

Úroveň strojového kódu – Intel Pentium 32bit

– na reálnom príklade (proc. Intel Pentium 32bit)

externé odkazy:

- [1] <http://user.edi.fmph.uniba.sk/tomcsanyi/Obr.PNG>
- [2] <http://user.edi.fmph.uniba.sk/tomcsanyi/Instrukcie%20Intel%20Pentium%20vyber%20pre%20AI.pdf>
- [3] <http://www.jegerlehner.ch/intel/IntelCodeTable.pdf>

V poznámkach chyba: vysvetlenie dvojkového doplnkového kódu (záporné čísla)

Historické okienko

1972 – prvý mikroprocesor – Intel 4004 (4bitový)

1974 – Intel 8080 (8bitový) – začiatok éry osobných počítačov v USA

1978 – Intel 8086 (16bitový) – moc sa neujal lebo všetci vývojári mali počítače pripravené na 8bit.
– začiatok x86 éry (kompatibilita až do dnešných čias)

1979 – Intel 8088 (vo vnútri 16bitový, ale navonok 8bitový) – na ňom bolo postavené prvé IBM PC

1983 – Intel 80286 (16bitový, priniesol chránený režim)

1985 – Intel 80386 (32bitový) – v podstate sa od 386 až dodnes „nič“ nezmenilo

1989 – Intel 80486

1993 – prvé Pentium (nenazvali to 586 ale Pentium kvôli ochranej známke)

Základné vlastnosti Pentium

pozri [1]

- rovnakoveľa bitové adresy aj dáta

Programátorovi prístupné registre:

EAX	CS
EBX	SS
ECX	DS
EDX	ES
ESI	FS
EDI	GS
EBP	EFLAGS
ESP	EIP

Registre EAX, EBX, ECX, EDX sú rovnocenné, sú 32bitové.

Spodná polovica EAX je označená AX, AX je ešte rozdelené na AH, AL (8bit). To isté platí aj pre EBX, ECX, EDX.

Register EIP (IP – instruction pointer („program counter“) – v ňom je uložená adresa nasledujúcej inštrukcie)

Register EFLAGS – príznakový register, rozdelený na bity, každý bit niečo znamená

-> sú tam „bočné“ (pomocné) výstupy z ALU:

CF – carry flag - n+1. bit aritmetickej operácie, prenos z najvyššieho rádu na n+1.
(čiže byt, ktorý sa v pôvodnom čísle „zabudne“)

ZF – zero flag - 1-ak výsledok je nula, 0-ak výsledok je nenulový

SF – sign(?) flag - 1-ak výsledok je záporný, 0-ak výsledok je kladný

PF – parity flag

OF – overflow flag; 1-ak výsledok pretiekol v dvojkovom doplnku

...

- > okrem toho obsahuje rôzne prepínače, ktorými sa dá modifikovať správanie procesora:
 - IF – či sú povolené prerušenia
 - DF – spôsob udávania adresy (od zadanej adresy hore / od zadanej adresy dole)
 - ...

Základné inštrukcie

pozri [2] pre podrobný popis inštrukcií

- na úrovni assemblera (symbolického jazyka – pomenúva inštrukcie)
- inštrukcie môžu mať 0-4 operandy (parametre), napr.
 - meno nejakého registra
 - literal (konštanta – nejaká hodnota)
- operandy sa čítajú zprava:


```
MOV EAX, 5    = presun hodnoty 5 do registra EAX (EAX <- 5)
```
- návestia – používajú sa na skoky („skoč na návestia XXX“):


```
@1: MOV EAX, 5
```
- inštrukcie presunov nemenia hodnoty v EFLAGS

Aritmetické inštrukcie (v celých číslach)

Sčítanie:

```
ADD EAX, EBX    EAX:=EAX+EBX    // sčítanie bez prenosu
ADC EAX, EBX
```

- sčítanie s prenosom (add with carry)
- pomocou tohto sa dá implementovať ľubovoľne bitové sčítanie, napr. 64 bitov:

```
EBX          EAX          <-- operand A
EDX          ECX          <-- operand B

ADD EAX, ECX    // EAX := EAX+ECX; 33. bit sa uloži do CF
ADC EBX, EDX    // EBX := EBX+EDX+CF;
```

Odčítanie:

obdobne ako sčítanie:

```
SUB EAX, EBC    // sčítanie bez pôžičky (prenosu)
SBB EAX, EBC    // sčítanie s pôžičkou (prenosom)
```

Násobenie:

```
MUL CL          AX:=AL*CL    // MUL – 2 nezáporné čísla
```

- násobenie má len jeden operand, implicitne sa totiž počíta s registrom AL
- výsledok sa uloží do takého registra, aby sa tam zmestil (čo v závislosti od vstupov môžu byť rôzne registre)

```
IMUL           // IMUL – môže byť aj záporné číslo
```

