

Assembly programozás

levelező tagozat

Varga László

Képfeldolgozás és Számítógépes Grafika Tanszék
Szegedi Tudományegyetem
(Tanács Attila nappali előadás fóliái alapján)

Elérhetőségek, anyagok

- **Személyesen**

- Előadás időpontjában
- Fogadóórán (Árpád tér 2, tetőtér 218)
 - Szerda 14-15
 - Péntek 9-10

- **Elektronikus formában**

- `vargalg@inf.u-szeged.hu`

- **Kurzus anyagok**

- `http://www.inf.u-szeged.hu/~vargalg`
- Coospace

Követelményrendszer

- **Gyakorlat (kivonat)**
 - „Táblás” gyakorlatok + számítógép használat
 - Füzet, toll szükséges!
 - 1 alkalommal „nagy ZH”, 40 pont
 - Utolsó gyakorlaton
 - A ZH 1 alkalommal javítható/pótolható
 - A vizsgaidőszak első hetében közösen
 - Otthoni (beadandó) feladat, 10-15 pont
 - Félév végén meg kell védeni

Gyakorlati teljesítés

- **Beadandó feladat**

- Beadható feladat (otthoni munka)
- Opcionális, de utólagos reklamációt nem fogadunk el.
- A választható feladatok várhatóan 2016. március 13-ig kerülnek fel a Coospace-re.
- 1 feladatra csoportonként maximum 2 hallgató jelentkezhet, a jelentkezést a Coospace kezeli.
- Feladatra jelentkezni 2016. április 24-ig lehet
- Megoldások beküldhetők 2016. május 8-ig

Gyakorlati teljesítés

- **Beadandó feladat**
 - Tetszőleges segédeszköz használható, de érteni kell a program működésének minden részét!
 - A megoldásokat meg kell védeni a szorgalmi időszak utolsó hetében.
 - **0-15 pont** szerezhető így.

Követelményrendszer

- **Gyakorlat (folytatás)**

- A félév végén gyakorlati jegy az összpontszám alapján
- Kerekítés nincs!

Összpontszám	Érdemjegy
Ha az elért pontszámok összege < 25	elégtelen (1)
Ha az elért pontszámok összege < 31 , de ≥ 25	elégséges (2)
Ha az elért pontszámok összege < 37 , de ≥ 31	közepes (3)
Ha az elért pontszámok összege < 43 , de ≥ 37	jó (4)
Ha az elért pontszámok összege ≥ 43	jeles (5)

Követelményrendszer

- **Kollokviumi vizsga**

- Előfeltétele a sikeres gyakorlati teljesítés
- 10 kérdésből álló teszt
 - $2 \times 10 = 20$ pont
 - 20 perc munkaidő
 - Rövid, tömör, lényegretörő válaszok!
- Kiadott tételjegyzékből 2 tétel kidolgozása
 - $2 \times 15 = 30$ pont
 - Esszé
- Összesen 50 pont szereshető
- Semmilyen segédeszköz nem használható!

Követelményrendszer

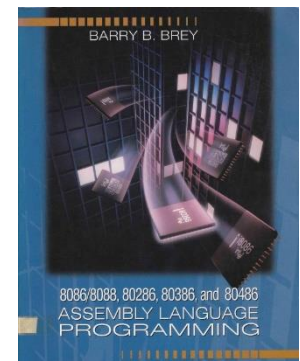
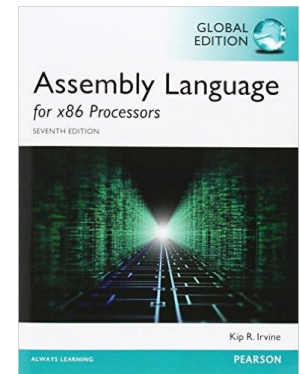
- **Jegy megállapítása**

- A teszten minimálisan 6 pontot; a tételekből minimálisan 5-5 pontot el kell érni! Ha ez nem teljesül, akkor a vizsga eredménye **elégtelen (1)**.
- Ha teljesül, akkor összpontszám alapján:

Összpontszám	Érdemjegy
Ha az elért pontszámok összege < 25	elégtelen (1)
Ha az elért pontszámok összege < 31, de ≥ 25	elégséges (2)
Ha az elért pontszámok összege < 37, de ≥ 31	közepes (3)
Ha az elért pontszámok összege < 43, de ≥ 37	jó (4)
Ha az elért pontszámok összege ≥ 43	jeles (5)

Ajánlott szakirodalom

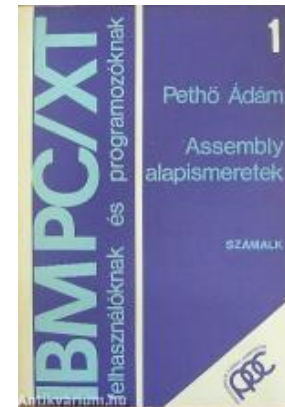
- **Előadások és gyakorlatok anyaga**
- S. Tanenbaum: **Számítógép-architektúrák**, 2. átdolgozott, bővített kiadás (Panem, 2006)
 - Csak a könyv 5. és 7. fejezete
 - Áttekintés
- Kip R. Irvine: **Assembly Language for x86 Processors**, 7th edition (Pearson, 2015)
 - Angol nyelvű, friss, alapos leírás
 - Központi könyvtárban 2 példányban elérhető
- Barry B. Brey: **8086/8088, 80286, 80386 and 80486 Assembly Language Programming** (Prentice Hall, 1993)
 - Angol nyelvű, viszonylag régi referenciakönyv



(Kevésbé) Használható szakirodalom

Magyar nyelven csak a régi, 16 bites MS-DOS operációs rendszerhez kapcsolódó Assembly programozásról találunk anyagot. Ezek csak korlátozottan használhatók a felkészüléshez!

- Pethő Ádám: **IBM PC/XT felhasználóknak és programozóknak 1. Assembly programozás** (SZÁMALK, 1992)
- Agárdi Gábor: **IBM PC gyakorlati Assembly** (LSI Oktatóközpont, 2002)
- Máté Eörs: **Assembly programozás** (NOVADAT, 1999, 2000)



Vázlatos tematika

- **Bevezetés, áttekintés**
 - Assembly alapfogalmak. Assembly nyelv előnyei, hátrányai, alkalmazási lehetőségei.
- **x86 memóriamodell, címzési módok**
 - Az x86 memória modellje és regiszterkészlete.
 - Adat- és kódterület címzése.
- **x86 utasításrendszer**
 - Aritmetikai, adatmozgató, logikai utasítások.
 - Vezérlésátadás, eljárás-hívás, ciklusszervezés.
 - Sztring műveletek, REP prefixumok.
- **Assembly programozási technikák**
 - Paraméterátadási lehetőségek eljárás-híváskor: regiszterekben, vermen keresztül.
 - Rekurzív és re-entráns eljárások, eljárásra lokális adatterület.
 - Rendszerhívások.

Eszközök

- **Gyakorlatokon**
 - Toll, papír, tábla, számítógép
 - Microsoft Visual Studio 2015 (32 bites programozáshoz)
- **Otthon**
 - MASM32 telepítése
 - Régebbi MASM fordítóra épül, de kompatibilis
 - <http://www.masm32.com/>
 - Kip Irvine könyvéhez tartozó függvénykönyvtár telepítése
 - <http://kipirvine.com/asm/examples/index.htm>
 - <http://kipirvine.com/asm/>
 - Microsoft Visual Studio
 - Ingyenes és régebbi változatokkal is működik
 - DosBox (16 bites programozáshoz)
 - Ingyenes, multiplatform DOS emuláció
 - Macro Assembler (rég), Borland Turbo Assembler, Debugger

Bevezetés, ismétlés

Gépi, nyelvi szintek

5. Probléma orientált nyelv szintje
fordítás (fordító program)
- 4. Assembly nyelv szintje**
fordítás (assembler)
3. Operációs rendszer szintje
részen értelmezés (operációs rendszer)
- 2. Gépi utasítás szintje**
ha van mikroprogram, akkor értelmezés
(Kompatibilitás!)
1. Mikroarchitektúra szintje
hardver
0. Digitális logika szintje

Modern asztali számítógép

- **Neumann-elvű gép**
 - Központi feldolgozó egység (**CPU**)
 - **Operatív memória** az adatok és a programok futás közbeni tárolására
 - Egységek közötti **sínrendszer** a kommunikációhoz
 - **Bementi/kimeneti rendszer** a felhasználóval való kapcsolattartáshoz
 - Működést biztosító **járolékos egységek**
 - Tápellátás, ...

Operatív memória

- **Felépítése**

- Alapja a **bit**: 0 vagy 1 érték
- Bitek rendszerint csoportosítva kerülnek feldolgozásra
 - Cella: legkisebb címezhető egység
 - **8 bit = 1 bájt** (Oka: ASCII 7 bites karakterkód + 1 paritás)
 - Lehetnek más, pl. 4, 16 vagy más csoportosítások is!
- A memória bájtok sorozata
- Bájtok elérése a ***címükkel***

Operatív memória

- **Bájtok értékeinek értelmezése**
 - Lehet **adat és program** is!
 - Adat: ASCII, UTF, BCD, kettes komplementum, lebegőpontos, ...
 - Gépi kód: a számok utasításokat jelentenek
 - A CPU csak ezeket az utasításokat tudja végrehajtani!
 - Magasabb szintű programozási nyelvekről gépi kódra kell fordítani.
 - Akár **önmódosító program** is készíthető
 - Veszélyes!
 - Hibás működés, ha adatrészre kerül a vezérlés!
 - A mai modern operációs rendszerek védik a kódterületet
 - Ez elősegíti a virtuális memória hatékonyabb kezelését is

Memóriatérkép

- **Memória felosztása**

- Nagy része szabadon használható terület
- Bizonyos címterületek a hardverrel való kapcsolattartásra vannak fenntartva
 - Pl. kijelző, merevlemez, külső meghajtók
- Bizonyos címek meghatározhatják az egyes címterületek tartalmát
 - Pl. RAM (írható/olvasható) vagy ROM (olvasható) legyen ott elérhető
 - Átlapolási lehetőség a címtartományban

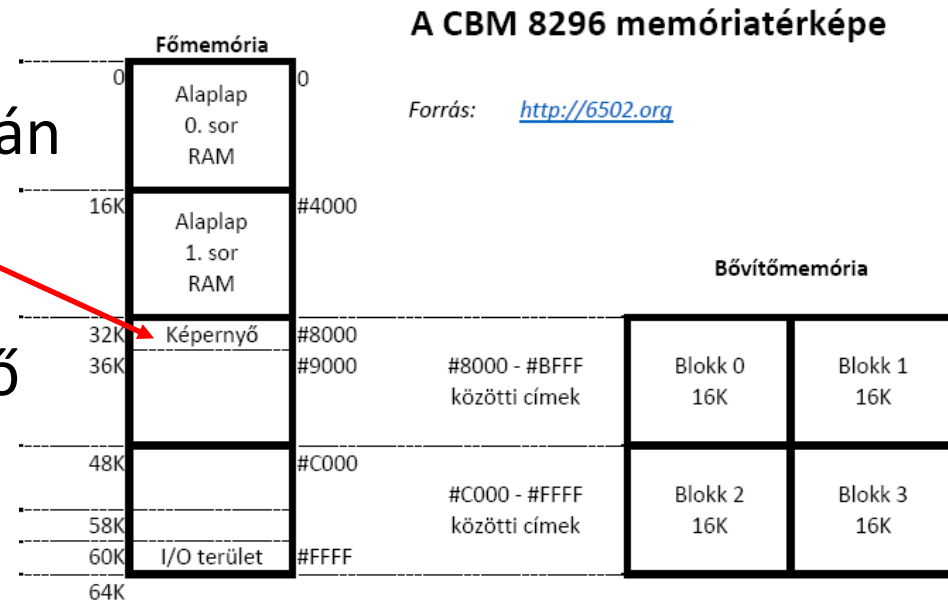
- **Mérete (PC-ken)**

- Korábban: pár kilobájt, megabájt
 - Pl. A Commodore 64 gép 64 kilobájtot ért el, ez megfelel egy 256x256 méretű szürkeárnyaltos kép mátrixának!
- Manapság: több gigabájt

Központi memória

- **Példa CBM 8296 esetén**

- 64 kilobájt címezhető
- Szöveges képernyő tartalma írható memórián keresztül
- A felső 32 KB memóriacímen vagy a fő memória, vagy a bővítő memóriamodulok tartalma jelenik meg
 - Kapcsolóval állítható
 - Kapcsoló = dedikált memóriacím tartalma



Központi feldolgozó egység (CPU)

- **Feladata**

- A programszámláló által mutatott memóriacímen lévő utasítást végrehajtja
- A mutató továbblép a következő utasításra, vagy ugró utasítás esetén a megadott címre

- **Regiszterek**

- **Nagyon gyors elérésű tárolóegységek**
- Általában 1 gépi szó hosszúságúak (8, 16, 32 vagy 64-bit)
- Áramköri vagy RAM megvalósítás
- Általános vagy dedikált regiszter
 - Aritmetikai műveletre, memória címzésére
 - Címregiszter, állapotregiszterek, ...
- Minél több van, annál jobb
 - A memória elérése sokkal lassabb!

Számítógép (PC) működési vázlat

- Bekapcsolás
- Programszámláló a BIOS EPROM-ban található gépi kódú rendszerbetöltő programjának az elejére (rögzített cím)
- A bájtt értékek által definiált utasítások végrehajtása
 - Rendszerteszt
 - Rendszerbetöltő megkeresi a rendszerindító egységet (pl. merevlemez) és annak betöltő programjára „ugrik” (rögzített helyen van)
 - Az betölti az operációs rendszert
 - Az operációs rendszer
 - Tartja a kapcsolatot felhasználóval,
 - Ütemezi a processzusokat,
 - Kezeli az erőforrásokat
 - ...

Fogalmak

- **Gépi kód**

- Numerikus gépi nyelv
- Az utasítások és az operandusok számok
- 1 utasítás 1 vagy több bajton kódolódik
- Elemi műveletek végrehajtására

Kód forrása: [Thinking Soul](#)

```
*****  
* FUNCTION: INITA - Initialize ACIA  
* INPUT: none  
* OUTPUT: none  
* CALLS: none  
* DESTROYS: acc A
```

- **Assembly nyelv**

- A numerikus gépi nyelv szimbolikus formája
- **Mnemonik**
 - emlékeztető kód a numerikus utasítások helyett
- Szimbolikus nevek és címek
- Makrók
- Feltételes fordítás
- ...

0013		RESETA EQU %00010011	
0011		CTLREG EQU %00010001	
C003	86 13	INITA LDA A #RESETA	RESET ACIA
C005	B7 80 04	STA A ACIA	
C008	86 11	LDA A #CTLREG	SET 8 BITS AND 2 STOP
C00A	B7 80 04	STA A ACIA	
C00D	7E C0 F1	JMP SIGNON	GO TO START OF MONITOR

Cím Gépi kód Szimbólumok + mnemonikok Megjegyzés
(hexa-
decimális
számok)

Fogalmak

- **Assembler**
 - A fordító, amely assembly nyelvről gépi kódra fordít
- **Disassembler**
 - Gépi kódból mnemonik kód listázása
 - Vigyázni kell, hogy a listázás kezdőcíme valóban utasításhatáron legyen!
 - Ha szimbólumlista elérhető, akkor Assembly-szerű lista kapható

Assembly nyelv

- **Jellemzők**

- Minden utasításnak egyetlen gépi utasítás felel meg
- Architektúránként különböző Assembly nyelv!
 - Pl. Intel, UltraSPARC, RISC-alapú architektúrák
 - Nincs a magasszintű nyelveknél tapasztalható portabilitás!

Assembly

- **Hátrányok**

- Nehézkes, időigényes, sok hibalehetőség
- Hosszadalmasabb hibakeresés, karbantartás
- Architektúrák közötti portabilitás hiánya

- **Előnyök**

- Hatékonyság
- A hardver teljes elérhetősége
 - Bizonyos regiszterek magasszintű nyelvekből nem használhatók
 - Hardver elemek közvetlen vezérlése
 - Játékprogramok, eszközmeghajtók, ...

Fordítás, szerkesztés

- **Fordító (Compiler)**

- *Forrásnyelvből célnyelvre alakít, pl.:*

- C++, C -> tárgykód (.o/.obj)

- **Assembly -> tárgykód**

Assembler

- Java -> class fájl
 - C -> Assembly (.asm)
 - FORTRAN -> C

- **Szerkesztő (Linker)**

- *Tárgykódok összeszerkesztése futtatható állománnyá*

- Külső hivatkozások feloldása
 - Virtuális címek feloldása
 - Címterek összefésülése (relokáció)
 - ...

Fordítás, szerkesztés

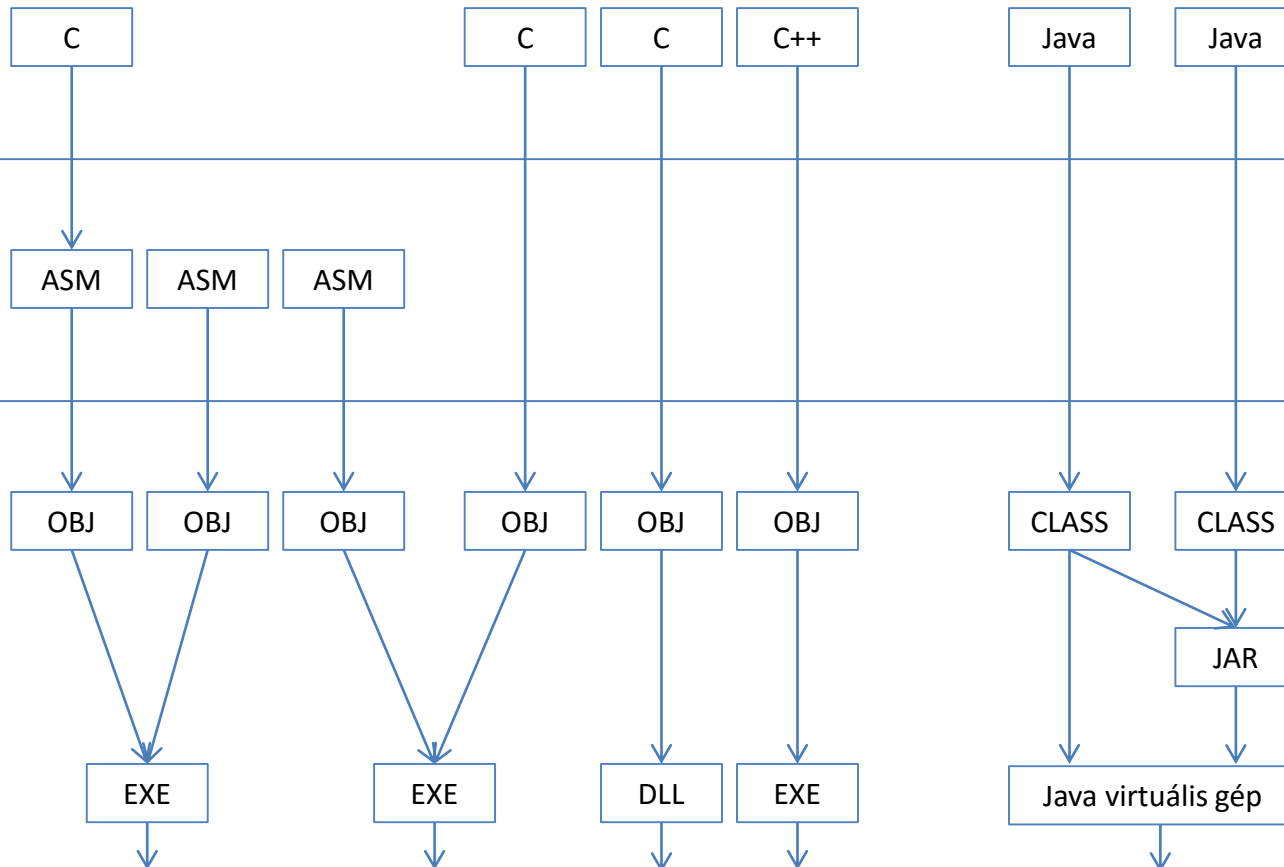
- **Fájlformátumok, kiterjesztések**

- **Tárgykód**

- .obj: DOS, Windows
- .o: Unix, Linux, Cygwin, ...
- Fordító-specifikus formátum!
 - Azonos kiterjesztés még nem jelenti azt, hogy összeszerkeszthetők!

