Úroveň strojového kódu procesor Intel® Pentium® Zásobník a konvencie volania

- Práca so zásobníkom
- Prenos parametrov do funkcie – konvencia cdecl
- Aktivačný záznam procedúry
- Volanie služby Windows - konvencia stdcall
- Konvencia fastcall
- Praktické programovanie assemblerových procedúr

Autor: Peter Tomcsányi,

Práca so zásobníkom

- Zásobník je údajová štruktúra LIFO - Last In, First Out
- Má definované operácie PUSH (pridaj do zásobníka) a POP (vyber zo zásobníka)
- Zásobník je vhodná dátová štruktúra pre niektoré typy algoritmov

- **Načo je zásobník v strojovom kóde?**
  - Na ukladanie návratových adries podprogramov
  - Na ukladanie lokálnych premenných
  - Na ukladanie medzivýsledkov aritmetických výpočtov

- **Implementácia zásobníka v procesoroch Intel Pentium**
  - Je uložený v časti pamäti
  - Adresa jeho vrcholu je uložená v registri ESP
  - Rastie smerom do nižších adries

Explicitné použitie zásobníka

Programátor môže používať zásobník na uloženie akýchkoľvek údajov:

- Ukladanie medzivýsledkov pri výpočte zložitých výrazov keď nie je dosť registrov
- Uchovanie registrov keď ich dočasne treba na niečo iné

PUSH EAX ← Ulož EAX do zásobníka

POP EBX ← Vyber EBX zo zásobníka

PUSHFD ← Ulož EFLAGS do zásobníka

POPFD ← Vyber EFLAGS do zásobníka

Do zásobníka sa ukladajú a z neho vyberajú vždy len 32-bitové hodnoty (štyri bajty)

Príklad – explicitné použitie zásobníka

najprv v C

Naprogramujte funkciu:
`void str_c(unsigned long x, char result[])`
ktorá prevedie číslo x do znakovej reprezentácie v poli **result**.
Teda pre vstup x=289 bude po zavolaní funkcie v poli **result** uložený reťazec "289".

```
void str_c(unsigned long x,char result[])
{
  char stack[11]; // long ma max. 10 cifier
  int i;
  char *p;

  // uloz zvysky po deleni 10 do stack
  i = 0;
  do {
    stack[i] = x % 10;
    i++;
    x = x / 10;
  } while (x != 0);

  // prepis zo stack do result
  p = result;
  do {
    i--;
    *p = stack[i] + '0'; // plus kod nuly
    p++;
  } while (i > 0);
  *p = 0;  // na konci znak s kodom 0
}
```

Kratší zápis: `stack[i++] = x % 10;`

Kratší zápis:
`*p++ = stack[--i] + '0';`

Príklad – explicitné použitie zásobníka

v assembleri

Naprogramujte assemblerovú funkciu:

```
void str(uint32_t x,
char result[])
```

ktorá prevedie číslo x do znakovej reprezentácie v poli **result**.

```
__asm {
    mov eax, x
    mov ebx, 10
    mov ecx, 0
a1: mov edx, 0
    div ebx
    push edx // do zasobnika
    inc ecx
    cmp eax, 0
    jne a1

    mov ebx, result
a2: pop eax  // zo zasobnika
    add eax, '0'
    mov [ebx], al
    inc ebx
    loop a2
    mov [ebx], 0
}
```

inštrukcia **div** s 32-bitovým operandom vydelí spojené registre EDX:EAX operandom a uloží podiel do EAX a zvyšok do EDX

Implicitné použitie zásobníka

```
Pocitaj:
   ADD ECX,ECX
   ADD ECX,10
   MOV EAX,ECX
   RET
F:
   MOV ECX,10
   CALL Pocitaj
   MOV EBX,EAX
   MOV ECX,20
   CALL Pocitaj
   ADD EAX,EBX
   RET
```

```
int pocitaj(int x) {
   return 2*x + 10;
}

int f (){
   return pocitaj(10)+pocitaj(20);
}
```

Volanie podprogramu - do zásobníka sa uloží obsah EIP a do EIP sa uloží adresa podprogramu.

