Amatérské RADIO

ŘADA B – PRO KONSTRUKTÉRY

ČASOPIS PRO ELEKTRONIKU
ROČNÍK XXXIX/1990 ● ● ČÍSLO 6

V TOMTO SEŠITĚ

● ● ●

AMATÉRSKÉ RADIO ŘADA B

Vydává vydavatelství MAGNET-PRESS, s. p., Vladislavova 26, 135 66 Praha 1, tel. 26 06 51–7. Šéfredaktor L. Kalousek, OK1FAC. Redakce Jungmannova 24, 113 66 Praha 1, tel. 26 06 51–7, linka 353, sekretářka linka 355. Ročně vyjde 6 čísel. Cena výtisku 6 Kčs, pololetní předplatné 18 Kčs. Rozšiřuje PNS, v jednotkách ozbrojených sil vydavatelství NAŠE VOJSKO, administrace Vladislavova 26, Praha 1. Informace o předplatném podá a objednávky přijímá každá administrace PNS, pošta, doručovatel a předplatitelská střediska. Objednávky do zahraničí vyřizuje PNS – ústřední expedice a dovoz tisku Praha, administrace vývozu tisku, Kovpakova 26, 160 00 Praha 6. Tiskne NAŠE VOJSKO, s. p., závod 08, 160 00 Praha 6, Vlastina ulice č. 889/23.
Za původnost a správnost příspěvku odpovídá autor. Návštěvy v redakci a telefonické dotazy po 14. hodině. Číslo indexu 46 044.
Toto číslo má vyjít podle plánu 10. 12. 1990.

Vážení čtenáři,

tímto číslem opět končí jeden z dosavadních ročníků Amatérského radia řady B. Potěšilo nás, že přes množství nových periodik i dalších publikací nám zůstali čtenáři věrni – snažíme se proto i v dalším ročníku přinést co největší množství nových a moderních informací z celé oblasti elektroniky. Přesto, že je koncepce našich elektronických výrobních závodů dosud velmi nejasná, domníváme se, že se díky soukromým podnikatelům v budoucnu podstatně zlepší např. nabídka součástek na našem trhu (tomu nasvědčují i inzeráty v sesterském „červeném" AR), možnosti nákupu desek s plošnými spoji, elektronické „bižuterie", skříněk, transformátorů atd., což všechno byly dosud úzké profily v zájmové (a nejen zájmové) elektronické činnosti, které omezovaly možnosti konstruktérů.

Příští ročník Amatérského radia řady B bychom chtěli zahájit monotematickým číslem, věnovaným konstrukci přístrojů, které souvisejí s elektronikou v hudbě. Kromě konstrukce kytarových snímačů a jejich úprav, různých elektronických efektů, syntezátorů apod. budou popsány i polovodičové a elektronkové (ty na přání čtenářů, kteří tvrdí, že zvuk elektronkových zesilovačů je polovodičovými součástkami nedosažitelný) zesilovače různých výkonů a uspořádání.

Z dalších připravovaných čísel uveďme např. námětově velmi oblíbené číslo, věnované zajímavým zapojením. Tentokrát by měl být obsah velmi pestrý – poplachová zařízení, zapojení pro motoristy, nf zařízení, síťové zdroje, řízení výkonu, časovací zařízení, zapojení s tranzistory s jedním přechodem, vf obvody atd. Toto číslo by mělo vyjít pravděpodobně v druhé polovině roku.

Jedno z připravovaných čísel bude věnováno optoelektronice (technologie a provedení diod LED, laserových diod, segmentovek, displejů z kapalných krystalů, fluorescenčních displejů, příklady zapojení, rady pro použití), další praktickým pomůckám pro elektroniky (vybavení dílny – svářečka, bodovka, malá ruční bruska, jednoduché stolní nůžky, pomůcky pro zemědělství a zahrádkáře – ionizace vody, automatické zalévání, sušení zdí elektroosmózou apod., elektronické hračky, světelná technika – světelné efekty, hlasité telefony, přístroje do domu a domácnosti – časové spínače, spínání zvukem, televize na sluchátka apod.).

Jako každoročně bude i v příštím roce jedno číslo věnováno zajímavým integrovaným obvodům, tentokrát budou uvedeny údaje o spínacích obvodech prahových úrovní, o senzorových obvodech, dále o operačních zesilovačích jak bipolárních, tak MOS.

Velký zájem předpokládáme o číslo, jehož náplní bude problematika rušení a odrušování. Kromě nezbytné teorie v nejnutnější míře budou obsahem tohoto čísla návody na stavbu a použití nejrůznějších odrušovacích prostředků, které by měly zabezpečit kromě jiného nerušený příjem rozhlasu a televize. Všechna uvedená odrušovací zařízení byla v praxi mnohokrát odzkoušena profesionálním pracovištěm, specializovaným na tuto oblast elektroniky.

Je uzavřena i smlouva na dodání rukopisu pro číslo, věnované osciloskopům a jejich použití.

To je tedy zhruba nástin obsahu čísel pro příští ročník. Aby byla informace úplná, je třeba dodat, že na všechna čísla redakce uzavírá s autory smlouvu, v níž se autoři zavazují dodat příslušné materiály do určitého termínu. Pokud tento termín nesplní, což se také stává, musí se posunout i termín vyjití toho kterého plánovaného námětu – v nejhorším případě je možný i skluz do dalšího ročníku. Proto se nehoršete na redakci, jestliže některý ze slíbených titulů nevyjde – vina není skutečně na naší straně.

Na závěr bych chtěl pouze upozornit, že redakce stále přijímá nabídky čtenářů (i nečtenářů) AR řady B na vypracování materiálů pro náplň jednotlivých čísel z celé oblasti elektroniky. Samozřejmě uvítáme i jakékoli náměty a připomínky k obsahu i zpracování jednotlivých čísel.

Redakce

Za ing. Františkem Smolíkem

29. září zemřel po dlouhé nemoci bývalý šéfredaktor AR, F. Smolík, OK1ASF. Vedl časopis od jeho založení až do svého odchodu do důchodu, kromě toho byl dlouholetým funkcionářem čs. radioamatérské organizace a propagátorem radiového orientačního běhu. Jako šéfredaktor i funkcionář má nepopiratelné zásluhy na popularizaci elektrotechniky a radiotechniky u nás.

Čest jeho památce.

ÚVOD DO TECHNIKY A PROGRAMOVÁNÍ MIKROPOČÍTAČŮ

František Kyrš, Tomáš Kyrš

Prudký a trvalý průnik číslicové a mikropočítačové techniky snad do všech oblastí nejen elektroniky staví dnes každého z nás, bez ohledu na specializaci, před nutnost zvládnout na určité úrovni i tento obor. Jako příklad současného vývoje snad stačí uvést donedávna klasickou techniku audio/video, v níž se to nyní číslicovými obvody, mikrokontroléry, paměťmi, kódy, sběrnicemi a programy jen hemží. A to je teprve začátek. Zůstat stát znamená v několika letech ztrátu kontaktu s celou touto oblastí, podobná situace je i v ostatních oblastech aplikované elektroniky.

V počáteční fázi seznamování se s mikroprocesorovou technikou je užitečné získat nejprve jistý, byť i povrchní, ale relativně ucelený přehled o této, pro mnohého zcela nové problematice. Což je ovšem pro jinak zaměřeného technika poměrně obtížné a zvláště časově náročné. Pokusem o pomoc na cestě k tomuto cíli je toto číslo AR-B, volně navazující na AR-B č. 5/89 [1]. Při koncipování obsahu a struktury jsme se především snažili obsáhnout přístupnou formou, uzpůsobenou spíše hardwarově orientovaným konstruktérům bez předběžných znalostí programování, co nejširší tématický rozsah, s logickou návazností jednotlivých podstatných partií. Při tom jsme nutně, vzhledem k omezenému rozsahu příspěvku, museli vycházet ze skutečnosti, že nežijeme ve vzduchoprázdnu – dané problematice, pochopitelně v různém rozsahu, na různé úrovni i s různým přístupem již byly, jsou a budou stále muset být věnovány příspěvky mnoha jiných autorů. Na některé vybrané práce se proto v jednotlivých kapitolách odvoláváme. Na druhé straně jsme naopak kladli zvýšený důraz na některé základní partie, o nichž si myslíme, že již při prvním kontaktu zasluhují zvláštní pozornost. Jsme si vědomi toho, že zvolený přístup může být terčem do jisté míry oprávněné kritiky, my ho však považujeme za správný a užitečný.

Snažili jsme se pochopitelně i o to, aby v příspěvku pro sebe našli něco zajímavého čtenáři různých kategorií. Hlavním záměrem však bylo to, aby co nejvíce z vás, kteří oblast mikropočítačů dosud nechráváte stranou, alespoň „uvízlo drápkem". Již proto, že tato technika již zdaleka není samostatným, izolovaným oborem, ale uplatňuje se prakticky všude a mění i dosavadní přístupy ke konstrukci elektronických zařízení. To lze ostatně dobře sledovat i na zelených stránkách AR, které představují trvalý zdroj informací pro další prohlubování znalostí i podnětů pro praktickou činnost.

Základy programování

Již v zmíněném AR-B č. 5/89 [1] jsme si vytvořili poměrně ucelenou představu o základních principech činnosti centrální procesorové jednotky i sestavy jednoduchého mikroprocesorového systému. Chápeme smysl a požadavky na skladbu instrukčního souboru konkrétní CPU i jeho reprezentaci v asemblerové mnemonice nebo strojovém binárním kódu. Znalost funkce základních technických prostředků a instrukčního souboru konkrétního mikropočítače však lze zhruba přirovnat k pouhé znalosti abecedy, o které nevíme, jak jí využít. K tomu nám chybí skutečná znalost konkrétního programového jazyka – v přeneseném významu pak jak jeho „mluvnice" (syntaxe – formálních pravidel práce s tímto jazykem), tak „literatury" (sémantiky – logiky správné a systematické tvorby programu). Počáteční orientace v oblasti programování je zpravidla obtížná z řady důvodů. Nicméně, naučit se programovat je možné pouze tak, že se programovat začne. A to nejlépe tak, aby postup byl přehledný, systematický a nebyl zdrojem špatných počátečních představ a návyků.

Před zahájením vlastní práce je nezbytné alespoň přehledové seznámení s konkrétním programovacím jazykem. Těch je ovšem celá řada a my jsme se dosud neseznámili se žádným. Popsali jsme si sice „asembler" I8080, ale pouze jako výčet typů instrukcí tohoto mikroprocesoru. Což ovšem programovací jazyk není, i když přechod k jazyku symbolických adres JSA 8080 je poměrně jednoduchý. Začínat s programováním na úrovni jazyka symbolických adres však není vhodné z celé řady důvodů.

Smyslem programování je vytvořit předpis pro jednoznačné a konečné řešení konkrétní úlohy, probíhající nad vstupními a vytvářenými daty s využitím konkrétního programovacího jazyka. Tento jazyk tedy pro programátora představuje prostředek vzájemné vazby mezi algoritmickým řešením úlohy a jejím skutečným řešením na úrovni strojového kódu vlastním počítačem.

Jazyk symbolických adres představuje právě nejnižší úroveň programovacího jazyka vůbec. Protože je zcela závislý na konkrétním typu procesoru, označuje se jako strojově orientovaný, kvůli užívání asemblerové mnemoniky většinou přímo jako asembler, i když tak tento pojem získává dva odlišné významy. Základním problémem při programování v asembleru není to, že je třeba znát instrukční soubor příslušného procesoru, ale především relativní nepřehlednost a složitost programu, vytvářeného na této úrovni. Začínající programátor, který se nejprve musí orientovat v základních principech programování, datových i příkazových typech a strukturách či rutinních postupech řešení klasických algoritmů, by se tak zbytečně dostával do nepřehledné a víceméně bezvýchodné situace. Pouze pro ilustraci si všimněme například problémů s interpretací různých datových typů. Uvažujeme-li běžný 8bitový počítač, je běžnou informační jednotkou byte, tedy možným rozsahem hodnot omezené binární číslo, které může být uloženo v jednoduchém registru nebo jednom paměťovém místě. V praxi je ovšem třeba pracovat s různými datovými typy, (integer, Boolean, real, char . .), jejichž zobrazení i zpracování na úrovni JSA musí být řešeno programovou cestou, což je mnohdy velmi složité a nad síly začínajícího programátora, který je těmito problémy zbytečně rozptylován. Podobné problémy přináší na úrovni JSA nutnost vytvářet potřebné příkazy a příkazové struktury. JSA jako jazyk nejnižší úrovně umožňuje vytvářet takové typy, které jsou v rozporu se základními požadavky rozumného a bezpečného programování. Technika programování je obor, který už má své tradice. Není proto vhodné ani nutné, aby každý, kdo začíná programovat, opakoval chyby svých předchůdců.

Nejsnadnější počáteční přístup k programování umožňují vyšší, problémově orientované jazyky (Basic, Fortran, Algol, Pascal . .). V podstatě jsou založeny na tom, že jejich implementace, ve srovnání s JSA, vytvářejí virtuální počítač s mnohem účinnějším souborem příkazů, přímo pracující s určitým souborem vyšších datových typů. Tyto jazyky jsou „strojově nezávislé", a tak, včetně další podpory, oprošťují programátora od triviálních problémů a technických prostředků konkrétní implementace. I když se to vše děje na úkor některých, mnohdy důležitých parametrů (rychlost zpracování, chování v reálném čase, nároky na kapacitu paměti . .), je výhodnost zahájení studia programování vyšším programovacím jazykem mimo diskusi. Zbývá odpovědět na otázku, který z dostupných jazyků je nejvhodnější.

BASIC

Prakticky na všech levných a každému dostupných domácích mikropočítačích je standardně implementován BASIC. Je to nejrozšířenější programovací jazyk, se kterým začínala pracovat většina současných programátorů. Od počátku byl koncipován tak, aby to byl jazyk co možno nejjednodušší, ale současně univerzální, použitelný i při vědeckotechnických aplikacích, aby kladl minimální požadavky na odbornou přípravu či specializaci uživatele (syntaxe, sémantika) a při tom byl snadno implementovatelný na libovolné typy počítačů. Toho bylo dosaženo mimo jiné omezením rozsahu užívaných příkazů a především datových typů, které jsou v BASIC tvořeny pouze znakovými a číselný-

mi konstantami a proměnnými, implicitně vyjádřenými ve formátu pohyblivé čárky. Na rozdíl od jiných vyšších jazyků tak byly omezeny požadavky na typovou kontrolu správnosti přiřazení příkazů a dat, což z hlediska uživatele znamená především výrazné zjednodušení vytvářeného programu tím, že je zcela vynechána jeho deklarační část. Jak však uvidíme později, je zde zároveň ukryta jedna z hlavních slabin jazyka BASIC.

Zmíněná koncepce BASIC umožňuje zápis, edici i ladění vytvářeného programu ve velmi příjemném dialogovém režimu – každý příkazový řádek je při jeho zápisu z konzoly počítače předzpracován a syntakticky kontrolován. Na případné chyby je uživatel upozorněn a dokud je neopraví, je mu znemožněno řádek odeslat. Zápis programu se děje v edičním módu. Zdrojový text programu v BASIC, uložený ve vyhrazené oblasti operační paměti, je samozřejmě třeba přeložit do strojového kódu mikropočítače. V uvažovaném případě pracuje příslušný překladač jako tzv. interpret, kdy se jednotlivé příkazové řádky definitivně překládají až v průběhu provádění spuštěného programu, tedy v prováděcím režimu činnosti. Pro opravy sémantických chyb se opět užívá automatického přechodu do edičního módu v místě detekované chyby, ve zvoleném místě, nebo je možné použít i trasování.

Uvedené vlastnosti patří k hlavním důvodům velké obliby jazyka BASIC, k níž přispívá i instalace grafického modulu na převážné většině i těch nejjednodušších domácích mikropočítačů. Jsou však i hlavní příčinou jeho nedostatků. Odhlédneme-li zatím od pomalosti v důsledku interpretačního překladu, je jeho hlavní slabinou to, že nerespektuje principy a zásady moderního, strukturového programování. U BASIC zcela postrádáme prostředky pro hierarchicky vázanou výstavbu programu pomocí programových modulů a tedy i vnější procedury a funkce, deklarace lokálních proměnných a vůbec možnost pracovat se složitějšími datovými strukturami vyjma polí. Proto běžný Micro nebo Minimal BASIC není jako průprava na soustavnou práci s jinými jazyky včetně asemblerů mikroprocesorů a mikrokontrolerů tím nejvhodnějším prostředkem. I když je jistě pravda, že vždy je lepší programovat nějak než vůbec ne, je důkazem správnosti předchozího tvrzení jistě i to, že poslední verze standardního jazyka BASIC již některé prvky, podmiňující výstavbu strukturovaného programu, od novějších a modernějších jazyků přebírají.

Principy strukturovaného programování

Každý rozsáhlejší program vždy představuje řešení složitého problému, čemuž odpovídá i jeho omezená přehlednost. V průběhu vývoje programovacích technik se velmi brzo ukázala potřeba zavést do programátorské činnosti určitý systém a to hned od počátku návrhu hrubé koncepce a struktury programu. Jedině tak je možno v průběhu prací udržet přehled o všech vzájemných vazbách uvnitř programu a zajistit jeho efektivní návrh, ladění, dokumentaci a údržbu. Postupně byly stanoveny určité zásady, podporující systematické programování a promítající se i ve filozofii programovacích jazyků. Dochází tak k určitému sjednocování programátorského myšlení a tedy i ke snadnějšímu vzájemnému dorozumívání, protože principy strukturovaného programování jsou založeny na omezení neefektivních intuitivních přístupů k tvorbě programu na všech úrovních.

Modulární koncepce

Návrh programu vždy začíná analýzou řešeného problému, přičemž se hledá jeho optimální programové řešení. Je to vlastně nejdůležitější a nejobtížnější úsek celé práce, protože určuje koncepci celého řešení. Každá úloha má vždy několik řešení, která jsou často velmi složitá a nepřehledná. Intuitivní přístup již od počátku návrhu, který bývá vícekrát přepracováván, než je nalezeno vhodné řešení, je pracný. Mimoto vždy hrozí nebezpečí, že v průběhu dalších, detailních prací bude třeba udělat takové úpravy v programu, které se mohou projevit katastrofickým vlivem na jeho již hotové úseky.

Efektivní metodu návrhu řešení programu představuje hierarchický rozklad úlohy tzv. *metodou shora dolů* (top-down). Analogii tohoto systematického postupu můžeme spatřovat např. v návrhu elektronického zařízení (koncepce, blokové schéma, detailní schéma . .), který také probíhá na různých hierarchických úrovních, z nichž každé přísluší určitá úroveň abstrakce.

Hierarchický rozklad programového řešení úlohy má za výsledek jeho rozložení do přesně vymezených programových modulů s přesně definovaným systémem vzájemných vazeb. Jednotlivé moduly příslušejí různým hierarchickým úrovním (např. podle obr. 1). Je důležité uvědomit si, že na rozdíl od známějších vývojových diagramů hierarchický diagram nepostihuje průběh prováděného programu, ale funkce a vzájemné vazby všech programových modulů v rámci programu, jehož celková struktura se tak stává přehlednou. Moduly vyšších vrstev ovládají moduly vrstev nižších. Tak např. na obr. 1 nejvyšší, řídicí modul M, který si můžeme představit jako hlavní program, řídí a pro svoji potřebu využívá služeb jemu podřízených modulů nižší vrstvy, které opět disponují moduly další, nižší úrovně. Nejnižší hierarchické úrovni pak odpovídají moduly, které již nevyžadují dalšího rozkladu.

Vzájemná vazba jednotlivých modulů může pro zabezpečení hierarchické orientace probíhat pouze ve vertikálním směru a pouze mezi moduly sousedních vrstev, např. modul M_A může komunikovat pouze s moduly M, A_1, A_2, A_3. Není možná jeho přímá vazba s M_B nebo A_{21}. Z tohoto hlediska má tedy každý modul jediný vstup a jediný výstup. Volaný modul po ukončení činnosti vždy vrací řízení programu tomu modulu, který jej vyvolal.

Výhodou hierarchického rozkladu není jen přehlednost návrhu celé programové struktury na různých úrovních jednotlivých modulů. Její přesné vymezení také umožňuje v podstatě nezávislé detailní řešení jednotlivých, funkčně i systémem vazeb přesně specifikovaných modulů. Rozsáhlé programy, často zpracovávané současně několika programátory, by jinak mohly být řešeny jen těžko.

Prostředky, umožňující modulární výstavbu programu, představují ve vyšších, strukturovaných jazycích procedury a funkce jako zvláštní typy podprogramů, umožňující vymezit potřebné hierarchické vazby mezi moduly.

Modulární koncepce přináší i další výhody. Zdaleka ne vše, co programátor ve svých programech užívá, je schopen pouze sám nebo stále znovu vymýšlet. To, co pro hardwarového specialistu znamená jeho knihovna standardních nebo ověřených řešení a zapojení, to v softwarové oblasti představují standardní a uživatelské rutiny, které mohou být do modulárně vytvářeného programu implementovány jako programové moduly buď přímo, nebo po příslušných úpravách.

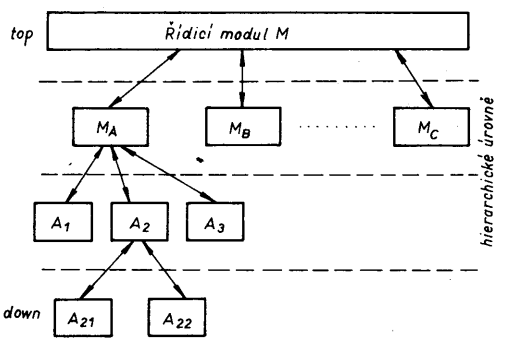
Příkazové a datové struktury

Hierarchicky koncipovaný program se tedy skládá z programových modulů, systematicky vytvářených tak, aby byla zajištěna jejich přehledná a bezpečná součinnost. Moderní programovací jazyk však musí mít k dispozici také prostředky pro uplatnění metod bezpečného, strukturového programování i uvnitř těchto modulů.

Provádění každého programu vždy znamená nutnost zpracovat určitá data posloupností vhodně volených příkazů užitého programovacího jazyka, přičemž data i příkazy mohou být různého typu. To však vždy znamená nebezpečí zavedení chyb a tedy i návrhu špatného programu. Ať už se to týká nesprávného přiřazení a zpracování datových typů, čehož důsledkem jsou nedostatečně přesné výsledky, nebo návrhu špatně nebo vůbec nepracujícího (zhroucení) programu. Bohužel, většina podobných nedostatků v činnosti vytvářených programů se zjišťuje velmi obtížně, protože se často vyskytují jen za určitých, těžko postižitelných podmínek. Odhalení všech syntaktických a sémantických chyb, které do návrhu programu může programátor zavést, vyžaduje účinný systém kontrol.

Strukturované jazyky minimalizují nebezpečí výskytu syntaktických chyb jak vlastní koncepcí, tak systémem kontrol v průběhu překladu zdrojového textu programu. Základem je omezení počtu příkazů, kterými jazyk disponuje, na vybrané typy a struktury, umožňující při dodržení určitých zásad vytvářet pouze bezpečné programové konstrukce, což do jisté míry omezuje volnost práce programátora. Dobrou představu o tomto vlivu umožňují každému dobře známé vývojové diagramy.

Vývojový diagram je velmi názornou pomůckou při prvních kontaktech s problematikou programování. Jeho největší předností je to, že detailně postihuje všechny možnosti průběhu zpracování programu a především jeho větvení na základě testů veškerých odmínek. To však lze současně z jiného



Obr. 1. Strukturované programování podporuje hierarchický rozklad programu a vazbu jednotlivých programových modulů (viz dále procedury, globální a lokální deklarace)

hlediska považovat i za jeho největší slabinu. Vyplývá z toho, že rozhodovací bloky v součinnosti s přiřazovacími a jinými příkazy mohou za různých podmínek vytvářet příkazové konstrukce se skrytými, vedlejšími důsledky. Jejich odhalení je vzhledem k relativní nepřehlednosti vývojových diagramů z tohoto hlediska mnohdy velmi obtížné.

Vývojový diagram, jak přímo vyplývá z principu jeho výstavby a původního smyslu určení, nijak nepodporuje principy bezpečného, strukturovaného programování, ale právě naopak. Umožňuje intuitivní vytváření nebezpečných příkazových konstrukcí.

Strukturované programovací jazyky užívají omezený, vybraný sortiment jednoduchých a stukturovaných příkazů, jejichž prostřednictvím však lze vytvořit všechny potřebné příkazové konstrukce bez zmíněných nežádoucích účinků. Všechny tyto příkazy jsou opět, jako programové moduly, charakteristické tím, že mají jediný vstup a jediný výstup. Již samotná koncepce strukturovaného programovacího jazyka tak omezuje nebezpečí vytváření nevhodné struktury programu podstatnou měrou.

Obdobné souvislosti nacházíme u strukturovaných jazyků i v oblasti užívaných datových typů a struktur. Jejich sortiment, práce s nimi i systém kontrol jsou voleny tak, aby bylo dosaženo co nejvyšší efektivnosti a bezpečnosti programování. Vedle implicitně definovaných, jednoduchých datových typů různých formátů lze zpravidla užívat i vyšší uspořádané datové struktury, např. pole (Array), záznam (Record) nebo soubor (File), výhodné z hlediska efektivního zpracování hromadných dat.

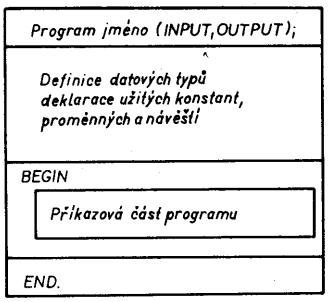
Domníváme se, že nyní už jsou přednosti strukturovaného programování zřejmé. Protože většinu zásad, které lze odvodit, je velmi užitečné aplikovat nejen ve většině vyšších programovacích jazyků, ale i v asembleru, popíšeme si v následující kapitole hlavní charakteristické rysy jazyka Pascal, který byl původně vyvinut jako školní jazyk pro výuku strukturovaného programování. Jeho pozdější hromadné rozšíření vyplývá ze systematické, přehledné a logické výstavby, vyvážené koncepce datových a příkazových struktur a účinného systému kontrol. Za zmínku stojí i to, že během času byla od jazyka Pascal odvozena i řada dalších moderních jazyků, např. Modula nebo Ada. I když tedy na několika stránkách můžeme těžko postihnout víc než pouze rámcový přehled, myslíme si, že právě ten může při prvních krocích vzhledem ke zhuštěné formě napomoci podstatnou měrou. Podrobnější informace pak lze čerpat z citované literatury a firemních manuálů.

Stručný přehled jazyka Pascal

Poznamenejme úvodem, že celá struktura jazyka Pascal je velmi systematická, což se zpočátku může jevit až jako příliš násilné. Nicméně, tento charakteristický rys je velmi užitečný a nutný vzhledem k možnostem práce s hierarchicky členěnými „moduly". Pascal sám ovšem pojem programového modulu obecně nespecifikuje, chápe celý program jako blok od hlavičky až po závěrečnou tečku.

Struktura pascalského programu

Zdrojový tvar (zápis) pascalského programu, viz obr. 2, se skládá ze čtyř hlavních částí: Záhlaví, definiční a deklarační části, vlastní programové části a ukončení.



Obr. 2. Schematické znázornění struktury pascalského programu

Hlavička programu se skládá ze jména programu a symbolického označení vstupních/výstupních souborů, s nimiž program pracuje. Toto označení souborů nám zatím nemusí plést hlavu, protože zejména u levných domácích počítačů bez vnějších diskových pamětí není povinné. Hlavička programu slouží pro jeho identifikaci.

Deklarační část obsahuje deklarace všech užitých konstant a proměnných, pojmenovaných symbolickými jmény (identifikátory), volenými programátorem. Pro definice a deklarace strukturovaných typů se užívá postupného popisu s využitím typů nižších úrovní. V této části se deklarují i návěští, procedury a funkce.

Příkazová část tvoří tělo programu, v němž se podle platných syntaktických pravidel jazyka, za pomoci příkazů, konstant, proměnných a výrazů zapisuje vlastní programová struktura. Textové řetězce se zapisují mezi apostrofy. Každý příkazový řádek, který však ve skutečnosti může zabírat i několik řádků fyzických, musí být ukončen středníkem. Návěští se oddělují dvojtečkou. Celá příkazová část je uvozována otevíracím klíčovým slovem begin. Obdobně složené programové bloky, sekvence, procedury, cykly apod. jsou ohraničovány svorkami z klíčových slov begin . . . . end, které posuvem v řádcích textu (zleva doprava) zpřehledňují zápis celého programu.

Program je ukončován *klíčovým slovem* end, tvořícím protějšek uvozovací svorky begin celé příkazové části, a tečkou, označující konec programu pro překladač.

Pro zpestření a vytvoření vlastního prvotního názoru je v tab. 1 uveden jako příklad jednoduchý pascalský program, který dává hádat určité zvolené číslo, předtím tajně zadané z klávesnice. Pro průběžnou orientaci hráče je po každém pokusu na displeji vypsáno hlášení, udávající, je-li správná hodnota tohoto čísla větší nebo menší než číslo, jím právě zadané. Počítá se počet pokusů, potřebných k uhádnutí zadaného čísla. Program byl, stejně jako všechny další ukázky, napsán a odladěn na Spectru v Hisoft Pascalu 4T.

Z uvedené ukázky, které zřejmě zatím zcela úplně rozumět nebudeme, je přesto vidět srozumitelná a přehledná konstrukce jazyka a výstavby programu. Na konci této kapitoly si zkuste sami nakreslit vývojový diagram nebo strukturogram programu, zkoušejte jej různě modifikovat (meze zadaného čísla, využití návěští, kritéria, způsoby vyhodnocení výsledků . . .)

Tab. 1. Program monitorující hádání tajně zadaného celého čísla

```
PROGRAM DELFY;
LABEL 1,2;
VAR I,K,L,N:=INTEGER;
BEGIN
1:K:=0;
WRITELN('ZADEJ  LIBOVOLNE  CELE  CISLO  0...1000  A
PREDEJ POCITAC HRACI :');
READ (L);
IF L>1000 THEN GOTO 1;
FOR I:=1 TO 24 DO WRITELN;
WRITELN ('HADEJ CISLO 0...1000 ');
WRITELN ('**********************************************');
REPEAT
  WRITELN;
  WRITELN;
  WRITELN ('ZADEJ HADANE CISLO :    ');
  READ (N);
  K:=K+1;
  IF N>L THEN WRITELN ('SPRAVNE CISLO JE MENSI');
  IF N<L THEN WRITELN ('SPRAVNE CISLO JE VETSI');
UNTIL N=L;
WRITELN ('**********************************************');
WRITELN;
WRITELN ('  ******* !!! Z A S A H !!! *******');
WRITELN ('**********************************************');
IF K=1 THEN WRITELN ('NA PRVNI POKUS')
ELSE BEGIN
       WRITELN ('POTREBOVAL JSI',K,'POKUSU');
       IF (K>1)AND(K<4) THEN WRITELN
       ('VYNIKAJICI VYSLEDEK')
     END;
IF K<4 THEN GOTO 2;
IF  (N<100)AND(K=4) OR (N>=100)AND(N<300)AND(K<8) OR
(N>=300)AND(K<10) THEN WRITELN ('TO JE DOBRY VYSLEDEK')
ELSE WRITELN ('CHCE TO JESTE TRENING!');
2:WRITELN;
WRITELN ('POKRACOVANI VE HRE:   ZADEJ R(UN),ENTER')
END.
```

Typy a struktury pascalských příkazů

Příkazová struktura Pascalu se skládá z jednoduchých a strukturovaných příkazů.

K jednoduchým příkazům patří přiřazovací příkaz, příkaz volání procedury a příkaz skoku. Jejich použití je jednoduché a přímo vyplývá z dalších ukázek. Proto se jimi zabývat nebudeme. Je však nutné upozornit na nebezpečí, které přináší příkaz skoku (goto) tím, že umožňuje přechod k provádění programu na libovolné jeho místo. Tím může být zcela narušena struktura programu jak v rámci „modulu", tak i celého programu, což může mít katastrofální důsledky. Příkaz goto je schopen narušit zásady strukturovaného programování. Je tedy nestandardní a jeho užívání je omezeno na speciální případy.

Bylo dokázáno, že pro tvorbu libovolné příkazové konstrukce lze vystačit v podstatě se třemi standardními, strukturovanými typy příkazů – sekvencí, alternativou a iterací.

Sekvence znamená jednoduše, postupně řadit dílčí operace za sebou tak, že vytvářejí ucelenou složitější akci. V příkazovém pojetí to znamená možnost využít jak jednoduchých příkazů, tak i dalších dvou typů příkazů strukturovaných. Vytvořená složená sekvenční struktura musí být ohraničena vnějšími svorkami z klíčových slov begin . . . end.

Alternativa znamená podmíněně vybírat jednu operaci na základě splnění nebo nesplnění podmínky. Základní formát příkazové struktury má první možný tvar

if p then P1 else P2

přičemž p = podmínka, P1, P2 = alternativní příkazy.

Nejprve se vyhodnocuje výraz, uvedený v příkazu jako podmínka. Při splnění podmínky (p = true) se realizuje příkaz P1, v opačném případě (tj. při p = false) alternativní příkaz P2. Podmínka je logického typu, výraz proto nemůže poskytovat žádnou jinou možnost vyhodnocení. Vývojový diagram tohoto příkazu je na obr. 3a.

Druhý možný a užívaný tvar příkazu tohoto typu, obr. 3b,

if p then P

realizuje při splnění podmínky příkaz P, v opačném případě (p = false) nastává přeskok a tedy ukončení celého podmíněného příkazu bez jakékoli aktivní činnosti.

Zvláštním případem alternativního výběru činnosti je příkaz case. Jedno z možných větvení se vybírá prostřednictvím tzv. selektoru. Příkaz case, obr. 4, má tvar

```
case SEL of
   const1 : P1;
   const2 : P2;
      –
      –
   constN : PN
end
```

Výraz, který určuje hodnotu selektoru, musí být ordinálního, tj. celočíselného nebo znakového typu. Po vyhodnocení selektoru se provádí ten příkaz, který odpovídá právě tomuto selektoru. Pokud však vyhodnocený selektor neodpovídá žádné z definovaných konstant, příkaz case se neuplatní a navíc je tato situace zpravidla indikována chybovým hlášením.

Iterace znamená opakované provádění příkazu na základě vyhodnocení určité podmínky. Opakované provádění příkazu nebo složeného příkazového bloku umožňují strukturované příkazové cykly. Podle způsobu uplatnění nebo stanovení podmínky opakování se příkazy cyklů dají dělit do dvou částí:

– while, repeat pro podmíněné cykly,
– for pro cykly s nastavením počtu opakování.

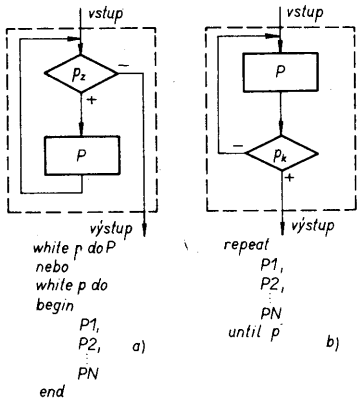
Příkaz cyklu s testem zahájení, obr. 5a, má tvar

while p do P.

U tohoto typu cyklu se nejprve vyhodnocuje výraz, určující opakovací podmínku. Je-li p = true, provádí se příkaz P, pak následuje opět vstup na začátek cyklu a znovu se vyhodnocuje podmiňující výraz atd. Cyklus je proto ukončen tehdy, je-li při vstupu do cyklu vyhodnocena podmínka p = false.

Má-li být v těle cyklu užito více než jednoho příkazu, musí vytvořená posloupnost být zapsána jako strukturovaná sekvence a jako taková uzavřena do svorek begin – end.

Všimněme si ještě, že u cyklu typu while nemusí být, při počáteční platnosti podmínky p = false, příkazová část cyklu provedena ani jednou. Říkáme, že za takové podmínky ke vstupu do cyklu vůbec nedojde.



Obr. 5. Zpětnovazebním uspořádáním příkazů větvení a přiřazení vznikají podmíněné cykly. Pascal definuje dva strukturované podmíněné cykly s jediným a tím bezpečným vstupem a výstupem: a) podmíněný cyklus s testem zahájení while p do P, b) podmíněný cyklus s testem ukončení repeat P until p

Příkaz cyklu s testem ukončení, obr. 5b, má tvar

```
repeat
      příkazová posloupnost P1 až PN
until p
```

Již ze zápisu je patrno, že do těla cyklu se vstupuje v každém případě, nezávisle na podmínce. Provedou se poprvé všechny příkazy, uvedené v příkazové části a teprve potom se vyhodnocuje platnost podmínky. Cyklus se opakuje tak dlouho, dokud (until) není splněna podmínka ukončení. Ta vyplývá z logického vyhodnocení podmiňujícího výrazu na konci každého průchodu cyklem. K výstupu z cyklu tedy dochází při splnění podmínky p = true. Vidíme, že u tohoto typu příkazu nastává průchod příkazovou částí vždy, minimálně jedenkrát.

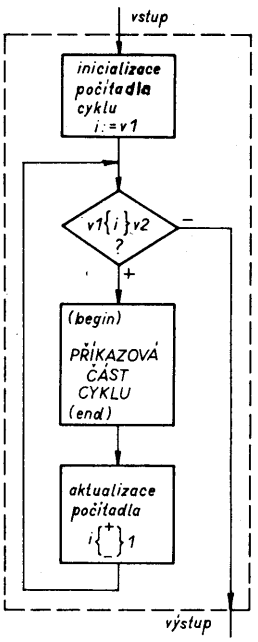
Příkaz nepodmíněného cyklu

Provádění iterace se zvoleným, explicitně nastavitelným počtem opakování umožňuje nepodmíněný příkaz cyklu typu

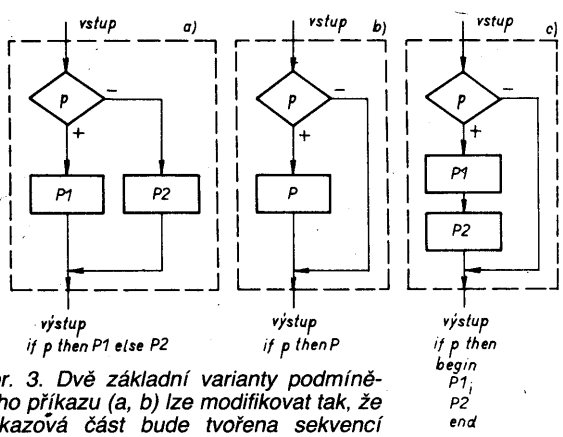
for i:=v1 { to / downto } v2 do

```
      begin
            příkazová posloupnost
      end,
```

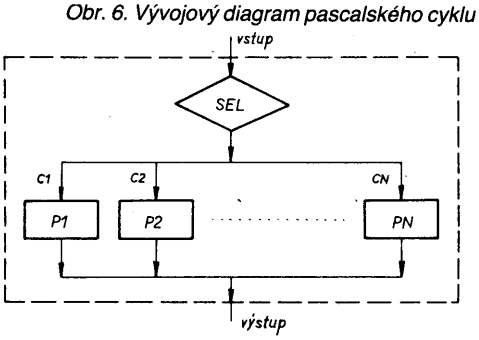
v němž i = řídicí proměnná cyklu, v1 a v2 = konstanty, výrazy nebo proměnné ordinálního typu, určující mezní (počáteční a konečné) stavy počitadla cyklu. Alternativní klíčová slova to/downto určují funkci (vpřed/vzad) tohoto počitadla. V případě, že tělo cyklu není tvořeno jednoduchým, ale složeným příkazem, musí být opět ohraničeno klíčovými slovy begin – end.



Obr. 6. Vývojový diagram pascalského cyklu FOR



Obr. 3. Dvě základní varianty podmíněného příkazu (a, b) lze modifikovat tak, že příkazová část bude tvořena sekvencí ohraničenou svorkami BEGIN-END; viz příklad c)



Obr. 4. Diagram příkazu několikanásobného větvení CASE

Graficky je průběh cyklu for znázorněn na obr. 6. Jeho provádění začíná přiřazením počáteční hodnoty řídicí proměnné i:= v1. V případě úspěšného testu okrajových podmínek (tj. u cyklu to při v1<v2, u cyklu downto při v1>v2 je vyhodnocena podmínka p = true a provádí se poprvé příkazová část cyklu, inkrementuje nebo dekrementuje se stav počitadla cyklů. Tím je ukončena příprava pro následující cyklus, přechází se opět na vstupní test. Cyklus se opakuje tak dlouho, dokud nebude vyhodnocena podmínka p = false. To nastane u cyklu typu to při i>v2, u typu downto při i<v2. Proto, při nevhodných podmínkách už při prvním vstupu do testu (viz výše) se může stát, že průchod cyklem vůbec nenastane, celý příkaz se chová jako prázdný.

Porovnáme-li pascalský tvar cyklu for např. s jeho basicovým protějškem, vidíme především, že se neuplatňuje volba kroku cyklu (step) a zvláště explicitní užívání klíčového slova pro návrat do cyklu (next). Obojí souvisí s programovou bezpečností, se zavedením systému strukturovaných dat a rychlostí provádění cyklu (vyloučením typu real z přípustných, ordinálních typů řídicí proměnné i).

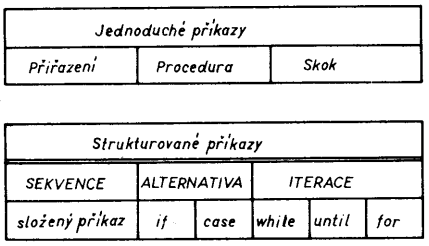
Pro názornost si uveďme jednoduchý příklad použití cyklu for pro výpis libovolného úseku písmen v abecením pořadí.

```
PROGRAM ABECEDA;
VAR I : CHAR;
BEGIN
FOR I:= 'I'TO 'N' DO
    WRITE(' ',I)
END.
```

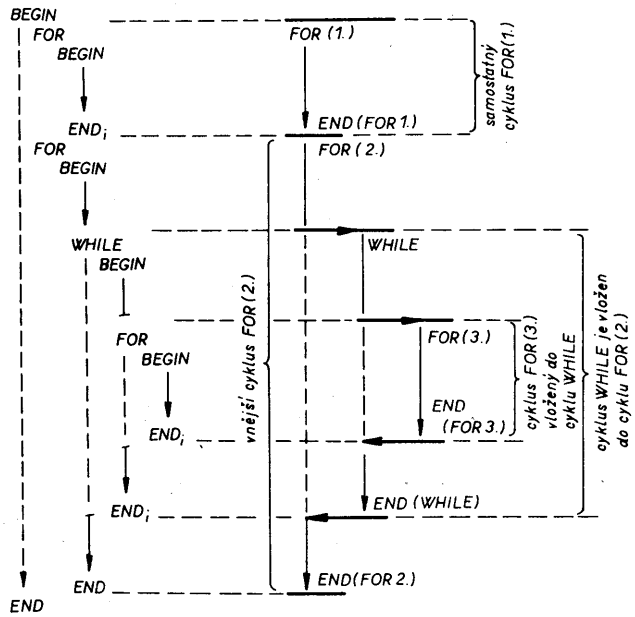
Program využívá ordinálního uspořádání typu char. Každý znak kromě prvního a posledního má svého předchůdce a následníka. V úvodu programu je deklarována řídicí proměnná cyklu I jako typ char. Mezní hodnoty počitadla cyklu jsou udány znakovými konstantami tak, že rovnou označují krajní znaky vybraného úseku abecedy, který chceme znázornit. Řídicí proměnná se po každém průchodu cyklem inkrementuje ordinálním přírůstkem (+1) a protože je typu char, vypisují se její jednotlivé znaky v cyklu opakovaným příkazem write. Každému znaku vždy předchází mezera (' '). Když zapíšeme v příkazu cyklu např. stejné meze, jaké jsme uvedli, vypíše se text I J K L M N. Zkuste si, pokud máte k dispozici počítač, i jiné meze nebo opačný typ cyklu (downto).

