

Assembly Tutorial

[1. Bevezetés](#)

[2. Hexaritmetika](#)

[3. Negatív számok](#)

[4. Regiszterek](#)

[5. Memória](#)

[6. Összeadás](#)

[7. A négy alpművelet](#)

[8. Megszakítások](#)

[9. Programok beírása](#)

[10. Karakterlánc kiírása](#)

[11. Átviteljelző bit \(carry flag\)](#)

[12. Ciklus képzése](#)

[13. Bináris szám kiírása](#)

[14. Állapotjelző bitek](#)

[15. Egy hex számjegy kiírása](#)

[16. Léptetés utasítások](#)

[17. Logika és az ÉS művelet](#)

[18. Két hex számjegy kiírása](#)

[19. Egy karakter beolvasása](#)

[20. Hex szám beolvasása](#)

[21. Eljárások](#)

[22. Verem és a visszatérési címek](#)

[23. A PUSH és POP utasítások](#)

[24. Hex számok beolvasása II](#)

[1] Bevezetés

A sugó segítségével megismerkedhetsz az assembly programozás alapfogalmaival. Az itt ismertetett anyag megtalálható Peter NORTON és John SOCHA "AZ IBM PC assembly nyelvű programozása" című könyvében. Az assembly nyelv alapfogalmainak bemutatására igen alkalmas a DOS "Debug" nevű programja, melyet eredetileg a hibák megkeresésének könnyítésére írtak. Neve az angol "bug" - azaz bogár, poloska - szóból ered, mely a számítástechnikai zsargonban programhibát jelöl. Egy működő programban nincs "poloska", míg egy nem működő, akadozó programban legalább egy van. A Debug használatával egy programot lépésenként futtathatunk, és miközben nyomon követhetjük működését, megtalálhatjuk és kijavíthatjuk a hibákat. Ez a folyamat a "debugging", azaz "poloskaírtás", amiből a Debug neve ered.

A számítástechnika hőskorából származó mondák szerint a kifejezés a kezdeti időkből származik - pontosabban attól a naptól, amikor a Harvard egyetemen levő Mark I számítógép elromlott. Hosszas keresgélés után a technikusok megtalálták a hiba forrását: egy kis lepke szorult egy relé érintkezői közé. A lepkét eltávolították, és az üzemelési könyvbe került egy bejegyzés a Mark I "bogármentesítéséről".

Olvasás közben lehetőség van az ismertetett parancsok kipróbálására. Az "Debug program indítása" gomb hatására a "debug" program indul, megjelenik a promptja, a "-", tehát utasításra vár. Ekkor beírhatod az egyes fejezetekben található javasolt parancsokat, de teljesen szabadon is dolgozhatsz vele. A "q" billentyű lenyomásával lépünk ki a programból.

A "debug" a DOS-ból közvetlenül indítható a "debug" szó begépelésével. Kilépés hasonlóképpen a "q" paranccsal történik.

[Top](#)

[2] Hexaritmetika

Kezdjük az assembly nyelvvel való ismerkedést annak megtanulásával, ahogyan a számítógépek számolnak. Ez elég egyszerűnek hangzik, hiszen pl. 11-ig úgyszámolunk el, hogy: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11.

A számítógép nem így számol, hanem pl. ötig így: 1, 10, 11, 100, 101. A 10, 11, 100, stb. számok bináris számok, olyan számrendszerre alapozva, amelyben csak két számjegy van, az egy és a nulla, a jobban ismert decimális számrendszerhez kapcsolódó tíz számjegy helyett. Így a 10 bináris szám ekvivalens azzal a decimális számmal, amit kettőként ismerünk. Azért érdekel bennünket a bináris számrendszer, mert a számítógépekben levő mikroprocesszor ilyen formában kezeli a számokat. De míg a számítógépek bináris számokon léteznek, az egyesek és nullák hosszú sorozatait igen nehézkes és hosszadalmas lenne leírni. A megoldást a hexadecimális számok jelentik, amik a bináris számok ábrázolásának lényegesen tömörebb módjai.

Hexadecimális, azaz tizenhatos számrendszerben tizenhat számjegyre van szükségünk, ezért a megszokott tíz számjegyet 0-tól 9-ig ki kell bővítsük még hat számjeggyel, megszokott jelölésük az ábcé első hat betűje, azaz A, B, C, D, E, F. A hexadecimális számokkal és ezek számítógépben történő ábrázolásával jobban megismerkedhetünk a Debug H (Hexaritmetika) utasítása révén. Ez az utasítás kiszámítja két hexadecimális szám összegét és különbségét. Próbáljuk ki! Ellenőrizzük az eredményeket papíron!

Gyakoroljuk papíron az átalakítást tízes és tizenhatos számrendszerek között mindkét irányba, hogy egyből tudjuk megbecsülni egy hexa szám nagyságrendjét. Mi történt, ha ötjegyű számot próbáltunk beírni? Hát ha két elég nagy négyjegyű szám összegét számítottuk ki? Az ötjegyű szám beírása esetén hibaüzenetet kaptunk, míg az ötjegyű eredmény első számjegye egyszerűen hiányzott. E furcsa jelenségre magyarázatot kaphatunk, ha megismerjük a számok ábrázolásának módját a számítógépben.

A mikroprocesszor minden képessége ellenére meglehetősen buta, csak a 0 és 1 számjegyeket ismeri, így minden számot amit használ 0 és 1 számjegyek hosszú sorából kell összeraknia, ez a bináris számrendszer. A Debug a számok hex formában való kiírásához egy rövid programot használ, amely a belső bináris számokat hexadecimálisra alakítja. A továbbiakban a bináris számokat a szám után írt b betűvel, míg a hexa számokat h betűvel jelöljük. Például: $1011b = Bh = 11$ $1111b = Fh = 15$. $1111b$ egyben a legnagyobb előjel nélküli négyjegyű bináris szám, míg $0000b$ a legkisebb. Tehát négy bináris számjeggyel 16 különböző számot tudunk kifejezni. Mivel éppen 16 hex számjegy van, így négy bináris számjeggyel tudunk kifejezni egy hex számjegyet. Egy kétjegyű hex szám, mint pl. a $4Ch = 0100\ 1100b$, leírható nyolc bináris számjegy segítségével. Az egyes bináris számjegyeket "bit" néven ismerik, a fenti szám tehát nyolc bit hosszú. Egy nyolc bináris számjegyből álló csoport neve "byte", míg egy 16 bitből, azaz két byte-ból álló sorozat elnevezése "szó".

Most már láthatjuk, miért kényelmes a hexadecimális jelölés: két hex számjegy fér el egy byte-ban, négy egy szóban. Ugyanez nem mondható el a tízes számrendszerről. Ugyanezzel magyarázható az is, hogy nem sikerült öt számjegyű hexadecimális számot kiíratnunk a Debug-al, ugyanis a Debug byte illetve szó hosszúságú számokat tud csak kezelni.

[Hexaritmetika gyakorló](#)

[Top](#)

[3] Negatív számok

Az előző példák során egy másik furcsaságra is felfigyelhettünk. 2-ből 3-at kivonva nem -1-et, hanem FFFFh-t kaptunk eredményként. Próbáljuk ki ismét ezeket az értékeket! Ha FFFFh-t tízes számrendszerre alakítjuk, 65535-öt kapunk eredménynek. Mi köze ennek az értéknek a -1-hez? A H 5 FFFF parancs eredményeként 4 és 6-ot kapunk, tehát az FFFFh -1-ként viselkedik. Próbáljuk meg papíron összeadni a fenti két számot:

0005h +

FFFFh

10004h

Ha az ötödik számjegyként kapott 1-et nem vesszük figyelembe, (amint a Debug teszi), megkaptuk a helyes eredményt. Tehát a "túlcsordulástól" eltekintve az FFFFh -1-ként viselkedik. Azért túlcsordulás (overflow) a neve, mert a szám most valójában öt számjegy hosszúságú, de a Debug csak az utolsó négy számjegyet tartja meg.

Most hát ez a túlcsorduláshiba, vagy a válasz helyes? Tulajdonképpen igen és igen, mindkét kérdésre. Nem mondanak ezek azonban ellent egymásnak? Igazából nem, mert kétféleképpen nézhetjük ezeket a számokat. Tegyük fel, hogy FFFFh egyenlő 65535-tel. Ez egy pozitív szám, és egyúttal a legnagyobb szám, amit négy hex számjeggyel ki tudunk fejezni. Azt mondjuk, hogy az FFFFh egy előjel nélküli szám. Ebben az esetben az FFFFh-hoz 5-öt adva az eredmény 10004h, semmilyen más válasz nem helyes. Előjel nélküli számok esetén tehát egy túlcsorduláshiba lép fel.

Másrészt viszont az FFFFh-t negatív számként is értelmezhetjük, ahogy a Debug is tette, amikor a H utasítást használtuk. FFFFh -1-ként viselkedik, amíg eltekintünk a túlcsordulástól. Valójában a 8000h és FFFFh közötti számok mind negatív számként viselkednek. Előjeles számok esetén a túlcsordulás nem hiba. A mikroprocesszor a számokat előjel nélküliként vagy előjelesként is képes kezelni, a választás joga a miénk.

Nézzük meg közelebbről a pozitív és negatív számok bináris alakját:

-pozitív számok:

0000h 0000 0000 0000 0000b

...

....

7FFFh 0111 1111 1111 1111b

-negatív számok:

8000h 1000 0000 0000 0000b

...

....

FFFFh 1111 1111 1111 1111b

Észrevehetjük, hogy pozitív számok esetén a bal szélső bit (15. bit) mindig 0, negatív számok esetén pedig mindig 1. Ez a különbség hozza a mikroprocesszor tudomására, hogy mikor negatív egy szám: megnézi a 15. bitet, az előjelbitet.

Ezek a negatív számok a pozitívak ún. kettes komplementesei (kiegészítik az adott pozitív számot 10000h-ra). A számítógép a negatív számokat komplementer kódban ábrázolja. Pozitív és negatív számok kódjai között az átalakítást elvégezhetjük akár úgy, hogy az adott kódot kivonjuk 10000h-ból, akár formálisan, két lépésben: a binárisan felírt szám minden számjegyét "megfordítjuk" (0 helyett 1-et, 1 helyett pedig 0-t írunk), majd hozzáadunk 1-et.

Például: -3C8h komplementer kódját megkaphatjuk:

3C8h = 0000 0011 1100 1000b

első lépésben 1111 1100 0011 0111 lesz,

majd 1111 1100 0011 0111+

1

-3C8h = 1111 1100 0011 1000b = FC38h

Ellenőrizzük a Debug segítségével, 0-ból 3C8-at kivonva! Hasonlóképpen kaphatjuk meg egy negatív szám komplementes kódjából a neki megfelelő pozitív számot.

[Negatív számok gyakorló](#)

[Top](#)

[4] Regiszterek

A regiszterek kis memóriaterületek, amelyekben számok tárolhatók. Hasonlítanak a magas szintű programozási nyelvekben szereplő változókhoz, de nem teljesen azonosak azokkal. A mikroprocesszor rögzített számú regisztert tartalmaz, és ezek a regiszterek nem képezik a számítógép memóriájának részét (a mikroprocesszor alkotó elemei).

