Učebnice Assembleru 86

Tato ucebnice mela být puvodne vydána jako skripta pro školu, na které ucím. Protože však došlo k mnoha objektivním problémum (proste nebyly peníze), nabízím je touto cestou všem, kterí programují v Turbo Pascalu. I když v soucasné dobe prevážne programuji v Delphi, znalosti z této ucebnice využívám. Pokud budete informace z tohoto dokumentu nekdo používat, cinte tak prosím s poznámkou o autorovi (copyright najdete na konci textu).

Obsah:

- **Instrukcní soubor 80[2]86**

Úvodem

Žijeme v dobe, kdy nás pocítace obklopují na každém kroku. Jejich výhodou je možnost ovlivnit své chování podle našich potreb. Proto i ten nejlepší pocítac nedokáže pracovat bez programu, který mu dává instrukce, jak se za kterých okolností chovat. Proto bude vždy nutné, aby urcitá skupina lidí byla schopna tyto programy tvorit. Myslím si, že programátori budou i pri sebedokonalejším technickém vybavení spolutvurci systému, které ulehcují lidem práci. Mezi základní znalosti každého programátora patrí alespon minimální znalost Jazyka symbolických adres, kterému ríkáme assembler. Protože tvorba složitejších programu jen v ASM86 by byla zdlouhavá, nabízí se možnost vytváret je ve vyšším programovacím jazyce, a v assembleru tvorit jen ty jeho cásti, které se casto opakují, a pritom není jejich tvorba nárocná. Tyto bloky se nejlépe programují v tzv. vloženém assembleru.

Terminologie

- **Bit** - nejnižší jednotka nesoucí informaci, muže nabývat hodnoty bud 1, nebo 0.
- **Slabika** - byte, pulslovo, to je oznacení pro 8 bitu (bity císlujeme 7-0 poporade), muže nést hodnotu císla se znaménkem (-128-127, shortint) nebo bez znaménka (0-255, byte); za pocet slabik píšeme B (kB, MB).
- **Pulslabika** - pulbyte, nibl, oznacení pro 4 bity.
- **Slovo** - word, dve slabiky, oznacení pro 16 bitu (bity císlujeme 15-0 (7-0, 7-0) poporade), muže nést hodnotu císla se znaménkem (-32768-32767, integer) nebo bez znaménka (napr. adresa, 0-65535, word).
- **Instrukce** - pokyn mikroprocesoru k vykonání nejaké cinnosti (presun, secti).
- **Program** - posloupnost instrukcí, které vedou k vykonání úlohy. Program je vetšinou uložen na disku ve forme souboru (typu EXE, COM). V nem je uložena rada císel, které znamenají jednotlivé instrukce (strojový kód). Po spuštení je bud celý program, nebo jeho cást, uložena do pameti pocítace.
- **Prekladac** - program umožnující prevést algoritmus zapsaný v textovém tvaru do strojového kódu mikroprocesoru. Ve strojovém kódu jsou jednotlivé instrukce zapsány s pomocí jedné, ci více slabik, které jsou pro každou instrukci odlišné. Jestliže tedy necháme pocítac, aby cetl instrukce z cásti pameti, kde jsou data (ne operacní kód instrukcí), dojde vetšinou k "zmrznutí" pocítace, protože data mohou obsahovat kódy znamenající instrukce nesmyslného programu.
- **Mikroprocesor** - v každém pocítaci nalezneme jeden, ci více mikroprocesoru. Jedná se o cást zajištující presuny dat v pocítaci a jejich zpracování. Uvnitr mikroprocesoru jsou vždy tyto cásti:
    - **Aritmeticko-logická jednotka (ALU)** - je to scítacka doplnená o posuvné registry a logické obvody. Vykonává operace spojené se zpracováním dat: matematické, logické a posuvy (rotace). Pocet bitu, se kterými je schopna ALU pracovat, udává, kolikabitový je celý mikroprocesor.
    - **Registry** - jsou rychlé pameti urcené pro zaznamenávání dat a adres. Jednotlivé mikroprocesory se od sebe liší poctem registru a jejich velikostí, která udává, jak velké císlo jsme schopni v nem uchovat. Jestliže registr slouží jako vstupní a výstupní pro hodnoty urcené ALU, ríkáme mu stradac.
    - **Dekodér instrukcí** - dekóduje císlo, které pro mikroprocesor znamená instrukci .
    - **Obvody rízení** - zajistí vykonání instrukce vytvorením posloupnosti impulsu, která ovlivní jednotlivé cásti procesoru tak, aby po ukoncení této posloupnosti byla instrukce vykonána. Tato posloupnost je ovlivnena mikroprogramem popisujícím jednotlivé instrukce.
- **Pamet** - je cást pocítace, kde je uložen program a data.
- **Zásobník** - je cást pameti sloužící k odkládání dat, prípadne k predávání hodnot mezi podprogramy.
- **Vstupní-výstupní porty** - za ne považujeme obvody, které jsou urcené k predání dat do, nebo z pocítace.
- **Adresa** - císlo oznacující místo slabiky v pameti, nebo vstupního/výstupního portu, se kterým chceme pracovat (tzn. kam chceme zapsat, odkud chceme císt). Maximální velikost adresy urcuje velikost adresového prostoru, tedy pocet slabik v pameti, nebo pocet vstupne/výstupních portu.
- **Systémová sbernice** - je soustava vodicu urcená k transportu dat, rídících signálu a adres mezi

mikroprocesorem, pametí a vstupne výstupními obvody. Má tyto cásti:

- **Datová sbernice** - urcená k presunum dat a kódu instrukcí.
- **Adresová sbernice** - urcená k presunum adres slabik v pameti a adres vstupne-výstupních portu.
- **Rídící sbernice** - urcená k synchronizaci všech cástí pocítace.

Programátorský model mikroprocesoru 8086

Obvod 8086 je univerzální šestnáctibitový mikroprocesor. Má šestnáctibitovou ALU, to znamená, že je schopen provádet operace s šestnáctibitovými císly. S okolím komunikuje po šestnáctibitové datové a dvacetibitové adresové sbernici.

Segmentace pameti

Vzhledem k tomu, že obvod 8086 je schopen práce s pametí o velikosti 1MB a obsahuje jen šestnáctibitové registry, je nutná tzv. segmentace pameti. Jedná se o logické delení pameti do bloku po 64kB. Tomuto bloku ríkáme segment a jeho pocátek urcuje programátor, prípadne je mu pridelen podle volného místa v pameti. Jediný požadavek na umístení pocátku segmentu je, aby jeho adresa byla násobkem šestnácti. Umístení jednotlivých slabik v segmentu urcuje offsetová cást adresy (offset). Ta urcuje, kolikátá je slabika od pocátku segmentu. Adresa se skládá ze dvou cástí: segment a offset. Obe tyto cásti jsou šestnáctibitové. Protože ale pro adresování pameti je nutné dvacet bitu, jsou za segmentovou adresu vyjádrenou binárne pridány ctyri bity s hodnotou nula (proto každý segment zacíná na násobku šestnácti). K tomuto dvacetibitovému císlu je potom pricteno šestnáctibitové císlo urcující offsetovou adresu. Takto vzniká dvacetibitové císlo znamenající skutecné umístení slabiky v pameti (fyzická adresa).

Výpocet skutecné adresy dvojkove:

```
segment:ssssssssssssssss0000
+offset:0000oooooooooooooo
        --------------------
 adresa:aaaaaaaaaaaaaaaaaaa
```

(segment jsou jednotlivé bity segmentové cásti adresy doplnené na konci o ctyri nuly, offset jsou jednotlivé bity offsetové cásti adresy doplnené na zacátku o ctyri nuly, adresa je soucet, tedy jednotlivé bity skutecné adresy)

Výpocet skutecné adresy hexadecimálne:

```
segment:ssss0
+offset:0oooo
        -----
 adresa:aaaaa
```

(segment jsou jednotlivé cifry segmentové cásti adresy doplnené na konci o jednu nulu, offset jsou jednotlivé cifry offsetové cásti adresy doplnené na zacátku o jednu nulu, adresa je soucet, tedy jednotlivé cifry skutecné adresy)

Napríklad: Místo v pameti s adresou segmentu $AB1E a offsetu $1111 má skutecnou adresu:

```
segment:AB1E0
+offset:01111
        -----
 adresa:AC2F1
```

Tento zpusob adresace umožnuje snadný prenos programu v pameti a jeho schopnost pracovat v každé její

casti. Program si pro svoji cinnost vyclení segment pro data, zásobník a strojový kód (instrukce). Na tyto bloky ukazují jednotlivé segmentové registry.

Dusledky segmentace:

- pricteme-li k segmentové casti adresy jednicku, zvýšíme skutecnou hodnotu adresy o šestnáct (což je to samé, jako bychom zvýšili offsetovou cást adresy o šestnáct)
- skutecnost, že se adresa tvorí souctem dvou císel, vede k tomu, že stejné místo v pameti mužeme urcit nekolika kombinacemi adres segmentu a offsetu dávajícími v souctu jeho fyzickou adresu

Pochopení segmentrace pameti je spíše ve znalosti dvojkové a šestnáctkové císelné soustavy.

POZOR!!! Nemeli bychom zamenovat pojmy segment a selektor. Segment urcuje jen umístení bloku pameti. Selektor je použit u vyšších typu procesoru a jedná se vlastne o poradové císlo v tabulce, která nese informace o vyclenených místech pameti a jejich vlastnostech.

Z hlediska programátora jsou nejduležitejší registry. Ty se delí na

- registry pro všeobecné použití:
  - datové registry - šestnáctibitové (všechny vyhovují definici stradace), které je možné delit na poloviny po osmi bitech, jejich použití bude probráno v dalších kapitolách:
    - **AX (AH,AL)** - stradac pro násobení a delení, vstupne-výstuní operace
    - **BX (BH,BL)** - neprímá adresace pameti (báze)
    - **CX (CH,CL)** - pocitadlo pri cyklech, posuvech a rotacích
    - **DX (DH,DL)** - neprímá adresace vstupu/výstupu
  - ukazatele a indexregistry - pro umístení adresy (offsetu):
    - **BP** - bázový registr
    - **SP** - ukazatel zásobníku
    - **DI** - adresa cíle
    - **SI** - adresa zdroje
    - **IP** - ukazatel na aktuální místo programu
- registr príznaku (**F**) - obsahuje šest bitu (indikátoru), které mikroprocesor nastavuje podle výsledku práve provedené operace, a umožnuje tak vetvit program:
  - **CF** - Carry Flag, nastaví se do log. jedna, jestliže pri práve provedené operaci došlo k prenosu z nejvyššího bitu osmibitového, nebo šestnáctibitového výsledku; tento indikátor je také využíván pri posuvech a rotacích
  - **PF** - Parity Flag, se nastaví do log. jedna, pokud dolních osm bitu výsledku obsahuje sudý pocet jednicek (a naopak)
  - **AF** - Auxiliary Carry Flag, nastaví se do log. jedna pri prenosu 1 ze spodní poloviny nižší slabiky do vyšší; využívá se v BCD aritmetice (prenos do vyššího rádu)
  - **ZF** - Zero Flag, je v log. jedna pri výsledku rovnému nule
  - **SF** - Sign Flag, je v log. jedna pri záporném výsledku
  - **OF** - Overlow Flag, nastaví se do log. jedna, jestliže došlo k aritmetickému pretecení (výsledek se nevešel do cíle)

  Tyto registry se nastavují automaticky, jestliže probehla instrukce, která je nastavuje. Registr F je doplnen i tremi rídicími registry, které ovlivnují beh programu:

  - **TF** - Trap Flag, jestliže je nastavený v log. jedna, mikroprocesor je uveden do krokovacího režimu; je tak umožneno odladení programu
  - **IF** - Interrupt Enable Flag, pri log. jedna umožnuje vykonání maskovatelného prerušení, tzn. programovou obsluhu událostí
  - **DF** - Direction Flag, je urcen k rízení smeru zpracování retezcových operací; pri log. jedna se data zpracovávají sestupne (a naopak)

Tyto tri registry muže nastavit jen programátor vhodnými instrukcemi. Mikroprocesor je sám nenastavuje. Jestliže s registrem príznaku jako s celkem pracujeme, je šestnáctibitový a má tvar: X, X, X, X, OF, DF, IF, TF, SF, ZF, X, AF, X, PF, X, CF (bity X nejsou obsazeny)

- segmentové registry - urcené pro uložení druhé cásti adresy, segmentu:
  - **DS** - segment dat (promenných)
  - **ES** - pomocný segment dat
  - **SS** - segment zásobníku
  - **CS** - segment programu

Mikroprocesor musí být schopen pracovat i se vstupy-výstupy. Umístení jednotlivých portu urcuje šestnáctibitová adresa umístená nejcasteji v registru DX. Pro programátora je duležitá i ta skutecnost, že si mikroprocesor vytvárí tzv. frontu instrukcí. Jedná se o šest slabik znamenajících nekolik instrukcí, které budou následovat po práve provádené instrukci. Tato fronta je prubežne doplnována pri operacích nezatežujících sbernice z pameti. Protože se ale jedná o deset za sebou jdoucích slabik v pameti, je pri instrukcích skoku v pameti vyprázdnena. Z tohoto duvodu je vhodné, aby program obsahoval co nejmenší pocet skoku. Proto je v poslední dobe kladen duraz na programovací jazyky, které podporují tzv. strukturované programování bez nepodmínených skoku. Mezi ne (cástecne) patrí Turbo Pascal a C. Je jasné, že programovací jazyk Basic se v tomto smyslu k mikroprocesoru nechová moc šetrne a zpomaluje tak beh programu. Mikroprocesor 80286 je strukturou i vlastnostmi podobný 8086. Je schopen pracovat ve dvou režimech. V základním reálném témer presne simuluje obvod 8086. Presto v tomto režimu prináší nekterá rozšírení pro nekteré instrukce. Pokud v následujícím výkladu použiji rozšírení instrukcí pro 80286, uvedu to poznámkou [286]. Zdrojový text programu sestaveného i s pomocí instrukcí 80286 ve vkládaném assembleru stací na prvním rádku (pred uses) oznacit direktivou {$G+}.

