# Predmet Asembler – organizácia

1. Motivácia, základné pojmy, prerekvizity, rodina mikroprocesorov Intel, Intel 8080.
2. Architektúra procesora Pentium, organizácia prístupu k pamäti, jazyk rodiny procesorov x86.
3. Tvorba programu v asembleri. Podprogramy a zásobník.
4. Adresovanie. Celočíselná aritmetika.
5. Skoky, iterácie. Implementácia riadiacich štruktúr jazykov vyššej úrovne.
6. Logické a bitové operácie. Polia, reťazce.
7. Prepojenie asembleru s jazykmi vyššej úrovne.
8. Prerušenia, služby operačného systému.
9. FPU a aritmetika v pohyblivej radovej čiarke.

+ prípadný pohľad na aktuálny vývoj v oblasti (podľa časových možností)

## Organizácia cvičení

* + podklady dostupné v rámci LMS Moodle

## Bodové hodnotenie

Zadanie č.1 – max. 15b (i80, 5.týždeň)

Zadanie č.2 – max. 25b (i86, 8.týždeň)

Zápočet spolu – max. 40b

Skúška – max. 60b

Spolu – max. 100b

Termíny orientačné, možnosť priebežného upresnenia. Poznámky k štúdiu (prednášky, cvičenia, samoštúdium). Literatúra.

# Motivácia pre štúdium v oblasti strojovo-orientovaných jazykov

* + cieľ predmetu – priblížiť programovanie počítača na nižšej úrovni (ako pri použití vyšších (HL) jazykov)
  + programátor – následne často produktívnejší aj pri práci vo vyšších jazykoch

Súčasná situácia

* + nie je bežné vytvárať rozsiahle programy (pre PC) v samotnom asembleri, ten je možné využiť pri tvorbe kritických častí
  + bežnejšie je využitie v prípade jednočipových systémov, vnorených systémov (zdroje, výkon)
  + stále sa však nájde dostatok dôvodov pre učenie sa a používanie asembleru [5]

Prečo asembler?

* + edukačné dôvody – cenné poznatky o činnosti počítačov, prekladačov, predikcia efektívnosti konštrukcií vyšších jazykov
  + ladenie programov, dekompilácia
  + tvorba prekladačov HL jazykov (napr. C), ladiacich nástrojov a pod.
  + ovládače zariadení, vývoj systémového softvéru, prístup k hardvéru počítača
  + efektívnosť (časová, priestorová) – program v asembleri často menší a/alebo rýchlejší ako kód generovaný kompilátorom
  + inštrukcie nedostupné v rámci vyššieho jazyka

Prečo nie asembler?

* + rýchlejší vývoj – výsledný kód zvyčajne kratší, prehľadnejší s využitím jazyka vyššej úrovne
  + spoľahlivosť a bezpečnosť
  + jednoduchšia údržba – oprava chýb, tvorba nových verzií
  + prenositeľnosť – kód ľahšie prenositeľný na iné platformy
  + vývoj v oblasti prekladačov v posledných rokoch, možnosti optimalizácie, často efektívny kód

Porovnanie výkonnosti HL jazyka a asembleru

* + násobenie dvoch 16-bit celých čísel, zdrojové texty procedúr dostupné v [1]
  + kompilátory prešli vývojom, výsledok môže byť aj lepší, ako kód vytvorený priemerným programátorom v asembleri

# Základné pojmy

Strojový jazyk (strojový kód, machine language)

* + každý procesor (typ) – vlastný strojový jazyk
  + inštrukcie tohto jazyka – čísla (bajty v pamäti)
  + každá inštrukcia – reprezentovaná jedinečným číselným kódom (operačný kód)
  + dĺžka inštrukcií – môže byť premenlivá, začína operačným kódom, nasledujú prípadné argumenty
  + programovanie značne komplikované

Asembler (jazyk, assembly language)

* + program uložený ako text
  + priama korešpondencia medzi inštrukciami asembleru a strojového jazyka

*Príklad*: add eax,ebx vs. 03 C3 (add – mnemonika inštrukcie sčítania) Asembler (program)

* + preklad z jazyka Asembler do strojového jazyka
  + podobne kompilátor realizuje preklad z HL jazyka (asembler jednoduchší ako kompilátor HL jazyka)
  + asembler pre každý konkrétny typ CPU (vlastný jazyk), ťažšie prenositeľný
  + niektoré známe asemblery: TASM, MASM, NASM

# Prerekvizity, poznatky nadobudnuté v rámci doterajšieho štúdia na FEI TU

* + v rámci predmetu Asembler predpokladáme zvládnutie vybraných okruhov z predmetov
    - Architektúry počítačových systémov (APS)
    - Princípy počítačového inžinierstva (PPI)
  + odporúčame – zopakovať vybrané poznatky získané v rámci predmetov APS a PPI

### Číselné sústavy, číselné kódy, aritmetika

* + pozičné číselné sústavy, prevody medzi sústavami

|  |  |  |  |
| --- | --- | --- | --- |
| (MSB) 32-bit data (LSB) | | | |
| 11110100 | 10011000 | 10110111 | 00001111 |
|  | |  | |
| Address | Content | Address | Content |
| ... | ... | ... | ... |
| 103 | 11110100 | 103 | 00001111 |
| 102 | 10011000 | 102 | 10110111 |
| 101 | 10110111 | 101 | 10011000 |
| 100 | 00001111 | 100 | 11110100 |
| ... | ... | ... | … |
| (a) Little-endian ordering | | (b) Big-endian ordering | |

* + priamy, inverzný, doplnkový kód
  + platné pravidlá, rozmer a povaha výsledku operácií
  + aritmetika, ukladanie viac-bajtových údajov (malý/veľký endian)

Ukladanie viac-bajtových údajov (byte-ordering scheme [1])

* + malý endian (little endian)
    - najnižší bajt (LSB) uložený najskôr (Intel Pentium)
  + veľký endian (big endian)
    - najvyšší bajt (MSB) uložený najskôr (MIPS, Power PC)
  + problém – prenos údajov medzi rôznymi systémami
  + uloženie 4B celočíselnej hodnoty

### Organizácia počítačov

* + základné stavebné prvky počítačov
  + hlavné časti počítačového systému a princípy ich činnosti
    - CPU
    - pamäť
    - vstup a výstup

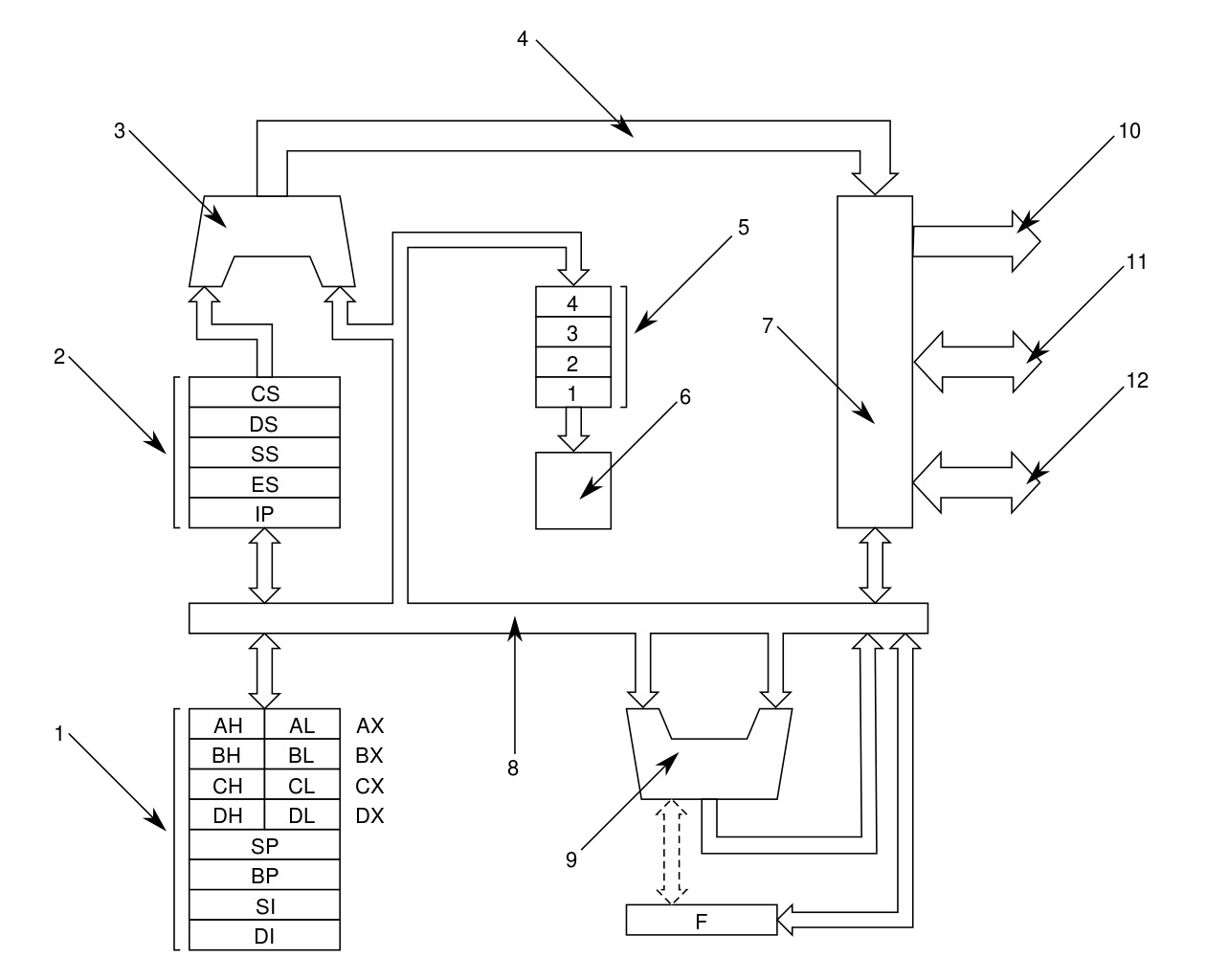
# Mikroprocesory

* + procesor ako celok integrovaný do jediného integrovaného obvodu
  + prvé dostupné mikroprocesory: [Intel 4004](https://www.intel.com/content/www/us/en/history/museum-story-of-intel-4004.html), Texas Instruments TMS 1000 (1971)
  + významné 8-bitové mikroprocesory
    - [Intel 8008](https://en.wikipedia.org/wiki/Intel_8008) (1972, prvý 8-bitový mikroprocesor), 8080 (1974, [MITS Altair 8800](https://en.wikipedia.org/wiki/Altair_8800), [PMD 85](https://pmd85.borik.net/wiki/PMD_85))
    - [Zilog Z80](http://www.z80.info/) (1976, SEGA, Nintendo, [ZX Spectrum](https://en.wikipedia.org/wiki/ZX_Spectrum))
    - Motorola 6800 (1974, MITS Altair 680, Tektronix 4051)
    - MOS Technology [6502](http://www.6502.org/)/[6510](https://en.wikipedia.org/wiki/MOS_Technology_6510) (1975, [Atari](https://en.wikipedia.org/wiki/Atari_8-bit_family) 400/800, [Apple II](https://en.wikipedia.org/wiki/Apple_II), [Commodore](http://lowendmac.com/2015/a-history-of-commodores-8-bit-computers/), Nintendo NES)
  + významné 16-bitové mikroprocesory
    - Intel 8086 (1978), 80286 (1982, IBM PC)
    - [Motorola 68000](https://en.wikipedia.org/wiki/Motorola_68000) (interne 32-bit, 1979, [Amiga](https://en.wikipedia.org/wiki/Amiga), [Atari ST](https://en.wikipedia.org/wiki/Atari_ST), Apple)
    - Zilog Z8000 (1979, Olivetti M20 - M60)

### Rodina mikroprocesorov Intel

*vybrané mikroprocesory 70-tych rokov*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Processor | Clock Speed(s) | Intro Date(s) | Mfg. Process/ Transistors | Transistors | Addressable Memory | Cache | Bus Speed | Typical Use |
| 4004 | 108 KHz | Nov-71 | 10-micron | 2,300 | 640 Bytes | None | 108 KHz | Busicom calculator, arithmetic manipulation |
| 8008 | 200 KHz | Apr-72 | 10-micron | 3,500 | 16 KB | None | 200 KHz | Dumb terminals, general calculators, bottling machines, data/character manipulation |
| 8080 | 2 MHz | Apr-74 | 6-micron | 6,000 | 64 KB | None | 2 MHz | Traffic light controller, Altair computer (first PC) |
| 8086 | 10 MHz  8 MHz  4.77 MHz | Jun-78 | 3-micron | 29,000 | 1 MB | None | 10 MHz  8 MHz  4.77 MHz | Portable computing |
| 8088 | 8 MHz  4.77 MHz | Jun-79 | 3-micron | 29,000 | 64 kB ([?](https://en.wikipedia.org/wiki/Intel_8088)) | None | 8 MHz  4.77 MHz | Desktops (standard CPU for all IBM PCs and PC clones at the time) |

* + i8086 – významný zástupca rodiny z konca 70-tych rokov
    - zavedenie segmentácie pamäti
    - významne obohatený súbor inštrukcií
    - významná zmena architektúry (x86) [6], [7]:
      * 1 – GPR, 2 – SR a IP, 3 – adresová sčítačka,
      * 4 – int. adresná zbernica, 5 – front inštrukcií,
      * 6 – riadiaca jednotka, 7 – rozhranie zbernice,
      * 8 – int. údajová zbernica, 9 – ALU
      * 10/11/12 – ext. adresná/údajová/riadiaca zbernica

*vybrané mikroprocesory 80-tych rokov* (Wikimedia, Harkonnen2)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Processor | Clock Speed(s) | Intro Date(s) | Mfg. Process/ Transistors | Transistors | Addressable Memory | Cache | Bus Speed | Typical Use |
| 80286 | 12 MHz  10 MHz  6 MHz | Feb-82 | 1.5-micron | 134,000 | 16 MB | None | 12 MHz  10 MHz  6 MHz | Desktops (standard CPU for all IBM PCs clones at the time) |
| Intel386™ DX Processor | 33 MHz  25 MHz  20 MHz  16 MHz | Oct-85 | 1.5 micron 1-micron | 275,000 | 4 GB | None | 33 MHz  25 MHz  20 MHz  16 MHz | Desktops |
| Intel486™ DX Processor | 50 MHz  33 MHz  25 MHz | Apr-89 | 1-micron 0.8-micron | 1.2 million | 4 GB | 8 kB | 50 MHz  33 MHz  25 MHz | Desktops and servers |

* + i80286 prináša – ochrana pamäti (16-bit protected mode), kompatibilita s 8086 (real mode)
  + i80386 prináša (významný pokrok) – 32 bitov (registre, zbernice)
    - ochrana pamäti (32-bit protected mode, segmenty môžu mať veľkosť až 4GB – flat memory model)
    - stránkovanie pamäti (virtuálna pamäť), kompatibilita (real mode)

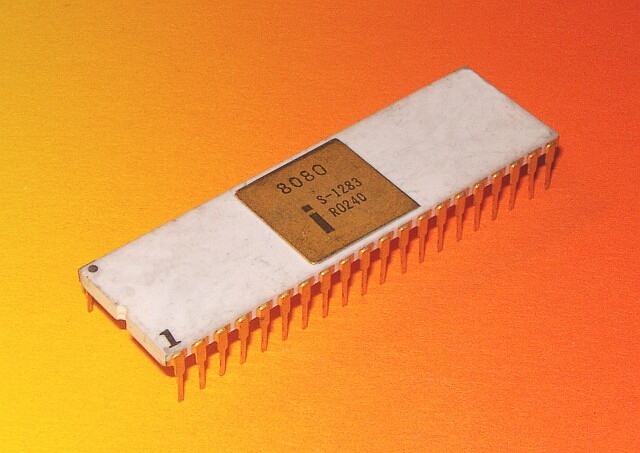
*vybrané mikroprocesory 90-tych rokov*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Processor | Clock Speed(s) | Intro Date(s) | Mfg. Process/ Transistors | Transistors | Addressable Memory | Cache | Bus Speed | Typical Use |
| Intel386™ SL Processor | 25 MHz  20 MHz | Oct-90 | 1-micron | 855,000 | 4 GB | None | 25 MHz  20 MHz | First CPU designed specifically for portables |
| Intel486™ SX Processor | 33 MHz  25 MHz  20 MHz  16 MHz | Sept-91 | 1 micron 0.8-micron | 1.2 million  900,000 | 4 GB | 8 kB | 33 MHz  25 MHz  20 MHz  16 MHz | Low-cost, entry-level desktops |
| Intel486™ SL Processor | 33 MHz  25 MHz  20 MHz | Nov-92 | 0.8-micron | 1.4 million | 4 MB | 8 kB | 33 MHz  25 MHz  20 MHz | First CPU specifically designed for Notebook PCs |
| Intel® Pentium® Processor | 66 MHz  60 MHz | Mar-93 | 0.8 micron | 3.1 million | 4 GB | 8 kB L1  Cache | 66 MHz  60 MHz | Desktops |
| Intel® Pentium® Processor with MMX™ Technology | 233 - 166  MHz | Oct-96 | 0.35 micron | 4.5 million | 4 GB | 16 kB L1  Cache | 66 MHz | High performance desktops and servers |
| Intel® Pentium® II Processor | 300 - 233  MHz | May-97 | 0.35 micron | 7.5 million | 64 GB | 512 kB  L2 Cache | 66 MHz | High-end business desktops, workstations and servers |
| Intel® Pentium® III Processor | 600 - 450  MHz | Feb-99 | 0.25 micron | 9.5 million | 4 GB | 512 kB  L2 Cache | 133  100 MHz | Business, consumer PCs;1- and 2-way servers and workstations |

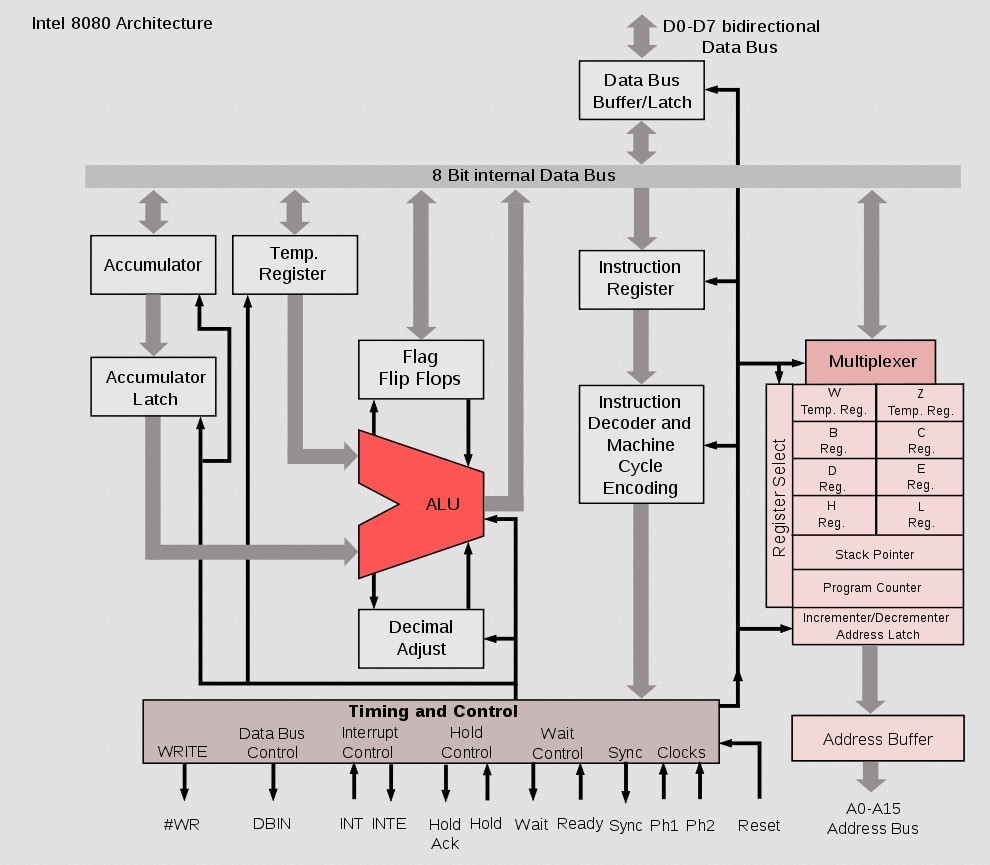
* + ďalší vývoj zameraný hlavne na zvýšenie výkonu; zmeny v rámci architektúry ([x86-64](https://en.wikipedia.org/wiki/X86-64))
  + rozšírenie jazyka o nové inštrukcie (napr. [SIMD](https://en.wikipedia.org/wiki/SIMD) rozšírenia – [MMX](https://en.wikipedia.org/wiki/MMX_(instruction_set)), [SSE](https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions), [AVX](https://en.wikipedia.org/wiki/Advanced_Vector_Extensions))
  + spracované podľa [Microprocessor Quick Reference Guide](http://www.intel.com/pressroom/kits/quickreffam.htm)

### Mikroprocesor Intel 8080

* + významný predstaviteľ 8-bitových CPU
  + detailnejší pohľad
  + prvé osobné počítače (napr. MITS [Altair 8800](https://en.wikipedia.org/wiki/Altair_8800), 1975)



(Wikimedia, Christian Bassow)



(Wikipedia, Appaloosa)

#### Štruktúra i8080

* + registre (8-bit): B, C, D, E, H, L, pomocné W, Z (nedostupné programovo), A, F (ALU)
  + registre (16-bit)
    - SP (adresa poslednej vloženej položky)
    - PC (adresa nasledujúcej inštrukcie)
    - registrové páry (BC, DE, HL, PSW [[3](https://altairclone.com/downloads/manuals/8080%20Programmers%20Manual.pdf#page%3D7)])
  + ALU – realizácia operácií aritmetických, logických a rotácií
    - A, F, pomocné registre
  + riadiace obvody, register a dekóder inštrukcií – riadenie činnosti procesora
    - výber 1.bajtu inštrukcie (register inštrukcií – RI)
    - dekódovanie inštrukcie dekóderom inštrukcií (DI)
    - signály z dekódera spolu s časovacími signálmi – riadia činnosť ostatných blokov CPU
  + zbernica (údajová – 8 bit., adresná – 16 bit.)

#### Príznaky (register F)

* + 5 príznakov (S, Z, Ac, P, Cy), programovo dostupné 4 z nich
  + možnosť zmeny vykonávania programu podľa výsledku predošlej operácie
    - podmienené skoky, volania podprogramov, návraty
  + potrebná informácia o vplyve realizácie jednotlivých inštrukcií na príznaky
  + Sign (S) – najvyšší bit výsledku, znamienko (1 – záporný výsledok)
  + Zero (Z) – nulový výsledok operácie (1 – nula)
  + Auxiliary Carry (Ac) – prenos z 3. bitu [[3](https://altairclone.com/downloads/manuals/8080%20Programmers%20Manual.pdf#page%3D12)] akumulátora, desiatková korekcia (inštrukcia DAA [[3](https://altairclone.com/downloads/manuals/8080%20Programmers%20Manual.pdf#page%3D21)])
  + Parity (P) – počet jednotiek vo výsledku (1 – párny)
  + Carry (Cy) – prenos z najvyššieho bitu výsledku operácie (1 – prenos nastal)

#### Kódovanie registrov a registrových párov i8080

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Register (DDD/SSS) | B | C | D | E | H | L | M | A |
| Kód | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |

M – pamäť adresovaná RP HL

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Registrový pár (RP) | B | D | H | SP, PSW |
| Kód | 00 | 01 | 10 | 11 |

#### Formát údajov a inštrukcií [4]

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 8-bit binary | | | | | | | |  |
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | DATA WORD |

* + údaje – 8-bitové binárne čísla
  + inštrukcie – dĺžka inštrukcie 1, 2 alebo 3 bajty, podľa typu

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1-byte instructions | | | | | | | |  |
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | OP CODE |
| 2-byte instructions | | | | | | | |  |
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | OP CODE |
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | OPERAND |
| 3-byte instructions | | | | | | | |  |
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | OP CODE |
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | Low address or OPERAND1 |
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | High address or OPERAND2 |

## Jazyk procesora Intel 8080

Adresovanie operandov (metóda sprístupnenia operandov)

* + *implicitné* (implied) – dané funkciou inštrukcie (STC, DAA)
  + *registrové* (register) – operand v registri, druhý implicitný operand – akumulátor (CMP, ADD), RP ako operand (INX, DCX)
  + *bezprostredné* (immediate) – údaj časťou inštrukcie (za operačným kódom, ADI, CPI, MVI, LXI)
  + *priame* (direct) – priama adresa (16-bitov) súčasťou inštrukcie (JMP, LDA, STA)
  + *nepriame* (register indirect) – pamäťová referencia v RP (LDAX, STAX)

Kódovanie a činnosť (vybraných) inštrukcií

* + D8 – 8-bitový údaj, D16 – 16-bitový údaj (DL/DH), P8 – 8-bitové číslo portu
  + ovplyvnené príznaky – S Z Ac P Cy („–“ – bez zmeny)
  + registrové páry (RP – B, D, H, SP, PSW)

#### Inštrukcie presunov

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Inštrukcia | 1.B (Op.kód) | 2.B | 3.B | Operácia | Príznaky | RP | RTL |
| MOV R1,R2 | 01DDDSSS | - | - | move | - - - - - |  | R1←R2 |
| MVI R,D8 | 00DDD110 | D8 | - | move immediate | - - - - - |  | R←D8 |
| LXI RP,D16 | 00RP0001 | DL | DH | load immediate RP | - - - - - | B,D,H | RP←D16 |
| LDA D16 | 00111010 | DL | DH | load A direct | - - - - - |  | A←[D16] |
| STA D16 | 00110010 | DL | DH | store A direct | - - - - - |  | [D16]←A |
| LDAX RP | 00RP1010 | - | - | load A indirect | - - - - - | B,D | A←[RP] |
| STAX RP | 00RP0010 | - | - | store A indirect | - - - - - | B,D | [RP]←A |
| LHLD D16 | 00101010 | DL | DH | load HL direct | - - - - - |  | HL←[D16] |
| SHLD D16 | 00100010 | DL | DH | store HL direct | - - - - - |  | [D16]←HL |
| XCHG | 11101011 | - | - | exchange HL,DE | - - - - - |  | HL↔DE |

**Aritmetické operácie**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Inštrukcia | 1.B (Op.kód) | 2.B | 3.B | Operácia | Príznaky | RP | RTL |
| ADD R | 10000SSS | - | - | add register to A | S Z AcP Cy |  | A←A+R |
| ADC R | 10001SSS | - | - | add reg. w.carry | S Z AcP Cy |  | A←A+R+Cy |
| ADI D8 | 11000110 | D8 | - | add imm. to A | S Z AcP Cy |  | A←A+D8 |
| ACI D8 | 11001110 | D8 | - | add imm. to A w.carry | S Z AcP Cy |  | A←A+D8+Cy |
| DAD RP | 00RP1001 | - | - | add RP to HL | - - - - Cy | B,D,H,SP | HL←HL+RP |
| SUB R | 10010SSS | - | - | subtract reg. from A | S Z AcP Cy |  | A←A-R |
| SBB R | 10011SSS | - | - | subtract reg. w bor. | S Z AcP Cy |  | A←A-R-Cy |
| SUI D8 | 11010110 | D8 | - | subtract immed. | S Z AcP Cy |  | A←A-D8 |
| SBI D8 | 11011110 | D8 | - | sub.imm.w.borrow | S Z AcP Cy |  | A←A-D8-Cy |
| INR R | 00DDD100 | - | - | increment reg. | S Z AcP - |  | R←R+1 |
| DCR R | 00DDD101 | - | - | decrement reg. | S Z AcP - |  | R←R-1 |
| INX RP | 00RP0011 | - | - | increment r.pair | - - - - - | B,D,H,SP | RP←RP+1 |
| DCX RP | 00RP1011 | - | - | decrement r.pair | - - - - - | B,D,H,SP | RP←RP-1 |
| DAA | 00100111 | - | - | decimal adjust accum. | S Z AcP Cy |  |  |

Poznámky k realizácii aritmetických operácií:

* + aritmetické operácie i8080 predpokladajú doplnkovú reprezentáciu (pri tvorbe doplnku pretečenie ignorované)
  + realizácia odčítania s využitím operácie sčítania – zjednodušenie obvodového riešenia
  + odčítanie vs. pripočítanie čísla s opačným znamienkom
    - rovnaký výsledok, ale Cy-bit odlišný
    - pri operácii odčítania (SUB, SBB, CMP, …) Cy-bit je invertovaný (signalizácia potreby pôžičky z vyššieho radu (borrow) pri odčítaní viacbajtových hodnôt)
  + DAA – úprava 8-bitového hexadecimálneho čísla v registri A na dve 4-bitové BCD číslice (detaily v napr. [[3](https://altairclone.com/downloads/manuals/8080%20Programmers%20Manual.pdf#page%3D21)])

*Príklad*:

1. 35-12

0010 0011

-12 = (1000 1100) = (1111 0011)i = (1111 0100)c

1. 12-35

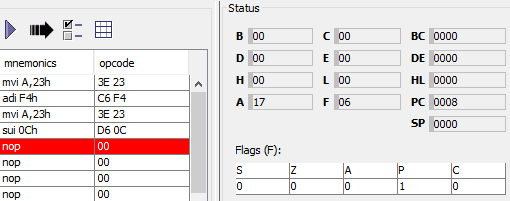
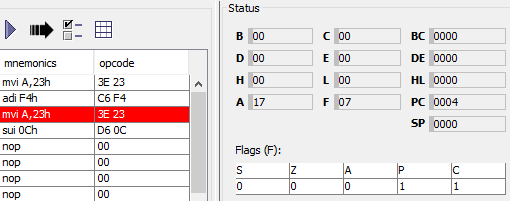
1 0001 0111 = (23)10

0000 1100

-35 = (1010 0011) = (1101 1100)i = (1101 1101)c

(1110 1001)c = (1001 0111) = (-23)10

|  |  |  |  |
| --- | --- | --- | --- |
| MVI A,35 ADI -12 | MVI A,35 SUI 12 | MVI A,12 ADI -35 | MVI A,12 SUI 35 |
| A = 23 | A = 23 | A = -23 | A = -23 |
| Cy = 1 | Cy = 0 | Cy = 0 | Cy = 1 |