Cykly představují pro programátora velmi účinný nástroj. Mohou být užívány i tak, že jeden je vždy vložen do druhého, přičemž hloubka vložení není omezena. Každý vložený cyklus pochopitelně musí být před návratem do vnějšího ukončen, protože každá strukturovaná příkazová konstrukce musí mít jediný vstup i výstup (obr. 7).

Můžeme říci, že jsme stručně prošli všechny rozhodující pascalské příkazy tak, abychom se mohli orientovat v konkrétních programech. Mnohé z nich jsme ovšem opomenuli, příkladem může být třeba příkaz přiřazení – to proto, že je považujeme buď za dostatečně srozumitelné, nebo naopak příliš speciální. Protože se však nechceme stát specialisty na Pascal, domníváme se, že náš přístup je rozumný. Za důležitější považujeme možnost poučit se z výstavby tohoto jazyka pro programování v asembleru, kde nestrukturovaná větvení běhu programu na základě nedomyšlených testů, hierarchicky nekontrolované užívání skokových instrukcí a nakonec i podprogramů představují velké nebezpečí. Vidíme už také, že začínat s programováním na úrovni asembleru není možné. Je nutné respektovat a využívat základní poznatky z výstavby strukturovaných jazyků jako je Pascal, včetně rozkladu programového řešení do jednotlivých programových modulů. Teprve po určitém, svým způsobem abstraktním zvládnutí vhodné koncepce modulu má smysl přecházet k jeho konkrétnímu řešení na úrovni asembleru – s tím, že prostřednictvím instrukčního souboru vytvářené příkazové typy by se měly blížit příkazové struktuře virtuálního pascalského počítače, přehledově zachycené na obr. 8.



Obr. 8. Hrubé rozdělení typů pascalských příkazů

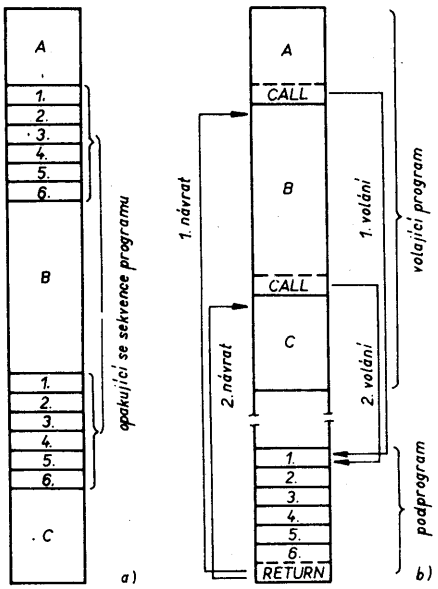


Obr. 7. Symbolické programové a grafické znázornění konstrukce a průběhu zpracování hierarchie vložených cyklů

Aby byl programovací jazyk efektivní a bezpečný, musí ovšem umožňovat mnohem víc, než práci s efektivní příkazovou strukturou. Musí dále především disponovat vhodnými prostředky pro výstavbu hierarchicky členěného programu.

Podprogramy

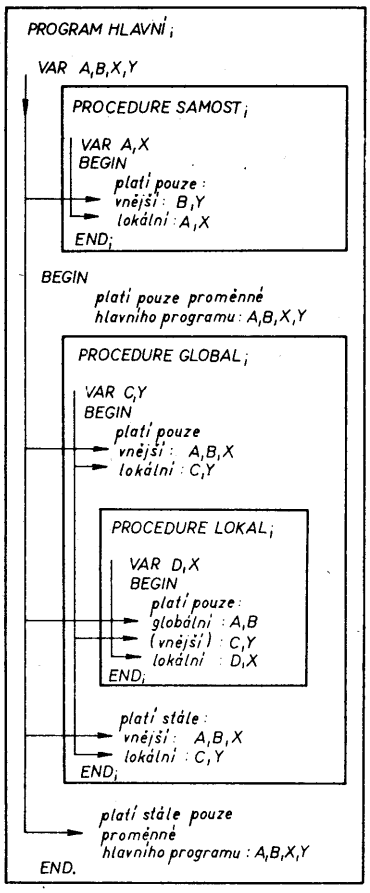
Podprogramy se běžně používají ve všech programovacích jazycích, tedy např. i v asembleru nebo v BASIC. Jejich původním posláním bylo usnadnit práci programátorovi tehdy, kdy by musel určitý blok v programu znovu a znovu mnohokrát opakovat. Z tohoto hlediska usnadní orientaci obr. 9, porovnávající potřebu paměťové kapacity při výstavbě programu bez a s využitím podprogramů, protože na první pohled uvedený postup šetří nejen práci, ale i místem programové paměti. Ovšem, jak obr. 9 naznačuje, úspora paměťových nároků se výrazněji projeví jen tehdy, je-li volaný podprogram dostatečně rozsáhlý. Velmi často ovšem užívání podprogramů přináší více problémů, než užitku.



Obr. 9. Porovnání programové výstavby s opakujícími se bloky (a) a s využitím prostředků volání a návratů z podprogramů

Klasický podprogram představuje uzavřený program, který v principu může být volán z libovolného místa hlavního programu. Volající program ovšem musí podprogramu nějakým způsobem předat data (parametry), které má zpracovat. Musí se zabezpečit uložení návratové adresy, obsahy proměnných či registrů před možnou destrukcí atd. Podprogram pak po ukončení své činnosti předává znovu řízení programu, který jej vyvolal. Tomu také musí předat případná zpracovaná data. To vše je samozřejmě časově náročné a navíc, jednoduše koncipovaná struktura podprogramu narušuje koncepci strukturovaného programování. I tak je však předávání parametrů v nižších jazycích a zvláště v asembleru obtížné. Proto se mnohdy vyplatí od užívání malých podprogramů ustupovat, protože kapacita paměti přestává u mikropočítačů hrát dominantní roli. Tu naopak přebírá systematičnost strukturované výstavby programového vybavení.

Podprogramy vyšší úrovně představují v jazyku Pascal procedury a funkce, kterých se užívá jako modulů podle zásad hierarchické vertikály. Jejich výstavba je podporována konvencí užívání vnějších a vnitřních (lokálních) proměnných, míněno ve vztahu k příslušnému modulu. Ve svém „modulu"



Obr. 10. Deklarace, platnost a zastiňování proměnných (globálních a vnějších) v jednoduchých a vnořených procedurách

je lokální proměnná schopna zastínit význam případné stejnojmenné vnější nebo globální proměnné, tedy např. proměnné volajícího modulu. Jsou-li v nadřazeném, volajícím programu deklarovány proměnné A, B, X, pak, pokud volaný podprogram (procedura) užívá lokální stejnojmenné proměnné A, X, vnější proměnné A, X se v proceduře neuplatní. Proměnná B ovšem zastíněna není (obr. 10).

Skutečný, hierarchicky členěný pascalský program tedy pro svoje hierarchické členění užívá procedur a funkcí, které mohou tvořit samostatné moduly programu. Mohou však být také vhnízděny, vnořeny jeden do druhého.

Procedury

Struktura zápisu procedury je v podstatě shodná se zápisem běžného programu. Skládá se opět z hlavičky, deklarační a příkazové části. Na rozdíl od programu se však ukončuje středníkem.

Hlavička procedury se skládá z klíčového slova procedure, jejího symbolického jména (identifikátoru) a seznamu formálních parametrů. Pole formálních parametrů není povinné, protože pochopitelně existují i procedury bez parametrů. Příklad:

```
PROCEDURE CVICNA (
                  VAR PROD1: INTEGER;
                      HODN: INTEGER;
                  VAR PROD2: INTEGER);
```

Ze zápisu hlavičky vidíme, že procedura CVICNA má tři formální parametry, všechny typu integer. Formální parametry se nahrazují skutečnými parametry shodného typu při vyvolání procedury.

Užívá se dvou typických způsobů předání parametrů:
– hodnotou,
– odkazem (referencí).

Formální parametry předávané hodnotou se v hlavičce blíže nespecifikují, parametry předávané odkazem se uvozují klíčovým slovem var.

V popisné části procedury se v případě potřeby deklarují užité lokální proměnné, které existují pouze po dobu provádění procedury. Lokální proměnné nemají při vyvolání procedury přiřazeny hodnoty a nemohou být nijak využity vnějším programem, protože jsou pro něj nepřístupné.

Tělo procedury tvoří běžná příkazová struktura, užívající formální parametry specifikované v hlavičce a příslušné lokální konstanty a proměnné.

Celý formální zápis procedury je popisem její činnosti. Procedura je aktivována teprve vyvoláním z vnějšího programu, které se provádí zápisem příkazového řádku volání procedury s uvedením identifikátoru a seznamu skutečných parametrů, např. CVICNA (A, A, B). Typy, pořadí i počet skutečných parametrů musí být ve shodě se specifikací parametrů formálních.

Obě metody předávání parametrů vyplývají jednak z lokálního charakteru procedury (uzavřený podprogram) a tedy i dočasného charakteru jejích proměnných, které s ukončením jejího zpracování zanikají, jednak z potřeby zajištění hierarchické bezpečnosti programové struktury.

Uvažujme nejprve předání skutečného parametru hodnotou. Při vyvolání procedury se nejprve po vyhodnocení parametru formálního (např. HODN: INTEGER) vytvoří uvnitř procedury pomocná lokální proměnná, do které se následně přenese hodnota skutečného parametru, specifikovaného při volání. Již na počátku zpracování příkazové části procedury má tedy pomocná proměnná přiřazenu počáteční hodnotu skutečného předávaného parametru. Dále jsou již skutečný parametr vnějšího programu a lokální parametr uvnitř procedury vzájemně závislé, neovlivňující se. Z toho také vyplývá, že předání parametru hodnotou je možné pouze směrem z vnějšího programu do procedury, ale ne naopak.

Pro výstup parametrů z procedury se užívá jejich předání odkazem. Rozdíl je v tom, že při vyvolání procedury nepředávají vnější proměnné jako parametr své hodnoty, ale adresy, na kterých jsou uloženy. Procedura proto při následném zpracování může obsah této pro ni vnější proměnné ovlivňovat.

Protože pochopení mechanismu předávání parametrů vyžaduje hlubší zamyšlení a má mimořádný význam, projdeme si nyní podrobněji velmi jednoduchý školní příklad programu s využitím procedury. Záměrně jsme zvolili stejný příklad, jaký je uveden v [2], str. 77, který jsme pro přehledný postih doplnili o výpis obsahů jednotlivých proměnných tak, jak jsou včetně činnosti při vyvolání procedury postupně přiřazovány vnějším i lokálním parametrům. Výpis programu je v tab. 2.

Realizace programu začíná přiřazením obsahu globální proměnné (A) = 5, obsah druhé globální proměnné B zatím zůstává nedefinovaný. V této fázi zatím můžeme sledovat pouze tyto proměnné, protože program probíhá mimo proceduru.

Následuje první vyvolání procedury se skutečnými parametry, CVICNA (A, A, B). Nejprve se tedy nahrazují formální parametry skutečnými. V našem případě je předáván jeden parametr hodnotou, dva odkazem.

Tab. 2. Demonstrační program předávání parametrů procedury

```
PROGRAM HDNODK;
VAR A,B:INTEGER;
PROCEDURE CVICNA (VAR PROD1:INTEGER;HODN:INTEGER;
                  VAR PROD2:INTEGER);
BEGIN
  WRITELN (A:6,B:6,PROD1:6,HODN:6,PROD2:6);
  PROD1:=PROD1+1;
  WRITELN (A:6,B:6,PROD1:6,HODN:6,PROD2:6);
  HODN:=HODN-1;
  WRITELN (A:6,B:6,PROD1:6,HODN:6,PROD2:6);
  PROD2:=HODN;
  WRITELN (A:6,B:6,PROD1:6,HODN:6,PROD2:6);
END;
BEGIN
  A:=5;
  WRITE (' A  ',' B  ',' PROD1', HODN ',' PROD2');
  WRITELN;
  WRITELN (A:6,B:6);
  CVICNA (A,A,B);
  WRITELN (A:6,B:6);
  CVICNA (A,A-3,A);
  WRITELN (A:6,B:6)
END.

RUN
  A     B     PROD1  HODN  PROD2
     5 25838
     5 25838     5     5 25838
     6 25838     6     5 25838
     6 25838     6     4 25838
     6     4     6     4     4
     6     4
     6     4     6     3     6
     7     4     7     3     7
     7     4     7     2     7
     2     4     2     2     2
     2     4
```

Proto je přiřazena jediná hodnota, odpovídající obsahu vytvořené pomocné lokální proměnné, HODN:=(A)=5. Odkazem jsou předány adresy skutečných parametrů, tedy PROD1 = A, PROD2 = B. Obsah těchto proměnných se ovšem v úvodní části při předávání odkazem nijak neaktualizuje, proto i nadále zůstává (A) = 5,(B) = nedefinovaná hodnota.

Po předání parametrů se při nevyužité možnosti deklarace lokálních proměnných přechází k provádění těla procedury se skutečnými parametry. Po třech jednoduchých přiřazovacích příkazech.

PROD1:=PROD1+1	(A)=(A)+1=5+1=6
HODN:= HODN − 1	5 − 1 = 4
PROD2:= HODN	(B) = 4

je první zpracování procedury ukončeno. Jak vidíme, obsahy globálních proměnných jsou nyní rovny (A) = 6, (B) = 4.

Vnější program ovšem pokračuje bezprostředním novým voláním stejné procedury, tentokrát s jinými skutečnými parametry, viz příkaz CVICNA (A, A−3, A). Po vyvolání se znovu předá počáteční hodnota HODN:=(A) − 3 = 3 a odkazy PROD1 = A, PROD2 = A. V tomto případě budou výjimečně známé i obsahy všech užitých proměnných ihned na počátku zpracování procedury. Po zpracování procedury.

PROD1 := PROD1 + 1	(A) = (A) + 1 = 6 + 1 = 7
HODN := HODN − 1	3 − 1 = 2
PROD2 := HODN	(A) = 2

budou globální proměnné obsahovat (A) = 2, (B) = 4.

Zkuste si opět program sami modifikovat, jak rozšiřováním těla procedury, tak využitím lokálních proměnných. Sledujte, jak se uplatňuje zastiňování vnějších objektů lokálními deklaracemi a také jak na vaše chyby reaguje kompilátor chybovými hlášeními.

Funkce

Také funkce je uzavřeným podprogramem a představuje tedy určitou, speciální obdobu procedury. Se dvěma základními rozdíly:

– procedura se vyvolává příslušným příkazem. Funkce se naopak v programu, kde má být užita, pouze zapisuje symbolickým jménem, doplněným seznamem skutečných vstupních parametrů;

– funkce může mít i několik vstupních parametrů, výsledkem však vždy bude jediná vypočtená hodnota.

Funkce jsou tedy speciální podprogramy pro výpočet jediné hodnoty určitého typu. Vypočtená hodnota funkce je předávána na pozici jejího zápisu, tedy identifikátoru ve volajícím programu. Využití vypočtené funkce je vždy záležitostí vnějšího programu, v němž může být identifikátor se skutečnými parametry součástí výrazu nebo přiřazovacího či jiného příkazu.

Deklarace funkce je velmi podobná deklaraci procedury. Stejně jako u procedur se umisťuje za deklaracemi proměnných. Hlavička se skládá z klíčového slova function, identifikátoru, seznamu formálních vstupních parametrů a označení typu počítané, předávané hodnoty. Pole lokálních deklarací je nakonec následováno blokem příkazové posloupnosti.

Osvětlit deklaraci i zápis funkce ve vnějším programu opět napomáhá jednoduchý příklad, řešící výpočet faktoriálu zadaného čísla, tab. 3. Toto číslo se zadává prostřednictvím klávesnice na příslušný náznak. Tělo funkce řeší výpočet ukládáním postupných součinů čísel (1, 2, 3 ... X) s využitím

Tab. 3. Příklad deklarace a zápisu funkce v programu

```
PROGRAM FUNKCE;
VAR
          A: INTEGER;
          Z: REAL;
FUNCTION FACT (X: INTEGER): REAL;
VAR
          I: INTEGER;
          F: REAL;
BEGIN
  F:=1;
  FOR I:=1 TO X DO
    F:=I*F;
  FACT:=F
END;
BEGIN
  WRITELN ('ZADEJ CISLO:');
  READ (A);
  Z:=FACT (A);
  WRITELN ('FAKTORIAL ',A,'=',Z)
END.

RUN
ZADEJ CISLO:
13
FAKTORIAL 13 = 6.22702E+09
```

pomocné lokální proměnné F. Ve vnějším programu je identifikátor funkce se skutečným parametrem zapsán na předposledním řádku jako součást přiřazovacího příkazu.

Datové typy a struktury

Ač stručně, prošli jsme podstatnými prostředky jazyka Pascal z hlediska jeho výstavby a příkazové struktury. Program je ovšem vytvářen nejen s využitím příkazů, ale i dat. Na data mohou být při řešení různých úloh kladeny různé požadavky, například z hlediska typu, informačního rozsahu, přesnosti nebo struktury. Proto jsou u vyšších jazyků datové reprezentace širšího sortimentu základních a speciálních typů doplněny možností vytvářet jejich složitější, hromadné struktury (pole, řetězce, záznamy, soubory . . .). To v souhrnu znamená možnost volit datové typy podle potřeby programu tak, aby nad nimi prováděné operace byly efektivní a dostatečně přesné nebo výstižné.

Každá práce s nějakými datovými reprezentacemi musí probíhat podle určitých pravidel, jinak jsou dosažené výsledky nepřesné nebo i zcela nesmyslné. Obdobné informace o datech, s nimiž pracuje, musí mít při schopnosti pracovat se širší základnou datových typů i pascalský program. Jsou mu předávány v definiční a deklarační části zdrojového textu, tedy v té části programu, které jsme si dosud všímali pouze okrajově.

Pascalský program se vždy skládá ze dvou vzájemně neoddělitelných částí, deklarační a příkazové, obr. 2.

Deklarační část musí jednoznačně, podle stanovených pravidel, popisovat každou užitou konstantu, proměnnou či návěští. Pro potřeby překladače musí být u každého z těchto objektů implicitně nebo explicitně definován typ a deklarován jeho identifikátor.

V jazyku Pascal jsou implicitně definovány tyto standardní, jednoduché datové typy: Integer, real, Boolean a char. Pro dokonalý postih každého takového objektu proto postačí v deklarační části programu objekt pouze deklarovat, např.:

```
const
  A = 13;
var
  B, C : integer;
  X : real;
  Y, Z : char.
```

Deklarace tedy znamená pojmenování objektu, např. proměnné, symbolickým jménem, kterým pak v programu tuto proměnnou identifikujeme. Víme, o jaký typ proměnné se jedná a díky systematickému zápisu deklarační části to ví i překladač. Proto může podle typu datového objektu přidělovat potřebný paměťový prostor, kontrolovat případné překročení mezí přípustných hodnot aj.

Pascal umožňuje pracovat i s několika dalšími typy dat, které však specifikuje pouze formálně a programátor si je může upravovat pro potřeby aplikace. Získává tak účinný nástroj k výstavbě svým způsobem vlastních datových typů a struktur. Ty nyní ovšem musí být nejprve definovány, teprve pak mohou být užity při deklaraci objektu. Explicitní definice vycházejí z typů již definovaných, které samozřejmě představují známé standardní typy.

Podobně jako u příkazových typů omezuje Pascal také počet vyšších datových typů. Mohou být definovány typy:

výčtový,	pole,
interval,	záznam,
množina,	soubor.

Jednotlivé datové typy si nyní stručně popíšeme.

Výčtový typ

Pomocí definice výčtového typu je možno zavést nový typ datového objektu charakteristický tím, že má programátorem stanovený, omezený rozsah zobrazení hodnot. Výčtový typ je typ ordinární, každá jeho položka má svoje pořadové číslo (0, 1 až *N*), platí u něj i standardní funkce ord (pořadové číslo), pred (předchůdce), succ (následník) pro vyhledávání jednotlivých položek. Rovněž může být při jejich zpracování využito relačních operátorů <, >, =, <>, <=, >=. Jako příklad definice výpočtového typu si můžeme uvést seznam schematických značek

```
TYPE ZNACKY = (ODPOR, CIVKA, DIODA, TRANZISTOR, . . . );
```

a následnou deklaraci proměnné tohoto typu VAR S: ZNACKY.

Typ interval

Uvnitř definovaného ordinálního typu může být definován nový typ interval s omezeným rozsahem hodnot vůči původnímu, tzv. hostitelskému typu. Tak například v oboru typu integer můžeme definovat interval

```
TYPE PORADI = 5 . . 13;
```

uvnitř typu char

```
TYPE ZNAMKA = '1' . . '5';
```

uvnitř výčtového typu ZNACKY interval

```
PASIVNI = ODPOR . . DIODA.
```

Jediný ze standarních typů, který nemůže tvořit položku intervalu, je číselný typ real, který není ordinal.

Při práci s proměnnými typu interval zůstávají i nadále platné veškeré operace, syntakticky odpovídající hostitelskému typu. Zavedené omezení rozsahu přípustných hodnot zpřehledňuje a usnadňuje zápis i vyhodnocování proměnných a výsledků.

Množiny

Pascal rovněž umožňuje práci s daty typu množina (set), jaká známe z množinové algebry. Nad těmito množinami je možno provádět obvyklé operace sjednocení (U =+), průniku (∩ =*) a rozdílu (−). Prvky množin mohou být opět pouze položky jednoho typu. Na rozdíl od předchozích typů však položky množiny nejsou nijak uspořádány.

Před deklarací musí být typ položek množiny samozřejmě opět definován, buď implicitně (u jednoduchých typů), nebo explicitně programátorem. Užijeme-li tedy např. jako základní již definovaný výčtový typ ZNACKY, pak můžeme definovat množinu typu,

SCHEMA = SET OF ZNACKY;

a pak deklarovat proměnné tohoto typu

VAR
PASIV, AKTIV, POLOV : SCHEMA;

Přiřazení hodnot jednotlivých proměnných typu množina se provádí pomocí tzv. *konstruktoru*, tj. seznamu prvků příslušného typu množiny, uvedeného v hranatých závorkách, tedy například

PASIV:= [ODPOR, CIVKA, DIODA,...].

Vedle již uvedených základních množinových operací umožňuje Pascal test množin na rovnost a nerovnost (=, < >) a obsažení jedné množiny v druhé (<=, >=). Pro test příslušnosti libovolného prvku k určité množině se užívá operátoru IN, např. zápis DIODA IN PASIV bude vyhodnocen jako true, protože DIODA je položkou množiny PASIV. Výsledků operací, prováděných nad množinami, tak lze využít jejich vyhodnocením podmíněnými příkazy k řízení programu.

Zatímco výčtové, intervalové i množinové typy představují do určité míry omezení a zjednodušení rozsahu bázových typů, jsou další datové typy, jejichž vytváření Pascal podporuje, naopak vůči výchozím typům podstatně rozsáhlejší a komplexnější. Vznikají složitější datové struktury, datové bloky. Přitom však i nadále zůstávají poměrně přehledné, protože jejich konstrukce je omezena pascalskou syntaxí na několik přípustných abstrakcí. Základní, nejdůležitější i nejužívanější takovou strukturou je pole.

Pole

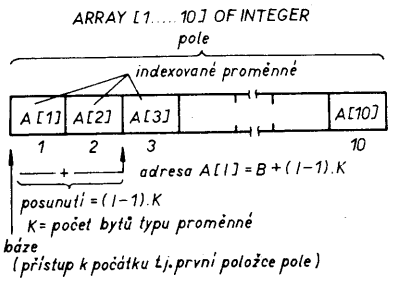
Pole (array) je základním představitelem složeného datového typu. Jeho položky mohou být tvořeny jak jednoduchými, tak složenými typy. Všechny položky pole však musí být vždy shodného typu. Tím, že příkazová struktura jazyka dovoluje provádět efektivní operace s hutnějšími datovými strukturami, roste jak výkonnost, tak přehlednost práce s počítačem.

Přesněji si můžeme pole definovat jako souvislý blok uspořádaných datových položek stejného typu, přičemž počet položek je přesně znám. Každá položka má proto v poli své přesně určené místo (obr. 11). Pole je pojmenováno symbolickým jménem, např. POL. Každé položce pak přísluší shodné jméno, doplněné jejím indexem v hranatých závorkách, např. POL [1] až POL [*N*]. Pomocí indexu, kterým je každá proměnná v rámci pole jednoznačně identifikovatelná, se k ní zajišťuje přístup (zápis, čtení).

Deklarace pole má tvar

VAR JMENO : ARRAY [m1 .. m2] OF TYP;

zde jsou array a of klíčová slova, výraz [m1 . . m2] určující typ a meze indexů je intervalového typu a TYP je pozice určená pro označení typu pole.



Obr. 11. Znázornění struktury pole v paměti mikropočítače a strategie přístupu k jeho jednotlivým položkám

Každá položka pole se, podle svého označení, nazývá indexovaná proměnná a má stejné vlastnosti jako běžná proměnná stejného typu. Organizovaná struktura indexovaných proměnných uvnitř pole umožňuje efektivní zpracování číselných a textových algoritmů, viz například vyhledávání extrémních prvků pole, jejich řazení nebo třídění podle určitých kritérií.

Až dosud jsme pole uvažovali pouze jako jednorozměrnou uspořádanou datovou strukturu, v níž je každá položka označena indexem. Pro úplnost je vhodné upozornit, že samozřejmě existují i několikarozměrná pole. Z nich nejznámější je pole dvourozměrné, tzv. matice. Jeho protějšek opět nacházíme v klasické matematice. Takovou matici můžeme deklarovat například následujícím způsobem

```
TYPE MAT = ARRAY [1 . . 10,1 . . 5] OF
           INTEGER;
VAR X : MAT;
```

Uvedená matice s dvojím indexem X [I, J] se skládá z deseti řádků a pěti sloupců. Protože je typu integer, bude při lokaci každé položky ve dvou bytech matice vyžadovat kapacitu paměťového procesoru 10×5×2 = 100 byte.

Orientační příklad práce s jednoduchým číselným polem je v tab. 4. Program SAPP ukazuje deklaraci číselného pole real, způsob zápisu hodnot jednotlivých položek (indexovaných proměnných) a jejich zpracování, tj. součet a výpočet průměrné hodnoty s využitím dvou cyklů for. První cyklus slouží pro načtení stanoveného počtu položek pole (N = 6) z klávesnice, druhý cyklus pro jejich nastřádání do pomocné proměnné ACC. Jak zadané položky, tak výsledný součet a průměr jsou v prováděcím režimu vypsány na obrazovce.

S typem pole se budeme velmi často setkávat i při práci v asembleru. Při konstrukci pole je pak vždy třeba:

– rezervovat odpovídající místo pro pole v datové části paměti, přičemž potřebný prostor je určen počtem prvků pole a jejich typem,

– zajistit odkaz na počátek pole (tj. bázovou adresu pole) a jeho jednotlivé položky, tj. indexované proměnné. Adresu každé položky je možno určit součtem bázové adresy a tzv. posunutí, určeného součinem indexu této proměnné a počtu bytů, potřebných pro reprezentaci typu jedné položky tohoto pole (obr. 11).

Indexovanou proměnnou tedy můžeme považovat za obdobu běžné statické proměnné stejného typu s tím, že je nedílnou součástí vyšší datové struktury, pole, v němž má svou přesně určenou pozici. Přívlastek statický znamená, že proměnná, popsaná v deklaračním úseku příslušného programového bloku (globálního, lokálního), existuje po celou dobu jeho aktivity.

Tab. 4. Příklad práce s jednoduchým číselným polem

```
PROGRAM SAPP;
CONST N=6;
VAR
        I:INTEGER;
        ACC,PRUM:REAL;
        P:ARRAY [1..N] OF REAL;
BEGIN
  WRITELN ('ZADEJ HODNOTY POLE:');
  FOR I:=1 TO N DO READ (P[I]);
  ACC:=0;
  FOR I:=1 TO N DO
    ACC:=ACC + P[I];
  PRUM:=ACC/N;
  WRITELN;
  WRITELN ('SOUCET HODNOT =',ACC);
  WRITELN ('PRUMERNA HODNOTA =',PRUM)
END.

RUN
ZADEJ HODNOTY POLE:
0.3
3.4
0.1
5
21.8
-6

SOUCET HODNOT = 2.460000E+01
PRUMERNA HODNOTA = 4.10000E+00
```

Textové řetězce

Dosud uvažovaný typ pole může jako svou položku použít libovolný typ, tedy i znakový typ char. Tento případ však v podstatě znamená plýtvání pamětí, protože v Pascalu je minimální adresovatelná kapacita paměťové buňky word = 2 byte. Za tímto účelem byl zaveden tzv. zhuštěný typ pole, packed array, využívající pro reprezentaci každé položky char právě jeden byte. Takto lze definovat jak znakové řetězce, tak konstanty.

Případ definice typu řetězce:

```
RETEZ = PACKED ARRAY [1 . . 10] OF
        CHAR;
```

a navazující deklarace proměnné

POPIS : RETEZ;.

Také při práci s řetězci je možno používat relační operátory.

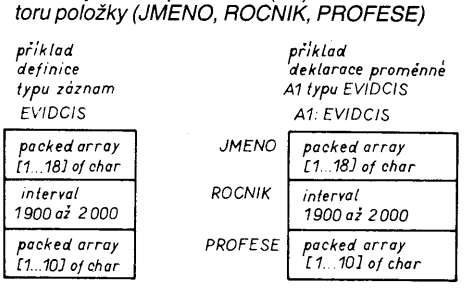
Záznam

Někdy je poměrně značným nedostatkem datového typu pole z hlediska obsahové účinnosti to, že jeho položky mohou být tvořeny pouze objekty shodného typu. V praxi je poměrně často, zvláště při hromadném zpracování dat, třeba sdružovat několik datových položek různého typu do jedné složky – například v nějaké agendě může takovou složku tvořit jméno pracovníka (char), ročník narození (integer), profese (char) atd. Pro podobné účely je v jazyku Pascal zaveden datový typ záznam.

Záznam představuje datovou strukturu, sestávající z definovaného počtu pojmenovaných položek, přičemž každá položka může být libovolného typu. Položka záznamu tedy může být tvořena i polem nebo jiným záznamem. Každá položka i celý záznam musí být definovány v deklaračním úseku programu. Definice, deklarace i užití proměnné typu záznam si všimneme v souvislosti s obr. 12.

Ve schématu definice záznamu se užívá klíčových slov type a record. Ještě jednou zdůrazněme, že jak celý záznam, tak všech-

Obr. 12. Deklarací konkrétní proměnné (A1) typu záznam musí předcházet jeho definice (EVIDCIS). K jednotlivým položkám této proměnné se přistupuje pomocí selektoru, tvořeného jménem proměnné (A1) a identifikátoru položky (JMENO, ROCNIK, PROFESE)



ny jeho položky musí být označeny jménem s uvedením typu. V našem příkladu zavedeme položky JMENO, ROCNIK, PROFESE, celý záznam pojmenujeme EVIDCIS. Definice takového záznamu pak může vypadat následovně

```
TYPE
   EVIDCIS = RECORD
   JMENO : PACKED ARRAY  |1..18|
                OF CHAR;
   ROČNÍK : 1920..1975;
   PROFESE : PACKED ARRAY
                |1..10|  OF CHAR
   END
```

Tímto způsobem je zaveden typ záznamu EVIDCIS, který lze použít pro deklarace příslušných proměnných tohoto typu, například takto

```
VAR A1 : EVIDCIS;
```

Deklarované jméno tvoří základní identifikátor proměnné A1 typu EVIDCIS. Pro specifikaci přístupu k jednotlivým položkám se doplňuje selektorem, tedy jménem té které položky, zavedeným při definici příslušného typu záznamu. V našem příkladu tedy třebas A1.JMENO = přístup k 1. položce typu char (JMENO),
A1.ROCNIK = přístup k 2. položce typu interval (ROCNIK).

Je-li dále příslušná položka záznamu strukturovaného typu, doplňuje se její selektor dalším selektorem tak, aby cesta od identifikátoru k položce byla jednoznačně určena.

Uvedeným způsobem lze přistupovat k celé struktuře, ke všem položkám proměnných typu záznam. To znamená definovat i číst jejich hodnoty jako u běžných proměnných, byť jsou součástí složitějšího, nehomogenního celku. Uvažujeme-li předběžně o způsobech možného vytváření záznamu při programování v asembleru, bude jistě logické předpokládat přístup k jednotlivým položkám za pomoci ukazatelů. Přístup ovšem bude ve srovnání s polem poněkud obtížnější, protože bude muset být respektována struktura celého záznamu, vliv počtu a typů všech položek na nastavení ukazatele. Zde již poměrně výrazně vystupuje do popředí nutnost systematické výstavby složitějších datových struktur – v datové části operační paměti musí být vyhrazen přesně definovaný prostor pro všechny datové položky záznamu. Jejich uložení pak musí korespondovat s optimální koncepcí programově řízeného přístupu k těmto položkám.

Soubor

Zcela mimořádný význam má v současné etapě vývoje výpočetní techniky datová struktura soubor (file). Podobně jako pole, představuje soubor uspořádanou množinu položek stejného typu, ať již jednoduchého, nebo strukturovaného. Na rozdíl od pole však počet položek není stanoven. Není, přesněji nemusí být ani označena pozice položky uvnitř souboru, jako tomu bylo například indexováním proměnné v poli. Běžný, sekvenční přístup k položce souboru probíhá tak, že se postupně zpřístupňují všechny jeho položky.

Soubory lze dělit několika způsoby, první je dělení na soubory pracovní a vnější. Pracovní soubor je datová struktura, okamžitě existující v operační paměti počítače. S ukončením činnosti určitého programu zaniká i existence jeho pracovních souborů.

Za skutečné soubory se považují soubory vnější, tj. takové, které se nacházejí mimo interní operační paměť, nejčastěji uloženy na médiích vnějších pamětí diskového typu (floppy, hard), ale i na mgf kazetách nebo páscích. Podstatnou charakteristikou vnějších souborů je to, že jejich existence je trvalá. Nezávislá na okamžitých vnějších ani vnitřních činitelích, výpadcích napájecích napětí, poruchách systému aj. Vnější paměť s velkou kapacitou, řádově převyšující kapacitu operační paměti, představuje za současného stavu techniky ideální prostředek archivace programů a dat, které mohou být do počítače zaváděny v případě jejich aktuální potřeby. Všechny programy a datové bloky se do vnějších pamětí ukládají ve formě souborů.

Způsoby, jakými vnější paměť s vlastním počítačem spolupracuje, jsou ovšem různé, závisí na technických prostředcích, jakými počítač i paměťový systém disponují a také na užitém operačním systému. Těmito záležitostmi se budeme zabývat později.

Další rozdělení lze odvodit přímo z určení souborů. Pro komunikaci typu operátor-počítač se z pochopitelných důvodů užívají soubory textové. Takový soubor proto musí být, před vlastním zpracováním v počítači, nejprve konvertován do příslušného kódu. To znamená vždy především prodloužení celkové doby zpracování souboru, mnohdy je i vyjádření informačního obsahu omezeným souborem alfanumerických znaků nevhodné nebo i nemožné. Nicméně, textové soubory představují i nadále jediný univerzální způsob komunikace operátora s počítačem.

Na druhé straně je mnoho situací, kdy textové soubory do značné míry nebo úplně ztrácejí opodstatnění a smysl. Je to vždy, když je třeba do vnější paměti až již pro systémové nebo uživatelské účely uložit takové datové či programové bloky, které budou později opět využity přímo počítačem, bez interakce operátora. Odpovídající datové a programové soubory se pak vytvářejí v původním kódu příslušných položek a v této formě se také opět zavádějí zpět do počítače.

Shrnuto, každý (datový, programový i textový) soubor chápeme jako lineárně uspořádanou množinu položek stejného typu, jejíž organizace umožňuje sekvenční zpracování souboru, tj. jeho postupné vytváření (zápis) nebo čtení. Pro postih základní charakteristiky práce se soubory je dobré uvědomovat si stále dvě skutečnosti.

První – při vytvoření souboru se současně, automaticky vytváří i tzv. přístupová proměnná tohoto souboru, jejíž typ je určen typem položek souboru. Prostřednictvím přístupové proměnné, kterou si můžeme představit jako buffer v operační paměti, jsou předávány jednotlivé položky souboru mezi počítačem a vnější pamětí. Ze shody formátu přístupové proměnné a položky souboru vyplývá, že v každém okamžiku může být přístup pouze k jedné položce souboru, obr. 13.

Druhou je skutečnost, že soubor s neurčeným počtem položek musí být při zpracování nějakým způsobem ohraničen, ukončen. K tomu se používá tzv. příznak konce souboru (end of file). Prostřednictvím standardní vestavěné pascalské funkce EOF (jméno souboru) je při čtení prováděn test na dosažení konce pojmenovaného souboru.

Různých souborů může být ve vnější paměti uložen prakticky „neomezený" počet. Systém jejich zpracování musí být proto zabezpečen tak, aby v okamžiku přístupu (čtení, zápis) k jednomu ze souborů byly ostatní soubory pro stejnou činnost nepřístupné. K řízení práce se soubory se užívá příslušných procedur, vyvolávaných standardními příkazy. Jak ukazují i následující schématické příklady vytváření a čtení souboru, soubor nemůže být současně otevřen pro čtení i zápis.

Vytvoření souboru F, tedy jeho zápis, se v podstatě skládá ze tří fází:

otevření souboru pro zápis, REWRITE (F),
výstup položek souboru, WRITE (F, parametry),
uzavření souboru, CLOSE (F).

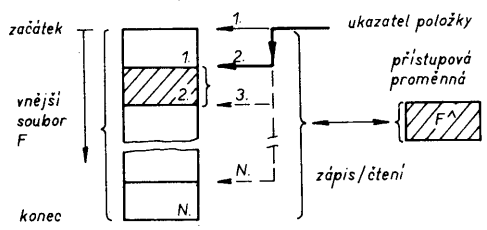
V příkladu jsou užity standardní příkazy zápisu jednotlivých procedur se jménem zpracovávaného souboru a předávanými parametry. Procedury rewrite a close představují určité vnější svorky, otevírající a ukončující přístup k souboru. Procedura rewrite implicitně nuluje případný dřívější obsah souboru, do něhož proto zápis začíná od první položky. Procedura WRITE (F, parametry) předává v cyklu hodnoty, odpovídající seznamu parametrů, prostřednictvím přístupové proměnné F ^. Závěrečná procedura close zpracovaný soubor uzavírá, obr. 14.

Mechanismus čtení vnějšího souboru, z hlediska počítače tedy vstupního, je obdobný. Je opět řízen třemi standardními procedurami:

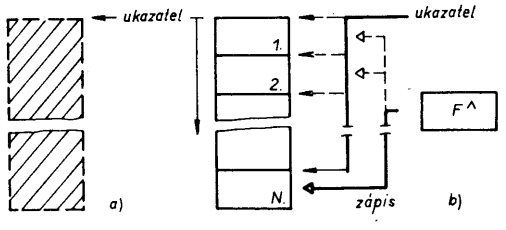
otevření vstupního souboru, RESET (F),
čtení jeho položek, READ (F, vstupní proměnná),
uzavření souboru, CLOSE (F).

Příkazem RESET (F) se do přístupové proměnné F ^ přenese hodnota první položky souboru a funkce EOF (F), hlídající konec souboru, se nastaví na hodnotu false. Procedura READ (F, vstupní proměnná) postupně čte jednotlivé položky do uvedené proměnné. S přečtením poslední položky je soubor vyčerpán, následuje čtení prázdné hodnoty do přístupové proměnné. To je vyhodnoceno funkcí EOF (F) jako true, tedy jako konec souboru. Příkazem close je příslušný soubor znovu uzavřen.

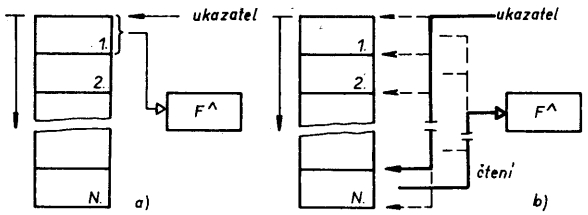
Soubor je tedy zpracováván s využitím přístupové proměnné F ^ a příznaku konce souboru EOF (F), obr. 15. Od tohoto sché-



Obr. 13. Znázornění přístupu k položkám souboru



Obr. 14. Vytváření souboru; a) příkazem REWRITE (F) se ruší případný předchozí obsah otevíraného souboru, b) procedura WRITE (F, parametry) realizuje zápis jednotlivých položek do otevřeného souboru



Obr. 15. Čtení souboru; a) příkazem RESET (F) se otevře soubor pro čtení. Do přístupové proměnné F^ se načte 1. položka a příznak EOF (F): =FALSE, b) procedura READ (F, vstup) čte postupně položky otevřeného souboru. S koncem souboru je vyhodnocen příznak EOF(F):= TRUE

matu se liší zejména textové soubory, organizované do textových řádků, přičemž každý řádek může obsahovat různý počet znaků. Proto je každý řádek ukončován speciálním služebním znakem, umožňujícím využít pro detekci konce čteného řádku standardní funkce EOLN (F).

Vytváření, čtení a modifikace souborů patří při práci s diskově orientovanými systémy k základním uživatelským a programátorským pracím. Práce se soubory je pochopitelně mnohem složitější a rozsáhlejší, než by se z nástinu základních principů mohlo zdát. Pro nás však zatím uvedené informace stačí. Uvědomme si ještě, že změnit obsah sekvenčně zpřístupňovaného souboru lze pouze jeho postupným čtením, zpracováním a znovuvytvořením. Příkladem může být postup cyklického vytváření kopie textového souboru, tab. 5, osvětlující názorně otevírání, zpracování i uzavření souboru. Dosud uvažované metody tedy umožňují modifikovat obsah souboru pouze jeho novým, úplným znovuvytvořením. Ve skutečnosti zpravidla existují i další možnosti. Jednou z nich je doplňování souborů připisem na jejich konec, druhou technika přímého přístupu k jednotlivým položkám.

Dynamické proměnné

Nakonec si ještě všimněme principu dynamických proměnných, tvořících spolu s ukazateli základní prvek pascalských dynamických datových struktur.

Všechny dosud uvažované datové typy a struktury mají vedle zřejmých předností i jedno společné omezení, vyplývající z principu hierarchické struktury programu.

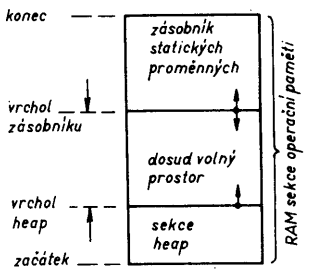
Tab. 5. Cyklické čtení a vytváření textového souboru

```
PROGRAM ZALOHA (INPUT,OUTPUT);
VAR
  FVST,FVYST : TEXT;
  Z :CHAR;

BEGIN
  RESET (FVST);                    otevření vstupního souboru
  REWRITE (FVYST);                 otevření výstup. souboru
  WHILE NOT EOF(FVST) DO           detekce konce FVST
    BEGIN
      WHILE NOT EOLN(FVST) DO      detekce konce řádku
        BEGIN
          READ (FVST,Z);           čtení dalšího znaku řádku
          WRITE (FVYST,Z);         zápis dalšího znaku řádku
        END;
      READLN (FVST);               odřádkování FVST
      WRITELN (FVYST)              odřádkování FVYST
    END
END.
```

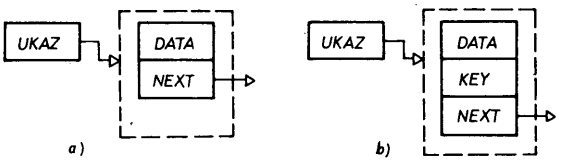
Pro zajištění izolace lokálních proměnných právě aktivního programového bloku se pro ně využívá určité oblasti volné datové paměti RAM, organizované z hlediska přístupu jako zásobník – s každým vyvoláním vnořené části programu (procedury) se, počínaje od vrcholu operační paměti, obr. 16, do zásobníku ukládají příslušné deklarované lokální proměnné, které pak s ukončením procedury automaticky zanikají. Rozsah zásobníku se tak v průběhu provádění programu mění.