A Debug R (regiszter) utasítása megjeleníti a képernyőn a regiszterek tartalmát. Próbáljuk ki! Az első négy - AX, BX, CX, DX - általános célú regiszterek, míg a többi - SP, BP, SI, DI, DS, ES, SS, CS és IP - különleges célú regiszterek. A regiszter nevét követő négyjegyű szám hex alakban látható. Tehát egy regiszter két byte, azaz szó hosszúságú (legalábbis az I8086, I8088-as mikroprocesszoroknál).

Említettük, hogy a regiszterek hasonlítanak a változókhoz, tehát tartalmuk megváltoztatható. Ha a Debug R utasítását egy regiszter neve követi, azt közöljük a Debuggal, hogy meg akarjuk tekinteni, majd változtatni az adott regiszter tartalmát. A parancs kiadása után megjelenik a regiszter tartalma, majd a következő sorban ":" jelzi, hogy a program várja az új értéket. Például:

-R AX

AX 0000 a gép válasz

:3A7 mi írjuk be

Így bármely regiszter tartalmát tetszőlegesen megváltoztathatjuk. Az R utasítás segítségével ellenőrizhetjük a regiszterek új értékeit.

[Regiszterek gyakorló](#)

[Top](#)

[5] Memória

Most már be tudunk írni két számot az AX illetve BX regiszterbe. Hogyan mondhatjuk meg a processzornak, hogy adja össze őket és az eredményt őrizze meg az AX regiszterben? Beteszünk néhány számot a számítógép memóriájába. A számítógép legalább 640 Kb memóriával rendelkezik. E hatalmas memóriaterület egyik sarkába két byte-nyi gépi kódot teszünk. Ebben az esetben a gépi kód két bináris szám lesz, amelyek megmondják a processzornak, hogy adja a BX regisztert AX-hez. Utána, hogy lássuk mi történik, ezt az utasítást a Debug segítségével végrehajtjuk.

De melyik részébe helyezzük a memóriának két byte-os utasításunkat, és hogyan mondjuk meg a processzornak, hogy hol keresse? A processzor a memóriát 64 Kbméretű részekre osztja, amelyeket szegmenseknek nevezünk. Többnyire egy szegmensen belüli memóriával foglalkozunk, de anélkül, hogy tudnánk, hol kezdődik a szegmens. Ezt a processzor memóriamegjelölő módja teszi lehetővé.

A memóriában levő minden byte meg van jelölve egy számmal, 0-val kezdődően, növekvő sorrendben. A legnagyobb négyjegyű hex szám (egy regiszter kapacitása) 65535, tehát a 64K a címkék felső korlátja. Egy apró trükk alkalmazásával a processzor ennél lényegesen nagyobb memóriaterületet tud címezni: két számot használ, egyet a 64Kb-os szegmensek jelölésére, és egyet mindegyik byte vagy offset (eltolás) jelölésére a szegmensen belül. Mindegyik szegmens 16 byte többszörösénél kezdődik. Így a processzor kezelni tudja az egy millió byte-ig (1Mb) terjedő memóriát.

Az általunk használt címek a szegmens elejétől számított offset (eltolás) címek. A címeket úgy írjuk, hogy megadjuk a szegmens számot, amelyet a szegmensen belüli offset követ. Például a 3756:0100 azt jelenti, hogy a 100h offsetnél vagyunk a 3756h szegmensben belül. Egyelőre a Debug gondoskodik róla, hogy egy szegmensben dolgozhassunk, a címeknél csak az eltolást adjuk meg. E címek mindegyike egy szegmensben belüli byte-ot jelöl, és a címek növekvő sorrendűek.

Két byte-os utasításunk kiírva a BX és AX összeadására így néz ki: ADD AX, BX. Ezt az utasítást a 100h és 101h memóriarekeszekbe írjuk, abba a szegmensbe, amelyet a Debug éppen használ. Amikor az ADD utasításunkra hivatkozunk, azt mondjuk, hogy a 100h helyen van, mivel ez az utasítás első byte-jának a helye. A memória megtekintésére és megváltoztatására az E (Enter, azaz beírás) parancs használható. Használjuk ezt a parancsot az ADD utasítás két byte-jának beírására a következő módon:

-E 100

beírása után megjelenik az aktuális cím (szegmens:offset) és a jelenlegi értéke, majd a "." után beírhatjuk az új értéket

xxxx: 0100 xx.01

-E 101

xxxx: 0101 yy.D8

A 01h és D8h számok az xxxx:0100 és xxxx:0101 memóriahelyeken az ADD utasítás 8088-as gépi kódú megfelelői. A szegmensszám, amit itt xxxx-szel jelöltünk, alkalmanként változik, de ez a programunkat nem befolyásolja. Ugyanígy a

Debug különböző kétjegyű számot jelenít meg mindegyik E utasításnál (az előbbiekben xx ill. yy-nal jelöltük). Ezek a számok a Debug által kiválasztott szegmens 100h és 101h offset címein levő memóiahelyek régi tartalmai - azaz a számok a Debug indítása előtt futtatott program által a memóriában hagyott adatok. Próbáljuk ki a memória írását!

[Memória gyakorló](#)

[Top](#)

[6] Összeadás

Most már be tudjuk állítani a regiszterek tartalmát, azaz be tudjuk írni az operandusokat, és be tudjuk írni a memóriába az összeadás utasítást gépi kódban. Ahhoz, hogy végrehajthassuk az összeadást, még meg kell mondanunk a processzornak, hogy hol keresse az utasítást.

A 8088-as a soron következő utasítás címét két speciális regiszterben, a CS és IP regiszterekben keresi, amelyeket az előző regiszterkiírásnál láthattunk. A szegmensszám a CS-ben (Code Segment) van. Ezt a Debug már be is állította. A cím másik része, az offset az adott szegmensben, az IP (Instruction Pointer) regiszterben van. Az IP regisztert utasításunk kezdőcíme IP = 100h - állítva mondhatjuk meg, hogy hol keresse a processzor az utasítást. De ha megnézzük, az IP már 0100h értékű. Ravaszok voltunk: a Debug az IP értékét indításkor 100h-ra állítja be. Ezért akarattal választottuk az 100h címet az első utasításunk kezdőcímeül, így kiküszöbölhettük az IP regiszterkülön állítását. Ezt a megoldást érdemes megjegyezni!

Most, hogy az utasítást a helyére tudjuk írni és a regisztereket be tudjuk állítani, megmondjuk a Debugnak, hogy hajtsa végre egyetlen utasításunkat. A Debug T (Trace - nyomkövetés) utasítását használjuk, amely egy utasítást hajt végre, majd kiírja a regiszterek tartalmát. Minden egyes végrehajtás után az IP-nek a következő utasításra kell mutatnia. Ebben az esetben 102h-t mutat. Nem írtunk utasítást a 102h helyre, így a regiszterkiírás utolsó sora valamely korábbi program utasítását tartalmazza. Írjunk be egy utasítást és kövessük nyomon végrehajtását a T paranccsal. Ha újra T parancsot adunk, ez a soronkövetkező (102h címen levő) utasítást hajtja végre, amit egyelőre még nem mi kértünk.

Ha az előző összeadást meg akarjuk ismételni, még egyszer hozzá akarjuk adni BX értékét AX-hez, akkor ismét 100h-ra kell állítsuk az IP regisztert egy R parancs segítségével. Próbáljunk ki néhány összeadást!

[Összeadás gyakorló](#)

[Top](#)

[7] A négy alapl művelet

Amint az összeadásnál láthattuk, a processzor két regiszter tartalmát adja össze és az eredményt az első regiszterben őrzi meg. Ez a többi alapl műveletre is érvényes. Az operandusok regiszterekben vannak, az eredményt szintén egy vagy két regiszterben kapjuk meg. Amikor az ADD utasítás két byte-ját beírtuk, kétszer használtuk az E parancsot: egyszer a 0100h címre írtuk be a 01h értéket, majd a 0101h címre a D8h értéket. A módszer működött, de lehetőségünk van arra is, hogy ezt egyetlen E parancssal megtegyük. Ha az első byte beírása után <Enter> helyett szóközkaraktert írunk, megjelenik a következő byte tartalma és folytathatjuk a beírást több byte-on keresztül, az értékeket szóközzel elválasztva. A bevitt <Enter>-rel fejezzük be.

A kivonás (SUB AX, BX) kódja 29D8h. Amint láthatjuk ez is egy két byte-os utasítás. Próbáljuk egyetlen E parancssal beírni. Hatására a processzor az AX regiszter tartalmából kivonja a BX regiszter tartalmát, és az eredményt az AX regiszterben őrzi meg. Próbáljunk a második byte-nak D8h helyett más értéket adni. Melyik regiszterekre vonatkozik az utasítás ezekben az esetekben? Ellenőrizhetjük az R parancs segítségével, a regiszterek alatti sor. Eddigi számolásainkat szavakkal végeztük, azaz négy hex számjeggyel. Vajon képes-e a mikroprocesszor byte-okkal is számolni? A válasz: igen. Mivel egy szó byte-okból tevődik össze, így mindegyik általános célú regiszter két byte-ra osztható, amelyeket felső byte-nak (high byte - az első kétszámjegy) és alsó byte-nak (low byte - a második két hex számjegy) nevezünk. E regiszterek mindegyikére a betűjelével hivatkozhatunk (A-tól D-ig), amelyet szó esetén X, felső byte esetén H, alsó byte esetén L követ. Így például DL és DH byte regiszterek, míg DX egy szóregiszter.

Próbáljuk ki a byte méretű számtant az ADD utasítással! Írjuk be a 00h és C4h két byte-ot a 0100h címtől kezdve! A regiszterkiírás alján az ADD AH, AL utasítást láthatjuk, amely összeadja az AX regiszter két byte-jának tartalmát, és a felső byte-ba, az AH-ba helyezi az eredményt. Hasonló képpen végezhetjük a kivonást is byte-okkal. Például a SUB AH, BL kódja 28h DCh. Hatására a processzor az AH regiszter tartalmából kivonja a BL regiszter tartalmát és az eredményt az AH regiszterben őrzi meg. Próbáljuk meg a fenti utasításokat úgy, hogy a második byte-ot módosítjuk. Melyik regiszterekre vonatkoznak így az utasítások? A Debug nem teszi lehetővé, hogy az R parancssal egy byte-os regisztert módosítsunk, így mindig a szó hosszúságú regisztert kell beírjuk, akkor is ha byte-os utasításhoz készítjük azt elő. A szorzási művelet neve a MUL, és a gépi kód az AX és BX összeszorozására F7hE3h. Ezt kell beírni a memóriába, de előbb ejtsünk néhány szót a szorzásról! Hol tárolja a MUL utasítás az eredményt? Az AX regiszterben? Nem egészen; itt óvatosnak kell lennünk. Két 16 bites szám szorzásakor az eredmény 32 bites lehet, így a MUL utasítás két regiszterben, DX-ben és AX-ben tárolja azt. A felső 16 bit a DX regiszterben van, míg az alsó 16 bit az AX-ben. Ezt regiszterkombinációt időnként DX : AX-nek jelöljük.

