

MÉRNÖKI TOVÁBBKÉPZŐ INTÉZET

előadássorozatából: 4679

Dr. Pálfi Gyuláné – Pázmány Béla

**Automatikus programozási nyelv
EMG 830 Autokód**

KÉZIRAT

BUDAPEST

1969

Ára: 33,- Ft

7385

A.2834

52284/313

MÉRNÖKI TOVÁBBKÉPZŐ INTÉZET

előadássorozatából: 4679

Dr. Pálfi Gyuláné – Pázmány Béla

Automatikus programozási nyelv EMG 830 Autokód

01-40-0821

KÉZIRAT

Változatlan utánnymás

BUDAPEST

1969

Kéziratot ellenőrizte:

Rinágel József

1997 MARC 10.

1990 -10- 10

1986 -04- 10

1995 OKT 17.

1970 SEP 20

1972 OKT 2

1980 OKT 40

1974 OKT 14

1983 OKT 17

1976 OKT 7

A kiadásért felelős: Talyigás Ferenc

Megrendelve: 1969. szeptember. Megjelent: 1969. október. Példányszám: 517

Készült: Rotaprint lemezzel (kicsinyítéssel), a MSZ 5601-59

és MSZ 5602-55 szabványok szerint, 15,- (A/5) iv terjedelemben, 8 ábrával

69-1864 - FELSŐOKTATÁSI JEGYZETELLÁTÓ VÁLLALAT, BUDAPEST

1 1978 OKT 7

TARTALOMJEGYZÉK

1. Bevezetés	5
2. Az univerzális, automatikus vezérlésű tárolt programu, digitális elektronikus számítógépek rövid ismertetése	6
3. Számítógép fő részei	10
4. A számábrázolás a program- vezérelt tárolt programu, digitális elektronikus számítógépeken	17
5. Logikai függvények - logikai utasítások	27
6. Gépi programozási nyelvek - automatikus programozási nyelvek	34
7. Az EMG 830 autókód utasítás rendszere	44
8. Az EMG 830 autókód utasításrendszerének részletezése ..	47
9. Hibajelzések	72
10. Feladatok	81

1. BEVEZETÉS

A jegyzet az EMG 830-as elektronikus számítógépen autokód programozási nyelven programozóknak kíván segítséget nyújtani. Mivel a számítógépen dolgozók között lesznek nagy gyakorlattal rendelkező programozók és olyanok is, akik a programozást ezen a számítógépen autokódban tanulják meg, ezért a jegyzet úgy készült, hogy a különböző felkészültségi fokon álló programozók egyaránt használni tudják.

Kezdő programozók számára készült a jegyzet 2. / az univerzális automatikus vezérlésű, tárolt programu, digitális elektronikus számítógépek rövid ismertetése/ a 3. / a számítógép fő részei /, a 4. / a számábrázolás a programvezérelt, tárolt programu, digitális elektronikus számítógépeken / és a 6. / gépi programozási nyelvek- automatikus programozási nyelvek / fejezete. E fejezetek átolvasása után a kezdő programozóknak a 10. fejezet / feladatok / részletes áttanulmányozását ajánljuk. A példákat úgy állítottuk össze, hogy vegyesen megtalálhatók benne az ügyviteli és matematikai jellegű feladatok. Érdekes a példákat a fenti csoportosítástól függetlenül átnézni, mivel a példákban nem a kitűzött feladatok programozása a lényeges, hanem azok a programozási fogások, melyeket e példákban bemutatunk. Ugyanez a szempont indokolja egy-egy feladat többféle megoldását is. A programozási fogások miatt gyakorlott programozók számára is célszerű e példák átnézése, mivel adott esetben, ha emlékeznek arra, hogy a példákban található valamelyik módszer számukra felhasználható, úgy könnyen megkereshetik a jegyzetben.

Az 5. fejezet / logikai függvények-logikai utasítások / az EMG 830 autokód logikai utasítása miatt kerültek be a jegyzetbe. A logikai utasítások alkalmazására néhány példát a 10. fejezetben / feladatok / mutatunk be.

A 7. / az EMG 830-as autokód utasításrendszere / a 8. / az EMG 830 autokód utasításrendszerének részletezése / és a 9. fejezet / hibajelzések / programírás közben és programfuttatás közben használható.

A 7. fejezet röviden összefoglalja az EMG 830-as autokód utasításrendszerét, megmutatja, hogy milyen utasításokat lehet használni, a 8. fejezet részletezi az utasításokat, megmutatja, hogy az utasításokat hogyan lehet használni, a 9. fejezet pedig a fordításnál, illetve futtatásnál jelentkező hibákat elemzi, feltüntetve, hogy az egyes hibaszámok milyen esetekben jelentkezhetnek.

Reméljük, hogy úgy a jegyzet, mint az EMG 830-as autokód programozási nyelv eléri célját, és segítségére lesz a programozóknak.

A szerzők.

2. AZ UNIVERZÁLIS, AUTOMATIKUS VEZÉRLÉSŰ, TÁROLT PROGRAMU, DIGITÁLIS, ELEKTRONIKUS SZÁMÍTÓGÉPEK RÖVID ISMERTETÉSE

Évtizedek óta széles körben elterjedtek a mechanikus asztali számológépek, amelyeket különösen műszaki - tudományos, valamint bizonyos pénzgazdálkodási számításoknál igen jó eredménnyel használnak ma is. Viszonylag nagyobb adattömeggel dolgozó, adatfeldolgozási jellegű számításoknál hasznos munkát lehet végezni lyukkártyás "hollerit" (Hollerith) rendszerű adatfeldolgozó gépekkel. Mindkét matematikai géptípusnak ma is megvan a jól bevált alkalmazási területe, de egyre több lesz az olyan probléma, amelyet ilyen gépekkel meg sem lehet közelíteni, egyrészt a számításokban szereplő adatok mennyisége, másrészt az elvégzendő numerikus számítások tömege miatt.

Mivel ezen géptípusok eredeti szerkezeti elveinek megtartása mellett lényeges továbbfejlesztésre már nem volt lehetőség, új elvi alapokon, a modern elektronikában használatos elemek alkalmazásával viszonylag igen gyorsan, nem egészen egy évtized alatt kialakult az az új matematikai géptípus, amely néhány év alatt világszerte elterjedt, és - ma már látjuk - felhasználásában korlátlan lehetőségek rejlenek.

Vegyük sorra a modern elektronikus számítógép a jelen fejezet címében felsorolt tulajdonságait. Itt említjük meg, hogy az ezen tulajdonságokkal rendelkező géptípust hazánkban számológépnek is nevezik. A magunk részéről a továbbiakban a "számítógép" megjelölést fogjuk használni, vagy a rövidség kedvéért esetleg csak "a gép" kifejezést használjuk.

A géptípus egyik legfőbb jellemzője, hogy digitális. Ez azt jelenti, hogy a számításokban résztvevő mennyiségeket számjegyekből álló számok formájában adják meg a gép részére, és a számítógép a számjegyeken végzett műveletek - amelyek bármilyen bonyolultak is, nem mások, mint alapl műveletek véges sorozatai - elvégzése után ugyancsak számok formájában szolgáltatja az eredményt. Ismeretes ezzel szemben a digitális számítógépektől lényegben eltérő géptípus, az ún. analóg gépek típusa, amelyeknél az adatok és az eredmények folytonos változású fizikai mennyiségek formájában jelentkeznek, és a számítást fizikai törvényszerűségek hatása által úgy oldják meg, hogy egy bizonyos adott számítási feladat fizikai analogonját valósítják meg. Analóg gépek programozásával a jegyzet nem foglalkozik.

Az elektronikus jelző arra utal, hogy a régebbi mechanikus, illetve elektro-mechanikus digitális számítógépekkel szemben ennél a géptípusnál elektroncsöveket, illetve ujabban kristálydiódákat, tranzisztorokat, integrált áramköröket, mágneses magokat alkalmaznak. Az elektronikus számítógép belsejében, egyes kivételektől eltekintve, nem mozgó alkatrészek végzik a számításo-

kat. (A kiinduló adatok bevitelénél és az eredmények kiadásánál egyelőre elkerülhetetlen a mechanikus, mozgó berendezések alkalmazása.) Az elektronikus gépek több nagyságrenddel nagyobb műveleti sebessége a nem elektronikus gépekkel szemben volt az a körülmény, amely a vezérlés új elvi alapokon nyugvó megszervezését vonta maga után.

Az univerzális tulajdonság azt jelenti, hogy a gép elvileg bármilyen számítási feladat elvégzésére képes. Kimutatták, hogy ilyen számítógépen minden olyan probléma megoldható, amely elvileg megfogalmazható. A matematikai gép további módosításai már csak arra valók, hogy a működést gyorsítsák, vagy a szerkezetet egyszerűsítsék.

Ebben az értelemben az asztali számológépek is univerzálisak, mert ha a bonyolult matematikai számításokra (pl. differenciál - egyenletek vagy - rendszerek, transzcendens egyenletek stb.) ismerünk célravezető, jól közelítő numerikus módszereket, akkor a számítás végső soron aritmetikai alpműveletek elvégzésére vezethető vissza, amelyeket akár asztali számológéppel is elvégezhetünk.

Hangsúlyozzuk, hogy tulajdonképpen nem a gép, hanem a segítségével végzett feladatok és annak programozói univerzálisak: a mai elektronikus számológép is önmagában csak az aritmetikai alpműveletek és néhány logikai művelet elvégzésére képes, de erre is csak abban az esetben, ha ilyen értelmű utasítást kapott. Mivel azonban teljesen egyértelműen, pontosan megadott utasítássorozat (program) alapján képes e programnak megfelelő, elvileg tetszőlegesen hosszú számítási eljárás véghezvitelére, mégis helytálló - ilyen értelemben! - az univerzális elnevezés.

Az automatikus vezérlésű számítógépek (használatos a "programvezérelt" elnevezés is) az elvégzendő számítási lépések ismeretében a továbbiakban önállóan üzemelnek, azaz üzem (számolás) közben emberi beavatkozást nem igényelnek. A számítás menetének ismeretén azt értjük, hogy a számítógéppel valamilyen, a gép részére érthető nyelven közöljük a végrehajtandó lépések teljes sorozatát, az adatbevitel módjától, sorrendjétől, a számítási lépéseken keresztül, végig, egészen az eredmények közlésére vonatkozó részletekig, mégpedig minden lépést általában az aritmetikai műveletek szintjéig lebontva, részletezve. A számítógép - a népszerű "gondolkodó gép" elnevezéssel ellentétben - nem gondolkozik, hiányos utasítássorozat esetén a hiányt nem észleli, hibás utasítást hibásan hajt végre.^{1/}

1. Az utasítássorozatnak tehát teljesnek és hibátlanak kell lennie. Arról, hogy az utasítássorozatot milyen alkalmas formában adjuk meg a számítógépnek és melyek a program megszerkesztésének előkészítő lépései, a továbbiakban még szó lesz.

^{1/} Ez nem áll ellentétben a különböző automatikus programnyelvek ún. hibajelzéseivel, ugyanis e hibajelzéseket is csak azért adja ki a gép, mert valamilyen program utasításai (általában a fordítóprogram) ellenőrző vizsgálatokat hajtatnak végre és hiba esetén jelzések kibocsátására késztetik.

Az adatok bevitele már magához a számoláshoz tartozik és csak akkor kerül sor rá, amikor a program végrehajtása során a gép adatbeviteli utasítást észlel. Az automatikus vezérlésű gép tehát automatikusan elvégzi az elemi számítások tetszőlegesen hosszú, szükséges sorozatát, de természetesen mód van arra is, hogy a gép által számolt és kiadott részleteredmények megvizsgálása céljából a számolás átmenetileg megszakítható legyen, és szükség esetén a programozó beavatkozhatson.

Hasonlítsuk össze az automatikus gépek működését a közönséges asztali számológépekével: az utóbbiakon mindenegyes aritmetikai műveletet a kezelőnek kell beállítania, gombok, kapcsolók segítségével. A művelet elvégzése után a kezelőnek kell gondoskodnia arról, hogy az eredmény "megmaradjon", tehát le kell azt írnia egy papírosra. Nyilvánvaló, hogy igen tekintélyes idő szükséges egy viszonylag hosszú és bonyolult számítási feladat elvégzéséhez még abban az esetben is, ha a műveletek végrehajtási ideje ugyyszólván elhanyagolható – tegyük fel, hogy a mp töredéke csupán –, hiszen az un. kézi vezérlés az, amelynek időszükséglete dominál.

Számítógépeknél általában automatikus vezérlésen tehát azt értjük, hogy a teljes utasítássorozatot (programot) előre megfogalmazzuk, és a gép számára hozzáférhetővé tesszük. Az egyes utasítások végrehajtásakor és azok között a gépet üzemeltető személyzetnek egyáltalán nem kell beavatkoznia. Bár a gép szerkezeti felépítése és a különböző programozási nyelvek módot adnak az ember és a gép "párbeszédére", lehetőleg mégiscsak ritkán, többnyire csak ellenőrzés céljából avatkozunk be.

Végül most kell megmagyaráznunk, mit értünk a "tárolt programu gép" meghatározáson.

Automatikus vezérlés, programutasítások automatikus végrehajtása elképzelhető úgy is, ha a számítógép az utasításokat valamilyen külső berendezésből egyenként kapja, s amint egy-egy utasítást végrehajtott, késedelem nélkül megkapja, értelmezi, és végrehajtja a következőt. Nyilván az időmegtakarítás a kézi vezérléshez viszonyítva lényeges, de egy sokkal fejlettebb megoldással egybevetve ez még mindig igen lassu. Ennek az un. külső programvezérlésnek még egy, a lassuságnál is jelentősebb hátránya van, amelyről majd alább lesz szó.

A tárolt programu számítógép számára nemcsak előre elkészítjük a teljes programot, hanem ezt el is juttatjuk a számológép belsejébe.

1945-ben Neumann János, a híres magyar származásu matematikus egy jelentésében megfogalmazta a tárolt programu gépek elvét, azt az elvet, hogy az utasításokat is – akárcsak azokat a memmiségeket, amelyek majd az utasítások operandusai lesznek –, a gép belső tárolójában, a memóriában tárolják. Ez az elgondolás azóta is a modern számítógépek alapelve. Ennek alapján az első, valóban teljesen elektronikus elemeket tartalmazó számítógéptől (ENIAC) eltérő gépeket szerkesztettek, amelyeknél a tárolt program és a kettes számrendszer alkalmazása sokkal nagyobb teljesítményt tett lehetővé kisebb költségű technikai eszközök által. A Neumann-elv legnagyobb jelentősége végül az, hogy az utasításokat a memóriában számok formájában ábrázolják. Az utasításokat jelentő számokkal szintén végezhető műveletek, amelyek ilyen

módon új utasításokat hozhatnak létre. Így a program a számolás során saját utasításait képes módosítani. Külső programvezérlés esetén erre természetesen nincs mód, ha valamilyen elgondolás alapján összeállítottunk egy programot, akkor a gép azt, és csakis azt hajtja végre, míg belső programvezérlés, Neumann elv alapján szerkesztett gép programja nem "statikus", hanem "dinamikus" utasítássorozat.

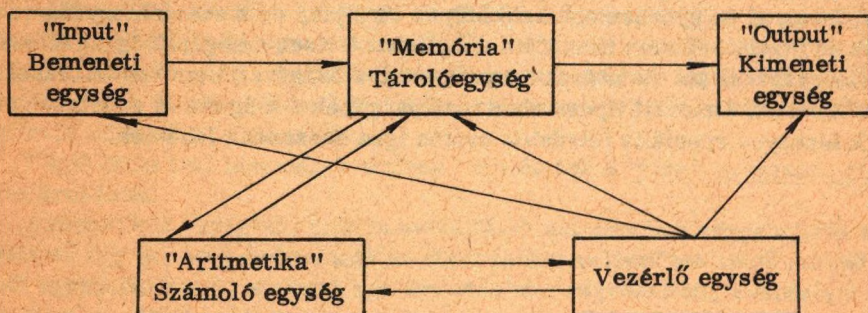
Nagyrészt csak a programozó ügyességétől függ, hogy az ebből az elvből eredő lehetőségeket milyen mértékben tudja kihasználni. A számítógép elvi felépítése lehetővé teszi a programozó elgondolásainak megvalósulását.

Az utasítások megváltoztatása a számolás során, az egyes aritmetikai műveletek elvégzése között, - ez igen bonyolult, és nagy szakértelmet és gyakorlatot igénylő kérdés. Automatikus programozási nyelvek használata nélkül még a matematikusok között is szűk lenne az a kör, amely a Neumann-gépekre teljesítményükkel arányos programokat tudna készíteni. Mint látni fogjuk, automatikus programozás által viszonylag igen egyszerűen készíthetők világos, áttekinthető programok igen bonyolult problémák megoldására is.

Végül megemlítjük, hogy a külső és belső programvezérlés típusa között léteznek átmeneti típusú gépek is. Ezek a programot valamely belső tárolójukban, könnyen és gyorsan elérhető helyen tárolják, de a számolóegység részére ezek az utasítások nem hozzáférhetők, tehát a Neumann-gépek igazi előnyös tulajdonságával (az utasítások megváltoztathatóságával) nem rendelkeznek. Mindamellet átmeneti típusú elektronikus gépeket még ma is gyártanak, és azok bizonyos speciális feladatok esetén igen hasznosak lehetnek.

3. A SZÁMITÓGÉP FŐ RÉSZEI

A programvezérlésű digitális számítógép öt fő egységből áll, annak az öt fő funkciónak megfelelően, amelyeket egy számítási feladat automatikus megoldásához el kell végezni. Az egységek a következők: bemeneti egység, tárolóegység (memória), számolóegység (aritmetikai egység), vezérlő egység, kimeneti egység. Ma is ezek a legfontosabb egységek, és el sem képzelhető, hogy ezek közül valamelyiket nélkülözni lehetne. Fentiekén kívül a számítógépnek tápegysége is van, de erre most nem térünk ki bővebben.



1. ábra

A fenti ábrán feltüntettük a számítógép öt fő egységét. A gép működése az egyes "dobozok" közötti kapcsolatot kifejező vonalak alapján végigkövethető. Hangsúlyozzuk, hogy bár a mai programvezérlésű számítógépek kivétel nélkül elektronikus stb. szerkezeteket használnak fel, de ez egyáltalán nem a programvezérlés lényegéből folyik. Ha csak mechanikus alkatrészeket alkalmaznának, a számítógép sokkal drágább és működési ideje összehasonlíthatatlanul lassabb volna, mégis, mechanikus alkatrészekkel is megvalósítható lenne az 1. ábrának megfelelő gép működése.

Az ábrán nem tüntettük fel, de csaknem minden számítógéphez tartozik egy kézi vezérlő asztal, amely különböző gombokat és karokat tartalmaz a számítógép ki- és bekapcsolására, a program bevitelére, a számolás elindítására és ezen található a különféle kijelző berendezések is (vizuális és akusztikus kijelzők), amelyek az esetleges gép - vagy programhibákat jelzik. A legtöbb gépen igen jó szolgálatot tesz az a hangszóró, amelynek különböző

hangjelzései a gyakorlott fül számára hasznos információt nyújt a számítás menetéről, a program várható időszükségletéről, esetleges hibáiról.

A számítógépek az utasításokat, a műveletekben résztvevő számokat és egyéb szükséges adatokat a memóriában tárolják. Mielőtt a gép elkezdene az az aritmetikai műveletek végrehajtását, a teljes programnak vagy legalább egy programrészletnek a memóriában kell lennie és sorra be kell juttatnunk a memóriába azokat az alapadatokat is, amelyekre az utasítások vonatkoznak. A memória rekeszekre oszlik, amelyek mindegyike egyaránt alkalmas utasítás és adat tárolására. Említettük, hogy az utasítások szintén számok formájában vannak ábrázolva és tárolva. Egy memóriarekesz tartalmát gépi szónak, vagy röviden szónak nevezzük. A tárolt programu gépekben nincs megkülönböztető jel arra, hogy egy szó számadatot vagy utasítást jelent-e, kizárólag a program útján és a memória rekeszeinek sorszámozása (címezése!) segítségével dől el, hogy a memória egy-egy szavát utasításnak vagy egyéb számnak kell-e tekinteni. A memória rekeszeinek sorszáját címnek nevezik.

Egy-egy géptípus jellemző adata a memória elérési (hozzáférési) ideje. Ezen azt az időt értjük, amely egy szónak a memóriába való beírásához vagy az onnét való kiolvasáshoz szükséges.

A jelenlegi gyors számítógépeknél a hozzáférési idő kb. milliomod mp nagyságrendű. Nyilvánvalóan gazdaságos, ha a nagysebességű memóriát nem tervezik túlságosan nagyra, hiszen ennek kivitelezése igen drága: általában kb. 2000-30 000 rekesz a gyorsmemória (másképp: központi vagy operatív memória) kapacitása. Ehhez különböző nagy kapacitású, de lassabb segédberendezéseket: háttérmemóriákat, pl. mágnesdob, mágneslemez, mágnesfilm, csatlakoztatnak.

A háttérmemóriák csak az operatív memóriákkal vannak összekötve, közvetlenül nem érhetők el. Az adatokat, mielőtt azokat a háttérmemóriába juttatnánk, előbb be kell vinnünk az operatív memóriába. A háttérmemóriának mégis igen nagy jelentősége van, mert általában többmillió adatot is képes tárolni. Kellő szervezéssel lehetőség szerint úgy csoportosítják az utasításokat, hogy az információknak (számadatoknak és utasításoknak) azt a részét, amelyre csak a számítás egy későbbi fázisában lesz szükség, átmenetileg a háttérmemóriában helyezik el. A gép vezérlőegysége a gyors tárolóból dolgozik, a program utasításait és az éppen szükséges adathalmazt onnan veszi, de amíg ezekkel számol, a háttérmemória már a legközelebb szükséges utasításcsoport vagy adathalmaz helyének megkeresésével foglalkozik, vagy esetleg a számolási munkával párhuzamosan már folyik is ezek átírása a háttérmemóriából az operatív memóriába.

Mind a gyors memória rekeszeire, mind a háttérmemória tömbjeire érvényes az, hogy egy új szó beírása törli a régi tartalmat, a memóriából történő kiolvasás után viszont a régi tartalom változatlan marad.

A tároló egység műszakilag többféle alakban valósítható meg. A régebben alkalmazott típusokat: akusztikus késleltető vonalas és elektrosztatikus tárolót, mágneses dobot (dobmemóriát), ma már szinte teljesen kiszorította a ferritgyűrűs rendszerű tárolóegység. Dobmemóriát háttérként alkalmaznak ma is.

Mivel a jelen jegyzet célja az, hogy az érdeklődők a lehető leggyorsab-

ban elsajátíthatassák az EMG 830 típusu számítógép autokódját, műszaki kérdések tárgyalására itt nem vállalkozunk.

A tulajdonképpeni számolás a számítógép aritmetikai egységében történik. Ennek elsőrendű feladata az aritmetikai alapműveletek elvégzése. Ha pl. két rekesz tartalmát összeadjuk, akkor a műveletben résztvevő számokat először be kell hoznunk az aritmetikai egységbe, majd a művelet elvégzése után arról is gondoskodnunk kell, hogy az eredmény a memória kívánt című rekeszébe jusson.

A kivonás művelete rendszerint az összeadásra van visszavezetve, a negatív számok különleges kódolása segítségével. A szorzást és az osztást, a mechanikus asztali számológépekhez hasonlóan összeadásra, illetve kivonásra lehet visszavezetni. Az algoritmus, amellyel a gép szoroz és oszt, lehet egészen különböző attól, amelyet a kézi számolásnál használunk, de meg is egyezhet vele. Az aritmetikai egység a négy alapműveleten kívül egyszerű logikai műveletek (Boole algebrai műveletek, pl. konjunkció, diszjunkció stb.) elvégzésére is képes. Az utóbbiakat az 5. fejezetben fogjuk röviden tárgyalni. Ezek a műveletek a legkülönbözőbb számítások során igen gyakran használatosak.

A vezérlőegység gondoskodik a számológép különböző egységei közötti összeköttetés megteremtéséről, lényegében nem csinál mást, mint megfelelő időpillanatokban mindig a megfelelő kapukat nyitja és csukja. Feladata a program utasításainak megvalósítása, a gép különböző, igen összetett szerkezetei irányítása révén.

A program utasításai - mint már mondtuk - programbevitelkor a memória rekeszeibe kerülnek. Amikor a kezelő elindítja a számolást, az első utasítás címe a sorszámtároló regiszterbe, másképpen utasításszámlálóba kerül. A vezérlőegység e cím alapján hívja be a memóriából a számológépben levő utasításregiszterbe az első utasítást, megállapítja annak minőségét, és a szükséges jelzések kibocsátása által megteremti a művelet végrehajtásának feltételeit.

A vezérlés szinkron vagy aszinkron rendszerű lehet. A szinkron rendszerű gépeknél egy-egy utasítás végrehajtási ideje állandó. Ezen pontosabban azt értjük, hogy egy bizonyos utasítás elvégzésére meghatározott időt szabtak. Nyilvánvalóan az előfordulható legnagyobb időszükségletet kénytelenek figyelembe venni a számítógép tervezői. (Pl. az Ural I. típusu számítógépnél nemcsak egy-egy utasítás típus végrehajtási ideje állandó, hanem csaknem valamennyi különböző utasításé is.) Aszinkron rendszernél az azonos típusú utasítás elvégzési ideje is különbözik, ekkor azonban szükség van egy jelzőszerkezetre, amely azonnal jelez, amint az utasítás végrehajtása befejeződött.

Felhívjuk a figyelmet arra, hogy a vezérlőegység nem tudja megkülönböztetni a rekeszekben tárolt szavakat abból a szempontból, hogy azok utasítások vagy adatok. Ha egy rekesz tartalma az utasításregiszterbe került, akkor azt a tartalmat már mint utasítást fogja kezelni. Hibás programozás esetén viszonylag gyakran előáll ez a helyzet, ami annál is kellemetlenebb, mert a továbbiakban a legabszurdabb számolási lépések következhetnek, a program futása félrevezető, vagy követhetetlen lehet.

Itt említjük meg egészen röviden azt az igen lényeges elvet, hogy a vég-

rehajtandó utasításokat a vezérlőegység általában az egymást követő rekeszekből veszi. Ez úgy valósul meg, hogy a számolás megkezdésekor az első utasítás címét tároló utasításszámláló regiszter tartalma mindenegybes az aritmetikai egység által végrehajtott utasítás elvégzése után eggyel növekszik és ez a szám jelzi a következő végrehajtandó utasítás helyét.

Vannak azonban olyan utasítások is, amelyek magának a vezérlőegységnek szólnak, annak működését befolyásolják.

Minden gép rendelkezik vezérlésátadó (ugró, ugrató) utasításokkal, amelyek arra szolgálnak, hogy az utasítások normális végrehajtási sorrendje megszakadjon. Ilyen esetben is azon rekesz címe kerül az utasításszámláló regiszterbe, amelyből majd a következő végrehajtandó utasítást venni kell, azonban ez a szám már nem eggyel nagyobb az előzőnél, hanem valamilyen, 1-től különböző pozitív vagy negatív szám hozzáadása révén alakul ki. A továbbiakban a vezérlőegység a megadott címen talált tartalmat az utasításregiszterbe tölti át és a továbbiakban ezen utasítás elemzésével és végrehajtásával folytatódik a számítás.

A vezérlésátadó utasítások feltétlenek és feltételesek lehetnek. Feltétlen vezérlésátadás esetén azonnal a megfelelőképpen módosul az utasításszámláló regiszter tartalma, feltételes vezérlésátadásnál azonban az aritmetikai egység megvizsgálja, hogy az utasításban megjelölt feltétel teljesül-e vagy sem. Teljesülés esetén megtörténik a vezérlésátadás, ellenkező esetben nem. Az utóbbi esetben az utasításszámláló regiszter tartalma ismét csak eggyel nő, vagyis a soronkövetkező rekeszben tárolt utasítás végrehajtása fog majd következni.

Az aritmetikai és a vezérlőegység elektronikus szerelvényekből áll és mindkettő lényegében logikai egységáramkörökből épül fel. A legtöbb számológép a bináris (kettes) számrendszer alkalmazásával épült fel. A bináris számrendszer alkalmazását is Neumann János javasolta a már említett jelentésben, mert ezzel lényegesen egyszerűsíteni lehetett a gépek szerkezetét, csökkent a szükséges alkatrészek száma, kisebb lett a meghibásodás valószínűsége is. Számoknak bináris számrendszerben való ábrázolásánál csak kétfajta számjegy szükséges: 1 és 0. Ezért alkalmas ez a számrendszer a digitális számítógépekben való felhasználásra: egy impulzus jelenléte jelentheti az 1-es, hiánya a 0 számjegyet. A bináris számrendszer alkalmazása nem szükségszerű körülmény a fentiekben vázolt szerkezetű számítógépeknél. Ha mód lenne pl. 4 különféle, jól meghatározott állapot megkülönböztetésére, elképzelhető lenne a négyes számrendszer használata is. Megjegyezzük, hogy főleg adatfeldolgozásra tervezett gépeknél a decimális rendszer használata ismét elterjedt, de ez nem jelenti azt, hogy decimális jegyeket ábrázoló, több állapotú elektronikus rendszereket alkalmaznak, hanem minden decimális jegyet bináris számjegyeknek (biteknek) egy csoportja képvisel.

Az aritmetikai és a vezérlőegységet alkotó logikai áramkörök között a legfontosabbak az un. "és", a "vagy" és a "nem" áramkörök, amelyek segítségével realizálni lehet a bináris összeadás logikai formáját és az összeadó blokk gyakorlatilag is megépíthető.

Mint már említettük, a kettes számrendszer különösen alkalmas arra, hogy matematikai gépekben használják. A tévedésmentes működés leginkább olyan

rekeszben tárolódik, de a gyakorlott programozó számára ez sem megoldhatatlan, bár bonyolultabb feladat.

A kimeneti egységek is különfélék lehetnek. A korszerű számítógépekhez a megrendelő kívánsága szerint csatolhatók az igényeit legjobban kielégítő berendezések: egy vagy több szalaglyukasztó, kártyalyukasztó, sornyomtató berendezés stb.

A szalaglyukasztó általában ugyanolyan kódban, amelyet a bevitelnél is használnak, szalagra nyomtatja a program által számított eredményeket. Ezt az eredményszalagot azokon a távgépiró rendszerű gépeken, amelyek a bemenő szalagokat készítik, ki lehet iratni. Annak ellenére, hogy így csak közvetve jutunk hozzá a valódi, olvasható eredményhez, mégis igen jó szolgálatot tesz a lyukszalag, ui. igen gyakran használjuk a valamely program által készített eredményszalagot egy másik program adatszalaiaként.

Olyan esetben, amikor már számolás közben látni akarjuk az eredményeket, és ezekkel nem akarunk tovább számolni, célszerű a kiíratást a számítógéphez csatolt írógéppel elvégeztetni. A szalaglyukasztó és az írógép a számolási időhöz viszonyítva elég lassan működik, ezért olyan programoknál, amelyek nagymennyiségű számokat adnak eredményül, szívesen használják a sornyomtató berendezéseket. A sornyomtató egy teljes sort (általában 120-150 jelet: számjegyet vagy betűt) nyomtat ki egyszerre. A kinyomtatandó számjegyeket, betűket, jeleket saját belső tárolójában gyűjti mindaddig, amíg a program nem érkezik sornyomtató utasításhoz.

A modern programozási nyelvek lehetőséget adnak arra, hogy a program váltakozva használja a különböző típusú kimeneti egységeket, s így megoldható az, hogy egy-egy számhalmaz a további rendeltetésének legjobban megfelelő egységen jelenjék meg.

Kimeneti egységnek lehet tekinteni a mágnesfilmeket (mágneslemezeket stb.) is, mert pl. abban az esetben, ha egy 1. sz. program viszonylag sok számot ad eredményül és ezekkel majd egy 2. sz. program tovább fog számolni később, időmegtakarítás szempontjából a legjobb megoldás, ha ezeket a számokat az 1. sz. program filmre írja és erről a filmről kapja majd a 2. sz. program a bemeneti adatait.

Megemlíthetjük még a különféle speciális kimeneti berendezéseket: a hangszórót, a televíziót, a rajzoló automatát. Az elektronikus számítógéppel szerzett "zene" tárgyalására itt nincs mód. A televízió mint a számítógép kijelző berendezése, valószínűleg el fog terjedni a következő években, de egyelőre a hazánkban alkalmazott számítógépek még nincsenek ilyen kimeneti egységgel felszerelve. A mérnöki gyakorlat számára nagy jelentősége van a rajzoló automatáknak. Ezeket a gépeket közvetlenül is csatlakoztatni lehet a digitális számítógépekhez, és akkor a számított eredmények közvetlenül grafikus formában, folytonos jelsorozat formájában nyerhetők.

Közvetlenül (angol kifejezés erre a megoldásra: on line) szoktak igen gyakran távgépirókat (telexgépeket) csatolni a számítógépekhez. Ezzel a megoldással a távoli adatforrásból érkező információ a lehető leggyorsabban eljut a számítógéphez és a számított eredmény ugyanilyen módon kerülhet vissza

ugyanoda, vagy máshová, az eredmény felhasználójához és alkalmazható pl. mint egy folyamatszabályozó berendezés része.

Amint láttuk, a be- és kimeneti egységek - közös néven: perifériák - nagy számban és választékban csatlakoztathatók egyetlen központi egységhez. A legújabb számítógépek - köztük az EMG 830-as is - lényeges jellemző vonása az, hogy egyszerre több program is futhat a számítógépen. Ez az ún. multiprogramozás, többféleképpen valósítható meg, de bármilyen rendszerű is a belső megoldás, lényegében az történik, hogy ha a memóriában elhelyezett egyik program futása közben olyan utasításokhoz ér, amelyek a perifériás egységeket működtetik, addig valamelyik másik, szintén a memóriában tárolt program igénybe veszi a számítógépet és teljesíti a saját utasításait. Ha most esetleg ez a második program is beolvasó, vagy kíró utasításhoz érkezik, akkor egy harmadik program "veszi birtokába" a szabaddá vált számoló egységet. Az egyes programok megosztoznak a bemeneti és kimeneti egységeken. Így szünet nélkül üzemben van a számítógép csaknem valamennyi egysége és végeredményben valamennyi program előbb lesz készen a számítással, mintha egyenként, egymás után futtattuk volna őket.

Multiprogramozható gépekhez rendszerint a gyártó cég maga készít egy szervező programot (angolul: executive), amelynek feladata a szimultán számoló programok koordinálása, a memória rekeszeinek felosztása a programok között, az állandó ellenőrzés, hogy az egyes programok ne zavarhassák meg egymás működését, ne használjanak olyan rekeszeket, amelyek kívül fekszenek a részükre megállapított határokon stb. A programozónak többnyire nem kell foglalkoznia ezekkel a bonyolult kérdésekkel, a gépet üzemeltető személyzet mindössze egy prioritási sorrendet ad meg a szimultán dolgozó programok számára. Véleményünk szerint a multiprogramozhatóság a közeljövőben a gyors számológépek egyik elengedhetetlen jellemvonása lesz, de ez nem valószínű volt még a Neumann elv alkalmazása nélkül.

4. A SZÁMÁBRÁZOLÁS A PROGRAMVEZÉRELT, TÁROLT PROGRAMU, DIGITÁLIS ELEKTRONIKUS SZÁMITÓGÉPEKEN

A továbbiakban - tekintet nélkül arra, hogy milyen műszaki megoldásokkal valósítható ez meg, - arról lesz szó, milyen elvek szerint ábrázolnak a memória egy-egy rekeszében számokat.

Gondoljuk meg, hogy egyes programok esetén ezeknek a számoknak $10^{20} - 10^{40}$ nagyságrendűeknek kell lenniök, máskor pedig olyan kicsiny mennyiségekkel akarunk műveleteket végezni, amelyek abszolút értékben közelállnak zérushoz. Hogyan lehetséges az, hogy sok nagyságrend különbséggel tudunk számokat ábrázolni ugyanazokban a rekeszekben? Hogyan tudunk negatív és pozitív számokat tárolni és velük egységes elveken alapuló számolási műveleteket végezni?

Mint már említettük, a legbiztosabb működésre leginkább abban az esetben számíthatunk, ha csak két diszkrét állapottal rendelkező alkatrészeket használunk fel. Minden rekeszben azonos számú, kétállapotú információhordozó egység található, ezen egységeket biteknek nevezzük. A bit szó az angol binary digit (bináris szám) szó alapján terjedt el az elektronikus számítógép technikában. Leggyakrabban - típustól függően - 20 - 48 bit egy memóriarekesz szóhossza, s ez egyben azt is jelenti, hogy ugyanannyi, egymástól független információ tárolható egy rekeszben. A bitekhez a lehetséges két állapot alapján a 0 vagy az 1 számot rendelhetjük hozzá.

Bináris gépeknél a memóriarekeszek száma általában kettő hatvány, így a gépek memóriakapacitása 4096, 8192, 16 384, 32 768 stb. rekesz, de szokás a memóriakapacitást bitekben is megadni.

Ha azt halljuk, hogy valamely gép memóriakapacitása 32 K, akkor ezen a 32 768 rekeszes kapacitást kell érteni.

Írjuk fel most egy kettes számrendszerben ábrázolt szám általános alakját.

Egy adott N valós decimális számot kettes számrendszerbe átírva az alábbi alakot nyerjük:

$$N = a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \dots + a_1 \cdot 2^1 + a_0 \cdot 2^0 + a_{-1} \cdot 2^{-1} + \\ + a_{-2} \cdot 2^{-2} + \dots + a_{-m} \cdot 2^{-m}$$

ahol az a_i ($i = n, n-1, \dots, 0, -1, \dots, -m$) együtthatók csak 0 vagy 1 lehetnek.

4. A SZÁMÁBRÁZOLÁS A PROGRAMVEZÉRELT, TÁROLT PROGRAMU, DIGITÁLIS ELEKTRONIKUS SZÁMITÓGÉPEKEN

A továbbiakban - tekintet nélkül arra, hogy milyen műszaki megoldásokkal valósítható ez meg, - arról lesz szó, milyen elvek szerint ábrázolnak a memória egy-egy rekeszében számokat.

Gondoljuk meg, hogy egyes programok esetén ezeknek a számoknak 10^{20} - 10^{40} nagyságrendűeknek kell lenniök, máskor pedig olyan kicsiny mennyiségekkel akarunk műveleteket végezni, amelyek abszolút értékben közelállnak zérushoz. Hogyan lehetséges az, hogy sok nagyságrend különbséggel tudunk számokat ábrázolni ugyanazokban a rekeszekben? Hogyan tudunk negatív és pozitív számokat tárolni és velük egységes elveken alapuló számolási műveleteket végezni?

Mint már említettük, a legbiztosabb működésre leginkább abban az esetben számíthatunk, ha csak két diszkrét állapottal rendelkező alkatrészeket használunk fel. Minden rekeszben azonos számú, kétállapotú információhordozó egység található, ezen egységeket biteknek nevezzük. A bit szó az angol binary digit (bináris szám) szó alapján terjedt el az elektronikus számítógép technikában. Leggyakrabban - típustól függően - 20 - 48 bit egy memóriarekesz szóhosszusága, s ez egyben azt is jelenti, hogy ugyanannyi, egymástól független információ tárolható egy rekeszben. A bitekhez a lehetséges két állapot alapján a 0 vagy az 1 számot rendelhetjük hozzá.

Bináris gépeknél a memóriarekeszek száma általában kettő hatvány, így a gépek memóriakapacitása 4096, 8192, 16 384, 32 768 stb. rekesz, de szokás a memóriakapacitást bitekben is megadni.

Ha azt halljuk, hogy valamely gép memóriakapacitása 32 K, akkor ezen a 32 768 rekeszes kapacitást kell érteni.

Írjuk fel most egy kettes számrendszerben ábrázolt szám általános alakját.

Egy adott N valós decimális számot kettes számrendszerbe átírva az alábbi alakot nyerjük:

$$N = a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \dots + a_1 \cdot 2^1 + a_0 \cdot 2^0 + a_{-1} \cdot 2^{-1} + a_{-2} \cdot 2^{-2} + \dots + a_{-m} \cdot 2^{-m}$$

ahol az a_i ($i = n, n-1, \dots, 0, -1, \dots, -m$) együtthatók csak 0 vagy 1 lehetnek.

Ha kétállapotú információhordozó egységek valamely sorozatában a tizedesponthelyét rögzítjük, az információhordozók egy-egy állapota egy-egy bináris számot reprezentál.

Szorítkozzunk most csak a decimális egész számokra. Nyilvánvaló, hogy a rekesz szóhossza (a rekeszben foglalt bitek száma) meghatározza az ábrázolható maximális egész számot. A biteknek jobbról kiindulva a helyiérték fogalmának megfelelően kettő hatvány értéket tulajdonítunk, és így pl. egy 5 bites memóriában a következő számot ábrázolhatjuk:

2^4	2^3	2^2	2^1	2^0
1	0	1	0	1

Az itt ábrázolt egész szám decimális értéke:

$$1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 21$$

Általában egy $n+1$ bites memóriarekesz egy bite (a szemléletnek megfelelően a bal szélső bit) a szám előjelét jelzi: a leggyakrabban 0 az előjelbit, ha a szám pozitív és 1, ha a szám negatív.

(Felhívjuk a figyelmet arra, hogy géptípustól függően másfajta megoldások is elképzelhetők az előjel jelzésére.)

Az ábrázolható maximális pozitív számot akkor kapjuk, ha az előjelbit kivételével minden helyen egyes áll:

Előjel	2^{n-1}	2^{n-2}	2^{n-3}		2^3	2^2	2^1	2^0
0	1	1	1		1	1	1	1
$n+1$. bit	n . bit	.	.		4. bit	3. bit	2. bit	1. bit

$$N = 2^{n-1} + 2^{n-2} + 2^{n-3} + \dots + 2^2 + 2^1 + 2^0 = 2^n - 1$$

A negatív számok ábrázolására - mint már jeleztük - több lehetőség van. A korszerű számítógépek többsége, így az EMG 830 számítógép is, ún. kettes komplementáris kódban ábrázolja a negatív számokat. Ezen ábrázolási módnak az a követelmény a lényege, hogy azonos abszolút értékű pozitív és negatív számok összege zérus legyen. Ez úgy érhető el formálisan, ha a negatív számot úgy képezzük, hogy a megfelelő pozitív bináris szám jegyeit jobbról kiindulva az első egyesig bezárólag nem változtatjuk meg, azután azonban valamennyi jegyet - beleértve az előjelet is - 0-ról 1-re, illetve 1-ről 0-ra változtatunk.

Tekintsünk most egy előjelbittel együtt 6 bites memóriarekeszt és ábrázoljuk abban a + 6 számot:

Előjel	2^4	2^3	2^2	2^1	2^0	
0	0	0	1	1	0	= + 6

A fenti szabály alapján - 6 ábrázolási módja a következő lesz:

1	1	1	0	1	0	= - 6
---	---	---	---	---	---	-------

Adjuk össze e két számot a bináris aritmetika törvényei szerint:

$$\begin{array}{r}
 000110 \\
 + 111010 \\
 \hline
 1\ 000000
 \end{array}$$

Mind a hat biten 0-t kaptunk, a bal szélén képződött egyes elvész, "kicsurog".

Mint látjuk, komplementáris kódban az előjelbit valódi, a számhoz aritmetikailag is hozzátartozó bit. Ha a szokásos, matematikai ábrázolási módot akarnánk megtartani, a számokat előjelükkel és abszolút értékükkel ábrázolnánk. Ez a kódolás az ún. direkt vagy egyenest kód. Az olvasónak valószínűleg áttekinthetőbbnek tűnik, mint a komplementáris kód, de a bemutatott példa alapján is meggondolhatjuk, hogy az összeadás műveletének belső megoldása műszakilag bonyolultabb lesz, mint komplementáris kódban ábrázolt számok esetén.

Többé-kevésbé használatos még a fordított, vagy másképp inverz kód, mely a komplementáris kódhoz némileg hasonlóképpen jár el, ekkor azonban valamennyi jegy megfordul. (Decimális rendszerben az inverz kódnak a helyiértékenként 9-re való kiegészítés felel meg.)

Az előbbiekben példákkal is bemutatott számábrázolás az ún. fixpontos számábrázolás volt.

Programozástechnikai kérdés az, hogy ha erre szükség van, a szóhosszúság meghatározta számnál nagyobb mennyiségeket is ábrázolni tudjunk hasonló módon. Mivel általában csak ritkán, speciális esetekben van arra szükség, hogy igen nagy egész számokat ábrázoljunk a fent ismertetett elvek szerint, nem gazdaságos a memória szóhosszuságát tulságosan növelni. A következőkben majd látni fogjuk, hogyan lehet ugyanolyan hosszú memóriarekeszben több nagyságrenddel nagyobb számokat is elhelyezni.

Fixpontosan ábrázolhatunk és tárolhatunk számokat oly módon is, hogy egy-egy számot két egymásután következő rekeszben helyezünk el. Az ilyen számábrázolást dupla hosszúságú fixpontos számábrázolásnak nevezzük. Elképzeltető, hogy a programozásban jártas programozó maga készít olyan el-

járásokat, amelyek segítségével egy-egy számot két egymást követő rekeszbe tud betáplálni, és az ezekkel való számolási műveletek kérdését is megoldja, a szokásos mégis az, hogy a tervezők a számológépet már eleve ilyen lehetőségekkel felszerelve bocsátják a felhasználók rendelkezésére.

