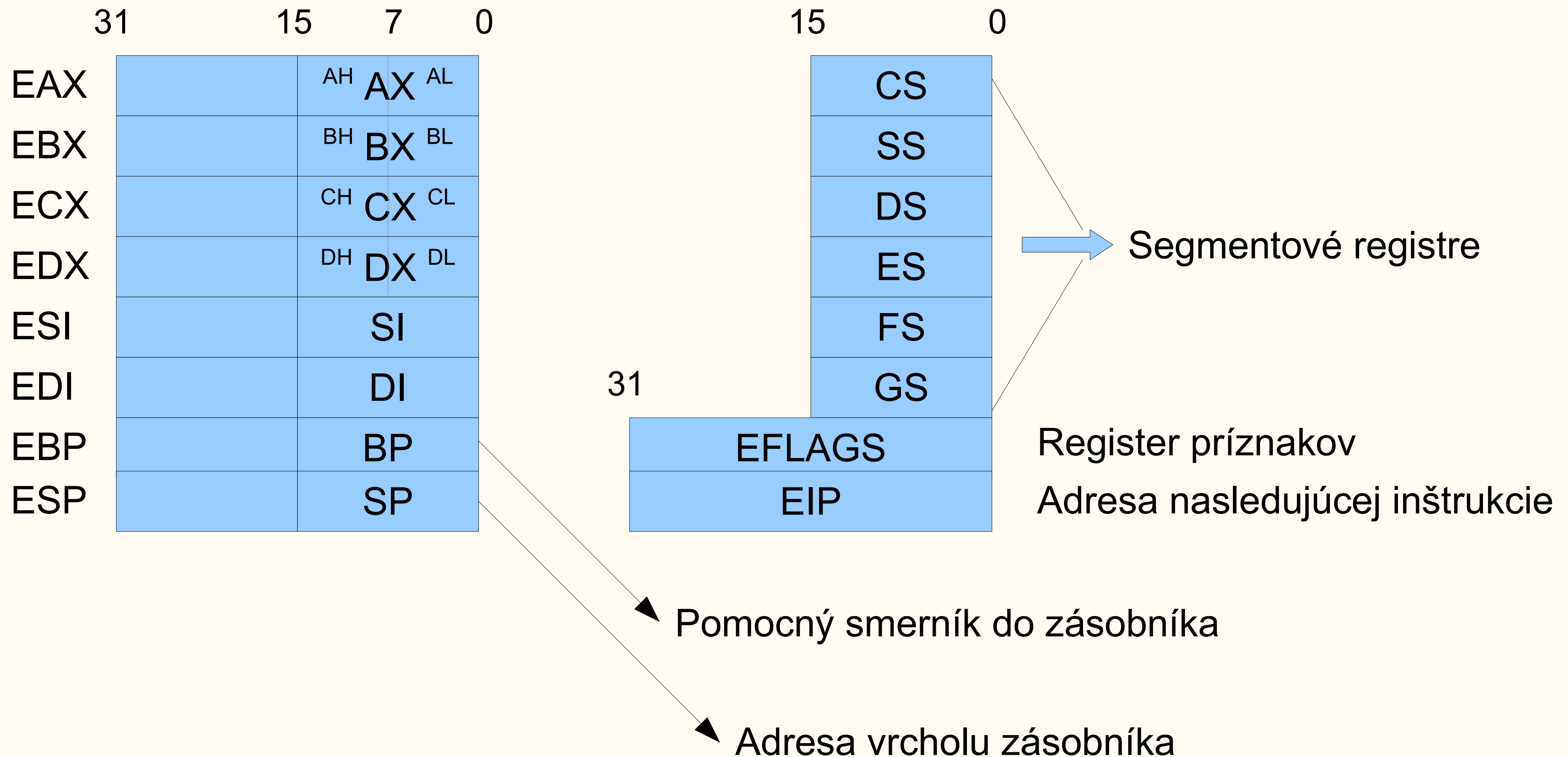
Úroveň strojového kódu procesor Intel® Pentium® Úvod

