7. Tétel

a. Ciklusok (iterációk), az Assembly és a C nyelv kapcsolata:

Ciklusok:

Akkor szoktunk ciklusokat szervezni, ha van egy tevékenységünk és azt sokszor kell elvégezni. Beszélünk **determinisztikus és nem determinisztikus ciklusokról**. A **determinisztikus ciklus** esetében tudjuk, hogy egy adott tevékenységet hányszor kell elvégezni. Magas szintű nyelvekben ezt "for" ciklusnak hívják. Például egy memóriaterület első három bájtját fel kell tölteni 0xAA értékkel. Az Intel Assembly-ben ez úgy kényelmes, hogy megadjuk a feltöltendő memóriacím szegmenscímét és az azon belüli indexet, majd egy tároló utasítást ciklikusan végrehajtunk.

Tudjuk, hogy a "REP" segítségével egy utasítást tudunk ciklikusan végezni. Hogy oldjuk akkor meg azt, hogy mégiscsak többet ismételjünk egyszerre, vagyis a ciklusmag egy utasításblokkból álljon? Megoldás lehet persze az is, hogy egy külső részben írunk egy sok utasításból álló rutint, s "RET"-tel (visszatérés) fejezzük be és ezt a rutint hívjuk egy utasítással, a "CALL"-lal ciklikusan. Ezt azonban már függvényhívásnak nevezik, és annak ciklikus formáját kultúrkörökben nem tartják teljesen etikusnak - ha csak nem direkt ez a cél...

A megoldás a "LOOP" utasítás: ezt a ciklusmag végére kell tenni és annyiszor ugrunk el vele a ciklus elejére, amennyit az ECX regiszterben beállítottunk.

```
    mov AL, 0AAh
    mov ECX, 3
    mov EDI, Offset terulet
    cld
    push CS
    pop ES
feltoltes:
    stosb
    loop feltoltes
    ; itt még lehetnek utasítások
terulet:
    db 0
    db 0
    db 0
```

Következzenek hát a **nem determinisztikus** ciklusok. Ezeknek két fajtája van: az elöltesztelő (while) és a hátultesztelő (do while). Az elöltesztelő ciklusnál a ciklusfejben van egy teszt, ha ez egy feltételt igaznak talál, akkor végrehajtódik a ciklusmag és visszatérünk a ciklusfejhez. Ezért hívjuk ezt while (amíg) ciklusnak. A hátultesztelő ciklusnál először lefut a ciklusmag, majd a ciklusfejben egy teszt eldönti, hogy szükséges-e a ciklusmagot újra végrehajtani. Ez más néven a do while (csináld, amíg) ciklus.

Nos, íme, egy **while** ciklus:

```
Ciklusfej:
    cmp AL, 0
    je vege
ciklusmag:
    dec AL
    jmp ciklusfej
vege:
```

Ha ezt C-nyelven írjuk le, az így néz ki:

```
   while( AL != 0x0 ) { --AL; }
```

Amíg "AL" nem egyenlő 0x0, addig dekrementáld (eggyel csökkentsd) "AL" értékét. Ha a program indulásakor "AL" értéke nulla volt, akkor be sem lépünk a ciklusba. Ellentétben áll ezzel a következő, do while példa(hátul tesztelős):

```
   ciklusmag:
      dec AL
   cikluslab:
      cmp AL, 0
      je vege
      jmp ciklusmag
   vege:
  vagyis

   do { --AL; } while( AL != 0x0 );
```

Először dekrementáljuk "AL" értékét, végül megnézzük, hogy nulla-e. Ha annyi, kilépünk, ha nem, akkor visszaugrunk a ciklusmagra.

C programba írt assembly utasítások

Egérpozíció állítása C-ben assembly hívással:

```
void SetMousePos(short x, short y) {
  _asm {
    mov ax, 4            ;Egérpozíció beállítás funkció
    mov cx, x            ;Koordináták (oszlop, sor)
    mov dx, y
    int 33h              ;Egér vezérlése szoftveres megszakítások át (reset mouse)
  }
}
```

Assembly rutinok C-hez

Egy MASM fordítónak megadható, hogy ha egy assembly rutint C programhoz írunk:
.MODEL SMALL,C

Emellett szükséges még a PUBLIC direktíva használata is, mely közli az assemblerrel, hogy az utána írt eljárást nyilvánossá kell tenni más programok számára. A PUBLIC miatt az assembler olyan információkat generál a LINK (szerkesztő) számára, amely lehetővé teszi a külső hozzáférést.

```
.MODEL SMALL,C
.CODE
PUBLIC clear_screen
clear_screen PROC
  XOR AL,AL                ;Ablak törlése
  XOR CX,CX                ;Bal felső sarok
  MOV DH,49                ;Képernyő alsó sora
  MOV DL,79                ;Jobb oldali oszlop sorszáma
  MOV AH,6                 ;Sorgörgetés felfelé
  INT 10H                  ;Képernyő törlése
  RET
clear_screen ENDP
```

Az AX, BX, CX és DX regiszterek tartalmát nem kell elmenteni ilyenkor. Az SI, DI, BP, SP, CS, DS és SS regiszterek tartalmának mentése és visszaállítása viszont szükséges.

