Assembly programozás

Dorkó Arnold

2023. szeptember 17.

Tartalomjegyzék

1. Elmélet **4**
- 1.1. Logikai műveletek . . . 4
- 1.2. Neumann-elvek . . . 4
- 1.3. Bináris összeadás, kivonás, szorzás műveletek . . . 5
- 1.4. Neumann-ciklus . . . 5
- 1.5. Adattípusok . . . 6
- 1.6. Fixpontos számábrázolás . . . 6
- 1.7. Lebegőpontos számábrázolás . . . 6
- 1.8. Programozási nyelvek összefoglalása . . . 7
- 1.9. Fordítóprogramok működésének ismertetése . . . 7
- 1.10. Lexikális elemzőről bővebben . . . 9
- 1.11. Szintatikus elemzőről bővebben . . . 9
- 1.12. Felülről-lefelé elemzések . . . 11
- 1.13. LL(k) grammatikák . . . 12
- 1.14. Alulról-felfelé elemzés . . . 12
- 1.15. LR(k) grammatika . . . 12
- 1.16. Szemantikus elemzőről bővebben . . . 13
- 1.17. Formális nyelvek és automaták összefoglalása . . . 13
  - 1.17.1. Determinisztikus véges automaták . . . 13
  - 1.17.2. Formális nyelvek . . . 14
  - 1.17.3. Grammatikák . . . 14
  - 1.17.4. Chomsky-féle nyelvosztályok . . . 15

2. Intel processzorok regiszterkészlete **16**
- 2.1. Általános regiszterek . . . 16
- 2.2. Vezérlőregiszterek . . . 16
- 2.3. Szegmensregiszterek . . . 17
- 2.4. Státuszregiszter – Flags . . . 17

3. Ugró utasítások **19**
- 3.1. Feltétel nélküli ugrás . . . 19
- 3.2. Előjeles feltételes ugró utasítások . . . 19
- 3.3. Előjel nélküli feltételes ugró utasítások . . . 19
- 3.4. Flag-eken alapuló ugró utasítások . . . 19

4. Bitmozgató műveletek **20**
- 4.1. Léptetések (shift) . . . 20
- 4.2. Rotálások (rotate) . . . 21

5. Az Intel processzorok utasításrendszere **22**
- 5.1. Az Intel processzorokról . . . 22
- 5.2. Operandusok és címzési módok . . . 22
- 5.3. Verem (stack) . . . 24
- 5.4. Memóriaszervezés . . . 24

6. Assembly programozás **25**
6.1. Szegmentálás . . . 25
6.2. Konstansok használata . . . 25
6.3. Karakter bekérés és kiíratás . . . 25
6.4. Szubrutinok használata . . . 27
6.5. Karakterkód konvertálása bináris számmá és kiírása a képernyőre . . . 28
6.6. Karakterkód konvertálása hexadecimális számmá és kiírása a képernyőre . . . 29
6.7. Karakterkód konvertálása decimális számmá és kiírása a képernyőre . . . 30
6.8. Decimális szám beolvasó rutin . . . 31
6.9. Hexadecimális szám beolvasó rutin . . . 32
6.10. Bináris szám beolvasó rutin . . . 33
6.11. Osztás szubrutin . . . 34
6.12. Adatszegmens használata . . . 35
6.13. Adatszegmens és indirekt címzés . . . 36
6.14. Szövegsor kiíratása a képernyőre . . . 37
6.15. write_string szubrutin . . . 38
6.16. DUP operátor és a kérdőjel . . . 39
6.17. Karakterek kiírása a-z-ig . . . 40
6.18. Három decimális szám szorzása . . . 41
6.19. Bináris szám beolvasó, csak 0 és 1 . . . 42
6.20. Pointerek TYPEDEF és PTR . . . 43
6.20.1. TYPEDEF direktíva . . . 43
6.20.2. PTR direktíva . . . 43
6.21. Sztring kiíratás/beolvasása INT 21h-val . . . 45
6.22. Lemezmeghajtó kezelése . . . 46
6.23. Karakteres videó-memória kezelése . . . 48

7. Forrás **48**

1. Elmélet

1.1. Logikai műveletek

Definíció (Negáció művelet)**.** Egyváltozós logikai művelet, mely az állítás tagadását adja eredményül.

$$|\neg p| = \begin{cases} 0, & \text{ha} \quad |p| = 1 \\ 1, & \text{ha} \quad |p| = 0 \end{cases}$$

Definíció (Konjunkció / logikai és)**.** Kétváltozós logikai művelet, amely pontosan akkor igaz, ha mindkét változó logikai értéke igaz. Ellentettje a NAND.

$$|p \wedge q| = \begin{cases} 0, & \text{más esetben} \\ 1, & \text{ha} \quad |p| = |q| = 1 \end{cases}$$

Definíció (Diszjunkció / logikai vagy)**.** Kétváltozós logikai művelet, amely pontosan akkor igaz, ha legalább az egyik változó logikai értéke igaz. Ellentettje a NOR.

$$|p \vee q| = \begin{cases} 1, & \text{más esetben} \\ 0, & \text{ha} \quad |p| = |q| = 0 \end{cases}$$

Definíció (Implikáció)**.** Kétváltozós logikai művelet, amely akkor hamis, ha $|p| = 1$ és $|q| = 0$.

$$|p \to q| = \neg(p \wedge \neg q) = \neg p \vee q$$

Definíció (Ekvivalencia)**.** Kétváltozós logikai művelet, amely akkor igaz, ha mindkét változó logikai értéke megegyezik. Ellentettje a XOR.

$$|p \leftrightarrow q| = (p \to q) \wedge (q \to p)$$

Definíció. Operátorok precedenciája: $\neg, \wedge, \vee, \to, \leftrightarrow$

1.2. Neumann-elvek

- Soros utasításvégrehajtás
- Bináris számrendszer használata
- Legyen teljesen elektronikus működésű
- Belső memória (operatív tár) adatok és programok tárolására
- A tárolt program elve
- A számítógép legyen univerzális: A számítógép különféle feladatainak elvégzéséhez nem kell speciális berendezéseket készíteni. Turing angol matematikus bebizonyította, hogy az olyan gép, amely el tud végezni néhány alapvető műveletet, akkor az elvileg bármilyen számítás elvégzésére is alkalmas
- A Neumann-elvű számítógépek elméleti felépítése:
  - Központi feldolgozó egység

    - a vezérlő egység (control unit)
    - az aritmetikai és logikai egység (ALU)
    - Regiszterblokk
    - Gyorsítómemória (cache)
    - Matematikai társprocesszor FPU
  - az operatív tár, ami címezhető és újraírható tároló-elemekkel rendelkezik
  - a ki/bemeneti egységek (perifériák)
  - háttértár (merevlemez, ssd)

1.3. Bináris összeadás, kivonás, szorzás műveletek

- **Összeadás**: A bináris összeadás hasonló, mint a decimális számoknál: összeadjuk az adott helyi értéken a számokat, és ha van maradék, akkor azt hozzáadjuk a magasabb helyi érték összegéhez.

A		B		S	C
0	+	0	=	0	0
0	+	1	=	1	0
1	+	0	=	1	0
1	+	1	=	0	1

- **Kivonás**: A kivonandó számhoz hozzáadjuk a különbség kettes komplemensét

- **Szorzás**: A bináris szorzást úgy végezzük, mintha decimális számok lennének: a szorzandót megszorozzuk egyesével a szorzó egyes helyi értékein álló számmal. A részszorzatokat úgy írjuk le, hogy mindig egy helyi értékkel jobbra toljuk azokat, majd az így kapott részeredményeket összeadjuk.

A		B		S
0	$*$	0	=	0
0	$*$	1	=	0
1	$*$	0	=	0
1	$*$	1	=	1

  Bájtos szorzásnál legalább kétbájtnyi adatterületre van szükség az eredmény tárolásához, így az eredmény egy kétbájtos regiszterbe kerül. Ha kétbájtos adatokkal dolgozunk, akkor négybájtos lesz a szorzat

1.4. Neumann-ciklus

1. Program betöltése és PC beállítása
2. Utasítás lehívása
3. PC növelése
4. Végrehajtás
5. Ha nincs vége vissza az elejére

1.5. Adattípusok

- Rövid egész (1 bájt)
- Egész szám (2 bájt)
- Hosszú egész (4 bájt)
- Kiterjesztett (8 bájt)
- BCD egész (10 bájt)
- Rövid valós (4 bájt)
- Hosszú valós (8 bájt)
- Kiterjesztett valós (10 bájt)

1.6. Fixpontos számábrázolás

- Egy tetszőleges egész számot kettes számrendszerben, a megfelelő bit-hosszúságban kell tárolni.
- Szabvány szerint a számítógépek ezeket 1, 2, 4 vagy 8 bájt hosszúságban tárolják
- Lehetséges törtek tárolása is
- Törtek esetében a tárolót egészrészre és törtrészre bontjuk
- Egész számok esetén a kettedespontot az utolsó bit után rögzítjük
- Az előjeles abban tér el az előjel nélkülitől, hogy a szám legnagyobb helyi értékű bitjét kinevezik előjelbitnek, amely pozitív számok esetén 0, negatív számoknál pedig 1

1.7. Lebegőpontos számábrázolás

- A lebegőpontos számábrázolás alapja az, hogy a számok hatványkitevős alakban is felírhatók: $\pm m \cdot p^k$, ahol p a számrendszer alapszáma, m mantissza, k karakterisztika és $\frac{1}{p} \leqslant m < 1$
- A fenti feltételeket teljesítő felírási módot normalizálásnak nevezzük, amit természetesen a számítógépek automatikusan elvégeznek. Egy, a gyakorlatban is használható lebegőpontos szám tárolására minimum 32 bitre (4 bájtra) van szükség.

1.8. Programozási nyelvek összefoglalása

- **Gépi kód**: Az utasításoknak egy-egy numerikus kód felel meg, az utasítások operandusaiban a memóriacímekre numerikus értékekkel, a memóriarekeszek címeivel kell hivatkozni
- **Assembly nyelv**: Az Assembly nyelv utasításai a gépi kódú szimbolikus utasítások megfelelői. Az assembly nyelven írt program sorokból áll, minden sor címkemezőre, utasításmezőre és operandusmezőre bontható
- **Assembler**: Az assembly nyelvű programot gépi kódú programra átalakító fordítóprogramot assemblernek nevezzük.
- **Magasszintű nyelvek**:
  - Imperatív programozási nyelvek
    - Értékadások (Assignment)
    - Szekvencia (Sorrendiség)
    - Elágazások (Szelekció)
    - Ciklusok
    - Részeredmények számítása
    - Visszatérési értékek
  - Deklaratív programozási nyelvek

1.9. Fordítóprogramok működésének ismertetése

- **Fordítóprogram**: A magasszintű nyelven írt programból egy alacsonyabb szintű (például assembly nyelvű vagy gépi kódú) programot készítünk. Az ilyen átalakítást végző fordítóprogramot **compilernek** nevezzük. A fordítóprogram a forrásprogram beolvasása után elvégzi a lexikális, szintaktikus és szemantikus elemzést, előállítja a szintaxis fát, generálja, majd optimalizálja a tárgykódot.
- **Többmenetes fordító**: A fordítási fázisok több különböző menetben futnak le. Ilyenkor szükség van közbenső programformák tárolására, ezek az egyes menetek outputjai és a másikak inputjai. Sok esetben a többmenetes fordítás az egyetlen lehetőség a nyelv komplexitása miatt.
- **Forrásprogram**: A fordítóprogram bemenetjét nevezzük így, amelyet át fog alakítani
- **Tárgyprogram**: A lefordított programot nevezzük így
- **Fordítóprogram szerkezete**:
  - **Analízis**: Lefordítja a forráskódot egy közbeeső kódra
    1. Lexikális elemző
    2. Szintatikus elemző
    3. Szemantikus elemző
  - **Szintézis**: Itt történik az úgynevezett fordítási eljárás
    1. Kódgeneráló
    2. Kódoptimalizáló

