Matijevics István

A PROGRAMOZÁS ALAPJAI ÉS PROGRAMOZÁSI NYELVEK

Tankönyv

SZABADKAI MŰSZAKI FŐISKOLA

SZABADKA, 2001

TARTALOMJEGYZÉK

0.	**.....**	**.....**	.....	**TARTALOMJEGYZÉK**	1.
1.	**.....**	**.....**	.....	**A PROGRAMOZÁS KEZDETEI**	5.
	1.1	**.....**	.....	**BEVEZETÉS**	5.
	1.2	**.....**	.....	**MONOLITIKUS PROGRAMOK – MONOLITIKUS PROGRAMOZÁS**	6.
	1.3	**.....**	.....	**A SZOFTVERTERMÉKEKKEL SZEMBEN TÁMASZTOTT IGÉNYEK**	7.
	1.4	**.....**	.....	**SZOFTVERKRÍZIS**	8.
	1.5	**.....**	.....	**A SZOFTVERKRÍZIS HIBAJELENSÉGEINEK OKAI**	9.
		1.5.1	.....	**A szoftver céljának felismerése**	9.
		1.5.2	.....	**A program bonyolultsága**	10.
		1.5.3	.....	**A programspecifikáció nehézségei**	10.
		1.5.4	.....	**A programozói környezet elhanyagolása**	10.
		1.5.5	.....	**A társadalmi környezet megváltozásának hatása a programozásra**	10.
	1.6	**.....**	.....	**A SZOFTVERKRÍZIS ENYHÍTÉSE**	11.
	1.7	**.....**	.....	**SZOFTVER ENGINEERING**	12.
		1.7.1	.....	**A szoftverminőség jellemzői**	12.
	1.8	**.....**	.....	**MODULÁRIS PROGRAMOZÁS**	13.
		1.8.1	.....	**Bevezetés**	13.
		1.8.2	.....	**A moduláris programozás előnyei**	14.
		1.8.3	.....	**A modul típusai**	14.
		1.8.4	.....	**Moduláris programtervezés**	15.
		1.8.5	.....	**Moduláris programok tesztelése, karbantartása**	17.
2.	**.....**	**.....**	.....	**SZOFTVERFEJLESZTÉS ÉS FÁZISAI**	19.
	2.1	**.....**	.....	**KÖVETELMÉNY ANALÍZIS**	19.
	2.2	**.....**	.....	**SPECIFIKÁCIÓ**	19.
	2.3	**.....**	.....	**TERVEZÉS**	20.
	2.4	**.....**	.....	**IMPLEMENTÁCIÓ**	21.
	2.5	**.....**	.....	**TELEPÍTÉS ÉS TESZTELÉS**	21.
	2.6	**.....**	.....	**KARBANTARTÁS**	21.
	2.7	**.....**	.....	**A SZOFTVERTERMÉKEK KOMPONENSEI**	22.
		2.7.1	.....	**Adatfolyam diagram**	22.
		2.7.2	.....	**Blokk diagram**	22.
	2.8	**.....**	.....	**A SZOFTVER TERVEZÉSI FÁZISAI**	23.
		2.8.1	.....	**A felhasználói felület tervezése**	24.
3.	**.....**	**.....**	.....	**A PROGRAMOZÁS MÓDSZERTANA**	25.
4.	**.....**	**.....**	.....	**ASSEMBLY PROGRAMOZÁS**	29.
	4.1	**.....**	.....	**BEVEZETÉS**	29.
	4.2	**.....**	.....	**ASSEMBLY PROGRAMOZÁSHOZ SZÜKSÉGES HARDVER ISMERETEK**	29.
	4.3	**.....**	.....	**IBM PC ÉS KOMPATIBILIS SZEMÉLYI SZÁMÍTÓGÉPEK**	30.
		4.3.1	.....	**Bevezetés**	30.
		4.3.2	.....	**A 8086, 8088 mikroprocesszor belső felépítése**	31.
			4.3.2.1	Az Intel 8086 processzor felépítése	31.

		4.3.2.2	A 8086 mikroprocesszor működése	33.
		4.3.2.3	A 8086 címzési technikája	34.
		4.3.2.4	A programozási címzési módok	34.
		4.3.2.5	Adatmemória címzési módok	35.
	4.3.3	.....	**80286, 80386, 80486, Pentium, Pentium II és Pentium III processzorok**	35.
	4.3.4	.....	**Az IBM PC sinrendszere**	35.
		4.3.4.1	**Bevezetés**	35.
		4.3.4.2	**IBM PC belső sinrendszer (busz)**	36.
	4.3.5	.....	**A számítógép felépítése a programozás szempontjából**	36.
4.4	**.....**	.....	**EGYSZERŰ ASSEMBLY PROGRAMOK**	38.
	4.4.1	.....	**A DEBUG program**	38.
	4.4.2	.....	**Assembly programok írása, fordítása, szerkesztése és futtatása**	41.
	4.4.3	.....	**A *.exe és *.com program felépítése**	43.
		4.4.3.1	1. feladat	44.
		4.4.3.2	2. feladat	45.
		4.4.3.3	3. feladat	46.
		4.4.3.4	4. feladat	47.
		4.4.3.5	5. feladat	48.
		4.4.3.6	6. feladat	49.
		4.4.3.7	7. feladat	51.
		4.4.3.8	8. feladat	55.
	4.4.4	.....	**Az IBM PC grafikus kártyájának vezérlése assembly-ből**	57.
		4.4.4.1	**Különféle grafikus üzemmódok programozása**	57.
		4.4.4.2	**Az IBM PC számítógép CGA grafikus üzemmódja**	57.
		4.4.4.3	**Az IBM PC számítógép VGA grafikus üzemmódja**	59.
5.	**.....**	**.....**	**PROLOG**	63.
5.1	**.....**	.....	**BEVEZETÉS**	63.
5.2	**.....**	.....	**A PROLOG NYELVTANI SZABÁLYAI**	64.
	5.2.1	.....	**Lekérdezés tények alapján**	66.
		5.2.1.1	1. példa	67.
		5.2.1.2	2. példa	68.
	5.2.2	.....	**Rekurzió**	68.
		5.2.2.1	3. példa	69.
	5.2.3	.....	**Tagadás**	70.
		5.2.3.1	4. példa	71.
		5.2.3.2	5. példa	71.
	5.2.4	.....	**Az eredmények kijelzése**	72.
		5.2.4.1	6. példa	72.
	5.2.5	.....	**Levágás és kudarc**	73.
	5.2.6	.....	**Levágás**	73.
5.3	**.....**	.....	**A TURBO-PROLOG BETÖLTÉSE, INDÍTÁSA ÉS MENÜRENDSZERE**	75.
5.4	**.....**	.....	**A LOGIKAI NYELVEKRE VONATKOZÓ NÉHÁNY FONTOSABB DEFINÍCIÓ**	75.
5.5	**.....**	.....	**A PROLOG PROGRAM TULAJDONSÁGA**	76.
5.6	**.....**	.....	**A TURBÓ-PROLOG PROGRAM SZERKEZETE**	77.

	5.7	**.....**	.....	**A LEGGYAKRABBAN HASZNÁLT PROLOG ALGORITMUSOK**.....	80.
		5.7.1	.....	**Szélsőérték-kiválasztás (legkisebb, legnagyobb elem keresése)**.....	82.
		5.7.2	.....	**Összegzés**.....	83.
		5.7.3	.....	**Elem törlése listából**.....	84.
		5.7.4	.....	**Összefuttatás**.....	87.
		5.7.5	.....	**Únióképzés, metszetképzés**.....	87.
		5.7.6	.....	**Rendezés permutálással**.....	89.
		5.7.7	.....	**Rendezés minimumkiválasztással**.....	91.
		5.7.8	.....	**Rendezés beszúrással**.....	91.
		5.7.9	.....	**Gyorsrendezés (Quicksort)**.....	92.
6.	**.....**	**.....**	.....	**MELLÉKLETEK**.....	95.
	6.1	**.....**	.....	**ASCII KÓDTÁBLÁZAT**	95.
	6.2	**.....**	.....	**I8086 ÖSSZEFOGLALÓ**	96.
7.	**.....**	**.....**	.....	**IRODALOM**	107.

1. A PROGRAMOZÁS KEZDETEI

1.1 BEVEZETÉS

Az 1950-es években indult programozásra jellemző volt, hogy a programozók különösen nagy hangsúlyt fektettek arra:
hogy a programok futási ideje a lehető legrövidebb legyen és
a programok a lehető legkisebb tárterületet (memóriát) foglaljanak el.

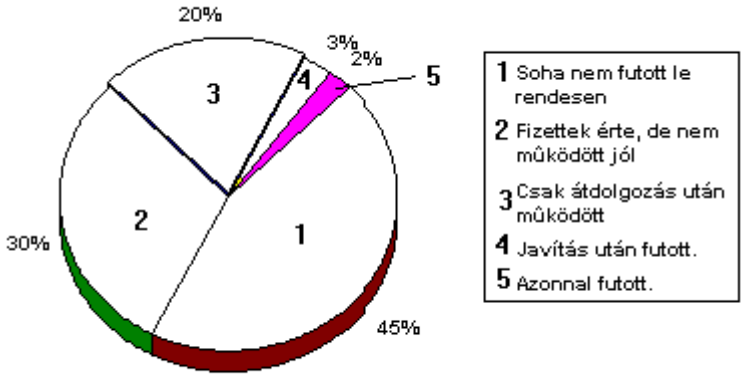
A fenti két követelmény a kor szerény műszaki feltételei miatt jelentkezett, a számítástechnika (h)őskorában igen kis műveletvégrehajtási sebességű (nagy ciklusidő) gépek álltak a rendelkezésre szerény tárkapacitásokkal. Igen gyakran előfordult, hogy a programozó csak akkor tudta a programot betölteni, ha néhány utasítást ügyes trükkökkel lefaragott. Ebben az időben a jó programozó rövid, elegáns és sokszor bonyolult programokat írt.

Az ily módon létrehozott programokra jellemző volt az, hogy hibása, vagy sokszor sehogy sem tudták kielégíteni a velük szemben támasztott igényeket. Ennek a helyzetnek a szemléltetéseére szolgál az 1.1 ábra.

Program: utasítások sorozata, melyet a számítógép azért hajt végre, hogy megoldjon egy feladatot. A számítógép csak az operatív memóriájában levő gépi kódú programot hajtja végre. A program lehet rendszerprogram és felhasználói program.

Programozás: a program létrehozása. Szűkebb értelemben a program lefutásásnak részletes leírása, a program kódolása, tesztelése és a programdokumentáció létrehozása. Tágabb értelemben az előzőeket még megelőzi a feladat elemzése és az egész programcsomag megtervezése nagy vonalakban.

Operatív memória: munkamemória, egy közvetlen elérésű memória, amelyben a program és adatok helyezkednek el.



1.1 ábra: a programozás első éveiben megírt programok hatékonysága

1.2 MONOLITIKUS PROGRAMOK – MONOLITIKUS PROGRAMOZÁS

A műszaki feltételek, vagyis a hardver és a szoftver fejlettsége a számítástechnika kialakulásakor csak úgynevezett monolitikus prgramozást tett lehetővé. Ezekre a programokra jellemző, hogy elejétől kezdve utasításonként kódolták a programlépéseket, valamint az is jellegzetességük, hogy csak ezután kerülhetett sor a tesztelésre. Az ismétlődő részeket annyiszor kellett megírni, ahányszor a programban előfordultak. Ez a korszak egészen a 60-as évek elejéig tartott.

Hardver: a számítógép összes mechanikus és elektronikus része.

Szoftver: a hardvert irányító összes program.

Ez a programozási technika ugyan elavult, de kisebb mai rendszereknél, például egyszerű mikrovezérlőknél előfordulhat alkalmazása, hiszen ezeknél a berendezéseknél néhány tíz bájt RAM memória, néhány szintes veremtár., stb. jelenti a hardverkapacitást.

A módszer nagy hátránya az, hogy a tesztelés időtartama bármilyen hosszú lehet, egy-két nap ugyanúgy mint 1 hónap, vagy hosszabb idő is. Sohasem lehet meghatározni azt, hogy a tesztelés milyen fázisban van. A nagy számú programban levő logikai ág teszi teljesen lehetetlenné a tesztelést. Ha ilyenkor a tesztelést időhiány miatt csak a főágakra korlátozzuk, megeshet, hogy így a mellékágakban meglevő hibák futtatáskor okoznak problémát. Ilyenkor a javítás, programfoltozgatás csak tovább rontja a helyzetet, ugyanis még áttekinthetetlenebbé válik a rendszer.

Tesztelés: olyan eljárás, melynek célja a programhibák felderítése. Programiba az eredmények minden eltérése az előre megadott értékektől.

Ha mégis sikerül belőni a programot, a program továbbfejlesztése lesz problématikus, az áttekinthetőség hiánya miatt. Amennyiben mégis szükséges a program továbbfejlesztése, akkor lehet hogy jobb megoldás a feladat újrafogalmazása és új program írása. Ez különösen akkor igaz, ha más munkáját szeretnénk továbbfejleszteni. Minden programozó szereti a problémákat a saját elgondolása, tapasztalata alapján megoldani, ami trükkös, csavaros megoldásokat eredményez, így mások számára érthetetlen megoldások keletkeznek.

Programozó: az a személy aki kialakítja, megírja és teszteli a programot. Van rendszerprogramozó és felhasználói programokat író programozó.

Monolitikus programok írásánál lehet a hatékonyságot növelni és a költségeket csökkenteni, ha a bonyolultabb, nagyobb szaktudást igénylő részeket tapasztalt programozókra, míg az egyszerűbb részeket kezdőkre, kevesebb tapasztalattal rendelkező szoftverfejlesztőkre bízzuk.

A monolitikus programok létrehozásának nagy hátránya az, hogy nehezen, esetleg

Erőforrás: az összes hardver elem és az összes szoftver elem (program).

sehogyan sem becsülhetjük meg előre a létrehozásához szükséges munkaidőt és erőforrásokat.

1.3 A SZOFTVERTERMÉKEKKEL SZEMBEN TÁMASZTOTT IGÉNYEK

A számítástechnika indulásakor a szoftverrel szemben támasztott követelmények a következők voltak:

- a feladat pontos végrehajtása,
- a lehető legrövidebb futási idő és
- a lehető legkisebb helyfoglalás (memória-kapacitás).

A mai modern szoftvertermékekkel szemben a következő elsődlegesen fontos követelményrendszert állíthatjuk fel:

- a program feltétlen megbízhatósága,
- a gyors és olcsó kivitelezés,
- a határidők pontos betartása,
- a hibátlan programozás minimális bejáratási időt adjon,
- a könnyű karbantarthatóság (maintability),
- egyszerű továbbfejleszthetőség (extensibility) és
- a programnak a programozó személyiségétől való függetlensége (egoless programming).

Megbízhatóság: egy funkcionális egységgel szemben támasztott követelmény, hogy a feladatát adott időtartam alatt végrehajtsa. Hardver megbízhatóság az egész gépre, a szoftvermegbízhatóság az adott programra vonatkozik.

Karbantartás: minden tevékenység, amely biztosítja a gép működését. Hardver karbantartás a hardver elemek hibátlan működését szolgálja, míg a szoftver karbantartás a szoftverhibák megszüntetése mellett az új operációs rendszerekhez való illesztést is jelenti.

Az imént felsorolt minőségi követelmények mellé mennyiségi követelmények is párosulnak. A számítástechnika fejlődése igen erőteljes volt, kezdetben csak kevés programot írtak, ma már gyakorlatilag életünk minden területét uralják azok a berendezések, melyek programok segítségével működnek, hiszen gyakorlatilag minden berendezés egy kisebb, vagy nagyobb teljesítményű számítógép, vagy mikrovezérlő.

Megváltoztak a programfejlesztési, programírási technológiák, a kezdeti monolitikus programírástól a moduláris és struktúrális programfejlesztésen keresztül az objektumorientált rendszerekig jutott el a számítástechnika.

Ha összehasonlítjuk a programfejlesztést az ipari nagy szorozatban gyártott berendezésekkel, akkor meg kell állapítani, hogy itt még mindig egyedi programfejlesztésekről van csak szó. Így igen sok hibaforrás marad a szoftver fejlesztésekben, ahol szoftverkrízisről beszélhetünk.

Mivel még ma is előfordul monolitikus programozás (az egyszerű mikrovezérlőknél), érdemes elemezni a szoftverkrízis kérdéseit.

1.4 SZOFTVERKRÍZIS

A fent tárgyalt monolitikus programozásnál megállapítható általánosságban, hogy létezik:

- megbízhatatlanság,
- az alkalmazkodóképesség hiánya,
- nehézkesség és
- az 'udvariatlanság'.

Amikor megbízhatatlanságról beszélünk, akkor különösen arról teszünk említést, hogy a programmal szemben felállított specifikációknak nem tesz eleget a megírt program, egyszóval nem hajtja végre az előírt feladatokat. Ilyenkor a program:

- a megengedett input adathalmaz nem minden elemére értelmezett, esetleg
- nem minden elemre fejeződik be, vagy
- hibás eredményeket szolgáltat.

Szintén megbízhatatlansági probléma az is, ha a program:

- nem védekezik az input szintaktikus hibái ellen, vagy
- a program adatait és adatbázisait illegális hozzáférések ellen nem védi, vagy
- a program nincs felkészítve a rendszerkatasztrófák elhárítására.

Szintaktikus hiba: elírás.

A program alkalmazkodóképességének hiánya a kész program azon tulajdonsága, hogy:

- 'érzékeny' az operációs rendszer módosításaival, vagy cseréjével szemben (máshogy működik DOS és máshogy WINDOWS, illetve UNIX operációs rendszer alatt), vagy
- a program nem használja ki az operációs rendszer és a hardver nyújtotta lehetőségeket, vagy
- új gépkonfiguráció esetén nem képes az új lehetőségeket használni, vagy
- más gépre való átvitelkor nagy módosításokat kell létrehozni (portábilitás hiánya), vagy
- növekvő terheléskor a teljesítőképessége rohamosan csökken.

Operációs rendszer: programok összessége, mely lehetővé teszi a felhasználó számára a számítógép összes erőforrásának (hardver és szoftver) egyszerű használatát.

A program nehézkessége a program elkészítésére és továbbfejlesztésére vonatkozik, ami lehet:

- összeállítási, részekből történő összeszerkesztési probléma, ami átláthatatlan programszerkezetet okoz, vagy
- tesztelési nehézség, tesztösszeállítás, eredménykiértékelés, ami nagy fáradságba kerül, vagy sok gépidőt foglal le, vagy,
- nehéz programhiba-hely, programhiba-ok felderítés, vagy
- bármilyen módosítás végrehajtása aránytalanul nagy munkabefektetés.

A program udvariatlanságáról akkor beszélünk, ha kezelésekor, futásakor (tehát a készülék használatakor) vagy az eredmények kiértékelésekor nehézségekbe ütközik a felhasználó. Udvariatlan a program, ha:

- helytelen kezeléskor nem ad lehetőséget a hiba kijavítására, vagyis lebénul, vagy
- csak egyféleképpen dolgozik, például nem lehet rövidítéseket használni, vagy
- a hibajelzések kétértelműek, nem elég világosak, vagy
- az input adatok bevitelekor ésszerűtlenül kell sok felesleges adatot bevinni, vagy
- a kimenő adatok, eredmények nem szelektálhatók, nem emelhetők ki közülük a számunkra fontosak.

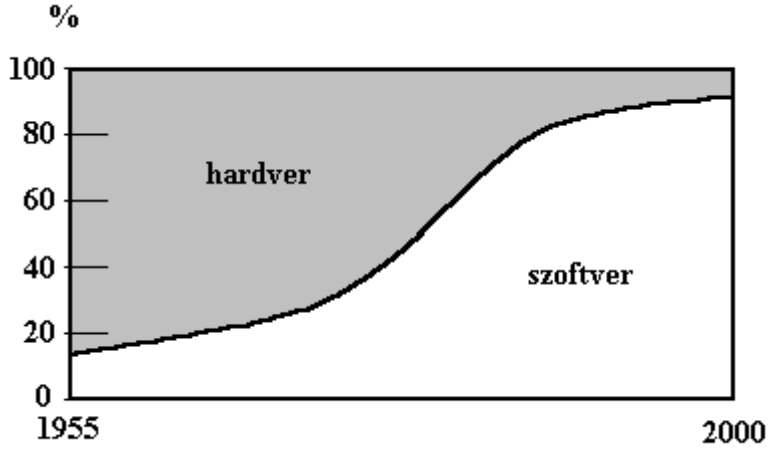
A szofverkrízis megjelenése időben a 60-as évek végére, a 70-es évek elejére tehető. Ennek egyik oka a hardverárak erőteljes csökkenése (megvásárolható a hardver, 'mindenki' számítógéppel oldja meg problémáit), a másik ok pedig a hardver-teljesítmény látványos megnövekedése (jobban ki lehet használni a rendszert, összetettebbek a megoldások).

1.5 A SZOFTVERKRÍZIS HIBAJELENSÉGEINEK OKAI

A szoftverkrízis mélyebb okainak néhány pontját a következőkben tárjuk fel.

1.5.1 A szoftver céljának felismerése

Mint már szó volt róla, a szoftverfejlesztés kezdeteinél komoly korlátozó tényező volt a korlátozott tárkapacitás és a lassú futási idő, amely tényezők a mai mikrogépeknél már nem jelentkeznek, illetve csak korlátozottabb mértékben. Mégis, előfordul, hogy programozók ma is ebbe a hibába esnek, nem használják ki a ma nyújtotta műszaki lehetőségeket. Az 1.2 ábrán az évek folyamán változó hardver-szoftver százalékos költségmegoszlását láthatjuk.



1.2 ábra: a hardver-szoftver százalékos költségmegoszlása az évek folyamán

1.5.2 A program bonyolultsága

A program bonyolultsága, vagy komplexitása azt fejezi ki, hogy a programnak 'sok' egymással többszörösen összefüggő program- és adatállapot egymásutánját kell megszerveznie.

A bonyolultság forrásai között szerepet játszik:

- a kivitelező rendezetlensége,
- a kivitelezés szervezetlensége,
- a szoftver-készítés tudományos megalapozottságának elégtelensége és
- a programozók tapasztalatlansága.

1.5.3 A programspecifikáció nehézségei

Nagyon sokszor a feladat megrendelője nem tudja teljesen megfogalmazni igényeit. Az is előfordul, hogy ugyan pontosan és helyesen megfogalmazott igénnyel lép fel a megrendelő, de az adott szakterületet a programozó nem ismeri megfelelő mélységben. Ilyenkor hibás, ellentmondó és hiányos specifikációk keletkeznek. A programozó ezek után hiába készít el a specifikációnak tökéletesen megfelelő, hibátlan programot, az nem a valós feladatot hajtja végre.

1.5.4 A programozói környezet elhanyagolása

A programozó, mint egyén tudományos-technikai módszerekkel oldja meg feladatát, de mindig egy (vagy több) csoport tagjaként. Mikor a csoport tagjaként dolgozik, akkor az emberi környezet felé fordul. Ha elhanyagolják ezt az emberi környezetet, akkor ez sokszor felelőssé tehető a szoftver-gyártás sikertelenségéért.

1.5.5 A társadalmi környezet megváltozásának hatása a programozásra

A szoftver-krízis valamikor a hatvanas években jelentkezett először. A nagyon drága felszerelés, nagy költségekkel kiképzett személyzet a kezdetekkor csak azt tette lehetővé, hogy nagy cégek az egyetemek kiváló szakembereivel együttműködve készítsenek programokat. Mind a cégeknél, mind az egyetemeken kiváló szakgárda vett részt a szoftver-gyártásban.

A nagyobb teljesítményű programok megírása, tesztelése, használata és karbantartása mindig egy-egy kiváló szakemberhez fűződött. Ez biztosíték volt a kiváló programok, munkák megszületésére.

A hatvanas évek elején bekövetkező számítástechnikai robbanás, amikor a számítástechnika kikerül a termelésbe, közgazdaságba, űrhajózásba, egyszóval a mindennapi életbe, azt eredményezi, hogy igény keletkezik a szoftver nagyipari gyártására. Ez a szoftverkészítés személyes jellegét, az egyéni, manufakturális gyártást megszűnteti, átalakítja a nagyipari gyártás mintájára.

Csakhogy a szoftver-gyártás nem működhet teljesen a nagyipari gyártás mintájára a következő eltérések miatt:

- a szoftver egyedi termék, tehát egy termék nem gyártható sorozatban,
- a szoftvertermékénél nincs redundancia, vagyis már egy rossz utasítás katasztrófális hibát okozhat,
- a szoftverelemek tudományos vizsgálata még nem érte el azt a szintet, mint a gépipari gyártásnál az egyes termékeket jellemző elemeknek az ismerete és
- a nagyipari gyártásnál rendkívül fejlett szervezettség és igen szigorú szabványosítás van, ami nem jellemző az egyedi szofvertermék előállítására.

Redundancia: egy adatban, utasításban vagy programban egy információ többszöri előfordulása.

1.6 A SZOFTVERKRÍZIS ENYHÍTÉSE

A szofverkrízis kiküszöbölésére olyan programfejlesztési módszerek alakultak, amelyek a következő csoportokba sorolhatók:

- moduláris programozás,
- strukturált programozás,
- a mesterséges intelligencia szabályait alkalmozó technikák és
- objektum orientált programozás.

Mesterséges intelligencia: tudomány, mely olyan feladatokat old meg, amiket az ember intelligenciája segítségével tud megoldani.

1.7 SZOFTVER ENGINEERING

A szoftverkrízis és annak következményei vezettek 1968-ban a *szoftver engineering* fogalmának megszületéséhez. A szoftver engineering egy új szemlélet, egy módszertant biztosít a programok, programfejlesztési eszközök termékszerű gyártására. Előtérbe került a szoftver, mint szellemi produktum termékszerű jellege, azaz a minőség, az eladhatóság. A szoftver termékké válását a köztudatban az nehezítette, hogy - "műszaki" termék létére - olyan tulajdonságokkal is rendelkezik, amelyek az ember által készített élvezeti termékeknek (művészet) is sajátjuk. Ezen tulajdonságok összefoglalása látható az 1.1 táblázatban.

1.1 táblázat: a művészet, a szoftver és a műszaki termék összehasonlítása

művészet	**szoftver**	**"mûszaki" termék**
elsõsorban esztétikai igényt elégít ki	elsõsorban célszerûségi igényt elégít ki	
nem az anyagi megjelenése a lényeges		anyagi megjelenésében funkcionál
nincs közvetlen kapcsolata a valósággal	közvetlen kapcsolatban van a valósággal	
nem sorozatgyártással készül		sorozatgyártással készül
fejlesztés = gyártás	fejlesztés + gyártás	
másolat eredeti	másolat = eredeti	
használat miatt nem igényel karbantartást	használata közben karbantartást igényel	
használata közben környezete nem befolyásolja a viselkedését	viselkedése függ a külsõ környezettõl (például helytelen kezelés által fellépõ hibák)	

A szoftver engineering néhány definíciója:

eszközök és módszerek a szoftver termékszerû elõállítására,

tudományos ismeretek célszerû alkalmazása számítógépprogramok valamint a fejlesztésükhöz, mûködésükhöz, karbantartásukhoz szükséges dokumentációk tervezésére és megalkotására (Boehm 1976),

szoftver termékek szisztematikus gyártására és karbantartására és költségvetésére vonatkozó technológiai és menedzsment vezérelv (IEEE 1983).

1.7.1 A szoftver minôség jellemzôi

A szoftver minősége egzakt mutatókkal nem leírható. Külső és belső tényezőket kell vizsgálunk, hogy képet alkothassunk a szoftver minőségéről. A külső tényezők határozzák meg, hogy a szoftvernek milyen feltételeknek kell megfelelnie, a belső tényezők a kódolási módszert jellemzik. Tulajdonképpen a végeredmény szempontjából csak a külső tényezők számítanak, azonban ezeknek jó része erősen függ a belsô tényezőktől.

Külső tényezők:

- Helyesség (correctness): a programnak pontosan a specifikációja szerint kell működni.
- Rabosztusság (robustness): szélsôséges (nem specifikált) feltételek közötti mûködési mód, a szoftver a legváratlanabb esetekben sem okozhat jelentős kárt.,
- Módosíthatóság (extendibility): a szoftver más környezetben kis kiigazításokkal is működik,
- Újrahasználhatóság (reuseability): szoftver, vagy egyes részeinek felhasználása más problémák megoldására,
- Kompatibilitás (compatibility): más szoftverekkel való együttmûködés,
- Könnyű használhatóság: a szoftver használatának gyors és könnyű elsajátítása, emberközeli és logikus szoftverkezelés,
- Hatékonyság,
- Hordozhatóság,
- Tesztelhetőség,
- Integritás.

Belső tényezők:

- Olvashatóság: a programnak világosan és érthetően kell kódolva lennie, tiszta logikával,
- Modularitás: mennyire van részekre bontva a program, és az egyes részek mennyire látnak el külön feladatokat.
- Strukturáltság.

A külső tényezők közül az első kettő a szoftver megbízhatóságát (reliability) jelenti. Egy szoftver akkor tekinthető megbízhatónak, ha minden körülmények között a specifikációjának megfelelően működik. Hibás adatok beérkezése váratlan körülmény, ezt a szoftvernek le kell tudnia kezelni. A szoftver biztonságosságának növelése költséges dolog, viszont egy olyan szoftver ami nincs felkészülve extrém körülményekre, sokkal nagyobb hibát képes okozni (pl. adatvesztés).

1.8 MODULÁRIS PROGRAMOZÁS

1.8.1 Bevezetés

A monolitikus, tehát egyben megírt programoknál már látható volt, hogy nem lehet kézben tartani a nagyobb terjedelmű programok írását, fejlesztését, tesztelését, karbantartását és bővítését.

Igen hamar felfigyeltek a programozók arra, hogy azokat a programokat egyszerű és könnyű implementálni és karbantartani, amelyekben egymástól elkülönített és jól érthető részek vannak. Ezeket a részeket moduloknak nevezzük. A probléma megoldása gyorsabbá és könnyebbé válik, ha a feladatot kódolás és tesztelés szempontjából külön kezelhető részekre bontjuk, vagyis nem kell a feladatot minden szempontból egyidejűleg szem előtt tartani.

L.L. Constantine definíciója szerint a modul formálisan utasítások sorozata, ahol minden modulnak saját elnevezése, neve van, a programrendszer más részeiből hívható, valamint rendelkezhet helyi, vagy lokális változókkal. Ilyen szempontból modul pl. az assembly program alprogramja (szubrutinja).

Assembly: szimbolikus nyelv, melynek utasításkészlete a gépi kódú utasításokkal egyezik meg.

