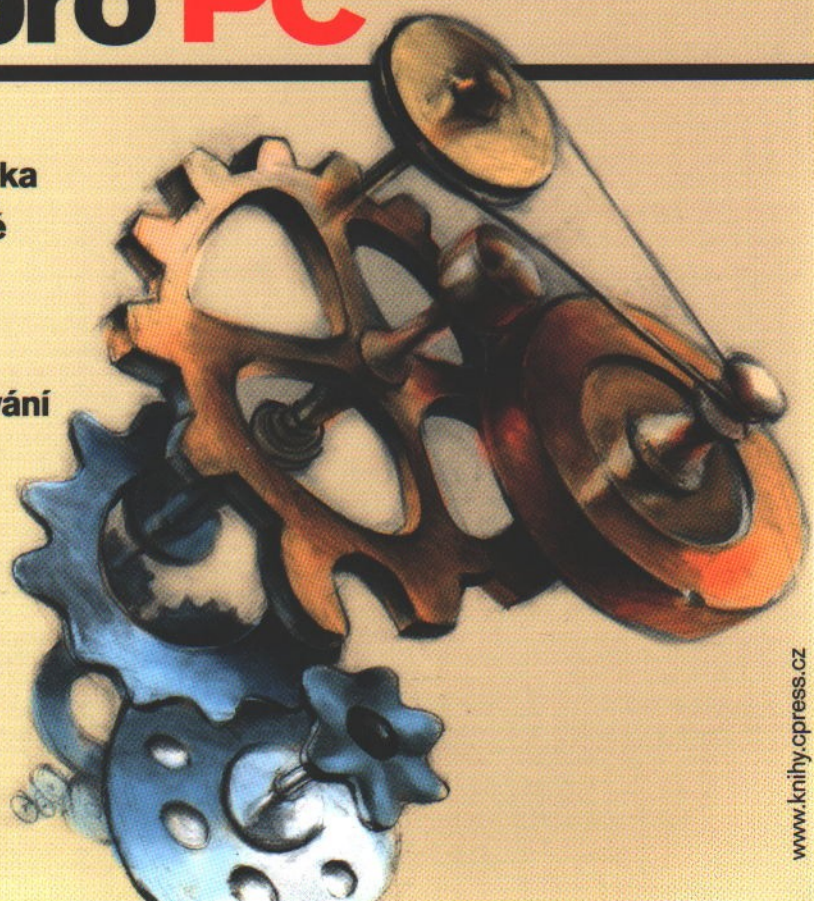
Rudolf Marek

Učíme se programovat v jazyce Assembler pro PC

- Principy nižšího programovacího jazyka
- Základní programové konstrukce
- Podrobný přehled instrukcí
- Praktické programování v DOSu/Windows i Linuxu



Na CD překladač NASM, zdrojový kód příkladů a mnoho dalších materiálů k Assembleru



computer press

www.knihy.cpress.cz

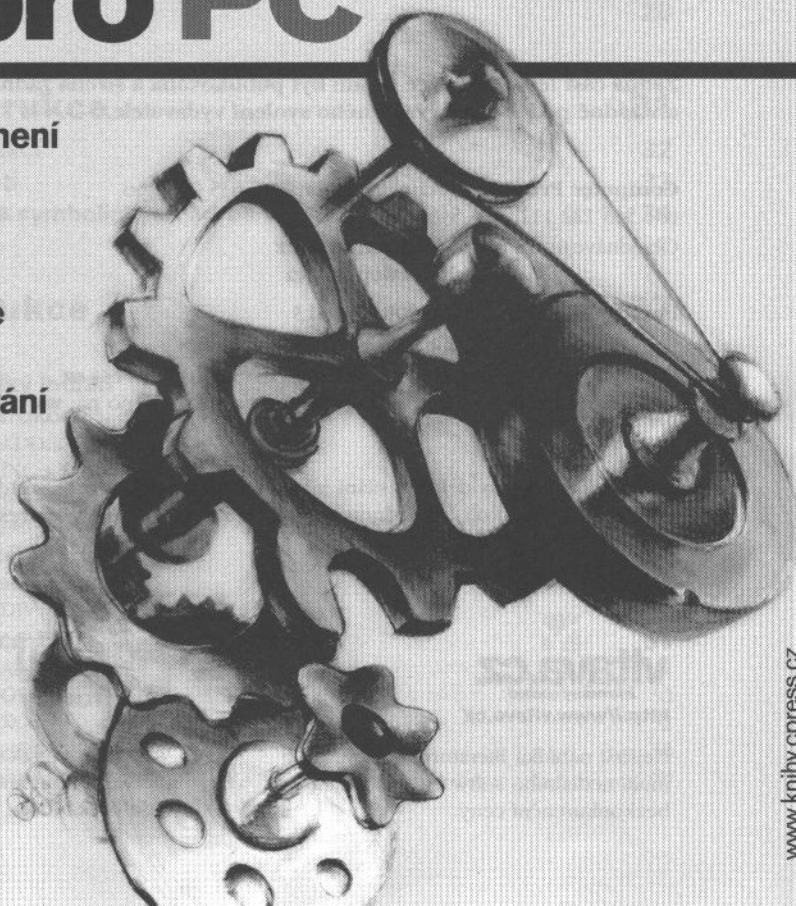
Rudolf Marek

Učíme se programovat v jazyce Assembler pro PC

Computer Press
Brno
2003

Rudolf Marek

Učíme se programovat v jazyce Assembler pro PC

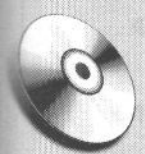


Srozumitelné seznámení s Assemblerem

Podrobný popis instrukcí

Základní programové konstrukce

Praktické programování v DOSu/Windows i Linuxu



Na CD překladač NASM, zdrojový kód příkladů a mnoho dalších materiálů k Assembleru

computer press

www.knihy.cpress.cz

Učíme se programovat v jazyce Assembler pro PC

Rudolf Marek

Copyright © Computer Press® 2003. Vydání první. Všechna práva vyhrazena.
Vydalo nakladatelství Computer Press jako svou 1275. publikaci.

Vydavatelství a nakladatelství Computer Press®,
Nám. 28. dubna 48, 635 00 Brno, http://www.cpress.cz

ISBN 80-7226-843-0

Prodejní kód: K0831

Odborná korektura: Jakub Trávník
Jazyková korektura: Marie Schreinerová
Vnitřní úprava: René Kašík
Sazba: René Kašík
Rejstřík: René Kašík
Obálka: Martin Sodomka

Komentář na zadní straně obálky: Ivo Magera
Technická spolupráce: Jiří Matoušek, Petr Klíma, Pavlína Bauerová, Petr Chládek, Mirek Zachrdle
Odpovědný redaktor: Ivo Magera
Technický redaktor: Jiří Matoušek
Produkce: Petr Baláš

Žádná část této publikace nesmí být publikována a šířena žádným způsobem a v žádné podobě bez výslovného svolení vydavatele.

Computer Press, a.s., Nám. 28. dubna 48, 635 00 Brno
tel.: 546 122 111, fax: 546 122 112
Objednávejte na: **www.knihy.cpress.cz**
distribuce@cpress.cz

Bezplatná telefonní linka: **800 555 513**

Dotazy k vydavatelské činnosti směřujte na: **knihy@cpress.cz**
Objednávat můžete na adresách vydavatelství nebo přímo na: **www.knihy.cpress.cz**

Máte-li zájem o pravidelné zasílání informací o knižních novinkách do Vaší e-mailové schránky, zašlete nám jakoukoli, i prázdnou zprávu na adresu **novinky@cpress.cz.**

vltava.cz
internetový obchod
http://www.vltava.cz

Nejširší nabídka literatury, hudby, MP3, multimediálního softwaru a videa za bezkonkurenční ceny.

knihy@cpress.cz
e-linka knihy

Vaše dotazy, vzkazy, náměty, připomínky ke knižní produkci Computer Press přijímá 24 hodin denně naše horká linka: **knihy@cpress.cz**

Obsah

Předmluva 13
Použité konvence 14

KAPITOLA 1
Základní číselné soustavy a pojmy 15
1.1 Číselné soustavy a převody 15
1.2 Datové typy 18

KAPITOLA 2
Seznámení s mikroprocesory řady x86 21
2.1 Počítač obecně 21
2.2 Historie mikroprocesorů řady x86 23
2.3 Mikroprocesor obecně 24
2.4 Mikroprocesor 80386 26
2.5 Přerušení 28

KAPITOLA 3
Anatomie instrukce 31
3.1 Operandy instrukcí 32
3.2 Adresace paměti 33
3.3 Instrukce jazyka symbolických adres 34

KAPITOLA 4
Základní instrukce 37
4.1 Instrukce MOV 37
4.2 Co je malý a velký indián 39
4.3 Aritmetické instrukce 40
4.3.1 Instrukce ADD a SUB 40
4.3.2 Instrukce INC a DEC 43
4.3.3 Záporná čísla – Čísla se znaménkem 44
4.3.4 Instrukce pro záporná čísla – NEG CBW CWD CDQ CWDE 46
4.4 Další aritmetické instrukce 48
4.4.1 Instrukce MUL a IMUL 48
4.4.2 Instrukce DIV a IDIV 50
4.5 Logické instrukce 51
4.5.1 Instrukce AND 51
4.5.2 Instrukce OR 52
4.5.3 Instrukce XOR 52
4.5.4 Instrukce NOT 52
4.5.5 Bitové vlajky – pole (mapy) 53

KAPITOLA 5
Základní programové konstrukce 55
5.1 Sekvence 55
5.2 Konstrukce "IF THEN" – selekce 56
5.2.1 Instrukce CMP a TEST 56
5.2.2 Instrukce nepodmíněného skoku JMP 57
5.2.3 Instrukce podmíněných skoků Jx 58
5.3 Konstrukce iterace – cykly 62
5.3.1 Instrukce LOOP 64
5.3.2 Instrukce LOOPZ a LOOPNZ 65
5.4 Instrukce pracující se zásobníkem 65
5.4.1 Co je a jak funguje zásobník 65
5.4.2 Instrukce PUSH a POP 67
5.4.3 Instrukce PUSHA/POPA PUSHAD/POPAD 69
5.4.4 Instrukce PUSHF/POPF a PUSHFD/POPFD 69
5.4.5 Instrukce CALL a RET 70
5.4.6 Instrukce INT a IRET 71

KAPITOLA 6
Ostatní instrukce 75
6.1 Instrukce přímo měnící registr příznaků 75
6.1.1 Instrukce CLI a STI 75
6.1.2 Instrukce STD a CLD 76
6.2 Instrukce XCHG 76
6.3 Instrukce LEA 76
6.4 Instrukce pro práci s řetězci 77
6.4.1 Instrukce STOSx 77
6.4.2 Instrukce LODSx 78
6.4.3 Instrukce CMPSx 78
6.4.4 Instrukce SCASx 78
6.4.5 Instrukce REP a REPZ 78
6.5 Instrukce pro vstup a výstup (I/O) 82
6.5.1 Instrukce IN a OUT 82
6.5.2 Instrukce NOP 84
6.6 Instrukce posuvů a rotací 84
6.6.1 Instrukce SHR a SHL 85
6.6.2 Instrukce SAL a SAR 86
6.6.3 Instrukce RCR/RCL 87
6.6.4 Instrukce ROR a ROL 87
6.7 Pseudoinstrukce 88
6.7.1 Pseudoinstrukce DB, DW a DD 88
6.7.2 Pseudoinstrukce RESB, RESW a RESD 89
6.7.3 Pseudoinstrukce TIMES 89
6.7.4 Pseudoinstrukce INCBIN 90
6.7.5 Pseudoinstrukce EQU 90
6.7.6 Operátor SEG 90
6.8 Triky s instrukcemi 91
6.8.1 Příkaz ALIGN 91
6.8.2 Naplňování registrů určitými hodnotami 92

6.8.3 Optimalizovaná aritmetika 92
6.8.4 Srovnáváme 93
6.8.5 Ostatní 93

KAPITOLA 7
Užitečné kousky kódu 95
7.1 Jednoduché příklady 95
7.1.1 Součet dvou čísel uložených proměnných 95
7.1.2 Součet čísel uložených v registrech 96
7.1.3 Součet pole čísel 97
7.1.4 Určení, zda číslo sudé nebo liché 98
7.1.5 Obrácení pořadí bitů čísla 98
7.1.6 Zjištění cifry v čísle v desítkové soustavě 99
7.2 Převody čísel na řetězec 99
7.3 Převody řetězců na čísla 104

KAPITOLA 8
Operační systém 109
8.1 Vývoj operačních systémů 109
8.2 Přidělování procesoru 110
8.2.1 Procesy 110
8.2.2 Princip plánování procesů 111
8.2.3 Reprezentace a stavy procesu 111
8.3 Správa paměti 113
8.3.1 Jednoduché přidělování paměti 113
8.3.2 Swapování 114
8.3.3 Virtuální paměť a stránkování 114
8.4 Souborové systémy 116
8.4.1 Struktura souborového systému 117
8.4.2 Mechanismy přístupu k souborům 117
8.4.3 Fyzická struktura disku 118
8.4.4 Organizace více souborových systémů na disku 119
8.5 Zavedení systému na PC 119
8.5.1 BIOS 119

KAPITOLA 9
Překladač NASM 121
9.1 Programová řádka 121
9.2 Výrazy 122
9.3 Lokální návěští 122
9.4 Preprocesor NASM 123
9.4.1 Jednořádková makra – %define, %undef 123
9.4.2 Víceřádková makra – %macro %endmacro 124
9.4.3 Deklarace makra – %assign 124
9.4.4 Podmíněný překlad – %if 124
9.4.5 Existence makra – %ifdef, %ifndef 125
9.4.6 Vlož soubor – %include 125
9.5 Direktivy Assembleru 126

9.5.1 Direktiva BITS 126
9.5.2 Direktivy SECTION a SEGMENT 127
9.5.3 Direktivy EXTERN, GLOBAL a COMMON 128
9.5.4 Direktiva CPU 128
9.6 Výstupní formáty 129
9.6.1 Výstupní formát bin 129
9.6.2 Direktiva ORG 130
9.6.3 Výstupní formát obj (Microsoft OMF) 130
9.6.4 Výstupní formát win32 131
9.6.5 Výstupní formát coff 131
9.6.6 Výstupní formát elf 131
9.6.7 Výstupní formát aout a aoutb 132
9.6.8 Symbolické informace 132

KAPITOLA 10
Programujeme v DOSu 133
10.1 Adresace paměti v reálném režimu 134
10.2 Jak vypadá uspořádání paměti v DOSu 135
10.3 Rozšíření paměti nad 1 MB 136
10.4 Druhy spustitelných souborů v DOSu 137
10.5 Základní systémová volání 139
10.5.1 Ukonči program 139
10.5.2 Vypiš řetězec 140
10.5.3 Hello world ! 140
10.5.4 Vstup z klávesnice 141
10.6 Vstupně-výstupní operace se soubory (I/O) 145
10.6.1 Otevři soubor 145
10.6.2 Uzavři soubor 146
10.6.3 Čti ze souboru 147
10.6.4 Zapiš do souboru 147
10.6.5 Inteligentní otevření/vytvoření souboru 149
10.6.6 Změň polohu v souboru (SEEK) 151
10.6.7 Ostatní funkce pro práci se soubory 152
10.6.8 Práce s dlouhými názvy souborů 154
10.7 Práce s adresáři 154
10.7.1 Vytvoř a zruš adresář (MKDIR, RMDIR) 154
10.7.2 Vyber aktuální adresář (CHDIR) 155
10.7.3 Zjisti vybraný adresář(GETCWD) 155
10.8 Alokujeme paměť 156
10.8.1 Změň velikost bloku paměti 157
10.8.2 Alokuj paměť 157
10.8.3 Uvolni paměť 158
10.9 Předávání příkazové řádky 158
10.10 Chybové kódy 159
10.11 Odlaďujeme 159
10.12 Debugger grdb.exe 160
10.13 Návrh a odladění programu 163
10.14 Rezidentní program aneb Peklo na zemi 171
10.15 Volně šiřitelné zdroje informací 175

KAPITOLA 11
Programujeme ve Windows 177
11.1 Úvod 177
11.2 Nativní aplikace pod Windows 177
11.2.1 Systémová volání API 178
11.2.2 Hello, World! 178
11.3 Kompatibilita programů 180
11.4 Spouštíme DOS 180
11.5 Volně šiřitelné zdroje informací 181

KAPITOLA 12
Programujeme v Linuxu 183
12.1 Úvod 183
12.2 Jak vypadá uspořádání paměti procesu 184
12.3 Předávání příkazové řádky a prostředí 185
12.4 Volání operačního systému 185
12.5 Chybové kódy 186
12.6 Manuálové stránky 186
12.7 Hello world! 188
12.8 Asmutils aneb jak si ulehčit práci 189
12.9 Práce s makry Asmutils 191
12.10 Vstupně-výstupní operace se soubory (I/O) 191
12.10.1 Otevři soubor 191
12.10.2 Uzavři soubor 193
12.10.3 Čti ze souboru 193
12.10.4 Zapiš do souboru 193
12.10.5 Změna polohy v souboru 196
12.10.6 Ostatní funkce pro práci se soubory 197
12.11 Práce s adresáři 198
12.11.1 Vytvoř a zruš adresář (MKDIR, RMDIR) 199
12.11.2 Vyber aktuální adresář (CHDIR) 199
12.11.3 Zjisti vybraný adresář(GETCWD) 199
12.12 Vstup z klávesnice 200
12.12.1 Systémové volání IOCTL 200
12.13 Alokace paměti 201
12.14 Odlaďujeme 202
12.15 Assembler GAS 204
12.16 Volně šiřitelné zdroje informací 205
12.17 Možnosti překladače a linkeru 206

KAPITOLA 13
Propojujeme 207
13.1 Předávání proměnných 207
13.2 Co je stackframe? 208

13.2.1 Stackframe v jazyce C (32bitová verze) 209
13.2.2 Stackframe v jazyce C (16bitová verze) 212
13.3 Propojujeme s jazykem C 212
13.4 Propojujeme s jazykem Pascal 215

KAPITOLA 14
Slovo závěrem 219

KAPITOLA 15
Nejčastěji používané instrukce 221
Rejstřík 223

PŘEDMLUVA

Kniha, jejíž první řádky právě čtete, je učebnicí programovacího jazyka Assembler. Svým zaměřením je určená všem zájemcům o tento nižší programovací jazyk, zejména pak naprostým začátečníkům, kteří mají chuť se s Assemblerem blíže seznámit a naučit se v něm řešit úlohy, které nelze pomocí vyšších jazyků zvládnout.

Programování v jazyce Assembler nelze oddělit od fyzické architektury počítače a často ani od operačního systému, pod kterým budeme chtít programy provozovat. V knize se seznámíme s instrukční sadou mikroprocesoru 80386, se kterým jsou kompatibilní všechny procesory v počítačích PC na našem trhu. Naučíme se psát programy pod operační systém DOS, Windows a Linux.

Postupovat budeme krůček po krůčku, naučíme se pracovat s šestnáctkovou a dvojkovou soustavou, popíšeme si architekturu procesoru 80386 i jeho instrukce. Jakmile budeme disponovat dostatečným stupňem znalostí, ukážeme si základní programové konstrukce, tak abychom byli schopni vytvořit podmínky, cykly a podprogramy.

Prvními skutečnými programy budou ty nejjednodušší, platformově nezávislé, ve kterých se naučíme řešit základní problémy (součet čísel uložených v paměti apod). Postupně si vytvoříme sadu užitečných podprogramů, kterou budeme používat i v dalších příkladech (převody čísel na řetězce a naopak).

Seznámíme se s vývojem i strukturou moderních operačních systémů. Stručně si popíšeme principy správy paměti, procesů i souborového systému.

Programy zapsané v Assembleru budeme muset přeložit, budeme používat volně šiřitelný překladač NASM, který je platformově (systémově) nezávislý. Probereme si všechny důležité rysy překladače (preprocesor, direktivy) i produkované výstupní formáty, tak abychom mohli překladač NASM používat s celou řadou operačních systémů i jiných programovacích jazyků.

V dalších kapitolách se seznámíme s prostředím jednotlivých operačních systémů i funkcí, které programátorovi v Assembleru nabízejí. Naučíme se vypisovat řetězce na obrazovku, načítat je z klávesnice, alokovat novou paměť i pracovat se soubory a adresáři. Všechny nové znalosti budeme aplikovat na četných příkladech. Naučíme se programy nejen psát, ale i odlaďovat a hledat chyby.

V poslední kapitole si ukážeme způsoby propojení programů v jazyce Assembler s vyššími programovacími jazyky – Pascal a C.

Na přiloženém CD najdete všechny programy (včetně překladače) i další zdroje informací.

Chtěl bych poděkovat Tomášovi Borlandovi Valentovi za nakreslení obrázků pro tuto knihu, děkuji také Michalovi Miškovi Medveckému za četné připomínky k obsahu knihy.

Rudolf Marek, autor

Použité konvence

Text psaný tímto stylem je text výkladu.

```
Pro programový kód bude použit tento styl.

mov eax,0xDEADBEEF  ;a bude mít komentář
mov ebx,eax
```

Text zapsaný tímto stylem označuje "obrazovku počítače" - okno s příkazovou řádkou.

ZÁKLADNÍ ČÍSELNÉ SOUSTAVY A POJMY

1

V této kapitole:

1.1 Číselné soustavy a převody
1.2 Datové typy

Architektura počítačů je spjata s dvojkovou číselnou soustavou, která obsahuje jen číslice 0 a 1. Dvojková (binární) soustava je pro počítače z konstrukčního hlediska výhodná, protože obě možné číslice lze „zakódovat" stavy vypnuto a zapnuto.

Jak se sami přesvědčíme, zápisy „větších" čísel ve dvojkové soustavě budou neúměrně narůstat, a proto se podrobně seznámíme s osmičkovou a šestnáctkovou soustavou. Oba základy soustav jsou mocninou čísla dvě, a tak nám umožní kratší a výstižnější zápis čísel.

1.1 Číselné soustavy a převody

Číselné soustavy se liší nejen počtem jednotlivých číslic, ale i vlastní konstrukcí. Vzpomeňme si například na soustavu, ve které počítali staří Římané, a kterou proto dnes známe jako soustavu římskou. Špatně se v ní počítalo a nešla ani zobrazit nula.

Desítková číselná soustava i ostatní, se kterými budeme pracovat, patří do skupiny polyadických soustav, kde lze libovolné číslo zapsat jako součet součinů jednotlivých číslic (a_n) a mocnin základu soustavy (z).

$a = a_n * z^n + a_{n-1} * z^{n-1} + ... + a_1 * z^1 + a_0 * z^0$ (n je počet číslic čísla)

Desítkové číslo 1 234 můžeme zapsat jako:

$1234 = 1*10^3 + 2*10^2 + 3*10^1 + 4*10^0$. Přičemž $10^0 = 1$.

Každá číslice má různou váhu, podle toho, na kterém místě v čísle je číslice obsažena. Velikost váhy je mocninou základu soustavy.

Pracujeme-li s různými číselnými soustavami, uvádíme čísla většinou v závorce s dolním indexem, kterým naznačíme základ soustavy. Číslo $(1100)_2$ je zapsané ve dvojkové číselné soustavě, a pokud ho rozepíšeme podle vzorce, dostaneme:

$(1100)_2 = 1*2^3 + 1*2^2 + 0*2^1 + 0*2^0$,

provedeme-li součet 8 + 4, dostaneme $(1100)_2 = 12$, číslo jsme převedli do desítkové soustavy. Všimněme si, že na dvanáctku jsme ve dvojkové soustavě potřebovali už čtyři řády. Nejvyšší číslo se čtyřmi řády bude 15, protože $(1111)_2 = 1*8 + 1*4 + 1*2 + 1*1 = 15$. Vypišme si prvních šestnáct dvojkových čísel:

Desítkově	Dvojkově	Desítkově	Dvojkově
0	0	8	1000
1	1	9	1001
2	10	10	1010
3	11	11	1011
4	100	12	1100
5	101	13	1101
6	110	14	1110
7	111	15	1111

Všimněme si, že čísla rostou pravidelně, po 11 následuje 100 atd. Proto i bez převádění správně odhadneme, že 16 zapíšeme jako $(10000)_2$.

Osmičkovou soustavu (základem soustavy je číslo 8) tvoří osm číslic 0 – 7. Pro převod čísel z osmičkové soustavy použijeme stejný princip jako při převodu z dvojkové (binární) soustavy.

číslo $(77)_8 = 7*8^1 + 7*8^0 = 63$

Osmičková číselná soustava byla velmi populární u majitelů tzv. osmibitových počítačů ATARI, PMD-85 atd. Její sláva pomalu ustoupila šestnáctkové soustavě, kterou budeme používat pro zápis čísel na PC i my.

V šestnáctkové soustavě nejsou pro číslice větší než devět zavedeny žádné speciální znaky, a proto se užívá prvních šest znaků abecedy.

A = 10, B = 11, C = 12, D = 13, E = 14, F = 15

Kdybychom snad chtěli věk uvádět v číselné soustavě o základu šestnáct, pak bychom občanský průkaz dostali v F a plnoletost by nastala ve 12 letech.

I pro šestnáctkovou soustavu zůstává princip převodu čísel do desítkové soustavy stejný:

číslo $(524D)_{16} = 5*16^3 + 2*16^2 + 4*16^1 + 13*16^0 = 20\,480 + 512 + 64 + 13 = 21\,069$

číslo $(DEAD)_{16}$ = $13*16^3 + 14*16^2 + 10*16^1 + 13*16^0$ = 57 005

číslo $(DEADBEEF)_{16}$ = $13*16^7 + 14*16^6 + 10*16^5 + 13*16^4 + 11*16^3 + 14*16^2 + 14*16^1 + 15*16^0$ = 3 735 928 559

číslo $(C001)_{16}$ = $12*16^3 + 0*16^2 + 0*16^1 + 1$ = 49 153

Prozatím jsme se naučili převádět čísla z libovolné polyadické soustavy o základu n (2, 8 a 16) do soustavy desítkové.

Jestliže budeme chtít převést číslo z desítkové soustavy do soustavy o základu n, budeme převáděné číslo opakovaně dělit n a zbytky sepisovat (to budou jednotlivé číslice, tedy výsledek). Celý proces budeme opakovat, dokud nebude výsledek z předchozího dělení nula.

Příklad: Číslo 14 převeďte do dvojkové soustavy!

14/2 = 7 zbytek **0**

7/2 = 3 zbytek **1**

3/2 = 1 zbytek **1**

1/2 = 0 zbytek **1**

Poslední výsledek je 0, a dělení je tedy ukonce. Zbytky sepíšeme zprava doleva a výsledek je $(1110)_2$.

Příklad: Číslo 13 převeďte do dvojkové soustavy!

13/2 = 6 zbytek **1**

6/2 = 3 zbytek **0**

3/2 = 1 zbytek **1**

1/2 = 0 zbytek **1**

Poslední výsledek je 0 a dělení je tedy ukonce. Zbytky opět zapíšeme zprava doleva (tj. tak, aby zbytek, který je na prvním místě v řadě byl nejvíce napravo) a výsledkem je číslo $(1101)_2$

Příklad: Převeďte číslo 123456 do šestnáctkové soustavy

123456/16 = 7716 zbytek **0**

7716/16 = 482 zbytek **4**

482/16 = 30 zbytek **2**

30/16 = 1 zbytek 14 = **E**

1/16 = 0 zbytek **1**

Poslední výsledek je 0, a dělení je tedy ukonce. Sepíšeme zbytky a dostáváme výsledek 123 456 = $(1E240)_{16}$.

Zápis se závorkou a základem soustavy je pro programování v Assembleru nepohodlný a proto se ujala následující konvence:

- Čísla šestnáctková začínají 0x nebo $0. Šestnáctkové číslo lze zapsat také tak, že za číslo připojíme písmeno h – 1234h. Aby bylo překladači jasné, že se jedná

o číslo, a nikoliv o identifikátor (proměnnou), je nutné před číslo vložit nulu. Např.: 0DEADh.

0x1234, $01234 a 01234h je $(1234)_{16}$

- Čísla desítková začínají číslicí a mohou končit písmenem d. 1234, 1234d je $(1234)_{10}$
- Čísla dvojková začínají číslicí (0 nebo 1) a končí písmenem b. 1100b je $(1100)_2$
- Čísla oktalová (v osmičkové soustavě) začínají číslicí a končí písmenem q. 12q je $(12)_8$

Šestnáctková čísla budeme zapisovat jako 0x... a dvojková čísla zase jako ...b. Desítková čísla budeme používat tak, jak jsme zvyklí – bez jakýchkoliv úprav (Pozor! Některé assemblery berou zápis bez prefixu jako šestnáctkové číslo!).

1.2 Datové typy

Základní a nedělitelnou jednotkou informace je jeden bit. Slovo bit je odvozeno z anglického binary digit – binární číslice. Ve dvojkové (binární) soustavě může číslice nabývat pouze hodnot nula nebo jedna, jedná se o elementární rozhodnutí stavu pravda – nepravda, vypnuto – zapnuto. Na dvouhodnotové logice jsou postaveny všechny logické obvody a z logických obvodů zase počítače, proto se seznámíme s pojmem bit podrobněji.

Počet bitů dvojkového čísla je stejný jako počet jeho číslic (cifer). Obecněji můžeme podle počtu bitů určit rozsah hodnot, které může číslo nabývat. Budeme-li uvažovat celé nezáporné čtyřbitové dvojkové číslo, bude nabývat 2^4 – (tedy šestnácti) různých hodnot.

Číslovaní (pořadí) bitů v čísle se určuje od nejnižších řádů k řádům nejvyšším (zprava doleva). *Nejpravější bit je bit číslo nula (první bit).* Současně je první bit bitem *nejméně významným*, anglicky se označuje jako LSB (least significant bit). *Nejvyššímu bitu* (řádu) v čísle se říká *nejvíce vyznamný* – MSB (most significant bit).

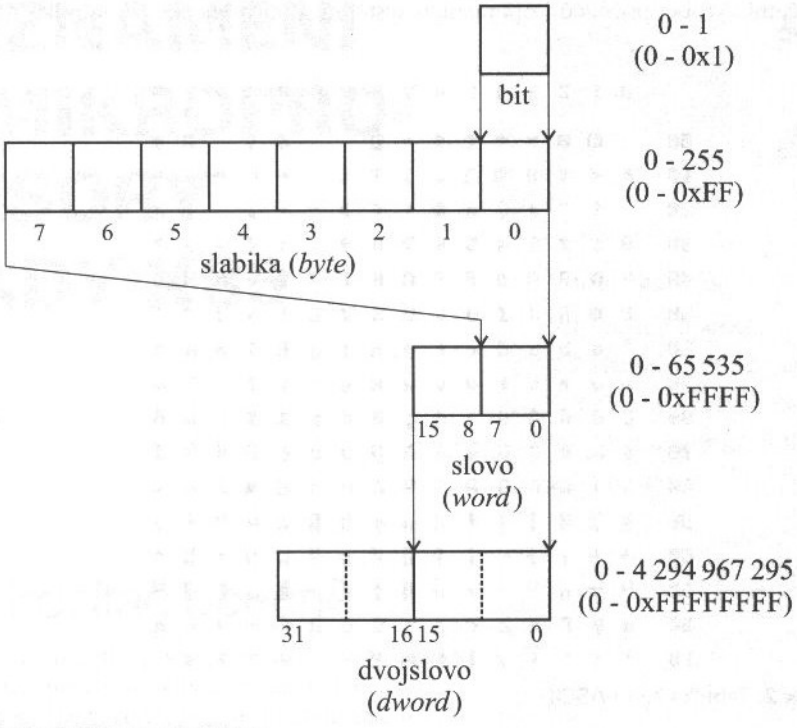
Bity čísla můžeme seskupit. Skupině čtyř bitů se říká nibble (housenka). Počítač nepracuje s jednotlivými bity, ale s jejich skupinami, např. osm bitů tvoří elementární datový typ, označovaný jako *byte* (slabika, bajt). Proč má byte právě osm bitů? V žádném případě se nejedná o přírodní zákon, odpověď hledejme u vývojářů firmy IBM, projektující první počítače.

Jednotlivé byty můžeme seskupit a utvořit tak *slovo* a *dvojslovo* (word, dword). Pro počítače PC platí:

1 byte = 8 bitů

1 slovo (word) = 2 byte = 16 bitů

1 dvojslovo (dword) = 4 byte = 32 bitů



Obrázek 1: Od bitu po dvojslovo

Jeden byte je zároveň nejmenší množství informace, kterou můžeme z paměti přečíst nebo do paměti zapsat (paměť se adresuje po jednotlivých bytech). V jednom bytu můžeme uložit číslo 0 – 255 (2^8 = 256 různých čísel). Nestačí-li tento rozsah, můžeme použít slovo, které má rozsah 0 – 65 535 (2^{16} = 65 536 čísel), nebo dvojslovo s rozsahem 0 – 4 294 967 295 (2^{32} = 4 294 967 296 různých čísel).

Kapacity paměťových médií se v prvopočátcích počítačů udávaly v bytech (1 B). S rostoucí kapacitou počítačových pamětí se objevily i odpovídající násobné jednotky bytu. První jednotkou, stejně jako v soustavě SI, je jedno kilo [k]. Počítače pracují s dvojkovou soustavou, a proto předpona kilo [k] není přesně 1 000, ale 1 024 B (což je 2^{10}). Ostatní předpony, mega [M], giga [G], tera [T] a peta [P], rostou geometrickou řadou tak, že každá další jednotka je 1 024x mohutnější než jednotka předcházející (Obdobně se používají násobné jednotky pro jednotky bitů kb (kilobit – 1 024 bitů), Mb (megabit) atd. Všiměme si, že byte označujeme velkým B, bit zase malým b.). Vraťme se zpět k datovým typům.

Ve vyšších programovacích jazycích existuje datový typ pro uložení znaků a řetězců. V Assembleru žádný speciální datový typ pro uložení znaků nebo řetězců neexistuje. Místo toho se používá byte pro uložení znaku a posloupnost bytů pro uložení řetězce. Zobrazení znaků do jednotlivých hodnot bytu zajišťuje norma ASCII (American Standard Code for Information Interchange). Prvních 128 řídicích znaků a symbolů je respekto-

váno všemi výrobci počítačů i operačních systémů. Podívejme se, jak tabulka ASCII vypadá:



Obrázek 2: Tabulka kódu ASCII

V záhlaví tabulky jsou použita hexadecimální čísla indikující číselnou hodnotu zobrazeného znaku na vybrané pozici. Například znak 'A' leží na souřadnicích 40, 01. Sečteme-li hodnoty 40 a 1, dostaneme v šestnáctkové soustavě hodnotu 0x41 (desítkově 65), která reprezentuje znak velké 'A' v kódu ASCII.

Tisknutelné znaky v tabulce začínají ASCII kódem 0x20 (desítkově 32,). Znaky s nižším kódem jsou tzv. znaky řídicí. Většina z nich má své jméno, my si však představíme pouze dva. Znak s číselnou hodnotou 0xA, LF – line feed (odřádkování) – a znak s číselnou hodnotou 0xD, nazývaný CR (carriage return, návrat vozíku).

Nejdůležitějším řídicím znakem je oddělovač řádku. Z jazyka C tento znak znáte pod symbolem \n. Naneštěstí se oddělovač řádku liší v závislosti na použitém operačním systému. V OS Windows (a DOS) tvoří oddělovač řádku posloupnost dvou znaků (CR, LF – 0xD, 0xA), v operačních systémech UNIX jen jeden znak (LF – 0xA).

Znaky obsažené v tabulce od hodnoty 128 (řádek tabulky začínající 0x80 a představující rozšíření původních 128 znaků na 256) se staly obětí různých standardů a norem. Americká norma samozřejmě neobsahovala znaky národních abeced (kromě abeced „západní Evropy"), a tak se pro zobrazení diakritiky (speciálních písmen národních abeced), tam kde bylo třeba, použily některé „nepotřebné" znaky z horních 128.

Pro češtinu se v současné době používají dva druhy kódování – norem. Norma ISO-8859-2 (standard organizace ISO je běžná v operačních systémech UNIX) a vlastní „norma" firmy Microsoft, CP 1250, je používaná v OS Windows.

SEZNÁMENÍ S MIKROPROCESORY ŘADY X86

2

V této kapitole:

2.1 Počítač obecně
2.2 Historie mikroprocesorů řady x86
2.3 Mikroprocesor obecně
2.4 Mikroprocesor 80386
2.5 Přerušení

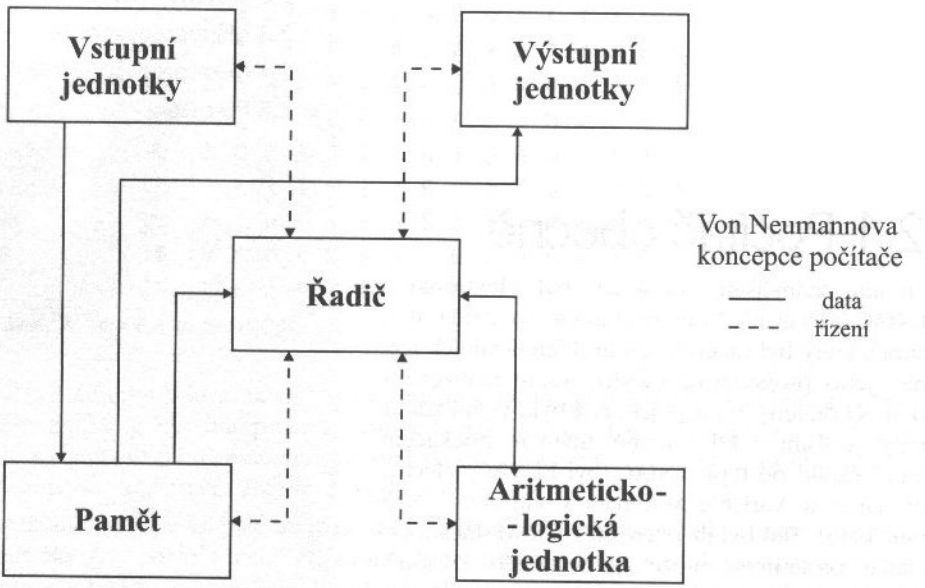
2.1 Počítač obecně

Prvním známějším počítačem byl elektronkový ENIAC (Electronic Numerical Integrator And Calculator), který byl určen k řešení diferenciálních rovnic. Jeho programování bylo ovšem realizováno pomocí drátěných propojek, což bylo velmi zdlouhavé. Dalším, i když méně známým počítačem, který vznikl po typu ENIAC, byl EDVAC (Electronic Discrete Variable Automatic Computer – vyroben 1946). Ten byl již řízen, stejně jako dnešní počítače, programem uloženým v paměti. Zasloužil se o to americký matematik maďarského původu John Ludwig von Neumann. Jeho koncepce je založena na těchto základních principech[1]:

1) Počítač tvoří tyto jednotky: řadič, aritmetická jednotka, paměť, vstupní a výstupní jednotky.
2) Struktura počítače je nezávislá na typu řešené úlohy, počítač se programuje obsahem paměti.
3) Instrukce a operandy (data, s nimiž instrukce pracují) jsou uloženy v téže paměti (Harvardská koncepce počítače, vycházející z von Neumannovy, oddělila paměť pro program od paměti pro data).

[1] Prof. Ing. Jan Hlavička, DrSc. strana 8-9, dotisk druhého vydání skript Architektura počítačů.

4) Paměť je rozdělena do buněk stejné velikosti, jejich pořadová čísla se používají jako adresy (1 buňka je jeden byte).
5) Program je tvořen posloupností elementárních příkazů (instrukcí), v nichž zpravidla není obsažena hodnota operandu (uvádí se pouze jeho adresa), takže program se při změně dat nemění. Instrukce se provádějí jednotlivě v pořadí, v němž jsou zapsány do paměti (dnešní mikroprocesory mimo jiné umožňují současné zpracování více instrukcí).
6) Změna pořadí provádění instrukcí se vyvolá instrukcí podmíněného nebo nepodmíněného skoku.
7) Pro reprezentaci instrukcí i čísel (operandů, výsledků, adres) se používají dvojkové signály a dvojková číselná soustava.



Obrázek 3: Von Neumannova koncepce počítače

Koncepce vešla ve známost jako von Neumannova koncepce počítače. Ukázala se natolik schopná a univerzální, že i dnešní počítače jsou založeny na jejích principech. Jak se počítače rozšířily do domácností?

Byla to dlouhá cesta, trvající téměř čtyřicet let. V padesátých letech minulého století byl vynalezen tranzistor, který nahradil velké a poruchové elektronky. Miniaturizace tranzistorů pokračovala, ovšem jejich nejobjemnější součástí zůstávalo jejich pouzdro. Řešení tohoto problému bylo jednoduché, do jednoho pouzdra se vložilo víc tranzistorů a tak se zrodily integrované obvody (chipy). Díky integrovaným obvodům se sice počítače zmenšily, ale stále zaujímaly celá poschodí budov. Tehdejší počítače byly velice choulostivé. Prostředí, ve kterém se nacházely, muselo být vybaveno výkonnými klimatizačními jednotkami.

Koncem sedmdesátých let počítačové čipy (integrované obvody) zlevnily natolik, aby si je mohli koupit i „obyčejní lidé". Možná se divíte, proč si nekoupili nějaký osobní počítač. To proto, že žádný v té době dosud neexistoval. A tak se jednoduché osmibitové počítače vyráběly doma na koleně (začala tak i firma Apple ve slavné garáži). Brzy se chopily příležitosti i další firmy a začaly vyrábět osmibitové počítače, které bylo možné zapojit k domácímu televizoru a hrát na nich hry nebo programovat. I firma IBM, která tehdy vyráběla velké sálové počítače, přišla v roce 1981 na trh s osobním počítačem IBM PC XT, který byl osazen osmibitovým mikroprocesorem Intel 8088, a tímto odstartovala dodnes trvající řadu počítačů PC.

2.2 Historie mikroprocesorů řady x86

Historie prvního šestnáctibitového mikroprocesoru řady x86, 8086 začíná jeho uvedením na trh firmou Intel už v roce 1978. Pracoval na taktovací frekvenci 5,8 nebo 10 MHz. Mikroprocesor umožňoval adresaci až 1 MB operační paměti (díky své dvacetibitové adresové sběrnici).

Tehdy byly populární osmibitové osobní počítače, proto Intel vyrobil mikroprocesor 8088, který byl programově kompatibilní s 8086, ale měl jen osmibitovou sběrnici, takže se k němu daly připojit podpůrné obvody populárního osmibitového procesoru 8080.

V roce 1982 uvedl Intel mikroprocesor 80286, který byl zpětně kompatibilní s oběma předcházejícími mikroprocesory, měl však širší adresovou sběrnici (24 b), takže mohl pracovat až s 16 MB operační paměti. Kromě rozšíření instrukční sady o pár nových instrukcí přinesl dva nové režimy procesoru – reálný režim a chráněný režim. Chráněný režim mikroprocesoru představoval mechanismus ochrany paměti, stránkování, přístupových práv a přepínání úloh, tedy nezbytných činností, které musí vykonávat každý víceúlohový operační systém. Reálný režim sloužil k zachování kompatibility s předchozími procesory x86.

O čtyři roky později (1986) Intel vyrobil mikroprocesor 80386 DX, který měl obě sběrnice (datovou i adresovou) plně 32bitové. Prodával se i 80386 SX, který byl uvnitř procesoru stejný jako 80386 DX, ovšem jeho vnější datová sběrnice byla jen šestnáctibitová. Tyto procesory pracovaly na taktovací frekvenci 20, 25 a 33 MHz. 80386 neměl matematický koprocesor integrován v pouzdře, tvořil ho samostatný čip 80387. V roce 1989 následovala další generace procesorů Intel – 80486DX a 80486DX/2 a DX/4, které se lišily taktovací frekvencí. Procesor 80486SX neměl na rozdíl od 80486DX integrován matematický koprocesor. Díky postupující integraci byla implementována i vyrovnávací paměť (8 kb) přímo na čipu procesoru.

V roce 1993 Intel uvedl první mikroprocesor Pentium, který jako první z řady svém počítači PC dodnes spustíte programy napsané o dvacet let dříve pro 8086.

Mikroprocesory x86 kompatibilní nevyráběl jen Intel, ale například i společnosti AMD, Cyrix, NEC nebo IBM. My se podrobněji seznámíme s 80386, protože je z hlediska programátora v Assembleru zcela kompatibilní i s dnešními mikroprocesory.

2.3 Mikroprocesor obecně

Ještě jsme si neřekli, jak vypadá mikroprocesor. Je to křemíková destička s logickými obvody (ty jsou složeny z transistorů), zalitá v plastovém pouzdře s vývody (kterým se říká piny). Na piny procesoru jsou napojeny obecně tři typy sběrnic – adresová, datová a řídicí. Tyto sběrnice propojují mikroprocesor se zbytkem počítače. Zbytek pinů slouží k napájení samotného procesoru. Sběrnice se skládá ze souboru vodičů, které plní nějakou funkci. V bodě sedm von Neumannovy koncepce se praví:

> **Pro reprezentaci instrukcí i čísel (operandů, výsledků, adres) se používají dvojkové signály a dvojková číselná soustava.**

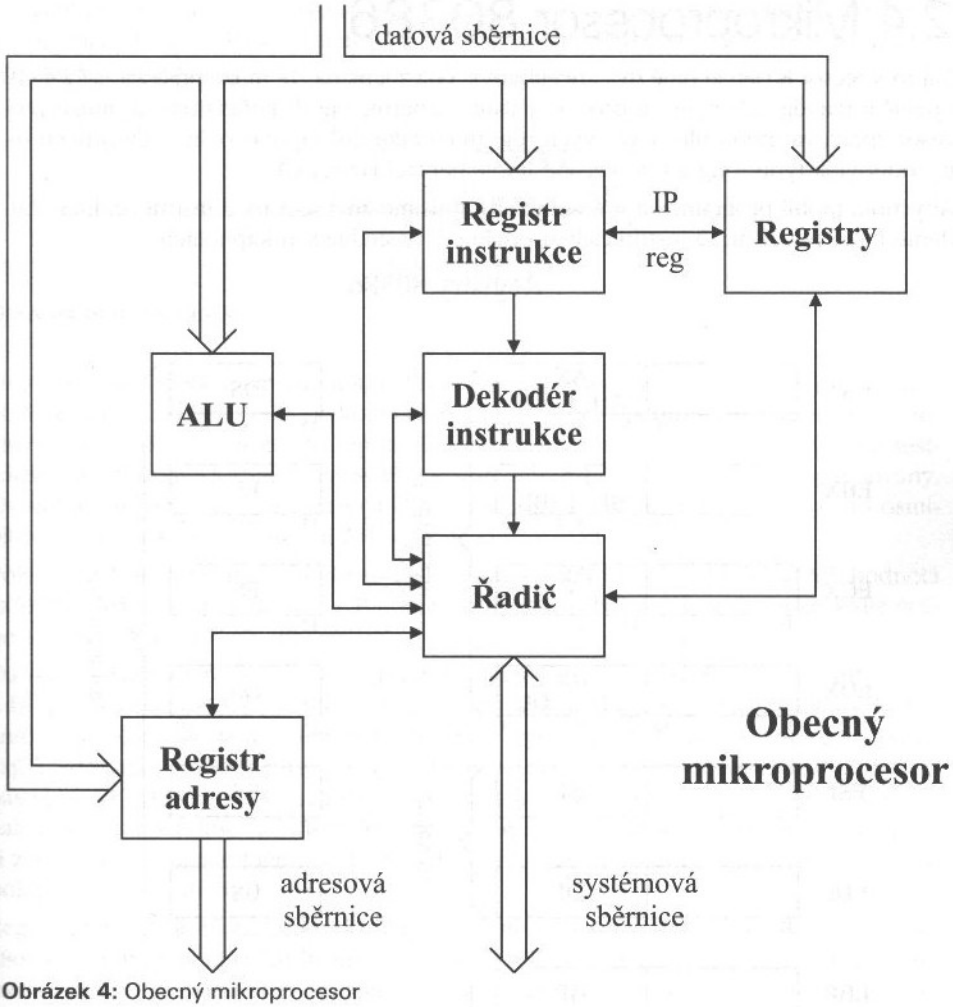
To znamená, že jeden vodič sběrnice počítače může současně přenášet jeden bit. (Rozlišuje se logická 1 a logická 0, podle toho, jaké napětí je na vodiči přítomno.) Proto má-li procesor jednu 16bitovou a druhou 8bitovou sběrnici, musí mít k pinům procesoru připojeno 16 a 8 (různých) vodičů, které ji budou tvořit. Pokud se bude po 8-bitové sběrnici přenášet například číslo 27 (binárně 00011011), bude na vodiči, který přenáší nejnižší („nejpravější") bit, sběrnice logická 1, na dalším vodiči také logická 1, na dalším logická 0 atd.

Zatím jsme si řekli, že v procesor je tvořen logickými obvody. Jejich prostřednictvím je realizován samotný mikroprocesor, který se obvykle skládá ze tří hlavních částí: řadiče, aritmeticko-logické jednotky a registrů.

Řadič se stará o načtení instrukce z paměti a o její dekódování. Dekódováním instrukce řadič zjistí, jaká instrukce se bude zpracovávat. Samotný řadič ji neprovádí, ale zadává povely (pomocí vnitřní řídicí sběrnice) ostatním jednotkám, které vykonají požadovanou činnost.

Aritmeticko-logická jednotka (ALU – Arithmetic/Logic Unit) je určena pro aritmeticko-logické operace s daty. V jednodušších mikroprocesorech aritmeticko-logická jednotka vystačí s negací a sčítáním, na které se dají ostatní aritmetické operace převést (odčítání, násobení a dělení v celočíselném oboru). Druhá část aritmeticko-logické jednotky umožňuje základní logické operace s daty – např. logický součet, součin (OR, AND) a non-ekvivalenci (XOR). Součástí jednotky je i tzv. barrel-shifter (válcový posouvač), který provádí různé druhy bitových posunů vlevo a vpravo.

Doba, kterou mikroprocesor potřebuje pro provedení instrukce, je mnohonásobně kratší než vybavovací doba pamětí (vybavovací doba je čas, za který je paměť schopna vydat data). Aby procesor nemusel s téměř každou instrukcí čekat na paměť, je vybaven dočasným úložištěm dat – registry, které mají svoji velikost (kapacitu) omezenou jen na několik bajtů, ale data jsou k dispozici „téměř okamžitě".



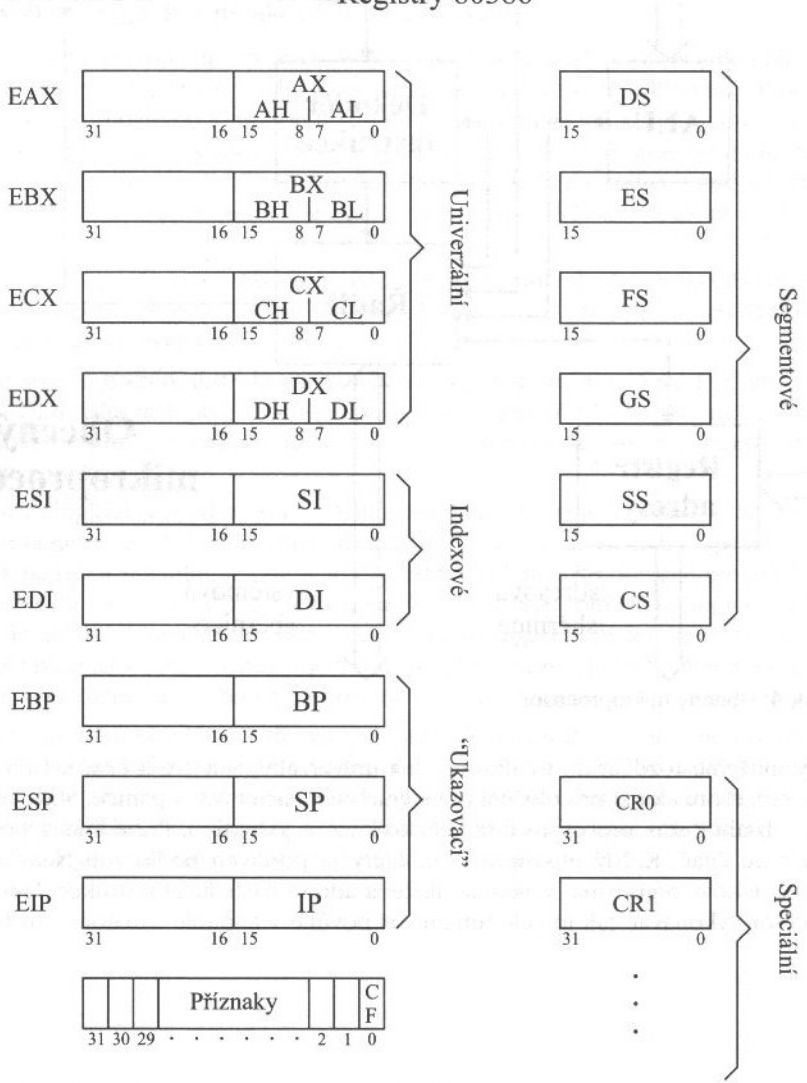
Obrázek 4: Obecný mikroprocesor

Registry můžeme rozdělit do tří skupin – na univerzální, stavové a čítače. Univerzální registry procesoru slouží pro uložení pracovních dat načtených z paměti. Stavové registry zase obsahují stav procesoru (aritmeticko-logické jednotky). Posledním typem jsou registry typu čítač. Každý mikroprocesor, který je postaven podle von Neumannovy koncepce, takový registr má. V něm je uložena adresa následující instrukce, kterou bude procesor vykonávat. Jak to celé funguje, si povíme v kapitole „Anatomie Instrukce".

2.4 Mikroprocesor 80386

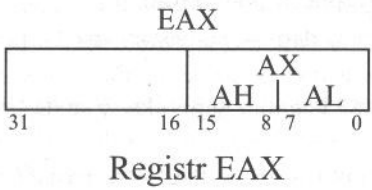
Mikroprocesor 80386 je plně dvaatřicetibitový, což znamená, že může pracovat až z 4 GB operační paměti (2^{32} byte), a protože i datová sběrnice je dvaatřicetibitová, může procesor zpracovat nebo uložit ve svých registrech také číslo (data) o šířce dvaatřiceti bitů (odpovídá typu long int ve většině implementací jazyka C).

Abychom mohli programovat v Assembleru, musíme znát registry a instrukce, které budeme moci používat. O instrukcích si povíme v následujících kapitolách.



Obrázek 5: Vybrané registry mikroprocesoru 80386

Seznámíme se nejprve se čtveřicí univerzálních registrů. Jmenují se EAX, EBX, ECX a EDX (accumulator, base, counter a data). Až na své názvy se nijak neliší, a proto si povíme jen o EAX.



Obrázek 6: Registr EAX

Mikroprocesor 80386 je zpětně kompatibilní s 80286, který měl registry nejvýše šestnáctibitové; jak je možné, že lze vykonávat instrukce, které operují s těmito „menšími" registry i na 80386? To proto, že registr EAX lze rozdělit na dvě podmnožiny – na šestnáctibitový registr AX (ten je na 80286) a horních 16 bitů, které nejsou pojmenovány. Ovšem registr AX, stejně tak jako na 80286, lze rozdělit na podmnožiny – na dva osmibitové registry, které se jmenují AH a AL.

Pokud bychom uložili do registru EAX hodnotu 0x12345678, bude v AX hodnota 0x5678 v AH 0x56 a v AL 0x78. „Podmnožiny" ostatních registrů se jmenují podle svého registru EBX – BX BL, BH atd.

Dalšími registry, které 80386 nabízí programátorovi, jsou registry ESI, EDI a EBP (SI, DI a BP pro šestnáctibitový přístup). Existují jen v šestnáctibitové a dvaatřicetibitové variantě přístupu. Společným rysem této skupinky registrů je skutečnost, že se často využívají k adresaci paměti (přístup do polí, indexy atd. – source index, destination index, base pointer). Trojici těchto registrů lze využít i k uložení nějakých dat, nikdo nám nenakazuje, že musí skutečně obsahovat nějakou adresu. Všechny registry, o kterých jsme si zatím řekli, můžeme měnit podle našich potřeb. Ostatní registry je dobré nechat „napokoji" (bez povšimnutí).

Registr ESP (SP) je ukazatelem do paměti, kde je uložen vrchol zásobníku (procesory x86 kompatibilní nemají hardwarový zásobník). Se zásobníkem se podrobněji seznámíme v kapitole o instrukcích pracujících se zásobníkem. Posledním z registrů, který se jako jediný nedá přímo měnit, je registr EIP (IP, instruction pointer). Ten ukazuje do paměti na instrukci, která se bude zpracovávat jako následující. Obsah tohoto registru mění přímo řadič mikroprocesoru podle toho, jak vybírá instrukce z paměti.

Poslední skupinou registrů jsou tzv. segmentové registry, které se podílejí na výpočtu skutečné adresy (adresy, která se bude vysílat po adresové sběrnici). Výpočet skutečné adresy se liší podle módu procesoru (chráněný/reálný) a princip výpočtu si ukážeme v některé z následujících kapitol. Tyto registry jsou stejně tak jako na 80286 pouze šestnáctibitové. Jejich názvy jsou odvozeny podle spoluúčasti na adresaci. Registr CS (code segment, kódový segment) společně s registrem EIP (IP) určuje adresu místa v paměti, odkud se bude načítat následující instrukce. Podobně registr SS (stack segment, zásobníkový segment) a jeho pár SS:ESP (SS:SP) tvoří společně výslednou adresu, na které je umístěn vrchol zásobníku. Segmentové registry DS, ES, FS a GS (data, extra, F a G segment) se účastní adresace dat v paměti.

Úplně posledním registrem mikroprocesoru, o kterém se zmíníme, je registr příznaků – EFLAGS (FLAGS). Obsahuje jednobitové příznaky indikující převážně stav aritmeticko-logické jednotky mikroprocesoru. Seznámíme se zatím jen s několika z 32 příznaků, které budeme v našich programech potřebovat:

- Příznak nuly – ZF (zero flag) – se nastaví (na 1), pokud byl výsledek operace nula.
- Příznak znaménka – SF (signum flag) – který se nastaví na 1, pokud je výsledek operace záporný.
- Příznak přetečení – OF (overflow flag) – se nastaví, pokud výsledek „přetekl", tzn. byl větší, než mohlo vymezené paměťové místo pojmout.
- Příznak přenosu – CF (carry flag) – se nastaví, pokud nastal přenos do vyššího řádu. (jak tohle funguje, si povíme v kapitole o aritmetických instrukcích).
- Příznak přerušení – IF (interrupt flag) – je-li nastaven na 1, procesor smí vykonávat přerušení.
- Příznak směru – DF (direction flag) – se používá společně se sadou speciálních instrukcí pro práci s řetězci, o kterých si povíme v samostatné kapitole.

V mikroprocesoru existují ještě další registry, které jsou použité během chráněného režimu, ovšem popis a principy funkce chráněného režimu jsou mimo rámec této publikace.

Pokud byl procesor 80386 vybaven matematickým koprocesorem 80387 (další chip na poč. desce), mohl mnohem rychleji zpracovávat čísla v plovoucí řádové čárce.

V dnešních procesorech je matematický koprocesor přímo součástí procesoru, ale dříve se jednalo o samostatný chip, kterým jste mohli dovybavit váš počítač. Pokud nebyl koprocesor přítomen, musely být instrukce koprocesoru emulovány(prováděny složitěji) procesorem, což drasticky snižovalo výkon aplikací, které prováděly výpočty v plovoucí řádové čárce.

Poznámka: Kdykoli budeme hovořit o šestnácti- i dvaatřicetibitových registrech současně, použijeme např. místo zápisu AX i EAX zkrácený zápis (E)AX.

2.5 Přerušení

Přerušení je událost, při které procesor přestane vykonávat instrukce právě prováděného programu a dočasně začne vykonávat program jiný, obsluhující přerušení. Po ukončení obsluhy přerušení se obnoví provádění předchozího programu.

Můžeme si to představit na příkladu. Sedím u stolu a čtu si. Z pohledu počítače vykonávám program čtení knihy. Náhle zazvoní telefon – přerušení, do knihy dám záložku, tak si poznamenám kontext – místo posledního čtení. Jdu a zvednu telefon, právě vykonávám program obsluhy přerušení. Po skončení telefonického hovoru se vracím ke čtení knihy z místa, odkud jsem přestal. Kontext, místo posledního čtení, obnovím snadno podle záložky.

Mikroprocesory x86 kompatibilní umožňují vygenerovat až 256 různých přerušení. Adresy všech 256 obslužných programů (vektory přerušení) jsou uloženy ve speciální tabulce vektorů přerušení.

Přerušení můžeme rozdělit na přerušení hardwarová a přerušení softwarová.

Hardwarové přerušení vznikne jako žádost o programovou obsluhu některého z periferních zařízení PC. Běžně se jim říká IRQ – interrupt request – žádost o přerušení. Jejich počet je omezen architekturou sběrnice ISA na 16 (IRQ0 – IRQ15).

K hardwarovým přerušením se řadí přerušení generované samotným mikroprocesorem, který tak žádá o obsluhu „mimořádné události" – neplatného operandu, neznámé instrukce, přetečení nebo jiné situace, se kterou si neumí sám poradit. Přerušení generované mikroprocesorem mají svá vlastní vyhrazená čísla a neřadí se do 16 přerušení vyhrazených periferním zařízením.

Hardwarová přerušení můžeme rozdělit do dvou skupin podle toho, jestli se dají zakázat nebo ne – na maskovatelná a nemaskovatelná přerušení. Zakázat (maskovat) hardwarová přerušení můžeme tak, že příznak IF v příznakovém registru procesoru nastavíme na nulu.

Jediné přerušení, které nelze nijak zamaskovat, NMI (nemaskovatelné přerušení), se vyvolává pouze v katastrofických situacích – selhání paměti nebo výpadku napájení.

Softwarová přerušení jsou generována speciální instrukcí přímo v programu, autorem požadavku na přerušení je programátor. Často se softwarová přerušení používají k implementaci rozhraní mezi operačním systémem a uživatelským programem.

ANATOMIE INSTRUKCE

3

V této kapitole:

3.1 Operandy instrukcí
3.2 Adresace paměti
3.3 Instrukce jazyka symbolických adres

Instrukce mikroprocesoru je povel, který provede žádanou operaci s daty nebo změní vnitřní stav procesoru.