Návrat z podprogramu - EIP sa vyberie zo zásobníka

Program v assembleri robí to isté, čo program v C, ale skutočný preklad z jazyka C by bol iný (vysvetlíme neskôr)

Narábanie s bitmi registra EFLAGS

- Niektoré bity registra EFALGS sa dajú meniť špeciálnymi inštrukciami:
- Je to napríklad bit CF:

  STC - Nastav CF na 1

  CLC - Nastav CF na 0

  CMC - Neguj CF
- alebo bit IF

  STI - Nastav IF na 1 (teda povoľ prerušenia)

  CLI - Nastav IF na 0 (teda zakáž prerušenia)

Adresovanie pamäte

Zhrnutie

- Priama adresa

  MOV EAX,[12840]

- Nepriama adresa

  MOV EDX,[EBX]

- Indexovaná adresa

  MOV [1244+EAX*4],ECX

- Bázovaná adresa

  MOV EAX,[EBP+16]

- Indexovaná a bázovaná adresa (najzložitejší možný prípad)

  MOV EDX,[EBP+12+ECX*4]

Keďže adresy sú 32-bitové, v adresovaní **musíme** vždy použiť 32-bitové registre

Jeden register (EAX, EBX, ECX,EDX, ESI, EDI, ESP alebo EBP)

Konštanta (kladná alebo záporná)

Druhý register

Násobiaci faktor (len 1, 2, 4 alebo 8)

Vnorené cykly

Cyklus s iným registrom, určenie dĺžky operandu

Naprogramujte assemblerovú funkciu:
`void pocty(int32_t x[], uint32_t n, int32_t prvky[], uint32_t m, uint32_t vysledok[])`
ktorá dostane pole **x** dĺžky **n** a pole **prvky** dĺžky **m** a výsledkom jej práce je pole **vysledok**, pre ktorý bude platiť: pre každé i, 0<=i<m: vysledok[i] je počet výskytov čísla prvky[i] v poli x.

Nulovanie prvku poľa vysledok
`mov [ebx], 0`

Pripočítanie 1 k prvku poľa vysledok
`inc [ebx]`

Vyššie uvedené by bolo logicky správne (a tým ušetríme register pre počítadlo), ale **nefungovalo by to správne**, lebo inštrukcie by pracovali s operandom dĺžky 1 bajt, pritom majú pracovať s operandom dĺžky 4 bajty.
Preto musíme určiť dĺžku operandu:
`mov dword ptr [ebx], 0`
`inc dword ptr [ebx]`
(tak, ako je to v programe)

```
__asm {
    mov eax,prvky
    mov ebx,vysledok
    mov ecx,m
    jecxz a4
a1:
    mov dword ptr [ebx],0
    mov edx, x
    mov esi, n
    cmp esi, 0
    je a5
a2:
    mov edi, [edx]
    cmp edi, [eax]
    jne a3
    inc dword ptr [ebx]
a3: add edx, 4
    dec esi
    jnz a2
a5: add eax,4
    add ebx,4
    loop a1
a4:
}
```

Nemáme dosť registrov, preto budeme počítať počet výskytov priamo v prvkoch poľa vysledok

Vonkajší cyklus prechádza polia prvky a vysledok, používa ecx a inštrukciu loop

Vnútorný cyklus prechádza pole x, používa esi. Preto nemôže použiť loop, ale musí použit dec a jnz

Nemôžeme napísať `cmp [edx],[eax]` lebo inštrukcia by mala oba operandy v pamäti, čo nie je dovolené.

Vnorené cykly (úloha z cvičenia 5)

Použitie zásobníka

Naprogramujte assemblerovú funkciu: `void pocty(long x[], long n, long prvky[], long m, long vysledok[])` ktorá dostane pole x dĺžky n a pole prvky dĺžky m a výsledkom jej práce je pole vysledok, pre ktorý bude platiť: pre každé i, 0<=i<m: vysledok[i] je počet výskytov čísla prvky[i] v poli x.

Nulovanie počítadla v esi

Pripočítanie 1 k počítadlu

Uloženie počítadla do poľa vysledok

Keďže sme ušetrili register esi, môžeme mať počítadlo v ňom (ale nemusíme, na počítanie môžeme použiť aj spôsob z predošlej strany).