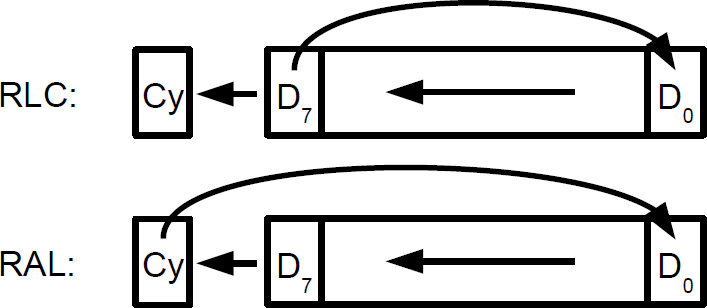
#### Logické operácie a rotácie

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Inštrukcia | 1.B (Op.kód) | 2.B | 3.B | Operácia | Príznaky | RP | RTL |
| ANA R | 10100SSS | - | - | AND reg. with A | S Z 0 P 0 |  | A←A and R |
| XRA R | 10101SSS | - | - | XOR with A | S Z 0 P 0 |  | A←A xor R |
| ORA R | 10110SSS | - | - | OR reg. with A | S Z 0 P 0 |  | A←A or R |
| CMP R | 10111SSS | - | - | compare with A | S Z AcP Cy |  | F←A ? R |
| ANI D8 | 11100110 | D8 | - | AND immediate w.A | S Z 0 P 0 |  | A←A and D8 |
| XRI D8 | 11101110 | D8 | - | XOR immediate w.A | S Z 0 P 0 |  | A←A xor D8 |
| ORI D8 | 11110110 | D8 | - | OR immediate w.A | S Z 0 P 0 |  | A←A or D8 |
| CPI D8 | 11111110 | D8 | - | cmp.immediate w.A | S Z AcP Cy |  | F←A ? D8 |
| RLC | 00000111 | - | - | rotate A left | - - - - Cy |  |  |
| RRC | 00001111 | - | - | rotate A right | - - - - Cy |  |  |
| RAL | 00010111 | - | - | r.A left t.carry | - - - - Cy |  |  |
| RAR | 00011111 | - | - | r.A right t.carry | - - - - Cy |  |  |
| CMA | 00101111 | - | - | complement A | - - - - - |  | A←A |
| STC | 00110111 | - | - | set carry | - - - - 1 |  | Cy←1 |
| CMC | 00111111 | - | - | complement carry | - - - - x |  | Cy←Cy |

Príznaky (Z, Cy) ovplyvnené inštrukciami CMP, CPI: A = X: Z = 1 A < X: Cy = 1

A ≠ X: Z = 0 A ≥ X: Cy = 0

|  |  |  |  |
| --- | --- | --- | --- |
| MVI A,35  CPI -12 (F4h) | MVI A,35 CPI 12 | MVI A,12  CPI -35 (DDh) | MVI A,12 CPI 35 |
| Cy = 1 | Cy = 0 | Cy = 1 | Cy = 1 |

Rotácie akumulátora:

RLC Di → Di+1, D7 → D0, D7 → Cy RRC Di → Di-1, D0 → D7, D0 → Cy

RAL Di → Di+1, D7 → Cy, Cy → D0 RAR Di → Di-1, D0 → Cy, Cy → D7

#### Skoky

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Inštrukcia | 1.B (Op.kód) | 2.B | 3.B | Operácia | Príznaky | RP | RTL |
| JMP D16 | 11000011 | DL | DH | jump unconditional | - - - - - |  | PC←D16 |
| JC D16 | 11011010 | DL | DH | jump on carry | - - - - - |  | IF Cy=1 PC←D16 |
| JNC D16 | 11010010 | DL | DH | jump on no carry | - - - - - |  | IF Cy=0 PC←D16 |
| JZ D16 | 11001010 | DL | DH | jump on zero | - - - - - |  | IF Z=1 PC←D16 |
| JNZ D16 | 11000010 | DL | DH | jump on no zero | - - - - - |  | IF Z=0 PC←D16 |
| JP D16 | 11110010 | DL | DH | jump on positive | - - - - - |  | IF S=0 PC←D16 |
| JM D16 | 11111010 | DL | DH | jump on minus | - - - - - |  | IF S=1 PC←D16 |
| JPE D16 | 11101010 | DL | DH | jump on par.even | - - - - - |  | IF P=1 PC←D16 |
| JPO D16 | 11100010 | DL | DH | jump on par.odd | - - - - - |  | IF P=0 PC←D16 |
| PCHL | 11101001 | - | - | HL to PC | - - - - - |  | PC←HL |

JMP D16 PC ← D16

JC D16 IF Cy = 1 THEN PC ← D16

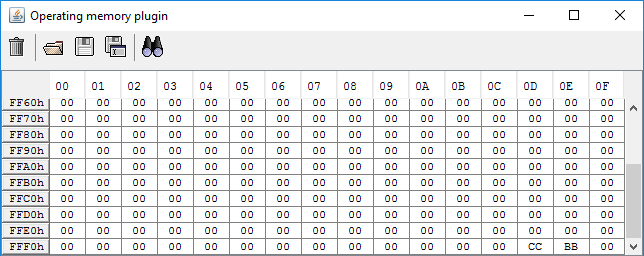
JNC D16 IF Cy = 0 THEN PC ← D16

#### Zásobník a podprogramy

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Inštrukcia | 1.B (Op.kód) | 2.B | 3.B | Operácia | Príznaky | RP | RTL |
| PUSH RP | 11RP0101 | - | - | push RP on stack | - - - - - | B,D,H,PSW |  |
| POP RP | 11RP0001 | - | - | pop RP off stack | - - - - - | B,D,H,PSW |  |
| XTHL | 11100011 | - | - | exch.top.st.w.HL | - - - - - |  | HL↔[SP] |
| SPHL | 11111001 | - | - | HL to SP | - - - - - |  | SP←HL |
| CALL D16 | 11001101 | DL | DH | call uncondition. | - - - - - |  |  |
| Cxx D16 | 11yyy100 | DL | DH | call on ... | - - - - - |  |  |
| RET | 11001001 | - | - | return | - - - - - |  |  |
| Rxx | 11yyy000 | - | - | return on ... | - - - - - |  |  |
| RST N | 11nnn111 | - | - | restart | - - - - - |  |  |

PUSH RP [SP-1] ← RP1, [SP-2] ← RP2, SP ← SP-2 POP RP RP1 ← [SP+1], RP2 ← [SP], SP ← SP+2

CALL D16 [SP-1] ← PCH, [SP-2] ← PCL, SP ← SP-2, PC ← D16 RST [SP-1] ← PCH, [SP-2] ← PCL, SP ← SP-2, PC ← 8\*nnn RET PCL ← [SP], PCH ← [SP+1], SP ← SP+2

*Príklad*:

|  |  |  |
| --- | --- | --- |
|  | | pamäť |
| DCX SP | ; SP = FFFF | FFFF 00 |
| LXI B,0BBCCh | ; B = BB, C = CC | FFFE BB |
| PUSH B | ; SP = FFFD | FFFD CC |
| POP D | ; SP = FFFF | FFFC 00 |
| HLT |  |  |

kódovanie podmienok (skokov, volaní a návratu)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| yyy | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| Jxx | JNZ | JZ | JNC | JC | JPO | JPE | JP | JM |
| Cxx | CNZ | CZ | CNC | CC | CPO | CPE | CP | CM |
| Rxx | RNZ | RZ | RNC | RC | RPO | RPE | RP | RM |

#### Ostatné inštrukcie (I/O, riadiace)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Inštrukcia | 1.B (Op.kód) | 2.B | 3.B | Operácia | Príznaky | RP | RTL |
| IN P8 | 11011011 | P8 | - | input | - - - - - |  | A←[P8] |
| OUT P8 | 11010011 | P8 | - | output | - - - - - |  | [P8]←A |
| EI | 11111011 | - | - | enable interrupt | - - - - - |  |  |
| DI | 11110011 | - | - | disable interrupt | - - - - - |  |  |
| NOP | 00000000 | - | - | no-operation | - - - - - |  |  |
| HLT | 01110110 | - | - | halt | - - - - - |  |  |

Vstup a výstup:

* komunikácia s perifériami zabezpečená cez 256 I/O portov
* IN port – do akumulátora prenesený bajt zo špecifikovaného portu
* OUT port – obsah akumulátora prenesený na daný port
* IN, OUT – iba iniciujú prenos (zariadenie je povinné detegovať, že je adresované)

Prerušenia, zastavenie a obnovenie činnosti procesora:

* HLT – zastavenie činnosti procesora (z tohto stavu možno činnosť procesora obnoviť len externou udalosťou, napr. prerušenie)
* periférne zariadenie môže iniciovať prerušenie pomocou signálu INT (spracované, ak prerušenia sú povolené)
* pre obsluhu prerušení bežne využívaná inštrukcia RST [[3](https://altairclone.com/downloads/manuals/8080%20Programmers%20Manual.pdf#page%3D43)]
* príslušnú inštrukciu poskytne zariadenie, ktoré iniciuje prerušenie (umiestnením na údajovú zbernicu) [4]

*Príklad*: násobenie 2 bezznamienkových 8-bitových čísel

* opakované sčítanie (HL ← C \* D)

MVI B,0 LXI H,0 MOV A,D CPI 0

JZ done next: DAD B

DCR D

JNZ next done: HLT

* násobenie s využitím posunov (rýchlejšie, posun vľavo/vpravo) [[3](https://altairclone.com/downloads/manuals/8080%20Programmers%20Manual.pdf#page%3D58)]
  + D – násobenec, C – násobiteľ (na začiatku), rozmer výsledku: 2*n* radov (16 bitov, BC)
  + algoritmus:
    1. test najnižšieho bitu násobiteľa: 0 – pokračuj krokom 2, 1 – pripočítaj násobenec k vyššiemu bajtu výsledku (VBV)
    2. posuň celý (2B) výsledok o 1 bit vpravo
    3. opakuj kroky 1 a 2, pokiaľ neboli otestované všetky bity násobiteľa

MULT: MVI B,0

MVI E,9

;inicializacia VBV

;pocitadlo

MULT0: MOV A,C

RAR

MOV C,A DCR E JZ DONE MOV A,B

JNC MULT1 ADD D

MULT1: RAR

MOV B,A JMP MULT0

DONE: HLT

;rotacia nizsieho bajtu vysledku (NBV)

;a posun LSb nasobitela do Cy

;koniec

;nasobenec k VBV ak Cy bolo 1

;posun VBV, v Cy hodnota pre posun do NBV

MULTIPLICAND 00001010

.

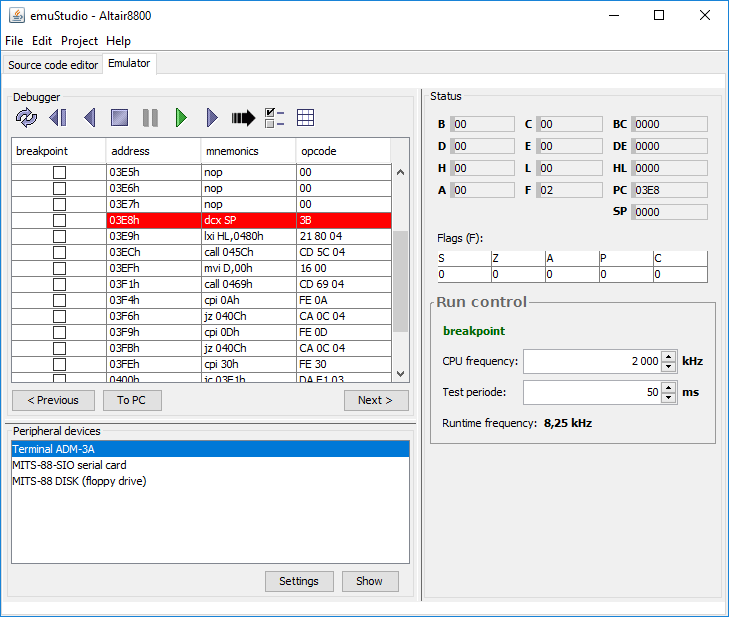
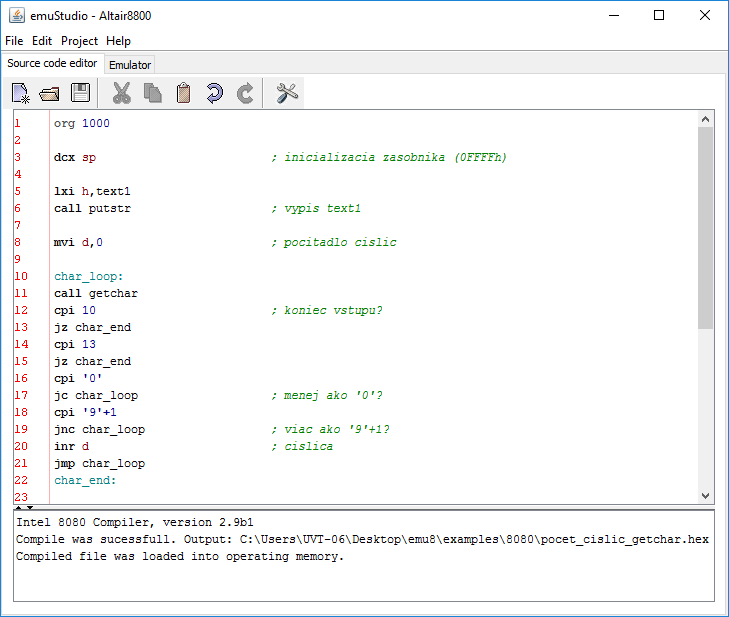
MULTIPLIER 00000101

0.0AH.27 + 0.0AH.26 + 0.0AH.25 + 0.0AH.24 +

0.0AH.23 + 1.0AH.22 + 0.0AH.21 + 1.0AH.20 =

00101000 + 00001010 = 00110010 = 5010

*Platforma emuStudio – emulácia počítača s procesorom Intel 8080*



#### Študijná literatúra:

1. Dandamudi, S.P.: Introduction to Assembly Language Programming, Springer Science+Business Media, Inc., 2005.
2. Carter, A.P.: [PC Assembly Language](https://pacman128.github.io/static/pcasm-book.pdf), 2019.
3. [Intel 8080 Assembly Language Programming Manual](https://altairclone.com/downloads/manuals/8080%20Programmers%20Manual.pdf), Intel Corporation, 1975.
4. [Intel 8080 Microcomputer Systems User's Manual](http://www.nj7p.info/Manuals/PDFs/Intel/9800153B.pdf), Intel Corporation, 1975.
5. Fog, A.: [Optimizing subroutines in assembly language](http://www.agner.org/optimize/optimizing_assembly.pdf), Technical University of Denmark, 1996 – 2021.
6. Hudák, Š: Strojovo-orientované jazyky, FEI TU v Košiciach, 2003.
7. Wikipedia, [Intel 8086](https://en.wikipedia.org/wiki/Intel_8086), 2023.

# Architektúra procesora Pentium

* + architektúra rodiny procesorov x86 (32-bit)
  + snaha o zachovanie spätnej kompatibility novších procesorov
  + organizácia pamäti, segmentácia

#### Registre

1. pre všeobecné použitie [1]

|  |  |  |  |
| --- | --- | --- | --- |
| EAX (32-bit) | | | |
|  | | AX (16-bit) | |
|  |  | AH (8-bit) | AL (8-bit) |
| 31 | 16 | 15 8 | 7 0 |

* + *hlavné údajové registre* (EAX, EBX, ECX, EDX)
  + možnosť použitia ako 32/16/8-bitových registrov
  + väčšina operácií presunov, aritmetických a logických operácií
  + špeciálne určenie pri vykonávaní špecifických operácií (násobenie, iterácie)
  + *ukazovatele* (pointer registers) – ESP, EBP (práca so zásobníkom)
  + *indexové* (index registers) – ESI, EDI (aj ich 16-bitové subregistre – SI, DI; reťazcové operácie)

1. riadiace [1]
   * EIP (instruction pointer, IP v 16-bit. režime) – adresa nasledujúcej inštrukcie, automatická aktualizácia
   * EFLAGS (FLAGS v 16-bit. režime) – stavové (6), riadiace (1) a systémové (10) príznaky
     + stavovými, riadiacim a vybranými systémovými príznakmi sa budeme zaoberať detailnejšie
     + DF (reťazce, 0 – autoincrement), TF (single step, ladenie), IF (povolenie obsluhy prerušení), VM (emulácia 8086)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| EFLAGS | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | | | | FLAGS | | | | | | | | | | | | | | | |
| 3  1 | 3  0 | 2  9 | 2  8 | 2  7 | 2  6 | 2  5 | 2  4 | 2  3 | 2  2 | 2  1 | 2  0 | 1  9 | 1  8 | 1  7 | 1  6 | 1  5 | 1  4 | 1  3 | 1  2 | 1  1 | 1  0 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  | I D | V I P | V I F | A C | V M | R F | 0 | N T | IO PL | | O F | D F | I F | T F | S F | Z F | 0 | A F | 0 | P F | 1 | C F |

|  |  |  |
| --- | --- | --- |
| **Status flags** | **Control flags** | **System flags** |
| CF = Carry flag | DF = Direction flag | TF = Trap flag |
| PF = Parity flag |  | IF = Interrupt flag |
| AF = Auxiliary carry flag |  | IOPL = I/O privilege level |
| ZF = Zero flag |  | NT = Nested task |
| SF = Sign flag |  | RF = Resume flag |
| OF = Overflow flag |  | VM = Virtual 8086 mode |
|  |  | AC, VIF, VIP, ID |

1. segmentové
   * šesť 16-bitových registrov, podpora segmentácie pamäti (umiestnenie segmentov)
   * CS (code), DS (data), SS (stack), ES, FS, GS (pomocné)

### Organizácia prístupu k pamäti v systémoch s procesorom Pentium

* chránený režim (protected mode) – natívny režim, podpora segmentácie i stránkovania (virtuálna pamäť, transparentná pre aplikácie)
* reálny režim (real mode) – spätná kompatibilita s programami pre i8086, segmentácia

## Organizácia pamäti v chránenom režime

* segmentácia – prevod logickej adresy na 32-bit. lineárnu adresu (odlišná od reálneho (16-bit.) režimu)
* stránkovanie [8] – prevod lineárnej adresy na fyzickú (ak sa stránkovanie nevyužíva, lineárna adresa = fyzická) [1]

Page translation

Segment translation

Logical address

32-bit linear address

32-bit physical address

|  |  |  |
| --- | --- | --- |
| 15 3 | 2 | 1 0 |
| INDEX | TI | RPL |

#### Segmentové registre (SR)

DESCRIPTOR TABLE

Segment descriptor

ADDER

32-bit base

LINEAR ADDRESS

0

31

OFFSET

0

31

SR obsahuje segmentový selektor (segment selector) [1]

* *index* (13-bit) do tabuľky deskriptorov (DT)
  + deskriptor segmentu (8-bajtov)

|  |
| --- |
|  |
|  |
| ACCESS RIGHTS |
| LIMIT |
| BASE ADDRESS |
|  |
|  |

* + - báza segmentu (32-bit), veľkosť, prístupové práva
    - adresa deskriptora: index\*8 + bázová adresa DT
  + výpočet lineárnej adresy
    - bázová adresa segmentu + posun (offset, 16/32-bit)
* *TI* (table indicator, 1-bit)
  + indikuje použitie lokálnej(1)/globálnej(0) DT
* *RPL* (requester privilege level, 2-bit)
  + úroveň oprávnenia prístupu k údajom
  + nižšia hodnota, vyššie oprávnenie (0 – jadro OS)

#### Segmentový deskriptor [1]

* poskytuje atribúty segmentu:
* bázová adresa (32-bit) – začiatok segmentu vo fyzickom adresnom priestore
* veľkosť segmentu (20-bit), dve interpretácie
  + 1B – 1MB (220B), krok 1B (G = 0)
  + 4KB – 4GB, krok 4KB (G = 1)
* stavové a riadiace informácie
  + G (granularity) – interpretácia veľkosti segmentu (jednotka 1B/4KB)
  + D/B – dvojaká interpretácia, podľa typu segmentu
    - kódový (D) – (default operand size) predpokladaná veľkosť operandov a posunutia (offset): 0 – 16-bit, 1– 32-bit.
    - dátový (B) – veľkosť zásobníka: B = 0 – použije sa SP register a max. veľkosť je FFFFH, ak B = 1 – ESP a FFFFFFFFH
  + AVL (available) – pre softvérové použitie, nevyužívané hardvérom
  + P (present) – (1) prítomnosť segmentu v pamäti, (0) – generovanie výnimky (prípadné zavedenie do pamäti) pri prístupe k seg.
  + DPL (descriptor privilege level) – úroveň oprávnenia potrebná pre prístup k segmentu, riadenie prístupu k segmentu
    - DPL ≥ RPL potrebné pre povolenie prístupu
    - (v skutočnosti sa overuje aj DPL ≥ CPL, kde CPL (current privilege level) je získaný z reg. CS, viac informácií napr. v [5])
  + S – systémový segment (0) využívaný pre špeciálne účely, segment (kódový alebo dátový) aplikácie (1)
  + TYPE – typ segmentu (kódový/dátový, čítanie/zápis, ...)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3  1 |  |  |  |  |  |  | 2  4 | 2  3 | 2  2 | 2  1 | 2  0 | 1  9 |  |  | 1  6 | 1  5 | 1  4 | 1  3 | 1  2 | 1  1 |  |  | 8 | 7 |  |  |  |  |  |  | 0 |  |
| BASE 31:24 | | | | | | | | G | D  / B | 0 | A V L | LIMIT 19:16 | | | | P | DPL | | S | TYPE | | | | BASE 23:16 | | | | | | | | +4 |
| BASE ADDRESS 15:00 | | | | | | | | | | | | | | | | SEGMENT LIMIT 15:00 | | | | | | | | | | | | | | | | +0 |
| 3  1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 1  6 | 1  5 |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 0 |  |

#### Tabuľky deskriptorov

* pole segmentových deskriptorov, 3 typy tabuliek:
  + GDT (global descriptor table) – deskriptory dostupné všetkým úlohám, jediná v systéme, OS
  + LDT (local descriptor table) – deskriptory jednotlivých úloh (kód, dáta, zásobník)
  + IDT (interrupt descriptor table) – spracovanie prerušení
* tabuľky LDT a GDT (8B – 64KB, max. 8192 (213) 8B deskriptorov)
* s tabuľkou asociovaný register (32-bit bázová adresa, 16-bit veľkosť tabuľky)
  + LDTR (inštrukcie LLDT/SLDT používané OS), GDTR (inštrukcie LGDT/SGDT používané OS)

#### Modely segmentácie

* jednotný (flat model) – ukrytie mechanizmu segmentácie (bázy segmentov = 0, veľkosť 4GB)
* viac-segmentový (multisegment model) – súčasne aktívnych max. 6 segmentov (ich selektory v segmentových registroch)
  + generovanie výnimky (general protection) pri pokuse o prístup k pamäti nad rámec veľkosti segmentu (limit)



ADDER

|  |  |
| --- | --- |
| 19 4 3 0 | |
| Segment register | 0 0 0 0 |

## Organizácia pamäti v reálnom režime

|  |  |
| --- | --- |
| 19 16 15 0 | |
| 0 0 0 0 | Offset value |

* Pentium ako rýchly 8086
  + 1MB pamäti, registre veľkosti 16-bit v 8086 (dopad na adresovanie)
  + veľkosť segmentu – 64KB
  + logická adresa (v programe) – 2 časti: segment:offset (SEG:OFF)
  + SEG – báza segmentu v pamäti
  + OFF – relatívna adresa v rámci segmentu
* výpočet fyzickej adresy z logickej [1]
  + predpokladajú sa 0000 v najnižších 4-bitoch bázy segmentu
  + umiestnenie začiatku segmentu v pamäti (napr. 00010H, FFFE0H, …)

20-bit physical memory address

0

19

*Príklad*: SEG = 1100H, OFF = 450H

11000 (segment \* 24)

450 (offset)

11450 (fyzická adresa)

* logická vs. fyzická adresa
  + pre každú logickú adresu existuje jedinečná fyzická
  + opačne to neplatí – viac ako jedna logická adresa môže referovať na rovnakú fyzickú adresu

*Príklad*: SEG1 = 1000, OFF1 = 20A9, SEG2 = 1200, OFF2 = A9.

10000 12000

20A9 00A9

120A9 120A9 (fyzická adresa)

* možno použiť 6 segmentových registrov (CS, DS, SS, ES, FS, GS), aj keď i8086 mal len prvé 4 [1]
* segmentové registre sú nezávislé – segmenty môžu byť oddelené, môžu sa čiastočne/úplne prekrývať

#### Činnosť procesora v zmiešanom režime (mixed-mode operation)

* operandy a adresy 16/32-bit
* bit D/B deskriptora určuje predpokladanú veľkosť
* prefixy určenia veľkosti 2 typov, umožnenie programovania v zmiešanom režime (signalizovaná iná ako predpokladaná veľkosť):
  + operandu
  + adresy

#### Formát inštrukcie

* detailný opis jednotlivých častí napr v. [6], [7]
* voliteľné prefixy i ďalšie časti inštrukcie

|  |  |  |  |
| --- | --- | --- | --- |
| Instruction prefix | Address-size prefix | Operand-size prefix | Segment override |
| 0 or 1B | 0/1B | 0/1B | 0/1B |

* ModR/M – špecifikácia typu adresovania
  + niektoré hodnoty indikujú ďalší bajt (SIB) nasledujúci za ModR/M bajtom
* SIB (Scale Index Base)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Opcode | ModR/M | SIB | Displacement | Immediate |
| 1/2B | 0/1B | 0/1B | 0,1,2 or 4B | 0,1,2 or 4B |

* max. dĺžka inštrukcie – 15B [[7](http://www.c-jump.com/CIS77/CPU/x86/lecture.html#X77_0020_encoding_overview)], [9]

#### Použitie segmentových registrov

v závislosti od typu pamäťovej referencie:

* výber inštrukcie – CS (báza segmentu), IP/EIP (offset)
* zásobníkové operácie – SS (báza), SP/ESP (offset) – push a pop, pre ďalšie operácie často BP/EBP
* prístup k údajom – obyčajne DS (báza), posun v závislosti od režimu adresovania

#### Inicializácia

* BIST (built-in self-test), výsledok testu v EAX (0 = OK), EDX – identifikácia procesora
* štartovacia adresa: 0FFFFFFF0H (16B od konca 32-bit adresného priestoru)
* ROM pamäť (BIOS) mapovaná v tejto oblasti – inicializácia systému
* Pentium štartuje v reálnom režime, prepnutie do chráneného režimu (tabuľky GDT, IDT, register CR0)

# Jazyk rodiny procesorov x86

Asembler – príkazy:

* *inštrukcie* (procesor, inštrukcie strojového jazyka)
* *direktívy* (prekladač – riadenie prekladu, nie generovanie strojových inštrukcií pre procesor)
* *makrá* (skupina príkazov, nahradenie počas prekladu – expanzia) Formát príkazov:

[návestie] mnemonika [operandy] [;komentár]

* návestie (oddelené „:“ od mnemoniky inštrukcie, nie však v prípade direktív)

*Príklad*:

opakuj: inc pocet ;zvýš 'pocet' o 1 CR EQU 0DH ;carriage return

### Alokácia údajových objektov

* HL jazyky – nepriamo, špecifikáciou typu premennej (int suma)
* okrem alokácie miesta, daný aj spôsob interpretácie (unsigned s1 vs. int s2; 4B obidve premenné)
* asembler – alokácia, príp. aj inicializácia (nie interpretácia), formát:

[premenná] direktíva poč.hodnota [, poč.hodnota]

* direktívy alokácie (inicializácia):
  + DB (define byte, 1B), DW (define word, 2B), DD (define doubleword, 4B), DQ (define quadword, 8B), DT (def. ten bytes, 10B)

*Príklad*:

sorted DB 'y' (rovnako: sorted DB 79H) value DW 25159 (6247H, Little endian: 47, 62) expr1 DW 7\*25 (inicializácia vo forme výrazu)

* rozsah hodnôt (znamienkové a bezznamiekové čísla)
* direktívy alokácie (bez inicializácie) – RESB (reserve byte, 1B), RESW (2B), RESD (4B), RESQ (8B), REST (10B)

*Príklad*:

buffer resw 100 (pole 100 slov)

Viacnásobné definície

* alokácia súvislého bloku pamäti pre postupnosť definícií
* možnosť definície skrátiť

*Príklad*:

message DB 'BYE', 0DH, 0AH

* direktíva TIMES – viacnásobná inicializácia na rovnakú hodnotu (polia)

*Príklad*:

marks TIMES 8 DW 0

### Operandy

* väčšina inštrukcií vyžaduje operandy
* *adresovanie* – spôsob špecifikácie umiestnenia operandu
* základné typy operandov (detaily neskôr):
  + *registrový* (register CPU obsahuje údaj, rýchlosť)

mov EAX,EBX

* + *bezprostredný* (údaj v rámci inštrukcie – kód. segment)

mov AL,65 (zdroj bezprostredný, cieľ - register)

* + *pamäťový* (efektívna adresa (offset): súčasťou inštrukcie – *priama*, v registri – *nepriama*)
    - adresa vs. hodnota (NASM: mov EBX,table, resp. mov EBX,[table])

table TIMES 20 DD 0

...

mov [table],'A' ; direct memory addressing mov EBX,table ; indirect memory addressing

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| mov | [EBX],100 | ; | table[0] = 100 |  |  |  | (rozmer | údaja?) |
| add | EBX,4 | ; | položka tabuľky | má | 4B | (DD) |  |  |
| mov | [EBX],99 | ; | table[1] = 99 |  |  |  |  |  |

* + *implicitný* operand

stc, clc, pusha, popa, ...

### Základné inštrukcie jazyka procesora Pentium

* referenčná príručka (napr. v [2], [3], [4])

#### Inštrukcie presunov

* inštrukcia MOV (kopírovanie údajov)
  + syntax, pravidlá použitia (src – bez zmeny, rovnaký rozmer operandov)

mov dst,src (dst ← src)

* + kombinácie operandov (R – register, M – memory, I – immediate)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| mov | R,R |  |  | mov | R,M |  | |
| mov | R,I | (mov | EAX,3) | mov | M,R |  |  |
| mov | M,I |  |  | mov | M,M | – neprípustná | kombinácia! |

* + (explicitná) špecifikácia rozmeru údajov (BYTE, WORD, DWORD, QWORD, TBYTE)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| mov | [EBX],100 | ; nie je jasný | rozmer údaja | pre | zápis |
| mov | BYTE [EBX],100 | ; zapíše sa 1B |  |  |  |
| mov | WORD [EBX],100 | ; zapíšu sa 2B |  |  |  |

* inštrukcia XCHG (exchange)
  + výmena operandov, syntax

xchg dst,src (dst ↔ src)

* + pravidlá (obidva operandy v pamäti – neprípustné)
  + výmena bez pomocného registra (triedenie, malý/veľký endian)

mov ECX,EAX vs. xchg EAX,EDX mov EAX,EDX

mov EDX,ECX

* inštrukcia XLAT (translate)
  + prevod znakov, syntax

xlat (AL ← [EBX + AL])

* + použitie
    - vstup: EBX – zač. adresa tabuľky, AL – index
    - výstup: AL (bajt z výslednej adresy)

#### Aritmetické operácie

* inštrukcie INC a DEC
  + zvýšenie/zníženie hodnoty operandu (8, 16, 32-bit; register/pamäť) o 1, syntax

*Príklad*:

inc dst (inc EBX)

dec dst (dec DL)

count DW 0

value DB 25

.code

inc [count] ; OK (typ 'count' známy) mov EBX,count

inc [EBX] ; nejednoznačnosť

inc WORD[EBX] ; OK

* inštrukcia ADD
  + sčítanie 8, 16, 32-bit operandov, syntax

add dst,src (dst ← dst + src)

* + pravidlá (podobne ako MOV)
  + INC vs. ADD (preferované INC – menej pamäti) inc EAX vs. add EAX,1
* inštrukcia SUB
  + odčítanie 8, 16, 32-bit operandov, syntax

sub dst,src (dst ← dst - src)

* inštrukcia CMP
  + podobne ako SUB, ale výsledok sa neukladá ('dst' bez zmeny, nastavenie príznakov)
  + typické použitie – podmienené skoky

#### Skoky a iterácie

mov inc\_again:

inc jmp mov

. . .

