

# Kitlei Róbert

## Assembly programozás

Lektorálta: Dévai Gergely

ELTE, 2007

# Tartalomjegyzék

Bevezető.....	3
Alapvető adatelemek.....	4
Logikai értékek.....	4
Előjel nélküli egész számok.....	4
Előjeles egész számok.....	5
Törtszámok.....	5
Karakterek, stringek.....	6
Feladatok.....	7
Az architektúra alapjai.....	8
A memória szervezése.....	8
A regiszterek.....	10
Az utasítások szerkezete.....	10
Konstansok.....	11
A memória címzése.....	11
Adatmozgatás.....	12
Logikai utasítások.....	12
Bitléptető és bitforgató utasítások.....	12
Aritmetikai utasítások.....	13
Vezérlésátadás.....	13
Programozási konstrukciók megvalósítása.....	14
Változók deklarációja, definíciója.....	14
Értékadás.....	14
Goto.....	14
Kifejezések.....	15
Szekvencia.....	16
Elágazás.....	16
Rövidzárás operátorok.....	17
Elöltesztelési ciklus.....	18
Hátultesztelési ciklus.....	19
Rögzített lépésszámú ciklus.....	20
Leszámláló ciklus.....	21
Feladatok.....	22
A futási idejű verem szerepe: rekurzív alprogramhívások megvalósítása.....	25
A futási idejű verem használata.....	25
Konkrét példa a futási idejű verem használatára: a faktoriális( 2 ) hívás megvalósítása.....	26
Konkrét példa a futási idejű verem használatára: a faktoriális függvény kódja.....	26
A veremkeret általános szerkezete.....	26
Parancssori paraméterátadás.....	27
A rendszerszolgáltatások elérése: fájlkezelés.....	28
Feladatok.....	29
Makrók.....	31
Egysoros makrók.....	31
Többsoros makrók.....	33
Feltételes fordítás makrókkal.....	35
Feladatok.....	37
A program fordításának menete.....	39
A gépi kód szerkezetéről.....	40
CISC-elvű architektúrák.....	41
RISC architektúrák.....	41
Listafájl.....	42
Hibakeresés a programban.....	43
Gyakran előforduló hibák.....	45
A feladatok megoldásai.....	49
Értékek ábrázolása.....	49
Utasítások.....	50
Parancssori paraméterátadás, rendszerszolgáltatások.....	57
Makrók.....	60
Irodalom.....	62

## Bevezető

Az **assembly** olyan nyelvek gyűjtőneve, amelyek segítségével számítógépes programok alacsony szintű, közvetlenül a processzor által értelmezhető működése írható le emberek számára olvasható, szöveges formában. Az egyes számítógép-architektúrákon<sup>1</sup> található assembly nyelvek nagymértékben különböznek, mivel az architektúrák hardveres felépítése is eltérő. A számítógép-családok általában az architektúra minden előírását megtartják a későbbi modellekben is – azaz visszafelé kompatibilisek –, emiatt a korábbi modellekre írt programok változtatás nélkül futtathatóak a későbbiekben is.

A munkafüzet főleg az **x86 architektúra** assembly nyelvével foglalkozik, azon belül is a 32 bites szervezésű processzorokéval (ezt szokták IA-32 vagy x86-32 architektúrának is nevezni). Az architektúrába tartozó legismertebb gépek a 8086, a 80386 (rövidebben 386-os) és a Pentium. Napjainkban az architektúrába tartozó processzorok két legnépszerűbb gyártója az AMD és az Intel. A gépek elterjedtsége miatt a munkafüzetben leírtak könnyen kipróbálhatóak a gyakorlatban is.

Bizonyos esetekben elkerülhetetlen, hogy az assembly programozás során igénybe vegyünk az **operációs rendszer** szolgáltatásait. Assembly szintről ezek eltérően kezelendők, ezért itt is választanunk kell. A munkafüzet a **Linux** rendszerek néhány fontos szolgáltatását írja le. Ez a választás abból a szempontból nem elsőrendű, hogy az architektúra határozza meg az assembly nyelv jellegét, azonban ha azt szeretnénk, hogy működjenek is a programjaink, például tudjunk a képernyőre írni, ismernünk kell ennek is a módját, ez pedig az operációs rendszeren múlik.

Ha eldöntöttük, melyik **platformra** (architektúrára, azon belül pedig operációs rendszerre) szeretnénk programokat fejleszteni, ki kell választanunk, melyik **assemblerrel** és **szerkesztőprogrammal (linkerrel)** dolgozzunk. Az assembler az a program, ami az assembly nyelvű forráskódot egy közbülső formátumra, **tárgykódra** fordítja, a szerkesztő pedig egy vagy több tárgykódból előállítja a gép által közvetlenül értelmezhető **futtatható állományt**. A legtöbb platformon több assembler és linker is elérhető. A munkafüzetben részletesen a **nasm** assemblerrel és a **gcc** szerkesztőprogrammal fogunk foglalkozni. A nasm fontos tulajdonságai, hogy a benne írt forráskód szintaxisa könnyen olvasható, az assembler maga pedig ingyenesen elérhető több platformra is. A gcc-t mint szerkesztőprogramot szintén elterjedtsége miatt választottuk.

A munkafüzet célja az assembly általános jellegének áttekintése egy adott assembly nyelv és környezet segítségével. Ennek érdekében a munkafüzet sok egyszerűsítést tartalmaz: az architektúra számos elemét, amelyeket a munkafüzetben vázolt alapelemek alapján immár könnyű megérteni, nem mutatjuk be. Szintén kihagyjuk az architektúra specifikus elemeit, amelyek rendszerprogramozás során szükségesek. A részletek megtalálhatóak a megfelelő dokumentációs anyagokban, amelyekhez a hivatkozást lásd a munkafüzet végén.

---

<sup>1</sup> számítógép-architektúra: a számítógéppel szemben támasztott hardveres követelmények leírása, kiemelten a processzor belső struktúrájának, ezen keresztül annak programozásának alapvetése. Az azonos architektúrával rendelkező számítógépeket számítógép-családoknak nevezzük.

## Alapvető adatelemek

A számítógépek adatábrázolásának alapeleme a bit. Ez egy olyan tároló, amely egy adott időpillanatban két lehetséges állapot közül az egyiket veszi fel: nullát vagy egyet tartalmaz, köznapi szóhasználatban „*le van kapcsolva*” és „*fel van kapcsolva*”. Minden adatot bitekkel reprezentálunk; a következőkben áttekintjük, hogy milyen adatokat szoktak ábrázolni biteken, illetve hogyan értelmezendők ezek az adatok, ha csak az ábrázolt alakjukkal találkozunk. Mivel a hardver szintjén csak maguk a bitek léteznek, a programozó felelőssége, hogy pontosan tudja a program minden pontján, melyik bit és bitsoport milyen adatot ábrázol, illetve melyik adatszerkezet része.

### Logikai értékek

Egy bitnek két állása van, ezért nagyon könnyen lehet egy biten egy logikai értéket ábrázolni. A konvenciók szerint a 0 a hamis, az 1 az igaz; a szóhasználatban ezeket egymással felcserélhetően használják, pl. „*0 vagy 1 értéke igaz*” ahelyett, hogy „*hamis vagy igaz értéke igaz*”.

A logikai értékekkel különböző műveleteket lehet végezni, amelyeket táblázatos alakban szoktak megadni. A műveleteket általában nem egyes bitekre alkalmazzák, hanem bitvektorokra, pozícióként.

not	0	1
	1	0

and	0	1
0	0	0
1	0	1

or	0	1
0	0	1
1	1	1

xor	0	1
0	0	1
1	1	0

Példa

not 1101 0101  
0010 1010

and 1010 0010  
0111 1101  
0010 0000

or 0011 1000  
1001 0001  
1011 1001

xor 0100 1001  
1111 0110  
1011 1111

### Előjel nélküli egész számok

<i>bitek állása</i>	<b>0</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>0</b>
<i>a bit sorszáma</i>	7	6	5	4	3	2	1	0
<i>a bit helyi értéke</i>	$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$

A számok bitjeit a legkisebb helyi értéket ábrázoló bittől (a legalsó bittől) szokták a magasabb helyi értéket ábrázoló (felső) bitek felé, nullával kezdődően számozni. Ez azért logikus, mert így a pozíciót egy kettes alapú hatvány kitevőjeként értelmezve rögtön adódik a bit helyi értéke, azaz, hogy számként „mennyit ér” a bit.

A fenti nyolcbites szám értéke:  $0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 01100010_2 = 98_{10}$ . Az alsó index a szám számrendszerét jelzi, ezt mindig tízes számrendszerben adjuk meg.

A kettes számrendszerbeli számokat szokták **bináris**nak, a tízes számrendszerbelieket **decimális**nak nevezni. Gyakran használatos még a tizenhatos, más néven **hexadecimális** (rövidítve **hexa**) számrendszer is, mert ezzel rövidebben le lehet írni a bináris számokat, és ha szükség van rá, kényelmesen oda-vissza lehet alakítani őket a két alak között. A hexadecimális számjegyek: 0-tól 9-ig mint a decimálisban, utána pedig a latin ábécé betűi: A, B, C, D, E, F (tízes számrendszerben az értékük sorra 10, 11, 12, 13, 14, 15). A könnyű egymásba alakíthatóság abból következik, hogy a bitek négyes csoportja pont egy hexadecimális számjegyet kódol.

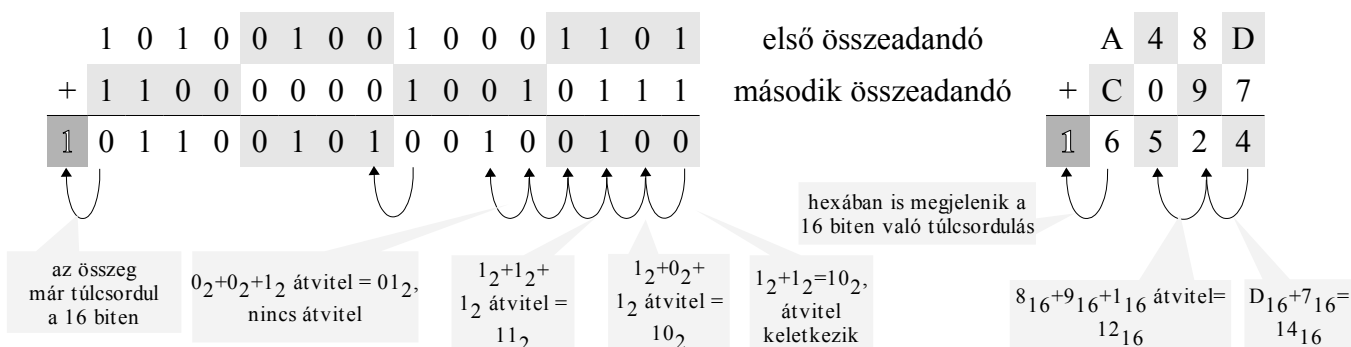
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Tíz-es számrendszerből kettesbe a következő algoritmussal konvertálhatunk. Írjuk le magát a számot. Osszuk el kettővel a számot. Az osztási maradékot írjuk a szám mellé, a hányados egészrészét pedig alá. Folytassuk az eljárást addig, amíg el nem jutunk nulláig.

Ekkor a bináris szám a kapott osztási maradékok sorozata *alulról felfelé olvasva*. Azért nem felülről lefelé, mert minden egyes osztás egy kettes szorzótényezővel bővíti az új bináris számjegyet, vagyis az n-edik sorban levő bináris számjegy a  $2^n$ -es helyi értékhez tartozik. Azaz felülről lefelé a bitek sorszámai rendre 0, 1, 2, ..., tehát fordítva, alulról felfelé olvasva kapjuk meg a számot. Jelen esetben  $42_{10} = 101010_2$ .

42	0	^
21	1	^
10	0	^
5	1	^
2	0	^
1	1	^

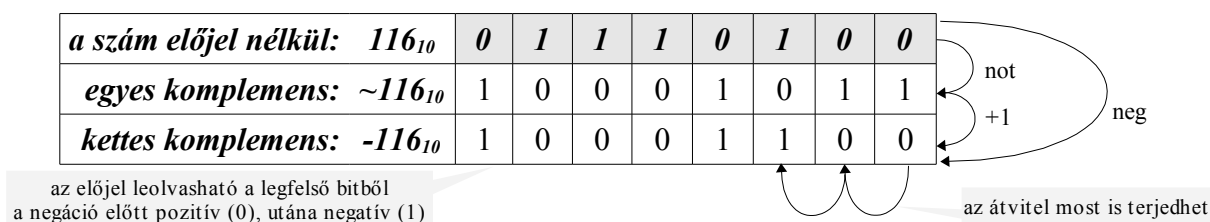
Bináris számokat összeadni a tízes számrendszerben megszokottakhoz hasonlóan lehet. Elindulunk az utolsó pozícióról, és összeadjuk az ott található két számjegyet. A két számjegy összege lesz a két szám összegének utolsó számjegye. Innen a következő pozícióra lépünk, és folytatjuk az összeadást, azonban inentől még egy számjegyet is hozzá kell adnunk az adott pozíción szereplőkhöz. Ez az átvitel, ami 1, ha az előző összeadás túlsordult, vagyis az eredménye már nem fért el egy számjegyben, különben pedig 0.



### Előjeles egész számok

A nemnegatív számok alakja ebben az ábrázolásban megegyezik az előjel nélküli ábrázolás szerintivel. Az összes lehetséges ilyen szám közül most azonban csak azokat a számokat tekintjük érvényes pozitív számnak, amelyeknek a legfelső bitje nulla.

A negatív számokat a következőképpen kaphatjuk meg a pozitív számokból. Írjuk fel az előjel nélkül tekintett számot binárisan, alkalmazzuk rá bitenként a not műveletet, majd növeljük meg eggyel a kapott számot. Ennek a műveletnek **negáció** vagy **negálás** a neve. Az előjelet a legfelső bit mutatja. Ezzel az eljárással van egy olyan szám, amit nem kapunk meg egyetlen nemnegatív számból kiindulva sem: az egyes bittel kezdődő, onnan végig csupa nullát tartalmazó szám. Ennek az értéke eggyel kisebb, mint a fent leírt módon ábrázolható számok közül a legkisebb; n bites szám esetén az értéke pontosan  $2^{n-1}$ .



Tekinhetjük úgy is, hogy az n biten ábrázolt számok a modulo  $2^n$  maradékosztályokat adják ki: az előjel nélküliek a  $0..2^n-1$  reprezentánsokkal, az előjelesek a  $-2^{n-1}..2^{n-1}-1$  reprezentánsokkal.

Előjeles számok összeadását a nemnegatív számokhoz hasonlóan végezhetjük el. Kivonáshoz adjuk hozzá a számhoz az összeadandó negáltját.

### Törtszámok

A törtszámok ábrázolását, amely általában ún. lebegőpontos formátumban történik, és az ezek kezelésére való utasításokat nem mutatjuk be részletesen a munkafüzetben. A lebegőpontos számok ábrázolását a numerikus analízis tárgyalja.

## Karakterek, stringek

Minden egyéb adathoz hasonlóan a karaktereket is bitekkel kell kódolnunk. Először is rögzítenünk kell, milyen karaktereket szeretnénk ábrázolni: ez a **karakterkészlet**. Minden karakternek adunk egy sorszámot, ez a **karakter kódja**. Ahhoz, hogy különböző számítógépeken is ugyanazt a tartalmat kódolja ugyanaz a szöveg, meg kell állapodnunk a karakterek **kódolásában**: melyik karaktert milyen bitsorozattal ábrázoljuk. Egyszerűbb a helyzetünk, ha minden karakter kódja azonos számú bitet tartalmaz (pl. nyolc bites ASCII, UCS-2), de helytakarékosági okokból lehet változó bithosszon is kódolni (pl. UTF-8). Minden esetben követelmény azonban, hogy az ábrázolt formából egyértelműen meg lehessen állapítani, mi a karakter kódja, abból pedig azt, hogy melyik karakterről van szó.

A számítástechnika korai éveiben nem volt szabványosítva, hogy a karaktereket hogyan kell kódolni, ezért a különböző architektúrájú gépek között nehéz volt adatcserét bonyolítani. Ezen sokat segített az 1967-ben megjelent ASCII kódolás, ami 7 bitben rögzítette a karakterkódok hosszát. Az ASCII karakterkészlete tartalmazza a számjegyeket, a latin kis- és nagybetűket, vezérlőkaraktereket, pl. sorvége, csengő, valamint különböző írásjeleket, pl. szökőz, pont, kérdőjel, aláhúzás. Később kiderült, hogy a felhasználóknak szüksége van a fentiekén kívül a saját nyelvükben előforduló, a latintól eltérő karakterekre is, ezért többféleképpen kiegészítették 8 bitesre: az újonnan nyert pozíciókon az egyes régiók karakterei kerültek, pl. az ISO-8859-1 a nyugat-európai, az ISO-8859-2 a közép- és kelet-európai nyelvek (köztük a magyar) speciális karaktereit tartalmazza.

Ez jópár évig elégnék tűnt, azonban több probléma is adódott. Egyrészt ezek a karakterkódolások csak korlátozottan tették lehetővé, hogy többféle nyelvet lehessen egy dokumentumon belül használni. Amennyiben nem volt feltüntetve, melyik kódolást alkalmazza a dokumentum, téves választás esetén egyes karakterek helytelenül jelentek meg. További hiányossága volt ennek a rendszernek, hogy a lefedett nyelvek még mindig meglehetősen kevesen voltak: a távol-keleti nyelveket nem támogatta egyik kiterjesztés sem, ezekhez ismét más kódrendszerek születtek.

A jelenlegi legátfogóbb karakterkódolási szabvány az 1991 óta létező Unicode, illetve ISO/IEC 10646. A kettő számunkra nem számottevő különbségektől eltérően megegyezik: a karakterkészlet és a kódolás ugyanaz a két rendszerben. Ez a karakterkészlet arra törekszik, hogy a világ összes létező és volt karakterét ábrázolni lehessen vele, azonban fenntartja a kompatibilitást az ASCII ISO-8859-1-es kiterjesztésével is, melyet a Unicode teljesen tartalmaz. Az Unicode-hoz többfajta kódolás létezik: a legismertebb a változó kódhosszú UTF-8. Ennek megvan az az előnye, hogy ha olyan szöveget kódolunk el vele, amely csak legfeljebb 127-es kódú ASCII karaktereket tartalmaz, akkor a kapott szöveg megegyezik az ASCII kódolással kapott szöveggel. Jelenleg a szabvány 5.0-ás verziója a legfrissebb, és folyamatosan fejlesztik.

karakterkód		vezérlőkarakter	karakterkód		karakter
0	00	szöveg vége	32	20	szökőz
10	0A	új sor	48..57	30..39	0..9
13	0D	kocsi vissza	65..90	41..5A	A..Z <sup>2</sup>
			97..122	61..7A	a..z

ASCII szövegekben a sorvégét Unix rendszereken **0xA**, Internetes protollokban és Windows alatt **0xD 0xA** jelzi; a Unicode szabványban mindkettő, sőt, más kombinációk is jelezhetik a sor végét.

Szövegek kódolásánál a karakterek kódjai sorban következnek egymás után. Kétféle konvenció van arra nézve, meddig tart a szöveg: a „C konvenció” szerint az első 0 kódú karakterig, a „Pascal konvenció” szerint pedig a szöveg kezdete előtt el van tárolva a hossza is.

2 Egyéb kódolásokban (pl. az IBM nagygépeken használatos EBCDIC) nem garantált, hogy a karakterek kódjai hézagtalanul követik egymást.

## Feladatok

1. Mekkora értékek tárolhatóak egy bitvektorban? Add meg általánosan is!

bitek száma	számrendszer	előjel nélkül		előjelesen	
		legkisebb érték	legnagyobb érték	legkisebb érték	legnagyobb érték
8	hexadecimális				
	decimális				
16	hexadecimális				
	decimális				
32	hexadecimális				
	decimális				
4 · n	hexadecimális				
	decimális				

2. Töltsd ki az üres cellákat a táblázatban!

bitek száma	bináris	decimális		hexadecimális
		előjel nélküli	előjeles	
8	1100 1111			
8				6B
8			-24	
16	1001 0110 0001 1110			
16				20 A5
16			-11 721	

3. Végezd el az alábbi műveleteket!

1001 1101  
and 1100 0111

0110 1110  
or 1000 0000

0101 1010  
xor 1111 0001

$$0101\ 0110\ 1101\ 1101_2 + 1C\ E3_{16} = \underline{\hspace{2cm}}_2 = \underline{\hspace{2cm}}_{10} = \underline{\hspace{2cm}}_{16}$$

$$16\ 537_{10} - 0000\ 0011\ 1010\ 1100_2 = \underline{\hspace{2cm}}_2 = \underline{\hspace{2cm}}_{10} = \underline{\hspace{2cm}}_{16}$$

$$54\ 32_{16}\ or\ 38\ 529_{10} = \underline{\hspace{2cm}}_2 = \underline{\hspace{2cm}}_{10} = \underline{\hspace{2cm}}_{16}$$

4. Írd le a saját nevedet ASCII kódolással!

5. Írd le a „Hello Vilag” szöveget ASCII kódolással!

## Az architektúra alapjai

### A memória szervezése

A memória bitekből épül fel, de a biteket nem tudjuk külön-külön elérni: a legkisebb egység a nyolc bit együtteséből felépülő *oktet*, amit általában **byte**-nak neveznek. A byte-ok sorban helyezkednek el a memóriában, mindegyik rendelkezik egy 32 bites sorszámmal, amit a byte **memóriacímének** nevezünk. Ez a **lineáris memóriamodell**; a hardver és az operációs rendszer elfedi előlünk a bonyolultabb **szegmentált modellt**.

A processzor és a memória közötti adatforgalom három fő vezetékkeg segítségével zajlik, amelyeket **síneknek** vagy **buszoknak** nevezünk. A memóriához való hozzáférés menete vázlatosan: a processzor először elküldi a **címsínen**, hányas sorszámu cím tartalmához fogunk hozzáférni, és a **vezérlősínen** (I/O buszon) a járulékos információkat, például az adatforgalom irányát<sup>3</sup> és az átvitelre váró adat méretét, ami lehet **byte**, **szó (word)**, 2 byte) vagy **duplaszó (dword)**, double word, 4 byte). Ezután maga az adatátvitel következik az **adatsínen**.

Mivel a vezetékek egyszerre csak egy bitnyi információt tudnak szállítani, és azt is csak egy irányban, a processzor kommunikációs képességeit behatárolja, hogy az egyes sínek hány vezetékot tartalmaznak. Az adatsín szélességét az architektúra **gépi szóhosszának** hívják. A processzoron belül, a regiszterekben tárolható adatok méretét **belső szóhossznak** nevezük. Az x86 architektúrán belül a 32 bites szervezésű gépek gépi és belső szóhossza, valamint a címsín szélessége egyaránt 32 bit. Az architektúra korai tagjaiban a gépi szó még 16 bites volt, a „szó” ebből a történelmi okból maradt meg 386-ostól felfelé ekkorának. Tehát a 32 bites processzorokon a *gépi szó a duplaszó*, és ha tehetjük, akkor ekkora méretű adatokkal dolgozunk.<sup>4</sup>

Az architektúra egy, a programozó számára kevésbé intuitív tulajdonsága, hogy a byte-nál nagyobb méretű adatok byte-jait ún. **little endian** sorrendben tárolja. Ez azt jelenti, hogy a legkisebb helyi értékű byte kerül a legkisebb címre, az alulról következő byte a soron következő címre stb. Ez azt jelenti, hogy a szám leírt alakjához képest, ha a memóriát balról jobbra növekedő címeken képzeljük el, éppen fordított sorrendbe kerülnek az adatok. Más architektúrák éppen ellenkezőleg, **big endian** sorrendben írnak a memóriába (ekkor a helyi értékek csökkennek a címek növekedtével). Egyes architektúrákon átprogramozható futás közben, melyik módon működjenek.