Térjünk vissza a Debug-hoz. Írjuk be a szorzás utasítást, az F7h E3h értéket a 0100h címre, és állítsuk be az AX és BX regisztert, az AX-nek adjunk négyjegyű értéket. A regiszterkiírásban a MUL BX utasítást látjuk, az AX regiszterre való bármilyen utalás nélkül. Szavak szorzásakor a 8088-as mindig az utasításban szereplő regiszter tartalmát szorozza az AX regiszterrel és az eredményt a DX:AX regiszterpárban tárolja. Használjuk a Debug-ot az utasítás nyomon követésére. Végezzük el az adott műveletet kézzel is. Próbáljuk ki különböző kisebb és nagyobb értékekre! Kövessük figyelemmel a DX regiszter változását a különböző esetekben!

Byte-ok szorzása hasonlóan történik. Két 8 jegyű szám szorzásának eredménye legfeljebb 16 jegyű szám lehet. A byte szintű szorzás az egyik operandust mindig az AL regiszterből veszi és az eredményt mindig az AX regiszterben őrzi meg. Szorozzuk össze az AL regisztert a BL-lel. A MUL BL utasítás gépi kódja: F6h E3h. Kísérletezzünk különböző értékekkel, kézzel is elvégezve a szorzásokat. Mi a helyzet az osztással? Amikor osztunk a processzor megőrzi mind az eredményt, mind a maradékot. Mint a MUL, a DIV is megkérdezés nélkül használja a DX : AX regiszterpárt. Ha szó hosszúságú az osztó, az osztandó mindig a DX : AX regiszterpárban van, a hányados az AX regiszterbe, míg a maradék a DX regiszterbe kerül. A DIV BX gépi kódja F7h F3h. Írjuk be és próbáljuk ki különböző nagyságú operandusokkal! Mi történik, ha egy nagy számot (például DX=1234h, AX=5678h) egy kis számmal (pl. BX=100h) osztunk?

Ha az osztó szó hosszúságú regiszter, a processzor az osztandót mindig a DX : AX regiszterpárból veszi, akkor is, ha az osztandó nem hosszabb 16 bit-nél. Ebben az esetben le kell nullázni a DX regisztert, különben az ott talált "hulladékot" az osztandó magasabb helyiértékű 16 bit-jének tekinti a processzor, és az eredmény lényegesen el fog térni az általunk elvárttól. A szorzással ellentétben, ahol két 16 bit-es szorzótényező szorzata nem lehet 32 bit-nél hosszabb, az osztásnál előfordulhat, hogy a hányados hosszabb 16 bit-nél (nagy osztandó, kicsi osztó). Ilyenkor "0-val való osztás" hibát kapunk.

Ha az osztó byte hosszúságú, az osztandó az AX regiszterben kell legyen, a hányadost az AL regiszterben kapjuk meg, a maradékot pedig az AH-ban. A DIV BL kódja: F6h F3h. A fent leírt szorzás és osztás előjel nélküli egész számokra vonatkozik. Előjeles számok szorzására és osztására az IMUL illetve IDIV utasításokat használhatjuk, melyek hasonló képpen működnek.

[A négy alapművelet gyakorló](#)

[Top](#)

[8] Megszakítások

A négy alapl műveletet el tudjuk már végeztetni a számítógéppel, csak arra kell még rávegyük, hogy ki is írja az eredményt. Lássuk először, hogyan írathatunk ki egy karaktert. Az INT utasítás segítségével utasíthatjuk a DOS-t, hogy egy karaktert írjon ki a képernyőre. Az INT utasítás meghív egy eljárást, amely elvégzi a kiírást. Mielőtt megtanuljuk, hogyan működik az INT, fussunk végig egy példán! A Debugban tegyünk 200h-t az AX regiszterbe és 41h-t a DX-be. Az INT utasítás a DOS funkciókra INT 21h - gépi kódban CDh 21h. Tegyük ezt a két byte-os utasítást a memóriába az 100h címtől kezdődően. Az R paranccsal ellenőrizzük, hogy az utasítás valóban az INT 21h.

Most már készen állunk az utasítás végrehajtására, de nem használhatjuk a nyomkövetés (T) parancsot úgy, mint az eddigiekben. A nyomkövetés egy utasítást hajt végre egy lépésben, de az INT utasítás egy nagyobb programot hív meg a DOS-ból a tényleges munka elvégzésére. Nem akarjuk a DOS eljárásban levő összes utasítást lépésenként nyomon követve, egyenként végrehajtani. Helyette futtatni akarjuk egysoros programunkat, de leállva, mielőtt végrehajtanánk a 102h címen levő utasítást. Ezt a Debug G (Go till) parancsával tehetjük meg, amelyet az a cím kövessen (töréspont), ahol meg akarunk állni.

-G 102A

DOS kiírja az A karaktert, majd visszaadja a vezérlést rövid programunknak. Próbáljuk ki! Vajon honnan tudta a DOS, hogy az A karaktert jelenítse meg? Az AH regiszterben levő 02h mondta meg neki, hogy egy karaktert jelenítsen meg. Az AH-ban egy másik szám egy másik művelet végrehajtására szólítja fel a DOS-t. A DOS a DL regiszterben levő számnak megfelelő karaktert írja ki a képernyőre, ez a szám a kiírandó karakter ASCII kódja. A 41h az A karakter ASCII kódja.

Ha kíváncsi vagy arra, mi történne, ha a nyomkövetés parancsot használnánk az INT utasítás végrehajtására írd vissza az eredeti 100h értéket az IP regiszterbe, majd kezd el az utasítások végrehajtását a T paranccsal. Ha meguntad lépj ki Q-val a Debug-ból. Az INT parancs végrehajtása után megváltozik a cím első száma is. A következő utasítások a DOS részei, és a DOS programja egy másik szegmensben van. A valóságban a DOS sok utasítást hajt végre mielőtt kiírhatna egy karaktert. Még egy ilyen egyszerűnek látszó feladat sem olyan egyszerű, mint amilyennek tűnik. Most már érthető, hogy azért használtuk a G parancssort a program 102h címig való futtatásához, mert különben DOS utasítások özöne árasztott volna el.

Ha az INT 21h utasításban kicseréljük a 21h-t 20h-ra az INT 20h utasítást kapjuk. Ez egy másik megszakítás utasítás, amely megmondja a DOS-nak, hogyki akarunk lépni a programunkból, hogy a DOS vehesse át a vezérlést. Esetünkben az INT 20h a Debug-nak adja vissza a vezérlést, hiszen programjainkat a Debug-ból indítjuk, nem a DOS-ból. Írjuk be az INT 20h (azaz CDh 20h) utasítást az 100h címtől kezdődően, majd indítsuk a G paranccsal. A G parancs, amelyet nem követ szám, az egész programot végrehajtja és visszatér a kezdethez. Az IP értéke újra 100h, ahol elkezdtük. Ellenőrizzük a regisztereket!

[Megszakítások gyakorló](#)

[Top](#)

[9] Programok beírása

Írjuk be a 100h címtől kezdve az INT 21h és INT 20h (CDh 21h CDh 20h) utasításokat egymás után. (ezentúl minden programot az 100h címnél kezdünk). Amíg csak egy utasításunk volt, "kilitázhattuk" az R paranccsal, most azonban két utasításunk van. Ezek megtekintésére az U (Unassemble - visszafordít) parancsot használjuk. Az első két utasítást felismerjük, ezeket most írtuk be. A többi utasítás csak maradék a memóriában.

Most töltsük fel az AH regisztert 02h-val, a DL regisztert pedig tetszőleges karakter kódjával, majd egyszerűen gépeljük be a G parancsot, hogy lássuk a karaktert. Kísérletezzünk a karakterek kódjaival! Ezentúl programjaink többsége egynél több utasítást tartalmaz, és e programok megjelenítéséhez az U parancsot használjuk.

Eddig programjaink utasításait közvetlenül számok formájában írtuk be, mint a CDh 21h. De ez nagy munka és van egy sokkal egyszerűbb módja az utasítások beírásának. Az U (unassemble) parancs mellett a Debug tartalmaz egy A (Assemble - fordít) parancsot is, amely lehetővé teszi, hogy egyenesen mnemonikus, azaz ember számára érthető formában írjunk be utasításokat. Így ahelyett, hogy azokat a rejtélyes számokat íránk be a rövid programunkhoz, használhatjuk az A parancsot a következők beírására:

-A 100

xxxx:0100 INT 21

xxxx:0102 INT 20

xxxx:0104

-

Amikor befejeztük az utasítások beírását, csak az Enter billentyűt kell lenyomni, és a Debug prompt újra megjelenik.

Az A utasítás mondta meg a Debug-nak, hogy az utasításokat mnemonikus alakban akarjuk beírni. A 100 azt jelenti, hogy a következő utasításokat a 100h címtől kezdve akarjuk írni. Noha idáig a Debug-ot használtuk, programjainkat nem mindig vele fogjuk futtatni. Általában egy program maga állítja be az AH és DL regisztereket egy INT 21h utasítás előtt. Ehhez egy másik, a MOV utasítással ismerkedünk meg.

Hamarosan a MOV utasítással visszük be a számokat az AH és DL regiszterekbe. Először azonban azt tanuljuk meg, hogyan lehet a MOV segítségével a regiszterek között mozgatni a számokat. Tegyük 1234h-t az AX-be, ABCDh-t a DX-be. Most írjuk be az A paranccsal a MOV AH, DL utasítást. Ez a DL-ben levő számot az AH-ba mozgatja, azaz egy másolatát elhelyezi az AH regiszterbe; a DL nem változik. Próbáljuk ki! Végrehajtás előtt ellenőrizzük a regisztereket! Noha van néhány korlát, a MOV utasítást használhatjuk arra, hogy számokat más regiszterpárok között másoljunk. Például a MOV AX, BX utasítással szavakat másolhatunk, a BX regiszterből az AX-be. A MOV utasítás mindig szó és szó, vagy byte és byte között működik, sohasem szavak és byte-ok között.

A MOV egy másik formájával számokat vihetünk be regiszterbe. Például a MOV AH,02 a 02 értéket helyezi az AH regiszterbe, anélkül, hogy hatással lenne az AL regiszterre. Vigyünk a MOV utasítással különböző értékeket a regiszterekbe. Ne felejtsük el az IP-t beállítani! Most illesszük össze a tanultakat, és írjunk meg egy "hosszabb" programot. Ez egy csillagot ír ki önállóan, anélkül, hogy állítanunk kellene a regisztereket (AH és DL):

```
MOV AH,02
```

```
MOV DL,2A
```

```
INT 21
```

```
INT 20
```

Írjuk be a programot az A paranccsal és ellenőrizzük U-val, hogy helyes-e. Győződjünk meg róla, hogy az IP 100h címre mutat-e, majd a G paranccsal futtassuk a programot! A képernyőn meg kell jelennie a * karakternek. Most, hogy már van egy önálló programunk, írjuk ki a lemezre egy .COM programként, hogy egyenesen a DOS-ból végrehajthassuk! Egy .COM programot a DOS-ból a nevének begépelésével futtathatunk. Mivel programunknak nincs neve, adjunk neki egyet!