Az EMG 830 típusu számítógép autokódja által - amint azt látni fogjuk, - a programozók viszonylag igen könnyen hozzájutnak a dupla hosszúságú szám-ábrázolás előnyeinek.

A fixpontos számábrázolási mód mellett minden esetben szükség van egy másik elven alapuló számábrázolási módszerre is. Mindaddig nem volt arról szó, hogy miképpen lehet ugyanolyan rekeszekben, mint amelyeket eddig vázoltunk, törtszámokat is elhelyezni. Erre szolgál az ún. lebegőpontos számábrázolási mód.

Ismeretes, mit értünk a számok normál alakján. Valamely viszonylag nagy vagy viszonylag kis szám esetén a kézi számolásnál is előnyt, könnyebb kezelhetőséget jelent az, ha a decimális számokat pl. $5,6 \cdot 10^9$ vagy $3,2 \cdot 10^{-8}$ alakban adjuk meg. Hasonló normalizálásról van szó a lebegőpontos számábrázolásnál is, de most nem 10, hanem 2 hatványai szerint történik a normalizálás. (A szakirodalomban ezt az eljárást számítógépekkel kapcsolatban standardizálásnak is nevezik.)

Lebegőpontos számábrázolásnál a rekeszeket meghatározott számú bitek közül álló két-két részre osztják. A bitek egyik csoportja a szám nagyságrendjét a karakterisztikát ábrázolja. A másik csoport magát a normalizált számot, a mantisszát ábrázolja, de csak olyan pontossággal, amelyet az arra a célra szolgáló bitek száma megenged. Minél több bit tartozik a karakterisztikához, annál kevesebb marad a mantisszára és fordítva, hiszen a rekesz biteinek száma adott. Mind a karakterisztikához, mind a mantisszához egy-egy előjel-bit tartozik.

A régebbi típusu számítógépeknél a programozónak saját programjában kellett előállítania a tárolandó szám lebegőpontos alakját. Általában rendelkezésre álltak olyan kész utasításcsoportok (szubrutinok), amelyeknek az volt a funkciója, hogy a beolvasott számnak megfelelő karakterisztika- és mantissza-biteket előállítsa és elhelyezze a kívánt rekeszbe. Ez az állandóan viszszafelelő művelet, a normalizálás nagymértékben megnövelte mind a programozó gondját, mind a program futási idejét. Kinyomtatás előtt el kellett végezni a normalizálás reciprok műveletét is.

A korszerű számítógépek lebegőpontos programozási lehetőséggel vannak felszerelve és ez megkönnyíti a programozót attól, hogy a normalizálás munkáját maga végezze el. A programban csak azt kell megadni, melyik rekeszben akarunk lebegőpontosan ábrázolt számot tárolni és melyikben fixpontosat, a továbbiakban a számítógép automatikusan megvalósítja a kívánt rekeszben a meghatározott számábrázolási formát.

Egész számokat mindkét típusban lehet ábrázolni. Hogy a kétféle számábrázolási lehetőség közötti különbséget személtessük, példaképpen bemutatjuk egy 39 bites memóriarekeszben a decimális 17 ábrázolását fixpontos és lebegőpontos eljárással.

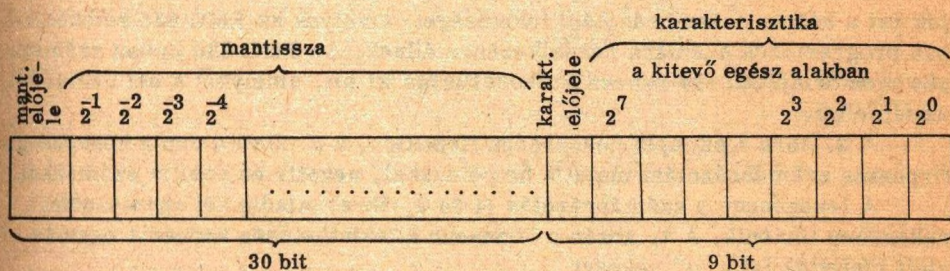
Fixpontosan:

$$\begin{array}{cccccccccccc}
 & 2^{38} & 2^{37} & 2^{36} & & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
 17 = & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{\dots} & \boxed{0} & \boxed{0} & \boxed{1} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{1} \\
 & 39 & 38 & 37 & & 7 & 6 & 5 & 4 & 3 & 2 & 1
 \end{array}$$

Az n szám normalizálásához kettőnek azt a k hatványát keressük, amelyre teljesül, hogy

$$\frac{1}{2} \leq \frac{n}{2^k} < 1$$

Hogy miért éppen az a követelmény, hogy a leosztott szám $1/2$ és 1 közé essék, kiderül a bitek helyértéke új jelentésének tanulmányozása során. Ekkor ugyanaz a 39 bites memóriarekesz pl. ilyen arányban osztható ketté a karakterisztika és a mantissza tárolására:



$$n = 17 = \frac{17}{2^5} \cdot 2^5 = \frac{17}{32} \cdot 2^5 \text{ ábrázolása ebben}$$

a rekeszben:

$2^{-1} \quad 2^{-2} \quad 2^{-3} \quad 2^{-4}$										$2^2 \quad 2^1 \quad 2^0$								
0	1	0	0	0	1	0	0			.	.	.	0	0	0	1	0	1

Természetesen más arányban is ketté lehet osztani a biteket a karakterisztika és a mantissza meghatározására. Nagyobb karakterisztika rész esetén kevesebb bit marad a mantisszára, s ez azt jelenti, hogy a leosztott számot csak kisebb pontossággal tárolja a memória. A karakterisztika és a mantissza "hossza" egy adott lebegőpontos egység esetén meghatározott állandó

számok, így a programozó tudja, hogy a lebegőpontosan ábrázolt szám hány decimális jegynek megfelelően pontos.

Lebegőpontos számábrázolásnál a karakterisztika bitjeinek száma szal meg az ábrázolhatóság határait. A bemutatott memóriatípus esetén ezek a határok:

$(2^{-256}, 2^{255})$, azaz a decimális számok kb. $10^{-78} - 10^{76}$ nagyságrendben ábrázolhatók. A fixpontos számábrázolási módszer ugyancsak 39 bites memóriarekesz esetén maximálisan csak 12 jegyű decimális szám ábrázolására ad lehetőséget.

A lebegőpontos számábrázolás tehát kettős haszonnal jár: megoldja a törtszámok ábrázolásának kérdését és sok nagyságrenddel kitölti az ábrázolhatóság határait. Egész számokat egyaránt lehet fixpontosan és lebegőpontosan is ábrázolni. Megjegyezzük, hogy lebegőpontosan ábrázolt számokkal a számítógép lassabban hajt végre műveleteket, mint fixpontosokkal, továbbá a fixpontos szám "pontosabb" is, ezért a lebegőpontos ábrázolás mellett továbbra is megvan a szerepe a fixpontos számábrázolásnak is, ha biztosan tudjuk, hogy milyen adatok azok, amelyek csak egész értéket vehetnek fel a programban.

A következőkben táblázatban összefoglalt különböző számokkal bemutatjuk azt a három számábrázolási lehetőséget, amelyek az EMG 830 autókódjában programozók számára rendelkezésre állnak. Az EMG 830 típusú számítógép memóriájában egy rekesz szóhosszúsága 21 bit, amelyből a bal szélső bit az előjelbit.

A 2. ábra a szimpla hosszúságú fixpontos, a 3. ábra a dupla hosszúságú fixpontos számábrázolást mutatja be példákkal, negatív és pozitív számokkal.

A lebegőpontos számábrázolás (4 és 5. ábra) mindig két egymásutáni rekeszben történik. A 4. ábrán a könnyebb áttekinthetőség kedvéért egymás alatt tüntettük fel a két rekeszt.

Felhívjuk az olvasó figyelmét arra, hogy EMG autókódban könnyen és jól programozható a soronkövetkező táblázatok áttanulmányozása nélkül is: elég azt tudnia, mi a háromféle számbárázolási mód lényege, melyek a határ

Szimia hosszúságu fixpontos számábrázolás

Az előjelet a 20. bit jelzi (0-20)

Negatív esetben a komplementáris ábrázolási mód a mérvédő.

20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$a = - \left\{ 2^{21} - 2^{20} \right\} = - \left\{ 2^{20} (2-1) \right\} = -2^{20}$ (legkisebb)
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	$a = - \left\{ 2^{21} - (2^{20} + 2^0) \right\} = - \left\{ 2^{20} - 1 \right\} = -2^{20} + 1$
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	$a = - \left\{ 2^{21} - (2^{20} + 2^1) \right\} = - \left\{ 2^{20} - 2 \right\} = -2^{20} + 2$
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	$a = - \left\{ 2^{21} - (2^{20} + 2^1 + 2^0) \right\} = - \left\{ 2^{20} - 2 - 1 \right\} = - \left\{ 2^{20} - 3 \right\} = -2^{20} + 3$
...																					
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	$a = - \left\{ 2^{21} - (2^{20} + 2^{18} + \dots + 2^0) \right\} = - \left\{ 2^{20} - (2^{19} - 1) \right\} = - \left\{ 2^{19} + 1 \right\} = -2^{19} - 1$
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$a = - \left\{ 2^{21} - (2^{20} + 2^{19}) \right\} = - \left\{ 2^{20} - 2^{19} \right\} = -2^{19}$
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	$a = - \left\{ 2^{21} - (2^{20} + 2^{19} + 2^0) \right\} = - \left\{ 2^{19} - 1 \right\} = -2^{19} + 1$
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	$a = - \left\{ 2^{21} - (2^{20} + 2^{19} + 2^1) \right\} = - \left\{ 2^{19} - 2 \right\} = -2^{19} + 2$
...																					
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	$a = - \left\{ 2^{21} - [(2^{20} + \dots + 2^2 + 2^1 + 2^0) - (2^1 + 2^0)] \right\} = - \left\{ 2^{21} - [2^{21} - 2 - 1] \right\} = -4$
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	$a = - \left\{ 2^{21} - [(2^{20} + \dots + 2^2 + 2^1 + 2^0) - (2^1)] \right\} = - \left\{ 2^{21} - [2^{21} - 1 - 2] \right\} = -3$
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	$a = - \left\{ 2^{21} - [(2^{20} + \dots + 2^1 + 2^0) - (2^0)] \right\} = - \left\{ 2^{21} - [2^{21} - 1 - 1] \right\} = -2$
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	$a = - \left\{ 2^{21} - (2^{20} + \dots + 2^0) \right\} = - \left\{ 2^{21} - (2^{21} - 1) \right\} = -1$
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$a = 0$
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	$a = 2^0 = 1$
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	$a = 2^1 = 2$
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	$a = 2^1 + 2^0 - 2 + 1 = 3$
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	$a = 2^2 = 4$
...																					
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	$a = 2^{18} + \dots + 2^0 = 2^{19} - 1$
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$a = 2^{19}$
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	$a = 2^{19} + 2^0 = 2^{19} + 1$
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	$a = 2^{19} + 2^1 = 2^{19} + 2$
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	$a = 2^{19} + 2^1 + 2^0 = 2^{19} + 3$
...																					
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	$a = 2^{19} + \dots + 2^2 + 2^1 + 2^0 - (2^1 + 2^0) = 2^{20} - 2^1 - 2^0 - 1 = 2^{20} - 4$
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	$a = 2^{19} + \dots + 2^2 + 2^1 + 2^0 - (2^1) = 2^{20} - 2^1 - 1 = 2^{20} - 3$
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	$a = 2^{19} + \dots + 2^1 + 2^0 - (2^0) = 2^{20} - 2^0 - 1 = 2^{20} - 2$
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	$a = 2^{19} + \dots + 2^0 = 2^{20} - 1$ (legnagyobb)

2. ábra

Duplahosszúságú fixpontos számábrázolás

Az előjelet az első rekesz előjelbitje jelzi, a második szó előjelbitje mindig 0.

Negatív esetben a komplementáris ábrázolási mód a mérvadó.

x.			x+1. rekesz		
20	19...	0	20	19	0
1	00... ..00	0	00... ..00	$a = - \left\{ 2^{41} - 2^{40} \right\} = - 2^{40}$ (legkisebb)	
1	00... ..00	0	00... ..001	$a = - \left\{ 2^{41} - (2^{40} + 2^0) \right\} = - \left\{ 2^{40} - 1 \right\} = - 2^{40} + 1$	
1	11... ..11	0	11... ..110	$a = - \left\{ 2^{41} - (2^{40} + \dots + 2^1 + 2^0 - 2^0) \right\} = - 2$	
1	11... ..11	0	11... ..11	$a = - \left\{ 2^{41} - (2^{40} + \dots + 2^0) \right\} = - 1$	
0	00... ..00	0	00... ..00	$a = 0$	
0	00... ..00	0	00... ..01	$a = 2^0 = 1$	
0	11... ..11	0	11... ..11	$a = 2^{39} + 2^{38} + \dots + 2^0 = 2^{40} - 1$ (legnagyobb)	

3. ábra

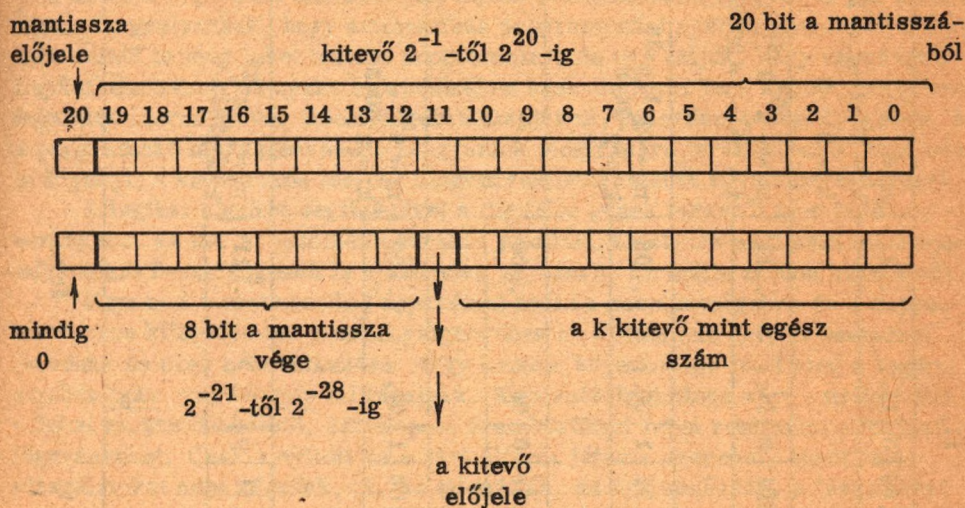
Lebegőpontos számábrázolás

Két egymásutáni rekeszben - kettes számrendszerbe konvertálva úgy,
 hogy

$$X = n \cdot 2^k$$

ahol

$$\left. \begin{array}{l} -1 \leq n < -\frac{1}{2} \text{ ha } n < 0 \\ \frac{1}{2} \leq n < 1 \text{ ha } n > 0 \end{array} \right\} \text{ mantissza}$$



4. ábra

- $n > 0$ esetben a legkisebb mantissza 2^{-1}
 legnagyobb mantissza $2^{-1} + \dots + 2^{-28} = 1 - 2^{-28}$
- $n < 0$ esetben a legkisebb mantissza -1
 legnagyobb mantissza -2^{-28}
- $k > 0$ esetben a legkisebb kitevő 1
 legnagyobb kitevő $2^{11} - 1$ (2047)
- $k < 0$ esetben a legkisebb kitevő -2^{11} (-2048)
 legnagyobb kitevő -1

20...	...	0	20...	..	11...	0			
1	00...	...00	0	00...	...00	0	11..	...11	$m = -2^0$ $k = 2^0 + 2^1 + \dots + 2^{10} = 2^{11} - 1$ -2^{2047}
legkisebb (negatív) szám									
1	00...	...000	0	00...	..001	0	11..	...11	$m = -2^0 - 2^{-28}$ $k = 2^{11} - 1 = 2047$ $-2^{2047} (2^{-28} - 1)$
2^{2019} -cel nagyobb									
1	011...	...11	0	11...	...11	1	00..	...00	$m = -2^{-1} - 2^{-28}$ $k = 2^{-2048}$ $-2^{-2043} (\frac{1}{2} + 2^{-28})$
legnagyobb negatív szám									
0	00...	...00	0	00...	...00	0	00..	...00	0
zérus									
0	100..	...000	0	00...	...00	1	00..	...00	$m = 2^{-1}$ $k = 2^{-2043}$ 2^{-2049}
legkisebb pozitív szám									
0	11...	...11	0	11...	...11	0	11..	...11	$m = 1 - 2^{-28}$ $k = 2^{2047}$ $2^{2047} (1 - 2^{-28})$
legnagyobb (pozitív) szám									

Az alábbiakban egész röviden tárgyalni fogjuk a G. Boole (1815-1864) által kialakított logikai algebra elemeit. A számolóautomatában az aritmetikai műveletek tulajdonképpen véghezvitele logikai műveletekre van bontva, mégsem az egyes funkciók műszaki megoldása szempontjából tárgyaljuk ezt a kérdést. Hangsúlyozzuk, hogy automatikus programozási nyelven mindezek ismerete nélkül is meg lehet tanulni programozni, de úgy véljük, hogy egyrészt a logikai utasítások könnyebb megértéséhez hasznos lesz, ha a kezdő programozó fogalmat szerez az un. Boole-algebra alapjairól, másrészt pedig a jövőben, ha a programozó majd igényesebb programok megírására is vállalkozik, tapasztalni fogja pl. a kétváltozós logikai függvények ismeretéből következő előnyöket.

A logikai algebra segítségével a digitális gépek tervezői igen hatékony eszközhöz, az un. kapcsolásalgebrához jutottak, amelynek segítségével bonyolult logikai összefüggések is viszonylag egyszerű, formális módon kezelhetők.

Ugyanazt a matematikai apparátust fedi az eseményalgebra és a kapcsolásalgebra kifejezés is. Eseményalgebrában az állítás szó helyett eseményt mondunk és ezek bekövetkezése vagy be nem következése felel meg a logikai algebra igaz vagy hamis állításainak. Kapcsolásalgebrában zárt - nyitott állapotokat megkülönböztető, kétállapotú áramkörökkel lehet realizálni a logikai függvényeket. Csak a nyitott és a zárt állapot játszik szerepet, quantitativ vizsgálatokat nem végzünk, pl. az ellenállás, az áramerősség, a feszültség nagysága nem képezi megfigyelés tárgyát.

A kapcsolásalgebra segítségével viszonylag egyszerűvé válik egy adott kapcsolás bizonyos eredményektől függő viselkedésének vizsgálata. A kapcsolások egyszerűsítése, a legkisebb technikai ráfordítással megvalósítható megoldások kikeresése szintén a kapcsolásalgebra alkalmazásával történik.

A kétértékű logikában csak kétféle ítélet fordulhat elő: igaz vagy hamis. Egy logikai változó csak kétféle értéket vehet fel: igaz vagy hamis értéket. Megállapodás, hogy az igaz állításhoz az 1 számot, a hamishoz a 0 számot rendeljük hozzá. Ha egy logikai változót A-val jelölünk, $\bar{A}$ - alatt azt a logikai változót értjük, amelynek értéke mindig ellentétes A aktuális értékével. Ha $A = 1$, ebből következik, hogy $\bar{A} = 0$, és megfordítva.

Két logikai állítás között akkor van kapcsolat, ha az egyik logikai értéke meghatározza a másik változó logikai értékét. Ilyen értelemben logikai függvényekről is beszélhetünk, $B = f(A)$, de meg kell jegyeznünk, hogy már egyváltozós esetben is lényeges a különbség a közönséges függvényekkel szemben.

Míg a közönséges egyváltozós függvények száma végtelen, addig az egyváltozós logikai függvények száma véges: 4.

A $B = f(A)$ logikai függvény lehetséges 4 értékét az alábbi táblázatban mutatjuk be:

A		0	1
B =	konstans 0	0	0
	$B = A$	0	1
	$B = \bar{A}$		
	tagadás, negáció	1	0
	konstans 1	1	1

A $z = f(x, y)$ alaku kétváltozós logikai függvényeknek argumentuma négyféle lehet, a független változók a következő lehetséges párokat alkotják:

00, 01, 10, 11. A kétváltozós logikai függvények száma 16, ugyanis minden kétváltozós logikai függvény 0 vagy 1 értéket vehet fel ezekre, összesen tehát

$2^4 = 16$ különböző kétváltozós logikai függvény létezik.

A 6. ábrában 0-tól 15-ig indexelve feltüntettük a 16 féle logikai függvényt. A logikai függvények mindig felírhatók negációk, diszjunkciók és konjunkciók segítségével, többváltozós esetben is, de az ilyen kifejezés sok változó esetén, bonyolult függvélynél igen hosszú lesz.

Kétváltozós logikai függvények

$$z = f(x, y)$$

x	1	0	1	0	Megnevezés és szimbólum	Előállítás negáció, konjunkció és diszjunkció segítségével
y	1	1	0	0		
z_0	0	0	0	0	Konstans 0	0
z_1	0	0	0	1		$\bar{x} \wedge \bar{y}$
z_2	0	0	1	0		$x \wedge \bar{y}$
z_3	0	0	1	1		$\bar{y}$
z_4	0	1	0	0		$\bar{x} \wedge y$
z_5	0	1	0	1		$\bar{x}$
z_6	0	1	1	0	Antivalencia $\neq$	$(x \wedge \bar{y}) \vee (\bar{x} \wedge y)$
z_7	0	1	1	1		$\bar{x} \vee \bar{y}$
z_8	1	0	0	0	Konjunkció $\wedge$	$x \wedge y$
z_9	1	0	0	1	Ekvivalencia $\equiv$	$(\bar{x} \wedge \bar{y}) \vee (x \wedge y)$
z_{10}	1	0	1	0		x
z_{11}	1	0	1	1		$x \vee \bar{y}$
z_{12}	1	1	0	0		y
z_{13}	1	1	0	1		$\bar{x} \vee y$
z_{14}	1	1	1	0	Diszjunkció $\vee$	$x \vee y$
z_{15}	1	1	1	1	Konstans 1	1

6. ábra

Az n változós logikai függvényeknél a lehetséges argumentumok száma 2^n , a lehetséges logikai függvények száma 2^{2^n} .

A továbbiakban az automatikus számítógép felhasználói számára gyakorlatilag legfontosabb logikai függvényeket és a logikai változókkal való formális számolás elemeit ismertetjük.

Diszjunkció (logikai összeadás)

Két logikai változó diszjunkcióján azt a függvénykapcsolatot értjük (jele: $A \vee B$), amelynek logikai értéke akkor 1, ha a két változó közül legalább az egyik logikai érték 1.

A diszjunkció számolás szabályai:

$$0 \vee 0 = 0$$

$$0 \vee 1 = 1$$

$$1 \vee 0 = 1$$

$$1 \vee 1 = 1$$

Amint látjuk, a logikai összeadás szabályai eltérnek az algebrai összeadás szabályaitól.

Konjunkció (logikai szorzás)

Két logikai változó konjunkcióján azt a függvénykapcsolatot értjük (jele: $A \wedge B$), amelynek értéke akkor és csak akkor 1, ha mind A, mind B logikai értéke 1.

A konjunkció számolási szabályai:

$$0 \wedge 0 = 0$$

$$0 \wedge 1 = 0$$

$$1 \wedge 0 = 0$$

$$1 \wedge 1 = 1$$

Mind a konjunkcióra, mind a diszjunkcióra érvényes a kommutativitás (felcserélhetőség) szabálya:

$$A \wedge B = B \wedge A$$

$$A \vee B = B \vee A$$

Fennáll az asszociativitás is:

$$(A \wedge B) \wedge C = A \wedge (B \wedge C) = A \wedge B \wedge C$$

$$(A \vee B) \vee C = A \vee (B \vee C) = A \vee B \vee C$$

Érvényes továbbá a disztributivitás szabálya is:

$$A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$$

$$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$$

Hasznos lehet az alábbi egyszerű, közvetlenül belátható szabályok ismerete is:

$$A \wedge 0 = 0$$

$$A \wedge 1 = A$$

$$A \wedge A = A$$

$$A \wedge \bar{A} = 0$$

$$A \vee 0 = A$$

$$A \vee 1 = 1$$

$$A \vee A = A$$

$$A \vee \bar{A} = 1$$

Bizonyítható és un. "igazságtáblázattal" közvetlenül is belátható a negálási szabály konjunkcióra és diszjunkcióra:

$$(\overline{A \wedge B}) = \bar{A} \vee \bar{B}$$

$$(\overline{A \vee B}) = \bar{A} \wedge \bar{B}$$

Ezekből az összefüggésekből következik, hogy ha egy áramkörnek megfelelő logikai kifejezést felírtuk, az ellentétes működésű áramkört úgy kapjuk meg, ha a felírt logikai kifejezés minden változója helyett annak negáltját vesszük és a műveleti jeleket is megfordítjuk. (Logikai összeadás helyett logikai szorzást írunk és viszont.)

A számítógépek felépítésében és gyakran a programozásban is szerepet játszik az ekvivalencia és az antivalencia függvény.

Az ekvivalencia függvény értéke akkor 1, ha mindkét változó azonos értékű, pl. egyszerre 0, vagy egyszerre 1, ($X = Y = 0$ vagy $X = Y = 1$), különben zérus.

Az antivalencia függvény ennek ellentéte olyan értelemben, hogy értéke akkor 1, ha a változók értéke nem egyenlő ($X = 0$ és ugyanakkor $Y = 1$ vagy $X = 1$ és ugyanakkor $Y = 0$), különben pedig zérus.

Az elektronikus számítógépekben a logikai változóknak az ábrázolása két állapotú elemekkel történhetik, a változó két lehetséges értékének megfelelően. Az 1-et meghatározott polaritású és nagyságú elektromos feszültség jelenléte, a 0-t ennek hiánya jelzi. Ennek az elvnek alapján például egy "és" áramkör olyan berendezés, amelyiknek két vagy több bemenő vezetéke és egy kimenő vezetéke van és amely a kimenő vezetéken akkor és csak akkor ad feszültséget, ha valamennyi bemenő vezetéken van feszültség. Az "és" áramkörnek mint logikai függvény, a konjunkció felel meg. A "vagy" áramkör akkor ad kimenő feszültséget, ha a bemenő vezetékek közül bármelyikén van feszültség. Itt tehát a diszjunkció realizálódik. Végül a "nem" vagy negáló áramkör akkor és csak akkor ad kimenő feszültséget, ha a bemenetén nincs jelen feszültség. Az utóbbi az egyváltozós negáció függvénynek felel meg.

A számítógépek gépi utasításai között általában szerepel a logikai szorzás művelete. Ezt a műveletet alkalmazzuk olyankor, ha a számot ábrázoló bitekből adott pozícióban levőket le akarjuk választani.

Legyen például az N szám bináris alakja egy 12 bites memóriában a következő:

1	1	0	1	0	1	1	0	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---

Ha valamilyen okból pl. a középső 4 biten ábrázolt részletre van szükségünk, ezt úgy tudjuk kiemelni a teljes számból, ha logikai szorzást végzünk az N szám és az

$$A =$$

0	0	0	0	1	1	1	1	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---

alaku szám között. Eredményképpen csak azokon a helyeken kapunk egyest, ahol A-ban és N-ben is egyes állt, tehát az eredmény:

$$N \wedge A =$$

0	0	0	0	0	1	1	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---

alaku lesz.

A gyakorlatban jól alkalmazhatjuk ezt a lehetőséget olyan esetekben, amikor például két vagy több számot tároltunk egy rekeszben, és ezeket szét akarjuk választani. Nagy adattömegnél és viszonylag kis fixpontos számoknál futási idő tekintetében sokkal előnyösebb, ha pl. valamilyen szempontból összetartozó adatokat egy rekeszben tárolunk és így a teljes adattömeg elfér az operatív memóriában, mintha helyhiány miatt a háttérmemóriát is igénybe kellene vennünk és az adatokat állandóan ki- és be mozgathánánk az operatív memória és a háttérmemória között. Ha a memória szóhosszúsága elég nagy, gyakran tárolnak 3-4 viszonylag kis fixpontosan ábrázolt számot is egy-egy rekeszben.

Ugyancsak logikai szorzás segítségével lehet elérni azt, hogy bizonyos különböző számítások után keletkezett számokat azonosnak tekintsen a számítógép, ha azok csak a mantissza néhány utolsó bitjében térnek el egymástól. Elképzelhető, hogy ezek a számok elvileg egyeznek, s az eltérést a különböző számítási eljárásokból eredő hibák, esetleg az egymástól különböző műveletekből következő különböző kerekítési hibák okozzák.

Ha ez a feladat, a megvizsgálandó számokat olyan számmal szorozzuk meg logikailag, amelyben az elhanyagolható bitek helyén (tehát a mantissza néhány utolsó bitjén, - ezek számának megállapítása természetesen a programozó feladata) zérus áll, különben pedig csupa egyes. Így a számok "érték-telen" és esetleg zavaró bitjei eltűnnek.

Itt említjük meg, hogy a számítógépben két lebegőpontos szám csak abban az esetben egyezik meg egymással feltétlenül, ha ezek beolvasott (és nem számított!) értékek. A kerekítési hibák halmozódása miatt nem lehet elvárni, hogy két lebegőpontos szám pontosan azonosnak adódjék, valamennyi bitje meg egyezzen. Így nem kapunk pontosan 0-t akkor sem, ha elvileg biztosak vagyunk abban, hogy egy rekeszben zérusnak kell képződnie. Általában 10^{-9} -et már

zérusnak tekintjük, de akkor járunk el helyesen, ha az adatok nagyságrendjéhez viszonyítva állapítjuk meg a megengedhető hibahatárt.

Az un. bitenként való műveletek által általában idő- és helymegtakarítást lehet elérni olyan esetekben is, amikor közönséges aritmetikai műveletekkel is meg tudtuk volna oldani a feladatot, sokszor pedig közönséges aritmetikai műveletek által meg sem oldható a probléma.

Tekintsük például a kettővel való szorzás és osztás műveletét. A decimális 6 bináris fixpontos ábrázolása:

..... 00110

Kettővel megszorozva

.....001100 adódik, azaz egy hellyel balra tolódott a szorzandó. 2^k -val való szorzásnál k pozícióval balra kell tolnunk a számot úgy, hogy jobbról zérusok lépnek be a bitekre; az előjelbit változatlan marad.

2^k -val való osztásnál k pozícióval jobbra toljuk az osztandót. A számítógépek utasításrendszerében léteznek ilyen rendeltetésű utasítások (un. aritmetikai eltolás), és sokkal gyorsabban szorzunk vagy osztunk így, mintha egy másik rekeszben elhelyezett operandus segítségével hajtottuk volna végre a szorzást, vagy osztást.

Meg kell említeni a logikai eltolást (balra vagy jobbra) ami abban különbözik az aritmetikai eltolástól, hogy az előjelbit is eltolódik. A szabaddá vált helyekre 0-k kerülnek be.

Nyilvánvaló, hogy pl. pozitív szám balra tolásakor olyan esetben, amikor az előjelbit és a helyére kerülő bit különböző volt, a logikai eltolás utján nyert szám nem 2^k -szorosa az eredetinek, mégis elképzelhetők olyan feladatok, amikor erre a műveletre van szükség.

Végül megemlítjük a ciklikus eltolás műveletét, amely abban különbözik a logikai eltolástól, hogy azok a bitek, amelyek balra vagy jobbra kitolódtak (elvesztek), az ellenkező oldalon lépnek be.

6. GÉPI PROGRAMOZÁSI NYELVEK - AUTOMATIKUS PROGRAMOZÁSI NYELVEK

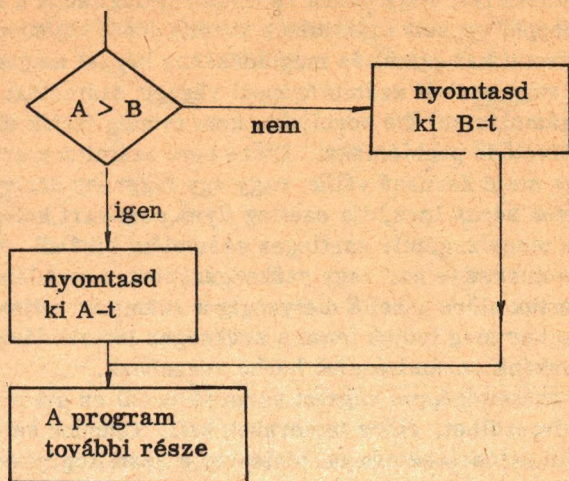
Egy adott speciális probléma gépi megoldására szolgáló utasítások összességét programnak nevezzük. A programozás tehát egy bizonyos program megszerkesztését, megírását jelenti, azzal a szükséges előkészítő munkával együtt, amely az utasítások részletes felírását megelőzi.

A programozás maga úgy tekinthető, mint az emberi (matematikai) nyelvről az adott gép "nyelvére" való fordítási művelet. Mivel a számítógép nyelve - akár a gép numerikus kódokból (számjegykombinációkból) álló saját nyelvéről, akár a matematikai formanyelvhez közelálló automatikus nyelvekről van szó - igen precíz, teljesen egyértelműen meghatározott, a programozónak nagy körültekintéssel kell eljárnia. Ezért azt ajánljuk, hogy mielőtt a tulajdonképpeni programíráshoz látna, előbb kísérelje meg újra meg újra teljes mélységben átgondolni a problémát. A matematikai modell megállapítása, azaz a feladatra jellemző lényeges összefüggések lerögzítése után a megoldási módszer meghatározása következik. Ha a módszerhez tartozó algoritmus (az elvégzendő műveletek egymásutánja) kiválasztása is megtörtént, célszerű az algoritmus főbb lépéseit un. blokkdiagramban lerajzolni.

A blokkdiagram az algoritmusok ábrázolásának elterjedt módja. Mint már említettük, a számítógép önállóan semmit sem hajt végre, utasítások alapján is csak aritmetikai műveletekre és ezeken kívül arra képes, hogy két jól megkülönböztethető állapot alapján egyik vagy másik alternatíva mellett döntsön és azt végrehajtsa. Ezért, mint látni fogjuk, a blokkdiagram jól megfelel a számítógép "gondolkodásmódjának".

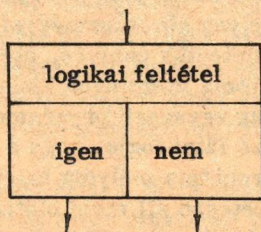
A blokkdiagramban az egyes műveleteket vagy műveletcsoportokat téglalapokkal, a feltételektől függő elágazásokat csucsára állított rombuszokkal (esetleg egyéb ábrával pl. ellipszissel) ábrázoljuk, és az egyes dobozokat nyílakkal kötjük össze.

Két szám közül a nagyobbik kiválasztásának és kinyomtatásának algoritmusát az alábbi blokkdiagrammal rajzolhatjuk le:



7. ábra

A téglalap alaku blokkokat aritmetikai, a rombusz alakuakat logikai blokkoknak nevezik. Minden blokknak egy bemenete van. Az aritmetikai blokknak egy, a logikai blokknak két kimenete van. Logikai blokkot szokás így is ábrázolni:



Természetesen nemcsak ilyen elemi algoritmusok vannak. Algoritmusok a lineáris egyenletrendszerek, a magasabb fokú algebrai egyenletek, a szállítási feladatok megoldási módszerei stb. Algoritmusok szolgálhatnak bizonyos kérdéseknek pl. telepítési, beruházási, termelési problémáknak az eldöntésére.

Ezek a feladatok mind valamilyen ún. matematikai programozási módszer szigorú algoritmusára vezethetők vissza. Tulajdonképpen egy probléma algoritmizációja azt jelenti, hogy a megoldási módszert teljesen tudatossá tesszük a korábbi ösztönös "intuitív" módszerek, tervezések helyett.

Akkor, amikor valamely probléma megoldása közben különböző alternatívák merülhetnek fel és bizonyos feltételek vizsgálata alapján döntünk arról, hogy a továbbiakban melyik utat kövessük, tulajdonképpen magunk is egy blokkdiagram néha igen bonyolult utjait járjuk végig. A kézi számolás során hosszú

szakaszok követhetik egymást anélkül, hogy döntenünk kellene bármilyen kérdésben, ilyenkor az egymást sorra követő lépések mintha egy belsőleg megfogalmazott "program" aritmetikai blokkjainak felelnének meg. Ezt a kissé erőszakolt példát azért hoztuk fel, hogy ennek kapcsán rávilágítsunk a kézzel és az automatikus számítógéppel végzett számítások közötti döntő különbségre.

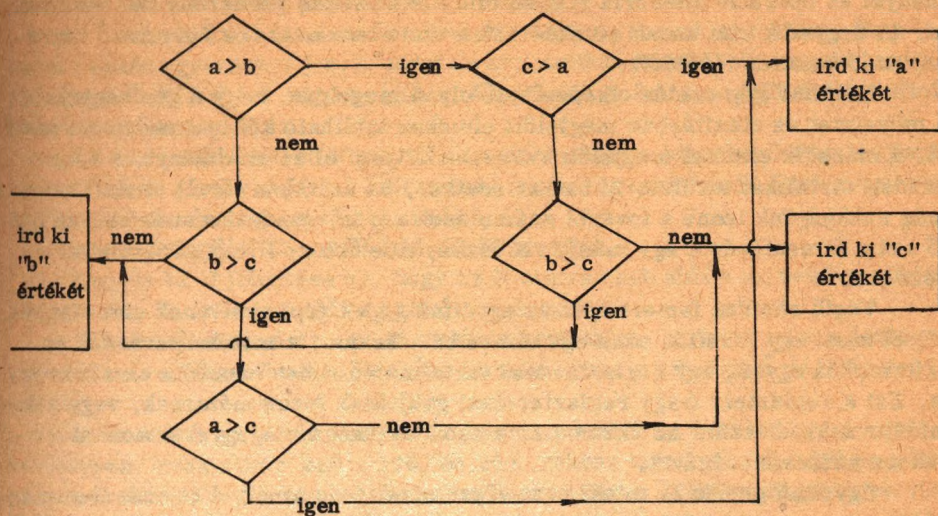
Amikor egy matematikai számítás megoldásához kézzel magunk látunk hozzá (a logarléccel vagy asztali számológéppel végzett számolást ebből a szempontból a kézi számolás körébe soroljuk), nem mindig látjuk előre a megoldás során esetleg felvetődő problémákat. Előre nem számítunk arra, hogy pl. valamely tört nevezője majd zérussá válik vagy egy függvény deriváltja egy számunkra fontos helyen zérus lesz. Ha esetleg ilyen nem várt helyzet tárul elénk, módunkban van megvizsgálni: esetleges számolási hibából vagy helytelen adatfelvételből következett-e az, vagy szükségszerűen a probléma velejárója, csak mi nem ismertük előre a kellő mélységig a számítás teljes menetét. Kézi számolásnál ilyenkor meg tudjuk tenni a szükséges intézkedéseket anélkül, hogy addig végzett munkánk, számításaink kárba vesznének.

Az automatikus számítógéppel végzett számításoknál ez nincsen így. Minden kérdést, amely felmerülhet, előre ismernünk kell. Tudnunk kell, hogy különböző extrém adatválasztások esetén mi történhet a számítógépben, és az esetleges kritikus helyzetekre vonatkozóan - feltételeken - megfelelő utasításokat kell adnunk. Ha ezt elmulasztjuk, a jobbik eset az, ha a számítógép valamilyen hibajelzéssel tudunkra adja, hogy számára megoldhatatlan helyzet elé ért, de bekövetkezhet az is, hogy a gép nem jelez hibát, érvényes utasítás birtokában "jogosan" végrehajtja a következő lépéseket, és a hiba csak jóval később, más formában, esetleg egyszerűen rossz eredmény alakjában fog jelentkezni. Még az is előnyösebb, ha az eredmény teljesen abszurd - hiszen azonnal keresni fogjuk a hibát, - mintha a számok nagyjából megfelelnek a várakozásnak. Az utóbbi esetnek igen veszélyes következményei lehetnek.

Mindezeket azért részleteztük ennyire, mert sok programozó esik abba a hibába, hogy amint a probléma megoldása kialakult benne, azonnal programíráshoz kezd. Még viszonylag egyszerű programok esetén is ajánljuk a blokkdiagram írást. A programozó látni fogja, hogy a diagramkészítés mennyire segíti majd abban, hogy a probléma mélyéig hatoljon, és az egész számítás menete áttekinthetően, szemléletesen áll majd előtte. Az esetleges programhibák megkeresésében is segítségére lesz, és a hibajavításnak a program többi részére való kihatását is megmutathatja.

Példaképpen bemutatunk egy olyan blokkdiagramot, amelynek segítségével három különböző szám közül a nagyság szerinti középső kiválasztására készíthetünk programot. Legyen a három szám: a , b és c . A feladat a középső mennyiség értékének kiírása.

Természetesen többféleképpen is megoldható a feladat.



8. ábra

Amint látjuk, már egy ilyen viszonylag egyszerű esetben is több logikai döntésre kerül sor, és valahányszor megvizsgálunk egy-egy relációt, a döntéstől függően mindkét esetben egy-egy pontosan meghatározott lépést kell tennünk.

Általában nagyobb feladatokra különböző mértékben részletezett blokkdiagramokat készíthetünk. Minél részletesebb a blokkdiagram, annál könnyebb lesz maga a programírás, vagyis az utasítások összeállítása. Az a véleményünk, hogy az aritmetikai blokkok összevonása egy-egy nagyobb egységgé inkább megengedhető, de valamennyi logikai blokk szerepeljen a sémában. Az ábrázolt blokkok mélysége és részletessége nagymértékben függ a feladat bonyolultságától.

Ha már előttünk áll a szemléletes blokkdiagram, a gyakorlott programozó a gép utasításrendszerének (vagy másképp: belső kódjának) ismeretében un. gépi kódban is megírhatja a program utasításait. A program gépi utasítások sorozata, ez az a nyelv, amelyet a számítógép megért.

A gépi utasítások számok, amelyek általában két fő részből állnak: az elvégzendő művelet kódjából és hivatkozásokból a memória rekeszeire. Aszerint, hogy egy utasítás az operatív memória egy, két vagy három rekeszére hivatkozik, megkülönböztetünk egy, két vagy háromcímű utasításrendszereket, és magát a számítógépet is egy-, két-, három- (esetleg négy -) című gépnek nevezzük. Háromcímű gép esetén pl. az első két cím azt jelzi, hogy milyen sorszámú rekeszek tartalmával kell a műveleti jel kódjának megfelelő arit-

metikai műveletet elvégezni, a harmadik cím pedig azt, hogy a művelet eredményét az operatív memória milyen című (sorszámú) rekeszébe kell elhelyezni. (A negyedik cím annak a rekesznek a címe lenne, ahol a következő végrehajtandó utasítás tárolódik.)

Kétcímű gép esetén elképzelhető olyan megoldás, hogy a gép végrehajtja a műveletet az utasításban megjelölt címeken található két operandussal, majd pl. a második címmel megjelölt rekeszbe helyezi el az eredményt, a rekesz eredeti tartalmát törölve. Abban az esetben, ha az abban tárolt eredeti számra még szükségünk lenne a további számolások során, gondoskodnunk kell az illető szám átmentéséről egy másik rekeszbe, mielőbb az illető műveletet elvégeznénk.

Végül röviden ismertetjük az egycímű gépek gépi nyelvének elvi alapjait. Mivel most egy utasítás csak egy címre hivatkozhat, a másik operandus az aritmetikai egység egy regiszterének tartalma, és ebben képződik az eredmény is. Ezt a regisztert (vagy regisztereket) gyűjtőnek (szummátornak, vagy akkumulátornak) nevezik. Az akkumulátor szóhosszusága megegyezik a memória-rekesz szóhosszuságával.

Egycímű gépeknek mindig van olyan utasítása, amely a címrészben megjelölt rekesz tartalmát beírja az akkumulátorba, törölve a régi tartalmat. Már említettük, hogy a rekesz eredeti tartalma a rekeszből való kiolvasáskor változatlanul megmarad. Ugyancsak kell olyan utasításnak is lennie, amely az akkumulátorban képződött eredményt a címben megjelölt rekeszbe viszi ki.

Ha a műveleti kódot egy-egy kétjegyű szám jelöli, az a program, amellyel pl. a 100, a 200 és a 300 című rekesz tartalmának összegét a 400 című rekeszbe akarjuk elhelyezni, egy fiktív gépi nyelven például így képzelhető el:

10	100	A 10 kódszámú utasítás a 100 cím tartalmát az akkumulátorba hozza
15	200	Az akkumulátor régi tartalmához hozzáadja a 200 című rekesz tartalmát
15	300	Az akkumulátor régi tartalmához hozzáadja a 300 című rekesz tartalmát
20	400	Az eredményt kiviszi 400-ba

Már ezen az egyszerű példán is láthatjuk a gépi programnyelv használatával járó nehézségeket. Az egyes műveletek kódját viszonylag gyorsan megtanuljuk, megszokjuk, de állandó koncentrációt jelent annak figyelése, hogy a különböző adatokat milyen címen találhatjuk meg.