- **Végrehajtható (futtatható)**

- .com: kis méretű DOS alkalmazások
- .exe: DOS, Windows formátum (többféle szerkezet!)
- .dll: dinamikusan szerkeszthető (Windows)
- Unix-alapú rendszereknél nincs futtatható kiterjesztés, azt a fájl jogosultsága mondja meg
- .so: dinamikusan szerkeszthető (Unix, Linux, ...)
- .a: statikusan szerkeszthető tárgykód gyűjtemény (Unix, Linux...)



Magasszintű, probléma orientált nyelvek szintje

Assembly nyelv szintje

Gépi nyelv szintje

Tárgykód

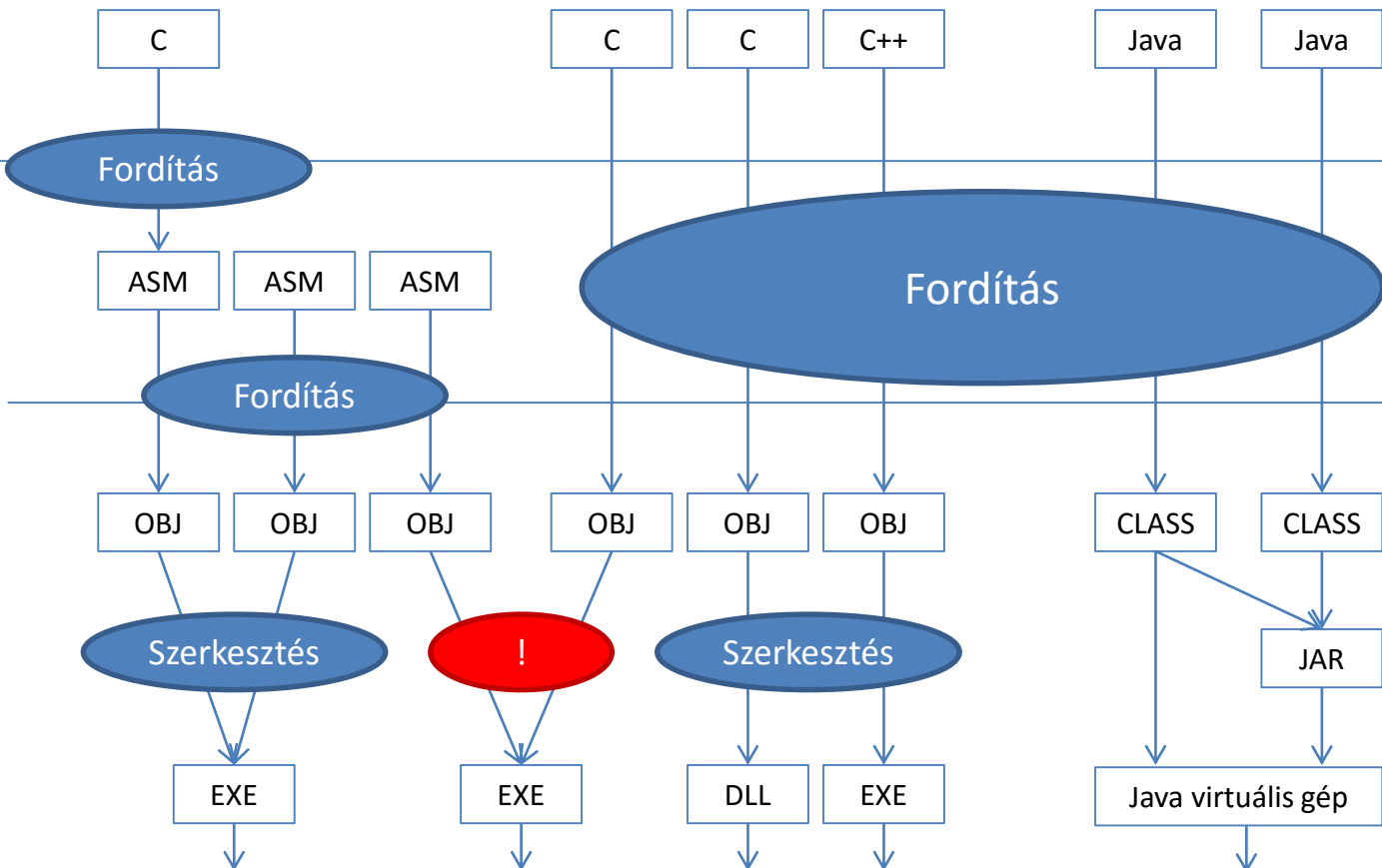
Végrehajtható program vagy dinamikusan szerkeszthető könyvtár

Operációs rendszer
 Processzusok ütemezése, erőforrások kezelése, fájlkezelés, hardver absztrakciós szint, ...

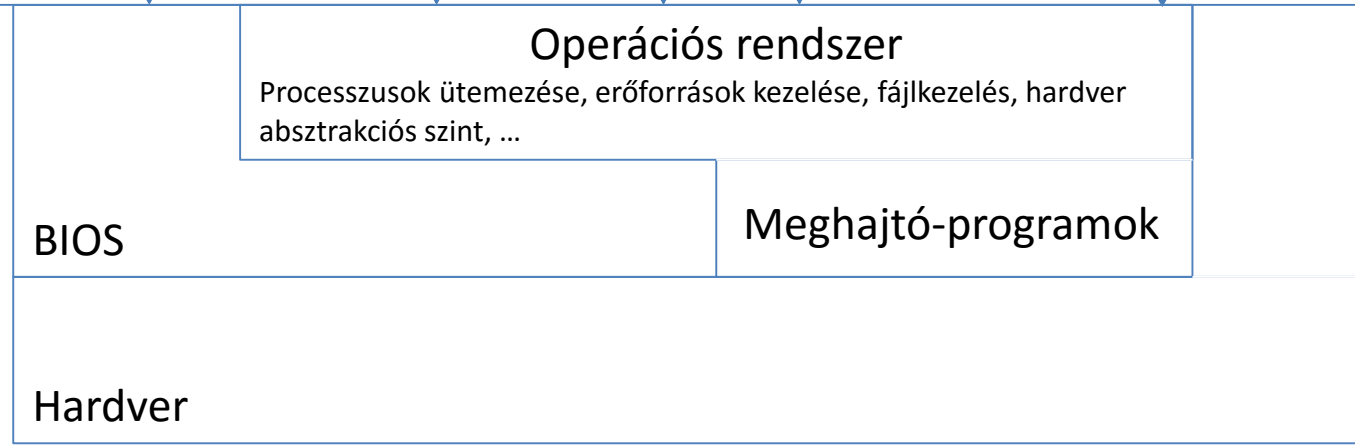
BIOS

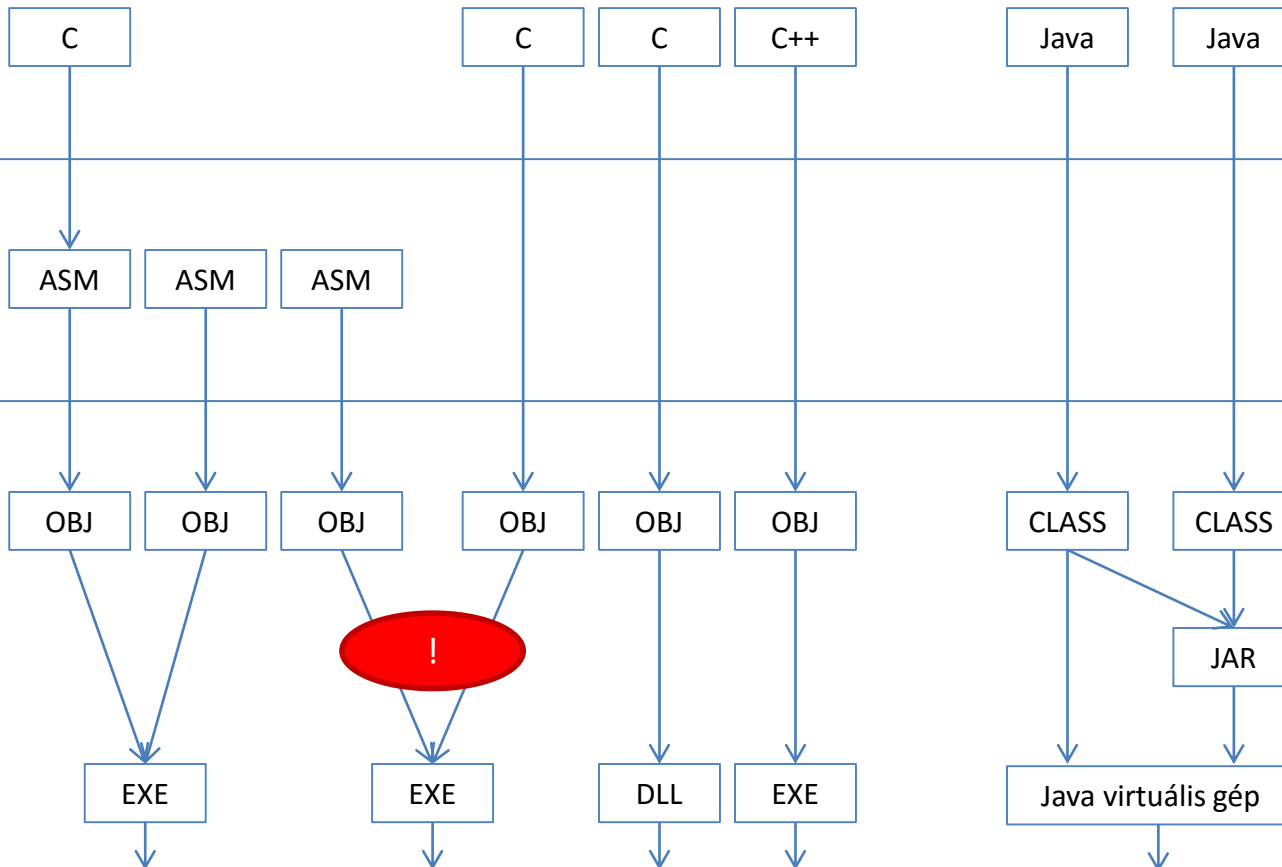
Meghajtó-programok

Hardver



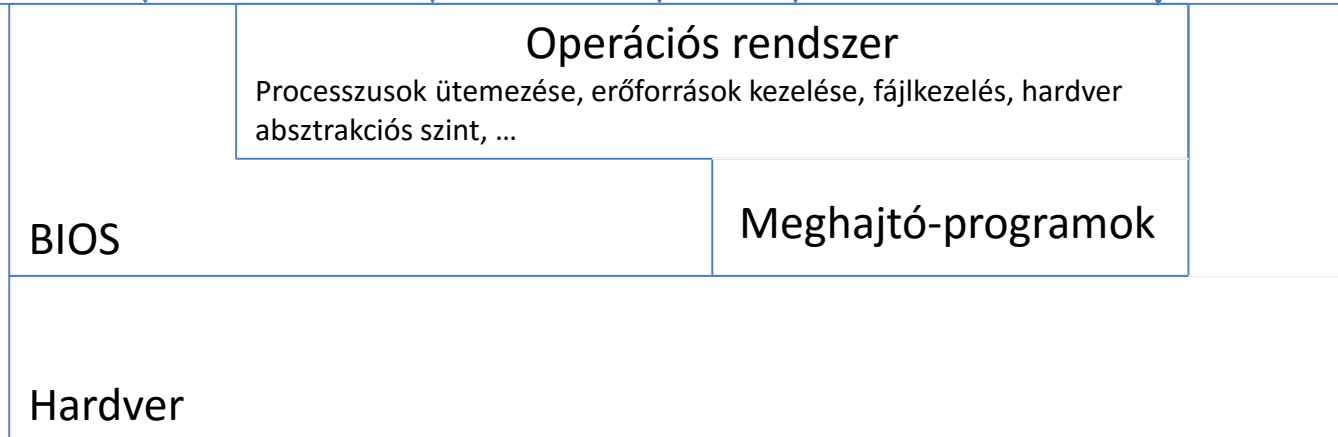
Különböző forrásnyelvekből származó tárgykódok akkor szerkeszthetők össze, amennyiben megegyező formátumúak, valamint egyeznek a hívási konvenciók, paraméterátadások, a memóriamodell, ...





Assembly:
hardverközeli és/vagy hatékony implementációt igénylő részek

Magas szintű nyelv:
GUI, vezérlés, Assembly betétek hívása



Hardver elérése

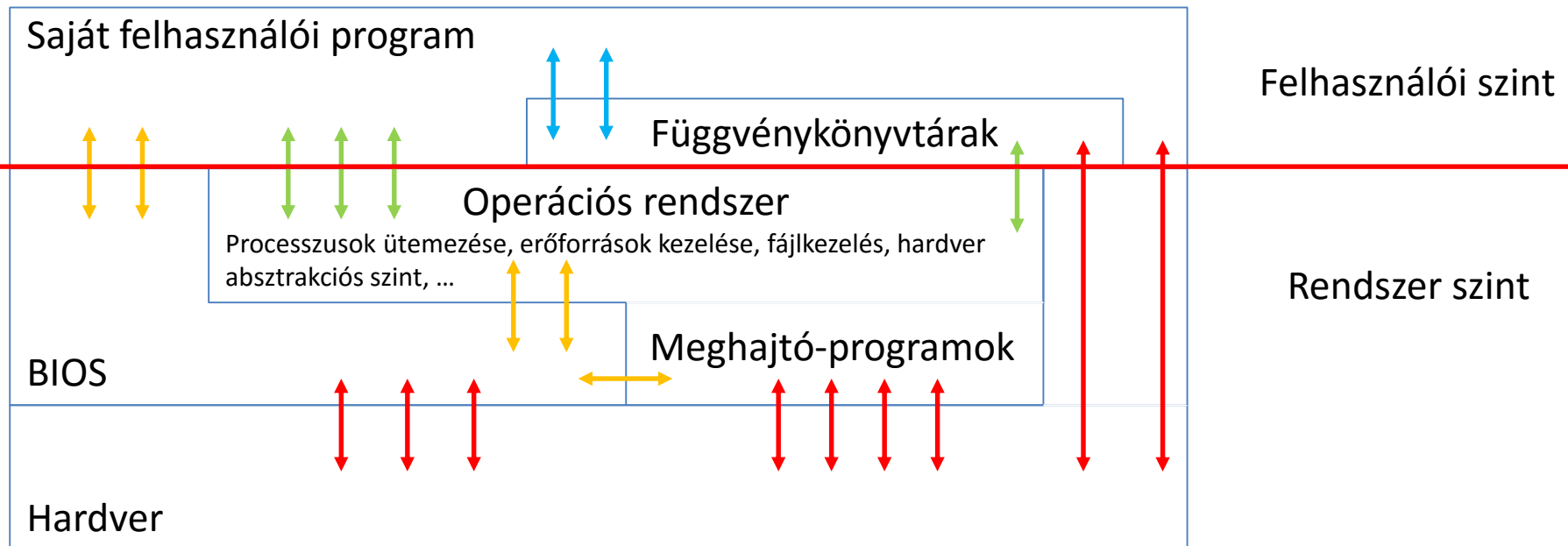
- **Közvetlenül**
 - B/K kapuk írása/olvasása
 - Közvetlenül a hardverrel kommunikálunk
 - A mai modern operációs rendszerek ezt tilt(hat)ják
 - Előnye a hatékonyság (pl. játékprogramok konzolokra)
 - Hátránya, hogy hardver eszközként/gyártónként más kommunikáció szükséges!
- **BIOS rendszeren keresztül**
 - BIOS gyártó specifikus lehet, de egységesebb a kezelése
 - Operációs rendszertől független, de túlságosan alapszintű hozzáférés, speciális hardver opciók esetleg nem érhetők el így

Hardver elérése

- **Operációs rendszeren keresztül**
 - „HAL” (*Hardware Abstraction Layer*): Hardver absztrakciós szint bevezetése
 - Adott típusú eszközök esetén egységes programozási felület. Könnyebb sokféle hardvert támogató szoftvert készíteni.
 - Az absztrakciós szint miatt kevésbé hatékony!
 - Meghajtó-programokon (driver) keresztül
 - Rosszul megírt driver miatt összeomolhat a rendszer!
 - Harmadik féltől származó kódok esetén digitális aláírás fokozhatja a megbízhatóságot

Hardver elérése

- **Függvénykönyvtárakon keresztül**
 - Felhasználói program szinten
 - Újabb absztrakciós szint a programunk és az operációs rendszer/BIOS/hardver között



Egyszerű példa

- C forráskód

```
#include <stdio.h>
int main( int argc, char *argv[] )
{
    int a, b, sum;
    a = 5;
    b = 7;
    sum = a + b;
    printf( "Osszeg: %d\n", sum );

    return 0;
}
```

Generált Assembly kód (Intel Ubuntu Linux)

```
.file "sum.c"
.section .rodata
.LC0:
.string "Osszeg: %d\n"
.text
.globl main
.type main, @function
main:
    pushl   %ebp
    movl   %esp, %ebp
    andl   $-16, %esp
    subl   $32, %esp
    movl   $5, 28(%esp)
    movl   $7, 24(%esp)
    movl   24(%esp), %eax
    movl   28(%esp), %edx
    leal   (%edx,%eax), %eax
    movl   %eax, 20(%esp)
    movl   $.LC0, %eax
    movl   20(%esp), %edx
    movl   %edx, 4(%esp)
    movl   %eax, (%esp)
    call   printf
    movl   $0, %eax
    leave
    ret
.size   main, .-main
.ident "GCC: (Ubuntu/Linaro 4.5.2-8ubuntu4) 4.5.2"
.section .note.GNU-stack,"",@progbits
```

Optimalizáció nélküli, több felesleges kódrészt is tartalmazó fordítási eredmény.

(A félév végére nagyrészt érteni fogjuk a bal oszlopban szereplő kódot.)