A Debug N (Name) parancsa nevet ad egy állománynak, mielőtt a lemezre írjuk. Írjuk be:

```
-N CSILLAG.COM
```

Ez a parancs nevet ad az állománynak, de nem írja ki a programot a lemezre. Utána a Debug-nak egy byte számot kell adni, megmondva a programunkban levő byte-ok számát, hogy tudja mekkora memóriaterületet akarunk beírni a file-ba. Ha megnézzük a programunk szétszedett listáját (U), láthatjuk, hogy négy kétbyte-os utasításunk van, tehát 8 byte hosszú. (A programunk utáni első címből kivonunk 100h-t). Ha már tudjuk a byte-jaink számát, azt valahova el kell tenni. A Debug a BX : CX regiszterpárt használja a file hosszára, így CX-be 8-at, BX-be 0-t téve megmondjuk a Debug-nak, hogy programunk 8 byte hosszú. Ha megadtuk a file nevét és hosszát, kiírhatjuk a lemezre a W (Write) parancs segítségével:

```
-W
```

Most van a lemezen egy CSILLAG.COM állomány, amit DOS-ban futtathatunk. Ehhez egyszerűen gépeljük be a nevét, és a képernyőn megjelenik egy *.

[Programok beírása gyakorló](#)

[Top](#)

[10] Karakterlánc kiírása

Az INT 21h megszakítással egy karakter helyett egy egész karakterláncot is ki tudunk írni, mégpedig úgy, hogy az AH regiszterbe egy másik műveletkódot teszünk. Karakterláncunkat a memóriába kell töltenünk, és meg kell mondanunk a DOS-nak, hogy hol találja meg ezt a karakterláncot. Már láttuk, hogy a 02h funkciószám az INT 21h-nál egy karaktert ír ki a képernyőre. Egy másik funkció, a 09h egy egész sor karaktert ír ki, és csak akkor hagyja abba, ha \$ jelet talál a sorban. Tegyük egy karaktersort a memóriába! A 200h címnél kezdjük, hogy a sor ne keveredjen a programunk kódjával! A gyakorló ablakban megvan a szöveg. Az utolsó beírt szám a 24h lesz, a \$ jel ASCII kódja, ez mondja meg a DOS-nak, hogy itt a karaktersor vége. A következő programot futtatva meglátjuk, mit tartalmaz a szöveg:

```
MOV AH,09
```

```
MOV DX,0200
```

```
INT 21
```

```
INT 20
```

200h a karakterlánc kezdőcíme, és a 200h DX regiszterbe töltésével megmondjuk a DOS-nak, hogy hol keresse ezt a karaktersort. Írjuk be az A parancssal, az U-val ellenőrizzük, majd G-vel futtassuk a fenti programocskát. Most, hogy már beírtunk karaktereket a memóriába, ideje megismernünk egy másik Debug parancsot, a D-t (Dump - memóriakiírás). A D kiírja a memória tartalmát a képernyőre, hasonlóan ahhoz, ahogyan az U kilistázza az utasításokat. Ugyanúgy, ahogy az U parancsnál, egyszerűen írunk egy címet a D parancs után, amely megmondja a Debug-nak, honnan kezdje a kiíratást. Például írjuk be a D 200 parancsot az előbb beírt karakterlánc kiíratására. Minden címpár után 16 db hex byte-ot látunk, amelyet a byte-ok ASCII karakter megfelelője követ.

Ahol pontot látunk az ASCII ablakban, az vagy pont, vagy olyan különleges karaktert jelöl, amelyet a Debug nem ismer fel (256 lehetséges karakterből csak 96-ot ismer fel). A továbbiakban a D parancsot használjuk a bevitt adatok ellenőrzésére.

[Karakterlánc kiírása gyakorló](#)

[Top](#)

[11] Átviteljelző bit (carry flag)

A hexaritmetikával történő ismerkedés során láttuk, hogy ha 1-et adunk hozzá FFFFh-hoz, 10000h-t kellene kapnunk, de mégsem ez történik. Csak a jobboldali négy hex számjegy fér be egy szóba; a 1 már nem. Azt is megtudtuk, hogy ez az 1 az ún. túlcordulás, és nem vész el. De akkor hová tűnik? Egy jelzőbit (flag) nevű valamibe, jelen esetben az átviteljelzőbe (carryflag), azaz CF-be kerül. A jelzők egy bites számokat tartalmaznak, így tartalmuk vagy egy vagy nulla. Ha tovább kell vinnünk egy egyest az ötödik hexa számjegybe, akkor az az átviteljelzőbe kerül.

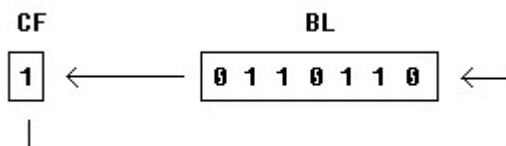
Írjuk be ismét az ADD AX, BX utasítást és tegyünk FFFFh-t AX-be, és 1-et BX-be, majd kövessük nyomon az ADD utasítást. A Debug R képernyőjének alján látni fogunk nyolc betűpárt. Ezek közül az utolsó - amely vagy NC vagy CY - az átviteljelző. Most, mivel az összeadás 1 túlcordulást eredményezett, látható, hogy az átviteli állapot CY (carry - átvitel). Az átvitelbit most 1 (vagy másképpen mondva be van állítva).

Annak ellenőrzésére, hogy valóban egy tizenhetedik bitet tárolunk itt, adjunk hozzá az AX-ben levő nullához 1-et az ADD utasítással. Az állapotjelzőre minen egyes összeadás hatással van, és most nem kell túlcordulásnak lennie, így az állapotjelzőnek új értéket kell kapnia. Valóban, az átvitel nulla lesz, ahogyan az NC jelzi a regiszterkiírás alján, ami a nincs átvitel (no carry) megfelelője. Hogy lássuk az átvitelinformáció hasznát, nézzük most át a bináris számok kiírásának feladatát. Egyszerre csak egy karaktert íratunk ki, és egyenként akarjuk leszedni a biteket a számunkról, balról jobbra. Például az első kiírandó karakter az 1000 0000h számnál az egyes lenne. Ha ezt az egész byte-ot egy hellyel balra mozdítanánk, az egyest az átviteljelzőbe ejtve és egy 0-t adva a jobb oldalra, majd az egész folyamatot ismételnénk a további számjegyekre, az átviteljelző leszededegetné a bináris számjegyeinket. Ezt megtehetjük egy új utasítással, amit RCL-nek nevezünk (Rotate Carry Left - átvitel forgatás balra).

Működésének bemutatásához írjuk be a következő utasítást:

RCL BL,1

Ez az utasítás a BL-ben levő byte-ot egy bittel balra forgatja (ezért az ,1 az utasításban), és ezt az átviteljelzőn keresztül teszi. Az utasítást azért nevezzük forgatásnak, mert a RCL a bal szélső bitet az átviteljelzőbe viszi, míg az éppen az átviteljelzőben levő értéket átviszi a jobb szélső (0) bit-pozícióba. A folyamat közben a többi bit is elmozdul, azaz balra forog. Megfelelő számú forgatás után (17 egy szóhoz, 9 egy byte-hoz) a bitek visszajutnak eredeti pozíciójukba, és visszakapjuk az eredeti számot.



Tegyünk B7h-t a BX regiszterbe, majd kövessük nyomon több alkalommal a forgatás utasítást. Az eredményeket bináris formába alakítva a következőket látjuk:

átvitel BL regiszter

átvitel BL regiszter

NC 0	1011 0111b	B7h eredeti szám	NC 0	1110 1011b	EBh
CY 1	0110 1110b	6Eh	CY 1	1101 0110b	D6h
NC 0	1101 1101b	DDh	CY 1	1010 1101b	ADh
CY 1	1011 1010b	BAh	CY 1	0101 1011b	5Bh
CY 1	0111 0101b	75h 9. forgatás:	CY 0	1011 0111b	B7h

Az első forgatáskor a 7. bit átmegy az átviteljelzőbe, az átviteljelző tartalma pedig átlép a BL 0. bitpozíciójába, míg az összes többi bit egy pozícióval balra vándorol. A további lépésekben a forgatás folytatásával a bitek tovább vándorolnak balra, kilenc forgatás után pedig újra az eredeti szám van a BL regiszterben.

Egyre közelebb kerülünk bináris számokat kiíró programunk befejezéséhez, de még hiányzik néhány építőköck. Lássuk, hogyan alakíthatjuk az átviteljelzőben levő bitet a 0 vagy 1 karakterekké? A normál ADD utasítás, mint pl. ADD AX, BX egyszerűen összeadja a két számot. Egy másik utasítás, az ADC (Add with Carry - összeadás átvitelrel) három számot ad össze: a két számot, mint az előbb, plusz az egy bitet az átviteljelzőből. Ha belenézünk az ASCII táblába, láthatjuk, hogy a 30h a "0" karakter és 31h az "1" karakter kódja. Így, ha a 30h-hoz hozzáadjuk az átviteljelzőt, akkor a "0" karaktert kapjuk, ha az átvitel üres, és az "1" karaktert, ha az átvitel beállított. Tehát, ha a DL=0 és az átviteljelző bevan állítva (1), végrehajtva az:

ADC DL,30

utasítást, a DL-hez (0) hozzáadjuk 30h-t ("0") és az 1h-t (átviteljelző), 31h-t ("1") kapunk. Így egy utasítással átalakítjuk átvitelünk értékét egy kiírható számjeggyé.

[Átviteljelző bit \(carry flag\) gyakorló](#)

[Top](#)

[12] Ciklus képzése

Gyakran szükségünk van bizonyos utasításcsoport ismétlésére. Ezt az előzőfejezetben úgy oldottuk meg, hogy az RCL BX, 1 utasítást 9-szer írtuk be. Ennek elkerülésére használjuk a ciklus, azaz LOOP utasítást. A LOOP-nak meg kell mondanunk, hányszor szaladjon újra végig a cikluson. Ezt úgy tehetjük meg, hogy az ismétlések kívánt számát a CX regiszterbe tesszük. A ciklus minden végrehajtásakor a processzor levon egyet a CX-ből, és amikor a CX értéke nulla lesz, a LOOP befejezi a ciklust.

Miért a CX regiszter? A C a CX-ben a Count (számlálás) rövidítése. Ez a regiszter általános célú regiszterként is megfelel, de gyakran más utasításokkal együtt használjuk, ha utasításokat akarunk ismételni.

A LOOP utasítás bemutatására írjunk egy rövid programot, amely a BX regisztert 4-szer balra forgatja, a BL első 4 bitjét átforgatva BH-ba.

```
100  MOV BX,006C
103  MOV CX,0004
106  RCL BX,1
108  LOOP 0106
10A  INT 20
```

A ciklus az 106h címnél kezdődik (RCL BL,1), és a LOOP utasítással ér véget. A LOOP utasítást követő szám (106h) a ciklus kezdőcíme. Amikor a programot futtatjuk a LOOP levon egyet a CX-ből, majd az 106h címre ugrik, ha a CX nem nulla. A RCL BX,1 utasítás itt 4-szer hajtódik végre, mert a CX értékét a ciklus előtt 4-re állítottuk be. A programot lépésenként végrehajtva követhetjük a CX regiszter változását. Ha egyszerűen G-vel hajtjuk végre a programot nem észlelünk semmit, mert a G az INT 20h utasítás hatására visszaállítja a regisztereket az eredeti állapotba. Ha mégis szeretnénk egyben végigfuttatni programunkat, a G 10A parancsot használjuk, így leállva az INT 20h előtt. Ekkor láthatjuk a regiszterek új értékét.