Alprogram: önmagában független programegység, de önmagában nem hajtható végre. Mindig a főprogram, vagy más alprogram hívja, esetleg lehet önmagát hívó (rekurzív). Futása befejezésekor a hívó program következő utasítására lép vissza.

Parnas azt mondja, hogy a feladat részfeladatokra való bontásának alapvető kritériuma a információk elrejtése. Egy program több 'programobjektumra' (algoritmusra, adatsruktúrára, konfigurációra stb.) vonatkozó ismeretet tartalmaz, amelyre a program több pontján is hivatkozhatunk. Így arra kell törekedni, hogy valamely 'programobjektumra' vonatkozó ismerethalmazt a programnak egy fizikailag a többitől jól elválasztható részébe tömörítsük, ezek a részek a modulok. Akkor tökéletesen modularizált egy program, ha minden modul bizonyos ismeretek kizárolagos birtokában van.

A modul:

- a feladatnak valamilyen logikai egysége,
- önálló névvel rendelkező, önmagában érthető programegység és
- önállóan kódolható, tesztelhető.

A modul a környezetével, vagyis a többi modullal paraméterek, nevek és közös változók segítségével tartja fenn a kapcsolatot.

A nagy modulokat az előző elvek alapján tovább kell bontani egyszerűbb modulokra.

1.8.2 A moduláris programozás előnyei

A program modulokra bontása nem mindig egyszerű, da a kapott modulok egyetlen logikai funkciót hajtanak végre, így a modul igen könnyen érthető, tiszta logikával rendelkezik, világos és áttekinthető.

Az egyszerű modul kis logikai egység, így valószínűleg kevés utasításból áll. Ezért könnyű a kódolási szabványok betartása és pontos és teljes dokumentáció elkészítése. A programozók munkája ezért könnyebben ellenőrizhető.

Rövid és alapos modulvizsgálat végezhető el a rövid modulokon, így minden kódolási egység jó minőségű, jól tesztelhető. A jó minőségű modulokból felépülő program is jó minőségű.

Moduláris programozásnál az egyes modulokat más és más programnyelven írhatjuk. Ebből következik az, hogy a program megírásához a programozókat is könnyebb kiválasztani, jobb lesz a munkamegosztás. A bonyolultabb programrészeket a tapasztaltabb programozók, míg az egyszerűbbet a kezdő programozók is megírhatják. Ez azt is eredményezheti, hogy a kezdőket korán be lehet kapcsolni komolyabb feladatokba, ahol a szerzett tapasztalatuk alapján haladhatnak előre.

További előnye a moduláris programozásnak, hogy az írási munkafolyamatot lerövidíti a több programozó bekapcsolása, hiszen párhuzamosan történik a modulok írása és tesztelése.

Moduláris programozással előrelátható, betervezhető a módosítások gyors, hibátlan kivitelezése, ami flexibilis (könnyen változtatható) programok létrehozását biztosítja. Ilyenkor csak a megfelelő modulokat kell cserélni. A szabványos modulokat modulkönyvtárakban lehet tárolni és más programok létrehozásánál beépíteni.

Modulkönyvtár: egy név alá gyüjtött modulok, meleyeket akár az operációs rendszer hívhat, de a programozó is beépítheti a modulokat a programjába.

Nagyobb programok fejlesztésének, belövésének ideje nehezen ,vagy sehogyan nem becsülhető meg, míg a rövid modulok fejlesztésének ideje könnyen becsülhető, ezzel együtt a modulokra bontott nagy programok fejlesztésére szánt idő is.

Megoldódik a moduláris programozással az esetleges fejlesztés közbeni programozócsere, hiszen a kis modulokat könnyebben veszik át az új programozók, így egy program sohasem marad ‘egy kézben’.

A moduláris programozás a karbantartást is megkönnyíti, hiszen ellentétben a monolitikus programokkal itt könnyeb megtalálni azt a pontot, ahol módosítani kell.

1.8.3 A modul típusai

A modul vizsgálatának szempontjai:

- a modul funkciója szerint,

- működési logikája szerint és
- kapcsolata más modulokhoz.

A modul fekete doboz, funkciója (feladata) az input paraméterek leképezése output paraméterekké. Nem törődünk azzal, hogyan működik.

A modul működési logikájának vizsgálatakor az érdekel bennünket, hogy hogyan hajtja végre funkcióját.

Arm szerint a modulok felosztása funkciójuk szerint:

- adatmodul (data modul),
- vezérlőmodul (control modul),
- eljárásmodul (process modul) és
- input-output modul (I/O modul).

Adatmodul

Az adatmodul passzív, a program összes adatmezeje együtt, egy helyen található.

Vezérlőmodul

Minden programban van egy vezérlőmodul és ez a modul hívja az eljárásmodulokat, szervezi és irányítja a program logikáját.

A vezérlőmodul funkciói:

- eldönteni, hogy melyik eljárásmodul kapja meg a vezérlést,
- minden eljárásmodulba nem illő tevékenység végrehajtása és
- a túl egyszerű tevékenységek végrehajtása.

Eljárásmodul

Az eljárásmodul feladata egy logikai funkció végrehajtása. Az eljárásmodul hivhat:

- másik eljárásmodult és
- I/O modult.

A modul végén visszatérési utasítás van a hívó modul következő utasítására.

I/O modul

Egyetlen perifériához kötött művelet.

1.8.4 Moduláris programtervezés

A moduláris programtervezés a következő lépésekből áll:

- a rendszer modulszerkezetének megtervezése a feladat specifikációja alapján,
- az egyes modulok feladatának írásban való rögzítése (modulspecifikáció), valamint az egyes modulok közötti kapcsolatok minimalizálása,

- a modulok ellenőrzése a szerint, hogy elég egyszerűek-e,
- a modulok algoritmusainak megtervezése és
- ha egyes modulok további elemekre bonthatók, akkor ezekre a modulokra az előző 4 lépést végrehajtjuk.

Dekompozíció: valamilyen szempont szerinti részekre bontás.

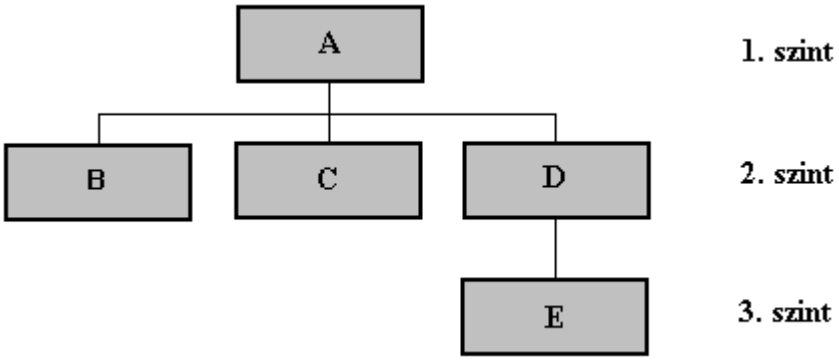
Moduláris tervezéskor ún. felülről lefelé (top down) tervezést alaklmazunk, ahol dekompozíciót hajtunk végre. Valamilyen programozási tevékenység felülről lefelé történik, a logikailag magasabb szintű egységektől kezdve az alacsonyabb szintűek felé.

A modul hívásának szabályai a következők:

- a modul ha befejezi működését a vezérlést visszaadja a hívási pont utáni utasításra,
- erős kötésnél (hierarchia) a modul csak az egy szinttel alatta levő modult hívhatja,
- gyenge kötésnél megengedett a több szinttel lejjebb levő modulok hívása is,
- nem alkalmazhatók rekurzív modulok (csak kivételesen) és
- magasabb szinten levő modult nem hívhat meg a modul.

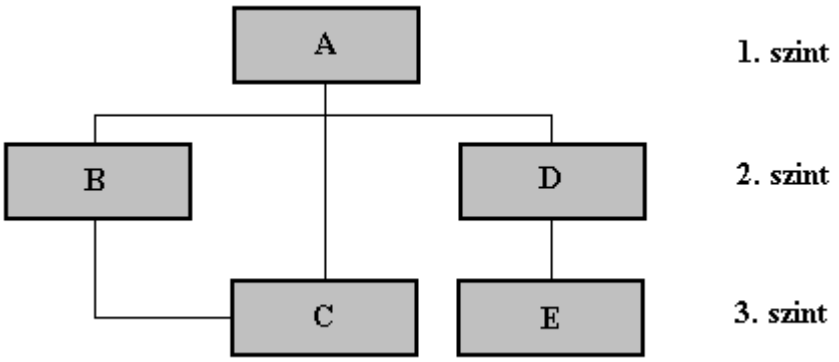
Az 1.3 ábrán egy erős kötést bemutató modulkapcsolat, míg az 1.4 ábrán egy gyenge kötésű modulkapcsolat látható.

Az erős kötésnél (1.3 ábra) 3 modulszint látható, itt kizárólag A modul hívhatja B, C vagy D modult, illetve kizárólag D hívhatja meg közvetlenül E modult. Itt kizárt például az, hogy B meghívja C modult, de nem lehetséges az sem, hogy A közvetlenül meghívja E modult, ez csak D-n keresztül történhet meg.



1.3 ábra: erős hierarchiájú programszerkezet

Amennyiben szükséges az, hogy B közvetlenül meghívja C modult, akkor a 1.4 ábra szerinti gyenge hierarchiájú programszerkezetet kell alkalmazni.



1.4 ábra: gyenge hierarchiájú programszerkezet

1.8.5 Moduláris programok tesztelése, karbantartása

A modulárisan megírt programok tesztelése két lépésben történik:

- a modulok tesztelése és
- a program tesztelése.

A modulokat külön-külön lehet csak megírni, így tesztelésük is csak egymástól függetlenül lehetséges. Ez veszélyt is rejt magában, hiszen a modulok között hívásukkor kapcsolat keletkezik (paraméterátadás). A modulok önmagukban még nem futtatható programrészek, tesztelésük bizonyos környezet létrehozását igényli. Ilyenkor meg kell oldani:

- a modul hívásának szimulálását,
- a modulból hívott modulok szimulálását és
- a tesztelt modul számára szükséges tesztadatok létrehozását és elérését.

Az lenne jó, ha:

- minden modul önmagában, más moduloktól függetlenül lenne tesztelhető,
- lényegtelen volna hogy a struktúra mely szintjén helyezkedik el a modul,
- illetve hogy a többi modul már létezik-e, vagy sem.

Hasonlóan a monolitikus program teszteléséhez itt is szükséges (lenne) az összes lehetséges eset figyelembe vétele.

A modul tulajdonságai a tesztelés szempontjából a következők:

- a program egyetlen funkcióját valósítja meg,
- áttekinthető és érthető a logikája,
- rövid,
- kis számú elágazást tartalmaz (logikai utak) és
- kis számú adaton hajt végre műveletet.

A sikeres modultesztek után a modulokat összefűzve kapjuk a programot. A program tesztelésekor a modulok összekapcsolásának sorrendjét kell ellenőrizni, melyik modul mikor hívhatja az alatta levő szinten elhelyezkedő modult. Hasonlóan a modulok

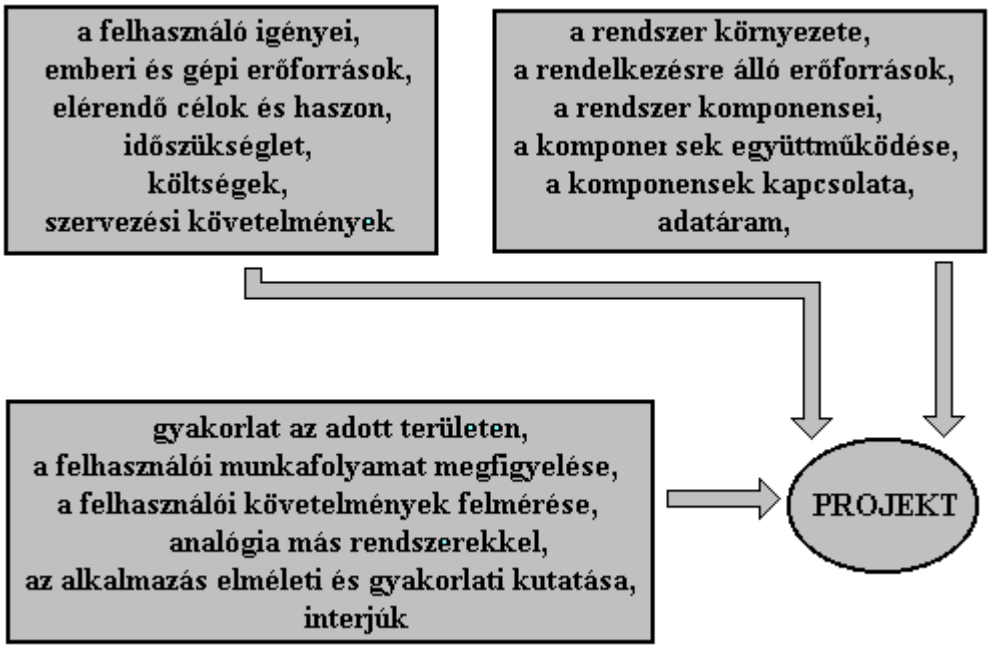
teszteléséhez, az egész program tesztelésekor is olyan adatokkal kell dolgozni, melyek az összes elágazást, ágat bejárják.

Sikeres programhasználat során is felmerülhet olyan igény, amely a program módosítását, karbantartását okozhatja. A moduláris programtechnika előnye az, hogy ilyenkor nem kell teljesen új programot írni, csak a modulok hívását, hívási sorrendjét megváltoztatni, vagy bizonyos modulokat kell csak újból létrehozni. Természetesen ilyenkor is szükséges a módosított, vagy új modulok tesztelése ugyanúgy, mint az egész, összeszerkesztett program tesztelése.

2. SZOFTVERFEJLESZTÉS ÉS FÁZISAI

2.1 KÖVETELMÉNY ANALÍZIS

A szoftverfejlesztők a megrendelőktől a szükséges információkat meg kell hogy szerezzék. A szükséges információknak valahogyan a birtokába kell jutniuk, aztán pedig rendszerezni kell a rendelkezésre álló információkat. Ez egy kétoldalú megbeszéléssorozat a majdani felhasználó és a fejlesztőcsapat között. A fázis végén megszületik az úgynevezett *Projekt terv*, amit a 2.1 ábrán láthatunk.



2.1 ábra: a projekt terv kialakítása

A Projekt tervnek egy lehetséges struktúrájára láthatunk példát a 2.1 táblázatban.

2.2 SPECIFIKÁCIÓ

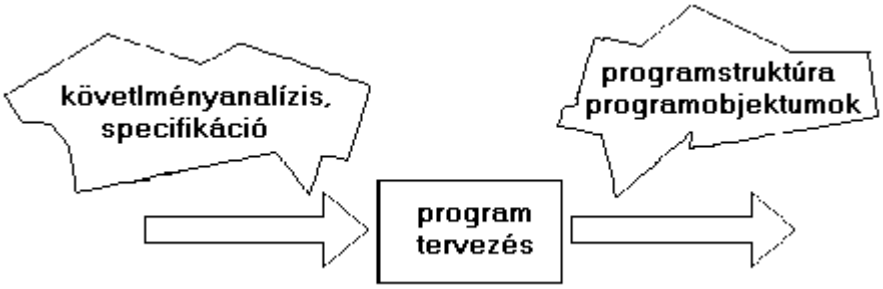
A fejlesztendő programrendszer részletes leírása szintén a két fél (megrendelő – kivitelező) közös megbeszélése után alakul ki, ami esetleg még a szerződés alapját is képezheti. Ez a részletes specifikáció úgy is felfogható, mint a követelmény analízis folytatása formális eszközökkel. A specifikáció meghatározásának három fő tevékenysége a 2.2 táblázatban látható.

2.1 táblázat: a Projekt terv kialakítása	
Bevezetés a probléma leírása a környezet leírása az elérendõ célok megfogalmazása a megoldási javaslatok összefoglalása	**Javaslatok** a fejlesztés általános stratégiája javaslat a megoldásra a felhasználó és a megrendelő szerepe a javasolt megoldásban a javasolt megoldás által biztosított funkciók a javasolt megoldás elõnyei és hátrányai
Feltételek, korlátok, követelmények a megrendelõ prioritása a felhasználó profilja a termék elvárt élettartama teljesítmény követelmények a meglévõ szoftver-hardver követelmények továbbfejlesztési elképzelések betanítási, installációs dokumnetációs követelmények a megrendelõ környezete (hardver és szoftver lehetõségei) alternatív megoldások a tervezett és az alternatív megoldások megvalósíthatósága	**Becslések** munkafelosztás ütemterv munkatársak (szervezet) költségvetés költséganalízis kockázat analízis szükséges eszközök
Eljárások a fejlesztési stratégia (életciklus modell) módszerek és jelölés-rendszerek szabványok és minõségbiztosítás költség-figyelés gyártásirányítás tesztelés, teszt adatok elfogadás feltétele fizetési feltételek	Hivatkozások felhasznált dokumentumok szakkifejezések jegyzéke szerzõdés (tervezet)

2.2 táblázat: a programrendszer kialakításának három fő tevékenysége		
Fogyasztó:	felhasználó	szoftver fejlesztõ
Nyelv:	természetes nyelv	formális nyelv
Pontosság:	nem precíz	precíz
Terminológia:	az alkalmazás szakkifejezései	szoftver terminológia
Szemlélet:	nem technológiai	technológiai

2.3 TERVEZÉS

A tervezési fázisban a felállított követelményrendszer elemzése és a specifikációk határozzák meg a program struktúráját, illetve a program elemeit (2.2 ábra).



2.2 ábra: programtervezés

Ekkor kell a program működéséhez elengedhetetlen adatszerkezetnek, rutinoknak és azok interfészének, moduloknak és azok kapcsolatainak, valamint a szükséges algoritmusoknak a kidolgozását is elvégezni. A tervezésben a követelmény analízis a *mit?*, a specifikáció a *milyet?*, a tervezés a *hogyan?* kérdésre adja meg a választ. A 2.3 táblázat mutatja hardver, szoftver és fejlesztő által meghatározott feltételeket, hatásokat.

2.3 táblázat: a tervezést befolyásoló tényezők

hardver	szoftver	személyi
kapacitás korlátok hálózati viszonyok speciális eszközök architektúra	fejlesztõi környezet adatbáziskezelõ operációs rendszer	kreativitás gyakorlat fantázia

2.4 IMPLEMENTÁCIÓ

Itt történik a programok írása, vagyis a kódolás.

2.5 TELEPÍTÉS ÉS TESZTELÉS

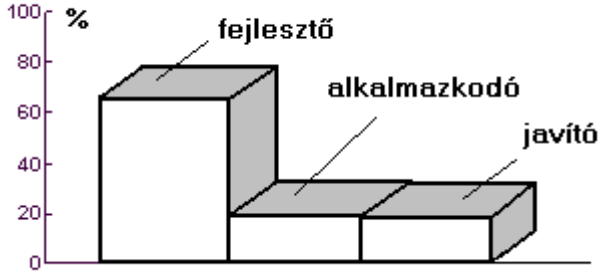
A program telepítése mellett 'élőben', tesztadatokkal történik meg a program viselkedésének ellenőrzése. Itt derülnek ki a tervezéskor nem előreláthető hibák.

2.6 KARBANTARTÁS

Még a leggondosabb tervezés, telepítés és tesztelés mellett is előfordulnak hibák, melyeket gyorsan és kis ráfordítással kell helyrehozni. A specifikációban rögzítettekhez kell tartani magát a megrendelőnek és kivitelezőnek is.

A karbantartás (2.3 ábra):

- *fejlesztô (perfective):* 65%, a felhasználó vagy a programozó által még a fejlesztés fázisában javasolt módosítások;
- *alkalmazkodó (adaptive):* 18%, a szoftver környezetében történt változásokhoz igazítás miatti módosítások;
- *javító (corrective):* 17% a mûködés közben felfedezett hibák miatti javítások.



2.3 ábra: A karbantartási költségek megoszlása

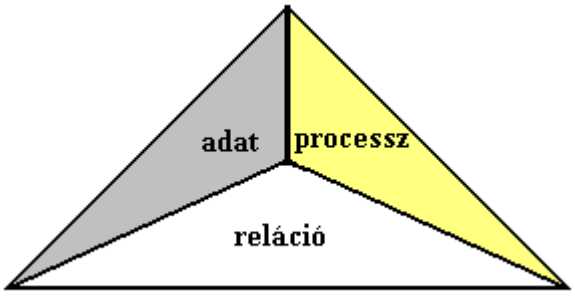
A szoftver költségét befolyásoló tényezők a következők:

- a szoftver tervezés és implementálás technikai faktorai:
  - a modulok egymástól való függetlensége
  - a programozási stílus
  - a tervezés ellenőrzése
  - a program tesztelése
  - programozási nyelv választás
  - szabványok, szabványos szoftver eszközök használata
  - a dokumentáció minősége
- a szoftver függősége a külső tényezőktől
- a működtető hardver stabilitása
- a szoftver várható élettartama
- a szoftvert támogató team folytonossága, állandósága, stabilitása .

2.7 A SZOFTVERTERMÉKEK KOMPONENSEI

A szoftvertermékek komponensei (2.4 ábra):

- adat,
- processz és
- reláció.



2.4 ábra: a szoftver komponensei

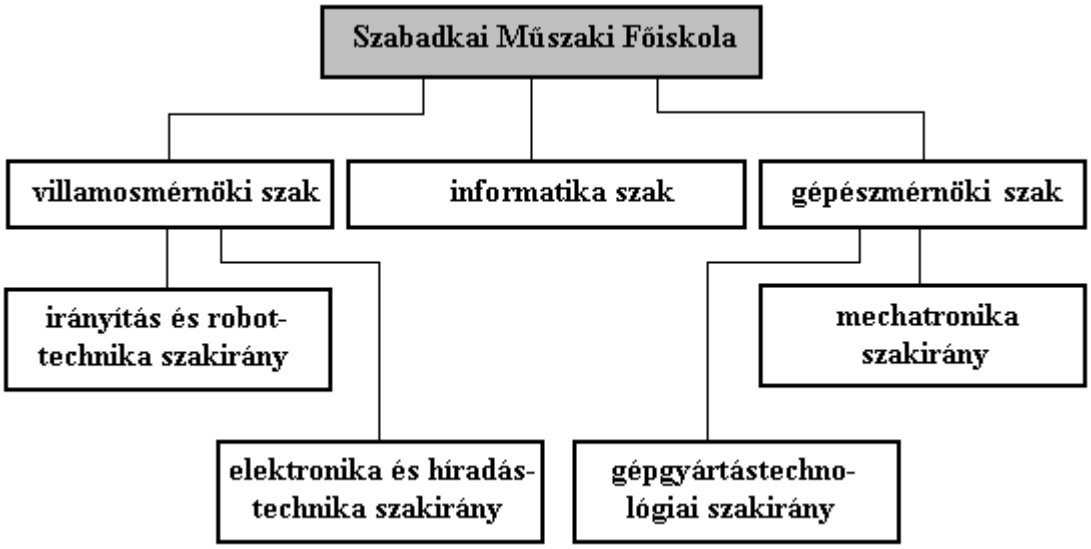
2.7.1 Adatfolyam diagram

Az adatfolyam diagram leírja

- a vizsgált rendszer be- és kimenõ adatait,
- a rendszerben tárolt adatokat,
- az egyes adatokon végzett műveleteket.

2.7.2 Blokk diagram

Blokk diagramot akkor használunk, mikor egy programrendszerben csak az adatok egymástól függõségét akarjuk szemléltetni, közömbös a az illetõ adatok tartalma, formátuma. Például a Szabadkai Műszaki Főiskola oktatási hierarchiája látható a 2.5 ábrán.



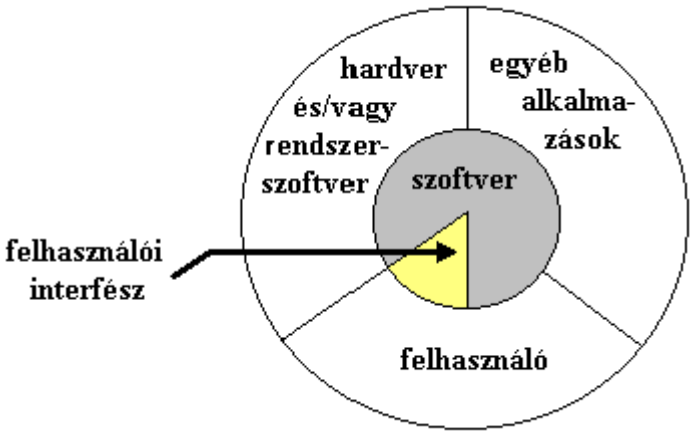
2.5 ábra: a Szabadkai Műszaki Főiskola oktatási hierarchiája

2.8 A SZOFTVER TERVEZÉSI FÁZISAI

A külvilággal való kapcsolat szerint három szoftvertervezési fázis különböztethető meg:

- *külsõ* (interface) fázis,
- *architekturális* fázis
- és *részletes* tervezési fázis.

A külsõ tervezésnél külvilágnak a működtetõ hardvert és/vagy rendszer szoftvert, egyéb szoftvereket, valamint a felhasználót tekintjük. Ezek a külsõ tényezõk a 2.6 ábra szerint állnak kölcsönös kapcsolatban.



2.6 ábra: a külső tényezők és a szoftver

Az ábrán **felhasználói interfésznek** (MMI – man-machine interface) nevezett rész igen fontos, ugyanis a felhasználó ezt látja a szoftverből, ennek a résznek a

hiányosságai teszik nehézkessé, megbízhatatlanná az egyszerű szoftverhasználatot. A más alkalmazásokkal, illetve szoftverekkel és hardverrel való összehangolás csak műszaki kérdés, ugyanakkor nem látványos része a szoftverműködésnek. Az itt fellépő hibák is a felhasználói interfészen keresztül válnak láthatóvá.

MMI – man-machine interface: ez kapcsolja össze a felhasználót (az embert) a számítógéppel, különösen a be- és kiviteli eszközökkel. Az interaktív működést egyre bonyolultabb és látványosabb eszközök biztosítják (mikrofon, hangszóró, lapolvasó stb.).

2.8.1 A felhasználói felület tervezése

A felhasználói felület tervezésénél fontos:
a felhasználók körének ismerete, az interfész a felhasználó gondolkodásmódjához és nyelvi terminológiájához kell hogy alkalmazkodjon, nem mindegy, hogy az adott rendszert mely munkakörben dolgozók fogják használni, az adott terület szakkifejezései miatt, de fontos a felhasználók gyakorlatának figyelembe vétele is (a 2.7 ábra példa egy lehetséges felosztásra, csoportosításra),
a teljesítményre vonatkozó követelmények meghatározása, például ha a szoftver gyors kell hogy legyen, az interfész működése nam lassíthatja azt,
a program használata során alkalmazott funkciók csoportosítása, a leggyakrabban használt funkciókat funkcióra emlékeztetõ parancsokkal kell ellátni, a veszélyes funkciókat kiváltó akciókat mindig többszöri ellenőrzéssel kell elláti,
alapértelmezett értékek használata célszerű,
egyes funkciók összekötése, például a programból való kilépés automatikusan tárolási feladatot is hajtson végre,
kezdő felhasználóknak más, gyakorlott felhasználóknak megint más kezelési módszert engedélyezni (pl. Windows-nál, egér és billentyű),
hiba jelentkezéséről a felhasználó amellett, hogy tudomást kell hogy szerezzen, meg kell engedni, hogy döntsön a következő lépésekről,

2.7 ábra: a szoftverfelhasználók körének felosztása

3. A PROGRAMOZÁS MÓDSZERTANA

Ebben a fejezetben az előző fejezetekben megismert módszereket, technikákat lehet megtalálni összefoglalva.

Az 1960-as évek végéig monolitikus programozás. jellemzői:

- Egy programozó egy program és
- a programoknak nincs szerkezete.

A jó program legfontosabb kritériumai:

- jól áttekinthető szerkezete,
- jól dokumentált és
- bebizonyítható módon azt csinálja, ami a feladata.

A megoldások:

- moduláris programozás,
- a megoldandó feladat komlexitásának csökkentése és a
- gyakorlat alakította ki (heurisztikus).

Két féle tervezési módszer létezik:

- Top-Down dekompenzáció, a feladatot részfeladatokra bontjuk, majd azokat további részfeladatokra, míg kezelhető méretu részproblémákhoz nem jutunk és
- Button-Up kompozíció, a részfeladatokból kell összeépíteni a programot, nincs módszer a modulok összefűzésére, illetve annak bebizonyítására, hogy a modulok együtt jól fognak dolgozni.

A moduláris programozás előnyei:

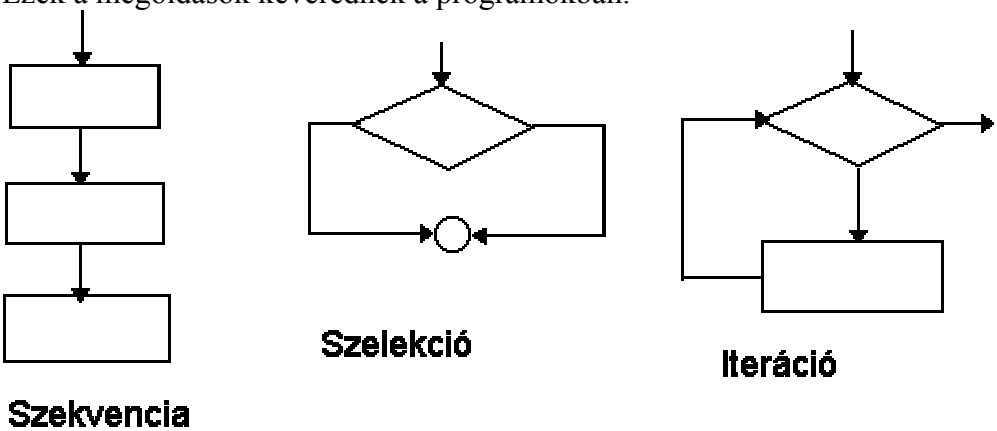
- a részprogramok könnyen áttekinthetők,
- könnyebben megírható,
- könnyebben tesztelhető,
- több modul írható egy időben (párhuzamos problémamegoldás),
- könnyebben javítható,
- a modulok szabványosíthatók,
- modulkönyvtárakban tárolhatók és
- újrafelhasználhatók.

A struktúrált programozás matematikai alapokon nyugszik.
Dijkstro kialakította a hiearchikus programozás módszerét. Ez a Top-Down dekompozíciót egészíti ki:

- az eredeti feladat részfeladatra bontásával kialakítható egy absztrakt program,
- finomítással csökken az absztrakció,
- további finomításokkal egy konkrét gép konkrét utasításkészletéig jut a megoldás, melyeket tesztekkel lehet ellenőrizni.