Určitým nedostatkem systému statických proměnných je v některých případech pevně stanovený formát užitého datového typu (definice) i jejich rozsah (deklarace), tedy právě to, co umožňuje zavést systém kontrol správnosti vzájemného přiřazení datových a příkazových typů nebo operátorů. Například do struktury statického pole nelze jednoduše na libovolnou pozici doplnit nebo naopak z ní odebrat další položku – je třeba vytvořit pole nové a nově přiřadit hodnoty a indexy. Což je samozřejmě z hlediska časové režie, zvláště u rozsáhlého pole, značně náročné. Podobně je při práci se složitější datovou strukturou (záznam) třeba stále pracovat s plným formátem deklarované proměnné, i když je často třeba přistupovat pouze k jediné položce. To pak znamená plýtvání aktuálně využitým paměťovým prostorem, včetně časových ztrát.

Z uvedených důvodů je v Pascalu zaveden systém dalších, tzv. dynamických proměnných. Pro ně je v datové části operační



Obr. 16. Organizace prostorů statických a dynamických proměnných v operační paměti. Zásobník statických proměnných se zvětšuje i zmenšuje automaticky se zpracovávaním lokálních procedur, prostor heap se zvětšuje s každým vytvořením nové dynamické proměnné procedurou NEW. Při zrušení proměnné procedurou DISPOSE se však nemění pozice vrcholu heap, ale uvnitř sekce heap se vytvoří volný prostor, sekce se fragmentuje



Obr. 17. K definici a formátu prvku dynamické datové struktury; a) k příkladu definice typu prvku jednoduchého seznamu a jeho statického ukazatele, b) prvek seznamu je typu RECORD. Může proto obsahovat několik informačních položek. Jednou z nich bývá klíč (KEY), podle něhož může být organizováno řazení prvků seznamu

paměti rezervován zvláštní prostor, označovaný jako heap (hromada). Zpravidla je orientován na opačném konci adresového prostoru vůči prostoru proměnných statických, obr. 16.

Umístění i formát dynamické proměnné ve vytvářené nebo zpracovávané vyšší datové struktuře je variabilní. Dynamickou proměnnou je možno vytvářet, číst, měnit její obsah i rušit v průběhu provádění programu. Ze všech těchto aspektů vyplývá, že dynamická proměnná nemůže být předem deklarována. Proto také nemůže mít svůj indentifikátor, pomocí kterého bychom se na ni v programu odkazovali. Odkaz může být pouze nepřímý, prostřednictvím tzv. ukazatele, pointeru.

Pro dynamické vytváření i rušení proměnných dynamického typu v průběhu programu se používají standardní procedury NEW (ukazatel) a DISPOSE (ukazatel). V deklarační části programu se vždy definují typy proměnných a ukazatelů, deklarují se pouze proměnné typu statických ukazatelů.

Příklad definice prvku nejjednodušší dynamické datové struktury, seznamu (jeho struktura je na obr. 17a):

```
TYPE
  UKAZ = ^ELEM;
  ELEM = RECORD
           DATA : typ dyn. proměnné;
           NEXT : UKAZ
           END;
```

Zde je typ UKAZ statickým ukazatelem na daný typ prvku seznamu ELEM, přičemž definovaný typ prvku ELEM se skládá ze dvou dále nedělitelných částí:

a) z vlastní dynamické proměnné DATA jako nositele informačního obsahu,

b) z dynamického ukazatele (odkazníku) na případný další prvek seznamu.

Jak patrno z definice, statický ukazatel UKAZ umožňuje přístup k oběma složkám dynamického prvku ELEM. K jejich rozlišení se používá tzv. *dereferenční operátor* ^. Při jeho užití se uskutečňuje přístup k dynamické proměnné DATA, bez něj k dynamickému ukazateli NEXT.

V deklaračním úseku programu se deklarují statické proměnné typu ukazatel, např.

```
VAR
  ZAC, POM, HEAP : UKAZ ;
```

Definovaný prvek seznamu typu ELEM se tedy skládá ze vzájemně vázané dvojice dynamická proměnná + „směrník" na další proměnnou. V programu se vytváří vyvoláním standardní procedury NEW s příslušným ukazatelem (proměnnou) jako parametrem, např. NEW (POM). Při vyvolání se uskuteční tyto akce:

V prostoru heap operační paměti se vytvoří a umístí blok, odpovídající definované struktuře prvku ELEM. Jeho adresa se uloží

do statického ukazatele, který byl parametrem procedury, tj. POM. Obě položky DATA a NEXT prvku mají po jeho vytvoření nedefinované obsahy. Přístup k jejich počáteční inicializaci i pozdějším změnám zajišťuje ukazatel.

S využitím dereferenčního operátoru ^ se zajišťuje přístup k vlastní dynamické proměnné prvku. Příklad aktualizace jejího obsahu

POM^, DATA := {hodnota odpovídajícího datového typu}.

Přístup k položce NEXT, jejímž smyslem je odkaz na další prvek seznamu, se syntakticky zapisuje bez zmíněného operátoru, např. POM.NEXT. Všimněme si však toho, že pokud vytváříme první z prvků nového seznamu, nemůže být jeho směrník inicializován, protože dosud další prvek seznamu, na který má ukazovat, neexistuje. A navíc, při vytváření jednoduchého, lineárně uspořádaného seznamu ani existovat nebude, protože jeho struktura odpovídá zásobníku LIFO. Proto se směrník prvního vytvořeného prvku seznamu definuje pomocí klíčového slova NIL jako prázdný, tj. neukazuje na žádný další prvek.

Mechanismus vytváření a následného čtení lineárně uspořádaného znakového seznamu názorně osvětluje program SEZNAM, tab. 6. Syntaxe definice dynamických proměnných a ukazatelů se od standardní u Hisoft Pascalu poněkud liší, nicméně bude jistě na první pohled srozumitelná. Procedura MARK vymezuje paměťový prostor dynamických proměnných, procedurou RELEASE se tento prostor a tím i vytvořené dynamické proměnné a směrníky ruší.

Tab. 6. Příklad zápisu a následného čtení dynamicky vázaného seznamu

```
PROGRAM SEZNAM;
TYPE
  elem=RECORD
         next:^elem;
         data:CHAR
         END;
  ukaz=^elem;
VAR
  A:=CHAR;
  zac,pom,heap:ukaz;
BEGIN
  WRITELN ('PORADI ZAPISU:')
  MARK (heap);
  zac:=NIL;
  REPEAT
    READ (A);
    NEW (pom);
    pom^.data:=A;
    pom^.next:=zac;
    zac:=pom;
  UNTIL EOLN;
  pom:=zac;
  WRITELN ('PORADI CTENI:');
  WHILE pom<>NIL DO
    BEGIN
      WRITE (pom^.data);
      pom:=pom^.next
    END;
  RELEASE (heap)
END.

RUN
PORADI ZAPISU:
qwerty 7BC a
PORADI CTENI:
a CB7 ytrewq
```

Vlastní program se skládá ze dvou samostatných cyklů. V prvním cyklu REPEAT, do kterého se vstupuje s „prázdným" ukazatelem ZAC, je postupně čten z klávesnice libovolný počet znaků. Každý znak je nejprve uložen do pomocné proměnné A (char) a následuje vytvoření jemu příslušného prvku seznamu. Jako parametru procedury se využívá pomocné proměnné POM, jejíž obsah po vytvoření příslušného prvku seznamu představuje dočasně jeho statický ukazatel. Pak vždy následuje aktualizace obsahu dynamické proměnné přesunem obsahu znakové proměnné A. Do směrníku stejného prvku se přesouvá obsah statického ukazatele ZAC. U prvního prvku vytvářeného seznamu proto bude POM^. NEXT = NIL, samotný prvek tedy na žádný další prvek seznamu neukazuje. Vzhledem k následujícímu přiřazení ZAC:=POM naopak bude na tento prvek odkazovat směrník dalšího vytvořeného prvku atd. Zápisový cyklus je ukončen vyhodnocením stisku klávesy ENTER. Na poslední z vytvořených prvků ukazuje statický ukazatel proměnné ZAC = POM, jako vzájemné ukazatele jednotlivých ostatních prvků slouží vždy směrníky sousedních, následně vytvořených prvků. Směrník prvku, který byl vytvořen jako první, je prázdný. Situaci postihuje obr. 18.

Průběh čtecího cyklu je jednoduchý. Jako ukazatel na první čtený prvek (který však byl vytvořen jako poslední) je k dispozici obsah statické proměnné ZAC=POM. Hodnota (znak) dynamické proměnné tohoto prvku je vypsána na displeji a následuje přepis obsahu směrníku právě čteného prvku jako ukazatele do POM. Tak je zajištěn odkaz na další čtenou dynamickou proměnnou. Zpracování seznamu probíhá v cyklu tak dlouho, až je přečten jeho poslední (první vytvořený) prvek s prázdným směrníkem.

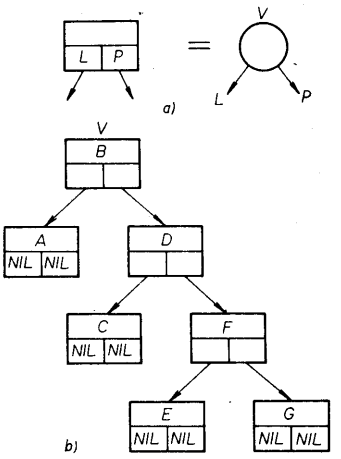
Uvedeným způsobem lze vytvářet seznamy různé, předem nedeklarované délky. Prvky seznamu, představované strukturou typu záznam, ovšem mohou mít vůči příkladu značně rozsáhlejší formát. Často se užívá takových formátů, v nichž některá položka představuje tzv. klíč seznamu, obr. 17b, podle kterého mohou být jeho prvky uspořádány. Seznam se také může skládat z prvků různého typu. A konečně, do seznamu mohou být na libovolná místa přidávány (i ubírány) další prvky. Různé varianty řešení těchto úloh jsou popsány. v [2, 3].

Seznam je tedy variabilní struktura jak z hlediska formátu jeho datových položek (obecně typu záznam), tak systému jejich vzájemných vazeb, ovládaných prostřednictvím ukazatelů. Přitom lze vidět dva zvláštní případy struktury seznamu, nevyužívající přístupu k jeho vnitřním prvkům:

a) seznam, který se rozšiřuje (zápisem) a zkracuje (čtením) pouze na jeho konci, je vlastně zásobník LIFO – čten může být vždy pouze jeho poslední zapsaný prvek;

b) seznam, který se doplňuje pouze na konci a je čten pouze na začátku, představuje frontu FIFO. Datové položky jsou proto zpracovávány přesně v tom pořadí, v jakém byly do fronty zařazovány.

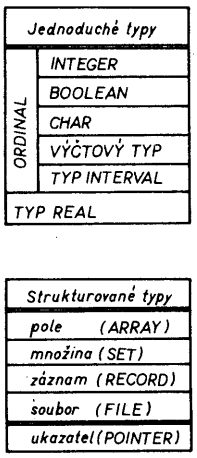
Příkladem druhého možnému směru využití dynamických proměnných jsou tzv. *stromové datové struktury.* Nejčastější uplatnění mezi nimi nacházejí binární stromy. Prvek binárního stromu obsahuje kromě dynamické proměnné ještě dva ukazatele, levý L a pravý P. Vzájemnou vazbou jednotlivých prvků prostřednictvím binárních ukazatelů vznikají dílčí stromy, tvořené vždy vrcholem V, levou a pravou větví, obr. 19a. Rozšířením této základní struktury vzniká strom s vrcholem V, jeho levý a pravý podstrom jsou dále tvořeny dílčími stromy, obr. 19b.



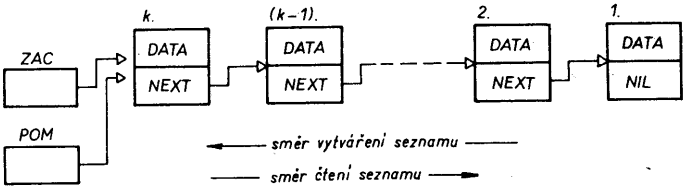
Obr. 19. Ke struktuře binárního stromu; a) prvek binárního stromu obsahuje dva ukazatele, b) hierarchická stromová struktura

Tak, na rozdíl od seznamů, vznikají složitější, hierarchicky uspořádané dynamické datové struktury. Různým pořadím vytváření, vyhledávání a zpracování datových položek stromu (preorder, inorder, postorder) lze dosahovat efektivního řazení dat uvnitř struktury při omezení časové režie.

Přehled pascalských datových typů je na obr. 20.



Obr. 20. Přehled pascalských datových typů



Obr. 18. Situace po vytvoření k prvkového seznamu. Oba ukazatele ZAC i POM odkazují na poslední vytvořený prvek seznamu jako na jeho začátek

I když jistě ne přesně (a už vůbec ne vyčerpávajícím způsobem) jsme shrnuli na několika stránkách hlavní principy, příkazové a datové struktury Pascalu jako základního představitele vyššího programovacího jazyka.

Následující kapitola, která je věnována několika ukázkám třídění datových struktur, sleduje v podstatě dva cíle: ukázat význam znalosti základních algoritmů a rutin pro tvorbu i jednoduchých programů a usnadnit prostřednictvím podrobně komentovaných a krátkých programů první kroky při práci s Pascalem.

Algoritmy

Již několikrát jsme se zmínili o významu efektivní algoritmizace řešených úloh. Algoritmem rozumíme obecný postup, vedoucí k hromadnému řešení úloh určitého typu, měl by tedy být v podstatě co nejuniverzálnější. To však vždycky možné není. V praxi proti sobě často stojí rozporné požadavky, mnohdy důležitější, než právě univerzálnost algoritmu. Mohou jimi být například rychlost výpočtu, nároky na operační paměť, rozsáhlost zpracovaných dat a jejich typy, uložení těchto dat a jiné.

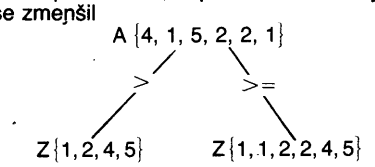
V zásadě můžeme rozlišit dva druhy algoritmů, numerické a ostatní. Výsledkem použití numerického algoritmu je výpočet určitých číselných hodnot. Příkladem může být algoritmus pro výpočet faktoriálu čísla v programu FUNKCE (tab. 3). K ostatním algoritmům patří takové, které se používají k vyhledání zadaných nebo extrémních prvků datových struktur, jejich řazení nebo třídění podle určitých kritérií, vytváření nových struktur, jejich porovnávání a k celé řadě dalších úloh. K řešení určité úlohy je zpravidla nutno využít celé řady různých algoritmů, které musíme znát nebo sami vytvářet. Sestavení určitého programu lze z tohoto hlediska přirovnat k návrhu schématu nového obvodu – i zde využíváme celé řady známých, klasických obvodových řešení (vlastně obvodových algoritmů) a jen zřídka kdy jsme nuceni nebo schopni vymyslet nový obvodový detail.

V této kapitole, která má být současně ukázkou převodu algoritmu do struktury zápisu pascalského programu, si ukážeme několik algoritmů třídění prvků datových struktur, které v praxi patří k nejdůležitějším. Pro zjednodušení se zaměříme na jednoduchá pole typu integer, uložená v operační paměti. Pro tuto třídu úloh existuje celá řada algoritmů, od velmi jednoduchých a snadno pochopitelných až po značně rafinované. Ukážeme si pouze několik nejznámějších.

Nejpřirozenější, ale současně i málo efektivní algoritmus třídění lze přímo odvodit analýzou obvyklého postupu člověka, řešícího stejnou úlohu s pomocí tužky a papíru. Chtějme např. z původní množiny celých čísel A {4, 1, 5, 2, 6, 8} vytvořit novou, vzestupně uspořádanou množinu Z. Zřejmě bychom postupovali tak, že bychom opakovaně procházeli celou množinou A, při každém průchodu vyhledali minimální prvek A_{min}, zapsali jej zleva na první volné místo v nově vytvářené množině Z a současně pro lepší orientaci tento prvek z původní množiny A vyškrtli či jinak označili jako již použitý. Po $n = 6$ průchodech (počet prvků množiny A) bychom tak vytvořili novou, vzestupně uspořádanou množinu Z {1, 2, 4, 5, 6, 8}. Všimněme si, že při tomto postupu se stále, až do vyčerpání počtu prvků množiny A, cyklicky opakují tyto operace:

1. Vyhledání minimálního prvku množiny A.
2. Zápis tohoto prvku do množiny Z.
3. Vyřazení tohoto prvku z množiny A.

Na stejném principu lze postavit i jeden z algoritmů třídění (sorting) prvků pole. Předem ještě zdůrazněme, že algoritmus musí mít konečné, jednoznačné řešení. To by v hořejším příkladu mohlo být narušeno třebas nevhodným výběrem prvku s minimální hodnotou, čemuž jsme zabránili vyřazením zpracovaného prvku z původní množiny A. Je možné i takové řešení, při němž se další prvek bude vybírat porovnáním s naposledy zpracovaným prvkem. Potom jeho výběr závisí na typu testu, který může být např. buď >, nebo >=. V prvním případě by za situace, kdy původní množina A bude obsahovat několik shodných prvků, počet prvků nové, uspořádané množiny Z se zmenšil



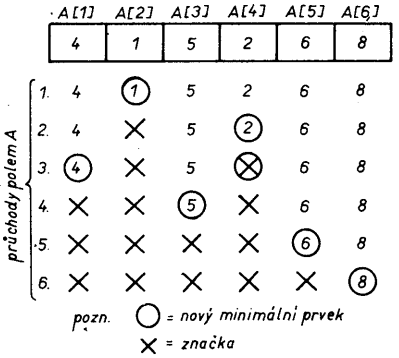
Aplikovat test pro výběr dalšího prvku $A_{min+1} > A_k$ by tedy znamenalo nejednoznačnost řešení vůči předchozímu příkladu, počet prvků uspořádané množiny Z by byl závislý na konkrétních hodnotách množiny A. Všechny podobné činitele je třeba při analýze algoritmu zvážit a při ladění programu důkladně prověřit.

Třídění pole s výběrem extrémního prvku

Následující dva programy ukazují třídění pole s využitím testu minimálního prvku, setříděné pole bude vždy uspořádáno vzestupně. První příklad je obdobou dosud uvažovaného algoritmu s tím, že je vztažen ke strukturám typu číselné pole integer.

Výstavba nového, vzestupně uspořádaného pole

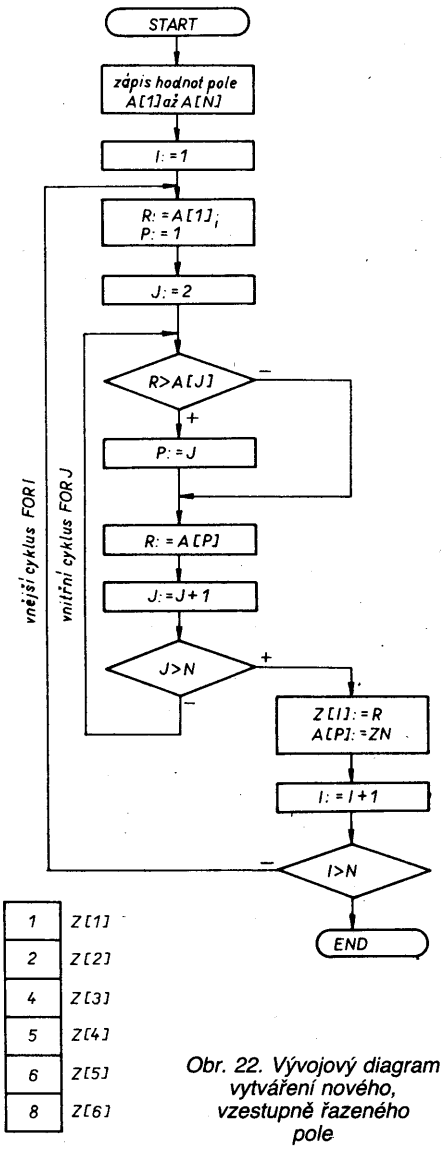
V tomto příkladu řešíme výstavbu nového, vzestupně uspořádaného celočíselného pole Z (1 až N) se stejným počtem prvků, jako má původní, neuspořádané pole A. Princip výběru minimálního prvku v každém průchodu pole A osvětluje obr. 21. Do referenční proměnné R je vždy uložen obsah proměnné A[1], který se postupně porovnává s dalšími proměnnými pole. V případě, že R > A[k], byl nalezen dosud nejmenší prvek pole. Ten je přesunut do referenční proměnné, R:= A[k] a do pomocné proměnné P je uložen index jeho původní pozice v poli A. Pokračuje se v průchodu až do konce původního pole. Tehdy je v referenční proměnné R uložen skutečně nejmenší prvek pole A, v pomocné proměnné P jeho původní index. Obsah proměnné R se ukládá na první volné místo v novém poli Z. Na původní pozici tohoto prvku v poli A, identifikovanou indexem, uloženým v proměnné P, se uloží značka, tvořená hodnotou, převyšující maximální možnou hodnotu libovolného prvku A. To je ekvivalentem vyřazení nalezeného minimálního prvku z dříve uvažované množiny. Celý cyklus hledání minimálního prvku se opakuje tolikrát, kolik prvků mají obě pole. Nakonec pole A obsahuje samé značky, pole Z vzestupně (podle velikosti) uspořádané prvky původního pole.



Obr. 21. Znázornění algoritmu třídění nově vytvářeného pole

Abychom si prakticky ověřili i dřívější úvahy, je na obr. 22 vývojový diagram zpracování uvažovaného algoritmu. Již z předchozího popisu vyplývalo, že bude probíhat ve dvou vložených cyklech. Vnější cyklus FOR určuje počet průchodů polem A, určený počtem jeho prvků, vnitřní cyklus zajišťuje pro každý průchod vyhledání minimálního prvku a jeho původní pozice, zápis prvku do nově vytvářeného pole Z a zápis značky využití tohoto prvku na jeho původní pozici v poli A. Povšimněme si toho, že vývojový diagram, i když byl kreslen co možná nejpřehledněji, včetně aktualizací řídicích proměnných obou cyklů, skutečně strukturované programování ani výběr příkazových struktur nijak nepodporuje. Při jeho konstrukci je třeba dávat pozor zvláště na nežádoucí účinky větvení programu a současně uvažovat i o možnostech jeho realizace pascalskými příkazy. Na druhé straně ovšem diagram poskytuje podrobný přehled o jednotlivých dílčích akcích, který lze u takto jednoduchých programů ještě snadno zvládnout.

Porovnejte si konstrukci vývojového diagramu s výkonnou částí výpisu programu



Obr. 22. Vývojový diagram vytváření nového, vzestupně řazeného pole

Tab. 7. Vytvoření nového setříděného pole

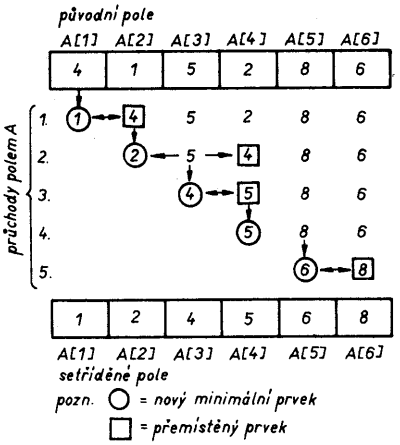
```
PROGRAM TMPAZ1;
CONST     N=6;
         ZN=999;
VAR
   I,J,R,P:INTEGER;
       A,Z:ARRAY [1..N] OF INTEGER;
BEGIN
  WRITELN ('ZADEJ HODNOTY POLE:');
  FOR I:=1 TO N DO READ (A[I]);
  FOR I:=1 TO N DO
    BEGIN
      R:=A[1];P:=1;
      FOR J:=2 TO N DO
         BEGIN
           IF R>A[J] THEN P:=J;
           R:=A[P]
         END;
         Z[I]:=R;A[P]:=ZN
      END;
  WRITELN ('OBSAH POLE A PO TRIDENI:');
  FOR I:=1 TO N DO WRITE (A[I]);
  WRITELN;
  WRITELN ('VZESTUPNE SETRIDENE POLE Z:');
  FOR I:=1 TO N DO WRITE (Z[I])
END.
```

```
RUN
ZADEJ HODNOTY POLE:
-15
423
-231
1232
562
-15
OBSAH POLE A PO TRIDENI:
999 999 999 1232 999 999
VZESTUPNE SETRIDENE POLE Z:
-231 -15 -15 423 562 999
```

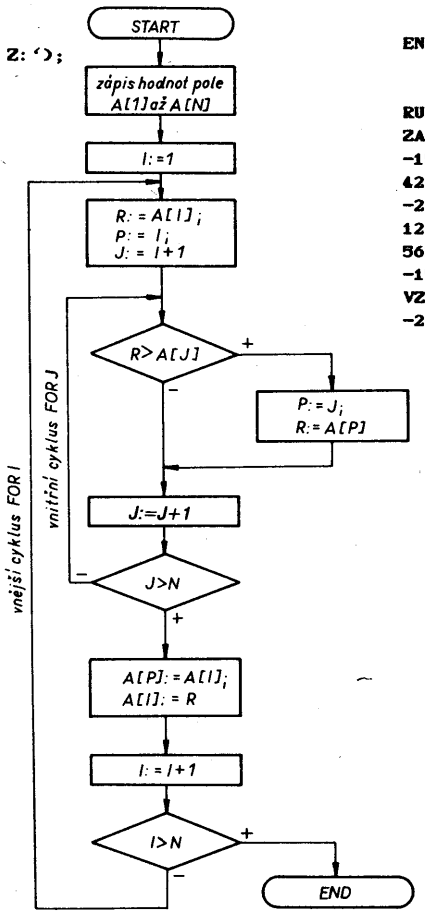
TMPAZ1 (Třídění výběrem Minimálního Prvku pole A do pole Z) v tab. 7. Zde jsou deklarovány konstanty N (volba počtu prvků obou polí) a ZN (hodnota, umísťovaná na pozici nalezeného minimálního prvku v poli A). V úseku proměnných jsou deklarovány řídicí proměnné dvou užitých cyklů FOR (proměnné I, J), referenční proměnná R, do níž se ukládá hodnota nejmenšího posledně nalezeného prvku a pomocná proměnná P, sloužící pro uložení indexu tohoto prvku. Konečně jsou deklarována i obě pole A, Z.

Na počátku programu jsou v cyklu FOR načteny hodnoty pole A, zadávané z klávesnice. Následuje vlastní třídicí program, odpovídající vývojovému diagramu. Skládá se ze dvou vložených cyklů FOR. Vnější cyklus s počtem opakování I = 1 to *N* určuje počet průchodů polem A, vnitřní cyklus s počtem opakování J = 2 TO *N* udává počet porovnání v každém průchodu. Vlastní porovnávání se uskutečňuje s využitím příkazu podmíněného větvení. Při každém výstupu z vnitřního cyklu je do pole Z uložen na nejnižší volnou pozici zleva další seřazený prvek, na jeho původní pozici v poli A značka. Při výstupu z vnějšího pole je již vytvořeno celé nové pole Z, původní pole A je přepsáno samými značkami.

V programu TMPAZ1 je, stejně jako v ostatních, uveden příklad činnosti po odstartování prováděcího režimu. Nejprve na náznak zadáme hodnoty prvků pole A, menší než deklarovaná hodnota ZN. Pak jsou vypsány hodnoty prvků obou polí po zpracování programu. Z příkladu vidíme, že v poli A jsou všechny původní hodnoty nahrazeny značkami kromě proměnné



Obr. 23. Algoritmus třídění v původním poli



Obr. 24. Vývojový diagram třídění v původním poli

A |4|, u níž byla záměrně zadána hodnota větší, než přípustná. Pole Z je vzestupně setříděné.

Vzestupné třídění v původním poli

Předchozí algoritmus i program lze snadno modifikovat tak, že nemusí být vytvářeno nové pole, ale že se prvky mohou třídit v původním poli. Princip je velmi jednoduchý. Je založen na tom, že do původního pole je zbytečné ukládat na místo nalezeného minimálního prvku značky tehdy, bude-li nalezený prvek umístěn přímo na jemu odpovídající pozici z hlediska třídění a na jeho původní místo bude uložen prvek, který mu tuto pozici musel uvolnit. Obojí v původním poli. Protože po prvním průchodu je v proměnné A | 1 |uložen minimální prvek celého pole, může další průchod polem začínat až od proměnné A | 2 | atd., čímž se ve srovnání

Tab. 8. Třídění s výběrem minimálního prvku, prováděné v původním poli

```
PROGRAM TMPAA2;
CONST     N=6;
VAR
       I,J,R,P,X:INTEGER;
              A:ARRAY [1..N] OF INTEGER;
BEGIN
  WRITELN ('ZADEJ HODNOTY POLE:');
  FOR I:=1 TO N DO READ (A[I]);
  FOR I:=1 TO N-1 DO
    BEGIN
      R:=A[I];P:=I;X:=I+1;
      FOR J:=X TO N DO
         BEGIN
           IF R>A[J] THEN
             BEGIN
               P:=J;R:=A[P]
             END
         END;
      A[P]:=A[I];A[I]:=R
    END;
    WRITELN ('VZESTUPNE USPORADANE POLE A:');
  FOR I:=1 TO N DO WRITE (A[I])
END.
```

```
RUN
ZADEJ HODNOTY POLE:
-15
423
-231
1232
562
-15
VZESTUPNE USPORADANE POLE A:
-231 -15 -15 423 562 1232
```

s předchozím algoritmem zmenšuje počet potřebných testů (obr. 23).

Vývojový diagram, obr. 24, ukazuje, že také řešení obou porovnávaných algoritmů budou velmi podobná. Vzhledem ke zkracování prohledávané části pole posuvem jejího počátku (I+1), vyplývajícímu z postupného umisťování nacházených minimálních prvků, je při každém vstupu do vnějšího cyklu FOR iniciován počáteční obsah referenční proměnné R:= A[I]. S tím je ve shodě i postupná změna řídicí proměnné vnitřního cyklu přiřazením J:= I+1. Obsah pomocné proměnné P je při každém vstupu do vnitřního prohledávacího cyklu iniciován indexem minimálního prvku, tvořícího referenci tohoto průchodu polem. Počet průchodů polem je roven *N*–1, protože zbývající, poslední prvek v původním poli, který dosud nebyl nalezen jako minimální, již musí být správně umístěn.

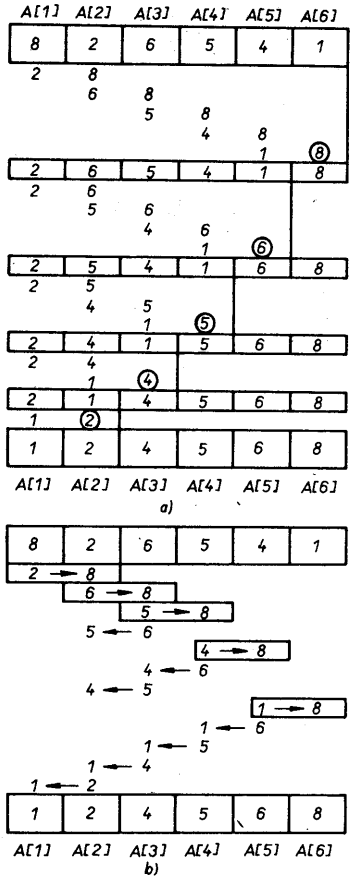
Vždy, když je v průběhu jednoho prohledávání pole nalezen nový minimální prvek, je jeho index uložen do pomocné proměnné P a jeho hodnota do referenční proměnné R. Na konci každého průchodu polem je vždy umístěn nově nalezený prvek tříděného pole na správnou pozici, na jeho místo přesunut původní prvek z této pozice. V případě, že hledaný prvek již byl v původním poli umístěn správně, přemisťování nenastane.

Odpovídající program TMPAA2 je v tab. 8. Tento druhý program, založený na již poněkud méně přirozeném algoritmu, je vůči předchozímu znatelně rychlejší.

Třídění záměnou (bubláním)

Tato metoda se od přechozích zásadně liší tím, že se přímo nehledá minimální prvek, ale vzájemně se vždy porovnávají dva sousední prvky a to takovým způsobem a tak dlouho, až při každém průchodu nepřímo nalezený extrémní prvek „vybublá" (metoda se označuje jako bubble sorting) na jemu odpovídající pozici. Třídění probíhá v původním poli.

Základní algoritmus nejlépe vystihne grafické znázornění praktického příkladu, obr.



Obr. 25. Metoda BUBBLE SORTING; a) příklad třídění pole s využitím algoritmu zaměňování sousedních prvků. Kroužkem je vždy označen prvek, který v příslušném průchodu „vybublal" na svoji příslušnou pozici, b) příklad třídění pole zaměňováním a využitím zpětných chodů. Běžně zaměňované páry jsou označeny rámečky, zpětné chody šipkami

25a. Při každém průchodu polem zleva se porovnávají hodnoty sousedních prvků J a K = J+1. Platí-li, že A [J] < A [K] porovnávají se následující dvojice A [J] ← A [K] a A [K] ← A [K + 1]. Pokud má však nižší prvek v porovnávané dvojice větší hodnotu než následující, tedy A [J]>A [K], zamění se hodnoty obou prvků v poli. S ukončením každého průchodu vyplyne hodnota dosud nezpracovaného největšího prvku pole na jemu odpovídající pozici.

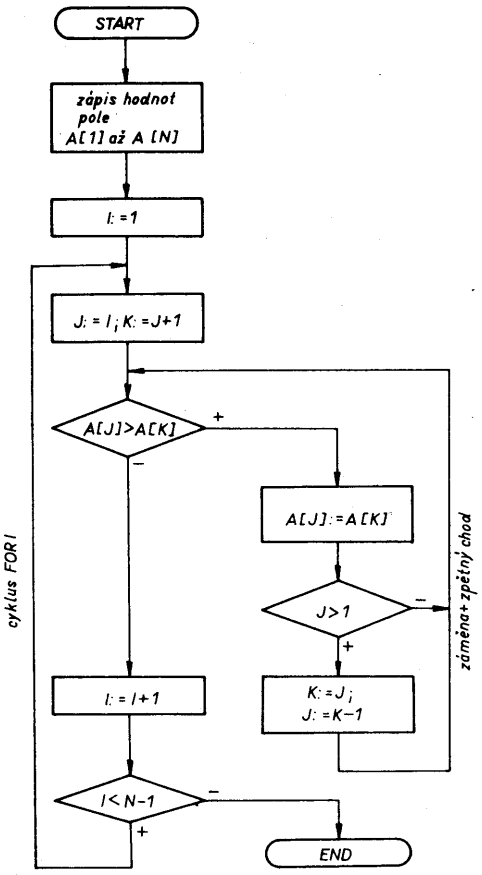
Efektivnost (tj. rychlost) metody podstatně zrychluje zavedení tzv. zpětného chodu. Princip: Je-li při porovnání zjištěna potřeba záměny příslušných prvků, provede se stejně jako v předchozím případě. Následuje porovnání dvojice pořadově nižších sousedních prvků s indexy K = J, J = K −1. Je-li hodnota nižšího prvku dvojice větší než hodnota prvku vyššího, prvky se opět zamění a pokračuje se v dekrementaci indexů prvků, jejich porovnávání a záměnách zpětnými chody do takové hloubky, až se narazí na uspořádaný blok pole. Potom pokračuje zpracování pole dvojicí prvků, následující za původní záměnou. Takto se dosahuje setřídění pole v jediném průchodu, který ovšem může obsahovat velké množství zpětných chodů, závislé na původním obsahu pole. Významné je, že při každé záměně se může narušit již setříděná část pole, která pak opět musí být řešena pomocí zpětných chodů (obr. 25b).

Odpovídající vývojový diagram je na obr. 26. Prohledávání *n*prvkového pole probíhá v konečném počtu *N*-1 cyklů. Při A[J]<A[K] jsou v každém cyklu pouze porovnávány dva sousední prvky s odpovídajícími indexy. Při A [J]>A[K] dochází k záměně a navíc, při J>1 je automaticky zařazen test na případný zpětný chod.

Než se podíváte na dvě navazující ukázky možného programového řešení, zkuste si vhodný program navrhnout sami.

Tab. 9. Bublinové třídění s využitím příkazu GOTO

```
PROGRAM BUBS3;
LABEL 6;
CONST N=6;
VAR
    I,J,K,P:INTEGER;
        A:ARRAY [1..N] OF INTEGER;
BEGIN
  WRITELN ('ZADEJ HODNOTY POLE:');
  FOR I:=1 TO N DO READ (A[I]);
  FOR I:=1 TO N-1 DO
    BEGIN
      J:=I;K:=J+1;
    6:IF A[J]>A[K]THEN
        BEGIN
          P:=A[K];A[K]:=A[J];A[J]:=P;
          IF J>1 THEN
            BEGIN
              K:=J;J:=K-1;
              GOTO 6
            END
        END
    END;
  WRITELN ('SETRIDENE POLE:');
  FOR I:=1 TO N DO WRITE (A[I])
END.


RUN
ZADEJ HODNOTY POLE:
4
8
-8
45
-3
3
SETRIDENE POLE:
-8  -3  3  4  8  45
```



Obr. 26. Vývojový diagram třídění pole na principu algoritmu bubble sorting, doplněného využitím zpětných chodů

Tab. 10. Bublinové třídění bez užití skokového příkazu

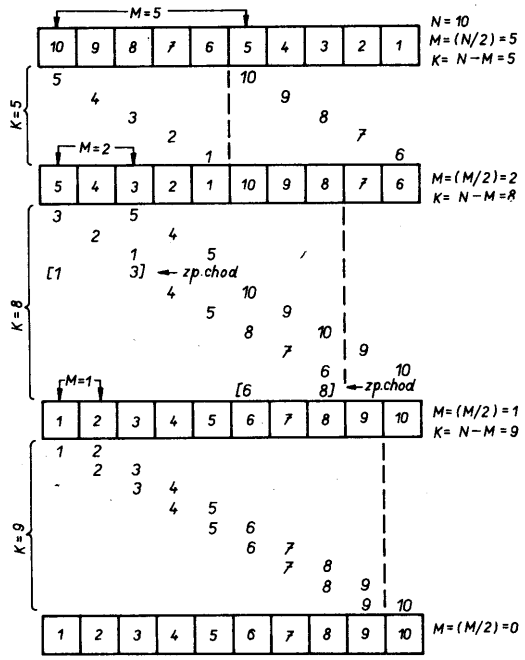
```
PROGRAM BUBS4;
CONST N=6;
VAR
        I,J,K,P:INTEGER;
            A:ARRAY [1..N] OF INTEGER;
BEGIN
  WRITELN ('ZADEJ HODNOTY POLE:');
  FOR I:=1 TO N DO READ (A[I]);
  FOR I:=1 TO N-1 DO
    BEGIN
      J:=I;K:=J+1;
      WHILE A[J]>A[K] DO
        BEGIN
          P:=A[K];
          A[K]:=A[J];
          A[J]:=P;
          IF J>1 THEN
            BEGIN
              K:=J;J:=K-1
            END
        END
    END;
  WRITELN ('SETRIDENE POLE:');
  FOR I:=1 TO N DO WRITE (A[I])
END.

RUN
ZADEJ HODNOTY POLE:
4
8
-8
45
-3
3
SETRIDENE POLE:
-8  -3  3  4  8  45
```

První program BUBS3 využívá pro volání zpětného chodu příkazu skoku GOTO na návěští, identifikující příkaz podmíněného větvení. Zmínili jsme se již několikrát o tom, že tento příkaz, umožňující skok do libovolné části programu, může zcela narušit programovou bezpečnost porušením hierarchické struktury. V tomto případě, kdy je program krátký a přehledný, a kdy ke skoku dochází uvnitř jediného bloku, lze ovšem skok použít. Navíc taková struktura programu dobře odpovídá jeho případné symbolice pro optimalizaci asemblerového řešení.

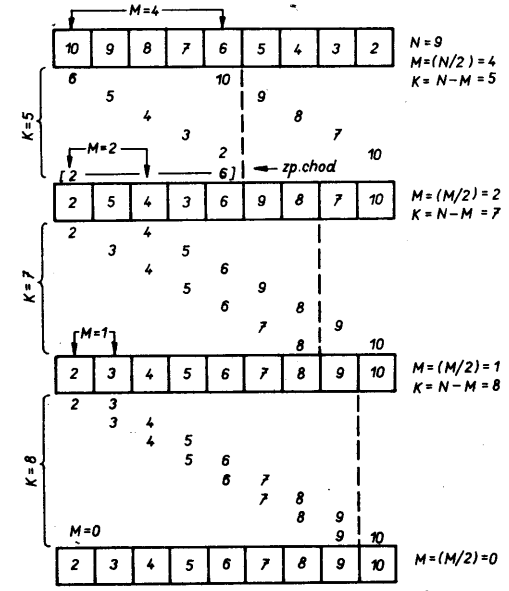
Alternativou programu BUBS3, tab. 9, je druhý, prakticky shodný program BUBS4, tab. 10, v němž je potřeba příkazu skoku vyloučena užitím vloženého podmíněného cyklu WHILE. Chápejme porovnání obou programů jako ukázku toho, že promyšleným návrhem programu a výběrem příkazů pascalské struktury lze skutečně potřebu skokových příkazů prakticky vyloučit.

Shellův algoritmus

Dalšího zkrácení doby, potřebné k třídění v poli určité délky se dosahuje využitím Shellova algoritmu, patřícího k nejrychlejším. Jeho smyslem je omezit počet potřeb-



Obr. 27. Průběh třídění pole se sudým počtem prvků



Obr. 28. Průběh třídění pole s lichým počtem prvků

ných komparací a zpětných chodů, jak jsme je viděli v posledním příkladu třídění záměnou prvků. Toho se dosahuje tím, že záměny neprobíhají v přísně sekvenčním pořadí sousedních prvků, ale jsou, podle Shellova algoritmu, organizovány tak, aby při následujících záměnách a zpětných chodech byl minimálně narušován již setříděný blok.

Podstatu algoritmu osvětluje obr. 27. Celočíselným půlením počtu prvků vstupního pole se určí doplňkový koeficient $M = (N/2)$, hodnoty indexů prvků levé části takto rozděleného pole se mohou pohybovat v rozmezí 1 až K, kde $K = N - M$. V prvním průchodu polem se uskuteční K porovnání a případné výměny obsahů příslušných prvků pole, jejichž ofset je v každém průchodu určen koeficientem M. V prvním průchodu pole se sudým počtem prvků se neuskutečňuje žádný zpětný chod, důvodem je nutnost zabránit překročení rozsahu pole na jeho počátku. Výjimka může nastat u pole s lichým počtem prvků, kdy bude poslední index $I = K > M$ (obr. 28).

V každém dalším průchodu se podle algoritmu nově aktualizuje počet potřebných porovnání i ofset porovnávaných prvků. Ofset se stále zmenšuje a odpovídá vždy celočíselnému půlení hodnoty ($M/2$), počet porovnání $K = N - M$ se tím naopak zvětšuje. Při každé záměně A [I] ↔ A [$I + M$] je v případě splněné podmínky $I > M$, kde I = index aktuálního prvku na levé straně pole, prováděn test na přípustnost a případné zařazení zpětného chodu. Tak se postupně pole setřiďuje, bezpečně je setřídění ukončeno při $M = 0$. Vidíme, že organizací prováděných záměn podle Shellova algoritmu je minimalizován jejich počet i potřebný počet zpětných chodů při zachování jednoduchého průběhu zpracování.

Vývojový diagram, popisující průběh akcí při třídění, je na obr. 29, odpovídající pascalský program je v tab. 11.