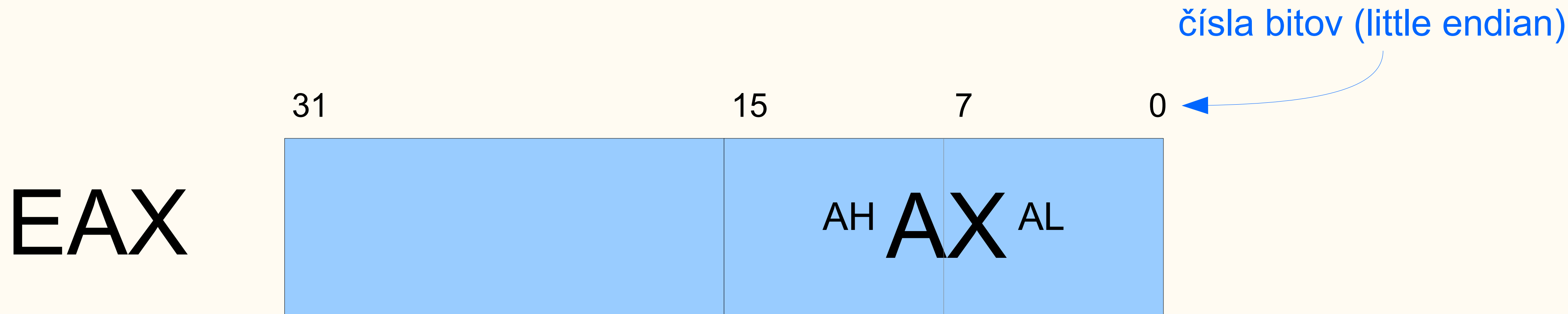
- Štruktúra procesorov Intel® Pentium®
- Základné inštrukcie
- Vetvenia a cykly
- Praktické programovanie jednoduchých assemblerových funkcií

Autor: Peter Tomcsányi,
Niektoré práva vyhradené v zmysle licencie Creative Commons
http://creativecommons.org/licenses/by-nc-sa/3.0/

Štruktúra 32-bitových procesorov Intel Pentium®



32, 16 a 8-bitové registre



- EAX je 32-bitový register
- Jeho spodných 16 bitov sa dá použiť ako 16-bitový register AX
- Spodná polovica AX sa dá použiť ako 8-bitový register AL
- Vrchná polovica AX sa dá použiť ako 8-bitový register AH
- Vždy sú to tie isté bity, takže nemôžeme použiť zároveň EAX aj AX (ale môžeme použiť zároveň AL a AH)
- Podobne pre EBX, ECX a EDX. Ale ESI, EDI, EBP a ESP majú len 16-bitové pod-registre SI, DI, BP a SP

Štruktúra 32-bitových procesorov Intel Pentium®

Register príznakov:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF
0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Niektoré príznaky sú výstupom z ALU - po vykonaní aritmetickej alebo logickej operácie nimi ALU oznamuje niektoré vlastnosti výsledku:

CF - Carry flag je N+1-vý bit N-bitovej aritmetickej operácie, teda ak je 1, tak sa výsledok operácie s číslami bez znamienka "nezmestil" do N bitov, v prípade neznamienkovej operácie to znamená pretečenie. Dá sa nastavovať aj programovo špeciálnymi inštrukciami.

PF - Parity Flag dopĺňa najnižší bajt výsledku do nepárneho počtu jednotiek

AF - Auxiliary Carry Flag obsahuje prenos z dolných štyroch bitov

ZF - Zero Flag je nastavený na 1 práve vtedy ak bol výsledok operácie nula

SF - Sign Flag obsahuje kópiu najvyššieho bitu výsledku, teda je 0 pre nezáporné výsledky a 1 pre záporné výsledky

OF - Overflow Flag je nastavený na 1 práve vtedy ak výsledok operácie s číslami so znamienkom "nezmestil" do N bitov - pretečenie

Iné príznaky umožňujú programátorovi nastaviť alebo zistiť isté nastavenie alebo režim práce procesora