Procesory můžeme rozdělit do dvou skupin podle počtu jejich instrukcí. První skupinou jsou procesory RISC (Reduced Instruction Set Computer), jejichž název vznikl podle počítače RISC I, který byl jako první vybaven redukovanou instrukční sadou.

Autoři návrhu architektury RISC vycházeli z potvrzeného faktu, že procesor vykonává nejčastěji jen omezený počet instrukcí (instrukce skoku, přiřazení). Proto ponechali v procesoru jen nezbytné instrukce a procesor tak „odlehčili". Díky jednodušší vnitřní logice procesorů (menší počet instrukcí, menší počet logických obvodů) bylo možné zkrátit dobu zpracování jednotlivých instrukcí a tak zvýšit výkonnost systému. Instrukční sadu RISC má nepochybně i váš pes, disponuje omezenou sadou příkazů, ale umí je vykonat rychle.

Druhou skupinu procesorů tvoří procesory s komplexní (složitou) instrukční sadou. Zkráceně se jim říká CISC (Complex Instruction Set Computer). Poznáme je podle toho, že ve své instrukční sadě obsahují instrukce, které lze přepsat několika instrukcemi jednoduššími. Do skupiny mikroprocesorů CISC patří všechny procesory x86 kompatibilní.

Podívejme se společně, jak bude mikroprocesor zpracovávat instrukci „do registru AX nahraj číslo 0x1234". V jazyku symbolických adres (Assembleru) bude tento požadavek splňovat instrukce MOV AX,0x1234. My už víme, že instrukce pro počítač musí být vyjádřené v nějakém číselném kódu. Číselné reprezentaci jednotlivých instrukcí se říká

strojový kód. Instrukce MOV ax,0x1234 zapsaná ve strojovém kódu (v šestnáctkové soustavě) bude vypadat takto:

```
0x11xx:   nějaká předcházející instrukce
0x1111:   0xB8, 0x34, 0x12
0x1114:   další instrukce
```

Pro naše „testovací" účely jsme instrukci umístili na adresu 0x1111, vidíme, že další instrukce bude následovat tři byte za instrukcí MOV. V druhém a třetím byte poznáváme přímý operand instrukce MOV. Co ale znamená 0xB8? Pokud si 0xB8 rozepíšeme do binární číselné soustavy, dostaneme: 10111000b, první část čísla – 1011 – se jmenuje operační znak. Řadič procesoru tak pozná, že se jedná o určitou instrukci MOV. Další jednička řadiči přikazuje použít šestnáctibitové operandy v právě zpracovávané instrukci MOV. Poslední tři bity kódují registr, do kterého se bude přímý operand ukládat. Tři binární nuly kódují registr AX (nebo AL, jestliže byl předchozí bit 0).

Aby mohl řadič vůbec nějaké instrukce dekódovat, musí si je přečíst z paměti počítače. Předpokládejme, že procesor právě dokončil předcházející instrukci, hodnota registru IP (ukazatel na další instrukci) je už nastavena na 0x1111. Před zpracováním další instrukce se procesor „rozhlédne" po řídicí sběrnici, jestli není požadována obsluha HW přerušení.

Pokud ne, načte do vnitřního (instrukčního) registru procesoru hodnotu z adresy 0x1111, tedy 0xB8. Instrukci dekóduje, tak jak jsme si to ukázali před chvílí, a zjistí, že bude muset do registru AX nahrát dva byte, které těsně následují za 0xB8, tedy z adres 0x1112 a 0x1113. Proto vydá patřičné povely na sběrnici, počká, až mu paměť na datovou sběrnici dodá požadovanou hodnotu (0x1234), a tu zapíše do registru AX. Zvýší hodnotu registru IP o tři a znovu se podívá, jestli na něj nečeká nějaké přerušení. Pak vybere 1 byte z adresy 0x1114 a pokračuje v provádění programu dál.

Bylo-li přerušení požadováno, zjistí, o které se jedná, zkontroluje, je-li příznak IF nastaven na 1, jinak přerušení ignoruje, uloží kontext a začne provádět první instrukci obslužného programu zjištěného z tabulky vektoru přerušení.

Naštěstí díky Assembleru nebudeme muset kódovat instrukce do strojového jazyka přímo, ale vystačíme si s jejich symbolickým názvem. Před tím, než si probereme další instrukce, poučme se o jejich operandech.

3.1 Operandy instrukcí

Data, se kterými instrukce pracují, nazýváme operandy. Operandy píšeme v Assembleru za instrukci; pokud jich je víc, jsou odděleny čárkou. Některé instrukce nemají žádný operand, často mají operandy dva. Operandy mohou být přímé, typu „registr", a nepřímé – odkaz do paměti. Z hlediska počtu bitů rozlišujeme operandy na dvaatřicetibitové, šestnáctibitové a osmibitové. V drtivé většině instrukcí musí mít všechny operandy instrukce stejnou velikost.

Instrukce mov ax,0x1234 má dva operandy. První zleva typu registr a druhý přímý (oba šestnáctibitové).

Poslední typ operandů – operand nepřímý – adresuje data v paměti, která se vyberou a použijí jako operand instrukce. Poznáme ho podle hranatých závorek. O způsobech, jak adresovat paměť, si povíme v následující kapitole.

V manuálech se často vyskytuje stručný popis všech možných tvarů operandů instrukce pomocí následujících zkratek:

reg8	– operand může být v univerzálním osmibitovém registru
reg16	– operand může být v univerzálním šestnáctibitovém registru
reg32	– operand může být v univerzálním dvaatřicetibitovém registru
m	– operand může být uložen v paměti
imm8	– operandem může být přímá osmibitová hodnota
imm16	– operandem může být přímá šestnáctibitová hodnota
imm32	– operandem může být přímá dvaatřicetibitová hodnota
segreg	– operand je jedním ze segmentových registrů

Připouští-li se více druhů operandů, oddělují se lomítkem, pak

reg8/imm8 – operand může být osmibitovým registrem nebo přímým osmibitovým operandem.

Někdy se velikost operandu uvádí až u poslední možnosti, zápis na následující řádce má stejný význam jako předchozí příklad.

r/imm8 – operand může být osmibitovým registrem nebo přímým osmibitovým operandem.

3.2 Adresace paměti

Víme, že stejně jako instrukce je i adresa jen číslo. Abychom si nemuseli pamatovat adresu každé „proměnné" použité v našem programu, přiřadíme jí v Assembleru symbol, který bude představovat „ukazatel" na naši „proměnnou".

Pokud budeme chtít použít jako nepřímý operand instrukce hodnotu uloženou na adrese zastoupené symbolem cislo, sdělíme to překladači jednoduše: [cislo]. Když bychom se podívali na výpis strojového kódu instrukce, viděli bychom, že místo symbolu cislo je doplněná adresa místa v paměti, na které je „proměnná" cislo uložena. Známe-li adresu uložených dat, můžeme ji zapsat jako nepřímý operand přímo do hranatých závorek: [0x594F]

Častěji budeme k adresaci používat registry mikroprocesoru, ve kterých už budeme mít nahranou adresu místa v paměti, odkud se bude operand vybírat. Nepřímý operand vytvoříme snadno, jméno registru s adresou uzavřeme do hranatých závorek.

Bude-li adresa uložena v registru ESI, přistoupíme k datům uloženým na adrese, na kterou ukazuje registr ESI, takto: [esi]

Představme si fragment programu, ve kterém registr ESI obsahuje adresu prvního (s indexem nula) prvku pole byte čísel. Jak přistoupit například k druhému prvku pole,

(prvku s adresou o +1 byte vyšší), aniž bychom museli předem přičíst k registru ESI jedničku? Procesor nám umožní použít složitější adresace, ve kterých lze tyto požadavky snadno splnit. V našem příkladu stačí použít nepřímý operand [esi + 1].

Existují i složitější adresace: [cislo + ebx + 4]. Procesor přičte k adrese reprezentované symbolem cislo hodnotu 4 a k tomu ještě hodnotu uloženou v registru EBX. Výsledek, kterému se říká efektivní adresa (EA – Effective Address), se použije jako adresa, odkud se operand vybere (ponechme zatím stranou účast segmentových registrů). Procesor 80386 umožňuje při výpočtu efektivní adresy použít také násobení jednoho operandu konstantní mocninou čísla dvě [cislo + ebx*4]. V nějaké divočejší adresaci můžeme použít i další registry, takže i takovýto zápis je správný:

```
[cislo-6+ebx*8+esi]
```

V praxi často vystačíme jen s jedním registrem [esi] nebo přičtením nějaké konstantní hodnoty [esi + 4]. Podle módu procesoru můžeme použít libovolný šestnáctibitový nebo dvaatřicetibitový univerzální registr [eax] , [ebx] ... [ebp]. Adresace registrem a číselnou konstantou byla limitována u starších procesorů (80286) na registry bp, si, di a bx.

Jak už jsme naznačili, adresace paměti se účastní i segmentové registry. Jejich úloha se liší podle módu procesoru. Každá adresace předpokládá nějaký výchozí segmentový registr, který se při výpočtu adresy použije. Změnit výchozí segmentový registr můžeme tak, že ho napíšeme před hranatou závorku.

```
es:[esi]
```

Některé Assemblery připouští i zápis segmentového registru do hranaté závorky:

```
[es:esi]
```

V našich příkladech budeme předpokládat že jsou všechny hodnoty segmentových registrů shodné, a proto je nebudeme při adresaci uvažovat.

3.3 Instrukce jazyka symbolických adres

Když už víme, jak vypadají operandy, popišme si obecnou instrukci.

název_instrukce *nápověda* operandy

Popis instrukcí i jejich funkce je náplní následující kapitoly. O operandech jsme si už řekli, a tak zbývá jenom nápověda, která se uplatní především u instrukcí s přímým operandem. Nápovědou sdělíme překladači požadovanou velikost operandu. *Nápovědou* může být: byte word (slovo) nebo dword (dvojslovo). Klíčovým slovem byte napovídáme osmibitový, word šestnáctibitový a dword dvaatřicetibitový operand.

Představme si situaci, ve které budeme chtít nějakou „proměnou" vynulovat (zapsat na její adresu nulu). Použitím nápovědy překladač seznámíme s požadovanou velikostí operandu, tzn. kolik bytů nul se od adresy „proměnné" zapíše. Neznáme sice instrukci, která provede zápis, ale příklad by vypadal takto:

```
mov dword [ 0x12345678 ],0 ;na adresy od 0x12345678 zapiš 4 byty 0
mov word  [ 0x12345678 ],0 ;na adresy od 0x12345678 zapiš 2 byty 0
mov byte  [ 0x12345678 ],0 ;na adresu    0x12345678 zapiš 1 byte 0
```

Instrukce na prvním řádku zapíše nulový byte (nulu) od adresy 0x12345678 až do adresy o 3 vyšší. Instrukce na druhém řádku zapíše nulu (nulový byte) na adresu 0x12345678 a na 0x12345679. Poslední instrukce zapíše nulový byte na adresu 0x12345678.

ZÁKLADNÍ INSTRUKCE

4

V této kapitole:

4.1 Instrukce MOV
4.2 Co je malý a velký indián
4.3 Aritmetické instrukce
4.4 Další aritmetické instrukce
4.5 Logické instrukce

V této kapitole si představíme základní instrukční výbavu mikroprocesorů řady x86. Většina z instrukcí, které si zde předvedeme, bude základním stavebním kamenem dalších jednoduchých i náročnějších programů napsaných v Assembleru. U každé instrukce, se kterou se seznámíme, jsou mimo syntaxe a slovního popisu činnosti uvedeny také četné příklady použití, tak abychom z nich mohli snadno čerpat i při vlastním programování.

4.1 Instrukce MOV

Její název je odvozen z anglického slova move – přesuň. Instrukce MOV vyžaduje dva operandy, první cílový a druhý zdrojový.

Syntaxe: ***MOV cíl, zdroj***

Instrukce slouží, jak už název napovídá, k přesunům, přesněji řečeno ke kopírování zdrojového operandu do cílového. Možné operandy instrukce jsme si také již představili, ukažme si proto několik příkladů:

```
mov ax,[cislo]      ;do registru AX uložíme hodnotu "proměnné" číslo.
mov [cislo],bx      ;do "proměné" cislo ulož hodnotu, kterou obsahuje
                    ;registr BX
mov bx,cx           ;do registru BX ulož hodnotu z registru CX
mov al,1            ;do registru AL ulož konstantu 1
mov dh,cl           ;do registru DH ulož to co je v registru CL
mov esi,edi         ;překopíruj hodnotu z registru EDI do registru ESI
mov word [cislo],1  ;do "proměnné" číslo ulož šestnáctibitovou
                    ;konstantu 1
```

Architektura mikroprocesorů x86 a kompatibilních dovoluje uvést nejvýše jeden operand instrukce jako nepřímý operand – ležící v paměti. Pokud budeme chtít překopírovat obsah jedné „proměnné" (cislo_jedna) do druhé (cislo_dva), nemůžeme použít následující instrukci MOV.

```
mov [cislo_dva],[cislo_jedna]      ;TO JE ŠPATNĚ !!! (překladač takovou
                                   ;ani instrukci nepřeloží)
```

Musíme použít registru jako prostředníka.

```
mov ax,[cislo_jedna]        ;do AX vložíme šestnáctibitovou hodnotu
                            ;"proměnné" cislo_jedna
mov [cislo_dva],ax          ;kterou nakopírujeme do "proměnné" cislo_dva
```

Všechny operandy instrukce MOV musí mít stejnou velikost:

```
mov ax,bl           ;Špatně! - nestejná velikost operandů
```

Pokud bychom z nějakého důvodu potřebovali hodnotu z registru BL dostat do AX (rozšířit rozsah), musíme do AX překopírovat celý registr BX a registr AH vynulovat.

```
mov ax, bx          ;do AX ulož BX
mov ah,0            ;vynuluj horní část registru AX
```

Připomeňme si, že registr AH představuje horních 8 bitů registru AX. Po vykonání instrukce MOV ax,bx je horních osm bitů registru AX nastaveno podle hodnoty v registru BH. Nemůžeme se spoléhat, že v registru BH byla uložena nula, a proto druhou instrukcí MOV ah,0 registr AH vynulujeme. Tak rozšíříme původní osmibitovou hodnotu uloženou v registru BL na šestnáctibitovou hodnotu uloženou v registru AX (uvažujeme stále čísla bez znaménka).

Stejného výsledku docílíme i kódem:

```
mov ax,0            ;vynulujeme registr AX - AH i AL budou nula
mov al,bl           ;do registru AL nahrajeme obsah BL
```

Obdobný postup můžeme uplatnit, kdykoli budeme potřebovat překopírovat šestnáctibitovou hodnotu do dvaatřicetibitového registru.

Pro úplnost zde uvádím popis všech možných (přípustných) tvarů operandů instrukce MOV, tak jak se často vyskytují v manuálech.

MOV r/m8,reg8

MOV r/m16,reg16

MOV r/m32,reg32
MOV reg8,r/m8
MOV reg16,r/m16
MOV reg32,r/m32
MOV reg8,imm8
MOV reg16,imm16
MOV reg32,imm32
MOV r/m8,imm8
MOV r/m16,imm16
MOV r/m32,imm32

4.2 Co je malý a velký indián

Na tomto místě plánovaně odbočíme od výkladu instrukcí. Představíme si situaci, ve které bychom chtěli ověřit, že se do dvaatřicetibitové „proměnné" nazvané pocitadlo nahrála skutečně hodnota 0x12345678, která byla uložena v registru EBP. Úsek programu by vypadal takto:

```
mov ebp,0x12345678          ;do registru EBP ulož číslo 0x12345678
mov [pocitadlo], ebp        ;hodnotu z registru EBP ulož do
                            ;"proměnné" pocitadlo
```

Zatím neumíme program ani přeložit, natož spustit v nějakém odladovacím programu, ale předpokládejme na malou chvilku, že to zvládneme.

Jak by přeložený program vypadal? Odladovací program převede zpět všechny strojové instrukce programu do jazyka symbolických adres. Všude, kde jsme použili jméno nějakého symbolu („proměnné"), se však objeví jen adresa, která byla symbolu přidělena.

```
0804808A    BD78563412                  mov ebp, 0x12345678
0804808F    892DC0900408                mov dword [+0x80490c0], ebp
```

Z prvního sloupce obou řádků výpisu zjistíme, na které adrese se instrukce nachází, v dalším je uveden její strojový kód a následuje forma zapsaná zpětně v jazyce symbolických adres. Z výpisu můžeme zjistit, že operand druhé instrukce MOV je uložen na adrese 0x80490c0, což je adresa naší „proměnné" pocitadlo.

Před vykonáním první instrukce mov ebp,0x12345678 vypadaly registry mikroprocesoru takto:

```
eax = 0x00000000 ebx = 0x00000000 ecx = 0x00000000 edx = 0x00000000
esp = 0xBFFFF910 ebp = 0x00000000 esi = 0x00000000 edi = 0x00000000
ds  = 0x0000002B es  = 0x0000002B fs  = 0x00000000 gs  = 0x00000000
ss  = 0x0000002B cs  = 0x00000023 eip = 0x0804808A eflags = 0x00200346

Flags: PF ZF TF IF ID
```

Po vykonání první instrukce se obsah registru EBP změnil na 0x12345678. Vypíšeme-li obsah paměti (dump) od adresy „proměnné" pocitadlo před vykonáním druhé instrukce, dostaneme:

```
Dumping 64 bytes of memory starting at 0x080490C0 in hex
080490C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Po vykonání druhé instrukce MOV, která zapíše hodnotu 0x12345678 na adresu 0x80490c0, bude výpis paměti vypadat takto:

```
Dumping 64 bytes of memory starting at 0x080490C0 in hex
080490C0: 78 56 34 12 00 00 00 00 00 00 00 00 00 00 00 00 xV4 ............
```

Hodnota 0x12345678 se skutečně uloží od vypisované adresy, jenže pozpátku. Proč? Protože počítače x86 kompatibilní jsou little-endian. Slovo little endian je „zkratka" anglického little-end-first, kterou označujeme pořadí ukládání jednotlivých byte slova nebo dvojslova v paměti počítače.

Pokud je procesor little-endian, pak ukládá nejnižší byte slova a dvojslova (word, dword) na nižší adresy. Často se little a big endian přezdívá malý a velký indián, protože je výslovnost obou slov v angličtině blízká. Velcí indiáni (big endian – big end of word) ukládají části slov a dvojslov do paměti zase opačně (např. procesory Motorola).

Pro úplnost, pokud načteme hodnotu „proměnné" pocitadlo zase zpátky do nějakého registru, bude opět obsahovat 0x12345678, tedy správnou hodnotu.

Ti zkušenější z vás se možná s malými a velkými indiány setkali při psaní síťových aplikací v programovacím jazyce C, kde bylo nutné některé části struktur ukládat také jako velké indiány. Za tímto účelem jste museli používat konverzní funkce (htonl, htons, ntohl, ntohs).

4.3 Aritmetické instrukce

Další často používané instrukce jsou instrukce aritmetické. Jak jsme si už řekli, procesor 80386 nemá matematický koprocesor, a tak si probereme jen celočíselnou aritmetiku, kterou 80386 plně podporuje. Všechny aritmetické instrukce mění příznaky v příznakovém registru procesoru.

4.3.1 Instrukce ADD a SUB

Začneme jako v první třídě s instrukcí ADD (sečti). Instrukce ADD má dva operandy, podobně jako instrukce MOV.

Syntaxe: ***ADD o1,o2***

ADD sečte oba operandy a výsledek uloží do operandu o1, jeho předchozí hodnota je ztracena.

Instrukce pro odčítání je označována SUB (z anglického substract) a její syntaxe je obdobná.

Syntaxe: ***SUB o1,o2***

Výsledek o1 – o2 se uloží do o1; jeho původní hodnota je opět ztracena. Ukažme si několik příkladů:

Př.: Sečtěte čísla 8 a 6 (uložená v registru AX a CX), výsledek uložte do DX:

```
mov ax,8            ;do registru AX vložíme 8
mov cx,6            ;do registru CX zase 6
mov dx,cx           ;CX okopírujeme do DX takže DX = 6
add dx,ax           ;sečteme DX = DX + AX
```

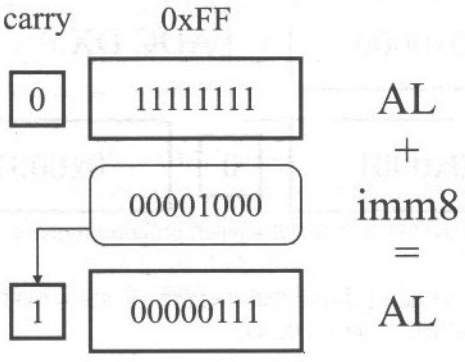
Protože jsme chtěli uchránit registr AX i CX před zničením (přepsáním výsledkem), překopírovali jsme hodnotu uloženou v registru CX do DX a sečetli registr DX s AX. Instrukce ADD výsledek součtu DX + AX uložila do DX.

Ukažme si další příklady, jak použít instrukce ADD a SUB.

```
add eax,8                   ;k registru EAX přičteme 8
sub ecx,ebp                 ;registr ECX = ECX - EBP
add byte [cislo], 4         ;k hodnotě "proměnné" cislo přičteme čtyři
                            ;(napovídáme rozsah 1 byte 0-256)
sub word [cislo], 4         ;od hodnoty "proměnné" cislo odečteme čtyři
                            ;(napovídáme rozsah 2 byte 0-65 535)
add dword [cislo], 4        ;k hodnotě "proměnné" cislo přičteme čtyři
sub byte [cislo], al        ;od hodnoty "proměnné" cislo odečteme
                            ;registr AL
sub ah,al                   ;od registru AH odečteme AL a výsledek
                            ;uložíme do AH
```

Co se stane, když k registru AL, jehož hodnotu nastavíme na nejvyšší možnou – 255, přičteme třeba 8?

```
mov al,255          ;do registru AL vložíme 255 - 0xFF
add al,8            ;a přičteme 8
```



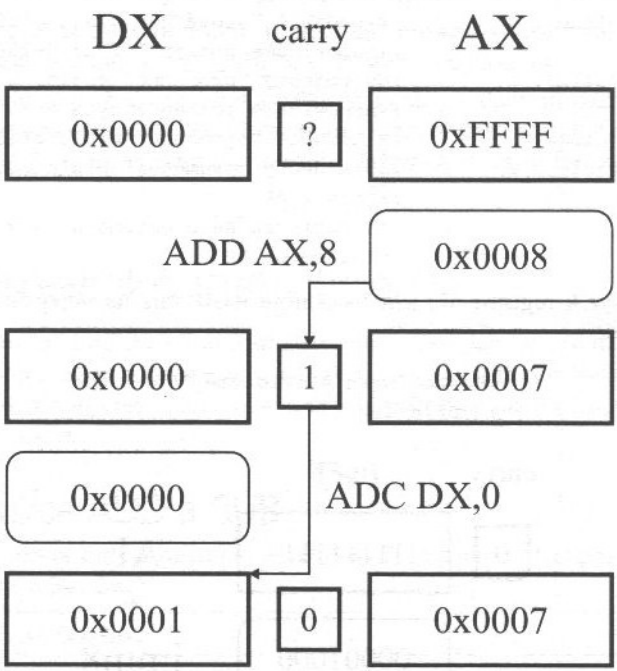
Obrázek 7 Součet čísel 255 (0xFF) s číslem 8

Výsledek součtu bude uložen v registru AL a bude 7. Spíše bychom očekávali výsledek 0x107 (263). Registr AL je však pouze osmibitový, a tak se hodnota do registru prostě nevešla. Devátý „ztracený" bit se schovává v registru příznaků, konkrétně v příznaku přenosu CF (carry). Přenos můžeme využít i v číselné aritmetice, ve které budeme pracovat s většími rozsahy čísel, než jsou rozsahy registrů. Pomáhat nám budou instrukce ADC (addition with carry – součet s přenosem) a SBB (substraction with borrow, odčítání s přenosem):

Syntaxe:	***ADC o1,o2***	***;o1 = o1 + o2 + CF***
	SBB o1,o2	***;o1 = o1 – o2 – CF***

Instrukce pracují úplně stejně jako ADD a SUB, ovšem ve svých výpočtech přičítají nebo odčítají k výsledku hodnotu příznaku přenosu (CF carry).

V souvislosti s aritmetikou si představme nový pojem „registrový pár". Registrový pár tvoří dvojice zvolených registrů, které se používají pro uložení jednoho čísla. Často se za takový pár volí registry EDX:EAX (nebo DX:AX), které využívá také instrukce pro násobení. V registru AX je uloženo spodních 16 bitů čísla a v registru DX zase horních 16 bitů čísla. Tak jsme mohli i na procesorech 80286 počítat s dvaatřicetibitovými čísly, aniž jsme měli tak velký registr.



Obrázek 8 Součet čísel 0xFFFF a 0x0008 a jejich průchod registry

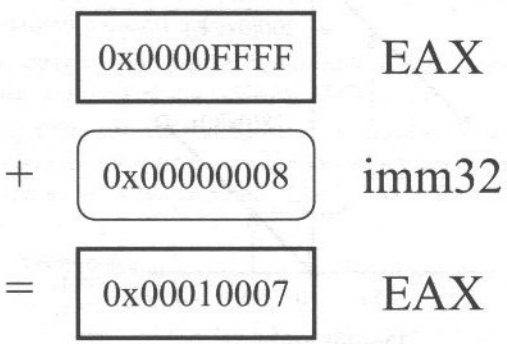
Příklad: V registru DX:AX je uloženo číslo 0xFFFF (v AX = 0xFFFF a DX = 0). Přičtěte k němu 8 a výsledek uložte opět v DX:AX.

```
mov ax,0xffff       ;do AX vložíme 0xFFFF
mov dx,0            ;do DX nulu
add ax,8            ;přičteme k AX 8
adc dx,0            ;a k DX přičteme s přenosem nulu.
```

První instrukce ADD přičte 8 k hodnotě registru AX. Výsledek součtu se celý do registru nevešel, nejvyšší bit byl uložen do příznaku CF (carry). Druhá instrukce přičítá k registru DX nulu a hodnotu příznaku přenosu. Předchozí součet nastavil příznak přenosu, který se instrukcí ADC přičetl k hodnotě registru DX (DX = 1). V registrovém páru je správný výsledek DX:AX 0x10007 (DX = 1 a AX = 0x0007).

Na procesoru 80386 můžeme počítat přímo s dvaatřicetibitovými čísly, a tak bychom výše uvedený program mohli napsat rovnou tak:

```
mov eax,0xffff      ;do EAX vlož 0xFFFF
add eax,8           ;přičti 8
```



Obrázek 9: Součet čísel 0xFFFF a 0x0008 do dvaatřicetibitového registru

V registru EAX bude správná hodnota 0x10007. Pokud budeme chtít pracovat s čísly v 64bitovém rozsahu, použijeme EDX:EAX jako registrový pár.

4.3.2 Instrukce INC a DEC

Syntaxe:	***INC o1***	***;o1 = o1 + 1***
	DEC o1	***;o1 = o1 – 1***

Jejich názvy jsou opět odvozeny z angličtiny a znamenají increment (zvýšit) a decrement (snížit).

Instrukce přičítá (INC) nebo snižuje (DEC) jediný operand o jedničku. Jeho typ je stejný jako instrukce ADD a SUB. Pokud použijeme operand ležící v paměti, musíme překladači opět napovědět požadovaný rozsah operandu.

POZOR! Tyto instrukce nemodifikují příznak přenosu! Jaké to má důsledky a jak bezpečně tyto instrukce používat, si řekneme v kapitole věnované optimalizaci.

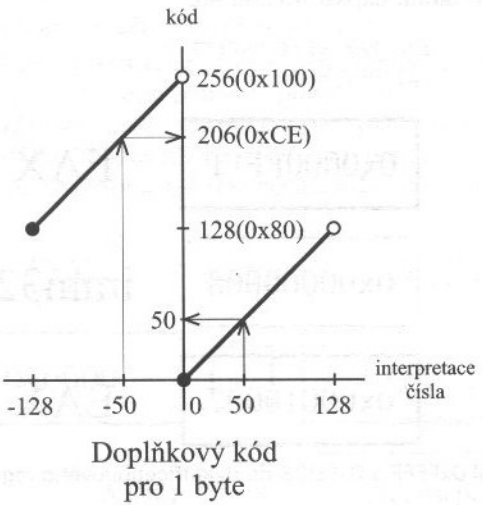
Příklad: Přičtěte k registru AL jedničku.

```
add al,1     ;je správně, ovšem díky instrukci INC můžeme požadavek
inc al       ;splnit i takto.
```

Příklad: Zvyšte hodnotu šestnáctibitové „proměnné" cislo o 1.

```
inc word [cislo]    ;přičetli jsme jedničku, napověděli jsme rozsah
                    ;0-65535, tj. 16 bitů.
```

4.3.3 Záporná čísla – čísla se znaménkem



Obrázek 10 Doplňkový kód pro jeden byte

Záporná čísla jsou na počítačích PC zobrazena v tzv. doplňkovém kódu. Doplňkový kód si můžeme představit jako zobrazení, které jistý interval kladných a záporných celých čísel promítne do intervalu kladných celých čísel. Ukažme si, jaký rozsah (interval) čísel budeme moci zobrazit, použijeme-li doplňkový kód pro jeden byte.

Rozsah hodnot jednoho bytu je 0 až 255, doplňkový kód rozsah zobrazitelných čísel změní tak, že bude možné jedním byte reprezentovat čísla od -128 do +127. Čísla v intervalu 0 až 127 se zobrazí na čísla 0 až 127. Záporná čísla se zobrazí do intervalu 128 až 255. Kladná čísla zachovávají svoji hodnotu, číslu 50 bude i v doplňkovém kódu přiřazeno číslo 50. Podle obrázku vidíme, že čísla záporná se přiřazují obráceně. Číslu -1 bude odpovídat hodnota 255 (0xFF), číslu -2 hodnota 254 (0xFE). Číslu -50 bude odpovídat číslo 206. Všimněme si, že záporné číslo zapsané ve dvojkové soustavě můžeme od kladného rozeznat podle nejvyššího bitu čísla, který bude vždy nastaven na 1.

Doplňkový kód můžeme rozšířit na dva byty (0 až 65 535) a získat tak rozsah hodnot známý z mnoha programovacích jazyků pod typem integer: -32 768 až 32 767. Často se používá rozšíření doplňkového kódu na 4 byte. Tak vznikne interval čísel -2 147 483 648 až 2 147 483 647, používaný jako integer v mnoha dvaatřicetibitových systémech.

Příklad: Převeďte čísla 4, -4, 386, -8 000 a 45 000 do doplňkového kódu. Rozsah obrazů čísel uvažuje šestnáct bitů (2 byty).

Nejprve si uvědomíme, kolik šestnáctibitových kladných čísel můžeme vytvořit. Jednoduše číslo dvě umocníme na šestnáct a odečteme jedničku. Dostaneme: $2^{16} - 1 = 65\,536 - 1 = 65\,535$. Obdobně jako na předchozím obrázku vypočteme meze zobrazitelných čísel: 65 356/2, dostaneme 32 768. Největší zobrazitelné kladné číslo bude 32 767 a nejmenší zobrazitelné záporné číslo bude -32 768.

Číslo 4 je v intervalu <0, 32 767>, a proto se zobrazí na 4. Číslo -4 je záporné a leží ve vyhovujícím intervalu <-32 768, 0). Číslo -4 zobrazíme jako 65 536 - 4 = 65 532. Číslo 386 se zobrazí jako 386. Číslo -8 000 se zobrazí na 65 536 - 8 000 = 57 536. Poslední číslo 45 000 nelze v daném rozsahu zobrazit.

Počítání se zápornými čísly v doplňkovém kódu bude fungovat správně, i když použijeme „obyčejné" instrukce pro sčítání a odčítání ADD a SUB.

Ukažme si princip jejich činnosti na součtu -6 a 7 v doplňkovém kódu (2 byty). Číslo 7 zobrazíme na 7 a číslo -6 zobrazíme na 65 536 - 6 = 65 530, což je v šestnáctkové soustavě 0xFFFA. Co se stane, když sečteme čísla 7 a 65 530? Zkusme problém zapsat v Assembleru:

```
mov ax,0xFFFA       ;do AX vlož -6 což je 65530 neboli 0xFFFA
mov dx,7            ;do DX vlož 7
add ax,dx           ;sečti AX a DX výsledek ulož do AX
```

Výsledek součtu hodnot 65 530 + 7 = 65 537 = 0x10001 se do registru AX nevejde, bude nastaven příznak přenosu. Pokud ho budeme ignorovat, dostaneme správný výsledek! Doplňkový kód byl navržen tak, aby se při počítání se zápornými čísly aspoň v případě sčítání a odčítání nemuselo nic měnit.

Ukažme si jiný příklad, kde zkusíme sečíst dvě záporná čísla -6. Assembler NASM je nepotřebuje ručně převádět, a proto i následující program bude pracovat bezchybně:

```
mov ax, -6          ;do AX vlož -6
mov dx,-6           ;do DX vlož -6
add ax,dx           ;sečti DX a AX, výsledek do AX
```

Výsledek součtu je 0xFFF4 (a je nastaven příznak přenosu, který budeme ignorovat), v desítkové soustavě je výsledek 65 524. Číslo v doplňkovém kódu zobrazíme zpátky snadno, stačí když jej odečteme od 65 536, dostaneme očekávaný výsledek -12.

Záporná čísla můžeme využít i při adresaci paměti. Představme si, že v registru bx je nějaká adresa a my budeme chtít přistoupit na adresu o 1 níž, ovšem hodnotu v bx nechceme měnit (předpokládáme realný režim procesoru).

```
mov ax,[bx-1]       ;načti do AX z adresy o 1 méně, než je
                    ;adresa v BX
```

Bude fungovat také tak, že -1 se přepíše na 0xFFFF, instrukce bude vypadat takto: MOV ax, [bx+0xFFFF], při výpočtu adresy žádný přenos neuplatňuje, výsledek se „přetočí" na hodnotu o 1 menší, tak jak jsme požadovali.

4.3.4 Instrukce pro záporná čísla – NEG CBW CWD CDQ CWDE

Instrukční sada mikroprocesoru 80386 obsahuje i instrukce, které ulehčují práci s čísly se znaménkem. Název instrukce je odvozen z anglického slova negate.

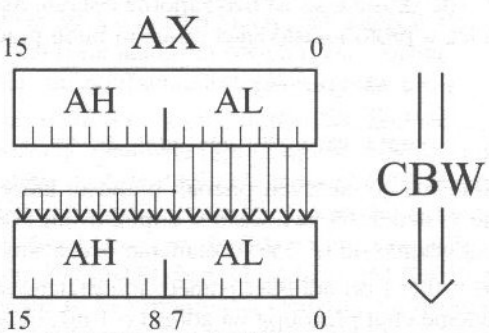
Syntaxe: ***NEG r/m8***
NEG r/m16
NEG r/m32

Pomocí instrukce NEG můžeme z kladného čísla utvořit záporné a naopak. Často se této operaci říká dvojkový doplněk, neboť tato instrukce neguje (invertuje) všechny bity a přičte jedničku. Instrukce má jen jeden operand, kterým může být registr nebo paměťové místo a jeho velikost může být 8, 16 nebo 32 bitů.

```
neg eax      ;číslu uloženému v registru EAX "změní znaménko"
neg bl       ;jako v prvním příkladu, ale změna znaménka je provedena s
             ;číslem uloženým v registru BL

neg byte [cislo]    ;změnili jsme znaménko číslu uloženému
                    ;v osmibitové "proměnné" cislo
```

Rozsah čísel bez znaménka jsme mohli měnit snadno. Jednoduše jsme číslo překopírovali do většího registru a místo, které jsme „přidali", jsme vynulovali. Uvažujeme-li čísla se znaménkem, musíme nově přidané „místo" vyplnit nejvyšším bitem převáděného čísla. Tak zachováváme i záporná i kladná čísla a zvětšíme jejich rozsah (počet bitů). Změně rozsahu čísel se znaménkem se říká znaménkové rozšíření.



Obrázek 11 Znaménkové rozšíření instrukcí CBW

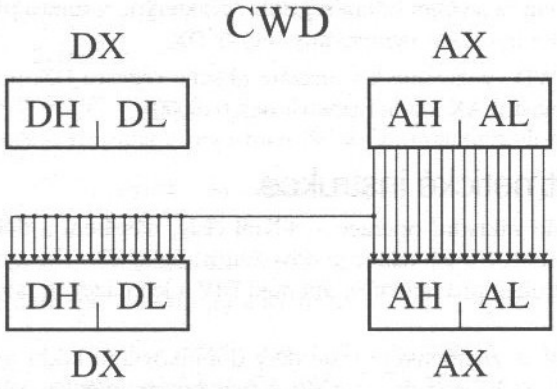
Mikroprocesor je vybaven speciálními instrukcemi s pevně danými operandy, které znaménkově rozšíří první operand do druhého operandu.

Syntaxe: ***CBW***

Překopíruje sedmý (nejvyšší bit) registru AL do registru AH, čímž původní hodnotu v registru AL znaménkově rozšíří do registru AX (registr AH bude 0x00 nebo 0xFF (11111111b) podle toho, jakou měl nejvyšší bit registru AL hodnotu).

Syntaxe: ***CWD***

Překopíruje nejvyšší bit registru AX do registru DX a tím AX znaménkově rozšíří do registrového páru DX:AX



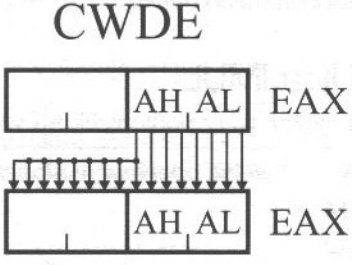
Obrázek 12 Znaménkové rozšíření instrukcí CWD

Syntaxe: ***CDQ***

Překopíruje nejvyšší bit registru EAX do registru EDX a tím EAX znaménkově rozšíří do registrového páru EDX:EAX.

Syntaxe: ***CWDE***

Překopíruje nejvyšší bit registru AX do horní poloviny registru EAX a tím AX znaménkově rozšíří do registru EAX.



Obrázek 13 Znaménkové rozšíření instrukcí CWDE

Ukažme si použití na příkladu:

```
mov al,-1    ;do AL uložíme -1 neboli hodnotu 0xFF
cbw          ;a znaménkově rozšíříme do AX
```

Registr AX bude po vykonání instrukce CBW 0xFFFF, tedy -1. Jednička z nejvyššího bitu registru AL se překopírovala do všech bitů registru AH, čímž AL znaménkově rozšířila do AX.

```
mov ax,4     ;do AX vložíme 4
cwd          ;a znaménkově rozšíříme do DX
```

Po vykonání první instrukce bude registr AX obsahovat hodnotu 4, druhá instrukce CWD znaménkově rozšíří registr AX do DX:AX. Původní hodnota registru DX bude přepsaná okopírovaným nejvyšším bitem registru AX, který je v tomto případě nula. Výsledek druhé instrukce bude jen vynulovaný registr DX.

Někdy instrukci CWD využíváme k vymazání obsahu registru DX, pokud máme jistotu, že je hodnota v registru AX kladná (menší než 0x8000).

4.4 Další aritmetické instrukce

Představme si zbylé základní operace s celými čísly, násobení a dělení. Instrukce pro násobení se nazývá MUL a její název je odvozen z anglického slova multiply – což znamená násobit. Instrukce pro dělení se jmenuje DIV a její název je podle anglického slova divide – dělit.

Pro výpočty s čísly se znaménkem jsme díky doplňkovému kódu mohli používat stejné instrukce (ADD, SUB) jako pro výpočty s čísly bez znaménka. Násobení a dělení by dávalo špatné výsledky, a tak existují odpovídající instrukce zohledňující práci s čísly se znaménkem – IMUL a IDIV (integer mul a integer div). Jejich operandy jsou stejné jako u instrukcí MUL a DIV.

Násobením čísel dostaneme vždy v nejhorším případě číslo, jehož počet bitů (rozsah) je dvojnásobný než počet bitů čísla původního. Dělení celých čísel je prováděno celočíselně, a proto dostaneme také dva výsledky (výsledek a zbytek po dělení). Aby nebyla instrukce MUL nebo DIV příliš implementačně složitá, bylo rozhodnuto, že jeden operand a výsledek bude uložen v (stejném) pevně daném registru. Umístění druhého operandu bylo ponecháno na programátorovi.

Obdobně jako instrukce ADD a SUB i instrukce MUL, DIV a IMUL, IDIV mění registr příznaků podle výsledku násobení nebo dělení.

4.4.1 Instrukce MUL a IMUL

Instrukce MUL může mít tři možné tvary podle velikosti volitelného operandu:

Syntaxe:	***MUL r/m8***	***;8-bit. operand je uložen v registru nebo v paměti***
	MUL r/m16	
	MUL r/m32	

V osmibitové variantě je volitelný operand libovolným osmibitovým registrem nebo místem v paměti. Předpokládá se, že druhý operand je v registru AL. Výsledek součinu se uloží do registru AX.

(r/m8) * AL -> AX

V šestnáctibitové variantě je volitelný operand libovolným šestnáctibitovým registrem nebo místem v paměti, druhý operand je v registru AX. Výsledek součinu se ukládá do registrového páru DX:AX.

(r/m16) * AX -> DX:AX

V dvaatřicetibitové variantě je volitelný operand libovolným dvaatřicetibitovým registrem nebo místem v paměti, druhý operand se předpokládá v registru EAX, výsledek součinu se ukládá do registrového páru EDX:EAX.

(r/m32) * EAX -> EDX:EAX

Uveďme si několik příkladů:

Příklad: Vynásobte čísla uložená v registrech BH a CL, výsledek uložte do registru AX.

```
mov al,bh          ;registr BH nahraj do AL
mul cl             ;vynásob AL s CL
```

Výsledek bude již uložen v registru AX.

Příklad: Umocněte číslo 486 na druhou a výsledek uložte v registrech DX:AX.

```
mov ax,486         ;nahrajeme hodnotu 486 do registru AX
mul ax             ;vynásobíme AX * AX -> DX:AX
```

Příklad: Převeďte poloměr uložený v osmibitové proměnné polomer1 do šestnáctibitové „proměnné" prumer1.

```
mov al,2           ;do AL uložíme dvojku
mul byte [polomer1] ;vynásobíme s obsahem "proměnné" polomer1
mov [prumer1],ax   ;a výsledek uložíme do "proměnné" prumer1
```

Možná přemýšlíte o tom, proč šestnáctibitové násobení ukládá výsledek v registrovém páru DX:AX, a ne v nějakém dvaatřicetibitovém registru. Je to z důvodu zachování kompatibility s předcházejícími mikroprocesory, které byly pouze šestnáctibitové, a nedisponovaly tudíž tak velkými registry.

Instrukce IMUL se používá pro násobení celých čísel se znaménkem, může mít jeden, dva nebo tři operandy.

Varianta s jedním operandem vyžaduje ostatní operandy ve stejných registrech jako instrukce MUL.

Instrukce IMUL se dvěma operandy výsledek násobení uloží do operandu prvního. První operand musí být vždy typu registr, druhý operand smí být libovolného typu (přímý, nepřímý v paměti, registr).

```
imul edx,ecx       ;EDX = EDX * ECX
imul ebx,[cosi]    ;vynásobí dvaatřicetibitovou "proměnnou" cosi s
```

```
                    ;registrem EBX, výsledek bude v EBX
imul ecx,6          ;ECX = ECX * 6
```

Varianta se třemi operandy vynásobí poslední dva a výsledek uloží do prvního. První operand musí být typu registr, druhý smí být libovolný, třetí musí být přímým operandem.

```
imul edx,ecx,7      ;EDX = ECX * 7
imul ebx,[cosi],9   ;vynásobí dvaatřicetibitovou "proměnnou" cosi
                    ;devíti, výsledek uloží do EBX
imul ecx,edx,11     ;ECX = EDX * 11
```

4.4.2 Instrukce DIV a IDIV

Obdobně jako instrukce MUL má i instrukce DIV tři možné tvary lišící se velikostí použitých volitelných operandů.

Syntaxe: ***DIV r/m8*** ***;osmibitový volitelný operand v paměti nebo v registru***
DIV r/m16
DIV r/m32

Volitelným operandem je dělitel, pevným operandem je dělenec.

V osmibitové variantě je volitelný operand libovolným osmibitovým registrem nebo místem v paměti. Pevný operand se předpokládá v registru AX, výsledek dělení (podíl) se uloží do registru AL a zbytek po dělení se uloží v registru AH.

AX / (r/m8) -> AL, zbytek v AH

V šestnáctibitové variantě je volitelný operand libovolným šestnáctibitovým registrem nebo místem v paměti. Pevný operand se předpokládá v registrovém páru DX:AX, výsledek dělení se uloží do registru AX, zbytek po dělení se uloží do registru DX.

DX:AX / (r/m16)-> AX, zbytek v DX

V dvaatřicetibitové variantě je volitelný operand libovolným dvaatřicetibitovým registrem nebo místem v paměti, pevný operand se předpokládá v registrovém páru EDX:EAX, výsledek dělení se ukládá do registru EAX, zbytek po dělení do registru EDX.

EDX:EAX / (r/m32) -> EAX, zbytek v EDX

Instrukci IDIV použijeme při dělení čísel se znaménkem, operandy instrukce jsou stejné jako u instrukce DIV.

Uveďme si opět několik příkladů.

Příklad: Celočíselně vydělte číslo 13 dvěma, výsledek uložte do registru BL a zbytek do registru BH.

```
mov ax,13           ;do AX vložíme 13
mov cl,2            ;do CL 2
```

```
div cl              ;vydělíme CL
mov bx,ax           ;výsledek očekáváme v BX, stačí překopírovat
```

Příklad: Převeďte šestnáctibitovou proměnnou prumer1 na poloměr a výsledek uložte do osmibitové proměnné polomer1 (zbytek zanedbejte).

```
mov ax,[prumer1]    ;načteme prumer1
mov bl,2            ;připravíme na dělení 2
div bl              ;vydělíme
mov [polomer1], al  ;podíl uložíme
```

4.5 Logické instrukce

Logické instrukce vykonávají základní logické operace, kterými jsou: logický součin – AND, logický součet – OR, logická non-ekvivalence – XOR a negace – NOT. Všechny logické instrukce mění stav registru příznaků.

4.5.1 Instrukce AND

Syntaxe: ***AND o1,o2***

Instrukce AND má dva operandy a provede logický součin operandu o1 a o2. Výsledek uloží do operandu o1 (změní se i příznakový registr procesoru). Druhy operandů jsou obdobné jako u instrukce pro sčítání ADD, akceptuje 8, 16 a 32bitové operandy, které mohou být v registrech, v paměti nebo přímé. Připomeňme si pravdivostní tabulku logického součinu: a AND b

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

Zkusme si napsat jednoduchý příklad, ve kterém provedeme logický součin log. jedničky a logické nuly.

```
mov al,1    ;do AL vložíme jedničku
mov bl,0    ;do BL zase nulu
and al,bl   ;provedeme logický součin AL and BL, výsledek bude v AL.
            ;1 and 0 = 0
```

Příklad lze zapsat i úsporněji:

```
mov al,1    ;do AL vložíme jedničku
and al,0    ;provedeme logický součin AL and 0, výsledek bude v AL.
            ;1 and 0 = 0
```

4.5.2 Instrukce OR

Syntaxe: ***OR o1,o2***

Instrukce OR provede nad operandy o1 a o2 logický součet a výsledek uloží do operandu o1. Druhy operandů jsou totožné jako u instrukce AND.

Pravdivostní tabulka logického součtu je:

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

S dalšími konkrétními příklady použití logických instrukcí se podrobněji seznámíme v podkapitole věnované bitovým vlajkám. Ukažme si pouze jednoduchý příklad, ve kterém nastavíme první bit zprava osmibitové proměnné maska na jedničku.

```
or byte [maska],1   ;nastavíme první bit osmibitové "proměnné" maska na
                    ;jedničku
```

4.5.3 Instrukce XOR

Syntaxe: ***XOR o1,o2***

S operandy o1 a o2 provede logickou non-ekvivalenci a výsledek uloží do operandu o1. Typy operandů jsou stejné jako u AND a OR. Její pravdivostní tabulka vypadá takto:

a	b	a XORb
0	0	0
0	1	1
1	0	1
1	1	0

Logická operace XOR (non-ekvivalence), je „vratná". Pokud provedeme: ((x xor y) xor y) dostaneme zase x.

```
mov al,0x55 ;do AL vložíme 0x55
xor al,0xAA ;provedeme operaci XOR s konstantou 0xAA
            ;dostaneme v AL 0xFF
xor al,0xAA ;dostaneme v AL opět původní hodnotu 0x55
```

4.5.4 Instrukce NOT

Syntaxe: ***NOT r/m8***
NOT r/m16
NOT r/m32

Provádí negaci (invertuje jednotlivé bity) svého jediného operandu. Ten může ležet v paměti nebo být uložen v nějakém registru. Přímý operand neexistuje. Operaci NOT se často říká jedničkový doplněk.

a	NOT a
0	1
1	0

Ukažme si na příkladu rozdíl mezi instrukcemi NOT a NEG.

```
mov al,00000010b    ;do AL vložíme dvojku
mov bl,al           ;do BL také 2
not al              ;po negaci (jedničkový doplněk) dostaneme
                    ;11111101b = 0xFD (-3)
neg bl              ;provedeme negaci (dvojkový doplněk)
                    ;11111110 = 0xFE (-2)(dvojkový doplněk = jedničkový
                    ;doplněk + přičtení jedničky)
```


4.5.5 Bitové vlajky – pole (mapy)

Každé číslo lze zapsat v binární soustavě jako posloupnost nul a jedniček. Například šestnáctibitové číslo má dohromady šestnáct různých pozic, kam můžeme zapsat nulu nebo jedničku. Toho lze využít v případě, když potřebujeme někam zaznamenat nějaký dvouhodnotový stav. Bitovou mapou (polem) tak ušetříme paměť tím, že na reprezentaci každého stavu místo jednoho byte použijeme pouze jeden bit.

Ve vyšších programovacích jazycích se bitové pole používá například při implementaci množin výčtových typů, kde se tak šetří paměť. V hardwarové oblasti jedno bitové pole už známe, a to registr příznaků mikroprocesoru. Další bitová pole najdeme často v periferních zařízeních počítače jako speciální stavové a řídicí registry (nejde samozřejmě o registry mikroprocesoru, ale registry uložené v nějakých řídicích obvodech periferních zařízení, jako třeba videokarta nebo zvuková karta atd.).

Ke změně jednotlivých bitů v bitovém poli můžeme použít logických instrukcí mikroprocesoru. Prvním operandem logické instrukce bude bitové pole a druhým tzv. maska, která slouží k výběru jednotlivých bitů.

Potřebujeme-li nějaký bit bitového pole nastavit na logickou 1 a ostatní bity zachovat, použijeme logickou funkci OR a masku. Postupovat budeme tak, že bity, které chceme mít nastavené na logickou 1, nastavíme také v masce na log. 1 a provedeme logický součet bitového pole a masky. Ostatní bity, které jsou v masce nula, nezmění díky logické funkci OR svoji hodnotu v bitovém poli, protože 0 OR x je vždy x.

Chceme-li naopak některé bity v bitovém poli vynulovat, musíme nastavit masku tak, že všechny ostatní bity nastavíme na logickou 1. Na nule necháme v masce pouze ty bity, které si přejeme vynulovat. Provedeme-li logický součin bitového pole a masky, dostaneme požadované bity na logickou 0 a ostatní bity zachováme, protože 1 AND x je vždy x.

Ukažme si to na několika příkladech:

Příklad: V bitovém poli uloženém v registru AL nastavte všechny liché bity na logickou 1. Předchozí stav bitového pole není znám.

```
or al, 10101010b    ;v masce budou všechny liché bity na 1
```

Příklad: V bitovém poli uloženém v registru AL vynulujte nultý a sedmý bit. Předchozí stav ostatních bitů není znám.

```
and al, 01111110b   ;všechny bity, které si přejeme zachovat, nastavíme
                    ;v masce na 1
```

Instrukcí XOR s maskou můžeme dokonce změnit hodnotu určitých bitů, aniž bychom znali jejich předchozí (původní) hodnotu. V masce uvedeme logickou 1 všude tam, kde chceme bit v masce invertovat (změnit). Provedeme-li tuto operaci dvakrát, dostaneme opět původní bitové pole. Vlastnost funkce XOR se často používá v primitivním šifrování obsahu tak, že provedeme XOR s nějakou konstantní hodnotou (klíč) přes všechny byty, které chceme zašifrovat. Dešifrování probíhá stejně jako zašifrování, provedeme XOR se stejnou konstantní hodnotou přes všechny byty.

ZÁKLADNÍ PROGRAMOVÉ KONSTRUKCE

5

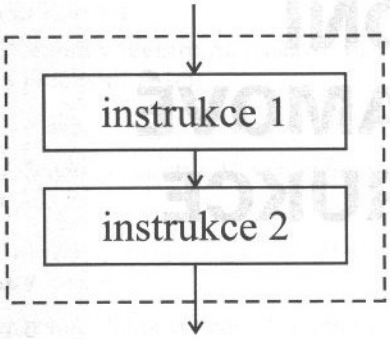
V této kapitole:

5.1 Sekvence
5.2 Konstrukce "IF THEN" – selekce
5.3 Konstrukce iterace – cykly
5.4 Instrukce pracující se zásobníkem

K provedení výpočtu jakékoliv úlohy zapsané v libovolném programovacím jazyce postačují tři základní řídicí programové konstrukce: sekvence, selekce a iterace. Postupně se seznámíme se všemi základními kameny programování, které i v prostředí Assembleru program zpřehlední a strukturují. Na konci kapitoly se seznámíme se zásobníkem, který budeme využívat při návrhu podprogramů.

5.1 Sekvence

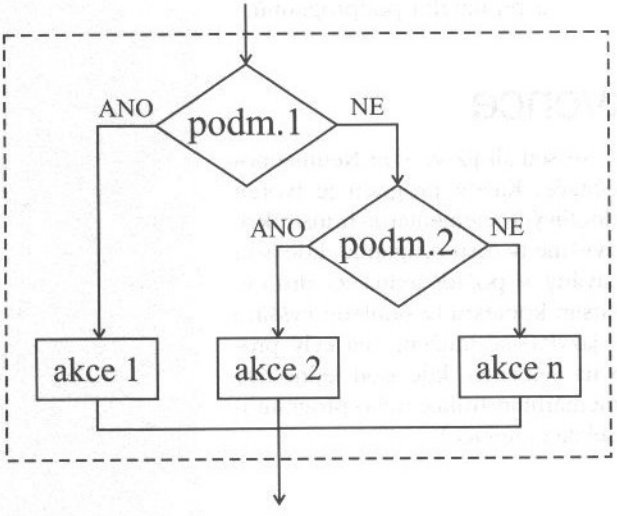
Se sekvencí jsme se setkali již ve von Neumannově koncepci počítače. Každý program je tvořen posloupností jednotlivých elementárních instrukcí. Sekvencí tak nazýváme ty části programu, kde jsou instrukce vykonávány v pořadí jedna za druhou (bez skoků). V širším kontextu (z pohledu vyššího programovacího jazyka) se můžeme na celý program dívat jako na sekvenci, kde jsou jednotlivé příkazy buď elementární instrukce nebo programové konstrukce (selekce, iterace).



Obrázek 14 Sekvence v programu

5.2 Konstrukce „IF THEN" – selekce

Selekci známe z vyšších programovacích jazyků pod příkazem if. Selekcí vybíráme jednu z několika možností podle splněné podmínky. Pro realizaci selekce budeme mít v Assembleru k dispozici instrukci porovnání a instrukci podmíněného a nepodmíněného skoku.



Obrázek 15 Selekce v programu

5.2.1 Instrukce CMP a TEST

Porovnání dvou operandů provádějí instrukce CMP a TEST. Jako operandy můžeme použít registry nebo operand uložený v paměti ve známých velikostech 8, 16 a 32 bitů.

Syntaxe: ***CMP o1, o2***

Název instrukce CMP je odvozen z anglického slova compare (porovnej). Instrukce je implementována podobně jako instrukce SUB. Operand o2 se odečte od operandu o1. Výsledek se nikam neuloží, změní se pouze příznaky v registru příznaků. Instrukci CMP můžeme použít na srovnání celých čísel bez znaménka i se znaménkem.

Instrukce TEST pracuje obdobně, místo rozdílu operandů se provede jejich bitový součin (AND). Výsledkem jsou nově nastavené příznaky v registru příznaků. Instrukci TEST používáme pro testování hodnot jednotlivých bitů v bitovém poli.

Ukažme si několik příkladů:

```
cmp ax,4            ;porovná registr AX s hodnotou 4
cmp dl,ah           ;porovná registr DL s registrem AH
cmp [prumer1],ax    ;porovná obsah "proměnné" prumer1 s registrem AX
cmp ax,[prumer1]    ;porovná registr AX s obsahem "proměnné" prumer1
cmp eax,ecx         ;porovná registry EAX a ECX

test ax,00000100b   ;testujeme bit 2 (třetí) bit, je-li nastaven
```

5.2.2 Instrukce nepodmíněného skoku JMP

Prvním způsobem, jak změnit sekvenční provádění instrukcí v programu, je použití instrukce nepodmíněného skoku. Instrukce přinutí procesor zpracovávat instrukce z jiného místa v paměti (přepíše hodnoty ukazatele na další instrukci, registr IP nebo i CS).

Syntaxe: ***JMP typ_skoku operand***

Nepodmíněný skok známý z vyšších programovacích jazyků pod názvem GOTO v Assembleru představuje instrukce JMP. Název vznikl z anglického slova jump (skoč). Instrukce vyžaduje jediný přímý operand, kterým je adresa místa v paměti, kam se má skočit (jako operand lze použít i univerzální registr naplněný adresou skoku, což bych začátečníkům nedoporučoval). V Assembleru jsou stejně jako ve vyšších programovacích jazycích adresy skoků reprezentovány návěštími (label).

Podle „vzdálenosti" skoku v programu rozlišujeme tři typy skoků: *short*, *near* a *far*. (krátký, blízký a vzdálený).

Maximální „délka" skoku (maximální absolutní rozdíl adres, které smíme skokem změnit) typu short je omezena. Ve druhém byte instrukce je uložena pouze osmibitová hodnota se znaménkem, proto adresa cíle skoku smí ležet pouze v rozmezí -128 až 127 byte. Skok se provede tak, že se osmibitová hodnota znaménkově rozšíří a přičte se k současné hodnotě registru (E)IP.

Instrukce skoku typu near již obsahuje ve své strojové reprezentaci novou hodnotu registru (E)IP, délka skoku je omezena pouze módem procesoru. V reálném režimu procesoru měníme instrukcí jmp near registr IP, proto se smíme pohybovat jen v rozmezí jednoho segmentu (64 KB), v chráněném režimu používáme registr EIP a tak cíl skoku smí ležet kdekoliv ve 4 GB adresového prostoru.

Skok far mění i hodnotu segmentového registru CS, který se podílí na výpočtu adresy instrukce v paměti. Součástí adresy skoku musí být i nová hodnota pro registr CS.

Není-li typ skoku uveden, předpokládá se typ *near*.

Nyní poněkud ve výkladu „přeskočíme" a řekneme si, jak vypadá návěští v Assembleru. Jedná se v podstatě o identifikátor zakončený dvojtečkou, jemuž byla při překladu přidělena adresa podle místa výskytu v programu. Ukažme si část programu, kde je použito návěští:

```
mov ax,4              ;do AX vlož hodnotu 4
novy_cyklus:          ;návěští novy_cyklus
mov bx,ax             ;do BX okopíruj AX
...
```

Chceme-li provést nepodmíněný skok na návěští novy_cyklus, napíšeme instrukci:

```
jmp novy_cyklus       ;skok na novy_cyklus
```

Po jejím provedení se program začne vykonávat od návěští novy_cyklus.

Nic nám nebrání nejprve provést skok a později v programu uvést návěští. Překladač pracuje „na více průchodů", a proto i takové dopředné skoky umí zpracovat. Část programu používající dopředné skoky by mohla vypadat takto:

```
jmp start             ;skok na start

cil:                  ;návěští cíl
...
...                   ;další instrukce
...
start:                ;návěští start
jmp cil               ;skok na cil
```

Vraťme se ještě krátce k typům skoků, které budeme sice jako začátečníci využívat velmi zřídka. Skok typu short použijeme všude tam, kde návěští „leží" do 128 B. Tak ušetříme 1B programového kódu, protože přeložená instrukce jmp short zabírá jen dva byte (instrukce JMP nebo JMP near 3 nebo 5 byte). Možná vás děsí představa, jak odhadnout vzdálenost návěští v nepřeloženém programu. Nemusíte se bát, prostě to vyzkoušejte, překladač v případě neúspěchu zahlásí chybu.

```
blizke_navesti:           ; návěští blizke_navesti
....                      ; nějaké další instrukce
....
jmp short blizke_navesti  ;provedeme skok na blizke_navesti
```

5.2.3 Instrukce podmíněných skoků Jx

Syntaxe: ***Jx návěští_cíle_skoku***

Dalším způsobem, jak změnit sekvenční provádění instrukcí v programu, je použití instrukce podmíněného skoku.

Instrukcí podmíněného skoku existuje celá řada. Navzájem se odlišují rozhodovací podmínkou, která řídí provedení skoku. V závislosti na vyhodnocení podmínky se vykonávání programu buď přesune na jiné místo nebo program pokračuje na následující adre-

se za instrukcí skoku. Rozhodovací podmínkou jsou stavy příznaků (několika nebo jen jednoho) příznakového registru procesoru.

Ukažme si nejpoužívanější instrukce, které skočí na návěští splněno, bude-li podmínka splněna.

```
jz splneno  ;skočí, je-li příznak ZF (zero flag, příznak nuly) nastaven
            ;na 1
jc splneno  ;skočí, je-li příznak CF (carry flag, příznak přenosu)
            ;nastaven na 1
js splneno  ;skočí, je-li nastaven příznak znaménka SF (signum flag)
jo splneno  ;skočí, je-li nastaven příznak přetečení
```

Všechny podmínky můžeme obrátit (negovat).