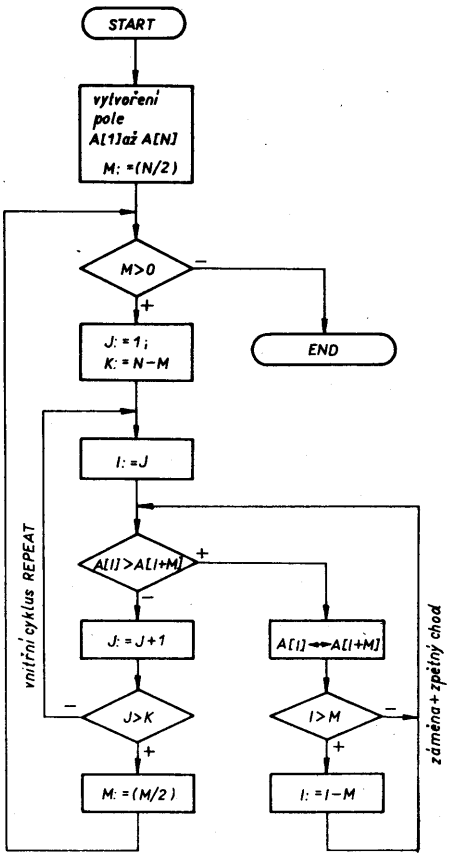
Program se v podstatě skládá ze tří vložených podmíněných cyklů. Před prvním vstupem do vnějšího cyklu WHILE (tedy cyklu s testem zahájení) je inicializována počáteční hodnota proměnné $M = (N/2)$. V tomto

Tab. 11. Třídění s využitím Shellova algoritmu

```
PROGRAM SHELL5;
CONST    N=10;
VAR
    I,J,K,M,P:INTEGER;
           A:ARRAY [1..N] OF INTEGER;
BEGIN
  WRITELN ('ZADEJ HODNOTY POLE:');
  FOR I:=1 TO N DO READ (A[I]);
  M:=N DIV 2;
  WHILE M>0 DO
    BEGIN
      J:=1;K:=N-M;
      REPEAT
        I:=J;
        WHILE A[I]>A[I+M]DO
          BEGIN
            P:=A[I];A[I]:=A[I+M];A[I+M]:=P;
            IF I>M THEN I:=I-M
          END;
        J:=J+1;
      UNTIL J>K;
      M:=M DIV 2;
    END;
  WRITELN ('SETRIDENE POLE:');
  FOR I:=1 TO N DO WRITE (A[I])
END.
```

```
RUN
ZADEJ HODNOTY POLE:
2
4
7
4
8
-9
-5
6
-9
4
SETRIDENE POLE:
-9  -9  -5  2  4  4  4  6  7  8
```

Obr. 29. Vývojový diagram třídění na principu Shellova algoritmu



cyklu, určujícím podle rozsahu pole počet jeho potřebných průchodů, je vložen cyklus REPEAT (s testem ukončení). Před každým vstupem do tohoto cyklu, určujícího počet potřebných základních komparací A ⌈*I*⌉ : A ⌊*I* + *M*⌋ jsou nastaveny mezní indexy $J = 1$, $K = N - M$. Konečně nejnižší vložený cyklus WHILE zajišťuje v případě výsledku předchozího testu A [*I*] > A [*I* + *M*] = true realizaci testu na přípustnost zpětného chodu (podmínka *I*>*M*) a je-li test pozitivní, inicializují se indexy odpovídajícího páru prvků jejich zpětným posuvem o ofset *M* pro komparaci a případnou záměnu zpětného chodu.

Výstup z vnitřního cyklu REPEAT je podmíněn splněním podmínky *J*>*K*. Po aktualizaci proměnné $M = (M/2)$ následuje vždy nový vstup do vnějšího cyklu a tedy nový průchod polem s novým počtem i ofsetem jeho porovnávaných dvojic prvků. Programový cyklus je ukončen s posledním vstupem do vnějšího cyklu, kdy hodnota proměnné $M = 0$.

Budete-li se chtít přesvědčit o tom, odpovídá-li průběh třídění popsanému algoritmu a uvedeným příkladům v jednotlivých obrázcích, není nic snazšího. Vložte do programu za konec cyklu REPEAT řádky

```
WRITELEN;
FOR Z:=1 TO N DO WRITE A [Z];
```

Je pochopitelně třeba také deklarovat řídicí proměnnou cyklu. Počet zadávaných prvků pole lze měnit změnou konstanty *N*. Program vypíše obsahy pole při jeho jednotlivých průchodech, tedy vlastně průběh třídění včetně zařazovaných zpětných chodů.

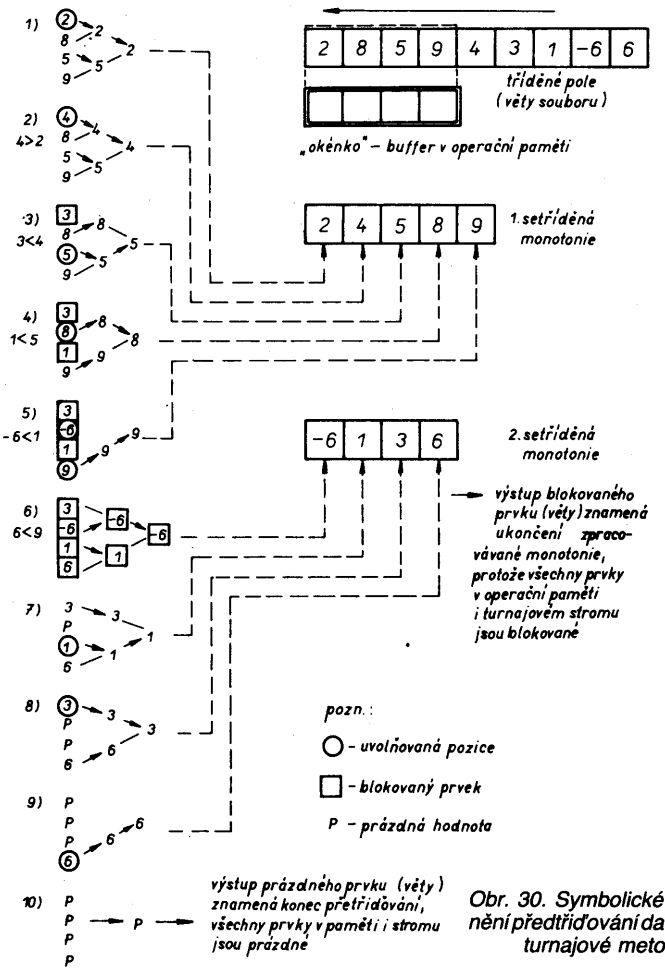
Uvedené algoritmy, které jsme právě probrali, tvoří pouze část z mnoha známých metod třídění datových množin. Patří však k nejužívanějším a nejdůležitějším, jejich znalost se proto jistě uplatní i v praxi. Omezením pro užívání takovýchto algoritmů je však skutečnost, že celé tříděné pole (nebo jiná datová struktura) musí být v průběhu celého třídění uloženo v operační paměti. Ve všech předchozích příkladech jsme pro přehlednost vždy uvažovali zpracování pouze několikaprvkového pole. V praxi ovšem mohou být tříděné nebo jinak zpracovávané datové množiny značně rozsáhlé. Mohou být například tvořeny datovými soubory, uloženými v externí paměti, které často svým rozsahem mnohonásobně převyšují volnou kapacitu datové oblasti operační paměti (RAM). Pro třídění takových souborů jsou již uvedené algoritmy nevhodné.

Předtřiďování, slévání

Všimněme si alespoň symbolicky také přístupu ke třídění rozsáhlých souborů. Z praktických důvodů (kapacita operační paměti) probíhá vždy v podstatě ve dvou fázích.

První fází je tzv. *předtřiďování*. Je založeno na předpokladu, že pokud nemůže být celá zpracovávaná množina dat zavedena do operační paměti najednou, musí být zpřístupňovaná po jednotlivých položkách, tedy například po větách souboru. Postupným zpracováním vstupních dat se vytvoří několik kratších, vnitřně setříděných datových bloků (tzv. monotonií), které dohromady obsahují buď všechny věty původního souboru, nebo (z praktických důvodů) častěji setříděné odkazy na tyto věty, jejich klíče (indexy). Tyto uspořádané monotonie jsou pak ve druhé fázi zpracování, označené jako *slévání*, opět postupně, po jednotlivých větách, slučovány do konečné, lineárně uspořádané, jediné setříděné množiny (souboru).

Vhodný a velmi zajímavý algoritmus představuje pro předtřiďovací fázi tzv. *turnajová metoda*, jejímž principem je hierarchické binární třídění. Pro snadné pochopení použijeme opět grafické znázornění postupu zpracování jednoduchého příkladu, obr. 30, věty souboru nahradíme prvky jednoduchého číselného pole. Předpokládáme, že naznačené 9prvkové pole nemůže být tříděno najednou, pro jeho zpracování je k dispozici podstatně menší kapacita operační paměti, označená symbolicky 4prvkovým „okénkem".



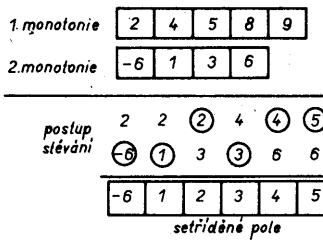
Obr. 30. Symbolické znázornění předtřiďování dat pomocí turnajové metody

V každém okamžiku se tedy třídění mohou aktivně účastnit pouze čtyři prvky celého pole. Při zahájení předtřiďovací fáze se do okénka zapíší první čtyři prvky vstupních dat. Prvotním binárním porovnáním sousedních dvojic jsou „určeni vítězové 1. kola", v našem příkladu prvky s hodnotami 2, 5. Ti postupují, podle binární stromové struktury do dalšího kola atd., až se nakonec střetnou dva jediní neporažení vítězové všech dosavadních kol turnaje ve finále. Tam je určen vítěz celého turnaje, v našem příkladu má vítězný prvek hodnotu 2. Takto by třídění probíhalo pouze v případě, že bychom postupně vyřazovali už zpracované prvky, tj. vítěze jednotlivých turnajů, do množiny výstupních dat. Aby nemohl být narušen setříděný úsek, mohli bychom na pozici vyřazeného prvku v okénku vždy uložit značku ZN. Tím by však nemohlo být zpracováno více prvků, než kolik jich obsáhne okénko současně.

Podstatou vlastního předtřiďování je to, že na pozici vítěze každého turnaje se neukládá žádná značka, ale další položka (prvek, věta) z pořadí dosud netříděné vstupní datové množiny. K jejímu zařazení do okénka lze výhodně využít zpětného chodu po původní cestě vítěze tohoto turnaje. Před zařazením však musí být hodnota nové položky N porovnána s právě vyřazovanou V. Je-li N>V, může být do okénka zařazena bez omezení. Je-li však N<V, tj. je „lepší než vítěz předchozího turnaje", musí být opatřena tzv. příznakem blokování, který zabraňuje jejímu srovnávání s běžnými položkami, pro které prostě neexistuje. Blokovaná položka může být využita až tehdy, když není v okénku žádná položka neomezená. Tím se zabraňuje narušení pořadí tříděných hodnot a současně umožňuje do okénka zařazovat další položky.

Po každém dalším novém obsazení třídicího okénka vždy začíná další „turnaj", vítězové tvoří další prvky výstupní, setříděné datové množiny, na jejich místa vstupují prvky dosud netříděné množiny vstupní. Při tom v okénku nutně přibývá blokovaných položek, až nakonec těmito položkami bude tvořeno celé okénko. V bezprostředně následujícím turnaji proto bude jeho vítězem blokovaná položka. Tím je ukončeno první předtřiďování, je vytvořen první blok setříděných výstupních dat, tzv. *monotonie*. Všechny zbývající položky okénka jsou okamžitě odblokovány a původně blokovaný vítěz je zařazen na první pozici další vytvářené výstupní monotonie. Nakonec, s vyčerpáním všech prvků množiny vstupních dat, nutně dochází ke čtení prázdných položek do okénka. Vstoupí-li z některého turnaje prázdný prvek, znamená to, že jsou prázdné všechny položky okénka, tím je i ukončeno zpracování poslední monotonie a tedy i celé fáze předtřiďování. Celý algoritmus přehledně postihuje obr. 30.

Teprve po vytvoření monotonií dochází ke slévání celého setříděného souboru. Samotný princip je jednoduchý, v návaznosti na předchozí příklad ho postihuje obr. 31. Zde jsou pro přehlednost uvažovány pouze dvě monotonie, ve skutečnosti jich ovšem může být mnohem více. Vzhledem k možnému rozsahu jednotlivých monotonií je slévání opět prováděno po jednotlivých položkách (prvcích, větách . .). Nejprve se vždy porov-



Obr. 31. Princip slévání monotonií

nají první položky všech setříděných monotonií, vítězná se uloží na první pozici setřiďovaných dat a mezi porovnávanými položkami je nahrazena pořadově bezprostředně následující položkou ze stejné monotonie. Následuje hledání další vítězné položky. Postupně dochází k vyčerpání jednotlivých monotonií. Nakonec, s vyčerpáním poslední monotonie, je přečtena prázdná položka a slévání je ukončeno. Je vytvořena setříděná množina výstupních dat o stejném počtu prvků, jako měla nesetříděná množina vstupní.

I když se uvedené příklady týkaly pouze úzce vymezené oblasti, jistě naznačily jak význam, tak i zajímavost studia algoritmických postupů. Doufáme, že objasnily i základy práce s Pascalem.

Zápis, překlad, ladění a ověřování programu

Postup, jakým musí projít uživatelský program od návrhu až do fáze praktického využití, v podstatě vystihuje už název této kapitoly.

Program se na počítači vytváří, zapisuje pomocí vhodného obslužného programu, tzv. textového editoru, přes standardní konzolu (klávesnici, displej). Výsledkem je zdrojový tvar programu. Pro pohodlnou práci musí editor, jehož produktem je textový soubor, umožňovat zápis a edici jednotlivých řádků programu. Tedy nejen zadávání, ale i doplňování, rušení, vyhledávání, přesouvání a spojování znaků, řádků i celých textových bloků. Vytvořený zdrojový textový soubor (ASCII, EBCDIC) se z praktických důvodů ukládá na nosné médium vnější paměti. Pro kontrolu, studium nebo evidenci je možný jeho výpis na tiskárně.

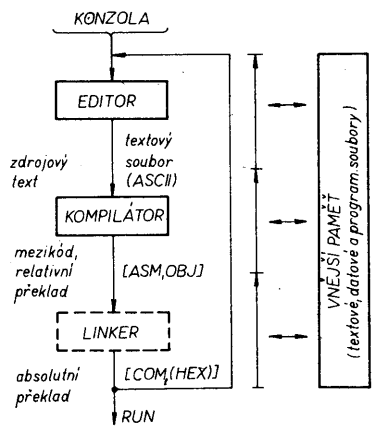
Při vytváření zdrojového textu se samozřejmě nevyhneme chybám. Ty mohou buď syntaktické (nesprávně napsaný, chybějící nebo přebytečný znak či operátor, klíčové slovo, proměnná, řádek, deklarace . .), nebo sémantické (špatně navržený program nebo jeho část, případně i detail). Na žádný z uvedených typů chyb nás textový editor při zápisu zdrojového programu neupozorňuje. V Pascalu to není technicky možné ani účelné vzhledem k systému hierarchii programových modulů a proměnných.

Zdrojový program musí být dále přeložen do cílového, spustitelného strojového kódu procesoru. Překlad zajišťuje další speciální program, tzv. *překladač* (kompilátor). Vedle vlastního překladu musí kompilátor umožňovat také interaktivní formu postupné kontroly a oprav syntaktické správnosti programu. Kompilace z řady důvodů většinou neprobíhá naráz, avšak postupně v několika fázích, které mnohdy, zvláště na malých počítačích bez diskových pamětí, ani nebývají viditelně rozlišeny. V první fázi je zdrojový pascalský text programu překládán do mezikódu na úrovni makroasembleru jazyka symbolických adres. Překlad probíhá podle obdobného schématu, jako překlad z jedné přirozené řeči do druhé, využívající slovníku. Každé definici, deklaraci, příkazu atd. kompilátor přiřazuje prefabrikované bloky makroinstrukcí. Bezprostředně po spuštění kompilace je, počínaje prvním řádkem, program kontrolován syntakticky. Jakmile kompilátor narazí na chybu, překlad se zastaví a celoobrazovkový nebo řádkový editor vhodným náznakem a textem (chybové hlášení) více nebo méně přesně upozorňuje na pozici výskytu a typ chyby. Většina chyb bývá patrná na první pohled, rafinovanější výskyty identifikujeme pomocí tabulky chybových hlášení (např. HP 4 T jich obsahuje na 70). Nalezené chyby postupně opravujeme v automaticky aktivovaném edičním módu, znovu spouštíme kompilaci atd., až je nakonec přeložen a tím i i syntakticky opraven celý program.

Jednoduché kompilátory umožňují překlad pouze celého, úplného zdrojového programu. Pak zpravidla po úspěšném průchodu celým kompilátorem následuje automatický překlad z mezijazyku ASM do strojového kódu v absolutních adresách. Výsledkem je tedy v operační paměti umístěný program ve strojovém kódu, který již může být spuštěn.

Některé komfortní kompilátory, užívané na počítačích s diskově orientovanými operačními systémy, umožňují překlady programových modulů rozsáhlého zdrojového programu. Tyto moduly musí být vzhledem k jejich možnému následnému spojování vytvářeny jako přemístitelné (relokativní), tedy ne s absolutními adresami. Ze stejného důvodu musí být možné i vzájemné odvolávky mezi jednotlivými moduly. Jednotlivé moduly se pak spojují ve výsledný uživatelský program. K tomu se užívá další obslužný program, tzv. *spojovač* (linker), respektující vzájemné vazby a odvolávky všech modulů. Výsledný spojený program může být generován jak v absolutním strojovém kódu, tak ve vhodném účelovém mezikódu, například Intel HEX.

Následuje etapa vlastního ladění programu, tedy jeho detailní sémantická kontrola. V tomto ohledu ovšem může být systémová podpora jen omezená. Nejjednodušší implementace pouze podporují kontrolu přípustných mezí výpočtu v dynamickém režimu, po spuštění běhu přeloženého programu. Funkci programu ověřujeme vyšetřováním jeho reakce na zadávaná vzorová data, včetně jejich extrémních hodnot a kombinací i všech možných pracovních podmínek. Fatální chyby, jako je překročení adresového prostoru nebo rozsahu dat, které by mohly způsobit i zhroucení systému, vyvolávají automatické zastavení běhu programu a výpis chybového hlášení (runtime error). Diagnózu chyby usnadňuje křížový výpis programového úseku, obsahující vždy minimálně aktuální stav programového čítače v místě výskytu chyby a odpovídající řádky zdrojového textu.



Obr. 32. Znázornění postupu zápisu, překladu, ladění a spojování pascalského programu

Vyšší programová vybavení ovšem nabízejí i při ladění programu komfort podstatně větší. Za určitých podmínek, které musí být dodrženy už při kompilaci, dovolují například i zpětnou kompilaci přeloženého, laděného programu do pascalského tvaru. Při dodržení jiných podmínek je možné nasadit ladicí program (debugger), umožňující užívat krokování programu, definovat programové zarážky, zobrazovat hodnoty proměnných, parametry procedur aj.

Z obou předchozích odstavců vidíme, že pojem strojově nezávislý jazyk neplatí do všech důsledků. Rozdíly mezi různými implementacemi Pascalu v podstatě vyplývají ze tří hlavních činitelů, kterými jsou šířka toku dat, užitý typ procesoru a konfigurace prostředků konkrétního počítače, a konečně operační systém, pod kterým počítač a tedy i užitá implementace pracuje. Při praktické práci je tedy nutno nejprve zvládnout dialekt Pascalu na konkrétní implementaci a možnosti jejího využití. Obojí vždy bývá popsáno v manuálu. Klasickým příkladem mohou být jednotlivé verze Turbo Pascalu, v nichž jsou možnosti praktického využití vůči standardnímu Pascalu podstatně rozšířeny jak v návaznosti na vývoj v oblasti programování, tak moderních operačních systémů.

Operační systémy

Již několikrát jsme se dotkli pojmu operační systém. Na dalších stránkách si operačních systémů budeme všímat blíže.

Nějaký operační systém je instalován na každém počítači, určeném pro univerzálnější aplikace. Programy, vytvářené pod stejným operačním systémem jsou, s určitými omezeními, přenositelné mezi obecně různými typy počítačů. Protože to je mimořádně významná vlastnost, je důležitá standardizace těchto systémů. Bez operačního systému pracují pouze jednoúčelové počítače, tedy v podstatě logické automaty, jejichž činnost je určena pevným naprogramováním a konfigurací technických prostředků pro konkrétní aplikaci.

Smyslem operačního systému je především podpořit práci uživatele počítače. Operační systém je tvořen souborem programového vybavení, které řídí průběh zpracování uživatelských programů a tomu odpovídající součinnost technických prostředků, které používá a na které také klade určité nároky. Obecně se každý operační systém skládá ze dvou částí:

- systémového programového vybavení, zajišťujícího dosud uvažované funkce,
- doplňkového programového vybavení, jehož smyslem je podpořit vlastní práci uživatele.

Poněkud konkrétněji lze skupiny programů, vytvářejících operační systém, dělit asi takto:

1. Řídicí (organizační) program

Tento program, tvořící jádro systémového programového vybavení, se skládá ze dvou částí.

Systém *řízení činností* zajišťuje inicializaci systému, běh i ukončení uživatelského programu, kterému přiděluje i volné místo v operační paměti a vstupní/výstupní zařízení.

Systém *správy dat* řídí a eviduje umísťování a přístup k vnějším souborům s využitím

operativně přístupných vnějších paměťových médií (floppy, hard disk).

2. Překladače

Tuto skupinu představují programy pro překlad různých zdrojových programů do požadovaného cílového tvaru.

3. Obslužné programy

Vytvářejí pomocné systémové vybavení, které je uživateli kdykoli k dispozici, a tak usnadňuje často se opakující nebo velmi namáhavé rutinní práce. S řadou těchto programů jsme se již seznámili (editory, loadery . .), dalších si všimneme později.

Operačních systémů dnes pochopitelně existuje celá řada, vyvíjely se dlouhou dobu a tento proces není ukončen a pravděpodobně asi ani nikdy nebude. Operační systémy se liší jak podle technické vyspělosti jednotlivých generací a typů počítačů, tak podle tříd úloh, pro které jsou určeny.

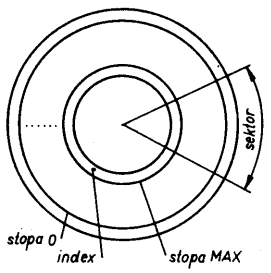
Do poměrně nedávné doby byly operační systémy doménou velmi omezeného okruhu specialistů. V souvislosti s hromadným rozšiřováním osobních počítačů s vnějšími diskovými a disketovými pamětmi se však dostávají do oblasti zájmu široké technické veřejnosti.

Diskové paměti

Mimořádnou šanci, přinášející technické i ekonomické zvládnutí hromadné výroby jednotek diskových pamětí a jejich integrovaných řadičů, vycítili brzo jak výrobci osobních mikropočítačů, tak softwarové firmy. I když se zpočátku mnoho z nich snažilo na trhu prosadit samostatně svými koncepcemi a řešeními, bylo patrné, že narůstající Babylon může zastavit pouze buď nějaké univerzální řešení operačního systému, vyhovující všem výrobcům osobních mikropočítačů (nebo alespoň naprosté většině), nebo naopak mocná, bezkonkurenčně dominující firma, která svou koncepci vnutí ostatním. Obojí se pak víceméně událo a tato tendence se projevuje i v dalším vývoji.

Pro usnadnění orientace v operačních systémech si nejprve povšimneme základních principů technické realizace a fyzické organizace záznamu na disketě jednotky floppydiskové paměti. Záznam, založený na feromagnetickém principu, se ukládá na rotující disketu v určitém, podle typu jednotky specifikovaném formátu. V každém případě však záznam vždy probíhá na definovaném, formátem určeném počtu stop. Každou stopu tvoří samostatná, uzavřená a od ostatních stop oddělená kruhová dráha, obr. 33. Aktuální stopa se vybírá lineárním pohybem čtecí/záznamové hlavičky po dráze, odpovídající poloměru diskety.

Základním požadavkem, kladeným na diskovou paměť, je pochopitelně především rychlost, bezpečnost a kompatibilita prováděného záznamu, ale i jeho znovuvyhledávání. To znamená, že organizace dat, ukládaných na disketu, způsoby přístupu k nim i jejich zabezpečení musí odpovídat určitým závazným normám. První z těchto norem je vždy formát diskety. K formátování se využívají speciální instrukce řadiče, zpravidla před prvním použitím diskety.



Obr. 33. Fyzická organizace záznamu na stopy O až MAX diskety a jejich rozdělení do sektorů

Na obr. 34 je hrubé znázornění standardního formátu jedné stopy floppydisku. Prvním krokem na cestě k organizaci ukládání dat je definované rozdělení všech stop diskety, jejichž kapacita je příliš velká, na menší úseky, tzv. sektory. Dalším krokem je potřeba identifikovat stopy i sektory, zabezpečit kontrolu jejich obsahu a přístupu k nim.

Počátek každé stopy je možno odvodit od indexového otvoru v disketě, obr 33. Na tento otvor navazuje první záznam ve formátu stopy, tzv. indexová adresová značka IAM, podle které řadič identifikuje počátek stopy. Po oddělovací mezeře již následují formátovaná pole záznamů jednotlivých sektorů. Každý sektor je rozdělen do dvou částí:

1. Identifikátor sektoru

představuje služební blok, začínající speciální identifikační značkou IDAM, využívanou pro synchronizaci odělení dat řadičem (viz dále). Další položky tohoto bloku obsahují informace o čísle stopy, straně diskety, čísle a délce sektoru, pomocí nichž může řadič kontrolovat správnost přístupu, a konečně pole zabezpečovacích bytů CRC.

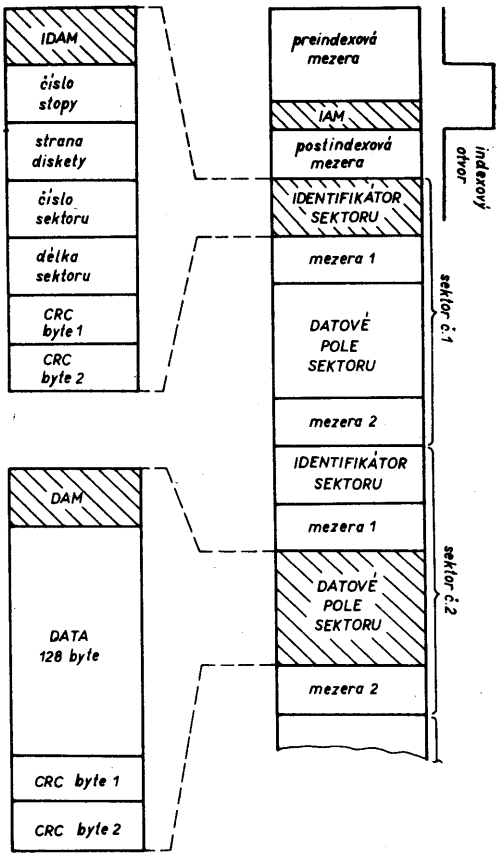
2. Datový blok

nesoucí vlastní informační obsah příslušného sektoru je opět oddělen příslušnou, definovanou mezerou. Také datový blok je uvozen speciální, datovou adresovou značkou DAM, následuje pole uživatelských dat a celý blok je opět uzavřen zabezpečujícími byty CRC.

Uvedeným způsobem jsou, odděleny příslušnými mezerami, zaznamenávány na stopě postupně jednotlivé sektory, jeden za druhým.

Pro identifikační značky IAM, IDAM, DAM musí být použito speciální kódování, odlišné od užité kódovací metody běžných dat. To proto, aby náhodný obsah datových a služebních bloků a polí nemohl být mylně interpretován jako některá ze značek. K tomu se používá princip vypouštění přesně definovaných taktových impulsů. Touto problematikou se zabývat nebudeme, obdobná řešení se užívají i v jiných oblastech, například při organizaci komunikace v přenosových sítích. Blíže si však povšimneme obou základních principů vlastního kódování dat, zapisovaných na disketu.

Data mohou být na stopu zaznamenávána pouze v „sériovém" tvaru, přičemž jednomu zapisovanému „bitu" vždy odpovídá změna smyslu magnetické orientace saturovaného elementu diskety vůči předchozímu. Mechanická orientace záznamu je podélná, ve směru rotace. Vzhledem ke konečné stabilitě i přesnosti a tolerancím rychlosti otáčení disket na jednotlivých mechanikách využívají obě metody společného a vlastně jediného možného základního principu – řízení záznamu i čtení systémovým taktem, přičemž

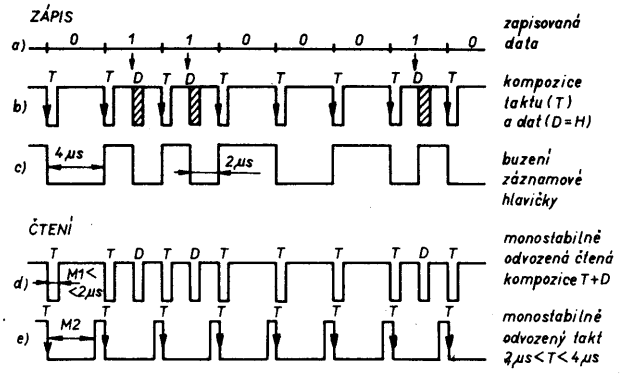


Obr. 34. Znázornění záznamu sektorů na stopě floppydisku

tento takt se vhodným způsobem komponuje s užitečnými daty a tvoří tak nedílnou součást záznamu na disketě.

Pro pochopení obou základních kódovacích systémů FM (Frequency Modulation nebo také SD – Single Density) a MFM (Modified FM, popř. DD – Double Density) je vhodné uvědomit si předem, že vlastnímu zápisu na disketu těsně předchází dělení kmitočtu již zakódované kompozice v poměru 1:2, čtený signál je naopak před zpracováním v poměru 2:1 násoben. Tímto vtipným způsobem je dosaženo již zmíněné střídavé magnetické orientace sousedních záznamových elementů (nezávisle na jejich obsahu), což prakticky omezuje počet potřebných magnetizačních změn zhruba na polovinu, potlačuje vliv rychlosti otáčení při záznamu a čtení na bezpečnost zpracovávaných dat a současně umožňuje dosáhnout větší hustoty záznamu.

Princip kódování záznamu *metodou FM* je na obr. 35. Stopa a) postihuje sled bitů, které mají být zpracovány jako data. Čisté impulsy T na stopě b) představují hodinové impulsy



Obr. 35. Znázornění kódování záznamu metodou FM

systémového taktu s konstantní periodou 4μs. Při kódování datových bitů s úrovní L se kompozice zakódovaného signálu, tvořená impulsy T+D, neovlivňuje, datové bity s úrovní H se do kompozice doplňují a to vždy tak, že jsou vůči impulsům T posunuty o π/2. Pro přehlednost jsou tyto impulsy ve stopě b) označeny šrafováním. Takto vytvořená zakódovaná kompozice vlastně představuje kmitočtově modulovaný signál se dvěma možnými dobami trvání jednotlivých intervalů mezi aktivními (sestupnými) hranami (2μs, 4μs). Tento signál se zavádí na záznamové obvody floppydisku. V jednotce je konečně dělen, například klopným obvodem D v poměru 1:2, čemuž odpovídá časový průběh magnetického záznamu na stopě c). Při čtení je možno původní kódovací kompozici T+D rekonstruovat. Pro názornost uvažujme její odvození pomocí obyčejného monostabilního obvodu s $T_{M1} < 2\mu s$, reagujícího na obě hrany čteného signálu, stopa d). Obdobně, pomocí monostabilního obvodu s $T_{M2} \cong 3$ μs lze demonstrovat oddělení impulsního sledu systémového taktu, stopa e), kterého lze využít pro obvody datového separátoru. V praxi se ovšem pro obnovu taktu nepoužívají monostabilní obvody, ale smyčka fázového závěsu PLL.

Princip *metody MFM*, umožňující vůči FM dvojnásobné zvětšení hustoty záznamu, je na obr. 36. Vychází se z poznání, že u metody FM tvoří podstatnou složku záznamu, zvyšující požadavky na šířku přenášeného kmitočtového spektra a tím i přesnost diskové mechaniky, impulsy systémového taktu *T*. Byla nalezena metoda, která činí užití těchto impulsů právě v kritických oblastech záznamu nadbytečným. Důsledkem jejího uplatnění je zmíněné zdvojnásobení hustoty záznamu bez zvětšení počtu magnetizačních změn. Stejně jako u FM zůstávají datové bity D = H aktivní součástí vytvářené kompozice. Impulsy taktu *T* se však nyní do kompozice zařazují pouze tehdy, když jim bezprostředně nepředchází ani je nenásleduje právě aktivní impuls D = H. Impulsy se tedy ve výsledné kompozici vyskytují pouze na hranách sousedních bitů D = L. Obnova taktu na straně čtení je nyní možná pouze s využitím závěsu PLL, zvětšují se i požadavky na přesnost mechaniky a čtecích zesilovačů jednotek. Pro větší bezpečnost čtených dat se na straně jejich zápisu používá princip tzv. prekompenzace.

Již z tohoto stručného přehledu fyzické struktury a organizace záznamu dat na diskové jednotky je patrna jejich značná složitost. Pro řízení jednotek se dnes již výlučně používají specializované procesory, monolitické řadiče diskových pamětí, viz např. klasický obvod 8272. Řadiče umožňují řídit na základě instrukcí postup akcí a kontrol řady funkcí přes vystavování hlavičky, záznam a čtení sektoru až například po formátování diskety. Přenos dat mezi operační pamětí a diskovou jednotkou vzhledem k dosažení velké přenosové rychlosti standardně zajišťuje další programovatelný obvod, řadič přímého přístupu k paměti DMA, například obvod 8257.

Zájemce o podrobnější informace k problematice diskových pamětí musíme odkázat na specializovanou literaturu. Pro naši další potřebu nám dosavadní informace stačí. Je však třeba uvědomit si, že dosud uvažované dělení diskového záznamu na stopy a sektory souvisí s fyzickou organizací diskové paměti (rotační pohyb, mechanismus vystavování hlavičky . .). Z hlediska programátora počítače, vybaveného diskovým operačním systémem, nejsou znalosti o okamžité pozici uložení dat na disku v podstatě vůbec zajímavé, právě naopak. Programátor potřebuje s těmito daty pracovat jako s vnějšími soubory a jako takové je také vytvářet, vyhledávat, zpracovávat i rušit. Ostatní informace jsou pro něj zbytečné a navíc mimořádně zatěžující, pokud podle nich má práci s vnějšími daty skutečně sám řídit. Právě efektivní organizaci a správu souborů na diskových pamětech přebírá za programátora i jiné uživatele operační systém, zajišťující i vzájemnou transformaci mezi logickou organizací souborů v operační paměti a fyzickou na diskových jednotkách.

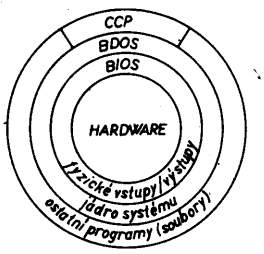
Operační systém CP/M

Okolnosti vývoje způsobily, že jako první standard diskově orientovaného operačního systému se ujal Control Program for Microcomputers autora G. Kindalla. Jeho koncepce je natolik geniální i jednoduchá zároveň, že se výrazně promítá i v nových operačních systémech současné doby, pro 16 a 32bitové mikropočítače. Zvláště zajímavé a vysoce účinné bylo rozdělení systému na dvě části, první zcela nezávislou a jednoznačně definovanou a druhou, modifikovatelnou podle konkrétní potřeby konfigurace technických prostředků implementace (počítače a periférií). Touto cestou byl systém CP/M koncipován jako velmi univerzální.

V původní verzi je CP/M určen pro 8bitové mikropočítače s CPU 8080 a Z80 jako systém, který z principu není schopen paralelního zpracování úloh. V současné době se CP/M v našich amatérských podmínkách s oblibou užívá prostřednictvím disků RAM.

CP/M je systém, umožňující uživateli operativně pracovat se soubory. V souvislosti s diskusí diskových pamětí jsme si jako základní prvky jejich organizace definovali stopy a sektory. CP/M ve svém jádru pochopitelně pracuje s jinou, logickou organizací dat. V ní jsou hlavními prvky soubory a jejich věty. Naše další úvahy velmi zpřehlední předpoklad, ostatně platný v základní verzi CP/M, že se soubory skládají z vět pevné délky (128 byte), shodné s rozměrem sektoru diskety (128 byte). Pak lze mezi základní fyzickou (sektor) a logickou (věta) přístupovou položku souboru položit rovnítko. To však už nelze jednoznačně předpokládat mezi pořadím vět v souboru a sektorů tohoto souboru na disku. Je také dobře hned zpočátku vědět, že CP/M umožňuje pracovat nejen se sekvenčním, ale také s přímým přístupem k jednotlivým větám souboru.

Přehled o vrstvené výstavbě struktury CP/M podává dnes již klasický obr. 37. Vlastní jádro tvoří prostřední vrstva BDOS (Basic Disc Operating System), řešená tak, že je zcela nezávislá na konkrétní konfiguraci technických prostředků počítače (užité periférie, jejich skladba a typy . .). BDOS zajišťuje inicializaci systému a logickou organizaci všech zařízení I/O včetně systému souborů a jejich adresářů.

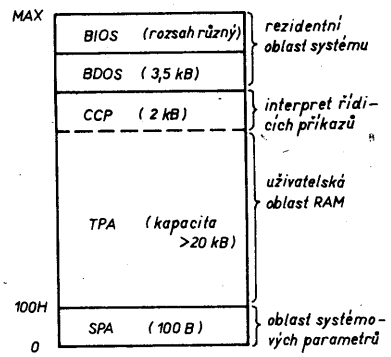


Obr. 37. Znázornění struktury operačního systému CP/M

Nejnižší vrstvou systému, znázorněnou na obr. 37, je BIOS (Basic Input/Output System), jímž je zajištěna vazba systému na konkrétní technické prostředky a tím i nezávislost a univerzálnost BDOS. Základní strukturu BIOS si uživatel přizpůsobuje podle svých aktuálních potřeb. Ze strany hardware, na které BIOS bezprostředně navazuje, lze tuto vrstvu vidět jako blok ovládačů (driverů) jednotlivých zařízení I/O a diskových jednotek.

Třetí, nejvyšší vrstva CCP (Control Command Processor) představuje interpret řídicích příkazů (interních a tranzientních), umožňující uživateli dialog se systémem prostřednictvím terminálu. Do stejné vrstvy, nad jádro systému, lze řadit i další systémové a uživatelské programy.

Umístění CP/M v operační paměti, která musí být umístěna od nulové adresy, je obr. 38. Nejnižších O až FF H bytů paměti RAM



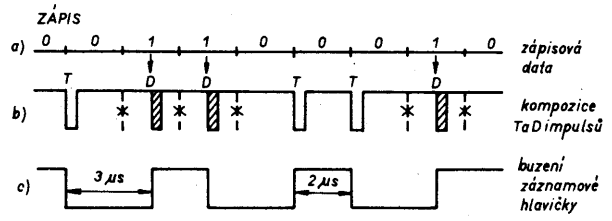
Obr. 38. Umístění jednotlivých částí CP/M v operační paměti

zabírá oblast systémových parametrů SPA, do které mj. patří i buffer (128 byte) přístupu k disku a tzv. blok řízení přístupu FCB (File Control Block), který systém používá k řízení přístupu k souborům na disku. Nad touto oblastí se nachází volitelně velká sekce RAM pro umisťování služebních i uživatelských programů. Sekce začíná adresou 100 H, která je současně startovacím bodem všech programů, využívaných pod CP/M. Část této sekce, označované jako TPA, zabírá také interpret CCP. Konečně nejvyšší souvislou sekci dostupné oblasti operační paměti zaujímá rezidentní prostor obou spolupracujících, do určité míry však i nezávislých modulů BDOS a BIOS.

Nyní je třeba pochopit využití a organizaci záznamového prostoru diskety pod systémem CP/M. Na každé disketě je rezervována oblast pro uložení systému, i když třeba není využita. Zbývajícího obsahu diskety se využívá pro:

adresář souborů,
vlastní soubory.

Celý tento prostor je z hlediska přidělování systému dělen na souvislé, tzv. alokační bloky sektorů. Smysl alokačních bloků vyply-



Obr. 36. Kódování záznamu metodou MFM (čárkovaně blokované taktovací impulsy)

ne v dalším. Paměťový prostor se tedy souborům přiděluje po blocích konstantní délky, kterou lze určit v BIOS. Vlastní přístup k souboru, ať už sekvenční nebo přímý, se však uskutečňuje po větách (sektorech).

Aby systém mohl práci se soubory (vytváření, vyhledávání, čtení, aktualizaci, rušení . .) organizovat se sekvenčním i přímým přístupem k jejich větám (sektorům), musí nějakým, z hlediska časové režie efektivním způsobem evidovat jejich aktuální stav. Současně musí zajišťovat transformaci mezi logickým (věty) a fyzickým (sektory) umístěním jejich položek.

Základním prostředkem pro evidenci souborů ve vnější paměti je adresář, nedílná složka obsahu každé diskety. Jedna položka adresáře detailně postihuje potřebné systémové informace o jednom souboru (nebo jeho části). V případě prázdného souboru je příslušná položka adresáře volná. Položka adresáře eviduje mj. jméno souboru a jeho typ, počet vět a čísla alokačních bloků, přidělených tomuto souboru. Alokační bloky na disku systém postupně čísluje a přiděluje počínaje od nuly. Adresář spravuje vrstva BDOS.

Popis obsahu diskety adresářem se často přirovnává k popisu hypotetického „logického disku", na kterém by na jedné stopě byly za sebou seřazeny všechny alokační bloky s jim přidělenými větami souborů.

Skutečné fyzické umístění záznamu na disku vyplývá z jeho rozložení do jednotlivých fyzických stop podle formátu diskety, popř. počtu sektorů/stopu. Při známém umístění prvního alokačního bloku se k přepočtu používá tabulka.

Mezi logickou a fyzickou strukturou je však třeba vidět jeden rozdíl, uvažujeme-li z hlediska souboru. Logickou strukturu souboru chápeme jako spojitou sekvenci jeho vět. Fyzická struktura souboru na disku však může být z řady důvodů nespojitá, v podstatě rozložená po celém disku. Kdyby měl být adresář tvrdě organizován tak, aby postihoval příslušnost každé věty ke konkrétnímu souboru a současně i její umístění, byl by neúnosně rozsáhlý. Právě proto užívá CP/M pro umisťování a evidenci souborů v adresářích systém alokačních bloků, nejčastěji o velikosti 1 nebo 2 kB. To přináší ostatně i jiné výhody, především zkracuje časovou režii při sekvenčním zpracování souboru, protože všechna data (věty) jsou v rozmezí každého bloku umisťována spojitě. Jeden nedostatek sdružování vět do větších bloků je však zřejmý na první pohled a stojí za zapamatování – při konstantní délce bloku (např. 1 kB) obsazuje i libovolně krátký soubor (např. s délkou jedné věty) vždy kapacitu celého bloku. Nevyužitá kapacita diskety v rámci bloku je z hlediska informačního obsahu ztracena. Tento jev se stává jedním z hlavních aspektů u systémů pro paralelní zpracování úloh (multitasking), kde je příčinou tzv. fragmentace paměti.

Uspořádání vět v logické struktuře souboru zajišťuje BDOS, spravující adresář. Využitím příslušných služeb BIOS, např. výběru stopy, sektoru, adresy DMA nebo čtení a zápisu sektoru a transformačních tabulek zajišťuje BDOS i odpovídající strukturu fyzickou. Takovou, jakou požaduje konkrétní typ diskové jednotky a jejího řadiče.

Zbývá vysvětlit, jak se vlastně adresář vytváří a využívá. To musí probíhat v součinnosti se systémem zabezpečení alokačních bloků a vět souboru na společné disketě. Při přístupu na disk CP/M využívá již dříve zmíněný blok řízení souborů FCB v oblasti systémových parametrů SPA. Většina položek bloku FCB má prakticky stejnou strukturu i význam, jako položky adresáře. Navíc blok FCB obsahuje položky pro určení pořadového čísla věty v rámci alokačního bloku při sekvenčním, nebo v rámci celého souboru při přímém přístupu. Pro jakýkoli odkaz na soubor systém využívá bloku FCB, jehož prostřednictvím se soubor identifikuje a adresují jeho položky.