[Ciklus képzése gyakorló](#)

[Top](#)

[13] Bináris szám kiírása

Láttuk már, hogyan vehetjük sorba a bináris számjegyeket, és hogyan alakíthatjuk őket ASCII karakterekké. Ha ezt kiegészítjük a számjegyeink kiírására szolgáló INT 21h utasítással, programunk elkészült.

```
100  MOV  AH,02
102  MOV  CX,0008
105  MOV  DL,00
107  RCL  BL,1
109  ADC  DL,30
10C  INT  21
10E  LOOP 105
110  INT  20
```

Láttuk, hogyan működnek külön-külön a részek, most pedig összeraktuk őket. A BL regisztert forgatjuk (a RCL BL,1 utasítással), hogy sorba leszedjük a biteket a számról, így töltjük be azt a számot a BL-be, amelyet binárisformában akarunk kiírni, majd futtassuk a programot a G paranccsal. Az INT20h utasítás után a G parancs visszaállítja a regisztereket eredeti formájukba, így BL értéke még mindig az a szám, amelyet binárisan kiírva látunk. Az első utasítás az AH értékét állítja be 02-re az INT 21h híváshoz (a 02 azt mondja meg a DOS-nak, hogy a DL-ben levő karaktert írja ki). Az ADC DL, 30 utasítás a programban átalakítja az átviteljelzőt egy nulla vagy egyes karakterre (előzőleg a MOV DL, 0 utasítás a DL értékét nullára állítja). Ha kíváncsi arra, mi történik a program futása közben, kövesse nyomon! Tartsa fejből azonban, hogy érdemes kissé vigyázni a T paranccsal való egyenkénti lépegetéssel! A program tartalmaz egy INT 21h utasítást, és (ahogy láttuk az INT 21h-val való első találkozásnál) a DOS elég sok munkát végez ezzel az egyetlen utasítással. Ezért nem használható a T az INT 21h-val.

A többi utasítás, a végső INT 21h kivételével, ennek ellenére nyomon követhető. A nyomkövetés közben minden alkalommal, amikor a hurok az INT 21h utasításhoz ér, írja be a G 10E parancsot. A G parancs, ha cím követi, megmondja a Debug-nak, hogy folytassa a program futtatását, de álljon meg, amikor az IP eléri a megadott címet. A szám, amelyet a G utasítás után beírunka töréspont (breakpoint) nevet viseli és nagyon hasznos, amikor egy program belső működését akarjuk megismerni. Akár végigpróbáltuk a program nyomonkövetését, akár nem, láthatjuk, hogy egy G 10E-hez hasonló utasítás enged átlépni egy olyan INT utasításra, amely mondjuk a 10Ch címen kezdődik. Ez azonban azt jelenti, hogy minden alkalommal, amikor egy INT utasítást nyomon akarunk követni, tudnunk kell a következő utasítás címét. Létezik egy Debug utasítás, amely nagyon leegyszerűsíti egy INT utasítás nyomon követését. A P (Proceed - folytassa) parancs elvégzi helyettünk az egész munkát (nem követi nyomon a meghívott eljárást vagy ciklust). Kövessük nyomon az egész programot, de ha elérjük az INT 21h utasítást G 10E helyett írjuk a P parancsot.

A P parancsot gyakran fogjuk használni, mivel ez a legkényelmesebb mód olyan utasítások nyomon követésére (mint pl. az INT), amelyek hosszabb eljárásokat hívnak meg (pl DOS rutinokat).

[Bináris szám kiírása gyakorló](#)

[Top](#)

[14] Állapotjelző bitek

Az előbbi fejezetekben már megtudtunk valamit az állapotjelző bitekről, és megvizsgáltuk az átviteljelzőt, ami vagy CY vagy NC jelöléssel volt kiírva a Debug R képernyőjén. A többi jelzők, amelyek ugyanennyire hasznosak, az előző aritmetikai utasítások állapotát követik figyelemmel. Összesen nyolc van belőlük, és majd akkor tanuljuk meg használatukat, ha szükség lesz rájuk. Emlékezzünk vissza, a CY azt jelenti, hogy az átviteljelző 1, azaz be van állítva, az NC pedig azt, hogy az átviteljelző 0. Az összes jelzőnél az 1 jelentése igaz, míg a 0 é hamis. Például, ha egy SUB utasítás eredménye 0, a nullajelző (zero flag) nevű jelző értéke 1 (igaz) lenne, és ez az R képernyőn ZR (zero - nulla) jellel látszana. Egyéb esetben a nullajelző értéke NZ-re (not zero - nem nulla) lenne visszaállítva.

Lássunk egy példát, amely ellenőrzi a nullajelző működését. A SUB utasítást használjuk két szám kivonására. Ha a két szám egyenlő, az eredmény nulla, és a nullajelző ZR formában jelenik meg a képernyőn. Írjuk be a SUB AX, BX utasítást, és kövessük nyomon néhány különböző számmal. Figyeljük meg, hogy ZR vagy NZ jelenik-e meg a képernyőn. Ha kivonunk egyet nullából, az eredmény FFFFh, ami a kettes komplementes formában -1. Az R képernyő alapján meg tudjuk-e mondani egy aritmetikai művelet eredményéről, hogy pozitív vagy negatív? Igen, egy másik jelző, az előjeljelző (sign flag) értéke 1, ha egy eredmény egy negatív kettes komplementes szám. Az R képernyőn NG (negatív) illetve PL (pozitív) formában jelenik meg. Egy másik új jelző, ami érdekelhet bennünket, a túlcsordulásjelző (overflowflag), ami az OV (Overflow - túlcsordulás) és az NV (No overflow - nincs túlcsordulás) állapotokat veheti fel. A túlcsordulásjelző akkor van beállítva, ha az előjelbit olyankor változik, amikor nem kellene. Például, ha két pozitív számot, mint a 7000h és a 6000h, összeadunk, egy negatív szám, D000h (azaz -12288) az eredmény. Ez hiba, mert az eredmény túlcsordul a szón. Az eredménynek pozitívnak kellene lennie, de nem az, így a processzor beállítja a túlcsordulásjelzőt. (Emlékezzünk, ha előjel nélküli számokkal dolgoznánk, ez nem lenne hiba, és akkor figyelmen kívül hagynánk a túlcsordulásjelzőt.)

Próbálkozzunk több számmal, hogy megbizonyosodjunk arról, be tudjuk-e állítani, majd újraállítani a jelzők mindegyikét, egészen addig, míg biztosak nem leszünk a dolgunkban. Most már készen állunk, hogy megnézzünk egy sor más utasítást, amelyeket feltételes ugrás utasításokként ismerünk. Ezek lehetővé teszik számunkra, hogy az állapotjelzőket az eddig megismerteknél kényelmesebben ellenőrizhessük. A JZ utasítás (Jump if zero - ugrás nulla esetén) új címre ugrik, ha az előző aritmetikai művelet eredménye nulla volt. Így ha egy SUB utasítást pl a JZ 15A utasítás követ, a kivonási művelet nulla eredménye esetén a processzor az 15Ah címre ugrik, és az ott található utasításokat kezdi elvégrehajtani ahelyett, hogy a következő utasítást venné. A JZ utasítás ellenőrzi a nullajelzőt (zero flag), és ha be van állítva (ZR) egy ugrást hajt végre. A JZ ellentéte a JNZ (Jump if not zero - ugrás, ha nem nulla). Nézzünk egy egyszerű példát, ami a JNZ utasítást tartalmazza, és vonjunk ki egyet egy számból, amíg az eredmény nulla nem lesz:

```
100 SUB AL,01
102 JNZ 0100
104 INT 20
```

Az AL regiszterbe kis számot tegyünk, hogy csak néhányszor kelljen a cikluson végigmennünk, majd kövessük nyomon a programot lépésenként, hogy lássuk a feltételes ugrások működését. Azért raktuk az INT 20h utasítást a végére, hogy véletlenül G-t írva se fussunk túl a programunk végén.

Talán feltűnt, hogy ha a SUB utasítást használjuk két szám összehasonlítására, fellép egy potenciálisan kellemetlen mellékhatás: az első szám megváltozik. Egy másik utasítás, a CMP (Compare - összehasonlít) lehetővé teszi, hogy

végrehajtsuk a kivonást, anélkül, hogy az eredményt bárhol tárolnánk és az első szám megváltozna. Az eredmény csak arra jó, hogy beállítsa a jelzőket, így használhatjuk a feltételes ugrások valamelyikét az összehasonlítás után. Hogy lássuk, mi történik, állítsuk be mind az AX, mind a BX értékét azonosra, majd kövessük nyomon a következő utasítást:

CMP AX,BX

A nullajelző be lesz állítva, de a két regiszter tartalma változatlan marad.

[Állapotjelző bitek gyakorló](#)

[Top](#)

[15] Egy hex számjegy kiíratása

Használjuk a CMP-t egy hex számjegy kiíratására. Feltételes ugrásokat fogunk használni a programunk menetének megváltoztatására. Ezek az új utasítások a jelzőket fogják használni olyan állapotok ellenőrzésére, mint pl. a kisebb mint, nagyobb mint és hasonló. Nem kell törődnünk azzal, hogy mely jelzők vannak beállítva, ha az első szám kisebb, mint a második; az utasítás tudja, mely jelzőket kell figyelembe venni. Kezdjük azzal, hogy egy kicsi (0 és Fh közötti) számot rakunk a BL regiszterbe. Mivel bármely 0 és Fh közötti szám egyetlen hex számjeggyel egyenértékű, választásunkat átalakíthatjuk egyetlen ASCII karakterre, amelyet kiíratunk. A 0-tól 9-ig terjedő ASCII karakterek értékei 30h-tól 39h-ig terjednek; az A-tól F-ig terjedő karakterek értékei azonban 41h-től 46h-ig. E két csoport ASCII karaktert hét másik választja el egymástól. Így az ASCII konverzió különböző lesz a két számcsoport esetén.

Használjuk a CMP utasítást, illetve egy feltételes ugrás utasítást, amelynek neve JL (Jump if less then - ugrás, ha kisebb mint). A program, amely egy hex számjegyet ír ki a BL regiszterből, a következő:

```
100  MOV  AH,02
102  MOV  DL,BL
104  ADD  DL,30
107  CMP  DL,3A
10A  JL   010F
10C  ADD  DL,07
10F  INT  21
111  INT  20
```

A CMP utasítás kivonja a két számot (DL-3Ah), de nem változtatja a DL-t. Így ha DL kisebb, mint 3Ah, a JL 10F utasítás az INT 21h utasításhoz ugrik az 10Fh címnél. Próbáljuk ki a program működését néhány értékkel. Nyomkövetés közben ne felejtsük el a G parancsban a töréspontot megadni vagy a P parancsot használni, amikor az INT utasításokat futtatjuk! Ne felejtsük el minden futtatás előtt az IP-t visszaállítani 100h-ra!

