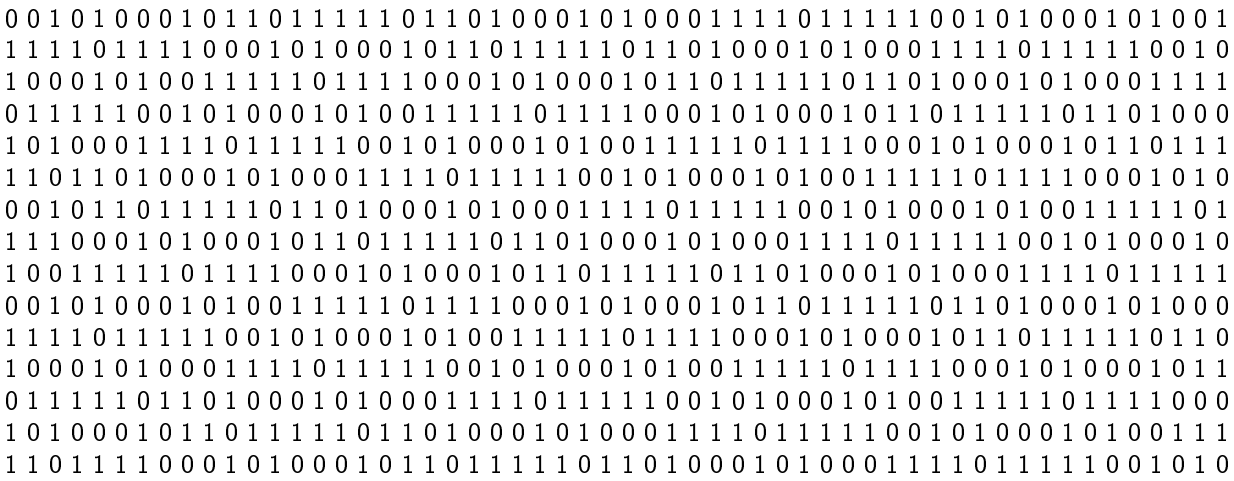
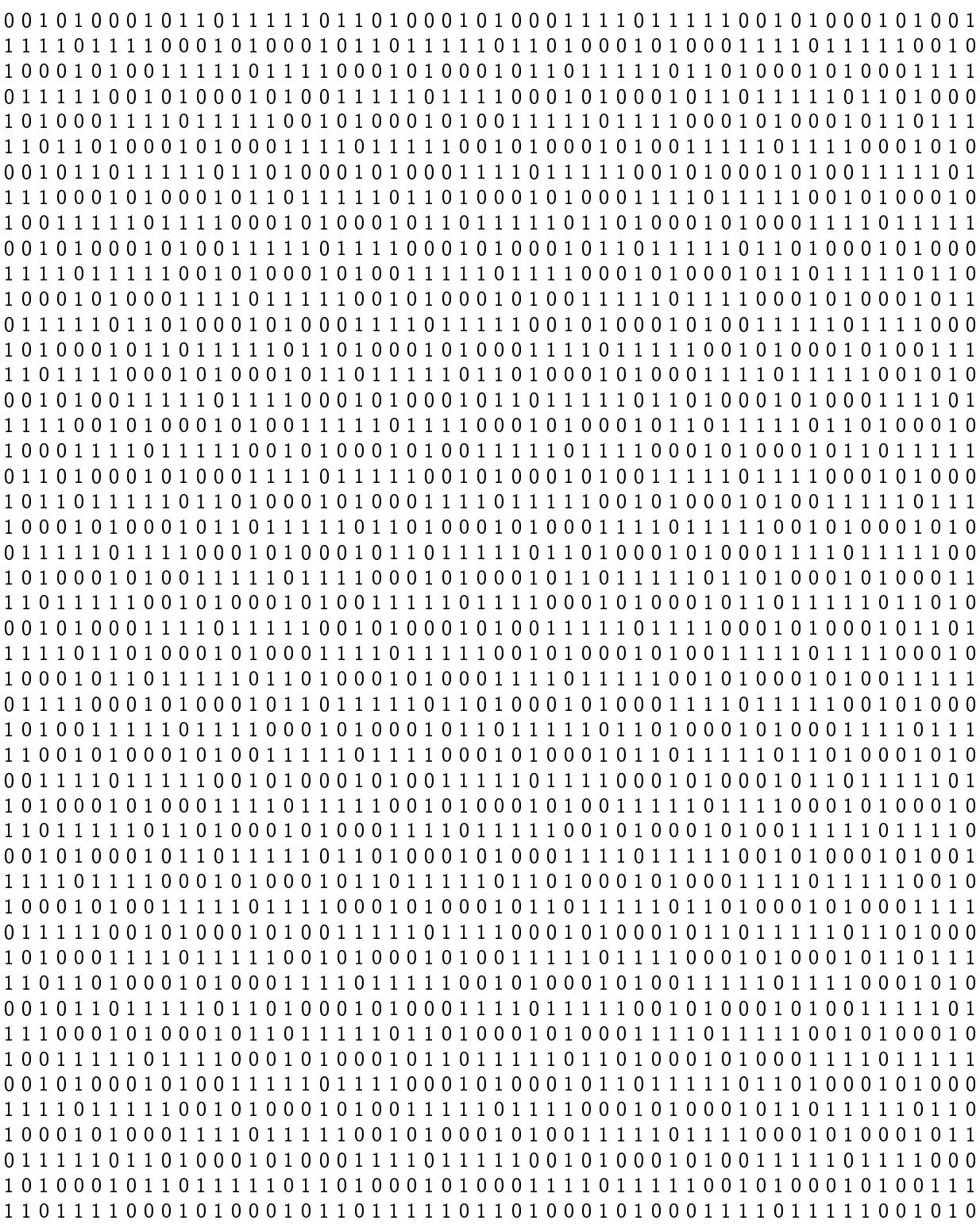


Egri Péter

NASM programozás Windows alatt



Tartalomjegyzék

1

Bevezetés

> „Everyone with more than a casual interest in computers will probably get to know at least one machine language sooner or later. Machine language helps programmers to understand what really goes on inside their computers. And once one machine language has been learned, the characteristics of another are easy to assimilate. Computer science is largely concerned with an understanding of how low-level details make it possible to achieve high-level goals."
>
> Donald E. Knuth

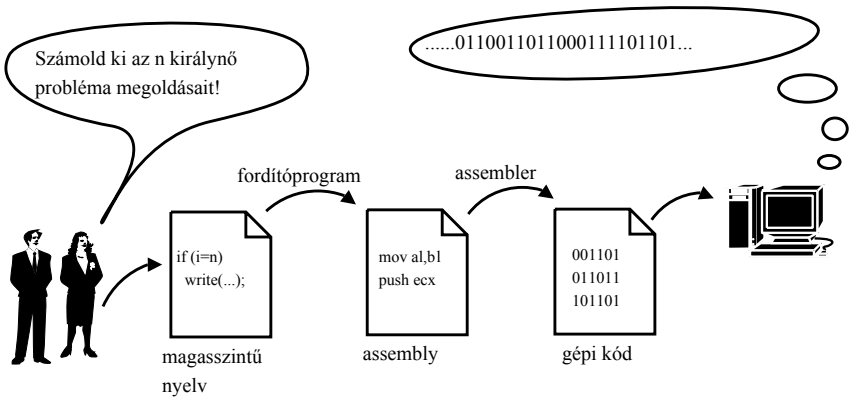
Ez a jegyzet az assembly programozás alapjait igyekszik bemutatni. Nem törekszik teljességre, inkább olyan alapismereteket kíván nyújtani, ami már önállóan könnyen továbbfejleszthető – ehhez az Interneten számtalan dokumentáció és példaprogram is rendelkezésre áll. Mivel az assembly a „gép saját nyelve", ezért némi hardver ismeret is szükséges a megértéséhez, ezek azonban csak nagyon vázlatosan, a szükséges mértékben kerülnek itt ismertetésre.

Nem lesz szó *védett mód*ú (protected mode) programozásról, ez a téma túlmutat a jegyzet célján és egyébként más könyvekben részletesen ismertetésre került. Nem lesz szó továbbá a hagyományos, kompatibilitási okokból még ma is meglévő *valós mód*ról (real mode), DOS megszakításokról sem, ezek már elavult technológiának számítanak. Amiről szó lesz, az a Windows rendszeren való programozás (tulajdonképpen a *flat mode*-ot használjuk, de ezt az operációs rendszer biztosítja számunkra, így ezzel nem kell foglalkoznunk).

Habár a példaprogramok Windows alatt működnek, a jegyzet nem Win API ismertető, viszont (remélhetőleg) elég alapot nyújt ahhoz, hogy az elérhető API dokumentáció már könnyen érthető és felhasználható lesz utána.

1.1. Az assembly nyelv

A hardware *gépi kód*ú programozása nagyon bonyolult, ennek egyszerűsítésére alkották meg az *assembly* nyelvet. A gép által értelmezhető bináris számok helyett egyszerű utasításokat használunk, amiket az *assembler* fordít át 0-k és 1-esek sorozatára. Mivel közvetlenül a gép utasításait, regisztereit, memóriacímzési módjait stb. használjuk, ezért a nyelv gépfüggő: pl. egy PC-re írt assembly program nem fordítható le mondjuk egy VAX-on (míg ez pl. egy `C++` program esetében általában megtehető). Továbbá a hatékony használatához ismerni kell a számítógép felépítését. Az assembly előnyeként szokták emlegetni, hogy sokkal kisebb, gyorsabb kódot lehet benne írni, mint egy magas szintű nyelven. Ez manapság már nem feltétlenül igaz: a fordítóprogramok hatékony kódoptimalizálási algoritmusai általában jobb kódot állítanak elő, mint egy átlagos assembly-programozó.



1. ábra. Az assembly nyelv helyzete

Mégis több ok miatt is érdemes megismerkedni az assemblyvel. Egy általános célú, magas szintű nyelven nem lehet kihasználni a gép minden lehetőségét: ha a hardware egyedi sajátosságait akarjuk használni, ezt csak assemblyben tehetjük meg. Használata közben megismerkedhetünk a gépek felépítésével, működési módjával, belső logikájával, amely a programozók számára is hasznos és tanulságos. Ráadásul a fordítóprogramok kódgeneráló és -optimalizáló részének megértéséhez elengedhetetlen is.

1.2. Szükséges eszközök

Az assembly programozáshoz való alapvető programok és dokumentációk szerencsére szabadon hozzáférhetők az Interneten.

1.2.1. Assembler

A fordító, ami az assembly forrásból előállítja a *tárgykód*ot, ami általában `.obj` vagy `.o` kiterjesztésű fájl. Az ebben a jegyzetben található példák a `NASM` assembler szintaxisát követik (a különböző assemblerek szintaxisa némiképpen eltér egymástól). A `NASM` Windows és Linux rendszerek alatt is elérhető a következő lapon: `http://sourceforge.net/projects/nasm`.

1.2.2. Linker

A linker (vagy összeszerkesztő program) tárgykódból, vagy tárgykódokból futtatható állományt készít. A példaprogramokat Windows alatt az ALINK programmal készítettem el: `http://alink.sourceforge.net`.

1.2.3. Debugger

A debugger segítségével a programot lépésről-lépésre hajthatjuk végre, megfigyelhetjük a regiszterek, flagek, verem és a memória tartalmát is, ami hibák okának kiderítését nagyon megkönnyíti. Assemblyben sokkal egyszerűbb hibás programot írni, mint magasszintű nyelven, ezért a debugger funkciókra még nagyobb szükség van, mint ott. A programok teszteléséhez én a `GoBug` nevű shareware programot használtam: `http://www.goprog.com`.

1.2.4. Win32API help

Mivel a példaprogramok Windows rendszerre készültek, ezért a rendszer szolgáltatásait (API) veszik igénybe. Segítségükkel könnyen tudunk mondjuk egy egyszerű messagebox-ot (ld. hamarosan a „Hello World" példaprogramot) kirakni a képrenyőre. Az operációs rendszer szolgáltatásainak igénybevétele nélkül kénytelenek lennénk jóval bonyolultabb módon az áramköröket, a képernyőmemóriát, stb. piszkálni – ami mellesleg valószínűleg nem is futna, mert a mai operációs rendszerek nem igazán tolerálják a kerülő utakat[1]. Windows alatt a rendszer szolgáltatásainak leírása a Win32API help fájlban található, ami szintén letölthető az Internetről. Az eljárások leírásánál a Quick Info gomb árulja el, hogy az adott eljárás törzse melyik .dll-ben (illetve .lib-ben) található meg. A help fájl letölthető a következő helyről: `http://win32assembly.online.fr/files/win32api.zip`.

1.2.5. Editor

A debuggoláson kívül a program megírásával töltjük a legtöbb időt, ezért nem árt, ha valami kényelmes editorral dolgozunk. Ez tulajdonképpen bármilyen editor lehet, szükség esetén a Notepad is megteszi, de léteznek direkt assembly programozáshoz kifejlesztett környezetek is syntax highlightinggal meg egyéb extrákkal. Ez editor választása teljesen szubjektív, a lényeg, hogy megfelelően konfigurálhassuk és kényelmesen használhassuk.

Például a Quick Editor (`http://www.movsd.com/qed.htm`) egy nagyon egyszerű editor, de lehet benne scripteket írni, plug-in-eket hozzáadni és új menüket definiálni. Például:

```
[&Assembly]
&Compile,cmd /C c:\nasm\nasmw -f obj {f} & c:\nasm\link {n}.obj & del {n}.obj & pause
&Run,cmd /C {n}.exe
&Debug,cmd /C c:\nasm\debug\gobug {n}.exe
&Template,c:\Program Files\QEdit\asm\template.qsc
```

Ez a beállítás létrehozza az „Assembly" menüt, amin belül az első menüpont elvégzi a fordítást: az assemblálást, az összeszerkesztést (sajnos a nagybetűs szöveget nem kezeli helyesen, ezért a szerkesztést a `link.bat` fájlon keresztül hajtom végre), letörli a közbenső tárgykódot és megáll (ha fordítási hiba történt, el tudjam olvasni a hibaüzeneteket). A „Run" futtatja a lefordított programot, a „Debug" pedig a debuggerbe tölti be. A „Template" menüpont egy assembly program vázat (fejléc komment, szegmens definíciók, `..start` címke, `ExitProcess` meghívása) rak bele a programba.

1.2.6. Egyéb dokumentációk

Az Interneten rengeteg (főleg angol nyelvű) leírás és példaprogram elérhető mind a Windows, mind a Linux assembly nyelvű programozásáról. A teljesség igénye nélkül néhány cím:

`http://www.drpaulcarter.com/pcasm`: PC Assembly Language könyv

`http://developer.intel.com/design/Pentium4/documentation.htm#manuals`: Intel architektúra dokumentációk

`http://win32assembly.online.fr/tutorials.html`: Bevezetés a Windows programozásba

`http://www.lookuptables.com`: ASCII és UNICODE karaktertáblák

1.3. A PC-k felépítése az assembly-programozó szemével

Nagyon leegyszerűsítve a számítógépben van egy *processzor* (CPU), ami különböző utasításokat hajt végre adatokon. Az utasítások és az adatok a program futásakor a *memóriá*ban foglalnak helyet.

[1] Ez szintén a védett mód használatának következménye.

1.3.1. Memória

Az információ alapegysége a *bit*, amely két értéket vehet fel, 0-t vagy 1-et. A memória bitekből áll, de a gyakorlatban csak 8 bitből álló *bájt*ként (byte) érhetők el. Minden bájtnak van egy egyedi címe, ezek címek két részből tevődnek össze: egy *szegmenscím*ből, ami a memóriának egy nagyobb csoportját (szegmensét) jelöli ki, valamint egy *offszetcím*ből, ami a szegmensen belül pontosan meghatározza a bájtot. Azt szokták mondani, hogy ha a memória egy könyv lenne, akkor a szegmenscím adná meg az oldalszámot, az offsetcím pedig azon belül pontosan meghatározná a betűt[2].

A bájtokon kívül gyakran lehet találkozni más egységekkel is: a *szó*val (*word*, 16 bájt) és a *duplaszó*val (*dword*, 32 bájt). A memóriában **minden** (programok, számok, karakterek, stb.) egy bájt vagy bájtsorozat formájában van tárolva.

1.3.2. Processzor

Feladata a memóriában található gépi kódú utasítások végrehajtása. A processzorban van néhány *regiszter* is, amik szintén adatokat tárolnak. Jó tulajdonságuk, hogy sokkal gyorsabban el tudja érni őket a processzor, mint a memóriát, viszont nagyon kevés van belőlük. A következő regisztereket használhatjuk[3]:

Szegmensregiszterek: CS (kódszegmens), DS (adatszegmens), SS (veremszegmens), ES, FS, GS (extra szegmensek).

Offsetregiszterek: ESP (veremmutató), EBP (bázismutató), EIP (utasításmutató), ESI (forrásindex), EDI (célindex).

Állapotjelző regiszter: EFLAGS.

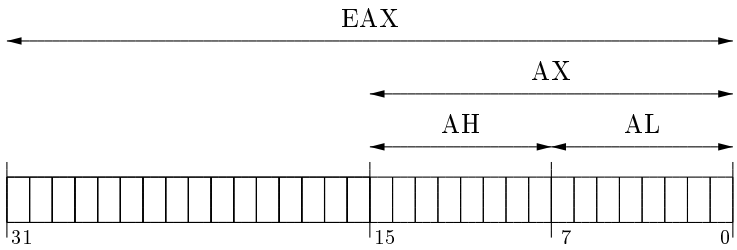
Általános célú regiszterek: EAX, EBX, ECX, EDX.