Při aktivaci diskety se v systému na základě obsahu jejího adresáře vytvoří tzv. bitová mapa, evidující všechny využité alokační bloky, tedy systémový stav využití diskety. S každým otevřením souboru se do bloku FCB přenese jeho adresářová položka a zde, v oblasti SPA, se při všech operacích s tímto souborem trvale aktualizuje. Po uzavření souboru se konečný obsah FCB vrací zpět na disk jako aktuální položka adresáře. Bitová mapa je při práci na příslušné disketě trvale aktualizována, protože je v ní zohledňováno každé přidělení i zrušení alokačního bloku.

Shrneme-li dosavadní úvahy, vidíme, že vytváření uživatelských programů pod systémem CP/M je podporováno především jeho službami pro řízení I/O periferních a diskových zařízení. Na nejvyšší, systémové úrovni tomu odpovídají služby BDOS, mimo rámec CP/M však lze využívat i přímých služeb BIOS. Standardní CP/M nabízí celkem 40 služeb BDOS pro inicializaci systému, vstupy a výstupy logických, znakově orientovaných zařízení a souborů. Všechny tyto služby jsou dostupné voláním společného vstupu (CALL 5) v oblasti SPA s předáním čísla požadované služby a případných parametrů. Modul BIOS je programově zpřístupňován skokovým vektorem, umožňujícím výběr ze 17 služeb, z nichž každá je tvořena příslušným podprogramem, modifikujícím nebo rozšiřujícím kostru základního modulu.

CP/M patří k relativně jednoduchým, ale efektivním operačním systémům s minimálními nároky na podporu ze strany hardware počítačového systému. Toho bylo dosaženo jednoznačnou orientací na práci v monoprogramovém režimu. Samozřejmě existují aplikační požadavky, které CP/M splnit nemůže. K typickým patří úlohy, vyžadující radikální zvětšení propustnosti a zrychlení odezvy systému či multiprogramového nebo multiuživatelského využití počítače. Této problematiky si ještě všimneme později při úvahách o mikroprocesorech a operačních systémech vyšších generací.

Programování v asembleru

V AR-B č. 5/89 [1] jsme se zhruba seznámili s instrukčním souborem jednoduchého mikroprocesoru (Intel 8080) prostřednictvím jeho asemblerové mnemoniky, tvořící vždy základ příslušného jazyka symbolických adres JSA, tedy asembleru (assembly language). Výhoda mnemonického, textového popisu jednotlivých instrukcí je evidentní – relativně snadná pochopitelnost, zapamatovatelnost, přehlednost a rozlišitelnost významu jednotlivých instrukcí, zápisu jejich operačních kódů a operandů. Ostatně, bez mnemonického značení instrukcí by pro množství dnes existujících mikroprocesorů již na přímé úrovni strojového kódu nebylo vůbec možné pracovat pro naprostou ztrátu přehledu.

Mezi vyššími jazyky, jako je Pascal, a jazyky, pracujícími s asemblerovou mnemonikou, je ovšem velká mezera. Zde nacházíme jiné jazyky, jako například C-jazyk, PL/M a jiné, které však musíme ponechat stranou našeho současného zájmu, i když mají mnohé přednosti a jsou velmi praktické. Nicméně, stále jsou a budou situace, v nichž je asembler mimořádně vhodný nebo i nezbytný. To platí zvláště pro realizaci programů pracujících s periferiemi, vyžadujících rychlou odezvu nebo optimální využití kapacity operační paměti (typicky jednočipové mikropočítače). Asembler umožňuje „ušít" programové vybavení přesně na míru konkrétní aplikace.

Programování v asembleru je náročné. Vedle detailních softwarových znalostí vyžaduje i značný přehled o hardware vyvíjeného systému. Programátor si musí sám vytvářet potřebné datové a příkazové typy a struktury, k čemuž má k dispozici pouze instrukční soubor konkrétního procesoru a podporu dostupného křížového programového vybavení hostitelského nebo vývojového systému. Nemůže, tak jako ve vyšším jazyce, spoléhat na výkonnou typovou kontrolu.

Práce v asembleru má mnohá specifika. Patří k nim mimo jiné práce s jednotlivými (zvláště příznakovými) bity, která je základem pro vytváření složitějších příkazů, rozhodovacích funkcí či algoritmů. Začátečník naráží především na dva odlišné, ve skutečnosti však do sebe zapadající problémy:
– přesné chápání významu a činnosti značného počtu jednotlivých instrukcí, zejména uplatnění adresovací metody a závislosti nebo naopak působení instrukce na stav indikátorů příznakového registru,
– správné a efektivní užívání jednotlivých instrukcí při tvorbě potřebných datových a příkazových struktur programu.

Obojí se společně promítá v obtížné orientaci při studiu publikovaných programových výpisů a pokusech o vlastní aktivní práci. Právě zde dochází plného uplatnění byť povrchní znalost vyššího programovacího jazyka.

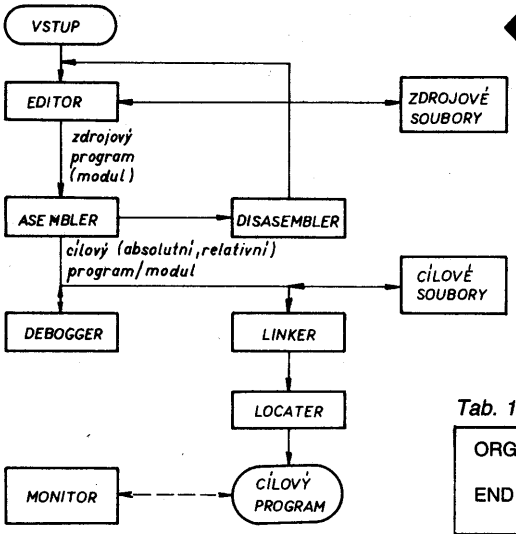
Jazyk symbolických adres

Mezi tvorbou programu ve vyšším jazyce a asembleru je i přes odlišnost obou úrovní nadále zásadní podobnost. Program musí být napsán ve zdrojovém, textovém tvaru (editor), přeložen do strojového kódu (asembler), odladěn (debugger, monitor), případně propojeny jeho moduly (linker) a umístěn do konkrétní aplikace (loader, programmer EPROM), obr 39. K tomu se opět využívá prostředků a pomocného programového vybavení operačního systému hostitelského počítače nebo speciálního vývojového systému, který většinou disponuje i dalšími efektivními prostředky pro vývoj a ladění programu (například emulátor).

Aby mohl být napsán asemblerový program, srozumitelný jak jeho tvůrci a dalším spolupracovníkům nebo pozdějším aplikátorům, tak také systému, na kterém má být vytvářen, musí být samozřejmě respektovány určité konvence, platné obecně pro různé procesory a tedy zavádějící do programování na úrovni JSA určité systematické, standardní postupy a přístupy. Nejdůležitější fakt vyplývá už z názvu JSA – jazyk symbolických adres – k popisu umístění jednotlivých instrukcí a jejich operandů v operační paměti neužívá konkrétních adres, ale pouze jejich symbolického označní (návěští, label) a to ještě pouze ve vybraných, významných místech programu.

Důvod zavedení adresové symboliky je zcela logický. Každá instrukce i data programu musí být umístěny na přesně definovaných místech operační paměti. V průběhu vytváření a zpracovávání programu však stále dochází k mnoha změnám (doplňování i vypouštění instrukcí . .), které by při absolutním adresování každé instrukce znamenaly nutnost trvale přepisovat ovlivněné adresy. To je ovšem pro systematickou práci nepřijatelné, každá identifikovaná chyba by měla za následek množství úmorné a zcela zbytečné práce. Stejné problémy by nastávaly při vytváření a umisťování přemístitelných, relokativních modulů.

Koncepce symbolických adres, využívající prostředků hostitelského nebo vývojového



Obr. 39. Znázornění postupu vytváření strojového programu prostřednictvím JSA a vývojového systému

Tab. 13. Formát zdrojového programu v JSA

Návěští	Instrukce		Komentář
	operační kód	operandy	
	ORG 0	0	; počáteční adresa ; modulu
START:	DI		; blokované přerušení
	LXI	SP, 1000H	; nastavení zásobníku
	JMP	INIC	; inicializace systému
	,		; pokračování programu
	END		; koncová pseudoin- ; strukce

systému, oprošťuje programátora od potřeby detailní správy a evidence paměťového prostoru na úrovni každé instrukce. To za něj zcela automaticky zajišťuje překladač, s nímž programátor komunikuje prostřednictvím několika pseudoinstrukcí, umožňujících překladači, linkeru atd. právě orientaci v systému symbolických i reálných adres. Průběh překladu, ladění, formu výpisu a jiné činnosti programátor řídí pomocí vhodných direktiv. Tvorba asemblerového programu tedy v zásadě probíhá velmi podobně jako ve vyšším jazyce. Hlavní rozdíl je v tom, že prostředky, kterými programátor nyní disponuje, jsou mnohem méně výkonné.

Praktickým důsledkem zavedení systému symbolického adresování do JSA je to, že v celém zdrojovém programu označujeme formou symbolického návěští pouze některé, mimořádně významné adresy. K takovým adresám patří například začátek programu, cílové adresy skoků a volání, vstupní body podprogramů apod. Zápis zdrojového textu JSA má určitý formát, zčásti vždy závislý na konkrétní implementaci a pochopitelně i typu procesoru.

Zdrojový tvar programu JSA

Můžeme přistoupit k orientačnímu popisu struktury zápisu a postupu zpracování programu v jazyce symbolických adres.

Program je vytvářen s využitím dvou odlišných typů instrukcí:

1. Běžných instrukcí, popsaných asemblerovou mnemonikou jednotlivých instrukcí CPU a příslušnými operandy.
2. Pseudoinstrukcí, které však vůbec nejsou instrukcemi CPU, ale slouží pro řízení činnosti překladače programátorem.

Pro první orientaci jsou v tab. 12 vypsány některé základní typy pseudoinstrukcí, s nimiž se v praxi u 8bitových systémů setkáváme nejčastěji. Zápis řádku, obsahujícího pseudoinstrukci, se běžně skládá ze tří položek:

1 – z návěstí nebo symbolického jména,
2 – ze zápisu vlastní pseudoinstrukce,
3 – z operandu pseudoinstrukce.

Pole 1 a 3 jsou ve výjimečných případech nepovinná. Tak jako každý jiný příkazový řádek, může být i řádek, obsahující pseudoinstrukci, doplněn polem komentáře. Příklad:

Label	Pseudoinstrukce	Operand	Komentář
MASKA	EQU	OF H	;maska horní bitové čtveřice

Tab. 12. Vybrané pseudoinstrukce jazyka ASM 80

ORG	Úvodní pseudoinstrukce. Nastavuje počitadlo adres překladače na hodnotu operandu.
END	Závěrečná pseudoinstrukce, označují konec překládaného zdrojového programu.
EQU	Pseudoinstrukce EQU přiřazuje symbolickému jménu, zapsanému v poli návěští (bez dvojtečky) hodnotu dat nebo výrazu z pole operandu.
DB	Pseudoinstrukce DB přiřazuje hodnotu z pole operandů 8bitovému paměťovému místu, identifikovanému návěštím. Jeho adresa je při překladu určena stavem počitadla adres překladače.
DW	Pseudoinstrukce DW přiřazuje aktuální lokaci (16 bitů), určené návěštím a stavem počitadla adres, hodnotu z pole operandů (nižší byte na první, vyšší na druhou adresu). Operandy pseudoinstrukcí DB a DW mohou být jednoduchá data, výraz nebo datový řetěz, případně řetězec operandů.
DS	Tato pseudoinstrukce rezervuje v paměťovém prostoru místo, počínaje návěštím (aktuálním stavem počitadla adres) s počtem bytů, uvedeným v poli operandu. Počitadlo adres se proto zvyšuje o tuto hodnotu.
ASEG	Tato pseudoinstrukce definuje při překladu absolutní, nepřemístitelný modul.
CSEG	Pseudoinstrukce CSEG (Code Segment) přiřazuje při překladu následujícímu bloku atribut programového modulu.
DSEG	Touto pseudoinstrukcí (Data Segment) je definován datový modul. Vytváří se tak, že do něj umisťujeme proměnné a alokujeme jej vždy v paměti RAM.
EXTRN	Pseudoinstrukce EXTRN umožňuje deklarovat symbolická jména, užitá v poli operandů jako externí, tj. skutečně umístěná v jiném, externím modulu.
PUBLIC	Pseudoinstrukce PUBLIC deklaruje návěští v modulu jako globální, tedy sdílená a přístupná jiným modulům. Prostřednictvím pseudoinstrukcí EXTRN a PUBLIC je možno vytvářet tabulku adres a dat, sdílených jednotlivými moduly. Ta je nezbytná pro linkování přemístitelných programů.

Každý program nebo programový modul se označuje symbolickým jménem, vázaným na pseudoinstrukci NAME, hlavní program se ukončuje pseudoinstrukcí END a operandem, odpovídajícím symbolické startovací adrese. Není-li operand uveden, je jako startovací implicitně nastavena adresa nulová.

Pseudoinstrukce EQU až DS v tab. 12 slouží pro definiční a deklarační účely.

Pomocí pseudoinstrukcí ASEG až DSEG se vymezují segmenty absolutních nebo relokativních programů.

Pseudoinstrukce EXTRN a PUBLIC umožňují pro potřebu spojování programových modulů deklarovat významná symbolická návěští jako buď umístěná v jiném modulu (EXTRN), nebo naopak jinému modulu dostupná (PUBLIC). Ostatní návěští v modulu mají i nadále lokální charakter, jsou tedy při linkování z ostatních modulů nedostupná. Tak je zajišťována jednoznačná struktura a hierarchie vazeb mezi jednotlivými moduly programu.

Ukázka formátu zdrojového tvaru takového modulu je znázorněna v tab. 13. Modul je uvozen pseudoinstrukcí ORG adr, nastavující počáteční hodnotu počitadla adres překladače a je ukončen pseudoinstrukcí END. V rozmezí obou těchto pseudoinstrukcí se zapisuje program, tvořený zdrojovými řádky. Každý řádek má opět k dispozici pole návěští, instrukcí či pseudoinstrukcí, operandů a komentáře. Pole návěští a komentářů jsou nepovinná, jejich využití závisí na potřebě, syntaxi (návěští) nebo vůli programátora (komentář). V určitých případech se neužívá ani pole operandů.

Pole návěští především umožňuje zápis symbolické adresy (ukončované dvojtečkou) daného řádku. Návěští musí začínat písmenem, nesmí být delší než 5 znaků a nesmí kolidovat s mnemonikou CPU a pseudoinstrukcí.

Pole instrukcí a pseudoinstrukcí slouží pro zápis příslušných mnemonických kódů. Operandy mohou být v dalším poli zadávány buď jako přímé (v kódu dekadickém, binárním, hexa nebo ASCII), nebo jako algebraický výraz a dále prostřednictvím registrového a nepřímého adresování, symbolického jméno nebo návěští.

Komentářové pole je nepovinné a také nemá na vlastní překlad žádný vliv. Uvádí se středníkem.

Vidíme, že symbolické adresy se ve zdrojovém programu vyskytují jak v poli návěští, tak v poli operandů. V poli návěští, v němž mají význam označení skutečné cílové adresy, se pochopitelně může každá symbolická adresa objevit pouze jedenkrát, na jediném místě programu. Výskyt stejné symbolické adresy v poli operandů je neomezený.

Překlad z JSA do strojového kódu

Uvažujme nyní postup překladu zdrojového programu ve formě textového souboru, vytvořeného v editoru. Vlastní překlad zpravidla probíhá ve dvou, navzájem velmi podobných průchodech.

Po spuštění překladu je odstartován první průchod. V něm jsou postupně čteny jednotlivé řádky zdrojového textu a pomocí přiřazovací, konverzní tabulky jsou překládány všechny jednoznačně definované objekty. Na počátku našeho příkladu, tab. 13, je pseudoinstrukcí ORG nastavena počáteční adresa počitadla adres překladače, přiřazená první insktrukci. U relokativních modulů se však používá relativní adresování, vztažené vždy k nulové počáteční adrese. Postupným překladem řádků zdrojového textu jsou přeloženy operační kódy a přímé nebo implicitně definované (registrové) operandy. Současně se všem návěštím přiřazují konkrétní adresy tím, že překladač automaticky zvyšuje obsah svého počitadla adres a vytváří tabulku uživatelských návěští. Po detekci koncové pseudoinstrukce END je ukončen první průchod. Překladač se vrací zpět na začátek a začíná druhý průchod. Postup překladu je prakticky stejný až na to, že jsou již k dispozici všechny potřebné adresy symbolických návěští i ostatní operandy a proměnné, závislé na návěštích a pseudoinstrukcích. V přeloženém programu nebo modulu ovšem může být celá řada chyb a neuspokojených odkazů (včetně externích).

Na konci překladu jsou proto vypisována chybová hlášení, včetně tabulek uživatelských návěští. Označují se jako křížové reference. Program se opravuje ve zdrojovém tvaru a překládá tak dlouho, až není hlášena žádná chyba. Výstup překladače na displej nebo tiskárnu má většinou volitený formát. Programátor si pomocí direktiv vybírá ten, který mu v dané chvíli vyhovuje nejvíce.

Plný formát, opomineme-li číslování řádků, může být postižen v šesti sloupcích, tab. 14. Z nich pouze první dva, tj. adresa a výpis strojového kódu se přímo týkají přeloženého programu. Ostatní sloupce poskytují možnost výpisu jednotlivých položek zdrojového textu jako reference.

Tab. 14. Výpis překladače z JSA do strojového kódu

Adresa	Kód	Návěští	Instrukce		Komentář
0000			ORG	0	
0000	F3	START:	DI		
0001	310010		LXI	SP, 1000H	
0004	C33800		JMP	INIC	
			;		
			;		
			END		

Překlad ze zdrojového asemblerového do cílového strojového kódu je ovšem teprve částí práce, spojené s naprogramováním konkrétní aplikace. Dále je třeba vyvíjený program odladit, ověřit sémantickou správnost jeho výstavby. K tomu je nutné program spustit a mít možnost sledovat a ovlivňovat průběh jeho činnosti. To opět vyžaduje speciální programové, někdy i technické prostředky.

K nejjednodušším prostředkům ladění programu patří monitory. Podle výkonnosti umožňují monitory ovládat a sledovat alespoň tyto funkce: start programu od zvolené adresy, jeho krokování po jednotlivých instrukcích nebo naopak zastavení na určené adrese po zvoleném počtu průchodů (zarážka), výpis a změny obsahu registrů včetně příznakového, výpis a změny obsahu paměťových míst.

Komfortnější formy monitorů se obvykle označují jako debuggery. Ty navíc umožňují i trasovat program, volitelně definovat zakázané paměťové oblasti, instrukce aj.

Velmi užitečným prostředkem je disasembler, tzn. zpětný překladač programu ze strojového do asemblerového tvaru.

Monitor i debugger mají jeden společný nedostatek, vyplývající z výlučně programového řešení. Je jím statický charakter ladění, nemožnost práce v reálném čase. To umožňuje emulátor, jímž bývají vybaveny speciální vývojové systémy. Prostřednictvím emulačního adapteru, který se osazuje namísto CPU do objímky vyvíjeného uživatelského systému, je možno jak plně, tak částečně emulovat jeho systémové, tedy technické (CPU, paměť..) i programové prostředky vývojovým systémem. Emulační adapter tedy nahrazuje mikroprocesor testovaného uživatelského systému a současně představuje interface vývojového systému, pod jehož kontrolou emulace probíhá. Kombinovaným využitím programových i technických prostředků vývojového a vyvíjeného systému je tak možno realizovat „komfortní debugger", pracující v režimu velmi blízkém režimu reálného času. Emulátor proto dovoluje užívat nejen všechny dosud uvažované funkce, usnadňující ladění vyvíjené aplikace s tím, že se přibližujeme reálným podmínkám, ale i mnohé jiné, spočívající v možnosti spolupráce či sdílení technických prostředků obou systémů. K nejvýznamnějším patří možnost předělovat vybrané paměťové sekce vývojového systému testované aplikaci (memory mapping).

Při ladění, testování a diagnostice vyvíjeného nebo opravovaného mikroprocesorového systému se užívají i speciální technické prostředky, z nichž nejvýznamnější je logický analyzátor, umožňující pohodlnou a přehlednou časovou nebo stavovou analýzu poměrně rozsáhlého úseku činnosti systému (počet zobrazených stop a stavů) před i po výskytu (pretrigger, posttrigger) požadované nebo předpokládané události. Užití analyzátoru je mimořádně cenné zvláště při diagnostice odezvy systému na jednorázové nebo pseudonáhodné akce (přerušení, komunikace . .). Jiným velmi užitečným prostředkem je při ladění programů simulátor EPROM. Jak u analyzátorů, tak simulátorů je zpravidla k dispozici disasembler, tedy možnost zpětného překladu strojového kódu zpět do asemblerového tvaru.

Práce v asembleru

Práci na úrovni asembleru (algoritmy, rutiny) jsme původně zamýšleli věnovat podstatnou část tohoto čísla. Mezitím však vyšla podle našeho názoru výborná Zajíčkova knížka Bity do bytu, kde je vlastně vše, čím jsme se chtěli zabývat a tak bychom pouze opakovali již hotovou a zdařilou práci. Proto jsme se rozhodli věnovat více prostoru Pascalu.

V této kapitole se tedy omezujeme na přehledový popis klíčových úvodních faktů s tím, že na vhodných místech odkazujeme na běžně dostupnou literaturu. Všímáme si základních paralel a odlišností mezi Pascalem a JSA.

JSA není a přirozeně nemůže být strukturovaný jazyk. Musí však umožňovat vytváření obdobných datových i příkazových typů a struktur. Jedinými prostředky k tomu jsou symbolické instrukce a pseudoinstrukce. S nimi je třeba zacházet v prostoru symbolických adres tak, aby vytvářené programy pokud možno odpovídaly zásadám strukturovaného programování.

Datové typy

Základními datovými typy, s nimiž pracují 8bitové procesory, jsou byte a word. Každý datový typ, dále užitý jako konstanta či proměnná, musí být v JSA nejprve definován či deklarován pomocí pseudoinstrukcí EQU a DB. Jazyk symbolických adres umožňuje přímou práci i s jinými typy dat (Boolean, dekadické, hexadecimální, ASCII . .) či datových struktur.

Konstanty mohou být definovány již zmíněnými dvěma typy pseudoinstrukcí, které pak řídí činnost překladače odlišným způsobem:

1. EQU chápe v poli návěští uvedené symbolické jméno (výjimečně neoddělované dvojtečkou) jako označení konstanty, a přiřadí mu hodnotu, uvedenou v poli operandů. Podle explicitně uvedeného datového typu se tak mimo prostor datové operační paměti vytvoří definovaná hodnota přímého operandu, která již nemůže být dále měněna, ale lze jí využívat v programu.

2. Pseudoinstrukce DB (resp. DW) přiřazují hodnotu ve formátu byte (popř. word), uvedenou v poli operandů, místu s uvedeným symbolickým návěštím. Obsah takto definovaného a deklarovaného paměťového místa (lokace) může být dále modifikován využitím instrukcí s různým adresovým přístupem, protože je umístěn na symbolicky označené pozici v datové operační paměti.

Příklad pro porovnání:

definice

```
VSTUP:   DB   96 H; pseudoinstrukce DB přiřadí lokací formátu byte
                  ; se symbol. adresou VSTUP
                  ; hodnotu 96 H
VYST     EQU 2E74 H; symbolickému jménu
                  ; VYST je přiřazena
                  ; hodnota přímého operandu
                  ; 2E74 H
```

program

```
         LDA VSTUP; přesun obsahu lokace
                  ; s adresou VSTUP
                  ; do akumulátoru
         STA VYST ; přesun obsahu akumulátoru (96 H) do
                  ; paměťové lokace s adresou, určenou definovaným obsahem symbolického
                  ; jména VYST (2E74 H:,,
                  ; tedy přímým operandem
```

Pseudoinstrukce DS rezervuje počet bytů v paměti, jejich obsah však nijak neovlivňuje. Užívá se pro definici vyšších datových struktur, například polí a tabulek. Příslušné návěští pak vždy odkazuje na prvý byte takové struktury.

Pomocí pseudoinstrukce EQU lze s odkazem na definici základních typů byte a word definovat (určit jejich rozměr) i ostatní v programu užité datové typy, např.:

```
BYTE    EQU   1
WORD    EQU   2
INT     EQU   BYTE
CHAR    EQU   BYTE
RETEZ   EQU   CHAR*7
```

Na tomto základě pak lze v JSA deklarovat konkrétní symbolické proměnné, tj. paměťové lokace s uvedenou symbolickou adresou a rozměrem, určeným definicí příslušného datového typu. Lze použít dva odlišné způsoby deklarace, buď s definovaným počátečním obsahem proměnné (pomocí pseudoinstrukcí DB nebo DW), nebo pouze s vymezením jejího paměťového prostoru (pseudoinstrukcí DS), např.:

```
STAV:    DB    7
POCET:   DS    INT
JMENO:   DS    RETEZ
```

Zde mimo jiné vidíme i základní rozdíl významu návěští v JSA vůči vyššímu programovacímu jazyku – neoznačuje přímo proměnnou, ale její místo (lokaci) v paměti.

Podrobněji se již datovými typy zabývat nebudeme, nejlepší cestou k jejich praktickému osvojení a užívání je studium programových výpisů.

Příkazové typy a struktury

Veškeré příkazy mohou být vytvářeny pouze vhodným řazením jednotlivých instrukcí tak, aby bylo využito jejich funkčních vlastností. Některé jednoduché, například přiřazovací a přesunové příkazy mohou být interpretovány přímo jednotlivými instrukcemi. MVI A, 3F H přiřadí akumulátoru obsah přímého operandu, MOV C, A přesune obsah akumulátoru do registru C, instrukce MOV M, A přesune obsah ACC na paměťovou lokaci, adresovanou registrovým párem HL atd. Pro dobrou orientaci v programu je třeba znát adresovací metodu, užívanou tou kterou instrukcí, jak vyplývá již z těchto primitivních ukázek.

Složitější příkazy mohou mít, vzhledem k možnosti využití různých typů instrukcí, registrů, příznaků, ale i algoritmů a práce v paměťovém prostoru, většinou několik alternativních řešení. Ta se však mohou lišit v rozsáhlosti, přehlednosti, ale i bezpečnosti konstrukce, vytvářené sledem instrukcí. Mohou se lišit i rychlostí provádění.

K vytváření těchto příkazů JSA může a musí užívat především testování stavu indikátorů (CY, S, P, Z) registru PSW jako odezvy na výsledek prováděných aritmetických a logických operací. Na jejich základě lze do programu zavádět podmíněná větvení, která jsou základní podmínkou vytvoření složitějšího příkazu nebo příkazové struktury. Právě zde je skryta síla i nebezpečí asembleru. Síla v tom, že asembler umožňuje optimálně realizovat příkazové struktury (minimalizace potřebné paměťové kapacity, rychlost), nebezpečí pak v ničím neomezené a nekontrolovatelné možnosti vytvářet nestrukturované příkazy.

Pro orientaci alespoň několik příkladů možné realizace strukturových příkazů v JSA.

Na obr. 40 je ekvivalent příkazu větvení programu IF p THEN P1 ELSE P2. Platnost podmínky p = true je zde vyhodnocována instrukcí podmíněného skoku JNC. Pro přehlednosti je na pravé straně obrázku i instrukce CMP C, porovnávající binárně obsah registrů ACC a C. Je-li (ACC) < (C), je nastaven indikátor CY = 1, realizuje se příkazová část P1, v ostatních případech část P2. Jediný a tedy strukturovaný výstup z příkazu zajišťuje skok na návěští N2. Poznamenejme, že instrukce CMP C není typickou součástí příkazu, příznak CY může být ovlivňován mnoha jinými způsoby. Funkci vlastního příkazu větvení lze modifikovat jak úpravou jeho vnitřní struktury (viz např. obr. 3), tak způsobem testu a volbou příznakového indikátoru (C/NC, Z/NZ . .).

Test příznakového indikátoru je i základem tvorby programových cyklů. Na obr. 41 je možná varinta realizace cyklu WHILE p DO P, na obr. 42 naopak cyklu REPEAT P UNTIL p.

V mnoha případech se příkazy na úrovni JSA vytvářejí, ve srovnání s vyššími jazyky, ve zjednodušené formě. Jako příklad si uveďme realizaci pascalského cyklu FOR se zjednodušeným nastavením počitadla (implicitní podmínka v1 = 0). Na obr. 43b je konstrukce, která by vznikla při šablonovitém úsilí o transformaci vývojového diagramu do asemblerové formy. Vyžadovala by naznačenou korekci při vstupu do cyklu, aby byl vždy první test JZ = false, nezávisle na předaném obsahu PSW. Stačí si uvědomit, že v daném případě, při implicitní podmínce v1=0, je vstupní test relevantnosti nastavení počitadla zbytečný, aby vznikla mnohem jednodušší konstrukce cyklu, obr. 43c.

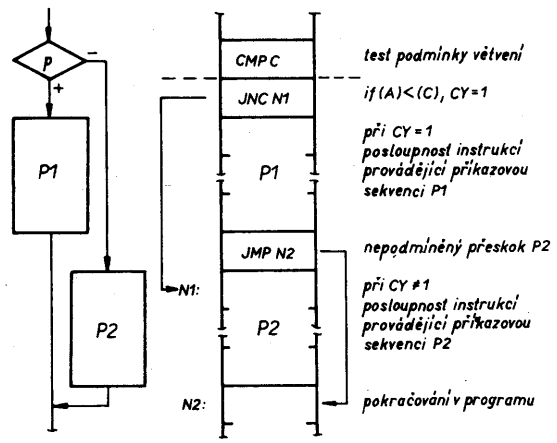
Všimněme si, že příkazové struktury využívají, ale také ovlivňují (a tím pro další použití znehodnocují) obsahy některých registrů (zde C a PSW). Mají tedy druhotné účinky, na což je třeba v programu dávat dobrý pozor. Na druhé straně můžeme vidět, že i při programování v asembleru se zhusta užívá mnoha víceméně prefabrikovaných konstrukcí, modifikovaných podle okamžité potřeby.

Makroinstrukce

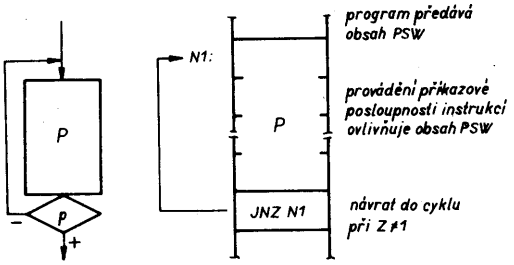
Vyšší typy diskově orientovaných překladačů (makroasemblery) umožňují práci s tzv. makry, makroinstrukcemi. Kdekoli v programu se častěji opakující posloupnost instrukcí může programátor předem definovat jako makroinstrukci s formálními parametry.

Definice makroinstrukce se skládá z úvodní části, tvořené symbolickým jménem v poli návěští, označením pseudoinstrukcí MACRO, a formálními parametry, např. PODIL MACRO SUMA, POCET. Následuje vlastní tělo makroinstrukce, užívající potřebných instrukcí a formálních parametrů. Celá definice „makra" je uzavřena koncovou pseudoinstrukcí ENDM. Z definice vidíme určitou formální podobnost s procedurou, ve skutečnosti je mezi makroinstrukcemi a procedurami zásadní rozdíl. Makroinstrukce je tzv. otevřený podprogram, který lze ve zdrojovém textu kdykoli volat zápisem symbolického jména a uvedených skutečných parametrů vždy, je-li makroinstrukce zapotřebí.

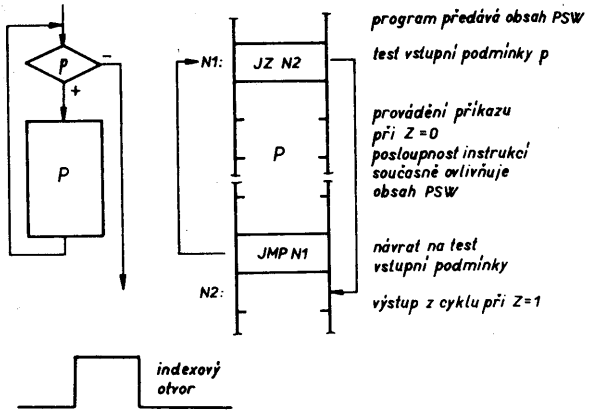
Makroasembler při překladu rozvíjí volanou makroinstrukci tak, že v jejím těle defino-

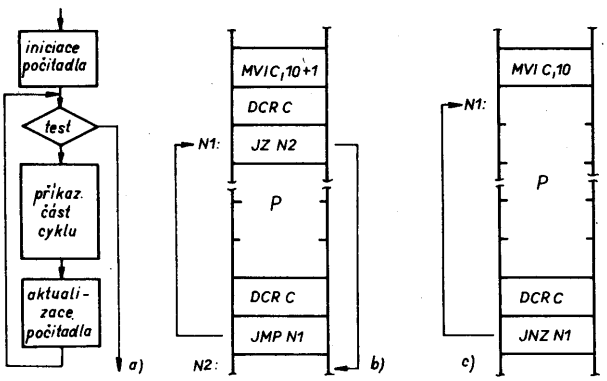


Obr. 40. Jedna z možností konstrukce příkazu IF p THEN P1 ELSE P2



Obr. 42. Příkazová struktura cyklu REPEAT s využitím testu indikátoru ZERO





Obr. 43. Příklady realizace pascalského cyklu FOR

Obr. 41. Příklad konstrukce cyklu WHILE p DO P s využitím testu indikátoru ZERO

vanou příkazovou posloupnost vždy znovu zapisuje ve strojovém kódu a formální parametry nahrazuje skutečnými, uvedenými v příslušném volání. Makroinstrukce tedy pouze zjednodušují práci programátora, ale nijak nešetří prostorem operační paměti.

Rutiny, podprogramy

Pro získání základního přehledu o instrukčním souboru a jeho užití v JSA je nejlépe přejít ke studiu standardních rutin. Nejprve těch jednodušších ([9], str. 211 až 233), později složitějších ([10], str. 123 až 161). Jejich znalost představuje v programování na úrovni JSA asi to, co znalost základních elektronických principů a schémat v hardwarové oblasti. Při takovém studiu, nejlépe podepřeném prací na konkrétním, i tom nejlevnějším a nejjednodušším počítači, si teprve člověk uvědomí veškeré funkce jednotlivých typů instrukcí, jejich veškeré účinky, souvislosti a návaznosti a tedy i možnosti optimálního využití nebo naopak dříve neviděná omezení.

K nejdůležitějším úlohám při studiu jakéhokoli programu nebo jeho úseku patří pochopení jeho struktury, řešící určitý algoritmus. Členění programové struktury na hlavní program, dílčí moduly a podprogramy umožňuje odděleně studovat jednotlivé programové bloky. Přitom je vždy nejdůležitější nejprve pochopit jejich podstatu, způsob začlenění do celého programu, vstupní a výstupní parametry včetně užitého způsobu jejich předávání. Zde doporučujeme podrobně prostudovat zejména [10], str. 113 až 122 a [11], str. 84 až 95 s typickými příklady.

Zvláštní kapitolu představuje styk mikropočítače s okolím, vyžadující určitou orientaci v práci s periferními obvody (I/O, čítače/časovače, přerušení . .). Zde je nejlepší školou studium komentovaných výpisů konkrétně zaměřených programů, jaké nacházíme například na zelených stránkách AR i jinde.

Orientace v programových výpisech

Jako příklad jsme vybrali program pro obsluhu a vyhodnocení činnosti převodníku A/D, uveřejněný v AR–A č. 5/89. Především proto, že využívá programových prostředků při řešení technické úlohy. I kdy se program asi bude zpočátku zdát příliš složitý, je to jen zdání. Je vytvořen skutečně systematicky, jak ukazuje „strukturogram" na obr. 44, vytvořený postupnou syntézou programového výpisu. Hlavní program tvoří smyčka, volající rutiny PROMER, ZOBRAZ a TIME2. Podobně PROMER, řídicí obsluhu měření a kalibrace, využívá podřízených modulů AUTEST a MERENI, ZOBRAZ disponuje rutinou binárně dekadické konverze BNDCZ atd. Postupným rozborem programové struktury podle vzájemných vazeb jednotlivých modulů a podprogramů (CALL až RET) tak můžeme vcelku pohodlně odvodit celou znázorněnou strukturu. Pak je již možno, s využitím komentářů, poměrně přehledně sledovat celý program a relativně nezávisle i jeho dílčí moduly včetně vzájemných návazností. Prostřednictvím „strukturogramu" si také můžeme učinit představu o možnostech případných úprav a změn nebo o využívání sdílených podprogramů (například rutiny SUBTRA a DMULTI jsou sdíleny programy AUTEST a MATIK). Při detailnějším rozboru si můžeme učinit i představu o výkonnosti a současně i o nebezpečí skoků mimo vlastní podprogram, v uvažovaném případě se jedná pouze o podmíněné skoky z MERENT a MATIK do rutiny chybových hlášení. Tyto skoky proto nejsou ve strukturogramu podchyceny.

Výstavba programu prozrazuje erudici jeho tvůrců. Mnohé úseky jsou zajímavé i tehdy, když se skládají byť jen z několika řádků. Příkladem může být rutina TIME, která po podrobném prostudování nebo praktickém ověření jednou provždy objasní práci se zásobníkem. Krokovat prováděním programu je vždy to nejlepší, co můžeme udělat v případě, že činnosti určitého programového úseku nerozumíme. Takovým úskalím budou pro začátečníka zřejmě rutiny BNDCZ a DMULTI. Je možná překvapivé, že právě ty, byť nejrozsáhlejší, nejsou komentovány vůbec, nebo jen sporadicky. Je to proto, že se jedná o modifikované standardní rutiny, porovnejte např. BNDC se stejnojmenným modulem z knihovny systému MRS.

Architektura a instrukční soubory moderních 8bitových mikroprocesorů a jednočipových mikropočítačů

V AR–B č. 5/89 jsme pro osvětlení základních principů, architektury, adresovacích metod nebo požadavků na skladbu instrukčního souboru použili jako názorný příklad relativně jednoduchý mikroprocesor I8080 s tím, že jeho zvládnutí je dobrým předpokladem snažší orientace i ve vývojově vyšších mikroprocesorových generacích. V návaznosti na dosud vytvořené názory a představy si nyní přehledově popíšeme architekturu a instrukční soubory mikroprocesoru Z-80 a jednočipových mikropočítačů řad 8048 a 8051, tedy relativně moderních a u nás běžně dostupných nebo perspektivních prvků.

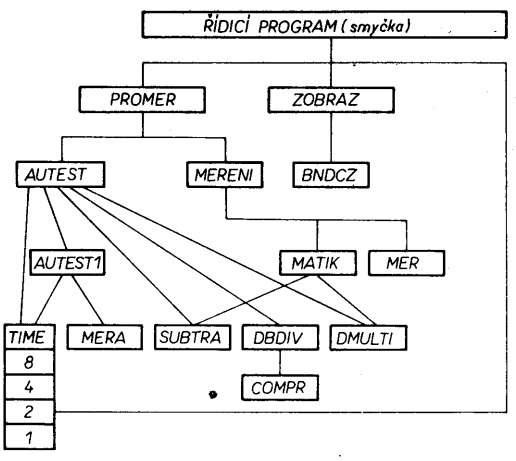
Mikroprocesor Zilog 80

Architektura mikroprocesoru Z-80, jehož blokové schéma je na obr. 45, je vůči I8080 podstatně komplexnější. Orientace v tomto stále populárním typu mikroprocesoru však bude snadná, protože představuje přímou inovaci 8080.

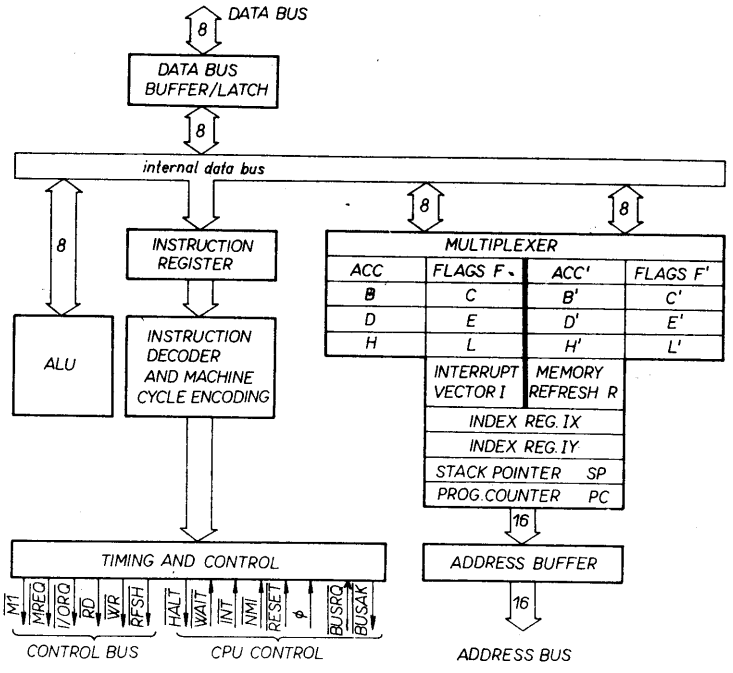
Z-80 užívá jako systémový takt snadno generovatelný jednofázový externí hodinový signál. Na čipu je již integrován celý systémový řadič. Další charakteristickou odlišností vůči I8080 je zdvojení všech zápisníkových registrů, rozdělených do dvou registrových bank. Programově vždy může být vybrána jedna z nich, což je výhodné v řadě praktických aplikací (jednoúrovňová přerušení, podprogramy . .), při kterých obvyklý úklid a znovuvyzvedávání obsahu ohrožených registrů nahradí programově řízená výměna registrových bank. Zdvojen je samozřejmě i příznakový registr F.

Podstatně praktičtěji je řešena i strategie přerušovacího systému. Z-80 je vybaven dvěma vstupy žádostí o přerušení, nemaskovatelným $\overline{NMI}$ s nejvyšší prioritou a běžným, programově maskovatelným vstupem $\overline{INT}$. Žádosti $\overline{NMI}$, kterou nelze programově zakázat, odpovídá technický skok na pevnou adresu 66 H, obsluha musí být ukončena speciální návratovou instrukcí RETN. Pokud se týká oblasti maskovatelných přerušení, mohou být programově aktivovány tři odlišné módy činnosti.

Mód 0 je v podstatě obdobou prioritního systému, užívaného již u I8080, včetně orientace na využití pevných adresových pozic instrukcí RST *n*. Vyžaduje proto výstavbu externího přerušovacího systému s příslušným řadičem. Pro programové povolení a zákaz přerušení se užívá instruk-



Obr. 44. Odvozená hierarchická struktura programu pro převodník A/D (AR A5/89)



Obr. 45. Blokové schéma mikroprocesoru Z-80

cí EI, DI, jako návratová instrukce RET.

Také pro mód 1 nacházíme obdobu jednoúrovňového přerušení, generovaného řadičem 8228 ve speciálním režimu, včetně zcela shodné technické adresy aktivace RST7, tj. 38 H. Jako návratová se opět užívá instrukce RET.

Nejzajímavější a zcela odlišný je mód 2, navazující na promyšlenou koncepci stavebnice podpůrných obvodů procesoru Z-80. Tyto obvody mají instalován rozvinutý decentralizovaný přerušovací systém, nahrazující v naprosté většině případů potřebu klasického externího řadiče přerušení. Přerušovací hierarchie se konfiguruje vhodným smyčkovým propojením speciálních vstupů IE IN/výstupů IE OUT periferních obvodů. Vytvořená smyčka pracuje jako sériový opakovač žádostí z jednotlivých vstupů se schopností blokovat žádosti s nižší prioritou, než je právě aktivovaná. Aktuální žádost je po zmíněné smyčce předávána na vstup $\overline{INT}$ mikroprocesoru. Zdroj žádosti pak předává na datovou sběrnici její identifikační byte, který spolu s podle potřeby programovatelným obsahem registru IR (Interrupt Register) určuje nepřímou adresu vektoru přerušení. Teprve na takto vytvořené adrese musí být uložena počáteční adresa obsluhy příslušného přerušení. Je jistě vidět mimořádná pružnost celého systému a široký rozsah adresovacích možností v celém paměťovém prostoru. Jako návratová instrukce slouží instrukce RETI. Programovou volbu jednotlivých módů přerušení umožňují instrukce IMO, IM1, IM2.