3. Tárgykód kiíratása a háttértárra

- **A lexikális elemző feladata**:
  - A bemenetként kapott karaktersorozatból **szimbólumsorozatot** készít, ki kell szűrnie a szóköz karaktereket a kommenteket, mivel ezek tárgykódot nem adnak.
  - A karaktersorozatban meghatározza az egyes szimbolikus egységeket, a **konstansokat**, **változókat**, **kulcsszavakat** és **operátorokat**. (Szimbólumtábla létrehozása)
  - A magasszintű programnyelvek egy utasítása több sorba is írható, a lexikális elemző feladata egy több sorba írt utasítás összeállítása is
  - A lexikális elemző által készített szimbólumsorozat (terminális szimbólumokból áll) a bemenete a szintatikus elemzőnek.
  - **Reguláris nyelvekkel** dolgozik
  - Ha a lexikális elemző egy karaktersorozatnak nem tud egy szimbólumot sem megfeleltetni, akkor azt mondjuk, hogy a karaktersorozatban **lexikális hiba** van (illegális karakterek, karakterek felcserélodnek vagy esetleg karakterek hiányoznak)
- **A szintatikus elemző feladata**:
  - A **környezetfüggetlen** grammatikákkal leírható tulajdonságok vizsgálatát szintaktikus elemzésnek nevezzük
  - A szintaktikus elemzés feladata eldönteni azt, hogy ez a szimbólumsorozat a nyelv egy mondata-e
  - Feladata a program struktúrájának a felismerése és ellenőrzése
  - A szintatikus elemző azt vizsgálja, hogy a bemeneti szimbólumsorozatban az egyes szimbólumok a megfelelő helyen vannak-e, a szimbólumok sorrendje megfelel-e a programnyelv szabályainak, nem hiányzik esetleg valahonnan egy szimbólum
  - A szintaktikus elemző működésének az eredménye az elemzett program szintaxisfája és ha vannak, akkor a szintaktikai hibák
- **A szemantikai elemző feladata**:
  - A forrásprogram környezetfüggő szabályainak ellenőrzése
    - Típusellenőrzés
    - Láthatóság-ellenőrzés
    - Eljárások hívásának ellenőrzése
  - Bemenete a szintaxisfa, szimbólumtábla
  - A szemantikus elemző feladat például az a+b kifejezés elemzésekor az, hogy az összeadás műveletének a felismerésekor megvizsgálja, hogy az „a” és a „b” változók deklarálva vannak-e, azonos típusúak-e és hogy van-e értékük.
  - Konstansok, változók érték- és típusellenőrzése
  - Aritmetikai kifejezések ellenőrzése
  - A szemantikus elemző kimenete lesz a szintetizálást végző programok bemenő adata, szemantikai hibák.
- **Szintézis**:

  - **Kódgenerátor**: Bemenete a szemantikusan elemezett program és kimenete a tárgykód, operációs rendszertől és platformtól függő
  - **Kódoptimalizáló**: (tárgykód): A kódoptimalizálás a legegyszerűbb esetben a tárgykódban lévő azonos programrészek felfedezését és egy alprogramba való helyezését vagy a hurkok ciklus változásától független részeinek megkeresését és a hurkon kívül való elhelyezését jelenti. Egy jó kódoptimalizálónak jobb és hatékonyabb programot kell előállítania, mint amit egy gyakorlott programozó tud elkészíteni.

- **Interpreter**: Hardveres, vagy virtuális gép, mely értelmezni képes a magas szintű nyelvet, vagyis melynek gépi kódja a magasszintű nyelv, futásidő és a fordítási idő egyben van.
- **Menetszám**: A fordítás annyi menetes, ahányszor a programszöveget (vagy annak belső reprezentációit) végigolvassa a fordító a teljes fordítási folyamat során.

1.10. Lexikális elemzőről bővebben

- A lexikális elemző **reguláris nyelvtanokkal** foglalkozik
- Mindig a **leghosszabb** karaktersorozatot ismeri fel
  - Az abc bemenet esetén a, ab és abc is legális változónév. Melyiket ismeri fel? A fentiből következik, hogy az "abc" a helyes.
- **Kulcsszó**: Olyan karaktersorozat, amelynek a jelentése nem írható felül pl. `while`
- **Komment**: pl. `/* Hello World! */`
- **Operátor**: pl. `+, -, *`
- **Azonosító**: pl. `valtozo`
- **Literál**: pl. `"alma", 3.14, true`
- **Elválasztó**: pl. zárójelek
- A kulcsszavakat nem véges determinisztikus automatákkal adjuk meg (mert így túl nagy automatát kapnánk), hanem táblázatban tároljuk őket, és ha egy beolvasott token benne van a táblázatban akkor azt nevezzük kulcsszónak
- A karaktersorozatok értelmezésekor mindig a kulcsszavakat kell előre venni, tehát `while` lehetne változó is de ez egy kulcsszó
- Hibákat felismer pl. illegális karakter és elgépelt kulcsszó

1.11. Szintatikus elemzőről bővebben

- A szintaktikus elemző a **környezetfüggetlen nyelvtanokkal** foglalkozik
- Gyakori jelölése a formális nyelveknél:
  - a, b, c - terminális szimbólum
  - A, B, C - nemterminális szimbólum

  - α, β, γ - terminális vagy nemterminális szimbólumok sorozata
  - x, y, z - terminális szimbólumok sorozata

- Grammatikában levezetések jelölése:
  - $A \models \alpha$ (1 lépéses levezetés)
  - $A \overset{*}{\models} \alpha$ (0, 1 vagy több lépéses levezetés)
  - $A \overset{+}{\models} \alpha$ (1 vagy több lépéses levezetés)

Definíció (Mondat)**.** Legyen $G =< T, N, S, P >$ egy grammatika. Ha $S \overset{*}{\Rightarrow} x$, akkor az x a grammatika által definiált nyelv egy mondata és mondatforma is (x nemterminális szimbólum).

Definíció (Mondatforma)**.** Legyen $G =< T, N, S, P >$ egy grammatika. Ha $S \overset{*}{\Rightarrow} \alpha$, akkor az α egy mondatforma (α tartalmaz terminális és nemterminális szimbólumokat is).

Definíció (Részmondat)**.** β az $\alpha_1\beta\alpha_2$ mondatforma részmondata, ha $S \overset{*}{\Rightarrow} \alpha_1 A \alpha_2 \overset{+}{\Rightarrow} \alpha_1\beta\alpha_2$

Definíció (Egyszerű részmondat)**.** β az $\alpha_1\beta\alpha_2$ mondatforma egyszerű részmondata, ha $S \overset{*}{\Rightarrow} \alpha_1 A \alpha_2 \Rightarrow \alpha_1\beta\alpha_2$

Definíció (Ciklusmentesség)**.** Nincs olyan szabály melyre, $A \overset{+}{\models} A$

Definíció (Redukáltság)**.** Nincsenek felesleges nemterminális jelek.

Definíció (Egyértelműség)**.** Minden mondathoz pontosan egy szintaxisfa tartozik.

Definíció (Legbal levezetés)**.** Mindig a legbaloldalibb nemterminálist helyettesítjük

Definíció (Legjobb levezetés)**.** mindig a legjobboldalibb nemterminálist helyettesítjük

Definíció (Felülről-lefelé elemzés)**.** A startszimbólumból indulva, felülről lefelé építjük a szintaxisfát. A mondatforma baloldalán megjelenő terminálisokat illesztjük az elemzendő szövegre. Azt hogy mikor melyik levezetési szabályt kell használni ezekkel a módszerekkel dönthejük el:

- **Visszalépéses keresés** (backtrack): ha nem illeszkednek a szövegre a mondatforma baloldalán megjelenő terminálisok, lépjünk vissza, és válasszunk másik szabályt (lassú)
- **Előreolvasás**: olvassunk előre a szövegben valahány szimbólumot, és az alapján döntsünk az alkalmazandó szabályról (**LL elemzések**)

Definíció (Alulról-felfelé elemzés)**.** Az elemzendő szöveg összetartozó részeit helyettesítjük nemterminális szimbólumokkal (redukció) és így alulról, a startszimbólum felé építjük a fát. Azt hogy mikor melyik levezetési szabályt kell használni ezekkel a módszerekkel dönthejük el:

- **Visszalépéses keresés** (backtrack): ha nem sikerül eljutni a startszimbólumig, lépjünk vissza, és válasszunk másik redukciót (lassú)
- **Precedenciák**
- **Előreolvasás**: olvassunk előre a szövegben valahány szimbólumot, és az alapján döntünk a redukcióról (**LR-elemzések**)

- A programozó által írt programot a lexikális elemző terminális szimbólumokból álló sorozattá alakítja, ez a terminálisokból álló sorozat a szintaktikus elemzés inputja
- A szintaktikus elemzés feladata eldönteni azt, hogy ez a szimbólumsorozat a nyelv egy mondata-e. A szintaktikus elemzőnek ehhez például meg kell határoznia a szimbólumsorozat szintaxisfáját, ismerve a szintaxisfa gyökérelemét és a leveleit, elő kell állítania a szintaxisfa többi pontját és élét, vagyis meg kell határoznia a program egy levezetését
- Ha ez sikerül, akkor azt mondjuk, hogy a program eleme a nyelvnek, azaz a program szintaktikusan helyes
- A szintaxisfa belső részének felépítésére több módszer létezik. Az egyik az, amikor az S szimbólumból kiindulva építjük fel a szintaxisfát, ezt felülről-lefelé haladó elemzésnek nevezzük. Ha a szintaxisfa építése a levelekből kiindulva halad az S szimbólum felé, akkor alulról-felfelé elemzésrol beszélünk.
- Egy mondat szintaxisfájának levelei a grammatika terminális szimbólumai, a szintaxisfa többi pontja a nemterminális szimbólumokat reprezentálja, a gyökérelem pedig a grammatika kezdőszimbóluma

1.12. Felülről-lefelé elemzések

- A kezdőszimbólumból (S), a szintaxisfa gyökeréből elindulva építjük fel a szintaxisfát
- Célunk, hogy a szintaxisfa levelein az elemzendő szöveg terminális szimbólumai legyenek
- Elemzéskor mindig a legbaloldalibb helyettesítéseket alkalmazzuk
- A bemeneti szövegben balról jobbra haladunk
- **Teljes visszalépéses elemzés** (Earley-algoritmus):
  1. Kiindulunk az S szimbólumból
  2. Az S helyettesítésére az első olyan szabályt alkalmazzuk, amely bal oldalán S áll
  3. Az így létrehozott mondatformában a legbaloldalibb nemterminálist a nemterminális első helyettesítési szabályával helyettesítjük
  4. A létrehozott új mondatformára a 3. pont műveletét ismételjük addig, amíg:
     - a mondatforma bal oldalára kerülő terminálisok megegyeznek az elemzendő szöveg prefixével, és
     - a mondatformában van nemterminális szimbólum
  5. Ha a mondatformában nincs több nemterminális szimbólum és a mondatforma azonos az elemzendő szöveggel, akkor az elemzést befejezzük, az elemzés sikeres, azaz az elemzett szöveg szintaktikusan helyes.
  6. Ha a mondatformában nincs több nemterminális szimbólum és a mondatforma nem egyezik meg az elemzendo szöveggel, vagy a mondatforma bal oldalára kerülő terminálisok nem egyeznek meg az elemzendő szöveg prefixével, akkor visszalépés következik
  7. Visszalépéskor az utoljára alkalmazott helyettesítést lépjük vissza, és az így kapott mondatforma legbaloldalibb nemterminális szimbólumára próbáljuk meg alkalmazni ennek a nemterminálisnak a következo helyettesítési szabályát:

- ha van következo helyettesítési szabály, akkor hajtsuk végre ezt a helyettesítést, folytassuk az elemzést a 4. pontban leírtakkal
- ha visszalépéskor az S szimbólumhoz jutunk vissza, és az S-nek nincs már további helyettesítési szabálya, akkor az elemzést befejezzük azzal, hogy az elemzendő szöveg nem mondata a grammatika által definiált nyelvnek, azaz az elemzett szöveg szintaktikusan hibás
- ha a nemterminális szimbólum nem az S, és ennek a szimbólumnak nincs már több helyettesítési szabálya, akkor még egy helyettesítést vissza kell lépnünk, azaz folytassuk az elemzést a 7. ponttal.

