Hogyan programoztunk mi?
A VIDEOTON VT 1010B és az R 10 gépei

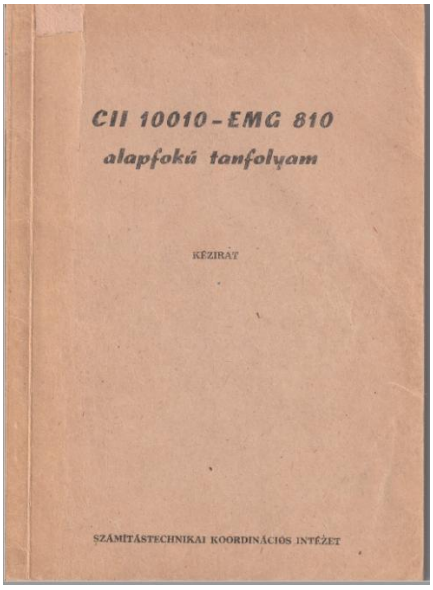
Bodnár László

Erre a kérdésre nem lehet egyszerűen válaszolni, mert egy olyan programozó, aki COBOL-ban, FORTRAN-ban, PL/1-ben vagy más magas szintű nyelven írt felhasználói programokat, valószínűleg másként programozott, mint például mi, akik gépközeli, alapszoftverfejlesztést végeztünk Assemblerben. A mi munkánk erősen gépfüggő volt, míg az előző kollégák esetében a géptől nem nagyon függtek.

A kérdésben lévő múlt idő számomra úgy 50 évvel ezelőtti időszakot, az 1970-es éveket jelentette, és bizony könnyen lehet, hogy az emlékeim megkoptak azóta.

Programozóként volt szerencsém két olyan gépen dolgozni, amelyeket nagy valószínűséggel a velem egykorúak is legfeljebb hallomásból ismerhettek, a mai kollégák meg még úgy sem: a VIDEOTON számítógép, a VT 1010 B, illetve az R 10 sorozat gépei. Ezek a gépek már tényleg muzeális darabok. Budapesten a Magyar Műszaki és Közlekedési Múzeumban, Szegeden az Informatika Történeti Kiállításon is van R 10-es kiállítva, de VT 1010B – legjobb tudomásom szerint – sehol sincsen már.

A MITRA 15-ről elég sok mindent meg lehet találni az interneten, a CII 10010-ről szinte semmit. Nekem is mindössze egy magyar kézikönyvem maradt meg a gépről:



Ezen a két, a hazai számítástechnika alapjait jelentő gépen alapszoftver fejlesztéseken dolgoztam, így csak arról tudok írni, hogy mi/én hogyan programoztunk.

Hogy hogyan programoztunk?

Az akkori programozási munka abban teljes egészében megegyezett a maival, hogy megold – leprogramoz – egy feladatot. Akkoriban szinte minden programfejlesztést támogató eszközt nélkülöznünk kellett, viszont állandóan figyelembe kellett vennünk azt a korlátot, amit maga a gép (a hardver, az utasításkészlet stb.) állított elénk. Nem volt meg az a réteg (monitor, operációs rendszer stb.), ami elfedte a gép hardverére jellemző sajátosságokat, így azok kezelése elsődleges feladatként jelentkezett. Némi túlzással: a programok/algoritmusok a gépre voltak optimalizálva.

Később, amint megjelent az operációs rendszer/monitor, már tudtunk azzal „is" foglalkozni, hogy a megírt programok futási időre, memóriahasználatra, erőforrás felhasználására stb. legyenek optimalizálva.

Viszont a szegényes eszközkészlet – lásd később – minél hatékonyabb kihasználása, véleményem szerint, bizonyos mértékben kihatott a programozási munka egészére; amit lehetett, azt megpróbáltuk – legalábbis én személyesen – gép nélkül elvégezni, illetve maximálisan kihasználni azokat a speciális lehetőségeket, amit a gépünk biztosított.

1972-ben lettem a VIFI (VIDEOTON Fejlesztési Intézet) Szoftver Fejlesztési Főosztályának munkatársa, amit dr. Baráth Csaba vezetett. Az Intézet igazgatója Kázsmér János, helyettese pedig dr. Gantner János volt.

Az első gép, amit programoztam, az a VT 1010 B volt, utána pedig a VIDEOTON R 10 és az azt követő modellek következtek, majd pedig a dr. Náray Zsolt által vezetett SZKI (Számítástechnikai Koordinációs Intézet) Hardware Laboratóriumába kerültem, amit akkor Hubert Béla és Mannhardt Endre vezetett. Itt az SZKI Proper gépeit, valamint az ott kifejlesztett Teleterm terminált programoztam.

Az SZKI után már nem foglalkoztam programozással, de mind a mai napig írogatok programokat, mostanában többnyire Python és R nyelven.

A VIDEOTON VT 1010 B számítógép

A VT 1010B vagy EMG 810 – lánykori nevén: CII 10010 (http://www.feb-patrimoine.com/projet/10010/cii_10010.htm) – akkoriban szinte egyedülálló volt a KGST piacon, talán még a lengyel ODRA sorozat egyes tagjai tartoztak a VT 1010 B kategóriájába.

A VIFI-ben akkoriban a VT 1010 B gépen folytak a fejlesztések, bár már előrehaladott tárgyalások zajlottak a francia féllel egy korszerűbb gép licencének megvételéről (MITRA 15), de a fejlesztési munkák még a VT 1010 B-n történtek.

Amikor én a VIFI-be kerültem, két gépteremben kettő 1010 B üzemelt: a nagy gépteremben lévő gép teljes memóriakiépítéssel, 800 kbyte-os SAGEM fix fejes diszkkel, egy Pertec PEN 5-ös mágnesszalagos egységgel, TALLY 132 karakteres dobnyomtatóval, ASR 33 írógéppel, kártyaolvasóval, lyukszalagolvasóval és -lyukasztóval volt kiépítve. Tartozott még hozzá egy szinkron (CTS) és aszinkron csatoló (CMLA) is. Lehet, hogy volt ott egy NML 67-es egység is, de ebben már nem vagyok biztos. Később a konfiguráció kiegészült a VT 340-es display-jel.

A kis gépteremben egy hasonló konfiguráció működött, de ott nem volt mágnesszalag, és talán sornyomtató sem.

A CII 10010/VT 1010 B hardverről röviden

A CII 10010 szekrényes alapkiépítésben az ASR 33-as konzollal

A gépet a francia gyártó real-time feladatok – pl. atomerőmű-vezérlés – elvégzésére fejlesztette ki, szilícium integrált áramkörökből (ún. emitter csatolt logika – ECL) felépített, egycímes, lapszervezésű gép, 4 és 64 Kbyte között 4 Kbyte-onként bővíthető ferritmagos memória (16 bites szóhossz, 8 bites címregiszter) 1 µsec-os memória ciklusidő mellett, 4 megszakítási osztállyal.

Az egycímes címzés azt jelentette, hogy a memória referenciális műveleteknél csak a forrás- vagy célcímet kellett megadni, a másik „cím” pedig implicit módon mindig valamelyik akkumlátort jelentette.

Érdekesség, hogy a 16 bites szó két 8 byte-ból tevődött össze, ahol mindkét byte-nak külön volt 1-1 paritásbitje, a paritásellenőrzési funkció a pultról be- vagy kikapcsolható volt.

A lapcímzés volt az az architektúra, amivel azóta sem találkoztam. Egy lap alatt egy olyan 256 byte-os memóriaegységet kell érteni, amely direkt címezhető volt. Egy memóriacím lényegében két részből állt: volt egy lapcím, amely meghatározta, hogy pontosan melyik 256 byte-os területről van szó, és egy lapon belüli cím, amely a kiválasztott 256 byte-os területen belüli címet jelölte ki. A lapok – természetesen – nem fedték át egymás.

A központi egységben (UC) a műveletek végzésére két 8 bites regiszter (A, B) vagy akkumulátor és egy 1 bites átviteli regiszter (R) állt rendelkezésre. Valójában volt még a 8 bites E regiszter, amely a memóriába írandó, valamint onnan kiolvasott értéket tárolta, illetve ennek a „bővítése" az INOP utasításregiszter (I: 2 bit, címzési mód, N: 2 bit, kiegészítő információk és OP: 4 bit, műveleti kód), valamint az S címregiszter.

Az aritmetikai egység négy műveletet tudott elvégezni:

- összeadás,
- kivonás (csak osztás esetén!),
- logikai összeadás,
- logikai szorzás.

A „közönséges" kivonást már programozni kellett: a kivonandó komplemensét hozzá kellet adni a kisebbítendőhöz.

Az aritmetikai egység ezeket a műveleteket két operanduson végezte el, ahol az egyik operandus az E regiszter tartalma, a másik operandus pedig vagy

- az A regiszter;
- a B regiszter;
- az N regiszter tartalma, magasabb helyiértékeken 6 darab nullával kiegészítve;
- a csatolóegységekből érkező 8 bites információ;
- vezérlőpult kulcsain beállított információ.

Az aritmetikai utasításokon kívül használhatók voltak még memória referenciás (írás, olvasás), logikai (ÉS, VAGY), byte-os szorzási és osztási, regiszterkezelő, léptető, feltétlen és feltételes ugrási, be- és kiviteli, valamint megállási utasítások. Az aritmetikai egység egy utasítást átlagosan 6 μsec alatt hajtott végre.

Három csatornán történhetett az információcsere a számítógép és a perifériák között, amelyet az átviteli egység (UE) vezérelt:

- Programozott csatorna – a program szervezi az adatcserét: ASR 33 írógép, lyukszalaglyukasztó lyukszalagolvasóval, gyors lyukszalagíró és -olvasó, real-time perifériák.
- Standard vagy multiplex csatorna – a program előkészíti az információcserét, de annak szervezése a programtól függetlenül történik: 800 Kbyte kapacitású fix fejes diszk, mechanikus írásvédelemmel, 9 csatornás mágnesszalag, sornyomtató, kártyalyukasztó és/vagy -olvasó, szinkron és aszinkron csatoló.
- Közvetlen memóriahozzáférési csatorna – az adatcsere a központi vezérlőegység igénybe vétele nélkül történik: mágneslemezes tároló, másik számítógép központi egysége.

A gépnek volt asztali, illetve szekrényes kivitele, az asztali gép kb. 90 kg-ot nyomott, a szekrényes kivitel pedig 200-600 kg között mozgott – kiépítéstől függően.

Amint azt már említettem, a memória 256 byte-os lapokra volt felosztva, amelyek közül volt egy kitüntetett lap: az ún. nullás lap, ezt ugyanis minden másik lapról közvetlenül el lehetett érni.

Ennek a mindegyik lapról elérhető nullás lapnak néhány helye vagy címe kiemelt jelentésű volt (zárójelben az Assembler mnemonik hivatkozás, ahol a **$** karakter jelezte a nullás lapon a direkt címzést, a pont pedig a hexadecimális számot jelölte, ahogyan azt az ASTROL Assemblerben használatos volt):

- a .00-ás címén volt az utasításszámláló ($CI);
- a .02-es címen az ún. interpreter vagy értelmező – erről majd később ($IN);
- a .08-0E címeken a megszakítási szintek utasításszámlálói ($I0-$I3);
- a .10-3F címeken az ún. pszeudó utasításszámlálók és akkumulátorok helyezkedtek el – ezekről is majd a későbbiekben lesz szó ($C1-$C4, $V0-$V3, $A1-$A10);
- a .40-4E címeken az A, B, R regiszterek mentésére szolgáló területnek voltak fenntartva, a különböző megszakítások kezeléséhez ($M0-$M3);
- az .50-56-os címeken a hexadecimális 0001, 0000, 8000, FFFF konstansok és a túlcsordulás jelző foglaltak helyet ($UN, $ZR, $S1, $MU, $OV);
- az .58-7E közötti terület általános munkamezőként volt használható – tárolás, vagy nullás lapon keresztüli indirekt címzés ($T1-$T20);
- a .80-FF címek pedig a megszakítások munkaterületei ($R0-$R3).

Az utasítások egy mezeje, egészen pontosan az INOP regiszter I bitjei – 2 biten – határozta meg, hogy a 8 bites címet hogyan kell értelmezni:

- nullás lapon direkt cím, I=00 – az utasítás címrésze a nullás lap egy adott címre mutat;
- nullás lapon keresztül indirekt cím, I=10 – a címrészben adott 0-s lapon lévő cím és a következő tartalmazza a tényleges címet;
- lapon belüli direkt cím, I=11 – azon az oldalon lévő direkt cím, ahonnan az utasítás kiolvasásra került;
- AB regiszteren keresztül indirekt cím, I=01 – az utasítás címrésze nincs használva, az AB akkumulátor tartalmazza a tényleges címet.