Generált Assembly kód (SUN Sparc)

```
.file "sum.c"
gcc2_compiled.:
.section .rodata
    .align 8
.LLC0:
    .asciz "Osszeg: %d\n"
.section .text
    .align 4
    .global main
    .type main,#function
    .proc 04
main:
    !##PROLOGUE# 0
    save   %sp, -128, %sp
    !##PROLOGUE# 1
    st     %i0, [%fp+68]
    st     %i1, [%fp+72]
    mov    5, %o0
    st     %o0, [%fp-20]
    mov    7, %o0
    st     %o0, [%fp-24]
    ld     [%fp-20], %o0
    ld     [%fp-24], %o1
    add    %o0, %o1, %o0
    st     %o0, [%fp-28]
    sethi  %hi(.LLC0), %o1
    or     %o1, %lo(.LLC0), %o0
    ld     [%fp-28], %o1
    call   printf, 0
    nop
    mov    0, %i0
    b     .LL2
    nop
.LL2:
    ret
    restore
.LLfe1:
    .size   main, .LLfe1-main
    .ident "GCC: (GNU) 2.95.3 20010315 (release)"
```

Tárgykód (Intel Borland C fordító)

```
push    ebp
mov     ebp,esp
mov     eax,5
?live1@32: ; EAX = a
mov     edx,7
?live1@48: ; EAX = a, EDX = b
add     edx,eax
mov     eax,edx
?live1@64: ; EAX = sum
push    eax
push    offset s@
call   _printf
add     esp,8
xor     eax,eax
pop     ebp
ret

00000000 80 07 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000010 42 61 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000020 31 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000030 25 41 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040 42 61 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000050 63 61 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060 28 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000070 61 61 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000080 65 50 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000090 28 D1 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000A0 63 63 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000B0 66 73 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000C0 3A 50 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000D0 69 61 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000E0 68 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000F0 6F 73 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000100 6C 75 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000110 00 E9 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000120 00 E9 00 00 96 28 00 00 05 00 00 00 00 00 00 00
00000130 45 00 00 05 5F 44 41 54 00 00 00 00 00 00 00 00
00000140 52 4F 55 50 04 5F 42 00 00 00 00 00 00 00 00 00
00000150 00 A9 23 00 01 02 00 00 00 00 00 00 00 00 00 00
00000160 00 00 98 07 00 A9 00 00 00 00 00 00 00 00 00 00
00000170 FF 03 FF 02 00 8C 17 00 00 00 00 00 00 00 00 00
00000180 67 76 5F 5F 00 07 5F 00 00 00 00 00 00 00 00 00
00000190 0C 00 00 01 05 5F 6D 61 69 6E 00 00 00 00 00 00
000001A0 00 01 00 00 55 8B EC B8 05 00 00 00 00 BA 07 00 00
000001B0 00 03 D0 8B C2 50 68 00 00 00 00 00 E8 00 00 00 00
000001C0 83 C4 08 33 C0 5D C3 00 9D 09 00 E4 13 54 02 A4
000001D0 18 56 02 00 A0 10 00 02 00 00 4F 73 73 7A 65 67
000001E0 3A 20 25 64 0A 00 00 8B 02 00 00 00 00
```

```
sum.c
Borland C++ 5.5.
1 . . . |sum.c.5
%a .) é (Ú(c:\
Borland\Bcc55\in
clude\_nfile.h .
( é (Ú( c:\Borl
and\Bcc55\includ
e\_null.h .( é
(Ú( c:\Borland\B
cc55\include\_de
fs.h .* é (Ú(c
:\Borland\Bcc55\
include\_stddef.
h .( é (Ú( c:\B
orland\Bcc55\inc
lude\_stdio.h ..
é.%a|sum.c .L
é -( |_TEXT|COD
E |_DATA|DATA-DG
ROUP|_BSS|BSS .
@# . . . @# . |
. . @ . . 5-
' . . . S . _setar
gv . . _printf .
. . |_main
. U<ě . | 5
L<âPh č
.Ä3R]Ä t. ä!!T*
↑U . + . Osszeg
: %d. < .
```

Fájl pozíció

Fájl tartalma:
kód, adat és járulékos (pl. szerkesztőnek szóló) információk.
Középen hexa számokként, jobb oszlopban karakterkódokként.

Adatformátumok x86 címzési módjai

Számrendszerek

- **Fontos számrendszerek Assembly nyelv esetén**
 - **Bináris:** 2-es számrendszer
 - 0 és 1 számjegyek
 - **Decimális:** 10-es számrendszer
 - 0-9 számjegyek
 - **Hexadecimális:** 16-os számrendszer
 - 0-9 számjegyek, A-F betűk
- **Átváltás számrendszerek között, aritmetikai műveletek**
 - Gyakorlaton, illetve korábban a Számítógép architektúrák előadáson

Adatformátumok

- **Szöveg ábrázolása**
 - ASCII, Unicode
- **Számértékek ábrázolása**
 - Egész számok
 - Kettes komplement
 - BCD (binárisan kódolt decimális)
 - Lebegőpontos számok
 - IEEE-754 szabvány szerint

Szöveg ábrázolása

- **ASCII szövegekódolás**

- A bájtok értékei karaktereket jelentenek
- Eredetileg az alsó 7 bit használatos
 - 0-31: vezérlő karakterek (pl. új sor, ESC, szöveg vége)
 - 32: szóköz
 - 33-47: írásjelek
 - 48-57: számjegyek
 - 58-64: írásjelek
 - 65-90: nagybetűk
 - 91-96: írásjelek
 - 97-122: kisbetűk
 - 123-126: írásjelek
 - 127: DEL

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Ábra forrása: Wikipedia

- **Kiterjesztett ASCII**

- 128-255 közötti értékek is definiáltak (pl. nemzeti karakterek, szimbólumok)

Egész számok ábrázolása

- **Előjel nélküli bájt**
 - Bitek kettes számrendszerbeli helyiértékének megfelelően
- **Előjeles bájt**
 - **Kettes komplement** ábrázolás:
 - Pozitív értékből negatív: minden bitet ellentettre (egyes komplement), majd 1 érték hozzáadása
 - Decimális 8 kettes számrendszerben: **00001000**
 - Ellentett képzés: **11110111** (egyes komplement)
 - 1 hozzáadása: **11111000** (-8 reprezentáció)

Számábrázolás

- **Egész számok**

- **Szó (*Word*)**

- x86: 16 bites (2 bájtos) egész

- **Dupla szó (*Double Word, DWORD*)**

- Két szó (x86: 32 bites (4 bájtos) egész)

- **Négy szó (*Quad Word, QWORD*)**

- Négy szó (x86: 64 bites (8 bájtos) egész)

- **Valós számok**

- Közelítő érték tárolása (tört számok)

- Előjel bit, mantissza és exponens

- IEEE-754 szabvány (ld. Architektúrák előadás)

- Egyszeres pontosság: 4 bájt
- Dupla pontosság: 8 bájt
- FPU (*floating point unit*) belső reprezentációja: 10 bájt

x86 üzemmódjai és regiszterkészlete

Első Assembly példaprogram

Processzor családok

- **RISC, CISC, és vegyes architektúrák**
- **Architektúránként rendszerint különböző Assembly nyelv**
- **Néhány ismertebb család**
 - Intel x86, AMD64
 - Sun UltraSparc I-IV
 - MIPS R3000-R10000
 - Motorola 68000
 - MOS 650x (Commodore)
 - Zilog Z80

Intel x86 processzor család

From Computer Desktop Encyclopedia
© 2010 The Computer Language Company Inc.

A táblázatot nem kell megtanulni!

x86 PROCESSORS (from Intel)							
Bits	Family	Clock Speeds (approximate range)	Bus Size (bits)	Max RAM	Floppy Disk	Hard Disk Range	OS
64	Core i3, i5, i7	2.6 - 3.3GHz	64	64GB	3.5" 1.44MB	30GB- 2TB	Win7 Win Vista Win XP Win 2000 Win NT Win 95/98 Win 3.x Linux
	Core 2 Duo	1.8 - 2.6GHz					
	Pentium 4	3 - 3.8GHz					
	Xeon	2.2 - 3.6GHz					
	Pentium D	2.8 - 3.4GHz					
32	Core Duo	1.6 - 2.2GHz	64	4GB	3.5"	500MB- 60GB	SCO Unix Solaris DOS DR DOS OS/2
	Pentium 4	1.4 - 2.8GHz					
	Xeon	400MHz - 3.2GHz					
	Celeron	266MHz - 2.4GHz					
	Pentium III	450MHz - 1.2GHz	64GB	1.44MB	3.5"	60GB	DOS DR DOS OS/2
	Pentium II	233 - 450MHz					
	Pentium Pro	150 - 233MHz	32	4GB	5.25"	200 - 500MB	Misc DOS Multiuser
	Pentium	60 - 200MHz					
	486DX	25 - 100MHz					
	486SX	20 - 40MHz					
386DX	16 - 40MHz						
386SX	16 - 33MHz						
386SL	20 - 25MHz						
16	286	6 - 12MHz	16	16MB	5.25" 1.2MB	20- 80MB	DOS DR DOS Win 3.x OS/2 1.x
	8086	5 - 10MHz					
	8088	5MHz	8	1MB	5.25"	10- 20MB	DOS DR DOS

Intel x86 processzor család

A táblázatot nem kell megtanulni!

Lapka	Dátum	MHz	Tranz.	Mem.	Megjegyzés
I-4004	1971/4	0.108	2300	640	Első egylapkás mikroprocesszor
I-8008	1972/4	0.108	3500	16 KB	Első 8 bites mikroprocesszor
I-8080	1974/4	2	6000	64 KB	Első általános célú mikroprocesszor
I-8086	1978/6	5-10	29000	1 MB	Első 16 bites mikroprocesszor
I-8088	1979/6	5-8	29000	1 MB	Az IBM PC processzora
I-80286	1982/6	8-12	134000	16 MB	Védett üzemmód
I-80386	1985/10	16-33	275000	4 GB	Első 32 bites mikroprocesszor, virtuális 8086 üzemmód
I-80486	1989/4	25-100	1.2M	4 GB	8 KB beépített gyorsítótár
Pentium	1993/5	60-233	3.1M	4 GB	Két csővezeték, MMX
P. Pro	1995/3	150-200	5.5M	4 GB	Két szintű beépített gyorsítótár
P. II	1997/5	233-400	7.5M	4 GB	Pentium Pro + MMX
P. III	1999/2	650-1400	9.5M	4 GB	SSE utasítások 3D grafikához
P. 4	2000/11	1300-3800	42M	4 GB	Hyperthreading + több SSE
Core Duo	2006/1	1600-3200	152M	2 GB	Két mag egy lapkán
Core	2006/7	1200-3200	410M	64 GB	64 bites, négymagos architektúra
Core i7	2011/1	1100-3300	1160M	24 GB	Integrált grafikus processzor

Rendszer tulajdonságok

- **Védelem nélküli rendszerek**
 - Minden utasítás végrehajtható a felhasználói programokban, és
 - „szabadon garázdálkodhat” a memóriában (akár az operációs rendszer kódját is felülírhatja).
 - Asztali számítógépeken idejétmúlt, mikrovezérlőkben OK.
 - Pl. DOS rendszer (16-bites)
- **Védett üzemmódok**
 - CPU-szintű hardveres védelem
 - Utasítás-végrehajtás, kódterület, adatterület védelem
 - Felhasználói mód
 - Bizonyos utasítások végrehajtásának tiltása
 - Pl. gyorsítótár manipulálása
 - Kernelmód
 - Operációs rendszer futtatásához használt
 - Nincs korlátozás
 - Pl. Windows, Linux, MacOS, ... (32- és 64-bitesek)

x86 üzemmódok

- **Valós mód**
 - 8086/8088-ként működik
 - Max. 1 MB memória címezhető
- **Védett mód (32-bit)**
 - A CPU lehetőségeinek teljes kihasználása
 - Védelmi mechanizmusok, gyorsítótár, virtuális memória kezelés, ...
 - Négy privilegizált szint (**FLAG** bitek vezérlik)
 - 0-s szint: kernel mód (operációs rendszer)
 - 1-es, 2-es szint: ritkán használt
 - 3-as szint: felhasználói programok
- **Virtuális 8086 mód**
 - Párhuzamosan több valós módú virtuális gép futtat
 - Védett módú környezet vezérli
 - Ha „lefagy” a virtuális gép, visszatérhetünk az indító környezethez

Intel Core i7 64 bites üzemmódjai

Üzem mód		Szükséges OS	Alkalmazás újrafordítás	Alapbeállítások (bit)		Regiszter kiterjesztés	Ált. célú reg. méret
				Cím	Operandus méret		
Long	64-bit	64-bites	Igen	64	32	Igen	64
	Kompatibilitási		Nem	32	32	Nem	32
				16	16		16
Legacy	Védett mód	32-bites	Nem	32	32	Nem	32
	Virtuális 8086			16	16		
	Valós mód	16-bites		16	16		16

Bár a 64 bites i7 processzor támogatja akár a 16 bites üzemmódot is, a Windows operációs rendszer 64 bites változata ezt nem implementálja. Emiatt régi, 16 bites programokat már nem tudunk közvetlenül futtatni ebben a környezetben.

Kiegészítő anyag!

x86 üzemmódok

- **Assembly alapok**
 - **32-bites védett üzemmód, flat memóriamoddellel kezdünk**
 - Később: 16- és 64-bites lehetőségek
- **Kompatibilitás**
 - Legújabb processzorok is képesek valós módban futni! (Mondjuk nem érdemes így használni...)
 - A 8086 alap utasításkészlete elérhető a védett és virtuális 8086 üzemmódokban is
- **Újdonságok (16 biteshez képest)**
 - Új regiszterek, nagyobb regiszterszélesség
 - Újabb utasítások (pl. MMX multimédiához)
 - Régi utasítások bővítése

x86 memória modell

- **Operatív memória felépítése**

- Bit: Alapegység , 0 vagy 1 érték (Szürke szín = előző előadás anyagában)
- Cella: Legkisebb címezhető egység
 - 8 bit = 1 bájt
- Szó: 16 bit (2 bájt)
- Paragrafus: 16 bájt, 16-tal osztható címen kezdődik
- Lap: 256 (512, 1024, akár több) bájt
- Bájt sorrend: kis-endián
 - A legkisebb helyértékű bájt szerepel alacsonyabb memóriacímen
- Negatív számok kettes komplementes alakban

x86 memória modell

- **Szegmensek**
 - A memória részekre osztva érhető el
 - Valós módban bármi szabadon olvasható és felülírható
 - Akár az operációs rendszer kódja is...
 - Védett módban a hozzáférést jogosultságok szabályozzák
 - Adat és kód esetén is
 - Jogosulatlan hozzáférés esetén hiba!
 - Processzor detektálja, operációs rendszer kezeli
- **Szegmens típusok**
 - Adat, kód, verem
- **Legnagyobb szegmens méretek**
 - Valós mód: 64 kilobájt (16 bites címezés, 2^{16} bájt)
 - Védett mód: 4 gigabájt (32 bites címezés, 2^{32} bájt)

x86 memória modell

- **Kódszegmens(ek)**
 - A program utasításait tartalmazza
- **Adatszegmens(ek)**
 - Konstansok, változók, struktúrák, tömbök területe
- **Veremsegmens**
 - Minden processzus saját vermet kap
 - FIFO adattárolási elv
 - *First In First Out*: amit utoljára tettünk bele, az vehető ki először
 - Kulcsfontosságú a megfelelő működéshez!
 - Eljáráshíváskor visszatérési cím tárolása
 - Eljáráshíváskor paraméterátadás
 - Lokális változók létrehozási helye
 - Adatok ideiglenes mentése pl. megszakításkezeléskor

x86 regiszterkészlete (32 bites)

- **Általános célú regiszterek**

- 32 bitesek
- **EAX, EBX, ECX, EDX, EBP, ESI, EDI**
- Alsó 16 bit, és annak alsó/felső bájtja külön elérhető
 - **EAX; AX; AH; AL**

- **Speciális célú regiszterek**

- 32 bitesek

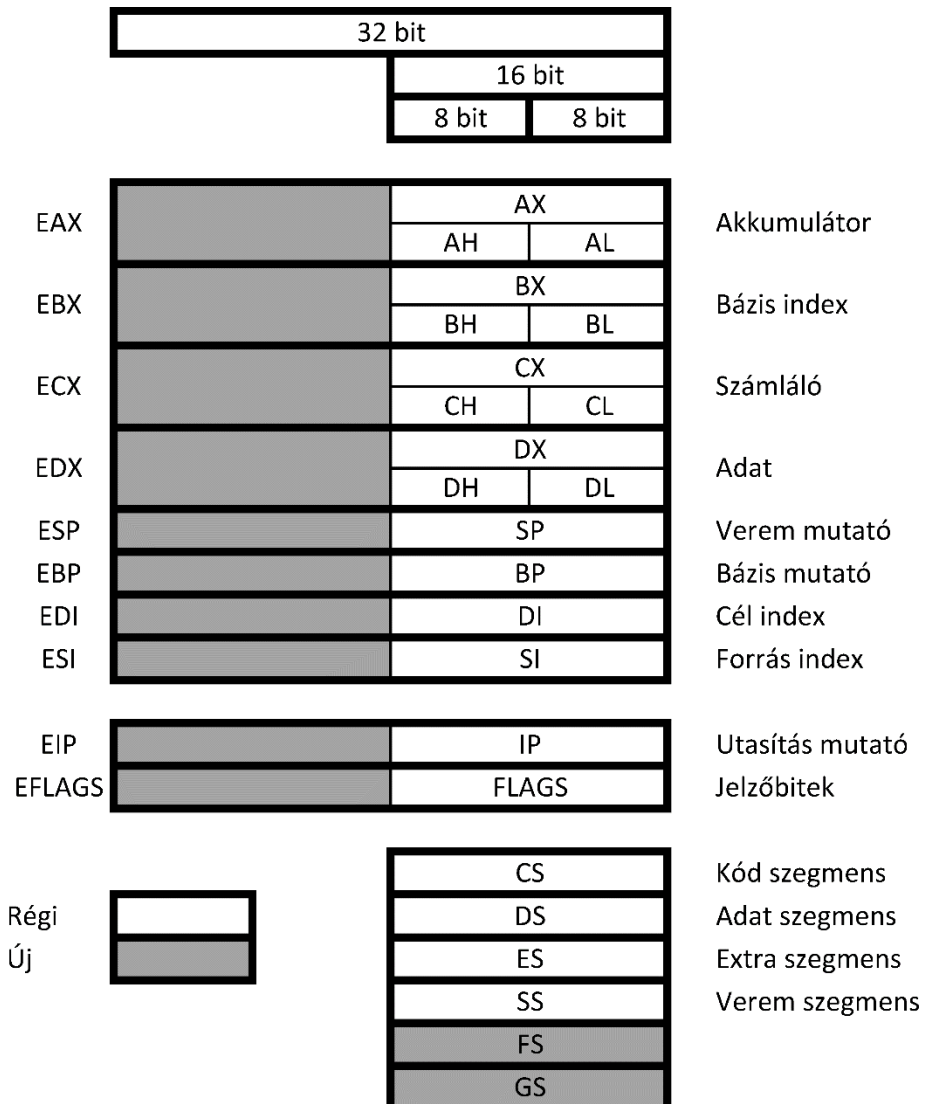
- **EIP, EFLAGS**

- **Szegmens regiszterek**

- 16 bitesek!