Néhány fontos tudnivaló a regiszrekről:

- A szegmensregiszterek 16 bitesek, az összes többi regiszter 32 bites.
- A CS, DS, SS, ESP, EIP regiszterek speciális feladatokat látnak el, értéküket csak kivételes esetekben módosítjuk közvetlenül.
- Az EFLAGS regiszterre nem lehet közvetlenül hivatkozni, csak az egyes bitjei érhetők el speciális utasításokkal.
- Az általános célú regiszterek egyes részeire külön nevekkel hivatkozhatunk: az AX, BX, CX, DX jelöli az alsó 16 bitjüket. Ezeknek is külön hivatkozhatunk az alsó és felső bájtjaira: AH, AL, BH, BL, CH, CL, DH, DL (ld. 2. ábra). Fontos megjegyezni, hogy ezek nem külön regiszterek, hanem a 32 bites regiszter részei – így ha pl. az AH-t módosítjuk, az az AX és EAX módosulását is jelenti.
- Az ECX regiszter (számláló) a ciklusoknál különleges szerepet fog kapni.

[2] Ez a hasonlat a védett mód használatánál kissé sántít, hiszen a szegmensek mérete különböző lehet és át is fedhetik egymást.

[3] A felsoroltakon kívül számos más regiszter is létezik amikkel nem foglalkozunk, pl. MMX regiszterek, védett módhoz szükséges regiszterek, stb.



2. ábra. Az EAX regiszter részei

2

Számábrázolás

A mindennapi életben használt decimális (vagy 10-es) számrendszert az indiai brahma papok találták fel és az arabok közvetítésével jutott el a X. században Európába. Ebben a rendszerben 10 számjegy[4] van, amiből az összes számot ki tudjuk fejezni, mégpedig úgy, hogy a számjegyeknek nem csak *alaki értékük*, hanem *helyiértékük* is van.

Például a 64027 számot a következő módon értelmezzük:

$$64027 = 6 \cdot 10^4 + 4 \cdot 10^3 + 0 \cdot 10^2 + 2 \cdot 10^1 + 7 \cdot 10^0$$

Egy decimális szám számjegyeit a következő eljárással kaphatjuk meg:

$$\begin{array}{rl} 64027 \text{ div } 10 = 6402 & 64027 \text{ mod } 10 = 7 \\ 6402 \text{ div } 10 = 640 & 6402 \text{ mod } 10 = 2 \\ 640 \text{ div } 10 = 64 & 640 \text{ mod } 10 = 0 \\ 64 \text{ div } 10 = 6 & 64 \text{ mod } 10 = 4 \\ 6 \text{ div } 10 = 0 & 6 \text{ mod } 10 = 6 \end{array}$$

Ahol a `div` a maradék nélküli osztás műveletét, a `mod` az osztás maradékát jelöli, az eredmény pedig az utolsó oszlopból, visszafele olvasható össze. Ez decimális számrendszer esetén eléggé értelmetlen számolásnak tűnhet, de más számrendszerekben ugyanezt az eljárást fogjuk használni.

Egy számrendszer alapja tetszőleges egynél nagyobb egész lehet, például a babiloniaiak a 60-as számrendszert használták, ahol 60 különböző számjegyre van szükség. Az assembly programozó számára a két legfontosabb a bináris és a hexadecimális számrendszer.

2.1. Bináris számrendszer

„10-féle ember létezik: az egyik érti a bináris aritmetikát, a másik nem!"

A PC-k technikai okokból nem decimális, hanem bináris (2-es) számrendszert használnak, ezért gépközeli programozáshoz elkerülhetetlen ennek ismerete. A bináris számok két számjegyből (0 és 1) állnak, utánuk egy „b" betűt írunk, így különböztetjük meg őket a decimálisaktól. A decimális és bináris számok közötti átváltás az előbbiekben ismertetett eljárással történik:

- Átváltás binárisról decimálisra
  $1001101b = 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 77$

[4] Ettől a szemléletmódtól különbözik például a római rendszer, ahol számjegyek helyett ún. számjelek vannak, ami eléggé nehézkes számolást okoz. (Mindazonáltal ők adták a vízvezetéket, csatornázást, utakat, öntözést, gyógyászatot, népfürdőt, közbiztonságot, ...)

- Átváltás decimálisról binárisra
  $42 \text{ div } 2 = 21 \quad 42 \text{ mod } 2 = 0$
  $21 \text{ div } 2 = 10 \quad 21 \text{ mod } 2 = 1$
  $10 \text{ div } 2 = 5 \quad 10 \text{ mod } 2 = 0$
  $5 \text{ div } 2 = 2 \quad 5 \text{ mod } 2 = 1$
  $2 \text{ div } 2 = 1 \quad 2 \text{ mod } 2 = 0$
  $1 \text{ div } 2 = 0 \quad 1 \text{ mod } 2 = 1$
  $\Rightarrow 42 = 101010b$

2.2. Hexadecimális számrendszer

A bináris számok nagyon „hosszúak”, pl. $5000 = 1001110001000b$. Megoldás a hexadecimális (16-os) számrendszer használata, ami „tömör” és könnyen lehet oda-vissza átváltani bináris és hexadecimális között. Itt 16 számjegyünk van (0-9, valamint A, B, C, D, E, F), melyeknek alaki értéke a 1. táblázatban található.

hex.	dec.	hex.	dec.	hex.	dec.	hex.	dec.
0	0	4	4	8	8	*C*	12
1	1	5	5	9	9	*D*	13
2	2	6	6	*A*	10	*E*	14
3	3	7	7	*B*	11	*F*	15

1. táblázat. A hexadecimális számjegyek alaki értékei

Egy hexadecimális szám csak 0-9 számjeggyel kezdődhet (szükség esetén 0-t írunk elé) és egy „h” betű hozzáadásával különböztetjük meg őket a decimális számoktól. A decimális és hexadecimális közötti váltás a már ismertetett eljárással történik, a bináris és hexadecimális közötti konverzió azonban sokkal egyszerűbb, ha észrevesszük, hogy egy hexadecimális számjegynek pontosan négy bináris számjegy felel meg (esetleges vezető nullákkal), hiszen $2^4 = 16$.

- Átváltás hexadecimálisról decimálisra
  $0D9FAh = 13 \cdot 16^3 + 9 \cdot 16^2 + 15 \cdot 16^1 + 10 \cdot 16^0 = 55802$

- Átváltás decimálisról hexadecimálisra
  $348 \text{ div } 16 = 21 \quad 345 \text{ mod } 16 = 12$
  $21 \text{ div } 16 = 1 \quad 21 \text{ mod } 16 = 5$
  $1 \text{ div } 16 = 0 \quad 1 \text{ mod } 16 = 1$
  $\Rightarrow 348 = 15Ch$

- Átváltás bináris és hexadecimális között
  0010 1101 0101 1111
  2 *D* 5 *F*
  $\Rightarrow 10\ 1101\ 0101\ 1111b = 2D5Fh$

2.3. Negatív számok néhány lehetséges reprezentálása

Innentől kezdve ha egy számról beszélünk, mindig adott számú bitet tartalmazó (8, 16 vagy 32 – bájt, word ill. dword esetén) számokra gondolunk, esetleges vezető nullákkal. Így van értelme a legmagasabb helyiértékű bitről beszélni (rendre a 7., 15., és 32. bit, lásd még a 2. ábrát).

- Előjelbites reprezentáció
  A legmagasabb helyiértékű bit jelzi az előjelet. Mivel így lenne egy -0 szám is, ez egy *extremális* elemet jelent, aminek a jelölése: NaN (Not a Number).

- Eltolt ábrázolás
  A számtartományt eltoljuk negatív irányba a számtartomány fele minusz eggyel.

- 1-es komplemens
  Egy szám negáltját úgy kapjuk, ha a bitjeit invertáljuk:
  $0110\ 1001b \Rightarrow 105$
  $1001\ 0110b \Rightarrow -105$

- 2-es komplemens
  Képzése: az egyes komplemenst növeljük eggyel:
  $0010\ 1110b \Rightarrow 46$
  $1101\ 0001b$
  $1101\ 0010b \Rightarrow -46$

Például 8 bites számok esetén az értékeket a 2. táblázat tartalmazza. **Ha negatív számokról beszélünk, ezentúl mindig a 2-es komplemens ábrázolást értjük alatta**, ezért a 3. táblázatban már csak ezeket az értékeket tüntetjük fel. Vegyük észre, hogy a legmagasabb helyiértékű bit 2-es komplemens esetén is meghatározza az előjelet. A számtartományokat a 4. táblázatban foglaljuk össze.

hexa	**dec**	**előjelbit**	**eltolt**	**1-es**	**2-es**
$00h$	0	0	-127	0	0
$01h$	1	1	-126	1	1
$02h$	2	2	-125	2	2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$7Fh$	127	127	0	127	127
$80h$	128	NaN	1	-127	-128
$81h$	129	-1	2	-126	-127
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$0FDh$	253	-125	126	-2	-3
$0FEh$	254	-126	127	-1	-2
$0FFh$	255	-127	128	NaN	-1

2. táblázat. Negatív számok reprezentálása egy bájton

2.4. Átvitel és túlcsordulás

Láttuk, hogy egy bájtos (wordös, dwordös) számot értelmezhetünk „egyszerű” decimális számként és előjeles egészként is (hamarosan még több értelmezést is megismerünk). Ez ne zavarjon meg senkit, a számítógép egyes utasításai egyértelműen meghatározzák, hogy milyen értelmezést használnak (pl. kétfajta szorzásunk lesz: előjeles és előjel nélküli számokra külön). Ezt az alapelvet jó szem előtt tartani.

Mivel a számítógépben a számok fix (8, 16 vagy 32) biten vannak ábrázolva, ezért valójában *maradékosztály*okkal dolgozunk. Ha mondjuk két darab egy bájton ábrázolt, nemnegatív egészként értelmezett számot összeadunk vagy kivonunk, az eredményt modulo 256 kapjuk meg. Például: $255 + 1 = 0$ vagy $250 + 8 = 2$. Hasonlóan a kivonásra: $0 - 1 = 255$ vagy $6 - 9 = 253$. Amikor ilyen módon átlépjük a számtartomány határát, az mondjuk, hogy *átvitel* keletkezik.

De nézzük csak meg a 2. táblázatot! Ha a bájtot 2-es komplemensként értelmezzük, akkor az előző egyenlőségek így néznek ki: $-1+1 = 0$, $-6+8 = 2$, illetve $0-1 = -1$ és $6-9 = -3$, amelyek megerősítik,

hexa	dec	2-es	hexa	dec	2-es
$0000h$	0	0	$00000000h$	0	0
$0001h$	1	1	$00000001h$	1	1
$0002h$	2	2	$00000002h$	2	2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$7FFFh$	32.767	32.767	$7FFFFFFFh$	2.147.483.647	2.147.483.647
$8000h$	32.768	-32.768	$80000000h$	2.147.483.648	$-2.147.483.648$
$8001h$	32.769	-32.767	$80000001h$	2.147.483.647	$-2.147.483.647$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$0FFFDh$	65.533	-3	$0FFFFFFFDh$	4.294.967.293	-3
$0FFFEh$	65.534	-2	$0FFFFFFEh$	4.294.967.294	-2
$0FFFFh$	65.535	-1	$0FFFFFFFh$	4.294.967.295	-1

3. táblázat. Negatív számok reprezentálása wordön és dwordön

Bitek száma	8	16	32
Előjel nélkül	0-255	0-65.535	0-4.294.967.295
Előjelesen	-128-127	-32.768-32.767	$-2.147.483.648$-2.147.483.647

4. táblázat. Számtartományok

hogy a 2-es komplemens helyes választás volt a negatív számok ábrázolására, hiszen ebben az esetben az átvitel nem zavar minket. Mivel azonban ebben az esetben is maradékosztályokkal dolgozunk (csak más reprezentáns elemekkel), így a probléma nem oldódott meg, csak eltolódott: ha megnézzük például a $127+1$ összeadást, akkor láthatjuk, hogy a 2-es komplemens szerint az eredmény -128. Amennyiben ilyen helyzet áll elő, azaz a 2-es komplemens értelmezésben lépjük át a számtartomány határát, *túlcsordulás*ról beszélünk.

2.5. Racionális számok néhány lehetséges reprezentálása

Decimális számrendszerben bevezethetjük a tizedespontot[5] és a pont jobboldalán álló számjegyek helyiértékének 10 negatív kitevőjű hatványait feleltetjük meg:

$$65.427 = 6 \cdot 10^1 + 5 \cdot 10^0 + 4 \cdot 10^{-1} + 2 \cdot 10^{-2} + 7 \cdot 10^{-3}$$

Ez az eljárás bináris számrendszerben is használható:

$$101.011b = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 5.375$$

A számítógépekben kétféleképpen szokás a racionális számokat ábrázolni: az egyszerűbb, szoftveresen is könnyen megvalósítható *fixpontos*, illetve a hardver (FPU) által támogatott *lebegőpontos* ábrázolással.

2.5.1. Fixpontos ábrázolás

Amint azt a neve is mutatja, ebben esetben a pont helye rögzített, azaz minden szám egészrésze és törtrésze adott számú biten van kódolva. Például azt mondjuk, hogy legyen a 32 bites EAX regiszter felső 16 bitje

[5]Mivel a `NASM` (és a legtöbb programozási nyelv) az angolszász országokban elterjedt tizedespontot használja az általunk megszokott tizedesvessző helyett, ezért a továbbiakban mi is ezt fogjuk használni.

az egészrész (előjelesen), az AX pedig a törtrész. Ekkor valójában az EAX az árázolt szám 2^{16}-szorosát tartalmazza.

Vegyük észre, hogy két fixpontos számot összeadva vagy kivonva, illetve egy fixpontos számot egész (azaz nem fixpontos formában lévő) számmal szorozva vagy osztva az eredményt is fixpontos formában kapjuk.

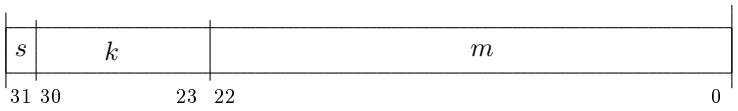
2.5.2. Lebegőpontos ábrázolás

A fixpontos ábrázolás rugalmatlansága rögtön látszik: ha például nullához közeli számokat akarunk ábrázolni, akkor az egészrész nulla lesz, de ezt továbbra is az adott számú biten ábrázolja és nem lehet biteket „átcsoportosítani" a törtrész számára (vagy fordítva). Ezt küszöböli ki a lebegőpontos ábrázolás, méghozzá a *normalizálás* segítségével.

Azt mondjuk, hogy egy bináris szám normalizált, ha $(-1)^s \cdot 1.m \cdot 2^k$ alakú, ahol m bináris egész. Nyilvánvaló, hogy minden nullától különböző szám normalizált formára alakítható. A k a szám *karakterisztikája*, az m a *mantisszája* és s az előjele. A lebegőpontos számok két legfontosabb fajtája az *egyszeres pontosságú* és a *dupla pontosságú* típus.

- Egyszeres pontosságú lebegőpontos szám

  Ezt a számot 32 biten ábrázolja a gép: 1 bit előjel, 8 bit karakterisztika és 23 bit mantissza, ld. 3. ábra. Figyelem: a karakterisztika nem 2-es komplemensként, hanem eltoltan van ábrázolva (ld. 2. táblázat), azaz a tényleges érték $k - 127$!



3. ábra. Egyszeres pontosságú lebegőpontos szám

  Néhány k és m értéknek speciális jelentése van, ld. 5. táblázat.

k	m	jelentés
0	0	± 0
0	$\neq 0$	denormalizált szám: $(-1)^s \cdot 0.m \cdot 2^{-126}$
$0FFh$	0	$\pm\infty$
$0FFh$	$\neq 0$	extremális elem (NaN)

5. táblázat. Egyszeres pontosságú lebegőpontos szám speciális értékei

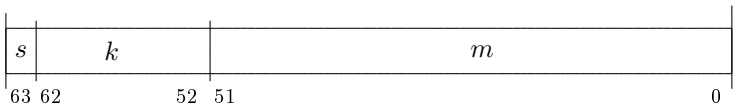
  Például: $-73.90625 = -1001001.11101b = (-1)^1 \cdot 1.00100111101b \cdot 2^6$, tehát a fixpontos ábrázolása: 1 10000101 00100111101000000000000.

- Dupla pontosságú lebegőpontos szám

  Az előjel továbbra is 1 bit, de a karakterisztika 11 bit a mantissza pedig 52 bit, azaz összesen 64 bit, ld. 4. ábra. Mivel itt a kakterisztika 11 bit, ezért az eltolás nem -127, hanem -1023.

  Néhány k és m értéknek itt is speciális jelentése van, ld. 6. táblázat.

A racionális számok használatához az FPU (koprocesszor) nyújt utasításokat, ld. 11. fejezet.



4. ábra. Dupla pontosságú lebegőpontos szám

k	m	jelentés
0	0	± 0
0	$\neq 0$	denormalizált szám: $(-1)^s \cdot 0.m \cdot 2^{-1022}$
$7FFh$	0	$\pm\infty$
$7FFh$	$\neq 0$	extremális elem (NaN)

6. táblázat. Dupla pontosságú lebegőpontos szám speciális értékei

2.6. Szövegek néhány lehetséges reprezentálása

A szövegek (*string*ek) két alapvető reprezentálási módja az ASCII és a UNICODE. ASCII esetén egy karaktert 1 bájtal ábrázolunk (azaz 256 különböző karakterünk lehet), ez elég a betűk, számok, írásjelek és néhány speciális karakter kifejezésére. UNICODE esetén 2 bájton írunk le egy karaktert (azaz 65536 karaktert tudunk megkülönböztetni), ami már számos speciális karakter és keleti írásjelek kifejezésére is alkalmas. A Windows külön eljárásokat biztosít mindkét kódolás használatának esetére (az „A" betűs az ASCII, a „W" betűs a UNICODE). A továbbiakban az ASCII kódolást használom a példákban. A 1.2.6. szakaszban található hivatkozások között az ASCII és UNICODE táblázatokra mutató linket is találhatunk.