DF - Direction Flag určuje z ktorej strany sa vykonávajú reťazcové inštrukcie

IF - Interrupt Enable Flag určuje, či sú dovolené prerušenia

TF - Trap Flag prikazuje procesoru vykonať za každou inštrukciou ladiace prerušenie

IOPL - Input-Output Privilege Level určuje najnižšiu úroveň oprávnenia pre vykonávanie vstupných a výstupných inštrukcií

Strojový kód
Inštrukcia strojového kódu

- Program v strojovom kóde sa skladá z postupnosti jednoduchých **inštrukcií** (príkazov)
- Typická inštrukcia definuje **operáciu** (čo urobiť) a niekoľko **operandov** (s akými dátami to urobiť)
- Napríklad "prenes do registra EAX číslo 5 "

- Operácie a operandy inštrukcií sú zakódované do čísiel a uložené v pamäti počítača (von Neumann)
- Počítač vieme ľahko "naučiť" napr. že 01 je "prenes" 02 je "pripočítaj" atď. Ale človek by mal problém uvažovať o programe zakódovanom v číslach
- Preto si ľudia zapisujú inštrukcie strojového kódu pomocou ľahšie zapamätateľných skratiek

Strojový kód a assemblerový jazyk

- Jazyk, v ktorom zapisujeme operácie aj operandy inštrukcií pomocou čitateľných skratiek (symbolov), nazývame **assemblerový jazyk** alebo **jazyk symbolických adries**.
- **Assembler** je program, ktorý preloží program napísaný v assemblerovom jazyku (teda v tvare textu) do skutočného strojového kódu (teda do čísiel).
- Ale často pod pojmom **assembler** máme na mysli assemblerový jazyk.

Assembler Intel

- Inštrukcia "prenes do registra EAX číslo 5 " je v **assembleri Intel** zapísaná takto:

- Môžeme napísať aj **mov eax, 5**
- Strojový kód pre túto inštrukciu je: **B8 05 00 00 00**

toto je 5 na 32 bitov v little endian

B8 znamená "mov eax, 32-bitová konštanta"

- Pre strojový kód Intelu existuje aj iný assemblerový jazyk - *assembler AT&T*, v ňom sa to isté zapíše:

V tomto semestri budeme používať len Intel assembler

Operandy

V strojovom kóde Intel poznáme tri druhy operandov:

- **priamy operand** - číslo: MOV EAX,**5**
- **register** - meno registra: MOV **EAX**,5
- **pamäťový operand** - odkaz na adresu v pamäti
  - číslo alebo výraz určujúci adresu v hranatých zátvorkách MOV EAX, **[12800]**
  - alebo aj (keď miešame C a assembler) meno premennej deklarovanej v C: MOV EAX, **X**

Operand môže byť **explicitný** (ako všetky vyššie) alebo **implicitný** - keď nie je priamo napísaný, ale inštrukcia ho používa (napr. MUL CL použije aj AL a AX)

Základné inštrukcie procesora Intel Pentium®

- Presuny údajov

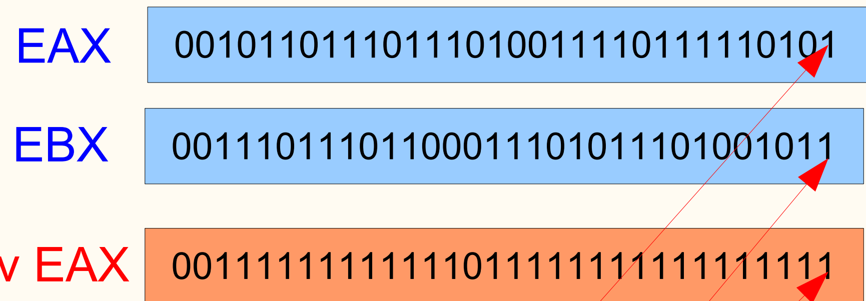
MOV EAX,EBX ← EAX = EBX
MOV EAX, 5
~~MOV EBX,AL~~ ← Nerovnaké dĺžky
MOVSX EBX,AL
MOVZX EBX,AL
XCHG AL,AH