Dalším novým a zajímavým registrem je osvěžovací registr dynamických pamětí R (Memory Refresh). Je to vlastně 7bitový čítač, inkrementovaný v průběhu každého instrukčního cyklu. Jeho obsah se vysílá na nižším bytu adresové sběrnice v dobách T3, T4cyklu M1, kdy při aktivním řídicím bitu $\overline{RFSH}$ = L a žádosti o přístup k paměti $\overline{MREQ}$ = L je možno oživit řádek DRAM.

Zcela nová, typická již pro mikroprocesory vyšších generací, je implementace 16bitových indexregistrů IX, IY. Jejich smyslem je zavést systém ukazatelů do operační paměti, umožňující efektivní indexové adresování uspořádaných datových položek a struktur.

To jsou hlavní rozdíly v architekturách I8080 a Z-80. Další ovšem nacházíme v řadě periferních obvodů stavebnice Zilog, obsahující obvody PIO (Paralel I/O Interface), SIO (Serial I/O Interface), CTC (Counter/Timer Circuit) a DMA (Direct Memory Access).

Instrukční soubor Z-80 je přímou nadmnožinou souboru I8080, a to až do úrovně strojového kódu. Celá společná skupina instrukcí se tedy liší pouze asemblerovou mnemonikou. Několikanásobného rozšíření souboru Z-80, k němuž vedlo zavedení nových instrukcí, se vůči souboru I8080, který již prakticky vyčerpal kapacitu 8bitového pole operačních kódů, dosahuje doplněním kódů těchto instrukcí o jeden nebo dva byty, tzv. prefixy. Tvoří je v souboru 8080 nevyužité kódy CB, DD, ED a EF, záměrně naznačené už v tabulce na obr. 100 v AR–B č. 5/89. Skladba instrukčního souboru Z-80 je proto pestrá nejen co do obsahu různých typů instrukcí, ale i formátů jejich operačních kódů v rozsahu 1 až 4 byty.

U Z-80 se také vzhledem k I8080 využívá nových adresovacích metod, především relativního a indexového adresování.

Relativní adresování je velmi výhodná adresovací metoda, vztažená vůči aktuálnímu obsahu programového čítače PC. K relativnímu postihu adresy užívá tzv. posunutí (displacement), interpretované v 8bitovém doplňkovém kódu jako číslo se znaménkem. To umožňuje adresování v celkovém rozpětí −128 až 127 bytů vůči běžnému obsahu (PC)+2. To +2 proto, že relativní adresování užívají výhradně dvoubytové skokové instrukce.

Indexované adresování je vlastně obdobou relativního s tím, že přímá odchylka není vztažena k programovému čítači, ale k obsahu indexregistru.

V souvislosti s vlastním instrukčním souborem Z-80 je tedy v první řadě třeba upozornit na možnosti, vyplývající z uplatnění nově zavedených registrů I, R, IX, IY a obou bank registrů zápisníkových. Odlišné možnosti práce se systémem přerušení jsme již uvedli.

Velmi nápadný a efektivní je nový typ blokových instrukcí, orientovaný na opakované provádění typických, často užívaných operací. Setkáváme se s nimi v různých aplikacích.

Jednou z nich je skupina instrukcí vstupů/výstupů. Vedle klasických instrukcí IN A (port) a OUT (port), A pracujících s obsahem akumulátoru, a nových instrukcí IN reg, (C) a OUT (C), reg. umožňujících přenos mezi některým z 8bitových registrů a portem, adresovaným obsahem registru C, zde nacházíme rozsáhlou skupinu blokových instrukcí I/O. Patří k nim instrukce INI a OUTI (Input/Output Increment), IND a OUTD (Decrement), INIR a OTIR (Increment Repeat) a konečně INDR a OTDR (Decrement Repeat). Increment nebo decrement se přitom vztahuje k adresování položek přenášeného datového bloku. Práce s těmito instrukcemi podstatně celý přenos I/O zjednodušuje a zrychluje.

Další uplatnění nacházejí blokové instrukce při interních přenosech datových bloků, viz instrukce LDI/LDD a LDID/LDDR. Účelovou obdobu jejich využití představují i instrukce blokového porovnávání a vyhledávání CPI, CPD (Compare Increment/Decrement) a CPIR/CPDR (Repeat).

Všechny instrukce blokového typu probíhají podle určitého, velmi podobného schématu, s využitím obdobných registrů. Není proto obtížné je pochopit a s výhodou užívat.

Rozšířena byla i skupina aritmetických instrukcí. V 8bitovém rozsahu se jedná o možnost využít dekadické korekce DAA i na záporné výsledky operací s čísly BCD, v rozsahu 16bitovém pak o snazší práci s čísly o vícenásobné přesnosti.

Skupina skokových instrukcí byla obohacena především o zmíněné relativní adresování. Zajímavá je instrukce DJNZ, která představuje podmíněný skok s počtem opakování, určeným obsahem registru B, pracujícího jako automaticky dekrementovaný čítač.

Instrukce aritmetických a logických rotací, včetně záměny bitových čtveřic, podstatně usnadňují výstavbu aritmetických a logických algoritmů. Sem také patří mimořádně rozsáhlá i užitečná skupina instrukcí pro práci s jednotlivými bity, umožňující nastavení (SET), nulování (RES) a testování (BIT) jednotlivých bitů libovolného 8bitového zápisníkového registru nebo místa operační paměti.

Z-80 je a zůstává moderním, dobře koncipovaným 8bitovým procesorem, vytvářejícím se svými podpůrnými obvody stavebnici vysokých kvalit. U nás je stále základem širším vrstvám dostupných domácích počítačů. Podrobný popis jednotlivých instrukcí, včetně příkladu jejich využití a přehledně uspořádaného instrukčního souboru je v [11]. Z praktického hlediska je velmi užitečné mít přehled instrukčního souboru (mnemonika, kódy) k dispozici na jednom listu papíru, lze jej získat z Bajtku 12/88.

Pro fázi seznamování s asemblerem Z-80 na úrovni JSA, ale i pro praktickou práci je dostupný poměrně bohatý sortiment programového vybavení na domácí počítače, orientované na tento procesor (Spectrum, Sharp . .). Velmi vhodný je například již zmíněný MRS (Memory Resident System), obsahující v jednom „balíku" editor, asembler, knihovnu binárních modulů, debugger, linker a disassembler. Tento soubor umožňuje styl práce, blížící se klasickým postupům, obvyklým při užívání mnohem dokonalejších hostitelských počítačů nebo vývojových systémů.

Jednočipové mikropočítače

Klasickým stavebnicím a počítačům s 8bitovými mikroprocesory (jako je Z-80) vyrostla v posledních letech velmi silná konkurence. Z jedné strany ji představují nesrovnatelně výkonnější 16 a 32bitové protějšky, okupující postupně celou oblast osobních mikropočítačů a výkonných řídicích systémů.

Opačný pól, který se stále více prosazuje, představují jednočipové, převážně 8bitové mikropočítače (mikrokontrolery). Umožňují ekonomicky realizovat automaty pro nejrůznější aplikace v oblasti jak unikátní, tak kusové, sériové nebo i masové výroby.

Vznik jednočipových mikropočítačů zcela přirozeně vyplývá z dosažené úrovně technologie. Možnost soustředit na hromadně vyráběném čipu nejen celý mikroprocesor, ale i všechny prvky vhodně redukovaného mikroprocesorového systému, byla již dlouho příliš zřejmá a lákavá, aby nebyla prakticky realizována.

O tom, co jednočipové mikropočítače nabízejí z různých hledisek (cena, rozměry, příkon, doba vývoje zařízení, výrobní náklady . .) není třeba diskutovat. Jejich prostřednictvím se mohou úspěšně prosazovat i malé týmy s omezeným kapitálem a technologickým zázemím.

Mikropočítače řady 8048

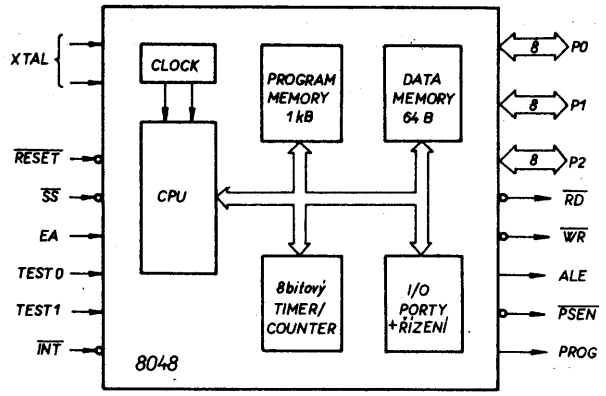
U nás jsou dosud relativně dostupné pouze mikropočítače řady 8048 (s rezidentní programovou pamětí ROM), případně jejich modifikace 8035 (bez rezidentní programové paměti) nebo 8748 (s pamětí EPROM). Tato řada opět představuje vhodný úvod do celé problematiky jednočipových mikropočítačů.

Jak vyplývá z orientačního blokového schématu (obr. 46), má jednočipový mikropočítač ve své struktuře integrovánu vedle kompletního procesoru i více či méně celou datovou a programovou operační paměť, stykové obvody I/O, alespoň jednoduchý přerušovací systém a interní čítač/časovač. Kapacita interních, rezidentních pamětí je samozřejmě vzhledem k technologickým možnostem, oboru předpokládaných aplikací a z ekonomických důvodů omezená. Jednočipové mikropočítače jsou však zpravidla řešeny tak, aby v určitém rozsahu mohly být velmi snadno rozšířeny jak o externí paměťový prostor, tak další stykové obvody I/O.

Jádro celého mikropočítače řady 8048 tvoří CPU, rezidentní paměťový systém a obvody I/O. Podrobnější analýza interní struktury CPU (viz [12], [13]) v určitém ohledu znovu připomíná koncepci jednotky 8080, včetně orientace na střadač a sadu zápisníkových registrů.

Pro jednočipové mikropočítače je charakteristické to, že z technologických, ale i jiných praktických důvodů nerespektují při správě adresového prostoru operační paměti koncepci von Neumannova typu. Operační paměť je organizována podle hardwarské koncepce jako rozdělená do dvou sekcí, datové a programové, s odlišnými přístupovými metodami.

Rezidentní paměť dat (RAM) s omezenou kapacitou 64 B obsahuje i dvě registrové



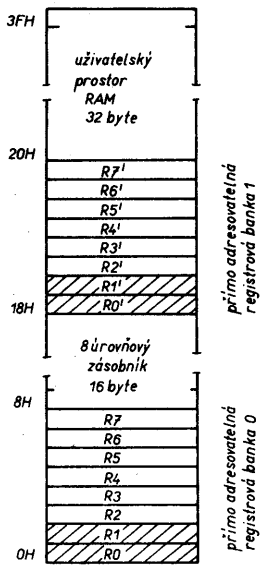
Obr. 46. Znázornění struktury jednočipového mikropočítače řady 8048

Legenda

XTAL – vstupy interního generátoru hodin, (krystal, obvod *LC* nebo externí signál)
RESET – vstup pro inicializaci procesoru
SS – vstup pro krokování programu (Single Step)
EA – vstup pro uvolnění/blokování přístupu k vnější programové paměti
TEST 0 – programově testovatelný vstup/(příp. výstup) tclock
TEST 1 – programově testovatelný vstup/příp. vstup interního čítače
INT – vstup žádosti o vnější přerušení
PO – obousměrný 8bitový statický port s výstupním latchem, při užití externích pamětí jako budič multiplexované systémové sběrnice BUS, řízené signály PSEN, ALE, RD, WR
P1 – 8bitový kvaziobousměrný port I/O
P2 – 8bitový kvaziobousměrný port I/O; při práci s vnější programovou pamětí jsou bity P20 až P23 nositeli stránkových adres, mohou být využity i pro připojení expanderu I/O 8243
RD – výstup strobující čtení z vnější datové paměti
WR – výstup, strobující zápis do vnější datové paměti
ALE – (Address Latch Enable), výstup řídící závěrnou hranou zápis nižší části adresy (ADR 0 až 7) do externího latche při přístupu k externí programové i datové paměti
PSEN – (Program Store Enable), výstup, aktivující čtení z externí programové paměti
PROG – výstupní signál k řízení expanderu 8243/ případně vstup programových impulsů pro interní paměť EPROM (8748)

banky RB0 (R0 až R7) RB1 (R0' až R7') a také osmiúrovňový zásobník, viz mapu na obr. 47. Přímo adresovatelné jsou v datovém prostoru pouze jednotlivé registry. Proto se jich přednostně užívá pro často zpřístupňované položky probíhajícího výpočtu. Ostatní pozice v prostoru datové RAM mohou být adresovány pouze nepřímo. Z tohoto hlediska mají zvláštní význam registry R0 a R1 (R0' a R1'), označené na obr. 47 šrafováním. Mohou být užity jako ukazatele do interní datové paměťové oblasti. Pro orientaci několik příkladů zápisu přímých dat a přesunů mezi střadačem a datovou pamětí:

```
MOV   R3, #5EH ; zápis přímého operandu
               ; do registru R3
MOV   A, R5    ; přesun z registru R5
               ; do střadače

MOV@  R1, A    ; přesun obsahu střadače
               ; na nepřímou interní
               ; datovou adresu, určenou
               ; obsahem registru R1.
```



Obr. 47. Mapa interní datové paměti

Obdobně se užívá ukazatelových registrů R0, R1 i pro adresování přístupu do vnější datové paměti. K tomu slouží speciální instrukce MOVX. Příklad:

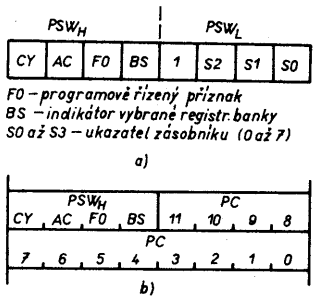
```
MOVX@  R1,A    ; přesun obsahu střadače
               ; na nepřímou adresu v
               ; externí datové paměti,
               ; určenou obsahem R1.
```

Druhá banka registrů RB1 může být užívána různými způsoby. Jedním z jich může být jednoduché rozšíření kapacity banky RB0. Velmi výhodné je její využití při obsluze přerušení – ukládání a výběr registrů při vyvolání a návratu pak odpadá, nahrazuje je jednoduchá záměna aktivity obou bank. Proto se také neužívá obvyklých instrukcí PUSH a POP, které by neúnosně zatěžovaly interní zásobník s malou kapacitou.

Osmiúrovňový zásobník užívá dvoubytový formát položky (obr. 48b). Vedle úplného obsahu programového čítače (12 bitů) se do zásobníku ukládají i čtyři indikátory příznakového registru PSW_H, obnovované při návratu z přerušení speciální instrukcí RETR.

Rezidentní paměť programu u 8048 (ROM) či 8748 (EPROM) má kapacitu 1 kB. Významné body v ní představují lokace 0,3 a 7, vyvolávané technicky aktivací vstupu RESET, externího INT nebo interního přerušení. Na tyto adresy (obr. 49) musí být umístěny skoky do příslušných obslužných programů. Z mapy ovšem vidíme, že programový čítač umožňuje adresovat prostor 4 kB. Ten je rozdělen na dvě banky MB0 a MB1 po 2 kB, volitelné opět programově. Prostor 4 kB může být externí pamětí pokryt buď zčásti, nebo úplně. Navíc nejnižší 1 kB může užívat buď rezidentní, nebo externí paměť. Volba se zajišťuje technicky, zapojením vstupní špičky EA (External Access) na logickou úroveň H nebo L.

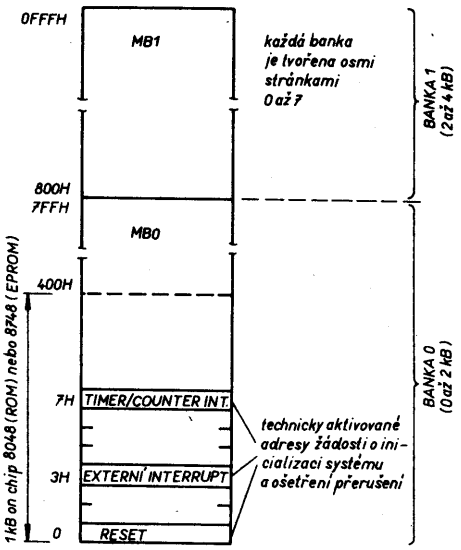
Adresování v prostoru programové paměti využívá stránkového mechanismu – obě banky 2 kB jsou rozděleny na osm stránek, každá stránka tedy má kapacitu 256 B. Pak pro adresování uvnitř stránky běžné paměti stačí 8 bitů, pro výběr jedné z osmi stránek v rámci banky 3 bity. Tomu odpovídá struktura a využití programového čítače PC (obr. 50). Nejvyšší bit rozlišuje ve výběru aktuální banky (MB0, MB1).



Obr. 48. Struktura a) stavového slova PSW, b) položky zásobníku

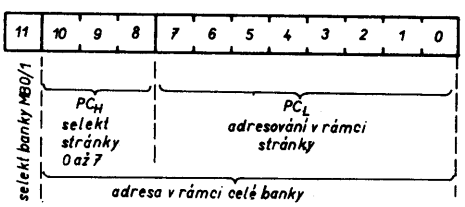
Adresovací metody, užívané instrukčním souborem při práci s programovou pamětí, se opírají především o časově i technicky efektivní adresování uvnitř běžné stránky. Pro čtení z této paměti se jako cílový registr užívá střadač, který současně slouží i jako registr, obsahující nepřímou relativní adresu, užívanou speciální instrukcí MOVP. Příklad:

```
MOVP A, @ A;  ; do střadače se přesune
              ; obsah buňky programové
              ; paměti, adresované v
              ; rámci stránky původním
              ; obsahem střadače (přene-
              ; seným do bitů 0 až 7
              ; čítače PC).
```

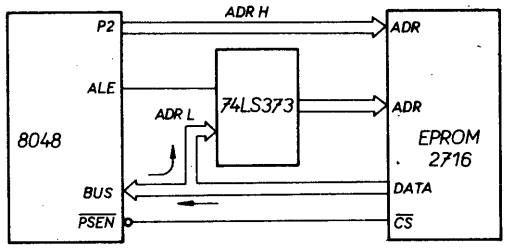


Obr. 49. Mapa programové paměti (podle typu mikrořadiče a jeho ošetření může být nejnižší 1 kB součástí buď interní, nebo externí paměti)

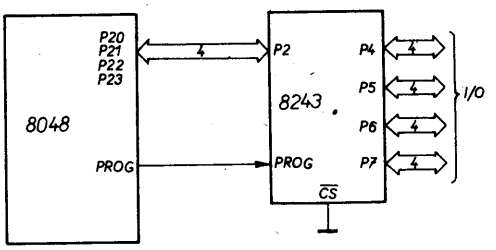
Zvláštní postavení má instrukce MOVP3 A, @ A, která obdobným způsobem vždy naplní střadač obsahem nepřímo adresované paměťové buňky ze stránky 3 (tj. adresového prostoru 300 H až 3FF H), zcela nezávisle na tom, jakou stránku, ale i banku právě adresoval programový čítač. Pro výjimeč-



Obr. 50. Využívání struktury 12bitového čítače instrukcí PC



Obr. 51. Připojení externí programové paměti 2 kB



Obr. 54. Rozšíření vstupů/výstupů 8048 expandérem 8243. 4bitové porty P4 až P7 lze použít pro obousměrný přenos mezi portem a nižší bitovou čtveřicí střadače a logické operace na portu

nost této instrukce se do stránky 3 umisťují konstanty a reference, užívané v rámci celého programu.

Zkrácené 8bitové adresování v rámci stránky programu používají i instrukce podmíněných skoků a instrukce DJNZ.

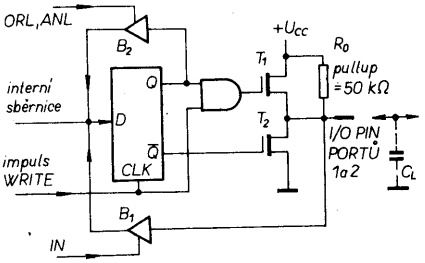
Plné adresování v rámci celé aktuální banky MB0 nebo MB1 užívají pouze instrukce přímého skoku JMP, volání CALL a návratů RET, RETR.

Na obr. 51 je příklad použití externí programové paměti EPROM 2 kB. Programový čítač PC může adresovat až 4 kB programové paměti, protože má 12bitový rozsah. Adresování opět využívá stránkového mechanismu. Vyšší část adresy v rámci banky umožňuje definovat až 16 stránek (A8 až A11), z nichž každá obsahuje 256 bytů, adresovaných nižším bytem (A0 až A7) programového čítače. Pro zachycení této nižší části PCL, přenášené po sběrnici DB multiplexně s daty, se používá latch, strobovaný signálem ALE (Address Latch Enable). Pro přenos zbývajících tří bitů adresy PCH (A8 až A10), potřebných pro adresování 2 kB, slouží část portu P2. Signál $\overline{PSEN}$ aktivuje výstup dat z takto adresované, čtené lokace externí programové paměti.

Vcelku jednoduše lze rozšířit i datovou paměť o externích 256 B, tedy opět o rámec jedné stránky. Jednoduché a praktické řešení je na obr. 52. Využívá toho, že statická paměť RAM 256 B je součástí programovatelného obvodu 8155, který již má instalován interní adresový latch a multiplexovaný bus ADR/DATA. Navíc doplňuje systém i o tři I/O porty a programovatelný 14bitový counter/timer.

Dotkli jsme se již využití I/O portů. Jsou celkem tři, P0 (BUS), P1 a P2, všechny 8bitové. Port P0 má dvě charakteristické aplikace. Může pracovat jako obousměrný budič multiplexované systémové směrnice (ADR 0 až 7/DATA 0 až 7) viz obr. 51, 52, nebo jako obousměrný, ale statický (vstupní nebo výstupní) třístavový port.

Porty P1 a P2 užívají technologicky odlišného řešení, to je typické tím, že jejich vstup/výstup nemůže být uveden do čistého 3. stavu. Princip, který pro jeden bit takového kvaziobousměrného kanálu znázorňuje obr. 53, znamená určité omezení při práci s těmito porty.



Obr. 53. Kvaziobousměrný bitový kanál portů P1 a P2

Data, případně adresy (P20 až P23 – adresování stránky), přenášená z interní sběrnice na výstup portu, se zachycují výstupním latchem, tvořeným obvodem D. Zápis se uskutečňuje čelní hranou strobovacího impulsu. Výstup $\overline{Q}$ ovládá tranzistor T2 dvojčinného koncového stupně MOS přímo, tranzistor T1 je ovládán logickým součinem úrovní výstupu Q a časově omezeného strobovacího impulsu.

Při zápisu úrovně H přecházejí výstupy Q→H, $\overline{Q}$→L, tranzistor T2 je trvale zavřen, T1 spíná, ale pouze po dobu trvání strobovacího impulsu. To stačí, aby na výstupním portu byl vytvořen ostrý náběh, potřebný pro rychlé vybuzení sběrnice s kapacitní zátěží (spoje, přívody, vstupy hradel). Po doznění strobingu oba tranzistory T1, T2 nevedou, úroveň H na výstupním portu staticky udržuje upínací odpor, R_0, tvořený ve skutečnosti strukturou MOS s charakterem zdroje konstantního proudu.

Při zápisu výstupní úrovně L přechází latch do stavu Q=L, $\overline{Q}$ = H. Tranzistor T1 je tedy zavřen, T2 sepnut. V tomto stavu již obvod setrvává až do nového zápisu výstupní úrovně H.

Z uvedeného vyplývá, že při případném čtení ze vstupního portu, které se uskutečňuje přes řízený budič B1, by na bitu, který předtím byl nastaven výstupní instrukcí do stavu L, nebylo zajištěno čtení správné hodnoty. To proto, že klopný obvod D i nadále udržuje tranzistor T2 v sepnutém stavu. Aby byla operace čtení z portů P1 a P2 korektní, musí jí vždy předcházet přípravný zápis výstupní úrovně H.

Druhý řízený budič B2, vázaný přímo na výstup Q klopného obvodu, se užívá pro realizaci logických operací mezi daty, zapsanými na výstup portu a přímým operandem.

Pro často potřebné rozšíření počtu vstupů/výstupů se využívá různých externích obvodů, příkladem může být již obr. 52. Nejčastěji se u řady 8048 užívá speciálního programovatelného obvodu, expanderu 8243, vázaného na mikrořadič prostřednictvím bitů P20 až P23 portu P2 a řídicího signálu PROG, viz obr. 54.

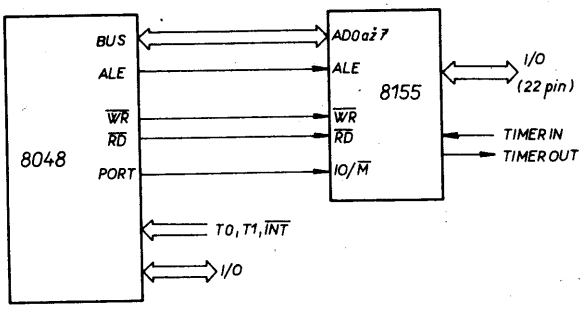
Další rozšíření aplikačních možností umožňuje interní 8bitový programově přednastavitelný a testovatelný čítač, který může být programovou cestou aktivován do dvou různých pracovních režimů (timer nebo counter).

Instrukcí STRT T dochází k vynulování obsahu a připojení interního děliče (prescaleru :32), buzeného systémovým taktem, na vstup čítače, který je tedy aktivován jako timer, inkrementovaný v intervalu každých 32 strojových cyklů, obr. 55. Volitelného zpoždění $\Delta T = (1 \text{ až } 256)32t_{clock}$ se dosahuje programovým přednastavením čítače a detekcí jeho přeplnění (indikátor TF) nebo využitím interního přerušení INT. Násobků těchto intervalů lze snadno dosáhnout vyhodnocováním většího počtu přetečení čítače programovou cestou. Timer se užívá i jako generátor přenosové rychlosti (Baude) programově realizovaného UART pro sériový přenos.

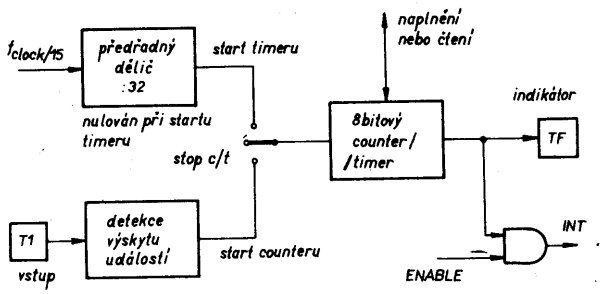
Ke generování intervalů kratších než $\Delta T = 32t_{clock}$ lze využít vyššího hodinového kmitočtu (odvozeného např. od ALE), zavedeného na externí vstup TEST1. V tom případě ovšem musí být čítač aktivován do druhého pracovního módu, jako čítač vnějších událostí, instrukcí STRT CNT. Pak je vstup čítače programově přepnut do opačné polohy, obr. 55. Sestupná hrana externího signálu, zaváděného na vstup TEST1, vždy způsobí inkrement obsahu čítače.

Stav interního čítače může být kdykoli, i v průběhu čítání, programově testován. V obou režimech lze čítač zastavit instrukcí STOP TCNT.

Přerušovací systém nabízí možnost využít tří různých variant. První představuje vstup $\overline{RESET}$, který může být aktivován staticky při počáteční nebo dynamicky při každé



Obr. 52. Rozšíření datové paměti obvodem 8155 (256 B RAM, obvody I/O a 14bitový časovač-timer)



Obr. 55. Základní schéma činnosti programově modifikovatelného interního čítače událostí/časovače

další inicializaci. Impuls na tomto vstupu vždy vyvolá nemaskovatelné nulování obsahů PC a SP, výběr nulových registrových a programových bank, porty P1 a P2 se nastaví do korektního vstupního režimu, zastaví se interní čítač (avšak beze změny obsahu) a nuluje jeho indikátor TF. Je zakázáno programové přerušení (interní, externí). Vstup RESET se často užívá externími obvody Watch-dog, zabraňujícími trvalému zacyklení či jinému zhroucení programu v důsledku reakce na nekorektní pracovní podmínky.

Maskovatelná přerušení jsou dvě, interní a externí. Interní přerušení vzniká přetečením nastaveného obsahu timeru/counteru v případě, že bylo programově povoleno. Pak je technicky vyvolána programová adresa 7, kde musí být uložen skok na obsluhu tohoto přerušení. Současně se automaticky zablokují případné další žádosti. Externí přerušení, stimulované žádostí na vstupu INT, aktivuje obdobným způsobem adresu 3, obr. 49. Pro ukončení obsluhy obou typů maskovatelných přerušení se užívá speciální návratové instrukce RETR (Return with PSW Restore), která kromě návratové adresy obnoví i vyšší část (indikátory) stavového slova PSW, obr. 48. Přerušovací systém je jednoúrovňový, při současném výskytu obou žádostí však má přednost obsluhy přerušení externí.

Řada 8048 představuje pro mnoho jednodušších aplikací ideální technické prostředky. Z cenových důvodů je výhodná i pro amatérské konstrukce. Podrobný technický popis lze nalézt v [12], [13], popis instrukčního souboru i s ukázkami jednoduchých programů v [14].

Jednočipové mikropočítače řady 8051

Pro náročnější aplikace má být i u nás v dohledné době dostupná řada 8051 (s interní pamětí ROM), představující s modifikacemi 8031 (bez rezidentní programové paměti) a 8751 (s pamětí EPROM) v oblasti jednočipových mikropočítačů současný světový standard. Při studiu těchto mikropočítačů je opět možno vycházet ze znalosti řady 8048, znovu se jedná o její inovaci. Orientační blokové schéma mikropočítače řady 8051 je na obr. 56.

Architektura, výkonnost i instrukční soubor těchto mikropočítačů jsou vůči řadě 8048 výrazně posíleny a rozšířeny se zřetelem na užití v náročnějších aplikacích, v nichž mohou nahrazovat i klasické mikroprocesorové stavebnice. Zvlášť výrazným rozdílem je vůči řadě 8048 především implementace výkonného booleovského procesoru.

Rezidentní programová paměť má zvětšenou kapacitu 4 kB, kapacita datové paměti je 128 B. V externím adresovém prostoru však mohou být obě paměti rozšířeny až na 64 kB.

Na vývodech pouzdra jsou k dizpozici čtyři 8bitové porty, které mohou být používány různými způsoby. při adresování externích pamětí však portům P0 a P2 přísluší stejná úloha jako u 8048 s tím rozdílem, že může být adresován prostor až 64 kB. Obdobnou úlohu mají i řidicí a strobovací signály ALE, PSEN, RD a WR.

Ve struktuře řady 8051 jsou dále implementovány dva 16bitové čítače/časovače, dvouúrovňový prioritní přerušovací systém a plně duplexní sériový kanál UART, který může pracovat ve čtyřech odlišných módech.

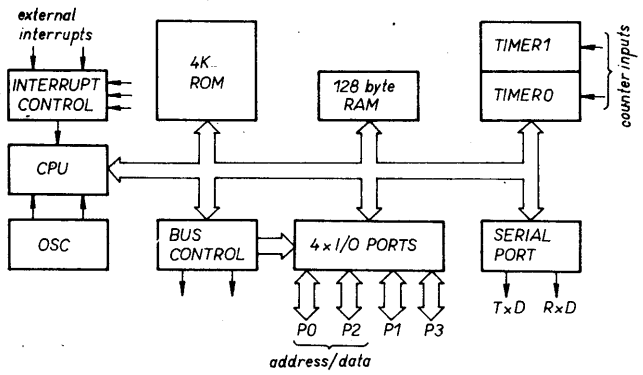
Na obr. 57 je v jeho spodní části znázorněna struktura rezidentní datové paměti o kapacitě 128 B, pokrývající adresový rozsah 0 až 7F H, označený po levé straně mapy. Tento prostor lze dělit na tři odlišné části.

Nejnižší z takto uvažovaných sekcí (0 až 1F H) přísluší čtyřem bankám zápisníkových registrů (R0 až R7), využitelných i pro zásobník. Banky se označují čísly 0, 1, 2, 3. Po resetu je vždy aktivní banka 0 a SP je inicializován na adresu 7, odkud se pak postupně inkrementuje, tedy přechází do banky 1 atd. Pokud má být využito předností přepínaných registrových bank, je nutno zásobník inicializovat mimo banky, do vyhrazeného prostoru ve vyšší oblasti RAM. Oblast registrů je adresovatelná přímo i nepřímo, ale vždy v běžném, celobytovém formátu.

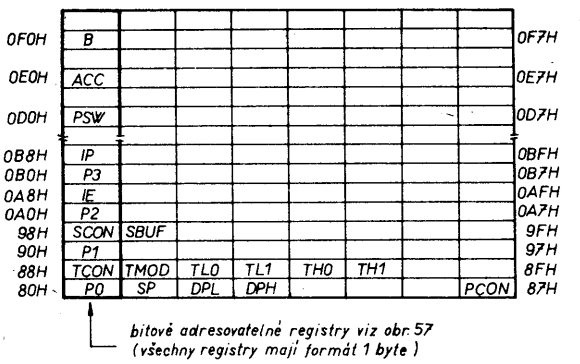
S implementací booleovského procesoru úzce souvisí organizace přístupu bloku rezidentní datové paměti, který se skládá ze 16 bytů na adresách 20 až 2F H. Celý tento prostor může být zpřístupňován jak běžným, tak také bitovým adresováním (bitové adresy jsou uvedeny uvnitř každého bytu, na pozicích odpovídajících bitů). To znamená, že každý jednotlivý bit v této oblasti může být nezávisle na zbývajících nulován, nastaven negován, mohou s ním být prováděny logické operace nebo může být využit jako podmínkový. Prostor bitových adres, využívaný booleovským procesorem, je v této oblasti spojitý, pokrývá rozsah 0 až 7F H. Bitové adresy jsou využívány speciálními instrukcemi a nemohou být zaměňovány s adresami běžnými.

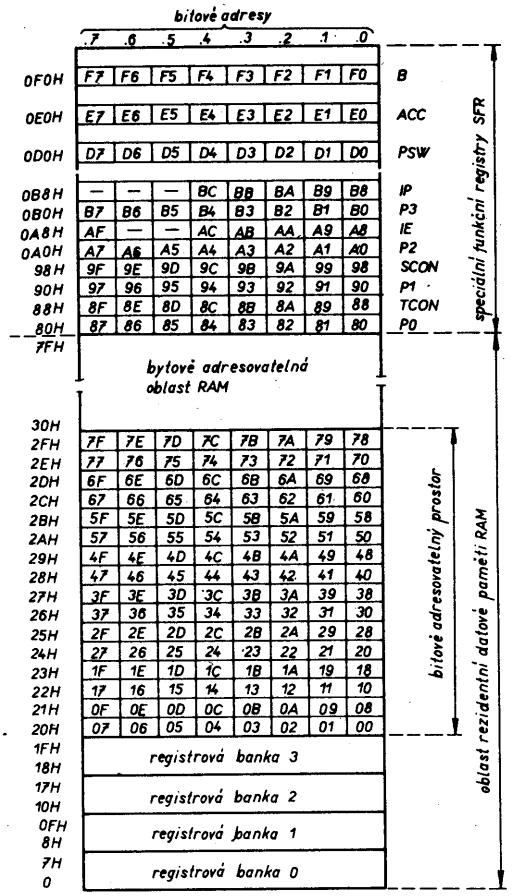
Konečně třetí sekce rezidentní RAM v oblasti běžných adres 30 až 7F H slouží opět pouze jako klasická paměť RAM s možností vyhrazení prostoru pro zásobník. Bitově adresovatelná však již tato oblast není.

Těsně nad rezidentní datovou pamětí se nachází dalších 128 bytů RAM, v nichž jsou opět převážně umístěny registry, tentokrát však hardwarové. Jednak takové, jaké se vyskytují i u běžných procesorů (ACC, PSW . .), ale také registry, které jsou obdobou speciálních řídicích či stavových registrů, jež u klasických mikroprocesorových stavebnic bývají rozptýleny v pomocných, programovatelných obvodech.



Obr. 56. Struktura jednočipového mikropočítače 8051





Obr. 57. Přehledová mapa rezidentní datové paměti (0 až 7F H) a oblasti hardwarových registrů (80 až FF H)

Obr. 58. U všech registrů v levém sloupci tabulky lze přímo adresovat jejich jednotlivé bity, ostatní registry jsou adresovatelné pouze jako byte

Celá zmíněná oblast tzv. speciálních funkčních registrů SFR je znázorněna na obr. 58. Některé z těchto registrů mohou být opět adresovány i bitově. Jsou to ty, které se na obr. 58 nacházejí v levém, orámovaném sloupci. Ostatní jsou adresovatelné pouze jako celý registr, mnemonická označení a bytové adresy vyplývají z obr. 58.

Mechanismus tvorby bitových adres registrů SFR je jednoduchý. Bitová adresa sleduje v nejvyšších 5 bitech adresu bytovou, spodní 3 bity určují adresu konkrétního bitu. Bitové adres relevantních, bitově adresovatelných registrů SFR, jsou přímo uvedeny uvnitř bitového pole na obr. 57.

Některé lokace v oblasti SFR jsou, jak vyplývá z obr. 58, buď prázdné, využívané interně, nebo vyhrazené pro další inovaci.

Práce se speciálními registry SFR je tedy obdobou práce s registry CPU a programovatelných obvodů klasické mikroprocesorové sestavy.

Asemblerové překladače umožňují specifikovat bitové adresy různými účelovými způsoby. Například číselným zápisem adresy podle obr. 57, nebo pomocí selektorů (0 až 7), označujících pozici bitu v rámci bytu, nebo konečně prostřednictvím asemblerové mnemoniky konkrétního bitu příslušného registru SFR. Jako příklad několik významově shodných definic bitově orientované instrukce v prostoru SFR:

```
SETB PSW.5 ; nastavení 5. bitu (příznaku 0,
           ; který má mnemonické označení
           ; F0) registru PSW s užitím bitové
           ; ho selektoru,
SETB F0    ; nastavení téhož bitu, adresova-
           ; ného jeho mnemonikou,
SETB OD5 H ; opět nastavení stejného bitu,
           ; adresovaného přímou bitovou
           ; adresou, viz obr. 57, 58.
```

Bitové adresy se přirozeně týkají pouze vybraných instrukcí booleovského procesoru. Význam, funkce, mnemoniku a podrobný popis registrů SFR a jejich bitů viz [13], [15], [16].

Programátorskou, technickou i časovou optimalizaci tvorby a provádění programu usnadňují i rozvinuté adresovací metody, uplatňující vedle relativního i indexové adresování. Jako příklad sledujme instrukci přímého, nepodmíněného skoku, která může být, podle odpovídajícího adresového rozpětí, interpretována třemi různými způsoby s formálně shodným, z hlediska provádění však zcela odlišným výsledkem či průběhem:

SJMP (Short Jump) je instrukce relativního skoku v rozpětí 8bitového doplňku vůči běžnému stavu programového čítače, tedy na adresu (PC) + 2 + relativní adresa! Instrukce je dvoubytová se všemi výhodami relativního adresování;

AJMP (Absolute Jump) opět uskutečňuje nepodmíněný skok, tentokrát v rozsahu bloku 2 kB. Skok již není relativní, ale instrukce zůstává dvoubytová. V prvém bytu jsou specifikovány tři bity A8 až A10, ve druhém spodní byte adresy A0 až A7; konečně

LJMP (Long Jump) představuje tříbytovou instrukci s úplnou 16bitovou adresou, umožňující definovat cílovou adresu skoku v plném rozsahu 64 kB.

Indexové adresování umožňuje instalace 16bitového ukazatele DPTR (DPH, DPL).

Značně neobvyklé jsou také instalované instrukce pro násobení, MUL, a dělení, DIV, celých 8bitových operandů (obsahů střadače A a registru B) bez znaménka.

Dva interní čítače/časovače mohou pracovat ve čtyřech různých módech. Jednoho časovače lze využít pro nastavení přenosové rychlosti sériového kanálu.

Sériový kanál umožňuje komunikaci s okolím prostřednictvím duplexního, softwarově programovatelného, ale technicky realizovaného UART.

Systém přerušení je schopen akceptovat pět zdrojů žádostí o přerušení. Má dva externí vstupy $\overline{INT0}$ a $\overline{INT1}$, dva interní vstupy lze odvodit od čítačů/časovačů, páté je přerušení od sériového kanálu. Každý zdroj přerušení může být programově přiřazen do jedné ze dvou prioritních úrovní nastavením nebo nulováním příslušného bitu v registru IP (interrupt Priority).

Vývoj jednočipových mikropočítačů je teprve na svém počátku. Rozvíjena a inovována je i řada 8051. Jedním z nejrozvinutějších prvků této řady je mikrokontrolér 80535 v pouzdře se 68 vývody. Obsahuje 8/10 bitový převodník A/D s programově přepínanými osmi vstupy, tři timery/countery, pět 8bitových portů, interní watch-dog aj. Přerušovací systém je schopen akceptovat 12 různých interních a externích žádostí o přerušení.

Poněkud odlišný trend vývoje naznačuje například kontrolér 80186 fy Intel, který v jediném pouzdru obsahuje prakticky ekvivalent 16bitového mikroprocesoru 8086 nejen včetně hodinového generátoru, ale i řadiče přerušení, timerů, kanálů DMA a obvodové logiky. Po doplnění vnějšími pamětmi, obvody I/O a případně řadičem diskové jednotky pak představuje velmi dobrý základ výstavby výkonného jednodeskového systému pro náročné průmyslové aplikace.

Vývoj nových generací mikroprocesorů a operačních systémů

Poslední rozsáhlejší kapitolu věnujeme přehledovému popisu postupného vývoje mikroprocesorů a mikropočítačů k současným standardům vyšších generací. K tomu jsme zvolili poněkud neobvyklý přístup. Diskusi vývoje 16 a 32bitových mikroprocesorů vážeme na paralelně probíhající vývoj operačních systémů. To proto, že je mezi nimi úzká souvislost, jedna oblast ovlivňuje druhou. Myslíme si, že právě takový, zevšeobecňující pohled umožňuje snáze pochopit podstatu celé, již poměrně složité problematiky i čtenářům bez předběžné přípravy. Celý vývoj můžeme sledovat například na postupném rozšiřování mikroprocesorových řad Intel (8080, 8088, 8086, 80286, 80386 . .) i úspěšných operačních systémů (CP/M, MS DOS, OS/2, UNIX . .), kde nacházíme nejednu zajímavou souvislost.

Nejprve si všimneme několika obecných technických inovací, souvisejících s úsilím o zvýšení rychlosti procesoru jako prvního kroku ke zvětšení výkonnosti celého počítače.

Úzké hrdlo mikropočítače – systémová sběrnice

Technologický pokrok posledních let umožnil podstatně zvýšit takt CPU, tedy rychlost zpracování jednotlivých instrukcí, a zároveň i rozšířit tok dat, tj. bitovou šířku datové sběrnice (8, 16, 32 bitů). Praktický dopad obou těchto jistě významných inovací na propustnost systému by však bez dalších opatření zdaleka nebyl tak markantní, jak bychom snad mohli očekávat.

Důvodů je několik. První vyplývá přímo z koncepce vazby CPU s relativně pomalou operační pamětí prostřednictvím systémové sběrnice, zajišťující nezbytný obousměrný přenos instrukcí a dat. Jakkoli je tedy systémová sběrnice pro činnost mikropočítače nezbytná, je i po zběžné analýze patrné, že po většinu doby trvání instrukčního cyklu je z hlediska CPU prakticky pasívní, nevyužívaná. Aby již zmíněná opatření měla smysl, musí být i tato slabina nějak eliminována. U vyšších generací mikroprocesorů se k tomu využívá především dvou strategií, předvýběru instrukcí (prefetch/pipeline) a skryté mezipaměti (cache).