Szót kell ejteni néhány speciális karakterről, amik gyakran előfordulnak:

szöveg vége: A Windows (a C nyelvhez hasonlóan) a szöveg végét a 0 kóddal zárja.

soremelés: A Windows az új sort két bájtal, a 13,10 (ilyen sorrendben!) bájtokkal jelöli (hexa: $0Dh$, $0Ah$), ebben a jegyzetben is leggyakrabban ez fordul elő soremelésként[6].

[6] Kivétel a magasszintű nyelvekkel való kapcsolódásnál lesz.

3

Egyszerű programok

Először néhány egyszerű példán keresztül megnézzük, hogyan néznek ki az assembly programok.

3.1. Hello World – MessageBox

Első programunkban egy *messagebox* ablakot teszünk ki a képernyőre, amelynek fejlécét és üzenetét mi adjuk meg.

```
;************************************************************************
; Program:    Hello World!
; File:       01_hello.asm
; Author:     Egri Péter "Pierre"
; Webpage:    http://fordprog.ini.hu
; Date:       01/09/2004
; Note:       Fordítás:
;                 nasmw -f obj 01_hello.asm
;                 alink -oPE 01_hello.obj
;************************************************************************
%include "win32n.inc"             ; gyakori definíciókat tartalmazó file

extern MessageBoxA                ; külso eljárások, amiket felhasználunk
import MessageBoxA user32.dll
extern ExitProcess
import ExitProcess kernel32.dll


;************************************************************************
; Adatszegmens
;************************************************************************
segment data use32 class=data     ; most következnek az adatok

 title      db 'This is the title',0
 message    db 'Hello World!',0


;************************************************************************
```

```
; Kódszegmens
;*************************************************************************
segment code use32 class=code     ; a kódszegmens következik

..start:                          ; itt kezdodik a végrehajtás

    push UINT MB_OK             ; a paraméterek átadása...
    push LPCTSTR title
    push LPCTSTR message
    push HWND NULL
    call [MessageBoxA]          ; ...és az eljárás hívása

    push UINT NULL              ; végül kilépés a programból
    call [ExitProcess]
```

Ezen az egyszerű példán keresztül vizsgáljuk meg, hogyan is néz ki egy assembly program. A ';' karakter után az adott sorba kommentet írhatunk. Mivel egy assembly forrás nehezebben olvasható mint egy imperatív (persze vannak ott is ellenpéldák), ezért a kommenteknek még fontosabb szerep jut a programozás során.

A `%include` direktíva hasonló feladatot lát el, mint `C` -ben: a fordítás során a megadott fájl tartalmát is belefordítja a kódba. A `win32n.inc` fájl a Windows programozásához kapcsolódó konstansokat definiál (a példaprogramokkal együtt megtalálható a `peldak.zip` fájlban). Ha a `win32n.inc` nem ott található, ahol a programunk, akkor az elérési utat is meg kell adnunk.

Ezután a programunkban felhasznált, külső eljárásokat definiáljuk az `extern` direktívával, majd az `import` direktívával megadjuk az eljárások törzsének helyét (pl. user32.dll).

Az *adatszegmens*ben két stringet definiálunk, amiket 0 kódokkal kell lezárni. Az utasítások a *kódszegmens*ben helyezkednek el, a végrehajtás a `..start` címkétől kezdődik. Kirakunk egy messageboxot a képernyőre, majd ha (pl. annak az *Ok* gombjára kattintva) onnan visszatérünk, befejezzük a programot az `ExitProcess` meghívásával.

Az eljárások hívása eltér a magasszintű nyelveken megszokottól: `push` utasítással egyenként kell átadni a paramétereket, majd a `call` utasítással kezdeményezzük az eljáráshívást. A nagy betűvel írt szavak (pl. `LPCTSTR`) olyan konstansok, amelyek definíciója a `win32n.inc` fájlban található.

A program fordítása: a `nasmw -fobj 01_hello.asm` és összeszerkesztése az `alink -oPE 01_hello.obj` utasításokkal történik.

3.2. Hello World – Konzol

Második programunk – hasonlóan az előzőhöz – egy üzenetet jelenít meg, de ezt egy szövegablakban (konzol) teszi. Kicsit bonyolultabb, mint a messageboxos változat, mert a konzol fájlként működik, ezért a fájlkezelő utasításokat kell alkalmazni hozzá.

```
%include "win32n.inc"

extern AllocConsole
import AllocConsole kernel32.dll
extern GetStdHandle
import GetStdHandle kernel32.dll
extern SetConsoleMode
import SetConsoleMode kernel32.dll
extern WriteFile
```

```
import WriteFile kernel32.dll
extern ReadFile
import ReadFile kernel32.dll
extern ExitProcess
import ExitProcess kernel32.dll

segment data use32 class=data
  message db 'Hello World!',13,10
  messageSize equ $-message

segment bss use32 class=bss
  hOut  resd 1
  hIn resd 1
  size  resd 1
  char  resb 1

segment code use32 class=code

..start:
    call [AllocConsole]

    push DWORD STD_OUTPUT_HANDLE
    call [GetStdHandle]
    mov [hOut],eax

    push DWORD STD_INPUT_HANDLE
    call [GetStdHandle]
    mov [hIn],eax

    push DWORD NULL
    push HANDLE [hIn]
    call [SetConsoleMode]

    push LPVOID NULL
    push LPDWORD size
    push DWORD messageSize
    push LPCVOID message
    push HANDLE [hOut]
    call [WriteFile]

    push LPVOID NULL
    push LPWORD size
    push DWORD 1
    push LPVOID char
    push HANDLE [hIn]
    call [ReadFile]

    push UINT NULL
    call [ExitProcess]
```

Itt is definiáltunk egy stringet, de ezt nem zártuk le 0-val, mert a kiírásnál a szöveg hosszát fogjuk megadni. A 13,10 azt jelenti, hogy a szöveg kiírása után a kurzor lépjen az új sorba (ld. 2.6. szakasz). A szöveg hosszát a `messageSize equ $-message` sorral definiáltuk, ami úgy működik, hogy az aktuális pozícióból levonjuk a szöveg elejének a pozícióját, így pont a méretet kapjuk.

A `bss` szegmens tulajdonképpen szintén az adatszegmenshez tartozik, de itt olyan változókat definiálunk, amelyeknek nem adunk kezdőértéket.

A programban először egy konzolablakot hozunk létre (`AllocConsole`), majd az írási és olvasási handlereket kérjük le (`GetStdHandle`), amiket a `hOut` és `hIn` változókban tárolunk (a `mov` utasítás az első operandusának értékül adja a második operandust). Később ezekkel a handlerekkel hivatkozhatunk a konzolképernyőre, illetve olvashatunk be a billentyűzetről. Az olvasás bufferelését kikapcsoljuk (`SetConsoleMode`), majd kiírjuk az üzenetet (`WriteFile`), végül várunk egy billentyűleütést (`ReadFile`). Amennyiben a bufferelést nem kapcsoltuk volna ki, a `ReadFile` nem tért volna vissza az első billentyű leütése után, csak ha betelt a buffer.

3.3. Állománymásolás

Állományt nagyon egyszerűen tudunk másolni a `CopyFile` eljárás segítségével. Most viszont egy másik lehetőséget mutatok be, ami könnyen módosítható úgy, hogy másolás közben változtassa a fájl tartalmát, pl. törölje ki a felesleges szóközöket, vagy mondjuk számlálja meg a fájlban található szavakat. Használata parancssorból: `03_copy <input.txt >output.txt`, amivel az `input.txt` tartalmát tudjuk átmásolni az `output.txt` fájlba.

```
%include "win32n.inc"

extern AllocConsole
import AllocConsole kernel32.dll
extern GetStdHandle
import GetStdHandle kernel32.dll
extern SetConsoleMode
import SetConsoleMode kernel32.dll
extern WriteFile
import WriteFile kernel32.dll
extern ReadFile
import ReadFile kernel32.dll
extern ExitProcess
import ExitProcess kernel32.dll

segment bss use32 class=bss
  hOut  resd 1
  hIn resd 1
  size  resd 1
  char  resb 1

segment code use32 class=code

..start:
    push DWORD STD_OUTPUT_HANDLE
    call [GetStdHandle]
    mov [hOut],eax
```

```
    push DWORD STD_INPUT_HANDLE
    call [GetStdHandle]
    mov [hIn],eax

    push DWORD NULL
    push HANDLE [hIn]
    call [SetConsoleMode]

read:
    push LPVOID NULL
    push LPWORD size
    push DWORD 1
    push LPVOID char
    push HANDLE [hIn]
    call [ReadFile]
    cmp dword [size],0
    je exit

    push LPVOID NULL
    push LPDWORD size
    push DWORD 1
    push LPCVOID char
    push HANDLE [hOut]
    call [WriteFile]
    jmp read

exit:
    push UINT NULL
    call [ExitProcess]
```

Ez a program nagyon hasonlít a konzolos példára, de nem hozunk létre konzolt (nincs `AllocConsole`), hiszen a standard inputról olvasunk és a standard outputra írunk. A `read` és az `exit` két címke, amikkel az adott sorokra hivatkozhatunk. Az `exit` jelöli a kilépés helyét, mivel csak akkor kell kilépni, ha a teljes fájlt átmásoltuk, azaz a beolvasás már 0 darab karakterrel tér vissza (a `cmp` végzi az összehasonlítást, a `je` pedig ugrik a címkére, ha az összehasonlítás operandusai egyenlők voltak). Amennyiben az olvasás sikeres volt, a karaktert kiírjuk és ugrunk a következő beolvasásra (`jmp`).

3.4. Listázás

A `NASM` a fordításkor a tárgykód mellett a program *listá*ját is megadja, ha használjuk a `-l listafájl` opciót. A listában jól látszik, hogy miképpen fordítja gépi kódra a `NASM` az utasításokat és hogyan tárolja az adatokat. Hogy a felesleges részek ne kerüljenek bele a listába, az újabb verziójú `NASM` a `[list -]` és `[list +]` direktívákkal megengedi a listázás felfüggesztését illetve folytatását. A listában egy sorszám, a szegmensen belüli offsetcím (a futás közbeni tényleges offsetcím ettől el fog térni), a generált kód és az eredeti forrás látszik, például az első programunk esetében:

```
 15518                                  [list -]
 15519                                  extern MessageBoxA
 15520                                  import MessageBoxA user32.dll
 15521                                  extern ExitProcess
```

```
15522                                   import ExitProcess kernel32.dll
15523
15524
15529                                   [list -]
15530                                   segment data use32 class=data
15531
15532 00000000 546869732069732074-       title      db 'This is the title',0
15533 00000009 6865207469746C6500
15534 00000012 48656C6C6F20576F72-       message    db 'Hello World!',0
15535 0000001B 6C642100
15536
15541                                   [list -]
15542                                   segment code use32 class=code
15543                                   ..start:
15544
15545 00000000 6800000000                   push UINT MB_OK
15546 00000005 68[00000000]                 push LPCTSTR title
15547 0000000A 68[12000000]                 push LPCTSTR message
15548 0000000F 6800000000                   push HWND NULL
15549 00000014 FF15[00000000]               call [MessageBoxA]
15550
15551 0000001A 6800000000                   push UINT NULL
15552 0000001F FF15[00000000]               call [ExitProcess]
```

A listázás nagyon hasznos lehet számunkra például a következő dolgok megfigyeléséhez és megértéséhez:

- stringek ábrázolása (pl. 15532-15535. sor)
- *little endian* adattárolás
- `equ` és `db` (`dw`, `dd`) közötti különbség
- stb.

4

Változók, kifejezések

Ebben a részben megismerkedünk az adatok definiálásával és tárolásával a memóriában. Az assemblyben az adatoknak nincsen állandó típusa, csak mérete. Például egy adott bájtot értelmezhetünk előjel nélküli vagy előjeles számnak, de akár ASCII karakternek is – az értelmezés mindig a használt utasítástól függ. Az adatok mérete lehet 8 bit (bájt), 16 bit (word) vagy 32 bit (dword).

4.1. Változók kezdőértékkel

Kezdőértékkel rendelkező adatokat az adatszegmensben (`data`) helyezhetünk el, pl. a következő utasítás egy 8 bites, `valtozo` nevű változót deklarál $3Ah$ kezdőértékkel:

```
valtozo db 3ah
```

Jegyezzük meg, hogy a `valtozo` a magasszintű nyelvek *pointer*ének felel meg, tehát egy memóriacímet (pontosabban offsetcímet) tartalmaz. Amennyiben a változó értékét szeretnénk, a `byte [valtozo]` formát használjuk.

Több bájtos tömböket is deklarálhatunk a `db` segítségével:

```
tomb db 1ah,2bh,3ch
```

Ez egy 3 bájtos tömb a megfelelő kezdőértékekkel. A `tomb` itt az első bájt memóriacíme lesz, a tömb elemeire a `byte [tomb]`, `byte [tomb+1]` és `byte [tomb+2]` alakokban hivatkozhatunk. A tömbök méretét a gép nem tudja meghatározni – hiszen ő egyszerű bájtsorozatként látja a memóriát – így a programozónak kell figyelnie a tömb határait. Ha a tömböt „túlírjuk", más változók értékét írhatjuk felül.

Stringeket is ilyen módon deklarálhatunk, erre már láttunk is példát:

```
message    db 'Hello World!',0
```

Itt a szöveg karaktereit tárolja egy-egy bájton a memória, az ASCII kódolást használva. Ugyanezt megadhatnánk karakterenként is:

```
message    db 'H','e','l','l','o',' ','W','o','r','l','d','!',0
```

Tömb deklarálására van még egy lehetőség, ám ilyenkor a tömb összes eleme ugyanaz lesz:

```
ujtomb times 5 db 98h
```

Ez egy 5 bájtos tömböt deklarál, amelynek minden bájtja a $98h$ értéket veszi fel.

A 16, illetve 32 bites változókat úgy deklarálhatunk, hogy `db` helyett a `dw`, illetve `dd` alakot használjuk. Ekkor azonban nagyon fontos szem előtt tartani, hogy az adatok tárolása az ún. „*little endian*" elv szerint történik, azaz **a kisebb helyiértékű bájt a kisebb memóriacímre kerül**!

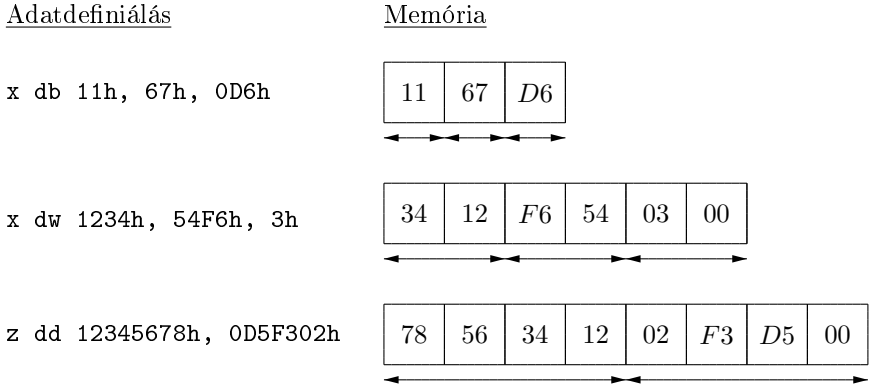
```
wvaltozo dw 1234h
```

Ekkor `word [wvaltozo]` értéke $1234h$ lesz, de `byte [wvaltozo]` értéke $34h$ és `byte [wvaltozo+1]` értéke $12h$, hiszen a kisebb helyiérték került a kisebb címre.

Tömb deklarálása hasonló, mint a `db` esetén:

```
wtomb dw 1234h,8765h
```

Ez egy két word hosszú tömb, `word [wtomb]` értéke $1234h$ és `word [wtomb+2]` értéke $8765h$. A little endian elv miatt azonban itt is `byte [wtomb]` értéke 34, `byte [wtomb+1]` értéke $12h$, `byte [wtomb+2]` értéke $65h$ és `byte [wtomb+3]` értéke $87h$. Little endianra és memóriacímzésre példák találhatók a 5. és 6. ábrákon.



5. ábra. Adatok definiálása

4.2. Változók kezdőérték nélkül

A kezdőérték nélküli változókat a `NASM` a `bss` nevű szegmensbe várja, de esetünkben az adatszegmensben tárolja[7]. Változókat csak tömbként definiálhatunk, de a tömb mérete egy is lehet. A következő sor 5 bájtnyi helyet foglal le:

```
valt    resb 5
```

Innentől ugyan úgy használható, mintha `db`-vel deklaráltuk volna, csak a kezdeti értéke nem definiált. A 16 és 32 bites tömböknek a `resw` és `resd` használatával foglalhatunk helyet.