- Aritmetické inštrukcie

ADD EAX,EBX ← EAX = EAX+EBX
SUB BH,6
ADC EAX,EBX ← EAX = EAX+EBX+ CF
MUL CL ← AX = AL*CL (neznamienkovo)
IMUL CL ← AX = AL*CL (znamienkovo)
CMP EBX,5 ← Vypočíta EBX-5 ale výsledok nikam neuloží
Väčšinou po nej nasleduje podmienený skok

- Logické inštrukcie

OR EAX,EBX



AND AX,15 ← Nechá len 4 najspodnejšie bity

TEST AL,7 ← Vypočíta AL AND 7 ale výsledok nikam neuloží
Väčšinou po nej nasleduje podmienený skok

Prvý príklad – lineárny výpočet

Naprogramujte funkciu, ktorá na vstupe dostane dve 32-bitové celé čísla so znamienkom A a B a jej výsledkom bude hodnota výrazu 2*A+4*B

```
int32_t pocitaj(int32_t a, int32_t b)
{
   __asm
   {
      mov eax, a
      add eax, b
      add eax, b
      add eax, eax
   }
}
```

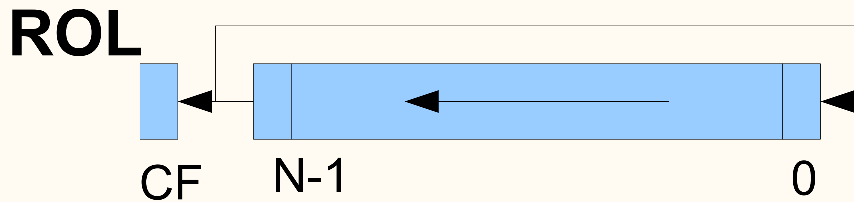
int32_t je 32-bitové celé číslo so znamienkom

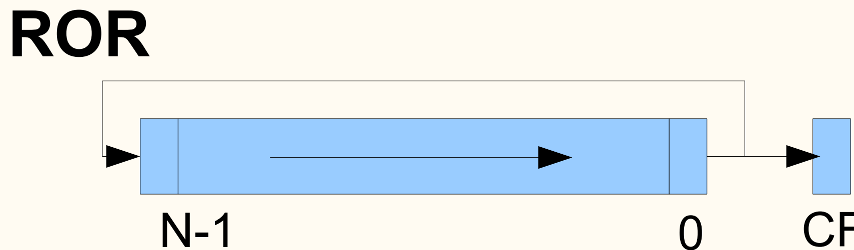
ako pamäťové operandy môžeme používať identifikátory premenných (detaily sa dozvieme neskôr)

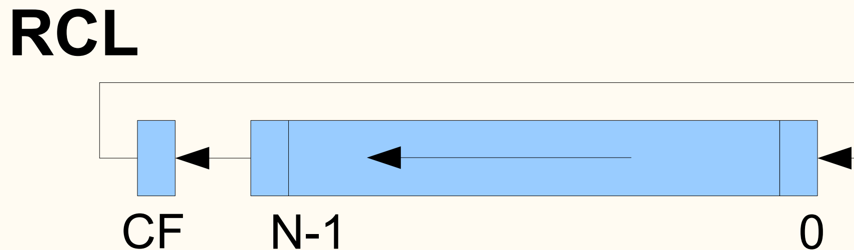
to, čo je teraz v EAX, bude výsledkom funkcie

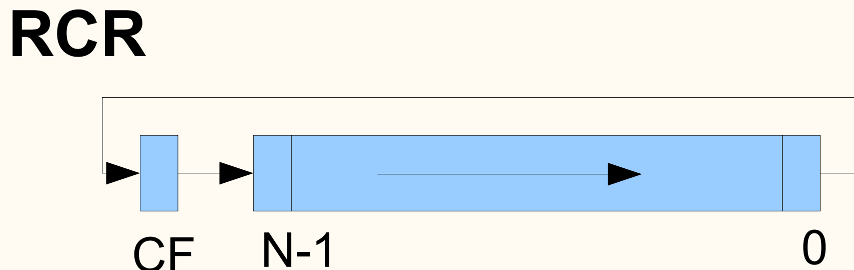
Pre programovanie v strojovom kóde je typické, že sa pri jednoduchom aritmetickom výpočte snažíme vyhnúť inštrukciám násobenia lebo ich použitie spotrebuje veľa registrov a zneprehľadní kód.