A fenti címzési módok mind memória referenciális, mind pedig ugrások esetén is érvényesek voltak.

Ez volt a hardverkörnyezet, amelyben a gépet programozni kellett.

A 101B programozása Assembler nyelven történt, az ASTROL nyelvű Assembler segítségével.

A francia gépben még az utasítás mnemonikok is francia szavak rövidítései voltak, tehát a program nem volt mindenki számára érthető.

Maga az ASTROL nyelv sem volt túlcizellálva, sok esetben hexadecimális kódok jelölték az utasítás csoporton belüli funkciót.

Például:

A feltételes ugrás mnemonikja az **SCN** volt, és az operandusa határozta meg, hogy milyen feltételről is van szó. Az SCN .04 (a pont jelezte, hogy hexadecimális értékről van szó) az A regiszter 0 értékénél történő ugrást jelentett. Az ugrás helyét a következő utasításként szereplő feltétlen ugrás (BRI) jelölte ki. Ha a feltétel nem teljesült, akkor a program a feltétlen ugrást követő utasításnál folytatódott, tehát egy feltételes ugrás az lényegében két utasítással volt megvalósítható:

- APM var * szóolvasás az A regiszterbe a var memória rekeszből;
- SCN .04 * ugrás, ha A = 0;
- BRI címke * ugrás, ha a feltétel teljesült; utasítások, ha a feltétel nem teljesült.

Mivel a gép lapszervezésű volt, ezért lehetőség szerint egy programrészt egy lapon – 1 lap: 256 byte – belül kellett elhelyezni, mert ha ez nem sikerült, akkor még külön foglalkozni kellett azzal, hogy vagy a változót, vagy a programfolytatást, amely kívül esett az aktuális

lapon, hogyan érhetjük el. Ezért sokszor az utasítások operandusát, de néha magát az utasításkódot használtuk fel konstansnak, hogy ne kellejen erre külön tárterületet fenntartani, és nyerjünk 1-2 byte-ot.

El lehet képzelni, mennyire voltak ezek a programok könnyen javíthatók és újrafelhasználhatók, valamint jól olvashatók. Néha még a program írójának is komoly erőfeszítésbe telt, hogy megfejtse, mit is akart valamikor egy programrészben megvalósítani.

Kiemelt szerepet játszott az előzőekben már említett ún. nullás lap, mert ez minden lapról automatikusan elérhető volt, így – többek között – közösen használt tárolóterületnek – akár értékek, akár címek és indirekt címek – tárolására volt alkalmas. A teljes memória csak indirekt címzések használatával volt hozzáférhető.

A nullás lap 256 byte-ja sem volt teljes egészben szabadon használható, egy része a gép működtetéséhez volt fenntartva, jó közelítéssel a lap fele – 120-130 byte – volt az, amit szabadon lehetett használni. Abban az esetben, ha az ún. gyári mikroprogramokat nem használtuk, akkor a számukra fenntartott .10-.3F munkaterület is szabadon használható volt.

A gépnek – ahogyan azt már az előzőekben írtam – négy címzési módja volt:

- aktuális (használt) lapon belül direkt – a cím egy lapon belüli címet határoz meg;
- nullás lapon belül direkt – a cím a nullás lapon belüli címet határoz meg;
- nullás lapon keresztül indirekt – a megadott cím egy másik címet (másik lap, és azon belüli cím) határoz meg;
- akkumulátorán keresztül indirekt.

Az akkumulátoron keresztüli indirekt címzés esetében az akkumulátor magasabb helyiértékein a lapcímet, az alacsonyabb helyiértékeken a lapon belüli byte-címet lehetett megadni, és erre a címre lehetett ugrani, illetve erről a címről lehetett olvasni vagy írni egy byte-ot vagy egy egész szót.

A VT 1010B szoftverről

A CII 10010-nek, illetve a VT 1010B-nek nem volt semmiféle operációs rendszere vagy ehhez hasonló szoftver támogatása, a gép „alapszoftverét” – már ha ezt annak lehet nevezni – a következő elemek alkották:

- AUTOCHARGEUR,
- CHARGEUR,
- ASSEMBLEUR,
- SORTIE BINARE,

- mikroprogram könyvtár.

Az Assembler-utasítás mnemonikok, a segédprogramok elnevezése, a kezelő pult feliratai stb. mind-mind francia nyelvű volt, ezért némi magyarázatot adok az alapszoftverhez: AUTOCHARGEUR volt az a program, amit ma leginkább boot strap-nek neveznék – a nullás lapon futott, és alkalmas volt arra, hogy egy ún. öntölthető formátumú program betöltését elvégezze. Az „öntölthető" praktikusan nem volt más, mint egy bináris memóriakép, kiegészítve még egy indítási címmel. Mindezek az információk egy 8 csatornás lyukszalagon jelentek meg, 4 bites bináris kódban.

Ami érdekes: ezt az ún. öntölthető formátumot nem az ASSEMBLEUR állította elő.

A CHARGEUR – töltő – annyival tudott többet az AUTOCHARGEUR-nál, hogy képes volt tárgyprogramot betölteni. A tárgyprogram abban különbözött az öntölthető programtól, hogy voltak benne olyan információk, amelyek a betöltés helyére, a címhivatkozások feloldására, az indítási cím megadására stb. szolgáltak. A tárgyprogram az ún. semi-bináris kódban került lyukszalagra.

Ez a kód a 8 bites információt két lyukszalag oszlopon rögzítette úgy, hogy az információ magasabb helyiértékei a lyukszalag első oszlopának első négy csatornáján jelentek meg, az alacsonyabb helyiértékek pedig a második oszlop első négy csatornáján kerültek rögzítésre. Az ötödik csatorna tartalma közömbös, a hatodik és hetedik csatornán pedig mindig volt lyukasztás, a nyolcadik csatorna pedig mindig üres volt.

Az Assembler a tárgyprogramban a fent említett speciális információkat különféle blokkokba lyukasztotta, ahol a blokkfej határozta meg a blokktípust 2 byte-on, ezt követték a blokk adatai, majd a blokk összes tartalmának összege mod 256, mint ellenőrző érték. Mindez semi-bináris kódban, tehát elég terjedelmes tudott lenni egy tárgyprogram.

Ez a formátum volt az Assembler kimenő formátuma, amely egymenetes lévén, kénytelen volt ilyen megoldásokat alkalmazni. A CHARGEUR csaknem 2 lap terjedelmű volt, és természetesen AUTOCHARGEUR segítségével volt csak betölthető.

A SORTIE BINARE – bináris kimenet – egy megadott címtől címig lyukszalagra lyukasztotta a kijelölt memóriatartalmat, valamint még a megadott indítási címet – lényegében egy bináris állományt állított elő, azaz egy öntölthető formájú lyukszalagot készített.

ASSEMBLEUR egy ASTROL nyelvű Assembler, amely az ASCII kódú forrásprogramból tárgyprogramot készített egy menetben, illetve elkészítette a program listáját – az utasítások és az adatok hexadecimális kódjait –, valamint a hibajelzéseket.

Az Assembler az utasítás mnemonikokon kívül képes volt kezelni a fordítás vezérlésére szolgáló speciális mnemonikokokat, amiket direktíváknak neveztek. Ezek a direktívák szolgáltak

- a program egyes részei elhelyezésének;
- a program indítási címének;
- a decimális vagy hexadecimális, illetve ASCII alfanumerikus konstansok;
- közös szimbólumok megadására, memóriaterületek lefoglalására, szimbolikus cím tényleges címmé alakítására stb. A BELL, RC, LF (RC, LF – kocsi vissza, soremelés) input karaktersorozat hatására az Assembler megállt és várt arra, hogy egy következő lyukszalag befűzésre kerüljön, és folytatódjon a fordítás.

A tárgyprogramot lyukszalagra lyukasztotta. A program képes volt több forrást egymás után beolvasni, így lehetőség volt arra, hogy a forrásállomány több darabból álljon össze, illetve a mikroprogram[1] könyvtár megfelelő elemei felhasználhatók legyenek. A tárgyprogram csak az CHARGEUR használatával volt memóriába tölthető.

A mai terminológia alapján kicsit furcsának tűnhet, hogy a csaknem mázsás súlyú gépet mikrogépnek tekintették, illetve a mikroprogram könyvárnak valójában a mikroprogramokhoz semmi köze nem volt, hanem egy forrásnyelvű eljáráscsomag kapta ezt a nevet.

Amint látható, a géphez semmiféle monitor, operációs rendszer nem volt: mindent ezzel a négy segédprogrammal, és persze programozói munkával kellett megoldani.

A CII10010 francia leírása szerint létezett egy FORTRAN fordító a géphez, de a legjobb tudomásom szerint nálunk az intézetben ilyen nem volt. Volt viszont egy BASIC programja, ami oktatási célokra jól használható lett volna, ha nem egykonzolos módon működött volna.

Eddig az összefoglaló, most nézzük, hogyan hatottak ezek az adottságok a programozásra!

Egy program elkészítése a következő módon történt:

- Egy lyukszalag lyukasztására képes írógépen – ez nálunk egy ASR33-as írógép volt – le kellett rögzíteni a szimbolikus vagy forráskódot.
- A forráskódot le kellett fordítani.
- A fordításból kapott lista alapján a hibákat ki kellett javítani, majd a forrást újrafordítani.

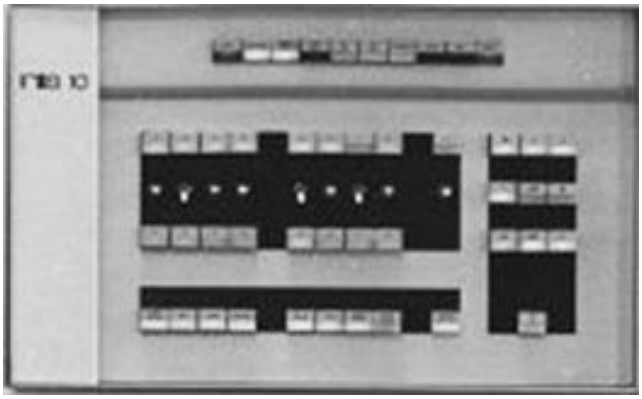
[1] A mikroprogram könyvtárról már szó esett az előzőekben. A mikroprogram könyvtár használatáról részletek az interpretatív programozásról szóló részben találhatók.

- Hibátlan fordítás után lehetett nekifogni a program tesztelésének, ami többnyire ismét forráskód-javítást, -módosítást jelentett.
- Ha a tesztelés és hibajavítás cikluson végigment a program, akkor az utolsó lépés az volt, hogy a memóriába betöltött programról a SORTIE BINARE program segítségével elkészülhetett az önbetölthető formátumú program – lyukszalag –, amit már az AUTOCHARGEUR képes volt egy lépésben betölteni.

A forráskód lefordítása sem volt egy egyszerű, mert ehhez több lépés végrehajtása volt szükséges:

- be kellett lyukszalagról tölteni az AUTOCHARGEUR programot;
- az AUTOCHARGEUR segítségével be kellett tölteni – szintén lyukszalagról – az ASSEMBLEUR-t;
- a lyukszalagon lévő forrásprogramot az ASSEMBLEUR lefordította, elkészítette a fordítási listát és a program tárgyprogramját.

Operációs rendszer nélkül minden műveletet – pl. betöltés, programindítás – a kezelőpultról kellett vezérelni.



A CII 10010 kezelő pultja

Az AUTOCHARGEUR betöltése pl. a következő lépésekkel volt végrehajtható:

- RAZ UC gomb megnyomása – a központi egység törlése,
- RAZ EU gomb megnyomása – az átviteli egység törlése,
- CHARGEMENT gomb benyomása – töltésfunkció indítása,
- a kezelőpult kulcsai segítségével az A regiszterbe .40-et beírni,
- MARCHE gomb benyomása – beolvasás indítása, majd a beolvasás után,
- a MARCHE gomb kiengedése,
- CHARGEMENT gomb kiengedése – töltésfunkció kész,

- RAZ UC gomb megnyomása – a központi egység törlése,
- RAZ EU gomb megnyomása – az átviteli egység törlése.

A fenti folyamatot leíró „verset” minden programozó betéve tudta:

raz uc, raz ue
sarzs be
40 az A-ba
mars be
mars ki
sarzs ki
raz uc, raz ue.