1.13. LL(k) grammatikák

- Olyan felülrol-lefelé elemzésekkel, melyek k szimbólum előreolvasásával egyértelműen meg tudják határozni a következő lépést, és az elemzés folyamán visszalépésekre nincs szükség
- LL = Left to right, tracing a Leftmost derivation
- **LL(1) grammatika**:
  - ϵ mentes
  - A helyettesítési szabályok jobboldala terminális szimbólummal kezdődik
  - Alternatívák esetén a jobboldalak kezdő terminális páronként különbözőek, azaz $A \rightarrow a_1\alpha_2|\ldots|a_k\alpha_k$ ahol $a_i \neq a_j$, ha $i \neq j$ $(1 \leqslant i, j \leqslant k)$

1.14. Alulról-felfelé elemzés

- Alulról felfelé elemzéseknél a terminális szimbólumokból kiindulva haladnak a kezdőszimbólum felé
- Ezek az elemzések egy vermet használnak, és a vizsgált módszerek léptetés-redukálás típusúak, azaz az elemző mindig egy redukciót próbál végezni, ha ez nem sikerül, akkor az input szimbólumsorozat következő elemét lépteti
- Ezek az elemzések a legjobboldalibb levezetés inverzét adják. Az elemzés alapproblémája a mondatformák nyelének a meghatározása
- Szintaxisfa felépítésénél az elemzendő szimbólumsorozatból indulunk ki, megkeressük a mondatforma nyelét, amit helyettesítünk a hozzá tartozó nemterminális szimbólummal.
- A cél, hogy elérjük a grammatika kezdőszimbólumát, ami a szintaxisfa gyökérszimbóluma. A fa levelei pedig az elemzendő program terminális szimbólumai kell, hogy legyenek. Egy mondatforma legbaloldalibb egyszerű részmondatát a mondatforma nyelének nevezzük

1.15. LR(k) grammatika

- Az LR a balról jobbra ("Left to Right") történő elemzésre utal, a k pedig azt jelenti, hogy k szimbólumot előreolvasva egyértelműen meghatározható a mondatforma nyele
- Első lépésben a grammatika szabályaiból egy táblagenerátorral elkészítünk egy elemző táblázatot, majd az elemző program a táblázattal és egy veremmel tudja elemezni a szöveget

1.16. Szemantikus elemzőről bővebben

- A szemantikus elemzés jellemzően a **környezetfüggő ellenőrzéseket** valósítja meg
- **Statikus típusozás**: A fordítási időben kalkulálódik és ellenőrződik a típus
- **Dinamikus típusozás**: A futási időben derül ki és ellenőrződik minden utasításnál a típus.
- **Típusellenőrzés**: Nekünk kell megadni a változók típusát
- **Típuslevezetés, típuskikövetkeztetés**: A fordítóprogram találja ki a változó típusát a megadott érték alapján

1.17. Formális nyelvek és automaták összefoglalása

1.17.1. Determinisztikus véges automaták

Definíció. A determinisztikus véges automatákat az alábbi rendezett ötössel definiáljuk:

$$M(Q, \Sigma, \delta, q_0, F)$$

, ahol:

- Q - az automata állapotainak véges halmaza
- Σ - a vizsgálandó jelsorozat ABC-je
- δ - automata mozgási szabályainak véges halmaza
- q_0 - az induló állapot, $q_0 \in Q$
- F - az elfogadó állapotok véges halmaza

továbbá: $F \subset Q$ és a mozgási szabályok azt mondják meg, hogy egy adott karakter beolvasására egy adott állapotból melyik másik állapotba lépünk át, ezt így írjuk matematikailag: $\delta(q_0, A) = B$. A szabályok halmaza egy leképezés tulajdonképpen: $Q \times \Sigma \to Q$.

Lehetnek olyan állapot-karakter párok, amelyek nincsenek definiálva a leképezésben. Továbbá ha egy állapotból egy bizonyos karakter hatására több állapotba is léphetünk akkor beszélünk **nemdeterminisztikus automatáról**, egyébként pedig **determinisztikus automatáról** van szó.

Az automatának van egy olvasófeje, egy szalagja amin vannak a karakterek. Akkor fogad el egy automata egy bemenetet, ha végigolvasta a bemenetet és végállapotban állt meg az automata. A determinisztikus véges automaták által felismert nyelvosztály a **reguláris nyelvek** osztálya.

A véges automaták egy **irányított gráffal** szemléltethetőek, és az egyes csomópontjai felelnek meg az automata állapotainak. Amennyiben egy karakter beolvasásának hatására az automata egyik állapotból egy másik állapotba megy át, akkor a két állapotnak megfelelő csomópontokat egy, az eredeti állapotból az új állapotba mutató, és az olvasott karaktert mint nevet viselő éllel kötjük össze.

Ha minden állapot–karakter pár esetében van mozgási szabály, akkor az automata **teljesen specifikált**.

Definíció. Az A véges determinisztikus automata által felismert nyelvnek azt a $T(A)$ nyelvet nevezzük, amely mindazon V-ből alkotott szavakból áll, melyeket az automata felismer.

$$T(A) = \{\alpha \in V^* \mid \alpha \text{ felismeri } A\}$$

Definíció. Két determinisztikus véges automatát, A_1-et és A_2-t akkor nevezünk ekvivalensnek, ha $T(A_1) = T(A_2)$, azaz, ha ugyanazt a nyelvet ismerik fel.

Tétel. Tetszőleges nem teljes definiált véges determinisztikus automatához létezik teljes definiált automata.
Bizonyítás: Elnyelő állapotot hozzunk létre, amelybe mennek nyilak a nem definiált állapotokból.

1.17.2. Formális nyelvek

Definíció (Ábécé)**.** Legyen V egy véges nem üres halmaz ($V \neq \varnothing$), ún. jelek halmaza. Ekkor V egy abc.

Definíció (Szó)**.** Legyen V egy abc. Ekkor a V abc feletti szón a V abc elemeiből álló véges, de nem korlátos hosszúságú jelsorozatot szónak hívjuk. (Az abc elemeit egymás mellé írjuk.)

Definíció (Összes V feletti szó)**.** V^* az összes V feletti szó halmaza (ami mindig végtelen!).

Definíció (Formális nyelv)**.** L a V abc feletti formális nyelv, ha:

$$L \subseteq V^*$$

Definíció (Formális nyelv iteráltja)**.** Ha L formális nyelv, akkor az L iteráltja a:

$$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup \cdots \cup L^i$$

Definíció. Nyelvek Chomsky-féle osztályozása: Legyen $L \subseteq V^*$ formális nyelv. Azt mondjuk, hogy L $i.$ típusú (i = 0, 1, 2, 3), ha $\exists G$ $i.$ típusú generatív grammatika, úgy hogy $L(G) = L$.

1.17.3. Grammatikák

Definíció. Formálisan a grammatikát egy négyes határozza meg:

$$G = (T, N, S, P)$$

, ahol:

- T - terminális jelek
- N - nemterminális jelek, és $T \cap N = \varnothing$
- S - A G kezdőszimbóluma, és $S \in N$
- P - helyettesítési szabályok

A nemterminális szimbólumokat a latin ábécé elejéről vett nagybetűk jelölik. A terminális szimbólumokat a latin ábécé elejéről vett kisbetűk jelölik. Az olyan jelsorozatokat, amelyek tartalmazhatnak mind terminális, mind nemterminális szimbólumokat is görög kisbetűk jelölik. A kezdőszimbólumot az angol sentence (mondat) szó kezdőbetűje S jelöli.

P olyan $< \alpha, \beta >$ rendezett pároknak a véges halmaza, melyeknél α és β $V \cup W$-ből alkotott szavak, és α-nak legalább egy betűje nemterminális jel

Definíció. P halmaz elemeit átírási szabályoknak vagy röviden szabályoknak vagy produkcióknak nevezzük, és az (x, y) jelölés helyett $x \to y$ jelöljük.

Definíció. A G grammatika által generált nyelv:

$$L(G) = \{\alpha \in V^* | S \vDash_G \alpha\}$$

ahol $S \vDash_G \alpha$ azt jelöli, hogy S-ből α levezethető ha |-t véges sokszor alkalmazva S-ből α-ba jutunk.

Definíció. Két grammatika ekvivalens, ha $L(G') = L(G'')$

Definíció. G grammatika generálja az α szót, ha $\alpha \in V^*$, és α levezethető S-ből, azaz $S \vDash \alpha$.

1.17.4. Chomsky-féle nyelvosztályok

Az egyes nyelvosztályokban a helyettesítési szabályok alakjára vonatkozóan Chomsky az osztály sorszámának növekedésével egyre szigorúbb megkötéseket írt elő. Minden típusra igaz, hogy akkor hívunk egy nyelvet az adott típusúnak, ha van olyan grammatika, ami az adott nyelvet generálja.

- **0-ás/általános**: Semmilyen megkötést nem teszünk a helyettesítési szabályokra
- **1-es/környezetfüggő**: Ha minden helyettesítési szabályra: $\alpha A \beta \to \alpha \omega \beta$ és $\alpha, \beta, \omega \in (T \cup N)^*, A \in N$ és megengedhető a $S \to \varepsilon$ szabály, ha az S nem szerepel egyetlen szabály jobb oldalán sem
- **2-es/környezetfüggetlen**: Ha minden szabálya $A \to \omega$ és $\omega \in (T \cup N)^*, A \in N$ és megengedhető a $S \to \varepsilon$ szabály, ha az S nem szerepel egyetlen szabály jobb oldalán sem
- **3-mas/reguláris**: Ha minden szabálya $A \to aB$ vagy $A \to a$ és $A, B \in N$ $a \in T$ és megengedhető a $S \to \varepsilon$ szabály, ha az S nem szerepel egyetlen szabály jobb oldalán sem

2. Intel processzorok regiszterkészlete

Az itt felsorolt regiszterek 16 bites regiszterek.

2.1. Általános regiszterek

- **Akkumlátorregiszter**
  - Jelölése: AX
  - Alsó 8 bitje: AL
  - Felső 8 bitje: AH
  - Szerepe van a szorzás, osztás és I/O utasításoknál.
- **Bázisregiszter**
  - Jelölése: BX
  - Alsó 8 bitje: BL
  - Felső 8 bitje: BH
  - Szorzás és osztástól eltekintve minden művelethez használható, általában az adatszegmensben tárolt adatok báziscímét tartalmazza.
- **Számlálóregiszter**
  - Jelölése: CX
  - Alsó 8 bitje: CL
  - Felső 8 bitje: CH
  - Szorzás és osztás kivételével minden művelethez használható, általában ciklus, léptető, forgató és sztring utasítások ciklusszámlálója.
- **Adatregiszter**
  - Jelölése: DX
  - Alsó 8 bitje: DL
  - Felső 8 bitje: DH
  - Minden művelethez használható, de fontos szerepe van a szorzás, osztás és I/O műveletekben.