Vkládaný assembler v jazyce Turbo Pascal

Vzhledem k jednoduchosti a názornosti se programovací jazyk Turbo Pascal vyucuje na školách. My se budeme zabývat tzv. vkládaným assemblerem. Jeho znalost umožní zrychlit námi psané programy, a pritom využívat výhod Pascalu ve snadném zápisu algoritmu. Vkládaný assembler je blok v programu psaném v jazyce Pascal. Tento blok zacíná klícovým slovem Asm a koncí end. Rádky programu ve vkládaném assembleru se necíslují a nemusí koncit stredníkem v prípade, že na jednom rádku není více jak jedna instrukce (pri více jak jedné instrukci musíme instrukce stredníkem oddelit). Komentáre se píší do složených závorek, nesmejí však být uvnitr oznacení instrukce. Ve vloženém assembleru mužeme menit obsahy registru AX, BX, CX, DX, SI, DI, ES, F. Pred návratem z bloku asm musíme obnovit hodnoty v registrech BP,SP, SS, DS.

Instrukce presunu dat

Každý program musí být schopen presunu dat a to mezi registry, registry a pametí, registry a vstupy/výstupy. Pri této operaci si musíme vždy uvedomit, kolikabitové císlo presouváme. Pocet bitu je vetšinou specifikován jménem použitého registru (osmibitové - AH, AL, BH, BL, . . ., šestnáctibitové - AX, BX, BP, DI, ES, DS . . .). V prípade, že používáme jen pamet, specifikuje pocet bitu pro operaci oznacení:

- **BYTE PTR** oznacení pam. místa - specifikuje slabiku
- **WORD PTR** oznacení pam. místa - specifikuje slovo

Presuny registr - registr, registr - pamet

Všechny presuny tohoto typu provedeme univerzální instrukcí:

- **MOV** cíl, zdroj - do cíle presun ze zdroje (registr - registr, registr - pamet, registr - hodnota, pamet - hodnota, seg. registr - registr, seg.registr - pamet)

Použití této instrukce demonstruje príklad:

```
uses crt;
var  slovo:word;  {v pameti rezervuj 16 bitu a oznac je slovo}
    slabika:byte;{v pameti rezervuj 8 bitu a oznac je slabika}
begin
 asm
  MOV AL,10       {do registru AL dosad 8 bitu, hodnotu 10}
  MOV slabika,AL  {do pameti na místo ozn. slabika dosad obsah AL}
  MOV BX,10       {do registru BX (16 bitový) dosad 10}
  MOV slovo, BX   {do pameti na místo ozn. slovo dosad 16 bitu BX}
 end;
 writeln (slabika,' ',slovo);
 readkey;
end.
```

Tento program má po prekladu na místech promenných v bloku asm oznacení pametového místa, které pro ne bylo vycleneno. Místo pro promenné je vždy v segmentu globálních promenných. Segmentová adresa tohoto bloku je vždy umístena v registru DS. To, že DS ukazuje na segment dat programu, muže vést k chybe, která spocívá v jeho zmene a následném ctení z globálních promenných. Takže pozor! Po zmene registru DS je práce s globálními promennými nemožná, protože jsme si k nim urízli cestu. Do segmentových registru nejde dosadit hodnota prímo. Tu nejrychleji dosadíme tak, že ji vložíme do nekterého univerzáního registru a z nej teprve do segmentového registru (napríklad **MOV** AX,adresa; **MOV** ES,AX).

Metody adresace

Místo (offset) v pameti oznacuje vždy urcitá hodnota zapsaná v hranatých závorkách. Instrukce **MOV BYTE PTR** ES:[$100F], 10 znamená: na adresu slabiky offset 100F ($ oznacuje použití hexadecimální soustavy) v segmentu urceném adresou v ES, dosad hodnotu 10. Jestliže segment nespecifikujeme oznacením a dvojteckou, vztahuje se adresa k segmentu v DS. V praxi by tato metoda omezovala programátora v rozletu. Proto ASM86 umožnuje i další metody adresace. Ale poporade . . .

- **Prímá adresa**
  MOV AH, ES:[$1A40] - do registru AH predej 8 bitu z adresy urcené ES a císlem
  Tuto metodu použijeme, jestliže predem víme adresu hledaného místa v pameti. Na pomoc v Turbo Pascalu jsou operátory:
  - **OFFSET** promenná - vrací offsetovou adresu promenné
  - **SEG** promenná - vrací segmentovou adresu promenné (pro globální promenné vrací vždy obsah DS)

  Jejich použití umožní zjistit adresu promenných deklarovaných v cásti var (const . . .).

  Príklad:

```
var promenna: byte;
begin
 asm
  MOV BYTE PTR [offset promenna], 10 {na adresu slabiky promenné dosad 10}
 end;
end.
```

  Segmentová adresa se v tomto príkladu nemusí urcit. Je v DS, a ten se nemusí uvádet. Prekladac Pascalu tuto metodu používá i pro naše globální promenné. Pri prekladu je totiž každé promenné prideleno místo v pameti s pevnou offsetovou adresou (takže zápis **OFFSET** promenná nese práve tuto adresu). Specifikace, jestli se jedná o slabiku, nebo slovo, je nutná, protože jinak by procesor nevedel, jestli má císlem obsadit jednu, nebo dve slabiky.

- **Neprímá adresa**
  MOV AH, ES:[BX] - do registru AH predej obsah pam. místa specifikovaného adresou v BX
  Pozor! Do registru AH je uložen obsah v pameti na adrese v BX, ne obsah registru BX. Offsetová cást adresy je uložena v nekterém z adresových registru BX, BP, SP, SI, DI. Vzhledem k tomu, že obsah techto registru mužeme menit, použijeme tuto metodu v prípade pohybu po pameti.
  Príklad:

```
var promenna: byte;
begin
 asm
  MOV BX, offset promenna {do BX dosad adresu promenné}
  MOV BYTE PTR [BX], 10   {na její adresu dosad hodnotu 10}
 end;
end.
```

- **Bázová adresa**
  MOV AH, [BX + adresa] - k registru BX pricti konstantu adresa, výsledná hodnota je adresou odkud se má nacíst do registru AH
  Bázová adresa se tvorí s pomocí obsahu jednoho z bázových registru BP, BX. Výraz v závorce se vyhodnotí, pritom oznacení registru zastupuje jejich obsahy. Tento druh adresy používáme pri zjištování hodnot parametru urcených pro podprogramy (prípadne k prístupu k lokálním promenným).
- **Indexovaná adresa**
  MOV AH, ES:[adresa + SI] <=> (je shodné) **MOV** AH, adresa[SI] - registr SI secti s konstantou adresa, výsledek je hodnota adresy offsetu do pameti
  Tento zpusob adresace je obdobou predchozí tvorby adresy. Používá se však pri práci s bloky v pameti. Zde jsou k dispozici indexové registry SI, DI.
  Príklad:

```
var pole: array [0..9] of byte;
begin
 asm
  MOV SI, 0                  {nuluj registr SI}
  MOV BYTE PTR offset pole[SI], 10{adr. pole secti s SI a dosad 10}
 end;
end.
```

  Program dosadí na první místo pole hodnotu. Protože registr SI mužeme zvyšovat, budeme tímto zpusobem realizovat pohyb v poli.

- **Kombinovaná adresa báze + index**
  MOV AH, [BX + SI] <=> **MOV** AH, [BX][SI] - obsahy registru BX a SI secti, výsledek je hodnota offsetu odkud se má císt
  Kombinovaná adresa umožnuje pracovat s adresou, která se skládá ze souctu dvou registru (jednoho bázového BX, BP a jednoho indexového SI, DI).
  Príklad:

```
var pole: array [0..9] of byte;
begin
 asm
  MOV BX, offset pole      {do registru BX dosad adresu pole}
  MOV SI, 0              {do registru SI dosad 0}
  MOV BYTE PTR [BX][SI], 10 {na první prvek v poli ulož 10}
 end;
end.
```

- **Kombinovaná adresa prímá + báze + index**
  MOV AH, [adresa + BX + SI] <=> **MOV** AH, adresa[BX][SI] - secti registry BX, SI a pricti hodnotu adresa, výsledek je hodnota offsetu
  Tuto adresaci použijeme napríklad pri práci s hlavickovými soubory. Bázový registr nastavíme na

pocátek bloku pameti vycleneného k uložení souboru. Indexový registr vynulujeme. Konstantní hodnota (adresa) muže být rovna délce hlavicky. Zvyšováním hodnoty v indexovém registru se pohybujeme v datech hlavickového souboru. Další možné použití této adresace je pri pohybu v dvourozmerných polích. Hodnoty v obou registrech jsou indexy pole. Konstantní adresa je adresou pocátku pole.

Prefix preskocení

V assembleru mikroprocesoru 8086 se objevuje i nový výraz. Prefix znamená urcitou specifikaci pro následující instrukci. Zatím jsme si ukázali, jak zmenit specifikaci segmentového registru adresy s pomocí jeho oznacení a dvojtecky. Dalším zpusobem je použití prefixu zmeny segmentu: **SEGDS**, **SEGES**, **SEGCS**, **SEGSS**. Tato oznacení jsou prefixy preskocení (zmeny segmentu) pro jednotlivé segmentové registry. Napríklad: **MOV** AX, ES:[BX] je stejné, jako bychom použili **SEGES MOV** AX, [BX] (i když zápis je ruzný, kód programu bude po prekladu stejný).

Práce se zásobníkem

Zásobník je cást v pameti pocítace vyhrazená k odkládání dat. Je organizovaná tak, že data, která jsou uložena naposledy, vyjímáme jako první. Na vrchol zásobníku ukazují adresy uložené v registrech SS a SP (prípadne BP). Pridáváním dat do zásobníku se offset v SP automaticky snižuje o dve (a naopak). Musíme si tedy uvedomit, že do zásobníku mužeme odkládat jen šestnáctibitová data. Pro práci se zásobníkem slouží instrukce:

- **PUSH** zdroj - do zásobníku ulož obsah zdroje (registr, pamet, [286] hodnota)
- **POP** cíl - ze zásobníku dosad do cíle (registr, pamet)
- **PUSHA** - [286] do zásobníku ulož postupne registry AX, CX, DX, BX, SP, BP, SI, DI
- **POPA** - [286] ze zásobníku dosad zpet registry uložené instrukcí PUSHA
- **PUSHF** - do zásobníku ulož obsah registru F v šestnáctibitovém tvaru
- **POPF** - hodnotou ze zásobníku obsad registr F

Príklad:

```
var promenna:word;
begin
 promenna:=10;      {do pameti na adresu promenné dosad 10}
 asm
  MOV AX, promenna  {obsah promenné dosad do registru AX}
  MOV BX,$BBBB      {do regisru BX dosad císlo}
  PUSH AX           {ulož obsah AX}
  PUSH BX           {ulož obsah BX}
  MOV AX,$AAAA      {prepiš obsah AX}
  MOV BX,$CCCC      {prepiš obsah BX}
  POP BX            {obnov obsah BX}
  POP AX            {obnov obsah AX}
  MOV promenna, AX  {vrat obsah AX do promenné}
 end;
end.
```

Tento program naznacuje postup ukládání a vybírání dat do a ze zásobníku. V zásobníku jsou uloženy i lokální promenné procedur a funkcí. Jsou zde i parametry, kterými je podprogram volaný. (Proto lokální promenné NEMAJÍ segmentovou adresu v DS.) Obcas potrebuje programátor uložit registr príznaku F, aby ho pozdeji mohl obnovit do puvodního stavu. K tomu požíváme instrukci **PUSHF** (pro uložení) a **POPF** (pro obnovení). Jestliže ve vkládaném assembleru chceme menit nekterý ze "zakázaných" registru (DS, BP), mužeme si jeho obsah uložit do zásobníku. Podmínkou je ale to, že nezmeníme registry SS, SP. Tím bychom si podrízli vetev pod sebou. Další možné použití zásobníku je pri práci s cástí pameti, ve které máme

pole slov (šestnáctibitových dat). Nasmerováním vrcholu zásobníku (SS:SP) na konec tohoto pole mužeme instrukcemi **PUSH** a **POP** s tímto polem pracovat. Pritom se bude automaticky zvyšovat a snižovat adresa. Pozor ale, obsahy SS a SP je nutné zase uschovat, nejlépe do pameti na místa promenných. V tom prípade ale nemužeme menit registr DS (ES).