Mivel a memóriacímek is csak számok, ezért a memóriában magában is el tudunk tárolni egy másik memóriacímet. Amikor egy 32 bites adatot nem pusztán számként, hanem memóriacímként értelmezzük, akkor az adatot **mutatónak (pointernek)** nevezük, és ezek segítségével építhetők fel láncolt adatszerkezetek.

Példa		C15AEF4C	C15AEF4D	C15AEF4E	C15AEF4F		a byte memóriacíme
	...	0101 0000	1110 1111	0101 1010	1100 0001	...	byte értéke (bináris)
		50	EF	5A	C1		byte értéke (hexa)

Little endian: a fenti C15AEF4C címen található duplaszó értéke nem 50EF5AC1, hanem C15AEF50.  
Mutató: ha mutatóként értelmezzük a duplaszót, akkor az utána következő byte-ra mutat.

Az adatok az **adatszégmensben** helyezkednek el, aminek a kezdetét a **section .data** direktíva vezet be. Adatot úgy lehet elhelyezni, hogy először leírjuk, mekkora méretű adatokkal foglalkozunk, majd leírjuk magukat az adatokat vesszőkkel elválasztva. A lehetséges adاتمéret: **db**, **dw** és **dd**, sorra byte, szó és duplaszó méretű adatokat jelentenek.

Az elhelyezett adatokat érdemes **címkével** is ellátni, mert ez megkönnyíti később a hozzáférést. A címkét a sorban csak szóközök és tabulátorok előzhetik meg; a címke neve tartalmazhat karaktereket, számjegyeket (számjeggyel nem kezdődhet) és pontot. A címke végére lehet egy kettőspontot tenni, de ez nem része a címke nevének. Amennyiben sok hasonló adatot szeretnénk elhelyezni, az adاتمéret jelzése elé írhatunk egy **times mennyiség** előtagot (a *mennyiség* egy konstans, pl. 15), ami a megadott mennyiséget helyez le az adatokból.

<sup>3</sup> processzor → memória vagy memória → processzor

<sup>4</sup> néhány éve megjelentek a legújabb, 64 bites processzorok is, amelyeken a gépi szó mérete 64 bit



```

        section .data
cimke   dd      1, 2, 0AF4B551Ch
        times 4 db 1           ; adat elhelyezése külön címke nélkül
        db      1, 1, 1, 1     ; hatása megegyezik az előző soréval: négy byte-nyi egyes
szoveg  db "ASCII szoveg", 0xA, 0

```

A fentiek hatására az alábbi byte-ok keletkeznek. Aláhúzás jelöli az egy egységként definiált byte-okat.

```

01 00 00 00 02 00 00 00 1C 55 4B AF 01 01 01 01
01 01 01 01 41 53 43 49 49 20 73 7A 6F 76 65 67 0A 00

```

Sokszor előfordul, hogy tudjuk, szükségünk lesz valamekkora adatterületre, de annak tartalmát kezdetben még nem töltjük fel. Ezek számára az inicializálatlan adatszégmensben tarthatunk fenn helyet. Ennek a szégmensnek a kezdetét a **section .bss** direktíva jelzi. Itt nem helyezünk el adatokat, csak tárterületet tartunk fenn számukra: **resb mennyiség**, **resw mennyiség**, **resd mennyiség** leírásával jelezhetjük, hogy az aktuális pozíciótól kezdődően adott mennyiségű byte-ra (resw esetén kétszer, resd esetén négyszer annyira) lesz szükségünk. Itt szintén érdemes címkéket alkalmazni.

A programkód írása során az inicializált és inicializálatlan adatszégmensbe tartozó címkék használata nem különbözik. Lényeges különbség viszont, hogy a program futásának a kezdetén ezeknek a területeknek a tartalma nem ismert, és amíg nem töltöttük fel, ismeretlen értéket (társzemetet) tartalmaznak.

```

        section .bss
tomb      resb 1024           ; 1024 inicializálatlan byte lefoglalása, a kezdetének címkéje tomb
fajlleiro resd 4             ; a fajlleiro címkén 4 byte érhető el

```

A nasm címkéi kétfajta lehetnek. A **globális címkék** karakterrel kezdődnek. A **lokális címkék** ponttal kezdődnek, és a teljes alakjukat úgy kaphatjuk meg, ha hozzáfűzzük a lokális nevet az előtte utoljára definiált globális címke nevéhez. Ez azért hasznos, mert ugyanazt a nevet újra fel lehet használni: globális nevet általában az alprogramok belépési pontjai és az adatszerkezetek szoktak kapni, lokális nevet az alprogramokban előforduló vezérlési szerkezeteknek és az adatszerkezetek komponenseinek adunk. Amíg egy globális címke hatókörében vagyunk, az alatta definiált lokális címkére a lokális névvel lehet hivatkozni, azonban egy másik globális címke hatókörében ki kell írni a lokális címke teljes nevét – ámbár ha más globális címke alól szeretnénk a címkére hivatkozni, akkor érdemes megfontolni, hogy globális nevet adjunk neki.

```

globalisCimke
.lokalisCimke           ; teljes alakja: globalisCimke.lokalisCimke

```

*Fontos: a memória nem „tud” arról, hogy mi hogyan szeretnénk értelmezni a tartalmát, nekünk kell arra ügyelnünk, hogy mindig megfelelően kezeljük azt. Ha pl. egy adatelem méretét dw adja meg, majd megpróbálunk belőle byte hosszan olvasni, akkor nem kapunk hibát, hanem sikeresen kiolvassuk az ott elhelyezkedő szó alsó felét (mivel a little endian tárolás miatt az kerül az első byte-ra).*

## A regiszterek

A legtöbbet használt adattárolók a **regiszterek**. Ezek a memória kiemelt szereppel rendelkező részei, melyek a memória többi részétől elkülönülnek mind fizikailag (a processzor belsejében helyezkednek el, és ezért nagyon gyorsan elérhetőek), mind kezelésüket tekintve.

																ax, bx, cx, dx, sp, bp, si, di															
																ah, bh, ch, dh				al, bl, cl, dl											
31. bit .....																15. bit .....				7. bit .....				0. bit							
0	1	1	0	0	0	0	1	1	1	0	1	1	0	1	1	1	1	1	1	0	1	1	0	1	1	0	0	0	0	0	0
eax, ebx, ecx, edx, esi, edi, esp, ebp																															

a regiszterek részeinek nevei

A programozás során nyolc **általános célú regisztert** használhatunk. Ezek 32 bitesek, neveik **eax**, **ebx**, **ecx**, **edx**, **esi**, **edi**, **esp** és **ebp**. Mindegyiknek elérhető külön regiszterként az alsó fele (a 15..0-adik bitekből álló részregiszter): **ax**, **bx**, **cx**, **dx**, **si**, **di**, **sp** és **bp**. Ezek közül az első négy regiszter felső (15..8-adik bit) és alsó fele (7..0-adik bit) is egy-egy névvel ellátott, 8 bites regiszter: **ah**, **al**, **bh**, **bl**, **ch**, **cl**, **dh**, **dl**<sup>5</sup>. A regiszterek közül leggyakrabban a 32 biteseket fogjuk használni, és esetenként, például karakteres adatok kezelésére, a 8 biteseket.

Van két olyan regiszter, amelyet nem lehet közvetlenül elérni, azonban a számítógép működéséhez elengedhetetlen a létük. Az **utasításmutató**, **eip** (*instruction pointer*), azt mutatja, hogy melyik memóriacímről folytatódik a végrehajtás, vagyis a memória melyik címéről kell a processzornak felolvasnia a következő utasítás kódját. A tartalma automatikusan változik, általában a közvetlenül az előző végrehajtott utasítás utáni címre mutat, kivéve az ugró utasítások esetén, amikor az ugrás céljának címét kapja értékül.

A **jelzőbitek regisztere** (eflags) az utoljára elvégzett számításról, illetve a processzor aktuális állapotáról tárol információkat.<sup>6</sup> Bitjei közül az alábbiak változhatnak meg aritmetikai műveletek eredménye szerint.

Jel	Név		Mikor egy az értéke?	Jel	Név		Mikor egy az értéke?
Z	zero	nulla	Az eredmény nulla.	S	sign	előjel	A szám negatív előjelesen.
C	carry	átvitel	Előjel nélküli (túl-/alul-)csordulás.	O	overflow	túlcsordulás	Előjeles (túl-/alul-)csordulás.
P	parity	paritás	Az eredmény páros.				

## Az utasítások szerkezete

Az **utasítások** a gép működésének elemi egységei: vezérlik a processzort, milyen műveletet végezzen az adatokkal, illetve melyik utasítással folytassa a végrehajtást. Felépítésüket tekintve egy **mnemoniknak** nevezett kulcsszóból és megfelelő számú, vesszővel elválasztott **operandusból** (paraméterből) állnak. Az operandusok számát a mnemonik után alsó indexben jelöljük az utasítások bemutatása során. Egy sorba legfeljebb egyetlen utasítás írható. Az utasítások a **kódszegmensben** találhatóak, ennek a kezdetét a **section .text** direktíva jelzi.

A kétoperandusú utasítások, ha az utasítás leírása nem mond mást, az általuk elvégzett művelet eredményét az első operandusban tárolják el, ezért arra **célként**, a második operandusra pedig **forrásként** fogunk hivatkozni. Az operandusok lehetséges kombinációi a munkafüzet végén levő táblázatban vannak összefoglalva. Két kivételtől (**movsx**, **movzx**) eltekintve teljesül továbbá minden ismertett kétoperandusú utasításra, hogy *a két operandus hosszának meg kell egyeznie*: mindkettőnek vagy byte-osnak, vagy szavasnak, vagy duplaszavasnak kell lennie.

<sup>5</sup> Az 'e' betű a regiszterek elején azt jelöli, hogy ezek a 16 bites regiszterek kiterjesztett (extended) változatai, mert a számítógépcsalád első gépein még csak a 16 bites regiszterek voltak megtalálhatóak. A 'h' és 'l' a „felső” és „alsó” (high és low) szavak rövidítései.

<sup>6</sup> Egyes architektúrákon (pl. IBM 360) az utasításmutató regiszter is tartalmaz jelzőbiteket.

## Konstansok

Szám konstans leírásakor el kell döntenünk, milyen számrendszerben ábrázoljuk.

- Ha **decimális** konstansról van szó, nem kell külön jelölést alkalmaznunk, csak leírjuk a számot: **157**.
  - **Bináris** konstans esetén egy 'b' betűt írunk a szám végére: **001011101b**.
  - **Hexadecimális** konstansokat kétféleképpen tudunk írni.
    - A **'0x'** prefix után írjuk le a konstans: **0xABCDE**.
    - A konstans után **'h'** posztfixet illesztünk: **8CDh**.
      - Ha a szám betűvel kezdődik, nulla prefixet kell írunk a szám elé: **0F352h**.
- Enélkül ugyanis előfordulhat, hogy valamelyik regiszter, például ah nevével ütközik a leírt alak.

**Karakterstring konstansok**at aposztrófok (') vagy idézőjelek (") közé írhatunk. Ezek értéke megegyezik azzal a bináris száméval, amit úgy kapunk, hogy a karakterek nyolc bites ASCII kódjait összefűzzük. Itt, a numerikus értékekkel ellentétben, nem számít, hogy little vagy big endian üzemmódú-e a számítógép, mindig abban a sorrendben tárolódnak a byte-ok, ahogy a stringben szerepelnek.

Lehetőség van arra, hogy a konstansokból kifejezéseket alkossunk, amelyeket a nasm assembler *fordítási időben*, 32 biten ábrázolva kiértékel. A következő operátorok állnak a rendelkezésünkre, csökkenő precedencia-sorrendben.  
*Fontos: ez csak konstansokra vonatkozik; regisztereket és memóriatartalmakat tartalmazó kifejezéseket csak utasításokkal tudunk kiszámítani.*

zárójelek	()
bitenkénti negáció	~
negatív előjel	-
szorzás	*
összeadás, kivonás	+ -
bitenkénti eltolás	<< >>
bitenkénti és, vagy, kizáró vagy	&   ^

A konstans hosszának értelmezésében szerepe lehet annak is, hogy mi a másik paraméter: **mov eax, 4** ugyanazt jelenti, mint **mov eax, 0x00000004**, azaz eax teljes tartalmát átírja, a második bit 1 lesz, a többi 0.

Példa	<pre>mov ecx, 183 mov dl, 101101b mov al, 'a'</pre>	<pre>mov esi, 0x554C mov ah, 0Ah mov ax, 5 + 8 / 4</pre>	<pre>mov eax, ebx + 5 mov cl, Eh</pre>	<pre>mov dl, [adat] / 2</pre>	Hibás
-------	---	--	--	-------------------------------	-------

## A memória címzése

A memória tartalmának hozzáférésére sokféle módszer közül válogathatunk. Általánosan igaz mindegyiknél, hogy azt a címet, ahol az adatelem elkezdődik a memóriában, szögletes zárójelek közé írjuk; ezt az alakot **memóriatartalomnak** nevezzük. A memória-hozzáférésnél fontos, hogy meg kell adnunk azt is, milyen hosszúságú adattal dolgozunk, mert önmagában a cím csak azt mutatja meg, hol kezdődik a memóriában az adat. Ezt a **byte**, **word** vagy **dword** kulcsszavakkal lehet megadni. A legegyszerűbb címzési mód, ha ismerjük a konkrét memóriacímet, ahová írni szeretnénk, ekkor egyszerűen leírjuk szögletes zárójelek között, elé pedig az adat hosszát: **word [11101b]** vagy **dword [0x1234ABCD]**. Ha egy utasítás egyik operandusáról egyértelműen kiderül annak hossza a másik operandus alapján, a hosszinformáció elhagyható, *különben nem*.

Nem egyértelmű a következő utasítás. A második operandusról nem derül ki, hogy byte, szó vagy duplaszó hosszan értelmezendő-e a 18 konstans. Meg kell adni az operandus hosszát. Az operandus hosszát megadhatjuk így is.	<pre>mov [1101011b], 18 mov [1101011b], dword 18 mov dword [1101011b], 18</pre>	Hibás
--	---	-------

Egyoperandusú utasításnál nincs lehetőség ilyen rövidítésre. Ekkor ki kell írni az operandus hosszát.	<pre>inc [794h] inc dword [794h]</pre>	<pre>mul [0B11h] mul byte [0B11h]</pre>	Hibás
--	--	---	-------

Általában nem ismerjük számszerűleg a memóriacímet, mert kézzel kiszámolni nehézkes lenne. Ennek megkönnyítésére lehetőség van a memória egyes pontjaihoz címkéket hozzárendelni. Címkét definiálni úgy lehet, hogy a forráskódban a sor elejére leírjuk a címke nevét. Ha a forráskód egy pontjain leírjuk a címke nevét, az ekvivalens azzal, mintha azt a memóriacímet írtuk volna le konstansként, ahol a címkét definiáltuk. Természetesen két különböző címke nem kaphatja ugyanazt a nevet.

A szögletes zárójelen belül az alább felsoroltak közül egy vagy több összetevő összege állhat.

- egy 32 bites regiszter
- egy skálázott (konstans szorzóval ellátott) 32 bites regiszter, a skálatényező 1, 2, 4 vagy 8 lehet
- egy 32 bites konstans<sup>7</sup>

Példa	mov	eax,	[cimke]	mov	al,	[ebx+esi]	mov	[1101011b],	18	inc	[794h]	Hibás	
	mov	eax,	[ebx]	mov	eax,	[edx+4*edi]	mov	[eax-ecx],	edx	mul	[0B11h]		
	mov	esi,	[esi]	mov	edx,	[adat+8]	mov	eax,	[[esi+8]+4]	mov	edi,		[al]
	mov	eax,	[ebp-4]										

## Adatmozgatás

Alapvető igény, hogy adatokat mozgassunk. Ezt a **mov**<sub>2</sub> utasítással lehet elérni. Hatására a céloperandus felveszi a forrás értékét. Lehetőség van arra is, hogy kisebb méretű adat kiterjesztésével töltsünk fel regisztert vagy memóriatartalmat. Ezt tehetjük előjel nélkül: **movzx**<sub>2</sub> (ekkor nullákkal töltődik fel a bővebb felső rész), vagy előjelesen: **movsx**<sub>2</sub> (a kisebb méretű adat legfelső bitjének értékével töltődik fel a maradék). Végrehajtásuk után a cél értéke megegyezik a forrás értékével előjel nélkül, illetve előjelesen. Amennyiben meg akarjuk cserélni a forrást és a célt, az **xchg**<sub>2</sub> utasítást használhatjuk. Ennek az utasításnak egyik operandusa sem lehet konstans.

## Logikai utasítások

A logikai utasítások: **not**<sub>1</sub>, **and**<sub>2</sub>, **or**<sub>2</sub>, **xor**<sub>2</sub> (negáció, és, vagy, kizáró vagy) bitenként valósítják meg a fent leírt logikai műveleteket az operandusokon. Az **and** segítségével törölni (nullára állítani) lehet kiválasztott biteket: olyan konstans kell második operandusnak adni, amely pontosan a törölni kívánt pozíciókon tartalmaz nullát, a többin egyes. Az **or**-ral éppen fordítva, beállítani lehet biteket, ha a konstans beállítandó pozícióin egyes, a többin nullát tartalmaz. Az **xor** pedig megfordítja azokat a biteket, amelyek egyesre vannak állítva.

**Regisztert úgy is törölhetünk, hogy xor-oljuk a regisztert önmagával.** Ez pontosan azokat a biteket fordítja meg a regiszterben, amelyek be vannak állítva benne, vagyis az eredmény valóban nulla.

A **setfeltétel**<sub>1</sub> utasítások egyetlen, *byte méretű* operandusukat 1-re állítják be, ha teljesül a megadott feltétel, 0-ra, ha nem. A feltételek lehetséges alakjait a feltételes ugrásoknál írjuk le részletesen; ezeknek az utasításoknak az alakja annyiban tér el a feltételes ugró utasításokétól, hogy 'j' helyett 'set' a mnemonik eleje.

## Bitléptető és bitforgató utasítások

Ezek az utasítások (**sar**<sub>2</sub> és **shl**<sub>2</sub>, **sal**<sub>2</sub>, **shr**<sub>2</sub>, **rol**<sub>2</sub>, **ror**<sub>2</sub>, **rcr**<sub>2</sub>, **rcl**<sub>2</sub>) kétoperandusúak, a második paraméterük konstans vagy a **cl** regiszter lehet. A működésük szerint a biteket léptetik annyi pozícióval, amennyit a második operandus megszab. A léptetés iránya a mnemonik utolsó betűjétől függ: ha 'l', felfelé (*left*, azaz balra), ha 'r', lefelé (*right*, jobbra). Ha az első betű 'r' (*rotate*), akkor forgatásról van szó, vagyis a regiszter végén túlcserélődő, kilépő bitek a másik oldalról jönnek be. Az **sar** utasítás esetén a legfelső bit értéke forgatódik be felülről, az összes többi léptető utasításnál 0 bitek lépnek be.

**Kettőhatvánnyal való szorzást** elvégezhethetünk **léptető műveletek segítségével**: előjel nélküli szorzásra és osztásra az **shl** és **shr**, előjeles szorzásra és osztásra a **sal** és **sar** utasítások használhatóak. Az operandus azt adja meg, kettő hányadik hatványával szorzunk/osztunk. Könnyebb megértéshez lásd a munkafüzet végén szereplő ábrákat.

<sup>7</sup> Konstans látszólag ki lehet vonni, ez azonban valójában egy modulo  $2^{32}$  összeadásként jelenik meg.

## Aritmetikai utasítások

Az **add<sub>2</sub>** és a **sub<sub>2</sub>** segítségével lehet összeadni és kivonni. Mindkettő működik előjeles és előjel nélküli számokra. Az eggyel való növelés és csökkentés annyira gyakori, hogy erre külön van egy-egy utasítás: **inc<sub>1</sub>** és **dec<sub>1</sub>**.

Szorozni a **mul<sub>1</sub>** *egyoperandusú* utasítással lehet előjel nélkül, az **imul<sub>1</sub>** segítségével pedig előjelesen. Ezek egy regisztert vagy memóriatartalmat kapnak; az operandusuk nem lehet konstans. Az operandus értékével megszorozzák *eax*-nak az operandus hosszával megegyező méretű részét (*al*, *ax* vagy *eax*) és a szorzatot elhelyezik *ax*-ben, *dx:ax*-ben vagy *edx:eax*-ben. Ez utóbbi kettő azt jelenti, hogy az eredmény felső fele kerül (*e*)*dx*-be, az alsó fele pedig (*e*)*ax*-be; a két regisztert átmenetileg egy nagyobb regiszter két felének képzelhetjük el. Erre azért van szükség, mert a szorzat közel kétszer akkora helyet is igényelhet, mint a tényezők.

Példa	<code>mov eax, 4141659</code>	<code>mov eax, 0xA7F04BFF</code> ; ekvivalens eredményt ad ezzel	
	<code>mov ebx, 2356781</code>		<code>mov ebx, 2356781</code> ; mert $414659 \cdot 2356781 =$
	<code>mul ebx</code>		<code>mov edx, 8E0h</code> ; $8E0_{16} \cdot 2^{32} + A7F04BFF_{16}$

Osztani a **div<sub>1</sub>** és az **idiv<sub>1</sub>** *szintén egyoperandusú* utasításokkal lehet. Ezeknél a forrás és a cél éppen fordítva van a szorzásokhoz képest (itt is a cél méretétől függően). Továbbá az osztás maradékát is megkapjuk, az operandus méretétől függően *edx*-ben, *dx*-ben illetve *ah*-ban.

Értéket negálni a **neg<sub>1</sub>** utasítással lehet. A negáció műveletének menetét lásd az előjeles számokat leíró részben.

## Vezérlésátadás

Előfordul, hogy a program végrehajtását egy másik címen szeretnénk folytatni. Egy ilyen eset, amikor végrehajtottuk egy elágazás igaz ágát, és a hamis ágat, aminek a kódja közvetlenül az igaz ág kódja után következik, már nem akarjuk végrehajtni. Ilyenkor az igaz ág kódjának a végére elhelyezünk egy feltétel nélküli ugrást, aminek hatására a vezérlés a hamis ág kódja után folytatódik, vagyis a teljes elágazás kódja után. A feltétel nélküli ugrás **jmp<sub>1</sub>**, operandusa pedig az a címke, ahová az ugrás után a vezérlést juttatni szeretnénk.

Vannak olyan esetek, amikor valamilyen feltételtől függően szeretnénk csak átadni a vezérlést máshová, ha pedig nem teljesül a feltétel, a következő utasítást akarjuk futtatni. Az előbbi példánál maradva, az elágazás feltételének vizsgálatakor pontosan ez a helyzet: a teljesül a feltétel, az igaz ág kódját hajtjuk végre, ami közvetlenül az ugrás után áll, ha nem teljesül a feltétel, akkor pedig el kell ugranunk a hamis ágra. A feltétel megvizsgálását a **cmp<sub>2</sub>**, a feltételes ugrásokat pedig a **jfeltétel<sub>1</sub>** utasításokkal tudjuk megvalósítani. Ezeknek az alakja: a 'j' (ugrás, *jump*) után opcionálisan egy 'n' (nem, *not*), aztán az 'a', 'b', 'c', 'g', 'l' közül valamelyik (felett, alatt, átvitel, nagyobb, kisebb: *above*, *below*, *carry*, *greater*, *less*), majd opcionálisan egy 'e' (egyenlő, *equal*); lehetséges alakok még önmagukban a *je* és *jne*. A *cmp* szerepe, hogy a jelzőbitek regiszterét beállítja a feltételes ugrás végrehajtásához szükséges értékekre, de a paraméterek tartalmát nem változtatja meg. A feltételes ugrások megvizsgálják a jelzőbitek állását, majd ennek eredménye szerint ugranak az operandusban kapott címre, vagy folytatják a végrehajtást. A feltételek a következők lehetnek.

