

MACHINE CODE - ASSEMBLER

Z80

2.díl - RUTINY

Učebnice programování mikroprocesoru Z80

Ladislav Zajíček, 1987

O B S A H

Předmluva

KAPITOLA 1

(Ne)místné úvahy o programování 1

KAPITOLA 2

Systémy, ach, systémy, kde ta ludia berú? 7

KAPITOLA 3

Aritmetika 15

KAPITOLA 4

Generování pseudonáhodných čísel 37

KAPITOLA 5

Konverze 42

KAPITOLA 6

Komprese a expanze dat 56

KAPITOLA 7

Pole 72

KAPITOLA 8

Retězce 80

KAPITOLA 9

Třídění 88

KAPITOLA 10

Tabulky 96

KAPITOLA 11

Jiné způsoby organizace dat 101

KAPITOLA 12

Rutinní drábky 106

RUTINA 13

Interfacing 115

Závěr (2), Index (1), příloha (2 listy)

Předmluva

Následující stránky přímo navazují na předchozí učebnici základů programování mikroprocesoru Z80 MACHINE CODE - ASSEMBLER Z80. Oproti původnímu záměru neobsahují jen prostý výčet rutin se specifickými funkcemi. Při promyšlení skladby a obsahu díla jsem si stále jasněji uvědomoval nutnost pokračovat v didaktické formě podání prezentované látky. Protože - k čemu by novicům assembleru byl samotný soupis komentovaných rutin bez znalostí, bez nichž se nelze obejít při tvorbě vlastního programu? Jakákoli rutina v jakémkoli programu hraje v podstatě tu nejposlednější, i když nikoli zanedbatelnou roli. Tak jsem postupně dospěl k názoru, že pokud to, co máte v ruce, nebude návaznou učebnicí programování, "nepustím to do světa". Jak se záměr podařil, se dozvíte až od vás.

V učebnici jsou použity původní i převzaté rutiny. Postupem doby bylo mnoho rutin, vykonávajících nejčastěji používané funkce, dovedeno do optimálního tvaru, který by nemělo smysl vymýšlet znova (ani softwarové firmy to nedělají). Tyto rutiny jsou převzaty ze zahraničních pramenů. Zde jsem se setkal s tím, s čím se musí v tomto druhu literatury už tak trochu počítat - s tiskovými chybami. Abych vás jich zcela vyvaroval, všechny převzaté rutiny jsem napřed zapsal do editoru assembleru a pomocí testů jejich funkcí je v případě nutnosti upravil. Abych se přepisem rutin do učebnice nedopustil chyb plynoucích z překlepů, převedl jsem je přímo z editoru assembleru na formát slovního procesoru, v němž je napsán text celé učebnice. Původní rutiny pocházejí z vlastních fungujících programů, jsou tedy ověřeny praxí.

První kapitola se zabývá některými zásadami assemblerové magie v obecnějším kontextu. Další kapitola - o systémech - nebylo možno pominout, přestože systémy jsou tou nejspodnější a nejrozdílnější vrstvou výpočetní techniky, mikropočítače nevyjímaje. Každý ze systémů by si žádal zvláštní publikaci, proto je téma spíše nastíněno, než aby kandidovalo na všeobjímající orákulum.

Od třetí kapitoly by učebnici bylo možno pomyslně rozdělit na dvě části. První - rutinní (ve smyslu prezentace elementárního tématu s mírně sníženým stupněm záživnosti) - obsahuje rutiny aritmetické a konverzní, bez nichž žádný program existovat nemůže. Zvláštní pozornost je věnována také generování pseudonáhodných čísel. Druhá část - kreativnější - má bližší k rozličnosti alternativ řešení širších programových ploch.

Ze široké palety rutin první části si budete moci vybrat v prakticky jakýchkoli programových situacích, vyžadujících použití základní aritmetiky nebo konverzí. Tato část obsahuje průvodní slovo pouze tam, kde je nezbytné (základní principy aritmetických operací byly podrobně vysvětleny v předešlé učebnici). Tak, jako všechny rutiny učebnice, jsou i tyto komentovány, a pro větší srozumitelnost je k nim připojen i základní algoritmus.

Druhá část je (dobromyslným) útokem na čtenářovu představivost. Velmi podrobně se zabývá konstrukcí datových tabulek a manipulací s řetězci a poli. Detailně jsou podány základní způsoby prohledávání (binary search) a třídění dat (insertion sort, bubble sort, quick sort), vysvětlen je princip práce s binárním stromem (heap sort) a hashing spolu s bitovou mapou a bins. Vše doplňují příklady komprimací dat, jimiž jako spojovací nit procházejí úvahy o komprimovaném srovnávacím slovníku. Souběžným záměrem této části je strhnout čtenářovu imaginaci k vlastnímu zamýšlení se nad různorodostí přístupů k řešení jakéhokoli problému. Připojeny jsou obecné informace o obvodech pro přenos dat mezi porty a perifériemi.

K učebnici je jako pomůcka k ruce programátora přiložen soupis některých základních operací pro rychlou volbu elementárních rozhodovacích procesů ve vytvářeném programu.

Všechny, kdož se hodlají pustit do čtení dalších stránek, si dovoluji upozornit, že se neobejdou bez znalosti instruktážního souboru mikroprocesoru Z80 a některých základních assemblerových operací. Na druhou stranu je to ale jediný předpoklad zdárného studia. Oproti minulé učebnici, která byla zaměřena na osvětlení funkcí jednotlivých instrukcí, je nutné se do rutin této učebnice "včíst" za podpory připojených informací. I zde platí, že kdykoli se při studiu ocitnete v postavení dona Quijota, můžete denně mezi 14. a 18. hod. volat "assembly hot line" 5358183 (Praha).

-elzet-

okolností vystěhovali. Vylít s vaničkou i dítě se třeba anglicky, v doslovném překladu, řekne - rozlévat mléko po palubě. "Náhlé" vylévání dítěte v kontextu se čteným textem by Angličana přivedlo přinejmenším do rozpaků.

Z hlediska používaného jazyka však mezi literáty a programátory existuje jeden zásadní rozdíl - zatímco přímými adresáty literátů jsou lidé, přímým adresátem programátora je...inu, počítač. Programovací jazyk (alespoň zatím) nezná expresi, která by v počítači evokovala pocity, podle nichž by se pak zachoval...i když někdy tak soupeření člověka s počítačem může vypadat, že?

Assembler je jazyk, jímž můžeme řešit mnohé algoritmy, které by vyšším programovacím jazykem byly řešitelné obtížně či vůbec ne. Třeba provádění souběžných operací při načítání dat z pásky, jak je uvedeno v našem problému se čtením řady čísel. Z toho je patrné, že volíme vždy taková řešení, která odpovídají dispozicím toho kterého jazyka. Myslíme tedy v onom jazyku, do určité míry jsme (a měli bychom být) ovlivněni jeho logikou už ve fázi algoritmizování problému. Proto je tak důležitý vyvinutý cit pro jazyk, s nímž pracujeme. Podvolení se logice jazyka při mnohatýdenní příliš intenzivní stavbě programu může programátora dohnat až tak daleko, že bude mít krátkodobé problémy s procitnutím do logiky běžného lidského života. Pokud nenajde pro své zaujetí zlatou střední cestu, může na něm zanechat jisté průvodní znaky trvalejšího rázu. Ty se projevují např. snahou vnímat svět jako soustavu podprogramů, které postižený interpretuje "svou" logikou přenášených dat, což může přerůst v postupnou ztrátu společenské senzibility a identity. Ale abych nerozléval vaničku po palubě - to neuvádím proto, že by každý programátor musel nutně být potencionálním zákazníkem léčebného ústavu, ale proto, abych dokreslil, že sžití se s logikou jazyka je nezbytným předpokladem úspěšného programování. I že všeho s mírou. O to víc mějme na paměti, že

Konečnými adresáty našeho programu jsou jeho budoucí uživatelé

Na tuto, jednu ze základních vět programování, se dost často zapomíná. Programátora nezřídka zaujme jen samotné vyřešení základního problému. Program sám mu "uvnitř počítače" funguje bezvadně. Jenže nikdo kromě něj neví, jak a k čemu všemu jej používat. Programátor by nikdy neměl zapomenout na to, že uživateli jeho programů je v drtivé většině případů zcela jedno, jak program data interně "přemílá". Nejdůležitější z hlediska uživatele je vždy to, jak se data projevují navenek a jak s nimi jednoduše pracovat, tedy jak zdárně a efektivně komunikovat s funkcemi programu. Této komunikaci se v zemích, kde si lze počítač koupit za normální peníze v normálním obchodě, říká interfacing mezi programem a uživatelem. V podstatě jde o interaktivní sledování zobrazených dat na monitoru a volbu funkcí ovládacími prvky (tlačítka, myši apod.). Dnes je kvalita tohoto interfacingu jedním z nejdůležitějších předpokladů uplatnění programů v praxi, tedy i jejich prodejnosti.

Co to znamená pro vás? Přesně tolik, že budoucí komunikace vašeho programu s uživatelem je tím prvním, čím se při přemítání nad programem musíte zabývat. Rekněme třeba, že chcete vytvořit databázi:

Pokud byste ihned, bez rozmyslu sedli k počítači a začali tukat instrukce, program by zahynul, než by se začal rodit.

Pokud byste začali přemýšlet o tom, jak mají být data řazena a prohledávána uvnitř programu, a pak vytvářeli izolované zkušební rutiny, s velkou pravděpodobností byste je později museli dost měnit.

Pokud byste si rovnou začali kreslit vývojový diagram vnitřních funkcí databanky, leccos by se vám vyjasnilo, ale z takto sestavené konstrukce byste posléze museli částečně či zcela odstoupit.

Ze všeho uvedeného plyne jistá časová, zcela zbytečná ztráta. Přirozeně se vynořuje otázka:

Odkud začít s tvorbou assemblerového programu?

Od vlastní představivosti. To platí značnou měrou i pro tvorbu programů ve vyšších programovacích jazycích. V případě assembleru, který má svá specifika, to platí zcela beze zbytku! Při tvorbě větších programů to ani jinak nejde. Zůstaňme u naší databanky. Po pevném rozhodnutí pustit se do její tvorby počítač zcela ignorujte. Každou volnou chvíli věnujte přemítání nad tím, co všechno by program měl umět. A představujte si hned, jak

KAPITOLA 1

(Ne)místné úvahy o programování v assembleru

Velký úkol pro malou paměť

Představte si, že stojíte před takovýmto úkolem - na magnetofonovém pásku dostanete řadu jednoho miliónu náhodně vygenerovaných dvoubajtových čísel z intervalu 0-65535. Máte zjistit, zda v této řadě některá čísla z uvedeného intervalu chybí, a když ano, tak která. K dispozici máte běžný mikropočítač s pamětí 64K.

Jde o typickou ukázkou problému, před níž začínající programátor couvne se slovy: "To nejde, počítač by přece musel mít větší paměť!" Na otázku, jak velkou, odpoví bez váhání: "Minimálně jeden megabajt." Což je opět typická ukáзка, tentokrát svědčící o malé praxi v procvičování "algoritmických závitů". Jak tedy řešit výše uvedený úkol s tím, co máme k dispozici? Možná jste už na to přišli sami. Pokud ne, zkuste na chvíli odložit učebnici a řešení najít.

Ze zadání úkolu vyplývá, že musíme nějak zaregistrovat existenci každého čísla, zaznamenaného na pásku. A to ať už se v řadě čísel vyskytuje jednou, či vícekrát. Nakonec musíme umět zjistit, které z čísel v intervalu 0-65535 se v řadě neobjevilo ani jednou. Základní algoritmus vyplývá z tohoto nápadu - pro vyjádření každého čísla v uvedeném intervalu nepotřebujeme dva bajty, ale jeden, jediný bit! Každý bajt paměti tedy bude reprezentovat osm čísel. Tak např. číslo 1000 bude tisícím bit v řadě, počínající prvním bitem adresy, od níž bude řada bitů začínat. Před čtením čísel z pásky musí mít všech 65536 bitů hodnotu log.0. V případě, že z pásky bude přečteno číslo 1000, bude tisícímu bitu přidělena hodnota log.1. Tak po přečtení celé řady jednoho miliónu čísel budou všechny bity s hodnotou log.0 reprezentovat čísla, která na pásku chyběla. Nutný rozsah paměti bude pouhých $65536:8=8192$ bajtů, tedy 8K RAM. Plus pár set bajtů pro vlastní program, který bude číst z pásky, přidělovat hodnotu log.1 pořadovým bitům a výsledek nakonec vyhodnotí. A nezáleží už vůbec na tom, zda budeme z pásky číst jeden či deset miliónů čísel.

Určité alternativy řešení se nabízejí ještě pro čtení z pásky. Pokud by na něm byla čísla zaznamenaná v ucelených blocích, z nichž by se nám každý vešel do volné paměti, mohli bychom každý blok po jeho načtení do paměti monitorovat zvlášť, postupně. Potřebná část paměti by tu hrála funkci bufferu. Pokud by ovšem nahrávka obsahovala šňůru čísel, která by se nám do volné paměti nevešla, a neměli bychom šanci ji nějak "roztrhat", museli bychom čtecí rutinu rozšířit o část, která by během čtení dokázala každé čtené číslo určit a odpovídajícímu pořadovému bitu přidělit hodnotu log.1.

Jak jste si všimli, při hledání algoritmu jsme ani slůvkem nezaváděli o jedinou instrukci. Hledání dobrých a ještě lepších algoritmů je to hlavní, co programátora dělá Programátorem. Instrukce jakéhokoli jazyka jsou "jen" výrazivem, kterým do počítače zapisujeme budoucí děj našich algoritmů. Podobně spisovatel či dramatik se nestává Mistrem proto, že zná několik tisíců slov své mateřštiny - tu nakonec zná každý, kdo onou řečí mluví. Ale ne každý umí vystavět architekturu slovesného díla tak, aby mělo nosný oblouk, kompozici, sdělnou expresi základní myšlenky. Už jen tento krátký výčet požadavků na skutečné dílo dává tušit, že být literátem není snadné. A že napsat dobrou knihu neznámá jen tak na pár dní si sednout k psacímu stroji či slovnímu procesoru. A už vůbec autor takového díla nemůže zápasit s instrumentářem jazyka, který při psaní používá. Valnou měrou to, co platí pro literáta, platí i pro programátora.

Cit pro jazyk,

s nímž pracuje, musí mít literát i programátor velmi silně vyvinutý. Každý jazyk má svou logiku. Chceme-li se umět perfektně vyjadřovat tím kterým jazykem, musíme mít jeho logiku bezpodmínečně vstřebanou. Musíme myslet i cítit daným jazykem. Sítí se s jakýmkoli jazykem je záležitostí tvrdé práce v delším časovém období. Jeden programátor řekl: "Mně ten assembler už tak čouhá z bot, že když mám najednou programovat třeba v Pascalu, trvá mi jistou dobu, než se na něj dokážu přepnout." Tím jen potvrdil výše řečené. Potvrzují to i nezdařené pokusy literátů, kteří zkoušeli psát cizí řečí země, do níž se vlivem

to, co umí, je ovládáno uživatelem, a jak se to promítá na obrazovce. Sem tam si už lehce můžete představit, co která funkce přinese do struktury programu, třeba i velmi detailně. Ale zůstávejte stále jen v rámci představ. Pouze když vás napadne něco, o čem si myslíte, že je to dobrá myšlenka, kterou byste neměli zapomenout, poznamenejte si ji na papír. Toto období imaginací je nesmírně důležité. Je nutné se dopracovat k nějaké představě o vnější prezentaci dat a ovládání programových funkcí uživatelem. Tím si zároveň ujasňujete základní zadání, které budete následně algoritmovat. Bez zřetelného zadání se nikdy do žádného programu nepouštíte. Vystavovali byste se tak nebezpečí příliš častých změn jednotlivých částí programu v průběhu jeho tvorby. A to je nepříjemné (nadělá se při tom i spousta chyb), časově ztrátové a v neposlední řadě to nijak nepřidává na sebevědomí.

Tato první fáze bývá různě dlouhá, může trvat i více týdnů. Různí programátoři se tou dobou projevují různě. Někdo chodí jako ve snách, někdo dlouhé hodiny leží a volně medituje, jiný přechází po místnosti jak lev v kleci... Jak kdo.

Poté, co vaše představy nabývají zřetelnějších kontur, zamyslete se už konkrétně, s tužkou v ruce, nad uživatelským ovládáním programu. Proč? Protože ve vztahu k budoucímu programu toto ovládání tvoří jeho nejzákladnější rozhodovací body. Z nich se budou skoro všechny programové funkce odvíjet. Orientací v hierarchii tohoto ovládání se vám odvine základní struktura programu. V jejím nejvyšším patře budou nejvyšší rozhodovací body, z nichž bude program odkakovat do subrutin ve stále nižších patrech - v nich bývají ty "nejotročtější" subrutiny, z nichž mnohé jsou "v jednom kole" (např. rutiny pro výpis znaků na obrazovku apod.). Když to váš počítač umožňuje, zahrňte do struktury i obslužné rutiny přerušeni (např. test klávesnice).

Pokud jde o ovládání programu uživatelem, je nutné, aby mělo svou logiku. Nesmí mít žádná nesmyslná vymezení, musí být naprosto jednoznačné. Nejfrekventovanější funkce musejí mít nejjednodušší ovládání, tedy žádné mnohonásobné šifty, či prostupování několika menu. Tím jsem se dotkl dvou hlavních typů ovládání - 1/ funkčními tlačítky, 2/ volbou funkcí z menu (anglicky menu-driven). První typ má výhody v rychlosti volby. Jeho dost podstatnou nevýhodou je nutnost pamatovat si, co které tlačítko kdy znamená. Takové ovládání je snesitelné pouze u programů, s nimiž jsme prakticky v denním styku. Jakékoli delší přerušeni kontaktu přináší nutnost znovu otevřít mnohastránkový manuál programu a vše si připomenout. Druhý, dnes mnohem užívanější typ ovládání trochu zdržuje díky neustálému kurzorování při volbě funkcí. Je však milejší proto, že k ovládání takového programu nepotřebujeme nijak zvlášť studovat manuál a pamatovat si spoustu informací, které stejně spolehlivě zapomeneme. U často používaných programů (slovní procesory, databanky) je někdy vhodné volit kombinaci obou typů ovládání, ale tak, aby měly jasnou, důvtipnou logiku a bylo nutno pamatovat si co nejméně.

Vnější prezentace dat je spolu s ovládáním programu jedním z největších kamenů úrazu. Základním zákonem je, že uživatel musí mít v kterémkoli momentu přístupné všechny informace, nezbytné pro orientaci a komfort obsluhy. Musíme si tedy vždy odpovědět na otázku, jak a kdy mají být na obrazovku promítnuta která data. Např. když uživatel v databance upravuje formát výpisu dat na obrazovce, musí vidět nejen novou pozici přemísťované položky, ale i všech ostatních, aby se ve vzhledu celého výpisu mohl plně a průběžně orientovat. Stejně jednoduché musí být zakládání nových položek či rušení starých. Obecně důležitá je snadnost provedení jakýchkoli oprav. Při volbě funkcí musí o nich být uživatel informován. Třeba když chce zapsat data na pásek, musí mu být jasné, že je v módu zápisu na magnetofon (po volbě zápisu tedy nesmí všechno zmizet z obrazovky, protože uživatel by tak okamžitě ztratil jistotu, zda provedl volbu správně). Ozřejnění toho, jak a kdy se která data budou objevovat na obrazovce, nám opět posune krystalizaci struktury programu kupředu. Nebudeme ji tak chytat ze vzduchu, budeme se mít o co opřít, z čeho vyjít.

Jak pokračovat?

Po vyjasnění vnější prezentace dat a způsobu ovládání si můžete nakreslit základní strukturu programu (málokdo je schopen nosit ji celou v hlavě). Souběžně s tím promýšlejte algoritmování problémů, na něž narazíte. Prakticky neexistuje program, který by v sobě neměl editor. Jeho základní rutiny by měly být prvními, které na počítači vytvoříte. K editoru postupně připojujete rutiny dalších funkcí. Proč začít editorem? Jím vyřešíte přenos dat z klávesnice do programu i přenos dat do obrazové paměti - z větší části tak budete moci správnost

průběžně tvořených programových funkcí a jejich kombinací ověřovat prostým pohledem na obrazovku. K ověřování funkcí rutin jejich krokováním na monitoru paměti se uchylujte jen výjimečně - tento způsob práce stojí zbytečně moc času.

Uvedený postup jde proti hlásaným pravidlům programování, která tvrdí, že při tvorbě programu je vždy nutno postupovat shora dolů. Praxe ukazuje, že zvláště assembler si žádá přístup poněkud odlišný. Proto - kdykoli se vám to v kontextu s vývojem programu zdá být vhodné - neváhejte sáhnout ke tvorbě jakékoli rutiny z nižších pater programové struktury, přestože v těch vyšších vám toho ještě spousta chybí. Ovšem hlavním předpokladem úspěchu tohoto postupu je - mít jasno v globální struktuře programu. Opět zdůrazňuji, že bez toho je zbytečně vůbec za počítač usadat.

Rutiny pro ovládání a prezentaci dat programu často tvoří jeho podstatnější část i z hlediska jeho rozsahu. Před časem jsem dělal jednoduchý program, který si hrál se zvukovým výstupem počítače. Zpracování základního algoritmu mělo ve výsledku asi 60 bajtů, rutiny ovládání programu cca 100 bajtů a na celkem střídmost vnější prezentaci dat jsem spotřeboval kolem 800 bajtů. Tento extrémní poměr byl pochopitelně dán tím, že program zpracovával jedinou základní myšlenku. Kdyby jich bylo implementováno víc, uvedený poměr by se adekvátně zmenšil.

Jak nezbloudit

I když dodržíte výše uvedená pravidla, při tvorbě programu v počítači na vás čeká řada úskalí. Některým se vyhnout nelze, jsou v podstatě přímo úměrná schopnostem a době praxe programátora. Něčemu se ale vyhnout lze. O tom blíže.

Začínající programátor mívá tendenci optimalizovat své rutiny jejich zkracováním ("zamotáváním"), případně již v prvních řázech zápisu programu do počítače slučuje sobě navzájem podobné subrutiny (jejich redundantní, tedy nadbytečné kódy) do jedné. Ani jedno ani druhé není přímo chybou. Ale takové úpravy by měly přijít ke slovu až zcela na závěr, kdy máte celý program ve tvaru fungujícího polotovaru. Důvod je jednoduchý. Všechny subrutiny, které tvoříte při rozvíjení nějakých programových funkcí, si vůbec nemusejí zachovat svůj prvotní tvar po celou dobu práce na programu. Naopak - jak už mnozí z prvních pokusů víte - mnoho subrutin během tvorby často i podstatně předěláváte. Když je budete slučovat hned od počátku, dostanete se dříve či později do situace, kdy budete chtít, aby vám takováto univerzální subrutina v některém případě dělala něco "trošku jiného". Pro ten případ do ní tedy zavedete umělý test - např. ve volající rutině nějaký bit jednoho registru či adresy nastavíte na log.1 (třeba SET 5, (IX+10)). Tento bit pak budete testovat v subrutině (BIT 5, (IX+10)). Když bude nastaven, udělá subrutina jedno, když ne, udělá něco jiného. Tím to ale nekončí. Nastavený bit musíte po skončení akce zase někde vynulovat (RES 5, (IX+10)). Tak si přiděláváte spoustu starostí. Když je taková "univerzální" subrutina po nějaké době protkána umělými testy natolik, že jen-tak-tak drží pohromadě, může vás další požadavek na zavedení dalšího podobného testu nakonec postavit do velmi obtížně řešitelné situace. Čeká vás složité rozmotávání programu se zaváděním nových volání nově tvořených subrutin při současném odstraňování umělých testů. Udržet si orientaci v programu při takovéto činnosti je dost náročné (a upřímně řečeno, člověk by s tím v takové chvíli nejraději praštil).

Příslovečný tenký led pod sebou rozehříváme zakládáním rozhodovacích bodů v rutinách nižších pater struktury programu. Proto dbejte vždy na to, aby hustota rozhodovacích bodů byla největší u špičky struktury a směrem dolů prudce řídla. Mějte stále na paměti, že nedodržením těchto pravidel se po dvou měsících driny skoro jistě dostanete do situace, kdy budete muset přepracovávat značnou část struktury programu. Je svým způsobem zvláštní, až poněkud metafyzické, že ať už chcete či ne, assemblerový program si na vás určitou svou strukturu nakonec vynutí. Pokud tomuto nucení nebudete schopni porozumět a vyhovět, rozsáhlejší assemblerový program nikdy nedoděláte. I když nikdy nic neplatí absolutně, v tomto případě jen velmi vzácné výjimky potvrzují pravidlo.

Jednou z velmi důležitých zásad interakce programátora s jím vytvářeným programem je nutnost kontinuální, ničím nepřerušované tvorby. Pro malé, pomocné rutiny, které nám zabere jen pár dní,

to pochopitelně neplatí. Ale v případě programu, který má ve výsledné podobě třeba 10K, to platí plně. Tvorba rozsáhlejšího programu klade nesmírné nároky na krátkodobou paměť programátora. Obsah této paměti se rychle mění podle toho, nad kterými funkcemi programu zrovna pracuje. Vedle toho musí mít v dlouhodobější paměti uloženy všechny základní informace o již vytvořených částech a funkcích programu, ale i své představy, k jejichž budoucí realizaci směřuje. Mezi krátko- a dlouhodobou pamětí musí proto existovat perfektní souhra. Je zřejmé, že jakékoli (třeba i jednodenní) přerušení práce tuto interakci poruší. Vrací-li se programátor ke tvorbě po delším přerušení, musí začínat prakticky od začátku. Z dlouhodobé paměti všechno postupně a nezadržitelně zmizí.

Z hlediska krátkodobé paměti je dobré, když každý pracovní den končí uzavřením jedné mikrokapitoly tvorby, aby tuto paměť bylo možno nechat vydechnout (vyprázdnit) a další den ji zatížit něčím novým. Je třeba si uvědomit, že program je de facto soustavou dějů, které jsou aktivovány podle scénáře, jehož autorem je programátor. Tento scénář vzniká ve zcela prvním období hledání kontur programu a postupně se dějově krystalizuje. Zatímco výstupem filmového či divadelního scénáře je lineární děj, programový pracuje s členitou strukturou, jejíž jednotlivé části mohou kdykoli vstoupit do děje, aniž by se jeho logika jakkoli narušila. A to jsme stále ještě v oblasti sériové logiky (jeden řídící mikroprocesor). U logiky paralelní (více mikroprocesorů pracujících současně) je programová scénaristika o poznání náročnější.

Pochopitelně je třeba počítat s tím, že i když se budete programu věnovat denně, všechny důležité informace o programu se vám v hlavě udržet nepodaří. Proto průběžně komentujte důležité body programu a na několik zvláštních papírů si poznamenávejte zásadní informace o stavbě programu a jeho funkcích. Bytší právě dnes je vám vše jasné a nevíte, proč byste si měli něco poznamenávat, věřte, že za měsíc vám to už tak jasné nebude. Tak, kdykoli na vás dolehne dezorientace, mrkněte na papír, jak kudy která data jdou, a vše se ozřejmí. Bez toho byste ztratili hodně času i nervů zbytečným ohledáváním a luštěním programu.

Uvedené náročné požadavky nasvědčují tomu, že profesionální programování je skutečnou reholí. Proto sami zvažte, nakolik se mu budete moci věnovat. Udělejte si takový svůj scénář vztahu k assembleru. Nemělo by smysl stavět si nereálné cíle, před jejichž dosažením čeká nepřemožitelné období roztrpčení. Na druhou stranu i amatérský programátor může vytvořit zajímavé kratší programy nebo pomocné rutiny, které najdou své uplatnění. Znalost assembleru v mnohém ulehčí život programátorům pracujícím s vyššími programovacími jazyky. Nejen tím, že budou umět vytvořit některé části programu ve strojovém kódu, ale i tím, že mohou poznat a využívat operační systém svého počítače i jeho periférií. Typickým příkladem u mikropočítačů jsou volání různých rutin ROMky.

Jak finišovat

Především rozvázně. Když dovedete svůj program do fáze, v níž vám všechno funguje, nahraďte si někde jeho kopii pro případné budoucí úpravy, resp. tvorbu jeho nových verzí. Protože jste dodrželi pravidlo o neslučování subrutin a neprovádění průběžných optimalizací, je program notně obtěžkán redundantními (nadbytečnými) kódy. Teď prakticky zapomeňte na logiku vašeho programu a pusťte se do hledání rutin, které jsou si buď zcela či z větší části podobné. Z těch prvních nechte v programu jen jednu, kterou budete volat z různých míst (někde tedy budete muset změnit adresu volání). Ty druhé můžete rozdělit do dvou či více "schodů" jednoho patra. Např. je-li první část rutiny A stejná jako rutina X, rozdělíme rutinu A na dvě části - A a B. A bude totéž, co X. Z místa volání původní rutiny A budeme nově rozdělenou rutinu volat dvakrát (CALL A; CALL B). Z místa volání původní rutiny X budeme volat subrutinu A (CALL A). Rutinu X zrušíme. Často se stává, že po předchozím několikaměsíčním souboji s programem nám na mnoha místech zstanou zbytečné testy a různé nešikovnosti, kterým se teď sami "zasmějeme". Odstraňujeme i drobnosti, které unikly naší pozornosti, dříve koncentrované hlavně na funkčnost rutin. Narazíme třeba na to, že po instrukci XOR A je o kus dál dvoubajtová instrukce LD B,0. Nahradíme ji jednobajtovou LD B,A. Nebo zjistíme, že po nějakém odečítání je indikátor CY vždy nastaven a my jej poté nastavujeme ještě jednou instrukcí SCF (zrušíme ji). Apod. I když se to nezdá, můžeme tak původní polotovar zkrátit třeba i o polovinu.

Takto optimalizovaný program budete moci nějak výrazněji modifikovat jen velmi těžko. Pro ten účel máte uschovány kopie polotovaru. Jak čas běží, většina důležitých informací se vám vykourí z hlavy. Proto onu kopii mějte v těch nejdůležitějších místech komentovanou (!!!) tak, abyste se kdykoli v budoucnosti v programu vyznali. Dobře provedenou vlastní dokumentaci oceníte vždy, když budete dělat další větší programy. Mnohé z rutin, které jste použili v programech předchozích, najdou využití i ve vaší další tvorbě. To se týká především rutin vstupu a výstupu pro práci s klávesnicí, obrazovkou, zvukem, záznamovými jednotkami a tiskárnami. Ale i mnoha dalších - editačních, kompresních atd. Čím více programů vytvoříte, o to lehčeji se vám bude dělat na dalších. Tomu chce napomoci i obsah této učebnice.

KAPITOLA 2

Systémy, ach, systémy, kde ťa ľudia berú?

Každý začínajúci programátor komunikuje se systémom počítača tak trochu na úrovni domluvy s kozou na lede. Pokus o standardizáciu mikropočítačových systémov pod zkratkou MSX zostal oslyšen. A tak čo počítač, to iný systém. Určitý pokrok je vidieť až u osobných počítačov - u veľkej časti jejich produkcie sa ako standard uchytil systém IBM PC. O určité, i keď trochu košatější standardizácii môžeme hovoriť i u operačných systémov typu CP/M, MS DOS apod. se všetmi jejich verziami. Väčšina mikropočítačov (tzv. domácich) má k jakémukoli standardu veľmi ďaleko. U nás však budú používané ešte mnoho let, a tak nezbýva, než je mať na vedomí i v budúcnosti.

Nad systémovým gulášom sa vznáša jedna otázka - máme ve svých programech využívať rutiny operačného systému, či nikoli? Odpoveď je jednoznačne nie. Pokiaľ dobrý znalec systému jedného počítača tieto rutiny ve svojom programe bohaté využije, bude stáť pred složitým problémom, keď bude chcieť previesť svoje dielo na iný počítač (byť se stejným mikroprocesorem). Jistě každý z vás dobře ví, že seznámení se se systémem počítače, který máte doma, vám dává dost zabrat. A nikdy nikdo nemůže prohlásit, že takový systém zná dokonale. Bez jeho popisu a komentovaného výpisu jeho rutin nedá ani ránu. Různé systémy jsou si sice rámcově hodně podobné. Jenže pro převod programu bohatě využívajícího systémových rutin to znamená seznámit se dost dobře s každým dalším systémem každého počítače, pro nějž budeme program přepracovávat.

Z bludiště systémů vedou různé cesty. Zde záleží jen na tom, na co si kdo troufá. Programátoři, schopní zapamatovat si telefonní seznam, mohou pochopitelně této vlastnosti využít. Ale my, kteří máme co dělat s tím, abychom ráno nevycházeli do práce v bačkorách, zvolíme adekvátní kompromis. Nejpřímější cestou je jakýkoli systém takřka zcela ignorovat. Pochopitelně kromě několika základních nutností, kterým se vyhnout nelze. To se týká adresování bodů obrazové paměti ve vztahu k jejím výsledné pozici na obrazovce a způsobu adresování důležitých vstupních a výstupních portů (klávesnice, magnetofon, zvuk, tiskárna apod.). Tak při převodu programu budeme muset přepracovat jen rutiny vstupu a výstupu a komunikace s obrazovou pamětí. S touto budoucí potřebou už při tvorbě programu počítáme a tyto rutiny tvoříme "izolovaně" tak, aby jejich úprava byla co nejjednodušší, abychom kvůli nim nemuseli předělávat cokoli jiného. Použijeme-li některé rutiny operačního systému, budeme se muset seznámit s jejich řešením na jiném počítači (tedy jak přenášejí data, jak které registry mikroprocesoru používají, ev. jak mění systémové proměnné). Samozřejmě, že čím víc využijeme operačního systému, tím kratší bude výsledný program. Ale tím víc práce můžeme mít s jeho převodem. Specialisté, kteří chtějí mít program co nejkratší, dokonce z rutin systému vybírají (porovnáváním řetězců) "vytržené" sledy instrukcí, které volají ze svého programu, aby tak opět ušetřili několik bajtů programu. Tomu se u potencionálně přenositelných programů samozřejmě vyhneme.

Otázka převodu programů na jiné počítače není u nás (ale ani ve světě) nikterak scestná. Vezměme si naši situaci ve druhé polovině roku 87. Vedle nejrozšířenějších Specter vyhřívají naše domácnosti Sharpy MZ-821, Atari 800XL, Sordy M5, Schneidery a Amstrady a Commodory... Hlášen je dovoz Atari 130XE a určitě časem přibude i leccos dalšího. A nesmíme zapomenout ani na to, že - oproti Tesle a spol. - JZD Slušovice nespí. Objeví-li se u nás konečně někdy nějaká softwarová "firmačka-družstvíčko z boží milosti", bude muset daný stav respektovat. A bude přirozeně nutit programátory, aby své programy převedli na uvedenou řadu počítačů. A bude-li programátor natolik inteligentní a bude s tím při tvorbě programů předem počítat, bude mít na delší dobu postaráno o práci i o slávu.

Jak z výčtu značek osmibitových mikropočítačů vysvítá, vedle mikroprocesoru Z80 se tu konkurenčně objevuje ještě Motorola 6502. Převod programu z jednoho mikroprocesoru na druhý je samozřejmě o něco komplikovanější. Zvláště mezi dvěma uvedenými. Zatímco Z80 (a třeba i 8080) patří k mikroprocesorům registrově orientovaným, 6502 je orientována pamětově.

Jak se do systému volá...

tak se ze systému ozývá. I analogie s hustotou lesa je tu namístě. Proto se učebnice nebude (a ani nemůže) zabývat jednotlivými operačními systémy různých počítačů. Povšimneme si obecně jen toho, bez čeho se zásadně neobejdeme. Při některých programových demonstracích bude použito pravidel systému nejrozšířenějšího Spectra. Nechtějí ti z vás, kteří mají jiný počítač, nepoklesávají v kolenou. Části rutin, které budou obsahovat systémovou komunikaci, můžete velmi jednoduše převést na svůj počítač, protože věc je svou podstatou triviální. Ale aby bylo vůbec co převádět, musí být něco demonstrováno, není-liž pravda?

1/ Klávesnice

je vždy sestavena do matice, jejímiž elementy jsou jednotlivá tlačítka. Třeba Commodore 16 má matici 8 x 8. U Spectra je matice 8 x 5 sestavena takto:

bity:	0	1	2	3	4	4	3	2	1	0	
3. linka											4. linka
2. "		Q	W	E	R	T		Y			5. "
1. "											6. "
0. "											7. "

Chceme-li zjistit, jaké tlačítko je stisknuto, adresujeme port tak, že z vyšších osmi linek adresové sběrnice nastavíme na log. 0 ty, které se váží k testované pětici tlačítek dle schématu. Samotný port klávesnice má adresu FEH. Získanou informaci ještě zpracujeme (při čtení z portu bude stisknuté tlačítko na odpovídající lince datové sběrnice reprezentováno log.0). Tak např. chceme-li zjistit, zda je stisknuto tlačítko E, budeme postupovat třeba takto:

```
LD B,11111011B ;Nastavení 2. linky na log.0
LD C,FEH        ;Port FEH
IN A,(C)         ;Čtení z portu
RRA              ;3 posuny bitu 2 do CY
RRA
RRA
JR NC,AND        ;Když je CY=0, tlač.E stisknuto
...              ;Při CY=1 ke stisku nedošlo
```

Když budeme chtít zjistit, zda bylo stisknuto jakékoli tlačítko třeba v peticích 3 a 4, můžeme použít tuto rutinu:

```
LD A,11100111B ;Nastavení 3. a 4. linky na log.0
IN A,(FEH)      ;Čtení z portu
CPL             ;Komplementace obsahu reg.A
AND 1FH        ;Vynulování vyšších 3 bitů a test
JR NZ,AND       ;obsahu reg.A. Když Z=0, stisk
...             ;Když A=0 (Z=1), nic nestisknuto
```

U jiného počítače budeme postupovat obdobně. Musíme znát jen číslo portu klávesnice, rozmístění tlačítek v matici a log. hodnotu, jakou se stisknuté tlačítko projeví na datové sběrnici. (Pokud snad někdo váhá nad tím, co znamenají písmena B a H, pak jsou to symboly pro označení binárního a hexadekadického čísla.)

Malá připomínka - pokud systém přerušení nevelí jinak, v průběhu přenosu dat přes vstupní/výstupní porty by maskovatelné přerušení mělo být zablokováno (instrukcí DI). A poté včas uvolněno (EI).

Instrukce AND 1FH ve druhé rutině maskuje obsah reg.A tak, aby před testem na nulu byly vynulovány přebytečné 3 vyšší bity (tlačítek je pět) a ponechána původní hodnota nižších pěti bitů.

Uvedené testy zjišťují pouze to, zda jsou fyzicky stisknuta nějaká tlačítka. Typické využití takových testů je ve hrách. Pokud však potřebujeme zjistit ASCII kód stisknutého tlačítka, bude lepší, když využijeme rutin testu klávesnice operačního systému. Někdy stačí jen test patřičné systémové proměnné, která vždy obsahuje ASCII kód právě stisknutého tlačítka (není-li stisknuto žádné, bývá tam nula nebo FFH). V jiné proměnné je uložen kód poslední stisknutého tlačítka. V proměnných jsou některé bity využity pro prezentaci módu klávesnice (např. pro mód grafický či pro avizování zápisu klíčového slova v Basicu) a pro vkládání velkých nebo malých písmen. To opět bývá u různých počítačů různé. Voláním zvláštních rutin u některých počítačů zjišťujeme, zda jsou stisknuta některá speciální tlačítka nebo jejich kombinace (ESC, CTRL apod.). U Spectra je to BREAK.

V mikropočítačích probíhá test klávesnice v pravidelných časových intervalech (1/50 vteřiny), nejčastěji v módu přerušování IM 1. V našem programu si můžeme s testem klávesnice všelijak hrát - třeba provádět test jen z některých jeho míst, z nichž budeme soustavu rutin klávesnice volat. Pro zdar toho ovšem musíme zablokovat provádění vnitřního přerušování instrukcí DI, aby se nám vnitřní test operačního systému zase nerozběhl. Získáme tím i menší zrychlení běhu programu. Někdy je to výhodné. Jindy se nám může hodit vlastní obslužná rutina (u Spectra pomocí IM 2), z níž budeme testovací rutiny volat podobně, jako to dělá sám počítač. Do obslužné rutiny přerušování ale budeme moci zařadit další funkce, které v rutinách operačního systému nejsou (a do něj bychom je těžko zařazovali, zvláště je-li uložen v ROMce). V některých částech programu bývá výhodnější použít rutiny pro testy stisků jen jednoho určitého tlačítka nebo pro zjištění, zda je vůbec nějaké tlačítko stisknuto. V tom případě neskačeme do dlouhých testovacích rutin systému, ale na patřičné programové místo uložíme rutinu, podobnou výše uvedeným. V programu můžeme mít např. subroutine:

COKOLI	XOR A	: Vynulování reg.A
PETICE	IN A, (FEH)	: Data z portu FEH do reg.A
	CPL	: Komplementace obsahu reg.A
	AND 1FH	: Maskování
	RET	: Návrat

Při volání adresy COKOLI zjišťujeme, zda bylo stisknuto jakékoli tlačítko. Když ano, vracíme se ze subrutiny s nenulovým obsahem reg.A a Z=0 (a naopak). Před voláním adresy PETICE registrem A adresujeme patřičnou linku, jak uvedeno výše. Je-li po návratu ze subrutiny PETICE reg.A nenulový, bylo některé z testovaných tlačítek stisknuto. Následným testem (rotací, maskováním) můžeme pak zjistit, jaké. Tento způsob testování klávesnice je velmi jednoduchý a hlavně rychlý. Pokud někomu není něco jasné ve věci adresování portů instrukcemi OUT a IN, ať se vrátí ke kapitole 0 interfacingu v minulém učebnici.

2/ Zvuk a barva borderu

U mikropočítačů, které mají speciální integrovaný obvod pro generování zvuku, můžeme signál tvarovat a přidávat k němu šum. Jak aktivovat a formovat všechny tyto složky, je třeba zjistit z technického popisu. Mikropočítače jako Spectrum uvedeny obvod nemají, zvuk vzniká prostým rychlým střídáním hodnot log.0 a log.1 na zvukovém výstupu. Tento výstup je opět ovládán prostřednictvím portu. U Spectra je to bit 5 výstupního portu PEH. Střídání log.1 a log.0 výstupu vytvoříme např. takto:

ZVUK	DI	: Blokování mask.přerušeni
	LD DE, AAAAH	: Inicial.doby trvání tónu
	LD C, 88H	: " " " páčky pulsu
	LD A, 7	: " " " pro bílou barvu borderu
HRANA	LD B, C	: " " " čítače pro CEKANI
	XOR 00100000B	: Bit 5 střídavě log.1 a log.0
	OUT (FEH), A	: Výstup na port FER
CEKANI	DJNZ CEKANI	: Prodléva pro páčku pulsu
	DEC E	: Zmenšení čítače doby trvání o 1
	JR NZ, HRANA	: Je E=0? Když NE, na adr.HRANA
	DEC D	: Je D=0?
	JR NZ, HRANA	: Když NE, pokračuj od adr.HRANA
	EI	: Uvolnění mask.přerušeni
	RET	: DE=0, návrat

V této subrutině je nepatrná časová disproporce, která v běžném užítí (např. "pípnutí" po stisku tlačítka) nevadí. Pokud by vadila, časový průběh upravíme. Nesrovnalost vzniká tehdy, když je reg.E nulový - doba trvání pátky taktu se prodlouží ještě o test obsahu reg.D. Provedeme si časovou analýzu kritického úseku:

```

JR NZ,HRANA      ;7 při E=0, jinak 12 (rozdíl 5)
DEC D            ;4
JR NZ,HRANA      ;12

```

Celkový rozdíl je $-5+4+12=11$ hodinových taktů. Vyrovnání rozdílu můžeme provést třeba takto:

11. $\partial^2 \mathbf{E} = 0$:

2) při nenulovém obsahu reg.E:

	JR Z, REGD	12
	JR Z, REGD	1
	JR HRANA	1
REGD	DEC D	4
	JP NZ, HRANA	10

	JR Z, REGD	7
	JR Z, REGD	7
	JR HRANA	12
REGD	DEC D	-
	JP NZ, HRANA	-

V obou případech je součet hodinových taktů 26. Vyrovnání bylo dosaženo vložení dvou "zbytečných" instrukcí JR Z,REGD a změnou posledního relativního skoku na skok s absolutní adresou. Rutina se prodloužila o 5 bajtů. Vůči původní nejdelší době trvání jedné poloviny zvukového pulsu (tj. při zahrnutí testu obsahu reg.D) je nový průběh delší jen o 3 hodinové takty.

Jednotlivé linky portů jsou využívány pro různé funkce. Výstupní port FEH Spectra kromě zvukového výstupu na lince 5 má na třech spodních linkách výstup barvy borderu obrazovky. Jednou z osmi možných kombinací tří bitů (bity 0, 1 a 2) nastavíme barvu borderu v rozsahu od 0 (černá) do 7 (bílá). Obdobně je tomu u všech mikropočítačů, které pracují s borderem. Instrukce XOR 00100000B v subrutině ŽVUK ponechává bity 0-4, 6 a 7 reg.A původní hodnotu (tedy i nastavená bílá barva borderu zůstane nezměněna) a přepíná pouze hodnotu bitu 5 pro zvukový výstup. Díky tomu, že výstupy pro ovládání zvuku i borderu jsou na jednom portu, můžeme rozličným algoritmováním výstupního obsahu reg.A vytvářet zajímavé efektní kombinace obou složek.

Když jsem měl možnost seznámit se z Amstradem CPC 464, vadilo mi, že při práci s magnetofonem neběhají v borderu vodorovné čáry, signalizující průběh přenosu dat jako u Spectra. Stačilo krátké záhlédnutí do firmwaru počítače, abych zjistil, že na určitém portu jsou patřičné 3 linky, jimiž mohou border Amstradu ovládat přímo, stejně jako u Spectra. Krátkým experimentem jsem si věc ověřil. To uvádím proto, že i když zde uváděné informace o práci se systémem počítače vycházejí z uspořádání Spectra, jsou svou podstatou univerzální.

3/ Obrazová paměť

Při práci s touto pamětí musíme brát ohled především na její rozlišovací schopnost a způsob adresování. Rozlišovací schopnost bývá různá. U starších mikropočítačů byla typickou bodovou sítí dlouhá ta, kterou má Spectrum - 256 bodů vodorovně a 192 svisle. Postupem doby se hustota zvyšovala především horizontálně. Dnes dosahuje až 640 bodů. To je dáno snahou dostat se k základnímu počtu 80 znaků typu pica na řádce dnešních běžných tiskáren. Každý znak má šířku 8 bitů, tedy 1 bajt. $640:8=80$ bajtů, zatímco u Spectra a spol. je to $256:8=32$ bajtů horizontálně. Tento nedostatek je často nahrazován zdvojnásobením počtu znaků na řádce, kdy 1 znak má šířku pouhých 4 bitů. Tím se však čitelnost zobrazovaného textu dostává na hranici snesitelnosti.

Způsob adresování obrazové paměti bývá na první pohled trochu komplikovaný, vždy však má svou logiku. U mikropočítačů jako Spectrum nebo SORD M5 je obrazovka rozdělena na 3 třetiny. První obrazová adresa Spectra je 4000H. Na ní je prvních osm horizontálních bodů. Až do adresy 4015H vše pokračuje normálně v jedné (nulté obrazové) horizontální lince. Adresa 4016H však začíná pod touto linkou, ale na sedmé, další je 15., pak 23., 31... až 55. linka. Po ní následuje adresace 1., 8., 16... až 57. linky. Poté 2. až 58... a nakonec 7. až 63. grafické linky. To vše dohromady čítá 32 bajtů na každé ze 64 grafických linek, tj. celkem 2048 bajtů, což jsou 2K na každou třetinu obrazovky. Logika adresování je pro každou třetinu stejná. Rozdíl je jen v tom, že 2. třetina začíná na adrese 4800H a 3. třetina na 5000H. Jeden znakový řádek je tvořen osmi sousedícími grafickými linkami. Tak každá třetina obsahuje 8 znakových řádek. Adresace levého horního rohu obrazovky vypadá takto:

Adresa 4000H	-----	8 bitů sloupce 0, linky 0
" " +1*256	-----	" 1
" " +2*256	-----	" 2
" " +3*256	-----	" 3
" " +4*256	-----	" 4
" " +5*256	-----	" 5
" " +6*256	-----	" 6
" " +7*256	-----	" 7

V bodové síti je inverzní zobrazení písmene A. Adresování obrazové paměti ve strojovém kódu má jednoduchou logiku (adresa je samozřejmě dvoubajtová):

bity 0-5	sloupec 0-15
" 6-8	znakový řádek 0-7 v každé třetině
" 7-9	grafická linka 0-7 v každém znakovém řádku
" 10-11	třetina 0-2
" 12-15	konstanta 010B

Z uvedeného je zřejmé, že jednoduchou změnou jednotlivých bitů můžeme adresovat jakýkoli bajt obrazové paměti. Cyklickým algoritmováním (pomocí maskování, rotací a posunů) můžeme na obrazovce vytvářet obrazce a animace. Toho se využívá ve hrách a různých grafických vyjádřeních.

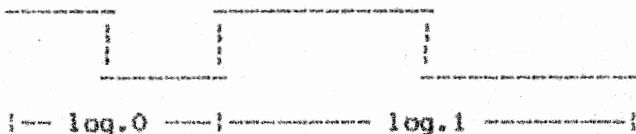
Součástí obrazové paměti jsou barevné atributy. U mikropočítačů obvykle nelze vybarvovat každý jednotlivý bod grafické sítě jinou barvou z nabízené palety. Nejčastěji lze barvit jen celou jednu znakovou pozici 8 x 8 bodů (průsečík sloupce a znakové řádky) jednou barvou papíru a jednou inkoustu. Neaktivovaný obrazový bod má barvu papíru, aktivovaný inkoustu. Pozici můžeme ještě přidělovat 2 stupně jasu a ev. i blikání. Adresování atributů je velmi jednoduché. U Spectra jde od levého horního rohu přes celou obrazovku přesně tak, jak bychom četli psaný text. Bity každé ze 768 adres atributů Spectra mají toto určení: bity 0-2 pro barvu inkoustu, bity 3-5 papíru, bit 6 pro jas, bit 7 pro blikání. Adresaci obrazové paměti se zabývá ještě předposlední kapitola.

Přenos dat z obrazové paměti do vstupních obvodů televizoru či monitoru zajišťuje obvykle zákaznický obvod ULA, který průběžně zjišťuje obsah paměti a převádí jej na signál odpovídající požadavkům vstupních obvodů monitoru. Pro TV anténní vstup je signál upravován vestavěným modulátorem. Do tohoto procesu uživatel softwarově zasahovat nemůže. ULA mívá na starosti ještě jiné funkce (např. ve Spectru obsluhuje všechny porty). Místo obvodu ULA se u počítačů s lepším grafickým podáním používají tzv. grafické karty, u nichž se v oblasti osobních počítačů dá dnes hovořit o určité, i když členité standardizaci (karty CGA, EGA, Hercules). Tyto karty obsahují řadu rychlých integrovaných obvodů pro zpracování a přenos obrazových dat.

Pevně situovaná obrazová paměť na nízkých adresách zneumožňuje využití mikropočítače pro práci s operačními systémy jako je CP/M. Tyto systémy obecně vyžadují, aby obrazová paměť byla umístěna v horní části paměti, zatímco v dolní je jen krátký služební program. Každý počítač, který svou konstrukcí umožňuje použití takových operačních systémů, přináší svému uživateli výhodu v možnosti používat programy vytvořené pro ten který operační systém, nemusí se omezovat na obvykle úzký výběr programů pro danou značku počítače. Softwarové firmy produkují mnohem více kvalitních programů pro operační systémy, které se ujal a staly se tak jistými standardy. Díky tomu, že nejsou tolik závislé na značce počítače, lépe se prodávají, jejich produkce je rentabilnější. A tato vyšší péče s sebou samozřejmě nese i vyšší kvalitu. Spectrum stojí mimo tuto oblast využití.

4/ Magnetofon

V operačních systémech mikropočítačů, obvykle zabudovaných do paměti ROM, jsou rutiny pro řízení přenosu dat mezi počítačem a magnetofonem. Jde o přenos sériový, zprostředkovaný vždy jednou linkou vstupní (do počítače) a výstupní (z počítače) portu. Základní funkcí tohoto přenosu je převedení obsahu určitého úseku paměti na takový formát záznamu (tvar signálu), který bude zpětně transformovatelný do paměti počítače bez jakéhokoli zkreslení původně zaznamenané informace. V raných dobách se pro záznam používala amplitudová modulace. Měl ji např. dědeček ZX-81. Jistě si jeho bývalí majitelé vzpomenu, co se nazlobili se čtením dat z pásky. Tato modulace je pro záznam dat nevhodná. Dnes se používá tzv. pseudo frekvenční modulace nebo její modifikace. U pseudo FM je rozdílná doba trvání pulsu, symbolizujícího hodnotu zaznamenávaného nebo čteného bitu. Log.0 je obvykle dvakrát kratší než log.1 (Spectrum, Amstrad aj.):



Při zápisu na pásek jde tedy o generování takového kmitočtu, který odpovídá hodnotě přečteného bitu určitého místa paměti. Při čtení z pásku je zjišťována délka čteného pulsu porovnáním s časovou konstantou. Výsledkem operace je stanovení hodnoty bitu a jeho uložení do určitého místa paměti. Bity jsou čteny i zapisovány sekvenčně. Rutiny, zajišťující tyto operace, jsou velmi precizně časovány. Přesto obvykle mají poměrně dost velkou toleranci, která umožňuje správné načtení záznamu i při menším kolísání mechaniky magnetofonu. Celý blok záznamu bývá schématicky takovýto:

zaváděcí signál--sync puls--indikační bajt--data--paritní bajt

Zaváděcí (angl. leader) signál s dlouhými pulsy je nezbytný proto, aby při čtení dat byly čtecí rutiny uvedeny do pohotovosti, tedy aby počítač "naskočil". Pak se čeká na krátký synchronizační puls, který dává na vědomí, že po něm již následuje vlastní informace. Po skončení sync pulsu jsou

aktivovány rutiny pseudo FM. Indikační bajt informuje o tom, zda následující data jsou hlavičkou nebo blokem paměťových dat. V závěru je jeden paritní bajt pro verifikaci správného průběhu čtení dat.

Obecně zápis bývá rozdělen do dvou základních částí. První - krátké - se říká hlavičková. Nese informaci o názvu následujícího bloku dat, o tom, na jaké adrese blok začíná a ev. další pomocné informace. Blok už obsahuje vlastní data paměti. Pokud hlavičková část chybí, označuje se soubor dat jako bezhlavičkový (angl. headerless file).

Některé mikropočítače (Amstrad, Atari) zapisují a čtou data v malých "subblocích" (např. po 0,5K). Jde o období dělení zápisu na délky sektorů disku. Když jsem u Amstradu CPC 464 zjistil tuto "přednost", která zápis na magnetofon i čtení vydatně časově prodlužuje (po každém subbloku následuje pauza a zaváděcí signál dalšího subbloku), vrhl jsem se na firmware počítače. S údivem jsem zjistil, že jeho systém má implementovány rutiny pro čtení i zápis bloku "v jednom kuse". Z Basicu jsou však nepřístupné. Pochopitelně jsem hned přešel na jejich využití.

Hezkou finesou, kterou má málo počítačů, používajících magnetofon, je nastavení časovacích parametrů čtecích rutin v průběhu čtení hlavičky, do níž je pro ten účel zapsáno třeba 256 bitů log.0. Při jejich čtení se počítač automaticky nastaví na frekvenci následujícího zápisu. Tuto dispozici má Amstrad.

V dnešní době je tak trochu zvráceností se ještě vůbec bavit o zápisu dat na magnetofon. Ale co dělat, když všeobecné nasazení disků je u nás stále ještě hudbou budoucnosti. V computerově rozvinutých zemích má kazeta s magnetofonovým páskem zajištěnou budoucnost jen při kopírování obsahu hard disku pomocí zařízení, zvaného streamer. Jde o kazetu velmi speciální, jejíž účel použití je pro naše domovy na úrovni sci-fi.

A tak stojíme před otázkou - tvořit své vlastní rutiny pro práci s magnetofonem nebo ne? Odpověď není tak jednoduchá. Chceme-li nebohému uživateli zápasícímu s magnetofonem aspoň trochu ulehčit jeho úkol, do rozsáhlejšího programu, který bude používat častěji, bychom takové rutiny udělat mohli, lépe řečeno - udělat měli. Co by takové rutiny měly obsahovat? Především zrychlení zápisu a čtení dat dle volby uživatele. Limitní je tu kvalita jeho magnetofonu a pásky. Dále bychom měli zařadit automatiku nastavení časování rutin. Více hlavičkových informací, než je obligátních 10 znaků názvu. A komprese dat pro zkrácení záznamu. Mám tu na mysli záznam a čtení dat z databank, slovních procesorů apod. Pro jiné účely je celkem zbytečné rutiny vymýšlet a zařazovat, tehdy je vhodnější využít rutin operačního systému počítače.

Řešení všech těchto požadavků budou obsahovat rutiny slovního procesoru v připravovaném 3.dílu programování Z80. Pro jejich rozsáhlost i potřebu delšího vysvětlující průvodního slova je nebylo možné zařadit do této učebnice. Námět na komprimaci přenášených dat je v kapitole, věnované komprimaci.

Kanály a proudy

Součástí některých operačních systémů mikropočítačů bývají tzv. kanály a proudy (angl. channels and streams). V podstatě jde o softwarové řešení aktivace požadovaných vstupně výstupních operací pomocí přenášených parametrů v soustavě rutin. Významnou roli tu hraje datová oblast proměnných, soustředěných do dvou tabulek. Jedna obsahuje startovací adresy rutin vstupu a výstupu jednotlivých kanálů (obrazovka, klávesnice, tiskárna apod.), druhá informace o tom, který kanál je otevřen a na kolikáté adrese od začátku kanálové tabulky je uložena adresa vstupu/výstupu toho kterého proudu. Jde vlastně o kombinaci dvou tabulek (o nich v kapitole Tabulky) - jedné s klíči ke druhé, skokové. Obsah obou je měnitelný (leží v RAMce). Přenos dat aktivovanými rutinami vstupu a výstupu probíhá bajt po bajtu obsahem registru A.

Jak jsem již uvedl, jsou všechny rutiny učebnice převedeny do slovního procesoru z assemblerového editoru. Pro ty z vás, kteří máte ZX Spectrum a nějakou verzi generátoru GENS, uvádím řešení převodu. Z tabulky kanálových adres jsem použil adresu výstupu na tiskárnu - převod z editoru assembleru bude probíhat po zadání příkazu k tisku zdrojového textu na tiskárnu. V systémové proměnné na adrese 23631 je uložena 1.adresa tabulky kanálových adres. O 15 adres dál leží dva bajty, které tvoří adresu výstupu

na tiskárnu. Tuto adresu měním na adresu začátku převodního programu. Tak nejde nic na tiskárnu, ale výstup je nasměrován na níže uvedený program, který zapisuje zdrojový text v potřebné úpravě do paměti RAM. Odtud jej lze nahrát na jakékoli vnější paměťové médium a z něj pak načítat přímo do slovního procesoru. Výhoda tohoto způsobu převodu tkví v tom, že v textu jednak eliminujeme funkční chyby, kterým bychom se při přímém zápisu do slovního procesoru těžko ubránili, i v tom, že je mnohem příjemnější zapisovat assemblerový text do editoru, který také všemu dá pěkný vzhled.

INIC	ORG 27000	
	LD HL, VY	; Přenos konstant VY, MO, CI do oblasti
	LD DE, VYNECH	; proměnných (DEFS)
	LD BC, 4	; Přenáší se 4 bajty
	LDIR	; Blokový přenos
KANAL	LD IX, (23631)	; 1. kanálová adr. do IX
	LD (IX+15), E	; a změna adresy výst. tisk. kanálu na adr.
	LD (IX+16), D	; v reg. DE (=adr. ODSUD)
	RET	; Návrat po inicializaci
VY	DEFB 129	; Výchozí hodn. čítače vynechaných znaků
MO	DEFW 48000	; 1. adr. uložení znaků textu assembleru
CI	DEFB 0	; Výchozí hodn. čítače znaků na řádce
VYNECH	DEFS 1	; Proměnná hodn. čítače vynech. znaků
MOMADR	DEFS 2	; " hodn. moment. adr. uložení znaku
CITAC	DEFS 1	; " hodn. čítače znaků/ř.
ODSUD	LD HL, VYNECH	; Do HL adr. proměnné VYNECH
	DEC (HL)	; Snížení prom. VYNECH o 1
	RET NZ	; Dokud nevynecháno 129 zn., vracej se
	LD DE, START	; Do reg. DE adr. START
	CALL KANAL	; A změna adr. výst. tisk. kanálu na START
START	CP 9	; Kód 9 (v reg. A) ignoruj
	RET Z	; Když v reg. A není kód 9,
	DI	; zablokuj mask. přerušení
	LD HL, CITAC	; Do HL adr. CITAC
	INC (HL)	; Zvýšení čítače znaků na řádce o 1
	LD IX, (MOMADR)	; Do IX momentální adresa pro uložení zn.
	CP #0D	; Má znak kód #0D?
	LD C, A	; Ušchovej znak z reg. A do reg. C
	LD A, (HL)	; Stav čítače zn/ř do reg. A
	JR Z, MEZ	; Když je to 0DH, skok na MEZ
	CP 11	; Už čítán 12. znak řádky assembleru?
	JR NC, PRENOS	; Když AND, skok na PRENOS
	JR VENO	; Když ještě NE, skok na VENO
MEZ	SUB 64+11	; V reg. A bude počet mezer pro doplnění
	NEG	; cílové řádky mezerami do jejího konce
	LD B, A	; Počet mezer z reg. A do reg. B
	LD A, 40	; Na 40. pozici zprava přijde středník
MEZERY	LD (IX+0), "	; Ulož mezeru na momentální adresu
	CP B	; Už je to 40. pozice zprava?
	JR NZ, MEZ1	; Když NE, skok na MEZ1
	LD (IX+0), ";"	; Když AND, uložení středníku na mom. adr.
MEZ1	INC IX	; Zvýšení moment. adr. uložení o 1
	DJNZ MEZERY	; Pokračuj, dokud řádka není zaplněna celá
	LD (HL), B	; Uložení nuly do proměnné CITAC pro
	JR VEN	; další řádku a skok na adr. VEN
PRENOS	LD (IX+0), C	; Uložení znaku na moment. adresu,
	INC IX	; její zvýšení o 1
VEN	LD (MOMADR), IX	; a uložení její hodnoty do prom. MOMADR
VENO	EI	; Uvolnění mask. přerušení
	RET	; Návrat

Přenos textu je uzpůsoben potřebě přenést jen návštěví a instrukce. Proto vynechává čísla adres a řádek výpisu (celkem 11 znaků). Rovněž ignoruje stále se opakující výpis názvu firmy na začátku. Protože je příjemné, když ve výsledném textu máte hned připravené i středníky pro připojení komentáře, program je generuje na určené místo. Nová řádka je zapisována do RAMky vždy poté, kdy se v reg. A objeví kód 0DH, kterým končí řádka v GENSu. Program doplní chybějící počet mezer do konce řádky dlouhé 64 znaků tak, jak to vyžadují slovní procesory typu Tasword 2 nebo Spectral Writer. Pokud váš slovní procesor používá koncový znak řádky (např. nulu), pak ji nahradíte kód 0DH a řádku nebudete doplňovat mezerami (malá úprava programu, vedoucí k jeho zkrácení). Zdrojový text GENSu má před každým operandem instrukce řídicí kód 09H. Ten program odstraňuje.