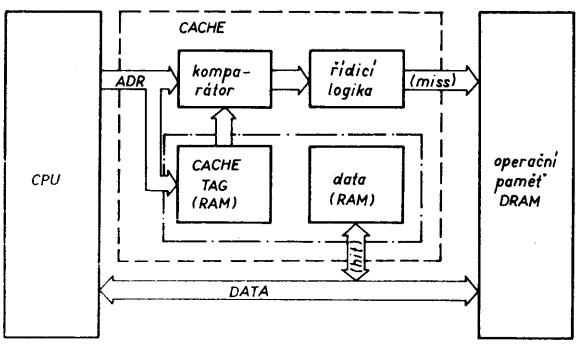
Prefetch/pipeline

Principem této strategie je předvýběr omezeného počtu běžných, sekvenčně řazených instrukcí z operační paměti vhodně upravenou vazbou CPU se systémovou sběrnicí, využívající krátké instrukční fronty. Předvýběr probíhá v těch intervalech strojového cyklu, kdy CPU nevyžaduje komunikaci se sběrnicí (když provádí vnitřní operace). Tím je vytvořen základní předpoklad pro pipeling – čtení nové instrukce při současném provádění instrukce aktuální. Obě fáze (předvýběr a zpracování odlišných instrukcí) se tak mohou překrývat, což odpovídá radikálnímu zlepšení propustnosti sběrnice, avšak pouze u sekvenčně řazených instrukcí, kterých je ovšem většina. Vyskytne-li se skoková instrukce, jednotka styku CPU se sběrnicí stávající frontu zruší a začne číst z běžné paměti – vytváří novou frontu instrukcí.

Cache

Operační paměti současných mikropočítačů zpravidla z technologických a ekonomických důvodů užívají dynamické paměti DRAM s dobou přístupu asi 100 až 250 ns. I při předvýběru instrukcí tedy opět dojde k situaci, kdy přístupová doba operační paměti představuje pro další zlepšování propustnosti bariéru. Proto se u posledních generací mikroprocesorů užívá na kritických pozicích styku CPU s operační pamětí tzv. cache (skrytá mezipaměť s různou organizací a kapacitou). V podstatě se vždy jedná o určitou, velmi rychlou alternativu paměti asociativního typu. Základem výstavby cache jsou rychlé, statické paměti RAM a velmi rychlé adresové komparátory.

Struktura cache, znázorněná na obr. 59, užívá dvou sekcí pamětí RAM s velmi krátkou



Obr. 59. Základní koncepce vnitřní struktury CACHE

dobou přístupu. Do první, datové sekce, se zapisují (podle určité strategie) kopie dat vybíraných z hlavní paměti. Do druhé sekce se jako ukazatele těchto dat zapisují informace o jejich uložení v cache. Vždy, když CPU požaduje čtení z operační paměti, blok rychlých komparátorů detekuje, zda se požadovaná data již nacházejí v cache. Pokud ano (hit – zásah), poskytuje data s minimálním zpožděním přímo cache. Když ne (mis – chyba, vedle), jsou data vyzvednuta z operační paměti s nezbytným zpožděním tak, jako by cache vůbec neexistovalo. Zároveň se však pořizuje jejich kopie do cache a proto v případě opětovné brzké další potřeby již nebude pomalý přístup do operační paměti nutný.

Účinnost systému cache je založena na předpokladu, že většina úseků prováděného programu má lokální charakter, opakuje se v cyklech a smyčkách. Při dostatečně vysoké kapacitě cache je velká pravděpodobnost úspěšné aktivity cache (hit rate) a tím také podstatného zlepšení propustnosti počítače. Cache se samozřejmě vytvářejí, implementují a užívají různými způsoby (interní on chip CPU nebo MMU, externí s relativně velkou kapacitou, instrukční, datová aj.).

Architektura RISC

Skutečnou cestu k systematickému zvětšování výkonnosti počítačů nabízí nejen koncepční systémový vývoj, ale i přehodnocování dosavadních vývojových cest.

K velmi diskutovaným, ale i prakticky využívaným cestám patří dnes architektura RISC (Reduced Instruction Set Code). Je vlastně založena na návratu k „jednoduché" koncepci procesorové jednotky s omezeným, relativně „primitivním", ale pečlivě vybraným souborem mikroinstrukcí s převážně pevným instrukčním formátem. Vychází se ze skutečnosti, že každý program je vždy překládán ze zdrojového tvaru do strojového kódu. K optimalizaci cílového programu však nejsou dosud uvažované několikacyklové instrukce nejvhodnější. Minimální časové režie lze naopak dosáhnout souborem jednoduchých, avšak velmi rychlých, nejlépe jednocyklových instrukcí. Přitom je technická stránka realizace procesoru, cílevědomě se zbavující složitého mikroprocesorového řadiče a dalších sdružených obvodů, jednodušší. Architektura RISC se dosud uplatňuje především v oblasti minipočítačů. Odtud se však do dnešních monolitických procesorů prosadily i koncepce pipelingu a cache.

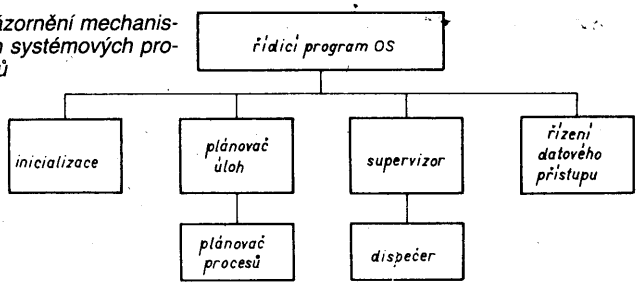
Multipolní operační systémy

Až potud jsme uvažovali průchodnost počítače pouze v závislosti na technických prostředcích, umožňujících rychlejší převzetí a interpretaci instrukce. To je ovšem část celého problému. Máme už určitou představu o tom, co z praktického hlediska využití počítače znamená vnější paměť a operační systém. Také jejich další zdokonalování vyžaduje podporu technických prostředků.

Monopolní operační systémy, tvořící i dnes základ standardních operačních systémů perzonálních mikropočítačů, mají řadu aplikačních nedostatků. Především mohou v určitém čase řešit vždy pouze jedinou úlohu jediného uživatele, které přidělují celý volný, spojitý nerezidentní prostor operační paměti. Teprve po vyřešení této úlohy může být aktivována další úloha. Tímto způsobem je však počítač včetně jeho periférií využíván z časového a ekonomického hlediska značně neefektivně.

S rozvojem technických prostředků dochází i v oblasti mikropočítačů k postupnému uplatňování podstatně výkonnějších multipolních (paralelních) operačních systémů, umožňujících víceuživatelský (multiuser) nebo multiprogramový (multitasking) pra-

Obr. 60. Symbolické znázornění mechanismu přidělování sdílených systémových prostředků



covní režim. To vše znamená, že buď může na jednom počítači současně pracovat několik uživatelů s iluzí, že počítač slouží výhradně jim, nebo může být souběžně, paralelně řešeno několik samostatných, případně i spolupracujících úloh.

Pro jednoznačnost budeme dále chápat program jako předpis definující přesně způsob provádění určité úlohy (task). Proces je činnost, odpovídající provádění programu, buď systémového, nebo uživatelského.

Program obecně nemusí být prováděn v jediném, spojitém časovém úseku. Může být řešen postupně, po jednotlivých procesech, přičemž se mohou střídat v okamžité aktivitě procesy jednotlivých úloh. Multipolní operační systémy musí zajišťovat vzájemnou synchronizaci úloh a procesů a přidělovat jim sdílené technické prostředky, především CPU a operační paměť. Aktivní může být vždy pouze jediný proces. Musí být evidovány stavy jednotlivých úloh a vzájemně si konkurujících procesů a na tomto základě, společně s řízením přenosů mezi operační a vnější pamětí, řízeno i pořadí zpracovávaných procesů a komunikace s operátorem. To vše v co možná optimálním časovém prostoru. Všechny činnosti musí být přesně určeny, systém musí vylučovat vzájemné kolize a zablokování jednotlivých úloh a procesů. Zvláště pak musí být schopen vyhodnotit, zpracovat a ošetřit všechny havarijní stavy, jako je například nekorektní přístup k systémové paměti.

K účinnému splnění uvedených požadavků je nezbytná výrazná podpora technických prostředků, především procesoru a jednotky řízení paměťového přístupu. CPU je jednotlivým procesům přidělována operačním systémem buď staticky, nebo dynamicky. Zpravidla však na určité, předem stanovené časové úseky s využitím prioritního systému.

Také paralelní systémy se skládají ze tří skupin systémových programů: řídicích (organizačních), služebních a překladačů. Hlavní část tvoří řídicí programy plánování a kontroly zpracovávání úloh, procesů a správy dat, obr. 60.

Plánovač úloh sleduje a eviduje stav všech úloh v systému; podle jejich prioritních a ochranných atributů a požadovaných prostředků určuje pořadí jejich provádění.

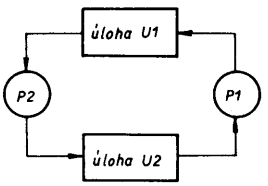
Plánovač procesů rozhoduje o tom, kterému z procesů ve stavu připraven bude přidělen procesor. Dispečer sleduje stav procesů, zajišťuje úklid stavu procesu při jeho odebírání a jeho obnovu při opětovné aktivaci.

Prostřednictvím supervizoru (dohlížecího programu) může uživatelský program volat složky řídicího programu (zavedení a využití nového programu, ošetření nestandardní situace . .). K volání se využívá mechanismu přerušení a privilegovaných instrukcí. Při ošetřování žádosti nesmí být operační systém přístupný žádnému uživatelskému programu, nesmí být sdíleny systémové a uživatelské programy.

Správa dat zajišťuje součinnost a sdílení hlavní paměti se systémem. Hlavní pamětí míníme obecně jak operační paměť, tak vnější diskovou (disketovou) paměť, přímo komunikující se systémem. Správa dat eviduje aktuální stav a umístění programových, datových a zásobníkových alokačních bloků jednotlivých úloh a zajišťuje, zpravidla s využitím transformace interních (logických, virtuálních) a vnějších (fyzických) adresových prostorů přístup systému k jednotlivým úlohám a programům.

Pro časově efektivní a přitom bezpečné přidělování systémových prostředků je nutno zamezit dlouhým dobám jejich užívání jedním procesem v situacích, kdy by to znamenalo blokování jiných urgentních činností. Tak by například mohla vzniknout situace, kdy úloha U1 čekáním na uvolnění určitého prostředku P1 blokuje prostředek P2, na který opět čeká úloha U2 – tím by se mohl zablokovat celý systém, obr. 61, čemuž zabrání pouze omezení doby trvání procesu, disponujícího kritickým prostředkem.

Optimalizace přidělování prostředků jednotlivým úlohám je založena na jejich rozdělení do určitých tříd, v závislosti na požadované prioritě zpracovávání. Sdílení systémových prostředků proto musí řídit výluč-

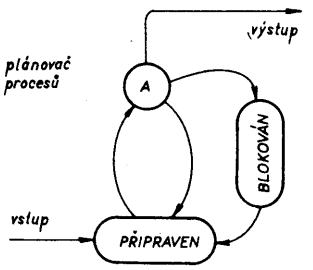


Obr. 61. Příklad možného zablokování systému čekáním na přidělení prostředku (CLINCH)

ně operační systém, nejlépe (z hlediska propustnosti) dynamickou správu procesů. Přitom systémový proces má vždy vyšší prioritu, než uživatelský.

Každý z dynamicky přidělovaných procesů multipolního systému má vždy přiděleny prioritní atributy a může se v určitém čase *t* nacházet minimálně v jednom ze schématicky znázorněných stavů, obr. 62:

1) aktivován – tj. má přiděleny systémové prostředky a právě probíhá,
2) připraven – je podle určitého kritéria zařazen do fronty dalších procesů, čekajících na aktivaci,
3) blokován – je opět zařazen do další fronty blokovaných procesů, čekajících na události, které je postupně převedou do fronty procesů připravených.



Obr. 62. Vytváření front připravených a blokovaných procesů, výběr a ukončení aktivního procesu

Aktivním (běžícím) může být v multipolním systému proces pouze po omezený časový úsek. Jestliže nedojde během tohoto intervalu k jeho úplnému zpracování, je po uplynutí přidělené doby přerušen. Mimoto může být přerušen aktivací jiného procesu s vyšší prioritou. Přerušený proces je vždy uklizen do některé z front (připravených, blokovaných) procesů. Při umísťování a výběru procesů užívají operační systémy různé algoritmy, zohledňující však vždy tyto požadavky:

1. *Optimalizace časového průběhu zpracování*
   I při preferenci určitých úloh dostávají ve frontách zpravidla přednost procesy s nejkratší požadovanou dobou trvání. Naopak pomalé procesy, typicky ty, které vyžadují přístup k zařízením I/O, mívají prioritu nejnižší.
2. *Vzájemná bezpečnost jednotlivých úloh* a tedy jednoznačně determinovaný přístup k jednotlivým procesům a vyloučení možnosti zablokování systému. Při přechodu od jednoho k druhému procesu musí být uklizen stav odkládaného a obnoven stav aktivovaného procesu.

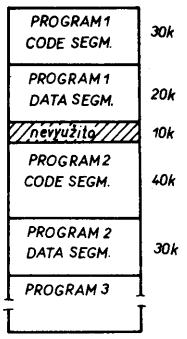
Fronty chápeme jako archiv neúplně vyřízených žádostí úloh nebo procesů o přidělení systémových prostředků. Pro synchronizaci přidělování systémových prostředků jednotlivým procesům se užívá logických konstrukcí, označovaných jako semafory a kritické oblasti, bližší informace viz např. [17], [18].

Nedílnou součástí každého multipolního operačního systému musí být efektivní, operativní součinnost operační a vnější sekce hlavní paměti. Vnější paměť je vždy z principu pomalá. Čím výkonnější a rozsáhlejší je operační systém, tím vyšší nároky klade na technické prostředky počítače, pomáhající omezit nezbytnou časovou režii při obousměrném přemisťování elementárních paměťových alokačních bloků (segment, stránka) mezi diskovou a operační pamětí. Postupem doby byly vyvinuty různé metody organizace a transformace paměťových prostorů, podporující tuto součinnost s ohledem na praktické požadavky a technické možnosti. Existují a užívají se především dvě základní koncepce, segmentace a stránkování. Pochopení jejich podstaty a principu virtualizace paměťového prostoru je podmínkou pro orientaci ve strukturách 16 a 32bitových mikroprocesorů, které se vyvíjely právě z hlediska podpory multipolních systémů.

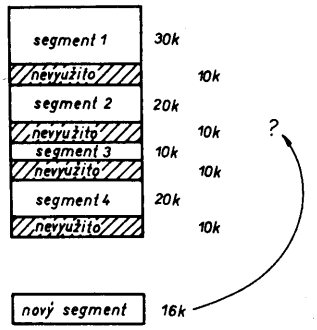
Segmentace

Paralelní zpracování úloh a programů nutně předpokládá jejich současnou existenci v hlavní paměti. V operační části této paměti přitom musí být vždy umístěny pouze právě potřebné bloky těchto programů. Jednotlivé operační systémy přitom kladou na rozsah či úplnost těchto složek různé požadavky. V návaznosti na diskutovanou koncepci segmentace operační paměti budeme dále zmíněné bloky označovat jako segmenty, přičemž segmentem rozumíme souvislý paměťový úsek v podstatě libovolné délky (zpravidla ≦ 64 kB), celý přidělený programové nebo datové složce jednoho programu, obr. 63.

Při segmentové organizaci mohou být mezi diskem a operační pamětí vyměňovány programové a datové segmenty bez omezení potud, pokud budou mít v operační paměti dostatečně velký volný prostor. Z hlediska praktického užití se ovšem nové segmenty umisťují v operační paměti s omezenou kapacitou vždy na místa segmentů, odložených zpět do vnější paměti. Jejich rozsah tedy musí být vždy menší nebo maximálně roven uvolněnému prostoru. Tak se projevuje jeden z hlavních nedostatků metody, kterým je tzv. externí fragmentace paměti, obr. 64. V operační paměti se vytvářejí nevyužité úseky, fragmenty. Může nastat (a nastává) situace, kdy třeba pro segment 16 kB není v paměti volné místo, i když je v ní 40 kB volného, nevyužitého prostoru. Čím větších segmentů se užívá, tím rozsáhlejší fragmentace nastává. Existuje řada metod, jak se s tímto jevem vyrovnat. S relativně nízkými nároky na technické prostředky CPU to umožňuje metoda přemisťování segmentů tak, aby vytvořily souvisle obsazené a tedy i uvolněné prostory. To ovšem vyžaduje postupně kopírovat segmenty do vnější paměti a zpětně je zapsat od jednoho konce do operační paměti. Tato časově neefektivní metoda není pro multipolní systém vhodná.

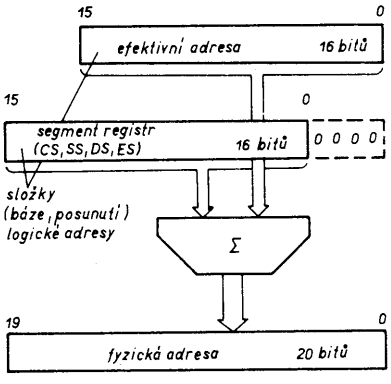


Obr. 63. Segmentace znamená dělení rozsahu operační paměti na segmenty různých délek. Každé úloze je přidělen samostatný programový a datový segment



Obr. 64. Znázornění příčin vzniku externí fragmentace paměti při segmentovém přidělování prostoru paralelním úlohám

Mechanismus segmentových výměn mezi vnější a operační částí hlavní paměti předpokládá systém segmentových bázových registrů, spravovaných výlučně řídicím programem. Poprvé byl implementován v CPU 8088 a 8086. Tyto procesory se užívají například v počítačích typu PC–XT s operačním systémem MS–DOS, který je však nadále monopolní. Adresování v celém prostoru systémové paměti je založeno na využití dvousložkové vytvářené logické adresy, která je základem efektivně vytvářených přemístitelných segmentů.



Obr. 65. Znázornění vztahu logické a fyzické adresy položky segmentu

Logická adresa využívá systému segmentových (bázových) a efektivních adres, obr. 65. Procesor je vybaven čtyřmi 16bitovými segmentregistry:
CS (Code Segment) DS (Data Segment)
SS (Stack Segment) ES (Extra Segment)
Těmi je možno kdekoli v operační paměti vytvořit a adresovat libovolně velké bloky, segmenty o max. délce 64 kB. Počátek (bázi) segmentu určuje příslušný segmentový registr. Umístění jednotlivé položky uvnitř segmentu (program, data, stack) pak určuje 16bitová efektivní adresa, která může být vytvořena různými způsoby podle adresovací metody a typu instrukce a představuje ofset položky vůči bázové adrese. Proto při přemístění celého segmentu v operační paměti stačí pouze změnit obsah segmentového registru.

Úplná logická adresa se tedy skládá ze dvou složek, 16bitové segmentové a 16bitové efektivní adresy. Ty společně určují umístění každé položky segmentu v logickém adresovém prostoru.

Na vývodech procesoru je však k dispozici pouze 20bitová fyzická adresa, odpovídající vždy umístění adresové položky v reálném, fyzickém adresovém prostoru. Tato adresa se technicky vytváří tak, že 16bitová segmentová složka je ve skutečnosti interpretována jako 20bitová (čtyřnásobným posuvem vlevo) a k ní se lineárně přičítá efektivní adresa jako ofset. Tak je vytvořen rozsah fyzických adres 0 až OFFFFF H, tj. 0 až 1 MB.

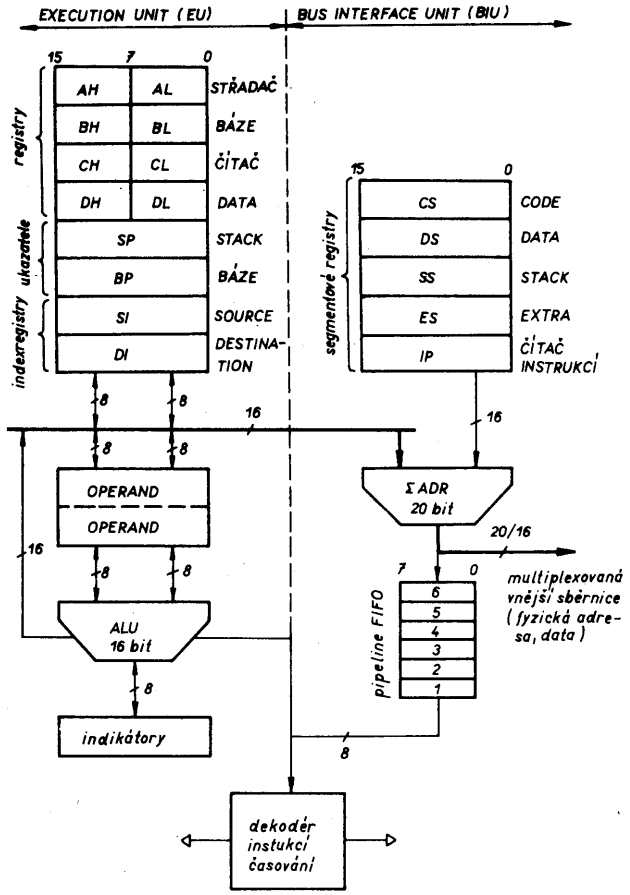
Logické adresování představuje prvý krok k vytvoření virtuálního adresového prostoru, kterým však CPU 8086 ještě nedisponuje. Přesto systém logických adres patřil, právě zavedením dosud nebývalého a přemístitelného adresového rozsahu, spolu se 16bitovou datovou sběrnicí, instrukční frontou, odpovídajícím rozšířením instrukčního souboru i adresovacích možností k hlavním trumfům tohoto mikroprocesoru v době jeho uvedení (blokové schéma je na obr. 66, literatura [21], [22]).

Shrneme-li, segmenty představují spojité programové nebo datové bloky, jejichž umístění v operační paměti lze snadno měnit. Segmenty mohou být umístěny v paměti kdekoliv, mohou se i částečně nebo úplně překrývat. Se segmenty pracují uživatelské programy, se segmentovými registry však musí, pro zajištění bezpečné spolupráce s vnější pamětí, zacházet výlučně operační systém. Předností segmentového mechanismu jsou z tohoto hlediska relativně nízké požadavky na podporu ze strany CPU.

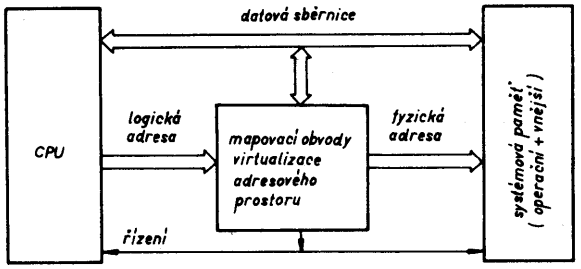
Virtuální paměť

Ačkoli v době nástupu procesorů 8088 a 8086 byli uživatelé vesměs šokováni „obrovsky zbytečným" adresovým prostorem 1 MB, začalo se brzy ukazovat, že „to zase není až tak moc". A v okamžiku zpřístupnění disků Winchester to najednou bylo málo. Východisko ze situace opět ukázaly minipočítače – byla a je jím nadále virtualizace adresového prostoru, naznačená už u 8086 logickým adresováním. To však dovolovalo pouze segmentaci v omezeném reálném adresovém prostoru operační paměti.

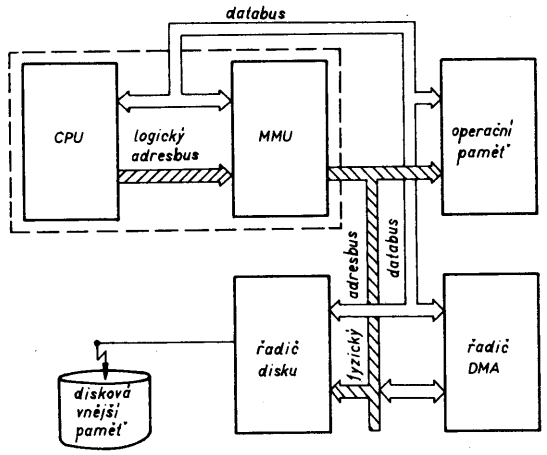
Skutečný virtuální paměťový systém však umožňuje vytvářet a dynamicky spravovat i mnohonásobně větší logický/virtuální adresový prostor, než jaký zabírá nebo vůbec



Obr. 66. Zjednodušená vnitřní struktura CPU 8086



Obr. 67. Virtualizace paměťového prostoru systému



Obr. 68. Vazba operační a vnější paměti na virtuálně orientovaný systém počítače

může zabírat instalovaný rozsah fyzické operační paměti a to při velké propustnosti systému. Uživatel má přitom dojem přístupu k celému tomuto virtuálnímu prostoru, i když ten je ve skutečnosti spravován mnohem menší kapacitou operační paměti. Proto také označení virtuální, zdánlivý. Tato mimořádně cenná možnost efektivní spolupráce nerovných logických a fyzických prostorů, umožňující šetřit drahým prostorem operační paměti, ovšem vyžaduje podstatně výkonnější a složitější podporu ze strany technických prostředků počítače, zvláště CPU a MMU. Vedle potřeby mapování paměťového prostoru a evidence umístění úloh v paměti je nutná identifikace všech nevhodných paměťových přístupů, které musí být okamžitě vyhodnoceny operačním systémem, aktivovaným vyvoláním odpovídajícího výjimečného stavu procesoru, obr. 67.

Virtuální adresování opět užívá dvousložkových logických adres. První složku však nyní představuje tzv. selektor, druhou znovu efektivní adresa, určená instrukcí. Vzájemné přiřazení logické, virtuální a fyzické adresy však už není určeno jednoznačnou lineární skladbou, ale uskutečňuje se prostřednictvím zvláštního „mapovacího" obvodu tzv. segmentových deskriptorových tabulek. Uživatelský program pracuje výhradně s logickými adresami a o skutečném umísťování segmentů v operační ani vnější systémové paměti nemá (a nepotřebuje) žádné informace. Vzájemné přiřazení obou prostorů zajišťuje efektivně (čas, bezpečnost, priority . .) operační systém. Virtuální paměť přitom tvoří všechna paměťová média, k nimž má procesor a celý systém přístup. Operační paměť z toho zpravidla představuje pouhý zlomek. Tu může procesor adresovat prostřednictvím fyzické adresové sběrnice. Zbytek náleží vnějším pamětem, tedy především disku floppy a Winchester. Vnější paměť je ovšem dostupná pouze nepřímo. To znamená, že segment z vnější paměti, k němuž má mít procesor přístup, musí být nejprve překopírován do paměti operační. Přesuny se musí uskutečňovat pod výlučnou kontrolou operačního systému, který při tom musí, za podpory technických prostředků uskutečňovat řadu zásadních rozhodování, např.:

- kam segment do operační paměti umístit?
- je před tím třeba některý segment uklidit, nebo může být přepsán?
- když „uklízet", tak který segment? Kritéria při tom mohou být různá, např. fragmentace nebo časová režie (uklizený segment může být vzápětí znovu požadován a tedy znovu umisťován).

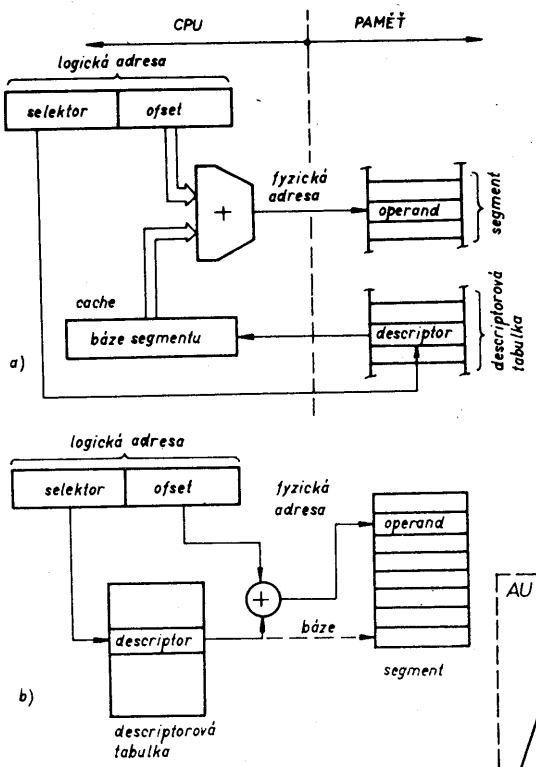
Způsob vazby operační a vnější paměti na systém s virtuálním adresováním prostřednictvím jednotky BU nebo MMU (viz dále), která může být buď samostatným, podřízeným procesorem, nebo tvořit součást CPU, schématicky postihuje obr. 68. Adresové výstupy MMU však vždy produkují reálnou, fyzickou adresu.

Vlastní princip virtuálního segmentového adresování znázorňuje obr. 69. Bázová adresa segmentu je určována nepřímo. Její vytvoření je iniciováno selektorem, který působí jako ukazatel do tzv. deskriptorových tabulek, vytvářených a aktualizovaných systémem v operační paměti. Každému segmentu je přiřazen deskriptor, tvořený několika byty. Při naplnění příslušného segmentového registru selektorem je z tabulky vybrán deskriptor, určující bázovou adresu odpovídajícího segmentu a tedy i jeho fyzické umístění v operační paměti. Po lineárním sloučení báze a ofsetu (efektivní adresy) je určena fyzická adresa odpovídající položky v rámci segmentu. Pro snazší pochopení je princip na obr. 69 znázorněn dvěma odpovídajícími si způsoby.

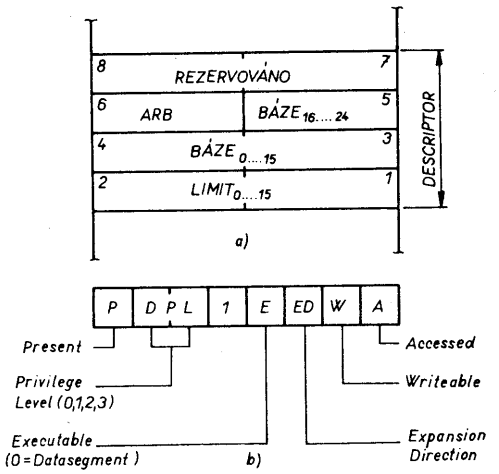
Při každém přístupu k segmentu úlohy potřebuje procesor znát obsah jeho deskriptoru. Stálé čtení z pomalé operační paměti (DRAM) by odezvu systému zbytečně zpomalovalo. Proto se obsah právě aktivního deskriptoru ukládá do speciálního interního registru, pracujícího jako cache. Každý segmentregistr má k dispozici vlastní cache, které po celou dobu užívání jednoho segmentu nahrazuje zbytečný, časově náročný přístup CPU k deskriptorové tabulce, obr. 69a.

Neméně důležitá je nutnost zabezpečit korektní paměťový přístup vzhledem k nerovným kapacitám virtuálního a fyzického adresového prostoru. Proto se vedle bázových adres segmentů umisťují do deskriptorů i další položky, atributy stavu, umístění a přístupových práv. Na obr. 70a je jako příklad znázorněn formát datového segmentového deskriptoru CPU 80286, který je 8bytový. Nejnižší dva byty nesou informaci o velikosti segmentu (limit), další tři (24 bitů) určují vlastní bázi segmentu. Nejvyšší dva byty pak jsou výrobcem rezervovány pro další inovace. Pro nás je nyní zajímavý 6. byte, tzv. Access Right Byte, obr. 70b. Jeho bit P (Present) udává, zda je segment umístěn v operační paměti (P=1), nebo zda musí být před užitím nejprve umístěn, tedy „natažen" z paměti vnější. Druhý krajní bit A (Accessed) může systém použít pro organizaci segmentových výměn. Tento bit je vždy automaticky nastaven při aktivaci segmentu. Periodickým čtením a nulováním bitů A může systém získat evidenci o četnosti užívání jednotlivých segmentů a podle toho řídit jejich úklid. Ostatní bity určují přístupová práva a privilegia. Každé porušení jimi evidovaných pravidel způsobí odpovídající typ přerušení, vyvolávajícího příslušné ošetření.

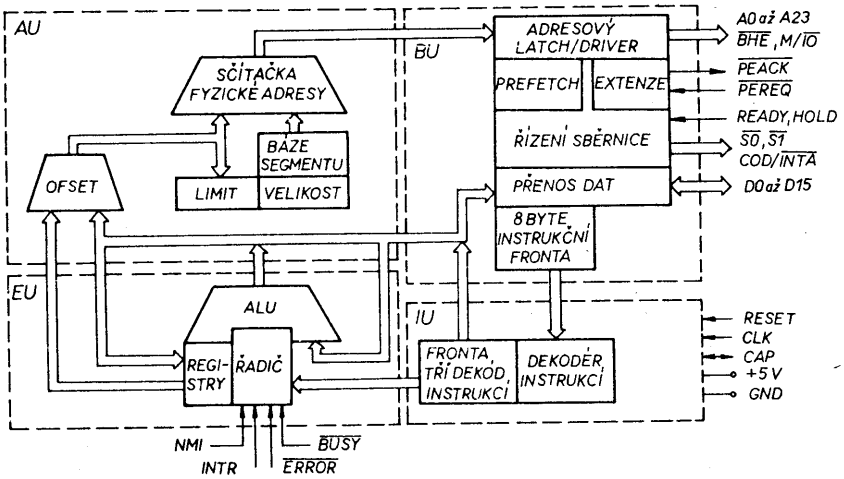
Popsaným adresovacím mechanismem disponuje ve virtuálním módu 16bitový mikroprocesor 80286, jehož orientační blokové schéma je na obr. 71. Procesor, který je inovací 8086, již může být užit ve virtuálním módu pro paralelní zpracování úloh. Umožňuje adresovat až 1 GB virtuální pa-



Obr. 69. Dvěma způsoby znázorněný vztah logické a fyzické adresy ve virtualizovaném segmentovém paměťovém prostoru



Obr. 70. a) formát segmentového deskriptoru, b) ACCESS RIGHT BYTE



Obr. 71. Blokové schéma mikroprocesoru 80286

měti do fyzického prostoru 16 MB. Jeho vnitřní struktura se skládá ze čtyř spolupracujících jednotek.

Výkonná jednotka EU (Execution Unit) provádí příslušné aritmeticko/logické operace. Instrukční jednotka IU zajišťuje dekódování instrukcí a ovládání řadiče EU. Adresovací jednotka AU transformuje virtuální adresu na fyzickou a zajišťuje její přenos do BU. Konečně jednotka BU (Bus Unit) zajišťuje vazbu celého procesoru na systémovou sběrnici (16 bitů data, 24 bitů adresa) a prefetch instrukcí.

Přehledový popis 80286 včetně obou módů (reálného a virtuálního), globálních a lokálních deskriptorových tabulek, systému privilegií aj. viz [23], podrobnější informace [22].

Stránkování

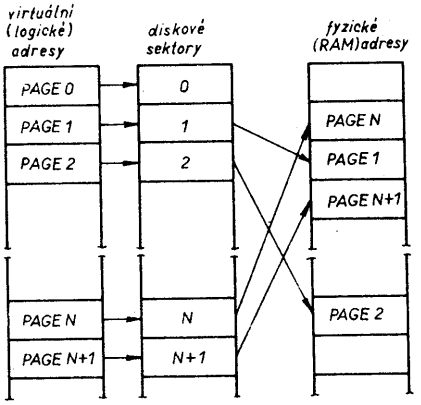
Ani virtuálně orientované segmentové systémy neřeší nedostatky, vyplývající z proměnného formátu logické jednotky (segmentu) do důsledku. Protože při zpracování úlohy musí být v operační paměti přítomen vždy úplný formát příslušného segmentu, nelze mezi operační pamětí a diskem přemisťovat jeho část, ale pouze celý, spojitý segment současně. To je při práci s rozsáhlými segmenty časově značně náročné a zásadně omezuje propustnost celého systému.

Koncepce virtuálních adres umožňuje zavést výkonnější system spoluprace obou složek (virtuální, fyzické) hlavní paměti. Je jím tzv. stránkování (paging). Tento způsob paměťového přístupu však také klade podstatně větší nároky na technické prostředky, především jednotku MMU. Podstatou stránkování je, jak už napovídá název, dělení spojitého logického paměťového prostoru na zcela shodné úseky, stránky. Těm ve fyzickém prostoru operační paměti odpovídají stejně velké úseky, označované jako stránkové rámy (Page Frames). Konstantní rozměr stránky jako základní přemístitelné jednotky automaticky odstraňuje problém externí fragmentace – při každé výměně stránky je vždy využit celý její prostor. Zatímco logický prostor segmentu je tedy i nadále spojitý, tvořený sousedními stránkami, jeho umístění v operační paměti už může být nespojité. Jednotlivá stránka může být umístěna kdekoliv, v libovolném rámu. Stačí, aby umístění stránky bylo přehledně a operativně řízeno a evidováno. Uvážíme-li dále možnou shodu rozměrů stránek/rámů s formátem sektoru pevného disku, obr. 72, máme rázem dobrou představu o vzájemné zaměnitelnosti lokace stránky v celém prostoru hlavní systémové paměti. Pro lepší propustnost systému může být nyní místo již zbytečného hledání prostoru pro umístění segmentu řešena strategie časově efektivního hospodaření s operační pamětí. Vychází se z různých algoritmů, založených však vždy na hodnocení pravděpodobnosti potřeby jednotlivých stránek v blízké budoucnosti podle četnosti jejich dosavadní aktivace v určitém časovém úseku. Při relativně velkém počtu malých stránek a potřebě rychlé odezvy je optimalizace složitý úkol. Má také smysl pouze v souvislosti se zvláštním přístupem k užívání stránkového systému, tzv. stránkování na žádost.

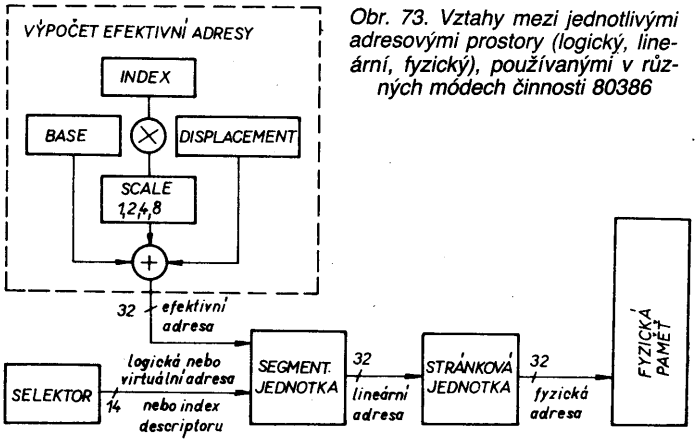
Všimněme si ještě, že stále zůstává neřešen problém interní fragmentace. Vyžaduje-li například program kapacitu 10 kB a stránka má standardní velikost 4 kB, je přidělený prostor tří rámů využit na 80 %. Vidíme, že čím menší budou stránky, tím menší bude i plýtvání prostorem RAM.

Stránkování na žádost

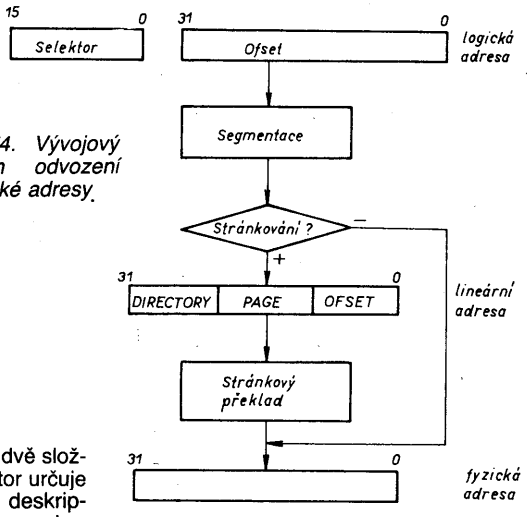
Metoda stránkování na žádost (Demand Paging) zavádí do využívání stránkového mechanismu zcela novou kvalitu. Základní jednotkou výměny už není segment, ale jednotlivá stránka. Operační systém využívající stránkování na žádost je pak koncipován tak, že v případě potřeby může umístit jednotlivou stránku do rámu operační paměti i teprve tehdy, když je už učiněn procesorem pokus o přístup k ní. Jednotlivé programové bloky (segmenty) pak nezpřístupňují své adresové prostory přímo, ale prostřednictvím příslušného počtu stránek, popsaného jejich



Obr. 72. Schematické znázornění koncepce a využití stránkové virtuální paměti



Obr. 73. Vztahy mezi jednotlivými adresovými prostory (logický, lineární, fyzický), používanými v různých módech činnosti 80386



Obr. 74. Vývojový diagram odvození fyzické adresy

seznamem. Jednotlivé stránky každé úlohy se pak mohou nacházet jak ve vnější, tak operační paměti. Umisťovaná stránka se ukládá na místo právě volného rámu. Můžeme rázem nahlédnout celou řadu problémů. Při takovémto nespojitém umisťování stránek se vždy musí dynamicky modifikovat jim příslušné fyzické adresy rámů. Mimoto je pro vzájemnou transformaci nutno vést evidenci o aktivním/pasívním stavu stránky, modifikované stránce (Page Copy), o četnosti využívání, prioritách, přístupových právech a privilegiích tak, aby MMU a operační systém měly stále dostatek informací pro řízení paměťové správy a korektnosti přístupů. Velké množství informací, potřebných k této činnosti, klade vysoké požadavky na podporu technických prostředků, zejména jednotky MMU, která je u systému se stránkováním na žádost nezbytnou podmínkou. Zdůrazněme, že na rozdíl od segmentů, s nimiž pracuje uživatel a které představují logické jednotky (code, data), jsou stránky/rámy alokační jednotky pevného rozměru, s nimiž pracuje výlučně operační systém a technické prostředky. Uživatelské programy nemají ke stránkovému mechanismu žádný vztah ani přístup.

Stránkování na žádost, které má opět původ u minipočítačů, přináší vzhledem k segmentaci dvě další významné přednosti:

1. Doba výměn (swapping) stránek a rámů je už vzhledem k jejich konstantnímu a malému rozměru řádově kratší. Technické prostředky (MMU, cache, DMA) ji dále radikálně omezují.
2. Umožňuje zpracovávat rozsáhlý program i v tom případě, že kapacita operační paměti nestačí na jeho umístění.

Naprostá většina mikroprocesorů vyšších generací je dosud orientována na segmentaci. K nejdokonalejším typům, které umožňují práci v segmentovém i stránkovém režimu a tedy i jejich využití pod různými operačními systémy, patří 32bitový procesor typu I80386, kompatibilní na nižších úrovních s 8086 i 80286.Může pracovat ve dvou hlavních operačních módech:

a) v reálném módu (Real Access Mode), v němž obdobně jako 80286 představuje velmi rychlý ekvivalent 8086;
b) v chráněném virtuálním módu (Protected Mode) využívá všech předností prostředků správy virtuálního paměťového prostoru. V chráněném módu se užívá výhradně systému nepřímých logických adres, opírajících se o ukazatele na paměťové segmenty. Tak lze procesor využívat souběžně jako 80286 s 16 MB a 80386 se 4 GB přímé paměti, s virtuálním prostorem 64 TB (terrabyte)!

Pro řízení transformace mezi logickým a reálným adresovým prostorem v jednotlivých modech a režimech činnosti má mimořádný význam jednotka MMU. Přehled o adresových transformacích podává obr. 73. Logickou adresu tvoří vždy její dvě složky, obecně selektor a ofset. Selektor určuje adresu příslušné segmentové deskriptorové tabulky. Ofset je určen adresovou částí instrukce. Lze užít maximálně $2^{14} = 16$ kB selektorů s ofsetem $2^{32} = 4$ GB. Tak je k dispozici $2^{46} = 64$ TB logického prostoru. Segmentová jednotka převádí logický prostor do 32bitového prostoru lineárního. Další zpracování závisí na aktualizovaném pracovním režimu. Pokud není užito stránkování, odpovídá fyzická adresa (dostupná na externím adresovém busu MMU) přímo adrese lineární. Rozdíl ve vytváření fyzické adresy v jednotlivých módech spočívá ve způsobu vytvoření lineární adresy a na tom, zda je uplatněna stránková jednotka.