```
jnz splneno ;skočí, není-li příznak ZF (zero flag, příznak nuly)
            ;nastaven (je nula)
```

Obdobně i ostatní JNC, JNS a JNO.

V následující tabulce se seznámíme se všemi druhy skoků a podmínek, které je mikroprocesor schopen vykonat.

	o1==o2 o1=o2	o1!=o2 o1<>o2	o1>o2	o1<o2	o1=<o2	o1>=o2
Instrukce neuvažují znaménka	JE (JZ)	JNE (JNZ)	JA (JNBE)	JB (JNAE)	JNA (JBE)	JNB (JAE)
	jump if equal jump if zero	jump if not equal jump if not zero	jump if above jump if not below or equal	jump if below jump if not above or equal	jump if not above jump if below or equal	jump if not below jump if above or equal
	skok když: se rovnají nebo je výsledek nula	skok když: se nerovnají nebo výsledek není nula	skok když: je větší nebo není menší a rovno	skok když: je menší nebo není větší a rovno	skok když: není větší nebo je menší a rovno	skok když: není menší nebo je větší a rovno
Instrukce uvažují znaménka	JE (JZ)	JNE (JNZ)	JG (JNLE)	JL (JNGE)	JNG (JLE)	JNL (JGE)
	jump if equal jump if zero	jump if not equal jump if not zero	jump if greater jump if not less or equal	jump if less jump if not greater or equal	jump if not greater jump if less or equal	jump if not less jump if greater or equal
	skok když: se rovnají nebo je výsledek nula	skok když: se nerovnají nebo výsledek není nula	skok když: je větší nebo není menší a rovno	skok když: je menší nebo není větší a rovno	skok když: není větší nebo je menší a rovno	skok když: není menší nebo je větší a rovno

V prvním řádku tabulky jsou uvedeny požadované podmínky, při kterých se má skok vykonat. Na druhém řádku jsou zapsány názvy instrukcí skoku včetně jejich alternativních jmen uvedených v kulatých závorkách. Význam jednotlivých instrukcí je nejprve uveden v angličtině, tak aby bylo zřejmé, jak jména instrukcí vznikla.

Podobně jako u instrukcí MUL a DIV musíme i zde použít odlišné verze instrukcí skoku, budeme-li chtít pracovat s čísly se znaménkem. Důvodem jsou různé příznaky, které se vyhodnocují během rozhodování o skoku.

Všechny cílové adresy skoků musí ležet v rozmezí 128 B, jedná se implicitně o skoky typu short. Chceme-li skokem pokračovat dále na adresách mimo interval 128 byte, musíme použít typ skoku near.

Nyní můžeme s našimi znalostmi příslušných instrukcí Assembleru podmínku IF naprogramovat. Ukažme si jednoduchý příklad, ve kterém budeme chtít předat řízení programu (skočit) na návěští je_tri, je-li hodnota v registru AX rovna třem.

V prvním kroku porovnáme hodnotu v registru AX s číslem 3, což provedeme instrukcí CMP.

```
cmp ax,3          ;registr AX porovnej s hodnotou 3
```

Při porovnání na rovnost použijeme instrukci JZ, která je v tabulce zapsaná na druhém řádku a sloupci.

```
jz je_tri         ;skočí na návěští je_tri, bude-li hodnota AX = 3
```

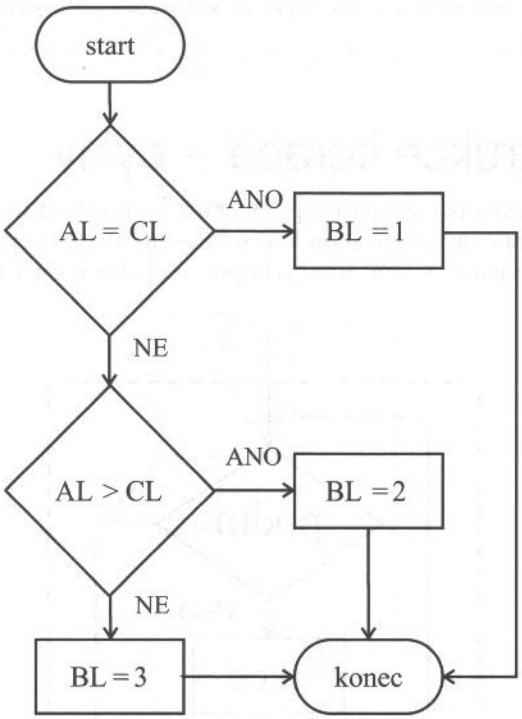
Všimněme si, že při porovnávání na rovnost čísel nezáleží na tom, jestli právě číslo chápeme jako se číslo se znaménkem nebo bez.

Instrukce JZ provede skok na návěští je_tri, pokud bude hodnota v registru AX rovna třem, jinak bude program pokračovat instrukcí následující po JZ.

V dalším příkladu budeme chtít porovnat bez znaménka hodnoty v registru CL a AL. Do registru BL zapsat jedničku, jsou-li si hodnoty rovny, dvojku, je-li AL větší než CL, a trojku, je-li v registru AL menší než v CL

Porovnáme registry AL a CL, výsledek porovnání se uloží do příznakového registru procesoru a použitím různých instrukcí skoku zajistíme výběr správné alternativy a tím i řešení problému.

```
cmp al,cl         ;porovnej hodnoty uložené v AL a CL
jz zapis_1        ;skoč na návěští zapis_1, jestliže AL = CL
cmp al,cl         ;porovnej hodnoty uložené v AL a CL
ja zapis_2        ;skoč na návěští zapis_2, jestliže AL > CL
mov bl,3          ;do BL zapiš 3
konec_if:         ;zde je konec našeho programu
...
zapis_1:          ;návěští zapis_1
mov bl,1          ;do BL zapiš 1
jmp konec_if      ;skok na konec našeho programu
zapis_2:          ;návěští zapis_1
mov bl,2          ;do BL zapiš 2
jmp konec_if      ;skok na konec našeho programu
```



Obrázek 16 Vývojový diagram příkladu

Museli jsme dokonce použít nepodmíněný skok, abychom vrátili řízení programu na původní místo. Řešení lze ještě podstatně vylepšit:

```
mov bl,1            ;do BL zapiš 1
cmp al,cl           ;porovnej AL a CL
je konec_if         ;skoč na konec programu, jestliže AL = CL
mov bl,2            ;do BL zapiš 2
cmp al,cl           ;porovnej AL a CL
ja konec_if         ;skoč na konec programu, jestliže AL > CL
mov bl,3            ;do BL zapiš 3
konec_if:           ;konec programu
```

Program jsme zkrátili a zpřehlednili, ale je možné ho ještě trochu vylepšit. Instrukce mov nemodifikuje nijak registr příznaků, proto je zbytečné al a cl znovu testovat. Zadanou úlohu můžeme vyřešit i takhle:

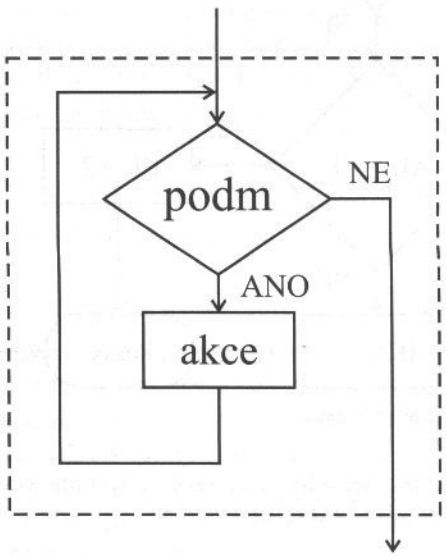
```
mov bl,1            ;do BL zapiš 1
cmp al,cl           ;porovnej AL a CL
jz konec_if         ;skoč na konec programu, jestliže AL = CL
mov bl,2            ;do BL zapiš 2
ja konec_if         ;skoč na konec programu, jestliže AL > CL
mov bl,3            ;do BL zapiš 3
konec_if:           ;konec programu
```

Problém vyřešený v Assembleru bude v jazyce C vyřešen ještě snadněji:

```
if (al == cl) bl = 1 else if (al > cl) bl = 2 else bl = 3;
```

5.3 Konstrukce iterace – cykly

Iterace, neboli cyklické opakování části programu v závislosti na uvedené podmínce, je poslední, avšak velmi často využívanou konstrukcí. Pro realizaci cyklu už potřebné příkazy, IF a GOTO, známe. S nimi jsme schopni implementovat i příkaz opakování – FOR.



Obrázek 17 Iterace v programu

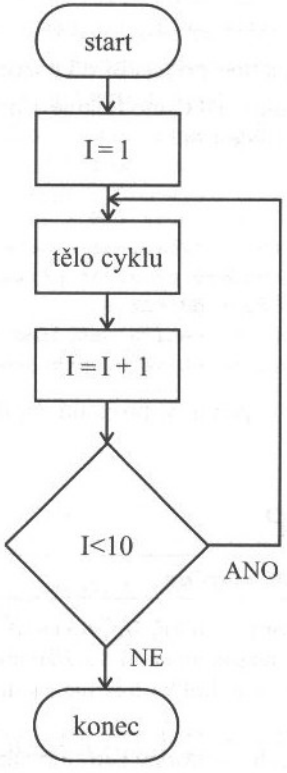
Ukažme si nyní na příkladu implementaci příkazu FOR s prázdným tělem cyklu (uvnitř cyklu nejsou vykonávány žádné instrukce).

Prvním problémem, který musíme vyřešit, je umístění řídicí proměnné cyklu. Instrukce porovnání nám nabízí dvě možnosti. Můžeme ji umístit v „proměnné" v paměti anebo, což je z hlediska optimalizačního výhodnější, v některém z registrů procesoru (ušetří se čas přesunu hodnoty proměnné do procesoru). Ukažme si implementaci příkazu FOR s prázdným tělem.

Zkusme implementovat for-cyklus, který se bude opakovat desetkrát, v každém průchodu se řídicí proměnná zvýší o 1. Zapišme ho nejprve příkazy IF a GOTO:

```
FOR_START:          ;návěští pro přípravu řídicí proměnné cyklu
I = 0               ;naplnění řídicí proměnné
FOR_SMYCKA:         ;návěští cyklu
...                 ;tělo cyklu (zde prázdné)
I=I+1               ;zvýšení řídicí proměnné
```

```
IF I < 10 THEN      ;test řídicí proměnné
GOTO FOR_SMYCKA     ;odskok na začátek cyklu nebo jeho ukončení
FOR_SKONCI:         ;koncové návěští cyklu
```



Obrázek 18 Vývojový diagram FOR cyklu

V Assembleru bude implementace obdobná. Zvolíme registr ECX pro řídicí proměnou I.

```
for_start:
mov ecx,0           ;naplníme registr ECX nulou
for_smycka:         ;na toto návěští se budeme vracet
...                 ;zde budeme provádět příkazy ve FOR-cyklu
inc ecx             ;ECX zvýšíme o 1
cmp ecx,10          ;srovnáme ECX s 10
jnz for_smycka      ;jestliže není 10, skočíme na for_smycka
for_skonci:         ;jestliže už je 10, pak skončíme
```

Ukažme si i variantu s proměnnou I umístěnou v paměti.

```
for_start:
mov dword [i],0     ;naplníme registr ECX nulou
```

```
for_smycka:          ;na toto návěští se budeme vracet
...                  ;zde budeme provádět příkazy ve FOR-cyklu
inc dword [i]        ;ECX zvýšíme o 1
cmp dword [i],10     ;srovnáme ECX s 10
jnz for_smycka       ;jestliže není 10, skočíme na for_smycka
for_skonci:          ;jestliže už je 10, pak skončíme
```

Druhá varianta bude mnohonásobně pomalejší díky častým přístupům do paměti.

Pokud si uvědomíme, že instrukce DEC modifikuje i příznak ZF (zero flag), můžeme for-cyklus zapsat také bez instrukce CMP.

```
for_start:
mov ecx,10           ;naplníme registr ECX 10
for_smycka:          ;na toto návěští se budeme vracet
...                  ;zde budeme provádět příkazy ve FOR-cyklu
dec ecx              ;odečteme od ECX 1
jnz for_smycka       ;jestliže není 0, skočíme na for_smycka
for_skonci:          ;jestliže už je 0, pak skončíme
```

Pro názornost: Týž FOR-cyklus v jazyce C program zapíšeme snadno:

for (i=0;i < 10;i++) {}

5.3.1 Instrukce LOOP

Syntaxe: ***LOOP návěští_smyčky***

V kapitole o procesoru 80386 jsme si řekli, že procesory kompatibilní s řadou x86 jsou „CISC", neboli mají komplexní instrukční sadu. Což znamená, že mají i složitější instrukce, které mohou nahradit činnost několika instrukcí jednoduchých, a to je i případ instrukce LOOP.

Instrukce LOOP má dva operandy, podobně jako instrukce MUL. První operand je pevně daný a je jím registr CX (nebo ECX) Druhý, volitelný operand je adresa návěští, kam se bude předávat řízení v případě splněné podmínky. Instrukce LOOP nejprve odečte jedničku od registru CX (ECX), a pokud výsledek není nula, pak předá řízení programu na místo určené svým volitelným operandem (cíl skoku musí ležet v intervalu + (nebo -) 128 bytů).

For-cyklus od 10 do 1 (včetně) vytvořený instrukcí LOOP by vypadal takto:

```
for_start:
mov cx,10            ;naplníme registr ECX 10
for_smycka:          ;na toto návěští se budeme vracet
...                  ;zde budeme provádět příkazy ve FOR-cyklu
loop for_smycka      ;jestliže CX není 0, skočíme na for_smycka
for_skonci:          ;jestliže už je 0, pak skončíme
```

5.3.2 Instrukce LOOPZ a LOOPNZ

Syntaxe: ***LOOPZ návěští_smyčky***
LOOPNZ návěští_smyčky

Instrukce LOOPZ rozšiřuje podmínku uskutečnění skoku. Skok na návěští uvedené v instrukci se uskuteční právě tehdy, když hodnota v registru CX (ECX) není rovna nule a zároveň je nastaven příznak ZF (zero flag) na 1. Synonymum instrukce je LOOPE.

Instrukcí LOOPZ snadno vytvoříme cyklus s dodatečnou podmínkou. Rozšiřme podmínku jednoho průchodu cyklu z předchozího příkladu o test registru BX na číslo 3. Cyklus se provede nejvýše desetkrát, ovšem za podmínky, že hodnota v registru BX = 3. Jinak se předčasně ukončí.

```
for_start:
mov cx,10           ;naplníme registr ECX 10
for_smycka:         ;na toto návěští se budeme vracet
...                 ;zde budeme provádět příkazy ve FOR-cyklu
                    ;
...                 ;možná zmena registru BX
                    ;
cmp bx,3            ;je BX = 3
loopz for_smycka    ;jestliže CX není 0 a BX = 3, skočíme na for_smycka
for_skonci:         ;jestliže už je 0 nebo BX není 3, pak skončíme
```

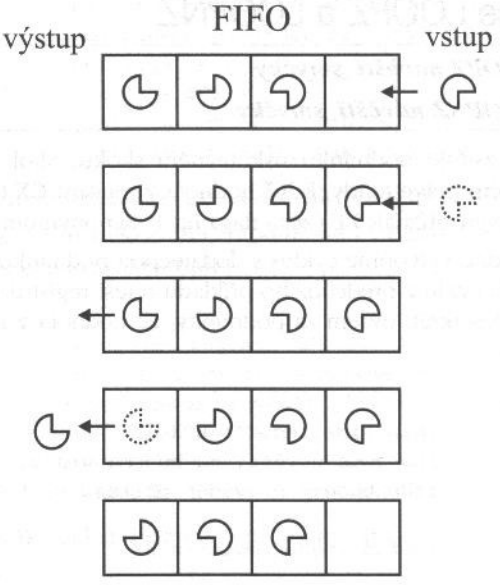
Instrukce LOOPNZ funguje analogicky, avšak s druhou podmínkou negovanou. Skok se uskuteční, není-li registr CX (ECX) roven nule a zároveň není nastaven příznak ZF (zero flag je nula). Její synonymum je LOOPNE.

5.4 Instrukce pracující se zásobníkem

Při programování potřebujeme často dočasně ukládat registry či obsah jednotlivých paměťových míst a po určité době se k nim zase vracet. Instrukční sada mikroprocesoru nám při řešení tohoto požadavku vychází vstříc speciálními instrukcemi, které pracují s paměťovou strukturou nazvanou zásobník.

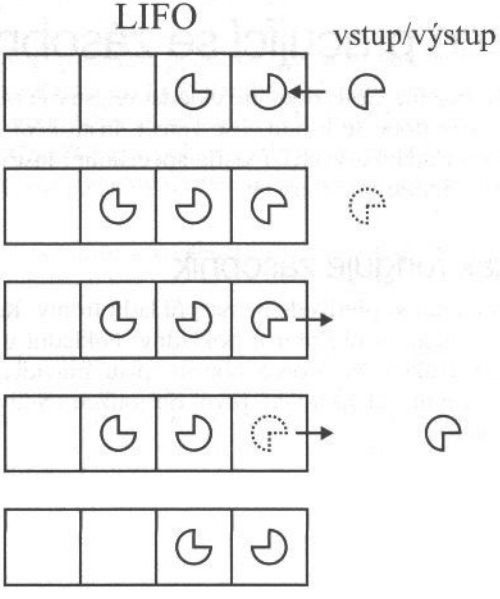
5.4.1 Co je a jak funguje zásobník

Vysvětlení funkce zásobníku si předvedeme na příkladu fronty. Každý z nás určitě stál ve frontě v obchodě a čekal na obsluhu u pokladny. Pokladní u pokladny obsluhuje frontu požadavků (zákazníků). Ve frontě obecně platí pravidla, že zákazník, který vstoupil do fronty jako první, odchází také první obsloužen (Neuvažujeme předbíhání, to by byla prioritní fronta).



Obrázek 19 Funkce FIFO

Frontě tohoto typu se říká FIFO (first in, first out – první dovnitř, první ven). Existuje však i druhý typ fronty – LIFO (last in, first out – poslední dovnitř, první ven). S takovou frontou bychom sice v obchodě neuspěli, ale běžně se s ní setkáváme. Frontu LIFO můžeme spatřit v podobě narůstajícího stohu papírů na pracovním stole. Chceme-li se dostat hlouběji, musíme nejprve odstranit všechny papíry z vrcholu hromady.



Obrázek 20 Funkce LIFO

Na stejném principu jako hromada papíru, na principu fronty LIFO, funguje i obsluha dat v zásobníku. Data, která jsme do zásobníku vložili naposledy, musíme vybrat jako první.

Počítačový zásobník není implementován na našem stole, ale prostor pro jeho realizaci je vyhrazen v hlavní paměti (existují i počítače (nikoli však PC), které mají zásobník implementován hardwarově, ne v hlavní paměti). Na vrchol zásobníku ukazuje dvojice registrů SS:SP (SS:ESP) – stack segment, stack pointer.

Na počítačích PC roste zásobník směrem dolů, což znamená, že data, která na zásobník ukládáme, budou na nižších adresách než data již uložená. Přesná adresa místa uložených dat nás nezajímá, protože kdykoli provedeme operaci se zásobníkem, změní se i registr SP (ESP), tak aby ukazoval vždy na vrchol zásobníku. Na zásobník můžeme uložit data, která jsou buď šestnáctibitová, nebo jen dvaatřicetibitová.

Mikroprocesor je vybaven dvojicí instrukcí, které realizují operace se zásobníkem.

5.4.2 Instrukce PUSH a POP

Syntaxe: ***PUSH o1***

Pomocí instrukce PUSH můžeme na zásobník uložit libovolný šestnáctibitový nebo dvaatřicetibitový registr nebo obsah paměťového místa (nepoužívá se příliš často).

```
push eax            ;na zásobník uložíme registr EAX
```

Instrukci push můžeme rozepsat instrukcemi:

```
sub esp,4           ;sniž ESP o 4
mov [ss:esp], eax   ;zapiš do zásobníku hodnotu registru EAX
```

nebo obecněji (s pomocí operátoru sizeof, vypůjčeného z vyšších prog. jazyků):

```
push o1
```

můžeme zapsat jako

```
(E)SP=(E)SP-sizeof(o1)
o1 -> SS:[(E)SP]
```

Instrukce POP přesune do operandu hodnotu (obsah) z vrcholu zásobníku a vrcholem zásobníku se stane hodnota uložená na vyšší adrese. Instrukce POP smí mít stejný typ operandů jako instrukce PUSH.

Opět lze instrukci POP přepsat pomocí instrukcí MOV a ADD:

```
mov eax,[ss:esp]    ;načti z vrcholu zásobníku do registru EAX
add esp,4           ;"smaž" nejvyšší dvojslovo ze zásobníku
```

Uveďme si několik příkladů.

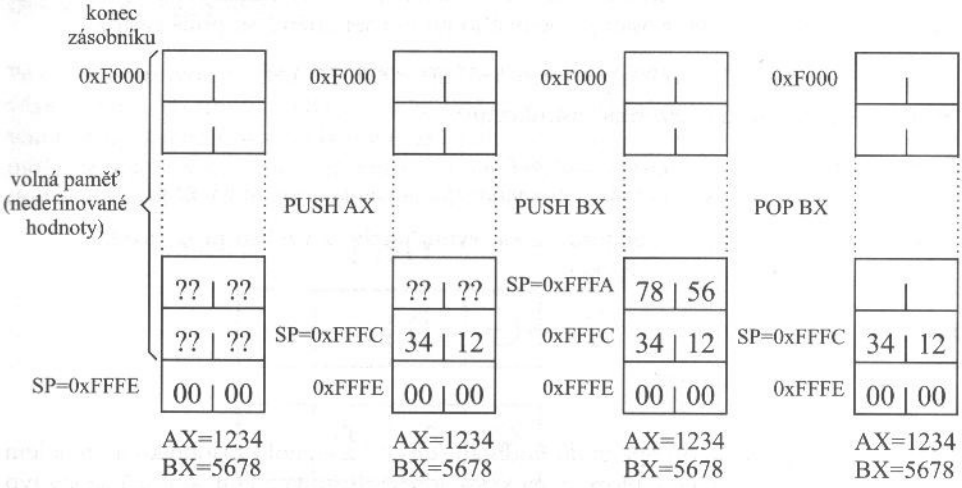
```
push eax    ;na zásobník ulož hodnotu uloženou v registru EAX
push esi    ;na zásobník ulož hodnotu registru ESI
...
pop eax     ;ze zásobníku vyber hodnotu a ulož ji v registru EAX
pop esi     ;ze zásobníku vyber hodnotu a ulož ji v registru ESI
```

Kdybychom program vyzkoušeli, tak po vykonání instrukce pop esi by byly původní hodnoty registrů EAX a ESI přehozené. Zásobník je fronta LIFO a vybrat poslední uloženou hodnotu musíme jako první. Správně zapsaný program by vypadal takto:

```
push eax     ;na zásobník ulož hodnotu uloženou v registru EAX
push esi     ;na zásobník ulož hodnotu registru ESI
...          ;modifikace registrů
pop esi      ;ze zásobníku vyber hodnotu a ulož ji v registru ESI
pop eax      ;ze zásobníku vyber hodnotu a ulož ji v registru EAX
```

Z důvodu zachování kompatibility s předchozími procesory je možné na zásobník ukládat i šestnáctibitové varianty registrů. Ukažme si příklad, včetně hypotetického chování zásobníku.

```
mov ax,0x1234        ;do AX vlož 0x1234
mov bx,0x5678        ;do BX vlož 0x5678
push ax              ;na zásobník ulož hodnotu uloženou v registru AX
push bx              ;na zásobník ulož hodnotu registru BX
...                  ;možná modifikace registrů
pop bx               ;ze zásobníku vyber hodnotu a ulož ji v registru BX
...
```



Obrázek 21 Změna zásobníku při instrukcích PUSH a POP

Před první instrukcí push je na vrcholu zásobníku uložena hodnota 0x0000 (na vrchol zásobníku ukazuje dvojice registrů SS:SP). Předpokládejme, že hodnota uložená v registru SP byla 0xFFFE. Po vykonání instrukce PUSH ax se snížil ukazatel na vrchol zásobníku SP o 2 na 0xFFFC a zároveň se na vrchol zásobníku zapsala hodnota 0x1234. Druhá instrukce, PUSH bx, sníží opět ukazatel na vrchol zásobníku o dvě na SP = 0xFFFA a zapíše na nový vrchol hodnotu 0x5678. Následující instrukce POP bx vybere z vrcholu zásobníku uloženou hodnotu (0x5678) a vloží ji do registru BX. Ukaza-

tel na vrchol se změní na předcházející hodnotu SP = 0xFFFC, na vrcholu zásobníku zůstává uložena hodnota 0x1234.

Připomeňme si, že na zásobník nelze pomocí instrukce PUSH uložit osmibitové registry a že také nelze přímo uložit obsah registru IP (EIP) (instruction pointer, ukazatel na další instrukci). Instrukce PUSH/POP ip nebo PUSH/POP eip neexistuje (lze ji ale nahradit jinak, jak si ukážeme dále).

5.4.3 Instrukce PUSHA/POPA a PUSHAD/POPAD

Syntaxe: ***PUSHA***
POPA

Někdy potřebujeme uložit všechny univerzální registry na zásobník. Za tímto účelem je mikroprocesor vybaven instrukcemi PUSHA a POPA, které na zásobník uloží (PUSHA) nebo ze zásobníku obnoví (POPA) hodnoty všech univerzálních šestnáctibitových u registrů. Instrukce PUSHA a POPA nemají žádný volitelný operand.

Instrukce PUSHA/POPA byla zavedena v předchůdci procesoru 80386, a proto neukládá dvaatřicetibitové registry (nemůže, protože neexistovaly). Chceme-li je uložit nebo vybrat, pak musíme použít „nové" instrukce PUSHAD a POPAD.

Pořadí ukládaných registrů na zásobníku od vrcholu dolů je:

(E)AX, (E)CX, (E)DX, (E)BX, původní (E)SP, (E)BP, (E)SI, (E)DI

```
pusha          ;uložíme na zásobník obsah všech univerzálních registrů

               ;provedeme velké změny (například komplikovaný výpočet,
               ;který změní většinu hodnot univerzálních registrů)
...
popa           ;zase všechny registry obnovíme
```

5.4.4 Instrukce PUSHF/POPF a PUSHFD/POPFD

Syntaxe: ***PUSHF***
POPF
PUSHFD
POPFD

Předchozí instrukce neposkytovaly žádnou možnost uložení registru příznaků. V procesorech vyrobených před procesorem 80386 byl registr příznaků jen šestnáctibitový a na zásobník se příznaky uložily, resp. vybraly, pomocí instrukce PUSHF, respektive POPF.

Podobně jako u instrukce PUSHA/POPA přibylo v dvaatřicetibitové variantě na konci písmeno D. Instrukce ukládající, respektive vybírající, celý dvaatřicetibitový registr příznaků se jmenuje PUSHFD, respektive POPFD. (Všechny operace, se kterými jsme se seznámili, neovlivňují horních šestnáct bitů příznakového registru, proto stačí, když budeme používat instrukce PUSHF/POPF)

```
cmp ax,bx    ;porovnáme registry AX a BX
pushf        ;výsledek porovnání uložíme na zásobník
...          ;zde můžeme provádět operace, které modifikují registr
             ;příznaků
add di,4     ;např. sčítání
popf         ;registr příznaků obnovíme
jz stejne    ;pokud se AX a BX rovnají, skočíme na návěští stejné
```

5.4.5 Instrukce CALL a RET

Existuje staré androidní rčení, které se sem teď hodí. Je to nějak takhle: „Když neodskočíš z cyklu programu, nikdy nebudeš v podprogramu."

(Kryton, Série IV, Epizoda 3, Spravedlnost)

Syntaxe: ***CALL typ_volání operandy***

RET

Instrukce CALL a RET používáme k implementaci podprogramů. Instrukce CALL má jeden, obvykle přímý operand (jako u instrukce skoku se připouští operand v registru nebo v paměti), kterým je adresa místa v paměti, kam se předá řízení. Na rozdíl od instrukce JMP se do zásobníku uloží hodnota registru IP (EIP) následující instrukce po instrukci CALL.

Parametr typ volání je podobný jako u instrukce JMP, není-li uveden, bude se předpokládat typ volání typu *near*, a tudíž *se* bude ukládat a měnit pouze registr IP (EIP). Při typu volání *far* se na zásobník kromě registru IP (EIP) uloží i hodnota segmentového registru CS (adresa volání se tak bude skládat z hodnoty pro registr IP (EIP) a hodnoty pro registr CS).

Návrat z podprogramu zajišťuje instrukce RET, která ze zásobníku vybere hodnotu z vrcholu a tu uloží do registru IP (EIP), čímž se začnou vykonávat instrukce uložené za volající instrukcí CALL. Pro návrat z podprogramu, který byl zavolán instrukcí CALL far, musíme obnovit i původní hodnotu registru CS, což můžeme zajistit použitím instrukce RETF, která ze zásobníku nejprve vybere hodnotu a uloží ji do registru IP, a pak vybere ještě jednu, kterou uloží do registru CS.

Situaci můžeme ještě poněkud zkomplikovat.

Instrukce RET nebo RETF může mít jako přímý operand počet položek, které se mají ze zásobníku vybrat po naplnění registru IP (EIP) (nebo i CS). Tento speciální tvar instrukce RET využijeme až v kapitole nazvané „Propojujeme", kde si povíme o možnostech propojení vyšších programovacích jazyků a Assembleru.

Ukažme si použití instrukcí CALL a RET na příkladu:

Pomocí podprogramu sečtěte dvě čísla uložená v registrech EAX a EBX, výsledek součtu uložte v registru ECX (ignoruje přetečení). Hodnoty uložené v registrech EAX a EBX musí být zachovány i po návratu z podprogramu.

Utvoříme podprogram secti. Nejprve musíme vyřešit předávání parametrů podprogramu. Ve vyšších programovacích jazycích se pro předání parametrů používá zásobník, my si prozatím vystačíme s dohodnutými registry (vhodné by bylo použít registry zada-

né). Jakmile provedeme instrukci pro sečtení ADD, bude jedna hodnota operandu přepsána výsledkem. Využijeme zásobník a hodnotu uloženou v postiženém registru si uložíme a později opět obnovíme.

```
secti:
push eax            ;uložíme hodnotu registru EAX na zásobník
add eax,ebx         ;sečteme EAX a EBX, výsledek bude v EAX
mov ecx,eax         ;výsledek z EAX uložíme do ECX
pop eax             ;obnovíme původní hodnotu v registru EAX
ret                 ;návrat z podprogramu.
```

Podprogram secti vyzkoušíme s hodnotami 4 a 8.

```
mov eax,4           ;do EAX uložíme 4
mov ebx,8           ;EBX = 8
call secti          ;zavoláme podprogram secti
                    ;Výsledek bude uložen v registru ECX a bude jím
                    ;hodnota ECX = 0x0000000C
```

Co by se stalo kdybychom vynechali instrukci POP eax? Instrukce RET by tak předala řízení na adresu, která byla původně uložena v registru EAX, což by vedlo k pádu programu a v horším případě pádu systému. Podobně opomenutí instrukce RET by vedlo k vykonávání instrukcí dávno nepatřících našemu podprogramu, což také snadno (a zcela určitě) vede k pádu programu.

Podprogram secti lze v našem jednoduchém případě zjednodušit tak, že se obejdeme bez instrukcí PUSH a POP.

```
secti:
mov ecx,eax         ;hodnotu registru EAX (první parametr) překopírujeme
                    ;do registru ECX
add ecx,ebx         ;a sečteme s druhým parametrem v registru EBX,
                    ;výsledek bude v požadovaném registru ECX
ret                 ;návrat z podprogramu.
```

5.4.6 Instrukce INT a IRET

Připomeňme si, co je přerušení.

Přerušení je událost, při které procesor přestane vykonávat instrukce právě prováděného programu a dočasně začne vykonávat program jiný, obsluhující přerušení. Po ukončení obsluhy přerušení se obnoví provádění předchozího programu.

V kapitole věnované přerušení jsme zatím obecně rozdělili možné zdroje přerušení do dvou kategorií – přerušení softwarová a přerušení hardwarová.

Instrukcí INT můžeme vyvolat softwarové přerušení. Jedná se o přerušení „na požádání", kdy programově procesoru sdělíme, které zvolené přerušení si přejeme vykonat. Instrukce INT vyžaduje jediný přímý osmibitový operand, kterým je číslo požadovaného přerušení.

Syntaxe: ***INT op***

O hardwarové přerušení žádají procesor periferní zařízení, která jsou připojena ke společné sběrnici (ISA nebo PCI). Žádá-li zařízení o obsluhu, vygeneruje požadavek na přerušení, kterému se říká IRQ (interrupt request). Již jsme si řekli, že těchto požadavků smí být jen šestnáct, protože sběrnice ISA měla vyhrazeno jen 16 vodičů pro tyto účely. Požadavky na přerušení jsou zavedeny do řadiče přerušení, který o přerušení zažádá mikroprocesor. „Současně" mu i s žádostí sdělí číslo požadovaného přerušení. Po startu počítače a pod operačním systémem DOS odpovídá žádost o přerušení IRQ 0 přerušení 8.

Jakmile procesor zjistí číslo přerušení, uloží na zásobník kontext právě prováděného programu. Ekvivalentem záložky v knížce jsou pro procesor hodnoty registrů ukazující na další prováděnou instrukci (CS a (E)IP) a obsah příznakového registru.

Dalším úkolem procesoru je nalezení adresy programu, který provede obsluhu přerušení (vykoná se). Často se takové adrese říká vektor přerušení, který je společně s ostatními uložen v hlavní paměti počítače ve speciální tabulce vektorů přerušení. Můžeme si ji představit jako pole adres podprogramů, jejichž indexem je číslo přerušení.

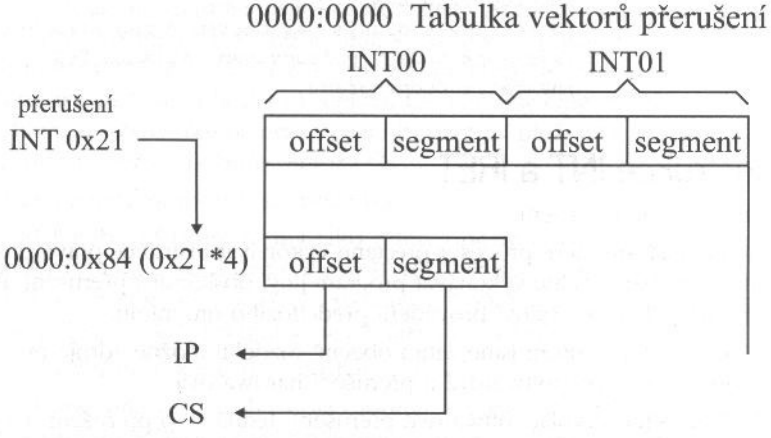
Posledním krokem je zápis vektoru přerušení do dvojice registrů CS a (E)IP. Tak se začne další instrukce provádět již z programu obsluhy přerušení.

V kapitole věnované operačnímu systému DOS budeme jeho služby využívat prostřednictvím přerušení 0x21, které budeme generovat instrukcí:

```
int 0x21    ;zavoláme DOS
```

Syntaxe: ***IRET***

Návrat z přerušení zajišťuje instrukce IRET, která vybere ze zásobníku původní hodnoty registru (E)IP, CS a příznaků.



Obrázek 22 Plnění registrů CS a IP novými hodnotami

Ukažme si na obrázku, co se stane, vyvolá-li program přerušení 0x21. Předpokládejme, že pracujeme v reálném režimu procesoru, a tudíž používáme jen šestnáctibitovou adresaci.

Prvním krokem procesoru je zjištění čísla přerušení. Instrukce INT generuje softwarové přerušení a tak číslo přerušení je jejím operandem. V následujícím kroku procesor uloží kontext programu na zásobník. Postup ilustrujme i instrukcemi Assembleru:

```
pushf              ;instrukce uloží obsah příznakového registru na
                   ;zásobník
push cs            ;instrukce uloží obsah registru CS na zásobník
push ip            ;instrukce neexistuje, ale předpokládejme, že uloží
                   ;hodnotu registru IP na zásobník - nasimulovat lze
                   ;jako:
                   ;call sem
                   ;sem:
```

Posledním krokem je skok na správnou adresu, která je reprezentovaná vektorem přerušení v tabulce vektorů přerušení. Použít můžeme instrukci JMP far, která z paměťového místa určeného operandem vybere nové hodnoty registrů CS a IP.

Poněkud předběhněme výklad a spokojme se s faktem, že počátek tabulky vektorů přerušení leží hned na počátku adresovatelné paměti, tedy na 0x0000:0x0000. Každý z vektorů přerušení zabírá čtyři byte. První dva byte určují novou hodnotu registru IP (offset) a druhé dva byte určují novou hodnotu registru CS (segment). Vektor přerušení 0x21 bude ležet na adrese 0x0000:(0x21*4), instrukci JMP far můžeme nyní zapsat jako:

```
jmp far [0x21*4]   ;instrukce vykoná skok na zvolenou adresu.
                   ;(předpokládejme, že je registr DS vynulován,
                   ;abychom adresovali paměť od 0x0000: . . . )
```

Celou instrukci INT si můžeme nasimulovat instrukcemi PUSHF a CALL far.

```
pushf              ;uložíme příznaky na zásobník
call far [0x21*4]  ;uložíme CS i IP na zásobník a provedeme skok.
```

Softwarová přerušení se často používají jako brána pro volání služeb operačního systému. Více si povíme v kapitolách věnovaných jednotlivým operačním systémům.

OSTATNÍ INSTRUKCE

6

V kapitole „Ostatní instrukce" se seznámíme s dalšími často používanými instrukcemi z bohaté instrukční sady procesorů x86. S jejich využitím můžete psát i náročnější programy řešící širokou škálu úloh nejrůznější povahy.

V této kapitole:

6.1 Instrukce přímo měnící registr příznaků
6.2 Instrukce XCHG
6.3 Instrukce LEA
6.4 Instrukce pro práci s řetězci
6.5 Instrukce pro vstup a výstup (I/O)
6.6 Instrukce posuvů a rotací
6.7 Pseudoinstrukce
6.8 Triky s instrukcemi

6.1 Instrukce přímo měnící registr příznaků

Některé instrukce umožňují přímou změnu registru příznaků, my se seznámíme se čtveřicí instrukcí ovládající příznaky IF a DF (příznak přerušení a směru).

6.1.1 Instrukce CLI a STI

Syntaxe: ***CLI***
STI

Instrukcemi CLI a STI se ovládá hodnota příznaku IF (interrupt flag, příznak přerušení). Je-li příznak nastaven na jedna, hardwarová přerušení jsou povolena. Instrukcí CLI (Clear Interrupt) se příznak nastaví na nulu. Povolit přerušení musíme použitím instrukce STI.

```
cli             ;zakážeme přerušení, použijeme pouze v prostředí OS DOS

...             ;provedeme nějakou operaci, při které si nepřejeme být
                ;"vyrušeni", například modifikace tabulky vektorů přerušení

sti             ;přerušení zase povolíme
```

6.1.2 Instrukce STD a CLD

Syntaxe: ***STD***
CLD

Instrukce STD a CLD jsou dvojicí instrukcí manipulujících s příznakem směru (DF). Příznak směru se využívá v sadě instrukcí pro práci s řetězci, o kterých si řekneme více v následující kapitole. Instrukce CLD nastavuje příznak směru na hodnotu nula (čítání nahoru). A instrukce STD nastavuje příznak směru DF zase na jedna (čítání dolů).

6.2 Instrukce XCHG

Syntaxe: ***XCHG o1,o2***

Často nastane situace, kdy bude třeba prohodit hodnoty dvou registrů. Mikroprocesory řady x86 mají pro výměnu operandů speciální instrukci, která se jmenuje XCHG – její název je odvozen ze slova exchange (vyměnit). Podobně jako instrukce MOV má dva operandy (o1, o2), které mohou být ve známých třech velikostech a pouze jeden z nich v paměti (platí stejná omezení jako pro instrukci MOV).

```
xchg eax,eax              ;vymění registr EAX s registrem EAX, takto je
                          ;implementována instrukce NOP
xchg ebx,ecx              ;prohodí obsah registrů ecx a ebx.
xchg al,ah                ;vymění obsah registrů al a ah
xchg dl,ah                ;vymění obsah registrů dl a ah
xchg byte [promena],cl    ;prohodí obsah paměťového místa zastoupený
                          ;symbolem promena a registru CL
```

6.3 Instrukce LEA

Syntaxe: ***LEA o1, [o2]***

Instrukce LEA dostala svůj název podle počátečních písmen anglických slov Load Effective Address – nahraj efektivní adresu. Instrukce spočítá efektivní adresu svého druhého operandu a umístí ji do operandu o1 (první operand smí být pouze registr). Ačkoli instrukce vyžaduje hranaté závorky kolem druhého operandu, žádnou skutečnou paměť neadresuje. Instrukce dovoluje použít všechny druhy

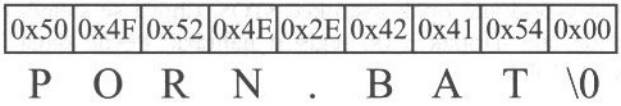
složitějších adresací, tak jak jsme se s nimi seznámili v kapitole věnované adresaci paměti.

Instrukci LEA můžeme využít v případě, že budeme výsledek nějaké složité adresace ještě potřebovat.

```
lea edi, [ebx*4+ecx]        ;do registru EDI se uloží adresa, která je
                            ;výsledkem výpočtu EDI = EBX*4+ECX.
```

S instrukcí LEA nemusíme vypočítávat jen adresy, výhodně se používá i při celočíselné aritmetice. S příklady věnovanými „nekonvenčnímu" použití instrukce LEA se seznámíme v kapitole o optimalizaci.

6.4 Instrukce pro práci s řetězci



Obrázek 23 Ukázka řetězce ukončeného nulovým byte

Řetězcem v Assembleru rozumíme posloupnost znaků (bytů) zakončených nulovým bytem (jako v řetězci jazyka C).

Instrukční sada mikroprocesorů x86 kompatibilních je vybavena instrukcemi pro snadnou práci s řetězci. Instrukce mohou být nahrazeny několika jednoduššími, stejně jako v případě instrukce LOOP. Program zjednodušíme a zpřehledníme, budeme-li tyto instrukce používat.

Instrukce pro práci s řetězci můžeme rozdělit podle velikosti operandu na tři známé skupinky. Instrukce pro osmibitové operandy, jejichž název je zakončen písmenem B (byte), instrukce pro šestnáctibitové operandy s názvem zakončeným písmenem W (word) a konečně instrukce pracující s dvaatřicetibitovým operandem, jejichž název je ukončen písmenem D (double word). Všechny instrukce v této kapitole mají pevně dané operandy, které nelze měnit.

Speciálním příznakem směru D (direction), uloženým v příznakovém registru procesoru, můžeme měnit směr čítání registrů použitých v řetězcových operacích.

6.4.1 Instrukce STOSx

Syntaxe:	***STOSB***
	STOSW
	STOSD

Instrukce STOSB nakopíruje registr AL na adresu danou dvojicí registrů ES:(E)DI a registr (E)DI sníží nebo zvýší o jedna podle vlajky směru (jaký registr se použije (DI nebo EDI), závisí na režimu procesoru).

Obdobně pracuje i instrukce STOSW, ale kopírovaný registr je AX a hodnota v registru (E)DI se zvětší nebo zmenší o dva. Instrukce STOSD pracuje s registrem EAX a registr (E)DI se sníží nebo zvýší o čtyři v závislosti na vlajce směru.

```
cld         ;nastaví příznak směru na čítání nahoru
stosw       ;zapíše hodnotu v registru AX na ES:[DI] nebo na ES:[EDI]
            ;(podle režimu procesoru) a zvýší hodnotu v registru
            ;(E)DI o 2
```

6.4.2 Instrukce LODSx

Syntaxe: ***LODSB***
LODSW
LODSD

Instrukce LODS je v podstatě opak instrukce STOS. Chová se stejně, ovšem hodnotu z registru neukládá, ale naopak ji do registru nahrává. Pevně daná dvojice je však DS:(E)SI, nikoliv ES:(E)DI.

6.4.3 Instrukce CMPSx

Syntaxe: ***CMPSB***
CMPSW
CMPSD

Instrukce CMPSB, CMPSW a CMPSD provede porovnání (CMP) hodnot nahraných z adres určených registry ES:(E)DI a DS:(E)SI a provede zvýšení nebo snížení registrů (E)DI a (E)SI o velikost operandu v závislosti na vlajce směru.

6.4.4 Instrukce SCASx

Syntaxe: ***SCASB***
SCASW
SCASD

Instrukce SCAS/B/W/D porovnává hodnotu v registru AL/AX/EAX s paměťovým místem určeným dvojicí registrů [ES:(E)DI]. Obsah registru (E)DI se změní podle vlajky směru.

6.4.5 Instrukce REP a REPZ

Syntaxe: ***REP***
REPZ

Instrukce REP se užívá podobně jako instrukce LOOP. Instrukce REP je prefixová instrukce, která vykonává následující instrukci tak dlouho, dokud je hodnota v registru (E)CX různá od nuly. V každém opakování sníží hodnotu v registru (E)CX o jedna. Používá se nejčastěji s instrukcemi MOVS nebo STOS.

```
rep
movsb       ;Zkopíruje prvních (E)CX bytů z adresy DS:(E)SI na adresu
            ;ES:(E)DI,
            ;což je assemblerovým ekvivalentem C funkce memcpy.
```

nebo

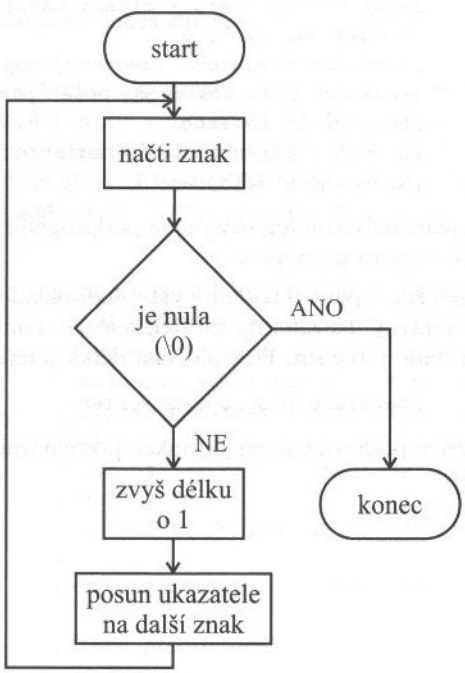
```
rep
stosb       ;na adresu ES:(E)DI okopíruj (E)CXkrát hodnotu v registru
AL,         ;instrukce rep stosb má v jazyce C ekvivalent ve funkci
            ;memset
```

Instrukce REPZ/REPE podobně jako instrukce LOOPZ zjemňuje požadavky ukončení opakování. Aby se provedlo další opakování, registr (E)CX nesmí být nulový a zároveň nesmí být nastaven příznak ZF. V instrukci REPZ/REPE lze negovat druhou podmínku a vznikne instrukce REPNZ/REPNE.

Prefixová instrukce REPZ se často používá s instrukcemi SCAS nebo CMPS.

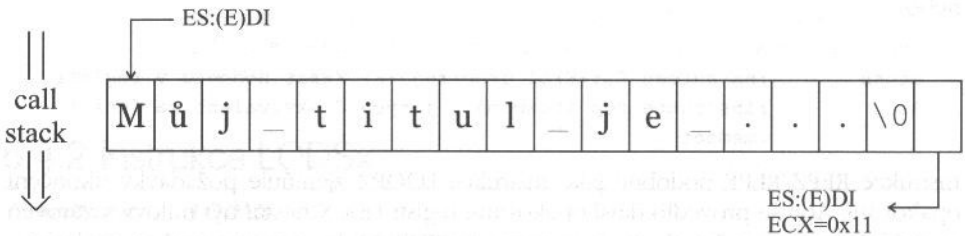
```
repz        ;podmínečně opakuj instrukci SCASB
scasb
nebo
repz        ;podmínečně opakuj instrukci CMPSB
cmpsb
```

Již máme dost znalostí, abychom si mohli ukázat často používaný podprogram (funkci), který zjišťuje délku řetězce. V jazyce C se nazývá strlen.



Obrázek 24 Vývojový diagram funkce strlen

Funkce vyžaduje jediný parametr, předávaný v registrech ES:(E)DI, kde je uložen ukazatel na řetězec. Funkce vrátí v registru ECX jeho délku včetně nulového bytu. Ukazatel uložený v registrech ES:(E)DI bude po návratu ukazovat na další byte za nulovým bytem.



Obrázek 25 Stav registrů ES:(E)DI před a po volání funkce strlen

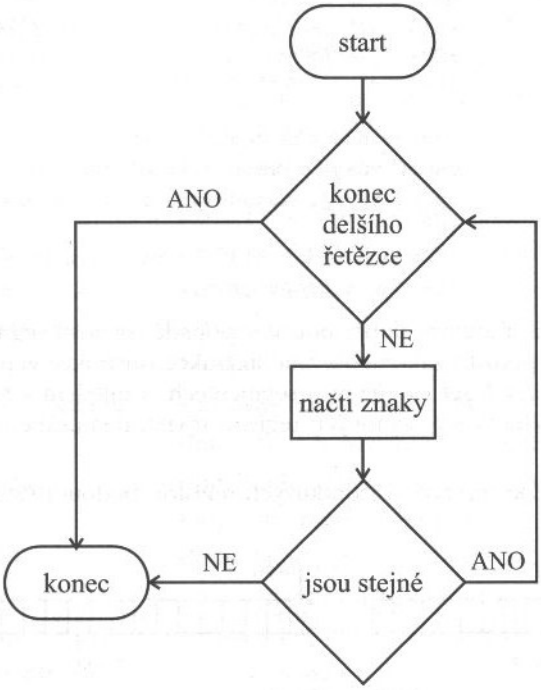
```
strlen:
push eax            ;ulož hodnotu registru EAX na zásobník
xor ecx,ecx         ;vynuluj registr ECX, ekvivalentní je instrukce
                    ;mov ecx,0
xor eax,eax         ;vynuluj registr EAX
dec ecx             ;sniž hodnotu v registru ECX o jedna - dostaneme
                    ;0xFFFFFFFF jako maximální délku řetězce, kterou
                    ;budeme prohledávat
cld                 ;smaž vlajku směru - čítání nahoru
repne scasb         ;hledej nulový byte
neg ecx             ;jestliže znegujeme (dvojkový doplněk) hodnotu v
                    ;registru ECX, dostaneme počet provedených opakování,
                    ;tedy délku řetězce.
pop eax             ;obnovíme původní obsah registru EAX
ret                 ;Ukončíme podprogram
```

Pro použití s pouze šestnáctibitovými registry půjde podprogram upravit snadno. Vynecháme všude v názvech registrů písmeno e.

Zatím ani nevíme, jak řetězec v paměti nadefinovat, předpokládejme, že jsme již do registrů ES:(E)DI nahráli správný ukazatel na začátek řetězce. Funkci zavoláme instrukcí CALL, po jejím návratu bude v registru ECX uložena délka „měřeného" řetězce.

```
call strlen         ;zavoláme podprogram strlen
```

Druhým často používaným podprogramem je funkce porovnávající dva řetězce, v jazyce C známá jako strcmp.



Obrázek 26 Vývojový diagram funkce strcmp

Parametry podprogramu budou ukazatelé na oba porovnávané řetězce. První bude uložen v registrech ES:(E)DI a druhý v registrech DS:(E)SI. Jestliže budou řetězce shodné, bude po skončení podprogramu hodnota registru ECX nulová, jinak bude obsahovat počet shodných znaků. Registry (E)SI a (E)DI budou ukazovat na první rozdílný znak, nebo za řetězec v případě shody. Podprogram používá funkci strlen.

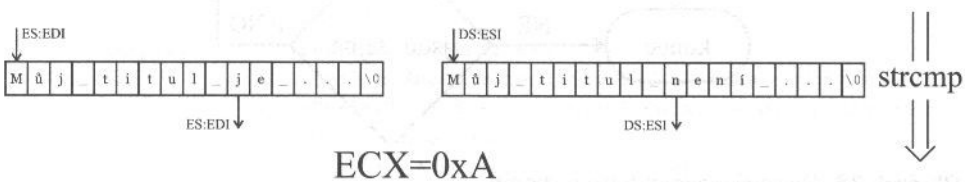
```
strcmp:
push  edx           ;ulož hodnotu registru EDX na zásobník
push  edi           ;ulož hodnotu registru EDI na zásobník
call  strlen        ;zavolej strlen na první řetězec
mov   edx,ecx       ;délku prvního řetězce ulož v EDX
mov   edi,esi       ;připrav parametr (registr) pro strlen do registru
                    ;EDI
    ;push ds        ;ulož na zásobník segmentový registr DS
    ;push ds        ;ještě jednou
    ;pop es         ; a ulož ji do registru ES (ES = DS)
                    ;
call  strlen        ;zavolej strlen na druhý řetězec

    ;pop ds         ;obnov původní DS
```

```
cmp    ecx,edx          ;porovnej délky obou řetězců
jae    .length_ok       ;skoč, je-li délka prvního (ECX) větší nebo rovna
                        ;délce druhého
mov    ecx,edx          ;budeme uvažovat delší řetězec (druhý)
.length_ok:
pop    edi              ;obnov hodnotu registru EDI
cld                     ;smaž vlajku směru (čítání nahoru)
repe   cmpsb            ;porovnávej, dokud jsou stejné a ecx je rozdílné
                        ;od nuly
pop    edx              ;obnov původní hodnotu registru EDX
ret                     ;návrat z podprogramu.
```

Funkce strcmp bude fungovat dobře pouze v případě rovnosti segmentových registrů. Správnější by bylo použít i zakomentované instrukce (instrukce zapsané v komentáři), tak aby řetězce mohly ležet i v různých segmentech. Vzhledem k tomu, že ve většině programů bude hodnota segmentových registrů rovna, nemusíme jejich změnu vůbec uvažovat.

Ukažme si na obrázku nastavení jednotlivých registrů, budou-li vstupem dva částečně shodné řetězce.



Obrázek 27 Nastavení registrů před a po volání funkce strcmp

6.5 Instrukce pro vstup a výstup (I/O)

Periferní zařízení komunikují s mikroprocesorem (s programem) pomocí vstupně výstupních bran, častěji nazvaných porty (I/O ports). Potřebujeme-li komunikovat s periferním zařízením na nejnižší (hardwarové) úrovni, použijeme k tomu instrukce IN a OUT.

6.5.1 Instrukce IN a OUT

Syntaxe:	***IN al, dx***	***OUT dx, al***
	IN ax, dx	***OUT dx, ax***
	IN eax, dx	***OUT dx, eax***
	IN al, imm8	***OUT imm8, al***
	IN ax, imm8	***OUT imm8, al***

Instrukce IN čte vstupně-výstupní port, jehož číslo se očekává v registru DX do registru AL. Do registru AX nebo EAX se uloží výsledek. Nejčastěji se používá osmibitová varianta čtení IN al, dx. Jiné registry než AL, AX, EAX a DX nejsou povoleny.

Data do zařízení můžeme poslat instrukcí OUT, pro kterou platí stejné omezení šířky operandů jako u instrukce IN. Pozor! Instrukce OUT má prohozen smysl operandů.

Ukažme si, jaké rozsahy portů mají jednotlivé běžné periferie přiděleny:

0000-001f : dma1 – první řadič DMA (přímý přístup do paměti)
0020-003f : pic1 – první řadič hardwarových přerušení
0040-005f : timer – systémový časovač
0060-006f : keyboard – klávesnice
0070-007f : rtc – hodiny
0080-008f : dma page reg – stránkový registr řadiče DMA
00a0-00bf : pic2 – druhý řadič hardwarových přerušení
00c0-00df : dma2 – druhý řadič DMA (přímý přístup do paměti)
00f0-00ff : fpu – tudy komunikuje koprocesor s hlavním procesorem
0170-0177 : ide1 – druhý řadič pevného disku (IDE)
01f0-01f7 : ide0 – první řadič pevného disku (IDE)
0213-0213 : isapnp read – rozhraní ISA PnP – plug-and-play
0220-022f : soundblaster – zvuková karta
0290-0297 : w83781d – hardwarový monitoring teploty a napětí zdroje
0376-0376 : ide1 – druhý řadič pevného disku (IDE) (pokračování)
03c0-03df : vga+ – grafická karta
03f2-03f5 : floppy – disketová mechanika
03f6-03f6 : ide0 – první řadič pevného disku (IDE) (pokračování)
03f7-03f7 : floppy DIR – disketová mechanika (pokračování)
03f8-03ff : lirc_serial – sériový port
0a79-0a79 : isapnp write – rozhraní ISA PnP – plug-and-play (pokračování)
0cf8-0cff : PCI conf1 – první konfigurační registr sběrnice PCI
4000-403f : Intel Corp. 82371AB/EB/MB PIIX4 ACPI – chipset ACPI
5000-501f : Intel Corp. 82371AB/EB/MB PIIX4 ACPI – chipset ACPI
e000-e01f : Intel Corp. 82371AB/EB/MB PIIX4 USB – chipset USB
f000-f00f : Intel Corp. 82371AB/EB/MB PIIX4 IDE – chipset řadič disků

Popis jednotlivých funkcí a registrů periferních zařízení značně přesahuje rámec této publikace.

```
in al,0x60  ;přečteme hodnotu z portu 0x60 (jedná se o scancode
            ;poslední stisknuté klávesy
```

6.5.2 Instrukce NOP

Syntaxe: ***NOP***

Instrukce NOP neprovádí žádnou operaci. Její anglický název je No Operation, žádná operace.

```
nop          ;provede "žádnou" operaci
```

Jaké je využití této funkce? Při zápisu do portů periferních zařízení je třeba mezi jednotlivými zápisy/čteními počkat. Periferní zařízení musí mít čas požadavky zpracovat. K tomu se instrukce NOP nehodí, protože trvá příliš krátkou dobu (obvykle jeden takt procesoru). Ke zpoždění se proto používala instrukce nepodmíněného skoku. Instrukcí jmp short se provedl skok ihned za instrukci.

```
jmp short zpozdeni1

zpozdeni1:
```

V Assembleru jsme mohli požadavek skoku na následující instrukci uskutečnit i pomocí zápisu:

```
jmp short $+2        ;skok "o 2 byte dál"
```

Znak '$' říká překladači, ať použije adresu místa v programu, kde je znak dolar zapsán. Protože instrukce jmp short je velká jen dva byty, stačí přičíst dvojku a instrukce provede skok za sebe.

Novější procesory byly vybaveny obvody pro predikci skoku, a tak i tato metoda nemohla být použita, protože skok díky predikci netrval tolik taktů. Jako nejschůdnější metoda čekání na periferní zařízení se ukázala metoda zápisu na port „kam to nevadí". Na PC se k tomuto účelu hodí osmibitový diagnostický port s číslem 0x80. Zápisem na tento port (čekání na sběrnici) se procesor dostatečně „opozdí", což bylo žádáno.

```
out 0x80,al          ;na port 0x80 pošleme libovolnou hodnotu, a tak
                     ;vytvoříme dostatečnou časovou prodlevu.
```

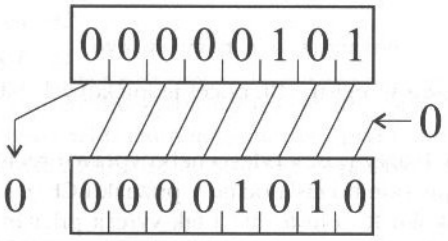
6.6 Instrukce posuvů a rotací

Bitový posuv použijeme pro změnu pořadí bitů v bajtu, slově nebo dvojslově tak, že jednotlivé bity jsou posouvány za sebou vlevo nebo vpravo. Bity, které při posuvu „vypadnou", nahradíme. Bitovou rotací provedeme stejnou změnu pořadí, s tím rozdílem, že bity, co „vypadnou", „nasuneme" ze strany druhé.

6.6.1 Instrukce SHR a SHL

Instrukcemi SHR a SHL můžeme provádět bitové posuny vlevo a v pravo celých čísel bez znaménka. Bitový posuv můžeme využít pro rychlé celočíselné násobení a dělení mocninou čísla dvě.

Představme si číslo 5 zapsané v binární číselné soustavě 0101b. Číslo pět vynásobené dvěma bude deset a v dvojkové soustavě bude zapsáno jako: 01010b. Jestliže porovnáme číslo deset a číslo pět, zjistíme, že číslo 10 vznikne tak, že číslo 5 posuneme o jeden bit doleva (a doplníme zprava jednou nulou – protože číslo dvě, kterým násobíme, je totéž jako dvě na první).



Obrázek 28 Násobení čísla pět dvěma jako binární posuv vlevo

Snadno můžeme násobit i jiná čísla jinými mocninami dvou. Budeme-li chtít např. vynásobit číslo šestnácti, pak jej posuneme o čtyři bity doleva (doplníme zprava čtyřmi nulami – protože číslo šestnáct, kterým násobíme, je totéž jako dvě na čtvrtou).

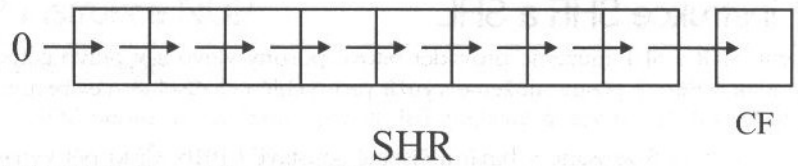
Dělení mocninami dvou je založeno na stejném principu. Místo bitového posunu vlevo se provádí bitový posun vpravo.

Syntaxe: ***SHL o1, o2***

Instrukce SHL (shift left) provádí bitový posun doleva. Instrukce SHL vyžaduje dva operandy. Prvním operandem (o1) může být registr nebo paměťové místo, kde budeme uskutečňovat bitový posun. Druhý operand (o2) specifikuje, o kolik bitů se bude posouvat, nejčastěji se jedná o přímý operand. Jediný z registrů, který můžeme použít jako druhý operand instrukcí posuvu a rotací, je registr CL. Počet posunů se bere modulo 32. Bit, který při operaci „vypadne" ven, se uloží do příznaku CF (carry) Nejnižší bit se nahradí novým nulovým bitem. Instrukce kromě příznaku CF nastaví i příznaky SF (znaménkový příznak) a OF (přetečení). Budeme-li pracovat s čísly se znaménkem, musíme dbát na změny příznaků SF a OF, tak abychom nedostali z kladného čísla násobením záporné. (Nastaví se SF a OF na 1.)

Syntaxe: ***SHR o1 ,o2***

Instrukce SHR provádí bitový posuv doprava a operandy jsou shodné s instrukcí SHL. Nulový bit se zleva doplní na místo bitu nejvyššího. Nejnižší bit, který vypadne, se přesune do carry (do příznaku CF).



Obrázek 29 Princip činnosti instrukce SHR

Uveďme si několik příkladů:

Příklad: Pomocí instrukce SHR vydělte hodnotu uloženou v registru AX šestnácti, přetečení ignorujte.