- *e* ugrik, ha az öt megelőző *cmp* két paraméterének az értéke azonos
- *a* és *b* ugrik, ha a *cmp* első paraméterének az értéke előjel nélkül nagyobb (*b*: kisebb)
- *g* és *l* ugrik, ha a *cmp* első paraméterének az értéke előjelesen nagyobb (*l*: kisebb)

Az 'e' betű megengedi az egyenlőséget is, az 'n' betű a feltétel tagadása.

A feltétel nélküli ugrások tetszőlegesen távoli kódra át tudják adni a vezérlést. A feltételes ugrások alapesetben csak „közelre” tudnak ugrani, ezért célszerű a **near** kulcsszót használni a feltételes ugrás mnemonikja után.

Példa	<code>cmp eax, ebx</code>	; akkor ugrik, ha <i>eax</i> előjel nélkül összehasonlítva
	<code>jnbe near címke</code>	

Az alprogramhívás eszköze a **call<sub>1</sub>** és a **ret<sub>0</sub>**, a kivételkezelésé az **int<sub>1</sub>**, melyeket a megfelelő helyeken ismertetünk.

## Programozási konstrukciók megvalósítása

### Változók deklarációja, definíciója

Amikor egy változóval dolgozunk egy programozási nyelven, mindig tudnunk kell, hogy mekkora területet foglal, illetve milyen műveleteket hajthatunk végre rajta. Magasszintű programnyelveken ezek az információk a deklarációkból derülnek ki, így a fordítóprogram akkor is képes a megfelelő kódot generálni az adat elérésére, ha az egy másik fordítási egységben helyezkedik el.

Az assembly szintjéről nézve az adatok pusztán byte-sorozatok. Itt a hosszra és az elérésre vonatkozó információkat közvetlenül a generált kódban kell helyesen felhasználni. Ehhez fordítóprogram írása során fel kell használni a rendelkezésre álló adatokat, forráskód közvetlen írása során pedig ügyelni kell, hogy ne helyesen írjuk le az adathozzáférés méretét, például duplaszavasán tárolt adatot ne próbáljunk nyolcbites regiszterbe tölteni.

### Értékadás

A `mov` (ha szükséges: `movsx`, `movzx`) utasítás segítségével lehet értéket adni egy memóriatartalomnak.

Ha az adat hossza nagyobb egy gépi szónál, akkor több mozgatásra van szükség. Ehhez az adatot fel kell bontani legfeljebb duplaszó hosszúságú részekre, és külön kell értéket adni a részeknek. A `mov` nem tud két memóriatartalmat operandusként fogadni, hiszen az adatsínen nem tudunk egyszerre befelé és kifelé is adatot mozgatni, a két lépés közti tárolásra az egyik általános célú regisztert használhatjuk.

A `v1` és `v2` változók 8 byte hosszon tartalmazzanak adatokat. Valósítsuk meg a `v1 := v2` értékadást!

Példa	<code>mov</code>	<code>eax,</code>	<code>[v1]</code>	<code>;</code>	az első 4 byte felolvasása
	<code>mov</code>	<code>[v2],</code>	<code>eax</code>	<code>;</code>	az első 4 byte kiírása
	<code>mov</code>	<code>eax,</code>	<code>[v1 + 4]</code>	<code>;</code>	a második 4 byte felolvasása
	<code>mov</code>	<code>[v2 + 4],</code>	<code>eax</code>	<code>;</code>	a második 4 byte kiírása

### Goto

Ez a konstrukció szinte minden programnyelven majdnem ugyanúgy jelenik meg, mint assemblyben, nem véletlen, hogy jóval megelőzte a strukturált programozást. Megvalósítani egy `jmp` utasítással lehet, amelynek operandusa a célcímke.

## Kifejezések

Magas szintű programozási nyelveken könnyen lehet egyetlen sorba hosszú kifejezéseket írni, ezeket assemblyre fordítva azonban akár több képernyőnyi hosszúságú kódot is kaphatunk. A kifejezések átírását a következő módszerrel tehetjük meg. Ennek a módszernek az alkalmazásához szükség van a futási idejű verem ismeretére is, lásd 25. oldal.

Alkossuk meg a kifejezés szintaxisfáját, majd járjuk be a fát posztorder sorrendben. Ha egy konstanssal vagy memóriatartalommal találkozunk, akkor azt tegyük be a verembe. Ha műveleti jelhez érünk, annak paraméterei sorban a verem tetején találhatóak; vegyük ki őket a regiszterekbe, hajtsuk végre a műveletet a megfelelő assembly utasítással, majd az eredményt tegyük ismét a verembe. Amikor a kifejezés végére értünk, a verem tetején a kifejezés értéke található.

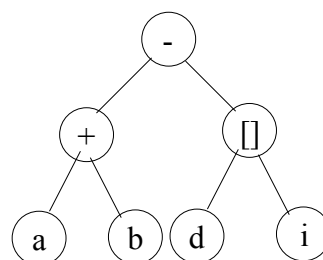
Komolyabb nyelv esetén a közbülső lépések során az eredmény veremelését típusellenőrzés előzi meg, például történt-e túlszordulás. A tömbök indexelése, lista következő elemére hivatkozás, és bármilyen olyan lépés, ami mutató vagy hivatkozás feloldását vonja maga után, szintén műveleti jelnek minősül, amit megvalósítani indirekcióval lehet.

Természetesen sokkal hatékonyabb és kényelmesebb módon is ki lehet értékelni a kifejezéseket, amennyiben a verem helyett regisztereket használunk az átmeneti értékek tárolására. Amint az ember egy kis gyakorlatra tesz szert az assembly programozás terén, kézzel elég egyszerű lesz olvashatóbb, rövidebb módon leírni egy kifejezés kiértékelését. Ugyanerre a fordítóprogramok fejlett technikákat, például gráfszínezést alkalmaznak.

Az a, b, c, i és j változók 32 bitesek, d 32 bites értékeket tartalmazó tömb.  
Valósítsuk meg a  $j := (a + b) * (c - d[i])$  értékadást!

Példa

		<u>a verem tartalma</u>	<u>eax</u>	<u>ebx</u>	<u>ugyanaz a számítás rövidebb kóddal</u>
push	dword [a]	; a			mov eax, [a]
push	dword [b]	; a b			add eax, [b] ; a + b
pop	eax	; a	b		mov ebx, [c]
pop	ebx	;	b	a	mov ecx, [i]
add	eax, ebx	;	a+b		mov edx, [d]
push	eax	; (a+b)			mov edx, [edx+4*ecx] ; d[i]
push	dword [c]	; (a+b) c			sub ebx, edx ; c - d[i]
push	dword [d]	; (a+b) c d			mul ebx
push	dword [i]	; (a+b) c d i			; a végeredmény eax-ben található
pop	eax	; (a+b) c d	i		
pop	ebx	; (a+b) c	i	d	
mov	eax, [ebx+4*eax]	; (a+b) c	d[i]		
push	eax	; (a+b) c d[i]			
pop	eax	; (a+b) c	d[i]		
pop	ebx	; (a+b)	d[i]	c	
sub	ebx, eax	; (a+b)		c-d[i]	
push	ebx	; (a+b) (c-d[i])			
pop	eax	; (a+b)	c-d[i]		
pop	ebx	;	c-d[i]	(a+b)	
mul	ebx	; feltételezzük, hogy a szorzat befér eax-be			
push	eax	; a végeredmény a verem tetején van			



## Szekvencia

Ez a legegyszerűbb vezérlési szerkezet: csak le kell írni a programrészletek kódját egymás után.

## Elágazás

Az elágazás minden ága számára tartunk fenn, még csak gondolatban, egy (lokális) címkét, valamint az elágazás vége számára is egyet. Az elágazások mindig egy kifejezés kiértékelésével kezdődnek; igaz-hamis elágazások esetén a kifejezés egy logikai értéket ad. Ezután sorban következnek az ágak kódjai.

Az elágazás elején a feltétel vizsgálata a következőképpen történik. Kiszámítjuk a feltétel értékét (ami egy logikai érték) egy változóba. Ezután megvizsgáljuk, hogy az elágazás melyik ágába kell átadnunk a vezérlést. Az értéket egy `cmp` és egy feltételes ugrás segítségével tudjuk összehasonlítani; többágú elágazás esetén több ilyen utasításpárt írunk le egymás után. Kihaszználható, hogy a vezérlés a következő utasításon folytatódik, ha nem teljesül a feltétel, ezért egy kiválasztott ág kódja közvetlenül az elágazás után következhet.

Gyakori eset, amikor azt kell megvizsgáljunk, hogy egy kifejezés értéke beleesik-e egy intervallumba. Ennek vizsgálatához, két `cmp/jmp` utasításpár szükséges: az első két utasítással vizsgáljuk meg, kisebb-e az alsó határnál, a második kettővel pedig azt, nagyobb-e a felsőnél. Mivel sem a `cmp`, sem a feltételes ugrások nem változtatják meg a vizsgálandó értéket, a konstrukció természetes módon konjunkcióként viselkedik.<sup>8</sup>

Írjunk elágazást attól függően, hogy az `a` és `b` változó értéke megegyezik-e.

```
mov eax, [a]
cmp eax, [b]
jne .hamis.ag
```

Példa

```
.igaz.ag          ; erre a címkére nincsen feltétlenül szükség, a jobb olvashatóság kedvéért szerepel
                  ; itt következik az igaz ág kódja
                  jmp .elagazas.vege

.hamis.ag
                  ; itt következik a hamis ág kódja
                  ; ide nem szükséges ugró utasítást elhelyezni, mert éppen az elágazás végénél vagyunk

.elagazas.vege
```

<sup>8</sup> nem túl gyakran előfordul, hogy ennél több összehasonlításra van szükség, például ha EBCDIC kódolás szerint vizsgáljuk meg, kisbetűről van-e szó. Ezekben az esetekben is fel lehet bontani a feltételeket intervallumok uniójára; sorban vizsgáljuk meg, hogy az egyes intervallumok közül beleesik-e valamelyikbe az érték.



## Rövidzáras operátorok

Kétfajta logikai vagy és logikai és szokott a programokban szerepelni. Az ún. **mohó operátorok** mindkét részkifejezést kiszámítják, majd az eredményekből meghatározzák a kifejezés eredményét a megfelelő logikai operátorral. A **lusta kiértékelésű** vagy más néven **rövidzáras operátorok** azonban, ha az első részkifejezésből már rögtön megadható az egész kifejezés értéke, a második részkifejezést nem számítják ki.

Egy lehetséges módszer mohó operátorok kiszámítására, hogy egyszerű kifejezéseknek tekintjük őket, a feltételeket pedig a `setfeltétel` utasítások közül a megfelelő segítségével állítjuk elő.

Rövidzáras operátoroknál a következő kódot írhatjuk. Ez több, azonos operátorból álló kifejezés-láncok kiszámítására alkalmas.

### konjunkciós lánc

; a két összehasonlítandó  
; meghatározása eax-be és ebx-be

```
cmp          eax, ebx
jfeltétel tagadása near .hamis.ag
```

; a fenti konstrukció  
; alkalmazása a konjunkciós lánc minden elemére

```
.igaz.ag      ; itt szerepel az igaz ág kódja
jmp .vege
```

```
.hamis.ag     ; itt szerepel a hamis ág kódja
```

```
.vege
```

### diszjunkciós lánc

; a két összehasonlítandó  
; meghatározása eax-be és ebx-be

```
cmp          eax, ebx
jfeltétel near .igaz.ag
```

; a fenti konstrukció alkalmazása  
; a diszjunkciós lánc minden elemére

```
.hamis.ag     ; itt szerepel a hamis ág kódja
jmp .vege
```

```
.igaz.ag      ; itt szerepel az igaz ág kódja
```

```
.vege
```

Valósítsuk meg az  $(a < b) \&\& (c - 1 < 2 * d)$  logikai feltétel kiértékelését rövidzáras operátorral!

```
mov  eax, [a]
mov  ebx, [b]
cmp  eax, ebx
jnl near .hamis.ag
```

```
mov  eax, [c]
dec  eax
mov  ebx, [d]
shl  ebx, 1
cmp  eax, ebx
jnl near .hamis.ag
```

```
.igaz.ag
...
jmp .vege
```

```
.hamis.ag
...
```

```
.vege
```

Valósítsuk meg az  $(a + b < 1) \|\| (c \leq a \& d)$  logikai feltétel kiértékelését rövidzáras operátorral!

```
mov  eax, [a]
add  eax, [b]
cmp  eax, 1
jl  near .igaz.ag
```

```
mov  eax, [c]
mov  ebx, [a]
and  ebx, [d]
cmp  eax, ebx
jnle near .hamis.ag ; az utolsó ág
; összevonható a
; jmp .hamis.ag utasítással
; (a feltétel megfordul)
```

```
.igaz.ag
...
jmp .vege
```

```
.hamis.ag
...
```

```
.vege
```

Példa

## Elöltesztelés ciklus

Elöltesztelés ciklus írásához két lokális címkére lesz szükségünk.

1. Az első címke a ciklus kezdetét jelzi, ide fogunk a ciklusmag végrehajtása után visszaugrani.
2. Ezután következik a belépési feltétel kiértékelése. Ha nem teljesül a belépési feltétel, akkor elugrunk a második lokális címkékre.
3. Ekkor következik a ciklusmag kódja.
4. A ciklusmag kódja után egy ugró utasítással átadjuk a vezérlést az első címkénkre.
5. Végül leírjuk a második címkét, amelyen a program tovább folytatódik.

Tegyük fel, hogy van egy `i` változónk, amelyet 32 biten, előjelesen ábrázolunk.

```
while ( 5 < i || i < 11 )
{
    ++i;
}
```

A fenti C programrészletet a következőképpen tudjuk megvalósítani.

```
.ciklus.kezdete                ; 1.
    cmp  dword [i], 5          ; 2.
    jg  near .ciklusmag       ; 2., 5 < i teljesül
    cmp  dword [i], 11        ; 2.
    jl  near .ciklusmag       ; 2., i < 11 teljesül
    jmp  .ciklus.vege

.ciklusmag                     ; ez a címke a vagy rövidzárassága miatt keletkezett
    inc  dword [i]            ; 3.

    jmp  .ciklus.kezdete      ; 4.

.ciklus.vege                   ; 5.
```

Példa

## Hátultesztelős ciklus

A hátultesztelős ciklus nagyon hasonló szerkezetű az előtesztelős ciklushoz. Szintén két lokális címkére lesz szükségünk.

1. Az első címke a ciklus kezdetét jelzi, ide fogunk a ciklus végén visszaugrani.
2. Itt következik a ciklusmag kódja.
3. Ezután következik a folytatási feltétel kiértékelése. Ha teljesül a feltétel, akkor elugrunk a ciklus kezdetén elhelyezett lokális címkékre, különben pedig a második címkére. A második címkére való ugrás egyszerű feltétel esetén el is maradhat, hiszen ekkor a feltételes ugráson túlhaladva a program futása ugyanott folytatódik.
4. A második címkével jelezzük a ciklus végét.

Példa: tegyük fel, hogy van két int típusú változónk, *i* és *j*, melyeket 32 biten, előjelesen ábrázolunk.

```
do
{
    --i;
} while ( 19 < i + j && i < 11 );
```

A fenti C programrészletet a következőképpen tudjuk megvalósítani.

```
.ciklus.kezdete          ; 1.
    dec    dword [i]      ; 2.

    mov    eax, [i]       ; 3.
    add    eax, [j]       ; 3.
    cmp    eax, 19        ; 3.
    jg     .ciklus.kezdete ; 3., 19 < i + j teljesül

    cmp    dword [i], 11  ; 2.
    jnl   .ciklus.vege   ; 2., i < 11 nem teljesül

    inc    dword [i]     ; 3.

    jmp   .ciklus.kezdete ; 4.

.ciklus.vege            ; 5.
```

Példa

## Rögzített lépésszámú ciklus

Amennyiben a ciklusmagot korlátos sokszor kell végrehajtani, a következő kódot generálhatjuk. Két lokális címkét fogunk elhelyezni a kódban.

- Szükségünk lesz a ciklusszámláló eltárolására. Erre a célra egyszerű ciklusok esetén fenntarthatjuk valamelyik regisztert, összetettebb ciklusok esetén pedig lefoglalhatunk egy memóriaterületet az adatszegmensben, vagy tárolhatjuk a ciklusszámlálót a futási idejű veremben.
- Miután eldöntöttük, hol tároljuk a ciklusszámlálót, töltsük fel a kezdeti értékét.
- Írjuk le az első lokális címkét, amely a ciklus kezdetét jelzi.
- Vizsgáljuk meg, hogy a ciklusszámláló túlhaladta-e már az értékhatárt. Ha igen, ugorjuk a második címkére. Amennyiben a ciklusszámláló még a határon belül van, a program futása folytatódik.
- Írjuk le a ciklusmag kódját. A ciklusmagon belül felhasználhatjuk a ciklusszámláló értékét, de ne változtassuk meg. A ciklusmagon belül elhelyezhetünk kiugrást a ciklus végére.
- Léptessük a ciklusszámlálót.
- Egy feltétel nélküli ugrással irányítsuk vissza a vezérlést az első címkére, a ciklus kezdetére.

Adott egy `int n` változó. Valósítsuk meg az alábbi programot assembly nyelven.

```
for ( int i = 0; i < 15; ++i )
{
    if ( n % 2 == 0)      n = n / 2;
    else                 n = 3 * n + 1;
    if ( 1 == n)        break;
}
```

---

```
mov eax, 14           ; eax regiszterben tároljuk az n változót
mov ecx, 0           ; a ciklusszámláló inicializálása
```

```
.ciklus.kezdete
    cmp ecx, 15
    jnl .ciklus.vege    ; a ciklusfeltétel vizsgálata

    mov edx, eax
    and edx, 1          ; a feltétel vizsgálatához csak az utolsó bitre van szükségünk
    cmp edx, 0
    jne .paratlan

.paros
    shr eax, 1          ; előjel nélküli, kettővel való osztás
    jmp .elagazas.vege

.paratlan
    mov edx, eax
    shl eax, 1          ; előjel nélküli, kettővel való szorzás
    add eax, edx        ; ... és még egyszer az eredeti
    inc eax

.elagazas.vege
    cmp eax, 1
    je .ciklus.vege    ; break

    inc ecx             ; a ciklusszámláló növelése
    jmp .ciklus.kezdete ; ugrás a ciklusmag elejére

.ciklus.vege
```

## Leszámláló ciklus

A leszámláló ciklus olyan rögzített lépésszámú ciklus, amelyben a ciklusszámláló csökken, és a ciklusfeltételt a ciklus végén ellenőrizzük. Ha nem végzünk külön ellenőrzést előtte, akkor a ciklusmag legalább egyszer lefut. Ügyelni kell arra, hogy a ciklusszámláló kezdetben ne legyen nulla, különben alulcsordulás miatt a ciklus szándékainkkal ellentétben nemhogy egyszer sem, de  $2^{32}$ -szer fut le. Amennyiben nem alkalmazunk egyéb számlálót, figyelembe kell venni a ciklusmagban, hogy a ciklusszámláló felülről lefele számol.

Szerkezete nagyon egyszerű. A ciklus kezdete előtt fel kell tölteni a ciklusszámlálót. Egyetlen címkére van szükségünk a ciklus elején. Utána közvetlenül a ciklus kódja következik, majd a ciklus végén alkalmazzuk a `loop` utasítást<sup>9</sup>. Ennek egy címke az operandusa, és a következő három utasítással ekvivalens működésű. Kötött, hogy `ecx` regisztert tekintik ciklusszámlálónak.

```
dec ecx
cmp ecx, 0
je loop_utasítás_operandusa_címke
```

Számítsuk ki az első  $n$  szám összegét ( $n \geq 1$ ).

```
int i = n;
int osszeg = 0;
do
{
    osszeg += i;
} while (--i);
```

```
mov ecx, n           ; eax regiszterben tároljuk a ciklusszámlálót
mov eax, 0           ; osszeg inicializálása
```

```
.ciklus.kezdete
add eax, ecx
loop .ciklus.kezdete ; a ciklusfeltétel vizsgálata
```

<sup>9</sup> Természetesen a vele ekvivalens kódrészletet is leírhatjuk. Ennek további előnye lehet, hogy ekkor `ecx` helyett más regisztert is választhatunk ciklusszámlálónak.

## Feladatok

1. Add meg a regiszterek tartalmát minden utasítás után! A feladatot a jelölt pontokon is el lehet kezdeni.

	reg. vagy mem.	bináris (megfelelő bithosszon)	decimális		hexadecimális
			előjel nélküli	előjeles	
a.	mov ebx, 1023	ebx			
	mov al, -100	al			
	movsx eax, bl	eax			
b.	mov eax, 1C34F6E2h	ax			
		ah			
		al			
c.	mov byte [0FAh], 01Ch	[0xFA]			
	mov byte [0FBh], 0x22	[0xFB]			
	mov ax, [0FAh]	ax			
d.	mov eax, 89ABCDEFh	ax			
	mov ah, al	ah			
	mov al, 42	al			
	mov bl, al	bl			
	mov bh, al	bx			
e.	mov ax, 0FF42h	ax			
	mov bx, 0DCBAh	bx			
	sub bh, al	bh			
	xor ax, bx	ax			
		bx			
f.	mov al, 01011010b	al			
	or ah, 10101010b	ah			
g.	mov al, 120	al			
	not al	al			
	neg al	al			
h.	mov bh, 254	bh			
	inc bh	bh			
	add bh, 1	bh			
i.	mov cx, 0x01F0b	cx			
	mov eax, -128	eax			
	mul cx	dx : ax			
	imul cl	ax			

2. a. Az x, y és z címen byte-os adatok vannak. Számítsd ki z-be (  $\text{not } x \mid y$  ) & x értékét!  
 b. Számítsd ki eax-be (  $\text{eax} - 1$  ) \* 2 értékét!

3. Adj meg minél több utasítást, amely
  - a. nullára állítja `eax` értékét, és a memória és minden egyéb regiszter tartalmát változatlanul hagyja!
  - b. nem változtatja meg a regiszterek értékét (legfeljebb a jelzőbitek regiszterének kivételével)!
  - c. `eax` 2. bitjét egyesre állítja, a többi pedig változatlanul hagyja!
  - d. az ellentétére állítja `ebx` legalsó (0.) és legfelső (31.) bitjét!
  - e. al tartalmát egy előre adott értékről egy másik adott értékre állítja! (Például 229-ről 33-ra.)
4. Az `x` és `y` két címke, melyeken 32 bites adatokat tárolunk.
  - a. Valósítsd meg az `x := y` értékadást! Közvetlenül egyik címről a másikra másoló utasítás nincsen.<sup>10</sup>
  - b. Cseréld meg `x` és `y` tartalmát!
5. Milyen byte-okat tartalmaz sorban az adatszegmens?

```

section .data

db 55h
db 55h, 56h
db 'a', 0Ah
db "hello"
dw 1234h
dd 89ABCDEFh

```

6. Valósítsd meg az ismertett utasítások segítségével az `adc eax, ebx` utasítást! Az `adc` az első operandushoz hozzáadja a másodikat (mint az `add`), és még ehhez az átviteli bitet (azaz hozzáad még egyet, ha az átviteli bit be van állítva, különben az eredmény a két operandus összege).

7. Mennyi a kódrészletek végrehajtása után `eax` értéke?

<pre> .a xor ebx, ebx inc ebx jmp .vege  dec ebx mov eax, ebx  .vege </pre>	<pre> .b cmp eax, 0 je near .vege  mov eax, eax  .vege inc eax </pre>	<pre> .c cmp eax, eax je near .vege dec eax  jmp .c  .vege </pre>
---	---	---

8. Írj olyan kódrészletet, amely `ecx`-be meghatározza `eax` és `ebx` maximumát! A számok előjel nélkül vannak ábrázolva.