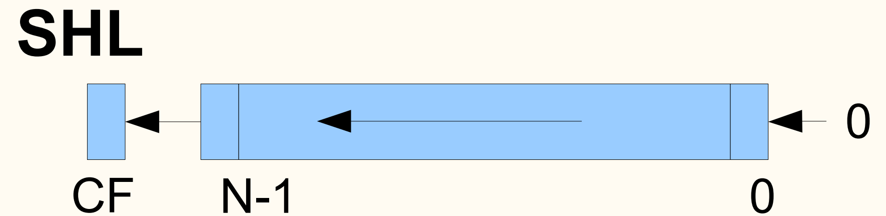
Rotácie a posuny

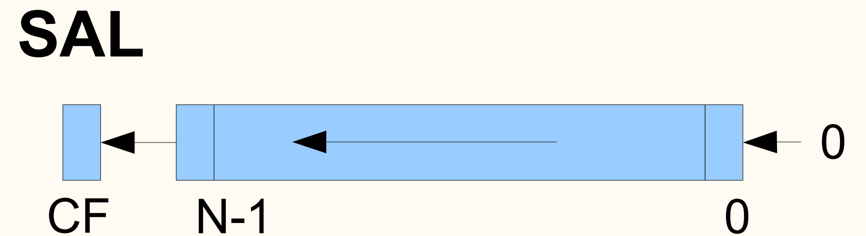


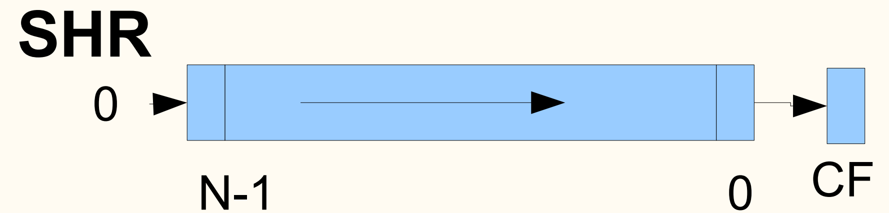


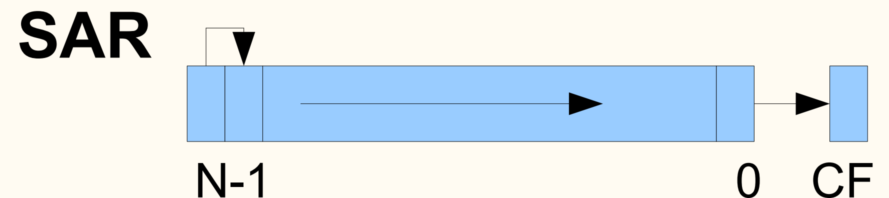












(bit N-1 zostáva na mieste aj sa skopíruje do bitu N-2)

Všetky tieto inštrukcie majú tvar
KÓD_INŠTRUKCIE cieľ, imm8/CL
Prvý operand je register alebo pamäťové miesto, ktoré sa rotuje alebo posúva.
Druhý operand určuje o koľko bitov sa rotuje alebo posúva a je priamy (teda číslo napr. 1 alebo 2 alebo 7) alebo register CL (počet bitov je daný obsahom registra CL).

Kontrolná otázka

Čo vypočíta tento program?

```
int32_t pocitaj(int32_t a, int32_t b)
{
   __asm
   {
      mov eax, a
      shl eax, 3
      mov ebx, b
      add ebx, ebx
      add eax, ebx
   }
}
```

Porovnania a skoky

- V assembleri neexistujú štruktúrované príkazy ako **if**, **while** alebo **for**
- Vetvenie programu sa implementuje pomocou porovnaní a skokov

```
int32_t vacsie(int32_t a, int32_t b)
{
   __asm
   {
      mov eax, a
      cmp eax, b
      jge a1
      mov eax, b
   a1:
   }
}
```