4.3. Kifejezések

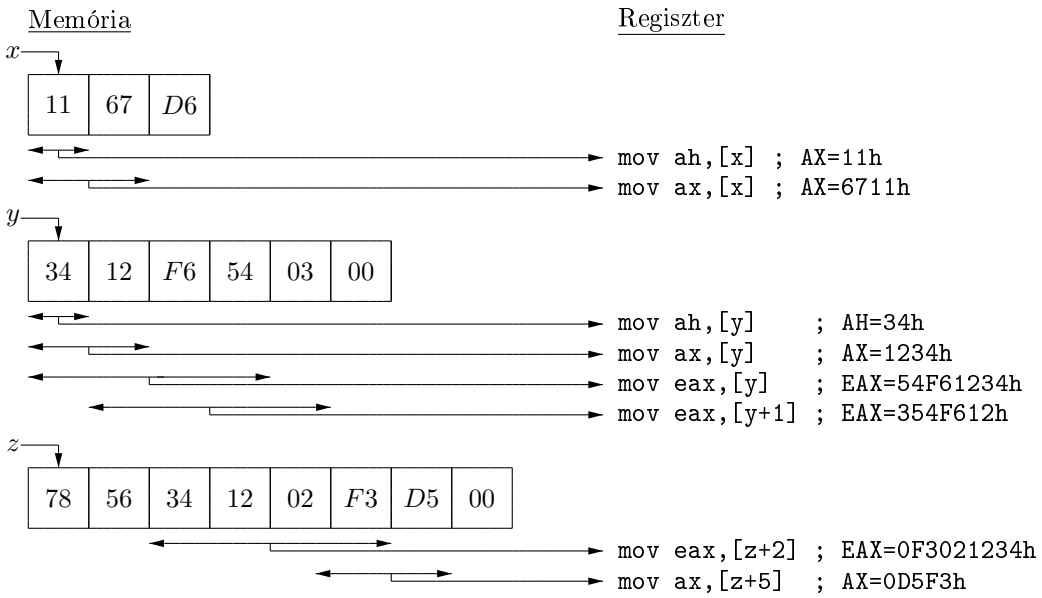
Assemblyben számkonstansok helyett fordítási időben kiértékelhető kifejezéseket is használhatunk, például:

```
mov eax,100b | 1        ; ezzel ekvivalens: mov eax,5
```

A legfontosabb használható operátorok:

A `NASM` nyújt egy speciális lehetőséget: a $ karakterrel az adott sorra hivatkozhatunk, mintha annak lenne egy címkéje és azt használnánk. Leggyakrabban szövegek hosszának meghatározásánál alkalmazzuk, amire már láttunk is példát:

[7] A `bss` szegmens az Unix tárgykód formátumából származik, Windows alatt megegyezik az adatszegmenssel.



6. ábra. Adatok felhasználása

|	bitenkénti diszjunkció
^	bitenkénti kizáró vagy
&	bitenkénti konjunkció
«	eltolás jobbra
»	eltolás balra
+	összeadás
-	kivonás
*	szorzás
/	előjel nélküli osztás
//	előjeles osztás
%	előjel nélküli modulo
%%	előjeles modulo
~	1-es komplemens

7. táblázat. A `NASM` legfontosabb operátorai

```
message db 'Hello World!',13,10
messageSize equ $-message
```

Itt tehát a $ a 10 utáni bájtra, a `message` a „H” betűre mutat, tehát a kettő különbsége adja meg a szöveg hosszát.

4.4. Szimbólumok

Az `equ` direktívával szimbólumot definiálhatunk, ami nem definiálható felül. A szimbólum nem foglal helyet az adatszegmensben, hanem fordítás közben behelyettesítődik a szimbólum konkrét értéke, ahol hivatkozunk rá. Fontos, hogy az `equ` után álló kifejezés a definiáláskor értékelődik ki, nem pedig a rá való hivatkozáskor! Leggyakrabban konstansok definiálására használjuk őket, mint az előző példában láttuk.

Figyeljünk arra, hogy a `NASM` alapból *kétmenetes* assembler[8]. Aki többmenetes működésre szeretné rábírni, az a `-O` (nagy ó) opció használatával teheti ezt meg.

[8] Amennyiben két menetben nem lehet meghatározni a szimbólumok értékét (túl bonyolult, előre mutató direktívákat használunk), az assembler hibával áll le.

5

Utasítások

Ebben részben megismerhetjük a legfontosabb assembly utasításokat. Ezeken kívül még számos utasítás létezik, amiknek az Intel processzor leírásokban, illetve a `NASM` dokumentációban nézhetünk utána.

5.1. Flagek

Az EFLAGS regiszter egyes bitjei, a *jelzőbitek*, a műveletek eredményeiről szolgáltatnak információkat. (A flagek nagy része csak a védett módban jut szerephez, ezekről nem lesz szó). A legfontosabb bitek:

S – sign (előjel): a legmagasabb helyiértékű bit (ld. 2-es komplemens, 2.3. szakasz)

P – parity: a legalacsonyabb helyiértékű bit inverze

C – carry (átvitel): számtartomány túllépése

O – overflow (túlcsordulás): az előjeles aritmetika szerint túl nagy vagy túl kicsi az eredmény

Z – zero: a művelet eredménye 0

D – direction (irányjelző): a string-kezelésnél (8.1. szakasz) jut szerephez

A C és D bitek értékét mi is megváltoztathatjuk:

`cld`	$D := 0$
`std`	$D := 1$
`clc`	$C := 0$
`stc`	$C := 1$
`cmc`	$C := \neg C$

8. táblázat. Flageket változtató utasítások

5.2. Értékadás, címzési módok

A `mov cél,forrás` (Kétoperandusú utasításoknál az első a céloperandus, a második a forrásoperandus, vö. magas szintű nyelveken *lhs* és *rhs*) értékadó utasítás nem változtatja meg a flageket. Használatánál alapszabály, hogy mindkét operandusa egyszerre nem lehet memóriacím (tehát a `mov [valtozo1],[valtozo2]` nem jó!) és a méreteiknek meg kell egyezniük. A címzési módok a következők (offset és immediate címzés értelemszerűen nem lehet céloperandusban):

```
mov ax,bx        ; regiszter
mov ax,[valtozo] ; memória
mov eax,valtozo  ; offset
mov ax,0B5FAh    ; immediate
```

A memóriacímzésnél alapértelmezésként az adatszegmenset használjuk, ha ettől el akarunk térni, az külön jelölni kell, pl. `mov ax,[es:valtozo]`. Ha a mozgatandó adat mérete nem derül ki az utasításból, akkor külön jelölni kell (a `mov [valtozo],1` nem jó függetlenül attól, hogy a változónak milyen méretet jelöltünk meg, helyette pl. a `mov byte [valtozo],1` alak használható – a `byte` helyett `word` vagy `dword` is használható a mozgatandó adat méretétől függően).

A `valtozo`-ra tekintsünk úgy, mint egy bájtra mutató *pointer*re (memóriacímre), míg a `[valtozo]` az ezen a memóriacímen kezdődő adat (aminek a méretét a konkrét utasítás határozza meg).

Ha egy nagyobb méretű regiszternek szeretnénk kisebb méretű számot értékül adni, akkor használhatók a 9. táblázatban található utasítások.

`movzx`	előjel nélküli (0-val egészít ki)
`movsx`	előjelhelyes
`cbw`	AL-t AX-be teszi előjelhelyesen
`cwde`	AX-et EAX-be teszi előjelhelyesen
`cdq`	EAX-et EDX:EAX-be teszi előjelhelyesen

9. táblázat. Speciális értékadó utasítások

Itt az EDX:EAX egy olyan 64 bites számot jelent, aminek a felső 32 bitje az EDX-ben, az alsó pedig az EAX-ben van.

A két operandus értékének felcserélésére használható az `xchg` utasítás.

5.3. Aritmetikai utasítások

`add`	összeadás (+=)
`sub`	kivonás (-=)
`inc`	növelés (++)
`dec`	csökkentés (--)
`mul`	előjel nélküli szorzás. A 8 (16, 32) bites operandust szorozza AL-lel (AX-el, EAX-el) és az eredményt az AX-ben (DX:AX-ban, EDX:EAX-ban) tárolja.
`imul`	előjeles szorzás
`div`	előjel nélküli osztás. A 8 (16, 32) bites operandussal elosztja az AX-et (DX:AX-et, EDX:EAX-et), az eredményt az AL-ben (AX-ban, EAX-ban), a maradékot pedig AH-ban (DX-ben, EDX-ben) tárolja.
`idiv`	előjeles osztás
`adc`	összeadás a C bittel együtt
`sbb`	kivonás a C bittel együtt

10. táblázat. Aritmetikai utasítások

A legfontosabb aritmetikai utasításokat a 10. táblázat foglalja össze. Itt a DX:AX egy 32 bites számot jelent aminek a felső 16 bitje a DX-ben, az alsó pedig az AX-ben van (kompatibilitási okokból nem 32 bites regisztert használnak).

Kettőhatvánnyal való szorzásra, osztásra a logikai utasításonál újabb lehetőségeket fogunk megismerni.

Az aritmetikai utasítások hatása a flagekre (példák):
$4F3Dh + 0FD81h \Rightarrow (O:0, S:0, Z:0, C:1)$
Megvalósítás:

```
mov ax,4F3Dh
add ax,0FD81h
```

$6B90h + 2D31h \Rightarrow (O:1, S:1, Z:0, C:0)$
$4064h + 0F0Fh \Rightarrow (O:0, S:0, Z:0, C:0)$
$4652h + 0F0F0h \Rightarrow (O:0, S:0, Z:0, C:1)$
$61h - 65h \Rightarrow (O:0, S:1, Z:0, C:1)$

5.4. Logikai és bit-utasítások

`and`	bitenkénti konjunkció
`test`	a flageket úgy állítja át, mint az `and`, de nem tárolja az eredményt
`or`	bitenkénti diszjunkció
`xor`	bitenkénti kizáró vagy (használható nullázásra: xor eax,eax)
`not`	1-es komplemens képzés
`neg`	2-es komplemens képzés
`ror`	jobbra forgatás, az átforduló bit a C flagbe kerül
`rol`	balra forgatás, az átforduló bit a C flagbe kerül
`shr`	jobbra shiftelés, a kihulló bit C-be kerül (használható előjel nélküli 2-hatvánnyal való osztásra)
`shr`	balra shiftelés, a kihulló bit C-be kerül (használható előjel nélküli 2-hatvánnyal való szorzásra)
`sar`	aritmetikai shiftelés jobbra (S flag értéke csorog be), a kihulló bit C-be kerül (használható előjeles 2-hatvánnyal való osztásra)
`rcr`	ua. mint a `ror`, csak a „becsorduló” bit a C flag
`rcl`	ua. mint a `rol`, csak a „becsorduló” bit a C flag
`bt`	bit-tesztelés (a C flagbe kerül az eredmény)
`btr`	`bt` + törli a bitet
`bts`	`bt` + beállítja a bitet
`btc`	`bt` + negálja a bitet

11. táblázat. Logikai és bit-utasítások

A legfontosabb logikai utasítások a 11. táblázatban találhatók meg. Egy számot saját magával xor-olva az eredmény mindig nulla, ezért ezt az utasítást gyakran regiszterek nullázására használják, pl. `xor eax,eax`.

5.5. Programkonstrukciók

A magas szintű nyelveken megszokott elágazások és ciklusok assemblyben nem állnak rendelkezésünkre[9], ezeket ugróutasításokkal – vö. a magas szintű nyelvek „mostoha” `goto` utasításai – tudjuk megvalósítani. A programok tervezésénél így kézenfekvő folyamatábrát használni.

[9] A 7. fejezetben írunk olyan makrókat, amelyekkel – bizonyos mértékig – ezek a programkonstrukciók megvalósíthatók.

5.5.1. Címkék

Címkével egy sornak adhatunk nevet, amire később hivatkozhatunk. A címkét kettőspont választja el az utána következő utasítástól.

A ponttal kezdődő címkék lokálisak, azaz az őket megelőző globális címkéhez tartoznak. Például:

```
elso:     ; ez az 'elso' címke
.ide:     ; ez az 'elso.ide' címke
          ; az 'elso.ide' címkére hivatkozhatunk '.ide' néven
          ; a 'masod.ide' címkére csak a teljes nevén hivatkozhatunk

masod:    ; ez a 'masod' címke
.ide:     ; ez pedig a 'masod.ide' címke
          ; a 'masod.ide' címkére hivatkozhatunk '.ide' néven
          ; az 'elso.ide' címkére csak a teljes nevén hivatkozhatunk
```

5.5.2. Feltétel nélküli ugrás

```
      mov ax,3FD4h
      jmp cimke
      xor ax,ax
cimke:
      mov bx,ax
```

A `jmp` utasítás a címkével jelölt memóriacímre ugrik (a szegmens alapértelmezésben a CS), tehát a `xor ax,ax` utasítást átugorjuk.

5.5.3. Feltételes ugrások és összehasonlítás

`jz`, `jnz`, `jc`, `jnc`, `jo`, `jno`, `js`, `jns`	a flagek értékétől függően ugrik vagy nem
`cmp`	összehasonlítás. A flageket ugyanúgy állítja át, mint a `sub`, de a kivonás eredményét nem tárolja el.
`je`, `jne`	egyenlőségvizsgálat, megegyezik a `jz`, `jnz` utasításokkal
`jb`, `jnb`, `jbe`, `jnbe`, `ja`, `jna`, `jae`, `jnae`	előjel nélküli összehasonlítások
`jl`, `jnl`, `jle`, `jnle`, `jg`, `jng`, `jge`, `jnge`	előjeles összehasonlítások
`jcxz`, `jecxz`	ugrik, ha CX/ECX nulla
`loop`	csökkenti ECX-et, és ha ezután ECX$\neq$ 0, akkor ugrik a címkére
`loopz`≡`loope`, `loopnz`≡`loopne`	mint a `loop`, de a Z flag értékét is figyelembe veszi

12. táblázat. Összehasonlító és ugróutasítások

A feltételes ugróutasításoknál (12. táblázat) a címzés *relatív*, ami azt jelenti, hogy a kódba nem az offsetcím kerül, hanem hogy az aktuális pozícióhoz képest hova kell ugrani. A relatív címet maximum

2 bájton tárolja (előjelesen), így nagyobb feltételes ugrásokat egy kisebb feltételes, és egy feltétel nélküli ugrás kombinációjával lehet megoldani.

5.6. Egyéb utasítások

Van még néhány ritkábban előforduló utasítás, amikről nem árt tudni (13. táblázat). Rengeteg speciális utasítás (védett mód, Pentium, MMX, stb.) is létezik, amik túlmutatnak ezen jegyzet keretein. A `NASM` vagy – még részletesebben – az Intel dokumentációkban megtalálható a leírásuk.

`enter`, `leave`	helyfoglalás és -felszabadítás a veremben (ld. eljárások)
`lahf`, `sahf`	flagek AH-ba, illetve AH a flagekbe töltése
`bswap`	32 bites regiszterek bájtsorrendjének megfordítása
`nop`	nem csinál semmit
`set`cc	8 bites operandus beállítása 0-ra vagy 1-re a feltétel szerint (konkrétan pl. `setne`, `setz`, `setg`, stb.)
`shld`, `shrd`	dupla pontosságú shiftelés
`xlatb`	táblázatokhoz – AL-be `[EBX+AL]`-t tölti
`in`, `out`, `insb`, `insw`, `insd`, `outsb`, `outsw`, `outsd`	portkezelés
`daa`, `das`, `aaa`, `aas`, `aad`, `aam`	BCD és ASCII aritmetika
`bsf`, `bsr`	legkisebb és legnagyobb 1-es bit keresése
`les`, `lds`	az ES illetve a DS regiszterek értékének beállítása

13. táblázat. Egyéb utasítások

Feladat: Írj olyan programrészletet, ami az AL regiszter tartalmát „tükrözi", azaz a bitek sorrendjét megfordítja (pl. $11001001b \Rightarrow 10010011b$)!

Megoldás: *Egy lehetséges megoldás:*

```
  mov ecx,8
fordit:
  shr al,1
  rcl ah,1
  loop fordit
  shr ax,8
```

6

Verem, eljárások

Gyakran ismétlődő feladatokra általában eljárásokat írunk. Az assemblyben az eljárások működésének megértéséhez elengedhetetlen a *verem* ismerete.

6.1. Szegmens és offset regiszterek

A memóriacím mindig egy szegmens és egy offset részből áll: úgy szokták szemléltetni, hogy ha a memória egy könyv lenne, akkor a szegmenscím adná meg az oldalszámot, az offsetcím pedig azon belül pontosan meghatározná a betűt. Amikor egy névvel hivatkozunk egy változóra (pl. `mov ax,[adat]`) vagy egy címkével hivatkozunk egy utasításra (pl. `jmp vege`), akkor látszólag nem adjuk meg a szegmenseket. Ilyenkor az alapértelmezett szegmenseket – adatszegmens illetve kódszegmens – használja a gép, ettől eltérni a szegmensek kiírásával lehet (pl. `mov eax,[es:esi]`). Gyakran használt szegmens- és offsetregiszter-párok:

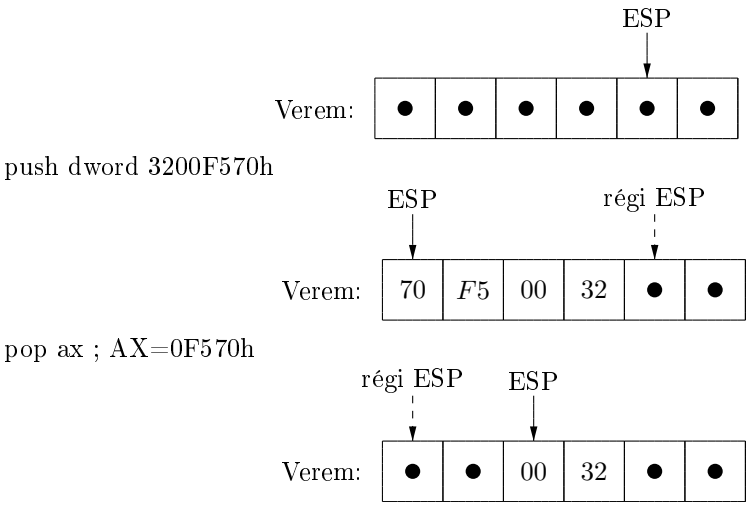
- CS:EIP
  Megadja a következő utasítás helyét a memóriában. Automatikusan változnak, csak ritka esetben van szükség a direkt módosításukra.
- SS:ESP
  A verem legfelső bájtjára mutat. Ez is automatikusan módosul a veremkezelő utasítások hatására, de néha szükség van közvetlenül is beállítani az értékét.
- DS:
  Ez jelöli az adatszegmenset, az ezen belül elhelyezett adatokra leggyakrabban a változó nevével, vagy az ESI, EDI regiszterekkel hivatkozunk.