9. Lineáris keresés: az adat címkétől kiindulva keresd meg az első olyan byte-ot, amelyik nullát tartalmaz, és ennek a címét add meg `eax`-ben!

10. Valósítsd meg assembly nyelven az alábbi C programrészletet! Az `a` és `b` változók típusa `int`.

```
for ( int i = 0; i < 10; ++i ) a += b;
```

11. Az `n` paraméter a memóriában található egy duplaszavas változóban. Számítsd ki az `eax` regiszterbe
  - a. az első `n` szám összegét!
  - b.  $n!$  értékét!

<sup>10</sup> Egy ilyen utasításnak egyszerre kellene a memóriabuszon beolvasnia és kiírnia az adatot, márpedig az csak egyirányú adatforgalomra képes.

12. Adott egy duplaszavas értékeket tartalmazó láncolt lista címke. A lista egy eleme egy duplaszavas mutatóból és a duplaszavas értékből áll. A mutató a következő elemre mutat; ha nincs következő elem, nullát tartalmaz. Add meg a lista hosszát!

13. Adott a szoveg címkén egy szöveg, amelynek ismert a hossza. Fordítsd meg helyben!

14. Adott három időpont: egy-egy óra és perc, amelyek duplaszavasán vannak eltárolva a memóriában. Az első két időpont között részleges napfogyatkozás következik be. A napfogyatkozás mértéke a félidőig lineárisan növekszik, félidőben pontosan 50%, onnantól lineárisan csökken. Add meg, hogy a harmadik időpontban (amelyről feltételezhető, hogy az előző kettő között helyezkedik el) mekkora volt egész százalékra kerekítve a napfogyatkozás mértéke!

15. A teniszben az a játékos nyer meg egy ún. rövidített játékot, aki először ér el legalább 7 pontot úgy, hogy legalább 2 ponttal vezet (azaz ha az állás 7-6, akkor még nem dőlt el a rövidített játék, például a második játékos megszerezheti a következő három pontot, és akkor ő nyer 7-9-re). Egy tömbben adott, hogy melyik játékos nyerte meg a soron következő játékot. A tömb hossza előre nem ismert. Add meg, ki nyerte a rövidített játékot, és milyen arányban!



## A futási idejű verem szerepe: rekurzív alprogramhívások megvalósítása

Az **alprogramok** (rutinok, szubrutinok) paraméterezhető, távolról meghívható programkód-részek. Két fajtájuk a **függvények**, melyek ezekből visszatérési értéket számolnak ki, és az **eljárások**, melyek a paraméterként kapott mutatókon keresztül megváltoztathatják a memória tartalmát. Főleg a C alapú nyelvekben a két fogalom nem különül el, ezekben mindkettőt függvénynek nevezik.

A verem célja az, hogy egyszerű módon lehessen kezelni az egymásba ágyazott alprogramhívások adatait. Akkor kap igazi jelentőséget, ha a hívások rekurzívak, azaz az alprogram futása közben ismét meghívjuk az alprogramot (általában más paraméterezéssel, mint első alkalommal). Íme egy példa arra C nyelven, miért is van erre szükség.

Hívjuk meg a függvényt **fakt( 2 )** paraméterezéssel.

Ekkor a vezérlés átkerül a függvény kódjának az elejére. Mivel  $2 \neq 0$ , az eljárás hamis ágat hajtjuk végre. Itt ki kell számítanunk  $\text{fakt}( 1 ) * 2$ -t, ehhez pedig ismét meg kell hívunk a függvényt, ezúttal **fakt( 1 )** paraméterezéssel.

*A függvény most két példányban fut: két, eltérő paraméterezéssel.*

A „belső” faktoriális még egy példányt elindít a függvényből, **fakt( 0 )** hívással, ami visszatér 1-gyel, majd ezt felhasználva **fakt( 1 )** is visszatér 1-gyel, és csak ekkor tud **fakt( 2 )** is visszatérni,  $2 * 1$ -el.

```
int fakt( int n )
{
    int v;

    if ( n == 0 ) v = 1;
    else
        v = fakt( n - 1 ) * n;

    return v;
}
```

Azokat a kódrészleteket, amelyek több példányban is futhatnak egyszerre, **reentránsnak** nevezzük. Azokat a memóriacímeket, ahová átadva a vezérlést elkezdhetjük a kódrészlet végrehajtását, a kód **belépési pontjainak** nevezzük. A programozó, illetve a fordítóprogram feladata garantálni, hogy a kódot csak ezeken a pontokon kezdjük el végrehajtani. A belépési pontokat címkével fogjuk megjelölni.

## A futási idejű verem használata

A verem egy hardveresen támogatott ábrázolású adatszerkezet. Adatot betenni a **push<sub>i</sub>**, kivenni a **pop<sub>i</sub>** utasításokkal lehet. A paraméter 16 vagy 32 bites lehet, 8 bites *nem*. A pop operandusa nem lehet konstans.

Példa	push eax push dword [esi]	push word 5 pop ecx	push al	pop dh	Hiba
-------	------------------------------	------------------------	---------	--------	------

A veremműveletek – 386-ostól felfelé – a következő műveletekkel ekvivalensek hatásukban.

push <i>adat</i>	sub esp, <i>adat hossza</i> ; 2, ha szó, 4, ha duplaszó mov [esp], <i>adat értéke</i>	pop <i>adat</i>	mov <i>cél</i> , [esp] add esp, <i>a mozgatott adat hossza</i>
------------------	--	-----------------	---

Fontos, hogy rögzítsük, hogyan kerülnek át a paraméterek a hívás helyéről az alprogramba, illetve hogyan adja vissza az alprogram a visszatérési értékét, ha van neki; ezt **hívási konvenciónak** nevezzük. A **cdecl** konvenció a paramétereket jobbról balra haladva teszi a verembe, a visszatérési értéket pedig az eax regiszterbe teszi. Ha a visszatérési érték kisebb (pl. ASCII karakter), csak ax vagy al tartalmaz értékes adatot, a felső bitek értéke figyelmen kívül hagyható; ha nagyobb struktúra, például egy rekord, eax-ban a struktúra címét adjuk vissza.

a bevezető kód az alprogram elején	push ebp mov ebp, esp sub esp, <i>lokális paraméterek össz hossza (byte)</i>	mov esp, ebp pop ebp ret	a kilépő kód az alprogram végén
------------------------------------	--	--------------------------------	---------------------------------

Az alprogramot a fenti kódrészletekkel kezdjük és fejezzük be. Az alprogramon belül az *i*-edik paramétert a **[ebp + 4 + *i* \* 4]** címen, a *j*-edik lokális változót az **[ebp - *j* \* 4]** címen érhetjük el.

Az alprogram egy futó példányához tartozó információk összességét **aktivációs rekordnak** nevezzük. Ez tartalmazza az összes paramétert és a lokális változót. A futási idejű veremben, **veremkeretekben** tároljuk az aktivációs rekordokat. Ezek megvalósításáról szól a következő oldal.

## Konkrét példa a futási idejű verem használatára: a faktoriális( 2 ) hívás megvalósítása

>>>>>>	Kezdetben a verem tartalma lényegtelen számunkra. Annyit tudunk csupán, hogy a verem tetejének a címe esp-ben található.	esp a verem alja >>>>>>>>>>
push dword 2	Amikor meghívjuk faktoriális függvényt, a paraméterét a verembe tesszük. Ezzel esp is csökken négygel, és a verem új tetejére mutat.	esp par <sub>1</sub> : 2 ...
call fakt	Ezzel az utasítással hívjuk meg a függvény kódját. Hatására a verembe bekerül a következő utasítás címe; visszatéréskor innen tudjuk majd, honnan kell folytatni a végrehajtást. Ezután a vezérlés átkerül a függvény kezdőcímére (ezt tartalmazza a faktoriális címke).	esp rég <sub>i</sub> eip par <sub>1</sub> : 2 ...
add esp, 4	Miután végrehajtottuk a függvény kódját, a veremből visszaáll eip regiszter értéke, és a végrehajtás ennél az utasításnál folytatódik. Itt még vissza kell állítanunk a vermet az eredeti állapotába, vagyis esp-hez annyit adunk hozzá, ahány byte-ot a paraméterekkel elfoglaltunk a veremből.	esp par <sub>1</sub> : 2 ...
<<<<<<	Készen vagyunk, innentől folytatódhat a program futása. Az eax regiszter tartalmazza 2! értékét.	esp ...

## Konkrét példa a futási idejű verem használatára: a faktoriális függvény kódja

fakt	A függvény kezdetét címke jelöli, hogy könnyen meghívható legyen.	rég <sub>i</sub> eip par <sub>1</sub> : 2 ... ebp+4 ebp+8 ebp+12
push ebp mov ebp, esp sub esp, 4	A szokásos bevezető kód. Az első két sor beállítja ebp-t arra a pozícióra, ahonnan a függvény végéig nem mozdul el. Ezért ebp-t a veremkeret <b>bázisának</b> nevezzük. A harmadik sor lefoglalja a helyet a lokális változónk, v számára.	esp v rég <sub>i</sub> ebp rég <sub>i</sub> eip par <sub>1</sub> : 2 ... ebp-4 ebp ebp+4 ebp+8 ebp+12
push ebx	Elmentjük a függvényben használt ebx-et. A cdecl alprogramhívási konvenció szerint az alprogramok eax, ecx és edx tartalmát változtathatják meg.	esp rég <sub>i</sub> ebx v rég <sub>i</sub> ebp rég <sub>i</sub> eip par <sub>1</sub> : 2 ... ebp-8 ebp-4 ebp ebp+4 ebp+8 ebp+12
cmp dword [ebp+8], 0 jne .hamis.ag	A második sorban szereplő elágazás feltétele és igaz ága.	
mov dword [ebp-4], 1 jmp .elagazas.vege	Az elágazás igaz ága. A v változó értéket kap, majd elugrunk az elágazás végére, hogy ne fussunk bele a hamis ágba.	esp rég <sub>i</sub> ebx v: 1 rég <sub>i</sub> ebp rég <sub>i</sub> eip par <sub>1</sub> : 2 ... ebp-8 ebp-4 ebp ebp+4 ebp+8 ebp+12
.hamis.ag		
mov eax, [ebp+8] dec eax push eax	Kiszámítjuk és a verembe helyezzük az egy szinttel mélyebben levő függvény paraméterét. Hasonló az alprogram külső példányának fent leírt meghívásához.	
call fakt add esp, 4	A függvény meghívása, majd a hívás paraméterének eltávolítása a veremből.	
mov ebx, [ebp+8] mul ebx mov [ebp-4], eax	Az előbb eax-be kiszámított fakt( n - 1 ) és az ebx-be betöltött n szorzatának kiszámítása v-be. Feltesszük, hogy a szorzat belefér eax-be.	esp rég <sub>i</sub> ebx v: 2 rég <sub>i</sub> ebp rég <sub>i</sub> eip par <sub>1</sub> : 2 ... ebp-8 ebp-4 ebp ebp+4 ebp+8 ebp+12
.elagazas.vege	Akármelyik ágon is jutottunk el ide, a veremkeret szerkezete ugyanúgy néz ki. Eltér v, azaz [ebp-4] tartalma.	esp rég <sub>i</sub> ebx v: ? rég <sub>i</sub> ebp rég <sub>i</sub> eip par <sub>1</sub> : 2 ... ebp-8 ebp-4 ebp ebp+4 ebp+8 ebp+12
mov eax, [ebp-4]	A visszatérési érték eax-be kerül.	
pop ebx	A használt regiszter visszaállítása. A hamis ágban a szorzás felülírta edx-et is, azonban ezt nem kell visszaállítani az alprogram kezdeti értékére.	esp v: ? rég <sub>i</sub> ebp rég <sub>i</sub> eip par <sub>1</sub> : 2 ... ebp-4 ebp ebp+4 ebp+8 ebp+12
mov esp, ebp pop ebp	A feleslegessé vált lokális változókat átugorjuk, majd visszaállítjuk ebp-t is. Ekkor ebp az előző szinten levő veremkeret bázisát mutatja.	esp rég <sub>i</sub> eip par <sub>1</sub> : 2 ...
ret	Visszatérünk. A program futása a <i>call faktoriális</i> hívás után következő kódsornál folytatódik.	

### A veremkeret általános szerkezete

<<<<<<<< verem	...	lokVált <sub>2</sub>	lokVált <sub>1</sub>	rég <sub>i</sub> ebp	rég <sub>i</sub> eip	par <sub>1</sub>	par <sub>2</sub>	...	az előző aktivációs rekordok >>>>>>>>>>
ebp-...	ebp-8	ebp-4	ebp	ebp+4	ebp+8	ebp+12	ebp+...		

## Parancssori paraméterátadás

A parancssori paraméterek átadása rendszerenként különbözik, azon belül szerkesztőprogramok között is eltérő lehet. Linux alatt, gcc szerkesztővel nagyon kényelmesen érhetőek el a paraméterek, mivel a főprogram is úgy viselkedik, mint egy `int main( int argc, char** argv )` szignatúrájú C függvény. Ennek megfelelően az előzőekben leírt programparaméter-elérési módszerek alkalmazhatóak.

```

section .text
global main
main
push ebp
mov ebp, esp           ; alprogram bevezető kódja

mov eax, [esp+4]      ; parancssori paraméterek száma plusz egy,
                    ; ami argv[0], a futtatott program neve

cmp eax, 2
jne .nem_egy         ; ha nem pontosan egy parancssori paramétert kaptunk, kilépünk

mov esi, [ebp+12]    ; esi = argv
mov esi, [esi+4]    ; esi = argv[1], az első paraméter

.szamlal
mov al, [esi]
cmp al, 0
je .tovabb

inc esi              ; byte-onként léptetjük a mutatót, mert a szöveg ASCII karakterekből áll
jmp .szamlal        ; megszámloljuk, milyen hosszú (ASCII string)

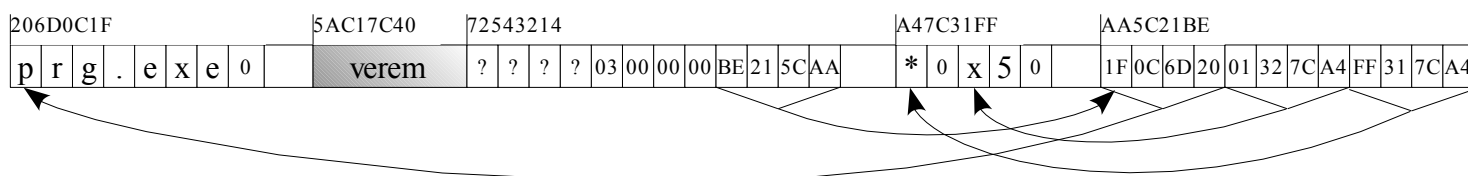
.tovabb
mov edx, esi
sub edx, ecx        ; hossz: a nulla tartalmú végpozíció – a szöveg kezdete

mov eax, 4          ; kiírás következik, bővebben lásd később
mov ebx, 1
mov ecx, [esp+8]
mov ecx, [ecx+4]
int 0x80

.nem_egy
...

```

A memóriában ezek egy potenciális elhelyezkedése két paraméter ('x5' és egy csillag) esetén a következő. A verem 5AC17C40 és 72543214 hexa címek között található, azaz 98872693 duplaszó fér bele. Az argc paraméter címe 72543218 (értéke 3), argv címe 7254321B. A filenév címe, argv értéke AA5C21BE, az első paraméterre mutató argv[1] címe AA5C21C2, a második paraméterre mutató argv[2] címe AA5C21C6. A filenév szövegesen a 206D0C1F és 206D0C26 címek között helyezkedik el (a végén egy hexa nullával), az első paraméter A47C3201 és A47C3203 között, a második A47C31FF és A47C3200 között.



## A rendszerszolgáltatások elérése: fájlkezelés

művelet	név <sup>11</sup>	eax	ebx	ecx				edx				vissza eax						
fájlműveletek	létrehozás	creat	<b>8</b>	a fájlnev címe	elérési jogok				-				fájlleíró					
	megnyitás	open	<b>5</b>		0	olvas	1	ír	2	ír/olvas	elérési jogok				fájlleíró			
	olvasás	read	<b>3</b>	a fájl leírója	puffer címe				max. hossz (byte)				olvasott byte-ok					
	írás	write	<b>4</b>		puffer címe				hossz (byte)				írt byte-ok száma					
	pozicionálás	lseek	<b>19</b>		előjeles eltolás az edx-ben kapott pozícióhoz képest				0	előlről		2	hátról		pozíció a fájlban			
	lezárás	close	<b>6</b>		-				-				0	OK				
	kilépés	exit	<b>1</b>		hibakód	-				-				-				

Az operációs rendszer szolgáltatásait **megszakítások** meghívásával lehet elérni. Ezek két dologban térnek el az alprogramhívástól: egyrészt a paramétereket más konvenció szerint, a regisztereken keresztül adjuk át, másrészt nem a call, hanem az **int** utasítást használjuk, mert más védetségű szintű kódot – az operációs rendszerét – érjük el. Az int-nek paraméterül mindig **80h**-t fogunk adni, ami a Linux rutinyűjteményének sorszáma a megszakítások között. Más operációs rendszerek is nyújtanak hasonló szolgáltatásokat, általában más sorszámú megszakítás alatt, és teljesen eltérő paraméterezéssel.

A szolgáltatások megváltoztatják az eax, ecx és edx regisztereket. Hiba esetén -1-et adnak vissza.

Egy fájl életciklusa a következőképpen alakul.

- létrehozás: ez a megnyitás egy speciális fajtája, a creat egy részben előre felparaméterezett open
- megnyitás
  - a megnyitandó fájlnev, amelynek a címét paraméterként kapja, egy nullára végződő ASCII string
  - az elérési jogok: egy-egy bit jelzi, hogy a felhasználó/csoport/külvilág írhatja/olvashatja/futtathatja-e a fájl<sup>12</sup>. Ezt a legegyszerűbb nyolcas számrendszerben leírni<sup>13</sup>.
  - siker esetén egy 32 bites leírókat kapunk vissza, innentől ennek a segítségével hivatkozhatunk a fájlra. Előre megnyitott fájlok a **sztenderd bemenet**, **kimenet** és **hibafolyam**. Ezek leírói sorban 0, 1, és 2.
- írás, olvasás
  - meg kell adnunk egy mutatót a forrás-, illetve célterületre (ez a **puffer**), és azt, hány byte-ot szeretnénk átvinni.
  - a fájlhoz tartozik egy mutató, amely kezdetben a fájl elején áll. Ezt minden olvasási és írási művelet mozgatja, de közvetlenül is lehet állítani. Figyelni kell arra, hogy a fájl végéről negatív értékkel tudunk előrébb pozicionálni. A fájl végén, ha túlmutatunk rajta, az átlépett pozíciókon 0 byte-ok szerepelnek.
- lezárás: a program befejeződésekor a rendszer lezárja a nyitva maradt leírókat, de kézzel is megtehetjük.

Példa	<pre> mov eax, 4      ; kiírást fogunk igénybe venni mov ebx, 1      ; a kiírás a sztenderd kimenetre történik mov ecx, uzenet ; az üzenet címe mov edx, 12     ; a kiírandó üzenet hossza int 80h                 </pre>	<pre> section .data uzenet      db  "Hello"             db  ' '             db  "vilag"             db  0xA                 </pre>
	<pre> mov eax, 1      ; kilépés következik xor ebx, ebx    ; nullás hibakód: minden rendben történt int 80h                 </pre>	

11 további információk találhatóak az `/usr/include/asm/unistd.h` fájlban, illetve a `man 2 név` parancs használatával

12 például a „bárminek bármit” 777q, a „felhasználónak bármit” 700q, a „felhasználó írhatja, bárki olvashatja” 644q

13 egy nyolcas számrendszerbeli szám 0 és 7 közötti számjegyeket tartalmaz, és 'q' vagy 'o' áll a végén

## Feladatok

- Írd ki a „Hello világ!” szöveget betűről betűre bővülve! Először csak egy „H” betűt írd ki és egy sorvégét, aztán a „He” szöveget és sorvégét stb.
  - Hasonlóan bővülő módon írd ki az első parancssori paramétert!
- Alkoss a saját (lefordított, futtatható) programkódját kiíró programot<sup>14</sup>! Feltételezheted, hogy a program neve ismert. Használj 1024 bites puffert a fájl tartalmának beolvasása során!
    - Nehezítés: a program nevét a parancssori paraméterek közül kell kinyerned.
  - Alkoss a saját forrásszövegét kiíró programot! Feltételezheted, hogy a program neve ismert.
    - Nehezítés: a program neve a futtatott fájl neve .asm kiterjesztéssel.  
Feltételezheted, hogy a fájl neve kevesebb, mint 256 karakter hosszú.
- Írj olyan eljárást, amely egy duplaszón elférő, előjel nélküli egész számot ír ki a képernyőre tízes számrendszerben! A számot paraméterként kapja meg az eljárás. A kiírandó string legyen lokális változó (fix hosszúságú ASCII karaktertömb); az eljárásnak ezt töltsse fel, majd hívja meg a kiírást.
- Írj programot, amely a következőképpen működik. A program két paramétert kap: egy filenevet és egy három karakteren ábrázolt, előjel nélküli számot, n-et, amely értéke ábrázolható egy byte-on. Olvasd fel a file első byte-ját, m-et. Vizsgáld meg, a file n-edik és m-edik byte-ja megegyezik-e.

Példa

A program meghívása: `./a.out 004 file.txt` ekkor  $n = 4$

A file.txt tartalma (hexa számok):

`02 07 FD 04 ...` ekkor  $m = 2$ , a második byte 7, a negyedik 4, nincs egyezés,  
`01 AB 7C 01 ...` ekkor  $m = 1$ , az első és a negyedik byte egyaránt 1, egyeznek

- Írj programot, amely a következőképpen működik. A program egy filenevet kap paraméterként. A file hossza hárommal osztható. A program vizsgálja meg, hogy minden harmadik byte egyenlő-e az előző kettő összegével (modulo 256, azaz 0xAB 0xCD 0x78 helyes).
- Írj programot, amely a következőképpen működik. A program egy filenevet kap paraméterként. A program keresse meg, melyik az az első pozíció, ahol a file előlről és hátulról olvasva eltér. Feltételezhető, hogy van ilyen pozíció.

Példa

Példa: ha a file tartalma abcdecba, akkor a program eredménye 4.

- Add meg, hogy a parancssori paraméterként kapott file a legvégén tartalmazza-e a saját nevét.
- Add meg a következő függvények kódját:

$$\begin{aligned} f(0) &= -2 \\ f(1) &= 1 \\ f(n) &= f(n-1) - 2 * (f(n-2) \text{ or } f(n-3)) \end{aligned}$$

$$\begin{aligned} g(0) &= 2 \\ g(1) &= -1 \\ g(n) &= \\ &\text{ha } f(n-1) \geq f(n-2) : (f(n-1) - n) \text{ xor } f(n-2) \\ &\text{különben : } f(n-1) \text{ xor } (f(n-2) + n) \end{aligned}$$

<sup>14</sup> A feladat jellegéből következően a kilisztázott tartalom meglehetősen olvashatatlan lesz, érdemes a diff programot használni az eredmény vizsgálatára.

9. Adott egy inteket tartalmazó tárhely kezdőcíme és a hossza.

a. Adott egy int kovetkezo() szignatúrájú függvény. Töltsd fel a tárhelyet sorban az 1, 2, 3 stb. számokkal úgy, hogy a függvény mondja meg, hány pozíciót kell az adott értékkel feltölteni!