EAX,1

EAX

inc\_again EBX,EAX

* nepodmienený skok – JMP
  + odovzdanie riadenia na návestie label

jmp label (EIP ← label)

* podmienené skoky – vykonávanie programu sa prenesie na návestie (pri splnení podmienky)

*Príklad*:

j<podm> label

cmp AL,0DH

je CD\_received ; jump on equal

* + register FLAGS (ZF = 1 pri rovnosti operandov CMP)
  + nie všetky inštrukcie modifikujú príznaky (napr. MOV)
  + podmienky
    - jednoduchý test hodnoty príznaku (jz, jnz, jc, jnc, …)
    - komplexnejšie podmienky (jg, jl, jge, jle, … )
* iterácie

mov repeat:

CL,50

<loop body> dec CL

jnz repeat

. . .

* + možno realizovať aj pomocou skokov
  + priama podpora iterácií – loop
    - syntax

loop label

mov repeat:

ECX,50

<loop body> loop repeat

. . .

* + - činnosť (ECX – počítadlo):

ECX = ECX – 1

if ECX  0 then EIP ← label

#### Logické operácie

* inštrukcie AND, OR, XOR, NOT
  + syntax

and dst,src (or, xor)

not dst (unárny operátor)

* + tabuľky hodnôt

|  |  |  |
| --- | --- | --- |
| and | 0 | 1 |
| 0 | 0 | 0 |
| 1 | 0 | 1 |

|  |  |  |
| --- | --- | --- |
| or | 0 | 1 |
| 0 | 0 | 1 |
| 1 | 1 | 1 |

|  |  |  |
| --- | --- | --- |
| xor | 0 | 1 |
| 0 | 0 | 1 |
| 1 | 1 | 0 |

*Príklad*: operácie AND, OR, XOR, NOT [1]

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| AL | | BL | | and AL,BL | | or AL,BL | | xor AL,BL | | not | AL |
| AL | | AL | | AL | | AL | |
| 1010 | 1110 | 1111 | 0000 | 1010 | 0000 | 1111 | 1110 | 0101 | 1110 | 0101 | 0001 |
| 0110 | 0011 | 1001 | 1100 | 0000 | 0000 | 1111 | 1111 | 1111 | 1111 | 1001 | 1100 |
| 1100 | 0110 | 0000 | 0011 | 0000 | 0010 | 1100 | 0111 | 1100 | 0101 | 0011 | 1001 |
| 1111 | 0000 | 0000 | 1111 | 0000 | 0000 | 1111 | 1111 | 1111 | 1111 | 0000 | 1111 |

*Príklad*: použitie logických operácií

and je

AL,01H

bit\_is\_zero

<code to be executed when the bit is 1> jmp skip1

bit\_is\_zero:

<code to be executed when the bit is 0>

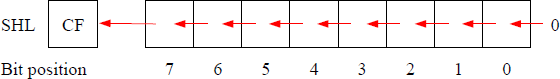
skip1:

<rest of the code>

* inštrukcia TEST
  + ak je problémom modifikácia operandu
  + vykonáva bitovú operáciu AND (ako inštrukcia and), nastaví príznaky rovnako, bez modifikácie operandu

test AL,01H ;možno použiť v predošlom príklade namiesto: and AL,01H

* posuny
  + SHL (shift left) – posun o jeden bit vľavo (CF  MSb, LSb   



* + SHR (shift right) – posun o jeden bit vpravo (CF  LSb, MSb  
  + syntax (SHL, SHR), dst (8, 16, 32-bit M/R)

|  |  |  |  |
| --- | --- | --- | --- |
| shl | dst,count | ; špecifikácia posunu priamo | (0-31) |
| shl | dst,CL | ; nepriamo, v registri CL |  |

*Príklad*: využitie operácií posunov (LSb, CF)

shr jnc

AL,1

bit\_is\_zero

<code to be executed when the bit is 1> jmp skip1

bit\_is\_zero:

<code to be executed when the bit is 0>

skip1:

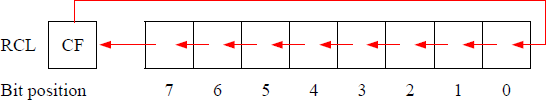
<rest of the code>

* rotácie
  + strata bitov pri posunoch – niekedy nežiadúca
  + rotácie bez CF – inštrukcie ROL, ROR
    - posledný vystupujúci bit zachytený v CF
    - syntax (ROL, ROR)

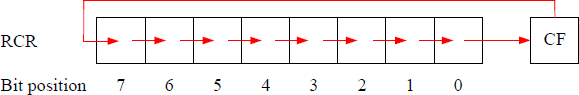
rol dst,count

rol dst,CL

* + - ROL – rotácia vľavo; bity vystupujúce vľavo vstupujú sprava (Di+1 Di, LSb  MSb, CF  MSb)
    - ROR – rotácia vpravo; bity vystupujúce vpravo vstupujú zľava (Di  Di+1, MSb  LSb, CF  LSb)
  + rotácie cez CF – inštrukcie RCL, RCR
    - CF zahrnutý v procese rotácie, syntax (ako u ROL, ROR)
    - RCL – rotácia vľavo; vystupujúci bit vstupuje do CF, CF vstupuje sprava (Di+1 Di, CF  MSb, LSb  CF) [1]



* + - RCR – rotácia vpravo; vystupujúci bit vstupuje do CF, CF vstupuje zľava (Di  Di+1, CF  LSb, MSb  CF) [1]



*Príklad*: využitie pri viacslovných posunoch (64-bit, EDX:EAX o 1 bit vpravo)

shr EDX,1

rcr EAX,1

### Konštanty a Makrá

NASM poskytuje viacero možností, priblížime 3 direktívy:

* direktíva EQU
  + syntax

name EQU expr

* + použitie – priradenie výsledku výrazu 'expr' menu 'name' (expr – možno vyhodnotiť v čase prekladu)
  + výhody: čitateľnosť programu, viacnásobný výskyt konštanty – zmena na jednom mieste
  + priradené hodnoty nemožno meniť v danom zdrojovom module

|  |  |  |
| --- | --- | --- |
| *Príklad*: |  | |
| ROWS | EQU | 40 |
| COLS | EQU | 20 |
| ARRAY\_SIZE | EQU | ROWS \* COLS |

* direktíva %assign – podobne ako EQU (definícia numerických konštánt), ale opätovná definícia možná

*Príklad*:

%assign i j+1

<fragment kódu>

%assign i j+2

* direktíva %define – numerické i reťazcové konštanty, opätovná definícia možná

*Príklad*:

%define X1 [EBP+4]

<fragment kódu>

%define X1 [EBP+20]

## Makrá

* reprezentácia bloku textu menom; pri výskyte mena makra v programe – nahradenie príslušným blokom (macro expansion)

%macro macro\_name para\_count

<macro body>

%endmacro

* volanie makra – použije sa meno makra s príslušnými parametrami
* makrá bez parametrov

|  |  |  |
| --- | --- | --- |
| definícia: | volanie makra: | po rozvinutí makra, asembler prekladá: |
| %macro multEAX\_by\_16 | ... | ... |
| sal EAX,4 | mov EAX,27 | mov EAX,27 |
| %endmacro | multEAX\_by\_16 | sal EAX,4 |
|  | ... | ... |

* makrá s parametrami – generalizácia makra; v tele makra – prístup k parametrom podľa poradia (%1)

|  |  |  |
| --- | --- | --- |
| definícia: | volanie makra: | po rozvinutí makra: |
| %macro mult\_by\_16 1 | ... | ... |
| sal %1,4 | mult\_by\_16 DL | sal DL,4 |

%endmacro ... ...

#### Študijná literatúra:

1. Dandamudi, S.P.: Introduction to Assembly Language Programming, Springer Science+Business Media, Inc., 2005.
2. [NASM](https://www.nasm.us/) – The Netwide Assembler, The NASM Development Team, 1996-2015.
3. Carter, A.P.: [PC Assembly Language](https://pacman128.github.io/static/pcasm-book.pdf), 2019.
4. Cloutier, F.: [x86 and amd64 instruction reference](https://www.felixcloutier.com/x86/), 2022.
5. Stack Overflow, [Difference between DPL and RPL in x86](https://stackoverflow.com/questions/36617718/difference-between-dpl-and-rpl-in-x86), 2014.
6. Intel 80386 Reference Programmer's Manual, [Instruction Format](https://pdos.csail.mit.edu/6.828/2018/readings/i386/s17_02.htm).
7. Kholodov, I.: Bristol Community College, CIS-77 Introduction to Computer Systems, [Encoding Real x86 Instructions](http://www.c-jump.com/CIS77/CPU/x86/lecture.html).
8. Intel 80386 Reference Programmer's Manual, [Page Translation](https://pdos.csail.mit.edu/6.828/2018/readings/i386/s05_02.htm).
9. [Intel® 64 and IA-32 Architectures Software Developer’s Manual](https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf), Intel Corporation, 2022.

# Tvorba programu v asembleri

* + preklad (assembling) – preklad programu v asembleri do strojového kódu
  + spájanie (linking) – spojenie strojového kódu a údajov v objektových súboroch, prípadne knižníc → vykonateľný súbor
  + ladenie (debugging)

#### Vstup a výstup

* + neexistencia štandardného mechanizmu pri programovaní v asembleri (systémovo závislé)
  + riešenia
    - priamy prístup k hardvéru
    - služby operačného systému
    - HL jazyky – štandardné knižnice
  + v tomto kurze použité riešenie [2]
    - podprogramy pre zjednodušenie základných I/O operácií (využívajú knižnicu jazyka C)
      * print\_int (celočíselná hodnota v EAX)
      * print\_char (ASCII v AL)
      * print\_string (adresa v EAX, nulový terminátor)
      * print\_nl (nový riadok)
      * read\_int (uloží hodnotu do EAX)
      * read\_char (ASCII do EAX)
    - zachovávajú hodnoty registrov (read\* – modifikujú EAX)
    - použitie
      * %include “asm\_io.inc”
      * volanie jednotlivých operácií (call)

#### Ladenie programu

* + zvolený prístup použitý v [2]
  + zobrazenie stavu počítača (bez jeho modifikácie)
  + makrá definované v asm\_io.inc
  + operácie:
    - dump\_regs (HEX výpis obsahu registrov, FLAGS, argument – číslo výpisu)
    - dump\_mem (HEX/ASCII výpis obsahu pamäti, arg1 – číslo výpisu, arg2 – adresa, arg3 – počet 16B blokov)
    - dump\_stack (výpis obsahu zásobníka, arg1 – číslo výpisu, arg2 – # dvoj-slov pod EBP, arg3 – # dvoj-slov nad EBP)
    - dump\_math (hodnoty v registroch FPU, argument – číslo výpisu)

#### Využitie jazyka C pri tvorbe programov v asembleri

* + programy (pre PC) v súčasnosti zriedka vytvárané v samotnom asembleri (HLL – jednoduchosť, prenositeľnosť, ...)
  + častejšie – kombinácia s HL jazykom (napr. C) – kritické časti v asembleri (rýchlosť, prístup k hardvéru, špeciálne inštrukcie ...)
  + štart (asm) programu z C ovládača (C-driver), motivácia:
    - prepnutie do chráneného režimu (inicializácia SR, tabuľky v pamäti)
    - dostupnosť štandardnej knižnice jazyka C (I/O, …)

#### Vzorový program

* + ovládač v jazyku C (driver.c) [2]

int main()

{

int ret\_status; ret\_status = asm\_main(); return ret\_status;

}

* + samotný program v jazyku asembler
    - segment .data (inicializované údaje, reťazce ukončené 0)
    - segment .bss (neinicializované údaje, ak sú potrebné)
    - segment .text (kód programu)
      * \_asm\_main (C-volacia konvencia; všetky symboly C (funkcie, globálne premenné) – prefix '\_' (Win))
      * global \_asm\_main (vytvorenie globálneho návestia, viditeľné aj v iných moduloch)
  + preklad (MinGW 32-bit)
    - asembler nasm -f win32 -d COFF\_TYPE asm\_io.asm (asm\_io.obj)
    - asembler nasm -f win32 prvy.asm (prvy.obj)
    - driver.c gcc -c driver.c (driver.obj)
    - spájanie gcc -o prvy driver.obj prvy.obj asm\_io.obj (gcc volá linker s vhodnými parametrami, prvy.exe)
  + preklad (MinGW 64-bit, „multilib“ verzia)
    - asembler – bez zmeny
    - spájanie gcc -o prvy -m32 prvy.obj driver.c asm\_io.obj (prvy.exe)
  + ďalšie výstupné formáty NASM napr. v [3]

*Príklad*: prevod a výpis hodnoty zadaného znaku v HEX formáte (adaptované podľa [1]).

|  |  |  |  |
| --- | --- | --- | --- |
| ; Objective: HEX hodnota zadaneho znaku.  ; Input: Znak z klavesnice.  ; Output: Vytlaci ASCII kod znaku v HEX. |  |  |  |
| %include "asm\_io.inc" segment .data  char\_prompt db "Zadaj znak: ",0 out\_msg1 db "ASCII kod znaku '",0 out\_msg2 db "' v HEX je ",0 |  |  |  |
| segment .text  global \_asm\_main  \_asm\_main:  enter 0,0 pusha  mov EAX,char\_prompt call print\_string call read\_char  mov EBX,EAX  call print\_nl  mov EAX,out\_msg1 call print\_string mov EAX,EBX  call print\_char  mov EAX,out\_msg2 call print\_string mov EAX,EBX  mov AH,AL  shr AL,4  mov ECX,2 | print\_digit:  cmp jg add jmp  A\_to\_F:  add  skip:  call mov and loop  popa mov leave ret | AL,9  A\_to\_F AL,'0'  skip AL,'A'-10  print\_char AL,AH AL,0FH  print\_digit EAX,0 | ; prevod na A – F  ; prevod na 0 – 9 |

# Zásobník, podprogramy

* + procedúry a modulárne programovanie

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| TOS →  ESP (256)  SS → |  | TOS →  ESP (254)  SS → |  | TOS →  ESP (250)  SS → |  |
| ?? | 21 | 21 |
| ?? | AB | AB |
| ?? | ?? | 7F |
| ?? | ?? | BD |
| ?? | ?? | 32 |
| ?? | ?? | 9A |
| ⁝ | ⁝ | ⁝ |
| ?? | ?? | ?? |
| ?? | ?? | ?? |
|  |  |  |
| (a) Empty stack (256B) | | (b) After pushing 21ABH | | (c) After pushing 7FBD329AH | |

* + úloha zásobníka pri používaní procedúr
  + implementácia zásobníka (Pentium) [1]

#### Zásobník

* + LIFO údajová štruktúra, operácie PUSH a POP
  + priamo prístupný len prvok na vrchole zásobníka (TOS)
  + poradie vkladaných/vyberaných prvkov

#### Implementácia zásobníka (Pentium)

* + oblasť pamäti rezervovaná v zásobníkovom segmente
  + vrchol zásobníka (TOS) daný SS:ESP
    - SS – začiatok segmentu zásobníka
    - ESP – relatívna adresa (offset)
  + možno vložiť slovo (16-bit), dvoj-slovo (32-bit)
  + zásobník narastá smerom k nižším adresám
  + TOS – posledná vložená položka (nižší byte)

*Príklad*: zásobník o veľkosti 256B

1. prázdny – pri pokuse o výber – chyba (stack underflow)
2. vložené slovo (2B) – zníženie ESP, potom uloženie (little endian)
3. vložené dvoj-slovo (4B)
   * plný zásobník (ESP = 0), pokus o uloženie – chyba (stack overflow)
   * výber zo zásobníka
     + zvýšenie hodnoty ESP (pamäť bez zmeny)
     + uvoľnené lokality použiteľné pre uloženie nových hodnôt

#### Operácie so zásobníkom

* + inštrukcie push a pop
    - uloženie slova/dvoj-slova na zásobník
    - syntax

push src (src – 16/32-bit GPR, SR, pamäť, konštanta) pop dst (dst - 16/32-bit GPR, SR, pamäť)

* + - sémantika

push src16 ESP ← ESP – 2, [SS:ESP] ← src16 push src32 ESP ← ESP – 4, [SS:ESP] ← src32 pop dst16 dst16 ← [SS:ESP], ESP ← ESP+2 pop dst32 dst32 ← [SS:ESP], ESP ← ESP+4

* + inštrukcie pushf a popf
    - uloženie a obnovenie registra príznakov (FLAGS)
    - syntax

pushfd (32-bit EFLAGS) pushfw (16-bit FLAGS)

popfd popfw

* + inštrukcie pusha a popa
    - uloženie GPR registrov (8)
    - syntax

pushad (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI)

popad (obnoví, okrem ESP)

pushaw (AX, CX, DX, BX, SP, BP, SI, DI)

popaw (okrem SP)

#### Typické využitie zásobníka

* + dočasné ukladanie údajov
    - limitovaný počet GPR registrov
    - uloženie a obnova obsahu registrov

*Príklad*: výmena obsahu dvoch 32-bit premenných (value1, value2)

1. push EAX ; uloženie registrov b) push value1 push EBX push value2

|  |  |  |  |
| --- | --- | --- | --- |
| mov | EAX,value1 | pop | value1 |
| mov | EBX,value2 | pop | value2 |
| mov | value1,EBX |  |  |
| mov | value2,EAX |  |  |
| pop | EBX |  |  |
| pop | EAX ; obnovenie obsahu |  |  |

* + riadenie vykonávania programu
    - volanie procedúr (návratová adresa)
  + odovzdávanie parametrov procedúram (parameter passing)
    - HL jazyky

### Procedúry

* + logicky samostatná časť kódu pre vykonávanie určitej úlohy
  + mechanizmy odovzdávania parametrov
    - hodnotou (call by value) – iba aktuálna hodnota parametrov (bez možnosti zmeny procedúrou)
    - referenciou (call by reference) – adresy parametrov (možnosť zmeny hodnôt parametrov)
  + HL jazyky (2 typy podprogramov)
    - funkcie (jediná návratová hodnota)
    - procedúry (žiadna/viac návratových hodnôt)

#### Procedúry v jazyku x86 (Pentium)

* + volanie procedúry (proc-name – relatívne voči inštrukcii nasledujúcej za call (offset))

call proc-name (ESP ← ESP-4, [SS:ESP] ← EIP, EIP ← EIP+rel.posun)

*Príklad* [1]:

; end of avg procedure

sum EAX,EBX

call mov

...

00000028 E8F0FFFFFF

0000002D 89D8

; end of sum procedure

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

avg:

...

EBP

push

...

0000001D 55

; end of main

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

sum:

sum EBX,EAX

main:

...

call mov

...

00000002 E816000000

00000007 89C3

machine code (hex)

offset (hex)

* + - po výbere call, EIP = 00000007H
    - uloženie EIP do zásobníka
    - prenos riadenia na začiatok procedúry 'sum'
      * pripočítanie posunu k EIP
      * 00000007H + 00000016H = 0000001DH
    - dopredné volanie – kladný posun (signed, 32-bit)
    - procedúra 'avg' – volanie späť (záporný posun)
  + návrat z procedúry
    - odovzdanie riadenia späť z volanej do volajúcej procedúry (inštrukcia nasledujúca za call)
    - návratová adresa (zásobník)

ret (EIP ← [SS:ESP], ESP ← ESP+4)

* + - parameter inštrukcie ret (počet bajtov pre uvoľnenie zo zásobníka, voliteľné)

ret n

## Odovzdávanie parametrov

* + uloženie parametrov na dohodnuté miesto (registre, zásobník)
  + volanie procedúry

#### Registrová metóda

* + GPR registre

|  |  |  |  |
| --- | --- | --- | --- |
| *Príklad*: odovzdávanie hodnotou, registrová metóda (adaptované [1]). |  | | |
| %include "asm\_io.inc" |  | call | print\_string |
| segment .data |  | call | read\_int |
| prompt\_msg1 db "Zadaj prve cislo: ",0 |  | mov | EDX,EAX |
| prompt\_msg2 db "Zadaj druhe cislo: ",0 |  | call | print\_nl |
| sum\_msg db "Suma cisel je: ",0 |  | mov | EAX,sum\_msg |
|  |  | call | print\_string |
| segment .text |  | call | sum |
| global \_asm\_main |  | call | print\_int |
| \_asm\_main: |  | call | print\_nl |
| enter 0,0 |  |  |  |
| pusha |  | popa |  |
|  |  | mov | EAX,0 |
| mov EAX,prompt\_msg1 |  | leave |  |
| call print\_string |  | ret |  |
| call read\_int | sum: |  |  |
| mov ECX,EAX |  | mov | EAX,ECX |
| call print\_nl |  | add | EAX,EDX |
| mov EAX,prompt\_msg2 |  | ret |  |

* + registrová metóda – vlastnosti
    - výhody
      * jednoduchšia pre malý počet parametrov
      * rýchla (parametre v registroch)
    - nevýhody
      * obmedzený počet GPR registrov (parametrov)
      * často potrebná dočasná úschova používaných GPR (zásobník) – strata výhody rýchlosti

#### Zásobníková metóda

* + všetky parametre v zásobníku pred volaním procedúry
  + problém – prístup k parametrom (call/EIP)

(offset)

stack

|  |  |
| --- | --- |
|  |  |
|  | number1 |
|  | number2 |
| ESP → | Return Add. |
|  |  |
|  |  |

push number1

adresy

push number2

call sum

* + riešenie – prístup s využitím EBP (obsah ESP sa mení – push/pop)
  + štandardný spôsob prístupu k parametrom

mov EBP,ESP

mov EAX,[EBP+4]

* + uschovanie obsahu EBP (možná obnova pôvodnej hodnoty pred návratom k volajúcej procedúre)

push EBP

mov EBP,ESP

* + informácia v zásobníku (parametre, návratová adresa, starý EBP, príp. lokálne premenné) – *zásobníkový rámec* (stack frame) [1]
    - zásobník po uložení EBP (a)
    - pred ukončením procedúry – obnova EBP (b)
    - po vykonaní inštrukcie ret (c)

(offset)

stack

(a)

(offset)

stack

(b)

(offset)

stack

(c)

|  |  |
| --- | --- |
|  |  |
|  | number1 |
| +8 | number2 |
| +4 | Return Add. |
| EBP,ESP → | EBP |
|  |  |

|  |  |
| --- | --- |
|  |  |
|  | number1 |
|  | number2 |
| ESP → | Return Add. |
|  |  |
|  |  |

|  |  |
| --- | --- |
|  |  |
|  | number1 |
| ESP → | number2 |
|  |  |
|  |  |
|  |  |

* + uvoľnenie parametrov zo zásobníka
    - úprava ESP *volajúcou* procedúrou (napr. jazyk C)

push number1

push number2

call sum

add ESP,4 (ak mame 2 premenne po 2B)

* + - úprava *volanou* procedúrou

ret hodnota (EIP ← SS:ESP, ESP ← ESP + 4 + hodnota)

* + uvoľnenie zásobníka – porovnanie prístupov
    - procedúry s pevným počtom parametrov (preferovaná druhá metóda)
    - procedúry s premenlivým počtom parametrov (potrebné použiť prvú)

## Uchovanie stavu procedúry

* + napr. volaná procedúra manipuluje s registrom využívaným volajúcou
  + obsahy registrov spravidla uchováva volaná procedúra
    - dôvody (zmeny volanej procedúry, viacnásobné volania – dlhší kód)
  + ktoré registre uchovať?
    - všeobecne: používané volajúcou, modifikované volanou procedúrou
    - push vs. pushad
      * rýchlosť (5x, efektívne pri úschove > 5 registrov)
      * návratová hodnota v registri (gcc – EAX), popa následne znehodnotí
      * modifikácia prístupu k parametrom (offset)

pusha

mov EBP,ESP

## Inštrukcie ENTER a LEAVE

* + enter – alokácia zásobníkového rámca pri vstupe do procedúry

enter bytes,level

* + - bytes – počet bajtov pre lokálne premenné
    - level – úroveň vnorenia procedúry (bežne 0)
  + leave – uvoľnenie zásobníkového rámca (bez parametrov)

proc-name:

enter XX,0

...

procedure body

...

leave

ret YY

|  |  |  |  |
| --- | --- | --- | --- |
| enter | XX,0 | leave |  |
| push | EBP | mov | ESP,EBP |
| mov sub | EBP,ESP ESP,XX | pop | EBP |

*Príklad*: odovzdávanie parametrov, zásobníková metóda (adaptované [1]).

(offset)

stack

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| %include "asm\_io.inc" segment .data  prompt\_msg1 db "Zadaj prve cislo: ",0 prompt\_msg2 db "Zadaj druhe cislo: ",0 sum\_msg db "Suma cisel je: ",0 |  | call mov call  call call call | print\_nl EAX,sum\_msg print\_string sum print\_int print\_nl |  |
| segment .text  global \_asm\_main  \_asm\_main:  enter 0,0 pusha |  | popa mov leave ret | EAX,0 |  |
| mov EAX,prompt\_msg1 call print\_string call read\_int  push EAX  call print\_nl  mov EAX,prompt\_msg2 call print\_string call read\_int  push EAX | sum: | enter mov  add leave ret | 0,0  EAX,[EBP+12] EAX,[EBP+8]  8 | ; save EBP  ; restore EBP  ; params |

|  |  |
| --- | --- |
|  |  |
| +12 | num1 |
| +8 | num2 |
| +4 | R.A. |
| EBP → | old EBP |
|  |  |

## Procedúry s premenlivým počtom parametrov

* + bežne prítomné napr. v jazyku C (scanf, printf)
  + volaná procedúra potrebuje informáciu o počte parametrov [1]
    - posledný parameter vložený do zásobníka – počet parametrov
  + parametre zo zásobníka odstránené volajúcim

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | N parameters  Number of parameters |
|  |  | parameter N |
|  |  | parameter N-1 |
| ⁝ |  | ⁝ |
| EBP+16 |  | parameter 2 |
| EBP+12 |  | parameter 1 |
| EBP+8 |  | N |
| EBP+4 |  | Return address |
| EBP, ESP | → | EBP |
|  |  |  |

## Lokálne premenné

* + vznikajú pri aktivácii procedúry, zanikajú pri jej terminácii (sú dynamické)
  + alokácia v dátovom segmente nežiadúca:
    - v DS alokácia statická (dostupné aj keď procedúra je neaktívna)
    - nepracuje korektne s rekurzívnymi procedúrami (volajú sami seba priamo/nepriamo)
  + preto – alokácia v zásobníku [1]

*Príklad*:

int compute(int a, int b)

{

int temp, N;

...

...

}

|  |  |  |
| --- | --- | --- |
| EBP+12 EBP+8 EBP+4 EBP EBP-4 EBP-8 |  | Parameters  Local variables  ← ESP |
| a |
| b |
| Return address |
| old EBP |
| temp |
| N |
|  |

* + prístup k obsahu zásobníkového rámca – EBP (frame pointer)
    - parametre (a, b – EBP+12, EBP+8) – poradie parametrov v zásobníku závislé od HL jazyka
    - lokálne premenné (temp, N – EBP–4, EBP–8)
  + možnosť sprehľadnenia programu:

Použitie:

%define a dword[EBP+12]

%define temp dword[EBP-4]

mov EBX,a namiesto: mov EBX,[EBP+12]

mov temp,EAX mov [EBP-4],EAX

## Viacmodulové programy

* + reálne aplikácie (často množstvo procedúr)
  + rozdelenie zdrojového textu na menšie časti (moduly)
  + výhody
    - zmeny v module, preklad (iba) konkrétneho modulu
    - jednoduchšia, bezpečnejšia modifikácia (menšie súbory)
  + oddelený preklad modulov – potrebná špecifikácia rozhrania

global error\_txt, sum, mylabel

.DATA

error\_txt db 'Out of range!',0 sum dw 0

...

.CODE

...

mylabel:

...

ret

* + - NASM – direktívy GLOBAL a EXTERN

#### Direktíva GLOBAL

* + sprístupní návestie iným modulom programu (mená procedúr, premenných, …)

global label1, label2, ...

#### Direktíva EXTERN

* + informuje asembler, že návestie nie je definované v aktuálnom module (definované v inom)

extern label1, label2, ... (label1, label2 – sprístupnené v inom module pomocou GLOBAL)

#### Študijná literatúra:

1. Dandamudi, S.P.: Introduction to Assembly Language Programming, Springer Science+Business Media, Inc., 2005.
2. Carter, A.P.: [PC Assembly Language](https://pacman128.github.io/static/pcasm-book.pdf), 2019.
3. [NASM](https://www.nasm.us/) – The Netwide Assembler, The NASM Development Team, 1996-2015.