Editor po skončení výpisu instrukcí připojuje ještě 4 řádky s informacemi, které tu nejsou potřebné. Abyste je nezapsali na záznam, odečtete od celkové délky přeneseného textu 4*64 bajtů. A ještě malá poznámka - na 1. řádku assemblerového textu v GENSu dejte příkaz *C- (ten je při přenosu rovněž ignorován). Při studiu programu poznáte, že kanálová adresa je modifikována - jednou má hodnotu ODSUD, pak START. Je to proto, že poprvé je rutinou ODSUD ignorováno 129 prvních znaků textu, pak už je

rutinou START přenášeno požadované. Před každým použitím programu pro přenos textu od adresy MO je nutno zvlášť zavolat rutinu INIC pro nastavení výchozích hodnot proměnných v datové oblasti. Délku načteného textu určíte odečtem obsahů adres MOMADR-MO, a to třeba pomocí Basicu, který za vás provede automaticky i nahrávku na pásek či disketu. V programu uvedené absolutní adresy 27000 a 48000 nejsou vůbec směrodatné, lze místo nich dát jiné. Jen je třeba mít na paměti, že program je dlouhý 101 bajtů a že je nutno rezervovat patřičnou část RAMky pro převáděný text. Pokud budete chtít do RAMky zapsat plnou informaci zdrojového textu, povede potřebná úprava přenosového programu k jeho podstatnému zkrácení.

Program zde uvádím na důkaz toho, že bez sebemenšího zásahu do jakéhokoli systémového nebo uživatelského programu z něj můžete "vytáhnout" data ve formě, jakou si budete přát. Ale napřed musíte prozkoumat jejich formát, vymyslet, jak je odklonit ze zavedených přenosových kolejí a napsat krátký formující program v assembleru. Pochopitelně, že analýzou GENSu apod. lze najít místo, kam bychom mohli přímo "přilepit záplatu", která by dělala totéž, přičemž by její funkce byla ovládána přímo z programu. Tento krok by byl vhodný pouze tehdy, kdybychom funkci záplaty potřebovali často. Jinak by byla neefektivním přívěškem.

Posledně uvedený příklad poukazuje na to, že i když mnohé mají všechny systémy společné, jakmile jdeme na jejich nejnižší úroveň, musíme nutně poznat samotnou "individualitu" počítače, s nímž pracujeme. Systémy počítačů prodělávají bouřlivý vývoj. Pokud se budete chtít věnovat programování i ve vzdálenější budoucnosti, nezbyde vám, než se důvěrněji seznámit s principy jejich funkcí. Pak, ať už se systém bude jmenovat CP/M, MS DOS, AMSDOS či jakkoli jinak, vždy si budete vědět rady, protože rozdíly základních funkcí budou jen v jejich soft/hardwarem uspořádání.

KAPITOLA 3

Aritmetika

Tzv. rutiny kalkulátoru, které v operačních systémech počítačů zahrnují i aritmetiku, bývají dost rozsáhlé. Kapitola samou pro sebe je práce s pohyblivou řádovou či desetinnou čárkou (angl. floating point), počet platných míst za ní, otázka přesnosti výpočtů, jejich zaokrouhlování atd. Zde lze jen doporučit, aby se každý, kdo programuje v assembleru a narazí na problém "vyšší" matematiky, seznámil s kalkulátorem počítače a ve svém programu jej využil. Psát svůj vlastní kalkulátor v assembleru by bylo zcela zbytečnou ztrátou času. Samozřejmě s výjimkou případu, kdy bychom tvořili kompilátor či interpreter nějakého nového jazyka (takové starosti má však jen velmi málo lidí). Psaní vlastního kalkulátoru v assembleru by bylo možno přirovnat ke snaze vyrábět si vlastní benzín. Je jednodušší opatřit si jej u pumpy. Mohou se vyskytnout případy, kdy někdo chce, aby mu ZX Spectrum či Atari 800XL počítalo s přesností na 1000 desetinných míst ve velkých rychlostech. V tom případě je lepší buď přemluvit kočku, aby štěkala, nebo (má-li se na to) opatřit si odpovídající počítač s odpovídajícím programovým vybavením. Zřejmě nebudu sám, kdo je "sociálně" předurčen k tomu prvnímu.

Rutiny této kapitoly umožňují pracovat s tou nejjednodušší aritmetikou. V naprosté většině programů, jejichž vlastní funkce nejsou složité výpočty, víc nic ani nepotřebujeme. Pokud jde o rutiny sečítání a odečítání, není třeba k nim cokoli dodávat. Bude-li se někdo cítit trochu nejistý u součtů a odečtů BCD čísel, odkazují ho na část minulých učebnic, která se zabývá čísly BCD a instrukcí DAA. Rovněž princip binárního násobení a dělení (algoritmy posuň-a-přičti a posuň-a-odečti) předchozí učebnice demonstruje velmi detailně.

Využití rutin v programech je velmi široké. Na prvním místě lze uvést interakci programu s jeho uživatelem, který je vybízen buď k přímému vkládání číselných parametrů, nebo se za jeho úkony vždy nutně nějaké aritmetické operace v programu "schovávají". Na aritmetiku přímo navazují rutiny pro převod mezi jednotlivými číselnými soustavami a kódy. Nebudeme přece uživatele nutit, aby číselné parametry vkládal binárně či hexadekadicky. Stejně tak by všechny číselné informace, které přicházejí z programu na obrazovku, měly mít dekadickou řádovku. V assemblerovém programu se nám ovšem často lépe pracuje ve dvojkové či HD soustavě. Proto tu jsou rutiny pro práci s čísly binárními, HD i BCD. Jejich převody na jiný číselný základ najdete v kapitole Konverze.

1/ Dělení 16-bitového BCD čísla dvěma

je demonstrováno dvěma následujícími rutinami. První je kratší, ale má delší dobu provedení, druhá naopak. Zatímco při dělení dvěma se u HD čísel nemusíme starat takřka o nic (posun doprava dělí dvěma bez nutnosti dodatečných úprav), u BCD čísel musíme vyrovnat přenosy mezi jednotlivými bajty i vyššími a nižšími 4mi bity uvnitř bajtů, tedy provést jejich adjustaci (k tomu blíže předchozí učebnice v části o instrukci DAA).

a/ pomalejší rutina

Algoritmus:

Pro každou číslici:

Při objevivším se přenosu odečti 6 od vyšší číslice

Jinak vždy rotuj vlevo přes CY o 1 číslici (4 bity)

Délka: 20 bajtů

Počet cyklů: 665-683

Vstup: v HL BCD číslo

Výstup: v HL celočíselný BCD výsledek, v CY zůstatek

Použité registry: HL,BC,AF

```

DABCNT EQU #A004      ;AOH je DK čísla 60H, 04 čítač posunů
HLFHL  PUSH BC         ;Ušchovej obsahy registrů BC a AF
        PUSH AF
        LD  A,4         ;Počet BCD číslic
        OR  A           ;Vynulování CY
DIGLP  JR  NC,CARYOK    ;Když CY=0, skok na CARYOK
        ADD HL,BC       ;Odečet 6 od vyšší číslice před posunem
CARYOK LD  BC,DABCNT    ;Do BC číslo DABCNT
BITLP  ADC  HL,HL       ;4 rotace HL doleva přes CY pro posun
        DEC  C          ;další BCD číslice na nejvyšší pozici
        JR  NZ,BITLP
        DEC  A          ;Zopakuj pro všechny 4 číslice, v HL zd-
        JR  NZ,DIGLP    ;stane adjust.č.posunuto o 1 poz.doprava
        POP  AF         ;Obnova obsahu registrů AF,BC
        POP  BC
        RET            ;Návrat

```

b/ rychlejší rutina

Délka: 23 bajtů
 Počet cyklů: 82-93
 Vstup: v HL BCD číslo
 Výstup: v HL BCD číslo, v CY zůstatek
 Použité registry: HL,BC,AF

```

HLFQ  LD  A,L          ;Desítky a jednotky do reg.A
        BIT 4,A         ;Test na přenos do jednotek
        JR  Z,UNITOK    ;Když bez přenosu, skok na UNITOK
        SUB 6           ;Jinak adjustace před dělením
UNITOK SRL  H          ;Dělení tisíců a stovek dvěma
        JR  NC,TENSOK   ;Skok na TENSOK bez přenosu do desítek
        ADD A,#A0       ;Při přenosu (přes CY) adjustace
TENSOK RRA          ;Dělení adjustovaných desítek a jednotek
        BIT 3,H         ;Test na přenos do stovek
        JR  Z,HNSOK     ;Když není přenos, skok na HNSOK
        DEC H          ;Jinak následuje adjustace
        DEC H
        DEC H
HNSOK  LD  L,A         ;Desítky a jednotky do L
        RET            ;Návrat

```

2/ Násobení dvou 16-bitových bin.čísels

Výsledek, rovněž 16-bitový, je v reg.HL. Rutina se nehodí pro čísla, jejichž vynásobením bychom získali výsledek přesahující 16 bitů. Použit je algoritmus posun-a-přičti. Násobenec je přičten k meziproductu vždy, když je v násobiteli nalezen bit log.1. Násobitel a meziproduct se posouvají vlevo 15 krát.

Délka: 22 bajtů
 Počet cyklů: 865-965
 Vstup: v HL násobenec, v DE násobitel
 Výstup: v HL 16-bitový výsledek
 Použité registry: HL,DE,BC,AF

```

MUL16 LD  C,L          ;Násobenec do BC
        LD  B,H
        LD  HL,0        ;Přednastavení "výsledku" na nulu
        LD  A,15        ;Čítač počtu bitů-1
MLP  SLA  E            ;Posun násobitele o 1 bit vlevo
        RL  D
        JR  NC,MLP1     ;Když CY=0, skok na MLP1
        ADD HL,BC       ;Když CY=1, sečti násobence s meziproduct.
MLP1 ADD  HL,HL        ;Posun meziproductu vlevo
        DEC  A          ;Zmenšení čítače bitů o 1
        JR  NZ,MLP      ;Pokračuj, dokud čítač není nulový
        OR  D           ;Test znaménka násobitele (nejvyšší bit)
        RET  P          ;Když je kladné (bit 15=0), návrat
        ADD HL,BC       ;Jinak sečti násobence s meziproductem
        RET            ;Návrat

```

3/ Dělení dvou 16-bitových čísel

V rutině jsou dva vstupy - SDIV16 bere ohled na znaménka čísel, UDIV16 nikoli. Když je dělitel nulový, pak CY=1 a výsledné registry jsou vynulovány. Jinak je CY=0. Rutina používá algoritmus posun-a-odečti. Násobenec a výsledek jsou posouvány vlevo, bit log.1 je umístěn do výsledného registru vždy, když výsledek dělicí meziprocedure splňuje subrutinou DVLOOP stanovené podmínky.

Délka: 108 bajtů (data 3)

Počet cyklů: 1770-2340

Vstup: v HL dělenec, v DE dělitel

Výstup: v HL výsledné celé číslo, v DE zůstatek

Použité registry: HL, BC, DE, AF

```
SDIV16 LD A,H ;Stanovení znaménka (bitu 15) dělence
        LD (SREM),A ;Zůstatek bude mít stejné znaménko
        XOR D ;XOR s vyšším bajtem dělitele
        LD (SQUOT),A ;Ušchování znaménka celočísl.výsledku

        LD A,D ;Následuje převod na abs.hodn.dělitele
        OR A
        JP P,CHKDE ;Skok na CHKDE při kladném děliteli
        SUB A ;Vynulování reg.A
        SUB E ;Následuje odečet dělence od nuly pro
        LD E,A ;získání jeho absol.,tedy kladné hodnoty
        SBC A,A ;A=#FF,když CY=1 z předchozího odečtu
        SUB D ;Dokončení převodu na abs.hodnotu
        LD D,A

CHKDE LD A,H ;Určení absolutní hodnoty dělence
        OR A
        JP P,DODIV ;Skok na DODIV při kladném dělenci
        SUB A
        SUB L
        LD L,A
        SBC A,A
        SUB H
        LD H,A

DODIV CALL UDIV16 ;Volání subrutiny dělení abs.hodnot
        RET C ;Návrat při dělení nulou
        LD A,(SQUOT) ;Následuje negace výsledku,je-li záporný
        OR A
        JP P,DOREM ;Skok na DOREM při kladném výsledku
        SUB A
        SUB L
        LD L,A
        SBC A,A
        SUB H
        LD H,A

DOREM LD A,(SREM) ;Následuje negace zůstatku,je-li záporný
        OR A
        RET P ;Návrat ze subrut.při kladném zůstatku
        SUB A
        SUB E
        LD E,A
        SBC A,A
        SUB D
        LD D,A
        RET ;Návrat ze subrutiny

UDIV16 LD A,E ;Dělení kladných hodnot;nejdřív zjiště-
        OR D ;ní,je-li dělitel nulový
        JR NZ,DIVIDE ;Když NE,skok na DIVIDE
        LD HL,0 ;Když AND,vynulovat výsledné registry
        LD D,H
        LD E,L
        SCF ;a nastavit CY na log.1
        RET ;Návrat ze subrutiny

DIVIDE LD C,L ;Do reg.C nižší bajt dělence/výsledku
        LD A,H ;Do reg.A vyšší
        LD HL,0 ;V HL bude zůstatek
        LD B,16 ;Čítač 16 bitů pro algoritmus smyčky
        OR A ;Vynulování CY
```

;Posun jednotlivých bitů výsledku do bitu 0 dělence
 ;Posun nejvyšších bitů dělence do bitu 0 zůstatku
 ;V reg.BC je v průběhu algoritmu současně dělenec i výsledek.
 ;Při posuvu nejvyššího bitu dělence posunujeme minulý obsah CY
 ;do nejnižšího bitu budoucího výsledku. Cíli dělenec "ubývá"
 ;zleva, výsledek "přibývá" zprava. V HL bude zůstatek.
 ;Celkově jde o 32-bitový posuv vlevo: CY do C, C do A, A do L a
 ;L do H.

```
DVLOOP RL C ;CY, jako další bit výsledku, do bitu 0
        RLA ;Následuje posuv zbývajících bajtů
        RL L
        RL H ;Rovněž nuluje CY (HL byl výše nulován)
```

;Když je zůstatek větší než dělenec nebo je mu roven, další bit
 ;výsledku bude log.1. Tento bit bude v CY po jeho komplementaci.


```

PUSH HL      ;Uchování moment.zůstatku
SBC HL,DE    ;Odečet dělitele od zůstatku
CCF          ;Komplementace přenosu
JR C,DROP    ;Při CY=1 je zůst.větší či roven dělenci
EX (SP),HL   ;Jinak odeber zůstatek ze zásobníku

DROP INC SP   ;Zvětšení SP o 2
INC SP
DJNZ DVLOOP  ;Proveň operaci se všemi bity
EX DE,HL     ;V DE zůstatek
RL C         ;CY do reg.C
LD L,C       ;Následuje posun posledního CY do vý-
RLA          ;sledku
LD H,A       ;Vynulování CY pro signalizaci platného
OR A         ;výsledku a návrat
RET

SQUOT DEFS 1 ;Znaménko celočísl.výsledku
SREM DEFS 1  ;Znaménko zůstatku
COUNT DEFS 1;Čítač smyčky dělení
    
```

4/ Multibajtový binární součet

Rutina nebere v úvahu znaménka. Nejnižší bajty každého z obou čísel jsou na jejich nejnižších adresách. Výsledek bude uložen na původním místě uložení čísla, k němuž se přičítá. Délka obou čísel může být maximálně 255 bajtů. Na začátku součtu je CY vynulován a postupně jsou čísla sečítána od nižších bajtů vzhůru. Nastavení CY na log.1 signalizuje ukončení součtu. Při nulové délce čísla je proveden okamžitý návrat. Instrukce ADC zajišťuje přenos bitu do vyššího bajtu v případě, že je součet obou nižších bajtů vyšší než 255.

Délka: 11 bajtů

Počet cyklů: 46 na jeden bajt plus 18

Vstup: v HL adresa nejnižšího bajtu 1.sčítance

v DE " " " " 2. " "

v B délka čísel v bajtech (čítač smyčky)

Výstup: Na místě prvního sčítance je uložen výsledek součtu

Použité registry: HL,DE,B,AF

```

MPBADD LD A,B      ;Čítač do reg.A pro zjištění,
AND A           ;zda není nulový (a vynulování CY)
RET Z           ;Když AND, návrat; jinak pokračuj

LOOP LD A,(DE)    ;Jeden bajt 2.sčítance do reg.A
ADC A,(HL)        ;Přičtení bajtu 2.sčítance
LD (HL),A         ;Uložení výsledku na adr.bajtu 2.sčít.
INC HL            ;Zvýšení obou adres
INC DE
DJNZ LOOP         ;Pokračuj, dokud není čítač bajtů nulový
RET              ;Návrat
    
```

5/ Multibajtové binární odečítání

Pro tuto rutinu platí analogicky totéž, co pro předcházející (ADC je nahrazeno SBC).

Délka: 11 bajtů

Počet cyklů: 46 na jeden bajt plus 22

Vstup: v HL adresa nejnižšího bajtu menšího

v DE " " " " menšence

v B délka čísel v bajtech (čítač smyčky)

Výstup: Na místo menšence je ukládán výsledek součtu

Použité registry: HL,DE,B,AF

```

MPBSUB LD A,B      ;Totéž jako výše
AND A
RET Z

LOOPS LD A,(DE)    ;Následuje provedení odečtu
SBC A,(HL)
LD (DE),A
INC DE
INC HL
DJNZ LOOPS
RET
    
```

6/ Multibajtové binární násobení

Rutina nebere v úvahu znaménka. Délka čísel může být maximálně 255 bajtů. Pokud výsledek přesahuje délku 255 bajtů (nebo stanovenou délku čísel), je přesah v poli HIPROD. Rutina používá algoritmus posuň-a-přičti, během jehož exekuce přičítá násobitele k meziproductu vždy, když je nalezen bit log.1 v násobenci. Násobehlec a meziproduct jsou posouvány o 1 pozici, a to tolikrát, kolik bitů každé číslo obsahuje, plus 1. Obě čísla mají stejný počet bajtů, tedy i bitů.

Délka: 104 bajtů plus 261 pro data

Počet cyklů: v průměru $(728 * \text{délka} ** 2) + (883 * \text{délka}) + 300$

Vstup: v HL adresa nejnižšího bajtu násobence

v DE " " násobitele

v B délka čísel v bajtech (čítač smyček)

Výstup: Na místě násobence je uložen výsledek, ev.v poli HIPROD je přesah násobení.

Použité registry: HL, DE, B, AF

```

MPBMUL LD A,B           ;Je délka čísel nulová?
      AND A
      RET Z             ;Když AND, návrat
      LD C,B           ;Délka čísel do BC
      LD B,0
      ADD HL,BC         ;Konec pole násobence
      EX DE,HL         ;přenesen do DE
      LD (MLIER),HL     ;1.adr.násobitele do dat.oblasti
      LD HL,HIPROD      ;1.adr.pole HIPROD do HL
      ADD HL,BC         ;Konec "
      LD (ENDHP),HL     ;přenesen do dat.oblasti
      LD L,C            ;Počet bajtů do HL pro následující
      LD H,B            ;určení celkového počtu bitů + 1 každého
      ADD HL,HL         ;z čísel
      ADD HL,HL
      ADD HL,HL         ;Dokončeno násobení osmi
      INC HL            ;plus 1
      LD (COUNT),HL   ;Výsledek do obl.dat

ZEROPD LD B,C           ;Následuje vynulování pole HIPROD
      LD HL,HIPROD
ZEROLP LD (HL),0
      INC HL
      DJNZ ZEROLP

LOOPM  AND A            ;Vynulování CY
      LD B,C           ;Počet bajtů do B
      LD HL,(ENDHP)    ;Konec pole HIPROD do HL
SRPLP  DEC HL           ;Ukazatel adres pole o 1 snížen
      RR (HL)          ;Posun všech bitů pole doprava a nejniž-
      DJNZ SRPLP       ;ší bit pole do CY

      LD L,E           ;Konec násobence do HL
      LD H,D
      LD B,C           ;Počet bajtů do B
SRAILP DEC HL          ;Ukazatel adres pole násobence minus 1
      RR (HL)          ;Přenos CY do nejvyššího bitu násobence
      DJNZ SRAILP      ;a nejnižšího bitu do CY
      JP NC,DECCNT     ;Když CY=0, skok na DECCNT
      PUSH DE          ;Jinak uschovat adr.konce pole násobence
      LD DE,(MLIER)    ;Do DE adr.násobitele
      LD HL,HIPROD     ;Do HL adr.pole HIPROD
      LD B,C           ;Do B počet bajtů
      AND A            ;Vynulování CY

ADDLP  LD A,(DE)        ;Přičtení obsahu pole násobitele k poli
      ADC A,(HL)        ;HIPROD
      LD (HL),A
      INC DE
      INC HL
      DJNZ ADDLP
      POP DE           ;Do DE adr.konce pole násobence

DECCNT LD A,(COUNT)   ;Zmenšení čítače bitů o 1
      DEC A
      LD (COUNT),A
      JP NZ,LOOPM      ;Když nižší bajt čítače není nulový,skok
      PUSH AF          ;Uschova CY
      LD A,(COUNT+1)
      AND A            ;Je nižší bajt čítače nulový?
      JP Z,EXIT        ;Když AND, skok na EXIT
      DEC A            ;Jinak zmenšit o 1,
      LD (COUNT+1),A  ;uložit do dat.oblasti
      POP AF           ;a obnovit stav CY
      JP LOOPM         ;Pokračovat smyčkou LOOPM

EXIT   POP AF          ;Odhoz zbytečné adresy ze zásobníku
      RET              ;Konečný návrat

COUNT DEFS 2          ;Čítač bitů každého z čísel
ENDHP  DEFS 2          ;Adresa konce pole HIPROD
MLIER  DEFS 2          ;1.adr.pole násobitele
HIPROD DEFS 255        ;1.adr.pole HIPROD

```

7/ Multibajtové binární dělení

Rutina nebere v úvahu znaménka. Obě čísla, nepřesahující délkou 255 bajtů, mají své nejnižší bajty na nejnižších adresách. Výsledek bude na původním místě dělení, adresa nejnižšího bajtu zůstatku v HL. CY=0, když nedošlo k chybě. CY=1 při dělení nulou - tehdy je dělenec ponechán beze změny a zůstatek nulový. Rutina používá algoritmus posuň-a-odečti. Posouván je výsledek a dělenec. Do výsledku se ukládá log.1 vždy, když dojde ke splnění podmínek stanovených v subrutině SUBLP. Tehdy je v akci zvláštní buffer. Když ke splnění podmínek nedojde, je buffer přepnut na pole dělení.

Délka: 161 bajtů a 522 pro data

Počet cyklů: v průměru $(1176 * \text{délka} ** 2) + (2038 * \text{délka}) + 515$

Vstup: v HL 1.adr.dělení, v DE dělitele, v B počet bajtů čísel

Výstup: v HL nejnižší bajt zůstatku, při CY=0 výsledek platný, celočíselný výsledek je na původním místě dělení

Použité registry: HL,DE,BC,AF

```
MPBDIV LD A,B ;Je-li délka čísel nulová,
OR A
JP Z,OKEXIT ;okamžitý návrat s CY=0
LD (DVEND),HL ;1.adr.dělení a
LD (DVSOR),DE ;dělitele do oblasti dat
LD C,B ;Počet bajtů do C
LD L,C ;i do HL
LD H,0
ADD HL,HL ;Určení počtu bitů + 1 v počtu bajtů
ADD HL,HL
ADD HL,HL ;Dokončení násobení osmi
INC HL ;plus 1
LD (COUNT),HL ;Výsledek do obl.dat
LD HL,HIDE1 ;Do HL adr.bufferu HIDE1
LD DE,HIDE2 ;Do DE " HIDE2
LD B,C ;Počet bajtů do B
SUB A ;Vynulování obsahu reg.A

ZEROLP LD (HL),A ;Vynulování obsahu obou bufferů
LD (DE),A
INC HL
INC DE
DJNZ ZEROLP
LD HL,HIDE1 ;Adr.bufferu HIDE1
LD (HDEPTR),HL ;do (HDEPTR)
LD HL,HIDE2 ;a HIDE2
LD (ODEPTR),HL ;do (ODEPTR)
LD HL,(DVSOR) ;Do HL adr.dělitele
LD B,C ;Do B počet bajtů
SUB A ;Vynulování obsahu reg.A pro další OR

CHKOLP OR (HL) ;Operace OR všech bajtů dělitele pro
INC HL ;zjištění, zda je dělitel nulový
DJNZ CHKOLP
OR A
JR Z,EREXIT ;Když je nulový, skok na EREXIT

LOOP OR A ;Vynulování CY
LD B,C ;Počet bajtů pro rotaci
LD HL,(DVEND) ;Do HL adr.dělení

SLLP1 RL (HL) ;CY, jako bit výsledku do bitu 0 dělení
INC HL ;a nejvyšší bit dělení do CY
DJNZ SLLP1

DECCNT LD A,(COUNT) ;Bitový čítač zmenšit o 1
DEC A
LD (COUNT),A
JR NZ,CONT ;Když nižší bajt čítače nenulový, skok
LD A,(COUNT+1)
DEC A ;Jinak zmenšit i vyšší bajt čítače
LD (COUNT+1),A
JP M,OKEXIT ;Při aritm.přetečení skok na OKEXIT

CONT LD HL,(HDEPTR) ;Do HL adr.1.bufferu
LD B,C ;Bajtový čítač

SLLP2 RL (HL) ;CY do bitu 0 1.bufferu a posuv všech
INC HL ;o 1 pozici vlevo
DJNZ SLLP2
PUSH BC ;Uschování počtu bajtů čísel
LD A,C
LD (SUBCNT),A ;i do oblasti dat pro subrutinu SUBLP
LD BC,(ODEPTR) ;Do BC 2.buffer
LD DE,(HDEPTR) ;Do DE 1.buffer
LD HL,(DVSOR) ;Do HL dělitel
OR A ;Vynulování CY
```

```

SUBLP LD A,(DE)      ;Odečítání dělitele od 1.bufferu
      SBC A,(HL)
      LD (BC),A      ;Výsledek do 2.bufferu
      INC HL
      INC DE
      INC BC
      LD A,(SUBCNT)
      DEC A          ;Zmenšení bajtového čítače o 1
      LD (SUBCNT),A
      JR NZ,SUBLP    ;Když ještě není nulový, skok na SUBLP
      POP BC         ;Když odečet skončen, obnov bajt.čítač

;Když CY=1, obsah 1.bufferu je menší než dělitel, takže další
;výsledný bit je 0. Při CY=0 bude bit výsledku 1 - tehdy pře-
;pne z dělení na zůstatek.

      CCF            ;Komplementace bitu přenosu, aby byl
      JR NC,LOOP     ;shodný s bitem výsledku; při CY=0 skok
      LD HL,(HDEPTR) ;Když CY=1, prohoď ukazatele adres obou
      LD DE,(ODEPTR) ;bufferů
      LD (ODEPTR),HL
      LD (HDEPTR),DE
      JP LOOP         ;a skoč na LOOP s CY i výsl.bitem=1

EREXIT SCF           ;Výstup s chybou, signalizace s CY=1
      JP EXIT
OKEXIT OR A          ;Skok na EXIT
EXIT LD HL,(HDEPTR)  ;Vynulování CY, výsledek platný
      RET            ;Do HL adresa zůstatku
                       ;Konečný návrat

DVEND DEFS 2         ;Adresa dělení
DVSOR DEFS 2         ;" dělitele
HDEPTR DEFS 2        ;" 1.bufferu (momentální)
ODEPTR DEFS 2        ;" 2. "
COUNT DEFS 2        ;Bitový čítač
SUBCNT DEFS 1        ;Proměnný bajtový čítač
HIDE1 DEFS 255       ;Obsah bufferu 1
HIDE2 DEFS 255       ;" " 2
    
```

8/ Multibajtové dekadické sečítání

Rutina nebere v úvahu znaménka. Obě čísla, jejichž maximální délka může být 255 bajtů, mají své nejnižší bajty na nejnižších adresách svého uložení. Rutina sečítá vždy dva bajty, tedy 4 BCD číslice najednou. Konečný stav CY reflektuje ev. přenos z posledního součtu. Při nulové délce čísel nastane okamžitý návrat. Výsledek se ukládá na původní místo 1. sčítance.

Délka: 12 bajtů

Počet cyklů: 50 pro každý jednotlivý součet 2 bajtů plus 18

Vstup: v HL adresa 1., v DE 2.sčítance, v B délka čísel

Výstup: na původním místě 1. sčítance bude výsledek

Použité registry: HL,DE,B,AF

```

MPDADD LD A,B        ;Je délka čísel nulová?
      OR A
      RET Z          ;Současné vynulování CY
                       ;Když AND, návrat

LOOPA LD A,(DE)
      ADC A,(HL)      ;Součet s CY
      DAA             ;dekadická adjustace
      LD (HL),A       ;Uložení výsledku
      INC HL
      INC DE          ;Přechod na další adresy
      DJNZ LOOPA      ;Pokračuj ve smyčce až do konce součtu
      RET            ;Návrat
    
```

9/ Multibajtové dekadické odečítání

Rutina nebere v úvahu znaménka. Obě čísla, jejichž maximální délka může být 255 bajtů, mají své nejnižší bajty na nejnižších adresách svého uložení. Rutina odečítá vždy dva bajty, tedy 4 BCD číslice najednou. Konečný stav CY reflektuje ev. přenos z posledního součtu. Při nulové délce čísel nastane okamžitý návrat. Výsledek se ukládá na původní místo menšence.

Délka: 13 bajtů

Počet cyklů: 50 pro každý jednotlivý odečet 2 bajtů plus 18

Vstup: v HL menšitel, v DE menšenec, v B délka čísel

Výstup: na původním místě menšitele bude výsledek

Použité registry: HL,DE,B,AF

```

MPDSUB LD  A,B      ;Test na délku čísel
      OR  A          ;a vynulování CY
      RET  Z          ;Návrat při nulové délce

LOOPS  LD  A,(DE)
      SBC A,(HL)      ;Odečet s CY
      DAA             ;Dekadická adjustace
      LD  (DE),A      ;Uložení výsledku
      INC HL
      INC DE
      DJNZ LOOPS      ;Pokračuj ve smyčce až do konce odečtu
      RET             ;Návrat

```

10/ Multibajtové dekadické násobení

Rutina nebere v úvahu znaménka. Obě čísla mají své nejnížší bajty na svých nejnížších adresách uloženy. Každé z čísel nemůže být delší než 255 bajtů. Rutina pracuje s každou BCD číslicí jednotlivě. Izoluje ji pomocí maskování, a je-li ve vyšších 4 bitech bajtu, posuny ji přenesou na nižší 4 bity. Číslice je použita jako čítač pro přičítání násobitele k meziprojektu. Nejnížší číslice meziprojektu je uschována jako další číslice výsledku a meziprojekt je posunut o 4 bity doprava. Je použit indikátor, signalizující, zda se pracuje s číslicí z vyšší nebo nižší poloviny bajtu. Nulová délka čísel způsobí okamžitý návrat.

Délka: 167 bajtů a 520 bajtů pro data

Počet cyklů: při průměrné hodnotě násobených číslic rovné 5:

$$(694 * \text{délka} * 2) + (1555 * \text{délka}) + 272$$

Vstup: v HL 1.adresa násobence, v DE 1.adresa násobitele
v B délka čísel v bajtech

Výstup: Na adresách původního násobence je uložen výsledek

Použité registry: HL,DE,BC,AF

```

MPDMUL LD  A,B      ;Test na bajtovou délku čísel
      OR  A          ;Když je nulová, návrat
      RET  Z          ;Jinak ji uschovej do dat.oblasti
      LD  (LEN),A    ;i coby čítač smyčky
      LD  (LPCNT),A  ;Adresy násobence a násobitele do ob-
      LD  (MCADR),HL ;lasti dat
      LD  (MPADR),DE ;Do DE adr.bufferu násobence a do pro-
      LD  DE,MCAND   ;měnné jako ukazat.adresy n-tého bajtu
      LD  (NBYTE),DE ;Délka čísel do BC
      LD  C,B        ;Do HL adr.bufferu násobence a do pro-
      LD  B,0        ;měnné jako ukazat.adresy n-tého bajtu
      LDIR           ;Přenesení obsahu násobence do bufferu
      LD  HL,(MCADR) ;Do HL adr.násobence
      LD  A,(LEN)    ;Do A bajtová délka čísel
      CALL ZERBUF    ;Volání subrut.pro vynul.pole násobence
      LD  HL,PROD    ;Do HL adr.pole výsledku
      CALL ZERBUF    ;a vynulování pole

LOOP   LD  A,1      ;Inic.čítače pozice BCD číslice v bajtu
      LD  (DCNT),A  ;a jeho uložení do obl.dat

DLOOP  SUB  A        ;Zmenšení čítače o 1 a zároveň vynulo-
      LD  (OVRFLW),A ;vání indik.aritm.přetečení a úschova
      LD  A,(DCNT)   ;Minulý stav čítače
      OR  A          ;Test na nulu
      LD  HL,(NBYTE) ;Do HL adr.n-tého bajtu buff.násobence
      LD  A,(HL)     ;Bajt z této adr.do reg.A
      JR  NZ,DLOOP1  ;Když reg.A není nulový, skok na DLOOP1
      RRCA           ;Když je nulový, jde o vyšší BCD čísli-
      RRCA           ;ci, proto je posunuta na pozici nižších
      RRCA           ;čtyř bitů

DLOOP1 AND  #0F      ;Vynulování 4 vyšších bitů bajtu maskou
      JR  Z,SDIGIT   ;Je BCD číslice nulou? Když ANO, skok
      LD  C,A        ;Když NE, její přenos do reg.C

ADDLP  LD  HL,(MPADR) ;Do HL adr.násobitele
      LD  DE,PROD    ;Do DE adr.výsledku
      LD  A,(LEN)    ;Bajtový čítač
      LD  B,A        ;a jeho přenos do B
      OR  A          ;Vynulování CY

INNERL LD  A,(DE)    ;K bajtu výsledku
      ADC A,(HL)     ;přičti bajt násobitele s CY
      DAA           ;a proved dekadickou adjustaci
      LD  (DE),A     ;Výsledek ulož na moment.adr.pole výsl.
      INC HL         ;Zvyš adresy obou polí o 1
      INC DE
      DJNZ INNERL   ;Operaci proved n-krát
      JR  NC,DECND   ;Když po posl.součtu CY=0, skok
      LD  HL,OVRFLW  ;Zvětšení obsahu bajtu OVRFLW o 1
      INC (HL)

```


DECND	DEC	C	;Je obsah obou BCD číslic nulový?
	JR	NZ,ADDLP	;Když NE, skok na ADDLP
SDIGIT	LD	A,(PROD)	;Když AND, bajt výsledku do reg.A
	AND	#0F	
	LD	B,A	
	LD	A,(DCNT)	;Čítač pozice číslice v bajtu
	OR	A	;Jde o nižší číslici bajtu?
	LD	A,B	;Číslice zpět do reg.A
	JR	NZ,SD1	;Když je to nižší číslice, skok
	RRCA		;Když jde o vyšší, její posuv na pozici
	RRCA		;nižší
	RRCA		
	RRCA		
SD1	LD	HL,(MCADR)	;Uložení číslice na adr.pole násobence
	OR	(HL)	;jakožto pole budoucího výsledku
	LD	(HL),A	
	LD	A,(LEN)	;Bajtový čítač
	LD	B,A	;do reg.B
	LD	E,A	;i reg.DE
	LD	D,0	
	LD	HL,PROD	
	ADD	HL,DE	;V HL adr.za koncem pole výsledku
	LD	A,(OVRFLW)	;Do A čítač OVRFLW
SHFTLP	DEC	HL	;Přechod na další nižší adr.pole výsl.
	RRD		;Rotace BCD číslic doprava v celém poli
	DJNZ	SHFTLP	
	LD	HL,DCNT	;Už jsme pracovali s oběma číslicemi
	DEC	(HL)	;bajtu, resp. je čítač pozice na vyš.č.?
	JR	Z,DLOOP	;Když NE, ujmí se ještě vyšší číslice
	LD	HL,(NBYTE)	;Když AND, do HL adr.n-tého b.pole výsl.
	INC	HL	;a přechod na další adr.bajtu
	LD	(NBYTE),HL	;Ulož ji do proměnné
	LD	HL,(MCADR)	;Do HL adr.násobence,
	INC	HL	;přechod na jeho další adresu
	LD	(MCADR),HL	;a její uložení do proměnné
	LD	HL,LPCNT	;Do HL čítač smyčky LOOP
	DEC	(HL)	;a jeho zmenšení o 1
	JR	NZ,LOOP	;Když není čítač nulový, zpět na smyčku
EXIT	RET		;Když vše hotovo, návrat
ZERBUF	LD	(HL),0	;Subrutina pro vynulování polí bufferů
	LD	A,(LEN)	;Čítač počtu bajtů
	DEC	A	;Jeho zmenšení o 1
	RET	Z	;Když už je nulový, návrat
	LD	D,H	;Bajt na adr.HL už je nulový, příprava
	LD	E,L	;adresy DE (o 1 vyšší než HL) pro bloko-
	INC	DE	;vý přenos
	LD	C,A	;Do BC počet bajtů
	LD	B,0	
	LDIR		;Provedení přenosu nuly celým polem
	RET		;Návrat
LEN	DEFS	1	;Délka čísla, čili pole
DCNT	DEFS	1	;Čítač pozice BCD číslice v bajtu
LPCNT	DEFS	1	;Čítač smyčky LOOP
OVRFLW	DEFS	1	;Proměnná OVRFLW
MCADR	DEFS	2	; " adresa v poli násobence
MPADR	DEFS	2	; 1.adr.pole násobitele
NBYTE	DEFS	2	;Proměnná adr.n-tého bajtu v poli
PROD	DEFS	255	;Pole mezivýsledku
MCAND	DEFS	255	;Pole násobence/budoucího výsledku

11/ Multibajtové dekadické dělení

Rutina nebere v úvahu znaménka. Obě čísla, jejichž délka může být maximálně 225 bajtů, mají své nižší bajty uloženy na nižších adresách. Celočíslný výsledek se objeví na původních adresách dělece. Adresa nejnižšího bajtu zůstatku je určena obsahem HDEPTR. Když je CY=0, je výsledek platný. Při CY=1 došlo k pokusu o dělení nulou - tehdy je obsah pole dělece nezměněn a zůstatek je nulový. Během dělení je dělenec i průběžný výsledek rotován číslicí po číslici vlevo.

Délka: 168 bajtů a 523 pro data

Počet cyklů: při průměrné hodnotě číslic výsledku ve výši 5:

$(1054 * \text{délka} ** 2) + (2297 * \text{délka}) + 390$

Vstup: v HL 1.adr.dělece, v DE 1.adr.dělitele
v B počet bajtů čísel

Výstup: viz text výše

Použité registry: HL,DE,BC,AF

```

MPDDIV LD (DVADR),HL ;Adresy dělence a dělitele do obl.dat
        LD (DSADR),DE
        LD A,B
        LD (LENGTH),A ;Uložení délky číslic do obl.dat
        OR A ;Test na nulovou délku
        JP Z,OKEXIT ;Je-li nulová, okamžitý návrat
        LD HL,HIDE1 ;Do HL a DE adresy dvou bufferů s jejich
        LD (HDEPTR),HL ;uložením do obl.dat
        LD DE,HIDE2
        LD (ODEPTR),DE
INITLP SUB A ;Vynulování obsahu reg.A pro následující
        LD (HL),A ;vynulování obsahů obou bufferů
        LD (DE),A
        INC HL
        INC DE
        DJNZ INITLP
        LD A,(LENGTH) ;Následuje určení počtu BCD číslic a
        LD L,A
        LD H,0
        ADD HL,HL
        INC HL ;jeho zvýšení o 1
        LD (COUNT),HL ;Uložení výsledku do obl.dat

        LD HL,(DSADR) ;Následuje test, zda dělenec není nulový
        LD A,(LENGTH)
        LD B,A
        SUB A ;Vynulování obsahu reg.A
        OR (HL) ;Operace OR s každým bajtem dělence
        INC HL
        DJNZ DVO1
        OR A ;Test na nulu
        JR Z,EREXIT ;Je-li výsledek nulou, skok na EREXIT
        SUB A ;Vynulování obsahu reg.A i
        LD (NDIGIT),A ;pozice "další číslice" v obl.dat

DVLOOP LD HL,(DVADR) ;Do HL adr.dělence
        CALL RLARY ;Volání rotace pole dělence a bud.výsl.
        LD HL,(COUNT) ;Zmenšení čítače BCD číslic o 1
        DEC HL
        LD (COUNT),HL ;a uložení jeho nové hodnoty do obl.dat
        LD A,H
        OR L ;Test, zda už není nulový
        JR Z,OKEXIT ;Když AND, skok na OKEXIT
        LD HL,(HDEPTR) ;Adr.1.bufferu pro rotaci vlevo - nej-
        CALL RLARY ;nižší číslice bufferu bude vyšší čís-
        SUB A ;licí bajtu NDIGIT. Vynulování reg.A
        LD (NDIGIT),A ;a její uložení do obl.dat

SUBLP LD HL,(DSADR) ;Do HL adr.dělitele
        LD DE,(HDEPTR) ;Do DE adr.1.bufferu
        LD BC,(ODEPTR) ;Do BC adr.2.bufferu
        LD A,(LENGTH) ;Do A počet bajtů čísel
        LD (CNT),A ;Inic.čítače vnitřní smyčky INNER
        OR A ;Vynulování CY

;Zjištění kolikrát se dělitel "vejde" do 1.bufferu. Při návratu
;ze smyčky vyšší číslice bajtu NDIGIT je další výslednou číslicí
;a obsah 1.bufferu je momentálně zůstatkem.

INNER LD A,(DE) ;Další bajt dělence do reg.A
        SBC A,(HL) ;a odečtení bajtu dělitele
        DAA ;Dekadická adjustace obsahu reg.A
        LD (BC),A ;Uložení výsledku do 2.bufferu
        INC HL ;Zvýšení všech adres o 1
        INC DE
        INC BC
        LD A,(CNT)
        DEC A ;Zmenšení čítače smyčky o 1
        LD (CNT),A
        JR NZ,INNER ;Zpět na smyčku pro odečet všech bajtů
        JR C,DVLOOP ;Když je rozdíl záporný, skok na DVLOOP
        LD HL,NDIGIT ;Jinak adr.bajtu "další číslice" do HL
        INC (HL) ;Zvýšení adr.NDIGIT o 1
        LD HL,(HDEPTR) ;Do HL adr.1.bufferu
        LD DE,(ODEPTR) ;Do DE adr.2.bufferu
        LD (HDEPTR),DE ;Přepnutí bufferů
        LD (ODEPTR),HL
        JR SUBLP ;Pokračuj, dokud výsledek není záporný

OKEXIT OR A ;Vynulování CY pro signal.platného výsl.
        RET ;Návrat

EREXIT SCF ;CY=1, výsledek chybný
        RET ;Návrat

```

;Rotace pole o 1 číslici vlevo. V HL 1.adr.pole. Nižší číslice bajtu adresy NDIGIT je rotována. Na výstupu je pole rotované doleva "skrz" nižší číslici bajtu z adr.NDIGIT.

```
RLARY LD A,(LENGTH) ;Délka pole v bajtech do B
      LD B,A
      LD A,(NDIGIT) ;Obsah adr.NDIGIT do reg.A

SHIFT RLD ;Rotace BCD číslice doleva
      INC HL ;Na další adr.pole
      DJNZ SHIFT ;Pokračuj, dokud nejsou rotovány všechny
      LD (NDIGIT),A ;čísllice. Ulož novou "další číslici"
      RET ;Návrat ze subrutiny

LENGTH DEFS 1 ;Počet bajtů v poli
NDIGIT DEFS 1 ;Další číslice v poli
CNT DEFS 1 ;Čítač odečítací smyčky INNER
DVADR DEFS 2 ;1.adr.dělece
DSADR DEFS 2 ;1.adr.dělitele
HDEPTR DEFS 2 ;Adr.1.bufferu
ODEPTR DEFS 2 ;Adr.2.bufferu
COUNT DEFS 2 ;Čítač smyčky DVLOOP
HIDE1 DEFS 255 ;Pole 1.bufferu
HIDE2 DEFS 255 ;Pole 2.bufferu
```

12/ 4-bajtové binární dělení

Rutina nebere v úvahu znaménka. Dělí čísla v maximální délce 32 bitů. Celočíslný výsledek je rovněž 4-bajtový, stejně tak i zůstatek dělení. Při nulovém děliteli okamžitý návrat s CY=1.

Algoritmus:

Vynulování 32-bitového pole MP (meziprojektu)

32 krát proběhne

vynásobení dělece MP dvěma,

vynásobení pole MP rovněž dvěma a přičtení bitu přenosu z násobení dělece

Když je obsah pole MP větší než dělitel nebo jemu roven, odečti dělitele od pole MP a dělece zvýš o 1

Celočíslný výsledek je na místě dělece, zůstatek v poli MP

Délka: 53 bajtů

Počet cyklů: max.11628

Vstup: v IX 2 vyšší, v IV 2 nižší bajty dělece

v BC 2 " v DE 2 " dělitele

Výstup: Při CY=1 byl dělitel=0; v A je nula, jinak beze změn
Při CY=0 je celočíselný výsledek v IX a IV, zůstatek v BC a DE (vyšší bajty jsou v IX a BC); A=0, v HL je vstupní hodnota DE.

Použité registry: IV,IX,HL,BC,DE,AF

```
DIV4 LD A,B ;Zjištění, zda jsou všechny bajty děli-
      OR C ;tele nulové
      OR D
      OR E
      SCF
      RET Z ;Když AND, návrat s CY=1
      LD A,32 ;Bitový čítač
      LD HL,0 ;Vynulování 4 bajtů pro ukládání zůstat-
      PUSH HL ;ku (HL a (SP))

TOP ADD IV,IV ;Posunutí obsahu dělece o 1 bit nahoru
      EX (SP),IX ;Posuv jde přes zásobník. Ve výsledku
      EX (SP),HL ;jde o násobení dvěma, přičemž nejvyšší
      ADC HL,HL ;bit se přesune do CY
      EX (SP),HL
      EX (SP),IX

      ADC HL,HL ;Totéž pro zůstatek, ale s tím, že CY z
      EX (SP),HL ;minulého posuvu se objeví v zůstatku
      ADC HL,HL
      EX (SP),HL

      OR A ;Vynulování CY
      SBC HL,DE ;Odečtení dělitele od zůstatku
      EX (SP),HL
      SBC HL,BC ;CY=1, když je výsledek záporný
      EX (SP),HL
      INC IV ;Přičtení 1 k výsledku
      JR NC,WENT ;Když CY=0, vše O.K., skok na WENT

      ADD HL,DE ;Když byl dělitel větší než dělece,
      EX (SP),HL ;přičti jej zpět pro obnovu stavu zů-
      DEC IV ;statku a zpětně odečti 1 od IV
```

```

WENT DEC A ;Zmenši čítač bitů o 1
JR NZ,TOP ;Zopakuj pro všech 32 bitů
EX DE,HL ;Nižší 2 bajty zůstatku do DE,
POP BC ;vyšší do BC
OR A ;Vynulování CY; výsledek platný
RET ;Návrat
    
```

13/ Dvojková komplementace (negace) 4-bajtového čísla

Číslo je uloženo v zásobníku. Svůj nejvyšší bajt má na nejvyšší adrese. Na těchto adresách bude jeho výsledný dvojkový doplněk se znaménkem v CY. Indikátor Z=0, když je číslo nulové. Nelze negovat číslo 80000000H, které v rozsahu 4 bajtů nemá doplněk (viz minulá učebnice o dvojkové komplementaci).

Délka: 23 bajtů
Počet cyklů: 255
Použité registry: F

```

SNEG4 PUSH HL ;Uchova původního obsahu registrů HL,
      PUSH BC ;BC a A
      LD C,A
      LD HL,6 ;Nejnižší bajt je o 6 adres zásobníku
      ADD HL,SP ;výš, převed na ni reg.HL
      LD B,4 ;Čítač počtu bajtů čísla
      OR A ;Vynulování CY

SNLP LD A,0 ;Vynulování obsahu reg.A bez změny CY
     SBC A,(HL) ;Odečet 0-Býtý bajt-CY
     LD (HL),A ;Výsledek zpět do zásobníku
     INC HL ;Na další adresu s dalším bajtem
     DJNZ SNLP ;Pokračuj pro všechny 4 bajty
    
```

; Pouze při nulovém čísle bude CY i reg.A=0, po RLA pak i Z=0

```

RLA ;SI znaménka do CY
LD A,C ;Obnovení uschovaných registrů
POP BC
POP HL
RET
    
```

14/ 4-bajtové násobení

Rutina bere v úvahu znaménka. Násobí dvě celá DK čísla zdědí 31 bitů (32.bit je znaménko). Při přetečení (přesahu délky výsledku nebo vložení čísla 80000000H) je indikována chyba. Rutina používá jako subrutinu SNEG4 (bod 13/ výše).

Algoritmus:

```

Je-li násobitel menší než 0, negace
Je-li i potom menší než nula, chyba
Opakuj pro násobenec
  Vynuluj 32-bitové pole meziprojektu MP
  a nastav čítač 32 bitů
  Dokud čítač není nulový, opakuj:
    MP=MP*2
    Násobitel=násobitel*2
    Při CY=1 MP=MP+násobenec
    Nastane-li přetečení, vystup ze smyčky pro chyb.sign
    Sniž čítač
  Když nenastane přetečení, neguj jen záporný výsledek
    
```

Délka: 99 bajtů

Počet cyklů: max.6264

Vstup: v IX 2 vyšší, v IV 2 nižší bajty násobitele
v BC 2 " " " " násobenec

Výstup: Při Z=0 násobení kompletní, výsledek v BC,DE, v CY je znaménko. Při Z=1 chyba - tehdy platí, že při CY=0 došlo k přetečení, tedy výsledek přesahuje délkou 32 bitů, při CY=1 bylo na vstupu číslo 80000000H. Při chybě zůstává obsah BC,DE nezměněn. Obsah IX,IV je vždy beze změny.

Použité registry: IV,IX,HL,DE,BC,F

```

SMUL4 PUSH IX ;IX do HL přes zásobník, v H je znaménko
      EX (SP),HL
      PUSH HL ;Uchování všech registrů
      PUSH IV
      PUSH BC
      PUSH DE
      LD L,A ;Uchova reg.A
      LD A,H ;Operace XOR mezi znaménky obou čísel
      XOR B
      LD A,L
      PUSH AF ;Uchova pův.hodn.reg.A a znaménka
      PUSH IX ;Uložení násobitele do zásobníku pro
      PUSH IV ;možnou potřebu provedení negace
      ADD HL,HL ;Posun HL o 1 bit vlevo, znaménko do CY
      CALL C,SNEG4 ;Když je číslo záporné, provést negaci
    
```

	POP	IX	;Obnovení hodnot násobitele již negova-
	POP	IX	;ného na 4 adresách zásobníku
	JR	C, SMERR	;Když je CY stále log.1, návrat s chybou
	PUSH	BC	;Totéž opakovat pro násobence
	PUSH	DE	
	RL	B	
	CALL	C, SNEG4	
	POP	DE	
	POP	BC	
	JR	C, SMERR	
	SBC	HL, HL	;Vynulování pole budoucího výsledku,
	PUSH	HL	;jeho uschování
	EX	(SP), IX	;a přenos do IX
	LD	A, 32	;Čítač bitů
SMLP	ADD	IX, IX	;Posun pole výsledku vlevo
	ADC	HL, HL	;Násobení dvěma s CY; byl-li nejvyšší
	JR	C, SMOVFW	;bit HL=1, skok na SMOVFW
	ADD	IX, IX	;Posun pole násobitele (je v (SP) a HL)
	EX	(SP), HL	
	ADC	HL, HL	;Opět posun s CY
	EX	(SP), HL	
	JR	NC, SMLPT	;Byl-li nejvyš. bit násobitele nula, skok
	ADD	IX, DE	;Jinak přičtení násobence
	ADC	HL, BC	
	JR	C, SMOVFW	;Příprava na návrat při přetečení
SMLPT	DEC	A	;Zmenšení čítače bitů o 1
	JR	NZ, SMLP	;Vykonej pro všech 32 bitů
	LD	A, H	;Bit 32 výsledku
	ADD	A, A	;do CY pro test přetečení (je-li log.1)
SMOVFW	POP	BC	;Odhoz nepotřebných bajtů
	CCF		;Komplementace CY
	JR	NC, SMERR	;Při CY=0 došlo k přetečení, skok
	POP	AF	;Při CY=1 O.K., obnova A a znaménka
	POP	DE	;Výměna pole násobence za výsledek
	EX	(SP), HL	
	PUSH	IX	
	CALL	M, SNEG4	;Negace, když výsledné znaménko záporné
	LD	H, 0	;Přednastavení H, aby po DEC H byl Z=0
	JR	SMEND	;Skok na SMEND
SMERR	POP	HL	;Obnova reg.A
	LD	A, H	
	LD	H, 1	;Přednastavení H pro Z=1
SMEND	DEC	H	;Signalizace chyby či správnosti Z=1/0
	POP	DE	;Výložení výsledku (při správnosti)
	POP	BC	;nebo násobence (při chybě)
	POP	IX	;Výložení násobitele
	POP	IX	
	POP	HL	;Obnova HL
	RET		;Návrat

15/ 4-bajtové dělení

Rutina bere v úvahu znaménka. Dělit můžeme celá čísla v rozsahu 31 bitů, když 32. bit je znaménkem. Nedošlo-li k chybě, je celočíselný výsledek umístěn v původních registrech dělitele, zůstatek v registrech dělece. Pro negaci je opět využita rutina SNEG4 z bodu 13/.

Délka: 120 bajtů

Počet cyklů: max. 8992

Vstup: v IX 2 vyšší, v IX 2 nižší bajty dělece

v BC 2 " v DE 2 " dělitele

Výstup: Při Z=0 dělení kompletní, výsledek v BC, DE, v CY je znaménko. Zůstatek v IX, IV. Při Z=1 chyba - tehdy platí, že při CY=0 byl dělitel nulový, při CY=1 bylo na vstupu číslo 80000000H. Při chybě zůstává obsah BC, DE, IX, IV beze změny.

Použité registry: IV, IX, HL, DE, BC, F

SDIV4	PUSH	IX	
	EX	(SP), HL	;Výměna IX s HL
	PUSH	BC	;Uschova dělitele pro případ chyby
	PUSH	DE	
	LD	L, A	;Uschova A
	LD	A, H	;Zjištění znaménka dělece
	XOR	B	;operací OR
	LD	A, H	
	RLA		;Znaménko do CY
	LD	A, L	;Obnova obsahu reg.A
	PUSH	AF	;a jeho uschova spolu s SI znaménka

	PUSH BC	;Uložení dělitele do zásobníku
	PUSH DE	
	CALL SNEG4	;Když je výsledek první negace záporný,
	CALL C,SNEG4	;neguj ještě jednou, nutná abs.hodnota
	POP DE	;Obnovení obsahu dělitele
	POP BC	
	JR Z,SDERR	;Když je nulový,
	JR C,SDERR	;nebo je to 80000000H, skok na SDERR
	PUSH IX	;Uložení dělenec do zásobníku
	PUSH IY	
	ADD HL,HL	;Znaménko do CY
	CALL C,SNEG4	;Když dělenec záporný, neguj pro abs.h.
	POP IY	;Obnovení obsahu dělenec
	POP IX	
	JR C,SDERR	;Je-li dělenec dále záporný, skok
	POP HL	;Vše O.K., zbav se dělitele v zásobníku
	POP AF	
	EX (SP),HL	;Ale znam.a pŕv.obsah reg.A ulož zpět
	LD HL,0	;Vynulování 32 bitů pole zůstatku
	PUSH HL	
	LD A,32	;Čítač 32 bitů
SDILP	ADD IX,IX	;Posun vyšších bajtů dělenec vlevo do
	ADC HL,HL	;nižší části pole zůstatku zprava
	RLA	;CY do bitu 0 reg.A
	SBC HL,DE	;Odečet nižších bajtů dělitele od nižší
	RRA	;části zůstatku; bit 0,A zpět do CY a
	EX (SP),HL	;přenos odečtu do bitu 7,A; do HL vyšší
	ADC HL,HL	;část pole zůstatku, CY do nejnižšího
	ADD IY,IY	;bitu pole a jeho posuv vlevo; posun
	JR NC,SDHIS	;nižší části dělenec; při CY=1 nastavit
	INC IX	;nejnižší bit reg.IX na log.1
SDHIS	RLCA	;Přenos odečtu do CY a do bitu 0,A
	SBC HL,BC	;Odečet vyšší části dělitele
	JR NC,SDQSET	;Když CY=0, dělitel menší, skok
	SRL A	;Bit 0,A do CY a vrácení A jeho pŕv.hod.
	ADC HL,BC	;Zpětné přičtení vyšší i nižší části
	EX (SP),HL	;zůstatku pro obnovení jeho pŕv.stavu
	ADD HL,DE	
	JR SDILPT	;Skok na SDILPT
SDQSET	INC IY	;Nejnižší bit IY bude 1
	EX (SP),HL	;Obnovení nižší části zůstatku
	RRA	;i pŕv.obsahu reg.A
SDILPT	DEC A	;Zmenšení čítače bitů o 1
	JR NZ,SDILP	;Opakuj smyčku pro všech 32 bitů
	POP BC	;Zůstatek v BC,HL
	POP DE	;Pŕv.uložené reg.AF do DE
	PUSH IX	;Celočís.výsledek uložen pro ev.negaci
	PUSH IY	
	PUSH BC	;Totéž se zůstatkem
	PUSH HL	
	SRA E	;Znaménko zůstatku do CY
	CALL C,SNEG4	;Když je záporné, provést negaci
	POP IY	;Zůstatek se správným znaménkem v IX,IY
	POP IX	
	SLA E	;Znaménko celoč.výsledku do CY
	CALL C,SNEG4	;Když je záporné, provést negaci
	LD E,A	;Vynulování reg.E pro následující sig-
	JR SDEND	;nalizaci pomocí indikátoru Z
SDERR	POP DE	;Uložení AF do DE
	LD E,1	;Nastavení reg.E pro signalizaci
SDEND	LD A,D	;Obnova obsahu reg.A
	DEC E	;Zmenšení obsahu reg.E kvůli Z
	POP DE	;DE,BC je výsledek při Z=0, nebo
	POP BC	;dělenec při Z=1
	POP HL	;Obnova obsahu HL
	RET	;Návrat

16/ Odmocňování

Následující rutiny budou poněkud obtížnější pro pochopení těmi, kteří včas seznali, že matematici patří k sociálně nejslabší skupině obyvatelstva. Chcete-li se přesto dostat na kloub věci, proveďte si detailní analýzu pohybu bitů v bajtech, podstata se vám nakonec vyjeví. Detailní písemný rozbor rutin "až na bit" by v učebnici zabral neúměrně mnoho místa. Rámcová analýza je u vysvětlení principu funkcí třetích odmocnin pod bodem c/ (takže IF váhavost THEN GOSUB c/).

a/ druhá odmocnina 16/15-bitového čísla

První rutina hledá kvadratický kořen dvoubajtového celého čísla. Má dva vstupy - pro kladná 15-bitová čísla, u nichž je nulový bit 15 zvažován jako znaménko plus a pro absolutní hodnoty 16-bitových čísel. Celočíslná část obsahuje 8 bitů, zůstatek (číslo-kořen**2) 9 bitů.

Algoritmus:

Vynuluj meziprodukt

Pro každý z 8 bitů výsledku opakuj

 menšitel=meziprodukt*2**((čítač*2)+1*2**((čítač-1)*2)

 Když je menšitel menší než číslo, či jemu roven, pak

 číslo=číslo-menšitel

 meziprodukt=meziprodukt*2+1

 jinak meziprodukt=meziprodukt*2

Převeď výsledkek do správných registrů a ukonči

Délka: 36 bajtů

Počet cyklů: max.847

Vstup: v HL 16-bitová hodnota

Výstup: S bitem 15 jako znaménkem - při Z=0 byla vložena záporné číslo, všechny registry beze změny; při Z=1 totéž jako níže.

Při plné hodnotě (bez znaménka) je vždy Z=1, A=B=0.

8-bitový celočísl.výsledek je v reg.L, 9-bitový zůst.v DE; bity 7 až 1 reg.D jsou nulové.

Použité registry: HL,DE,B,AF

;Vstup pro číslo se znaménkem

```
SQR15 BIT 7,H      ;Je číslo záporné?
      RET NZ       ;Když AND, okamžitý návrat, Z=0
```

;Vstup pro číslo bez znaménka

```
SQR16 LD D,0      ;Inic.reg.D a E; nastavení bitu 6,E na
      LD E,#40     ;log.1 (o 2 bity níž, než mezipr.v D)
      LD A,L       ;Vytvoření 24-bitového pole A,L,H (kořen
      LD L,H       ;bude v D, zůstatek v H a bitu 0,A)
      LD H,D
      LD B,8
      OR A         ;Čítač osmi bitů
      ;Vynulování CY
```

```
SQR1P SBC HL,DE    ;Odečet meziproduktu
      JR NC,BTIN   ;Když O.K., skok na BTIN
      ADD HL,DE    ;Jinak zpět s nastavením CY na log.1
```

```
BTIN  CCF          ;Komplementace CY a jeho posun do bitu 0
      RL D         ;meziproduktu*2 při CY=0; či +1 při CY=1
      ADD A,A      ;Posun čísla o 2 bity vlevo, aby byly
      ADC HL,HL    ;operandy při příštím odečtu na správném
      ADD A,A      ;místě
      ADC HL,HL
      DJNZ SQR1P   ;Pokračuj pro všech 8 bitů výsledku
```

```
      RLA         ;9.bit zůstatku z CY do bitu 0,A
      LD L,D       ;Převod výsledku do správných registrů
      LD E,H
      LD D,A
      LD H,B
      XOR A        ;Z=1 pro signalizaci platného výsledku
      RET         ;Návrat
```

b/ druhá odmocnina 32/31-bitového čísla

Další rutina hledá kvadratický kořen čtyřbajtového celého čísla. Má dva vstupy - pro kladná 31-bitová čísla, u nichž je nulový bit 31 zvažován jako znaménko plus, a pro absolutní hodnoty 32-bitových čísel. Celočíslná část výsledku obsahuje 16 bitů, zůstatek (číslo-kořen**2) 17 bitů.

Algoritmus je tentýž jako u předcházející rutiny:

Vynuluj meziprodukt

Pro každý z 16 bitů výsledku opakuj

 menšitel=meziprodukt*2**((čítač*2)+1*2**((čítač-1)*2)

 Když je menšitel menší než číslo, či jemu roven, pak

 číslo=číslo-menšitel

 meziprodukt=meziprodukt*2+1

 jinak meziprodukt=meziprodukt*2

Převeď výsledkek do správných registrů a ukonči

Délka: 50 bajtů
 Počet cyklů: max. 2792
 Vstup: v BCDE 32-bitová hodnota
 Výstup: S bitem 31 jako znaménkem - při Z=0 byla vloženo záporné číslo, všechny registry beze změny; při Z=1 totéž jako níže.
 Při plné hodnotě (bez znaménka) je vždy Z=1, BC=0.
 16-bitový celočísl. výsledek je v DE, 17-bitový zůst. v AHL;
 bity 7 až 1 reg. A jsou nulové.