Az itt használható három programstruktúra a 3.1 ábrán látható, ahol megkülönböztetünk:
soros (szekvencia),
elágazásos (szelekció) és
ciklikus (iteráció) szerkezetet.

Ezek a megoldások keverednek a programokban.



3.1 ábra: a háromféle programszerkezet

Az objektumorientált programozás (OOP) kialakulása a 70-es évekre tehető. 1969-ben Alen Key dilplomamunkájában jelenik meg az egér, ikon, ablak és menürendszer. 1972-ben konkrét elképzelések jelennek meg az OOP-re.

Az OOP-re jellemző, hogy:

- az adat és a funkcionális modell nem elválasztható, az adatok és a rajtuk végrehajtható műveletek elválaszthatatlanok és
- ezek is algoritmikus nelvek.

Az objektumorientált nyelv tulajdonságai:

objektum, ami a változó általánosítása:

- az állapotait (atributumok) tetszolegesen komplex adatok irják le (adatelemek + szerkezet),
- a viselkedését módszerek írják le (fügvények és eljárások),
- az objektum-állapotok lekérdezhetők (mi az értéke egy-egy objektumnak),
- egyik állapotból a másik állapotba léteznek átvivő módszerek (értékváltoztatás),
- egy objektum csak önmagával azonos és minden máik objektumtól különbözik (azonosság),
- megállapítható, hogy azonos állapotban vannak-e,
- megállapítható, hogy ugyanarról az objektumról van-e szó,

osztály, a típus fogalmának általánosítása:

- azonos attributumú és módszerű objektumok együttese az osztály,
- az objektumok az osztály példányai,

bezárás, vagyis a hatáskör fogalmának általánosítása:

- olyan eszközrendszer, mely segítségével megmondható, hogy az osztály attributumaiból és módszereiből kívülről mi látszik,
- absztrakció, mely során a specifikáció és az implementáció elválik, de ez szabadon előírható,

öröklés, vagyis az újrafelhasználhatóság csúcsa:

- szuper osztály elsődleges a kapcsolatban, alosztály csak hozzá kapcsolódhat,
- alosztály:
  - örökli a szuperosztály attributumait és módszereit,
  - új attributumokat és módszereket definiálhat ezek mellé,
  - elhagyhat belőlük,
  - átnevezheti az atributumokat és módszereket,
  - megváltoztathatja a láthatósági viszonyokat,
  - felülbírálhatja a módszerek implementációit,
- lehetőségek:
  - egy alosztálynak egy szuperosztálya lehet (1-szeres öröklodést támogató nyelvek), a hiearchia alakja egy **fa**, egy alosztálynak több szuperosztálya lehet (többszörös öröklodést támogató nyelvek), a hierarchia elkja egy **gráf**,

polimorfizmus, vagyis többalakúság:

- példány polimorfizmus (egy konkrét háromszög példánya a háromszög osztálynak is és a poligon osztálynak is,
- módszer polimorfuzmus, az alosztály újraimplementálhat egy módszert kérdés melyik kód fut le, ezt a kötés mondja meg,

kötés:

- korai kötés, fordításkor eldől, hogy a meghívott módszerhez melyik kód tartozik,
- késői kötés, futás közben dől el, hogy a módszerspecifikációhoz melyik kód tartozik,
- minden példány tudja melyik osztály közvetlen példánya (aktuális példány),
- minden példány egy jól definiált példány aktuális példánya,
- előny, pl. ha egy módszer (A) hív egy másikat (B) a másikat újraimplementálom és meghívom (A)-t, ez esetben az újraimlementált fut le az első esetben az eredeti,

üzenet (az objektumok közötti kapcsolat):

- osztályokat definiálása,
- az öröklodési hiearchiák elhelyezése,
- osztályokon belüli objektumok származtatása,
- a program futása közben az objektumok működnek, hatnak egymásra, üzeneteket küldenek egymásnak, válaszolnak az üzenetekre,
- az üzenet nem más, mint egy módszer meghívása (eljárás vagy függvényhívás),

Az objetumorientált nyelvek lehetnek:

tiszta nyelvek:

- csak az objektumorientált eszközrendszer van,

- van egy standard osztályhiearchia, a programozó mindig ezt a hiearchiát bővíti, ezekhez fűz objektumokat,
- minden eszköz objektum (az osztály, a módszer és az attributum is)
- Smalltalk, Eiffel,

Hibrid nyelvek:

- eljárásorientált nyelvek objektumorientált eszközrendszerrel kiegészítve,
- C++,

Java nyelv a tiszta és hibrid nyelvek között helyezkedik el.

A TP6.0 néhány jellegzetessége:

- osztályt csak a legkülső szinten lehet definiálni, csak önálló osztályokat lehet definiálni, egyszeres öröklődés lehetséges (Object(szuperosztály név),
- az osztályt típusként értelmezi (TYPE név = object),
- az eszközrendszert a rekordon építi fel, a rekord lehet:
    - attributm rekordmező
    - módszerspecifikáció rekordmező

4. ASSEMBLY PROGRAMOZÁS

4.1 BEVEZETÉS

A különféle mikroprocesszorok, mikrovezérlők mind saját, hardverhez kötött belső, alacsonyszintű utasításkészlettel rendelkeznek. Ez az utasításkészlet adott, kizárólag az ebben levő utasításokat tudja a processzor végrehajtani. Az utasítások gépi kódok, melyek 0 és 1 számok kombinációjából tevődnek össze. Az egyes utasítások egymás után elhelyezve adják a programot, mely a számítógép operatív memóriájában helyezkedik el végrehajtásakor, illetve a háttértárakon tárolás céljából.

Egy program összeállítása (megírása) gépi kódban rendkívül bonyolult, gyakorlatilag kivitelezhetetlen. Ennek oka az, hogy egy adott gépi kódú utasítás nehezen megjegyezhető, valamint nehezen írható le. Célszerűbb bevezetni egy olyan szimbólikus leírási módot, ahol könnyen megjegyezhető rövidítések jelölik az utasításokat, valamint a hardver elemeit. Ezeket a mnemonikus kódokat egy segédprogram, az asszembler fordítja le gépi kódra. Itt kell megjegyezni, hogy a magasszintű programozási nyelveken megírt programok is végül gépi kóddá alakulnak át a fordítóprogram (compiler) segítségével. Külön csoportba sorolhatók azok a magasszintű nyelvek, amelyek interpreteresek, itt nem történik fordítás.

Feltehető a kérdés, mi az oka annak, hogy tanulni kell az assembly programozást, akkor, amikor ma már igen könnyen kezelhető, nagy hatékonyságú objektum orientált programnyelvek állnak rendelkezésünkre? A magasszintű programozási nyelvek nem használják ki optimálisan egy számítógép hardver képességeit. Sokszor a mikrogépek, mikrovezérlők, tehát kisebb rendszerek legfeljebb 20 % -ban hatásosak. Amikor nincsenek különleges igények, akkor ez nem probléma, de ha nagy válaszidő-igény jelemtkezik, vagy kis rendszereknél kicsi az operatív memória kapacitása, akkor fontos az assembly használata. Egy példa, PC számítógépen számoljunk el a képernyőn 0-tól 65535-ig. A 4.1 táblázatban láthatók az eredmények. Innen látszik, hogy különlegesen gyors válaszigény esetén csak asszemblerben tudunk dolgozni, ezek különlegesen gyors adatátvitelek, játékprogramok.

4.1 táblázat: különböző programnyelveken megírt program összehasonlítása

programnyelv	a program hossza [bájt]	futási idő [sec]
Borland Turbo C 2.0	8994	2.4
Borland Pascal 6.0	3888	0.5
Assembly	80	0.15

4.2 ASSEMBLY PROGRAMOZÁSHOZ SZÜKSÉGES HARDVER ISMERETEK

Amikor assembly programozást használunk, akkor teljesen tisztában kell lenni a processzor, memóriák, ki- és bimenő egységek, tehát a mikroszámítógép, mikrovezérlő felépítésével, hiszen a gépi kódú utasítás köti össze a programozási szintet a hardverrel. Minden mikroszámítógépnek más és más a hardver felépítése, valamint a hardvert működtető utasítások alakja. Ennek ellenére vannak hasonlóságok

a különböző megoldások között, úgyhogy egy mikroprocesszor megismerése már igen jó alap arra, hogy egy másik processzort könnyebben ismerjünk meg.

Melyik processzorral ismerkedjünk meg először? Természetesen azzal, amelyikhez a legkönnyebben hozzáférünk, ez pedig a minden személyi számítógépben, az IBM PC-ben, vagy azzal kompatibilis gépben megtalálható Intel 8086-os processzor (vagy annak továbbfejlesztett változata).

4.3 IBM PC ÉS KOMPATIBILIS SZEMÉLYI SZÁMÍTÓGÉPEK

4.3.1 Bevezetés

A mikroprocesszor 1974-es megjelenése lehetővé tette a számítástechnika igen gyors, tömeges elterjedését. Igen rövid idő alatt az ipari vezérlő-, irányító-, adatgyűjtő berendzések nélkülözhetetlen építőelemeivé váltak ezek az eszközök. Nagyon gyorsan igény jelentkezett személyi számítógépek kifejlesztésére, melyek ára alacsony, így akár irodai, akár otthoni használatra is a széles rétegek számára elérhető a berendezés. A kezdeti kísérleteknél 8 bites processzorokat építettek be a számítógépekbe, a hardver aránylag egyszerű volt, de mindig specifikus. Jelentkezett egy másik nagy probléma, ami az olcsó háttértárak hiánya volt. A ZX Spectrum, Atari vagy Commodore a maga idejében forradalmian új berendezés volt alcsony áron, de mindegyik fejlesztés hibája a nem továbbfejleszthető hardver és szoftver.

Az 1981-ben megjelent IBM PC XT gyártmányú személyi számítógép (PC – Personal Computer) döntően meghatározta és ma is meghatározza a számítástechnika fejlődését. Elérhető áron egy nyitott (könnyen továbbfejleszthető, bővíthető) hardverrel és szoftverrel rendelkező rendszer született és fejlődik állandóan. Az Intel 8088 (illetve a vele teljesen kompatibilis 8086) processzorral megépített számítógép az otthoni, irodai és ipari alkalmazások nélkülözhetetlen eszköze. Döntően megszabta a mikroprocesszorok, merevlemezes háttértárak, az Internet és még sok minden hihetetlen méretű továbbfejlődését.

Az IBM PC gépekre jellemző tulajdonságok:

- nyitott hardver és szoftver felépítés, vagyis az igényeknek megfelelő egységek, programok rendszerbe építése, ami a gép legkülönbözőbb területeken való hasszználatát teszi lehetővé egy viszonylag egységes magra támaszkodva,
- szabványos, piacon kapható elemekre épülő hardver, ami egyszerű tervezést, kivitelezést tesz lehetővé,
- az újabb gépek lefelé szoftverkompatibilisak, vagyis a régebbi típusokra megírt programok futnak az újabb gépeken,
- az IBM nem szabadalmaztatta a gépet, ami lehetővé teszi a bárki által való gyártását, ezzel versenyhelyzetet hozva létre a piacon.

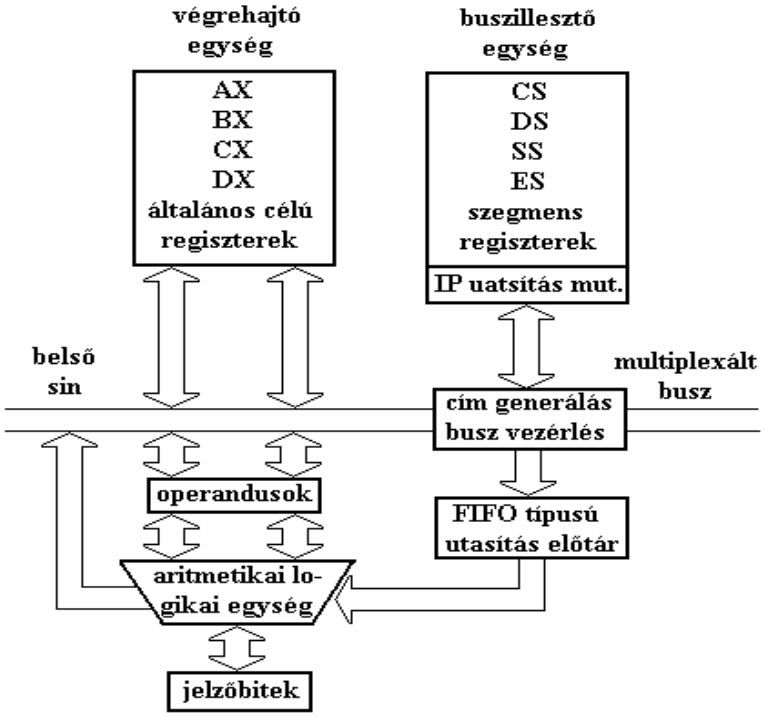
Meg kell említeni a PC-k azon tulajdonságát is, hogy ipari berendezésekben is előszeretettel használják alacsony ára és gyors működése, szabadon kialakítható hardver-szoftver konfigurációja miatt.

4.3.2 A 8086, 8088 mikroprocesszor belső felépítése

Több más komoly cég fejlesztési stratégiájával ellentétben az Intel fejlesztési staratégiájának része volt mindig a kompatibilitás, vagyis a hardver és szoftver azon tulajdonsága, hogy a technika, technológia fejlődése ne olyan új megoldások kifejlesztését eredményezze, melyeknél a régebbi berendezések, eszközök teljesen használhatatlanná válnának. Ennek az elvnek köszönhetően az Intel gyártmányok ma is nagyban uralják a piacot, ezzel a cég döntően meghatározza a fejlődés mai és jövőbeli irányait.

4.3.2.1 Az Intel 8086 processzor felépítése

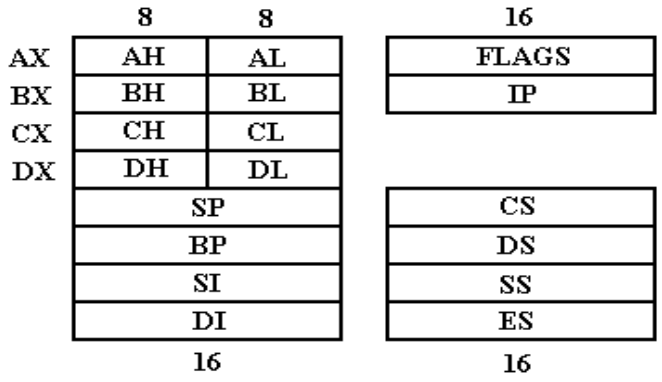
Ennek a processzornak 2 változata létezik, a 8 bites külső sinnel rendelkező 8088-as és a 16 bites 8086-os, melyeknél teljes a szoftverkompatibilitás. A 4.1 ábrán látható a processzorarchitektúra.



4.1 ábra: Az Intel 8086 processzor architektúrája

Az ábrán szereplő végrehajtó egység (EU – Execution Unit) és busz interfész egység (BIU – Bus Interface Unit) adatcserét végez egymással, de feladatuk teljesen más. A BIU feladata a külvilággal való kapcsolattartás.

A 4.2 ábrán az Intel 8086 processzor legfontosabb regiszterei láthatók.



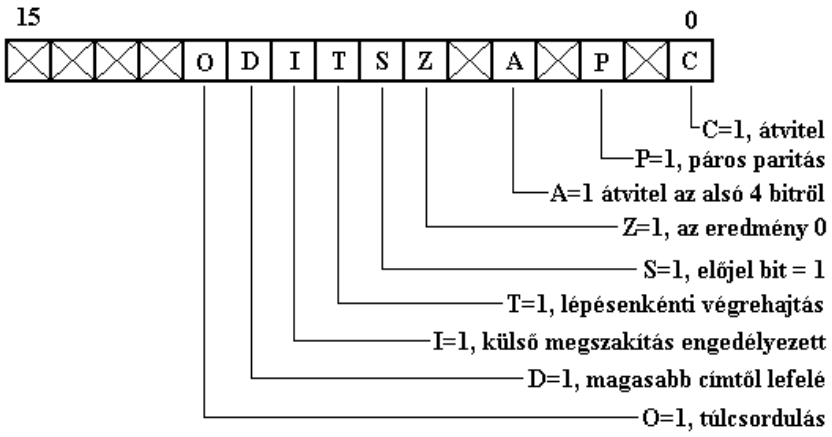
4.2 ábra: Az Intel 8086 legfontosabb programozáshoz szükséges regiszterei

Ez a 16 bites architektúra megörökölt néhány 8 bites gépnél is használt megoldást, pl. a 16 bites akkumulátor 2 8 bitesként is használható, vagy a memória címzésére itt is 16 bites regiszterek szolgálnak, de érdekes megoldásokkal kilépett a 8 bites struktúra korlátai közül.

A processzor regiszterei a következők:

- AX – általános célú 16 bites akkumulátor, használható mint 2 8 bites akkumulátor is (AH és AL),
- BX – 16 bites bázis (base) regiszter, lehet mint 2 8 bites regisztert is használni (BH és BL),
- CX – 16 bites számláló regiszter (count), mint 2 8 bites regiszter is használható (CH és CL),
- DX – 16 bites adatregiszter (data), mely két 8 bites regiszterként is használható (DH és DL),
- SP – 16 bites veremtár mutató regiszter (stack pointer),
- BP – 16 bites bázismutató regiszter (base pointer),
- SI – 16 bites forrásindex regiszter (source index),
- DI – 16 bites célindex regiszter (destination register),
- FLAGS – 16 bites regiszter, mely részletesen a 3. ábrán látható,
- IP – 16 bites utasítás mutató regiszter (instruction pointer),
- DS – 16 bites adatszegmens-regiszter (data segment), az adatók érhetők el ezzel a regiszterrel,
- CS – 16 bites kódszegmens-regiszter (code segment), programutasítások érhetők el ezen a regiszteren keresztül,
- SS – 16 bites veremtár-szegmens regiszter (stack segment), a veremtár címzésére szolgál és
- ES – 16 bites különleges memória regiszter (extra segment), egyéb memóriaterületek területek érhetők el ezen a regiszteren keresztül.

A 4.3 ábra a műveletekkel vezérelt jelzőbiteket mutatja.



4.3 ábra: A jelzőbitek (FLAGS) regiszter bitjei

4.3.2.2 A 8086 mikroprocesszor működése

Mint a 4.2 ábrán látható, a processszorban két egység található, az EU, vagy végrehajtó egység, illetve a BIU, vagy buszillesztő egység.

A **BIU** a **FIFO** (First In First Out – az először beírt adatot veszi ki először) regiszterblokkba előre beolvassa az utasításokat az operativ memóriából, amit **utasításlehívásnak** nevezünk. Az EU végrehajtó egység ebből a FIFO memóriából veszi ki ezek után az utasításokat és az operandusokat, így nincs szükség kivárni a memóriahozzáférés idejét. A BIU másik feladata az operandusok, események továbbítása, átvitele az EU, a memória és az I/O eszközök között. Ezt a feladatot úgy tudja végrehajtani, hogy létrehozza a fizikai címeket egy beépített összeadómű segítségével. Ez a mechanizmus azért lényeges, mert így lehet nagy sebességű processzorok mellett viszonylag lassú memóriákat használni.

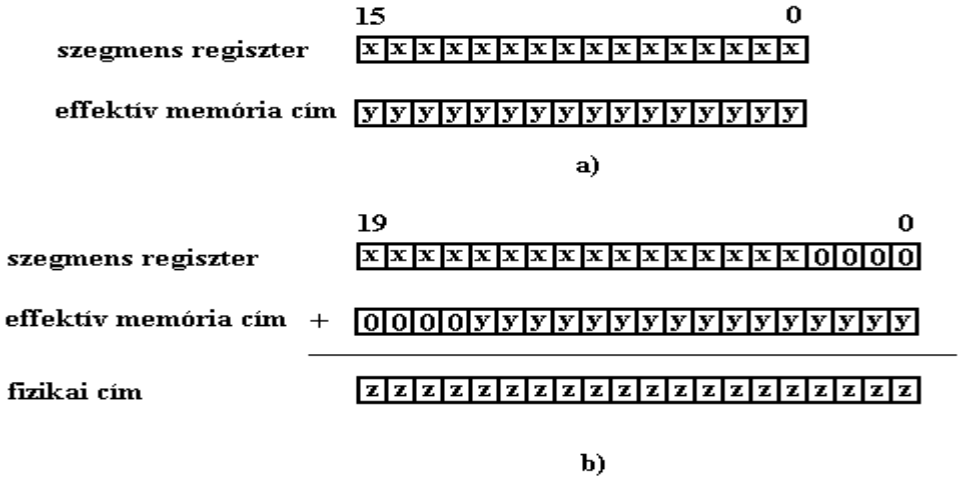
Az EU, a végrehajtó egység az általános célú regeszterekből, az ALU aritmetikai-logikai egységből, az operandusregiszterből és flag, vagy jelzőbitregiszterből áll. Ez az egység értelmezi és hajtja végre a BIU által lehívott utasításokat, valamint adja át az adatokat a BIU-nak.

Összehasonlítva az első 8 bites gépekkel az először nagyobb méretekben elterjedt Intel 8086 16 bites gépet, megállapíthatjuk, hogy kb. négyszeres órajelfrekvecia a hatékonyságban már mint tízszeres érték jelentkezik. Ennek oka nemcsak abban keresendő, hogy 8 helyett 16 bites adatátvitellel, ALU-val stb. dolgozik a rendszer, hanem a fent ismertetett **átlapolásos** üzemmóddal. Ez a **pipeline** technológia, mely megalapozta a **RISC** struktúrák kialakulását.

RISC (reduced instruction set computer): csökkentett utasítás-készletû számítógép. Mivel a tapasztalat az, hogy az utasítások egy részét keveset, vagy egyáltalán nem használják a programozók, ezért hoztak részre olyan gépeket, amelyeknek optimalizált, kevés utasításból álló készlete van.

4.3.2.3 A 8086 címzési technikája

A gépnél alkalmazott 16 bites címregiszterek nem teszik lehetővé 64 kB-nál nagyobb memória közvetlen címzését. Ezt a problémát egy egyszerű technikával lehet áthidalni, mely a szegmens regiszter tartalmából és egy effektív memóriacím összegéből határozza meg az effektív, fizikai memóriacímet. Ez a megoldás látható a 4.4 ábrán.



4.4 ábra: A fizikai cím meghatározása szegmensregiszter és cím összegeként

A fizikai cím kiszámítása úgy történik, hogy az adott szegmensregiszter tartalmat balra ellépteti a rendszer 4 bittel, tehát szorozza 16-tal (16 x 64 kB = 1 MB), majd az effektív memória címet hozzáadja, a legnagyobb helyértéken, 4 biten 0 kerül.

A szegmensregiszterek tartalmára nincs semmilyen megkötés, így tetszés szerinti szegmensregisztertartalom átlapolódást lehet használni.

A 8086 címzései a következők:

- programmemória címzés és
- adatmemória címzés.

4.3.2.4 A programozási címzési módok

A kiszámított fizikai címről a processzor bekéri az utasítást, az operandusokkal, címekkel együtt, ez a **fetch** eljárás, a **PC** programszámláló (program counter) értéke ezután annyival nő, amennyi az utasítás hossza volt (**CISC** struktúra miatt az utasítások hossza változó), így az újabb utasítás kezdetére, a műveleti kódra mutat. Minden esetben, amikor nem következő címről kell az utasítást bekérni, vagyis ugrásra, alprogramhívásra került sor, a következő módon számolja ki a processzor az új cím helyét:

CISC (complex instruction set computer): összetett utasítás-készletû számítógép. Viszonylag nagy számú utasítást (több mint 200) használó gép, ahol az utasítások végrehajtási ideje is különböző.

- program-relatív, szegmensen belüli címzés, ahol a szegmensen belül egy az utasításban közvetlenül szereplő 8 vagy 16 bites előjeles adat hozzáadódik PC tartalmához,
- direkt címzéssel, szegmensek közötti címzéssel, amikor az utasításban mint adat foglal helyet a cím, ez PC-be, illetve a szegmensregiszterbe töltődik,
- indirekt címzéssel, amikor egy adatot a JUMP (ugrás), vagy CALL (szubrutinhívás) memóriacímként értelmez.

4.3.2.5 Adatmemória címzési módok

Eltérően a 8 bites mikroprocesszoroktól, de akár a RISC struktúrájú gépektől a 8086 igen sok, változatos címzési móddal rendelkezik, amik a következők:

- közvetlen címzési mód,
- direkt címzési mód,
- direkt, indexelt címzési mód,
- implicit címzési mód,
- bázis relatív címzési mód és
- stack címzési mód.

4.3.3 80286, 80386, 80486, Pentium, Pentium II és Pentium III processzorok

A technológia fejlődése az újabb típusú mikroszámítógépeknél mindig nagyobb működési sebességet biztosít az előző típusokhoz képest, amit nemcsak órajelfrekvencia-növeléssel érnek el, hanem párhuzamos feldolgozással, gyorsítómemóriákkal. Az Intel processzorok fejlesztésénél az alapelv az, hogy a későbbi típusok képesek a régebbi, egyszerűbb típusokra fejlesztett szoftver futtatására. Így csak érdekességként említjük meg, hogy az újabb 32 bites processzorok természetesen 32 bites általános célú regiszterekkel rendelkeznek, amikor 16 bites szoftverrel dolgoznak, akkor csak az alsó 16 bitet használják.

4.3.4 Az IBM PC sinrendszere

4.3.4.1 Bevezetés

Az IBM PC alaplap a szükséges mikroelektronikai elemek (mikroprocesszor, memória, I/O eszközök stb.) mellett tartalmaz csatlakozókat (slot), melyek az IBM PC mikroszámítógép külső buszának jeleit tartalmazzák. Több fajta csatlakozó használatos, szerepük akkor van, ha a számítógép hardverét szeretnénk új egységekkel bővíteni, pl. EPROM programozóval, MODEM-mel, A/D, D/A kártyával stb. Természetesen általunk fejlesztett eszközök is csatlakoztathatók a rendszerhez ezen a sinen keresztül, ilyenkor a laptervezéskor, illetve megépítésekor tiszteletben kell tartani a különféle elektromos, mechanikai, illesztési előírásokat.

A mikroszámítógép buszrendszere több tucat vezetékből áll, melyeken címek, adatok, parancsok jelenhetnek meg.

A kiegészítő lap tervezésekor a következő feladatokat kell megoldani:
eszközök kijelölése, ez a címvezetéken érkező cím azonosítása, felismerése,
adatátviteli irány meghatározása (rákapcsolódás az adatvezetékre és lekapcsolódás az adatvezetékről) és
az adatátvitelben szereplő eszközök szinkronizálása.

IBM PC mikroszámítógépek esetén két sinrendszer létezik:

- belső és
- külső.

4.3.4.2 IBM PC belső sinredszer (busz)

Ez a sinrendszer az alaplap tulajdonságát határozza meg, melyet az elérni kívánt teljesítmény szab meg. Három féle kialakítása lehetséges:

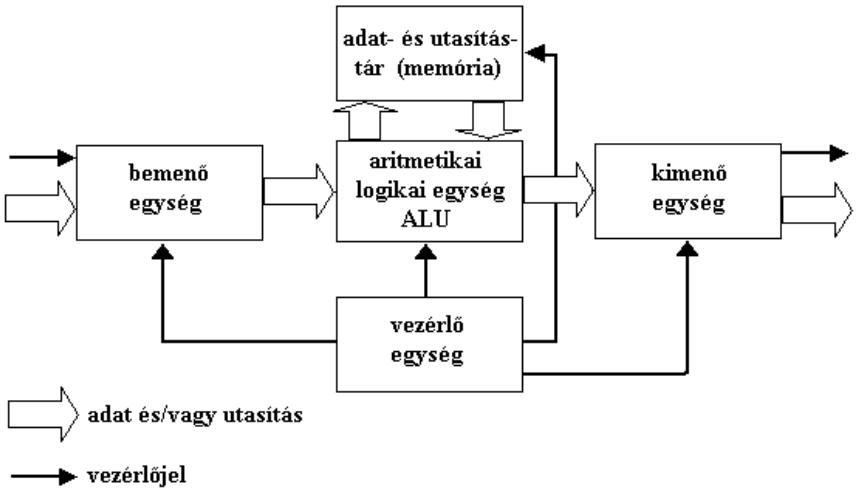
- nagyobb sebességeknél 3-sines rendszer használatos, ahol a címsin mellett külön adatsin van írásra és külön adatsin olvasásra, így, ha lehet azonos idejű adatolvasás és adatírás alakítható ki,
- 2-sines rendszer, vagyis külön címsin és külön adatsin található a legtöbb gépnél és
- közös adat- és címsin olyan rendszereknél, ahol egyszerű célfeladatok létrehozása szükséges.

A belső sinrendszerhez hardver eszközökkel nem férhetünk hozzá, ez adott az alaplapoknál.

4.3.5 A számítógép felépítése a programozás szempontjából

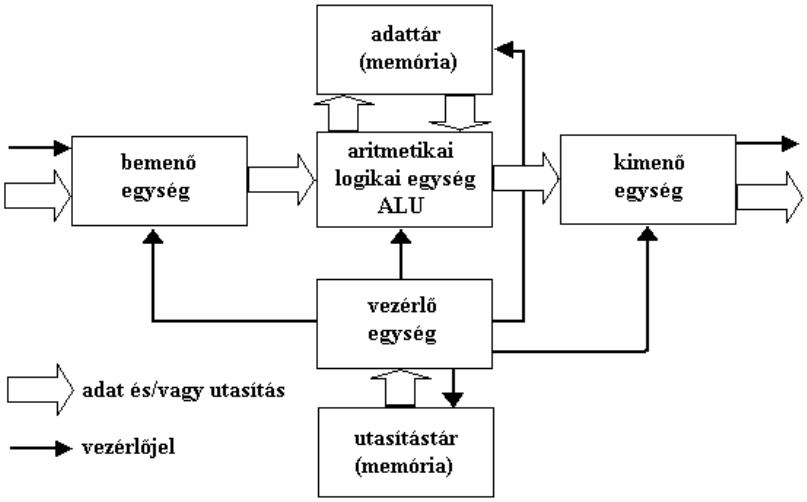
Akkkor, amikor programozni szeretnénk egy mikroszámítógépet assembly nyelven, szükségünk van bizonyos hardver alapismeretekre, a rendszer felépítésére. Ezek az ismeretek egyszerűbbek, mint a hardver tervezésekor szükségesek.

A Neumann-féle számítógép (4.5 ábra) egy vezérlőegységből, valamint operatív memóriából áll, a megfelelő input-output egységekkel kielégítve. Az operatív memória az adatokat és programot tárolja. A vezérlő egység egyenként, sorban értelmezi és végrehajtja az utasításokat.