- **CS, DS, ES, SS**

- **FS** és **GS** csak védett módban!



x86 regiszterkészlete

- **Általános célú regiszterek**

- **EAX:** Akkumulátor; alsó szava **AX**, annak alsó és felső bájtja **AL** és **AH**
Egyedi szerep szorzásnál és osztásnál
- **EBX:** Bázis (címezés); alsó szava **BX**, annak alsó és felső bájtja **BL** és **BH**
Egyedi szerep memóriacímzésnél
- **ECX:** Számláló; alsó szava **CX**, annak alsó és felső bájtja **CL** és **CH**
Egyedi szerep bitléptető, sztring és ismétléses műveleteknél
- **EDX:** Adat; alsó szava **DX**, annak alsó és felső bájtja **DL** és **DH**
Egyedi szerep szorzásnál és osztásnál
- **EBP:** Bázis mutató; alsó szava **BP**
Egyedi szerep verem indexelt címzésére
- **ESI:** Forrás index; alsó szava **SI**
Egyedi szerep sztring műveleteknél és memóriacímzésnél
- **EDI:** Cél index; alsó szava **DI**
Egyedi szerep sztring műveleteknél és memóriacímzésnél
- **ESP:** Veremmutató
Verem műveletek állítják. Erős speciális jellege miatt általános célra nem célszerű használni.

x86 regiszterkészlete

- **Speciális célú regiszterek**

- **EIP**: Utasítás számláló (csak vezérlésátadással írható felül)

- **EFLAGS**: CPU állapotát jelző bitek összessége

- **C** (Carry): Átvitel előjel nélküli műveleteknél
- **P** (Parity): Az eredmény alsó 8 bitjének paritása
- **A** (Aux Carry): Átvitel a 3. és 4. bitek között (BCD számoknál)
- **Z** (Zero): 1 (igaz), ha az eredmény 0, különben 0 (hamis)
- **S** (Sign): Az eredmény legmagasabb helyiértékű bitje (előjel)
- **T** (Trap): 1: debug mód, 0: automatikus
- **I** (Interrupt): 1: maszkolható megszakítás engedélyezve, 0: tiltva
- **D** (Direction): Sztring műveletek iránya 1: csökkenő, 0: növekvő
- **O** (Overflow): Előjeles túlcsordulás

8086-tól

- **IOP, NT, ...**: Részletesen nem tárgyaljuk

80286, 80386,
80486, Pentium

FLAGS/EFLAGS regiszterek

FLAGS / EFLAGS bitek kiosztása

Új jelzőbitek

IOP Bement/kimenet privilégium szintek (2 bites)

NT Beágyazott feladat

RF folytatás debug esetén

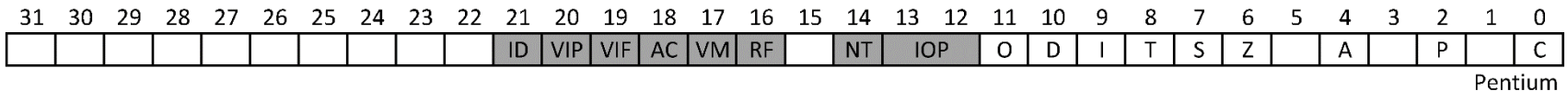
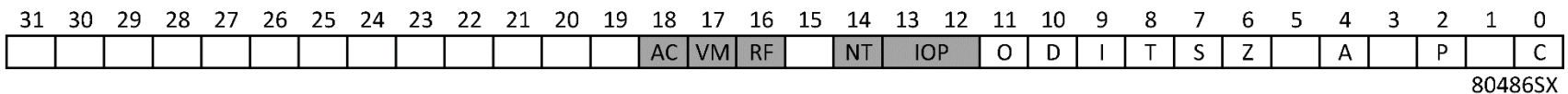
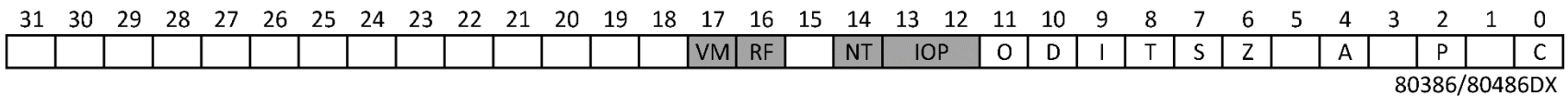
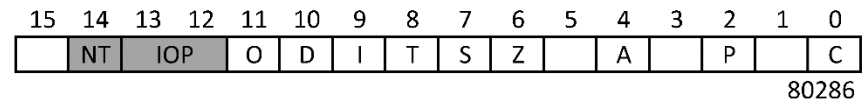
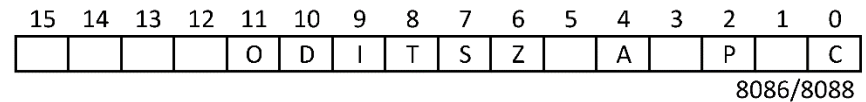
VM virtuális mód (védett üzemmódon belül)

AC igazítás ellenőrzés (80486SX mat. társprocesszorának)

VIF Virtuális megszakítás jelző

VIP Függhőben lévő virtuális megszakítás

ID CPUID utasítás végrehajthatósága



Kiegészítő anyag!

x86 regiszterkészlete

- **Szegmens regiszterek**
 - **CS** (Code Segment): kód szegmens
 - **DS** (Data Segment): adat szegmens
 - **ES** (Extra Segment): (másodlagos) adat szegmens
 - **SS** (Stack Segment): verem szegmens
 - **FS**: extra adatszegmens (csak védett módban)
 - **GS**: extra adatszegmens (csak védett módban)

Valós mód szegmentált címképzése

- **1 megabájt címezhető memória** (2^{20} bájt)
- **20 bites címbusz**, de csak **16 bites regiszterek**
 - **1 regiszter nem elegendő** a címzéshez!
- **Használjunk kettőt**
 - SZEGMENS : OFFSZET alakban írjuk
 - Fizikai cím előállítás: a SZEGMENS értéke 16-tal (10H) szorzódik / 4 bittel balra tolódik / hexa 0 íródik utána
 - Hozzáadódik az OFFSZET értéke
- **Vagyis**
 - Egy szegmens mérete maximálisan 64 KB lehet
 - Egy fizikai címnek több szegmentált címe létezik!
 - Túlcsordulás kezelés nincs
 - FFFF:0010 az első (0.) memóriapozíciót jelenti!
- Könnyebb **relokáció**
- Az egyes szegmensek részben vagy teljesen **átfedőek lehetnek!**

```
SZEGMENS: 1234H
OFFSZET:  0012H

12340H   * 10H
+ 0012H   + 0F.
-----
12352H
```

Memóriacímzés védett módban

- **Változik a szegmens regiszterek szerepe!**
 - Paragrafus cím helyett ***szelektor*** érték
 - Amely egy ***deszkriptort*** (leíró) választ ki
 - Globális deszkriptor tábla (8192 darab bejegyzés)
 - Minden alkalmazás számára közös
 - Lokális deszkriptor tábla (8192 darab bejegyzés)
 - Az adott alkalmazásra nézve lokális
 - **Tartalma**
 - **Szegmens helye, mérete, elérési jogai**
 - 1 deszkriptor mérete 8 bájt → 1 tábla 64 KB helyet foglal
 - A táblák a központi memóriában helyezkednek el
- **A címzés szintaktikája változatlan (szegmens + offszet), „csak” máshogy működik a helymeghatározás!**

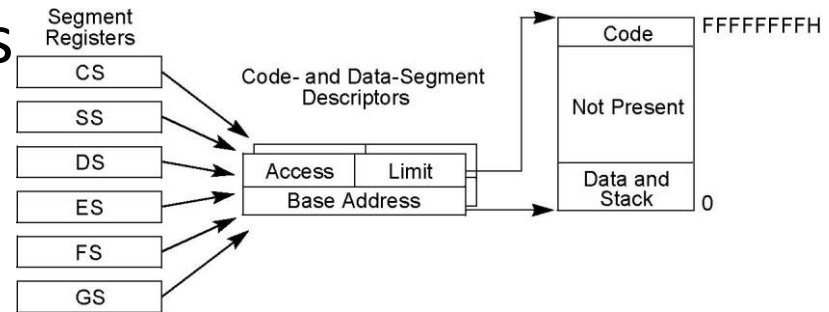
Később tárgyaljuk
részletesebben!

Memória használati stratégiák

Kezdésként ezt választjuk!

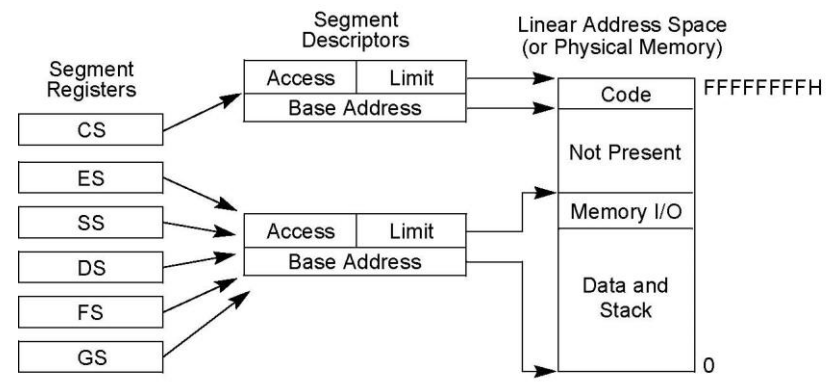
- **Egyszerű flat modell**

- 1 darab 4 GB-os szegmens
- Méret maximumra állítva
- Egyszerű használat



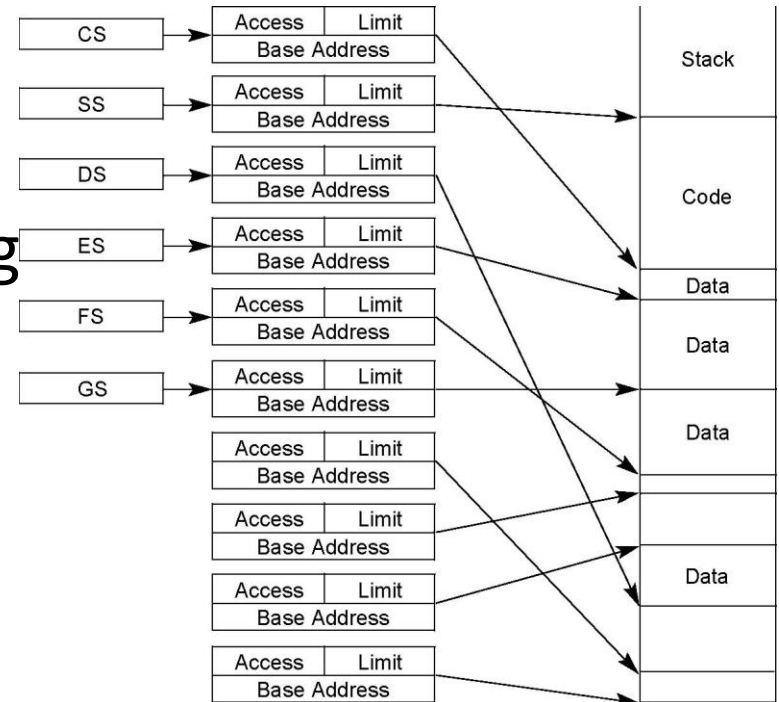
- **Védett flat modell**

- Kód külön kezelve
- Méretek megadva <4 GB



Memória használati stratégiák

- **Multiszegmens modell**
 - Bonyolultabb
 - Jobb ellenőrzési lehetőség
 - Érvénytelen memória hivatkozások detektálása



Assembly program felépítése

- **Egyszerű szöveges fájl**
 - Kiterjesztése **.asm**
 - Kis- és nagybetűk nincsenek megkülönböztetve
- **Minden sorban 1 elemi utasítás**
 - Gépi kódra fordítandó ***Assembly mnemonic kód***
 - Utasítás + operandus(ok)
 - Fordítónak szóló utasítás, ***direktíva***
 - Szegmensek definiálása
 - Használandó memória modell megadása
 - Más modul definícióinak beillesztése
 - Adatterület foglalás, címkék
 - Megjegyzés, kommentár
 - ...

Visual Studio használata

- **MASM Assembly fordító elérhető a Visual Studio-ban**
 - Macro Assembler (Microsoft)
 - 32 és 64 bites fordításra is alkalmas
 - Az ingyenes VS változatokban is
 - Leírás pl. itt (VS 2012 verzióhoz, angolul):
 - <http://www.kipirvine.com/asm/gettingStartedVS2012/index.htm>
 - Kip Irvine eljárásgyűjteménye
 - Rengeteg hasznos segédfüggvényt definiál
 - Leírás a fenti címen
- **Célszerű a MASM32 telepítése is**
 - MASM egy régebbi, ingyenes változata
 - <http://www.masm32.com>


```
TITLE Hello, Assembly! (MASM32)
```

```
.586
```

```
.model flat,stdcall
```

```
option casemap:none
```

```
include \masm32\include\windows.inc
```

```
include \masm32\include\kernel32.inc
```

```
includelib \masm32\lib\kernel32.lib
```

```
.stack 4096
```

```
.data
```

```
        HelloTxt  BYTE    "Hello, Assembly!", 0ah, 0
```

```
        A          DWORD   5
```

```
.code
```

```
; Belépési pont
```

```
main PROC
```

```
        MOV        EDX,OFFSET HelloTxt
```

```
        INVOKE     GetStdHandle, STD_OUTPUT_HANDLE
```

```
        INVOKE     WriteConsole, EAX, ADDR HelloTxt, SIZEOF HelloTxt, EBX, NULL
```

```
        MOV        EAX,7
```

```
        ADD        EAX,A
```

```
        INVOKE     ExitProcess,0
```

```
main ENDP
```

```
END main
```

MASM szintaktika
MASM32 keret
Win32 rendszerhívások

```
TITLE Hello, Assembly! (MASM32)
```

```
.586  
.model flat,stdcall  
option casemap:none
```

```
include \masm32\include\windows.inc  
include \masm32\include\kernel32.inc  
includelib \masm32\lib\kernel32.lib
```

Inicializálás

Zöld színűek: direktívák
Kék színűek: Assembly utasítások, regiszterek

```
.stack 4096
```

Verem

```
.data
```

```
        HelloTxt  BYTE    "Hello, Assembly!", 0ah, 0  
        A          DWORD   5
```

Adatok

```
.code
```

```
; Belépési pont
```

```
main PROC
```

```
        MOV        EDX,OFFSET HelloTxt  
        INVOKE     GetStdHandle, STD_OUTPUT_HANDLE  
        INVOKE     WriteConsole, EAX, ADDR HelloTxt, SIZEOF HelloTxt, EBX, NULL
```

```
        MOV        EAX,7  
        ADD        EAX,A
```

```
        INVOKE     ExitProcess,0
```

```
main ENDP
```

Kódterület

```
END main
```

Forráskód vége

```

TITLE Hello, Assembly! (MASM32)           ; Program megnevezése

.586                                       ; Pentium utasításkészlet
.model flat,stdcall                       ; Memóriamodell, eljáráshívási mód
option casemap:none                       ; Kis- és nagybetűk egyformák

include \masm32\include\windows.inc      ; Win32 eljárásfejlécek, definíciók
include \masm32\include\kernel32.inc     ; behúzása
includelib \masm32\lib\kernel32.lib      ; Win32 tárgykód helye (szerkesztéshez)

.stack 4096                               ; Verem: 4096 bájt (1 lapkeret) méretre

.data
    HelloTxt BYTE "Hello, Assembly!", 0ah, 0 ; Szöveg definiálása
    A        DWORD 5                        ; Változó kezdőértékkel

.code
; Belépési pont
main PROC
    MOV     EDX,OFFSET HelloTxt
    INVOKE  GetStdHandle, STD_OUTPUT_HANDLE
    INVOKE  WriteConsole, EAX, ADDR HelloTxt, SIZEOF HelloTxt, EBX, NULL

    MOV     EAX,7
    ADD     EAX,A

    INVOKE  ExitProcess,0                   ; Kilépés
main ENDP

END main

```

```
TITLE      Hello, Assembly! (32 bites)
INCLUDE   Irvine32.inc                ; Függvények, definíciók
                                           ; Flat memóriamodell, stack beállítása

.data

    HelloTxt  BYTE    "Hello, Assembly!", 0ah, 0
    A         DWORD   5

.code
; Belépési pont
main PROC

    MOV     EDX,OFFSET HelloTxt
    CALL   WriteString                ; Irvine sztring kiírató függvénye

    MOV     EAX,7
    ADD    EAX,A
    CALL   WriteDec                   ; Irvine szám kiírató függvénye

    INVOKE  ExitProcess,0

main ENDP

END main
```

x86 adatterület címezési módjai

Memóriaterület foglalása

- Adat szegmensben

[Azonosító] Típus érték[, érték, ...]

.data

ADAT1 DW 2016, 2018, 2020

TEXT1 DB 'Szoveg'

ADAT2 DB ? ; Inicializálatlan

Memóriaterület foglalása

- **Azonosító (címké)**
 - A sor legelején kell legyen
 - Memóriapozíciót jelent a szegmens elejétől
 - Címké (kódterület) vagy változó (adatterület)
 - 1—247 közötti karakterszám
 - Régi MASM fordítóknál 1—31 lehetett csak
 - Kis- és nagybetű egyenértékűek
 - Az első karakter **betű**, **aláhúzás**, **@**, **?**, vagy **\$** lehet
 - **Aláhúzást** és a **@** karaktert célszerű elkerülni kezdőbetűként
 - A továbbiak akár számok is lehetnek

Memóriaterület foglalása

- **Típus**

– Csak a mérete (bitszélessége) számít

Direktíva (régi)	Jelentése
DB	8-bites egész (bájt)
DW	16-bites egész (szó)
DD	32-bites egész vagy float
DQ	64-bites egész vagy double
DT	80-bites egész (10 bájt)

Direktíva (új)	Jelentése
BYTE	8-bites előjel nélküli egész
SBYTE	8-bites előjeles egész
WORD	16-bites előjel nélküli egész
SWORD	16-bites előjeles egész
DWORD	Előjel nélküli duplaszó (32-bit)
SDWORD	Előjeles duplaszó
FWORD	48-bites egész (távoli mutató védett módban)
QWORD	64-bites egész
TBYTE	80-bites egész
REAL4 , REAL8 , REAL10	IEEE 754 szabvány szerinti float, double és kiterjesztett lebegőpontos

Értékek megadása

[] opcionális
{ | } 1 választandó

- **Egész számok**

[{ + | - }] számjegyek [radix]

Radix jelölés	Jelentése (számrendszer)	Példák
d vagy t	Decimális (10-es), alapértelmezett	12, -1010, 12d, 1010d
h	Hexadecimális (16-os)	12h, 1010h, 0ach
q vagy o	Oktális (8-as)	12o
b vagy y	Bináris (2-es)	1010b
r	Lebegőpontos	3F800000r

- Szám nem kezdődhet betűvel! Az ilyen **hexadecimális számok** elé vezető nullát kell írni!
- Decimális, pozitív szám az alapértelmezés
- Inicializálatlan értéket **?** karakterrel adhatunk meg

Értékek megadása

- **Karakterek, sztringek**

- Bájt típusként kell megadni (**DB** vagy **BYTE**)
- Aposztrófok vagy idézőjelek között
 - Szöveg esetén az eltérő jelek egymásba ágyazhatók
 - ASCII karakterkódolás használatos

K1 **DB** 'c'

K2 **DB** "d", 'a', "Szöveg"

T1 **DB** 'Hello, Assembly!'

T2 **DB** 'Ágyazzuk "ezeket" egymásba'

Memóriacímzési módok

- **Adatterület címzése**

- MOV utasításon keresztül

- **MOV op1, op2**

- Adat mozgatása **op2**-ből **op1**-be

- **op1** és **op2** regiszter vagy memóriacím

- Legalább az egyik operandus regiszter kell legyen!

- **Kódterület címzése**

- Vezérlésátadási lehetőségek

- Rövid, közeli és távoli ugrások

Később tárgyaljuk
részletesebben!

Regisztercímezés

- **Regisztercímezés**

- Az operandusok regiszterek

- `MOV AX, BX`

- `MOV EBP, ESP`

- **Nem megengedett esetekre példa**

- `MOV BX, AL` ; hibás, eltérő típusok!

- `MOV CS, AX` ; CS nem írható!

- `MOV ES, DS` ; hibás, szegmens regiszterek
; között közvetlenül nem megy!

- `MOV DS, DAT_SEG` ; hibás, DS, ES, SS adattal
; közvetlenül nem írható!

Adatterület címzése

- **Kódba épített adat (közvetlen címzés)**
 - Az *adat* (operandus) az *utasítás része*

MOV AX, 07FFH ; AX tartalma 07FFH lesz

MOV CL, 34H ; CL 34H lesz,
; CH nem változik meg!
; Ha CX 1256H volt, akkor
; ezután 1234H lesz.

~~**MOV DS, 02H**~~ ; szegmens regiszterbe nem!

~~**MOV 34H, CL**~~ ; érték nem lehet cél!