Példa

Ha a visszatérési értékek sorban 1, 3, 2, 1, akkor a feltöltött tárhelynek így kell kinéznie: 1, 2, 2, 2, 3, 3, 4.

b. Adott két index, melyekre teljesül, hogy  $0 < i \leq j \leq \text{hossz}$ , valamint egy int f( int ) szignatúrájú függvény. Töltsd fel a tömböt úgy, hogy tetszőleges k indexre  $f(k) + \text{tomb}[k-1]$  kerüljön a tömbbe, ha  $i \leq k \leq j$ , különben pedig k.

c. Adott egy int permutal() szignatúrájú alprogram. Ennek segítségével permutáljuk a tömböt helyben a következőképpen. A permutal() első meghívásakor azzal a pozícióval tér vissza, ahová az eredeti első elemnek kell kerülnie. A következő híváskor a visszatérési érték az előbb lecserélt elem új pozíciója lesz. Ez addig folytatódik, amíg az első pozícióhoz vissza nem jutunk.

Példa

Ha az eredeti tömb tartalma 6, 2, 9, -4, 3, a visszatérési értékek sorban 3, 5, 4, 1, 2, akkor a tömb új tartalma -4, 3, 6, 9, 2 lesz.

10. Valósítsd meg a következő függvényt!

```
int osszegzo( int* tomb, int hossz )
{
    int i, j;
    int osszeg = 0;

    for ( i = 0; i < hossz; ++i )
        for ( j = i; j < hossz; ++j )
        {
            if ( tomb[i] <= tomb[j] )    osszeg = osszeg + tomb[i];
            else    osszeg = osszeg + tomb[j];
        }
    return osszeg;
}
```

## Makrók

Az eddigiekben az assembly nyelvek fordításának utolsó fázisát tárgyaltuk, amikor az assembler a forrásprogramból gépi kódot állít elő, amely **futási időben** működik: megváltoztatja a regiszterek értékét, alprogramokat hív meg stb. Ebben a fejezetben a **fordítási időben**, az **előfordítási fázisban** működő makrókkal foglalkozunk. Ezek a forráskód általunk megírt szövegét alakítják át; csak azután kezd el működni a gépi kódra fordító fázis, hogy a makrófordító fázis véget ért. Tartsuk végig szem előtt, hogy bármennyire is hasonló dolgokat lehet elvégezni a makrók segítségével, mint az utasításokkal, lényeges különbség, hogy a makrók nem férnek hozzá a regiszterekhez és a memóriához, és nem tudnak elugrani a kód más pontjaira.

A nasm assembler **-e** kapcsolójával lehet az előfordítási fázis végeredményét kiírni.

A **szimbólumok** a fordítóprogram által értelmezett, a forráskódban előforduló karaktorsorozatok. Már találoztunk a szimbólumok egy fajtájával: a címkék egy memóriacím szimbolikus megjelenési formái. A mnemonikok és a regiszterek nevei nem szimbólumok, hanem kulcsszavak; ezeket a neveket is fel lehet venni a szimbólumok közé is, de határozottan nem ajánlott. A **makrók** a szimbólumok paraméterezhető, felüldefiniálható fajtája.

Az előfordítási fázis a következőképpen működik. Az assembler lineárisan végighalad a kódon, és sorban azt vizsgálja, hogy talál-e makródefiníciókat vagy makróhívásokat. A **makródefiníció** tartalmazza a makró **nevét**, **paramétereit** és **törzsét**, amely függ a paramétereiktől. Innentől, ha az assembler olyan szimbólumsorozattal találkozik, amely a makró nevével kezdődik, és megfelelő számú szimbólummal folytatódik (úgy, hogy azok értelmezhetőek legyenek a makró paramétereiként), azt makróhívásnak tekinti, és **kifejti**. Ez úgy történik, hogy a kifejtendő kódrészletet, vagyis a makró nevét és a paraméterezést, eltávolítja a forrásszövegből, majd a helyébe beszurja a makró törzsének az aktuális paraméterek szerint meghatározott alakját. A fordítóprogram ezután folytatódik a kifejtett rész *kezdetétől*, ezért a kifejtés után rögtön annak a vizsgálata következik, hogy a beillesztett részben található-e újabb kifejtendő makróhívás.

Figyelem: az alprogram paraméterei, a makró paraméterei és a parancssorból átvett paraméterek három teljesen különböző fogalom! Adódhat feladat, mely során olyan makrókat kell írni, amely paraméterül kapja, hogy egy alprogram melyik paraméterei jelentenek fájl-paramétert, és ennek megfelelően kezeli őket...

### Egysoros makrók

A makróknak egy egyszerű fajtája az **egysoros makró**. Egyszerűsége abból fakad, hogy egyetlen sorra tud csak kifejtődni, ami legfeljebb egy utasítást tartalmazhat, ezért bonyolultabb kódot nem lehet vele generálni. Az egysoros makrókat a **%define makrónév tartalom** vagy a **%xdefine makrónév tartalom** kulcsszó vezeti be, utána kell leírni a makró nevét, majd zárójelek között, ha vannak, a **formális paramétereket**, majd a makró törzsét. A formális paraméter szintén szimbólum, egy névvel jelöli meg a paraméter pozícióját; kifejtéskor a törzsben minden előfordulása lecserélődik a makróhívás megfelelő pozícióján szereplő aktuális paraméterre. A makródefiníciók **felüldefiniálhatóak**: az azonos nevű és paraméterszámú makródefiníciók közül az utoljára elemzett van érvényben.

Példa

A tartalomban érdemes megfelelően zárójellezni a paramétereket.

```
%define helytelen(a, b)      a*b
%define helyes(a, b)        ((a)*(b))

mov eax, helytelen(2, 3 + 4) ;erre fejtődik ki: mov eax, 2 * 3 + 4
mov eax, helyes(2, 3 + 4)   ;erre fejtődik ki: mov eax, ((2) * (3 + 4))
```

Az első sor is érvényes utasítás, de a programozónak valószínűleg nem ez volt a szándéka.

A **%xdefine** annyiban tér el a **%define**-től, hogy míg az utóbbit alkalmazva a makró értéke a szövegszerűen lemásolt törzs lesz, addig az előbbi a törzs kifejtett alakját veszi fel a szimbólum tartalmának, ezért az szövegszerűen nem tartalmaz már makrókat. Ennek akkor van szerepe, ha a definíció és a kifejtés helye között felüldefiniálunk egy olyan makró, amely szerepel a törzs szövegében: a **%define** esetében ez hatással lehet az eredeti makróra, a **%xdefine** esetében nem.

Példa	<code>%define a 1</code>	
	<code>%define b a</code>	
	<code>%xdefine c a</code>	
	<code>%define a 2</code>	
	<code>mov eax, b</code>	<code>; erre fejtődik ki: mov eax, 2</code>
	<code>mov eax, c</code>	<code>; erre fejtődik ki: mov eax, 1</code>

A *nullától eltérő* paraméterszámú makrók **túlterhelhetőek** is: érvényben lehet egyszerre több olyan makródefiníció is, amelyeknek ugyanaz a neve, de eltérő a paraméterszáma. A makródefiníciók **nem rekurzívak**: ha a kifejtésen belül újra találkozunk az assembler ugyanazzal a makróval, azt nem fejt ki még egyszer.

Az egysoros makrók további, speciális fajtája a fordítási időben történő **értékadás**. A **%assign szimbólumnév érték** kulcsszó után egy szimbólumnév következik, majd a sor további része egy *fordítási időben kiértékelhető* kifejezés<sup>15</sup>, amely meghatározza a szimbólum felvett, új értékét. Ez hatásában nagyon közel áll ahhoz, mintha **%define** segítségével hoztunk volna létre egy szimbólumot, a különbség egyrészt abban áll, hogy a **%define** nem csak számokkal dolgozhat, másrészt pedig a **%assign** a definíció során **kiértékeli** a kifejezést, ezért tömörebben tudja az assembler ábrázolni. További előnye, hogy ránézésre egyből látszik róla, hogy szám jellegű szimbólummal dolgozunk.

Példa	<code>%assign n 1</code>	<code>%define n 1</code>
	<code>%assign n n+1</code>	<code>%xdefine n n+1</code>
	<code>... ; összesen hússzor</code>	<code>... ; összesen hússzor</code>
	<code>%assign n n+1</code>	<code>%xdefine n n+1</code>
	<code>mov eax, n</code>	<code>mov eax, n</code>
	<code>; erre fejtődik ki: mov eax, 21</code>	<code>; erre fejtődik ki: mov eax, 1+1+...+1</code>

Ezzel rokon konstrukció a **konstans szimbólum**. Alakja: **szimbólumnév equ kifejezés**. Fontos, hogy a kifejezésnek *fordítási időben kiértékelhetőnek* kell lennie. A szimbólum felveszi a kifejezés értékét, mintha **%assign** segítségével definiáltuk volna, azonban innentől nem változtatható meg az értéke.

Példa	<code>negyvenketto equ 4 * 10 + 2</code>
	<code>mov eax, negyvenketto ; ekvivalens ezzel: mov eax, 42</code>

<sup>15</sup> konstansok, makrókkal definiált szimbólumok és azokkal végzett műveletek; ha ide egy regiszter, pl. `eax` nevét írjuk, az nem hiba, de a fordítóprogram érdeklődni fog, hogy mi az `eax` szimbólum tartalma, ha nem definiáltuk



## Többsoros makrók

Bonyolultabb működés leírására alkalmasak a **többsoros makrók**. Ezek törzsét a **%macro** és a **%endmacro** (rövidíthető: **%endm**) kulcsszavak közé lehet leírni. A kulcsszavakat külön sorba kell írni; a %macro kulcsszó után a sorban következik a makró neve, a paraméterek száma, és ha vannak, az alapértelmezett paraméterek.

**Makródefiníciókat törölni** – akár egysorosakat, akár többsorosakat – **%undef makrónév** leírásával lehet.

A **paraméterek számának** alakjai a következők lehetnek.

- egyetlen szám: pontosan ennyi paraméterrel kell meghívni a makrót
- két szám kötőjellel elválasztva: a paraméterszámnak bele kell esnie ebbe a (zárt) intervallumba
  - a paraméterek száma valójában a második szám; meg kell adni vesszőkkel elválasztva, milyen értékeket vegyenek fel azok a paraméterek, amelyeket a makróhívás nem adott meg explicit módon
- egy szám, kötőjel és csillag: a paraméterszám legalább annyi, mint az első szám, felső korlát nincs
  - a makró törzsének írásakor nem ismert, hány paramétert kapott
  - az **aktuális paraméterek számát %0** tartalmazza.

A makrókat lehetséges specializálni is, azaz egyes paraméter-kombinációkhoz külön kódot is meg lehet adni.

A **formális paraméterekre**, mivel most nincsenek névvel ellátva, **%1, %2** stb. leírásával lehet hivatkozni a törzsben. Főleg többsoros makrók használatánál fordul elő, hogy **vesszőt is tartalmazó paramétereket** is át szeretnénk adni. Ezeket *kapcsos zárójel*ek közé kell tenni híváskor, mert különben az assembler külön makróparamétereknek nézné őket. Amikor a törzsben felhasználjuk őket, már nincs körülöttük a kapcsos zárójel, ezért ha egy másik makrónak szeretnénk átadni őket, akkor ismét be kell kapcsos zárójelezni őket.

Példa	<code>%macro egyop 2</code>	<code>%macro ketop 3</code>	<code>%macro hanyop 1-*</code>
	<code>  %1 %2</code>	<code>  %1 %2, %3</code>	<code>  mov eax, %0 - 1</code>
	<code>%endm</code>	<code>%endm</code>	<code>%endm</code>
	<code>  egyop inc, eax</code>	<code>  ketop mov, eax, ebx</code>	<code>  hanyop 1, 2, 3, 4, 5</code>
	<code>; erre fejtődik ki: inc eax</code>	<code>; erre fejtődik ki: mov eax, ebx</code>	<code>; erre fejtődik ki: mov eax, 5 - 1</code>

**Feltételes ugró utasítások feltétel-részét** is átadhatjuk makróparaméterként, a makró törzsében pedig a j betű után fűzve megkapjuk a megfelelő feltételes ugró utasítást. A j betű után **%-1** fűzésével az ellentétes hatású ugró utasítást kapjuk; természetesen 1 helyett tetszőleges sorszám szerepelhet.

Példa	<code>%macro felteteles 5</code>
	<code>  cmp %1, %2</code>
	<code>  j%+3 %4</code>
	<code>  j%-3 %5</code>
	<code>%endm</code>
	<code>  felteteles eax, ebx, ne, .nem_egyenlo, .egyenlo</code>
	<code>; erre fejtődik ki: cmp eax, ebx</code>
	<code>; jne .nem_egyenlo</code>
	<code>; je .egyenlo</code>

A **makróra lokális szimbólumokat** `%%szimbólumnév` alakban készíthetünk. Ezek a makró kifejtése során egyedi nevekre fordulnak le, vagyis a makró két kifejtésével különböző neveket kapunk ugyanabból a lokális szimbólumból. Ennek akkor van szerepe, ha címkékre vagy számítások végzéséhez szimbólumokra van szükségünk, azonban azt szeretnénk, hogy a makró kifejtése előtt definiált szimbólumokat a kifejtés után is fel lehessen használni: a makró ne változtassa meg őket. Ezért nem egy véletlenszerűen választott szimbólumnevet használunk a makrón belül, amelyről nem tudjuk, létezik-e már, és kockáztatjuk, hogy felülírjuk, hanem egy olyan lokális nevet vezetünk be, amelyről az assembler garantálja, hogy a makrón belül jelenik meg először.

Százalékos prefixszel csak konkrét makróparaméter-pozíciókra lehet hivatkozni. Nem konstansként megjelenő (pl. paraméterként kapott) indexű makróparamétert a **makróparaméterek forgatásával** lehet elérni. Tipikusan ilyen eset áll elő, amikor egy nemkorlátos paraméterszámú makró paramétereit szeretnénk elérni. Ennek eszköze a **%rotate** kulcsszó: utána egy *fordítási időben kiértékelhető* kifejezést írva megváltoztathatjuk a makróparaméterek sorrendjét. Például `%rotate 3` hatására `%1` az eddigi `%4` értékét veszi fel, `%2` `%5`-ét stb., illetve a régi `%1`, `%2` és `%3` innentől `%9`, `%10` és `%11`-ként érhetőek el, ha összesen tizenegy paraméterünk volt. A `%rotate` után negatív szám is állhat, ekkor a másik irányba forognak a makróparaméterek.

Példa	<code>mov eax, %1 + %2</code>	<code>db %3, %2, %4</code>	<code>xor eax, %xyz</code>	<code>inc %%0</code>	Hibás
	<code>push %5</code>		<code>sub ecx, %%a</code>	<code>mov eax, %{%0}</code>	
			<code>add edx, %{%b}</code>		

Ha a forráskód egy részét szeretnénk konstans sokszor megismételni, a **%rep** és **%endrep** kulcsszavak között írhatjuk le. A `%rep` után *fordítási időben kiértékelhető* kifejezésként kell megadni, hányszor akarjuk kifejtetni az ismétlendő kódrészletet. Ez a konstrukció makrókon kívül is használható, azonban a leggyakrabban nemkonstans paraméterszámú makrók paramétereinek bejárására alkalmazzák úgy, hogy a mag utolsó sora egy `%rotate`-tel forgatja a makróparamétereket.

Írjunk olyan makrót, amely a kapott paramétereinek összegét számolja ki `eax` regiszterbe!

```

%macro osszeg 0-*
    %define %%masikmakro %1
    %define %%kezdet %2
    %define %%darab %3
    %rotate %%kezdet + 2
    %rep %%darab
        %%masikmakro %1
        %rotate 1
    %endrep
%endm

    forgat mar_definialt_makro, 2, 3, "a", "b", "c", "d", "e", "f", "g"
; ennek hatására a következő makróhívások keletkeznek a kódban:
;     mar_definialt_makro "b"
;     mar_definialt_makro "c"
;     mar_definialt_makro "d"
; azok pedig továbbfejtődnek annak a makrónak a kódja szerint

```

## Feltételes fordítás makrókkal

Gyakran előfordul, hogy egy kódrészletet csak akkor akarunk belefordítani a kódba, ha valamilyen *fordítási időben kiértékelhető* kifejezés teljesül. Természetesen ezt kézzel is meg tudnánk tenni, azonban nagyon kényelmes lehet, hiszen így a forráskód egyetlen pontját megváltoztatva több helyen is megváltoztathatjuk a program működését. Jellemző alkalmazás a hibakeresés, amikor egy hibakereső szimbólumot átállítva, annak tartalmától függően, a kódba különböző hibakezelési, naplózási stb. részletek is belekerülnek a megfelelő pontokon.

Az általános feltételes fordítást a **%if feltétel** konstrukcióval végezhetjük. Ennek hatására az ettől az **%endif** kulcsszóig terjedő kódrészlet csak akkor jelenik meg a kódban, ha a feltétel fordítási időben igazra értékelődik ki. A feltételes fordításhoz további ágakat írhatunk **%elif feltétel** alakban, illetve lezáró ágot **%else** alakban.

```
Példa %macro eliranyit 3-5
      %if %0 = 3
          %1 %2, %3
      %elif %0 = 4
          %1 %2, %3, %4
      %else
          %1 %2, %3, %4, %5
      %endif
%endm

      eliranyit hivott_makro, 1, 2
      eliranyit hivott_makro, 1, 2, 3
      eliranyit hivott_makro, 1, 2, 3, 4
```

; ennek hatására az első paraméterben megadott makrónak mindig a megfelelő paraméterszámú változata hívódik meg

Hasonló konstrukciókkal ellenőrizhetjük makrók létét. Egysoros makrók létének vizsgálatára alkalmas az **%ifdef makrónév**, amelyet előszeretettel alkalmaznak a fent említett hibakeresésre; ennek a további ágait **%elifdef makrónév** alakban adhatjuk meg. Többsorosakat az **%ifmacro makrónév paraméterszám** segítségével vizsgálhatunk, ahol a paraméterszám lehet egy konkrét szám, illetve egy intervallum.

A fentiekben hasznos lehet a **%error üzenet** direktíva, amely fordítási időben kiírja az utána megadott üzenetet.

```
Példa %macro ellenor 2
      %ifdef %1
          %1 %2
      %else
          %error Nem található a makro: %1
      %endif
%endm

      eliranyit nem_letezo_makro, 12456
```

; ennek hatására a fordítóprogram ezt írja ki:

```
a.asm: warning (ellenor:4) Nem található a makro: nem_letezo_makro
```

Még egy hasznos direktíva: fájlt beilleszteni a forráskód adott pontjára így lehet: **%include "fájlnév"**.

Írjunk olyan makrót, amely tetszőlegesen sok paramétert kaphat. A paraméterek alprogramok címei, amelyek egy duplaszavas paramétert várnak. A makró generáljon olyan főprogramot, amely sorban meghívja az alprogramokat. A főprogram paraméterei 2 karakteren ábrázolt decimális számok; ha van elég parancssori paraméter, akkor az n-ediknek meghívott alprogram kapja paraméterként az n-edik parancssori paraméter értékét, ha nincs, akkor nullát.

```
%macro alprogramhivo 0-*
main
    %assign %%i 0

    %rep %0
        mov     edx,      [ebp+12]
        mov     edx,      [edx+4+4*%%i]
        mov     al,      [edx]
        mov     cl,      10
        mul     cl
        add     al,      [edx+1]
        cmp     dword [esp+4], %%i + 1
        setbe  dl
        dec     dl
        and     al,      dl
        movzx  eax,      al
        push   eax
        call   %1
        add    esp,      4

        %rotate 1
        %assign %%i %%i + 1
    %endrep
%endm
```

## Feladatok

1. Írd meg az `adc` utasítást általánosan, kétparaméteres makróként. Az utasítás leírása egy korábbi feladatban szerepel.

2. Mit kapunk, ha kiíratjuk az adat címke tartalmát?

```
%macro rajzol 4
    db %1
    %rep %2
        db ' '
    %endrep
    db %3
    %rep hossz - ( 2 * %2 )
        db %4
    %endrep
    db %3
    %rep %2
        db ' '
    %endrep
    db %1
    db 0ah
%endmacro
```

```
%define hossz 7
```

**adat.a**

```
rajzol ' ', 1, 'v', 'v'
rajzol ' ', 0, '=', '='
rajzol '|', 2, 'X', 'X'
rajzol '|', 0, ' ', ' '
rajzol '|', 2, '~', '-'
rajzol '|', 0, ' ', ' '
rajzol ' ', 0, 'v', '-'

db 0
```

```
%macro rajzol 5
    db %1
    %rep %2 - 1
        db ' '
    %endrep
    db %3
    %rep %4 - %2 - 1
        db ' '
    %endrep
    db %5
    %rep hossz - %4 - 1
        db ' '
    %endrep
    db %1
    db 0ah
%endmacro
```

```
%assign hossz 8
```

**adat.b**

```
%assign i 0
%rep 3
    rajzol ' ', 4 - i, '/', 5 + i, '\'
    %assign i i + 1
%endrep

db '+'
db '-----'
db '+', 0ah

rajzol '|', 5, 'o', 6, 'o'
rajzol '|', 5, 'o', 6, 'o'
rajzol '|', 1, ' ', 2, ' '
rajzol '|', 4, '|', 5, '|'

db '======'
db 0
```

3. a. Készíts deriváló makrót. A makró tetszőleges számú paramétert kaphat, amelyek egy polinom együtthatóit adják. A makró tegye a verembe sorban a polinom deriváltjának együtthatóit szavakként! A deriválás szabálya:

$$(c \cdot x^n)' = c \cdot n \cdot x^{n-1}$$

b. Készíts integráló makrót a deriváló makróhoz hasonlóan. A konstans tagot nem kell figyelembe venni. Feltehető, hogy minden kapott együttható egész szám.

**Példa**

A derivál 1, 5, -4, 6 makróhívás esetén a polinom  $1 \cdot x^3 + 5 \cdot x^2 - 4 \cdot x^1 + 6 \cdot x^0$ , a derivált  $3 \cdot x^2 + 10 \cdot x^1 - 4 \cdot x^0$ , a vermelendő duplaszavak tehát 3, 10, -4.  
Az integrál 6, -2, 3 hívás esetén az integrál  $2 \cdot x^3 - 1 \cdot x^2 + 3 \cdot x^1 + c$ , a vermelendő duplaszavak: 2, -1, 3.

#### 4. Készíts torpedó programot.

- a. Az adatok generálása makróval: egy olyan byte-tömböt kell létrehozni, amely a paraméterként kapott pozíciókon 1-et (van hajó), különben 0-t (nincs hajó) tartalmaz. A makró tetszőleges számú paraméterrel hívható, a pozíciók sorrendben egymás után jönnek.

Példa

A hajo 2, 4, 7, 8, 9, 11 makróhívás szerint a következő tömböt kell kapnunk:

```
db 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1
```

#### b. A program menete:

- mi lövünk
  - beolvassuk a bemenetről, melyik pozícióra lövünk
  - meghívjuk az `int mi_lovunk( int pozíció )` szignatúrájú C függvényt vele
    - ha az egyet ad vissza, találatot értünk el
    - ha kettőt, egyúttal az ellenfél minden hajója elsüllyedt. Ekkor hívjuk meg a `void nyert( void )` szignatúrájú C függvényt, majd lépünk ki.
    - különben nincs találat
- ők lönek
  - az `int ok_lonek( void )` szignatúrájú C függvény adja meg, melyik pozícióra lőttek
    - ha volt ott hajónk, elsüllyed (nullázódik a pozíció)
    - ha minden hajónk elsüllyedt, hívjuk meg a `void vesztett( int általunk_kilött_hajók_száma )` szignatúrájú C függvényt, és lépünk ki.

5. Írj olyan makrót, amely 1 + legalább még 2 paramétert vár, és olyan kódot generál, amely eldönti, hogy a paraméterek (amelyek regiszterek és memóriatartalmak lehetnek) szigorúan növekvő sorrendben vannak-e. Amennyiben nem, akkor ugorjunk az első paraméterben megadott címkére.