```
shr ax,4            ;posuneme o 4 bity vpravo
```

Příklad: Spočítejte a uložte v registru BL počet jedničkových bitů čísla uloženého v registru AX.

Šestnáctkrát provedeme bitový posuv (vlevo nebo vpravo) registru AX. Při každém posuvu se nám jeden bit původního čísla octne v příznaku CF (carry), kde jej můžeme instrukcí podmíněného skoku JC „otestovat" a tak vyřešit přičítání do registru BL.

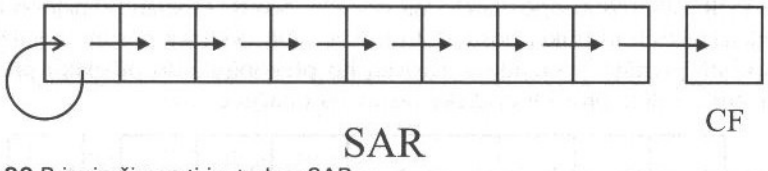
```
mov bl,0            ;můžeme zapsat i jako xor bl, bl
mov cx,16           ;do CX vložíme 16
opakuj:
shr ax,1            ;posuneme o 1 bit doprava, nejnižší bit se posune
                    ;do příznaku přenosu CF.
jnc neni_jedna      ;nebudeme přičítat, je-li bit nulový
inc bl              ;jinak do BL přičteme jedničku
neni_jedna:
loop opakuj         ;zopakujeme 16x
```

Po vykonání programu bude v registru BL počet jedničkových bitů v registru AX. Hodnota registru AX bude na konci nula.

6.6.2 Instrukce SAL a SAR

Syntaxe: ***SAL o1, o2***
SAR o1, o2

Chceme-li bitově posouvat čísla se znaménkem, užijeme tuto dvojici instrukcí. Instrukce pro bitový aritmetický posuv doleva SAL je synonymum instrukce SHL. Instrukce SAR zachovává nejvyšší (znaménkový) bit, nejnižší bit uloží do carry (příznaku přenosu CF). Operandy má shodné s instrukcí SAL (nebo SHR).

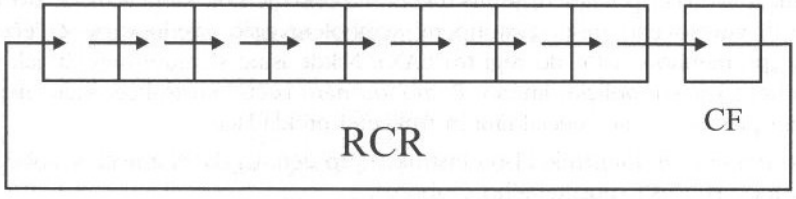


Obrázek 30 Princip činnosti instrukce SAR

6.6.3 Instrukce RCR/RCL

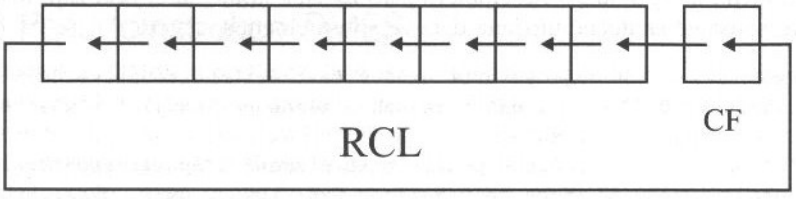
Syntaxe: ***RCR o1, o2***
RCL o1, o2

Instrukce RCL a RCR provádí cyklický bitový posuv. Instrukce RCR provede bitový posuv podobně jako instrukce SHR. Do nejvyššího bitu prvního operandu se neuloží nula, ale hodnota příznaku CF. Původní nejnižší bit se přesune do příznaku přenosu CF (carry).



Obrázek 31 Princip činnosti instrukce RCR

Instrukce RCL provádí stejnou akci, ovšem směr pohybu je opačný. Nejvyšší bit se přesune do příznaku CF (carry) a do nejnižšího bitu se uloží původní hodnota příznaku carry.

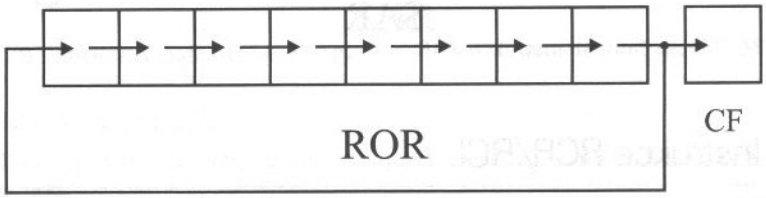


Obrázek 32 Princip činnosti instrukce RCL

6.6.4 Instrukce ROR a ROL

Syntaxe: ***ROR o1 ,o2***
ROL o1, o2

Instrukce ROR a ROL jsou opět variací na cyklický posuv. Operandy mají obdobné jako u předcházejících instrukcí posuvu. Instrukce ROR cyklicky posune nejnižší bit do nejvyššího bitu prvního operandu a zároveň ho překopíruje do příznaku přenosu CF (carry). Instrukce ROL provádí cyklický posuv na opačnou stranu.



Obrázek 33 Princip činnosti instrukce ROR

6.7 Pseudoinstrukce

Některé instrukce, s nimiž jsme se zatím seznámili, mohou mít svůj operand umístěn v paměti. Klasickým příkladem je instrukce MOV eax,[cislo], která nahraje do registru eax obsah adresového místa zastoupený symbolem cislo (zjednodušeně řečeno nahrajeme „proměnnou cislo" do registru EAX). Nikde jsme si zatím neřekli, jak přiřadit určité adrese její symbolické jméno. Pomohou nám pseudoinstrukce, které nejsou instrukcemi procesoru, ale speciálními instrukcemi překladače.

První skupinou pseudoinstrukcí jsou instrukce pro definici dat známých v době překladu (zůstanou součástí spustitelného souboru).

6.7.1 Pseudoinstrukce DB, DW a DD

Nejčastější pseudoinstrukcí je DB (define byte). Jejím použitím můžeme definovat číselné konstanty a řetězce. Uvedeme si několik příkladů.

Za středníkem, který slouží v Assembleru jako začátek komentáře, jsou zapsána v šestnáctkové soustavě skutečně uložená data ve spustitelném souboru.

```
db 0x55              ;jeden byte 0x55 v šestnáctkové soustavě 0x55
db 0x55,0x56,0x57    ;tři bajty za sebou těsně následující. 0x55, 0x56,
                     ;0x57
db 'a',0x55          ;můžeme použít také písmeno v apostrofech 0x61, 0x55
db 'Ahoj',13,10,'$'  ;nebo celý řetězec 0x41, 0x68, 0x6F, 0x6A, 0xD,
                     ;0xA,0x24
```

Cítíme-li potřebu definovat data po slovech (word), můžeme použít pseudoinstrukci DW (define word).

```
dw 0x1234            ;0x34, 0x12
dw 'a'               ;0x61, 0x00 druhý byte se vyplní číslem 0
```

Data lze definovat instrukcí DD (define dword) i po dvojslovech:

```
dd 0x12345678        ;0x78 0x56 0x34 0x12
dd 1.234567e20       ;dokonce můžeme definovat čísla v plovoucí
                     ;řádové čárce(jednoduchá přesnost)
```

A jak nadefinujeme „proměnnou" cislo?

```
cislo dd 0x1         ;v "proměnné" cislo bude počáteční hodnota 1
```

Symbol cislo nyní zastupuje adresu místa v paměti, kde bude hodnota 0x00000001 uložena.

6.7.2 Pseudoinstrukce RESB, RESW a RESD

Druhou skupinou jsou pseudoinstrukce pro definici neinicializovaných dat. Použijeme ji v případech, kdy nám nezáleží na počáteční hodnotě definované proměnné. Paměť přidělená pro neinicializovaná data se přidělí dynamicky až během startu programu, netvoří součást spustitelného souboru.

K dispozici jsou tři pseudoinstrukce definující paměť po třech základních úsecích. Po byte, slově a dvojslově – RESB, RESW, RESD (reserve byte, reserve word, reserve dword). Jejich argumentem je počet rezervovaných míst.

```
resb 1               ;rezervuj jeden byte
resb 2               ;rezervuj dva byte
resw 2               ;rezervuj 4 byte
resd 1               ;rezervuj 4 byte
cislo   resd 1       ;rezervuj proměnné cislo 4 byte
buffer  resb 64      ;rezervuj 64 byte pro buffer
```

V některých Assemblerech se můžeme setkat s konstrukcí DB ?, DW ? a DD ?. Assembler NASM, který budeme používat, nepodporuje takový typ konstrukce, místo toho budeme muset použít právě představené pseudoinstrukce RESB, RESW a RESD.

6.7.3 Pseudoinstrukce TIMES

Pseudoinstrukce TIMES je prefixová instrukce, která zopakuje následující instrukci nebo pseudoinstrukci. Má jediný parametr, který specifikuje, kolikrát se následující (pseudo) instrukce zopakuje. Svým chováním je podobná pseudoinstrukci DUP, kterou možná znáte z Assembleru TASM firmy Borland.

Můžeme ji použít všude tam, kde chceme vyplnit paměť nějakým opakujícím se vzorem. Chceme-li vytvořit „proměnnou", která bude obsahovat 64krát řetězec 'Ahoj', použijeme pseudoinstrukci TIMES takto:

```
moc_ahoj: times 64 db 'Ahoj'
```

Parametrem pseudoinstrukce TIMES může být i výraz. Představme si situaci, ve které potřebujeme do paměti zapsat určitý řetězec a zbytek, do 32 byte, vyplnit mezerami. Instrukcí times problém vyřešíme snadno:

```
buffer db "Ahoj"                    ;nadefinujeme řetězec
times 32-($-buffer) db ' '          ;vytvoříme odpovídající počet mezer
```

Hodnota výrazu 32-($-buffer) je podle očekávaní 28, výraz v závorce $-buffer se vyhodnotí jako 4, protože se odečte adresa aktuální pozice (operátor $) od adresy začátku řetězce. Počet mezer, které si přejeme doplnit, je rozdíl do 32, proto 32 - 4 = 28.

Instrukcí times se nemusíme omezovat jen na pseudoinstrukce, opakovat můžeme i „normální" instrukce procesoru.

```
times 5 inc eax     ;vytvoříme 5 instrukcí inc eax za sebou.
```

Výsledek se přepíše jako:

```
inc eax
inc eax
inc eax
inc eax
inc eax
```

6.7.4 Pseudoinstrukce INCBIN

Pseudoinstrukce INCBIN spíše poslouží zkušenějšímu vývojáři. Chceme-li mít ve výsledném spustitelném souboru uložena grafická nebo zvuková data, můžeme je pomocí pseudoinstrukce INCBIN přibalit.

```
incbin "zvuky.wav"          ;přibalíme celý soubor
incbin "zvuky.wav",512      ;přeskočíme prvních 512 byte
incbin "zvuky.wav",512,80   ;přeskočíme prvních 512 byte a neuložíme
                            ;posledních 80 byte.
```

6.7.5 Pseudoinstrukce EQU

Instrukce EQU slouží k definování symbolu určité konstantní hodnoty, která je známá v době překladu programu (konstantní hodnotou může být i hodnota složitějšího konstantního výrazu). Před pseudoinstrukcí EQU musíme vždy uvést jméno symbolu.

```
ctyrka EQU 4        ;poněkud hloupý příklad, ale uvidíme později, že
                    ;existuje daleko lepší.
```

6.7.6 Operátor SEG

Pokud budeme chtít vytvářet velké programy pro reálný režim procesoru, neobejdeme se bez většího počtu programových a datových segmentů, tak abychom obešli problém šestnáctibitové adresace. Prozatím předpokládejme, že segment je určitá součást adresy „proměnné".

Segmentový registr naplníme adresou segmentu ve kterém je identifikátor (symbol) fyzicky umístěn pomocí operátoru SEG.

```
mov ax,seg pocitadlo        ;v registru AX je adresa segmentu, ve kterém
                            ;je umístěn symbol (proměnná) pocitadlo
mov es,ax                   ;adresu segmentového registru musíme vždy
                            ;nahrát nepřímo
mov bx,pocitadlo            ;nahrajeme adresu (offset) do registru BX
                            ;párový registr ES:BX je naplněn kompletní
                            ;adresou "proměnné" pocitadlo
mov cx,es:[bx]              ;a přečteme z adresy do registru CX
```

V drtivé většině našich programů budeme moci segmentové registry ignorovat (i příkaz SEG), protože již budou obsahovat správnou hodnotu.

6.8 Triky s instrukcemi

Společně jsme tedy poznali všechny běžně používané instrukce. Nyní nastává vhodná doba k tomu, abychom si prozradili něco o optimalizaci programového kódu, který je těmito instrukcemi tvořen.

V dnešní době, kdy jsou počítače dostatečně levné a rychlé, už nikdo příliš nedbá na spotřebu paměti programem a ani na jeho rychlost. Programovací jazyk Assembler se ale často používá tam, kde máme k dispozici jen minimální paměť a kde musíme (ať chceme nebo ne) brát v úvahu i rychlost procesoru. Program zapsaný pomocí jiných vhodnějších instrukcí můžeme podstatně zmenšit, zrychlit a snadno znepřehlednit. Zrychlení programu docílíme většinou na úkor jeho velikosti a naopak. Podstatně lze program urychlit i vhodným umístěním do paměti počítače.

6.8.1 Příkaz ALIGN

Už víme, že procesor pracuje mnohonásobně rychleji se svými registry než s operandy umístěnými v paměti. Ve svých programech se proto musíme snažit veškeré výpočty provádět v registrech procesoru a do paměti ukládat až výsledky. Při práci s velkými poli se neobejdeme bez častých přístupů do paměti. Rychlost zpracování programu (přístupu do paměti) zvýšíme tak, že data umístíme na adresy dělitelné mocninami čísla dvě (na sudé, dělitelné 4, 16 atd.).

Mezi pamětí počítače a procesorem leží systém vyrovnávacích pamětí, který je z programátorského hlediska zcela skrytý. Ve vyrovnávacích pamětech jsou uloženy bloky nejčastěji používaných dat z hlavní paměti nebo dat dosud nezapsaných. Zarovnáním dat v programu na určité adresy se vyrovnávacím pamětem „uleví" (požadovaná data jsou součástí jednotlivých bloků) a data mohou proudit z procesoru a do procesoru mnohem rychleji.

Bohužel každý typ procesoru upřednostňuje jiné zarovnání, překladači sdělíme skutečnost, že chceme zarovnávat příkazem ALIGN. Jeho parametr je číslo vyjadřující násobky adresy, na které chceme zarovnávat:

```
align 4     ;zarovnej na adresy, které jsou násobky 4
align 16    ;zarovnej na adresy, které jsou násobky 16 (zarovnali jsme
            ;i na 2 a 4)
```

6.8.2 Naplňování registrů určitými hodnotami

Další přenosy dat z paměti do procesoru můžeme omezit tak, že nebudeme používat příliš často přímé operandy. Vynulovat registr EAX můžeme použitím logické funkce XOR, která pro stejné argumenty generuje nulový výsledek.

```
xor eax,eax          ;ve strojovém kódu 0x33,0xC0
```

Vynuluje registr EAX a bude podstatně rychlejší než:

```
mov eax,0            ;ve strojovém kódu: 0xB8,0,0,0,0
```

Instrukce XOR jako svůj „vedlejší" efekt změní registr příznaků procesoru. Použijeme ji všude tam, kde nebude změna příznaků nijak vadit.

Další, často používanou dvojicí jsou instrukce:

```
xor eax,eax          ;vynuluje EAX
inc eax              ;zvýší o 1
```

které zapíší do registru EAX jedničku. Záměnou instrukce INC za instrukci DEC zapíšeme do registru EAX -1.

6.8.3 Optimalizovaná aritmetika

Často je výhodnější (rychlejší) použít více instrukcí INC k přičtení určité konstantní hodnoty než použití jedné instrukce ADD.

Proto pišme místo:

```
add eax,4            ;k EAX přičti 4
```

Čtyři instrukce INC:

```
inc eax              ;k EAX přičti 1
inc eax
inc eax
inc eax
```

Instrukce INC ani DEC nenastavují příznak přenosu, a proto bychom se jich při aritmetických výpočtech uvažujících přenos měli vyvarovat (64bitová aritmetika). Počítáme-li novou hodnotu ukazatele, můžeme bez obav instrukce použít, při aritmetice s ukazateli se přenos nikdy neuplatňuje.

Instrukce bitových posunů vlevo a vpravo jsme již použili pro rychlé násobení a dělení. Násobit však můžeme i instrukcí LEA, která vypočítává efektivní adresu svého druhého operandu. Vzpomeňme si na široké možnosti adresace na procesorech 80386. Násobení instrukcí LEA můžeme zkombinovat i se sčítáním nebo odčítáním (všechny operace neuvažují přenos). Ukažme si několik příkladů:

```
lea ebx,[ecx+edx*4+0x500]  ;do EBX ulož výsledek výrazu
                           ;ECX + EDX*4 + 0x500
lea ebx,[eax+eax*4-1]      ;EBX = EAX*5 - 1
lea ebx,[eax+eax*8]        ;EBX = EAX*9
lea ecx,[eax+ebx]          ;provede se ECX = EAX + EBX
```

6.8.4 Srovnáváme

Velice často potřebujeme zjistit, je-li hodnota nějakého registru nula. Nemusíme použít instrukci CMP, ale instrukci OR (nebo TEST), takže místo

```
cmp eax,0          ;je EAX nula?
jz je_nula         ;ano? pak skoč na je_nula
```

zapíšeme:

```
or eax,eax         ;log. funkci OR nastaví stejný příznak, je-li
                   ;výsledek nula
jz je_nula         ;ano? pak skoč na je_nula
```

Instrukce OR zachovává hodnotu prvního operandu, je-li hodnota druhého operandu totožná. Navíc nastavuje příznakový registr procesoru podle výsledku. Je-li výsledek operace OR nula, nastaví se příznak ZF na 1.

Samotné aritmetické instrukce nastavují příznak nuly, proto nemusíme instrukcemi CMP provádět další test na nulu. Rozhodneme se jednoduše podle nastavených příznaků po aritmetické operaci.

```
dec eax
jz uz_je_nula      ;skočí, jestliže je eax po odečtení nula
```

Někdy potřebujeme zjistit, jestli je číslo nahrané v registru záporné.

Využít můžeme instrukci TEST, která provede logický součin operandů (výsledek se nikam neukládá, jen se nastaví příznaky, jak už víme). Příznak SF se nastaví vždy, je-li výsledek operace záporné číslo, které poznáme podle nejvyššího bitu (bude 1). Instrukci TEST vyvoláme se stejnými operandy, bude-li uložené záporné číslo, bude výsledkem nastavený příznak SF (provedli jsme bitový součin stejných záporných čísel, které mají nejvyšší bit 1, a protože 1 AND 1 = 1, pak se jako výsledek nastaví SF na 1).

```
test eax,eax       ;zavoláme instrukci TEST se stejnými operandy
js je_zaporne      ;můžeme se řídit příznakem SF
```

6.8.5 Ostatní

Často je snazší použít složitější instrukci než několik jednodušších. Složitější instrukce jsou optimalizované a stejnou „práci" vykonají rychleji než několik jednodušších instrukcí. Ke „složitějším" instrukcím můžeme zařadit skupinu instrukcí LOOP a instrukce pro práci s řetězci.

Velikost spustitelného souboru můžeme ještě snížit použitím vhodných instrukcí skoku. Tam, kde leží cíl v rozmezí 128 byte, můžeme místo instrukcí jmp a jmp near použít instrukci jmp short, která bude v paměti zabírat jen dva byte místo tří nebo pěti (podle režimu procesoru, pro který je program určen).

UŽITEČNÉ KOUSKY KÓDU

7

V této kapitole:

7.1 Jednoduché příklady
7.2 Převody čísel na řetězec
7.3 Převody řetězců na čísla

V této kapitole se seznámíme s rutinami, které jsou součástí drtivé většiny všech programů napsaných v čistém Assembleru. Budeme pracovat pouze s kódem, který není závislý na výběru operačního systému. S programy využívajícími operační systém se seznámíme v dalších kapitolách.

Ve všech příkladech budeme předpokládat správné nastavení segmentových registrů procesoru (ES, DS, SS). Blíže se seznámíme s touto problematikou v kapitole věnované operačnímu systému DOS.

7.1 Jednoduché příklady

Začneme s nejjednoduššími příklady, které nám pomohou překonat propast mezi vyššími programovacími jazyky a jazykem symbolických adres, Assemblerem.

7.1.1 Součet dvou čísel uložených proměnných

Příklad: Sečtěte dvě 32bitová čísla uložená v proměnných cislo1 a cislo2, výsledek uložte do proměnné výsledek (také dvaatřicetibitová proměnná).

Řešení příkladu musíme rozložit do tří částí. V první naplníme vybrané registry požadovanými hodnotami, ve druhé je sečteme, a ve třetí části zapíšeme výsledek. Výběr registrů řídíme velikostí použitých dat.

```
mov eax,[cislo1]    ;pamatujete? hranaté závorky značí přístup do paměti
mov ebx,[cislo2]    ;do registru EBX načteme obsah druhé "proměnné"
add eax,ebx         ;EAX = EAX + EBX
mov [vysledek],eax  ;výsledek uložíme do "proměnné" výsledek
...
cislo1 dd 08        ;tady jsme definovali "proměnnou" cislo1 a zároveň
                    ;jsme do ní vložili číslo 8
cislo2 dd 2         ;do proměnné "cislo2" jsme vložili číslo 2
vysledek dd 0       ;výsledek bude 8 + 2 = 010
```

Můžeme ušetřit registr EBX, pokud program přepíšeme takto:

```
mov eax,[cislo1]    ;nahraj hodnotu "proměnné" cislo1 do registru EAX
add eax,[cislo2]    ;sečti registr EAX a "proměnné" cislo2
mov [vysledek],eax  ;výsledek ulož na adresu určenou symbolem vysledek
...
cislo1 dd 08        ;tady jsme definovali "proměnnou" cislo1 a zároveň
                    ;jsme do ní vložili číslo 8
cislo2 dd 2         ;do proměnné "cislo2" jsme vložili číslo 2
vysledek dd 0       ;výsledek bude 8 + 2 = 010
```

Co kdyby zadání znělo takto?

7.1.2 Součet čísel uložených v registrech

Příklad: Sečtěte dvě 32bitová čísla, ukazatel na první sčítanec je uložený v registru EDI, druhý sčítanec je uložen těsně za prvním. Výsledek zapište do registru EDX.

V podstatě se jedná o dvouprvkové pole dvaatřicetibitových čísel, ukazatel na ně je uložen v registru EDI. Dvaatřicetibitová položka pole zabírá v paměti 4 byte, další bude umístěna na adrese o 4 byte vyšší.

```
mov edx,[edi]       ;první prvek pole ulož do registru EDX
add edx,[edi+4]     ;sečti registr EDX s druhým prvkem pole
```

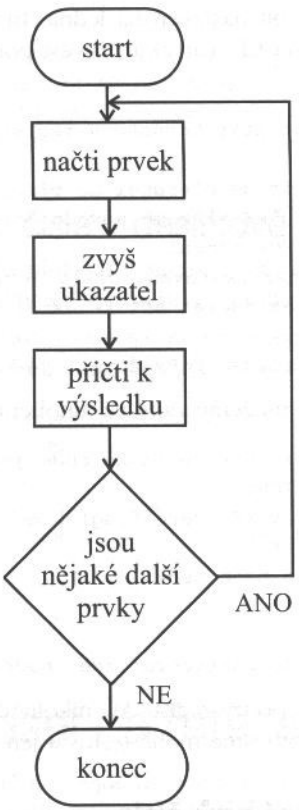
Kompletní program i s naplněním registru EDI:

```
mov edi,cisla       ;do edi jsme uložili adresu pole cisla
...                 ;mezitím můžeme provést i jiné výpočty
mov edx,[edi]       ;nahrajeme do registru EDX první prvek pole
add edx,[edi+4]     ;a sečteme s druhým prvkem pole
...
cisla dd 1          ;incializovali jsme čísla na jedničku a dvojku, takže
                    ;výsledek v registru EDX bude 3
      dd 2          ;druhý prvek pole se nemusí nijak jmenovat
```

Zatím jsme nepotřebovali implementovat rozhodování v programu, ukažme si to v následujícím příkladu.

7.1.3 Součet pole čísel

Příklad: Sečtete pole osmibitových čísel, na které ukazuje registr ESI (ukazatel je nahrán do registru ESI). Pole je ukončeno nulovým bytem.



Obrázek 34 Vývojový diagram součtu prvků pole

Výsledek budeme ukládat do dvaatřicetibitového registru, hodnoty budeme sčítat tak dlouho, dokud nenarazíme na nulový byte.

```
mov esi,pole            ;do ESI nahrajeme adresu začátku pole
mov ebx,0               ;EBX = 0
mov eax,ebx             ;EAX = 0
znovu:
mov al,[esi]            ;prvek z pole nahrajeme do registru AL
inc esi                 ;nastavíme ukazatel na další prvek
add ebx,eax             ;EBX = EBX + EAX
cmp al,0                ;je AL nula?
```

```
jnz znovu                ;skoč na znovu, pokud registr AL není nula
pole db 1,2,3,4,5,6,7,8,0  ;testovací hodnoty, součet v EBX bude 36
```

Instrukcemi posuvů můžeme snadno zjistit, je-li číslo sudé nebo liché.

7.1.4 Určení, zda je číslo sudé nebo liché

Příklad: Určete, je-li v registru AX uloženo číslo sudé nebo liché!

Každé liché číslo má nejnižší bit nastaven na jedna. Instrukcí SHR můžeme tento bit přesunout do příznaku přenosu CF (carry) a provést podmíněný skok podle příznaku přenosu instrukcí JC.

```
push ax      ;nechceme ztratit hodnotu v registru AX, uložíme ji na
             ;zásobník
shr ax,1     ;nejnižší bit se přesunul do příznaku CF
pop ax       ;do registru AX vrátíme původní hodnotu, registr příznaků
             ;instrukce POP nemění.
jc liche     ;je-li příznak nastaven, číslo je liché
sude:        ;zde může následovat nějaká akce, je-li sudé
...
liche:       ;zde bude program pokračovat, je-li v AX číslo liché.
```

Jak už bývá zvykem, program můžeme zapsat i mnohem jednodušeji:

```
test al,1    ;nejnižší bit bitové masky bude jedna, provedeme
             ;instrukci TEST
jz je_sude   ;příznak nuly ZF (zero flag) bude nastaven, bude-li
             ;nejnižší bit
             ;nula, číslo bude sudé
liche:
...
sude:        ;program bude pokračovat zde, bude-li číslo v AX sudé.
```

Všimněme si, že jsme testovali pouze registr AL, nikoliv celý AX. Vyšší bity registru AX jsou pro nás nezajímavé, a proto jsme mohli testovat jen registr AL.

7.1.5 Obrácení pořadí bitů čísla

Příklad: Převraťe pořadí bitů čísla, které je uloženo v registru AL, tak aby se nejnižší bit stal nevyšším bitem výsledku, druhý bit zprava druhým bitem zleva atd. Výsledek uložte do registru AH.

Představme si například číslo 0x15, které lze binárně zapsat jako: 00010101b Výsledkem má být číslo „zrcadlově" obrácené 1000 1010b čili 0x8A. Budeme potřebovat zařídit přeskupení bitů tak, aby se nejnižší bit registru AL stal nejvyšším bitem registru AH, druhý bit zleva registru AL druhým bitem regitru AH zprava atd.

Vyřešit příklad lze jako obvykle několika způsoby.

Mohli bychom jednotlivé bity registru AL testovat na 0 a 1, tak jak jsme si ukázali v kapitole věnované bitovým polím, a podle výsledku zase nastavovat bity v registru AH.

Lepší řešení nabízí instrukce posuvů a rotací. Například instrukcí SHR z předchozího příkladu jsme schopni vždy jeden bit přesunout do příznaku přenosu CF (carry), instrukcí RCL zase přesuneme bit z příznaku přenosu do nejnižšího bitu operandu a zároveň docílíme posunu ostatních. Pokud zopakujeme obě instrukce osmkrát, příklad úspěšně vyřešíme.

```
mov cx,8     ;do registru CX vložíme požadovaný počet opakování
smycka:
shr al,1     ;posuneme bity v registru AL o 1 doprava, nejnižší bit bude
             ;v příznaku přenosu CF
rcl ah,1     ;bity v registru AH posuneme o 1 doleva, nejnižší bit
             ;nahradíme bitem uloženým v příznaku přenosu
loop smycka ;zopakujeme 8x
```

7.1.6 Zjištění cifry v čísle v desítkové soustavě

Příklad: Zjistěte, zda má číslo uložené v registru AX nulu v nejnižším řádu desítkové soustavy. Předejte řízení na návěští ano nebo ne podle výsledku.

Prostým porovnáním bitů v čísle výsledku nedosáhneme, budeme muset číslo vydělit deseti (0xA). Pokud vyjde zbytek nula, skočíme na návěští ano.

```
mov bl,0xA  ;do BL nahrajeme konstantu 10
div bl      ;AX vydělíme BL (deseti)
cmp ah,0    ;je zbytek nula?
jz ano      ;pokud ano, skočíme na ano
ne:         ;zde pokračujeme, pokud není nula
...
ano:
...
```

7.2 Převody čísel na řetězec

V našich programech se určitě neobejdeme bez vypisování proměnných na obrazovku, proto bychom se měli naučit, jak číslo převést na řetězec.

Postupovat budeme stejně jako při převádění čísla z šestnáctkové soustavy. Číslo budeme opakovaně dělit deseti a zbytky sepisovat (to budou jednotlivé číslice, které převedeme na znaky – tedy výsledek). Celý proces budeme opakovat, dokud nebude výsledek z předchozího dělení nula. Ve vyšším programovacím jazyku to bude vypadat nějak takhle:

```
#include <unistd.h>

void main(void) {
unsigned int cislo;
char zbytek;
```

```
cislo=12345678;

while (cislo != 0)
 {
 zbytek = (cislo % 10)+'0'; /* zbytek = cislo mod 10 + char('0') */
 cislo /=10;  /* cislo = cislo div 10*/
 printf("%c",zbytek);
 }
}
```



Obrázek 35 Vývojový diagram převodu čísla na řetězec

Číslo, které převádíme, je 12345678, ovšem výsledek na obrazovce říká 87654321; to proto, že postupným dělením dostáváme nejprve nižší řády a nakonec ten nejvyšší. Jak otočit jednotlivé znaky – ASCII hodnoty (které vznikly po přičtení znaku '0' k proměnné zbytek) – tak aby byly ve správném pořadí? Jelikož pracujeme v Assembleru, máme k dispozici zásobník procesoru. Během převádění můžeme jednotlivé zbytky ukládat na zásobník a nakonec je vybrat, což nám zajistí změnu pořadí a tak i správný výsledek.

Celou konverzní funkci navrhneme jako podprogram. První věc, kterou musíme vyřešit, je předávání parametrů podprogramu. Už víme, že se parametry podprogramům předávají ve vyšších programovacích jazycích přes zásobník. My použijeme rychlejší variantu a předáme parametry ve smluvených registrech.

V registru EAX předáme číslo, které chceme převést na řetězec, a v registru EDI zase adresu (ukazatel) volného místa v paměti, kam budeme ukládat výsledný řetězec ukončený nulovým byte (jako řetězec v jazyce C).

Celý podprogram preved složíme ze dvou cyklů. V prvním, který odpovídá while cyklu z našeho příkladu, budeme postupně ukládat na zásobník jednotlivé cifry čísla převedené na znaky. V druhém cyklu je postupně vybereme ze zásobníku a zapíšeme do připraveného místa.

Funkce preved by mohla vypadat takto:

```
preved:

mov ecx,0           ;ECX = 0
mov ebx,010         ;EBX = 010

.vydel:

mov edx,0           ;EDX = 0
div ebx             ;EDX:EAX / EBX , výsledek je v EAX,
                    ;zbytek v EDX
add edx,'0'         ;ke zbytku přičteme hodnotu nuly v ASCII
push edx            ;uložíme na zásobník
inc ecx             ;poznamenáme si, kolik jich je na zásobníku
cmp eax,0           ;je už zpracováno celé číslo? (EAX? 0)
jnz .vydel          ;jestliže není nula, tak skoč na .vydel
                    ;převedli jsme celé číslo, jednotlivé znaky máme
                    ;uloženy na zásobníku, v ECX je jejich počet
.prohod:

pop eax             ;vyber znak ze zásobníku
mov [edi],al        ;zapiš do řetězce
add edi,1           ;nastav ukazatel na řetězec na další znak
dec ecx             ;
cmp ecx,0           ;je už ECX nula?
jnz .prohod         ;Ne? prohodíme další
ret                 ;Ano? návrat z podprogramu
```

Navrhovaný program obsahuje záměrně několik chybiček. Na konci minulé kapitoly jsme si řekli něco o optimalizaci, proveďme ji i zde.

Hned na začátku můžeme poněkud těžkopádné mov ecx,0 vyměnit za elegantní xor ecx,ecx. Stejně tak naplnění registru ebx konstantou 10 lze udělat tak, že celý registr nejprve vynulujeme a do registru BL vložíme konstantu 10. Vyhneme se tak zbytečně velkému přímému operandu. Vyměnit můžeme i test nuly v registru EAX za or eax,eax (nebo za test eax,eax). Dokonce můžeme změnit celý konec funkce tak, že zaměníme následující řádky:

```
mov [edi],al
add edi,1
```

za instrukci

```
stosb
```

instrukce stosb vykonává totéž, ale zabírá v paměti jen jeden byte.

Zbytek funkce (testování ECX na nulu, odečtení a skok) můžeme zaměnit jedinou instrukcí LOOP. Celá přepsaná funkce bude vypadat takto:

```
preved:
xor ecx,ecx          ;ECX = 0
xor ebx,ebx          ;EBX = 0 místo xor můžeme použít mov ebx,ecx
mov bl,10            ;registr EBX = 010
.vydel:
xor edx,edx          ;EDX = 0
div ebx              ;EDX:EAX/EBX, výsledek je v EAX, zbytek v EDX
add dl,'0'           ;ke zbytku přičteme hodnotu nuly v ASCII
push edx             ;uložíme na zásobník
inc ecx              ;poznamenáme si, kolik jich je na zásobníku
or eax,eax           ;je už zpracováno celé číslo? (EAX? 0)
jnz .vydel           ;jestliže není nula, tak skoč na .vydel
                     ;převedli jsme celé číslo, jednotlivé znaky máme
                     ;uloženy na zásobníku, v ECX je jejich počet
.prohod:
pop eax              ;vyber znak ze zásobníku
stosb                ;zapíše hodnotu registru AL na místo, kam ukazuje
                     ;registr EDI, a zvýší jeho hodnotu o jedna
loop .prohod         ;odečti od ECX jedničku, skoč na prohod, dokud je ECX
                     ;rozdílné od 0
ret                  ;návrat z podprogramu
```

Program jsme jistě vylepšili, ale nepracuje úplně správně. Po převodu se nezapíše nulový byte za konec řetězce. Chybu můžeme napravit dodatečnou instrukcí

```
MOV byte [edi],0
```

kterou vložíme mezi instrukci LOOP a RET.

Slůvko byte je v instrukci MOV podstatné neboť určuje „velikost" zapisované nuly. Překladač nemůže určit, kolik bitů má nula v paměti zabírat. Řetězec ukončujeme nulou osmibitovou, proto použijeme klíčové slovo byte.

Funkce ničí původní obsah některých registrů (EAX, EBX, ECX, EDX a EDI). Chceme-li je zachránit, musíme je před změnou uložit na zásobník a na konci funkce zase ve správném pořadí vybrat.

Celou konverzní funkci budeme pak volat jako podprogram takto:

```
...
mov eax,012345678    ;samozřejmě pokud je převáděná hodnota jinde
                     ;nahrajeme ji do EAX jinak
mov edi,buff
call preved
...
```

Převáděné číslo před zavoláním umístíme do registru EAX. Druhým parametrem bude ukazatel na volné místo, kam se zapíše číslo převedené na řetězec. Ukazatel vložíme do registru EDI nebo DI, podle toho, pro jaký režim procesoru programujeme (předpokládáme, že jsou segmentové registry DS a ES správně nastaveny). Instrukcí CALL za-

voláme podprogram preved. Na připraveném volném místě bude uložen nově vytvořený řetězec.

Funkce preved zatím převádí čísla na řetězec, který reprezentuje číslo zapsané v desítkové soustavě. Jak bychom museli funkci preved modifikovat, aby čísla převáděla na řetězce zapsané v šestnáctkové soustavě?

Zbytky po dělení deseti dostaneme v registru DL. Abychom je mohli převést na znaky, musíme ke zbytku uloženému v registru DL přičíst ASCII hodnotu znaku '0'. V ASCII tabulce jsou ostatní číslice uloženy za sebou, proto dostaneme správný výsledek. Kdybychom ovšem dělili šestnácti, zbytky by vycházely v intervalu 0 až 15. Přičtením konstantní hodnoty znaku '0' bychom pro číslice vyšší než devět nedostali znaky A až F, jak potřebujeme, ale znaky, které v ASCII tabulce následují za devítkou, čili: ':;<=>?'.

Zavedením testu pro číslo vyšší než devět a přičtením jiné konstanty můžeme náš problém snadno vyřešit.

Upravíme náš program tak, že změníme konstantu dělení z deseti na šestnáct (nezapomeňme ale, že číslo šestnáct je v šestnáctkové soustavě 0x10). A místo instrukce add dl,'0' zavoláme další podprogram, který pro čísla 0–9 vrátí v registru DL ASCII hodnotu čísel 0 až 9. A pro čísla 10–15 vrátí znaky 'A' až 'F'.

```
HexDigit:           ;v registru DL očekáváme číslo 0-15, které převedeme
                    ;na šestnáctkovou číslici
cmp dl, 10          ;je DL deset?
jb .mensi           ;skočíme, když je menší
add dl,'A'-10       ;přičteme konstantní hodnotu, tak abychom získali
                    ;znaky 'A'-'F'
ret                 ;ukončíme podprogram
.mensi:
or dl,'0'           ;místo instrukce add dl,'0' lze v tomto případě
                    ;použít instrukci OR
ret                 ;ukončíme podprogram
```

Co kdybychom převod zobecnili? Stačila by výměna dělicí konstanty, protože podprogram pro převod číslic na jednotlivé znaky jsme již upravili. Nový podprogram pro převod čísel na řetězce bude očekávat na svém vstupu kromě převáděného čísla i základ soustavy, ze které číslo převádíme. Všimněme si, že stačí vypustit řádek, kde registr BL (EBX) naplňujeme konstantou 10, a prohlásit hodnotu uloženou v registru EBX za parametr podprogramu. Posledním parametrem bude adresa volného místa, kde se uloží výsledek převodu.

V předchozím příkladu jsme po návratu z podprogramu nezachovávali obsah registrů, doplňme ještě na začátek a na konec instrukce PUSHAD a POPAD, kterými uložíme obsah všech univerzálních registrů na a ze zásobníku.

Nově vytvořený podprogram budeme používat i v dalších příkladech.

```
;-------------------------------------------------------------------
; NumToASCII
;-------------------------------------------------------------------
;
```

```
; eax = 32 - bit číslo
; ebx = základ soustavy
; edi = pointer na buffer, kam se zapíše
;Vrací:
;naplněný buffer
;------------------------------------------------------------------------

NumToASCII:

pushad                  ;ulož všechny univerzální registry na zásobník
xor esi, esi            ;vynuluj registr ESI

smycka_prevodu:

xor edx, edx            ;EDX = 0
div ebx                 ;EDX:EAX / EBX = EAX
                        ;zbytek v EDX
call HexDigit           ;převedu do ASCII
push edx                ;uložím na zásobník
inc esi                 ;ESI = ESI + 1 kolik slov jsem uložil na zásobník
test eax, eax           ;EAX = 0?
jnz smycka_prevodu      ;ne? skočím a převádím dál
cld                     ;smažu vlajku směru, čítám vzhůru

smycka_zapis:

pop eax                 ;vyberu znak ze zásobníku
stosb                   ;uložím ho na adresu ES:(E)DI
dec esi                 ;odečtu registr ESI
test esi, esi           ;na zásobníku už nic není, tj. ESI není nula
jnz smycka_zapis        ;jestliže není, skočím a vyberu další

popad                   ;obnovím hodnoty registrů
ret                     ;ukončím podprogram
```

7.3 Převody řetězců na čísla

Často potřebujeme opačnou funkci, která převede číselný řetězec na číslo. Ve vyšších programovacích jazycích jsou k dispozici konverzní knihovní funkce (readln, scanf), zkusme si podobnou, mnohem jednodušší funkci naprogramovat.

Pro převod znaku na číslo zavedeme nový podprogram preved_znak tak, že znaky z intervalu '0' – '9' bude převádět na číselný interval 0 – 9 a znaky z intervalů 'a' – 'f' a 'A' – 'F' zase na interval 10 – 15 (0xA – 0xF). Jediným vstupně-výstupním parametrem bude registr AL, který před zavoláním naplníme jedním převáděným znakem (ASCII hodnotou). Výstupem podprogramu bude jeho číselná hodnota uložená ve stej-

ném registru. Podobně jako v minulém případě budeme muset vyřešit problém vstupu malých a velkých písmen abecedy. Ukažme si navrhovaný podprogram:

```
preved_znak:

sub al,'0'          ;od znaku odečteme hodnotu znaku nula
cmp al,010          ;test, jsme-li už v intervalu 0 - 9
jb jsme_0az9        ;ano? skočíme na ok
and al,05fh         ;převedeme na velká písmena
add al,9            ;přičteme 9, abychom dostali správný interval 10 - 15

jsme_0az9:

and al,0fh          ;nedovolíme převod ze soustavy vyšší než šestnáctková
                    ;vymažeme horní 4 bity registru AL
ret                 ;návrat
```

Prvním krokem podprogramu je odečtení ASCII hodnoty znaku '0' od převáděného znaku, leží-li výsledná hodnota v intervalu 0 – 9, potom provedeme skok na návěští jsme_0az9. Nyní máme jistotu, že vstupem bude malé nebo velké písmeno (zlé uživatele neuvažujeme). Díky dobrému návrhu ASCII tabulky stačí vynulovat pátý bit znaku (maska 0x5F) a tak změnit malé písmeno na velké. Změna bitu u velkého písmena nemá žádný vliv. Nakonec přičteme konstantu 9 a dostaneme tak výsledný interval čísel, kde znak 'A' bude převeden na číslo 0xA atd. Posledním logickým součinem omezíme interval převáděných čísel na 0x0 – 0xF (maska 0xF ponechá pouze spodní čtyři bity, ostatní vynuluje).

Zamysleme se nad implementací převodní funkce, která bude uvažovat čísla se znaménkem. Zjednodušme si zadání a předpokládejme, že pouze záporné číslo je uvozeno „těsně" přiléhajícím znakem '-'.

Stačí si uvědomit, že číslo převedeme úplně stejně jako kladné a nakonec ho znegujeme (provedeme dvojkový doplněk instrukcí NEG). Na začátku převodu přeskočíme pouze případné znaménko.

Zkusme si navrhnout algoritmus, který řetězec na číslo převede.

V kapitole věnované číselným soustavám jsme si ukázali, jak lze každé číslo zapsat jako součet součinů jednotlivých cifer čísla s odpovídající mocninou základu soustavy. Připomeňme si:

$a = a_n*z^n + a_{n-1}*z^{n-1} + \ldots + a_1*z^1 + a_0*z^0$ (n je počet číslic čísla)

Desítkové číslo 1 234 můžeme zapsat jako:

$1\,234 = 1*10^3 + 2*10^2 + 3*10^1 + 4*10^0$

Podobně šestnáctkové číslo 0x524D zapíšeme jako:

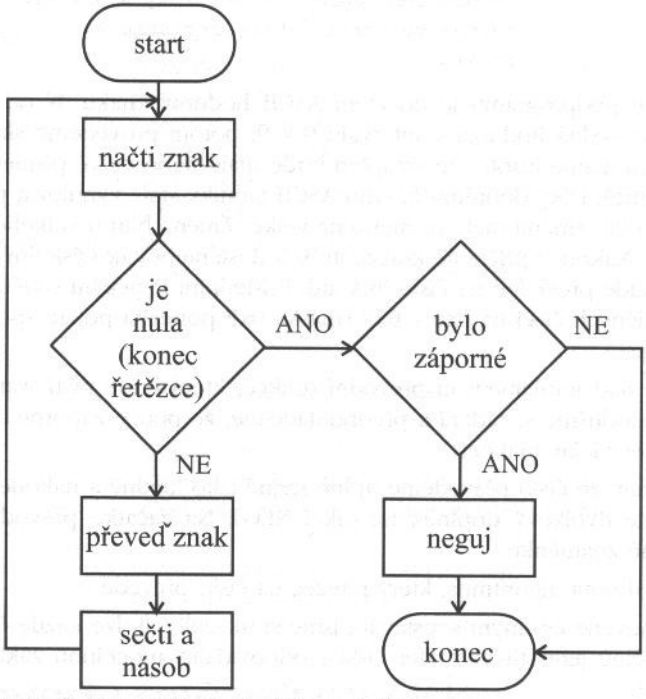
$(524D)_{16} = 5*16^3 + 2*16^2 + 4*16^1 + 13*16^0 = 21\,069.$

První zamyšlení by mohlo vést k domněnce, že stačí algoritmus implementovat tak, jak jsme napsali rozklad. Postupně převádět jednotlivé znaky na čísla funkcí preved_znak a výsledek vynásobit mocninou základu v příslušném řádu a nakonec provést součet. Navrhovaný postup má však jeden háček, číslo je zapsané od nejvyšší mocniny a my nevíme, kolik to bude. Nehezkým řešením by bylo spočítat počet cifer čísla a použít navrhovaný postup.

Ovšem číslo 1 234 lze rozepsat i takto:

1 234 = ((((1)*10 + 2)*10 + 3)*10) + 4

Pak stačí první převedenou cifru (zleva) vynásobit deseti, k výsledku přičíst druhou cifru a opět vynásobit deseti atd. Díky postupnému násobení nemusíme znát ani délku převáděného čísla.



Obrázek 36 Vývojový diagram převodu řetězce na číslo

Seznamme se s celým podprogramem, který převede řetězec na číslo:

```
;-------------------------------------------------------------------------
; ASCIIToNum
;-------------------------------------------------------------------------
; esi = ukazatel na řetězec, odkud se převádí, do té doby, než je
; nalezen znak ;0x0
; ecx = základ soustavy
; Vrací:
```

```
; eax = výsledek

ASCIIToNum:

push esi                    ;uložíme ukazatel na řetězec
xor eax,eax                 ;EAX = 0
xor ebx,ebx                 ;EBX = 0

cmp [esi], byte '-'         ;otestujeme, jestli se jedná o záporné číslo
jnz .dalsi                  ;pokud není, nenačítáme další cifru
inc esi                     ;připravíme ukazatel za znak '-'
.dalsi:

lodsb                       ;čteme číslici
or al,al                    ;otestujeme konec řetězce
jz .done
call preved_znak            ;převedeme ji na číslo, uložíme v registru AL
imul ebx,ecx                ;posuneme řád předchozímu číslu
add ebx,eax                 ;přičteme právě převedené
jmp short .dalsi            ;a zopakujeme

.done:

xchg ebx,eax                ;výsledek uložíme do EAX
pop esi                     ;obnovíme původní hodnotu v ESI
cmp [esi], byte '-'         ;podíváme se, jestli výsledkem má být
jz .done_neg                ;záporné číslo
ret                         ;nemá a skončíme
.done_neg:
neg eax                     ;ano, má, pak negujeme
ret                         ;skončíme rekurzivní volání
```

OPERAČNÍ SYSTÉM

8

V této kapitole:

8.1 Vývoj operačních systémů
8.2 Přidělování procesoru
8.3 Správa paměti
8.4 Souborové systémy
8.5 Zavedení systému na PC

Operační systém vytváří rozhraní mezi aplikačními programy a holým počítačem. Spravuje a rozděluje systémové prostředky jednotlivým programům (procesům).

8.1 Vývoj operačních systémů

Historie operačních systémů se začala psát v padesátých letech dvacátého století. První počítače vyžadovaly neustálou pozornost operátora, který vkládal ručně programy psané na děrných štítcích a ovládal tlačítka na řídicím pultu. Bylo však velmi drahé nechat počítač během vkládání úloh (jednoho nebo více programů) zahálet, a tak byly „vynalezeny" první druhy operačních systémů, které umožňovaly při skončení jedné úlohy začít automaticky provádět úlohu další. Část operačního systému, která řídila postupné zpracování úloh, se nazývala monitor a umožňovala tak poprvé prakticky realizovat automatizované dávkové zpracování.

V šedesátých letech se operační systém dále vyvíjí a jako specializovaná část vzniká program – jádro operačního systému. Důvod jeho vzniku byl poměrně prozaický. Počítače byly po hardwarové stránce složitější, a tak bylo i pro programátora složité psát programy, které budou obsluhovat periferie počítače. Jádro bylo natrvalo zavedeno v operační paměti počítače a programátorovi nabízelo rozhraní pro ovládání periferních zařízení. Postupně se na jádro „nabalovaly" i další služby. Z histo-

rického hlediska zajímavou službou bylo tzv. účtování (Accounting). Firma vlastnící počítač vždy zákazníkovi naúčtovala čas, který jeho úloha spotřebovala.

Snahy o efektivnější využití zdrojů počítače vrcholily kolem roku 1964. Vycházely z myšlenky, že není příliš efektivní provádět v systému pouze jednu úlohu, když by další úloha užívající jiné periferní zařízení mohla běžet současně. Tak se zrodilo multiprogramování.

Operačním systémům přineslo multiprogramování řadu nových úkolů: Jak jednotlivým programům přidělovat zdroje počítače – procesor, paměť a periferní zařazení? Odpovědi na otázky najdeme v následujících podkapitolách.

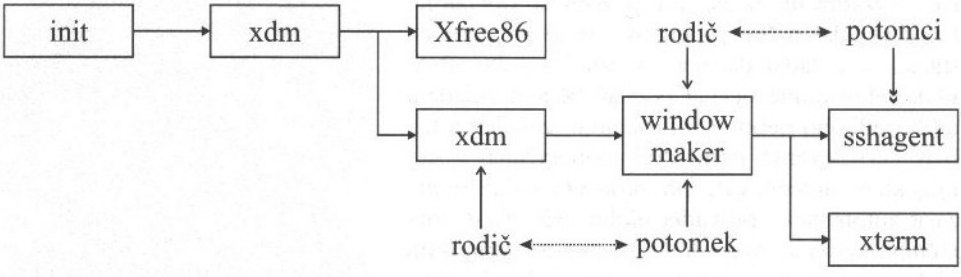
8.2 Přidělování procesoru

Přidělování procesoru znamená poskytnutí jeho výpočetní kapacity jednotlivým programům.

8.2.1 Procesy

Proces je zjednodušeně řečeno program, který byl zaveden do paměti a dostává přidělen procesor.

Každý proces, stejně tak jako lidé, má záznam o svém rodiči. Rodičovský proces může utvořit proces nový, označovaný jako potomek (child-proces). Procesy tvoří něco jako rodokmen, kterému se říká hierarchie procesů.



Obrázek 37 „Rodokmen" procesů

Jádro operačního systému po svém spuštění vytvoří první proces, který se v operačním systému UNIX nazývá init. Proces init je jakýmsi prapředkem všech procesů v systému.

V operačním systému DOS je takovým prapředkem program COMMAND.COM.

Potomek může od rodičovského procesu zdědit některé jeho vlastnosti nebo některá data. Srovnání s rodiči a dětmi poněkud pokulhává v případě smrti (ukončení) rodiče. Potomka buď adaptuje jiný proces (init), anebo dojde k jeho ukončení.

8.2.2 Princip plánování procesů

Procesy na počítačích neběží paralelně (každý proces by musel běžet na samostatném procesoru), ale tzv. pseudoparalelně. Trochu to připomíná přísloví „každý chvilku tahá pilku".

Jádro operačního systému spravuje ve svých interních strukturách tzv. frontu úloh (požadavků), ve které jsou zařazeny všechny spuštěné procesy. Část jádra zvaná plánovač úloh vybere podle určitého kritéria proces z fronty a ten „probudí". Probuzením se rozumí přidělení procesoru danému procesu po určitý časový interval a provádí jej část jádra zvaná dispečer.

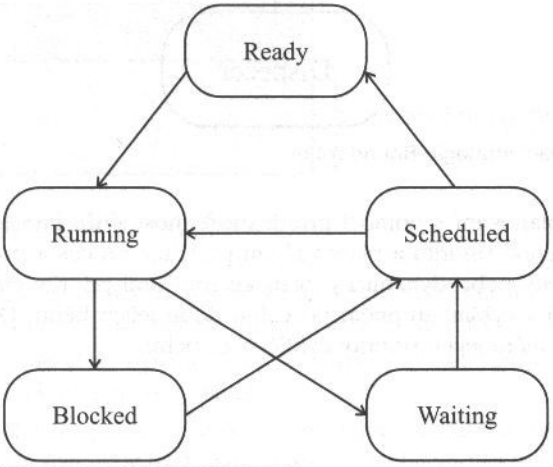
Dispečer musí procesu obnovit jeho kontext. Trochu to připomíná obsluhu přerušení, kdy jsme přerušenému programu museli uchovat všechny hodnoty registrů, tak aby mohl řádně pokračovat tam, odkud byl přerušen. Jakmile je kontext obnoven, jádro předá procesu řízení (přidělí procesor). Po určité době se běh právě prováděného procesu přeruší a řízení se předá zpět jádru. Pomocí plánovače úloh vybere další vhodný proces a dispečer jej spustí. Celý cyklus se opakuje.

8.2.3 Reprezentace a stavy procesu

Jádro operačního systému musí mít o každém procesu přehled. Struktura, kde se uchovávají informace o procesu, se nazývá proces control block (PCB).

V operačním systému UNIX často hovoříme o pěti možných stavech procesu:

- Running – proces má zrovna přidělen procesor
- Ready – proces je připraven, ale nemá přidělen procesor
- Blocked – proces nedostává přidělen procesor (byl úplně zastaven – zmrazen)
- Scheduled – proces je naplánován a bude za okamžik spuštěn
- Waiting – proces čeká na splnění nějaké podmínky, aby mu byl přidělen procesor, čeká například na uvolnění nějakého sdíleného prostředku



Obrázek 38 Stavy procesu a jejich přechody

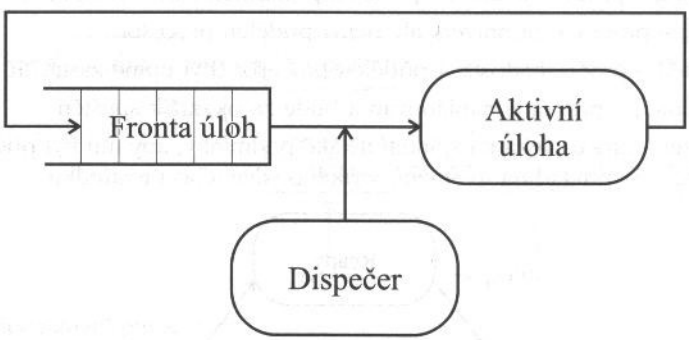
Procesy v operačním systému DOS nelze takto rozdělit, operační systém DOS je jednoúlohový, procesor je vždy přidělen pouze jednomu, právě spuštěnému programu.

Jádro operačního systému uchovává i další informace o procesu:

- přidělená paměť
- přidělení prostředků operačního systému (otevřené soubory, zařízení atd.)
- kontext procesu
- stav procesu
- jméno procesu
- identifikační číslo procesu PID
- identifikační číslo rodiče
- práva přístupu
- právě používaný adresář
- priorita

Ještě jsme si neřekli, podle čeho vybírá jádro operačního systému (plánovač) procesy vhodné k přidělení procesoru.

Nejjednodušší strategie přidělování procesoru se nazývá Round Robin. Procesy jsou uloženy ve frontě. Plánovač vybere první proces z fronty, dispečer jej spustí. Po uplynutí pevně stanovené doby se proces přestane provádět (uloží se na konec fronty.) Vybere se další úloha a celý cyklus se opakuje.



Obrázek 39 Plánovací strategie Round robin

Jinou strategií je plánování s pomocí priorit (upřednostnění). Procesy, které čekají ve frontě, mají přiřazenou prioritu a plánovač jim přiděluje procesor podle zvolené metody, a to buď staticky nebo dynamicky. Statické přidělení priority znamená přednostní provádění procesů s vyšším stupněm po celou dobu jejich běhu. Dynamické přidělování naproti tomu mění jejich priority cyklicky za běhu.

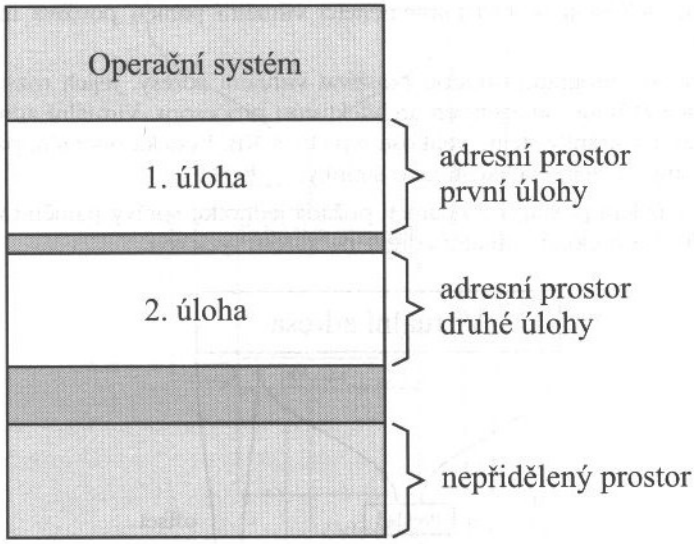
8.3 Správa paměti

Operační systém musí při správě operační paměti zajistit sledování volného místa, strategii přidělení, realizaci přidělení vhodnými mechanismy a vyřešit její uvolňování k dalšímu použití. Rozděluje při tom operační paměť mezi jádro a zpracovávané procesy.

8.3.1 Jednoduché přidělování paměti

Jednoduché přidělování paměti bylo nejstarším způsobem, jak rozdělit operační paměť mezi procesy. Existovaly různé metody, které ji rozdělovaly buď technickými nebo programovými prostředky na určitý počet pevných, později proměnných oblastí.

Paměť se rozdělila mezi jádro operačního systému a pevné úseky přidělované uživatelským procesům. Každý úsek paměti používala jen jedna úloha, který „okupovala", i když neměla zrovna přidělen procesor. Jakmile se úloha dokončila, byl na její místo natažen program monitor, který byl připraven přijmout další řídicí příkazy, kde posledním příkazem bylo opět spuštění úlohy. Jádro operačního systému zajišťovalo pro každý proces ochranu jednotlivých bloků paměti každého úseku.



Obrázek 40 Pevné přidělování paměti úlohám

Dalším vylepšením bylo přidělování úseků s proměnnou velikostí. Procesy na začátku svého běhu jádru sdělily, kolik paměti budou potřebovat, a jádro jim ji vyhradilo.

Oba druhy přidělování úseků paměti byly využívány s ohledem na hardwarové prostředky počítače (velikost operační paměti).

Další vývoj v přidělování paměti vycházel z poznatku, že úloha nemusí mít po celou dobu svého běhu přidělen celý úsek paměti. Pro realizaci této strategie byly využity velkokapacitní disky.

8.3.2 Swapování

Organizování procesů do pevně přidělených úseků paměti bylo výhodné při zpracovávání dat, která neměla interaktivní vstup. Programy požadující uživatelský vstup (z terminálu) zbytečně zahálely čekáním na uživatele a zabíraly paměť, kterou by mohla využít jiná úloha.

Pomocí metody zvané swapping (swapování, prohazování) bylo možné úlohu čekající na uživatelský vstup „odstranit" z operační paměti a nahrát na disk. Tak se uvolnila paměť a mohla se zavést úloha jiná. Když byla „odstraněná" úloha plánovačem vybrána, musela se znovu celá nahrát do paměti. Pokud nebyla žádná paměť volná, musela se z paměti „odstranit" jiná úloha.

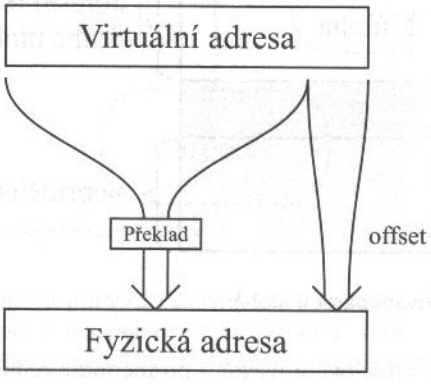
8.3.3 Virtuální paměť a stránkování

Murphyho zákon praví, že počítačový program se rozrůstá natolik, až zaplní celou operační paměť.

Řešením problému by bylo předstírat, že operační systém má k dispozici více operační paměti (tzn. více adres) než ve skutečnosti. A přesně na stejném principu pracuje virtuální paměť. Nejčastěji se pro implementaci virtuální paměti používá mechanismus stránkování.

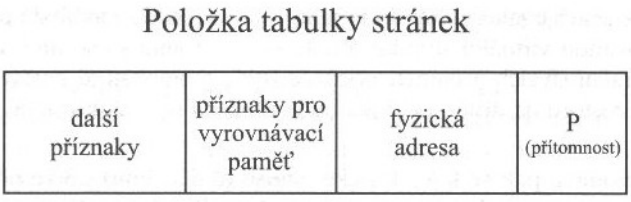
Všechny procesy (programy) budou používat virtuální adresy. Jejich rozsah (všechny dostupné adresy) bude omezen jen architekturou procesoru. Virtuální adresní prostor bude rozložen na stránky stejné velikosti typicky 4 KB. Fyzická operační paměť se rozdělí na tzv. rámy o stejné velikosti jako stránky.

Procesor při každém přístupu do paměti požádá jednotku správy paměti (Memory Management Unit) o překlad virtuální adresy na adresu fyzickou.



Obrázek 41 Překlad virtuální adresy na fyzickou

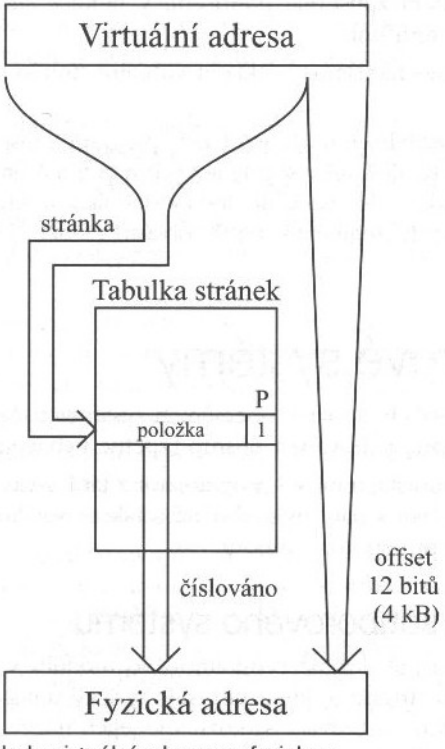
Překlad virtuálních adres na fyzické probíhá pomocí tabulky stránek, ve které je pro každou virtuální stránku vyhrazena jedna položka.



Obrázek 42 Položka tabulky stránek

Nejdůležitějším bitem je bit přítomnosti stránky. Je-li nastaven, pak je tato stránka uložena v rámu, jehož číslo (adresa) je uloženo v položce fyzická adresa. Ostatní bity tabulky specifikují práva přístupu ke stránce (čtení/zápis/spouštění), jsou použity pro potřeby vyrovnávací paměti. Indexem do tabulky stránek je číslo každé virtuální stránky.

Ukažme si příklad překladu nějaké virtuální adresy na fyzickou.



Obrázek 43 Princip překladu virtuální adresy na fyzickou

Jednotka správy paměti (MMU) z virtuální adresy zjistí číslo stránky, která je požadována. Rozdíl adres mezi začátkem adresy stránky a požadovanou adresou budeme nazývat offset.

Číslo stránky se použije jako index do tabulky stránek, kde se vyhledá příslušná položka pro požadovanou virtuální stránku. Podle bitu přítomnosti se určí, zdali je stránka uložena v operační (fyzické) paměti počítače (bit přítomnosti je nastaven na 1) nebo v odkládacím prostoru na disku. (stránka bude uložena ve stránkovacím prostoru na disku).

Je-li stránka přítomna, pak se k její fyzické adrese (číslo rámu) přičte offset. S takto získanou fyzickou adresou pak procesor pracuje. Není-li stránka přítomna v operační paměti, pak procesor (MMU) vyvolá přerušení stránka nenalezena, které musí obsloužit operační systém.

Nastalo-li přerušení stránka nenalezena, musí operační systém požadovanou stránku do fyzické operační paměti nahrát. Nejsou-li všechny rámy obsazeny, pak OS umístí požadovanou stránku ze stránkovacího souboru do volného rámu (fyzické operační paměti) a opraví tabulku stránek a opustí přerušení.

Jsou-li všechny rámy obsazeny, operační systém rozhodne podle nějakého kritéria, kterou stránku z rámu uloží do odkládacího prostoru na disku. Uvolněný rám nahradí požadovanou stránkou, která způsobila přerušení. V tabulce stránek opraví bit přítomnosti stránky a opustí přerušení.

Z hlediska procesů je mechanismus překladu virtuální stránky na fyzickou plně transparentní (neviditelný).

Na počítačích x86 kompatibilních může mít každý program k dispozici několik 4GB adresových prostorů. Hodnoty uložené v segmentových registrech procesoru se v chráněném režimu využívají jako indexy do speciální tabulky (tabulka deskriptorů), kde jsou definované vlastnosti jednotlivých „segmentů" (spíše oblastí) paměti. Jejich popis je však zcela mimo rámec publikace.

8.4 Souborové systémy

Pro práci s daty na nosičích slouží v operačních systémech část nazývaná souborový systém. Realizuje ukládání, uchování a přístup (zpětné vybavení) k datům.

V pionýrských dobách počítačů musel programátor znát fyzické umístění dat na záznamovém médiu. Manipulace s daty byla obtížná a tak se součástí jádra operačního systému stal i podsystém pro správu souborů.

8.4.1 Struktura souborového systému

Souborový systém musí mít, kromě programového modulu v jádře, na záznamovém médiu uloženu datovou strukturu, která popisuje fyzické umístění jednotlivých souborů na disku, jejich přístupová práva a samozřejmě jejich název. Datová struktura obsahující všechny režijní informace o uložených souborech se nazývá metadata.

Vlastní soubor se skládá ze záznamů, pevné nebo proměnné délky. Přístup k záznamům souboru je určen jejich typem. Nejčastější přístup k souboru je sekvenční, kdy pro přečtení určitého záznamu musíme přečíst vždy všechny předchozí. Jsou-li k záznamům vytvořeny indexy podle různých klíčů, můžeme do souboru přistupovat podle zadané-

ho klíče. Soubory s těmito vlastnostmi se nazývají index-sekvenční soubory (IBM 390, AS/400).

Data v souboru nemusí být z pohledu operačního systému tvořena záznamy. Na soubor pak operační systém nahlíží jako na posloupnost jednotlivých byte. Jejich struktura je známá až aplikaci (DOS, UNIX). Zaměřme se na mechanismy přístupu k souborům pod operačními systémy DOS a UNIX.

8.4.2 Mechanismy přístupu k souborům

Každý soubor, se kterým chceme pracovat, musíme nejprve otevřít. V rámci požadavku předáme jádru operačního systému jméno souboru a požadovaný typ přístupu (čtení, zápis). Od jádra dostaneme zpět identifikační číslo souboru, kterému se česky říká rukojeť, anglicky handle nebo file descriptor. Identifikační číslo souboru budeme používat pro další práci se souborem.

Ve jméně souboru se obvykle připouští tzv. cesta k souboru. Leží-li soubor v jiném adresáři, můžeme k němu přistoupit pomocí specifikace všech adresářů nutných k nalezení souboru. Adresáře v cestě jsou oddělovány oddělovačem, jehož tvar závisí na operačním systému. Do nadřazeného adresáře se můžeme dostat pomocí speciálního symbolu (většinou).

Pozice v souboru je určena prostým číselným ukazatelem. Jeho hodnota vzroste vždy o počet přečtených nebo zapsaných byte. Ukazatel lze systémovým voláním nastavit na požadovanou hodnotu a tak lze číst nebo zapisovat od zvolené pozice.

Velikost maximálního bloku dat, který můžeme najednou přečíst nebo zapsat, je většinou omezena velikostí předávacího registru. Práci se souborem ukončíme jeho uzavřením.

S myšlenkou, že s klávesnicí a obrazovkou lze pracovat jako se soubory, přišel operační systém UNIX. Abychom mohli používat pro čtení znaků z klávesnice a zápis na obrazovku stejné systémové volání jako pro čtení a zápis souboru, musíme klávesnici i obrazovce přidělit speciální identifikační číslo souboru.

Identifikační číslo pro klávesnici je 0 a správně se takovému vstupu říká standardní vstup (stdin). Obrazovka dostala hned dvě identifikační čísla. První pro standardní výstup s číslem 1 (stdout) a druhé s číslem 2 pro standardní chybový výstup (stderr).

Operační systém umí nejen zapisovat a číst ze souborů, ale i přesměrovat identifikační čísla souboru. Pak můžeme výstup na obrazovku přesměrovat do souboru před použitím programu a jeho zdrojový kód nemusíme vůbec měnit. Pro přesměrování se používá speciální systémové volání, které obyčejně vyžaduje dva parametry – identifikační čísla souboru, které „sloučíme". Přesměrování známe z operačních systémů DOS a UNIX. Výpis na obrazovku přesměrujeme do souboru takto:

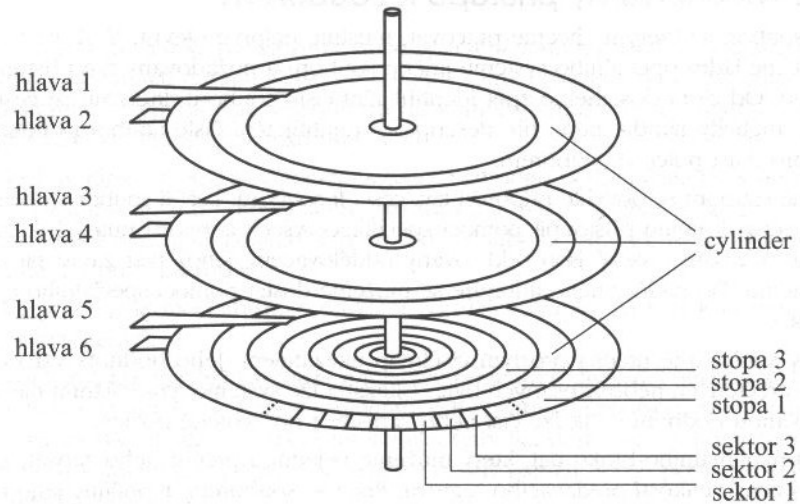
```
ls -l > soubor1
```

Ve skutečnosti příkazový interpret před spuštěním příkazu ls otevře soubor soubor1 a přesměruje systémovým voláním standardní výstup do indentifikačního čísla souboru soubor1. Nakonec program ls s parametrem -l spustí. Program ls nemá žádné tušení, že

bylo přesměrování provedeno a zapisuje stále na standardní výstup. Díky předchozímu volání se na obrazovku nic nevypíše, ale výpis příkazu ls se zapíše do souboru.

8.4.3 Fyzická struktura disku

Souborový systém přistupuje k disku přímo a musí znát jeho fyzickou strukturu. Magnetická záznamová média (disky) mají svoji strukturu z historického důvodu určenou trojicí geometrických „veličin" – válec (stopa), hlava, sektor (cylinder, head, sector).



Obrázek 44 Fyzická struktura disku

Stopy tvořily na disku soustředné kruhy a byly rozděleny na jednotlivé sektory. Nad diskem se pohybovaly čtecí/zápisové hlavy. Hodnota všech tří veličin určila sektor disku, ze kterého se data budou číst nebo na který se budou zapisovat. Podobně jako stránka, sektor je nejmenší jednotka informace, se kterou lze pracovat. Typická velikost jednoho sektoru je 512 byte.

Na počítačích PC byly pro jednotlivé „veličiny" vyhrazeny rozsahy hodnot, kterých mohly nabývat. Kapacita disků rostla a s ní i maximální hodnoty hlaviček, sektorů a stop, až se jednoho dne do intervalu nevešly. Místo úpravy rozsahů hodnot nastoupily různé transformace snažící se kombinací hodnot sektorů, hlaviček a stop navrátit zpět do povolených intervalů hodnot. Časem se ukázalo i toto řešení nedostačující.

Místo geometrické adresace sektorů na disku se začala používat adresace lineární (logická). Jednotlivé sektory na disku se jednoduše očíslovaly od 0 až do posledního přístupného sektoru na disku. K disku se přistupuje pomocí čísla požadovaného sektoru.

8.4.4 Organizace více souborových systémů na disku

Souborový systém nemusí obsadit celý disk, operační systémy mají mechanismy na logické rozdělení disku mezi více souborových systémů. Na počítačích PC je vždy první logický sektor disku vyhrazen na tabulku rozdělení disku (partition table). Každý záznam tabulky určuje první a poslední adresu místa na disku vyhrazenou souborovému systému. Informace je v tabulce rozdělení disku duplikována, jsou uvedeny jak geometrické, tak logické adresy. V případě, že je geometrická hodnota mimo rozsah, je naplněna speciální smluvenou hodnotou.

8.5 Zavedení systému na PC

Po stisku tlačítka síť (power) počítač ožije. První program zpracovávaný procesorem je uložen v paměti typu ROM, jejíž obsah je nezávislý na napájení počítače. Program uložený v paměti ROM se jmenuje BIOS (basic input output system) a jeho úkolem ve startovací fázi počítače je provést oživení (naprogramování periferních zařízení) a kontrolu periferních zařízení počítače (POST – power on self test, test paměti, detekce disků). Po úspěšné inicializaci zařízení začne BIOS zavádět operační systém. Podle nastavení se zavaděč operačního systému hledá nejprve v disketové mechanice nebo častěji na prvním (číslo nula) sektoru prvního hard disku. V prvním sektoru je kromě zavaděče operačního systému uložena již zmiňovaná tabulka rozdělení disku. Celému prvnímu sektoru disku se říká master boot record (MBR). BIOS nahraje kód z master boot sektoru do operační paměti a spustí jej.

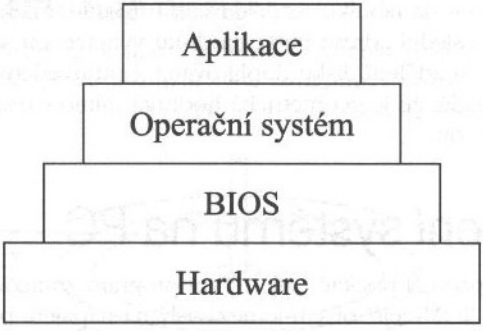
Jeho úkolem je nalezení dalšího zavaděče, uloženého v některém z logických disků, které jsou specifikovány v tabulce rozdělení disku. Kritériem výběru je příznak aktivity logického disku, přičemž označen jako aktivní smí být pouze jeden logický disk. Každý logický disk má také svůj první sektor vyhrazený pro zavaděč operačního systému. Program z MBR vybraný boot sektor logického disku nahraje do paměti a předá mu řízení. Program uložený v boot sektoru logického disku nahraje jádro operačního systému a tak operační systém nastartuje.

8.5.1 BIOS

Program BIOS je nedílnou součástí všech počítačů PC. Jeho hlavním cílem je usnadnit a zobecnit přístup k hardwarovým prostředkům počítače. V érách operačního systému DOS byl prostředníkem mezi operačním systémem a hardwarem počítače.

Číslo přerušení	Funkce
0x10	Služby obrazovky
0x13	Služby disků
0x14	Služby sériových portů
0x15	Rozšířené služby
0x16	Služby klávesnice
0x17	Služby tiskárny

Moderní operační systémy využívají služeb Biosu jen při zavádění operačního systému do paměti a ovladače hardwaru implementují samy. Rozhraní služeb Biosu je přístupné pomocí sady softwarových přerušení.



Obrázek 45 Pyramida přístupu k hardwaru počítače

Grafická karta byla vybavena také svým vlastním modulem Biosu a tak se odstínily rozdíly v hardwarovém návrhu různých grafických karet.

Pro vstup i výstup na obrazovku použijeme služeb operačního systému, a proto se Biosem nebudeme dále zabývat. Případní zájemci si mohou přečíst plnou specifikaci všech služeb Biosu na přiloženém CD.

PŘEKLADAČ NASM

9

V této kapitole:

9.1 Programová řádka
9.2 Výrazy
9.3 Lokální návěští
9.4 Preprocesor NASM
9.5 Direktivy Assembleru
9.6 Výstupní formáty

Pro překlad našich programů z jazyka symbolických adres (Assembleru) do strojového kódu budeme používat překladač NASM – Netwide Assembler, který je k dispozici zdarma (i se zdrojovým kódem překladače) pod licencí LGPL. Najdete ho na přiloženém CD nebo aktuální verzi na http://nasm.sourceforge.net.

Překladač má podobnou syntaxi jako komerční překladače MASM nebo TASM. Proto bude snadno použitelný i pro programátory, kteří se s programováním v Assembleru již setkali.

9.1 Programová řádka

Každá řádka programu se může skládat (nepovinně) z těchto částí:

návěští, instrukce, operandy, libovolný komentář.

Před návěštím i za instrukcí může být libovolný počet mezer nebo jiných oddělovačů (whitespace). Podle druhu instrukce se očekává odpovídající počet operandů, které jsou odděleny čárkou. Začátek komentáře je označen znakem středník ';' a ukončen koncem řádky.

Následující řádka se pokládá za pokračování předchozí, pokud je předchozí řádka zakončena zpětným lomítkem '\' (backslash, jako v jazyce C).

9.2 Výrazy

Syntaxe výrazů je velmi podobná jako u jazyka C. Hovoříme-li o výrazech, máme na mysli konstantní výrazy, jejichž hodnota je známa už v době překladu.

Výraz může být i součet adresy symbolu s nějakým číslem. Takový druh výrazu jsme použili při přístupu k druhému prvku pole v instrukci MOV eax,[pole+4]. Hodnota výrazu v hranaté závorce byla známa už v době překladu a byla to adresa začátku pole zvýšená o čtyři.

Výrazy jsme používali i jinde:

```
add dl,'A'-10          ;přičteme konstantní hodnotu, tak abychom získali
                       ;znaky 'A'-'F'
```

Výraz 'A'-10 se spočítal během překladu, překladač vygeneroval instrukci ADD dl,0x55.

Nemusíme se omezovat pouze na jednoduché výrazy, k dispozici je většina operátorů známých z jazyka C. V tabulce jsou uvedeny všechny podporované operátory podle vzrůstající priority.

Operátor	Význam
|	logický součet (OR)
^	logická non-ekvivalence (XOR)
&	logický součin (AND)
<< >>	bitový posuv vlevo a vpravo
+ -	plus a mínus
* / % // %%	krát, celočíselné děleno, modulo, celočíselné dělení se znaménkem a celočíselné modulo se znaménkem
unární operátory + - ~	unární plus, mínus a jedničkový doplněk (NOT)

Ukažme si několik dalších příkladů:

```
mov eax,((5*6+2)+(0x40<<2))/8      ;se vyhodnotí jako: MOV eax,0x24
```

Někdy se hodí i jedničkový doplněk (negace):

```
mov cl,~1                  ;se vyhodnotí jako: mov cl,11111110b
                           ;nebo jako: mov cl 0xFE
```

9.3 Lokální návěští

S návěštími (labels) jsme se již seznámili u instrukcí JMP a CALL. Návěští je identifikátor nepovinně zakončený dvojtečkou (dvojtečku doporučuji, protože jinak může překladač za návěští pokládat i zkomolený název instrukce). Každý identifikátor v programu musí být unikátní a tak s narůstající složitostí programu docházejí nápady na nové názvy. NASM nabízí elegantní řešení problému pomocí lokálních identifikátorů. Zaměřme se na návěští.

Lokální návěští je identifikátor začínající tečkou a je platný pouze mezi návěštími globálními. Globální návěští jsou všechny identifikátory, které nezačínají tečkou.

```
podprogram:          ;globální návěští podprogram
...                  ;program ...
.lokalni_smycka:     ;jméno lokálního návěští
...                  ;další instrukce programu

podprogram2:         ;globální návěští podprogram2
...
.lokalni_smycka:     ;můžeme použít stejné jméno, druhý identifikátor
                     ;.lokalni_smycka je nedostupný.
...
```

9.4 Preprocesor NASM

Preprocesor Assembleru NASM je velice silný a obsáhle podporuje pro práci s makry (přetěžování, rotace parametrů maker, nepovinnost parametrů atd.). Makrem rozumíme náhradu identifikátoru makra jeho expanzí. Jak se makra expandují, si ukážeme v další podkapitole.

Všechny direktivy preprocesoru začínají znakem '%' (procento). Názvy maker a definic jsou case-sensitive (rozlišují se malá a velká písmena). Nechceme-li zohledňovat velikost písmen, můžeme užít alternativní direktivy, které začínají %i (například %idefine, %imacro...). Probereme si pouze nejčastěji používané konstrukce.

9.4.1 Jednořádková makra – %define, %undef

Mohou mít na rozdíl od jazyka C parametry, ale definují se podobně. Utvořme makro pro průměr dvou parametrů.

```
%define prumer(a,b) (((a)+(b))/2)
```

V programu makro použijeme jednoduše:

```
mov al,prumer(3,7)
```

Makro se vyhodnotí jako:

```
mov al,5
```

Častěji se direktiva %define používá jako v jazyce C k definování konstant:

```
%define SEKUND_ZA_MIN  60
%define SEKUND_ZA_HOD  SEKUND_ZA_MIN * 60
```

Nebo k definování vlastností programů (podmíněný překlad).

```
%define POUZIJ_MMX
```

Existenci maker můžeme podobně jako v jazyce C otestovat direktivami %ifdef a %ifndef. Makro můžeme zrušit pomocí direktivy %undef.

9.4.2 Víceřádková makra - %macro, %endmacro

Víceřádkové makro začíná direktivou %macro, následuje jeho název a počet parametrů makra. Makro musíme ukončit direktivou %endmacro.

```
%macro odecti 3
sub %1,%2
sub %1,%3
%endmacro
```

Makro odecti musíme zavolat se třemi parametry.

```
odecti eax,ecx,[promenal]
```

Expanduje se na:

```
sub eax,ecx
sub eax,[promenal]
```

Nemusíme se omezovat na pevně zadaný počet parametrů makra, preprocesor NASM dovoluje počet parametrů zapsat jako interval. Neuvedené parametry se doplní výchozími hodnotami, které jsou uvedeny na konci v definici makra.

```
%macro secti 2-3 0
add %1,%2
adc %1,%3
%endmacro
```

Modifikované makro secti je nutné vyvolat vždy se dvěma povinnými parametry, třetí parametr nemusíme uvádět. Třetí parametr se použije buď z volání makra anebo se místo něj v případě makra secti dosadí nula, která je uvedená v definici makra.

```
secti eax ebx
```

Bude se expandovat jako:

```
add eax,ebx
add eax,0
```

9.4.3 Deklarace makra - %assign

Direktiva %assign nám umožňuje alternativní deklaraci jednořádkových maker. Makro nesmí mít žádné parametry a výsledek musí být číselná hodnota. Stejně jako u direktivy %define můžeme makro redefinovat. Direktiva %assign najde širší uplatnění v pokročilé práci s makry.

Příkazem %assign i i+1 můžeme zvýšit o jedna numerickou hodnotu makra.

9.4.4 Podmíněný překlad - %if

Stejně jako preprocesor jazyka C umožňuje NASM podmíněný překlad částí kódu.

Obecně vypadá konstrukce podmíněného překladu takto:

```
%if<podmínka>
            ;Kód mezi if a elif se přeloží pouze tehdy, je-li
            ;podmínka splněna.

%elif<podmínka2>
            ;Pro překlad se použije kód ležící mezi %elif a %else
            ;jestliže nebyla splněna první podmínka, ale druhá ano.

%else
            ;Nebyla-li podmínka a ani druhá podmínka splněna, použije
            ;se pro překlad část mezi %else a %endif
%endif
```

Části %elif a %else jsou nepovinné a lze je vynechat. Blok %elif lze použít i vícekrát.

Podmínka překladu může být (konstantní) výraz. Výrazy v podmínce jsou obohaceny o relační operátory =, <, >, <=, >= a <> (je rovno, je menší, je větší, je menší nebo rovno, je větší nebo rovno, nerovná se). Příznivci jazyka C jistě ocení možnost použití operátoru == a != místo = a <>. V podmínce jsou dovoleny i logické operátory ||, ^^, && (OR, XOR a AND), tak jak je známe z jazyka C .

9.4.5 Existence makra - %ifdef, %ifndef

Častěji než %if se pro podmíněný překlad částí kódu používá direktiva %ifdef, která testuje existenci (definici) makra.

```
%define TESTUJ

%ifdef TESTUJ
cmp eax,3
...
%endif
```

Instrukce cmp eax,3 a další uzavřené v bloku %ifdef se přeloží pouze tehdy, je-li definováno makro TESTUJ.

Podmínku existence makra můžeme negovat pomocí direktivy %ifndef. Potom se blok přeloží, když nebude makro definováno.

9.4.6 Vlož soubor - %include

Direktivou %include se preprocesor NASM opět velice přibližuje preprocesoru jazyka C. Direktiva %include má jeden parametr a tím je soubor, který se má vložit do zdrojového kódu. Název souboru se píše do uvozovek a nemusí ležet ve stejném adresáři. Překladač má přepínač, kterým můžeme specifikovat, kde tyto „hlavičkové" soubory vyhledat.

```
%include "macro.mac"
```

Nejčastěji se používají „hlavičkové" soubory pro vložení nějakých symbolických konstant nebo maker. Na rozdíl od jazyka C nemusíme v „hlavičkovém" souboru uvádět klasickou konstrukci:

```
%ifndef MACROS_MAC
    %define MACROS_MAC
    ;teď můžeme definovat makra
%endif
```

Případné vícenásobné vložení souboru totiž není chybou. Makra by se jen redefinovala a nezpůsobila by chybu překladu.

9.5 Direktivy Assembleru

Direktivy Assembleru NASM jsou makra definovaná překladačem. Na rozdíl od svých kolegů (MASM a TASM) není počet direktiv nijak nabubřelý, a proto si vystačíme jen s pár základními.

9.5.1 Direktiva BITS

Direktiva BITS sděluje překladači, v jakém režimu procesoru chceme náš program provozovat.

Její syntaxe je BITS 16 nebo BITS 32. V prvním případě překladači sdělíme, že má generovat šestnáctibitový kód, ve druhém si naopak přejeme generovat kód dvaatřicetibitový. V drtivé většině případů direktivu BITS nebudeme vůbec potřebovat, protože překladač NASM ji správně nastaví podle výstupního formátu přeloženého souboru (viz podkapitola „Výstupní formáty").

V kapitole věnované mikroprocesoru 80386 jsme si řekli, že procesor má dva režimy – reálný a chráněný. Reálný režim sloužil k zachování kompatibility s předchozími procesory. V reálném režimu se používá šestnáctibitový kód (a adresace). Mikroprocesor 80386 však má i dvaatřicetibitové instrukce. Znamená to že v šestnáctibitovém režimu se nedají dvaatřicetibitové instrukce používat?

Samozřejmě, že ne. Každá instrukce má svůj operační kód (svoji strojovou formu, tak jak jsme si to ukázali v kapitole Anatomie instrukce). Vývojáři firmy Intel museli vyřešit problém jak přidat spoustu nových instrukcí pracujících s dvaatřicetibitovými operandy, když byla tabulka operačních kódů skoro plná. Situaci vyřešili elegantně.

Šestnáctibitová instrukce mov AX,0x1234 vypadá ve strojovém kódu jako 0xB8, 0x34, 0x12.

Dvaatřicetibitová instrukce mov EAX,0x00001234 je zapsaná ve strojovém kódu jako 0x66, 0xB8, 0x34, 0x12, 0x00, 0x00.

V šestnáctibitovém (reálném) režimu procesoru začínají všechny dvaatřicetibitové instrukce ve strojovém kódu prefixem 0x66 (a reference do paměti zase prefixem 0x67). Po prefixu následuje operační kód šestnáctibitové varianty instrukce. Operandy instrukce díky prefixu mohou být i dvaatřicetibitové.

V režimu chráněném používá procesor dvaatřicetibitovou adresaci i dvaatřicetibitový kód. I když je v chráněném režimu zapsáno 0xB8, 0x34, 0x12, 0x00, 0x00 (bez prefixu 0x66), znamená to pro procesor instrukci mov eax, 0x00001234.

Direktivou BITS určujeme, do jaké formy se instrukce přeloží (s prefixem nebo bez).

9.5.2 Direktivy SECTION a SEGMENT

Každý program, nezávisle na programovacím jazyku, se skládá ze tří základních sekcí. Z programového kódu, z dat známých v době překladu (statických) a z dat dynamických (neinicializovaných).

Direktivou SECTION (nebo SEGMENT) můžeme sekce nadefinovat.

Z historických důvodů se sekce pro programový kód jmenuje .text, pro statická data (data známá v době překladu) se sekce nazývá .data, pro dynamická data (neinicializovaná) .bss.

Ukažme si typický program v Assembleru užívající všechny sekce.

```
;Program v Assembleru by měl začínat vždy popisem funkce
;a měl by mít případně vyznačena autorská práva

SECTION .text

;do sekce .text zapíšeme náš program
;v našem příkladu poněkud hloupý program, který sečte několik
;hodnot. Budeme předpokládat, že segmentové registry jsou správně
;nastaveny.

mov eax,2           ;registr EAX naplníme číslem 2
mov ebx,5           ;registr EBX naplníme číslem 5
add eax,ebx         ;provedeme součet EAX a EBX
add eax,[stat1]     ;k výsledku předchozího součtu přičteme obsah
                    ;statické "proměnné" stat1 (číslo 1)
mov [dyn1],eax      ;celkový výsledek uložíme do dynamické proměnné dyn1

SECTION .data
                    ;s pomocí sady pseudoinstrukcí DB, DW a DD v sekci
                    ;.data můžeme nadefinovat data známá v době překladu

stat1: dd 1

SECTION .bss

;V sekci pro dynamická data můžeme nadeklarovat ostatní data, u kterých
;nezáleží na počáteční hodnotě. Pro definici dat budeme v sekci .bss
;používat sadu pseudoinstrukcí RESB, RESW a RESD. Spustitelný soubor
;(na disku) nebude obsahovat vyhrazené místo pro tato data. Paměť pro
;sekci .bss se přidělí až během spouštění programu.
```

```
dyn1: resd 1
;konec programu
```

Někdy se uvádí jako další sekce programu sekce pro zásobník. Spokojme se s tvrzením, že se o zásobník postará operační systém.

Direktivy SECTION a SEGMENT jsou svými synonymy a je jedno, kterou direktivu budeme používat. V podkapitole o výstupních formátech překladače si povíme o některých rozšířeních direktiv SECTION a SEGMENT.

9.5.3 Direktivy EXTERN, GLOBAL a COMMON

Trojici direktiv EXTERN, GLOBAL a COMMON užijeme až v kapitole věnované propojení programů v Assembleru s vyššími programovacími jazyky.

Direktiva EXTERN se velmi podobá direktivě extern z jazyka C. S její pomocí můžeme deklarovat symboly, které nejsou definovány v tomto modulu (programu). Direktiva extern smí mít libovolný počet parametrů oddělených čárkou, každý parametr je název symbolu, který se bude importovat.

Direktiva GLOBAL má stejnou syntaxi jako direktiva EXTERN. Direktivou GLOBAL říkáme ostatním modulům, že je symbol veřejný a k dispozici ostatním.

Direktivu COMMON užijeme místo direktivy GLOBAL všude tam, kde budeme chtít exportovat symboly, které jsou deklarovány jako neinicializovaná data v sekci .bss.

Bude-li jiný modul exportovat stejný symbol (neinicializovaný) pomocí direktivy COMMON, sloučí se se symbolem z tohoto modulu. Ve výsledném spustitelném souboru budou mít takové symboly přidělenou stejnou adresu v paměti.

9.5.4 Direktiva CPU

Direktivou CPU můžeme po překladači požadovat užití instrukcí dostupných pouze pro určitý typ procesoru.

Procesory x86 kompatibilní během svého dlouhého vývoje dostávaly nové instrukce zjednodušující instrukce stávající.

My jsme se seznámili pouze s instrukcemi procesoru 80386, a tak ani direktivu CPU nebudeme potřebovat. Specifikaci instrukční sady (direktivu CPU) použije spíše zkušenější programátor, který bude využívat nové instrukce určitého procesoru.

Direktiva CPU má jeden parametr, kterým je typ procesoru, jehož instrukční sadu budeme chtít používat. Není-li direktiva CPU v programu uvedena, budou se moci používat všechny instrukce všech procesorů x86 kompatibilních.

```
CPU 8086    ;Budeme moci používat pouze instrukce původního procesoru
            ;Intel 8086 Obdobně můžeme uvést restrikce i pro jiné
            ;procesory 286, 386, 486 Pentium ...
```

9.6 Výstupní formáty

NASM – Netwide Assembler – je snadno přenositelný na mnoho operačních systémů, které jsou postaveny na procesorech x86 a kompatibilních. Z tohoto důvodu překladač NASM podporuje široké spektrum výstupních (objektových) formátů.

Druh formátu výstupního souboru můžeme specifikovat na příkazové řádce pomocí přepínače -f. Některé výstupní formáty přidávají další direktivy nebo rozšiřují direktivy stávající.

Vytvoření spustitelného souboru programu můžeme rozdělit do dvou kroků. Prvním krokem bude překlad (kompilace) zdrojového kódu (v našem případě Assembleru) do některého z objektových formátů.

Objektový formát už obsahuje kód programu zapsaný ve strojovém kódu, ale adresy symbolů (proměnných a dalších symbolů) nejsou ještě určeny.

Druhým krokem k vytvoření spustitelného souboru je tzv. sestavení (link). Sestavovací program (linker) z objektových souborů „vyrobí" soubor(y) spustitelné. Sestavovací program propojí symboly definované v jednom modulu programu s jinými moduly programu, tak jak jsme požadovali pomocí direktiv EXTERN a GLOBAL. Všem symbolům se přiřadí nějaká konkrétní adresa v paměti nebo se zajistí dynamicky její výpočet.

9.6.1 Výstupní formát bin

Výstupní formát bin není objektovým formátem. Jeho výstupem je pouze přeložený kód, který jsme předložili překladači.

Výstupní formát bin můžeme použít i k vytvoření textového souboru, který bude obsahovat pouze jednu řádku – řetězec Hello World!

```
;pokud není uvedeno jinak, bude binární soubor adresován od 0x0
;pomocí direktivy DB napíšeme řetězec "Hello World" ukončený
;oddělovačem řádky

retezec: db "Hello, World!",0xd,0xa
```

Zdrojový soubor pojmenujeme hello.asm. Přeložit ho můžeme pomocí překladače NASM s parametrem -f:

```
nasm -f bin hello.asm
```

Výsledkem bude soubor hello, který bude obsahovat řetězec „Hello, World!". Kdybychom do programu uvedli i instrukce mikroprocesoru, dostali bychom v textovém editoru „rozsypaný čaj", protože operační kódy instrukcí mikroprocesoru jsou různé znaky z ASCII tabulky (jsou to různé byte hodnoty, které se pokoušíme interpretovat jako znaky).

Výstupní formát bin dovoluje deklaraci sekcí programu (programová, datová a datová pro neinicializovaná data). Za parametr jména sekce můžeme připsat direktivu align a určit tak zarovnání sekce na určité adresy. Například chceme-li definovat kódovou

sekci zarovnanou na násobky šestnácti, použijeme direktivu SECTION s těmito parametry:

```
section .text align=16
```

Typ výstupního formátu bin můžeme použít pro tvorbu spustitelných programů pro operační systém DOS (.COM a .SYS) nebo pro zavaděče operačního systému apod. Výchozí nastavení direktivy BITS je 16.

Výstupní formát bin přináší jednu novou direktivu.

9.6.2 Direktiva ORG

Direktivou ORG můžeme nastavit počáteční adresu, od které se budou ostatní adresy v programu odvozovat.

Výhodné použití direktivy ORG bude u spustitelného souboru typu COM, kdy direktivou ORG 0x100 nastavíme počáteční adresu programu na 0x100 (OS DOS zavádí soubory typu .COM v rámci segmentu od adresy 0x100).

Následující odstavec je vhodný pro programátory, kteří si chtějí prohloubit své znalosti při práci s překladačem NASM. Začátečníci mohou s klidem přejít na novou podkapitolu.

Direktivu ORG nemůžeme použít vícekrát v rámci jedné sekce, tak jako u překladačů MASM nebo TASM.

Typickým případem vícenásobného užití direktivy ORG je použití druhé direktivy ORG k označení konce boot-sektoru.

V překladači MASM nebo TASM jsme mohli uvést:

```
ORG 0
                ;kód boot-sektoru
...
                ;značka konce boot-sektoru přesně na 511 a 512 byte
ORG 510
DW 0xAA55
```

Překladačem NASM musíme problém vyřešit s pomocí direktivy TIMES.

```
ORG 0
                ;kód boot-sektoru
...
                ;značka konce boot-sektoru přesně na 511 a 512 byte
TIMES 510-($-$$) DB 0
DW 0xAA55
```

9.6.3 Výstupní formát obj (Microsoft OMF)

Výstupní formát OBJ je stejný jako produkují překladače TASM nebo MASM. Jak už název napovídá, jedná se o objektový formát, proto bude nutné přeložený objektový sou-

bor ještě sestavit sestavovacím programem (linker), abychom získali spustitelný soubor. Výchozí přípona výsledného objektového souboru je .obj.

Ačkoli byl obj-formát původně navržen pouze pro šestnáctibitový mód (adresaci), překladač NASM podporuje i dvaatřicetibitové rozšíření formátu. Což umožňuje překladač NASM používat s dvaatřicetibitovými překladači firmy Borland, které používají právě tento druh rozšířeného dvaatřicetibitového obj-formátu. (A ne formát užívaný firmou Microsoft ve svých produktech.)

Výstupní formát OBJ také rozšiřuje syntax některých direktiv, především direktivy SEGMENT (SECTION). Popis jednotlivých rozšíření je ovšem mimo rámec této publikace a případné zájemce bych prozatím odkázal na manuál (anglický), který je součástí překladače NASM.

Přidanou direktivou, kterou použijeme v kapitole věnované programování ve Windows, je direktiva IMPORT.

Direktiva IMPORT zajistí vložení (import) požadovaného symbolu z dynamické knihovny. Očekává dva parametry, jméno symbolu a jméno dynamické knihovny.

Formát OBJ vyžaduje, aby aspoň jeden soubor měl definován vstupní bod (entry point) programu (to je místo, kam se předá řízení po zavedení programu do paměti). Vstupní bod programu vyznačíme speciálním symbolem (návěštím) ..start:

Pomocí přepínače -f obj překladači sdělíme, že si přejeme použít výstupní formát obj.

9.6.4 Výstupní formát win32

Pokud budeme chtít náš program propojit s Microsoft Visual C++ apod, užijeme výstupní dvaatřicetibitový formát -f win32. Ačkoli má být tento formát kompatibilní s formátem COFF (Common Object File Format), tak bohužel není.

Chceme-li použít sestavovací programy používající COFF formát, budeme muset použít další z podporovaných formátů -f coff.

9.6.5 Výstupní formát coff

Výstupní formát COFF podporuje například známé volně šiřitelné prostředí pro vývoj aplikací v jazycích C a C++ – DJGPP (DJ's GNU Programming Pack). Výstupní soubor bude ve formátu COFF, vyvoláme-li překladač s parametrem nasm -f coff. Výchozí přípona objektového souboru je .o.

9.6.6 Výstupní formát elf

S výstupním formátem ELF (Executable and Linkable Format) se setkáme v mnoha systémech „Unix-kompatibilních". Formát ELF použijeme při překladu programů (nejen v Assembleru) v operačních systémech Linux, Solaris x86, Unixware, SCO-UNIX a UNIX System V.

Výchozí přípona vytvořeného objektového souboru je .o. Jednotlivé programové sekce mohou mít další atributy.

My se spokojíme s konstatováním, že správné atributy sekcí budou vybrány, pokud použijeme již připravené názvy. Pro kódovou sekci (segment) užijeme název .text (kód bude spustitelný a chráněný proti zápisu). Datovou sekci pojmenujeme .data (do sekce lze zapisovat). A nakonec sekci pro neinicializované (dynamické) proměnné pojmenujeme .bss.

Výstup ve formátu ELF zajistíme použitím přepínače -f elf.

9.6.7 Výstupní formát aout a aoutb

Objektový formát a.out se dříve používal v operačním systému Linux. Vylepšený formát a.out je užit ve skupině operačních systémů BSD (NetBSD, FreeBSD a OpenBSD). NASM umí generovat oba druhy výstupní formátů a.out parametrem -f aout pro Linux a parametrem -aoutb pro operační systémy BSD.

9.6.8 Symbolické informace

Různé výstupní formáty nabízejí tzv. přibalení symbolických informací do výsledného objektového souboru. Symbolické informace pomáhají odladovacímu nástroji (debuggeru) zpětně přiřadit přeložené adresy v programu na jména jednotlivých symbolů (návěští, názvy proměných apod.). Parametrem nasm -g můžeme překladač požádat o přibalení odladovacích informací do objektového souboru. Relevantní bude přepínač pro programy ve výstupním formátu OBJ nebo ELF.

PROGRAMUJEME V DOSU

10

V této kapitole:

10.1 Adresace paměti v reálném režimu

10.2 Jak vypadá uspořádání paměti v DOSu

10.3 Rozšíření paměti nad 1 MB

10.4 Druhy spustitelných souborů v DOSu

10.5 Základní systémová volání

10.6 Vstupně-výstupní operace se soubory (I/O)

10.7 Práce s adresáři

10.8 Alokujeme paměť

10.9 Předávání příkazové řádky

10.10 Chybové kódy

10.11 Odladujeme

10.12 Debugger grdb.exe

10.13 Návrh a odladění programu

10.14 Rezidentní program aneb Peklo na zemi 1

10.15 Volně šiřitelné zdroje informací

„640 kB should be enough for everybody."

(Bill Gates 1981)

Operační systém DOS (MS-DOS, Microsoft Disk Operating System) byl prvním operačním systémem pro počítače IBM PC. Svojí architekturou (i kompatibilitou) vycházel z tehdy oblíbeného operačního systému CP/M, který se používal v šestnáctibitových a osmibitových počítačích.

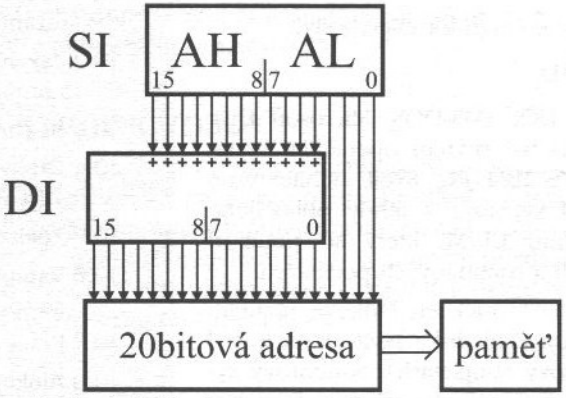
Operační systém DOS měl jen řádkové rozhraní (příkazovou řádku). Z hlediska počtu úloh to byl systém jednoúlohový (singletask). Souborový systém neměl nijak ošetřena práva přístupu k souborům, tak jak je známe z operačních systémů UNIX. Původně měl k dispozici 640 kB (cca 0,5 MB) operační paměti. Později se různými mechanismy dala paměť rozšiřovat.

Nikdy neběžel v chráněném režimu (byl navržen pro 8086), a tak programy mohly přímo ovládat hardware počítače i zapisovat do paměti, která jim nepatřila. S příchodem procesorů 80286 a 80386 přišla možnost práce v chráněném režimu. Aplikace, které chtěly chráněný režim používat, musely samy zajistit přepnutí módu procesoru do chráněného a stejně tak obstarat i stránkování a ostatní režijní činnosti. Je pochopitelné, že nemohlo probíhat více různých aplikací v chráněném režimu současně.

10.1 Adresace paměti v reálném režimu

Mikroprocesor 8086 měl jen dvacetibitovou adresovou sběrnici, což znamenalo, že byl schopen adresovat 2^{20} byte operační paměti (jeden megabyte). Víme už, že byl vybaven jen šestnáctibitovými registry, jak je tedy možné, že mohl adresovat dvacet bitů paměti?

Odpověď hledejme v segmentových registrech. Výsledná adresa místa v paměti, tedy adresa, co se „pouštěla" na adresovou sběrnici počítače, se vypočítala jako součet příslušného šestnáctibitového registru s hodnotou umístěnou v segmentovém registru vynásobenou šestnácti (posunutou o 4 bity doleva).



Obrázek 46 Princip výpočtu fyzické adresy

Ukažme si příklad:

```
mov al,[ds:si]      ;instrukce naplní registr AL z adresy určené dvojicí
                    ;registrů DS:SI
```

Nechť je registr DS nastaven na hodnotu 0x559E a registr SI na hodnotu 0x100.

Výsledná dvacetibitová adresa se určí takto:

0x559E*0x10 (0x10 je 16) + 0x100

Výsledná adresa místa v paměti bude: 0x559E0 + 0x100 = 0x55AE0

Hodnotě uložené v segmentovém registru se říká segment a hodnotě druhé (uložené v univerzálním registru nebo přímé) se říká offset (posuv).

Všimněme si, že adresa 0x55AE0 je určena také jinou dvojicí segmentu a offsetu. Například vyhovuje i segment 0x558E a offset 0x200.

Adresace pomocí segmentů a offsetů přináší řadu nevýhod. Offset smí být jen šestnáctibitový, což znamená, že celková velikost jednoho segmentu v paměti může být maximálně 64 KB. Chceme-li utvořit program, jehož programový kód nebo data přesáhnou hranici 64 KB, musíme program rozdělit do několika segmentů.

Segmentové registry budeme muset měnit podle potřeby na jiné adresy. Řešení problémů přinášející rozdělení programu do několika segmentů ponechme na kapitole věnované typům spustitelných souborů v operačním systému DOS.

I s příchodem novějších procesorů se v reálném režimu adresovalo šestnáctibitově (z důvodu zachování programové kompatibility), a proto je dodnes možné provozovat stařičké programy pod operačním systém DOS včetně operačního systému na nejnovějších procesorech x86 kompatibilních.

10.2 Jak vypadá uspořádání paměti v Dosu

Již jsme si řekli, že operační systém DOS mohl díky reálnému režimu procesoru adresovat jen 1 MB operační paměti. Podívejme se, jakým způsobem jsou jednotlivé adresy rozděleny pro operační systém, programy (aplikace) a pro hardware.

Adresa (segment:offset)	Užití
0x0000:0x0000	Tabulka vektorů přerušení
0x0040:0x0000	Tabulka proměnných Biosu
0x0????:0x0000	Jádro operačního systému DOS
0x0????:0x0000	Vyrovnávací paměti, obslužné programy přerušení, ovladače zařízení
0x0????:0x0000	Rezidentní část interpretu příkazů COMMAND.COM
0x0????:0x0000	Rezidentní programy (TSR)
0x0????:0x0000	Právě běžící program (Volkov commander, T602 apod.)
0x0????:0x0000	Volná paměť
0x0A000:0x0000	Začátek video ram (v grafickém režimu)
0x0B000:0x0000	Video ram monochromatického grafického adaptéru (v textovém režimu)
0x0B800:0x0000	Video ram grafického adaptéru v textových režimech
0x0C800:0x0000 - 0x0E000:0x0000	Volné adresy, pro přídavné ROM
0xF000:0x0000	ROM BIOS

Začátek adresového prostoru je vyhrazen pro tabulku přerušení. Její velikost můžeme zjistit snadno. Procesor podporuje celkem 256 vektorů přerušení, každý vektor přerušení se skládá z dvojice offset, segment, čili zabírá 4 byte. Vynásobením 256*4 dostáváme 1 KB.

Ihned za tabulkou přerušení na fyzické adrese 0x400 (0x0040:0x0000) následuje tabulka proměnných Biosu. Obsahuje seznamy zařízení připojených k počítači i jejich konfiguraci. Popis jednotlivých položek je zcela mimo rámec publikace a případné zájemce o tuto problematiku bych odkázal na přiložené CD, kde mohou použít elektronický manuál Ralf Brown Interrupt List.

Na následujících adresách je nahrán operační systém, včetně různých vyrovnávacích pamětí a obsluhy přerušení. Za operačním systémem je nahrán příkazový interpret COMMAND.COM, který umožňuje komunikaci s uživatelem prostřednictvím známé příkazové řádky.

V další položce tabulky se ukrývá tajemný název rezidentní program. Rezidentní program se spustil jako obyčejný program, jenže po svém ukončení zůstal nahrán v paměti. Typickým rezidentním programem je například ovladač myši. Program ovládající myš musí zůstat v paměti, aby ostatní aplikace mohly myš používat.

Programové rozhraní umožňující aplikacím pod OS DOS spolu (mezi sebou nebo i s OS) komunikovat je realizováno pomocí vektorů přerušení. Podrobněji se s problematikou seznámíme v kapitole věnované základním systémovým voláním.

Na následujících adresách je nahrán právě spuštěný program, který může využívat paměť až do 640 KB, neboli do adresy 0x0A000:0x0000. Adresy od 640 KB až do 1MB jsou vyhrazeny pro hardware počítače.

Prvních 64 KB paměti je použito grafickou kartou. Zápisem na tyto adresy můžeme kreslit po obrazovce jednotlivé body (zjednodušeně řečeno).

Adresy od 0x0B800:0x0000 jsou využívány grafickým adaptérem k uložení jednotlivých znaků viditelných na obrazovce. Adresa 0x0B800:0x0000 je levý horní roh prvního zobrazeného řádku v textovém režimu. Data jsou organizována po dvojici byte. První byte je ASCII hodnota zobrazovaného znaku a druhý byte určuje jeho barvu.

Zbytek adresového prostoru je určen pro programy uložené v pamětích typu ROM (Read only Memory). Najdeme zde BIOS grafického adaptéru i BIOS počítače.

10.3 Rozšíření paměti nad 1 MB

Za několik let začalo být 640 KB operační paměti málo, a tak se začaly vymýšlet způsoby, jak paměť rozšířit, a přitom zachovat reálný režim procesoru. Procesor 80286 měl již čtyřiadvacetibitovou adresovou sběrnici, ovšem v reálném režimu mohl pořád adresovat jen dvacet bitů, tedy 1 MB.

Poslední segment, který se dal využít celý, byl segment 0x0F000, kde je uložen BIOS. Kdybychom chtěli použít další segment ležící za Biosem, museli bychom do segmentového registru nahrát hodnotu 0x0FFFF. Jenže jakmile k této hodnotě přičteme offset, dostáváme se za hranici 1 MB (za dvacátý bit adresové sběrnice).

Vývojáři se rozhodli vybavit počítač speciálním hardwarem, který by ovládal jednadvacátý bit adresové sběrnice tak, aby se ještě dal využít segment s adresou 0x0FFFF. Mikroprocesor by přistoupil do paměti za jedním megabajtem správně, pokud by „někdo z venku" nastavil jednadvacátý bit adresové sběrnice na 1. „Někdo" se jmenuje hardware A20 a ovládá zmíněný jednadvacátý bit adresové sběrnice a tak přidává 64 KB operační paměti (A20 proto, že se čísluje od nuly).

Přidali jsme 64 KB a to je málo, mezi 640 KB a 1 MB je spousta nevyužité paměti, rezervované pro systémové ROM. Existoval by způsob, jak volné adresy využít?

Samozřejmě, že ano. Konsorcium tří firem Lotus, Intel a Microsoft vytvořilo specifikaci paměti, která dostala název LIM EMS, expanded memory. V oblasti mezi 640 KB a 1 MB vytvořili 64 KB velké okno, které se skládalo ze čtyř 16KB stránek. 64KB okno se nazývá stránkový rám. Název je to příznačný, neboť EMS umožňovala do takového „rámu" namapovat paměť, která ležela za 1 MB operační paměti. Na počítačích 80286 byla paměť EMS implementována hardwarově, později se díky procesoru 80386 a jeho virtuálnímu reálnému módu mohla implementovat softwarově.

Posledním možným druhem „přístupu" nad jeden megabajt v reálném režimu byla takzvaná paměť XMS (extended memory specification). Rozhraní bylo implementováno pomocí ovladače HIMEM.SYS, který na požádání přesunul blok paměti nad 1 MB do paměti do 1 MB (nebo opačně). Ovladač HIMEM.SYS vždy při požadavku na přesun přepnul počítač do chráněného režimu, kde mohl již adresovat celou paměť. Bloky přesunul a přepnul režim procesoru zase zpět.

10.4 Druhy spustitelných souborů v Dosu

Operační systém DOS podporuje tři druhy binárních spustitelných souborů.

Prvním z nich je systémový ovladač zařízení, jehož název je zakončen klasickou koncovkou .SYS. Zavádí se během spouštění operačního systému. Seznam zaváděných zařízení je uveden v konfiguračním souboru CONFIG.SYS. Popis a možné funkce jsou mimo rámec této publikace.

Aplikační programy mohou být uložené ve zbývajících dvou spustitelných formátech. Jejich koncovky jsou .COM a .EXE.

Spustitelný soubor typu COM nemá žádnou hlavičku jako ostatní spustitelné formáty (soubory). Jeho maximální možná velikost včetně dat může být jen 64 KB - 256 B. Během zavádění programu operační systém alokuje nový segment paměti (64 KB). Program se z disku nahraje od adresy 0x100.

Při spouštění programu musí operační systém nastavit segmentové registry (CS, ES, DS a SS) na adresu segmentu, kam byl program zaveden. Ukazatel na vrchol zásobníku SP se nastaví téměř na konec segmentu – tedy na adresu 0x0FFFE. Řízení se předá na offset 0x100, kde je nahraná první instrukce ze spustitelného souboru.

Paměť od offsetu 0 do 0x100 je vyhrazena pro operační systém. Říká se jí PSP (program segment prefix). Operační systém zde (například) ukládá informace o otevřených souborech a o adrese PSP rodičovského procesu. Od offsetu 0x80 jsou uloženy parametry programu.

Představme si program typu COM nahraný na segmentovou adresu 0x1234. Mapa paměti včetně prvních 0x100 byte vypadá takto:

Adresa (segment:offset)	Význam
0x1234:0x0000	Začátek vyhrazené oblasti pro OS, obsahuje některé interní struktury
0x1234:0x0080	Od této adresy jsou programu předány jeho parametry.
0x1234:0x0100	Od adresy 0x100 operační systém nahrál z disku COM soubor
0x1234:0x????	Konec programového segmentu (sekce)
0x1234:0x????	Konec datového segmentu (sekce)
0x1234:0x????	Začátek sekce pro nezinicializovaná data
....	Až do konce segmentu je paměť přidělena, zásobník roste směrem k nižším adresám.
0x1234:0xFFFE	Vrchol zásobníku

Spustitelný soubor typu EXE je oproti spustitelnému souboru typu COM podstatně vylepšen.

Hlavní vylepšení spočívá v podpoře programů přesahujících velikost 64 KB. Soubor typu EXE může obsahovat několik kódových a datových segmentů. I pro zásobník je vyhrazen samostatný segment.

Používáme-li v reálném režimu více segmentů, musíme během chodu programu vyměňovat hodnoty v segmentových registrech, tak abychom pracovali vždy se žádaným segmentem. Bude-li podprogram ležet v jiném segmentu, můžeme k jeho vyvolání použít instrukci call far, která správně modifikuje oba registry (CS i IP). K návratu použijeme instrukci retf. Obdobně bude-li nějaká proměnná ležet v jiném datovém segmentu, nastavíme pomocí pseudoinstrukce SEG její segmentovou adresu, kterou uložíme do segmentového registru. Pak budeme mít zajištěno, že společně s jejím offsetem (ten je reprezentován jejím identifikátorem) budeme skutečně přistupovat k žádané proměnné.

Navržené schéma funkce je sice správné, ovšem s malým zádrhelem. Jak zjistíme adresy segmentů, když nevíme, od které adresy se segmenty (program) zavedou?

Přeložený program na disku je uložen ve stejném tvaru, jako by byl zaváděl na adresu 0x0000:0x0000. Potom je všechno v pořádku. První segment začíná adresou 0x0000:0x0000 a končí adresou 0x0000:0xFFFF. Druhý segment bude začínat na adrese 0x1000:0x0000 atd.

Všechna místa v programu, na kterých se pracuje s hodnotami segmentů (0x0000, 0x1000 atd), se zaznamenají do tzv. relokační tabulky.

Program se nahraje do paměti od nějaké základní adresy. Pomocí relokační tabulky se ke všem místům v programu (mohou ležet i v polovičce instrukce) přičte hodnota základní adresy a tak se provede relokace programu na právě zavedenou základní adresu. Program má již všechny hodnoty segmentových registrů na správných hodnotách a může se vykonat.

10.5 Základní systémová volání

Operační systém nabízí spuštěnému programu (úloze) řadu služeb. S pomocí operačního systému bude moci náš program vypisovat zprávy na obrazovku, číst z klávesnice, pracovat ze soubory nebo požádat o přidělení paměti. Služby operačního systému jsou dostupné přes přerušení 0x21. Parametry systémových volání se předávají v dohodnutých registrech. V operačním systému DOS se číslo požadované funkce předává v registru AH. Skončí-li systémové volání (funkce) s chybou, bude nastaven příznak přenosu a registr AX bude obsahovat kód chyby.

10.5.1 Ukonči program

Nejjednodušší spustitelný program bude takový, který ihned po svém spuštění skončí. Požadavek na ukončení programu sdělíme operačnímu systému vyvoláním funkce Dosu 0x4C, která program ukončí a řízení předá rodiči.

Vstup:	Výstup:
Ukonči program: AH = 0x4C AL = návratový kód	nic

V registru AH předáme číslo požadované funkce (ukonči) a v registru AL předáme rodiči tzv. návratový kód. Dohodnutým návratovým kódem můžeme rodiči sdělit, jestli jsme naši činnost skončili bez chyby, nebo vhodným kódem signalizovat typ chyby. Funkce se podobá funkci exit známé z jazyka C.

Náš první program bude mít jen kódovou sekci:

```
SECTION .text
;První program pro operační systém DOS
;Jen ukončí svoji činnosti

org 0x100           ;budeme generovat soubor COM, zavádí se od adresy
                    ;0x100
mov ah,0x4C         ;funkce ukonči program
mov al,0            ;návratový kód 0
int 0x21            ;zavoláme jádro operačního systému

;konec programu
```

Program uložíme do souboru skonci.asm. Ve svých dalších experimentech v operačním systému DOS budeme používat typ souboru COM. Nebudeme muset složitě nastavovat segmentové registry a budeme se moci soustředit na řešení ostatních problémů.

Program skonci.asm přeložíme překladačem nasm.

```
nasm -f bin -o skonci.com skonci.asm
```

Parametrem -f specifikujeme typ výstupního formátu na bin a parametrem -o název výstupního souboru na skonci.com.

Spustit vytvořený program skonci.com můžeme tak, že na příkazové řádce zapíšeme skonci a stiskneme klávesu Enter. Program nic nevypsal, řízení se vrátilo zpět příkazovému interpretu.

10.5.2 Vypiš řetězec

Důležitou funkcí, kterou budeme často potřebovat, je vypsání znakového řetězce na obrazovku. Prozatím nebudeme využívat funkci pro zápis do souboru, ale představíme si funkci jinou, která bude jen vypisovat řetězec na obrazovku.

Z historických důvodů není vypisovaný řetězec zakončen nulovým byte, ale znakem '$' (0x24).

Vstup:	Výstup:
Vypiš řetězec: AH = 0x09 DS:DX = ukazatel na řetězec zakončený znakem '$'	nic

Budeme-li chtít vypsat pouze jeden znak na standardní výstup, použijeme funkci Dosu 0x02 a v registru DL předáme ASCII kód vypisovaného znaku.

Vstup:	Výstup:
Vypiš znak: AH = 0x02 DL = vypisovaný znak v kódu ASCII	nic

Již máme dostatek informací, abychom realizovali první program, který vypíše na obrazovku řetězec „Hello, World!" a ukončí se.

10.5.3 Hello world!

Předchozí program skonci.asm mírně modifikujeme. Přidáme datovou sekci .data, ve které nadefinujeme „proměnnou" hello, která bude obsahovat text „Hello, World!". Za řetězcem uvedeme ještě dva speciální znaky CR (0xd), LF (0xa), které při „vytištění" na obrazovku provedou odřádkování. Řetězec ukončíme požadovaným znakem '$'.

```
SECTION .text
;Program vypíše text Hello, World! na obrazovku, odřádkuje a skončí


org 0x100           ;budeme generovat soubor COM, zavádí se od adresy
                    ;0x100
mov ah,0x9          ;Funkce Dosu vypiš řetězec
mov dx,hello        ;registr DX naplníme ukazatelem (offset) na řetězec
int 0x21            ;vyvoláme přerušení Dosu
mov ah,0x4C         ;funkce ukonči program
mov al,0            ;návratový kód 0
int 0x21            ;zavoláme jádro operačního systému
```

```
SECTION .data
hello DB "Hello, World!",0xd,0xa,'$'
;konec programu
```

Segmentový registr DS jsme nemuseli nijak nastavovat. Operační systém ho nastavil na náš jediný programový a zároveň datový segment.

Program nazveme hello.asm. Přeložíme překladačem nasm -f bin -o hello.com hello.asm.

Program spustíme příkazem hello.

10.5.4 Vstup z klávesnice

Chceme-li číst znaky postupně jeden za druhým tak jako pomocí funkce getchar jazyka C nebo pascalské readkey, použijeme funkci Dosu číslo 0x01.

Vstup:	Výstup:
Přečti znak: AH = 0x01	V registru AL se vrátí znak načtený ze standartního vstupu (klávesnice)

Funkce bude čekat na stisk klávesy. Po jejím stisku ji opíše na standardní výstup – obrazovku (nechceme-li ji současně vypsat, použijeme funkci Dosu 0x08).

V registru AL se vrátí nula, byla-li stisknuta klávesa s tzv. rozšířeným kódem ASCII. Pak je třeba zavolat funkci znovu a v registru AL se vrátí číslo rozšířeného kódu ASCII (který přiřazuje identifikační číslo i jiným klávesám – šipky, page up, page down, home, end, F1...F12 apod.).

Často budeme požadovat načtení řetězce určité délky ukončeného stiskem klávesy Enter. Operační systém DOS pro nás připravil i tuto funkci.

Vstup:	Výstup:
Přečti více znaků. AH = 0x0A DS:DX = ukazatel na buffer, speciálně formátovaný	Buffer bude vyplněn řetězcem.

Ukazatel musí odkazovat na speciální strukturu, která bude naplněná přečteným řetězcem.

První byte struktury určuje, kolik znaků (1 – 254) se může přečíst, než bude pípnutím vyžadován stisk klávesy Enter. V následujícím byte bude uložen počet skutečně přečtených znaků bez započítaného ukončovacího znaku 0x0D.

```
mov ah, 0x0A        ;do registru AH ulož číslo volání funkce vstup
                    ;řetězce
mov dx,retezec      ;do registru DX ulož adresu (offset) struktury
                    ;retezec
int 0x21            ;vyvoláme přerušení Dosu
...
retezec db 5,0,0,0,0,0,0
```

Po provedení instrukce int 0x21 bude operační systém číst klávesy tak dlouho, dokud nebude stisknuta klávesa Enter. Do struktury řetězec se uloží maximálně 4 znaky (nepočítáme Enter). Každý pokus o uložení dalšího skončí pípnutím. Kdybychom stiskli klávesy A, B, C a nakonec Enter, bude struktura retezec naplněna takto:

```
retezec db 5,3,0x65,0x66,0x67,0x0D,0
```

První byte se nezměnil, druhý obsahuje délku načteného řetězce bez ukončovacího znaku 0x0D a od následujícího byte bude uložen načtený řetězec (ABC) ukončený znakem 0x0D.

Ukažme si několik příkladů:

Př.: Vypracuje program, který každý stisknutý znak přepíše dvakrát na obrazovku. Program se ukončí po stisku klávesy Enter.

Pro vstup znaku z klávesnice použijeme funkci Dosu 0x01, která zároveň stisknutý znak vypíše na obrazovku. Druhý (stejný) znak vypíšeme na obrazovku funkcí Dosu 0x02, která tak učiní.