Presuny vstup-výstup - registr

Každý se nekdy pokusíme zapsat na port a císt z nej. Je dobré si uvedomit, že mužeme zapisovat osm i šestnáct bitu. Každý port, stejne jako slabika v pameti, má svojí adresu. Pri zápisu šestnácti bitu zapisujeme tedy i na port s adresou o jednu vyšší. Práci s porty provedeme instrukcemi:

- **OUT** adresa portu, zdroj - pro zápis na port (AL, AX-> port)
- **IN** cíl, adresa portu - pro ctení z portu (port-> AL, AX)

Data se ctou, nebo zapisují z (do) registru AL (osmibitový prístup), AX (šestnáctibitový prístup). Adresu portu specifikuje bud prímo adresa (**IN** AL, $0F) pri adrese osmibitové (spodních 256 portu), nebo registr DX, ve kterém je šestnáctibitová adresa (**MOV** DX, $F10; **OUT** DX, AL).

Další presuny

Mezi presuny dat patrí i:

- **XCHG** cíl, zdroj - vzájemná výmena hodnot zdroje a cíle (pamet - registr, registr - registr)
- **LAHF** - do registru AH dosad nižší slabiku registru príznaku F
- **SAHF** - z registru AH dosad do nižší slabiky registru príznaku F
- **XLAT** - do AL dosad obsah slabiky v pameti s adresou v DS:[BX + AL] (práce s tabulkou) a nekteré retezcové instrukce o kterých bude rec pozdeji.

Instrukce dosazení adresy

I když jsme si již popsali, jak dosadit hodnotu adresy do nekterého z adresových registru, nebyly možnosti ješte vycerpány. Nejjednodušší je použití instrukce:

- **LEA** adresový registr, pamet - do adresového registru dosad adresu offsetu pameti

Pamet je v tomto prípade oznacena jako v instrukci **MOV**. Instrukce **LEA** BX, **BYTE PTR** [$FF00] a **MOV** BX, $FF00 jsou ekvivalentní. Protože druhá instrukce je jednodušší, nemela by instrukce **LEA** význam. Proto ji casteji použijeme pri hledání hodnoty kombinované adresy (**LEA** DI, 100[BX][SI] - secte registry s císlem 100 a dosadí výsledek do DI). Pro nás má význam i ve vkládaném assembleru. Zápis **LEA** BX, promenná je jednodušší než **MOV** BX, offset promenná (i když instrukce vykonají stejnou práci).

Príklad:

```
var pole: array [0..9] of byte;
begin
 asm
  LEA BX, pole        {do registru BX dosad adresu pole}
  MOV BYTE PTR [BX],10 {na první místo v poli napiš 10}
 end;
end.
```

Zatím jsme ovlivnovali jen registry s offsetem. Prestože bychom byli schopni dosadit i segment, bylo by nutné použít nejméne tri instrukce (nezapomente, že **MOV** neumí dosadit hodnotu do segmentového registru prímo). Abychom pochopili úspornejší instrukci, musíme si zopakovat pojem ukazatel.

Ukazatel

Je typ promenné, který nese celou adresu urcitého místa v pameti. S pomocí techto promenných mužeme potom dosazovat hodnoty na místa, kam ukazují. Casteji myslíme oznacením ukazatel práve tyto promenné.

Príklad:

```
var cislo:byte;      {vyclen v pameti slabiku, oznac jí císlo}
   ukazatel:^byte;  {vyclen v pameti ctyri slabiky, které ponesou}
                    {adresu na promennou typu byte, oznac je ukazatel}
begin
 ukazatel:=@cislo;   {ukazateli prirad adresu promenné císlo}
 ukazatel^:=10;      {na místo kam smeruje ukazatel zapiš 10}
 writeln ('Hodnota promenné císlo:',cislo,'=',ukazatel^); {vypiš}
end.
```

Krome ukazatelu na daný typ existují i ukazatele obecne (typ pointer). Tyto typy jsou pro nás duležité. Ctyri slabiky, které jsou pro promennou tohoto typu vycleneny, nesou totiž segment i offset adresy, kam ukazatel smeruje. V assembleru existují dve instrukce, které jsou schopny adresy uložené v ukazateli dosadit do registru segmentu i offsetu:

- **LES** registr, ukazatel - do ES dosad segment a registru offset adresy smeru ukazatele
- **LDS** registr, ukazatel - do DS dosad segment a registru offset adresy smeru ukazatele

Príklad:

```
var promenna: byte;   {v pameti vyclen slabiku s oznacením promenná}
   ukazatel: poiter; {v pameti vyclen ctyri slabiky pro ukazatel}
begin
 ukazatel:=@promenna; {nasmeruj ukazatele na promennou}
 asm
  LES BX, ukazatel    {nastav ES:BX na adresu promenné}
  SEGES MOV BYTE PTR [BX],10 {zapiš na tuto adresu}
 end; writeln (promenna);    {vypiš obsah promenné}
end.
```

Aritmetické instrukce

Programátor pri své cinnosti potrebuje nejen presuny dat. V každém programu jsou nutné i výpocty a to s bežnými daty, nebo s adresami. Ty se v assembleru provádejí jen s celými císly. Operace s desetinnými císly jsou zdlouhavé, i když jsou proveditelné pomocí urcitých algoritmu. ASM86 pro ne ale nemá instrukce. Vetšina matematických operací se provádí s císly v registrech nebo v pameti. Oznacení operandu je shodné jako pri presunech. Zároven tyto instrukce nastavují indikátory registru F. Umožní tak vetvit program. Informace o nastavovaných indikátorech najdeme v tabulce instrukcí (**+**).

Scítání

Pri tvorbe programu si musíme ujasnit, jestli chceme k cílovému místu pricíst 1, nebo jiné císlo. Podle toho volíme instrukci:

- **INC** cíl - k cíli pricti jedna (registr, pamet)
- **ADD** cíl, zdroj - k cíli pricti zdroj (registr - hodnota, pamet -hodnota, registr - registr, pamet - registr, registr - pamet)

- **ADC** cíl, zdroj - stejne jako **ADD**, ale pricti i bit CF (prenos)

Príklady:

INC AX - pricti k registru AX hodnotu 1
INC WORD PTR [BX] - pricti k slovu na adrese urcené DS:BX hodnotu 1
INC BYTE PTR CS:[adresa] - pricti k slabice na adrese urcené CS:adresa (konstantní) 1
SEGES INC BYTE [DI + 2] - pricti k slabice na adrese ES:DI + 2 hodnotu 1
ADD AX, BX - ke slovu v registru AX pricti obsah registru BX (slovo)
ADD AH, 8 - k slabice v registru AH pricti císlo 8}
SEGCS ADD DX, **WORD PTR** [BX] - k registru DX pricti slovo na adrese CS:BX
ADD promenna, 5 - k deklarované promenné pricti 5
ADD BYTE PTR [SI], 30 - k slabice na adrese DS:SI pricti 30}
ADD BYTE PTR ES:[BP], AL - k slabice na adrese ES:BP pricti obsah registru AL

Pokud pri techto operacích dojde k preplnení cíle, nastaví se registr OF do log. 1. Aby pri odladování vašich programu nedošlo ke zbytecným hádkám s prekladacem, uvedomte si, že zdroj i cíl musí mít stejný pocet bitu (tzn. 8, nebo 16).

Odcítání

Instrukce sloužící k odcítání jsou zápisem operandu shodné s instrukcemi pro scítání. Proto si uvedeme jen jejich seznam:

- **DEC** cíl - d cíle odecti 1 (registr, pamet)
- **SUB** cíl, zdroj - od cíle odecti zdroj (registr - hodnota, pamet - hodnota, registr - registr, pamet - registr, registr - pamet)
- **SBB** cíl, zdroj - stejne jako SUB, ale odecti i bit CF Príklady by byly shodné se scítáním.

Presto jsou zde specifické instrukce:

- **NEG** cíl - otoc znaménko v cíli (registr, pamet)
- **CMP** cíl, zdroj - odecti bez zmeny cíle, nastav jen registr F (registr - hodnota, pamet - hodnota, registr - registr, pamet - registr, registr - pamet)

Instrukce **CMP** porovnává dve císla odectením. Protože ale nedojde k jejich zmene, použijeme tuto instrukci pred vetvením programu. Za **CMP** totiž vetšinou následu+jí instrukce skoku závislé na stavu príznaku registru F.

Príklad:

```
uses crt;
var a,b,s,r:integer;
begin
 clrscr;       {vymaž obrazovku}
 write ('a=');
 readln(a);    {vstup hodnoty a}
 write ('b=');
 readln(b);    {vstup hodnoty b}
 asm           {zacátek bloku asm}
  MOV AX, a    {do AX vlož hodnotu promenné a (z pameti)}
  ADD AX,b     {k AX pricti hodnotu promenné b}
  MOV s, AX    {do promenné s vlož soucet z registru AX}
  MOV AX,a     {znovu naber a}
  SUB AX,b     {odecti od AX hodnotu b}
```

```
  MOV r,AX      {do promenné r vlož rozdíl z registru AX}
  INC a         {k a pricti 1}
  DEC b         {od b odecti 1}
 end;           {konec bloku asm}
 writeln ('a+b=',s,' a-b=',r);{vypiš obsahy promenných}
 writeln ('a+1=',a,' b-1=',b);
end.
```

Uvedený príklad ukazuje nejjednodušší použití instrukcí **ADD**, **SUB**, **INC**, **DEC**. Všimnete si, že se zápisy adres promenných si nemusí programátor ani moc lámat hlavu. V tom mu totiž pomáhá prekladac Pascalu.

Násobení

I když programátori neradi používají instrukce násobení a delení pro jejich dlouhou dobu provádení (na procesoru 8086, u jiných procesoru je už rychlé), ASM86 je má. Nekdy dokonce neexistuje jiná možnost než je použít. I tyto operace jsou definovány jen na celých císlech. Rozlišujeme také, jestli je provádíme se znaménkem, nebo bez znaménka.

- **MUL** zdroj - registr AL vynásob se zdrojem (osmibitový registr, nebo pamet) a výsledek zapiš do registru AX (osmibitové násobení).
- **MUL** zdroj - registr AX vynásob se zdrojem (šestnáctibitový registr, nebo pamet) a výsledek (32 bitu) zapiš do registrového páru DX,AX za sebou (šestnáctibitové násobení).
- **IMUL** zdroj - jako **MUL** ale násobení se znaménkem **IMUL** cíl,[zdroj,]konstanta - [286], do cíle vlož soucin zdroje a konstanty (šestnáctibitový registr - šestnáctibitový registr - hodnota, šestnáctibitový registr - slovo v pameti - hodnota, šestnáctibitový registr - osmibitová hodnota, to znamená cíl := zdroj * konstanta, nebo cíl := cíl * konstanta)

POZOR!, o kolikabitové násobení se jedná urcuje oznacení místa zdroje.

Delení

Tato operace je jednou z nejzdlouhavejších. Její provádení trvá (na 8086) až 190 period hodin (scítání trvá kolem 3 period). Jeho výhodou je ale to, že je možné zjistit jak výsledek po celocíselném delení (DIV), tak i zbytek po celocíselném delení (MOD). A to všechno jen jednou instrukcí.

- **DIV** zdroj - registr AX vydel zdrojem (osmibitový registr, nebo pamet) a podíl ulož do AL, zbytek po delení ulož do AH (Osmibitové delení)
- **DIV** zdroj - dvojslovo v registrech DX, AX vydel zdrojem (šestnáctibitový registr, nebo pamet) a podíl ulož do AX, zbytek po delení ulož do DX (Šestnáctibitové delení)
- **IDIV** zdroj - jako **DIV** ale delení se znaménkem Použití techto instrukcí je podobné jako násobení. Program si musíme ošetrit tak, aby nemohlo dojít k delení nulou. Jestliže k nemu presto dojde, procesor zavolá prerušení **INT** 0.

Príklad:

```
uses crt;
var a,b,d,z:byte;
   s:word;
begin
 clrscr;
 write ('a=');
 readln (a);
 write ('b=');
 readln (b);
```

```
asm
 MOV AL,a {do AL vlož hodnotu a}
 MUL b    {vynásob hodnotou b (v pameti)}
 MOV s,AX {do promenné s vlož soucin z registru AX}
 MOV AH,0 {nuluj AH (císlo je jen 8 bitové)}
 MOV AL,a {do AL vlož hodnotu a}
 DIV b    {vydel promennou b}
 MOV d,AL {výsledek vlož do promenné d}
 MOV z,AH {zbytek po delení vlož do promenné z}
end;
writeln ('a*b=',s);
writeln ('a div b=',d,' a mod b=',z);
readkey;
end.
```

Zmena poctu bitu

Casto potrebujeme opravit šestnáctibitové císlo na osmibitové a naopak. Pri této zmene muže ale dojít ke ztráte informace v prípade úbytku bitu. Prevod císel bez znaménka provedeme nejjednodušeji využitím pulení registru.