[Egy hex számjegy kiíratása gyakorló](#)

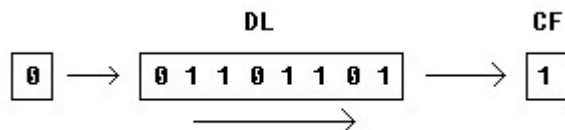
[Top](#)

[16] Léptetés utasítások

Előző programunk működik bármely egyjegyű hex számra. A továbbiakban egy kétjegyű hex számot szeretnénk kiírni. Ehhez el kell különítenünk az egyes számjegyeket (négy bit) a kétjegyű számon belül. Néhány újabb utasításra lesz szükségünk. Kezdetnek emlékezzünk vissza, hogy a RCL utasítás egy byte-ot vagy szót forgat balra az átviteljelzőn keresztül. Programunkban a RCL BL, 1 utasítást használtuk, amivel megmondtuk a processzornak, hogy 1 bittel forgassa a BL-t. Többet is forgathatunk, mint 1 bit, ha akarunk, de nem írhatjuk egyszerűen azt az utasítást, hogy RCL BL, 2. (Ez az utasítás működik a 80286-os mikroprocesszoroktól kezdődően, de ehhez az assembler-nek jelezni kell a processzor típusát. Debug nem ismeri fel.) Az egy bitnél nagyobb forgatásokhoz egy forgatásszámlálót kell a CL regiszterbe raknunk.

A CL regiszter szerepe nagyon hasonló ahhoz, ahogyan a CX regisztert használja a LOOP utasítás annak meghatározására, hogy hányszor hajtsa végre aciklust. A 8088-as a CL-t használja a CX regiszter helyett azért, hogy tudja hányszor forgasson egy byte-ot vagy szót. Mivel nincs értelme 16-nál többször forgatni, a 8 bites CL regiszter bőven elég arra, hogy tartalmazza a lehető legnagyobb forgatásszámot. Hogyan jön mindez a kétjegyű hex számok kiírásához? Tervünk az, hogy a DL-t négy bittel jobbra forgatjuk. Ehhez egy kicsit másféle léptető utasítást használunk, amelynek neve SHR (Shift right - léptetés jobbra). Az SHR használatával a számunk felső négy bitjét elmozdíthatjuk a jobb oldali bit-négyesbe.

Arra is szükségünk van, hogy a DL felső négy bite nulla legyen, hogy az egész regiszter egyenlő legyen azzal a bitnégyessel, amelyet a jobb oldali bitnégyesbe helyeztünk. Ha beírjuk, hogy SHR DL, 1, ez az utasításunk a DL bitjeit egy bittel jobbra lépteti, és ugyanakkor a 0 pozícióban levő bitet az átviteljelzőbe teszi, míg 0-t léptet be a 7. bitbe (a legfelső, vagy baloldali bit DL-ben).



Ha ezt még háromszor megteesszük, elértük amit akartunk: a felső négy bit átkerül jobbra, az alsó négy bitbe, a felső négy bit pedig nullát tartalmaz. Ez a léptetés elvégezhető egyetlen hex utasítással, a CL regisztert használva az eltolások számlálására. A CL regiszter értékét a SHR DL, CL utasítás előtt négyre állítva biztosíthatjuk, hogy DL értéke egyenlő lesz a felső hex számjeggyel. Lássuk hogyan működik mindez! Tegyük 4-et a CL-be és 5Dh-t DL-be, majd írjuk be és kövessük nyomon a SHR DL, CL utasítást. A DL értéke most 05h kell, hogy legyen, ami az első számjegye az 5Dh számnak, és ezt a számjegyet az előzőekben használtához hasonló program segítségével ki is írhatjuk:

```
100  MOV AH,02
102  MOV DL,BL
104  MOV CL,04
106  SHR DL,CL
108  ADD DL,30
10B  CMP DL,3A
```

10E JL 0113

111 ADD DL,07

113 INT 21

115 INT 20

[Léptetés utasítások gyakorló](#)

[Top](#)

[17] Logika és az ÉS művelet

Most, hogy ki tudjuk írni az első egy hex szám két számjegye közül, lássuk, hogyan különíthetjük el és írathatjuk ki a második számjegyet! Itt megtanuljuk, hogyan nullázhatjuk le az eredeti (nem a léptetett) számunk felső négy bitjét, hogy a DL az alsó négy bittel legyen egyenlő. Egyszerűen állítsuk a felső négy bitet nullára egy AND (ÉS) nevű utasítással! Az AND az egyike a logikai utasításoknak, melyek a formális logikában gyökereznek. Lássuk, hogyan működik az AND! A formális logikában állíthatjuk, hogy "A igaz, ha B és C külön-külön igaz". De ha B vagy C hamis, akkor A szintén hamis. Ha vesszük ezt az állítást, és behelyettesítjük az igaz értéket eggyel, a hamisat pedig nullával, akkor az A, B és C különféle kombinációiból úgynevezett igazságtáblázatot készíthetünk. Hasonló képpen értelmezhetjük a VAGY műveletet is: "A igaz, ha B igaz vagy C igaz". Itt látható az igazságtáblázat két bit ÉS és VAGY műveletére:

ÉS	H	I	ÉS	0	1	VAGY	H	I	VAGY	0	1
H	H	H	0	0	0	H	H	I	0	0	1
I	H	I	1	0	1	I	I	I	1	1	1

A bal oldali oszlopban és a felső sorban találhatók a két bit értékei. Az ÉS illetve VAGY művelet eredményei a táblázatban vannak. Így látható, hogy 0 ÉS 1 eredménye 0.

Az AND utasítás byte-okon és szavakon is működik, végrehajtva a logikai ÉS műveletet, mindegyik byte vagy szó azonos pozícióban levő bitjein. Például az AND BL, CL utasítás egymás után végrehajtja az ÉS műveletet BL és CL 0. bitjein, majd az 1. bitjein, 2. bitjein és így tovább, majd az eredményt a BL-ben tárolja. Tegyük ezt világossá egy példával:

1011 0101	0111 1011	7Bh
AND 0111 0100	AND 0000 1111	AND 0Fh
-----	-----	-----
0011 0100	0000 1011	0Bh

Ha bármely számhoz ÉS műveletkor a 0Fh számot tesszük, a felső négy bitet nullára állíthatjuk. Építsük be ezt a logikát egy rövid programba, amely elkülöníti a BL-ben levő szám alacsonyabb hex számjegyet a 0Fh és az ÉS műveletet használva, majd egy karakter formájában kiírja az eredményt. A program legtöbb részletét már láttuk, amikor kiírtuk a felső hex számjegyet. Az egyetlen új részlet az AND utasítás:

```
MOV AH,02
```

```
MOV DL,BL
```

```
AND DL,0F
```

```
ADD DL,30
```

CMP DL,3A

JL 0112

ADD DL,07

INT 21

INT 20

Próbáljunk ki néhány hex számot BL-ben, mielőtt a részleteket egyberaknánk mindkét számjegy kiíratására.

[Logika és az ÉS művelet gyakorló](#)

[Top](#)

[18] Két hex számjegy kiírása

Most már valójában alig van változtatni való, amikor a részeket összerakjuk. Csak a második JL utasítás címét kell megváltoztatnunk, amit a második hex számjegy kiírásakor használtunk. Íme a teljes program:

```
MOV AH,02
MOV DL,BL
MOV CL,04
SHR DL,CL
ADD DL,30
CMP DL,3A
JL 0113
ADD DL,07
INT 21
MOV DL,BL
AND DL,0F
ADD DL,30
CMP DL,3A
JL 0125
ADD DL,07
INT 21
INT 20
```

Ha már begépeltük ezt a programot, U 100 begépelésével, majd egy újabb U-val kilistázható az egész visszafordított program. BX-be megfelelő értéket töltve a G parancs segítségével kiírathatjuk a BL-ben levő két hex számjegyet.

[Két hex számjegy kiírása gyakorló](#)

[Top](#)

[19] Egy karakter beolvasása

A DOS INT 21h funkcióhívás, amelyet eddig használtunk, tartalmaz egy bevitel (input) funkciót, az 1-es számút, amely beolvas egy karaktert a billentyűzetről. Amikor az INT 21h megszakításról tanultunk, láttuk, hogy a funkciószámot az INT 21h meghívása előtt az AH regiszterbe kell helyezni. Próbáljuk ki az INT 21h 1-es funkcióját. Írjuk be az INT 21h utasítást az 100h címre, majd rakjunk 1-et az AH regiszterbe. Egy G 102h vagy P parancs segítségével futtassuk ezt az utasítást! Semmi sem történt? Úgy tűnik, nem - csak egy villogó kurzor látható. Valójában a DOS most megállt, és várja, hogy lenyomjunk egy billentyűt. A lenyomott karakter ASCII kódját a DOS az AL regiszterbe helyezi. Próbáljuk ki! Ismétléskor ne felejtsük el az IP-t visszaállítani 100h-ra!

De mi történik, ha egy nyomtatható karakter helyett egy funkcionális billentyűt, például az F1-et nyomjuk meg? Próbáljuk meg! (Ellenőrizzük, hogy az IP 100h legyen!) Az F1 billentyű lenyomásával a DOS egy 0-t helyez az AL-be, és egy pontosvessző (;) jelenik meg a Debug kötőjel promptja után.

Az F1 egyike azoknak a különleges billentyűknek, amelyek kiterjesztett kódúak, amit a DOS másként kezel, mint a normális ASCII karaktereket képviselő billentyűket. E különleges billentyűk mindegyikénél a DOS két karaktert küld, rögtön egymás után. Az első visszaküldött karakter mindig nulla (hexa) jelezve, hogy az ezt követő karakter a különleges billentyű letapogató kódja (scan code). Hogy mind a két karaktert beolvashassuk, az INT 21h funkciót kétszer kell végrehajtani. De példánkban csak az első karaktert, a nullát olvastuk be, a letapogató kódot a DOS-ban hagytuk. Amikor a Debug befejezte a G 102h vagy P parancs végrehajtását, egy új parancs karaktert kezdett beolvasni, és ez az első beolvasott karakter az F1 benthagyott letapogató kódja volt, pontosabban 59 (3Bh), ami a pontosvessző ASCII kódja. Ezt láthatjuk a Debug promptja után.