2.2. Vezérlőregiszterek

- **Forrás cím**
  - Jelölése: SI
  - A forrásadat indexelt címzésére. Szorzás és osztás kivételével minden műveletnél használható.
- **Cél cím**
  - Jelölése: DI

  - A céladat indexelt címzésére. Kitüntetett szerepe van a sztring műveletek végrehajtásában. Az SI regiszterrel együtt valósítható meg az indirekt és indexelt címzés. Szorzás és osztás kivételével minden műveletnél használható.

- **Verem mutató**
  - Jelölése: SP
  - A verembe utolsóként beírt elem címe. A mutató értéke a stack műveleteknek megfelelően automatikusan változik.

- **Bázis mutató**
  - Jelölése: BP
  - A verem indexelt címzéséhez. Használható a stack-szegmens indirekt és indexelt címzésére. Más műveletben nem javasolt a használata.

- **Utasítás mutató**
  - Jelölése: IP
  - A végrehajtandó utasítás címét tartalmazza, mindig a következő utasításra mutat. Az utasítás beolvasása közben az IP az utasítás hosszával automatikusan növekszik.

2.3. Szegmensregiszterek

Ezek a regiszterek tárolják a különböző funkciókhoz használt memória- és szegmenscímeket.

- **Kódszegmens**
  - Jelölése: CS
  - Az utasítások címzéséhez szükséges, az éppen futó programmodul báziscímét tartalmazza. Minden utasításbetöltés használja. Tartalma csak vezérlésátadással módosulhat.

- **Veremszegmens**
  - Jelölése: SS
  - A verem címzéséhez használatos, a stack-ként használt memóriaterület báziscímét tartalmazza.

- **Adatszegmens**
  - Jelölése: DS
  - Az adatterület címzéséhez kell, az adatszegmens bázis címét tartalmazza.

2.4. Státuszregiszter – Flags

A státuszregiszter bitjei az utolsó művelet eredményének megfelelően állnak be. Tartalmuk utasítással módosítható és lekérdezhető. A feltételes vezérlőátadó utasítások a bitek állásától függően működnek. A jelzőbitek (flags) elnevezései:

- **CF**: Például bitléptesnél a regiszterből kikerülő bit értéke

- **PF**: 1, ha az eredmény alsó 8 bitjében lévő egyesek száma páros, egyébként 0
- **ZF**: 1, ha az eredmény nulla, egyébként 0
- **SF**: 0, ha az eredmény pozitív, egyébként 1
- **OF**: 1, ha az eredmény nem fér el a célregiszterben

3. Ugró utasítások

3.1. Feltétel nélküli ugrás

- Jelölése: JMP
- Feltétel nélküli ugrásnál az utasításban szereplő címmel tölti fel a processzor az utasításszámláló regiszter tartalmát, amely a következő utasítás címe lesz és a program innen folytatódik.

3.2. Előjeles feltételes ugró utasítások

Utasítás	**Jelentés**	**Feltétel(ek)**
JG	Jump if greater	ZF = 0 és SF = OF
JGE	Jump if greater equal	SF=OF
JL	Jump if less	SF != OF
JLE	Jump if less equal	ZF = 1 vagy SF != OF
JE	Jump if equal	ZF = 1
JNE	Jump if not equal	ZF = 0

3.3. Előjel nélküli feltételes ugró utasítások

Utasítás	**Jelentés**	**Feltétel(ek)**
JA	Jump if above	CF = 0 és ZF = 0
JAE	Jump if above equal	CF = 0
JB	Jump if below	CF = 1
JBE	Jump if below equal	CF = 1 vagy ZF = 1
JE	Jump if equal	ZF = 1
JNE	Jump if not equal	ZF = 0

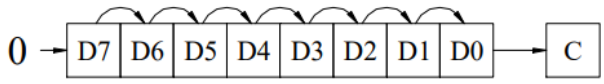
3.4. Flag-eken alapuló ugró utasítások

Utasítás	**Jelentés**	**Feltétel(ek)**
JC	Jump if carry	CF=1
JNC	Jump if no carry	CF=0
JZ	Jump if zero	ZF=1
JNZ	Jump if no zero	ZF = 0
JS	Jump if sign	SF = 1
JNS	Jump if no sign	SF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if no overflow	OF = 0
JP	Jump if parity	PF = 1
JPE	Jump if parity even	PF = 1
JNP	Jump if no parity	PF = 0
JPO	Jump if parity odd	PF = 0

4. Bitmozgató műveletek

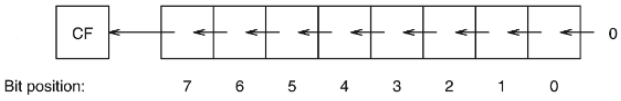
4.1. Léptetések (shift)

- **Logikai shift jobbra**: Minden bit értékét eggyel jobbra mozgatja. A legnagyobb helyi érték 0 lesz, a legkisebb helyi értéken kilépő bit egy erre a célra fentartott egy bites tárolóba, a Carry-be kerül. Assemblyben: `SHR`



- **Aritmetikai shift jobbra**: Minden bit értékét eggyel jobbra mozgatja. A legnagyobb helyi értékre az előjel kerül (vagyis saját maga, hogy a szám előjele ne változzon meg), a legkisebb helyi értéken kilépő bit a Carry-be kerül. Assemblyben: `SAR`

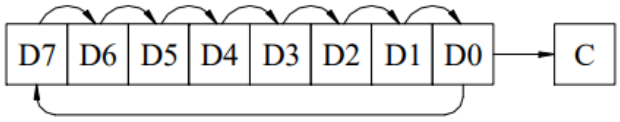
- **Logikai és aritmetikai shift balra**: Minden bit értékét eggyel balra mozgatja. A legkisebb helyi érték 0 lesz, a legnagyobb helyi értéken kilépő bit egy erre a célra fentartott egy bites tárolóba, a Carry-be kerül. Assemblyben: `SHL`, `SAL` (ekvivalensek)



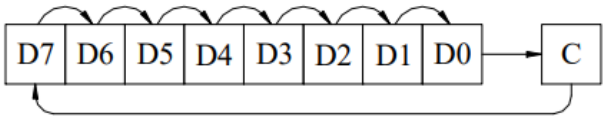
- A fixpontos számoknál a balra léptetés 2-vel szorzásnak, a jobbra léptetés kettővel osztásnak felel meg

4.2. Rotálások (rotate)

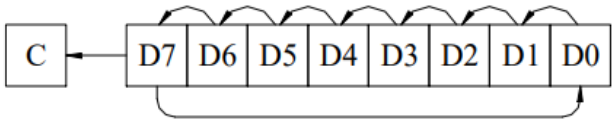
- **Rotálás jobbra**: Minden bit értékét eggyel jobbra mozgatja. A legnagyobb helyi értékre és a Carry-be a legkisebb helyi értéken kilépő bit kerül. Assemblyben: `ROR`



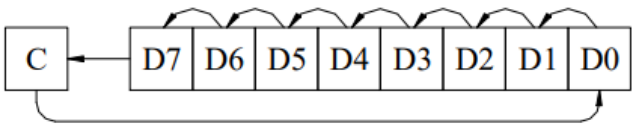
- **Rotálás jobbra Carry-n keresztül**: Minden bit értékét eggyel jobbra mozgatja. A legnagyobb helyi értékre a Carry tartalma, a legkisebb helyi értéken kilépő bit a Carry-be kerül. Assemblyben: `RCR`



- **Rotálás balra**: Minden bit értékét eggyel balra mozgatja. A legkisebb helyi értékre és a Carry-be a legnagyobb helyi értéken kilépő bit kerül. Assemblyben: `ROL`



- **Rotálás balra Carry-n keresztül**: Minden bit értékét eggyel balra mozgatja. A legkisebb helyi értékre a Carry tartalma, a legnagyobb helyi értéken kilépő bit a Carry-be kerül. Assemblyben: `RCL`



5. Az Intel processzorok utasításrendszere

5.1. Az Intel processzorokról

- Minden számítógép rendelkezik operatív tárral, amiben az éppen futó programok és adatok vannak
- Intel CPU-k memóriája bájt szervezésű, minden bájtnyi területnek van címe
- Intel CPU-k növekvő bájtsorrendet (little endian) alkalmaznak, azaz az alacsonyabb helyiérték van az alacsonyabb memóriacímen.
- A memóriában található programok azok bináris számok, amelyeket a CPU sorban egymás után beolvas és végrehajt
- Intel CPU-k egycímesek, ami azt jelenti, hogy egy utasítással csak egy memóriahelyet lehet megcímezni. Azaz, ha egy utasítás két operandusú, akkor csak az egyik lehet memóriacím, a másik regiszter kell, hogy legyen
- A regiszterek méretével jellemezhető egy CPU, azaz lehet 8, 16, 32 stb bites. A nagyobb méret (elméletileg) gyorsabb processzort jelent, de csak egyre nagyobb adatmennyiségeknél igaz ez
- Bármely két szomszédos byte egy 16 bites szót alkot. (1 WORD = 2 bájt)
- A gépi kódú utasítások felépítése:

Prefixum	Operáció kód	Címzési mód	Operandus

1. Prefixum módosítja az utasítás értelmezését, pl. címkék a ciklusokhoz (opcionális)
2. Az operáció kód adja meg, hogy a processzornak milyen műveletet kell végrehajtani. Minden utasításban szerepelnie kell!
3. A címzési mód az operandusok – a későbbiekben részletesen tárgyalt – értelmezését adja meg. (Csak operandust tartalmazó utasításokban található.)
4. Az operandus lehet konstans, cím vagy címzéshez használt érték. Hossza változó, de el is maradhat.

5.2. Operandusok és címzési módok

- **Utasítások**: Az assembly utasítások pontosan megfelelnek egy gépi kódú utasításnak, azaz a fordító program (assembler) minden utasítást egy gépi kódra fordít le.
- **Operandusok**: Lehetnek konstansok, memóriacímek vagy regiszterek
- **Regiszteroperandus**: Az utasítás paraméterei regiszterek, amelyekben tároljuk a művelet elvégzéséhez szükséges adatokat. Az operandusokat a regiszterek nevével adjuk meg. Nem igényel memória hozzáférést, leggyorsabb.

```
MOV AH, AL ;AL tartalmát AH-ba másolja (8 bites adatmozgatás)
```

- **Közvetlen (immediate) operandus**: Egyik operandus egy regiszter, a másik egy konstans érték.

```
MOV AX, 2 ; AX-be kettőt teszi
```

- **Direkt memóriacímzés**: (Memóriából -> regiszterbe) Ebben az esetben az operandus egy előre (a DATA szegmensben) deklarált adat. Ennek a memóriacíme szerepel az utasításban. Az adat szegmenscímének (alapértelmezésben) a DS-ben kell lennie, ezért ezt használat előtt be kell állítani:

```
.DATA
    Adat DB 10h, 11h, 12h, 13h    ;DB jelentése: az adategységünk bájtos
.CODE
    MOV AX, DGROUP      ;Adatszegmens címe AX-be.
    ; A DGROUP a program betöltésekor kap értéket az operációs rendszertől.
    MOV DS, AX          ;Adatszegmens címe a DS-be
    MOV AX, Adat        ;AX-be tölti az Adat címen lévő értéket (10h)
    MOV AX, [Adat]      ;Ua. mint az előző
    MOV AX, Adat[2]     ;AX-be tölti az Adat+2 címen lévő értéket (12h)
    MOV AX, Adat+2      ;Ua. mint az előző
```

- **Indirekt memóriacímzés**: Itt az operandussal megadott helyen nem az adatot, hanem annak címét találjuk. Az indirekt memóriacímzés során egy regiszterben vagy egy memóriaterületen tárolt címet használunk a memória elérésére.