A gépi utasítások között fontos szerepet játszanak azok, amelyeknek nem aritmetikai művelet, hanem az a hatásuk, hogy eltérést hoznak létre a műveletek végrehajtásának "természetes" sorrendjétől. Ezeket az utasításokat vezérlésátadó (ugró, vagy ugrató) utasításoknak nevezik. (Lásd a 3. fejezetben, a vezérlőegység működésének ismertetését.)

A vezérlésátadás feltétlen vagy feltételes lehet. Az első esetben mindig

létrejön az utasítások természetes sorrendjétől való eltérés, a másodikban az utasítás hatása valamely logikai feltételtől függ.

A vezérlésátadó utasítások műveleti kódja ugyancsak egy szám. Ez a szám szintén ugyanannyi jegyű, mint az aritmetikai utasítások, vagyis annyi számjegyből áll, ahány jegyet az utasítást jelentő számban a műveletek definiálására rögzítettek.

Jelentse például a 40 N utasítás az alábbiakat: a következő végrehajtandó utasítást az N című rekeszből kell venni, (vagy a programozók által megszokott fogalmazásban: "ugorj a 40. címre!")

Ez tehát egy feltétlen vezérlésátadó utasítás.

Legyen 41 N jelentése az, hogy az N címre csak abban az esetben történjék ugratás, ha az akkumulátor tartalma zérus.

Ellenkező esetben az ugrató utasítás hatástalan, s ekkor a természetes sorrendben, a soronkövetkező rekesz tartalma lesz a következő végrehajtandó utasítás.

Módosítsuk az előbbi kis példát (három mennyiség összeadása) a következőképpen: csak akkor kell a 100. és 200. rekesz tartalma összegéhez a 300. rekesz tartalmát is hozzáadni, ha ez az összeg zérus. Fiktív programnyelvünkön ezt így lehet megoldani:

(10)	10	100
(11)	15	200
(12)	41	88
(13)	a program további utasításai	

(87)	40	87
(88)	15	300
(89)	40	13

A bal oldalon, zárójelbe tett számok azt jelzik, hogy az illető utasítások milyen sorszámú rekeszekben tárolódnak. Az eddig elmondottak alapján a programrészlet, úgy véljük, nem szorul további magyarázatra, kivéve a 87. címen tárolt utasítást.

Ebben feltétlen ugratás történik újra meg újra ugyanarra a rekeszre: a számítógép ún. dinamikus stop-ot hajt végre. Így fejeződik majd be a program futása. Ilyen utasítás után hiába írunk bármit, azt a program már nem hajtja végre, ha csak nem ugrató utasítással jut el esetleg oda.

A különböző géptípusoknál a feltételes vezérlésátadó utasítások általában nagy választékban állnak a programozó rendelkezésére: ugratás történhet pl. akkor, ha az akkumulátor aktuális tartalma nem zérus, - ha pozitív, - ha negatív.

Ha a számítógép központi egységében több olyan regiszter van, amely az akkumulátorhoz hasonlóan fontos szerepet tölt be az utasítások végrehajtása közben, akkor a gépi utasításrendszer általában tartalmaz olyan utasításokat

is, amelyeknél a feltételes vezérlésátadás ezen regiszterek aktuális tartalmától függ.

Amint látjuk, gépi nyelven történő programozásnál nemcsak azt kell állandóan szem előtt tartanunk, hogy az adatok, amelyekkel számolunk, hol tárolódnak, hanem még az utasítások helyét (címét!) is.

A korszerű elektronikus számítógépeknél a programozás feladata lényegesen leegyszerűsödött. Nem kell a feladatokat a gép saját nyelvén megadni, hanem olyan közvetítő nyelveket (automatikus programozási nyelveket) használhatunk, amelyek nagyon közel állnak az eredeti matematikai vagy ügyviteli írásmódhoz. Az ilyen nyelveken írott programokat a számítógép csak úgy érti meg, ha megfelelő szótár: fordítóprogram áll a rendelkezésére. Egyre több olyan gép van, amelyhez a tervezők nem is egy, hanem több automatikus programozási nyelvhez szolgáló fordítóprogramot mellékelnek.

Az automatikus programozási nyelvek közös sajátossága az, hogy a rekeszek tartalmára nem a rekeszek címével, hanem alfanumerikus azonosítók segítségével hivatkoznak. A műveleti kódokban sem kell számokat szerepeltetnünk, hanem a megszokott matematikai írásjeleket használhatjuk. A vezérlés-átadó utasításokban sem valódi rekeszcímek állnak, hanem ún. címkék, helymegjelölő kifejezések. Ugyanezeket a címkéket (autokódban "referenciaszám"-nak nevezzük ezeket) kiírjuk azok elé az utasítások elé, amelyekre az ugratás történik, így a program elágazásai is szembeötlőek lesznek.

Ha el akarjuk érni, hogy egy elektronikus számítógép képleteket megértsen, gondoskodnunk kell arról, hogy azokat be lehessen vinni a gépbe. Az olyan gépet, amelynek számokon kívül más jeleket is be lehet adni (mindenekelőtt betűket) alfanumerikus beviteli rendszerűnek nevezzük. Az alfanumerikus kiviteli rendszer arra szolgál, hogy a számítások eredményeit a gép tetszőleges és áttekinthető alakban nyomtassa ki, továbbá, hogy bizonyos esetekben különféle közléseket írthasson ki a kezelőnek. Ezenkívül még elegendően nagy operatív memóriát és műveleti sebességet kívánunk meg attól a géptől, amelyre automatikus programozási nyelven akarunk programot készíteni.

Ezek a műszaki alapfeltételek a korszerű számítógépeken biztosítva vannak. Ma már tisztán műszaki uton is sok olyan feladat elvégzését (például magának a bevitelnek a megoldását) veszi át a számítógép, amely a régebbi típusú gépeknél a programozót terhelte. Ha műszaki uton, a gépbe beépített elemek segítségével történik a programozás megkönnyítése, akkor ez ún. hardware segítségével történik. Ezzel szemben "software"-nek nevezzük azoknak a programoknak az összességét, amelyek a programozás automatizálását eredményezik. A "software" szintén szervesen az illető géptípushoz tartozik. A software legfontosabb részei a különböző fordítóprogramok, de ide tartoznak a leghasználatosabb matematikai programok (pl. lineáris egyenletrendszer, magasabb fokú algebrai egyenleteket megoldó programok, korreláció, regresszió számítás), ökonometria programok (szállítási feladatok, lineáris programozás) és a különféle tiszt-, hibafelderítő- és hibajavító programok is.

Az automatikus programozási nyelvről a gépi nyelvre történő fordítást - a fordítóprogram alapján - maga a gép végzi, automatikusan elvégezve az

utasításokban a címzést, a számok bevitelére és kiírására szolgáló programrészletek beillesztését stb.

Az automatikus programozási nyelvekben különféle függvényutasítások (abszolútérték, négyzetgyökvonás, entier-függvény, trigonometrikus függvények és ezek inverzei stb.) egy-egy utasítással adhatók meg. Amikor a fordítóprogram ezeket értelmezi, úgy fordítja le gépi nyelvre, hogy az egyetlen függvényutasítás helyett egy egész sor gépi utasítást (valamilyen közelítő számítás algoritmusát végrehajtó utasítássort) helyez a program megfelelő helyére.

Az automatikus programozási nyelvek utasításaiban az utasítások címét alfanumerikus jelcsoportok alkotják, biztosítva a számoknál könnyebb azonosítást. Különböző aritmetikai és egyéb műveletek jelölésére különféle jelek vannak definiálva. Vezérlésátadó utasításokban helymegjelölő kifejezések útján jelezzük, hogy melyik utasításra történjék ugrás. Ezek a helymegjelölő kifejezések (cimkék) az automatikus nyelv szabályai szerinti számok, esetleg olyan alfanumerikus jelcsoportok lehetnek, amelyeket operandus jelölésére nem használunk a programban.

Az un. deklarációs utasításokban a programutasítások beolvastatását megelőzően közölnünk kell a fordítóprogrammal, milyen alfanumerikus jeleket fogunk alkalmazni operandusként (ezeket nevezzük változóknak), illetve címkeként. Egyes automatikus nyelvekben ugyancsak a deklarációs részben kell felsorolni azokat a függvényutasításokat, amelyeket a program a fordítóprogram révén a rendelkezésre álló standard rutinok alapján fog majd használni.

Ismerni kell, és pontosan kell írni azt a néhány - többnyire az angol nyelvből átvett - un. alapszót, amelyek használata az illető automatikus nyelvben megengedett. Ezek az alapszavak teszik a program utasításait a beszélt nyelvhez hasonlóvá, viszonylag könnyen érthetővé.

A változókat (azaz az operandusokat) valamennyi automatikus nyelvben indexszel lehet ellátni, s így mód van a tömbök (esetleg többdimenziós tömbök) könnyű kezelésére. Általában pontos és részletes szabályok írják elő a programírás formátumát: mi jelzi az utasítás végét, kis- vagy nagybetűket szabad-e, illetve lehet-e használni stb.

A különböző automatikus programozási nyelvek különböző igényeket óhajtának kielégíteni. Ma már több problémára orientált nyelvet készítenek világszerte a programozás megkönnyítésére. Azt, hogy egy automatikus nyelv mikor "jó" vagy "könnyű", nehéz definiálni. A programozó számára általában előnyös, ha az automatikus nyelv a lehető legjobban hasonlít a matematikai kifejezésmóddhoz, illetve a beszélt nyelvhez, pl. ha szövegek és alfanumerikus jelek helyezhetők el a memóriában, és ezeket vissza is lehet nyomtatni, ha a programozó minél több standard függvényt tud használni, anélkül, hogy ezek programját neki kellene megírnia stb. Mindezek azonban nagy megterhelés elé állítják a fordítóprogramot. Ha az automatikus nyelv nagyon fejlett, a fordítóprogram igen sok helyet fog elfoglalni a memóriában, s így kevesebb rekesz marad adattárolásra, továbbá a fordítás is hosszabb időt vesz majd igénybe.

Külön kérdés az, hogy célszerű-e (és ha igen, milyen mértékben) szintaktikai (formai) és esetleges szemantikai (tartalmi) hibákat jelző részleteket

beépíteni a fordítóprogramba. Az automatikus nyelvek készítői általában igyekeztek az optimális megoldást megtalálni.

Köztudomásu, hogy az igen fejlett és a legnehezebb műszaki- tudományos problémákra talán leginkább alkalmas Algol nyelven írt programokat a számítógépek viszonylag lassan fordítják le. Azok, akik megtanulták és megszokták az Algol nyelvet, szívesen vállalják a lassu fordítás kényelmetlenségét is az Algol nyelv tömörsége és kényelmes programozási lehetősége miatt, más programozóknak viszont mások a fő szempontjai.

Az EMG 830 autokódja a kialakult hazai igények felmérése és nagyrészt az Elliott gépek autokódjánál szerzett gyakorlat hasznosítása alapján készült.

IRODALOMJEGYZÉK

1. Handbook of Automation and Control I-II (New York. 1958 Wiley)
2. V. Wilkes - D.J. Wheeler - S. Gill:
The Preparation of Programs for an Electronic Digital Computer
(Cambridge, Mass. 1951 Adison-Wesley)
3. Békéssy: Programvezérlésű digitális számológépek programozása
Tankönyvkiadó. Bp. 1961
4. Ferentzy - Krepuska: Autokód programozás.
(A National - Elliott 803 B programvezérelt digitális elektronikus számológép A 103 jelzésű autokódja.) NIMIGÜSZI. Bp. 1964. (Jegyzet)
5. Programok és algoritmusok. MTKI, Bp. 1966.
6. A GIER ALGOL III. Kézikönyve. VEGYTERV. Bp. 1966. (Jegyzet)
7. Gnegenko, etc.: Bevezetés a programozásba. I-II Műszaki Kiadó Bp. 1964.
8. Kítov - Krinyickij: Elektronikus digitális számológépek és a programozás. Akadémia Kiadó Bp. 1964.
9. Elliot - 803 programkönyvtár, I-II (Nyersfordítás) NIMIGÜSZI Bp. 1967. (Jegyzet)
10. Krepuska: Számolás elektronikus számológéppel.
Tankönyvkiadó Bp. 1968. (Egyetemi Jegyzet)
11. Krepuska - Havas: Elektronikus számológépek és vegyészeti alkalmazásuk. NIMIGÜSZI. Bp. 1967. (Jegyzet)
12. NIMIGÜSZI által készített programok.
NIMIGÜSZI Bp. 1967. (Jegyzet)

7. AZ EMG 830 AUTOKÓD UTASÍTÁSRENDSZERE

A következőkben röviden összefoglaljuk az EMG 830-as számítógépre készült autokód fordítóprogram által megengedett utasításokat, illetve utasítás részeket.

1. Operandusok:
 - 1.1 konstansok
 - 1.2 változók
2. Referencia-számok
3. Műveleti jelek:
 - 3.1 aritmetikai
 - 3.2 logikai és léptetési
4. Deklarációs utasítások:
 - 4.1 Fixpontos
 - 4.2 Lebegőpontos
 - 4.3 Duplafixpontos
 - 4.4 Függvény
 - 4.5 Referenciaszám
5. Alapszavak:
 - 5.1 JUMP
 - 5.2 JUMP IF
 - 5.3 JUMP UNLESS
 - 5.4 SUBR
 - 5.5 EXIT
 - 5.6 VARY
 - 5.7 CYCLE
 - 5.8 REPEAT
 - 5.9 INPUT
 - 5.11 PRINT
 - 5.12 OUTPUT
 - 5.13 CHECK
 - 5.14 WAIT
 - 5.15 STOP
 - 5.16 LINE
 - 5.17 LINES
 - 5.18 SPACE
 - 5.19 SPACES
 - 5.20 TITLE

- 5.21 READER
- 5.22 PUNCH
- 5.23 SET
- 5.24 START

6. Függvények:

- 6.1 SIN
- 6.2 COS
- 6.3 TAN
- 6.4 ARCTAN
- 6.5 LOG
- 6.6 EXP
- 6.7 SQRT
- 6.8 FRAC
- 6.9 STAND
- 6.10 INT
- 6.11 MOD

7. Aritmetikai utasítások

8. Logikai és léptetési utasítások

9. Precedencia szabályok:

- 9.1 aritmetikai utasításban
- 9.2 logikai és léptetési utasításban

10. Változók indexel:

- 10.1 Indexnélküli változók
- 10.2 Egydimenziós indexszel ellátott változók
- 10.3 Kétdimenziós indexszel ellátott változók

11. Beolvasó utasítások:

- 11.1 READ
- 11.2 INPUT

12. Kinyomtató utasítások:

- 12.1 PRINT
- 12.2 OUTPUT
- 12.3 TITLE

13. Nyomkövetési utasítások:

- 13.1 CHECK
- 13.2 Referenciaszám nyomtatás

14. Periféria egység kijelölő utasítások:

- 14.1 READER
- 14.2 PUNCH

15. Vezérlés átadási utasítások:

- 15.1 Feltétel nélküli ugrást előíró utasítások:
 - JUMP
 - SUBR
 - EXIT

- 15.2 Feltétel teljesülése esetén ugrást előidéző utasítások JUMP IF
JUMP UNLESS
- 15.3 Ugrás az adatsz'alagra lyukasztott "trigger" hatására
- 16. Ciklusképző utasítások:
 - 16.1 1-típusu CYCLE
 - 16.2 2-típusu CYCLE
 - 16.3 VARY
 - 16.4 REPEAT
- 17. Programfutas megállító utasítások:
 - 17.1 WAIT
 - 17.2 STOP
- 18. Formátum vezérlő utasítások:
 - 18.1 LINE
 - 18.2 LINES
 - 18.3 SPACE
 - 18.4 SPACES
- 19. Kezdeti értékadó utasítás: SET
- 20. Programforditást befejező utasítás: START
- 21. A programírás formátuma.
- 22. Az adatsz'alag.

8. AZ EMG 830 AUTOKÓD UTASÍTÁSRENDSZERÉNEK RÉSZLETEZÉSE

1. Operandusok
 - 1.1 Programkonstansok
 - 1.1.1 Fixpontos konstans
Fixpontos konstans csak egész szám lehet.
Egy fixpontos konstans nem lehet nagyobb 2^{20} -nál és nem lehet kisebb -2^{20} -nál.
 - 1.1.2 Lebegőpontos konstans.
Lebegőpontos konstans egész szám, vagy egész résszel ellátott tizedestört lehet.
Pl. 156, -2548, 33.12, 0.14
(0.14 helyett . 14 nincs megengedve, ilyen alakú konstans használata a programban hibajelzést okoz.)
Lebegőpontos konstans nagysága $-1/2 \cdot 10^{614}$ és $1/2 \cdot 10^{614}$ közé esik.
 - 1.1.3 Duplafixpontos konstans.
Duplafixpontos konstans alakja csak egész szám lehet.
Pl. 214, 123456789
 - 1.1.4 Az autokód fordítóprogram fordítás közben automatikusan az utasítás típusának megfelelően ábrázolja a programkonstansokat. Az egyes típusokba a következő típusú számformákról képes konvertálni:

<u>Utasítás típusa</u>	<u>Beolvasott szám típusa</u>
Fixpontos	Fixpontos
Duplafixpontos	Fixpontos, duplafixpontos
Lebegőpontos	Fixpontos, duplafixpontos, lebegőpontos.

- 1.2 Program változók.
A program alapváltozóit az angol ABC 26 nagy és 26 kisbetűjével jelöljük. Az 52 alapváltozó egyike sincs kitüntetett helyzetben, valamennyit egy- vagy két dimenziós indexszel is el lehet látni. Valamennyi alapváltozót tetszőleges típusúnak lehet deklarálni, tehát a deklaráció típusától függően fixpontos, lebegőpontos, vagy duplafixpontos változó lehet.

A könnyebb írásmód kedvéért az utasításrendszer leírásánál a következő konvenciót vezetjük be:

A, B, C, D-vel jelöljük a duplafixpontos,

I, J, K, L, M, N-nel jelöljük a fixpontos,

X, Y, V, Z-vel jelöljük a lebegőpontos konstansokat.

A példák kidolgozásánál ezt a konvenciót nem alkalmaztuk, ott mindig az aktuális programdeklaráció a mérvadó.

1.2.1 Fixpontos változó.

A fixpontos változókat az angol ABC olyan nagybetűivel és kisbetűivel jelöljük, melyek még nincsenek igénybe véve lebegőpontos, vagy duplafixpontos változók jelölésére.

1.2.2 Lebegőpontos változó.

A lebegőpontos változókat az angol ABC olyan nagybetűivel és kisbetűivel jelöljük, melyek még nincsenek igénybe véve fixpontos, vagy duplafixpontos változók jelölésére.

1.2.3 Duplafixpontos változó.

A duplafixpontos változókat az angol ABC olyan nagybetűivel és kisbetűivel jelöljük, melyek még nincsenek igénybe véve fixpontos, vagy lebegőpontos változók jelölésére.

1.2.4 Minden egyes deklarált

a) fixpontos változó a memória egy rekeszét

b) lebegőpontos és

c) duplafixpontos változó a memória két rekeszét foglalja el.

Ez azt jelenti, hogy ha $I[10]$ $A[3]$ és $X[15]$ van deklaráva ez $I[10]$ -nél 11 rekeszt, $A[3]$ -nál 8 rekeszt és $X[15]$ -nél 32 rekeszt jelent, vagyis $I[10]$, $A[3]$ és $X[15]$ együttes memóriai igénye 51 rekesz.

2. Referenciaszámok.

Az EMG 830-as autokód rendszer lehetőséget biztosít a programozók számára, hogy az autokód programok egyes utasításait megcímkezhessék.

2.1 Az EMG 830-as autokód rendszerben a címkék csak pozitív egész számok lehetnek. Minden egyes címkét a) (záró kerek zárójel) jel határol. Pl. 5), 10), ...

2.2 Egy programon belül egy adott referenciaszám csak egyszer fordulhat elő. Nem szükséges a referenciaszámokat nagyságszerinti sorrendben elhelyezni. Nem szükséges minden egyes deklarált referenciaszámot címkéként felhasználni.

2.3 A referenciaszámokat közvetlenül a program utasítás elé kell kitenni.

- 2.4 Egy programutasítás referenciaszám nélkül is és egy vagy több referenciaszámmal is állhat.
- Pl. $A=B+C$ (nincs referenciaszám)
 $3)A=B+C$ (egy referenciaszám van)
 $3)1)12)A=B+C$ (három referenciaszám van)
- 2.5 Ha egy referenciaszám egy adott programon belül többször fordul elő, ez hibajelzést okoz.
- 2.6 Referenciaszámra hivatkozni az összes JUMP típusu és a SUBR utasítással, valamint adatszalagon "trigger"-rel lehet.
- 2.7 A program futása során érintett referenciaszámokat ki lehet iratni, ha a programozó a programot referenciaszámos nyomkövetéssel futtatja.
3. Műveleti jelek.
A műveleti jelek általában két operandus között elvégzésre kerülő műveletet jelölik ki.
Különleges esetben műveleti jel egy operandusra is vonatkozhat.
- 3.1 Aritmetikai műveleti jelek.
- 3.1.1 Összeadás: műveleti jele: +
A + műveleti jelnek mindig két operandus között kell állnia.
Pl. $C=A+B$
- 3.1.2 Kivonás: műveleti jele: -
A - műveleti jel vonatkozhat egy operandusra is.
Ebben az esetben hatása olyan, mintha a - műveleti jelet a 0 operandus előzné meg.
Pl. $D=A-B$, $K=-I$ (ez megfelel $K=0-I$ -nek)
- 3.1.2 Szorzás: műveleti jele: *
A * műveleti jelnek mindig két operandus között kell állnia. Pl.
Pl. $V=X*Y$
- 3.1.4 Osztás: műveleti jele: /
A / műveleti jelnek mindig két operandus között kell állnia.
Pl. $D=A/6$
- 3.1.5 Valamennyi aritmetikai műveleti jel fixpontos, lebegőpontos és duplafixpontos típusu utasításokban is szerepelhet.
- 3.2 Logikai és léptetési műveleti jelek.
- 3.2.1 Negáció: műveleti jele: !
A ! műveleti jel mindig csak egy operandusra, a közvetlenül mögötte álló operandusra vonatkozik. E megkötésből következik, hogy mivel

a ! műveleti jel nem két operandus közti műveletet jelöl, csak egy utasítás jobb oldalán közvetlenül az = jel után állhat. Pl. $I = ! J$

3.2.2 Antivalencia: műveleti jele: %

A % műveleti jel két oldalán álló operandusra vonatkozik.

Pl. $I = J \% K$

3.2.3 Logikai szorzás: műveleti jele: &

A & műveleti jel mindig a & jel két oldalán álló operandusra vonatkozik. Pl. $L = K \& 127$

3.2.4 Aritmetikai léptetés: műveleti jele: 'a'

A 'a' műveleti jel bal oldalán álló operandus mutatja, hogy mit kell léptetni, a jobb oldalán álló operandus, hogy mennyivel kell léptetni. Pl. $L = I 'a' K$

A léptetés K tartalmára nézve modulo 63-mal történik, úgy, hogy ha $K > 0$, balra léptetés, ha $K < 0$, jobbra léptetés jön létre.

3.2.5 Boole - léptetés: műveleti jele: 'b'

A 'b' műveleti jel mindig két operandusra vonatkozik. A műveleti jel bal oldalán álló operandus mutatja, hogy mit kell léptetni, a jobb oldalán álló operandus, hogy mennyivel kell léptetni.

Pl. $I = 5 'b' J$

A léptetés a J operandus tartalmára nézve modulo 63-mal történik, úgy, hogy ha $J > 0$, balra léptetés, ha $J < 0$, jobbra léptetés jön létre.

3.2.6 Ciklikus - léptetés: műveleti jele: 'c'

A 'c' mindig két operandus között áll. A műveleti jel bal oldali operandusa mutatja, hogy mit kell léptetni, a jobb oldali operandus, hogy mennyivel kell léptetni. Pl. $I = K 'c' L$

A léptetés az L operandus tartalmára nézve modulo 63-mal megy végbe, úgy, hogy ha $L > 0$ balraléptetés, ha $L < 0$ jobbraléptetés jön létre.

3.2.7 Valamennyi logikai és léptetési műveleti jel csak fixpontos típusu utasításokban alkalmazható.

3.3 Műveleti jelek keverése.

Egy utasításon belül vagy csak aritmetikai, vagy csak logikai és léptetési műveleti jel alkalmazható.

Egy utasításon belül a műveleti jelek számára nézve nincs megszorítás.

4. Deklarációs utasítások.

Minden autokód programot programdeklarációval kell bevezetni.

A programdeklarációban fel kell tüntetni a programban felhasznált változókat legalább a programban előforduló legnagyobb index-szel,

a programban alkalmazott standard függvényeket és egy olyan referenciaszámot, melynél nagyobb a programban nem használunk.

A deklarációs utasítások:

SETS fixpontos változók felsorolása
SETV lebegőpontos változók felsorolása
SETD duplafixpontos változók felsorolása
SETF standard függvényes felsorolása
SETR referenciaszám feltüntetése

A deklarációs utasításokat tetszőszerinti sorrendben lehet feltüntetni a SETR- kivételével. A SETR-nek mindig utoljára kell állnia. Egy deklarációs sort az "uj sor" karakter zár le. A SETR kivételével valamennyi típusdeklaráció többször is szerepelhet. Ha valamelyik típust a programban nem alkalmazzuk, a deklarációban nem kell feltüntetni. A SETR-nek mindig szerepelnie kell.

4.1 Fixpontos változók felsorolása.

SETS után fel kell sorolni a programban szereplő összes fixpontos változót.

A változó után [] közé kell írni a változónak a programban előforduló legnagyobb indexét, ill. kétdimenziós tömb esetén , -vel elválasztott index-részeit, számindex alakjában. Pl.

SETS I[3] J[5,8]

SETV vagy SETD után már feltüntetett változó

SETS után nem állhat.

4.2 Lebegőpontos változók felsorolása.

SETV után fel kell sorolni a programban szereplő összes lebegőpontos változót.

A változó után [] közé kell írni a változónak a programban előforduló legnagyobb indexét, ill. kétdimenziós tömb esetén -vel elválasztott index-részeit, számindex alakjában. Pl.

SETV X[100] Y[1,6]

SETS vagy SETD után feltüntetett változó

SETV után nem állhat.

4.3 Duplafixpontos változók felsorolása.

SETD után fel kell tüntetni a programban szereplő összes duplafixpontos változót.

A változó után [] közé kell írni a változónak a programban előforduló legnagyobb indexét, ill. kétdimenziós tömb esetén , -vel elválasztott index-részeit, számindex alakjában. Pl.

SETD A[12] C[4,4] B

SETS vagy SETV után feltüntetett változó

SETD után nem állhat.

4.4 Függvénydeklaráció.

SETF után fel kell sorolni a programban használt valamennyi standard függvényt.

- 4.5 Referenciaszám deklaráció.
SETR után fel kell tüntetni a programban használható legnagyobb referenciaszámot.
5. Alapszavak.
Az EMG 830 autokód rendszer a következő alapszavakkal rendelkezik. (Minden alapszó csak programutasítás elején állhat.) Az alapszavak csak nagybetűket tartalmazhatnak.
- 5.1 JUMP Hatása: Feltétel nélküli ugrás az alapszó után meghatározott referenciaszámra.
- 5.2 JUMP IF Hatása: Feltételes ugrás az alapszót követő reláció teljesülése alapján.
Formái: JUMP IF $A < B \odot K$
JUMP IF $A = B \odot K$
JUMP IF $A > B \odot K$
JUMP IF $I \odot K$
- 5.3 JUMP UNLESS. Hatása: Feltételes ugrás az alapszót követő reláció nem teljesülése alapján.
Formái: JUMP UNLESS $A < B \odot K$
JUMP UNLESS $A = B \odot K$
JUMP UNLESS $A > B \odot K$
A JUMP UNLESS $A < B \odot K$ egyenértékű a
JUMP IF $A = B \odot K$ és
JUMP IF $A > B \odot K$ utasítások együttes hatásával.
A JUMP UNLESS $A = B \odot K$ egyenértékű a
JUMP IF $A < B \odot K$ és
JUMP IF $A > B \odot K$ utasítások együttes hatásával.
A JUMP UNLESS $A > B \odot K$ egyenértékű a
JUMP IF $A < B \odot K$ és
JUMP IF $A = B \odot K$ utasítások együttes hatásával.
- 5.4 SUBR Hatása: Feltétel nélküli ugrás az alapszó után meghatározott referenciaszámra, az elugrási pont címének feljegyzésével.
- 5.5 EXIT Hatása: Ugrás a SUBR követő utasításra.
- 5.6 VARY Hatása: Ciklus kezdés.
A ciklust annyiszor kell végrehajtani, amennyit a VARY $I = 1 : J : K$ típusu utasításban a K operandus meghatároz.
- 5.7 CYCLE
Formái: CYCLE $I = 1 : K : L$ Hatása: Cikluskezdés.
A ciklust addig kell végrehajtani, míg I aktuális értéke el nem éri, vagy túl nem haladja az L operandus értékét.

CYCLE I = 1, 5, K, L Hatása: Cikluskezdés.

A ciklust a feltüntetett operandusokkal, sorrendben kell végrehajtani.

- 5.8 REPEAT Hatása: Lezárja a legutoljára megnyitott ciklust, eldönti, hogy a ciklust újabb értékkel végre kell-e hajtani. E feltétel eredményétől függően ugrást eredményez a ciklusmag első utasítására, vagy a program a REPEAT-ot követő utasításnál folytatódik.
- 5.9 READ Hatása: Az olvasóba helyezett lyukszalagról egy szám beolvasódik az alapszó után feltüntetett változó rekeszébe.
- 5.10 INPUT Hatása: Az olvasóba helyezett lyukszalagról egy karakter beolvasódik az alapszó után feltüntetett változó rekeszébe.
- 5.11 PRINT Hatása: Az alapszó után feltüntetett operandus kinyomtatásra kerül.
- 5.12 OUTPUT Hatása: Az alapszó után feltüntetett operandus karakter értéke kinyomtatásra kerül.
- 5.13 CHECK Hatása: Az alapszó után feltüntetett operandus kinyomtatásra kerül, ha a kapcsolószó 20. bitjének értéke: 1.
- 5.14 WAIT Hatása: A program futása megszakad, konzolüzenetre a WAIT-ot követő utasítástól folytatódik.
- 5.15 STOP Hatása: A programfutása megszakad, konzolüzenettel adott referenciaszámról a program újra elindítható.
- 5.16 LINE Hatása: Egy soremelés nyomtatódik ki.
- 5.17 LINES Hatása: Az alapszót követő operandusnak megfelelő számú soremelés nyomtatódik ki.
- 5.18 SPACE Hatása: Egy szóközzel nyomtatódik ki.
- 5.19 SPACES Hatása: Az alapszót követő operandusnak megfelelő számú szóközzel nyomtatódik ki.
- 5.20 TITLE Hatása: Az alapszót követő jelek a legközelebbi karakterig kinyomtatódnak.
- 5.21 READER Hatása: A beolvasó perifériák sorszáma alapján meghatározza, hogy a soron következő lyukszalag olvasás melyik egységen történjen.
- 5.22 PUNCH Hatása: A nyomtató perifériák sorszáma alapján meghatározza, hogy a soron következő nyomtatás melyik egységen történjen.
- 5.23 SET Hatása: Kezdeti értékadás az alapszót követő változó feltüntetett indexértékeinek.
- 5.24 START Hatása: A programfordítás befejezését jelzi.

6. Függvények.

Az EMG 830 autókód rendszer a következő standard függvényekkel rendelkezik.

- 6.1 SIN Értelmezése: Megegyezik a matematikában szokásos $\sin x$ értelmezéssel, de teljesülnie kell:

$$-\frac{\pi}{2} \leq x \leq \frac{\pi}{2} \text{ -nek.}$$

Az utasítás formája: $Y = \text{SIN } X$ (lebegőpontos)

- 6.2 COS Értelmezése: Megegyezik a matematikában szokásos $\cos x$ értelmezéssel, de teljesülnie kell:

$$-\frac{\pi}{2} \leq x \leq \frac{\pi}{2} \text{ -nek.}$$

Az utasítás formája: $Y = \text{COS } X$ (lebegőpontos)

- 6.3 TAN Értelmezése: Megegyezik a matematikában szokásos $\text{tg } x$ értelmezéssel, de teljesülnie kell:

$$-\frac{\pi}{2} \leq x \leq \frac{\pi}{2} \text{ -nek.}$$

Az utasítás formája: $Y = \text{TAN } X$ (Lebegőpontos)

- 6.4 ARCTAN Értelmezése: Megegyezik a matematikában szokásos $\text{arctg } x$ - értelmezéssel, de teljesülnie kell:

$$-\frac{\pi}{2} \leq \text{arctg } x \leq \frac{\pi}{2} \text{ -nek.}$$

Az utasítás formája: $Y = \text{ARCTAN } X$ (lebegőpontos)

- 6.5 LOG Értelmezése: $y = e^{\log x} = \ln x$ ($x > 0$)

Az utasítás formája: $Y = \text{LOG } X$ (lebegőpontos)

- 6.6 EXP Értelmezése: $y = e^x$

Az utasítás formája: $Y = \text{EXP } X$ (lebegőpontos)

- 6.7 SQRT Értelmezése: $y = \sqrt{x}$

Az utasítás formája: $Y = \text{SQRT } X$ (Legegőpontos)

- 6.8 FRAC Értelmezése: Egy operandus törtrészének leválasztása.

Az utasítás formája: $Y = \text{FRAC } X$ (lebegőpontos)

- 6.9 STAND Értelmezése: Az operandus értékét más típusú számmá konvertálja az alábbiak szerint:

a) fixpontos argumentum $\rightarrow$ lebegőpontos függvényérték

b) duplafixpontos argumentum $\rightarrow$ lebegőpontos függvényérték

c) fixpontos argumentum $\rightarrow$ duplafixpontos függvényérték

Az utasítás formái:

Y = STAND J (lebegőpontos)
 Y = STAND D (lebegőpontos)
 B = STAND I (dupla fixpontos)

- 6.10 INT Értelmezése: Az operandus értékének egész részét képezi és az így kapott számot a függvényérték típusának megfelelő alakra konvertálja.

Az utasítás formái:

Y = INT X (lebegőpontos)
 I = INT X (fixpontos)
 B = INT X (duplafixpontos)

- 6.11 MOD Értelmezése: Az operandus abszolút értékét képezi.

Az utasítás formái:

I = MOD J (fixpontos)
 X = MOD Y (lebegőpontos)
 A = MOD B (duplafixpontos)

- 6.12 A felsorolt függvények argumentumában szereplő operandusok a függvények értelmezése szerint változók és konstansok, vagy csak változók lehetnek. A következő táblázat áttekintést ad az operandusokra nézve:

Y = SIN X
 Y = COS X
 Y = TAN X
 Y = ARCTAN X X helyett konstans is állhat
 Y = LOG X
 Y = EXP X
 Y = SQRT X
 Y = FRAC X

Y = STAND J
 Y = INT X
 Y = MOD X
 Y = STAND D J, X, D helyet nem állhat konstans

B = INT X
 B = MOD D
 B = STAND J

I = INT X
 I = MOD J

7. Aritmetikai utasítások.

Az aritmetikai utasítások alakja az alábbi:

aritmetikai utasítás bal oldal = aritmetikai utasítás jobb oldal.

- 7.1 Aritmetikai utasítás bal oldala:
Az aritmetikai utasítás bal oldalának lehetséges típusai:
- | | |
|---|------------------|
| A | (duplafixpontos) |
| I | (fixpontos) |
| X | (lebegőpontos) |
- vagyis az aritmetikai utasítás bal oldalán mindig egy és csak egy változó állhat. (Az A, I, vagy X indexében esetleg szereplő változókat figyelmen kívül hagyjuk, csak az indexen kívüli változók számítanak.)
- 7.2 Az aritmetikai utasítás típusa:
Az aritmetikai utasítás típusát az aritmetikai utasítás bal oldalán álló változó típusa határozza meg.
- a) Fixpontos az az aritmetikai utasítás, melynek bal oldalán fixpontos változó áll.
 - b) Lebegőpontos az az aritmetikai utasítás, melynek bal oldalán lebegőpontos változó áll.
 - c) Duplafixpontos az az aritmetikai utasítás, melynek bal oldalán duplafixpontos változó áll.
- 7.3 Aritmetikai utasítás jobb oldala:
Aritmetikai utasítás jobb oldalán tetszőszerinti számú, de legalább egy operandus állhat, két egymásután következő operandus között aritmetikai műveleti jelnek kell állnia.
Az utasítás jobb oldalán álló operandusoknak azonos típusuaknak kell lenniük az utasítás típusával.
- 7.4 Az aritmetikai utasítás végét "uj sor" karakter jelzi.
- 7.5 Aritmetikai utasítás jobb oldalán operandusként függvény is állhat.
- 7.6 Néhány példa: $A = B$
 $A = B * INT \quad C [3] + MOD B$
 $D = -A/2 * STAND J + 123456789$
8. Logikai és léptetési utasítások.
A logikai és léptetési utasítások alapja az alábbi:
logikai és léptetési utasítás bal oldal = logikai és léptetési utasítás jobb oldal.
- 8.1 Logikai és léptetési utasítás bal oldala:
Az utasítás bal oldalának lehetséges típusa:
- | | |
|---|-------------|
| I | (fixpontos) |
|---|-------------|
- vagyis a logikai és léptetési utasítás bal oldalán (az indexben szereplő esetleges változókat nem számítva) mindig egy és csak egy változó állhat.
- 8.2 A logikai és léptetési utasítás típusa:
Logikai és léptetési utasítás típusa csak fixpontos lehet.

- 8.3 Logikai és léptetési utasítás jobb oldala:
 Logikai és léptetési utasítás jobb oldalán tetszőszerinti szám, de legalább egy operandus állhat. Két egymásután következő operandus között logikai vagy léptetési műveleti jelnek kell állnia. Az utasítás jobb oldalán álló operandusok csak fixpontosak lehetnek.
- 8.4 Egy logikai és léptetési utasítás végét "uj sor" karakter jelzi.
- 8.5 Logikai és léptetési utasítás jobb oldalán operandusként fixpontos függvény is állhat.
- 8.6 Negáció jel (!) csak közvetlenül az = jel után állhat.
- 8.7 Néhány példa: $I = J \& K$
 $I = ! J 'c' K \% INT X$
 $K = ! L \% I 'a' MOD J \& I$

9. Precedencia szabályok.

- 9.1 Aritmetikai utasításra vonatkozó precedencia szabályok. Az aritmetikai műveletek elsőbbség szempontjából két csoportra bonthatók:
- 9.1.1 Elsődleges műveletek: π , /
- 9.1.2 Másodlagos műveletek: +, -
- 9.1.3 Azonos csoporton belül a balról - jobbra haladás elve érvényesül.
- 9.1.4 Függvényutasítás végrehajtása megelőzi az aritmetikai műveletek végrehajtását.
- 9.1.5 Példák: $A + B \pi C + D/E$ végrehajtás szempontjából megfelel $A + (B \pi C) + (D/E)$ -nek.
 $A + B \pi 3/SQRT\ 15/C + 2 \pi A \pi D$ megfelel $A + (B \pi 3)/((SQRT\ 15) \pi C) + (2 \pi A \pi D)$ -nak.
- 9.2 Logikai - léptetési utasításra vonatkozó precedencia szabályok.
- 9.2.1 Logikai - és léptetési utasításokban csak a balról - jobbra haladás elve érvényesül.
- 9.2.2 Függvényutasítás végrehajtása megelőzi a logikai léptetési művelet végrehajtását.
- 9.2.3 Példák: $I 'a' 2 \% INT X$ megfelel $(I 'a' 2) \% (INT X)$ -nek

10. Változók indexei.

Valamennyi változó típusba sorolásától függetlenül három csoportba osztható.

- Indexnélküli változó
- Egydimenziós index-szel ellátott változó
- Kétdimenziós index-szel ellátott változó.

- 10.1 Indexnélküli változó.
- 10.1.1 Indexnélküli változónak nevezzük azt a változót, mely az angol ABC egy és csak egy nagy vagy kisbetűjéből áll.
- 10.1.2 Indexnélküli változó lehet
- a) az a változó, melyet már a deklarációban indexnélküli változónak deklaráltunk.
 - b) Egydimenziós index-szel deklarált változó, melyhez a 0 indexet rendeljük hozzá.
Pl. I vagy X
- 10.2 Egydimenziós index-szel ellátott változó.
- 10.2.1. Egydimenziós indexű változónak nevezzük azt a változót, mely az angol ABC egy nagy- vagy kisbetűjéből és közvetlenül mögé [] közé irt tetszőleges, fixpontos, index nélküli változókból álló, aritmetikai utasítás jobb oldalból áll.
Pl. B $[3 \times I + J/K - 2]$
- 10.2.2 Egydimenziós index-szel ellátott változó csak az a változó lehet, melyet már a deklarációban egydimenziós változónak deklaráltunk.
- 10.2.3 A programban alkalmazott maximális index-et a deklarációban a változó után [] zárójelek közé irt számérték határozza meg.
- 10.2.4 Negatív index hibajelzést okoz.
- 10.2.5 A deklaráltnál nagyobb index nem okoz hibajelzést, csak akkor, ha az így indexelt változó az összes deklarált változó határain kívül esik.
- 10.3 Két dimenziós index-szel ellátott változó.
- 10.3.1 Kétdimenziós indexű változónak nevezzük azt a változót, mely az angol ABC egy nagy vagy kisbetűjéből és közvetlenül mögé [] közé irt és , -vel elválasztott két, tetszőleges, fixpontos, index nélküli változókból álló, aritmetikai utasítás jobb oldalból áll.
Pl. A $[3 + I \times K, -2 + J/L \times \text{INT } X]$
- 10.3.2 Kétdimenziós index-szel ellátott változó csak az a változó lehet, melyet már a deklarációban kétdimenziós változónak deklaráltunk.
- 10.3.3 Kétdimenziós index első index része a sorok számát, második index része az oszlopok számát mutatja.

10.3.4 A kétdimenziós változó oszlopainak és sorainak számozása 1-gyel kezdődik. (Tehát, ha a deklarációban

$A \begin{bmatrix} 5, 3 \end{bmatrix}$, akkor a mátrix elemei:

$A \begin{bmatrix} 1, 1 \end{bmatrix}$ $A \begin{bmatrix} 1, 2 \end{bmatrix}$ $A \begin{bmatrix} 1, 3 \end{bmatrix}$

$A \begin{bmatrix} 2, 1 \end{bmatrix}$ $A \begin{bmatrix} 2, 2 \end{bmatrix}$ $A \begin{bmatrix} 2, 3 \end{bmatrix}$

$A \begin{bmatrix} 3, 1 \end{bmatrix}$ $A \begin{bmatrix} 3, 2 \end{bmatrix}$ $A \begin{bmatrix} 3, 3 \end{bmatrix}$

$A \begin{bmatrix} 4, 1 \end{bmatrix}$ $A \begin{bmatrix} 4, 2 \end{bmatrix}$ $A \begin{bmatrix} 4, 3 \end{bmatrix}$

$A \begin{bmatrix} 5, 1 \end{bmatrix}$ $A \begin{bmatrix} 5, 2 \end{bmatrix}$ $A \begin{bmatrix} 5, 3 \end{bmatrix}$

Nincs értelmezve az az eset, ha valamelyik indexrész nullává válik.

10.3.5 A programban alkalmazott maximális indexrészeket a deklarációban a változó után [] zárójelek közé irt és , -vel elválasztott számértékek határozzák meg.

10.3.6 Ha a kiszámított index negatív, ez hiba jelzést okoz.

10.3.7 A deklarálnál nagyobb index nem okoz hibajelzést, csak akkor, ha az így indexelt változó az összes deklarált változó határán kívül esik.

11. Beolvasó utasítások.

11.1 READ

READ utasítás hatására az adatszagról egy adat beolvasásra kerül a READ alapszó után irt változó rekeszébe.

11.1.1 Az utasítás típusai.

READ I A beolvasott szám fixpontos formában kerül a memóriába.

READ A A beolvasott szám duplaxfixpontos formában kerül a memóriába.

READ X A beolvasott szám lebegőpontos formában kerül a memóriába.

11.1.2 READ alapszó után csak változó állhat, konstans nem.

11.2 INPUT

INPUT utasítás hatására az adatszagról egy karakter beolvasásra kerül az INPUT alapszó után irt változó rekeszébe.

11.2.1 Az utasítás típusa.

INPUT I A beolvasott karakter, a karakter bináris értékének megfelelő fixpontos formában kerül a memóriába.

11.2.2 INPUT alapszó után csak fixpontos változó írható, más típusú változó vagy konstans nem.

12. Kinyomtató utasítások.

12.1 PRINT

Az utasítás hatására az alapszó után feltüntetett operandus kinyomtatásra kerül.