```
;Program vypíše všechny stisknuté znaky na obrazovku dvakrát
SECTION .text

znovu:
mov ah,0x01          ;funkce Dosu cti znak z klávesnice
int 0x21             ;vyvolej přerušení Dosu
mov dl,al            ;načtený znak zkopíruj do registru DL
cmp dl, 0xD          ;byla stisknuta klávesa Enter?
jz konec             ;jestliže ano, skoč na konec
mov ah, 0x02         ;funkce Dosu vypiš znak na obrazovku
int 0x21             ;vyvolej přerušení Dosu
jmp znovu            ;zopakuj, načítej další znak
konec:
mov ah,0x4C          ;funkce Dosu ukonči program
int 0x21             ;vyvolej přerušení Dosu

;konec programu
```

Program můžeme pojmenovat opakuj.asm. Spustitelný program opakuj.com vytvoříme pomocí překladače nasm -f bin -o opakuj.com opakuj.asm.

Př.: Vytvořte program, který přečte řetězec znaků zakončený klávesou Enter a vypíše ho pozpátku.

Funkcí Dosu 0x0A přečteme řetězec do nějaké dočasné „proměnné", kterou vypíšeme pozpátku pomocí funkce Dosu 0x02 vypiš znak.

```
SECTION .text
;Program vypíše zadaný řetězec ukončený klávesou Enter na
;obrazovku pozpátku.

org 0x100            ;budeme generovat soubor COM, zavádí se od adresy
                     ;0x100
```

```
mov ah, 0x0A          ;do registru AH ulož číslo volání funkce vstup
                      ;řetězce
mov dx,retezec        ;do registru DX ulož adresu (offset) struktury
                      ;retezec
int 0x21              ;vyvoláme přerušení Dosu
xor ax,ax             ;vynulujeme registr AX
mov al,[retezec+1]    ;přečteme počet znaků řetězce
inc dx                ;přeskočíme první byte struktury
add dx,ax             ;přičtením počtu znaků k adrese (offsetu) začátku
                      ;struktury retezec dostáváme ukazatel na poslední
                      ;znak
mov si,dx             ;obsah registru DX okopírujeme do SI
std                   ;nastavíme vlajku směru na čítání směrem dolů
tiskni_dalsi_znak:
lodsb                 ;vybereme z adresy DS:SI jeden znak a registr SI
                      ;snížíme o 1
cmp si,retezec+1      ;nejsme už na konci řetězce?
jb konec              ;ano, jsme, skončíme program
mov dl,al             ;znak uložíme do registru DL tak jak to vyžaduje
                      ;funkce.
mov ah,0x02           ;připravíme volání funkce Dosu tiskni znak
int 0x21              ;vyvoláme přerušení
jmp tiskni_dalsi_znak ;načteme další znak
konec:
mov ah,0x4C           ;funkce ukonči program
mov al,0              ;návratový kód 0
int 0x21              ;zavoláme jádro operačního systému
SECTION .data
retezec db 255,0
times 253 db ' '      ;253krát zopakujeme mezeru
;konec programu
```

Program nazveme pozpatku.asm a přeložit ho můžeme stejně jako předchozí:

```
nasm -f bin -o pozpatku.com pozpatku.asm
```

Program pozpatku patří k programům složitějším, proberme si jeho funkci detailněji.

Řetězec musíme nejprve přečíst. Operační systém nám nabízí přímo službu pro čtení řetězce (0x0A). Nesmíme zapomenout vytvořit speciální strukturu retezec, která je nutná pro správnou funkci služby.

```
mov ah, 0x0A          ;do registru AH ulož číslo volání funkce vstup
                      ;řetězce
mov dx,retezec        ;do registru DX ulož adresu (offset) struktury
                      ;retezec
int 0x21              ;vyvoláme přerušení Dosu
```

Jakmile uživatel zadal řetězec ukončený klávesou Enter, náš program bude pokračovat v činnosti. Řetězec chceme vypisovat pozpátku, a proto nejprve spočítejme adresu posledního znaku řetězce.

```
xor ax,ax           ;vynulujeme registr AX
mov al,[retezec+1]  ;přečteme počet znaků řetězce
```

I po návratu ze systémového volání bude v registru DX stále uložena adresa struktury retezec. My bychom rádi zjistili adresu (offset) konce řetězce, která bude DX + 2 + délka řetězce – 1 (v prostředí programů typu COM nemusíme celou adresu vůbec uvažovat, stačí když budeme měnit jen offset, segmentová část adresy je pro všechny sekce programu stejná).

Začátek řetězce je uložen na adrese DX + 2, přičtením jeho délky získáme ukončovací znak 0x0D. My budeme chtít vypisovat poslední znak, a proto musíme odečíst jedničku.

Místo přičítání dvojky a odčítání jedničky můžeme ušetřit a přičíst pouze jedničku. Všimněme si, že jsme nejprve museli registr AX vynulovat, i když je délka řetězce jen osmibitová. K adresám (offsetům) musíme vždy přičítat pouze stejně velké operandy.

```
inc dx              ;přeskočíme první byte struktury
add dx,ax           ;přičtením počtu znaků k adrese(offsetu) začátku
                    ;struktury retezec dostáváme ukazatel na poslední
                    ;znak
mov si,dx           ;obsah registru DX okopírujeme do SI
```

Výslednou adresu (offset) uložíme do registru SI, který používá instrukce pro práci s řetězci LODSB. Další znak bude uložen na nižší adrese, a proto musíme nastavit směr čítání „dolů", což provedeme pomocí instrukce std.

```
std                 ;nastavíme vlajku směru na čítání směrem dolů
tiskni_dalsi_znak:
lodsb               ;vybereme z adresy DS:SI jeden znak a registr SI
                    ;snížíme o 1
```

Instrukce LODSB uloží v registru AL znak načtený z adresy [ds:si] a registr SI po načtení sníží o jedna. Ihned za instrukcí LODSB provedeme srovnání na adresu retezec+1, na které už žádný načtený znak nebude nikdy uložen. Nesmíme zapomenout, že jakmile instrukce LODSB přečte první znak uloženého řetězce (adresa retezec + 2), adresa v registru SI bude po odečtení již retezec+1. Kdybychom provedli podmíněný skok v případě rovnosti srovnávaných adres (JZ), nevytiskl by se nám první (poslední) znak. Podmíněný skok JB se provede, bude-li adresa (offset) v registru SI menší než retezec+1. V registru AL bude sice místo znaku uložena hodnota z adresy retezec+1, ale bude splněna ukončovací podmínka a program skončí.

```
cmp si,retezec+1    ;nejsme už na konci řetězce?
jb konec            ;ano jsme, skončíme program
```

Zbytek programu bude jednoduchý, znak uložený v registru AL uložíme do registru DL, tak jak to vyžaduje funkce Dosu pro tisk znaku. Po vytištění znaku se musíme vrátit na instrukci LODSB, abychom přečetli znak další.

```
mov dl,al                ;znak uložíme do registru DL, tak jak to
                         ;vyžaduje funkce.
mov ah,0x02              ;připravíme volání funkce Dosu tiskni znak
int 0x21                 ;vyvoláme přerušení
jmp tiskni_dalsi_znak    ;načteme další znak
```

Program ukončíme pomocí systémového volání Dosu, funkce 0x4C.

```
konec:
mov ah,0x4C          ;funkce ukonči program
mov al,0             ;návratový kód 0
int 0x21             ;zavoláme jádro operačního systému
```

V datové sekci nadefinujeme strukturu retezec.

```
SECTION .data
retezec db 254,0
times 253 db ' '     ;253krát zopakujeme mezeru
                     ;konec programu
```

První byte struktury určuje maximální délku řetězce bez ukončovacího znaku. Druhý byte je vyhrazen pro aktuální délku uloženého řetězce, který prozatím vynulujeme. Direktivou TIMES doplníme zbytek struktury 253 mezerami.

10.6 Vstupně-výstupní operace se soubory (I/O)

V kapitole věnované operačním systémům jsme si nastínili mechanismy práce se souborovým systémem (soubory). V operačním systému DOS je princip totožný. Před zápisem nebo čtením ze souboru musíme soubor otevřít. Od operačního systému dostaneme identifikační číslo souboru (handle, file descriptor), který bude reprezentovat soubor při volání dalších funkcí. Soubor musíme po ukončení práce uzavřít.

Původní operační systém DOS ještě před sloučením s OS Windows dovoloval pouze max. osmiznakové názvy souborů a tříznakovou příponu. S příchodem Windows se objevily i tzv. dlouhé názvy souborů, dovolující max. 256znakové jméno s libovolným počtem přípon. Proberme si nejprve systémová volání (funkce) pro práci s pouze 8+3 znakovými soubory.

10.6.1 Otevři soubor

Funkce 0x3D otevře soubor. Ukazatel na jméno souboru se předává v registrech DS:DX (DS segment, DX offset). V registru AL můžeme specifikovat, jakou formu přístupu k souboru budeme potřebovat. Nejčastěji budeme soubor otevírat pro čtení a zápis (AL = 0). Výstupem funkce je buď přidělené identifikační číslo souboru v registru AX nebo chyba. Číslo chyby se vrátí v registru AX a zároveň se nastaví vlajka přenosu carry CF na 1. Návrat bez chyby je vždy indikován CF = 0.

Vstup:	Výstup:
Otevři soubor:	
AH = 0x3D	CF (příznak carry) = 0 pak,
DS:DX = ukazatel na název souboru (jméno soubor ukončené znakem 0x0)	AX = přidělený identifikátor souboru.
AL = mód přístupu	Nebo:
AL = 0 pro čtení i zápis	CF = 1, pak
AL = 1 pouze pro zápis	AX = kód chyby, nejčastější:
AL = 2 pouze pro čtení	AX = 0x0002 - soubor nenalezen
	AX = 0x0003 - cesta nenalezena

10.6.2 Uzavři soubor

Dříve než se seznámíme s dalšími operacemi nad souborem, ukažme si funkci na uzavření souboru.

Vstup:	Výstup:
Uzavři soubor:	
AH = 0x3E	CF (příznak carry) = 0 pak,
BX = identifikační číslo souboru (handle)	nic
	Nebo:
	CF = 1, pak
	AX = kód chyby

V registru BX předáme získané číslo od funkce Dosu otevři soubor a tím soubor uzavřeme. Jednoduchý program, který by soubor otevřel a ihned uzavřel, by vypadal takto:

```
SECTION .text
org 0x100
mov ax, 0x3D00      ;služba Dosu otevři soubor, mód pro čtení a zápis.
mov dx,jmeno_souboru ;a předáme ukazatel na jméno souboru, registr DS je
                    ;již správně nastaven.
int 0x21            ;vyvoláme službu Dosu
jc chyba            ;chyba?, jestliže ano, skočíme na návěští chyba.
mov bx,ax           ;AX obsahuje identifikační číslo souboru (handle)
                    ;uložíme do registru BX
mov ah, 0x3E        ;služba Dosu uzavři soubor.
int 0x21            ;vyvoláme službu Dosu
mov al,0            ;návratový kód nastavíme na úspěch
konec:
mov ah,4Ch          ;služba Dosu ukonči.
int 0x21            ;vyvoláme službu Dosu, není návratu.
chyba:
mov al,1            ;návratový kód nastavíme na 1
jmp short konec     ;skočíme na návěští konec.

SECTION .data
jmeno_souboru db "text.txt",0       ;nadefinujeme jméno souboru na text.txt
```

Program není příliš užitečný, rozšiřme jeho možnosti o výpis souboru na obrazovku. Data ze souboru budeme muset nejprve přečíst a pak vypsat na obrazovku. Pro čtení souboru je vyhrazena služba Dosu 0x3F, čti ze souboru.

10.6.3 Čti ze souboru

Data ze souboru můžeme číst v blocích libovolné délky. Opakovaným voláním přečteme vždy bloky následující. (Vnitřní ukazatel pozice v souboru se po každém čtení zvýší o počet přečtených byte).

Vstup:	Výstup:
Čti ze souboru:	
AH = 0x3F	CF (příznak carry) = 0 pak,
BX = identifikační číslo souboru (handle)	AX = počet skutečně přečtených byte ze souboru (nula je konec souboru EOF)
DS:DX = ukazatel na místo, kam se zapíší přečtená data.	Nebo:
CX = kolik byte se bude max. číst	CF = 1, pak AX = kód chyby

Funkci musíme předat v registru BX identifikační číslo souboru (handle, file descriptor). Ukazatel na buffer (tam, kam budeme chtít data přečíst) předáme v registrech DS:DX (segment, offset) a nakonec v registru CX uvedeme max. počet byte, které budeme chtít najednou přečíst, většinou je tato hodnota shodná s velikostí (délkou) bufferu. Po návratu z volání bude v registru AX uložen počet skutečně přečtených byte ze souboru. Chyba bude indikována nastavením příznaku přenosu carry CF = 1. Pak se v registru AX předá kód chyby.

Přečtená data ze souboru bychom mohli vypsat na obrazovku pomocí funkce Dosu 0x09 vypiš řetězec. Problémy by nastaly se souborem, který by obsahoval znak '$'. Vypisování by se tak předčasně ukončilo. V kapitole věnované souborům v operačním systému jsme se zmínili o speciálních identifikačních číslech souboru pro standardní vstup a výstup. Problém můžeme elegantně vyřešit zápisem na standardní výstup (identifikační číslo souboru 0x0001).

10.6.4 Zapiš do souboru

Funkce Dosu pro zapsání souboru má stejné parametry jako funkce pro čtení souboru.

Vstup:	Výstup:
Zapiš do souboru:	
AH = 0x40	CF (příznak carry) = 0 pak,
BX = identifikační číslo souboru (handle)	AX = počet skutečně zapsaných byte do souboru
DS:DX = ukazatel na místo, odkud zapíší data.	Nebo:
CX = kolik byte se bude zapisovat	CF = 1, pak AX = kód chyby

Po návratu bez chyby je v registru AX uložen počet skutečně zapsaných byte do souboru. Liší-li se, byl pravděpodobně zaplněn disk.

Již máme dostatek informací, abychom mohli realizovat náš program vypisující textový soubor text.txt na obrazovku.

Nejprve budeme muset pomocí funkce Dosu 0x3D soubor text.txt otevřít. Funkcí Dosu 0x3F budeme ze souboru číst po blocích zvolené délky, dokud nebude počet přečtených byte nula. Stejný blok pomocí funkce Dosu 0x40 zapíšeme na standardní výstup. Po ukončení výpisu soubor zase uzavřeme.

Celý program bude vypadat následovně:

```
%define DELKA_B 80          ;zadefinujeme symbolickou konstantu
%define STDOUT 0x0001       ;speciální identifikační číslo souboru pro
                            ;standardní výstup.
SECTION .text
org 0x100

mov bp,STDOUT               ;speciální identifikační číslo souboru pro
                            ;standardní výstup.
                            ;prozatím ho uložíme v tomto registru, kde
                            ;se nám nebude plést.
mov ax, 0x3D00              ;služba Dosu otevři soubor, mód pro čtení
                            ;a zápis.
mov dx,jmeno_souboru        ;a předáme ukazatel na jméno souboru,
                            ;registr DS je již správně nastaven.
int 0x21                    ;vyvoláme službu Dosu
jc chyba                    ;chyba?, jestliže ano, skočíme na návěští
                            ;chyba.
mov bx,ax                   ;AX obsahuje identifikační číslo souboru
                            ;(handle) uložíme do registru BX
nacti_dalsi:
mov ah,0x3F                 ;služba Dosu čti ze souboru
mov dx,buffer               ;offset na začátek místa paměti určené
                            ;pro načtená data
mov cx,DELKA_B              ;do registru CX uložíme hodnotu symbolické
                            ;konstanty (makra)
int 0x21                    ;vyvoláme službu Dosu
jc chyba                    ;něco se nepodařilo, ihned skončíme
or ax,ax                    ;je registr AX nula? můžeme použít též cmp ax,0
jz konec_cteni              ;jestliže ano skok na návěští konec čtení.
mov cx,ax                   ;počet přečtených byte může být v posledním
                            ;kole čtení menší, proto budeme zapisovat
                            ;stejný počet byte, který jsme přečetli.
mov ah,0x40                 ;služba Dosu zapiš do souboru
xchg bp,bx                  ;identifikační číslo souboru dočasně uložíme
                            ;v registru BP a hodnotu v registru BP
                            ;uložíme do BX
int 0x21                    ;vyvoláme službu Dosu
```

```
xchg bp,bx                  ;prohodíme BX a BP zase zpět
jmps nacti_dalsi            ;skokem na nacti_dalsi zahájíme čtení
                            ;dalšího bloku
konec_cteni:
mov ah, 0x3E                ;služba Dosu uzavři soubor.
int 0x21                    ;vyvoláme službu Dosu
mov al,0                    ;návratový kód nastavíme na úspěch
konec:
mov ah,4Ch                  ;služba Dosu ukonči.
int 0x21                    ;vyvoláme službu DOSu, není návratu.
chyba:
mov al,1                    ;návratový kód nastavíme na 1
jmp short konec             ;skočíme na návěští konec.

SECTION .data
jmeno_souboru db "text.txt",0 ;nadefinujeme jméno souboru na text.txt
SECTION .bss
buffer RESB DELKA_B         ;nadefinujeme neinicializovanou proměnnou
                            ;délky DELKA_B (80)
```

Program nazveme vypis.com, přeložíme ho stejně jako u předchozích programů příkazem:

```
nasm vypis.asm -o vypis.com -f bin
```

Soubor text.txt musíme ještě vytvořit v textovém editoru (notepad) nebo rovnou z příkazové řádky příkazem.

```
echo ALL YOUR BASE ARE BELONG TO US > text.txt
```

Soubor bude obsahovat zadaný text a program vypis.com ho po spuštění vypíše na obrazovku.

Program vypis můžeme snadno změnit tak, aby soubory kopíroval. Místo speciálního identifikačního čísla pro standardní výstup by stačilo předat číslo souboru otevřeného (vytvořeného pro zápis). Tak bychom obsah jednoho souboru překopírovali do souboru jiného. Zatím ovšem nevíme, jak nový soubor vytvořit.

10.6.5 Inteligentní otevření/vytvoření souboru

Funkce Dosu 0x6C zajišťuje a ošetřuje otevření nebo případné utvoření souboru nového.

Vstup:	Výstup:
Inteligentní otevření/vytvoření souboru.	
AH = 0x6C	CF (příznak carry) = 0 pak,
AL = 0x00 (musí být vždy nula)	AX = identifikační číslo otevřeného souboru
BX = bitové pole pro rozšířený režim	CX = provedená akce
CX = atributy nově vzniklého souboru	Nebo:
DX = bitové pole akce	CF = 1, pak
DS:SI = ukazatel na název souboru	AX = kód chyby

Funkce Dosu 0x6C kombinuje celkem tři možné způsoby otevření souboru. Otevřít soubor, vytvořit soubor a otevřít a zkrátit na nulovou délku a otevřít.

V registru DX můžeme specifikovat akci, kterou funkce provede v závislosti na existenci souboru. Registr DX je bitové pole, jako parametr volání můžeme zvolit i více přípustných kombinací bitů.

Spodní 4 bity (registr DL)

DL = 0000b – skonči s chybou, jestliže soubor existuje

DL = 0001b – otevři, jestliže soubor existuje

DL = 0010b – zkrať soubor na nulovou délku (nahraď) a otevři, pokud existuje

Horní 4 bity (registr DH)

DH = 0000b skonči s chybou, jestliže soubor neexistuje

DH = 0001b vytvoř soubor, pokud neexistuje

Když bychom chtěli použít funkci 0x6C v programu kopírujícím soubory, pak bychom do registru DX uložili hodnotu 0x0012. Soubor by se vytvořil a otevřel, pokud by neexistoval, jinak by se zkrátil na nulovou délku a otevřel (nahradil).

V registru CX funkce požaduje předání atributů souboru. V operačním systému DOS nejsou rozlišeni ani vlastníci, ani přístupová práva k souboru, tak jak je známe z OS UNIX. Atributy v souboru v OS DOS určují některé jeho vlastnosti.

Předávaná hodnota v registru CX je opět bitové pole specifikující jednotlivé jednobitové atributy souboru:

bit 0: je-li nastaven, pak je soubor jen pro čtení (otevření pro čtení i zápis selže!)

bit 1: je-li nastaven, pak se jedná o soubor skrytý

bit 2: je-li nastaven, pak se jedná o soubor systémový

bit 3: je-li nastaven, pak se jedná o název média

bit 4: je-li nastaven, pak se jedná o adresář

bit 5: je-li nastaven, pak soubor NEBYL archivován

bity 15-6: jsou rezervované pro budoucí použití

Při vytváření souboru budeme nejčastěji předávat hodnotu CX = 0x20, vytvoř obyčejný (archivní) soubor.

V registru BX musíme předat příznaky rozšířeného režimu. My se spokojíme pouze se třemi známými hodnotami. Ostatní příznaky značně přesahují hloubku probíraného tématu.

BX = 0 otevřít pro čtení a zápis

BX = 1 otevřít pouze pro zápis

BX = 2 otevřít pro čtení a zápis

Po úspěšném návratu funkce (CF = 0) je v registru CX uložena použitá akce a v registru AX je uloženo identifikační číslo souboru (stejně tak jako při návratu z funkce 0x3D otevři soubor).

CX = 1 soubor byl otevřen

CX = 2 soubor byl vytvořen a otevřen

CX = 3 soubor byl nahrazen a otevřen

Program vypis.asm bychom mohli změnit na kopírování obsahu dvou souborů snadno. Pomocí funkce 0x6C bychom vytvořili (přepsali) cílový soubor. Zbývá jen předat identifikační číslo takto otevřeného souboru na místo volání funkce zápisu do souboru. Připišme na začátek programu instrukce otevírající druhý soubor.

```
mov ax, 6C00h       ;funkce Dosu inteligentní otevření souboru
mov cx, 0x20        ;atribut nově vzniklého souboru
mov dx, 0x12        ;požadovaná akce
mov si, jmeno_druheho_souboru      ;offset na řetězec specifikující
                                   ;jméno souboru
int 0x21            ;vyvolání služby Dosu
jc chyba            ;skok na návěští chyba, jestliže volání skončilo s
                    ;chybou.
mov bp,ax           ;identifikační číslo souboru uložíme do registru BP.
```

Samozřejmě odstraníme původní instrukci mov bp,STDOUT. Druhý soubor musíme uzavřít. Kód můžeme umístit hned za uzavření souboru prvního.

```
mov ah,0x3e ;uzavři soubor
mov bx,bp
int 0x21
```

Nakonec do datové oblasti připíšeme název druhého souboru.

```
jmeno_druheho_souboru db "text1.txt",0
                      ;nadefinujeme druhé jméno souboru
```

Program bude první soubor text.txt kopírovat do souboru druhého. Číst budeme z prvního souboru, identifikační číslo souboru bude uloženo v registru BX. Zapisovat do druhého souboru budeme pomocí jeho identifikačního čísla, které si po otevření uložíme do registru BP.

Program pojmenujeme copy.asm a přeložíme jako obvykle (nasm -f bin -o copy.com copy.asm).

10.6.6 Změň polohu v souboru (SEEK)

Doposud jsme se soubory pracovali sekvenčně, zapisovali jsme a četli bloky v pořadí, v jakém jsou v souboru uloženy. V souborech se složitější strukturou se často budeme muset pohybovat tam i zpět. Funkce, která nastaví vnitřní ukazatel pozice v souboru, se jmenuje SEEK a je také implementována v operačním systému DOS.

Vstup:	Výstup:
Změna pozice v souboru AH = 0x42 AL = typ změny BX = identifikační číslo soubor (handle) CX:DX = pozice	CF (příznak carry) = 0, pak DX:AX nová pozice v souboru CX = provedená akce Nebo: CF = 1, pak AX = kód chyby

Funkce požaduje v registru BX identifikační číslo souboru, které nám vrátila funkce otevři soubor. V párovém registru CX:DX předáváme novou pozici. Abychom mohli pracovat až se 4GB soubory (skutečný limit 2 GB u FAT16), pozici předáváme v dvojici šestnáctibitových registrů. Registr CX obsahuje horních šestnáct bitů pozice a registr DX zase šestnáct bitů spodních. Hodnota v registru AL určuje, k jaké výchozí pozici se nová pozice přičte.

AL = 0 pozice v CX:DX se přičte k začátku souboru, nová pozice bude určena pouze registrovým párem CX:DX. (SEEK_SET)

AL = 1 předávaná pozice se přičte k pozici stávající (SEEK_CUR)

AL = 2 předávaná pozice se přičte k pozici určené koncem souboru (SEEK_END)

Funkce pro kontrolu předá výslednou pozici v párovém registru DX:AX.

Funkci seek můžeme použít ke zjištění velikosti souboru. Soubor otevřeme a funkci seek (0x42) zavoláme s těmito parametry.

```
mov ax,0x4202        ;nastav pozici od konce souboru
mov bx,...           ;do bx ulož identifikační číslo souboru
xor dx,dx            ;vynuluj DX
xor cx,cx            ;vynuluj CX
int 0x21             ;zavolej službu Dosu
```

V párovém registru DX:AX se nám vrátí délka souboru, protože jsme ke konci souboru přičetli nulovou pozici.

10.6.7 Ostatní funkce pro práci se soubory

Seznámíme se se zbytkem operací nad souborem: jeho výmazem, změnou atributů a přejmenováním.

Soubory můžeme z disku smazat pomocí funkce Dosu 0x41 vymaž soubor.

Vstup:	Výstup:
Vymaž soubor AH = 0x41 DS:DX = ukazatel na jméno souboru	CF (příznak carry) = 0, pak nic Nebo: CF = 1, pak AX = kód chyby

Má-li soubor nastaven atribut pouze pro čtení (read-only), musíme nejprve atribut odstranit a pak smazat. Provést změnu atributů souboru lze pomocí systémové služby zjisti/nastav atributy.

Vstup:	Výstup:
Zjisti atributy souboru AH = 0x43 AL = 0x00 DS:DX = ukazatel na jméno souboru Nebo: Nastav atributy souboru CX = požadované atributy AL= 0x01 DS:DX = ukazatel na jméno souboru	 CF (příznak carry) = 0, pak nic CX = atributy souboru Nebo: CF = 1, pak AX = kód chyby

Přejmenovat soubory můžeme pomocí systémového volání 0x56 – přejmenuj soubor. Přejmenováním můžeme soubor umístit i do jiného adresáře. Cílový adresář musí být umístěn na stejném logickém disku jako původní soubor.

Vstup:	Výstup:
Přejmenuj soubor AH = 0x56 DS:DX = ukazatel na původní jméno souboru ES:DI = ukazatel na nové jméno souboru	 CF (příznak carry) = 0, pak nic Nebo: CF = 1, pak AX = kód chyby

Ukažme si příklad.

Př.: Přesuňte soubor text.txt ležící v aktuálním adresáři do kořenového adresáře disku.

Použijeme funkci přejmenuj soubor. Program bude jednoduchý, bude obsahovat dvě systémová volání (přejmenuj a skonči).

```
SECTION .text          ;programová sekce
org 0x100
mov ah, 0x56           ;funkce Dosu 0x56 přejmenuj soubor
mov dx,puvodni         ;do DX adresa řetězce s původním jménem
mov di,novy            ;a s novým
int 0x21
mov ax,0x4c00          ;ukončíme program
int 0x21
SECTION .data
puvodni db "text.txt",0
novy db "\text.txt",0
```

Program nazveme rename.asm a přeložíme ho obvyklým způsobem. Jeho velikost bude jen pár byte, ale dala by se ještě snížit. Až na lomítko se oba soubory nijak neliší, proto můžeme datovou sekci uspořádat i takto a dostaneme stejný výsledek:

```
SECTION .data
novy db "\"
puvodni db "text.txt",0
```

Oba názvy souborů sdílí stejnou paměť. Konstrukce tohoto typu si ve vyšších programovacích jazycích nemůžeme dovolit.

10.6.8 Práce s dlouhými názvy souborů

S příchodem operačního systému Windows 95 bylo zrušeno omezení max. 8+3znakové názvu souborů. V rámci zachování kompatibility se staršími programy se dlouhý název souboru na disk uloží i pod zkráceným klasickým 8+3názvem. Např. adresář s názvem LinuxRulez se zobrazí na starý název jako LINUXR~1. Chceme-li pracovat s dlouhými názvy souborů (až 256 znaků), musíme použít nová systémová volání, která jsou k dispozici pouze za běhu operačního systému Windows. Jejich popis naleznete na přiloženém CD v Ralf's Brown Interrupt List. Všechny služby pro práci s dlouhými názvy souborů jsou zahrnuty pod službu Dosu s číslem 0x70. V registru AL se předává číslo služby původní (krátké). Služby, které předávaly své parametry v registru AL, nejsou podporovány. Příkladem je funkce Dosu otevři soubor AH = 0x3D. Chceme-li použít dlouhé názvy souborů, budeme muset použít funkci 0x6C (inteligentní otevření/vytvoření souboru), registr AX bude obsahovat 0x706C, ostatní parametry jsou shodné s původní variantou.

10.7 Práce s adresáři

Práce s adresáři v Assembleru se nijak neliší od práce s adresáři ve vyšších programovacích jazycích. Známé funkce (MKDIR, RMDIR, CHDIR...) jsou implementovány pomocí sady systémových volání, které si popíšeme.

10.7.1 Vytvoř a zruš adresář (MKDIR, RMDIR)

Vytvořit adresář můžeme pomocí funkce MKDIR, která očekává jako svůj jediný parametr ukazatel na jméno vytvářeného adresáře.

Vstup:	Výstup:
Vytvoř adresář AH = 0x39 Zruš adresář AH = 0x3A DS:DX = ukazatel na jméno adresáře	 CF (příznak carry) = 0, pak nic Nebo: CF = 1, pak AX = kód chyby

Naopak adresář zrušíme zavoláním funkce Dosu RMDIR. Z adresáře musíme předtím ovšem odstranit všechny soubory.

10.7.2 Vyber aktuální adresář (CHDIR)

Užitečnou funkcí Dosu je funkce vyber adresář (CHDIR). Funkce změní aktuální adresář běžícího programu. Jiné funkce Dosu očekávající jméno souboru budou soubor hledat v právě aktuálním adresáři, nebude-li uvedeno jinak. (Pokud zadáme funkci Dosu otevři soubor plnou cestu k souboru, nebude mít samozřejmě výběr aktuálního adresáře žádný vliv.)

Vstup:	Výstup:
Vyber aktuální adresář AH = 0x3B DS:DX = ukazatel na jméno adresáře	CF (příznak carry) = 0, pak nic Nebo: CF = 1, pak AX = kód chyby

10.7.3 Zjisti vybraný adresář (GETCWD)

Často potřebujeme zjistit, ve kterém adresáři se nacházíme. Funkce Dosu 0x47 nám vrátí aktuální adresář, ovšem bez označení disku a prvního lomítka.

Vstup:	Výstup:
Zjisti aktuální adresář AH = 0x47 DL = na kterém disku DL = 0x00 aktuální DL = 0x01 na disku A ... DS:SI = ukazatel na volnou paměť (buffer), kam se uloží právě používaný adresář.	CF (příznak carry) = 0, pak nic Nebo: CF = 1, pak AX = kód chyby

Ukažme si implementaci programu pwd, který vypíše na obrazovku aktuální adresář i s označením disku. Ke zjištění aktuálního disku budeme potřebovat funkci Dosu zjisti aktuální disk.

Vstup:	Výstup:
Zjisti aktuální disk AH = 0x19	AL = číslo právě používaného disku AL = 0x00 disk A: AL = 0x01 disk B: ...

Výstupní buffer si připravíme ve tvaru:

```
buffer db "A:\",0
```

Pak ke změně písmena disku postačí přičtení výstupní hodnoty v registru AL (po volání funkce zjisti aktuální disk). Adresa (offset) předávaná funkci zjisti aktuální adresář bude o 3 byte vyšší, tak aby zapisovala přímo za lomítko. Cesta k souboru je v Dosu

omezena jen na 64 znaků. Musíme počítat i s nejhorší alternativou a pro všech 64 znaků vyhradit místo. Nakonec bufferu zapíšeme speciální znaky 0x0D, 0x0A, abychom odřádkovali.

Celý program pwd.asm bude vypadat takto:

```
SECTION .text
org 0x100
mov ah,0x19                ;služba Dosu zjisti vybraný disk
int 0x21                   ;zavolej službu Dosu
add byte [buffer],al       ;přičti výslednou hodnotu
                           ;(AL = 0x00 A:) k znaku 'A',
                           ;tak dostaneme písmena ostatních disků
xor dl,dl                  ;ptáme se na adresář z aktuálního disku
mov ah,0x47                ;služba Dosu zjisti aktuální adresář
mov si,buffer+3            ;začni zapisovat na domluveném místě
int 0x21                   ;zavolej službu Dosu
mov ah,0x40                ;vypiš vše na obrazovku
mov bx,0x0001              ;identifikační číslo standardního výstupu
mov cx,BUFF_LEN            ;kolik znaků se má vypsat
mov dx,buffer
int 0x21                   ;zavolej službu Dosu
mov ax,0x4c00              ;ukonči program
int 0x21                   ;zavolej službu Dosu
SECTION .data
buffer db "A:\"
times 64 db " "            ;zbytek bufferu vyplň mezerami
db 0x0D,0x0A               ;a nakonec odřádkuj

BUFF_LEN equ $-buffer

    ;Stejný trik, jako jsme už použili, operátor $ zastupuje
    ;aktuální adresu, tj. adresu konce bufferu; odečtená od
    ;adresy(offsetu) jeho začátku dohromady určuje jeho délku
```

Pro úplnost se seznamme ještě s opačnou funkcí, vyber aktuální disk.

Vstup:	Výstup:
Vyber aktuální disk AH = 0x0e DL = 0x00 A: DL = 0x01 B: ...	AL celkový počet diskových jednotek

10.8 Alokujeme paměť

Každý program spuštěný pod operačním systémem DOS má veškerou zbývající volnou paměť přidělenou. Budeme-li chtít pracovat s pamětí hospodárně a přenechat její sprá-

vu operačnímu systému, musíme nejprve pomocí funkce 0x4A (změň velikost bloku paměti) zbývající paměť prohlásit za volnou.

10.8.1 Změň velikost bloku paměti

Změnit velikost bloku paměti lze pomocí služby Dosu 0x4A. Blok můžeme zvětšit i zmenšit podle potřeby. Neprojde-li požadavek, nastaví se příznak carry CF = 1 a v BX se vrátí maximální možná velikost bloku paměti.

Vstup:	Výstup:
Vyber aktuální disk AH = 0x0e DL = 0x00 A: DL = 0x01 B: ...	AL celkový počet diskových jednotek

Velikosti bloků paměti se udávají v paragrafech. Jeden paragraf je 16 byte.

V programech typu COM se klasicky změní velikost přidělené paměti na 64 Kb. (0x1000 paragrafů). Provést změnu můžeme pomocí následujícího kódu:

```
mov ah, 0x4A        ;služba Dosu změň velikost bloku paměti
mov bx, 0x1000      ;kolik paragrafů žádáme, registr ES je již nastaven
int 0x21            ;vyvoláme službu Dosu
jc chyba
```

O další bloky paměti pak můžeme žádat pomocí funkce Alokuj paměť.

10.8.2 Alokuj paměť

Operační systém můžeme kdykoli požádat o nový blok paměti, bude-li k dispozici, bude nám přidělen. Jeho segmentová adresa bude předána v registru AX, přidělená paměť pak začíná na adrese AX:0.

Vstup:	Výstup:
Alokuj nový blok paměti. AH = 0x48 BX = žádaná velikost v paragrafech	CF (příznak carry) = 0, pak nic AX = segmentová adresa přiděleného bloku Nebo: CF = 1, pak AX = kód chyby BX = velikost největšího možného bloku (v paragrafech)

Ukažme si příklad alokace nového 64 Kb bloku paměti. Předpokládejme, že jsme již operačnímu systému nějakou paměť vrátili nazpět (funkcí 0x4A).

```
mov ah,0x48         ;služba Dosu alokuj paměť
mov bx,0x1000       ;žádáme o 64 Kb
int 0x21            ;zavoláme DOS
jc chyba            ;jestli CF = 1, paměť jsme nedostali, v BX je max.
```

```
                    ;počet volných paragrafů
push es             ;na zásobník uložíme původní obsah registru ES
mov es,ax           ;do ES nahrajeme získanou segmentovou adresu
```

Nová paměť přidělená programu je pak dostupná přes segmentový registr ES. V tuto chvíli budeme muset brát ohled na obsah segmentových registrů a podle potřeby jejich hodnoty měnit tak, abychom přistupovali k „proměnným" uloženým v různých segmentech pomocí správného segmentového registru.

10.8.3 Uvolni paměť

Paměť, kterou jsme alokovali, můžeme uvolnit a tak navrátit zpět operačnímu systému.

Vstup:	Výstup:
Uvolni blok paměti. AH = 0x49 ES = segmentová adresa uvolňovaného bloku	CF (příznak carry) = 0, pak nic Nebo: CF = 1, pak AX = kód chyby

10.9 Předávání příkazové řádky

Přístup k parametrům programu je v jazyce C možný přes známé proměnné argv (pole jednotlivých parametrů) a argc. (počet parametrů). Operační systém DOS předá parametry spuštěného programu jen jako řetězec, který je uložen od offsetu 0x81 od začátku programového segmentu. Délka řetězce je uložena na offsetu 0x80 (jeden byte) a je do ní započítán i ukončovací znak 0x0D (viz podkapitola „Druhy spustitelných programů"). Jejich separaci si stejně jako knihovna jazyka C musíme zajistit sami.

Ukažme si program který pouze vypíše svoje parametry na obrazovku a ukončí se. S podobným programem jsme se již setkali, řetězec ležící na adrese CS:0x81 budeme tisknout na standardní výstup.

```
SECTION .text       ;začátek kódové sekce
%define STDOUT 1    ;standardní výstup
org 0x100           ;COM začíná od adresy 0x100
mov ah,0x40         ;služba Dosu zapiš soubor
mov bx,STDOUT
mov dx,0x81         ;odkud budeme zapisovat
xor cx,cx           ;vynulujeme délku (registr CX)
mov cl,[0x80]       ;do registru CL nahrajeme délku parametrů (řetězce)
mov di,cx           ;okopírujeme do DI
add di,dx           ;přičteme délku a tak se dostaneme na konec [0x0D]
inc di              ;DI nastavíme za poslední byte
mov byte [di],0xA   ;zapíšeme 0x0A, abychom odřádkovali
inc cx              ;a zvětšíme délku řetězce o 1
int 0x21            ;vypíšeme na obrazovku
mov ax,0x4c00       ;služba Dosu skonči program, návratový kód 0
int 0x21            ;ukončíme program
```

10.10 Chybové kódy

Návrat s chybou ze systémového volání je indikován nastaveným příznakem CF (carry). Registr AX pak obsahuje chybový kód. Podrobnější informace o chybě (a navržené akce) lze získat zavoláním funkce AH=0x59. Zvídavého čtenáře bych opět odkázal na elektronickou příručku Ralf's Brown Interrupt List na přiloženém CD.

V tabulce jsou uvedeny nejčastější chybové kódy.

Kód	Popis chyby
0x0002	Soubor nenalezen
0x0003	Cesta nenalezena
0x0005	Přístup znemožněn
0x0008	Nepostačující paměť
0x0009	Neplatná adresa bloku paměti
0x000f	Specifikován neplatný disk
0x0010	Adresář nelze odstranit
0x0011	Nestejné médium

10.11 Odlaďujeme

Žádný program napsaný v Assembleru nefunguje napoprvé nikdy podle představ autora. Abychom se nemuseli spoléhat jen na kontrolní výpisy programu (které bychom si stejně museli naprogramovat), můžeme použít speciálních odlaďovacích programů.

Anglický výraz pro chybu bug (česky brouk) vznikl kuriózně. Technici obsluhující sálový počítač nemohli dlouho objevit příčinu poruch počítače. V jednom hardwarovém modulu našli broučí rodinku, která svoji blíže nespecifikovanou činností ovlivňovala jeho chod.

Podle brouků dostaly název i speciální programy – debuggery – navržené pro odladění programů. Jejich prostřednictvím můžeme krok za krokem sledovat (dle potřeby i měnit) provádění jednotlivých instrukcí sledovaného (odlaďovaného) programu, zjišťovat stav registrů mikroprocesoru nebo nahlížet do paměti patřící odlaďovanému programu.

Standardní součástí operačního systému DOS (a Windows) je textově orientovaný debugger debug.exe, se kterým se seznámíme podrobněji v následující kapitole. Pokročilým uživatelům bych doporučil Turbo Debugger firmy Borland nebo IDA – Interactive Disassembler firmy DataRescue.

Popišme si obecně jednotlivé funkce debuggeru tak, abychom mohli používat i programy, se kterými jsme se zde neseznámili.

Po spuštění odlaďovacího programu musíme odlaďovaný program nejprve nahrát do paměti (příkazy open, load). Vyžaduje-li program parametry příkazové řádky, předáme je prostřednictvím debuggeru právě nahranému programu (set program arguments, program parameters apod.). Používáme-li nějaký graficky orientovaný debugger, uvidíme na obrazovce většinou okno s dekódovanými instrukcemi ze strojového kódu, stav procesoru (hodnoty jednotlivých registrů) a okno s výpisem paměti (často i s výpisem zá-

sobníku) patřící odladovanému programu. Máme-li k dispozici jen textově orientovaný debugger, vhodným příkazem zobrazíme informace, které potřebujeme.

Nejdůležitější funkcí každého debuggeru je krokování programu. Příkaz pro provedení jedné instrukce odladovaného programu se jmenuje step (z anglického slova pro krok). Po provedení jednoho kroku programu jsou vypsány nové hodnoty registrů mikroprocesoru včetně instrukce, která se bude provádět jako další.

Příkazem step-over lze přeskakovat krokování podprogramů volaných instrukcí call (tak se provede podprogram, aniž by se krokovaly jeho jednotlivé instrukce).

Chceme-li nechat program pokračovat, použijeme příkaz pokračuj (continue, go).

Všechny debuggery umí pracovat s tzv. breakpointy. Breakpoint přeruší provádění programu a vrátí řízení zpět debuggeru, je-li splněna určitá podmínka. Podmínkou může být například provedení instrukce z určité adresy, změna dat v paměti apod.

Odladovaný program je přeložen ve strojovém kódu počítače, nejsou tedy k dispozici symbolická jména identifikátorů (návěští, jména proměnných). Uvidíme jen jejich skutečné adresy (některé souborové formáty (EXE nebo ELF) dovolují přibalení tzv. symbolů – symbolických jmen do spustitelného souboru a tak ulehčí uživateli debuggeru orientaci v odladovaném programu).

Seznamme se se snadno dostupným debuggerem, kterým je standardně vybaven OS DOS (Windows). Existuje i jeho volně šiřitelná implementace, nazývaná grdb, kterou naleznete na přiloženém CD. Původní program debug.exe rozšiřuje o podporu instrukcí 80386, což budeme potřebovat.

10.12 Debugger grdb.exe

Program byl napsán v Assembleru a je šířen jako freeware. Uživatelským prostředím je příkazová řádka.

Debugger spustíme příkazem grdb.

```
C:\>grdb
GRDB version 3.6 Copyright (c) LADsoft 1997-2002

History enabled
eax:00000000 ebx:00000000 ecx:00000000 edx:00000000 esi:00000000
edi:00000000 ebp:00000000 esp:0000FFEE eip:00000100 eflags:00000202
NV UP EI PL NZ NA PO NC
ds: 10FB es:10FB fs:10FB gs:10FB ss:10FB cs:10FB
10FB:0100 74 1E      jz      0120

->
```

Ukažme si krokování programu pwd.com, který na standardní výstup vypíše aktuální adresář. Program pwd.com předpokládáme v aktuálním adresáři. Příkazem **l pwd.com** nahrajeme program pwd.com do paměti (parametry odladovaného programu můžeme uvést za jméno souboru).

```
->l pwd.com
Size: 00000069
->
```

Na obrazovce se objeví prompt (->), program je připraven pro příjem našich příkazů. Jednu instrukci (krok, step) provedeme příkazem **t**. Napíšeme **t** a stiskneme klávesu Enter. Na obrazovce naskočí:

```
->t

eax:00001900 ebx:00000000 ecx:00000069 edx:00000000 esi:00000000
edi:00000000 ebp:00000000 esp:0000FFEE eip:00000102 eflags:00000202
NV UP EI PL NZ NA PO NC
ds: 10FB es:10FB fs:10FB gs:10FB ss:10FB cs:10FB
10FB:0102 CD 21      int     21

->
```

Příkazem **t** se vykonala v našem programu první instrukce. Do registru AH jsme vložili hodnotu 0x19. Následující instrukce, která se bude provádět, je INT 21, volání operačního systému. První údaj na posledním řádku je její adresa (tak je nastavena i dvojice registrů CS:IP), 0xCD 0x21 je strojová reprezentace instrukce INT 21. Po stisku klávesy t a Enter se instrukce int 0x21 provede a v registru AL bychom měli obdržet číslo právě aktuálního disku.

```
->t

eax:00001902 ebx:00000000 ecx:00000069 edx:00000000 esi:00000000
edi:00000000 ebp:00000000 esp:0000FFEE eip:00000104 eflags:00000202
NV UP EI PL NZ NA PO NC
ds: 10FB es:10FB fs:10FB gs:10FB ss:10FB cs:10FB
10FB:0104 00 06 24 01  add     [0124],al        ds:[0124]=41

->
```

V registru AL je uložena hodnota 02, která reprezentuje disk C:.

Další připravenou instrukcí je instrukce ADD [buffer],al, která změní písmeno disku z A na skutečně aktuální (přičte k znaku 'A' hodnotu z AL).

Instrukce ADD [0124],AL je skutečně instrukce z našeho programu, ovšem bez symbolické adresy „proměnné" buffer. Z instrukce můžeme vyčíst, že symbolická proměnná buffer je uložena na adrese (offsetu) 0x124. Příkazem **d 124** můžeme oblast paměti vypsat. Nejčastěji používaným parametrem příkazu d je adresa, od které se data budou vypisovat (jako parametr příkazu d může být i jméno registru).

```
->d 124
10FB:0120           -41 3A 5C 20-20 20 20 20-20 20 20 20   A:\
10FB:0130 20 20 20 20-20 20 20 20-20 20 20 20-20 20 20 20
10FB:0140 20 20 20 20-20 20 20 20-20 20 20 20-20 20 20 20
10FB:0150 20 20 20 20-20 20 20 20-20 20 20 20-20 20 20 20
10FB:0160 20 20 20 20-20 20 20 0D-0A C7 06 8C-CD 01 00 EB     ........
10FB:0170 60 26 8A 47-01 32 E4 40-D1 E0 03 D8-26 8A 0F 32 `&.G.2.@....&..2
10FB:0180 ED 0B C9 74-0F 43 53 26-8B 1F E8 C6-00 5B 73 41 ...t.CS&.....[sA
10FB:0190 43 43 E2 F2-2E C7 06 8C-CD 03 00 EB-34 26 8A 47 CC..........4&.G
10FB:01A0 01 32 E4 40-D1 E0 03 D8-26 8A 07 32-E4 D1 E0 40 .2.@....&..2...@
->
```

Po vykonání instrukce ADD, po opětovném provedení příkazu **t** se řetěz „A:\" přepíše na „C:\", protože aktuální disk je C: (AL=0x02). Přesvědčit se můžeme znovu pomocí příkazu d (vypiš adresu).

```
->d 124
10FB:0120           -43 3A 5C 20-20 20 20 20-20 20 20 20   C:\
```

Další řádky výpisu neuvádíme, protože nepřinesly nic nového.

Zbytek programu můžeme buď krokovat (příkazem **t**) anebo nechat program dokončit příkazem **g**, který se vrátí do příkazové řádky debuggeru až po skončení odlaďovaného programu.

Příkazem **a** můžeme v programu modifikovat instrukce i data. Příkaz vyžaduje jako svůj parametr adresu (offset), kde chceme zadat novou instrukci. Pro začátečníka bude lepší, pokud chybu v programu opraví ve zdrojovém kódu, program znovu přeloží a krokuje v debuggeru.

Při krokování programu můžeme příkazem **p** přeskakovat podprogramy (step over). Výpis kódu programu získáme příkazem **u**, jehož parametrem je adresa začátku výpisu.

```
->u 100
10FB:0100 B4 19      mov     ah,0019
10FB:0102 CD 21      int     21
10FB:0104 00 06 24 01  add     [0124],al
10FB:0108 30 D2      xor     dl,dl
10FB:010A B4 47      mov     ah,0047
10FB:010C BE 27 01     mov     si,0127
10FB:010F CD 21      int     21
10FB:0111 B4 40      mov     ah,0040
10FB:0113 BB 01 00     mov     bx,0001
10FB:0116 B9 45 00     mov     cx,0045
10FB:0119 BA 24 01     mov     dx,0124
10FB:011C CD 21      int     21
10FB:011E B8 00 4C     mov     ax,4C00

->
```

Odladovací program ukončíme příkazem q.

Příkaz	Význam
l <filename> [p]	Nahraje soubor pojmenovaný filename do debuggeru, nepovinně lze uvést parametry p.
t	Provede jednu instrukci (nebo lze nepovinně uvést počet instrukcí, které se budou provádět). Místo příkazu step stačí napsat příkaz s.
p	Krokuje instrukce podobně jako step, vykonávání podprogramů přeskočí. Nepovinným parametrem n můžeme uvést počet kroků.
u	Převede instrukce ze strojového kódu zpět do Assembleru. Nejčastějším parametrem je adresa, od které si budeme přát program vypisovat.
g	Dokončí prováděný program.
a <adr>	Zapisuj nové instrukce od adresy adr .
?	Příkazem ? získáme seznam všech dostupných příkazů s krátkou nápovědou.
b <num>,adr	Nastaví breakpoint (zarážku) číslo num na adresu adr (num smí být z intervalu 0..F).
b	Vypíše všechny breakpointy.
q	Ukončíme práci s debuggerem. (alias q).

10.13 Návrh a odladění programu

Navrhněme společně program, který vypíše na obrazovku součet dvou (desítkových) čísel načtených z příkazové řádky a oddělených mezerou (i několika).

Pro převody řetězců na čísla (a naopak) můžeme použít konverzní podprogramy z kapitoly užitečných kousků kódu. Výpis na obrazovku též zvládneme. Ukažme si celý postup návrhu programu včetně odladění i ladění v debuggeru.

Připomeňme si konverzní podprogram z řetězce na číslo.

```
;-------------------------------------------------------------------------
; ASCIIToNum
;-------------------------------------------------------------------------
; esi = pointer, odkud se převádí do té doby, než je nalezen znak 0x0
; ecx = základ soustavy
;Vrací:
; eax = výsledek
```

Pracujeme v OS DOS, kde jsme nuceni užít jen šestnáctibitovou adresaci, proto pointer (jen offset) předáme v registru SI a ne ESI.

Parametry programu musíme zpracovat. Utvořme podprogram, který přeskočí libovolný počet mezer. Jeho vstupem bude ukazatel na řetězec, výstupem ukazatel na řetězec začínající jiným znakem než mezera.

```
;-------------------------------------------------------------------------
; SkipSpace
;-------------------------------------------------------------------------
; si = pointer na řetězec
```

```
;
;Vrací:
; si první znak různý od mezery

.znovu:
lodsb                ;načti do AL z DS:SI
cmp al,' '           ;porovnej s mezerou
jz znovu             ;je-li mezera, porovnej znovu
ret
```

Parametry programu dostaneme na adrese (offsetu) 0x81, řetězec bude ukončen znakem 0x0D. Převeďme první číslo z řetězce na číslo. Podprogram ASCIIToNum očekává řetězec čísla ukončený (binární) nulou, proto budeme muset za první číslo (tj. místo první mezery) zapsat nulový byte. Napišme ještě jeden podprogram, který přeskočí řetězec číslic a zastaví se ihned za číslem, ukazatel vrátí v registru DI.

```
;-------------------------------------------------------------------------
; SkipNum
;vstup SI ukazatel (offset) řetězec
;výstup: DI ukazatel za číslo v řetězci
;-------------------------------------------------------------------------
SkipNum:
mov di,si                    ;jako ukazatel budeme používat registr DI
.najdi_konec_cisla:
inc di                       ;DI=DI+1
cmp [di],'0'                 ;jedná se o znak s ASCII kódem menší než
                             ;znaku '0'?
jb .mam_konec                ;pak jsme hotovi
cmp [di],'9'                 ;jedná se o znak s kódem vyšším než '0'?
ja .mam_konec                ;ano? skončíme
jmp .najdi_konec_cisla       ;porovnáme další znak
.mam_konec
ret
```

Tělo hlavního programu by mohlo vypadat takto:

```
mov si, 0x81         ;do SI umístíme ukazatel na začátek řetězce
call SkipNum         ;přeskočíme na konec čísla
mov [di],0           ;zapíšeme nulový byte
mov ecx,10           ;první číslo převedeme
call ASCIIToNum      ;v EAX dostaneme výsledek, registr SI je nezměněn
mov edx,eax          ;číslo uložíme do EDX
mov si,di            ;načteme do registru SI ukazatel na řetězec těsně za
                     ;prvním
inc si               ;číslem
call Skipspace       ;přeskočíme mezery
call SkipNum         ;i číslo
mov [di],0           ;na konec druhého čísla zapíšeme nulu
call ASCIIToNum      ;v EAX dostaneme výsledek, registr SI je nezměněn
add eax,edx          ;sečteme s číslem předchozím
```

Nejprve přeskočíme číslo a zapíšeme za něj nulový byte. Potom číslo převedeme. Po převodu se musíme přesunout za konec čísla, který máme uložen v registru DI, za číslem bude určitě nulový byte, a proto budeme přičítat jedničku. Funkce pro převod znaků na číslo očekává ukazatel v registru SI, proto musíme ukazatel uložený v registru DI překopírovat do SI a přičíst jedničku. Před druhým číslem zase přeskočíme mezery. Pak přeskočíme číslo a zapíšeme za něj nulový byte. Nakonec zavoláme podruhé převodní funkci a čísla pomocí instrukce ADD sečteme.

Číslo uložené v registru EAX musíme opět převést na řetězec a nakonec vytisknout. Použijeme opačnou konverzní funkci NumToASCII:

```
;-------------------------------------------------------------------------
; NumToASCII
;-------------------------------------------------------------------------
;
; eax = 32 - bit číslo
; ebx = základ soustavy
; edi = pointer na buffer, kam se zapíše
;Vrací:
;naplněný buffer
;-------------------------------------------------------------------------
```

```
mov ebx,10          ;uložíme do EBX základ soustavy
mov di, buffer      ;do DI adresu, kam zapíšeme
call NumToASCII     ;zavoláme podprogram, v EAX očekáváme převáděné číslo
```

Převedené číslo vypíšeme na obrazovku, použijeme známou funkci zapiš do souboru. Problémem zůstává délka vypisovaného čísla, tu musíme zjistit programově (dynamicky) nebo utvořit dostatečně velký prostor a ten celý vytisknout i s přebývajícími mezerami. Později můžeme program modifikovat tak, aby vypsal vždy jen celý řetězec bez následujících mezer.

```
mov ah,0x40         ;funkce Dosu zapiš soubor
mov dx,di           ;ukazatel je vyžadován v DS:DX
mov cx,25           ;vypisovaný počet
int 0x21
mov ax, 0x4c00      ;skončíme
int 0x21
```

V datové sekci připravíme buffer (místo)

```
buffer times 25 db ' '
```

Zbývá zapsat začátky programových sekcí.

```
SECTION .text
org 0x100

mov si, 0x81        ;na této adrese začínají parametry programu
call SkipNum        ;přeskočíme číslo
mov [di],0          ;za číslo zapíšeme nulový byte
```

```
mov ecx,10          ;základ soustavy
call ASCIIToNum     ;v EAX dostaneme výsledek, registr SI je nezměněn
mov edx,eax         ;číslo uložíme dočasně do registru EDX
mov si,di           ;do SI uložíme ukazatel za číslo
inc si              ;i za nulový byte
call Skipspace      ;přeskočíme všechny mezery
call SkipNum        ; i číslo
mov [di],0          ;zapíšeme za něj nulový byte
call ASCIIToNum     ;v EAX dostaneme výsledek, registr SI je nezměněn
add eax,edx         ;sečteme s EDX
mov ebx,10          ;funkce vyžaduje převod soustavy, tj. víme, že bude 10
mov di, buffer      ;tam, kam zapíšeme řetězec znaků,
call NumToASCII     ;převedeme obsah registru EAX na číslo

mov ah,0x40         ;služba Dosu zapiš soubor
mov dx,di           ;ukazatel už máme v DI, vyžadujeme v DX
mov cx,25           ;počet znaků, co budeme tisknout
int 0x21            ;tiskneme
mov ax, 0x4c00      ;ukončíme program
int 0x21

;-------------------------------------------------------------------------
; SkipSpace
;-------------------------------------------------------------------------
; si = pointer na řetězec
;
;Vrací:
; si, první znak různý od mezery
SkipSpace:
.znovu:
lodsb               ;načteme znak do AL z DS:SI
cmp al,' '          ;porovnáme s mezerou
jz znovu            ;je-li mezera, načteme další
ret
;-------------------------------------------------------------------------
; SkipNum
;vstup SI ukazatel (offset) řetězec
;výstup: DI ukazatel za číslo v řetězci
;-------------------------------------------------------------------------
SkipNum:
mov di,si           ;do DI vložíme hodnotu z registru SI
.najdi_konec_cisla
inc di              ;DI = DI + 1
cmp [di],'0'        ;porovnáme s nejnižším možným číslem
jb .mam_konec       ;je-li hodnota menší, jsme na konci čísla
cmp [di],'9'
ja .mam_konec       ;je-li větší, pak také
jmp .najdi_konec_cisla     ;zkusíme porovnat další znak
```

```
.mam_konec
ret                         ;skončíme

; ***** Sem musíme nakopírovat všechny potřebné podprogramy *****

SECTION .data

buffer times 25 db ' '      ;sem uložíme text výsledku
```

Celý program uložíme pod jménem test.asm a zkusíme přeložit.

```
nasm -f bin -o test.com test.asm
test.asm:6: error: operation size not specified
test.asm:11: error: symbol 'Skipspace' undefined
test.asm:13: error: operation size not specified
test.asm:38: error: symbol 'znovu' undefined
test.asm:50: error: operation size not specified
test.asm:52: error: operation size not specified
test.asm:141: error: phase error detected at end of assembly.
```

Program obsahuje značné množství chyb. Pokusme se je odstranit. První chyba je na řádku 6.

```
mov [di],0          ;za číslo zapíšeme nulový byte
```

V instrukci chybí specifikace velikosti operandu. Instrukci opravíme na:

```
mov byte [di],0     ;za číslo zapíšeme nulový byte
```

Na řádku 11 je špatně zapsán název podprogramu má být SkipSpace. Na řádku 13 je stejná chyba jako na řádku sedm. Na řádku 38 jsme před lokálním symbolem znovu v instrukci jz znovu zapomněli udělat tečku.

```
jz .znovu
```

Na řádcích 50 a 52 chybí v instrukci cmp [di],'0' velikost prvního operandu. Napravíme to jednoduše tak, že mezi instrukci cmp a hranatou závorku napíšeme klíčové slovo byte.

Opravený program je uložen na přiloženém CD jako test1.asm. Program jsme sice přeložili, ale nevíme, jestli bude správně fungovat. Ačkoli je program přeložen, nic na výstup nevypíše. Zkontrolujme kód zajišťující výpis na obrazovku.

```
mov ah,0x40         ;služba Dosu zapiš soubor
mov dx,di           ;ukazatel už máme v DI, vyžadujeme v DX
mov cx,25           ;počet znaků, co budeme tisknout
int 0x21            ;tiskneme
```

Zapomněli jsme nastavit registr BX na identifikační číslo souboru pro standardní výstup 0x01, systémové volání dostane místo hodnoty 0x01 hodnotu původní (10), která byla v registru zapsána. Takový soubor nemáme otevřen pro zápis, a proto i volání Dosu skončí s chybou (tu ale netestujeme, proto jsme nic nezjistili). Opravme program a pojmenujme ho test2.asm. Vyzkoušejme program pro čísla 45 a 50.

```
C:\test2 45 50
945
```

Program již vypisuje na obrazovku, jenže špatný výsledek. Musíme zjistit, kde je chyba, a použijeme k tomu odladovacího programu.

```
C:>grdb
GRDB version 3.6 Copyright (c) LADsoft 1997-2002
->
```

Program nahrajeme i s parametry příkazem l test2.com 45 50:

```
->l test2.com 45 50
Size: 000000E1
->
```

Předpokládejme, že podprogramy jsou v pořádku. Příkazem **p** je budeme přeskakovat. Zaměřme se na podprogramy ASCIIToNum a registr EAX, ve kterém vrací výsledek. První podprogram ASCIIToNum se volá těsně za instrukcí MOV ecx,10. Ukažme si celý výpis práce s debuggerem až za první volání podprogramu ASCIIToNum:

```
->l test2.com 45 50
Size: 000000E1
->p

eax:00000000 ebx:00000000 ecx:000000E1 edx:00000000 esi:00000081
edi:00000000 ebp:00000000 esp:0000FFEE eip:00000103 eflags:00000202
NV UP EI PL NZ NA PO NC
ds: 10FB es:10FB fs:10FB gs:10FB ss:10FB cs:10FB
10FB:0103 E8 44 00    call     014A

->p

eax:00000000 ebx:00000000 ecx:000000E1 edx:00000000 esi:00000081
edi:00000084 ebp:00000000 esp:0000FFEE eip:00000106 eflags:00000287
NV UP EI MI NZ NA PE CY
ds: 10FB es:10FB fs:10FB gs:10FB ss:10FB cs:10FB
10FB:0106 C6 05 00    mov     byte [di],0000     ds:[0084]=20
```

```
->p

eax:00000000 ebx:00000000 ecx:000000E1 edx:00000000 esi:00000081
edi:00000084 ebp:00000000 esp:0000FFEE eip:00000109 eflags:00000287
NV UP EI MI NZ NA PE CY
ds: 10FB es:10FB fs:10FB gs:10FB ss:10FB cs:10FB
10FB:0109 66 B9 0A 00 00 00 mov ecx,0000000A

->p

eax:00000000 ebx:00000000 ecx:0000000A edx:00000000 esi:00000081
edi:00000084 ebp:00000000 esp:0000FFEE eip:0000010F eflags:00000287
NV UP EI MI NZ NA PE CY
ds: 10FB es:10FB fs:10FB gs:10FB ss:10FB cs:10FB
10FB:010F E8 6D 00    call     017F

->p

eax:000003B1 ebx:00000000 ecx:0000000A edx:00000000 esi:00000081
edi:00000084 ebp:00000000 esp:0000FFEE eip:00000112 eflags:00000297
NV UP EI MI NZ AC PE CY
ds: 10FB es:10FB fs:10FB gs:10FB ss:10FB cs:10FB
10FB:0112 66 89 C2    mov      edx,eax

->
```

Výsledek prvního převodu je 0x3B1, což určitě není 45. Kde je chyba? Zkontrolujeme, jestli funkce ASCIIToNum dostala správné parametry. Zkusíme vypsat paměť, na kterou ukazuje parametr předávaný v registru SI.