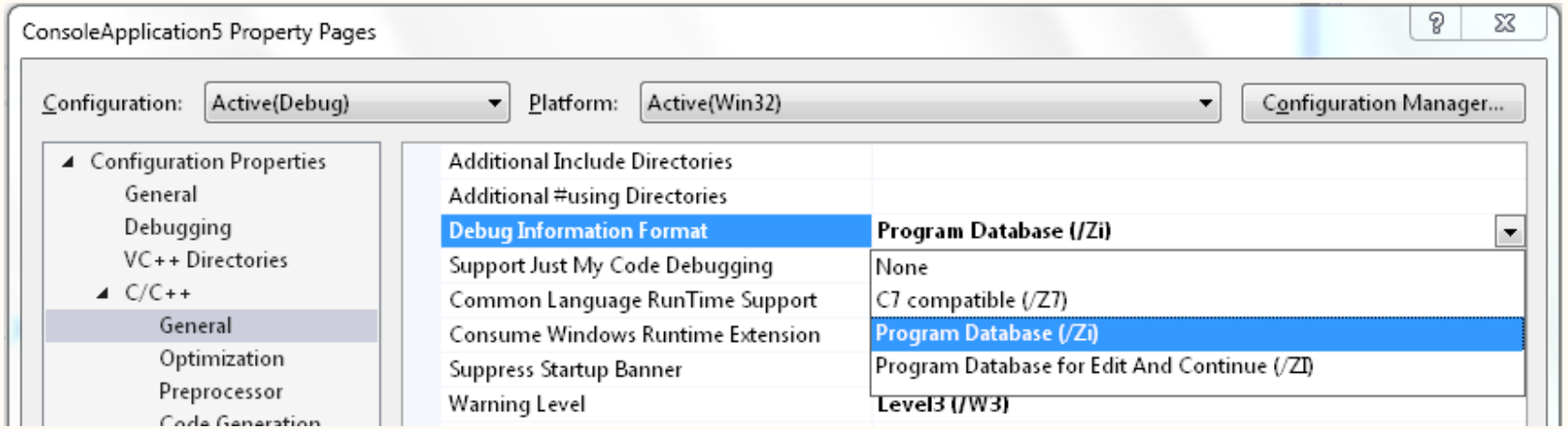
```
__asm {
    mov eax,prvky
    mov ebx,vysledok

    mov ecx,m
    jecxz a4
a1: mov esi,0
    mov edx, x

    push ecx

    mov ecx, n
    jecxz a5
a2: mov edi, [edx]
    cmp edi, [eax]
    jne a3
    inc esi
a3: add edx, 4
    loop a2

a5: mov [ebx],esi

    pop ecx

    add eax,4
    add ebx,4
    loop a1

a4:
}
```

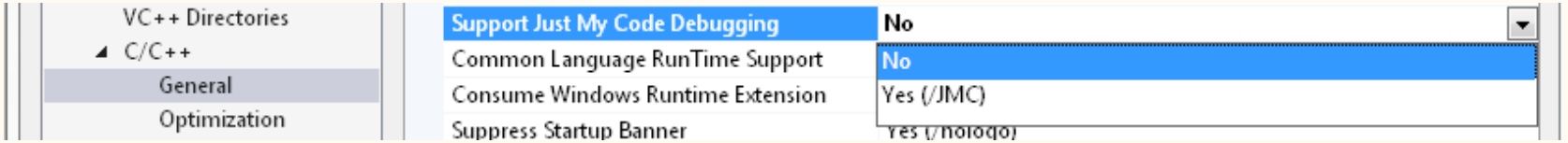
Vonkajší cyklus prechádza polia `prvky` a `vysledok`, používa ecx a inštrukciu `loop`