- 12.1.1 Az utasítás típusai:
 PRINT I Az operandus értéke fixpontos formában kinyomtatódik.
 PRINT A Az operandus értéke duplafixpontos formában kinyomtatódik.
 PRINT X Az operandus értéke lebegőpontos formában kinyomtatódik.
- 12.1.2 PRINT alapszó után konstans is állhat. Pl. PRINT 3.2
- 12.1.2 PRINT utasítások nyomtatási paraméterei:
 a) Fixpontos változó esetén:
- 12.1.3 PRINT I Jelentése: Nyomtatás a legutolsó fixpontos nyomtatási formátum szerint. Ha a programban még nem volt fixpontos nyomtatási paraméter, a nyomtatás 4 számjegyre történik.
- 12.1.4 PRINT I, 3 Jelentése: Nyomtatás 3 számjegyre. (A 3 helyett más poz. konstans is állhat, de nem lehet nagyobb 15-nél.)
- 12.1.5 PRINT I, N Jelentése: Nyomtatás N számjegyre.
- 12.1.6 PRINT I', 3 Jelentése: Ugyanaz, mint PRINT I, 3-nál, de $I \geq 0$ esetén az előjel nyomtatás pozíciója kimarad.
- 12.1.7 PRINT I', N Jelentése: Ugyanaz, mint PRINT I, N-nél, de $I \geq 0$ esetén az előjel nyomtatás pozíciója kimarad.
- b) Lebegőpontos változó esetén
- 12.1.8 PRINT X Jelentése: Nyomtatás a legutolsó lebegőpontos nyomtatási formátum szerint. Ha a programban még nem volt lebegőpontos nyomtatási paraméter a nyomtatás 6 számjegyre a PRINT X, /-nek megfelelő kitevős alakban történik.
- 12.1.9 PRINT X, 3 Jelentése: A nyomtatás 3 jegyre történik (3 helyett más pozitív konstans is vállalhat, de nem lehet nagyobb 15-nél)
- 12.1.10 PRINT X, N Jelentése: Nyomtatás N jegyre.
- 12.1.11 PRINT X', 3 Jelentése: Ugyanaz, mint PRINT X, 3-nál de ha $X \geq 0$, az előjel nyomtatás pozíciója kimarad.
- 12.1.12 PRINT X', N Jelentése: Ugyanaz, mint PRINT X, N-nél, de ha $X \geq 0$, az előjel nyomtatás pozíciója kimarad.
- 12.1.13 PRINT X, / Jelentése: Nyomtatás 6 számjegyre kitevős alakban.
- 12.1.14 PRINT X, 3/ Jelentése: Nyomtatás 6 számjegyre kitevős alakban. (Ugyanez a jelentése bármely más konstanssal is.)

12.1.15	PRINT X,N/	Jelentése: Ugyanaz, mint PRINT X, /-nál
12.1.16	PRINT X', /	Jelentése: Ugyanaz, mint PRINT X, /-nak, de ha $X \geq 0$, az előjel nyomtatás pozíciója kimarad.
12.1.17	PRINT X',3/	Jelentése: Ugyanaz, mint PRINT X', /-nak. (Ugyanaz a jelentése bármely más konstanssal is.)
12.1.18	PRINT X',N/	Jelentése: Ugyanaz, mint PRINT X', /-nak.
12.1.19	PRINT X,3:M	Jelentése: Nyomtatás 3 egész és M tizedesjegyre. (A 3 helyett más, 15-nél nem nagyobb poz. konstans is állhat.)
12.1.20	PRINT X,3:2	Jelentése: Nyomtatás 3 egész és 2 tizedesjegyre (A 3, ill. 2 helyett más, 15-nél nem nagyobb poz. konstans is állhat.)
12.1.21	PRINT X,N:M	Jelentése: Nyomtatás N egész és M tizedesjegyre.
12.1.22	PRINT X,N:2	Jelentése: Nyomtatás N egész és 2 tizedesjegyre (A 2 helyett más, 15-nél nem nagyobb poz. konstans is állhat.)
12.1.23	PRINT X',3:M	Jelentése: Ugyanaz, mint PRINT X,3:M-nél, de ha $X \geq 0$, az előjel nyomtatás pozíciója kimarad.
12.1.24	PRINT X',3:2	Jelentése: Ugyanaz, mint PRINT X,3:2-nél, de ha $X \geq 0$, az előjel nyomtatás pozíciója kimarad.
12.1.25	PRINT X',N:M	Jelentése: Ugyanaz, mint PRINT X,N:M-nél, de ha $X \geq 0$, az előjel nyomtatás pozíciója kimarad.
12.1.26	PRINT X',N:2	Jelentése: Ugyanaz, mint PRINT X,N:2-nél, de ha $X \geq 0$, az előjel nyomtatás pozíciója kimarad.

c) Duplafix pontos változó esetén:

12.1.27	PRINT A	Jelentése: Ugyanaz, mint PRINT I-nek
12.1.28	PRINT A,3	Jelentése: Ugyanaz, mint PRINT I,3-nak
12.1.29	PRINT A,N	Jelentése: Ugyanaz, mint PRINT I,N-nek
12.1.30	PRINT A',3	Jelentése: Ugyanaz, mint PRINT I',3-nak
12.1.31	PRINT A',N	Jelentése: Ugyanaz, mint PRINT I',N-nek
12.1.32	Amennyiben a kinyomtatandó szám negatív, a szám előtt - előjel, ha pozitív a szám előtt "szóköz" kerül kinyomtatásra (A " , " ezt a "szóköz" kinyomtatást tiltja le.)	

- 12.1.33 Valamennyi kinyomtatott szám után elválasztójelként egy "szóköz" kerül kinyomtatásra.
- 12.1.34 Ha egy kinyomtatásra kerülő operandus meghaladja azt a nagyságrendet, melyre a nyomtatási paraméter lehetőséget ad, a szám új sorban, a típus nyomtatási alapformátuma szerint, egy ? - lel megelőzve kerül kinyomtatásra.
- 12.1.35 A PRINT utasítások összefoglalása:

Fixpontos nyomtatás:

PRINT	I	I [J]	I [J,K]	15
	I,3	I [J],3	I [J,K],3	15,3
	I,N	I [J],N	I [J,K],N	15,N
	I',3	I [J]',3	I [J,K]',3	15',3
	I',N	I [J]',N	I [J,K]',N	15',N

Lebegőpontos nyomtatás:

PRINT	X	X [J]	X [J,K]	1.234
	X,3	X [J],3	X [J,K],3	1.234,3
	X,N	X [J],N	X [J,K],N	1.234,N
	X',3	X [J]',3	X [J,K]',3	1.234',3
	X',N	X [J]',N	X [J,K]',N	1.234',N
	X,/	X [J],/	X [J,K],/	1.234'/
	X,/	X [J]',/	X [J,K]',/	1.234'/'
	X,3:M	X [J],3:M	X [J,K],3:M	1.234, 3:M
	X,3:2 X	X [J],3:2	X [J,K],3:2	1.234, 3:2
	X,N:M	X [J],N:M	X [J,K],N:M	1.234, N:M
	X,N:2	X [J],N:2	X [J,K],N:2	1.234, N:2
	X',3:M	X [J]',3:M	X [J,K]',3:M	1.234', 3:M
	X',3:2	X [J]',3:2	X [J,K]',3:2	1.234', 3:2
	X',N:M	X [J]',N:M	X [J,K]',N:M	1.234', N:M
	X'N:2	X [J]',N:2	X [J,K]',N:2	1.234', N:2

Duplafixpontos nyomtatás:

PRINT	A	A [J]	A [J,K]	139
	A,3	A [J],3	A [J,K],3	139,3
	A,N	A [J],N	A [J,K],N	139,N
	A',3	A [J]',3	A [J,K]',3	139',3
	A',N	A [J]',N	A [J,K]',N	139',N

12.2 OUTPUT

Az utasítás hatására az alapszó után feltüntetett operandus utolsó 8 bitjének megfelelő karakter kerül kinyomtatásra.

- 12.2.1 Az utasítás típusa:
OUTPUT I
Az utasítás csak fixpontos változóra van értelmezve.
- 12.2.2 OUTPUT alapszó után poz. fixpontos konstans is állhat.
- 12.3 TITLE
Az utasítás hatására az alapszó után feltüntetett jelek kinyomtatódnak.
A kinyomtatandó jelek végét , jelzi.
13. Nyomkövetési utasítások.
- 13.1 CHECK
Ha a kapcsolószo 20 bitjének állása 1 az utasítás hatása megegyezik a PRINT utasítással.
Ha a kapcsolószo 20 bitjének állása 0, az utasítás üres utasítás.
- 13.1.1 Az utasítás típusai:
- | | |
|---------|---|
| CHECK I | Az utasítás, nyomtatás esetében megfelel a PRINT I-nek. |
| CHECK A | Az utasítás, nyomtatás esetében megfelel a PRINT A-nak. |
| CHECK X | Az utasítás, nyomtatás esetében megfelel a PRINT X-nek. |
- 13.1.2 Minden CHECK utasítás új sorban a típus alapnyomtatásának megfelelő formátumban, egy * jellel megelőzve történik.
- 13.1.3 A CHECK alapszó után változó is, konstans is állhat.
Pl. CHECK I, CHECK 13
- 13.2 Referenciaszám nyomtatás
Ha a kapcsolószo 21 bitjének állása 1, minden a programfutás során érintett referenciaszám kinyomtatásra kerül.
- 13.2.1 Minden kinyomtatott referenciaszám új sorban, egy)-lel követve nyomtatódik ki.
14. Periféria egység kijelölő utasítások.
- 14.1 READER
Az utasítás a beolvasó periféria egységek kiválasztására szolgál.
Az utasítás hatására a következő READER utasításig minden beolvasás, az alapszó után következő operandus által meghatározott szalagolvasó berendezésen keresztül történik.
- 14.1.1 Az utasítás típusa:
READER I Az utasítás operandusa csak fixpontos lehet.
- 14.1.2 Az alapszó után változó is, konstans is állhat, de értékük csak 1, vagy 2 lehet.

- 14.1.3 READER 1 az 1-sz. olvasót, a
READER 2 a 2-sz. olvasót jelenti.
- 14.1.4 Ha egy programban nincs READER utasítás, vagy még nem került sorra, minden beolvasás automatikusan az 1-sz. olvasón történik.
- 14.2 PUNCH
Az utasítás a nyomtató periféria egységek kiválasztására szolgál. Az utasítás hatására a következő PUNCH utasításig minden nyomtatás az alapszó után következő operandus által meghatározott nyomtató berendezésen keresztül történik.
- 14.2.1 Az utasítás típusa:
PUNCH I Az utasítás operandusa csak fixpontos lehet.
- 14.2.2 Az alapszó után változó is, konstans is állhat.
Értékük 1, 2, 3, 4 lehet.
- 14.2.3 PUNCH 1 az 1- szalaglyukasztót
PUNCH 2 a 2- szalaglyukasztót
PUNCH 3 a teleprintert
PUNCH 4 a lineprintert jelenti.
- 14.2.4 Ha egy programban nincs PUNCH utasítás, vagy még nem került sorra, minden nyomtatás automatikusan az 1-sz. szalaglyukasztón történik.
15. Vezérlés átadási utasítások.
- 15.1 Feltétel nélküli ugrást előidéző utasítások.
- 15.1.1 JUMP
Az utasítás hatására a program futása az alapszó után feltüntetett operandus értékének megfelelő referenciaszámtól folytatódik.
- 15.1.2 A JUMP utasítás típusa:
JUMP @ I Az utasítás operandusa csak fixpontos aritmetikai utasítás jobb oldal állhat.
pl. JUMP @ I + 3 * INT x/10
- 15.1.4 Az I csak a programban valahol feltüntetett referenciaszám értékét veheti fel.
- 15.1.5 SUBR
Az utasítás hatására a program futása az SUBR alapszó után feltüntetett operandus értékének megfelelő referenciaszámtól folytatódik.
- 15.1.6 A SUBR utasítás típusa:
SUBR I Az utasítás operandusa csak fixpontos lehet.
I helyett tetszőleges fixpontos aritmetikai utasítás jobb oldal állhat pl. SUBR I + 3 * K/L

- 15.1.7 Az I csak a programban valahol feltüntetett referenciaszám értékét veheti fel.
- 15.1.8 Minden SUBR utasításhoz, EXIT utasításnak is kell tartoznia.
- 15.1.9 EXIT
Az utasítás hatására a program futása a hozzárendelt SUBR utasítást követő utasítástól folytatódik.
- 15.1.10 SUBR utasítások 10-szeres mélységben lehetnek egymásba építve. Egy EXIT mindig a legbelül levő szubrutin végpontját jelenti.
- 15.2 Feltétel teljesülése esetén ugrást előidéző utasítások.
- 15.2.1 JUMP IF
Az utasítás hatására, amennyiben a JUMP I F után írt feltétel teljesül a program futása a feltételt követő referenciaszámtól folytatódik. Ha a feltétel nem teljesül, az utasítás üres utasítás.
- 15.2.2 Az utasítás típusai:
- a) JUMP IF $J < K \text{ @ } I$
 - JUMP IF $J = K \text{ @ } I$
 - JUMP IF $J > K \text{ @ } I$
 - JUMP IF $A < B \text{ @ } I$
 - JUMP IF $A = B \text{ @ } I$
 - JUMP IF $A > B \text{ @ } I$
 - JUMP IF $X < Y \text{ @ } I$
 - JUMP IF $X = Y \text{ @ } I$
 - JUMP IF $X > Y \text{ @ } I$
- 15.2.3 A feltétel bal oldala tetszőleges típusu utasítás bal oldal lehet.
- 15.2.4 A feltétel jobb oldala, a bal oldal típusának megfelelő aritmetikai, vagy logikai léptetési utasítás jobb oldal lehet.
- 15.2.5 A feltétel bal és jobb oldala között $<$, $=$, $>$ jel állhat.
- 15.2.6 I helyett tetszőleges fixpontos aritmetikai utasítás jobb oldal állhat.
- 15.2.6 I csak a programban valahol feltüntetett referenciaszám értékét veheti fel.
- 15.2.7 Az utasítás típusai:
- b) JUMP IF $K \text{ @ } I$
- Az utasítás a kapcsolószó K tartalmának megfelelő bitjét vizsgálja meg. Ha a kérdéses bit 1, a program futása az I referenciaszámtól folytatódik. Ha a kérdéses bit 0, az utasítás üres utasítás.
- 15.2.8 K csak egyetlen fixpontos változó, vagy konstans lehet. $1 \leq K \leq 21$ feltételnek teljesülnie kell.

15.2.9 JUMP UNLESS

Az utasítás hatására, amennyiben a JUMP UNLESS után írt feltétel nem teljesül, a program futása a feltételt követő referenciaszámtól folytatódik. Ha a feltétel teljesül az utasítás üres utasítás.

15.2.10 Az utasítás típusai:

JUMP UNLESS $J < K \text{ @ } I$
JUMP UNLESS $J = K \text{ @ } I$
JUMP UNLESS $J > K \text{ @ } I$
JUMP UNLESS $A < B \text{ @ } I$
JUMP UNLESS $A = B \text{ @ } I$
JUMP UNLESS $A > B \text{ @ } I$
JUMP UNLESS $X < Y \text{ @ } I$
JUMP UNLESS $X = Y \text{ @ } I$
JUMP UNLESS $X > Y \text{ @ } I$

15.2.11 Valamennyi típusu JUMP UNLESS utasítás egyenértékű a másik két relációjellel álló JUMP IF utasításokkal:

JUMP UNLESS $J < K \text{ @ } I$ egyenértékű a
JUMP IF $J = K \text{ @ } I$ és
JUMP IF $J > K \text{ @ } I$ utasításokkal
JUMP UNLESS $J = K \text{ @ } I$ egyenértékű a
JUMP IF $J < K \text{ @ } I$ és
JUMP IF $J > K \text{ @ } I$ utasításokkal.
JUMP UNLESS $J > K \text{ @ } I$ egyenértékű a
JUMP IF $J < K \text{ @ } I$ és
JUMP IF $J = K \text{ @ } I$ utasításokkal.

15.2.11 Az előző (15.2.11) következtében, a JUMP UNLESS utasításra ugyanazok a megszakítások érvényesek, mint a JUMP IF utasításra.

15.3 Ugrás az adatszagra lyukasztott trigger hatására. Ha egy READ utasítás hatására az adatszagról egy poz. egész szám, melyet) - zárójel követ, kerül beolvasásra, a program futása a beolvasott számnak megfelelő referenciaszámtól folytatódik.

16. Ciklusképző utasítások.

16.1 1-típusu CYCLE

Az utasítás egy utasítás sorozat többszöri végrehajtását teszi lehetővé, úgy, hogy az első végigfutás alkalmával beállított ciklusváltozó kezdőértéke a további végigfutásokban megadott lépésközzel változik, egészen egy előre megadott érték teljesüléséig. Ha a ciklusváltozó elérte az előre adott értéket, vagy túlhaladta, a ciklusmag utasításai többször nem kerülnek végrehajtásra.

- 16.1.1 Az utasítás típusai:
 $CYCLE\ I = J : K : L$
 $CYCLE\ A = B : C : D$
 $CYCLE\ X = Y : Z : V$
- 16.1.2 Az alapszó után felsorolt valamennyi operandus típusának azonosnak kell lennie.
- 16.1.3 A ciklusváltozó tetszőleges típusu változó lehet.
- 16.1.4 A ciklusváltozó kezdőértéke, a lépésköz és a ciklus végrehajtási határ változó, vagy konstans lehet.
- 16.1.5 Ha a kezdőérték, a lépésköz és a végrehajtási határérték között el-
 lentmondás van, a ciklus csak egyszer kerül végrehajtásra.
 Pl.: $CYCLE\ I = 1 : -2 : 5$, akkor a ciklus csak az $I = 1$ -gyel kerül végrehajtásra.
- 16.1.6 Ha a lépésköz 0, a ciklus végrehajtásra kerül, de egy konzol üzenet felhívja a programozó figyelmét arra, hogy esetleg végtelen sok ciklus végrehajtás következne be.
- 16.1.7 Az 1. típusu $CYCLE$ működésének vázlata:
- a) A ciklusba lépéskor:
 - I cím azonosítása
 - J cím azonosítása
 - $I = J$,
 - Ugrás a ciklusmag első utasítására.
 - b) A ciklus további végrehajtásánál:
 - I cím azonosítása
 - L cím azonosítása
 - $S = L - I$
 - K cím azonosítása
 - Ciklusfolytatás vizsgálat:
 - $K > 0$: $\begin{cases} S \geq K: \text{akkor } I = I + K \text{ és ugrás a ciklusmagba} \\ S < K: \text{akkor ugrás a REPEAT utáni utasításra.} \end{cases}$
 - $K = 0$: jelzés a konzolon, ugrás a ciklusmagba.
 - $K < 0$: $\begin{cases} S \leq K: \text{akkor } I = I + K \text{ és ugrás a ciklusmagba} \\ S > K, \text{ akkor ugrás a ciklus utáni címre.} \end{cases}$

A ciklus változó értékének helyes haladási irányát

$SIGN\ (L - I)$ mutatja.

Amennyiben az aktuális haladási irány

$SIGN\ K = SIGN\ (L - I)$, a

ciklus újból végrehajtásra kerül az

$I = I + K$ értékkel

Ha $\text{SIGN } K \neq \text{SIGN } (L-I)$, [és $\text{SIGN } K \neq 0$] a ciklusváltozó elérte a kívánt értéket és így a ciklusmag utasításait többször már nem kell végrehajtani.

Ha $\text{SIGN } K = 0$, akkor $K = 0$, a ciklus ugyanazzal a ciklusváltozó értékkel kerül végrehajtásra, mint korábban.

16.2 2-típusu CYCLE

Az utasítás egy utasítás sorozat többszöri végrehajtását teszi lehetővé, úgy, hogy a ciklusváltozó minden végrehajtás alkalmával egy lista soron következő elemének értékét veszi fel.

16.2.1 Az utasítás típusai:

$\text{CYCLE } I = J, K, L$

$\text{CYCLE } A = B, C, D,$

$\text{CYCLE } X = Y, Z, V, W$

16.2.2 Az alapszó után felsorolt valamennyi operandus típusának azonosnak kell lennie.

16.2.3 A ciklusváltozó tetszőleges típusu változó lehet.

16.2.4 Az = jel után tetszőleges számú operandusból álló lista írható.

16.2.5 A lista elemei változók, vagy konstansok lehetnek.

16.3 VARY

Az utasítás egy utasítás sorozat többszöri végrehajtását teszi lehetővé úgy, hogy az első végigfutás alkalmából beállított ciklusváltozó kezdőértéke a további végigfutásokban megadott lépésközzel változik, egészen addig, míg a ciklus megadott számszor végrehajtásra nem került.

16.3.1 Az utasítás típusai:

$\text{VARY } I = J : K : L$

$\text{VARY } A = B : C : L$

$\text{VARY } X = Y : Z : L$

16.3.2 A ciklus végrehajtások számát jelentő L operandus csak fixpontos operandus, a többi tetszőleges, de egy VARY utasításon belül azonos típusu változó illetve operandus lehet.

16.3.3 A ciklusváltozó tetszőleges típusu változó lehet.

16.3.4 A kezdő érték, a lépésköz és a végrehajtások száma változó vagy konstans lehet.

16.3.5 Ha a végrehajtások száma 0, vagy negatív szám, a ciklus egyszer sem kerül végrehajtásra.

16.4 REPEAT

Az utasítás a legutoljára megnyitott CYCLE, vagy VARY-val kezdődő ciklus lezárását jelenti.

- 16.4.1 Az utasítás típusai:
 REPEAT I
 REPEAT A
 REPEAT X
- 16.4.2 A REPEAT után irt I, A, X csak informatív jellegű, helyettük tetszőleges, de egy sorba irt jelsorozat is állhat, vagy el is maradhatnak.
- 16.4.3 A REPEAT után szóköznek mindig kell állnia.
- 16.4.4 Ciklus utasítások a fordítóprogramban beállított mélységig építhetők egymásba.
17. Programfutás megállító utasítások.
- 17.1 WAIT
 Az utasítás hatására a program futása felfüggesztődik és csak konzol-üzenettel indítható újra. Konzolüzenet hatására a program futása a WAIT-ot követő utasítástól folytatódik.
- 17.1.1 WAIT utasítás után tetszőleges jelekből álló jelsorozat írható (egy sorban).
 A jelsorozat első három jele a WAIT utasítás elérésekor kinyomtatásra kerül.
- 17.2 STOP
 Az utasítás hatására a program futása felfüggesztődik. Konzolüzenettel a program futása tetszőleges referenciaszámtól folytatható.
- 17.2.1 STOP utasítás után tetszőleges jelekből álló jelsorozat írható (egy sorban).
 A jelsorozat első három jele a STOP utasítás elérésekor kinyomtatásra kerül.
18. Formátum vezérlő utasítások.
- 18.1 LINE
 Az utasítás hatására egy "uj sor" karakter nyomtatódik ki.
- 18.2 LINES
 Az utasítás hatására az alapszót követő operandusnak megfelelő számú "uj sor" karakter nyomtatódik ki.
- 18.2.1 Az utasítás típusa:
 LINES I Az utasítás operandusa csak fixpontos lehet.
- 18.2.2 LINES után fixpontos változó, vagy 4096-nál kisebb poz. konstans állhat.
- 18.2.3 Ha a LINES utasításban a változó értéke 0, vagy negatív szám, egy "uj sor" karakter nyomtatódik ki.

- 18.2.4 Ha a kinyomtatandó soremelések száma 127-et meghaladja, a hatás ugyanaz lesz, mintha 127 "uj sor"-t kellene kinyomtatni.
- 18.3 SPACE
Az utasítás hatására egy "szóköz" karakter nyomtatódik ki.
- 18.4 SPACES
Az utasítás hatására az alapszót követő operandusnak megfelelő számu "szóköz" karakter nyomtatódik ki.
- 18.4.1 Az utasítás típusa:
SPACES I Az utasítás operandusa csak fixpontos lehet.
- 18.4.2 SPACES után fixpontos változó, vagy 4096-nál kisebb poz. konstans állhat.
- 18.4.3 Ha SPACES utasításban a változó értéke 0, vagy negatív szám, egy "SZOKÖZ" karakter nyomtatódik ki.
- 18.4.4 Ha a kinyomtatandó szóközők száma 127-et meghaladja, a hatás ugyanaz lesz, mintha 127 "szóköz"-t kellene kinyomtatni.
- 19, Kezdeti értékadó utasítás
- 19.1 SET
Az utasítással azonos betűvel kezdődő és egymásutáni index-szel jelölt változóknak lehet már a programfordítás során értéket adni.
- 19.2 Az utasítás típusai:
- | | |
|-------------------|--|
| SET I (4:9) | Egy dimenziós index-szel ellátott fixpontos változók kezdeti értékadása. |
| SET A (3:8) | Egy dimenziós index-szel ellátott dupla-fixpontos változók kezdeti értékadása. |
| SET X (12:40) | Egy dimenziós index-szel ellátott lebegőpontos változók kezdeti értékadása. |
| SET I (4:9, 3:7) | Két dimenziós index-szel ellátott fixpontos változók kezdeti értékadása. |
| SET A (3:7, 4:6) | Két dimenziós index-szel ellátott dupla-fixpontos változók kezdeti értékadása. |
| SET X (1:5, 4:10) | Két dimenziós index-szel ellátott lebegőpontos változók kezdeti értékadása. |
- 19.3 A változók után álló = jelet követve fel kell sorolni azokat az értékeket, melyeket a változók sorrendben felvesznek. Az értékeket, választja el.
- Pl.: SET I (4:9) = 3, 2, 7, 0, 2, -3.
Jelentése: $I[4] = 3, I[5] = 2, I[6] = 7$
 $I[7] = 0, I[8] = 2, I[9] = -3$

SET A (3:5, 2:4) = 3, 2, 5, 4, 10, -2, -3, 10, 0

Jelentése: $A \begin{bmatrix} 3, 2 \end{bmatrix} = 3$ $A \begin{bmatrix} 3, 3 \end{bmatrix} = 2$ $A \begin{bmatrix} 3, 4 \end{bmatrix} = 5$

$A \begin{bmatrix} 4, 2 \end{bmatrix} = 4$ $A \begin{bmatrix} 4, 3 \end{bmatrix} = 10$ $A \begin{bmatrix} 4, 4 \end{bmatrix} = -2$

$A \begin{bmatrix} 5, 2 \end{bmatrix} = -3$ $A \begin{bmatrix} 5, 3 \end{bmatrix} = 10$ $A \begin{bmatrix} 5, 4 \end{bmatrix} = 0$

- 19.4 Az index értékek és az egyenlőség jobb oldalán álló értékek csak számok lehetnek.
- 19.4 A SET utasítás nincs benne a lefordított programban. Ha SET utasítással értéket kapott változók tartalma a program futása során megváltozik, egy újabb programindítással ezek a változók nem nyelik vissza a SET-ben beállított értéküket.
20. Programfordítást befejező utasítás.
- 20.1 START
Az utasítás hatására a programfordítás befejeződik, ezért a START utasítást mindig a program utolsó utasításaként kell írni.
- 20.2 START után tetszőleges jelsorozat írható.
21. A programírás formátuma.
- 21.1 Minden utasítás előtt tetszőleges számú "szóköz" írható.
- 21.2 Minden alapszó csak nagybetűkből állhat.
- 21.3 Minden utasítás sor végét "új sor" karakter jelzi.
- 21.4 SETS SETD SETV SETF SETR JUMP SUBR IF
UNLESS VARY CYCLE REPEAT READ INPUT PRINT
OUTPUT CHECK LINES SPACES TITLE READER
PUNCH SET START alapszavak után kötelező "szóköz"-jelet tenni.
- 21.5 EXIT LINE SPACE után nem szabad "szóköz" jelet tenni.
- 21.6 WAIT STOP után lehet "szóköz" jelet tenni, de nem kötelező.
- 21.7 Az utasításokban a fentieket kizárva, sehová sem szabad "szóköz" jelet tenni.
22. Az adatszalog.
Az adatszalongon csak számok és a "trigger"-t jelző) karakter lehet. Az adatokat "szóköz" és "új sor" választja el egymástól. A tizedes-pontot . jelzi.

9. HIBAJELZÉSEK

Az EMG 803 autokód fordítóprogram fordítás közben szintaktikus ellenőrzést végez. Az első megtalált szintaktikus hiba az autokód program bináris szalagra való fordítását automatikusan letiltja. A fordítóprogram jelenlegi formájában utasításonként legfeljebb egy szintaktikus hiba észlelésére képes. Ha egy utasításban a fordítóprogram ellenőrző része egy hibát megtalál, az utasítás többi részét szintaktikailag már nem ellenőrzi. A talált hibák kilistázása a sornyomtatón megy végbe, úgy, hogy előbb az utasítás sorszáma majd a hiba tipusszáma kerül kinyomtatásra. A kinyomtatott utasítás sorszám alapján a legépeltebb programban könnyen megtalálható a hibás utasítás. A program utasításait a következőképpen kell számolni.

0-dik utasításnak számít a teljes programdeklaráció. Ez magában foglalja a SETS, SETV, SETD, SETF és SETR-rel kezdődő, a SETR kivételével akár többször is előforduló deklarációs utasításokat.

A programdeklaráció utolsó sorát, a SETR-rel kezdődő referenciaszám deklarációt, követő utasítás viseli az 1-es számot.

Minden további utasítás eggyel nagyobb sorszámmal szerepel, mint az előző. Ezáltal a hibás utasítás és így magának a hibának megtalálása is lényegesen egyszerűbb feladat, mint pl. az ICT algoritmusban a hiba megkeresése, ahol a kilistázott utasításszámok nem pontosan fedik a program utasításainak számát.

Az EMG 830 autokód fordítóprogram az alábbi szintaktikus hibákhoz a felüntetett hibaszámot írja ki.

Hiba azonosító száma

A hiba megnevezése

- | | |
|----|--|
| 1. | Hibás a típusdeklaráló szó. |
| 2. | A program ezen a helyén nem megengedett jel. |
| 3. | Ismétlődő változó deklarációban. |
| 4. | 2-nél több dimenziós tömb a deklarációban. |
| 5. | Betelt az R mező. |
| 6. | A deklaráltnál nagyobb referenciaszám. |
| 7. | Ismétlődő referenciaszám. |

Hiba azonosító száma

A hiba megnevezése

- | | |
|-----|--|
| 8. | Hivatkozás a kapcsolószó 21-nél magasabb sor-számu (nem létező) bitjére. |
| 9. | Ciklusnyitás nélküli REPEAT utasítás. |
| 10. | Hibás műveleti jel. |
| 11. | Hibás típusu változó. |
| 12. | LINES vagy SPACES után túl nagy szám. |
| 13. | Hibás függvényszó, vagy az utasítás típusa alapján nem megengedett függvény. |
| 14. | Hibás típusu függvény argumentum, vagy hiányzik a függvényargumentum. |
| 15. | Nem megengedett számforma, vagy túl hosszú szám: |
| 16. | Az utasítás típusa alapján nem egyeztethető számforma. |
| 17. | Betelt az F mező. |
| 18. | Az utasításban beolvasott index típusa nem felel meg a deklarált típusnak. |
| 19. | A beolvasott változó nincs deklarálva. |
| 20. | Helytelen alapszó. |
| 21. | A program befejezése után lezáratlan ciklus maradt. |

A hiba megnevezések bővebb ismertetése:

1. A hiba csak a deklarációban jelentkezhethet akkor, ha a változók típusát meghatározó SETS, SETV, SETD, vagy a függvényeket, illetve a felhasználható referenciaszámok maximumát kijelölő SETF illetve SETR helyett más szó áll.
A SETS, SETV, SETD, SETF, SETR azonosítása a soreljétől számított első S-től kezdődik, az utána következő két jel hatástalan, a deklarációs típus azonosítása a következő jel alapján következik be. Amennyiben ez a karakter nem S, V, D, F, vagy R, az 1-es típusu hiba jelentkezik.
2. Ez a hibajelzés következik be szintaktikusan helytelen jel beolvasása esetén. Az alábbiakban felsorolunk néhány esetet:

SETS X [3]

A SETS és az X [3] között nincs szóköz.

SETR- A

A SETR után csak számjegy következhet.

+ A=2	Utasítás elején csak számjegy vagy betű állhat.
A+B=C	Utasítás bal oldalán nem állhat műveleti jel.
2+A=C	Utasítás elején beolvasott számot csak) zárhat.
AB] =2	Kimaradt a [zárójel
A [B=C	Kimaradt a] zárójel
CYCLE I: 1:1:3	CYCLE I után nincs = jel
VARY I=3: 2=4	A 2=4 között az = jel hibás
VARY I=3: 2-4	Kimaradt a : jel.
INPUT 3	INPUT után csak változó állhat.
INPUT A+B	INPUT után csak egy változó állhat.
OUTPUT I'	A ' hibás az OUTPUT-ban.
I=A [2,K,4]	A második , hibás karakter.
JUMP IF A+B=C @ 3	JUMP utasítások bal oldalán egy és csak egy változó állhat.
REPEAT I	A REPEAT után hiányzik a szóköz.

3. Ez a hiba csak a deklaráció beolvasásánál jelentkezik, akkor, ha egy beolvasott változó azonosításánál kiderül, hogy akár ugyanabban a típusban, akár más típusban korábban már deklaráltuk.
4. Ez a hiba a deklarációban akkor jelentkezik, ha egy beolvasott változó indexében 2 vagy több , karakter van.
5. Ez a hiba túl hosszú program esetén jelentkezik, ha a programfordítás során kiderül, hogy futtatás közben a program nem fog beférni a memória rendelkezésére álló részébe.
6. Utasítás elején beolvasott referencia szám nagyobb, mint amit a SETR után beolvasott maximális referenciaszám lehetővé tesz.
7. A hiba akkor jelentkezik, ha egy programon belül egy adott referenciaszám másodszor is ki van téve egy utasítás elején.
8. A JUMP IF i@K típusu utasításban az i helyett csak 1 és 21 közé eső számérték állhat, mivel ez az utasítás a kapcsolószo i-edik bitjét vizsgálja meg.
9. A hiba akkor fordul elő, ha a fordítás folyamán egy beolvasott REPEAT utasításhoz a fordítóprogram nem tud hozzárendelni egy, még REPEAT utasítás nélküli CYCLE vagy VARY utasítást.
Egy REPEAT utasítás mindig a legutoljára megnyitott ciklust zárja le, függetlenül attól, hogy a ciklusváltozó és a REPEAT-ot követő változó megegyezik-e. Mivel a REPEAT utasítást követő változót a fordítóprogram nem azonosítja, ezért a REPEAT után tetszőleges string helyezhető el, anélkül, hogy hibajelzés következne be.

Ugyancsak nincs hibajelzés, ha a REPEAT-ot követő szóköz után nincs változó, hanem mindjárt az "ujsor" karakter következik.

Előfordulhat a 9 számú hiba akkor is, ha valamelyik korábbi CYCLE, vagy VARY utasításban volt hiba. Ebben az esetben ugyanis a hibás CYCLE vagy VARY miatt az utasítás nem került lefordításra, tehát az utoljára beolvasott REPEAT-nél kijelzett hiba a CYCLE-nél ill. VARY-nál észlelt hiba következménye.

Hibajelzést okoz, ha egy ciklust az alábbiak szerint programozunk:

5) REPEAT I

JUMP @6
CYCLE I=1:1:2
READ AI
JUMP @5

6)

Ebben az esetben a REPEAT I beolvasásánál a fordítóprogram nem találja meg a később beolvasásra kerülő CYCLE utasítást, és ezért hibajelzést ad.

Hibásan fog működni a következő program:

VARY X=10: -1:4

JUMP @3

5) REPEAT I

JUMP @6
CYCLE I=1:1:2

JUMP @5

3) REPEAT X

6)

Ebben az esetben a fordítóprogram nem jelez hibát, de a programot az alábbi programként értelmezi és fordítja:

VARY X = 10: -1:4

JUMP @3

5) REPEAT X

JUMP @6
CYCLE I=1:1:2

JUMP @5

3) REPEAT I

6)

Hasonló módon hibás a következő program:

```
VARY K = 1:1:10  
CYCLE I=1:1:5
```

REPEAT K

REPEAT I

Ezt a programot a fordítóprogram

```
VARY K=1:1:10  
CYCLE I=1:1:5
```

REPEAT I

REPEAT K programnak értelmezi és fordítja.

10. A hiba egy utasításban előforduló hibás műveleti jelet jelez. Ez a hiba csak akkor fordulhat elő, ha egy utasításon, illetve egy önálló utasítás részen belül az aritmetikai műveleti jelek (+, -, *, /) keverednek a logikai - léptetési műveleti jelekkel (!, %, &, 'a', 'b', 'c',')

Önálló utasításrésznek számít a JUMP típusu utasítások ω jel után következő része.

A kijelzett hiba után a programozó feladata eldönteni, hogy a műveleti jelek típuskeveredése hibáját melyik műveleti jel okozta.

Az $A=B+C*D$ 'a' '5 & C [I+K, 4] utasításban a hibajelzés a % logikai műveleti jel hatására következik be, annak ellenére, hogy az utasítás típusa alapján a + aritmetikai műveletijel a hibás.

10 hibajelzés következik be akkor is, ha logikai - léptetési utasítás műveleti jelei esetén a ! (negáció) műveleti jel nem közvetlenül az = jel után áll.

11. sorszámú hiba hibás típusu változót jelent. Ez a hibajelzés akkor következik be, ha egy utasításon, vagy önálló utasításrészben belül szereplő változók nem azonos deklarációs csoportba tartoznak.

A fordítóprogram az utasítások típusát mindig az első beolvasott változó típusa alapján állítja be. Ha valamelyik, az utasításban később beolvasásra kerülő változó típusa nem egyezik meg a legelőször beolvasott változó típusával, hibajelzés következik be.

Hibajelzés következik be akkor is, ha a

```
VARY I=1:1:K típusu utasításokban a  
K helyén nem fixpontos változó áll.
```

Ugyanígy, csak fixpontos változók állhatnak az indexekben (fixpontos válto-

zónak számát indexben az INT X, ahol X lebegőpontos változó) JUMP típusu utasításokban a @ jelet követve, SUBR alapszó után, a kapcsolószó bitjeit vizsgáló JUMP IF A@ K utasításban, a SPACES, a LINES, az INPUT, az OUTPUT, a PUNCH, és a READER alapszavak után.

12. sorszámú hibajelzés LINES és SPACES alapszavak után következhet be, ha a LINES-t vagy SPACES-t 4095-nél nagyobb szám követi.
13. A hibajelzés hibás függvény-szót, illetve az adott utasítás típusa alapján nem megengedett függvényt jelez. Hibásnak értelmezi a fordítóprogram a függvényszót, ha a függvényszó első három betűjében van hiba, és ha az első három betű hibátlan ugyan, de a függvényszó nem az előírt számú betűből áll. Hibás tehát, ha INT helyett ENT áll, vagy SQRT helyett SQRTT, de nem okoz hibajelzést és a program is jól működik, ha SQRT helyett SQRR áll.
Ugyancsak hibás az A=SQRT X utasítás, ha az A változó nem lebegőpontosnak van deklarálva.

Általában a 13 sorszámú hibajelzés egy tetszőleges függvény pl.

A=SQRT X típusu utasítás bekeretezett részében talált hibarészre vonatkozik.

14. A hibajelzés hibás típusu függvényargumentumot jelent. Ilyen lehet az A=SQRT X, ha X nem lebegőpontos változó. Általában a 14 sorszámú hibajelzés egy tetszőleges függvény, pl. SQRT X típusu utasítás bekeretezett részében talált hibára vonatkozik.
15. A hibajelzés nem megengedett számformára, vagy az utasítás típusán belül túl hosszú számra utal.
Nem megengedett számforma pl.
.32 (A . előtt 0-nak kell állnia)
0.3.5 (Csak egy tizedespont állhat egy számban.)
Túl hosszú szám a 1 2 3 4 2 5 8 4 2 5 ha az utasítás típusa fixpontos.
16. A hibajelzés az utasítás típusával össze nem egyeztethető számformát jelent.
Pl. I=2.3 ahol I fixpontos szám,
mivel a tizedespont csak lebegőpontos esetben van megengedve.
17. Ez a hibajelzés akkor következik be, ha a memória F mezőben rendelkezésre álló része kevésnek bizonyul a program számára. Az F mezőbe kerülnek a deklarált változók rekeszei, a referenciátáblázat, a programnak literálként nem értelmezhető konstansai, (minden előforduló konstans csak egyszer) és a TITLE utasítást követő karakterek.
18. A hibajelzés akkor következik be, ha egy változó valamely program utasításban beolvasott indexe nem felel meg a változó deklarációjában feltüntetett indexének.

Pl. SETS I [3] esetén egy utasításban.

A = I [3, 2]

vagy SETD K [3, 4] esetén egy utasításban:

B = C + K [3]

19. A hibajelzés nem deklarált változót jelent.

Pl. SETS IJK [3]

SETR 10 deklaráció esetén egy programutasítás:

I = A + K [1]

Bekövetkezik ez a hibajelzés a deklarációban elkövetett olyan hiba esetén is, melyet az 1, 2, 3, 4, számú hibajelzések nem jeleztek.

Ha pl. a deklaráció:

SETD T

CETV A [10] B [8]

SETR I

SETR 10 akkor az A [10] B [8]

változókat a fordítóprogram nem értelmezi deklarált változóknak, mivel a típus azonosítást, majd ezt követve a változók azonosítását az adott deklarációs sorban beolvasott első S-betű megtalálásakor kezdi el.

A fenti program deklarációt a

fordító program SETD T

SETS I

SETR 10

programdeklarációnak érzékeli.

20. A hibajelzés programutasítás elején beolvasott hibás alapszót jelent. Hibásnak érzékeli a fordítóprogram azt az alapszót, melynek első három betűjének valamelyikében van a hiba, és ha az első három betű hibátlan ugyan, de az alapszó nem az előírtak megfelelő számú betűből áll.

Hibás utasítás tehát, ha VARY helyett WARY-t vagy UNLESS helyett UNLES-t írunk, de nem okoz hibajelzést, és a program is hibátlan, ha UNLESS helyett UNLES-t írunk.

20 számú hibajelzést kapunk ha a programdeklaráció:

SETS I

SETR 5

SETV A esetén, A fordítóprogram

ugyanis a programdeklarációt a SETR 5-tel lezártnak tekinti, így a SETV-et alapszónak azonosítja, de ilyen alapszó nincs a megengedett alapszavak között.

21. Ezt a hibajelzést csak a START utasítás beolvasása után kaphatjuk. Azt jelzi, hogy a programban lezáratlan ciklus maradt. Ezt okozhatta, hogy a programozó elfelejtette a REPEAT utasítást leírni, de okozhatta az is, hogy egy REPEAT utasítás hibás volt, és ezt a fordítóprogram nem érzékelt.

Hibajelzések programfutás közben

1. Indexhiba
2. CYCLE utasításban a lépésköz 0.
3. Hibás referencia szám.
4. Nem létező referenciaszám.
5. Kapcsolószó hivatkozás hibás.
6. Megengedettnél több egymásba épített szubrutin.
7. Hibás EXIT
8. Nem megengedett szám.
9. Tipushibás szám.
10. Hibás periféria hívás.

A hibajelzések részletes ismertetése

1. Indexhiba jelentkezik, ha egy változó kiszámított indexe hibás.
Hibás a negatív index, és az olyan index, melyet az alapváltozóhoz hozzászámítva, a kapott cím a deklarált változók rekeszein kívül esik.

Ha pl. a deklaráció:

SETS A [20] B [20]

SETD C 5

SETV DE 100

A deklaráció alapján	A [20]	21 rekeszt
	B [20]	21 rekeszt
	C [5]	12 rekeszt
	D	2 rekeszt
	E[100]	202 rekeszt

foglal el így a teljes memória szükséglet: 258 rekesz.

A deklaráció alapján a programban a változókat a következő maximális indexértékekkel lehet használni, szintaktikus hiba nélkül:

A [257,] B [236,] C [107,] D [101,] E [100]

2. CYCLE utasításban a lépésköz 0.
Ebben az esetben a program futás tovább folytatódik.
3. Hibás referenciaszám.
Ez a hiba akkor jelentkezik, ha 0, negatív, vagy a deklarálnál nagyobb referenciaszámra hivatkozunk.
4. Nem létező referencia szám.
Ez a hiba akkor jelentkezik, ha a hivatkozott referenciaszám nem található a programban.

5. Hiba a kapcsolószo hivatkozás.
A hiba akkor jelentkezik, ha a kapcsolószo 0, negativ, vagy 21-nél nagyobb sorszámu bitjére hivatkozunk.
6. A megengedettnél több szubrutin van egybeépítve.
A hiba akkor jelentkezik, ha az egymásba épített szubrutinok száma meghaladja a 10-et.
7. Hiba EXIT utasítás.
A hiba akkor jelentkezik, ha SUBR utasítás nélkül kerül sor egy EXIT utasítás végrehajtására.
8. Nem megengedett szám.
A hiba olyan esetben jelentkezik, ha adatszagról nem megengedett típusu szám, pl. fixpontos törtszám kerül beolvasásra.
9. Tipushiba szám.
A hiba akkor jelentkezik, ha adatszagról a változó típusának nem megfeleltethető szám került a memóriába.
10. Helytelen periféria használat.
A hiba akkor jelentkezik, ha PUNCH utasítás után nem 1, vagy 2, READER utasítás után nem 1, 2, 3 vagy 4 áll.

10. FELADATMEGOLDÁSOK

Példák:

Egy adott feladatot többféleképpen is be lehet programozni. Egy programozó egy feladat programozása közben amellet, hogy jó végeredmény elérésére törekszik, több programozási szempontot is szem előtt tarthat.

Ilyen szempontok lehetnek:

A program könnyű áttekinthetőségének biztosítása.