Paraméterátadás

A C programok vermet használnak a paraméterek átadására, az assembly rutinokban viszont rendre regiszterekben történt az adatátadás. Azzal, hogy az assembler tudja, hogy C programban lesz használva az assembly eljárás, számos egyszerűsítés vihető végbe a programban. Egy C programban karakterkiíró program:

```
PUBLIC write_string
write_string PROC USES SI, string: PTR BYTE
  PUSHF
  CLD
  MOV SI, string
...
  RET
write_string ENDP
```

Az újdonságot két dolog jelzi:

- A USES SI, ami megmondja a fordítónak, hogy az SI regisztert fogjuk használni, tehát el kell menteni azt a verembe, majd visszatölteni
- A string nevű mutató, ami a kiírt sztring első karakterére mutat

Egy általános szubrutin hívásakor a következők történnek:

- A jobb szélső paramétertől kezdve betölti a paraméterek a verembe (azaz utoljára az első paraméter kerül be)
- Menti a visszatérési címet a verembe
- Átadja a vezérlést a függvénynek

A paraméterek által elfoglalt stack nagysága függ a paraméterek típusától, így paraméterek címe függ a típusuktól.

```
PUBLIC goto_xy
goto_xy PROC x:WORD, y:word
  MOV DH, BYTE PTR(y)
  MOV DL, BYTE PTR(x)
...
goto_xy ENDP
```

Függvényérték visszaadása

C-ben 3 regisztert használhatunk érték visszaadására:

- AL: byte hosszúságú adatnál
- AX: szó esetén
- DX:AX: dupla szavas adatnál

b. Adatábrázolás, adattípusok:

Adatábrázolás

Kétféle adatot különböztethetünk meg:

- A gépi számábrázolású adatok, ezeken végez műveleteket a számítógép
- Egyéb adatok, ezek állhatnak tetszőleges karakterekből. Ezeket a gép kódoltan ábrázolja, tárolja, és csak speciális esetben végez velük műveletet

Fixpontos számábrázolás

A legegyszerűbb számábrázolási mód, az egész számok ábrázolása és tárolása. Egy tetszőleges számot kettes számrendszerben, az annak megfelelő bithosszúságon kell tárolni. Általában a gépek 1,2,4 vagy 8 byte hosszúságban tárolnak értékeket. Pl.: 1 byte-on (8 bit) 2^8=256 különböző érték tárolható, azaz számok 0-255-ig.
Törtek esetén rögzítjük a tizedespont helyét, ettől lesz fixpontos a számábrázolás. Ennek a speciális esete az egész számok ábrázolása, amikor is a tizedespontot az utolsó bit után rögzítjük. Megkülönböztetünk *előjeles és előjel nélküli* számábrázolást is: az előjelesnél az első bit az előjelbit, pozitív számoknál 0, negatívoknál 1 az értéke.
Fixpontos számoknál figyelni kell a túlcsordulásra, azaz hogy egy művelet eredménye ne legyen hosszabb, mint ami a rendelkezésre álló helyen elférhet, azaz 1 byte esetén ne legyen nagyobb 255-nél. A túlcsordult számjegyek elvesznek, a maradékkal való számolás pedig hibás eredményt adhat. A fixpont hátránya még, hogy nem lehet tetszőleges pontossággal ábrázolni az értékeket.

Lebegőpontos számábrázolás

A fixpontos ábrázolás hátrányai küszöböli ki, de bonyolultabban kezeli a számokat. A lebegőpontos ábrázolás alapja, hogy a számok felírhatóak hatványkitevős alakban is: $\pm m \times p^k$, ahol a p a számrendszer alapszáma (általában 2), az m a mantissza, a k pedig a karakterisztika. Ahhoz, hogy a felírás mindig egyértelmű legyen, az m mindig kisebb mint 1, és a tizedesponttól balra eső számjegy nem lehet 0.
$1/p \leq m < 1$
A fenti felírási módot *normalizálásnak* nevezik, és a gépek automatikusan végzik el. Egy lebegőpontos szám ábrázolásához a következő adatokra van szükség:

- A mantissza abszolút értéke
- A mantissza előjele
- A karakterisztika abszolút értéke
- A karakterisztika előjele

A mantissza előjelét az első biten jelezzük. A következő bitek tartalmazzák a karakterisztikát, és jobb oldalon van a mantissza. A mantissza tárolására korlátozott számú bit van, így a pontosság behatárolt: ez a számábrázolási pontosság.