Použité registry: HL, DE, BC, AF

;Vstup pro číslo se znaménkem

```
SQR31 BIT 7,B      ;Je číslo záporné?
      RET NZ       ;Když AND, okamžitý návrat, Z=0
```

;Vstup pro číslo bez znaménka

```
SQR32 LD  A,B      ;Uložení 32-bitového čísla na nižší baj-
      PUSH DE      ;ty 48-bitového pole HL,A,C,IX
      EX  (SP),IX  ;IX do zásobníku
      LD  D,0      ;Vynuluj menšitele i coby meziproduct
      LD  E,D      ;
      LD  H,D      ;a rovněž 2 nejvyšší bajty 48-bit.pole
      LD  L,D      ;
      LD  B,16     ;Čítač 16 bitů výsledku

DSQRLP SUB #40     ;Proveď odečet meziproductu
      SBC HL,DE
      JR  NC,DBIN  ;Když odečet D.K., skok na DBIN
      ADD A,#40    ;Jinak vše vrať do původního stavu
      ADC HL,DE

DBIN  CCF          ;Komplementace CY pro přičtení 0 nebo 1
      RL  E        ;k reg.E spolu s posunem DE o 1 bit
      RL  D        ;doleva, tj. násobení dvěma
      ADD IX,IX    ;Posun čísla v poli o 2 bity vlevo, aby
      RL  C        ;se meziproduct objevil na správném
      RLA         ;místě
      ADC HL,HL
      ADD IX,IX
      RL  C
      RLA
      ADC HL,HL
      DJNZ DSQRLP ;Opakuj pro všech 16 bitů výsledku

      RLA         ;17.bit zůstatku z CY do bitu 0,A
      CP  A        ;Z=1 pro signalizaci platného výsledku
      POP IX      ;Odhoz ze zásobníku pro přístup
      RET        ;k adrese návratu
```

c/ třetí odmocnina 16-bitového čísla

Metoda použitá k výpočtu třetí odmocniny je podobná té, která je užitá u rutin pro výpočet odmocniny druhé. Jsou tu však dva rozdíly. V každém průchodu smyčkou se pracuje nikoli se dvěma, ale se třemi bity. A formovaný kořen nemůže být použit jako menšitel. Následuje aritmetický pohled na algoritmus druhé i třetí odmocniny, jak je použit v uvedených rutinách. Z tohoto algoritmu můžete vyjít i při konstrukci rutin pro další výpočty.

V algoritmu druhé odmocniny je vstupní číslo posouváno o dva bity vlevo do připraveného pole. Této části "naposouvání bitů" říkáme virtuální vstup, označme si jej jako x. Výsledek operace každého průchodu smyčkou rutiny je meziproductem, řekněme s, se současným "nárůstem" zůstatku r. Po posledním průchodu smyčkou má meziproduct hodnotu celočíselné části výsledku, mezizůstatek zůstatku.

virtuální vstup = x
 meziproduct = s
 mezizůstatek = r

Vše můžeme přepsat do rovnice

$$s**2 = r + x$$

Z pohledu funkce rutiny uvedené relace upravíme. Ve smyčce jsou posunutím vlevo do virtuálního vstupu přidány 2 bity skutečného vstupního čísla. Toto dvoubitové číslo označíme y . Příští nejnižší bit mezivýsledku bude d . Mezivýsledek bude tedy $2s + d$ (posun vlevo, který je násobením dvěma!). A vstup? Ten se posunul dvakrát vlevo, tedy byl de facto vynásoben čtyřmi! Mění se i mezizůstatek. Pak můžeme psát:

```
virtuální vstup = 4x + y
mezivýsledek = 2s + d
mezizůstatek = q
```

Rovnice pak bude vypadat takto:

$$(2s + d) * 2 + q = 4x + y$$

Zkombinováním obou rovnic eliminujeme x . Mocnina d bude rovna d , protože jde o jeden bit. Když bude mít hodnotu $\log_2 0$, pak vyplyne tento zápis rovnice:

$$q = 4r + y$$

Při $d=1$ to bude:

$$q = 4r + y - (4s + 1)$$

Z toho je patrné, že požadovaný menšítelem musí mít hodnotu $4s+1$, což je předchozí mezivýsledek posunutý dvakrát vlevo a zvětšený o 1.

Obdobnou analýzu můžeme uplatnit u hypotetického algoritmu pro odmocňování třetího stupně.

```
virtuální vstup = x
mezivýsledek = c
mezizůstatek = r
```

A pochopitelně: $c * 3 + r = x$

Do připraveného pole budou nasouvány 3 bity, proto:

```
virtuální vstup = 8x + y
mezivýsledek = 2c + d
mezizůstatek = q
```

Kombinací odpovídajících rovnic dostaneme:

$$(2c + d) * 3 + q = 8x + y$$

Při nulovém bitu d :

$$q = 8r + y$$

Při $d=1$:

$$q = 8r + y - (12c * 2 + 6c + 1)$$

Tvar menšítele ve druhém případě poněkud vrásčí čelo. Ale jen do té doby, než si řekneme, že v rutině budeme pracovat s nenulovou hodnotou c vždy maximálně 1, tedy $c * 2$ bude rovněž 1. Rozvinutí této finesy ukazují následující 2 rutiny.

Algoritmus:

```
Vynuluj pole mezivýsledku, menšítele a zůstatku
Pro 6 bitů výsledku nastav čítač na 6
Je-li čítač menší než 6, pak
    posuň číslo do pole zůstatku o 2 bity
    posuň " " " " " " " " 1 bit
    Když je zůstatek menší než menšítel, pak
        zůstatek=zůstatek-(menšítel+1)
        mezivýsledek=mezivýsledek*2+1
        menšítel=menšítel*4+mezivýsledek*18
    Jinak mezivýsledek=mezivýsledek*2
        menšítel=menšítel*4+mezivýsledek*6
```

Délka: 64 bajtů

Počet cyklů: max.1509

Vstup: v BC 16-bitová hodnota

Výstup: v C 6-bitová celočíselná část výsledku, B=0
v HL 13-bitový zůstatek

Použité registry: HL,BC

CURT16	PUSH AF	;Uchova registrů do zásobníku a odmoc-
	PUSH BC	;ňovaného čísla do IX
	EX (SP), IX	
	PUSH DE	
	LD A, 6	;Čítač bitů celočísl.výsledku
	LD BC, 0	;Vynulování pole výsledku,
	LD D, C	;menšíte
	LD E, C	
	LD H, C	;a zůstatku
	LD L, C	
	JR START	;Poprvé posuň jen 1 bit
RPEAT	ADD IX, IX	;Posun čísla
	ADC HL, HL	;do pole zůstatku o 1...
	ADD IX, IX	
	ADC HL, HL	;...o 2...
START	ADD IX, IX	
	ADC HL, HL	;...a nakonec 3 bity
	RL C	;Rotace pole budoucího výsledku
	SCF	
	SBC HL, DE	;Odečet menšíte od pole zůstatku
	JR C, ABDAK	;Když je výsledek záporný, obnov stav
	INC C	;Jinak zvyš obsah pole bud.výsledku o 1
	EX DE, HL	;Menšítel do HL kvůli součtům
	ADD HL, BC	;Nový menšítel=starý menšítel
	ADD HL, BC	;*4+pole bud.výsledku*16 (starý menš.*2
	ADD HL, BC	;+nové pole výsl.*9 vypočteno zde; další
	ADD HL, BC	;násobení dvěma
	ADD HL, HL	
	ADD HL, BC	
	JR SHIFT	;je v části SHIFT)
ABDAK	ADC HL, DE	;Obnova obsahu zůstatku
	EX DE, HL	;Menšítel do HL pro součty
	OR A	;Vynulování CY
	SBC HL, BC	;Nový menšítel=starý menšítel
	ADD HL, HL	;*4
	SBC HL, BC	;--nové pole výsl.*6
SHIFT	ADD HL, HL	;Doplnění nového menšítele
	EX DE, HL	;a jeho převod do DE
	DEC A	;Zmenšení čítače smyčky o 1
	JR NZ, RPEAT	;Vykonej pro všech 6 bitů výsledku
	POP DE	;Obnova obsahů registrů
	POP IX	
	POP AF	
	RET	;Návrat s výsl.v C. zůst.v HL

d/ třetí odmocnina 32-bitového čísla

Poznámka - jak minulá, tak tato rutina počítá se všemi bity vstupního čísla jako binárními číslicemi. Pokud budeme chtít odmočkovat číslo, u nějž je nejvyšší bit znaménkem, předřadíme test, jaký je na začátku obou rutin pro odmočkování druhého stupně, abychom zabránili prostupu čísla záporného.

Algoritmus:

Algorithm:

Vynuluj pole meziprojektu, zůstatku a menšitele
Pro 11 bitů výsledku opakuj

Když je čítač bitů výsledku menší než 11, pak
posuň číslo do zůstatku o 1 bit
" " " 2 bity

Když je zůstatek větší než ^{z díly} menšitel, pak

```
zdstatek=zdstatek-(menšitel+1)
```

```
meziprodukt=meziprodukt*2+1
```

```
menšitel=menšitel*4+meziprodukt*18
```

```
jinak meziprodukt=meziprodukt*2
```

menyitai=menyitai*4-meziprodukt*6

Délka: 97 bajtů

Počet cyklů: max. 5480

Vstup: v BCDE 32-bitové číslo

Výstup: v DE 11-bitová celočísl.část výsledku

vystup: v BE 11-bitová ležba; EASE vy
v AHL 23-bitový zůstatek; BC=0

Použité registry: HL, DE, BC, AF

CURT32	PUSH	BC	;32-bitové číslo do reg.IX,IV a jejich
	EX	(SP),IX	;uložení do zásobníku

PUSH DE

EX (SP).IY

LD BC, #0000

LD BL
LD A.C

1000

1000

LD H, C

LD L, C

PUSH HFO

JR BEGIN

;32-bitové číslo do reg.IX,IV a jejich
uložení do zásobníku

```

;V reg.B čítač 11 bitů výsledku, další
;registry vynulovat, tj.pole zůstatku
;AHL a pole menšitele CDE

```

; Vynul. prac. prostoru mezivýsl. v zásobn.
; Poprvé posun jen dvou bitů

AGAIN	CALL	SLEFT	;Posun čísla o 3 bity vlevo, aby operan-
BEGIN	CALL	SLEFT	;dy odečtu byly větší sobě na správném
	CALL	SLEFT	;místě
	SCF		
	SBC	HL, DE	; "Odběr" menšítele+1 ze zůstatku
	SBC	A, C	
	JR	C, NOSUB	; Když je výsledek záporný, obnovit stav
	EX	(SP), HL	; Je-li kladný, výměna zůstatku s posled-
	ADD	HL, HL	; ním meziproduktem; nový meziprodukt=
	INC	HL	; starý mezipr.*2+1
	PUSH	HL	; Uložení nového mezipr.do zásobníku
	ADD	HL, HL	; Meziprodukt*4
	ADD	HL, HL	
	EX	DE, HL	; Mezipr.do DE, menšítel do HL
	ADD	HL, DE	; Menšítel+meziprodukt
	JR	NC, NOINC	; Když CY=0, skok na NOINC
	INC	C	; Když CY=1, zvýš obsah nejv.bajtu menš.
NOINC	ADD	HL, HL	; Celý menšítel má teď hodnotu=starý men-
	RL	C	; šítel*2+meziprodukt*8
	POP	DE	; Následuje přičtení meziprodktu k men-
	ADD	HL, DE	; šíтели, který teď je starý menš.*2+me-
	JR	NC, NEXT	; zipr.*9; zdvojnásobení v části NEXT,
	INC	C	; aby nový menš.=starý m.*4+mezipr.*18
	JR	NEXT	
NOSUB	ADC	HL, DE	; Obnovení min.stavu zůstatku (menšítel
	ADC	A, C	; byl příliš velký)
	EX	(SP), HL	; Výměna zůstatku s posl.meziproduktem
	ADD	HL, HL	; Nový mezipr.=starý mezipr.*2
	EX	DE, HL	; Menšítel do HL
	SBC	HL, DE	; Odečet meziprodktu
	JR	NC, NODEC	; Když CY=0, skok na NODEC
	DEC	C	; Při CY=1 zmenšení obs.nejv.bajtu menš.
NODEC	ADD	HL, HL	; Starý menš.*2-mezipr.*2
	RL	C	
	SBC	HL, DE	
	JR	NC, NEXT	; Opakuj, dokud CY=0, aby nový menš.byl=
	DEC	C	; st.menš.*4-mezipr.*6
NEXT	ADD	HL, HL	; Kompletace nového menšítele v poli CHL,
	RL	C	
	EX	DE, HL	; jeho 2 nejnižší bajty do DE,
	EX	(SP), HL	; mezipr.do zásobníku, zůst.do HL
	DJNZ	AGAIN	; Opakuj pro všech 11 bitů výsledku
	POP	DE	; Celočís.část výsledku ze zásob.do DE
	LD	C, B	; BC=0
	POP	IX	; Obnovení obsahu registrů
	POP	IX	
	RET		; Návrat s výsl.v DE, zůst.v AHL

; Subrutina pro nasunutí dalšího bitu čísla do pole zůstatku

SLEFT	ADD	IX, IX	; Posun 2 vyš.bajtů čísla o 1 bit vlevo
	ADC	HL, HL	; Nejvyšší bit čísla do CY se současným
	RLA		; posunem pole zůst.také o 1 bit vlevo
	ADD	IX, IX	; Posun 2 nižších bajtů čísla vlevo
	RET	NC	; Návrat, když bez přenosu bitu s log.1 do
	INC	IX	; nejniž.bitu vyšších 2 bajtů čísla, jinak
	RET		; je větší o 1 a návrat ze subrutiny

17/ Multibajtový aritmetický posun vpravo

Následujících 5 rutin patří do oblasti bitových manipulací. V podstatě však reprezentují násobení a dělení (vyjma rotací). Proto je můžeme využít i ve výpočtech. Další užití mají např. při animaci počítačové grafiky.

První rutina provádí uvedený posun o určený počet bitů v poli o maximální délce 255 bajtů. Po skončení posunu je v CY stav poslední posunutého nejnižšího bitu celého pole. Nejnižší bajt pole je na nejnižší adrese. Operace posunu o 1 bit pole jde od jeho nejvyššího bajtu směrem k nejnižšímu a opakuje se pro žádaný počet posunů. Protože jde o aritmetický posun, rutina zajišťuje, aby nejvyšší bit pole (coby znaménko v něm uloženého čísla) zůstal na svém místě. Při animaci grafiky bude vhodné tento bit posouvat spolu s ostatními.

Délka: 28 bajtů

Počet cyklů: $((34 \cdot \text{délka pole}) + 46) \cdot \text{počet posunů} + 59$

Vstup: v HL nejnižší adresa pole, v B počet bajtů pole,

v C o kolik bitových pozic má být pole posunuto

Výstup: nejvyšší bit pole zůstává nezměněn

v CY je log.stav posledně posunutého, nejniž. bitu pole

Použité registry: HL, DE, BC, AF

```

MPASR LD A,C                ;Je počet posunů celého pole nulový?
      OR A                  ;Když AND, návrat
      RET Z
      LD A,B
      OR A                  ;Je počet bajtů v poli nulový?
      RET Z                ;Když AND, návrat
      LD E,B                ;Vypočet adresy nejvyššího bajtu pole,
      LD D,0                ;počet bajtů pole do DE
      ADD HL,DE              ;Výsledek součtu ukazuje "za pole",
      DEC HL                ;proto o 1 adresu níž

LOOP  LD B,(HL)              ;Nejvyšší bajt do reg.B; při RR (HL) zpět
      RL B                  ;Do CY log.stav nejvyššího bitu pole
      LD B,A                ;B je čítač počtu bajtů pole
      LD E,L                ;Uchova adr.nejvyš.bajtu pole do DE
      LD D,H

ASRLP RR (HL)                ;Provedení rotací o 1 bit
      DEC HL                ;Na další bajt pole na nižší adrese
      DJNZ ASRLP            ;Opakuj pro celé pole

CONT  LD L,E                 ;Obnovení adr.nejvyš.bajtu pole v HL
      LD H,D
      DEC C                 ;Zmenšení čítače posunů celého pole
      JR NZ,LOOP            ;Posunuto o daný počet bit.pozic?
      RET                  ;Když AND,návrat; když NE, zpět na LOOP
  
```

18/ Multibajtový logický posun vlevo

Bitový posun pole s maximální délkou 255 bajtů o žádaný počet bitových pozic. Protože jde o posuv logický, je vždy do nejnižšího bitu pole nasunuta nula. Nejnižší bajt pole je na nejnižší adrese.

Délka: 21 bajtů

Počet cyklů: $((34 \cdot \text{délka pole}) + 27) \cdot \text{počet posunů} + 31$

Vstup: v HL nejnižší adresa pole, v B počet bajtů pole,

v C počet posunů celého pole

Výstup: počet nulových nejvyšších bitů je roven počtu posunů,

v CY je log.stav posledně posunutého, nejvyš. bitu pole

Použité registry: DE, BC, AF

```

MPLSL LD A,C                ;Testy na nulovou délku pole a nulový
      OR A                  ;počet posunů jako výše
      RET Z
      LD A,B
      OR A                  ;Délka pole v A
      RET Z

LOOPL LD E,L                ;Uchova adr.nejniž.bajtu pole do DE
      LD D,H
      LD B,A                ;B je čítač počtu bajtů v poli
      OR A                  ;Vynulování CY

LSLLP RL (HL)                ;Rotace o 1 bit vlevo
      INC HL                ;Na další adresu s dalším bajtem pole
      DJNZ LSLLP            ;Opakuj pro všechny bajty pole
      LD L,E                ;Obnova adresy nejniž.bajtu pole v DE
      LD H,D
      DEC C                 ;Zmenšení čítače posunů celého pole
      JR NZ,LOOPL          ;Když ještě zbývá provést posuny, skok
      RET                  ;Když vše hotovo,návrat
  
```

19/ Multibajtový logický posun vpravo

Posunutí pole s maximální délkou 255 bajtů o požadovaný počet bitových pozic. Nejnižší bajt pole je na nejnižší adrese. Adekvátně logickému posunu je na nejvyšší bit pole nasouvána log.0.

Délka: 26 bajtů

Počet cyklů: $((34 \cdot \text{délka pole}) + 35) \cdot \text{počet posunů} + 59$

Vstup: v HL nejnižší adresa pole, v B počet bajtů pole,

v C počet posunů celého pole

Výstup: počet nulových nejvyšších bitů je roven počtu posunů

v CY je log.stav posledně posunutého, nejniž. bitu pole

Použité registry: HL, DE, BC, AF

```

MPLSR LD A,C      ;Testy na nulový počet bajtů pole a
      OR A         ;pozičních posunů jako výše
      RET Z
      LD A,B
      OR A
      RET Z

      LD E,B       ;Počet bajtů pole do DE
      LD D,0
      ADD HL,DE     ;HL ukazuje na adr."za polem",
      DEC HL        ;proto se musí o 1 zmenšit

LOOPR OR A         ;Vynulování CY
      LD B,A       ;B je čítač počtu bajtů pole
      LD E,L       ;Uschova adr.nejvyš.bajtu pole do DE
      LD D,H

LSRLP RR (HL)      ;Provedení posunu 1 bajtu o 1 bit.pozici
      DEC HL       ;Na další adresu s dalším bajtem pole
      DJNZ LSRLP   ;Opakuj pro všechny bajty pole

      LD L,E       ;Obnova adr.nejvyš.bajtu pole v HL
      LD H,D
      DEC C        ;Snížení čítače posunů celého pole
      JR NZ,LOOPR  ;Když ještě zbývá povést posuny, skok
      RET          ;Vše hotovo, návrat
    
```

20/ Multibajtová rotace vpravo

Rotace pole s maximální délkou 255 bajtů o 1 bitovou pozici vpravo. Jak víte, je čistá rotace jednoho bajtu uzavřeným okruhem, kde je krajní bit přenesen do CY a původní obsah CY do bitu opačné krajní pozice bajtu. Při rotaci vícebajtového pole vpravo bude rotovat jeho nejnižší bit přes CY do jeho bitu nejvyššího.

Délka: 33 bajtů

Počet cyklů: $((34 \cdot \text{délka pole}) + 58) \cdot \text{počet rotací} + 83$

Vstup: v HL nejnižší adresa pole, v B počet bajtů pole,
v C počet rotací celého pole

Výstup: v CY nejnižší bit pole z poslední rotace

Použité registry: IX,HL,DE,BC,AF

```

MPRR LD A,C      ;Dva testy na nulu, jako výše
      OR A
      RET Z
      LD A,B
      OR A
      RET Z

      PUSH HL     ;Adresa nejnižšího bajtu pole do IX
      POP IX
      LD E,B
      LD D,0
      ADD HL,DE   ;Výpočet adresy za koncem pole,
      DEC HL      ;proto její snížení o 1

LOOPRR LD B,(IX+0) ;Do B nejnižší bajt pole, do CY nejn.bit
      RR B        ;a jeho rotace
      LD B,A
      LD E,L
      LD D,H     ;Uschova adresy nejvyš.bajtu pole do DE

RRLP RR (HL)      ;Rotace bajtu o 1 bit
      DEC HL      ;Na další adresu s bajtem pole
      DJNZ RRLP   ;Opakuj pro všechny bajty pole
      LD L,E
      LD H,D
      DEC C       ;Zmenš.čítače počtu rotací celého pole
      JR NZ,LOOPRR ;Opakuj pro zadaný počet rotací
      RET        ;Návrat
    
```

21/ Multibajtová rotace vlevo

Vše je analogické k předchozí rotaci.

Délka: 35 bajtů

Počet cyklů: $((34 \cdot \text{délka pole}) + 58) \cdot \text{počet posunů} + 104$

Vstup: v HL nejnižší adresa pole, v B počet bajtů pole,
v C počet rotací celého pole

Výstup: v CY nejvyšší bit pole z poslední rotace

Použité registry: IX,HL,DE,BC,AF

```

MPRL  LD  A,C          ;Obvyklé dva testy na nulu jako výše
      OR  A
      RET Z
      LD  A,B
      OR  A
      RET Z

      PUSH HL          ;Uložení adresy nejnižšího bajtu pole
      LD  E,B          ;Následuje výpočet nejvyš.bajtu pole
      LD  D,0
      ADD HL,DE
      DEC HL
      PUSH HL
      POP  IX          ;Adr.nejvyš.bajtu pole do IX
      POP  HL          ;Do HL adr.nejniž.bajtu pole

LOOPRL LD  B,(IX+0)    ;Nejvyšší bajt pole do reg.B
      RL  B            ;a jeho rotace; v CY nejvyšší bit pole
      LD  E,A          ;B bude čítač počtu bajtů pole
      LD  E,L
      LD  D,H          ;Uchova adr.nejniž.bajtu pole do DE

RLLP  RL  (HL)         ;Rotace bajtu o 1 bit
      INC HL          ;Na další adresu s bajtem pole
      DJNZ RLLP        ;Opakuj pro všechny bajty pole
      LD  L,E          ;Obnova adr.nejniž.bajtu pole v HL
      LD  H,D
      DEC C            ;Zmenšení čítače rotací celého pole
      JR  NZ,LOOPRL    ;Když čítač nenulový, skok na LOOPRL

```

KAPITOLA 4

Generování pseudonáhodných čísel

Náhodná čísla jsou samostatnou vědní disciplínou. Přestože je počítač přímo přetpán kvantem rychle probíhajícími operacemi s jedničkami a nulami, není tak jednoduché vytvořit program, který by generoval čísla s nejvyšším stupněm náhodnosti. Tento stupeň je u mikropočítače přese všechny jeho operační schopnosti dost nízký. Digitální počítače obecně mohou generovat jen sekvence pseudonáhodných čísel. Dobrý generátor, pracující se 16-bitovými čísly, by měl umět opakovat generovanou sekvenci ne dříve než po každých 65536 různých číslech.

Generování náhodných čísel má význam nejen pro matematiku, ale v mnoha oblastech lidského konání. Pomocí náhodných čísel můžeme testovat třeba "neprůstřelnost" programů. Nebo pevnost materiálů a konstrukci při jejich matematickém modelování (náhodné změny více parametrů současně). Což souvisí s nesmírným významem náhodných čísel při zkoumání statických i dynamických struktur a procesů obecně. Určitá část (nejen herního) softwaru by bez pseudonáhodných čísel vůbec nevznikla.

Aby byla opakující se sekvence pseudonáhodných čísel dostatečně "květnatá", jednotlivá čísla jsou ještě dále upravována. Cesty k dosažení tohoto cíle jsou různé. Někdy je generovaným číslem vždy násobeno číslo předchozí, dalším generovaným číslem tento násobek atd., stále dokola. Nebo jsou čísla násobena pečlivě vybraným faktorem, pak je k výsledku ještě přičtena konstanta a výsledek se něčím vydělí, atd. atp. Alternativy nepřebírně. Ne všechny však přinášejí dobré výsledky. Své slovo tu má matematika. Ani my se jí nevyhneme, ale až o něco níže.

1/ Generování náhodných znaků pomocí reg.R

Náhodný výběr znaku z tabulky s maximálně 128 znakovými kódy. Použita je nenáročná finta, vycházející z proměnlivosti registru R. Ten mění (zvysuje) svou hodnotu před každým oběristvovacím cyklem, během nějž jsou "oživovány" log. stavy buněk jednotlivých bloků dynamické paměti RAM. Nevýhodou uvedené rutiny je, že když se sejdou časové průběhy naší programové smyčky, z níž rutinu voláme, a vlastní rutiny v nevídaném poměru, budou vybírány jen některé znaky, v nehorším případě jen jeden. Proto zde hraje svou roli časování v závislosti na strojovém cyklu a počtu bajtů ve znakové tabulce.

Algoritmus:

Přečtením obsahu reg.R dostaneme pseudonáhodnou hodnotu
 Ofset znak.tabulky = zůstatek = hodnota/počet bajtů v tab.
 Najdi znak v tabulce podle ofsetu

Délka: 15 bajtů

Vstup: v HL 1.adr.znakové tabulky - na ní je info-bajt, obsahující počet znaků v tabulce (ale nezahrnuje do ní sebe)

Výstup: v A je vybraný znak

Použité registry: AF

```

RNDCH  PUSH HL           ;Uchova 1.adr.tabulky
        LD  A,R           ;Momentální číslo z reg.R do reg.A
RCLP   SUB  (HL)          ;Odečet počtu bajtů v tabulce od čísla
        JR  NC,RCLP       ;Při kladném výsledku skok na RCLP
        ADC A,(HL)         ;Jinak přičti zpět s CY=1
        ADD A,L           ;Přičti ofset k HL
        LD  L,A
        JR  NC,RCGET
        INC H
RCGET  LD  A,(HL)          ;Do reg.A přenes znak z tabulky
        POP HL            ;Obnov 1.adr.tabulky v HL
        RET               ;Návrat

```

2/ Generování pseudonáhodných 32-bitových binárních/BCD čísel

Rutina používá algoritmus pro výpočet dalšího nového čísla Cnov z hodnoty minulého Cmin:

$$Cnov = (Cmin * a + c) \bmod m$$

kde pro čísla binární platí: a=257, c=41, m=2**32
 a " BCD " : a=101, c=29, m=10**8

Algorithm:

Nové pseudonáh.číslo = nejmenší 4 bajty starého + staré posunutě doleva o 1 bajt + konstanta

Délka: 28 bajta

Počet cyklů: pro bin.č.457, pro BCD 501

Vstup: v DEHL Cmin: pro bin CY=0, pro BCD CY=1

Výstup: v DEHL je Čnov; F beze změny

Použité registry: HL, DE, F

RANDI	PUSH BC	;Uložení registrů do zásobníku,
	PUSH AF	;CY na spodek pro jeho přístup
	LD BC,#0429	;Do B čítač, do C konstantu
	OR A	;CY=0, žádný přenos do nejniž.bajtu
RILF	LD A,C	;Posun Cmin vpravo přes CY, poprvé kon-
	LD C,L	;stanta, po2.,po3.a po4.další bajt Cmin
	LD L,H	;do A,
	LD H,E	
	LD E,D	;ponechání D pro novou číslici
	EX (SP),HL	;Zjištění stavu CY v zásobníku
	BIT 0,L	
	EX (SP),HL	;Vrácení CY do zásobníku
	JR Z,BINADD	;Pracujeme s bin? Pak skok na BINADD
	ADC A,C	;Přičtení vyššího bajtu a CY
	DAA	;Dekadická adjustace akumulátoru
	JR INRI	;Skok na INRI
BINADD	ADC A,C	;Binární součet
INRI	LD D,A	;Do D nový "náhodný bajt"
	DJNZ RILF	;Opakuji pro všechny 4 bajty
	POP AF	;Obnova registrů a CY
	POP BC	
	RET	;Návrat s Cnov v DEHL

Tato i dvě další rutiny vycházejí z obecné formule:

$$C_{nov} = A * C_{min} + K \bmod M$$

kde $M=2^{**16}$, tj. 65536; K je jakékoli liché číslo; A je modulus poměru např. 1 mod 4 nebo 5 mod 8 apod. Stupeň náhodnosti generovaných čísel záleží na volbě A. Tento stupeň, označovaný jako potencia generátoru, udává exponent mocniny výrazu $A-1$, když je mocnina dělitelná číslem 65536. Např. číslo 257 jako A dává potenci 2, protože $(257-1)**2=65536$. Potenci lze určit rovněž jako počet opakovaných dělení výrazu $A-1$ dvěma, dokud nebude výsledek lichý; počtem dělení pak vydělíte číslo 16 (jsme u čísel 16-bitových) a dostanete potenci generátoru. Obecně by A mělo být v intervalu 0,01M..0,99M. Tak číslo 257 je menší než $0,01*65536$, proto není vhodné. Experimentátoři nakonec došli k výsledku, že pro 16 bitů se jako nejlepší jeví A rovné jednomu z čísel: 293, 389, 1509, 249, 1785, 685. Jak vidět, názory se poněkud rozcházejí, protože 249 je ještě menší než 257, a přesto je mezi vhodnými. U 32-bitových čísel může vhodné A mít hodnotu 69049, 71365 nebo 100485.

3/ 16-bitový generátor pseudonáhodných čísel

V rutíně je $A=1509$. Cnov = $(Cmin * 1509 + 41) \bmod 2^{**}16$.

Násobek $1509 \cdot C_{min}$ je složen z tohoto algoritmu:

$$((32 * Cmin + (16 * Cmin - Cmin)) * 8 + Cmin) * 4 + Cmin$$

aby se předešlo zdlouhavému 1509-násobnému sečítání. Díky tomu je rutina krátká a rychlá.

Délka: 32 baita

Počet cyklů: 304

Vstup: v HL Cmin

Výstup: v HL Čnov

Použité registry: HL

```

RNDM2  PUSH AF          ;Uschova registrů používaných rutinou
        PUSH BC
        PUSH DE
        LD D,H          ;Cmin do DE
        LD E,L
        ADD HL,HL       ;Násobení součty
        ADD HL,HL
        ADD HL,HL
        ADD HL,HL       ;HL=16*Cmin
        PUSH HL         ;Uložení násobku do zásobníku
        ADD HL,HL       ;HL=32*Cmin
        EX (SP),HL      ;HL do zás., do HL 16*Cmin
        OR A            ;Vynulování CY
        SBC HL,DE       ;HL=16*Cmin-Cmin
        POP BC          ;BC=32*Cmin

```

```

ADD HL,BC      ;HL=32*Cmin+(16*Cmin-Cmin)=47*Cmin
ADD HL,HL
ADD HL,HL
ADD HL,HL      ;HL=8*47*Cmin
ADD HL,DE      ;HL=HL+Cmin=377*Cmin
ADD HL,HL
ADD HL,HL      ;HL=4*HL
ADD HL,DE      ;HL=HL+Cmin=1509*Cmin
LD DE,41       ;Do DE konstanta
ADD HL,DE      ;HL=(1509*Cmin+41) mod 2**16 = Cnov
POP DE         ;Obnovení obsahu registrů
POP BC
POP AF
RET            ;Návrat s Cnov v HL
    
```

4/ Generátor 32-bitových pseudonáhodných čísel

Zde je A=69069 (10DCDH). V rutině je vtipně využito rozdělení tohoto čísla na dvě: 10101H a CCCH. První je jako sestrojené pro operace s posuny, ale i druhé má svůj výhodný algoritmus, který celkově vypadá takto:

$$Cnov = (Cmin * 69069 + 41) \bmod 2^{**}32$$

$$\text{přičemž číslo } 69069 = 3 * (2^{**}10 + 2^{**}6 + 2^{**}2) + 2^{**}16 + 2^{**}8 + 2^{**}0$$

Délka: 43 bajtů
Počet cyklů: 1049
Vstup: v HLIX Cmin
Výstup: v HLIX Cnov
Použité registry: IX,HL

```

RDM32  PUSH AF      ;Uložení registrů používaných rutinou
        PUSH BC
        PUSH DE
        LD B,H       ;Vyšší 2 bajty Cmin do BC
        LD C,L       ;a nižší 2 bajty do DE, takže Cmin=BCDE
        PUSH IX
        POP DE
        LD A,-11     ;Čítač násobící smyčky
    
```

;Postupná tvorba Cnov. Nyní bude Cnov=3*Cmin*(2**10+2**6+2**2)

```

SHIFT  ADD IX,IX     ;Cnov=Cnov*2
        ADC HL,HL     ;Když BIT 1,A=0, pak Cnov=Cnov+Cmin (to
        BIT 1,A       ;se stane v průbězích 1,4,5 a 9). Kdyby
        JR NZ,NEXTA   ;se čítalo od +11 dolů, stalo by se to v
        ADD IX,DE     ;průbězích 3,4,7,8,11 a výsledných 2866
        ADC HL,BC     ;místo 3276 by bylo špatně.
    
```

```

NEXTA  INC A
        JR NZ,SHIFT   ;Nakonec HLIX=3276*Cmin
    
```

5/ Generátor 31-bitových pseudonáhodných čísel

Rutina rotuje vyšších 9 s nižšími 22 bity, a tak Cnov=Cmin*(2**9+1) mod (2**31-1), což vypadá hrůzostrašně, ale algoritmus rutiny je velmi rychlý. Sekvence generovaných čísel se opakuje po každých (2**31-2) číslech a kromě 0 a 7FFFFFFH obsahuje všechna čísla. Primitivní kořen (2**9+1) je poměrně malý. Lépe by bylo nahradit jej (2**17+1). Tak by místo 1 bajtu+1 bitu byly rotovány 2 bajty+1 bit. Uprava rutiny v tom smyslu není obtížná.

Základní algoritmus: Cnov=(Cmin*a) mod m; m=2**31-1; a=2**9+1
RN je proměnný obsah 4 bajtů čísla v datové oblasti a RNTMP je proměnný obsah registrů BCDE, spolupracujících při vytváření Cnov v RN.

Rotace bitů 30-22, RNTMP do bitů 8-0, RNTMP,
bity 21-0, RNTMP do bitů 30-9, RNTMP
Použito 32-bitové sečítání RN=RN+RNTMP
Když bit 31,RN=1, pak bit 31,RN=0 a RN=RN+1

Délka: 53 bajtů a 4 pro číslo
Počet cyklů: 242-336
Vstup: Cmin ve 4 bajtech paměti (nejnižší bajt na nejniž.adr.)
Výstup: Cnov
Použité registry: HL,DE,BC,AF

```

RN      DEFS 4          ;dd,cc,bb,aa, tj.4 bajty uložení čísla
;Bity 30-22 do CY a reg.C

RD31    LD    HL,(RN+2)  ;Do HL nejvyšší 2 bajty,tj.bity 31-16
        RL    L          ;Posun bitů 23 a 22 do reg.H
        RL    H
        RL    L
        RL    H
        LD    C,H        ;Bit 30 do CY
                        ;Bity 29-22 do reg.C
;Bity 21-0 a 30 do reg.DEBC s kompletní rotací reg.DEBC
        LD    A,(RN)     ;Do A bity 7-0
        RLA           ;Nasunutí bitu 30 z CY, do CY bit 7
        LD    B,A        ;Přenos obsahu A do B
        LD    DE,(RN+1)  ;Prostřední 2 bajty,tj.bity 23-8 do DE
        RL    E          ;Nasunutí bitu 7 z CY
        RL    D          ;Dokončení posunu DE; bit 7,D=bitu 22,
        RES   7,D        ;proto jeho vynulování
;Přičti rotovanou hodn.k původní; když menší než (2**31-1), pryč
        LD    HL,(RN)    ;Do HL 2 nižší bajty z RN
        ADD   HL,BC       ;Přičtení nižších 2 bajtů RNTMP
        LD    (RN),HL    ;Výsledek zpět do nož.2 bajtů RN
        LD    HL,(RN+2)  ;Do HL oba vyšší bajty RN
        ADC   HL,DE       ;Přičtení vyš.bajtů RNTMP s ev.CY=1
        RES   7,H        ;Vynulování bitu 31 beze změny indik.S
        LD    (RN+2),HL  ;Uložení výsledku do vyšší části RN
        RET    P          ;a návrat, když O.K.
;Uprava pro RN větší než (2**31-1). RN=(2**31-1) nelze, proto
;při úpravě nebude nastaven bit 31 na log.1.
RNINC    LD    HL,RN      ;Do HL adr.nejniž.bajtu čísla
        INC   (HL)        ;Zvětšení jeho obsahu o 1
        RET   NZ          ;Návrat, když bez přenosu
        INC   HL          ;Jinak na další bajt, dokud nebude
        JR    RNINC       ;úprava O.K.

```

6/ Generování 16-bitových pseudonáh.čísel bitovou metodou

Rutina se skládá ze dvou částí. SEED využívá proměnlivosti reg.R. RANDOM vypočítává 2-bajtové Cnov z obsahu 3 bajtů v DEFS RND. Použijeme-li SEED, bude obsah RND závislý na momentálním obsahu reg.R. Poté musíme vždy zvlášť volat rutinu RANDOM. Vynecháním rutiny SEED bude v RND Cmin rovno poslednímu Cnov. Výhoda této sestavy tkví v tom, že její části můžeme volat střídavě, a tak dosahovat rozličné plejády čísel.

Algoritmus:

```

SEED: 3 přenosy momentálních obsahů reg.R do 3 bajtů RND
RANDOM: Opakuj 16-krát:
      RND=RND*2+(bit 18,RND) XOR (bit 8,RND)

```

Délka: 52 bajtů (18 pro SEED, 34 pro RANDOM) a 3 pro RND

Počet cyklů: SEED 190, RANDOM 2133

Vstup: Pro SEED nic.

U RANDOM RND obsahuje předešlou 3-bajtovou proměnnou hodnotu buď z operace rutiny RANDOM nebo SEED.

Výstup: v (RND+1) je vyšší, v (RND+2) nižší bajt Cnov

Použité registry: ---

```

RND      DEFS 3          ;3-bajtová proměnná

SEED     PUSH AF          ;Uložení obsahu registrů,užitých rutinou
        PUSH BC
        PUSH HL
        LD    HL,RND     ;Do HL adresa RND
        LD    B,3        ;Do B čítač bajtů v dat.obl.RND

SEEDA    LD    A,R        ;Obsah reg.R do reg.A
        LD    (HL),A      ;a na adresu v oblasti RND
        INC   HL          ;Na další adresu v RND
        DJNZ SEEDA       ;Opakuj pro 3 bajty RND; průběh smyčkou
        POP   HL          ;trvá 8 stroj.cyklů; obnovení obsahu
        POP   BC          ;registrů
        POP   AF
        RET              ;Návrat

RANDOM    PUSH AF          ;Uložení registrů
        PUSH BC
        PUSH HL
        LD    HL,RND     ;Do HL adresa RND
        LD    B,16       ;V B čítač 16ti bitů Cnov

```

```

RANDO LD C, (HL)      ;Nejnižší 2 bajty RND do CA
      INC HL
      LD A, (HL)
      INC HL           ;V HL adr.nejvyš.bajtu RND
      SRL C           ;Posun bitu 3 na pozici bitu 0
      SRL C
      XOR C           ;Bit 3,RND XOR bit 0,RND+1
      RRA             ;Výsledek XORu bitů do CY
      RL (HL)         ;Rotace nového bitu na pozice
      DEC HL          ;bitu 3,RND a bitu 0,RND+1
      RL (HL)
      DEC HL
      RL (HL)
      DJNZ RANDO      ;Opakuj pro všech 16 bitů
      POP HL          ;Obnov obsah registrů
      POP BC
      POP AF
      RET             ;Návrat

```

Poznámka k části SEED. Každý průběh smyčkou SEEDA zabere 8 strojových cyklů, což znamená, že:

$$\begin{aligned}
 (RND+0) &= n * (n \text{ menší než } 128) \\
 (RND+1) &= n + 8 \pmod{128} \\
 (RND+2) &= n + 16 \pmod{128}
 \end{aligned}$$

a to zase znamená nevíтанou pravidelnost v případě, že n bude násobkem osmi.

Rutina RANDOM je orientována paměťově. Můžeme ji zrychlit, když ji budeme orientovat registrově. Jinými slovy - proměnná RND nebude ve 3 bajtech paměti, ale ve 3 registrech, zde CDE, do nichž se bude ukládat číslo podobně jako do RND. Uprava se týká části RANDO:

```

RANDO LD A,C
      RRA
      RRA
      RRA
      XOR D
      RRA
      RL E
      RL D
      RL C
      DJNZ RANDOV

```

ja ještě následuje obnova registrů a návrat.

KAPITOLA 5

Konverze

Konverze, neboli převody z jedné číselné soustavy do jiné, či jedné znakových kódů na jiné apod. jsou nedílnou součástí každého programu. Pokud jde o převody číselných soustav, lze říci, že přímo navazují na použití rutin uvedených v kapitole 3. Kromě zde uvedených soustav a kódů jich existuje ještě celá řada - např. Gray code, kód EBCDIC a další. Jejich využití je však dost specifické, proto tu rutiny na jejich převody nenajdete. V případě potřeby takového převodu budete ale moci postupovat analogicky k převodům zde uvedeným. Nejklasikářším případem aplikace konverzních rutin je obousměrný převod mezi čísly binárními a BCD pro interakci programu s uživatelem, který zadává a čte data v dekadickém vyjádření, zatímco program "pracuje binárně". Jde tedy o jakýsi číselný interfacing mezi programem a uživatelem. Ale jak už asi sami víte nejlépe, jedním z nejčastějších případů konverze "uvnitř" programu je převod mezi binární a HD soustavou.

1/ Jednobajtová konverze bin./BCD

Převod jednoho bajtu binárních dat na 2 bajty BCD dat. Jde o jeden z nejfrekventovanější používaných převodů pro projekci číselné informace na monitoru. Rutina opakovaně odečítá napřed 100 a pak 10. Počet korigovaných odečtů se zaznamenává, v zůstatku jsou jednotky. Výsledkem jsou 3 BCD čísla ve 12ti bitech dvou bajtů (stovky, desítky a jednotky).

Délka: 27 bajtů
Počet cyklů: 497
Vstup: v A bin.data
Výstup: v HL BCD data
Použité registry: HL,C,AF

```

BN2BCD LD    H,#FF          ;V H -1 pro vyrovnání na 0 na zač.D100LP
D100LP INC    H              ;Zvýš čítač stovek
      SUB     100            ;Odečti 100 od obsahu reg.A
      JR      NC,D100LP      ;Když CY=0, opakuj odečet
      ADD     A,100          ;Vyrovnání obsahu reg.A; v H stovky

D10LP  LD      L,#FF         ;Do L opět -1
      INC     L              ;Zvýš čítač desítek
      SUB     10             ;Odečti 10 od obsahu reg.A
      JR      NC,D10LP      ;Dokud CY=0, opakuj
      ADD     A,10           ;Vyrovnaj obsah reg.A; v L desítky
      LD      C,A            ;Jednotky do C
      LD      A,L            ;Desítky do A
      RLCA                  ;Posun počtu desítek do vyšších 4 bitů A
      RLCA
      RLCA
      RLCA
      OR      C              ;Do nižších 4 bitů jednotky (maskování)
      LD      L,A            ;Desítky a jednotky do L
      RET                    ;Návrat

```

2/ Multibajtová konverze bin./BCD

Binární číslo může obsahovat maximálně 32 bajtů. Rutina nebere v úvahu znaménko. POZOR - nejvyšší bajt bin.číslo je na nejnižší adrese. Pokud by byly bajty vašeho čísla v opačném pořadí, můžete pro jejich prohození použít např. tuto rutinu:

```

START  LD DE,nejnižší adr.prac.oblasti
      LD BC,pocet bajtů čísla
      LD HL,nejvyš. adr.převáděného čísla+1
PRENOS DEC HL
      LD A,(HL)
      LD (DE),A
      INC DE
      DEC C
      JR NZ,PRENOS

```

Nakonec můžete číslo z pracovní oblasti přenést na místo původního instrukcí LDIR některou z rutin blokového přenosu z předposlední kapitoly.

Algoritmus:

Vynuluj pracovní oblast pro BCD číslo
 Do prac.obl. pro bin.data přenes bin.číslo
 Vypočti počet bitů bin.čísla pro bit.čítač
 Dokud bit.čítač není nulový, nebo nenastane BCD přetečení,
 opakuj:
 bin.č.=bin.č.*2+bit s použitím dekadického sečítání
 zmenši bit.čítač o 1

Délka: 43 bajtů

Vstup: v HL adr.nejvyš.bajtu bin.č. (na nejniž.adrese)
 v DE adr.1.bajtu prac.oblasti/výsledku
 v B čítač počtu bajtů BCD čísla
 v C " " " " bin.

Výstup: v C délka bin., v E délka BCD čísla, B=0, obsah A a prac.
 oblasti bin.dat je nejistý;
 při CY=1 došlo k přetečení, převod není kompletní, D=?,
 v HL adr.posl.bajtu prac.obl.
 při CY=0 konverze D.K., D=0, v HL 1.adr.prac.oblasti, tj.
 nejvyššího BCD bajtu.

Použité registry: HL,DE,BC,AF

```

CVD    XOR    A           ;Vynulování obsahu reg.A
        PUSH   BC         ;Uchova čítačů délek

CLRW    LD     (DE),A      ;Vynulování prac.obl.pro BCD
        INC    DE
        DJNZ   CLRW       ;Nakonec DE=1.adr.prac.obl.pro bin.data

        LDIR          ;Přenos bin.č.do jeho prac.obl.bin.dat
        EX     DE,HL      ;Do HL adr.nejniž.bajtu bin.č.
        DEC    HL
        POP    BC         ;Obnova obsahu čítačů
        LD     E,B        ;Délka BCD do E
        LD     A,C        ;Počet bajtů bin.č*B=počtu bitů v bin.č.
        ADD    A,A
        ADD    A,A
        ADD    A,A
        LD     D,A        ;Bitový čítač v D

BITLP   PUSH   HL         ;Uchova nižší adr.prac.obl.bin.dat
        LD     B,C        ;V B čítač počtu bajtů bin.č.
        BINLP  RL      (HL) ;Posun všech bitů bin.č. o 1 bit vzhůru,
        DEC    HL         ;tj.bin.č.*2
        DJNZ   BINLP

BCDLP   LD     B,E        ;V B čítač počtu bajtů BCD
        LD     A,(HL)     ;Proved dekad.BCD*2
        ADC    A,A        ;s připočtením bitu vysunutého z bin.č.
        DAA          ;Dekad.adjustace
        LD     (HL),A     ;Uložení výsledku do prac.obl.
        DEC    HL        ;Jdi na adr.dalšího bajtu
        DJNZ   BCDLP     ;Opakuj pro všechny BCD bajty

        POP    HL        ;Obnov nejniž.adr.bin.č.v HL
        RET    C         ;Návrat při přetečení

        DEC    D         ;Zmenši bitový čítač o 1
        JR     NZ,BITLP  ;a opakuj pro všechny bity bin.čísla
        SBC    HL,BC     ;Do HL adr.nejvyš.bin.bajtu-1
        SBC    HL,DE     ;V HL " " " " BCD " " -1
        INC    HL        ;V HL " " " " BCD bajtu
        RET          ;Návrat s CY=0,B=D=0,C=bin.,E=BCD délka
    
```

3/ Jednobajtová konverze BCD/bin.

Převod 1 bajtu s BCD daty do 1 bajtu s bin.daty.
 Algoritmus:

Vynásob vyšší 4 bity 10ti; pozn.: $*10 = (*8) + (*2)$
 Přičti nižší 4 bity k bin.ekvivalentu vyšších 4 bitů

Délka: 14 bajtů

Počet cyklů: 60

Vstup: v A BCD data

Výstup: v A bin data

Použité registry: BC,AF

```
BCD2BN LD B,A      ;Uchova p v.BCD hodnoty do B
        AND #F0     ;Vynulov n  n z   ch 4 bit 
        RRCA        ;Posun o 1 bit doprava, tedy stav, v
        LD C,A      ;n m z vy .4 bity maj  hodn.*8; A do C
        RRCA        ;Bity na pozici, v n   maj  hodn.*2
        RRCA
        ADD A,C      ;Sou et, v sl.=bity*10
        LD C,A      ;V sledek z A do C
        LD A,B      ;Do A p v.hodnota BCD
        AND #0F      ;Vynulov n  vy   ch 4 bit 
        ADD A,C      ;Sou et n z  ch s minul.v sledkem
        RET          ;N vrat s bin.  slem v A
```

4/ Konverze 1 bin.bajtu/2 HD-ASCII znaky

P evod 1 bajtu bin.dat na dv  HD   slice vyj d en  v ASCII. Ka d  polovina bajtu je p evedena na ASCII k d p   ten m 30H, kdy  jde o k d   sla, nebo v p  pad  jin ho znaku dal  m p   ten m   sla 7 pro p  ekon n  mezery mezi ASCII k dem   sla 9 (39H) a p  smenem A (41H) v ASCII tabulce k d .

D lka: 28 bajt 
Po et cykl : p  m rn  162
Vstup: v A bin.data
V stup: v HL dva ASCII k dy
Pou it  registry: HL,B,AF

```
BN2HEX LD B,A      ;Uchova p v.bin.  sla
        AND #F0     ;Vynulov n  n z    p lky bajtu
        RRCA        ;P esun vy    p lky do n z   
        RRCA
        RRCA
        CALL NASCII ;P evod t to p lky na ASCII
        LD H,A      ;K d p vodn  vy    p lky do H
        LD A,B      ;P v.bin.  slo do A
        AND #0F      ;Vynulov n  vy    p lky bajtu
        CALL NASCII ;a je  p evod na ASCII
        LD L,A      ;K d n z    p lky do L
        RET          ;N vrat

NASCII CP 10        ;Bude znak   slem?
        JR C,NAS1   ;Kdy  AND, skok na NAS1
        ADD A,7      ;Kdy  bude jin m znakem, p   ti 7
        NAS1 ADD A,"0" ;P   ti 30H pro v sledn  k d ASCII
        RET          ;N vrat
```

5/ Konverze 2 HD-ASCII znak /1 bin.bajt

Konverze opa n  k p edchoz , postup analogick . M sto p   t n  30H a ev.n davkem 7, se ode  t  od ka d ho z obou k d  a v sledn  dv    sla se maskov n m spo   v jednom bin.bajtu.

D lka: 24 bajt 
Po et cykl : p  m rn  148
Vstup: v HL k dy dvou ASCII znak 
V stup: v A bin.data

```
HEX2BN LD A,L      ;N z   bajt HD-ASCII do A
        CALL A2HEX  ;a jeho p evod na "  st " HD   slo
        LD B,A      ;HD tvar n z   ho bajtu do B
        LD A,H      ;Tot   pro vy    bajt HD-ASCII
        CALL A2HEX
        RRCA        ;Jeho HD tvar do vy   ch 4 bit  reg.A
        RRCA
        RRCA
        RRCA
        OR B        ;"Vmaskov n " n z    p lky z B do A
        RET          ;N vrat

A2HEX SUB "0"      ;Ode  et 30H od ACII k du
        CP 10       ;Je to   slice?
        JR C,A2HEX1 ;Kdy  AND, sm r n vrat
        SUB 7        ;U p v.jin ch znak  je   ode  st 7
        A2HEX1 RET   ;N vrat
```

6/ DK konverze 16-bitov ho bin./dek-ASCII

Rutina bere v  vahu znam nko p ev d n ho   sla; de facto je tedy p ev d no 15-bitov  bin.  slo, v jeho  nejvy   m bitu je znam nko. Kdy  je   slo z porn , je p ev d no na svou absolutn  hodnotu, tedy dvojkov  komplement. Pak je hodnota bin. . st le d lena 10ti, dokud celo  seln  v sledek nen  nulov . Kompletn  dek-ASCII je v bufferu. Kdy  bylo p vodn  bin.  slo z porn , je mu je   p ed azen ASCII znak "-".

Délka: 107 bajtů a 4 pro data

Počet cyklů: průměrně 7200

Vstup: v HL adresa výstupního bufferu

v DE 16-bitové bin.číslo

Výstup: v 1.bajtu bufferu je délka, následovaná znaky dek-ASCII

Použité registry: HL,DE,BC,AF

```
BNZDEC LD (BUFPTR),HL ;Ukazatel adresy bufferu do obl.dat
        EX DE,HL ;a do DE, bin.č.do HL
        LD A,0
        LD (CURLN),A ;Vynulování 1.bajtu bufferu
        LD A,H ;Vyšší bajt bin.č.do A
        LD (NGFLAG),A ;Uchova znaménka do obl.dat
        OR A ;Je bit 7, A nulový, číslo je kladné?
        JP F,CNVERT ;Když AND, skok na CNVERT
        EX DE,HL ;Když je záporné, bin do DE
        LD HL,0 ;Vynulování HL
        OR A ;a CY před odečtem
        SBC HL,DE ;Dvojk.kompl., tj.převod na abs.hodnotu
```

```
CNVERT LD E,0 ;Vynulování zůstatku
        LD B,16 ;V B čítač 16ti bitů
        OR A ;Vynulování CY
```

;V HL bude HL div 10 (celoč.výsl.), v DE pak HL mod 10 (zůstatek)
;Posun dalšího bitu výsl.do bitu 0 dělence (obligátní šetření
;paměti) a nejvyššího bitu dělence do bitu 0 zůstatku. Jde tedy
;o posouvání v poli reg.LHE vlevo.

```
DVLOOP RL L ;CY do bitu 0,L
        RL H ;Posun LH
        RL E ;Dokončení posunu LHE
        LD A,E ;Mezizůstatek do A pro
        SUB 10 ;odečet 10; když je obsah E větší než
        CCF ;10, další bit výsledku bude log.1
        JR NC,DECCNT ;Když je menší než 10, skok na DECCNT
        LD E,A ;Zbytek odečtu do E
        DECCNT DJNZ DVLOOP ;Opakuj pro všech 16 bitů bin.čísla
```

```
        RL L ;Poslední CY do výsledku
        RL H
```

```
CHINS LD A,E ;Převod hodn.zůstatku na kód ASCII
        ADD A,"0" ;Přičtení 30H
        CALL INSERT ;Uložení kódu do bufferu
        LD A,H ;Když je výsledek ještě nenulový,
        OR L
        JR NZ,CNVERT ;pokračuj v konverzi
```

```
EXIT LD A,(NGFLAG) ;Jinak zjistí znaménko
        OR A
        JP F,POS ;Když je číslo kladné, směr návrat
        LD A,"-" ;Při záporném předsuň znak minus
        CALL INSERT
        POS RET ;Konečný návrat
```

```
INSERT PUSH HL ;Uložení HL
        PUSH AF ;a znaku v A do zásobníku
        LD HL,(BUFPTR) ;Ukazatel adr.bufferu do HL
        LD D,H ;a do DE
        LD E,L
        INC DE ;V DE moment.konec bufferu+1
        LD (BUFPTR),DE ;Uložení nové adr.buf.do obl.dat
        LD A,(CURLN) ;Momentální počet znaků do A
        OR A ;Když je nulový,
        JR Z,EXITMR ;skok na EXITMR
        LD C,A ;Počet znaků do BC jako čítače
        LD B,0
        LDDR ;pro jejich přesun o 1 znak nahoru
```

```
EXITMR LD A,(CURLN) ;Zvětšení čítače počtu znaků
        INC A ;o 1
        LD (CURLN),A ;a jeho uložení do obl.dat
        LD (HL),A ;i na začátek bufferu
        EX DE,HL ;V HL adr.1.znaku
        POP AF ;Do A ASCII kód znaku
        LD (HL),A ;Jeho uložení na volnou adresu bufferu
        POP HL ;Obnovení obsahu reg.HL
        RET ;Návrat ze subrutiny
```

```
BUFPTR DEFS 2 ;Proměnný ukazatel adr.umís.posl.znaku
CURLN DEFS 1 ;Proměnná délky bufferu
NGFLAG DEFS 1 ;Bit 7,(NGFLAG) je znaménkem čísla
```

7/ DK konverze 32-bitového bin./dek-ASCII

Rutina bere v úvahu znaménko, převádí tedy 31-bitové bin.číslo na řetězec dek-ASCII číselných znaků. Rutina je velmi podobná předcházející, ale leccos řeší zcela jinak. Zajímavé je ukládání BCD čísl do zásobníku instrukcí PUSH AF, kde hraje svou roli stav indikátoru Z. Převáděné DK číslo může být v rozsahu hodnot od -2^{31} , což je 80000000H do $2^{31}-1$, tedy 7FFFFFFFH. První z těchto čísel ale nemá v rozsahu 32 bitů svůj DK, proto si na něj rutina musí dát pozor.

Algoritmus:

Když je bin.č.= 80000000H, pak
signalizuj chybu
Jinak, když je bin.č. záporné, pak
urči jeho abs.hodnotu
zapiš znak "-" na 1.místo BCD řetězce
adresuj další volný bajt řetězce
Dokud je bin.č. větší než 0
čísllice=(čísl/10)*zůstatek
čísl=(čísl/10)*celočís.výsledek
čísllice=čísllice+ASCII vyšší 4 bity
uschovej čísllici do zásobníku
Pro uschované čísllice
Vyjmi čísllici ze zásobníku a zapiš ji do řetězce
Nakonec do něj zapiš mezeru jako znak konce řetězce

Délka: 68 bajtů

Počet cyklů: max.23332

Vstup: v BCDE 32-bitové číslo

v HL 1.adresa řetězce

Výstup: při Z=0 převod D.K., v CY je znaménko

v DE 1.bajt řetězce BCD čísla

při Z=1 je i CY=1, na vstupu bylo 80000000H, všechny

registry beze změny

Použité registry: RL, DE, BC, F

Pozn.: Rutina používá subrutinu SNEG4 (viz Kapitulu 3, bod 13/)

SBAD4	PUSH BC	: Ulož bin.č.do zásobníku pro ev.potřebu
	PUSH DE	: jeho negace (dvojk.komplementace)
	LD C,A	: Uchova A do C
	RL B	: Znaménko do CY
	SBC A,A	: Při CY=1 bude A=FFH, při CY=0 A=0
	CALL C,SNEG4	: Je-li číslo záporné, proved negaci
	CCF	: Komplementace CY
	ADC A,1	: A+1+CY
	LD A,C	: Obnova pův.obsahu reg.A
	POP DE	: Abs.hodn.bin.č., nebo 80000000H do BCDE
	POP BC	: Při 80000000H je
	RET Z	: výsledek ADC A,1 nulový, návrat
	PUSH HL	: Adresa řetězce do zásobníku
	JR NC,SBZS	: Když ADC A,1 bez přenosu, číslo kladné
	LD (HL), "-"	: Jinak zapiš do řetězce znak minus
	INC HL	: Na další adr.řetězce
SBZS	JR SBPSH	: Skok na SBPSH se Z=0 (důvod viz SBPOP)
SBALP	BIT 7,A	: Nastavení Z na log.1
SBPSH	PUSH AF	: Stavba čísllicového zásobníku
	PUSH HL	: s adresou řetězce vespod
	EX DE,HL	: Do BCHL číslo pro dělení
	LD D,32	: Čítač 32 bitů
	XOR A	: Vynulování pole dělení
SBDIV	ADD HL,HL	: Posun dělence do výsledku o 1 bit na-
	RL C	: horu
	RL B	
	ADC A,A	: Nejvyšší vysunutý bit do z CY do reg.A
	CP #0A	: Když je výsledek menší než 10,
	JR C,SBDIVT	: žádné další odečty, skok na SBDIVT
	SUB #0A	: Jinak odečíst 10 a umístit výsledkový
	INC HL	: bit na správné místo
SBDIVT	DEC D	: Zmenšení čítače bitů o 1 a opakovat pro
	JR NZ,SBDIV	: všechny bity; BCHL=výsl.,A=zůstatek
	ADD A,#30	: Převod výsledku na ASCII kód
	LD D,H	: "Mezičíslo" zpět do BCDE
	LD E,L	
	ADC HL,BC	: Součet nižš.a vyšš.bajtu pro indik.0
	POP HL	: Obnova adresy řetězce v HL
	JR NZ,SBALP	: Jen při nulovém výsledku součtu končit
	JR C,SBALP	
SBPOP	LD (HL),A	: Zapis čísllice do řetězce
	INC HL	: Na další adr.řetězce
	POP AF	: Vytažení další čísllice ze zásobníku,
	JR Z,SBPOP	: Opakuj, dokud Z=0
	POP DE	: Do DE 1.adr.řetězce
	LD (HL),#20	: Uložení znaku mezery pro indik.konce
	RET	: Návrat

B/ DK konverze dek-ASCII/2-bajtové bin.

Převod řetězce dek-ASCII znaků na 16-bitovou bin. hodnotu. Řetězec musí na začátku mít informační bajt o jeho délce v bajtech (včetně ev. znaménka). Rutina zaznamená znaménko mínus, plus ignoruje. Následuje převod na BCD odečtem 30H od každé číslice, přičemž na BCD již převedené jsou násobeny 10ti a přičítány k meziproductu. Nakonec je v případě záporného znaménka provedena dvojková komplementace.