[Egy karakter beolvasása gyakorló](#)

[Top](#)

[20] Hex szám beolvasása

Fordítsuk meg az előzőekben használt konverzió menetét. Kiíratás helyett olvassunk be egy kétjegyű hex számot. Ahhoz, hogy egy karaktert (0-9, A-F) egy byte-tá alakíthassunk, ki kell vonnunk vagy 30h-t (0-tól 9-ig), vagy 37h-t (A-tól F-ig) a karakter ASCII kódjából. Ezt megtehetjük a következő utasításokkal:

```
MOV AH,01
```

```
INT 21
```

```
SUB AL,30
```

```
CMP AL,09
```

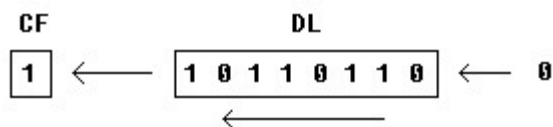
```
JLE 010C
```

```
SUB AL,07
```

```
INT 20
```

Ezen utasítások többsége már ismerős, de van egy új, a JLE (Jump if Less than or Equal - ugrás, ha kisebb vagy egyenlő). A programunkban ez az utasítás akkor ugrik, ha AL kisebb vagy egyenlő 9h-val.

Két hex számjegyet beolvasni nem sokkal bonyolultabb, mint egyet, de sokkal több utasításra van szükség. Az első számjegy beolvasásával kezdjük, majd a hex értéket a DL regiszterbe helyezzük és megszorozzuk 16-tal. A szorzás elvégzéséhez a DL regisztert balra léptetjük négy bittel, egy hex nullát (négy nulla bit) helyezve az imént beolvasott számjegytől jobbra. A SHL DL, CL utasítás CL négy értéke mellett megteszi ezt, négy nulla jobbról való beszúrásával. Valóban a SHL (SHift Left - léptetés balra) utasítás aritmetikai léptetésként is ismert, mert hatása ugyanaz, mintha szoroznánk kettővel, négygel, nyolccal, stb., a CL-ben levő számtól függően.



Miután az első számjegyet eltoltuk a helyére, a második számot hozzáadjuk a DL-ben levő számhoz (amely az első számjegy * 16). A teljes program:

```
MOV AH,01
```

```
INT 21
```

```
MOV DL,AL
```

```
SUB DL,30
```

CMP DL,09

JLE 0111

SUB DL,07

MOV CL,04

SHL DL,CL

INT 21

SUB AL,30

CMP AL,09

JLE 011F

SUB AL,07

ADD DL,AL

INT 20

Most, hogy van egy működő programunk, érdemes ellenőrizni a határeseteket, és néhány más esetet, hogy megfelelően működik-e. (Határesetek: 00, 09, 0Ah, 0Fh, 90h, A0h, F0h) Egy töréspontot használva futtassuk a programot anélkül, hogy végrehajtanánk az INT 20 utasítást, különben a G parancs a program normális befejezése esetén visszaállítja a regisztereket az indulási értékre, és nem láthatjuk az eredményt. Ne felejtsük el minden futtatás előtt visszaállítani az IP értékét 100h-ra. Próbáljunk begépelni néhány karaktert, mint a k vagy a kis d, amelyek nem hex számjegyek. Látható, hogy ez a program csak akkor működik helyesen, ha a bevitt karakterek érvényes hex számjegyek (számjegyek 0-tól 9-ig, nagybetűk A-tól Z-ig). Ezt a hibát kijavítjuk a következő fejezetekben, addig egy kicsit megalkuvók leszünk, és a hibás eseteket nem vesszük figyelembe: a helyes karaktereket kell begépelnünk ahhoz, hogy a programunk megfelelően működjön.

[Hex szám beolvasása gyakorló](#)

[Top](#)

[21] Eljárások

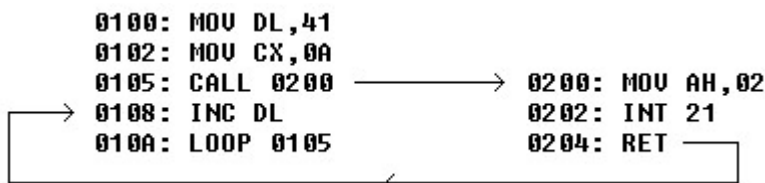
Egy eljárás olyan utasítások sorozata, amelyeket egy program több különböző pontjáról végrehajthatunk, ahelyett, hogy az utasítások sorozatát minden szükséges helyen meg kellene ismételnünk. Egy eljárást a CALL (hív) utasítással hívhatunk meg. A visszatérés egy eljárásból a RET utasítással történik. Az eljárást a 200h címre írjuk, hogy helyet hagyjunk az 100h címen kezdődő főprogramnak.

A következő program A-tól J-ig kiíratja a nagybetűket, egy rövid eljárást használva a kiíratásra:

A főprogram 100h címtől	Az eljárás a 200h címtől
MOV DL,41	MOV AH,02
MOV CX,0A	INT 21
CALL 200	RET
LOOP 0105	
INC DL	
INT 20	

Az első utasítás 41h-t (az A betű ASCII kódját) tesz a DL regiszterbe, mert az INT 21h utasítás a DL-ben levő ASCII kódhoz tartozó karaktert írja ki. Maga az INT 21h utasítás egy kicsit távolabb, a 200h-nál levő eljárásban van INC DL, az új utasítás növeli a DL regiszter tartalmát eggyel, így a DL-t az ábécé következő karakterére állítja.

A G paranccsal hajtsuk végre a programot, hogy lássuk az eredményt, majd lépésenként kövessük nyomon, hogy lássuk a működését is (ne felejtsük az INT utasítást P paranccsal hajtatni végre).



[Eljárások gyakorló](#)

[Top](#)

[22] Verem és a visszatérési címek

A programunkban levő CALL utasításnak valahol tárolnia kell a visszatérési címet, hogy a processzor tudja, hol folytassa az utasítások végrehajtását, amikor a RET utasítást találja. Magára a tárolásra van a memóriában egy elkülönített hely, amit veremnek (stack) nevezünk. A veremben levő adatok nyomonkövetéséhez van két regiszter, amit az R képernyőn láthatunk: a SP (Stack Pointer - veremmutató) regiszter, amely a verem tetejére mutat, és a SS (Stack Segment - veremsegment) regiszter, ami a szegmensszámot tartalmazza.

Működés közben a verem olyan, mint egy önkiszolgáló vendéglőben a tálcátartó ahol a felül levő tálca eltakarja az alatta levőket. A tartóba került utolsó tálcát vesszük le először, így a verem másik elnevezése LIFO (Last In FirstOut - utoljára be, elsőként ki). Ez a sorrend, a LIFO pontosan az, amire a visszatérési címek előhívásához szükségünk van, amikor egymásba ágyazott CALL utasításokat végzünk, mint például a következő:

Program	Cím	Verem	Cím	Verem
0100: CALL 200	FFF8:		SP → FFF8:	0303
...				
0200: CALL 300	SP → FFFA:	0203	FFFA:	0203
0203: RET				
...	FFFC:	0103	FFFC:	0103
0300: CALL 400				
0303: RET	FFFE:	0000	FFFE:	0000
...				
0400: RET				

Itt a 100h címen levő utasítás hívja a 200h-nál levőt, ami a 300h-nál levőt hívja, ez pedig a 400h-nál levőt, ahol végre egy visszatérés (RET) utasítást találunk. Ez a RET az előző CALL utasításra, azaz 300h-t követő utasításra tér vissza, így a processzor a 303h címen levő utasítással folytatja. Ott újra egy RET utasítást talál, ami a következő régebbi címet (203h) húzza elő a veremből. Így a processzor az utasítások végrehajtását a 203h címnél folytatja és így tovább. Minden egyes RET a verem tetejéről szedi ki a visszatérési címet, így a RET utasítások fordítva járnak végig azt az utat, amit a megelőző CALL utasítások jártak az ellenkező irányban. Próbáljunk bevinni egy olyan programot, mint az előbbi! Használjunk többszörös hívásokat, majd kövessük nyomon a programot, hogy lássuk a hívások és visszatérések működését! A verem tetején levő elemet a Debugban a regiszterkiírás jobb alsó sarkában találjuk meg.

[Verem és visszatérési címek gyakorló](#)

[Top](#)

[23] A PUSH és POP utasítások

A verem alkalmas arra, hogy ott adatszavakat tároljunk egy ideig, feltéve, hogy ügyelünk a verem eredeti állapotának visszaállítására egy RET utasítás előtt. Láttuk, hogy a CALL utasítás lenyomja (push) a verembe a visszatérési címet (egy szó), míg a RET utasítás kihúzza (pop) ezt a szót a verem tetejéről, betölti az IP regiszterbe és szabaddá teszi az alatta lévő szót. Valami egészen hasonlót lehet a PUSH és POP utasításokkal elérni, amelyek lehetővé teszik szavak lenyomását és kihúzását. Vajon mikor célszerű ezt megtenni? Gyakran lesz szükség arra, hogy a regiszterek tartalmát egy eljárás elején tároljuk, és a végén, közvetlenül a RET utasítás előtt visszaállítsuk. Így szabadon használhatjuk ezeket a regisztereket az eljárás során, mivel eredeti értéküket az eljárás végén visszaállítjuk.

A programok többszintű eljárásokból épülnek fel, és minden szint a következő szintnél lejjebb levő eljárásokat hívja. Tárolva a regiszterek tartalmát egy eljárás elején és visszaállítva őket a végén, kiküszöbölhetők a nemkívánt kölcsönhatások a különböző szinten levő eljárások között, ez pedig sokkal könnyebbé teszi a programozást. A következő példa tárolja és visszaállítja a CX és DX regisztereket:

```
200:    PUSH CX
```

```
        PUSH DX
```

```
        MOV DL,61
```

```
        MOV CX,0A
```

```
        CALL 0300
```

```
        INC DL
```

```
        LOOP 205
```

```
        POP DX
```

```
        POP CX
```

```
        RET
```

Látható, hogy a POP utasítások a PUSH utasításokhoz képest fordítva vannak, mivel a POP a legutoljára berakott szót veszi ki a veremből, és a DX régi értéke a CX régi értéke felett van. A CX és DX tárolása és visszaállítása lehetővé teszi, hogy a regisztereket megváltoztassuk a 200h címnél kezdődő eljárásban, anélkül, hogy megváltoztatnánk azokat az értékeket, amelyeket az ezt hívó eljárás használt. Miután tároltuk a CX és DX értékeit, ezeket a regisztereket helyi változók tárolására használhatjuk - olyan változókra, amelyeket ebben az eljárásban használunk anélkül, hogy hatással lennének a hívó program által használt értékekre.

Ilyen helyi változókat fogunk használni programozási feladataink egyszerűsítésére. Addig, amíg vigyázunk arra, hogy az eredeti értékeket visszaállítsuk, nem kell attól félnünk, hogy eljárásaink megváltoztatják a hívó program által használt regisztereket.

[24] Hex számok beolvasása II

Egy olyan eljárást akarunk létrehozni, amely addig olvassa be a karaktereket amíg egy olyat nem kap, amit egy 0 és Fh közötti hex számmá tud alakítani. Nem akarunk kiíratni érvénytelen karaktereket, így megszűrjük a bemenő adatokat egy új INT 21h funkcióval, a 8. számúval, ami beolvas egy karaktert, de nem küldi tovább a képernyőre. Így csak azokat a karaktereket fogjuk visszhangozni (echo) - azaz kiírni a képernyőre - amelyek érvényesek. Rakjunk 8h-t az AH regiszterbe, majd futtassuk a következő utasítást, rögtön a G 102 begépelése után egy A-t gépelve (új sorba): INT 21. Az A betű ASCII kódja (41h) az AL regiszterben van, de az A nem jelent meg a képernyőn.