4.5 ábra: Neumann-féle számítógépmodell

A 4.6 ábrán látható Harvard-féle számítógép nagyban hasonlít a Neumann-géphez, csak itt az operatív memória két egymástól teljesen elválasztott részből áll, a programmemóriából, illetve adatmemóriából.



4.6 ábra: Harvard-féle számítógépmodell

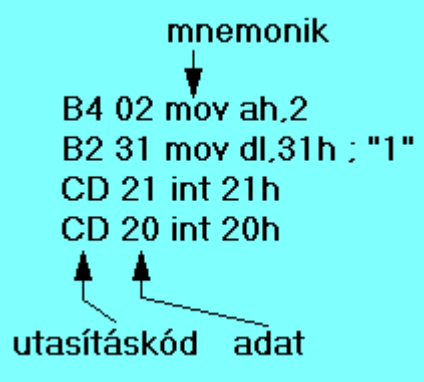
A 4.5 és 4.6 ábrán látható modelleknél a vezérlő egység (CU – Control Unit) és az aritmetikai-logikai egység (ALU – Arithmetical and Logical Unit) a központi egység (CPU – Central Processing Unit). Az aritmetikai-logikai egység, mint ahogyan a neve is jelzi különböző aritmetikai (összeadás, kivonás, szorzás és osztás) és logikai (ÉS, VAGY, NEM és KIZÁRÓ VAGY) műveletek végrehajtását teszi lehetővé, a központi egységben elhelyezkedő különböző regiszterek segítségével. Ezeket a regisztereket gyorsabban éri el az aritmetikai-logikai egység, mint az operatív memóriát és háttértárakat.

4.4 EGYSZERŰ ASSEMBLY PROGRAMOK

Anélkül, hogy bövebben magyaráznánk, végezzük el a következő egyszerű feladatot. A **PC** számítógépen váltsunk át **DOS** üzemmódba. Ezt a következő módon tehetjük meg: a START menüben kijelöljük a PROGRAM részt, megkeressük a DOS szimbólumot és rákattintunk. A gép a WINDOWS operációs rendszert elhagyja és átvált DOS operációs rendszerre. Itt a főkönyvtárból nyissunk egy **ASS** nevű alkönyvtárat, ahol dolgozunk (d:\ ENTER és md ASS ENTER és cd ASS ENTER).

Vigyük be az ELSO.EXE programba a következő néhány számot : 180, 2, 178, 49, 205, 33, 205, 32. Ezt a **copy con elso.exe ENTER** utasítással tehetjük meg, úgy, hogy az ALT billentyű folyamatos tartása mellett beírjuk a számot, ALT elengedése beírja a kódot, azzal, hogy a képernyőn mindenféle karakter megjelenik. Az utolsó szám bevitele után CTRL Z utasítással zárjuk be a fájlt, ami a D:\ASS alkönyvtárba másolja. Ellenőrizzük az elso.exe fájl meglétét a DIR paranccsal, a type elso.exe DOS parancs pedig a beírt kódokat kilistázza a képernyőre, ami számunkra érthetetlen jelek sorozata.

Ez a programunk gépi kódja, ezek után írjuk be az **elso** programnevet. Ha mindent jól csináltunk, megjelenik egy **1** szám a képernyőn.
Valójában közvetlenül PC gépi kódot töltöttünk egy állományba, ami mnemonikus (tehát szimbólikus) kódrendszerben a következő:



4.7 ábra: Assembly program az ’1’ szám kiírására

A programban, mivel assembly programozásnál áttekinthetőbb a rendszer általában hexadecimális alakban adjuk meg a számértékeket.

4.4.1 A DEBUG program

Érdemes megismerkedni a DEBUG programmal, hiszen egyszerűen és gyorsan lehet vele létrehozni rövid assembly programokat. Indítása a debug fájlnévvel történik, ami után egy ’-’ (godolatjel), a prompt jelenik meg. A ’?’ (kérdőjel) és ENTER begépelése kiadja a lehetséges parancsokat és azok formai megadását.

Az ’a’ parancs az asszembler indítása, ami hatására kiíródik a szegmenscím és a program helye a memóriában. Itt kell megadnunk a mnemonikus kódokat az operandusokkal együtt, ami ennél a programnál csak hexadecimális érték lehet (a

valódi asszemblernél szimbólikus címekkel is dolgozhatunk). Az ENTER után a következő sor beírása lehetséges, egészen a CTRL C befejező parancsig. A műveletsor a 4.8 ábrán látható.

```
164C:0100 mov ah,02
164C:0102 mov dl,31
164C:0104 int 21
164C:0106 int 20
164C:0108 ^C
-g
1
Program terminated normally
```

4.8 ábra: Első példaprogramunk beírása a DEBUG program a utasításával

Az 164C cím a kódszegmens-cím, a 0100, 0102 stb. A gépi kód címe, ami most azért lép 2-vel, mert minden utasításunk 2 bájtos (1. ábra). A '-' prompt utáni g és ENTER hatására a program végrehajtódik és a képernyőre kiíródik az '1' szám, majd egy üzenetet küld a képernyőre a DEBUG, a program helyesen fejeződött be, vagyis sikeresen visszaadta a vezérlést a DEBUG-nak. Ha hibásan fejezzük be a programunkat, akkor 'elszáll' a számítógép, vagyis a mi esetünkben nem tud DOS üzemmódban tovább dolgozni, a WINDOWS visszaveszi a vezérlést.

A DEBUG alkalmas a gépi kódokból visszaállítani az assembly programot, ahogyan azt a 4.9 ábrán láthatjuk.

```
-u
164C:0100 B402          mov ah,02
164C:0102 B231          mov dl,31
164C:0104 CD21          int 21
164C:0106 CD20          int 20
164C:0108
```

4.9 ábra: a DEBUG disassembláló utasítása

A 0108 címtől kiírt (visszafejtett) program a számunkra értelmetlen, ugyanis a mi programunk a 0107 címen ér véget, a memóriában pedig az előző programok maradványai vannak.

Elemezzük, mit jelentenek az egyes sorok a programban:

- **mov ah,02** az ax akkumulátor felső 8 bitjébe 02 érték betöltése,
- **mov dl,31** a dx regiszter alsó 8 bitjébe az 1-es szám ASCII kódjának betöltése,
- **int 21** a rendszer belső

ASCII (American Standard Code for Information Interchange): a 7 bites kódok és a betűk, számok, írásjelek és vezérlő jelek közötti egyértelmű kapcsolatot adja meg. A berendezések mind ismerik ezt a kódrendszert, így egyszerű közöttük kapcsolatot teremteni. Pl. Nyomtató.

alprogramja, amely többek között egy karaktert tud kiírni a képernyőre, ha előzőleg ax regiszter felső bájtjába 2-t írunk és a dl-be a kívánt értéket írtuk be és

- **int 20** alprogram-hívás. Mely biztosítja azt, hogy a DEBUG-ból meghívott programunk visszaadja a vezérlést ismét a DEBUG-nak.

A **d** utasítás egy memóriarész tartalmát küldi ki a képernyőre hexadecimális és ASCII alakban. Számunkra, az adott programnál csak az első 8 bájt értelmes, a többi adat véletlenszerű. Vegyük észre azt, hogy az ASCII értékeknél a '**...1.!** ' sorban megjelenő 1-es szám az 1 ASCII kódja, amit mi töltöttünk be a dl regiszterbe. A '.' Akkor jelenik meg, ha a kódnak nincs megfelelő képe, pl. képernyővezérlő parancskód. Az 4.2 táblázat tartalmazza az ASCII kódokat. Meg kell jegyezni, hogy a D7 bit használata az ASCII táblázatnál megduplázza a táblázat méretét, hiszen 8 bites lesz, az így keletkezett újabb 128 helyre szintén kódok írhatók, amik már egy adott rendszernél eltérhetnek egy másik rendszer kódjaitól, így lehet ékezetes karakterekkel is dolgozni. DOS-ban az ALT billentyű folyamatos lenyomása mellet írjuk be a 144 számot, majd engedjük el az ALT billentyűt, megjelenik az É betű.

4.2 táblázat: az ASCII kódtáblázat

	000	001	010	011	100	101	110	111
0000	NUL	DLE	blank	0	@	P	'	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EDT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

A regiszterek és jelzőbítek, flagek tartalma a DEBUG-ban az r paranccsal jelentethető meg.

A p (proceed) parancs egyenként hajtja végre az utasításokat (a g paranccsal ellentétben), de emellett mindig kiírja az r parancs által is megjelenített tartalmat, a regisztereket és flageket.

Mit kell tenni akkor, ha nem az 1-es számot, hanem például az A betűt szeretnénk kiíratni? Természetesen nem kell az egész programot újból begépelni, csak a megfelelő sort kell újból írni. Az u paranccsal fejtsük vissza a programot, itt láthatjuk,

hogy a 0102 címen található a helettesítendő sor, majd írjuk be a következőt: a 0102, ekkor erre a helyre enged beírást a DEBUG. Az ASCII táblázatból keressük az A betű küdját, ami binárisan 0100 0001, vagyis 41h (illetve 65d). Így az új sor beírása a 4.10 ábrán látható. A DEBUG most is folyamatosan engedi az utasítások beírását, a CTRL C paranccsal léphetünk ki.

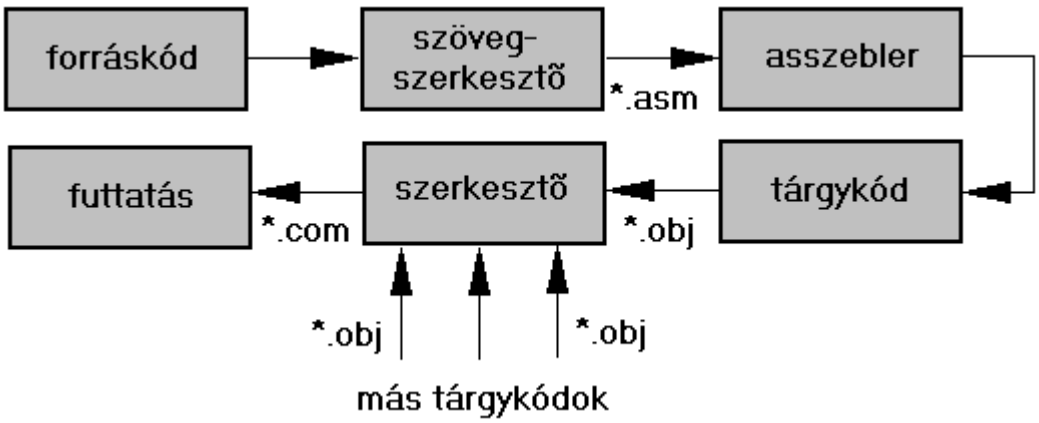
```
-a 0102
146C:0102 mov dl,41
146C:0104 ^C
-g
A
Program terminated normally
```

4.10 ábra: új érték beírása a programba

A DEBUG program segítségével sok mindent el lehet érni, közvetlenül a hardver elemeket prgramozhatjuk. Mégsem ezt a programot használjuk assembly fejlesztésre, hiszen az aránylag egyszerű kezelés nem tesz lehetővé valódi programfejlesztést. Többféle fejlesztőszoftver áll rendelkezésünkre a programok írására, vannak közöttük bonyolultak, de megtalálhatók az egyszerű változatok is. Az eltérések mellett is nagyon sok a hasonlóság a használatukban, könnyen tudunk áttérni más rendszerekre.

4.2.2 Assembly programok írása, fordítása, szerkesztése és futtatása

Az assembly programok létrehozását különböző segédprogramok biztosítják. A 4.11 ábra mutatja a program létrehozásának fázisait.



4.11 ábra: assembly program léterehozása

A forráskód, vagyis az adott processzor (esetünkben Intel 8086) utasításkészletét és a fordító utasításait tiszteletben tartva megírt program egy **ASCII** szövegszerkesztővel ***.asm** fájlba kerül a Winchesterre. Az **asm**, rövidítés utal arra, hogy asszemblerben dolgozunk. Szövegszerkesztő

***.asm:** a '*' (csillag) a DOS (és sok más) operációs rendszerben az állománynév helyén szerepel, ami mindig betűvel kezdődik, számokat is tartalmazhat és hossza legfeljebb 8 karakter.

bármely ismert program lehet, pl. a Norton Comander F4 helyén levő beépített szövegszerkesztője.

Az ***.asm** állományt le kell fordítani, amit szintén többféle szoftverrel elvégezhetünk, itt a Turbo Assemblert használjuk, ami a **TASM** nevet viseli. Egyszerűen be kell írni a **D:\ASS>\TASM *.asm** utasítást. A fordítók rendszerint két menetben (lépésben) működnek:

- a **név**, **érték** és **típus** alapján kitöltik a szimbólumtáblázatot, majd
- lefordítják a mnemonikokkal megadott utasításkódokat gépi kódokká és kitőltik számértékekkel a szimbólumtáblában található értékeket, léterhozva az ***.obj** (object) tárgykódot.

A kapott tárgykód szabványos felépítésű, ilyen kódot más programnyelvek is létrehoznak, ezzel biztosítva azt, hogy a szerkesztőprogramok több programozási nyelv ***.obj** állományait egy programmá kössék össze. Erre a feladatra is több különböző kész szoftvert használhatunk, mi a Turbo Link programot használjuk. Be kell írni a **D:\ASS\>TLINK *.obj** utasutasítást, ami végül ***.EXE** végrehajtható kódot ad. A TLINK program engedélyezi egy szünettel (blank) elválasztva több tárgykód felsorolását, ami ezen kódok összefűzését biztosítja.

Az *.exe és *.com program felépítése

A 4.12 a) ábrán egy *.EXE, míg a 4.12 b) ábrán egy *.COM programszerkezet látható. Az *.EXE program bármilyen hosszúságú lehet, a COM bele kell hogy férjen egy szegmensbe, így legfeljebb 64 kB hosszú lehet.

*.EXE szerkezet

```
Kód
        Segment
        assume CS:Kód,DS:Adat,SS:Stack
Start:  .
        .
        .
Kód
        Ends
Adat
        Segment
        .
        .
        .
Adat
        Ends
Stack   Segment
        .
        .
        .
Stack   Ends
        End     Start
```

a)

*.COM szerkezet

```
Kód
        Segment
        assume  CS:Kód,DS:Kód
        Org     100h
Start   .
        .
        .
Kód     Ends

        End     Start
```

b)

4.12 ábra: a *.EXE és a *.COM típusú fájlok felépítése

Segment jelöli a szegmens kezdetét, elötte egy tetszőleges elnevezés a szegmerns nevét, a szegmens végét pedig az **Ends** kifejezés.
Az **assume** a szegmensregiszterekbe tölti a hozzá tartozó szegmenscímet. SS nem kötelező, így elhagyható. A COM programoknál kód és adatszegmens kezdet megegyezhet..
Org adja meg a szegmensen belül a program kezdetét, 100h alatt programparaméterek találhatók.
A PC személyi számítógép BIOS (Basic Input/Output System) szoftvere tartalmaz már megírt alprogramokat, amiket előnyösen használhatunk fel. Így az int 21h utasítás is egy ilyen alprogram, melynek paramétere szabja meg a visszatérés módját, a mintapéldában a 4c00 érték a programból való üzenet nélküli vésszatérést jelenti.

4.4.3.1 - 1. Feladat

A programból térjünk vissza a DOS-ba.

```
;*****************************************************************
Prog01      Segment                         ;szegmensdefinico
            assume CS:Prog01,DS:Prog01      ;CS es DS beallitas a szegmens
                                            ;elejere
;*****************************************************************
Start:  mov ax,Prog01                       ;DS regiszter beallitasa
            mov   ds,ax
            mov   ax,4c00h                  ;kilepes a DOS-ba
            int   21h
;*****************************************************************
Prog01      Ends                            ;a szegmens vege
      End   Start
;*****************************************************************
```

A programban a ';' (pontosvessző)utáni szöveg tetszőleges, nem kerül fordításra.
A programírás lépései:
Pl. a Norton Commander SHIFT F4 utasításával elindítjuk a szövegszerkesztőt, ahova beírjuk a fenti programot,
a TASM Prog01 ENTER utasítással fordítsuk le a programot (eredménye **Prog01.obj**),
a Link ENTER után írjuk be a Prog01 fájlnevet és minden kérdésre ENTER-rel válaszoljunk, létrejön a Prog01.exe fájl,
írjuk be, tehát futtassuk a programot.

Ennél a prgramnál, ha minden rendben van, csak azt látjuk, hogy újból megjelenik az alkönyvtár promptja.

A Prog01.asm programban a **mov** utasítás háromszor is szerepel. Ez az utasítás a leggyakrabban használatos, nagyon sok változata van. Ezekből a mov utasításokból a 4.3 táblázatban találhatunk meg néhányat. Az összes lehetőség az utasításkészletben található.

4.3 táblázat: néhány mov utasítás		
mov	ax,1839	ax akkumulatorba kerül az 1839-es szám
mov	ax,3Bh	az akkumulatorba kerül a szám hexadecimális alakban
mov	al,45	ax akkumulátor alacsonyabb helyértékű al bájtjába keül 45
mov	ax,bx	ax –be kerül bx regiszter értéke, bx értéke változatlan marad
mov	al,bh	al –be kerül bh regiszter tartalma, bh változatlan marad
mov	ax,cimke	ax –be kerül a címke szegmenscímét

A mov, de más utasítás is igen sokféle kombinálást enged meg, ennek ellenére érdemes néhány alapvető szabályt megtanulni, mely szabályok más utasításoknál is előfordulnak:

- az utasítás utáni operandusok legalább egyike regiszter,
- az első operandusba (cél) kerül a második (forrás) operandus,
- a forrás tartalma az utasítás végrehajtása után nem változik,

- amennyiben az adat a memóriában van, akkor a memória címe '[ ]' (szögletes) zárójelek között van,
- ha az adatot cimke határozza meg, akkor az elérendő adat típusát meg kell adni (pl. word ptr),
- a két operandus mérete azonos kell hogy legyen (a mov ax,3Bh esetében ax –be 003Bh kerül).

A PC személyi számítógépek képernyőkezelésénél ismerni kell a kiírás módját. Az operatív memória 0b800h címétől (szegmenscím) kezdődik a video memória, ahova ASCII alakban beírt karaktereket tudunk küldeni, amiket azután a hardver a képernyőn jelenít meg. Ez a pont a képernyő bal felső sarka. Tetszőleges pozícióba lehet írni a következő képlet alapján, ha tudjuk azt, hogy egy karakterhez két bájt tartozik, a kód és a szín:
cím = 80 * 2 * sor + 2 * oszlop.
Példa: írjunka 7. sor 13. oszlopába, 1146-ot kapunk, ami 47ah, vagyis a cím 0b800:47ah.

A színkódok a 4.4 táblázatban találhatók.

4.4 táblázat: a karakterek színkódjai			
bit	jelentés	bit	jelentés
0.	az előtér kék színösszetevője	4.	a háttér kék színösszetevője
1.	az előtér zöld színösszetevője	5.	a háttér zöld színösszetevője
2.	az előtér piros színösszetevője	6.	a háttér piros színösszetevője
3.	az előtér intenzitása	7.	villogás ki- bekapcsolása

4.4.3.2 - 2. feladat

A képernyő 7. sorának 13. oszlopába írjunk egy X betűt.

```
;*******************************************************************
Prog02      Segment                     ;szegmensdefinico
        assume CS:Prog02,DS:Prog02      ;CS es DS beallitas a szegmens elejere
;*******************************************************************
Start:  mov     ax,Prog02               ;DS regiszter beallitasa
        mov     ds,ax
        mov     ax,0b800h               ;es regiszterbe videomemoria
        mov     es,ax
        mov     di,1146                 ;offset-cim di-be
        mov     al,"X"                  ;al-be X betu
        mov     ah,7                    ;fekete alapon feher szin
        mov     es:[di],ax              ;betu a videomemoriaba
        mov     ax,4c00h                ;kilepes a DOS-ba
        int     21h
;*******************************************************************
Prog02      Ends                        ;a szegmens vege
        End     Start
;*******************************************************************
```

4.4.3.3 – 3. feladat:

A képernyő tetszőleges helyére írjunk ki egy karaktert, de a cimkek tartalmazzak a sort, oszlopot, karaktert és színt, amely adatok alapján kiszámolja a program a helyét.

```
;*****************************************************************
Prog03      Segment                     ;szegmensdefinico
            assume CS:Prog03,DS:Prog03  ;CS es DS beallitas a szegmens
                                        ;elejere
;*****************************************************************
Start:  mov   ax,Prog03                 ;DS regiszter beallitasa
        mov   ds,ax
        mov   ax,0b800h                 ;es regiszterbe videomemoria
        mov   es,ax
        mov   al,byte ptr [Y]           ;y betoltese
        mov   ah,0
        mov   bl,160                    ;1 sor 80 * 2 bajt
        mul   bl                        ;ax = al * bl
        mov   di,ax                     ;di-be tolteni a helyet
        mov   al,byte ptr [X]           ;vizszintes koordinata
        mov   ah,0
        mov   bl,2                      ;vizszintes koord. * 2
        mul   bl
        add   di,ax                     ;vegleges koordinata
        mov   al,byte ptr [KARAKT]      ;karakter kodja
        mov   ah,byte ptr [SZIN]        ;szinkod
        mov   es:[di],ax                ;betu a videomemoriaba
        mov   ax,4c00h                  ;kilepes a DOS-ba
        int   21h
;*****************************************************************
X:      db    40
Y:      db    12
KARAKT:db "X"
SZIN:   db    7
;*****************************************************************
Prog03      Ends                    ;a szegmens vege
        End   Start
;*****************************************************************
```

Azt, hogy hova kerül a betű a memóriában tároljuk, mégpedig úgy, hogy megadjuk a változó szimbólikus nevét, a db direktívával meghatározzuk hogy ez az adat a memóriába kerül, végül megadjuk az értéket (pl. X: db 40).

A mul a bl és al regisztereket szorozza össze, de mivel két nyolcbites szám szorzata egy 16 bites szám, ezért az eredmény ax-ben kapjuk. A 16 bites szorzás ax és bx között történik, az eredmény dx és ax regiszterekbe íródik.

4.4.3.4 – 4. feladat

Írjunk ki a képernyőre egy szöveget, a szöveget a memóriában tároljuk, úgy, hogy a végére egy nem nyomtatandó karaktert, pl. 0-át tegyünk.

```
;*****************************************************************
Prog04      Segment                     ;szegmensdefinico
            assume CS:Prog04,DS:Prog04  ;CS es DS beallitas a szegmens
                                        ;elejere
;*****************************************************************
Start:mov   ax,Prog04                   ;DS regiszter beallitasa
            mov   ds,ax
            mov   ax,0b800h             ;es regiszterbe videomemoria
            mov   es,ax
            mov   al,byte ptr [Y]       ;Y betoltese
            mov   ah,0
            mov   bl,160                ;1 sor 80 * 2 bajt
            mul   bl                    ;ax = al * bl
            mov   di,ax                 ;di-be tolteni a helyet
            mov   al,byte ptr [X]       ;vizszintes koordinata
            mov   ah,0
            mov   bl,2                  ;vizszintes koord. * 2
            mul   bl
            add   di,ax                 ;vegleges koordinata
            mov   ah,byte ptr [SZIN]    ;szinkod
            mov   si,offset SZOVEG      ;a szoveg cime si-be
cikl:       mov   al,[si]               ;adat al-be (betu)
            cmp   al,0                  ;szoveg vege?
            jz    vege                  ;vege
            mov   es:[di],ax            ;videomemoriaba betu
            add   di,2                  ;kovetkezo hely kepernyon
            inc   si                    ;kovetkezo betu
            jmp   cikl                  ;tovabb
vege:       mov   ax,4c00h              ;kilepes a DOS-ba
            int   21h
;*****************************************************************
X:          db    30
Y:          db    12
SZOVEG:     db    "SZABADKAI MUSZAKI FOISKOLA",0
SZIN: db    7
;*****************************************************************
Prog04      Ends                        ;a szegmens vege
            End   Start
;*****************************************************************
```

A címke egy hivatkozási hely, ugráskor, alprogram kezdetekor. Mindig "**:**" (kettőspont) követi. A programban összehasonlításra a **cmp** utasítás szolgál, ami állítja a jelzőbíteket, mi jelen estben a **Z** bitet ellenőrizzük összehasonlítás után. Az **inc** utasítás eggyel növeli az adott opernadus értékét.

4.4.3.5 – 5. feladat

A program két nyolcbites szám között végezze el a logikai ÉS műveletet, írja ki a két bináris számot, valamint az eredményt is. A képernyőtörlés történjen meg a számok kiírása elött, alkalmazzon alprogramot az azonos feladatok megoldására, mint például a szöveg kiírása, a bináris számok kiírása.

```
;*****************************************************************
Prog05      Segment                     ;szegmensdefinico
            assume CS:Prog05,DS:Prog05  ;CS es DS beallitas a szegmens
                                        éelejere
;*****************************************************************
Start:mov   ax,Prog05                   ;DS regiszter beallitasa
      mov   ds,ax
      mov   ax,0b800h                   ;es regiszterbe videomemoria
      mov   es,ax
      mov   ax,3                        ;80*25          uzemmod,
kepernyotorles
      int   10h
      mov   di,0*160                    ;elso szoveg kezdocim
      mov   si,offset SZOV1             ;SZOV1 szoveg kezdocime
      call  szovir                      ;alprogram szovegirasra
      mov   bl,byte ptr [SZ1]           ;szam beallitas
      call  szamir                      ;alprogram szamirasra
      mov   di,1*160                    ;masodik szoveg kezdocime
      mov   si,offset SZOV2             ;SZOV2 kezdocime
      call  szovir
      mov   bl,byte ptr [SZ2]           ;szam beallitas
      call  szamir
      mov   di,2*160
      mov   si,offset SZOV3
      call  szovir
      mov   bl,byte ptr [SZ1]           ;elso szam
      and   bl,byte ptr [SZ2]           ;masodik szammal AND
      call  szamir
      mov   ax,0                        ;varakozas billentyure
      int   16h
vege: mov   ax,4c00h                    ;kilepes a DOS-ba
      int   21h
;*****************************************************************
szovir Proc                             ;szovegkiiro alprogram
      mov   cx,16                       ;16 karakteres szoveg
      mov   ah,15                       ;fekete hatter, feher betu
cikl1:mov   al,[si]                     ;kiirando betu al-be
      mov   es:[di],ax                  ;videomemoriaba betu
      add   di,2                        ;kovetkezo pozicio
      inc   si                          ;kovetkezo karakter
      loop  cikl1                       ;cx=cx-1 es ugras ha cx<>0
      ret                               ;visszateres foprogramba
szovir Endp                             ;alprogram vege
```

```
;*****************************************************************
szamir Proc                                     ;szamkiiro alprogram
        add     di,6                            ;harom hellyel tovabb
        mov     cx,8                            ;8 bites adat
        mov     ah,15                           ;fekete hatter, feher betu
cikl2:mov       al,48                           ;0 ASCII kodja
        shl     bl,1                            ;CY <== d7 bit
        jnc     ki2                             ;ha 0, ugras ki2-re
        mov     al,49                           ;1 ASCII kodja
ki2:    mov     es:[di],ax                      ;szam kiirasa videomemoriaba
        add     di,2                            ;kovetkezo hely
        loop    cikl2                           ;cx=cx-1 es ugras ha cx <>0
        ret                                     ;alprogram vege
szamir Endp                                     ;alprogram vege
;*****************************************************************
SZOV1:          db      "Az elso byte   :"
SZOV2:          db      "A masodik byte :"
SZOV3:          db      "Az eredmeny    :"
SZ1:    db      01011101b
SZ2:    db      10101011b
;*****************************************************************
Prog05          Ends                            ;a szegmens vege
        End     Start
;*****************************************************************
```

A program bemutatja az alprogram használatát és annak írását is. Minden esetben, amikor valamely programrész-ismétlődésre kerül sor, alprogramot írunk, egyrészt rövidebbé, másrészt áttekinthetőbbé téve magát a programot.

A számláló használata is látszik, a **cx** regiszter tartalmát a **loop** utasítás csökkenti és egyben ellenőrzi a kapott eredményt, a feltétel teljesülésekor feltételes ugrást hajt végre a program, nem teljesülésekor pedig a következő utasítástól folytatja a program végrehajtását.

Az int 10h megszakítás a 0 értékkel a képernyő 80*25 üzemmódját állítja be, az int16 megszakítás pedig a billentyű lenyomásáig vár, a beütött karakter ASCII kódját pedig ah regiszterben tárolja (a példában ezt nem használtuk).

4.4.3.6 – 6. feladat

Olvassuk ki a számítógép óráját és írjuk ki a képernyőre.

```
Prog06          Segment                         ;szegmensdefinico
        assume CS:Prog06,DS:Prog06              ;CS es DS beallitas a szegmens elejere
;*****************************************************************
Start:mov       ax,Prog06                       ;DS regiszter beallitasa
        mov     ds,ax
        mov     ax,0b800h                       ;es regiszterbe videomemoria
        mov     es,ax
        mov     ax,3                            ;80*25 uzemmod, kepernyotorles
```

```
        int     10h
cikl:   mov     ax,0200h                ;idokiolvasas
        int     1ah
        call    ir                      ;az ido kiirasa
        mov     ax,0100h                ;billentyu?
        int     16h
        jz      cikl                    ;nincs, vissza
        mov     ax,0                    ;kiolvassa az erteket
        int     16h
        mov     ax,4c00h                ;vissza a DOS-ba
        int     21h
;*********************************************************************
ir      Proc                            ;kiiras alprogram
        mov     di,0                    ;kepernyo bal felso sarka
        mov     ah,15                   ;fekete hatter, feher betu
        mov     bx,cx                   ;bx <== ora, perc
        call    ir1                     ;ora kiirasa
        inc     di                      ;elvalasztojel helye
        inc     di
        call    ir1                     ;perc kiirasa
        inc     di                      ;elvalsztojel helye
        inc     di
        mov     bh,dh                   ;bh <== masodperc
        call    ir1                     ;masodperc kiirasa
        mov     al,"-"                  ;elvalasztojel
        mov     es:[di-6],ax            ;a megfelelo helyre
        mov     es:[di-12],ax
        ret                             ;visszateres foprogramba
ir      Endp                            ;alprogram vege
;*********************************************************************
ir1     Proc                            ;alprogram
        mov     cx,2                    ;2 szamjegy kiirasa
cikl1:push      cx
        mov     cx,4                    ;BCD szam (4 bit)
        mov     al,0
cikl2:shl       bx,1                    ;bx felso BCD szamjat
        rcl     al,1                    ;al- be teszi at
        loop    cikl2
        or      al,30h                  ;bin ==> ASCII
        mov     es:[di],ax              ;kepernyore
        inc     di                      ;kovetkezo hely
        inc     di
        pop     cx                      ;masik szamjegy
        loop    cikl1
        ret                             ;alprogram vege
ir1     Endp                            ;alprogram vege
;*********************************************************************
Prog06          Ends                    ;a szegmens vege
        End     Start
;*********************************************************************
```

A program egy belső beépített órából olvassa ki az időt, ciklikusan. Itt BCD-ben jelenik meg az érték, ezt kell átalakítani ASCII alakká, a 4 bites bináris számhoz 48-at kell hozzáadni. A megoldásban az or al,30h ugyanezt az eredményt adja.