Délka: 79 bajtů a 1 pro data

Počet cyklů: průměrně 152 na bajt + max. 186

Vstup: v HL 1. adr. vstupního bufferu

Výstup: v HL bin.č., při CY=1 chyba

```

DEC2BN LD A, (HL)      ; Délka v bajtech přenesena z informač-
LD B, A                ; ního bajtu do B
INC HL                 ; Na další adr. řetězce
SUB A                  ; Vynulování obsahu reg. A
LD (NGFLAG), A        ; Předpoklad kladného čísla
LD DE, 0               ; Začneme s nulovým meziproductem v DE
OR B                   ; Je délka čísla nulová?
JR Z, EREXIT           ; Když AND, návrat

INITI LD A, (HL)       ; První znak do A
CP "-"                 ; Je to znak mínus?
JR NZ, PLUS            ; Když NE, skok na PLUS
LD A, #FF              ; Do A indikace záporné hodnoty
LD (NGFLAG), A        ; a její uložení do obl. dat
JR SKIP                ; Skok na SKIP

PLUS CP "+"            ; Je to znak plus?
JR NZ, CHKDIG          ; Když NE, začni konverzi na CHKDIG
SKIP INC HL            ; Jinak na další znak
DEC B                  ; Zmenšení čítače bajtů
JR Z, EREXIT           ; Když bylo jen znaménko, návrat

CNVERT LD A, (HL)      ; Další znak do A

CHKDIG SUB "0"         ; Odečet 30H
JR C, EREXIT           ; Když znak menší než "0", chyba
CP "9"                 ; 9+1
JR NC, EREXIT          ; Když znak větší než "9", chyba
LD C, A                ; Znak je číslicí, uschovej jej do C

PUSH HL                ; Uschova ukazatele buffreru
EX DE, HL              ; Meziproduct do HL
ADD HL, HL              ; HL*2
LD E, L                ; HL do DE
LD D, H                ; HL*4
ADD HL, HL              ; HL*8
ADD HL, HL              ; HL*8+HL*2=HL*10
ADD HL, DE              ; Číslice z C do DE
LD E, C                ; Přičtení číslice k meziproductu
LD D, 0                ; a jeho přenos do DE
EX DE, HL              ; a jeho přenos do DE
POP HL                 ; Na další znak
INC HL                 ; Opakuj pro všechny bajty
DJNZ CNVERT

EX DE, HL              ; Výsledek do DE
LD A, (NGFLAG)         ; Indikace znaménka do A
OR A                   ; Je to plus?
JR Z, OKEXIT           ; Když AND, skok na OKEXIT
EX DE, HL              ; Výsledek do HL pro nalezení jeho
LD HL, 0               ; dvojkového komplementu
SBC HL, DE             ; dvojkového komplementu

OKEXIT OR A            ; Vynulování CY, aneb data platná
RET                    ; Návrat

EREXIT EX DE, HL       ; V HL hodnota čísla
SCF                    ; CY=1 pro signalizaci chyby
RET                    ; Návrat

NGFLAG DEFS 1          ; Indikace znaménka (=FFH pro mínus)

```

9/ Konverze ASCII čísla ze soustavy 2 až 36/2-bajtové bin.

Zajímavá rutina, která převádí ASCII čísla z jakékoli číselné soustavy se základem v intervalu 2..36 na 16-bitové bin. číslo. Za pozornost stojí především subrutina DEXC, která volí velmi rychlý způsob násobení obsahů reg.DE s reg.C pomocí posunu reg.C v přímém vztahu k číselnému řádu. Kódy ASCII musejí prezentovat velká písmena. Pracujete-li i s malými, zde jedna pidirutinka pro jejich převod na velká:

```

CP "a"
JR C, VEN
CP "z"+1
JR NC, VEN
SUB "a"-"A"
VEN RET

```

Výsledný kód je v reg.A. Protože následující rutina pracuje se základem až 36, používají se pro číslice i další písmena, než jsme zvyklí v soustavě 16kové, kde je to jen A-F. Při základu 36 to bude o 20 písmen víc. Rutina v průběhu odběru ASCII číslic zjišťuje, zda jsou menší, než základ (tak tomu musí být). Např. v soustavě dvojkové je nejvyšší číslice 1, v desítkové 9, v 16kové 15, resp. 0FH apod. Rutina je využitelná jako univerzální pro všechny neužívanější číselné převody z ASCII. Na své si konečně přijdou i ti, kdož jsou zvyklí na soustavu osmičkovou.

Algoritmus:

```

Při chybě návrat s CY=1
Vynuluj pole výsledku
Opakuj:
  Přečti znak a adresuj další
  Když znak není ASCII 0-9 nebo A-Z, pak chyba
  Převed' znak na binární číslo
  Přičti bin.č.k poli výsledku
  Když dojde k přetečení, pak chyba
  Čítač zmenši o 1
  Když je čítač větší než 0, pak
    vynásob pole základem
  Když dojde k přetečení, pak chyba
  Dokud čítač není nulový

```

Délka: 62

Vstup: v DE adresa 1.bajtu ASCII řetězce

v B počet bajtů ASCII řetězce

v C číselný základ (báze)

Výstup: při CY=1 chyba, beze změny jen C, ostatní reg.=?

při CY=0 O.K., v HL bin.výsledek

v DE adr.posl.bajtu řetězce+1

B=0, A=H, C beze změny

```

ANYNO LD HL, 0 ; Vynuluj pole výsledku
AY1 LD A, (DE) ; Do A ASCII znak
INC DE ; a adresuj další v řetězci
CALL CTON ; Převod z ASCII na číslo
RET C ; Když znak nebyl 0-9, A-Z, návrat
CP C ; Porovnej bázi s číslem
CCF ;
RET C ; Návrat, když číslo nepatří do soustavy
ADD A, L ; Přičti číslo k poli výsledku
LD L, A
LD A, H
ADC A, 0 ; a CY
LD H, A ; do nejvyš.bajtu
RET C ; Návrat při přetečení
DEC B ; Zmenši čítač bajtů o 1
RET Z ; Návrat při nulovém čítači
PUSH DE ; Ušchovej všechny vstupní registry
PUSH BC
EX DE, HL ; Pole výsledku do DE
CALL DEXC ; v HL bude mezivýsledek*báze
POP BC ; Obnova obsahů registrů
POP DE
RET C ; Návrat při přetečení
JR AY1 ; Pokračuj na AY1

CTON SUB #30 ; Převod z ASCII na číslo
RET C ; Návrat, když je číslo menší než 0
CP #0A ; Byla to ASCII číslice?
CCF ;
RET NC ; Když AND, návrat
SUB 7 ; Byl to ASCII kód menší než "A"?
RET C ; Když AND, návrat
CP #24 ; Byl znak velkým písmenem?
CCF ;
RET ; Když NE, návrat s CY=1, jinak s CY=0

```



```

DEXC LD HL,0 ; Vynuluj výsledné pole
DEX1 SRL C ; Nejnižší bit násobitele do CY
JR NC,DEX2 ; Když byl bit 0, C=0, skok na DEX2
ADD HL,DE ; Jinak přičti násobence k poli výsledku
RET C ; Návrat při přetečení
DEX2 RET Z ; Je-li násobitel C=0, O.K., návrat
EX DE,HL ; Násobenec do DE
ADD HL,HL ; a jeho vynásobení dvěma
EX DE,HL ; Výsledek do DE
RET C ; Návrat při přetečení
JR DEX1 ; Jinak opakuj pro další bit násobitele

```

10/ Konverze až 255-bajtového bin./dek-ASCII

Rutina místo běžného násobení mezivýsledku číslem 10 používá metodu, zvanou Couleurovo kritérium. Spočívá v porovnávání každé výsledné číslice (násobené bázi 2) sází samotnou. Když číslice převyšuje bázi, je od něj odečtena s následným přenosem do příští číslice (která bude opět násobena bázi, atd...). Užití kritérium umožňuje přepracovat uvedenou rutinu pro převod na jiné báze, než je desítková (viz poznámka za rutinou).

Algoritmus:

```

Adresuj nejvyšší bajt bin.čísla
Když bin.č.=0, pak
    ASCII=0 s nulovým zakončovacím znakem
Jinak opakuj:
    Adresuj nejvyšší bit v bajtu
    Opakuj:
        Vynásob meziproduct dvěma a připočti moment.bin.bit
        Při adjustaci výsl.bajtu proved ev.přenos do dalšího
        Když dojde k přenosu z mezivýsledku, pak
            zvýš celk.délku mezivýsledku o 1 a jeho nejvyšší-
            mu bajtu uděl hodnotu 1
        Adresuj další nižší bit v bajtu
        Dokud nejsou zpracovány všechny bity bajtu, zpět
    Adresuj další nižší bajt bin.čísla
    Dokud nejsou zpracovány všechny bajty bin.čísla, zpět
    Umísti nulový zakončovací znak k mezivýsledku
    Nastav ukazatele na nejvyšší a nejnižší konec výsledku
    Je-li vyšší ukazatel větší než nižší, pak
        prováděj převod bajtů na ASCII znaky výměnou indexova-
        vaných bajtů a přičtením 30H ke každému z nich
    Zvýš nižší ukazatel a sniž vyšší

```

Délka: 114 bajtů

Vstup: v HL adr.1.bajtu bin.čísla (jeho nejn.bajt na nejn.adr.)
v DE adr.1.bajtu cílového pole, které musí být dostatečně velké pro uložení dek.výsledku (2,42*bin.délka+1)
v A bin.délka (počet bajtů bin.čísla v intervalu 1..255)
Výstup: Řetězec číselných kódů ASCII znaků má nejvyšší bajt na nejnižší adrese s nulou na konci; řetězec není adresován (nutno uschovat jeho adresu předem!); bin.číslo beze změny, obsah všech registrů nepředpokládátejný

Použité registry: HL,DE,BC,AF

```

BASE EQU NN ; NN je číslo báze převodu
VPTR DEFS 2 ; Uložení adr.adresovaného bajtu bin.čís.

LNGIBD LD C,A ; Délka bin čísla do BC
LD B,0
DEC HL ; Jdi na adresu před bin.č.
LD (VPTR),HL ; a ulož ji do dat
LD A,-1 ; Na konec dek.řetězce ulož dočasně FFH
LD (DE),A
ADD HL,BC ; V HL posl.bajt bin.čísla

NXTMSB LD A,(HL) ; Do A bin.bajt
OR A ; Je nulový?
JP NZ,MSBFND ; Když NE, je to první nejvyš.bin.bajt
DEC HL ; Na další adresu bin.čísla
DEC C ; Zmenši čítač délky bin.čísla o 1
JP NZ,NXTMSB ; Když bin.bajty ještě nehotovy, skok

EX DE,HL ; Jinak adresuj ASCII řetězec, ulož do
LD (HL),"0" ; něj ASCII znak nuly
INC HL ; a zakončovací nulu
RET ; Konverze hotova, návrat

MSBFND LD B,A ; Do B nejvyšší bajt
LD A,#80 ; Maska pro bit 7

NXTMSK CP B ; Porovnej bajt s maskou
JP C,MSKFND ; Když je výsledek záporný nebo nulový,
JP Z,MSKFND ; skok na MSKFND beze změny masky
RRCA ; Bit masky o 1 bit níž
JP NXTMSK ; Zopakuj porovnání

```

```

MSKFND LD B,A      ;Ulož masku do B
        PUSH BC     ;a spolu s čítačem do zásobníku
        LD HL,(VPTR) ;Do HL adr.bin.pole-1
        LD B,0
        ADD HL,BC    ;Adresuj momentální bajt
        AND (HL)     ;"Vmaskuj" do něj moment.mask.bit
        ADD A,#FF    ;CY=1,když výsl.bit=log.1
        LD L,E       ;Do HL adresa řetězce dek.č.
        LD H,D

NXTOPV LD A,(HL)    ;Do A výsledný bajt
        INC A        ;Test na zakonč.bajt FFH
        JP Z,OPVDON ;Když je to on, skok
        DEC A        ;Jinak vrať obsah A na hodnotu bajtu
        ADC A,A       ;a vynásob ji dvěma s přičtením CY
        CP BASE      ;Je číslice mešší než báze?
        JP C,NOCOUL  ;Když AND, skok
        SUB BASE      ;Jinak od ní bázi odečti
        CCF          ;a CY na správnou hodnotu
        LD (HL),A     ;Výsledek násobení dvěma ulož
        INC HL        ;a adresuj další bajt
        JP NXTOPV    ;Opakuj

OPVDON JP NC,EXTDON ;Skok, když nedošlo k přetečení
        LD (HL),1     ;Při přeteč.mohla být nová číslice jen 1
        INC HL
        LD (HL),-1    ;Ulož nový zakonč.bajt FFH

EXTDON POP BC       ;Obnov obsah masky a čítače
        LD A,B       ;Maska do A
        RRCA         ;Posun bitu masky o 1 bit níž
        JP NC,MSKFND ;Když bit log.1 "nespadl do CY", skok
        DEC C        ;Jinak zmenši čítač délky o 1
        JP NZ,MSKFND ;A pokračuj, dokud čekají další bajty
        LD (HL),0    ;Vyměň zakonč.bajt FFH za nulu

NXTCVN DEC HL       ;Vyšší ukazatel o 1 bajt dolů
        LD A,L       ;Porovnání vyššího ukazatele s nižším
        SUB E        ;
        LD A,H       ;
        SBC A,D      ;Když se ukazatele "míjejí",
        RET C        ;převod na ASCII hotov, návrat

        LD A,(DE)    ;Jinak v něm pokračuj
        OR #30       ;K číslici maskou přičti 30H
        LD B,A       ;Výsledný ASCII kód do B
        LD A,(HL)    ;Totéž u nižšího ukazatele
        OR #30
        LD (HL),B    ;A dej oba bajty na svá místa
        LD (DE),A
        INC DE       ;Nižší ukazatel o 1 adr.výš
        JP NXTCVN    ;Opakuj
    
```

Jednoduchým zásahem můžete rutinu upravit pro převod na jiné báze, čili číselné soustavy v ASCII vyjádření (s max. bází 36). Níže uvedenou subrutinu připojte na konec rutiny a obě instrukce OR 30H v rutině (5. a 8. instrukce od konce) změňte na CALL BASASC. Jde o nám již známý převod na "širší abecedu".

```

BASASC ADD A,30H
        CP 3AH
        RET C
        ADD A,7
        RET
    
```

11/Konverze dek-ASCII/32-bitové DK bin.

Rutina bere v úvahu znaménko, které obsadí nejvyšší bit bin. čísla. Použito je obligátní násobení číslem 10. Retezec ASCII znaků musí být uložen s nejvyšším bajtem na nejnižší adrese a zakončen nečíselným znakem.

Algoritmus:

```

Když je 1.znak "-", pak indikuj negaci
Vynuluj 32-bitové pole
Nastav chybový bajt
Proveď adjustaci pro správné přečtení 1.bajtu
Opakuj:
    Když bajt není zakonč.znakem, pak
        vynuluj chybový bajt
        převed číslici na bin.tvar
        pole=pole*10+čísllice
    Když dojde k přetečení za bit 30,pole, pak
        signalizuj přetečení
    Dokud nenarazíš na zakonč. znak nebo přetečení, zpět
    Když bez chyby a indikována negace, pak ji proved
    
```

Délka: 88 bajtů

Počet cyklů: max. 4569 (vyjma ev. představených nul)

Vstup: v HL adr. 1. bajtu řetězce ASCII znaků

Výstup: při Z=0 konverze O.K.; v CY znaménko, v HL adr. zakonč. znaku, v BCDE DK bin. číslo

při Z=1 chyba; HL beze změny, bit 6, C=log. 1, B=0 když řetězec začíná znakem "-"

když CY=0, před zakonč. zn. není žádná platná číslice v DE adr. zakonč. znaku

když CY=1, přetečení, v DE adr. "přetečené" číslice

Použité registry: HL, DE, BC, F

Upozornění: subrutina negace SNEG4 je v kapitole 3, odst. 13/

```

SADB4  PUSH  AF          ; Uschova reg. AF
      EX   DE, HL       ; Do DE adr. řetězce
      LD   A, (DE)      ; Je 1. znak "-"?
      SUB  "-"          ; Když AND, Z=1 bude indikovat negaci
      PUSH AF          ; Uschova indikátoru
      PUSH DE          ; a adr. 1. bajtu
      LD   HL, 0        ; Vynulování 32-bitového pole, složeného
      PUSH HL          ; z HL (vyšší) a 2 bajtů zásob. (nižší)
      LD   B, 1         ; Nastavení chybového bajtu na 1
      JR   Z, SADLP     ; Když je 1. znak "-", skok
      ADD  A, 2         ; Je to "+"?
      JR   NZ, SADDL    ; Když NE, nebude zvýšen ukazatel adresy

SADLP  INC   DE         ; Jdi na další adresu řetězce
SADDL  LD   A, (DE)     ; a přenes z ní znak do reg. A
      ADD  A, #C6       ; Převod jej na bin. tvar
      JR   C, ENDLP     ; Je větší než 9?
      SUB  #F6         ; Nebo menší než 0? Pak směr závěr
      JR   C, ENDLP     ; Jinak uschovej adresu řetězce
      PUSH DE          ; a nastav čítač sčítací smyčky
      LD   B, 2

SAINLP EX   (SP), HL    ; Prohod nižší a vyšší bajty pole
      PUSH BC          ; Uschovej čítač v B
      LD   C, A         ; Číslice do C
      XOR  A           ; Vynuluj CY
      ADD  HL, HL       ; AHL střídavě vyšší a nižší část pole
      ADC  A, A         ; násobená dvěma
      LD   B, A         ; Přenos z pole do BDA=část pole*2
      LD   D, H
      LD   E, L
      ADD  HL, HL       ; AHL=část pole*4
      ADC  A, A         ; *8
      ADD  HL, HL       ; *8
      ADC  A, A         ; *8
      ADD  HL, DE       ; a ve výsledku *10, tj. (*8)+(*2); vý-
      ADC  A, B         ; sledek je v AHL

      LD   B, 0         ; B=0 pro tříbajtový
      ADD  HL, BC       ; součet s CY do AHL, CY teď v reg. A
      ADC  A, B         ; Obnovení čítače smyčky v B
      POP  BC           ; Zopakuj ještě pro druhou část pole
      DJNZ SAINLP

      POP  DE           ; Když obě hotovy, obnov adr. řet. v DE
      JR   NZ, SAOVFW   ; Při přetečení směr závěr
      BIT  7, H         ; Nastalo přetečení za bit 30, pole?
      JR   Z, SADDL     ; Když NE, pokračuj

SAOVFW INC   B          ; Při chybě bude B=1
ENDLP  DEC   B          ; i Z=1
      JR   Z, SAERR     ; V tom případě směr konec
      POP  BC           ; Jinak vyšší část pole do BC
      POP  AF           ; Odhoz 1. adr. řetězce ze zásobníku
      POP  AF           ; Indikace negace v Z
      PUSH HL          ; Vyšší
      PUSH BC          ; a nižší část pole do zásobníku
      EX   DE, HL       ; Ukazatel adr. řetězce do HL
      CALL Z, SNEG4     ; Při potřebě negace bin. čísla ji proved
      POP  DE           ; Výsledek ze zásobníku do BCDE
      POP  BC
      LD   A, 0         ; Vynulování Z bez vlivu na CY
      DEC  A
      JR   SAEND       ; Skok na závěr

SAERR  POP  BC          ; Odhoz nižší část pole ze zásobníku
      POP  HL          ; Do HL 1. adr. řetězce
      POP  BC          ; Odhoz indikace negace

SAEND  EX   (SP), HL    ; Obnovení pův. obsahu reg. A přes zásobník
      LD   A, H
      POP  HL          ; Do HL adr. zakonč. znaku
      RET
    
```

12/ Konverze dek-ASCII/BCD

Rutina zjišťuje, zda je počet ASCII číslic lichý. V tom případě do 4 vyšších bitů nejvyššího bajtu pole BCD číslic umístí BCD nulu. Když je ASCII číslo uloženo s představenými mezerami, rutina je převádí na představené BCD nuly, s nimiž pak už můžeme podniknout, co libo (nechat či odstranit, podle programové situace).

Algoritmus:

Opakuj:

Přečti ASCII číslici
Přesuň cílové vyšší 4 bity do cílových nižších 4
Přesuň nižší 4 bity číslice do cílových vyšších 4
Adresuj další ASCII číslici
Zmenš čítač o 1
Když čítač nenulový, pak
Přesuň cílové vyšší 4 bity do cílových nižších 4
Přesuň 4 nižší bity číslice do cílových vyšších 4
Adresuj další ASCII číslici
Adresuj další cílový bajt
Zmenš čítač o 1
Jinak přesuň cílové vyšší 4 bity do cílových nižších 4
a umístí nulu do cílových vyšších 4 bitů
Dokud čítač nenulový, zpět

Délka: 19 bajtů

Vstup: v DE adr. nejnižšího bajtu řetězce ASCII znaků
v HL adr. posledního bajtu cílového pole BCD číslic
v B počet ASCII znaků v řetězci

Výstup: v DE adresa před řetězcem
HL může adresovat nejvyšší bajt pole BCD číslic nebo bajt před ním; B=0; A beze změny

SNBF	LD	A, (DE)	;Do reg.A ASCII znak a rotace jeho niž.4
	RRD		;bitů (dek.hodnoty) do vyš.4 bitů (HL)
	DEC	DE	;Na další adresu ASCII řetězce
	DEC	B	;Zmenšení čítače ASCII znaků o 1
	JR	Z, ODD	;Když vše hotovo, ale počet lichý, skok
	LD	A, (DE)	;Do reg.A ASCII znak a opět rotace jeho
	RRD		;dek.hodnoty do vyš.4 bitů (HL), přičemž
	DEC	DE	;předchozí hodnota se posune na 4 niž.b.
	DEC	HL	;Na další adresy ASCII řet. i BCD pole
	DJNZ	SNBF	;Opět zmenšení čítače, vše převedeno?
	RET		;Když AND, návrat
ODD	XOR	A	;Vynulování reg.A a přenos BCD nuly na
	RRD		;vyšší 4 bity (HL), pův. obsah těchto bi-
	RET		;tů přejde na niž.4 bity (HL); návrat

13/ Konverze BCD/dek-ASCII s pravou zarážkou

Velmi užitečná rutina, která najde užití při mnohých přenosech číselné informace na obrazovku nebo tiskárnu. Čísla jsou hezky zarovnána k pravému okraji. Rutina ignoruje všechny představené nuly, tj. ty, které stojí před vlastním číslem, a jsou tedy pro jeho hodnotu bez významu. V programové práci však tyto nuly mívají velký význam, a když v takovém případě chybějí, musíme je dodat. Jak už je pro programování symbolické, to, co jsme těžce vystavěli, někdy potřebujeme rychle zbořit (a naopak).

Algoritmus:

Dokud je BCD bajt=0 a bajtový čítač větší než nula, opakuj:
Přeskoč 2 adresy cílového pole
Adresuj a přečti další BCD bajt
Když je čítač=0, pak nastav čítač na 1
Když vyšší 4 bity BCD bajtu=0, pak
adresuj další ASCII bajt
Přesuň 4 nižší BCD bity do nižších 4 bitů ASCII bajtu
Nastav 4 vyšší bity ASCII bajtu na hodnotu 30H
Adresuj další BCD a ASCII bajt
Zmenš čítač o 1
Dokud je čítač větší než 0, opakuj:
Přesuň vyšší 4 bity BCD bajtu do 4 nižších bajtu ASCII
Nastav 4 vyšší bity ASCII bajtu na hodnotu 30H
Adresuj další ASCII bajt
Přesuň 4 nižší bity BCD bajtu do 4 nižších bajtu ASCII
Nastav 4 vyšší bity ASCII bajtu na hodnotu 30H
Adresuj další BCD a ASCII bajt
Zmenš čítač o 1

Délka: 35 bajtů

Vstup: v HL adr. nejvyššího bajtu BCD pole
v DE adr. 1. cílového bajtu pole ASCII znaků
v B počet BCD bajtů pro konverzi

Výstup: v HL adr.posl.bajtu BCD pole+1
v DE " ASCII pole+1
B=0, A beze změny

Použité registry: HL,DE,B,AF

;POZOR - startovací adresa je BFSR!

```

BFSR1  INC  DE      ;Přeskoč další dva cílové bajty, neboť
        INC  DE      ;BCD bajt je nulový
        INC  HL      ;Na další BCD bajt

BFSR    LD   A,(HL)   ;Do reg.A BCD bajt
        AND  #FF      ;Zjistí, zda je nulový
        JR   NZ,BFSR2 ;Když NE, na konverzi
        DJNZ BFSR1    ;Když ANO, pro další bajt, když ještě je
        INC  B        ;Nastav čítač na 1 pro poslední nulu

BFSR2   AND  #F0      ;Vynuluj nižší 4 bity a zjisti, zda vyšší
        JR   Z,BFSR4   ;nejsou nulové; když ANO, skok
        LD   A,#30     ;Jinak připrav konverzi na ASCII

BFSR3   RLD          ;A=30H+4 vyš.bity (HL)
        LD   (DE),A    ;Hotový znak ASCII do pole ASCII znaků
        RRD          ;Rotace do pův.stavu v poli BCD

BFSR4   INC  DE      ;Na další bajt v ASCII poli
        LD   A,#30     ;Příprava na ASCII konverzi
        RRD          ;A=30H+nižší 4 bity (HL)
        LD   (DE),A    ;Hotový ASCII znak do pole ASCII znaků
        RLD          ;Uveď BCD bajt v poli BCD do pův.stavu
        INC  HL        ;Na další adresu pole BCD
        INC  DE        ;i ASCII znaků
        DJNZ BFSR3     ;Opakuji pro všechny BCD bajty
        RET            ;Návrat
    
```

14/ Konverze BCD/dek-ASCII k levé zarážce

Budeme-li potřebovat sesunout čísla k levému okraji, můžeme použít tuto rutinu. Proces umístění ASCII znaků k opačné zarážce a o 2 bajty kratší kód této rutiny jsou jedinými rozdíly mezi touto a předchozí rutinou.

;POZOR - startovací adresa je BFSL!

```

BFSL1  INC  HL      ;Na další BCD bajt

BFSL    LD   A,(HL)   ;Vše ostatní viz předcházející rutina
        AND  #FF
        JR   NZ,BFSL2
        DJNZ BFSL1
        INC  B

BFSL2   AND  #F0      ;Zde je rozdílné umístění instrukce
        JR   Z,BFSL4   ;
        LD   A,#30

BFSL3   RLD          ;
        LD   (DE),A
        RRD
        INC  DE

BFSL4   LD   A,#30
        RRD
        LD   (DE),A
        RLD
        INC  HL
        INC  DE
        DJNZ BFSL3
        RET
    
```

15/ Konverze 16ti bitů/16 znaků ASCII

Někdy narazíme na potřebu zjištění bitového obrazu nějakých dat - zvláště když hledáme nějaké souvislosti třeba pro kompresi dat nebo při pokusech s přenosem dat přes interfaci či na paměťová média apod. Tehdy nám (třeba pro přípravu výpisu jednotlivých bitů na tiskárnu) přijde vhod další rutina. I když v předchozích rutinách byly posuny o 1 bit nahoru pomocí součtu už reprezentovaný více než dost, pro upřesnění:

Posun 16 bitů o 1 bit nahoru v párovém registru HL provedeme součtem ADD HL,HL, což je ekvivalent 8-bitového posunu v reg.A, asice SLA, stejně tak ADD A,A (zároveň je tato operace násobením dvěma). Hraje-li před posunem svou roli stav CY, pak ekvivalentem 8-bitové instrukce RL je 16-bitová ADC HL,HL, při

níž se CY přičte k bitu 0, HL, tedy vlastně se na toto místo "nasune", podobně jako při rotaci RL. Co platí pro 16-bitové součty s reg. HL, platí i pro 8-bitové sečítání reg. A s jinými. Tento princip je v následující rutině bohatě využit.

Algoritmus rutiny je velmi jednoduchý. Každý bit je přičten k ASCII kódu nuly, tj. k 30H. Tak bude výsledný znak buď "0" nebo "1".

Délka: 16 bajtů

Počet cyklů: 781

Vstup: v HL 16 bitů bin. čísla, v DE 1. adr. cílového pole ASCII

Výstup: HL=0, DE bude o 16 větší.

Použité registry: HL, DE

```
CVBA  PUSH AF          ;Uchování obsahu reg. užitých rutinou
      PUSH BC
      LD  B, 16        ;Čítač 16ti bitů
      LD  C, "0"       ;Do C ASCII nula

CVBALP XOR  A          ;Vynulování posledního obsahu reg. A
      ADD HL, HL        ;"Vysunutí" bitu 15, HL do CY
      ADC A, C          ;a jeho přičtení k reg. A spolu s 30H
      LD  (DE), A       ;ASCII kód do cílového pole ASCII znaků
      INC DE            ;Na další adresu ASCII pole
      DJNZ CVBALP       ;Opakuj pro všech 16 bitů
      POP BC           ;Obnov obsahu registrů
      POP AF
      RET              ;Návrat
```

Pokud byste nechtěli použít reg. C, změňte instrukci LD C, "0" na LD A, "0", a místo prvních tří instrukcí smyčky CVBALP dejte:

```
CVBALP RRA
      ADD HL, HL
      RLA
```

16/ Konverze znaků textového souboru

Jak ze zkušenosti víte, vlivem nejednotnosti úprav znakových kódů pro naši mateřštinu v různých slovních mikroprocesorech jsou texty napsané jedním editorem někdy nečitelné na jiném. Než dojde k jasné standardizaci rozmístění písmenek naší abecedy nad kódy ASCII, budeme občas potřebovat provést konverzi nestejných kódů pro tuž písmenka. Následující rutina tento problém řeší tabulkou konverzních párů, v nichž první kód je hledán v textu, aby byl v případě nálezů nahrazen kódem v páru druhým. Před takovouto konverzí musíme provést vlastní detektivní pátrání, abychom zjistili, které kódy jsou odlišné, jaká je jejich hodnota a jakými kódy je budeme nahrazovat. Protože rozdílnosti jsou většinou patrné na první pohled, pátrání nezabere moc času.

Délka: 41 bajtů a 5 pro data (plus tabulka)

Vstup: viz data

Výstup: BC za posl. adr. textu, DE=0

Použité registry: HL, DE, BC, AF

```
ADRTEX EQU  XXXX      ;1. adr. textového pole
DELTEX EQU  NNNN      ;Délka text. pole (počet znaků)
PARY    EQU  NN       ;Počet konverzních párů v tabulce

START   LD  BC, ADRTEX ;Inicial. registrů
      LD  DE, DELTEX
      XOR A          ;Vynulování CY a reg. A
      LD  H, A       ;a reg. HL
      LD  L, A
      SBC HL, DE     ;Negace počtu kódů v textu, výsledek
      EX  DE, HL     ;do DE jako čítač počtu testů

DALSI   LD  HL, ADRTAB ;Do HL 1. adr. tabulky
      LD  A, PARY     ;Celý počet konv. párů do reg. A
POROV   PUSH AF       ;Zmenšující se počet porov. kódů do zás.
      LD  A, (BC)     ;Kód z textu do BC
      CP  (HL)        ;Porovnání s kódem z tabulky
      JR  Z, KONV     ;Když shodné, jdi na konverzi
      INC HL          ;Jinak pro další kód v tabulce
      INC HL
      POP AF          ;Čítač počtu konv. párů
      DEC A           ;zmenší o 1
      JR  NZ, POROV   ;Když ještě v tab. zbývají, porovn. dál
      JR  TEST        ;Jinak test na konec textu

KONV    INC HL        ;Jdi na tab. adr. měněného kódu
      LD  A, (HL)     ;a přes reg. A
      LD  (BC), A     ;jej umístí do textu
```


TEST	POP	AF	;V zásobníku jdi k adrese návratu
	INC	BC	
	INC	DE	
	LD	A,D	;Všechny znaky otestovány?
	OR	E	
	RET	Z	;Když AND, návrat
	JR	DALSÍ	;Jinak pokračuj s dalším znakem textu
ADRTAB	DEFB	NN,NN	;Začátek tabulky konverzních párů
	DEFB	NN,NN	

Převod textu o délce cca 20000 znaků při 20ti rozdílných kódech trvá průměrně 10 vteřin (CPU 4MHz). Tento čas by bylo možno zkrátit zařazením testu na mezery (objevují se nejčastěji) a vymezením jednoho či dvou intervalů, v nichž se převáděné kódy nenacházejí. Kdyby pak rutina na takový kód narazila, neprovedla by porovnávání s kódy v tabulce a rovnou by šla na další znak v textu. Rychlost hledání v tabulce bychom mohli ještě zvýšit použitím binárního prohledávání, jak o něm hovoří kapitola o tabulkách.

V případě, že tiskárna, na níž chceme vytisknout text, "nerozumí" jiným než ASCII znakům, musíme provést převod textu na tyto kódy. Jinými slovy - odstranit všechna diakritická znaménka a přitom nezměnit to, co je "pod nimi". Tedy z ě musí zůstat vždy jen n, z á jen a apod. Kde je rozdíl mezi kódy znaků bez dia-znamének a s nimi rovných 128, je věc jednoduchá - provedeme pouhou inverzi nejvyššího bitu osmibitového kódu. Kde tomu tak není, tam je třeba provést konverzi podobnou předchozí. Takovýto převod můžeme provést destruktivně, nebo beze změny textu uloženého v paměti. Destruktivní převod provede konverzi přímo na adresách uložení znaků. Nedestruktivní postupně převádí každý kód tak, jak přichází do rutiny výstupu na tiskárnu. Konverzní rutina by v takovém případě byla součástí výstupní rutiny.

KAPITOLA 6

Komprese a expanze dat

Aneb - jak zhubnout a přesto nic neztratit ze svého obsahu. Představte si úkol, před nímž se ocitnete - vytvořit slovník jazyka (třeba češtiny) tak, aby byl přístupný při práci se slovním procesorem a aby uměl rozpoznat překlepy v textu, který jsme napsali, ba aby uměl i dělit slova na konci řádky. Pokud bychom chtěli do slovníku uložit slova ve všech jejich možných jazykových tvarech, brzy bychom začali být v úzkých, nehledě na to, že by takový slovník dopisovali naši potomci. Jeho rozsah by byl srovnatelný s Ottovým naučným. Nezbyvá, než se zamyslet (hloupá hlava přidělává rukám práci).

Tak třeba slova jako přednes, předkus, představa atd. mají předponu "před" (i ta předpona ji má). Vida, jen tímto nápadem jsme se zadarmo zbavili pěkného kvanta slov a náš slovník vydatně pohlubl. Aby s před/příponami mohl počítač rychle operovat, dáme jim nějaké kódy. Předpon víc než 256 nebude, stačí nám na ně 1 bajt. Ale ouha, jak má počítač poznat, že nějaký bajt je předponou, jiný příponou... a bůhví, co nás ještě napadne? Srazíme předpony a přípony na 7 bitů, bude jich po 128. Bit 7 bajtu bude při log.1 znamenat předponu, při log.0 příponu. Jenže... českých přípon asi bude víc než 128. Nezbyde, než to vzít "přes dva bajty"... nebo použijeme jako 9. indikační bit jeden ze stavových indikátorů? Před- a přípony seřadíme podle abecedy, aby se v nich dalo rychle hledat. Jádro slovníku budou tvořit slovní kořeny nejužívanějších slov - pro intelektuály tak kolem 10000. Když bude mít každý v průměru asi 8 písmen, je to pořád slušná hromada bajtů. Jakou dietu jim předepsat, si ukážeme o něco níže.

Pokračujeme v úvahách nad zadaným úkolem. Představme si, že máme v počítači text a chceme zjistit chybně zapsaná slova jejich porovnáním se slovy ve slovníku. Jak se dostat ke kořenu slova? Osaháme slovo z obou stran a zjistíme, zda nemá nějakou před- nebo příponu. Anglicky mluvící programátoři, kterým nechybí humor, tuto operaci nazvali stripping, v žertovně přeneseném smyslu "umělecké obnažování". Každé slovo tedy rozebereme na jednotlivé části, z nichž jedna určité bude jeho kořenem. Ten pochopitelně může stát i sám o sobě. Zjištěný kořen porovnáme s kořeny ve slovníku. Buď bude nalezen a vše je v pořádku, nebo nalezen nebude. Co pak? Stát se může leccos. Počítač nemusí (když si neporadí), ale může provést opravu na tvar, který je danému kořenu nejbližší, a to buď přímo, nebo nám takovou opravu nabídne a počká, co my na to. Zde může dojít k různým groteskním roztodivnostem, zvláště není-li námi myšlené slovo zaneseno ve slovníku. Program také musí umět oddělit slova, mezi nimiž jsme nenapsali mezeru. Není-li dobře domyšlen, mohou nastat problémy podobné tomuto:

Je slova "přesně" počítač svlékne domnělou předponu přes-. Jako kořen vezme "ně". Ale kořenem je pro něj i slovo přes jakožto předložka. A tu si řekne: "Ten trouba zase nenapsal mezeru!" A upraví větu na tvar: Pepa přišel s holkama přes ně. A kdyby to hned překládal do angličtiny, bylo by z toho: Joe came with girls over them. A už by to nikdy nikdo nedal do kupy, zvláště jsa-li Angličan. Jak vidno, musí tu být ještě další operace, která bude zjišťovat, zda "předstíraná" předpona není částí kořenu slova. Ale i - zda nejde o dva kořeny, mezi nimiž chybí mezerka. V takovém případě počítač současných schopností nesmí zasáhnout do textu sám, ale nabídnout případnou opravu člověku.

Pokud jde o dělení slov, už to, že program díky našemu komprimovanému slovníku pozná předpony a přípony, znamená, že právě toho můžeme při dělení využít. Za předponu dáme znak "-" a zbytek slova na začátek nové řádky. Při příponě přijde "-" za obnažené slovo a přípona na další řádku. Ale pozor - co se slovem jako třeba přesnídávka? Počítač mu nesmí svléknout "přes"! A ke všemu může mít slovo i více než jednu předponu! Dělení uvnitř kořene slova by si vyžádalo další drinu i paměť.

Bez komprimace a souvisejících operací, založených především na důvtipu programátora, by byl slovník slovního procesoru nemyslitelný. Nejen tedy hraje komprimace dat velkou roli. Vztahuje se prakticky k jakýmkoli "velkým" datům - v databázích, kalkulačních programech, ale i při přenosu dat na záznamová média, přes modemy apod. Jedna z typických komprimací, kterou "ze zvyku" ani jako komprimaci nebereme, jsou ASCII kódy, za nimiž se skrývá celá typografie znaků, přenášených na obrazovku

nebo tiskárnu. Co jeden ASCII bajt, to kupa bitů grafického tvaru znaku. Což je dokladem toho, že vedle programu, který data komprimuje, většinou potřebujeme i program, který je bude umět expandovat.

1/ Komprese velkých ASCII znaků

Rutina pracuje pouze se znaky v intervalu ASCII kódů 32..95 (20H..5FH), tedy jen s velkými písmeny. S menší ztrátou efektivity lze upravit i pro malá. Komprese vychází z faktu, že uvedené kódy potřebují jen 6 bitů. Tak pro 4 znaky vystačíme se 3 bajty. Uspora je 25 procent. Rutina má dva vstupy - kompresní PAC a expanzní UNPAC.

Algoritmus funkce PAC:

Vynuluj sekvenční čítač bajtů

Dokud čítač znaků není nulový, opakuj:

Nastav sekv.čítač

přečti zdrojový znak a odečti od něj 20H

posuň jeho bity vlevo tolikrát, kolik je obsah sekv.čít.*2

připoj výsledné 4 vyšší bity k bajtu na cílové adr.-1

ulož 4 nižší bity na cílovou adresu

přejdi na další adresu zdroje a cíle

proved adjustaci obsahu sekv.čítače mod 4

Algoritmus funkce UNPAC:

Vynuluj sekvenční čítač bajtů

Dokud čítač znaků není nulový, opakuj:

nastav sekv.čítač

přečti bajty ze zdroj.adr.-1 a zdroj.adresy

posuň jejich bity vpravo tolikrát, kolik je obsah sekv.č.*2

izoluj znak jako součet bitů 5 až 0 plus 20H

ulož znak na cílovou adresu

když je sekv.čítač menší než 4, pak na další adr.zdroje

přejdi na další cílovou adresu

proved adjustaci obsahu sekv.čítače mod 4

Délka: 64 bajtů

Vstup: v HL zdroj.adr., v DE cíl.adr., v BC počet bajtů pro kompresi/expenzi

Výstup: v HL adr.posl.zdroj.bajtu+1, v DE adr.posl.cíl.bajtu+1
BC=0

PAC	LD	A, #80	; Vynuluj sekv.čítač, signalizuj PAC
	JR	B, INC	; Skok na kompresi
UNPAC	XOR	A	; Vynuluj sekv.čítač, zahaj expanzi
BINC	INC	B	; Zvětšení znakového čítače o 1
LOOP	PUSH	BC	; Uchovej znak.čítač
	PUSH	HL	; a adr.zdroje
	INC	A	; Signalizuj 1.bajt ze sekvence čtyř a
	LD	C, A	; ulož jej do reg.C
	ADD	A, A	; Do CY signál pro komp./exp.
	LD	B, A	; Výsledek posunu do B
	LD	A, (HL)	; Další zdroj.bajt do A
	JR	C, DOPAC	; Když CY=1, skok na kompresi
	DEC	HL	; Jinak expanze
	LD	H, (HL)	; Další zdroj.bajt do H
RISH	RR	H	; Posun obsahu reg.HA vpravo, až budou
	RRA		; bity 5-0 obsahovat expandovaný znak
	DJNZ	RISH	; Opakuj, dokud sekv.čítač nenulový
	AND	#3F	; Maskou odstraň nechtěné bity
	ADD	A, #20	; Převed na interval kódů 20H-5FH
	JR	LODE	; Skok pro uložení znaku
DOPAC	SUB	#20	; Převed z interv.20H-5FH na 00H-3FH
	LD	L, A	; Převedený znak do HL
	LD	H, 0	
LESH	ADD	HL, HL	; Posun znaku v reg.HL vlevo
	DJNZ	LESH	; Opakuj, dokud sekv.čítač není nulový
	LD	B, L	; Nejnižší pozůstalé bity do B
	DEC	DE	; Protože může být znak v cílovém bajtu
	LD	A, (DE)	; nekompletní, doplní se maskováním
	OR	H	; reg.H do znaku v reg.A
LODE	LD	(DE), A	; Konvertovaný bajt do cílového pole
	POP	HL	; Obnov zdroj.adr.v HL
	LD	A, C	; Do A sekvenční čítač
	AND	#83	; mod 4 pro správný sled sekvence
	JR	Z, DINC	; Skok na DINC, když expanze 4.bajtu
	CP	#81	; Skok na HINC, když komprese 4.bajtu
	JR	C, HINC	; nebo expanze 1.či 3.bajtu
	INC	DE	; Nižší bity expandovaných 1.a 3.bajtů
	EX	DE, HL	; budou uloženy
	LD	(HL), B	; z reg.B do další volné cílové adresy
	EX	DE, HL	

HINC	INC	HL	;Na další zdroj.adr. (kromě exp.4.bajtu)
DINC	INC	DE	;Na další cíl.adr.
	POP	BC	;Obnovení znak.čítače
	DEC	C	;a jeho zmenšení o 1
	JR	NZ, LOOP	;Opakuj pro všechny bajty v C
	DJNZ	LOOP	;i B
	RET		;Návrat

Chtěl bych upozornit na způsob, jakým je testován stav čítače v párovém reg.BC. Obvyklý postup by byl LD A,B; OR C. V rutině však nesmí být dotčen obsah reg.A, proto je zvoleno uvedené. Tomu ovšem musí předcházet zvýšení obsahu reg.B o 1, aby v případě jeho nulové vstupní hodnoty nedošlo při exekuci instrukce DJNZ k jeho snížení na FFH, a tím i k vykonání nechtěných 256 průchodů smyčkou navíc. Myslím, že lepší by bylo střídavě ukládat a odebírat obsah reg.A v oblasti dat a vrátit se k tradičnímu testu obsahu párových registrů.

Uprava rutiny pro komprimaci malých písmen, jejichž kódy potřebují 7 bitů (tedy o jeden víc), spočívá v tom, že před ně uložíme nějaký málo používaný znak a je samotné převedeme na velká písmena. Když skončí série malých písmen, musíme ovšem zase předřadit nějaký speciální znak, upozorňující na tuto skutečnost. Při expanzi bude postup analogický - kódy malých písmen obdržíme zpětným přičtením rozdílu, přičemž speciální znaky kromě jejich signalizační funkce ignorujeme. Jistá výhoda tohoto uspořádání je v tom, že při ignoraci speciálních znaků můžeme obdržet všechna písmena velká.

2/ Expanze komprimovaných tokens

Tokens jsou kódy, většinou reprezentující znakové řetězce abecedy. Známé jsou tokens basicových příkazů u Spectra. Stiskem jednoho tlačítka se na obrazovce objeví několikaznakový příkaz. Program vyhledává příslušný řetězec znaků v tabulce znaků podle kódu stisknutého tlačítka.

Tokenizování můžeme využít při komprimaci textu, nebo podobně jako je tomu u spectrovských basicových příkazů. Následující rutina dovoluje použít 128 tokens v rozsahu kódů 128-255. Tokenizované řetězce jsou ve zvláštní tabulce, mezi sebou jsou odděleny nulou. Výsledný text může obsahovat kódy v intervalu 2-127, tj. včetně řídících kódů ASCII. V rutině je volání subrutiny OUCH, která provede přenos expandovaných znaků na místo určení (obrazovka, tiskárna apod.). Subrutina zde uvedena není, protože její konstrukce je závislá jednak na uživatelem zvolené funkci (což je věc programátora) a pochopitelně i na systému daného počítače (což je věcí věci toho, jak dalece jej programátor využívá). Rutina se "zasekne", když je zpracováván token kód větší než největší token v tabulce token-řetězců.

Algoritmus:

```

Přečti bajt ze zdroje a adresuj další
Dokud bajt není nulou, opakuj:
  Když je bajt ASCII znakem, pak
    přenes znak na výstup OUCH
  Jinak najdi token-řetězec v tabulce
    a přenes jeho znaky na výstup

```

Délka: 37 bajtů

Vstup: v HL první adresa tokenizovaného textu, který musí být zakončen nulou
v DE 1.adr.tabulky token-řetězců, které jsou odděleny nulou; řetězce mohou rovněž obsahovat tokens; pokud řetězec začíná nulou, okamžitý návrat

Výstup: v HL adr.nulbajtu+1 komprimovaného textu, DE beze změny
Použité registry: HL,DE,B,AF

TOKN	LD	A, (HL)	;Do A zdroj.bajt
	INC	HL	;Adresuj další
	OR	A	;Je to nulbajt?
	RET	Z	;Když AND, návrat
	PUSH	HL	;Ulož adr.zdroje
	BIT	7,A	;Je to znak ASCII nebo token?
	CALL	NZ,XPAND	;Když token, volej expanzní část
	CALL	Z,OUCH	;Když ASCII znak, pošli jej na výstup
	POP	HL	;Obnov adr.zdroje v HL
	JR	TOKN	;Zpět pro další zdroj.bajt
XPAND	PUSH	AF	;Uchova indikátorů
	PUSH	DE	;a 1.adr.tabulky token-řetězců
	AND	#7F	;Vynulování bitu 7,A, aby zůstal jen
	JR	Z,XGOT	;token; když je to nulbajt, skok na XGOT

```

XFIN LD B,A ;Token do reg.B jako čítač nul
LD A,(DE) ;Do reg.A bajt z tabulky token-řetězců
INC DE ;Adresuj další
OR A ;Je to nulbajt?
JR NZ,XFIND ;Když NE, hledej dál
DJNZ XFIND ;Když B=0, našli jsme hledaný token-řet.

XBOT EX DE,HL ;Dále čti token-řetězec jako zdroj.text
POP DE ;Obnov l.adr.tabulky v DE
CALL TOKN ;Přečti řetězec a pošli znaky na výstup
POP AF ;Obnov stav indikátorů
RET ;a vrať se nebo pokračuj v textu
    
```

Instrukce RET zde hraje dvojí roli. Buď jako konečný návrat při nalezení nulbajtu, nebo návrat za instrukce volání CALL NZ či CALL Z. Rutinu můžeme využít např. ve slovním procesoru, kde se nám po stisku určitých tlačítek ve stanoveném módu vypíše na obrazovku (pak i na tiskárnu) řetězec, které užíváme nejčastěji a nebaví nás je stále dokola vypisovat (např. naši adresu apod.). Nebo když píšeme text, v němž se často objevuje dlouhé slovo, třeba s (rozšířeným) slovním kořenem elektron-. Místo kořene budeme psát nějaký určený znak, např. "!"; elektronizace pak bude v textu zapsána jako izace, neelektronický jako nelický. Po skončení zápisu zavoláme funkci, která všude, kde je znak "!", doplní slovní kořen.

Vzhledem k tomu, že rutina pracuje i s řídícími kódy, můžeme do subrutiny DUCH pro výpis znaků na obrazovku dát nějaké své vlastní, pomocí nichž umístíme vypisovaný text na chtěnou pozici na řádce. Tak bychom mohli třeba adresu napsat hezky do sloupečku pod sebe. V tom případě před vlastní textové znaky každého řetězce umístíme řídící kódy. Např.:

13 44 Jméno 0 10 13 44 Ulice 0 10 13 36 PSC město 0.

Kurzor při psaní ve slovním procesoru umístíme kamkoli na řádku, od níž budeme chtít adresu vypsat a stiskneme potřebné tlačítko, pod nímž jsou ukryty 3 token kódy pořadí oněch 3 řetězců v tabulce. Jinak 10 je kód pro LF, tedy posun na novou řádku, 13 pro CR, tj. návrat na levý okraj řádky a čísla jako 44 a 36 jsou informací o čísle sloupce, od něž má být následující řetězec vypsan. Subrutina DUCH samozřejmě nepropustí řídící kódy včetně čísel sloupců, a výsledek (po stisku jediného tlačítka) bude vypadat následovně:

Pipi Funčochatá
U Puštěného oka 100
123 45 Silónkov

3/ Expanze textu přímo adresovanými řetězci

Rozdíl mezi touto a minulou rutinou je v tom, že zatímco předešlá adresovala řetězce nepřímou (jejich pořadím v tabulce s pevnou l.adresou), tato rutina udává přímou adresu řetězce, který tak může být kdekoli. Tento způsob, se svými různými alternativami, je vhodný pro práci se slovníkem. Adresování řetězců si ukážeme na tomto příkladu:

adresa	znaky a kódy	po expanzi	zisk (bajtů)
8000	... e l e k t r o n 00	elektron	-1
	FF 00 80 FF 00 90 00	elektronický	5
	FF 00 80 i k 00	elektronik	4
	FF 00 80 i k a 00	elektronika	4
	FF 00 80 FF 05 90 00	elektronizace	6
	FF 00 80 k a 00	elektronka	4
9000	... i c k ý 00	ický	-1
9005	i z a c e 00	izace	-1
	...		

Kód FFH signalizuje, že po něm následuje adresa (napřed její nižší, pak vyšší část) uložení řetězce, do něž je část slova komprimována. Nula na konci ukončuje slovo. Pár poznámek na okraj:

Ve slovníku vidíme, že -izace, -ický, -ika jsou přípony, které mohou být reprezentovány kódem, předznamenaným třeba znakem "-". Také proto, abychom mohli slova obnažovat, jak řečeno v úvodu. Níže uvedená rutina nám nabízí jen řešení, uvedené u slov elektronizace a elektronický. I s předponami bychom měli něco udělat. Jak zanést třeba slovo antielektronický? Pro předponu můžeme opět využít znak "-" na začátku slova (počítač pozná, že jde o předponu podle toho, že je před kořenem slova, resp. po předcházející nule, resp. po ještě jedné předcházející

předponě). Přidělíme-li x/ponám anti- kód 22H a -ický kód 99H, bude výsledek komprimace: - 22 FF 00 80 - 99 00. Což je o 2 bajty méně, než by dokázal způsob uvedený (nebude-li celkový počet x/pon větší než 256) a ještě se můžeme oddávat strippingu. Program by měl také rozlišovat mezi x/ponami a složenými slovy. Tak třeba listopad je složen ze dvou kořenů - list a opad. Ani jedna část nemůže mít znak "-". Komprimace je ale jednoduchá: FF XX XX FF YY YY 00, i když komprimát je jen o 1 bajt kratší. Ale zpět k rutině.

Algoritmus:

Přečti znak

Dokud obsah přečteného bajtu není nulový, opakuj:

Adresuj další znak

Když je to kód ASCII, pak jej pošli na výstup

Jinak proved expanzi podle udané adresy

Přečti další znak

Délka: 25 bajtů

Vstup: v HL adresa expandovaného textu, který musí být zakončen nulou, stejně tak každý expanzní řetězec

Výstup: v HL adresa nulbajtu textu

Použité registry: HL, AF

;AR-56 EXPANZE TEXTU

```

MAKMSG LD A, (HL)      ;Přečti znak z textu
        OR A           ;a zkus, jestli není nulbajt
        RET Z          ;Když AND, návrat
        INC HL          ;Adresuj další znak
        CP #80          ;Je to ASCII (max.7FH)?
        JR NC,ESCP      ;Když NE, skok na expanzi
        CALL OUTCH      ;Pošli znak na výstup
        JR MAKMSG       ;Opakuj, dokud neobjevíš nulbajt

ESCP LD A, (HL)         ;Adresa expanzního řetězce do HL přes
        INC HL           ;reg.A
        PUSH HL          ;Ušchovej poslední text.adresu
        LD H, (HL)
        LD L, A
        CALL MAKMSG      ;Pošli na výstup jeho znak.obsah
        POP HL           ;Návrat sem při nalezení nulbajtu; obnov
        INC HL           ;text.adresu v HL
        JR MAKMSG        ;a jdi pro další znak

```

Tuto rutinu můžete i kombinovat s předchozí. Tokens pak budou v rozsahu 80H-FEH. Při FFH bude muset být vždy aktivována poslední rutina. Proto se musí jemně upravit testy na uvedené kódy. Vraťte se zase k povídání o slovnících. Co byste řekli takovémuto způsobu komprimace slovníku?

adresa	znaky a kódy	po expanzi	zisk (bajtů)
8000	00 e l e k t r o n 00	elektron	-2
	09 FF 00 90 00	elektronický	7
	0E i k 00	elektronik	6
	04 a 00	elektronika	8
	15 FF 05 90 00	elektronizace	3
	1A k a 00	elektronka	6
	FF 09 98 FF 05 80 FF 00 90 00	hypertronický	3
9000	i c k ý 00	ický	-1
9005	i z a c e 00	izace	-1
9800	a n t i 01	anti	-1
9805	b e z 01	bez	-1
9809	h y p e r 01	hyper	-1
	...		

Slovník je rozdělen na část slovní, příponovou a předponovou. V části slovní první bajt určuje, o kolik adres zpět (v intervalu 1..254; FFH si ponecháme zvlášť) leží řetězec pro expanzi slova. Je-li bajt nulový, slovo naopak začíná na první vyšší adrese. Jde tedy o obdobu relativního adresování. Algoritmus s kódem FFH zůstává v platnosti. Část příponová má zakončovací kód 00, předponová 01. Protože však leží od pevných adres, není toto rozlišení nutné, ale může se hodit. Zisk z komprimace je mnohem větší, než v případě předešlém. Časová ztráta vyplývající z operací s úvodními bajty slov je minimální. Pochopitelně, že stripping bude muset probíhat trochu jinak, než při vložení znaku "-"; díky němuž je vše jasné a rychlé. Zde bude třeba zjistit, zda potenciální x/pona leží v oblasti x/pon. Výhoda použití těchto oblastí je v tom, že se nemusíme

omezovat na počet 256 x/pon. Ale zase tu vyvěrá problém se složenými slovy. A hned tu máme další oblast dat, v níž může být to, co není žádnou x/ponou, ale může se podobně použít. Např. slovo logie (biologie, zoologie atd.). A hned máme z 5ti bajtů 3. I když -ie je příponou, ale to nám někdy nemusí vadit, pokud tím nezaneseme zmatek do programu. A ještě - nebylo by přece jen lepší dát úvodní adresovací bajt i před x/pony? Vždyť kromě přípony -ický je ještě -ické, -ická a jejich pádové koncovky -ickou, -ických, -ickými atd. Nebo že bychom rozdělili příponu na dva kusy? -ick- a pádové koncovky, kterým bychom přisoudili další datovou oblast? Ale pozor - máme tu ještě zvláštní tvar -ičtí! Ach, ta čeština! Pádové koncovky se vztahují i na podstatná jména a zájmena, další úpravy přípon a koncových proměnlivých znaků je nutno provést pro slovesa. Nebo přijdete na lepší řešení? Akorát snad pozor na to, že v podobných hrátkách bychom mohli pokračovat gerade aus až do Svatého Dyndy. Nevěříte? Slovní část slovníku můžeme upravit zase jinak a lépe:

adresa	znaky a kódy	po expanzi	zisk (bajtů)
8000	00 e l e k t r o n	elektron	-1
	08 FF 00 90	elektronický	8
	09 k	elektronik	8
	0A a	elektronika	9
	08 FF 05 90	elektronizace	9
	08 k a	elektronka	7
	...		

Tentokrát každý úvodní kód ukazuje, kolik znaků z předešlého slova je použito ve slově za kódem. Když bude tento kód menší než 20H, což určitě bude, můžeme odstranit zakončovací nulbajty, které nahradí úvodní kódy (programově test na kód menší než 20H). Když se podíváte na zisk, je u základu elektron značný - z původních 64 znaků (aniž jsem chtěl, magických 64 je tu) zůstalo jen 24! Efektivita komprimace obrovská - přes 60 procent!

Otázkou zůstává, zda má být text takto postupně komprimován již během jeho zápisu do počítače, nebo naráz před přenosem dat na záznam či modem. Práce s komprimovaným textem během zápisu mikroprocesor dost zdržuje, i když výkonnější kolegové už leccos snesou. Počítač by si mohl pobrukovat do algoritmu komprimace během zápisu asi takhle:

Přifitil jsem se do přerušení X a koukám, jestli ten nešika už napsal mezeru (; , -) za to své věčně nedopsané slovo.
Když nenapsal, pak nashle zase v dalším přerušení X
Jinak čapnu slovo a protáhnu je slovníkem
Chyby teď neopravuju, nevím, kam dřív skočit
Když slovo ve slovníku najdu, pak
provedu komprimaci a kódy uložím do cílového pole
Jinak smůla, nechám to, jak to je, a uložím do cíl.pole
Vymažu to slovo z bufferu, kam je ten živočich ukládá a jdu
z přerušení pryč (díkybohu mi teď do toho žádné jiné nepřišlo). Tak ahoj, ale jsem tu zpátky cobydup.

Algoritmus může být i takový, že nebude pracovat s přerušením, ale bude volán po každém zápisu mezery, kdy bude počítač muset zjistit, zda je co komprimovat. Tedy jestli nastala nějaká změna oproti minulému stavu, a vše, co s touto změnou souvisí, adekvátně změnit i v cílovém komprimovaném poli. Samozřejmě, že pro takovéto průběžné úkony musí být počítač o poznání výkonnější než Spectrum a spol.

Nyní si ještě ukážeme, jak by mohl vypadat čistě komparativní slovník pro hledání chyb v napsaném textu, aniž bude mít s komprimací příliš mnoho společného.

adresa	znaky a kódy	
8000	00 e l e k t r 00	
	FF 01 80 01	;Komprimace kořene lekt
	s n i d 00	
9000	00 t 00	
	á r n a 00	
	á v k a 00	
	i c k ý 00	
	i k 00	
	i k a 0	
	i z a c e 00	
	k a 00	
	o n 00	
	o r 00	
	s k ý 00	
	...	

```

9800  a n t i 01
      b e z 01
      h y p e r 01
      n a 01
      p r e 01
      p r e s 01
      ...

```

Když ve slovní části budou jen krátké kořeny slov, komprimace většinou k ničemu nebude, ale sem tam se použít může. Jistou přednost má řešení uvedené u slova lekt, kde se čte část lekt z kořene elektr. Kód 01 škrtá jeden znak "odzadu", tj. "r". Tento kód, který musí být jako třetí za FFH, nesmí být větší než 1FH (20H už je mezera). Tak může sloužit i jako zakončovací znak.

Dejme tomu, že počítač v textu narazí na slovo přesnídávka. Napřed se podívá, zda takové slovo není v tabulce slovních kořenů. Kdyby ano, měl by po starostech. V našem případě tam ale nenajde nic. Započne stripping. Najde nejkratší příponu -ka. Mrkne se do tabulky kořenů. Nic. Zkusí nejkratší předponu pře-. Zase nic. Zkusí přes-...a nic. Přeskočí zase na přípony. Což takhle -ávka? S kořenem -snid- vlezte do tabulky - a ejhle, je to tady! Kdybychom napsali chybně -síd-, počítač by to jednak bral jako chybu, a (byl-li by naprogramován na odstraňování chyb) s dost velkou pravděpodobností by nám nabídl správný tvar. U slova elektronický se bude muset dopracovat ke zjištění, že v něm jsou dvě přípony -on a -ický. Apod. Samozřejmě, že takové porovnávání textu se slovníkem zabere hodně času. Pro osmibitový mikroprocesor je to vyložená dřina. Jinak v podstatě záleží jen na vás, pro jaký způsob komprimace textu či slovníku se sami rozhodnete, který vám bude bližší, a který dokážete dovést do efektivního provedení pro ten který účel. O tom, že komparativní slovník může být ještě - a to podstatně - kratší, si povíme v kapitole Tabulky.