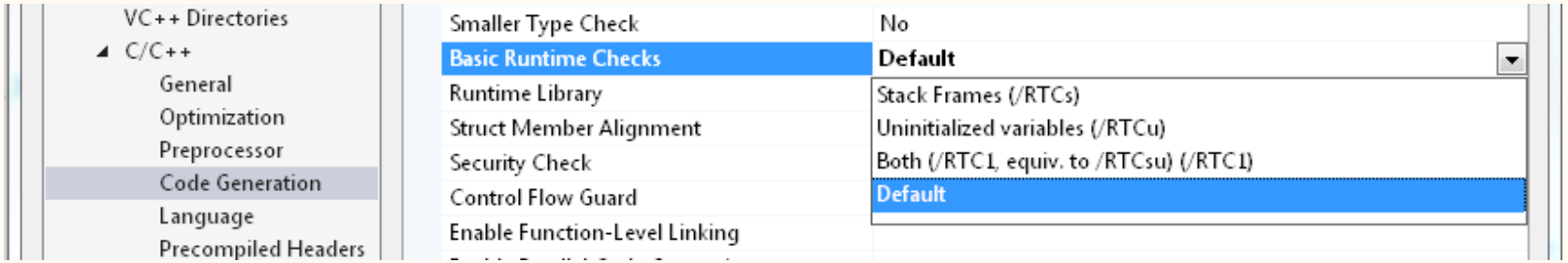
Vnútorný cyklus odloží ecx vonkajšieho cyklu na zásobník, preto môže tiež použiť ecx a inštrukciu `loop`

Nastavenie Vlastností projektu pre ďalšie ukážky

V menu **Project** otvorte poslednú položku (***<meno projektu>*** **Properties**) a zmeňte tri nastavenia podľa obrázkov. Nastavenia zakážu kompilátoru generovať časť kódu, ktorá je potrebná len pre ladenie a vďaka tomu môžeme lepšie študovať vygenerovaný kód.

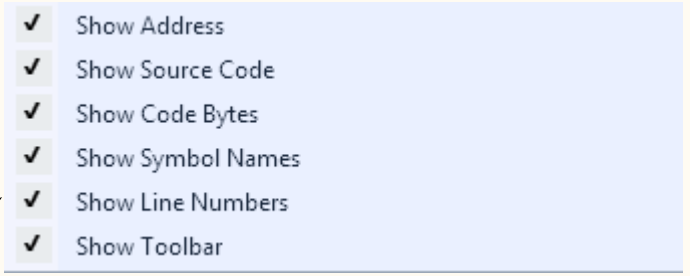






Parametre funkcií

- Zoberme Siedmy príklad z minulej prednášky, breakpoint na prvú inštrukciu, Run (F5), po zastavení na breakpointe zvoliť Debug/Windows/Disassembly.



- V záložke Disassembly pravý klik, zaškrtnúť *Show line numbers, Show symbol names* a *Show Code Bytes*

riadky začínajúce číslom riadku a dvojbodkou zobrazujú zdrojový program

```
     7:    __asm {
     8:        mov ebx, a
00CF3114 8B 5D 08 mov ebx,dword ptr [a]
     9:        mov edx, x
00CF3117 8B 55 10 mov edx,dword ptr [x]
```

explicitné určenie dĺžky operandu (byte, word, dword)

disassembler nám stále ukazuje mená parametrov

8B 55 10 je strojový kód pre inštrukciu mov edx, x (3 bajty)

riadky začínajúce adresou obsahujú strojový kód ako bajty v šestnástkovej sústave a aj v assembleri (disassemblovaný)

- V záložke Disassembly pravý klik, odškrtnúť *Show symbol names*:

```
     7:    __asm {
     8:        mov ebx, a
00CF3114 8B 5D 08 mov ebx,dword ptr [ebp+8]
     9:        mov edx, x
00CF3117 8B 55 10 mov edx,dword ptr [ebp+10h]
```

teraz vidíme skutočnú adresu – bázované adresovanie registrom EBP.

vysvetlenie nasleduje na ďalších stranách

Bázovaná adresa

Jednoduchá lokálna premenná alebo parameter

Parametre aj lokálne premenné sú uložené v zásobníku. Register EBP pomáha pri adresovaní lokálnych premenných.

```
void p(int32_t i)
   int32_t j;
   ...
   j = i + 1;
   ...
}

int main()
{
   p(12);
}
```

```
PUSH EBP
MOV EBP,ESP
SUB ESP,4
```