4.4.3.7 – 7. feladat

Írjunk egy programot, mely a képernyő térlése után kiír a képernyőre 5 sorban választási lehetőséget a következő alakban:

- ELSO
- MASODIK
- HARMADIK
- NEGYEDIK
- OTODIK.

Jelenjen meg egy zöld háttér az ELSO felirat mögött, amit a lefelé és felfelé nyíllal lehet mozgatni. A kiválasztott sor után az ENTER lenyomása után a program írja ki a következő szöveget:

- KIVALASZTVA: xxxxx MENU,

majd tegye fel a következő kérdést:

- UJ VALASZTAS? (i/n),

ami után ismételhető a program, illetve ki lehet lépni belőle. Hibás betű megnyomása esetén ne történjen semmi. Ez a szöveg piros betűkkel jelenjen meg.

```
Prog07      Segment                         ;szegmensdefinico
            assume CS:Prog07,DS:Prog07      ;CS es DS beallitas a szegmens
                                            ;elejere
;*****************************************************************
Start:mov   ax,Prog07                       ;DS regiszter beallitasa
      mov   ds,ax
      mov   ax,0b800h                       ;es regiszterbe videomemoria
      mov   es,ax
      mov   ax,3                        ;80*25uzemmod,kepernyotorles
      int   10h
      call  write1                          ;a menupontok kiirasa
cimke1:call line                            ;a mutato megjelenitese
cimke2:mov  ax,0100h                        ;billentyuzet?
      int   16h
      jz    cimke2
      mov   ax,0
      int   16h
      cmp   ah,1ch                          ;ENTER?
      jz    ent                             ;igen
      cmp   ah,48h                          ;^?
      jnz   cimke3                          ;nem, tovabb vizsgalni
```

```
        cmp     byte ptr [POINTER],1            ;POINTER nem 1,
        jz      cimke2
        dec     byte ptr [POINTER]              ;POINTER - 1
        jmp     cimke1                          ;ujbol billentyufigyeles
cimke3:cmp      ah,50h                          ;nyil lefele?
        jnz     cimke2                          ;nem, billentyufigyeles
        cmp     byte ptr [POINTER],5            ;POINTER nem mutat az
        jz      cimke2                          ;5. helyre, novelni
        inc     byte ptr [POINTER]
        jmp     cimke1                          ;ujbol billentyufigyeles
ent:    mov     cl,byte ptr [POINTER]           ;cl-be POINTER erteke
        mov     si,offset ADDRESS               ;cimbetoltes bs-be, ahova
ent1:   mov     bx,[si]                         ;POINTER mutat
        add     si,2
        loop    ent1
        jmp     bx                              ;ugras bx tartalma alapjan
ent2:   mov     si,offset QUEST                 ;kerdes kiirasa
        call    write2
ent3:   mov     ax,0100h                        ;igen? nem?
        int     16h
        jz      ent3
        mov     ax,0
        int     16h
        cmp     ah,31h                          ;nem, kilepes
        jz      ent4
        cmp     ah,17h                          ;mas, ujbol figyelni
        jnz     ent3
        jmp     start                           ;igen, program elejere
ent4:   mov     ax,4c00h                        ;vissza a DOS-ba
        int     21h
;*********************************************************************
pr1:    mov     si,offset TXT1                  ;szovegek kiirasa
        call    write2
        jmp     ent2
pr2:    mov     si,offset TXT2                  ;szovegek kiirasa
        call    write2
        jmp     ent2
pr3:    mov     si,offset TXT3                  ;szovegek kiirasa
        call    write2
        jmp     ent2
pr4:    mov     si,offset TXT4                  ;szovegek kiirasa
        call    write2
        jmp     ent2
pr5:    mov     si,offset TXT5                  ;szovegek kiirasa
        call    write2
        jmp     ent2
;*********************************************************************
line    Proc                                    ;csik alprogram
        mov     al,byte ptr [POINT2]            ;a mutato elozo erteket
        mov     bl,160                          ;al, be a hosszat bl-be
```

```
        mov     ah,0
        mul     bl                              ;mutato kezdocimenek meghat.
        add     ax,1499
        mov     di,ax
        mov     cx,21
        mov     al,7                            ;fekete hatter, feher betu
line1:mov       es:[di],al                  ;szininformacio video memoriaba
        add     di,2
        loop    line1
        mov     al,byte ptr [POINTER]           ;mint elozoleg, de uj pozicio
        mov     byte ptr [POINT2],al            ;es kiemelt hatter szin
        mov     bl,160
        mov     ah,0
        mul     bl
        add     ax,1499
        mov     di,ax
        mov     cx,21
        mov     al,47
line2:mov       es:[di],al
        add     di,2
        loop    line2
        ret                                     ;visszateres foprogramba
line    Endp                                    ;alprogram vege
;*********************************************************************
write1Proc                                      ;szoveg kiiro alprogram
        mov     si,offset CHOICE                ;0 ==> di+160 (soremeles)
        mov     ah,7                            ;255 ==> szoveg vege es kilepes
        mov     di,1660
wr1:    push    di
wr2:    mov     al,[si]
        inc     si
        cmp     al,0
        jz      wr3
        cmp     al,0ffh
        jz      wr4
        mov     es:[di],ax
        add     di,2
        jmp     wr2
wr3:    pop     di
        add     di,160
        jmp     wr1
wr4:    pop     di
        ret                                     ;alprogram vege
write1Endp                                      ;alprogram vege
;*********************************************************************
write2Proc                                      ;szovegkiiras
        mov     al,[si]                         ;elso 2 byte koordinata
        mov     ah,0                            ;amibol         memoriacim
szamolhato
        shl     ax,1
```

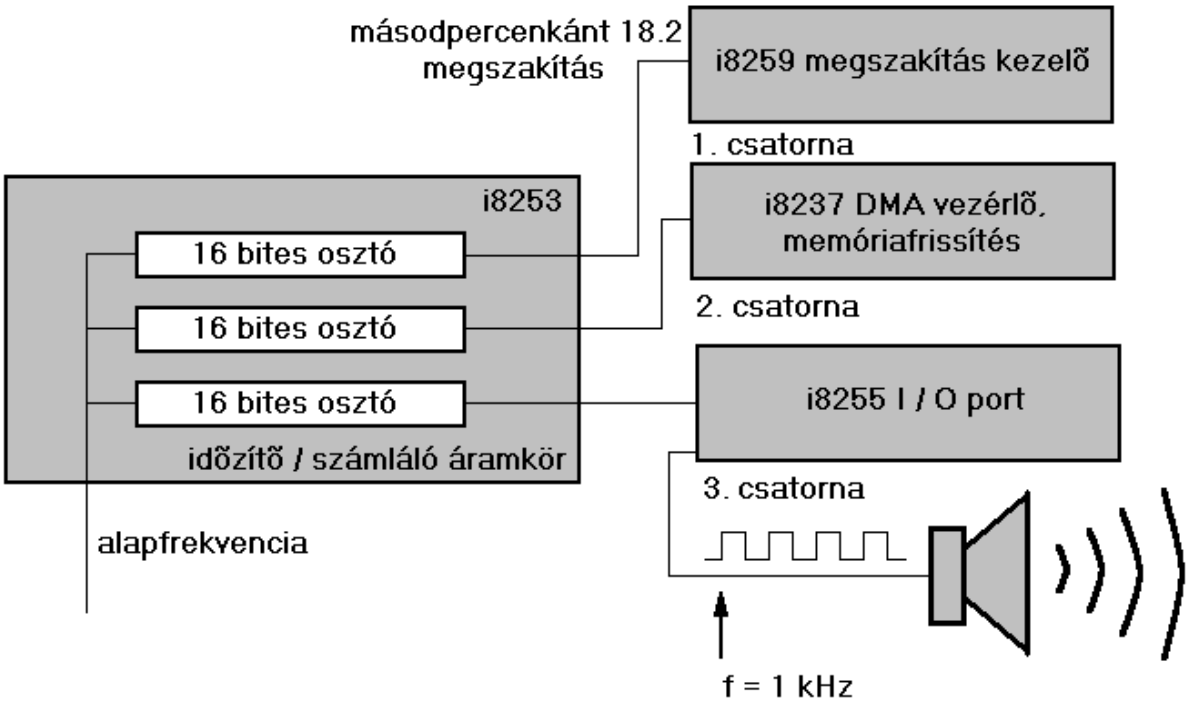
```
        mov     di,ax
        mov     al,[si+1]
        mov     bl,160
        mul     bl
        add     di,ax
        add     si,2
        mov     ah,12
wr21:   mov     al,[si]
        inc     si
        cmp     al,0
        jz      wr22
        mov     es:[di],ax
        add     di,2
        jmp     wr21
wr22:   ret
write2Endp
;*****************************************************************
POINTER:db          1
POINT2:db   1
CHOICE:db   "ELSO",0
        db      "MASODIK",0
        db      "HARMADIK",0
        db      "NEGYEDIK",0
        db      "OTODIK",255
TXT1:db     22,5,"KIVALASZTVA: EGYES MENU",0
TXT2:db     21,5,"KIVALASZTVA: KETTES MENU",0
TXT3:db     21,5,"KIVALASZTVA: HARMAS MENU",0
TXT4:db     21,5,"KIVALASZTVA: NEGYES MENU",0
TXT5:db     21,5,"KIVALASZTVA: OTOS MENU",0
QUEST:db    28,20,"UJ VALASZTAS? (i/n)",0
ADDRESS:dw          offset pr1
        dw      offset pr2
        dw      offset pr3
        dw      offset pr4
        dw      offset pr5
;*****************************************************************
Prog07Ends                                      ;a szegmens vege
        End     Start
;*****************************************************************
```

4.4.3.8 – 8. feladat

Írjunk olyan programot, amelyik 1 kHz-es frekveciával egy csipogó hangot ad.

Mielött a programíráshoz kezdenénk, meg kell vizsgálni, mi áll rendelkezésünkre a hang előállítására. A képernyőre való kiírás azért viszonylag egyszerű, mert a videomemória része a gép operatív memóriájának, így egyszerűen csak azt kellett tudnunk, hol kezdődik a videomemória, milyen a szervezése. Gyakorlatilag egy hardver oldja meg ezután a képernyőre való információ (esetünkben a szöveg) kiírását. Ez így is kell hogy legyen, hiszen a gyakorlatban a legtöbb művelet a képernyő felé történik, itt kapjuk a legtöbb információt, akár szöveg, akár kép alakjában.

A 4.13 ábrán látható a hardver egy részlete, ahonnan megérthetjük a hangszóró használatát, illetve ahol megtalálhatjuk a hangszóró megszólaltatását biztosító paramétereket, amik segítségével programozhatjuk a hardvert.

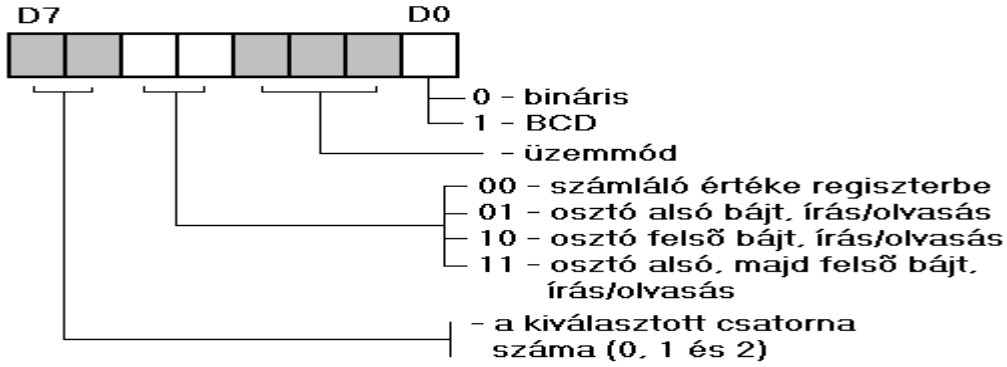


4.13 ábra: A PC egy hardver-részlete az i8253 időzítő / számláló áramkörrel

Az 1. csatorna egy i8259 megszakításvezérlőre van kötve, ami másodpercenként 18.2-szer küld megszakítást a számítógép felé, a 2. csatorna az i8237 DMA (közvetlen memóriahozzáférést vezérlő) áramkörrel van kapcsolatban, számunkra a 3. csatorna az érdekes, amely a hangszóróra kapcsolódik (a megfelelő elektronika segítségével).

A hangszóró megszólaltatása a 3. csatorna segítségével úgy történik, hogy az alapfrekvenciát (1.19318 MHz-et) leosztjuk 1 kHz-re. Ehhez szükséges az i8253 áramkör programozását ismerni.

Az áramkör a 043h címen fogadja a parancsokat, amiket írásparanccsal közölhetünk, az egyes bitek jelentése a következő (4.14 ábra):



4.14 ábra: az i8253 időzítő / számláló vezérlőbitjeinek jelentése

A periodikus négyszögjelek beállítása után a hnagszóró bekapcsolása az i8255 porton keresztül történik. Ez a port 3 8 bites kapuáramkört tartalmaz, a hangszórót a B port D1-D0 bitje befolyásolja. A program a következe

```
Prog08      Segment                         ;szegmensdefinico
            assume CS:Prog08,DS:nothing     ;CS beallitas a szegmens elejere
                                   ;es nincs adat
;*******************************************************************
prog_55     equ    061h                     ;8255 port cime
prog_53     equ    043h                     ;8253 cime
ido         equ    042h
;*******************************************************************
******
Start:  mov  al,10110110b                   ;timer2 lesz az oszto
            out    prog_53,al               ;kivinni
            mov    ax,1193                  ;osztasarany 1 kHz
            out    ido,al
            mov    al,ah
            out    ido,al
;*******************************************************************
            in     al,prog_55          ;hangszoro bekapcsolasa, de mivel
                                       ;ez a port mas periferiakat is
                                       ;vezerel, ezek bitjeit nem szabad
                                       ;valtoztatni
            or     al,00000011b        ;bekapcsolas D1,D0 bitekkel
            out    prog_55,al
            mov    cx,0ffffh           ;varakozas 0ffffh ideig
cikl:       loop   cikl                ;a hang ideje
            in     al,prog_55          ;kikapcsolni a hangszorot
            and    al,11111100b
            out    prog_55,al
            mov    ax,4c00h            ;visszateres DOS-ba
            int    21h

;*******************************************************************
Prog08      Ends                       ;a szegmens vege
            End    Start
;*******************************************************************
```

4.4.4 Az IBM PC grafikus kártyájának vezérlése assembly-ből

A szöveges üzemmód használata nem enged meg rajzolást, csak néhány karakter egymás mellé helyezésével lehet igen egyszerű ábrákat létrehozni. Amennyiben bonyolultabb ábrákat szeretnénk létrehozni, esetleg azokat még mozgatni is, akkor grafikus üzemmódban kell használni a gépet.

Az IBM PC gépekre jellemző az, hogy a felhasználó igényeihez mérten alakítja ki a hardver konfigurációt, illetve választja meg a szoftver eszköztárat. Ez a nagy szabadság a hardver megválasztásánál azzal jár, hogy többféle videokártyával is találkozhatunk a gépeknél. A grafikus kártyák igen eltérő képességűek, sok tipus ma már nem is használatos közülük.

4.4.4.1 Különféle grafikus üzemmódok programozása

Több módon is megoldható a grafikus kártya programozása:
egyik módszer az, amikor a már meglevő BIOS alprogramokat használjuk, ez kényelmes, hiszen már helyettünk megírták a programokat, csak a megfelelő paramétereket kell beállítani az alprogram meghívásakor,
a másik módszer, amikor az úgynevezett direkt programozást alkalmazzuk, ez ugyan nehezebb mint az előző, de sokkal gyorsabban működnek a programok.

A szöveges kiírásnál egy mátrix megfelelő helyeire írtunk ASCII karaktereket, azok alakját, képét nem mi határoztuk meg. A felbontás 80*25 volt, a grafikus üzemmódnál csak pontokat tudunk egy sokkal nagyobb felbontású mátrix egyes pontjaiba helyezni. Ha alakzatot akarunk létrehozni, akkor azt nekünk kell összerakni, például az A betűt sok kis pontból hozhatjuk létre.

4.4.4.2 Az IBM PC számítógép CGA grafikus üzemmódja

Ez a kártya egy egyszerű, kis felbontású hardver, amely az első IBM PC számítógépeknél jelent meg. Felbontása 320*200 képpont, legfeljebb négy színnel. Ezt a négy színt 16 színből lehet kiválasztani. Ha nincs szükségünk színre, pontosabban egy háttérszín és egy előtérszín elegendő, akkor beállíthatjuk a 640*200-as felbontást.

1 bájton 4 képpont tárolható, hiszen a 4 színhez 2 bit kell. A 320*200 képpont 8000 bájton így elfér. Egy kis furcsaság is található ennél a kártyánál, a memória első részében a páros, míg a második részében a páratlan sorokat tárolja, így 0b800:0000h a páros, a 0b800:2000 pedig a páratlan sorok kezdőcíme.

A 4.5 táblázat szerint lehet a színeket kiválasztani.

4.5 Táblázat: a CGA kártya színei				
1. paletta	0	1	2	3
	háttérszín	zöld	piros	sárga/barna
2. paletta	0	1	2	3
	háttérszín	türkiz	lila	fehér/szürke

A 16 színt az intenzitás beállításával lehet létrehozni.

Feladat: CGA üzemmódban a bal felső sarokba rajzoljon ki a program egy kék, lila és fehér vonalat. Az első billnetyűlenyomás után történjen meg a palettaváltás, második billentyűnyomás után sötétre váltsanak a színek, egyidejűleg a háttér kék színű legyen. Az utolsó billentyű állítsa vissza az erősebb színintenzitást. A program feltétlenül DOS üzemmódba váltson vissza a kilépés elött.

A program listája a következő:

```
Prog10      Segment                         ;szegmensdefinico
        assume CS:Prog10,DS:Prog10      ;CS es DS beallitas a szegmens elejere
;*****************************************************************
Start:mov   ax,Prog10                   ;DS beallitas a szegmens elejere
        mov     ds,ax
;*****************************************************************
        mov     ax,0b800h               ;CGA videomemoria kezdocime
        mov     es,ax
;*****************************************************************
        mov     ax,4                    ;ah=0, kepernyo uzemmod
        int     10h                     ;al=4, CGA, 320*200/4
                                        ;grafikus uzemmod
;*****************************************************************
        mov     ah,0bh                  ;1-es paletta
        mov     bh,1                    ;palettaallitas
        mov     bl,1                    ;1-es paletta sorszam
        int     10h                     ;0 vagy 1 lehet
;*****************************************************************
        mov     al,01010101b            ;8 1-es szinkod bal felso sarokba
        mov     es:[0],al
        mov     al,10101010b            ;8 2-es szinkod melle
        mov     es:[1],al
        mov     al,11111111b            ;8 3-as szinkod melle
        mov     es:[2],al
;*****************************************************************
        xor     ax,ax                   ;billentyu lenyomasra varni
        int     16h
;*****************************************************************
        mov     ah,0bh                  ;0. paletta beallitasa
        mov     bh,1
        mov     bl,0
        int     10h
;*****************************************************************
        xor     ax,ax                   ;billentyu lenyomasra varni
        int     16h
;*****************************************************************
        mov     ah,0bh                  ;bh = 0, akkor
        mov     bh,0
        mov     bl,1                    ;akkor bl hatterszin, most kek
        int     10h
```

```
;*****************************************************************
        xor     ax,ax                   ;billentyu lenyomasra varni
        int     16h
;*****************************************************************
        mov     ah,0bh                  ;hatter 4. bit eloter intenzitasa
        mov     bh,0                    ;barna => sarga
        mov     bl,1+16                 ;szurke => feher
        int     10h
;*****************************************************************
        xor     ax,ax                   ;billentyu lenyomasra varni
        int     16h
;*****************************************************************
        mov     ax,3                    ;80*25 karakteres mod visszaallitas
        int     10h
;*****************************************************************
        mov     ax,4c00h                ;visszateres DOS-ba
        int     21h
;*****************************************************************
Prog10  Ends                            ;a szegmens vege
        End     Start
;*****************************************************************
```

A program értékelése:

A CGA 320*200-as üzemmód beállítása után az egyes palettát választjuk ki. Ezután a képernyő bal felső sarkába egymás mellé 3 vonal kerül kirajzolásra, egy kék, egy lila és egy fehér. A programba egy BIOS rutinnal billentyűvárakozást építettünk be. A billentyű lenyomásakor a színek változnak, mégpedig zöldre, pirosra és sárgára. A következő billentyűlenyomáskor elsötétülnek a színek, ugyanakkor a háttér kék színű lesz. Az utolsó elötti billentyűlenyomás nagyobb intenzitással jeleníti meg a vonalakat. Legutolsó billentyű hatására a program visszaadja a vezérlést a DOS-nak. Nem szabad elfelejteni a grafikus üzemmódot visszaváltani szöveges üzemmódra.

4.4.4.3 Az IBM PC számítógép VGA grafikus üzemmódja

Feladat: A program rajzoljon ki VGA grafikus üzemmódban egy zászlót.

A program listája a övetkező:

```
Prog11  Segment                         ;szegmensdefinico
        assume CS:Prog11,DS:Prog11      ;CS es DS beallitas a szegmens elejere
;*****************************************************************
******
Start:  mov     ax,Prog11               ;DS beallitas a szegmens elejere
        mov     ds,ax
;*****************************************************************
        mov     ax,0a000h               ;VGA videomemoria kezdocime
        mov     es,ax
;*****************************************************************
        mov     ax,13h                  ;VGA 320*200 /256 szin
```

```
        int     10h
;*****************************************************************
        call    color                           ;szinbeallitas
;*****************************************************************
        mov     di,320*69                       ;hova, 70. sor eleje
        mov     al,1                            ;piros eloterszin
        cld                                     ;di novekedjen
        mov     cx,3                            ;3 szin kirajzolasa
;*****************************************************************
cikl1:  push    cx                              ;szamlalot elmenteni
        mov     cx,320*30                       ;30 keppont
        rep stosb                               ;kiiras ismetles
        pop     cx                              ;szamlalot visszamenteni
        inc     al                              ;kovetkezo szin
        loop    cikl1
;*****************************************************************
        mov     ax,0                            ;billentyu?
        int     16h
;*****************************************************************
        mov     ax,3                            ;szoveges kepernyo
        int     10h
;*****************************************************************
        mov     ax,4c00h                        ;visszateres DOS-ba
        int     21h
;*****************************************************************
;#################################################################
color   Proc                                    ;szinbeallitas alprogram
                                        ;a hatter es 3 zaszloszin beallitasa
;*****************************************************************
        push    ax                      ;a hasznalt regiszterek mentese
        push    cx
        push    dx
;*****************************************************************
        mov     dx,3c9h
        mov     cx,4                            ;hatterszin + 3 eloterszin
        xor     al,al                           ;hatterszin
        mov     si,offset colors
;*****************************************************************
vga:    push    ax
        dec     dx
        out     dx,al
        inc     dx
        mov     al,[si]
        out     dx,al
        mov     al,[si+1]
        out     dx,al
        mov     al,[si+2]
        out     dx,al
        add     si,3
        pop     ax
```

```
        inc     al                              ;kovetkezo szin
        loop    vga
;*****************************************************************
        pop     dx                              ;regiszterertekek visszaallitasa
        pop     cx                              ;vigyazat, forditott sorrendben
        pop     ax
        ret
;*****************************************************************
color   Endp                                    ;alprogram vege
;#################################################################
;*****************************************************************
colors: db      0,0,63                          ;kek hatter
        db      63,0,0                          ;piros eloter
        db      63,63,63                        ;feher eloter
        db      0,63,0                          ;zold eloter
;*****************************************************************
Prog11          Ends                            ;a szegmens vege
        End     Start
;*****************************************************************
```

Az előző példában CGA (vagyis a legegyszerűbb üzemmódban) állítottuk be a képernyőt. Ma már gyakorlatilag nem használjuk ezt az eléggé korlátozott képességekkel rendelkező kártyát. A sokkal jobb felbontású és sokkal több színt használó VGA üzemmód (kártya) használata indokolt.

A program elemzése:

Mivel a programban a háttérszín meghatározása után három különböző téglalapból állítjuk össze a piros-fehér-zöld zászlót, ezért itt is alprogramot használunk az azonos, vagy hasonló feladatok megoldására. A programból való kilépést bármely billentyűvel megvalósíthatjuk. Fontos a kilépés elött a szöveges üzemmód visszaállítása is.

A VGA videomemóriának a címe 0a000h, így ezt kell beállítani.Úgyszintén szükséges a 320*200 / 256 szín üzemmód beállítása, amit az int 10h megszakítás 13h paraméterével hozhatunk létre.

A megoldás használ egy összetett utasítást, a rep stosb tulajdonképpen a következő műveleteket valósítja meg:

```
ciklus:  mov    es:[di],al
         inc    di
         loop ciklus.
```

mivel di igényli a növekedés, illetve csökkenés meghatározását, ezt a cld utasítással érjük el (példánkban növekszik).

A színek beállítására egy color nevű alprogramot használunk, ahol a háttérszín és a három előtérszín meghatározása ugyanazon eljárással történik.

Fontos része az alprogramnak az alprogramban használt regisztertartalmak belépéskor való elmentése. Ez azért fontos, mert az alprogram hívásakor meghatározott regiszterértékekre a programnak az alprogramból való viszatéréskor szüksége lesz. A alprogramból való kilépéskor a regisztertartalmakat a pop utasítással vissza kell tölteni az erdeti helyekre. Az alprogram végén ez fordított sorrendben történik (a mentéshez képest), hiszen a használt stack memória LIFO típusú, vagyis a legutoljára beírt adat helyezkedik el a stack-memória legtetején. Az elfelejtett, vagy a rossz sorrendű regisztertartalom-visszamentés összezavarja a programvégrehjtást, ami a számítógép leállásához vezet. Itt kell megjegyezni, hogy néha szándékosan, adatcserére is felhasználhatjuk a nem sorrendben végrehajtott adtvisszamentést.

5 PROLOG

5.1 BEVEZETÉS

A Prolog egy deklaratív számítógépes programozási nyelv, amely lényegesen eltér a procedurális (eljárásalapú) számítógépes nyelvektõl. Az eltérések a következők:

- a Prologban *tények* halmaza van, amelyek a valós világ objektumaira vonatkoznak, valamint szabályok, amelyek ezen objektumokra vonatkoznak,
- a Prolog program végrehajtását a részproblémák megoldásához szükséges információk befolyásolják, és *nem* a program utasításainak sorrendje és
- a Prolog program egy *kérdéssel* indul, amely az információszerzés elsõdleges kiinduló pontja

Nézzünk a fentiekre egy példát:

A Prolog objektumokat és azok kapcsolatait használja a valós világ leírására. Vegyük a következõ deklarációt:

Az oktató órát tart.

Ez a mondat a tart viszonyt állítja fel két objektum (az oktató és óra) között. A viszony (reláció) elvontabb fogalom, mint az objektum. Ez azt jelenti, hogy ugyanazt a viszonyt más objektumok között is felállíthatjuk:

A tanársegég órát tart.
Az oktató tanfolyamot tart.
A meghívott oktató órát tart.

A tények ezen halmazával megadott kis *világban* kérdéseket tehetünk fel errõl a világról. (Ezt a *zárt világ feltételezésének* nevezik, mert csak a leírt tények és szabályok képviselik az igazságot. Ha nincs rögzítve, akkor akármilyen más állítás hamis értékû.)

Például ha azt mondjuk: "Igaz-e, hogy a meghívott oktató órát tart?"
A válasz erre **Igen**.

Arra a kérdésre, hogy "Igaz-e, hogy a meghívott oktató tanfolyamot tart?"
a válasz természetesen **Nem**.

Ráadásképpen megkérdezhetjük, hogy "Ki tart órát?"
A válasz:

az oktató,
a tanársegéd,
a meghívott oktató

azért, mert ezek a személyek elégítik ki a *tart* relációt az *óra* objektummal.

A tények mellett az objektumoknál absztraktabb (elvontabb) és általánosabb fogalom még a szabály. Ugyanannál a példánál maradva tegyük fel, hogy minden órát tartó személy tanár.

Felállíthatjuk ezek szerint a következõ szabályt:

Ha VALAKI órát tart akkor VALAKI tanár.

Itt VALAKI objektumok halmazának egy elemét tároló hely (változó). Megjegyzendõ, hogy a szabály **Ha** része általános szabályt fogalmaz meg a világra vonatkozóan, mert kapcsolatot ad a VALAKI-ben tárolt ismeretlen (változó) objektum és az 'óra' között a 'tart' relációval. Ha behelyettesítünk egy értéket a VALAKI tárolóba, meg tudjuk mondani, hogy ez kapcsolatban van-e a tanár objektummal vagy sem.

Az értelmes megfogalmazás a programozó dolga, tehát hogy a változóba egy személy neve kerüljön.
Ezzel a szabállyal és a fenti tényekkel *következtethetünk* néhány új kapcsolatra, például:

Az oktató tanár.
A tanársegéd tanár.
A meghívott oktató tanár.

Ezek abból következnek, hogy az oktató, a tanársegéd és a meghívott oktató teljesítik azt a szabályt, hogy:
VALAKI órát tart.

Hasonlóképpen kérdéseket fogalmazhatunk meg kizárólag a tények halmazának és a szabálynak a felhasználásával. Például ha megkérdezzük: "Az oktató tanár" ?
A válasz az lesz, hogy IGEN.
Összetettebb kérdés a "Ki tanár ?"
A válasz erre az oktató, a tanársegéd, a meghívott oktató lenne.

Így mûködik a PROLOG, deklarálni kell a tények halmazát a fentiekhez hasonlóan, majd néhány általános szabályt az objektumok halmazára vonatkozóan, aztán a PROLOG kombinálja a tények és szabályok halmazait a kérdésekre adandó válasz keresése során.