A programozási folyamat – a tesztelésig – batch jellegű felhasználást jelentett: az Astrol kódlapokon megírt programot lyukasztásra leadtuk a géptermi operátoroknak, akik a forrásszalagot és a fordítási listát visszaadták, és kezdődhetett a szintaktikai hibák javítása.

Mivel semmiféle szövegszerkesztő program nem állt rendelkezésre, a forrásprogram javításakor két lehetőség közül lehetett választani:

- újra lyukasztani a forráskódot – ez azzal a veszéllyel járt, hogy lyukasztási hiba miatt esetleg olyan hiba is megjelent, ami eddig nem volt, ezért inkább azt az eljárást követtük – pontosabban: követték az adatrögzítő operátorok, hogy a hibás helyig átmásolták a forrásprogramot, beírták/lyukasztották a javítást, átlépték a hibás részt, és folytatták a másolást;
- vagy kézi lyukasztó és vágó eszközzel – a filmvágáshoz hasonló módon – javítottuk a forráskódot.

A lapszervezésből következett, hogy ha egy programrész (pl. ugrási cím) vagy adat kívül esett a laphatáron, akkor azt direkt módon nem lehetett elérni, ezért a következő lehetőségek valamelyikét kell választani:

- az adott lapon a program szempontjából „hasztalan” adatként egy területet fenn kellett tartani, mint címhivatkozást;
- a nullás lap munkaterületén egy helyet fenn kellett tartani, mint címhivatkozás.

Aztán, hogy a tárolt tényleges címet a lehetséges módok közül hogyan (akkumulátoron vagy indirekt módon) éri majd el a program, az ismételten döntést igényelt.

Viszont: a nullás lap korlátozott kapacitású munkaterületéből, vagy a lapon hasznos 256 byte-ból 2 byte-ot „hasztalanul” használtunk el, azaz ennyivel csökkent az adott lap hasznosítható program területe.

A fentiek miatt a programot úgy kellett megtervezni, hogy a laphatár túllépését elkerüljük, illetve a laphatár elérésekor gondoskodni kellett arról, hogy folytatódni tudjon a program futása (valamilyen indirekt ugrással). Ezért mindent elkövettünk, hogy a lehető legrövidebb, legtömörebb kódot írjuk meg, így gyakran előfordult az, hogy utasításkódot pl. konstansnak használtunk. Ennek viszont az volt a hátulütője, hogy néha már a kód írója sem tudta módosítani a saját kódját, nemhogy egy kívülálló. És a kommentek használata már akkor sem volt a legfontosabb.

Minden abba az irányba mutatott, hogy

- a lehető legrövidebb programot írjuk meg a forrásprogram lyukasztási hibáinak az elkerülésére és a memória minél jobb kihasználására;
- nagyon jól, többször is gondoljuk át a program logikáját, futását, száraz teszteljünk;
- igyekezzünk mindent a lehető legegyszerűbben, leghatékonyabban megcsinálni.

Persze volt még egy „lehetőség” a program gyors módosítására: az operátori pult kulcsairól (8+1 kétállású kapcsoló) be lehetett vinni adatokat – utasításokat – a memóriába. Ezzel a megoldással – szerencsés esetben – legalább addig el tudtunk jutni, hogy egy-két lépéssel tovább tudtuk futtatni a programot, és nem kellett a forráshoz visszatérni.

Ezzel a ma igen szegényesnek tűnő eszköztárral is sikerült olyan fejlesztéseket végrehajtani, mint pl. a VT 1010B-t az akkoriban a legkorszerűbbnek számító, még nem IBM klón szovjet MINSZK 32-vel összekötni, amivel a MINSZK 32-nek front-end terminálvezérlője lett.

Nem pont a hogyan programoztunk témakörébe tartozik, hanem inkább az akkori „szakértők” felkészültségét mutatja a történet.

A frissen kifejlesztett VT 340-es displayt aszinkron vonalon sikerült hozzákapcsolni a VT 1010-hez. A hozzánk érkezett VT 340-es magyar-cirill karakterkészletű volt, így a szovjet piacon nagy sikerre számíthatott, de használhatóságáról a potenciális vevőket valahogyan meg kellett győzni.

Egy nap jelzést kaptunk arról, hogy másnap egy szovjet küldöttség érkezik, akiknek feltétlenül be kell mutatni a display-t. Sok időnk nem volt, ezért azt találtuk ki, hogy írunk egy nyúlfarknyi programot, ami azt tudta, hogy egy bizonyos karakterig tartó karaktersorozatot eltárol a memóriába, és minden érkező ETX karakterre visszaírja a displayre a rögzített

szöveget. Oroszul tudó kollégák jól kitalálták, milyen szöveget visznek be, és hogy mutassák a VT 340 képességeit, a visszalépés, illetve a sortörlés funkciókat, hibás szövegeket írtak be, amelyeket természetesen ki is javítottak. A végső nagy durranás a rakéta volt: tele karakterekből ki lehetett „rajzolni" – karakteres volt a display – egy rakétát, aminek be lehetett gyújtani a hajtóművét, karakterek villogtatásával és sorok beszúrásával a rakétát ki lehetett lőni. Ma, így utólag azt mondanám, hogy nem volt egy nagy durranás, de akkoriban tényleg nagyon látványos volt, mivel egy nem papíros terminál nem volt mindennapos látvány.

Megérkezett a küldöttség, a bemutató rendben lezajlott, de feltűnt, hogy az orosz kereskedők/szakértők hirtelen nagyon felélénkülve beszéltek valamit egymással, a mi kollégáink meg alig bírták visszatartani a nevetésüket. Mi – több éves orosz tanulmányaink ellenére – sokat nem értettünk az elhangzottakból, de kíváncsiak voltunk: mégis mi történt?

Megkaptuk a választ: amikor bemutatták, hogy hogyan javítható a szöveg (visszalépés, sortörlés stb.), a küldöttség egyik tagja azonnal kijelentette, hogy erre a berendezésre feltétlen szükségük van, hiszen képes arra, hogy saját maga kijavítsa a hibákat.

A történet utóéletéhez még annyit, hogy kb. egy hét múlva utazhattunk Moszkvába gépestől, VT 340-estől, hogy ott is bemutathassunk a display hatalmas tudását. Télen…

Volt még egy elég különleges programozási lehetősége is gépnek – ez volt az ún. interpretatív programozás, ami szintén a gép egyedi sajátossága volt.

Magát az elvet többször is használtam, de soha sem az eredeti, előre megírt mikroprogramokkal, hanem a saját magam által, az adott feladathoz sokkal jobban illeszkedő eljárások elkészítésével.

Egy interpretatív programot úgy kell elképzelni, mintha a program – assembler nyelvű programról van szó – folytonos, egymás utáni „blokkokból" állna, ahol egy „blokk" egy eljárás-hívásból és az eljárás be- és kimenő paramétereiből áll. Maga az eljárás-hívás sem egy hívási utasításként jelenik meg – az assemblernek nem is volt eljárás-hívási utasítása –, hanem az eljárás belépési címeként. Az eljárás paraméterei pedig eljárásától függően címek vagy értékek lehettek. Valójában tehát a programot – leszámítva az interpretatív szekvenciára történő áttérést – adatok sorozata alkotta. A tényleges eljárások pedig ettől az „adathalmaztól" függetlenül másutt helyezkedtek el a memóriában.

Az egyes eljárások – amelyek természetesen Assembler nyelven voltak megírva – maguk is hívhattak ilyen előre megírt eljárást, 4 szintű beágyazást engedett meg a rendszer, de bármikor vissza lehetett térni az Assembler nyelvhez, a megfelelő konvenciók betartásával.

Ezeket az eljárásokat – a VT 1010B terminológia szerint mikroprogramokat – a programozók rendelkezésére bocsájtotta a szállító, a megfelelő – természetesen francia nyelvű – dokumentációval, amely elengedhetetlenül szükséges volt az egyes eljárások használatához, mert ott voltak leírva a paraméter átvételi, átadási követelményei, a kötelezően hívandó más eljárások stb.

A mikroprogram-könyvtár lényegében egy forrásnyelvű több, ún. szekcióra osztott állomány volt, amelyben a belső műveleteket (ugrások, inkrementálások, logikai műveletek, mozgatások), az egész- és fixpontos (szimpla és dupla szóhosszúságú) műveleteket, a formátum- konverziókat, a matematikai függvényeket, a programozott csatorna ki- és beviteli műveleteit valósítottak meg. A szekciók végét a BELL (.07), RC (.0D), LF (.0A) (kocsi vissza, soremelés) karaktersorozat jelezte, ami az Assembler számára azt jelezte, hogy új forrás beolvasásával folytatnia kell a fordítást.

Belső művelet alatt itt a mikroprogram-könyvtár saját belső műveleteit – pl. adatmozgatás, ugrás stb. – kell érteni. Ugyanis, ha nem akarunk állandóan az interpretatív és Assembly között váltogatni, akkor pl. az ugrást – feltételes vagy feltétlen – is, mint eljárást kell megvalósítani. Nem is beszélve arról, hogy az át- és visszatérések kódjai csak feleslegesen foglalnának el értékes memóriát.

Ez a programozási stílus/módszer annyira hozzátartozott a géphez, hogy bizonyos részei hardveresen beépítettek voltak: ez volt az ún. interpreter, ami a nullás lap 2-es címén ($IN) 6 byte-ot foglalt el. Ez a 6 byte tulajdonképpen 3 utasítás gépi kódját tartalmazta. Amennyiben az interpretatív programozás részletei érdekeltek valakit, akkor a leírás végén további részletek is megtalálhatók voltak.

Az interpretálás/értelmezés lényegében azt jelentette, hogy a programot – ami nem gépi utasításokból épült fel, hanem egymást követő adatok halmazából – programként értelmezni tudja, tehát megkülönböztette, hogy az adatok közül mi az eljárás, amelyet végre is hajt, és mi(k) az eljáráshoz tartozó paraméter(ek).

Dióhéjban ennyi volt az, amit a hardver sajátosságai miatt figyelembe kellett venni a programozásnál.

A VIDEOTON R 10/MITRA 15

A R 10 alapgépe a szintén francia fejlesztésű MITRA 15 volt. A gépet a francia cég, a CII/SEMS, általános használatra alkalmas kisgépnek tervezte, az asztali, klimatizálást nem igénylő kiviteltől a több rack-es kivitelig kiépíthető volt.

A VIDEOTON R10 sorozat hardverről röviden

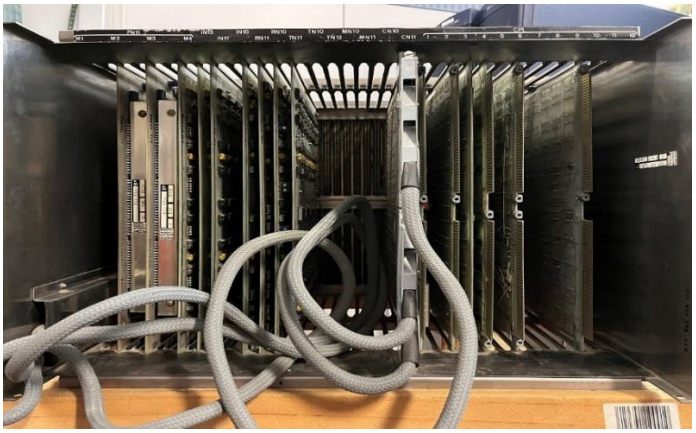
Az 1970-es évek elején még a több rack-es kivitel is kisgépnek számított.

A gép már TTL integrált áramkörökkel működött, egycímes, 4 és 64 Kbyte között 4 Kbyte-onként bővíthető ferritmagos memóriával (18 bites, 16 bites szavak, 1 paritás és 1 memóriavédelmi bit) 800 nanosec-os memória ciklusidő mellett, 32 megszakítási osztállyal. A memória referenciális utasítások végrehajtási ideje hozzávetőlegesen 2 µsec körül volt, a szorzás/osztás pedig 8 µsec-ot igényelt.

A nyugaton gyártott (Texas Inc, Fairchild stb.) TTL integrált áramkörök egy részét később szovjet gyártású áramkörök váltották fel/ki. A szervizes gyakorlatban a hibakeresés többnyire azzal kezdődött: van a kártyán szovjet IC? Ha igen, akkor cseréljük. És ez néha elegendőnek is bizonyult.