# Adresovanie

* + špecifikácia prístupu k údajom s ktorými inštrukcie pracujú
  + polia – špecifikácia a manipulácia
  + vplyv adresovania na výkon

#### Architektúra procesorov

* + CISC (vyšší počet režimov adresovania)
  + RISC (podpora obmedzeného množstva režimov)
    - inštrukcie pracujú s operandmi v registroch CPU
    - load/store inštrukcie – presuny medzi registrami a pamäťou

#### Adresovanie procesorov Pentium (x86, CISC)

* + registrový (Register Addressing Mode) – registre CPU, rýchlosť
  + bezprostredný (Immediate Addressing Mode) – najviac jeden, operand časťou inštrukcie
  + pamäťový (Memory Addressing Mode) – množstvo režimov (spôsob špecifikácie efektívnej adresy (offset))

Pamäťový režim adresovania

* + motivácia – efektívna podpora konštrukcií HL jazykov a údajových štruktúr
  + dostupné režimy adresovania (podľa použitej veľkosti adresy)
    - 16-bit adresy (podobne ako i8086)
    - 32-bit adresy (vyššia flexibilita)

16-bit adresy [1]

Memory

Direct [disp]

Indirect

Register Indirect [BX] [BP] [SI] [DI]

Based [BX+disp] [BP+disp]

Indexed [SI+disp] [DI+disp]

Based-Indexed

Based-Indexed with no displacement

[BX+SI] [BP+SI] [BX+DI] [BP+DI]

Based-Indexed with displacement [BX+SI+disp] [BX+DI+disp] [BP+SI+disp] [BP+DI+disp]

32-bit adresy [1]

Addressing Modes

Register

Memory

Immediate

Direct [disp]

Indirect

Register Indirect [Base]

Based [Base+disp]

Indexed [(Index\*scale)+disp]

Based-Indexed

Based-Indexed with no scale factor [Base+Index+disp]

Based-Indexed with scale factor

[Base+(Index\*scale)+disp]

#### Porovnanie 16-bit a 32-bit režimov

* 32-bit režim prináša väčšiu flexibilitu v použití registrov [1]
* možnosť zohľadniť rozmer operandov (scale factor)

|  |  |  |
| --- | --- | --- |
|  | **16-bit addressing** | **32-bit addressing** |
| Base register | BX  BP | EAX, EBX, ECX, EDX  ESI, EDI, EBP, ESP |
| Index register | SI  DI | EAX, EBX, ECX, EDX  ESI, EDI, EBP |
| Scale factor | None | 1, 2, 4, 8 |
| Displacement | 0, 8, 16 bits | 0, 8, 32 bits |

* ktorý adresový režim má CPU použiť?
  + bit D segmentového deskriptora CS (D = 1: 32-bit default)
  + možnosť implicitnú voľbu explicitne zmeniť (size override prefix)
    - 66H (operand size override prefix)
    - 67H (address size override prefix)
* použitím prefixov zmeny veľkosti – možné zmiešané použitie 16/32-bit údajov a adries
* štandardne (predmet ASM) využívame 32-bit režim

|  |  |  |  |
| --- | --- | --- | --- |
| *Príklad*: asembler |  | generovaný kód |  |
| mov | EAX,123 | B8 0000007B |
| mov | AX,123 | 66 B8 007B | (prefix automaticky vlozeny) |
| mov | AL,123 | B0 7B | (odlisny op.kod) |
| mov | EAX,[BX] | 67 8B 07 |  |
| mov | AX,[BX] | 66 67 8B 07 | (obidva prefixy) |

## Bázové adresovanie

* jeden z registrov v úlohe bázy pri výpočte adresy operandu
* efektívna adresa – súčet obsahu registra a posunutia (so znamienkom)

Base + disp ; báza a posunutie so znamienkom

* prístup k prvkom štruktúry

*Príklad*: počet voľných miest v danom kurze? [1]

|  |  |  |
| --- | --- | --- |
| Course number Course title Term offered Room number  Number registered  Enrollment limit | Integer Character string Single character Character string Integer  Integer | 2 bytes  38 bytes  1 byte  5 bytes  2 bytes  2 bytes |
| Total storage per record | 50 bytes | |

|  |
| --- |
| ... |
| Enrollment |
| # registered |
| Room # |
| Term |
| Title |
| Course #  Enrollment |
| # registered |
| Room # |
| Term |
| Title |
| Course # |
| ... |

nech EBX obsahuje SSA (Structure Starting Address):

...

mov AX,[EBX + 48]

sub AX,[EBX + 46]

...

SSA+100

SSA+50

displacement 46 bytes

SSA

2

2

5 Second

course record

1 (50 bytes)

38

2

2

2

5 First

course record

1 (50 bytes)

38

2

## Indexové adresovanie

* výpočet efektívnej adresy

(Index \* scale) + disp ; posunutie so znamienkom

* prístup k prvkom poľa
  + začiatok poľa (disp), prvok v poli (index register)
  + veľkosť prvkov (scale – 2, 4, 8 – iba 32-bit režim)

|  |  |  |  |
| --- | --- | --- | --- |
| *Príklad*: | mov | EAX,[marks\_table + ESI\*4] | ; prvky marks\_table a table1 – 4B (scale) |
|  | add | EAX,[table1 + ESI] | ; ESI - offset v bytoch (napr.36 pre 10.prvok) |

## Bázovo – indexové adresovanie

* dve verzie
  + bez zohľadnenia veľkosti operandu (B-I with No Scale Factor)

Base + Index + disp ; posunutie so znamienkom (8/16 v 16-bit, 8/32 v 32-bit)

* + - dvojrozmerné polia (disp – začiatok poľa)
    - polia záznamov (disp – offset položky záznamu)
  + so zohľadnením veľkosti operandu (B-I with Scale Factor)
    - efektívny spôsob prístupu k prvkom dvojrozmerných polí (prvky veľkosti 2, 4, 8 B)

Base + (Index \* scale) + disp

*Príklad*: porovnanie dvoch nasledujúcich prvkov (4B) poľa (adresa v EBX). mov EAX,[EBX+ESI]

cmp EAX,[EBX+ESI+4]

*Príklad*: triedenie vkladaním – program získa postupnosť celých čísel, vypíše ich v utriedenom poradí (adaptované podľa [1]).

* činnosť algoritmu (vložiť nový prvok do utriedeného poľa na správnu pozíciu)
  + na začiatku prázdne pole
  + po vložení prvého prvku – utriedené

insertion\_sort(array,size)

**for**(i=1 to size-1) temp:=array[i] j:=i-1

**while**((temp<array[j])AND(j≥0)) array[j+1]:=array[j]

j:=j-1

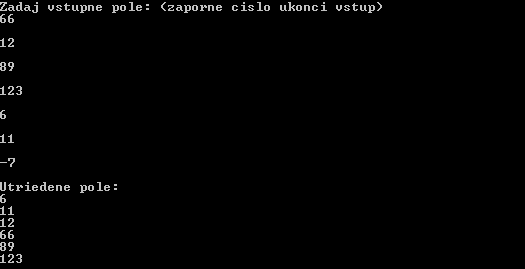
**end while**

array[j+1]:=temp

**end for**

**end** insertion\_sort

* + vložiť nový prvok na správnu pozíciu
  + opakovať proces, kým nie sú vložené všetky prvky
* pseudokód
  + index i – vkladaný prvok
  + prvky vľavo od i – utriedené
  + prvky pre vloženie – vpravo od i (vrátane)



Hlavný program:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 01 | %include "asm\_io.inc" | | 29 | exit\_loop:  mov sub shr push push call mov call call mov mov  display\_loop:  mov call call add loop  done:  popa mov leave ret |  |
| 02 | MAX\_SIZE | EQU 100 | 30 | EDX,EBX |
| 03 | segment .data | | 31 | EDX,array |
| 04 | input\_prompt | db "Zadaj vstupne pole: " | 32 | EDX,2 |
| 05 | db "(zaporne cislo ukonci vstup)",0 | | 33 | EDX |
| 06 | out\_msg | db "Utriedene pole:",0 | 34 | array |
| 07 |  | | 35 | insertion\_sort |
| 08 | segment .bss | | 36 | EAX,out\_msg |
| 09 | array | resd MAX\_SIZE | 37 | print\_string |
| 10 |  | | 38 | print\_nl |
| 11 | segment .text | | 39 | ECX,EDX |
| 12 | global \_asm\_main | | 40 | EBX,array |
| 13 | \_asm\_main: | | 41 |  |
| 14 | enter | 0,0 | 42 | EAX,[EBX] |
| 15 | pusha | | 43 | print\_int |
| 16 |  | | 44 | print\_nl |
| 17 | mov | EAX,input\_prompt | 45 | EBX,4 |
| 18 | call | print\_string | 46 | display\_loop |
| 19 | mov | EBX,array | 47 |  |
| 20 | mov | ECX,MAX\_SIZE | 48 |  |
| 21 | array\_loop: | | 49 | EAX,0 |
| 22 | call | read\_int | 50 |  |
| 23 | call | print\_nl | 51 |  |
| 24 | cmp | EAX,0 |  |  |
| 25 | jl | exit\_loop |  |  |
| 26 | mov | [EBX],EAX |  |  |
| 27 | add | EBX,4 |  |  |
| 28 | loop | array\_loop |  |  |

Procedúra insertion\_sort:

|  |  |  |  |
| --- | --- | --- | --- |
| 52 | %define SORT\_ARRAY EBX | 66 | while\_loop:  cmp EDX,[SORT\_ARRAY+EDI]  ; temp < array[j]  jge exit\_while\_loop  ; array[j+1] = array[j]  mov EAX,[SORT\_ARRAY+EDI] mov [SORT\_ARRAY+EDI+4],EAX  sub EDI,4 ; j = j-1  cmp EDI,0 ; j >= 0 jge while\_loop  exit\_while\_loop:  ; array[j+1] = temp  mov [SORT\_ARRAY+EDI+4],EDX  add ESI,4 ; i = i+1 dec ECX  cmp ECX,1  jne for\_loop sort\_done:  popad  ret 8 |
| 53 | insertion\_sort: | 67 |
| 54 | pushad | 68 |
| 55 | mov EBP,ESP | 69 |
| 56 |  | 70 |
| 57 | mov EBX,[EBP+36] | 71 |
| 58 | mov ECX,[EBP+40] | 72 |
| 59 | mov ESI,4 | 73 |
| 60 | for\_loop: | 74 |
| 61 | ; mapovanie premennych: | 75 |
| 62 | ; EDX = temp, ESI = i, EDI = j | 76 |
| 63 | mov EDX,[SORT\_ARRAY+ESI] | 77 |
| 64 | mov EDI,ESI ; j = i-1 | 78 |
| 65 | sub EDI,4 | 79 |
|  |  | 80 |
|  |  | 81 |
|  |  | 82 |
|  |  | 83 |
|  |  | 84 |
|  |  | 85 |

* procedúra nevracia hodnotu (pushad/popad)

insertion\_sort(array,size)

**for**(i=1 to size-1) temp:=array[i] j:=i-1

**while**((temp<array[j])AND(j≥0)) array[j+1]:=array[j]

j:=j-1

**end while**

array[j+1]:=temp

**end for**

**end** insertion\_sort

* prístup k parametrom (pushad – 32B)
* slučka while (r.66 – 76)
* slučka for (r.60 – 82)
* bázové adresovanie (r.57, 58)
* bázovo-indexové adresovanie (r.63, 67, 71, 72, 78)

### Polia

#### Jednorozmerné polia

* jednorozmerné pole v C (index začína 0)

int test\_marks[10];

* + HL deklarácia (veľkosť 40B)
    - meno poľa
    - počet prvkov (10)
    - veľkosť prvku (4)
    - typ prvku (int)
    - indexy (0 – 9)
* pole v asembleri – alokácia požadovaného miesta [1]

test\_marks resd 10

High memory

Low memory

← test\_marks

|  |
| --- |
| test\_marks[9] |
| test\_marks[8] |
| test\_marks[7] |
| test\_marks[6] |
| test\_marks[5] |
| test\_marks[4] |
| test\_marks[3] |
| test\_marks[2] |
| test\_marks[1] |
| test\_marks[0] |

* + korektný prístup k prvkom – úloha programátora (indexy, veľkosť prvkov)
  + prvky usporiadané lineárne
  + potrebné zistiť posun od začiatku poľa (posun = index \* veľkosť prvku)

#### Viacrozmerné polia

* dvojrozmerné pole v C (5r x 3s)

int class\_marks[5][3];

* + reprezentácia v pamäti (lineárne pole bajtov, transformácia)

1. po riadkoch (row-major ordering, napr. C)
2. po stĺpcoch (column-major ordering, napr. Fortran)

* dvojrozmerné pole v asembleri [1]
  + reprezentácia poľa v pamäti podstatná
  + alokácia (60B)

class\_marks resd 5\*3

* + preklad indexov na hodnotu posunu (po riadkoch)

posun = (i \* COLS + j) \* ELM\_SIZE

* + COLS – počet stĺpcov, i – riadok, j – stĺpec

High memory High memory

r4

|  |
| --- |
| class\_marks[4,2] |
| class\_marks[4,1] |
| class\_marks[4,0] |
| class\_marks[3,2] |
| class\_marks[3,1] |
| class\_marks[3,0] |
| class\_marks[2,2] |
| class\_marks[2,1] |
| class\_marks[2,0] |
| class\_marks[1,2] |
| class\_marks[1,1] |
| class\_marks[1,0] |
| class\_marks[0,2] |
| class\_marks[0,1] |
| class\_marks[0,0] |

|  |
| --- |
| class\_marks[4,2] |
| class\_marks[3,2] |
| class\_marks[2,2] |
| class\_marks[1,2] |
| class\_marks[0,2] |
| class\_marks[4,1] |
| class\_marks[3,1] |
| class\_marks[2,1] |
| class\_marks[1,1] |
| class\_marks[0,1] |
| class\_marks[4,0] |
| class\_marks[3,0] |
| class\_marks[2,0] |
| class\_marks[1,0] |
| class\_marks[0,0] |

r3

r2

r1

r0

class\_marks →

Low memory Low memory

1. Row-major order (b) Column-major order

# Celočíselná aritmetika

* + vplyv realizácie aritmetických a logických inštrukcií na stavové príznaky (FLAGS)
  + násobenie a delenie
  + aritmetické operácie nad viacslovnými údajmi (multi-word arithmetic)

### Stavové príznaky

* + 6 príznakov – monitorovanie výsledkov operácií
  + ZF, CF, OF, SF, AF, PF
  + ak je príznak aktualizovaný – ostáva nezmenený, pokiaľ jeho stav nezmení ďalšia inštrukcia
    - nie všetky inštrukcie ovplyvňujú stavové príznaky (add, sub – 6; inc, dec – okrem CF; mov, push – žiadne)
  + príznaky možno testovať (individuálne, kombinácie) za účelom riadenia vykonávania programu

#### Príznak nuly (zero flag)

* + výsledkom poslednej operácie (ovplyvňujúcej ZF) bola 0 – ZF = 1, ináč ZF = 0
  + sub – intuitívne, iné inštrukcie možno menej

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Príklad*:  mov | AL,0FH |  | | | |
| add | AL,0F1H | (nastaví | ZF | = | 1, všetkých 8 bitov AL – 0) |
| mov inc | AX,0FFFFH AX | (nastaví | ZF | = | 1) |
| mov dec | EAX,1 EAX | (nastaví | ZF | = | 1) |

* + inštrukcie podmienených skokov: jz (ak ZF = 1), jnz (ak ZF = 0)
  + využitie príznaku ZF
    - test rovnosti (často inštrukcia cmp) cmp char,'$'

cmp EAX,EBX

* + - počítanie do zadanej hodnoty [1]
      * M, N ≥ 1, vnútorná slučka (ECX/loop – neovplyvňuje príznaky)
      * vonkajšia slučka (EDX/dec/jnz)

outer\_loop:

mov ECX,N inner\_loop:

inc EAX

loop inner\_loop dec EDX

jnz outer\_loop exit\_loops:

mov sum,EAX

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| sub | EAX,EAX | ; | EAX | = | 0 |
| mov | EDX,M |  |  |  |  |

sum = 0

**for**(i=1 to M)

**for**(j=1 to N)

sum=sum+1

**end for end for**

#### Príznak prenosu (carry flag)

* + výsledok aritmetickej operácie na bezznamienkových číslach prekročil rozsah cieľa (R/M)

*Príklad*:

mov AL,0FH 00001111

add AL,0F1H 11110001

100000000

* + v prípade 8-bit registra potreba 9.bitu (AL – 8-bitov)
  + rozsah hodnôt bezznamienkových čísel [1]

|  |  |
| --- | --- |
| Size (bits) | Range |
| 8 | 0 to 255 |
| 16 | 0 to 65 535 |
| 32 | 0 to 4 294 967 295 |

* + operácia produkujúca výsledok mimo rozsahu nastaví CF
  + záporný výsledok teda mimo rozsahu

*Príklad*:

mov EAX,12AEH mov EAX,0

sub EAX,12AFH dec EAX

(4782 - 4783 = -1, CF = 1) (CF = 0, inc, dec – neovplyvňujú CF)

* + inštrukcie podmienených skokov: jc (ak CF = 1), jnc (ak CF = 0)
  + využitie CF
    - šírenie prenosu/výpožičky (carry/borrow) pri viacslovných operáciách sčítania/odčítania
      * inštrukcie – operandy rozmeru 8, 16, 32b, pri rozmernejších – po krokoch, zohľadnenie prenosov
    - detekcia pretečenia/podtečenia (overflow/underflow)
      * indikácia výsledku mimo rozsahu (ošetrenie situácie programátorom)
    - test bitu s využitím posunov/rotácií
      * bit (MSb, LSb) zachytený v CF – možno využiť, podmienené skoky (podmienečné vykonanie kódu)
  + inštrukcie inc, dec neovplyvňujú CF
    - často počet iterácií slučiek (32b hodnota postačuje pre väčšinu aplikácií)
    - podmienka detegovaná CF detegovateľná aj pomocou ZF (nastavenie CF redundantné)
      * ak ECX = FFFFFFFFH a vykoná sa

inc ECX

* + - * očakávame CF = 1, ale rovnako možno detegovať túto podmienku pomocou ZF (ECX = 0)

#### Preplnenie (overflow flag)

* + obdoba CF pre operácie s číslami so znamienkom (signed)
  + indikácia výsledku operácie mimo platného rozsahu
  + rozsahy čísel so znamienkom [1]

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Size (bits) | Range | | | | | | | | |
| 8 | -128 to +127 | | | | | | | | |
| 16 | -32 768 to +32 767 | | | | | | | | |
| 32 | -2 | 147 | 483 | 648 | to | +2 | 147 | 483 | 647 |

*Príklad*:

mov AL,72H

add AL,0EH (114 + 14 = 128, OF = 1)

* + 128 (80H) je korektný výsledok súčtu bezznamienkových čísel
  + pri znamienkovej interpretácii nekorektný: 80H predstavuje –128

*Znamienková/bezznamienková interpretácia*

* + ako systém rozpozná spôsob interpretácie reťazca bitov programom? (nijako)
  + procesor zohľadňuje obidve interpretácie – podľa toho nastavuje CF a OF

mov AL,72H

add AL,0EH (114 + 14 = 128: CF = 0, OF = 1)

* + zohľadnenie príslušného bitu je úlohou programátora
  + inštrukcie podmienených skokov: jo (ak OF = 1), jno (ak OF = 0)
  + inštrukcia SW prerušenia: into (interrupt on overflow, generuje INT 4)

#### Znamienko (sign flag)

* + znamienko výsledku operácie
  + užitočný len pri znamienkovej interpretácii
  + kópia najvyššieho bitu výsledku

|  |  |  |  |
| --- | --- | --- | --- |
| *Príklad*:  mov | EAX,15 |  | |
| add | EAX,97 | (15 + 97 = 112, | SF = 0) |
| mov sub | EAX,15 EAX,97 | (15 – 97 = -82, | SF = 1) |

15 + (-97): 00001111 (15)

10011111 (-97, c-repr.)

10101110 (-82, c-repr.)

* + inštrukcie podmienených skokov: js (ak SF = 1), jns (ak SF = 0)
  + použitie
    - znamienko výsledku
    - slučky s hodnotou riadiacej premennej klesajúcou k nule (vrátane) [1]

**for**(i=M downto 0)

<loop body>

**end for**

mov ECX,M for\_loop:

...

<loop body>

...

dec ECX

jns for\_loop

#### Pomocný prenos (auxiliary carry flag)

* + prenos z (výpožička do) nižších 4 bitov (nibble) operandu

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| mov | AL,43 | 00101011 (43) | mov | AL,43 |  |  |  | 00101011 | (43) |
| add | AL,94 (AF = 1) | 01011110 (94) | add | AL,84 | (AF | = | 0) | 01010100 | (84) |
|  |  | 10001001 (137) |  |  |  |  |  | 01111111 | (127) |

* + príbuzné inštrukcie
    - neexistencia podmienených skokov s testom AF
    - aritmetické operácie s číslami v BCD formáte využívajú AF
      * aaa, aas, aam, aad (ASCII adjust for addition, subtraction, …) [3]
      * daa, das (decimal adjust for addition, ...)

#### Parita (parity flag)

* + parita 8-bit výsledku operácie (iba spodných 8 bitov má vplyv na PF)
  + párny počet 1 (PF = 1), nepárny (PF = 0)

|  |  |  |  |
| --- | --- | --- | --- |
| mov | AL,53 |  | 00110101 (53) |
| add | AL,89 | (PF = 1) | 01011001 (89) |
|  |  |  | 10001110 (142) |

* + príbuzné inštrukcie – skoky: jp (ak PF = 1), jnp (ak PF = 0)
  + použitie (napr. kódovanie údajov)

*Príklad*: prenos cez modem využívajúci 7-bit ASCII kód [1]

* + detekcia jednoduchých chýb prenosu – pridanie paritného bitu (k 7-bit údaju)
  + predpokladáme kódovanie párnej parity (doplnenie 8.bitu podľa potreby)
  + prijímač spočíta počet jednotiek v prijatom bajte (chyba, ak obsahuje nepárny počet)
    - A – 41H (kód: 01000001, MSb – 0)

parity\_encode PROC shl AL

jp parity\_zero

stc ; CF=1 jmp move\_parity\_bit

parity\_zero:

clc ; CF=0 move\_parity\_bit:

rcr AL parity\_encode ENDP

* + - C – 43H (kód: 11000011, MSb – 1, nastavený)

*Príklad*: vplyv realizácie aritmetických operácií na príznaky [1]

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Code | | AL | CF | ZF | SF | OF | PF |
| Example1 | mov sub | AL,-5 AL,123 | 80H | 0 | 0 | 1 | 0 | 0 |
| Example2 | mov sub | AL,-5 AL,124 | 7FH | 0 | 0 | 0 | 1 | 0 |
|  | mov | AL,-5 |  |  |  |  |  |  |
| Example3 | add | AL,132 | 7FH | 1 | 0 | 0 | 1 | 0 |
|  | add | AL,1 | 80H | 0 | 0 | 1 | 1 | 0 |
| Example4 | sub | AL,AL | 00H | 0 | 1 | 0 | 0 | 1 |
| Example5 | mov add | AL,127 AL,129 | 00H | 1 | 1 | 0 | 0 | 1 |

### Aritmetické inštrukcie

* + sčítanie (add, adc, inc) a odčítanie (sub, sbb, dec, neg, cmp)
  + násobenie a delenie (mul, imul, div, idiv)
  + príbuzné inštrukcie (cbw, cwd, cdq, cwde, movsx, movzx)
  + inštrukcia neg – zmena znamienka operandu (IF dst = 0 THEN CF ← 0; ELSE CF ← 1; dst = -dst) neg dst (dst - 8, 16, 32-bit GPR, pamäť)

## Inštrukcie pre násobenie

* + vlastnosti operácie násobenia
    - rozmer výsledku (2*n* bitov pri násobení dvoch *n*-bit. čísel)
    - násobenie čísel so znamienkom realizované odlišne od násobenia čísel bez znamienka (dôsledok – odlišné inštrukcie mul/imul)
  + násobenie čísel bez znamienka (mul)
    - syntax

mul src (src – 8, 16, 32-bit GPR, pamäť)

* + - sémantika (podľa rozmeru src)
      * 8 bitov: AX ← src \* AL
      * 16 bitov: DX:AX ← src \* AX
      * 32 bitov: EDX:EAX ← src \* EAX
    - inštrukcia ovplyvňuje všetky stavové (6) príznaky, nastavuje však len CF a OF, zvyšné nedefinované
      * CF a OF nastavené, ak vrchná (upper half) časť výsledku je nenulová (AH, DX, EDX)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Príklad*: | mov | AL,10 |  | mov | AL,10 |  |
|  | mov | DL,25 |  | mov | DL,26 |  |
|  | mul | DL | ; CF = OF = 0 | mul | DL | ; CF = OF = 1 |

* + násobenie čísel so znamienkom (imul)
    - syntax (podobne ako mul, podpora ďalších formátov, napr. bezprostredný údaj ako parameter)
    - CF, OF – nastavené, ak vrchná časť výsledku nie je znamienkovým rozšírením spodnej

*Príklad*: znamienkové rozšírenie hodnoty –66

(-66)10 = (10111110)2 8-bit

(-66)10 = (1111111110111110)2 16-bit

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Príklad*: | mov | DL,0FFH | ; DL = -1 |  |
|  | mov | AL,42H | ; AL = 66 |  |
|  | imul | DL | ; AX = -66 | (1111111110111110)2, CF = OF = 0 |

## Inštrukcie pre delenie

* + vlastnosti operácie delenia
    - výsledkom sú dve hodnoty – podiel a zvyšok
    - pri násobení (výsledok s dvojnásobnou dĺžkou) pretečenie sa nevyskytuje, pri delení sa môže vyskytnúť (divide overflow)
  + syntax

|  |  |  |
| --- | --- | --- |
| div | src | (bezznamienkové, src – 8, 16, 32-bit GPR, pamäť) |
| idiv | src | (znamienkové) |

* + sémantika inštrukcie div (podľa rozmeru deliteľa, src)
    - 8 bitov: AL ← quot(AX/src), AH ← rem(AX/src)
    - 16 bitov: AX ← quot(DX:AX/src), DX ← rem(DX:AX/src)
    - 32 bitov: EAX ← quot(EDX:EAX/src), EDX ← rem(EDX:EAX/src)
  + príznaky – ovplyvnené inštrukciami – nedefinované
  + sémantika inštrukcie idiv – rovnaký formát a správanie ako div
    - komplikácia – ak delenec je záporný – potrebné znamienkové rozšírenie

*Príklad*: delenie –251/12 (16-bit)

* + - (–251) = FF14H, preto DX inicializovaný na FFFFH
    - ak DX inicializovaný na 0000H (ako pri div), DX:AX reprezentuje kladné číslo!
    - ak delenec kladný – DX má byť 0000H
  + inštrukcie pre znamienkové rozšírenie
    - cbw (convert byte to word) – rozšírenie AL do AH (8-bit idiv)
    - cwd (convert word to doubleword) – rozšírenie AX do DX (16-bit idiv)
    - cdq (convert doubleword to quadword) – rozšírenie EAX do EDX (32-bit idiv)
  + ďalšie príbuzné inštrukcie
    - cwde – znamienkové rozšírenie AX do EAX
    - movsx dst,src (move sign-extended src to dst)
      * dst – R, src – R/M, ak src 8-bit → dst 16-bit alebo 32-bit, ak src 16-bit → dst 32-bit
    - movzx dst,src (move zero-extended src to dst)
      * ako movsx

*Príklad*: 16-bit. znamienkové delenie

|  |  |  |
| --- | --- | --- |
| mov cwd | AX,-5147 | ; DX = FFFFH |
| mov | CX,300 |  |
| idiv | CX | ; AX = FFEFH (-17) podiel, DX = FFD1H (-47) zvyšok |

#### Použitie posunov pre násobenie a delenie

* + efektívna alternatíva pre realizáciu týchto operácií; ak je to vhodné, možno využiť (násobenie/delenie mocninou 2)

*Príklad*: AX \* 32, 2 alternatívy (b – rýchlosť, miesto)

a) mov CX,32 b) sal AX,5 mul CX

### Aritmetické operácie nad viacslovnými údajmi (multi-word arithmetic)

* + aritmetické inštrukcie pracujú s údajmi o veľkosti 8, 16, 32-bit (rozmernejšie údaje – problém)
  + základy viacslovnej aritmeticky (sčítanie, odčítanie, násobenie, delenie)

#### Sčítanie a odčítanie (64-bit, unsigned)

* + relatívne nenáročné operácie, sčítanie – sčítame pravých 32 bitov, v ďalšom kroku ľavých (s prenosom z prvého kroku)

*Príklad*: sčítanie dvoch 64-bit čísel v EBX:EAX a EDX:ECX, výsledok v EBX:EAX. Pretečenie indikované pomocou CF.

add64: add EAX,ECX ; odčítanie – podobne (add→sub, adc→sbb) adc EBX,EDX

ret

#### Násobenie (64-bit, unsigned)

* + viacero známych prístupov, uvádzaný algoritmus – tzv. longhand multiplication [1]
  + testujú sa bity násobiteľa (A) sprava a podľa ich hodnoty sa (ne)pripočítava násobenec (B)
* činnosť algoritmu (4-bit čísla A = 13, B = 5)

P:=0

A:=multiplier B:=multiplicand count:=64 **while**(count>0)

**if**(LSB of A = 1)

**then**

P:=P+B

CF:=carry generated by P+B

**else**

CF:=0

**end if**

shift right CF:P:A by one bit position

{LSB of multiplier is not used in the rest of the algorithm} count:=count-1

**end while**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | After P+B | | | After the shift | | |
| CF | P | A | CF | P | A |
| init.state | ? | 0000 | 1101 | - | - | - |
| iterat. 1 | 0 | 0101 | 1101 | ? | 0010 | 1110 |
| iterat. 2 | 0 | 0010 | 1110 | ? | 0001 | 0111 |
| iterat. 3 | 0 | 0110 | 0111 | ? | 0011 | 0011 |
| iterat. 4 | 0 | 1000 | 0011 | ? | 0100 | 0001 |

* implementácia (dve 64-bit čísla v EBX:EAX (A) a EDX:ECX (B), 128-bit výsledok: EDX:ECX:EBX:EAX)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| %define COUNT | word[EBP-2] | ; | lokalna premenna | shift1: |  |
| mult64: |  |  |  | rcr | EDX,1 |
| enter | 2,0 | ; | 2-byte | rcr | ECX,1 |
| push | ESI |  |  | rcr | EBX,1 |
| push | EDI |  |  | rcr | EAX,1 |
| mov | ESI,EDX | ; | ESI:EDI = B |  |  |
| mov | EDI,ECX |  |  | dec | COUNT |
| sub | EDX,EDX | ; | P = 0 | jnz | step |
| sub | ECX,ECX |  |  | pop | EDI |
| mov | COUNT,64 | ; | 64-bit cislo | pop | ESI |
| step: |  |  |  | leave |  |
| test | EAX,1 | ; | LSB A je 1? | ret |  |
| jz | shift1 | ; | ak nie, nepricitaj |  |  |
| add | ECX,EDI | ; | P = P + B |  |  |
| adc | EDX,ESI |  |  |  |  |

* mapovanie premenných:

P:=0

A:=multiplier B:=multiplicand count:=64 **while**(count>0)

**if**(LSB of A = 1)

**then**

P:=P+B

CF:=carry generated by P+B

**else**

CF:=0

**end if**

shift right CF:P:A by one bit position

{LSB of multiplier is not used in the rest of the algorithm} count:=count-1

**end while**

* + ESI:EDI – B, EBX:EAX – A, EDX:ECX – P
* lokálna premenná COUNT
* 64-bit sčítanie (P + B)
* posun vpravo CF:P:A (rcr)
  + inštrukcia test nuluje CF

#### Delenie (*n*-bit unsigned)

* niekoľko známych algoritmov ('nonrestoring' division algorithm) [1]
* A – delenec, B – deliteľ, A ← quot(A/B), P ← rem(A/B)
* uvádzame iba pseudokód (bez asm-implementácie)

P:=0

A:=dividend B:=divisor count:=64 **while**(count>0)

**if**(P is negative)

**then**

shift left P:A by one bit position P:=P+B

**else**

shift left P:A by one bit position P:=P-B

**end if**

**if**(P is negative)

**then**

set low-order bit of A to 0

**else**

set low-order bit of A to 1

**end if** count:=count-1 **end while**

**if**(P is negative)

P:=P+B

**end if**

* pomocný register (P) dĺžky *n*+1 bitov
* metóda
  + testuje sa znamienko P, podľa neho P ← P + B, alebo P ← P – B
  + P:A posunutý vľavo s manipuláciou LSb A
  + po *n*-opakovaniach, podiel v A, zvyšok v P

*Príklad*: činnosť algoritmu (4-bit. čísla A = 0101 (5), B = 0010 (2); pomocný register P = 00000 (0))

|  |  |  |  |
| --- | --- | --- | --- |
|  | P:A | P:A (shift left) | P ± B:A |
| count = 4 | 00000:0101 | 00000:1010 | 11110:1010 |
| count = 3 | 11110:1010 | 11101:0100 | 11111:0100 |
| count = 2 | 11111:0100 | 11110:1000 | 00000:1000 |
| count = 1 | 00000:1001 | 00001:0010 | 11111:0010 |
|  |  |  | 00001:0010 |

–1 = (11111)C výsledok: A = 2 (podiel)

–2 = (11110)C P = 1 (zvyšok)

–3 = (11101)C

#### Študijná literatúra:

1. Dandamudi, S.P.: Introduction to Assembly Language Programming, Springer Science+Business Media, Inc., 2005.
2. Carter, A.P.: [PC Assembly Language](https://pacman128.github.io/static/pcasm-book.pdf), 2019.
3. Cloutier, F.: [x86 and amd64 instruction reference](https://www.felixcloutier.com/x86/), 2022.