```
->d si
10FB:0080    20 34 35-00 35 30 0D-01 01 01 01-01 01 01 01  45.50.........
10FB:0090 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 ................
10FB:00A0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 ................
10FB:00B0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 ................
10FB:00C0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 ................
10FB:00D0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 ................
10FB:00E0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 ................
10FB:00F0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 ................
10FB:0100 BE 81 00 E8-44 00 C6 05-00 66 B9 0A-00 00 00 E8 ....D....f......
->
```

Konverzní funkce nekontroluje nijak vstupní znaky, na vstupu zůstala jedna mezera (ASCII kód 0x20). Na začátku programu musíme zavolat podprogram SkipSpace, který se úplně první mezery zbaví. Opravme program a uložme ho pod jménem: test3.asm.

```
C:\test3 45 50
5
```

Výsledek součtu je pět. V programu bude zřejmě ještě nějaká chyba. Znovu krokujme program až do místa prvního volání podprogramu ASCIIToNum:

```
->p

eax:00000034 ebx:00000000 ecx:0000000A edx:00000000 esi:00000083
edi:00000084 ebp:00000000 esp:0000FFEE eip:00000112 eflags:00000287
NV UP EI MI NZ NA PE CY
ds: 10FB es:10FB fs:10FB gs:10FB ss:10FB cs:10FB
10FB:0112 E8 6D 00    call     0182

->d si
10FB:0080          35-00 35 30 0D-01 01 01 01-01 01 01 01    5.50.........
10FB:0090 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 ................
10FB:00A0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 ................
10FB:00B0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 ................
10FB:00C0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 ................
10FB:00D0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 ................
10FB:00E0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 ................
10FB:00F0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 ................
10FB:0100 BE 81 00 E8-41 00 E8 44-00 C6 05 00-66 B9 0A 00 ....A..D....f...
->
```

Zdá se, že problém bude v podprogramu SkipSpace, který přeskakuje o jeden znak navíc. Tak se na vstup podprogramu dostanou číslice 5 a 0, což by minulý výsledek 5 opravňovalo. Zkontrolujme podprogram:

```
SkipSpace:
.znovu:
lodsb                ;načteme znak do AL z DS:SI
cmp al,' '           ;porovnáme s mezerou
jz znovu             ;je-li mezera, načteme další
ret
```

Skutečně, instrukce LODSB zvýší pokaždé hodnotu v registru SI o jedna, i v případě, že nebyla načtena mezera. Na konci podprogramu stačí registr SI snížit o jedna (stačí přidat instrukci DEC si).

```
SkipSpace:
.znovu:
lodsb                ;načteme znak do AL z DS:SI
cmp al,' '           ;porovnáme s mezerou
jz znovu             ;je-li mezera, načteme další
```

```
dec si
ret
```

Chybu opravme a opravený program pojmenujme test4.asm. Přeložme a vyzkoušejme:

```
C:\test4 45 50
95
```

Zdá se, že program pracuje správně.

10.14 Rezidentní program aneb peklo na zemi

Rezidentní program i po svém ukončení zůstane zaveden v paměti. Většinou na sebe přesměruje některé přerušení (procesoru), aby s ním uživatel nebo jiné programy mohli komunikovat. Klasickým příkladem rezidentního programu je ovladač myši, který bude obsluhovat sériový port počítače, číst kódy, které myš vysílá, a pomocí aplikačního rozhraní (systémových volání) na přerušení int 0x33 předávat události aplikacím.

Operační systém DOS není reentrantní, což znamená, že nelze provádět volání Dosu v případě, že přerušení přerušilo nějakou systémovou službu (volání) operačního systému (to se bohužel stává velice často). Opětovné volání Dosu z přerušení by způsobilo pád systému. Existují sice možnosti, jak se s omezením slušně vypořádat, ovšem svojí značnou složitostí přesahují tematický rozsah knihy.

Předveďme si konstrukci jednoduchého rezidentního programu, který aktivujeme klávesou Scroll-Lock. Program obarví všechna písmena na obrazovce na červeno (tak se vyhneme systémovým voláním a snadno poznáme, jestli program funguje).

Ačkoli jsme se až dosud úzkostlivě vyhýbali přímé komunikaci s periferním zařízením pomocí instrukcí in a out, v případě rezidentního programu se již bez přímé komunikace s periferním zařízením neobejdeme.

Spokojme se s faktem, že čtením z portu 0x60 dostaneme scan kód stisknuté klávesy, který určuje pouze její polohu na klávesnici, a nikoliv její ASCII kód. Scan kód klávesy Scroll-Lock je 0x46.

Při každém stisku klávesy se generuje přerušení IRQ 1, které znak (scan kód) přijatý z klávesnice (scan kód) převede na ASCII kód a zařadí do fronty nepřečtených kláves. Z fronty je čte operační systém (nebo aplikace) pomocí služeb Biosu.

Abychom mohli náš program aktivovat, budeme muset přesměrovat obsluhu přerušení IRQ 1 (int 0x9) do našeho programu. Pak můžeme (opakovaně) číst z portu 0x60 číslo scan kódu klávesy. Nebude-li Scroll-Lock stisknut, předáme řízení původní obsluze přerušení. Jinak provedeme požadovanou akci a předáme řízení originální obsluze (kdybychom již nechtěli klávesu zpracovat a vrátit se rovnou, museli bychom přerušení odhlásit u řadiče přerušení dvojicí instrukcí mov al, 0x20 out 0x20,al a pak teprve vykonat IRET – návrat z přerušení).

Během přerušení musíme zachovat kontext běžícího programu, tedy uchovat hodnoty všech používaných registrů uvnitř našeho podprogramu. Jinak by přerušený program

ztratil hodnoty uložené ve svých registrech a následky ztráty by byly jistě katastrofální (pád systému).

Napišme nejprve podprogram obarvi, který obarví všechny znaky na obrazovce na červeno. Organizací videopaměti jsme se krátce zabývali na začátku kapitoly věnované operačnímu systému DOS. Od segmentové adresy 0xB800 jsou uloženy právě zobrazené znaky na obrazovce. Organizované jsou do dvojic byte znak a barva. Naším úkolem bude všechny byte barev ležících na adresách 0xB800:0x0001, 0xB800:0x0003, 0xB800:0x0005 atd. zapsat tak, aby byla barva červená. Bez dalšího odůvodňování bude červený byte 0x04.

Podprogram vyplní dohromady 80x25 byte v paměti, což je standardní rozlišení textového režimu (všechny znaky).

```
obarvi:
push ax              ;ulož registr AX na zásobník
push cx
push si
push es              ;stejně tak i ostatní
xor si,si            ;vynuluj SI
mov ax, 0xB800       ;nahraj segmentovou adresu do registru AX
mov es,ax            ;a z registru AX do ES
mov cx,80*25         ;budeme opakovat 25*80
.opakuj:
inc si               ;zvyš si o jedna
mov byte [es:si],0x4 ;zapíšeme barvu 0x04 - červená
inc si               ;zvýšíme o 1
dec cx               ;snížíme CX o 1
jnz .opakuj          ;dokud není nula, tak skočíme na .opakuj
pop es               ;obnovíme původní hodnoty registrů
pop si
pop cx
pop ax
ret
```

Podprogram můžeme vyzkoušet tak, že ještě doplníme hlavičku programové sekce a podprogram zavoláme, pak ukončíme program pomocí služby Dosu 0x4C.

```
SECTION .text
call obarvi
mov ax, 0x4c00
int 0x21
obarvi:
...
```

Zbývá už jen napsat novou obsluhu přerušení IRQ 1. Mohla by vypadat takto:

```
nova_obsluha:
push ax                  ;ulož hodnotu v registru AX
in al, 0x60              ;zjisti scan kód klávesy
cmp al, 0x46             ;porovnej s 0x46 (Scroll-Lock)
```

```
jnz predej_dal              ;pokud není, skocíme na predej_dal
call obarvi                 ;zavoláme obarvi
predej_dal:
pop ax                      ;obnov hodnotu v AX
jmp far [cs:stary_vektor]   ;předej řízení na původní obsluhu
```

Ostatní segmentové registry mohou být v průběhu volání přerušení nastaveny na libovolné hodnoty, proto musíme „proměnnou" stary_vektor vztáhnout k segmentovému registru CS, který bude určitě ukazovat na správný segment v paměti. Do proměnné stary_vektor musíme předem nahrát původní vektor přerušení. Ten nastavíme v podprogramu setup a zároveň do tabulky zapíšeme vektor nový, který bude ukazovat na návěští nova_obsluha.

```
setup:
cli                         ;zakaž přerušení
xor ax,ax                   ;vynuluj AX
mov es,ax                   ;tabulka přerušení je na segmentové adrese 0
mov ax,nova_obsluha         ;nová obsluha přerušení začíná na návěští
                            ;nova_obsluha
xchg ax,[es:0x9*4]          ;vypočteme index do tabulky, do AX uložíme
                            ;původní offset a zároveň zapíšeme offset
                            ;nový

mov [ds:stary_vektor],ax    ;a uložíme ho do "proměnné" stary_vektor
mov ax,cs                   ;segmentová adresa je CS
xchg ax,[es:0x9*4+2]        ;vypočteme index do tabulky, do AX uložíme
                            ;původní segment a zároveň uložíme segment
                            ;nový

mov [ds:stary_vektor+2],ax  ;uložíme ho také do "proměnné"
                            ;stary_vektor, ale o 2 byte dál
sti                         ;povol přerušení
ret                         ;návrat z podprogramu
```

Operační systém musíme požádat, aby program po ukončení nezrušil, ale ponechal v paměti. Zajistit požadovanou funkci můžeme systémovým voláním 0x31.

Vstup:	Výstup:
Ponech rezidentní AH = 0x31 AL = výstupní kód DX = počet paragrafů paměti, které zůstanou rezidentní	nic

Celý program rezident.asm bude vypadat takto:

```
SECTION .text
org 0x100
jmp inicializace
```

```
nova_obsluha:
push ax              ;ulož hodnotu v registru AX
in al, 0x60          ;zjisti scan kód klávesy
cmp al, 0x46         ;porovnej s 0x46 (Scroll-Lock)
jnz predej_dal       ;pokud není, skočíme na predej_dal
call obarvi          ;zavoláme obarvi
predej_dal:
pop ax               ;obnov hodnotu v AX
jmp dword [cs:stary_vektor]  ;předej řízení na původní obsluhu

obarvi:
push ax              ;ulož registr AX na zásobník
push cx
push si
push es              ;stejně tak i ostatní
xor si,si            ;vynuluj SI
mov ax, 0xB800       ;nahraj segmentovou adresu do registru AX
mov es,ax            ;a z registru AX do ES
mov cx,80*25         ;budeme opakovat 25*80
.opakuj:
inc si               ;zvyš si o jedna
mov byte [es:si],0x4 ;zapíšeme barvu 0x04 - červená
inc si               ;zvýšíme o 1
dec cx               ;snížíme CX o 1
jnz .opakuj          ;dokud není nula, tak skočíme na .opakuj
pop es               ;obnovíme původní hodnoty registrů
pop si
pop cx
pop ax
ret

stary_vektor dd 0

inicializace:
call setup           ;zavolej nastavení proměnné stary_vektor
mov ax,0x3100        ;služba Dosu zůstaň rezident
mov dx,inicializace  ;vypočteme počet potřebných paragrafů paměti,
                     ;v paměti musí zůstat vše až do návěští inicializace.
shr dx,4             ;vydělíme 16
inc dx               ;přičteme 1
int 0x21             ;ukončíme program, zůstane rezidentní

setup:
cli                  ;zakaž přerušení
xor ax,ax            ;vynuluj AX
mov es,ax            ;tabulka přerušení je na segmentové adrese 0
mov ax,nova_obsluha  ;nová obsluha přerušení začíná na návěští
                     ;nova_obsluha
```

```
xchg ax,[es:0x9*4]          ;vypočteme index do tabulky, do AX uložíme
                            ;původní offset a zároveň doplníme offset
                            ;nový

mov [ds:stary_vektor],ax    ;a uložíme ho do "proměnné" stary_vektor

mov ax,cs                   ;segmentová adresa je CS
xchg ax,[es:0x9*4+2]        ;vypočteme index do tabulky, do AX uložíme
                            ;původní segment a vyměníme za nový

mov [ds:stary_vektor+2],ax  ;uložíme ho také do "proměnné"
                            ;stary_vektor, ale o 2 byte dál
sti                         ;povol přerušení
ret                         ;návrat z podprogramu
```

Program přeložíme obvyklým způsobem (nasm -f bin -o rezident.com rezident.asm).

Po svém spuštění program neudělá nic. Kdykoli později stiskneme klávesu Scroll-Lock, písmena na obrazovce se zbarví na červeno. Programu se můžeme zbavit buď restartováním počítače nebo uzavřením okna, ve kterém program běžel.

10.15 Volně šiřitelné zdroje informací

Na přiloženém CD naleznemte systémový manuál v anglickém jazyce Ralf's Brown Interrupt List. Manuál je rozdělen do několika tematických okruhů, nejdůležitějším je popis všech možných služeb přerušení, a to jak Biosu, tak i operačního systému DOS. V dalších částech manuálu najdeme popis hardwarových I/O portů nejčastějších periferních zařízení.

Na CD najdeme také překladač NASM s jeho úplným popisem (včetně popisu funkce všech instrukcí).

Z internetových stránek bych doporučil stránky věnované Assembleru http://www.assembler.cz, kde najdete dostatek dalších informačních zdrojů i příkladů.

PROGRAMUJEME VE WINDOWS

11

V této kapitole:

11.1 Úvod
11.2 Nativní aplikace pod Windows
11.3 Kompatibilita programů
11.4 Spouštíme DOS
11.5 Volně šiřitelné zdroje informací

11.1 Úvod

Operační systém Microsoft Windows byl původně jen grafickou nadstavbou operačního systému MS-DOS. Postupně se však vyvinul v plnohodnotný operační systém, který používá chráněný režim procesoru. Oproti operačním systémům typu „UNIX" (Linux, BSD atd.) OS Windows ve svém jádře poskytují služby grafického rozhraní (GUI).

11.2 Nativní aplikace pod Windows

Nativní aplikace systému Windows komunikují s jádrem operačního systému prostřednictvím tzv. volání API (application program interface, aplikační programové rozhraní). Rozhraní API poskytuje všechny běžné služby operačního systému i služby pro ovládání grafického uživatelského rozhraní (GUI).

Struktura grafických prvků je objektová, volání API vyžadují často předávání komplikovaných struktur i mnoho parametrů. Proto jednoduchý program v Assembleru využívající okna a několik tlačítek bude rozsahem na několik stránek.

Ukažme si jen kostrukci jednoduché grafické aplikace, která na obrazovku vypíše dialogové okno s textem Hello, World! Po stisku tlačítka OK aplikace skončí.

11.2.1 Systémová volání API

Pod operačním systémem DOS jsme jádro systému volali přes vektor přerušení 0x21. Pod OS Windows se služby API volají jako funkce systémových DLL knihoven (user32.dll, kernel32.dll). Popis rozhraní API najdeme například v Borland Delphi jako soubor win32.hlp. Stáhnout si ho můžeme také na:

ftp://ftp.borland.com/pub/delphi/techpubs/delphi2/win32.zip

11.2.2 Hello, World!

Navrhovaný program má vytvořit dialogové okno a skončit. Dialogové okno vytvoříme zavoláním API funkce MessageBoxA a o ukončení programu se postaráme funkcí rozhraní API zvanou ExitProcess.

V dokumentaci k MessageBoxA se dočteme:

```
int MessageBox(

    HWND hWnd,              // handle of owner window
    LPCTSTR lpText,         // address of text in message box
    LPCTSTR lpCaption,      // address of title of message box
    UINT uType              // style of message box
);
```

Prvním parametrem bude popisovač vlastněného okna, podobný, jako jsme používali při práci se soubory. Žádné nemáme, proto předáme nulu. Druhým parametrem je ukazatel na text, který se v dialogovém okně zobrazí (klasický řetězec zakončený nulovým byte), třetím ukazatelem je text titulku okna. Poslední parametr určuje typ dialogového okna (použijeme symbolickou konstantu MB_OK).

Druhé volání API funkce ExitProcess, stejně jako v Dosu, svým jediným parametrem určuje chybový kód ukončeného programu. Po jeho zavolání program skončí.

Aby byl život programátora v Assembleru jednodušší, byl vytvořen „include soubor" win32n.inc, ve kterém jsou definovány všechny typy parametrů API (např. HWND i LPCTSTR odpovídají v Assembleru obyčejnému dvojslovu – dword) i hodnoty symbolických konstant. Do programu tento soubor vložíme direktivou include:

```
%include "win32n.inc"              ;vložíme soubor win32n.inc
```

Použité API funkce uložené v dynamických knihovnách nesmíme zapomenout zpřístupnit našemu programu. Stačí použít direktivy EXTERN a IMPORT:

```
EXTERN MessageBoxA                 ;symbol MessageBoxA je definován jinde
IMPORT MessageBoxA user32.dll      ;konkrétně v user32.dll
EXTERN ExitProcess                 ;symbol ExitProcess je definován jinde
IMPORT ExitProcess kernel32.dll    ;konkrétně v kernel32.dll
```

Obdobně jako pro program v Dosu vytvoříme programové sekce CODE a DATA.

```
SECTION CODE USE32 CLASS=CODE
;sem vložíme kód

SECTION DATA USE32 CLASS=DATA
;sem zase data známá v době překladu
```


Zbývá vyřešit jediný problém: Jak předat API funkcím jednotlivé parametry. V kapitole „Propojujeme" se budeme detailně zabývat spojením programu v Assembleru s vyššími programovacími jazyky. Seznamme se zde pouze stručně s předávací konvencí parametrů rozhraní Win32 API.

Předávací konvence se jmenuje STDCALL. Parametry se předávají přes zásobník zprava doleva (jako v jazyce C), výmaz zásobníku provádí volaný (jako v jazyce Pascal).

Na zásobník uložíme parametry instrukcemi PUSH, požadovanou funkci zavoláme nepřímo instrukcí CALL, o výmaz zásobníku se nemusíme starat. Celý program bude vypadat takto:

```
%include "win32n.inc"                ;vložíme soubor win32n.inc

EXTERN MessageBoxA                   ;symbol MessageBoxA je definován jinde
IMPORT MessageBoxA user32.dll        ;konkrétně v user32.dll
EXTERN ExitProcess                   ;symbol ExitProcess je definován jinde
IMPORT ExitProcess kernel32.dll      ;konkrétně v kernel32.dll

SECTION CODE USE32 CLASS=CODE        ;začátek kódové sekce

..start:                             ;speciální návěští pro linker, které
                                     ;označuje vstupní bod programu
                                     ;(entrypoint)

push UINT MB_OK                      ;na zásobník uložíme poslední parametr,
                                     ;dialogové okno bude mít jen tlačítko OK

push LPCTSTR titulek                 ;na zásobník vložíme adresu řetězce
                                     ;ukončeného nulovým byte, který se
                                     ;zobrazí jako titulek okna

push LPCTSTR napis                   ;na zásobník vložíme adresu řetězce
                                     ;ukončeného nulovým byte, který se
                                     ;zobrazí jako text okna

push HWND NULL                       ;na zásobník uložíme nulu, žádné
                                     ;nadřazené okno nemáme

call [MessageBoxA]                   ;zavoláme API - vytvoříme dialogové
                                     ;okno návrat až po stisku tlačítka OK
```

```
push UINT NULL                    ;vrátíme rodiči nulový chybový kód
call [ExitProcess]                ;ukončíme proces

SECTION DATA USE32 CLASS=DATA

napis db 'Hello world!',0xD,0xA,0
                                  ;řetězec Hello, World! včetně
                                  ;odřádkování
titulek db 'Hello',0              ;titulek okna bude hello
```

Abychom mohli program přeložit a sestavit, budeme potřebovat verzi překladače NASM pod Windows, kterou najdeme na přiloženém CD nebo na http://nasm.sourceforge.net. Překladač NASM vytvoří objektový soubor, který budeme muset ještě sestavit linkerem. Použijeme volně šiřitelný linker alink, najdeme ho rovněž na CD nebo na jeho domovské stránce http://alink.sourceforge.net.

Soubor pojmenujeme třeba msgbox.asm. Překladač nasmw spustíme s parametrem -fobj:

```
C:\WIN32>NASMW -fobj msgbox.asm
```

Vytořený msgbox.obj sestavíme sestavovacím programem (alink):

```
C:\WIN32>ALINK -oPE msgbox
```

Parametr -o vybírá výsledný typ spustitelného souboru. Pro nativní Windows aplikace to je PE. Po sestavení vznikl nový soubor msgbox.exe, který můžeme spustit.

Další ukázkové programy včetně návodu na instalaci nasmw a alink najdeme na přiloženém CD.

11.3 Kompatibilita programů

Mikroprocesor běžící v chráněném režimu má k dispozici ještě jeden speciální mód procesoru, ve kterém může úloha běžet. Jmenuje se VM86, jedná se o speciální režim, který simuluje zpětně adresaci i chování reálného režimu procesoru. Všechny privilegované instrukce (např. cli popf atd.) i přístup k periferním zařízením (instrukce in, out) jsou zachytávány jádrem operačního systému a simulovány, tak aby si úloha (program) myslela, že skutečně ovládá počítač, a ne jen jakýsi simulátor. Všechna volání Dosu i Biosu jsou přesměrována zpět do jádra operačního systému.

11.4 Spouštíme DOS

Abychom si mohli doposud probírané programy spustit i v prostředí operačního systému Windows, musíme spustit aplikaci „Příkazová řádka" nebo v nabídce Start/Spustit spustit program cmd anebo command. Potom se na obrazovce objeví okno běžící ve VM86 režimu a simulující chování původního operačního systému DOS. Předpokládejme, že jsme překladač nasm i zdrojové soubory programů překopírovali do adresáře NASM, který se nachází v kořenovém adresáři disku C. Příkazem

```
cd C:\NASM
```

se do adresáře přesuneme a tam budeme provádět i testování našich programů, které jsme si doposud probrali. Pamatujme na omezení operačního systému DOS jen na 8+3 znaková jména souborů, a proto i při tvorbě vlastních programů pod OS DOS dbejme na toto omezení. K editaci zdrojových souborů použijme nejlépe editor, který zobrazuje i čísla řádků v souboru, tak abychom mohli chybu v souboru snadno lokalizovat. Pozor si musíme dát i na příliš přátelské aplikace, které ke klasické příponě souboru .asm přidají ještě .txt.

11.5 Volně šiřitelné zdroje informací

Na přiloženém CD najdeme kromě překladače i linkeru další příklady a různé návody zabývající se programováním v Assembleru pod Windows. Z internetových stránek lze doporučit např: http://win32asm.cjb.net, http://rs1.szif.hu/~tomcat/win32, http://www.eskimo.com/~htak/win32asm nebo http://www.assembler.cz, kde najdemte další odkazy i užitečné informace.

PROGRAMUJEME V LINUXU

12

V této kapitole:

12.1 Úvod
12.2 Jak vypadá uspořádání paměti procesu
12.3 Předávání příkazové řádky a prostředí
12.4 Volání operačního systému
12.5 Chybové kódy
12.6 Manuálové stránky
12.7 Hello world!
12.8 Asmutils aneb jak si ulehčit práci
12.9 Práce s makry Asmutils
12.10 Vstupně-výstupní operace se soubory (I/O)
12.11 Práce s adresáři
12.12 Vstup z klávesnice
12.13 Alokace paměti
12.14 Odlaďujeme
12.15 Assembler GAS
12.16 Volně šiřitelné zdroje informací
12.17 Možnosti překladače a linkeru

12.1 Úvod

Linux patří mezi moderní víceúlohové operační systémy. Jeho jádro je z drtivé většiny naprogramováno v jazyce C, ale malá část je napsána v Assembleru, který je použit pouze v nezbytně nutných místech – tedy v hardwarově závislých částech operačního systému. Díky snadné přenositelnosti jazyka C se Linux snadno rozšířil i na jiné procesory než x86 kompatibilní.

Jako každý jiný moderní víceúlohový operační systém dbá i Linux na důsledné oddělení jednotlivých procesů. To znamená, že neexistuje způsob, jak modifikovat cizí proces (nebo jádro) z procesu našeho. (Pád jedné aplikace nezpůsobí pád jiné nebo rovnou celého operačního systému.) Zajistit požadované chování musíme s využitím hardwarových prostředků počítače.

V počítačích x86 kompatibilních zajistí ochranu paměti (tedy i procesů) chráněný mód procesoru (protected mode). Připomeňme si, že tzv. chráněný režim (procesoru) umožňuje kontrolu nad činností programů v počítači. Prostřednictvím chráněného režimu je možné ovlivňovat chování právě zpracovávaného kódu z hlediska přístupových práv do paměti a k periferním zařízením. Ochrany jsou rozděleny mezi jádro operačního systému (které může „vše") a procesy. Procesy mohou vykonávat jen neprivilegované instrukce a zapisovat do paměti, která jim byla přidělena.

Chráněný režim procesoru nabízí i podporu pro virtuální paměť. Jádro operačního systému pak vy-

konává všechny činnosti spojené s výměnou stránek v paměti (viz kapitola o operačním systému). O překlad logické adresy na fyzickou adresu se stará samozřejmě HW počítače.

Proces (program) v Linuxu má díky virtuální paměti (a 32bitovým registrům) k dispozici 4 GB adresového prostoru pro každý proces. Pokud bychom si snad udělali výpis paměti procesu od adresy nula až na konec, ostatní procesy ani jádro tam nenajdeme. Zůstávají před procesem ukryty v jiné části paměti, do které jde přistoupit pouze privilegovanou změnou některého z bývalých segmentových registrů.

12.2 Jak vypadá uspořádání paměti procesu

Už víme, že v celých 4 GB paměti, co má proces k dispozici, není nikdo jiný kromě našeho procesu (programu). Jak už jsme si řekli v předcházejících kapitolách, každý program se skládá z několika sekcí. V jakém pořadí (myslí se pořadí podle vzestupných adres) budou tyto sekce nahrány, záleží na formátu spustitelného souboru. Pod Linuxem existuje sice více formátů spustitelných souborů, ale my se zaměříme na nejpoužívanější formát ELF – executable and linkable format (spustitelný a linkovatelný formát). Pro zjednodušení předpokládejme, že program nepoužívá žádné dynamicky vázané knihovny. Pak paměťový prostor programu vypadá takto:

adresa:

0x08048000 (128MB)

.text	Spustitelný kód
.data	Data známá v době překladu (statická)
.bss	Data dynamická (heap neboli hromada)
...	volno
.stack	zásobník

0xBFFFFFFF (3GB)

Obvykle se program nahraje od 128MB adresového prostoru. Program se zavede do paměti tak, že se do paměti uloží pouze jedna stránka programu. Ostatní se nahrají až v případě výpadku stránky (tj. v okamžiku, kdy je potřeba jiná část programu, nepoužité části programů se v paměti nikdy fyzicky neobjeví).

V samotném obrazu spustitelného souboru (v programu uloženém na disku) není obsažena ani sekce .stack, ani .bss. Paměť se pro sekce přidělí až během zavádění programu do paměti.

Pokud program obsahuje dynamické knihovny, pak i moduly dynamických knihoven budou umístěny do stejného adresového prostoru, ovšem na jinou počáteční adresu (typicky od 1GB a výš). Rozložení sekcí těchto knihoven je úplně stejné, jako jsme si uvedli na obrázku.

Možná, že teď hloubáte nad tím, co je v té 3GB mezeře mezi koncem dynamických dat (.bss) a začátkem (vrcholem) zásobníku. Paměť sice patří procesu, ovšem nejsou pro ni přiděleny žádné stránky. Zápis na nealokované místo způsobí výpadek stránky, jádro zjistí, že stránku programu nikdy nepřidělilo, a program násilně ukončí.

Pokud budeme chtít vytvářet dynamická data v programu, musíme požádat operační systém (jádro), aby nám požadovanou paměť přidělilo. Jak se to dělá, si ukážeme později.

12.3 Předávání příkazové řádky a prostředí

Už víme, že proces je v úplné izolaci. Jak se tedy k němu dostanou například parametry z příkazové řádky nebo proměnné prostředí? Pokud budeme uvažovat stejný formát spustitelného souboru (tedy ELF), pak parametry programu z příkazové řádky a prostředí se do našeho procesu dostanou na samém vrcholu zásobníku. Podívejme se na jeho rozložení.

ESP po startu programu	volno
argc	počet parametrů (dword)
argv[0]	ukazatel na název programu
argv[1] ... argv[argc-1]	ukazatele na parametry programu
NULL	značí konec ukazatelů parametrů
env[0] env[1] ... env[n]	
NULL	označuje konec ukazatelů na prostředí

„Vybrat" parametry ze zásobníku můžeme jednoduše pomocí instrukce POP. Nesmíme zapomenout, že zásobník roste od horních adres k spodním. Proto první věc, kterou ze zásobníku hned po startu programu vybereme, bude argc – počet parametrů.

Pokud vybereme další ukazatel ze zásobníku, získáme tak ukazatel na jméno spuštěného programu. Má-li program nějaké další parametry (argc>1), budou následovat jejich ukazatele uložené postupně na zásobníku.

Příklady, jak zpracovávat příkazovou řádku, si ukážeme, až budeme umět vypisovat na obrazovku.

12.4 Volání operačního systému

V Dosu jsme se ke službám operačního systému dostávali přes přerušení int 0x21. V Linuxu je to obdobné a jde o přerušení int 0x80. Jak se tedy pomocí přerušovací instrukce int dostaneme do jádra, když v našem paměťovém prostoru vůbec není? Připomeň-

me si, že v souvislosti s instrukcí int jsme mluvili také o změně kontextu. Díky chráněnému režimu se po provedení instrukce int 0x80 změní kontext (vymění se hodnota v segmentovém registru) a začne se vykonávat kód, který je v jádru. Po jeho skončení instrukce int normálně pokračuje ve vykonávání našeho programu.

Podobně jako v Dosu se parametry jednotlivých systémových volání (syscall) předávají v registrech (je to plus z hlediska rychlosti). Konkrétně číslo služby, kterou chceme vyvolat, se předává v registru EAX. Pokud má služba nějaký parametr, potom se předá v registru EBX, má-li další, je předán v registru ECX, atd. Z toho plyne, že parametry systémových volání se po řadě předávají v pevně stanovených registrech EBX, ECX, EDX, ESI, EDI. Jádro 2.4.x a vyšší přišlo s posledním parametrem, který se bude předávat v registru EBP.

12.5 Chybové kódy

Návratová hodnota systémového volání, předávaná v registru EAX, vždy indikuje stav, s jakým systémové volání skončilo. Chybový kód je vždy záporné číslo. Jeho hodnota specifikuje blíže situaci, kvůli které došlo k chybě. V programech napsaných v Assembleru stačí většinou rozlišit stav úspěch – neúspěch. Avšak během odlaďovací fáze jsou chybové kódy vždy správným vodítkem k nalezení chyby v programu.

V každé manuálové stránce popisující určité systémové volání je vždy uveden seznam všech možných chybových kódů včetně popisu možných příčin selhání.

12.6 Manuálové stránky

Předpokládejme, že budeme chtít navrhnout ten úplně nejjednodušší program, a to takový, který hned po svém startu ukončí svoji činnost. Vzpomínáte? V Dosu to bylo AH=0x4C, ukonči program. V Linuxu tedy musíme najít něco podobného. Podívejme se do souboru unistd.h, který se obyčejně nachází v adresáři /usr/src/linux/include/asm:

```
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_
/*
 * This file contains the system call numbers.
 */
#define __NR_exit          1
#define __NR_fork          2
#define __NR_read          3
#define __NR_write         4
#define __NR_open          5
#define __NR_close         6
....
#define _syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
```

```
long __res; \
__asm__ volatile ("int $0x80" \
    : "=a" (__res) \
    : "0" (__NR_##name),"b" ((long)(arg1))); \
__syscall_return(type,__res); \
}
...
```

Zbytek souboru unistd.h obsahuje makra zajišťující volání přerušení int 0x80 pro různý počet parametrů. Pro nás je zajímavý řádek

```
#define __NR_exit          1
```

Číslo služby jádra operačního systému pro ukončení procesu je 1.

Popis služeb jádra je uložen ve druhé sekci manuálových stránek. Vyvoláme je příkazem:

```
ruik:asm$ man 2 exit

_EXIT(2)      Linux Programmer's Manual      _EXIT(2)


NAME
    _exit, _Exit - terminate the current process

SYNOPSIS
    #include <unistd.h>

    void _exit(int status);

    #include <stdlib.h>

    void _Exit(int status);

DESCRIPTION
The function _exit terminates the calling process "immediately". Any
open file descriptors belonging to the process are closed; any children
of the process are inherited by process 1, init, and the process's
parent is sent a SIGCHLD signal. The value status is returned to the
parent process as the process's exit status, and can be collected using
one of the wait family of calls. The function _Exit is equivalent to
_exit.

RETURN VALUE
  These functions do not return.
....
```

Systémové volání exit potřebuje stejně jako v Dosu pouze jeden parametr a tím je návratový kód programu. Má samozřejmě stejný význam jako v OS DOS.

Tělo našeho „ukončovacího" programu by mohlo vypadat prozatím takto:

```
mov eax,1    ;číslo volání jádra - exit
mov ebx,0    ;návratový kód 0
int 0x80     ;zavoláme jádro a tak ukončíme náš proces (program)
```

12.7 Hello, World!

I pod operačním systémem Linux bude náš program vypisovat na obrazovku Hello, World! Připomeňme si, že existují speciální číselné identifikátory souborů, nazvané STDIN STDOUT a STDERR, se kterými jsme se seznámili už v kapitole o operačním systému. Náš program „Hello World" bude vypisovat tento text na obrazovku (tedy zapisovat na zařízení STDOUT) a pak skončí. Jak ukončit program, už víme, teď si ukážeme, jak zapsat do souboru, tedy na STDOUT.

Nahlédneme-li znovu do souboru unistd.h, zjistíme, že systémové volání pro zápis do souboru se jmenuje write.

```
WRITE(2)        Linux Programmer's Manual        WRITE(2)

NAME
    write - write to a file descriptor

SYNOPSIS
    #include <unistd.h>

    ssize_t write(int fd, const void *buf, size_t count);
...
```

První zádrhel, píšeme v Assembleru, ale manuálová stránka nám dává prototyp systémového volání write v jazyce C. Pokud neznáte jazyk C, bude se vám prototyp číst hůř, to ovšem neznamená, že nebudete moci vyluštit, co kam uložit. V Assembleru můžeme do registru zapsat buď přímo hodnotu nebo adresu. Myslím, že můžeme přijmout zjednodušení, že všechny parametry předáváme hodnotou, kromě parametrů označených hvězdičkou (to je ukazatel).

Funkce write vyžaduje tři parametry, podobně jako v Dosu: identifikační číslo souboru (handle nebo rukojeť), ukazatel (adresa) na data, která se mají uložit (vidíte, je tam napsáno *buf), a počet bajtů, které chceme zapsat. Výstupem funkce je počet skutečně zapsaných byte nebo chyba. Chybový stav rozpoznáme od zapsaných bajtů tak, že se vždy jedná o záporné číslo.

Zdrojový program přeložíme překladačem NASM do objektového souboru, který sestavíme sestavovacím programem ld. Získáme tak spustitelný soubor, který na obrazovku vypíše Hello, World!

Pro překlad použijeme verzi překladače NASM pro Linux. Objektový soubor typu elf vytvoříme přepínačem -f elf.

Program ld bude potřebovat adresu první instrukce, která se bude vykonávat po zavedení programu do paměti. Ve zdrojovém programu označíme místo první instrukce speciálním globálním návěštím _start.

Stejně jako v OS DOS musíme program rozložit do programových sekcí.

Program Hello, World! by mohl vypadat takto:

```
SECTION .text
global _start       ;aby linker věděl, kde je začátek našeho programu,
                    ;musíme definovat globálně symbol, _start
_start:
mov eax,4           ;první parametr, číslo systémového volání
mov ebx,1           ;víme, že konstanta STDOUT má hodnotu 1
mov ecx,hello       ;překladač sem doplní adresu řetězce hello
mov edx,012         ;počet znaků řetězce Hello World! včetně odřádkování
int 0x80            ;zavoláme jádro
                    ;připojíme i kód potřebný pro ukončení programu
mov eax,1           ;číslo volání jádra - exit
mov ebx,0           ;návratový kód 0
int 0x80            ;zavoláme jádro a tak ukončíme náš proces (program)

SECTION .data

hello db "Hello, world!",0xa ;náš řetězec včetně odřádkování
len equ $ - hello            ;symbolu len bude přiřazena délka řetězce
```

Program přeložíme snadno:

```
nasm -f elf hello.asm
```

A sestavíme linkerem ld.

```
ld -s -o hello hello.o
```

Parametr -o udává jméno výsledného spustitelného souboru. Parametrem -s chceme odstranit všechny nepotřebné (symbolické) informace.

Výsledný program můžeme spustit příkazem:

```
./hello

Hello World!
```

12.8 Asmutils aneb jak si ulehčit práci

Asmutils je sbírka systémových programů napsaných v Assembleru. Ke své činnosti nevyžadují přítomnost knihovny LIBC, protože komunikují přímo s jádrem operačního sy-

stému. Asmutils jsou vhodným nástrojem pro malé (embedded) systémy a záchranné (rescue) disky.

Programy jsou psané v NASM Assembleru pro procesory x86 kompatibilní. Díky sadě maker, která zastupují přímé systémové volání (v Linuxu int 0x80), jsou Asmutils velice snadno přenositelné na jiné operační systémy (stačí změna konfiguračního souboru). Podporována je řada operačních systémů: skupinya OS BSD (FreeBSD, OpenBSD, NetBSD), dále Unixware, Solaris a AtheOS (byl podporován BeOS).

Asmutils nabízí pro každý z operačních systémů sadu symbolických konstant a systémových maker, které zajišťují přenositelnost programu na jiné operační systémy. Vedlejším efektem je zvýšená přehlednost samotného kódu Assembleru.

Ukážeme si na příkladu Hello, World!, jak nám Asmutils mohou ulehčit práci.

```
%include "system.inc"
CODESEG                     ;značí začátek programové sekce
START:                      ;určuje začátek programu pro linker,
                            ;v podstatě stejné jako _start

sys_write STDOUT,hello,len  ;makro také naplní registry správnými
                            ;hodnotami
sys_exit 0                  ;ukonči program, výstupní kód 0

DATASEG

hello db "Hello World!",0xa
len equ $-hello
END
```

Po rozvinutí maker obdržíme stejný kód jako v předchozím příkladu. Budeme-li si přát program Hello, World! spustit pod OS FreeBSD změníme pouze cílovou platformu překladu (soubor MCONFIG) a makra zařídí všechny změny symbolických konstant i systémových volání.

Domovská stránka projektu Asmutils je http://linuxassembly.org/asmutils.html, kde lze stáhnout nejnovější verzi.

Makra i symbolické konstanty z balíku Asmutils budeme používat i v dalších programech. Ukažme si, jak program vytvořit a přeložit pomocí Asmutils.

Přesvědčíme se, že máme k dispozici překladač NASM. (Debian i Red Hat má balíček.) Na disk nakopírujeme archív asmutils-0.17.tar.gz. Po rozbalení obdržíme klasickou adresářovou strukturu doc/ src/ a inc/. Souborem určujícím cílovou platformu je soubor MCONFIG, který upravíme podle pokynů v souboru (nebo použijeme předpřipravený na CD).

Abychom mohli soubor hello.asm vytvořený v předchozím textu použít, musíme připsat název programu hello do souboru Makefile v src/. První řádky souboru Makefile určují názvy programů, které se budou kompilovat, proto název souboru hello (bez přípony .asm) do seznamu přidáme. Kompilaci spustíme příkazem make. Soubor hello.asm se přeloží.

Balík Asmutils je šířen pod licencí GNU GPL.

12.9 Práce s makry Asmutils

Parametry systémových volání předáme jako součást makra. Předávané parametry se doplní do registrů specifických pro určitý operační systém, stejně tak se vygenerují ostatní instrukce potřebné pro zavolání jádra OS (doplní se číslo volané služby a spouštěcí instrukce).

Všechna makra systémových volání začínají prefixem sys_, zbytek je název systémového volání, tak jak ho známe z manuálových stránek. V operačním systému Linux se parametry předávají po řadě ve stále stejných registrech nezávisle na druhu systémové služby. Návratová hodnota se vrací v registru EAX.

Makro sys_exit 0 se bude expandovat na:

```
mov eax,1           ;číslo volání jádra - exit
mov ebx,0           ;návratový kód 0
int 0x80            ;zavoláme jádro a tak ukončíme náš proces (program)
```

Jako parametr makra můžeme uvést jakýkoli přípustný operand, bude-li návratový kód uložen v „proměnné" rtn, můžeme samozřejmě zapsat:

`sys_exit [rtn]` a makro se expanduje na:

```
mov eax,1           ;číslo volani jadra - exit
mov ebx,[rtn]       ;návratový kód je uložen v proměnné rtn
int 0x80            ;zavoláme jádro a tak ukončíme náš proces (program)
```

Víme-li, že správnou hodnotu registr již obsahuje, můžeme parametr makra přeskočit klíčovým slovem EMPTY. Příklad užívající EMPTY si předvedeme v příkladu kopírujícím soubory.

12.10 Vstupně-výstupní operace se soubory (I/O)

S prací se soubory jsme se seznamovali již několikrát, principy práce se soubory zůstávají stejné i ve skupině operačních systémů Unix. Nově musíme brát v úvahu přístupová práva k jednotlivým souborům i jejich vlastníky.

Vekou oporou nám bude i druhá sekce manuálových stránek, popisující všechny možné operace se soubory.

12.10.1 Otevři soubor

Systémové volání open zajišťuje kromě otevření souboru také jeho případné vytvoření. V druhé sekci manuálových stránek (man 2 open) nalezneme kompletní popis jednotlivých parametrů systémového volání v jazyce C. V krátkosti se seznámíme s jednotlivými parametry.

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Prvním parametrem systémového volání open je ukazatel na jméno souboru (ukončené nulovým byte), druhým parametrem předáváme bitové pole příznaků, kterými jádru operačního systému sdělíme mód otevření souboru, a nakonec nepovinný třetí parametr bude specifikovat přístupová práva nově vzniklého souboru.

Zpět dostaneme identifikační číslo souboru (file descriptor, handle) nebo chybu reprezentovanou záporným číslem.

Díky Asmutils můžeme použít tytéž názvy bitových příznaků, jako jsou uvedeny v manuálové stránce:

Příznak	Funkce
O_RDONLY	Otevři soubor jen pro čtení
O_WRONLY	Otevři soubor jen pro zápis
O_RDWR	Otevři soubor jen pro čtení i zápis
O_CREAT	Vytvoř soubor, neexistuje-li
O_TRUNC	Zkrať soubor na nulovou délku
O_APPEND	Po otevření zapisuj vždy od konce souboru (nebezpečné s NFS)
O_LARGEFILE	Kdykoli si budeme přát pracovat se 4GB soubory, musíme použít tento příznak

Bude-li druhý parametr obsahovat příznak O_CREAT, musíme třetím parametrem specifikovat přístupová práva nově vzniklého souboru. Stejně jako v předchozím případě můžeme použít stejné symbolické konstanty, jako jsou v manuálových stránkách. Uveďme si pouze nejčastěji používané:

Příznak	Funkce
S_IRWXU	Vlastník může číst, zapisovat a spouštět
S_IRGRP	Skupina smí číst
S_IROTH	Others smí číst

Soubor, jehož jméno je uloženo v „proměnné" jméno, otevřeme pro čtení a zápis jednoduše:

```
sys_open jmeno, O_RDWR, EMPTY

test eax,eax        ;jak snadno testovat hodnoty v registrech na záporné
                    ;číslo, jsme si ukázali v kapitole
js .chyba_open      ;užitečné kousky kódu
```

Jméno souboru v datové sekci nadefinujeme pseudoinstrukcí DB.

```
jmeno DB "muj_soubor_ktery_ma_libovolne_dlouhy_nazev",0
```

Nic nám nebrání příznaky v bitovém poli různě kombinovat. Jednotlivé příznaky budeme pak oddělovat operátorem or |. V následujícím příkladu si budeme přát soubor otevřít pro čtení a zápis nebo vytvořit a otevřít, pokud nebude existovat (s právy 0700).

```
sys_open jmeno, O_RDWR | O_CREAT, S_IRWXU
test eax,eax
```

```
js .chyba_open      ;skočíme na .chyba_open, když se otevření nezdařilo
...                 ;v registru EAX bude uloženo identifikační číslo
                    ;souboru
```

12.10.2 Uzavři soubor

Soubor uzavřeme stejně jako v OS DOS systémovým voláním close – uzavři soubor. Makro se bude jmenovat sys_close a bude jako svůj jediný parametr očekávat identifikační číslo souboru.

```
sys_close eax       ;v předchozím případě jsme id. číslo souboru měli
                    ;v registru EAX
```

12.10.3 Čti ze souboru

Data ze souboru lze číst v blocích libovolné délky. Neurčíme-li jinak, budou se číst od pozice posledního čtení nebo zápisu (sekvenčně).

Systémové volání čtení ze souboru se nazývá read. Z jeho manuálové stránky vyčteme potřebné parametry.

```
ssize_t read(int fd, void *buf, size_t count);
```

Prvním parametrem je identifikační číslo souboru, druhým je ukazatel na místo v paměti, kam budeme chtít přečtená data uložit, třetí parametr specifikuje maximální počet byte, které se přečtou (bývá maximální velikost vyhrazeného místa). Návratová hodnota systémového volání vrací buď chybový kód (určité záporné číslo) nebo počet skutečně přečtených byte.

Makro čtení ze souboru se jmenuje sys_read a využít ho můžeme i ke čtení z klávesnice, stačí když budeme data číst ze standardního vstupu (STDIN).

12.10.4 Zapiš do souboru

Zápis do souboru vyžaduje stejné parametry jako funkce pro čtení ze souboru. Výstupem bude počet zapsaných byte nebo chyba.

```
ssize_t write(int fd, const void *buf, size_t count);
```

Ukažme si jednoduchý program, který přečte znaky z klávesnice (až do stisku klávesy Enter) a převede text na velká písmena (nebudeme uvažovat národní abecedu). Program budeme ukončovat prozatím stiskem kláves Ctrl+c.

Stisk klávesy Enter ukončí načítání kláves a systémové volání read se vrátí zpátky do programu (chování terminálu lze změnit a vracet se po stisku každé klávesy). Všechny načtené znaky z intervalu 'a' – 'z' převedeme na interval 'A' – 'Z', ostatní ponecháme. Celý převedený řetězec vytiskneme (zapíšeme) na standardní výstup.

Programovou sekci budeme v Asmutils označovat jako CODESEG, datovou DATASEG a původní sekci .bss jako UDATASEG. Makra přibalíme direktivou include.

```
%include "system.inc"
%define MAX_DATA 10
```

```
CODESEG
START:

znova:                  ;odtud budeme číst další řádek

sys_read STDIN,prectena_data, MAX_DATA

test eax,eax            ;zkusíme, jestli jsme skončili s chybou (záp) číslo
js konec                ;ano? ihned skončíme
                        ;v EAX je uložen počet skutečně přečtených dat
add ecx,eax             ;spočítáme pozici posledního prvku + 1
porovnej_dalsi:
dec ecx                 ;snížíme ukazatel v řetězci
cmp byte [ecx],'a'      ;pokud je znak nižší než 'a', pak neleží v intervalu
jb neprevadej           ;a nebude se převádět
cmp byte [ecx],'z'      ;jestliže jeho hodnota je vyšší než 'z'
ja neprevadej           ;pak také ne
sub byte [ecx],0x20     ;malá písmena převedeme na velká odečtením konstanty
                        ;0x20
neprevadej:
cmp ecx,prectena_data ;je hodnota ukazatele stejná jako začátek bufferu?
jz vytiskni             ;ano, pak skončíme převádění a budeme tisknout
jmp short porovnej_dalsi    ;jinak zkusíme převést další písmeno

vytiskni:               ;tisknout budeme EAX byte, tj. počet přečtených znaků
                        ;makra se zpracovávají zprava doleva, takže je možné
                        ;použít registr EAX

sys_write STDOUT,prectena_data, eax

jmp short znova         ;a po vytištění přečteme další řádek
konec:
   sys_exit 255         ;ukončíme program s chybovým kódem 255

UDATASEG                ;začátek sekce pro neinicializované proměnné

prectena_data resb MAX_DATA

END                     ;některá makra potřebují znát konec souboru
```

Po přečtení řetězce postupně převedeme všechna malá písmena na velká. Převádíme od konce řetězce, ukazatel do řetězce je nahrán v registru ECX. Po převodu vytiskneme stejný počet byte, jako jsme přečetli, a začneme číst řetězec další.

Program se bude chovat správně, i když najednou zadáme více znaků, než je maximální délka připravené „proměnné" prectena_data (max. délku čtených dat předáváme jako parametr systémovému volání). Jádro operačního systému zbylé znaky uschová a pošle je aplikaci až během dalšího volání sys_read.

Ukažme si program, který překopíruje soubor A do souboru B. Soubory A a B budou určeny parametry programu z příkazové řádky.

Ze zásobníku přečteme počet parametrů, bude-li menší než 3, skončíme (jako první parametr se předává jméno souboru, ostatní dva budou požadované názvy souborů). Instrukcemi POP přečteme ze zásobníku ukazatele na jména souborů. Soubor A otevřeme pro čtení, soubor B vytvoříme nebo zkrátíme. Nastala-li chyba, program ukončíme. Soubor A budeme číst tak dlouho, dokud bude obsahovat nějaká data, ta budeme zapisovat do souboru B. Nakonec oba soubory uzavřeme.


```
%include "system.inc"
%define BUFF_LEN 4096
CODESEG
START:
pop eax                                 ;na zásobníku máme uložen počet
                                        ;parametrů příkazové řádky, nyní v EAX
cmp eax,3                               ;musí být víc než tři 1. nazev
                                        ;programu, 2. nazev A, 3. nazev B.
jae mame_dost_parametru                 ;ano, pak zkusíme soubory otevřít
mov eax,255                             ;ne, pak navrátíme návratový kód 255
konec:
sys_exit eax                            ;jako návratový kód použijeme hodnotu
                                        ;v EAX

mame_dost_parametru:
pop ebx                                 ;ze zásobníku přečteme 1. ukazatel na
                                        ;jméno spouštěného souboru - copy
pop ebx                                 ;pak jméno prvního souboru
sys_open EMPTY,O_RDONLY|O_LARGEFILE     ;otevřeme jen pro čtení
test eax,eax                            ;když chyba, pak se skončíme
js konec
mov ebp,eax                             ;identifikační číslo uložíme do
                                        ;registru EBP
pop ebx                                 ;do EBX uložíme ukazatel na druhý
                                        ;parametr soubor B

sys_open EMPTY,O_WRONLY|O_LARGEFILE|O_CREAT|O_TRUNC,S_IRWXU


                                        ;otevřeme a zkrátíme nebo vytvoříme
                                        ;s právy 0700
test eax,eax
js konec;                               ;když chyba, pak skončíme
mov ebx,eax                             ;identifikační číslo souboru uložíme
                                        ;do EBX
kopiruj_dalsi:
xchg ebp,ebx                            ;vyměníme EBX a EBP
sys_read EMPTY,buff,BUFF_LEN            ;z prvního souboru přečteme 1 blok dat
test eax,eax                            ;test na chybu i na nulu
```

```
    js konec_zavri                  ;chyba? pak uzavri soubory a skonci
    jz konec_zavri                  ;konec souboru? pak uzavri soubory a
                                    ;skonci
    xchg ebp,ebx                    ;vyměníme EBP a EBX, aby byl v EBX
                                    ;druhý soubor (id)
    sys_write EMPTY,EMPTY,eax       ;zapíšeme počet přečtených byte
                        ;z prvního souboru
    test eax,eax
    js konec                        ;otestujeme na chybu
    jmp short kopiruj_dalsi         ;budeme kopírovat další blok

    konec_zavri:
    sys_close EMPTY                 ;soubory před koncem uzavřeme
    xchg ebp,ebx                    ;i druhý soubor
    sys_close EMPTY
    jmp short konec                 ;a ukončíme

    UDATASEG
    buff resb BUFF_LEN              ;zde jsme vyhradili 4 KB na buffer
    END
```

Po úspěšném otevření souborů budeme číst data ze souboru A a ukládat je do „proměnné" buff. Po návratu ze systémového volání read dostaneme v registru EAX délku přečtených dat. Do souboru B zapíšeme stejný počet byte, jako jsme ze souboru A přečetli. Čtení ukončíme, přečteme-li ze souboru A nula byte (EAX = 0).

12.10.5 Změna polohy v souboru

Pozici příštího čtení nebo zápisu souboru lze změnit, stejně tak jako v OS DOS, systémovým voláním lseek. Ukažme si prototyp systémového volání:

```
off_t lseek(int fildes, off_t offset, int whence);
```

Prvním parametrem je již tradičně identifikační číslo souboru (file descriptor, handle), druhým parametrem vybíráme novou pozici (v bytech) v souboru a třetím parametrem, kterým je jedna ze tří symbolických konstant, určujeme, odkud bude nová pozice v souboru uvažována. Návratová hodnota funkce je nová pozice v souboru počítaná od jeho začátku (v byte) anebo chyba.

Chceme-li novou pozici v souboru určit absolutně od začátku souboru, užijeme symbolickou konstantu SEEK_SET. Pokud si přejeme novou pozici v souboru získat přičtením zadané pozice k pozici stávající, učiníme tak pomocí konstanty SEEK_CUR. Třetí možnost, konstanta SEEK_END, bude zadanou pozici přičítat ke konci souboru. Poslední možnost můžeme využít ke zjištění délky souboru tak, že vyvoláme:

```
sys_lseek [fd], 0, SEEK_END
```

Návratová hodnota bude délka souboru.

Uvažujme následující situaci: Co se stane, když vytvoříme nový soubor, posuneme ukazatel za konec souboru, zapíšeme a soubor uzavřeme?

V operačním systému DOS bylo místo od začátku souboru až do místa prvního zápisu vyplněno náhodným obsahem disku. Ve skupině operačních systémů Unix se vytvoří „děravý soubor", který nebude mít v „díře" přiděleno místo na disku, a tak vytvoříme soubor, jehož logická velikost bude větší než fyzická. Vytvoříme-li „děravý" soubor na disketě (nebo kdekoli jinde), nemusíme se omezovat kapacitou média, soubor pak může zabírat i více místa (logického), než je celková kapacita média.

12.10.6 Ostatní funkce pro práci se soubory

Některé souborové systémy v Un*x-like systémech dovolují vytvoření dalšího jména (reference) souboru. Podle typu reference na soubor rozlišujeme dva druhy odkazů na soubor.

Prvním typem, dnes již méně používaným, je reference obsahem (hardlink). V rámci jednoho souborového systému existuje více (názvů) souborů, které sdílí stejný obsah (práva i vlastníka). Nevýhodu je nemožnost vytvořit odkaz na soubor z jiného souborového systému. Nový odkaz na soubor můžeme vytvořit pomocí systémového volání link.

```
int link(const char *oldpath, const char *newpath);
```

Prvním parametrem je ukazatel na „původní" název souboru, druhým parametrem je ukazatel na nový název. Návratová hodnota volání je nula v případě úspěchu, jinak chyba.

Flexibilnější variantu nabízí druhé řešení – reference odkazem (symlink). Nebudeme se odkazovat přímo na data v souboru, ale na jeho původní název. Odkaz můžeme vytvořit na libovolný soubor (i neexistující) ležící v kterémkoli souborovém systému. Symlink si můžeme představit jako „zástupce" souboru. Kdykoli bude požadován název souboru, symlink „přihraje" operačnímu systému název souboru, na který ukazuje. Výjimku tvoří samozřejmě destruktivní volání jako smaž nebo přejmenuj, která pracují se symlinkem, a nikoliv s odkazovaným souborem. Symlink utvoříme pomocí systémového volání symlink.

```
int symlink(const char *oldpath, const char *newpath);
```

Parametry volání jsou totožné jako u funkce link. Proberme si zbylé systémové volání smaž a přesuň.

Systémové volání unlink maže soubory, přesněji jejich jména. Soubor se vymaže až po smazání posledního hardlinku odkazujícího se do souboru. (Drtivá většina souborů na disku má pouze jeden hardlink, a to název souboru, tak jak jsme ho pojmenovali při jeho vytvoření.)

```
int unlink(const char *pathname);
```

Systémové volání má pouze jediný paramter, ukazatel na název souboru, který si přejeme smazat. Návratovou hodnotou je nula v případě úspěchu, jinak chyba.

Uveďme si i prototyp systémového volání rename – přejmenuj soubor.

```
int rename(const char *oldpath, const char *newpath);
```

Parametry jsou totožné s již probranými voláními (link), a tak by bylo zbytečné je zde znovu popisovat.

Ukažme si program (symhard.asm), který vytvoří pro zadaný název souboru jeden hardlink a jeden softlink, který pojmenuje 1 a 2.

Parametry programu (příkazovou řádku) načteme ze zásobníku. První dvojslovo uložené na vrcholu zásobníku po startu programu obsahuje počet parametrů programu, další ukazatele na jednotlivé parametry programu. Hardlinky a softlinky utvoříme pomocí systémových volání link a symlink.

```
%include "system.inc"
CODESEG                 ;začátek kódové sekce
START:                  ;startovní symbol
pop ebx                 ;načteme počet parametrů programu
cmp ebx,2               ;první je název programu, druhý je soubor, na který
                        ;budeme vytvářet oba druhy odkazů
jz ok                   ;ano, máme dost parametrů
konec:
sys_exit 0              ;skončíme
ok:
pop ebx                 ;zahodíme ukazatel na jméno našeho programu
pop ebx                 ;do EBX uložíme ukazatel na soubor
sys_link EMPTY,jedna    ;vytvoříme hardlink
sys_symlink EMPTY,dva ;a softlink, netestujeme chybové kódy
jmp short konec         ;skončíme
DATASEG                 ;začátek datové sekce

jedna DB "1",0
dva DB "2",0

END                     ;označíme konec souboru
```

Jméno programu doplníme do Makefile a spustíme make. Jako parametr můžeme programu předat libovolný název souboru nebo adresáře. Soubor 1 bude představovat hardlink a soubor 2 zase softlink, nebude-li některý ze souborů 1 nebo 2 po spuštění existovat, došlo k chybě – souborový systém pravděpodobně nepodporuje typ operace, kterou jsme se chystali provést.

Příkazem ./symhard ./symhard vytvoříme symlink i hardlink na soubor symhard. Pomocí příkazů ls -l, chown a chmod můžeme zkoumat chování trojice souborů.

12.11 Práce s adresáři

Stejně jako v operačním systému DOS, i v Linuxu (nebo jiných Linux-like operačních systémech) má programátor k dispozici sadu systémových volání pro práci s adresáři. Díky Asmutils se jejich používání značně přiblíží volání obvyklému ve vyšších programovacích jazycích.

12.11.1 Vytvoř a zruš adresář (MKDIR, RMDIR)

Adresář vytvoříme pomocí systémového volání mkdir, jeho prototyp vypadá takto:

```
int mkdir(const char *pathname, mode_t mode);
```

Prvním parametrem je ukazatel na jméno nového adresáře, druhým parametrem předáme přístupová práva, která bude mít nový adresář. Pro práva lze užít symbolické konstanty, tak jako v případě volání open. Návratová hodnota je stejně jako u většiny podobných funkcí chyba nebo úspěch. Program tvořící adresář muj_adresar v adresáři /tmp by mohl vypadat takto:

```
%include "system.inc"
CODESEG                    ;začátek kódové sekce
START:                     ;startovní symbol
sys_mkdir jmeno, S_IRWXU   ;vytvoříme adresář s právy 0700
sys_exit 0                 ;ukončíme program
DATASEG
jmeno DB "/tmp/muj_adresar",0
END
```

Přístupová práva k novému adresáři jsme mohli zapsat i v oktalové soustavě, ovšem nikoliv 0700, jak jsme zvyklí z jazyka C (nebo z chmod), ale 700q.

Adresář zrušíme za pomoci systémového volání RMDIR, jehož jediný parametr je název adresáře, který si přejeme smazat.

```
int rmdir(const char *pathname);
```

12.11.2 Vyber aktuální adresář (CHDIR)

Funkce má jediný parametr a tím je adresář (společně s jeho cestou), který si přejeme označit jako aktuální.

```
int chdir(const char *path);
```

Kdybychom chtěli, aby předchozí program nastavil aktuální adresář na právě vytvořený, pak stačí vložit:

```
sys_chdir jmeno
```

před systémové volání sys_exit (ukončení programu).

12.11.3 Zjisti vybraný adresář (GETCWD)

Systémové volání zjisti vybraný adresář je implementováno v OS Linux od verze jádra 2.0. Ke zjištění aktuálního adresáře v systému se starším jádrem bylo nutné, stejně jako v případě BSD, použít jiný, mnohem složitější postup (viz pwd.asm). Systémové volání vyžaduje dva parametry: první parametr ukazuje na místo, kam se aktuální adresář uloží, druhý parametr specifikuje velikost volné paměti, na kterou odkazuje parametr první.

```
long sys_getcwd(char *buf, unsigned long size)
```

Ukažme si část programu pwd.asm, která vypíše aktuální adresář a odřádkuje.