Törekvés rövid, tömör program írására.

Törekvés olyan program írására melyet az adott feladatot a lehető legrövidebb idő alatt lehet lefuttatni.

Ha az adatok nem férnek el a memóriában olyan program írása, mellyel az adatokat esetleg kettesével – vagy hármassával lehet egy memóriarekeszbe elhelyezni, és így a feladat megoldhatóságát biztosítani.

A felsorolt szempontok természetesen nem meritik ki az összes lehetőséget, melyre a programozó a program készítése során figyelhet, vagy mint az utójára felsorolt esetben figyelnie is kell.

Azonos szempontok szem előtt tartása mellett is egy elkészült program magánvisel olyan ismertető jeleket, melyek a program írójára jellemzők.

Éppen ezért, ha egy feladatot több programozónak adunk oda programozni, az elkészült programok nem lesznek azonosak. Még a legegyszerűbb programok esetében is a programozónak igen sok lehetősége van a változók vagy a referenciaszámok megválasztására, ez egyes részeredmények ellenőrzésére, vagy akár a nyomtatási formátum megválasztására. Bonyolultabb feladatok esetén az eltérések lehetőségeinek száma megsokszorozódik. Nagy gyakorlattal rendelkező programozók programjait szinte ránézéssel meg lehet különböztetni egy kevés gyakorlattal rendelkező programozó munkájától. Ugyanaz a programozó is, ha előveszi egy régebben készített munkáját, rájön, hogy egyszerűbben, vagy legalább is másképpen is meg lehetett volna oldani a feladatot. A jegyzetben szereplő feladatok megoldására többféle megoldást dolgozhat ki. Ezzel éppen azt akartuk megmutatni, hogy különböző szinteken álló

, vagy különböző szempontokat szem előtt tartó programozók egy feladat beprogramozására milyen egymástól eltérő módokat alkalmazhatnak. Természetesen, mint korábban elmondottuk, ezzel korántsem meritettük ki az összes egymástól lényegileg eltérő lehetőségeket. Minden program előtt közöljük a feladat megoldásának blokkdiagramját. Felhívjuk a programozók figyelmét arra, hogy érdemes egy feladat programozásához blokkdiagramot készíteni. Érdemes azért, mert blokkdiagram alapján a programozói munka egybe egyszerű-

sődik, nagyobb az áttekintési lehetőség, világosabbak a kapcsolatok, tisztán látszanak az elágazási és csatlakozási pontok, a ciklusvégződés stb. De érdemes azért is, mert blokkdiagram készítés közben alaposan át kell gondolni a feladatot és így már akkor kiderülnek a feladat esetleges homályos, nem kellően tisztázott részei. Javítás vagy a feladat kibővítése a blokkdiagram alapján sokkal egyszerűbb, szemléletesebb, mint blokkdiagram nélkül, amikor is közvetlenül a programban kell módosításokat végrehajtani. Szokjunk hozzá azért is a blokkdiagram készítéséhez, mert így elkerüljük azt a hibát, amibe nagyon sok programozó beleesik, ti. azt, hogy a feladatot rögtön kapásból elkezdje programozni és csak később jön rá, hogy a kapott eredmény azért nem kielégítő, vagy bizonyos esetekben a program egyáltalán nem is működik, mert bizonyos szempontokat, kikötéseket figyelmen kívül hagyott. Blokkdiagram hiányában igen nagy fáradtságába kerül a program javítása, módosítása, vagy esetleg teljesen új program írása. Mindezt a sok időt és fáradtságot igénybevevő programmódosítást el lehet kerülni ha egy feladatot csak akkor programozunk már be, ha a blokkdiagramot minden irányban végigkövetve megállapítjuk, hogy valóban a kitűzött feladat szerint működik.

A kidolgozott példákat részletes magyarázattal láttuk el. Ezzel azt szeretnénk elérni, hogy egyes programozási fogások világossá váljanak a jegyzet olvasója előtt és ha fejben tartja azokat a programozási fogásokat, adott esetben vissza tudjon lapozni a jegyzet megfelelő helyére.

1. Feladat:

Másodfoku egyenlet megoldása

Oldjuk meg az $ax^2+bx+c=0$ másodfoku egyenletet, ahol az a, b, c számokat adatszálagról kell beolvasni. Az egyenlet megoldását adó

$$X_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

képletet csak akkor lehet alkalmazni, ha $a \neq 0$. Ha ugyanis $a=0$ lenne, és mivel " a " a tört nevezőjében szerepel, az osztást nem tudjuk elvégezni. $a=0, b \neq 0, c \neq 0$ esetében a megoldás az, hogy az egyenlet

$$bx+c=0, \text{ alaku és innen}$$

$$X = -\frac{c}{b} \text{ az egyenlet egyetlen megoldása.}$$

Ha $a=0$, és $b=0, c \neq 0$ akkor az egyenletnek nincs megoldása.

Ha $a=0, b \neq 0, c=0$ akkor $x=0$ az az egyetlen megoldás.

Ha $a=0, b=0, c=0$, nincs értelme megoldásról beszélni, mivel végtelen sok megoldás van.

Ha $a \neq 0, b=0, c \neq 0$, akkor az egyenlet

$$ax^2+c=0 \text{ alaku és a}$$

megoldás közvetlenül az

$$x_{1,2} = \pm \sqrt{-\frac{c}{a}}$$

szolgáltatja.

Ha $a \neq 0$, $b \neq 0$, de $c=0$ akkor az egyenlet $ax^2+bx=0$, vagyis

$$x(ax+b) = 0,$$

azaz $x_1=0$, és $x_2 = -\frac{b}{a}$

Ha $a \neq 0$, de $b=0$ és $c \neq 0$, akkor

$$ax^2=0 \text{ és így } x_1=x_2=0$$

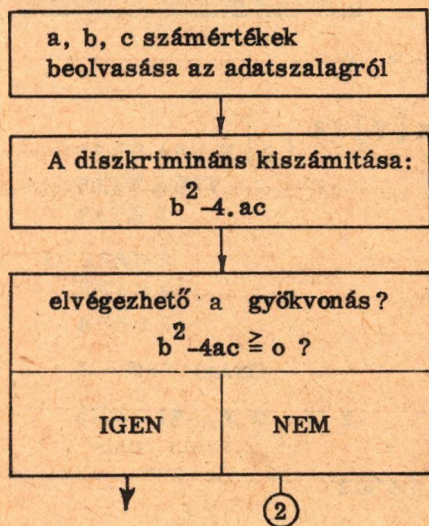
láthatjuk tehát, hogy az igen egyszerűen megoldható másodfoku egyenlet programozása is milyen bonyolult, ha minden lehetséges előfordulható esetre figyelemmel kell lennünk. Egyszerűsíthetjük viszont a megoldást, ha már eleve csak olyan a , b , c számhármass esetében végezzük el a számítást melyek egyike sem 0.

Megoldás:

Tegyük fel, hogy olyan adatok esetében kell a másodfoku egyenlet megoldását alkalmazni melyek között egyetlen 0 sincs. Kössük ki továbbá, hogy csak a valós megoldásokat keressük.

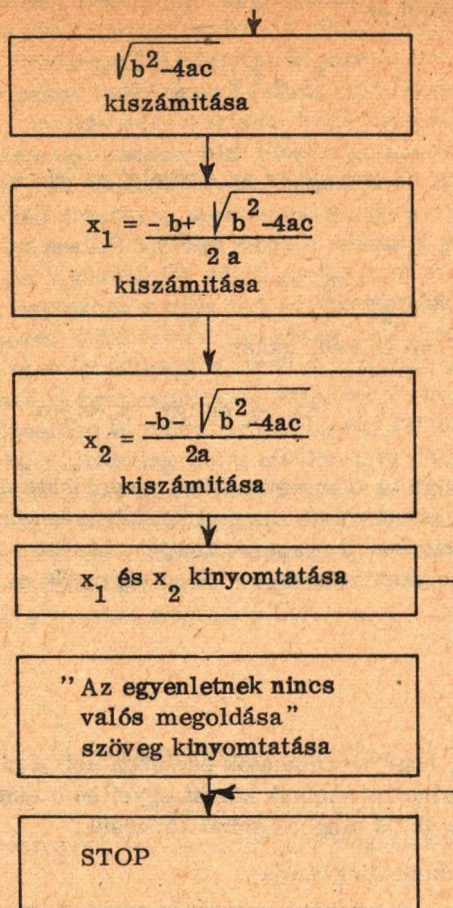
A program blokkdiagramja:

1.



2.

3.



A program:

```

SETV ABC X [ 2 ] YZ
SETF SQRT
SETR 3

```

```

1) READ A
   READ B
   READ C

```

```

X=A * C
X=4 * X
Y=B * B
Y=Y-X

```

```

JUMP IF Y < 0 2

```



```

Y=SQRT Y
Z=2 * A
X=-B+Y
X [1] =X/Z
X=-B-Y
X [2]=X/Z

PRINT X [1]
PRINT X [2]
JUMP @ 3

```

2) TITLE AZ EGYENLETNEK NINCS VALÓS MEGOLDÁSA

3) STOP
START 1

A feladatot a legegyszerűbb autokód utasítások felhasználásával programoztuk. Mint látható az aritmetikai utasítások jobb oldalán csak egy művelet szerepel. Ilyen autokód műveleti utasítások vannak megengedve az ELLIOTT 803 B autokódjában. Az EMG 830- autokód tetszőszerinti hosszúságú aritmetikai utasítást is megenged, ha ezt a lehetőséget kihasználjuk a program lényegesen egyszerűsödik.

A programban A, B, C változóknban tároljuk az a, b, c számértékeket,

x_1 és x_2 az eredményt tárolja, míg

X, Y, és Z munkarekesz a kiszámított részeredmények megőrzésére.

Valamennyi felhasznált változót lebegőpontosként deklaráltuk, mivel egy utasításon belül a típusok keveredése nincs megengedve, és a négyzetgyökvonás (SQRT) kivezet az egész (fixpontos) számok köréből.

2. program:

```

SETV ABC X [2]
SETF SQRT
SETR 3

1) READ A
   READ B
   READ C

X = B*B-4*A*C

JUMP IF X < 0 @ 2

```



```

X[1] = -B + SQRT X
X[1] = X[1] / 2 / A
X[2] = -B - SQRT X
X[2] = X[2] / 2 / A

PRINT X [1]
PRINT X [2]
JUMP @ 3

```

2) TITLE AZ EGYENLET NINCS VALÓS
MEGOLDÁSA

3) STOP
START 1

A 2. program $x = B \pm \sqrt{B^2 - 4AC}$
JUMP IF X < 0 @ 2

sorait akár fel is cserélhettük volna:

```

JUMP IF  $B^2 - 4AC < 0$  @ 3
 $x = B \pm \sqrt{B^2 - 4AC}$ 

```

esetben azonban a feltételes ugró utasításban és az X-nek értéket adó aritmetikai utasításban is külön-külön rá kell számolni a

$$B^2 - 4AC - t.$$

Az utóbbi variáció nagyobb memóriafelhasználást is jelent és lassabb is az első megoldásnál.

A négyzetgyökvonást (SQRT X) kétszer egymás után végrehajtani viszont nem gazdaságos azért, mivel egy függvény utasítás igen sok gépkód műveletből tevődik össze, és így egy-egy feleslegesen alkalmazott függvényutasítás a program futását igen csak lelassítja. Többszöri felhasználás esetén érdemes tehát az SQRT X-et egy segédváltozóban elhelyezni.

3. program:

```

SETV ABC X [2]
SETF SQRT
SETR 3

1) READ A
   READ B
   READ C

 $X = B \pm \sqrt{B^2 - 4AC}$ 

JUMP IF X < 0 @ 3

X = SQRT X

```


$X[1] = -B/2/A + X/2/A$

$X[2] = -B/2/A - X/2/A$

PRINT X [1]

PRINT X [2]

JUMP @ 3

2) TITLE Az egyenletnek nincs valós megoldása

3) STOP

START 1

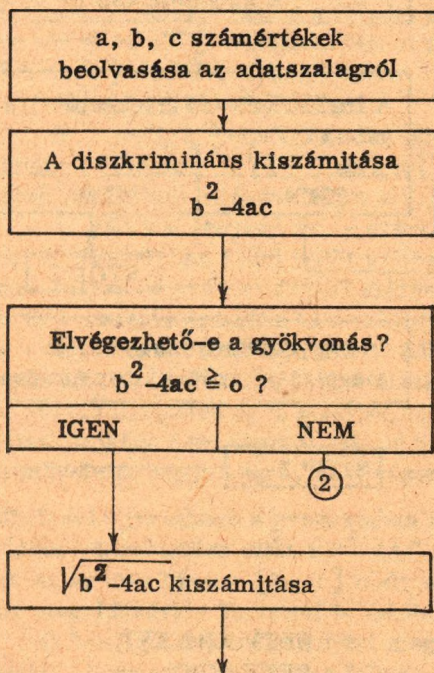
Az SQRT X-hez hasonlóan a tört 2A nevezőjének kiszámítását is el lehetett volna külön végezni.

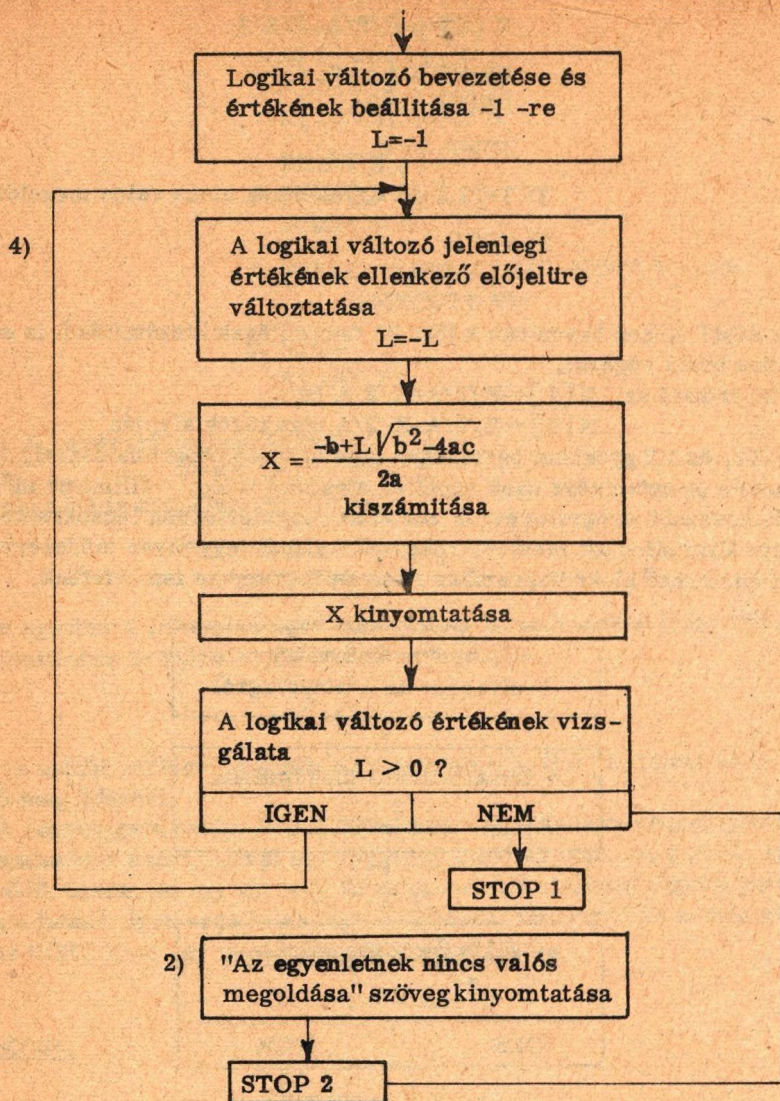
Mint látható az $X[1] = -B/2/A + X/2/A$ és az

$X[2] = -B/2/A - X/2/A$ utasítások alapján

az X1, és X2 gyököket egymáshoz igen hasonló programozással, (ebben az esetben) a programrész csak egyetlen utasítást jelent állítottuk elő. Hasonló, de hosszabb programrészek esetében nagymértékben lecsökkenthetjük a program által elfoglalt memóriarekeszek számát egy olyan módszerrel, melyet a következő blokkdiagramban illetve programban ismertetünk.

1)





4. program

SETV ABC XYL
SETF SQRT
SETR 4


```

1) READ A
   READ B
   READ C

```

$$X = B \times B - 4 \times A \times C$$

```

JUMP IF X < 0 @ 2

```

```

X = SQRT X
L = -1

```

```

4) L = -L
   Y = -B/2/A + L * X/2/A
   PRINT Y
   JUMP IF L = 1 @ 4
   STOP 1

```

```

2) TITLE AZ EGYENLETNEK NINCS VALÓS
   MEGOLDÁSA

```

```

STOP 2
START 1

```

A $\sqrt{B^2 - 4AC}$ diszkrimináns kiszámításáig a feladat programozásának 4. változata semmiben sem tér el az előző változatoktól. A megoldó képlet program sora viszont az előző $X[1] = -B/2/A + X/2/A$ illetve

$$X[2] = -B/2/A - X/2/A \text{ -ről}$$

$$Y = -B/2/A + L \times X/2/A \text{ -ra változott.}$$

Az, hogy az eredmény jelölésére $X[1]$ illetve $X[2]$ helyett az Y változót alkalmaztuk, ez csak jelölésbeli eltérést jelent a 3., illetve az ezt megelőző változatok között. A 4. változat lényege az, - ezért mutattuk be ezt a módszert, - hogy egy új, a programban L -lel jelölt változóval, illetve a bevezetett változó értékének helyes megválasztásával elérhetjük, hogy ugyanaz a

$Y = -B/2/A + L \times X/2/A$ utasítás számítja ki a másodfoku egyenlet mindkét gyökét. Kövessük végig a 4. változat utasítássorait: A három együttható (a C -t mint X^0 együtthatóját foghatjuk fel) beolvasása után a diszkrimináns kiszámítása következik, majd az ezt követő feltételes ugró utasítás két ágra bontja a programot. Ha mármost a diszkrimináns nem negatív, akkor a következő utasítássor a négyzetgyökvonás műveletét hajtja végre, és a végeredményt, vagyis a

$\sqrt{B^2 - 4AC}$ számértékét elhelyezi ugyanabban a programban X -szel jelölt változóban, melyben az utasítássor végrehajtása előtt a $B^2 - 4AC$ számértéke volt.

Ez után következik az az utasítássor [$L = -1$] mely első pillantásra nincs összefüggésben a feladat megoldásával. Ez annál is inkább így látszik, mivel a következő utasítássor [4] $L = -L$] hatására mindjárt meg is változtatjuk az L változó számértékét anélkül, hogy az L eredetileg felvett -1 értékével bármilyen számítást is végrehajtottunk volna.

Az L értéke tehát most 1. Ez pedig azt jelenti, hogy a gyököt kiszámító $Y = -B/2/A + L \times X/2/A$ utasítássorban az L -et az 1 értékkel kell számításba venni.

Igy $Y = -B/2/A + L \cdot X/2/A$ egyenértékű az $Y = -B/2/A + X/2/A$ utasítással. A gyök kinyomtatása után [PRINT Y] egy ugró utasítással gondoskodunk a gyök újbóli kiszámításáról. A második gyök kiszámításához viszont -ként kell figyelembe venni $Y = -B/2/A + L \cdot X/2/A$ utasításokban szereplő + előjelet. Így mielőtt a fenti utasítás végrehajtására sor kerülne az L értéket 1-ről -1-re kell változtatni. Ezt pedig elérjük, ha az ugróutasítást a 4)-referenciaszámra az $L = -L$ utasításához "cimezzük". A második végrehajtás során tehát az

$Y = -B/2/A + L \cdot X/2/A$ utasítás egyenértékű az

$Y = -B/2/A - X/2/A$ utasítással.

Most még csak arról kell gondoskodni, hogy a gyököt kiszámító programrész többször már ne kerüljön végrehajtásra. Eszerint a 4)-referenciaszámra ugrató utasításnak feltételes ugró utasításnak kell lennie, mégpedig olyan feltétel szerint, mely ugrást eredményez a programrész első végrehajtása során, de a második végrehajtás esetén már nem. Ezt a feltételt pedig kielégíti a JUMP IF L=1 @ 4 utasítás.

Az L változó értékét természetesen másképpen is megválaszthattuk volna. Ha például valakinek az tűnik fel, hogy az

$$X[1] = -B/2/A + X/2/A \text{ és}$$

$$X[2] = -B/2/A - X/2/A$$

utasításokban a jobb oldal minden egyes tagjának nevezője 2A, ezt az észrevételt felhasználhatja az L értékének beállítására. Ebben az esetben, --- a 4. program blokkdiagramját lényegében változatlanul hagyva, mivel a blokkdiagram blokkjai változatlanok maradnak, csak az L értéket beállító blokk változik L=-1 -ről $L = -\frac{1}{2A}$ -ra ---- a program az alábbi lehet.

5. program

```
SETV ABC X
SETF SQRT MOD
SETR 4
```

```
1) READ A
   READ B
   READ C
   X=B*B-4*A*C
   JUMP IF XLo 2
   X=SQRT X
   L=-1/2/A
```

```
4) L=-L
   Y=-B*MOD L+L*X
   PRINT Y
   JUMP IF L=1 @ 4
   STOP 1
```


2) TITLE AZ EGYENLETNEK NINCS VALÓS MEGOLDÁSA,

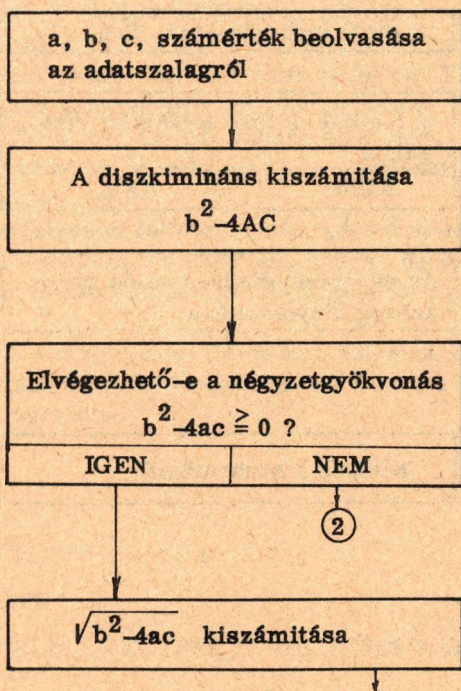
STOP 2

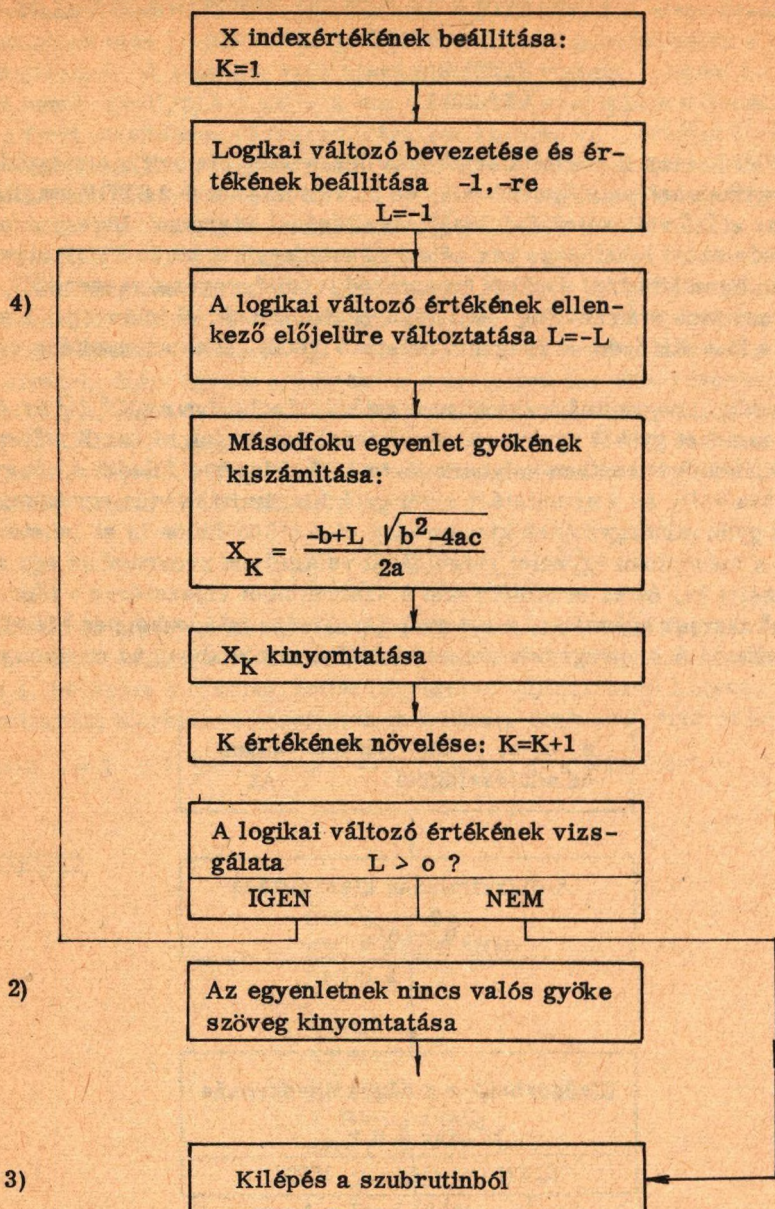
START 1

A MOD L azaz L abszolút értékének bevezetése biztosítja a megoldó képlet első tagjának helyes előjelét. A 4. és 5. változatban 2-2 STOP utasítás szerepel az előző változatok 1-1 STOP utasításával szemben. Ha egy programnak több befejezési lehetősége van célszerű mindegyiket külön STOP utasítással ellátni. Nyomkövetési okokból az autokód fordítóprogram megengedi, hogy a STOP utasítások után "string"-et, jelsorozatot írjunk. E jelsorozat első három jelét a konzolirógép a program befejező STOP-jának azonosítása céljából kiírja.

Az eddigi programváltozatokban a gyökök tárolására az $X[1]$ és $X[2]$, illetve ha mindkét gyököt ugyanazon utasítással számoltuk ki, az X változót használtuk. Mindkét esetben helyesen jártunk el, hiszen a feladat a két gyök kinyomtatása volt, és a nyomtatást a két gyök kiszámítása után együttesen, vagy a két gyök mindegyikének kiszámítása után külön-külön is el lehetett végezni. Ha a másodfoku egyenlet megoldását valamilyen programban egy szubrutin számolja ki, és az eredményeket a szubrutinból visszatérve a főprogramban fel akarjuk használni, a két gyök tárolására mindenképpen két változót kell biztosítani. A 4. programváltozat blokkdiagramja abban az esetben így módosul.

1)





6. program:

```

1)  READ A
    READ B
    READ C
    X=B X B-4 X A X C
    JUMP IF X < 0 @ 2
    X=SQRT X
    K=1
    L=-1

4)  L=-L
    X [K] =-B/2/A+L X X/2/A
    PRINT X [K]
    K=K+1
    JUMP IF L=1 @ 4

3)  EXIT

2)  TITLE AZ EGYENLETNEK NINCSEK VALÓS
    GYÖKE JUMP @ 3

```

A program futtatása során valahányszor a SUBR 1 utasítás kerül végrehajtásra, a program futása az 1)-referenciaszámtól folytatódik. A belépési pont referenciaszámával azonosított szubrutin végét egy EXIT utasítás jelzi, az amelyik a szubrutinba való belépés után leghamarabb kerül végrehajtásra. Nem szükséges az EXIT-et a szubrutin végére elhelyezni, mint a programban látjuk, az EXIT utasítás megelőzi a szöveg kinyomtatását, mely akkor kerül sorra, ha a diszkrimináns negatív.

A TITLE után elhelyezett feltétel nélküli ugró utasítás biztosítja a szubrutinból való kilépést ugyanazon az EXIT utasításon keresztül, mely lezárja a szubrutint nem negatív diszkrimináns esetén vagyis ha a másodfoku egyenletnek gyökei valósak.

Egy szubrutinból a programban különböző helyeken elhelyezett EXIT utasításokkal is vissza lehet térni a főprogramnak a SUBR-t követő utasítására. Ha például a 6. változat:

```

      .  

      .  

      .  

      .  

JUMP IF L=1 @ 4  

3) EXIT  

2) TITLE AZ EGYENLETNEK NINCSEK VALÓS GYÖKE,  

EXIT
```


utasításokkal fejeződik be, akármelyik EXIT utasítás kerül sorra, a program futása a szubrutinba való belépést előidéző SUBR 1-et követő utasítássor végrehajtásával folytatódik.

Ezzel szemben igaz, hogy minden autokód programban elég egyetlen egy EXIT utasítást elhelyezni, függetlenül attól, hogy hány különböző szubrutin szerepel a programban. Ha például egy autokód programban a

SUBR 1
SUBR 5
SUBR 8
SUBR 9 és

SUBR 11 szerepel, akkor egyetlen EXIT utasítással valamennyi felsorolt szubrutin befejezését az utóbbi módon lehet megoldani:

1) _____

SUBR 1 utasítás sorai:

JUMP @ 15

5) _____

SUBR 5 utasítás sorai

JUMP @ 15

8) _____

SUBR 8 utasítás sorai

JUMP @ 15

11) _____

SUBR 11 utasítás sorai

JUMP @ 15

9) _____

SUBR 9 utasítás sorai

15) EXIT

A SUBR 1, SUBR 5, SUBR 8,
SUBR 9, SUBR 11 utasítás sora.

⋮

Visszatérve a másodfoku egyenlet megoldásának 6. változatára, a két gyököt, az $X[1]$ és az $X[2]$ változóiban helyezzük el úgy, hogy $X[K]$ -val hivatkozunk mind az $X[1]$ mind az $X[2]$ változóra. Ha a program futása során a K aktuális értéke 1, akkor az $X[K]$ végrehajtása során az $X[1]$, ha K aktuális értéke 2, akkor $X[2]$ kap értéket.

Vigyázzunk az index értékek helyes kialakítására. Ha például a 6. változat programjában a $K=1$ utasítássort kifejeztettük volna és a programnak valamilyen korábbi, akár szubrutinon kívüli, vagy akár szubrutinon belüli értékadó utasítása alapján a K aktuális értéke 1526 lenne, akkor a SUBR 1 legközelebbi behívása után a kiszámított gyökök az $X[1526]$ illetve $X[1527]$ rekeszekbe kerülnének. Ha ezekkel az index értékekkel meghatározott rekeszek kívül esnek a deklarált változók által elfoglalt memória rekeszeken, az autokód fordító programfutás közben működő része hibát jelez. Nem kapunk viszont hibajelzést, ha a hibás index-szel hivatkozott rekesz a programdeklaráció alapján belül esik a deklarált változók határán, vagyis valamelyik másik változó számára van fenntartva. Ebben az esetben hibára csak a program végeredményéből lehet következtetni.

Hibajelzést kapunk, ha egy változó aktuális indexértéke negatív szám, függetlenül attól, hogy az a memória rekesz, melyre ezzel az index értékkel hivatkozunk kívül, vagy belül esik a deklarált változók számára fenntartott változók határán.

Tegyük fel ezekután, hogy igen sok másodfoku egyismeretlenes egyenletet akarunk egy programmal megoldani. Azt is tegyük fel, hogy tudjuk a megoldandó egyenletek számát. Legyen ez például 78. Ebben az esetben 78 számhármaszt kell az adatszalagra legépelni. Felhívjuk a lyukkártyás gyakorlattel és így lyukkártyás szemlélettel rendelkező programozók figyelmét arra, hogy lyukszalagra gépelés esetében a gépelés formátumának semmiféle szerepe nincs. Lyukszalagos adatbeolvasás esetén csak a lyukszalagon lelyukasztott számok sorrendje számít. Az természetesen szükséges, hogy a számokat adatokat valamilyen terminátor adathatárolójel válassza el egymástól. Ilyen jel az EMG-Autokódban a "szóköz" (space) jel, és az "ujrsor" (newline) jel. A fenti megszorításon kívül az adatok elrendezésére egyéb megkötés nincs. Ha például a megoldandó első egyenlet együtthatói: 2 -1 3 , a második egyenlet együtthatói: 19 3 5 ,

a harmadik egyenlet együtthatói: -7 5 1 ,
stb., akkor ezek az adatok az adatszálagon akár mind egy sorban

(2 -1 3 19 3 5 -7 5 1 stb...)

akár hármassával egy sorba

(2 -1 3
19 3 5
-7 5 1)

akár ötösével egy sorba

(2 -1 3 19 3
5 -7 5 1 stb.)

vagy akár számonként külön sorba

(2
1
3
19
3
5
-7
5
1
stb..)

kerülhetnek, a hatás azonos lesz. A programban kell arról gondoskodni, hogy az egyenlet megoldó program az első végrehajtás során három adatot, esetünkben ez 2 -1 3 , olvasom be, mégpedig a megfelelő rekeszekbe.

Ezeket a rekeszeket az eddigi program változatokban A B C-vel jelöltük. Az első három adat beolvasása után az adatszálagnak az a része, mely az első három adat karaktereit tartalmazza, már átjutott az olvasó fénycsikja alatt, és így a program második végrehajtása során a gép és így a program számára is olyanná vált, mintha egyáltalán nem is szerepelne a lyukszalagon. A program második végrehajtása során így az eredetileg negyedik, ötödik és hatodiknak gépelt számot, esetünkben a 19 3 5 -öt találja a számológép a soronkövetkező első, második és harmadik adatnak, és így ezeket olvassa be a fent említett A B C rekeszekbe.

A program ismételt végrehajtásai során ez az eljárás ugyanígy megy végbe. Láthatjuk tehát, hogy a leírt folyamat független az adatszálagra lyukasztott számok elrendezésétől, egyes egyedül a számok sorrendje számít.

Vigyáznunk kell viszont arra, hogy a program adatszálagról beolvasó READ utasításai és az adatszálag adatainak sorrendje és száma összhangban legyenek egymással.

Ha például az eddigi programváltozatok

```
READ A
READ B
READ C
```

programrészletét (ahol az A változóba olvastuk be a másodfoku tag, a B változóba az elsőfoku tag együtthatójának, és a C változóba az egyenlet konstansának számértékét) a

```
READ C
READ B
READ A
```

programrészlettel cseréljük fel és az A B C változók fenti értelmezését megtartjuk, akkor a

2	-1	3
19	3	5
-7	5	1

adatszalag helyett a

3	-1	2
5	3	19
1	5	-7

adatszalagot kell írunk.

Ha a két programrészlet hatását megvizsgáljuk, láthatjuk, hogy mindkét esetben az A változó számértéke 2 19 és 7 , a
B változó számértéke -1 3 és 5 , a
C változó számértéke 3 5 és 1 a

program egymásutáni három végrehajtása során.

Hasonló a helyzet akkor is, ha az adatszalagon az adott feladat számára felesleges adatok vannak lelyukasztva. Ilyen például az az eset, amikor az egyenlet együtthatóit jelentő három szám elé a soronkövetkezés száma van lyukasztva. Ekkor az adatszalag:

1	2	-1	3
2	19	3	5
3	7	5	1
4	-	-	-
5	-	-	-
6	-	-	-

Mint az alábbi esetben láttuk, a

```
READ A
READ B
READ C
```

programrészlet a program egy végrehajtása során három adat beolvasására képes.

Ha a programot és az adatszalagot is változatlanul hagynánk, az első számhármassal 2 19 7 helyett 1 2 -1, a további program végrehajtá-

sok esetében 19 3 5 helyett 3 2 19,
7 5 1 helyett 3 5 3

stb.. lenne, és így az esetek legnagyobb részében helytelen eredményt kapnánk. Hogy a helyes eredményt megkapjuk, gondoskodnunk kell a felesleges adatok törléséről. A legegyszerűbb esetnek egy új, mostmár a programmal összhangban álló adatszalog elkészítése látszik. Sok adatot tartalmazó adatszalog esetén azonban ez igen nagy munkát jelent. Célszerűbbnek látszik, ha a programot változtatjuk meg egy olyan utasítás beiktatásával mely a számunkra felesleges adatokat "elnyeli". Ez elérhető ha a

READ A
READ B
READ C

programrészletet

READ A
READ A
READ B
READ C - re változtatjuk.

Nézzük meg a változtatás hatását:

A program végrehajtása során az A változó előbb felveszi az 1, majd közvetlenül utána a 2 számértéket B értéke -1, C értéke 3.

Az egyenlet gyökeit tehát a

2 -1 3 számhármassal

számoljuk.

A következő végrehajtás során az A változó előbb felveszi a 2 majd közvetlenül utána a 19 számértéket

B értéke 3, C értéke 5.

Az egyenlet gyökeit tehát a

19 3 5 számhármassal számoljuk.

Láthatjuk tehát, hogy az adatok kívánt sorrendje ezzel helyreállt.

Ha a felesleges számok negyedik adatként jelentkeznek, úgy az előbbi kettős beolvasást az A változóba, nem tudjuk megismételni, hiszen az első három szám a számunkra értékes adat, ekkor a

READ A
READ B
READ C

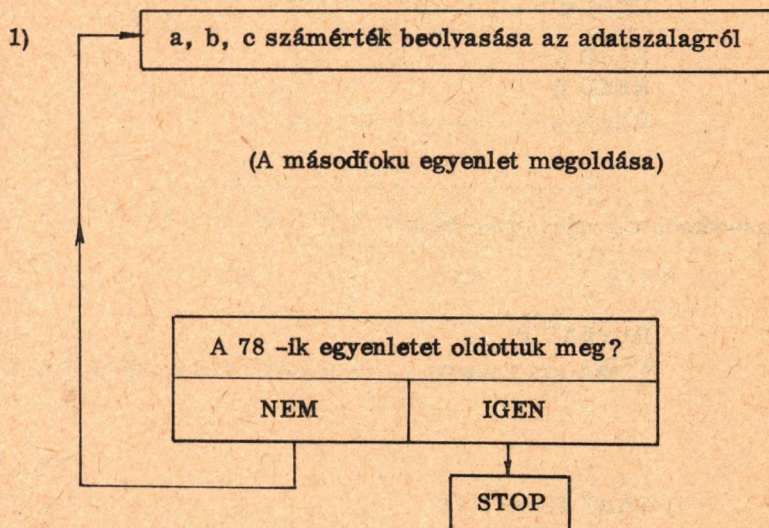
programrészletet a

READ A
READ B
READ C
READ X

programrészletre kell kicserélnünk, ahol az X változónak csak az a szerep

jutott, hogy a felesleges adat beolvasásával az adatszalogot abba a pozícióba juttassa, hogy a program legközelebbi végrehajtásakor az A B és C változóba helyes számérték kerüljön.

E kitérés után térjünk vissza a felvetett problémára, arra, hogy igen sok másodfoku egyenletet akarunk egy programmal megoldani. Ha tudjuk a megoldandó egyenletek számát, akkor ezt már programozás közben felhasználhatjuk, és beépíthetjük a programba. Legyen például a megoldandó egyenletek száma 78. Ebben az esetben a program blokkdiagramja az alábbi



A blokkdiagram alapján a program elkészítésére több lehetőségünk van.

A megoldás egyik módja az, hogy felvesszünk a deklarált változók közé egy P változót és a program ide vonatkozó részét így készítjük el:

```

      :
      :
      :
P = 78
1) READ A
   READ B
   READ C

(A másodfoku egyenlet megoldása)
      :
      :
      :
P = P-1
JUMP IF P > 0 @ 1
STOP
  
```

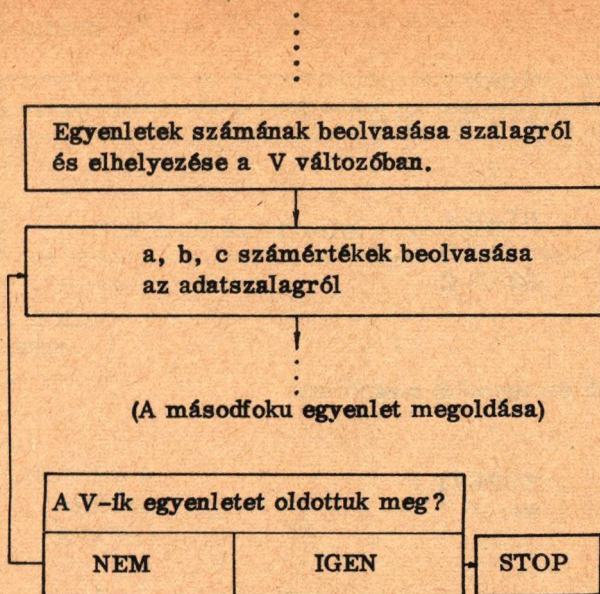

Látjuk, hogy a P változó a program futása során a 77, 76, 75, ... 1, 0, értékeket veszi fel, úgy, hogy az értékadás mindig egy egyenlet megoldása után következik be. Így 78 egyenletet old meg a program addig, míg a feltételes JUMP utasítás feltétele először nem teljesül.

A megoldás másik módja ciklusutasítás alkalmazása. Ezt a módot CYCLE és VARY utasításra is bemutatjuk:

```
      .  
      .  
      .  
1) CYCLE P = 1:1:78  
      READ A  
      READ B  
      READ C  
      .  
      .  
      .  
(A másodfoku egyenlet megoldása)  
      .  
      .  
      .  
      REPEAT P  
      STOP
```

```
      .  
      .  
      .  
1) VARY P = 0:1:78  
      READ A  
      READ B  
      READ C  
      .  
      .  
      .  
(A másodfoku egyenlet megoldása)  
      .  
      .  
      .  
      REPEAT P  
      STOP
```

Térjünk most újra vissza oda, hogy igen sok másodfoku egyismeretlenes egyenletet kell egy programmal megoldanunk, de ezek száma előre ismeretlen. Elképzelhető, hogy milyen nehézséget jelent, ha a program futtatása előtt minden alkalommal meg kell számolnunk az adatszalai számhármait, és ennek alapján módosítani a program 78-as konstansát. Célszerű, ha már a program készítése során gondolunk ilyen változtatásra és módot adunk arra, hogy a program automatikusan állítsa be a ciklus utolsó végrehajtási értékét.



A blokkdiagramhoz tartozó három programvariáns:

JUMP típus:

```

READ V
1) READ A
   READ B
   READ C
      (A másodfoku egyenlet megoldása)

V=V-1
JUMP IF V > 0 @ 1
STOP
  
```


CYCLE típus:

```
    READ V
    CYCLE P=1:1:V
```

⋮

```
    READ A
    READ B
    READ C
```

⋮

(A másodfoku egyenlet megoldása)

⋮

```
    REPEAT P
    STOP
```

VARY típus:

```
    READ V
    VARY P=0:1:V
```

```
    READ A
    READ B
    READ C
```

⋮

(A másodfoku egyenlet megoldása)

⋮

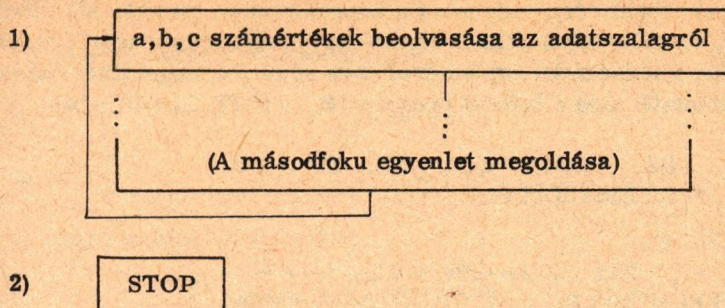
```
    REPEAT P
    STOP
```

Megemlítjük, hogy amennyiben az egyenlet megoldásban a P változót csak a ciklusok számlálására használjuk, a

VARY P=0:1:V utasítássorban a 0 és 1 szám, tehát a ciklus kezdő érték és a lépésköz helyén, tetszésszerűen konstans állhat anélkül, hogy a program bármelyik rész-, vagy végeredménye megváltozna.

Még általánosabbá tehetjük a programot, ha kihasználjuk az autokód un. "trigger"-rel vezérlő lehetőségét. Trigger vezérlésnél a JUMP -típusu megoldást alkalmazzuk, de egyáltalán nem törődünk azzal, hogy hány egyenletet kell megoldanunk, és így a feltételes JUMP utasítás helyett feltétel nélküli JUMP-ot írunk a programba.

A blokk diagram:



A program:

```

1) READ A
   READ B
   READ C
   .
   .
   .
(A másodfoku egyenlet megoldása)
   .
   .
   .
JUMP @ 1

2) STOP
  
```

Már a blokkdiagramnál rögtön feltűnik, hogy a STOP utasítás semmiféle kapcsolatban nincs a blokkdiagram többi részével. Mi biztosítja mégis, hogy az utolsó adat beolvasása után, a program ne kezdje el egy újabb számhármassal egy újabb másodfoku egyenlet gyökeit kiszámolni? Erre biztosíték a trigger. A trigger ugyanugy az adatszalagra van lelyukasztva mint az adatok, de program vezérlés hatása. A trigger egy egész számból és egy közvetlenül a szám mögé lyukasztott (-ből áll. (kerek nyitózárójel) Hatására a program futása ott folytatódik, ahol a triggernek megfelelő referenciaszám a programban el van helyezve. Természetesen a trigger csak egy READ utasítás alapján fejtheti ki hatását. Példánkban a STOP utasítás elé a 2 referenciaszámot helyeztük el. Ennek megfelelően az adatszalag utolsó adata után

2 (-t kell lyukasztani.