4/ Komprimace pro přenos datových bitů

Jak už bylo řečeno, celý komprimačně-expanzní program pro přenos dat na vnější paměťové médium bude uveden jako součást slovního procesoru v další učebnici. Dále uvedený princip komprimace se opírá o zavedení dalšího pulsu, který bude mít poloviční délku než puls pro log.0. S ním je spojen i trochu odlišný způsob ochrany před přenosovým zkreslením, než je "klasický" (o něm rovněž další učebnice). Vezměme si jakoukoli šňůru bitů z paměti, resp. oblasti uživatelských dat, což bývá nejčastěji text. Bude vypadat třeba takto:

```
0000011100010101111110000101100101111100001010110101011000001
```

Momentálně nás zajímají jen skupiny bitů se stejnou hodnotou, nic jiného. Zleva je to 5-0 (pět nul), 3-1, 3-0, 1-1, 1-0, 6-1, 4-0 atd. Bity se na pásek přenášejí bit po bitu, jako korálky. Každá log. hodnota je převedena na jiný průběh pulsu. U pseudo FM je puls pro bit log.0 dvakrát kratší než pro bit log.1. My to zkusíme jinak, i když tvar pulsů zůstane beze změny. Pro zdar věci je nutno zjistit, zda šňůra bitů začíná log.0 nebo log.1. Pro signalizaci toho přidáme jeden puls na začátek záznamu navíc. Ten nám při čtení záznamu sdělí, zda se začíná jedničkami (delší puls) či nulami (kratší). Počet bitů ve skupinách budeme sečítat v jednobajtovém čítači. Při zápisu tedy spočítáme, kolik je v paměti za sebou stejných bitů a teprve tuto informaci zapíšeme na pásek, přičemž nadbytečných nul v bajtu vlevo před číslem v čítači si nebudeme všimnout. Tato část rutiny by (pouze rámcově!) mohla vypadat takto:

	LD DE, počet adres zdroje	
	LD HL, 1. adr. zdroje	
	PUSH DE	; Uložení počtu adres zdroje
	LD C, FFH	; Přednastavení čítače stejných bitů
bajt	LD B, B	; Čítač bitů bajtu paměti
další	LD A, (HL)	; Bajt paměti do reg. A
další	RRA	; Jeho bity postupně do CY
	INC C	; Zvětšení čítače stejných bitů o 1
	LD E, A	; Uchova rotovaného bajtu do reg. E
modif	JR C, změna	; Při změně log. hodnoty konec načítání
		; a modifikace JR C/JR NC, resp. JR C/JR NC
	LD A, C	; Číslo, repr. počet stejných bitů, do A
	XOR FFH	; Je čítač plný (=FFH)?
	LD A, E	; Rotovaný bajt zpět do reg. A
	JR Z, plný čítač	; Když čítač plný, skok
bitčít	DJNZ dalbit	; Jinak pokračuj pro další bit
	POP DE	; Obnov počet adres zdroje v DE,
	DEC DE	; zmenši jej o 1
	PUSH DE	; a opět ulož do zásobníku

```

LD A,D      ;Zjistí, zda zpracovány všechny bajty
OR E        ;zdroje
JR NZ,další ;Když NE, jdi pro další bajt
POP DE      ;Když AND, posuň se k adrese návratu
RET         ;a vrať se na ni

změna EX DE,HL ;Uchova moment.adr.zdroje do DE
LD HL,modif ;Do HL adr."modif"
LD A,38H    ;Do A operační kód instr.JR C
CP (HL)     ;Je na adr."modif" instrukce JR C?
JR NZ,změna ;Když NE (je tam JR NC), změň na JR C
SUB B       ;Když AND, změň na JR NC
změna LD (HL),A ;Změněný oper.kód uloží na adr."modif"
EX DE,HL    ;Obnov moment.adr.zdroje v HL
plný SCF     ;CY=1 pro koncový signál zápisu
LD A,C      ;Do A stav čítače stejných bitů
LD D,B      ;Do D počet odečtených bitů z adr.zdroje
LD B,9      ;Čítač smyčky "rotace" (8+CY), po ní bu-
           ;de obsahovat počet zapisovaných bitů
rotace RLA   ;Rotuj bit čítače stejných bitů do CY
DEC B       ;Zmenší čítač smyčky "rotace" o 1
JR NC,rotace ;Nuly zleva ignoruj
JR NZ,puls  ;Při první log.1 a nenul.B jdi na "puls"
INC B       ;Čítač stejných bitů byl nulový, zvýš B
           ;(zapiše se jen 1 puls pro log.0)
puls RR C    ;Obsah čítače stejných bitů (počet zapi-
CALL zápis   ;sovaných bitů čítače je teď v B) zapiš
DJNZ puls   ;na pásek
CALL syncpuls ;Všechny jeho bity zapsány, ukonči sync
LD B,D      ;pulem a počet odečtených bitů z adr.
LD C,FFH    ;zdroje do B, přednastav čítač C
JR další    ;a pokračuj v počítání šňůry dalších
           ;stejných bitů

```

Sync puls je použit jako dělicí. Zakončuje zápis čísla, vyjadřujícího počet jednoho či skupiny stejných bitů, které se na magnetofon posílají ve formátu jejich počtu. Přičemž nulová hodnota čítače počtu bitů znamená 1 bit, hodnota 1 pak 2 bity atd. Tak je nejnižší bit využit pro počet až 2 bitů, což bude poměrně dost častý případ. Důležité je rovněž to, že zde nedochází k tomu, k čemu dochází u mnohých komprimačních programů. Nežádka se u nich stává, že komprimát je rozsáhlejší než zdrojová data. Tady se to stát nemůže. Ale abych byl přesný - když se zapisuje jediný bit (čítač vynulován, zapisuje se tedy 1 bit s hodnotou log.0), je v případě zápisu bitu s originální hodnotou log.0 zápis delší o sync puls. Ale naopak při originálním bitu log.1 je zápis kratší o polku delšího pulsu minus délka sync pulsu. Při jakémkoli větším shluku originálních bitů log.1 je pak vlivem komprimace do čítače, tedy i výslednou dobou trvání pulsu, vše o dost kratší. Ukažme si to na tabulce:

log.hodn. orig.bitů	Obsah bit.č. /poč.orig.b.	Počet pulsů dl. kr.	Kolik pulsů klas.met. kr.	Ušetř.počet kr.pulsů(čas)
1	00000000 / 1	- 1	1 dl.	1
0	00000000 / 1	- 1	1 kr.	-
1	00000001 / 2	1 -	2 dl.	2
0	00000001 / 2	1 -	2 kr.	-
1	00000010 / 3	1 1	3 dl.	3
0	00000010 / 3	1 1	3 kr.	-
1	00000011 / 4	2 -	4 dl.	4
0	00000011 / 4	2 -	4 kr.	-
1	00000100 / 5	1 2	5 dl.	6
0	00000100 / 5	1 2	5 kr.	1
1	00000101 / 6	2 1	6 dl.	7
0	00000101 / 6	2 1	6 kr.	1
1	00000111 / 7	3 -	7 dl.	8
0	00000111 / 7	3 -	7 kr.	1
1	00001000 / 8	1 3	8 dl.	11
0	00001000 / 8	1 3	8 kr.	3
1	00001001 / 9	2 2	9 dl.	12
0	00001001 / 9	2 2	9 kr.	3
1	00001011 / 10	3 1	10 dl.	13
0	00001011 / 10	3 1	10 kr.	3
...				
1	01000000 / 65	1 6	65 dl.	122
0	01000000 / 65	1 6	65 kr.	57
1	10000000 / 129	1 7	129 dl.	249
0	10000000 / 129	1 7	129 kr.	120
1	11111111 / 256	8 -	256 dl.	496
0	11111111 / 256	8 -	256 kr.	250

Efektivita prudce roste při delších šňůrách stejných bitů. Bohužel, to je poezie. Próza nás tlačí na začátek tabulky. Nejčastější seskupení budou po dvou, třech bitech. Od tabulkových hodnot musíme ještě odečíst sync pulsu, takže budeme nakonec asi na 1 ušetřeném krátkém pulsu. Ale to je zisk 25 procent! Rozhodně nijak málo. Přirozeně, že při příliš častém výskytu osamocených 1 a 0 o něco klesne. Jindy ale zase může velmi výrazně stoupnout. A když si to dáme dohromady s předchozí textovou komprimací, která se pohybuje kolem 50 procent, rázem jsme na 75 procentech ušetřeného času, záznamového materiálu i poplatků za čas modemového spojení.

Ve spolupráci se sync pulsem můžeme komprimovat i jinými způsoby. Jedním z docela zajímavých je převedení každých 4 bitů každého bajtu na kód v rozsahu 0-15, tedy 1-4 bitů. Těm nejčastějším přiřadíme nejnižší kódy. Třeba sestavě 1101 1-bitový kód 0, 1011 kód 1, nakonec skupině 0000 přiřadíme 1110 a pro 1111 kód 1111. U nejnižších kódů tedy opět budeme zapisovat jen 1 bit místo čtyř atd. Pro "vylouhování" nižších a vyšších 4 bitů zdrojových bajtů použijeme např. jeden z těchto sledů instrukcí:

LD A, (HL)	XOR A
AND 0FH	RDD
CALL tabulka	CALL tabulka ; Nalezení kódu a
CALL pulsy	CALL pulsy ; odesl.jeho bitů
LD A, (HL)	RDD
AND 0FH	CALL tabulka
RRA	CALL pulsy
RRA	RDD ; (HL) do pův.stavu
RRA	
CALL tabulka	
CALL pulsy	

Skupinky kódových bitů na pásku opět oddělíme sync pulsem. Když budou sestavy bitů ve zdrojovém textu tomuto kódování příznivě nakloněny, může efektivita dosáhnout i více než 50 procent. Kombinace sestav čtyř bitů dáme do tabulky, která bude prohledávána a po nalezení bude sestavě přidělen patřičný kód, který se zapíše na pásek apod. Jde tedy o obdobu nám již známých tokens. Bylo by jistě možno uvažovat i o tabulce obsahující všech 256 sestav 8 bitů. Pro ten případ bychom museli zvolit takový způsob jejího prohledávání, který by nezdržoval, protože u rutin sériového přenosu je jejich časování velmi kritické.

Zajímavým rozdílem mezi oběma způsoby komprimace je, že u první klesá její efektivita směrem k nejčastějším výskytům bitových seskupení, u druhé je tomu opačně. Obě uvedené metody se opírají o generování oddělovacího sync pulsu, který je kratší než krátký puls pro log.0. "Krátkost" sync pulsu je limitujícím faktorem přenosové rychlosti, závislé na kvalitě a rychlosti pohybu magnetického materiálu vůči záznamové hlavě (resp. šířce přenosového pásma modemové líky).

5/ Komprimace obrazových dat

Specializovaná zařízení pro práci s počítačovou grafikou mívají adresování obrazové paměti různé komplikované. Obrazová paměť osmibitových počítačů je v drtivé většině případů rozdělena do dvou oddělených částí - zvlášť posloupný sled adres grafických bajtů a zvlášť barevných atributů. Proto nemusíme při komprimaci obrazových dat brát ohled na zvláštnosti v adresování kterékoli z obou částí paměti (samozřejmě pokud nechceme komprimovat jen její část). Další rutina komprimuje obrazová data "jedním tahem" od jejich první do poslední adresy. V grafické části je zvolen nejjednodušší postup "vymáčení" všech bajtů, jejichž osm bitů má shodnou hodnotu - bajty nulové (00000000B) nebo FFH (11111111B), které se v nekomplikovaných kresbách objevují nejčastěji. Ve výsledné sestavě komprimátů (čteno od jeho prvního bajtu) pak mohou nastat tyto varianty:

Když je obsah bajtu 0 nebo FFH, další dva bajty za ním obsahují informaci o jejich počtu v sledu za sebou.

Jakýkoli bajt, který nespadá do této trojice, je kopií obsahu bajtu, přečteného z obrazové paměti.

Při expanzi dat je bajty 0 nebo FFH obsazeno vždy tolik následných adres, kolik je jich uvedeno v páru informačních bajtů. Je-li další bajt komprimátu jiný než 0 nebo FFH, je jeho obsah umístěn přímo na nejbližší adresu paměti, atd. Když

bude výchozí obsah paměti složen jen z bajtů s jiným obsahem než 0 nebo FFH, bude komprimát přesnou kopií zdroje (ztráta rovna zisku, tedy kde nic tu nic). Budou-li se ve zdroji pravidelně střídát bajty 0 a FFH, výsledek bude nejhorší možný (třikrát delší). Proto je dobré mít o výsledku přehled a podle něj zvolit, zda komprimát použít či nikoli. Délka komprimátu je uložena ve dvou info-bajtech před komprimovaným blokem dat.

U atributů je postup komprimace jen málo odlišný. Každý bajt, tedy bez ohledu na to, zda obsahuje 0 či FFH, je následován dvěma info bajty o počtu shodných bajtů v jednom sledu. Nepříznivý výsledek dostaneme tehdy, když se hodnoty bajtů zdroje příliš často mění (mnoho "sledů" v délce 1). Rutina pro manipulaci s atributovými bajty může být obdobná konstrukci následující rutiny pro kompresi (vstup KOMPR) a expanzi (vstup EXP) obrazové grafiky.

Délka: 137 bajtů a 7 pro data

Vstup: viz datová oblast

Výstup: KOMPR: v HL adr.info-bajtu, v DE délka kompr.bloku dat,
v BC posl.adr.graf.obr.paměti

EXPAN: v HL " " kompr.bloku dat, BC=HL
v DE " " " "

Použité registry: IX, HL, DE, BC, AF

OBDRADR DEFW XXXX ;1.adr.graf.dat obraz.paměti
KOMPAD DEFW XXXX ;1.adr.uložení 2 info-bajtů komprim.dat
KONOBDR DEFW XXXX ;Posl.adr.graf.dat obraz.paměti
KOMEXP DEFB NN ;Indik.bajt pro rozlišení kompr./expanze

;Vstup pro komprimaci

KOMPR XOR A ;Indikace komprimace (A=0)
JR KOEX

;Vstup pro expanzi

EXP LD A,1 ;Indikace expanze (A=1)

KOEX LD (KOMEXP),A ;Uložení indik.bajtu do obl.dat
LD IX,KOMEXP ;Inicializace registrů
LD HL,(OBDRADR)
DEC HL
LD DE,(KOMPAD)
INC DE
AND A ;Jdeme na komprimaci?
JR NZ,COJEXP ;Když na expanzi, skok na COJEXP

COJET0 INC HL ;Jdi pro další obr.bajt
COJET1 INC DE ;a na další adr.komprim.dat
LD A,(HL) ;Obr.bajt přes reg.A
LD (DE),A ;přenes do kompr.dat

COJET2 CALL CITAC ;Jsme na konci obr.paměti?
JR Z,KONEC ;Když AND, směr KONEC
CP FFH ;Jinak porovnej přečtený bajt s FFH
JR Z,NULYFF ;Když je to FFH, jdi na adr.NULYFF
AND A ;Když to není FFH, je to nula?
JR Z,NULYFF ;Když je to nula, jdi na adr.NULYFF
BIT 0,(IX+0) ;Když to není nic z toho, pokračuj
JR Z,COJET0 ;na COJET0 nebo COJEXP

COJEXP INC HL ;Jdi na další obr.adresu
INC DE ;a další kompr.bajt
LD A,(DE) ;Kompr.bajt přes reg.A
LD (HL),A ;přenes do obr.paměti
JR COJET2 ;a pokračuj na adr.COJET2

CITAC PUSH HL ;Zjištění, zda už jsou všechny obraz.
LD B,H
LD C,L
LD HL,(KONOBDR)
AND A ;bajty porovnány (Když AND, tak po CALL
SBC HL,BC ;z části COJET0 Z=1, po CALL z části
POP HL ;KOLIK CY=1)
RET ;Návrat ze subrutiny

NULYFF BIT 0,(IX+0) ;Když probíhá expanze,
JR NZ,FFNULY ;skok na adr.FFNULY
PUSH DE ;Při kompr.uschovej moment.adr.kompr.dat
LD DE,0 ;Vynuluj čítač počtu stejných bajtů

KOLIK INC DE ;a zvětší jej o 1
INC HL ;Jdi na další adr.obr.bajtu
CALL CITAC ;Jsme za koncem obr.paměti?
JR C,UZNE ;Když AND, další bajty už neber
CP (HL) ;Jinak porovnej další obr.bajt s 0/FFH
JR Z,KOLIK ;Když je stejný, pokračuj na KOLIK


```

UZNE  AND  A           ;Vynulování CY (ještě nejsme za obr.p.)
      EX   (SP),HL     ;HL do zás., do HL moment.adr.kompr.dat
      INC  HL          ;Do HL ulož informaci o délce šňůry
      LD   (HL),E      ;stejných bajtů
      INC  HL
      LD   (HL),D
      EX   DE,HL
      POP  HL          ;Moment.adr.kompr.dat do DE
      JR   NC,COJET1   ;Do HL moment.adr.obr.paměti
                        ;Když nejsme za obr.paměti, pokračuj

KONEC BIT 0,(IX+0)    ;Jinak test (komprimace/expanze?)
      RET  NZ          ;Když probíhá expanze, zpět
      EX   DE,HL
      LD   DE,(KOMPAD) ;V závěru komprimace výpočet délky
      INC  DE          ;bloku kompr.dat s uložením obou info-
                        ;bajtů před kompr.data
      AND  A
      SBC  HL,DE
      EX   DE,HL
      LD   (HL),D
      DEC  HL
      LD   (HL),E
      RET                      ;Návrat z komprimace

FFNULY EX DE,HL       ;Do HL adresa kompr.bloku s informací o
      INC  HL          ;počtu stejných bajtů
      LD   C,(HL)      ;Do BC počet bajtů 0/FFH
      INC  HL
      LD   B,(HL)
      DEC  BC          ;Zmenšení čítače přenosů LDIR o 1
      LD   A,B         ;Jde jen o 1 bajt?
      OR   C
      JR   NZ,ROZVES   ;Když NE, jdi "rozvěsit" bajty v obr.p.
      EX   DE,HL       ;Když ANO, prohoď obě adresy v DE a HL
      JR   COJEXP      ;a pokračuj v expanzi

ROZVES PUSH HL        ;Uložení moment.adr.kompr.bloku dat
      LD   L,D
      LD   L,E
      INC  DE          ;Do DE adr.o 1 vyšší
      LDIR                ;Vyplň část paměti bajtem 0/FFH
      POP  DE          ;Obnov moment.adr.kompr.bloku dat v HL
      CALL CITAC       ;Je HL=BC?
      RET  Z           ;Když ANO, návrat z expanze
      JR   COJEXP      ;Když NE, pokračuj v expanzi

```

Díky tomu, že pracuje s celými bajty, je rutina rychlá. Expanze proběhne bleskově. Pro výukovou názornost je zaveden jeden programátorský hřích - umělý indikátor KONEC. Na jeho inicializaci a testy se spotřebuje 31 bajtů paměti! To je pětina rozsahu rutiny! V takovém momentu je dobré si k výtvaru sednout a upravit jej tak, aby indikátor zmizel. Budete-li mít čas a chuť, zkuste to sami. Protože na výstupu z rutin máme vždy poslední adresu komprimovaného bloku dat, můžeme provést postupnou komprimaci a expanzi více obrázků, jejichž komprimáty budou v paměti seřazeny za sebou.

Po této úvodní hříčce se teď podíváme na o něco zajímavější způsob komprimace obrazových dat. Nebudou načítány sledy stejných bajtů, ale bitů, a to všech. Formát komprimátu je založen na představených bitech k číslu, vyjadřujícímu počet stejných bitů. Představené bity informují o bitové délce za ním umístěného čísla. Pro tuto manipulaci se musíme zbavit představ o paměti "dělené na bajty". Na obsah komprimované paměti budeme pohlížet jako na jednorozměrné bitové (neuspořádané) pole. Co prvek pole, to jeden bit. Do tohoto pole zavedeme zákonitost danou formátem komprimace:

	Sled stej- ných bitů	Komprimát	Zisk počet bitů
1	0 (nebo 1)	01	-1
2	00	0010	-2
3	000	0011	-1
4	0000	10100	-1
5	00000	10101	0
6	000000	10110	1
7	0000000	10111	2
8	00000000	1001000	1
9	000000000	1001001	2
10	0000000000	1001010	3
11	00000000000	1001011	4
12	000000000000	1001100	5
13	0000000000000	1001101	6
14	00000000000000	1001110	7
15	000000000000000	1001111	8
16	0000000000000000	1101000	8


```

256 ..... 111101000000000 242
512 11110010000000000 496
1024 11111010000000000 1007

```

U komprimátu si povšimněte, že hranicí mezi info-bity a číslem je vždy první předěl 01 zleva. Vlevo před touto jedničkou je informace o počtu míst čísla, začínajícího touto jedničkou. Každá jednička před hraniční nulou reprezentuje dvě místa, nula (v počtu jedné nebo dvou) jedno. Pro orientaci v této sestavě jednička na předělu být musí, stejně tak nula před ní. V případě sudého počtu míst čísla jsou před hraniční jedničkou dvě nuly, při lichém počtu jedna.

Z tabulky je patrné, že do sledu 4 stejných bitů je komprimace ztrátová. Při pěti shodných bitech jsme "na svém". Teprve výš to začíná být zajímavé, ba čím dál zajímavější. Při sledu 15 a 16 bitů se efektivita komprimace přehupuje přes 50 procent a pak už jde tvrdě nahoru, prakticky až k nějakým 99 procentům. Ztrátovost této komprimace na začátku tabulky ji předurčuje pouze pro některé aplikace. Jednou z nich je právě počítačová grafika. Málodky na mikropočítači najdeme tak "protečkovanou" kresbu, aby na ni tato komprimace pohořela. Zvláště pro technické výkresy je jednou z nejlepších. Přednost této komprimace oproti minulé je v tom, že ji nevyvádí z rovnováhy, když dlouhá série nultých bitů končí "uvnitř" obrazového bajtu a v něm navazuje další dlouhá série bitů s log.1 (a naopak). Pro předchozí komprimaci to znamenalo ukončit načítání jedné série, okopírovat "přerušeny" i následující bajt, a založit ještě dva info-bajty pro načítání další série. Protože si následující rutina hraje s jednotlivými bity, je pomalejší, než její "celobajtová kolegyně". Rutina má opět dva různé vstupy - KOMPR a EXPAN. Před blokem komprimovaných dat jsou 4 informační bajty. V prvních dvou je uložena bajtová délka komprimovaného bloku, v dalším (jeho bitu 0) informace o tom, zda obrazová data začínají log.0 nebo log.1, a v posledním z této čtveřice je uložen poslední stav čítače bitů posledního komprimovaného bajtu.

Délka: 424 bajtů a 15 pro data

Vstup: viz data

Výstup: KOMPR - v HL adr.1.info-bajtu o délce bloku kompr.dat

v DE délka bloku kompr.dat

EXPAN - v HL při obsahu reg.C různém od 8 posl. adr. bloku kompr.dat, při C=8 adresa o 1 vyšší

Použité registry: IX, HL, DE, BC, AF

```

OBRADR DEFW XXXX ;1.adresa graf.obraz.paměti
KOMPAD DEFW XXXX ;1.adr.kompr.bloku dat; za 4 info-bajty!
KOMZAC DEFW KOMPAD-4 ;Adr.1.info-bajtu celého bloku
DELOBR DEFW XXXX ;Délka graf.obraz.paměti (v bajtech)
DELKA DEFS 2 ;Zmenšující se délka zpracováváných dat
ROT DEFS 2 ;2-bajtové pole pro narotování čísla
KONBIT DEFS 1 ;Posl.stav čítače bitů posl.bajtu k.dat
KOMLOG DEFS 1 ;Průběžná uschova "vnitřního" bitu roto-
vaného bajtu (do bitu 7, (KOMLOG))
LOGLOG DEFS 1 ;Signalizační přepínač log.hodnoty uklá-
dané šňůry bitů téže hodnoty (v bitu 0)

```

;Vstup pro komprimaci

```

KOMPR LD HL,(DELOBR) ;Vstupní inicializace registrů a dat
LD HL,(DELKA),HL
LD HL,(KOMPAD)
PUSH HL
POP IX
LD HL,(OBRADR)
LD A,(HL) ;Log.hodnota 1.bitu obr.paměti
RLA
LD (IX-2),A ;do 3.info-bajtu před blokem kompr.dat
XOR A ;A=0 pro signalizaci komprimace (při ex-
LD (KONBIT),A ;panzi je (KONBIT) jen v intervalu 1..8)
BIT 7,(HL) ;Zjištění log.stavu 1.bitu obr.paměti
LD DE,0 ;Vynulování čítače stejných bitů
LD BC,#0808 ;C je čítač bitů obr.bajtů, B bitů bu-
JR NZ,START3 ;doucích kompr.bajtů; Když log.1, skok

```

```

START2 CALL LOG0 ;Počítání série bitů log.0
CALL MISTA ;Vytvoření představených 1 a 0
START3 CALL LOG1 ;Počítání série bitů log.1
CALL MISTA
JR START2 ;Po sérii log.1 zase na log.0

```

LOG0	RL JR INC DEC CALL JR	(HL) C,ZMENA DE C Z,DALSI LOG0	; Levá rotace bitu do CY ; Když už to není log.0, změna, skok ; Když AND, zvýš čítač bitů log.0 o 1 ; Všechny 8 bitů bajtu vyhodnoceno? ; Když AND, skok na adr.DALSI ; Pokračuj ve "sběru" bitů log.0
LOG1	RL JR INC DEC CALL JR	(HL) NC,ZMENA DE C Z,DALSI LOG1	; Totéž analogicky pro bity log.1
ZMENA	RR RRA LD RET	(HL) (KOMLOG),A 	; "Vrotuj" bit změny zpět; touto operací ; zprava "vypadlý" bit nasuň do bitu 7,A ; a uschovej jej na adr.KOMLOG ; a vrať se ze subrutiny
DALSI	PUSH RL LD DEC LD AND JR LD OR LD POP JR INC LD RET	HL (HL) HL,(DELKA) HL A,(KONBIT) A NZ,EXDAL A,H L (DELKA),HL HL Z,KONEC HL C,B 	; Všechny 8 bitů bajtu vyhodnoceno, proto ; proved ještě 1 rotaci pro uvedení bajtu ; do původního stavu a zmenš čítač počtu ; bajtů ke zpracování o 1 ; Probíhá komprimace nebo expanze? ; Když expanze (Z=0), skok ; Při komprimaci zjisti, jestli už zpra- ; covány všechny obraz.bajty ; Čítač délky bloku dat do proměnných dat ; Obnovení obsahu reg.HL ; Když při komprimaci Z=1, směr konec ; Na další adresu bloku dat ; Opětná inicializace čítače bitů na 8 ; Návrat ze subrutiny
MISTA	PUSH PUSH LD RL RL DEC JR RR RR PUSH POP PUSH XOR DEC RR JR DEC DEC DEC DEC MISTA2 DEC MISTA4 DEC JP SCF	HL BC C,17 E D C NC,MISTA1 D E IX HL BC A A C C,MISTA2 C A C A M,MISTA3 	; Uložení adr.zdroj.bloku dat ; Uložení moment.stavu obou bit.čítačů ; Nyní je C čítač bit.délky čísla v DE ; Vyrotování zbytečných nul zleva ; Snížení čítače o 1 při každé nule a ; první jedničky, ; kterou je nutno vrátit zpět do DE ; Moment.adr.kompr.bloku dat ; do HL ; Uložení délky čísla (v reg.C) ; Vynulování reg.A (čítač hraničních nul) ; a snížení na FFH ; Vydělení délky dvěma a test, zda je ; délka lichá (CY=1) či sudá (CY=0) ; Když je sudá, hranič.nuly budou dvě, ; proto zmenš délku o 1 čítač nul o 1 ; Pro sudou délku sniž A ještě jednou ; C slouží jako čítač smyčky ; Když nabyde záporné hodnoty, skok ; Jinak rotuj budoucí představené ; jedničky do (HL) zleva ; Zmenš čítač bitů kompr.bajtu o 1 ; Když není "plný", skok ; Jinak nastav čítač bitů opět na 8 ; a jdi na další adr.kompr.bloku dat ; Je obsah čítače představ.nul nulový? ; Když NE, pokračuj v ukládání jedniček ; Zvyš čítač představ.nul o 1 ; Už je nulový? Když ještě NE, jdi uložit ; ještě 1 (druhou) předst.(hraniční) nulu
MISTA1	RL RL DEC JR RR RR PUSH POP PUSH XOR DEC RR JR DEC DEC DEC DEC MISTA2 DEC MISTA4 DEC JP SCF	D D C NC,MISTA1 D E IX HL BC A A C C,MISTA2 C A C A M,MISTA3 	
MISTA2	DEC	A	
MISTA4	DEC	A	
MISTA5	RL DEC JR LD INC AND JR INC AND JR	(HL) B NZ,MISTA4 B,B HL A NZ,MISTA4 A A NZ,MISTA5	
MISTA3	INC AND JR	A A NZ,MISTA5	
CISLO1	LD POP RL RL DEC JR LD INC DEC JR PUSH POP LD POP LD POP LD RLA RET	A,B BC E D (HL) A NZ,CISLO2 A,B HL C NZ,CISLO1 IX DE,0 BC B,A HL A,(KOMLOG) 	; Do A ulož stav bit.čítače kompr.bajtu ; V C obnov délku čísla počtu stejn.bitů ; Číslo z DE narotuj ; do bajtu (bajtů) kompr.dat ; Je moment.bajt kompr.dat už "plný"? ; Když NE, přeskoč další dvě instrukce ; Když AND, bit.čítač opět na 8 ; a na další adr.v bloku kompr.dat ; Zmenšení čítače délky čísla o 1 ; Když číslo ještě nedorotováno, pokračuj ; Když hotovo, převed obsah HL ; do IX (moment.adr.kompr.bloku) ; Vynuluj čítač série stejných bitů ; Obnov stav bit.čítače obr.bajtu v C ; a do B ulož moment.stav bit.čítače kom- ; prim.bajtu; Do HL adr.obr.paměti ; Do bitu 7,A stav posl."vnitřního" bitu ; a jeho přesun do CY pro další operaci ; Návrat ze subrutiny
CISLO2	DEC JR PUSH POP LD POP LD POP LD RLA RET	C NZ,CISLO1 IX DE,0 BC B,A HL A,(KOMLOG) 	

```

KONEC CALL MISTA ;Jsme na posl.obr.adr.,dokonči přesuny
      POP DE ;V zásobníku se dober k adrese návratu
      POP DE
      LD A,B ;Posl.stav bit.čítače kompr.bajtu do A
      LD DE,(KOMPAD) ;a jeho uložení
      DEC DE
      LD (DE),A ;na 3.pozici v info-bajtech bloku
      CP 8 ;Je stav čítače roven 8?
      JR NZ,KONEC1 ;Když NE, skok
      DEC IX ;Když ANO, jsme o 1 adr.dál, proto zpět,
      JR KONEC2 ;kompr.bajt "plný", ničeho netřeba, skok
      AND A ;Když NE, musíme "přisunout" narotované
      RL (IX+0) ;bity k levému kraji kompr.bajtu
      DJNZ KONEC1 ;Dokud bity "nepřisunuty", rotuj
      KONEC2 PUSH IX ;Posl.adr.kompr.bloku z IX
      POP HL ;do HL
      XOR A ;Vynulování CY pro odečet bajtové
      SBC HL,DE ;délky kompr.bloku
      EX DE,HL ;Výsledek ulož
      DEC HL
      DEC HL
      LD (HL),D ;prvních dvou info-bajtů před blok
      DEC HL ;komprimovaných dat
      LD (HL),E
      RET ;Konečný návrat z komprimace

;Vstup pro expanzi
EXPAN LD HL,(OBRADR) ;Vstupní inicializace registrů
      PUSH HL
      POP IX
      LD HL,(KOMZAC)
      LD E,(HL) ;Do DE délka kompr.bloku
      INC HL
      LD D,(HL)
      LD (DELKA),DE ;a její uložení do proměnné
      INC HL
      LD A,(HL)
      LD (LOGLOG),A ;Bit 0,A obsahuje log.úroveň prvního
      INC HL ;obraz.bitu; jeho přenos do přepínače
      LD A,(HL)
      LD (KONBIT),A ;Ulož posl.stav bit.čítače posl.bajtu
      INC HL ;Na 1.adr.kompr.bloku dat
      LD BC,#0808 ;C je bit.čítač kompr.bajtu, B obr.bajtu
      LD DE,0 ;Vynulování čítače stejných bitů
      EXPANX LD A,(KOMLOG) ;"Vnitřní" bit do A
      RLA ;a z bitu 7,A do CY
      BIT 7,(HL) ;Jaká je log.hodnota moment.kompr.bitu?
      JR NZ,EXPAN1 ;Když je to log.1, skok

EXPAN0 CALL LOG0 ;Jdi počítat sérii bitů log.0
      JR EXMIST ;Zjištění představených bitů a čísla
EXPAN1 CALL LOG1 ;Jdi počítat sérii bitů log.1

EXMIST BIT 7,(HL) ;Jaký je další bit v kompr.bloku?
      JR NZ,POCETB ;Když log.1, je to 1.bit čísla
      PUSH DE ;Jinak je to hraniční nula (nebo dvě)
      LD DE,0 ;Po úschově představ.jedniček vynuluj DE
      LD A,(KOMLOG) ;coby čítač hranič.nul; "vnitřní" bit
      RLA ;rotovaného bajtu do CY
      CALL LOG0 ;Spočítej nuly (jedna nebo dvě)
      LD A,E ;Počet nul z E do A
      POP DE ;Obnov počet jedniček v DE (nemusí tam
      SLA E ;být žádná!); vynásob obsah DE dvěma
      RL D
      ADD A,E ;a přičti nižší baj k počtu nul
      JR NZ,EXMIS1 ;Když přenos,
      INC D ;zvyš obsah vyššího bajtu o 1
      EXMIS1 LD E,A ;Hotový nižší bajt z E do A; v DE je teď
      POCETB PUSH IX ;počet míst čísla za hraniční nulou
      PUSH BC ;Úschovej moment.adr.kompr.bajtu
      LD IX,ROT ;a stav obou čítačů
      LD (IX+0),0 ;Do IX adr.16-bit.pole pro narotování
      LD (IX+1),0 ;čísla; napřed pole vynuluj
      LD A,(KOMLOG) ;"Vnitřní" bit
      POCET1 RLA ;do CY (přijde do bitu 0,(HL))
      RL (HL) ;Vyrotuj další bit čísla do CY
      SBC A,A ;Při CY=1 A=FFH a CY=0 A=CY=0
      RL (IX+0) ;Bit čísla narotuj do pole ROT
      RL (IX+1)
      LD B,A ;Obsah A zapiš do B

```

```

RLA                ;Posledně rotovaný bit do CY
DEC C              ;Zmenšení bit.čítače kompr.bajtu o 1
CALL Z,DALSI      ;Když je bajt "plný", jdi na další
DEC DE            ;Zmenšení počtu míst čísla o 1
LD A,D            ;Už celé číslo přesunuto do pole ROT?
OR E
LD A,B            ;B s bity o hodn.posl."vnitř." bitu do A
JR NZ,POCET1      ;Když číslo ještě nepřesunuto, pokračuj

LD (KOMLOG),A     ;Číslo hotovo, "vnitřní" bit do obl.dat
LD A,C            ;Stav bit.čítače kompr.bajtu do A
POP BC            ;Obnov stav bit.čítače obr.bajtu v B
LD C,A            ;Bit.čítač kompr.bajtu z A do C
POP DE            ;Moment.obr.adr.do DE
PUSH HL           ;Uložení moment.adr.kompr.bloku
PUSH BC           ;a bit.čítače kompr.bajtu
LD HL,LOGLOG      ;Přepnutí posl.log.hodnoty bitů minulé
INC (HL)          ;série na opačnou hodnotu (opačný vztah)
BIT 0,(HL)        ;Jaká bude nová log.hodnota bitů?
EX DE,HL          ;Moment.obr.adr.do HL
LD DE,(ROT)       ;Do DE adr.pole ROT
JR NZ,BITYO       ;Když to budou bity log.0, skok

BITY1 SCF         ;Umístění série bitů log.1 do obr.paměti
RL (HL)           ;Bit log.1 zleva do bajtu obr.paměti
CALL OBRBIT       ;Volej subrutinu pro testy a ev.inical.
JR NZ,BITY1       ;Když série ještě nehotova, pokračuj
JR BITOK          ;Série hotova, skok na BITOK

BITY0 SLA (HL)     ;Umístění série bitů log.0 do obr.paměti
CALL OBRBIT       ;a dále totéž jako u BITY1
JR NZ,BITYO

BITOK LD A,B       ;Série hotova, ulož stav bit.čítače obr.
POP BC            ;bajtu přes A do B a obnov stav bit.čí-
LD B,A            ;tače kompr.bajtu v C
EX (SP),HL        ;Moment.adr.kompr.bloku do HL a obr.adr.
POP IX            ;z HL do zásobníku a do IX
PUSH HL           ;Uchovej obsah HL
LD HL,(DELKA)     ;Zjisti, zda jsme na posl.obr.adrese
SCF               ;(pak Z=1, C v interv. 1..8), nebo za ní
SBC HL,DE         ;(pak CY=1 a C=8)
POP HL
RET C             ;Když jsme za ní, návrat z expanze
JP NZ,EXPANX      ;Když ani za, ani na ní, expanduj dál
LD A,(KOMBIT)     ;Jsme na ní, zjisti, zda zpracovány
CP C              ;všechny bity posl.kompr.bajtu
JP NZ,EXPANX      ;Když ještě NE, dokonči expanzi
LD A,(KOMLOG)     ;Když AND, zbývá "přirazit" je k levému
RLA               ;krají kompr.bajtu; dej "vnitřní" bit
LD B,C            ;do CY, bit.čítač kompr.bajtu do B, zvyš
INC B             ;čítač o 1 (CY je "9.bit")
EXKONI RL (HL)    ;a proved "přirazení" platných bitů
DJNZ EXKONI
RET               ;Návrat z expanze

OBRBIT DEC B       ;Zmenši bit.čítač obr.bajtu o 1
JR NZ,OBRBIT1     ;Je bajt "plný"? Když NE, skok
INC HL            ;Když AND, jdi na další obr.adresu
LD B,B            ;a nastav bit.čítač opět na 8
OBRBIT1 DEC DE     ;Zmenši počet bitů série o 1
LD A,D            ;Test, zda série hotova
OR E              ;Když AND, návrat ze subrutiny
RET               ;se Z=1, když ještě NE, se Z=0
    
```

Jedním z cílů konstrukce rutiny bylo výhradní použití bitových manipulací. Každý jednotlivý bit zdrojového i cílového pole je rotován či posouván. Tak je rutina velmi příhodná pro procvičení bitových rotací. Velkou pozornost při konstrukci takové rutiny či jejích dalších úpravách je nutno věnovat bitům, v komentáři označovaným jako "vnitřní". Jsou to bity, které byly zleva či zprava vyrotovány z bajtu do CY, a jimž hrozí "ztráta původnosti" v průběhu dalších operací. Je nezbytné jejich log. stav uchovat, aby při jejich příštím nasunutí do bajtu nedošlo ke změně jeho obsahu. Díky všem použitým ochranným opatřením je komprimace i expanze nedestruktivní, tj. ponechává zdrojové pole v původním stavu.

Zatímco předchozí rutina byla na světě za necelé dvě hodiny, tato velmi srdnatě vzdorovala. Před usednutím k počítači mi bylo všechno "jasné". S přibývajícimi hodinami se zatemňovalo. Vyskakovaly problémy, o kterých jsem předtím neměl ani potuchy. Odbytá předběžná analýza se vymstila. Přesto jsem pokračoval ve stylu "hic sunt leones", krok za krokem. Teprve třetí den, patřičně s(eš)rotován, jsem mohl zvolat Evoé! To je důkazem

toho, že předběžnému rozboru je dobré věnovat maximální péči. Ušetří se jím spousta hodin mučivých rekonstrukcí programu. Otázkou ovšem zůstává, nakolik je kdo schopen odhalit maximum zálužností řešeného úkolu již v této předběžné fázi.

Velmi zvědav jsem byl na časové proporce "bitové" rutiny. Kupodivu nejsou nijak špatné. U obrazové paměti Spectra (49152 bitů) každá z obou základních funkcí netrvá déle než dvě vteřiny. Příjemné překvapení mi připravila výsledná délka komprimátů. Pro porovnání s předchozí rutinou jsem zkomprimoval 21 obrázků z přílohy minulé učebnice. Celobajtová komprimace přinesla celkový výsledek v délce cca 64K. Bitová cca 34K. To vše z původních 126K! Komprimaci jsem vyzkoušel i na několika titulních obrazovkách her. Zatímco výsledek celobajtové komprimace byl občas delší než originál, bitové komprimaci se to nestalo ani jednou.

Dobu exekuce rutiny by bylo možno zmenšit celobajtovými přenosy tam, kde je v sérii stejných bitů více bajtů 0 či FFH za sebou. Pokud má váš počítač delší obrazovou paměť než 65535 bitů, budete muset rozšířit 16-bitové pole pro tvorbu informací o počtu míst čísla i číslo samotné na vícebitové. Např. Amstrad PCW 8512 apod. má na jedné grafické lince $80 \times 8 = 640$ bitů a $25 \times 8 = 200$ linek, to je celkem 128000 bitů. Pro takovou obrazovou paměť bude stačit i bit navíc (může to být třeba bit na nějaké adrese v oblasti programových dat). Manipulace s bitem navíc bude pochopitelně vyžadovat i adekvátní úpravu rutiny.

Jedny ze základních sentencí programování říkají, že každý program lze zkrátit a že neexistuje program, v němž by nebyla nějaká chybička. Tak - ad humorné absurdum - lze každý program zkrátit na jedinou instrukci, která bude špatně. I většinu zde uvedených rutin by bylo možno zkrátit o nějaký ten bajtík, případně jít na věc jinak - lépe a efektivněji. Na tomto místě mám k vám jednu výzvu. Podaří-li se vám vytvořit nějakou univerzální rutinu, zpracovávající jednu základní myšlenku neotřele, prostě zajímavě, pošlete ji na mou adresu. Udělám vše pro to, aby se (samozřejmě vaším jménem) objevila tiskem v nějakém adekvátním periodiku. Vaše důvtipné rutiny budu předběžně konzultovat s assemblerovými kapacitami a pokud výsledky konzultací povedou k dalším vylepšením, budu vás o nich informovat. Pro jistotu opakuji, že rutiny musejí být původní a univerzální, tj. nevázané na zvláštnosti systému jedné značky počítače. Otevřeme společně mnohotvárné pole assemblerové magie i ku prospěchu všech budoucích kouzelníků. Těším se na vaše příspěvky.

KAPITOLA 7

Pole

Na úvod bychom si mohli ujasnit rozdíl mezi poli a řetězcí. Řetězec je množina posloupných prvků stejného typu, nejčastěji umístěných na těsně sousedících adresách. Typem prvku může být znak, bit apod. Řetězec, v němž není žádný prvek, se označuje jako prázdný, jeho délka je nulová.

Pole je N-rozměrná soustava X prvků (elementů). Typ prvků je buď shodný pro celé pole, nebo může být určitým způsobem diferencován. Např. řádky 2-rozměrného pole mohou být různého typu. Prvkem pole, jako položkou zpracovávaných dat, může být řetězec. Vzáto do důsledků, samotný řetězec lze brát jako 1-rozměrné pole, v němž je uloženo X prvků téhož typu. Podobně jako řetězec, může být i pole prázdné. Splnění typových podmínek je předpokladem pružného zpracování prvků pole. Pro orientaci v poli jsou prvky značeny indexy. Pro orientaci v jejich uložení v paměti jim někdy bývají přiděleny klíče, jimiž se určuje jejich vzájemný vztah, ev. další relace. N-rozměrná pole se do paměti nejčastěji ukládají jako posloupnost (N-1)rozměrných polí.

Práce s poli s sebou nese některé typické elementární operace obecného charakteru. Např. vyplnění úseku paměti bajty jedné hodnoty, blokové přenosy atd. Rutiny těchto operací najdete v předposlední kapitole.

1/ Transpozice matic

Elementy dvourozměrného pole můžeme ukládat do matic sekvenčně, a to buď po řádkách nebo sloupcích. Tak např. pole:

A	B	C
D	E	F
G	H	I

můžeme zvažovat jako sekvenci ABCDEFGHI nebo ADGBEHCFI (v tom sledu je rovněž uložena na adresách paměti). U malého pole si s jeho prohledáváním nemusíme dělat zvláštní starosti. Ale u velkého už záleží na tom, jak se k požadovanému elementu dostaneme. V naší malé matici se třeba k elementu H dostaneme po řádkách výpočtem $2*3+2$, nebo po sloupcích $1*3+3$. Jak víte, je násobení u Z80 operací, jejíž doba provedení roste s počtem provedených součtů ve smyčce. Už z této malé matice je patrné, že doba provedení operace $3*3+2$ bude delší než $3*3$. Proto je někdy vhodné matici převést z jedné sekvence do druhé. Pochopitelně, že takovou potřebu můžeme obejít i programově. Přesto - zvláště v průběhu tvorby vlastních konstrukcí, nebo při převodu polí z jedné programů do druhých - může transpozice dvourozměrného pole přijít vhod. Své využití najde především v databázových, kalkulačních programech a při práci s grafikou.

Následující dvě rutiny jsou rozdílné uložením cílové matice. První rutina ukládá výsledek transpozice na jiné místo v paměti, než kde leží původní matice. Druhá provádí transpozici na místě matice původní. Tím je dána i různá doba jejich provedení - ve druhém případě je mnohem delší. Obě rutiny převádějí ze sekvence řádkové do sloupcové a pracují s jednotatovými elementy.

a/ transpozice matice na nové místo v paměti

Algoritmus:

```
Pro daný počet sloupců opakuj:
  Uchovej adr.ukazatele zdroje
  Pro daný počet řádek opakuj:
    přenes element ze zdroje do cíle
    zvětš cílový ukazatel o 1
    zvětš zdrojový ukazatel o počet sloupců
  Obnov zdrojový ukazatel
  Zvětš zdrojový ukazatel o 1
```

Délka: 35 bajtů

Počet cyklů: (počet řádek*60+67)*počet sloupců+129

Vstup: v HL adr.1.bajtu zdrojové matice, v IX cílové matice
v B čítač řádek (1..255), v C sloupců (1..255)

Výstup: Všechny registry nezměněny

Použité registry: IX,HL,BC


```

MATRAN PUSH AF          ;Uložení registrů do zásobníku
        PUSH HL
        PUSH IX
        PUSH DE
        PUSH BC          ;Čítač řádek v B je na spodku zásobníku
        LD E,C           ;Do DE čítač sloupců
        LD D,0

COLLP  PUSH HL          ;Uchova adr.1.sloupce

ROWLP  LD A,(HL)         ;Do reg.A element ze zdroje
        LD (IX+0),A      ;a jeho přenos do cíle
        ADD HL,DE        ;Výpočet adresy další řádky téhož sloup.
        INC IX           ;Nastav IX na adresu dalšího cíl.sloupce
        DJNZ ROWLP      ;Opakuj pro všechny řádky daného sloupce

        POP HL           ;Jdi na adresu dalšího sloupce zdroje
        INC HL
        LD A,C           ;Uchova čítače sloupců
        POP BC           ;Obnova čítače řádek v reg.B
        PUSH BC          ;a jeho opětná úschova
        LD C,A           ;Do C moment.stav čítače sloupců
        DEC C            ;a jeho zmenšení o 1
        JR NZ,COLLP     ;Opakuj pro všechny sloupce

        POP BC           ;Obnova všech registrů
        POP DE
        POP IX
        POP HL
        POP AF
        RET              ;Návrat

```

b/ transpozice matice na totéž místě uložení zdrojové matice

Tato transpozice je vhodná z hlediska šetření paměti, a jak to tak už bývá, zisk místa přináší ztrátu času, přestože rutina používá instrukci LDIR. Každý přemísťovaný element je uložen na konec matice, zatímco všechny ostatní jsou přemístěny o 1 místo "dolů". Postup demonstruje tabulka provedení jednotlivých kroků smyčky u transpozice takovéto matice:

		1	2	3	4
	a	.	.	.	.
	b	.	.	.	.
	c	.	.	.	.
řád.	sl.				
4	3	(a1 a2 a3 a4 b1 b2 b3 b4 c1 c2 c3 c4)			
4	2	a2 a3 a4 (b1 b2 b3 b4 c1 c2 c3 c4 a1)			
4	1	a2 a3 a4 b2 b3 b4 (c1 c2 c3 c4 a1 b1)			
3	3	(a2 a3 a4 b2 b3 b4 c2 c3 c4 a1 b1 c1)			
3	2	a3 a4 (b2 b3 b4 c2 c3 c4 a1 b1 c1 a2)			
3	1	a3 a4 b3 b4 (c2 c3 c4 a1 b1 c1 a2 b2)			
2	3	(a3 a4 b3 b4 c3 c4 a1 b1 c1 a2 b2 c2)			
2	2	a4 (b3 b4 c3 c4 a1 b1 c1 a2 b2 c2 a3)			
2	1	a4 b4 (c3 c4 a1 b1 c1 a2 b2 c2 a3 b3)			
1	3	(a4 b4 c4 a1 b1 c1 a2 b2 c2 a3 b3 c3)			
1	2	(b4 c4 a1 b1 c1 a2 b2 c2 a3 b3 c3 a4)			
1	1	(c4 a1 b1 c1 a2 b2 c2 a3 b3 c3 a4 b4)			
0		a1 b1 c1 a2 b2 c2 a3 b3 c3 a4 b4 c4			

Řádek 0 ukazuje výslednou transpozici. Elementy v závorce jsou "rotovány". Element vpravo vedle kulaté závorky je vždy přenesen na konec matice, zbývající posunuty o 1 místo doleva. Uvedený stav na jednotlivých řádkách tabulky (kromě ř.0) je vždy před vstupem do smyčky NXTCOL.

Algoritmus:

```

Pro daný počet sloupců opakuj:
  Uchovej adr.ukazatele zdroje
Pro daný počet řádek opakuj:
  Rotuj element zdroje na konec matice
  zvětš zdrojový ukazatel o počet sloupců-1
  Obnov zdrojový ukazatel

```

Délka: 35 bajtů

Vstup: v DE adr.1.bajtu matice, v HL adr.posl.bajtu matice
v B počet řádek (1..256, tj. 01H..00H)
v C " sloupců (1..256, tj. 01H..00H)

Výstup: DE,HL,B beze změny; C=0; v A poslední bajt matice
Použité registry HL,DE,BC,AF

;Toto není startovací adresa rutiny; startuje se od adr.OSMTRN

```

NXTROW LD  A,C           ;Výpočet adresy bajtu pro rotaci souč-
      DEC  A             ;tem obsahu čítače sloupců o 1 zmenše-
      ADD  A,E           ;ného a adresy v DE
      LD   E,A
      LD   A,0
      ADC  A,D           ;Při CY=1 zvětšení obsahu D o 1
      LD   D,A

NXTCOL LD  A,(DE)        ;Rotovaný bajt do reg.A
      PUSH DE            ;Uložení ukazatele adresy
      PUSH BC            ;a čítač řádek a sloupců
      AND  A             ;Vynulování CY
      SBC  HL,DE         ;Výpočet počtu bajtů pro jejich posun
      LD   B,H           ;paměti dolů, výsledek do BC jakožto
      LD   C,L           ;čítače instrukce LDIR
      LD   H,D           ;V DE je 1.adr.přenosu (zdroje)
      LD   L,E
      INC  HL            ;HL=DE+1
      LDIR              ;Provedení přenosu určených bajtů
      DEC  HL            ;V HL adresa konce matice
      LD   (HL),A        ;Uložení rotovaného bajtu na konec
      POP  BC            ;Obnovení obsahu čítačů
      POP  DE            ;a adr.zdroje
      DJNZ NXTROW        ;Opakuj pro všechny řádky ve sloupci

      POP  BC            ;Obnova čítačů řádek a sloupců
      POP  DE            ;a 1.ar.matice
      DEC  C             ;Zmenši čítač sloupců
      RET  Z             ;Všechny sloupce hotovy? Když ANO,návrat
                        ;Když NE, pokračuj na dál na OSMTRN
OSMTRN PUSH DE          ;Uložení registrů do zásobníku
      PUSH BC
      JR   NXTCOL        ;Skok na zpracování jednotlivých sloupců

```

2/ Vyhledání adresy 1-bajtového elementu matice

Matice je dvourozměrným polem, v našem případě řádkově orientovaným. Počet sloupců je zároveň délkou řádky. Element je určen indexy I, J, tedy jako průsečík řádky I a sloupce J. První element matice má indexy 0,0. Dále je zadána adresa prvního elementu a délka řádky. Algoritmus vyhledání elementu I,J: $\text{adr.1.elementu} + I * \text{délka řádky matice} + J$. Rutina bohatě využívá zásobník, v němž jsou uloženy všechny vstupní informace.

Délka: 44 bajtů a 4 pro data

Počet cyklů: průměrně 1100

Vstup: od spodní adresy zásobníku:

návratová adresa
2 bajty čísla J (sloupec)
2 bajty délky řádky matice
2 bajty čísla I (řádka)
1.adr.matice

Výstup: v HL adr.hledaného elementu

Použité registry: HL,DE,BC,AF

```

D2BYTE POP  HL          ;Uložení návratové adresy
      LD   (RETADR),HL  ;do obl.dat
      POP  HL           ;Uložení sloupce J
      LD   (SS2),HL     ;do obl.dat
      POP  DE           ;Do DE délka řádky matice
      POP  BC           ;Do BC řádka I
      LD   HL,0         ;Příprava pro operaci I*délka řádky
      LD   A,15         ;pomocí algoritmu posuň-a-přičti

MLP    SLA  E           ;Posun násobitele DE vlevo (*2)
      RL   D
      JR   NC,MLP1      ;Když po posunu CY=0, vynech přičtení BC
      ADD  HL,BC         ;Jinak přičti BC (násobence) k meziproduct.
MLP1   ADD  HL,HL        ;Posuň meziproduct vlevo
      DEC  A             ;Zmenši bit.čítač o 1
      JR   NZ,MLP       ;Opakuj pro zbývajících 15 bitů
      OR   D             ;Indik.S=nejvyš.bit násobitele
      JP   P,MLP2        ;Když výsledek kladný, skok na MLP2
      ADD  HL,BC         ;Ještě jedno přičtení, když S=1
MLP2   LD   DE,(SS2)    ;Číslo J (sloupec)
      ADD  HL,DE         ;přičti k meziproductu (kon.poz.v mat.)
      POP  DE           ;Do DE adr.1.elementu matice
      ADD  HL,DE         ;Přičti k ní pozici elementu v matici
      LD   DE,(RETADR)  ;Do DE návratová adresa
      PUSH DE           ;Ulož ji do zásobníku
      RET              ;a vrať se na ni

RETADR DEFS 2           ;Uložení návrat.adresy
SS2    DEFS 2           ;Uložení čísla J (sloupce)

```

3/ Vyhledání adresy 2-bajtového elementu matice

Předchozí rutinu můžete velmi snadno upravit pro práci se 2-bajtovými elementy dvourozměrného pole. Stačí jediný zásah - mezi 3. a 4. instrukcí (mezi POP HL a LD (SS2), HL) vložte ADD HL, HL pro vynásobení čísla J (sloupce) dvěma. Nezapomeňte, že bajtová délka řádky ve "vstupním" zásobníku musí být dvojnásobkem počtu elementů na řádce. Analogicky lze rutinu upravit pro libovolnou délku elementu.

4/ Vyhledání adresy elementu v N-rozměrném poli

V další rutině musejí mít všechny elementy všech rozměrů pole stejnou délku. Pole je orientováno opět řádkově. Princip hledání i uložení prvků v paměti si osvětlíme na polích s jednobajtovými prvky a postupně vzrůstajícími rozměry. Máme-li jednorozměrné pole s prvním prvkem na adrese 1, je situace jednoduchá. Pořadí prvku určuje jeho adresu. V paměti jsou prvky uloženy za sebou. Přidáme-li druhý rozměr, pak při řádkové orientaci řadíme za sebou jednotlivé řádky, jejichž počet odpovídá druhému rozměru. Má-li druhý rozměr hodnotu 4, pak vlastně za sebe řadíme 4 jednorozměrná pole, v nichž původní index pořadí prvku v těchto jednorozměrných polích zůstává stále týž. Ale druhý rozměr mu přiřazuje druhý index - pořadí celého jednorozměrného pole od prvního zleva (včetně). To je i případ naší předchozí matice.

Pro třetí rozměr si představte několik matic seřazených jako kartičky za sebou. Počet kartiček je třetím rozměrem celé soustavy třírozměrného pole. Chtěli-li jsme se ve dvourozměrném poli dostat k prvku 5,3 (číslo 5 vztaženo ke 2. rozměru, 3 k prvnímu), museli jsme ke 3 přičíst 5-krát délku řádky (jedna řádka je pomyslně výchozím jednorozměrným polem). Budeme-li se chtít dostat k prvku 4,5,3 třírozměrného pole, musíme ke 3 (z "vnitřku" jednorozměrného pole) přičíst 5-krát délku celého jednorozměrného pole a 4-krát délku celého dvourozměrného pole. V paměti je tedy postupně uložena jedna matice za druhou, každá je sálově uspořádána tak, jak se na matici sluší - postupně po řádkách, přičemž (stejný) počet řádek každé matice je dán hodnotou druhého, počet matic hodnotou třetího rozměru.

Přidáme čtvrtý rozměr. Slovo rozměr je tu poněkud matoucí. Pod ním si každý představí rozměr v nějakém prostoru, který v obecném povědomí může mít rozměry nanejvýš tři. Na tento podmíněný reflex zapomeňte. Naopak - uložení prvků pole si představujte přímo na lince adres paměti. Tedy v jednom prostorovém rozměru. Z předchozího návazně vyplývá, že k prvku 8,4,5,3 čtyřrozměrného pole se dostaneme tak, že ke 3 přičteme 5-krát celou délku jednorozměrného, 4-krát celou délku dvourozměrného a 8-krát celou délku třírozměrného pole. Když byly ve třírozměrném poli umístěny matice za sebou do jedné sady, pak ve čtyřrozměrném poli budou za sebou umístěny tyto sady tolikrát, kolik velí hodnota čtvrtého rozměru. Dál už můžete pokračovat sami třeba až do nekonečna.

N-tý rozměr tedy vytváříme tak, že sestavu (N-1)tého pole opakujeme za sebou tolikrát, kolik určuje rozdíl horní a dolní hranice (zvětšený o 1) N-tého rozměru. Z toho m.j. plyne závěr, že s každým přibývajícím rozměrem stále vydatněji ubývá paměti. Dále se podíváme, jak s rozměry pracuje následující rutina. Mějme třírozměrné pole 0..3, 0..5, 0..6. Všechny dolní hranice jednotlivých rozměrů jsou rovny nule. Před vstupem do rutiny provedeme přípravu vstupních dat v zásobníku, z nějž je rutina odebírá:

NRDZM	EQU	3	; Počet rozměrů pole
R3DOL	EQU	0	; Dolní
R3HOR	EQU	3	; a horní hranice 3.rozměru
R2DOL	EQU	0	; Totéž pro 2.
R2HOR	EQU	5	
R1DOL	EQU	0	; a 1.rozměr
R1HOR	EQU	6	
ELEM1	EQU	3	; Bajtová délka elementu 1.rozměru
ELEM2	EQU	((R1HOR-R1DOL)+1)*ELEM1	
ELEM3	EQU	((R2HOR-R2DOL)+1)*ELEM2	

; ELEM2 je délka "jednorozměrného", ELEM3 délka "dvourozměrného" pole. Vysvětleno výše. Dejme tomu, že hledáme adresu prvku ; 1,3,0. Pokračovat v přípravě budeme následovně:

LD	HL, BASE	; 1.adresa pole
PUSH	HL	
LD	HL, 1	; Ostatní data viz výše
PUSH	HL	
LD	HL, ELEM3	
PUSH	HL	
LD	HL, 3	

```

PUSH HL
LD HL,ELEM2
PUSH HL
LD HL,0
PUSH HL
LD HL,ELEM1
PUSH HL
LD HL,NROZM
PUSH HL
CALL NDIM ;Volání rutiny pro výpočet adresy prvku

```

Výpočet provedený rutinou bude: $BASE + (1 \cdot 126) + (3 \cdot 21) + (0 \cdot 3)$. Výsledkem je hledaná adresa prvku. Pokud jsou čísla dolních hranic jednotlivých rozměrů nenulová, je nutno provést úpravu, protože rutina vyžaduje, aby spodní hranici všech rozměrů byla nula. Např. před hledáním adresy prvku -1,6 dvourozměrného pole -5..-1, 2..10 provedeme úpravu v adekvátní části přípravy takto:

```

...
LD HL,-1
LD DE,-(-5)
ADD HL,DE
PUSH HL

...
LD HL,6
LD DE,-(+2)
ADD HL,DE
PUSH HL

```

Výpočet rutiny pak bude: $BASE + (((-1) - (-5)) \cdot 18) + ((6 - 2) \cdot 2)$

Rutina urychluje výpočet offsetu prvku mezi jednotlivými rozměry, když je délka "předchozího pole" menší než 256 a současně je mocninou dvěma. Pokud tomu tak není, musí se provést vynásobení podle algoritmu "posuň-a-přičti".

Délka: 120 bajtů a 5 pro data

Počet cyklů: průměrně 1300 na každý rozměr, plus 165

Vstup: viz příprava výše

Výstup: v HL hledaná adresa prvku pole

Použité registry: HL,DE,BC,AF

```

NDIM POP HL ;Adresa návratu do HL
LD (RETAD),HL ;a do obl.dat
LD HL,0 ;Inicializace
LD (OFFSET),HL ;offsetu na nulu
POP HL ;Počet rozměrů pole do L (H nevyužito)
LD A,L ;a přes reg.A
LD (NUMDIM),A ;do obl.dat
OR A ;Je počet nulový?
JR Z,ADBASE ;Když AND, směr konec

LOOP POP DE ;Délka jednotlivého rozměru
POP HL ;Index hledaného prvku v tomto rozměru
CALL NXTOFF ;Postupný výpočet offsetu prvku
LD HL,NUMDIM
DEC (HL) ;Zmenšení počtu rozměrů o 1
JR NZ,LOOP ;Pokud ještě nějaký zbývá, pokračuj

ADBASE LD HL,(OFFSET) ;Vypočtený celkový offset do HL
POP DE ;Do DE 1.adr.pole
ADD HL,DE ;Určení adresy hledaného prvku
LD DE,(RETAD) ;Do DE adresa návratu
PUSH DE ;a její uložení do zásobníku
RET ;Konečný návrat

```

;Výpočet offsetu postupně rozměr po rozměru:

;offset := offset + (index * délka)

```

NXTOFF PUSH HL ;Uchovej index prvku v moment. rozměru
LD A,D ;Zjistí, zda je délka menší než 256
OR A ;(vyšší bajt musí být nulový)
JR NZ,BIGSZ ;Když NE, skok na násobení
LD A,E ;Nižší bajt délky do reg.A
LD HL,EASYAY ;Do HL adr.1.faktoru posunů
LD B,SZEASY ;Do B počet faktorů
LD C,0 ;Vynulování čítače posunů

EASYLP CP (HL) ;Porovnej s faktorem
JR Z,ISEASY ;Když shoda, je to mocnina dvou, skok
INC HL ;Jdi na adr.dalšího faktoru
INC C ;Zvyš čítač posunů
DJNZ EASYLP ;Opakuj porovnání, dokud buď shoda nebo
JR BIGSZ ;délka není mocninou dvěma, pak skok

```

```

ISEASY POP HL ; Do HL index prvku v rozměru
LD A,C ; Počet posunů do A
OR A ; Je nulový?
JR Z,ADDOFF ; Když AND, faktor=0, skok
LD B,A ; Počet posunů do B

SHIFT ADD HL,HL ; Proved posuny (násobení dvěma)
DJNZ SHIFT
JR ADDOFF ; Jdi přičíst index k offsetu

BIGSZ POP BC ; Index do BC
LD HL,0 ; Do HL bude ukládán výsledek násobení
LD A,15 ; Bitový čítač algoritmu "posuň-a-přičti"

MULP SLA E ; Posun násobitele vlevo
RL D
JR NC,MULP1 ; Když nejvyšší bit=0, skok
ADD HL,BC ; Jinak přičti násobence k meziproductu
MULP1 ADD HL,HL ; Posuň meziproduct doleva
DEC A ; a sniž bitový čítač smyčky o 1
JR NZ,MULP ; Opakuji pro 15 bitů
OR D ; Má bit 15 násobitele hodnotu log.1?
JP P,ADDOFF ; Když NE (kladné číslo), skok
ADD HL,BC ; Když AND, přičti ještě jednou násobence

ADDOFF EX DE,HL ; Následuje přičtení výsledku násobení
LD HL,(OFFSET) ; (index*délka)
ADD HL,DE ; k moment.hodnotě offsetu
LD (OFFSET),HL ; Offset do obl.dat
RET ; Návrat ze subrutiny

EASYAY DEFB 1 ; Oblast definovaných faktorů posunů
DEFB 2 ; pro čísla, která jsou mocninou dvěma
DEFB 4
DEFB 8
DEFB 16
DEFB 32
DEFB 64
DEFB 128
SZEASY EQU $-EASYAY ; Počet faktorů
RETAD DEFS 2 ; Adresa návratu z rutiny
OFFSET DEFS 2 ; Práběžné ukládání hodnoty offsetu

```

Principiálně stejnou funkci mají rutiny interpreterů Basicu při hledání prvku v poli, zadaném příkazem DIM (v Pascalu je to ARRAY). Jak z programování v těchto jazycích víte, nevýhodou je konstantní délka prvků rozměru. Pole tohoto typu jsou "žrouty" paměti (i prvky neobsahující informaci mají stanovenou délku). Uvedená rutina má všechny prvky všech rozměrů stejně dlouhé. Menší úpravou by bylo možno docílit toho, aby délka mohla být definována pro každý rozměr jiná. Z uvedeného vyplývá obecný závěr - tak jako nebudeme rozbíjet ořech počítačem, ale radši kladivem, na kterém zase nic nevypočítáme, a ze všeho toho je k jídlu jen ořech, stejně musíme zvážit volbu typu konstrukce a zpracování dat podle dané situace. Aneb - všechno je k něčemu dobré - ale opravdu jen k něčemu.

5/ Hledání největšího bajtu v poli s max.255 1-bajt.elementy

Rutina prohledává dané pole tak, že jeho 1.bajt považuje za největší do té doby, než najde větší. Pokud se tak stane, zapíše jeho adresu, a pokračuje s tímto bajtem jako největším dál, dokud opět nenajde větší...atd. Pokud je v poli více bajtů s největší hodnotou, adresován zůstane ten, který byl nalezen jako první. Rutina nebere v úvahu znaménka.