2. Volaný uloží EBP, nastaví EBP a urobí miesto pre lok. premenné. Tým dobuduje svoj aktivačný záznam

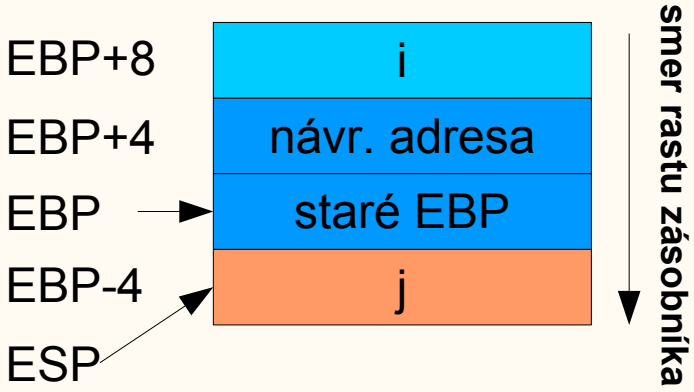
```
MOV EAX,[EBP+8]
ADD EAX,1
MOV [EBP-4],EAX
```

```
MOV ESP,EBP
POP EBP
RET
```

```
PUSH 12
CALL p
ADD ESP, 4
```

1. Volajúci uloží do zásobníka paremeter (**PUSH 12**) a návratovú adresu (**CALL p**).
Tým sa vytvorí časť aktivačného záznamu pre **p**.

Aktivačný záznam (**Stack frame**) je úsek zásobníku, ktorý obsahuje informácie jedného vyvolania funkcie.
Aktivačný záznam našej funkcie **p**:



3. Parametre a lokálne premenné v zásobníku sa adresujú relatívne k registru EBP, nazývame to **bázovaná adresa**

4. Volaný odstráni tú časť aktivačného záznamu, ktorú vytvoril: Zníži zásobník, vyberie staré EBP (na vrch zásobníka sa dostane návratová adresa) a vykoná návrat (RET).

5. Volajúci odstráni zo zásobníka parameter a tým je odstránený celý aktivačný záznam funkcie **p**.

Čo naozaj vygeneruje kompilátor

```
     7: void p(long i) {
00E713A0 55                   push        ebp
00E713A1 8B EC                mov         ebp,esp
00E713A3 51                   push        ecx
     8: long j;
     9: j = i + 1;
00E713A4 8B 45 08             mov         eax,dword ptr [ebp+8]
00E713A7 83 C0 01             add         eax,1
00E713AA 89 45 FC             mov         dword ptr [ebp-4],eax
    10: }
00E713AD 8B E5                mov         esp,ebp
00E713AF 5D                   pop         ebp
00E713B0 C3                   ret
--- No source file -----------------------
00E713B1 CC                   int         3
....
00E713BF CC                   int         3
--- d:\prednasky\2019\leto\...\consoleapplication5.cpp
    11:
    12: int main()
    13: {
00E713C0 55                   push        ebp
00E713C1 8B EC                mov         ebp,esp
    14: p(12);
00E713C3 6A 0C                push        0Ch
00E713C5 E8 0D FD FF FF       call        00E710D7
00E713CA 83 C4 04             add         esp,4
    15: return 0;
00E713CD 33 C0                xor         eax,eax
    16: }
00E713CF 5D                   pop         ebp
00E713D0 C3                   ret
```

Namiesto SUB ESP,4 vygeneroval PUSH ECX, tým sa ESP tiež zníži o 4

Adresy 00E713B1 až 00E713BF sú nepoužité (asi aby funkcie začínali na adrese deliteľnej 16), sú vyplnené inštrukciou INT 3, ladiace prerušenie, aby debugger vedel reagovať ak tam program chybne skočí

Aj **main** je funkcia, teda má vstupný kód. Keďže nemá lokálne premenné neznižuje sa ESP a vstupný kód má len 2 inštrukcie

CALL volá funkciu p, tá je ale na adrese 00E713A0, na adrese 00E710D7 však kompilátor vygeneroval pomocný skok JMP 00E713A0

Uloží do EAX nulu - return 0

Keďže **main** nemá lokálne premenné, výstupný kód nepotrebuje inštrukciu MOV ESP,EBP

Pri každej kompilácii môže kompilátor uložiť program na iné adresy, ale ich obsah bude takýto (ak ste nastavili kompilátor podľa inštrukcií na predošlých slajdoch).