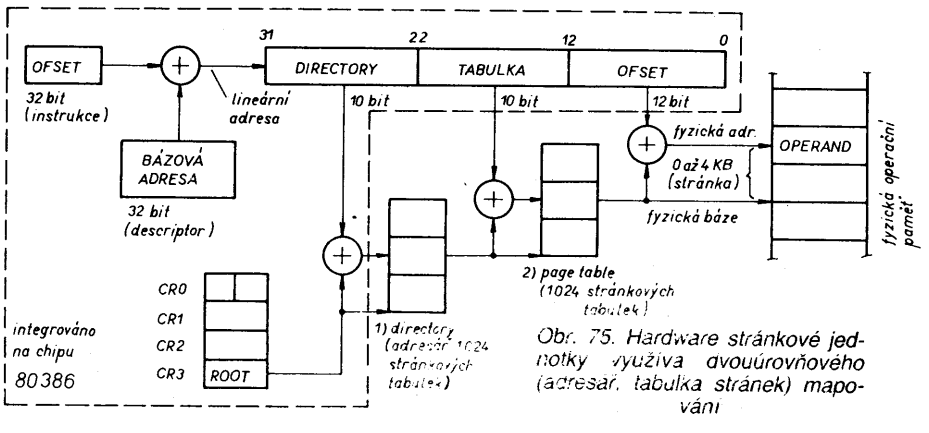
V reálném módu probíhá překlad adresy obdobně jako u 8086, obr. 65.

V chráněném módu je lineární adresa interpretována jako pole o třech složkách. První dvě (directory, page) jsou vyhodnocovány jako indexy do speciálních dvouúrovňových tabulek stránkového překladu, vytvářených systémem. Poslední se využívá jako ofset pro adresování položky uvnitř stránkového rámu fyzické paměti. Vztahy mezi logickou, lineární a fyzickou adresou jsou na obr. 74 znázorněny vývojovým diagramem, podrobněji postihuje překlad obr. 75.

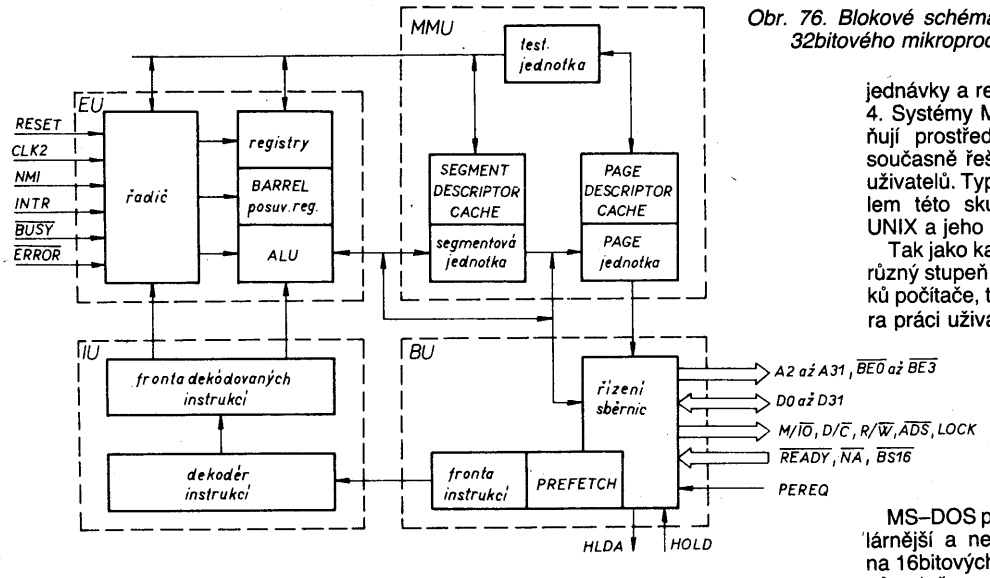
Blokové schéma 80386 je na obr. 76. Architektura podporuje 32bitové datové typy a adresové módy. Registry, interní sběrnice a fyzický datový a adresový bus jsou 32bitové. Segmentové registry CS, DS, SS, ES, FS, GS jsou 16bitové. Kromě obecných registrů obsahuje 80386 blok uživatelsky nepřístupných segmentových registrů – registry deskriptorových tabulek (globální, lokální, přerušení) a řídicí registry CR0, CR2 a CR3. Nyní si již můžeme popsat činnost stránkové jednotky, umožňující kombinovat segmentaci a stránkování paměťového prostoru (obr. 75.).

Jak patrno, stránkování volitelně navazuje na segmentaci, která v podstatě pracuje stejně jako u 80286, obr. 69, s tím rozdílem, že báze i ofset mají 32 bitový rozsah. Použije-li se stránkování, překlad lineární-fyzická adresa užívá dvouúrovňové konverze prostřednictvím dvou tabulek, directory (adresář) a page table (stránková tabulka), organizovaných v RAM. Každá tabulka má pevný rozsah právě jedné stránky (4 kB) a tak umožňuje postihnout 1024 položek délky 4 B. Z toho vždy 20 bitů přísluší adresovému poli. Aktuální úloze může být prostřednictvím indexů do těchto dvou tabulek a ofsetu lineární adresy přidělen v paměti prostor $2^{(10+10+12)} = 2^{32} = 4$ GB, rozdělený do stránek. Při každém zavedení nového aktuálního selektoru je do registru CR3 předána bázová adresa (ROOT) první, adresářové tabulky. Spolu s horními 10 bity lineární adresy (bity 22 až 31), představujícími index, je vytvořena konečná adresa konkrétní položky první tabulky. Na této jsou kromě bázové adresy druhé (page) tabulky uloženy i další informace. Pomocí aktivovaného výstupu adresářové tabulky a bitů 12 až 21 lineární adresy je obdobně získán přístup k aktuální položce stránkové tabulky. Ta již vedle dalšího obsahuje fyzickou adresu požadovaného stránkového rámu, adresu položky v rámci stránky/rámu udává ofset, tj. zbývající bity 0 až 11 lineární adresy. Formát každé z obou překladových tabulek je orientačně znázorněn na obr. 77. Zatímco nejvyšší bitové pole je vyhrazeno adresám, slouží bity P (Present), A (Accessed), D (Dirty) pro řízení přístupu, bity R/W a U/S (User/Supervisor) jako ochranné atributy. Bity 9 až 11 jsou rezervovány pro operační systém, který jich může využít libovolně, zpravidla však pro realizaci algoritmu výměny stránek.

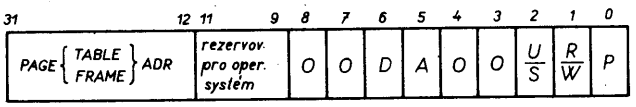
V blokovém schématu na obr. 76 jsou naznačena i využití interních cache jednotky MMU. Smysl objasní nejlépe aplikace cache ve stránkové jednotce – potřeba častého přístupu k pomalé dvouúrovňové překladové tabulce (DRAM) by neúnosně prodlužovala odezvu celého systému. Proto



Obr. 75. Hardware stránkové jednotky využívá dvouúrovňového (adresář, tabulka stránek) mapování



Obr. 76. Blokové schéma interní struktury 32bitového mikroprocesoru 80386



Obr. 77. Formát položek tabulek DIRECTORY a PAGE

se v MMU udržují kopie většiny naposledy zpřístupňovaných stránek právě ve stránkovém cache TLB (Translation Lookaside Buffer).

Podrobnější informace o procesoru 80386 lze nalézt např. v [22], [24]. Na obr. 78 je pro orientaci znázorněno blokové schéma počítače s tímto mikropocesorem a typickými periferiemi osobního počítače.

Stručný přehled operačních systémů

Nakonec si uveďme alespoň stručný přehled nejrozšířenějších operačních systémů s návazností na jednotlivé procesory Intel. Obecně lze rozlišit čtyři základní typy operačních systémů podle jejich schopnosti práce ve víceuživatelském (Single/Multi User) a multiprogramovém (Single/Multi Tasking) režimu:

1. Systémy Single User/Single Tasking jsou výlučně monopolní. Může je provozovat jediný uživatel pro řešení vždy jen jedné úlohy. Typickými představiteli jsou CP/M a MS-DOS.
2. Systémy Single User/Multi Tasking může opět provozovat jediný uživatel s tím, že lze současně zpracovávat několik programů. Zástupci této skupiny jsou systémy Concurent DOS a OS/2. Výhodou multitaskingu je lepší časová odezva a lepší vytížení počítače.
3. Systémy Multi User, označované také jako účastnické, nabízejí několika nebo i mnoha účastníkům společnou datovou základnu, v každém okamžiku ovšem dostupnou jedinému uživateli na principu časového sdílení. Žádný z uživatelů nemůže ovlivňovat program. Typickými příklady aplikace těchto systémů jsou účastnické sítě pro objednávky a rezervace (jízdenky, letenky . .).
4. Systémy Multi User/Multi Tasking umožňují prostřednictvím terminálových stanic současně řešit několik úloh větším počtem uživatelů. Typickým a vedoucím představitelem této skupiny operačních systémů je UNIX a jeho deriváty.

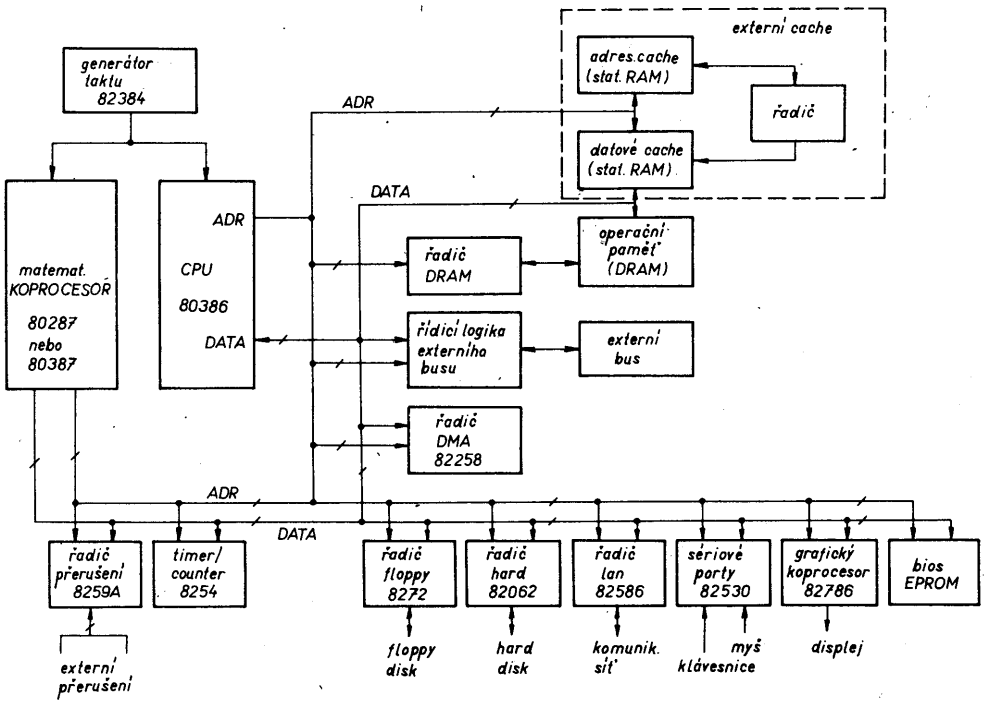
Tak jako každý operační systém vyžaduje různý stupeň podpory technických prostředků počítače, tak různá je jeho vlastní podpora práci uživatele či uživatelů.

MS–DOS

MS–DOS představuje již řadu let nejpopulárnější a nejrozšířenější operační systém na 16bitových počítačích. Trvale se vyvíjí od původně zcela monopolního systému přes určité možnosti zapojování do sítí, rozšiřování uživatelského komfortu (okna, myš . .), interního a externího paměťového prostoru (Winchester) a mnohá další zdokonalení. To vše také odpovídá postupnému přechodu do procesorů 8088 a 8086 k 80286, jehož možnosti však systém, vedle adresového rozšíření, prakticky nevyužívá. Podrobnější informace o systému, jehož hlavní předností je bohaté programové zázemí, neustále doplňované a rozšiřované, viz např. [28].

OS/2

Operační systém OS/2 je, stejně jako MS-DOS, produktem fy Microsoft. Systém byl vyvíjen s cílem dosáhnout razantního zlepšení výkonnosti osobních počítačů IBM se zaměřením na využití 80286. Je to typický systém multitasking, současně probíhající programy si mohou vzájemně předávat a sdílet svá data. 80286 se využívá v chráněném virtuálním módu. Je samozřejmě možné používat procesor i pod MS–DOS, v běžném reálném módu. OS/2 má dosud nedostatky, které zabránily jeho většímu rozšíře-



Obr. 78. Stylizované blokové schéma osobního počítače s procesorem 80386

ní. Nelze jej totiž jednoduše bez úprav implementovat na stávající počítače standardu AT, protože vyžaduje větší kapacitu operační paměti a některé speciální obvody. Mimoto přináší i některé problémy s kompatibilitou, například grafiky. To je vzhledem k potřebě přenositelnosti MS DOS závažný problém.

Concurent DOS

Systém CDOS je produktem fy Digital Research, umožňujícím rozsáhlejší multitasking a s určitými omezeními i multiuživatelské aplikace. Je dostupný ve dvou verzích. Pro aplikace s 8086/80286 je možný přístup ke společné datové základně pro maximálně 6 účastníků. Varianta CDOS na počítače s procesorem 80386 umožňuje propojit 9 terminálů na společný centrální počítač přes sériový kanál RS 232.

CDOS i MS–DOS mohou být instalovány na společném disku, uživatel má možnost volit ten který systém. Programy MS–DOS mohou být spouštěny i pod CDOS. V chráněném módu 80386 může být současně aktivováno až 255 úloh.

PC–MOS/386

Také tento systém fy Softwarelink je založen na rozšíření MS–DOS o multiuživatelské a multiprogramové služby. Pro vybavení jednotlivých uživatelů přitom stačí jednoduché terminály (klávesnice, displej), propojené s centrálním, hostitelským počítačem třídy 80386 sériovým kanálem V.24. Každé pracoviště se při tom chová jako plnohodnotný počítač, všichni účastníci užívají společný pevný disk, na kterém je každé aplikaci přidělen zvláštní pracovní rozsah paměti.

UNIX

Nejrozšířenějším a velmi výkonným, plně multipolním operačním systémem je bezesporu UNIX, vyvinutý u fy Bell Laboratories již v 60. letech. Po celou dobu existence byl systém rozvíjen a inovován na vysokých školách a u významných producentů. UNIX byl původně programován v asembleru. V r. 1974 byla podstatná část napsána v jazyku C, čímž se UNIX stává prakticky technicky nezávislý, přenositelný na každý počítač (mini, mikro) s postačující podporou technických prostředků. Programová přenositelnost je vůči jiným systémům největší předností systému UNIX. Jeho koncepci vychází vstříc zvláště řešení procesoru 80386 s jeho jednotkou MMU. Byly vytvořeny i různé „odnože" tohoto systému, z nichž nejvýznamnější jsou **XENIX** fy Microsoft a **AIX** fy IBM. Stručný přehled systému UNIX viz [29].

Literatura

[1] *Kyrš, F.; Kyrš, T.:* Úvod do číslicové a mikropočítačové techniky. AR B5/1989.

[2] *Jinoch, J.; Muller, K.; Vogel, V.:* Programování v jazyku Pascal. SNTL: Praha 1986.

[3] *Molnár, L.:* Programovanie v jazyku Pascal. Alfa: Bratislava 1987.

[4] *Šaňgin, V. F. a kol.:* Programovanie na jazyke Pascal. Moskva 1988.

[5] *Olehla, M.; Král, F.; Švarc, I.; Tišer, J.:* Programování, programovací jazyky a operační systémy. Skripta FS VŠST, Liberec 1985.

[6] *Kollert, E.:* Výpočetní technika. SNTL: Praha 1987.

[7] *Klein, R. D.:* Floppy-Disk-Aufzeichnungsverfahren. MC 7/84.

[8] *Richta, K.; Zajíc, J.:* Operační systém CP/M pro mikropočítače. ČSVTS 1986.

[9] *Zdeněk, J.:* Technika mikropočítačů, ČSVTS 1983.

[10] *Starý, J.:* Mikropočítač a jeho programování. SNTL: Praha 1988.

[11] *Zajíček, L.:* Bity do bytu. Mladá fronta: Praha 1988.

[12] *Černoch, M.; Stehno, Z.; Vybulková, V.:* Mikropočítač 8048. ST 8/1983.

[13] Embedded Controller Handbook. INTEL 1987.

[14] *Nohel,J. a kol.:* Základní instrukce-mikroprocesor 8048. TESLA ELTOS 1986.

[15] *Zrůst, J.; Šulc, S.:* Jednočipový mikropočítač a mikroprocesor 8051. ST 1/1988.

[16] *Babák, M.; Laurynová, V.:* Programovací jazyk Assembler 8051. TESLA ELTOS 1987.

[17] *Madnick, S. E.; Donovan, J. J.:* Operační systémy. SNTL: Praha 1981.

[18] *Navrátil, V.; Sokol, J.; Žák, V.:* Operační systémy JSEP. SNTL: Praha 1984.

[19] *Hudson, M.; Hausmann, G.:* A designer's guide to virtual memory management. Electronic Engineering, červen 1985.

[20] *Trattnig, W.:* Virtuelle Speicherverwaltung in uC-Systemen. Elektronik č. 2/1986.

[21] *Valášek, P.:* Mikroprocesor 8086, 8088 (obvodová problematika). ČSVTS 1987.

[22] Microprocesor and Peripheral Handbook (část 1.). INTEL 1987.

[23] *Götzinger, J.; Špička, P.:* Úvod do problematiky mikroprocesoru 80286. ST 5/1988.

[24] *Götzinger, J.; Špička, P.:* Úvod do problematiky mikroprocesoru 80386. ST 9/1988.

[25] *Hindin, H. J.:* 32-bit parts and architectures vie for attention. Computer Design,leden 1986.

[26] *Rash, B.; Bodenkamp, J.:* MS–DOS und Unix auf einem Prozessor. Elektronik č. 23/1985.

[27] *Young, S. J.:* Programovací jazyky pro RT-aplikace. SNTL: Praha 1989.

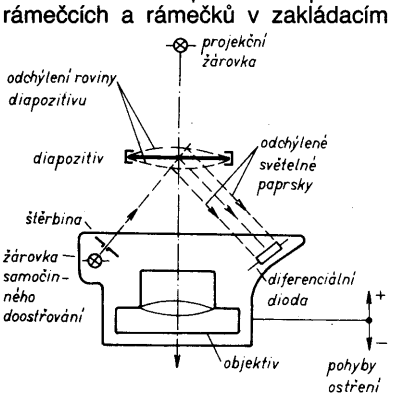
[28] *Richta, K.:* MS–DOS. Elektronika č. 3/1989.

[29] *Mařík, Z.:* UNIX. Elektronika č. 6/1989.

[30] *Hernighan, B. W.; Ritchie, D. M.:* Programovací jazyk C. Alfa: Bratislava 1989.

Samočinné zaostřování u diaprojektorů

Mnohá projekční zařízení bývají vybavena elektronickým optomechanickým doplňkem pro samočinné ostření zobrazovaného předmětu. Nejčastěji se jedná o diaprojektory, kde je samočinně udržována konstantní vzdálenost diapozitivu od optického systému. Značně se tím zjednodušuje obsluha diaprojektoru, neboť diapozitiv se často působením tepla z osvětlovače prohýbá a je nutné během promítání diaprojektor průběžně doostřovat. Také vlastní upevnění diapozitivů v rámečcích a rámečků v zakládacím

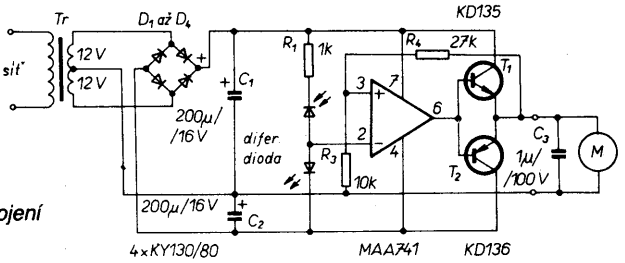


Obr. 76. Samočinné zaostřování u diaprojektorů

mechanismu nezaručuje reprodukovatelnost polohy promítané roviny.

Popisovaným doplňkem může být dodatečně vybaven diaprojektor, který již má dálkově ovládaný motorový mechanismus pro doostřování. U diaprojektorů s ručním ovládáním ostření by byl takový doplněk značně mechanicky náročný a znamenal by prakticky rekonstrukci celého optického systému.

Princip samočinného ostření spočívá v tom, že motorek ostření je zapojen v servosmyčce, jejímž vstupním signálem je napětí z diferenciální fotodiody, na kterou dopadá světelný paprsek, odražený od povrchu promítaného diapozitivu (obr. 76). Diferenciální fotodioda obsahuje dvě fotodiody na jediném polovodičovém čipu. Totožnou technologií výroby a stejnými rozměrovými vlastnostmi obou fotodiod je zaručena jejich stejná tepelná závislost a citlivost. Když úzký světelný paprsek, odražený od diapozitivu, zaměříme na dělicí čáru mezi oběma diodami, je napětí z obou diod totožné. Při odchýlení paprsku směrem k jedné z fotodiod získáme elektrické napětí, které po patřičném zesílení může přímo ovládat stejnosměrný elektormotorek tak, aby odražený paprsek dopadal opět na rozhraní fotodiod. Tímto způsobem se opět rovina diapozitivu ve sledovaném bodě vrátí do původní, zaostřené roviny.



Obr. 77. Elektrické zapojení

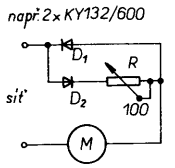
Na obr. 76 je optické schéma doplňku. Žárovka Ž je zdrojem světelného paprsku, který je zaměřen na rovinu diapozitivu, který je umístěn mezi projekční žárovkou a objektivem. Průhyb diapozitivu je znázorněn mezními polohami. Průhyb může být nahrazen různými rozměry rámečku diapozitivu. Odražený paprsek

dopadá na diferenciální fotodiodu. Vzhledem k tomu, že se v ČSSR diferenciální diody běžně nevyrábějí, je nutno použít diody zahraniční produkce (např. BPX48). V krajním případě by bylo možno použít dvě velkoplošné diody, umístěné vedle sebe. Takové diody by bylo nutno vybrat tak, aby měly stejnou citlivost, případně i stejnou teplotní závislost.

Na obr. 77 je schéma elektrického zapojení obvodu. Diferenciální fotodiody jsou zapojeny v sérii a signál z jejich společného bodu je zesílen operačním zesilovačem. Na výstupu operačního zesilovače jsou dva tranzistory opačné vodivosti, které napájí motorek, pohybující diapozitivem ve směru optické osy. Odpory rezistorů R_3 a R_4 je nutno upravit tak, aby byly dynamické vlastnosti obvodu optimální.

Řízení rychlosti otáčení asynchronních motorů

Výhody střídavých asynchronních motorů zajistily tomuto druhu motorů převahu v jejich používání v mnoha aplikacích. Pokud je však požadována změ-



Obr. 75. Řízení rychlosti otáčení asynchronních motorů

na rychlosti jejich otáčení, je nutno změnit kmitočet napájecího napětí, což je však u běžných aplikací prakticky nemožné. Kromě zapojování rezistorů do série s vinutím motoru (tím se zvětšuje „skluz"), existuje však ještě další možnost jak zmenšovat rychlost otáčení zatíženého asynchronního motoru v rozmezí 20 až 30 %. Princip je znázorněn na obr. 75. Vychází z toho, že průchodem stejnosměrného proudu vinutím motoru je motor brzděn. Brzdicí síla je úměrná velikosti stejnosměrného proudu a rychlosti otáčení, což vyplývá ze základních pouček o pohybu vodiče v magnetickém poli.

Pokud má tedy střídavé napájecí napětí motoru stejnosměrnou složku, je motor průchodem střídavého proudu roztáčen a stejnosměrným proudem současně brzděn. Motor se tedy točí pomaleji.

Když je odpor proměnného rezistoru R blízký nule, protéká motorem symetrický střídavý proud a motor není obvodem brzděn. Zvětšujeme-li odpor proměnného rezistoru, rychlost otáčení se zmenšuje.

Proud, který proměnným rezistorem protéká, se přitom příliš neodlišuje od jmenovitého proudu motoru a proto není proměnný rezistor zatěžován příliš velkým výkonem.

Také průběh závislosti mezi rychlostí otáčení motoru a příkonem se výrazně nemění. **Ar**

NOVÁ GENERACE OBVODŮ PRO BTV

Ing. Václav Teska

(Pokračování)

V KO je provozován výstupní signál regulačního zesilovače a obvodu detekce přetížení se signálem obvodu řízení kolektorového proudu a výsledný signál je veden do řídicí logiky. Obvod „vnější blokování" na vývodu *5* IO_1 umožňuje další blokování. Při $U_5 = 0{,}5U_1 - 0{,}1$ V se uzavírá výstup z vývodu *8* IO_1.

V závislosti na obvodu náběhu, identifikace průchodu nulou a uvolněním klopného obvodu se řídicí logice nastavuje „klopný obvod", řídicí obvod zesílení proudu báze a odpojení proudu báze. Zesilovač proudu báze generuje pilovité napětí na vývodu *8* IO_1. Mezi vývody *7* a *8* je přes R_6 zavedena zpětná vazba; R_6 určuje maximální proud do báze T_1. Obvodem odpojení proudu báze řízeného z řídicí logiky se při ochranném provozu zmenší napětí na vývodu *7* IO_1 na 1,6 V a uzavře se zesilovač proudu báze. Ochrana se zapojí, když $U_9 < 6{,}7$ V, nebo když $U_5 < 0{,}5U_{ref} - 0{,}1$ V. Při zkratu na sekundární straně se IO_1 dostává do stavu trvalého dotazu a při odlehčené sekundární straně nastaví IO_1 malou střídu. V obou dvou případech se příkon zmenší na 6 až 10 W. Po uzavření výstupů ($U_9 < 6{,}7$ V) a dalším zmenšení o $dU_9 = 0{,}6$ V se odpojí referenční napětí. K ochraně před podpětím na primární straně a přepětím na sekundární straně Tr_1 (prahová napětí nedosáhne úrovně $U_5 = 0{,}5U_1$) se odpojí výstup na vývodu *8* IO_1, čímž se zmenší příkon zdroje (při $U_9 = 10$ V je $I_9 = 14$ mA). Při velkém odporu při náběhu se U_9 zmenšuje pod práh odpojení (5,7) a odpojí se referenční napětí U_1. Při zmenšeném výkonu se zvětší $U_9 = 12{,}3$ V (práh sepnutí), odblokuje se ochrana na vývodu *5* IO_1 a zdroj se pokouší znovu sepnout. V případě trvající nebo nové závady, když $U_5 = 0{,}5U_1 - 0{,}1$ V, se spínání přeruší, zablokuje se vývod *9* IO_1 a U_9 se zmenší. Příklad provedení plošných spojů je na obr. 23. Parametry TDA4601 jsou v tab. 24.

Ovládání BTVP

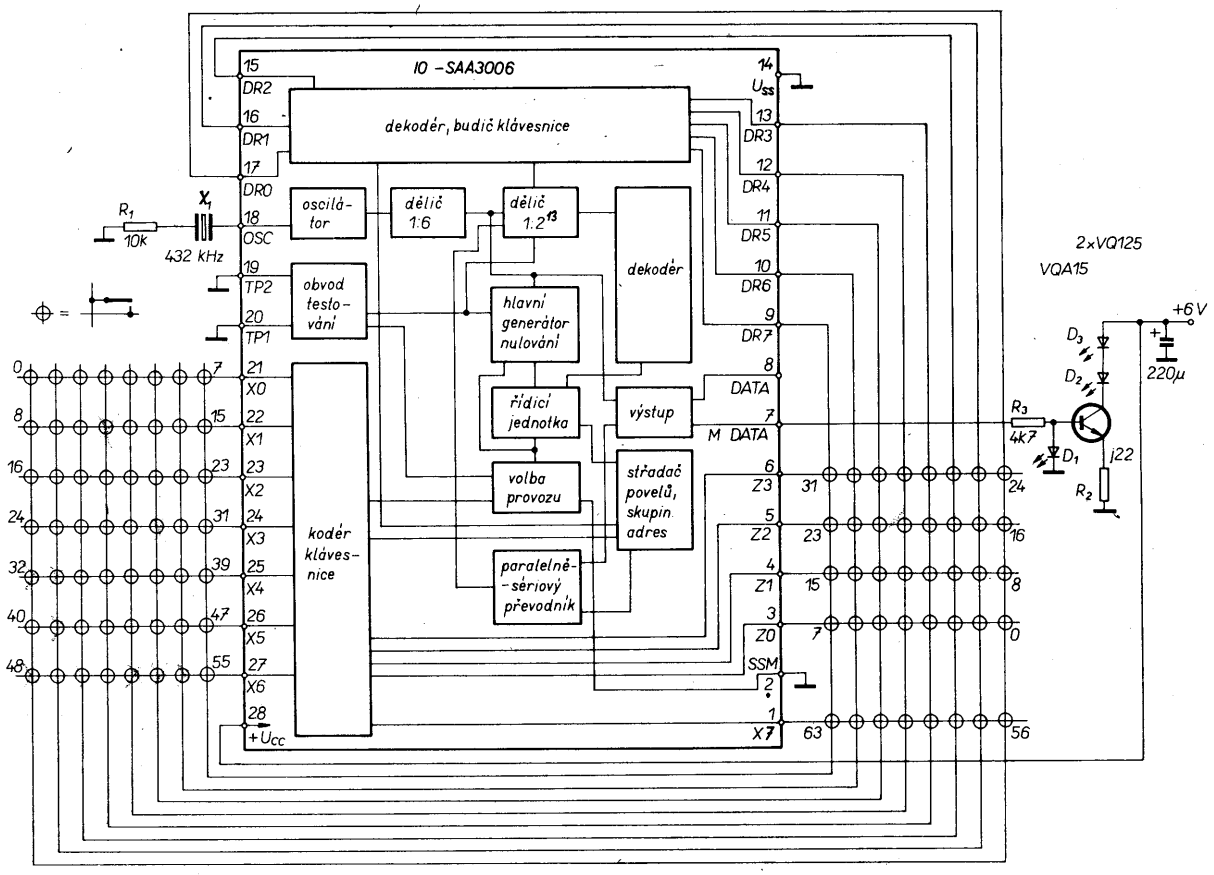
Moderní BTVP používají pro řízení svých funkcí dálkové ovládání – povely z vysílače dálkového ovládání, které jsou přijímány přijímačem dálkového ovládání, jsou zpracovány mikropočítačem a realizovány přes vnitřní sběrnici příslušnými IO.

SAA3006 – kodér infračerveného dálkového ovládání, IDO

SAA3006 je kodér IDO, určený pro vysílače IDO pracující v soustavě RC-5, který generuje 2048 povelů rozdělených do 32 skupin po 64 povelech, z nichž každý je vyvolán tlačítkem s jedním spínacím kontaktem. Přenos je uskutečněn modulací signálu

Tab. 24. Parametry TDA4601

Parametr	Min.	Jmen.	Max.
Napájecí napětí, U_9 [V]	0		20
Referenční napětí, U_1 [V]	0		6
Identifikace průchodu nulou, U_2 [V]	−0,6		0,6
Napětí regulačního zesilovače, U_3 [V]	0		3
Blokovací napětí, napětí úměrné kolektorovému proudu, $U_{4,5}$ [V]	0		8
Napětí na vývodech *7, 8*, $U_{7,8}$ [V]	0		U_9
Vstupní proud, I_2 [mA]	−5		5
I_3 [mA]	−3		3
Výstupní proud, I_7 [A]	−1		1,5
I_8 [A]	−1,5		0
Napájecí napětí, U_9 [V]	7,8		18
Proud při náběhu pro $U_9 = 2$ V, I_9 [mA]			0,5
$U_9 = 5$ V, I_9 [mA]		1,5	2
$U_9 = 10$ V, I_9 [mA]		2,4	3,2
Bod sepnutí U_1, U_9 [V]	11	11,5	12,3
Napájecí proud pro $U_{reg} = -10$ V, I_9 [mA]	110	135	160
$U_{reg} = 0$ V, I_9 [mA]	50	75	100
Referenční napětí pro $I_1 \leqq 0{,}1$ mA, U_1 [V]	4	4,2	4,5
$I_1 = 5$ mA, U_1 [V]	4	4,2	4,4
Teplotní činitel U_1 [1/K]		10^{-3}	
Regulační napětí při $U_{reg} = 0$ V, U_3 [V]	2,3	2,6	2,9
Napětí úměrné kolektorovému proudu pro			
$U_{reg} = 0$ V, U_4 [V]	1,8	2,2	2,9
$U_{reg} = 0/-10$ V, dU_4 [V]	0,3	0,4	0,5
Svorkové napětí, U_7 [V]	2,7	3,3	4
Výstupní napětí při $U_{reg} = 0$ V, U_7 [V]	2,7	3,3	4
U_8 [V]	2,7	3,4	5
Výstupní napětí při $U_{reg} = 0/-10$ V, dU_8 [V]	1,6	2	2,4
Napájecí proud při $U_5 \leqq 1{,}9$ V, I_9 [mA]	14	22	28
Vypínací napětí při $U_5 \leqq 1{,}9$, U_7 [V]	1,3	1,5	1,8
U_4 [V]	1,8	2,1	2,5
Blokovací napětí pro $U_{reg} = 0$ V, U_5 [V]	$-0{,}5U_1 - 0{,}1$		$0{,}5U_1$
Napájecí napětí při $U_{reg} = 0$ V a blok. U_8, U_9 [V]	6,8	7,4	7,8



Obr. 24. Zapojení vysílače dálkového ovládání

kmitočtu asi 36 kHz (infračervený paprsek), a to buď v mnohonásobném provozu (32×64 povelů), nebo v jednoduchém provozu (1×64 povelů), dvoufázovou technikou s poměrně krátkou dobou vysílání. IO má malé napájecí napětí, malý příkon, všechny vstupy s ochranou, pro oscilátor potřebuje jen jeden vývod a je jej možné přepnout na testování správné funkce. Obvod je sestaven z ocsilátoru, děliče 1:6, děliče $1:2^{13}$, dekodéru, řídicí jednotky, hlavního generátoru nulování, obvodu nastavení testování, obvodu pro volbu provozu, kodéru klávesnice, budicího dekodéru klávesnice, paralelně-sériového převodníku, střadače skupinových adres a povelů a výstupního stupně.

Po stisknutí tlačítka klávesnice nejprve po dobu 2 bitů trvá protizákmitová doba, která zajišťuje, že nejsou přeneseny falešné impulsy, vznikající zakmitáváním tlačítka. Po této době jsou vyslány 2 bity startovací, za nimi jeden bit kontrolní, 5 bitů skupinové adresy a 6 bitů vlastního povelu. Všech těchto 14 bitů vytváří kód, za ním následuje mezera 50 bitů a za ní opět kód. Doba trvání jednoho bitu je dána vztahem 1 bit = $3\times2^8 \times T_{osc}$, kde T_{osc} je doba trvání periody oscilátoru. Při kmitočtu oscilátoru 432 kHz trvá jeden bit 1,778 ms.

Obvodem volba provozu (vstup na vývodu *2* – SSM) je možné volit buď mnohonásobný (SSM = L) nebo jednoduchý (SSM = H) provoz. Při mnohonásobném provozu (SSM = L) jsou v klidovém stavu vstupy X a Z na úrovni H. Při správném stlačení tlačítka v jedné z matic Z–DR nebo X–DR se spouští 2bitový protizákmitový cyklus. Když během těchto dvou bitů není přerušen kontakt tlačítka, signál uvolnění oscilátoru je blokován a tlačítko může být uvolněno. Přerušení kontaktu během dvou bitů nuluje vnitřní děje. Na konci protizákmitové doby se odpojí výstupy DR a jsou nastartovány dva ohledávací cykly, během nichž jsou spínány výstupy DR jeden po druhém. Když jsou snímané vstupy Z nebo X na úrovni L, signál uvolní střadače a je zaveden do střadače skupinových adres nebo povelů, podle toho, který vstup matice Z nebo X je snímán. Po blokování čísla skupiny adres generuje obvod poslední povel (všechny bity jsou 1) v daném systému po dobu stisknutí tlačítka. Blokování čísla povelu způsobí generování tohoto povelu současně s číslem skupinových adres. Při uvolnění tlačítka jsou nulovány vnitřní děje, pokud data nejsou vyslána v této době. Pokud je vysílání odstartováno, je signál přenesen celý.

Při jednoduchém provozu jsou v klidovém stavu vstupy X na úrovni H a přes PUT (pull-up tranzistor) jsou odpojeny vodiče Z a vstupy Z jsou blokovány. Pouze správné stlačení tlačítka v matici X–DR startuje protizákmitový cyklus. Není-li během tohoto cyklu přerušeno spojení kontaktu tlačítka, signál pro uvolnění oscilátoru je blokován a tlačítko může být uvolněno. Přerušení během dvou bitů nuluje vnitřní děje. Na konci protizákmitové doby se odpojí PUT vodičů X, kdežto na vodičích Z se PUT sepnou během prvního cyklu ohledání. Propojení v matici Z je přeneseno do skupiny čísel adres a zapamatováno ve střadači skupiny adres. Na konci prvního ohledávacího cyklu se odpojí PUT na vodičích Z a vstupy se znovu zablokují, kdežto na vodičích X se tranzistory znovu připojí. Druhým cyklem ohledání se generuje číslo povelu, které se po blokování přenáší současně s číslem skupinové adresy.

Vstupy povelů X0 až X7 jsou v klidovém stavu na „1", což je dáno PUT. Také při SSM=L jsou i vstupy Z0 až Z3 v klidovém stavu na „1". Když SSM=H, tranzistory jsou odpojeny a proud v matici Z–DR neprotéká.

Kmitočet oscilátoru je určen keramickým rezonátorem 432 kHz na vývodu *18* IO. Při napájecím napětí 2 až 5,25 V lze rezonátor připojit přímo, při napájecím napětí 2,6 až 7 V přes rezistor 10 kΩ.

Detekce uvolnění tlačítka – zvláštní kontrolní bit je přičítán jako doplňující po uvolnění tlačítka. Touto cestou dostává dekodér informaci, jedná-li se o nový povel. To je velmi důležité při vícenásobném zadávání čísla, jako je např. číslo kanálu nebo stránka teletextu. Kontrolní bit je doplňován pouze po ukončení vysílání nejnižšího kódu. Cyklus ohledání se opakuje před vysíláním každého kódu, čímž je zajištěno, že při stlačeném tlačítku během vysílání kódu je generováno správné číslo skupiny a povelu.

Výstupy dat – na výstupu DATA (vývod *8*) je generována informace ve tvaru 14bitového slova, které je sestaveno ze čtyř částí:
– START – 2 bity,
– kontrola – 1 bit,
– skupina adres – 5 bitů,
– povel – 6 bitů.

Výstup MDATA přenáší stejnou informaci jako výstup DATA jen s tím rozdílem, že tato informace je namodulována na 1/12 kmitoč-

Tab. 25. Funkce při testování

TP1	TP2			
L	L	vstup matice	vstup matice	normální
L	H	vstup matice	vstup matice	ohledání a výstupní f 6× kratší doba než norm.
H	výstup f_{osc}6	L	L	nulování
H	výstup f_{osc}6	H	H	výstupní f 3× 2^7 vyšší než norm.

tu oscilátoru, takže bit je sestaven z 32 impulsů. Tím se zmenšuje příkon vysílače IDO, neboť nosný kmitočet má jen 25 % plnění. V klidovém stavu jsou oba výstupy nevodivé (třístavový výstup). Ohledávané budiče DR0 až DR7 mají tranzistor s otevřenou elektrodou D (kolektorem) a jsou v klidovém stavu vodivé. Při správném stlačení tlačítka výstupy budičů přecházejí do stavu velké impedance a při ohledávání se spínají jeden po druhém.

Nulování – obvod může být vynulován přímo při uvolnění tlačítka během protizákmitové doby nebo mezi dvěma kódy. Uvolnění tlačítka během ohledávání matice, nulování obvodu, může nastat, když:
- je tlačítko uvolněno v době, kdy jeden z výstupů budiče je ve stavu „0".
- je tlačítko uvolněno před detekcí tlačítka,
- chybí propojení v matici Z–DR při SSM=H.

Testování – vývody *19* (TP2) a *20* (TP1) jsou použity pro testování ve spojení se vstupy Z2 (vývod *5*) a *6* (vývod *6*), jak vyplývá z tab. 25.

Aktivování tlačítek – každé spojení vstupu X s výstupem DR je rozpoznáno jako správné a obvod generuje příslušný kód. Aktivování několika vstupů X současně je hodnoceno jako špatné a nenastartuje se oscilátor. Při SSM=L je každé propojení jednoho vstupu Z a výstupu DR hodnoceno jako správné a IO generuje odpovídající kód.

(Pokračování)

INZERCE

Inzerci přijímá osobně a poštou vydavatelství Magnet-Press, inzertní oddělení (inzerce ARB) Vladislavova 26, 113 66 Praha 1, tel. 26 06 51–9 linka 294. Uzávěrka tohoto čísla byla 1. 10. 1990, do kdy jsme museli obdržet úhradu za inzerát. Nepomeňte uvést prodejní cenu, jinak inzerát neuveřejníme. Text inzerátu pište čitelně, aby se předešlo chybám vznikajícím z nečitelnosti předlohy.

PRODEJ

BFG65 (130), BB405 (35), BFR90, 91, 96 (30, 31, 32), BFT66 (130), SO42 (90), TL074, 084, 082 (50, 50, 35), celá řada CMOS. D. Cienciala, 739 38 Soběšovice 181.

Prodám TRANSCEIVER fy KENWOOD TR751E-2m, TS711E-2m, 144 MHz ALL MODE 25 W VF výkon, možno instalovat do automobilu, CB stanice (~ 33 000, 48 900, cena inf.), antény, S-metry a jiná vysílací zařízení, doplňky a součásti pro VF techniku dle katalogu Conrad, zařízení ihned k dodání + záruka, prospekty zašlu, výhodné ceny inf. OK2UZF-Franta. F. Hennig, J. Jabůrkové 4, 736 01 Havířov-město.

C520D (75) od 3 ks (55). M. Lhotský, Komenského 465, 431 51 Klášterec n. Ohří.

BFR90, 91, 96 (27, 28, 32) orig, BFR90, 91, BFG65 (38, 40, 80), BF, BC, BD tranzistorů, int. obvodů CMOS, LS, TL, CA, NE a jiného rad. mat. i pas. prvků, BNC pár (100). Končím! J. Toporský, K ostrůvku 12, 794 01Krnov.

BFQ69, BFG65, BFR90, 91, 96, (100, 135, 26, 32, 38), počítač EURO PC/XT (21 900), BF961, 963, 966 (24, 28, 34), sym. členy UHF (15). J. Zavadil, Box 27, 142 00 Praha 411.

Inteligentní dekodér sat. kanálu FilmNet – Astra pro systémy MASPRO, samonastavitelný kód, předvedu i na dobírku, kvalita (3750). Ing. R. Juřík, Foltýnova 15, 635 00 Brno.

TDA5660P (290), SL1451 (890), SL1452 (890), MC14566B (120). Min. varicap ITT 1÷9 pF, BB601 (60), sat. kon. Maspro, F ≐ 1,3 dB (5700), Fuba OEK888 (6500), kon. Amstrad (kon.+fid.) (5900). F. Krount, Řepová 554, 196 00 Praha 9, tel. 687 08 70.