5.2 A PROLOG NYELVTANI SZABÁLYAI

A PROLOG tények és szabályok halmazából áll. A tények objektumok közötti, míg a szabályok a tények általánosításai közötti viszonyokat írnak le.
Tények
Vegyük a már bemutatott tényeket:

- az oktató órát tart.
- a tanársegéd órát tart.
- a tanszékvezető tanfolyamot tart.
- a meghívott oktató órát tart.

A *tart* relációt a Prologban mint **predikációt** vagy kijelentést a következõképpen írjuk le:

- tart(oktato, ora)

amely úgy olvasandó, hogy az elsõ objektum (**oktato**) 'tart' relációban van a második objektummal, azaz az **ora**-val. Ebben az esetben a 'tart' reláció két objektum között teremt kapcsolatot, de ezek száma bármely nemnegatív szám lehet. Ezt a számot a kijelentés *argumentum-számának* nevezzük. A Prolog nyelvtani szabályai néhány megszorítást tartalmaznak a tényekre vonatkozólag:

- A kijelentések és az objektumok kisbetûvel kezdõdnek. Így a "oktató", "tanársegéd" és "tanszékvezető" objektumok írásmódja "oktato", "tanarseged" és "tanszekvezeto" lesz.
- A kijelentések neve írandó elõre, és ha vannak objektumok (argumentumként), azokat zárójelben, vesszõvel elválasztva kell írni.
- Minden tényállítást ponttal kell lezárni.

E szabályok figyelembe vételével a fenti példa a következõképpen írandó:

- tart(oktato, ora).
- tart(tanarseged, ora).
- tart(oktato, tanfolyam).
- tart(meghivott_oktato, ora).

Meg kell jegyeznünk azt, hogy ezeket a szabályokat csak egyféleképpen olvashatjuk: az elsõ objektum áll kapcsolatban a másodikkal. Így a

- tart(ora, oktato).

nem ugyanaz, mint:

- tart(oktato, ora).

Néhány további tény ismerõs lehet:

- apja(peter, alice).
- tulajdonos(gerry, arany).
- szereti(janos, hal).
- lany(maria).
- fiu(janos).
- jatekos(michael, rugby).
- allat(koala).
- osszead(2,3,5).

5.2.1 Lekérdezés tények alapján

Ha már vannak tényeink, kérdéseket tehetünk fel velük kapcsolatban. Egy Prolog program egy kérdés feltevésével indul. Szintaktikailag a kérdés a tényekhez hasonlít, de ez a tényállítás változókat is tartalmazhat. Ezek a változók, például objektumokat tároló memóriahely-jelölõk mindig nagybetûvel kezdõdnek, például: X, Nev, TANAR, ...
A kérdésre adott válasz vagy Igen/Nem [Yes/No] vagy egy objektumhalmaz azon objektumokkal, amelyek szerepelhetnek az adott relációban. Nézzünk néhány Igen/Nem lekérdezést.
Legyen adott a tények következõ halmaza:

- tart(oktato, ora).
- tart(tanarseged, ora).
- tart(oktato, tanfolyam).
- tart(meghivott_oktato, ora).

Egy kérdés lehet a következõ:

- tart(tanarseged, ora).

amelyre a válasz **Yes**, míg a:

- tart(tanarsegd, tanfolyam).

kérdésre adott válasz **No** lesz.

Az általánosabb kérdések objektumok halmazára vonatkoznak, mint például: "Ki tart órát?" Prologban ezt így kell feltenni:

- tart(X, ora).

ahol X bármely (személy tartalmú) objektumot képviselhet, amely kielégíti az **ora** objektummal való relációt. A válasz a következõ lehet:

- X = oktato
- X = tanarseged
- X = meghivott_oktato

Másrészrõl megkérdezhetjük, hogy "Mit tart János?", aminek Prologbeli megfelelõje:

- tart(oktato, X).

A válasz a következõ lesz:

- X = ora
- X = tanfolyam

5.2.1.1 - 1. példa

Írjuk be az előző példát a PROLOG szövegszerkesztő segítségévelk és fordítsuk le:

```
tart(oktato, ora).
tart(tanarseged, ora).
tart(oktato, tanfolyam).
tart(meghivott_oktato, ora).
```

Ha beírjuk a következőt, a PROLOG értelmező (interpréter) elemzi a kérdésünket:

tart(tanarseged, ora).

és a következő választ adja:

Yes.

A PROLOG a következő sorra:
tart(X, ora).

a következő válaszokat adja:

X = oktató

X = tanarseged

X = meghivott_oktato.

Ahogyan látható, roppant egyszerû a tények halmazára vonatkozó kérdéseket feltenni. Bátran próbálkozzon az alábbiakhoz hasonló kérdések feltevésvel:

- tart(oktato, Mit).
- tart(Ki, tanfolyam).
- tart(Ki, Mit).

és így tovább.

Egy kérdés emellett tartalmazhat logikai ÉS kapcsolatokat is. Például megkérdezhetjük, hogy "Ki tart órát is és tanfolyamot is?". Ezt két relációt összekapcsoló AND mûvelettel tehetjük meg, amelyet a Prologban a vesszõ "," képvisel. A fenti kérdés tehát Prologban így teendõ fel:
---
tart(X, ora) , tart(X, tanfolyam).

X = oktato

5.2.1.2 - 2. példa

Írjuk meg a következő programot:

```
szuloje(a,b).
szuloje(a,c).
szuloje(b,d).
szuloje(e,f).
ferfi(b).
fiutestver(Y,X) :- szuloje(P1,X),
                   szuloje(P1,Y),
                   ferfi(Y),
                   not (X=Y).
```

Futtassuk a következõ kérdéseket:

- fiutestver(a,b).
- fiutestver(b,c).
- fiutestver(c,b).

és figyelje meg az eseményeket. Megfelel a várakozásainak?
Növelje a tények számát és futtasson további kérdéseket.
A fiutestver szabályból kiindulva írjon szabályokat a:

- nover
- nagyapa
- nagymama

számára! Futtasson kérdéseket az újonnan beírt szabályokra is!

5.2.2 A rekurzió

A Prolog egyik legérdekesebb vonása a rekurzív információk definiálásának lehetõsége. Rekurzióról akkor beszélünk, ha valamit önmagával és egy kilépõ feltétellel definiálunk. A klasszikus példa a faktoriális definiálása, amely a következõ:

- N faktoriálisa bármely pozitív számra egyenlõ N-1 faktoriálisának és N-nek a szorzatával.
- Nulla faktoriálisa legyen egyenlõ 1-el.

Ez a nagyon egyszerû, mégis hatékony és teljes faktoriális-definíció könnyen kifejezhetõ Prologban:

```
faktorialis(0,1).
faktorialis(N,X):- M is N - 1,
                   faktorialis(M,Y),
                   X is N * Y.
```

Ebben az esetben a tényállításnak meg kell elõznie a szabályt, mert A Prolog a faktorialis(M,Y) megoldását a szabályok leírt sorrendjében próbálja megoldani. Maga a program is itt végzõdik, mert a számláló itt csökken zérusra. Ellenkezõ esetben a program végtelen ciklusba esne.

A rekurzió nagyon jól használható, egyszerû és elegáns programot eredményez, de könnyen kicsúszhat az ellenõrzésünk alól. Ezért nagyon óvatosan járjunk el rekurzív programok futtatásakor.

Nagyon gyakran használunk rekurziót adatszerkezetek megadásakor is. Prologban az adat lehet atomi (egyszerû érték) vagy lista (értéksorozat). A lista elemek halmaza, amelyek lehetnek számok, betûk, szövegek és listák (listák listája).

A listák elemei szögletes zárójelek közé írandók és vesszõvel választandók el. Néhány példa egyszerû listákra:

- [szomba, konyha, fürdõszoba, nappali]
- [123, 12, 45]
- [a, b, c, d, e, f, g,h]

Mivel a lista elemek halmaza, néhány alapvetõ mûvelet kapcsolódhat a listákhoz, mint például a metszet-, únió- és különbség képzése, stb., amelyeket könnyen megfogalmazhatunk Prologban.

Például a bennevan1(X,L) szabály igaz, ha X eleme az L listának. Ezt a szabályt Prologban így fogalmazzuk meg:

```
bennevan1(X,[X|_]).
bennevan1(X, [_|L]) :- bennevan1(X,L).
```

A "|" jel osztja kellé a listát az elsõ elemre ("fej elem") és a lista maradékára. A fenti szabályok közül az elsõ bennevan1(X,[X|_]) akkor teljesül, ha X a lista elsõ eleme. (Az aláhúzásjel azt jelenti, hogy a lista azon részének értéke a megoldásban most érdektelen.
Ha az elsõ szabály nem teljesülne, akkor a szabály második sora leveszi a lista elsõ elemét, és rekurzív hívással ismét végrehajtódik a bennevan1 szabály szerint, de már a lista elsõ eleme nélkül.

5.2.2.1 - 3. példa

Írja be a bennavan1 szabályokat egy állományba és ellenõrizze a mûködését a következõ kérdésekkel:

- bennevan1(a,[b,c,a,d]).
- bennevan1(a,[b,c,f,d]).
- bennevan1(20, [15,16, 17, 18, 19, 20]).
- bennevan1(mike, [ana, paul, mary, tom]).

Hogy követhesse azt, amit a Prolog tesz a végrehajtás során, figyelje meg azt nyomkövetéssel. A nyomkövetés bekapcsolásához írja be a kérdés kiadása elõtt:

```
trace.
```

Ezek után <enter> -t kell nyomnia minden következõ lépés végrehajtásához. Próbálja ki ezt a fenti kérdések némelyikével és tanulmányozza a kiírtakat!

Egy másik hasznos mûvelet a reszhalmaza. A reszhalmaza1(X,Y) igaz, ha X minden eleme eleme az Y listának is. Ennek a problémának a megoldásához ellenõriznünk kell X minden elemére, hogy az (az elõzõ szabály szerint) bennevan az Y listában is. A program akkor áll le igaz értékkel, ha nincs már több elem az X listában, ellenkezõ esetben hamis értéket ad a leállása után. Maga a program a következõ:

```
reszhalmaza1([],_).
reszhalmaza1([X|L],Y) :- bennevan1(X,Y), reszhalmaza1(L,Y).
```

ahol [] az üres listát jelenti.
Helyezze be ezt a szabályrendszert abba az állományba, amelyben a **bennevan1** is van, és hajtassa végre a következõ lekérdezéseket:

- reszhalmaza1([a,b,c],[x,a,f,b,g,h,c]).
- reszhalmaza1([1,2,100],[100,2,1,3]).
- reszhalmaza1([1,2,100],[100,2]).

5.2.3 Tagadás

Elõfordulhat, hogy úgy találja, jobb lenne, ha egy feltétel NEM teljesülne. Ezt a Prologban a **not** meta-kijelentéssel valósíthatjuk meg. A Prologban a **not** azt jelenti, hogy: *"amennyiben a program tényei és szabályai szerint az, hogy ezt a feltételt NEM lehet teljesíteni: IGAZ, akkor a feltétel minden bizonnyal HAMIS"*. Ez a *zárt világ feltételezése*.

Például egy szobát (négy falával) a következõképpen definiálunk: szoba(E,D,N,K), ahol E északit, D délit, N nyugatit és K keletit jelent, és bármelyik változó felveheti a következõ értékeket:

- 'ajto' ha ott ajtó van,
- 'ablak' ha ott ablak van,
- 'fal' ha ott csupasz fal van

Ha egy olyan szobát akarunk, amelynek nincs északra nyíló ajtaja, akkor a következõ szabályt kell megfogalmaznunk:

```
szoba1(E,D,N,K) :- szoba(E,D,N,K),
                   not (E=ajto).
```

ahol a szoba1 csak olyan szobákra vonatkozhat, amelyeknek nincs északi oldalukon ajtó. Mielõtt ellenõriznénk ennek a szabálynak a mûködését, definiálnunk kell a szoba kijelentést. Definiálhatnánk egy halom tényállítást az ajtónak, az ablaknak és a csupasz falaknak az összes lehetséges irányban való elhelyezésével. De ahelyett, hogy ezt a 81 tényállítást leírnánk, elõállíthatjuk ezeket a következõ kijelentéssel:

```
szoba(E,D,N,K) :- faltipus(E),
                  faltipus(D),
                  faltipus(N),
```

```
faltipus(K).
```

és a következõ tényállításokkal:

```
faltipus(ajto).
faltipus(ablak).
faltipus(fal).
```

Írja meg a program szabályait a szoba, szoba1 és faltipus kijelentésekre. Futtassa a következõ kérdésekkel (';'-vel az összes lehetséges válaszért):

- szoba1(E,ajto,N,K).
  ** ez lekérdezi az összes olyan elrendezést, amelyben a nem északi, hanem délre nyíló ajtajú szobák vannak, keletre és nyugatra pedig bármi állhat. Figyelje meg a kérdés második, konstans 'ajto' értékû argumentumát.
- szoba1(E,D,N,K).
  ** Ez a lekérdezés megadja az összes olyan elrendezést, amelyben az ajtó nem északi fekvésû.

5.2.3.1 - 4. példa

Módosítsa a fenti programot azzal, hogy legyen egy szoba2, amelynek pontosan egy ajtaja van, de az vagy déli vagy északi lehet csak.

5.2.3.2 - 5. példa

Adjon hozzá még egy faltipust és futtassa az egészet újra, hogy lássa a kombinációk sokkal nagyobb számát.

Érdekes megfontolni, hogyan kombináljuk össze a tagadást a rekurzióval. Tegyük fel, hogy meg akarjuk számolni egy adott lista adott tulajdonságú elemeinek a számát. A lista_elemszam(L,Q,E) az L lista Q tulajdonságú elemeinek E számával tér vissza. A megvalósítás egy módja a következõ:

```
lista_elemszam([],_,0).
lista_elemszam([X|L],X,N) :- lista_elemszam(L,X,N1), N is N1 + 1.
lista_elemszam([Y|L],X,N) :- not (X=Y), lista_elemszam(L,X,N).
```

Ezek a szabályok a rekurzió egy érdekes formáját használják, amikoris a rekurzióból való visszatéréskor valamilyen mûveletet hajt végre. Ebben az esetben a rekurzióból való kilépéskor számlálja a megfelelõ tulajdonságú elemeket.

A program teljes megértéséhez nézzük sorra az egyes szabályokat. A

```
lista_elemszam([],_,0).
```

üres lista esetén befejezi a rekurziót és nullát ad eredményül.
A

```
lista_elemszam([X|L],X,N) :- lista_elemszam(L,X,N1), N is N1 + 1.
```

szabály akkor teljesül, ha a keresett X elem a lista fejeleme is. Ebben az esetben a lista_elemszam ismét meghívásra kerül a lista fejelem nélküli L maradékával. Ha

ebbõl a rekurzióból visszatér, akkor eggyel növeli az L-ben lévõ megfelelõ elemek számát tartalmazó N1 értékét, és ezt az összeget teszi az N-be válaszként.

Végül a

```
lista_elemszam([Y|L],X,N) :- not (X=Y), lista_elemszam(L,X,N).
```

csak rekurzív lista_elemszam hívást tartalmaz abban az esetben, amikor a lista Y fejeleme nem egyenlõ a keresett X elemmel. Eltérõen az elõzõ szabálytól, most nem csinál semmit amikor a rekurzióból visszatér.

Írja be a lista_elemszam szabályokat a programba és futtassa a következõ kérdések kiadásával:

- lista_elemszam([a,b,c,a,f],a,N).
  ** leszámolja az 'a' elemeket a [a,b,c,a,f] listában
- lista_elemszam([a,b,c,a,f],g,N).
- lista_elemszam([john,peter,jose,mary,peter],mary,N).

5.2.4 Az eredmények kijelzése

A legtöbb esetben a keresõ rendszerek megadják a minket érdeklõ lekérdezések eredményeit. De ha mi a választ egyéni formában és tartalommal akarjuk, akkor szükségünk lesz az elõre definiált *write* kijelentésre. Például a szoba(E,D,N,K) kijelentés eredményét kiírhatjuk a szoba_kiiras(E,D,N,K) szabállyal:

```
szoba_kiiras(E,D,N,K) :- write(E), write(' '),
                         write(D), write(' '),
                         write(N), write(' '),
                         write(K), nl.
```

Ez a szoba falainak típusait egy-egy szóközzel elválasztva írja ki. Az 'nl' egy soremelést hajt végre a standard kimeneten.

5.2.4.1 - 6. példa

Ebben a feladatban használhatja a már definiált

- szoba
- faltipus
- lista_elemszam
- szoba_kiiras

szabályokat.

Definiáljon egy külön szabályt a nappali szobára! A nappali szobának pontosan két ajtaja van.

Készítsen egy másik szabályt egy kétszobás lakásra, ahol az egyik egy nappali, a másik egy szokásos szoba. A szobáknak egy közös ajtajuk van. Írassa ki a szobák falainak elhelyezkedését a következõhöz hasonlóan:

```
terv1(X).
   nappali szoba d d w w
   masik szoba d w w w


  X = 'tovabbi valasz ?' ;
   nappali szoba d d w w
   masik szoba d w w d


  X = 'tovabbi valasz ?' ;
   nappali szoba d d w w
   masik szoba d w w x


  X = 'tovabbi valasz ?'
```

Használja a ';'-et az összes válasz lekérdezéséhez! Ehhez efféle kijelentést kell készítenie:

```
terv1(X) :- ..... valami van itt.. .....
        X = 'tovabbi valasz ?',
        ..... tovabbi sorok vannak itt.....
```

5.2.5 Levágás és kudarc

Az eddigiekbõl láthatja, hogy a Prolognak van egy beépített rendszere a megoldások és bizonyítások megkeresésére. Alapesetben ezek az eljárások elegendõk a lehetnek minden probléma megoldására, de a hatékonyság és a kisebb kódméret kedvéért két meta-kijelentés is rendelkezésünkre áll a Prolog mûködésének befolyásolására: a **cut** (levágás) és a **fail** (kudarc).

5.2.6 Levágás

A felkiáltójellel jelölt levágás a visszalépéses keresés leállítására szolgál, és akkor használjuk, ha egy ponttól kezdve bármilyen eset kudarchoz vezetne. Például az alábbi bennevan(E,L) szabály akkor teljesül, ha az E elem benne van az L listában:

```
bennevan(X,[X|L]) :- !.
bennevan(X,[_|L]) :- bennevan(X,L).
```

Az elsõ szabályban lévõ levágás szerint *ha egyszer már megtaláltuk az elemet a listában, akkor nem szükséges visszalépni és további megoldásokat keresni*. Egy másik egyszerû példa a max(X,Y,Z) kijelentés, amely Z-ben megadja az X vagy az Y közül a nagyobbat. A kijelentés szabályai a következõk lehetnek:

```
max(X,Y,X) :- X >= Y.
```

```
max(X,Y,Y) :- X < Y.
```

Ha megfigyeli, akkor rájöhet, hogy ezek a szabályok egymást kölcsönösen kizárják. Ha az elsõ teljesül, akkor a második nem és fordítva. Elkerülendõ a késõbbi problémákat, lehetséges (és haszontalan) visszalépéseket, a jobb szabályok ezek:

```
max(X,Y,X) :- X >= Y, !.
max(X,Y,Y).
```

Ha az elsõ szabály teljesül, akkor a Prolog nem keres további megoldásokat a második szabály segítségével. A levágás használatának van még sok más, bonyolultabb oka is.

KUDARC

A **fail** (kudarc) egy a Prologba beépített kijelentés, amely hamis logikai értéket ad a végrehajtása esetén. Ez azt is jelenti, hogy a Prolog visszatér az ezt megelõzõ esetvizsgálathoz. Például álljon a következõ program:

```
allat(koala).
allat(tigris).
allat(kutya).

allat(X), write(X), nl.
```

csak a *koala* választ adja eredményül a képernyõre. Módosítjuk a programot azért, hogy a visszalépéses kereséssel minden megoldást kiírjon a program:

```
allat(koala).
allat(tigris).
allat(kutya).

allat(X), write(X), nl, fail.
```

Mikor a Prolog végrehajtja a fail kijelentést, visszatér további megoldásokat keresni. A write és nl beépített kijelentések nem adnak további megoldásokat, de az allat() szabály más esetben is teljesülhet. A Prolog addig keres újabb és újabb megoldásokat amíg csak lehetnek, azután áll csak le azzal, hogy nincs megoldás. Ezért lát ennek végrehajtásakor No -t a program végrehajtása után. Hogy ezt a 'No'-t ne kapjuk meg, adjunk hozzá egy üres 'run' szabályt is:

```
allat(koala).
allat(tigris).
allat(kutya).

run :- allat(X), write(X), nl, fail.
run.
```

amely így örökké teljesül.

5.3 A TURBO-PROLOG BETÖLTÉSE, INDÍTÁSA ÉS MENÜRENDSZERE.

A

C:>prolog

billentyûzéssel betöltõdik és el is indul a program. A menüsor alsó sorában vannak az aktuális legfontosabb parancsok jelei, melyek zöme mindig valamelyik F funkcióbillentyû.

A menü a szokásos módon, a vízszintes kurzorvezérlõ billentyûkkel és az ENTER lenyomásával mûködik, amire általában egy újabb, most már lefutó menüben találjuk magunkat. Ezek közül a függõleges kurzorvezérlõkkel és az ENTER-rel választhatunk.

A programozás elsõ (technikai) lépése a program szövegének elõállítása, vagyis a forráskód megírása. Erre használjuk az EDIT funkciót. A program lefordítása a COMPILE funkcióval lehetséges, aminek eredménye egy *.obj kiterjesztésű tárgykód. Hiba esetén kijelzi a hiba helyét és az észlelt hibát, bár ez nem garancia arra, hogy az igazi hiba is ott van. Hiba esetén -még a menüpont hatásköre alatt- kijavíthatjuk a hibát, az F10-re pedig befejezi a fordítást.

A memóriában lévõ program futtatása a RUN opció alatt történik. Ha elõtte nem fordítottuk le a programot, vagy a legcsekélyebb mértékben megváltoztattuk, akkor a futást megelõzi a fordítási procedúra. A futási kép a legegyszerûbb esetben a DIALOG nevû ablakban történik, erre akkor kerül sor, ha nem határoztunk meg ablakot, vagy ablakokat a felhasználó és program közötti adatcerére. A programszöveg mentésére, betöltésére, átnevezésére, az aktuális drive és alkönyvtár változtatására és az operációs rendszerbe történõ ideiglenes kilépésre a FILE funkció áll rendelkezésünkre.

5.4 A LOGIKAI NYELVEKRE VONATKOZÓ NÉHÁNY FONTOSABB DEFINÍCIÓ

A logikai program alapstruktúrája a kifejezés. A kifejezés lehet konstans (állandó), változó vagy összetett kifejezés. A konstansok számok vagy elemi (atomi) jellegûek lehetnek, míg a változók egyedi, de nem specifikált egyedeket jelentenek. Az "atomok" szimbóluma bármilyen karakterlánc lehet, amely kizárja bármely más (változó vagy szám) szimbólummal való összetévesztés lehetõségét. A változók neveit a kezdõ nagybetû különbözteti meg.

A szabály egy függvénynevet (a szabály megkülönböztetõ jelét) és egy vagy több kifejezést, mint argumentumot tartalmaz. A szabály jellemzõi tehát a neve, amely tovább nem bontható, és a kiterjedése (dimenziója), vagyis az argumentumainak a száma.

A tények nulla változós szabályoknak tekinthetõk.

Szintaktikailag a szabály formája:

f(t1,t2,...,tn) ahol a szabály neve f, dimenziója n, és az argumentumai ti.

A logikai program szabályok véges készlete. A feltétel vagy szabály

A :- B1,B2,...,Bk. k>0

általános alakú mondatok, ahol A és Bi célkifejezések. Az ilyen mondatok kijelentõ módú deklaratív olvasása: A tartalmazza Bi együttes teljesülését. Végrehajtás szempontjából procedurálisan ugyanez így hangzik: az A kérdésre válaszolni annyi, mint válaszolni a B1,B2,...,Bk kérdésekre. "A" a szabály feje, míg Bi a teste.

Ha k=0, akkor a szabályt ténynek nevezzük, és A.-tal jelöljük.
Ha k=1, akkor a szabályt iteratív szabálynak nevezzük.

A kérdés az A1,...,An? n>0 alakú kifejezés, ahol A célkifejezések. A kérdésben lévõ változók értelme azok létezésének meghatározása.

A P logikai program lefuttatása egy olyan eset keresése, amely az adott kérdésre a programból logikailag következik. A G célállítás következik a P programból, ha a G-nek van olyan A esete, ahol

A:-B1,...,Bn, n>0,

A a P program egy szabályának alaphelyettesítése és Bi-k a P programból következnek.

5.5 A PROLOG-PROGRAM TULAJDONSÁGA

A jól ismert Pascal, a BASIC és a többi hagyományos programozási nyelv procedurális: a programozónak kell lépésrõl lépésre leírnia az eljárásokat, amelyek megmondják a számítógépnek, hogy hogyan kell a problémát megoldani. Ezzel szemben a PROLOG deklaratív nyelv. Ez azt jelenti, hogy megadva a szükséges tényeket és szabályokat, képes deduktív következtetésekkel megoldani a programozott problémákat.

A PROLOG-programozónak csak a probléma leírását (goal=cél) kell felállítania a megoldás alapszabályaival, és a PROLOG rendszer fogja a megoldáshoz vezetõ utat meghatározni. A célkifejezésrõl elõbb-utóbb kideríti a Prolog rendszer, hogy az adott körülmények között teljesül-e vagy nem, illetve milyen feltétellel teljesülhet.

A célkifejezés vizsgálatának tehát kétféle módja van:

- ha a célkifejezésben lévõ szabály csak konstansokat tartalmaz, akkor a rendszernek nincs más dolga, mint addíg helyettesítgetni a megengedett tényállításokat a beépített backtrack (visszalépés) algoritmus segítségével, amíg a szabály nem teljesül, amikoris válaszként közli, hogy TRUE, ellenkezõ esetben, tehát ha az összes lehetõség kimerült, és még mindig nem volt sikeres a keresés, a válasz FALSE és

- ha a célkifejezésben van változó, akkor a rendszernek az a feladata, hogy ennek keressen legalább egy olyan értéket, amely kielégíti a célt, bizonyos körülmények között az összes megoldást megkapjuk.

Hogyan kapcsolódik össze procedurális és a deklaratív gondolkodásmód a logikai program építésénél? Célszerûségbõl általában mindenki procedurálisan gondolkodik, amikor programoz. Azonban deklaratívan gondolkodik, amikor a program kívánt és a megvalósított jelentésének eredményét veti össze. Ezek keverésének egy módja a logikai programozásban a program szabályait procedurálisan összeállítani, majd az eredményt mint deklaratív állítást értelmezni. Építsük fel tehát procedurális gondolkodásmóddal a programot, majd mérlegeljük, hogy a különbözõ alkalmazások során az milyen deklaratív értelmet kap. Ezt majd a feladatokon keresztül be is mutatjuk.

5.6 A TURBO-PROLOG PROGRAM SZERKEZETE

Nagy hatással volt a programok világos szerkezetének kialakításában a felülrõl lefelé (top – down) való kivitelezés metódusának hangsúlyozása, amit a lépésenkénti finomítás elvével együtt használunk. Nagy vonalakban ez a metódus azt jelenti, hogy felállítjuk a fõ problémát, széttördeljük alproblémákra, majd megoldjuk ezeket a részeket. A felülrõl lefelé való programozás természetes módja a logikai programok felépítésének. A közölt programleírások is kizárólag felülrõl lefelé haladnak.

Egy Prolog program tények és szabályok alapján keresi a megoldást. Ehhez ismernie kell a szabályok szerkezetét, és a szerkezeti elemek minõségét. Az igazán teljes Prolog rendszerek ezt automatikusan elvégzik, bár így sok hiba rejtve is maradhat.

Valószínûleg a takarékos memóriagazdálkodás kényszerûsége és a hibák korai kiszûrése miatt a Turbo-Prolog megköveteli az elemi objektumok és a szabályok formai deklarációját.

Ennek két lépcsõje:

domains

Az elemi objektumok típusainak deklarációja. Ez a rész nem kötelezõ. Ha a szabályok leírásánál (predicates, lásd késõbb) csak a rendszer által biztosított alaptípusokkal definiálunk, akkor kihagyható. Ellenkezõ esetben -és a beszédes változónevek elve betartásakor méginkább- nem kerülhetjük meg. Az sem árt, hogy a paraméterek átadásakor ezáltal elkerülhetjük a nem kívánt adatkeveredést is.

Az elemi objektumok nevei kisbetûkkel kezdõdnek. A név és egy egyenlõségjel után vagy egy már létezõ standard változótípus nevét, vagy egy konstanskifejezést kell tenni, ahogyan az az alábbi példák mutatják.

A standard típusok a következõk:

integer, real, char, symbol, string.

Például

elem = symbol

szam = integer
nagyszam = real
i,j,k = integer
rud = kozepso

Ezek után tehát például az elem szó felhasználható a szabályok megfogalmazásánál olyan pozíciókban, ahol szimbólumokat kívánunk elhelyezni, illetve ahol szimbólumokat várunk eredményként. Az elmondottak szerint a rud olyan típus, ahol az egyetlen behelyettesíthetõ érték a kozepso szó.

Ennek persze nem sok értelme lenne, ezért rögtön ejtsünk szót egy fontos lehetõségrõl, amelyet alternatív deklarációnak nevezhetünk. Itt olyan deklarációról van szó, ahol egy típusnév többféle típust jelenthet. Legegyszerûbben ezt konstansokkal szemléltethetjük:

rud = jobboldali ; kozepso ; baloldali

Most a rud típus háromféle értéket kaphat, ezeket a pontosvesszõ választja el egymástól. Az alternatív deklaráció kezelése nagy óvatosságot, gyakorlatot igényel.

A meglévõ típusokkal újabb deklarációk építhetõk fel, mint azt az alábbi példa mutatja:
elem = symbol
ujelem = elem

Mód van arra, hogy egy meglévõ típust felhasználjunk összetettebb adatszerkezet elõállítására. Ennek legegyszerûbb és elõnyben részesített módja a lista definiálása. A későbbiekben látni fogjuk, hogy a lista, illetve a listával való műveletek a Prolog egy fontos tulajdonsága.