Decimális számok ábrázolása

Egy régebbi fajta ábrázolási megoldás, főleg adatbázisokban volt használatos. Itt kevés a művelet, de sok az adat beírás és megjelenítés, és ez a számrendszerek közötti konverziók számát növeli. Másrészt bizonyos felhasználások esetében nem megengedett a konverziók miatt fellépő pontatlanság, így olyan ábrázolásmódot találtak ki, aminél szükségtelen a kettes számrendszerre való konvertálás. Ezzel a módszerrel tetszőleges pontosságú szám ábrázolható. A számábrázolás formája a számítógéptől függött: az IBM gépeken a BCD, itt a számokat és az előjelet is fél byte-on ábrázolták.

Logikai értékek ábrázolása

Egy logikai állítás két értéket vehet fel (igaz vagy hamis), ezért tárolására elég egyetlen bit. Általában az 1 az igaz, a 0 pedig a hamis jelölése. De mivel gyakorlatban egy byte a legkisebb tárolható adatmennyiség, ezért muszáj ekkora helyen tárolni az értékeket.

Karakterek ábrázolása

Egy karaktert tároláskor a numerikus kódja azonosít, és az van letárolva. Az egyértelműséghez szükség van kódtáblázatok használatára, hogy világos legyen, melyik kódhoz melyik karakter tartozik. A legáltalánosabb az ASCII kódtábla, ez 128 karaktert tartalmaz, így 7 biten ábrázolható, a 8. bit a paritásbit. A nemzeti karakterek használata miatt a kódtábla később 8 bitesre nőtt, így lehetséges volt már 256-féle karakter ábrázolása.

Adattípusok

- Rövid egész: 1 byte-os
- Egész szám: 2 byte-os (word)

- Hosszú egész: 4 byte-os
- Kiterjesztett: 8 byte-os
- BCD egész: 10 byte-os
- Rövid valós: 4 byte-os (0-22 bit mantissza, 23-30 bit Exp, 31 bit előjel)
- Hosszú valós: 8 byte-os (0-51 bit mantissza, 52-62 bit Exp, 63 bit előjel)
- Kiterjesztett valós: 10 byte-os (0-63 bit mantissza, 64-78 bit Exp, 79 bit előjel)

Bármelyik típusú szám előjele az utolsó (legnagyobb helyiértékű) biten található. A valós számoknál a kitevő (Exp.) határozza meg, hogy mekkora a szám értelmezési tartománya, a mantissza nagysága a szám pontosságára van hatással.

c. Felülről-lefelé elemzés:

A kezdőszimbólumból (S), a szintaxisfa gyökeréből elindulva építjük fel a szintaxisfát. Célunk, hogy a szintaxisfa levelein az elemzendő szöveg terminálisai legyenek. Felülről-lefelé elemzéskor mindig a legbaloldalibb helyettesítéseket alkalmazzuk (ha $A \rightarrow \alpha \in P$, akkor az $x A \beta$ mondatforma legbaloldalibb helyettesítése $x\alpha\beta$, azaz $xA\beta \Rightarrow_{legbal} x\alpha\beta$). Végrehajtás az Earley algoritmussal:
Kiindulunk az S szimbólumból
Az S helyettesítésére az első olyan szabályt alkalmazzuk, amely bal oldalán S áll
Az így létrejött mondatforma legbaloldalibb nemterminálisára alkalmazzuk a szabályt
A létrejött mondatformára alkalmazzuk a 3. pontot mindaddig, míg
    a mondatforma baloldalán lévő terminálisok megegyeznek az elemzendő szöveggel
    a mondatformában van nemterminális szimbólum
Ha nincs több nemterminális és a mondatforma azonos az elemzendő szöveggel: VÉGE
Ha a mondatformában nincs több nemterminális, és nem egyezik meg a szöveggel, vagy a baloldalra kerülő terminálisok nem egyeznek meg a szöveg prefixével: VISSZALÉPÉS
Mindig az utoljára alkalmazott helyettesítést lépjük vissza:
Ha van következő helyettesítési szabály akkor azt hajtjuk végre, folytatás a 4.-től,
Ha S-hez jutunk vissza, és S-nek nincs több helyettesítési szabálya: VÉGE (nem mondat),
Ha a nemterminális nem S, és nincs több helyettesítési szabálya: egyet visszalépünk.

Példa: elemezzük a for a to b mondatot az alábbi grammatika segítségével (ponttal jelöljük, hogy hol tartunk az elemzésben).
$G=(\{A,B,S\},\{a,b,c,d\},P,S)$, $S \rightarrow aAd|aB$, $A \rightarrow b|c$, $B \rightarrow ccd|ddc$.

Lépés	Mondatforma	Állapot	Megjegyzés
1.	S	.accd	
2.	aAd	a.ccd	
3.	abd	a.ccd	a mondat nem azonos a szöveggel, vissza a 2-re
4.	aAd	a.ccd	
5.	acd	acc.d	a mondat nem azonos a szöveggel, vissza a 4-re
6.	aAd	a.ccd	A-ra nincs több szabály, vissza az 1-re
7.	S	.accd	
8.	aB	a.ccd	
9.	accd	accd.	az elemzés sikeres