# Skoky a iterácie

* + podmienené a nepodmienené skoky
  + iterácie (LOOP)
  + implementácia riadiacich štruktúr HL jazykov

### Nepodmienený skok

* + nepodmienený prenos riadenia na špecifikovaný cieľ, syntax:

jmp cieľ

#### Špecifikácia cieľa

* + cieľová adresa špecifikovaná
    - priamo (časť inštrukcie, dopredu/späť – znamienková reprezentácia)
    - nepriamo (register/pamäť obsahuje adresu)
  + priame skoky (direct jumps)
    - adresa špecifikovaná v inštrukcii – relatívny posun medzi cieľom a inštrukciou nasledujúcou za jmp (!)
    - po výbere inštrukcie jmp, EIP automaticky aktualizovaný
    - posun – číslo so znamienkom (kladné – skok vpred)
    - relatívne adresy – vhodné pre dynamicky relokovateľný kód (position-independent code)
  + cieľ skoku
    - v rámci segmentu – cieľ v rovnakom segmente ako inštrukcia jmp (intrasegment jump)
      * doposiaľ uvažovaný tento typ
      * EIP ← EIP + rel. posun
    - v inom segmente (intersegment jump, far jump)
      * CS ← cieľ. segment
      * EIP ← cieľ. offset
      * segment aj offset špecifikované v rámci inštrukcie (pre 32-bit. segment inštrukcia – 7B)
  + väčšina skokov – v rámci segmentu, 2 spôsoby špecifikácie cieľa podľa veľkosti rel. posunu [[4](https://www.felixcloutier.com/x86/jmp)]
    - short jump (2B, 1B – op. kód (EBH) + 1B – rel. posun, znamiekové číslo v rozsahu –128/+127)
    - near jump (3/5B, 1B – op. kód (E9H) + 2/4B – rel. posun)
      * 2B rel. posun pre 16-bit. segmenty, 4B pre 32-bit segmenty
  + špecifikácia krátkych skokov (SHORT)
    - chceme použiť krátky skok – informácia pre prekladač:

jmp SHORT ECX\_init\_done

* + - ak cieľ je vzdialenejší – chybové hlásenie
    - asembler automaticky doplní SHORT pre skoky vzad (ak je cieľ v platnom rozsahu)
    - skoky vpred – asembler nepozná vzdialenosť cieľa, pomoc programátora vítaná

*Príklad*: kódovanie short/near skokov (adaptované [1])

* + r.8 (2B) – špecifikovaný ako SHORT, op. kód EBH, ofset 14H
  + r.10 (5B) – špecifikovaný ako NEAR, op. kód E9H, offset 0000000AH
  + r.18 (2B) – skok späť, asembler dokáže rozhodnúť, že SHORT stačí (FDH = –3)
  + r.13 (5B) – NEAR, malý endian, offset 00000017H

...

1. ; unconditional jumps encoding
2. 00000000 EB14 jmp SHORT ECX\_init\_done
3. 00000002 B978563412 mov ECX,12345678H
4. 00000007 E90A000000 jmp NEAR ECX\_init\_done
5. init\_ECX:
6. 0000000C B912EFCDAB mov ECX,0ABCDEF12H
7. 00000011 E917000000 jmp NEAR near\_jump
8. ECX\_init\_done:

15 00000016 89C8 mov EAX,ECX

16 repeat1:

17 00000018 49 dec ECX

18 00000019 EBFD jmp repeat1

...

24 00000021 EB05 jmp short\_jump

25 00000023 B9FFFF00FF mov ECX,0FF00FFFFH

26 short\_jump:

27 00000028 BA32547698 mov EDX,98765432H

1. near\_jump:
2. 0000002D EBDD jmp init\_ECX

...

* + nasm – možnosť generovať tzv. listing file pomocou voľby -l, napr.

nasm -f win32 L5\_jumps\_encoding.asm -l L5\_jumps\_encoding.lst

### Inštrukcia porovnania (cmp)

* + nastavenie príznakov, nasledujúca inštrukcia podmieneného skoku ich otestuje (bolo prediskutované, prednáška 4)
  + implementácia HL konštrukcie IF-THEN-ELSE v asembleri v dvoch krokoch
    - aritmetická/porovnávacia inštrukcia
    - podmienený skok

### Podmienené skoky

* + možno rozdeliť do troch skupín
    - podľa hodnoty jedného príznaku
    - podľa výsledku bezznamienkových porovnaní
    - podľa výsledku znamienkových porovnaní

#### Skoky podľa hodnoty jedného príznaku

* + dve inštrukcie (0/1) pre každý stavový príznak okrem AF
  + výskyt dvojitých pomenovaní (alias) pre ZF, PF
  + príznak nuly (ZF)
    - jz, je (ZF = 1)
    - jnz, jne (ZF = 0)
    - jecxz (jump if ECX = 0, bez testovania príznakov), jcxz (if CX = 0)
  + prenos (CF)
    - jc (CF = 1), jnc (CF = 0)
  + preplnenie (OF)
    - jo (OF = 1), jno (OF = 0)
  + znamienko (SF)
    - js (SF = 1), jns (SF = 0)
  + parita (PF)
    - jp, jpe (PF = 1)
    - jnp, jpo (PF = 0)

#### Skoky podľa výsledku bezznamienkových porovnaní

* + pri porovnaní dvoch čísel (cmp num1,num2) – znamienkové alebo bezznamienkové čísla?

*Príklad*: AL = 10110111 (183/–73)10, DL = 01101110 (110)10

cmp AL,DL

AL > DL (bezznamienková interpretácia) AL < DL (znamienková interpretácia)

* + poradie v porovnaní (cmp num1,num2) – vždy vzťah num1 k num2, možné relácie (6):
    - num1 = num2, num1 ≠ num2
    - num1 > num2, num1 ≥ num2
    - num1 < num2, num1 ≤ num2
  + pre čísla bez znamienka CF a ZF relevantné, výskyt synoným (aliases)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *mnemonika* | **je/jz** | **jne/jnz** | **ja/jnbe** | **jae/jnb** | **jb/jnae** | **jbe/jna** |
| *význam* | equal/ zero | not equal/ not zero | above/  not below or equal | above or equal/ not below | below/  not above or equal | below or equal/ not above |
| *podmienka* | ZF = 1 | ZF = 0 | CF = 0 AND ZF = 0 | CF = 0 | CF = 1 | CF = 1 OR ZF = 1 |

#### Skoky podľa výsledku znamienkových porovnaní

* + porovnania =, ≠ pracujú rovnako na číslach so znamienkom i bez znamienka
  + pre čísla so znamienkom relevantné SF, OF, ZF

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *mnemonika* | **je/jz** | **jne/jnz** | **jg/jnle** | **jge/jnl** | **jl/jnge** | **jle/jng** |
| *význam* | equal/ zero | not equal/ not zero | greater/  not less or equal | greater or equal/ not less | less/  not greater or equal | less or equal/ not greater |
| *podmienka* | ZF = 1 | ZF = 0 | ZF = 0 AND SF = OF | SF = OF | SF ≠ OF | ZF = 1 OR SF ≠ OF |

* + predpokladajme inštrukciu cmp snum1,snum2; 8-bitové operandy [1]:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **snum1** | **snum2** | **ZF** | **OF** | **SF** |
| 56 | 55 | 0 | 0 | 0 |
| 56 | -55 | 0 | 0 | 0 |
| -55 | -56 | 0 | 0 | 0 |
| 55 | -75 | 0 | 1 | 1 |

* + - podmienky pre snum1 > snum2 (jg)
    - podmienky pre snum1 < snum2 (jl, ZF – redundantný, ZF = 1 → SF = OF = 0)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **snum1** | **snum2** | **ZF** | **OF** | **SF** |
| 55 | 56 | 0 | 0 | 1 |
| -55 | 56 | 0 | 0 | 1 |
| -56 | -55 | 0 | 0 | 1 |
| -75 | 55 | 0 | 1 | 0 |

#### Inštrukcia setCC (Set Byte on Condition) [[6](https://www.felixcloutier.com/x86/setcc)]

* + nastaví dest na 1 ak je splnená podmienka CC, ináč nastaví dest na 0

setCC dest (dest – 8-bit register, pamäť)

* + podmienka CC reprezentuje jednotlivé príznaky, ako aj bezznamienkové/znamienkové porovnania (SETZ, SETNZ, SETA, SETG, ...)

## Vzdialenosť cieľa podmienených skokov

* + podmienené skoky – SHORT/NEAR [[5](https://www.felixcloutier.com/x86/jcc)]
  + najefektívnejšie, ak sú kódované ako 2B inštrukcie (SHORT – 1B op. kód + 1B offset)
    - rozsah –128/127 B (SHORT)
    - pri prekročení tejto vzdialenosti – možnosť náhrady (negácia podmienky + nepodmienený skok)
  + podmienené skoky NEAR – op. kód 2B + offset 2/4B

*Príklad*: kód vľavo – možnosť náhrady kódom vpravo [1]

...

target1:

cmp jne jmp

... AX,BX

skip1 ; skip1 is a short jump target1

skip1:

mov

CX,20

...

...

target1:

cmp je mov

... AX,BX

target1 ; not a short jump CX,20

...

### Iterácie

* + inštrukcie iterácií využívajú CX/ECX register (počet opakovaní) podľa veľkosti operandu (v ďalšom predpokladáme 32-bit)
  + dekrementujú register pred testom na nulu (bez ovplyvnenia príznakov)
  + vzdialenosť cieľa v dosahu –128/127 B (1B posun)

#### Inštrukcie loop, loope/loopz, loopne/loopnz

* + synonymá (aliases), syntax

loop cieľ

loope cieľ loopne cieľ

* + inštrukcie loope/loopz, loopne/loopnz – podpora cyklov s dvoma podmienkami terminácie

|  |  |  |  |
| --- | --- | --- | --- |
| *mnemonika* | **loop** | **loope/loopz** | **loopne/loopnz** |
| *význam* | loop | loop while equal/ loop while zero | loop while not equal/ loop while not zero |
| *sémantika* | ECX = ECX – 1  IF ECX ≠ 0 skok na cieľ | ECX = ECX – 1  IF (ECX ≠ 0 AND ZF = 1) skok na cieľ | ECX = ECX – 1  IF (ECX ≠ 0 AND ZF = 0) skok na cieľ |

*Príklad*: program číta z klávesnice čísla, končí po zadaní stanoveného počtu čísel (SIZE), alebo nuly

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| %include "asm\_io.inc" | | | read\_more:  call mov add cmp loopne popa mov leave ret |  |
| SIZE | EQU | 10 | read\_int [EBX],EAX EBX,4 EAX,0  read\_more  EAX,0 |
| segment .bss | | |
| buffer | resd | SIZE |
| segment .text | | |
| global \_asm\_main | | |
| \_asm\_main: | | |
|  | enter | 0,0 |
| pusha | | |  |
|  | mov | EBX,buffer |  |
|  | mov | ECX,SIZE |

* + problém: ak na začiatku ECX = 0 (FFFFFFFFH opakovaní / nula na vstupe), riešenie: inštrukcia jecxz pred vstupom do slučky
  + rýchlosť vykonávania inštrukcií loop a jcxz (pre potreby optimalizácie, údaje platné pre procesor Pentium) [3]
    - dvojica inštrukcií (spolu 2 hod. cykly) sa vykoná rýchlejšie ako zodpovedajúca (loop cieľ, 5/6 cyklov)

dec ECX

jnz cieľ

* + - dvojica inštrukcií (spolu 2 hod. cykly) sa vykoná rýchlejšie ako zodpovedajúca (jecxz cieľ, 5/6 cyklov)

cmp ECX,0

jz cieľ

* + určenie časovania inštrukcií pre rôzne x86/x86-64 procesory náročnejšie, detailné informácie napr. v [9]

### Implementácia riadiacich štruktúr HL jazykov

* + s využitím inštrukcií porovnania, skokov a iterácií, príklady inšpirované [1]
  + gcc – možnosť generovať asm kód pomocou voľby -S, napr. gcc -S -masm=intel L5\_hl\_constr\_1.c -m32

#### Konštrukcia if-then-else

if (condition) then

true-alternative

else

false-alternative end if

*Príklad*: konštrukcia if a relačný operátor (C kód – priradí väčšiu z dvoch hodnôt (typ int) premennej bigger)

a) kód v jazyku C b) po preklade (GCC: (GNU) 9.1.0)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ... |  | | |
| mov | eax,value1 | ; | value1 |
| cmp | eax,value2 | ; | value2 |
| jle | L2 |  |  |
| mov | eax,value1 | ; | redundant |
| mov | bigger,eax | ; | then part |
| jmp | L1 |  |  |
| L2: |  |  |  |  |
|  | mov | eax,value2 | ; | else part |
|  | mov | bigger,eax |  |  |
| L1: |  |  |  |  |
|  | ... |  |  |  |

if(value1 > value2) bigger = value1;

else

bigger = value2;

* + podmienka testovaná pomocou cmp/jle
  + generovaný redundantný kód

*Príklad*: konštrukcia if a logický operátor (&&) – (test, či sa jedná o malé písmeno a prípadný prevod na veľké) [1]

a) kód v jazyku C b) po preklade (GCC: (GNU) 9.1.0)

if((ch >= ’a’) && (ch <= ’z’)) ch = ch – 32;

...

cmp jle cmp jg movzx sub mov

byte\_ch,96 L4

byte\_ch,122 L4

eax,byte\_ch eax,32 byte\_ch,al

; 'a' = 97

; not lower case

; 'z'

; not lower case

; lower case

L4:

...

* + zložená podmienka – dva páry cmp/jx inštrukcií
  + generovaný redundantný kód

Iteratívne konštrukcie

* + konštrukcie ako while, repeat-until, for

#### Slučka while

* + test podmienky pred vykonaním tela slučky (entry-test loop)
  + telo slučky sa vykonáva opakovane, pokiaľ podmienka je splnená

a) kód v jazyku C b) po preklade (GCC: (GNU) 9.1.0)

while(total < 700)

{

<loop body>

}

...

jmp

L2

; test condition

L3:

<while loop body>

...

; loop body

L2:

cmp jle

...

total,699 L3

; while condition

* + nepodmienený prenos riadenia na začiatku (test podmienky)

#### Slučka repeat-until

* + podmienka testovaná po vykonaní tela slučky (exit-test loop)
  + príkazy v tele teda vykonané aspoň raz

a) kód v jazyku C b) po preklade (GCC: (GNU) 9.1.0)

do

{

<loop body>

}

while (number > 0);

...

L2:

<do-while loop body>

...

; loop body

cmp jg

...

number,0 L2

; cond. test

#### Slučka for

* + pevne daný počet iterácií (counting loop)

a) kód v jazyku C b) po preklade (GCC: (GNU) 9.1.0)

...

mov jmp

var\_i,0 L2

; i = 0

L3:

<for loop body>

...

; loop body

add var\_i,1 ; increment i

L2:

cmp jle

...

var\_i,SIZE-1 L3

; for condition

for(i = 0; i < SIZE; i++)

{

<loop body>

}

* + nepodmienený skok na začiatku (test podmienky)
  + zvýšenie i (zníženie podobne)

### Nepriame skoky

* + doposiaľ inštrukcie priamych skokov (adresa cieľa zakódovaná v inštrukcii samotnej)
  + uvažujeme skoky v rámci segmentu
  + adresa cieľa špecifikovaná R/M [4]
  + špecifikovaná absolútna hodnota posunu (offset, u priamych skokov relatívna)
  + použitie

jmp [ECX]

#### Viaccestné vetvenie (multiway conditional execution)

...

L7:

eax,count[4] eax,1 count[4],eax

mov add mov

L3:

eax,count[3] eax,1 count[3],eax L7

mov add mov jmp

L2:

eax,count[2] eax,1 count[2],eax L7

mov add mov jmp

L4:

eax,count[1] eax,1 count[1],eax L7

mov add mov jmp

L6:

eax,count[0] eax,1 count[0],eax L7

mov add mov jmp

L5:

; '2'

; '0'

; '1'

; '3'

eax,byte\_ch eax,51

L2

eax,51 L3

eax,50 L4

eax,50 L3

eax,48 L5

eax,49 L6

L3

...

movsx cmp je cmp jg cmp je cmp jg cmp je cmp je jmp

* + pri väčšom počte vetiev použitie if neefektívne, možnosť zanesenia chyby [1]

#### Konštrukcia switch

1. kód v jazyku C b) po preklade (GCC: (GNU) 9.1.0)

switch(ch)

{

case '0':

count[0]++; break;

case '1':

count[1]++; break;

case '2':

count[2]++; break;

case '3':

count[3]++; break;

default:

count[4]++;

}

* + inštrukcia movsx – znamienkové rozšírenie

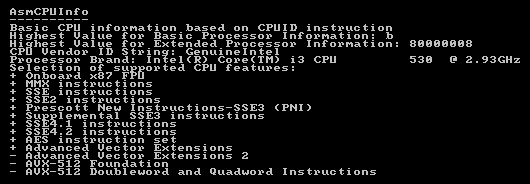
# Ďalšie užitočné inštrukcie (miscellaneous instructions)

#### Inštrukcia LEA (Load effective address) [7]

lea r,m (r – 16/32-bit register, m – pamäť)

* + vypočíta efektívnu adresu druhého operandu (m) a uloží ju do prvého (r)
  + druhý (zdrojový) operand je pamäťová referencia (offset) špecifikovaná niektorým z režimov adresovania

#### Inštrukcia CPUID (Processor identification) [8]

cpuid

* + umožňuje softvéru zistiť detaily o procesore
  + vráti informácie v registroch EAX, EBX, ECX, EDX
  + parameter v registri EAX (niekedy aj ECX) určuje kategóriu informácií, ktoré inštrukcia vráti

#### Študijná literatúra:

1. Dandamudi, S.P.: Introduction to Assembly Language Programming, Springer Science+Business Media, Inc., 2005.
2. Carter, A.P.: [PC Assembly Language](https://pacman128.github.io/static/pcasm-book.pdf), 2019.
3. Rafiquzzaman, M.: Microprocessor Theory and Applications with 68000/68020 and Pentium, [Appendix F](https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470391396.app6), John Wiley & Sons, Inc., 2008.
4. Cloutier, F.: [x86 and amd64 instruction reference](https://www.felixcloutier.com/x86/), [JMP — Jump](https://www.felixcloutier.com/x86/jmp), 2022.
5. Cloutier, F.: [x86 and amd64 instruction reference](https://www.felixcloutier.com/x86/), [Jcc — Jump if Condition Is Met](https://www.felixcloutier.com/x86/jcc), 2022.
6. Cloutier, F.: [x86 and amd64 instruction reference](https://www.felixcloutier.com/x86/), [SETcc — Set Byte on Condition](https://www.felixcloutier.com/x86/setcc), 2022.
7. Cloutier, F.: [x86 and amd64 instruction reference](https://www.felixcloutier.com/x86/), [LEA — Load Effective Address](https://www.felixcloutier.com/x86/lea), 2022.
8. Cloutier, F.: [x86 and amd64 instruction reference](https://www.felixcloutier.com/x86/), [CPUID — CPU Identification](https://www.felixcloutier.com/x86/cpuid), 2022.
9. Fog, A.: [Instruction tables](https://www.agner.org/optimize/instruction_tables.pdf), Technical University of Denmark, 1996 – 2022.

# Logické a bitové operácie

* + inštrukcie pre realizáciu logických operácií, posuny a rotácie [1], [2]
  + logické výrazy a bitové manipulácie
  + reťazce

### Logické operácie

* + bit ako jednotka informácie 0/1 (True/False)
  + inštrukcie (and, or, not, xor, test) pre realizáciu logických operácií
  + binárne a unárna (not) operácie – nad 8,16,32-bit operandmi
  + ovplyvňujú stavové príznaky (okrem CF, OF – 0, AF – nedefinovaný)
  + inštrukciám sme sa už venovali, poukážeme na ich typické použitie

#### inštrukcia and

* + podpora zložených log. výrazov a bitová operácia log. súčinu HL jazykov (neskôr)
  + nulovanie bitov
  + izolácia bitov

##### Nulovanie bitov

* + bitové masky (ak bit masky = 0 → výstup: 0, ak bit masky = 1 → výstup: kópia druhého vstupného bitu)

AL = 11010110 (operand) BL = 11111100 (maska)

and AL,BL = 11010100

*Príklad*: konverzia ASCII znakov na čísla [1]

* + relácia medzi ASCII kódom a binárnou reprezentáciou čísel 0 – 9

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Decimal digit** | **8-bit** | **binary** | **ASCII** | **code** |
| 0 | 0000 | 0000 | 0011 | 0000 |
| 1 | 0000 | 0001 | 0011 | 0001 |
| 2 | 0000 | 0010 | 0011 | 0010 |
| 3 | 0000 | 0011 | 0011 | 0011 |
| 4 | 0000 | 0100 | 0011 | 0100 |
| 5 | 0000 | 0101 | 0011 | 0101 |
| 6 | 0000 | 0110 | 0011 | 0110 |
| 7 | 0000 | 0111 | 0011 | 0111 |
| 8 | 0000 | 1000 | 0011 | 1000 |
| 9 | 0000 | 1001 | 0011 | 1001 |

* + maskovanie horných 4 bitov (AL): and AL,0FH

##### Izolácia bitov

* + odmaskovanie všetkých ostatných bitov

*Príklad*: párne/nepárne číslo? [1]

* + test (LSb = 1 – nepárne)

and AL,1 ; mask jz even\_num

odd\_num:

...

<code for odd number>

...

even\_num:

...

<code for even number>

...

#### inštrukcia or

* + podpora zložených log. výrazov a bitová operácia log. súčtu HL jazykov (neskôr)
  + nastavenie bitov (ak bit masky = 0 → výstup: kópia druhého vstupného bitu, ak bit masky = 1 → výstup: 1)

AL = 11010110 (operand) BL = 00000011 (maska)

or AL,BL = 11010111

*Príklad*: konverzia čísel (8-bit bez znamienka, 0-9) na ASCII znaky

* + nastavenie bitov b4, b5 bez zmeny ostatných (maska 00110000)

or AL,30H

##### Vystrihnutie (cut) a vkladanie (paste) bitov

* + nový bajt v AL – kombinácia nepárnych bitov z AL a párnych z BL

|  |  |  |
| --- | --- | --- |
| and | AL,55H | ; nepárne bity z AL |
| and | BL,0AAH | ; párne bity z BL |
| or | AL,BL | ; spojenie |

#### inštrukcia xor

* + podpora zložených log. výrazov HL jazykov (neskôr)
  + preklopenie (toggle) hodnoty bitu/bitov
  + inicializácia registrov (0)

##### Preklopenie hodnoty bitov

* + maska má hodnotu 1 na pozícii, kde sa má hodnota preklopiť
  + použitie xor druhý krát – pôvodná hodnota

*Príklad*: jednoduché kódovanie údajov

* + kľúč pre kódovanie ako maska pre inštrukciu xor xor AL,26H ; kľúč 26H
  + dekódovanie – rovnaký proces nad zakódovaným údajom

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| kódovanie: | 01000010 (znak B, 42H) | dekódovanie: | 01100100 | (znak d) | |
|  | 00100110 (maska, 26H) |  | 00100110 | (maska) | |
| 01100100 (znak d, 64H) | | | 01000010 | (znak | B) |

##### Inicializácia registrov

* + mov – viac miesta v pamäti

|  |  |  |  |
| --- | --- | --- | --- |
| mov | EAX,0 | (B800000000) |  |
| xor | EAX,EAX | (31C0, ovplyvňuje | príznaky) |

#### inštrukcia not

* + podpora zložených log. výrazov HL jazykov (neskôr)
  + negácia (complement) bitov operandu
  + zmena znamienka operandu – inštrukcia neg

#### inštrukcia test

* + logický ekvivalent inštrukcie cmp (log. súčin bez zmeny cieľového operandu – nedeštruktívny and) [1]

test AL,1

jz even\_num odd\_num:

...

even\_num:

...

* + za účelom nastavenia príznakov, často nasleduje podmienený skok

### Posuny

* + dva typy posunov
    - logické (čísla bez znamienka) – shl, shr
    - aritmetické (čísla so znamienkom) – sal, sar
  + stavové príznaky
    - AF – nedefinovaný
    - ZF, PF – podľa výsledku operácie, CF – posledný bit vysunutý z operandu
    - OF – nedefinovaný pre viac-bitové posuny
      * posuny o 1 bit – OF = 1 pri zmene bitu znamienka, ináč OF = 0

## Inštrukcie logických posunov

* + bitové manipulácie

*Príklad*: iné kódovanie – výmena hornej a dolnej polovice bajtu (obnova údaja – druhá aplikácia)

|  |  |
| --- | --- |
| **Binary** | **Decimal** |
| 00011100 | 28 |
| 00111000 | 56 |
| 01110000 | 112 |
| 11100000 | 224 |
| 10101000 | 168 |
| 01010100 | 84 |
| 00101010 | 42 |
| 00010101 | 21 |

|  |  |  |
| --- | --- | --- |
| mov shl | AH,AL AL,4 | ; AL obsahuje bajt pre kódovanie  ; shl/shr – na uvoľnené pozície vstupujú 0 |
| shr | AH,4 |  |
| or | AL,AH |  |

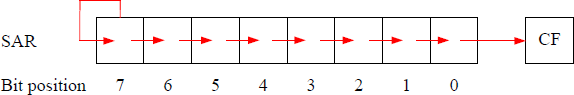
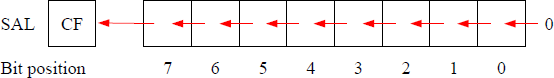
* + násobenie a delenie čísel bez znamienka mocninou 2 [1]
    - dvojnásobok/polovica čísla bez znamienka (všeobecne – mocnina 2)
    - delenie – celočíselné (prípadná desatinná časť zahodená)

*Príklad*: 28, 168

## Inštrukcie aritmetických posunov

|  |  |  |  |
| --- | --- | --- | --- |
| * syntax |  | | |
| sal | dest,count | sar | dest,count |
| sal | dest,CL | sar | dest,CL |

* + sémantika [1]



#### Zdvojnásobenie čísel so znamienkom

|  |  |
| --- | --- |
| **Signed binary** | **Decimal** |
| 00001011 | +11 |
| 00010110 | +22 |
| 00101100 | +44 |
| 01011000 | +88 |
| 11110101 | -11 |
| 11101010 | -22 |
| 11010100 | -44 |
| 10101000 | -88 |

* + posun vľavo o 1 bit (MSb – znamienko – nepredstavuje problém) [1]
    - znamienkové rozšírenie (na väčší počet bitov ako je potrebné na reprezentáciu čísla)
    - žiadny rozdiel (v operácii) v porovnaní s číslom bez znamienka
    - nie je potrebná špeciálna inštrukcia (sal je alias pre shl)

#### Delenie čísel so znamienkom na polovicu

|  |  |
| --- | --- |
| **Signed binary** | **Decimal** |
| 01011000 | +88 |
| 00101100 | +44 |
| 00010110 | +22 |
| 00001011 | +11 |
| 10101000 | -88 |
| 11010100 | -44 |
| 11101010 | -22 |
| 11110101 | -11 |

* + uvoľnený ľavý bit – potrebné nahradiť kópiou znamienkového [1]
  + potrebná špeciálna inštrukcia na tento účel – sar
  + posuny – efektívnejšie ako zodpovedajúce inštrukcie násobenia/delenia

#### Dvojité posuny

* + dve inštrukcie pre 32 a 64-bit posuny
  + syntax (count – bezprostredný alebo CL)

shld dest,src,count ; dest/src – word, doubleword shrd dest,src,count ; dest – R/M, src – R

* + rozdiel v porovnaní s posunmi – bity vysunuté zo src vstupujú do dst (src – bez modifikácie) [1]

15/31 0 15/31 0

CF

src (register)

dest (register or memory)

shld

15/31 0 15/31 0

CF

dest (register or memory)

src (register)

shrd

### Rotácie

* + posuny – vysunuté bity stratené (nie vždy žiadúce)
  + inštrukcie už ozrejmené (prednáška 2) – typické použitie

#### Rotácie bez CF (rol, ror)

* + preusporiadanie bitov v bajte, slove, dvojslove

*Príklad*: kódovanie – výmena hornej a dolnej polovice bajtu (s využitím rol/ror – jednoduchšie)

|  |  |  |  |
| --- | --- | --- | --- |
| mov | CL,4 |  | |
| ror | AL,CL | ; podobne rol | AL,CL |

#### Rotácie cez CF (rcl, rcr)

* + CF ako vstup

*Príklad*: posuny 64-bit čísel (násobenie 64-bit čísla (EDX:EAX) bez znamienka 16-timi, delenie analogicky)

a) s využitím rotácií b) s využitím dvojitých posunov

mov ECX,4 shld EDX,EAX,4 ; EAX bez zmeny shift\_left: shl EAX,4

shl EAX,1 ; MSb EAX → CF rcl EDX,1 ; CF → LSb EDX

loop shift\_left

### Logické výrazy v HL jazykoch

* + reprezentácia booleovskej hodnoty
    - stačí jediný bit, nevýhoda – potreba jeho izolácie
    - väčšina jazykov – využíva bajt (0 – false, ináč – true)
  + logické výrazy
    - napr. jazyk C – logické operátory (&& – AND, || – OR)