Delenie:

- môže nastať „Delenie nulou“ aj v prípade, že sa výsledok nezmesť do výstupného registra
- DIV aj IDIV počítajú zároveň podiel aj zvyšok

Opačné číslo:

NEG AX – vyrobí číslo opačné

Porovnanie:

CMP EBX, EAX – vypočíta EBX-EAX, ale výsledok nikam neuloží
 – nastaví ale EFLAGS, čiže sa za ním dá pracovať s podmienenými skokmi

Logické inštrukcie

– na obsah registrov nepozerajú ako na čísla, ale ako na pole bitov a inštrukciu vykonajú na každom bite zvlášť

– AND, OR, XOR – porovnávajú postupne bity v dvoch registroch

AND AX, 15

AX: 011100101011**0110**

maska 15: 000000000000**1111**

výsledok: 000000000000**0110** -> ponechá mi len posl. 4 bity
 (vyžitie v registroch ako EFLAGS)

– NOT – zmení bity v registri (0->1, 1->0)

– TEST

TEST AL, 7 // spraví AL AND 7, výsledok neuloží, ale nastaví EFLAGS

Rotácie a posuny

*toto bolo pekne nakreslené na slidoch, žiaľ, ešte nie sú na Tomcsányiovej stránke (stav – piatok 15.3.)
 nákresy posunov a rotácií nájdete aj v [3]*

napr. chcem zistiť konkrétne 2 bity z registra (chcem zistiť YY, XX ma nezaujímá)

AL: XXXX**YYXX**

AND AL, 12 12: 0000**1100** -> týmto odstránim čo ma nezaujímá a už to
 môžem rotovať (aby som to napr. dostal
 doprava a vedel prečítať ako číslo 0..3)

RO_ rotácie – bit ktorý ide „dokola“ sa skopíruje do CF

RC_ rotácie – okrem bitov z registra sa do rotácie zapojí aj CF

SH_ posuny – číslo chápeme bez znamienka

SA_ posuny – číslo chápeme v dvojkovom doplnku so znamienkom

– samozrejme rotovať sa dá vpravo (R – ROR, RCR, SHR, SAR) aj vľavo (L – ROL, RCL, SHL, SAL)

– napr. SHR AL 6 – register AL posunie o 6 doprava

SHR AL, 1 AL: 00000110 // 6

AL: 00000011 // 3 (delenie dvomi)

```

        AL: 11111110 // -2
SHR AL, 1
        AL: 01111111 // 127 (asi nie delenie dvomi)

        AL: 11111110 // -2
SAR AL, 1
        AL: 11111111 // -1 (správne delenie dvomi)

```

Vetvenie programu, podmienky a cykly

pozri [2]

```

...
CMP EAX, 5
JNP @4
...
...
@4: ...

```

Pozor na porovnávanie:

znamienkovo:

11111111 < 00000001 (-1 < 1)

neznamienkovo:

11111111 > 00000001 (255 > 1)

- poučka: v stroj. kóde robím opačné testy ako vo vyšších jazykoch

Príklady:

<pre> IF ECX = 5 THEN ... (1); </pre>	<pre> CMP ECX, 5 JNE @1 ... (1) @1: ... // pokrač. programu </pre>
<pre> IF EAX > EBX THEN //neznamienkovo ... (1); ELSE ... (2); </pre>	<pre> CMP EAX, EBX JBE @1 // skoci ak EAX < EBX ... (1) JMP @2 // koniec 1. vetvy @1: ... (2) @2: ... // pokrač. programu </pre>
<pre> REPEAT ... (1) UNTIL EAX <= 5 //znamienkovo </pre>	<pre> @1: ... (1) CMP EAX, 5 JG @1 ... // pokrač. programu </pre>
<pre> WHILE AL = 0 DO ... (1); </pre>	<pre> @1: CMP AL, 0 JNE @2 ... (1) JMP @1 @2: ... // pokrač. programu </pre>
<pre> FOR I := 1 TO 5 DO ... (1); </pre>	<pre> MOV ECX, 5 @1: ... (1) SUB ECX, 1 JNZ @1 </pre>
<pre> // N moze byt aj 0 FOR I := 1 TO N DO ... (1); </pre>	<pre> MOV ECX, N JECXZ @2 @1: ... (1) SUB ECX, 1 JNZ @1 @2: ... // pokrač. programu </pre>

	<pre> to isté ako: MOV ECX, N JECXZ @2 @1: ... (1) LOOP @1 // vždy s reg. ECX!! @2: ... // pokrač. programu </pre>
--	---

Vysvetlenie k poslednému príkladu:

Cykly typu FOR sa dajú jednoducho spraviť tak, že do ECX (jedine ECX!) sa uloží počet opakovaní a použije sa LOOP (ten skočí na danú návěst', ak ECX > 0; a spraví ECX:=ECX-1). Samozrejme, že sa takto nedajú spraviť vnorené for cykly.