Az utolsó egyenlet megoldása után a JUMP @ 1 feltétlen ugró utasítás hatására a program futása az 1 referenciaszámtól folytatódik. Ekkor a READ A utasítás hatására a program beolvassa az adatszalag soronkövetkező karaktereit.

Mielőtt azonban adatelválasztójel jönne, és az A változóba bekerülne a beolvasott szám, a (hatására a program futása a 2 referenciaszámtól folyta-

tódik. Nem mondhatjuk tehát, hogy a STOP utasítás nincs semmiféle kapcsolatban a program többi részével, mert a kapcsolatot az utasítás elé kitett referenciaszám, és valamelyik READ utasítással beolvasott trigger biztosítja. Mielőtt lezárjuk a másodfoku egyismeretlenes egyenlet megoldását összehasonlításképpen tekintsük meg a feladat programját, - GIER ALGOL -ban:

```
PROCEDURE EQ2 (A,B,C Z1r, Z1i, Z2r, Z2i,
                INDETERMINATE);
```

```
VALUE A, B, C;
```

```
REAL A, B, C, Z1r, Z1i, Z2r, Z2i;
```

```
LABEL INDETERMINATE;
```

```
BEGIN
```

```
REAL discriminant;
```

```
    if A  $\neq$  0 then goto normal;
```

```
    if B = 0 then goto INDETERMINATE;
```

```
    Z1r:=Z2r:=-C/B;
```

```
    goto set zero;
```

```
normal:
```

```
discriminant:= B  $\uparrow$  2-4xAxC;
```

```
if discriminant > 0 then goto realsolution;
```

```
complex:
```

```
Z1r:=Z2r:=-B/2/A;
```

```
Z1i:=sqrt /-discriminant/ /2/A;
```

```
Z2i=-Z1i;
```

```
goto finis;
```

```
real solution:
```

```
if B < 0 then Z1r:=-1 else Z1r:=1;
```

```
Z1r:= (-B-Z1r x sqrt (discriminant)) /2/A;
```

```
Z2r:= C/A/Z1r;
```

```
set zero: Z1i:=0;
```

```
finis : end of EQ2;
```

2. Feladat

Adott 50 számpárt jelöljük $m_1, n_1, m_2, n_2 \dots m_{50}, n_{50}$ -nel. Számítsuk ki mindegyik számpárra a következő kifejezés értékét:

$$X = \frac{2 m n^2 - 3 m^2 n + 1}{3 m - n}$$

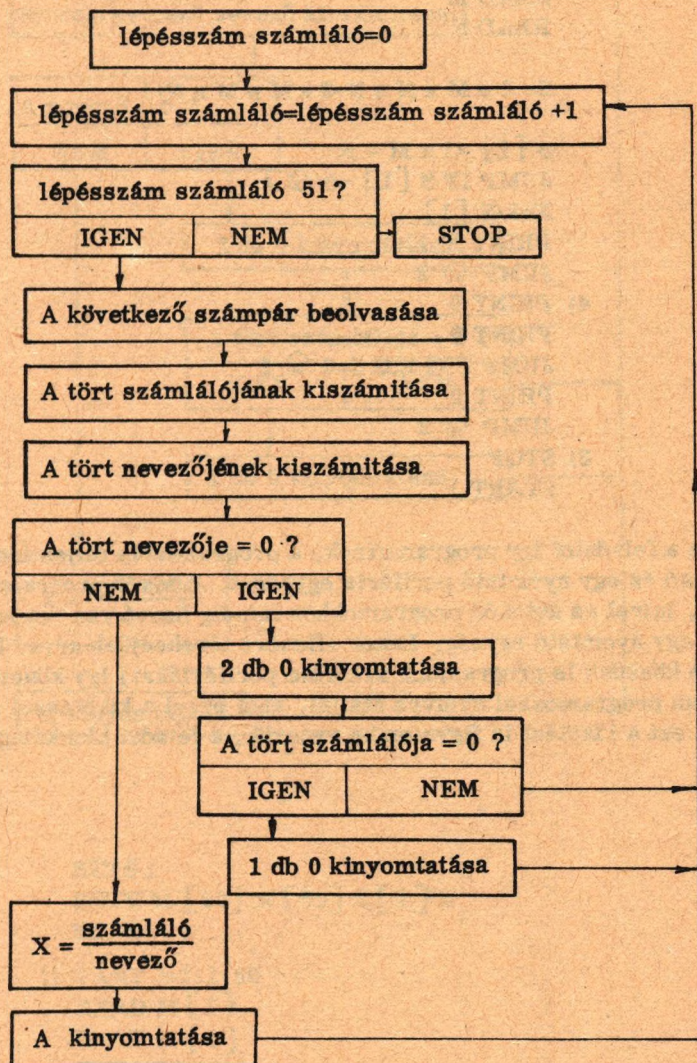
Nyomtassuk ki minden m_i, n_i ($1 \leq i \leq 50$) számpárra x értékét, kivéve az alábbi eseteket:

Ha $X = \frac{0}{a}$, akkor nyomtassuk: $X=0$,

ha $X = \frac{b}{0}$, " " : $X=0\ 0$

ha $X = \frac{0}{0}$, " " : $X=0\ 0\ 0$

A program blokkdiagramja:



A program:

```
SETS L
STV MNS [1] X
SETR 4
1) L = 0
```

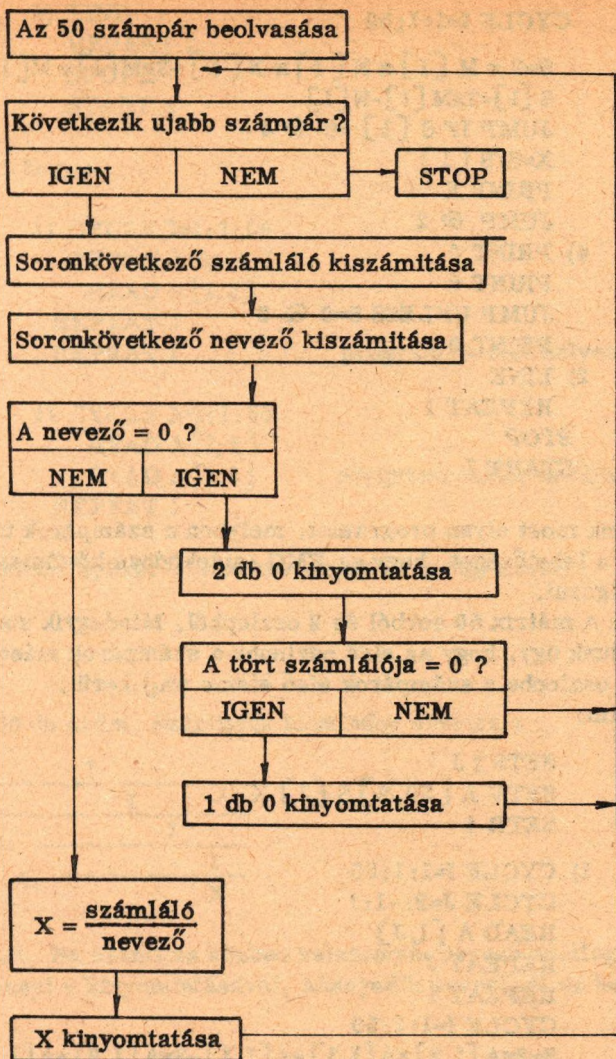
```
2) L = L+1
   LINE
   JUMP IF L > 50 @ 3
   READ M
   READ N
```

$S = 2 * M * N * N - 3 * M * M * N + 1$

```
S [1] = 3 * M - N
JUMP IF S [1] = 0 @ 4
X = S / S [1]
PRINT X
JUMP @ 2
```

```
4) PRINT 0
   PRINT 0
   JUMP UNLESS S = 0 @ 2
   PRINT 0
   JUMP @ 2
3) STOP
   START 1
```

Ha a feladatot így programozzuk, a programfutás teljes idejére lekötünk egy olvasó és egy nyomtató periféria egységet. Autokód program esetében ez nem baj, mivel az autokód programokhoz mindig hozzá van rendelve egy olvasó és egy nyomtató egység. Ennek ellenére törekedjünk arra, hogy feleslegesen ne kössünk le programunk számára perifériákat, így könnyebben tudunk más olyan programozási nyelvre átállni, ahol ezzel a kikötéssel számolnunk kell. Ha ezt a kikötést is figyelembe vesszük, a feladat blokkdiagramja a következő:



A program:

```

SETS I
SETV M [50] N [50] S [1] X
SETR 4

1) VARY I=1:1:50
  READ M [I]
  READ N [I]
  REPEAT I

```



```

CYCLE I=1:1:50
S=2 * M [ I ] * N [ I ] * N [ I ] - 3 * M [ I ] * M [ I ] * N [ I ] + 1
S [ 1 ] = 3 * M [ I ] - N [ I ]
JUMP IF S [ 1 ] = 0 @ 4
X=S/S [ 1 ]
PRINT X
JUMP @ 2
4) PRINT 0
PRINT 0
JUMP UNLESS S=0 @ 2
PRINT 0
2) LINE
REPEAT I
STOP
START 1

```

Készítsünk most olyan programot, melyben a számpárok tárolására kihasználjuk azt a lehetőséget, hogy az EMG -autokódban kétdimenziós mátrixokkal is lehet dolgozni.

Álljon az A mátrix 50 sorból és 2 oszlopból. Mindegyik sorban egy-egy számpárt tárolunk úgy, hogy az első oszlopba a számpárok második eleme (n_i) a második oszlopba a számpárok első eleme (m_i) kerül.

A program:

```

SETS I J
SETV A [ 50, 2 ] S [ 1 ] X
SETR 4
1) CYCLE I=1:1:50
CYCLE J=2: -1:1
READ A [ I, J ]
REPEAT J
REPEAT I
CYCLE I=1:1:50
S=2 * A [ I, 2 ] * A [ I, 1 ] * A [ I, 1 ] - 3 * A [ I, 2 ] * A [ I, 2 ] * A [ I, 1 ] + 1
S [ 1 ] = 3 * A [ I, 2 ] - A [ I, 1 ]
JUMP IF S [ 1 ] = 0 @ 4
X=S/S [ 1 ]
PRINT X
JUMP @ 2
4) PRINT 0
PRINT 0
JUMP UNLESS S=0 @ 2
PRINT 0

```



```

2) LINE
   REPEAT I
   STOP
   START 1

```

Megjegyezzük, hogy a

```

1) CYCLE I=1:1:50
   CYCLE J=2=-1:1
   READ A [I,J]
   REPEAT J
   REPEAT I

```

programrészlet helyett a

```

1) CYCLE I=1:1:50
   READ A [I, 2]
   READ A [I, 1]
   REPEAT I

```

programrészlet is állhatna, az

előbbivel egymásba épített ciklusokra hívjuk fel a figyelmet.

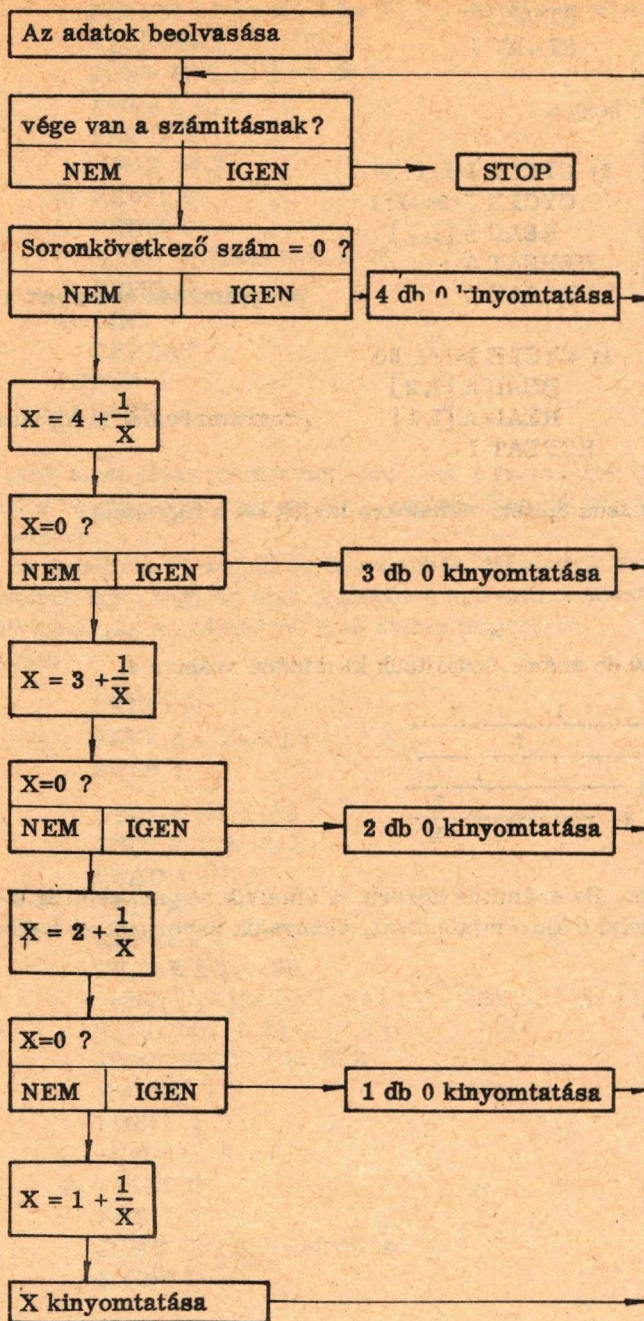
3. Feladat:

Adott 100 db szám. Számítsuk ki minden számra a

$$\begin{array}{r}
 J=1+ \frac{1}{2+ \frac{1}{3+ \frac{1}{4+ \frac{1}{X}}}}
 \end{array}$$

lántört értékét. Ha számítás közben valamelyik nevező esetleg 0-vá válik, jelezzük ezt annyi 0 kinyomtatásával, ahányadik nevezőnél ez bekövetkezett.

Blokkdiagram:



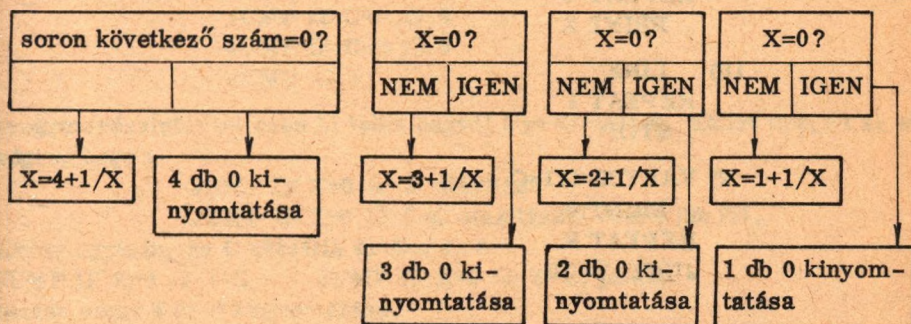
A program:

```

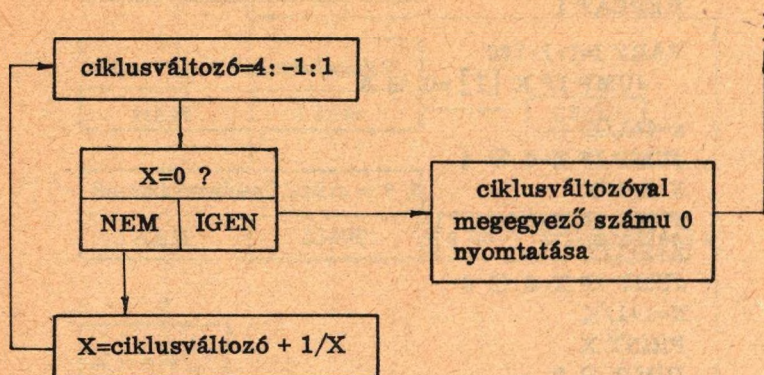
VARY I=1:1:100
  READ X [I]
  REPEAT I
    VARY I=1:1:100
      JUMP IF X [I] =0 @ 3
    x=4+1/X
    JUMP IF X=0 @ 4
    X=3+1/X
    JUMP IF X=0 @ 5
    X=2+1/X
    JUMP IF X=0 @ 6
    X=1+1/X
    PRINT X
    JUMP @ 2
  3) PRINT 0
  4) PRINT 0
  5) PRINT 0
  6) PRINT 0
  2) LINE
  REPEAT I
  STOP

```

Ennél a feladatnál valóban érdemes volt blokkdiagramot készíteni. Ha ugyanis az elkészült blokkdiagramot megnézzük, rögtön észrevehetjük a közel azonos részeket:



Ezeket a közel azonos blokkdiagram részeket igen könnyen lehet egybe foglalni:



A program:

```

1) VARY I=1:1:100
   READ X [I]
   REPEAT I
     VARY I=1:1:100
     X=X [I]
     CYCLE C=4: -1:1
     JUMP IF X=0 @ 2
     X=C+1/X
     REPEAT C
     PRINT X
10) LINE
     REPEAT I
     STOP
2) VARY K=1:1:C
   PRINT 0
   REPEAT K
   JUMP @ 10
   .
   .
   .
  
```


Ha a 0-k nyomtatását nem akarjuk ciklusutasítással végezni, akkor a blokkdiagram érvényben maradása mellett a program az alábbi:

```

1) VARY I=1:1:100
   READ X [I]
   REPEAT I
   VARY I=1:1:100
   X=X [I]

   CYCLE C=4:-1:1
   JUMP IF X=0 @ 2
   X=C+1/X
   REPEAT C
   PRINT X
10) LINE
   REPEAT I
   STOP

2) JUMP IF C=4 @ 3
   JUMP IF C=3 @ 4
   JUMP IF C=2 @ 5
   JUMP @ 6

3) PRINT 0
4) PRINT 0
5) PRINT 0
6) PRINT 0
   JUMP @ 10

```

Ügyes fogással a

```

2) JUMP IF C=4 @ 3
   JUMP IF C=3 @ 4
   JUMP IF C=2 @ 5
   JUMP @ 6

```

programrészletet teljesen ki lehet hagyni a programból. Ehhez csupán az szükséges, hogy a

```

JUMP IF X=0 @ 2 utasítást
JUMP IF X=0 @ 7-C utasítással cseréljük fel.

```

Ekkor ugyanis, ha C aktuális értéke 4, a
 JUMP IF X=0 @ 7-C JUMP IF X=0 @ 3 -at jelent,
 hatása pedig 4 db 0 kinyomtatása,
 ha C aktuális értéke 3
 a JUMP IF X=0 @ 7-C JUMP IF X=0 @ 4 -et jelent,
 hatása pedig 3 db 0 kinyomtatása,
 ha C aktuális értéke 2, a
 JUMP IF X=0 @ 7-C JUMP IF X=0 @ 5-öt jelent,
 hatása pedig 2 db 0 kinyomtatása, végül
 ha C aktuális értéke 1, a

JUMP IF X=0 @ 7-C JUMP IF X=0 @ 6 -ot jelent,
hatása 1 db 0 kinyomtatása.

A feltételes ugró utasításban tehát kihasználtuk, hogy négy egymást követő referenciaszámra kell a feltétel teljesülése esetén ugrani! Tegyük fel most, hogy valaki a program elkészítése után veszi észre ezt az egyszerűsítési lehetőséget, és nincs a deklarált referenciaszámok között négy egymásutáni, még fel nem használt referenciaszám. A referenciaszám deklarációt viszont nem tudja tovább bővíteni, mivel a memóriának az a része ahová a referenciaszám táblázat kerül, ezt a bővítést már nem bírja el. Ezzel szemben szabadon vannak a 4, 39, 45- és a 47-es referenciaszámok. A referenciaszámok és a hivatkozási helyek átszámozása igen bonyolult és igen sok hiba-lehetőséget rejt magában:

Tegyük fel, hogy a memóriának abban a részében, ahová a lefordított autokód program futás közben kerül, még van szabad hely. Munkarekeszként rendelkezésre áll az A 10, A 20, A 30, A 40-es rekesz.

Ekkor a program az alábbi:

```
1) A 10 = 47
   A 20 = 45
   A 30 = 39
   A 40 = 4

VARY I=1:1:100
  READ X [I]
REPEAT I
VARY I=1:1:100
  X=X [I]
CYCLE C=4:-1:1
JUMP IF X=0 @ A [10xC]
  X=C+1/X
REPEAT C
  PRINT X
10) LINE
  REPEAT I
  STOP

4) PRINT 0
39) PRINT 0
45) PRINT 0
47) PRINT 0
    JUMP @ 10
```

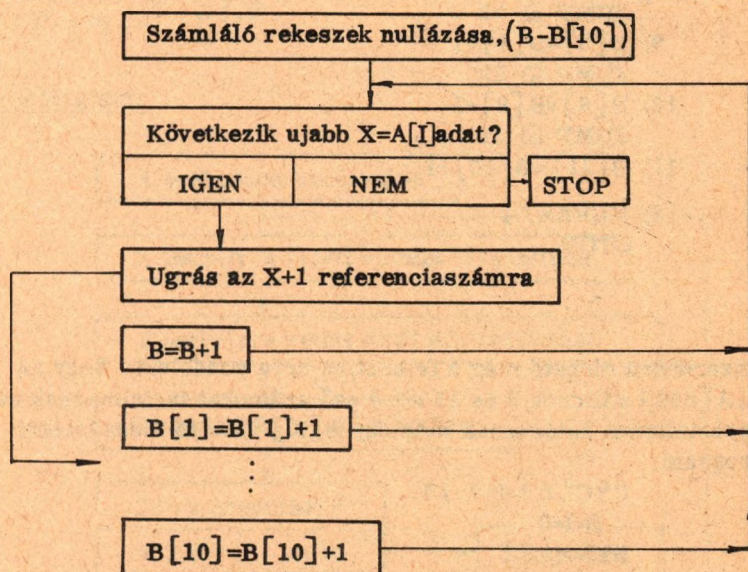
Ekkor ugyanis ha $C=4$, az $A [10xC] \equiv A [40]$,
a program futása a 4/-től folytatódik,
ha $C=3$ az $A [10xC] \equiv A [30]$,

a program futása a 39/-től folytatódik,
 ha $C=2$ az $A[10 \times C] \equiv A[20]$,
 a program futása a 45/-től folytatódik,
 ha $C=1$ az $A[10 \times C] \equiv A[10]$,
 a program futása a 47/-től folytatódik, ugyanis minden esetben a kívánt
 eredményhez jutottunk.

4. Feladat

Tartalmazzon az $A[I]$ $I=1, 2, \dots, 500$ vektor mindegyik komponense egy
 0- és 10 közé eső pozitív számot. Számoljuk meg, hogy melyik számból hány
 darab van.

A blokkdiagram



A program:

```

      ⋮
VARY I=0:1:11
  B[I]=0
REPEAT I
VARY I=1:1:500
  X=A[I]
  JUMP @ X+1
    
```



```

1) B=B+1
   JUMP @ 12
2) B[1]=B[1]+1
   JUMP @ 12
3) B[2]=B[2]+1
   JUMP @ 12
4) B[3]=B[3]+1
   JUMP @ 12
5) B[4]=B[4]+1
   JUMP @ 12
6) B[5]=B[5]+1
   JUMP @ 12
7) B[6]=B[6]+1
   JUMP @ 12
8) B[7]=B[7]+1
   JUMP @ 12
9) B[8]=B[8]+1
   JUMP @ 12
10) B[9]=B[9]+1
   JUMP @ 12
11) B[10]=B[10]+1
12) REPEAT I
    STOP
    :
    :
    :

```

Egyszerűbben oldható meg a feladat, ha arra gondolunk, hogy az $A[1]$, $A[2]$, ..., $A[500]$ változók 0 és 10 közé eső számokat tartalmaznak és ezek számok egyértelműen határozzák meg egy $B[K]$ változó indexértékét.

A program:

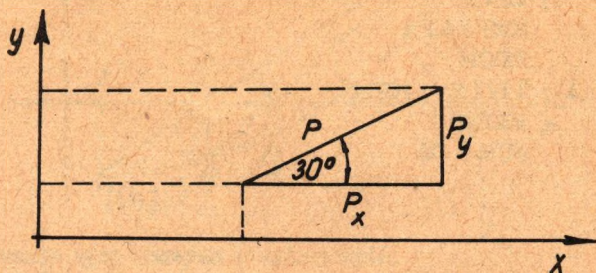
```

CYCLE I=0:1:10
  B I=0
  REPEAT I
    VARY I=1:1:500
      K=A[I]
      B[K]=B[K]+1
  REPEAT I
  STOP
  :
  :
  :

```


5. Feladat

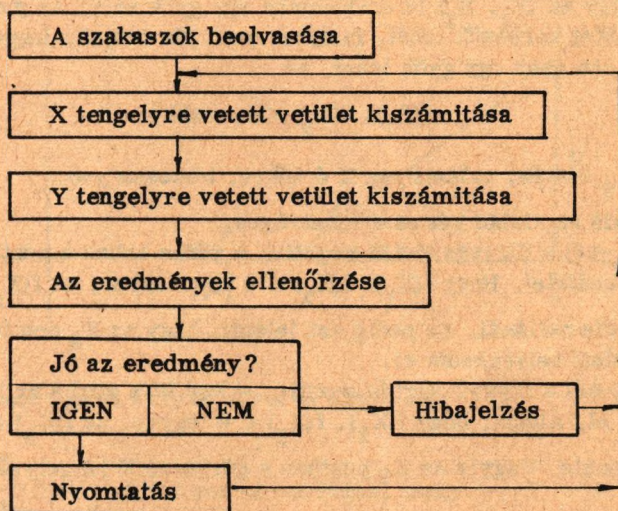
Adott 50 db pozitív szám. Mindegyikük egy-egy olyan szakasz hosszát jelenti, mely 30° -os szöget zár be az X tengellyel. Számítsuk ki e szakaszok vetületeinek hosszát az X és az Y tengelyre.



$$P_x = P \cdot \cos 30^\circ = P \frac{\sqrt{3}}{2}$$

$$P_y = P \cdot \sin 30^\circ = \frac{P}{2}$$

Blokkdiagram:



A program:

```
VARY I=1:1:50
  READ P[I]
  REPEAT I
    VARY I=1:1:50
    X=P[I]*SQRT 3/2
```



```

Y=P[I] /2
P=P[I] * P[I]
JUMP UNLESS P=X*X+Y*Y @ 11
PRINT X
PRINT X
7) LINE
REPEAT I
STOP
11) TITLE HIBA'
PRINT I
JUMP @ 7

:
:
:

```

5. Feladat

Számítsuk ki az ún. "hur módszer"-rel egy adott $y=f(x)$ függvény (X_1, X_2) intervallumba eső gyökét. Legyen az $y=f(x)$ függvény folytonos, és tegyük fel, hogy az (X_1, X_2) intervallumba egy gyök esik. Az $y=f(x)$ függvény olyan X értékét keressük tehát, melyre: $y=0$. Ha tudjuk, hogy az (X_1, X_2) intervallumban csak egy gyök lehet, és

$$f(X_1) \cdot f(X_2) < 0, \text{ akkor}$$

$X_1 < X < X_2$ esetén, valamilyen X értékre, biztosan $y=0$.

Ugyanis az alábbi két eset lehetséges:

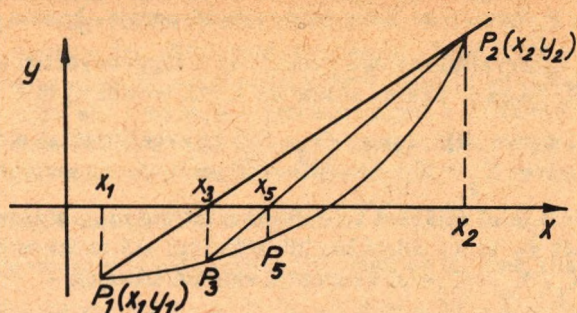
Ha X_1 -nél a függvényérték negatív, a görbe abban a pontban az X tengely alatt helyezkedik el. Hogy $f(X_1) \cdot f(X_2) < 0$ igaz legyen, az $f(X_2)$ -nek nagyobb-nak kell lennie nullánál. Ez pedig azt jelenti, hogy az X_2 pontban a görbe az X tengely felett helyezkedik el.

Ha X_2 -nél a függvényérték pozitív, akkor itt a görbe az X tengely felett helyezkedik el. Ahhoz, hogy $f(X_1) \cdot f(X_2) < 0$ legyen, az $f(X_2)$ -nek kell nullánál kisebbnek lennie. Vagyis az X_2 pontban a görbe az X tengely alatt kell hogy haladjon.

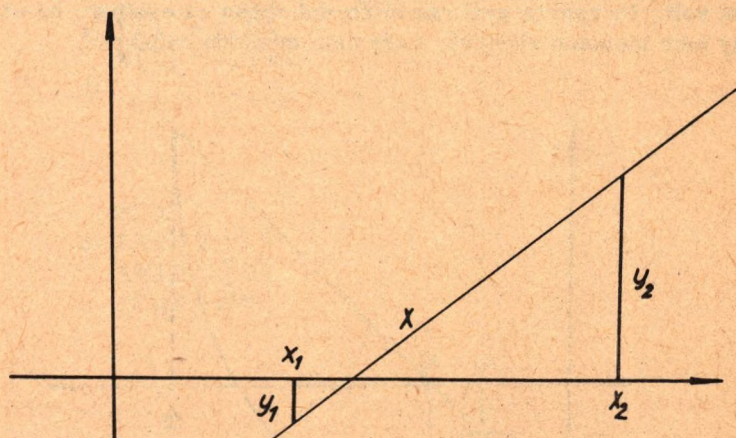
Mindkét esetben, éppen az $y=f(x)$ függvény folytonossága miatt, a görbe keresztezi az X tengelyt. Ebben a pontban az $f(x)$ függvényérték 0, vagyis megtaláltuk a X_1, X_2 intervallumba eső gyököt.

Tegyük fel, hogy az első esettel kell dolgoznunk. Ekkor az X_1 és X_2 intervallum határok vetülete meghatároz két pontot a görbén. Jelöljük e két pont koordinátáit $P_1(X_1, Y_1)$ és $P_2(X_2, Y_2)$ -vel. A $P_1 P_2$ pontokat összekötő egyenes szakasz a görbe hurja.

A hurra nézve is igazak a görbére tett megállapítások, a hur az X_1 és X_2 intervallum határok között valahol metszi az X tengelyt.



A hurnak ez a pontja igen könnyen kiszámolható.



Ugyanis a két adott ponton átmenő egyenes egyenlete:

$$y - y_1 - \frac{y_2 - y_1}{x_2 - x_1} (x - x_1) \quad \text{Azt a pontot keressük, melyre } y=0.$$

$$-y_1(x_2 - x_1) = (y_2 - y_1)(x - x_1)$$

$$-y_1 x_2 + y_1 x_1 = y_2 x - y_2 x_1 - y_1 x + y_1 x_1$$

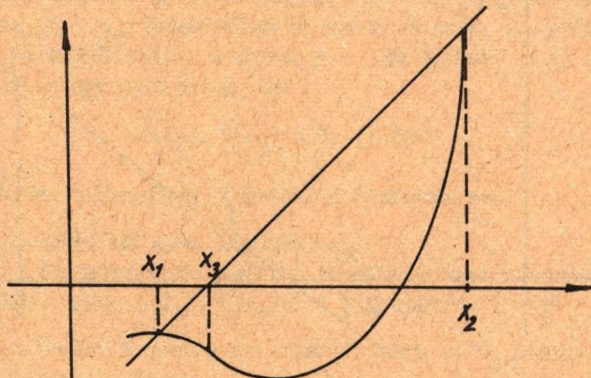
$$x = \frac{x_1 y_2 - x_2 y_1}{y_1 - y_2}$$

A kapott X értéket behelyettesítve a görbe $y=f(x)$ egyenletébe, kiszámítjuk az X -hez tartozó függvényértéket. Ha az így kapott függvényérték 0, akkor X nemcsak a P_1 P_2 hur és az X tengely metszéspontja, hanem az $y=f(x)$ görbének is X_1 , X_2 intervallumba eső gyöke. Ha a függvényérték az X pontban nem 0, hanem pl. negatív szám, akkor az $X_3=X$ pontot új intervallum határnak fogjuk fel, és az egész eljárást az (X_3, X_4) intervallummal előlről kezdjük.

($X_4=X_2$). Mivel $X_3 > X_1$, az intervallum hossza, melybe a gyök esik, rövidebb lett. Ha a leírt eljárást tetszés szerint sokszor elvégezzük, az intervallum hosszát az előre adott kis küszöbszám alá csökkenthetjük, mivel - ezt az $X_3 > X_1$, $X_4=X_2$ -vel azonos módon beláthatjuk -

$$X_1 \leq X_3 \leq X_5 \dots \leq X_{2n-1} \leq X \leq X_{2n} \dots \leq X_6 \leq X_4 \leq X_2$$

Nem kell megijednünk, ha valamelyik új függvényérték kiszámításakor a kapott eredmény távolabbra kerülést mutat az X tengelytől, mint az, az előző lépésben volt. Ez nagyon gyakran előfordul abban az esetben, ha az $y=f(x)$ függvény nem monoton növekvő, vagy nem monoton csökkenő.



A módszer tehát az, hogy mindig elhagyjuk azt az intervallum határpontot, amely függvényértékében a kiszámított $f(x)$ azonos előjelű, a megmaradt régi határpont és a kiszámolt új, már rövidebb intervallumot ad. Ezt az iterációs eljárást addig folytatjuk, míg az (X_{2n-1}, X_{2n}) intervallum hossza egy előre megadott kis küszöbszám alá csökken, jelöljük ezt E_2 -vel, és a kiszámított $f(x)$ függvényérték adott mértékben megközelíti a 0-t. Jelöljük ezt a számértéket E_1 -gyel.

Az iterációs programot rendszerint úgy készítik, hogy a program alkalmazója állíthassa be azt az E_1 és E_2 értéket amellyel számolni akar, és amely a feladat egyéb hibakorlátjainak megfelel. Nem célszerű ugyanis felesleges pontosságra törekedni, mivel az ilyen törekvés a programfutás idejét többszörösére növelheti.

Ha pl. egy program egyéb számítása 10^{-3} nagyságrendig pontosak, felesleges az intervallum hosszát pl. 10^{-8} nagyságrendig csökkenteni, hiszen az itt kapott "pontos" végeredmény már semmivel sem javítja a teljes feladat végeredményének pontosságát. Előfordulhat viszont, hogy nagy pontossáig kell számolni. Ebben az esetben ezt a pontosságot éppen a küszöbszámok megválasztásával érhetjük el. A program leírások egyik követelménye, hogy tartalmazza a programba beépített küszöbszámok nagyságát, és azt a módot, ahogyan ezeket más küszöbszámokra lehet felcserélni.

Pl. " a programot az 1)-ről indítva $E_1 = 0.001$ és $E_2 = 0.0003$.

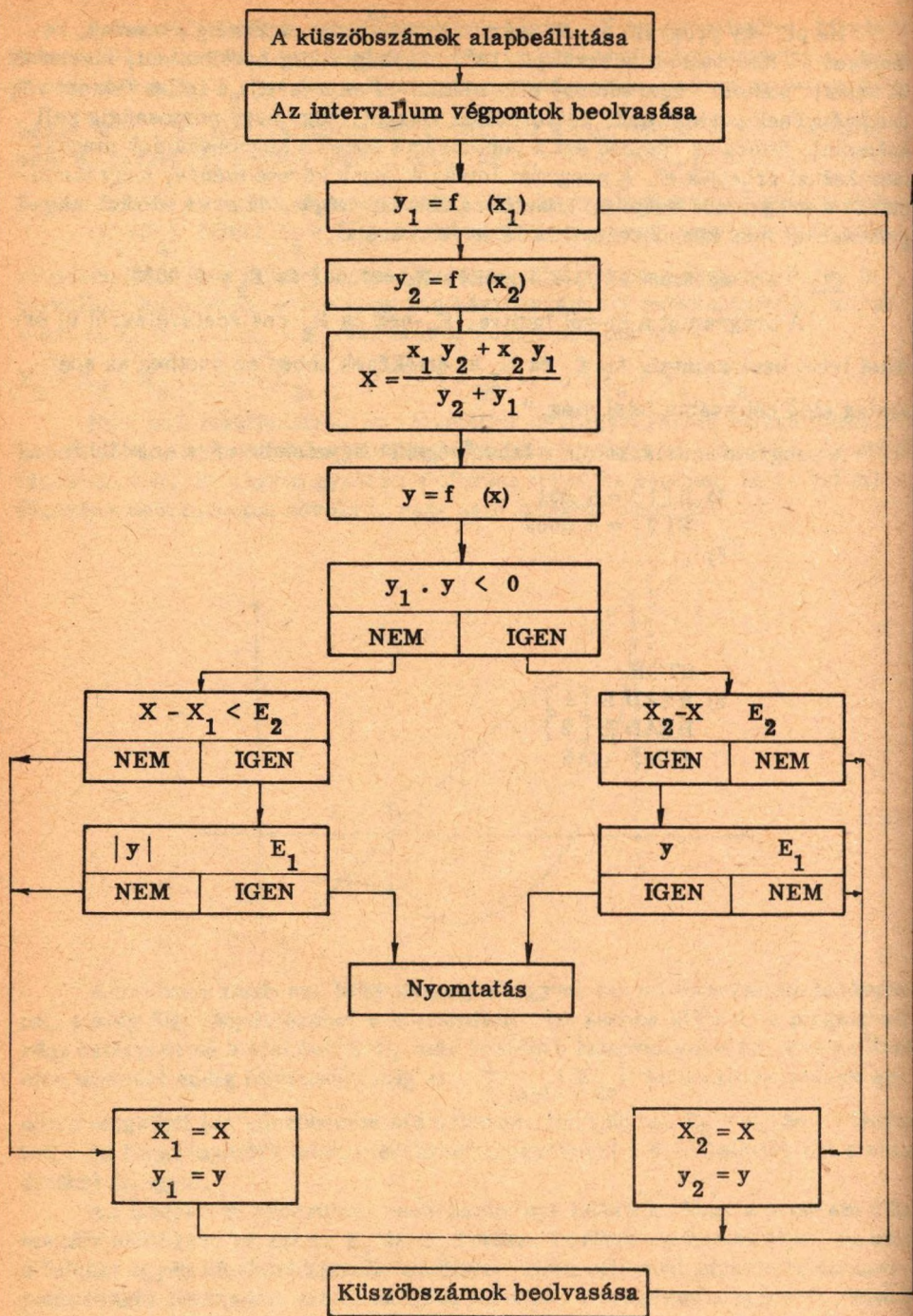
A programot a 8)-ről indítva, E_1 -nek és E_2 -nek adatszálagra új értéket lehet beolvastatni. Az E_1 és E_2 új értékének ebben az esetben az adatszálag első két száma felel meg. "

A program konstrukciója e lehetőségeket figyelembe véve az alábbi:

```

1) E [1] = 0.001
   E [2] = 0.0003
6) .....
   :
   :
   :
   STOP
8) READ E [1]
   READ E [2]
   JUMP Q6

```

A hurmódszer programja [$y = f(x) = x^2 - 3$ esetben]

SETS D
SETV X[2] y[2] E[2] A B
SETR 15

- 1) E[1] = 0.001
E 2 = 0.0003
10) READ X[1]
READ Y[2]

$y[1] = X[1] \cdot X[1] - 3$
 $y[2] = X[2] \cdot X[2] - 3$

- 11) $X = X[1] * y[2] - X[2] * y[1]$
 $A = y[2] - y[1]$

JUMP IF A = 0 @ 7

X = y/A

y = X * X - 3

A = y * y[1]

JUMP IF A < 0 @ 2

B = X - X[1]

D = 1

JUMP UNLESS B < E[2] @ 3

- 6) JUMP IF y > 0 @ 4

y = -y

- 4) JUMP IF y < E[1] @ 5

- 3) X[D] = X

y[D] = y

JUMP @ 11

D = 2

JUMP UNLESS B < E[2] @ 3

JUMP @ 6

- 5) PRINT X

LINE

- 7) STOP

- 8) READ E[1]

READ E[2]

JUMP @ 10

START 1

6. Feladat

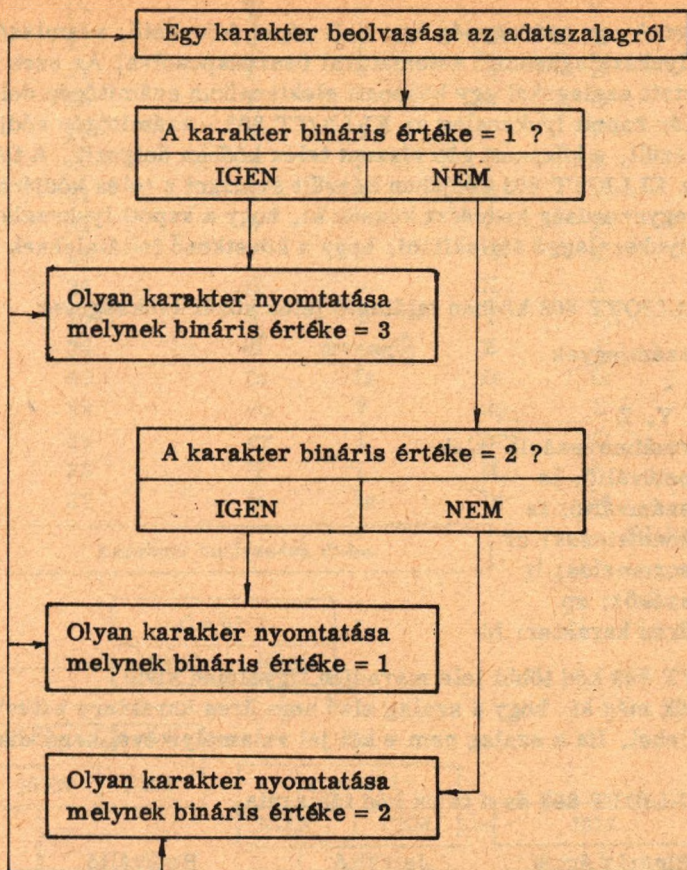
Lyukszalag beolvasásával rendelkező elektronikus számítógépek esetében igen gyakran fordul elő az az eset, amikor olyan lyukszalagot kell a számítógépnek helyesen érzékelnie, melyet nem az adott számítógép saját kódjában lyukasztottak. Ez olyankor fordul elő amikor egy másik, más típusú számítógép meghibásodik, vagy annyira túl van terhelve munkával, hogy lehetetlen minden eredetileg oda előkészített munkát lefuttatni rajta anélkül, hogy a munkák elkészülésének határidejét ne kellene lényegesen későbbre helyezni. De előfordulhat olyan eset is, amikor a munkát már eredetileg is az adott számítógépre szervezték, de a felhasznált programot vagy adatokat nem a számítógépet üzemeltető számítóközpontban lyukasztották lyukszalagra, hanem valahol máshol, ahol nem állt rendelkezésre olyan szalaglyukasztó berendezés, mely azonos kódban működik a számítóközpont szalaglyukasztó berendezéseivel. Vegyük pl. azt a legegyszerűbb esetet, hogy a számítógép ötcsatornás lyukszalaggal képes dolgozni. A számítóközpont szalaglyukasztó gépei is képesek az összes lehetséges lyukkombinációk gépelésére, és egy külső vállalatnál elhelyezett berendezés is. A tulajdonképpeni problémát a következő jelenti:

A számítóközpont berendezései az A betű leütésekor mondjuk a 0 0 0 0 1 a B betű leütésekor a 0 0 0 1 0, a C leütésekor a 0 0 0 1 1 lyukkombinációt lyukasztják (az 1 jelenti a lyukat a szalagon, a 0 pozícióban nincs kilyukasztva a szalag).

A számítógép fordítóprogramjaiba beépített szalagbeolvasó szubrutinok természetesen a számítóközpont szalaglyukasztó berendezéseivel összhangban értelmezik a beolvasott karaktereket, tehát a 0 0 0 0 1-et azonosítják A-nak a 0 0 0 1 0-t B-nek, a 0 0 0 1 1-et C-nek. Ha most a külső vállalat szalaglyukasztó berendezése olyan, hogy az A betű leütésekor a 0 0 0 1 0-t a B leütésekor 0 0 0 1 1-et, a C leütésekor 0 0 0 0 1-et lyukaszt, akkor a számítóközpontba elküldött szalagot a számítógép képes beolvasni, de pl. a külső vállalatnál CBA-nak gépelt betűhármast ACB-nek értelmezi. Elképzelhető, hogy milyen nagy problémát jelentene és hány hibára adna lehetőséget, ha azt mondanák a vállalat gépirónőinek, hogy minden A betű olvasásakor C betűt, minden B betű olvasásakor A betűt és minden C betű olvasásakor B betűt gépeljenek. E helyett a külső vállalat a számítóközpontban elkészített egy olyan programot, mely a lyukszalagot beolvasva azonosítja a vállalatnál használt lyukasztóberendezés karaktereit, és olyan lyukszalagot lyukaszt, mely már alkalmas arra, hogy a fordítóprogramok teljesen értelmezzék. Ettől kezdve nyugodtan lyukasztathatják a vállalatnál a programokat a saját kódjuk alapján.

A következő program a fent leírt esettel foglalkozik.