- **Slabika -> Slovo**
  Do šestnáctibitového registru nacteme do dolní poloviny slabiku. Horní polovinu nulujeme. Slovo potom nacteme ze všech šestnácti bitu:
  ```
  var b:byte;
     w:word;
  begin
   b:=10;
   asm
    MOV AL, b    {do AL osm bitu z promenné b}
    MOV AH, 0    {nuluj AH}
    MOV w, AX    {do promenné w vlož všech šestnáct bitu}
   end;
  end.
  ```
- **Slovo-> Slabika**
  Operace je opacná. Šestnáctibitové císlo vložíme do celého šestnáctibitového registru. Do slabiky potom vložíme jen spodních osm bitu. Ale pozor, tady muže dojít ke ztráte bitu v horních osmi bitech. Protože úprava císel se znaménky by byla složitá, prichází opet na pomoc ASM86 s instrukcemi:
  - **CBW** - preved obsah AL do AX se zachováním znaménka
  - **CWD** - preved obsah AX do DX, AX (32 bitu) se zachováním znaménka

Práce s císly v kódu BCD

Císla v BCD kódu mohou být uložena v techto formátech:

- **Nezhuštený tvar**
  V jedné slabice je uložena jedna císlice v BCD kódu. Má hodnotu 0-9 a obsazuje tedy jen spodní 4 bity. Horní polovina slabiky je nulová (to se doporucuje pro operace násobení a delení, pro scítání a odcítání muže mít libovolný obsah). Tento tvar je vhodný pro prevod do kódu ASCII. Stací jen k slabice pricíst císlo 48 (logický soucet s císlem $30).
- **Zhuštený tvar**
  V jedné slabice jsou uloženy dve BCD císlice. Spodní 4 bity nesou hodnotu nižšího rádu (jednotky), horní 4 nesou hodnotu vyššího rádu (desítky). Do slabiky jde tedy uložit císlo v rozsahu 0-99. ASM86

nepodporuje prímo matematické operace s takto kódovanými císly. Presto obsahuje instrukce pro jejich úpravu po provedení bežných operací urcených pro císla v prirozeném dvojkovém kódu (obycejné dvojkove uložené císlo). V ASM86 nejdeme i instrukce, které pro tyto operace císla v BCD kódu pripraví. Jedná se o instrukce: **AAA**, **AAD**, **AAM**, **AAS**, **DAA**, **DAS** (bližší informace v tabulce instrukcí).

Instrukce logických operací

Logické instrukce jsou jednou z dobrých pomucek programátoru. ASM86 je schopen provádet všechny bežné logické operace, a to se slovem nebo slabikou. Chybí zde tedy instrukce pro jednotlivé bity. Ty však volbou vhodných algoritmu mužeme lehce nahradit.

- **NOT** zdroj - neguj všechny bity zdroje
- **AND** zdroj, cíl - logický soucin zdroje s cílem ulož do zdroje (registr - hodnota, pamet - hodnota, registr - registr, pamet - registr, registr - pamet)
- **TEST** zdroj, cíl - logický soucin zdroje s cílem, ale nastav jen registr príznaku F (registr - hodnota, pamet - hodnota, registr - registr, pamet - registr, registr - pamet)
- **OR** zdroj, cíl - logický soucet zdroje s cílem ulož do zdroje (registr - hodnota, pamet - hodnota, registr - registr, pamet - registr, registr - pamet)
- **XOR** zdroj, cíl - logický vylucovací soucet zdroje s cílem ulož do zdroje (registr - hodnota, pamet - hodnota, registr - registr, pamet - registr, registr - pamet)

Kolikabitová operace je, urcuje opet specifikace zdroje a cíle. Instrukci **TEST** použijeme k nastavení príznakového registru, a tak mužeme vetvit program, aniž bychom ovlivnili hodnoty zdroje a cíle.

Použití logických operací

Vymaskování slabiky nebo slova
Casto potrebuje programátor nastavit nekteré bity slabiky, nebo slova do hodnoty log. 1, nebo 0. K tomu mu velmi dobre poslouží práve logické operace AND nebo OR. Máme-li slabiku ve tvaru XXXXAXXX v registru AL a chceme, aby bity X mely hodnotu 0 a hodnota bitu A zustala zachována, provedeme instrukci **AND** AL, $08 (=00001000). Máme-li slabiku ve tvaru XXXXAXXX v registru AL a chceme, aby bity X mely hodnotu 1 a hodnota bitu A zustala zachována, provedeme instrukci **OR** AL, $F7 (=11110111).

Nulování registru
Zajímavejší než instrukce **MOV** registr,0 je nulovat pomocí **XOR** registr, registr. Efekt je stejný, doba vykonání operace je kratší.

Zjištení zbytku po celocíselném delení mocninami 2
Kdybychom vždy, když chceme zjistit zbytek po delení mocninami 2 (a ten casto potrebujeme) používali instrukci **DIV**, program bychom zdržovali. Stací si jen uvedomit, že mužeme zjistit hodnotu bitu v rádech za log. 1 v binárním tvaru delence. Chceme-li zjistit zbytek po celocíselném delení 2 (sudé, liché císlo) císla v registru AL, stací jen použít instrukci **AND** AL,1. V registru AL je potom jen bud 1 (liché císlo), nebo 0 (sudé císlo). Pro lepší orientaci poslouží prehled:

- **AND** AL, 1 (1 = 00000001) -> AL := AL mod 2 (2 = 00000010)
- **AND** AL, 3 (3 = 00000011) -> AL := AL mod 4 (4 = 00000100)
- **AND** AL, 7 (7 = 00000111) -> AL := AL mod 8 (8 = 00001000)
- **AND** AL, 15 (15 = 00001111) -> AL := AL mod 16(16 = 00010000)

Prevod císla v nezhušteném BCD na ASCII
Velmi jednoduchým prostredkem, jak prevést císlo v rozsahu 0-9 do hodnoty jeho znaku v tabulce ASCII, je logický soucet s císlem $30 (to je stejné jako prictení 48). Použitím této úpravy císel v kódu BCD je zobrazení i velkých císel jednoduché.

Príklad:

```
var slabika:byte;
   znak:char;
begin
 repeat
  readln (slabika);
 until slabika in [0..9];
 asm
  MOV AL, slabika {do registru AL predej hodnotu slabiky}
  OR AL, $30     {preved na ASCII}
  MOV znak, AL    {do promenné znak predej ASCII hodnotu císla}
 end;
 writeln (znak);
end.
```

V príkladu je nactené císlo z intervalu 0..9 prevedeno do ASCII s pomocí log. instrukce **OR**.

Pred dalším príkladem si musíme vysvetlit, jak ukládá Pascal retezce (typu string). Za retezec je zde považováno pole slabik, které má na prvním míste délku retezce a na dalších místech jsou kódy ASCII zapsaných znaku. Informace o délce retezce je duležitá pro jeho správné zobrazení. Ten proto nemusí obsahovat speciální ukoncovací znak.

Príklad:

```
var slovo:string;
   i:byte;
begin
 for i:=0 to 9 do
 asm
  MOV DI, OFFSET slovo {do registru DI ulož adresu promenné slovo}
  INC DI             {posun se až za slabiku délky retezce}
  XOR AH,AH          {nuluj AH} MOV AL,i {do AL vlož krok i}
  ADD DI,AX          {pricti krok k adrese (posuv po retezci)}
  OR AL,$30          {preved obsah AL na ASCII znak}
  MOV [DI],AL        {presun znak do retezce}
  INC BYTE PTR [OFFSET slovo]{zvyš délku retezce}
 end;
 writeln (slovo);
 readln;
end.
```

Tento príklad vytvorí slovo typu string s císly od 0 do 9. To, že zatím nevíme, jak se v ASM86 tvorí cykly, není na závadu. Proste si pomužeme znalostmi z Pascalu.

Kódování

Každý rád chrání svá data pred neoprávneným prístupem kódováním. K tomu dobre slouží logická operace XOR. Postup kódování naznacuje postup. Provedeme-li operaci XOR s konstantou a kódovaným císlem, získáme kódované císlo. Pokud s kódovaným císlem provedeme opet XOR se stejnou konstantou, získáme zpet puvodní císlo. Císla kódovaná pridáme do EXE souboru programu. Pred jejich použitím je dekódujeme. Protože tato císla mohou nést napr. jméno autora (v ASCII), je jméno pro bežného uživatele po zakódování necitelné (a tedy lehce neprepsatelné v souboru EXE). Pozor! Hodnota konstanty musí být pri kódování i dekódování stejná. Tento postup mužeme libovolne pozmenovat podle úrovne našich znalostí (napr. xorovat první znak s druhým, druhý s tretím, . . .).

Instrukce posuvu a rotací

Tyto instrukce jsou dobrým pomocníkem každému, kdo je umí používat. Jedná se o bitový posuv uvnitr slabiky, nebo slova. Pocet bitu posuvu je specifikován použitým registrem, nebo oznacením pametového místa.

Posuvy:

- **SHL** cíl, pocet <=> SAL cíl, pocet - v cíli posun tak, že nejnižší bit nahradíš nulou, ostatní presun z nižšího místa o jedno výše, nejvyšší bit presun do registru CF (registr - CL (který nese pocet kroku posuvu), registr - 1, [286] registr - pocet kroku posuvu)
- **SHR** cíl, pocet - v cíli posun tak, že nejvyšší bit nahradíš nulou, ostatní presun z vyššího místa na nižší, nejnižší bit presun do registru CF (registr - CL (který nese pocet kroku posuvu), registr - 1, [286] registr - pocet kroku posuvu)
- **SAR** cíl, pocet - v cíli posun tak, že nejvyšší bit nezmeníš a kopíruj ho do nižšího bitu, ostatní presun z vyššího místa na nižší, nejnižší bit presun do registru CF (registr - CL (který nese pocet kroku posuvu), registr - 1, [286] registr - pocet kroku posuvu)

Rotace:

- **ROL** cíl, pocet - v cíli posun tak, že každý nižší bit kopíruj do vyššího, nejvyšší kopíruj na místo nejnižšího a do registru CF (registr - CL (který nese pocet kroku posuvu), registr - 1, [286] registr - pocet kroku posuvu)
- **ROR** cíl, pocet - v cíli posun tak, že každý vyšší bit kopíruj do nižšího, nejnižší kopíruj na místo nejvyššího a do registru CF (registr - CL (který nese pocet kroku posuvu), registr - 1, [286] registr - pocet kroku posuvu)
- **RCL** cíl, pocet - v cíli posun tak, že každý nižší bit kopíruj do vyššího, nejvyšší kopíruj do registru CF, obsah CF prenes na místo nejnižšího (registr - CL (který nese pocet kroku posuvu), registr - 1, [286] registr - pocet kroku posuvu)
- **RCR** cíl, pocet - v cíli posun tak, že každý vyšší bit kopíruj do nižšího, nejnižší kopíruj do registru CF, obsah CF prenes na místo nejvyššího (registr - CL (který nese pocet kroku posuvu), registr - 1, [286] registr - pocet kroku posuvu)

Použití posuvu a rotací

Kontrola jednotlivých bitu
Jestliže potrebujeme zkontrolovat, jakou hodnotu nekterý z bitu nese, stací slovo nebo slabiku rotovat pres registr CF. Hodnotu, kterou bit nese, potom zjistíme kontrolou registru CF.

Tvorba masky
Jestliže nevíme, jak vytvorit slabiku nebo slovo pro vymaskování, použijeme instrukci posuvu. **MOV** AL, 1; **SHL** AL, 3. Takto získáme slabiku s nastaveným bitem na ctvrtém míste (00001000).

Celocíselné delení mocninou 2 a násobení konstantou
Je to nejduležitejší použití posuvu. Vychází z faktu, že bitový posuv císla doleva o jeden krok je stejný, jako bychom císlo vynásobili dvema. Naopak bitový posuv císla doprava o jeden krok je stejný, jako bychom císlo delili dvema. Delení: Do registru umístíme delence. Ten potom posuneme doprava o tolik, kolikátou mocninou 2 je delitel:

- **SHR** AL, 1 - AL := AL div 2
- **SHR** AL, 2 - AL := AL div 4
- **SHR** AL, 3 - AL := AL div 8
- **SHR** AL, 4 - AL := AL div 16 . . .