Zásobník

- LIFO – last in first out
- používa sa pre pamätanie návratových adries pri použití podprogramov

<pre> BEGIN ... X; END; PROCEDURE X; BEGIN ... Y; ... Z; END; PROCEDURE Y; BEGIN ... Z; END; PROCEDURE Z; BEGIN ... END; </pre>	<pre> @MAIN: ... CALL @X ... ABC // = koniec programu @X: ... CALL @Y ... CALL @Z ... RET // navrat do posl. volania @Y: ... CALL @Z RET @Z: ... RET </pre>
--	---

- zásobník je v princípe pamäť – je nutné si povedať, kde v pamäti budeme mať zásobník – to sa robí pri štarte programu
- register ESP – Stack Pointer – register ktorý ukazuje na vrchol zásobníka
- register EBP – pomocný register – ukazuje na miesto, kde má aktuálne bežiaci procedúra parametre
- do zásobníka však nejdú len návratové adresy, ale napr. aj lokálne premenné, parametre s ktorými volám procedúry, ...

ESP, EBP bližšie vysvetlené v časti „Adresovanie pamäte“

Inštrukcie pracujúce so zásobníkom

- CALL, RET – robia implicitne so zásobníkom
- okrem toho existujú aj PUSH, POP, PUSHFD, POPFD

výpočet výrazu $(A+3)*(C+4^{(A-B)})$ pri počítaní zľava doprava:

```
MOV EAX, A
ADD EAX, 3
PUSH EAX
MOV EAX, C
PUSH EAX
MOV EAX, 4
PUSH EAX
MOV EAX, A
SUB EAX, B
POP EBX
* umocnenie EBX, EAX
MOV EAX, EBX
POP EBX
ADD EAX, EBX
POP EBX
MUL EAX, EBX
```

- napríklad tu sa používa zásobník.... Ale príklad to bol úplne na hovno. :-)

Narábanie s EFLAGS

pozri [2]

STC, CLC, CMC

Príklad: Na vstupe v EAX je číslo X, na výstupe bude v AL počet jedničiek v dvojkovom zápise X.

```
MOV ECX, 32
MOV BL, 0
@2: SHL EAX, 1
JNC @1
INC BL // = ADD BL, 1
@1: LOOP @2
MOV AL, BL
```

Optimalizácia:

- na úrovni kompilátora – nájst' spôsob ako spraviť niektoré veci efektívnejšie
- predchádzajúci príklad prepísaný (bez skoku):

```
MOV ECX, 32
MOV BL, 0
@2: SHL EAX, 1
ADC BL, 0 // BL := BL + 0 + CF
LOOP @2
MOV AL, BL
```

- ešte inak (skončiť okamžite, ak už sú ďalej len nuly):

```
MOV BL, 0
@1: SHL EAX, 1
ADC BL, 0
CMP EAX, 0
JNE @2
MOV AL, BL
```

- tento program je síce možno dlhší (nie na počet riadkov v asembleri, ale na počet bitov v ktorých sú zakódované inštrukcie), ale môže byť rýchlejší a ušetrili sme 1 register

Príklad: Na vstupe je v AL číslo X, na výstupe je v AL číslo obrátené

Napríklad: 01011001 -> 10011010

```
MOV ECX, 8
@1: SHL AL, 1
    RCR BL, 1
    LOOP @1
MOV AL, BL
```

Assembler v Delphi

Predávanie parametrov

```
// v registri: EAX      EDX      ECX      EAX
// resp. jeho
// casti:      AL      EDX      CL      AX
function Ahoj(a:byte, b:integer, c:boolean) : word;
asm
...
end;
```

- čiže vstupy sú v EAX, EDX, ECX (v takomto poradí! - resp. ich príslušne bitovej podčasti), výstup je v EAX (v príslušnej podčasti) – napr. byte má 8 bitov, uloží sa do 8 bitovej podčasti EAX, čiže do AL; integer má 32 bitov, preto zaberie celé EDX, ...
- registre EAX, EDX a ECX môžem ľubovoľne meniť (môžem pritom ale prísť o parametre)
- registre EBX, ESI, EDI treba uchovať (najjednoduchšie na začiatku pushnúť do zásobníka a nakoniec popnúť naspäť)
- registre EBP a ESP nemeniť

```
function ObratCislo(x:byte) : byte
asm
MOV ECX, 8
@1: SHL AL, 1
    RCR DL, 1
    LOOP @1
MOV AL, DL
end;
```