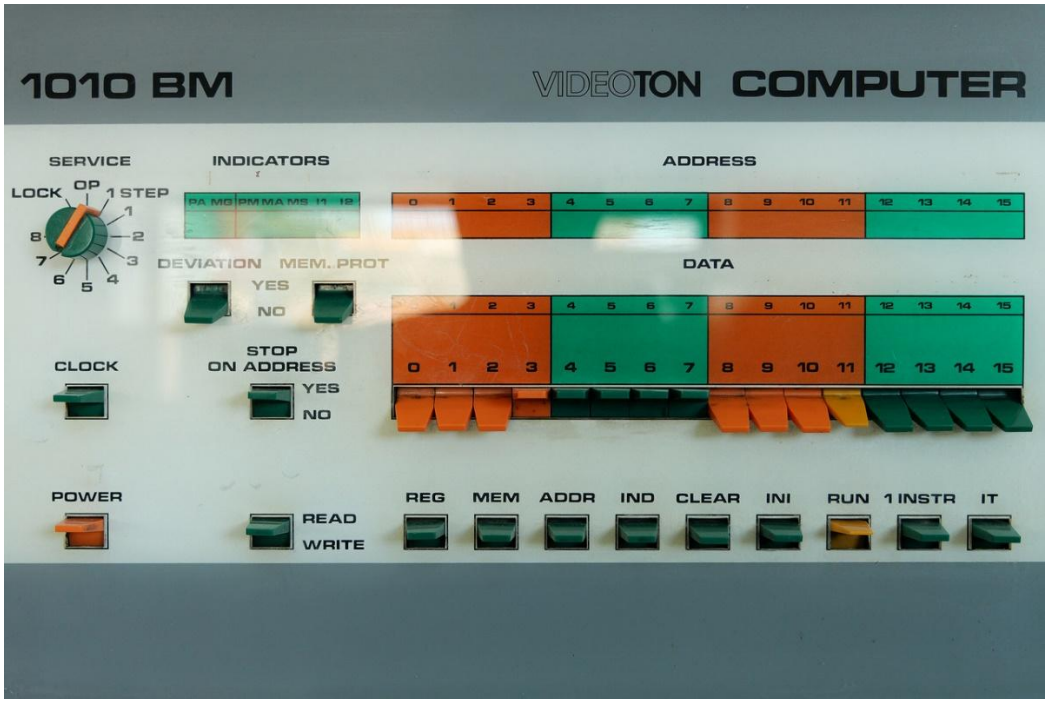
Itt talán érdemes megjegyezni, hogy bár az R10 sorozat tagjai a szocialista országok ún. egyesített számítógéprendszer (ESzR – EC) legkisebb elemei voltak, valójában azonban „rendszeridegen" elemek voltak. Az R betű az orosz „ряд" – **r**jad, azaz sor, sorozat – szóra utal. Az ESzR más gépei – R20-R70 – mind IBM 360 klónok voltak.

A perifériacsatolók és a gép más egységei egy egységes ún. minibus-hoz csatlakoztak. Ez a busz a vezérlő pult mögötti nagyméretű áramköri lapon lett kialakítva, ahová az egyes egységek kártyái bedugaszolhatók voltak, ahogyan azt a következő ábra mutatja.



A kép bal oldalán a memória blokkok láthatóak.

A MITRA 15 előlapja



Az R 10 előlapja

A központi egységben a műveletek végzésére 3 16 bites regiszter (A, E, X) állt rendelkezésre, de kiépítésre került egy bővíthető gyorsregiszter tár is. Ez a gyorselérésű tár alapkiépítésben 2x8 16 bites MSI bipoláris regisztert tartalmazott, 60 nsec hozzáférési idővel. Az első regiszterblokk tartalmazta még a G, L, P regisztereket is. (G volt az ún. globális bázis regiszter, L a lokális bázis regiszter, P pedig az utasítás számláló.) Ilyen tárból 4 volt beépíthető egy konfigurációba. A gyorsregiszter tár csak az ún. privilégizált üzemmódban volt elérhető.

Az A és E regiszterek együttesen 32 bites lebegőpontos aritmetikai műveletek megvalósítását is lehetővé tették, az X regiszter főként az indexelt címzésnél kapott szerepet, de mint általános regiszter is használható volt.

Az R 12-höz a SZKI kifejlesztett egy mikroprogramozott decimális aritmetikát, amivel a BCD kódolású számokkal közvetlenül lehetett műveleteket végezni.

Az R 10 mikro-programozott volt, abban az értelemben, hogy a műveletvégzőt, ami az egyes gépi utasítások végrehajtását végezte (utasítás dekódolás, címzés, memória hozzáférés, olvasás, regiszterbe töltés stb.) egy ROM memóriába beégetett mikro-program vezérelte. A műveletvégző egységnek volt két 1 bites átviteli és túlcsordulás jelző regisztere is, amelyek vizsgálhatók voltak.

A minibuszos csatlakozásnak köszönhetően az R 10-esnek praktikusan nem volt „alapkiépítése”, minden megrendelő olyan konfigurációt kapott, amelyet kért.

Piaci területtől függően a központi egység konzolja az ASR 33-as hengerfejes írógép volt, vagy egy Consol írógép – esetlegesen cirill klaviatúrával, ami teljesen más beosztású volt, mint az általunk megszokott QWERT.

Mivel a Consol írógépek igazi kocsival rendelkeztek, a kocsi visszafutása például komoly mechanikai rázkódást okozott. A kezelőprogramjába külön be kellett építeni egy időzítést, hogy addig, amíg a kocsi visszafut, ne lehessen kiírni karaktereket a gépre, mert különben 2-4 karaktert el tudott az írógép veszíteni, vagy teljesen véletlenszerű helyekre ütötte azokat.

Később a VT 340-es display-ek lettek a kezelői konzolok.

Rendszerdiszkként a már a VT 1010B-nél alkalmazott, ekkor már a MOM által gyártott 800 Kbyte kapacitású fixfejes diszk került alkalmazásra.

Adat be- és kivitelre a „klasszikus” lyukszalagos egységek szolgáltak, amelyek általában a MOM gyártmányú olvasó, lyukasztó és csévélő egységek valamilyen kombinációjaként kerültek beépítésre.

Szekrénybe épített lyukszalag olvasó- lyukasztó és csévélő

Lemezes tárolók tekintetében is volt választék: az ESzR (EC) típusszám szerinti EC 5052-es lemezegység (mi csak mosógépként emlegettük, ugyanis kb. fél íróasztalnyi helyet foglaltak el), 25 Mbyte-os cserélhető lemezes diszkegységek, vagy a bolgár IZOT 1370-es 6 Mbyte-os lemezes diszkek (3,5 Mbyte fix lemez, 2,5 Mbyte cserélhető egység) közül lehetett választani. Ha több IZOT 1370-es egység került egy rendszerbe – ami gyakori volt –, a szervizes kollégáktól komoly erőfeszítést igényelt, hogy az egységek egymás kazettáját olvashassák. Ennek a cserélhető diszk-egységnek volt „nyugati” változata, a DRY/DRI? diszk, hasonló kapacitás-paraméterekkel, de nagyobb megbízhatósággal.

EC (ESZ) 5052-es egységek és a cserélhető lemezek

Az IZOT 1370 a cserélhető kazetta nélkül

Az cserélhető egység – kazetta – egy lemezt tartalmazott

Mágnesszalagos háttértárak közül a legelterjedtebbek a 9 csatornás NML 67-ös szovjet mágnesszalagos egységek (ezekről mindössze egy ilyen képet találtam csak) voltak, vagy az NDK Zeiss gyártású egységek jöhettek szóba.

Egy NML 67-es egység bemérés közben.

Az NDK Zeiss gyártmányú EC 5017 mágnesszalag egységek.

Nyomatókból is nagy volt a választék: a VIDEOTON saját gyártású – Data Product licence alapján – 80, illetve 132 karakter széles dob nyomtatói vagy a mátrix nyomtatók közül lehetett választani, de plotter is szerepelt a választható berendezések között.

A képen elől a 132 karakteres VT nyomtatók, balra egy Zeiss szalagos egység, jobbra pedig a VDT terminálsorozat berendezései láthatók.

Adatátvitelei, illetve hálózati eszközök nagy választéka állt rendelkezésre: aszinkron- és szinkroncsatolók, illetve a saját fejlesztésű, COS (Common Scanner) nevű egység, amely képes volt egyszerre több fajta hálózati csatolót (aszinkron, telex, szinkron, SDLC, HDLC stb.) kezelni.

Ez a nagy eszközválaszték igen flexibilissé és – ma így mondanánk – jól skálázhatóvá tette az ESzR-sorozat legkisebb gépét.

Az R 10 szoftvere és programozása

A MITRA 15 – amiből az R 10 kialakításra került – megjelenése alapvető változásokat hozott a programozási környezetben. A MITRA 15-öt később követte a MITRA 125, amely gyorsabb volt, nagyobb memóriát volt képes kezelni, ezen túl – programozási szempontból – nem sokban különbözött az elődjétől, mint ahogyan az R 10 család R 12, R 11, R 10M tagjai sem.

Az utasítások száma jelentősen bővült: az R 10-nek már 77 utasítása volt (ebből 33 memória referenciális utasítás).

A MITRA 15-nek már volt monitor programja/operációs rendszere, ami egyrészt megkönnyítette a gép használatát, másrészt lehetővé tette a monitor által nyújtott szolgáltatások programból történő elérhetőségét, és így használatát is.

Monitorból is volt az „alapváltozat”, a MOB (franciául: Moniteur de Base), amely egyszerre egy program futtatására volt képes. A real-time monitor, MTR (franciául: Moniteur temps rèel) már tartalmazta azokat a funkcionalitásokat, amelyek a valós idejű feladatok futtatásához voltak szükségesek, de létezett az MMT, az ún. multi taszkos (franciául: Moniteur MultiTâches) változat, amely már megengedte a több feladat, több program egyidejű futtatását.

Az első hazai fejlesztések közé tartozott a PCM – Process Controll Monitor –, amely az ipari folyamatok kezeléséhez szükséges szinkronizáló és sorkezelő funkcióit is tartalmazta.

A TÁKI – Távközlési Kutató Intézet – kifejlesztett az R 10-re egy time-sharing operációs rendszert, ennek további sorsáról azonban nincsenek információim.

Az R 10-re a SZTAKI készített egy IDOS nevű rendszert, amely sokkal felhasználóbarátabb volt az eredeti operációs rendszereknél.

A géphez tartozó alapszoftver már lényegesen bővebb volt, mint a VT 1010B esetében. A gép alapvető működtetését és kezelését szolgáló monitor(ok) – operációs rendszerek – mellett megjelentek az ún. utility programok is:

- BIB diszkes könyvtárkezelő;
- BATCH parancsnyelv értelmező;
- MITRAS 1 és MITRAS 2 Assembler;
- PDL 2 programnyelv;
- EDL szerkesztők: az Assembler által generált ún. BT (Binaire Translatable) formátumú állományból vagy állományokból futtatható ún. IMT (Image Mèmoire Translatable) állományt állított elő;
- RCEDIT szövegszerkesztő;
- MAG 2 makrógenerátor, amely egy mai REGEX feldolgozó korai megfelelője volt;
- AMAP nyomkövető program – trace, snap, dump, patch, címen megállás stb. funkciókkal;
- Fortran;
- COBOL;
- PASCAL.

A monitor megjelenésével sokat egyszerűsödött a programfejlesztés: a különböző programok már a lemezes tárolóról – ami még mindig a VT 1010B-nél is használt 800 kbyte-

os fix fejes diszk volt – voltak betölthetők, a forrásállományok szintén lemezen tárolhatók voltak, így a gép kihasználtsága lényegesen javult.

Még egy újdonság is megjelent a gépen: az ún. bázisregiszter rendszer. Az R 10 4 bázisregiszterrel rendelkezett:

- G – globális bázisregiszter,
- L – lokális bázisregiszter,
- P – utasításszámláló,
- X – index regiszter.

Ezeknek a bázisregisztereknek a használatával biztosítottá vált a programok memóriában való mozgathatósága. Míg a VT 1010B-nél már a fordításkor meg kellett adni, pontosan hol – melyik lapon, azon belül milyen címen – kell az adott egységnek elhelyezkednie a memóriában, ez a megkötöttség az R10-nél megszűnt: a betöltő program – alapesetben – oda töltötte be a programot, ahol szabad helyet talált, ami általában egy, a monitort követő szabad hely volt. Természetesen volt lehetőség arra, hogy meg lehessen határozni, hogy a memória melyik címére történjen a betöltés, de a töltő azt már nem ellenőrizte, hogy az adott terület szabad-e.