Példa

Ha így hívjuk meg: `vizsgal címke, eax, {dword [x]}, esi`  
akkor azt kell megvizsgálnia a kódnak, hogy `eax < dword [x] < esi` teljesül-e.

6. Adott egy `exp` nevű, háromparaméteres makró: `exp x, 3, 4` beállítja az `x` szimbólumot 34 értékére. Készítsd el az `ellenor n, m` makrót, ami ellenőrzi, hogy minden `k`-ra egytől `m`-ig helyesen számítja-e ki az `exp` makró `nk`-t. Azt kell megvizsgálni, hogy a makrót két szomszédos `k`-ra meghívva, a nagyobb érték megegyezik-e a kisebb érték `n`-szeresével. Ha jól működik a makró, állítsuk be az `ok` szimbólumot egyre, ha pedig nem, akkor szüntessük meg az `ok` szimbólumot.

7. Adott egy `Fibonacci` nevű, kétparaméteres makró: `Fibonacci x, 4` beállítja az `x` szimbólumot a negyedik Fibonacci-szám értékére. Készíts olyan egyparaméteres makrót, ami ellenőrzi, hogy a kapott paraméteréig jól működik-e a `Fibonacci` makró. Azt kell megvizsgálni, hogy a kezdőértékek jók-e, és hogy három szomszédos számra meghívva a makrót, a két nagyobbik kapott érték különbsége megegyezik-e a legkisebbel. Ha jól működik a makró, állítsuk be az `ok` szimbólumot egyre, ha pedig nem, akkor szüntessük meg az `ok` szimbólumot.

8. Írj olyan makrót, amely tetszőleges számú paramétert vár, és olyan kódot generál, ami megfordítja azokat a biteket `eax`-ben, amelyek sorszámát paraméterként kapta. A generált kód több utasításból is állhat, de minden egyéb regiszter értékét meg kell tartania.

Példa

Ha `eax` tartalma kezdetben `01110101011110101011110110101010`,  
akkor `fordit 1, 16, 3, 31` után `11110101011110111011110110100000` lesz `eax` új tartalma.

## A program fordításának menete

Amikor elkészült a program forráskódja, át kell alakítanunk futtatható állománnyá. Ennek első lépcsőjében hívjuk meg az assemblert, amely még csak egy közbülső formátumot, **tárgykódot** állít elő. A tárgykód már gépi kódot tartalmaz, de még nincsen készen: más tárgykódokban levő memóriacímekre való utalások lehetnek benne, és ezek csak a következő fázisban, az **összeszerkesztés** során válnak ismertté. A szerkesztőprogram (linker) feladata az, hogy mindezen információk segítségével összeállítsa a végső, futtatható fájlt.

A mi esetünkben az assembler neve **nasm**. A következő paramétereit használjuk a könyvben.

- meg kell adni az elkészítendő tárgykód-formátumot a **-f formátum** kapcsolóval: számunkra ez mindig **elf** lesz
- meg kell adni a **forrásfájlt**, amelynek a kiterjesztését általában **.asm**-nak választjuk
- ha szükségünk van rá, készíthetünk listafájlt a **-l listafájl-neve** kapcsolóval
- megadhatjuk a kimeneti fájl nevét a **-o fájlnev** kapcsolóval, alapértelmezés szerint ez a fájl neve **.asm** helyett **.o** kiterjesztéssel

Szerkesztőprogramnak a **gcc**-t használjuk<sup>16</sup>. Paramétereit közül csak a tárgykód-fájl nevét kell megadnunk. Ez létrehozza a futtatható fájlt, amely az **a.out** nevet kapja, ha felül nem bíráljuk a **-o fájlnev** kapcsolóval. Ha mindent jól csináltunk, már csak futtatni kell az elkészült fájlt.

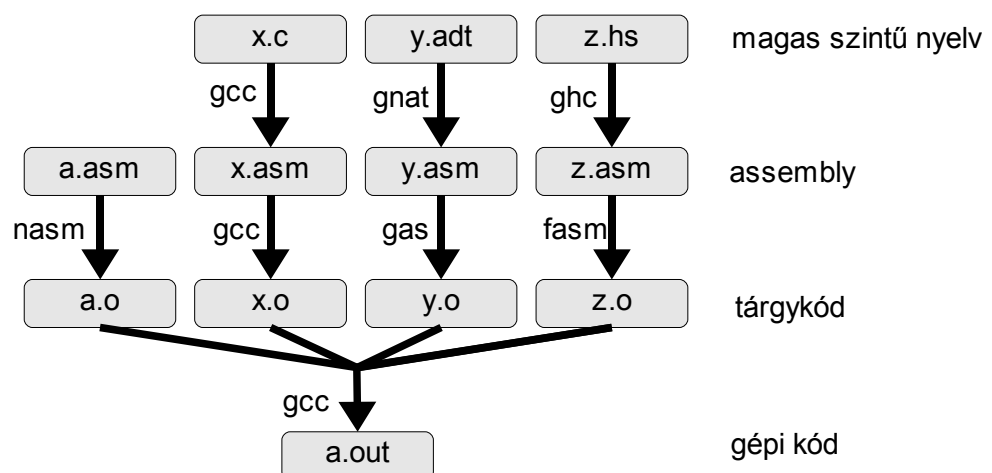
Összefoglalva: a következő parancsokra lesz szükségünk egy assembly fájl fordításához.

```
nasm -felf fájlnev.asm
gcc fájlnev.o -o fájlnev
./fájlnev
```

Amennyiben a program több tárgykódból áll, akkor a második lépés a következő alakú.

```
gcc fájl1.o fájl2.o fájl3.o -o futtatható_állomány_neve
```

Ha egy másik (akár más programnyelven megírt) fájlban akarjuk felhasználni valamelyik címkénket, akkor azt a **global címkénév** direktívával kell megjelölni.<sup>17</sup> Mivel a címkénév nem tartalmaz információt a címkén tárolt adat vagy kódrészlet típusáról, ezeket a programnyelvben az importálás során leírt deklarációval kell elérhetővé tenni. Fordítva, ha egy másik tárgykódból használunk fel egy címkét, akkor azt az **extern címkénév** direktívával kell jelezni.



<sup>16</sup> A gcc többek között C nyelvről assemblyre fordító fordítóprogramként is működik. Fordítsunk le egy tetszőleges, egyszerű C programot a **-S** kapcsolóval! Ez egy más szintaxisú assemblyre fordít, mint a könyvben tárgyalt **nasm**.

<sup>17</sup> Mivel a főprogramnak mindig láthatónak kell lennie kívülről, a főprogram kódjában szerepelnie kell a **global main** sornak.

## A gépi kód szerkezetéről

Egy utasítás gépi kódja (ami egy bitsorozat) több mezőre bontható. Általában vagy egy byte osztódik több mezőre, vagy egy mező tartalmaz több byte-ot. Az utasítás következő bitjeinek értelmezése (milyen mezőből áll) függhet az előző mezők értékétől.

Az utasítások gépi kódjának legfontosabb mezője az operációs kód (röviden opkód). Ez dönti el, hogy milyen utasításról van szó. Közeli kapcsolatban áll a mnemonikkal, de mégis különböznek: a mov jelenthet 8, 16 vagy 32 bites adatmozgatást, az opkód azonban ezt az információt is tartalmazza. Az opkód maga is tartalmazhat mezőket, például a duplaszavas adatot megnövelő `inc` egy byte-os gépi kódjának alsó három bitje azt kódolja, hogy melyik regisztert növeljük meg. A byte operandusú `inc` gépi kódja ezzel szemben két byte hosszú.

Az utasítások dekódolását tovább nehezíti, ha egyes mezők vegyesen értelmezettek. Egyes utasításoknál ugyanis előfordul, hogy egy mező lehetséges értékei közül egy adott bitminta speciális jelentést kap, a mező egyéb értékei pedig valamilyen más logika szerint szerveződnek. Például: az általános célú regisztereket el lehet kódolni három bit segítségével. Ilyen mezők szerepelnek majdnem minden utasítás gépi kódjában, mert a legtöbb utasításnak van regiszter operandusa, vagy tartalmazhat regiszteren keresztüli memóriahozzáférést. Azonban az `[ebp]` memóriacímzés nem kódolható el közvetlenül: legfeljebb egy regisztert használó memóriacímzés esetén az az érték, amely más esetekben `ebp` regisztert jelképezi, fenn van tartva a direkt (csak konstanssal való) memóriacímzés számára. Ezt a címzést más, ekvivalens módon kell elkódolni, `[ebp+0]` alakban.

```
0100 0000      inc  eax
0100 0001      inc  ecx
0100 0010      inc  edx
0100 0011      inc  ebx
0100 0100      inc  esp
0100 0101      inc  ebp
0100 0110      inc  esi
0100 0111      inc  edi
```

Az utasítás most csak az opkódból áll. Az alsó három bit jelöli ki, hogy melyik regiszterről van szó.

```
1111 1111 0000 0000      inc  dword [eax]
1111 1111 0000 0110      inc  dword [esi]
```

A második byte alsó három bitje a fentihez hasonló szerepet tölt be.

```
1111 1111 0000 0101
0111 1000 0101 0110 0011 0100 0001 0010      inc  dword [0x12345678]
```

A kivételes eset az, ami az előbb `ebp` regisztert kódolta. Memóriahozzáférés esetén az 101 bitsorozat a direkt címzés számára van fenntartva. Ekkor az utasítás kódjához tartozik még egy négy byte hosszú mező is, ahol maga a konstans jelenik meg.

```
1111 1111 0100 0101  0000 0000      inc  dword [ebp]
1111 1111 0100 0110  1010 1011      inc  dword [esi+0xAB]
```

Az assembler mégis le tudja fordítani az utasítást, ha csak `ebp` regiszterrel címzünk. Úgy tekint, mintha egy nulla konstanssal eltöltük volna a címet. A második byte 6. bitje jelzi azt, hogy a címzésben konstans is szerepel. Maga a konstans egy egy byte-os mezőn kap helyet. Ebben a konstrukcióban `ebp` ismét a többi regiszterhez hasonlóan van kódolva.



## CISC-elvű architektúrák

A korai számítógépek számítási kapacitása alacsony volt, lassú a memória-hozzáférésük és kevés a memóriájuk, ezért tömör és hatékony kódot kellett rájuk készíteni. Ennek támogatására az utasításkészleteket úgy tervezték, hogy az egyes utasítások összetett számításokat legyenek képesek végrehajtani, amelyek gyakran megvalósíthatóak voltak más utasításokból alkotott rövid kódrészletekkel is. Innen ered a megközelítés neve: Complex Instruction Set Computer, röviden **CISC**. Egy példa erre az IBM 360-as nagygép, amelynek egyik utasításával egy bináris fa adatszerkezetet lehetett kiegyensúlyozni.

A kódot az is tömörebbé tette, hogy bonyolult címzések segítségével egy utasítással lehetett adatelemeket lehetett egy lépésben elérni.

Az x86 architektúrán, ha nem használhatnánk az összetett címzéseket, a következő kifejezést sokkal hosszabb kóddal tudnánk csak megvalósítani.

Példa

```
mov eax, [ecx+4*esi+28]          mov eax, ecx
                                  mov ebx, esi
                                  shl ebx, 2
                                  add eax, ebx
                                  add eax, 28
                                  mov eax, [eax]
```

Természetesen a CISC architektúrájú gépek által végrehajtott gépi kód összetettsége is tükrözi a fentieket. Az egyszerű utasítások gépi kódú alakjában sokkal kevesebb információnak kell megjelennie, mint az összetettekében. Az architektúrák tervezőinek logikus döntése alapján ezért a gépi kódjuk is rövidebb. Így viszont a gépi kód végrehajtás előtti dekódolása bonyolultabbá válik, hiszen még az az információ sem áll rendelkezésünkre előzetesen, hogy hány byte alkotja a következő utasítást. Ebből következően a processzorok belső szerkezete is bonyolultabb, mivel ott találhatóak az utasítások dekódolását végző áramkörök is. Az x86 architektúrán 1 és 15 byte közötti hosszúságú utasítások fordulhatnak elő.

## RISC architektúrák

Az 1980-as évekre nyilvánvalóvá vált, hogy a programok túlnyomó többsége fordítóprogram segítségével készül. Kiderült, hogy a fordítóprogramok kódgenerálás során általában nem használják ki a CISC architektúrák teljes kínálatát. A hardver sebessége és tárolókapacitása is rohamosan növekedett.

A **RISC** (Reduced Instruction Set Computer) elvű architektúrákat az új hardveres kritériumoknak megfelelően tervezték. Az utasítások többsége gyorsan – tipikusan egy órajel alatt – hajtható végre. Az utasítások egyszerű szerkezetűek, jellemzően minden utasítás gépi kódja azonos hosszúságú. Az utasítások szerkezete egyszerű, kevés mezőt tartalmaznak, az utánuk következő mezők szerkezetét lehetőleg nem módosítják, ebből következően könnyen dekódolhatóak. Az egyszerű szerkezet csak egyszerű címzéseket tesz lehetővé.

Az x86 architektúra modern gépeinek processzora RISC elveken alapuló magra épül, amely értelmezi és végrehajtja azt a CISC kódot, amely a külvilág számára látható módon hajtódik végre a processzoron. Ezt a kétszintű felépítést mikroarchitektúrának nevezzük.

## Listafájl

Az assembler által generált kódot a **listafájlban** tudjuk megtekinteni. Ezt a nasm **-l listafájl** kapcsolójával tudjuk létrehozni. A listafájl a tárgykód speciális nézete, amely a következőket tartalmazza.

- minden sor elején a sorszámát,
- a kódsor eltolását az utoljára megkezdett szegmens kezdetétől,
- a kódsorból generált byte-okat,
- a forrásprogramszöveget eredeti alakjában, soronként,
- a makrókifejtések szintjét.

A listafájlban megjelenő számok hexadecimális számrendszerben vannak kódolva (a sorszámot kivéve). A listafájlból nem tudjuk közvetlenül kiolvasni az utasítások mezőinek szerkezetét, mivel csak a generált byte-ok hexadecimális alakja jelenik meg benne.

	<u>sorszám</u>	<u>cím a szegmens kezdetétől</u>	<u>lefordított kód (byte-ok)</u>	<u>az eredeti assembly kód változtatás nélkül</u>
Példa	1			
	2			section .text
	3			global main
	4			
	5			main
	6	00000000	B804000000	mov eax, 4
	7	00000005	BB01000000	mov ebx, 1
	8	0000000A	B9[00000000]	mov ecx, szoveg
	9	0000000F	BA04000000	mov edx, 4
	10	00000014	CD80	int 0x80
	11			
	12	00000016	B801000000	mov eax, 1
	13	0000001B	BB00000000	mov ebx, 0
	14	00000020	CD80	int 0x80
	15			
	16			section .data
	17			
	18	00000000	48656C6C6F20	szoveg db "Hello "
	19	00000006	56696C61670A	db "Vilag", 0xA

## A program betöltése

A listafájlban és a tárgykódban nem a kód végleges alakja szerepel. Amikor a kódban egy memóiahivatkozás található, a fordítóprogram több okból sem tudja a kódot végső alakban generálni.

Egyrészt előfordulhat, hogy a memóiahivatkozás egy másik tárgykódban szereplő címkére vonatkozik, amelyet az extern kulcsszóval jelöltünk meg. A címke pontos címét addig nem tudjuk eldönteni, amíg a másik tárgykódot nem generáltuk. Ezért olyan forrásfájlból, amely külső hivatkozásokat tartalmaz, nem lehet listafájlt készíteni.

Másrészt még akkor is, ha minden címkénk ismert, a generált kód nem a végleges címeket tartalmazza. A program betöltésekor a kód- és adatszegmenseket a futtató környezet beolvassa és elhelyezi a memóriában. A szegmensek általában nem a 0 kezdőcímmel kerülnek, ezért az összes rájuk vonatkozó memóiahivatkozáshoz hozzá kell adni a szegmens kezdőcímét. Ezért a kódszegmensben található mindegyik, az adatszegmensben levő adatokat elérő memóiahivatkozásból generált kódot módosítani kell. Ezt **relokációnak** nevezzük.

## Hibakeresés a programban

A programok elkészítése során általában elsőre nem sikerül tökéletes munkát végezni. Előfordulhatnak egyszerűbb hibák, például lemarad egy záró zárójel; ezeket a fordítóprogram képes jelezni még azelőtt, hogy futtatható állományt állítana elő a kódból. Nagyobb fejtörést okoz azonban az, ha a program lefordul, elindul, viszont a végrehajtás során mást csinál, mint amit a programozó elvár tőle. Ekkor meg kell keresni a hiba forrását, majd javítani kell.

A hiba megkeresésére egy kézenfekvő módszer, ha a program egyes pontjain jól látható módon szövegeket íratunk ki, amelyek jelzik számunkra, hogy a futás elérte az adott kódrészt. Ez a megközelítés azonban nehézkes, és kevés információt ad a hiba felmerülése időpontjában a regiszterek és a memória tartalmáról. Erre a célra külön **hibakereső** szoftvereket, angolul **debugger** programokat fejlesztettek ki. A továbbiakban a **gdb** hibakereső néhány funkcióját tekintjük át. Ha grafikus hibakeresőre van szükségünk, a **ddd** programot használhatjuk.

Tegyük fel, hogy a következő programban szeretnénk hibát keresni. A program eredetileg az első nyolc természetes számot írta ki egy-egy sorban, de elrontottuk.

Fordítsuk le a programot a szokásos módon, majd indítsuk el a hibakeresőt a

```
gdb
```

paranccsal. Ekkor egy megkapjuk a hibakereső parancssorát. Minden parancs az egyértelműségig rövidíthető, ezt aláhúzással jelöljük a továbbiakban. Segítségét a help paranccsal tudunk kérni.

```
section .text
global main

main
    mov ecx, 0

.ciklus
    cmp ecx, 8
    je .vege

    push ecx
    call kiir
    add esp, 4

    dec ecx
    jmp .ciklus

.vege
    mov eax, 1
    xor ebx, ebx
    int 0x80

kiir
    push ebp
    mov ebp, esp

    push ecx

    mov eax, 4
    mov ebx, 1
    mov ecx, adat
    mov edx, 2
    int 0x80

    pop ecx

    mov esp, ebp
    pop ebp
    ret

section .data
adat    db    "0"
        db    0xA
```

Először is töltsük be a programot.	<code>file kiir0</code>
Ahhoz, hogy a nasm-hoz hasonló formátumban írja ki a hibakereső az adatokat, adjuk ki a következő parancsot.	<code>set disassembly-flavor intel</code>
Először is nézzük meg a betöltött programot. Ehhez a következő két parancsot adjuk ki.	<code>disassemble main</code> <code>disassemble kiir</code>
Az első parancs a main címkétől a kiir címkéig írja ki az utasításokat, a második pedig a megadott címkét (jelen esetben a kiir-t) körülvevő függvényt írja ki, ami most pont a kiir függvény lesz. Pontosán ugyanezt a kimenetet kapnánk például a <code>disas kiir+3</code> parancs eredményeül. Ha a <code>main.ciklus</code> -t szeretnénk kiírni, egyszeres idézőjelek közé kell tennünk. Kényelmesen használható a <code>tab</code> gombbal az automatikus kiegészítés.	<code>disassemble kiir+3</code> <code>disassemble 'main.ciklus'</code>
Definiáljunk egy megszakítási pontot a program kezdetén. Lokális címkére így lehet megszakítási pontot definiálni: <code>break *'main.vege'</code>	<code>break main</code>
Ezután indítsuk el a programot, amely rögtön fenn is akad a megszakítási ponton. A programnak átadandó paramétereket a parancs után kell leírni.	<code>run</code>
Hajtsuk végre lépésenként a programot. Ehhez a <code>ni</code> (következő utasítás, next instruction) parancsot alkalmazzuk. Csak egyszer szükséges begépelni, mivel üres parancssorban egy enter leütésével meg lehet ismételni az előző kiadott parancsot. Az <code>ni</code> -nek megadható, hány utasítást futtasson le egyszerre.	<code>ni</code> <code>ni 50</code>
A sok kiadott <code>ni</code> hatására látszik, hogy a programunk ciklizálni kezd: ugyanazok a címek tűnnek fel sorban. Tegyük fel, hogy az a gyanúnk támadt, hogy a <code>kiir</code> függvényben van a gond; ekkor a <code>stepi</code> segítségével úgy hajtjuk végre a programot lépésenként, hogy a meghívott alprogramokba belépünk.	<code>stepi</code>
Sajnos, ez sem segített, de újabb ötletünk támadt: mivel a <code>kiir</code> -t megnézve az jónak tűnik, megvizsgáljuk, megfelelő paraméterrel hívtuk-e meg. A hibakeresőben felvesszük az <code>ecx</code> regisztert figyelte értéknek. A hibakereső sajátossága, hogy a regiszterek neveit a dollár prefixszel kell ellátni.	<code>display \$ecx</code>
A hibakövetést tovább könnyítendő, beállítjuk, hogy a hibakereső minden lépés után jelezze ki a következő utasítást. A <code>/i</code> formátumkód azt jelzi, hogy utasítást íratunk ki, az <code>eip</code> pedig az a számunkra el nem érhető regiszter, amely az aktuális utasítás címét tartalmazza.	<code>display/i \$eip</code>
Innen, folytatva a léptetést, megállapítjuk, hogy a paraméter megfelelő, azonban a kiírás mégsem jó. Kilépünk a hibakeresőből, és <code>ecx</code> elvermelése után betoldjuk a következő sort.  <code>mov [adat], cl</code>	<code>quit</code>
A hibakeresés kezdete után látszik, hogy most már valóban különböző karakterek jelennek meg, de ezek nem a 0, 1, ... számok. Kapcsoljuk folytonos megjelenítésre az adat címkét, hiszen ennek a tartalmát írjuk ki. A <code>/c</code> kapcsoló azt jelzi, hogy karakteresen jelenítünk meg.	<code>display/c adat</code>
Egy idő után észrevevesszük, hogy a karakterek nem növekednek, hanem csökkennek, ezért át kell írunk a <code>main.ciklus</code> -ban a <code>dec ecx</code> utasítást <code>inc-re</code> . Ezután még az is kiderül, hogy elfelejtettük, hogy a karakterkódolás szerint a számjegyek nem a nulla pozícióról indulnak, hanem <code>30h</code> -ről, ezért a <code>kiir</code> függvényben <code>ecx</code> vermelése után még meg kell növelnünk <code>cl</code> -t a következő utasítással.  <code>add cl, 30h</code>  A program készen van.	

## Gyakran előforduló hibák

Az alábbiakban következnek néhány, zárthelyi dolgozatokban gyakran előforduló hiba leírása. Ha adható rövid példa, dőlt betűvel szedve egy-egy konkrét előfordulást mutatunk be, majd pedig egy konkrét javítást. Csak az assemblyhez közvetlenül kapcsolódó hibák szerepelnek, így pl. a kiolvashatatlan kézírás, a feladat félreértése, csak speciális esetekre megoldott feladat stb. nem.

1. Nem elsősorban assembly vonatkozású, de a **kiolvashatatlan kézírás** következtében sokszor nem egyértelmű, hogy melyik regiszterről van szó, mennyi a leírt konstans értéke stb.

### 2. Szintaktikai hibák.

#### a) Számábrázolás.

i. **Betűvel kezdődő hexadecimális szám elejéről lemarad a nulla.**

ii. **Hexadecimális szám végéről lemarad a »h« betű** (és az elején sincsen 0x).

b) Utasítások operandusai közül lemarad a **vessző**.

### 3. Az adatok mérete nem megfelelő.

#### a) A regiszterek méretének eltévesztése.

*Az al regiszter 4 bitesként kezelése.*

b) **Az utasítás operandusainak mérete nem egyezik meg** (kivéve movsx és movzx, ahol éppen eltérőnek kell lenniük).

`mov            eax, bl                            →    mov eax, ebx vagy mov al, bl`

c) **Nem megfelelő adathosszat** használunk adatok definiálásához.