Adresovanie pamäte

[...] - adresa pamäte

- **priama adresa**
 - bude sa čítať priamo z uvedenej adresy
 - používa sa pri globálnych premenných (tieto sú uložené niekde v pamäťovom priestore a procedúry vedia kde (odkiaľ ich majú čítať))
- ```
MOV EAX, [12840] // do EAX sa uloží hodnota pamäte 12840
```

- **nepriama adresa**

var a : array[1..10] of byte;

nech začiatok pola je na adrese 1200, potom adresa  $a[i] = 1200 + i - 1$

```
MOV EDX, [EBX]
```

- do EDX sa uloží to, čo je na adrese uloženej v EBX

- **indexovaná adresa**

- používa sa na indexovanie globálnych polí, to čo príklad vyššie (v EAX je index pola)

```
MOV BL, [1199+EAX]
```

- 1200 je začiatok pola, v EAX je index

```
MOV [1199+EAX*4], ECX
```

- ak máme pole typu integer (integer=4 bajty) tak ešte krát 4
- var a:array [1..10] of integer;
- $adr(A[i]) = adr(A) + I*4 - 4*1$
- indexované adresovanie je možné len pre polia, ktorých prvok má veľkosť 1, 2, 3 alebo 4 bajty. Pri väčších prvkoch treba adresu najprv vypočítať a potom použiť nepriame adresovanie.
- Lokálne premenné sú uložené v zásobníku. Každá procedúra tam má svoje parametre, návratovú adresu a lokálne premenné. Tento úsek (jednej procedúry) sa vola aktivačný úsek (stack frame). Procedúra si počas svojho behu môže hádzať nejaké veci do zásobníka, ale aby sa v ňom stále dalo vyznať, existuje EBP. Ten sa počas behu procedúry nemení (zatiaľčo ESP áno), preto sa od EBP dajú stále adresovať lokálne premenné.
- Ešte raz a zrozumiteľnejšie: Ja som aktuálne bežiaci procedúra, nachádzam sa preto na vrchu zásobníka. EBP je nastavené na začiatok môjho stack framu – preto viem, že napr. na EBP - 4 mám 1 vstupnú premennú, na EBP - 5 ďalšiu, atď. A tieto adresy EBP - 4, EBP - 5, ... budú stále rovnaké, lebo EBP sa počas behu procedúry nemení. Keďže používam zásobník, vždy keď do neho niečo vložím/z neho niečo vyberiem ESP sa posunie – preto nemôžem adresovať od ESP, ale od konštantného EBP. Zrozumiteľnejšie? :-)

- **bázovaná adresa**

```
MOV EAX, [EBP+10]
```

- od EBP o 10 vyššie (v stack frame procedúry premenná na 10. mieste)

- **indexovaná a bázovaná adresa**

```
MOV EDX, [EBP+10+ECX*4]
```

- pre adresovanie lokálnych polí (čiže kombinácia indexovanej a bázovanej adresy)

- Nesmie sa urobiť inštrukcia MOV [pamäť], [pamäť] – vždy to musí ísť cez pomocný register
- Operandy musia byť rovnako veľké. MOV EAX, DL je neprípustné.

*Tejto časti ďalej som nepochopil, snád' v slajdoch*

- MOV EBX, [EAX + 5] – do EBX sa priradia 4 bajty

/ doplniť zo slajdov:

```
INC [EDX + 8]
```

```
INC BYTE PTR [EDX + 8]
```

```
INC WORD PTR [EDX + 8]
```

## INC DWORD PTR [EDX + 8]

&lt;/cast ktorej som nepochopil&gt;

- Pri prenášaní polí ako parametrov sa do EAX uloží adresa pola
- To isté pri **var** parametroch – uloží sa len adresa

```
type Pole = array[1..10] of integer;
var a:pole = (2,5,12,-6,90,37,-98,-1,3,12)
```

|                                                                                                                                                                                           |                                                                                                                                                                                                                                                                                                                              |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>function MaxPrvok(a : Pole) : integer; asm     mov edx, [eax]     mov ecx, 9 @1: add eax, 4     cmp epx, [eax]     jge @2     mov edx, [eax] @2: loop @1     moc eax, edx end;</pre> | <p>v EAX bude adresa pola, výsledok sa uloží do EAX</p> <p>edx := a[1]<br/> cyklus bude bežať 9 krát<br/> eax (smerník) posuniem na ďalší prvok<br/> porovnam epx s a[i] (a[i] - aktuálny prvok)<br/> podmienka (greater or equal)<br/> edx := a[i]<br/> koniec cyklu<br/> doterajsie max mam v edx, presuniem ho do eax</p> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|



This work is licensed under a [Creative Commons Attribution-NonCommercial-Share Alike 3.0 License](http://creativecommons.org/licenses/by-nc-sa/3.0/).  
Original source: <http://matfyz.adammuller.sk/opsys>