Mekkora lehet egy szegmens mérete? Mivel az offsetregiszterek 32 bitesek, ezért 2^{32} bájt lehet egy szegmens mérete. Ez 2^{22} KB, azaz 2^{12} MB, azaz 2^2 GB, tehát elvileg 4 gigabájt lehet egy szegmens. Továbbá 2^{16} szegmens lehet, ezért elvileg hatalmas memóriaméretek is kezelhetők ezzel a címzésmóddal[10].

6.2. Verem

A verem egy LIFO (last-in-first-out) adatszerkezet, azaz amit a legutoljára beletettünk, azt vehetjük ki legelőször. A verem is a memóriában van, a legfölső elemére az SS:ESP mutat. Verembe a `push` utasítással helyezhetünk adatot és a `pop` utasítással vehetjük ki onnan (az utasítások automatikusan változtatják az ESP-t). A verem fordítva helyezkedik el a memóriában, tehát, ha belerakunk valamit a verembe, az ESP csökken, ha kiveszünk nő. A *little endian* elv itt is érvényes (ld. 7. ábra).

[10] A flat mode használata esetén a gyakorlatban egy szegmens létezik, minden szegmensregiszter erre mutat és ezen keresztül érhető el a teljes memória. A védelem miatt viszont az operációs rendszer meg tudja akadályozni, hogy más program kódját vagy adatait véletlenül felülírjuk.



7. ábra. A verem működése. A fekete körök az jelentik, hogy az adott bájt értéke ismeretlen vagy nem definiált.

Mire jó a verem? Például regiszterek ideiglenes elmentésére, memória-memória értékadásra (amikor a `mov` nem használható), eljárásoknál paraméterátadásra, lokális változók tárolására, stb. A veremkezelő utasítások néhány változata: `pushad`, `popad` (több regiszter belerakása és kivétele a veremből), `pushf`, `popf` (flagek belerakása és kivétele a veremből).

6.3. Eljárások

Eljárások törzsét egy címke (az eljárásnév) és az eljárásból való visszatérést jelölő `ret` utasítás közé írhatjuk:

```
eljaras:
      ; utasítások
  ret
```

Meghívása: `call eljaras`. Az eljárások hívásakor az EIP regiszter a verem tetejére kerül, mert a végén a visszatérésnél tudni kell, hogy hol folytatódik a program. Ebből következik, hogy nagyon ajánlatos **visszatérés előtt minden értéket kivenni a veremből, amit beleraktunk**[11]! Figyeljünk arra is, hogy az eljárásoknak nincsenek „lokális regiszterei", azaz ha megváltoztatjuk egy regiszter értékét, az a visszatérés után is megmarad, tehát eljáráshívások előtt a szükséges regiszterek értékeit nem árt elmenteni. Sem a `call`, sem a `ret` nem változtatja meg a flagek értékét.

6.3.1. Paraméterátadás regisztereken keresztül

Ez a legegyszerűbb módszer, a paramétereket előre meghatározott regiszterekbe töltjük és a visszatérési értékeket is regiszterekben kapjuk meg. Kisebb eljárásainkban általában ezt használjuk. Hátránya, hogy a regiszterek (és így az átadható paraméterek) száma erősen korlátozott.

Most néhány kisebb feladat megoldására nézünk eljárásokat. Az eljárásoknak az egyszerűség kedvéért regisztereken keresztül adjuk át a paramétereket[12].

[11] Lásd még az `enter` és `leave` utasításokat!

[12] A gyakorlatban a számolásigényes feladatokra ritkán írnak eljárásokat. Gyakrabban fordul elő, hogy előre kiszámolják

Fibonacci számok. Számoljuk ki a $F_1 = 1$, $F_2 = 1$ és $F_i = F_{i-1} + F_{i-2}$ $(i \geq 3)$ rekurzióval adott Fibonacci sorozat n. tagját!

```
; IN:     ECX - sorszám
; OUT:    EAX - Fibonacci szám
;
; Kiszámolja az n. Fibonacci számot
;
fib:
  mov eax,1
  mov ebx,1
  cmp ecx,2
  ja .c1
  ret
.c1:
  sub ecx,2
.c2:
  xadd eax,ebx
  loop .c2
  ret
```

Az eljárásban végig az EAX-ben van az i-edik, az EBX-ben az $(i-1)$-edik Fibonacci szám. A `xadd` utasítás felcseréli a két paraméterét és összeadja őket. Ha valaki nem ismerte volna, használhatta volna helyette az

```
xchg eax,ebx
add eax,ebx
```

utasításokat.

Prímszámok. Számoljuk ki az n. prímszámot!

```
; IN:      ECX - sorszám
; OUT:     EBX - prím
;
; Kiszámolja az n. prímszámot
;
nthprime:
  mov ebx,2
  loop .c1
  ret
.c1:
  mov esi,2
  inc ebx
.c2:
  xor edx,edx
  mov eax,ebx
  div esi
  or edx,edx
```

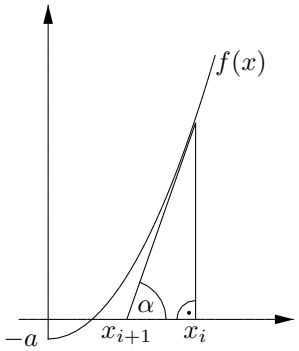
az adatokat és az adatszegmensben tárolják őket, így a futás során egyetlen memóriacímzéssel megvan a kívánt eredmény a „hosszadalmas” számolás helyett. Lásd még az `incbin` direktívát.

```
jz .c1
inc esi
cmp esi,ebx
jb .c2
loop .c1
ret
```

A program azt a nagyon egyszerű módszert használja annak eldöntésére, hogy egy m szám prím-e, hogy 2-től $m-1$-ig minden számmal megpróbálja elosztani m-et és ha egyik sem osztja, akkor prím. Erre szolgál a `.c2` ciklus, ahol EBX tartalmazza m-et. Mivel az n. prímre vagyunk kíváncsiak, ezért az egészet n-szer megismétli (`.c1` ciklus).

Közelítő gyökvonás. Amennyiben ismerjük a Newton-módszert, egyszerűen írhatunk közelítő gyökvonó eljárást. (Pontosabb gyökvonásról a 11. fejezetben lesz szó.)

A Newton-módszert – ami nemlineáris függvények gyökeit határozza meg – az $f(x) = x^2 - a$ függvényre alkalmazzuk, hiszen ennek gyöke éppen $x = \sqrt{a}$ lesz. Az eljárás lényege, hogy tetszőleges x_0-ból (pl. a-ból) indulva vesszük az $f(x)$ görbe érintőjét az $f(x_i)$ pontban és az x_{i+1} pontot az érintő és az y tengely metszeténél vesszük fel. Az így kapott sorozat $\sqrt{a}$-hoz konvergál. Ezt mutatja a 8. ábra.



8. ábra. Newton-módszer a gyök meghatározására

Ebben az esetben:

$$\frac{f(x_i)}{x_i - x_{i+1}} = \tan\alpha = f'(x_i)$$

Átalakítva:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{x_i^2 - a}{2x_i} = \frac{\left(x_i + \frac{a}{x_i}\right)}{2}$$

Azaz az algoritmus felépítése a következő:

1.) $x_0 := a$

2.) $x_{i+1} := \left(x_i + \frac{a}{x_i}\right)/2$

3.) ha $x_{i+1} = x_i$, akkor $\sqrt{a} = x_i$, különben `goto 2`

Ezt felhasználva a következő assembly programot kapjuk (most a ∼EBX, x_i ∼ESI és x_{i+1} ∼EDI):

```
; IN:       EBX - Ebbôl kell gyököt vonni
; OUT:      EDI - Gyök
;
; Közelítô gyökvonás
;
sqrt:
    mov esi,ebx
.c: mov eax,ebx
    xor edx,edx
    div esi
    lea edi,[eax+esi]
    shr edi,1
    cmp esi,edi
    je .sqrtend
    mov esi,edi
    jmp .c
.sqrtend:
    ret
```

A `lea edi,[eax+esi]` utasítással az `[eax+esi]` memóriacímét töltjük be EDI-be, azaz pont EAX+ESI értékét. Nyugodtan használhatjuk helyette a:

```
mov edi,esi
add edi,eax
```

utasításokat.

6.3.2. Paraméterátadás a vermen keresztül

Ebben a részben az eljáráshívás *stdcall* típusát ismerhetjük meg, amit a Windows használ, így az API hívásoknál mi is ezt használjuk. Később szó lesz a *cdecl*-ről, amit a C/C++ nyelv használ és némileg eltér a *stdcall*-tól (ld. 10. fejezet).

Vermen keresztüli paraméterátadással már találkoztunk például az első „Hello World” programban:

```
push UINT MB_OK
push LPCSTR title
push LPCSTR message
push HWND NULL
call [MessageBoxA]
```

Ha belenézünk a Windows API leírásba, a következőt látjuk:

```
int MessageBox(
  HWND hWnd,          // handle of owner window
  LPCTSTR lpText,     // address of text in message box
  LPCTSTR lpCaption,  // address of title of message box
  UINT uType          // style of message box
);
```

Ebből látszik az *stdcall* két legfontosabb szabálya:

- a paramétereket fordított sorrendben helyezzük a verembe és
- a meghívott eljárás gondoskodik a paraméterek eltávolítására a veremből.

A visszatérési értéket az EAX (AX, AL – mérettől függően) regiszterben kapjuk vissza. Figyeljünk arra, hogy a `call` utasítás miatt **a verem tetejére a visszatérési cím kerül**. A paramétereket általában közvetlenül memóriacímzéssel érjük el, amihez az EBP regisztert használjuk. A visszatérésnél a paramétereket el kell távolítani a veremből, ezért a `ret` utasításnak paraméterül megadható, hogy a visszatérési cím kivétele után még mennyi bájtot vegyen ki a veremből.

Az eljáráshíváskor az eljárás neve után álló „A" azért kell, mert néhány eljárásnak két formája van: „A"-s és „W"-s (ld. 2.6. szakasz). A hívásnál azért raktuk a függvény nevét szögletes zárójelbe (`call [MessageBoxA]`), mert az `import` direktíva használatánál nem a függvény címét, hanem egy arra mutató pointert kapunk.

A paraméterek fordított sorrendű verembe helyezése lehetőséget ad arra, hogy az eljárást változó számú paraméterrel hívjuk meg és az első paraméterben – ami a veremben közvetlenül az EIP érték alatt van – határozzuk meg a paraméterek számát. Ilyen változó számú paraméterrel meghívható függvény pl. az `int fprint(char*, ...)` a C nyelven.

Szám kiírása. Most nézzünk egy olyan eljárást, ami a paraméterként kapott számot kiírja a konzolra. Az eljárás lényege, hogy a számot folyamatosan osztja tízzel és a maradéknak megfelelő számjegyet a verembe helyezi, amíg a szám nulla nem lesz. A végén pedig kiírja a veremben lévő számjegyeket. Mivel a `push` utasítás operandusa nem lehet 8-bites regiszter, ezért közvetlenül a memóriába fogjuk írni az értéket és az ESP regiszter értékét is módosítjuk.

```
;*************************************************************************
; void print_int(int);
;*************************************************************************
print_int:
    mov ebp,esp
    mov eax,dword [ss:ebp+4]
    mov ebx,10
    push word 0A0Dh
    or eax,eax
    jnz .c
    dec esp
    mov byte [ss:esp],'0'
    jmp .vege
.c:
    or eax,eax
    jz .vege
    xor edx,edx
    div ebx
    add dl,'0'
    dec esp
    mov [ss:esp],dl
    jmp .c
.vege:
    mov esi,esp
    mov ebx,ebp
    sub ebx,esp
    and esp,~11b
    push LPVOID NULL
    push LPDWORD size
    push DWORD ebx
```

```
    push LPCVOID esi
    push HANDLE [hOut]
    call [WriteFile]
    mov esp,ebp
    ret 4
```

Az eljárás kezdetén a verem tetején a visszatérési érték, alatta a paraméter helyezkedik el. Mivel az ESP-t változtatni fogjuk, ezért az eredeti értékét EBP-be mentjük. Az EAX regiszterbe rakjuk a paramétert (memóriacímzéssel szedjük ki a veremből), majd az újsor karaktereit rakjuk be. Amennyiben EAX nulla, akkor a 0 karaktert rakjuk a verembe, különben kezdődik a tízzel osztás ciklusa. A `push` utasítás nem kezeli megfelelően a bájt méretű operandust, ezért közvetlen memóriacímzéssel és az ESP regiszter változtatásával helyezzük el a karaktereket a veremben.

A kiírásnál ESP mutat a szöveg elejére[13], aminek hossza EBP−ESP. Ha az ESP értéke nem osztható néggyel (ami azt jelenti, hogy nem 32 bites adatok vannak a veremben), a `WriteFile` eljárás hibásan működik[14], ezért a paraméterek átadása előtt úgy csökkentjük ESP értékét, hogy a két kis helyiértékű bitjét töröljük.

A kiírás után visszaállítjuk ESP eredeti értékét és visszatérünk úgy, hogy a 4 bájtos paramétert is kivesszük a veremből.

Feladat: Módosítsd az eljárást úgy, hogy a negatív számokat is ki tudja írni!

Feladat: Módosítsd az eljárást úgy, hogy bináris formában írja ki a számot (2-vel való osztáshoz nem használjuk a `div`-et)!

Feladat: Módosítsd az eljárást úgy, hogy hexadecimális formában írja ki a számot (itt se használjuk a `div`-et)!

Feladat: Írj olyan programot, ami egy szám prímtényezős faktorizációját írja ki!

Feladat: Írj olyan programot, ami a pithagoraszi számhármasokat sorolja fel!

Feladat: Írj olyan programot, ami egy vektor elemeit növekvő sorrendbe rendezi.

6.3.3. Rekurzív eljárások

Az eljárások rekurzív módon meghívhatják saját magukat. Ilyenkor arra kell ügyelni, hogy az eljárás elmentse az EBP regiszter értékét és visszatérés előtt állítsa vissza az eredeti értéket, hiszen a hívó is használja azt a paraméterek elérésére. A következő példa egy szám faktoriálisát számolja ki rekurzívan:

```
;*************************************************************************
;  n! kiszámítása rekurzívan
;*************************************************************************
fakt:
    push ebp
    mov ebp,esp
    mov eax,[ss:ebp+8]
```

[13] Itt kihasználjuk azt, hogy a flat mode miatt SS=DS.

[14] Őszintén szólva ezt a korlátozást nem értem. Ha valaki tudja, miért kell a `WriteFile`-nak néggyel osztható ESP, az legyen szíves megírni nekem!

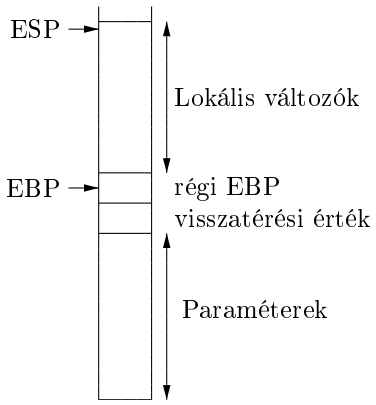
```
    or eax,eax
    jz .ret1
    dec eax
    push eax
    call fakt
    mul dword [ss:ebp+8]           ; EDX:EAX !!!!
    pop ebp
    ret 4
.ret1:
    mov eax,1
    pop ebp
    ret 4
```

Az eljárás az elején elmenti az EBP regisztert a verembe, így a paraméter címzésénél még négy bájtot hozzá kell adni a veremtetőhöz. Az eljárás működése nagyon egyszerű: ha $n = 0$, akkor 1-et ad vissza, különben rekurzívan kiszámolja $(n-1)!$-t és ezt megszorozza n-el. A szorzás tulajdonképpen az eredményt EDX:EAX-be teszi, de mi most ebből csak EAX-et használjuk fel: ha az eredmény kisebb kb. 4 millárdnál (ld. 2.3), akkor az belefér EAX-be.

Feladat: Írj olyan programot, ami kiírja a Hanoi tornyai probléma megoldását tetszőleges számú korong esetén!

6.3.4. Eljárások lokális változói

Lokális változóknak a verem tetején tudunk helyet foglalni az ESP regiszter csökkentésével. Ezeket a változókat szintén memóriacímzéssel és az EBP használatával tudjuk elérni. A verem felépítése ekkor a 9. ábrán látható.



9. ábra. Lokális változók tárolása a veremben

Ez a szerkezet már nagyon hasonlít a magasszintű nyelvek kódgenerálásánál használt *aktivációs rekord*ra.

6.4. A megszakításokról röviden

A program futását néha egy egy esemény miatt meg kell szakítani, például ha egy hiba (mondjuk nullával való osztás) történik. Ilyenkor lefut a megfelelő *megszakításkezelő* program, majd ideális esetben folytatódik a programunk végrehajtása. Tulajdonképpen tekinthetők speciális eljárásoknak is, de a paramétereket itt csak a regisztereken keresztül adhatjuk át.

A DOS-ban az operációs rendszerek különböző szolgáltatásai nem API hívásokként voltak elérhetők, hanem egy megszakításon keresztül ($21h$), ezért a programozónak is tudnia kell megszakítást kiváltani, amit az `int` utasítással tehet meg. Linux alatt a rendszer szolgáltatásait a $80h$-s megszakításon keresztül érhetjük el. A számítógép BIOS nevű része is nyújt néhány hasznos szolgáltatást. A megszakításokkal a továbbiakban nem foglalkozunk.

7

Makrók

A NASM előfeldolgozó rendszer lehetőséget nyújt makrók definiálására, amik a programozást megkönnyítik. A makrók első pillantásra hasonlítanak az eljárásokra, de a fordítás első lépésében a makró-hívások helyére fejti ki őket az assembler, nem lesz belőlük `call` utasítás. Amennyiben a NASM -ot a `-e` opcióval hívjuk meg, nem végzi el a fordítást csak kifejti a makrókat – ez segít a makrók működésének megértésében és az esetleges hibák felderítésében.