*Duplaszavas adatot szeretnénk definiálni, de csak szavas adatterületet foglalunk le.*

`adat            resw 1                            →    adat resd 1`

d) A **memóriához való hozzáférés** nem a megfelelő adathosszúságban történik. Ritka esetekben szándékosan tehetünk ilyet, de általában ez hiba.

*Olyan címre, amelyen duplaszavas méretű adat van eltárolva, byte hosszan írunk.*

`adat    dd                            0`  
`mov    [adat],            al →        mov [adat], eax`

*Természetesen az is helytelen, ha ugyanarról a címről például szó hosszan próbálunk olvasni.*

`mov    ax,                    [adat] →        mov eax, [adat]`

### 4. Adatábrázolás.

a) **Szám ábrázolásánál az (előjel nélkül ábrázolt) szám konstans és a számjegy karakter összekeverése.**

*Pl. szövegesen leírt szám (számjegyeket tartalmazó string) konverziójánál minden karaktert előbb előjel nélkül ábrázolt számmá kell alakítani. Ezt legegyszerűbben a 0 számjegy kódjának levonásával lehet elérni. Ez a módszer kihasználja, hogy a számjegyek karakterkódjai sorrendben találhatók.*

### 5. Szegmensek.

a) A **szegmenshatárok jelölése lemarad.**

b) **Kódszegmensben adat, adatszegmensben kód szerepeltetése.** Az előbbi ekvivalens azzal, mintha gépi kódot íránk kézzel.

- c) Az **inicializált és inicializálatlan adatszegment**s összekeverése (.data és .bss), a másikba való adatok szerepeltetése.

## 6. Utasítások operandusai.

- a) **Adathossz explicit megadása hiányzik**, amikor ez nem derül ki máshonnan, pl. kétoperandusú utasítás esetén a másik operandusból.

```
mov      [eax], 2          →  mov dword [eax], 2 vagy mov word [eax], 2  
                                vagy mov byte [eax], 2
```

- b) **Nem megfelelő számú vagy fajtájú operandus** megadása utasításhoz.

*Az alábbiakban elkerültük a 3. fajtájú hibát, mivel megadtuk az adat hosszát (ami máshonnan nem derülne ki), azonban a div-nek nincsen konstans operandust fogadó változata. A javításhoz felhasználjuk bl regisztert.*

```
div      byte 2           →  mov bl, 2  
                                div bl
```

- c) **Túlságosan összetett számítási műveletek egy utasításon belül.** Ezeket csak több utasítással lehet kiváltani, amihez szükség lehet további regiszterek felhasználására. Ebben az esetben a regiszter eredeti tartalmát a verembe menthetjük, vagy lefoglalhatunk egy tárterületet erre a célra.

*A címke címke tartalma plusz egyet szeretnénk eax-be tölteni.*

```
mov      eax, [címke] + 1  →  mov eax, [címke]  
                                inc eax
```

*Az ebx és ecx regiszterek összegét szeretnénk eax-be tölteni.*

```
mov      eax, ebx + ecx    →  mov eax, ebx  
                                add eax, ecx
```

*Egy komplex indexelési műveletet szeretnénk végrehajtani. Az utasítás az eredeti formájában nem érvényes, a helyettesítő kódban az egyszerűség kedvéért felhasználjuk eax és edx regisztereket. Megoldható a feladat úgy is, hogy átmeneti tárolókat veszünk fel az inicializálatlan adatszegmensbe.*

```
cmp [esi-(ecx-4)*4], [esi-(ebx-1)*4] → mov eax, ebx ; eax = ebx  
                                         dec eax      ; eax = ebx - 1  
                                         shr eax, 2   ; eax = (ebx - 1) * 4  
                                         sub eax, esi ; eax = (ebx - 1) * 4 - esi  
                                         neg eax     ; eax = esi - (ebx - 1) * 4  
                                         mov eax, [eax] ; eax = [esi - (ebx - 1) * 4]  
                                         mov edx, ecx ; edx = ecx  
                                         sub edx, 4   ; edx = edx - 4  
                                         shr edx, 2   ; edx = (edx - 4) * 4  
                                         sub edx, esi ; edx = (edx - 4) * 4 - esi  
                                         neg edx     ; edx = esi - (edx - 4) * 4  
                                         mov edx, [eax] ; edx = [esi - (edx - 4) * 4]  
                                         cmp edx, eax ; edx = [esi - (edx - 4) * 4]
```

**d) Adat és cím fogalmának összekeverése.**

- i. Adat címe helyett csak az adat szerepel: hiányzik a szögletes zárójel.

*A címke címke tartalmát szeretnénk eax-be tölteni.*

`mov eax, címke` → `mov eax, [címke]`

- ii. Adat helyett adat címe szerepel: feleslegesen megjelenő szögletes zárójel. Előfordul, hogy ez a hiba párosul 3.c)-vel.

- e) A cél és a forrás összekeverése (az operandusok fordított sorrendben vannak leírva). Különösen gyakran fordul elő, hogy konstans vagy címke az első operandus, amelyeknek futási időben nem lehet értéket adni.

**f) Indexelési hibák.**

- i. **Indexregiszter negatív előjellel.** Ezt 3.c)-hez hasonlóan csak hosszabb kódrészlettel lehet kiváltani.  
ii. **Adat hosszának figyelmen kívül hagyása** indexeléskor.

Duplaszó hosszú adatok tömbjének indexelésekor csak byte-onkénti indexelés.

`mov [bazis+ecx], eax` → `mov [bazis+4*ecx], eax`

- iii. A **little endian tárolási mód** figyelmen kívül hagyása.

**7. Programkonstrukciók.**

- a) **Elágazás ágaiban lemarad a kiugrás az elágazás végére.** Ez többirányú elágazásra is vonatkozik.

*Az eax regisztert akarjuk nullára vagy egyre állítani annak függvényében, hogy az adat címkén található duplaszavas adat értéke egy-e. A kiugrás hiánya miatt az igaz ág végrehajtása eax nullára állítása után ráfut a másik ág kódjára is, ami elállítja eax-ot egyre.*

<code>cmp dword [adat], 1</code>		<code>cmp dword [adat], 1</code>
<code>jne .else_ag</code>		<code>jne .else_ag</code>
<code>mov eax, 0</code>		<code>mov eax, 0</code>
	→	<b><code>jmp .elagazas_vege</code></b>
<code>.else_ag</code>		
<code>mov eax, 1</code>		<code>.else_ag</code>
		<code>mov eax, 1</code>
		 <code>.elagazas_vege</code>

**b) Ciklusok.**

- i. **A ciklus inicializáló lépése bekerül a ciklusmagba.**

- ii. **A ciklus inicializáló lépése lemarad.**

iii. **C stílusú for ciklusok.**

- 1) A **feltételvizsgálatnak** a ciklus elején (közvetlenül a ciklus címkéje után) kell megtörténnie.  
2) A **ciklusváltó léptetésének** a ciklus végén (közvetlenül a ciklusmag elejére való visszaugrás előtt) kell megtörténnie.

- c) **Elérhetetlen programkód.** Mivel sosem fut le, felesleges létrehozni.

- d) **Alprogramok.** Az alprogramra lokális, a futási idejű veremben elhelyezkedő változó helyett globális változó létrehozása az adatszemensben. Ez nem feltétlenül hiba...

**8. Egyes utasításokhoz köthető hibák.**

- a) **A not és a neg utasítás keverése.** A neg számítása során lemarad a +1.



- b) **Szorzás és osztás során a forrás- és célregiszterek eltévesztése.** Helyes alkalmazáshoz lásd a mul, imul, div, idiv leírását és a könyv végén a táblázatot.

## 9. Megszakítások, alprogramok.

- A megszakítás meghívása lemarad. A paraméterek beállításán kívül szükség van az „int 80h” sorra is.
- Alprogram címkéje lemarad.** Nélküle nem lehet (kényelmesen) meghívni a belépési pontot a kód más részeiről.
- Alprogram címkéje nem lehet lokális.** Az *alprogramra* lokális címkéket érdemes létrehozni.
- Az alprogram bevezető és/vagy kilépő kódrészlete lemarad.** Kellő tapasztalattal a kódrészletek egyszerűsíthetők, de az alprogram végén legalább egy visszatérő „ret” utasításnak szerepelnie kell.
- Az alprogramok elején és végén a **használt regiszterek nincsenek elmentve és visszatöltve.**

## 10. Fájlok.

**Lemarad a fájlleíró eltárolása.** Ha megnyitás után nem tároljuk el a fájlleírót a memóriában (lokális vagy globális változóként), akkor a továbbiakban nem tudjuk elérni a fájlt.

## 11. Verem.

- Nem megfelelően rendezett a verem.** Általában: a push-ok és pop-ok nincsenek párban.
- A verembe tett és abból kivett adatok mérete nem egyezik.

*Az eax regiszter alsó szavát szeretnénk áttölteni bx-be, és ehhez a vermet próbáljuk felhasználni. Az eredeti kódrészlet ezt elvégzi, mivel a byte-sorrend fordul, azonban két byte-tal többet hagy a veremben, mint amennyi kezdetben benne volt. Ennek elkerülése érdekében a teljes ebx-be pop-olunk. Ha el akarjuk kerülni, hogy ebx felső felét is felülírjuk, akkor a pop után vissza is léptethetjük esp-t a kiinduló helyzetbe.*

push	eax	→	push eax	vagy	push eax
pop	bx		pop ebx		pop bx
					add esp, 2

- Nemsztenderd hozzáférés a veremhez.** A verem push-on, pop-on és alprogramokon belül ebp-n való kezelésén kívül más módszerrel nagyon oda kell figyelni ahhoz, hogy a verem szerkezete el ne romoljon.

## 12. Makrók.

- A futási és fordítási idő fogalmának összekeverése.** A makrók fordítási időben, az előfordítási fázis során alakítják át a forráskód szövegét, a regiszterek csak futási időben léteznek, az utasítások futási időben hatnak.

mov eax, 1	→	mov eax, 1
mov ebx, 2		mov ebx, 2
%assign ecx eax + ebx		<b>mov ecx, eax</b>
		<b>add ecx, ebx</b>

- Makróra lokális címkék alkalmazásának hiánya.** Ha globális címkéket alkalmazunk makrókban, akkor nem lehet őket több helyen kifejteni, mert a fordító hibát ad többször definiált név miatt. Makróra lokális címke esetén minden kifejtéskor új, egyedi név keletkezik.

%macro faktorialis 1	→	%macro faktorialis 1
%assign fakt 1		%assign fakt 1
%assign i 1		%assign %i 1
%rep %0 - 1		%rep %0 - 1
%assign fakt i * fakt		%assign fakt %i * fakt
%endrep		%endrep
%endm		%endm

## A feladatok megoldásai

### Értékek ábrázolása

1. Mekkora értékek tárolhatóak egy bitvektorban? Add meg általánosan is!

bitek száma	számrendszer	előjel nélkül		előjelesen	
		legkisebb érték	legnagyobb érték	legkisebb érték	legnagyobb érték
8	hexadecimális	00	FF	80	7F
	decimális	0	255	-128	+127
16	hexadecimális	00 00	FF FF	80 00	7F FF FF FF
	decimális	0	65 535	-32 768	+32 767
32	hexadecimális	00 00 00 00	FF FF FF FF	80 00 00 00	7F FF FF FF
	decimális	0	4 294 967 295	-2 147 483 648	+2 147 483 647
4 · n	hexadecimális	00 00 ... 00 n számjegy	FF ... FF n számjegy	80 00 ... 00 n számjegy	7F FF ... FF n számjegy
	decimális	0	16 <sup>n</sup> -1	-2 <sup>4n-1</sup>	+2 <sup>4n-1</sup> -1

2. Töltsd ki az üres cellákat a táblázatban!

bitek száma	bináris	decimális		hexadecimális
		előjel nélküli	előjeles	
8	1100 1111	207	-49	CF
8	0110 1011	107	+107	6B
8	1110 1000	232	-24	E8
16	1001 0110 0001 1110	38 430	-27 106	96 1D
16	0010 0000 1010 0101	8357	+8357	20 A5
16	1101 0010 0011 0111	53 815	-11 721	D2 37

3. Végezd el az alábbi műveleteket!

$$\begin{array}{r} 1001\ 1101 \\ \text{and}\ 1100\ 0111 \\ \hline 1000\ 0101 \end{array}$$

$$\begin{array}{r} 0110\ 1110 \\ \text{or}\ 1000\ 0000 \\ \hline 1110\ 1110 \end{array}$$

$$\begin{array}{r} 0101\ 1010 \\ \text{xor}\ 1111\ 0001 \\ \hline 1010\ 1011 \end{array}$$

$$0101\ 0110\ 1101\ 1101_2 + 1C\ E3_{16} = 0111\ 0011\ 1100\ 0000_2 = 29\ 632_{10} = 73\ C0_{16}$$

$$16\ 537_{10} - 0000\ 0011\ 1010\ 1100_2 = 0011\ 1100\ 1110\ 1101_2 = 15\ 597_{10} = 3C\ ED_{16}$$

$$54\ 32_{16}\ \text{or}\ 38\ 529_{10} = 0101\ 0110\ 1011\ 0011_2 = 22\ 195_{10} = 56\ B3_{16}$$

4. Írd le a saját nevedet ASCII kódolással, ékezetek nélkül!

K	i	t	l	e	i	szóköz	R	o	b	e	r	t	sorvége
4B	69	74	6C	65	69	20	52	6F	62	65	72	74	0A

5. Írd le a „Hello Vilag” szöveget ASCII kódolással!

H	e	l	l	o	szóköz	V	i	l	a	g	sorvége
48	65	6C	6C	6F	20	56	69	6C	61	67	0A

## Utasítások

1. Add meg a regiszterek tartalmát minden utasítás után! A feladatot a jelölt pontokon is el lehet kezdeni.

	reg. vagy mem.	bináris (megfelelő bithosszon)	decimális		hexadecimális	
			előjel nélküli	előjeles		
a.	mov ebx, 1023	ebx	0000 0000 0000 0000 0000 0011 1111 1111	<b>1023</b>	<b>+1023</b>	00 00 02 FF
	mov al, -100	al	1001 1100	<b>156</b>	<b>-100</b>	9C
	movsx eax, bl	eax	1111 1111 1111 1111 1111 1111 1111 1111	<b>65 535</b>	<b>-1</b>	FF FF FF FF
b.	mov eax, 1C34F6E2h	ax	1111 0110 1110 0010	<b>63 202</b>	<b>-2334</b>	F6E2
		ah	1111 0110	<b>246</b>	<b>-10</b>	F6
		al	1110 0010	<b>226</b>	<b>-30</b>	E2
c.	mov byte [0FAh], 01Ch	[0xFA]	0001 1100	<b>28</b>	<b>+28</b>	1C
	mov byte [0FBh], 0x22	[0xFB]	0010 0010	<b>34</b>	<b>+34</b>	22
	mov ax, [0FAh]	ax	0010 0010 0001 1100	<b>8732</b>	<b>+8732</b>	221C
d.	mov eax, 89ABCDEFh	ax	1000 1001 1010 1011 1100 1101 1110 1111	<b>2 309 737 967</b>	<b>-1 985 229 329</b>	89 AB CD EF
	mov ah, al	ah	1110 1111	<b>239</b>	<b>-17</b>	EF
	mov al, 42	al	0010 1010	<b>42</b>	<b>+42</b>	2A
	mov bl, al	bl	0010 1010	<b>42</b>	<b>+42</b>	2A
	mov bh, al	bx	0010 1010 0010 1010	<b>10794</b>	<b>+10 794</b>	2A 2A
e.	mov ax, 0FF42h	ax	1111 1111 0100 0010	<b>65 346</b>	<b>-190</b>	FF 42
	mov bx, 0DCBAh	bx	1101 1100 1011 1010	<b>56 506</b>	<b>-9 030</b>	DC BA
	sub bh, al	bh	1001 1010	<b>154</b>	<b>-102</b>	9A
	xor ax, bx	ax	0110 0101 1111 1000	<b>26 104</b>	<b>26 104</b>	65 F8
bx		1001 1010 1011 1010	<b>39 610</b>	<b>-25 926</b>	9A BA	
f.	mov al, 01011010b	al	0101 1010	<b>90</b>	<b>+90</b>	5A
	or ah, 10101010b	ah	1111 1010	<b>250</b>	<b>-6</b>	FA
g.	mov al, 120	al	0111 1000	<b>120</b>	<b>+120</b>	78
	not al	al	1000 0111	<b>135</b>	<b>-121</b>	87
	neg al	al	0111 1001	<b>121</b>	<b>121</b>	79
h.	mov bh, 254	bh	1111 1110	<b>254</b>	<b>-2</b>	FE
	inc bh	bh	1111 1111	<b>255</b>	<b>-1</b>	FF
	add bh, 1	bh	0000 0000	<b>0</b>	<b>0</b>	00
i.	mov cx, 0x01F0b	cx	0000 0001 1111 0000	<b>496</b>	<b>+496</b>	01F0
	mov eax, -128	eax	1111 1111 1111 1111 1111 1111 1000 0000	<b>4 294 967 168</b>	<b>-128</b>	FF FF FF F0
	mul cx	dx : ax	0000 0000 0000 0000:1111 1000 0000 0000	<b>0 : 63 488</b>	<b>0 : -2048</b>	00 00:F8 00
	imul cl	ax	0	<b>0</b>	<b>0</b>	0

2. a. Az x, y és z címen byte-os adatok vannak.  
Számítsd ki z-be  
(not x | y) & x értékét!

```
mov al, [x]
not al
or al, [y]
and al, [x]
```

2. b. Számítsd ki eax-be  
(eax - 1) \* 2 értékét!

```
dec eax
shr eax, 1
```

3. Adj meg minél több utasítást, amely  
a. nullára állítja `eax` értékét!

```
mov  eax, 0          xor  eax, eax          shr  eax, 32
xor  eax, eax        sub  eax, eax          shl  eax, 32
and  eax, 0          sal  eax, 32
```

- b. nem változtatja meg a regiszterek értékét (legfeljebb a jelzőbitek regiszterének kivételével)!

```
nop                  bt    eax, 0          ror  eax, 0
mov  eax, eax        and  eax, 0xFFFFFFFF rol  eax, 0
xchg al, al         and  eax, eax        cmp  eax, ebx
add  eax, 0          or   eax, 0          jmp  .kovetkezo
sub  eax, 0          or   eax, eax        .kovetkezo
xor  eax, 0          xor  eax, 0
```

- c. `eax` 2. bitjét egyesre állítja, a többbit pedig változatlanul hagyja!

```
or   eax, 100b      bts  eax, 2
```

- d. az ellentétére állítja `ebx` legalsó (0.) és legfelső (31.) bitjét!

```
xor  ebx, 0x8001
```

- e. `al` tartalmát egy előre adott értékről egy másik adott értékre állítja! (Például 229-ről 33-ra.)

```
mov  al, 33         add  al, 33-229      sub  al, 229-33
```

**A feladat megoldható még az `xor al, eltérés` utasítással, ahol az *eltérést* a két értékből az `xor` művelettel kaphatjuk meg, jelen esetben ez *1100 0100b*. Speciális esetekben további utasításokat is lehet alkalmazni, például bitforgatást, regiszter nullázását stb.**

4. Az `x` és `y` két címke, melyeken 32 bites adatokat tárolunk.

- a. Valósítsd meg az `x := y` értékadást! Közvetlenül egyik címről a másikra másoló utasítás nincsen.

```
mov  eax, [x]          vagy          push dword [x]
mov  [y], eax          pop  dword [y]
```

- b. Cseréld meg `x` és `y` tartalmát!

```
mov  eax, [x]          vagy          push dword [x]
mov  edx, [y]          push dword [y]
mov  [x], edx          pop  dword [x]
mov  [y], eax          pop  dword [y]
```

5. Milyen byte-okat tartalmaz sorban az adatszegmens?

```
section .data
db 55h
db 55h, 56h
db 'a', 0Ah
db "hello"
dw 1234h
dd 89ABCDEFh
```

**A byte-ok sorban, hexadecimális alakban:**

**55 55 56 61 0A 68 65 6C 6C 6F 34 12 00 EF CD AB 89.**

6. Valósítsd meg az ismertetett utasításokkal az *adc eax, ebx* utasítást! Az *adc* az első operandushoz hozzáadja a másodikat (mint az *add*), és még ehhez az átviteli bitet (azaz hozzáad még egyet, ha az átviteli bit be van állítva, különben az eredmény a két operandus összege).

```
jnc .nincs_carry
inc eax
add eax, ebx
jmp .vege

.nincs_carry
add eax, ebx

.vege
```

A feladat rövidebben is megoldható. Ehhez felhasználjuk, hogy az egyik ág kódja posztfixe a másik ágának: a *carry* ág végén nem ugrunk el, hanem hagyjuk, hogy a vezérlés ráfusson a másik ág kódjára. Hasonló kód keletkezik, ha egy C programban a *switch* szerkezet egy ágát nem zárjuk le *break* utasítással.

```
jnc .nincs_carry
inc eax

.nincs_carry
add eax, ebx
```

7. Mennyi a kódrészletek végrehajtása után *eax* értéke?

<pre>.a xor ebx, ebx inc ebx jmp .vege  dec ebx mov eax, ebx  .vege</pre>	<pre>.b cmp eax, 0 je near .vege  mov eax, eax  .vege inc eax</pre>	<pre>.c cmp eax, eax je near .vege dec eax  jmp .c  .vege</pre>
---	---	---

- Ugyanannyi, mint a program elején, mert nem hajtunk végre olyan utasítást, amely megváltoztatná *eax* értékét. Az utolsó két utasítás elérhetetlen, mert előtte feltétel nélkül elugrunk a *.vege* címkére; az ugrás előtti két utasítás *ebx* értékét állítja egyre.
- A feltételvizsgálat szerint két eset lehetséges: vagy nulla volt kezdetben *eax* értéke, vagy sem. Amennyiben nulla volt, elugrunk a *.vege* címkére. Amennyiben nem nulla volt, folytatódik a végrehajtás, és a következő utasítás kinullázza *eax*-ot. Tehát a *.vege* címkére érve *eax* értéke nulla, bármelyik úton is érkezett ide a vezérlés. Innen még egyetlen utasítás van hátra, amely megnöveli *eax* értékét egyre.
- A kezdeti feltételvizsgálat akkor ugrik a *.vege* címkére, ha *eax* regiszter értéke megegyezik *eax* regiszterével. Ez mindig teljesül, ezért a következő két kódsor elérhetetlen, mert rögtön elugrunk. A regiszter értéke nem változik meg.

8. Írj olyan kódrészletet, amely ecx-be meghatározza eax és ebx maximumát! A számok előjel nélkül vannak ábrázolva.

```
    cmp     eax, ebx
    ja near .eax_nagyobb    ; előjel nélküli összehasonlítás

    mov     ecx, ebx        ; itt biztosan ebx a nagyobb (vagy egyenlőek)
    jmp     .vege          ; kiugrás az elágazásból

.eax_nagyobb
    mov     ecx, eax

.vege
```

9. Lineáris keresés: az adat címkétől kiindulva keresd meg az első olyan byte-ot, amely nullát tartalmaz, és ennek a címét add meg eax-ben!

```
    mov     eax, adat      ; az adat címke címét betöltjük eax-ba

.ciklus
    cmp     byte [eax], 0
    je near .vege         ; ha az eax címen található byte értéke nulla,
                          ; kiugrunk a ciklusból
    inc     eax           ; különben a következő byte-ra lépünk
    jmp     .ciklus       ; ... és folytatjuk a keresést

.vege
```

10. Valósítsd meg assembly nyelven az alábbi C++ programrészletet! Az a és b változók típusa int.

```
for ( int i = 0; i < 10; ++i ) a += b;
```

```
    mov     ecx, 0        ; i-t ecx-ben tároljuk

.ciklus
    cmp     ecx, 10
    jnl near .vege       ; előjeles vizsgálat, ha i >= 10, ciklus vége

    mov     eax, [a]
    add     [b], eax     ; a += b;

    dec     ecx          ; ++i
    jmp     .ciklus

.vege
```

11. Az n paraméter a memóriában található egy duplaszavas változóban. Számítsd ki az eax regiszterbe  
 a. az első n szám összegét!  
 b. n! értékét!