Délka: 19 bajtů

Počet cyklů: 36-58 pro každý element plus 35

Vstup: v HL 1.adr.pole, v B délka pole (počet bajtů)

Výstup: CY=1 při nulové délce pole

CY=0 v A hodnota největšího bajtu, v HL jeho adresa

```

MAXELM LD A,B ; Test na nulovou délku pole
OR A
SCF ; Nastavení CY na log.1 pro ev.indikaci
RET Z ; nulové délky pole; nastane-li, návrat

MAXLP LD A,(HL) ; Momentálně největší element do reg.A
LD E,L ; Uložení adr.největšího elementu do DE
LD D,H

```

```

MAXLP1 DEC B      ;Zmenšení čítače počtu bajtů pole o 1
JR      Z,EXIT    ;Když všechny porovnány, skok
INC     HL        ;Na adresu dalšího elementu
CP      (HL)      ;Porovnej element s moment.největším
JR      NC,MAXLP1 ;Když element není větší, pokračuj klid-
JR      MAXLP     ;ně dál, jinak jej jdi uložit

EXIT    OR      A      ;Vynulování CY pro signalizaci "vše O.K."
EX      DE,HL        ;Adr.největšího elementu do HL
RET     ;Návrat
    
```

6/ Hledání 1-bajtového elementu s nejmenší hodnotou v poli s max.255 elementy

Rutina analogická k předchozí. Komentář je proto jen v odlišné části.

Délka: 21 bajtů

Počet cyklů: 36-65 na každý element plus 35

```

MINELM LD      A,B
OR      A
SCF
RET     Z

MINLP   LD      A,(HL)
LD      E,L
LD      D,H

MINLP1  DEC     B
JR      Z,EXITS
INC     HL
CP      (HL)
JR      C,MINLP1 ;Pokračuj v hledání, když je elem.větší
JR      Z,MINLP1 ;nebo stejně velký
JR      MINLP

EXITS   OR      A
EX      DE,HL
RET
    
```

7/ Binární prohledávání pole o max.délce 255 bajtů

Dostáváme se k jednomu z nejpoužívanějších způsobů prohledávání polí (angl.binary search, čte se bájnyrý sérč). Binární se mu říká proto, že zde velkou roli hraje dělení dvěma. Pro toto prohledávání musí být splněn jeden základní předpoklad - prvky pole musejí být seřazeny podle jejich hodnot. Zda vzestupně, či sestupně to už záleží na konkrétním případě. Zatímco nejprimitivnější způsob prohledávání pracuje způsobem "od jednoho konce ke druhému", binární prohledávání jde na věc jinak. Představme si sekvenci čísel, v níž máme najít číslo 88. Pod čísla prvků pole je uvedeno pořadí kroků prohledávání, nad čísla jejich pozice v poli (rutina sama pak používá přímo adresy):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	3	7	11	16	23	24	32	46	48	55	66	87	88	89	97	107	111	134	156	177
										1		3	4		2					

Hledané číslo je nalezeno po 4 krocích. Kdybychom hledali bajt po bajtu, našli bychom je zleva po 14.kroku, zprava po 8. Jak jste asi z nástinu poznali, při binárním prohledávání jdeme vždy do středu té části pole, v níž bude ležet hledané číslo. Do které části to bude, zjistíme porovnáním hledaného prvku s momentálně nalezeným (větší/menší). Je-li prvek shodný, jsme doma. Jediným problémem je určit, zda v případě sudého počtu prvků v části, do jejíhož středu se chystáme skočit, máme zvolit prvek sudý či lichý. To záleží v podstatě jen na programátorovi. Uvedené krokování je adekvátní tomu, s jakým pracuje následující rutina. Nyní akci rozvedeme do jednotlivých kroků. Hranice části pole, do jejíž středu budeme skákat, si označíme LEH (levá) a PRH (pravá), střed jako ST. Při dělení rutina bere v úvahu jen celočíselnou část výsledku.

ST1 = (LEH + PRH)/2 = (0 + 20)/2 = 10
 Porovnání: 88 větší než 55; LEH = ST1 + 1 = 11

ST2 = (LEH + PRH)/2 = (11 + 20)/2 = 15
 Porovnání: 88 menší než 97; PRH = ST2 - 1 = 14

ST3 = (LEH + PRH)/2 = (11 + 14)/2 = 12
 Porovnání: 88 větší než 87; LEH = ST3 + 1 = 13

ST4 = (LEH + PRH)/2 = (13 + 14)/2 = 13
 Porovnání: 88 nalezeno! Jeho adresa = 1.adr.pole + ST4

Když hledané číslo v poli není, bude nakonec LEH větší než PRH. Následující rutina hledá zadaný 1-bajtový prvek v poli o max. délce 255 bajtů. I když to není moc, může vám být vodítkem při konstrukci rutin prohledávání delších a komplikovanějších polí.

Délka: 37 bajtů

Počet cyklů: průměrně 114 na 1 průchod smyčkou plus 53

Vstup: v HL 1. adr. pole, v C délka pole

v A bajt hledané hodnoty

Výstup: Když CY=0, bajt nalezen, v HL jeho adresa v poli

Když CY=1, bajt nenalezen

Použité registry: HL, DE, BC, AF

```

BISCH  INC  C
        DEC  C                ;Je délka pole nulová?
        SCF
        RET  Z                ;Když AND, návrat
        LD   E, L              ;LEH pole do DE
        LD   D, H
        LD   B, 0
        ADD  HL, BC            ;PRH pole + 1 do HL
        DEC  HL                ;proto zmenšit o 1
        LD   C, A              ;Uchova hledaného bajtu do C

LOOP    PUSH HL                ;Uchova PRH
        ADD  HL, DE            ;PRH + LEH
        RR   H                  ;Výsledek vydělen 2 dvěma (posun vpravo)
        RR   L                  ;a zároveň vzata jen jeho celočísl. část
        LD   A, (HL)           ;Do A bajt ze ST (středu části pole)
        CP   C                  ;Porovnej jej s hledanou hodnotou
        JR   NC, TOOLRG        ;ST bajt větší nebo stejný, skok

        EX   DE, HL            ;Bajt ze ST menší, LEH do DE, PRH do HL
        INC  DE                ;proto LEH = ST + 1
        POP  HL                ;Obnov PRH v HL
        JR   CONT              ;a pokračuj na CONT

TOOLRG  INC  SP                ;Stará PRH v zásobníku nepotřebná
        INC  SP
        RET  Z                  ;Návrat při nálezu s CY=0
        DEC  HL                ;PRH = ST - 1

CONT    LD   A, L              ;Test, zde je LEH větší než PRH
        CP   E
        LD   A, H
        SBC  A, D
        JR   NC, LOOP          ;Když NE, pokračuj v hledání
        RET                    ;Když AND, návrat s CY=1

```

KAPITOLA 8

Retězce

Po namáhavé práci na polích se pustíme do řetězců. Bez manipulace s nimi se žádný program neobejde. Každou skupinku znaků, každé slovo, či skupinu slov, vět, čísel a jiných dat obecně můžeme považovat za řetězec a jako s takovým s ním všelijak cílevědomě nakládat. Nejčastější operací je porovnávání řetězců - je základem prohledávání databank, editorů apod.

1/ Porovnávání dvou řetězců po max.255 bajtech

Podmínkou porovnávání v další rutině je informační bajt na začátku řetězce. Jeho obsah oznamuje, z kolika bajtů se řetězec skládá. Informační bajt není součástí obsahu porovnávaného řetězce. V případě, že jsou porovnávány řetězce s rozdílnou délkou, jsou považovány za stejné, když jsou shodné v délce řetězce kratšího. Rozdílnost délek je indikována. V případě, že jsou právě porovnané znaky neshodné, na výstupu budou jejich adresy.

Délka: 33 bajtů a 2 pro data

Počet cyklů: 60 pro každý bajt plus 95, resp. plus dalších 40,

když jsou řetězce shodné v délce kratšího

Vstup: v HL adr.1.řetězce, v DE adr.2.řetězce

Výstup: Když C=0, délka 1.řetězce nulová, pak při

Z=1 oba řetězce s nulovou délkou

Z=0 totéž jako u B=0

jinak když B=0, obsah porovnaných délek shodný, pak při

Z=1 i délka řetězců shodná

Z=0 a CY=1 2.řetězec delší, při CY=0 kratší

jinak když B nenulové, znaky na adr.HL,DE různé, pak při

CY=0 znak 2.řetězce větší

CY=1 " " " "

B určuje počet znaků do konce kratšího řetězce

Použité registry: HL,DE,BC,AF

```

STRCMP LD A,(HL) ;Začíná se zjištěním, který řetězec je
LD (LENS1),A ;kratší; napřed se uloží délky obou do
LD A,(DE) ;obl.dat
LD (LENS2),A
CP (HL) ;Porovnání
JR C,BEGCMP ;Skok, když 2.řetězec kratší
LD A,(HL) ;Jinak je kratší 1.řetězec

BEGCMP OR A ;Je délka 1.řetězce nulová?
LD C,A ;Pro případ nulové délky
JR Z,CMPLEN ;Když AND, směr konec
LD B,A ;V B čítač bajtů
EX DE,HL ;V DE adr.1.řetězce, v HL 2.řetězce

CMPLP INC HL ;Jdi na další adresy řetězců
INC DE
LD A,(DE) ;Do A bajt 1.řet.
CP (HL) ;Porovnej s bajtem 2.řet.
RET NZ ;Když jsou různé, návrat se Z=0
DJNZ CMPLP ;Opakuji pro všechny shodné bajty

CMPLN LD A,(LENS1) ;Porovnej délky shodných řetězců
LD HL,LENS2
CP (HL)
RET ;Návrat s patřičně nastavenými SI

LENS1 DEFS 1 ;Uložení délky 1.řetězce
LENS2 DEFS 1 ;" " 2. "
    
```

2/ Spojování dvou řetězců do max.255 bajtů

Představte si, že máte komprimovaný text (třeba v databance). Jeden z údajů je uložen jako 25Punčochatá. Uživatel si přeje údaj promítnout. 25 je token pro Pipi, což je jeden řetězec. Ten se umístí třeba do "projekčního" bufferu a k němu se připojí další, a to příjmení Punčochatá. Řetězce, které rutina spojuje, mohou být dlouhé rovněž až 255 bajtů. Délku cílového řetězce stanovíme předem. Pokud cílový řetězec "přeteče", je stav signalizován. V řetězci bude jen to, co nepřeteklo. Každý z obou spojovaných řetězců musí mít před začátkem svého obsahu informační bajt o své délce. 2.řetězec je v paměti připojen k prvnímu přímo za jeho poslední bajt.

Délka: 83 bajtů a 5 pro data

Počet cyklů: průměrně $21 \times$ délka 2.řetězce plus 288

Vstup: v HL adr.1.řetězce, v DE adr.2.řetězce

v B požadovaná délka cílového řetězce

Výstup: 1.řetězec je prodloužen o bajty druhého

při CY=0 vše O.K.

při CY=1 buď řetězec "přetekl"; nebo byla žádána délka větší než 255 (tehdy se nic nepřipojí)

Použité registry: HL,DE,BC,AF

```

CONCAT LD (SIADR),HL ;Adr.1.řetězce do obl.dat
        PUSH BC ;Uchova délky cílového řetězce
        LD A,(HL) ;Délka 1.řet.do reg.A,
        LD (SILEN),A ;do obl.dat
        LD C,A ;a do BC
        LD B,0
        ADD HL,BC
        INC HL ;V HL adr.připojení 2.řetězce
        LD A,(DE) ;Délka 2.řet.do reg.A
        LD (S2LEN),A ;a do obl.dat
        INC DE ;V DE adr.prvního bajtu obsahu 2.řet.
        POP BC ;Do B délka cílového řetězce
        LD C,A ;Zjistí předem, zda součet délek
        LD A,(SILEN) ;obou řetězců
        ADD A,C ;není větší než maximum 255
        JR C,TOOLNG ;Když AND, skok
        CP B ;Jinak porovnej s požad.délkou cíl.řet.
        JR Z,LENOK ;Když není přesazena, skok na LENOK
        JR C,LENOK

TOOLNG LD A,#FF ;Indikace přetečení řetězce
        LD (STRGOV),A ;do obl.dat
        LD A,(SILEN) ;Délka 1.řet.do reg.A
        LD C,A ;a do C
        LD A,B ;Délka cíl.řet.do reg.A
        SUB C ;Od ní odečti délku 1.řet.
        RET C ;Když 1.řet.delší, návrat
        LD (S2LEN),A ;Rozdíl (v reg.A) je max.délkou připoje
        LD A,B ;Délka cíl.řet.z reg.B do A
        LD (SILEN),A ;a do obl.dat
        JR DOCAT ;Skok na provedení spoje

LENOK LD (SILEN),A ;Součet délek do obl.dat
        SUB A ;Indikace "žádné přetečení"
        LD (STRGOV),A ;Indik.bajt do obl.dat

DOCAT LD A,(S2LEN) ;Počet znaků připojovaného řetězce do A
        OR A ;Je nulový?
        JR Z,EXIT ;Když AND, směr konec
        LD C,A ;Jinak počet znaků do BC jako čítače
        LD B,0 ;instrukce LDIR
        EX DE,HL ;V DE 1.bajt cíle, v HL zdroje
        LDIR ;Proved přenos 2.řet.k 1.řetězci

EXIT LD A,(SILEN) ;Nová délka 1.řetězce do reg.A
        LD HL,(SIADR) ;Obnovení jeho 1.adr.v HL
        LD (HL),A ;Délka do info-bajtu řetězce
        LD A,(STRGOV) ;Do A indikační bajt přetečení
        RRA ;a jeho rotace do CY
        RET ;Návrat

SIADR DEFS 2 ;1.adr.1.řet.
SILEN DEFS 1 ;Délka 1.řet.
S2LEN DEFS 1 ;Délka 2.řet.
STRGOV DEFS 1 ;Bajt indikace přetečení

```

3/ Porovnávání dvou řetězců po maximálně 65536 bajtech

Tato důvtipná kračoučká rutina porovnává obsahy dvou řetězců bajt po bajtu, dokud se neliší, nebo dokud bajtový čítač není nulový (pak jsou řetězce shodné). Podobně jako u rutiny pod bodem 1/ můžeme porovnat i obsah kratšího s delším, když nastavíme čítač bajtů na jejich počet v kratším řetězci. Na doplnění - porovnávat můžeme i jen části obou řetězců nastavením čítače bajtů.

Délka: 8 bajtů (!)

Počet cyklů: $51 \times$ délka minus 6 (nebo plus 40 při roznosti)

Vstup: v DE a HL adresy řetězců, v BC počet porovnávaných bajtů

Výstup: Když Z=1, shodnost; BC=0, v HL=HL+BC, DE=DE+BC

Když Z=0, neshodnost; BC=?, při S=1 bajt z (HL-1) větší
při S=0 " z (DE-1) "

Použité registry: HL,DE,BC,AF

CPSTR	LD	A, (DE)	; Do A bajt z jednoho řetězce
	INC	DE	; Postup na jeho další adresu
	CPI		; Porovnej s bajtem druhého řetězce
	RET	NZ	; Když jsou různé, návrat
	RET	PD	; Při nulovém čítači BC návrat
	JR	CPSTR	; Opakuji porovnání

Pokud váháte nad RET PD, vysvětlení najdete v minulé učebnici.

4/ Porovnávání s řetězcem vloženým za instrukci volání

V některých případech je škoda nevyužít toho, že po exekuci instrukce CALL máme v zásobníku adresu následující hned za CALL. Využívá se toho různě. Např. tak, že se za CALL umísťuje nějaké hlášení, které bude promítnuto, nebo tabulka, která bude prohledávána volanou rutinou apod. Další rutina porovnává řetězec umístěný těsně za ní s jiným. Oba řetězce musejí končit indikačním nulbajtem. Rutina se vrací na adresu za nulbajtem řetězce umístěného za CALL.

Délka: 25 bajtů

Vstup: Jeden řetězec za instrukcí volání rutiny, v DE adr. druhého

Výstup: Při Z=1 shodnost, při Z=0 neshodnost, DE beze změny

Použité registry: DE, F

COMPAR	EX	(SP), HL	; Uchova HL, nově v HL 1. adr. řet. za CALL
	PUSH	DE	; Uchova adr. 2. řetězce
	PUSH	AF	; a reg. AF
CPLP	LD	A, (DE)	; Bajt z 2. řetězce do reg. A
	CP	(HL)	; Porovnání s bajtem řetězce za CALL
	JR	NZ, NULLLP	; Když jsou různé, skok na hledání nulb.
	INC	HL	; Na další adresy v obou řetězcích
	INC	DE	
	OR	A	; Už se objevil nulbajt?
	JR	NZ, CPLP	; Když ne, pokračuj v porovnávání
EXIT	POP	DE	; Obnovení pův. obsahu reg. A
	LD	A, D	; bez ovlivnění stavu F
	POP	DE	; Do DE 1. adr. 2. řetězce
	EX	(SP), HL	; Do zásobníku dej adr. za nulbajtem
	RET		; a skoč na ni
NULLLP	LD	A, (HL)	; Proběhni zbytek řetězce,
	INC	HL	
	OR	A	
	JR	NZ, NULLLP	; dokud nenajdeš nulbajt
	INC	A	; Překlop na Z=0
	JR	EXIT	; Směr konec

5/ Porovnání dvou řetězců se zápisem diferencí

Zatímco předchozí rutiny vždy "vypadly z akce", když narazily na neshodnost v porovnávaných bajtech, tato ji registruje. A to tak, že staví datové pole, v němž po porovnání všech bajtů budou uloženy rozdílné i s jejich adresami v obou řetězcích. Oba řetězce mohou být libovolně dlouhé, počet porovnávaných bajtů stanovíme čítačem. Své využití taková funkce najde třeba při práci se slovníkem, kdy program porovnáním chybně zapsaného slova se správným (resp. chybně zapsanému tvaru nejbližším) najde rozdílnosti a ev. je pak opraví. Ve výukových programech lze žákovi ukázat chybně zapsaná písmenka i kontrolovat prováděné opravy. Rutinu by bylo možno použít i při komprimaci slovníku do tvaru, o němž pojednávám v kapitole o komprimacích. Zjistila by, kolik znaků je v předcházejícím slovu stejných s porovnávaným a určila jejich počet, což je princip oné komprimace. Na základě těchto informací by mohla pracovat komprimační rutina, která by (třeba i s určitými omezeními) zkomprimovala celý slovník. V některých slovníkových procesorech lze prohledávat text řetězcem znaků, z nichž několik není definováno (např. cenž??ra?je??esmr?elná). Výhodné zvláště při hledání všech možných tvarů českých či slovenských slov, kde před a za kořen slova dáme univerzální znaky a pak ať už je slovo napsáno v jakémkoli mluvnickém tvaru, stejně nám neunikne. Dalším adeptem hledání částí řetězců je databanka. Pro takové prohledávání můžeme samozřejmě zvolit i jiný postup než v rutině uvedený.

Délka: 43 bajtů

Vstup: v DE adr. 1. řet., v HL 2. řet., v IV datového pole

v BC počet bajtů pro porovnání

Výstup: DE a HL obsahují adr. o 1 vyšší než byly adr. posl. bajtů

IV beze změny, v BC počet neshodností

Formát uložení neshodných bajtů s jejich adresami:

Adresa z 1. řet. a bajt z této adresy

Adresa z 2.řet. a bajt z této adresy
(Rutina neprovádí kontrolu, zda polem nepřepisuje
něco, co by neměla)
Použité registry: IV, HL, DE, BC

```
CPARE  PUSH AF      ;Uchovej reg.AF,
        PUSH IV     ;adr.stavěného pole
        PUSH BC     ;a čítač bajtů
        LD  BC,0     ;Vynuluj střadač diferencí

NXTBYT  LD  A,(DE)
        CP  (HL)     ;Porovnání bajtů obo řetězců
        JR  Z,EQUAL  ;Když shodně, skok na EQUAL
        PUSH IV      ;Adr.pole do zásobníku

GETDIF  CP  (HL)     ;Adjustace Z pro smyčku GETDIF
        EX  (SP),HL  ;Adr.pole ze zásobn.do HL kvůli vyšší
        LD  (HL),E   ;rychlosti instrukcí s HL než s IV
        INC HL       ;Adresa nálezu difference do pole (po-
        LD  (HL),D   ;prvé z 1.řet.,podruhé ze 2.řet.; to
        INC HL       ;platí i pro rozdílné bajty)
        LD  A,(DE)
        LD  (HL),A
        INC HL
        EX  (SP),HL  ;Adr.pole zpět do zásobníku
        EX  DE,HL    ;Prohoz adres 1.a 2.řetězce
        JR  NZ,GETDIF;Po prvním proved ještě druhé uložení
        POP IV       ;(CP (HL) bude podruhé s nulou, Z=1)
        INC BC       ;Obnov obsah IV a připočti i diferencí

EQUAL   INC HL       ;Po shodě jdi na další bajty řetězců
        INC DE
        EX  (SP),HL  ;Do HL čítač bajtů ze zásobníku
        DEC HL       ;Zmenši obsah čítače
        LD  A,H      ;Test, zda je nulový
        OR  L
        EX  (SP),HL  ;Obnov obsah HL, čítač do zásobníku
        JR  NZ,NXTBYT;Když čítač nenulový, pokračuj na NXTBYT
        POP AF       ;Odhoď bajt.čítač
        POP IV       ;Obnov 1.adr."chybového" pole v IV
        POP AF       ;a obsah AF
        RET          ;Návrat
```

6/ Hledání dílčího řetězce v řetězci max.255 bajtů dlouhém

nalezne uplatnění např.v databankách při hledání všech Nováků
i s jejich manželkami -ovými, či prohledávání textu kořenem
slova, který bude dílčím znakovým řetězcem všech mluvnických
tvarů slova apod. V podstatě jde o alternativu předchozí úvahy o
prohledávání řetězců za účelem nalezení jejich podobnosti. V
rutině musí oba řetězce předcházet info-bajt o jejich délce. V
komentáři rutiny je dílčí řetězec označen zkratkou subř.,
prohledávaný řetězec jen jako řet.

Délka: 69 bajtů a 7 pro data
Počet cyklů: 36 při nálezu shodného bajtu (jinak 148) plus 154
Vstup: v HL 1.adr.prohledávaného řetězce
v DE 1.adr.hledané části
Výstup: Když A=0, hledaná podobnost žádná
Jinak je v A vzdálenost umístění 1.bajtu dílčího řetězce
od začátku prohledávaného řetězce
Použité registry: HL, DE, BC, AF

```
PDS    LD  (STRING),HL ;1.adr.řet.do obl.dat
        EX  DE,HL      ;a do DE; do HL 1.adr.subř.
        LD  A,(HL)     ;Info-bajt subř.do reg.A
        OR  A          ;Test na nulu
        JR  Z,NOTFND   ;Když délka subř.nulová, směr konec
        INC HL         ;Jdi na 1.bajt obsahu subř.
        LD  (SUBSTG),HL; a jeho adr.
        LD  (SUBLEN),A  ;a délku subř.ulož do obl.dat
        LD  C,A        ;Jeho délka do C
        LD  A,(DE)     ;Totéž pro řet.
        OR  A
        JR  Z,NOTFND   ;Je subř.kratší než řet.?
        SUB C          ;Když ANO, směr konec
        JR  C,NOTFND   ;Rozdíl délek zvyš o 1
        INC A          ;a ulož do B
        LD  B,A
        SUB A          ;Vynuluj obsah reg.A a ulož nulu do obl.
        LD  (INDEX),A  ;dat pro ev.index vzdálenosti 1.bajtu
```

```

SLP1  LD  HL, INDEX      ; subř. od začátku řet.
      INC (HL)           ; Zvětší index o 1
      LD  HL, SUBLEN     ; Délka subř. do C
      LD  C, (HL)        ; Momentální adresu řet.
      LD  HL, (STRING)   ; zvýš o 1
      INC HL              ; a opět ulož do obl. dat
      LD  (STRING), HL   ; Do DE moment. adr. subř.
      LD  DE, (SUBSTG)

CMPLP LD  A, (DE)        ; Bajt ze subř. do reg. A
      CP  (HL)           ; Porovnej s bajtem v řet.
      JR  NZ, SLP2       ; Při neshodnosti skok na SLP2
      DEC C              ; Při shodnosti zmenši čítač délky subř.
      JR  Z, FOUND       ; Když délka nulová, řet. obsahuje subř.
      INC HL              ; Když nenulová, jdi na další adresy
      INC DE              ; v řet. i subř.
      JR  CMPLP          ; Pokračuj v porovnávání

SLP2  DJNZ SLP1          ; Dokud by se subř. vešel do řet., pokračuj
      JR  NOTFND         ; v hledání shodn. bajtu, jinak bez nálezu

FOUND LD  A, (INDEX)     ; Poloha subř. v řet. do reg. A
      RET                ; Návrat

NOTFND SUB A             ; Vynuluj obsah reg. A pro signalizaci ne-
      RET                ; shodnosti a vrať se

STRING DEFS 2            ; Moment. adr. řet.
SUBSTG DEFS 2            ; Moment. adr. subř.
SLEN  DEFS 1             ; Délka řet.
SUBLEN DEFS 1            ; Délka subř.
INDEX DEFS 1             ; Index pozice subř. v řet.

```

7/ Kopie části řetězce max. 255 bajtů dlouhého

V případě, že námi požadovaná kopie nepřesahuje hranice řetězce, je cílový řetězec přenesen na určené místo paměti. Zvláště u této rutiny si povšimněte, kolik testovacích a ev. úpravných a indikačních operací je třeba učinit před tím, než je provedena vlastní kopírovací akce, která zabírá pár bajtů. To je zcela typické pro veškeré assemblerové programování - neustále dávat pozor, aby něco "neujelo" někam, kde by to zcela uniklo naší laskavé pozornosti (paranoia mia).

Délka: 73 bajtů a 2 pro data

Počet cyklů: 21*počet bajtů subř. plus 237

Vstup: v HL 1. adr. zdrojového řetězce, v DE 1. adr. cílového řet.

v A max. délka cíl. řet., v B žádaná délka kopie

v C index pozice 1. kopir. bajtu od začátku zdroj. řetězce, přičemž C=1 znamená 1. bajt jeho obsahu

Výstup: Při CY=0 vše O.K.

Při CY=1 buď délka řet. nulová, nebo kopie přesahuje svou posl. cíl. adr. (tehdy bude kratší o přesah), nebo špatně zadaná pozice 1. kop. bajtu ve zdroj. řet.

Použité registry: HL, DE, BC, AF

```

COPY  LD  (MAXLEN), A    ; Délka cíl. řet. do obl. dat
      SUB A              ; Vynuluj obsah reg. A
      LD  (DE), A        ; a inicializuj jím délku cíl. řet.
      LD  (CPYERR), A    ; a indik. chybový bajt
      OR  B              ; Je-li požadovaná délka kopie nulová,
      RET Z              ; okamžitý návrat
      LD  A, (MAXLEN)     ; Délka cíl. řet. do reg. A
      OR  A              ; Je-li nulová,
      JR  Z, EREXIT       ; směr konec s chybou
      LD  A, C            ; Do A pozice 1. kop. bajtu ve zdroj. řet.
      OR  A              ; Je-li pozice 1. kop. bajtu nulová, chyba
      JR  Z, EREXIT       ; Když je délka zdroj. řet.
      LD  A, (HL)         ; menší než zadaná pozice 1. kop. bajtu,
      CP  C              ; návrat s CY=1
      RET C
      LD  A, C
      ADD A, B            ; 1. pozice kop. bajtu + délka kopie
      JR  C, RECALC       ; Když výsl. přes 255, bude kopie kratší
      DEC A              ; Zvýš spoučet o 1 a zjisti, zda žádaný
      CP  (HL)           ; počet kop. bajtů nejde za posl. kop. bajt
      JR  C, CNT10K       ; Když je posl. kop. bajt před koncem
      JR  Z, CNT10K       ; nebo i přesně na konci zdroje, vše O.K.

RECALC LD  A, #FF        ; Jinak nutno zmenšit čítač kopir. bajtů
      LD  (CPYERR), A    ; Indikace zmenšení čítače do chyb. bajtu
      LD  A, (HL)
      SUB C              ; Délka zdroj. řet. - pozice 1. kop. bajtu
      INC A              ; +1
      LD  B, A           ; V B nový počet bajtů kopie

```



```

CNT10K LD A, (MAXLEN) ;Vejde se kopie do cíl.řetězce?
CP B
JR NC, CNT20K ;Když AND, D.K., skok
LD B, A ;Když NE, zmenši čítač kopír.bajtů o
LD A, #FF ;přesah a indikuj
LD (CPYERR), A ;chybovým bajtem

CNT20K LD A, B ;Ještě zjistí, zda počet kop.bajtů není
OR A ;nulový
JR Z, EREXIT ;Když AND, směr konec
LD B, 0 ;Vynulj obsah B pro součet, určující
ADD HL, BC ;adresu pozice 1.kop.bajtu ve zdroj.řet.
LD (DE), A ;Počet kop.bajtů do info-bajtu cíl.řet.
LD C, A ;a do BC
INC DE ;Jdi na 1.bajt obsahu cíl.řet.
LDIR ;a konečně proved přenos kopie

OKEXIT LD A, (CPYERR) ;Indik.cybový bajt do reg.A
OR A ;Když je nulový, vše D.K.,
RET Z ;a návrat s CY=1

EREXIT SCF ;Jinak nastav CY na log.1
RET ;a vrať se s chybou

MAXLEN DEFS 1 ;Max.délka cíl.řetězce
CPYERR DEFS 1 ;Indikační chyb.bajt

```

B/ Vymazání části řetězce max.255 bajtů dlouhého

Rutina vymaže stanovený počet bajtů z řetězce od stanovené pozice v řetězci. Zbýde-li něco za vymazanou částí, je to připojeno přenesením přes vymazanou část. Pro použití této rutiny v programovém toku musí program tyto parametry určit samozřejmě sám. Volba způsobu jejich určení je závislá na požadované operaci. Když budeme např. chtít vymazat všechny předpony ze slovníku, jehož slova je mají zapsány, pro předpony zvolíme pozici 1 a porovnáním obsahu možných předpon s předponou u každého slova pak program určí počet znaků předpony pro vymazání. U přípon to vezmeme v opačném pořadí (1.pozici bude poslední znak celého slova). Tuto i jiné podobné úvahy berte relativně, neboť v programování jde vše řešit nejen tak, ale především i jinak.

Délka: 58 bajtů a 1 pro data

Počet cyklů: průměrně $21 * (\text{délka řet.} - \text{pozice} - \text{počet maz.bajtů} + 1)$
plus 224

Vstup: v HL 1.adr.řetězce, v B počet maz.bajtů, v C pozice bajtu v řetězci od jeho začátku (1.bajt obsahu má pozici 1, tzn., že na pozici 0 je info-bajt délky řetězce)

Výstup: Při CY=0 vše D.K., zadaná část řet.vymazána
Při CY=1 buď je pozice nulová či leží za koncem řet.
(pak zůstane řet.beze změny); nebo zadaný počet maz.bajtů přesahuje konec řetězce (pak se vymaže od zadané pozice do jeho konce)

Použité registry: HL, DE, BC, AF

```

DELETE SUB A ;Inicializace indik.chyb.bajtu na nulu
LD (DELERR), A ;a jeho uložení do obl.dat
OR B ;Je počet maz.bajtů nulový?
RET Z ;Když AND, návrat
LD A, C ;Totéž pro pozici 1.maz.bajtu
OR A
SCF ;Signalizace nul.pozice (CY=1)
RET Z ;Návrat
LD A, (HL) ;Do A délka řet.
CP C ;Je 1.poz.maz.bajtu kdesi za řetězcem?
RET C ;Když AND, chyba, návrat s CY=1
LD A, C ;Přenes pozici do reg.A
ADD A, B ;a přičti počet maz.bajtů
JR C, TRUNC ;Když je to víc, než 255, skok
LD E, A ;Jinak ulož výsledek do E
DEC A ;Sniž součet o 1
CP (HL) ;a porovnej s délkou řetězce

```

;Když bude mazaná část přesahovat řetězec, nebo bude končit na
na jeho posl.bajtu, pak se změní jen obsah info-bajtu o jeho
délce. Nová délka=pozice 1.maz.bajtu-1

```

JR C, CNTOK ;Je maz.část uvnitř řetězce (AND-skok)
JR Z, TRUNC ;nebo končí na jeho posl.bajtu? ( " )

LD A, #FF ;Je za koncem řet., signalizuj přesah
LD (DELERR), A ;indik.chyb.bajtem a ulož jej do obl.dat

```

```

TRUNC LD A,C ;Změň jen info-bajt
DEC A
LD (HL),A ;a ulož jej na pozici 0 řetězce
LD A,(DEERR) ;Zjistí stav indik.chyb.bajtu
RRA ;jeho posunem vpravo do CY
RET ;Návrat s hlášením v CY

CNTOK LD A,(HL) ;Následuje mazání
LD D,A ;Původní délka řet.do D
SUB B ;minus počet maz.bajtů=nová délka
LD (HL),A ;Ulož ji na pozici 0 řetězce
LD A,D ;Počet znaků pro jejich posun=přv.délka
SUB E ;řet.-(poz.1.maz.bajtu+počet maz.bajtů)
INC A ;+1
PUSH HL ;Ušchovej 1.adr.řetězce
LD B,0 ;Do BC pozice 1.maz.bajtu pro
ADD HL,BC ;určení místa spoje obou částí řetězce
EX (SP),HL ;a jeho uložení do zás.,do HL 1.adr.řet.
LD D,0 ;Výpočet adr., odkud se bude zbývajících
ADD HL,DE ;část řet.přenášet
POP DE ;Obnova adr.spoje (adr.určení LDIR)
LD C,A ;V BC počet přenášených bajtů
LDIR ;a jejich přenos
OR A ;Vynulování CY (signál vše D.K.)
RET ;Návrat

DEERR DEFS 1 ;Indikační chyb.bajt

```

9/ Vložení jednoho řetězce do jiného (délka max.255 bajtů)

Velmi častá operace ve všech editorech (angl.zvaná insertion). V nich je za řetězec (jakožto 1-rozměrné pole) považována celá oblast paměti určená k ukládání textu. Ta je samozřejmě delší než 255 bajtů, není však problém upravit následující rutinu pro manipulaci s delším řetězcem. Podstatou operace je rozdělení výsledného řetězce na dvě části od pozice, na níž bude vkládán jiný. Část vpravo od pozice vložení je po uložení vkládaného řetězce připojena k jeho konci. Výsledný řetězec bude tedy delší o délku řetězce vloženého (za splnění určitých podmínek). Pro lepší orientaci budu v komentáři rutiny používat pro vkládaný řetězec zkratku subř., pro ten, do nějž je vkládáno, jen řet.

Délka: 90 bajtů a 1 pro data
 Počet cyklů: průměrně $21 * (\text{dél.řet.} - \text{pozice} + 1) + 21 * \text{dél.subř.} + 290$
 Vstup: v HL 1.adr.řet., v DE 1.adr.subř., v B max.délka výsl.řet.
 v C pozice vložení v řet. (1.bajt v řet.má pozici 1)
 Výstup: při CY=0 vše D.K.
 když CY=1, pak při pozici=0 nebo délce řet. větší než max.délka výsl.řet. jsou oba řet.beze změny
 jinak když je pozice za koncem řet., je subř. připojen těsně za řet.
 jinak když součet délek obou řetězců je větší než max.dél.výsl.řet., je subř.připojen tak, že končí posl.bajtem výsl.řet., přesah ignor.

Použité registry: HL,DE,BC,AF

```

INSERT SUB A ;Indik.chyb.bajt inicializuj na nulu
LD (INSERR),A ;a ulož do obl.dat
LD A,(DE) ;Info-bajt subř.do reg.A
OR A ;Je jeho délka nulová?
RET Z ;Když AND, návrat

IDX0 LD A,C ;Pozice vložení do C
OR A ;Je to pozice 0?
SCF ;Indikuj chybu (CY=1)
RET Z ;a vrať se

CHKLEN LD A,(DE) ;Do A opět info-bajt délky subř.
ADD A,(HL) ;Přičti délku řet.
JR C,TRUNC ;Když je větší než 255, skok
CP B ;Výsledek porovnej s max.dél.výsl.řet.
LD A,(DE) ;Do A opět info-bajt subř.
JR C,IDXLEN ;Když max.dél.výsl.řet.větší
JR Z,IDXLEN ;nebo stejná, skok na IDXLEN

TRUNC LD A,#FF ;Když bude vložení s přesahem, signali-
LD (INSERR),A ;zuj to indik.chyb.bajtem a ulož jej
LD A,B ;Max.délka do reg.A
SUB (HL) ;Odečti délku řet.; když je sám řet.
RET C ;ještě delší, návrat s chybou (CY=1)
SCF ;Pro případ, že bude řet.stejně dlouhý
RET Z ;jako max.délka nutno nastavi CY zvlášť

```

```

IDXLEN LD B,A ; Délka subř.do B
LD A,(HL) ; Délka řet.do A
CP C ; Porovnej ji s pozicí vložení
JR NC,LENOK ; Když není někde za koncem řet., skok
LD C,A ; Jinak dél.řet.do C
ADD A,B ; Součet délka řet.+dél.subř.=nová délka
LD (HL),A ; výsl.řet.; ulož ji na jeho pozici 0
EX DE,HL ; Prohod 1.adresy obou řet.
LD A,C ; Páv.dél.řet.do C
INC A ; Zvětši ji o 1 (kvůli info-bajtu)
ADD A,E ; a přičti k DE (v DE bude posl.adr.řet.)
LD E,A
JR NC,IDXLI
INC D

```

```

IDXLI LD A,#FF ; Indikuj přesah chybovým bajtem,
LD (INSERR),A ; ulož jej do obl.dat
JR MVESUB ; a skoč na připojení části subř.

```

;Následuje operace vložení řetězce. Bajty řet. od pozice vložení
;se posunou o počet bajtů subř.doprava. Na uvolněné místo bude
;vložena subřetězec.

```

LENOK PUSH BC ; Uchova délky
PUSH DE ; a adresy subř.
LD E,A ; Do DE délka řet.
LD D,0
ADD A,B ; Nová dél.řet.=páv.dél.řet.+dél.subř.
LD (HL),A ; Ulož novou dél.řet.do jeho info-bajtu
LD A,E ; Od páv.dél.řet.
SUB C ; odečti pozici vložení, a přičti 1; v A
INC A ; bude počet bajtů v řet.vpravo od pozice
ADD HL,DE ; Do HL adr.posl.bajtu v řet.
LD E,L ; Přenes ji i do DE
LD D,H
LD C,B ; Do BC délka subř.
LD B,0
ADD HL,BC ; Urči budoucí krajní adr.řetězce
EX DE,HL ; Přenes ji do DE, do HL adr.posl.b.řet.
LD C,A ; Počet přenáčených bajtů do BC
LDDR ; Proved přenos (uvolní místo pro subř.)
EX DE,HL
INC DE ; V DE adr.ulození subř.v řetězci
POP HL ; V HL adr.subř.
POP BC ; V B dél.subř.
MVESUB INC HL ; Jdi na 1.bajt obsahu subř.
LD C,B ; Do BC délka subř.
LD B,0
LDIR ; Proved přenos subř.do výsl.řetězce
LD A,(INSERR) ; Do A obsah indik.chyb.bajtu
RRA ; a jeho posun vpravo do CY
RET ; Návrat s hláškou v CY

```

```

INSERR DEFS 1 ; Uložení indik.chybového bajtu

```

KAPITOLA 9

Třídění

Už Popelka před dávnými lety musela neutříděné pole s prvky typu hrášek a typu čocka utřídít na dvě pole s prvky stejného typu. Protože zlá macecha neměla ani ponětí o výpočetní technice, nenutila Popelku ve třídění obou polí pokračovat - třeba podle velikosti hrachových kuliček, které by Popelka značila indexy apod. Popelce při třídění pomáhali holubi. Nám pomůže počítač. Ale stejně jako holubi dostali informaci o tom, co a jak mají třídít (Popelka je naprogramovala), musíme my naprogramovat počítač.

Při manipulaci s daty jsme se dosud zabývali hlavně jejich vyhledáváním, čili prohledáváním řetězců a polí. Pro četné operace s daty je však zcela nezbytné, aby pole dat byla předem setříděna podle určitých zákonitostí. Těto nezbytnosti jsme se dotkli už při binárním prohledávání pole.

1/ Insertion sort neboli třídění s vkládáním

Rutinu pro insertion sort (čte se inzéršn sôrt) můžete hned použít pro sekvenční seřazení jednobajtových čísel, abyste mohli v jejich poli binárně prohledávat. Pro svou orientaci rutina poli předřazuje 1 nulbajt, aby nepřepsala adresy pod polem. Pokud by vám nulbajt vadil, můžete napřed najít bajt s nejmenší hodnotou v celém poli a použít jej jako záložku místo nulbajtu. Jenže to je operace navíc, a škoda času. Rutina je sympatická právě tím, že třídění provádí "jedním tahem". Na druhou stranu je třeba si uvědomit, že čím více přenosů částí řetězců pro uvolnění pozice k vložení prvku na správnou pozici bude potřeba, tím delší bude čas třídění. Výsledné řazení je vzestupné. Rutina nebere v úvahu znaménka. První tříděný prvek je vždy na 2. pozici pole. Nyní si předvedme názorně podstatu insertion sortu na neutříděném poli:

```

vlož nulu před zač.pole, tříděný prvek bude 3      0 2 3 1 4 5
porov.3-2 (CY=0), ale rozdíl pozic je 1, proto nic, 0 2 3 1 4 5
jdi pro další (vyšší) prvek, tj.1
porov.1-3 (CY=1), porovnávej dál
porov.1-2 ( " ) "
porov.1-0 (CY=0), rozdíl pozic větší než 1, proto
proved posun, X je bud.pozice      0 X 2 3 4 5
a teď vlož prvek 1 na uvolněnou pozici      0 1 2 3 4 5
pokračuj v porovnávání 4-3 a 5-4, vše v pořádku, konec

```

Algoritmus:

Vynuluj bajt před 1.adr.pole

Pro každý bajt pole opakuj:

 přečti tříděný bajt

 Opakuj:

 porovnávej jeho obsah s předešlými

 dokud nenajdeš jeho správnou pozici

 Když jeho moment.pozice není tou správnou, pak

 posuň všechny bajty mezi správnou a moment.pozicí

 o 1 bajt nahoru

Délka: 36 bajtů

Vstup: v DE adr.o 1 menší než 1.adr.pole, v HL počet bajtů pole

Výstup: v DE adresa o 1 větší, než posl.adr.pole; HL=BC=0

na adr.o 1 menší než 1.adr.pole je nulbajt

Použité registry: HL,DE,BC,AF

```

INSERT XOR A          ;Umísti nulbajt
LD      (DE),A        ;před tříděné pole
INC     DE            ;a jdi na jeho 1.adr.

ISLP    PUSH HL        ;Uschovej obsah čítače bajtů pole
INC     DE            ;Jdi na další adr.pole
LD      H,D           ;Zapiš ji i do HL
LD      L,E
LD      A,(DE)        ;Do reg.A bajt z pole

BACK    DEC HL         ;Jdi o 1 adr.zpět
CP      (HL)          ;Bajt na této adrese porovnej s tříděným
JR      C,BACK        ;Když je tříděný menší, hledej dál
PUSH    DE            ;Když je větší, ulož adr.tříděného bajtu
EX      DE,HL         ;do zásobn.a do HL; v DE adr.nálezu

```

DEC	HL	; Zmenši adr. tříd. bajtu o 1 a obě adresy
SBC	HL, DE	; odečti; když je rozdíl nulový, třídění
JR	Z, NXT	; bajt je těsně nad menším prvkem, proto
		; můžeš jít pro další bajt na třídění
LD	B, H	; Jinak musíš provést posun všech bajtů
LD	C, L	; mezi správnou a moment. poz. bajtu nahoru
ADD	HL, DE	; Do BC počet bajtů k přesunu; v HL adr.
LD	D, H	; pod moment. pozicí tříděného bajtu
LD	E, L	; v DE adr.
INC	DE	; o 1 vyšší
LDDR		; Proveď přenos
LD	(DE), A	; a vlož tříd. bajt na uvolněnou pozici
NXT	POP DE	; V DE obnov adr. utříděného bajtu
	POP HL	; Do HL čítač bajtů
	SCF	; Nastav CY na log. 1 a
	SBC HL, BC	; zmenši jím obsah čítače (po LDDR BC=0)
	JR NZ, ISLP	; Když je ještě co třídít, pokračuj
	RET	; Jinak návrat

2/ Bubble sort, neboli třídění záměnami

Toto třídění patří k nejpomalejším vůbec. Ale pracovat s počítačem a nevědět nic o bubble sortu, to prostě nelze. Hlavně z toho důvodu, že je to jediné třídění, u nějž nemusíme provádět žádné příliš složité předběžné přípravné práce pro třídění řetězců o různé délce. Princip třídění si ukážeme na neutříděném poli s pěti 1-bajtovými elementy (probublávaný prvek je podtržen):

Vstupní stav	5 4 2 3 1	Proběhlé výměny:
po 1. průběhu	4 2 3 1 5	5-4 5-2 5-3 5-1
po 2. "	2 3 1 4 5	4-2 4-3 4-1
po 3. "	2 1 3 4 5	3-1
po 4. "	1 2 3 4 5	2-1
po 5. "	1 2 3 4 5	nedošlo k výměně, konec

Z toho je patrné, že začátku pole bližší prvek, který je menší, než následující, probublává polem vzhůru, dokud další následující prvek není větší. Pro probublání každého prvku rutina spouští samu sebe. Tento pochod se označuje slovem rekurzivní. Doba bublání je závislá na vstupním rozmištění prvků. Kdyby prvky byly rozmištěny přesně opačně, než jak budou po svém seřazení, počet průběhů i doba jejich třídění by byly nejdelší z možných. Z časových důvodů se bubble sort nerad používá, a když, tak jen pro třídění menších polí, či pro jejich dotřídění po předchozím zpracování velkého pole quick sortem (k tomu viz quick sort). Nebo když máme k dispozici příliš malou paměť pro třídění velkých polí.

V následující rutině je rozhodující stav indikátoru výměny. Po probublání každého prvku rutina zjišťuje, zda si po cestě vyměnil pozici s jiným. Když indikátor signalizuje, že k takové výměně došlo, rutina se opět rekurzivně spustí. Když projde pole, aniž došlo k výměně, třídění je hotové. Rutina seřazuje libovolně dlouhé prvky o stejné délce. Pro práci s prvky o různé délce je nutno provést drobné úpravy, které se budou opírat o informaci o délce každého prvku - info-bajt na jeho začátku, nebo nul-bajt jako jeho zakončovací znak. Rutiny s touto identifikací jsou v učebnici. Porovnávání by pak probíhalo v délce kratšího z obou prvků (rutina rovněž v učebnici). Volba postupu probublávání nestejně dlouhých prvků je pak už závislá jen na invenci programátora. Je zřejmé, že mění-li si pozici nestejně dlouhé prvky, ty, které jsou v poli umístěny mezi nimi, se musejí přesunout o rozdíl v délce měněných, a to buď nahoru nebo dolů. Když do tohoto pole něco přidáváme, nebo z něj bereme, dochází k posunům se změnou délky celého pole. Tomu se hezky říká dýchání pole. Bylo by možné i předem stanovit pevnou max. délku prvků, a ty, které jsou kratší, doplnit zprava stejnými bajty (třeba u textových řetězců mezerami). Takováto úprava však zbytečně promarní spoustu paměti.

Rutina k řazení používá 1- nebo 2-bajtový klíč, který může být prvku představen jako jeho index určující jeho požadovanou (!!!) pozici v poli (ve smyslu nižší než soused s vyšším klíčem a vyšší než soused s menším klíčem). Nebo klíči mohou být bajty na těchto pozicích všech prvků. Třeba při řazení nestejně dlouhých prvků budou po prvním řazení 1-bajtovým klíčem na pozici 1 prvky Novotný a Novák před Procházkou a Paškem, ale to ještě není abecedně v pořádku. Po druhém řazení, před nímž musíte upravit

pozici klíče na 2, bude výsledné řazení: Novotný, Novák, Pašek, Procházka. Nováka dostaneme před Novotného až po třídění s klíčem na 4.pozici prvků. K výměně prvků během probublávání je použit buffer (napřed první prvek do bufferu, pak druhý na jeho místo a nakonec první z bufferu na uvolněnou pozici po druhém). Celková doba rekurzí tedy závisí i na délce prvků.

Rutina využívá faktu, že po každém průběhu se na správnou pozici dostane prvek, který má menší hodnotu než již seřazené prvky s hodnotou vyšší (prvek s hodnotou nejvyšší je umístěn na konci pole). Proto se může počet zpracovávaných prvků pole po každém průběhu zmenšit o 1, přičemž každý průběh začíná vždy od nejnižší pozice pole. Díky tomu, podle stupně "nepříznivosti" vstupního rozmištění prvků můžeme ušetřit až 50 procent času.

Délka: 170 bajtů a 13 pro data; délka bufferu podle délky prvků
Počet cyklů: většinou moc

Vstup: viz definovanou oblast dat; POZOR - bajty adres i čísel jsou uloženy "v opačném gardu"

Výstup: všechny registry změněny, oblast dat až po BIT včetně beze změny

Použité registry: HL, DE, BC, AF

;Konstantní data:

STRT	DEFS	2	;1.adr.pole
DISP	DEFS	2	;Vzdálenost mezi sousedními prvky
TEMP	DEFW	STRS	;Adr.začátku bufferu
LEN	DEFS	2	;Délka prvku
KEY	DEFS	2	;Pozice klíče v prvku (klíč+1)
BIT	DEFS	1	;0 pro 1-bajt.klíč, jiná hodn.pro 2-baj.

;Proměnná data:

CNT	DEFS	2	;Počet prvků pole (-1 po každ.průběhu)
TOGO	DEFS	2	;Počet smyček v 1 průběhu rutiny
ADDR	DEFW	2	;Adr.klíče moment.proublávaného prvku
FLAG	DEFS	1	;Bajt indikace proběhlé výměny prvků
STRS	DEFS	(LEN)	;Bufferu v délce LEN
BSORT	LD	HL, (CNT)	;Proved nutnou inicializaci registrů
	DEC	HL	;Počet smyček v průběhu
	LD	(TOGO), HL	;ulož do obl.dat
	SUB	A	;Vynuluj indik.bajt výměny pro signali-
	LD	(FLAG), A	;zaci "žádná výměna" a ulož do obl.dat
	LD	HL, (STRT)	
	LD	DE, (KEY)	
	ADD	HL, DE	;Najdi adr.klíče v 1.prvku
	DEC	HL	
	LD	(ADDR), HL	;a ulož ji do obl.dat
BUBLP	LD	D, (HL)	;Do DE obsah klíče z pozice klíč+0
	INC	HL	
	LD	E, (HL)	;a klíč+1
	DEC	HL	;Vrať na adresu klíč+0
	LD	A, (BIT)	;Zjistí, zda je klíč
	CP	0	;1- či 2-bajtový
	JR	NZ, KEYE	;Když 2-bajtový, skok
	LD	E, 0	;Pro 1-bajtový vynuluj bajt klíče v E
KEYE	LD	BC, (DISP)	;Vypočti adr.uložení klíče vyššího
	ADD	HL, BC	;sousedů
	LD	B, (HL)	;a klíč ulož do BC
	INC	HL	
	LD	C, (HL)	
	DEC	HL	
	CP	0	;Opět vynulování bajtu klíče (v C), když
	JR	NZ, KEYC	
	LD	C, 0	;je klíč 1-bajtový
KEYC	LD	HL, (ADDR)	;Adr.klíče bubl.prvku do HL
	LD	A, B	;Porovnej vyšší bajty obou klíčů
	CP	D	
	JP	M, EXCH	;Když je bubl.klíč vyšší, skok na výměnu
	JR	NZ, NEXT	;Když je menší, posuň se na další prvek
	LD	A, C	;Porovnej nižší bajty obou klíčů
	CP	E	
	JP	P, NEXT	;Když bubl.klíč menší či stejný, posuň
			;se na další prvek
EXCH	LD	DE, (KEY)	;Následuje výměna prvků; do DE poz.klíče
	AND	A	;Vynuluj CY pro SBC
	SBC	HL, DE	;Odvoď 1.adr.bubl.prvku od pozice klíče
	INC	HL	
	LD	DE, (TEMP)	;Adr.bufferu pro uložení bubl.prv.do DE
	LD	BC, (LEN)	;Do BC délka prvku
	LDIR		;Proved přenos bubl.prvku do bufferu


```

LD DE, (LEN)      ;Následuje přenos vyššího souseda na
AND A              ;přev. pozici bubl.prvku
SBC HL, DE         ;Obnov přev.adr.bubl.prvku v HL
PUSH HL            ;a uschovej ji
LD DE, (DISP)      ;Z rozdílu uložení adres obou prvků
ADD HL, DE          ;odvod 1.adr.vyššího souseda
POP DE             ;Do DE zapiš přev.adr.bubl.prvku
LD BC, (LEN)       ;a souseda na ni přenes
LDIR               ;

LD DE, (LEN)       ;Nyní z bufferu přeneseme bubl.prvek
AND A              ;na přev.adr.souseda, takže bubl.prvek
SBC HL, DE         ;s větším klíčem se dostane za prvek
PUSH HL            ;s menším klíčem; nová adresa klíče
POP DE             ;bubl.prvku bude vypočtena prvními in-
LD HL, (TEMP)      ;strukcemi části NEXT
LD BC, (LEN)
LDIR               ;

LD A, 1            ;Indikuj proběhlou výměnu pro rekurzi
LD (FLAG), A       ;a informaci ulož do obl.dat

NEXT LD DE, (DISP)
LD HL, (ADDR)
ADD HL, DE          ;V HL bude adr.klíče bubl.prvku
LD (ADDR), HL       ;Ulož ji do obl.dat
LD DE, (TOGO)       ;Zmenší čítač vnitřní smyčky o 1
DEC DE
LD (TOGO), DE       ;a ulož jej do obl.dat
LD A, D             ;Je už nulový (celý 1 průběh hotov)?
OR E
JP NZ, BUBLP        ;Když NE, probublávej dál
LD HL, (CNT)        ;Když ANO, pak
DEC HL              ;pro ev.další rekurzi zmenšení vstupní
LD (CNT), HL        ;čítače vnitřní smyčky
LD A, (FLAG)        ;Zjistí, zda v průběhu došlo
CP 1                ;k výměně prvků
JP Z, BSORT         ;Když ANO, rekurze bubláni
RET                 ;Když vše seřazeno, návrat

```

Na první pohled je patrné, že rutinu by bylo možno leccjak upravit, zkrátit. Ponechal jsem ji v tomto provedení, aby nepřišla o možnou maximální míru srozumitelnosti.

3/ Quick sort, neboli třídění segmentováním pole

Tímto tříděním se dostáváme do oblasti vyšších rychlostí zpracování. U rychlosti se na chvíli zastavme. Provedení každé rutiny trvá nějaký čas, který je v určité závislosti na její konstrukci a funkci. Zakopaný pes u rutin manipulujících s hromadou dat je hlavně v jejich "smyčkování". Čím je komplikovanější, čím větší části polí mnohonásobně přenáší, porovnává, upravuje, tím je doba mezi startem a cílem delší. Teoretici programování znázorňují dobu provedení algoritmu v závislosti na počtu N zpracovávaných prvků symbolem $O(f(N))$. Jím říkájí, že doba provedení je proporcí např. k $N \log N$ (veškeré uvedené logaritmy jsou dvojkové). Vyjádření se pak zapisuje ve tvaru $O(N \log N)$. Zrovna tento případ hovoří o vysoké rychlosti zpracování dat. O něco pomalejší je $O(N)$, dost pomalé $O(N^2)$, o $O(N^3)$ už nemluvě. Např. rychlost zpracování dat algoritmem $O(N^2)$ oproti algoritmu $O(N \log N)$ pro $N = 10000$ může při stejných vstupních podmínkách být - a teď se podívejte - u algoritmu prvního 1,5 dne, u druhého pouze 80 vteřin! Náš milý bubble sort patří k těm "dlouhověkým".

Jak tedy třídění zrychlit? Dejme tomu, že máme utříděné pole, do něhož chceme ukládat další prvky. Což takhle použít binární prohledávání k nalezení pozice pro vložení dalšího prvku a pak jej do pole zašoupnout jen vkládací částí rutiny insertion sortu? Neboli - po nalezení pozice pro nový prvek provést pouze jeho vložení. Algoritmus binárního prohledávání dokáže v utříděném poli najít hledaný prvek v čase $O(\log N)$! Ale když máme pro vložení prvku do tohoto pole použít tu část algoritmu insertion sortu, která posouvá část pole (N prvků) pro uvolnění místa vložení nového prvku, bude to vyžadovat čas $O(N)$ (celý insertion sort ale $O(N^2)$!). Obojí je pro každý nový prvek nutno opakovat. No, není to nejhorší, ale nešlo by to rychleji?

Quick sort nabízí fintu segmentování pole. Jde o rozdělení jednoho velkého pole na dva segmenty A a B. Tak si rozdělíme jeden velký problém na dva menší. Hovoříme dále pro jednoduchost o třídění pole, jehož prvky jsou čísla. V obou nově vzniklých segmentech jsou čísla sice zpráházená, ale v segmentu A budou jen čísla menší, než je nejmenší číslo segmentu B. Kdybychom teď použili insertion sort, výsledný čas zpracování obou segmentů

(každého zvlášť) by byl kratší, než kdybychom jím třídili celé vstupní pole. Rutina quick sortu je rekurzivní, může tedy segmenty dělit pořád dál. Tak by teoreticky zbylo N neprázdných subsegmentů, které by představovaly jednotlivé seřazené prvky pole o N prvcích. Na této nejspodnější úrovni segmentování leží problém quick sortu. Je jasné, že čím více subsegmentů bude rutina vytvářet, tím více času na to bude potřebovat. Proto je vhodné zpracování zastavit vždy, když subsegment obsahuje v průměru 10-20 prvků. A pak takto vzniklé pole buď vcelku (např. insertion sortem) nebo po částech (třeba i bubble sortem) dorazit jiným tříděním. Tím se velmi podstatně (až o 50 procent!) zvýší efektivita třídění oproti tomu, kdybychom nechali quick sort dojet až do konce. Touto kombinací dosáhneme velmi výhodného času $O(N \log N)$.

Nyní k tomu, jak rozdělit pole na segmenty. Má-li být segment A přibližně stejně dlouhý jako segment B, je nutné najít takové číslo, které je co nejblíže ideální střední pozici v poli. Pokud máme pole, o němž víme, že od nejvyššího prvku dolů je každý další menší o 1, a navíc známe jeho hodnotu i počet prvků, problém odpadá. Jenže to je gusta teorie. V radě prvků pole mohou být "dirty", neruku-li, že mnohé mohou mít stejnou hodnotu (to je vůbec ten nejhorší případ) a taky nemusíme vědět o hodnotách prvků vůbec nic. Což vše dohromady se stane nejčastěji. Nejjednodušším řešením pro volbu hodnoty prvku s možnou střední pozicí v poli je porovnání tří čísel - dvou z obou konců pole a jednoho z jeho středu. To, které bude mít střední hodnotu, se stane hraničním mezi oběma budoucími segmenty, resp. subsegmenty. Jiné, elegantnější řešení, používá generátor pseudonáhodných čísel v intervalu daném oběma krajními čísly pole. V quick sortu tedy hraje náhodnost svou velkou roli. Při nepříznivé volbě hraničního čísla můžeme vstupní pole rozdělit i na segmenty, z nichž A bude obsahovat 1 prvek, a v B bude jejich zbytek. A takhle by to mohlo pokračovat pořád dál. Pravděpodobnost této směly je však velmi malá (sázíte sportku?), proto segmentování "vždy nějak dopadne".

Určené hraniční číslo se uloží na začátek pole. A pak začne porovnávání tohoto čísla s ostatními, přičemž se začne prvkem na opačném konci pole/segmentu. Tím se vytvoří i výchozí (nejvzdálenější) pozice okrajů, které se k sobě budou postupně přibližovat. Znárodně si to schématicky (hraniční číslo, již umístěné v čele, je 3):

Pole	3	6	1	5	4	2	
1.krok	2	1					3
2.krok		3			1	6	
3.krok			3		1		
4.krok			3	1			
5.krok	2	1	3	1	5	4	6

3-2 ! prohod a zvyš.lev.okraje
3-6 ne jen sniž.prav. "
3-4 ne jen " "
3-5 ne jen " "
3-1 ! prohod a zvyš.lev. "

Na začátku 6.kroku rutina zjistí, že oba okraje určené pozicí hraničního čísla a svislou čárkou jsou shodné (resp. jsou na stejných adresách) a zpracování tohoto pole ukončí. Výsledkem jsou segmenty 2 1 a 5 4 6 s hraničním číslem 3, které je na své správné pozici v celém poli - proto se nemusí účastnit dalšího třídění. Quick sort obecně postupuje tak, že postupně setřídí levý segment a nakonec pravý. Nejdůležitější momenty při porovnávání prvků s hraničním je vzájemné přibližování okrajů segmentu a průběžné prohazování hraničního prvku s ostatními v případě přesunu testovaného prvku do nižší/vyšší části sub/segmentu.

Následující rutina neřeší případy výskytu prvků stejných hodnot. Pokud se vyskytnou, je jejich konečná pozice nepředpokladatelná. Proto si před použitím quick sortu musíte být jisti, že všechny hodnoty jsou rozdílné, nebo po quick sortu použít ještě jiné vhodné dotřídění (které, jak uvedeno výše, se tak jako tak doporučuje). Pro zajímavost - nejdéle quick sortu trvá třídění pole, jehož všechny prvky jsou shodné. Rutina třídí 2-bajtové prvky a nebere v úvahu znaménka.