7.1. Egysoros makrók

Konstansok, egyszerű függvények definiálására használható a `%define` makró. A `%define` nem szimbólumot definiál mint az `equ`, hanem szövegszerűen behelyettesítődik a makró-hívás helyére. Lehetőség van paraméteres makrók írására is:

```
%define param(a,b) ((a)+(a)*(b))
```

Lehet felüldefiniálni, illetve a `%undef` direktívával érvényteleníteni.

7.2. Többsoros makrók

Többsoros makrókat a `%macro` direktívával lehet definiálni. Első példaként nézzünk egy nagyon hasznos makrót, amit a továbbiakban gyakran fogunk használni. Eddig a következőképpen nézett ki egy eljáráshívás:

```
push UINT MB_OK
push LPCTSTR title
push LPCTSTR message
push HWND NULL
call [MessageBoxA]
```

Ezt ezután a következő alakra egyszerűsödik az `invoke` makró segítségével:

```
invoke [MessageBoxA], NULL, message, title, MB_OK
```

Nézzük, hogyan lehet ezt megvalósítani:

```
%macro invoke 1-*
%rep %0-1
  %rotate -1
  push dword %1
```

```
%endrep
%rotate -1
call %1
%endmacro
```

A makró a következő módon működik: legalább egy paramétert vár, ami a függvény neve, utána lehet a függvény paramétereit felsorolni. A `%0` a makró paramétereinek a száma, ennél eggyel kisebb a függvény paramétereinek a száma, amiket fordított sorrendben a verembe írunk.A `%rotate` elforgatja a paramétereket az adott számmal – ha a paramétere pozitív akkor balra, ha negatív akkor pedig jobbra. Végül meghívjuk a függvényt.

7.3. Makró-lokális címkék használata

Címkék használatánál a makróban vigyázni kell:

```
%macro buta 0
  jmp ide
ide:
%endmacro
```

Ezt a makrót csak egyszer lehet meghívni, ugyanis többszöri használatnál az assembler az `ide` címke többszöri definiálására panaszkodik! E helyett makró-lokális címkéket kell használni, amik `%%`-al kezdődnek. Fordítás közben ezek a címkék különböző konkrét példányokban jelennek meg.

```
%macro buta 0 ; többször használható
  jmp %%ide
%%ide:
%endmacro
```

7.4. A kontextus-verem

Néha szükség van arra, hogy több makró között megosszuk a címkéket, ilyenkor egy *kontextus*t kell elhelyezni a *kontextus-verem*ben és kontextus-lokális címkéket (amik `%$`-al kezdődnek) kell definiálni. Például egy többször használható, egymásba ágyazható `while-wend` ciklus a következőképpen definiálható:

```
%macro while 3
  %push while
%$loop_top:
  cmp %1,%3
  j%2 %$body
  jmp %$exit
%$body:
%endmacro

%macro wend 0
  jmp %$loop_top
%$exit:
  %pop
%endmacro
```

Használata például (amíg EAX=100h):

```
while eax,le,100h
      ; utasítások
wend
```

A kontextus-verem fordítási időben létezik, nem tévesztendő össze a veremszegmenssel!

7.5. Feltételes fordítás, változók az előfeldolgozás alatt

Előfordulhat, hogy az egysoros makrók szövegszerű behelyettesítése nem elég, ekkor használhatunk változókat is az előfeldolgozás alatt. Változót definiálhatunk, ha értéket adunk neki az `%assign` direktívával. Ennek segítségével saját magunk is létrehozhatunk egyedi címkéket úgy, hogy egy címkenévhez – legyen az hagyományos, makró-lokális vagy kontextus-lokális – hozzákonkatenálunk egy egyedi számot. A számot egy változóban tároljuk és ha új címkére van szükség, növeljük a változó értékét. Konkatenálni a `%+` direktívával tudunk, hamarosan erre is látunk példát.

Lehetőség van feltételes fordításra is az `%ifndef`, `%ifdef`, `%elifndef`, `%elifdef`, `%else`, `%endif` makrókkal. Például debug-módban (ha definiáljuk a DEBUG konstansot) speciális ellenőrzéseket végezhet a program:

```
%ifdef DEBUG
    ; utasítások
%endif
```

A következőt használjuk annak biztosítására, hogy egy include fájlt csak egyszer illesszünk be ugyanabba a programba.

```
%ifndef incfile
%define incfile
    ; utasítások
%endif
```

Hasonlóan a kontextusokra is lehetnek feltételek `%ifctx`, `%else`, `%endif`. A következő példa a fordítási idejű hiba jelzését is bemutatja.

```
%macro IF 3
 %push if
 %assign __curr 1
    cmp %1,%3
    j%2 %%if_code
    jmp %$loc %+ __curr
 %%if_code:
%endmacro

%macro ELSIF 3
 %ifctx if
    jmp %$end_if
  %$loc %+ __curr:
  %assign __curr __curr+1
    cmp %1,%3
    j%2 %%elsif_code
    jmp %$loc %+ __curr
  %%elsif_code:
 %else
```

```
    %error "'ELSIF' can only be used following 'IF'"
  %endif
 %endmacro

 %macro ELSE 0
  %ifctx if
     jmp %$end_if
   %$loc %+ __curr:
   %assign __curr __curr+1
  %else
   %error "'ELSE' can only be used following an 'IF'"
  %endif
 %endmacro

 %macro ENDIF 0
  %ifctx if
   %$loc %+ __curr:
   %$end_if:
   %pop
  %else
   %error "'ENDIF' can only be used following an 'IF'"
  %endif
 %endmacro
```

Használata értelemszerűen egy `IF`, nulla vagy több `ELSIF` opcionálisan egy `ELSE` majd végül egy `ENDIF` makró:

```
IF Value, Cond, Value
      ; utasítások
ELSIF Value, Cond, Value
      ; utasítások
ELSIF Value, Cond, Value
      ; utasítások
ELSE
      ; utasítások
ENDIF
```

A feltételekre is definiálhatunk makrókat, pl.:

```
%define EQUAL e
```

stb.

A `NASM` dokumentációban el lehet olvasni, hogyan lehet a makróknak alapértelmezett paramétereket adni, valamint a további makró-direktívák ismertetését is ott találhatjuk, pl. `%xdefine`, `%repl`, `%if`, `%ifmacro`, stb.

Feladat: Egészítsd ki a `wend` makrót ellenőrzéssel: ha nem „`while`" kontextusban hívjuk meg, adjon hibaüzenetet!

Feladat: Írj olyan `proc` (n paraméteres) és `endproc` (paraméter nélküli) makrókat, amelyek az eljárások elejét és végét generálják. A `proc` paraméterei az eljárás paramétereinek nevei, amikkel az eljárásban

hivatkozunk a paraméterekre. Az eljárás elején mentsük el az EBP-t és a végén állítsuk vissza, továbbá az ESP-t is a `mov esp,ebp`-vel. A paraméterek számát egy változóban tárolhatjuk, amit az `endproc` paraméterül ad a `ret`-nek.

Feladat: Írj egy `local` n paraméteres makrót, amely az eljárásban lefoglal n darab 32 bites lokális változót (ld. 6.3.4. szakasz), amelyekre a makrónak megadott paraméterekkel, mint nevekkel hivatkozhatunk az eljárásban. (Feltehetjük, hogy az eljárás végén a `mov esp,ebp` utasítással visszaállítjuk a verem tetejét, ezért `endlocal` makróra nincs szükség.

7.6. „Önmódosító" makrók

Mivel a makrók kódját minden makró-hívás helyére behelyettesíti az assembler, ezért a gyakori makróhívások nagyméretű tárgykódot eredményezhetnek. A Makro-assembler (`MASM`) lehetőséget nyújtott arra, hogy makrón belül újabb makrót definiáljunk, akár az eredeti makró felüldefiniálásával is, és így az önmódosító makrók segítségével az eljárások és a makrók előnyös tulajdonságait egyesítsük. A módszer a következő volt: a makró első hívásakor definiált egy eljárást, amit azonnal meg is hívott, illetve felüldefiniálta saját magát, hogy a további makró-hívásoknál már csak az eljáráshívást végezze el. Így ha a makrót egyáltalán nem használjuk, akkor a kódja bele sem kerül a tárgykódba, ha pedig többször használjuk, akkor is csak egyszer kerül bele egy eljárásként, amit több helyről hívhatunk.

A `NASM` nem engedi meg, hogy a többsoros makró-definíciókat egymásba ágyazzuk, de a feltételes fordítással elérhetjük ugyanezt a célt:

```
%macro clss 0
%ifndef clss_def
%define clss_def
  jmp skp
cls_sub:
      ; utasítások
  ret
skp:
  call cls_sub
%else
  call cls_sub
%endif
%endmacro
```

A makró a következőképpen működik: Az első meghívásnál a `clss_def` még nincs definiálva, így az if-ág kerül végrehajtásra, definiálja a `clss_def`-et és a `cls_sub` eljárást, amit rögtön meg is hív. A további hívásoknál csak az else-ág (az eljáráshívás) helyettesítődik be a kódba. Vegyük észre, hogy mivel az if-ág csak egyszer kerül bele a kódba, ezért nincs szükség makró-lokális `skp` címke használatára, sőt a `cls_sub` címke lokálissá tétele el is rontaná az eljáráshívást a többi makróhívásból.

Hasonló elven működik a következő makró, ami a szövegek konzolra írását segíti elő. Az első hívásnál lefoglal helyet a visszatérési értéknek, a többi hívásnál már ugyanezt használja. A makró arra is példa, hogy a szegmensek tartalmát tetszőleges sorrendben, akár részenként is definiálhatjuk, a `class` miatt az assembler ezeket egymás mellé helyezi a tárgykódban.

```
%macro print_str 1-*
  %ifndef print_def
    %define print_def
    segment bss use32 class=bss
      str_size  resd 1
```

```
  %endif
  segment data use32 class=data
    %%str
    %rep %0-1
      db %1
      %rotate 1
    %endrep
    db %1
    %%strhossz equ $-%%str
  segment code use32 class=code
    push LPVOID NULL
    push LPDWORD str_size
    push DWORD %%strhossz
    push LPCVOID %%str
    push HANDLE [hOut]
    call [WriteFile]
%endmacro
```

Használata például:

```
    print_str 'Hello',13,10,'World!'
```

8

Tömbök, struktúrák

Ebben a fejezetben néhány új utasítást ismerhetünk meg, amellyel a gép a tömbök kezelését könnyíti meg, majd megnézzük, hogyan segíti elő a `NASM` az összetettebb objektumok létrehozását és használatát.

8.1. String-kezelő utasítások

A processzor nyújt néhány utasítást tömbökkel való munkához, ezeket nevezik általában string-kezelő utasításoknak. Itt jut szerephez a D flag, ami a műveletek irányát fogja jelölni.

`lodsb`	`al = byte [ds:esi]` `esi ±= 1 (esi += 1-2*D)`
`stosb`	`byte [es:edi] = al` `edi ±= 1`

`movsb`	`byte [es:edi] = byte [ds:esi]`
	`esi ±= 1`
	`edi ±= 1`

`cmpsb`	`cmp byte [ds:esi], byte [es:edi]`
	`esi ±= 1`
	`edi ±= 1`

`scasb`	`cmp al,[es:edi]` `edi ±= 1`

14. táblázat. String-kezelő utasítások

A táblázatot úgy kell értelmezi, hogy például a `lodsb` utasítás az AL regiszterbe tölti a `[ds:esi]`-n található bájtot, majd egyel változtatja az ESI regiszter értékét. A változtatás növelés, ha D=0 és csökkentés, ha D=1. Az első három utasítás nem változtatja meg a flageket, a két utolsó pedig az összehasonlítás (`cmp`) szerint állítja be őket.

Ezeknek az utasításoknak nem csak bájtos, hanem wordös, doublewordös változataik is vannak (pl. `lodsw`, `lodsd`) értelemszerű módosításokkal (pl. a `lodsd` az EAX regiszterbe tölti a `[ds:esi]`-n található doublewordöt, majd néggyel változtatja az ESI értékét).

Stringkezelő utasítások a használatánál gyakran alkalmazunk *prefix*eket. A `rep` prefix addig csökkenti ECX-et és hajtja végre az utána következő utasítást, amíg ECX$\neq$ 0. A `repe`, `repz`, `repne`, `repnz` prefixek hasonlóak, de figyelembe veszik a Z flag értékét is.

A következő példában a mintaillesztés egy egyszerű algoritmusára láthatunk eljárást vermen keresztüli paraméterátadással:

```
; hívás: invoke search,s1,s2,s1_hossza,s2_hossza
;
; Megkeresi az s2 memóriacímen található vektor első előfordulását
; az s1 címen található vektorban. A visszatérési érték a keresett
; memóriacím vagy NULL.
;
search:
  mov ebp,esp                ; a verem tetejének eltárolása
  %define s1 dword [ss:ebp+4]      ; így fogunk hivatkozni a paraméterekre
  %define s2 dword [ss:ebp+8]
  %define s1_hossza dword [ss:ebp+12]
  %define s2_hossza dword [ss:ebp+16]
  mov ecx,s1_hossza   ; megkeressük az első egyező bájtot
  sub ecx,s2_hossza
  inc ecx
  push ds
  pop es
  cld
  mov edi,s1
  mov esi,s2
  mov al,[esi]
.c:
  repne scasb
  jne .nemtalalt
  push ecx            ; egyezik-e a két vektor
  push edi
  push esi
  mov ecx,s2_hossza
  dec edi
  repe cmpsb
  pop esi
  pop edi
  pop ecx
  jne .c
  mov eax,edi
  dec eax
  jmp .vege
.nemtalalt:
  mov eax,NULL
.vege:
  ret 4*4             ; visszatérés és a paraméterek kiszedése
                      ; a verembôl
  %undef s1
  %undef s2
  %undef s1_hossza
  %undef s2_hossza
```

Az eljárásban az ES regiszter értékét a vermen keresztül állítottam be, ugyanis a forrás- és céloperandus egyszerre nem lehet szegmensregiszter a `mov` utasításnál.

Az eljárás először megkeresi `s2` első karakterét `s1`-ben, majd összehasonlítja az a két stringet az adott

pozíciótól. Ilyenkor a már megtalált első karaktert is újból megvizsgálja (feleslegesen). Meg lehetne tenni, hogy csak a maradék stringet vizsgáljuk, ám ekkor vigyázni kell, ha a maradék hossza nulla (azaz az `s2` egy karakterből áll), ugyanis a prefixek is először csökkentik EAX-et és utána vizsgálják, hogy nulla-e.

8.2. Struktúrák

A `NASM` a struktúrák (egyes nyelveken rekordok) kezelését előre definiált makrókkal oldja meg. Például definiáljunk egy struktúrát a komplex számoknak:

```
struc complex
  .re resd 1
  .im resd 1
endstruc
```

Ez tulajdonképpen néhány szimbólumot definiál (pl. `complex.re`, `complex_size`, stb.). Inicializálatlan struktúrát a következő módon hozhatunk létre:

```
comp1 resb complex_size
```

Ez annyi bájtot foglal le, amekkora a struktúra mérete. Ha kezdeti értékekkel rendelkező rekordot akarunk létrehozni:

```
comp2 istruc complex
 at complex.re, dd 42
 at complex.im, dd 103
iend
```

Ami a $42 + 103i$ komplex számnak felelhet meg. A struktúra egy elemére a következő módon hivatkozhatunk:

```
mov eax,[comp2+complex.im]
```

Dátum és idő lekérdezése. A Windows alatt a `GetLocalTime` eljárás hívásával kérdezhetjük le az aktuális dátumot és időt. Az eredményt a következő struktúrában kapjuk vissza:

```
STRUC SYSTEMTIME
  .wYear       RESW 1
  .wMonth      RESW 1
  .wDayOfWeek  RESW 1
  .wDay        RESW 1
  .wHour       RESW 1
  .wMinute     RESW 1
  .wSecond     RESW 1
  .wMilliseconds RESW 1
ENDSTRUC
```

Minden adat két bájton van tárolva. Fontos szem előtt tartani azt is, hogy a hónapokat egytől számozzuk (1=január, stb.), míg a hét napjait nullától (0=vasárnap, 1=hétfő, stb.). Az alábbi példaprogram kiírja az aktuális dátumot:

```
%include "win32n.inc"
%include "macros.inc"
```

```
extern GetLocalTime
import GetLocalTime kernel32.dll
extern AllocConsole
import AllocConsole kernel32.dll
extern GetStdHandle
import GetStdHandle kernel32.dll
extern SetConsoleMode
import SetConsoleMode kernel32.dll
extern WriteFile
import WriteFile kernel32.dll
extern ReadFile
import ReadFile kernel32.dll
extern ExitProcess
import ExitProcess kernel32.dll

segment bss use32 class=bss
 datetime   resb SYSTEMTIME_size
 size resd 1

segment code use32 class=code

..start:
    init_console

    invoke [GetLocalTime],datetime

    print_str 'A mai datum: '
    movzx eax,word [datetime+SYSTEMTIME.wYear]
    invoke print_int,eax
    print_str '/'
    movzx eax,word [datetime+SYSTEMTIME.wMonth]
    invoke print_int,eax
    print_str '/'
    movzx eax,word [datetime+SYSTEMTIME.wDay]
    invoke print_int,eax

    read_key
    invoke [ExitProcess],NULL
```

Ahol a `print_int` a 6.3.2. szakaszban ismertetett függvény, csak sortörés nélkül, az `init_console` a szokásos konzolmegnyitás és handlerek lekérdezésére írt makró, míg a `read_key` egy makró, amely egy billetyűleütésre vár – ezek definíciója megtalálható a mellékelt példaprogramok között.

9

Fájl- és memóriakezelés

A következőkben megismerkedünk a Windows fájlkezelésének alapjaival, valamint a futásidejű memóriafoglalással.