- **Regiszter indirekt címzés**: A regiszter indirekt memóriacímzés azt jelenti, hogy a programban egy regiszter tartalmát használjuk a memóriacím megadására. A regiszter értéke lesz az indirekt cím, amelyen keresztül elérhetjük a memóriában tárolt adatokat. Az alapértelmezett adatszegmens címe DS. Más szegmensre történő hivatkozás:.

```
    MOV AX, [BX] ;AX-be tölti a BX által megcímzett memória tartalmát
    MOV AX, ES:[BX] ;AX-be tölti az ES:BX által megcímzett memória
↪   tartalmát
```

- **Indexelt (bázis relatív) címzés**: A műveleti kód után található egy eltolási érték (offset), ehhez hozzáadva a bázis regiszterben található kezdőcímet megkapjuk az operandus címét

```
.DATA
 adat DB 10h,20h
.CODE
 MOV AX,adat[BX] ;AX-be tölti az adat+BX címen lévő adatot
```

- **Bázis+index címzés**: A bázis+index címzés egy memóriacímzési mód, amelyben egy bázisregiszter értékéhez hozzáadunk egy indexregiszter értékét, hogy meghatározzuk a kívánt memóriacímet.

```
MOV AX,[BX][DI] ;AX-be tölti a BX+DI címen lévő adatot
```

- **Bázis+relatív címzés**: Ez a megoldás használható többdimenziós tömbök kezelésére.

```
MOV AX,adat[BX][DI] ;AX-be tölti az adat+BX+DI címen lévő adatot
```

5.3. Verem (stack)

- A stack (verem) egy olyan - az operációs rendszer által kijelölt memóriaterület - ahova közvetlen utasításokkal lehet adatot elmenteni, és onnan visszaolvasni
- LIFO elven működik, azaz amit utoljára betettünk, azt vehetjük ki elsőként
- Veremben átmenetileg tárolhatjuk az adatokat
- A CALL szubrutin hívás elmenti a stack tetejére a következő (CALL utáni) utasítás címét. Ezt a címet tölti be az IP-be (utasításpointerbe) a RET utasítás.
- A processzor regiszterkészletének ismertetésénél láttuk, hogy a verem tetején tárolt (aktuális) adat teljes címe az SS:SP regiszterpárban található.
- Assembly nyelven a PUSH forrás utasítást használjuk adattárolásra, és a POP cél utasítást az adatok visszatöltésére a veremből
- Amikor betesztünk egy adatot a verembe, a veremmutató értéke 2-vel nő, amikor kiveszünk 2-vel csökken, a veremmutató - utoljára betett adat memóriacímére mutat
- A PUSH utasítás az SP (veremmutató) értékét kettővel csökkenti, majd az operandusban (forrásban) tárolt kétbájtos adatot elhelyezi a verem tetején
- A POP utasítás kiveszi a verem tetején lévő (kétbájtos adatot), amit a cél-lal megadott címre tölt. Ezután az SP értékét kettővel növeli.
- Az Intel processzorok esetében, a verembe mindig szavas (16 bites) számokat tudunk eltárolni. Mivel ez 2 bájtnyi adatot jelent, így a veremmutató értéke kettővel változik

5.4. Memóriaszervezés

- CPU és az operatív memória külön helyezkedik el
- Közöttük a kapcsolatot a busz biztosítja
- A címbuszon n db memóriaelem található, így 2^n db memóriaelem címezhető meg
- A memória lineárisan viselkedik, folyamatos elérést biztosít 0-tól $2^n - 1$ címig
- A program betöltésekor döntődik el, hogy a program és az adatok ténylegesen hová kerülnek
- Korai Intel processzorok szegmentált memóriacímzést használnak, azt jelentette, hogy a memória címzésére szolgáló regiszterek 16 bit hosszúak, ezekkel maximum 64 kilobájt memória megcímzése volt lehetséges
- Ennek kiküszöbölésére felhasználtak a címzéshez szegmensregisztereket, így a címek két részből álltak: egy szegmens- és báziscímből
- Egy szegmens mérete 64 kilobájt, és egy szegmensen belül bármilyen adatot elérhetünk

6. Assembly programozás

6.1. Szegmentálás

- Olyan parancsok, amelyek vagy a fordítást, vagy pedig a futtatást befolyásolják. A direktívákat és az operandusaikban lévő szimbólumokat is a lexikális elemző határozza meg, ez általában előfeldolgozó modul segítségével történik.
- DOSSEG direktíva jelzi a fordítónak, hogy a programszegmenseket egy szigorúan meghatározott rendben akarjuk betölteni
- MODEL direktívával megadjuk, hogy az eljárás hívások közeli (NEAR) vagy távoli (FAR) jellegűek
- STACK direktívát kötelező megadni .EXE fájlok esetében
- .CODE direktíva a kódszegmens kezdetét jelzi
- .DATA direktívára akkor van szükség, ha adatszegmenst akarunk használni, pl. globális változót
- END direktíva jelzi a program végét

6.2. Konstansok használata

- A fordítóprogram alapértelmezett számrendszer a decimális (10-es)
- A számrendszer alapszámát (radix) mindig a szám után írt betűvel jelezzük
  - Y vagy B - bináris szám
  - O vagy Q - oktális szám
  - T vagy D - decimális szám
  - H - hexadecimális szám - hogy a változókkal ne keverjük össze, a szám elé „vezető 0” kerül, pl.: 0FFh
- Pl. ha a radix értéke 10, akkor a számok végéről elhagyható a `D` betű
- A karakter-konstansok megadásánál használhatjuk a karakter ASCII kódját, de magát a karaktert is pl.: `MOV AL, "A"` vagy `MOV AL, 65`

6.3. Karakter bekérés és kiíratás

DOS megszakítások segítségével tudunk karaktert beolvasni vagy írni a standard input/output-ra. Az INT utasítás segítségével lehet megszakítási kérést küldeni a processzornak.

- **Karakter beolvasása a standard input-ról echo-val** (AH=1)
  - Várakozik a felhasználói input-ra
  - echo azt jelenti, hogy a karakter megadása után megjelenik a képernyőn
  - Visszatérés: AL = beolvasott karakter
- **Karakter kiíratása a standard output-ra echo-val** (AH=2)

  - Fontos, ilyenkor az AL regiszter értékét átmozgatni a DL regiszterbe
  - Bemenet: DL regiszter értéke
  - Visszatérés: AL = utolsó kiíratott karakter

- **Exit – kilépés visszatérési kóddal** (AH=4Ch)
- **Karakter beolvasása a standard input-ról echo nélkül** (AH = 8)

```
.MODEL SMALL     ; A kód 64 kB-nál kisebb

.STACK           ; Veremszegmens

.CODE            ; Kódszegmens

MOV AH, 1        ; AH=1 akkor stdin
INT 21h          ; Karakter bekérése és elmentése AL-be

MOV DL, AL       ; AL-ben lévő karakter áthelyezése DL-be
MOV AH, 2        ; AH=2 akkor stdout
INT 21h          ; Karakter kiírása a képernyőre DL-ből

MOV AH, 4Ch      ; Kilépés, terminálás visszatérési kóddal
INT 21h          ; AL-ben lévő értékkel tér vissza a program

END
```

6.4. Szubrutinok használata

A szubrutinokat akkor szoktuk használni, amikor szeretnénk hogy a gyakran használt kódrészleteket ne kelljen újra-újra megírni. Az alábbiakban összefoglalom a legfontosabb tudnivalókat az alprogramokról (szubrutin):

- **Létrehozás**: név PROC ... tartalom ... név ENDP
- **Meghívás**: CALL szubrutin neve
- **Visszatérés**: A szubrutin végén vissza kell adni a vezérlést a hívónak a RET kulcsszó segítségével
- **Veremhasználat**: A regiszterekben (AX, BX, ...) lévő értékek megőrzése érdekében a verembe kell helyezni őket (PUSH) a szubrutin lefutásakor, majd a végén visszarakni (POP).

Az EQU kulcsszó segítségével lehet egy változóhoz, címkéhez konstans értéket rendelni. A sor elejére kódja 13, az új sor kódja 10

```
.MODEL SMALL
.STACK

.CODE

    ; Karakter beolvasása stdinput-ról
read_char PROC
                PUSH AX             ; AX elmentése a verembe, mert itt használjuk
↪   ne írjuk felül
                MOV  AH, 1          ; AH=1 stdin
                INT  21h            ; Karakter bekérése és elmentése AL-be
                MOV  DL, AL         ; AL-ből DL-be rakjuk a karaktert
                POP  AX             ; AX visszatöltése a veremből
                RET                 ; Visszatérés a hívóhoz
read_char ENDP

    ; Karakter kiíratása az stdout-al DL-ből olvassa ki
write_char PROC
                PUSH AX
                MOV  AH, 2          ; AH=2 stdout
                INT  21h            ; Karakter kiíratása a képernyőre
                POP  AX
                RET
write_char ENDP

    CR          EQU  13             ; konstans deklarálás: név EQU érték
    LF          EQU  10

    ; Új sor kiíratása
cr_lf PROC
                PUSH DX
                MOV  DL, CR         ; Carriage return
                CALL write_char
                MOV  DL, LF         ; Line feed
                CALL write_char
                POP  DX
                RET
cr_lf ENDP

    ; A program kezdőpontja
main PROC                           ; main eljárás kezdete
                CALL read_char      ; read_char szubrutin meghívása
                CALL cr_lf
                CALL write_char     ; write_char szubrutin meghívása

                MOV  AH, 4Ch
                INT  21h

main ENDP                           ; main eljárás vége

END main
```

6.5. Karakterkód konvertálása bináris számmá és kiírása a képernyőre

Az alábbi kódban láthajuk hogyan lehet egy beolvasott karaktert kiírni bináris számmal. Azt használjuk ki, hogy bitforgató művelettel lehet a számjegyeket megkapni, hiszen balra forgatáskor (rotálás balra carry-n keresztül) a carry-be kerül az adott számjegy és azt `ADC` (add with carry-vel) 0-hoz adjuk, ami 1 vagy 0 lesz. Ezt 8-szor végezzük el, mert `CX` értéke 8-ra lett beállítva. `binary_digit` az egy címke, amire lehet hivatkozni. Fontosabb észrevételek:

- Ha a `XOR` műveletnek mindkét operandusza ugyanaz a regiszter, akkor lenullázhatjuk a tartalmát a regiszternek, lehetne `MOV` utasítással is elvégezni, de ez így 2 bájtnyi helyet foglal nem 3-mat.
- `ADC` utasítás a második operanduszt hozzáadja az első operanduszhoz, majd a (`CF`) carry flag értékét az első operanduszhoz hozzáadja
- `LOOP` utasítással lehet ciklust készíteni, meg kell adni neki a címkét, addig fut a ciklus, amíg a `CX` regiszter értéke nem nulla, iterációként 1-gyel csökkenti a `CX` regiszter értékét a LOOP
- `RCL` (Rotate with Carry Left), az első operandusz ahol a biteket forgatjuk, a második operandusz adja meg hogy hányszor forgatunk. A legszignifikánsabb (legbaloldalibb) bit kerül a `CF`-be, a `CF`-ben lévő bit pedig a legjobboldalibb helyre kerül.

```
.MODEL SMALL
.STACK
    ; Bitforgatás balra carry-vel
    ; a - 11000010 forgatás után
    ; a - 01100001 forgatás előtt
.CODE
    ; Program kiindulási része
main PROC
                CALL read_char
                CALL cr_lf
                CALL write_binary

                MOV  AH, 4Ch
                INT  21h
main ENDP

    ; A regiszter bitjeinek kiírása balról jobbra
write_binary PROC
                PUSH BX
                PUSH CX
                PUSH DX

                MOV  BL, DL          ; beolvasott karakter elmentése BL-be
                MOV  CX, 8           ; 8-szor írunk ki számjegyet

    binary_digit:
                XOR  DL, DL          ; DL nullázása
                RCL  BL, 1           ; BL elforgatása balra 1-gyel CF-be kerül az eredmény
                ADC  DL, "0"        ; 0 + 0 = 1 VAGY 0 + 1 = 1, add with carry
                CALL write_char
                LOOP binary_digit    ; amíg CX nem nulla addig csökkenti a tartalmát
                POP  DX
                POP  CX
                POP  BX
                RET
write_binary ENDP

END MAIN
```

6.6. Karakterkód konvertálása hexadecimális számmá és kiírása a képernyőre

Egy hexadecimális számjegy 4 bites bináris számmal írható le, tehát egy bájton 2 hexadecimális számjegy fér el. Válasszuk ketté a 8-bites `DL` regiszter tartalmát, alsó 4 bit és felső 4 bitre. Külön-külön végezzük el a konvertálást a `write_hexa_digit` alprogrammal.