Délka: 206 bajtů a 8 pro data

Počet cyklů: $200 * (N \log N) + 300 * (2 * N + 1)$

Vstup: v HL adr.1.prvku pole, v DE adr.posl.prvku pole
v BC nejnižší přípustná adresa zásobníku

Výstup: Když CY=0, pole setříděno

Při CY=1 přetekl zásobník, pole nedotříděno

Použité registry: HL,DE,BC,AF

```
QSORT  PUSH HL          ;Ulož 1.adr.pole
        LD HL,10         ;Stanovení prahu zásobníku radši o 10
        ADD HL,BC        ;adres výš
        LD (STKBTM),HL   ;Práh do obl.dat
        LD HL,2          ;V zásobníku adresuj
        ADD HL,SP        ;návrátovou adresu
        LD (OLDSP),HL    ;ulož ji do obl.dat pro ev.přeteč.zásob.
        POP HL           ;V HL obnov 1.adr.pole
```

```

SORT  LD  (FIRST),HL ;a ulož ji do obl.dat jako levý okraj
      EX  DE,HL
      LD  (LAST),HL  ;Totéž pro adr.posl.prvku (pravý okraj)

PARTIO LD  A,E        ;Zjistí, zda levý okraj (dále jen LOKR)
      SUB L          ;je na stejné adrese jako pravý (POKR)
      LD  A,D        ;nebo je mezi nimi rozdíl jen 1 adresy
      SBC A,H        ;(v poli/segmentu pak není žádný nebo je
      RET NC         ;jen 1 prvek); když AND, návrat, tuto
                        část už nelze segmentovat
      CALL MEDIAN    ;Jinak jdi najít hraniční prvek (HP)
      LD  C,0        ;Inic.směrového reg.C (viz část OK)

PARTLP LD  A,E        ;Následuje zjištění, zda pohyblivý LOKR
      SUB L          ;i POKR už nejsou na stejné adrese
      LD  A,D        ;(provedení testu stejné jako v PARTIO)
      SBC A,H
      JR  NC,DONE    ;Když AND, skok na rekurzi nebo návrat

      CALL COMPARE   ;Porovnej HP s dalším prvkem
      JR  C,OK       ;Když je vzájemná pozice v pořádku,
      JR  Z,OK       ;resp.oba prvky jsou shodné, skok na OK
      CALL SWAP      ;Jinak HP s prvkem prohoď
      INC C          ;Dej na vědomí změnu směru posunu okraje

```

;Reg.C hraje roli výpravčího posund. Na začátku je inicializován na nulu, ale jen kvůli vynulování svého bitu 0. Jeho stav rozhoduje o tom, zda se po prohození prvků bude posouvat LOKR nahoru nebo POKR dolů na další prvek. Každou instrukcí INC C se mění obsah bitu 0, C buď z 1 na 0 nebo z 0 na 1. Rytmus stídání posund je týž.

```

OK     BIT  0,C        ;Test stavu bitu 0,C
      JR  Z,UP        ;Když je nulový, půjde POKR dolů
      EX  DE,HL       ;Jinak LOKR nahoru; dej adr.horn.prvku
      CALL NEXT       ;do HL a posuň se nahoru na další prvek
      EX  DE,HL       ;Vrať adresy zpět (v DE adr.LOKR,
      JR  PARTLP      ;v HL POKR) a pokračuj v krasojízdě

UP     CALL PREV      ;Sniž POKR na další prvek
      JR  PARTLP      ;a swinguj dál

DONE   LD  HL,(STKBTM);Když LOKR i POKR na stejných adresách,
      OR  A           ;zjistí, jestli zásobník nepřekročil
      SBC HL,SP       ;práh
      JR  NC,ABORT    ;Když AND, směr konec, nedotříděno

```

;Následuje důležitá rekurzní část. První CALL SORT rekurzivně seřadí levý segment. Jak vidíte, do zásobníku jsou ukládány obě registry DE a HL, ale i - a v tom je kus chytrosti - postupně i návratové adresy prvního CALL SORT. Všechny tyto uložené adresy zůstávají uloženy v zásobníku do doby, než je provedena instrukce RET NC v části PARTION. Poté jsou odebírány instrukcemi POP DE a POP HL za 1. CALL SORT a instrukcí RET za 2. CALL SORT (když dojde k návratu z RET NC). Razení adres je přesně takové, jak je potřeba, na RET vždy přijde adresa prvního POP DE (nebo OR A) a do DE a HL vždy adresy pole. Pro každou rekurzi volanou 1. CALL SORT se do zásobníku ukládají 3 adresy, tedy 6 bajtů. 2. CALL SORT je pak postupně odebírá, až (po seřazení pravého segmentu) skončí na adrese návratu z rutiny. Z bajtů, které zůstanou v zásobníku zapsány, můžete zpětně odvodit průběh třídění. Vyzkoušejte si třídění malého pole pomocí krokování monitorem strojového kódu a sledujte hlavně zásobník vždy, když jsou prováděny tyto instrukce. Za tu podívanou to stojí.

;Nyní je na řadě rekurze levého segmentu

```

      LD  HL,(LAST)   ;Posl.adr.POKR do HL
      PUSH HL         ;i do zásobníku
      EX  DE,HL       ;DE do HL
      CALL PREV       ;pro posun na prvek pod ST (ten je na
      EX  DE,HL       ;svém místě); ulož jej do DE
      LD  HL,(FIRST)  ;Do HL adr.LOKR
      CALL SORT       ;Proveď rekurzi

```

;Další část je pro rekurzi pravého segmentu a závěrečný návrat

```

      POP DE          ;Do DE POKR
      POP HL          ;a posl.LOKR ze zásobníku

```

	CALL NEXT		; Posuň se na adr.vyššího prvku
	CALL SORT		; Proved rekurzi
	OR A		; Vynuluj CY pro případ konečného návratu
	RET		; "Návraty" na POP DE nebo OR a nakonec
			; závěrečný návrat
ABORT	LD SP, (OLDSP)		; Když zásobník přejde přes práh, do SP
	SCF		; návratová adresa, CY=1 pro signal.stavu
	RET		; a návrat
MEDIAN	LD A, L		; Najdi střed segmentu: INT (POKR-LOKR)/2
	ADD A, E		; Použity 8-bitové operace, protože adresy musejí zůstat nezměněny
	LD C, A		
	LD A, H		
	ADC A, D		
	LD B, A		
	RR B		; Výsledek odečtu vyděl dvěma posunem
	RR C		; vpravo
	RES 0, C		; Učiň adr.středu lichou (střed dále ST)
	BIT 0, E		; Je adr.LOKR sudá?
	JR Z, MED1		; Když AND, skok
	INC C		; Když NE, udělej lichou i adr.ST
MED1	PUSH HL		; Ulož POKR
	LD L, C		; Adr.ST z BC do HL
	LD H, B		
	CALL COMPAR		; Porovnej obsah LOKR a ST
	POP HL		; Obnov POKR v HL
	JR NC, MIDD1		; Když LOKR=ST nebo je větší, skok
	PUSH DE		; Když je menší, zbývá porovnat POKR
	LD E, C		; se ST
	LD D, B		
	CALL COMPAR		; Porovnej
	POP DE		; Obnov adr.POKR v DE
	JR C, SWAPMF		; Skok, když POKR větší či
	JR Z, SWAPMF		; roven ST; pak ST=HP
	CALL COMPAR		; Porovnej LOKR a POKR
	RET NC		; Návrat, když POKR větš.či stejný; LOKR=HČ
	JR SWAPLF		; Jinak POKR=HP
MIDD1	CALL COMPAR		; Porovnej LOKR a POKR
	RET C		; Návrat, když POKR větší
	RET Z		; či stejný; pak LOKR=HP
	PUSH DE		; Jinak porovnej ST a POKR; ulož LOKR
	LD E, C		; Adr.ST do DE
	LD D, B		
	CALL COMPAR		; Porovnej
	POP DE		; Obnov adr.ST v DE
	JR C, SWAPLF		; Návrat, když POKR větší než ST; POKR=HP
SWAPMF	PUSH HL		; Prohoď HP v ST s LOKR (na čelo segm.)
	LD L, C		; Po uložení POKR zapiš do HL adr.ST
	LD H, B		
	CALL SWAP		; Proved prohoz
	POP HL		; Obnov adr.POKR v HL
	RET		; Návrat
SWAPLF	CALL SWAP		; Prohoď HP v POKR s LOKR (na čelo segm.)
	RET		; Návrat
NEXT	INC HL		; Jdi na adr.dalšího prvku vzhůru
	INC HL		
	RET		; Návrat
PREV	DEC HL		; Jdi na adr.dalšího prvku dolů
	DEC HL		
	RET		; Návrat
; Porovnávání prvků segmentů s jejich adresami v DE (prvek1) a v			
; HL (prvek2). Na výstupu platí, že když:			
		CY=0 a Z=0	prvek1 větší než prvek2
		CY=1 a Z=0	" " menší "
		CY=0 a Z=1	oba prvky stejné
COMPAR	INC HL		; Do DE a HL adr.vyšších bajtů
	INC DE		; obou porovnávaných prvků
	LD A, (DE)		; Porovnej je
	CP (HL)		
	DEC DE		; Vrať adresy na nižší bajty obou prvků
	DEC HL		
	RET NZ		; Když jsou vyšší různé, návrat
	LD A, (DE)		; Jinak porovnej nižší
	CP (HL)		
	RET		; Návrat

;Prohození obou prvků, v DE adr.prvku1, v HL adr.prvku2

```

SWAP  LD  B,(HL)      ;Napřed prohoz nižších bajtů
      LD  A,(DE)
      LD  (HL),A       ;prvek1 z reg.A na adr.prvku2
      LD  A,B          ;prvek2 z reg.B do reg.A
      LD  (DE),A       ;a na adr.prvku 1
      INC HL           ;Totéž pro vyšší bajty prvků
      INC DE
      LD  B,(HL)
      LD  A,(DE)
      LD  (HL),A
      LD  A,B
      LD  (DE),A
      DEC HL           ;Zpět na nižší adresy obou prvků
      DEC DE
      RET              ;Návrat

```

```

FIRST DEFS 2          ;Výchozí adr.LOKR
LAST  DEFS 2          ;" " " " POKR
STKBTM DEFS 2         ;Adresa spodního prahu zásobníku
OLDSP DEFS 2          ;Návratová adresa z celé rutiny

```

Protože s délkou zpracovávaných prvků pracují jen subrutiny, nebyl by problém tuto délku upravit podle potřeby. A ještě k rychlosti - rozházené pole 2-bajtových prvků v rozsahu 32K dokáže rutina, vedená Z80 při 4 MHz, zpracovat za cca 40 vteřin. Možná se vám to zdá příliš, ale uvědomte si, že tolik prvků nějaké položky v databance mikropočítače nikdy dohromady nedáme. A hlavně - prvky, s nimiž v databance pracujeme, jsou většinou již seřazeny před přidáváním nových. Následné "vtřídění" nových prvků je rychlejší než řazení prvků v rozházeném poli.

Ještě poznámka k rekurzi. Zcela na začátku je instrukcí PUSH HL (před 1.CALL SORT) uložena pravá krajní adresa celého pole. Ta bude odebрана instrukcí POP DE (před 2.CALL SORT) až po seřídění celého levého segmentu, jak byl vytvořen zcela prvním dělením celého pole. Jakmile se uvedeným odběrem POP DE začne tříditi pravý segment, začne zase zásobník narůstat do doby, než bude znova plně aktivována instrukce RET NC. Pak už zásobník hubne až k adrese konečného návratu z celé rutiny. Zásobník se tedy "nafukuje a vypouští" v průběhu třídění dvakrát. Přitom se pochopitelně přepisují trojice adres, které v něm byly zapsány při třídění levého segmentu. Proto potřebný rozsah zásobníku neodpovídá celkovému počtu rekurzí. V průměru je pro něj zapotřebí prostor log N. Možná, že někomu pochopení principu třídění rutinou quick sortu dá trochu zabrat (mně svítilo až na druhý den).

KAPITOLA 10

Tabulky

Konstrukce datových tabulek a manipulace s nimi jsou důležitou složkou každého programu ve strojovém kódu (tedy i interpreteru Basicu, kompilátoru Pascalu a pod.). Tabulky obsahují konstantní i proměnné hodnoty. Jimi mohou být adresy skoků, klíče (indexy dat), (systémové) proměnné programu, znakové řetězce apod. Zatímco vyšší programovací jazyk veškerou práci s tabulkami a jejich daty dělá za vás, při tvorbě programu v assembleru musíte konstrukce tabulek dat i způsoby manipulace s nimi zvolit a vytvořit sami.

Základními operacemi při práci s tabulkami jsou - jak už ani jinak být nemůže - čtení a zápisy určitým způsobem organizovaných dat. V podstatě jde o to, abychom ve zmeti dat přečetli vždy to, co přečíst chceme, a na správné místo zapsali to, co zapsat chceme, aniž bychom poškodili organizaci dat a jejich formát. Tak např. je více způsobů uložení uživatelských dat v databankách. Jsou-li formáty těchto dat různé, nejsou data jedné databanky přímo zpracovatelná druhou bez patřičné konverze formátu na takový, s nímž ta která databanka pracuje. To se týká uživatelských dat obecně. Standardizace formátu těchto dat by byla vítaná. Jsme však teprve na počátku vývoje computerové techniky, členitá cesta ke standardu bude ještě dlouhá.

Kromě uživatelských dat (a informačních hlášení pro orientaci uživatele) tabulky obsahují data nezbytná pro samotnou funkci programu. O tom, jaká data to budou, musíte rozhodnout sami. Faktem je, že šikovnou konstrukcí tabulek dosáhneme zkrácení programu, zvýšení jeho přehlednosti, stupně upravenosti na jiné aplikační podmínky a neřídka i rychlosti provedení částí programových funkcí. Názory na to, kde a jak mají být data uložena, se různí. Někdo dává přednost jedné velké oblasti dat na začátku či konci programu, jiný je má "rozházená" v různých programových částech s použitím rozličných fint. I to tvoří rukopis autora. Niže uvedené rutiny poskytují základní orientaci ve způsobech práce s tabulkami dat.

1/ Jump table, neboli tabulka adres programového větvení

Pro jednoduchost ji budu dále říkat přímým překladem - skoková tabulka. Je zvláštní, jak někteří čeští technici, zabývající se výpočetní technikou vědecky, začnou být nerudní, když místo stykového rozhraní (co na to Sigmund Freud?) použijete interface, místo počítače computer, místo jazyka symbolických adres napíšete assembler atd. Dokonce bajt týrají slabikou! Skoková tabulka je už vůbec hříchem z nejtěžších. Zato jim ale vůbec nevadí, že se dámský kapesníček nejmenuje kabelníček a že vteřinu zničili sekundou. S přecráváním computerové latiny na mateřštinu mají problémy všechny slovanské jazyky. Mám zato, že by se touto činností měli zabývat literáti (nejlépe básníci, pokud možno se smyslem pro humor), nikoli technici, kteří použijí první vysušené popisný tvar, jaký je napadne, a hned z něj udělají svátost. Když jde o termíny úzkého vědního oboru, s nímž lid obecný nepříjde do styku, budiž. Ale počítače jsou "kabelníčkem" celé společnosti. A do tvorby jejího obecného slovního vybavení by hlavy magnetoskopové měly mít vstup zakázán.

Takže zpět ke (s prominutím) skokové tabulce. Jsou momenty v životě programátora, kdy se mu v jedné rutině začnou hromadit rozhodovací momenty, vnášející nutnost v tom kterém případě vyskakovat z toho kterého místa rutiny na různé adresy. Nejlepším řešením situace je právě tabulka s adresami skoků (ale i volání). Testy v rozhodovací rutině zjistíme, co se má stát, a podle toho určíme buď pořadí adresy v tabulce skoků či vyhledáme její klíč. Programové řízení pak převedeme na nalezenou adresu. Následující rutina vyhledává adresy v tabulce podle pořadí jejich uložení. To může být v intervalu 0..128. Skok typu JP na vyhledanou adresu je imitován odběrem adresy skoku ze zásobníku instrukcí RET na konci rutiny. Proto návrat z hledané subrutiny půjde za poslední adresu volání (nebude-li programové stanovené jinak). Pro ty, co si přesně nevzpomínají - instrukce RET nedělá nic jiného, než že přenesla do reg.PC adresu, která je na spodku zásobníku. Její název RETurn (vrát se) platí jen ve spojitosti s "její" instrukcí CALL. Jinak je to ale jen skok na adresu ze spodku zásobníku. Podobně CALL je rovněž pouhým skokem (ale s přidruženou výrobou návratové adresy v zásobníku, která ovšem nemusí nutně být návratová).

Délka: 21 bajtů
 Počet cyklů: 117
 Vstup: v A pořadí adresy v tabulce, tabulka začíná na adr.JMPTAB
 a počet adres je na adr.LENSUB
 Výstup: CY=1 chyba, zadané pořadí přesahuje nejvyšší existující
 CY=0 vše O.K.
 Použité registry: AF

JTAB	CF	LENSUB	;Když je vstupní hodnota A vyšší než
	CCF		;nejvyšší pořadí v tabulce,
	RET	C	;návrat s CY=1
	PUSH	HL	;Uchova obsahu reg.HL
	ADD	A,A	;Pořadí*2
	LD	HL,JMPTAB	;Do HL 1.adr.tabulky
	ADD	A,L	;K přičtení pořadí*2 použito osmibitové
	LD	L,A	;operace, aby ostatní registry zůstaly
	LD	A,0	;nezměněny
	ADC	A,H	;Do A přenos z ADD A,L (CY=0 nebo 1)
	LD	H,A	;Adresa umístění hledané adresy v HL
	LD	A,(HL)	;Nižší bajt hledané adresy do A,
	INC	HL	
	LD	H,(HL)	;vyšší do L,
	LD	L,A	;nižší z A do L
	EX	(SP),HL	;Hotová adresa do zásobníku a
	RET		;skok na ni jejím odběrem instrukcí RET
LENSUB	EQU	NN	;NN je počet adres v tabulce
JMPTAB	DEFW	XXXX	;1.adr.tabulky
	DEFW	XXXX	;a její pokračování

2/ Skoková tabulka klíčovaných adres s formátem dat: info-bajt - 1-bajt.klíče - 2-bajt.hodnoty

Dostáváme se ke druhé alternativě uložení adres v tabulce. Rutina prohledává tabulku pomocí 1-bajtového klíče a vrátí 2-bajtovou hodnotu. Těmito dvěma bajty nemusí být zrovna adresa, ale cokoli, třeba zase klíč k něčemu jinému. Tak můžeme vytvářet několikavrstvou (vektorovanou), provázanou strukturu, jejíž jednotlivé vrstvy odemykáme pomocí klíčů. Anglicky se uvedenému způsobu prohledávání tabulky říká associative search, což nemá nic společného s představivostí (kromě nutné programátorovy imaginace při konstrukci tabulky), ale značí to hledání něčeho, k danému klíči přidruženého (asociace jako sdružení).

V další rutině musí tabulka ve svém prvním bajtu obsahovat informaci o počtu klíčů v tabulce umístěných, přičemž nulový bajt znamená maximálních 256 klíčů, FFH je 255 klíčů atd. Základní možnosti konstrukce tabulky jsou dvě - buď po každém klíči následuje k němu přidružená hodnota, nebo celá tabulka sestává ze dvou částí (v jedné jsou jen klíče, ve druhé přidružené hodnoty). Následující rutina pracuje s druhým způsobem uložení dat; první v pořadí jsou klíče.

Délka: 31 bajtů
 Vstup: v A hledaný klíč, v HL 1.adr.tabulky
 Výstup: A beze změny
 při Z=1 klíč nalezen, v HL hledaná hodnota
 při Z=0 klíč nenalezen, v HL adr.posl.klíče v tabulce
 Použité registry: HL,AF

ASRCH	PUSH	BC	;Uchovej obsah reg.BC
	LD	B,0	;Inic.B na nulu
	PUSH	AF	;Ulož hledaný klíč v reg.A
	LD	A,(HL)	;Do A info-bajt o počtu klíčů
	CP	B	;Je jich 256?
	JR	NZ,BCND	;Když méně, skok
	INC	B	;Jinak nastav BC na 0100H
BCND	LD	C,A	;V BC počet klíčů
	POP	AF	;Obnov klíč v A
	ADD	HL,BC	;Do HL adr.posledního klíče
	PUSH	HL	;Ulož ji
	CPDR		;Porovnej tab.klíčů s reg.A odzadu
	POP	HL	;Obnov adr.posl.klíče v HL
	JR	NZ,EXIT	;Když klíč nenalezen, směr konec
	PUSH	DE	;Uchovej DE
	PUSH	AF	;a AF
	INC	HL	;Jdi na 1.adr.tab.2-bajt.hodnot
	ADD	HL,BC	;V BC po CPDR pozice klíče; HL+2*pozice
	ADD	HL,BC	;klíče je adresa hledané 2-bajt.hodnoty
	LD	E,(HL)	;Do DE postupně ulož oba nalezené bajty
	INC	HL	
	LD	D,(HL)	
	EX	DE,HL	;a přenes je do HL
	POP	AF	;Obnov klíč v A
	POP	DE	;a obsahy reg.DE a BC
EXIT	POP	BC	
	RET		;Návrat

Pokud by hledanou hodnotou byla adresa, na niž by mělo ihned přejít programové řízení, upravíme konec rutiny za POP DE třeba takto:

```

      POP BC
      JP (HL)
EXIT  POP BC
      RET

```

Rutina je velmi snadno upravitelná i pro jiný počet bajtů hledané hodnoty (např. pro převod tokens na text, řetězce apod.).

3/ Skoková tabulka klíčovaných adres s formátem dat: info-bajt a opakované 1-bajt.klíč/2-bajt.hodnota

Tato, již zmíněná konstrukce má své nevýhody v rychlosti zpracování i menší variabilitě při větších změnách obsahu tabulky - musí se vynechávat tu 1, tu 2 bajty, a měnit délku hledaných hodnot v průběhu programu je prakticky nemožné. Na druhou stranu je prohledávací rutina o poznání kratší než předchozí. Tabulka opět začíná info-bajtem o počtu klíčů (podmínky jako minule).

Délka: 19 bajtů

Vstup: v A hledaný klíč, v HL 1.adr.tabulky

Výstup: A beze změny
při Z=1, CY=0, nález, HL=BC=nalezená hodnota
při Z=0, CY=1, klíč nenalezen, B=0, C beze změny, v HL
adr.o 1 vyšší než konec tabulky

```

KYCSH LD B,(HL)      ;Do B (čítač DJNZ) počet klíčů v tabulce
      INC HL         ;Na adr.1.klíče

KYSLP CP (HL)        ;Klíč tabulky porovnej s hledaným klíčem
      JR NZ,NXT      ;Když to není on, skok
      INC HL         ;Když je to on, jdi na adr.hledané hod-
      LD C,(HL)       ;noty a oba její bajty umísti přes BC
      INC HL
      LD B,(HL)
      LD L,C
      LD H,B
      RET            ;a vrať se

NXT   INC HL         ;Posuň se na adr.dalšího klíče v tabulce
      INC HL         ;(je o 3 adresy dál)
      INC HL
      DJNZ KYSLP      ;Všechny klíče porovnány? Když NE, zpět
      SCF            ;Když ANO, klíč nenalezen; signalizuj
      RET            ;to CY=1 a vrať se

```

Pokud bychom po nalezení hledané hodnoty chtěli hned přejít na adresu v HL, místo RET na konci části KYSLP použijeme JP (HL).

4/ Skoková tabulka s formátem uložení dat: 3 info-bajty a opakované 1-bajt.klíč/2-bajt.adresa

Rutina využívá nalezenou hodnotu jako absolutní adresu skoku, který také provede. Kromě prvního info-bajtu o počtu klíčů má tabulka v dalších dvou adresách skoku pro případ, že klíč v tabulce nalezen nebude. Toto řešení je v určitých případech velmi výhodné.

Délka: 30 bajtů

Počet cyklů: 38*pořadí klíče+158 při nálezu (jinak +162)

Vstup: v A hledaný klíč, tabulka uložena od adr.CASE

Výstup: všechny reg.beze změny, při nálezu klíče progr.řízení
převedeno na nalezenou adresu, jinak na adresu uloženou
ve 2.a 3.bajtu tabulky

Použité registry: A

```

CASBYT PUSH HL      ;Uložení obsahu registrů do zásobníku
      PUSH DE
      PUSH BC
      PUSH AF
      LD HL,CASE     ;1.adresa tabulky do HL
      LD B,(HL)       ;Počet klíčů tab.do B
      LD DE,3        ;Rozdíl adres mezi klíči do DE

LOOP  ADD HL,DE      ;Jdi na adr.dalšího klíče
      CP (HL)        ;Porovnej jej s hledaným v reg.A
      JR Z,KIN       ;Když je to on, skok na KIN
      DJNZ LOOP      ;Jinak pokračuj v porovnávání

KOUT  LD HL,CASE     ;Když klíč nenalezen, 1.adr.tab.v HL
      INC HL         ;Jdi na adr.nížšího bajtu skok.adresy
      LD C,(HL)
      INC HL         ;a ulož ji do C

```

LD	H, (HL)	;a vyšší bajt do H,
LD	L, C	;nakonec nižší ho L
POP	AF	;Obnov obsahy registrů
POP	BC	
POP	DE	
EX	(SP), HL	;Připrav adresu skoku do zásobníku
RET		;a proved skok instrukcí RET

**5/ Skoková tabulka s formátem uložení dat (opakovaně):
adresa skoku - info-bajt (délka klíče) - klíč (max.255 bajtů)**

Touto rutinou zpracovávaná data se ve svém formátu liší hlavně tím, že klíče k jednotlivým adresám skoků mohou být různé dlouhé. Tak jimi mohou být i textové řetězce, třeba příkazy, které jsou ihned provedeny exekucí subrutiny na adrese tomu kterému příkazu přidružené. Není-li klíč v tabulce nalezen, je programové řízení převedeno na adresu, uloženou na dvou předposledních adresách tabulky. Na té poslední leží nulbajt, jenž celou tabulku zakončuje.

Délka: 40 bajtů

Vstup: v DE 1.adr.porovnávaného řetězce

Výstup: převedené řízení na jednu z adres skoků

Použité registry: DE

CASXT	PUSH HL	;Uschova registrů do zásobníku
	PUSH BC	
	PUSH AF	
LD	HL, CASE	;Do HL 1.adr.tabulky
NXTKEY	LD C, (HL)	;Skoková adr.do BC
	INC HL	
	LD B, (HL)	
	INC HL	
	PUSH BC	;Uschova skok.adr.do zásobníku
	LD A, (HL)	;Do A info-bajt
	OR A	;Je nulový?
	JR Z, DONE	;Když AND, je to zakončovací bajt, skok
	LD B, A	;Když NE, dej info-bajt z A do B (čítač
	INC HL	;délky řetězce); jdi na 1.bajt řetězce
	PUSH DE	;Uschovej 1.adr.hledaného klíče
CPSTRG	LD A, (DE)	;Porovnej bajty obou řetězců
	CP (HL)	
	JR NZ, NEXT	;Když jsou různé, skok na NEXT
	INC DE	;Při shodnosti jdi na další adr.obou
	INC HL	;řetězců
	DJNZ CPSTRG	;Dokud neporovnány celé, pokračuj
DONE	POP DE	;Obnov 1.adr.výchozího klíče v DE,
	POP HL	;adr.skoku do HL
	POP AF	;a obnov obsahy reg.AF a BC
	POP BC	
	EX (SP), HL	;Do zásobníku zapiš nalezenou adr.skoku
	RET	;a skoč na ni instrukcí RET
NEXT	LD C, B	;Když řetězce neshodné,
	LD B, 0	
	ADD HL, BC	;posuň se na další v tabulce
	POP DE	;Do DE zpět 1.adr.hledaného klíče
	POP BC	;Odhoď nepotřebnou ar.skoku
	JR NXTKEY	;a jdi porovnávat další řetězec

6/ Table look-up, neboli hledání řetězce v tabulce dat

Další rutina dělá část toho, co minulé, ale pro doplnění tématu ji zařazuji. Rutina porovnává jeden řetězec s řetězcí umístěnými v tabulce. Účelem pátrání je zjistit, zda porovnávaný řetězec má svého jmenovce v tabulce. Máme-li takovou funkci v programu, může po ní následovat větvení (je-li řetězec v tabulce, učin to, jinak ono). Délka řetězců může být max.255 bajtů, přičemž první bajt informuje o délce řetězce (včetně sebe). Pro vyhledání každého řetězce je třeba znát adresu uložení jeho info-bajtu. Hledání postupuje rychle, protože při první neshodě dvou bajtů obou řetězců rutina v dalším jejich porovnávání nepokračuje a jde hned na začátek dalšího řetězce v tabulce. Kromě rychlosti je výhodou možnost porovnávání řetězců s různou délkou. Použít rutinu můžete např. při "šmátrání" v položkách databanky (uživatel si zadáním nějakého řetězce přeje zjistit, zda databanka obsahuje stejnojmennou informaci) apod. Subrutinou rutiny je CPSTR, kterou najdete v části o řetězcích.

Délka: 23 bajtů

Vstup: v HL 1.adr.tabulky, v DE adr.info-bajtu porovnávan.řetězce
v BC počet řetězců v tabulce

Výstup: při Z=0 řetězce neshodné, BC=0, DE beze změny
 při Z=1 nalezen shodný, v BC pořadí řet.v tab.odzadu
 DE=1.adr.porovn.řet.+jeho délka
 HL=" shodn. " + " "

Použité registry: HL,DE,BC,AF

;POZOR - startovací adresa je TLOOK

```

TSTLP  PUSH DE          ;Uchovej 1.adr.hledaného řet.
       PUSH BC          ;a počet řetězců v tabulce
       LD  B,0           ;Do BC délka momentálně porovnávaného
       LD  C,(HL)        ;řet.z tabulky (BC bude čítačem CPI
       CALL CPSTR        ;v subrutině CPSTR)
       ADD HL,BC          ;Jdi na 1.adr.dalšího řetězce v tab.
       POP BC            ;V BC obnov čítač počtu řet.v tabulce
       JR  Z,MATCH       ;Při shodnosti obou řet.Z=1, skok
       POP DE
       DEC BC            ;Zmenši čítač počtu řetězců o 1

TLOOK  LD  A,B           ;a zjisti, zda je už nulový
       OR  C
       JR  NZ,TSTLP      ;Když ještě NE, pokračuj v hledání
       INC A             ;Nastav Z na 0
       RET              ;a vrať se bez nálezu

MATCH  INC  SP           ;Posuň se v zásobníku
       INC  SP
       RET              ;na návratovou adresu
  
```

Někdy by se nám mohlo spíš hodit, abychom na výstupu měli v DE i HL první adresy obou řetězců. K tomu stačí pár drobných úprav, které pro vás už neznamenaají žádný problém.

KAPITOLA 11

Jiné způsoby organizace dat

1/ Hashing

Tato kapitola neobsahuje přímo rutiny, ale zabývá se principy některých dalších způsobů manipulace s daty. Konstrukce tabulek i klíčování dat mají mnoho podob. Představím vám teď jednu, která mluví za mnohé. Ale napřed pár nezbytných zasněcovacích informací.

S každou tabulkou se pojí určitý způsob prohledávání, a jde-li o dynamická data (především uživatelská), i třídění. Oběho je mnohem více druhů, než se může do rozsahu učebnice vejít. Jednou z vtipných manipulací s daty je tzv. hashing (čte se hešing). Je to určení klíče výpočtem. Klíč je tedy proměnnou hodnotou, která určuje, co se kde má hledat, či kam co uložit. Dejme tomu, že máme do pole textových řetězců uložit slovo - kazeta. Program před jeho uložením provede výpočet podle formule, která je platná (ale nemusí být) pro všechny zpracovávané řetězce. Naši jednoduchou formulkou bude součet všech ASCII kódů řetězce a vynásobení výsledku dvěma. U slova kazeta bude výsledek 1280. Naše pole ukládaných slov začíná na adrese třeba 30000. K ní přičteme 1280. Na adresu 31280 se chystáme uložit slovo kazeta. A ejhle - počítač signalizuje, že už tam nějaké slovo má. Rekne mu - tak najdi první volnou adresu za ním (či za dalšími slovy, pokud tam ještě nějaká jsou) a slovo ulož od ní. Počítač provede, slovo uloží. A tak to udělá se všemi. Občas při hashingu může dojít k některým komplikovanějším situacím, které jsou ale programově řešitelné. Prostoru uložení slov, která mají stejný klíč, se anglicky říká list, který je tímto klíčem linkován (linked list). Jednoduše - je to tabulka, jejíž počáteční adresa je dána určitým klíčem, jenž je společný všem prvkům v tabulce umístěným. Tabulek tedy bývá větší množství.

Budeme-li hledat nějaký řetězec v poli slov, počítá opět vypočte klíč, přejde na vypočtenou adresu a začne hledat. Hledaný řetězec buď najde nebo ne, což nám způsobem jemu vlastním oznámí. Hlavní předností hashingu (při dobře zvoleném koeficientu pro výpočet klíče) je rychlost ukládání i hledání. Jeho nevýhodou je velká spotřeba paměti (mnoho adres uvnitř pole může zůstat neobsazeno). Proto ne vždy je tím nejlepším řešením, což se vlastně vztahuje na všechno.

A nyní slíbený bonbónek. Bude jím konstrukce komparativního slovníku, v němž nebude ani jedno slovo. Podobný slovník jsme si představili v části o komprimování dat. Jeho úkolem je upozornit nás na slova, která nezná. Z našeho pohledu to budou slova buď špatně napsaná (překlep, hrubka apod.) nebo napsaná správně, ale ve slovníku nezanesená. Stačí nám tedy, když slovník zjistí, jen to, zda kontrolovaný tvar slova obsahuje či ne. Slovník samozřejmě bude provádět stripping (svléknutí před/přípon) testovaných slov, takže bude v sobě mít uloženy jen slovní kořeny. Pro jejich uložení použijeme hashing. Jak jsme si v části o komprimaci dat ukázali, u slovních kořenů s běžnou komprimací moc neuspějeme. Ale na druhou stranu nechceme všechny slovní kořeny ukládat v jejich plném tvaru (moc paměti nazmar).

2/ Bitová mapa

V úvodu první kapitoly jsem předestřel úlohu, která byla výhodně řešitelná pomocí bitové mapy (používá se i označení bitový vektor). Mapu použijeme při konstrukci našeho slovníku. Místo přímého ukládání celých řetězců do jednotlivých tabulek použijeme pro jejich umístění opět nějaký koeficient, pomocí něž vypočteme jen bitovou pozici uložení prvního znaku řetězce v tabulce od jejího začátku (nejde tedy o ukládání na první volnou adresu v tabulce, jak bylo zvažováno výše). Tak dostaneme např. (střeleno od boku) tabulku, v níž budou slova reprezentována jen jedním bitem ve stavu log.1:

nultá pozice tabulky

00100001100000011000000010010000000100000000000000000000000000
kočka kurděje
kůže hrábě šásek km houně forma

Bity jsou počítány od 1. bitu 1. adresy každé tabulky. V pořadí první bit je poziční bit 0 tabulky, z uvedeného vidíme, že po

159							
35				101			
22		29		44		48	
15	17	25	26	21	24	33	47

Každý prosívaný prvek si vyměňuje pozici s prvkem, na jehož místo se posouvá (vzdálená podobnost s výměnami v quick sortu). Připojujeme-li prvky na nejnížší úrovni stromu, hovoříme o prosívání nahoru. Můžeme je ale prosívat i shora dolů. Prosejme tak třeba 11. Vkládaný prvek je porovnán se 159. 11 je menší, tak dolů. Je menší než 35 i 101, ale z těch je 35 menší, proto pod něj. Atd., až pod 15, a to na (ze zásady) levou větev, protože je volná. Obdobně postupujeme při hledání prvku ve stromu. Protože jsou jednotlivá patra směrem dolů stále hustší, dalším prostupováním jich vlastně stále více máme - odtud vysoká rychlost operace. Razení prvků podle jejich hodnoty může být i vzestupné. V tom případě je na špičce nejnížší hodnota.

Chceme-li z neuspořádaného pole sestavit binární strom, můžeme postupovat dvěma způsoby. Buď prosíváním shora nebo zdola. Každý z obou způsobů přináší odlišné výsledky. Nejzávažnějším problémem při prosívání shora je, jak zajistit, aby prvek vlevo pod špičkou měl střední hodnotu. Jinak může v průběhu jeho konstrukce (ale i při jiných operacích) dojít ke stavu tzv. nevyváženého stromu. Představte si, že máte neutříděné pole 100 prvků s hodnotou od 1 do 100. Po prvních krocích prosívání shora bude špička takováto:

100	
1	99

Dále budou všechny prvky směřovat jen pod číslo 99, pod 1 nic, protože menší než 1 už v poli nic není. Strom bude nejnevyváženější možný, časový zisk v poměru k počtu zpracovávaných prvků nejmenší možný, mrhání pamětí největší. Takový strom je proto více než vhodně vyvážit. Ideální by byla tato výchozí sestava:

100	
50	99

Všechna čísla 99..51 by byla v pravé větvi, 50..1 v levé. Reorganizaci lze provést nejjednodušeji tak, že obě větve pod špičkou zrušíte, najdete prvek se střední hodnotou a usadíte jej vlevo pod špičkou. Další prosívání už povede k vyváženému stromu. Je zřejmé, že vyvážení stromu během vkládání většího počtu prvků dosáhneme spíše prosíváním zdola.

K tomu si musíme říci něco o uložení prvků stromu do paměti a orientaci v nich. Jsou dva základní způsoby. Jeden ukládá data tak, že na 1. adrese uložení prvků je špička, dál nižší patro zleva, další opět zleva, atd. Protože každé patro má dvakrát víc prvků než předchozí, je orientace v takto uloženém stromu bez větších problémů. Vše probíhá bez nutnosti klíčování pozic prvků v paměti, jejichž ukládání kráčí pamětí. Tím se vyznačuje druhý způsob. Při pohledu na strom vidíte, že klíče pro každý prvek mohou být až tři. S uložením každého nového prvku s jeho klíči paměti valem ubývá, ale někdy nelze jinak, než klíče použít.

Nyní si probereme uložení pozičně indexovaných prvků vyváženého stromu v paměti počítače. Prvky p označíme indexem, uvádějícím pozici prvku od 1. adresy uložení (na té bude špička):

p1							
p2				p3			
p4		p5		p6		p7	
p8	p9	p10	p11	p12	p13	p14	p15

Uložení v paměti bude v pořadí daném indexem:

p1 p2 p3 p4 p5 p6 p7 p8 p10 p11 p12 p13 p14 p15

Povšimněte si, že všechny prvky na levé straně každých dvou větvíček mají sudý index, na pravé lichý. Lichá je i špička. Z indexu prvku vycházíme při práci se stromem. Když manipulujeme nějakým prvkem a potřebujeme jej porovnat s jeho sousedem, napřed programově zjistíme, zda je index sudý či lichý. Když je sudý (je vlevo), pak má soused index o 1 větší (a naopak).

Porovnáváme-li vertikálně, pak - basicovsky řečeno - najdeme index prvku ve vyšším patře výpočtem $\text{INT}(pX/2)$, pascalovsky $pX \text{ div } 2$. Směrem dolů je to $2*pX$ na sudý prvek, na lichý $2*pX+1$. Eventuálně neznámý index X prvku odvodíme z jeho vzdálenosti od 1. adr. tabulky prvků. Tak také poznáme, kdy jsme na špičce (abychom v porovnávání nešli nad ni). Z paměťové mapy stromu je zřejmé, že připojování nových prvků zdola jednoduše sekvenčně prodlužujeme pole prvků v paměti. Tak bude strom stále vyvážený. Nevýhodou tu však je, že v celé levé větvi mohou být i prvky s vyšší hodnotou, než bude mít nejmenší prvek v celé pravé části stromu a naopak. Při prosívání shora budou co do hodnot prvky rozděleny podle svislé osy procházející špičkou, ale strom může být nevyvážený v závislosti na rozdilu mezi hodnotou prvku vlevo pod špičkou vůči hodnotě centrálního prvku celého pole. Při pohledu na rozmístění prvků v paměti vás jistě napadnou další možnosti manipulace s nimi.

Binárnímu stromu se anglicky říká jak binary tree (čte se bájnry trít), tak také heap (híp), v překladu hromada. Třídění prvků pomocí stromu se označuje slovy heap sort. Jeho podstatou je sled dvou operací - napřed se z neuspořádaného pole sestaví strom, a z něj pole uspořádané. Jednou z možností, jak pracovat s daty, je jejich sekvenční setřídění "jednou za čas". Z neutříděného pole napřed sestavíme strom a z něj prvky převedeme do sekvenčně utříděného pole. Když přibývají nějaká další data, pracujeme s nimi jiným způsobem (třeba v bufferu), ale ve vztahu k utříděnému poli. Když už se data příliš "nažouknou", nebo nastane jiná nevyhnutelná nutnost, prvky znovu proženeme stromem. Na první pohled se vám to může zdát komplikované. Ale výsledné třídění, byť složené ze dvou operací, je velmi rychlé. O to rychlejší, oč máme určitou část dat už sekvenčně zpracovanou předem, a víme tak kudy na ni. Každá z obou operací - sestavení stromu a jeho převedení na utříděné pole - zabere čas po $O(\log N)$, dohromady $O(N \log N)$. Při každém převodu pole na strom musíme počítat s potřebou části paměti navíc (zvláště bude-li nevyvážený). V opačném směru už je to lepší - prvky ze stromu snadno převedeme do sekvence bez použití jediného bajtu navíc (sekvence "vroste do stromu"). Nevýhodou binárního stromu je obtížná manipulace s prvky o proměnné délce. Zde se bez indexování, často i kombinací s dalšími typy operací, neobejdeme.

Ne vždy je třeba převádět strom na utříděnou sekvenci dat. Takový "trvalý strom" má své využití např. v operaci zvané řazení priorit. Při obsluze velkého množství periferních zařízení (nebo v soustavě vnitřních přerušení počítače či jiných hierarchických funkcí) může být stanoveno pořadí jejich priorit. Jinými slovy - periférie se řadí do fronty na obsluhu. Přidělíme-li těm nejprivilegovanějším "legitimace s nejnižšími čísly", budou se prvky s touto hodnotou (při vzestupném řazení) objevovat vždy ve špičce stromu. Jakmile počítač ve stanovených intervalech (či programově naveden) přiběhne ke stromu a zatřese s ním, vypadne momentálně nejmenší prvek ze špičky (bude ze stromu vymazán). Počítač zjistí, kterému zařízení patří, a hned se s ním spojí. Tato fronta se anglicky jmenuje priority queue (prájrity kjó). V ní není třeba porovnávat hodnoty sousedních prvků, porovnávání (i výměny prvků) probíhají jen vertikálně.

4/ Souhrn

Závěrem si představme nejužívanější seskupování prvků v poli. Máme interval celých čísel 1..10, z nějž jsou náhodně generována čísla. Počet čísel, které se ve vygenerované řadě vyskytnou nejméně jednou, označme M . Tak např. v řadě 3,1,4,1,5,9 bude z možného počtu $N=10$ čísel $M=5$. Neutříděné pole: 3,1,4,5,9. V utříděných polích budou prvky uspořádány takto:

Bitová mapa 1 0 1 1 1 0 0 0 1 0

Sekvenční pole 1 3 4 5 9

Bins 1 3 5 : 9

Binární strom 3
1 4
5 9

Vysvětlení si žádají ještě bins. V uvedeném případě jde o rozdělení pole do dvou, jejichž prvky jsou uloženy tak, že mají vůči sobě určitý vztah. Uložení každého prvku se tedy musí vypočítat. Pokud vám to připomíná hashing, nejste nijak daleko. V uvedených bins se k sudému číslu dostaneme přes klíč k jeho lichému sousedu. Jsou-li čísla malá, mohou být uložena v jednom bajtu, větší na sousedících adresách apod.

Následující tabulka ukazuje časové proporce jednotlivých operací při uvedeném zadání s jednotlivými poli (b je počet bitů bitové mapy):

Pole	čas pro 1 operaci				Průměrný čas	Požadovaný prostor
	Inic.	Hled.	Vložení	Výpis		
Neutříd.	1	M	1	M log M	$O(M^2)$	M
Bit.mapa	N/b	1	1	N/b	$O(N/b)$	N/b
Sekvenční	1	log M	M	M	$O(M^2)$	M
Bins	M	1	1	M	$O(M)$	3M
B.strom	1	log M	log M	M	$O(M \log M)$	3M

Při volbě manipulace s daty v poli se můžete rozhodovat podle této tabulky, samozřejmě v závislosti na dalších okolnostech zpracování dat. Tabulka poskytuje základní přehled o čase a prostoru ve vztahu k počtu zpracovávaných prvků. Nebere tedy ohled např. na způsob zpracování prvků s rozdílnou délkou a jiné event. požadavky, kladené na funkci programu v nějaké konkrétní situaci. Ti z vás, kdož vlastní ZX Spectrum, jistě znají první čs. profesionální databanku pro tento počítač, zvanou Datalog. Autor zde musel brát ohled na malou paměť počítače i rozdílnou délku prvků. Proto nemohl volit třídění, vyžadující velký buffer, o němž předem přesně nevíme, kam až "naroste". Pro třídění v Datalogu chtěl-nechtěl musel zvolit bubble sort, ovšem o dost vylepšený (přibližující se shell sortu, což je, s přivřením obou očí, jistá kombinace principů binárního prohledávání a quick sortu). Příčinou volby tohoto třídění bylo i to, že Spectrum není součástí konfigurace, k níž by patřila disková jednotka, s jejíž pomocí, coby vedlejší paměti, by třídění mohlo probíhat. Tento druh třídění by zase byl zcela zvláštní kapitolou.

Věřím, že poskytnutý přehled o třídění typu insertion, bubble, quick, hash a heap, včetně tabulkových a klíčových magií, vám bude dobrým vodítkem při programování. I když nebudete vytvářet přímo databázi, bez nějakého, třeba jen minimálního třídění dat se prakticky neobejdete. Naopak je dobré použít je hned, jakmile okolnosti dějů nasvědčují tomu, že by se třídění měly otevřít dveře do programu. Otázku - které a jak? - si teď už budete umět vyřešit sami.

KAPITOLA 12

Rutinní drábky

V této kapitole jsou zařazeny rutiny, které svou povahou nespádají přímo ani do jedné z předešlých.

1/ Vyplnění úseku paměti bajty jedné hodnoty

je naprosto elementární operací, kterou při manipulacích s prvky pole budete používat zvláště při nulování jejich obsahu nebo zaplňování textovými mezerami. Pro jistotu připomínám jednu zrádnou skutečnost v instrukcích LDDR a LDIR - pokud je spustíte s nulovým čítačem bajtů (BC=0), bude se přenášeným bajtem postupně zaplňovat celý rozsah 65536 adres paměti do momentu zničení programu.

Délka: 16 bajtů

Počet cyklů: 21 pro každý bajt plus 67

Vstup: v HL spodní adresa bloku paměti, v BC počet adres bloku

v A bajt, jímž bude blok zaplněn

Výstup: BC adres zaplněno vstupní hodnotou reg.A

ABC=0, v HL posl.adr.bloku, DE=HL+1

Použité registry: HL,DE,BC,AF

```
MFILL LD E,A ;Uchova A do E
      LD A,B
      OR C
      RET Z ;Když BC=0, okamžitý návrat
      DEC BC
      LD A,B
      OR C
      RET Z ;Totéž pro BC=1 (nic se nikam nepřenáší)
      LD (HL),E ;Přenášený bajt na nejnižší adr.bloku
      LD D,H
      LD E,L
      INC DE ;Do DE adr.o 1 vyšší než v HL
      LDIR ;Proveď přenos
      RET ;Návrat
```

2/ Inteligentní přenos bloku dat

Tento název vznikl v době, do níž byly bloky dat přenášeny "neinteligentně", tedy tak, že docházelo k jejich přepsání, když se cílový blok překrýval zdrojový. Následující dvě rutiny takovou kolizi zcela vylučují. S tímto typem operace se setkáte při potřebě přenosu řetězců na jiné místo v paměti - typicky z/do bufferu nebo pole, při práci s počítačovou grafikou atd. Rutina před provedením přenosu zjišťuje, zda cílová adresa leží v poli zdrojových. Když ano, tak pro případ, že by došlo k chybnému přepisování části zdrojového pole, uzpůsobí směr přenosu. Opět je třeba dávat pozor na to, abychom nepřepsali něco, co by mělo zůstat netknuto: program, zásobník, potřebná data a programově vyhrazené úseky paměti.

Délka: 20 bajtů

Počet cyklů: 41 (DE=HL), BC*21+49 (HL men.), BC*21+89 (HL větší)

Vstup: v HL 1.adr.zdroje, v DE 1.adr.cíle

v BC počet přenášených bajtů (délka bloku)

Výstup: Když Z=1, bez pohybu, zadáno bylo DE=HL

Př: Z=0 je BC=0, přenos proveden, tehdy při

CY=0 přenos dolů, v HL a DE adresy o 1 větší než

konec jimi adresovaných bloků

CY=1 přenos nahoru, v HL a DE adresy o 1 menší než

začátky jimi adresovaných bloků

Použité registry: HL,DE,BC,F

```
IBT AND A ;Vynulování CY pro odečet
    SBC HL,DE ;adres obou bloků
    ADD HL,DE ;Obnov adr.v HL
    RET Z ;Když mezi adresami není rozdíl, návrat
    JR C,BACK ;Když cíl výš než zdroj, skok na BACK
    LDIR ;Jinak je cíl níž; proveď přenos
    RET ;a vrať se

BACK ADD HL,BC ;Urči jako 1.bajt přenosu
    DEC HL ;konec zdroj.bloku
    EX DE,HL ;a jeho adr.uschovej do DE
    ADD HL,BC ;Totéž pro cílový blok
    DEC HL
    EX DE,HL ;V DE cíl, v HL zdroj
    LDDR ;Proveď přenos,
    SCF ;signalizuj pomocí CY=1
    RET ;a vrať se
```

3/ Inteligentní přenos bloku dat s jejich rotací

Jde o rotaci obdobnou vám už důvěrně známým rotacím bitů. Zde jsou bajty zdroje rotovány (dolů nebo nahoru) k jednomu konci bloku, a co "přepadne", je přeneseno do cíle. Rozdíl mezi tímto a předchozím blokovým přenosem je tedy v tom, že počet bajtů zdroje se zmenšuje tak, jak roste počet bajtů cíle. Po skončení přenosu nezůstane na původním místě zdroje "nic", ev. tam bude jen to, co do něj bylo ze sebe sama přeneseno. Tato funkce se používá např. ve slovních procesorech, když přenášíme část textu z jednoho místa na druhé, přičemž požadujeme, aby z místa, z něž přenos provádíme, přenášený text zmizel. Využití najdeme i v insertion sortu. V komentáři rutiny je pro zmenšující se zdrojový blok použit termín subblok, pro prostor, který zaujímá postupně narůstající cílový blok spolu se zmenšujícím se zdrojovým, jen blok.

Délka: 68 bajtů

Vstup: v BC spodní adr.subbloku, v HL horní adr.subbloku

v DE 1. adr.budoucího bloku

Výstup: HL=0, v BC délka bloku-1

při CY=0 rotace dolů, v DE spodní adr.bloku

při CY=1 " nahoru, v DE horní adr.bloku

Použité registry: HL, DE, BC, AF

```

IBR      AND  A           ;Vynulování CY pro odečet
        SBC  HL,DE        ;horní adr.subbl.-1. adr.bud.bloku
        ADD  HL,DE        ;Obnov obsah HL
        JR   C,RDWD5      ;Když hor. adr.subbl.nad 1. adr.bud.bloku,
                        ;skok na rotaci dolů
        PUSH HL           ;Jinak rotace nahoru; uschovej obsah HL
        SBC  HL,BC        ;HL=délka bloku-1
        PUSH HL           ;Uchovej jako čítač LDDR
        ADD  HL,BC        ;Obnov pův. obsah HL
        SBC  HL,DE        ;V HL délka subbl. jako čítače posunů-1
        EX   (SP),HL      ;Ulož ji do zásobníku
        POP  BC           ;Do BC čítač pro LDDR
        POP  DE           ;Do DE adr. rotovaného bajtu
        INC  HL           ;V HL délka subbl.
        SCF           ;CY=1 pro signalizaci rotace nahoru
        JR   ROTLP        ;Proveď rotaci

RDWD5    AND  A           ;Vynulování CY, rotace dolů
        INC  HL           ;Následuje analogická příprava
        SBC  HL,BC        ;výchozích parametrů jako ve druhé
        PUSH HL           ;části IBR; na konci bude v BC délka
        EX   DE,HL        ;bloku-1 (čítač posunů LDIR), v HL délka
        ADD  HL,DE        ;subbloku, v DE adr. zdroje rotov. bajtu a
        EX   DE,HL        ;cíl prvního přenosu každé exekuce LDIR
        ADD  HL,BC
        EX   DE,HL
        SBC  HL,BC
        LD   D,B
        LD   E,C
        EX   (SP),HL
        POP  BC
        DEC  BC

ROTLP     PUSH HL          ;Ulož čítač rotací,
        PUSH DE          ;zdroj. adr. rotace (=cíl posunu)
        PUSH BC          ;a čítač posunů jednotlivých bajtů
        LD   H,D         ;DE do HL jako zdroj posunů +/- 1
        LD   L,E
        LD   A,(DE)      ;Do A bajt pro jeho rotaci
        JR   C,SHFTUP    ;Když signál "nahoru", skok na SHFTUP
        INC  HL          ;Jinak sniž start. adr. zdroje pro LDIR
        LDIR          ;a proveď přenos
        JR   STBYTE      ;Skok na STBYTE

SHFTUP    DEC  HL         ;Pro rotaci nahoru vše opačně
        LDDR

STBYTE    LD   (DE),A     ;Bajt z A do "volného prostoru"
        POP  BC          ;Obnov obsahy všech registrů
        POP  DE
        POP  HL
        DEC  HL          ;Zmenši čítač rotací o 1
        PUSH AF          ;Ulož CY
        LD   A,H         ;a zjisti, zda všechny rotace provedeny
        OR   L
        JR   Z,EXITS     ;Když AND, směr konec
        POP  AF          ;Když NE, obnov stav CY
        JR   ROTLP       ;a pokračuj od adr. ROTLP

EXITS     POP  AF
        RET              ;Posuň se v zásobníku na návratovou adr.
                        ;a vrať se na ni

```

4/ Tabulka či řetězec za CALL

Další rutina představuje způsob práce s řetězcem uloženým za instrukcí CALL, kterou rutinu voláme. Využívá se toho, že CALL ukládá do zásobníku adresu, která leží za CALlem.

```
CALL SAY
DEFB "Ahoj!"
DEFB 0
```

V uvedeném případě bude po exekuci CALL SAY v zásobníku adresa uložení prvního znaku řetězce Ahoj!. Řetězec je zakončen nulbajtem, aby rutina poznala, kde končí, a mohla také odvodit adresu návratu, jež bude za nulbajtem. Jednotlivé znaky řetězce jsou postupně odeslány do subrutiny OUTCH, která s nimi naloží, jak třeba. OUTCH zde uvedena není, její funkce a provázanost s celým programem závisí na tvůrci programu.

Délka: 16 bajtů

Vstup: 1. znak řetězce hned za instrukcí volání rutiny SAY

Výstup: Návrat za poslední znak řetězce

Použité registry: ---

```
TERM EQU 0 ;Zakončovací bajt řetězce (lze zvolit
;jakýkoli možný, zde je to nula)
SAY EX (SP),HL ;Adr.1. znaku řet.ze zásobníku do HL a
PUSH AF ;přev. obsah HL a AF do zásobníku

SAYLP LD A,(HL) ;Do reg.A znak řetězce
INC HL ;Jdi na adr.dalšího znaku
CP TERM ;Zjistí, jestli už to není zakonč.bajt
JR Z,EXIT ;Když AND, směr konec
CALL OUTCH ;Jinak pošli znak ke zpracování
JR SAYLP ;Pokračuj v odběru znaků řetězce

EXIT POP AF ;Obnov obsah AF
EX (SP),HL ;a HL, do zásob.adr.návratu za zakonč.
RET ;bajt a návrat na ni
```

Za CALL můžeme uložit nejen řetězec k jeho výpisu na obrazovku, ale třeba i tabulku, v níž budeme hledat nějaký bajt, porovnávat jej s jiným apod. V tom případě může finta s CALL vypadat takto:

```
CALL ADR
DEFB bajty tabulky
ADR POP reg. (v reg. bude adr.1.bajtu tabulky)
... pokračování programu hledáním apod.
```

Navíc v tomto případě můžeme určit velikost tabulky odečtem adresy uložení 1.bajtu tabulky a adresy ADR. To se hodí, když nechceme použít zakončovací bajt tabulky a když ji v průběhu tvorby programu často měníme. Zároveň tak v nějakém registru můžeme pak mít přímou adresu nalezeného bajtu v tabulce.

5/ DEFB za CALL jako relativní offset volání

Rutina je podobná předchozí. Bajt, který odebere za instrukcí jejího volání, však reprezentuje relativní offset v intervalu -128...+127, podobně jako u relativního skoku JR. Proto můžeme takto provedený CALL nazvat relativním voláním. Konečný návrat po provedení CALL offset půjde na adresu za uloženým offset bajtem, čili:

```
CALL RLTV
DEFB offset bajt
... ;adresa návratu
```

Délka: 31 bajtů

Počet cyklů: 229

Vstup a výstup zřejmý z úvodu

Použité registry: ---

```
RLTV PUSH HL ;Uchova obsahu registrů
PUSH DE
PUSH AF
LD HL,6 ;Dostaň se přes právě uložené registry
ADD HL,SP ;na návratovou adresu (o 6 adres výš)
LD E,(HL) ;a přenes ji do DE (na ní leží offset.
INC HL ;bajt)
LD D,(HL)
LD A,(DE) ;Odeber z ní offset.bajt do reg.A
INC DE ;Adresa návratu z CALL offset půjde za
LD (HL),D ;adr ulož.offset.bajtu; zapiš ji do
DEC HL ;zásobníku (je teď adresován i reg.HL)
LD (HL),E
LD L,A ;Offset do L
RLA ;Bit znaménka do CY
```



```
SBC A,A      ;Při CY=0 i A=0; při CY=1 pak A=FFH
LD H,A      ;Při CY=1 16-bit.negace (dvojk.komplem.)
ADD HL,DE    ;adresy relat.volání v HL
POP AF       ;Obnova obsahu registrů
POP DE
EX (SP),HL   ;Přev.HL do HL, adr.rel.volání do zásob.
RET          ;Skoč na tuto adresu (v zás.je teď adr.
              ;návratu za adr.uložení offset.bajtu)
```

6/ DEFB za CALL jako klíč k tabulkové adrese volání

Rutina opět podobná předchozí, ale DEFB obsahuje pořadové číslo uložení volací adresy v tabulce. Po jejím nalezení je převedeno programové řízení na tuto adresu s konečným návratem za adresu, na níž leží DEFB. Pozor - rutina nekontroluje, zda je obsah DEFB větší, než pořadí poslední adresy v tabulce.

Délka: 31 bajtů
Počet cyklů: 229
Vstup a výstup zřejmý z úvodu
Použité registry: ---

```
ISTAB DEFS 2      ;V ISTAB 1.adr.tabulky
ISCALL PUSH HL     ;Uložení obsahu registrů
      PUSH DE
      PUSH AF
      LD HL,6
      ADD HL,SP
      LD E,(HL)
      INC HL
      LD D,(HL)
      LD A,(DE)
      INC DE
      LD (HL),D
      DEC HL
      LD (HL),E    ;Až potud jako v minulé rutině
      LD E,A       ;Klíč do DE
      LD D,0
      LD HL,(ISTAB) ;Do HL 1.adr.tabulky
      ADD HL,DE     ;Dvojit. přičtení klíče (adresy jsou
      ADD HL,DE     ;samočřejmě dvoubajtové)
      LD E,(HL)     ;Odběr hledané adresy do DE
      INC HL
      LD D,(HL)
      EX DE,HL      ;a její přepis do HL
      POP AF
      POP DE
      EX (SP),HL    ;a z HL do zásobníku
      RET           ;Skoč na tuto adresu
```

7/ Určení adresy podle vzdálenosti od místa volání

Minirutinka vypočte adresu jakéhokoli místa v paměti, určeného vzdáleností (offsetem) mezi ním a první adresou za instrukcí volání rutinky. Provádí hezké hrátky se zásobníkem, které vás mohou inspirovat i k jinému využití.

Délka: 8 bajtů
Počet cyklů: 75
Vstup: v HL offset
Výstup: v HL offsetová adresa
Použité registry: HL,F

```
FORWARD EX DE,HL   ;Offset do DE, pův.obsah DE do HL
      EX (SP),HL   ;a do zásob., návrat.adr.do HL
      EX DE,HL     ;a do DE, v HL offset
      ADD HL,DE     ;Sečti offset a návrat.adr.
      EX DE,HL     ;Opačným postupem
      EX (SP),HL   ;dej vše na místo
      EX DE,HL     ;a s offset.adresou v HL
      RET          ;se vrať
```

8/ Změna návratové adresy za sled definovaných bajtů

Další miniatura, která v zásobníku změní adresu návratu za řadu DEFB umístěných za instrukcí volání z hlavní rutiny. Děj je dvouступňový:

```
hlavní ...      ;Volání nějaké subrut.z hlavní rutiny
      CALL něco
      DEFB       ;Začátek definovaných bajtů
      DEFB
      DEFB
návrát ...      ;Konec def.bajtů a požadovaný
              ;návrat ze subr.něco za posl.def.bajt
```

něco CALL ALRET ; Volání ALRET pro změnu návratu do
RET ; hlavní rutiny na adr.za posl.def.bajt
; Návrat na změněnou adresu "návrat"

Pole definovaných bajtů obvykle obsahuje parametry přenášené do subrutiny, která je má zpracovat. Rutinu ALRET můžeme použít v programu pro univerzální určování návratových adres za poslední z řady definovaných bajtů umístěných za instrukcí CALL ve "vyšší patře" programové struktury. Uvedený způsob je mírně "ujetý", navíc mu v případě vstupu přerušení hrozí přepsání potřebných bajtů v zásobníku (obslužná rutina přerušení je přepíše obligátní vstupní úschovou obsahů registrů) - na což pozor obecně. Rutinu jsem zařadil, protože je dobrou výukovou miniaturoou.

Délka: 10 bajtů
Počet cyklů: 91
Vstup: v DE počet DEFB
Výstup: návratová adresa zvýšená o obsah DE, v HL adr.1.DEFB,
v DE vstupní obsah HL
Použité registry: DE,HL,F

```
ALRET INC SP ;Jdi nad adr.návratu do subrutiny
      INC SP
      EX (SP),HL ;Do HL adr.návratu do hlavní rutiny
      EX DE,HL ;Přenes ji do DE, do HL počet DEFB
      ADD HL,DE ;V HL bude adr.za posl.DEFB v hl.rutině
      EX (SP),HL ;Tuto novou adr.návratu dej do zásob.
      EX DE,HL ;Do DE pův.obsah HL, v DE adr.1.DEFB
      DEC SP ;Jdi zpět pro adr.návratu do subrutiny
      DEC SP
      RET ;a vrať se na ni
```

9/ 64 znaků na šířce 32 obrazových bajtů

Málokdo dnes pracuje s editací 32 znaků na řádce u mikropočítačů, jejichž základní systém jinou možnost neposkytuje. Naprostá většina editorů takových mikropočítačů je dnes upravena pro projekci 42, 51 nebo 64 znaků/ř. Další rutiny ukazují "projekční jádro" editoru se 64 znaky na řádce. Grafická tvorba znaků vychází z adresace obrazové paměti, jak je uvedena ve 2.kapitole (ZX Spectrum, SORD M5 a ev. jiné podobné). S ohledem na dosažení nejvyšší možné rychlosti přenosu znaků na obrazovku je jejich grafická síť uložena v tabulce přesně v jejich výsledné podobě. Grafika znaků tedy není komprimována, resp. odvozována nějakým výpočtem.

Každý znak obsahuje síť 4 obrazových bodů na šířku a 8 na výšku. T.zn., že do jedné "klasické" znakové pozice 8 x 8 bodů se vedle sebe vejdou dva znaky. V tabulce je v každých osmi po sobě jdoucích bajtech levá polovina (tj.vyšší 4 bity) věnována znaku s kódem v intervalu 0..127, zatímco v pravé polovině (nižších 4 bitech) je vždy znak s kódem o 128 vyšší (interval 128..255). Dejme tomu, že tabulka znakových bajtů začíná na adrese 33000. Od ní leží 8 bajtů znaků s kódy 0 a 128, na dalších 8 adresách jsou znaky s kódy 1 a 129...až do 127 a 255. Tabulka obsahuje 256 znaků * (4 * 8 bitů)=8192 grafických bitů, tj.1024 bajtů, což je 1K paměti. Pokud se ptáte, jak může mít nějaký znak kód třeba 1, je to dáno vnitřními funkcemi editoru, který, jsa k tomu donucen speciálními řídícími kódy, může interpretovat spodní kódy (které jsou řídícími ASCII kódy) jako znaky. Podobně bývá rozšířen rejstřík znaků mnoha tiskáren.