Pozor! Toto delení je sice velmi rychlé, ale použitelné jen tehdy, jestliže chceme císlo delit mocninou 2 (a to

bývá naštestí nejcasteji). Ke zjištení zbytku po celocíselném delení použijeme operaci AND (jak bylo popsáno výše).
Násobení císla konstantou: Do tolika registru, kolik je log. 1 v binárním vyjádrení konstanty, umístíme hodnotu císla. Potom jednotlivé registry posuneme doleva. Každý o tolik, na kolikátém míste byla log. 1 v binárním vyjádrení konstanty. Nakonec všechny registry pricteme k jedinému, ve kterém bude výsledek.
Príklad: Vynásobme konstantou 18 vložené císlo:

- 18 : 2 = 9 (0)...0
- 9 : 2 = 4 (1)...1
- 4 : 2 = 2 (0)...2
- 2 : 2 = 1 (0)...3
- 1 : 2 = 0 (1)...4

Logická 1 je tedy na míste c.1 a c.4. Proto použijeme dva registry, ty posuneme o 1 a 4 kroky. Nakonec je secteme.

```
{$G+}
var cislo:word;
begin
 readln (cislo);
 asm
  MOV AX,cislo  {naber císlo do prvního registru}
  MOV BX, AX    {naber císlo do druhého registru}
  SHL AX, 1     {v prvním registru jednou doleva<=>vynásob 2}
  SHL BX, 4     {v druhém registru ctyrikrát doleva<=>vynásob 16}
  ADD AX, BX    {secti obsahy obou registru}
  MOV cislo, AX {vrat  pres promennou cislo}
 end;
 writeln ('Císlo*18=',cislo);
end.
```

Uvedený postup mužete snadno prevést na libovolnou konstantu. Vzhledem ke zdlouhavosti násobení instrukcí **MUL** vám tento algoritmus obcas zrychlí program.

Následující príklad vytvári retezec informací o case. Ten si zjistí z pameti CMOS. Ctení provádíme tak, že na adresu portu $70 vyšleme císlo ctené slabiky (0 - sekundy, 2 - minuty, 4 - hodiny) v CMOS. Z portu $71 potom precteme její hodnotu. Ta je v CMOS ve zhušteném BCD tvaru. Proto ji prevedeme na nezhuštený a teprve potom na kód ASCII. Nakonec data zapíši do promenné slovo typu string ve tvaru, v jakém je zvykem cas zapisovat. Program jsem optimalizoval tak, aby mel co nejmenší pocet instrukcí. Vzhledem k tomu, že ve vloženém assembleru jsem nepoužil cyklus, tvorím jej s pomocí pascalovského for cyklu. Podobným zpusobem bychom cetli i jiné užitecné informace z pameti CMOS (datum, konfigurace . . .).

Príklad:

```
uses crt;
var i:byte;
   slovo:string;
begin
 slovo[0]:=#8;
 slovo[3]:='.';
 slovo[6]:='.';
 clrscr;
 repeat
  for i:=0 to 2 do
```

```
 asm
  MOV BX,offset slovo {naber adresu promenné slovo do BX}
  XOR AH,AH        {vymaž horní polovinu registru AX}
  MOV AL,i         {naber do dolní poloviny AX krok i}
  SUB BX,AX        {odecti od BX obsah AX}
  SHL AL,1         {vynásob, AL:=AL*2}
  SUB BX,AX        {odecti od BX obsah AX}
  OUT $70,AL       {pošli na CMOS adresu ctené slabiky}
  IN AL,$71        {precti z CMOS obsah ctené slabiky}
  MOV AH,AL        {zkopíruj obsah prectené slabiky do AH}
  SHR AH,4         {desítky posun do dolní poloviny AH}
  AND AX,$0F0F     {odstran zbytecné bity}
  OR AX,$3030      {proved prevod do ASCII}
  MOV 8[BX],AL     {nastav jednotky v promenné slovo}
  MOV 7[BX],AH     {nastav desítky v promenné slovo}
 end;
 gotoxy (1,1);
 write(slovo);
until keypressed;
 readkey;
end.
```

Instrukce skoku

Protože si mikroprocesor vytvárí frontu instrukcí, nejsou z hlediska rychlosti behu programu skoky to pravé. Presto bychom složitejší programy bez nich asi težko tvorili. Abychom mohli instrukce skoku používat, musíme umet vytvorit náveští.

Náveští

Assembler je správne jen název prekladace "Jazyka symbolických adres", který se pro nej cassem vžil. Název "Jazyk symbolických adres" vyjadruje to, že místo adres instrukcí používáme symboly. V Turbo assembleru nejsme v názvech náveští nijak zvlášt omezováni. Ve vkládaném assembleru mužeme za název náveští použít posloupnost znaku zacínající znakem @ (@1, @zacatek, @navesti). Jestliže používáme náveští, deklarované mimo vkládaný assembler (s pomocí LABEL), není prítomnost znaku @ nutná. Náveští s dvojteckou uvedeme pred instrukci, na kterou se odkazujeme. Pri prekladu je v místech odkazu na náveští jeho název nahrazen skutecnou adresou instrukce.

Nepodmínený skok

Je to nepodmínený skok na jiné místo programu. To musí být oznacené náveštím. Za instrukcí skoku je potom uveden jeho název.

- **JMP** náveští - proved skok programu na náveští (ve skutecnosti se jen zmení obsah cítace instrukcí IP, prípadne CS pri vzdáleném skoku) V programu potom nepodmínený skok vypadá takto:

```
@navesti: instrukce na kterou bude odkaz
        .
        .
        JMP @navesti
```

Jestliže skoky používáme, hrozí vždy nebezpecí, že se program zacykluje (a nikdy neskoncí). Proto je duležité si vždy rozmyslet, za jakých okolností by k této kolizi mohlo dojít.

Podmínený skok

Jedná se o skok podmínený stavem jednoho nebo více, bitu registru príznaku F. Jen tímto zpusobem je možné provádet v assembleru prímé vetvení programu. Pred instrukcí podmíneného skoku proto vždy provedeme instrukci, která použitý príznak nastaví. V prípade, že není splnena podmínka skoku, pokracuje program dál, jako by se nic nedelo. Instrukce podmíneného skoku zacínají vždy písmenkem J. Za ním je zkratka udávající na jakých bitech registru F je skok závislý.

- **JE** náveští - skok na náveští pri ZF = 1
- **JZ** náveští - skok na náveští pri ZF = 1
- **JNE** náveští - skok na náveští pri ZF = 0
- **JNZ** náveští - skok na náveští pri ZF = 0
- **JC** náveští - skok na náveští pri CF = 1
- **JNC** náveští - skok na náveští pri CF = 0
- **JS** náveští - skok na náveští pri SF = 1
- **JNS** náveští - skok na náveští pri SF = 0
- **JO** náveští - skok na náveští pri OF = 1
- **JNO** náveští - skok na náveští pri OF = 0
- **JP** náveští - skok na náveští pri PF = 1
- **JNP** náveští - skok na náveští pri PF = 0
- **JPE** náveští - skok na náveští pri PF = 1
- **JPO** náveští - skok na náveští pri PF = 0
- **JA** náveští - skok na náveští pri (CF = 0) AND (ZF = 0)
- **JNBE** náveští - skok na náveští pri (CF = 0) AND (ZF = 0)
- **JAE** náveští - skok na náveští pri CF = 0
- **JNB** náveští - skok na náveští pri CF = 0
- **JB** náveští - skok na náveští pri CF = 1
- **JNAE** náveští - skok na náveští pri CF = 1
- **JBE** náveští - skok na náveští pri (CF = 1) OR (ZF = 1)
- **JNA** náveští - skok na náveští pri (CF = 1) OR (ZF = 1)
- **JG** náveští - skok na náveští pri (ZF = 0) OR (SF = OF)
- **JNLE** náveští - skok na náveští pri (ZF = 0) OR (SF = OF)
- **JGE** náveští - skok na náveští pri SF = OF
- **JNL** náveští - skok na náveští pri SF = OF
- **JL** náveští - skok na náveští pri SF <> OF
- **JNGE** náveští - skok na náveští pri SF <> OF
- **JLE** náveští - skok na náveští pri (ZF = 1) OR (SF <> OF)
- **JNG** náveští - skok na náveští pri (ZF = 1) OR (SF <> OF)

Pri hledání instrukce podmíneného skoku musíme myslet na to, za jakých okolností chceme skok vykonat. K tomu je také dobré si uvedomit:

- A < B => A - B < 0 => SF = 1
- A = B => A - B = 0 => ZF = 1
- A > B => A - B > 0 => SF = 0

Rozdíl císel v tomto prípade provedeme nejlépe instrukcí **CMP**. Pro tvorbu cyklu mužeme použít jeden z registru, který si pro krokovací promennou vycleníme. Jednoduchý cyklus pak vytvoríme podmíneným skokem:

```
begin
 asm
  MOV CL, 10 {do registru CL dosad 10, pocet kroku}
 @nav:      {náveští, tady umístíme opakovanou cinnost}
```

```
  DEC CL     {odecti od CL císlo 1}
  JNZ @nav   {jestliže není nula skoc na náveští}
 end;
end.
```

Program opakuje skok dokud není v registru CL nulový výsledek.

Nepodmínený a podmínený cyklus

ASM86 má i pro cyklus instrukci. Její použití však predpokládá to, že si rezervujeme registr CX pro cítání. Do nej pred cyklem umístíme pocet opakování. Instrukce **LOOP** pak cyklus umožní realizovat.

- **LOOP** náveští - od CX odecti jedna, jestliže je CX<>0 skoc na náveští

Príklad:

```
uses crt;
var pole:array [0..9] of byte;
    i:byte;
begin
 clrscr;
 asm
  XOR DI, DI {nuluj registr DI}
  MOV CX, 10 {do CX dej délku pole}
 @nav:       {náveští, zacátek cyklu}
  MOV BYTE PTR [DI+OFFSET pole], cl{presun do pole na místo urc. DI}
  INC DI     {na další prvek pole}
  LOOP @nav  {odecti od CX 1, není-li nula na @nav}
 end;
 for i:=0 to 9 do
  writeln (pole[i]);
 readkey;
end.
```

Uvedený príklad naplní pole hodnotami 1-10. Obsah v registru CX je použit ke krokování, a soucasne se s ním plní pole. Prvky pole jsou slabiky. Proto se obsah registru DI zvyšuje o jednu. V prípade, že by se jednalo o slova, musíme k registru DI pricítat 2. Cyklu vytvorenému pomocí **LOOP** se mužeme programove vyhnout instrukcí **JCXZ** náveští - jestliže je v CX nula presun se na náveští.

Príklad:

```
uses crt;
var pole1,pole2:array [0..9] of byte;
    i:byte;
    pocet:word;
begin
 clrscr;
 repeat {vstup poctu prvku kopie s kontrolou hodnoty pocet}
  write ('Zadej pocet kopirovanych prvku (0..10):');
  {$I-}readln (pocet);{$I+}
 until (ioresult=0) and (pocet in [0..10]);
 randomize;
```

```
for i:=0 to 9 do
begin
 pole1[i]:=random(256);
 pole2[i]:=random(256);
end;
asm
 MOV CX, pocet       {do registru CX dej pocet prvku kopie}
 JCXZ @konec         {jestliže je nulový jdi na konec}
 MOV SI, OFFSET pole1 {naber adresu pole1}
 MOV DI, OFFSET pole2 {naber adresu pole2}
@cykl:              {zacátek cyklu}
 MOV AL, [SI]        {do registru AL presun prvek z pole1}
 MOV [DI], AL        {z registru AL presun prvek do pole2}
 INC SI              {posun se na další prvek v polích}
 INC DI
 LOOP @cykl          {sniž CX o jednu, jestli je ruzné od nuly}
                     {skok na @cykl}
@konec:             {konec bloku asm}
end;
for i:=0 to 9 do
 writeln (pole1[i],'..',pole2[i]);
readkey;
end.
```

Až dosud jsme za podmínku opakování považovali nenulové císlo v registru CX. ASM86 však umožnuje podmínky opakování obohatit testováním príznaku ZF.

- **LOOPE** náveští <=> **LOOPZ** náveští - sniž CX o jednu a presun se na náveští pri (CX <> 0) AND (ZF = 1)
- **LOOPNE** náveští <=> **LOOPNZ** náveští<=> LOOP náveští - sniž CX o jednu a presun se na náveští pri (CX <> 0) AND (ZF = 0)

Pri použití techto instrukcí dáváme v programu možnost uniknout z cyklu i nastavením príznaku ZF. Nezapomente ale, že ZF se musí pred koncem cyklu opet nastavit vhodnou instrukcí.

Nastavení registru príznaku

Registr príznaku se cástecne nastavuje soucasne s vykonáváním nekterých instrukcí. Obsahuje ale i registry, které se automaticky nenastavují (IF, DF, TF). Proto ASM86 má instrukce, kterými mužeme prímo ovlivnit hodnoty nekterých bitu registru F.