Ezzel a funkcióval programunk beolvashat karaktereket a képernyőre visszhangozásuk nélkül, addig, amíg egy valós hex számjegyet (0-tól 9-ig vagy A-tólF-ig) nem kap, amit majd visszhangoz. A következő eljárás beolvas egy karaktert AL-be és ellenőrzi, hogy érvényes-e a CMP és feltételes utasításokkal. Ha az előbb beolvasott karakter érvénytelen, a feltételes ugrások visszaküldik a processzort a 203-as címre, aholaz INT 21h egy újabb karaktert olvas be (JA, Jump Above - ugrás, ha nagyobb; JB, Jump Below - ugrás, ha kisebb, mindkettő előjel nélkül kezeli a számokat, ellentétben a korábban már használt JL utasítással, amely előjeles számokat kezel).

200: PUSH DX

 MOV AH,08

 INT 21

 CMP AL,30

 JB 203

 CMP AL,46

 JA 203

 CMP AL,39

 JA 21B

 MOV AH,02

 MOV DL,AL

 INT 21

 SUB AL,30

 POP DX

 RET

 CMP AL,41

 JB 203

 MOV AH,02

```
MOV DL,AL
```

```
INT 21
```

```
SUB AL,37
```

```
POP DX
```

```
RET
```

Megjegyzendő, hogy két RET utasításunk van az eljárásban, de lehetett volna több is vagy akár csak egy. Egy nagyon egyszerű programmal ellenőrizhetjük az eljárást:

```
100: CALL 200
```

```
INT 20
```

Ahogy korábban is tettük, vagy a G 103 parancsot használjuk, vagy a P parancsot. A CALL 200 utasítást akarjuk végrehajtani az INT 20h utasítás nélkül, hogy láthassuk a regiszterek tartalmát, mielőtt a program befejeződik és a regiszterek tartalma törlődik.

Most, hogy megvan az eljárásunk, egy kétjegyű hex számot beolvasó, hibakezeléssel ellátott program már elég egyszerű:

```
100: CALL 200
```

```
MOV DL,AL
```

```
MOV CL,04
```

```
SHL DL,CL
```

```
CALL 200
```

```
ADD DL,AL
```

```
MOV AH,02
```

```
INT 21
```

```
INT 20
```

Ez a program a DOS-ból futtatható (ha kimentettük az N és W parancsok segítségével), mivel egy kétjegyű hex számot olvas be, majd kiírja a begépelt szám ASCII karakter megfelelőjét.

Magától az eljárástól eltekintve főprogramunk lényegesen egyszerűbb lett, mint az előzőekben bemutatott változat, és nem kell ismételní a karaktereket beolvasó utasításokat. Hozzá tettünk hibakezelést, ami - bár eljárásunkat egy kicsit bonyolította - biztosította, hogy a program csak érvényes karaktereket fogadjon el bemenetként. Azt is láthatjuk, mi értelme volt a DX regiszter tárolásának az eljárás elején. A főprogram a DL-ben tárolja a hex számot, ezért nem akarjuk, hogy a 200h-nál levő eljárásunk megváltoztassa DL értékét. Másrészt viszont az eljárásnak szüksége van a DL regiszterre a karakterek képernyőre visszahangozásához. A PUSH DX használatával az eljárás elején és a POP DX utasítással az eljárás végén ezek a problémák kiküszöbölhetők.

[Hex számok beolvasása II gyakorló](#)

[Top](#)

[Hexaritmetika gyakorló]

-H 3 2

-H D C

-H FFFF 1

-H 2 3

-H 9 1

-H CD1 92A

-H FFFF A

-H 9 3

-H BCD8 509

-H F451 CB03

-H 9 7

-H 4567 23AF

-H BCD8 FAE9

[Top](#)

[Negatív számok gyakorló]

-H 2 3

-H 0 1

-H 1 FFF

-H 0 3C8

-H 5 FFFF

-H 3C8 FC38

-H FF FF

-H 0 C4

[Top](#)

[Regiszterek gyakorló]

Jelenítsd meg a regisztereket, majd változtasd meg tartalmukat és ellen ôrizd az eredményt. Például:

-R

-R AX

: 3C5

-R BX

:C4

-R CX

:55F

-R

[Top](#)

[Memória gyakorló]

-E 100

xxxx:0100 xx.01

-E 101

xxxx:0101 yy.D8

-R

A regiszterek tartalma alatti sorban láthatjuk a beírt memóriarészt.

[Top](#)

[Összeadás gyakorló]

-R AX

:3D4

-R BX

:123

-E 100

.01

-E 101

.D8

-R

-T

-R IP

:100

-R

-T

Néhány összeadás kódja:

ADD AX, BX 01D8

ADD AX, CX 01C8

ADD AX, DX 01D0

ADD BX, CX 01CB

ADD BX, DX 01D3

ADD CX, DX 01D1

[Top](#)

[A négy alapművelet gyakorló]

-r ax

:1234

-r bx

:345

-e 100

xxxx:0100 yy.29 zz.d8

-r

-t

SUB AX,BX 29 D8

SUB AX,CX 29 C8

SUB AH,BL 28 DC

ADD AH,AL 00 C4

MUL BX F7 E3

MUL BI F6 E3

DIV BX F7 F3

DIV BL F6 F3

[Top](#)

[Megszakítások gyakorló]

-R AX

:200

-R DX

:41

-E 100

... CD ...21

-R

-G 102

-E 100

... CD ...20

-R IP

:100

-R

-G

[Top](#)

[Programok beírása gyakorló]

-E 100

CD 21 CD 20

-U 100

-A 100

XXXX:0100 INT 21

XXXX:0102 INT 20

XXXX:0104

-R AX

:1234

-R DX

:ABCD

MOV AH,DL

MOV BX,AX

-A 100

MOV AH,02

MOV DL,2A

INT 21

INT 20

-G

-N CS.COM

CX <- 8

BX <- 0

-W

[Top](#)

[Karakterlánc kiírása gyakorló]

-E 200

48 65 6C 6C 6F 2C

20 44 4F 53 20 68

65 72 65 2E 24

-A 100

MOV AH,09

MOV DX,0200

INT 21

INT 20

-U 100

-G

-D 200

-N IRSOR.COM

-R CX

:211

-R BX

:0000

-W

[Top](#)

[Átviteljező bit (carry flag) gyakorló]

-A 100

ADD AX,BX

-R AX

:FFFF

-R BX

:1

-R

-T

-A 102

ADD AX,1

-R

-T

-A 105

RCL BL,1

RCL BL,1

... 9-szer

-R BX

:B7

-R

-T 9-szer

[Top](#)

[Ciklus képzése gyakorló]

-A 100

MOV BX,6D

MOV CX,0004

RCL BX,1

LOOP 106

INT 20

-U

-R

-T amíg befejeződik, INT 20h nélkül!

-R IP

:100

-R BX

:0

-G 10A

[Top](#)

[Bináris szám kiíratása gyakorló]

-A 100

MOV AH,02

MOV CX,0008

MOV DL,00

RCL BL,1

ADC DL,30

INT 21

LOOP 0105

INT 20

-U

-R BX

:00D6

-R

-G

Egészítsd ki a programot úgy, hogy a szám után írja ki a b betűt, kód 62h.

[Top](#)

[Állapotjelző bitek gyakorló]

-A 100

SUB AX,BX ;AX és BX-be azonos érték

-A 100

SUB AL,01

JNZ 0100

INT 20

-A 100

CMP AX,BX ;AX és BX-be azonos érték

Ne felejtsük el IP-t minden alkalommal visszaállítani! Próbáljuk ki különböző értékekkel a jelzők beállítását!

[Top](#)

[Egy hex számjegy kiíratása gyakorló]

-R BX

:000D

A 100

MOV AH,02

MOV DL,BL

ADD DL,30

CMP DL,3A

JL 010F

ADD DL,07

INT 21

INT 20

-U 100 120

-T és P -k

[Top](#)

[Léptetés utasítások gyakorló]

-R CX

:0004

-R DX

:005D

-A 100

SHR DL,CL

-R

-T

-R BX

:005D

-A 100

MOV AH,02

MOV DL,BL

MOV CL,04

SHR DL,CL

ADD DL,30

CMP DL,3A

JL 0113

ADD DL,07

INT 21

INT 20

[Top](#)

[Logika és az ÉS művelet gyakorló]

-A 100

MOV AH,02

MOV DL,BL

AND DL,0F

ADD DL,30

CMP DL,3A

JL 112

ADD DL,07

INT 21

INT 20

-R BX

:00B5

-U 100

-R

-T ill. P a program végéig

-R IP

:100

-R BX

:006A

-G

stb.

[Top](#)

[Két hex számjegy kiírása gyakorló]

-A 100

MOV AH,02

MOV DL,BL

MOV CL,04

SHR DL,CL

ADD DL,30

CMP DL,3A

JL 0113

ADD DL,07

INT 21

MOV DL,BL

AND DL,0F

ADD DL,30

CMP DL,3A

JL 0125

ADD DL,07

INT 21

INT 20

-U 100

-U

-R BX

:005A

-R

-G

[Top](#)

[Egy karakter beolvasása gyakorló]

-A 100

INT 21

INT 20

-R AX

:0100 ;ismételd más karakterekkel

-R

-P

-R IP

:100

-P ;F1 billentyű

-R IP

:100

-R BX

:FFFF

-A 100

INT 21

MOV BL,AL

INT 21

INT 20

-R AX

:0100

-G 106

[Top](#)

[Hex szám beolvasása gyakorló]

-A 100

MOV AH,01

INT 21

MOV DL,AL

SUB DL,30

CMP DL,09

JLE 0111

SUB DL,07

MOV CL,04

SHL DL,CL

INT 21

SUB AL,30

CMP AL,09

JLE 011F

SUB AL,07

ADD DL,AL

INT 20

-U 100

-R

-G 121

-R IP

:100

-G 121

[Top](#)

[Eljárások gyakorló]

-A 100

MOV DL,41

MOV CX,000A

CALL 0200

INC DL

LOOP 0105

INT 20

-A 200

MOV AH,02

INT 21

RET

-U 100

-U 200

-R

-G

Kövessük a programot utasításonként a T ill. P parancsokkal

[Top](#)

[Verem és visszatérési címek gyakorló]

-A 100

CALL 200

INT 20

-A 200

CALL 300

RET

-A 300

CALL 400

RET

-A 400

RET

-R

-T - vel kövessük a regiszterek és a verem alakulását

[Top](#)

[Hex számok beolvasása II gyakorló]

```
PUSH DX  
  
MOV AH,08  
  
INT 21  
  
CMP AL,30  
  
JB 203  
  
CMP AL,46  
  
JA 203  
  
CMP AL,39  
  
JA 21B  
  
MOV AH,02  
  
MOV DL,AL  
  
INT 21  
  
SUB AL,30  
  
POP DX  
  
RET  
  
CMP AL,41  
  
JB 203  
  
MOV AH,02  
  
MOV DL,AL  
  
INT 21  
  
SUB AL,37  
  
POP DX  
  
RET
```

[Top](#)