Ezek a regiszterek a címzés – memória hozzáférés – szempontjából voltak lényegesek. A G regiszter jelölte ki azt a memóriaterületet, amely a teljes program számára elérhető volt, e regiszterhez képest lehetett direkten vagy indirekten címezni.

Az L regiszter és a P regiszter együttesen egy ún. szekciót határozott meg: a P regiszterben lévő címtől kezdődő programrész a hozzá tartozó L regiszter által kijelölt memóriaterülethez fért csak hozzá, szintén direkt vagy indirekt és indexelt címzéssel. Külön utasítással – CLS (call section) – lehetett az operandusban szereplő szekciót meghívni, ekkor a hívott szekció tárolt L és P regiszter értékei beíródtak a megfelelő regiszterekbe, és P értékének megfelelő helytől elindult a program végrehajtása. Egy ilyen hívás az A, E, X regiszterek tartalmát változatlanul hagyták, így ott is át lehetett paramétereket adni a hívott szekciónak. A szekcióból való visszatérésnél az elmentett P és L érékek visszaíródtak a regiszterekbe, így folytatódhatott a hívó szekció végrehajtása.

Szekción belül is volt lehetőség eljárás-hívásra, az SPA (store program address) utasítás az operandusában megadott helyre elmentette a saját címe+4 értéket, ami nem más, mint az utasítást követő második utasítás címe, azaz a folytatási pont az eljárásból való visszatérés után. Az SPA utasítás után ui. kellett egy ugrás – BRU (branch unconditional) –, ami az eljárás belépési pontjára történt, ezért kellett a 4-el való növelés. Az eljárásból való visszatérés az SPA

utasítás operandusa által meghatározott címen keresztüli – itt volt eltárolva a folytatási cím – indirekt ugrással volt megoldható.

A forrásszövegek javítására szolgáló RCEDIT is jelentősen megkönnyítette a munkát, még ha a működési filozófiája nagyban eltért a ma megszokott szövegszerkesztőkétől.

E működési filozófia arra alapult, hogy az alap forrásállományt, amin a szerkesztés megindult, azt mindaddig megtartotta, és kiindulási alapnak tekintette, amíg egy külön parancs hatására az addigi összes szerkesztési parancs-konszolidálásra nem került.

A forrásszöveg szerkesztése úgy történt, hogy a szerkesztési parancsokkal, amelyek a törlés, mozgatás, beillesztés funkciókat valósították meg, a forrásállomány sorainak számára hivatkozva, előállt egy új forrásállomány – ezt azonban közvetlenül vizualizálni nem lehetett – , amely azután ismét forrásállományként szolgált. Az így megszerkesztett forrásállományból keletkező lista – pontosabban: a lista sorszámai – lettek a következő szerkesztés hivatkozási helyei.

Az egyes parancsok tehát mindig meghatároztak egy funkciót, valamint egy kiindulási pontot – sorszámot –, amelytől vagy amelyen az adott funkciót végre kellett hajtani, illetve kiegészítő paramétereket is meg lehetett adni, mint például a törlendő vagy átmozgatandó sorok számát. Beszúrás művelet esetében a parancsot követő sorok – a következő parancsig – beszúrásra kerültek a forrásállomány megadott helyéről.

Mai szemmel nézve elég bonyolult és sok hibalehetőséget rejtő megoldás volt ez a szerkesztési mód, viszont remekül illeszkedett a batch feldolgozási módhoz.

A batch feldolgozási mód azt jelentette – némileg hasonlóan az MS-DOS batch parancsához –, hogy vezérlő parancsokkal helyettesíteni lehetett az operátori beavatkozást, így egy komplett munkamenetet egy szöveges állományban – ez vagy lyukkártyák sorozatában vagy bármilyen szöveges forrásállományban – testesült meg, amit az operátornak csak el kellett indítania. Természetesen az olyan műveleteket, mint diszk lemez vagy mágnesszalag csere, azt az operátornak manuálisan kellett végrehajtania.

Ez a batch feldolgozási mód kétségkívül hatékonyságnövelő volt ott, ahol az R 10-et adatfeldolgozás jellegű feladatokra használták.

Az intézetben ez az üzemmód sok előnnyel nem bírt, hisz nálunk a fejlesztési idő túlnyomó többségét a tesztelés és a hibakeresés tette ki, amit ráadásul olyan környezetben – operációs rendszer, perifériacsatolók, adatátviteli vezérlők (eszköz és/vagy szoftver) – kellett elvégezni, amely erősen eltért az általánosan használt környezettől, és a tesztelés gyakran azzal járt, hogy a teljes rendszert újra kellett indítani.

Erre való tekintettel alakult ki az a megoldás, hogy fordításra, forrásszerkesztésre, a program összeállítására dedikált gépek szolgáltak, majd az eredményt már egy önálló konfiguráción teszteltük, hogy a gyakori leállítás, újraindítás, memóriában való módosítás stb. ne zavarjon senkit.

Mivel az ilyen teszt-konfigurációkból soha sem lehetett elég, az éjszakai munkavégzés – megfelelő szervezéssel – bevett gyakorlattá vált. A megfelelő szervezés azt jelentette, hogy 2-3 fejlesztő csoport összeállt, és amíg az egyik csoport a programlistákat bújta, forrásszöveget javított, a másik csoport tesztelte a programját, majd cseréltek.

A szigorúan vett programozás szinte semmiben sem különbözött a VT 1010 B-n történt programozási munkától, de megszabadultunk a lapszervezés kötöttségeitől, általában nem kellett a ki- és beviteli műveletek részletes programozását végrehajtani, mert azokat, mint monitorszolgáltatást lényegesen egyszerűbben elérhettük. Ugyancsak megszűntek a forrásállományok újra előállításakor gyakran fellépő gépelési hibák.

A futtatható program előállításának folyamata is jelentősen módosult a VT 1010B-nél alkalmazott folyamathoz képest. Míg a VT 1010B-nél a nagyobb programok összeállítása forrás szinten történt – az összes forrásállományt megfelelő sorrendben be kellett olvasni egy fordítási menethez –, addig az összeállítás feladatát az R 10-en átvette az EDL szerkesztőprogram. Az EDL megengedte, hogy az Assembler által lefordított ún. BT állományokból állítsa össze a futtatható állományt.

Ezzel a megoldással sikerült megvalósítani a programok memória foglalásának csökkentést biztosító ún. overlay szerkezetű programok készítését. Az overlay technika alkalmazásával optimalizálni lehetett a programok memória foglalását. A multi task-os környezetben, ahol a 64 kbyte-os memóriában a monitor mellett több program is helyet foglalhatott, az egyes programok helyfoglalása már nem volt közömbös. Hozzátéve, hogy az R 10 alap kiépítése egy 8 kByte-os memóriát jelentett, így egy-egy program memória helyfoglalása egyáltalán nem volt közömbös.

Az overlay szerkezetű program úgy volt felépítve, hogy volt egy ún. törzse, ahol azok a szekciók szerepeltek, amelyeknek állandóan a memóriában kellett lenniük, míg azok a szekciók, amelyeknél nem volt szükség arra, hogy állandóan a memóriában legyenek, az overlay fa ágaiba kerültek, figyelemmel arra, hogy két ág szekciói nem hívhatják egymást, mert egy ág voét memóriába tölthető. Egy ágnak szintén lehetettek ágai. Ezt a fa struktúrát a programozónak meg kellett meghatároznia, az egyes szekció-hívásoknál az EDL biztosította, hogy a megfelelő ág a háttértárolóból – rendszer diszk – a memóriába kerüljön.

Programozói szinten – a fa kialakításán túl – a technika alkalmazása azt jelentette, hogy néha át kellett az programstruktúrát alakítani, hogy a párhuzamos ágak egymás közötti hívásainak tilalmát biztosítani lehessen. Ráadásul többnyire a program elkészülte után derült ki, hogy az esetleg túl sok helyet foglalna el az operatív memóriában, így utólagosan kellett az overlay fa struktúrát kialakítani, hogy minél kisebb memória helyfoglalást lehessen elérni. Hozzátéve, hogy semmiféle eszköz nem állt rendelkezésünkre ahhoz, hogy a szekció hívási struktúrája megjeleníthető legyen.

Ez a szervezési mód tényleg jelentős memória helyfoglalás csökkentést eredményezhetett, természetesen a futási idő – az ágak betöltéséhez szükséges overhead idő miatt – megnövekedett, de a nagy memóriafoglalású programok nem voltak futásidő kritikusak, így ez a megoldás senkit sem zavart.

Az Assembler mnemonikjai angol alapra kerültek (pl. egy szó A regiszterbe töltését a VT 1010B-n az APM mnemonik jelezte, az R 10-en ez már LDA volt), így a programok érthetősége nagyban javult – már ha az olvasónak voltak némi Assembler ismeretei.

Egy nagyobb program esetében nem kellett állandóan az összes forrásállományt újrafordítani, elég volt egy-egy egység – alprogram, szekció – fordítását elvégezni, a tényleges program-összeállítás az EDL feladata volt.

Ebben az időben már ismert volt a „strukturált programozás”, vagy a „GOTO nélküli programozás” paradigma, ami a magas szintű nyelveken viszonylag egyszerűen alkalmazható volt, mi azonban Assemblerben programoztunk.

A strukturált programozás és az Assembler nyelvű programozás összeházasításából született meg a PDL 2 (Program Definition Language) „nyelv”. Az alapötlet az volt, hogy írjuk le a programot jól strukturáltan, tehát a jól ismert **begin end**, **if then else**, **case of other**, **while do, for** stb. utasításokkal kellett leírni a program működését, pontosabban a program logikai struktúráját írtuk így le. A „változók” pedig hosszabb szövegek is lehettek, ami elősegítette a megértést. Természetesen a commentek is megengedettek voltak. A PDL nyelv annyira „PASCAL like” volt, hogy pl. a **structure** használata is megengedett volt.

A PDL fordító a megfelelő helyekre elhelyezte azokat az ugró utasításokat és címkéket, amelyek az adott struktúrának megfeleltek. Ezután a programozó az egyes címkék által azonosított helyekre beírhatta a szükséges Assembler-utasításokat, és/vagy a feltételeknek megfelelő ugró utasításokat stb. Így előállt egy forrásállomány, amelyet az Assembler már le tudott fordítani úgy, hogy a PDL utasítások automatikusan már commentként jelentek meg.

Ezzel a megoldással a programok struktúráltsága szükségszerűen javult, áttekinthetővé, könnyen érthetővé, olvashatóvá váltak, a program szinte automatikusan jól kommentált lett,

ugyanakkor sikerült megtartani az Assembler által biztosított rugalmasságot. Csak egy apró megjegyzés a fentiekhez: a PDL francia fejlesztés volt, ezért a fentebb leírt „magas szintű utasítások” nem angol, hanem francia nyelvűek voltak (pl.: az IF THEN ELSE helyett a SI ALORS SINON szerepelt). Ennyit a (köz)érthetőségről.

Később – amikor megoldottá vált a távoli hozzáférés, kiegészítve a multitasking működési móddal – jelentek meg azok a szövegszerkesztők, amelyek már a szöveges forrásállományok tartalmát a ma megszokott módon tudták szerkeszteni.

A mi csoportunk adatátviteli területen dolgozott, egyrészt a szocialista táboron belüli alkalmazásokon (Rostocki kikötő, Bolgár posta telexközpontja, Szovjetunió Gázipari Minisztériuma, Szovjet vasút stb.), másrészt a francia partner által tervezett ún. adatátviteli monitorral kapcsolatos munkákon dolgoztunk. A francia rendszer képes volt integráltan kezelni csomagkapcsolt (X.25, TransPac – https://en.wikipedia.org/wiki/Transpac_(data_network)) hálózatokat, a szinkron X.21 kapcsolatokat, telex és más aszinkron kapcsolatokat. A rendszer az OSI 7 rétegű modelljén (https://en.wikipedia.org/wiki/OSI_model) alapult, mi főleg a Transport – Session rétegekhez tartozó területén dolgoztunk.

Mivel akkoriban itthon elérhető X.25-ös hálózat sem, a Transpac elérés meg egyáltalán nem létezett, a megírt programok éles tesztelését főleg Franciaországban végeztük, azután hogy itthon pl. szimulátorral kipróbáltuk. Hibák azonban maradtak, és ezek a hibák – többnyire – bizonyos események egybeesésének a következményei voltak, ebből következően nehezen reprodukálhatók voltak.

