gyakorlattal szeretnék elkezdeni az IBM PC/XT assembler programozását. Hasznos lehet azok számára is, akik más nyelven (pl. PASCAL, C nyelv) programozzák az IBM PC/XT-t és ki akarják használni az operációs rendszer és a beépített rendszerprogramok olyan sajátosságait, melyek „kívül esnek" a felhasznált programozási nyelv lehetőségein.

SZÁMALK