9.1. Fájlkezelés

Fáljt megnyitni akár írásra, akár olvasásra a `CreateFile` eljárással lehet, amelynek paraméterei:

```
HANDLE CreateFile(
    LPCTSTR lpFileName,             // fájlnév
    DWORD dwDesiredAccess,          // hozzáférés módja
    DWORD dwShareMode,              // megosztás
    LPSECURITY_ATTRIBUTES lpSecurityAttributes, // biztosági attribútumok
    DWORD dwCreationDistribution,   // megnyitás módja
    DWORD dwFlagsAndAttributes,     // attribútumok
    HANDLE hTemplateFile            // template
   );
```

Ezek közül számunkra most a fájlnév a fontos, a hozzáférés módja (`GENERIC_READ` – olvasásra, `GENERIC_WRITE` – írásra) és a megnyitás módja (`CREATE_NEW` – új fájl létrehozása, `CREATE_ALWAYS` – új fájl létrehozása, ha már létezik, felülírja, `OPEN_EXISTING` – létező fájl megnyitása, `OPEN_ALWAYS` – fájl megnyitása, ha nem létezik, létrehozza, `TRUNCATE_EXISTING` – létező fájl megnyitása és felülírása). A visszatérési érték sikeres megnyitás esetén egy fájl handler, különben az `INVALID_HANDLE_VALUE`.

Ha a fájlok neveit nem akarjuk „bedrótozni” a programba, kiválaszthatjuk őket futás közben, erre szolgál a `BOOL GetOpenFileName(LPOPENFILENAME lpofn);` és a `BOOL GetSaveFileName(LPOPENFILENAME lpofn);` eljárás. Ha a felhasználó nem választ ki fájlt, a visszatérési érték nulla lesz. A paraméterként várt struktúra felépítése a következő:

```
typedef struct tagOFN { // ofn
    DWORD         lStructSize;
    HWND          hwndOwner;
    HINSTANCE     hInstance;
    LPCTSTR       lpstrFilter;
    LPTSTR        lpstrCustomFilter;
    DWORD         nMaxCustFilter;
    DWORD         nFilterIndex;
    LPTSTR        lpstrFile;
```

```
    DWORD         nMaxFile;
    LPTSTR        lpstrFileTitle;
    DWORD         nMaxFileTitle;
    LPCTSTR       lpstrInitialDir;
    LPCTSTR       lpstrTitle;
    DWORD         Flags;
    WORD          nFileOffset;
    WORD          nFileExtension;
    LPCTSTR       lpstrDefExt;
    DWORD         lCustData;
    LPOFNHOOKPROC lpfnHook;
    LPCTSTR       lpTemplateName;
} OPENFILENAME;
```

Ezek közül a legfontosabbak: `lStructSize` (a struktúra mérete), `lpstrFilter` (itt lehet beállítani, hogy milyen kiterjesztésű fájlokat ajánljon fel kiválasztásra), `lpstrFileTitle` (ide írja bele a kiválasztott fájl nevét), `nMaxFileTitle` (maximum ilyen hosszú lehet a fájlnév), `lpstrTitle` (ez lesz az ablak fejlécének szövege) és `Flags`. A `Flags` mezőbe számos paramétert beállíthatunk, amiknek az API leírásban utána lehet nézni, itt csak a példában előfordulókat említem meg: `OFN_EXPLORER` – az ablak explorer-stílusú legyen, `OFN_HIDEREADONLY` – csak az írható fájlokat jelenítse meg és `OFN_FILEMUSTEXIST` – a fájlnak léteznie kell, tehát egy nemlétező fájl nevének begépelését nem fogadja el.

A fájlba író és abból olvasó eljárásokkal már találkoztunk, hiszen a konzolt is ugyanezekkel lehet elérni:

```
BOOL ReadFile(
  HANDLE hFile,                  // fájl handler
  LPVOID lpBuffer,               // ide olvassa be a fájl tartalmát
  DWORD nNumberOfBytesToRead,    // ennyi bájtot olvas
  LPDWORD lpNumberOfBytesRead,   // ide írja, mennyit olvasott be
  LPOVERLAPPED lpOverlapped      // erre aszinkron esetben van szükség
);
```

és

```
BOOL WriteFile(
  HANDLE hFile,                  // fájl handler
  LPCVOID lpBuffer,              // ezt írja ki a fájlba
  DWORD nNumberOfBytesToWrite,   // ennyi bájtot ír ki
  LPDWORD lpNumberOfBytesWritten, // ide írja, mennyit írt ki
  LPOVERLAPPED lpOverlapped      // erre aszinkron esetben van szükség
);
```

A `DWORD GetFileSize(HANDLE hFile, LPDWORD lpFileSizeHigh);` az első paraméterként megkapott fájl méretét adja vissza. A második paraméterre csak akkor van szükség, ha a fájl mérete 4 gigabájtnál nagyobb.

Végül fájlt bezárni a `BOOL CloseHandle(HANDLE hObject);` eljárással lehet.

9.2. Dinamikus memóriakezelés

A programokban gyakran szükség van akkora memória használatára, aminek a méretét a programozó előre nem tudja. Ilyenkor futás közben kell a Windowstól memóriát kérni a következő eljárással:

```
HGLOBAL GlobalAlloc(
```

```
    UINT uFlags,    // paraméterek
    DWORD dwBytes   // igényelt memória mérete bájtokban
   );
```

Paraméterként általában a `GMEM_FIXED` opciót adjuk meg, ami nem mozgatható memóriát kér. Amennyiben nincs elég memória, az eljárás nulla értékkel tér vissza. A lefoglalt memóriát expliciten fel kell szabadítani a`GlobalFree` eljárással, ha már nincs rá szükség.

A következő példában megnyitunk egy fájlt, beolvassuk a tartalmát, megfordítjuk, majd kiírjuk egy másik fájlba. (Vigyázat: szövegfájl esetén az új sorokat jelző $0Dh$,$0Ah$ bájtok sorrendje is felcserélődik!)

```
%include "win32n.inc"
%include "macros.inc"

MAXSIZE equ 260

extern GetOpenFileNameA
import GetOpenFileNameA comdlg32.dll
extern CreateFileA
import CreateFileA kernel32.dll
extern GetFileSize
import GetFileSize kernel32.dll
extern GlobalAlloc
import GlobalAlloc kernel32.dll
extern ReadFile
import ReadFile kernel32.dll
extern WriteFile
import WriteFile kernel32.dll
extern GlobalFree
import GlobalFree kernel32.dll
extern CloseHandle
import CloseHandle kernel32.dll
extern ExitProcess
import ExitProcess kernel32.dll

segment data use32 class=data
 filterString   db "Text files (*.txt)",0,"*.txt",0
        db "All files (*.*)",0,"*.*",0,0
 title db 'Válassz egy fájlt',0

segment bss use32 class=bss
 ofn  resb OPENFILENAME_size
 outfilename db 'rev_'
 filename times MAXSIZE db 0

 hFile  resd 1
 fileSize resd 1
 pMemory  resd 1
 number resd 1

segment code use32 class=code
..start:
```

```
    mov dword [ofn+OPENFILENAME.lStructSize],OPENFILENAME_size
    mov dword [ofn+OPENFILENAME.hWndOwner],NULL
    mov dword [ofn+OPENFILENAME.hInstance],NULL
    mov dword [ofn+OPENFILENAME.lpstrFilter],filterString
    mov dword [ofn+OPENFILENAME.lpstrFileTitle],filename
    mov dword [ofn+OPENFILENAME.nMaxFileTitle],MAXSIZE
    mov dword [ofn+OPENFILENAME.lpstrTitle],title
    mov dword [ofn+OPENFILENAME.Flags], OFN_EXPLORER | OFN_HIDEREADONLY | \
      OFN_FILEMUSTEXIST
    invoke [GetOpenFileNameA], ofn
    or eax,eax
    jnz .tovabb
    jmp .vege

.tovabb:
    invoke [CreateFileA],filename,GENERIC_READ, \
       NULL,NULL,OPEN_EXISTING,NULL,NULL
    cmp eax,INVALID_HANDLE_VALUE
    jne .fileok
    jmp .vege
.fileok:
    mov [hFile],eax

    invoke [GetFileSize],[hFile],NULL
    mov [fileSize],eax

    invoke [GlobalAlloc],GMEM_FIXED,[fileSize]
    or eax,eax
    jne .memok
    jmp .vege
.memok:
    mov [pMemory],eax

    invoke [ReadFile],[hFile],[pMemory],[fileSize],number,NULL
    invoke [CloseHandle],[hFile]

    mov ecx,[fileSize]
    shr ecx,1
    or ecx,ecx
    jz .forditvege
    mov esi,[pMemory]
    mov edi,esi
    add edi,[fileSize]
    dec edi
.fordit:
    mov al,[esi]
    mov ah,[edi]
    mov [esi],ah
    mov [edi],al
    inc esi
```

```
    dec edi
    loop .fordit
.forditvege:

    invoke [CreateFileA],outfilename,GENERIC_WRITE, \
      NULL,NULL,CREATE_ALWAYS,NULL,NULL
    cmp eax,INVALID_HANDLE_VALUE
    je .vege
    mov [hFile],eax

    invoke [WriteFile],[hFile],[pMemory],[fileSize],number,NULL
    invoke [CloseHandle],[hFile]
    invoke [GlobalFree],[pMemory]

.vege:
    invoke [ExitProcess],NULL
```

A példában az input fájl neve a `filename` változóba kerül, az output fájl nevét pedig úgy kapjuk, hogy az input fájl nevét elől a „rev_” stringgel egészítjük ki, kihasználva, hogy ezek egymás után helyezkednek el a memóriában.

10

Kapcsolódás magasszintű nyelvhez

Most arra látunk példát, hogy hogyan lehet a program egy részét assemblyben, egy részét pedig C-ben[15] megírni. A következő példaprogram két szám legnagyobb közös osztóját számolja ki. Két lehetőséget is bemutat: assembly függvény hívását C-ből és C függvény hívását assemblyből. Vigyázat, a következő programban néhány dolog eltér az eddig megszokottól! Az `10_lnko_c.c` fájl a következőképpen néz ki:

```
main() {
    int a = lnko(12,28);
    printf("%d\n", a*a);
}
```

A programban az `lnko` eljárást hívjuk meg, a visszatérési értéket az `a` változóban tároljuk, majd kiírjuk ennek a négyzetét. Lássuk, hogyan valósítjuk meg az eljárást assemblyben:

```
;************************************************************************
; Program:  Legnagyobb közös osztó kiszámítása
; File:     10_lnko.asm
; Author:   Egri Péter "Pierre"
; Webpage:  http://fordprog.ini.hu
; Date:     17/10/2005
; Note:     Fordítás:
;               nasmw -f win32 10_lnko.asm
;               gcc -o 10_lnko.exe 10_lnko_c.c 10_lnko.obj
;************************************************************************
%include "macros.inc"
global _lnko
extern _printf

segment data use32 class=data
  writenum db '%d',10,0

segment code use32 class=code

;************************************************************************
; int lnko(int a, int b);
;************************************************************************
```

[15] A példaprogramot a szabadon elérhető DJGPP-vel fordítottam.

```
_lnko:
    push ebp
    mov ebp,esp
    %define a dword [ss:ebp+4+4]
    %define b dword [ss:ebp+4+8]
    mov eax,a
    mov ebx,b
.begin:
    cmp eax,ebx
    je .return
    ja .above
    sub ebx,eax
    jmp .begin
.above:
    sub eax,ebx
    jmp .begin
.return:
    push eax
    invoke _printf,writenum,eax
    add esp,8
    pop eax
    pop ebp
    ret
    %undef a
    %undef b
```

Első észrevétel, hogy a programot nem `obj`, hanem `win32` formátumú tárgykódra kell fordítani, mivel ezúttal nem az ALINK, hanem a DJGPP fogja a tárgykódokat összeszerkeszteni.

A `global _lnko` direktívával tehetjük az `_lnko` eljárásunkat külső modulból láthatóvá, ennek „ellentéte" az `extern _printf`, amivel a külső `_printf` eljárást tesszük használhatóvá. A `C` fordító tulajdonsága, hogy az eljárás- és változóneveket egy alulvonás (_) karakterrel kezdi, így nem keveredhetnek össze, ha assemblyben azonos nevű változókat használunk.

Ezután egy szöveget definiálunk, ami a `printf` első paramétere lesz és egy szám kiírását fogja végezni (%d). Fontos, hogy a \n helyett a 10 kódú karaktert kell használni soremelésre.

Az eljárásban **az eddig megismert *stdcall* helyett a `C` típusú (*cdecl*) eljáráshívás**hoz kell alkalmazkodni. Ami marad, az a paraméterek fordított sorrendben a verembe helyezése és a visszatérési érték az EAX (AX, AL – mérettől függően) regiszterben. Ami változik:

- Az EBP regiszter értékét kötelezően meg kell őrizni.
- A hívó program törli a paramétereket a veremből az eljárásból való visszatérés után.

Tehát ezért mentjük a verembe rögtön az EBP értékét és állítjuk vissza a visszatérés (`ret`) előtt. Így természetesen a paraméterek is lejjebb kerülnek a veremben, amire a hivatkozásnál figyelni kell.

A `printf` meghívásánál is hasonló a helyzet, így visszatérés után a paramétereket ki kell venni a veremből – ezt itt nem `push` utasításokkal tesszük, hanem a veremmutató átállításával, hiszen ez így gyorsabb és a paraméterek értékére nincs már szükségünk. A visszatéréskor az EAX-ben megmarad a legnagyobb közös osztó.

A példaprogram fordítása a `nasmw -f win32 10_lnko.asm` és `gcc -c 10_lnko_c.c` utasításokkal, összeszerkesztése pedig a `gcc -o 10_lnko.exe 10_lnko_c.o lnko.obj` utasítással történik. Az utóbbi kettő egybe is írható: `gcc -o 10_lnko.exe 10_lnko_c.c 10_lnko.obj`.

Feladat: Írj az `invoke`-hoz hasonló makrót, ami a *cdecl* eljáráshívást valósítja meg, tehát a visszatérés után korrigálja ESP értékét!

11

A koprocesszor használatának alapjai

A koprocesszor (matematikai társprocesszor) vagy más néven FPU (Floating Point Unit) nyújt eszközöket a lebegőpontos számok kezelésére. A 2.5.2. szakaszban megismertük az egyszeres és a dupla pontosságú lebegőpontos számok ábrázolását. Tulajdonképpen létezik egy harmadik fajta is, a 10 bájtos *kiterjesztett pontosságú lebegőpontos szám* és a koprocesszor mindent ebben a formában tárol. Amikor egyszeres és dupla pontosságú számokat adunk meg vagy kérünk le az FPU-tól, a konvertálás automatikusan megtörténik, ezért a továbbiakban a kiterjesztett pontosságú számokkal nem foglalkozunk.

Egy lebegőpontos számot exponenciális alakban kell megadni, pl. a `1.234567e20` szám értelmezése a következő: $1.234567 \cdot 10^{20}$. Ha egyszeres pontosságú (32 bites) változót definiálunk, azt a már ismert `dd` direktívával tudjuk megtenni, míg dupla pontosságúhoz (64 bit) egy új direktíva, a `dq` tartozik. Kezdőérték nélküli dupla pontosságú tömböt a `resq` direktívával hozhatunk létre és a változók használatánál a `qword` szóval fejezhetjük ki, hogy 64 bit a változó mérete.

A koprocesszorban (hasonlóan a CPU-hoz) vannak különböző regiszterek és állapotjelző flagek. Legfontosabb számunkra a nyolc lebegőpontos regiszter: ST0,...,ST7. A hagyományos regiszterekkel szemben ezek veremszerűen működnek, azaz nem lehet őket szabadon elérni, csak speciális utasításokkal számot helyezni a verembe, kivenni onnan vagy műveletet végrehajtani rajtuk. A debugger programokban általában be lehet kapcsolni, hogy az FPU regisztereinek értékére is kíváncsiak vagyunk. **A verem legfelső eleme mindig az ST0-ban van** és lebegőpontos értékű függvényeknél a visszatérési értéket is ST0 tartalmazza (EAX helyett).

Az utasításoknak sok változata létezik, pl. a következő utasítás mind kivonást jelent: `fsub`, `fsubr`, `fsubp`, `fsubrp`, `fisub`, `fisubr`. Ez elsőre nagyon bonyolultnak tűnhet, de az egyes betűknek mind megvan az értelmezése: „sub” – kivonást jelent, ld. 5.3. szakasz; „f” – float, azaz lebegőpontos utasítás (minden FPU utasítás ezzel kezdődik); „p” – pop, azaz az utasítás végrehajtása után a legfölső számot kidobja a veremből; „r” – reverse order, a nem kommutatív műveleteknél (kivonás, osztás) az operandusok fordított sorrendjét jelzi; „i” – integer, azaz lebegőpontos helyett 2-es komplemens értelmezést használ.

Az utasításoknak általában háromféle operandusa lehet: FPU regiszter (ezt a továbbiakban `reg`-gel jelölöm), memóriában tárolt adat[16] (`mem`) vagy a kettő közül bármelyik (amit egyszerűen `op`-pal jelölök). Néhány utasításnál az ST0 operandus opcionálisan elhagyható, mert egyértelmű a használata, ezt zárójellel jelölöm, pl. `faddp reg(,ST0)`. Ez tipikusan a pop-os utasításoknál fordul elő, mivel az a lehetőség, hogy kiszámítunk valamit az ST0-ban, majd rögtön ki is dobjuk a veremből értelmetlen.

A koprocesszor inicializálására a `finit` utasítás szolgál.

Nullával való osztásnál nem keletkezik kivétel, hanem az eredmény ∞ lesz, ld. 2.5.2.