Adatterület címzése

- **Direkt memóriacímzés**

- Az *operandus címe* az utasítás része

- Assembly kódban **címkét** használunk

```
ADAT DW 34F2H ; 1 szóhossznyi adat
```

```
; . . .
```

```
MOV AX, ADAT ; AX tartalma 34F2H lesz
```

```
MOV AX, [ADAT] ; ez is ugyanaz
```

- Disassemblerben pl. így láthatjuk

```
MOV AX, [07FE] ; amennyiben ADAT címe  
; 07FEH volt (16-bites)
```

```
MOV AX, word ptr ds:[406000] ; 32-bites
```

Lista fájl (részlet)

Offszet Generált
címek bináris kód

00000000	
00000000	48 65 6C 6C 6F
	2C 20 41 73
	73 65 6D 62
	6C 79 21 0A
	00
00000012	00000005

00000000	
00000000	
00000000	BA 00000000 R
00000005	6A F5 *
00000007	E8 00000000 E *
0000000C	6A 00 *
0000000E	53 *
0000000F	6A 12 *
00000011	68 00000000 R *
00000016	50 *
00000017	E8 00000000 E *
0000001C	B8 00000007
00000021	03 05 00000012 R
00000027	6A 00 *
00000029	E8 00000000 E *
0000002E	

```
.stack 4096

.data
    HelloTxt    BYTE    "Hello, Assembly!", 0ah, 0

    A           DWORD   5

.code
; Belépési pont
main PROC
    MOV     EDX,OFFSET HelloTxt
    INVOKE GetStdHandle, STD_OUTPUT_HANDLE
    push   -00000000Bh
    call   GetStdHandle
    INVOKE WriteConsole, EAX, ADDR HelloTxt, SIZEOF HelloTxt, EBX, NULL
    push   +000000000h
    push   ebx
    push   +000000012h
    push   OFFSET HelloTxt
    push   eax
    call   WriteConsoleA

    MOV     EAX,7
    ADD     EAX,A

    INVOKE ExitProcess,0
    push   +000000000h
    call   ExitProcess

main ENDP

END main
```

Lista fájl (részlet)

```
                                .stack 4096
00000000
00000000 48 65 8C 8C 81                .data
                                HelloTxt    BYTE    "Hello, Assembly!", 0ah, 0
                                2C 20 41 73
                                73 65 6D 62
                                6C 79 21 0A
                                00
00000012 00 00 00 05                A      DWORD    5
00000000
                                .code
                                ; Belépési pont
                                main PROC
00000000 BA 00000000 R                MOV     EDX,OFFSET HelloTxt
                                INVOKE GetStdHandle, STD_OUTPUT_HANDLE
                                push    -00000000Bh
                                call    GetStdHandle
                                INVOKE WriteConsole, EAX, ADDR HelloTxt, SIZEOF HelloTxt, EBX, NULL
                                push    +000000000h
                                push    ebx
                                push    -0000000012h
                                push    OFFSET HelloTxt
                                push    eax
                                call    WriteConsoleA
0000001C B8 00000007                MOV     EAX,7
00000021 03 05 00000012 R            ADD     EAX,A
                                INVOKE ExitProcess,0
                                push    +000000000h
                                call    ExitProcess
                                main ENDP
                                END main
```

Címkéből offszetcím
(szegmens kezdettől
számított eltolási
számérték) lesz!
Fordítási időben ismert!

Adatterület címzése

- **Direkt memóriacímzés**

- Az *adatok típusának* egyezni kell!

- `ADAT1 DB 12H, 34H; bájt`

- `ADAT2 DW 1234H ; Little-endian szó`
`; tárolás: 34H 12H`

- `; . . .`

- ~~`MOV AX, ADAT1;`~~ hibás, mert ADAT1 bájt!

- `MOV AL, BYTE PTR ADAT2; OK, típus`
`; átalakítás: AL=34H`

Adatterület címzése

- **Regiszter indirekt címzés**
 - A címzéshez használt **[regiszter]** értéke adja az operandus memóriacímét
 - **EAX, EBX, ECX, EDX, ESI, EDI, EBP** használható
 - **EBP** vagy **BP** használata esetén a verem szegmens, minden más esetben az adat szegmens kerül címzésre!
 - 16-bites esetben csak **BX, BP, SI, DI** lehet

```
MOV EAX, [ EDI ]
```

```
; EAX tartalma az EDI érték mint címen  
; lévő 32-bites egész érték lesz
```

```
MOV [ EDI ], AX
```

```
; Az EDI értéke szerinti memóriacímen  
; tárolásra kerül AX 16-bites tartalma
```

Adatterület címzése

- Regiszter indirekt címzés

```
ADAT1 DW 912, 920, 928
```

```
ADAT2 DB 11, 15, 20
```

```
; 912 szó érték AX-be (hexa 0390)
```

```
; OFFSET direktíva: címke cím értéke
```

```
MOV EDI,OFFSET ADAT1
```

```
MOV AX,[ EDI ]
```

```
; 144 (hexa 90) bájtos érték írása
```

```
; 11 helyére
```

```
MOV EDX,OFFSET ADAT2
```

```
MOV [ EDX ],AL
```

Adatterület címzése

- **Regiszter relatív címzés**
 - Az utasításban elhelyezett **számot** (eltolás) hozzáadja a kiválasztott **[regiszterhez]**
 - **EAX, EBX, ECX, EDX, ESI, EDI, EBP**
 - 16 bites esetben csak **SI** vagy **DI** lehet!

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

```
; AX tartalma DI + 7FH
```

```
; memóriacím szavas tartalma lesz
```

```
MOV [ EDI + 07FH ],AX
```

```
; AX értéke a fenti memóriacímre kerül
```

Adatterület címzése

- **Bázis-index címzés**

- Két regiszter összege adja a címet

- **EAX, EBX, ECX, EDX, ESI, EDI, EBP**

- 16-bites esetben csak egy bázis (**BX** vagy **BP**) és egy index regiszter (**SI** vagy **DI**) használható

```
MOV AX, [ EBX + ESI ]
```

```
MOV AX, [ EBX + ECX ]
```

Adatterület címzése

- **Bázis-index relatív címzés**

– **Bázis** + **index** + **eltolás** adja a címet

```
MOV AX, 057FH [ ESI ] [ EBX ]
```

```
; AX értéke ESI + EBX + 057FH
```

```
; címen található szó értéke lesz
```

```
MOV AX, [ EBX + ESI + 057FH ] ; ugyanaz
```

Adatterület címzése

- **Skála-index címzés**

- **Bázis** + **skála** * **index** + **eltolás** adja a címet

- Eltolás el is maradhat
 - Skála lehet: (1,) 2, 4, vagy 8
 - 16 bites változata nincs!

- **Ideális 1- és 2-dimenziós tömbök indexelésére**

- 1D tömb: Tömb kezdőcíme a bázis, skála a tömbelem típusának mérete, az index az elérendő elem sorszáma (0-bázisú)
 - 2D tömb: Eltolás a tömb kezdőcíme, bázis a sor kezdőcíme, skála a tömbelem típusának mérete, index az oszlop száma (0-bázisú)

Címzés	Példa	Tipikus használat
Regiszter	<code>MOV EAX, EBX</code>	Regiszterek közötti adatáramlás
Közvetlen	<code>MOV EAX, 1492</code>	Fordítási időben ismert konstans számérték közvetlen használata
Direkt	<code>ADAT1 DW 5192, 14, 6</code> ... <code>MOV AX, ADAT1</code>	Adatterületen elhelyezett, fordítási időben megadott változó elérése nevének keresztül
Regiszter indirekt	<code>MOV ESI, OFFSET ADAT1</code> <code>ADD ESI, 4</code> <code>MOV AX, [ESI]</code>	Futási időben kiderülő címen található változó értékének beolvasása
Regiszter relatív	<code>MOV EAX, [ESI+4]</code> <code>MOV AX, ADAT1 [ESI]</code> <code>MOV AX, [ADAT1+ESI]</code>	Futási időben megadott helyen lévő tömb fordítási időben ismert eleme (bájt pozíció!) Fordítási időben ismert helyen lévő tömb futási időben megadott eleme (bájt pozíció!)
Bázis-index	<code>MOV EAX, [EBX+ESI]</code>	Futási időben megadott helyen lévő tömb futási időben kiderülő elemének elérése (bájt pozíció!)
Bázis-index relatív	<code>ADAT2 DD 1, 2, 3, 4, 5, 6</code> ... <code>MOV EAX, ADAT2 [EBX+ESI]</code>	Fordítási időben ismert helyen lévő 2D tömb futási időben kiderülő címzése (sor, oszlop; bájt pozíció!) (A 2D tömb technikailag 1D tömbként kerül reprezentálásra.)
Skála-index	<code>MOV AX, [EBX+2*ESI]</code>	Futási időben megadott helyen lévő tömb fordítási időben ismert méretű elemmel, futási időben való címzése (elem index!) (2D tömbre is, lásd előzőt)

Adatterület címzése

- **Verem**
 - LIFO-elvű (utoljára be – először ki) adatterület
 - **ESP regiszter** automatikusan állítódik eljáráshíváskor és veremműveletek esetén
 - Csökken berakáskor, növekszik kivételkor
 - **SS : ESP** a verem tetejét (legfelső elemét) mutatja
 - Ha címzésben szerepel **ESP** vagy **EBP**, akkor alapértelmezés szerint a veremterület kerül címzésre
- **Használható például**
 - Regiszterek értékének átmeneti tárolására
 - Eljárás híváskor a visszatérési cím tárolására
 - Eljárásnak paraméterek átadására
- **Verem címzése**
 - Részletesen az eljárások paraméterátadásánál foglalkozunk vele

Alapértelmezett szegmens regiszterek

- **Aktuális utasítás címe**
 - **CS : EIP**, csak ugró utasítással módosítható!
- **Adat szegmens**
 - **EAX , EBX , ECX , EDX , ESI , EDI ,**
címké/offszet/eltolás esetén **DS** az alapértelmezett
 - Sztring műveleteknél **EDI** alapértelmezett szegmens regisztere **ES** (**DS : ESI** a forrás, **ES : EDI** a cél)
- **Verem**
 - **SS** az alapértelmezett szegmens regiszter
 - **EBP** és **ESP** használható címzésre, minden más módhoz felüldefiniálás kell (pl. **MOV AX , SS : [ESI]**).

Alapértelmezés felülbírálása

- **Alapértelmezett szegmens regiszter felülbíráható**

- Prefix alkalmazásával

- **FS** és **GS** szegmensek csak így érhetőek el

```
MOV AX,DS:[EBP] ; verem helyett adat szegmens
MOV AX,ES:[EBP] ; verem helyett extra szegmens
MOV AX,SS:[EDI] ; adat szegmens helyett verem sz.
MOV AX,CS:[ESI] ; adat szegmens helyett kód sz.
MOV AX,FS:LIST ; adat szegmens helyett F sz.
LODS ES:DATA ; adat szegmens helyett extra sz.
MOV AX,SS:[ESI]
```

További szabályok

- **Kétooperandusú műveleteknél**
 - legalább egyik operandusa regiszter kell legyen,
 - a két operandus mérete egyező kell legyen,
 - (kivéve **MOVZX** és **MOVSX**)
 - a két operandus nem lehet két szegmens regiszter.
- **DS, SS, ES beépített adattal nem írható**
 - Először valamelyik általános célú regiszterbe töltsük az értéket, majd innen a szegmens regiszterbe
- **SS írásával vigyázzunk**, mert elvész a verem!
- **Szegmens regiszter** nem lehet aritmetikai művelet operandusa
- **CS és EIP regiszter** nem írható, nem tehető verembe, értéke csak ugrással állítható

```
MOV AX, [SI] ; OK
MOV DS, AX   ; OK
MOV [SI], [DI] ; hiba!
```

```
ADAT SEGMENT
SZAM DB 00,01,02
ADAT ENDS
; ...
MOV DS, ADAT ; hiba!
MOV AX, ADAT ; cím be
MOV DS, AX   ; OK
```

```
MOV CS, AX ; lefagy!
JMP cimke   ; OK
CALL eljárás ; OK
JNE ciklus  ; OK
```

Adatmozgató utasítások

Utasítások

- **Adatmozgatás operandusok között**
 - **MOV, MOVZX, MOVSX**
- **Operandusok felcserélése**
 - **XCHG**
- **Kódkonverzió táblázat alapján**
 - **XLAT, XLATB**
- **Cím betöltése**
 - **LDS, LES, LLFS, LGS, LSS**
 - **LEA**
- **FLAG regiszter alsó bájtja mentése AH-ba és visszaállítása**
 - **LAHF, SAHF**
- **Verem műveletek**
 - **PUSH, POP; PUSHF, POPF; PUSHFD, POPFD; PUSHA, POPA; PUSHAD, POPAD**

Mozgatás

- **MOV op1 , op2**
 - op2 tartalma op1-be
 - Legalább az egyik operandus regiszter kell legyen
 - **op1 és op2 bitszélességének egyezni kell**
 - Viszont ha 8, vagy 16-bites regiszterbe mozgatunk (pl. AL, AH, AX), a 32-bites regiszter (EAX) címzetlen bitjei változatlanok maradnak!
 - Részletesen ismertetésre került a címzési módoknál

Mozgatás eltérő bitszélességgel

Nullázás

- **MOVZX op1, op2**
 - **op2** tartalma **op1**-be
 - **op1** 16 vagy 32 bites regiszter kell legyen!
 - **op2** memóriacím vagy regiszter
 - **op2** bitszélessége kisebb, mint **op1**!
 - **op1** magasabb helyiértékű bitjei nullázásra kerülnek
 - **Viszont ha 16-bites regiszterbe mozgatunk (pl. AX), a 32-bites regiszter (EAX) felső 16 bitje változatlan marad!**

```
ADAT1 DB 10
```

```
...
```

```
MOVZX CX, ADAT1 ; CH nullázódik
```

```
MOVZX EDX, ADAT1 ; EDX felső 3 bájttja 0
```


Mozgatás eltérő bitszélességgel

Előjel-kiterjesztés

- **MOVSX op1, op2**
 - **op2** tartalma **op1**-be
 - Mint **MOVZX**-nél
 - **op1** magasabb helyiértékű bitjei az előjel értékét veszik fel!
 - **Viszont ha 16-bites regiszterbe mozgatunk (pl. AX), a 32-bites regiszter (EAX) felső 16 bitje változatlan marad!**

```
ADAT2    DB    5, -5
```

```
...
```

```
; CH nullázódik (pozitív)
```

```
MOVSX    CX, ADAT2
```

```
; EDX felső 3 bájtja csupa 1 bit értékű
```

```
; (negatív előjel)
```

```
MOVSX    EDX, [ADAT2+1]
```

MOVZX, MOVSX

Bitszélesség kiterjesztés előjel nélkül vagy előjellel (MOVZX, MOVSX)

Előjeles dec.	Hexa	Negatív érték esetén probléma! (Kettes komplement számábrázolás miatt.)			
-5	FB		1 1 1 1 1 0 1 1	8-bites	ADAT1
32-bites regiszter felső bitjei változatlanok!					
251	00FB		0 0 0 0 0 0 0 0 1 1 1 1 1 0 1 1	Kiterjesztés 0-val	MOVZX AX, ADAT1
-5	FFFB		1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1	Előjeles kiterjesztés	MOVSX AX, ADAT1
251	000000FB		0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 1 1	Kiterjesztés 0-val	MOVZX EAX, ADAT1
-5	FFFFFFFB		1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1	Előjeles kiterjesztés	MOVSX EAX, ADAT1
Pozitív érték esetén megegyeznek.					
5	05		0 0 0 0 0 1 0 1	8-bites	ADAT2
5	0005		0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1	Kiterjesztés 0-val	MOVZX EAX, ADAT2
5	00000005		0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1	Előjeles kiterjesztés	MOVSX EAX, ADAT2

Csere

- **XCHG op1 , op2**
 - **op1** és **op2** tartalmának felcserélése
 - Segédváltozó használata nélkül
 - Legalább az egyik operandus regiszter kell legyen
 - **op1** és **op2** bitszélességének egyezni kell

Kódkonverzió táblázat alapján

- **XLAT**

- Táblázat bájtos adatokkal
- Legfeljebb 256 érték szerepelhet benne

- **AL** ← [**AL** + **EBX**]

- **EBX**: táblázat kezdőcíme
- **AL**: elem indexe
- Eredmény: **AL**-be bekerül a táblázat adott indexű eleme

```
HEXTB DB '0123456789ABCDEF' ; hexa szjegyek
```

```
...
```

```
MOV AL,12D ; AL 12D
```

```
MOV EBX,OFFSET HEXTB
```

```
XLAT ; AL értéke 'C' karakterkódja lesz
```

Cím és szegmens betöltése

- **Használatuk**
 - Ki lehet váltani velük több műveletet
 - **Flat memóriamodell esetén nem kerülnek használatra!**
- **LDS *reg, mem*** (*Load Data Segment*)
 - Az adott *mem* címen lévő első **2 vagy 4 bájtt** *reg*-be kerül
 - *reg* bitszélességének megfelelően
 - A rákövetkező 2 bájtt **DS** regiszterbe
- **LES *reg, mem*** (*Load Extra Segment*)
- **LFS *reg, mem*** (*Load FS*)
- **LGS *reg, mem*** (*Load GS*)
- **LSS *reg, mem*** (*Load Stack Segment*)
 - Ua. mint **LDS**, csak más szegmens regiszterrel

Tényleges cím betöltése

- **LEA reg,mem** (*Load Effective Address*)
 - A memóriacímzés offszetje a kért regiszterbe kerül
 - Tetszőleges memóriacímzés eredménye 1 utasítással meghatározható
 - **Futási időben történik a számítás!**
 - Akár **konstans + reg + skála * reg** aritmetikai számításra is jó
- ```
; EBX + EDI + 322
LEA ESI, 322 [EBX] [EDI]
; További példák a memóriacímzésnél
```

# Mozgatás

- **LAHF** (*Load AH from FLAGS*)
  - **FLAGS** regiszter alsó bájtyának értéke **AH**-ba
- **SAHF** (*Save AH into FLAGS*)
  - **AH** értéke **FLAGS** regiszter alsó bájtyába

# Verem műveletek

- **PUSH op**

- Operandus értéke a verembe kerül

- Regiszter vagy memóriacím értéke

- Csak 16 vagy 32 bites érték mozgatható!

- A verem az alacsonyabb memóriacímek felé bővül!

- $SS: [ESP-2] = op; ESP = ESP - 2$  (16 bit)

- $SS: [ESP-4] = op; ESP = ESP - 4$  (32 bit)

- **POP op**

- A verem tetejének értéke **op**-ba kerül

- Csak 16 vagy 32 bites érték mozgatható!

- $Op = SS: [ESP]; ESP = ESP + 4$

- $Op = SS: [ESP]; ESP = ESP + 4$



# Verem műveletek

- **FLAG regiszter mentése**

- **PUSHF, POPF**

- **FLAGS** regiszter (16 bites) tartalma kerül a verembe, illetve onnan visszaolvasásra

- **PUSHFD, POPFD**

- **EFLAGS** regiszter (32 bites) tartalma kerül a verembe, illetve onnan visszaolvasásra

- **Általános célú regiszterek mentése**

- **PUSHA, POPA**

- Az összes 16 bites általános célú regiszter tartalma kerül a verembe, illetve onnan visszaolvasásra

- **AX, BX, CX, DX, SP, BP, SI, DI**

- **PUSHAD, POPAD**

- Az összes 32 bites általános célú regiszter tartalma kerül a verembe, illetve onnan visszaolvasásra

- **EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI**

# Verem inicializálása

- **Automatikus**

- Ha a forráskódban megfelelően definiáltuk a verem szegmenst, akkor

- **SS** felveszi a szegmenscímet,

- **ESP** pedig a lefoglalt verem terület legvégére mutat

- a program indulásakor.