- **CLC** - do registru CF vlož hodnotu log. 0
- **CMC** - neguj obsah registru CF
- **STC** - do registru CF vlož hodnotu log. 1
- **CLD** - do registru DF vlož hodnotu log. 0 (DI, SI pri práci s retezci zvyšuj)
- **STD** - do registru DF vlož hodnotu log. 1 (DI, SI pri práci s retezci snižuj)
- **CLI** - do registru IF vlož hodnotu log. 0 (zakaž prerušení)
- **STI** - do registru IF vlož hodnotu log. 1 (povol prerušení)

Jestliže chceme nastavit hodnotu v príznaku, pro který instrukce neexistuje, použijeme algoritmus:

- registr F predáme pres zásobník do nekterého z registru pro všeobecné použití
- v tomto registru logickou operací nastavíme bit príznaku

- pres zásobník opet predáme obsah registru do registru F

Príklad:

```
var promenna:byte;
begin
 asm
  MOV promenna,0  {nastav promennou do hodnoty 0}
  PUSHF          {ulož registr príznaku do zásobníku}
  POP AX         {presun obsah vrcholku zásobníku do registru AX}
  OR AX,1        {nastav poslední bit (CF) do logické 1}
  PUSH AX        {ulož obsah AX do zásobníku}
  POPF           {presun nazpátek do registru príznaku}
  JNC @konec     {otestuj nastavení CF}
  MOV promenna,1  {CF byl v 1, nastav hodnotu promenné do 1}
 @konec:
 end;
 writeln (promenna); {vypiš obsah promenné}
end.
```

Jednotlivé bity cásti registru príznaku mužeme také ovlivnit vhodným použitím instrukcí **LAHF** a **SAHF**.

Vyclenení pameti pro promenné v bloku asm

Ne vždy je vhodné používat pro naše promenné pamet hlavního programu. Možnost vyclenit si nekolik slabik dává i vložený assembler. Ve skutecnosti se jedná o cást pameti urcenou pro strojový kód. My si ale do ní umístíme hodnoty, na které vetšinou nezbylo místo v registrech. Protože je tento blok v segmentu programu, musíme tento blok promenných programove obejít. Mikroprocesor by totiž tyto hodnoty v pameti považoval za instrukce. Vyclenit místo si mužeme pomocí direktiv:

- **DB** - zde vyclen slabiky (8 bitu, hodnoty -128-255)
- **DW** - zde vyclen slova (16 bitu, hodnoty -32 768-65 535)
- **DD** - zde vyclen dvojslova (32bitu, hodnoty -2 147 483 648-4 294 967 295)

Za direktivu považujeme príkaz pro prekladac, není to tedy instrukce. S pomocí techto direktiv ríkáme prekladaci, aby v kódu programu rezervoval urcitý pocet slabik pro naše úcely. Za tyto direktivy rovnou píšeme pocátecní hodnoty slabik, slov a dvojslov oddelené cárkou. Pokud napíšeme jméno promenné deklarované pomocí var nebo jméno procedury, jedná se o jejich adresy (za direktivou **DW** offsetová cást adresy, za direktivou **DD** celá adresa, tedy ukazatel). Pro názornost si rovnou uvedeme program s temito direktivami.

Príklad:

```
var promenna:byte;
begin
 asm
  JMP @dal
 @slabiky:
  DB 10, 200,'M','Ahoj'
 @slova:
  DW 32000,'A',promenna
 @dvojslova:
  DD promenna
```

```
@dal:
 MOV AL, CS:[OFFSET @slabiky]   {do AL presun slabiku z adresy}
                    {@slabiky, AL:=10}
 MOV AL, CS:[OFFSET @slabiky+1] {do AL presun slabiku}
                    {z @slabiky+1, AL:=200}
 MOV AL, CS:[OFFSET @slabiky+2] {do AL presun hodnotu ASCII}
                    {znaku 'M'}
 MOV AL, CS:[OFFSET @slabiky+3] {do AL presun ASCII prvního znaku}
                    {retezce 'Ahoj'}
 MOV AL, CS:[OFFSET @slabiky+4] {do AL presun ASCII druhého znaku}
                    {retezce 'Ahoj'}
 MOV AX, CS:[OFFSET @slova]     {do AX presun slovo z adresy}
                    {@slova, AX:=32000}
 MOV AX, CS:[OFFSET @slova+2]   {do AX presun hodnotu ASCII znaku}
                    {'A', AH:=0,AL:=65}
 MOV BX, CS:[OFFSET @slova+4]   {do BX presun offset promenné}
                    {promenna}
 MOV BYTE PTR [BX], AL          {do této promenné zapiš obsah}
                    {registru AL}
 LES BX,CS:[OFFSET @dvojslova]  {naber obsah ukazatele, tedy}
                    {celou adresu promenné do ES:BX}
 SEGES MOV BYTE PTR [BX], AL    {na celou adresu promenné zapiš}
                    {obsah AL}
end;
end.
```

Na takto vytvorená místa mužeme samozrejme i zapisovat. Pokud nechceme používat náveští pro každou cást, stací si jen pamatovat, kolik místa zabere slabika, slovo, nebo dvojslovo. Potom se na hledanou cást dostaneme pricítáním, nebo odcítáním urcitých hodnot k offsetu náveští. Zajímavé je i využití adres promenných. Protože promenná za direktivou **DD** je celá adresa, mužeme naplnit instrukcí **LES** (**LDS**) oba registry, tedy segment i offset. Pokud zapíšeme **DB** 4, 'Ahoj', jedná se o klasický pascalovský retezec z délkou na zacátku.

Instrukce pro práci s retezci

ASM86 má velmi silný nástroj v retezcových instrukcích. Za retezec je zde na rozdíl od Pascalovského považován blok dat v pameti o témer libovolné délce (podle definice jsme omezeni jen velikostí segmentu, to se ale dá snadno obejít). Pro použití retezcových instrukcí jsou vycleneny dvojice registru, které nesou adresy:

- DS:SI - pro adresu zdrojového retezce
- ES:DI - pro adresu cílového retezce

V praxi to znamená, že vždy jeden blok v pameti je oznacen za zdrojový, druhý za cílový. Duležitou roli zde hrají i registry:

- CX - nese délku retezce
- DF - urcuje smer zpracování retezcu (0 - adresy se zvyšují, 1 - adresy se snižují)

Retezové instrukce pak jsou

- **LODSB** (**LODSW**) - presun z adresy DS:SI do registru AL (AX) a zvyš SI o jednu (o dve)
- **STOSB** (**STOSW**) - presun z registru AL (AX) na adresu ES:DI a zvyš DI o jednu (o dve)

- **MOVSB** (**MOVSW**) - presun z adresy DS:SI slabiku (slovo) na adresu ES:DI a SI, DI zvyš o jednu (o dve)
- **CMPSB** (**CMPSW**) - porovnej (odecti) slabiku (slovo) na adrese DS:SI se slabikou (slovem) na adrese ES:DI, podle výsledku nastav príznaky (ZF = 1 pri shode, ZF = 0 pri neshode), potom zvyš adresy SI a DI o jednu (o dve)
- **SCASB** (**SCASW**) - porovnej (odecti) slabiku z adresy ES:DI z registrem AL (AX), podle výsledku nastav príznaky (ZF = 1 pri shode, ZF = 0 pri neshode), potom zvyš adresu DI o jednu (o dve)
- **INSB** (**INSW**) - [286], presun z portu s adresou v DX do pameti s adresou ES:DI slabiku (slovo) a adresu DI zvyš o jednu (o dve)
- **OUTSB** (**OUTSW**) - [286], presun z pameti s adresou DS:SI slabiku (slovo) na port urcený adresou v DX a zvyš adresu SI o jednu (o dve)

Slovo zvýšit v techto popisech cinnosti nahradíme slovem snížit pri DF = 1. Tyto instrukce umožní najednou provést urcitou cinnost a pritom aktualizují adresy podle stavu DF a podle toho, jestli pracujeme se slabikami nebo slovy.

Následující príklad využívá prímého zápisu do videopameti (VRAM) v textovém režimu VGA k výstupu pascalovského retezce. VRAM, zacíná na adrese $B8000. Je organizovaná jako pole slov nesoucích informace o zobrazovaných znacích. Každé slovo nese slabiku atributu (barva znaku a jeho pozadí) a slabiku s ASCII kódem zobrazeného znaku. 80 slov VRAM je jeden rádek na obrazovce. Proto pri zvýšení adresy $B8000 o 160 mužeme pracovat s druhým rádkem atd.

Príklad:

```
var slovo:string;
begin
 slovo:='Ahoj';
 asm
  PUSH DS       {ulož obsah DS do zásobníku, budeme ho menit}
  JMP @dal      {obejdi data}
 @vram:
  DW $0000,$B800 {offset:segment VRAM, Pozor! je to obrácene}
 @adsl:
  DD slovo      {adresa slova, ukazatel na nej}
 @dal:          {zacátek programu}
  LDS SI,CS:[OFFSET @adsl] {DS:SI nasmeruj na zdroj (na slovo)}
  LES DI,CS:[OFFSET @vram] {ES:DI nesmeruj na VRAM}
  XOR CH,CH     {nuluj CH}
  MOV CL,[SI]   {do CL dej délku retezce slovo, 1. slabiku}
  INC SI        {posun se za slabiku s délkou}
  MOV AH,$6F    {do AH dej atributy nápisu}
 @cyk:          {cyklus pro znak po znaku}
  LODSB         {naber kód znaku z retezce do AL a zvyš SI+1}
  STOSW         {ulož obsah AX do VRAM, zvyš DI+2}
  LOOP @cyk     {sniž CX o jednu, není-li nula jdi na @cyk}
  POP DS        {vrat registr DS do puvodního stavu}
 end;
end.
```

Uvedený program zmení slabiku na slovo v registru AX s tím, že bude kód znaku doplnen o atributy. Jestliže zmeníme hodnotu v AH ovlivníme tím barvu výstupu.

Prefix opakování

Dosud známe jen prefix preskocení. Prefix opakování se používá pred retezcovými instrukcemi a umožnuje tak jejich podmínené i nepodmínené opakování. Jejich použitím zrychlíme a zjednodušíme program. Nepodmíneným prefixem je

- **REP** instrukce - opakuj instrukci tolikrát, kolik je uvedeno v registru CX (CX := CX - 1, opakuj dokud CX <> 0)

Tento prefix píšeme vetšinou pred instrukci **MOVSB** (**MOVSW**). Jestliže máme nastavený registr CX na pocet prvku retezce a adresové registry zdrojového a cílového retezce, zajistí **REP** jejich zkopírování na jednom rádku programu (napr. **REP MOVSB**).

Príklad:

```
var slovo1,slovo2:string;
begin
 slovo1:='Ahoj';
 asm
  PUSH DS        {ulož do zásobníku obsah DS, zmeníme ho}
  JMP @dal       {skoc na zacátek, obejdi data}
 @adr:
  DD slovo1,slovo2 {definice ukazatelu na pole}
 @dal:
  LDS SI,CS:[OFFSET @adr]   {naber adresu zdrojového retezce}
  LES DI,CS:[OFFSET @adr+4] {naber adresu cílového retezce}
  XOR CH,CH              {nuluj CH}
  MOV CL,[SI]            {do CL dej délku retezce}
  INC CX                 {pascalovský retezec nese o slabiku více}
  REP MOVSB                {kopíruj retezce po slabikách}
  POP DS                 {vrat  obsah DS ze zásobníku}
 end;
 writeln (slovo1,' ',slovo2);
 readln;
end.
```

V príkladu kopírujeme jen tolik prvku, kolik má zdrojové slovo slabik. Tuto informaci si zjistíme z první slabiky promenné slovo1. K tomu musíme ješte pricíst 1, protože pascalovský retezec nese navíc informaci o délce. I když veškeré presuny se odehrávají v datovém segmentu s adresou v DS, je dobré si zvyknout na to, že vždy, když meníme DS, ukládáme jeho obsah pro jistotu do zásobníku.

Retezcové instrukce vyhledání a porovnání využívají registr príznaku ZF. Proto ASM86 obsahuje navíc prefixy podmíneného opakování:

- **REPE** instrukce <=> **REPZ** instrukce - opakuj tolikrát, kolik je v registru CX a dokud je ZF = 1 (CX := CX - 1, zopakuj pokud je (CX <> 0) AND (ZF = 1))
- **REPNE** instrukce <=> **REPNZ** instrukce - opakuj tolikrát, kolik je v registru CX a dokud je ZF = 0 (CX := CX - 1, zopakuj pokud je (CX <> 0) AND (ZF = 0)) Opakování je tedy prerušeno nejen pri nulovém CX, ale i pri nastavení ZF do log. 1 nebo 0.