Először a felső 4 bitre van szükségünk, a `DL` tartalmát 4 bittel jobbra shifteljük, így a felső 4 bit jobbra csúszik és helyükbe 0 kerül.

A második konvertálásnak a szám jó helyen van az alsó 4 biten, de a felső négy bitet törölni kell. Amikor `write_hexa_digit` alprogramot használjuk meg kell vizsgálni, hogy 10-nél nagyobb egyenlő vagy kisebb-e a szám. Itt is összegyűjtöttem a fontosabb tudnivalókat:

- `SHR` utasítás az első operendusz tartalmát shifteli jobbra annyiszor ahányszor megadjuk a második operanduszban
- `CMP` utasítás összehasonlítja a két operandusz tartalmát, ha a két operandusz megegyezik akkor a (`ZF`) zero flag értéke 1, különben 0. Ha az 1. operandusz kisebb, mint 2. operandusz akkor (`CF`) carry flag értéke 1, különben 0.
- `AND` utasítással lehet két operandusz között „bitenkénti és" műveletet végrehajtani, a felső 8 bit törléséhez olyan bináris számot kell megadni, aminek az első 8 helyén 0 van, mert így az és művelet után törlődik a felső 8 bit.
- `JB` (Jump Below) utasítás akkor ugrik a megadott címkére, ha a `CF` értéke 1, azaz 1. operandusz kisebb mint a 2. operandusz.
- `ADD` utasítás az 1. operandusz értékéhez hozzáadja a 2. operandusz értékét.

```
.MODEL SMALL
.STACK

.CODE

main PROC
                    CALL read_char
                    CALL cr_lf
                    CALL write_hexa
                    MOV  AH, 4Ch
                    INT  21h
main ENDP

    ; Karakter kiíratása hexadecimális számként
write_hexa PROC
                    PUSH CX
                    PUSH DX
                    MOV  DH, DL             ; DL elmentése
                    MOV  CL, 4              ; Shiftelések száma
                    SHR  DL, CL             ; DL shift-elése 4 hellyel jobbra
                    CALL write_hexa_digit
                    MOV  DL, DH             ; Elmentett DL visszahelyezése
                    AND  DL, 0Fh            ; Felső 4 bit törlése
                    CALL write_hexa_digit
                    POP  DX
                    POP  CX
                    RET
write_hexa ENDP

write_hexa_digit PROC
                    PUSH DX
                    CMP  DL, 10             ; DL összehasonlítása 10-zel
                    JB   non_hexa_letter    ; Ugrás, ha kisebb mint 10
                    ADD  DL, "A"-"0"-10     ; A-F betű kiírása
    non_hexa_letter:
                    ADD  DL, "0"            ; ASCII kód megadása
                    CALL write_char
                    POP  DX
                    RET
write_hexa_digit ENDP

END main
```

6.7. Karakterkód konvertálása decimális számmá és kiírása a képernyőre

A számot elosztjuk 10-zel, majd az eredménnyel addig végezzük az osztást, amíg 0 nem lesz. A maradékokat tároljuk, amelyek megadják a tízes számrendszerbeli számjegyeket a legkisebb helyiértéktől kezdve.

Az eljárásban verembe gyűjtjük a maradékokat, majd ciklus segítségével visszaolvassuk őket fordított sorrendben. Az osztások számlálásához a `CX` regisztert használjuk, így a visszaolvasáskor nem kell külön törődnünk a ciklusváltozó kezdőértékével. Fontosabb utasítások, amik előkerülnek itt:

- `OR`: bitenkénti vagy művelet elvégzése a két operanduszon
- `JNE`: (Jump if Not Equal) Akkor ugrik a megadott címkére ez az utasítás, ha a `ZF` értéke 0.
- `XOR`: Kizáró vagy, akkor igaz ha igaz vagy hamis, egyébként hamis

```
.MODEL SMALL
.STACK

.CODE

main PROC
                    CALL read_char
                    CALL cr_lf
                    CALL write_decimal
                    MOV  AH, 4Ch
                    INT  21h
main ENDP

write_decimal proc
                    push AX
                    push CX
                    push DX
                    push SI
                    XOR  DH, DH             ; Felső 8 bit törlése, csak az
↪  alsó 8 van meg stdin-ból
                    mov  AX, DX             ; osztandó szám AX-be
                    mov  SI, 10             ; osztó
                    XOR  CX, CX             ; CX = 0
    decimal_non_zero:
                    XOR  DX, DX             ; DX = 0
                    DIV  SI                 ; AX osztása SI-vel, eredmény
↪  AX-be, maradék DX-be kerül
                    push DX                 ; maradék mentése verembe
                    inc  CX                 ; osztások száma + 1
                    OR   AX, AX             ; státuszbitek beállítása AX-nek
↪  megfelelően
                    JNE  decimal_non_zero   ; Ugrás, ha nem nulla
    decimal_loop:
                    pop  DX                 ; maradék kivétele veremből
                    ADD  DL, "0"            ; ASCII kód megadása
                    CALL write_char         ; kiíratás
                    loop decimal_loop       ; ciklus
                    pop  SI
                    pop  DX
                    pop  CX
                    pop  AX
                    ret

write_decimal endp

END MAIN
```

6.8. Decimális szám beolvasó rutin

Készítünk egy olyan alprogramot, amely segítségével lehet beolvasni tízes számrendszerbeli számot. A szubrutin működésének a lépései: beolvasunk a billentyűzetről karaktereket, először magasabb majd egyre alacsonyabb helyi értékűeket az ENTER-ig.

Ha a karakterekből kivonunk 48-at "0" kódját, akkor a megfelelő számértéket kapjuk meg. Ezután a számokat helyi értéküknek megfelelően kell eltárolni. Vesszük a legmagasabb helyen lévő számjegyet, amit eggyel magasabb helyi értékre tolunk (megszorozzuk a számrendszer alapszámával), majd hozzáadjuk a következő értéket. Ezt addig folytatjuk, amíg van alacsonyabb helyi értékű számjegy. Fontosabb utasítások:

- `JE` (Jump Equal): Akkor ugrik, ha ZF = 1
- `MUL`: Szorzás, az AX regiszterben lévő értéket megszorozza a megadott értékkel
- `JMP` ugrás a címkére

```
.MODEL SMALL
.STACK

.CODE

main PROC
                    CALL read_decimal
                    CALL write_decimal
                    MOV  AH, 4Ch
                    INT  21h
main ENDP

read_decimal PROC
                    PUSH AX
                    PUSH BX
                    MOV  BL, 10
                    XOR  AX, AX

    read_decimal_new:
                    CALL read_char
                    CMP  DL, 13             ; input karakter = ENTER
                    JE   read_decimal_end   ; ugrik, ha enter a karakter
                    SUB  DL, "0"            ; karakter - '0'
                    MUL  BL                 ; AX = AX * 10
                    ADD  AL, DL             ; Következő helyi érték
↪   hozzáadása
                    JMP  read_decimal_new   ; Következő karakter beolvasása

    read_decimal_end:
                    MOV  DL, AL
                    POP  BX
                    POP  AX
                    RET
read_decimal ENDP

END MAIN
```

6.9. Hexadecimális szám beolvasó rutin

Itt az alapszám 16 (10h) lesz. A konvertálásnál figyelembe kell venni, hogy a betűkarakterek ASCII kódja nem közvetlenül a számoké után következik, ezért külön kell választanunk a számjegy és betű karaktereket. Számjegy esetén ugyanúgy működik, mint decimális esetben, betű esetén a kódból le kell vonni "0"-át és még 7-et. Fontosabb utasítások:

- `JBE` (Jump Below or Equal): Akkor ugrik a megadott címkére, ha CF vagy ZF értéke 1.
- `JB` (Jump Below)
- `JA` (Jump Above)
- `SUB`: Kivonás

```
.MODEL SMALL
.STACK

.CODE

main PROC
                        CALL read_hexa
                        CALL cr_lf
                        CALL write_hexa
                        MOV  AH, 4Ch
                        INT  21h
main ENDP

read_hexa proc
                        PUSH AX                     ;AX mentése a verembe
                        PUSH BX                     ;BX mentése a verembe
                        MOV  BX, 10h                ;BX-be a számrendszer alapszáma, ezzel
↪   szorzunk
                        XOR  AX, AX                 ;AX törlése
    read_hexa_new:
                        CALL read_char              ;Egy karakter beolvasása
                        CMP  DL, 13                 ;ENTER ellenőrzése
                        JE   read_hexa_end          ;Vége, ha ENTER volt az utolsó
↪   karakter
                        CALL upcase                 ;Kisbetű átalakítása naggyá
                        SUB  DL, "0"                ;Karakterkód minusz ”0” kódja
                        CMP  DL, 9                  ;Számjegy karakter?
                        JBE  read_hexa_decimal      ;Ugrás, ha decimális számjegy
                        SUB  DL,7                   ;Betű esetén még 7-et levonunk
    read_hexa_decimal:
                        MUL  BL                     ;AX szorzása az alappal
                        ADD  AL, DL                 ;A következő helyi érték hozzáadása
                        JMP  read_hexa_new          ;A következő karakter beolvasása
    read_hexa_end:
                        MOV  DL, AL                 ;DL-be a beírt szám
                        POP  BX                     ;BX visszaállítása
                        POP  AX                     ;AX visszaállítása
                        RET                         ;Visszatérés a hívó rutinba
read_hexa endp

upcase proc                                         ;DL-ben lévő kisbetű átalakítása
↪   nagybetűvé
                        CMP  DL, "a"                ;A karakterkód és ”a” kódjának
↪   összehasonlítása
                        JB   upcase_end             ;A kód kisebb, mint ”a”, nem kisbetű
                        CMP  DL, "z"                ;A karakterkód és ”z” kódjának
↪   összehasonlítása
                        JA   upcase_end             ;A kód nagyobb, mint ”z”, nem kisbetű
                        SUB  DL, "a"-"A"            ;DL-ből a kódok különbségét
    upcase_end:
                        RET                         ;Visszatérés a hívó rutinba
upcase endp

END MAIN
```