# x86 logikai és aritmetikai utasításai

# Logikai utasítások

- **AND , OR , XOR**

- Logikai ÉS, VAGY, KIZÁRÓ VAGY művelet az operandusok bitjei között
- A „szokásos” igazságtáblák szerint
- Eredmény **op1**-be

**AND op1 , op2**

- **NOT**

- Operandus bitjeit negálja (egyes komplement)

**NOT op**

- (Kettes komplement: **NEG** utasítás (aritmetikai művelet))

- **TEST**

- Logikai ÉS művelet az operandusok között, de csak a **flag**-ek állítódnak, az operandusok változatlanok maradnak!
- Általában 1 bit tesztelésére használatos

**TEST op1 , op2**

# Aritmetikai utasítások

- **Összeadás, kivonás**
  - Előjeles (kettes komplement): **ADD**, **ADC**, **SUB**, **SBB**
  - BCD: **DAA**, **DAS**
  - ASCII: **AAA**, **AAS**
- **Szorzás, osztás**
  - Előjeles, előjel nélküli: **MUL**, **IMUL**, **DIV**, **IDIV**
  - BCD: **AAD**, **AAM**
- **Növelés, csökkentés, negálás**
  - **INC**, **DEC**, **NEG**
- **Előjel kiterjesztés**
  - **CBW**, **CWD**, **CWDE**, **CDQ**
- **Összehasonlítás**
  - **CMP**

# Aritmetikai utasítások

- **Hatásuk a FLAGS regiszter bitjeire**
  - **I, D, T**: nem változnak
  - **C, P, A, Z, S, O**: változhatnak
    - Átvitel (**C**): 8 vagy 16 biten nem fér el az eredmény
    - Paritás (**P**): 1-es értékű bitek (csak az alsó bájton!)
    - Másodlagos átvitel (**A**): 3. és 4. bitek közötti átvitel
    - Zéró (**Z**): eredmény nulla-e? (1 = igen, 0 = nem)
    - Előjel (**S**): eredmény legmagasabb bitje (kettes komplement)
    - Túlcsordulás (**O**): 1, ha az eredmény nem fért el az adott típus értéktartományán
- **Az eredmény elkészülte mellett**
  - Feltételes ugrásokat végezhetünk a **flag** bitek alapján
  - **C** és **O** vizsgálatával az eredmény előjelét megállapíthatjuk

# Összeadás (ADD)

- **Előjeles (kettes komplement), átvitel nélkül**
  - **Z, C, A, S, P, O** flageket állítja

```
ADD EAX,EBX ; EAX = EAX + EBX
```

```
ADD AL,BL ; AL = AL + BL
```

```
ADD CL,44H ; CL = CL + 44H
```

```
ADD [BX],AL ; AL értéke hozzáadódik BX által
; mutatott memóriacím értékéhez,
; tárolás a memóriacímen
```

```
ADD DS,4 ; szegmens regiszterhez nem!
```

```
ADD [BX],ADAT ; legalább 1 regiszter kell!
```

# Összeadás (ADC)

- **Előjeles (kettes komplement), átvitel**
  - **Z, C, A, S, P, O** flageket állítja
  - **C** flag bit értékét is hozzáadja az összeghez

```
ADC AL,AH ; AL = AL + AH + C
```

```
ADC CL,44H ; CL = CL + 44H + C
```

```
; 64 bites összeadás EBX:EAX + EDX:ECX
```

```
ADD EAX,ECX ; alsó 32 bit, átvitel C-be
```

```
ADC EBX,EDX ; felső 32 bit: EBX + EDX + C
```



# Növelés, csökkentés, negálás

- **INC op**
  - Operandus értékét 1-gyel növeli
  - 8 vagy 16 bites
  - **C flaget NEM állítja**
- **DEC op**
  - Operandus értékét 1-gyel csökkenti
  - 8 vagy 16 bites
  - **C flaget NEM állítja**
- **NEG op**
  - Operandus ellentettjét képz
  - Kettes komplement alakban!

| Bináris érték | Decimális |
|---------------|-----------|
| 00000101      | 5         |
| 11111011      | -5        |

# Kivonás (SUB)

- **Előjeles (kettes komplement), átvitel nélkül**
  - **Z, C, A, S, P, O** flageket állítja

```
SUB EAX,EBX ; EAX = EAX - EBX
```

```
SUB CL,44H ; CL = CL - 44H
```

```
SUB [BX],AL ; AL értéke kivonódik a BX által
; mutatott memóriacím értékéből,
; tárolás a memóriacímen
```

```
MOV CH,22H ; decimális 34-ből
```

```
SUB CH,44H ; decimális 68 kivonása
; az eredmény DEh lesz, ami
; helyes (-34)
; nincs túlcsordulás sem!
```

# Kivonás (SBB)

- **Előjeles (kettes komplement), átvitel**
  - **Z, C, A, S, P, O** flageket állítja
  - **C** flag bit értékét is kivonja

**SBB AL, AH** ; **AL = AL - AH - C**

**SBB CL, 44H** ; **CL = CL - 44H - C**

# Összehasonlítás (CMP)

- **Hatása**

`CMP op1, op2`

- Az operandusok nem változnak!
- A **flag**-ek az **op1-op2** kivonásnak megfelelően állnak be
- Feltételes vezérlésátadáshoz jól felhasználható

# Szorzás (8 bites)

- **Általános szabály 8-bites szorzásra**
  - A szorzandó mindig az **AL** regiszter!
  - Eredmény **AX**-ben képződik (16 bites eredmény)!
  - **C** flag 1 értéke jelzi, ha van átvitel **AH**-ba
- **Előjeles szorzás: **IMUL****  
**IMUL DH** ; AX = AL \* DH
- **Előjel nélküli: **MUL****  
**MOV BL, 5** ; BL előkészítése  
**MOV CL, 100** ; CL előkészítése  
**MOV AL, CL** ; CL AL-be töltése  
**MUL BL** ; AX = AL \* BL  
**MOV DX, AX** ; DX = AX = AL \* BL = CL \* BL

# Szorzás (16 bites)

- **Általános szabály 16-bites szorzásra**
  - A szorzandó mindig az **AX** regiszter!
  - Eredmény **DX:AX** regiszterekben képződik
    - 32 bites eredmény! (**DX**-ben a felső helyiértékű bitek)
  - **IMUL**, **MUL** utasítások, mint 8-bites esetben
  - **C** flag 1 értéke jelzi, ha van átvitel **DX**-be

```
MUL CX ; DX:AX = AX * CX
```

```
MUL WORD PTR [SI] ; DX:AX = AX *
; SI helyen a memóriában található
; szó értéke
; Kell a WORD PTR, mert a fordító nem
; tudja, hány bites a művelet!
```

# Szorzás (32 bites)

- **Általános szabály 32-bites szorzásra**
  - A szorzandó mindig az **EAX** regiszter!
  - Eredmény **EDX : EAX** regiszterekben képződik
    - 64 bites eredmény! (**EDX**-ben a felső helyiértékű bitek)
  - **IMUL**, **MUL** utasítások, mint 8-bites esetben
  - **C** flag 1 értéke jelzi, ha van átvitel **EDX**-be

```
MUL ECX ; EDX:EAX = EAX * ECX
```

```
MUL DWORD PTR [ESI] ; EDX:EAX = EAX *
; ESI helyen a memóriában található
; szó értéke.
; Kell a DWORD PTR, mert a CPU nem
; tudja, hány bites a művelet!
```

# IMUL specialitások

A **MUL** utasítás továbbra is csak 1 operandusú lehet!

- **Lehet két operandusa is (80286-tól)**
  - Az első operandus egy **regiszter** (szorzandó)
    - 16 vagy 32 bites
  - A szorzó lehet **regiszter**, **memóriahivatkozás** vagy **közvetlen érték**
    - Elsővel egyező bitszélességű!
  - Eredmény az első operandusban keletkezik
    - **Itt nincs bitszélesség kiterjesztés!**
    - **C** és **O** flag bit értékek mutatják, ha van elveszett átvitel!

```
IMUL ECX, 4
```

```
MOV AX, 20000d
```

```
IMUL AX, 90d ; Ez már nem fér el AX-ben!
; És itt nincs átvitel DX-be!
```



# IMUL specialitások

- **Lehet három operandusa is (80386-től)**
  - Az első operandus egy **regiszter** (szorzandó)
    - 16 vagy 32 bites
  - A második operandus lehet **regiszter** vagy **memóriahivatkozás**
    - Elsővel egyező bitszélességű!
  - A harmadik operandus egy **közvetlen érték**
  - Eredmény az első operandusban keletkezik
    - **Itt nincs bitszélesség kiterjesztés!**
    - **C** és **O** flag bit értékek mutatják, ha van elveszett átvitel!

```
IMUL AX, DX, 6
```

```
IMUL ECX, [EDX+6], 5
```

# Osztás (DIV, IDIV)

- Előjeles osztásnál a **maradék előjele az osztandó előjele**
- **16 bites osztandó (AX), 8 bites osztó (operandus)**
  - **AL** a hányadost, **AH** az egész maradékot tartalmazza osztás után
- **32 bites osztandó (DX : AX), 16 bites osztó**
  - **AX** a hányadost, **DX** az egész maradékot tartalmazza osztás után
- **64 bites osztandó (EDX : EAX), 32 bites osztó**
  - **EAX** a hányadost, **EDX** az egész maradékot tartalmazza osztás után

# Osztás (DIV, IDIV)

- **Szükség esetén az osztandó szélességét előjelhelyesen ki kell terjeszteni!**
  - Egyező méretű osztandó és osztó esetén
  - Előjel nélküli esetben 0-val
    - **AH**, illetve **DX**, **EDX** tartalma
  - Előjeles esetben minden bitet az előjel bit értékével

# Előjel-kiterjesztés

Nem csak osztáshoz, más célból történő kiterjesztéshez is használhatók!

- **CBW** (*Convert Byte to Word*)
  - Bájtból előjelhelyes szó
  - **AL** regisztert egészíti ki
    - Negatív esetben hexa **FF** lesz **AH**, egyébként **0**
- **CWD** (*Convert Word to Double Word*)
  - Szóból előjelhelyes dupla szó
  - **AX** regisztert egészíti ki
    - Negatív esetben hexa **FFFF** lesz **DX**, egyébként **0**
- **CWDE** (*Convert Word to Extended Doubleword*)
  - **AX** regisztert egészíti ki előjelhelyes **EAX**-re
    - Negatív esetben **FFFF** lesz **EAX** felső két bájtja, egyébként **0**
- **CDQ** (*Convert Doubleword to Quadword*)
  - **EAX**-et egészíti ki előjelhelyes 64-bites értékre
    - A felső 32 helyiértékű bitek **EDX**-be kerülnek

# Osztás (IDIV, DIV)

- **Vizsgálatok osztás előtt**
  - 0-val osztás hibát okoz (csapda)!
  - Ha az osztás eredménye nem fér el **AL**-ben (8 bites), **AX**-ben (16 bites), illetve **EAX**-ben, akkor ...
    - 0. megszakítás hívódik
      - Alapértelmezés szerint kilép a programból
      - Átirányíthatjuk saját kódra (lásd a megszakításoknál)
    - Inkább előzetes ellenőrzés javasolt
      - Pl. 8 bites osztásnál: ha **AH**  $\geq$  **op**, akkor túl nagy az osztandó!
    - Vagy 8 bites osztás helyett végezzünk 16 bitest!
      - 16 helyett 32 bitest, ...

# Osztás

```
; 8 bites előjel nélküli osztás
MOV AL,NUMB ; NUMB adat betöltése
MOV AH,0 ; AH nullázása előjel nélküli
 ; osztás előtt
DIV NUMB1 ; előjel nélküli osztás

; 16 bites előjeles osztás
MOV AX,-100
MOV CX,9
CWD ; DX-AX: előjelhelyes -100 érték
IDIV CX
```

# Osztás

- Példa: maradék vizsgálata kerekítéshez

```
DIV BL
```

```
ADD AH,AH ; maradék duplázása
```

```
CMP AH,BL ; nagyobb-e az osztónál?
```

```
JB NEXT ; ha nem, akkor jó az eredmény
```

```
INC AL ; ha igen, akkor felfelé kerekít
```

```
NEXT: ...
```

# Flag-ek viselkedése

- **1. példa**

$$53 + 18 = 71$$

$$\begin{array}{r} 00110101 \\ + 00010010 \\ \hline \end{array}$$

**0** **0** **0**1000111

Rendben, előjelhelyes is.

Túlcsordulás (O), Átvitel (C), Előjel (S)

- **2. példa**

$$53 + 83 = 136$$

$$\begin{array}{r} 00110101 \\ + 01010011 \\ \hline \end{array}$$

**1** **0** **1**0001000

Ez -120! Előjel rossz, túlcsordulás is van!



# Flag-ek viselkedése

- **3. példa**

$$-53 + -18 = -71$$

$$\begin{array}{r} 11001011 \\ + 11101110 \\ \hline \end{array}$$

**0** **1** **1** 0111001      Rendben, de C=1!

- **4. példa**

$$-53 + -83 = -136$$

$$\begin{array}{r} 11001011 \\ + 10101101 \\ \hline \end{array}$$

**1** **1** **0** 1111000      Ez +120! Előjel is rossz,  
túlcsordulás és átvitel  
is van!

# Flag-ek viselkedése

- **5. példa**

$$-53 + 18 = -35$$

$$\begin{array}{r} 11001011 \\ + 00010010 \\ \hline \end{array}$$

**0** **0** **1**1011101      Rendben (végre valami...)

- **6. példa**

$$53 + -18 = 35$$

$$\begin{array}{r} 00110101 \\ + 11101110 \\ \hline \end{array}$$

**0** **1** **0**0100011      Jó, de C=1!

# Flag-ek viselkedése

- **7. példa**

$$-18 + 18 = 0$$

11101110

+ 00010010

-----

0 1 00000000

Jó, de C=1!

# Flag-ek viselkedése a példákban

- **Z (Zero) flag**
  - Csak a 7. példában 1, mert csak akkor 0 az eredmény. Más példákban 0.
- **S (Sign) flag**
  - Eredmény legmagasabb bitjének értéke
  - 2. és 4. példában nem a valódi előjelet adja! Belső átvitel elrontotta...
- **C (Carry) flag**
  - Legfelső biten keletkezett túlcordulás
  - Logikus értelmé csak azonos előjelű számok összeadásakor van a példákban: ekkor az eredmény helyes előjelét adja (**S** flag helyett...)
- **O (Overflow) flag**
  - Ha az eredmény értéke kívül esik az ábrázolható tartományon (itt: [-128,+127])
  - Összeadás után **O** alapján dönthető el, hogy az eredmény előjelhelyes-e?
    - Ha **O** flag értéke 1, akkor **C** a helyes előjel. (2. és 4. példa)
    - Ha **O** flag értéke 0, akkor **S** a helyes előjel. (A többi példában.)
- **P (Parity) flag**
  - 1-es értékű bitek paritása (számának párossága) az eredményben
  - Értéke 1 az 1., 2., 4., 5. és 7. példákban

# Flag-ek viselkedése

- **Konklúzió**

- **O**, **S** és **C** flag-ek alapján dönthetünk a helyes értékről
- Magasszintű nyelveket is érinti ez a probléma, de ott nincs lehetőségünk a **flag** értékek vizsgálatára!

- **Fontos**

- Olyan regiszter(eke)t célszerű választani, amelyben az eredmény is „elfér”
- Ha pl. előfordulhat, hogy az összeg kivezet a  $[-128,+127]$  tartományból, akkor használjunk 16 bites regisztereket

# Probléma magasszintű nyelvben

## C forráskód

```
#include <stdio.h>
int main(int argc, char *argv[])
{
 char a, b, c;
 a = 53;
 b = 83;
 c = a + b;
 printf("Osszeg: %d\n", (int)c);

 return 0;
}
```

## Eredmény

- -120 értéket kapunk!
- Az Assembly-vel ellentétben itt nem „látjuk” a flag-ek értékét a művelet elvégzése után!

# Vezérlésátadó utasítások

# Vezérlésátadás

- **Feltétel nélküli**
  - A program az utasításban megadott helyen folytatódik
  - **JMP, CALL**
- **Feltételes**
  - Ha a feltétel teljesül, akkor a címke által mutatott helyen, egyébként az elágazás utánin
  - **FLAGS** regiszter bit-értékeinek megfelelően
    - Számos utasítás az **S, Z, C, P, O** FLAG bitek, illetve az **ECX** regiszter vizsgálatával
    - **FLAG** biteket aritmetikai, logikai, bitforgató, flag beállító műveletek módosítanak, ezek után lehet feltételes ugrás
  - Ciklusszervezés a **LOOP** utasításcsaláddal
    - **ECX** és **FLAGS** bitek



# Vezérlésátadás (védett mód)

- **Direkt**
  - Az utasítás tartalmazza **EIP** és **CS** új értékét
- **EIP-relatív**
  - Előjeles egész érték hozzáadása az utasítás-számlálóhoz
  - Ha negatív, akkor „visszafelé” ugrunk
  - A szegmensen belül körkörös a címzés: ha a szegmens végén ugrunk előre, túl a szegmens határán, akkor a szegmens elején „találjuk magunkat”
- **Indirekt (regiszter vagy címzés)**
  - **EIP** új értéke egy regiszter értékét veszi fel
  - Címzéssel elért memóriaterület adja meg **EIP** és **CS** új értékeit (pl. ugrótábla)

# Vezérlésátadás (védett mód)

- **Rövid ugrás**
  - **EIP** relatív, 1 előjeles bájton ábrázolt távolságra
    - Szegmensen belül (**CS** változatlan)  
[−128 , +127] bájttartományon belül!
- **Közeli (**NEAR**) ugrás**
  - **EIP** relatív, 4 előjeles bájton ábrázolt távolságra
    - Szegmensen belül (**CS** változatlan)  
[−2GB , +2GB] bájttartományon belül!
- **Távoli (**FAR**) ugrás**
  - **CS** és **EIP** is változik, 6 bájttartomány szükséges

# Kódterület címzése (védett mód)

|                                      | Követlen (direkt)                     | EIP-relatív                                    | Regiszter indirekt               | Címzett indirekt                                                    | Megjegyzés                                        |
|--------------------------------------|---------------------------------------|------------------------------------------------|----------------------------------|---------------------------------------------------------------------|---------------------------------------------------|
| <b>Rövid ugrás</b>                   | ?                                     | -128, +127 közötti távolság,<br>EIP-hez adódik | -                                | -                                                                   | LOOP csak ilyen lehet!<br>Feltételes lehet ilyen. |
| <b>Közeli ugrás</b>                  | ?                                     | -2GB, +2GB közötti távolság,<br>EIP-hez adódik | JMP AX<br>Regiszter<br>-><br>EIP | JMP [DI+2]<br>Memóriatartalom<br>-><br>EIP                          | Feltételes elágazás lehet ilyen is.               |
| <b>Távoli (szegmens közti) ugrás</b> | Műveleti kód után EIP és CS új értéke | -                                              |                                  | JMP FAR PTR [AX]<br>memóriacímen található két szó kerül EIP, CS-be | Csak feltétel nélküli ugrás lehet.                |

Megjegyzések:

A szegmensek címzése „körkörös”, nincs túlcscordulás kezelés!

‘?’: Egyes könyvek állítják, hogy ilyen létezik, de nem írják le, hogyan érhető el

# Kódterület címzése (védett mód)

- **Rossz hír**
  - Ellentmondó szakirodalom
- **Jó hír**
  - Ezzel nem kell foglalkoznunk
  - Címkéket használva a fordító (assembler) feladata a megfelelő címzés kiválasztása és használata
    1. **Rövid IP-relatív ugrás** (2 bájt: **opkód + távolság**)
      - ha szegmensen belüli és a távolság -128 és +127 közötti,
      - vagy LOOP valamelyik változata
    2. **Közeli IP-relatív ugrás** (5 bájt: **opkód + távolság**)
      - ha nagyobb, de szegmensen belüli
    3. **Direkt ugrás** (7 bájt: **opkód + EIP + CS új értéke**)
      - szegmensen kívüli vagy FAR címkére történik (feltétel nélküli)

# Kódterület címzése

- **Főbb különbségek 8086 valós módban**
  - **EIP** helyett **IP**
  - A közeli ugrás csak 2 bájtos, [-32768, +32676] tartományban
  - A feltételes ugrások csak rövidek lehetnek!
    - [-128, +127] tartományban
    - Mint védett módban a **LOOP**

# Feltétel nélküli vezérlésátadás

- **Ugrás**
  - Rövid, közeli, vagy távoli címre (**NEAR, FAR**)
  - **JMP címke**
- **Eljárás**
  - Rövid, közeli, vagy távoli címre (**NEAR, FAR**)
  - **CALL címke**
    - Vezérlésátadás a címre, visszatérési cím a verembe
  - **RET**
    - Visszatérés a verem tetején található címre
  - **RET n**
    - Visszatérés, és n darab érték kivétele a veremből
    - Vermen keresztül átadott paraméterek takarítására
  - Célszerűbb eljárásokat használni ugrások helyett!