Blokkdiagram:



A program:

```
SETS I
SETR 4
4) INPUT I
JUMP @ I
1) TITLE C'
JUMP @ 4
2) TITLE A'
JUMP @ 4
3) TITLE B'
JUMP @ 4
START 4
```


7. Feladat

Legyenek egy gyáregység ügyviteli gépei (könyvelő, számlázó, összeadó stb..) egy lyukszalagkészítő automatával összekapcsolva. Az ezen az automata-
tán lyukasztott szalagokat egy központi elektronikus számítógép dolgozza fel.
Az automatán kapott lyukszalag az ELLIOTT 803 - számítógép kódjának meg-
felelően készült, a központi gép viszont telex kódban dolgozik. A feladat tehát
az, hogy az ELLIOTT 803 kódjában készült szalagot a telex kódjára konvertál-
juk át. Az egyszerűség kedvéért kössük ki, hogy a kapott lyukszalagot úgy kell
telex kódu lyukszalaggá átalakítani, hogy a következő feltételeknek kell teljesül-
niük:

Az ELLIOTT 803 kódban található jelek közül szükségesek

a) a számjegyek

b) +, -

c) X, Y, Z

d) formátumvezérlő jelek:

betűváltó: ls

számváltó: fs

kocsivissza: cr

soremelés: lt

szóköz: sp

üres karakter: bl

Az ELLIOTT 803 kód többi jele maradjon figyelmen kívül.

Kössük még ki, hogy a szalag első nem üres karaktere betűváltó, vagy
számváltó lehet. Ha a szalag nem e két jel valamelyikével kezdődik, ez hibát
jelent.

Az ELLIOTT 803 és a telex kód táblázata.

Bináris érték	Jelváltó		Betűváltó	
0	b1	b1	b1	b1
1	1	5	A	T
2	2	cr	B	cr
3	x	9	C	
4	4	sp	D	sp
5			E	H
6	=	,	F	N
7	7	.	G	M
8	8	lf	H	lf
9	,	)	I	L
10	,	4	J	R
11	+		K	G
12	:	8	L	I
13	-	0	M	P
14	.	:	N	C

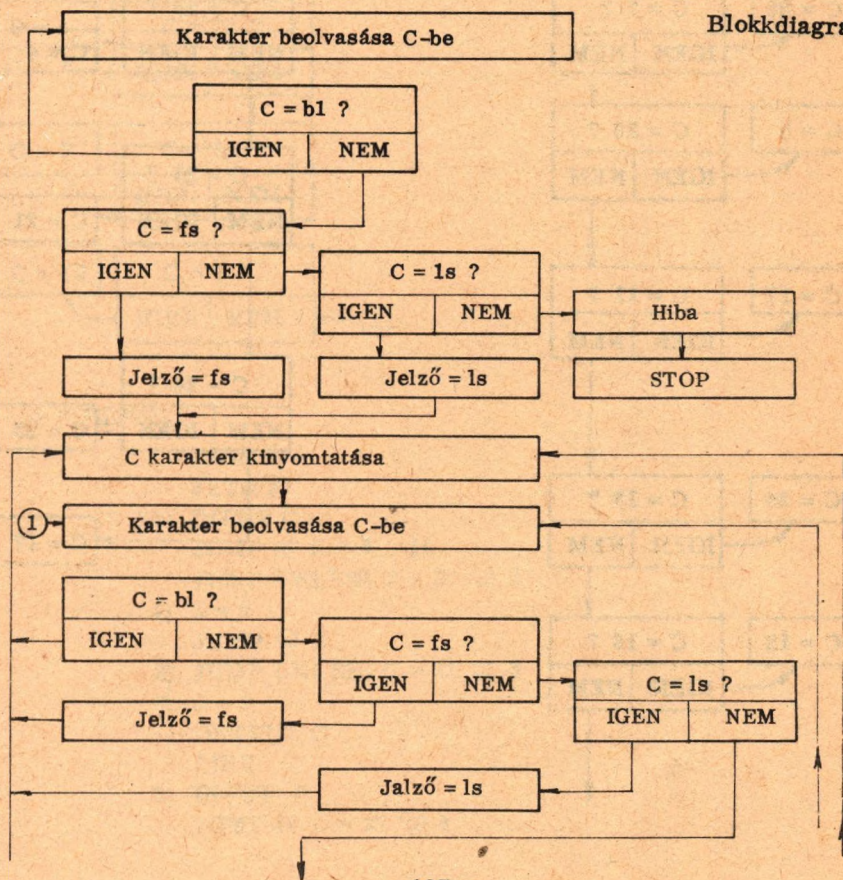
Bináris érték

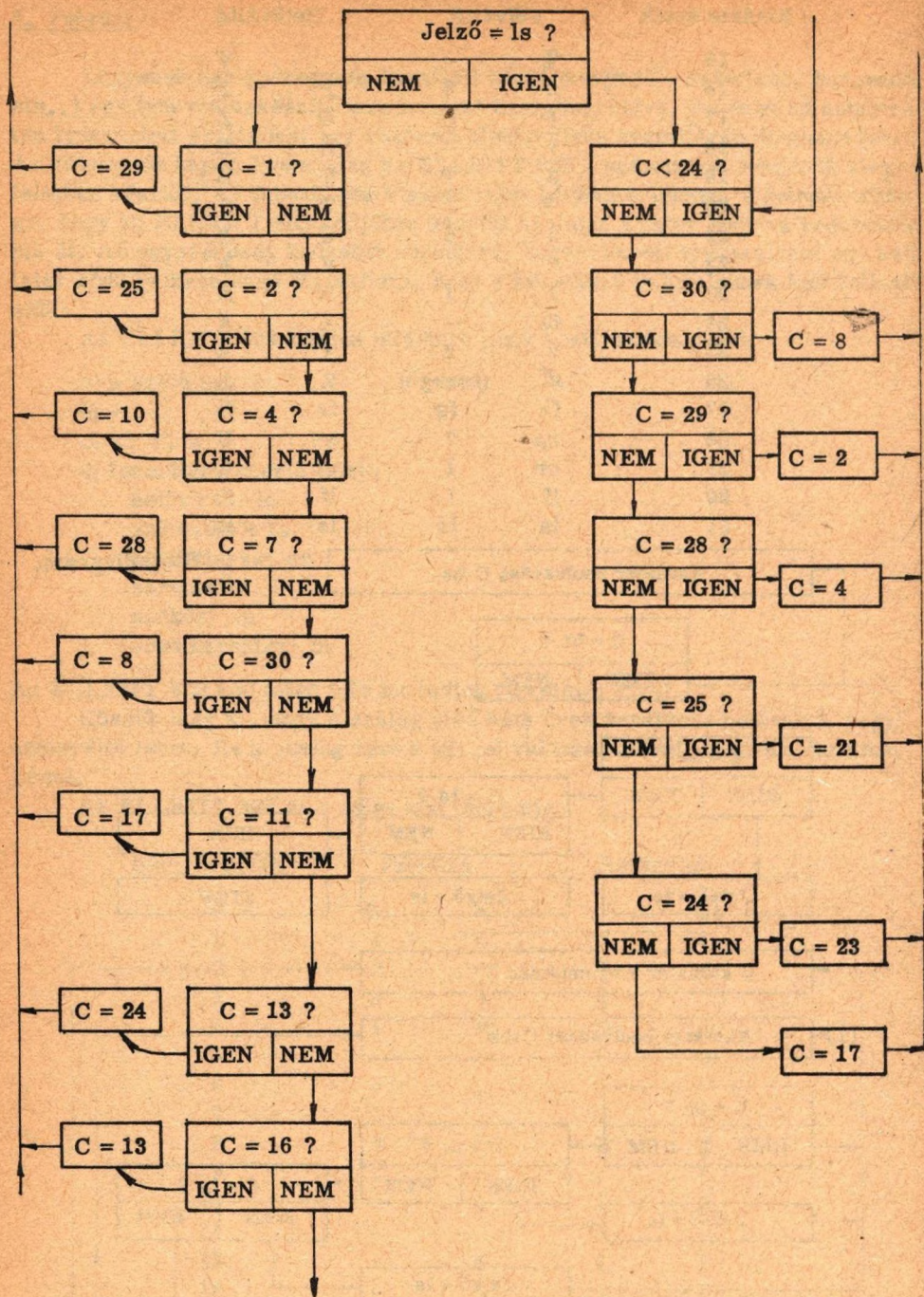
Jelváltó

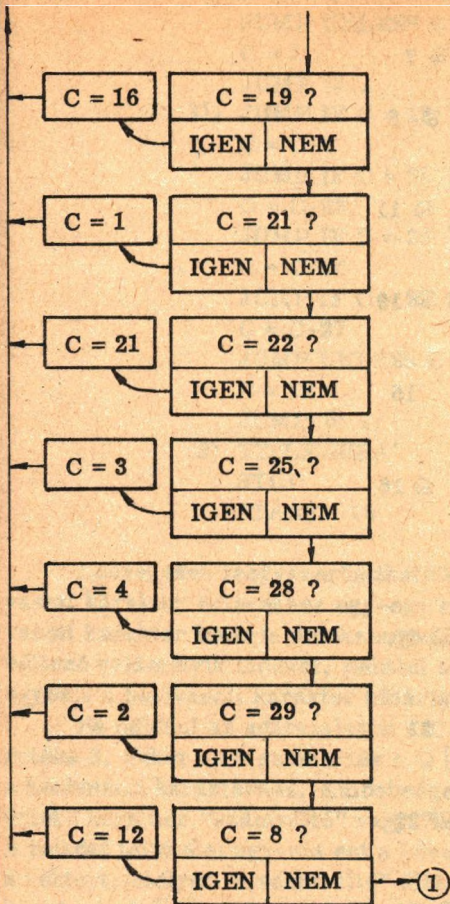
Betűváltó

15	%	=		V
16	0	3	P	E
17	(	+	Q	Z
18	)		R	D
19	3	?	S	B
20	?	,	T	S
21	5	6	U	Y
22	6		V	F
23	/	/	W	X
24	@	-	X	A
25	9	2	Y	W
26	£	(csengő)	Z	J
27	fs	fs	fs	fs
28	sp	7	sp	U
29	cr	1	cr	Q
30	lf	(	lf	K
31	ls	ls	ls	ls

Blokkdiagram:







A program:

```

SETS C J
SETR 35
10) INPUT C
    JUMP IF C = 0 10
    JUMP UNLESS C = 27 @ 2
4) J = 0
    JUMP @ 1
2) JUMP UNLESS C = 31 @ 3
6) J = 1
1) OUTPUT C
    LINE
5) INPUT C
    JUMP IF C = 27 @ 4
  
```


JUMP IF C = 31 a 6
 JUMP IF C = 0 a 1
 JUMP UNLESS J = 1 @ 7
 JUMP IF C < 24 @ 5
 JUMP UNLESS C = 30 @ 8
 9) C = C -22
 JUMP @ 1
 8) JUMP UNLESS C = 29 @ 11
 12) C = C -27
 JUMP @ 1
 11) JUMP UNLESS C = 28 @ 13
 14) C = C -24
 JUMP @1
 13) JUMP UNLESS C = 25 15
 C = 21
 JUMP @ 1
 15) JUMP UNLESS C = 24 @ 16
 17) C = C -1
 JUMP @ 1
 16) C = 17
 JUMP @1
 7) JUMP UNLESS C = 1 @ 20
 C = 29
 JUMP @1
 20) JUMP UNLESS C = 2 21
 C = 25
 JUMP @ 1
 21) JUMP UNLESS C = 4 @ 22
 23) C = C+6
 JUMP @ 1
 22) JUMP UNLESS C = 7 @ 24
 C = 28
 JUMP @ 1
 24) JUMP IF C = 30 @ 9
 C = C -22
 JUMP IF C = 11 @ 23
 C = C+6
 JUMP UNLESS C = 13 @ 27
 C = 24
 JUMP @ 1
 27) JUMP UNLESS C = 16 28
 29) C = C -3
 JUMP @ 1
 28) JUMP IF C = 19 @ 29
 C = C-3


```

JUMP UNLESS C = 21  @ 31
C = 1
JUMP @ 1
31) JUMP IF C = 22  @ 17
C = C-1
JUMP IF C = 25  @ 9
C = C-22
JUMP IF C = 28  @ 14
C = C-24
JUMP IF C = 29  @ 12
C = C-27
JUMP UNLESS C = 8  @ 5
C = 12
JUMP @ 1
3) TITLE HIBA'
STOP
START 10

```

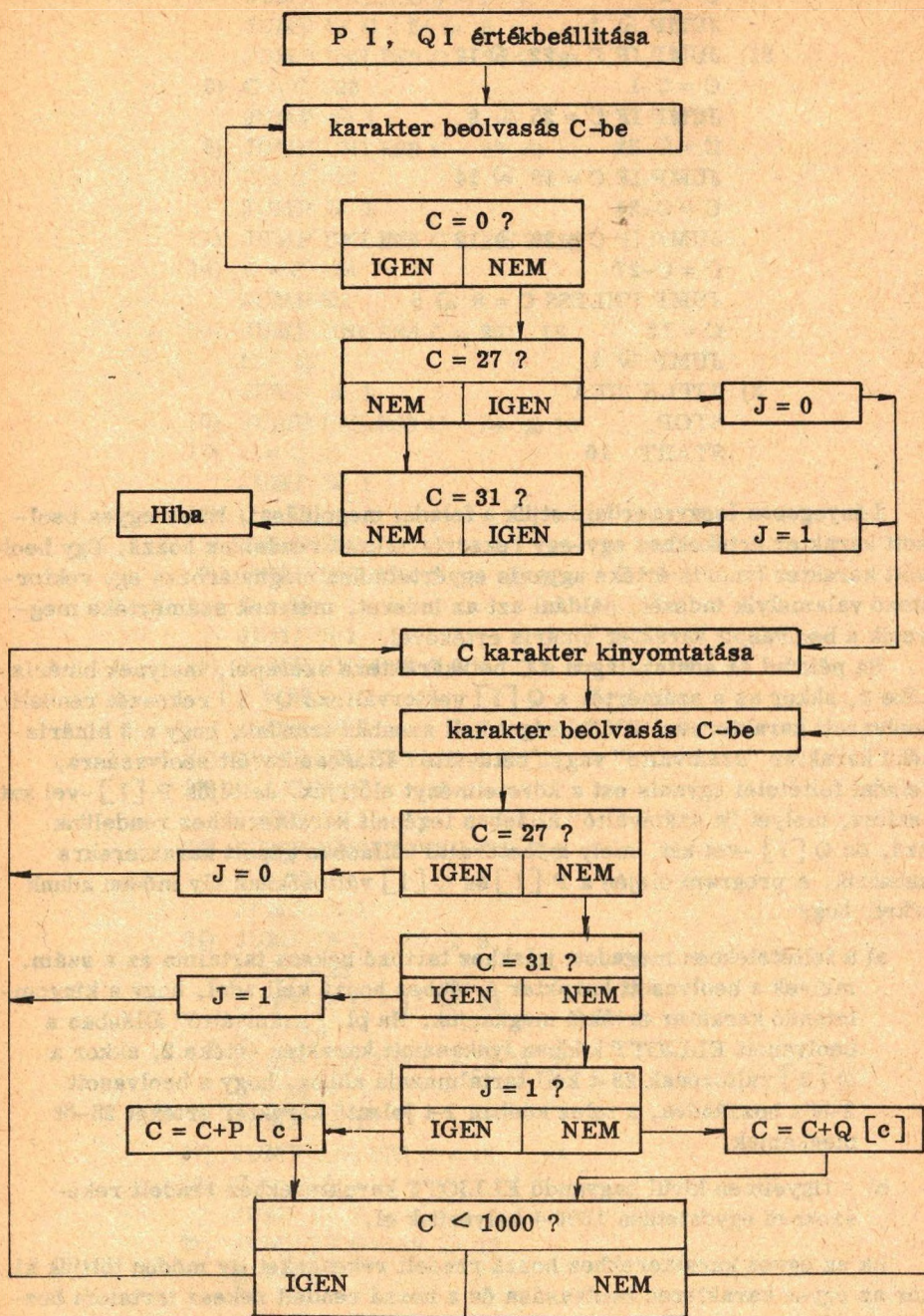
Lényegesen leegyszerűsíthetjük a feladat megoldását, ha az egyes beolvasott karakter értékekhez egy-egy rekesztartalmat rendelünk hozzá. Egy beolvasott karakter bináris értéke ugyanis egyértelműen meghatározza egy vektor-változó valamelyik indexét, például azt az indexet, melynek számértéke meg-egyezik a beolvasott karakter bináris értékével.

Ha például az adatszalonon a C betűkaraktere szerepel, melynek bináris értéke 3, akkor az a számérték a $Q[I]$ vektorváltozó $Q[3]$ rekeszét rendeli a beolvasott karakterhez. Különbséget kell azonban tennünk, hogy a 3 bináris értékű karakter "számváltó" vagy "betűváltó" állásban került beolvasásra. A feladat feltételei ugyanis ezt a követelményt előírják. Jelöljük $P[I]$ -vel azt a vektort, melyet "a számváltó" állásban legépelt karakterekhez rendelünk hozzá, és $Q[I]$ -vel azt, mely a "betűváltó" állásban gépelt karakterekre vonatkozik. A program elején a $P[I]$ és $Q[I]$ változóknak oly módon adunk értéket, hogy

- a) a feltételekben megadott jelekhez tartozó rekesz tartalma az a szám, melyek a beolvasott karakter értékhez hozzá kell adni, hogy a kinyomtatandó karakter értékét megkapjuk. Ha pl. "számváltó" állásban a beolvasott ELLIOTT kódban lyukasztott karakter értéke 2, akkor a $Q[2]$ változónak 23-t kell tartalmaznia ahhoz, hogy a beolvasott 2-höz hozzáadva, a telex kódban 2-t jelentő karakter értéket 25-öt megkapjuk.
- b) A figyelmen kívül hagyandó ELLIOTT karakterekhez rendelt rekeszekben egyöntetűen 1000-t helyezünk el.

Ha az egyes karakterekhez hozzá rendelt rekeszeket ily módon töltjük ki, akkor az egyes karakterek beolvasása és a hozzá rendelt rekesz tartalom hozzáadása után egyetlen feltételes ugró utasítással eldönthető, hogy az új karakter

A blokkdiagram:



értéket a megadott feltételeknek megfelelően ki kell-e nyomtatni, vagy sem. Abban az esetben ugyanis, amikor a karakter értéket nem kell kinyomtatni, a rekesz tartalommal növelt karakter érték meghaladja az 1000-t.

A program:

SETS CJK P[31] Q[31]
SETR 11

1) VARY K= 1:1:31

P[K]=1000

Q[K]= 1000

REPEAT K

2) Q = 0

Q[1] =28

Q[2] =23

Q[4] =6

Q[7] =21

Q[8] =4

Q[11] =6

Q[13] =11

Q[16] =-3

Q[19] =-3

Q[21] =-20

Q[22] =-1

Q[25] =-22

Q[28] =-24

Q[29] =-27

Q[30] =-22

P = 0

P[24] =-1

P[25] =-4

P[26] =-9

P[28] =-24

P[29] =-27

P[30] =-22

3) INPUT C

JUMP IF C = 0 @ 3

JUMP UNLESS C = 27 @ 4

7) J = 0

JUMP @ 5

4) JUMP UNLESS C = 31 @ 6

8) J = 1

5) OUTPUT C

LINE

Valamennyi

P[K] és Q[K] tartalmát egyöntetűen 1000-re állítjuk be, a 2)-referenciaszám utáni részben a feltételek alapján szükséges rekeszekbe a kívánt értékeket vesszük.


```

10) INPUT C
    JUMP IF C = 27 @ 7
    JUMP IF C = 31 @ 8
    JUMP IF J = 1 @ 9
    C = C + Q [c]
11) JUMP UNLESS C < 1000 @ 10
    JUMP @ 5
9) C = C + P [c]
    JUMP @ 11
6) TITLE HIBA'
    LINES 3
    STOP
    START 1

```

Kezdeti értékadó utasításokkal a program még egyszerűbbé tehető:

```

SETS C J K P [31] Q [31]
SETR 11
SET P [24:31] = -1, -4, -9, 1000, -24, -27, -22, 1000
SET Q [0:31] = 0, 28, 23, 2000, 6, 1000, 1000, 21,
               4, 1000, 6, 1000, 11, 1000, 1000,
               -3, 1000, 1000, -3, 1000, -20, -1,
               1000, 1000, -22, 1000, 1000, -24,
               -27, -22, 1000

```

```

1) VARY K = 0:1:25
    P [K] = 1000
    REPEAT K
3) INPUT C
    JUMP IF C = 0 @ 3
    JUMP UNLESS C = 27 @ 4
7) J = 0
    JUMP @ 5
4) JUMP UNLESS C = 31 @ 6
8) J = 1
5) OUTPUT C
    LINE
10) INPUT C
    JUMP IF C = 27 @ 7
    JUMP IF C = 31 @ 8
    JUMP IF J = : @ 9
    C = C + Q [c]
11) JUMP UNLESS C < 1000 @ 10
    JUMP @ 5
9) C = C + P [c]
    JUMP @ 11

```


6) TITLE HIBA'

LINES 3

STOP

START 1

8. Feladat

Egy vállalat készletnyilvántartásában 6 jegyű számmal jelölik a termékeket. E 6 jegyű szám első számjegye alapján a termékek három csoportra oszthatók, mégpedig úgy, hogy a 0, 1, 2, 3, 4, és 5-től kezdődő számok képezik az I csoportot, a 6, 7, 8-al kezdődő számok a II csoportot, a 9-cel kezdődő kódszámok pedig a III. csoportot. Mindegyik kódszámot a kódszám által meghatározott termék nyilvántartásra kerülő adatai követik, mégpedig úgy, hogy az I csoporthoz tartozó kódszám után 3 adat,

a	II	"	"	"	"	8 adat,
---	----	---	---	---	---	---------

a	III	"	"	"	"	2 adat
---	-----	---	---	---	---	--------

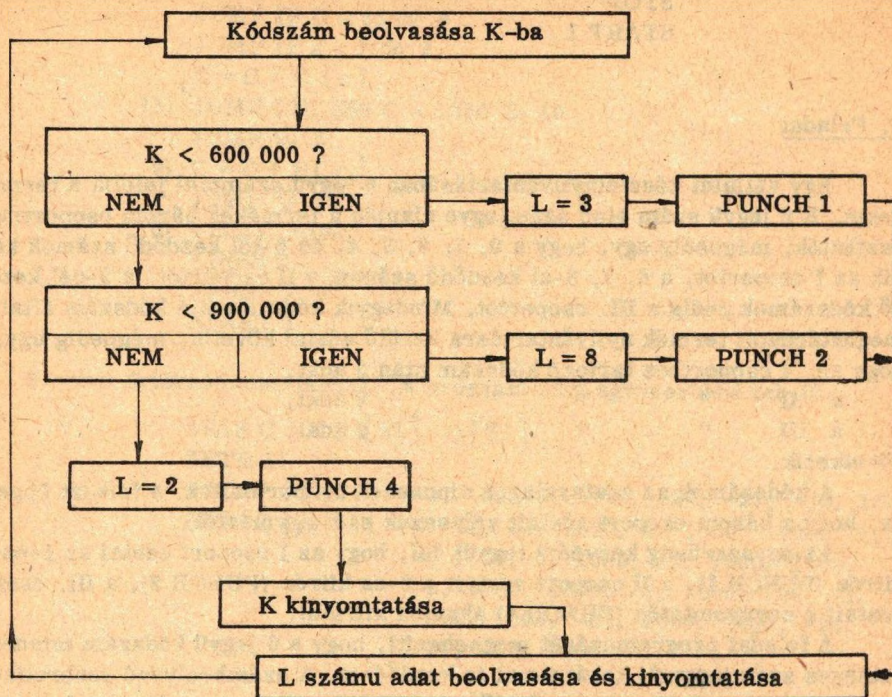
következik.

A kódszámok az adatszalagon nincsenek csoportosítva, a feladat éppen az, hogy a három csoport adatait válasszuk szét egymástól.

Az egyszerűség kedvéért tegyük fel, hogy az I csoport adatai az 1-es kiírón (PUNCH 1), a II csoport adatait a 2-es kiírón (PUNCH 2), a III. csoport adatait a sornyomtatón (PUNCH 4) akarjuk kiíratni.

A feladat programozását megnehezíti, hogy a 6 jegyű kódszám minden lehetséges számjeggyel, tehát 0-val is kezdődhet. A számbeolvasó szubrutinok ugyanis a szám elején elhelyezkedő 0-kat beolvassák ugyan, de a beolvasott szám ezeket már nem tartalmazza és így semmiféle információnk nincs arra nézve, hogy az adatszalagon volt-e szám elejei 0 vagy sem. Ebben a feladatban azonban kihasználhatjuk azt a tényt, hogy minden kódszám 6 jegyű és a 0-val kezdődő számok azonos csoportba esnek az 1, 2, 3, 4 és 5-tel kezdődő kódszámokkal.

A program blokkdiagramja:



A program:

SETS AKIL

SETR 5

1) READ K

JUMP IF K < 600 000 @ 2

JUMP IF K < 900 000 @ 3

L = 2

PUNCH 4

5) PRINT K

VARY I = 1:1:L

READ A

PRINT A

REPEAT I

LINE

JUMP @ 1

2) L = 3

PUNCH 1

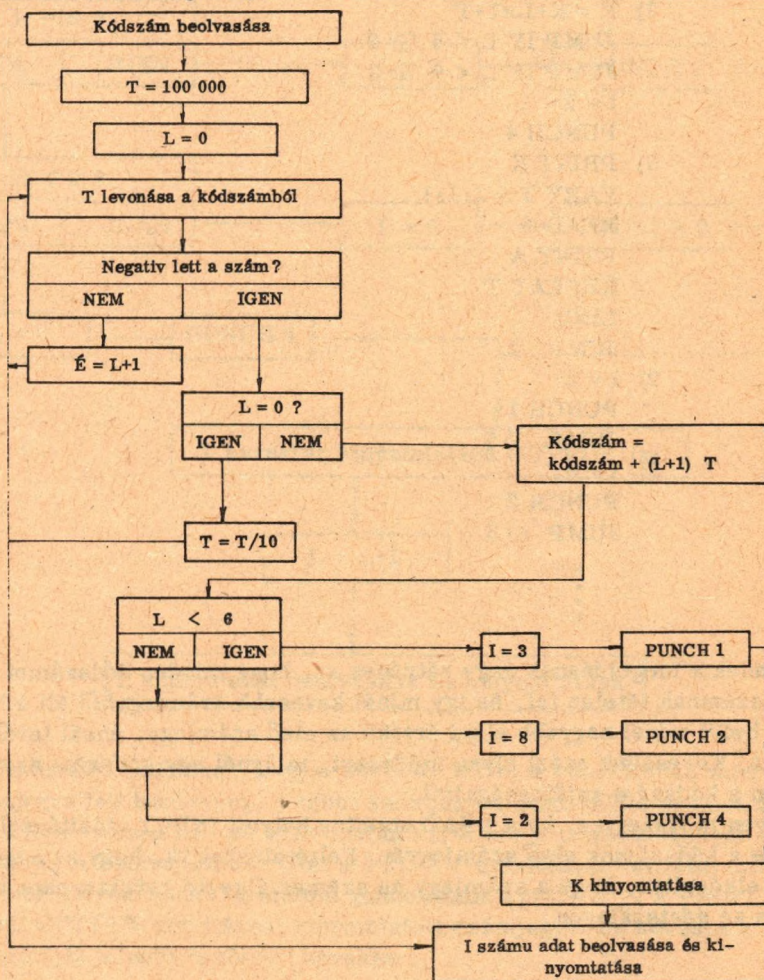
JUMP @ 5

3) $L = 8$
 PUNCH 2
 JUMP @ 5
 START 1

9. Feladat

A feladat hasonló az előzőhöz, de most az 1, 2, 3, 4, 5-tel kezdődő kódszámok képezik az I, a 6, 7, 8-al kezdődő kódszámok a II, és a 9-cel kezdődő kódszámok a III. csoportot. A kódszámok ebben a feladatban álljanak különböző számú számjegyből. A kódszámok jegyeinek száma legalább 2, de legfeljebb 6 lehet. Ez azt jelenti, hogy a csoport azonosító kezdő számjegy után még legalább 1, de legfeljebb 5 számjegy következik.

A feladat blokkdiagramja:



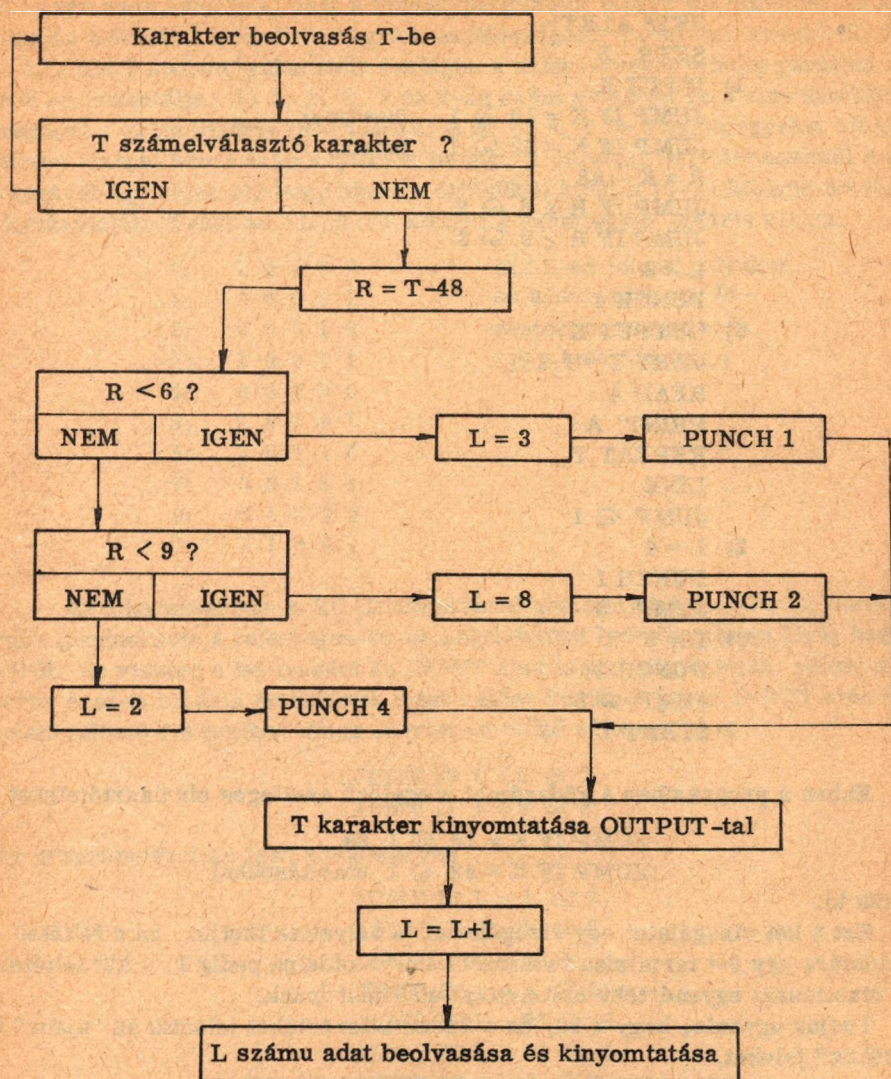
A feladat programja:

```
SETS KLTIA
SETR 10
1) READ K
L = 0
CYCLE T = 100 000, 10 000, 1000, 100, 10
6) K = K - T
JUMP IF K < 0 @ 4
L = L + 1
JUMP a 6
4) JUMP UNLESS L = 0 @ 7
K = K + T
REPEAT T
7) K = K + L * T + T
JUMP IF L < 6 @ 2
JUMP IF L < 9 @ 3
I = 2
PUNCH 4
5) PRINT K
VARY T = 1:1:I
READ A
PRINT A
REPEAT T
LINE
JUMP 1
2) I = 3
PUNCH 1
JUMP @ 5
3) I = 8
PUNCH 2
JUMP @ 5
:
:
:
```

Ennek a megoldásnak nagy hátránya az, hogy minden kódszámot eleve 6 jegyű számnak tételez fel, és így minél kevesebb számjegyből áll a kódszám, és ezen belül minél nagyobb alaki értékű az első számjegy, annál tovább tart számítás. Keressünk ezért olyan módszert, melynél egy kódszám azonosítása független a kódszám sajátosságaitól.

Ilyen módszer az, ha a READ utasítás helyett INPUT utasítással olvashatjuk be a kódszámok első számjegyét. Feltételezhetjük, hogy az adatszalog úgy van elkészítve, hogy a számjegy és számelvlasztó karaktereken kívül más jel nincs az adatszalongon.

Blokkdiagram:



Az L értékét azért kell 1-gyel növelni, mivel a kódszámnak még csak az első számjegye lett beolvasva, a többi számjegyet mint önálló számot az első READ utasítás olvassa be, és csak ezután kerül sor a tulajdonképpeni adatok beolvasására. A beolvasott adatokat a PRINT ' utasítással kell kinyomtatni, mert csak ez a nyomtatási formátum gondoskodik arról, hogy a PRINT' előtt közvetlenül OUTPUT utasítással kinyomtatott számjegyet ne kövesse a PRINT előjel pozícióját jelentő "szóköz" karakter.

A program:

```
SETS ALKTR
SETR 5
1) INPUT K
JUMP IF K = 10 @ 1 (newline)
JUMP IF K = 32 @ 1 (space)
R = K - 48
JUMP IF R < 6 @ 2
JUMP IF R < 9 @ 3
L = 3
PUNCH 4
5) OUTPUT K
VARY T = 1:1:L
READ A
PRINT' A
REPEAT T
LINE
JUMP @ 1
2) L = 4
PUNCH 1
JUMP 5
3) L = 9
PUNCH 2
JUMP @ 5
START 1
```

Ebben a programban a kódszámot megelőző esetleges elválasztójeleket a

```
JUMP IF K = 10 @ 1 és
JUMP IF K = 32 @ 1 utasításokkal
```

szűrtük ki.

Ezt a két vizsgálatot egy vizsgálattal is helyettesíthetjük, ha a feltétel bal oldalára egy 0-t tartalmazó változót, a jobb oldalra pedig 1, a két feltételes ugró utasítással egyenértékű aritmetikai utasítást írunk.

Tudjuk ugyanis, hogy a 10, és a 32 karakterértéket jelentik az "uj sor" és a "szóköz" jeleket.

Az $(K = 10) (K - 32) = 0$ szorzat eleget tesz a kívántaknak. Ez a szorzat ugyanis akkor és csak akkor 0, ha valamelyik tényezője 0. Ez pedig csak akkor teljesül, ha vagy $K = 10$, "uj sor" karakter beolvasásakor, vagy $K = 32$, "szóköz" karakter beolvasása esetén.

$$(K - 10) (K - 32) = K^2 - 42K + 320$$

Tehát a programrész:

```
1) INPUT K
D = 0
JUMP IF D = K*K - 42 K + 320 @ 1
```


Az ELLIOTT 803-B autókódjában ennek a feladatnak egyszerűbb megoldása is van azok számára, akik a fordító programot eléggé jól ismerik, és így kitudják használni a fordító program sajátosságait. Tudni kell ezenkívül, hogy az ELLIOTT kódban lyukasztott szalagon a számjegyeket mindig páratlan számú lyuk reprezentálja. Az 1, 2, 4, 7 és 8-as szám esetében ez a reprezentáció megfelel a szám bináris felírásának. A 3, 5, 6, 9 és 0 számjegyeket binárisan felírva viszont páros számú lyukhoz jutunk. A páratlan paritást ezeknél a számjegyeknél úgy érjük el, hogy bináris értékükön kívül a 16-os bináris értéket is kilyukasztják. Tehát az ELLIOTT kódban a számjegyek bináris alakja:

0:	1 0 0 0 0	(Az 1-es jelenti a lyukat
1:	0 0 0 0 1	az adatszalog egy öt-
2:	0 0 0 1 0	csatornás karakterén,
3:	1 0 0 1 1	a 0 a lyuk hiányát.)
4:	0 0 1 0 0	
5:	1 0 1 0 1	
6:	1 0 1 1 0	
7:	0 0 1 1 1	
8:	0 1 0 0 0	
9:	1 1 0 0 1	

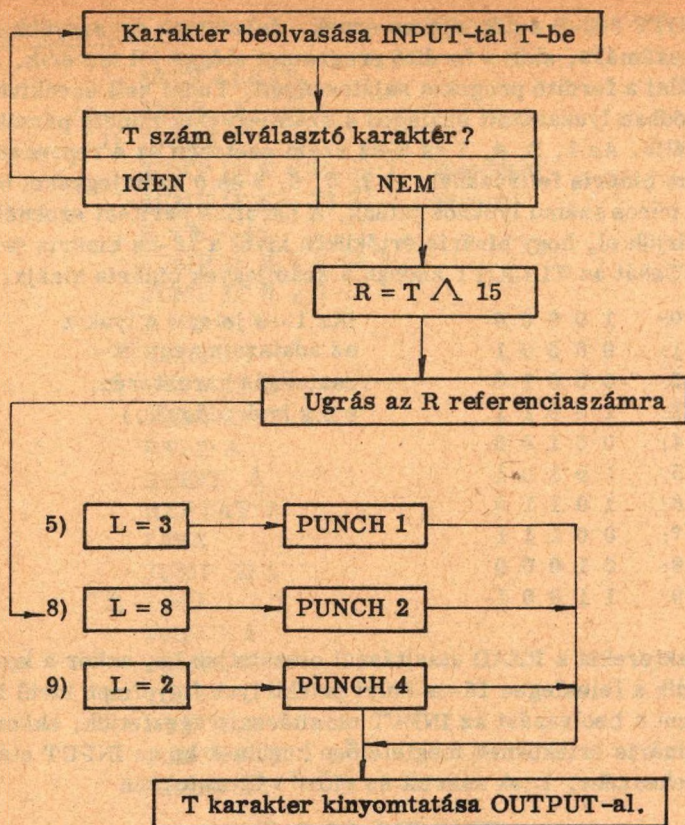
Ha a karaktereket a READ utasítással olvastatjuk be, akkor a konverziós rutin gondoskodik a felesleges 16-os helyi értékű lyuk figyelmen kívül hagyásáról. Ha viszont a beolvasást az INPUT utasítással végeztetjük, akkor a karakterek eredeti bináris értéküknek megfelelően kerülnek be az INPUT után feltüntetett változó rekeszébe. Ezek szerint az előző változatokban

```
JUMP IF L < 6 @ 2
JUMP IF L < 9 @ 3
```

írt programrészt az igen körülményes

```
JUMP IF L = 1 @ 2
JUMP IF L = 2 @ 2
JUMP IF L = 19 @ 2
JUMP IF L = 4 @ 2
JUMP IF L = 21 @ 2
JUMP IF L = 22 @ 3
JUMP IF L = 7 @ 3
JUMP IF L = 8 @ 3
```

utasítás sorozattal kellene helyettesíteni. Az ELLIOTT autókód fordítóprogram sajátosságait jól ismerők viszont programjukat a következő blokkdiagram alapján készíthetik el.



Az ELLIOTT 803 program:

10) M = 15

11) INPUT T

JUMP IF T > 26 @ 11

(számelválasztó karakterek:
szóköz: 28 bináris érték
soremelés: 30 bináris érték)

@ 30 T : 03 M

20 R : 00 0

JUMP @ R

5) L = 3

PUNCH 1

JUMP @ 12

8) L = 8

PUNCH 2

JUMP @ 12

(A gépi kód blokk a logikai sorozást és az eredmény R-be vitelét végzi el.)

- 9) L = 2
PUNCH 4
- 12) OUTPUT T (A kódszám többi számjegyét is
INPUT T INPUT-tal kell beolvasni, mivel
JUMP IF T = 0 @ 12 az ELLIOTT autokódban PRINT'
JUMP IF T > 26 @ 12 típusu utasítás nincs.)

VARY I = 1:1:T
READ A
PRINT A
REPEAT I
LINE
JUMP @ 11
STOP 10

Ha megfigyeljük a programot láthatjuk, hogy csak az 5) 8) 9) 10) 11) és 12) referenciaszámok fordulnak elő, annak ellenére, hogy a

JUMP @ R utasítás aktuális jelentése
JUMP @ 1, JUMP @ 2, JUMP @ 3, ... JUMP @ 9 lehet.

Éppen a referenciaszámok hiánya az, amit a fordító program ismerője kihasználhat. E program alapján ugyanis a referenciaszám táblázat a következőképpen alakul.

1) Tárolója	üres	üres	2) tárolója
3) "	üres	üres	4) "
5) "	ugrás a program	üres	6) "
	5) -referenciájára		
7) "	üres	ugrás a program	8) "
		8) -referenciájára	
9) "	ugrás a program	ugrás a program	10) "
	9) -referenciájára	10) -referenciájára	
11) "	ugrás a program	ugrás a program	12) "
	11) -referenciájára	12) -referenciájára	

Ha most a JUMP @ R aktuális jelentése JUMP @ 3, a program futása a 3) referenciaszám tárolójánál folytatódik. Mivel ez üres utasítás, a következő végrehajtásra kerülő utasítás a 4) referenciaszám tárolójában található utasítás lesz. Mivel ez is üres utasítás hatására az 5) referenciaszám tárolójának utasítása kerül végrehajtásra. Ez az utasítás pedig a program futását az 5) referenciaszámmal megjelölt pontnál folytatja.

Az EMG 830-as autokódban ezt a megoldást nem választhatjuk a referenciaszám táblázat másféle elrendezése miatt.

Az EMG 830-as autokód program esetén ugyanis a referenciaszám táblázat így alakul:

(Tegyük fel, hogy ebben a programban is csak az 5) 8) 9) 10) 11) és 12) referenciaszámokat használtuk fel):

1) Tárolója	Ugrás hibajelzésre
2) "	Ugrás hibajelzésre
3) "	Ugrás hibajelzésre
4) "	Ugrás hibajelzésre
5) "	Ugrás a program 5) referenciájára
6) "	Ugrás hibajelzésre
7) "	Ugrás hibajelzésre
8) "	Ugrás a program 8) referenciájára
9) "	Ugrás a program 9) referenciájára
10) "	Ugrás a program 10) referenciájára
11) "	Ugrás a program 11) referenciájára
12) "	Ugrás a program 12) referenciájára

EMG 830-as autokódban tehát ha a JUMP @ R utasítás aktuális érték megfelel a JUMP @ 3 utasításnak, a programfutás a hibajelző szubrutintól folytatódik. A hibajelző szubrutin kiírja, hogy "hivatkozás nem létező referenciaszámra" és STOP-pal megáll.

Az EMG 830 autokódban is van mód arra, hogy ügyes programozási fogással egyszerűbbé tehesünk egy programot. Ennek illusztrálására megmutatjuk azt az EMG 830 autokódban írt programot, mely a legutóbbi blokkdiagram alapján készült.

Az EMG 830 program:

```

11) INPUT T
    JUMP IF T = 10 @ 11 Számelvásztó
    JUMP IF T = 32 @ 11 karakterek kiszűrése
    R = T - 48 Számjegyekarakterek bináris
                  értékének helyreállítása

    JUMP @ R
1)2)3)4)5) L = 4
    PUNCH 1
    JUMP @ 12
6)7)8) L = 9
    PUNCH 2
    JUMP @ 12
9) L = 3
    PUNCH 4
    OUTPUT T

    VARY I = 1:1:L
    READ A
    PRINT' A
    REPEAT I
    LINE
    JUMP @ 11
    START 11

```


A programozási fogás tehát az, hogy egy utasítást több referenciaszám-mal láttunk el. Ellentétben az ELLIOTT 803 autokóddal, mely nem engedi meg, csak egyetlen referenciaszám feltüntetését egy utasítás előtt, az EMG 830 autokód tetszés szerinti számu referencia számot elfogad egy utasítás előtt. A fenti programban bemutatott lehetőségen kívül nyomkövetési célokból is el lehet látni egy utasítást több referenciaszámmal. Ha pl. egy utasítás előtt a 3) 7) 2) 11) referenciaszám sorozat állt, és JUMP 11 utasítás alapján került sor a referenciaszámokat követő utasítás végrehajtására, referenciaszám nyomkövetéses futtatás esetén csak a 11) referenciaszám kerül kinyomtatásra, míg a JUMP 3 utasítás következménye a 3), 7), 2), 11) referenciaszámok kinyomtatása lesz.

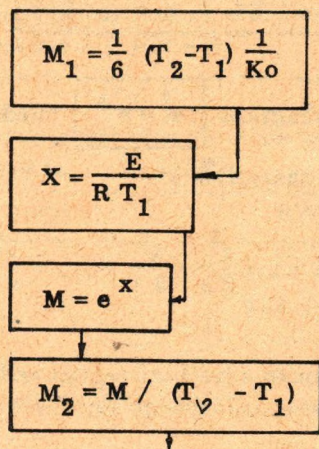
10. Feladat

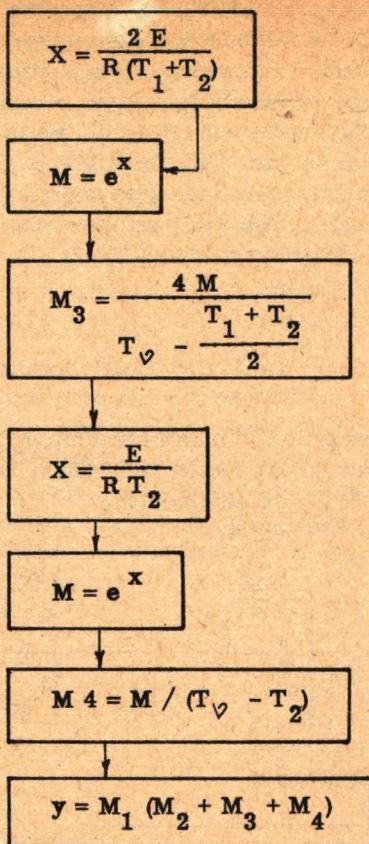
Egy kémiai üzembn azt az időt kell kiszámítani, mely monomolekuláris irreverzibilis adiabatikus reakció esetén kell ahhoz, hogy valamely reagáló anyag hőmérséklete T_1 -ről T_2 -re emelkedjen.