Az adatszégmensben tárolt n mindkét esetben a következőképpen néz ki.

```
section .data
n    dd    9
```

Amennyiben egy másik kódrészlet határozza meg n értékét számunkra, az inicializálatlan adatszégmensben is elhelyezhetjük.

```
section .bss
n    resd 1
```

```
a
mov eax, [n]
mov ebx, 0
mov ecx, 0
; ebx: részletösszeg
; ecx: számláló

.ciklus
add ebx, ecx
inc ecx

cmp ecx, eax
ja .vege
jmp .ciklus

.vege
```

```
b
mov eax, 1
mov ebx, 1

.ciklus
cmp ebx, [n]
jg .vege

mul ebx
inc ebx
jmp .ciklus
```

12. Adott egy duplaszavas értékeket tartalmazó láncolt lista címke. Add meg a hosszát!

```
mov ebx, fejelem
mov eax, 1

.ciklus
cmp dword [ebx], 0
je .vege

mov ebx, [ebx]
inc eax
jmp .ciklus

.vege
```

Ez a kódrészlet négygel tölti fel eax-et, ha az alábbi adatokra hívjuk meg.

```
elem2    dd    elem3
          dd    3
fejelem  dd    elem2
          dd    150
elem4    dd    0
          dd    365
elem3    dd    elem4
          dd    8163
```

13. Adott a szoveg címken egy szöveg, amelynek ismert a hossza. Fordítsd meg helyben!

```
mov esi, szoveg
mov edi, szoveg + hossz - 1 ; szoveg + hossz már a szöveg utáni pozíció

.ciklus
cmp esi, edi
jnb .vege

mov al, [esi]
mov ah, [edi]
mov [esi], ah
mov [edi], al

inc esi
dec edi

jmp .ciklus

.vege
```



14. Adott három időpont: egy-egy óra és perc, amelyek duplaszavasán vannak eltárolva a memóriában. Az első két időpont között részleges napfogyatkozás következik be. A napfogyatkozás mértéke a félidőig lineárisan növekszik, félidőben pontosan 50%, onnantól lineárisan csökken. Add meg, hogy a harmadik időpontban (amelyről feltételezhető, hogy az előző kettő között helyezkedik el) mekkora volt egész százalékra kerekítve a napfogyatkozás mértéke!

Tegyük fel, hogy az adatok a következőképpen vannak eltárolva.

```

section .data
kezdet
.ora    dd    10
.perc   dd    00
veg
.ora    dd    14
.perc   dd    00
idopont
.ora    dd    10
.perc   dd    30

```

```

mov ebx, 60

mov eax, [veg.ora]
sub eax, [kezdet.ora]
mul ebx
add eax, [veg.perc] ; eax = a teljes napfogyatkozás
sub eax, [kezdet.perc] ; időtartama
shr eax, 1

mov ecx, eax ; ecx = a teljes időtartam fele

mov eax, [idopont.ora]
sub eax, [kezdet.ora]
mul ebx
add eax, [idopont.perc]
sub eax, [kezdet.perc] ; eax = a kezdettől eltelt idő

mov ebx, ecx

cmp ebx, eax ; ugrás, ha a félidő előtt vagyunk
ja .tovabb

sub eax, ecx ; félidő után vagyunk
sub eax, ecx
neg eax ; eax = idő a napfogy. végéig

.tovabb
mov ebx, 50
mul ebx ; eax * 50 / (a teljes idő fele)
div ecx ; pontosan a kért eredményt adja

```

15. A teniszben az a játékos nyer meg egy ún. rövidített játékot, aki először ér el legalább 7 pontot úgy, hogy legalább 2 ponttal vezet (azaz ha az állás 7-6, akkor még nem dőlt el a rövidített játék, például a második játékos megszerezheti a következő három pontot, és akkor ő nyer 7-9-re). Egy tömbben adott, hogy melyik játékos nyerte meg a soron következő játékot. A tömb hossza előre nem ismert. Add meg, ki nyerte a rövidített játékot, és milyen arányban!

Tegyük fel, hogy a következőképpen adott a nyert játékok tömbje.

```
section .data
```

```
jatek db 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0
```

A következő kódrészlet kiszámítja al-be és ah-ba, melyik játékos hány játékot nyert. A program futása az .Anyert illetve a .Bnyert címkén folytatódik attól függően, melyik játékos nyerte a rövidített játékot.

```
xor eax, eax ; ah és al (a játékosok által nyert pontok) kinullázása
xor ebx, ebx ; bl (a játékosok által nyert pontok különbsége) kinullázása
mov esi, jatek ; esi mutatja az aktuális pozíciókat a tömbben
```

```
.ciklus
```

```
cmp bl, 2
jnge .Anemvezet ; annak ellenőrzése, vezet-e a játékos legalább 2 ponttal
```

```
cmp al, 7
jae .Anyert ; ha igen, akkor nyert, ha elért legalább 7 pontot
```

; itt következhetne egy jmp nincsnyertes utasítás, mert ha az egyik játékos vezet, akkor a másik biztosan nem, de az sem okoz problémát, ha a vezérlés ráfut a másik ellenőrzésre

```
.Anemvezet
```

```
cmp bl, -2
jnle .nincsnnyertes
```

```
cmp ah, 7
jae .Bnyert ; az előzőhöz hasonló ellenőrzés a másik játékosra
```

```
.nincsnnyertes
```

```
add al, [esi]
inc ah
sub ah, [esi] ; ah és al (a játékosok által nyert pontok) állítása
mov bh, [esi]
shl bh, 1
dec bh ; bh = +1 vagy -1 attól függően, melyik játékos nyerte a pontot
add bl, bh ; bl (a játékosok által nyert pontok különbsége) állítása
inc esi ; a mutató léptetése
jmp .ciklus
```

1. a. Írd ki a „Hello világ!” szöveget betűről betűre bővülve! Először csak egy „H” betűt írd ki és egy sorvégét, aztán a „He” szöveget és sorvégét stb.

```

section .text
global main

main
    push    ebp
    mov     ebp, esp
    sub     esp, 4          ; [ebp-4] : ciklusszámláló

    mov     [ebp-4], dword 0
    ; kezdetben a szöveg nulladik pozícióján állunk

.ciklus
    cmp     dword [ebp-4], 11
    je     near .vege

    mov     eax, [ebp-4]
    mov     dl, [szoveg + eax]
    mov     [kiir + eax], dl
    mov     [kiir + eax + 1], byte 0xA
    ; a következő karakter és egy sorvége

    mov     edx, eax          ; a kiírás hossza
    add     edx, 2           ; +1 az indexelés miatt
                                ; +1 a sorvége miatt

    mov     eax, 4
    mov     ebx, 1
    mov     ecx, kiir
    int     0x80

    inc     dword [ebp-4] ; a következő karakter
    jmp     .ciklus

.vege
    mov     eax, 1
    xor     ebx, ebx
    int     0x80

section .data
szoveg    db        "Hello Vilag"

section .bss
kiir      resb 12
    
```

1. b. Hasonlóan bővülő módon írd ki az első parancssori paramétert!

```

section .text
global main

main
    push    ebp
    mov     ebp, esp
    sub     esp, 4          ; [ebp-4] : ciklusszámláló

    mov     [ebp-4], dword 0
    ; kezdetben a szöveg nulladik pozícióján állunk

.ciklus
    mov     ecx, [ebp+12]
    mov     ecx, [ecx+4]    ; a szöveg (argv[1])
    mov     eax, [ebp-4]
    mov     dl, [ecx+eax]  ; a következő karakter

    cmp     dl, 0
    je     near .vege

    mov     [kiir + eax], dl
    mov     [kiir + eax + 1], byte 0xA
    ; a következő karakter és egy sorvége

    mov     edx, eax          ; a kiírás hossza
    add     edx, 2
    mov     eax, 4
    mov     ebx, 1
    mov     ecx, kiir
    int     0x80

    inc     dword [ebp-4] ; a következő karakter
    jmp     .ciklus

.vege
    mov     eax, 1
    xor     ebx, ebx
    int     0x80

section .data
szoveg    db        "Hello Vilag"

section .bss
kiir      resb 12
    
```

2. a. i) Alkoss a saját (lefordított, futtatható) programkódját kiíró programot! Feltételezheted, hogy a program neve ismert. Használj 1024 bites puffert a fájl tartalmának beolvasása során!  
 ii) Nehezítés: a program nevét a parancssori paraméterek közül kell kinyerned.
- b. i) Alkoss a saját forrásszövegét kiíró programot! Feltételezheted, hogy a program neve ismert.  
 ii) Nehezítés: a program neve a futtatott fájl neve .asm kiterjesztéssel.  
 Feltételezheted, hogy a fájl neve kevesebb, mint 256 karakter hosszú.

<pre> section .text global main  main push ebp mov ebp, esp  mov eax, 5 mov ebx, fajlnev  mov ecx, 644q int 0x80  mov [leiro], eax  .beolvas mov eax, 3 mov ebx, [leiro] mov ecx, adat mov edx, 1024 int 0x80  cmp eax, 1024 jne near .vege  mov eax, 4 mov ebx, 1 mov ecx, adat mov edx, 1024 int 0x80  jmp .beolvas  .vege mov edx, eax mov eax, 4 mov ebx, 1 mov ecx, adat int 0x80  mov eax, 1 mov ebx, 0 int 0x80  section .data fajlnev db"a.out", 0  section .bss adat resb 1024 leiro resd 1 </pre>	<pre> section .text global main  main push ebp mov ebp, esp  mov eax, 5 mov ebx, [ebp+12] mov ebx, [ebx] mov ecx, 644q int 0x80  mov [leiro], eax  .beolvas mov eax, 3 mov ebx, [leiro] mov ecx, adat mov edx, 1024 int 0x80  cmp eax, 1024 jne near .vege  mov eax, 4 mov ebx, 1 mov ecx, adat mov edx, 1024 int 0x80  jmp .beolvas  .vege mov edx, eax mov eax, 4 mov ebx, 1 mov ecx, adat int 0x80  mov eax, 1 mov ebx, 0 int 0x80  section .bss adat resb 1024 leiro resd 1 </pre>	<pre> section .text global main  main push ebp mov ebp, esp  mov eax, 5 mov ebx, fajlnev mov ecx, 644q int 0x80  mov [leiro], eax  .beolvas mov eax, 3 mov ebx, [leiro] mov ecx, adat mov edx, 1024 int 0x80  cmp eax, 1024 jne near .vege  mov eax, 4 mov ebx, 1 mov ecx, adat mov edx, 1024 int 0x80  jmp .beolvas  .vege mov edx, eax mov eax, 4 mov ebx, 1 mov ecx, adat int 0x80  mov eax, 1 mov ebx, 0 int 0x80  section .data fajlnev db "listazo2.asm", 0  section .bss adat resb 1024 leiro resd 1 </pre>	<pre> section .text global main  main push ebp mov ebp, esp  mov esi, [ebp+12] ;argv mov esi, [esi];argv[0] mov edi, fajlnev  .masol cmp byte [esi], 0 je near.masolas.vege  mov al, [esi] mov [edi], al  inc esi inc edi  jmp .masol  .masolas.vege mov byte [edi], '.' mov byte [edi+1], 'a' mov byte [edi+2], 's' mov byte [edi+3], 'm' mov byte [edi+4], 0  mov eax, 5 mov ebx, fajlnev mov ecx, 644q int 0x80  mov [leiro], eax  .beolvas mov eax, 3 mov ebx, [leiro] mov ecx, adat mov edx, 1024 int 0x80  cmp eax, 1024 jne near .vege  mov eax, 4 mov ebx, 1 mov ecx, adat mov edx, 1024 int 0x80  jmp .beolvas  .vege mov edx, eax mov eax, 4 mov ebx, 1 mov ecx, adat int 0x80  mov eax, 1 mov ebx, 0 int 0x80  section .bss adat resb 1024 leiro resd 1 </pre>
---	--	--	---

3. Írj olyan eljárást, amely egy duplaszón elférő, előjel nélküli egész számot ír ki a képernyőre tízes számrendszerben! A számot paraméterként kapja meg az eljárás. A kiírandó string legyen lokális változó (fix hosszúságú ASCII karaktertömb); az eljárásnak ezt töltsse fel, majd hívja meg a kiírást.

```

kiir
    push        ebp
    mov         ebp, esp
    sub         esp, 12          ; ezen a 12 byte-on alkotjuk meg a számot

    mov         byte [ebp-1], 0xA
    mov         byte [ebp-2], '0' ; kezdetben a szám egy nulla karakterből és egy sorvégéből áll

    mov         edi, ebp
    sub         edi, 2          ; edi a nulla karakteren áll
    mov         eax, [ebp+8]    ; eax az első paraméter

    cmp         eax, 0          ; ha a kapott paraméter nulla, akkor nullát kell kiírni
    je near     .kiir

    .ciklus
    cmp         eax, 0          ; ha a számból fennmaradó kiírandó rész nulla, akkor készen
    je near     .vege          ; vagyunk az átalakítással

    xor         edx, edx        ; 32 bitesen osztunk, ehhez kinullázzuk edx:eax felső felét

    mov         ecx, 10         ; osztás tízzel, a maradék (a következő karakter) edx-ben
    div         ecx             ; jelenik meg; mivel értéke 0..9 közötti, ezért dl-ben elfér

    add         dl, 30h         ; dl eddig szám volt, most számjeggyé alakítjuk
    mov         [edi], dl      ; ... és beírjuk az aktuális pozícióra

    dec         edi             ; a mutató léptetése
    jmp         .ciklus

    .vege
    inc         edi             ; a ciklusmag utolsó végrehajtása végén túlléptettük edi-t,
                                ; ezt itt korrigáljuk

    .kiir
    mov         eax, 4
    mov         ebx, 1
    mov         ecx, edi        ; kiírni az aktuális pozícióról fogunk
    mov         edx, ebp
    sub         edx, ecx        ; ki kell írni az összes karaktert edi-től ebp-ig
    int         0x80

    mov         esp, ebp
    pop         ebp
    ret                         ; visszatérés az alprogramból

```

## Makrók

1. Írd meg az adc utasítást általánosan, kétparaméteres makróként!

```
%macro    adc    2
    jnc %%nincs_carry    ; ha nincs beállítva az átviteli bit, az utasítás úgy működik,
                        ; mint az összeadás
    inc %1                ; különben eggyel meg kell növelni a regiszter értékét
%%nincs_carry
    add %1, %2
%endm
```

2.

				/	\			
				/	\			
		/	\					
+	-	-	-	-	-	-	-	+
					o	o		
					o	o		
=	=	=	=	=	=	=	=	=

		v	v	v	v	v	v		
	=	=	=	=	=	=	=	=	
			X				X		
			~	-	-	-	~		
	V	-	-	-	-	-	-	V	

3. a. Készíts deriváló makrót. A makró tetszőleges számú paramétert kaphat, amelyek egy polinom együtthatóit adják. A makró tegye a verembe sorban a polinom deriváltjának együtthatóit szavakként! A deriválás szabálya:

$$(c \cdot x^n)' = c \cdot n \cdot x^{n-1}$$

b. Készíts integráló makrót a deriváló makróhoz hasonlóan. A konstans tagot nem kell figyelembe venni. Feltehető, hogy minden kapott együttható egész szám.

```
%macro    deriv    1-*
    %assign    %%i    %0
    %rep %0 - 1
        push    dword ( %1 * %%i )
        %assign %%i    ( %%i - 1 )
        %rotate 1
    %endrep
%endm
```

```
%macro    integral    1-*
    %assign    %%i    %0 + 1
    %rep %0 - 1
        push    dword ( %1 / %%i )
        %assign %%i    ( %%i - 1 )
        %rotate 1
    %endrep
%endm
```

4. Add meg, hogy a parancssori paraméterként kapott file a legvégén tartalmazza-e a saját nevét. A fájl egy sorvége karakterrel zárul, ezt ne vedd figyelembe.

```

section .text
global main

main
    push ebp
    mov ebp, esp

    mov esi, [ebp+12]
    mov esi, [esi+4]      ; esi a fájlnev kezdetére mutat
    mov eax, -1

.hossz
    inc eax
    inc esi
    cmp byte [esi-1], 0
    jne .hossz           ; megszámoljuk, hány karakter hosszú a fájlnev

    mov [hossz], eax

    mov eax, 5
    mov ebx, [ebp+12]
    mov ebx, [ebx+4]
    mov ecx, 0
    mov edx, 777q
    int 0x80            ; megnyitjuk a fájlt

    mov [leiro], eax    ; eltároljuk a fájlleíró

    mov eax, 19
    mov ebx, [leiro]
    mov ecx, [hossz]
    neg ecx
    dec ecx
    mov edx, 2
    int 0x80           ; pozícionálás a szöveg kezdetére a fájl végén

    mov eax, 3
    mov ebx, [leiro]
    mov ecx, filevege
    mov edx, [hossz]
    int 0x80          ; beolvasás

    mov ecx, [hossz]   ; ecx a ciklusszámláló
    mov esi, [ebp+12]
    mov esi, [esi+4]   ; esi a fájlnev kezdetére mutat
    mov edi, fajlvege ; edi a fájl végéről származó, vizsgálandó szövegre
                    ; mutat

.vizsgal
    cmp ecx, 0
    je .egyezik       ; ha minden olvasott karakter egyezett, és már nincs
                    ; több, akkor a szöveg megfelelő

    mov al, [esi]
    cmp al, [edi]
    jne .nemegeyedik ; a következő karakter vizsgálata; ha eltér,
                    ; a szöveg nem megfelelő

    inc esi
    inc edi
    dec ecx
    jmp .vizsgal     ; ha egyezett a karakter, a következő vizsgálatával
                    ; folytatjuk

```

```

section .bss
hossz    resd 1
; a fájl kiszámított hosszát
; itt tároljuk átmenetileg

leiro    resd 1
; a fájl leírójának helye

fajlvege resb 16
; ide olvassuk fel a fájl
; feltételezzük, hogy 16
; karakternél nem hosszabb

```

## Irodalom

(1) [http://www.intel.com/design/intarch/intel386/docs\\_386.htm](http://www.intel.com/design/intarch/intel386/docs_386.htm)

Az Intel, az x86 architektúra megalkotóinak honlapja, azon belül a 80386-os processzor dokumentációs oldala. Innen elérhetőek a Software Developer's Manual (Szoftverfejlesztői kézikönyv) kötetei. Nagyon részletesen, több száz oldalon keresztül mutatja be az architektúra minden részletét.

(2) <http://developer.amd.com/documentation.aspx>

Az AMD, az x86 architektúrájú processzorok másik népszerű gyártójának honlapja. Az Architecture Programmer's Manual (Architektúra-programozói kézikönyv) írja le a processzorok felépítését és programozását. Az általunk tárgyalt pontokon nem tér el az Intel architektúrától.

(3) <http://nasm.sourceforge.net/>

A Netwide Assembler honlapja.

- <http://nasm.sourceforge.net/doc/nasmdoc0.html>

A Netwide Assembler on-line dokumentációjának tartalomjegyzéke.

- <http://nasm.sourceforge.net/doc/nasmdoc4.html>

A dokumentáció előfordítási fázisról (makrókról) szóló fejezete.

- <http://developer.apple.com/documentation/DeveloperTools/nasm/nasmdocb.html>

Az utasításokat bemutató függelék a dokumentációban. A munkafüzet írásakor a nasm hivatalos honlapján nem volt elérhető, pedig gyakorló assembly programozó számára a dokumentáció talán leghasznosabb része.

(4) <http://asm.sourceforge.net/>

Linux alatti assembly programozáshoz hasznos oldal. Régebben <http://linuxassembly.org/> címen működött.



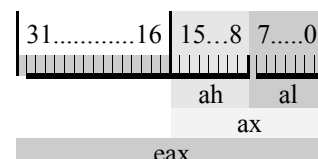
utasítás mnemonikája	operandusok	működés	
nop	<i>nincs operandusa</i>	semmit sem csinál	
adatmozgatás	mov	*	cél := forrás
	movsx   movzx	reg, reg/mem	cél := forrás előjeles/előjel nélküli kiterjesztéssel
	xchg	reg, reg/mem mem, reg	cél := forrás, forrás := cél
logikai	not	reg/mem	cél := cél bitenként negálva
	bt   bts   btc   btr	r/m <sup>32/16</sup> , r <sup>32/16</sup> /k <sup>8</sup>	átvitel := cél forrás-adik bitje, bit marad/0/1/fordul
	set <i>feltétel</i>	reg <sup>8</sup> /mem <sup>8</sup>	cél := 1, ha teljesül a feltétel, cél := 0, ha nem
aritmetikai	and   or   xor	*	cél := cél és/vagy/kizáró vagy forrás bitenként
	add   sub	*	cél := cél + forrás <i>illetve</i> cél := cél - forrás
	inc   dec		cél := cél + 1 <i>illetve</i> cél := cél - 1
	neg		cél := 0 - cél
	mul	reg/mem	előjel nélküli ax := al · forrás dx:ax := ax · forrás edx:eax := eax · forrás, <i>az op. méretétől függően</i> vagy
	imul		előjeles
	div		előjel nélküli al := ax / forrás, maradék: ah vagy ax := dx:ax / forrás, maradék: dx vagy
idiv	előjeles eax := edx:eax / forrás, maradék: edx		
<i>léptetések</i>	r/m, konst/cl	<i>külön táblázatban</i> (a 2. operandus lehet a cl regiszter is)	
vezérlésadás	cmp	*	összehasonlítás, a jelzőbitek beállítása
	jmp   <i>jfeltétel</i>	címke	ugrás feltétel nélkül vagy feltétellel; <i>külön táblázatban</i>
	call		alprogramhívás
	ret	<i>nincs operandusa</i>	visszatérés alprogramhívásból
	int	konst <sup>8</sup>	szoftveres kivétel kiváltása
veremkezelés	push	r <sup>32/16</sup> /m <sup>32/16</sup> /k <sup>32/16/8</sup>	érték verembe helyezése
	pop	reg <sup>32/16</sup> /mem <sup>32/16</sup>	érték visszatöltése a veremből
	pushad	<i>nincs operandusa</i>	eax, ecx, edx, ebx, esp, ebp, esi, edi verembe mentése
	popad	<i>nincs operandusa</i>	edi, esi, ebp, esp, ebx, edx, ecx, eax töltése a veremből

A táblázatok csak a legfontosabb utasítás-operandus párokat tartalmazzák. Pontos információk az AMD és az Intel dokumentációiban találhatóak.

\* Ezeknél az utasításoknál a reg/mem, reg/mem/konst operandusok megengedettek, a mem/mem kivételével.

feltételes ugrások	cél < forrás		cél ≤ forrás		cél ≥ forrás		cél > forrás	
	igaz	hamis	igaz	hamis	igaz	hamis	igaz	hamis
nem előjeles	je	jz	jbe	jnb	jae	jnb	ja	jnbe
előjeles	jl	jz	jle	jng	jge	jnl	jg	jnle

előjel nélküli léptetés		előjeles léptetés		forgatás	
shr	0 → felső bit 0. bit → átvitel	sar	↻ felső bit 0. bit → átvitel	ror	↻ felső bit 0. bit → átvitel
shl	átvitel ← felső bit 0. bit ← 0	sal	átvitel ← felső bit 0. bit ← 0	rol	átvitel ← felső bit 0. bit



A fentiekben r = reg = regiszter, m = mem = memóriatartalom, k = konst = konstans.  
 A felső indexben szereplő 8, 16 és 32 a méretet jelenti bitben.