6.10. Bináris szám beolvasó rutin

Már korábban láttuk, hogy a bináris számok kettővel (alapszámmal) való szorzása úgy is elvégezhető, hogy eggyel balra léptetünk (shiftelünk)

```
.MODEL SMALL
.STACK
.CODE
main PROC
                    CALL read_binary
                    CALL cr_lf
                    CALL write_binary
                    MOV  AH, 4Ch
                    INT  21h
main ENDP
read_binary proc
                    PUSH AX                 ;AX mentése a verembe
                    XOR  AX, AX             ;AX törlése
    read_binary_new:
                    CALL read_char          ;Egy karakter beolvasása
                    CMP  DL, 13             ;ENTER ellenőrzése
                    JE   read_binary_end    ;Vége, ha ENTER volt az utolsó
↪   karakter
                    SUB  DL, "0"            ;Karakterkód minusz ”0” kódja
                    SAL  AL, 1              ;Szorzás 2-vel, shift eggyel balra
                    ADD  AL, DL             ;A következő helyi érték
↪   hozzáadása
                    JMP  read_binary_new    ;A következő karakter beolvasása
    read_binary_end:
                    MOV  DL, AL             ;DL-be a beírt szám
                    POP  AX                 ;AX visszaállítása
                    RET                     ;Visszatérés a hívó rutinba
read_binary endp
write_binary PROC
                    PUSH BX
                    PUSH CX
                    PUSH DX

                    MOV  BL, DL             ; beolvasott karakter elmentése
↪   BL-be
                    MOV  CX, 8              ; 8-szor írunk ki számjegyet

    binary_digit:
                    XOR  DL, DL             ; DL nullázása
                    RCL  BL, 1              ; BL elforgatása balra 1-gyel CF-be
↪   kerül az eredmény
                    ADC  DL, "0"            ; 0 + 0 = 1 VAGY 0 + 1 = 1, add
↪   with carry
                    CALL write_char
                    LOOP binary_digit       ; amíg CX nem nulla addig csökkenti
↪   a tartalmát
                    POP  DX
                    POP  CX
                    POP  BX
                    RET
write_binary ENDP
END MAIN
```

6.11. Osztás szubrutin

1. Írjuk fel a 0. sorba az elvégzendő műveletet, az osztandót (A), az osztót (B), a hányadost (H) és a maradékot (M).
2. Léptessük A-t és M-et balra úgy, hogy az A-ból kilépő bit az M-be kerüljön. Ezzel együtt léptessük H-t is eggyel balra.
3. A 2. pontban leírt lépést addig végezzük, amíg az M $\geqslant$ B nem lesz. (*-gal jelölt sor). Ekkor módosítsuk az M értékét M-B-re és a H értékét H+1-re.
4. Folytassuk a 2. és 3. pontban leírtakat annyiszor, amilyen hosszú az osztandó. Esetünkben 8-szor. (A balra mozgatások számát a sorok elején látható szám mutatja.)

```
.MODEL SMALL
.STACK

.CODE

main PROC
                    CALL read_decimal       ; osztandó beolvasása
                    MOV  AL, DL
                    CALL cr_lf
                    CALL read_decimal       ; osztó beolvasása
                    MOV  BL, DL
                    CALL cr_lf
                    XOR  DX, DX             ; DX törlése
                    MOV  CX, 8              ; Ciklusszám

    Cycle:
                    SHL  AL, 1              ; Osztandó eggyel balra, CR-be a
↪   kilépő bit
                    RCL  DH, 1              ; Maradék eggyel balra, belép a
↪   CR tartalma
                    SHL  DL, 1              ;Hányados eggyel balra
                    CMP  DH, BL
                    JB   Next               ; A maradék kiseb,mint az osztó
                    SUB  DH, BL             ; Az osztó kivonása a maradékból
                    INC  DL                 ;Hányados növelése
    Next:
                    LOOP Cycle
    Stop:
                    CALL write_decimal      ; Hányados (DL) kiírása
                    CALL cr_lf
                    MOV  DL, DH
                    CALL write_decimal      ; Maradék (CL) kiírása
                    MOV  AH, 4CH
                    INT  21H
main ENDP
END MAIN
```

6.12. Adatszegmens használata

Mivel az adatszegmens helye csak betöltés után vállik véglegessé, ezért szükség van egy `DGROUP` mutatóra, ahová az operációs rendszer beírja az adatszegmens címét. Ezt az adatszegmens használatakor mindig be kell tölteni a `DS` regiszterbe. Mivel a memóriából közvetlenül nem tölthetünk be adatot a `DS`-be, így ezt két lépésben kell megtennünk: először `AX`-be tesszük a `DGROUP`-un lévő értékét, majd ezt írjuk a `DS`-be.

```
.MODEL SMALL
.STACK

.DATA

    adat DB 65     ; A betű ASCII kódja

.CODE

main PROC
                MOV  AX, DGROUP     ; Adatszegmens helyének lekérdezése
                MOV  DS, AX         ; DS mutasson adatszegmensre
                MOV  DL, adat
                CALL write_char
                MOV  AH, 4CH
                INT  21H
main ENDP

END MAIN
```

6.13. Adatszegmens és indirekt címzés

Itt a `BX` regiszteren keresztül címeztük meg az adatot (indirekt címzés). Ezt a megoldást akkor célszerű alkalmazni, amikor egy változó névhez nemcsak egy érték kapcsolódik, például tömbök vagy sztringek használatakor.

```
.MODEL SMALL
.STACK

.DATA

    adat DB 65     ; A betű ASCII kódja

.CODE

main proc
                MOV  AX, DGROUP     ;Adatszegmens helyének lekérdezése
                MOV  DS, AX         ;DS beállítása, hogy az adatszegmensre
 ↪  mutasson
                LEA  BX, adat       ;Az adat offset címének betöltése BX-be
                MOV  DL, [BX]       ;DL-be tölti a BX-el címzett memória
 ↪  tartalmát
                CALL write_char     ;Karakter kiírása
                MOV  AH,4Ch         ;Visszatérés az operációs rendszerbe
                INT  21h
main endp

END MAIN
```

6.14. Szövegsor kiíratása a képernyőre

```
.MODEL SMALL
.STACK

.DATA

    adat DB "Ez egy pelda szoveg",0

.CODE

main proc
                MOV  AX, DGROUP     ;Adatszegmens helyének lekérdezése
                MOV  DS, AX         ;DS beállítása, hogy az adatszegmensre
↪   mutasson
                LEA  BX, adat       ;Az adat offset címének betöltése BX-be
    new:
                MOV  DL, [BX]       ;DL-be egy karakter betöltése
                OR   DL, DL         ;Státuszbitek beállítása DL-nek megfelelően
                JZ   stop           ;Kilépés a ciklusból, ha DL=0
                CALL WRITE_CHAR     ;Egy karakter kiírása
                INC  BX             ;BX növelése, BX a következő karakterre
↪   mutat
                JMP  new            ;Vissza a ciklus elejére
    stop:
                MOV  AH,4Ch         ;Kilépés
                INT  21h
main endp

END MAIN
```

6.15. write_string szubrutin

```
.MODEL SMALL
.STACK
.DATA
    adat_1 DB "Elso sor",0
    adat_2 DB "Masodik sor",0
.CODE

main PROC
                    MOV  AX, DGROUP
                    MOV  DS, AX
                    LEA  BX, adat_1
                    CALL write_string
                    LEA  BX, adat_2
                    CALL write_string
                    MOV  AH, 4CH
                    INT  21H
main ENDP

write_string PROC
                    PUSH DX
                    PUSH BX
    write_str_new:
                    MOV  DL, [BX]
                    OR   DL, DL
                    JZ   write_str_end
                    CALL write_char
                    INC  BX
                    JMP  write_str_new
    write_str_end:
                    POP  BX
                    POP  DX
                    RET
write_string ENDP

END MAIN
```

6.16. DUP operátor és a kérdőjel

Az n DUP (val) operátor segítségével n darab val értéket helyezünk el az adatszegmensre. A .DATA utáni ? azt jelenti, hogy olyan változókat akarunk definiálni, amelyeknek nincs kezdeti értéke. Ezért nem is kell, hogy a programállományban (pl. az EXE fájlban) helyet foglaljon. Az adatterületre csak a program betöltése után van szükségünk. Ha a .DATA? után valamilyen konkrét értéket (pl. adat DB "A") definiálunk, akkor az assembler az összes változónak helyet foglal az EXE állományban. Ezért a kezdeti értékkel rendelkező változókat a .DATA, a többit pedig a .DATA? részben kell definiálni DUP (?) segítségével.

```
.MODEL SMALL
.STACK
.DATA?
    txt  DB 100 DUP (?)
.CODE

main PROC
                    MOV  AX, DGROUP
                    MOV  DS, AX
                    LEA  BX, txt
                    CALL read_string
                    CALL cr_lf
                    CALL write_string
                    MOV  AH, 4Ch
                    INT  21h
main ENDP

read_string PROC
                    PUSH DX
                    PUSH BX
    read_string_new:
                    CALL read_char
                    CMP  DL, 13
                    JE   read_string_end
                    MOV  [BX], DL
                    INC  BX
                    JMP  read_string_new
    read_string_end:
                    XOR  DL, DL
                    MOV  [BX], DL
                    POP  BX
                    POP  DX
                    RET
read_string ENDP

write_string PROC
                    PUSH DX
                    PUSH BX
    write_str_new:
                    MOV  DL, [BX]
                    OR   DL, DL
                    JZ   write_str_end
                    CALL write_char
                    INC  BX
                    JMP  write_str_new
    write_str_end:
                    POP  BX
                    POP  DX
                    RET
write_string ENDP

END MAIN
```

6.17. Karakterek kiírása a-z-ig

```
.MODEL SMALL
.STACK
.CODE

main proc

                mov  cx, 26
                mov  dx, 65
    innerLoop:
                mov  ah, 2
                int  21h

                inc  dx
                loop innerLoop

                mov  ah, 4ch
                int  21h

main endp

END MAIN
```

6.18. Három decimális szám szorzása

```
.MODEL SMALL
.STACK

.DATA
    num1            DW  ?
    num2            DW  ?
    num3            DW  ?
.CODE
MAIN PROC
                    MOV  AX, @DATA
                    MOV  DS, AX

                    XOR  DH, DH

                    CALL read_decimal
                    MOV  num1, DX

                    CALL read_decimal
                    MOV  num2, DX

                    CALL read_decimal
                    MOV  num3, DX

                    XOR  AX, AX
                    MOV  AX, num1
                    MUL  num2
                    MUL  num3
                    MOV  DX, AX
                    CALL write_decimal

                    MOV  AH, 4Ch
                    INT  21h

MAIN ENDP

END MAIN
```

6.19. Bináris szám beolvasó, csak 0 és 1

```
.MODEL SMALL
.STACK
.CODE

main PROC
                        CALL read_binary
                        CALL cr_lf
                        CALL write_binary
                        MOV  AH, 4Ch
                        INT  21h
main ENDP

read_char PROC
                        PUSH AX
                        MOV  AH, 8                  ; Stdin without echo
                        INT  21h
                        MOV  DL, AL
                        POP  AX
                        RET
read_char ENDP

read_binary proc
                        PUSH AX                     ;AX mentése a verembe
                        XOR  AX, AX                 ;AX törlése
    read_binary_new:
                        CALL read_char              ;Egy karakter beolvasása
                        CMP  DL,'0'                 ; Összehasonlítom
 ↪  nullával
                        JE   read_binary_continue   ; ha 0, mehet tovább
                        CMP  DL,'1'                 ; -,,- eggyel
                        JE   read_binary_continue   ; tovább
                        CMP  DL,13                  ; Enter is lehet
                        JE   read_binary_continue   ; tovább
                        JMP  read_binary_new        ; ha nem ment tovább,
 ↪  újra próbáljuk

    read_binary_continue:
                        CALL write_char             ; Csak ha helyes akkor
 ↪  írja ki
                        CMP  DL, 13                 ;ENTER ellenőrzése
                        JE   read_binary_end        ;Vége, ha ENTER volt az
 ↪  utolsó karakter
                        SUB  DL, "0"                ;Karakterkód minusz ”0”
 ↪  kódja
                        SAL  AL, 1                  ;Szorzás 2-vel, shift
 ↪  eggyel balra
                        ADD  AL, DL                 ;A következő helyi érték
 ↪  hozzáadása
                        JMP  read_binary_new        ;A következő karakter
 ↪  beolvasása
    read_binary_end:
                        MOV  DL, AL                 ;DL-be a beírt szám
                        POP  AX                     ;AX visszaállítása
                        RET                         ;Visszatérés a hívó
 ↪  rutinba
read_binary endp
END MAIN
```