# Feltételes vezérlésátadás

- **FLAG** bitek vagy **ECX** regiszter vizsgálata
  - Ugrás, ha feltétel teljesül
  - Következő utasítás végrehajtása, ha nem

| Feltétel | Előjeles | Előjel nélküli | Flag/reg. | 0 értékű    | 1 értékű |
|----------|----------|----------------|-----------|-------------|----------|
| =        | JE, JZ   | JE, JZ         | O         | JNO         | JO       |
| ≠        | JNE, JNZ | JNE, JNZ       | P         | JNP, JPO    | JP, JPE  |
| >        | JG, JNLE | JA, JNBE       | S         | JNS         | JS       |
| ≥        | JGE, JNL | JAE, JNB       | C         | JNC         | JC       |
| <        | JL, JNGE | JB, JNAE       | Z         | JNZ, JNE    | JZ, JE   |
| ≤        | JLE, JNG | JBE, JNA       | CX        | JCXZ, JECXZ |          |

Jump, Not, Equals, Zero, Less than,  
Greater than, Above, Below

Jump, Overflow, Parity, Sign, Carry, Zero, CX,  
Not, Equals, Parity Even, Parity Odd, Zero

# Ciklusok szervezése

- **LOOP címke**
  - **ECX** számláló regiszter csökkentése 1 értékkel, ugrás ha még nem nulla az értéke
  - Egyébként következő utasítás jön
- **LOOPNZ , LOOPNE**
  - Ha **Z** FLAG értéke 0, akkor **ECX** csökkentése, ugrás ha az még nem nulla
  - Egyébként következő utasítás jön
- **LOOPZ , LOOPE**
  - Ha **Z** FLAG értéke 1, akkor **ECX** csökkentése, ugrás ha az még nem nulla
  - Egyébként következő utasítás jön



# Ciklusok szervezése

- **LOOP** további változatai
  - **LOOPW**
    - **CX** használata **ECX** helyett
  - **LOOPD**
    - **ECX** használata, megegyezik **LOOP**-pal
  - Fentiek **FLAG** bitekkel kombinált változatai
    - **LOOPEW, LOOPED, LOOPNEW, LOOPNED**
    - **LOOPZW, LOOPZD, LOOPNZW, LOOPNZD**

Kiegészítő anyag!  
Csak hogy illet is lássunk...

# Rövid ugrás kikerülése

- **Probléma**

- **LOOP** ciklus ugrások csak rövid távolságra használhatók (-128,+127 bájt között)

- **Megoldás**

**Cimke:**

```
; ... hosszú kód rész, > 128 bájt
LOOPCimke ; nem jó, túl nagy távolság!
DEC ECX
JNZ Cimke ; OK, mivel JNZ mehet közeli címre
; Folytatás
```

# Példaprogram #1

- **Adatszegmensben található tömbök értékeinek összeadása**
  - Konstans méretű és megadott kezdőértékű tömbök
  - Az eredmény ezekkel egyező méretű tömb, előre lefoglalt területen
  - (Rodek Lajos és Diós Gábor megoldása)

```
TITLE Vektorok összeadása adatterületen (32 bites)
```

```
INCLUDE Irvine32.inc
```

```
.data
```

```
ElemSzam EQU SIZEOF Vekt1 ; 5 elemű tömbök
Vekt1 DB 6,-1,17,100,-8 ; tömb értékek
Vekt2 DW 1000,1999,-32768,4,32767 ; eltérő típusok!
Eredm DW ElemSzam DUP (?) ; helyfoglalás eredménynek
```

```
.code
```

```
PRGSTART:
```

```
MOV ECX,ElemSzam ; Elemszám számláló reg.-be
XOR EBX,EBX ; EBX nullázása
MOV ESI,EBX ; ESI nullázása
```

```
Ciklus:
```

```
MOV AL,[Vekt1+EBX] ; AL-be Vekt1 értéke
CBW ; Előjelhelyes kiterjesztés AL->AX
ADD AX,[Vekt2+ESI] ; Vekt2 elemének hozzáadása
MOV [Eredm+ESI],AX ; Összeg tárolása
INC EBX ; EBX növelése (DB adat!)
LEA ESI,[ESI+2] ; ESI = ESI + 2 ! (DW adat!)
LOOP Ciklus ; ECX csökkentése, ugrás ha > 0
```

```
; Eredm memóriatartalmának kiírása konzolra (Irvine)
```

```
MOV ESI, OFFSET Eredm ; starting offset
MOV ECX, LENGTHOF Eredm ; # of units
MOV EBX, TYPE Eredm ; double format
CALL DumpMem
; Eredm: 03EE 07CE 8011 0068 7FF7
; (decimális 1006, 1998, -32751, 104, 32759)
```

```
exit
```

```
END PRGSTART
```

**Alternatívák ESI  
növelésére:**

**ADD ESI, 2**

- OK, de **FLAG** regiszter bitek is állítódnak!

**INC ESI**

**INC ESI**

- Két utasítás 1 helyett

**LEA ESI, [ESI+2]**

- op2 offset címének betöltése
- 1 utasítás, nincs FLAG állítás
- Összetettebb címezésnél is használható!

# Példaprogram #2

- **Adatszegmensben található karakter tömb megfordítása verem segítségével**
  - A sztring végjelét mi definiáljuk
  - Végjelig olvasunk, az értékek verembe kerülnek
  - Ha elértük a végjelet, akkor kivesszük sorban az elemeket a veremből és a kezdeti memóriaterületre tesszük

TITLE Szófordító veremmel (32 bites)

INCLUDE Irvine32.inc

.data

VEGJEL DB '\$'  
SZO DB 'Ez a szoveg\$'

.code

PRGSTART:

MOV EBX,OFFSET SZO ; szó offszet címe EBX regiszterbe  
MOV ESI,0 ; ESI nullázása  
MOV AH,0 ; AH nullázása (szöveg bájtos, verembe AX tehető)

PAKOL:

MOV AL,[ESI+EBX] ; aktuális karakter AL-be töltése  
CMP AL,VEGJEL ; összehasonlítás végjel karakterrel  
JE PAKOLVEGE ; ha elértük, akkor lépünk ki a ciklusból  
PUSH AX ; karakter verembe kerül  
INC ESI ; következő karakterre pozicionálunk  
JMP PAKOL ; ciklusmag újbóli feltétel nélküli végrehajtása

PAKOLVEGE:

CMP ESI,0 ; ha csak végjel volt  
JE KILEP ; kilépés  
MOV ECX,ESI ; a sztring megszámlolt hossza ECX-be  
XOR ESI,ESI ; ESI nullázása: sztring elejére állás, EBX változatlan

FORDIT:

POP AX ; utolsó karakter kivétele a veremből  
MOV [ESI+EBX],AL ; berakása az elején következő helyre  
INC ESI ; következő karakterpozícióra állás  
LOOP FORDIT ; CX csökkentése, ha még nem nulla, akkor ciklusmag újra

KILEP:

exit ; kilépés

END PRGSTART

# Példaprogram #2

- **Veszélyforrások!**
  - **Ha a sztring végéről lefelejtjük a végjelet**
    - A verembe pakolás folytatódik a verem- és kódszegmens területekről is
    - A verem túlcsordulhat
    - Kódterület felülírásra kerülhet
  - **Sztring hossza nagyobb mint a verem mérete**
    - Betelik a verem, a szegmens végétől felülírja az ott lévő tartalmat
- **Eredmény**
  - Védelmi hibát okoz(hat) a program!
  - Ennek oka, hogy olyan területre is írunk, ahová nincs jogosultságunk

# Paraméterátadás eljáráshíváskor

## Rekurzív, re-entráns eljárások



# Paraméterátadás eljárásnak

- **Technika**

- **Magasszintű nyelvek**

- Formális paraméterlista; rögzített mechanizmus (automatikusan)

- **Assembly**

- Átadás módja nincs szabályozva, a programozónak kell gondoskodnia róla
    - Magasszintű nyelvekkel kombinált Assembly program esetén a másik nyelv hívási mechanizmusát kell használni az Assembly részben

# Paraméterátadás eljárásnak

- **Történhet**
  - **Adat területen keresztül**
  - **Regiszter(ek)en keresztül**
    - Változó értékek
    - Memóriaterületek címei
    - Probléma: globális változók, rekurzió támogatása nem megoldott
  - **Verem keresztül**
    - Változók értékei, címek a verembe kerülnek
    - Bázis-relatív címezéssel elérhető az eljárásban
    - Rekurzió esetén saját lokális változókészlet használható

# Paraméterátadás eljárásnak

- **Összeadás példa**

- **Medoldások**

- Eljárás nélkül
    - Eljárással, adat szegmensben levő adatok összeadása
    - Paraméterek átadása regiszterekben
    - Paraméter átadás veremben

```
TITLE Összeadás közvetlenül (32 bites)
```

```
INCLUDE Irvine32.inc
```

```
.data
```

```
 a_val DD 5 ; 32-bites egészek
 b_val DD 7
 c_val DD ?
```

```
.code
```

```
_Main PROC
```

```
 MOV EAX,b_val ; Paraméterek a verembe ...
 MOV EBX,a_val ; ... fordított sorrendben
 ADD EAX, EBX
 MOV c_val, EAX ; ... fordított sorrendben
```

```
 CALL WriteDec ; Konzolra írás
```

```
 INVOKE ExitProcess,0
```

```
_Main ENDP
```

```
END _Main
```

```
#include <stdio.h>

int a_val, b_val, c_val;

int main(int argc , char **argv)
{
 a_val = 5;
 b_val = 7;
 c_val = a_val + b_val;

 printf("Összeg: %d\n", c_val);

 return 0;
}
```

# Paraméterek regiszterekben

- **Probléma**

- Csak *A\_val* és *B\_val* összegét tudja kiszámítani!

- **Megoldás**

- Az eljárás paraméterként kapja meg a **két értéket**, így tetszőleges számok összegét számítani tudja.

- A paramétereket két regiszterben (**EAX,EBX**) adjuk át.

TITLE Érték szerinti paraméterátadás (32 bites)

INCLUDE Irvine32.inc

.data

```
a_val DD 5 ; 32-bites egészek
b_val DD 7
c_val DD ?
```

.code

\_Main PROC

```
MOV EAX,b_val ; Paraméterek a verembe ...
MOV EAX,a_val ; ... fordított sorrendben
CALL Sum ; Függvényhívás eredmény az AX-ben
MOV c_val,EAX ; Adatterületre írása

CALL WriteDec ; Konzolra írás
INVOKE ExitProcess,0
```

\_Main ENDP

Sum PROC NEAR ; Közeleli hívás, 4 bájtos visszatérési cím

```
ADD EAX, EBX ; b_val hozzáadása
```

```
RET ; Hívó fél takarít majd!
```

Sum ENDP

END \_Main

```
#include <stdio.h>

int a_val, b_val, c_val;

int main(int argc , char **argv)
{
 a_val = 5;
 b_val = 7;
 c_val = sum(a_val, b_val);

 printf("Osszeg: %d\n", c_val);

 return 0;
}
```

Vagy legalábbis valami hasonló.

C-ben nincsenek regiszterek, ennek az assembly programnak igazából nincs C-s megfelelője.

# Paraméterek a veremben

- **Probléma**

- Regisztereken keresztül csak limitált számú paraméter adható át
- Rekurzív és re-entráns eljárások hívásakor új paraméterkészlet átadása szükséges

- **Megoldás**

- A paramétereket egy memóriaterületen helyezük el és annak a címét adjuk át
- A **paramétereket helyezük el a veremben**, ahol a meghívott eljárás eléri

```
TITLE Érték szerinti paraméterátadás (32 bites)
```

```
INCLUDE Irvine32.inc
```

```
.data
```

```
 a_val DD 5 ; 32-bites egészek
 b_val DD 7
 c_val DD ?
```

```
.code
```

```
_Main PROC
```

```
 PUSH 0 ; Helyfoglalás a függvény eredményének
 MOV EAX,b_val ; Paraméterek a verembe ...
 PUSH EAX ; ... érték szerint ...
 MOV EAX,a_val ; ... fordított sorrendben
 PUSH EAX
 CALL Sum ; Függvényhívás
 ADD ESP,8 ; Paraméterek takarítása a veremből
 POP EAX ; Eredmény kiolvasása
 MOV c_val,EAX ; Adatterületre írása
 CALL WriteDec ; Konzolra írás
 INVOKE ExitProcess,0
```

```
_Main ENDP
```

```
Sum PROC NEAR ; Közeleli hívás, 4 bájtos visszatérési cím
```

```
 PUSH EBP ; EBP mentése ...
 MOV EBP,ESP ; ... és beállítása
 MOV EAX,[EBP+8] ; a_val értéke
 ADD EAX,[EBP+12] ; b_val hozzáadása

 MOV [EBP+16],EAX ; eredmény eltárolása

 MOV ESP,EBP ; Regiszterértékek ...
 POP EBP ; ... visszaállítása
 RET ; Hívó fél takarít majd!
```

```
Sum ENDP
```

```
END _Main
```

```
#include <stdio.h>
```

```
int a_val, b_val, c_val;
```

```
int sum(int, int); /* prototípus */
```

```
int main(int argc , char **argv)
```

```
{
 a_val = 5;
 b_val = 7;
 c_val = sum(a_val, b_val);

 printf("Osszeg: %d\n", c_val);

 return 0;
}
```

```
int sum(int a, int b)
```

```
{
 AX = a + b;
 return AX;
}
```



# Eljáráshívási konvenciók

- **Célszerű egységesen kezelni**
  - Különösen publikus DLL-ek, lib-ek készítésekor!
- **Kérdések**
  - Regiszterekben vagy vermen keresztül?
  - Ki takarítsa ki a vermet?
    - *Hívó vagy hívott?*
- **32-bites MASM assemblernek jelezhetjük a forráskódban**
  - `.model` direktívával, pl.:  
`.model flat, stdcall`

# Eljáráshívási konvenciók

- **cdecl**

- C programozási nyelv elvét követi
- A **hívó** a verembe teszi a paramétereket
  - Fordított sorrendben (hátról előre)
- Visszatérés után a **hívó takarít**
- Így valószínűleg meg hatékonyan változó számú paraméterlista használata (ld. C nyelv, **printf()**)

**f(x, y);**

**push y**

**push x**

**call f**

# Eljáráshívási konvenciók

- **stdcall**

- Win32 rendszerhívások ezt a konvenciót használják
- A **hívó** a verembe teszi a paramétereket
- A **hívott fél** takarítja a vermet

- **fastcall**

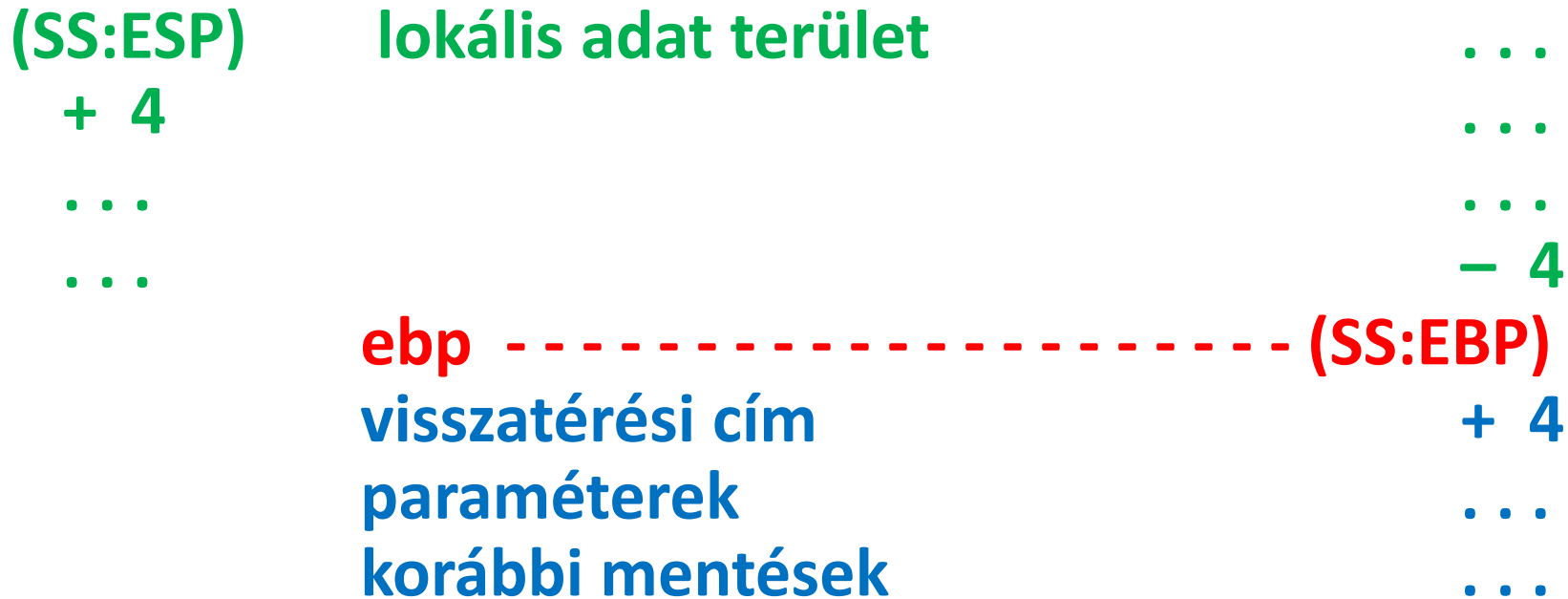
- x64 esetén alapértelmezett
- Az első hat paraméter regisztereken keresztül
  - **RDI, RSI, RDX, RCX, R8, R9**
- A többi a vermen keresztül, ha van
- Lebegőpontos paraméterek száma
  - **RAX** regiszterben

# Lokális adat terület

- Ha egy eljárás működéséhez **lokális adatterületre, munkaterületre** van szükség, és a működés befejeztével a munkaterület tartalma felesleges, akkor a munkaterületet célszerűen a **veremben** alakíthatjuk ki.
- A munkaterület **lefoglalásának ajánlott módja**

```
. . . proc . . .
PUSH BP ; BP értékének mentése
MOV BP,SP ; BP ← SP,
 ; a stack relatív címzéshez
SUB SP,n ; n byte-os munkaterület lefoglalása
. . . ; további regiszter mentések
```

## Lokális adat terület (NEAR eljárás esetén)



A **munkaterület** negatív eltolási érték mellett verem relatív címezéssel érhető el. (A **veremben** átadott **paraméterek** ugyancsak verem relatív címezéssel, de pozitív eltolási érték mellett érhetőek el.)

- A munkaterület felszabadítása visszatéréskor a

```
. . . ; visszaállítások
MOV ESP,EBP ; a munkaterület felszabadítása
POP EBP ; BP értékének visszaállítása
ret ; visszatérés
```

utasításokkal történhet.