A rendszert fejlesztő/működtető/használó francia kollégáktól természetesen lehető legrészletesebb hibaleírást kaptunk (hol állt meg a program, mi volt a regiszterek tartalma, mikor jelentkezett a hiba stb.), de ezek az információk csak arra voltak elegek, hogy a kapott információkból ki tudtuk találni a lehetséges hiba okát, rosszabb esetben az okokat.

Ahhoz, hogy biztosak lehessünk abban, hogy valóban a feltételezett ok vagy okok együttes fennállása okozta a hibát, nyomon kellett követni a program futását, de úgy, hogy minimálisan avatkozzunk be a program futásába. Ezen felül pedig arra is ügyelni kellett, hogy a számunkra érdektelen esetek, állapotok ne kerüljenek be a nyomkövetési információk közé.

Erre azt a megoldást találtuk ki, hogy a megírt programok után 100-200 szónyi üres területeket fenntartottunk, ahová pultról be tudtunk írni olyan kis programot gépi kódban, ami figyelte a kritikusnak feltételezett állapotok előfordulását, és ekkor a program állapotáról a szükségesnek tartott információkat elmentette egy ciklikus pufferbe. Utána a pufferterületről készített memória dump már többnyire elég információt nyújtott ahhoz, hogy itthon kielemezve az adatokat, megoldást találjunk a problémára.

A javítást – ha az megoldható volt – ugyanígy a szabadon hagyott helyekre a pultból vittük be, és vártuk, hogy éles üzemben hogyan teljesít. Ha a megoldás sikeres volt, akkor a javítás bekerült a forrásállományba, és a legközelebbi általános javításban már meg is jelent. Ellenkező esetben újra hibát kellett keresni, viszont annyival jobb helyzetben voltunk, hogy több információ állt rendelkezésünkre.

Az R 10-hez már volt monitor/operációs rendszer, ami nekünk, programozóknak igazán nagy segítséget jelentett, mert egyszerűsítette és meggyorsította a programok fejlesztését.

A standard monitora számos, nagyon gyakran használt funkcionalitást biztosított – pl. kódkonverziók, adatmozgatás stb. –, de hiányoztak olyan funkciók, amelyek a folyamatirányítás, a real-time alkalmazások területén elengedhetetlenül szükségesek voltak – események, szemaforok, sorok kezelése –, ezért kifejlesztésre került a PCM – Process Controll Monitor –, amelyben megvalósították a fejlesztők az említett funkcionalitásokat.

Egy elég érdekes jellemzője volt még az operációs rendszernek/monitornak: a hibakezelése. A francia fejlesztők nem cizellálták túl a dolgot: ha a monitor valami olyan ágra került, ami valamiféle hiba – operációs rendszer szintű hiba – fellépését jelezte, akkor a monitor egy önmagára történő ugrásra futott (hex. C700 – BRU $). A pult adat-lámpáin ez a kód került kijelzésre, a cím-lámpákon az utasítás címe jelent meg, innentől kezdve indult a „hibakeresés”. Kezdetben az monitor bináris szalagjából egy általunk írt „antiassembler”-el előállított forráskódból a cím alapján megkerestük az aktuális BRU $ utasítást, és megpróbáltuk visszafelé indulva megfejteni, mégis mi okozhatta a megállást. Későbbiekben annyit javult a helyzet, hogy megkaptuk a monitor forráskódját, amin azért egyszerűbb volt eligazodni – elvileg. Hozzátéve, hogy a monitorforrás nyomtatva körülbelül arasznyi vastag nyomtatott listát jelentett. Ez a nyomtatott forráskód egy valamikori állapotot tükrözött, aztán hogy az épp futó monitor milyen változat volt, az egy másik kérdés volt, de szerencsére a jellegzetes utasítást egy viszonylag kis környezetben – cím alapján – elég könnyen meg tudtuk találni, és indulhatott a nyomozás. Viszont ha sikerült tisztázni a leállás okát, akkor a francia monitorfejlesztő kollégák – később a kooperációban dolgozó hazai fejlesztőkkel együtt – igyekeztek megoldani, hogy ilyen megállás többé ne fordulhasson elő. Szerencsére egyre ritkábban kellett ilyen jellegű hibákat keresnünk.

A monitor az adatok be- és kivitelében nyújtott támogatása lehetővé tette, hogy két operációs rendszerhívással lehetett megvalósítani egy be- vagy kivitelt. Egy blokkban meg kellett adni a be- és/vagy kiviteli művelet fő leíróit (cél- vagy forrásterület, adatmennyiség, a logikai eszközt – operációs címke –, amiről vagy amire az átvitel irányult, és a megfelelő parancsot). Ezt a blokkot át kellett adni az operációs rendszernek egy monitorhívás – CSV M:IO

– paramétereként, ami a kért átvitelt elindította, ezt követően pedig egy CSV M:WAIT hívással meg tudtuk várni az átvitel végét. (A Call Supervisor – CSV – utasítás jelentette egy monitor-szolgáltatás hívását, az M:IO vagy M:WAIT pedig a kiválasztott monitor funkciót azonosította).

Természetesen, ezek az átviteli műveletek elemi átvitelekre vonatkoztak, ha már valamilyen filerendszer – diszkes vagy mágnesszalagos, a maguk nem éppen egyszerű címkerendszerével – is érintett volt, akkor már nem lehetett ilyen egyszerű módon megoldani, de a különböző filekezelők szolgáltatásainak az elérése sem volt már programozói szempontból nagyon munkaigényes.

Egy ilyen elemi átvitel forrása vagy célja az operációs rendszerben egységesített lett: egy ún. operációs címke határozta meg. Ez a címke egy előre definiált menmonik volt, amit az Assembler értelmezni tudott. Voltak ún. standard, illetve felhasználói címkék.

A standard címkék (M:xx) ismertek voltak a monitor számára, a felhasználói címkék (U:F1- U:FF) kezeléséről pedig – értelemszerűen – a felhasználónak (felhasználói programnak) kellett gondoskodnia.

Példaként néhány operációs címke:

- M:SI – source input – forrásbemenet,
- M:LO – listing output – listakimenet,
- M:DO – diagnostic output – hibakimenet,
- M:OC – operational consol – kezelői konzol,
- M:UL – user library – felhasználói könyvtár a rendszerdiszken.

Ezekhez az operációs címkékhez a monitor hozzárendelt alapértelmezett perifériákat, de operációs rendszer paranccsal – %ASSIGN – ezt a hozzárendelést meg lehetett változtatni. A perifériák is standard neveket kaptak (T:yy), például:

- T:TY – teletype – operátori be- és kiviteli berendezés,
- T:LP – line printer – sornyomtató,
- T:DC – system disc,
- T:9T – 9 track magnetic tape – 9 csatornás mágnesszalag,
- T:PR – punch reader – lyukszalagolvasó,
- T:DM –moblie disc – cserélhető lemezes diszk,
- T:NO – NOP – nincs periféria művelet.

Azzal, hogy monitor/operációs rendszer bevezette az operációs címke, illetve a standard perifériák fogalmát, és tulajdonképpen két operációs rendszerhívásra (M:IO, M:WAIT)

egyszerűsítette a ki- és bevitel megvalósítását, sok programozási munkát takarított meg számunkra. Ugyanakkor kellő flexibilitást biztosított arra, hogy célnak és feladatnak legjobban megfelelő be- és kiviteli egységet ki lehetett választani úgy, hogy a programon nem kellett semmit sem változtatni.

Az egyes perifériák hardver szintű kezelő programjai – handlerek az R10 terminológiában – természetesen részei voltak az operációs rendszernek. Egy ilyen perifériakezelő program – handler – két részből állt: az átvitel inicializálását és tényleges fizikai indítását végző részből, illetve a megszakítási szinten futó, a perifériától visszajövő jelzések, adatok stb. kezelését, illetve az átvitel befejezését, és esetleges hibajelzést kezelő részből. Az inicializáló rész bemenő paraméterei, amelyeket a monitor adott át a kezelő programnak – ez volt az ún. H1 –, szabványosítottak voltak, ugyanúgy szabványos volt a monitor számára visszaadandó, az átvitel lezajlásával kapcsolatos információk átadási módja is, amit az ún. H2-nek kellett megvalósítania. Ezeknek a szabványos interfészeknek a megtartásával már bármilyen periféria fizikai szintű kezelését meg lehetett oldani.

Nyilvánvaló memóriapazarlás lett volna, ha minden periféria kezelő programja állandó részét képezte volna az operációs rendszernek, ami memória rezidens volt, ezért – némi túlzással – minden egyes konfiguráció egyedi operációs rendszert kellett hogy kapjon, hogy optimálisan illeszkedjen az aktuális periféria kiépítéshez. Jelentős fejlesztési munka eredményeként sikerült megvalósítani, hogy néhány parancs és tábla megadásával az operációs rendszer generálása – ami nem volt egy egyszerű feladat – szinte automatizálható vált.

Nekünk meg elég munkát adott az, hogy az egyes perifériák és/vagy csatolóik kezelő programjait kifejlesszük és/vagy karbantartsuk, mert csaknem minden hardverfejlesztés és -javítás szoftver vonzattal járt: változtatni kellett a hardver szintű kezelő programokon.

Ezekkel a ma talán szegényesnek nevezhető eszközökkel azért sikerült olyan programokat létrehozni, mint az 1975. évi Budapesti Vívó Világbajnokságon használt rendszer, az 1980-as Moszkvai Olimpia kijelző vezérlő rendszere, a Berlin Schönefeldi repülőtéren működő beszállító rendszer, valamint – amit akkortájban sokan láthattak és használhattak is – a DOMUS bútoráruház rendszere (az áruház lépcsőházában az egyik szinten a működő R10-es látható is volt).

Akkoriban nem kapott nagy nyilvánosságot, de jelentős számú katonai rendszerben működött az R 10 vagy annak katonai változata, itthon és külföldön is.

Nagy népszerűségnek örvendett az R 10 alapú VIDEOPLEX rendszer, amely speciális munkaállomásokkal (VIDEOTON kis hordozható TV, speciális klaviatúra) adatrögzítő feladatokat látott el.

Hogyan és miket programoztuk/programoztam

A következőkben néhány olyan fejlesztésről, programról szeretnék írni, amelyek szorosan kapcsolódnak a VIDEOTON gépekhez, illetve még vissza tudok emlékezni rájuk, mert valamiért ilyen mély nyomokat hagytak bennem. A következők tehát személyesek és ebből következően szubjektívek lesznek.

Az első „igazi" R 10-es fejlesztési munkám a VT 340-es display párhuzamos csatolójának az operációs rendszerbe való integrálása volt, így a VT 340 már egy gyors operátori konzolként is használható volt, így a mechanikus írógépek kiváltásra kerültek.

A VT 340 display

Közreműködtem egy „anti-assembler" program elkészítésében is, amivel sikerült az operációs rendszer mélységeibe is behatolni, amire a többnyire francia nyelvű dokumentáció alapján esélyünk sem lehetett. A program első változata a memóriaképből volt képes arra, hogy magát a monitort egy programlista formájában előállítsa.

Sajnos az akkoriban még kötelező 2 éves sorkatonai szolgálat miatt a fejlesztési munkákból kiestem, viszont a Magyar Néphadsereg (akkor még így hívták) számítógépesítésében aktívan részt vehettem. (Nem mintha lett volna sok választásom…) Itt értelemszerűen a felhasználói programozás került előtérbe: készítettünk bérszámfejtő programot, na meg igazi katonai célú programot, amit lehetett a megfelelő helyeken mutogatni. A program képes volt arra, hogy meghatározza, hogy mi fog történni akkor, ha a Magyar Néphadsereg egy bizonyos egysége (esetleg az itt tartózkodó Szovjet Hadsereg és/vagy a Varsói Szerződés egységeivel kiegészülve) szembe kerül a NATO valamilyen egységével. A szükséges algoritmusokat megkaptuk, „csak" a programokat kellett megírni. Mindezt természetesen Assembler nyelven, mert nem nagyon volt más akkoriban. További szépség volt

a dologban, hogy az összes érintett – Varsói Szerződés, Magyar Néphadsereg, NATO stb. – teljes személyi és fegyveres állománya össze volt gyűjtve egy adatbázisban. És ha valami titkos, na ez akkor a titkosnál is titkosabb volt! Még az eredményeket is csak megfelelő mértékben titkosított csatornán lehetett elküldeni. Ennek akkor volt jelentősége, ha hadgyakorlat volt, a felhasználók távol voltak, a futás eredményét is távoli helyszínre kellett kiküldeni. Amíg kiépült a megfelelő titkosítás, úgy biztosítottuk az üzenet megfejthetetlenségét, hogy az eredménytáblázat(ok) úgy lettek telexen elküldve, hogy az oszlop- és sorfeliratok hiányoztak, viszont egy átlátszó lapon a teljes táblázat(ok) fel volt(ak) rajzolva, és ha valaki alátette a megfelelő telexlapot, akkor értelmes, értelmezhető táblázatot látott. Futtatás után a teljes mágneslemezes állományt mágnesszalagra kellett kivinni, a lemezt letörölni, U* karakterekkel teleírni a használt területet, a szalagok meg mentek a páncélszekrénybe. Persze a szalagokról kellett egy másolat, mert nem volt 100%-osan biztos, hogy vissza is lehet olvasni/tölteni egy mentést. Hozzátéve, hogy maga a gépterem, ahol az R 10 üzemelt, egy szigorúan őrzött területen belül volt.