Funkcia vráti väčšie z dvoch 32-bitových čísiel so znamienkom

Porovnaj eax a b (výsledok porovnania je v eflags)

Ak bolo "znamienkovo väčšie", tak skoč na miesto a1

Sem príde len ak predošlý skok neskočil

Tu je miesto a1

Všetky inštrukcie, ktoré budeme používať nájdete tu

Kontrolná otázka

Čo treba doplniť na vybodkovaný riadok, bolo výsledkom funkcie neznamienkovo menšie číslo?

```
uint32_t mensie(uint32_t a, uint32_t b)
{
   __asm
   {
      mov eax, a
      cmp eax, b
      ...................
      mov eax, b
   a1:
   }
}
```

Vetvenie programu

Namiesto štrukturovaných príkazov C (alebo Pythonu) sa program v strojovom kóde vetví pomocou podmienených skokov, ktoré typicky (ale nie vždy) nasledujú po operácii porovnania.

```
int32_t x;
if (x == 5)
   { príkazy }
```

```
   MOV EAX,x
   CMP EAX,5
   JNE a1
   { príkazy }
a1:
```

porovnanie

podmienený skok je "opačný" než céčkový operátor lebo hovorí: "ak sa EAX nerovná 5 tak preskoč <príkaz1>"

"znamienkovo väčší" lebo x aj y sú znamienkové typy

```
int32_t x, y;
if (x > y)
   { príkazy1 }
else
   { príkazy2 }
```

```
   MOV EDX,x
   CMP EDX,y
   JLE a1
   { príkazy1 }
   JMP a2
a1:{ príkazy2 }
a2:
```

nepodmienený skok

Skoč ak je znamienkovo menší alebo rovný - zase "opačný skok"

Pri **nerovnostiach** musíme vedieť, či sú čísla chápané znamienkovo (v dvojkovom doplnku) alebo neznamienkovo.
Kompilátory jazykov ako Pascal alebo C to zistia z deklarácie premenných (viď Celočíselné typy v C), v strojovom kóde na to ale musí myslieť programátor.

Jednoduché cykly

```
uint8_t a, b;
while (a < b)
   { príkazy }
```

```
a2:MOV CL, a
   CMP CL, b
   JAE a1
   { príkazy }
   JMP a2
a1:
```

```
int16_t s;
do
   { príkazy }
while (s <= 18);
```

```
a1:{ príkazy }
   MOV BX, s
   CMP BX, 18
   JLE a1
```

Používame "opačné" skoky
Pri nerovnostiach záleží, či ide o čísla so znamienkom alebo bez znamienka.
Tentokrát je a aj b bezznamienkového typu, preto použijeme neznamienkový skok JAE.

Tu jediný je (jediný raz) porovnanie rovnaké ako skok v strojovom kóde.
Je znamienkový lebo s je **int16_t** je znamienkový typ.

Jednoduché cykly typu **for**

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

```
   MOV ECX,10
a1:{ príkazy }
   LOOP a1
```

Cyklus typu „opakuj N-krát“.
Tento prístup funguje len ak je počet opakovaní nenulový.

LOOP a1 znamená:
DEC ECX
JNZ a1
Teda zníži ECX o jednu a ak výsledok nie je nula, tak skočí na a1

```
uint32_t n;
for (i=0; i < n; i++)
  { príkazy }
```

```
   MOV ECX,n
   JECXZ a2
a1:{ príkazy }
   LOOP @1
a2:
```

Vo všeobecnosti (napr. keď je počet opakovaní výsledok predošlého výpočtu), musíme pridať test na nulu

Ak môže byť výsledok predošlého výpočtu aj záporný, tak musí byť kód ešte zložitejší

Pre všeobecný tvar cyklu for alebo ak vo vnútri cyklu potrebujeme premennú i sa musí cyklus for rozpísať na while.

Druhý príklad – cykly a vetvenia skúsme to najprv v C

Naprogramujte funkciu:

```
uint8_t pocet1c(uint32_t x)
```

Jej výsledkom je počet jedničiek v dvojkovom zápise čísla x.