Príklad:

```
uses crt;
var pole:array [0..9] of word;
   hledany,pozice:word;
```

```
   i:byte;
begin
 clrscr;
 randomize;
 for i:=0 to 9 do
  pole[i]:=random(65535);   {do pole náhodná císla}
 hledany:=pole[random(10)]; {vyber hledané císlo}
 writeln ('Hledam:',hledany);
 asm
  JMP @zac                 {skok na zacátek}
 @adr:
  DD pole                  {definice ukazatele na pole}
 @zac:
  MOV AX,hledany           {do AX vlož hledané císlo}
  MOV CX,10                {do CX vlož délku retezce (pole)}
  LES DI,CS:[OFFSET @adr]   {naber adresu retezce}
  REPNE SCASW              {opakuj do shody porovnání}
  MOV pozice,9             {spocítej kolikátý je hledaný,}
  SUB pozice,CX            {k tomu použiješ to, co zbylo v CX}
 end;
 for i:=0 to 9 do
 begin
  if i<>pozice then textcolor(15) else textcolor(12);
  writeln (pole[i]);
 end;
 readkey;
end.
```

Tento program vyhledá slovo v poli. K tomu slouží jen rádek **REPNE SCASW**. Ten opakuje pohyb po poli, dokud nenajde shodu s hodnotou v registru AX (ta se projeví nastavením ZF do 1) . K zjištení pozice hledaného dobre poslouží zbytek v registru CX. Kdyby byl zbytek nulový, hledaný prvek by v poli nebyl.

Príklad:

```
uses crt;
var slovo1,slovo2:string;
   ukazatel:pointer;
   i,misto,delka:word;
begin
 slovo1:='Nazdar programátori! '+
       'Zkuste vyhledat nejaké slovo z této vety.';
 slovo2:='slovo';
 delka:=length(slovo2);
 asm
  PUSH DS                  {ulož DS, budeme ho menit}
  JMP @dal                 {preskoc data}
 @ukp:
  DD slovo1,slovo2         {ukazatele na retezce}
 @dal:
  LDS SI,CS:[OFFSET @ukp]  {naber adresu zdroje}
  INC SI                   {preskoc délku retezce}
 @cyk:
```

```
 LES DI,CS:[OFFSET @ukp+4]{naber adresu cíle, hledaného slova}
 INC DI              {preskoc slabiku s délkou retezce}
 MOV CX,delka          {do CX vlož délku retezce}
 REPE CMPSB            {opakuj do neshody (konce hledaného)}
 JZ @konec           {byla shoda tak na konec}
 SUB SI,delka        {nebyla shoda tak se v SI vrat }
 INC SI
 ADD SI,CX           {k návratu v SI použij zbytek v CX}
 JMP @cyk            {a znovu hledat}
@konec:
 POP DS              {vrat  obsah DS, už ho nebudeme menit}
 MOV misto,SI        {vypocítej místo v prohledávaném}
 MOV SI,CS:[OFFSET @ukp]  {k tomu použiješ délku retezce zdroje}
 ADD SI,delka        {délku cíle, tedy hledaného}
 SUB misto,SI
end;
clrscr;
for i:=1 to length(slovo1) do
begin
 if not(i in [misto..misto+delka-1]) then
  textcolor (15)
                      else
  textcolor(12);
 write(slovo1[i]);
end;
readkey;
end.
```

V príkladu prohledáváme retezec slovo1. Hledáme v nem umístení podretezce slovo2. Program má dva cykly v sobe. První zajištuje pohyb po prohledávaném retezci v prípade neshody (je realizován **JMP**). Druhý vnitrní zajištuje pohyb po prohledávaném s kontrolou s hledaným (je realizován **REPE**). V prípade shody je po cyklu **REPE** v registru ZF = 1 (proste nevyskocil neshodou ale nulou v CX=> konec hledaného slova a shoda). Proto cyklus prohledávání ukoncíme podmíneným skokem JZ na konec. Zde ze zjistí adresa v prohledávaném retezci. To je ale adresa za posledním znakem shody. Proto se vrátíme nazpátek o délku slova (tam je hledané slovo).

Nedokumentované instrukce

Když firma Intel navrhovala mikroprocesor 8086, byly vloženy do instrukcního souboru i instrukce, které nebyly oficiálne uvedeny v tabulkách. Presto je metodou pokusu programátori objevili. Ve svých programech mužeme tyto instrukce používat. Máme však následující omezení:

- Prekladace assembleru tyto instrukce neznají, proto je do programu vložíme napríklad následovne: **DB** $D4, 10. Kde **DB** je definice slabiky (libovolné), $D4 je kód instrukce, která má jeden operand, nyní hotnotu 10. Cást **DB** v tomto prípade samozrejme neobcházíme **JMP**, necháme jí tedy provést, jakoby se jednalo o program.
- Do budoucnosti není zarucena funkcnost techto instrukcí na nových procesorech rady 86.

Seznam a funkci pro nás použitelných nedokumentovaných instrukcí najdete v tabulce instrukcí.

Volání podprogramu

V úvodu jsem upozornil na to, že využití vkládaného assembleru je v tvorbe podprogramu. Predem si ale

musíme ukázat, jak se podprogramy volají.

Volání podprogramu spocívá v uložení parametru do zásobníku a zmene adresy v registru IP (cítac instrukcí) na adresu podprogramu s tím, že je uschována adresa odkud provádíme volání (to aby procesor vedel kam se má vrátit). Parametry do zásobníku ukládáme my, zbytek zarídí instrukce **CALL**.

Ukládání parametru do zásobníku

V hlavicce procedury (nebo funkce) najdeme témer vždy definici parametru volaných:

- hodnotou - podprogram jejich hodnoty pouze využívá
- odkazem - podprogram je muže císt a muže do nich i zapsat

Napríklad: procedure soucet (a,b:word;var c:word); je definice procedury s názvem soucet s parametry a, b volanými hodnotou a c volaným odkazem. Pri volání této procedury z nekteré cásti programu psaném v Pascalu na místa a, b zapíšeme konkrétní hodnoty (nebo promenné (ty ale podprogram nezmení) s temito hodnotami) a na místo c zapíšeme promennou, ve které najdeme hodnotu po provedení procedury (napr. soucet (1,3,promenna_c);). Z místa volání predáváme parametry do podprogramu vždy pres zásobník v poradí definice v hlavicce podprogramu. Do zásobníku pred voláním procedury ukládáme odlišne u parametru volaných hodnotou a odkazem.

- **Pri volání hodnotou**
  Uložíme konkrétní hodnoty (prectené treba i z pameti). Vzhledem k organizaci zásobníku jsou parametry volané hodnotou uloženy po slovech následovne:
  - parametry o délce jedné slabiky (byte, shortint, char, boolean) - obsadí celé slovo (pametí nešetrí)
  - parametry o délce jednoho slova (word, integer) - obsadí slovo
  - parametry o délce dvojslova (pointer, longint) - obsadí dve slova (ukazatel je adresa, do zásobníku tedy napred uložíme segmentovou a pak offsetovou cást adresy)
  - parametry o délce 6 slabik (real) - obsadí v zásovníku tri slova
  - parametry delší (retezce, množina, pole, záznamy) - se ukládají jako ukazatele na hodnotu.
- **Pri volání odkazem**
  Uložíme celou adresu místa (tedy segment i offset) odkud se má hodnota císt nebo kam se má zapsat (to je vlastne obsah ukazatele na pametové místo).

Samotné volání podprogramu

Musíme rozlišovat volání blízkého podprogramu a vzdáleného. Za vzdálený v tomto prípade považujeme podprogram s adresou v odlišném segmentu. I když se pro programátora nic nemení je dobré vedet, že pri vzdáleném volání se mení nejen IP, ale i CS. Oznacení místa skoku nese tedy navíc informaci o segmentové adrese. Skok do podprogramu zajistí instrukce

- **CALL** adresa - na vrchol zásobníku ulož obsah (CS pri vzdáleném volání a) IP a napln tyto registry adresou uvedenou v parametru (pro nás slovo adresa nahradíme názvem podprogramu)

Ukoncení samotného podprogramu zajistí instrukce

- **RET**[**F**] - z vrcholu zásobníku vezmi adresy a dosad je do (CS a) IP Volání podprogramu je tedy jednoduché.

Jednoduše napíšeme instrukci **CALL** se jménem podprogramu (tedy procedury nebo funkce). Ostatní zarídí prekladac, který zjistí, jestli se jedná o blízké nebo vzdálené volání. Podle toho dosadí adresu. Návrat si opet zarídí prekladac pri ukoncení podprogramu.

Príklad:

```
{$G+}
uses crt;
procedure pocitej (a,b:word;var c,d:word);
begin
 c:=a+b;
 d:=a-b;
end;

var a_,b_,c_,d_:word;
begin
 a_:=40;
 b_:=5;
 clrscr;
 asm
  PUSH a_      {procedure posíláme hodnotu a_}
  PUSH b_      {procedure posíláme hodnotu b_}
  LEA DI,c_    {zjistíme adresu promenné c_}
  PUSH DS      {do zásobníku segment adresy c_}
  PUSH DI      {do zásobníku offset adresy c_}
  LEA DI,d_    {to samé pro d_}
  PUSH DS      {stejný segment}
  PUSH DI      {offset d_}
  CALL pocitej {a zavoláme pocítej}
 end;
 writeln (a_,'+(-)',b_,'=',c_,'(',d_,')');
 readkey;
end.
```

Stejnou posloupnost instrukcí jako blok asm v tomto programu provede rádek pocítej (a_,b_,c_,d_);

Návrat hodnoty z funkce

Funkce je podprogram, který vrací jednu hodnotu typu uvedeného v záhlaví. Vracenou hodnotu zjistíme po návratu z funkce vždy v registrech:

- AL - funkcní hodnota o velikosti slabiky
- AX - funkcní hodnota o velikosti slova
- DX, AX - funkcní hodnota o velikosti dvojslova (u ukazatele DX - segment, AX - offset)
- DX, BX, AX - funkcní hodnota typu real

Pokud funkce vrací retezec, musí být volána i s adresou místa, kam má výsledný retezec zapsat.

Príklad:

```
{$G+}
uses crt;
function bez1 (a:word):word;
begin
 bez1:=a-1;
end;
```

```
var a_,c_:word;
begin
 a_:=40;
 clrscr;
 asm
  PUSH a_   {posíláme hodnotu a_}
  CALL bez1 {zavoláme }
  MOV c_,AX {slovo si vyzvedneme v registru AX}
 end;
 writeln (a_,'-1=',c_);
 readkey;
end.
```

Tvorba podprogramu

Bloky programu, které vykonávají cinnost casto se opakující, nazveme podprogramem. Jejich použitím zjednodušíme program. Za podprogramy pokládáme procedury a funkce. Pascal umožnuje vkládat assembler i do obycejných podprogramu. Mužeme také tvorit podprogramy pouze v assembleru. To vyjádríme zápisem assembler za definici procedury nebo funkce. Ty potom neobsahují klasické vymezení bloku begin...end, stací jen assemblerovské asm..end (pokud tedy tvoríme podprogram jen v assembleru, uvedeme za definici oznacení assembler, blok vymezíme asm...end). S parametry pracujeme v podprogramech v souladu s tím, jak jsme je pres zásobník predávali. To znamená, že k parametrum volaným hodnotou pristupujeme jako ke klasickým promenným, k parametrum volaným odkazem pristupujeme jako k ukazatelum (dosazujeme jejich adresu instrukcí **LES**, **LDS**).

Lokální promenné

V okamžiku vstupu do podprogramu se na vrcholu zásobníku automaticky vytvorí místa pro lokální promenné definované v cásti var podprogramu. V prípade, že se jedná o pascalovskou funkci (není oznacena slovem assembler v definici), je navíc vložena speciální promenná @RESULT urcená k predání funkcní hodnoty (ta je i stejného datového typu). Pred návratem z funkce je obsah promenné @RESULT automaticky predán do registru predepsaných pro návrat hodnoty (pokud tedy tvoríme funkci s vloženým assemblerovským blokem, predáme funkcní hodnotu do promenné @RESULT, ve funkci s oznacením assembler vracíme funkcní hodnotu v registrech, ve kterých funkcní hodnotu ocekává volající (AL, AX,..), jak bylo uvedeno v cásti o volání podprogramu). Lokální promenné používáme stejne jako globální (s tím rozdílem, že jejich segmentová adresa není v DS).

Význam registru BP

Registr BP je v dobe vykonávání podprogramu nasmerován na vrcholek zásobníku v okamžiku vstupu do nej. Proto použitím neprímé bázové adresace s pomocí tohoto registru mužeme pristupovat k:

- parametrum - prícítáním k hodnote v BP (napr. [BP + 6] je oznacení pro prístup k parametru)
- lokálním promenným - odecítáním od hodnoty v BP (napr. [BP - 2] je oznacení prístupu k prvnímu parametru typu word)

Vzhledem k tomu, že se o tyto prepocty adres muže postarat prekladac, je jednodušší používat pro prístupy k promenným a parametrum jen jejich symboly uvedené v definici podprogramu nebo cásti var.