A seregben is történtek furcsa dolgok, az első mindjárt az R 10 telepítésénél. Vadonatúj gépterembe került a gép, üvegfallal elválasztva a mágnesszalagos egységek (NML 67-es, 4 darab), külön klímával, álpadló, nagy ablakok a gépteremmel szomszédos helyiség felé, hogy a látogatók jól láthassák a gépet.

De felmerült a kérdés: hogyan is helyezkedjenek el a konfiguráció elemei, hogy dolgozni is rendben lehessen, meg a látogatók is láthassák, amit kell. Sikerült kitalálni egy elrendezést, ami nagyjából megfelelt mindenkinek és minden igénynek. Vitték a tervet a vezetésnek, ahonnan élből jött a kritika: nem jó ez így. Kiderült, azt szerették volna, ha a mágnesszalagok a központi egység szekrénye mögé kerülnek. Arra a kérdésre, hogy mégis miért, megkapták a választ: nem tudjátok? Hiszen ezek háttértárak.

A másik furcsa dolog a már említett katonai programmal volt kapcsolatos. A program annyit tudott, hogy kiszámolt egy arányt – mindenféle titokzatos paramétert figyelembe véve – a két szembenálló fél erőforrásai között. Ebből az arányból származott a probléma: előfordult, hogy egy bizonyos erőforrás valamelyik oldalon nem létezett, azaz néha nullával vagy nullát kellett volna osztani. Erre azt a megoldást találtuk ki, hogy az eredmény * lett. Ez a megoldás mindenki számára elfogadható volt, ez került alkalmazásra.

Az egyik gyakorlaton, ahol a már említett titkosítással folyt az adatcsere, telefonált a nagyfőnök: valami nagyon nincsen rendben a kiküldött adatokkal, mert vannak helyek, ahol * jelenik meg. Mondtuk, hogy ez bizony azért van, mert nullával nem lehet osztani. Némi

gondolkodás után megkaptuk a parancsot: nem érdekes, oldjátok meg! (Megoldottuk: a kisebb főnökség elmagyarázta, nem kevés idő ráfordítással, mit is jelentett a *.)

Leszerelés után ismét az alapszoftver-fejlesztések kerültek előtérbe: mágnesszalagos filekezelő, VT 56100 típusú szinkronterminál – ami akkoriban 900 – 1200 bps sebességű telefonvonalon, valamint VIDEOTON gyártású modemekkel pont-pont kapcsolatokat építettünk ki. A 2400 bps-es sebesség különlegesnek számított.

5 évet Prágában töltöttem mint vevőszolgálati munkatárs, az akkor még Csehszlovákiában működő R 10 majd R 11-es gépek szoftverszervizét láttuk el, illetve némi fejlesztéseket végeztünk. Azokban az években Csehszlovákiában sok R 10 üzemelt, amikre még emlékszem, mert eléggé különlegesek voltak:

- A prágai repülőtéren üzemelt egy két R 10-ből álló konfiguráció, amelyek a memóriák direkt összekapcsolásával alkottak egy egyedi, hibatűrő rendszert. Ennek a memória-memória összekötésnek a megvalósítása nem volt egyszerű feladat.
- 2 prágai kórházban is üzemelt R 10.
- Osztravában, egy nagy kohóműben folyamatszabályozási feladatokra alkalmaztak R 10-et, a kladnói vasműben is üzemelt R 10.
- Csehszlovákiában két helyen is katonai célra használtak R 10-eket, a pontos cél – számomra – nem volt publikus, de mindkét helyen 5-5 mágnesszalagos egység volt telepítve, így feltehetően adatfeldolgozást végezhettek velük.
- Pozsonyban a meteorológia radarállomáson dolgozott R 10, valamint a Jaslovske Bohunice-i atomerőműben is használtak R 10-et.
- Üzemelt még R 10 Kassán, Besztercebányán, Nyitrán, Zsolnán, Ceské Budejovicében.

Visszatérve itthon az adatátviteli programokra „szakosodtam”:

- Az R 12-hez készült egy csatoló és illesztő program, amely az IBM 3275-ös terminált emulálta.
- Készítettünk egy aszinkron X.21-es felületet megvalósító telex-kezelő programot, amelyet az NDK-ban a rostock-i kikötőben, Szófiában a posta telex központjában használtak. Bulgáriában még meg kellett oldani az európai, cirill és görög 5 bites telexkód eltéréseit. Ugyanez a telexcsatoló és illesztése működött a Szovjet Vasút számára szállított rendszerben, aminek az volt a különlegessége, hogy egy olyan városban üzemelt, amely akkoriban ún. zárt város volt. Ez azt jelentette, hogy oda

külföldi be nem tehette a lábát. Viszont a rendszer néha némi javításra, karbantartásra, illetve továbbfejlesztésre, bővítésre szorult. Ezt úgy oldotta meg a szovjet fél, hogy a gépet szétszedték, Moszkvába szállították, ahol egy pályaudvar várócsarnokában összerakták, és itt már elvégezhettük a szükséges feladatokat, majd a kijavított, bővített rendszer visszakerült állandó működési helyére. Egyszerű, nem?

- Francia kollégákkal együttműködve fejlesztettük a TCM-et (**T**ele**c**ommunication **M**onitor), amely az R 11 hálózati lehetőségeit bővítette, és Franciaországban éles üzemben működött. Itt szerezhettünk tapasztalatokat a nálunk akkoriban még terjedőben lévő X.25-ös csomagkapcsolt hálózatokról. Később a kezelt hálózati protokollok kibővültek az SDLC és HDLC kezelőkkel is. TCM alapon építettünk ki Szibériában, Tyumenyben a Szovjet Gázipari Minisztériumnak egy 8 vagy 10 csomópontból álló számítógépes hálózatot, ami 9600 bps-es telefonhálózatra lett tervezve, de igazán inkább 4800–2400 bps-on üzemelt. Tyumeny is igen furcsa város volt több szempontból is: télen rettenetesen hideg tudott lenni (a –30 fok vagy hidegebb szinte normális volt), a csapvíz gyakorlatilag ihatatlan volt. Ha állni hagytuk a kiengedett vizet, észrevehetővé vált a víz felszínén egy vékony olajfilm. Nem tudni, mi okból, de ide, Tyumenybe is, és vissza Moszkvába csak úgy repülhettünk, ha éjszakai gépet vettünk igénybe.
- Mivel a fejlesztők közül mi voltunk azok, akik a legtöbb adatátviteli, hálózati tapasztalattal rendelkeztek, minket hívtak minden olyan rendszerhez – vagy jobb esetben: tervezett rendszerhez –, ahol adatátvitel előfordult vagy felmerült. A székesfehérvári gyárban is dolgoztak olyan vevőszolgálati kollégák, akik szintén rendelkeztek adatátviteli tapasztalatokkal (amelyet többnyire úgy szereztek meg, hogy együtt dolgoztunk egy-egy rendszeren), így meg tudtuk osztani a feladatokat.

Interpretatív programozás

Az interpretatív programozás lehetővé tette, hogy egy programot programozói ismeretek/tudás nélkül is meg lehessen írni, mindössze néhány jól meghatározott utasítássorozatot kellett csak „megtanulni”. Ez a „megtanulás” tulajdonképp néhány sor másolását jelentette. Amit csak másolni kellett, az az a szekvencia volt, hogy hogyan kell áttérni Assembler nyelvről interpretatív nyelvre. Ugyancsak meg kellett tanulni, hogyan lehet eljárásra és változókra hivatkozni – milyen Assembler-utasítást/direktívát kell használni.

Amit nem lehetett megspórolni: a program logikáját pontosan meg kellett határozni, célszerűen egy blokkdiagramot – folyamatábrát – kellett legalább megrajzolni.

Amennyiben a program végrehajtásához elegendőek voltak azok a standard funkciók, amelyeket a szállító a géphez mikroprogram könyvtárként biztosított, akkor tényleg gyorsan, hatékonyan, minden programozói tudás nélkül elég bonyolult programok is megírhatók voltak, és a gép tényleg általánosan használható volt pl. adatfeldolgozási feladatok, mérnöki számítások stb. elvégzésére.

Hogyan oldották meg ezt a VT 1010B-n?

Egy interpretatív nyelven megírt program nem volt más, mint egymást követő eljárások sorozata. A feltétel-vizsgálatok, elágazások is mind-mind eljárásként voltak megvalósítva, azaz egy vizsgálat pl. úgy nézett ki, hogy meg kellett adni a vizsgálati feltételt (ami gyakran az eljárás nevét jelentette), és paraméterként az a címet – címkét –, ahol a megfelelő feltétel bekövetkezésekor a programnak folytatódnia kell.

Egy konkrét eljárás meghívása – úgy, hogy nincsen gépi utasítás eljárás-hívásra – olyan módon volt megvalósítható, hogy követni kellett egy megadott, egyszerű szabályt: eljárás neve, paraméterek felsorolása, következő eljárás neve, paraméterek és így tovább.

Mivel a program fordítására csak egy Assembler állt rendelkezésre, ezért az Assembler nyújtotta lehetőségeket kellett kihasználni.

Az Assembler képes volt arra, hogy

1. egy szimbolikus címből – címke – memóriacímet állítson elő;
2. egy adott memóriaterületet lefoglaljon – változó helyfoglalása;
3. numerikus vagy szöveges értékeket rendeljen egy memóriaterülethez – változó inicializálása, vagy konstans előállítása;
4. kezelje a program indításával kapcsolatos információkat;
5. kezelje a program memóriában való elhelyezésével kapcsolatos információkat;
6. lefordítson Assembler nyelvű utasításokat.

A szigorúan vett interpretatív program szempontjából az 1-3 pontokban leírt szolgáltatások voltak a fontosak, a 4-6. pontok szolgáltatásai „csak” arra kellenek, hogy

- meghatározható legyen a program indítási pontja, illetve a memórián belüli elhelyezkedési helye;
- át- esetleg vissza lehessen térni az interpretatív módból a normál assembly módra – hozzátéve, hogy erre a visszatérésre csak abban az esetben van szükség, ha nem csak a standard mikroprogramokat kell a programban használni, de most tételezzük fel, hogy nem ilyen esetről beszélünk.

A standard mikroprogramok forrásprogramjai ún. szekciókba voltak csoportosítva, olyan módon, hogy a szekció eljárásai által közösen használt részek után következtek a tényleges eljárások. Ez a megoldás lehetővé tette, hogy a szekció végéről tetszés szerinti számú mikroprogram – eljárás – elhagyható volt. Ami egyben azt is jelentette, hogy az utolsó mikroprogram volt csak használható, ha az előtte lévő teljes állomány is lefordításra került.

A szekciók forrásállományában nem voltak megadva memória elhelyezési információk, pontosabban lapokra vonatkozó információk, ezeket a fordítás során kellett megadni. Ugyanakkor voltak olyan szekciók, ahol az oldalon belüli kezdőcímre megkötések szerepeltek, amelyek figyelmen kívül hagyása problémákat okozhatott. Ezek a megkötések természetesen az egyes mikroprogramok dokumentációiban szerepeltek.

Egy eljárás hívása pedig mindig olyan formában történt, hogy meg kellett adni az eljárás nevét, utána pedig – az eljárás dokumentációjának megfelelő sorrendben – az eljárás paramétereit. Az eljárás paraméterei vagy változók vagy konstansok lehettek. Ha változó, akkor minden esetben kötelezően a változó nevét – pontosabban: annak címét – kellett megadni, amit az ASTROL Assembler egy pszeudo utasítással támogatott: elég volt a

* változó név – pl. * varx

formát használni. Ha paraméter konstans volt, akkor annak az értékét kellett megadni, amit az Assembler szintén támogatott: vagy a NB vagy a CH (szöveges konstans esetében) pszeudo utasításokkal. Az, hogy az eljárás paraméterei közül melyek a bemenő, illetve kimenő paraméterek, azt az eljárás dokumentációja határozta meg.