Pro názornost - grafika znaků s kódy 110 (ASCII n) a 238 (dáme tam naše ň) bude uložena takto (svislé čárky jsou přidány jen pro orientaci):

adresy:

pro n: 33000+8*110=33880		- -	pro ň: 33000+8*(238-128)
33881		-	a dále jako u n;
33882		-	(odečet 128 se týká všech
33883		- - - -	kódů vyšších než 127)
33884		- - - -	
33885		- - - -	
33886		- - - -	
33887		- - - -	

Najít adresu uložení znaku teď umíme. Čekají nás ale některé další problémy. Musíme zajistit:

a/ přenos grafické sítě 4 * 8 bitů z místa jejího uložení v tabulce do obrazové paměti s ohledem na způsob jejího adresování
 b/ aby grafická síť z vyšších 4 bitů byla z tabulky přenositelná nejen do vyšších 4 bitů obrazových bajtů, ale i do nižších; totéž pro nižší 4 bity sítě
 c/ aby bylo v každé chvíli známo, do které půlky (4 bitů) obrazových bajtů i na jaký sloupec a řádku na obrazovce má být právě přenášený znak zapsán
 d/ aby při přenosu půlek bajtů znaků do obrazových bajtů zůstaly druhé půlky těchto bajtů nedotčeny, tedy aby již zobrazený znak nebyl zničen zápisem svého souseda do druhé půlky
 e/ rychlý přenos mezi tabulkou a obrazovkou (pokud možno bez brzdicího bufferování)

Tyto požadavky řeší následující rutina slovního procesoru. Je natolik univerzální, že nic nebrání její aplikaci v jakémkoli editoru. Základním východiskem nezbytné orientace v tom, kam a jak má být znak přenesen, je kombinace dvou proměnných. První je průběžně v reg.HL, jehož obsah informuje o pozici znaku na obrazovce. V každé její třetině k tomu používá jen obsah reg.L, jehož obsah se mění v rozsahu 0..255. To je 8 znakových řádek po 32 bajtech. Když tento interval "přeteče", změníme i obsah reg.H, který je pro pozičně nejvyšší 1. třetinu 40H a pro každou ze zbývajících dvou o 08H větší (tedy 48H a 50H). U reg.L je využito faktu, že v každé třetině horní grafická linka znaků narůstá zleva doprava postupně všemi 8 znakovými řádkami. Tak po první znakové řádce, jejíž horní linka začíná na 0 (nižší bajt obrazové adresy) a končí na 31, horní linka další znakové řádky pokračuje 32 atd... Poslední bajt horní linky poslední znakové řádky třetiny je na adrese, jejíž nižší bajt má hodnotu 255. Další třetina začíná zase nulou. Tak můžeme říci, že horní linka je pro nás jako šňůra na prádlo, na níž věšíme jednotlivé znaky. Každý obrazový bajt opticky sousedící linky pod touto šňůrou je v obrazové paměti o 256 adres dál. Což je z pohledu programátora skvělé, neboť přičíst k adrese 256 znamená jen zvýšit obsah jejího vyššího bajtu o 1. Když budeme chtít na obrazovce vlevo nahoře namalovat černý obdélníček 4*8 bitů, můžeme to provést touto smyčkou:

```
LD A,11110000B
LD HL,4000H
LD B,8
OBDEL LD (HL),A
INC H
DJNZ OBDEL
```

Jednoduché, že? Mohli bychom i vynechat inicializaci reg.A a do smyčky místo LD (HL),A dát LD (HL),11110000B. První z obou instrukcí však trvá kratší dobu, proto je pro použití ve smyčce o moc vhodnější.

Druhým důležitým informačním prvkem je adresa uložení znaku do uživatelské paměti slovního procesoru. V ní je každé řádce vyhrazeno celých 64 adres. Momentální adresa je průběžně v reg.DE. Důležitý je obsah reg.E. Podle toho, zda je lichý nebo sudý, budeme znak rozvěšovat do levých nebo pravých půlek obrazových bajtů, adresovaných registrem HL. A jak určitě víte, bajt je lichý, když jeho bit 0 je ve stavu log.1 a naopak. Jádro řešení projekce znaku je tedy v obsahu registrů L a E. I proto musí uživatelský text v paměti začínat na liché adrese. V komentáři rutiny bude adresování označeno symboly TV pro HL (umístění bajtu horní graf. linky znaku v obraz. paměti) a TXT pro DE (umístění kódu znaku v paměti vyhrazené uživatelskému textu).

Délka: 118 bajtů

Vstup: v HL moment. adr. horní graf. linky znakové řádky (adr. TV)
 v DE adr. uložení kódu znaku v uživat. paměti (adr. TXT)
 Kód znaku, rutinou zpracováváný, již leží v (DE)
 Tabulka graf. sestavy znaků začíná na adr. TABZN

Výstup: když byl znak ukládán do levé půlky obrazových bajtů, pak HL beze změny, jinak HL o 1 větší
 Když došlo k posunu do další třetiny obrazovky, obsah reg.H se zvýšil o 08H
 Při doplnění do pravého kraje nad dialogovou řádkou výstup z rutiny
 Obsah DE beze změny

Použité registry: HL, DE, AF

```
ZNAK  PUSH BC           ;Uchova obsahu registrů
      PUSH DE
      PUSH IX
      LD IX,TABZN       ;Do IX 1. adr. znak. tabulky
      LD B,0            ;Vynulování B
      EX DE,HL          ;Přenos adr. TV do DE a adr. TXT do HL
      LD C,(HL)         ;Kód znaku z TXT do reg.C
```

```

LD A,C      ;a do reg.A
RES 7,C      ;Když kód větší než 127, odečti 128
SLA C        ;Následuje vynásobení kódu osmi (2**3)
RL B         ;pro určení vzdálenosti uložení jeho
SLA C        ;1.graf.bajtu od začátku znak.tabulky
RL B         ;Vzdálenost bude v BC
SLA C        ;
RL B         ;
ADD IX,BC     ;Zjištěnou vzdálenost přičti k 1.adr.tab
BIT 7,A       ;Je kód větš.než 127 (levá/pravá půlka)?
JR Z,ZN1      ;Když je menší, bude v pravých půlkách
PUSH HL       ;Když je větší, uschovej adr.TXT

```

; Rutina přenáší na obrazovku graf.půlky znaku jen z vyšších půlek zdroje. Protože grafika znaků s kódy vyššími než 127 je v tabulce uložena v pravých půlkách, musíme je do levých napřed přesunout

```

ZN2 LD HL,PRABUF ;Do HL adr.meziuložení graf.půlek
LD B,8         ;V reg.B čítač smyčky
LD A,(IX+0)    ;Graf.bajt z tabulky do reg.A
LD (HL),A      ;a jeho uložení do oblasti PRABUF
RLD           ;V (HL) rotace nižší půlky do vyšší
INC HL         ;Jdi na další adr.v obl.PRABUF
INC IX        ;a na další graf.bajt v tabulce
DJNZ ZN2      ;Opakuj pro všech 8 graf.bajtů
POP HL        ;V HL obnov adr.TXT
LD IX,PRABUF  ;Do IX adr.1.zrotovaného graf.bajtu

ZN1 EX DE,HL   ;Do DE adr.TXT, do HL adr.TV
BIT 0,E        ;Půjde znak do levé/pravé půlky obr.baj-
PUSH HL        ;tů?; uschovej adr.TV (horní graf.linka)
LD B,8         ;V B čítač smyčky přenosu do obr.paměti
JR Z,SUDY      ;Když znak do pravé půlky obr.bajtu,skok

LICHY LD A,(IX+0) ;Grafika znaku do vyšší půlky reg.A,
RRD           ;přirotování obsahu nižší půlky obr.baj-
LD (HL),A      ;tu a uložení celku zpět na obraz.adresu
INC H          ;Přičtení 256 k předchozí obraz.adrese
INC IX        ;Na adresu další graf.půlky
DJNZ LICHY    ;Zopakuj pro všech 8 obraz.bajtů
POP HL        ;Obnov adr.TV v HL,
POP IX        ;obsah IX,
POP DE        ;adr.TXT v DE
POP BC        ;a obsah BC
RET           ;Návrat

SUDY LD A,(HL)  ;Do reg.A obsah obraz.bajtu
LD C,(IX+0)    ;Graf.bajt znaku do reg.C
LD (HL),C      ;a do obraz.bajtu
RLD           ;Narotování vyšší půlky (HL) do reg.A
LD (HL),A      ;Uložení výsledné sestavy do obraz.bajtu
INC H          ;Dále jako u části LICHY
INC IX        ;
DJNZ SUDY     ;
POP HL        ;
POP IX        ;
POP DE        ;
POP BC        ;

;Protože jsme zapisovali znak do pravých půlek obraz.bajtu,
;musíme pro náš slovní procesor posunout adr.TV o 1 doprava, aby
;na ni mohl být zapsán ev.další znak (jako při psaní na psacím
;stroji). Před tím je ale třeba zjistit některé okolnosti. Pře-
;devším - zda nejsme na konci prostoru pro projekci uživatelské-
;ho textu na obrazovce, tj.na konci poslední text.řádky, za níž
;následuje řádka dialogová (značky tabulace, kolmý průmět kurzo-
;ru apod.). Do ní nesmíme "vjet", v tom momentě se musí text na
;celé obrazovce posunout o 1 znak.řádku nahoru. Tento případ mů-
;že nastat jen tehdy, když jsme v poslední třetině obrazovky.
;Jsme-li v 1.nebo 2.třetině, pak musíme ošetřit přechod z 1.do
;2. a z 2.do 3. třetiny (to je signalizováno zvýšením obsahu
;reg.L z FFH na 0)

```

```

LD A,H        ;Jsme za koncem 2.třetiny obrazovky (je
CP #49        ;obsah H větší než 48H)?
JR C,SUDY2    ;Když NE, skok
CP 4*32-#FF   ;Jsme na konci řádky před dialog.řádkou?
JR NZ,SUDY2   ;Když ještě NE, skok
JP TVHORE     ;Když ANO, posuň text na obrazovce
              ;o 1 znak.řádku nahoru
SUDY2 INC L    ;Zvyš adr.TV o 1 a vrať se, když jsme
RET NZ        ;nepřešli do další třetiny (z FFH na 0)
LD A,H        ;Jinak uprav obsah reg.H přičtením 08H
ADD A,8        ;(adresace další třetiny)
LD H,A        ;a vrať se
RET

```

Po proběhnutí touto rutinou pak slovní procesor musí o 1 zvýšit adresu TXT v reg.DE pro další znak, zvýšit odečet sloupců o 1 (v intervalu 1..64), ev. při přechodu na další řádku sloupec nastavit na 1 a zvýšit řádku o 1, a čekat na stisk dalšího tlačítka (buď to bude kód dalšího znaku nebo nějaká jiná akce) atd. Rutina TVHORE zde uvedena není, je jednou z mnoha, které vládnou funkcemi slovního procesoru. Budete-li chtít rutinu upravit na jinou adresaci obrazové paměti, nedá vám to moc práce, protože rutina pracuje s vyšší i nižší částí obrazové adresy zvlášť. Rychlost projekce znaku na obrazovku je velmi vysoká, při praktickém používání není vůbec patrný minimální časový skluz způsobený meziukládáním grafiky vyšších kódů do oblasti PRABUF, či testy závěru části SUDY.

Protože editace bez kurzoru je nemyslitelná, ukážeme si ještě, jak je tvořen v našem slovní procesoru. Při psaní musí kurzor být vždy tam, kde se má objevit další znak. Zároveň s jeho horizontálním pohybem se pohybuje jeho kolmý průmět v dialogové řádce (těch může být i více než jedna; záleží na tom, kolik informací v průběhu zápisu chce programátor uživateli poskytovat). Základním problémem v práci s kurzorem je, aby při svém posunu kamkoli byl na své minulé pozici vymazán bez ztráty toho, co "přikrývá", a posunul se tam, kde má být, aniž by vymazal to, co na jeho nové pozici bylo před jeho příchodem. V podstatě jde o střídavou inverzi všech grafických bodů znakové sítě 4*8 bitů.

Délka: 45 bajtů

Vstup: v HL obraz.adr.horní graf.linky znakové řádky

v DE adr.TXT uložení znaku v uživat.paměti

Výstup: kromě AF všechny registry nezměněny

Použité registry: HL,DE,AF

```

KURZOR CALL KURZ01      ;Vstup při horizontálním posunu kurzoru
KURZ00 PUSH HL          ;Uchovej moment.adr.TV
LD HL,32*4+#5000        ;Do HL adr.levého kraje dialog.řádky
LD A,(SLOUP)            ;Číslo sloupce do reg.A
DEC A                   ;Zmenši je o 1 (interv.bude 0..63)
SRL A                   ;a vyděl dvěma
ADD A,L                 ;K výsledku přičti reg.L (pozice průmě-
LD L,A                  ;tu kurzoru do dialog.řádky); ulož do L
CALL KURZ01             ;Zavolej inverzi
POP HL                  ;V HL obnov moment.adr.TV
RET                     ;a vrať se

KURZ01 BIT 0,E           ;Liché/sudé (levé/pravé) půlky obraz.
PUSH HL                 ;bajtů?; ulož adr.TV do zásobníku
LD B,8                  ;Čítač smyček inverze
JR Z,KSUDY              ;Když sudé,tj.pravé půlky, skok na KSUDY
; Jinak liché, tj.levé půlky
KLICHY LD A,(HL)         ;Do reg.A obsah obr.bajtu
XOR 11110000B           ;Inverze jeho pravé půlky (vyš.4 bitů)
LD (HL),A               ;Po inverzi zpět do obraz.bajtu
INC H                   ;Další znak.bajt o 256 adres dál
DJNZ KLICHY             ;Opakuj pro všech 8 bajtů znaku
POP HL                  ;Obnov adr.TV v HL
RET                     ;a vrať se

KSUDY LD A,(HL)          ;U nižších 4 bitů proved inverzi obdobně
XOR 00001111B          ;jako u vyšších v KLICHY
LD (HL),A
INC H
DJNZ KSUDY
POP HL
RET

```

Pokud máte dojem, že je v rutině příliš mnoho "pušů a popů" reg.HL, je to proto, že slovní procesor někdy volá jen samotné části KURZ00 nebo KURZ01. Když zavoláme KURZOR na pozici, na níž je, bude z ní vymazán, resp. bude provedena inverze celé sítě znaku 4*8 bitů. A naopak – když jej umístíme tam, kde není, celá síť bude invertována, a my nabydeme optického dojmu přítomnosti kurzoru. Při postupném zápisu znaků síť právě zapisovaného znaku kurzor na své pozici přepíše, je tedy nutno volat KURZOR jen pro inverzi prostoru pro další znak (následující znakovou pozici). Když však "běháme" kurzorem v textu, musíme při posunech horizontálních celý KURZOR volat dvakrát, při vertikálních jen část KURZ01 (jinak by se střídavě vymazával a obnovoval průmět kurzoru v dialogové řádce).

10/ Přenos textových řetězců z tabulky

Přestože rutin s touto funkcí učebnice obsahuje dost, nepoužívám při tvorbě programů ani jednu z nich, i když v principu mají všechny mnoho společného. Svou rutinou promítám

různá hlášení a informace určené uživateli programu. Textové řetězce jsou v tabulce odděleny nulbajty. Pro plynulost výkladu uvedu rutinu v návaznosti na předchozí rutiny slovního procesoru, který v určitých fázích používá dvě spodní znakové řádky obrazovky pro komunikaci s uživatelem. Tyto řádky si nazveme menu. Znaky v nich budeme rozvěšovat opět na "prádelní šňůru" - horní grafickou linku znakových řádek.

Délka: 65 bajtů a 2 pro data (MEZERY)

Vstup: v BC obraz.adr.projekce prvního znaku řetězce
v (IX-6) pořadové číslo nulbajtu (i hledaného řetězce)
první řetězec v tabulce má číslo 1, max.počet řetězců 255
formát tabulky - 0,1,řet.,0,2,řet.,0,...,0,N-tý řet.,0
délka řetězce v intervalu 1..255 bajtů

Výstup: (IX-6) beze změny, B=0, v C negat.délka řetězce

Použité registry: IX,BC,AF

```

TEXTMEN PUSH HL      ;Uchova obsahu reg.HL,
        PUSH DE      ;DE a
        PUSH BC      ;BC (obsahuje obr.adr.projekce řetězce)
        CALL MAZMEN   ;Vymaž v menu, co tam dosud bylo
        POP HL        ;Do HL obr.adr.projekce řetězce
        CALL OBRMEN   ;Vyhledej a promítni řetězec
        POP DE        ;Obnov obsahy uschovaných registrů
        POP HL
        RET           ;Konečný návrat

MAZMEN LD HL,XXXX     ;Vymazání menu; v HL 1.obr.adr.menu; do B
        LD B,NN        ;počet obr.bajtů horní graf.linky v menu
MAZME1 LD DE,MEZERY    ;Do DE adr.dvou mezer v program.obl.dat
        CALL ZNAK      ;Promítni NN-krát páry mezer do celého
        INC DE         ;menu (každá mezera má síť 4*8 bitů)
        DJNZ MAZME1    ;Opakuj pro všech NN párů mezer
        RET            ;Návrat ze subrutiny

OBRMEN PUSH HL        ;Uchovej obsah HL (1.obr.adr.řetězce)
        LD HL,NAPISY   ;Do HL 1.adr.tabulky řetězců (1.nulbajt)
        LD BC,0        ;Vynulování čítače prohledávání tabulky
        XOR A          ;Vynulování reg.A (hledáme nulbajty)
OBRME1 CP IR          ;Po nalezení nulbajtu
        DEC (IX-6)     ;zmenši pořadové číslo hledaného řetězce
        JR NZ,OBRME1   ;Když to ještě není on, hledej dál
        PUSH HL        ;Uchovej 1.adr.řetězce v HL
        LD BC,0        ;Vynuluj čítač pro budoucí určení délky
        CP IR          ;řetězce a hledej nulbajt za ním
        LD A,C         ;Obsah C (menší o délku řetězce+1) do A,
        CPL            ;proto proved jedničkovou komplementaci
        LD B,A         ;Délku řetězce dej do B
        POP HL         ;1.adresu řetězce v tabulce
        EX DE,HL       ;přemísti z HL do DE
        POP HL         ;Do HL obr.adr.projekce řetězce
OBRME2 CALL ZNAK      ;Promítni znak řetězce na obrazovku
        INC DE         ;Jdi na adresu dalšího znaku v tabulce
        DJNZ OBRME2    ;Opakuj pro všechny znaky řetězce
        RET            ;Návrat ze subrutiny

MEZERY DEFB 32,32     ;ASCII kódy mezer (adr.MEZERY je lichá)
NAPISY DEFB 0         ;Nulbajt 1.řetězce
        DEFM "AND/NE (A/N)?"
        DEFB 0        ;2.řetězec
        DEFM "LOAD/MERGE (L/M)?"
        DEFB 0        ;3.řetězec
        DEFM "Všechno správně?"
        DEFB 0        ;Konečný nulbajt celé tabulky

```

Nevýhody rutiny: Počet znaků řetězce omezený do 255 tu nevadí, informace nemá být upovídána. Je-li požadována větší délka, lze řetězec rozdělit do dvou či více. Kombinace řetězců jsou jinak běžné (viz 3. a 1.řetězec). Omezení na maximálních 255 řetězců u užitečného programu mikropočítače rovněž nevadí. Lze rozšířit na 65535 řetězců přidáním druhého bajtu k (IX-6), třeba (IX-5). Není ošetřen případ "hledání za tabulkou", řetězec nulového či delšího než 255 bajtů apod. Pozorný programátor takové testy ryze vnitřních funkcí svého programu nepotřebuje.

Výhody: Při tvorbě programu se ukládá jen obrazová adresa umístění řetězce na obrazovce (do reg.BC) a jeho pořadové číslo (do (IX-6)). O ostatní není třeba se starat, rutina vše vykoná sama. V průběhu tvorby lze bez obav tabulku doplňovat novými řetězci různé délky. Lze volat i samostatné části rutiny pro mazání či projekci textu řetězců. Projekce není omezena jen na prostor menu. Doba provedení operace je velmi rychlá. Orientace v listingu tabulky blesková.

KAPITOLA 13

Interfacing

Problematika přenosu dat mezi počítačem a perifériemi by si žádala zvláštní publikaci, která by osvětlila samotnou podstatu hardwarových funkcí. Jinak člověka nad softwarem pro přenos dat napadá řetěz otázek, na něž bez "první pomoci" bude hledat odpověď jen obtížně. Přesto by takový software neměl v této učebnici chybět.

1/ Z80 PIO

je zkratkou slov Parallel Input/Output. Stručně - integrovaný obvod pro paralelní přenos dat. Vyvinul jej výrobce Z80, tedy firma Zilog. Při tomto přenosu je možno po osmi linkách datové sběrnice současně (paralelně) přenášet všech 8 bitů datového bajtu najednou. Před zahájením přenosu musíme obvod inicializovat. Co probíhá dál, nám ukáže rutina pro I/O přenos tzv. nebufferovaných dat. Buffery jsou zde nejmenší možné - jednobajtové. Jeden pro vstup a jeden pro výstup. Rutina obsahuje tyto subrutiny:

INCH - přečte znak ze vstupního bufferu, když je indikátor RECD (Data Ready - data připravena) v nenulovém stavu. V ten moment indikátor vynuluje a přečtený znak přemístí do reg.A.

INST - zjišťuje, zda je vstupní buffer prázdný, resp. uděluje povolení subrutině INCH k provedení akce signalizací v CY.

OUTCH - zapiše znak do výstupního bufferu, když je prázdný a nastaví indikátor TRNDF pro signalizaci možnosti odběru dat. Když není očekáváno žádné výstupní přerušení (bylo zablokováno), data jsou hned odeslána do PIO.

OUTST - na základě stavu indikátoru TRNDF zjišťuje, zda je výstupní buffer plný (obsazen).

INIT - inicializuje řídicí registry a vektor přerušení obvodu PIO, vektorové adresy přerušení softwaru a jeho indikátory, které jsou řídicími prvky přenosu dat mezi hlavním programem a následujícími dvěma obslužnými rutinami přerušení:

RDHDLR - odpovídá na přerušení přečtením znaku z PIO a jeho přenesením do vstupního bufferu. Po akci nastaví indikátor RECD (Data Ready).

WRHDLR - reaguje na přerušení přenesením znaku z výstupního bufferu do PIO. Před tím musí zjistit, zda jsou v bufferu data k dispozici. Když ne, zablokuje vstupní přerušení (port B PIO). Jinak by mohlo dojít k věčné smyčce. Když jsou data k dispozici, provede přenos a vynuluje indikátor TRNDF. Když je zablokováno přerušení v momentě, kdy jsou data k dispozici, bez nějaké pomocné indikace bychom se nedozvěděli, zda je (minule, když byl buffer prázdný, bylo) výstupní zařízení připraveno k akci. Tuto signalizaci na sebe bere indikátor OIE. Když je nulový, je možno přenos provést i přes zablokováné přerušení. V tom případě OUTCH nečeká na přerušení (je zablokováno) a hned provede přenos. Po přenosu je OIE nastaven (má nenulovou hodnotu) do stavu očekávání příchodu přerušení. V tom případě OUTCH čeká na výstupní přerušení také. Problémy tohoto druhu jsou jen na výstupu. Když vstupní zařízení žádají o obsluhu, mají data k přenosu vždy připravena.

Před popisem rutiny se podíváme, jak s ní může uživatel spolupracovat. Prezentován je obousměrný přenos jednoduchý (smyčka LOOP) a asynchronní (ASYNLDP). Rutina tu hraje roli "odrazové stěny":

ESCAPE EQU	#1B	;ASCII znak pro Escape
TESTCH EQU	"A"	;Testovací znak asynchronního přenosu
LOOP	CALL INIT	;Inicializuj PIO a systém přerušení
	CALL INCH	;Přečti znak ze vstupního bufferu
	PUSH AF	
	CALL OUTCH	;Zapiš znak do výstupního bufferu
	POP AF	
	CP ESCAPE	;Je to Escape?
	JR NZ, LOOP	;Když NE, pokračuj ve smyčce přenosu
ASYNLDP	CALL OUTST	;Je výstupní buffer plný (BUSY)?
	JR C, ASYNLP	;Když AND, zpět na ASYNLP
	LD A, TESTCH	;Když NE, dej do reg.A testovací znak
	CALL OUTCH	;a pošli jej do výst. bufferu
	CALL INST	;Je něco na vstupu?
	JR NC, ASYNLP	;Když NE, vrať se na ASYNLP
	CALL INCH	;Jinak přečti znak
	CP ESCAPE	;Je to Escape?
	JR Z, ASDONE	;Když AND, "zmiz"
	CALL OUTCH	;Jinak pošli znak na výstup
	JP ASYNLP	;a pokračuj dál
ASDONE	JP LOOP	;Asynchronní přenos skončil

Kód Escape (čte se iskejp) najdete v manuálech tiskáren ve zkrácené podobě ESC. Při přenosu dat na tiskárnu uvozuje sled řídících kódů, které nejsou tisknuty jako znaky, ale volí se jimi některá z funkcí tiskárny či módů tisku. Ve výše uvedené rutině ESC signalizuje skončení přenosu z periférie do počítače.

Délka: 166 bajtů a 5 pro data

Počet cyklů: INCH 72, INST 27, OUTCH 150, OUTST 27, INIT 377,
RDHDLR 82, WRHDLR 178

Vstup: OUTCH - přenášený znak v reg.A

Výstup: INCH - znak v reg.A,

INST - při CY=1 vstupní buffer prázdný

při CY=0 v bufferu znak k odběru

OUTST - při CY=1 výstupní buffer plný

při CY=0 není plný

Použité registry: INIT - I, HL, BC, AF, v ostatních jen AF

;PIO programováno takto: port A jako vstupní, port B výstupní.

;Adesování portů a vektorů přerušení:

```

PIOAD EQU #90      ;Data portu A
PIOAC EQU #91      ;Řízení portu A
PIOBD EQU #92      ;Data portu B
PIOBC EQU #93      ;Řízení portu B
INTRPV EQU #8000   ;Základní adresa vektorů přerušení
PIOIVA EQU INTRPV  ;Vektorová adresa přerušení portu A
PIOIVB EQU INTRPV+2 ;" " " " portu B

INCH CALL INST      ;Zjistí stav na vstupu
JR NC, INCH         ;Když žádný znak k dispozici, opakuj
DI                 ;Jinak zablokuj maskovatelné přerušení
SUB A              ;Vynuluj obsah reg.A
LD (RECF), A       ;Signalizuj prázdný buffer
LD A, (RECF)       ;Do reg.A přenes znak ze vstup.bufferu
EI                 ;Uvolni přerušení
RET                ;a vrať se

INST LD A, (RECF)   ;Do A indikátor Data Ready
RRA                ;Stav do CY (při CY=1 znak k dispozici)
RET                ;Návrat se stavem v CY

OUTCH PUSH AF       ;Uschovej znak pro odeslání z výstupu
WAITOC CALL OUTST   ;Zjistí stav na výstupu
JR C, WAITOC        ;Když je výst.buffer plný, opakuj
DI                 ;Jinak zablokuj mask.přerušení
POP AF              ;Přenášený znak do reg.A
LD (TRNDAT), A      ;a do výst.bufferu
LD A, #FF           ;Signalizace "buffer plný"
LD (TRNDF), A       ;do indikátoru TRNDF
LD A, (OIE)         ;Je očekáváno přerušení?
OR A                ;Když NE, ihned znak odešli
CALL Z, OUTDAT      ;Uvolni přerušení
EI                 ;a vrať se
RET

OUTST LD A, (TRNDF) ;Stav na výstupu
RRA                ;do CY (při CY=1 výst.buffer plný)
RET                ;Návrat se stavem v CY

INIT DI            ;Před inicializací zablokuj mask.přeruš.
SUB A              ;Vynuluj obsah reg.A a indikátory
LD (RECF), A       ;Data Ready (žádný znak na vstupu),
LD (TRNDF), A      ;Character Available (na výstupu nic),
LD (OIE), A        ;výstupní přerušení není očekáváno
LD A, INTRPV SHR 8 ;Vyšší bajt vektoru přerušení do reg.A
LD I, A            ;a do vektorového reg.přerušení
IM 2               ;Nastav mód přerušení na 2
LD HL, RDHDLR      ;Ulož vstupní vektor přerušení
LD (PIOIVA), HL    ;do obl.dat
LD HL, WRHDLR      ;Totéž pro výstupní vektor
LD (PIOIVB), HL
LD HL, PIOINT       ;Do HL 1.adresa inicializačního pole
CALL IPORTS        ;Inicializuj PIO
EI                 ;Uvolni přerušení
RET                ;a vrať se

RDHDLR PUSH AF      ;Uschovej obsah reg.A
IN A, (PIOAD)       ;Přečti data z PIO
LD (RECF), A        ;a ulož je do vstupního bufferu
LD A, #FF           ;Signalizuj "data k dispozici"
LD (RECF), A        ;indikátorem Data Ready
POP AF              ;Obnov obsah reg.A
EI                 ;Uvolni přerušení
RET                ;a vrať se

```

```

WRHLDR PUSH AF      ;Ušchovej obsah reg.AF
LD A,(TRNDF)        ;Zjistí stav výst.bufferu
RRA                 ;rotaci indikátoru do CY
JR NC,NODATA        ;Když CY=0, v bufferu nic není, skok
CALL OUTDAT         ;Jinak pošli data do PIO
JR WRDONE           ;Po přenosu skoč na WRDONE

NODATA SUB A         ;Vynuluj reg.A a indikuj,
LD (OIE),A          ;že výstupní přerušeni není očekáváno
LD A,00000011B      ;a zablokuj výstupní přerušeni
OUT (PIOBC),A

WRDONE POP AF        ;Obnov obsah reg.AF
EI                  ;Uvolni přerušeni
RET                 ;a vrať se

OUTDAT LD A,(TRNDAT) ;Data z výstupního bufferu do reg.A
OUT (PIOBD),A        ;a do PIO
SUB A                ;Vynuluj obsah.reg.A a signalizuj
LD (TRNDF),A         ;"prázdný buffer"
DEC A                ;V reg.A bude FFH (11111111B) pro sig-
LD (OIE),A           ;nalizaci očekávání výstupního přerušeni
LD A,10000011B      ;Uvolni výstupní přerušeni
OUT (PIOBC),A
RET                  ;a vrať se

IPOINTS LD A,(HL)    ;V HL je adr.PIOINT, do A DEFB (3 nebo 0)
OR A                ;Je to už zakončovací nulbajt?
RET Z                ;Když AND, návrat
LD B,A              ;B bude čítač OTIR
INC HL              ;Na další adresu v tab.PIOINT
LD C,(HL)           ;Obsah adresy do C (=číslo portu)
INC HL              ;Opět na další adresu
OTIR                ;Proved přenos dat
JR IPOINTS          ;a vykoněj ještě pro další port

PIOINT DEFB 3        ;Počet přenosů instr.OTIR
DEFB PIOAC           ;Bude inicializováno řízení portu A
DEFB PIOIVA AND FFH ;Vektor přerušeni portu A
DEFB 10001111B      ;Bity 3,2,1,0 výběr módu; 5,4 bez význa-
                    ;mu; 7,6 vstupní mód
DEFB 10000111B      ;Bity 3,2,1,0 řízení přerušeni; 6,5,4
                    ;bez významu, 7,6 vstupní mód
DEFB 3              ;Počet přenosů instr.OTIR
DEFB PIOBC           ;Bude inicializováno řízení portu B
DEFB PIOIVB AND FFH ;Vektor řízení přerušeni portu B
DEFB 11001111B      ;Bity 7,6 mód řízení; ostatní jako port A
DEFB 00000111B      ;Bit 7 blokov.přerušeni; "
DEFB 0              ;Zakončovací nulbajt tabulky inic.dat
RECDAT DEFS 1        ;Vstupní buffer
RECDF DEFS 1         ;Indik.Data Ready (0 bez dat, FFH s daty)
TRNDAT DEFS 1        ;Výstupní buffer
TRNDF DEFS 1         ;Indik.Character Available (0 buffer
                    ;prázdný, FFH plný)
OIE DEFS 1           ;Indik.očekávání výst.přerušeni (0 pře-
                    ;rušení neočekáváno, FFH očekáváno)

```

Malá poznámka: SHR 8 znamená shift right 8 times, čili 8 bitových posunů doprava. Pokud máte generátor strojového kódu, který tuto operaci nezná, budete muset vyšší bajt adresy přerušeni dostat do reg.A jiným způsobem. Totéž platí pro funkci AND, kterou však většina generátorů umí.

Jak vidíte, je vlastní řízení přenosu docela jednoduché. Jediný problém tkví v naprogramování PIO, resp. v nejlepší volbě z nabízejících se možností při řešení té které komunikace. Z80 PIO bylo vyvinuto speciálně pro spolupráci s CPU Z80. Přenos je tzv. interrupt driven, tedy řízen systémem přerušeni. Díky tomu je možná realizace obsluhy většího počtu periférií podle jejich priorit (zapojení daisy chain i nested - viz minulá učebnice). PIO má dva nezávislé porty A a B. Každý z nich pracuje s osmi datovými bity a dvěma signály handshake - Strobe a Ready, jimiž je řízen přenos. Ready informuje periférii, že port je připraven k přenosu. Strobe je odezvou periférie na příjem dat - tehdy je periférie Busy, tedy zaměstnána, a s nikým se nebaví. Teprve po spolknutí dat (ev. může být signalizováno periferním signálem Acknowledge) je periférie připravena pro další příjem, a má-li co odebrat (je-li port Ready), cyklus se opakuje. PIO může pracovat v jednom ze 4 módů:

V módu 0 jsou oba porty výstupní. Kdykoli lze na kterýkoli z nich poslat data. V ten moment PIO hlásí periférii signálem Ready, že je možno data odebrat. Přenos je ze strany periférie stvrzen signálem Strobe.

V módu 1 jsou oba porty vstupní. Když CPU přečte data, port signálem Ready dává periférii na vědomí, že může poslat další. Když je má periférie připravena, vyšle Strobe, port si data převezme a zruší Ready, dokud data neodebere CPU - může se mu vnutit požadavkem na přerušení (je-li uvolněno) v IM 2, nebo musí počkat, až si na něj CPU "vzpomene".

Mód 2 je obousměrný. Přenos probíhá jen prostřednictvím portu A, i když jsou využity přerušení a signály obou portů. Port B musí být nastaven na mód 3 a "vymaskován". Výstup je obdobný módu 0, vstup módu 1. Při vstupu jsou užita jen přerušení a signály portu B.

Mód 3 je trochu zvláštní. Mohou v něm pracovat oba porty. Každé z osmi datových linek je možno určit, zda bude vstupní nebo výstupní. Proto se tomuto módu říká bitový. Za vydatné softwarové podpory jím můžeme imitovat sériový přenos. Další zvláštností je, že nepoužívá ani jeden ze signálů Ready a Strobe. Řízení je čistě interruptové. Přerušení je generováno vždy, když se změní log.stav jednoho nebo všech vstupů. Výstupy jsou řízeny CPU.

Inicializace portů, volba módu a provozních podmínek je věcí naprogramování PIO. Používá se k němu sled řídicích kódů, v nichž log.stav jednotlivých bitů má svůj význam pro aktivaci jednotlivých funkcí portů. Protože jde o záležitost dost specifickou, neuvádím zde jejich soupis. Zájemcům mohou poslat xeroxovou kopii originální dokumentace firmy Zilog.

U nás se poslední dobou uchytil programovatelný obvod 8255, který je dostupnější než Z80 PIO. Díky tomu bylo již o něm zveřejněno poměrně dost informací. Najdete je především v Amatérském rádiu a časopisu Mikrobáze.

Oproti nebufferovanému přenosu se v praxi častěji užívá bufferovaný. Práce s bufferem má četné výhody. Pole přenášených dat můžeme připravit ještě před přenosem, který pak proběhne "jedním vrzem" do bufferu periférie nebo počítače. Tak není třeba nečinně čekat na postupné zpracování dat bajt po bajtu. Díky tomu, že počítač nebo periférie nejsou blokovány procedurou postupného přenosu, mohou se věnovat jiným činnostem. Ruku v ruce s tím prudce stoupá efektivita jejich využití. Buffer lze použít i pro spoustu jiných interních operací.

Bufferovaný přenos se od nebufferovaného liší jen způsobem nakládání s daty v bufferu. Buffer má své čelo a týl. Při I/O operacích je znak odebírán z čela a všechny znaky v bufferu jsou posunuty o jednu pozici směrem k čelu. Tak se uvolňuje týl, do nějž můžeme uložit další znak (je-li). Jde tedy o princip vzniku a odbavování zcela běžné fronty, jak ji vidáme v obchodech. Přitom paní vedoucí stojí u dveří prodejny a po odchodu jednoho zákazníka vpustí do obchodu dalšího, který se postaví na konec fronty v obchodě. Počet rezervovaných míst fronty je konstantní. Jiný způsob manipulace s bufferem data neposouvá, ale po naplnění bufferu se proces jeho plnění zastaví a odběr jde od čela k týlu bez pohybu dat v bufferu. Po dosažení týlu je vyslán signál pro případné opětovné naplnění bufferu. A tak stále dokola. Převáděno na frontu v obchodě - lidé ve frontě se neposouvají směrem k prodáváči, ale prodáváčka nehybnou frontu "sjedne" sama od čela k týlu. Poté paní vedoucí u dveří prodejny celou frontu v obchodě bleskově nahradí dalšími nadočkavci v tomtéž počtu.

2/ Z80 SIO

Tento obvod byl zkonstruován pro sériový přenos dat, jímž jsou data přenášena bit po bitu po jedné lince. Přenos lze časovat, přenášena data oddělovat stop bity, zařazovat paritní bit atd. Na rozdíl od přenosu s PIO může být přerušení SIO vynulováno i bez souvisejícího přenosu dat. Proto je práce s tímto přerušením trochu odlišná. Následující rutina obsahuje části, které jsou funkčně zcela shodné se stejnojmennými částmi rutiny předchozí.

Délka: 202 bajtů a 5 pro data
Počet cyklů: INIT 618, WRHDLR 160, ostatní jako u PIO
Vstup, výstup a použité registry jako u PIO

SIOCAD EQU	#1C	;	Datový kanál A
SIOCRD EQU	#1E	;	" " B
SIOCAS EQU	#1D	;	Řídicí/stavový kanál A
SIOCBS EQU	#1F	;	" " " B

```

SIOIV EQU #B000      ;Vektorová adresa přerušení
SIOV EQU SIOIV+8      ;Výstupní vektor přerušení kanálu A
SIOEV EQU SIOIV+10     ;Externí/stavový vektor přerušení kan.A
SIOV EQU SIOIV+12     ;Vstupní vektor přerušení kanálu A
SIOV EQU SIOIV+14     ;Spec.příjmový vektor přerušení kanálu A

INCH CALL INST        ;Totéž jako u PIO
JR NC, INCH
DI
SUB A
LD (RECD), A
LD A, (RECDAT)
EI
RET

INST LD A, (RECD)     ;Totéž jako u PIO
RRA
RET

OUTCH PUSH AF         ;Totéž jako u PIO
WAITOC CALL OUTST
JR C, WAITOC
DI
POP AF
LD (TRNDAT), A
LD A, #FF
LD (TRNDF), A
LD A, (OIE)
OR A
CALL Z, OUTDAT
EI
RET

OUTST LD A, (TRNDF)   ;Totéž jako u PIO
RRA
RET

INIT SUB A            ;Totéž jako u PIO
LD (RECD), A
LD (TRNDF), A
LD (OIE), A
LD A, SIOV SHR 8
LD I, A
LD IM, 2
LD HL, RDHDLR        ;Vektor vstupního přerušení
LD (SIOV), HL
LD HL, WRHDLR        ; " výstupního "
LD (SIOV), HL
LD HL, EXHDLR        ;Externí/stavový vektor
LD (SIOEV), HL
LD HL, REHDLR        ;Vektor chyby na vstupu
LD (SIOV), HL
LD HL, SIOINT        ;1.adr.tabulky inicializačních dat
CALL IPORTS          ;Inicializuj SIO
EI                   ;Uvolni přerušení
RET                  ;a vrať se

RDHDLR PUSH AF        ;Totéž jako u PIO
RDI
IN A, (SIOCAD)
LD (RECDAT), A
LD A, #FF
LD (RECD), A
POP AF
EI
RET

WRHDLR PUSH AF        ;Totéž jako u PIO
LD A, (TRNDF)
RRA
JR NC, NODATA
CALL OUTDAT
JR WRDONE

NODATA SUB A          ;Vynuluj obsah reg.A a signalizuj
LD (OIE), A          ;"přerušení neočekáváno"
OUT (SIOCAS), A       ;Příprav selekci reg.0 SIO
LD A, 00101000B      ;Inicial.přerušení vysílače SIO
OUT (SIOCAS), A

WRDONE POP AF         ;Totéž jako u PIO
EI
RET

```

```

EXHDLR PUSH AF      ;Došlo ke změně vlivem ext./stav.
LD A,00010000B      ;Inicial.stavové přerušeni
OUT (SI0CAS),A
EI                  ;Změnil se stav DCD nebo CTS, nebo došlo
POP AF             ;k zastavení přenosu
RETI               ;V případě nutnosti proved' ošetření

REHDLR PUSH AF      ;Spec.přerušeni při chybě na příjmu
LD A,00110000B      ;Inicial.toto přerušeni
OUT (SI0CAS),A
EI
POP AF
RETI               ;V případě nutnosti proved' ošetření

OUTDAT LD A,(TRNDAT) ;Totéž jako u PIO, kromě toho, že se
LD (SI0CAD),A       ;není třeba uvolňovat výstupní přerušeni
SUB A
LD (TRNDF),A
DEC A
LD (OIE),A          ;(OIE)=FFH
RET

IPOINTS LD A,(HL)   ;Totéž jako u PIO
OR A
RET Z
LD B,A
INC HL
OTIR
JR IPOINTS

;Kanal A do výchozího stavu:
SI0INT DEFB 1        ;Čítač instr.OTIR
DEFB SI0CAS          ;Adr.řídícího stavu kanálu A
DEFB 00011000B      ;Bity 2,1,0 reg.0 (zápis) SI0; 5,4,3
                      ;inic.kanálu; 7,6 bez vlivu

;Nastavení vektorového přerušeni a umožnění vlivu stavu na přer.
DEFB 4              ;Čítač instr.OTIR
DEFB SI0CBS          ;Adr.řídícího registru B SI0
DEFB 00000010B      ;Selekce zápisového reg.2 SI0
DEFB SI0IV AND FFH   ;Nastavení vektoru přerušeni SI0
DEFB 00000001B      ;Selekce zápisového reg.1 SI0
DEFB 00000100B      ;Umožnění vlivu stavu na vektor přeruš.

;Inicial.kanálu A (rychlosti přenosu, řízení příjmu a vysílání)
DEFB 8              ;Čítač instr.OTIR
DEFB SI0CAS          ;Adr.řídícího reg.A SI0
DEFB 00010100B      ;Selekce zápis.reg.4, vynul.ext./st.přer.
DEFB 01001000B      ;Bity 0 bez parity; 1 bez vlivu; 3,2 je-
                      ;den a půl stop bitu; 5,4 bez vlivu; 7,6
                      ;rychlost přenosu 1/16 násobek hod.taktu
DEFB 00000011B      ;Selekce zápis.reg.3
DEFB 11000001B      ;Bit 0 uvolnění příjmu; 4,3,2,1 bez vli-
                      ;vu; 5 blokáda autom.uvolnění; 7,6 pří-
                      ;jem ve formátu 8 bitů/znak
DEFB 00000101B      ;Selekce zápis.reg.5
DEFB 11101010B      ;Bit 0 bez CRC (kontroly cyklickým kó-
                      ;dem) při vysílání; 1 požadavek
                      ;na odeslání; 2 bez vlivu; 3 uvolnění
                      ;vysílání; 4 neposílej BREAK; 6,5 vysí-
                      ;lej ve formátu 8 bitů/znak; 7 datový
                      ;terminál připraven
DEFB 00000001B      ;Selekce zápis.reg.1
DEFB 00011011B      ;Bit 0 externí přerušeni; 1 uvolnění
                      ;přerušeni vysílání; 2 bez vlivu; 4,3
                      ;příjem přerušeni pro všechny znaky, pa-
                      ;rita neovlivňuje vektor; 7,6,5 funkce
                      ;wait/ready neaktivována
                      ;Zakončovací nulbajt datové tabulky
RECDAT DEFB 0        ;Totéž jako u PIO
RECDAT DEFS 1
RECDAT DEFS 1
TRNDAT DEFS 1
TRNDAT DEFS 1
OIE DEFS 1

```

Na první pohled je patrné, že programování SI0 je o dost komplikovanější než je tomu u PIO. Sériový přenos si žádá zaručené znalce. Věc je obtížnější nejen složitostí programování obvodu, ale i tvorbou formátu dat. Ten závisí i na volbě mezi asynchronním a synchronním přenosem. Pro ten druhý existují různé standardy, z nichž některými SI0 disponuje -

monosync, bisync, SDLC, HDLC a CCITT-X.25. Synchronizace může být i externí. SIO umí přenášet nejen bajtové, ale i bitové orientovaná data, a to po 5, 6 a 7 bitech. Totéž platí pro synchronizační data. Jistou složitostí oplývá ošetřování různých signalizací a případných chyb v přenosu. SIO umí spolupracovat s obvodem DMA pro rychlý přenos dat z/do paměti bez účasti mikroprocesoru. To vše, na rozdíl od PIO, činí Z80 SIO univerzálním obvodem, který může spolupracovat i s jinými mikroprocesory.

SIO může pracovat v jednom z těchto módů: s výzvami (polling), přerušeními (vektorovanými nebo nevektorovanými) nebo blokovým přenosem dat, stavů a řídicích kódů do/z CPU (týká se i DMA).

SIO má 2 kanály (A a B). Kanál A obsahuje 2, kanál B 3 čtecí registry, číslované RR0-RR2. Kanál B má navíc RR2 s možností interní modifikace vektoru přerušení. Ze čtecích registrů můžeme číst stavové informace. Přímě to jde jen z registru RR0. Pro přečtení ostatních se napřed musíme obrátit na reg. WR0 (viz dále) a patřičným kódem se přes něj dostat ke čtecímu. Zapisovacích registrů má kanál A sedm, B osm (WR0-WR7). Obdobně jako u RR2 má kanál B navíc WR2 pro vektor přerušení obou kanálů. WR0 vyžaduje pro zápis jen 1 bajt, ostatní po dvou. Je to dáno tím, že bity 2-0 WR0 adresujeme požadovaný registr. Rozpis významu jednotlivých bitů všech 11 registrů zde opět neuvádím. Podobně jako v případě PIO mohou zájemcům zaslat xeroxovou kopii firemní dokumentace.

Jistou obdobou SIO je obvod DART. Není vybaven tolika vymoženostmi řízení sériového přenosu jako SIO, ale umí převádět přenášený signál z paralelního formátu na sériový a naopak. Hodí se zvláště pro modemový přenos.

Do rodiny obvodů Z80 patří ještě obvod CTC (Counter/Timer Circuit) pro časové operace, resp. jejich řízení v reálném čase. Může rovněž spolupracovat se Z80 SIO při generování kmitočtů pro různé rychlosti přenosu. Často se používá jako "digitální hodinky" počítače, čehož se využívá v případech, kdy je vítané nebo přímo nutné informaci o čase zahrnout do zpracovávaných dat. CTC má 4 nezávislé programovatelné kanály, jejichž stav můžeme kdykoli přečíst. Taktů vnitřních hodin počítače mohou být převedeny na jednotkové cykly po 16 nebo 256 taktů. Do registru pro časovou konstantu můžeme uložit číslo v intervalu 1..256 (256 je 0). Jím je naplněn cyklicky se zmenšující registr, který může být dekrementován v jednom ze dvou módů. Když dosáhne nuly, znova se naplní časovou konstantou. V časovacím módu je jednotkový časový úsek součinem tří faktorů: periody vnitřních hodin, výše uvedeným faktorem odečtu taktů po 16 nebo 256 a časové konstanty. Při hodinách kmitajících ve 4 MHz je nejkratší dosažitelný úsek 4 mikrosec, nejdelší 16,4 milisek. Při potřebě delších úseků je možné řadit obvody CTC do kaskád.

Rodinku doplňuje programovatelný obvod DMA pro rychlý přenos bloku dat, resp. hledání zadaného bajtu. Přenos dat (sled po jednom bajtu) může probíhat mezi dvěma porty (perifériemi, pamětmi). Hledání specifikovaného bajtu (nebo jeho maskované části) se může vztahovat k jednomu portu paměti nebo periférie. Je možné volit i kombinovaný přenos s hledáním mezi dvěma porty. Hledání je velmi rychlé - obvod Z80A DMA za 1 vteřinu prohledá 2M paměti. Operace probíhají ve třech módech: a/ Bajt-po-bajtu - po každém přeneseném bajtu je uvolněna sběrnice pro CPU a pro další přenos si ji DMA musí znova vyžádat. b/ S přerušením - DMA okupuje sběrnici (CPU je "odstaveno") a provádí zadanou operaci, která může být přerušena signálem linky Ready z portu. Po "dokončení bajtu" DMA uvolní sběrnici pro CPU. c/ S pokračováním - jako u b/, ale po signálu Ready není předána sběrnice CPU; DMA po jeho obdržení pouze "pauzuje". Po následujícím signálu Ready opačné log. úrovně se zase rozjede. Sběrnici uvolní až po dokončení zadaného úkolu. DMA pracuje vždy se dvěma 16-bitovými adresami. Jednou z vymožeností DMA je auto restart. Startovací adresa kteréhokoli z obou portů přenosu/prohledávání je uschována a po skončení operace může další začít opět od/ná ní. CPU je tak uvolněno od softwarových činností pro zadávání této adresy. Na druhou stranu však u operací bajt-po-bajtu nebo s přerušením může CPU v momentu přístupu ke sběrnici adresu auto restartu změnit. Při hledání zadaného bajtu může po jeho nalezení proběhnout jedna z naprogramovatelných akcí v případné návaznosti na spolupracující software. Obvody DMA lze řadit do řetězce podle priorit.

V závěru krátkého přehledu součástkové základny firmy Zilog si lze jen postesknout, že se firmě nepodařilo zopakovat úspěch s jejím osmibitovým mikroprocesorem také u šestnáctibitového Z800. Zatímco Intel musel se svým 8080 vyklidit pole Zilogu, obvody 8088/86 mu vše vrátil i s úroky. A my se při přechodu na 16-bitové počítače musíme učit novou mneoniku. To je ale jen malá daň opravdové kvality.

Závěr

Spisovatel se musí naučit písmenka, malíř tahy štětcem. Ve věcech základů techniky jeden člověk druhého vést může. Ale tvorba sama je (by měla být) bez hranic. Kdokoli se je druhým snaží vytyčovat a vnucovat, je roztomilým hlupákem. Jeho roztomilost ovšem klesá se vzrůstající potencí držby vnucovacích prostředků. Rozdávát nějaká moudra v závěru učebnice tohoto druhu by proto byl holý nesmysl. Dopustil-li jsem se v ní "assemblerového moralizování", přijměte jeho podstatu jediné tehdy, poznáte-li sami, že je v ní něco pravdy. Doberete-li se vlastní práci k jinému pohledu, řiďte se jím. A nebojte se jej změnit, kdykoli se ukáže, že je to dnes najednou trochu jinak, než tomu bylo ještě včera. Jedině tak se dá jít dopředu.

V opoře o znalost základů systému vašeho počítače můžete z rutin této učebnice sestavit jakýkoli obecnější program - databanku, slovní procesor, kalkulační program...ani Jet Set Willy před vámi neobstojí. Mimochodem - v této dnes již klasické hře tvoří tabulky dat kolem 70 procent celého programu. Pouhých 30 procent jsou instrukce, manipulující s daty i frenetickými oběťmi herního softwaru. Nejen kvůli Willymu a spol. věnujte organizaci dat i způsobu jejich manipulace náležitou pozornost a péči. Daty nejsou jen čísla či písmenka, která počítač na vstupu nutíme ke spolknutí. To je jen polotovár, z něž rutiny na výstupu vyhodí obrobek v určité formě. Nesmírně důležitá jsou programová, tedy vnitřní data, uživateli programů skrytá - systémové proměnné a konstanty všeho druhu, jimiž mohou být třeba i adresy programových funkcí řazených do hierarchie operačních substruktur s různými cykly a vzájemnými prostupy. To platí nejen v systémech řízených přerušováními - i v Jet Set Willy se neustále prolíná test klávesnice s projekcí grafických dat a testy, zda některá z adres, na niž je Willy vykreslen, nesměruje do "nebezpečné adresy". Tabulky dat vám ulehčí třírozměrnou projekci, když do nich převedete část údajů ze školních tabulek goniometrických funkcí. Jednoduchým prohlédáváním tabulky docílíte mnohem vyšší rychlosti než programovým výpočtem funkcí.

Vedle generální strategie organizace vnějších i vnitřních dat je tím nejkratičtějším polem programování tvorba algoritmů funkcí - od nejvyšší hladiny řešení problému až po členité dno programového zpracování s jeho nezbytnou vnější prezentací. Nevěřte rutině, prohlašující, že všechno je na jedno brdo - tady data, táhle jejich třídění, tuhle prohlédávání, k tomu něco přenosů a šmytců. Sochařství bychom pak mohli považovat jen za trochu opadaného kamene z něčeho, čemu se říká socha. Rutinní řemeslníci budou vždy ve všech oblastech konání. Sériovost jejich počínání je fenoménem nízké úrovně nabídky, kterou stimuluje i nízká úroveň poptávky. Obecná nenáročnost rodí podprůměr. Všechno jde smontovat z panelů.

Programování jako panelstory? Mistři plátna nechávali řemeslníky domalovávat na svých obrazech koňské zadky. Program složený z koňských zadků může být funkční. Nakonec - i samotný koňský zadek může zdobit kdejakou síň. Ale - ptám se - je tu někde nějaký Mistr?

S každým novým dnem se stává jasnější, že matematika, logika, kybernetika a další nově vznikající vědní obory, zabývající se chováním systémů, mají co říci všem oborům lidského konání. Probíhající revoluční zvrat chápání dějů probíhajících kolem nás i v nás je umožněn kvalitativním skokem z oblasti industriální manufaktury, degradující člověka na výrobní nástroj, do nové éry osvobození lidského jedince z historického područí psycho-sociální entropie. Tento skok je přímo podmíněn všeobecnou aplikací systémů výpočetní techniky, které - z pohledu společenské fylogeneze - jsou sekvencně vyšší vývojovou etapou interakce a identity společenských struktur v celé jejich hierarchii. Revolučnost nasazení výpočetní techniky spočívá v otevírání různou měrou uzavřených systémů. Jak známo, uzavřený systém není schopen vývoje, chybí mu vnitřní dynamika procesů, protože nepřijímá a nezpracovává podněty (zákon termodynamiky). Výpočetní systémy mohou svou měrou přispět k otevírání zrezivělých zámků. Na druhou stranu by bylo chybou podlehnout přílišné euforii nad pseudokouzelným elektronickým klíčem k budoucnosti. Kolik podobných už tu bylo, jenže...

Jak z 1.kapitoly učebnice vyplývá, program je cílevědomým produktem - systémem s nejvyšším stupněm organizovanosti, dané funkčními vztahy mezi jeho jednotlivými částmi a celou jejich strukturou. Tyto vztahy jsou tvořeny sítí spojovacích svazků (pomyslných nervových vláken) substruktur. Chování celého systému je závislé na tom, jak dalece je organizován pro plnění jím vykonávaných funkcí. Ironií současného programování je, že od systému požadujeme - matematicky řečeno - nulovou entropii, tedy aby žádná jeho substruktura neměla tu nejmenší svobodu projevu. Pokud ji má, projeví se chybou, s možností zhroucení celého systému. Tak vlastně s nejvyšším nasazením své kreativity tvoříme systém, který žádnou kreativitu nemá. Je to dáno tím, že u sériové logiky - přese všechnu složitost nervové sítě - v ní může proběhnout vždy jen jeden pomyslný impuls či přenos po jednom jejím vlákně. Všechna ostatní zůstávají nevyužita, i když jsou v potenciální pohotovosti. Která ze substruktur bude kdy aktivována, je dáno rozhodovacími body - architektonický souhrn jejich binárního větvení do substruktur s jejich funkční projekcí determinuje chování systému. Tím je zároveň řečeno, že použijeme-li substrukturu jednoho systému v jiném, může v něm být funkční, i když se druhý systém bude chovat odlišně.

Jistě není bez zajímavosti, že binární větvení se používá i v jiných oborech. Jistou obdobou binárního stromu z 11.kapitoly je rozhodovací strom v psychologii. Např. máme zjistit, který z osmi předmětů má někdo na mysli. K výběru máme 4 kolečka a 4 kostky, z nichž jsou vždy dvě větší, dvě menší; tyto páry jsou dále odlišeny barvou - červenou a bílou. Pokud bychom postupovali lineárně, tedy přímo otázkou: "Je to tohle?", strefili bychom se přinejlepším poprvé, přinejhorším poosmé. Když postupíme binárně:

vstup			
kolečko?		kostka?	
malé?	velké?	malá?	velká?
červ.? bílé?	červ.? bílé?	červ.? bílá?	červ.? bílá?

pak se dostaneme k cíli vždy napotřetí. Možnost použití takového stromu (i s obsahově jinou hierarchií) při tvorbě programové nebo datové struktury je nabílední.

Použití počítače bez napojení na informační síť je dnes už možno považovat za pustý anachronismus. Takto izolovaný počítač je uzavřeným systémem. Nejde tu jen o spojení s databankou, která vám prozradí dnešní počasí na Sumatře či ovládání libovolného interfacu. Jde i o možnost napojení vašeho počítače coby terminálu na tzv. centrální počítač, který velmi rychle provede operace, na něž byste jinak potřebovali spoustu času a paměti. Propojováním počítačů do sítě se rapidně zvyšují interaktivní a realizační schopnosti celého takového systému. Vedle nově vznikajících projektů paralelní logiky s perspektivně kreativními subsystémy je tvorba rozsáhlých informačních sítí největším příslibem efektivní reflexe předlouhé řady problémů rozvoje současné společnosti.

I když je váš počítač doma na stole řádně odizolován od jakéhokoli nedejbožesystému, nevěšte hlavu. A tvořte. Je prokazatelné, že míra prožitku z tvorby je stejná u génia jako u kohokoli jiného, kdo pozná, co to je. Dokáže snad génius být zamilován "o 10 deka víc" než ostatní lidé? Tvořte i bez nároku na Nobelovu cenu. Na jakoukoli cenu. Ocenění, jako vnější data, začnou časem přicházet sama. Sebelepší důvtipný objev v procesu programové tvorby vás vždy obohatí o nové poznatky, přístupy - o vnitřní data, která nikde jinde, nijak jinak neseženete. Při absenci těchto dat bývají ta vnější hluchým torzem. Tvůrčí proces integruje zažité poznání se vším předchozím, čas dodává jeho plodům zralosti. Subtilní interakce toho typu by měla být jednou z hlavních spojnic obsahu všech stránek učebnice. Neberte proto jednotlivé rutiny a úvahy jako izolované střípky, ale zkuste z nich v mysli skládat obrazy. Jejich expresi máte jen ve svých rukou.

INDEX

Aritmetické rutiny

BCD/2 (15)
Binární čísla - násobení a dělení 2-bajt.č. (16), 4-bajt.č. (25-27), multibajt.součet a odečet (18), násobení (18), dělení (20)
Dekadická čísla - multibajt.součet a odečet (21), násobení (22), dělení (23)
Odmocňování 2.stupně (28-29), 3.stupně (30-33)
Multibajtový posun vpravo aritm.(33), logický (34)
Multibajt.rotace vlevo a vpravo (35)

Generování pseudonáhodných čísel

Náhodné znaky (37), 4-bajt.bin./BCD (37), 2-bajt.bin.(38), 4-bajt.bin.(39), 2-bajt.č.bitovou metodou (40)

Konverze

1-bajt.bin./BCD (41), multibajt.bin./BCD (41), 1-bajt.BCD/bin. (42), 1-bajt.bin./2 HD-ASCII znaky a naopak (44), 2-bajt.bin./dek-ASCII (44), 4-bajt.bin./dek-ASCII (46), dek-ASCII/2-bajt.bin.(47), ASCII s čís.zákl.2-36/2-bajt.bin.(48), až 255-bajt.bin./dek-ASCII (49), dek-ASCII/4-bajt.bin.(50), dek-ASCII/BCD (52), BCD/dek-ASCII s pravou zarážkou (52), k levé zarážce (53), 16 bitů/16 znaků ASCII (52), konv.znaků text.souboru (54), převod zdroj.textu assembleru/slovní procesor (13)

Kompresce a expanze dat

Velkých ASCII znaků (57), exp.tokens (58), exp.textu přímo adresovanými řetězci (59), kompr.pro přenos dat (62), kompr.a exp.obraz.dat bajtová (64), bitová (66)

Pole

Transpozice matic (72,73), vyhled.adr.1-bajt.(74) a 2-bajt.elem.matic (75), vyhled.adr.elem.v N-rozm.poli (75), hledání bajtu s největší (77) a nejmenší (78) hodnotou v poli s max.255 1-bajt.elementy, binary search (78)

Řetězce

Porovn.dvou max.255-bajt.řetězců (80), spojení dvou řetězců (80), porovn.dvou max.65536-bajt.řet.(81), porovnání s řet.za CALL (82), porovn.dvou řet.se zápisem diferencí (82), hledání dílčího řetězce (83), kopie části řetězce (84), výmaz části řetězce, vložení jednoho řet.do jiného (86)

Třídění

Insertion sort (88), bubble sort (89), quick sort (91)

Tabulky

Jump table: podle pořadí (96), info bajt/1-bajt.klíč/adresa (97), info-bajt a opakovaně 1-bajt.klíč/adresa (98), totéž, ale se 3 info-bajty (98), opakovaně adresa/info-bajt/klíč (99), Table look-up: hledání řetězce v tabulce dat (99)

Rutinní drábky

Vyplnění paměti (105), intelig.přenos bloku dat (106) s jejich rotací (107), tab.či řet.za CALL (108), relat.ofset volání (108) DEFb za CALL jako klíč k adr.volání (109), určení adr.volání podle ofsetu (109), změna návrat.adresy (109), 64 znaků na šifce 32 bajtů (110), kurzor pro totéž (113), přenos text.řetězců z tabulky (114), test tlačítka (8,9), zvuk a boder (9)

Interfacing

Paralelní (114), sériový (118)

Jiná témata

Kompr.tabulky slovníku (60-62,101), hashing (101), bitová mapa (101), binární strom a heap sort (102), časové proporce (91,104)

Literatura

Assembler Routines for the Z80 (David Barrow)
Z80 Assembly Routines (Lance A.Leventhal/Winthrop Saville)
Programming Pearls (Jon Bentley)
Zilog Components Data Book 1983/4
Machine Code - Assembler Z80 (L.Zajíček)
Učíme stroje česky (Hajičová/Piřha/Sgall)
Výpočetní a řídicí technika (Encyklopedie SNTL)
Obecná vývojová psychologie (H.D.Schmidt)