Bázovaná a indexovaná adresa (2)

Prvky lokálnych polí

Ak je lokálna premenná pole, môžeme pri jej indexovaní použiť **bázované a indexované adresovanie** s registrom EBP ako bázou.

```
void p2(int32_t i, int32_t j) {
   int32_t a[2];
   ...
   a[i] = j+1;
   ...
}

int main()
{
   p2(1,89);
}
```

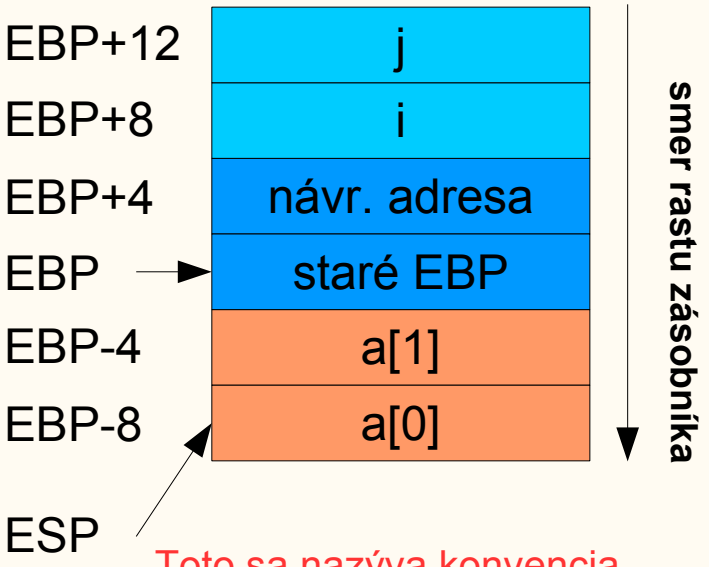
```
PUSH EBP
MOV EBP,ESP
SUB ESP,8
```

```
MOV EAX,[EBP+12]
ADD EAX,1
MOV ECX,[EBP+8]
MOV [EBP-8+ECX*4],EAX
```

```
MOV ESP,EBP
POP EBP
RET
```

```
PUSH 89
PUSH 1
CALL P2
ADD ESP,8
```

Stack frame funkcie **p2**:



4 je sizeof long

EPB-8 je adresa začiatku poľa **a**

Toto sa nazýva konvencia volania (**calling convention**) **cdecl** a používa ho prevažná väčšina kompilátorov jazyka C.

Ale existujú aj iné konvencie volania...

parametre sa dávajú do zásobníka **v opačnom poradí** než sú zapísané vo volaní

odstránenie parametrov zo zásobníka.

Čo naozaj vygeneruje kompilátor

```
    89: void p2(long i, long j) {
011231E0 55                   push        ebp
011231E1 8B EC                mov         ebp,esp
011231E3 83 EC 08             sub         esp,8
    90: long a[2];
    91: a[i] = j + 1;
011231E6 8B 45 0C             mov         eax,dword ptr [ebp+0Ch]
011231E9 83 C0 01             add         eax,1
011231EC 8B 4D 08             mov         ecx,dword ptr [ebp+8]
011231EF 89 44 8D F8          mov         dword ptr [ebp+ecx*4-8],eax
    92: }
011231F3 8B E5                mov         esp,ebp
011231F5 5D                   pop         ebp
011231F6 C3                   ret
...
    93:
    94: int main()
    95: {
01123200 55                   push        ebp
01123201 8B EC                mov         ebp,esp
    96: p2(1, 89);
01123203 6A 59                push        59h
01123205 6A 01                push        1
01123207 E8 C5 DF FF FF       call        011211D1
0112320C 83 C4 08             add         esp,8
    97: }
0112320F 33 C0                xor         eax,eax
01123211 5D                   pop         ebp
01123212 C3                   ret
```

Úloha z cvičenia
(trochu zmenená)