Az alkalmazásra kerülő képlet:

$$y = \frac{1}{6} (T_2 - T_1) \frac{1}{K_0} \left[\frac{\frac{E}{RT_1}}{T_v - T_1} + \frac{4e \frac{2E}{2(T_1 + T_2)}}{T_v - \frac{T_1 + T_2}{2}} + \frac{\frac{E}{RT_2}}{T_v - T_2} \right]$$

A program blokkdiagramja:





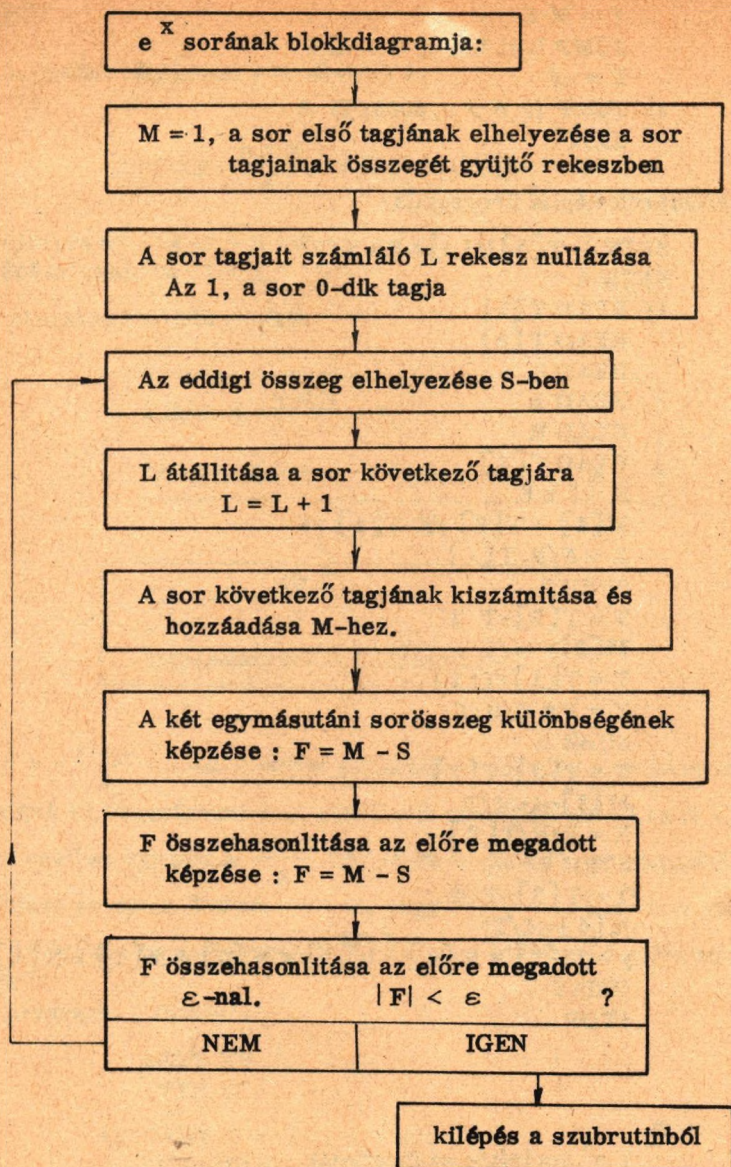
A programban a blokkdiagram $M = e^X$ blokkját szubrutinnal hívjuk meg.

Az e^X -et sorfejtéssel számoljuk ki

$$e^X = 1 + X + \frac{X^2}{2!} + \frac{X^3}{3!} + \dots + \frac{X^n}{n!}$$

végtelen sor alapján.

E végtelen sor egymásután következő tagjait addig összegezzük, míg két egymásután kiszámított összeg különbsége egy előre adott kis ε szám alá nem esik.



Az e^x sorfejtés szubrutinja.

- 2) $M = 1$
 $L = 0$
- 3) $S = M$
 $L = L + 1$
 $M = S + S \cdot X / L$


```

F = M-S
JUMP UNLESS F < 0  4
F = -F
4) JUMP IF F > 0.00001  3
EXIT

```

Az alkalmazott képlet programja:

```

SETV L T[3]M[4]KER XYSF
SETR 4

```

```

1) READ T[1]
   READ T[2]
   READ K
   READ E
   READ R
   READ T[3]
   K = 6 * K
   M[1] = T[2] / K - T[1] / K
   X = E / R / T[1]
   SUBR 2
   T = T[3] - T 1
   M[2] = M / T
   T = T[1] + T[2]
   X = 2 * E / R / T
   SUBR 2
   T = T[3] - T[2]
   M[3] = 4 * M / T
   X = E / R / T[2]
   SUBR 2
   T = T[3] - T 2
   M[4] = M / T
   Y = M[1] * M[2] + M[1] * M[3] + M[1] * M[4]
   PRINT Y
   STOP

```

```

.
.
.

```

```

2) ( ex sorfejtés szubrutinja)

```

```

.
.
.

```

```

EXIT
START 1

```


11. Feladat

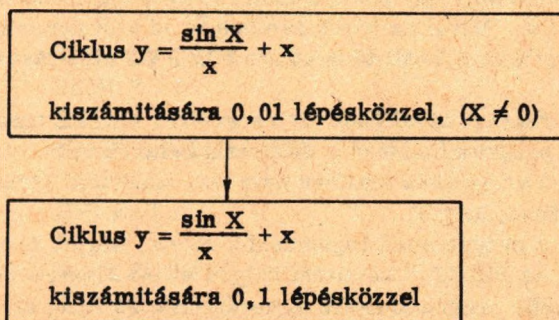
Készítsünk táblázatot y értékeiről az

$$y = \frac{\sin X}{X} + x \text{ képlet alapján,}$$

0,01 lépésközzel, ha $-1 \leq X \leq 1$

0,1 lépésközzel, ha $1 < X \leq 10$

A program blokkdiagramja



Az $y = \frac{\sin x}{x} + x$ -et mindkét ciklusban ugyanazon a szubrutinra hivatkozva számítjuk ki. Vigyáznunk kell a ciklusok végrehajtásával az $X = 0$ esetben. Ebben az esetben ugyanis a $\frac{\sin x}{x}$ képletben 0-val kellene osztanunk, ami tulcsorduláshoz vezetne. Tudjuk viszont, hogy $\frac{\sin x}{x} = 1$ (L'Hospital - szabály alapján) és így ezt az esetet nem a $\frac{\sin x}{x} + x$ -et kiszámoló szubrutinnal számoltatjuk ki.

A program: (Hibás!)

SETV XY

SETR 10

4) VARY X = -1: . 01: 201

JUMP IF X = 0 $\odot$ 2

SUBR 1

REPEAT X

VARY X = 1.1: . 1 : 90

SUBR 1

REPEAT X

STOP

- 1) $Y = Y / 3.14\ 159$
 $Y = \sin Y/X+X$
- 3) PRINT X
 PRINT Y
 LINE
 EXIT
- 2) $Y = 1$
 JUMP @ 3
 START 4

A program készítője úgy gondolkozott, hogy X és Y kinyomtatását is a szubrutinban végzi el. Mivel az $X = 0$ esetben is ki kell nyomtatni X és Y értékét ezért az Y értékének beállítása után a JUMP @3 utasítással a program futását a PRINT X

PRINT Y program utasításokhoz vezérelte. A LINE utasítást követő EXIT hatására, a programfutásnak a szubrutin behívás utáni utasítástól kellene folytatódnia. Ebben az esetben azonban nem volt szubrutin hívás így a programfutás hibajelzéssel megáll.

Amennyiben a programozó ragaszkodik ahhoz, hogy a programban csak egyetlen PRINT X és PRINT Y utasítást helyez el (ez ebben a programban nem indokolt, de hosszabb programrészek egyezése esetén igen) akkor a programot a következő módon készítheti el:

- ```

SETS I
SETV XY
SETR 10
4) VARY X = -1: . 01:201
 I = 0
 JUMP IF X = 0 @ 2
 SUBR 1
5) REPEAT X
 VARY X = 1.1: .1:90
 SUBR 1
 REPEAT X
 STOP
1) $Y = X/3.14159$
 $Y = \sin Y/X+X$
3) PRINT X
 PRINT Y
 LINE
 JUMP IF I = 1 @ 5
 EXIT
2) $Y = 1$
 I = 1
 JUMP @ 3
 START 4

```



Ebben a programban a

```
3) PRINT X
 PRINT Y
 LINE
```

JUMP IF I = 1 @ 5 utasítások minden esetben a SUBR 1 utasításait alkotják, kivéve az X = 0 esetet, amikor a fenti utasítások a főprogramnak szubrutinon kívüli utasításainak számítanak. Ebben az X = 0 esetben helyes működésükről az I kapcsoló átállítása és a ciklus REPEAT utasítására való visszaugrás gondoskodik.

Elkészíthetjük a programot szubrutinok alkalmazásával is:

```
SETV XY
SETR 10
4) VARY X = -1: .01: 201

 JUMP IF X = 0 @ 2
 SUBR 1
5) REPEAT X
 VARY X = 1.1: .1: 90
 SUBR 1
 REPEAT X
 STOP
1) Y = X/3.14159
 Y = SIN Y/X+X
3) PRINT X
 PRINT Y
 LINE
 EXIT
2) Y = 1
 SUBR 3
 JUMP @ 5
 START 4
```

Ebben az esetben a programban két egymástól függetlenül működő szubrutin van: a SUBR 1, és a SUBR 3. Mindkét szubrutint ugyanaz az EXIT zárja le. A SUBR 3 hivatkozásnál ügyelnie kell a programozónak arra, hogy az EXIT utasítás végrehajtása után a ciklus REPEAT-jára vonatkozó JUMP @ 5 utasítást ne felejtse ki a programjából.

A két különböző lépésközzel végrehajtott ciklus egy ciklussá is egyenlíthető: Ebben az esetben:

```
SETV XY A
SETR 10
4) A = . 001
 VARY X = -1:A: 291
 SUBR 1
 REPEAT X
 STOP
```



```

1) Y = 1
 JUMP IF X = 0 @ 3
 Y = X/3.1415
 Y = SIN Y/X+X

3) PRINT X
 PRINT Y
 LINE
 JUMP IF X=1 @ 7
 EXIT

7) A = .1
 EXIT
 START 4

```

Ebben a programban a szubrutin állítja át a ciklus végrehajtás lépésköz értékét. A programban a két EXIT ellenére csak egyetlen szubrutin szerepel. Mindkét EXIT utasítás a SUBR 1 befejezési pontját jelöli, és így mindkettő végrehajtása után a programfutás a REPEAT X utasítástól folytatódik. Ebben a programváltozatban az  $X = 0$  esetet a SUBR 1-ben vizsgáljuk meg. Hogy ekkor  $Y=1$  legyen, ezért minden más esetben is, közvetlenül a szubrutinba ugrás után Y felveszi ezt az értéket.

Ha ugyanis a JUMP IF  $X=0$  3 utasítás hatástalan az  $Y=X/3.14159$  és  $Y=\sin Y/X+X$  utasítások hatására Y-nak X-től függő aktuális értéke kerül kiszámításra.

## 12. Feladat

Rendezzünk nagyság szerint sorrendbe  $n$  db pozitív egész számot. A rendezést e számok bitenkénti rendezésével végezzük el. E módszer lényegét az alábbi példán mutatjuk meg.

Legyen 10 db számunk az A 1 , ... A 10 változóiban.  
5, 2, 8, 11, 33, 1, 4, 2, 5 és 7

E számok bináris felírásban:

|     |             |
|-----|-------------|
| 5:  | 0 0 0 1 0 1 |
| 2:  | 0 0 0 0 1 0 |
| 8:  | 0 0 1 0 0 0 |
| 11: | 0 0 1 0 1 1 |
| 33: | 1 0 0 0 0 1 |
| 1:  | 0 0 0 0 0 1 |
| 4:  | 0 0 0 1 0 0 |
| 2:  | 0 0 0 0 1 0 |
| 5:  | 0 0 0 1 0 1 |
| 7:  | 0 0 0 1 1 1 |



Irjuk át azokat a számokat a  $B [I]$  ( $I=1, \dots, 10$ ) rekeszekbe, melyeknek utolsó bináris jegye 1.

Irjuk át azokat a számokat a  $C [K]$  ( $K=1, \dots, 10$ ) rekeszekbe, melyeknek utolsó bináris jegye 0.

Vigyük vissza azután előbb a betöltött  $B [I]$ , majd folyamatosan a betöltött  $C [K]$  rekeszek tartalmát  $A [J]$  -be ( $J = 1, \dots, 10$ )

E lépés után a számok sorrendje:

```
5: 0 0 0 1 0 1
11: 0 0 1 0 1 1
33: 1 0 0 0 0 1
1: 0 0 0 0 0 1
5: 0 0 0 1 0 1
7: 0 0 0 1 1 1
2: 0 0 0 0 1 0
8: 0 0 1 0 0 0
4: 0 0 0 1 0 0
2: 0 0 0 0 1 0
```

Végezzük el ugyanezt az eljárást úgy, hogy most a számokat jobbról második bitje szerint rendezzük.

A második lépés után:

```
11: 0 0 1 0 1 1
7: 0 0 0 1 1 1
2: 0 0 0 0 1 0
2: 0 0 0 0 1 0
5: 0 0 0 1 0 1
33: 1 0 0 0 0 1
1: 0 0 0 0 0 1
5: 0 0 0 1 0 1
8: 0 0 1 0 0 0
4: 0 0 0 1 0 0
```

A harmadik lépésben a jobbról harmadik bit szerint rendezzünk. A harmadik rendezési lépés elvégzése után:

```
7: 0 0 0 1 1 1
5: 0 0 0 1 0 1
5: 0 0 0 1 0 1
4: 0 0 0 1 0 0
11: 0 0 1 0 1 1
2: 0 0 0 0 1 0
2: 0 0 0 0 1 0
33: 1 0 0 0 0 1
1: 0 0 0 0 0 1
8: 0 0 1 0 0 0
```



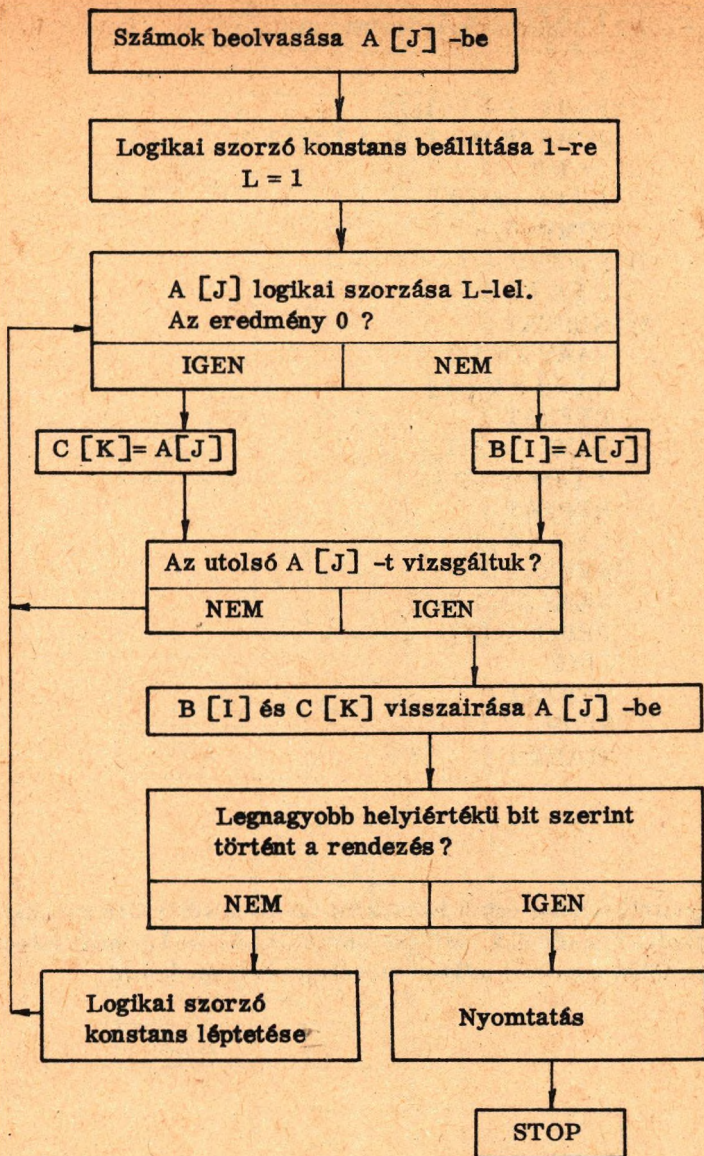
A negyedik lépésben a negyedik bit értéke szerint válogatunk: A ciklus végrehajtása után:

|     |             |
|-----|-------------|
| 11: | 0 0 1 0 1 1 |
| 8:  | 0 0 1 0 0 0 |
| 7:  | 0 0 0 1 1 1 |
| 5:  | 0 0 0 1 0 1 |
| 5:  | 0 0 0 1 0 1 |
| 4:  | 0 0 0 1 0 0 |
| 2:  | 0 0 0 0 1 0 |
| 2:  | 0 0 0 0 1 0 |
| 33: | 1 0 0 0 0 1 |
| 1:  | 0 0 0 0 0 1 |

A jobbról ötödik bittel végzett rendezés a fenti sorrenden nem változtat, de a jobbról hatodik bittel végzett rendezés után a sorrend:

|     |             |
|-----|-------------|
| 33: | 1 0 0 0 0 1 |
| 11: | 0 0 1 0 0 1 |
| 8:  | 0 0 1 0 0 0 |
| 7:  | 0 0 0 1 1 1 |
| 5:  | 0 0 0 1 0 1 |
| 5:  | 0 0 0 1 0 1 |
| 4:  | 0 0 0 1 0 0 |
| 2:  | 0 0 0 0 1 0 |
| 2:  | 0 0 0 0 1 0 |
| 1:  | 0 0 0 0 0 1 |





```

SETS I J K L A[10] B[10] C[10] M T
SETR 3
1) VARY J = 1:1:10
 READ A [J]
 REPEAT J
 M = 0
 L = 1

```



```

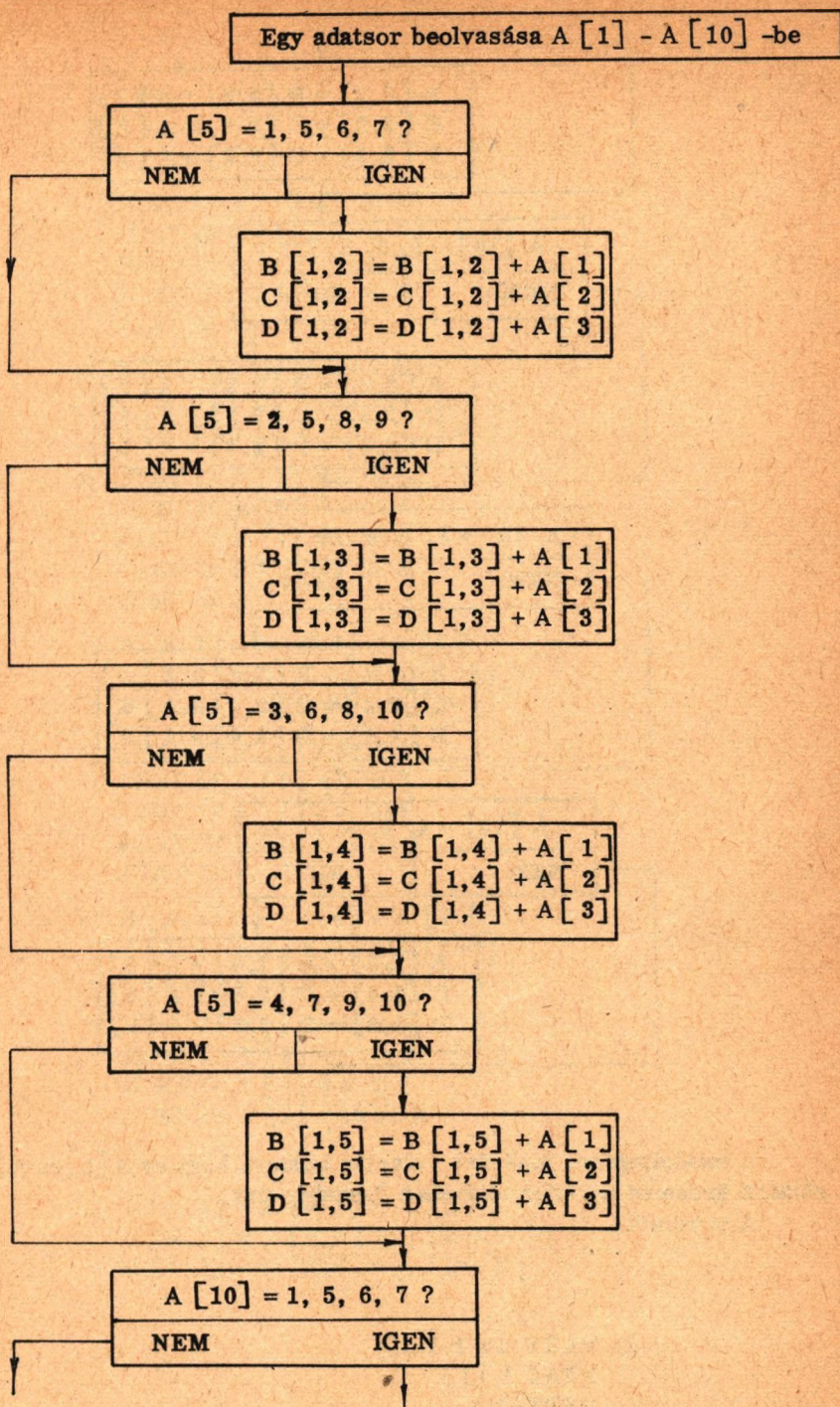
VARY T = 1:1:6
I = 0
K = 0
VARY J = 1 :1:10
JUMP IF M = L & A [J] @ 2
I = I+1
B [I] = A [J]
JUMP @ 3
2) K = K+1
C [K] = A [J]
3) REPEAT J
VARY J = 1:1:K
A [J] = C [J]
REPEAT J
VARY J = 1:1:I
A [J+K] = B [J]
REPEAT J
L = L 'a' 1
REPEAT T
VARY J = 1:1:10
PRINT A [J]
LINE
REPEAT J
STOP
START 1

```

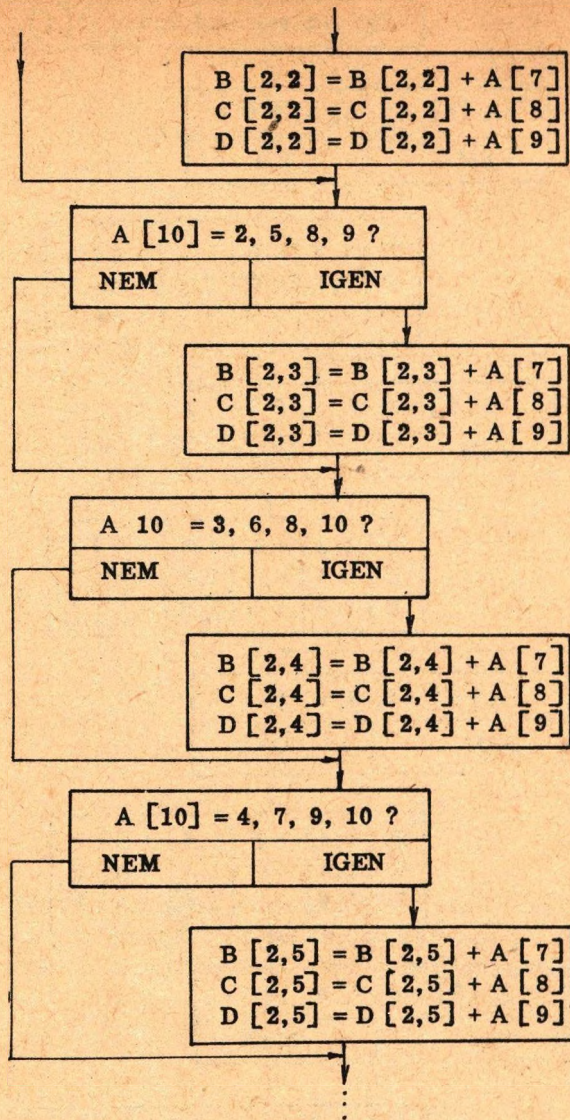
### 13. Feladat

Egy ügyviteli feladat egyik részeként 10-10 adatból álló sorokat kell szalagról beolvasni a memóriába. Minden sorban az 5. és 10. adat jelenti, hogy a beolvasott sor többi adatával milyen műveletet kell elvégezni.









A blokkdiagram átnézése alapján látható, hogy az A [5] és A [10] vizsgálatait érdemes egyetlen ciklusban összefoglalni.

A program:

```

1) VARY I=1:1:10
 READ A [I]
 REPEAT I

```



K = 0

L = -5

VARY I = 5:5:2

K = K+1

L = L+6

JUMP IF A [I] = 1 @ 2

JUMP IF A [I] = 5 @ 2

JUMP IF A [I] = 6 @ 2

JUMP IF A [I] = 7 @ 2

JUMP @ 3

2) B [K,2] = B [K,2] + A [L]  
C [K,2] = C [K,2] + A [L+1]  
D [K,2] = D [K,2] + A [L+2]

3) JUMP IF A [I] = 2 @ 4  
JUMP IF A [I] = 5 @ 4  
JUMP IF A [I] = 8 @ 4  
JUMP IF A [I] = 9 @ 4  
JUMP @ 5

4) B [K,3] = B [K,3] + A [L]  
C [K,3] = C [K,3] + A [L+1]  
D [K,3] = C [K,3] + A [L+2]

5) JUMP IF A [I] = 3 @ 6  
JUMP IF A [I] = 6 @ 6  
JUMP IF A [I] = 8 @ 6  
JUMP IF A [I] = 10 @ 6  
JUMP @ 7

6) B [K,4] = B [K,4] + A [L]  
C [K,4] = C [K,4] + A [L+1]  
D [K,4] = D [K,4] + A [L+2]

7) JUMP IF A [I] = 4 @ 8  
JUMP IF A [I] = 7 @ 8  
JUMP IF A [I] = 9 @ 8  
JUMP IF A [I] = 10 @ 8

JUMP @ 9

8) B [K,5] = B [K,5] + A [L]  
C [K,5] = C [K,5] + A [L+1]  
D [K,5] = D [K,5] + A [L+2]



9) REPEAT I

⋮

JUMP @ 1

Érdemes tovább rövidíteni a programot a

$$\begin{aligned} B[K,] &= B[K,] + A[L] \\ C[K,] &= C[K,] + A[L+1] \\ D[K,] &= D[K,] + A[L+2] \end{aligned}$$

tipusu programrészek szubrutinba foglalásával.

A program:

- 1) VARY I = 1:1:10  
READ A [I]  
REPEAT I  
K = 0  
L = -5  
VARY I = 5:5:2  
K = K+1  
L = L+6  
JUMP IF A [I] = 1 @ 2  
JUMP IF A [I] = 5 @ 2  
JUMP IF A [I] = 6 @ 2  
JUMP IF A [I] = 7 @ 2  
JUMP @ 3
- 2) T = 2  
SUBR 10
- 3) JUMP IF A [I] = 2 @ 4  
JUMP IF A [I] = 5 @ 4  
JUMP IF A [I] = 8 @ 4  
JUMP IF A [I] = 9 @ 4  
JUMP @ 5
- 4) T = 3  
SUBR 10
- 5) JUMP IF A [I] = 3 @ 6  
JUMP IF A [I] = 6 @ 6  
JUMP IF A [I] = 8 @ 6  
JUMP IF A [I] = 10 @ 6  
JUMP @ 7
- 6) T = 4  
SUBR 10



```

7) JUMP IF A [I] = 4 @ 8
 JUMP IF A [I] = 7 @ 8
 JUMP IF A [I] = 9 @ 8
 JUMP IF A [I] = 10 @ 8

 JUMP @ 9

8) T = 5
 SUBR 10

9) REPEAT I
 :
 :
 :
 JUMP @ 1

10) B [K, T] = B [K, T] + A [L]
 C [K, T] = C [K, T] + A [L+1]
 D [K, T] = D [K, T] + A [L+2]
 EXIT

```

Még egyszerűbbé tehetjük a programot, ha az összegezési feltételeket tartalmazó feltételes ugró utasításokat is szubrutinba helyezzük. A probléma megoldása az első pillanatban igen nehéznek látszik. A probléma megoldását nem az egyes JUMP IF utasítás csoportokban hivatkozott különböző referenciaszámok, és a JUMP IF-eket követő JUMP utasítások különböző referenciaszámai nehezítik. Ha ugyanis ezeket az utasításokat is a SUBR 10-be helyezzük ez a probléma könnyen áthidalható!

```

10) JUMP IF A [I] = @ 2
 JUMP IF A [I] = @ 2
 JUMP IF A [I] = @ 2
 JUMP IF A [I] = @ 2
 JUMP @ 3

2) B [K, T] = B [K, T] + A [L]
 C [K, T] = C [K, T] + A [L+1]
 D [K, T] = D [K, T] + A [L+2]

3) EXIT

```

A problémát az jelenti, hogy a szubrutin behívások alkalmával a JUMP IF A[I]= után különböző számértékek szerepelnek. Megoldható azonban a probléma úgy, hogy a feltételes ugró utasításokat JUMP IF A [I] = E [i] (i = 1, 2, 3, 4,) alakban írjuk fel és az E [i] - nek az egyes szubrutin behívások előtt adunk konkrét értéket.



Ez esetben a program:

1) VARY I = 1:1:10

READ A [I]

REPEAT I

K = 0

L = -5

VARY I = 5:5:2

K = K+1

L = L+6

T = 2

E [1] = 1

E [2] = 5

E [3] = 6

E [4] = 7

SUBR 10

T = 3

E [1] = 2

E [2] = 5

E [3] = 8

E [4] = 9

SUBR 10

T = 4

E [1] = 3

E [2] = 6

E [3] = 8

E [4] = 10

SUBR 10

T = 5

E [1] = 4

E [2] = 7

E [3] = 9

E [4] = 10

SUBR 10

REPEAT I

⋮

JUMP @ 1



- 10) CYCLE  $i = 1:1:4$   
 JUMP IF A [I] = E [1]  $\odot 2$   
 REPEAT  $i$   
 JUMP  $\odot 3$
- 2)  $B[K, T] = B[K, T] + A[L]$   
 $C[K, T] = C[K, T] + A[L+1]$   
 $D[K, T] = D[K, T] + A[L+2]$
- 3) EXIT

Az E [i] -k bevezetésével a program szerkezete lényegesen egyszerűsödött és áttekinthetőbbé vált. Gondoljuk azonban meg, hogy a ciklusba tett JUMP IF utasítás, végrehajtás szempontjából továbbra is 4 utasítást jelent, sőt az eddigi megoldásokkal szemben a ciklushatároló utasítások bevezetése még többletet is jelent a program futása során. Most bemutatunk egy olyan módszert, melynek segítségével az eddig szükséges 4 JUMP utasítást a végrehajtás szempontjából is egyetlen egy utasításra lehet csökkenteni.

Gondoljuk meg, hogy a

```
JUMP IF A [I] = 1 $\odot 2$
JUMP IF A [I] = 5 $\odot 2$
JUMP IF A [I] = 6 $\odot 2$
JUMP IF A [I] = 7 $\odot 2$
```

utasítás sorozat szavakban azt jelenti, hogy ugrani kell akkor, amikor az A [I] tartalma vagy 1, vagy 5, vagy 6, vagy 7. Ezt a feltételt átfogalmazhatjuk úgy, hogy ugrani akkor kell: ha az E rekesz tartalmát vagy 1-gyel, vagy 5-tel, vagy 6-tal, vagy 7-tel jobbra léptetve az utolsó bit értéke 1 lesz. Vagyis az E rekeszt ábrázolva:

0 0 . . . . . 0 0 0 1 1 1 0 0 0 1 0

Egy rekesz adott bitjének értéke pedig megfelelő számú léptetéssel és logikai szorzással igen egyszerűen megtudható, ha ugyanis az n-edik bitet akarjuk megvizsgálni, akkor a rekesz tartalmát (n-1)-gyel jobbra léptetve a kérdéses bitet az utolsó pozícióba tolhatjuk, és az így előállított új rekesztartalmat 1-gyel logikailag szorozva megállapíthatjuk, hogy az eredmény 1-lesz, ha a kérdéses bit 1 volt, és 0 lesz, ha kérdéses bit 0 volt.

A négy JUMP IF A [I] = E [1]  $\odot 2$  feltételt tehát az egyetlen JUMP IF M=E 'b' A [I]&1  $\odot 2$  feltételben foglalható össze, ha M=1.

A program:

- 1) VARY I = 1:1:10  
 READ AI  
 REPEAT I  
 K = 0  
 L = -5  
 M = 1



VARY I = 5:5:2

K = K+1

L = L+5

T = 2

E = 226

SUBR 10

T = 3

E = 804

SUBR 10

T = 4

E = 1352

SUBR 10

T = 5

E = 1680

SUBR 10

REPEAT I

JUMP @ 1

10) JUMP IF M = E 'b' A I &1 @ 2

JUMP @ 3

2) B [K, T] = B [K, T] + A [L]

C [K, T] = C [K, T] + A [L+1]

D [K, T] = D [K, T] + A [L+2]

3) EXIT

#### 14. Feladat

Egy asztali elektronikus számológépet gyártó cég elektronikus gépeinek lemérésére alkalmazza a következő képletet:

$$y = \left\{ \arcsin [\cosh^2 (3X)] \right\} \operatorname{tg} \left\{ \arcsin \frac{X}{\cos [\ln (x^2)]} \right\}$$

A feladat a formula beprogramozása. Ezért először a formulát olyan alakra kell hozni, mely már közvetlenül programozható!

A formula programozhatóvá tételéhez a következő azonosságokat fogjuk felhasználni:

$$1) \arcsin \alpha = \operatorname{arc} \operatorname{tg} \frac{\alpha}{\sqrt{1 - \alpha^2}}$$

$$2) \cosh \alpha = \frac{e^{\alpha} + e^{-\alpha}}{2}$$



$$3) y = a^{\operatorname{tg} b}$$

$$\ln y = \ln (a^{\operatorname{tg} b})$$

$$\ln y = \operatorname{tg} b \cdot \ln a$$

$$e^{\ln y} = e^{\operatorname{tg} b \cdot \ln a}$$

$$y = e^{\operatorname{tg} b \cdot \ln a}$$

vagyis:

$$y = a^{\operatorname{tg} b} = e^{\operatorname{tg} b \cdot \ln a}$$

$$4) \operatorname{tg} (\operatorname{arctg} \alpha) = \alpha$$

A fenti azonosságok alapján a formula:

$$y = \left\{ \operatorname{arctg} \frac{\left[ \frac{e^{3X} + e^{-3X}}{2} \right]^2}{\sqrt{1 - \left[ \frac{e^{3x} + e^{-3x}}{2} \right]^4}} \right\} \operatorname{tg} \left\{ \operatorname{arctg} \frac{\frac{x}{\cos [\ln (x^2)]}}{\sqrt{1 - \left[ \frac{x}{\cos [\ln (x^2)]} \right]^2}} \right\}$$

A 4/ azonosság figyelembevételével:

$$y = \left\{ \operatorname{arctg} \frac{\left[ \frac{e^{3x} + e^{-3x}}{2} \right]^2}{\sqrt{1 - \left[ \frac{e^{3x} + e^{-3x}}{2} \right]^4}} \right\} \frac{\frac{x}{\cos [\ln (x^2)]}}{\sqrt{1 - \left[ \frac{x}{\cos [\ln (x^2)]} \right]^2}}$$

A 3/ azonosság figyelembevételével kapjuk azt a végleges formulát, melyet már programozni lehet.

$$y = e^{\left[ \operatorname{arctg} \frac{\left[ \frac{e^{3x} + e^{-3x}}{2} \right]^2}{\sqrt{1 - \left[ \frac{e^{3x} + e^{-3x}}{2} \right]^4}} \right] \frac{\frac{x}{\cos [\ln (x^2)]}}{\sqrt{1 - \left[ \frac{x}{\cos [\ln (x^2)]} \right]^2}}}$$



A formula programja:

```
SETF LOG FRAC SQRT EXP ARCTAN COS
SETV XABCY
SETR 5
```

- 1) READ X  
A = X \* X  
JUMP IF A = 0 @ 5  
A = LOG A / 3.141592  
A = 3.14 1592 \* FRAC A  
JUMP IF A < 0 @ 2  
A = A - 1.57 0796
- 2) A = X / cos A  
B = 1 - A \* A  
JUMP UNLESS B > 0 @ 5  
A = A / SQRT B  
B = 3 \* X  
C = -B  
B = EXP B/2 + EXP C/2  
B = B \* B  
C = 1 - B \* B  
JUMP UNLESS C > 0 @ 5  
B = B / SQRT C  
B = ARCTAN B  
JUMP UNLESS B > 0 @ 5  
B = LOG B  
A = A \* B  
Y = EXP A  
PRINT Y, 6  
STOP
- 5) TITLE A FUEGGVEENY ARGUMENTUMA NEM  
MEGFELELOE'  
STOP  
START 1

### 15. Feladat

Ügyviteli feladatokban igen gyakran szerepel az adatkikeresés. Pl. tároljuk cikkszámonként egy üzem termeléséhez tartozó törzsadatokat. Egy ebből készített "szótár" pl.

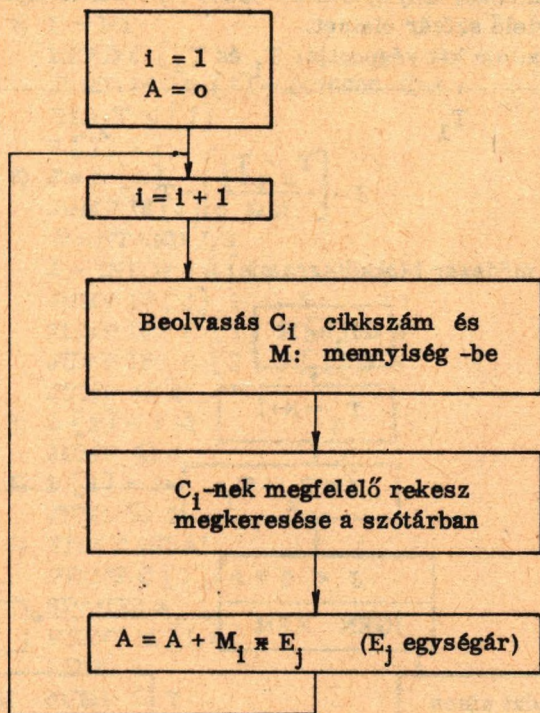
| Cikkszám | Egységár |
|----------|----------|
| 1        | 23       |
| 2        | 48       |
| 17       | 1016     |
| :        | :        |
| :        | :        |



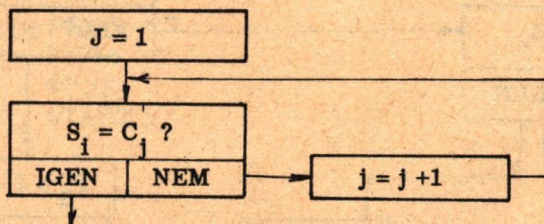
Tegyük fel, hogy ez a szótár már a memóriában van és rendelkezésünkre áll. (Az utolsó érték: 100 000) Naponta kapunk adatszalogokat a legyártott termékek cikkszámáról és mennyiségéről, és ki kell számítani, hogy mennyi a gyár bevétele.

A naponként kapott adatok rendezetlenek, de a memóriában levő szótár rendezett.

A feladat blokkdiagramja:



A szótár keresés blokkdiagramja a következő lehet:



Ezzel a blokkdiagrammal készítve el a programot, 1000 szavas szótár esetén kedvezőtlen esetben (ha az utoljára szereplő cikket keressük) 1000 összehasonlítást kell elvégezni.



Kihasználhatjuk viszont azt, hogy a szótár rendezett, és így a következő eljárást is alkalmazhatjuk: Kiválasztjuk a szótár megközelítőleg középső elemét, és megnézzük, hogy a  $C_j \leq S_k$ , illetve  $C_j > S_k$  teljesül-e.

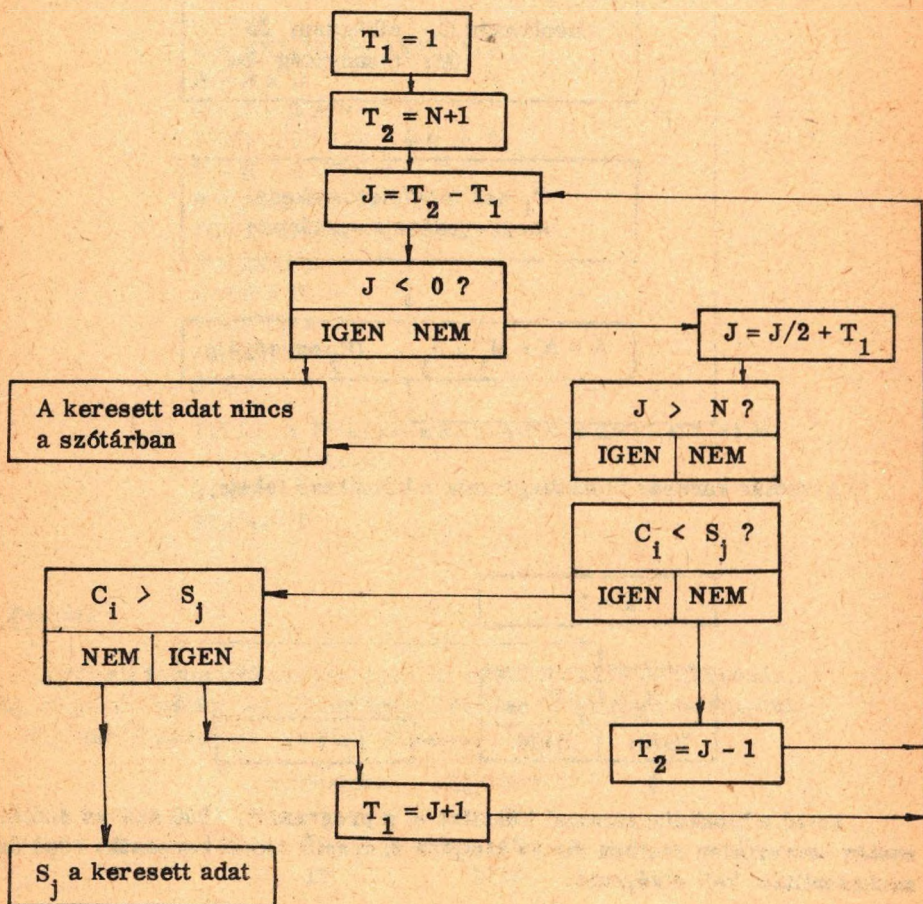
A feltételek valamelyikének teljesülésétől függően a szótár mezőjének vagy az egyik felébe, vagy a másik felébe esik a beolvasott adat.

A megtalált fél mezőre nézve tovább folytatva a felezési eljárást az előbbi módszernél (általában) lényegesen kevesebb lépéssel találjuk meg a beolvasott adatnak megfelelő szótár elemet.

Ha az intervallum két végpontja:  $T_1$  és  $T_2$

$$J = \left[ \frac{T_2 - T_1}{2} \right] + T_1$$

A rendezési módszer blokkdiagramja:





```

SETS IN J S [2]
SETV WC [100] M [100] P [100] A
SETF INT STAND
SETR 8
1) I = 0
 A = 0
 N = 1000
7) I = I+1
 READ C [I]
 JUMP IF C [I] = 1000000 @ 8
 READ M [I]
 S [1] = 1
2) J = S [2] - S [1]
 JUMP IF [J] @ @ 5
 W = STAND J/2
 J = INT W + S [1]
 JUMP IF [J] N @ 5
 JUMP IF C [I] P [J] @ 3
 JUMP IF C [I] P [J] @ 4
 JUMP @ 6
3) S [2] = J - 1
 JUMP @ 2
4) S [1] = J + 1
 JUMP @ 2
5) TITLE HIBA'
 PRINT C [I]; 4
 SPACES 3
 PRINT M [I,6]
 LINE
 JUMP @ 7
6) A = A + M [I] * E[J]
 JUMP @ 7
8) LINE
 TITLE A NAPI TERMELT EERTEEK:'
 SPACE
 PRINT A,8
 TITLE FT.'
 STOP
 START 1

```