Ha egy szabály egyik paramétere nem elemi objektum, hanem egy lista, akkor definiálni kell egy ilyen típust a következõ módon:

adat = symbol*

Az adat nevû típus ezek után olyan adatszerkezet, amelyben symbol típusú elemi objektumok vannak listaszerûen összefûzve. Az sem helytelen, ha a lista helyett a verem (stack) fogalmával azonosítjuk a Prolog-beli listát, mert közvetlenül csak a lista elsõ tagja (a verem legfelsõ eleme) kezelhetõ, illetve választható le a listáról, tehát inkább veremszerûnek képzelhetõ el ez az adatszerkezet. Jelölésmódja egyszerû: ha az elemeket a,b,c,d-vel jelöljük, akkor elfogadottak az alábbiak:

["a","b","c","d"]
["a"|["b","c","d"]]
["a"|["b"|["c"|["d"]]]]

Az üres lista jele a [], amely minden lista végére implicite odaértendõ.

predicates

A szabályok felsorolása a formalizmus rögzítésével. Ebben a szekcióban azokat a szabályokat kell bevezetnünk, amelyeket alkalmazni fogunk. Ezzel rögzítjük az alkalmazott szabályok neveit és a használható paraméterek típusait.

Legalább egy ilyen szabálynak kell lennie, máskülönben nem is beszélhetünk Prolog programról. Néhány példa:

```
szabaly1(symbol)
szabaly2(integer,integer,symbol)
masik_szabaly(real,symbol)
uj_szabaly(adat,symbol,adat)
```

A deklarációval foglalkozó szekciók után most a további program-egységek következnek. Ezek közül az elsõ a program célját van hivatva rögzíteni.

goal

A program addíg fut, amíg ez a cél nem teljesül, vagy meg nem állapítja, hogy sosem teljesülhet. Ez a rész is elmaradhat, de a léte vagy nemléte erõsen befolyásolhatja a program mûködését. Ha van, akkor pontosan azt teszi, amire az ezt követõ utasítások kényszerítik, és az elsõ kielégítõ eset után leáll (hacsak másra nem utasítjuk). Ha nincs, akkor a menübõl választott RUN parancsra a DIALOG ablakban megjelenik a

```
GOAL=?
```

kiírás, ami után az aktuális cél, másnéven a kérdés begépelése következhet. Maga a program tehát csak egy bizonyos kérdés megválaszolásáért hajlandó futni.

A cél elérésére két fontos, a nyelvbe beépített mechanizmus mûködik. Az egyik a mintaillesztés, amely két kifejezés (az elérendõ cél és a rendelkezésre álló szabályok egyike) közötti, argumentumonkénti összehasonlításra szolgál. A másik a backtrack algoritmus, amely a megoldáshalmaz összes elemét igyekszik megtalálni. Ha a lehetséges összes (jó és rossz) megoldást egy bináris fa ágaiba és leveleibe helyezzük, akkor az említett algoritmus a fa módszeres bejárását végzi, minden megoldás után visszalépve az utolsó elágazásra ahonnan újra próbálkozik másik megoldást találni.

clauses

A Prolog program utolsó része a tények és szabályok felsorolása. A tények és szabályok tetszõleges sorrendben követhetik egymást, az azonban fontos, hogy az azonos nevû szabályok egymást követõen álljanak, az azon belüli sorrend viszont már a végrehajtás sorrendjét fogja befolyásolni.

A szabályok formai megjelenésben lényegében kétfélék.

Az egyik neve tényállítás. Ez konstans argumentumú feltétel nélküli szabály, a szabály nevével és argumentumlistával, a végén egy pont. A másik egy olyan kifejezés, amely egy vagy több kisebb célállítás logikai kapcsolatából áll. A részcélok argumentumlistáiban szerepelhet egy vagy több változó, és az azonos nevû változónevek egy szabályon belül mindig ugyanazt a behelyettesítést jelentik. A szabály teljesülése feltételektõl függ.

A feltételek az if kulcsszó vagy a vele ekvivalens :- jel után állnak, és több szabály esetén logikai kapcsolatukat is meg kell jelölnünk. Ilyen logikai alapszavak és a vele ekvivalens írásjelek a következõk:

and illetve , (vesszõ)
or illetve ; (pontosvesszõ)
not

Néhány Prolog implementáció csak az egyik formát fogadja el, mások egy szabályon belül csak egyfajta jelölést engednek meg, a Turbo Prolog viszont teljes szabadságot ad a program szerkesztõinek e kérdésben. Stilisztikai szempontból azonban jó, ha egységes jelölésrendszernél maradunk, én például a kulcsszavas írásmódhoz igyekszem ragaszkodni, mert a szavak kifejezõbbek és fõleg kevésbé téveszthetõek, mint az írásjelek.

Természetesen a feltételek végén minden esetben ott kell lennie a szabályt lezáró pontnak.

5.7 A LEGGYAKRABBAN HASZNÁLT PROLOG ALGORITMUSOK

Az eldöntés programozási tétele arra ad algoritmust, hogy egy adatstruktúrában van-e T tulajdonságú elem.

A Prolog legtermészetesebb adatstruktúrája a lista, így a feladatot egy kicsit egyszerûsítve úgy fogalmazzuk át, hogy van-e a listában egy bizonyos elem? Lássunk egy kész megoldást:

```
domains

    elem = symbol
    lista = elem*

predicates

    bennevan(elem,lista)

clauses

    bennevan(E,[E|_]).
    bennevan(E,[_|Lv]) if bennevan(E,Lv).
```

A **domains** szekció deklarál egy elem nevû, tetszõleges szimbólumból álló elemi objektumot, valamint egy lista nevû, elem típusú tagokból álló listát.

A **predicates** egyetlen szabályt deklarál, amelynek a neve **bennevan**, két argumentummal, amelyek közül az elsõ **elem** típusú, a második pedig **lista** típusú. E deklaráció után ragaszkodnunk kell a paraméterek ilyen sorrendjéhez!

Az a tény, hogy nincs **goal**, vagyis a programban meghatározott célkifejezés, azt eredményezi, hogy egy RUN parancsra megkérdezi a célt. Ezt beadva fog csak mûködni a program.

A **clauses** szekció tartalmazza a **bennevan** szabályait. Az elsõ szabállyal nincs is semmi probléma: az **elem** típusú, tehát szimbólumot tartalmazó E változó benne van a listában, ha az **E** a lista feje, akármi is áll a lista további részében. Ha ez teljesül, akkor már nem kell semmi mást csinálni, ezt jelzi az, hogy elértük a szabály végét jelzõ pontot. Ez egy eklatáns példa az argumentumaikban kielégített szabályokra.

Aki írt már rekurzív programot, annak a második szabály nem különösebben meglepõ: az **E** elem benne van a listában, ha a lista elsõn kívüli részében benne van. Ezzel a lehetséges eseteket szétválasztottuk, hiszen egy lista a lista fejébõl és a lista elsõnkívüli elemeibõl álló listából áll. A szabály nemrekurzív elsõ sora elintézi az elsõ esetet, a második, rekurzív sor pedig a másodikat.

Procedurálisan úgy képzelhetjük el a program mûködését, hogy megnézi a lista elsõ elemét, és ha az egyenlõ a keresett szimbólummal, akkor a **bennevan**-ság teljesül. Ha nem, akkor keres egy újabb szabályt, ami most azt mondja ki, hogy akkor teljesül a **bennevan**-ság, ha a lista maradék részére az teljesül. Tehát újra kezdi a vizsgálatot, már egy rövidebb listával, és természetesen újból az elsõ szabályt próbálja alkalmazni. Ha valaha teljesül az egyezés, akkor visszalép a rekurzió egy magasabb szintjére, ahol most pontot talál. Ennek az a jelentése, hogy nincs további teendõ, tehát ismét visszalép, egészen addig, amíg a parancs-szintig el nem ér. Itt kiírja, hogy **True**, angolul ez igaz-at jelent. Ha a sorozatos rekurzív hívások által annyira lerövidül a lista, hogy már nincs benne semmi, csak az üres lista, akkor konstatálja, hogy nincs meg a keresett elem a listában, majd a rekurzív hívásokból visszalépve azt írja a képernyõre, hogy **False**. Nézzük a fenti program futását:

```
Goal:bennevan("f",["a","b","f","c"])
```

céllal, a TRACE bekapcsolásakor adott visszajelzésen keresztül:

```
CALL: bennevan("f",["a","b","f","d"])
REDO: bennevan("f",["a","b","f","d"])
CALL: bennevan("f",["b","f","d"])
REDO: bennevan("f",["b","f","d"])
CALL: bennevan("f",["f","d"])
RETURN:*bennevan("f",["f","d"])
RETURN: bennevan("f",["b","f","d"])
RETURN: bennevan("f",["a","b","f","d"]) True
```

Most egy olyan célállításról kérdezzük a gép véleményét, amely szemmel láthatóan kudarccal fog végzõdni:

```
Goal:bennevan("z",["a","b","c"])
```

A nyomkövetés eredménye az alábbi lista:

```
CALL: bennevan("z",["a","b","c"])
```

```
REDO: bennevan("z",["a","b","c"])
CALL: bennevan("z",["b","c"])
REDO: bennevan("z",["b","c"])
CALL: bennevan("z",["c"])
REDO: bennevan("z",["c"])
CALL: bennevan("z",[])
REDO: bennevan("z",[])
FAIL: bennevan("z",[]) False
```

5.7.1 szélsõérték-kiválasztás (legkisebb, legnagyobb elem keresése)

Ez a tétel tulajdonképpen az adathalmazból egy minimum- vagy maximumérték kiszámítására ad algoritmust. a Prologban nincs ciklusutasítás, ezért egészen más módon kell a problémát megoldani.

Az elsõ és legegyszerûbb szabály az lehet, hogy az egyelemû lista egyetlen eleme maga szélsõérték is:

```
maximum([E],E).
illetve
minimum([E],E).
```

Az alkalmazott E változó olyan típusú, amilyent a szabály e helyére deklaráltunk, értéket pedig akkor kap, ha a szabály ilyen formája valaha teljesül. Konkrétan tehát akkor, amikor olyan paraméterekkel hívjuk meg a szabályt, hogy a lista az E elembõl áll, és az elem (amit a második helyre tettünk), ugyanez az elem. A következõ két-két szabály gondoskodik az összes további esetrõl. Az második szerint a lista E feje nem a szélsõérték, mert a lista E nélküli részének F maximuma vagy minimuma nagyobb vagy egyenlõ, illetve kisebb vagy egyenklõ, mint az E.

```
maximum([E|List],F) if maximum(List,F) and F >= E and !.
illetve
minimum([E|List],F) if minimum(List,F) and F <= E and !.
```

A sorok végén álló felkiáltójelekrõl még sok szó esik, most csak néhány szót a szerepérõl. A ! funkció neve cut, ami vágást, elvágást jelent, és értelme az, hogy a lehetséges megoldás keresését ezen az úton már nem engedjük meg. A harmadik szabály szerint a lista feje szélsõérték, mert az E fej nélküli lista F szélsõértéke kisebb illetve nagyobb, mint E.

```
maximum([E|List],E) if maximum(List,F) and F < E.
illetve
minimum([E|List],E) if minimum(List,F) and F > E.
```

Nézzük ezek után a két szélsõérték kiszámítását végzõ programot!

```
domains

    elem = symbol
```

```
    lista = elem*

predicates

    maximum(lista,elem)
    minimum(lista,elem)

clauses

    maximum([E],E).
    maximum([E|List],F) if maximum(List,F) and F >= E and !.
    maximum([E|List],E) if maximum(List,F) and F <= E.
    minimum([E],E).
    minimum([E|List],F) if minimum(List,F) and F <= E and !.
    minimum([E|List],E) if minimum(List,F) and F > E.
```

A teljes program két különbözõ funkcióra oszlik. Fontos formai szabály, hogy az összetartozó szabályok szigorúan egymást követõen helyezkedjenek el. Nagyon fontos ezen belül még az is, hogy a szabályok milyen sorrendben követik egymást, hiszen a már említett szekvenciális végrehajtási mód miatt az oly szokványos rekurzív hívások mindig elölrõl kezdik értékelni a szabályokat.

Ilyen egyszerû relációk leírása esetén az esztétikus és elegáns axióma-felállítás tûnik a nyilvánvalóan leghelyesebbnek. Valóban, de az elegancia legtöbbször nem adódik azonnal, hanem sok gondolkodás gyümölcse. Prologban is lehet ugyanarra a feladatra többféle megoldás, ez általában a programozótól függ. A végeredmény eleganciája viszont már sokat sejtet a megoldás jóságáról és a programozó gyakorlatáról, de fõleg a képességérõl.

5.7.2 Összegzés

Egy listában elhelyezett számsorozat összegét kell most kiszámítanunk. Mivel itt az elemek nem akármilyen szimbólumok, hanem számok, az elem sem symbol, hanem integer vagy real lesz.

```
elem = integer
lista= elem*
```

A szabály alakja hasonló az elõzõekhez:

```
osszegzes(lista,osszeg)
```

Maga a szabály rekurzív. Elsõ sora (a vészkijárat...) azt mondja, hogy egy egyelemû lista összege maga az elem értéke:

```
osszegzes([E],E).
```

A második szabály pedig azt mondja ki, hogy egy lista összege úgy számítandó ki, hogy a lista fejének értékét hozzáadjuk a lista farkának összegéhez:

```
osszegzes([Fej|Farok],Osszeg) if Osszegzes(Farok,Reszeredmeny)and
                                Osszeg=Fej+Reszeredmeny.
```

Deklaratív olvasata a szabálynak az, hogy az összegszámítás úgy teljesülhet, hogy az Osszeg a Fej és a Farokra vonatkozó részeredmény összege. A teljes program tehát:

```
domains

    elem = integer
    lista = elem*

predicates

    osszegzes(lista,elem)

clauses

    osszegzes([E],E).
    osszegzes([Fej|Farok],Osszeg) if osszegzes(Farok,Reszeredmeny) and
                                    Osszeg=Fej+Reszeredmeny.
```

A program nem tartalmaz célállítást, tehát a RUN parancsra egy kérdést vár. Legyen ez a

```
Goal:osszegzes([1,3,5,7],Sum)
```

A cél most az, hogy meghatározzuk, ez az állítás milyen feltétel mellett teljesül. A Prolog rendszer azt válaszolja, hogy ez csak a Sum=16 mellett teljesül, egyben választ is adva számunkra. Ha nem változót, hanem konstanst, azaz egész értéket teszünk az összeg helyére, akkor a célállítást másképpen értelmezi a rendszer:

```
Goal:osszegzes([1,2,3],6)
```

hatására azt vizsgálja, hogy igaz-e az állítás, azaz tényleg 6-e az összeg. Mivel ezt igaznak találja, azt írja ki, hogy True, és befejezi a program futását. Hasonlóképpen értelmezi a feladatát akkor, ha a

```
Goal:osszegzes([1,2,3],5)
```

célállítással kerül szembe, de ekkor az állítást nem tudván soha kielégíteni, a válasza False. Nem szabad viszont az elsõ pozícióra változót helyezni, mert a szabályokban szabad, tehát nem levezethetõ változók szerepelnének.

5.7.3 elem törlése listából

Nem tartozik a klasszikus programozási tételek közé, de egyike a legszükségesebb és legtanulságosabb programoknak, így itt kell róla beszélnünk. Az elsõ és legfontosabb

lépés meghatározni a reláció kívánt értelmezését. Világos, hogy a lista egy elemét törlõ programnak három argumentuma lesz:

a törlendõ E elem,
az E elemet tartalmazó L1 lista és
az E elemet sehol sem tartalmazó L2 lista.

Az alkalmas reláció alakja tehát:

torol(L1,X,L2).

Szöveges értelmezése pedig az, hogy L2 minden esetben L1, elhagyva E minden elõfordulását. A program átgondolásánál a legegyszerûbb egy speciális esetre gondolni. Gondoljuk végig a

torol([a,b,c,b],b,L)

kérdés megoldását, amely tipikus esete az elem törlése után maradó lista megkeresésének. A válasz itt

L=[a,c].

A program rekurzív lesz az elsõ argumentumra nézve. Vegyük elõ a procedurális gondolkodásmódunkat! Kezdjük a rekurzív résszel.

A listákra általánosan használt rekurzív forma az [E|Lista]. Két lehetséges eset van:

az egyik, mikor az E fej-elem a törölni kívánt elem,
a másik, mikor nem az.

Az elsõ esetben E-nek a Lista listából való rekurzív törlésének eredménye adja a kielégítõ eredményt a kérdésre. Az alkalmas szabály tehát:

torol([E|Lista],E,Maradek]) if torol(Lista,E,Maradek).

Gondolkodásmódot váltva, a szabály deklaratív olvasata ez: E-nek [E|Lista]-ból való törlése Maradek, ha E-nek Lista-ból való törlésének eredménye Maradek.

A feltétel, hogy a lista feje és a törlendõ elem ugyanaz, a szabály fejében lévõ E változó által biztosított, hiszen ha a szabály igaznak bizonyul, akkor azonos változónév helyén csak azonos értékek állhatnak.

A második esetben, ahol a törlendõ elem különbözik F-tõl, azaz a lista fejétõl, egyszerû. A kívánt eredmény egy lista, amelynek a feje F és farka az elem rekurzív törlésének eredménye. A szabály:

torol([F|Lista],E,[F|Maradek]) if E<>F and torol(Lista,E,Maradek).

A szabály deklaratív olvasata: E törlése [F|Lista]-ból [F|Maradek], ha E különbözik F-tõl és E Lista-ból való törlésének eredménye Maradek. Az elõzõ szabállyal ellentétben

a feltétel, hogy a lista feje és a törölni kívánt elem különbözik, explicit módon ki van fejezve a szabály törzsében, különben az E és az F változók felvehetnék ugyanazt az értéket is.

Az alapeset (vészkijárat...) egyértelmû. Üres listából nem lehet elemet törölni, és az elvárt eredmény szintén az üres lista. Ez adja az utolsó szabályt:

```
torol([],_,[]).
```

Az aláhúzásjel a középsõ paraméter helyén azt jelenti, hogy itt akármi állhat, a szabály szempontjából közömbös. Így lehet kivédeni a rendszer egy hibaüzenetét, mely szerint a torol([],E,[]). szabályban az E csak egyszer szerepel, így definiálatlan. A teljes program tehát:

```
domains

    elem = symbol
    lista = elem*

predicates

    torol(lista,elem,lista)

clauses

    torol([E|Lista],E,Maradek]) if torol(Lista,E,Maradek).
    torol([F|Lista],E,[F|Maradek]) if E<>F and torol(Lista,E,Maradek).
    torol([],_,[]).
```

Nézzük át újra a megírt programot, és próbálkozzunk alternatív megoldásokkal. Elhagyva a második szabályból az E<>F feltételt, egy másik variánst kapunk a törlõ programunkra. Ez a variáns kisebb megszorítást tartalmaz, értelmezése tágabb lesz, mint a minden elõfordulást törlõ teljes programnak. Például a

```
torol([a,b,c,b],b,[a,c]),
torol([a,b,c,b],b,[a,c,b]),
torol([a,b,c,b],b,[a,b,c])
```

mindegyike benne van a variáns értelmezésében, tehát célként megadva a ezeket, a válasz mindig True. Mind a fenti, mind a teljes program értelmezési tartományába beletartozik az az eset, amikor a törölni kívánt elem nincs benne egyik listában sem, például a

```
torol([a],b,[a])
```

igaz állítás. Vannak esetek, amikor ez nem kívánatos.

5.7.4 Összefuttatás

Ez a tétel már egy kissé bonyolultabb. Adott két rendezett lista, ezekbõl kell egy olyan listát készíteni, amely továbbra is rendezett és tartalmazza a két kiinduló lista elemeit, az azonosakat persze csak egyszer.

A program a következõ:

```
domains

    elem = symbol
    lista = elem*

predicates

    osszefut(lista,lista,lista)

clauses

    osszefut([X|Xs],[Y|Ys],[X|Zs]) if X<Y and ! and osszefut([X|Xs],Ys,Zs).
    osszefut([],[],[]).
    osszefut(Xs,[],Xs).
    osszefut([],Ys,Ys).
```

Vegyük sorra a szabályokat!

```
osszefut([X|Xs],[Y|Ys],[X|Zs]) if X<Y and ! and osszefut([X|Xs],Ys,Zs).
```

Vázlatosan: két nem üres lista összefuttatását úgy kapjuk meg, hogy ha az elsõnek a fej-eleme kisebb, ez lesz a fej-eleme annak az eredmény-listának, amit a Harmadik szabályunkban pontosan az történik, ami az elsõben, csak most a két lista szerepe felcserélõdött.

```
osszefut([],[],[]).
osszefut(Xs,[],Xs).
osszefut([],Ys,Ys).
```

Az utolsó három szabály "csupán" triviális dolgokat mond. De pontosan ezek a trivialitások vetnek véget a rekurzív hívásoknak, és kezdik visszafejteni a megoldást. A szabályok szerint az üres listák összefuttatásának eredménye az üres lista, valamint egy üres lista és egy tetszõleges lista összefuttatásának eredménye maga az említett lista, akár az elsõ, akár a második helyen áll is a nemüres lista.

5.7.5 Únióképzés, metszetképzés

Az únióképzés tétele szerint az adott két -azonos elemeket egyenként nem tartalmazó- halmaz elemeit úgy kell összegezni, hogy az eredményhalmazban mindkét halmaz elemei szerepeljenek, de ne legyen benne két azonos elem. A metszetképzés

eredménye a közös elemeket egyetlen egyszer tartalmazó halmaz. A halmazokra nem teszünk más (pl. rendezettségi) kikötést.

A két tétel megoldása Prologban teljesen hasonló elveken nyugszik, ezért is vonható össze, mint az alábbi programlista is mutatja:

```
domains

    elem = symbol
    lista = elem*

predicates

    bennevan(elem,lista)
    nincsbenne(elem,lista)
    unio(lista,lista,lista)
    metszet(lista,lista,lista)

clauses

    bennevan(E,[E|_]).
    bennevan(E,[_|Lv]) if bennevan(E,Lv).
    nincsbenne(F,L) if not(bennevan(F,L)).
    unio([E|L1],L2,[E|L3]) if unio(L1,L2,L3) and nincsbenne(E,L3) and !.
    unio([_|L1],L2,L3) if unio(L1,L2,L3) and !. unio([],_,_).
    metszet([E|L1],L2,[E|L3]) if metszet(L1,L2,L3) and bennevan(E,L2) and !.
    metszet([_|L1],L2,L3) if metszet(Lista1,Lista2,Lista3) and !.
    metszet([],_,[]).
```

Kezdjük az elemzést a bennevan szabály ellentétével, a nincsbenne szabállyal! Ennek képzése egyszerûen a not beépített függvénnyel történhet. Ha teljesül a belsõ kifejezés, akkor hamissá válik az érték, míg a belsõ kifejezés kudarca esetén lesz kielégítve az egész szabály. Az únióképzés elsõ és legfontosabb szabálya szerint:

```
unio([E|L1],L2,[E|L3]) if unio(L1,L2,L3) and nincsbenne(E,L3) and !.
```

azaz az elsõ lista E feje akkor lesz eleme az eredménylistának, ha van olyan L3 lista, ami az elsõ lista maradéka és az L2 lista úniója révén keletkezik, és nincs benne E az L3 listában.

Az azonnali rekurzív hívás azt eredményezi, hogy az elsõ lista rögtön "elfogy", így végülis hátulról kezdi majd a következõ szabályok alapján felépíteni az eredménylistát.

```
unio([_|L1],L2,L3) if unio(L1,L2,L3) and !.
```

Ez a szabály az elõzõ kudarca esetén lép életbe, és azt fejezi ki, hogy az elsõ lista feje nem lehet eleme az eredménylistának, és az únióképzést a maradékkal kell folytatni.

```
unio([],_,_).
```

Itt van az a szabály, ami visszafordítja az elsõ két szabály rekurzív hívásait, mert azt mondja, hogy egy üres lista és egy tetszõleges lista úniója egyenlõ ugyanazzal a listával. Az metszetképzés az elõzõtõl csak nagyon kicsiben különbözik. Az elsõ szabály:

```
metszet([E|L1],L2,[E|L3]) if metszet(L1,L2,L3) and bennevan(E,L2) and !.
```

mely szerint az E elem benne van az eredménylistában, ha benne van a másik listában is, és az eredménylista további része metszete az elsõ lista maradékának és a második "érintetlen" listának.

```
metszet([_|L1],L2,L3) if metszet(Lista1,Lista2,Lista3) and !.
```

A metszetképzés szabálycsoportjának második sorára ugyanaz vonatkozik, mint az únióképzés megfelelõ sorára.

```
metszet([],_,[]).
```

E harmadik sornak a magyarázata is a metszetképzés definíciójára utal, hiszen egy üres lista és bármi másnak a metszete csakis az üres lista lehet. Éppen úgy, ahogyan az únióképzésnél, ennél is itt fordul vissza a rekurzív hívások sorozata. E két tétel egyszerûsége és hasonlatossága talán az Olvasót is meggyõzte a Prolog fölényérõl más nyelvekkel szemben a tiszta és az emberi gondolkodásmódot követõ feladatmegfogalmazása által.

5.7.6 Rendezés permutálással

A listák rendezésének logikai specifikációja az, hogy keressük a lista olyan permutációját, amely rendezett. Ez azonnal leírható,
mint egy logikai program. Az alapreláció:

```
rendperm(Xs|Ys)
```

ahol Ys az Xs elemeibõl álló növekvõen rendezett lista:

```
rendperm(Xs,Ys) if permutacio(Xs,Ys) and rendezett(Ys) and !.
```

A felkiáltójel (cut) letiltja a további megoldások keresését, hiszen annak nincs értelme. A rendezés legfelsõ szintjét lebontottuk, a felülrõl lefelé haladó programfejlesztés szellemében. Most már csak a permutacio és a rendezett szabályokat kell definiálnunk.

Annak eldöntése, hogy egy lista növekvõen rendezett-e, kifejezhetõ az alábbi két szabállyal. Az egyik kijelentés szerint az egyelemû lista szüségszerûen rendezett. A másik szabály szerint egy lista rendezett, ha az elsõ eleme kisebb vagy egyenlõ mint a második, és a maradék lista a második elemtõl kezdve rendezett.

```
rendezett([_]).
```

```
rendezett([X|[Y|Ys]]) if X<=Y and rendezett([Y|Ys]).
```

A permutálást végzõ program még érdekesebb. A permutálási eljárás egy megoldása az lehet, hogy véletlenszerûen választunk egy elemet a listából, azt a lista elejére helyezzük, majd rekurzívan permutáljuk a maradék listát. Az alapkijelentés szerint az üres lista egyetlen permutációja önmaga.

```
permutacio(Xs,[Z|Zs]) if kiejt(Z,Xs,Ys) and permutacio(Ys,Zs).
permutacio([],[]).
```

A **kiejt** szabály szerepe az, hogy a lista elejétõl kezdve a vizsgálatot, az elsõ kívánt elemet kiemelve adja vissza a bemenõ listát. Tehát ez nem azonos azzal a programmal, amely az összes elõfordulást kitörli, csak hasonlít rá.
Deklaratív olvasásban így hangzik: E-t kiemelhetjük az [E|Lista] listából a Lista visszaadásával, vagy E-t kiemelhetjúk az [F|Lista1]-ból, ha megadjuk az [F|Lista2] listát, ahol a Lista2 egyenlõ az E elemnek Lista1-bõl való kiemelése eredményével. Érdemes most áttekinteni a teljes programot:

```
domains

    elem = symbol
    lista= elem*

predicates

    permutacio(lista,lista)
    kiejt(lista,elem,lista)
    rendezett(lista)
    rendperm(lista,lista)

clauses

    rendperm(Xs,Ys) if permutacio(Xs,Ys) and rendezett(Ys) and !.
    permutacio(Xs,[Z|Zs]) if kiejt(Xs,Z,Ys) and permutacio(Ys,Zs).
    permutacio([],[]).
    kiejt([E|Lista],E,Lista).
    kiejt([F|Lista1],E,[F|Lista2]) if kiejt(Lista1,E,Lista2).
    rendezett([_]).
    rendezett([X|[Y|Ys]]) if X<=Y and rendezett([Y|Ys]).
```

A permutációs rendezés sajnos nem túl szerencsés megoldás, mert a permutációk elõállítása csak 1/n!-al növeli a rendezettség elérésének valószínûségét lépésrõl lépésre. A permutációk sorában "egyszer csak" felbukkan az "igazi", és ennek ugyanakkora a valószínûsége, mint bármely másik permutációnak.

5.7.7 Rendezés minimumkiválasztással

A listák rendezésének egy másik módja, hogy az adott listából kiemeljük a legkisebbet és az eredménylista elejére tesszük, amelynek a "vége" már rendezett, ugyanilyen módon.

Deklaratívan megfogalmazva ez úgy hangzik, hogy egy lista akkor rendezett, ha a lista feje kisebb a rendezett részlistában szereplõ összes elemnél.
Ez utóbbi megközelítés ihlette a minimumkiválasztásos rendezés megvalósítását az alábbi módon:

```
domains

    elem = symbol
    lista = elem*

predicates

    minimum(lista,elem)
    kiejt(lista,elem,lista)
    rendmin(lista,lista)

clauses

    minimum([E],E).
    minimum([E|Lista],F) if minimum(Lista,F), F<=E and !.
    minimum([E|Lista],E) if minimum(Lista,F) and F>E.
    kiejt([E|Lista],E,Lista) if !.
    kiejt([F|Lista1],E,[F|Lista2]) if kiejt(Lista1,E,Lista2) and !.
    kiejt([],_,[]).
    rendmin([],[]).
    rendmin(Lista1,[E|Lista2]) if minimum(Lista1,E) and kiejt(Lista1,E,Lista3)
    and rendmin(Lista3,Lista2) and !.
```

A program trükkje az, hogy kiválasztjuk az adott Lista1 E minimumát a minimum szabályokkal, majd ezt az elemet kiejtjük a Lista1 eredeti listából, hogy egy olyan Lista2 végére fûzhessük, amit már lerendeztünk. A Lista3 olyan átmeneti lista, amely már nem tartalmazza az E elemet.

Az elsõ szabály szerint az üres lista eleve rendezett. Az említett kiejtés az elõzõ fejezetben említett kiejt eljárással történik.