Ilyen módon egy interpretatív program a következő struktúrában jelenik meg:

Deklarációk (változók, konstansok)
változók, konstansok
Program áttéréshez szükséges Assembler-utasítások
eljárás1
változó1 *eljárás1 paraméter
változó2 * eljárás1 paraméter
eljárás2
változó2 * eljárás2 paraméter
változó1 * eljárás2 paraméter
változó3 * eljárás2 paraméter

Ha a fenti struktúrát Assembler-programként írnánk le, akkor a következő forrásállományt állítanánk elő:

VAR1 RES 2 * 1. változó, 2 byte lefoglalása

VAR2 NB 0 * 2. változó

VAR3 NB 0 * 3. változó

áttérési szekvencia

* **PROC1 * első eljárás neve – az eljárás meghívása**

*** VAR1 * első paraméter**

*** VAR2 * második paraméter**

*** PROC2 * második eljárás neve – az eljárás meghívása**

*** VAR2 * első paraméter**

*** VAR1 * második paraméter**

*** VAR3 * harmadik paraméter**

A programrészt lefordítva a következő „tárgy programot” kapnánk:

proc1 címe – adat

var1 címe – adat

var2 címe – adat

proc2 címe – adat

var2 címe – adat

var1 címe – adat

var3 címe – adat

Láthatóan előállt egy adatsorozat.

A már előzőleg leírt szabály szerint az tudhatjuk, hogy az első adat az az eljárás címe, a következő adat az eljárás első paraméterének címe.

De hol kezdődik a következő eljárás, illetve hogyan folytatódik a program?

Egyáltalán: hogyan lesz egy adatsorból program?

Erre a kérdésre a választ az ún. interpreter, vagy értelmező adja meg, amely itt forrásprogram szinten nem is látható.

Ha itt nem látható, akkor viszont egy helyen lehet csak: a mikroprogramban!

Az interpreter, ami három Assembler-utasítás gépi kódja, a már említett nullás lapon a $IN címen helyezkedik el, és a mikroprogramból kerül meghívásra; ott, ahol a mikroprogram visszatérne a hívójához.

Hogyan is működik ez az interpreter/értelmező?

Ennek a megértéséhez szükséges még annak a szekvenciának az ismerete, amivel áttérünk az Assembler programozási módból az interpretatív módra. Ez a bizonyos szekvencia a következő:

```
APM   *+8    * interpretatív nyelvre áttérés kezdő utasítása
      RTM    $C1
      APM    **
      BRI    **
      ADS    *+2     * áttérési szekvencia vége
```

A részletek mellőzésével a szekvencia biztosítja, hogy az ún. pszeudo utasításszámlálóban – $C1 – a kezdő eljárás belépési pontja – annak a címe – kerüljön tárolásra, és a belépési pont fő utasításszámlálóba is bekerüljön, azaz a végrehajtandó program innen kezdve nem más, mint az első eljárás.

Tehát: sikerült „meghívni” az első eljárást, és eltároltuk azt a helyet, ahová az eljárásból vissza kell majd térni.

Innentől kezdve egyszerű a dolog: a mikroprogram végrehajtása alatt a $C1 tartalmát annyiszor kell 2-vel – 2 byte egy szó – megnövelni, ahány paramétere az eljárásnak van. Ilyen módon minden eljárás a paraméterinek száma szerint módosítja a $C1 értékét. Ez biztosítja azt, hogy az $C1 mindig az eljárás utolsó paraméterének a címére mutasson. Az eljárás utolsó paraméterét követő címen pedig mit is találunk: a következő végrehajtandó eljárás belépési címét.

Amikor az eljárás befejezte a működését, akkor meghívja a nullás lapon tárolt értelmezőt – BRI $IN –, ami a következő 3 utasítás gépi kódja:

```
IM2   $C1
APM   **
BRI   **
```

Természetesen ezt a szekvenciát bele is lehet írni a mikroprogram forrásába, de a helytakarékosság miatt 3 utasítás helyett 1 használata elegáns és helytakarékos.

Ez az utasításszekvencia a következőket jelenti:

1. Az IM2 $C1 az első pszeudo utasításszámláló értékét – ami a következő végrehajtható utasításra mutat – 2-vel megnöveli. A 2-vel megnövelt érték tárolásra is kerül a megadott címen, de benne marad az A akkumulátorban. Ez az érték most nem más, mint a BRI ** utasítást követő cím, ami nem más, mint az utasítást követő első utasítás. Ez az utasítás pedig a következő eljárás belépési pontja.

2. Az APM ** utasítás hatására az A akkumulátorba lévő értéknek – amit most címnek tekint a műveletvégző – tartalma (akkumulátoron keresztüli indirekt címzés) került az akkumulátorba.
3. A BRI ** pedig egy feltétlen ugrás arra a címre, amit az A akkumulátor tartalmaz – ami most a következő eljárás belépési pontja, azaz a program végrehajtása innen folytatódik tovább.

Így tehát minden mechanizmus rendelkezésre áll ahhoz, hogy az adathalmaz programként értelmezhető – interpretálható – legyen. Természetesen ezeket a fenti „részleteket", illetve ennél még kicsivel többet is csak azoknak kellett ismerniük, akik ilyen eljárásokat, mikroprogramokat írtak.

Innentől kezdve már az Assembler programot is leírhatjuk – tudva azt, hogy az áttérési szekvenciát csak be kell másolni, valamint meg kell adni, hová kell majd a programot betölteni a memóriába.

Tehát a programunk Assembler formában, ha PROC1 és PROC2 az előre meghatározott eljárás:[2]

```
        PAG   xx      * elhelyezés helye
VAR1 RES   2      * 1. változó, 2 byte lefoglalása
VAR2 NB    0      * 2. változó
VAR3 NB    0      * 3. változó
STRT APM   *+8    * interpretatív nyelvre áttérés kezdő utasítása
        RTM   $C1
        APM   **
        BRI   **
        ADS   *+2     * áttérési szekvencia vége
*       PROC1         * első eljárás neve – az eljárás meghívása
*       VAR1 * első paraméter
*       VAR2 * második paraméter
*       PROC2         * második eljárás neve – az eljárás meghívása
*       VAR2 * első paraméter
*       VAR1 * második paraméter
```

[2] Vastagon az interpretatív program és változói. A program egyéb részeit – elhelyezés (PAG), programindulás (FIN STRT), áttérési szekvencia – csak a megfelelő helyre le kellett írni, és készen is volt a program. Természetesen a „főprogram" után még a szükséges eljárások forrásait is le kellett fordítani.

```
*       VAR3 * harmadik paraméter
FIN     STRT   * program indulási címe
```

Láthatóan működni tud az a logika, hogy elég tudni azt, hogy egy-egy eljárásnak mi a neve, milyen paramétereket milyen sorrendben igényel, és utána már csak a feladat logikájának megfelelő sorrendben kell ezeket az eljárásokat és azok paramétereit egymás után leírni.

Ezek az eljárások – a mikroprogramok – forrásnyelven álltak rendelkezésre, és a fordítás során kellett arról gondoskodni, hogy a megfelelő mikroprogram-részek lefordításra kerüljenek, valamint a memória megfelelő helyeire kerüljenek majd elhelyezésre.

Fentiek természetesen igazak a programozó által megírt saját eljárásokra is, de itt még van egy követelmény: az eljárásnak egy kötelező módon kell befejeződnie, hogy biztosított legyen a visszatérés a hívó programhoz.

Ez az eljárás-hívási lánc természetesen megszakítható, vissza lehet térni „normál” Assembly nyelvre, majd folyatható az eljárás hívási lánc.

A mikroprogram könyvtár forrásállománya úgy volt előkészítve, hogy az egyes részek végére egy olyan karakterkombináció – BELL, RC, LF (kocsi vissza, soremelés) – volt lyukasztva, aminek a hatására a fordító megállt, és várta a következő forráskódú lyukszalagot.

Egy nagyobb, több mikroprogram-részt – szekciót – használó program fordításához több tekercs lyukszalagot kellett a megfelelő sorrendben beolvasni.

Csak példaként: hogyan is nézett ki egy ilyen program-összeállítás (a fordító programot már előre betöltöttük a memóriába). Legyen tehát egy egyszerű kis program, amelynek van egy főprogramja, ami két mikroprogram szekciót használ. Ez a kis, egyszerű program fizikailag a következő részekre – különálló szalagállományokra – oszlott:

1. Főprogram
   a. Program elhelyezkedési címe
   b. Program utasításai
   c. BELL RC, LF
2. Az első mikroprogram szekció elhelyezési címe
   a. Program elhelyezési címe
   b. BELL, RC, LF
3. Az első mikroprogram szekció forrása

   A forrás végén már gyárilag ott van a BELL, RC, LF karaktersorozat, ami jelzi, hogy folytatás következik.
4. A második mikroprogram szekció elhelyezési címe

a. Program elhelyezési címe
b. BELL, RC, LF

5. Az második mikroprogram szekció forrása

   A forrás végén már gyárilag ott van a BELL, RC, LF karaktersorozat, ami jelzi, hogy folytatás következik.

6. A program indítási címének megadása

a. FIN indítási cím

A kis, egyszerű program 6 különálló lyukszalagot igényelt. Persze, minden szalagra rá kellett írni, mi is az, és mikor olvasandó be. Olyan kisebb szalagoknál, mint amik az elhelyezési címeket határozták meg, néha már nem is mindig írtunk feliratot, mert némi gyakorlás után az ASCII kódú szalagokat úgy olvastuk, mintha azok nyomtatva lennének.

A már említett mikroprogram könyvtár eljárásai a nullás lapon a .10-.3F címeken erre a célra fenntartott munkaterületet használják a végrehajtáshoz, így biztosított volt az, hogy a munkaterületükön kívüli tárterületekhez nem fognak hozzáférni, tehát – elvileg – mellékhatás-mentesek.

Viszont, ha a programozó saját eljárást írt, akkor figyelembe kellett vennie, hogy a standard mikroprogramok mely nullás lapon lévő tárolókat használják, mert azok a saját eljárás esetében nem használhatók.

Itt kell megjegyezni, hogy az előzőekben vázolt mechanizmus biztosítani tudja, hogy mikroprogram mikroprogramot hívjon, csak azt kell biztosítani, hogy a $C1 tartalma mentésre és visszaállításra kerüljön – erre a célra a nullás lapon dedikált helyek vannak fenntartva –, valamint hogy a nullás lapon használt tárolók értékeit a hívott mikroprogram meg ne változtassa.

Az interpretatív nyelven programozónak, aki esetleg egy mérnök volt, elég volt az előzőekben már ismertetett fő szabályokat tudnia és betartania. A program fordításával, összeállításával stb. kapcsolatos feladatokat nagy valószínűséggel az ilyen feladatokban jártas operátorok végezték már el.

Az az eszközkészlet, amely az interpretatív programot írók rendelkezésére állt, a következő eljárásokat foglalta magába:

ugrások

inkrementálások

logikai műveletek

áthelyezések

egész- és fixpontos formátumú számokra behívás

egész- és fixpontos formátumú számokra tárolás
egész- és fixpontos formátumú számokra műveletvégzés
egész típusú formátum és decimális ábrázolás közötti konverziók
lebegőpontos formátum és decimális ábrázolás közötti konverziók
decimális lebegőpontos szám konverzió lebegőpontossá
fix- és lebegőpontos formátumok közötti konverziók
matematikai függvények: sin, cos, arc tg, ln, e^x, négyzetgyökvonás

Input-output műveletek
közös minden i/o műveletekhez
programozott csatorna foglaltságának ellenőrzése
bináris kódban lyukasztott szalag beolvasása a memóriába, szűréssel
bináris kódban lyukasztott szalag beolvasása a memóriába, szűrés nélkül
bináris kódban lyukasztott szalag beolvasása a memóriába, RC- kocsi vissza - karakterig
4 bites bináris kódban lyukasztott szalag beolvasása a memóriába
ASCII kódban lyukszalag lyukasztás
8 bites bináris kódban lyukszalag lyukasztás
beolvasás az ASR 33 írógép klaviatúrájáról a memóriába
nyomtatás

2025. szeptember