Naprogramujte assemblerovú funkciu:
`uint8_t ntybitx(uint8_t n, uint32_t x)`
Jej výsledkom je hodnota n-tého bitu čísla x v zmysle číslovania bitov podľa mocnín dvojky, ktorú daný bit zastupuje.
Výsledkom je 0 alebo 1.
Môžete predpokladať, že nedostanete nesprávny vstup, teda, že n<=31

```
uint8_t ntybitx(uint8_t n, uint32_t x)
{
    __asm {
        mov eax,x
        mov cl,n
        shr eax,cl
        and eax,1
    }
}
```

Vyvolanie funkcie z assembleru

Naprogramujte assemblerovú funkciu:

```
void vyber_bity(uint32_t vstupy[], uint8_t vystupy[],
                uint8_t bit, uint32_t n)
```

Pre všetky hodnoty v **n**-prvkovom poli vstupy vyvolá funkciu **ntybitx(bit,vstupy[i])** a výsledok priradí do **vystupy[i]**.

```
__asm {
    mov esi,vstupy
    mov edi,vystupy
    mov ecx,n
a1: push ecx     // uschovanie registrov pred volanim
    push esi     // pre istotu uchovame vsetky pouzivane registre
    push edi
    push [esi+ecx*4-4]  // druhy parameter funkcie
    push bit            // prvy parameter funkcie
    call ntybitx        // volanie funkcie
    add esp,8           // odstranenie parametrov
    pop edi             // obnovenie uschovanych registrov
    pop esi
    pop ecx
    mov [edi+ecx-1],al  // zapisanie vysledku do pola vystupy
    loop a1
}
```

Volanie služby Windows

Výpis na konzolu (do čierneho okna)

- Potrebujeme volať dve služby Windows: **GetStdHandle** a **WriteConsoleA** a jednu štandardnú funkciu C: **strlen**.
- Služby Windows používajú inú konvenciu volania, **stdcall**: parametre sa dávajú do zásobníka rovnako, ako v konvencii **cdecl**, ale o odstránenie parametrov sa stará volaná funkcia
- Funkcia **strlen** používa bežnú konvenciu volania **cdecl**

```
void hello_windows(const char s[]) {
   HANDLE h = GetStdHandle(STD_OUTPUT_HANDLE);
   int len = strlen(s);
   WriteConsoleA(h, s, len, NULL, NULL);
}
```

Výpis na konzolu Windows

v assembleri

Naprogramujte assemblerovú funkciu:
`void hello_windows_a(const char s[])`
Ktorá vypíše text s na konzolu Windows (teda do "čiernej obrazovky")

```
__asm {
    push -11
    call GetStdHandle  // h = GetStdHandle(STD_OUTPUT_HANDLE)

    push eax           // v eax je h, pred volanim strlen ho uchovame

    push s
    call strlen        // eax = strlen(sprava)
    add esp,4

    pop ebx            // ebx = h

    push 0
    push 0
    push eax
    push s
    push ebx
    call WriteConsoleA // WriteConsoleA(h,s,strlen(s),NULL,NULL)
}
```

po volaní konvenciou **cdecl** musíme odstrániť parametre zo zásobníka.

po volaniach konvenciou **stdcall**, neodstraňujeme parametre, teda nemeníme ESP.

Konvencia volania _fastcall

- Na prenos prvých dvoch parametrov použije registre ECX a EDX
- Ostatné parametre sa prenesú ako pri cdecl

```
int _fastcall pocitaj(int x) {
   return 2 * x + 10;
}

int f() {
   return pocitaj(10) +
      pocitaj(20);
}
```

```
int f() {
003F3CA0  push        ebp
003F3CA1  mov         ebp,esp
003F3CA3  push        esi
return pocitaj(10) + pocitaj(20);
003F3CA4  mov         ecx,0Ah
003F3CA9  call        pocitaj (03F1235h)
003F3CAE  mov         esi,eax
003F3CB0  mov         ecx,14h
003F3CB5  call        pocitaj (03F1235h)
003F3CBA  add         eax,esi
}
003F3CBC  pop         esi
003F3CBD  pop         ebp
003F3CBE  ret
```

Súčasťou konvencií volania je aj **dohoda o použití registrov**: funkcia smie zmeniť ľubovoľne registre EAX, ECX a EDX, ale registre EBX, ESI a EDI musí zachovať. Preto kompilátor vygeneroval push esi a pop esi.