```
sys_getcwd path,PATHSIZE          ;vyplní "proměnnou" path aktuálním
                                  ;adresářem

mov esi,ebx                       ;ukazatel na path uloží do ESI
xor edx,edx                       ;vynulujeme EDX

.next:

inc edx                           ;v EDX počítáme počet znaků, co budeme
                                  ;tisknout
lodsb                             ;načteme znak do AL (a k ESI přičteme 1)
or   al,al                        ;je už konec ?
jnz   .next                       ;ne? pak přečteme další
mov   byte [esi-1],__n            ;jinak místo nulového byte zapíšeme
                                  ;odřádkování
sub   esi,edx                     ;získáme znovu začátek řetězce (adresu)
                                  ;(autor si zřejmě neuvědomil, že
                                  ;hodnotu má již v registru EBX)

sys_write    STDOUT,esi, EMPTY    ;vypíšeme aktuální adresář na STDOUT,
                                  ;délku vypisovaného řetězce máme již
                                  ;uloženu v EDX
sys_exit_true                     ;speciální makro na ukončení programu
```

12.12 Vstup z klávesnice

Klávesnice je v operačních systémech Un*x reprezentována standardním vstupem. Budeme-li chtít číst znaky z klávesnice zakončené klávesou Enter, použijeme systémové volání read, tak jak jsme si ukázali v programu převádějícím malá písmena na velká.

Někdy potřebujeme číst klávesy po jedné nebo je při čtení nezobrazovat na standartní výstup. Změnit chování standardního vstupu můžeme pomocí speciálního systémového volání ioctl.

12.12.1 Systémové volání IOCTL

Ovladače zařízení nepřidávají nová systémová volání, ale vytvářejí pro obsluhu svého zařízení soubor funkcí dostupných přes systémové volání IOCTL. Prostřednictvím systémového volání IOCTL můžeme komunikovat s ovladačem příslušného zařízení a tak ovlivňovat jeho funkci. Zkratka IOCTL znamená Input/Output Control. Popis všech možných typů IOCTL by jistě naplnil další knížky, proto se omezíme jen na důležité IOCTL terminálu.

Volání IOCTL mají jeden povinný parametr a tím je identifikační číslo souboru, nad kterým si přejeme IOCTL provádět. Ostatní parametry jsou specifické pro každou funkci IOCTL.

Pod OS Linux měníme chování terminálu pomocí dvou nejčastěji používaných IOCTL funkcí: TCGETS a TCSETS. Voláním TCGETS dostaneme nastavení terminálu, změny můžeme provést voláním TCSETS. Kompletní popis najdeme v manuálové stránce man termios.

Struktury potřebné pro manipulaci s terminálem máme díky Asmutils také k dispozici. Chceme-li klávesy číst po jedné a nezobrazovat při stisku na obrazovku, změníme (vynulujeme) dva atributy terminálu – ICANON a ECHO. Atributy přečteme funkcí TCGETS, bitovou masku změníme dle požadavků a uvedeme v platnost voláním IOCTL TCSETS. V definici struktury B_STRUC se objevuje speciální makro, které rozšiřuje standardní makro STRUC (jeho definici nalezneme v souboru system.inc).


```
mov   edx,termattrs                ;do EDX uložíme adresu počátku struktury
sys_ioctl STDIN,TCGETS             ;tu naplníme
mov   eax,[termattrs.c_lflag]      ;přečteme bitové pole příznaků
push  eax                          ;terminálu uložíme na zásobník
and   eax,~(ICANON|ECHO)           ;změníme příznaky ECHO a ICANON
mov   [termattrs.c_lflag],eax      ;uložíme do struktury
sys_ioctl STDIN, TCSETS            ;promítneme změny
pop   dword [termattrs.c_lflag]    ;do struktury uložíme staré nastavení
```

Struktura bude deklarována v UDATASEG.

```
termattrs B_STRUC termios,.c_lflag
```

Jednodušší varianta změny chování klávesnice (standardního) vstupu neexistuje.

12.13 Alokace paměti

Během spouštění nového programu jádro přidělí programu paměť, tak jak bylo uvedeno v době jeho překladu. Potřebujeme-li další paměť, musíme požádat operační systém (jádro), aby nám ji přidělil. Narozdíl od OS DOS nežádáme jádro o blok paměti konkrétní velikosti, ale o prodloužení sekce .bss (poslední sekce programu pro neinicializované proměnné). Jakým způsobem budeme udržovat informace o volné a přidělené paměti v rámci sekce .bss, je záležitost programátora. Neužíváme-li ani knihovnu jazyka C, musíme si vlastní manažer paměti naprogramovat (nebo použít heap.asm z Asmutils).

Systémové volání prodlužující (nebo zkracující) délku sekce .bss se jmenuje brk.

```
void * brk(void *end_data_segment);
```

Jediným parametrem je nová adresa konce sekce .bss. Funkce vrací pro kontrolu poslední platnou adresu datové sekce (.bss). Potřebujeme-li zjistit poslední platnou adresu, můžeme systémové volání brk zavolat s nulovým ukazatelem. Funkce pak vrátí požadovanou poslední adresu. Typicky se brk volá takto:

```
sys_brk 0
add eax,kolik_byte_potrebuji
sys_brk eax
```

K dosavadnímu konci datové sekce (.bss) přičteme počet byte, které chceme přidělit, a zavoláme znovu systémové volání brk.

12.14 Odladťujeme

Standardním odladťovacím nástrojem pro operační systémy Un*x je gdb, který je však pro odladťovaní programů v Assembleru příliš těžkopádný. (navíc používá jinou syntax Assembleru). Seznamme se s odladťovacím nástrojem vyvinutým speciálně pro programy psané v Assembleru – ALD (Assembly Language Debugger).

Ovládá se přes příkazovou řádku několika příkazy. Program je zatím na počátku svého vývoje a tak podporuje jen procesory 0x86 kompatibilní a spustitelné soubory ve formátu ELF. I přes tato omezení je však pro programátora v Assembleru pro PC naprosto dostačující. Debugger ALD lze stáhnout z adresy: http://dunx1.irt.drexel.edu/~psa22/ald.html. Program je šířen pod licencí GNU GPL.

Ukažme si práci s ALD na programu, který převáděl malá písmena na velká.

Debugger spustíme příkazem ald.

```
ruik:Asm_lab$ ald
Assembly Language Debugger 0.1.3
Copyright (C) 2000-2002 Patrick Alken

ald>
```

Program preved nahrajeme příkazem load:

```
ald> load preved
echo: ELF Intel 80386 (32 bit), LSB, Executable, Version 1 (current)
Loading debugging symbols...(no symbols found)
ald>
```

Základním příkazem je stejně jako u ostatních debuggerů příkaz **s** (step), který provede jednu instrukci programu a vrátí řízení debuggeru:

```
ald> s
eax = 0x00000000 ebx = 0x00000000 ecx = 0x00000000 edx = 0x00000000
esp = 0xBFFFF8CC ebp = 0x00000000 esi = 0x00000000 edi = 0x00000000
ds = 0x0000002B es = 0x0000002B fs = 0x00000000 gs = 0x00000000
ss = 0x0000002B cs = 0x00000023 eip = 0x08048082 eflags =0x000000346

Flags: PF ZF TF IF


08048082 5A              pop edx
```

Následující instrukce, která se bude provádět, je instrukce pop edx. Leží na adrese 0x8048082. Registr příznaků má nastaven příznak ZF (a ostatní, o kterých jsme nemluvili). Chceme-li provést stejný příkaz, stačí stisknout klávesu Enter (která zopakuje poslední napsaný příkaz). Přesuňme se v programu dále, až před systémové volání int 0x80, které bude načítat řetězec ze standardního vstupu (EAX = 0x00000003).

```
ald>
eax = 0x00000003 ebx = 0x00000000 ecx = 0x080490C8 edx = 0x0000000A
esp = 0xBFFFF8D0 ebp = 0x00000000 esi = 0x00000000 edi = 0x00000000
ds = 0x0000002B es = 0x0000002B fs = 0x00000000 gs = 0x00000000
ss = 0x0000002B cs = 0x00000023 eip = 0x0804808D eflags =0x00000346

Flags: PF ZF TF IF


0804808D CD80              int 0x80
```


Hodnotu v registru EDX limituje délka načítaného řetězce pro jedno čtení na deset znaků (byte). V registru ECX se předává ukazatel na připravenou paměť, kam se řetězec uloží, kterou si můžeme prohlédnout příkazem **e** (examine) e ecx:

```
ald> e ecx
Dumping 64 bytes of memory starting at 0x080490C8 in hex
080490C8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
080490D8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
080490E8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
080490F8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Instrukci int 0x80 vykonáme příkazem **s** a tak začneme načítat řetězec (nějaký napište). Zopakujeme-li příkaz e ecx, můžeme si přečtený řetězec prohlédnout, adresa bude pořád uložena v registru ECX, protože jiné registry systémové volání nemodifikuje.

```
ald> e ecx
Dumping 64 bytes of memory starting at 0x080490C8 in hex
080490C8: 61 73 6D 20 72 75 6C 65 7A 0A 00 00 00 00 00 00 asm rulez.......
080490D8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
080490E8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
080490F8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Příkazem **c** (continue) vykonáme zbytek programu, který převedený řetězec vypíše a bude připraven načítat řetězec další (program lze násilně přerušit stiskem kláves Ctrl+c).

Příkazem help dostaneme jména všech použitelných příkazů; pokud zadáme název příkazu jako parametr příkazu help, dostaneme kompletní nápovědu. Uveďme si nejpoužívanější příkazy v tabulce.

Příkaz	Význam
load <filename>	Nahraje soubor do debuggeru.load echo.
set args <args>	Nastav parametry spouštěného programu set args, moje parametry
step [n]	Provede jednu instrukci (nebo lze nepovinně uvést počet instrukcí, které se budou provádět). Místo příkazu step stačí napsat příkaz s.
next [n]	Krokuje instrukce podobně jako step, vykonávání podprogramů přeskočí. Nepovinným parametrem n můžeme uvést počet kroků.
disassemble	Převede instrukce ze strojového kódu zpět do Assembleru. Nejčastějším parametrem je adresa, od které si budeme přát program vypisovat (alias d)d 0x08048061.
continue	Dokončí prováděný program (alias c).
examine	Vypíše data z paměti v požadovaném formátu a číselné soustavě. Nejčastějším parametrem je registr nebo přímo adresa dat, která se mají vypsat, e edxe 0x08048000.
register	Příkazem register vypíšeme hodnoty registrů procesoru.
help	Příkazem help získáme seznam všech dostupných příkazů. Jako nepovinný parametr můžeme uvést jméno příkazu, ke kterému si přejeme podrobnější nápovědu help examine.
break <addr>	Nastaví breakpoint (zarážku) na adresu adr.
lbreak	Vypíše všechny breakpointy.
quit	Ukončíme práci s debuggerem (alias q).

ALD podporuje nastavení tzv. breakpointů. Dostane-li se vykonávaný program na určité místo (specifikované právě breakpointem), dočasně se přeruší a řízení se navrátí zpátky ladicímu programu. Tak můžeme prohlédnout nějaké klíčové místo v programu (nebo v paměti), aniž bychom museli krokovat celý program.

Současná verze 0.1.3 si neporadí se symbolickou informací obsaženou v souborech (symbolická informace pomáhá zpětně přiřadit symbolická jména adresám ve spustitelném souboru). Díky H-Petru Recktenwaldovi a jeho vylepšení z http://www.lxhp.in-berlin.de/lhplinks.html#ald můžeme zpětně překládat adresy na symboly a tak práci s ALD zpříjemnit. Pak lze místo adres používat znovu názvy symbolů (navěští, jména proměnných), tak jak jsme je zapsali ve zdrojovém kódu. Všude, kde bude možné adresu převést zpět na identifikátor (symbol použitý ve zdrojovém kódu), bude tak učiněno.

Symbolickou informaci musíme do objektového souboru „přibalit" pomocí přepínače (parametru) nasm -g. (Používáme-li Asmutils, musíme v souboru MCONFIG zadat řádek DEBUG=y)

12.15 Assembler GAS

Seznamme se krátce s původním Assemblerem v unixovém světě – Assemblerem GAS. Jeho hlavním posláním je sloužit překladači jazyka C gcc. Jeho funkce je podstatná, produkovaný kód překladačem GCC (v Assembleru) překládá do strojového tvaru.

Kvůli těsné spolupráci s překladačem GCC není tak rozvinutá podpora práce s makry jako v jiných Assemblerech, stejně tak jeho chybová hlášení jsou strohá.

Syntaxe Assembleru GAS je odlišná od syntaxe, kterou jsme používali společně s překladačem NASM. Překladače typu NASM (MASM, TASM) používají syntaxi typu Intel, překladač gas však používá syntaxi typu AT&T, která vyšla z jiných typů procesorů než Intel. Ukažme si program Hello, World! zapsaný v syntaxi typu AT&T, tak jak je předveden v Linux-Assembly-HOWTO:

```
.data                          #začátek datové sekce

msg:

.ascii "Hello, world!\n"       # řetězec, co budeme tisknout
len = . - msg                  #jeho délka

.text                          # začátek kódové sekce

                               # návěštím _start
.global _start                 # označíme místo, odkud se program
                               # začne provádět.


_start:
                               # zapíšeme na stdout
movl $len,%edx                 # třetí parametr - délka řetězce
movl $msg,%ecx                 # druhý - ukazatel na řetězec
movl $1,%ebx                   # první - identifikační číslo souboru
                               # - STDOUT - 1
movl $4,%eax                   # číslo systémového volání write
int $0x80                      # zavoláme jádro

                               # a končíme
movl $0,%ebx                   # do EBX návratový kód
movl $1,%eax                   # do EAX číslo sys. volání exit
int $0x80                      # a zavoláme jádro
```

12.16 Volně šiřitelné zdroje informací

Pro programátora v Assembleru pod jiné operační systémy než DOS a Windows (Linux, FreeBSD apod.) je nejvhodnější informační server http://linuxassembly.org, jehož česká verze je k dispozici na http://www.assembler.cz. Najdeme zde odkazy nejen na programy (překladače, editory), ale i návody a dokumentaci systémových volání. Linuxassembly.org poslouží i jako rozcestník k dalším zajímavým stránkám zabývajícím se problematikou Assembleru a Unix-like operačních systémů.

Na stránkách linuxassembly.org je odkaz do internetové konference linux-assembly (vedené anglicky), kde vám jistě poradí s řešením vašich problémů. Pozor, nesnažte se zneužívat dobroty jiných lidí a požadovat řešení vašich domácích úkolů.

Nejpřesnější dokumentací programátora v Assembleru jsou bezpochyby zdrojové kódy jádra operačního systému, ve kterých najdeme odpovědi na všechny otázky týkající se poskytovaných služeb operačního systému. Selže-li i tato cesta, pak zbývá jedině prohledat Internet, např. s pomocí vyhledávače Google na http://google.com.

Možnosti programování v Assembleru nejsou omezeny na pouhou práci se soubory. V Asmutils naleznete i programy pracující se sítěmi. Kde jinde byste mohli najít nejmenší web server na světě, jehož délka po překladu je pouhých 514 byte? (httpd.asm)

12.17 Možnosti překladače a linkeru

Uveďme si přehledně v tabulce nejčastěji používané přepínače překladače NASM:

Přepínač	Význam
-v	Vypiš číslo verze překladače NASM (ve starších verzích je použit přepínač -r)
-g	„Přibal" symbolické informace
-f <fmt>	Nastav výstupní formát souboru na fmt (viz kapitola věnovaná překladači NASM)
-fh	Vypiš seznam všech podporovaných výstupních formátů
-o <name>	Pojmenuj výstupní soubor podle parametru name
-I <path>	Přidej adresář, kde se budou hledat „přibalované" soubory

I pro sestavovací program ld

Přepínač	Význam
-o <name>	Výstupní spustitelný program se bude jmenovat podle parametru name
-s	Odstraň symbolickou informaci ze spustitelného souboru

PROPOJUJEME

13

Dosud jsme psali programy bez použití knihovních funkcí, používali jsme pouze služeb operačního systému. V kapitole „Propojujeme" si ukážeme integraci Assembleru s vyššími programovacími jazyky nebo knihovnami. Omezíme se na programovací jazyk C a Pascal, abychom si příliš nekomplikovali život s pokročilými konstrukcemi v moderních programovacích jazycích.

V této kapitole:

13.1 Předávání proměnných
13.2 Co je stackframe?
13.3 Propojujeme s jazykem C
13.4 Propojujeme s jazykem Pascal

13.1 Předávání proměnných

Ve vyšších programovacích jazycích (uvažujeme Pascal a C) můžeme jednotlivé proměnné podprogramům (funkcím nebo procedurám) předávat hodnotou nebo odkazem. Proměnné se nepředávají „přes registry", ale v určitém pořadí se uloží na zásobník. V podprogramu se s takto předanými parametry pracuje jako s proměnnými uloženými v hlavní paměti.

Z pohledu programátora v Assembleru jsou podprogramy utvářeny stejně užitím instrukcí CALL a RET. Před instrukcí CALL následuje většinou několik instrukcí PUSH, které jednotlivé parametry (nebo ukazatele) uloží na zásobník. Po vstupu do podprogramu (instrukcí CALL) se na zásobníku vyhradí místo pro lokální proměnné a mezivýsledky (často se místo vyhradí jen kvůli zarovnání adres zásobníku na mocniny 2). O jednotlivých parametrech i lokálních proměnných je známo již v době překladu, v jaké relativní vzdálenosti od vrcholu zásobníku se nacházejí, a proto se k jednotlivým parametrům přistupuje jednoduše prostřednictvím instrukce MOV.

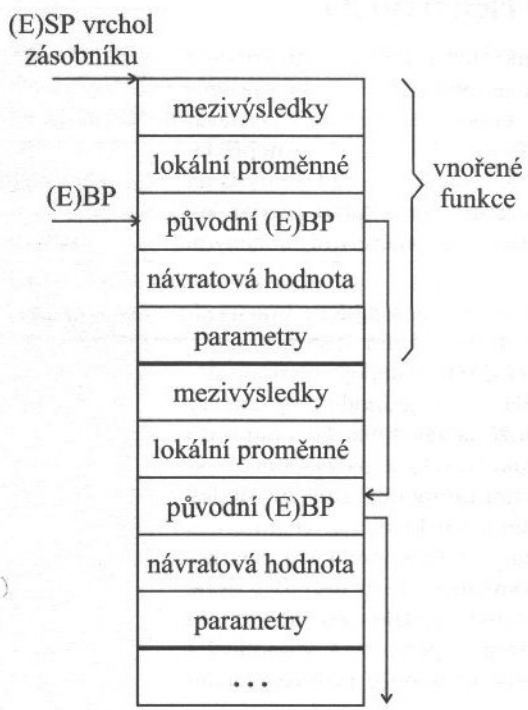
Místo pro lokální proměnné (nebo i zarovnání) získáme na zásobníku prostým odečtením požadované velikosti od ukazatele na vrchol zásobníku (E)SP. Zásobník roste směrem dolů (k nižším adresám), odečtením získáme prostor v paměti, kam lze dočasné lokální proměnné uložit. Původní obsah paměti se nijak neupravuje, a proto jsou počáteční hodnoty lokálních proměnných podprogramů nedefinované.

Na konci podprogramu se zruší místo po lokálních proměnných (přičtením instrukcí ADD) a instrukcí RET se podprogram opustí. Dříve uložené parametry podprogramu se vyjmou buď užitím instrukce RET s výmazem zásobníku (např.: ret 8...) nebo až po opuštění podprogramu instrukcí ADD, která zvýší ukazatel na vrchol zásobníku o potřebný počet byte, uložené parametry podprogramu z něj tak „vymaže".

Různé programovací jazyky používají různý protokol předávání proměnných přes zásobník. Seznámíme se s konvencí jazyka C a Pascal.

13.2 Co je stackframe?

Přistupovat k lokálním proměnným přímo přes ukazatel na vrchol zásobníku (E)SP se ukázalo jako nevýhodné (komplikace s proměnnou délkou mezivýsledků). Snazší by bylo poznamenat si do nějakého dalšího registru adresu (offset) na zásobníku těsně po vstupu do volaného podprogramu a tak adresovat všechny parametry i lokální proměnné relativně k této adrese. Vhodný se ukázal registr (E)BP.



Obrázek 47 Rozložení zásobníku (stackframe)

Proto je ihned po vstupu do podprogramu původní hodnota registru (E)BP uložena na zásobník a registr (E)BP naplněn hodnotou ukazatele na vrchol zásobníku (E)SP. Následuje vyhrazení místa na zásobníku pro lokální proměnné a mezivýsledky, ke kterým se stejně jako k parametrům podprogramu přistupuje relativně vzhledem k obsahu registru (E)BP.

Zásobníkovému oknu, které se skládá s parametrů, návratové adresy, lokálních proměnných a původní hodnoty (E)BP se říká anglicky stackframe.

Čím hlouběji se program ve svých podprogramech zanoří tím více je na zásobníku takových oken. Původní hodnota registru (E)BP uložená na zásobníku je ukazatelem na předchozí stackframe. Vzniká tak spojový seznam jednotlivých zásobníkových oken, které můžeme využít v ladicím programu pro výpis aktuálního zanoření funkcí v programu.

13.2.1 Stackframe v jazyce C (32bitová verze)

Ukažme si předávání proměnných funkcím v jazyce C. Uvažujme následující program, přeložený ve dvaatřicetibitovém prostředí (OS Linux).


```
int secti(int a ,int b) {
int d = a + b;
return d;

}
int main(void) {
int e
e = secti(0x55,0xAA);
}
```

Program secti.c poněkud nešikovně sčítá hodnoty 0x55 a 0xAA pomocí podprogramu secti. Podívejme se na výpis funkce main v Assembleru:

```
080483F4    55              push ebp
080483F5    89E5            mov dword ebp, esp
080483F7    83EC18          sub dword esp, 0x18
080483FA    83C4F8          add dword esp, 0xf8
080483FD    68AA000000      push 0xaa
08048402    6A55            push 0x55
08048404    E8C7FFFFFF      call near +0xffffffc7
08048409    83C410          add dword esp, 0x10
0804840C    89C0            mov dword eax, eax
0804840E    8945FC          mov dword [ebp+0xfc], eax
08048411    89EC            mov dword esp, ebp
08048413    5D              pop ebp
08048414    C3              retn
```

Funkce main začíná instrukcí push ebp, která uloží původní hodnotu ukazatele na předchozí okno (stackframe) na zásobník. Druhá instrukce uloží do registru EBP stávající hodnotu registru ESP a tak označí začátek nového zásobníkového okna. Dvěma ná-

sledujícími instrukcemi SUB a ADD se od adresy vrcholu zásobníku odečtou hodnoty 0x18 a 0x8 (druhá instrukce přičítá zápornou hodnotu -8 0xF8 = -8), vyhradí se tak místo pro lokální proměnnou e a místo pro mezivýsledky a zarovnání.

Parametry se podprogramu secti předávají zprava doleva, tedy v opačném pořadí, než jsme je zapsali ve zdrojovém kódu. Důvodem je podpora funkcí s neurčeným počtem parametrů (operátor ...) v jazyce C. Dvojicí instrukcí PUSH se předávané parametry hodnotou uloží na zásobník (0x55 a 0xAA). Následuje instrukce CALL, která zavolá podprogram secti.

Ukažme si, jak vypadá zásobník těsně po vstupu do podprogramu secti.

Adresa v zásobníku	Hodnota uložená na adrese
0xBFFFF860 = ESP	0x08048409 (návratová adresa zpět do funkce main)
0xBFFFF864	0x00000055
0xBFFFF868	0x000000AA
0xBFFFF86C - 0xBFFFF84	prostor dočasných proměnných funkce main (nebo zarovnání)
0xBFFFF888	???????? - lokální proměnná e
0xBFFFF88C = EBP	0xBFFFF8C8 původní EBP nějaké knihovní funkce libc
0xBFFFF890	0x40039CA2 - návratová adresa volající funkce LIBC

Na vrcholu zásobníku je uložena dvaatřicetibitová návratová adresa zpět do funkce main (samozřejmě, že ukazuje těsně za instrukci call). Následují dvě dvojslova uložených parametrů. Další adresy nejsou v programu nijak využité a slouží pouze k zarovnání zásobníku. Za výplní je připraveno místo pro lokální proměnnou e, jejíž hodnota není v tuto chvíli definovaná. Ihned za proměnnou e následuje původní obsah registru (E)BP, platný před voláním funkce main (předcházející zásobníkové okno). Na následující adrese je uložena dvaatřicetibitová návratová adresa kamsi do knihovny libc, která zavolala funkci main. Další položky budeme prozatím ignorovat.

Ukažme si výpis funkce secti:

```
080483D0    55                  push ebp
080483D1    89E5                mov dword ebp, esp
080483D3    83EC18              sub dword esp, 0x18
080483D6    8B4508              mov eax, dword [ebp+0x8]
080483D9    8B550C              mov edx, dword [ebp+0xc]
080483DC    8D0C02              lea ecx, [eax+edx]
080483DF    894DFC              mov dword [ebp+0xfc], ecx
080483E2    8B55FC              mov edx, dword [ebp+0xfc]
080483E5    89D0                mov dword eax, edx
080483E7    EB07                jmp short +0x7
080483E9    8DB42600000000      lea esi, [esi+0x0]
080483F0    89EC                mov dword esp, ebp
080483F2    5D                  pop ebp
080483F3    C3                  retn
```

Funkce secti začíná stejně. Hodnota registru EBP, ukazující na zásobníkové okno funkce main, se uloží na zásobník. Do registru EBP se uloží stávající hodnota registru ESP a tak se založí zásobníkové okno (stackframe) funkce secti.

Na zásobníku se vyhradí místo pro lokální proměnnou d (instrukcí SUB), do které se později uloží výsledek součtu a+b. Ukažme si výpis zásobníku programu těsně za instrukcí sub (tj. před vykonáním prvního příkazu podprogramu).

Adresa v zásobníku	Hodnota uložená na adrese
0xBFFFF844 = ESP	???????? - nedefinovaná - zarovnání
0xBFFFF848- 0xBFFFF854	???????? - nedefinovaná - zarovnání
0xBFFFF858 = EBP - 4 (+0xFFFFFFFC)	???????? - místo pro lokální proměnnou d
0xBFFFF85C = EBP	0xBFFFF88C - původní EBP funkce main (ukazatel na zásobníkové okno)
0xBFFFF860 = EBP + 4	0x08048409 (návratová adresa zpět do funkce main)
0xBFFFF864 = EBP + 8	0x00000055
0xBFFFF868 = EBP + 0xC	0x000000AA
0xBFFFF86C - 0xBFFFF84	prostor dočasných proměnných funkce main (nebo zarovnání)
0xBFFFF888	???????? - místo pro lokální proměnnou e
0xBFFFF88C	0xBFFFF8C8 původní EBP nějaké knihovní funkce libc
0xBFFFF890	0x40039CA2 - návratová adresa volající funkce LIBC

Několik dvojslov zásobníku tvoří, stejně jako v předchozím případě, pouze výplň. Na offsetu 0xBFFFF858 je vyhrazeno jedno dvojslovo pro lokální proměnnou d, jejíž hodnota bude výsledek součtu parametrů a + b. Následující adresa (offset) na zásobníku má shodnou hodnotu jako offset uložený v registru EBP. Na tomto paměťovém místě je uložen původní offset registru EBP, který se používal k relativní adresaci parametrů i lokálních proměnných ve funkci main. Se zbytkem tabulky jsme se již seznámili v předchozím textu. Následující instrukce budou načítat parametry funkce secti do registrů EAX a ECX.

```
080483D6    8B4508          mov eax, dword [ebp+0x8]
080483D9    8B550C          mov edx, dword [ebp+0xc]
```

Z popisu zásobníkového okna je již jasné, že nejbližší parametr předávaný podprogramu bude ležet o osm byte (dvě položky) od hodnoty uložené v registru EBP.

Shrňme si, které adresy jsou z programátorského hlediska zajímavé (relativně k obsahu registru EBP)

Obsah adresy	Význam
[ebp – 4]	místo pro lokální proměnnou d
[ebp + 0]	původní hodnota registru EBP (původní adresa zásobníkového okna)
[ebp + 4]	návratová adresa
[ebp + 8]	první parametr
[ebp + 0xC]	druhý parametr

První parametr funkce secti byl 0x55, a protože byl na zásobníku uložen později (jako poslední předávaný parametr), je relativně k stávající hodnotě registru EBP blíže.

Zbývající část podprogramu secti, provede součet obou hodnot nahraných v registrech EAX a EDX, překladač použil instrukci LEA. Má-li funkce výstup, očekává se v registru EAX. Na konci funkce secti se zásobníkové okno zruší. Instrukcí mov esp,ebp se zruší lokální proměnné a zarovnání, pop ebp se obnoví původní zásobníkové okno funkce main. Odstranění parametrů ze zásobníku je povinností volajícího (funkce) main, neboť pouze volající zná skutečný počet parametrů uložených na zásobníku.

Relativní vzdálenost jednotlivých parametrů předávaných přes zásobník, vzhledem k hodnotě registru EBP, je závislá na typu předávaného parametru. Typy menší než 1 dvojslovo (4 byte) jsou zarovnány na jedno dvojslovo (char, short).

13.2.2 Stackframe v jazyce C (16bitová verze)

Vše, co jsme si zatím řekli o zásobníkovém okně, platí samozřejmě i pro šestnáctibitové platformy jazyka C. Nejmenší uložitelnou hodnotou na zásobníku nebude 1 dvojslovo, ale jen jedno slovo (2 byte). Registry budeme předpokládat pouze v šestnáctibitových variantách (BP místo EBP, SP místo ESP atd.), výstupní hodnota funkce se nebude předávat v registru EAX, ale v párovém registru DX:AX. Nikdo nám nebrání v našem podprogramu používat dvaatřicetibitové instrukce, jsme-li si jistí, že je máme k dispozici (pokud programy nespouštíme na historické 80286, tak určitě).

V šestnáctibitovém světě se dostaneme do problémů kvůli rozdílnému typu volání podprogramů. V programovacím jazyce C byly vymyšleny tzv. paměťové modely, které specifikovaly kolik segmentů (datových a kódových) může program maximálně obsahovat. V paměťových modelech medium, large a huge jsou generována volání typu FAR, ve zbývajících modelech jsou generována volání typu NEAR.

Volání typu near uloží na zásobník jen offset následující instrukce, segmentovou část adresy nemění a ani neukládá na zásobník. Na zásobníku pak bude „vzdálenost" od obsahu registru BP k prvnímu parametru o 2 byte „kratší" než při použití volání typu FAR, které při volání ukládá na zásobník i segmentovou část adresy. I návratovou instrukci ret budeme muset při volání typu FAR změnit na retf.

Jako součást překladače NASM jsou dodávána makra, která řeší nepříjemnou situaci s různými paměťovými modely i makra pro snazší propojení s jazyky C a Pascal. Více si povíme v následující kapitole.

13.3 Propojujeme s jazykem C

Většina kompilátorů jazyka C doplňuje před všechny globální symboly z jmenného prostoru jazyka C podtržítko. Funkce printf bude pro programátora v Assembleru dostupná pod symbolem _printf. Výjimku tvoří formát ELF, běžně používaný v OS Linux, který podtržítko před jménem nevyžaduje.

Vytvořme jednoduchý program v jazyce C, který bude volat funkci vypis. Funkci vypis implementujeme v Assembleru a bude vypisovat součet globální proměnné s jejím jediným parametrem typu int. Volat bude knihovní funkci printf.

Příklad implementujeme v prostředí OS Linux.

Část zapsaná v jazyce C bude snadná.

```
const int plus = 6;

void vypis(int);

int main(void) {
vypis(5);
 }
```

Nadefinujeme globální konstantu plus, které přiřadíme hodnotu 6. Nadeklarujeme prototyp funkce vypis (patří spíše do hlavičkového souboru). Nakonec ve funkci main zavoláme funkci vypis s parametrem 5.

V „assemblerovské" části příkladu si musíme nejprve zpřístupnit globální proměnné plus a printf.

```
extern plus
extern printf
```

Používáme překladač gcc s výstupním formátem ELF, proto nemusíme s podtržítky nijak manipulovat.

```
%include "misc/c32.mac"
```

Direktivou include si zpřístupníme makra proc, arg a endproc, která nám ulehčí konstrukci podprogramu vypis.

```
proc vypis
%$co arg
```

Makrem proc označíme začátek funkce vypis. Makro provede sekvenci push ebp, mov ebp,esp. Dalším makrem arg nadefinujeme náš jediný parametr (jestliže by bylo parametrů více, první parametr arg bude nejlevější z pohledu jazyka C). Není-li u makra arg uveden jeho nepovinný parametr – velikost parametru uloženého na zásobníku – uvažuje se automaticky 4 (pro c32.mac). Zbytek programu bude snadný:

```
mov eax,[ebp + %$co]        ;ze zásobníku přečteme 1. parametr
add eax,[plus]              ;jeho hodnotu přičteme ke globální proměnné
                            ;plus
push eax                    ;na zásobník uložíme poslední parametr
                            ;funkce printf
push str1                   ;prvním bude formátovaný řetězec str1
call printf                 ;zavoláme printf

endproc                     ;makro endproc obnoví původní EBP, zruší
                            ;lokální proměnné (žádné nebyly) a opustí
                            ;podprogram
```

Celý program včetně jeho datové sekce bude vypadat takto:

```
%include "misc/c32.mac"     ;nahraje soubor s makry
section .text               ;začátek kódové sekce
extern plus                 ;zpřístupníme globální symbol plus
extern printf               ;i printf
global vypis;               ;funkci vypis uděláme "public"

proc vypis                  ;makrem proc začne funkce vypis
%$co arg                    ;pojmenujeme náš jediný parametr jako
                            ;"co"
mov eax,[ebp + %$co]        ;uložíme hodnotu parametru do registru
                            ;EAX
add eax,[plus]              ;přičteme hodnotu globální konstanty plus
push eax                    ;na zásobník uložíme poslední parametr
                            ;funkce printf
push str1                   ;prvním bude formátovaný řetězec str1
call printf                 ;zavoláme printf

endproc                     ;makro endproc obnoví původní EBP, zruší
                            ;lokální proměnné (žádné nebyly) a opustí
                            ;podprogram
section .data               ;začátek datové sekce

str1 db "Vysledek je %d, coz je snad spravne.",0x0A,0x0
                            ;vytisknutý text
```

Soubor pojmenujeme vypis.asm a přeložíme nasm -f elf vypis.asm. Zdrojový kód v jazyce c pojmenujeme vypisc.c a přeložíme i s modulem vypis.o.

```
gcc -o vypisc vypisc.c vypis.o
```

Po spuštění programu vypisc dostaneme správnou hodnotu součtu 11.

Šestnáctibitová varianta příkladu bude mít jen zmenšené registry a jiný soubor s makry – c16.mac. Problémy s near a far jsou díky makrům proc a endproc již vyřešeny. V případě, že používáme paměťový model generující FAR volání, stačí, když nadefinuje symbol FARCODE (%define FARCODE), a makra se postarají o zbytek.

Používáme-li překladač vyžadující podtržítka ve jménech symbolů, můžeme si pomoci makry, která nám podtržítka přidají.

```
%macro cglobal 1
global _%1
%define %1 _%1
%endmacro

%macro cextern 1
extern _%1
%define %1 _%1
%endmacro
```

Pak můžeme místo direktiv global a extern používat makra cglobal a cextern, která problémy s podtržítky odstraní (nebudou se stejně jako v případě ELF formátu vyžadovat u všech globálních symbolů ve zdrojovém programu).

13.4 Propojujeme s jazykem Pascal

Předávání proměnných v jazyce Pascal se principiálně neliší od předávaní proměnných v jazyce C. Parametry procedurám nebo funkcím se předávají přes zásobník, podprogramy používají zásobníkové okno (jeho 16bitovou verzi). K proměnným i parametru se přistupuje přes registr BP.

Jazyk Pascal byl navržen jako výukový jazyk, a proto nepodporuje některé konstrukce z jazyka C, jako například předávání libovolného počtu parametrů. Jazyk Pascal proto ukládá na zásobník své parametry zleva doprava, a ne opačně jako jazyk C. Pascal ulehčuje programátorovi v Assembleru volání externích podprogramů (v objektovém souboru), neboť užívá pouze volání typu far. O výmaz parametrů ze zásobníku se nestará volající, ale volaná funkce, která musí instrukcí retf s výmazem parametry ze zásobníku odstranit. Ukažme si rozložení zásobníku v podprogramu relativně k registru BP.

	Význam
[BP-...]	Lokální proměnné
[BP + 0]	Původní hodnota registru BP (2 byte)
[BP + 2]	Návratová adresa - registr IP (2 byte)
[BP + 4]	Návratová adresa - registr CS (2 byte)
[BP + 6]	První parametr
[BP + ...]	Další parametry

Ukažme si stejný program, sčítající dvojici čísel pomocí funkce secti. Tentokrát necháme tisk na funkci writeln, kterou zavoláme přímo z programu jazyka Pascal.

```
{$L secti.obj}
uses crt;
var plus:integer;

function secti(x:integer):longint;far;external;

begin
plus := 6;
writeln('Vysledek bude: ', secti(5),' Coz je snad spravne');
end.
```

Funkci secti, kterou budeme implementovat v Assembleru, „představíme" Pascalu klíčovým slovem external. Zároveň Pascal požádáme, aby k této funkci vždy přistupoval typem volání far. Nesmíme zapomenout na direktivu $L, kterou překladači oznámíme název objektového souboru s přeloženou funkcí secti. Zbytek programu je jednoduchý. Výsledkem bude součet čísel 5 a 6, tedy 11.

Zbývá naprogramovat funkci secti, kterou uložíme do souboru secti.asm. Turbo Pascal firmy Borland neužívá jako svůj objektový formát klasický „obj", ale vlastní formát. Výsledkem jsou striktní požadavky překladače na jména jednotlivých sekcí objektového souboru. Kódová sekce programu se musí jmenovat CODE nebo CSEG, dovolena je koncovka jména _TEXT. Datová sekce se smí jmenovat buď CONST anebo její jméno musí končit _DATA. Konečně sekce pro neinicializované proměnné se musí jmenovat DATA, DSEG anebo aspoň končit _BSS.

V našich programech jsme doposud pojmenovávali jednotlivé sekce .text, .data a .bss. V programech určených Pascalu přidáme za tečku podtržítko a tak uspokojíme potřeby překladače.

Funkce secti bude volána v šestnáctibitovém prostředí, kde je velikost typu integer právě 2 byte. Návratová hodnota funkce secti je deklarována jako longint, jež se vrací v párových registrech DX:AX. Na pomoc si pozveme makra ze souboru c16.mac, s kterými jsme se seznámili již dříve.

```
SECTION ._TEXT              ;začátek kódové sekce
%define PASCAL              ;používáme volání typu FAR
%include "misc/c16.mac"     ;přibalíme makra
extern plus                 ;zpřístupníme symbol plus
global secti                ;zveřejníme funkci secti

proc secti                  ;definujeme začátek funkce (včetně
                            ;vytvoření nového zásobníkového okna)
%$co arg                    ;Funkce vyžaduje jeden parametr,
                            ;zpřístupníme si ho pod jménem co
xor dx,dx                   ;vynulujeme DX
mov ax,[bp+%$co]            ;do AX uložíme hodnotu parametru co
add ax,[plus]               ;přičteme globální proměnnou plus
adc dx,0                    ;nezapomeneme na příznak přenosu
                            ;který "uložíme" do DX
endproc                     ;makro endpoc "vymaže" předávané
                            ;parametry a opustí podprogram.
```

Makrem arg přidělíme symbolu co „relativní vzdálenost" mezi hodnotou uloženou v registru BP a prvním parametrem. Makro arg akceptuje jako svůj nepovinný parametr velikost předávaného parametru (kolik byte na zásobníku zabírá). Výchozí hodnota pro šestnáctibitové prostředí je 2, a proto nemusíme nic měnit. Pokud bychom předávali parameter typu pointer nebo longint, musíme makru arg předat správnou velikost parametru, a to 4.

Ukažme si rozvinutá makra ve výpisu funkce secti:

```
cs:0076    55          push    bp              ;makro proc
cs:0077    89E5        mov     bp,sp           ;makro proc
cs:0079    31D2        xor     dx,dx           ;vynulujeme DX
cs:007B    8B4606      mov     ax,[bp+06]      ;vybereme první
                                               ;parametr
cs:007E    03065200    add     ax,[0052]       ;přičteme k
                                               ;proměnné plus
```

```
cs:0082     81D20000        adc  dx,0000          ;přičteme s carry
cs:0086     89EC            mov  sp,bp            ;makro endproc
cs:0088     5D              pop  bp               ;makro endproc
cs:0089     CA0200          retf  0002            ;makro endpoc,
                                                  ;ukončíme funkci
```

Po zkompilování a spuštění provede program součet a vypíše ho na obrazovku.

14

SLOVO ZÁVĚREM

Už je to tak, doputovali jsme společně až na konec. Ať už vás k Assembleru dovedl vlastní zájem nebo nutnost, doufám, že jste po přečtení této knihy nějaké ty zkušenosti a vědomosti získali.

Ty z vás, kteří snad v knize nenalezli odpověď na složitější otázky a problematiku, mohu jen odkázat na nějakou z dalších knih, která se bude věnovat „Assembleru pro pokročilé". Mohli bychom se seznámit s chráněným režimem procesoru, s instrukčními rozšířeními (MMX, SSE, 3DNow) nebo se zaměřit na programování periferních zařízení. Věnovat se můžeme i pokročilejšímu programování v Assembleru pod operačním systémem Linux.

Rád bych znovu poděkoval všem svým kolegům, kteří mi s knihou pomáhali. Zdravím také své spolužáky a učitele z ČVUT FEL.

Zbývá udělat radost právníkům a jiným šťouralům :)

Všechny programy (i části), které jsou v knize uvedeny, jsou volně šiřitelné (public domain) bez dalších licenčních omezení. Materiály na disku CD (včetně programů či příkladů) musí být šířeny podle individuálních licenčních ujednání, tak jak je uvedeno v souborech LICENCE nebo podobných.

Registrované ochranné známky jsou vlastnictvím jejich držitelů.

14

SLOVO ZÁVĚREM

NEJČASTĚJI POUŽÍVANÉ INSTRUKCE

15

Instrukce:	
Přesunů	MOV – přesuň (kopíruj)
	XCHG – prohoď
	PUSH – ulož na zásobník
	POP – vyber ze zásobníku
Aritmetické	ADD – přičti
	SUB – odečti
	MUL – násob
	DIV – děl
	INC – přičti jedna
	DEC – odečti jedna
Logické	AND – logický součin
	OR – logický součet
	XOR – logická non-ekvivalence
	NOT – negace
Porovnání	CMP – porovnej
	TEST – proveď bitové porovnání
Posuvů	SHR – log. posuv vpravo
	SHL – log. posuv vlevo
Rotací	RCR – rotace přes příznak přenosu (carry) vpravo
	RCL – rotace přes příznak přenosu (carry) vlevo

Skoků	JMP – nepodmíněný skok LOOP – skoč dokud není (E)CX nula Skok když: JZ – nastaven příznak nuly (ZF) JC – nastaven příznak přenosu (CF) JNZ – není nastaven příznak nuly (ZF) JNC – není nastaven příznak přenosu (CF)
Podprogramů	CALL – zavolej podprogram RET – návrat z podprogramu INT – vyvolej přerušení
Řetězců	REP – opakuj následující instrukci pokud není (E)CX nula MOVSx – kopíruj CMPSx – porovnej SCASx – hledej

000	00		043	2B	+	086	56	V	129	81	ü	172	AC	¼	215	D7	╫
001	01	☺	044	2C	,	087	57	W	130	82	é	173	AD	¡	216	D8	╪
002	02	☻	045	2D	-	088	58	X	131	83	â	174	AE	«	217	D9	┘
003	03	♥	046	2E	.	089	59	Y	132	84	ä	175	AF	»	218	DA	┌
004	04	♦	047	2F	/	090	5A	Z	133	85	à	176	B0	░	219	DB	█
005	05	♣	048	30	0	091	5B	[	134	86	å	177	B1	▒	220	DC	▄
006	06	♠	049	31	1	092	5C	\	135	87	ç	178	B2	▓	221	DD	▌
007	07	•	050	32	2	093	5D	]	136	88	ê	179	B3	│	222	DE	▐
008	08	◘	051	33	3	094	5E	^	137	89	ë	180	B4	┤	223	DF	▀
009	09		052	34	4	095	5F	_	138	8A	è	181	B5	╡	224	E0	α
010	0A		053	35	5	096	60	`	139	8B	ï	182	B6	╢	225	E1	ß
011	0B	♂	054	36	6	097	61	a	140	8C	î	183	B7	╖	226	E2	Γ
012	0C	♀	055	37	7	098	62	b	141	8D	ì	184	B8	╕	227	E3	π
013	0D		056	38	8	099	63	c	142	8E	Ä	185	B9	╣	228	E4	Σ
014	0E	♫	057	39	9	100	64	d	143	8F	Å	186	BA	║	229	E5	σ
015	0F	☼	058	3A	:	101	65	e	144	90	É	187	BB	╗	230	E6	µ
016	10	►	059	3B	;	102	66	f	145	91	æ	188	BC	╝	231	E7	τ
017	11	◄	060	3C	<	103	67	g	146	92	Æ	189	BD	╜	232	E8	Φ
018	12	↕	061	3D	=	104	68	h	147	93	ô	190	BE	╛	233	E9	Θ
019	13	‼	062	3E	>	105	69	i	148	94	ö	191	BF	┐	234	EA	Ω
020	14	¶	063	3F	?	106	6A	j	149	95	ò	192	C0	└	235	EB	δ
021	15	§	064	40	@	107	6B	k	150	96	û	193	C1	┴	236	EC	∞
022	16	▬	065	41	A	108	6C	l	151	97	ù	194	C2	┬	237	ED	ø
023	17	↨	066	42	B	109	6D	m	152	98	ÿ	195	C3	├	238	EE	∈
024	18	↑	067	43	C	110	6E	n	153	99	Ö	196	C4	─	239	EF	∩
025	19	↓	068	44	D	111	6F	o	154	9A	Ü	197	C5	┼	240	F0	≡
026	1A	→	069	45	E	112	70	p	155	9B	¢	198	C6	╞	241	F1	±
027	1B	←	070	46	F	113	71	q	156	9C	£	199	C7	╟	242	F2	≥
028	1C	∟	071	47	G	114	72	r	157	9D	¥	200	C8	╚	243	F3	≤
029	1D	↔	072	48	H	115	73	s	158	9E	₧	201	C9	╔	244	F4	⌠
030	1E	▲	073	49	I	116	74	t	159	9F	ƒ	202	CA	╩	245	F5	⌡
031	1F	▼	074	4A	J	117	75	u	160	A0	á	203	CB	╦	246	F6	÷
032	20		075	4B	K	118	76	v	161	A1	í	204	CC	╠	247	F7	≈
033	21	!	076	4C	L	119	77	w	162	A2	ó	205	CD	═	248	F8	°
034	22	"	077	4D	M	120	78	x	163	A3	ú	206	CE	╬	249	F9	∙
035	23	#	078	4E	N	121	79	y	164	A4	ñ	207	CF	╧	250	FA	·
036	24	$	079	4F	O	122	7A	z	165	A5	Ñ	208	D0	╨	251	FB	√
037	25	%	080	50	P	123	7B	{	166	A6	ª	209	D1	╤	252	FC	ⁿ
038	26	&	081	51	Q	124	7C	|	167	A7	º	210	D2	╥	253	FD	²
039	27	'	082	52	R	125	7D	}	168	A8	¿	211	D3	╙	254	FE	■
040	28	(	083	53	S	126	7E	~	169	A9	⌐	212	D4	╘	255	FF	
041	29	)	084	54	T	127	7F	⌂	170	AA	¬	213	D5	╒			
042	2A	*	085	55	U	128	80	Ç	171	AB	½	214	D6	╓			

Obrázek 48 Rozvinutá tabulka kódu ASCII, včetně rozšířených znaků. (PC verze, bez znaků národních abeced)

REJSTŘÍK

A

ADC o1,o2, 42
ADD o1,o2, 40
adresace paměti, 33
– v reálném režimu, 134
ALIGN příkaz, 91
alokace paměti, 156, 157, 201
anatomie instrukce, 31
AND o1,o2, 51
aritmetické instrukce, 40, 48
ASCII, 19
asmutils – ulehčení práce, 189
assembler GAS, 204

B

BIOS, 119
bitové vlajky – pole (mapy), 53
breakpoint, 204

C

CALL typ_volání operandy, 70
CBW, 47
CDQ, 47
CISC (Complex Instruction Set Computer), 31
CLD, 76
CLI, 77
CMP o1,o2, 57
CMPSB, 78
CMPSD, 78
CMPSW, 78
CONFIG.SYS, 137
CWD, 47
CWDE, 47

Č

číselné soustavy a převody, 15
čísla
– desítková, 18
– dvojková, 18
– oktalová, 18
– se znaménkem, 44
– šestnáctková, 17
čtení ze souboru, 147, 193

D

datové typy, 18
debugger grdb.exe, 160
DEC o1, 43
deklarace makra – %assign, 124
desítková číselná soustava, 15
direktiva
– %assign, 124
– BITS, 126
– CPU, 128
– ORG, 130
direktivy
– Assembleru, 126
– EXTERN, GLOBAL a COMMON, 128
– SECTION a SEGMENT, 127
DIV r/m16, 50
DIV r/m32, 50
DIV r/m8, 50
druhy spustitelných souborů v DOSu, 137
dvojslovo, 19
DX offset, 145

E

EDVAC, 21
EMPTY, 193

ENIAC, 21
existence makra
– %ifdef, %ifndef, 125

F

formát ELF – executable
and linkable format, 184
fyzická struktura disku, 118

G

GETCWD, 155
grafické rozhraní (GUI), 177
grdb.exe – debugger, 160

H

Hello, World!, 140, 178, 188
historie mikroprocesorů řady x86, 23

CH

CHDIR, 155
chybové kódy, 159, 186

I

imm16 – operand, 33
imm32 – operand, 33
imm8 – operand, 33
INC o1, 43
instrukce
– ADD, 40
– AND, 51
– CALL, 70
– CLD, 76
– CLI, 75
– CMP, 56
– CMPSx, 78
– DEC, 43
– DIV, 50
– IDIV, 50
– IMUL, 48
– IN, 82
– INC, 43
– INT, 71
– IRET, 71
– jazyka symbolických adres, 34
– LEA, 76
– LODSx, 78
– LOOP, 64
– LOOPNZ, 65
– LOOPZ, 65
– MOV, 37
– MUL, 48
– nepodmíněného skoku JMP, 57
– NOP, 84
– NOT, 52
– OR, 52
– OUT, 82
– podmíněných skoků Jx, 58
– POP, 67
– posuvů a rotací, 84
– pracující se zásobníkem, 65
– pro práci s řetězci, 77
– pro vstup a výstup (I/O), 82
– pro záporná čísla – NEG CBW CWD CDQ CWDE, 46
– přímo měnící registr příznaků, 75
– PUSH, 67
– PUSHA/POPA, 69
– PUSHAD/POPAD, 69
– PUSHF/POPF, 49
– PUSHFD/POPFD, 69
– RCR/RCL, 87
– REP, 78
– REPZ, 78
– RET, 70
– ROL, 87
– ROR, 87
– SAL, 86
– SAR, 86
– SCASx, 78
– SHL, 85
– SHR, 85
– STD, 76
– STI, 75
– STOSx, 77
– SUB, 40
– TEST, 56
– XCHG, 76
– XOR, 52

INT op, 72
integrované obvody (chipy), 22
Intel, 23
inteligentní otevření/vytvoření souboru, 149
IOCTL Input/Output Control, 200
IRET, 72
iterace v programu, 62

J

jednoduché
– přidělování paměti, 113
– příklady, 95
jednořádková makra – %define, %undef, 123
jednotka správy paměti (MMU), 115
JMP typ_skoku operand, 57
Jx navěští_cíle_skoku, 58

K

kapacity paměťových médií, 19
kompatibilita programů, 180
konstrukce
– IF THEN – selekce, 56
– iterace – cykly, 62

L

LEA o1, [o2], 76
linker, 206
LODSB, 78
LODSD, 78
LODSW, 78
logické instrukce, 51
lokální návěští, 122
LOOP návěští_smyčky, 64
LOOPNZ návěští_smyčky, 65
LOOPZ návěští_smyčky, 65

M

m – operand, 33
magnetická záznamová média, 118
malý indián, 39
manuálové stránky, 186
master boot record (MBR), 119
mechanismy přístupu k souborům, 117
Microsoft OMF, 130
mikroprocesor
– 80386, 26
– obecně, 24
– Pentium, 23
MOV
– cíl, zdroj, 37
– instrukce, 37
možnosti překladače a linkeru, 206
MS-DOS, Microsoft Disk Operating System, 133
MUL r/m16, 48
MUL r/m32, 48
MUL r/m8, 48

N

naplňování registrů určitými hodnotami, 92
nápověda, 34
nativní aplikace pod Windows, 177
návrh a odladění programu, 163
NEG r/m16, 46
NEG r/m32, 46
NEG r/m8, 46
nejpravější bit (první bit), 18
NMI – nemaskovatelné přerušení, 29
NOP, 84
norma ASCII, 19
NOT r/m16, 52
NOT r/m32, 52
NOT r/m8, 52
nulový byte, 35

O

obecná instrukce, 34
obrácení pořadí bitů čísla, 98
odladění, 159, 202
operační systém, 109
operand
– imm16, 33
– imm32, 33
– imm8, 33
– instrukcí, 32
– m, 33
– r/imm8, 33
– reg16, 33

– reg32, 33
– reg8, 33
– reg8/imm8, 33
– segreg, 33
operátor SEG, 90
optimalizovaná aritmetika, 92
OR o1,o2, 52
organizace více souborových systémů na disku, 119
osmičková soustava, 16
osobní počítač IBM PC XT, 23
ostatní
– funkce pro práci se soubory, 152, 197
– instrukce, 75
otevření souboru, 145, 191

P

plánovací strategie Round robin, 112
podmíněný překlad – %if, 124
podprogram ASCIIToNum, 164
POPA, 69
POPF, 69
POPFD, 69
POST – power on self test, 119
práce
– s adresáři, 154, 198
– s dlouhými názvy souborů, 154
– s makry Asmutils, 191
preprocesor NASM, 123
princip plánování procesů, 111
proces control block (PCB), 111
procesy, 110
programová řádka, 121
programování
– v DOSu, 133
– v Linuxu, 183
– ve Windows, 177
propojování, 207
– s jazykem C, 212
– s jazykem Pascal, 215
předávání
– proměnných, 207
– prostředí, 185
– příkazové řádky, 158
překlad virtuální adresy na fyzickou, 114
překladač NASM, 121, 190
přerušení, 28
– IRQ 1, 171
převody
– čísel na řetězec, 99
– řetězců na čísla, 104
přidělování procesoru, 110
příkaz ALIGN, 91
pseudoinstrukce, 88
– DB, DW a DD, 88
– EQU, 90
– INCBIN, 90
– RESB, RESW a RESD, 89
– TIMES, 89
PUSH o1, 67
PUSHA, 69
PUSHF, 69
PUSHFD, 69

R

r/imm8 – operand, 33
RCL o1,o2, 87
RCR o1,o2, 87
reg16 – operand, 33
reg32 – operand, 33
reg8 – operand, 33
reg8/imm8 – operand, 33
registr
– DL, 103
– EAX, 27
REP, 78
reprezentace procesu, 111
REPZ, 78
RET, 70
rezidentní program, 171
RISC (Reduced Instruction Set Computer), 31
ROL o1,o2, 89
ROR o1,o2, 89
Round robin, 112
rozšíření paměti nad 1 MB, 136

S

SAL o1,o2, 86
SAR o1,o2, 86
SBB o1,o2, 42
SCASB, 78
SCASD, 78
SCASW, 78
segmentový registr
– DS, ES, FS a GS, 27
– ES, 158
segreg – operand, 33
sekvence, 55
sestavovací program, 206
seznámení s mikroprocesory řady x86, 21
SHL o1,o2, 85
SHR o1,o2, 85
služby API, 178
souborové systémy, 116
součet
– čísel uložených v registrech, 96
– dvou čísel uložených proměnných, 95
– pole čísel, 97
spouštění DOSu, 180
správa paměti, 113
srovnávání, 93
stackframe, 208
– v jazyce C (16bitová verze), 212
– v jazyce C (32bitová verze), 209
stavy procesu, 111
STD, 76
STI, 77
STOSB, 77
STOSD, 77
STOSW, 77
stránkování, 114
struktura souborového systému, 117
SUB o1,o2, 41
swapování, 114
symbolické informace, 132
systém UNIX, 110
systémová volání
– API, 178
– IOCTL, 200

Š

šestnáctková čísla, 17

T

tabulka ASCII, 20
test paměti, 119
triky s instrukcemi, 91

U

ukončení programu, 139
určení zda číslo sudé nebo liché, 98
uspořádání paměti
– procesu, 184
– v DOSu, 135
uvolnění paměti, 158
uzavření souboru, 146, 193
užitečné kousky kódu, 95

V

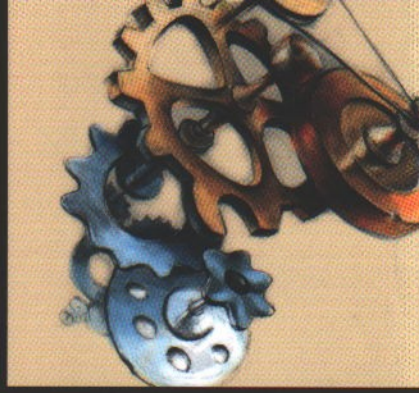
velký indián, 39
víceřádková makra – %macro %endmacro, 124
virtuální paměť, 114
vlož soubor – %include, 125
volání
– operačního systému, 185
– typu far, 215
volně šiřitelné zdroje informací, 175, 205
Von Neumannova koncepce počítače, 22
vstup z klávesnice, 141, 200
vstupně-výstupní operace se soubory (I/O), 145, 191
vyber aktuální adresář (CHDIR), 155, 199
vypsání řetězece, 140
výrazy, 122
výstupní formát, 129
– aout a aoutb, 132
– bin, 129
– coff, 131
– elf, 131
– obj (Microsoft OMF), 130
– vytvoř adresář (MKDIR, RMDIR), 154, 199
vývoj operačních systémů, 109

X

XCHG o1,o2, 76
XOR o1,o2, 52

Z

základní
– číselné soustavy a pojmy, 15
– instrukce, 37
– programové konstrukce, 55
– systémová volání, 139
zapiš do souboru, 147, 193
záporná čísla – čísla se znaménkem, 44
zásobník, 65
zavedení systému na PC, 119
zdroje informací, 181
zjisti vybraný adresář (GETCWD), 155, 199
zjištění cifry v čísle v desítkové soustavě, 99
změň
– polohu v souboru (SEEK), 151
– velikost bloku paměti, 157
změna polohy v souboru, 151, 196
zruš adresář (MKDIR, RMDIR), 154, 199



Učíme se programovat v jazyce Assembler pro PC

Kniha *Učíme se programovat v jazyce Assembler pro PC* je srozumitelnou učebnicí a praktickou příručkou programování v jazyce Assembler. Svým zpracováním je určena zejména studentům středních odborných a vysokých škol, kteří potřebují zvládnout programování v tomto nižším počítačovém jazyce. Podrobností informací i množstvím příkladů z praxe však uspokojí i činné programátory a zájemce o programování unikátních aplikací, jimž dovolí psát prostřednictvím Assembleru malé, rychlé a efektivní programy využívající přímo hardware počítače a obcházející operační systém.

Krok po kroku projdete jednotlivými tématy a postupně pochopíte, jak Assembler souvisí s fyzickou architekturou počítače, jak pracuje s registry, jakým způsobem se v Assembleru implementují základní programové konstrukce, jak se vytvořený program překládá a realizuje nebo jak program v Assembleru komunikuje s operačním resp. souborovým systémem.

Osvojíte si instrukční výbavu procesoru dnešního PC a od řešení elementárních problémů nezávislých na platformě postupně dopějete až k užitečným podprogramům napsaným specificky pro operační systémy DOS, Windows nebo Linux a k propojování programů v Assembleru s aplikacemi ve vyšších programovacích jazycích. Díky přehledně zpracovaným kapitolám s popisem a příklady instrukcí pak kniha poslouží jako pohotová příručka na dosah ruky při řešení programátorských úkolů a problémů.

Hlavními body obsahu knihy jsou:

- Architektura procesoru, funkce operačního systému, strojový kód a jazyk symbolických adres
- Výpočty a převody v číselných soustavách
- Základní i pokročilé instrukce procesoru
- Základní programové konstrukce
- Hotové rutiny pro typické operace
- Práce s volně šiřitelným překladačem Netwide Assembler (NASM)
- Praktické programování v DOSu, Windows a Linuxu či Unixu
- Integrace Assembleru do aplikací ve vyšších programovacích jazycích (C a Pascal)

Popisovaná instrukční sada se vztahuje k mikroprocesorům řady x86, s níž jsou kompatibilní současné procesory v počítačích PC na našem trhu.

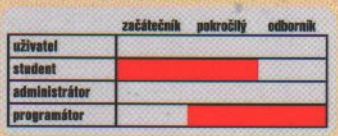
Na přiloženém CD najdete překladač NASM pro DOS, Windows i Linux, příklady a řadu informačních materiálů k programování v Assembleru včetně výukového videa o programování v Assembleru pod Linuxem.

O autorovi:



Rudolf Marek (*1981) je agilním tvůrcem a propagátorem na poli systémového programování a free-softwaru. Výrazně přispěl např. do projektu Asmutils, je spoluautorem nejmenšího WWW serveru na světě (514 bajtů), vytvořil programový modul do linuxového přehrávače MPlayer. Pro klub Sillicon Hill spolupracuje na organizaci mezinárodních konferencí OpenWeekend, na nichž i příležitostně přednáší. V době vydání knihy je studentem 5. ročníku Fakulty elektrotechnické ČVUT v Praze a správcem počítačové sítě na VŠ koleji Strahov.

Zařazení publikace:



Computer Press, a.s.
nám. 28. dubna 48
635 00 Brno

Objednávejte na:
www.knihy.cpress.cz
distribuce@cpress.cz

Bezplatná telefonní linka:
800 555 513

ISBN 80-7226-843-0
PRODEJNÍ KÓD: K0831

Doporučená cena
197 Kč
297 Sk
CD v ceně

9 788072 268436