6.20. Pointerek TYPEDEF és PTR

6.20.1. TYPEDEF direktíva

- Segítségével lehet létrehozni pointerváltozót
- Használata: `típusnév TYPEDEF távolság PTR adattípus`
  - Típusnév: bármi lehet
  - TYPEDEF kötelező kulcsszó
  - távolság: FAR, NEAR, de el is maradhat
  - PTR kötelező kulcsszó
  - BYTE, WORD stb...
- Ezután lehet .DATA szegmensben pointer deklarását elvégezni

6.20.2. PTR direktíva

- Eddig amikor két adattal végeztünk valamilyen műveletet meg kellett egyezniük a típusuknak
- Amikor regiszter is szerepel az utasításban, akkor az assembler a regiszter nevéből tudja milyen típusú adattal kell a műveletet elvégeznie
- Helyes: `MOV [BX], DL`
- Helytelen: `CMP [BX], 0` (szavakat, vagy bájtokat akarunk összehasonlítani?)
- Megoldás: `CMP BYTE PTR [BX], 0` ([BX] egy bájt hosszúságú adatra mutat megmondtuk neki)
- A **LES** és **LDS** utasítások mutatót töltenek be a regiszterbe, ezért használatuk előtt definiálni kellett azt a mutatót, amely az adatterületre mutat. A mutató megadásánál újabb direktívákkal találkozunk.
- A sztringek beolvasását és kiírását elvégezhetjük az INT 21h megszakítás segítségével is, ekkor figyelni kell ezekre:
  - Az írás és olvasás adatterületének címét a DS:DX regiszter-párba kell megadni
  - Kiíráskor a zárókarakter a $
  - Beolvasás esetén az adatterület első bájtja a puffer hosszát kell, hogy tartalmazza (az ENTER-rel együtt). A másodikba pedig a ténylegesen beolvasott karakterek száma kerül.
  - Beolvasáskor, ha a megadott puffer méreténél több karaktert akarunk megadni, azt nem engedi és sípolással figyelmeztet
  - A pufferbe csak az ENTER leütése után kerül a karaktersor. Tehát a beolvasáshoz szükséges adatterület mérete a két első (hossz) bájtból, a beolvasandó karakterek számából és az ENTER-ből tevődik össze.

```
.MODEL SMALL
FPBYTE TYPEDEF FAR PTR BYTE
.STACK
.DATA?

    buf    DB    100 DUP (?)
           adat FPBYTE buf

.CODE
main proc
                      mov    ax, dgroup          ; Adatszegmens beállítása
                      mov    ds, ax
                      les    di, adat            ; Sztring-pointer betöltése ES:DI-be
                      call   read_string
                      call   cr_lf
                      lds    si, adat            ; Sztring-pointer betöltése DS:DI-be
                      call   write_string
                      mov    ah, 4ch
                      int    21h
main endp

    CR                EQU    13
    LF                EQU    10

write_string proc
                      PUSH   AX                  ;AX mentése
                      PUSH   DX                  ;DX mentése
                      CLD                        ;Irányjelző beállítása
    write_string_new:
                      LODSB                      ;DS:SI tartalma AL-be, SI-t eggyel
↪   megnöveli
                      OR     AL, AL              ;Sztring végének ellenőrzése
                      JZ     write_string_end
                      MOV    DL, AL              ;DL-be a kiírandó karakter
                      CALL   write_char          ;Karakter kiírása
                      JMP    write_string_new    ;Új karakter
    write_string_end:
                      POP    DX                  ;DX visszaállítása
                      POP    AX                  ;AX visszaállítása
                      RET
write_string endp

read_string proc
                      PUSH   DX                  ;DX mentése a verembe
                      PUSH   AX                  ;BX mentése a verembe
                      CLD                        ;Irányjelző beállítása
    read_string_new:
                      CALL   read_char           ;Egy karakter beolvasása
                      CMP    DL, CR              ;ENTER ellenőrzése
                      JE     read_string_end     ;Vége, ha ENTER volt az utolsó karakter
                      MOV    AL, DL              ;AL-be a karakter
                      STOSB                      ;AL tartalma ES:DI címre, DI-t eggyel
↪   megnöveli
                      JMP    read_string_new     ;Következő karakter beolvasása
    read_string_end:
                      XOR    AL,AL
                      STOSB                      ; Sztring lezárása 0-val
                      POP    AX                  ;BX visszaállítása
                      POP    DX                  ;DX visszaállítása
                      RET                        ;Visszatérés
read_string endp

END main
```

6.21. Sztring kiíratás/beolvasása INT 21h-val

- A programunkban 10 karakter méretű puffert foglaltunk le, de ebbe csak 9 karaktert tudunk beírni, mivel az ENTER (Dh) karakternek is kell egy hely
- Egy adat címét megadhatjuk az **OFFSET** operátor segítségével is. Írjuk át az alábbi programot úgy, hogy a LEA DX, Adat helyett a MOV DX, OFFSET Adat utasítást használjuk. Mindkét esetben a DX regiszterbe az Adat címe töltődi

```
.MODEL SMALL
.STACK
.DATA
    Attr DB 10,0              ;Hossz adatok: puffer méret, karakterszám
    Adat DB 11 DUP ('$')      ;Adatterület, feltöltve ”$” karakterrel
.CODE
main proc
                MOV  AX, DGROUP     ;Adatszegmens beállítása
                MOV  DS, AX
                LEA  DX, Attr       ;Input puffer, az első két bájt a hosszakhoz
                MOV  AH, 0Ah        ;Puffered keyboard input
                INT  21h            ;Sztring beolvasása
                CALL cr_lf          ;Soremelés
                LEA  DX, Adat       ;Output puffer
                MOV  AH, 9h         ;Print string
                INT  21h            ;Sztring kiírás
                MOV  AH,4Ch         ;Visszatérés az operációs rendszerbe
                INT  21h
main endp

END main
```

6.22. Lemezmeghajtó kezelése

- A programban az INT 25h megszakítást használjuk, amely segítségével egy blokkot (szektort) olvasunk be a lemezről. A beolvasás nem karakterenként történik, mint például a billentyűzetről, hanem blokkosan. Ilyenkor csak a forrás és cél címet, valamint a másolandó bájtok számát kell megadnunk.

- A floppy lemezen egy blokk mérete 512 bájt, így az adatok tárolására 512 bájtnyi helyet foglalunk le az adatszegmensben.

- Programunk segítségével a beolvasott adatokat kétféleképpen írjuk ki: hexadecimális és karakteres formában is. Egy 80 karakter/soros képernyőn egy sorban 16 bájtnyi adat fér el. Így egy blokk kiírásához 32 sorra lesz szükség

```
.MODEL SMALL
Space EQU " " ;Szóköz karakter
.STACK
.DATA?
    block DB 512 DUP (?)    ;1 blokknyi terület kijelölése
.CODE
main proc
                    MOV  AX, Dgroup          ;DS beállítása
                    MOV  DS, AX
                    LEA  BX, block           ;DS:BX memóriacímre tölti a blokkot
                    MOV  AL, 0               ;Lemezmeghajtó száma (A:0, B:1, C:2, stb.)
                    MOV  CX, 1               ;Egyszerre beolvasott blokkok száma
                    MOV  DX, 0               ;Lemezolvasás kezdőblokkja
                    INT  25h                 ;Olvasás
                    POPF                     ;A veremben tárolt jelzőbitek törlése
                    XOR  DX, DX              ;Kiírandó adatok kezdőcíme DS:DX
                    CALL write_block         ;Egy blokk kiírása
                    MOV  AH,4Ch              ;Kilépés a programból
                    INT  21h
main endp
write_block proc                             ;Egy blokk kiírása a képernyőre
                    PUSH CX                  ;CX mentése
                    PUSH DX                  ;DX mentése
                    MOV  CX, 32              ;Kiírandó sorok száma CX-be
    write_block_new:
                    CALL out_line            ;Egy sor kiírása
                    CALL cr_lf               ;Soremelés
                    ADD  DX, 16              ;Következő sor adatainak kezdőcíme;
                    LOOP write_block_new     ;Új sor
                    POP  DX                  ;DX visszaállítása
                    POP  CX                  ;CX visszaállítása
                    RET
write_block endp

out_line proc
                    PUSH BX                  ;BX mentése
                    PUSH CX                  ;CX mentése
                    PUSH DX                  ;DX mentése
                    MOV  BX,DX               ;Sor adatainak kezdőcíme BX-be
                    PUSH BX                  ;Mentés a karakteres kiíráshoz
                    MOV  CX, 16              ;Egy sorban 16 hexadecimális karakter
    hexa_out:
                    MOV  DL, Block[BX]       ;Egy bájt betöltése
                    CALL write_hexa          ;Kiírás hexadecimális formában
                    MOV  DL,Space            ;Szóköz kiírása a hexa kódok között
                    CALL write_char
                    INC  BX                  ;Következő adatbájt címe
                    LOOP hexa_out            ;Következő bájt
                    MOV  DL, Space           ;Szóköz kiírása a kétféle mód között
                    CALL write_char
                    MOV  CX, 16              ;Egy sorban 16 karakter
                    POP  BX                  ;Adatok kezdőcímének beállítása
    ascii_out:
                    MOV  DL, Block[BX]       ;Egy bájt betöltése
                    CMP  DL, Space           ;Vezérlőkarakterek kiszűrése
                    JA   visible             ;Ugrás, ha látható karakter
                    MOV  DL, Space           ;Nem látható karakterek cseréje szóközre
    visible:
                    CALL write_char          ;Karakter kiírása
                    INC  BX                  ;Következő adatbájt címe
                    LOOP ascii_out           ;Következő bájt
                    POP  DX                  ;DX visszaállítása
                    POP  CX                  ;CX visszaállítása
                    POP  BX                  ;BX visszaállítása
                    RET                      ;Vissza a hívó programba
out_line endp

END main
```

6.23. Karakteres videó-memória kezelése

- A képernyőn megjelenő kép nem más, mint a képernyő-memória leképezése
- A videó-memória szegmenscíme 0B800h, ami megfelel a képernyő első karakterpozíciójának
- Minden karakterpozíciót egy 16 bites szóval jellemezhetünk
- Alsó 8 bit jellemzi a karakterkódot
- Felső 8 bit jellemzi a megjelenítő attribútumot
- Az attribútum értékének kiszámítása: 128*villogás + 16 * háttérszín + karakterszín. Az AX regiszter használata esetén az AL-be kerül a karakterkód és AH-ba az attribútumot.

```
.MODEL SMALL
.STACK

.CODE

main proc
        mov ax, 0b800h          ; videó-memória szegmenscíme
        mov es, ax              ; szegmenscím behelyezése es-be

        mov ah, 0h              ; képernyőírás kódja
        mov al, 3h
        int 10h
        MOV DI, 1838            ;Képernyő közepének ofszetcíme
        MOV AL, "*"             ;Kiírandó karakter
        MOV AH, 128+16*7+4      ;Színkód: szürke háttér, piros karakter, villog
        MOV ES:[DI], AX         ;Karakter beírása a képernyő-memóriába
        MOV AH, 4Ch             ;Kilépés
        INT 21h
main endp

END MAIN
```

7. Forrás

- Intel processzorok programozása assembly nyelven (Gimesi László) – Programkódok illetve magyarázatok
- Csörnyei Zoltán - Fordítóprogramok-Typotex (2006) – Elméleti rész