Riešenie z Cvičení 1:

```
uint8_t pocet1c(uint32_t x)
{
  uint8_t res = 0;
  while (x)
  {
    res += x & 1;
    x = x >> 1;
  }
  return res;
}
```

Druhý príklad – cykly a vetvenia
Prepíšme do assembleru

Naprogramujte assemblerovú funkciu:

```
uint8_t pocet1a(uint32_t x)
```

Jej výsledkom je počet jedničiek v dvojkovom zápise čísla x.

```
uint8_t pocet1a(uint32_t x)
{ __asm {
    mov ebx, x
    mov al,0      // res = 0;
a1: cmp ebx,0     // while (x != 0) {
    je  a2
    mov cl,bl
    and cl,1
    add al,cl     //     res += x & 1;
    shr ebx,1     //     x = x >> 1;
    jmp a1        // }
a2:               // return res;
}}
```

Druhý príklad – cykly a vetvenia
Vylepšime

Naprogramujte assemblerovú funkciu:

```
uint8_t pocet1a2(uint32_t x)
```

Jej výsledkom je počet jedničiek v dvojkovom zápise čísla x.

```
uint8_t pocet1a2(uint32_t x)
{ __asm {
    mov ebx,x
    mov al,0    // res = 0;
                // do {
a1: shr ebx,1   //   CF = x & 1; x = x >> 1;
    adc al,0    //   res += CF;
    cmp ebx,0   // } while (x != 0);
    jne a1      //
                // return res;
}}
```

Tretí príklad

Naprogramujte assemblerovú funkciu:

```
uint8_t obratene_cislo(uint8_t x)
```

Jej výsledkom je číslo, ktoré vznikne obrátením zápisu x v dvojkovej sústave.

```
uint8_t obratene_cislo(uint8_t x)
{ __asm {
                // res = 0; // v asm netreba (prečo?)
    mov bl,x
    mov ecx,8   // for (i = 0; i < 8; i++) {
a1: shr bl,1    //   CF = x & 1; x = x >> 1;
    rcl al,1    //   res = (res << 1) + CF;
    loop a1     // }
                // return res;
}}
```

Štvrtý príklad

Naprogramujte assemblerovú funkciu:

bool je_palindrom(**uint32_t** x)

Jej výsledkom je true, práve vtedy ak je zápis vstupnej hodnoty x v dvojkovej sústave na 32 bitov palindróm

bool je zobrazený v jednom bajte, 0 je **false** a 1 je **true**. Je určite k dispozícii v nových verziách C++. V C možno treba použiť #include <stdbool.h>

```
bool je_palindrom(uint32_t x)
{ __asm {
    mov ebx,x   // Otočíme x podľa 3. príkladu
    mov ecx,32  //   ale teraz je x 32-bitové
a1: ror ebx,1   // ror namiesto shr zachová x
    rcl eax,1   //   v eax (prečo ?)
    loop a1
    cmp eax,ebx   // if (otocene_x == x)
    jne a2
    mov al,1    //   return true;
    jmp a3      // else
a2: mov al,0    //   return false ;
a3:
}}
```

Štvrtý príklad - Vylepšime

Naprogramujte assemblerovú funkciu:
`bool je_palindrom(uint32_t x)`
Jej výsledkom je true, práve vtedy ak je zápis vstupnej hodnoty x v dvojkovej sústave na 32 bitov palindróm.

```
bool je_palindrom(uint32_t x)
{ __asm {
    mov ebx,x  // Otočíme x podľa 3. príkladu
    mov ecx,32 //   ale teraz je x 32-bitové
a1: ror ebx,1  // ror namiesto shr zachová x
    rcl eax,1  //   v eax (prečo ?)
    loop a1
    cmp eax,ebx // return otocene_x == x;
    sete al
}}
```

Inštrukcia **sete al** dá do al 1 práve vtedy ak by skok **je** skočil (teda ak **cmp** nastavil EFLAGS tak, že sa jeho operandy rovnajú).
Inak povedané: **sete** prevedie výsledok porovnania z bitov EFLAGS do hodnoty typu bool