*Príklad*: preklad logického výrazu jazyka C (inšpirované [1])

1. kód v jazyku C b) po preklade (GCC: (tdm-1) 5.1.0)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | cmp je cmp je mov | var\_x,0 L2  var\_y,0 L2  eax,1 | ; | eax | - term | (x && y) | = | 1 |
| L2: | jmp  mov | L3  eax,0 | ; | eax | - term | (x && y) | = | 0 |
| L3: | not | eax | ; | ~(x | && y) |  |  |  |
| L4: | cmp jne cmp je  mov | var\_y,0 L4  var\_z,0 L5  edx,1 | ; | edx | - term | (y||z) = | 1 |  |
| L5: | jmp  mov | L6  edx,0 | ; | edx | - term | (y||z) = | 0 |  |
| L6: |  |  |  |  |  |  |  |  |
| xor mov | | eax,edx var\_z,eax | ; | ~(x | && y)^(y||z) | | | |

z = ~(x && y)^(y||z);

* + test premennej x (ak je 0, netestuje sa y, presun na L2)
  + výsledok (x && y) v EAX
  + návestie L3 – negácia EAX: ~(x && y)
  + test premennej y (ak y ≠ 0, netestuje sa z, presun na L4)
  + návestie L6 – operácia xor
  + uloženie výsledku do premennej z

*Implementácia operácií HL jazykov*

* + logických – tokom riadenia
  + bitových – ekvivalenty v inštrukciách

#### Bitové manipulácie

* + bitové logické operátory jazyka C: and (&), or (|), xor (^), not (~)
  + operátory bitových posunov jazyka C: vľavo (<<), vpravo (>>)

*Príklad*: preklad bitových operácií (premenná mask v registri SI) [1]

|  |  |  |
| --- | --- | --- |
| C statement | ASM code | |
| mask = mask>>2  (right-shift mask by two bit positions) | shr | SI,2 |
| mask = mask<<4  (left-shift mask by four bit positions) | shl | SI,4 |
| mask = ~mask  (complement mask) | not | SI |
| mask = mask & 85  (bitwise and) | and | SI,85 |
| mask = mask | 85  (bitwise or) | or | SI,85 |
| mask = mask ^ 85  (bitwise xor) | xor | SI,85 |

#### Vyhodnocovanie logických výrazov

* + úplné vyhodnotenie (full evaluation)
    - celý logický výraz je vyhodnotený pred priradením hodnoty výrazu (Pascal)
  + čiastočné vyhodnotenie (partial evaluation)
    - výsledok vyhodnotenia možno získať bez vyhodnotenia celého výrazu (jazyk C)
    - používané pravidlá:
      * cond1 AND cond2 (výsledok false, ak jeden zo vstupov je false)
      * cond1 OR cond2 (ak cond1 je true, netreba vyhodnotiť cond2)
    - čiastočné vyhodnotenie – efektívnejší kód

### Bitové inštrukcie

* + test a modifikácia bitu (bit test and modify)
    - 4 inštrukcie, syntax

bt operand,bit\_pos (operand: 16/32-bit R/M, bit\_pos: I/R, LSb – pozícia 0, bts, btr, btc – podobne)

* + - sémantika (kopíruje bit do CF) [1]

|  |  |
| --- | --- |
| **Inštrukcia** | **Efekt na vybraný bit** |
| bt (Bit Test) | bez efektu |
| bts (Bit Test and Set) | vybraný bit ← 1 |
| btr (Bit Test and Reset) | vybraný bit ← 0 |
| btc (Bit Test and Complement) | vybraný bit ← !(vybraný bit) |



* + prehľadávanie operandu (bit scan)
    - dve inštrukcie, smer prehľadávania (bit scan forward/reverse)
    - prehľadávanie operandu, ak sa nájde bit nastavený na hodnotu 1, vrátia pozíciu v registri

|  |  |  |  |
| --- | --- | --- | --- |
| bsf | dst\_reg,operand | (operand: 16/32-bit R/M, dst\_reg: 16/32-bit | R, pozícia bitu) |
| bsr | dst\_reg,operand |  |  |

* + - ak všetky bity operand-u nulové: ZF = 1
    - ináč: ZF = 0 a dest\_reg obsahuje pozíciu prvého nájdeného bitu nastaveného na 1
    - ostatné stavové príznaky – nedefinované

# Práca s reťazcami

### Reprezentácia reťazcov

* + reprezentácia s pevnou dĺžkou
    - kratšie reťazce – doplnené na zadanú dĺžku
    - dlhšie reťazce – skrátené
    - nevýhody reprezentácie – neefektívne využívanie pamäti (ak sa chceme vyhnúť skracovaniu)
  + reprezentácia s premenlivou dĺžkou
    - odstránenie nedostatkov predošlej reprezentácie
    - atribút reťazca udávajúci jeho dĺžku
      * dĺžka explicitne určená

|  |  |  |  |
| --- | --- | --- | --- |
| string | DB | 'Error message' |  |
| str\_len | DW | $ - string | ; $ - aktuálna pozícia v kóde (location counter) |

* + - * použitá značka konca reťazca (špeciálny znak, nevyskytuje sa v reťazci, sentinel character)
        + obyčajne 00H (ASCIIZ string), jazyk C, ďalej uvažovaná táto reprezentácia

string DB 'Error message',0

|  |  |  |
| --- | --- | --- |
| **Mnemonika** | **Význam** | **Operandy** |
| LODS | LOaD string | source |
| STOS | STOre string | destination |
| MOVS | MOVe string | source, destination |
| CMPS | CoMPare strings | source, destination |
| SCAS | SCAn string | destination |

### Reťazcové inštrukcie

* + 5 základných reťazcových inštrukcií v jazyku x86 [1]
    - operandy (source, destination) implicitné
    - explicitná špecifikácia veľkosti operandu (NASM)
    - použitie – aj iné účely (kopírovanie údajov pamäť – pamäť)

#### Operandy reťazcových inštrukcií

* + zdroj (DS:ESI), cieľ (ES:EDI), obidva
  + pre 16-bit segmenty (SI, DI)

#### Variácie

* + podpora 8,16,32-bit údajov; automatická aktualizácia (inc/dec) použitých indexových registrov (1,2,4)
  + prefix – podpora opakovaného vykonávania operácie (repetition prefix)
  + smer spracovania – vpred/vzad (direction flag, DF)

#### Prefixy reťazcových inštrukcií

* + nepodmienené/podmienené opakovanie [1]
  + neovplyvňujú stavové príznaky
  + prefix **rep** (nepodmienené opakovanie, podľa hodnoty v ECX/CX)
    - najprv test ECX na 0 (rozdiel v porovnaní s loop)

**while**(ECX ≠ 0)

execute the string instruction; ECX:=ECX-1;

**end while**

* + prefix **repe/repz** (okrem ECX relevantný aj ZF)

|  |  |  |
| --- | --- | --- |
|  | **Prefix** | **Význam** |
| nepodmienené opakovanie | rep | REPeat |
| podmienené opakovanie | repe/repz | REPeat while Equal REPeat while Zero |
| repne/repnz | REPeat while Not Equal REPeat while Not Zero |

**while**(ECX ≠ 0)

execute the string instruction; ECX:=ECX-1;

**if**(ZF = 0)

**then**

exit loop

**end if end while**

* + prefix **repne*/*repnz** (ECX a ZF)

**while**(ECX ≠ 0)

execute the string instruction; ECX:=ECX-1;

**if**(ZF = 1)

**then**

exit loop

**end if end while**

#### Smer spracovania reťazcov

* + podľa hodnoty DF (DF = 0 – vpred (auto-increment), DF = 1 – vzad)
  + manipulácia DF (2 inštrukcie bez operandov, 1B)

std (set DF) cld (clear DF)

* + často smer spracovania nie je podstatný (v niektorých prípadoch však áno)
    - napr. posun reťazca o jednu pozíciu vpravo (začíname od konca: abc0 → aabc0, od začiatku: abc0 → aaaa0)

#### Presuny reťazcov (movs, lods, stos)

* + 3 formy pre každú z inštrukcií, syntax:

movsb, movsw, movsd (podobne lods, stos)

* + prípona b, w, d – explicitná špecifikácia veľkosti operandu (aj pre iné reťazcové inštrukcie)
  + movs – kopírovanie hodnoty (b, w, d) zdrojového reťazca do cieľového
  + lods – kopírovanie hodnoty zdrojového reťazca (DS:ESI) do AL (lodsb), AX (lodsw), alebo EAX (lodsd)
  + stos – kopírovanie hodnoty v AL, AX, alebo EAX do cieľového reťazca (ES:EDI)
  + sémantika inštrukcií movs, lods, stos [1]:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **movsb (move a byte string)** | **lodsb (load** | **a** | **byte string)** | **stosb (store** | **a** | **byte string)** |
| ES:EDI:=(DS:ESI) ; byte  if(DF = 0) ; forward then  ESI:=ESI+1 EDI:=EDI+1  else ; backward ESI:=ESI-1  EDI:=EDI-1  end if | AL:=(DS:ESI) if(DF = 0)  then  ESI:=ESI+1  else  ESI:=ESI-1  end if | ; byte  ; forward  ; backward | | (ES:EDI):=AL if(DF = 0)  then  EDI:=EDI+1  else  EDI:=EDI-1  end if | ; byte  ; forward  ; backward | |
| Ovplyvnené príznaky: žiadne | Ovplyvnené príznaky: žiadne | | | Ovplyvnené príznaky: žiadne | | |

#### Porovnanie reťazcov (cmps)

* + porovná bajty (slová, dvoj-slová) na DS:ESI a ES:EDI a nastaví príznaky (ako cmp)
  + aktualizuje hodnoty ESI, EDI (podľa hodnoty DF a veľkosti operandov)

segment .data

string1 db "abcdfghi", 0 strLen EQU $ - string1 string2 db "abcdefgh", 0

segment .text

...

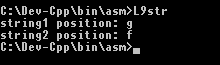
mov AX,DS mov ES,AX

mov ECX,strLen mov ESI,string1 mov EDI,string2 cld

repe cmpsb

*Príklad*: porovnanie reťazcov [1]

* + zanechá ESI ukazujúce na 'g' v string1
  + EDI ukazujúce na 'f' v string2
  + po vykonaní dec ESI, dec EDI, ukazujú tieto na prvý výskyt odlišných znakov
  + možnosť použiť podmienené skoky ...



#### Prehľadávanie reťazcov (scas)

* + vyhľadanie určitej hodnoty v reťazci
  + hodnota v AL (scasb), AX (scasw), alebo EAX (scasd), ES:EDI – prehľadávaný reťazec
  + porovná hodnotu v AL (AX, EAX) s údajom na ES:EDI a nastaví príznaky (ako cmp)
  + aktualizuje EDI (podľa DF)
  + možno použiť prefixy repe/repz, repne/repnz

#### Inštrukcie lds, les

* + syntax

lds reg,src (reg – 32b GPR, src – smerník na 48-bit operand v pamäti) les reg,src

segment .data

string1 db "abcdfghi",0 strLen EQU $ - string1 string1p dd string1

dw 0

string2 db "abcdefgh",0 string2p dd string2

dw 0

segment .text

...

mov [string1p+4],DS mov [string2p+4],DS mov ECX,strLen

lds ESI,[string1p] les EDI,[string2p] cld

repe cmpsb

* + sémantika (32-bit hodnota kopírovaná do reg, nasledujúca 16-bit hodnota do segmentového registra, bez ovplyvnenia príznakov)

lds: reg ← [src], DS ← [src+4] les: reg ← [src], ES ← [src+4]

* + inštrukcie možno využiť pri nastavení registrov (napr. ES:EDI) používaných pri reťazcových operáciach (les EDI, [string2p])

#### Výhody využitia reťazcových inštrukcií

* + automatická aktualizácia indexových registrov
  + schopnosť pracovať s dvoma operandmi v pamäti
  + v porovnaní s použitím mov a pomocného registra - elegantné a efektívne riešenie

#### Študijná literatúra:

1. Dandamudi, S.P.: Introduction to Assembly Language Programming, Springer Science+Business Media, Inc., 2005.
2. Carter, A.P.: [PC Assembly Language](https://pacman128.github.io/static/pcasm-book.pdf), 2019.

# Prepojenie asembleru s HL jazykmi

* + motivácia, programovanie v zmiešanom režime (mixed-mode)
  + volanie asm-procedúr z C, volanie C-funkcií z asembleru
  + spôsob výmeny údajov medzi volaným a volajúcim kódom (calling conventions – odovzdávanie parametrov, návratové hodnoty, ...)

### Prečo kombinácia asembleru a HL jazyka?

* + programovanie v zmiešanom režime – časť programu v C, iná časť v asembleri
  + demonštrácia princípov s využitím gcc a NASM (možno rozšíriť na iné prekladače)
  + pripomenieme nevýhody asm-programovania (nízka produktivita, náročná údržba, nízka prenositeľnosť)
  + a jeho výhody (prístup k hardvéru, k inštrukciám nedostupným vo vyššom jazyku, efektívnosť, ...)
  + dôsledok – niektoré programy napísané v zmiešanom režime (napr. systémový softvér)

### Základy programovania v zmiešanom režime

* + dva spôsoby
    - in-line asembler (program v C obsahuje aj asm-inštrukcie; informácia pre prekladač – asm; vhodné pri malom rozsahu asm v C)
    - oddelené asm-moduly (preferované pri väčšom rozsahu asm-kódu)
  + oddelené asm-moduly (začneme predstavením tohto spôsobu, in-line neskôr)
    - preklad modulov do objektového tvaru
    - spoločné linkovanie objektových súborov

*Príklad*: súbory sample1.c, sample2.asm

nasm -f win32 sample2.asm → sample2.obj gcc -o sample.exe sample1.c sample2.obj → sample.exe

### Volanie asm-procedúr z C-programu

* + spôsob komunikácie medzi C a asm programom (parametre – zásobník, návratová hodnota – registre)

#### Odovzdávanie parametrov

* + odovzdávanie parametrov (v akom poradí sú parametre ukladané na zásobník)
    - zľava doprava (left-pusher languages, väčšina HL jazykov)
    - sprava doľava (right-pusher, napr. jazyk C – predmet nášho záujmu)

*Príklad*: preklad volania procedúry (test1) [1] Detaily gcc optimalizácie v [4]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| volanie procedúry v | C: | preklad (gcc hll\_test.obj hll\_ex1c.c -m32 -S -masm=intel -Og): | | |
| int main(void) | | \_main: | push mov and sub  ...  mov mov mov call  ... | ebp ebp,esp esp,-16 esp,16  DWORD PTR [esp+8],5 DWORD PTR [esp+4],70 DWORD PTR [esp],25  \_test1 |
| { | |  |
| int | x = 25, y = 70; |  |
| int | value; |  |
| extern int test1(int, int, int); | |  |
| value = test1(x, y, 5); | |  |
| ... | |  |
| return 0; | |  |
| } | |  |

* + - pri volaní test1 – argumenty uložené sprava (5, y(70), x(25))
  + odstránenie parametrov C programom (detailnejšie neskôr)
    - konvencia jazyka C (cdecl) – volajúca funkcia uvoľní miesto v zásobníku použité na odovzdanie argumentov
    - C povoľuje premenlivý počet argumentov funkcií

#### Návratové hodnoty

* + využívaný register EAX (pre 8, 16, 32-bit celočíselné hodnoty), EDX:EAX (64-bit)
  + plávajúca čiarka (float, double) – register ST0 (FPU)

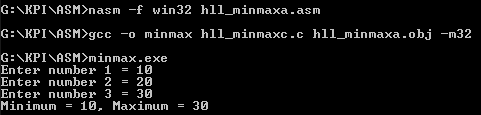
#### Uchovanie obsahu registrov

* + EBX, ESI, EDI, EBP a segmentové registre – mali by ostať zachované (prvé 3 obyčajne používané pre *registrové* premenné)
  + v prípade ich zmeny v rámci podprogramu – obnova pôvodných hodnôt (napr. zo zásobníka)

*Príklad*: ilustrácia odovzdávania parametrov hodnotou/referenciou (adaptované z [1])

* + asm-funkcia min\_max (nájde maximum a minimum zadaných hodnôt)
  + 2 súbory: hll\_minmaxc.c (vľavo), hll\_minmaxa.asm (vpravo)

#include <stdio.h> int main(void)



{

int value1, value2, value3; int minimum, maximum;

extern void min\_max (int, int, int, int\*, int\*);

printf("Enter number 1 = "); scanf("%d", &value1); printf("Enter number 2 = "); scanf("%d", &value2); printf("Enter number 3 = "); scanf("%d", &value3);

min\_max(value1, value2, value3, &minimum, &maximum); printf("Minimum = %d, Maximum = %d\n", minimum, maximum); return 0;

}

segment .text global \_min\_max

\_min\_max:

enter 0,0

; EAX - minimum, EDX - maximum mov EAX,[EBP+8] ; value 1 mov EDX,[EBP+12] ; value 2 cmp EAX,EDX

jl skip1

xchg EAX,EDX

skip1:

mov ECX,[EBP+16] ; value 3 cmp ECX,EAX

jl new\_min

cmp ECX,EDX

jl store\_result

mov EDX,ECX

jmp store\_result new\_min:

mov EAX,ECX store\_result:

mov EBX,[EBP+20] ; EBX=&minimum mov [EBX],EAX

mov EBX,[EBP+24] ; EBX=&maximum mov [EBX],EDX

leave ret

#### Global/Extern

* + samostatné moduly (C a asm)
    - deklarácia funkcií z iného modulu (extern)
    - procedúry využívané iným modulom (deklarácia global)
  + väčšina prekladačov jazyka C – externé návestia (funkcie, globálne premenné) s prefixom ( \_ )
    - prefix prekladač pripája automaticky
    - Linux gcc – štandardne prefix nevyužíva

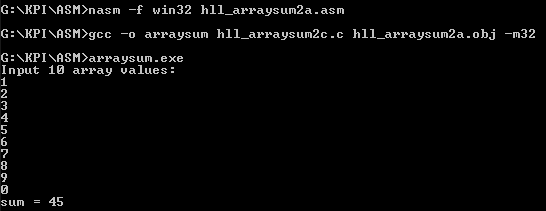
### Volanie C-funkcií z asm-programu

* + niekedy výhodné také volanie (komplikované úlohy, vstup/výstup, …)
  + odovzdávanie parametrov (konvencie používané jazykom C)

*Príklad*: výpočet súčtu prvkov poľa, využitie funkcií jazyka C printf() a scanf() [1]

* + súbory: hll\_arraysum2c.c, hll\_arraysum2a.asm
  + pred volaním týchto funkcií – ich argumenty v zásobníku (r. 14 – 16 printf(), r. 23 – 25 scanf())
  + napr. printf() – posledný uložený parameter – adresa formátovacieho reťazca (teda je možné určiť počet parametrov funkcie)
  + výpočet súčtu prvkov (r. 36 – 40)

hll\_arraysum2c.c



#include <stdio.h> #define SIZE 10

int main(void)

{

int

value[SIZE];

extern int array\_sum(int\*, int); printf("sum = %d\n",array\_sum(value,SIZE));

return 0;

}

hll\_arraysum2a.asm

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 01 | segment .data | 21 | read\_loop: |  |
| 02 | scan\_format db "%d",0 | 22 | push | ECX |
| 03 | printf\_format db "Input %d array | 23 | push | EDX |
| 04 | values:",10,13,0 | 24 | push | dword scan\_format |
| 05 |  | 25 | call | \_scanf ; scanf("%d",&value[i]) |
| 06 | segment .text | 26 | add | ESP,4 ; clear one argument |
| 07 |  | 27 | pop | EDX |
| 08 | global \_array\_sum | 28 | pop | ECX |
| 09  10 | extern \_printf,\_scanf | 29  30 | add  dec | EDX,4 ; update array pointer  ECX |
| 11  12 | \_array\_sum:  enter 0,0 | 31  32 | jnz | read\_loop |
| 13 | mov ECX,[EBP+12] ; array size | 33 | mov | EDX,[EBP+8] |
| 14 | push ECX | 34 | mov | ECX,[EBP+12] |
| 15 | push dword printf\_format | 35 | sub | EAX,EAX |
| 16 | call \_printf | 36 | add\_loop: |  |
| 17  18 | add ESP,8 | 37  38 | add  add | EAX,[EDX]  EDX,4 |
| 19 | mov EDX,[EBP+8] ; array pointer | 39 | dec | ECX |
| 20 | mov ECX,[EBP+12] | 40 | jnz | add\_loop |
|  |  | 41 | leave |  |
|  |  | 42 | ret |  |

### In-line (vložený) asembler

(offset)

stack

|  |  |
| --- | --- |
|  |  |
| +12 | SIZE |
| +8 | value ptr. |
| +4 | R.A. |
| EBP → | old EBP |
|  |  |

* + asm-príkazy vložené v C-kóde, použitie konštrukcie asm [1,7]
  + syntax používaná prekladačom gcc (AT&T) je odlišná od syntaxe v NASM (Intel)
  + uvedieme stručný prehľad hlavných syntaktických odlišností

#### AT&T syntax

* + *mená registrov* – prefix % (napr. %eax)
  + *poradie operandov* – opačné poradie (zdroj vľavo, cieľ vpravo)

mov eax,ebx je nahradené: movl %ebx,%eax

* + *rozmer operandov* – inštrukcia špecifikuje rozmer operandu (b, w, l)
    - nie je potrebná explicitná špecifikácia jeho rozmeru (byte, word, dword)

movb %bl,%al

movw %bx,%ax

movl %ebx,%eax

* + *bezprostredné operandy, konštanty* – špecifikované prefixom $

|  |  |  |  |
| --- | --- | --- | --- |
| movb | $255,%al |  | |
| movl | $total,%eax | (total – globálna C-premenná, ináč | rozšírená asm-konštrukcia) |

* + *adresovanie* – okrúhle zátvorky (namiesto [])

mov eax,[ebx] nahradené: movl (%ebx),%eax

* + - úplný 32-bit formát: imm32(base,index,scale) výpočet adresy: imm32 + base + index \* scale

*Príklad*: marks[5] → EAX, kde marks – globálne pole (int)

movl $5,%ebx

movl marks(,%ebx,4),%eax

#### Jednoduché in-line príkazy

* + asm-príkaz (inkrement EAX): asm("incl %eax");
  + skupina asm-príkazov:

asm("pushl %eax"); asm("incl %eax");

asm("popl %eax");

alebo: asm("pushl %eax; incl %eax; popl %eax");

#### Rozšírené in-line príkazy

* + možnosť prístupu k neglobálnym C-premenným, informácia pre prekladač o nami používaných registroch
  + formát príkazu asm: (4 komponenty, posledné 3 – nepovinné)

asm(asm-code

:outputs

:inputs

:clobber list);

asm-code

* + komponent obsahuje asm-príkazy (jednotlivé/skupiny)
  + inštrukcie v tejto sekcii môžu používať operandy špecifikované v ďalších dvoch komponentoch (outputs, inputs)
  + pokiaľ nemá prekladač vykonávať optimalizáciu – kľúčové slovo volatile (za asm)

outputs

* + špecifikácia výstupných operandov, formát:

"=op-constraint" (C-expression)

• = identifikuje výstup, napr. "=r" (sum), kde premenná (sum) bude mapovaná na register (r)

* + ďalšie prípustné voľby: m (memory), i (immediate), rm, ri, g (general) = rim
  + špecifikácia registra (gcc) [1, 5]:

|  |  |
| --- | --- |
| **Označenie** | **Register** |
| a | EAX |
| b | EBX |
| c | ECX |
| d | EDX |
| S | ESI |
| D | EDI |
| r | Ľubovoľný z 8 GPR (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP) |
| q | Ľubovoľný zo 4 údajových registrov (EAX, EBX, ECX, EDX) |
| A | 64-bit hodnota v EAX a EDX |
| f | FPU register |
| t | Prvý (top) FPU register |
| u | Druhý FPU register |

inputs

* + podobne ako výstupy, okrem symbolu =
  + operandom (špecifikovaným v inputs a outputs) sú pridelené poradové čísla (0,1,2 …, 9 – max. 10 operandov)
  + v kóde je možné sa na ne odvolávať (%číslo)
  + ak je operand vstupom a zároveň výstupom (prijíma výsledok) – uvedený v obidvoch zoznamoch
  + registre – prefix % (keďže AT&T syntax tu využíva tiež %, EAX označíme %%eax)

*Príklad*: sum = sum + number1 (premenná sum – vstup aj výstup súčasne, avšak v inputs uvedená už ako "0") [1]

asm("addl %1,%0"

:"=r" (sum)

:"r" (number1), "0" (sum)

);

/\* output \*/

/\* inputs \*/

clobber list

* + špecifikuje zoznam registrov modifikovaných v rámci príkazu asm (informácia pre gcc – prípadná obnova obsahov)
  + "memory" – pri zmene v pamäti, "cc" – pri zmene registra príznakov

*Príklad*: žiadny výstup, EAX modifikovaný (uvedený v zozname 'clobber list') [1]

asm("movl %0,%%eax"

:/\* no output \*/

:"r" (number1)

:"%eax"

);

/\* inputs \*/

/\* clobber list\*/

*Príklad*: procedúra test1 v in-line režime (vstupné operandy – x, y, z; modifikované – EAX, cc; výsledok (x+y-z) vrátený v EAX) [1]

|  |  |  |
| --- | --- | --- |
| #include <stdio.h> | int test1(int x, int y, int z)  {  asm("movl %0,%%eax;" "addl %1,%%eax;" "subl %2,%%eax;"  :/\* no outputs \*/  :"r"(x), "r"(y), "r"(z)  :"cc","%eax");  } | /\* inputs \*/  /\* clobber list \*/ |
| int main(void) |
| { |
| int x = 25, y = 70; |
| int value; |
| int test1(int, int, int); |
| value = test1(x, y, 5); |
| printf("Result = %d\n", value); |
| return 0; |
| } |

#### Konvencie volania funkcií [3]

* + uvedené sú štandardné C-konvencie volania funkcií
  + prekladače často podporujú aj ďalšie konvencie (CL-prepínače, rozšírenie syntaxe jazyka C)
  + prekladač GCC
    - podporuje rôzne konvencie volania funkcií [3, 6], uvedieme niektoré z nich
    - možnosť špecifikácie atribútov ( attribute ) pri deklarácii funkcie, napríklad:
      * cdecl – prekladač bude predpokladať, že *volajúca* funkcia uvoľní miesto v zásobníku použité na odovzdanie argumentov
      * stdcall – prekladač bude predpokladať, že *volaná* funkcia uvoľní miesto, pokiaľ nemá premenlivý počet argumentov
      * regparm – celočíselné argumenty budú odovzdané v *registroch* (EAX, EDX, ECX), funkcie s premenlivým počtom argumentov budú naďalej používať zásobník

int test(int, int, int) attribute ((cdecl)); int test(a, b, c)

{

return a;

}

void main(void)

{

int x = 25, y = 70, value; value = test(x, y, 5); printf("value: %d\n", value); return;

}

\_test:

push mov mov pop ret

ebp ebp,esp

eax,DWORD PTR [ebp+8] ebp

int test(int, int, int) attribute ((stdcall)); int test(a, b, c)

{

return a;

}

void main(void)

{

int x = 25, y = 70, value; value = test(x, y, 5); printf("value: %d\n", value); return;

}

\_test@12:

push mov mov pop ret

ebp ebp,esp

eax,DWORD PTR [ebp+8] ebp

12

int test(int, int, int) attribute ((regparm(3))); int test(a, b, c)

{

return a;

}

void main(void)

{

int x = 25, y = 70, value; value = test(x, y, 5); printf("value: %d\n", value); return;

}

\_main:

...

mov mov mov call

...

edx,DWORD PTR [esp+24] eax,DWORD PTR [esp+28] ecx,5

\_test

\_test:

push mov sub mov mov mov mov leave ret

ebp ebp,esp esp,12

DWORD PTR [ebp-4],eax DWORD PTR [ebp-8],edx DWORD PTR [ebp-12],ecx eax,DWORD PTR [ebp-4]

#### Študijná literatúra:

1. Dandamudi, S.P.: Introduction to Assembly Language Programming, Springer Science+Business Media, Inc., 2005.
2. Carter, A.P.: [PC Assembly Language](https://pacman128.github.io/static/pcasm-book.pdf), 2019.
3. A GNU Manual, [Declaring Attributes of Functions](http://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html), Free Software Foundation, Inc., 1988-2023.
4. A GNU Manual, [Options That Control Optimization](http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html), Free Software Foundation, Inc., 1988-2023.
5. A GNU Manual, [Constraints for Particular Machines](https://gcc.gnu.org/onlinedocs/gcc/Machine-Constraints.html), Free Software Foundation, Inc., 1988-2023.
6. A GNU Manual, [x86 Function Attributes](https://gcc.gnu.org/onlinedocs/gcc/x86-Function-Attributes.html), Free Software Foundation, Inc., 1988-2023.
7. A GNU Manual, [How to Use Inline Assembly Language in C Code](https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html), Free Software Foundation, Inc., 1988-2023.