5.7.8 Rendezés beszúrással

A rendezési feladatnak az eddiektõl eltérõ megoldása az lehet, hogy ha adott egy rendezett lista (és az üres lista mindig adott!), akkor a kívánt E elemet úgy illesztjük bele a rendezett listába, hogy azt szétválogatjuk E-nél kisebb-egyenlõ és nagyobb részekre, majd összerakjuk a három részlistát. Ennek megvalósítása a következõ program:

```
domains

    elem = symbol
    lista = elem*

predicates

    rendbesz(lista,lista)
    szetval(lista,elem,lista,lista)
    osszerak(lista,lista,lista)

clauses

    szetval([F|L1],E,[F|L2],L3) if F <= E and szetval(L1,E,L2,L3).
    szetval([F|L1],E,L2,[F|L3]) if F > E and szetval(L1,E,L2,L3).
    szetval([],_,[],[]).

    osszerak([E|L1],L2,[E|L3]) if osszerak(L1,L2,L3).
    osszerak([],Lista,Lista).
    rendbesz([E|L1],L2) if rendbesz(L1,L3) and szetval(L3,E,Lk,Ln)and
                    osszerak(Lk,[E|Ln],L2) and !.
    rendbesz([],[]).
```

Kezdjük az ismerkedést a **szetval** részprogrammal, amelynek az a feladata, hogy az elsõ paraméterként megadott [F|L1] lista F elemét elhelyezze vagy a kisebb-egyenlõk közé (l.szabály), vagy az F-nél nagyobbak közé (2.szabály), attól függõen, hogy melyik reláció teljesül. Mielõtt azonban elhelyezné, egy rekurzív hívásra kényszeríti önmagát, hogy az L2 és az L3 listák már rendelkezésre álljanak. Ha az elsõ paraméter az üres lista, akkor persze nincs mit tenni, két üres listát ad vissza eredményként.

Másik segédprogramocskánk az osszerak nevû, amely szintén ravasz módon dolgozik. Feladata az, hogy egyesítsen két listát. Az únió-képzéssel ellentétben most nem kell azt vizsgálnunk, hogy létezik-e egy elem már a listában, mert ez megengedett.

Ellenben ügyelni kell a sorrendre, amit úgy old meg, hogy sorozatos rekurzív hívásokkal "kiüríti" az elsõ argumentumban adott listát, majd érvénybe lép a második

5.7.9 Gyorsrendezés (quicksort)

Az egyik leghatékonyabb rendezési algoritmus a Quicksort. Elve az, hogy egy elemet kiválasztva, kettéosztjuk a listát nála kisebb és nála nagyobb elemekbõl álló részlistákra, majd ezeket rendezzük, végül újra egyesítjük a kisebbek listáját, a kiemelt elemet és a nagyobbak listáját.

Már az elve is rekurziót sejtet, így igazán hatékonyan csak a rekurziót megengedõ programozási nyelvekben valósulhat meg. Tulajdonképpen csak a fõ eljárás az új, azok a részprogramok, amiket használ, már ismerõsek a beszúrásos rendezésbõl. Maga a program az alábbi:

```
domains

    elem = symbol
    lista = elem*

predicates

    rendquick(lista,lista)
    szetval(lista,elem,lista,lista)
    osszerak(lista,lista,lista)

clauses

    szetval([F|L1],E,[F|L2],L3) if F <= E and szetval(L1,E,L2,L3).
    szetval([F|L1],E,L2,[F|L3]) if F > E and szetval(L1,E,L2,L3).
    szetval([],_,[],[]).
    osszerak([E|L1],L2,[E|L3]) if osszerak(L1,L2,L3).
    osszerak([],Lista,Lista).
    rendquick([E|L1],L2) if szetval(L1,E,Kis,Nagy) and
                            rendquick(Kis ,Kisrend) and
                            rendquick(Nagy,Nagyrend) and
                            osszerak(Kisrend,[E|Nagyrend],L2).
    rendquick([],[]).
```

Nézzük meg ugyanennek a feladatnak a megoldását Pascalban. A Pascalban megengedett a rekurzió, ellentétben a BASIC-el ahol erre nincs lehetőség. A programlista hossza és áttekinthetõsége magáért beszél.
Adottnak tételezzük fel az egészekkel feltöltött a[1..vege] tömböt.

```
procedure quick(also,felso:integer);
  procedure quick1(also,felso:integer);
    var
      bal,jobb,valaszto,id : integer;
    begin
      bal:=also; jobb:=felso;
      valaszto:=a[(also+felso) DIV 2];
      repeat
        while a[bal] < valaszto do bal:=bal+1;
        while a[jobb] > valaszto do jobb:=jobb-1;
        if bal<=jobb
          then
            begin
              id:=a[bal]; a[bal]:=a[jobb]; a[jobb]:=id; bal:=bal+1; jobb:=jobb-1
            end;
      until bal>jobb;
      if also < jobb then quick1(also,jobb);
      if bal < felso then quick1(bal,felso);
    end;
  begin
```

```
  quick1(1,vege)
end;
```

6.2 I8086 ÖSSZEFOGLALÓ

i8086 16 bites regiszterei	
- általános célú regiszterek ax (al,ah): akkumulátor bx (bl,bh): bázis regiszter cx (cl,ch): számláló regiszter dx (dl,dh): adat regiszter	- index regiszterek si : forrás index regiszter di : cél index regiszter
- stack kezelés regiszterei sp : stack pointer bp : bázis (frame) pointer	- szegmens regiszterek ds :adat szegmens regiszter cs :program szegmens regiszter ss :stack szegmens regiszter es :extra adat regiszter
- speciális regiszterek ip : utasitás számláló status: státusz és vezérlő regiszter	

Cimzési módok

Az i86 szegmentált memóriaképet valósit meg a moduláris programozás támogatására.

fizikai cim = szegmens * 16 + offset

Hosszú (FAR) cimzésnél a szegmens cimet explicite meg kell adni. Rövid (SHORT) cimzésnél a szegmens cim egy szegmens regiszter tartalma. A vonatkozó szegmens regisztert meg kell adni, vagy az értelemszerű alapértelmezést használhatjuk:

vonatkozó szegmensregiszterek	
adatműveletek	ds
stack és BP relativ műveletek	ss
ugrások, szubrutin hivások	cs
blokk cél operandusok	es

```
Pl.:
sub   cx, kivon       ;ds
and   cx, es:valt     ;es
stosb                 ;es
push  ax              ;ss
mov   dx, [bp-8]      ;ss
inc   cs:valt1        ;cs
call  rutin           ;cs
```

Az offset kialakitásához használható cimzési módok és példák	
-memória direkt:	mov BYTE PTR cim, 1
-regiszter direkt:	xor ax, ax
-regiszter indirekt:	cmp [bx], 2 (csak si,di,bx és bp)
-bázis relativ:	inc [bp-6] (csak si,di,bx és bp)
-indexelt:	and tomb[si](csak si,di,bx és bp)
-bázisrelativ indexelt:	dec [bx+si+3](csak si,di,bx és bp)

Utasitás készlet (real mód)

Az utasítás alakja:
cimke mnemonik cél-, forrásoperandus ; művelet

pl: add ax, bx ----------> ax = ax + bx

ahol forrásoperandus lehet konstans, pl. sub dx, 1

Adatmozgató utasítások	
MOV op1, op2	op1 = op2
XCHG op1, op2	op1 és op2 felcserélése
LEA reg, cim	reg = cim;
LDS reg, cim	reg = *(cim) ; ds = *(cim+2)
LES reg, cim	reg = *(cim); es = *(cim+2)
LAHF	ah = flag-ek
SAHF	flag-ek = ah
XLAT	al = *(bx + al)

Stack kezelő utasitások	
PUSH op	SP -= 2; stack[SP] = op;
POP op	op = stack[SP]; SP += 2;
PUSHA, POPA	általános, index és stack kezelő regiszterek mentése és visszaállitása
PUSHF, POPF	flag-ek mentése visszaállitása

I/O műveletek:	
IN al, portcim	portcim-ről beolvasás
IN al, dx	*(dx)-ről beolvasása
OUT portcim, al	portcim-re kiirás
OUT dx, al	*(dx)-re kiirás

Aritmetikai utasitások: carry, auxiliary-carry, parity, zero, sign, overflow bitek állítása	
ADD op1, op2	op1 = op1 + op2
ADC op1, op2	op1 = op1 + op2 + carry
SUB op1, op2	op1 = op1 - op2;
SBB op1, op2	op1 = op1 - op2 - carry;
INC op	op+1;
DEC op	op-1;
NEG op	op = 2-s komplemens(op);
NEG op	op = 1-s komplemens(op);
CMP op1,op2	flag-ek = (op1 - op2) tulajdonságai
CBW	ax = előjelkiterjesztés(al)
CWD	dx:ax = előjelkiterjesztés(ax)
Előjel nélküli számokra:	
MUL byteop	ax = al * byteop
MUL wordop	(dx,ax) = ax * wordop
DIV byteop	al = ax / byteop; ah = ax % byteop;
DIV wordop	ax = (dx,ax)/byteop; dx = (dx,ax)%byteop;
Előjeles számokra:	
IMUL byteop	ax = al * byteop
IMUL wordop	(dx,ax) = ax * wordop
IDIV byteop	al = ax / byteop; ah = ax % byteop;
IDIV wordop	ax = (dx,ax)/byteop; dx = (dx,ax)%byteop;

Logikai műveletek zero, carry, overflow flagek állítása	
NOT op	op = ~op;
AND op1, op2	op1 = op1 & op2;
OR op1, op2	op1 = op1 | op2;
XOR op1, op2	op1 = op1 ^ op2;
TEST op1, op2	flag-ek mint AND op1, op2
SHL op1,op2	op1 <<= op2; (előjel nélkül)

SHR op1,op2	op1 >>= op2; (előjel nélkül)
SAL op1,op2	op1 <<= op2; (előjel marad)
SAR op1,op2	op1 >>= op2; (előjel marad)

Ciklikus léptetések: ROL, RCL, ROR, RCR

BCD aritmetika: AAA, AAD, AAM, AAS, DAA, DAS

Blokk (string) kezelő utasitások:	
MOVSB	(es:di)byte = (ds:si)byte; si+= dsi; di+= ddi;
MOVSW	(es:di)word = (ds:si)word; si+= dsi; di+= ddi;
CMPSB	CMP (es:di)byte, (ds:si)byte; si+= dsi; di+= ddi;
CMPSW	CMP (es:di)word, (ds:si)word; si+= dsi; di+= ddi;
SCASB	CMP al, (es:di)byte; si+= dsi; di+= ddi;
SCASW	CMP ax, (es:di)word; si+= dsi; di+= ddi;
LODSB	MOV al, (es:di)byte; si+= dsi; di+= ddi;
LODSW	MOV ax, (es:di)word; si+= dsi; di+= ddi;
STOSB	MOV (es:di)byte, al; si+= dsi; di+= ddi;
STOSW	MOV (es:di)word, ax; si+= dsi; di+= ddi;
INSB	IN (es:di)byte, dx; di+= ddi;
INSW	IN (es:di)word, dx; di+= ddi;
OUTSB	OUT dx, (ds:si)byte, dx; si+= dsi;
OUTSW	OUT dx, (ds:si)word, dx; si+= dsi;

Vezérlés átadó utasitások:	
JMP cim	feltétel nélküli ugrás
JZ, JNZ	Logikai művelet után
JE, JNE, JL, JG, JLE, JGE	Előjeles aritmetikai művelet után
JE , JNE, JB, JA, JBE, JAE	Előjel nélküli műveletek után
JCXZ, JNO, JS, JNS, JP, JPO	Speciális

Feltételes ugrást csak 128 byte-tal nem messzebbi cimkére lehet kiadni.

Szubrutinhivás:	
CALL cimke	cimkén kezdődő rutin meghivása
CALL op	indirekt szubrutinhivás
RET	szubrutinból visszatérés
INT vector	software interrupt
IRET	visszatérés software és hardware interrupt rutinból

Processzorvezérlő utasítások	
CLC-törlés, STC-beállitás, CMC-komplementálás	Carry flag:
CLD-törlés, STD-beállitás	Direction flag:
CLI-tiltás, STI-engedélyezés	Interrupt Engedélyezés:
HLT	Processzor leállitás:
NOP	üres operáció:

MACRO-ASSEMBLY PROGRAMOK FELÉPÍTÉSE

Utasitás szerkezet: **[cimke:] [mnemonik [operandusok]] [;megjegyzés]**

Modulszerkezet kialakitásához használható direktivák:

SEGMENT - szegmens definició kezdete
    szegmensnév SEGMENT [igazitás][kombináció]['osztálynév']

```
                        BYTE      PUBLIC
                        WORD      STACK
                        PARA      COMMON
                        PAGE      MEMORY
                                  AT cim

ENDS    - szegmens definició vége
    szegmensnév ENDS

END     - forrás file vége
    END [program belépési pont]

GROUP   - szegmens csoportok kialakitása
    csoportnév GROUP szegmensnevek felsorolása vesszővel elválasztva

ASSUME  - szegmens regiszterek töltésekor a cimek referenciájának
megadása
    ASSUME szegmens-regiszter: szegmensnév

ORG     - szegmens kezdőcimének a rögzitése
    ORG kezdőcim

PROC    - szubrutin definiálás kezdete
    szubrutin-név PROC [tipus]
                       NEAR
                       FAR

ENDP    - szubrutin definiálás vége
    szubrutin-név ENDP
```

Utasitáskészletet beállitó direktivák:	
.8086	- alap utasitáskészlet
.8087	- +lebegőpontos coprocesszor utasitások
.80286	- 286 utasitások
.80286p	- 286 protected módú utasitások

Adatok definiálása:

```
Adatok számára helyfoglalás a memóriában:
  [azonositó]   tipus           kezdőértékek
                DB  ;byte       felsorolás
                DW  ;szó        szám DUP(kezdőérték)
                DD  ;duplaszó   ?
;inicializálatlan
```

Igy annyi memóriát foglalunk le az adatok számára, amennyi a megadott számű kezdőérték és a méret szerint szükséges.

Constansdeklarációk:

```
    azonositó   EQU érték
```

Assembler műveletei:

```
Objektumméret explicit megadása
        tipus   PTR kifejezés
        BYTE
        WORD
        DWORD
        FAR
        NEAR
```

```
        Pl.:    mov ax, WORD PTR adat
        call FAR PTR rutin
```

Cim szegmens és offset értékének a meghatározása:

```
    SEG     adat
    OFFSET  adat
Pl.:
    mov ax, SEG adat
    mov ds, ax
    mov bx, OFFSET adat ; ds:bx -> adat
```

Több file-ból álló programok:

Szimbólumok exportálása(PUBLIC) és importálása(EXTRN):

```
    PUBLIC név          ; ebben a file-ban definiált név más file-ból
                        ; is elérhető
    EXTRN  név: elérés  ; más file-ban definiált nevet innen is
használjuk
                NEAR    ; a név ugyanebben a szegmensmensben van
                FAR     ; a név más szegmensben van.
Pl.
FILE1:                                  FILE2:
=============                           ===============
EXTRN dat: FAR                          EXTRN rutin: FAR

PUBLIC rutin                            dat DW 0
rutin  PROC NEAR                        PUBLIC dat
       ...                              ...
rutin  ENDP                             call rutin
       ...
       mov  ax, WORD PTR dat
       ...
```

I8086 MEGSZAKITÁS RENDSZERE

Az i8086 vektoros megszakitási rendszerrel rendelkezik, amely az IT rutin cimét a hw IT kiszolgálása alatt a kérő egység (illetve az i8259 IT vezérlő) által beadott vektor alapján, illetve sw. IT alatt az int utasitás paramétereként átadott vektor határozza meg:

```
        IT rutin offset   = vector * 4
        IT rutin szegmens = vector * 4 + 2
```

A lehetséges vektorok 0..255 tartományban lehetnek, igy a fizikai memória első 1kb tartományát a IT rutinok cimeivel kell feltölteni. Egy IT-re rákapcsolódás esetén az IT tábla megfelelő elemét kell átirni, és az i8259 megszakitásvezérlő egységben az IT-t engedélyezni (ha idáig nem volt). Ehhez az i8259 mask (21h I/O cim) regiszterének a IRQ számnak megfelelő bitjét kell 0-ba állitani.

IBM AT IT KIOSZTÁSA

Vektor	i8259 #IT	Processzor exception-ok
00	-	0-val osztás exception
01	-	Lépésenkénti utasitás végrehajtás excetion
02	-	NMI - nem maszkolható IT
03	-	Töréspont (breakpoint)
04	-	Tulcsordulás (overflow)

Hardware IT-k

```
08      IRQ0            Timer IT
09      IRQ1            Klaviatúra IT
0A      IRQ2            Slave IT kontroller
0B      IRQ3            COM2 IT
0C      IRQ4            COM1 IT
0D      IRQ5            Winchester controller IT
0E      IRQ6            Floppy controller IT
0F      IRQ7            Nyomtató IT
70      IRQ8            Real-time óra IT
71      IRQ9            LAN adapter IT
75      IRQ13           Matemetikai koprocesszor IT
76      IRQ14           Második Winchester controller IT
```

Pl. a COM1 IT (IRQ4) engedélyezése:

```
        I8259_BAS   EQU 20h             ;i8259 bázis cim

        cli
        in      al, I8259_BAS+1
        and     al, 0efh            ;reset 4. bit
        out     I8259_BAS+1, al
        sti
```

Pl. a COM1 IT (IRQ4) tiltása:

```
        cli
        in      al, I8259_BAS+1
        or      al, 10h            ;set 4. bit
        out     I8259_BAS+1, al
        sti
```

Az IT_NUM vektorral jelzett megszakitásra saját FUNCTION függvényt a következő MACRO-val kapcsolhatunk:

```
SET_IT MACRO    IT_NUM, FUNCTION
    push    ax          ; regiszter ment
    push    bx
    push    es
    mov     ax, 0       ; IT tábla megfelelő bejegyzésének cime es:bx
    mov     es, ax
    mov     bx, 4 * (IT_NUM)
    cli                 ; IT tilt
    mov     ax, OFFSET FUNCTION ;tábla kitöltés
    mov     WORD PTR es:[bx], ax
    mov     ax, SEG FUNCTION
    mov     WORD PTR es:[bx+2], ax
    sti                 ; IT engedélyezés
    pop     es          ; regiszter vissza
    pop     bx
    pop     ax
ENDM
```

Az IT tábla egy elemének mentése és visszaállitása, a code szegmensben
lefoglalt mentési terület feltételezésével:

```
SAVE_IT MACRO    IT_NUM, SAVE_AREA
    push    ax          ; regiszter ment
    push    bx
    push    es
    mov     ax, 0       ; IT tábla megfelelő bejegyzésének cime es:bx
    mov     es, ax
    mov     bx, 4 * (IT_NUM)
```

```
    mov     ax, WORD PTR es:[bx]
    mov     WORD PTR cs:SAVE_AREA, ax
    mov     ax, WORD PTR es:[bx+2]
    mov     WORD PTR cs:SAVE_AREA[2], ax
    pop     es          ; regiszter vissza
    pop     bx
    pop     ax
ENDM

RESTORE_IT MACRO    IT_NUM, SAVE_AREA
    push    ax          ; regiszter ment
    push    bx
    push    es
    mov     ax, 0       ; IT tábla megfelelő bejegyzésének cime es:bx
    mov     es, ax
    mov     bx, 4 * (IT_NUM)
    cli                 ; IT tilt
    mov     ax, WORD PTR cs:SAVE_AREA, ax
    mov     WORD PTR es:[bx], ax
    mov     ax, WORD PTR cs:SAVE_AREA[2], ax
    mov     WORD PTR es:[bx+2], ax
    sti                 ; IT engedélyezés
    pop     es          ; regiszter vissza
    pop     bx
    pop     ax
ENDM
```

IT rutin szerkezete teljesen átvett IT esetében

```
      I8259_BAS   EQU 20h             ;i8259 bázis cim
      EOI         EQU 20h             ;i8259 End-of-IT karakter

ITrutin:
     pusha                      ; regiszterek mentése
     push   ds                  ; szegmens regiszterek mentése
     . . .
     mov    ax, adatszegmens    ; ha használunk globális adatokat
     mov    ds, ax
     . . .
     mov    al, EOI             ;i8259 felé az IT vég jelzése
     out    I8259_BAS, al
     pop    ds                  ;regiszterek visszaállitása
     popa
     iret                       ;IT-ből visszatérés
```

IT rutin szerkezete felfűzött IT esetében
Feltételezzük, hogy az elmentett korábbi IT rutin cim a code szegmensben lévő oldIT cimen került elmentésre.

```
ITrutin:
     pusha                      ; regiszterek mentése
     push   ds                  ; szegmens regiszterek mentése
     . . .
     mov    ax, adatszegmens    ; ha használunk globális adatokat
     mov    ds, ax
     . . .
     pop    ds                  ;regiszterek visszaállitása
     popa
     jmp    DWORD PTR cs:oldIT  ;régi IT-re ugrás
```

Pl. a COM1 IT (IRQ4) teljes átvétele:

```
        I8259_BAS   EQU 20h             ;i8259 bázis cim
```

```
        COM1_IT     EQU 0ch             ;COM1 IT vektor

com1it               DD      ?          ;IT vector save area
        ...
        SAVE_IT COM1_IT, com1it
        SET_IT  COM1_IT, ITrutin
        cli
        in      al, I8259_BAS+1
        and     al, 0efh            ;reset 4. bit
        out     I8259_BAS+1, al
        sti
```

DOS HIVÁSOK

A DOS operációs rendszer szolgáltatásai a három software interrupton keresztül hivhatók:

```
    int 21h - általános DOS szolgáltatások
    int 25h - lemez szektor közvetlen olvasása
    int 26h - lemezszektor közvetlen irása
```

Az int 21h-n megvalósitott fontosabb funkciók:
A funkció kódot a DOS az ah regiszterben várja.

```
getche - Klaviatúra beolvasás echo-val
    Vár amig a kezelő leüt egy billentyűt a klaviatúrán, azt echo-za,
    és az ASCII kódot visszaadja.
    be: ah = 1
    ki: al = ASCII kód, ha normál billentyű
        először 0, majd kód, ha kiterjesztett billentykód
getch - Klaviatúra beolvasás echo nélkül
    Vár amig a kezelő leüt egy billentyűt a klaviatúrán,
    és az ASCII kódot visszaadja.
    be: ah = 8
    ki: al = ASCII kód, ha normál billentyű
        először 0, majd kód, ha kiterjesztett billentykód
putch - képernyőre karakter kivitel
    Egy karaktert a standard outputra kivisz,
    be: ah = 2, dl = kiviendő karakter kód
    ki: -
puts - képernyőre string kivitel
    Egy stringet $ végjelig a standard outputra kivisz,
    be: ah = 9, ds:dx = kiviendő string cime
    ki: -
exit - visszatérét a DOS-hoz
    Kilép az adott programból.
    be: ah = 4ch, al = exit kód
    ki: -
```

DOS FILE KEZELÉS

Create - file létrehozása: adott névvel és attribútummal létrehoz egy file-t, illetve a már létezőt 0 hosszúra csonkitja, és megnyitja a file-t update műveletre. A file pointer a file elejére mutat.

```
    be: ah = 3ch, ds:dx = path név cime (string 0 végjellel),
        cx = file attribútum
    ki: Ha sikerült: carry = 0, ax = file handle
        Hiba esetén: carry = 1, ax = hiba kód
```

Open - file megnyitása
Már létező file-t a megadott műveletre megnyit. A file pointer a file elejére mutat.

```
    be: ah = 3dh, ds:dx = path név cime (string 0 végjellel),
        al = művelet (0-olvasás, 1-irás, 2-update)
    ki: Ha sikerült: carry = 0, ax = file handle
        Hiba esetén: carry = 1, ax = hiba kód

Close - file lezárása
Megnyitott file-t lezár.
    be: ah = 3eh, bx = file handle
    ki: Ha sikerült: carry = 0
        Hiba esetén: carry = 1, ax = hiba kód

Read - file-ból olvasás
A file handle által megadott file-ból a file pointer által mutatott
helyről a bufferbe adott számú byte-t olvas. A file pointer
továbbviszi
    és visszaadja a ténylegesen beolvasott byte-ok számát.
    be: ah = 3fh, bx = file handle, ds:dx = buffer cim, cx = byte
szám
    ki: Ha sikerült: carry = 0, ax = olvasott byte szám
        Hiba esetén: carry = 1, ax = hiba kód

Write - file-ból olvasás
A file handle által megadott file-ba a file pointer által mutatott
helyre a bufferből adott számú byte-t ir. A file pointer-t
továbbviszi és visszaadja a ténylegesen kiirt byte-ok számát.
    be: ah = 40h, bx = file handle, ds:dx = buffer cim, cx = byte
szám
    ki: Ha sikerült: carry = 0, ax = olvasott byte szám
        Hiba esetén: carry = 1, ax = hiba kód

Seek - file pointer mozgatása
A file pointernek új értéket ad, referencia + 32 bites eltolás
formában.
    be: ah = 42h, bx = file handle, cx:dx = eltolás,
        al = referencia (0-file eleje, 1-aktuális file pointer, 2-
file vége)
    ki: Ha sikerült: carry = 0, dx:ax = új file pointer
        Hiba esetén: carry = 1, ax = hiba kód

A lehetséges hibaüzenetek:
1- érvénytelen seek referencia tipus
2- nincs ilyen nevű file
3- nincs ilyeb path
4- túl sok file van nyitva egyidejűleg
5- tiltott hozzáférés
6- érvénytelen file handle
12- érvénytelen művelet

Lemezszektor közvetlen olvasása int25h DOS hivással
be: al = meghajtó szám, cx = egyszerre olvasandó szektorok száma
    ds = elsőnek olvasandó szektor száma, ds:bx = buffer cim
Lemezszektor közvetlen irása int26h DOS hivással
be: al = meghajtó szám, cx = egyszerre irandó szektorok száma
    ds = elsőnek irandó szektor száma, ds:bx = buffer cim

Képernyő üzemmód kezelő BIOS hivás - int10h
üzemmód állitás
    be: ah = 0,
        al = 0  - 40x25 fekete-fehér karakteres
        al = 1  - 40x25 szines karakteres
        al = 2  - 80x25 fekete-fehér karakteres
```

```
        al = 3  - 80x25 szines karakteres
        al = 4  - 320x200 pixeles szines grafikus
        al = 5  - 320x200 pixeles fekete-fehér grafikus
        al = 6  - 640x200 pixeles fekete-fehér grafikus

Egy karakter kivitele
    be: ah = 14
        al = kiirandó karakter
        bl = karakter szine (grafikus módban)
        bh = szövegmód

Klavitúra kezelés BIOS int16h hivással:
getch - Klaviatúra beolvasás echo nélkül
    Vár amig a kezelő leüt egy billentyűt a klaviatúrán,
    és az ASCII kódot visszaadja.
    be: ah = 0
    ki: al = ASCII kód, ha normál billentyű
             0, ha kiterjesztett billentykód
        ah = scan kód
kbhit - klaviatúrán billentyű leütés ellenőrzése
    Ellenőrzi, hogy a kezelő lenyomott-e billentyűt
    be: ah = 1
    ki: zero flag = 1, ha van várakozó karakter
                  = 0, ha nincs várakozó karakter

;*******************************************
; VII. Mintaprogram
;*******************************************
NAME    proba       ;program név
.286                ;utasitás készlet direktiva
;*-------------------------------------------
;* KONSTANSOK
;*-------------------------------------------
TERMINATE   EQU     04ch
CR          EQU      0dh
;*-------------------------------------------
;* MACROK
;*-------------------------------------------
CALLDOS MACRO   FUNCTION
    mov ah, FUNCTION
    int 21h
ENDM
;*-------------------------------------------
;* ADAT SZEGMENS
;*-------------------------------------------
data SEGMENT WORD PUBLIC 'DATA'
ASSUME ds:data
valt    DW  0               ; "valt" 2 byte-s adat 0 kezdőértékkel
tomb    DB  10 DUP(' ')     ; 10 byte-s "tomb" SPACE kezdőértékkel
szoveg  DB  'Hello'         ; ASCII string
        DB  13, 0ah, '$'    ; CRLF + lezáró $ az int21h/9 -nek
data ENDS
;*-------------------------------------------
;* STACK SZEGMENS
;*-------------------------------------------
stack SEGMENT WORD STACK 'STACK'
ASSUME ss:stack
DW  2048    DUP(?)          ;2048 byte-os inicializálatlan stack
terület
stack ENDS
;*-------------------------------------------
```

```
;* KÓD SZEGMENS
;*-----------------------------------------
code SEGMENT WORD PUBLIC 'CODE'
ASSUME cs:code
;*-----------------------------------------
;* Szubrutin
;*-----------------------------------------
hello   PROC NEAR
    push    dx          ;regiszter mentés
    push    ds
    lds     dx, szoveg  ;paraméter átadás
    mov     ah, 09h     ;DOS print string ($-ig)
    pop     ds          ;regiszter visszaállitás
    pop     dx
    ret                 ;visszatérés
hello ENDP
;*-----------------------------------------
;* Főprogram
;*-----------------------------------------
foprog:
    mov ax, data            ;adatszegmens regiszter beállitása
    mov ds, ax
    mov WORD PTR valt, 1    ;word változó elérése
    mov si, 0               ;ciklus egy tomb CR-kkel feltöltésére
CIC:mov BYTE PTR tomb[si],CR
    inc si
    cmp si,10
    jl  CIC
    call hello              ;függvény hivás
    CALLDOS TERMINATE       ;macro hivás
code ENDS                   ;code szegmens vége
END  foprog                 ;program belépési pont
```

PROGRAMBETÖLTÉS

A DOS a program inditása során a következő feladatokat végzi el:

1. 256 byte-os program szegmens prefix (PSP) létrehozása a memóriában:

    A PSP szerkezete:

offset	tartalom
0	int 20h
2	memória limit
5	call far
0ah	program befejezési cime
0eh	Ctrl break rutin cime
12h	Hibakezelő rutin cime
2ch	environment cime
50h	int 21h ret
6ch	file leiró blokk
80	parancssor argumentumok száma ill. diszk buffer
81-0ffh	parancssor parancsot követő része ill. diszk buffer

2. A program adat és stack szegmensek elhelyezése a memóriában a PSP után relokáció elvégzése
3. cs = ds = es = &PSP
4. ss = stack szegmens, sp = stack szegmens utolsó szava
5. jmp program belépési pont

Mivel az inditáskor a ds nem a program adatszegmensére mutat, a programokat általában a következő sorral kell kezdeni:

```
            MOV     ax, adatszegmens
```

```
        MOV     ds, ax
```

6.0 MELLÉKLET

6.1 ASCII KÓDTÁBLÁZAT

4.2 táblázat: az ASCII kódtáblázat

	000	001	010	011	100	101	110	111
0000	NUL	DLE	blank	0	@	P	’	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	“	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EDT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	‘	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

7. IRODALOM

-: Dictionary of Computing, Oxford University Press, Oxford
Agárdi Gábor: IBM PC gyakorlati Assembly, LSI, Budapest
Agárdi Gábor, Ágoston György: Assembly, LSI, Budapest
Angszter Erzsébet: Programozás tankönyv I és II, GDF, Budapest
Aszalós János, Erki Irén: Bevezetés a strukturált programozásba, Számok, Budapest
Cserny László: Mikroszámítógépek, LSI, Budapest
Dujmović Jozo: Programski jezici i metode programiranja, Naučna Knjiga, Sarajevo
Kovačević Vladimir: Programska podrška računarskih sistema