Príklad:

```
uses crt;
procedure pocitej (a,b:word;var c,d:word);assembler;
asm
```

```
 MOV AX,a      {do registru ax, vlož hodnotu a}
 ADD AX,b      {pricti b}
 LES DI,c      {do ES:DI vlož adresu c (to je výstup souctu)}
 MOV ES:[DI],AX {na adresu ES:DI zapiš soucet}
 MOV AX,a      {to samé pro rozdíl}
 SUB AX,b
 LES DI,d
 MOV ES:[DI],AX {a na adresu d zapiš rozdíl}
end;

function bez1 (a:word):word;assembler;
asm
 MOV AX,a      {do AX vlož hodnotu parametru a}
 DEC AX        {kdyby to nebyla ciste assemblerovská funkce, tak}
               {pridám rádek:}
               {MOV @RESULT, AX fce hodnotu pak také vrátí v AX}
end;

var a_,b_,c_,d_:word; {hlavní program}
begin
 a_:=40;b_:=5;
 clrscr;
 pocitej (a_,b_,c_,d_);
 writeln (a_,'+(-)',b_,'=',c_,'(',d_,')');
 c:=bez1 (a_);
 writeln (a_,'-1=',c_);
 readkey;
end.
```

Prerušení

V dobe vykonávání úlohy musí být zajištena i programová obsluha nekterých událostí. Za tyto události považujeme napríklad: stisk klávesy, pohyb myší, hrozící výpadek napájení, kritická chyba v pameti, . . . I když by bylo možné testovat stisk klávesy v rámci provádené úlohy, je pohodlnejší, jestliže obsluhu této události zajistí pocítac sám na úrovni technického vybavení. Presto je k této cinnosti nutný mikroprocesor. Proto je docasne prerušena probíhající úloha. Po obsluze se procesor vrací zpet k té casti úlohy, ze které byl prerušen.

Celý mechanismus prerušení se dá popsat v nekolika krocích:

- Do radice prerušení prichází požadavek o prerušení, ten vyhodnotí jeho prioritu. Jestliže je prerušení možné, je vyslán do procesoru signál požadavku o prerušení.
- Mikroprocesor prijal signál požadavku prerušení. Jestliže je prerušení možné (není zakázáno nastavením IF = 0), po dokoncení probíhající instrukce vyšle procesor signál potvrzení prerušení.
- Radic prerušení prijal signál povolení prerušení. Vyšle na datovou sbernici instrukci prerušení **INT** císlo, ta zajistí, že procesor provede tyto cinnosti:
    - do zásobníku se uloží registr príznaku F (po návratu se musí obnovit)
    - vynulují se príznaky IF (zakáže se další prerušení) a TF (nejde krokovat program)
    - do zásobníku se uloží obsahy CS a IP (místo, kde byla prerušovaná úloha)
    - registry CS a IP se naplní adresou, prectenou z tabulky vektoru prerušení (to je tabulka na zacátku pameti, v ní jsou za sebou uloženy celé adresy všech obsluh prerušení, klícem pro hledání v této tabulce je práve císlo prerušení uvedené za instrukcí INT)
- Probehne obsluha prerušení (napríklad nactení dat, hláška na obrazovku,...).

- Po obsluze je ze zásobníku obnoven obsah registru IP, CS, F (procesor se vrátí k puvodní úloze, príznaky TF, IF se obnoví s registrem F). Obnovu techto registru zajistí instrukce **IRET** (která je na konci obsluhy prerušení).

Za instrukcí **INT** muže být císlo v rozpetí 0..255. Toto císlo v prípade obsluhy programové události udává, odkud požadavek prišel. Protože je ale nemožné, aby všech 256 úrovní prerušení bylo obsazeno, jsou nekteré hodnoty obsazeny tzv. službami.

Za služby mužeme považovat podprogramy, které jsou soucástí operacního systému nebo BIOSu. Jsou umísteny v pameti pocítace. Umožnují jednoduše provádet cinnosti, které se v programech casto opakují, jsou pracné nebo se liší na pocítacích s ruznou konfigurací.

Služby voláme stejne jako obsluhy prerušení instrukcí **INT** císlo. Hodnota císlo urcuje, o jakou službu se jedná. Casto se v rámci jedné služby muže vyskytovat i nekolik cinností. Tem budeme ríkat podslužby. Pred voláním podslužeb musíme napred nastavit v urcitém registru (nejcasteji v AH) hodnotu jim urcenou. Potom teprve voláme službu instrukcí **INT**. Mnoho služeb se chová jako podprogramy volané parametry. Hodnoty parametru se neukládají do zásobníku, ale do nekterých registru. Výstupy z techto "podprogramu" najdeme opet v registrech. Informace o službách DOSu i BIOSu najdete v odborných publikacích nebo v SYSMANu. Zde také najdete informace o tom, které registry k cemu použijete.

Nejpoužívanejší službou je **INT** $21. Ta zahrnuje služby DOSu jako je vstup a výstup dat, práce se soubory, cas, . . . Je také použita k výstupu pascalovského retezce na obrazovku v následujícím príkladu. Výstup retezce realizuje podslužba AH = $9. Vstupem do podslužby je adresa retezce v registrech DS, DX. Výstup podslužba nemá. Jediná cinnost je výpis na obrazovku. Duležité je oznacení konce retezce znakem $. V prípade, že tento znak na konci není, vypíše se obsah cásti pameti až do jeho náhodného výskytu.

Príklad:

```
procedure outstring (retezec:string);assembler;
asm
 PUSH DS        {ulož DS, budeme ho menit}
 MOV AH,$09     {nastav hodnotu podslužby}
 LDS DI,retezec {cti adresu retezce}
 MOV DX,DI      {vlož ji do registru DX pro podslužbu}
 INC DX         {zvyš adresu až za informaci o délce}
 XOR BH,BH      {nuluj BH}
 MOV BL,[DI]    {do BL vlož délku retezce}
 MOV BYTE PTR [DI+BX+1],'$'{na konec retezce dosad ukoncovací znak}
 INT $21        {volej služby DOSu}
 POP DS         {vrat DS}
end;

begin
 outstring ('Ahoj'); {zkus vypsat}
end.
```

Uvedený program prevede pascalovský retezec do podoby retezce, ve které ho ocekává služba. Nastaví registry hodnotami vstupu a zavolá podslužbu DOSu. Výstup retezce touto procedurou mužeme realizovat na libovolném grafickém adaptéru. Možné odlišnosti si vyreší práve služba DOS.

Rezidentní programy

Velká skupina programu je schopna pracovat na pozadí provádené úlohy. Patrí mezi ne ovladace (myši, klávesnice, . . .), utility (hodiny, antivirová kontrola, stahovace obrazovek, . . .), viry (bez komentáre). Temto programum pridáváme oznacení rezidentní.

Jejich základní vlastností je jejich neustálá prítomnost v pameti pocítace a schopnost se vyvolat, jestliže je to nutné. Z toho vyplývají i požadavky na ne: malá délka kódu (musí obsadit co nejméne pameti) a nezávislost na spuštených aplikacích.

Cinnost techto programu na pozadí aplikací zarucuje jejich volání spolu s obsluhami prerušení. Jestliže tedy dojde k nejaké události (stisk klávesy, prijetí dat na port, uplynutí urcité doby, . . .), je voláno prerušení obsluhující tuto událost. Po této obsluze (,nebo pred ní) probehne i cást rezidentního programu pripojeného k ní. Aby k tomu došlo, musí tvurce rezidentního programu zmenit adresu v tabulce vektoru prerušení na adresu svého podprogramu. Pritom si starou adresu obsluhy uschová, aby mohl zajistit volání puvodní obsluhy události. Je jen na tvurci, jestli starou obsluhu bude volat nebo ne (jestliže ji ale nezavolá, mohou se vyskytnout problémy). Programátor se také muže rozhodnout, ve které cásti svého programu bude obsluhu volat (napr. nemohu císt jaká klávesa byla stisknuta, když ješte neprobehla obsluha klávesnice). Rezidentní program má tyto cásti:

- podprogramy, které jsou volány s prerušením (za jejich hlavickou následuje slovo interrupt) vykonávající užitecnou nebo záškodnickou cinnost; ty navíc mohou volat puvodní obsluhy (posloupností instrukcí PUSHF, CALL adresa staré obsluhy)
- hlavní program, který má za úkol:
    - prectení adresy puvodní obsluhy prerušení a její uložení do promenné (typu procedure); to zajistí procedura z knihovny DOS: GetIntVec (císlo prerušení, adresa promenné typu procedure)
    - zmena puvodní adresy na adresu našeho podprogramu; to zajistí procedura z knihovny DOS: SetIntVec (císlo prerušení, adresa našeho podprogramu)
    - ohlášení instalace (napr. Writeln ('Rezidentní program instalován.');)
    - ukoncení programu s tím, že zustane v pameti; to zajistí procedura Keep (0)

V Pascalu musíme navíc v rezidentním programu ohranicit podprogramy interrupt direktivou {$F+}, která zajistí, že bude uvnitr použito vzdálené volání (za podprogram napíšeme {$F-} pro návrat do automatického zjištování vzdálených adres). Navíc musíme zajistit správnou alokaci pameti pro rezidentní program oznacením v úvodu programu {$M 400,0,0}, které vymezí oblast rezervovanou pro zásobník atd. (hodnoty je nejlepší vyzkoušet).

Nejcasteji se pro rezidentní programy používají prerušení:

- $1C - volané 18,2krát za vterinu
- $09 - volané po události na klávesnici (stisk klávesy)
- $28 - volané v prípade, že mikroprocesor není zatížený (ceká . . .)

Ostatní hodnoty prerušení se dají zjistit z literatury (nebo SYSMANu).

Na jaké prerušení rezident pripojíme, závisí do znacné míry na tom, co má delat a na co má reagovat. Obcas je dobré si v obsluze jednoho prerušení nastavit promenné a v závislosti na jejich stavu vykonat (nebo nevykonat) urcitou cinnost v obsluze jiného prerušení. Casto si ani neuvedomíme, že náš podprogram pripojený k urcitému prerušení, ho neprímo volá. Dojde tak k zacyklení. Toho se cástecne vyvarujeme tím, že veškeré cinnosti, spojené se vstupy a výstupy, provádíme sami a nevoláme pascalovské procedury (napr. výstup na obrazovku realizujeme prímým zápisem do VRAM, použití writeln vede k chybe).

Príklad:

```
{$M $400,0,0}          {nastav pamet : zásobník $400 slabik}
uses Dos;
var IntVec : Procedure; {promenná pro adresu staré obsluhy}

{$F+}                  {vzdálená volání}
```

```
procedure hodiny;interrupt;assembler; {nová obsluha prerušení}
asm
 JMP @zac              {preskoc data}
@vid:
 DW 156,$B800          {adresa místa VRAM, kde budou hodiny}
@zac:
 MOV CL,2              {hodiny, minuty, vteriny (cyklus)}
@c1 :                  {zacátek cyklu}
 LES BX,CS:[OFFSET @vid]{naber adresu promenné slovo do BX}
 XOR AH,AH             {vymaž horní polovinu registru AX}
 MOV AL,CL             {naber do dolní poloviny AX krok i}
 SHL AL,1              {vynásob, AL:=AL*2}
 SUB BX,AX             {odecti od BX obsah AX}
 OUT $70,AL            {pošli na CMOS adresu ctené slabiky}
 SHL AL,1              {vynásob, AL:=AL*2}
 SUB BX,AX             {odecti, to ovlivní tvaru výstupu}
 IN AL,$71             {precti z CMOS obsah ctené slabiky}
 MOV DL,AL             {zkopíruj obsah této slabiky do AH}
 SHR DL,4              {desítky posun do dolní poloviny AH}
 AND AX,$F             {odstran zbytecné bity}
 AND DX,$F
 OR AX,$1F30           {proved prevod do ASCII, pridej atr.}
 OR DX,$1F30 MOV ES:2[BX],AX {nastav jednotky ve VRAM}
 MOV ES:[BX],DX        {nastav desítky ve VRAM}
 DEC CL                {snížit CL}
 JNS @c1               {konec cyklu}
 MOV WORD PTR ES:[154],$1F00+'.'{ve VRAM oddel vteriny a minuty}
 MOV WORD PTR ES:[148],$1F00+'.'{ve VRAM oddel minuty a hodiny}
 PUSHF                 {do zásobníku registr príznaku}
 CALL IntVec           {volej starou obsluhu $1C}
end;
{$F-}                  {konec vzdálených volání}

begin                  {hlavní program}
 GetIntVec($1c,@IntVec);     {cti adresu staré obsluhy}
 SetIntVec($1c,Addr(hodiny));{na její místo dej adresu mojí obsluhy}
 Writeln('Rezidentní hodiny instalovány.');{informuj o instalaci}
 Keep (0);             {ukonci s tím, že zustane program v pameti}
end.
```

Uvedený program cte pri obsluze prerušení $1C stav hodin z pameti CMOS. Po prepoctu adres a úprave znaku z BCD kódu do ASCII je informace o case zobrazena v pravém horním rohu obrazovky. Hlavní program má za úkol jen zmenu adresy puvodní obsluhy na naší.

Literatura:

- Michal Brandejs: Mikroprocesory INTEL 8086-80486, Grada 1991
- Tomáš Novák: Turbo Pascal 6 - Popis jazyka, Grada 1991
- Pavel Mikula, Katerina Juhová, Jirí Soukenka: Borland Pascal 7.0, Kompendium, Grada 1994

© 1996-1997 Mgr. Tomáš Papoušek

Tento text je možné používat pro studijní úcely bez omezení. V prípade komercního využití kontaktujte autora.