# Két egész érték összeadása függvénnyel, lokális változóval

- C programozási nyelven, érték szerinti átadás

```
#include <stdio.h>

int a_val, b_val, c_val;

int sum(int a, int b)
{
 int temp;
 temp = a + b;
 return temp;
}

int main(int argc , char **argv)
{
 a_val = 5;
 b_val = 7;
 c_val = sum(a_val, b_val); /* Változók értékének átadása */

 printf("Osszeg: %d\n", c_val);

 return 0;
}
```

- **temp** változó használata felesleges a **sum()** függvényben
  - Lehetne egyszerűen **return a + b;** a függvény törzse
- Nem kell foglalkoznunk a paraméterátadás módjával!

```
TITLE Érték szerinti paraméterátadás (32 bites)
```

```
INCLUDE Irvine32.inc
```

```
.data
```

```
 a_val DD 5 ; 32-bites egészek
 b_val DD 7
 c_val DD ?
```

```
.code
```

```
__Main PROC
```

```
 PUSH 0 ; Helyfoglalás a függvény eredményének
 MOV EAX,b_val ; Paraméterek a verembe ...
 PUSH EAX ; ... érték szerint ...
 MOV EAX,a_val ; ... fordított sorrendben
 PUSH EAX
 CALL Sum ; Függvényhívás
 ADD ESP,8 ; Paraméterek takarítása a veremből
 POP EAX ; Eredmény kiolvasása
 MOV c_val,EAX ; Adatterületre írása
 CALL WriteDec ; Konzolra írás
 INVOKE ExitProcess,0
```

```
__Main ENDP
```

```
Sum PROC NEAR ; Közeli hívás, 4 bajtos visszatérési cím
```

```
 PUSH EBP ; EBP mentése ...
 MOV EBP,ESP ; ... és beállítása
 SUB ESP,4 ; Helyfoglalás 1 32-bites lokális változónak
 MOV EAX,[EBP+8] ; a_val értéke
 ADD EAX,[EBP+12] ; b_val hozzáadása
 MOV [EBP-4],EAX ; eredmény lokális változóba

 MOV EAX,[EBP-4] ; eredmény visszaadása
 MOV [EBP+16],EAX ; eredmény eltárolása

 MOV ESP,EBP ; Regiszterértékek ...
 POP EBP ; ... visszaállítása
 RET
```

```
Sum ENDP
```

```
END __Main
```

```
#include <stdio.h>

int a_val, b_val, c_val;

int sum(int, int); /* prototípus */

int main(int argc , char **argv)
{
 a_val = 5;
 b_val = 7;
 c_val = sum(a_val, b_val);

 printf("Osszeg: %d\n", c_val);

 return 0;
}

int sum(int a, int b)
{
 int temp;
 temp = a + b;
 return temp;
}
```

Jól látható, hogy a lokális változóra itt nincs szükség, dolgozhattunk volna közvetlenül az `[EBP+16]` címmel.

A C függvényhívás mechanizmusának bemutatása miatt maradt benne.



- C programozási nyelven, cím szerinti átadás

```
#include <stdio.h>

int a_val, b_val, c_val;

void sum(int *a, int *b, int *c)
{
 (*c) = (*a) + (*b); /* c_val közvetlenül kap értéket címén keresztül! */
}

int main(int argc , char **argv)
{
 a_val = 5;
 b_val = 7;
 sum(&a_val, &b_val, &c_val); /* Változók címének átadása */

 printf("Osszeg: %d\n", c_val);

 return 0;
}
```

```
TITLE Cím szerinti paraméterátadás (32 bites)
```

```
INCLUDE Irvine32.inc
```

```
.data
```

```
 a_val DD 5 ; 32-bites egészek
 b_val DD 7
 c_val DD ?
```

```
.code
```

```
_Main PROC
```

```
 MOV EBX,OFFSET c_val ; Paraméterek a verembe ...
 PUSH EBX
 MOV EAX,OFFSET b_val ; ... cím szerint ...
 PUSH EAX
 MOV EAX,OFFSET a_val ; ... fordított sorrendben
 PUSH EAX
 CALL Sum ; Függvényhívás
 ADD ESP,12 ; Paraméterek takarítása a veremből
 MOV EAX,c_val ; Eredmény beolvasása ...
 CALL WriteDec ; ... és konzolra írása
 INVOKE ExitProcess,0
```

```
_Main ENDP
```

```
Sum PROC NEAR ; Közeli hívás, 4 bájtos visszatérési cím
```

```
 PUSH EBP ; EBP mentése ...
 MOV EBP,ESP ; ... és beállítása
 MOV EBX,[EBP+8] ; a_val címe
 MOV EAX,[EBX] ; a_val beolvasása
 MOV EBX,[EBP+12] ; b_val címe
 MOV EBX,[EBP+16] ; c_val címe
 MOV [EBX],EAX ; eredmény eltárolása
 MOV ESP,EBP
 POP EBP
 RET ; Hívó fél takarít majd!
```

```
Sum ENDP
```

```
END _Main
```

```
#include <stdio.h>

int a_val, b_val, c_val;

void sum(int *, int *, int *);

int main(int argc , char **argv)
{
 a_val = 5;
 b_val = 7;
 sum(&a_val, &b_val, &c_val);

 printf("Osszeg: %d\n", c_val);

 return 0;
}

void sum(int *a, int *b, int *c)
{
 (*c) = (*a) + (*b);
}
```

A lokális változót itt nem hoztuk létre, közvetlenül `c_val` kap értéket.

A függvényben lehetőségünk lenne az `a_val` és `b_val` értékek megváltoztatására is, ha a függvény működése megkívánná.

# PROTO, PROC, INVOKE

- **Paraméterátadás egyszerűsített szintaktikája**
  - A veremkezelő kódot nem mi írjuk, a fordítóprogram generálja!
  - Paramétertípusok megadása és ellenőrzése!
- **PROTO**
  - Eljárás nevének, paramétereinek típusának megadása
  - Az eljárás hívása előtt derülni kell ki (definíció vagy deklaráció)
- **PROC**
  - Megadható, hogy
    - mely regiszterek tartalma kerüljön mentésre és visszaállításra (**USES**);
    - eljárás milyen típusú paramétereket vár, milyen sorrendben;
    - paraméterekhez szimbólumnevek társíthatók (**EBP**-relatív címzés helyett).
- **INVOKE**
  - Paraméterek verembe helyezése
    - **C** és **stdcall** típus esetén jobbról balra haladva!
  - Paraméter típusok ellenőrzése / konverziója
    - Nem kompatibilis típusok esetén fordítási hiba!
  - Verem takarítása eljárás végén

- C programozási nyelven, érték és cím szerinti átadás vegyesen

```
#include <stdio.h>

int a_val, b_val, c_val;

void sum(int a_value, int b_value, int *c_offset)
{
 (*c_offset) = a_value + b_value; /* c_val közvetlenül kap értéket címén keresztül! */
}

int main(int argc , char **argv)
{
 a_val = 5;
 b_val = 7;
 sum(a_val, b_val, &c_val); /* Változók értéknek és c_val címének átadása */

 printf("Osszeg: %d\n", c_val);

 return 0;
}
```

TITLE Érték és cím szerinti paraméterátadás (32 bites)

INCLUDE Irvine32.inc

.data

```

a_val DD 5 ; 32-bites egészek
b_val DD 7
c_val DD ?

```

.code

**\_Main** PROC

```

MOV EBX,OFFSET c_val ; Eredmény címe ...
PUSH EBX ; ... verembe
MOV EAX,b_val ; Paraméterek a verembe ...
PUSH EAX ; ... érték szerint ...
MOV EAX,a_val ; ... fordított sorrendben
PUSH EAX
CALL Sum ; Függvényhívás
ADD ESP,12 ; Paraméterek takarítása a veremből
MOV EAX,c_val ; Eredmény beolvasása ...
CALL WriteDec ; ... és konzolra írása
INVOKE ExitProcess,0

```

**\_Main** ENDP

**Sum** PROC NEAR ; Közeleli hívás, 4 bájtos visszatérési cím

```

PUSH EBP ; EBP mentése ...
MOV EBP,ESP ; ... és beállítása
MOV EAX,[EBP+8] ; a_val értéke
ADD EAX,[EBP+12] ; b_val hozzáadva
MOV EBX,[EBP+16] ; c_val címe
MOV [EBX],EAX ; eredmény paraméterként kapott címre
MOV ESP,EBP
POP EBP
RET ; Hívó fél takarít majd!

```

**Sum** ENDP

END **\_Main**

```

#include <stdio.h>

int a_val, b_val, c_val;

void sum(int, int, int *);

int main(int argc , char **argv)
{
 a_val = 5;
 b_val = 7;
 sum(a_val, b_val, &c_val);

 printf("Osszeg: %d\n", c_val);

 return 0;
}

void sum(int a_value, int b_value,
int *c_offset)
{
 (*c_offset) = a_value + b_value;
}

```

A lokális változót itt nem hoztuk létre,  
közvetlenül **c\_val** kap értéket.

```
TITLE Érték és cím szerinti paraméterátadás + INVOKE (32 bites)
```

```
INCLUDE Irvine32.inc
```

```
Sum PROTO NEAR C, a_value:DWORD, b_value:DWORD, c_offset:NEAR PTR DWORD;
```

```
.data
```

```
 a_val DD 5 ; 32-bites egészek
 b_val DD 7
 c_val DD ?
```

```
.code
```

```
_Main PROC
```

```

; MOV EBX,OFFSET c_val ; Eredmény címe ...
; PUSH EBX ; ... verembe
; MOV EAX,b_val ; Paraméterek a verembe ...
; PUSH EAX ; ... érték szerint ...
; MOV EAX,a_val ; ... fordított sorrendben
; PUSH EAX
; CALL Sum ; Függvényhívás
 INVOKE Sum, a_val, b_val, offset c_val
 ADD ESP,12 ; Paraméterek takarítása a veremből
 MOV EAX,c_val ; Eredmény beolvasása ...
 CALL WriteDec ; ... és konzolra írása
 INVOKE ExitProcess,0 ; Kilépés

```

```
_Main ENDP
```

```
Sum PROC NEAR C USES EAX EBX, a_value:DWORD, b_value:DWORD, c_offset:NEAR PTR DWORD
```

```

; PUSH EBP ; EBP mentése ...
; MOV EBP,ESP ; ... és beállítása
 MOV EAX,a_value ; a_val értéke
 ADD EAX,b_value ; b_val hozzáadva
 MOV EBX,c_offset ; c_val címe
 MOV [EBX],EAX ; eredmény paraméterként kapott címre
; MOV ESP,EBP
; POP EBP
 RET ; Hívó fél takarít majd!

```

```
Sum ENDP
```

```
END _Main
```

```

#include <stdio.h>

int a_val, b_val, c_val;

void sum(int, int, int *);

int main(int argc , char **argv)
{
 a_val = 5;
 b_val = 7;
 sum(a_val, b_val, &c_val);

 printf("Osszeg: %d\n", c_val);

 return 0;
}

void sum(int a_value, int b_value,
int *c_offset)
{
 (*c_offset) = a_value + b_value;
}

```

**PROTO, PROC és INVOKE elintézi a paraméterátadás részleteit! Ezek használatával a fordító generálja le a szükséges kódrészeket!**

**Cím átadása esetén a címzésről nekünk kell gondoskodni!**

**MOV [c\_offset],EAX hibás!**

**EAX és EBX mentésre és visszaállításra kerül.**

# LOCAL

- **Lokális változók használatához**
  - **PROC** direktíva után közvetlenül adhatók meg
  - Név és típus rendelhető hozzájuk
    - Ezek használatát a fordító ellenőrzi
  - Veremkezelő kód írása elkerülhető
    - Vagyis nem kell
      - **SUB ESP, 4**
      - **[EBP-4]** címzés
      - takarítás a végén

```
TITLE Értékek megcserélése + lokális változó + INVOKE (32 bites)
```

```
INCLUDE Irvine32.inc
```

```
Swap PROTO NEAR C, a_offset:NEAR PTR DWORD, b_offset:NEAR PTR DWORD
```

```
.data
```

```
 a_val DD 5 ; 32-bites egészek
```

```
 b_val DD 7
```

```
.code
```

```
_Main PROC
```

```
 INVOKE Swap, offset a_val, offset b_val
```

```
 ADD ESP,8 ; Paraméterek takarítása a veremből
```

```
 MOV EAX,a_val ; Eredmény beolvasása
```

```
 CALL WriteDec ; Eredmény kiírása
```

```
 CALL CrLf
```

```
 MOV EAX,b_val
```

```
 CALL WriteDec
```

```
 INVOKE ExitProcess,0
```

```
_MAIN ENDP
```

```
Swap PROC NEAR C, a_offset:NEAR PTR DWORD, b_offset:NEAR PTR DWORD
```

```
 LOCAL Temp:DWORD ; Lokális változó
 ; SUB ESP,4 és [EBP-4] címzés helyett
```

```
 MOV ESI,a_offset ; Címzés előkészítése
```

```
 MOV EDI,b_offset
```

```
 MOV EAX,[ESI] ; Temp = a_val
```

```
 MOV Temp,EAX
```

```
 MOV EAX,[EDI] ; a_val = b_val
```

```
 MOV [ESI],EAX
```

```
 MOV EAX,Temp ; b_val = Temp
```

```
 MOV [EDI],EAX
```

```
 RET
```

```
Swap ENDP
```

```
END _MAIN
```

```
#include <stdio.h>

int a_val, b_val;

void swap(int *, int *);

int main(int argc , char **argv)
{
 a_val = 5;
 b_val = 7;
 swap(&a_val, &b_val);

 printf("Csere: %d %d\n", a_val,
b_val);

 return 0;
}

void swap(int *a_offset, int
*b_offset)
{
 int Temp;
 Temp = (*a_offset);
 (*a_offset) = (*b_offset);
 (*b_offset) = Temp;
}
```

A lokális változót a **LOCAL** direktívával készíthetünk a veremben. Így nem kell a **SUB ESP, 4** előkészítés, az **[EBP-4]** címzés, valamint a felszabadítás a végén. Ezeket a fordító generálja helyettünk! Megjegyzés: Temp változóra itt nincs igazán szükség, Assembly nyelven regiszterben megoldható lenne a csere.



# Rekurzív és re-entráns eljárások

- **Rekurzív**

- Ha önmagát hívja közvetlenül, vagy más eljárásokon keresztül

- **Re-entráns**

- Ha többszöri belépést tesz lehetővé, ami azt jelenti, hogy az eljárás még nem fejeződött be, amikor újra hívható

- A rekurzív is egyfajta re-entráns
    - A re-entráns nem feltétlenül rekurzív!

# Re-entráns eljárások

- A rekurzív eljárással szemben a különbség az, hogy a **rekurzív eljárásban „programozott”**, hogy mikor történik az eljárás újra hívása, **re-entráns eljárás** esetén az újra **hívás ideje „kiszámíthatatlan”**
  - Példák
    - Többfeladatos operációs rendszerek esetén „párhuzamosan” futó alkalmazások hívhatják ugyanazt a rendszerfüggvényt.
    - Egy eljárást meghívhat a programunk és egy általunk kezelt megszakítási rutin is.
      - A programból meghívott eljárás futása bármikor megszakítható és a megszakítási rutin meghívhatja újra.
  - Emiatt **az eljárás minden változóját minden híváskor lokálisan célszerű létrehozni**, a munkaterületek keveredésének elkerülése érdekében!

# Re-entráns eljárások

- **Munkaterületek összekeveredése ellen**
  - Ne legyen önmódosító a program
  - Minden regiszter értéket menteni és visszaállítani kell
  - A paramétereket a vermen keresztül célszerű fogadni és visszaadni
  - A lokális változókat a vermen célszerű létrehozni
  - Processzusok között megosztott kód esetén a processzusok egymástól független, önálló verem szegmensekkel rendelkeznek

# Példa rekurzióra

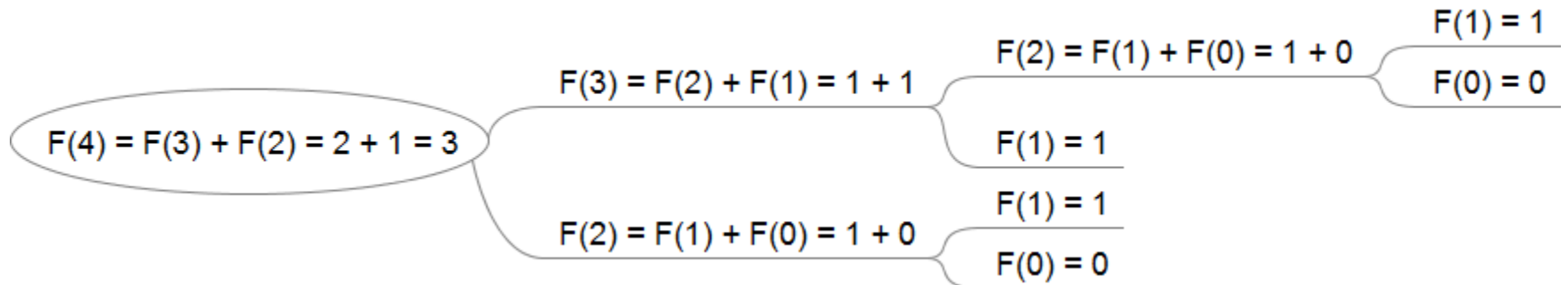
- **Fibonacci sorozat**

- **Rekurzív definíció**

- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$

- **Első néhány eleme**

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368



```
TITLE Fibonacci number
```

```
INCLUDE Irvine32.inc
```

```
.data
```

```
input DB "Which Fibonacci-number do you want to compute: ", 0
```

```
error DB "A positive integer should be specified!", 0
```

```
info DB "Requested Fibonacci-number: ", 0
```

```
text DB "Result: ", 0
```

```
.code
```

```
MAIN PROC
```

```
ELOLROL:
```

```
 ; egész szám beolvasása EAX regiszterbe
```

```
 MOV EDX,OFFSET input
```

```
 CALL WriteString ; Szöveg szám bekéréséhez
```

```
 CALL ReadInt ; Szám beolvasása billentyűzetről
```

```
 ; ellenőrzés
```

```
 CMP EAX,0
```

```
 JG COMPUTE
```

```
 ; hibaüzenet, újratezdés
```

```
 MOV EDX,OFFSET error
```

```
 CALL WriteString ; Hibaüzenet a konzolra
```

```
 CALL CrLf ; Soremelés
```

```
 JMP ELOLROL
```

COMPUTE:

```
; info szöveg kiírása
MOV EDX,OFFSET info
CALL WriteString
CALL WriteDec ; Beolvasott szám kiírása
CALL CrLf

; rekurzív eljárás meghívása
CALL FIBO
; eredmény EAX-be
MOV EAX,EDX

; eredménykiírás
MOV EDX,OFFSET text
CALL WriteString
CALL WriteDec ; Eredmény mint decimális szám
CALL CrLf

;; kilépés
CALL WaitMsg ; Billentyűlenyomásig vár
exit

MAIN ENDP
```

**FIBO PROC NEAR** ; Bemenet EAX-ben, eredmény EDX-ben

```

 CMP EAX,0
 JNE TOVABB1
 MOV EDX,0 ; F(0) = 0
 JMP KILEP

```

TOVABB1:

```

 CMP EAX,1
 JNE TOVABB2
 MOV EDX,1 ; F(1) = 1
 JMP KILEP

```

TOVABB2:

```

 PUSH EAX ; n értékének mentése
 SUB EAX,1 ; n-1
 CALL FIBO ; F(n-1) számítása

```

```

V1: POP EAX ; n elővétele veremből
 PUSH EDX ; F(n-1) részeredmény verembe
 SUB EAX,2 ; n-2

```

```

 CALL FIBO ; F(n-2) számítása

```

```

V2: POP EAX ; F(n-1) részeredmény elővétele
 ADD EDX,EAX ; F(n) = F(n-1) + F(n-2)

```

KILEP:

```

 RET ; Eredmény EDX regiszterben

```

**FIBO ENDP**

**END MAIN**

# Fibonacci sorozat

- **Feladat**

- Verem tartalmának követése különböző  $F(n)$  értékek számításakor