# Prerušenia

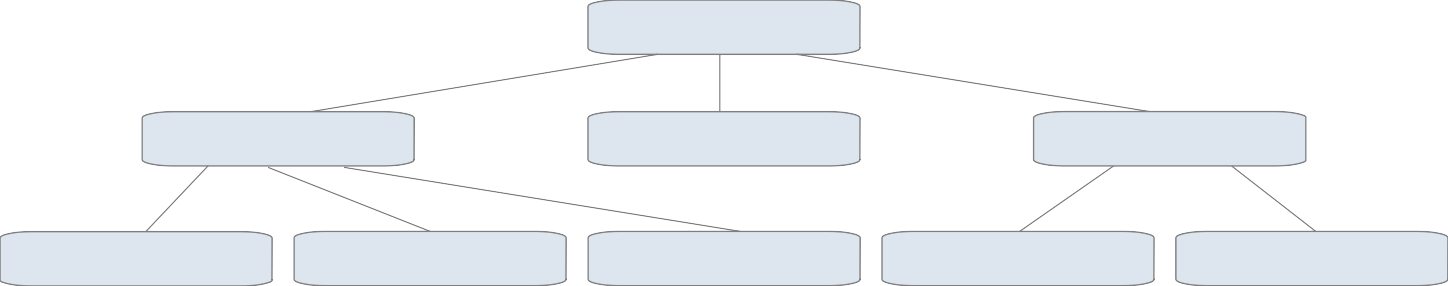
* + mechanizmy prerušení v chránenom režime (Pentium)
  + HW a SW prerušenia, výnimky, systémové volania (Linux)
  + prerušenia v reálnom režime (volania služieb systémov DOS, BIOS)

### Úvod

* + prerušenie ako mechanizmus pre zmenu sekvenčného vykonávania programu (aj skoky a procedúry), podobnosť s volaním procedúr
    - prenos riadenia na procedúru obsluhy prerušenia (ISR – Interrupt Service Routine)
    - po ukončení ISR, obnova vykonávania prerušeného programu
  + odlišnosti
    - prerušenia možno iniciovať softvérovo aj hardvérovo (externé zariadenia)
    - softvér (software interrupts) – inštrukcia int (očakávaná/plánovaná udalosť)
    - hardvér – napr. ctrl-c/ctrl-break – návrat riadenia OS (neočakávaná udalosť)
    - ďalšie odlišnosti (ISR – bežne rezidentné v pamäti, identifikácia číslom miesto mena, automaticky uložený FLAGS, …)

### Klasifikácia prerušení

* + okrem dvoch spomínaných kategórií (SW a HW – inicializované) aj výnimky (exceptions) [1]
    - ošetrenie chýb pri spracovaní inštrukcií (napr. delenie 0)
  + SW prerušenia (int) – prístup k I/O zariadeniam (systémové (system defined) / používateľské (user defined))
    - možno použiť na emuláciu výnimiek programovo (špecifikovaním príslušného vektora ako operandu)
  + HW prerušenia (generované hardvérom – získanie času procesora zariadením)
    - maskovateľné (maskable) – spracovanie možno odložiť
    - nemaskovateľné (non-maskable, NMI) – vždy akceptované/spracované procesorom (RAM parity error)



Interrupts

Exceptions

Software Interrupts

Hardware Interrupts

Aborts

Faults

Traps

Maskable

Nonmaskable

### Spracovanie prerušení v chránenom režime (protected mode)

* + identifikácia prerušenia číslom (Pentium – 256 typov prerušení)
  + číslo (typ) prerušenia (vector, 0-255) – index do tabuľky s adresami ISR
    - tabuľka IDT (Interrupt Descriptor Table), číslo prerušenia \* 8 → index do tabuľky (položka – descriptor, 8B) [1]
      * podobne ako (spomínané) tabuľky LDT a GDT
    - IDT – umiestnená v pamäti, pozícia daná obsahom registra IDTR (48b, 32b báza + 16b limit)
    - špeciálne inštrukcie lidt, sidt (load/store, 6B pamäťový operand)
    - IDT – môže obsahovať 3 typy deskriptorov (interrupt gate, trap gate, task gate)
      * task gate – ďalej nebudeme uvažovať (nesúvisí priamo s mechanizmom prerušení)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3  1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 1  6 | 1  5 | 1  4 | 1  3 | 1  2 |  |  |  | 8 | 7 |  | 5 | 4 |  |  |  | 0 |
| Offset 31:16 | | | | | | | | | | | | | | | | P | DPL | | 0 1 1 1 0 | | | | | 0 0 0 | | | Not used | | | | |
| Segment selector | | | | | | | | | | | | | | | | Offset 15:00 | | | | | | | | | | | | | | | |
| 3  1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 1  6 | 1  5 |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 0 |
| Interrupt gate | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

* + - * interrupt gate, trap gate – 16b segment selector, 32b offset, DPL, P (present)
        + segment – voľba selektora z GDT alebo LDT (bit TI)
        + offset – z interrupt gate

Organizácia IDT [1]:

ADD

47 16 15 0

Gate for interrupt N

IDT limit

IDT base address

32 bits 16 bits

IDTR

4 bytes

|  |  |
| --- | --- |
| Offset | |
| Seg. selector |  |
|  | |
| Offset | |
| Seg. selector |  |
| Offset | |
| Seg. selector |  |

Gate for interrupt 1

Gate for interrupt 0

#### Výskyt prerušenia (prípad bez zmeny privilégií)

* + EFLAGS → zásobník
  + IF = 0 a TF = 0 (zakázanie ďalších prerušení; možné opätovné povolenie v ISR (sti, cli), pokiaľ nie je dôvod ich zakazovať)
  + CS, EIP → zásobník
  + CS ← 16b segment selector (interrupt gate)
  + EIP ← 32b offset (interrupt gate)

#### Prerušenie cez Trap gate

* + podobne ako Interrupt gate (doposiaľ), ale bez modifikácie IF
  + niektoré typy výnimiek (8, 10-14, 17) ukladajú na zásobník aj chybový kód (identifikácia príčiny výnimky v rámci obsluhy)

#### Návrat z obsluhy prerušenia

stack

stack

* + podobne ako procedúry (inštrukcia iret), činnosť:

|  |  |  |
| --- | --- | --- |
|  |  | |
|  |  | |
|  | EFLAGS | |
|  |  | CS |
| ESP → | EIP | |
|  |  | |

|  |  |  |
| --- | --- | --- |
|  |  | |
|  |  | |
|  | EFLAGS | |
|  |  | CS |
|  | EIP | |
| ESP → | Error code | |

* + - EIP ← zásobník (32b)
    - CS ← zásobník (16b)
    - EFLAGS ← zásobník (32b)

#### Vektory prerušení a výnimiek v chránenom režime

* + vektory 0 – 31 rezervované; architektúrou definované výnimky a prerušenia (nie všetky nutne obsadené) [6]
  + vektory 32 – 255 používateľsky definované; všeobecne dostupné pre externé zariadenia

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Vector no.** | **Mnemonic** | **Description** | **Type** | **Error code** | **Source** |
| 0 | #DE | Divide Error | Fault | No | DIV and IDIV instructions |
| 1 | #DB | Debug Exception | Fault/Trap | No | Breakpoints, single-step, ... |
| 2 | – | NMI Interrupt | Interrupt | No | Nonmaskable external interrupt |
| 3 | #BP | Breakpoint | Trap | No | INT 3 instruction |
| 4 | #OF | Overflow | Trap | No | INTO instruction |
| 5 | #BR | BOUND Range Exceeded | Fault | No | BOUND instruction |
| … | … | … | … | … | … |
| 22-31 | – | Intel reserved. Do not use. | Interrupt |  | External interrupt or INT *n* instruction |
| 32-255 | – | User defined (non-reserved) interrupts |

### Výnimky

* + klasifikované do 3 skupín (*faults*, *traps*, *aborts*) podľa spôsobu hlásenia ich výskytu a možnosti prípadného reštartu inštrukcie
    - **fault** (porucha) – výnimočný stav, ktorý všeobecne možno napraviť v rámci obsluhy
      * v prípade úspešnej nápravy program reštartovaný bez straty kontinuity
      * obnova stavu do bodu pred vykonaním (výnimku generujúcej) inštrukcie (uchované CS:EIP ukazujúce na danú inštrukciu)
        + napr. výpadok segmentu (ISR zabezpečí zavedenie z disku, vráti riadenie programu – reštart inštrukcie)
    - **trap** (pasca) – výnimka signalizovaná po vykonaní inštrukcie (napr. overflow)
      * umožňuje pokračovanie v programe bez straty kontinuity; bez reštartu inštrukcie
      * návratová adresa ukazuje bezprostredne za (výnimku generujúcu) inštrukciu
        + napr. volanie služieb systému
    - **abort** (zlyhanie) – výnimka nie vždy signalizovaná presne na mieste generujúcej inštrukcie
      * neumožňuje pokračovanie programu, ktorý zlyhanie spôsobil
        + závažné chyby (chyby HW, nekorektné hodnoty v systémových tabuľkách)
        + úlohou obsluhy je zber diagnostických informácií a ukončenie aplikácie čo najelegantnejšie
  + existencia vyhradených prerušení (dedicated interrupts), napr. prvých 5 prerušení:
    - Divide error (0, #DE) – div/idiv (podiel väčší ako špecifikovaný cieľ)
    - Single-step (1, #DB) – krokovanie programu (ladiace nástroje), ak TF = 1, CPU generuje toto prerušenie po vykonaní inštrukcie
    - Nonmaskable interrupt (NMI, 2) – neskôr
    - Breakpoint (3, #BP) – spracovanie bodov prerušenia, inštrukcia int 3 (CCh, 1B)
    - Overflow (4, #OF) – dva spôsoby generovania (int 4, into), bežne pretečenie ošetrené podmieneným skokom (jo, jno)

### Softvérové prerušenia

* + iniciované vykonaním inštrukcie prerušenia

int int-type (int-type: 0-255)

* + jedná sa o typy prerušení
  + parametrizácia (napr. systémové volania Linuxu int 0x80, cca.180 volaní (v závislosti od verzie), číslo volania v EAX)

## Systémové volania – Linux

* + Linux využíva prvých 32 vektorov (0-31) pre výnimky a NMI
  + ďalších 16 (32-47) – HW prerušenia (IRQ)
  + jeden vektor (128/0x80) – SW prerušenia (služby systému)

#### Súborové I/O operácie (file I/O)

* + Linux (Unix) – intenzívne využívanie súborov (napr. klávesnica, displej)
  + vstup a výstup z pohľadu systému – prúd bajtov (stream)
  + štandardne definované 3 súbory – stdin (klávesnica), stdout a stderr (displej)

#### Deskriptor súboru (file descriptor)

* + každý otvorený súbor – 16b celé číslo (file id), prístup k súboru
  + návratová hodnota volaní *file open*, *file create*
  + identifikátory s najnižšími hodnotami: stdin (0), stdout (1), stderr (2)

#### Ukazovateľ pozície v súbore (file pointer)

* + asociovaný s každým otvoreným súborom
  + posun (offset) v bajtoch od začiatku súboru (pozícia v súbore pre operácie read/write), pri otvorení súboru nulový
  + sekvenčný (aktualizácia ukazovateľa vzhľadom na prečítané/zapísané bajty) vs. priamy prístup (manipulácia ukazovateľa) k súboru

#### Systémové volania pre prácu so súbormi (file system calls)

* + začiatok práce so súborom – vytvorenie (call 8)/otvorenie (call 5) súboru
  + po otvorení (vytvorení) dostupné ďalšie operácie – čítanie (call 3), zápis (call 4)
  + priamy prístup (direct access) k údajom – manipulácia ukazovateľa (call 19)
  + po ukončení práce – uzatvorenie súboru (call 6)

*System call 8 (Create and open a file)*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| R | W | X | R | W | X | R | W | X |
| User | | | Group | | | Other | | |

Vstup: EAX = 8, EBX = file name (smerník), ECX = file permissions [1] Výstup: EAX = file descriptor (kladný)

Chyba: EAX = error code (záporný)

*System call 5 (Open a file)*

Vstup: EAX = 5, EBX = file name (smerník), ECX = file access mode (R-only (0), W-only (1), R-W (2), ...), EDX = file permissions Výstup: EAX = file descriptor

Chyba: EAX = error code

*System call 3 (Read from a file)*

Vstup: EAX = 3, EBX = file descriptor, ECX = input buffer (smerník), EDX = buffer size (bytes) Výstup: EAX = number of bytes read

Chyba: EAX = error code

* + po ukončení čítania, ukazovateľ aktualizovaný (bajt za posledným prečítaným bajtom), ďalšie volanie – sekvenčný prístup
  + ak EAX < EDX – dosiahnutý koniec súboru

*System call 4 (Write to a file)*

Vstup: EAX = 4, EBX = file descriptor, ECX = output buffer (smerník), EDX = buffer size (bytes) Výstup: EAX = number of bytes written

Chyba: EAX = error code

* + ak EAX < EDX – chyba (napr. disk full)

*System call 6 (Close a file)*

Vstup: EAX = 6, EBX = file descriptor Výstup: EAX = –

Chyba: EAX = error code

*System call 19 (lseek – update file pointer)*

Vstup: EAX = 19, EBX = file descriptor, ECX = offset, EDX = whence Výstup: EAX = offset from the beginning of file

Chyba: EAX = error code

* + nesekvenčný (priamy, náhodný) prístup k údajom v súbore
  + offset (ECX) sa pripočíta k pozícii udanej EDX (0 – začiatok súboru, 1 – aktuálna pozícia, 2 – koniec súboru)
  + ďalšie volania a ich opis napr. v [3, 4]

*Príklad*: načítanie reťazca z klávesnice (zachovanie príznakov pushf/popf), pripojenie NULL na konci (ASCIIZ) [1]

; getstr receives input buffer pointer in EDI, buffer size in ESI

getstr:

pusha pushf mov mov mov mov int dec mov popf popa ret

EAX,3 EBX,0 ECX,EDI EDX,ESI 0x80 EAX

; read file

; 0 = stdin

byte[EDI+EAX],0 ; NULL character appended

### Hardvérové prerušenia

* + SW prerušenia – synchrónne udalosti (int v programe)
  + HW prerušenia – asynchrónne (pôvodcom je hardvér – I/O zariadenia)
    - nemaskovateľné (NMI) – pripojené priamo na NMI vývod CPU (vždy na ne reaguje, prerušenie typu 2)
    - maskovateľné – väčšina HW prerušení (signál na vývod (pin) INTR – INTerrupt Request) – obslúžené, ak IF = 1
      * IF (Interrupt Enable), inštrukcie sti, cli

#### Typ prerušenia

* + aký je typ (číslo) prerušenia iniciovaného hardvérom?
    - ako odpoveď na signál INTR – procesor vyšle signál INTA (interrupt acknowledge)
    - zariadenie umiestni vektor prerušenia na údajovú zbernicu

#### Spracovanie prerušení od viacerých zariadení

* + vývod INTR – ako pripojiť viac zariadení?
  + mechanizmus určovania priorít (pri súčasnom príchode)
    - jednu požiadavku prepustiť ďalej, ostatné podržať
    - mechanizmus implementovaný špeciálnym čipom – Intel 8259 (Programmable Interrupt Controller, PIC) [1], [8]
      * možnosť priamo obslúžiť až 8 zariadení (cez linky IRQ0 – IRQ7)
      * dvojica (8-bit) registrov pre konfiguráciu: ICR (interrupt command register) a IMR (interrupt mask register)
      * IMR – povolenie/zakázanie jednotlivých požiadaviek
      * ICR – prideľovanie priorít požiadavkám (bežná inicializácia BIOS-om: IRQ0 – najvyššia, IRQ7 – najnižšia priorita)
      * v rámci inicializácie priradené tiež typy (vektory) prerušení (špecifikovaný len najnižší, ostatné priradené automaticky)
        + ak 8259 prijme signál INTA, vyšle tento údaj (vektor) na údajovú zbernicu
      * komunikácia CPU a 8259 cez údajovú zbernicu; registre ICR a IMR mapované v I/O adresnom priestore (20H a 21H)
      * prerušenia od 8259 akceptované ak IF = 1; selektívne povolenie/zakázanie prerušení – pomocou IMR (ak bit = 1 – zakázať)
      * viacero požiadaviek súčasne – serializácia podľa priority; ukončenie obsluhy signalizované zápisom hodnoty 20H do ICR
  + v moderných PC nahrádzaný novším systémom APIC (Advanced Programmable Interrupt Controller) [9]

mov out

AL,20H 20H,AL

INTR INTA

IRQ0 IRQ1 IRQ2 IRQ3 IRQ4 IRQ5 IRQ6 IRQ7

### Prerušenia v reálnom režime

8259

PIC

CPU

8-bit data bus

* + MS DOS, BIOS poskytujú niekoľko služieb prostredníctvom prerušení
  + Pentium používa v tomto prípade mechanizmy prerušení procesora 8086
    - IDT začína na adrese 0
    - každý vektor má veľkosť 4B (typ prerušenia \* 4 pre získanie správnej položky)
    - vektor obsahuje CS:IP smerník na ISR

#### Pri výskyte prerušenia Návrat z prerušenia (iret)

FLAGS → zásobník IP ← zásobník

IF = 0, TF = 0 CS ← zásobník

CS, IP → zásobník FLAGS ← zásobník

CS ← 16b (typ prerušenia \* 4 + 2) IP ← 16b (typ prerušenia \* 4)

## Softvérové prerušenia

* + služby systému DOS (int 21h), vyše 80 funkcií
  + DOS a BIOS obsahujú aj služby pre prístup k I/O zariadeniam
  + spôsoby interakcie s I/O zariadeniami (DOS, BIOS, priame riadenie) [1]:

DOS Support

BIOS Support

Input/Output Devices

Application Program

address 003FF

003FE

003FD

003FC

… 0000B

0000A

00009

00008

00007

00006

00005

00004

00003

00002

00001

00000

CS

int type 255

IP

...

CS

int type 2

IP

CS

int type 1

IP

CS

int type 0

IP

#### Práca s klávesnicou

|  |
| --- |
| CS high byte |
| CS low byte |
| IP high byte |
| IP low byte |
| ... |
| CS high byte |
| CS low byte |
| IP high byte |
| IP low byte |
| CS high byte |
| CS low byte |
| IP high byte |
| IP low byte |
| CS high byte |
| CS low byte |
| IP high byte |
| IP low byte |

* + všeobecne – radič zariadenia (I/O controller, HW rozhranie) a ovládač zariadenia (device driver, SW rozhranie)
    - klávesnica – špeciálny obvod, pri stlačení/uvoľnení klávesu generované prerušenie (int 9)
    - prerušenie obslúžené BIOS-om (prijíma tzv. scan-kód klávesy, generuje ASCII kód, uloží do frontu (FIFO buffer) klávesnice)

*Základné služby systému DOS pre prácu s klávesnicou*:

*Funkcia 01H (Keyboard input with echo)*

Vstup: AH = 01H Výstup: AL = ASCII code

* + - ak je zadaný ctrl-break, vyvolané je prerušenie 23H

*Funkcia 08H (Keyboard input without echo)*

Vstup: AH = 08H Výstup: AL = ASCII code

*Funkcia 0AH (Buffered keyboard input)*

Vstup: AH = 0AH

DS:DX = pointer to the input buffer (1.byte should have the buffer size) Výstup: string in the buffer

* + - zadaný reťazec umiestnený od 3. bajtu, číta sa po Enter, alebo zaplnenie buffra
    - po ukončení čítania – počet prečítaných znakov v 2.bajte buffra
  + špeciálne klávesy (extended keyboard keys)
    - nie sú súčasťou ASCII (F1, F2, ... Kurzorové šípky, Home, End, …)
    - pri výskyte takého klávesu – 2B vo fronte (00H a príslušný scan-kód) – potrebné dve volania funkcie ...

*Základné služby systému BIOS pre prácu s klávesnicou*:

* + dostupné pomocou int 16H

*Funkcia 00H (Read a character from the keyboard)*

Vstup: AH = 00H

Výstup: if AL ≠ 0 then AL = ASCII code, AH = scan-code if AL = 0 then AH = scan-code

*Funkcia 01H (Check keyboard buffer)*

Vstup: AH = 01H

Výstup: ZF = 1 if keyboard buffer is empty

ZF = 0 if nonempty (returns ASCII in AL and scan-code in AH, does not remove them from buffer)

*Funkcia 02H (Check keyboard status)*

Vstup: AH = 02H

Výstup: AL = status of shift and toggle keys (0 – RShift, 1 – LShift, 2 – Control, ...)

* + ďalšie služby systémov DOS, BIOS – napr. [5]

#### Výstup textu na obrazovku

* + DOS aj BIOS poskytujú niekoľko funkcií pre zobrazovanie znakov (uvádzame služby DOS-u)

*Funkcia 02H (Display a character on the screen)*

Vstup: AH = 02H

DL = ASCII code to be displayed Výstup: nothing

*Funkcia 09H (Display a string of characters)*

Vstup: AH = 09H

DS:DX = pointer to a character string ($ - terminated) Výstup: nothing

*Príklad*: verzia programu pre výpis HEX kódu zadaného znaku s využitím volaní systému DOS

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| segment .data  prompt db "Zadaj znak: ",'$',0  msg1 db 10,13,"ASCII kod znaku '",'$',0 msg2 db "' v HEX je ",'$',0  segment .stack stack | | | | mov mov int  mov mov | | ah,2 ; vypis nacitaneho znaku dl,cl  21h  ah,9 dx,msg2 |
| resb 64  segment .text  ..start:  mov ax,data  mov ds,ax  mov ax,stack | | |  |  | int  mov shr call mov call | 21h  dl,cl ; prevod do HEX dl,4  HEX\_znak dl,cl HEX\_znak |
| mov | ss,ax |  | |  | |  |
| mov mov int  mov int mov  mov mov int | ah,9 dx,prompt 21h  ah,1 21h cl,al  ah,9 dx,msg1 21h | ; vypis retazca  ; nacitaj znak  ; vypis retazca | | mov int  HEX\_znak:  and cmp jle add  addzero:  add mov int ret | | ax,4C00h ; ukoncenie programu 21h  dl,0fh ; spodne 4 bity DL dl,9  addzero dl,'A'-10-'0'  dl,'0' ah,2 21h |

* + využívané služby systému DOS (01h – vstup znaku, 02h – výpis znaku, 09h – výpis reťazca, 4Ch – ukončenie programu)
  + preklad programu: nasm -f obj dprvy.asm, linkovanie: alink -oEXE dprvy.obj
  + spustenie: na 64-bit systémoch v prostredí DOSBox [7]

*Priame riadenie I/O zariadení*

* + neexistujúca podpora cez DOS/BIOS
  + neštandardný prístup k zariadeniu
  + Pentium – špeciálny I/O adresný priestor (64K 8-bit portov, 32K 16-bit portov, alebo 16K 32-bit portov, resp. ich kombinácie)
  + prístup k I/O portom
    - registrové I/O inštrukcie (prenosy register-port, 8/16/32b – podľa voľby acc)

|  |  |  |
| --- | --- | --- |
| in | acc,port8 | (acc – AL, AX, EAX, port8 – priamo adresovaný port 0 – FFH) |
| in out  out | acc,DX port8,acc  DX,acc | (DX – adresa portu, 0 – FFFFH) |

* + - blokové I/O inštrukcie (pamäť-porty, podobnosť s reťazcovými inštrukciami)
      * bez operandov, možno použiť prefix rep (nie však repe, repne), DF – ako u reťazcových inštrukcií
      * DX a indexové registre automaticky aktualizované

ins (insb, insw, insd – adresa portu v DX, pamäť: ES:EDI)

outs (outsb, outsw, outsd – adresa portu v DX, pamäť: DS:ESI)

#### Študijná literatúra:

1. Dandamudi, S.P.: Introduction to Assembly Language Programming, Springer Science+Business Media, Inc., 2005.
2. Carter, A.P.: [PC Assembly Language](https://pacman128.github.io/static/pcasm-book.pdf), 2019.
3. Boldyshev, K.: [List of Linux/i386 system calls](http://asm.sourceforge.net/syscall.html), 1999-2000.
4. Linux Syscall Reference (32 bit), <https://syscalls32.paolostivanin.com/>
5. Jurgens, D.: HelpPC Reference Library, [Interrupt Services DOS/BIOS/EMS/Mouse](https://www.stanislavs.org/helppc/idx_interrupt.html)
6. [Intel® 64 and IA-32 Architectures Software Developer’s Manual](https://software.intel.com/en-us/articles/intel-sdm), Volume 3A: System Programming Guide, Chapter 6, 2023.
7. DOSBox, <https://www.dosbox.com/>, 2021.
8. OSDev.org, [8259 PIC](https://wiki.osdev.org/8259_PIC), 2023.
9. Wikipedia, [Advanced Programmable Interrupt Controller](https://en.wikipedia.org/wiki/Advanced_Programmable_Interrupt_Controller), 2022.

# Aritmetika s pohyblivou rádovou čiarkou

* + reprezentácia čísel s pohyblivou rádovou čiarkou
  + organizácia jednotky pre prácu s pohyblivou rádovou čiarkou (FPU)
  + inštrukcie pre prácu s číslami s pohyblivou rádovou čiarkou

### Reprezentácia čísel

* + doposiaľ na reprezentáciu hodnôt využívané celé čísla
  + čísla s pohyblivou rádovou čiarkou pre reprezentáciu reálnych čísel (*float*, *double* v jazyku C)
  + problém zaokrúhľovania (neprítomný v aritmetike s celými číslami)
  + reprezentácia s pevnou rádovou čiarkou (fixed-point representation)
    - jednoduchá (N = I + F, I – celá časť, F – desatinná časť, N – celkový počet bitov)

aI-1....a1a0 . a-1a-2 … a-F

* + - presnosť (F) vs. rozsah (I)
    - vážne nedostatky (obmedzený rozsah, resp. vysoké priestorové požiadavky)
  + reprezentácia s využitím exponenciálnej notácie (znamienko, mantisa, exponent)
    - lepšie využitie daného počtu bitov pre reprezentáciu reálneho čísla

veľkosť = mantisa \* 2exponent

* + - normálna forma (binárna reprezentácia), Xi, Yj (1 ≤ i ≤ M, 0 ≤ j < N) – bity mantisy resp. exponentu

±1. *X* 1 *X* 2 ... *XM* −1 *X M* ×2±*Y N* −1 *Y N* −2 ... *Y* 1 *Y* 0

*Príklad*: normálna forma čísla +1101.101 x 2+11010 je:

+1.101101 x 2+11101 resp. +1.101101E11101

* + implementácia na procesore Pentium v súlade so štandardom IEEE 754
    - výmena údajov medzi systémami, numerické knižnice
  + FPU procesora Pentium podporuje 3 formáty čísel [1]:
    - čísla s jednoduchou presnosťou (32-bit, float v C)
    - čísla s dvojitou presnosťou (64-bit, double v C)
    - rozšírený formát (80-bit, interný)

1 bit 8 bits 23 bits

|  |  |  |
| --- | --- | --- |
| sm | exponent | mantissa |

31 30 23 22 0

1 bit 11 bits 52 bits

|  |  |  |
| --- | --- | --- |
| sm | exponent | mantissa |

63 62 52 51 0

* + poznámky k formátom čísel s jednoduchou a dvojitou presnosťou:
    - *mantisa* – uchováva iba desatinnú časť normalizovaného čísla (1 naľavo od čiarky sa neukladá (predpokladá sa stále 1))
    - *exponent* – bez znamienkového bitu, kód s posunutou nulou (pripočítanie 127 (7FH) resp. 1023 (3FFH))

##### Prevod reálneho čísla do formátu s pohyblivou rádovou čiarkou

* + ilustrácia na prevode čísla 78.8125
  + krok 1 – prevod čísla do binárneho tvaru

(78)10 = (1001110)2, (0.8125)10 = (0.1101)2, (78.8125)10 = (1001110.1101)2

* + krok 2 – normalizácia

(1001110.1101E0)2 = (1.0011101101E110)2

* + krok 3 – prevod exponentu do kódu s posunutou nulou

(110)2 + (1111111)2 = (10000101)2 takže: (78.8125)10 = (1.0011101101E10000101)2

* + krok 4 – oddelenie jednotlivých častí

znamienko: 0, mantisa: 0011101101 (1. - implicitná), exponent: 10000101

##### Špeciálne hodnoty

* + reprezentácia špeciálnych hodnôt – dohodnutou kombináciou hodnôt Z, M, E [1]

|  |  |  |  |
| --- | --- | --- | --- |
| **Hodnota** | **Znamienko** | **Exponent** | **Mantisa** |
| +0 | 0 | 0 | 0 |
| -0 | 1 | 0 | 0 |
| +∞ | 0 | FFH | 0 |
| -∞ | 1 | FFH | 0 |
| NaN | 0/1 | FFH | ≠0 |
| Denormals | 0/1 | 0 | ≠0 |

* + NaN (not-a-number) – reprezentácia hodnôt, ktoré sú nedefinované, resp. nereprezentovateľné, napr. 0/0 alebo √−1
  + Denormals – hodnoty menšie ako je možné zobraziť v normálnej forme (implicitná 1 nahradená 0)

##### Operácie s číslami s pohyblivou rádovou čiarkou

* + sčítanie (odčítanie) *Príklad*: 13.25 + 4.75 (1.10101 x 23 + 1.0011 x 22)
    - vyrovnanie exponentov (na vyšší) 1.0011 x 22 = 0.10011 x 23
    - sčítanie (odčítanie) mantís 1.10101 + 0.10011 = 10.01
    - normalizácia výsledku 1.001 x 24 (=1810)
    - test na pretečenie/podtečenie žiadne
  + násobenie *Príklad*: 15 \* 5 (1.111 x 23 \* 1.01 x 22)
    - sčítanie exponentov (pozor – kód s posunutou nulou) 3 + 2 = 5
    - násobenie mantís 1.111 x 1.01 = 10.01011
    - určenie znamienka kladné
    - normalizácia súčinu 10.01011 x 25 = 1.001011 x 26 (=7510)
    - test na pretečenie/podtečenie žiadne

### Jednotka FPU

* + zvýšenie výkonu procesora v oblasti výpočtov s pohyblivou rádovou čiarkou – pridaním špeciálneho hardvéru
    - procesor bez FPU – emulácia činnosti FPU (strata rýchlosti) – procedúry obsahujúce bežné (non-floating point) inštrukcie pre vykonávanie operácií s pohyblivou čiarkou
  + v minulosti vo forme samostatných koprocesorov (8087 pre 8086, podobne 80287 a 80387) [[5](https://en.wikipedia.org/wiki/Floating-point_unit)]
  + počnúc procesorom 80486 DX – FPU integrovaná do procesora (1989)
  + v ďalšom texte predmetom nášho záujmu – jednotka FPU integrovaná v procesore Pentium

## Organizácia jednotky FPU

* + registre jednotky FPU rozdelené do troch skupín [1]
    - údajové (data registers, 8x80 bitov)
    - riadiace a stavové (control and status registers, 3x16 bitov)
    - ukazovatele (pointer registers) – podpora spracovania výnimiek

|  |
| --- |
| Control register |
| Status register |
| Tag register |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 79 | 78 | 64 63 | 0 |  |  | 15 |  | 0 |
| ST7 |  |  |  |  |  |  |  |  |  |
| ST6 |  |  |  |  |  |  |  |  |  |
| ST5 |  |  |  |  |  |  |  |  |  |
| ST4 |  |  |  |  |  |  |  |  |  |
| ST3 |  |  |  |  |  |  |  |  |  |
| ST2 |  |  |  |  |  | 47 |  |  | 0 |
| ST1 |  |  |  |  |  |  |  | Instruction pointer |  |
| ST0 |  |  |  |  |  |  |  | Data pointer |  |

##### Údajové registre

|  |  |  |
| --- | --- | --- |
| sign | exponent | mantissa |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

* + osem registrov pre uloženie operandov s pohyblivou rádovou čiarkou (80 bitov)
  + organizované vo forme zásobníka
  + dostupné individuálne pod menami ST0, ST1, …, ST7
    - mená nie sú priradené staticky (ST0 – register predstavujúci vrchol zásobníka, ST1 – ďalší v poradí, …)
    - v stavovom registri FPU – ukazovateľ vrcholu zásobníka (TOS, 3 bity)
  + stav a obsah registra indikovaný značkou (2 bity, register značiek (Tag register))