Mint említettem a koprocesszor saját flagekkel rendelkezik és a 21. táblázatban található összehasonlítások – az utolsó kettő kivételével – ezeket a flageket állítják be. Sajnos a 5.5.3. szakaszban található

[16] Meglepő módon a változóknak való értékadáskor ugyanezt kell használni és nem a változó kell címét átadni!

`fld op`	lebegőpontos szám berakása a verembe
`fild mem`	egész szám berakása a verembe lebegőpontossá konvertálva
`fld1`	az 1 berakása a verembe
`fldz`	a 0 berakása a verembe
`fldpi`	a π berakása a verembe

15. táblázat. Betöltő utasítások

`fst op`	a verem tetejét átmásolja a memóriába vagy egy másik regiszterbe
`fstp op`	`fst` + pop
`fist mem`	a verem tetejét kerekítve átmásolja a memóriába (2-es komplemens alak)
`fistp op`	`fist` + pop

16. táblázat. Eltároló utasítások

`fadd op`	az ST0-hoz hozzáadja az operandust
`fadd reg,ST0`	a regiszterhez hozzáadja az ST0-t
`faddp reg(,ST0)`	`fadd reg,ST0` + pop
`fiadd mem`	az operandust lebegőpontossá konvertálja és hozzáadja az ST0-hoz

17. táblázat. Összeadó utasítások

`fsub op`	az ST0-ból levonja az operandust
`fsub reg,ST0`	a regiszterből levonja az ST0-t
`fsubr op`	az operandusból levonja az ST0-t és az eredményt az ST0-ban tárolja
`fsubr reg,ST0`	az ST0-ból levonja a regisztert és az eredményt a regiszterben tárolja
`fsubp reg(,ST0)`	`fsub reg,ST0` + pop
`fsubrp reg(,ST0)`	`fsubr reg,ST0` + pop
`fisub mem`	ST0-ból levonja a lebegőpontossá konvertált operandust
`fisubr mem`	a lebegőpontossá konvertált operandusból levonja az ST0-t és az eredményt az ST0-ban tárolja

18. táblázat. Kivonó utasítások

`fmul op`	az ST0-t megszorozza az operandussal
`fmul reg,ST0`	a regisztert megszorozza az ST0-val
`fmulp reg(,ST0)`	`fmul reg,ST0` + pop
`fimul mem`	az ST0-t megszorozza a lebegőpontossá konvertált operandussal

19. táblázat. Szorzó utasítások

feltételes ugrásoknak nincs meg a lebegőpontos flagekre vonatkozó változata, ezt megkerülendő, a FPU flagjeit át kell töltenünk az EFLAGS regiszterbe a következő módon:

```
fstsw ax
sahf
```

`fdiv op`	az ST0-t elosztja az operandussal
`fdiv reg,ST0`	a regisztert elosztja az ST0-val
`fdivr op`	az operandust elosztja az ST0-val és az eredményt az ST0-ban tárolja
`fdivr reg,ST0`	az ST0-t elosztja a regiszterrel és az eredményt a regiszterben tárolja
`fdivp reg(,ST0)`	`fdiv reg,ST0` + pop
`fdivrp reg(,ST0)`	`fdivr reg,ST0` + pop
`fidiv mem`	ST0-t elosztja a lebegőpontossá konvertált operandussal
`fidivr mem`	a lebegőpontossá konvertált operandust elosztja az ST0-val és az eredményt az ST0-ban tárolja

20. táblázat. Osztó utasítások

`fcom op`	ST0-t összehasonlítja az operandussal
`fcomp op`	`fcom op` + pop
`fcompp`	ST0 és ST1 összehasonlítása, majd mindkettő eltávolítása a veremből
`ficom mem`	ST0-t összehasonlítja a lebegőpontossá konvertált operandussal
`ficomp mem`	`ficom mem` + pop
`ftst`	ST0-t összehasonlítja nullával
`fcomi reg`	ST0-t összehasonlítja a regiszterrel
`fcomip reg`	`fcomi reg` + pop

21. táblázat. Összehasonlító utasítások

A `fstsw ax` utasítás az FPU flageket (status word) átmásolja az AX regiszterbe, majd ennek a felső bájtját betöltjük az EFLAGS megfelelő bitjeibe a `sahf` utasítással. Ezzel a bitek **úgy lesznek beállítva, mintha előjel nélküli összehasonlítás történt volna**, tehát az előjel nélküli feltételes ugróutasításokat (`ja`, `jb`,...) használjuk!

A 21. táblázat két utolsó utasítása közvetlenül az EFLAGS bitjeit állítja be – így a flagek áttöltögetését kihagyhatjuk –, viszont csak regiszterrel való összahasonlítást tesz lehetővé. Feltételes ugrásoknál szintén az előjel nélküli változatokat kell használni. Ebben a két utasításban az „i” betű nem egész értékre vonatkozik, ne keverjük őket össze a `ficom` és `ficomp` utasításokkal!

`fchs`	az ST0 előjelét változtatja meg
`fabs`	az ST0 előjelét pozitívra állítja (abszolút érték)
`fsqrt`	az ST0-ból gyököt von
`fsin`	az ST0 szinuszát számolja ki
`fcos`	az ST0 koszinuszát számolja ki
`fscale`	az ST0-t megszorozza $2^{\lfloor ST1 \rfloor}$-vel (kettőhatvánnyal való szorzás)
`frndint`	az ST0-t kerekíti
`fprem`	az ST0/ST1 osztás maradékát adja
`fxch reg`	az ST0 és a regiszter tartalmának felcserélése
`ffree reg`	a regiszter tartalmának kiürítése

22. táblázat. Egyéb utasítások

További utasítások a `NASM` és az Intel dokumentációkban találhatók.

A most következő egyszerű példaprogram az ismertetett alapokat mutatja be egy gömb térfogatának ($V = \frac{4\pi r^3}{3}$) kiszámolásán keresztül.

```
%include "win32n.inc"
%include "macros.inc"

extern ExitProcess
import ExitProcess kernel32.dll

segment data use32 class=data
 r dq 3.e0                        ; a gömb sugara
 harom dd 3
 negy dd 4

segment bss use32 class=bss
 V resq 1                          ; a gömb térfogata

segment code use32 class=code
..start:

  finit
  fld qword [r]                    ; ST0 = r
  fld st0                          ; ST0 = r; ST1 = r
  fmul st1,st0                     ; ST0 = r; ST1 = r*r
  fmulp st1,st0                    ; ST0 = r*r*r
  fldpi                            ; ST0 = pi; ST1 = r*r*r
  fmulp st1,st0                    ; ST0 = pi*r*r*r
  fimul dword [negy]               ; ST0 = 4*pi*r*r*r
  fidiv dword [harom]              ; ST0 = 4*pi*r*r*r/3
  fstp qword [V]

  invoke [ExitProcess],NULL
```

Feladat: Írj eljárást a másodfokú egyenlet (valós) megoldásainak kiszámítására!

12

Ablak létrehozása Windows alatt

Végül ízelítőként a Windows programozásának további lehetőségeiből, az ablak kezelésének alapjait ismerhetjük meg. Akit még behatóbban érdekel a téma, az számos gyakorlatias ismertetést találhat az Interneten (Win API, DirectX, DirectSound, stb.).

12.1. Window osztály létrehozása

Egy ablak létrehozásához először a `RegisterClassEx` eljárással egy osztályt kell regisztrálni. A paraméterként várt `WNDCLASSEX` struktúrában sokféle tulajdonságot állíthatunk be (pl. a menü felépítése, ikonok, egérkurzor), de a legalapvetőbbek az eseménykezelő eljárásunk címe: `lpfnWndProc` (az eseménykezelőt nekünk kell megírni), az osztály neve: `lpszClassName` (ezt tetszőlegesen választhatjuk), az ablak stílusa: `style` és a programunk handlere: `hInstance`, amit a `GetModuleHandleA` eljárással kérdezhetünk le.

12.2. Az ablak létrehozása

Ablakot létrehozni a `CreateWindowEx` eljárással tudunk. Paraméterként meg kell adni a létrehozott window osztályunkat, valamint egyéb jellemzőket, pl. magasság, szélesség, pozíció, stb. A létrehozott ablakot a `ShowWindow` eljárással jelentetjük meg. Az `UpdateWindow` eljárással egy kirajzolást kérő üzenetet küldünk az ablaknak (WM_PAINT, ld. később).

12.3. Várakozás az üzenetekre

A `GetMessage` eljárás addig várakozik, amíg egy üzenet nem érkezik. Amennyiben a visszatérési érték nulla, az azt jelenti, hogy kilépésre irányuló kérelem érkezett (WM_QUIT, ld. később), tehát ki kell lépni. Különben először át kell alakítani a szöveges üzeneteket (`TranslateMessage` eljárás), majd az továbbítani az eseménykezelőnek a `DispatchMessage` eljárással. Az üzenetek egy sorban várakoznak, ha gyorsabban érkeznek, mint ahogy feldolgoznánk őket – így nem vesznek el.

12.4. Eseménykezelő

Ezt az eljárást az operációs rendszer hívja meg, itt dolgozzuk fel az üzeneteket. A legfontosabb üzenetek a következők:

WM_CREATE: az ablak létrehozása után küldi a Windows.

WM_QUIT: programból való kilépés esetén.

WM_DESTROY: az ablak bezárása.

WM_PAINT: az ablak tartalmának kirajzolása.

WM_KEYDOWN, WM_KEYUP: billentyűleütés, illetve felengedés történt.

WM_MOUSEMOVE: egérkurzor helyváltoztatása és a gombok állapota.

WM_LBUTTONDOWN, WM_LBUTTONUP: bal egérgomb lenyomása, illetve felengedése.

WM_RBUTTONDOWN, WM_RBUTTONUP: jobb egérgomb lenyomása, illetve felengedése.

WM_ACTIVATE: az ablak fókuszba került.

WM_COMMAND: egy menü kiválasztása történt.

Ha WM_DESTROY üzenetet kapunk, a `PostQuitMessage` eljárást kell meghívni. Amennyiben egy üzenetet nem kívánunk feldolgozni, a `DefWindowProc` eljárással háríthatjuk át a feladatot a Windowsra.

12.5. Rajzolás az ablakba

A `WindowProc` eljárást (a nevét mi adjuk meg a `RegisterClassEx` eljárás hívásakor) a Windows hívja meg, valahányszor egy esemény történt az ablakunkban. Az eljárásnak négy paramétere van:

```
LRESULT CALLBACK WindowProc(
 HWND hwnd,      // az ablak azonosítója
 UINT uMsg,      // az üzenet azonosítója
 WPARAM wParam,  // az üzenet első paramétere
 LPARAM lParam   // az üzenet második paramétere
);
```

A rajzolást végző eljárásokat nem ismertetem, a Win32API helpben részletes leírás található róluk.

```
%include "win32n.inc"
%include "macros.inc"

extern GetModuleHandleA
import GetModuleHandleA kernel32.dll
extern LoadIconA
import LoadIconA user32.dll
extern LoadCursorA
import LoadCursorA user32.dll
extern RegisterClassExA
import RegisterClassExA user32.dll
extern CreateWindowExA
import CreateWindowExA user32.dll
extern ShowWindow
import ShowWindow user32.dll
extern UpdateWindow
import UpdateWindow user32.dll
extern GetMessageA
import GetMessageA user32.dll
extern TranslateMessage
import TranslateMessage user32.dll
extern DispatchMessageA
```

```
import DispatchMessageA user32.dll
extern PostQuitMessage
import PostQuitMessage user32.dll
extern DefWindowProcA
import DefWindowProcA user32.dll
extern BeginPaint
import BeginPaint user32.dll
extern GetClientRect
import GetClientRect user32.dll
extern MoveToEx
import MoveToEx gdi32.dll
extern LineTo
import LineTo gdi32.dll
extern EndPaint
import EndPaint user32.dll
extern ExitProcess
import ExitProcess kernel32.dll

segment data use32 class=data
 ClassName  db "Ablakosztaly",0     ; ez a window-osztályunk neve
 AppName    db "Háromszögek",0      ; fejléc
 seed       dw 412              ; random seed

segment bss use32 class=bss
  hInst     resd 1
  CommandLine   resd 1
  hwnd      resd 1
  wc    resb WNDCLASSEX_size
  msg   resb MSG_size
  hdc   resd 1
  ps    resb PAINTSTRUCT_size
  rect  resb RECT_size

segment code use32 class=code
..start:
    call WinMain
    invoke [ExitProcess],NULL

;************************************************************************
; Az ablak elôállítása
;************************************************************************
WinMain:
    mov dword [wc+WNDCLASSEX.cbSize],WNDCLASSEX_size
    mov dword [wc+WNDCLASSEX.style], CS_HREDRAW | CS_VREDRAW
    mov dword [wc+WNDCLASSEX.lpfnWndProc],WndProc
    mov dword [wc+WNDCLASSEX.cbClsExtra],NULL
    mov dword [wc+WNDCLASSEX.cbWndExtra],NULL
    invoke [GetModuleHandleA],NULL
    mov [hInst],eax
    push eax
```

```
    pop dword [wc+WNDCLASSEX.hInstance]
    mov dword [wc+WNDCLASSEX.hbrBackground],COLOR_WINDOW+1
    mov dword [wc+WNDCLASSEX.lpszMenuName],NULL
    mov dword [wc+WNDCLASSEX.lpszClassName],ClassName
    invoke [LoadIconA],NULL,IDI_APPLICATION
    mov [wc+WNDCLASSEX.hIcon],eax
    mov [wc+WNDCLASSEX.hIconSm],eax
    invoke [LoadCursorA],NULL,IDC_ARROW
    mov [wc+WNDCLASSEX.hCursor],eax
    invoke [RegisterClassExA],wc

    invoke [CreateWindowExA],NULL,ClassName,AppName,WS_OVERLAPPEDWINDOW, \
          CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,NULL,NULL,hInst,NULL
    mov [hwnd],eax

    invoke [ShowWindow],[hwnd],SW_SHOWDEFAULT
    invoke [UpdateWindow],[hwnd]

.msgloop:               ; egy ciklusban várjuk a felhasználó reakcióját
    invoke [GetMessageA],msg,NULL,NULL,NULL
    or eax,eax
    jz .endmsgloop
    invoke [TranslateMessage],msg
    invoke [DispatchMessageA],msg
    jmp .msgloop
.endmsgloop:
    mov eax,[msg+MSG.wParam]
    ret

;*************************************************************************
; Eseménykezelô
;*************************************************************************
WndProc:
      mov ebp,esp
      %define hWndParam dword [ss:ebp+4]
      %define uMsgParam dword [ss:ebp+8]
      %define wParam dword [ss:ebp+12]
      %define lParam dword [ss:ebp+16]

    cmp dword uMsgParam,WM_DESTROY     ; ha be akarja zárni az ablakot, ám legyen
    je .destroy
    cmp dword uMsgParam,WM_PAINT       ; ki kell rajzolni az ablak tartalmát
    je .paint
                                          ; különben a Windows kezelje az eseményt
    invoke [DefWindowProcA],hWndParam,uMsgParam,wParam,lParam
    ret 4*4

.destroy:               ; ez itt az ablak bezárása
    invoke [PostQuitMessage],NULL
    xor eax,eax
```

```
    ret 4*4

.paint:                 ; kirajzolás
    invoke [BeginPaint],hWndParam,ps
    mov dword [hdc],eax

    invoke [GetClientRect],hWndParam,rect

; A háromszögek kirajzolása a következô elven működik: elindulunk az
; ablak bal felsô sarkából és minden lépésben választunk egy sarkot a
; bal felsô, bal alsó és jobb alsó sarkok közül. Az aktuális pont és
; a kiválasztott pont közötti felezôpontra ugrunk és kirajzolunk oda
; egy pontot. Az aktuális koordinátát az (ebx,edx) tárolja.

    mov ebx,dword [rect+RECT.right]
    sub ebx,dword [rect+RECT.left]
    mov edx,dword [rect+RECT.bottom]
    sub edx,dword [rect+RECT.top]
    mov ecx,0FFFFh

.rajzciklus:
    cmp word [seed],21845
    jb .balfelso
    cmp word [seed],43690
    jb .balalso
    jmp .jobbalso
.balfelso:
    sub ebx,dword [rect+RECT.left]
    shr ebx,1
    add ebx,dword [rect+RECT.left]
    sub edx,dword [rect+RECT.top]
    shr edx,1
    add edx,dword [rect+RECT.top]
    jmp .rajzol
.balalso:
    sub ebx,dword [rect+RECT.left]
    shr ebx,1
    add ebx,dword [rect+RECT.left]
    mov eax,dword [rect+RECT.bottom]
    sub eax,edx
    shr eax,1
    add edx,eax
    jmp .rajzol
.jobbalso:
    mov eax,dword [rect+RECT.right]
    sub eax,ebx
    shr eax,1
    add ebx,eax
    mov eax,dword [rect+RECT.bottom]
    sub eax,edx
```

```
    shr eax,1
    add edx,eax
.rajzol:
    push ecx               ; elmentjük a regisztereket
    push edx
    push ebx

    invoke [MoveToEx],[hdc],ebx,edx,NULL
    pop ebx
    pop edx
    push edx
    push ebx
    inc edx
    inc ebx
    invoke [LineTo],[hdc],ebx,edx    ; draw point
    call randomize

    pop ebx                ; visszatöltjük a regisztereket
    pop edx
    pop ecx

    dec ecx
    or ecx,ecx
    jz .rajzvege
    jmp .rajzciklus
.rajzvege:
    invoke [EndPaint],hWndParam,ps
    xor eax,eax
    ret 4*4
    %undef hWndParam
    %undef uMsgParam
    %undef wParam
    %undef lParam

;************************************************************************
; Pseudo-random szám
;************************************************************************
randomize:
    mov ax,word [seed]
    mov bx,9821
    imul bx
    inc ax
    ror al,1
    rol ah,1
    mov word [seed],ax
    ret
```

Tárgymutató

Sz

T

U, Ú

Ü, Ű

V

W