##### Riadiace a stavové registre

* + tri registre rozmeru 16 bitov
    - Riadiaci register (FPU Control Register)
    - Stavový register (FPU Status Register)
    - Register značiek (Tag Register)

*Riadiaci register* [1]

* + voľba parametrov činnosti FPU (control word)
  + masky pre spracovanie výnimiek 15
    - Exception masks, 6 najnižších bitov

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | RC | PC |  |  | P M | U M | O M | Z M | D M | I M |

* + - nastavený bit blokuje generovanie výnimky
  + presnosť (PC, 2 bity)
    - zníženie internej presnosti
    - kompatibilita so staršími FPU
    - urýchlenie výpočtu
    - 00 – 24-bit, 01 – N/A, 10 – 53-bit, 11 – 64-bit
  + zaokrúhľovanie (RC, 2 bity)
    - 00 – najbližšie
    - 01 – nadol, 10 – nahor
    - 11 – skrátenie (orezanie nadbytočných bitov)

Rounding control

Precision control

0

Exception masks PM = Precision UM = Underflow OM = Overflow

ZM = Divide-by-zero

DM = Denormalized operand IM = Invalid operation

*Stavový register* [1]

* + stav jednotky FPU
  + signalizácia povahy výsledku aritm. operácií (C0 – C3)
    - podobnosť s príznakmi registra FLAGS (C0~CF, 15

Condition code

Stack fault

0

C2~PF, C3~ZF)

* + - C1 – podtečenie/pretečenie zásobníka
    - využitie pri realizácii podmienených skokov –

C TOS C C C E S

2 1 0

B

3

S

F

P U O

E E E

Z D I

E E E

kopírovanie do FLAGS registra CPU

* uloženie stavu (napr. do registra AX – fstsw)
* naplnenie FLAGS registra (sahf)
* identifikácia registra na vrchole zásobníka (TOS, 3 bity)
  + 8 údajových registrov FPU organizovaných cyklicky
  + TOS identifikuje register na vrchole
  + TOS aktualizovaný pri vkladaní/výbere údajov

Busy Top-of- stack

Error status

Exception flags PE = Precision

UE = Underflow OE = Overflow

ZE = Divide-by-zero

DE = Denormalized operand IE = Invalid operation

* príznaky výnimiek (nastavené, ak sa vyskytne výnimka počas spracovania)
  + 6 najnižších bitov
  + IE – výnimku môžu spôsobiť: aritmetická operácia, operácia so zásobníkom (SF bit nastavený)
    - podtečenie (C1 = 0) / pretečenie (C1 = 1) zásobníka ďalej indikované bitom C1
  + OE, UE – číslo príliš veľké / malé
  + PE – číslo nie je možné reprezentovať presne
  + ZE – delenie nulou
  + DE – aritmetická inštrukcia sa pokúša spracovať denormalizovaný operand

*Register značiek* [1]

* informácia o stave a obsahu údajových registrov
* značka indikuje, či príslušný register je prázdny, ak nie je, identifikuje jeho obsah
  + 00 – platný (valid)
  + 01 – nula (zero)
  + 10 – špeciálny (invalid, infinity, denormal)
  + 11 – prázdny (empty)

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ST7  Tag | ST6  Tag | ST5  Tag | ST4  Tag | ST3  Tag | ST2  Tag | ST1  Tag | ST0  Tag |

## Inštrukcie jednotky FPU

* inštrukcie pre presuny údajov, aritmetické operácie, porovnanie, transcendentné funkcie a ďalšie
* uvádzame výber z množiny podporovaných inštrukcií, detaily napr. v [2, 3, 4]
* všetky mnemoniky FPU inštrukcií začínajú písmenom f
* operácie so zásobníkom POP/TOP

##### Presuny údajov

* vloženie (load) operandu do zásobníka FPU (FPU stack)

fld src (src – FPU R/M; M:32/64/80-bit číslo v pohyblivej čiarke)

* + dekrementuje TOS a uloží src do (nového) ST0, čísla nižšej presnosti konvertované na internú presnosť (80-bit)
  + ak je operandom FPU R, STi – pred zmenou TOS
* vloženie celého čísla

fild src (src – 16/32-bit celé číslo v pamäti)

* + konvertuje číslo na internú presnosť a uloží do ST0
* inštrukcie pre vloženie konštánt (bez operandov) [1]

|  |  |
| --- | --- |
| fldz | +0.0 do zásobníka FPU |
| fld1 | +1.0 do zásobníka FPU |
| fldpi | π do zásobníka FPU |
| fldl2t | log210 do zásobníka FPU |
| fldl2e | log2e do zásobníka FPU |
| fldlg2 | log102 do zásobníka FPU |
| fldln2 | loge2 do zásobníka FPU |

* presun (store) hodnoty z vrcholu FPU zásobníka (bez odstránenia, TOP)

fst dest (dest – FPU R/M)

* + pamäťový operand 32/64/80-bit (v prípade 32 a 64-bit – konverzia)
* výber hodnoty z vrcholu FPU zásobníka (s odstránením – inkrement TOS po výbere z ST0, POP)

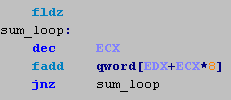
fstp dest (dest – FPU R/M)

* + ak je operandom FPU R, STi – pred zmenou TOS

|  |  |  |  |
| --- | --- | --- | --- |
| *Príklad*: | fstp | ST1 | ; odstráni ST1, ST0 ponechá na vrchole |
|  | fstp | ST0 | ; odstráni ST0 (POP) |

* celočíselná verzia – presun (konvertuje hodnotu v ST0 (znamienkové celé číslo) a uloží do pamäti)

fist dest (dest – M)

* celočíselná verzia – s výberom (POP) *Príklad*:

fistp dest

##### Aritmetické operácie

* operácia sčítania

fadd (výber ST0 a ST1, sčítanie a uloženie sumy späť na FPU zásobník) fadd src (src – FPU R/M, M:32/64-bit; ST0 ← ST0 + src)

|  |  |  |  |
| --- | --- | --- | --- |
| fadd | dest,src | (src, dest | – FPU R, jeden z nich ST0; dest ← dest + src) |
| faddp | dest,src | (src, dest | – FPU R, src = ST0; dest ← dest + src; POP) |

faddp dest (dest – FPU R, dest ← dest + ST0; POP) fiadd src (src – M, 16/32-bit celé číslo)

* operácia odčítania

fsub (výber ST0 a ST1, výpočet ST1 – ST0, uloženie výsledku späť)

fsub src (src – M, 32/64-bit; ST0 ← ST0 - src)

fsub dest,src (src, dest – FPU R, jeden z nich ST0; dest ← dest - src) fsubp dest,src (verzia s výberom ST0 (POP); src = ST0)

fsubp dest (dest – FPU R; dest ← dest – ST0, POP)

fsubr src (ST0 ← src – ST0, reverse subtract)

fisub src (src – M, 16/32-bit celé číslo; dostupné aj fisubr src)

* operácia násobenia

fmul (výber ST0 a ST1, výpočet ST0 \* ST1, uloženie späť)

fmul src (src – M, 32/64-bit; ST0 ← ST0 \* src)

|  |  |  |  |
| --- | --- | --- | --- |
| fmul | dest,src | (src, dest – FPU R, jeden z nich | ST0; dest ← dest \* src) |
| fmulp | dest,src | (verzia s výberom ST0 (POP); src | = ST0) |
| fmulp | dest | (dest – FPU R, dest ← dest \* ST0; POP) | |
| fimul | src | (src – M, 16/32-bit celé číslo; ST0 ← ST0 \* src) | |

* operácia delenia

fdiv (výber ST0 a ST1, výpočet ST1 / ST0, uloženie, dostupné aj fdivr)

fdiv src (src – M, 32/64-bit; ST0 ← ST0 / src)

fdiv dest,src (src, dest – FPU R, jeden z nich ST0; dest ← dest / src) fdivp dest,src (verzia s výberom ST0 (POP); src = ST0)

fdivr src (src – M, 32/64-bit; ST0 ← src / ST0) fdivrp dest (dest – FPU R; dest ← ST0 / dest, POP)

fidiv src (src – M, 16/32-bit celé číslo; dostupné aj fidivr src)

##### Operácie porovnania

* porovnanie dvoch čísel s pohyblivou rádovou čiarkou

fcom src (src – FPU R/M; M:32/64-bit)

* + porovnanie ST0 a src, nastavenie príznakov stavového registra FPU

|  |  |  |  |
| --- | --- | --- | --- |
| **Relácia** | **C3** | **C2** | **C0** |
| ST0 > src | 0 | 0 | 0 |
| ST0 = src | 1 | 0 | 0 |
| ST0 < src | 0 | 0 | 1 |
| Not comparable | 1 | 1 | 1 |

* + príznaky C0, C2, C3 – indikácia relácie:

fcom (bez operandov)

* + porovnanie ST0 s ST1, nastavenie príznakov
  + porovnanie s výberom ST0 (fcomp), resp. s výberom ST0 a ST1 (fcompp)
* porovnanie ST0 s celým číslom v pamäti

ficom src (src – M, 16/32-bit)

* + porovnanie s výberom (ficomp src)
* porovnanie s nulou (ST0 s 0.0)

|  |  |  |  |
| --- | --- | --- | --- |
| **Typ** | **C3** | **C2** | **C0** |
| Unsupported | 0 | 0 | 0 |
| NaN | 0 | 0 | 1 |
| Normal | 0 | 1 | 0 |
| Infinity | 0 | 1 | 1 |
| Zero | 1 | 0 | 0 |
| Empty | 1 | 0 | 1 |
| Denormal | 1 | 1 | 0 |

ftst

* preskúmanie povahy operandu v ST0 (examine)

fxam

* + znamienko v C1 (1 – záporné), ďalšia informácia v príznakoch C0, C2 a C3 [1]

##### Ďalšie operácie

* funkcie *sin* (*cos*) – výber argumentu (radiány) z FPU zásobníka, výpočet funkcie *sin* (*cos*), uloženie hodnoty späť

fsin (fcos)

* výpočet odmocniny z hodnoty na vrchole zásobníka (ST0) a nahradenie hodnoty výsledkom (záporná hodnota – výnimka)

fsqrt

* zmena (inverzia) znamienka čísla v ST0

fchs

* absolútna hodnota čísla v ST0 (výsledok v ST0)

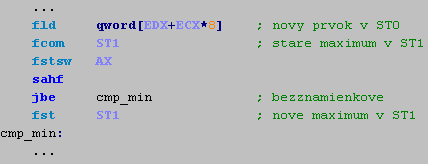
fabs

* načítanie riadiaceho slova (control word, 16-bit) do riadiaceho registra FPU

fldcw src (src – M, 16-bit)

* uloženie riadiaceho slova (po vykonaní inštrukcie sú bity C0 – C3 nedefinované)

*Príklad*:

fstcw dest (dest – M, 16-bit)

* uloženie stavového slova (status word, 16-bit), C0 – C3 nedef.

fstsw dest (dest – M/register AX)

* kopírovanie AH do FLAGS registra CPU

sahf

*Príklad*: Riešenie kvadratickej rovnice (C časť – interakcia s používateľom, ASM časť – výpočet koreňov) [1].

C časť:

#include <stdio.h>

int main(void)

{

double a, b, c, root1, root2;

extern int quad\_roots(double, double, double, double\*, double\*);

printf("Zadaj konstanty a, b, c: ");

Kvadratická rovnica:

*ax*2+*bx*+*c*=0

Výpočet koreňov:

−*b*+√*b*2−4 *ac*

scanf("%lf %lf %lf",&a, &b, &c);

if (quad\_roots(a, b, c, &root1, &root2))

printf("koren1 = %lf a koren2 = %lf\n", root1, root2);

else

printf("Neexistuju realne korene.\n");

return 0;

}

*root* 1=

*root* 2=

2 *a*

−*b*−√*b*2−4 *ac*

2 *a*

ASM časť:

|  |  |  |
| --- | --- | --- |
| %define a qword[EBP+8]  %define b qword[EBP+16]  %define c qword[EBP+24]  %define root1 dword[EBP+32]  %define root2 dword[EBP+36]  segment .text global \_quad\_roots  \_quad\_roots:  enter 0,0  fld a ; a  fadd ST0 ; 2a  fld a ; a,2a  fld c ; c,a,2a fmulp ST1 ; ac,2a fadd ST0 ; 2ac,2a fadd ST0 ; 4ac,2a fchs ; -4ac,2a  fld b ; b,-4ac,2a  fld b ; b,b,-4ac,2a fmulp ST1 ; b\*b,-4ac,2a faddp ST1 ; b\*b-4ac,2a  ftst ; porovnaj b\*b-4ac s 0 fstsw AX ; stavove slovo do AX sahf | jb no\_real\_roots  fsqrt ; sqrt(b\*b-4ac),2a fld b ; b,sqrt(b\*b-4ac),2a fchs ; -b,sqrt(b\*b-4ac),2a  fadd ST1 ; -b+sqrt(b\*b-4ac),sqrt(b\*b-4ac),2a fdiv ST2 ; -b+sqrt(b\*b-4ac)/2a,sqrt(b\*b-4ac),2a mov EAX,root1  fstp qword[EAX] ; uloz koren1  fchs ; -sqrt(b\*b-4ac),2a  fld b ; b,-sqrt(b\*b-4ac),2a  fsubp ST1 ; -b-sqrt(b\*b-4ac),2a  fdivrp ST1 ; -b-sqrt(b\*b-4ac)/2a mov EAX,root2  fstp qword[EAX] ; uloz koren2  mov EAX,1 ; realne korene existuju jmp short done  no\_real\_roots:  sub EAX,EAX ; EAX=0 (ziadne realne korene) done: | |
| leave ret |  |

#### Študijná literatúra:

1. Dandamudi, S.P.: Introduction to Assembly Language Programming, Springer Science+Business Media, Inc., 2005.
2. Carter, A.P.: [PC Assembly Language](https://pacman128.github.io/static/pcasm-book.pdf), 2019.
3. [Intel® 64 and IA-32 Architectures Software Developer’s Manual](https://software.intel.com/en-us/articles/intel-sdm), Volume 2: Instruction Set Reference, Chapter 3, 2023.
4. [NASM](https://www.nasm.us/) – The Netwide Assembler, The NASM Development Team, 1996-2015.
5. Wikipedia, [Floating-point unit](https://en.wikipedia.org/wiki/Floating-point_unit), 2023.

# Stručne o x86-64

* + pôvodná špecifikácia vytvorená spoločnosťou AMD, 64 bitová verzia súboru inštrukcií x86 (1999, plná špecifikácia 08/2000) [1]
  + prvé významné rozšírenie architektúry x86 navrhnuté inou spoločnosťou ako Intel (ten ju nakoniec prijal a implementoval)
  + AMD64 ako evolúcia existujúcej x86 architektúry s podporou 64-bit výpočtov (Intel vytvoril vlastnú, nekompatibilnú IA-64)
  + prvý procesor s podporou AMD64, Opteron, uvedený v apríli 2003 (prvý Intel x64 procesor Xeon, 2004)
  + dva nové operačné (pod)režimy (*Long mode*: *64-bit mode* + *Compatibility mode*) a nový 4-úrovňový systém stránkovania

*64-bit mode*

* podpora väčšieho množstva pamäte, rozšírenie rozmeru (64 bit) a počtu (16) GPR registrov, ďalšie rozšírenia
* operácie s pohyblivou čiarkou – preferované SSE(2) inštrukcie, avšak x87/MMX stále dostupné
* vektorové registre (16 x 128 bit, neskôr rozšírené aj tie), rôzne formáty údajov
* podpora 64 bitových operandov a 64 bitového režimu adresovania

*Compatibility mode*

* podpora behu 16 a 32 bitových aplikácií (16/32 bitové inštrukcie stále dostupné)
* x86-64 procesor stále štartuje v reálnom režime (real mode) za účelom spätnej kompatibility

#### Základné vlastnosti architektúry

* GPR rozšírené na 64 bitov, ako aj príslušné (celočíselné) operácie (aritmetické, logické, presuny údajov)
* ďalšie registre (RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI, R8, R9, … R15)
* ďalšie XMM (SSE) registre (128 bitov) – z pôvodných 8 na 16 (XMM8 – XMM15) [2]
* väčší virtuálny adresný priestor – 64-bitový formát virtuálnej adresy (aktuálne využívaných spodných 48 bitov)
* väčší fyzický adresný priestor – pôvodná špecifikácia 40-bitové fyzické adresy, aktuálne implementácie 48-bitové
* väčší fyzický adresný priestor aj v Legacy mode
* RIP relatívne adresovanie (position independent code, x86 – iba inštrukcie skokov, v x86-64 aj prístup k údajom) [6]
* SSE inštrukcie (adoptovanie SSE a SSE2 od spoločnosti Intel)
* No-execute bit (NX bit) – OS môže určiť, ktoré stránky môžu obsahovať vykonateľný kód a ktoré nie (podobná možnosť bola dostupná aj prípade využitia segmentových deskriptorov (avšak len pre celý segment); moderné OS segmentáciu obchádzajú)
* odstránenie/redukcia niektorých vlastností (segmentované adresovanie, TSS mechanizmus, virtual 8086 mode – ostávajú prítomné v Legacy mode); niektoré inštrukcie (uloženie/obnova SR (CS, SS, DS, ES) na zásobníku, PUSHA/POPA, BOUND, INTO ...) [1,3]

**Operačné režimy architektúry** (ďalšie detaily napr. v [1])

|  |  |  |  |
| --- | --- | --- | --- |
| Operačný | | Požadovaný OS | Kód |
| režim | pod-režim |
| Long mode | 64-bit mode | 64-bit OS, 64-bit UEFI | 64-bit |
| Compatibility mode | Bootloader alebo 64-bit OS | 32-bit |
| 16-bit protected mode |
| Legacy mode | Protected mode | Bootloader, 32-bit OS, 32-bit UEFI | 32-bit |
| 16-bit protected mode OS | 16-bit protected mode |
| Virtual 8086 mode | 16-bit protected mode OS alebo 32-bit OS | podmnožina real mode |
| Unreal mode | Bootloader alebo real mode OS | real mode |
| Real mode | Bootloader, real mode OS, OS využívajúci BIOS rozhranie | real mode |

**Registre** [6]

|  |  |  |  |
| --- | --- | --- | --- |
| RAX | | | |
| zero-extended | EAX | | |
| not modified | | AX | |
| not modified | | AH | AL |

|  |  |  |  |
| --- | --- | --- | --- |
| R8 | | | |
| zero-extended | R8D | | |
| not modified | | R8W | |
| not modified | | | R8B/L |

* NASM štandardne podporuje názvy R8B – R15B [7]
  + v niektorých dokumentoch (Intel) použité názvy R8L – R15L
  + možnosť použiť makrá v NASM pre názvy R8L – R15L
* SIMD registre neskôr ďalej rozširované (rozmer, počet – XMM (128 bitov), YMM (256 bitov), ZMM (512 bitov)) [8]

#### Stránkovanie (x86)

* transformácia lineárnej adresy na fyzickú (voliteľná, bit PG v registri CR0) [4]
* stránka (*page frame*) – 4KB súvislý blok fyzickej pamäti
* lineárna adresa – odkazuje nepriamo na fyzickú adresu (špecifikuje tabuľku stránok, stránku v rámci tejto tabuľky a offset)

|  |  |  |
| --- | --- | --- |
| 31 22 | 21 12 | 11 0 |
| DIR | PAGE | OFFSET |

* spôsob prevodu polí DIR, PAGE a OFFSET na fyzickú adresu (2 úrovne tabuliek)
  + DIR – index (10 bitov) do adresára stránok (*Page directory*)
  + PAGE – index (10 bitov) do tabuľky stránok (*Page table*) určenej adresárom stránok
  + OFFSET – 12 bitov, adresovanie bajtu v rámci stránky (určenej z tabuľky stránok)

Dir entry

DIR

Page directory

Page table

Page frame

CR3

Physical address

PG table entry

OFFSET

PAGE

Tabuľka stránok (page table)

* pole 32-bitových špecifikátorov stránok (tiež stránka, 4KB, max. 1024 položiek)
* dve úrovne tabuliek
  + prvá úroveň – adresár stránok (max. 1K tabuliek stránok druhej úrovne)
  + druhá úroveň – tabuľka stránok, adresovanie max. 1K stránok
* všetky tabuľky adresované 1 adresárom: 1M stránok (220)
* každá stránka má rozmer 4KB (212 B), tabuľky jedného adresára môžu obsiahnuť celý 32-bitový adresný priestor (232 B)
* zmena adresára stránok – zmenou registra CR3

Položky tabuľky stránok

* položky majú jednotný formát pre každú z úrovní
* *page frame address* – fyzická začiatočná adresa stránky (vyšších 20 bitov)
* stránky začínajú na hraniciach 4K, t.j. nižších 12 bitov nevyužitých (pre adresovanie)
  + tieto bity využité na uloženie ďalších informácií (napr. P – present, ak P = 0 položka nie je platná pre prevod adresy) [4]

*Ďalší vývoj stránkovania*

* PAE (*Physical Address Extension*) – zavádza 3-úrovňovú hierarchiu tabuliek (namiesto 2-úrovňovej), Intel Pentium Pro, 1995 [5]
  + položka tabuľky stránok (*Page directory* aj *Page table*) má veľkosť 64 bitov
  + pribudla ďalšia tabuľka (*Page directory pointer table*) so 4 položkami, [pôvodná implementácia](https://en.wikipedia.org/wiki/Physical_Address_Extension#Design):
    - pole *page frame address* – rozšírené z 20 na 24 bitov (veľkosť offset-u zostáva 12 bitov)
    - zvýšenie veľkosti fyzických adries z 32 na 36 bitov (fyzická pamäť adresovaná CPU 4GB → 64GB)
  + aplikácie – naďalej 32-bitové adresy (limit 4GB vo flat režime)
  + OS mohol mapovať tento virtuálny priestor na fyzickú pamäť (max. 64GB)
* x86-64 CPU (*Long mode*) – ďalej rozširuje hierarchiu stránkovacích tabuliek na 4 úrovne
* v procesoroch, ktoré podporujú “no-execute“, NX bit je najvyšší bit (bit 63) položky tabuľky stránok

#### Formát inštrukcií

V 64 bitovom režime využívané aj ďalšie prefixy [11,12]:

* REX prefix (špecifikácia GPR a SSE registrov, 64 bitových operandov, …)
  + nie všetky inštrukcie vyžadujú prefix v 64-bit režime
* VEX prefix (kódovanie AVX inštrukcií) / EVEX (AVX-512 inštrukcií)

#### Programovanie

* ABI (Application Binary Interface) – parametre, návratová hodnota, zásobník … [6,10]
* dôležité pri volaní C funkcií, služieb OS ...
* volacie konvencie – Microsoft x64 ABI (Windows) [9], System V x64 ABI (Linux, BSD, Mac)

*Microsoft x64 ABI*

* prvé 4 parametre v registroch, ďalšie v zásobníku
  + RCX, RDX, R8, R9 (celočíselné)
  + XMM0, XMM1, XMM2, XMM3 (s plávajúcou čiarkou)
* pre funkcie s premenlivým počtom parametrov (*vararg*) – hodnoty s plávajúcou čiarkou duplikované v zodpovedajúcich GPR
* návratová hodnota v RAX alebo XMM0 (*float*, *double*, …)
* hodnoty v niektorých registroch môžu byť funkciou zmenené (*volatile* – RAX, RCX, RDX, R8, R9, R10, R11)
* hodnoty niektorých registrov musí funkcia zachovať (push/pop) (*non-volatile* – RBX, RBP, RDI, RSI, R12, R13, R14, R15)
* zásobník
  + alokovanie miesta pre uložené registre, lokálne premenné, parametre
  + zarovnaný na 16B (*non-leaf* funkcia)
  + volajúci (caller) alokuje 32B (*shadow space*) pred volaním funkcie (pre registre RCX, RDX, R8, R9)
  + volajúci vyčistí zásobník (po návrate z volanej funkcie)

*Príklad*: ilustrácia vlastností operácií na registroch v 64 bitovom režime.

global main extern printf section .text

pformat: db "%.16llx",13,10,0

main:

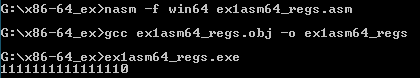
sub mov dec mov lea call add ret

rsp,28h ; vyhradenie a zarovnanie r8,1111111111111111h

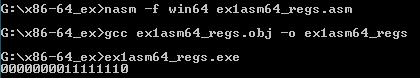
r8b rdx,r8

;dec r8d

rcx,pformat printf rsp,28h



Po zmene (dec r8d):



*Príklad*: ilustrácia využitia vektorových registrov (XMM) a SIMD inštrukcií pri realizácii operácií s plávajúcou čiarkou.

global main extern printf

section .data

nums1 dq 2.5,3.6

nums2 dq 3.1,4.2

nums3 dq 0.0,0.0

f\_sum db "The sum of %f and %f is %f",10,0

section .text main:

sub rsp,28h movupd xmm0,[nums1] addpd xmm0,[nums2] movupd [nums3],xmm0 mov rcx,f\_sum

mov rdx,[nums1]

mov r8,[nums2]

mov r9,[nums3]

call printf

mov rcx,f\_sum

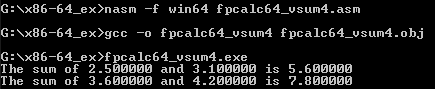
mov rdx,[nums1+8]

mov r8,[nums2+8]

mov r9,[nums3+8]

call printf

add rsp,28h ret



* SSE2 inštrukcia movupd pre presun zabalených (packed) čísel s plávajúcou čiarkou s dvojitou presnosťou
* SSE2 inštrukcia addpd pre realizáciu sčítania zabalených čísel s plávajúcou čiarkou s dvojitou presnosťou [3]

*Príklad*: ilustrácia vyhradenia miesta a zarovnania údajov v zásobníku.

#include <stdio.h>

long sum(long a, long b, long c);

void main()

{

long a = 1; long b = 2; long c = 3; long d = 4;

c = sum(a,b,c);

asm("movl %%esp,%0" :"=r" (d)); printf("Sum: %lx Stack: %lx", c, d);

}

long sum(long a, long b, long c)

{

return a + b + c;

}

.LC0:

.ascii "Sum: %lx Stack: %lx\0"

...

main:

push rbp

mov rbp, rsp sub rsp, 48

mov DWORD PTR -4[rbp], 1 mov DWORD PTR -8[rbp], 2 mov DWORD PTR -12[rbp], 3 mov DWORD PTR -16[rbp], 4

mov ecx, DWORD PTR -12[rbp] mov edx, DWORD PTR -8[rbp] mov eax, DWORD PTR -4[rbp] mov r8d, ecx

mov ecx, eax call sum

mov DWORD PTR -12[rbp], eax

/APP

# 11 "ex2\_stack\_rsp.c" 1 movl %esp,eax

# 0 "" 2

/NO\_APP

mov DWORD PTR -16[rbp], eax mov edx, DWORD PTR -16[rbp] mov eax, DWORD PTR -12[rbp] mov r8d, edx

mov edx, eax

lea rcx, .LC0[rip] call printf

add rsp, 48

pop rbp ret

|  |  |  |  |
| --- | --- | --- | --- |
| main |  |  | |
| RBP → | old RBP | |
|  | a,b | local vars |
|  | c,d |
|  |  | shadow space |
|  | R8D |
|  | EDX |
| RSP → | ECX |
| sum |  | RIP | |
| RBP → | old RBP | |

(offset)

stack

(offset)

stack

sum:

push rbp

mov rbp, rsp

mov DWORD PTR 16[rbp], ecx mov DWORD PTR 24[rbp], edx mov DWORD PTR 32[rbp], r8d mov edx, DWORD PTR 16[rbp] mov eax, DWORD PTR 24[rbp] add edx, eax

mov eax, DWORD PTR 32[rbp] add eax, edx

pop rbp ret

.ident "GCC: (GNU) 9.1.0"

|  |  |  |  |
| --- | --- | --- | --- |
| main |  |  | |
| RBP → | old RBP | |
|  | a,b | local vars |
|  | c,d |
|  | e |
|  | alignment | |
|  |  | shadow space |
|  | R8D |
|  | EDX |
| RSP → | ECX |
| sum |  | RIP | |
| RBP → | old RBP | |

Ako sa zmení situácia v zásobníku, ak pridáme lokálnu premennú (long e = 5;)?

*System V x64 ABI*

* 6 celočíselných parametrov v RDI, RSI, RDX, RCX, R8, R9
* 8 registrov pre parametre s plávajúcou čiarkou XMM0 – XMM7 (*variadic* funkcie – počet použitých vektorových registrov v RAX)
* ďalšie parametre v zásobníku; návratové hodnoty – celočíslené RAX (RDX), s plávajúcou čiarkou XMM0 (XMM1)
* bez *home/shadow space* ako je to v prípade Microsoft x64 ABI
* využívané zarovnanie (hranica 16B)
* funkcie, ktoré nevolajú ďalšie funkcie (*leaf-node functions*) – môžu využiť tzv. *red zone* (128B oblasť pod RSP) [10]

*Príklad*: ilustrácia programovania v prostredí OS Linux.

global main extern printf

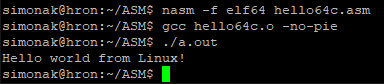
section .data

msg db "Hello world from Linux!",10,0

section .text main:

push mov mov mov call mov pop ret

rbp rbp,rsp rdi,msg rax,0 printf rsp,rbp rbp



Zdroje:

1. Wikipedia, [x86-64](https://en.wikipedia.org/wiki/X86-64), 2023.
2. Wikipedia, [Streaming SIMD Extensions](https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions), 2023.
3. Cloutier, F.: [x86 and amd64 instruction reference](https://www.felixcloutier.com/x86/), 2022.
4. Intel 80386 Reference Programmer's Manual, [Page Translation](https://pdos.csail.mit.edu/6.828/2018/readings/i386/s05_02.htm), 1986.
5. Wikipedia, [Physical Address Extension](https://en.wikipedia.org/wiki/Physical_Address_Extension), 2023.
6. Larimer, J.: [Intro to x64 Reversing](http://lolcathost.org/b/introx86.pdf), SummerCon 2011.
7. NASM – The Netwide Assembler, Documentation, [Chapter 12: Writing 64-bit Code (Unix, Win64)](https://www.nasm.us/xdoc/2.15.05/html/nasmdo12.html), 2020.
8. Wikipedia, Registers available in the x86-64 instruction set, 2014.
9. Microsoft technical documentation, [x64 calling convention](https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170), 2022.
10. Wikipedia, [x86 calling conventions](https://en.wikipedia.org/wiki/X86_calling_conventions), 2022.
11. [Intel® 64 and IA-32 Architectures Software Developer’s Manual](https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf), Intel Corporation, 2023.
12. OSDev Wiki, [X86-64 Instruction Encoding](https://wiki.osdev.org/X86-64_Instruction_Encoding), 2021.