Úroveň strojového kódu procesor Intel® Pentium® Adresovanie pamäte

- Pamäťový operand
- Priama nepriama a indexovaná adresa
- Práca s jednorozmerným poľom
- Praktické programovanie assemblerových funkcií

Autor: Peter Tomcsányi,

Adresovanie pamäte

- Opakovanie: operand inštrukcie môže byť **priamy** (číslo), **register, pamäťový** alebo **implicitný** (nie je ho vidno, je súčasťou definície inštrukcie)
- Keď je operand inštrukcie v operačnej pamäti, hovoríme mu **pamäťový operand**
- Bežné inštrukcie môžu mať **najviac jeden pamäťový operand**
- Aby vedel procesor nájsť pamäťový operand, musí poznať jeho adresu, musíme ho nejako **adresovať**
- Pamäťové operandy zodpovedajú premenným vo vyšších programovacích jazykoch
- Preto tvorcovia strojového kódu navrhujú také spôsoby adresovania, aby sa pomocou nich dali adresovať všetky druhy premenných, ktoré poznáme z vyšších programovacích jazykov

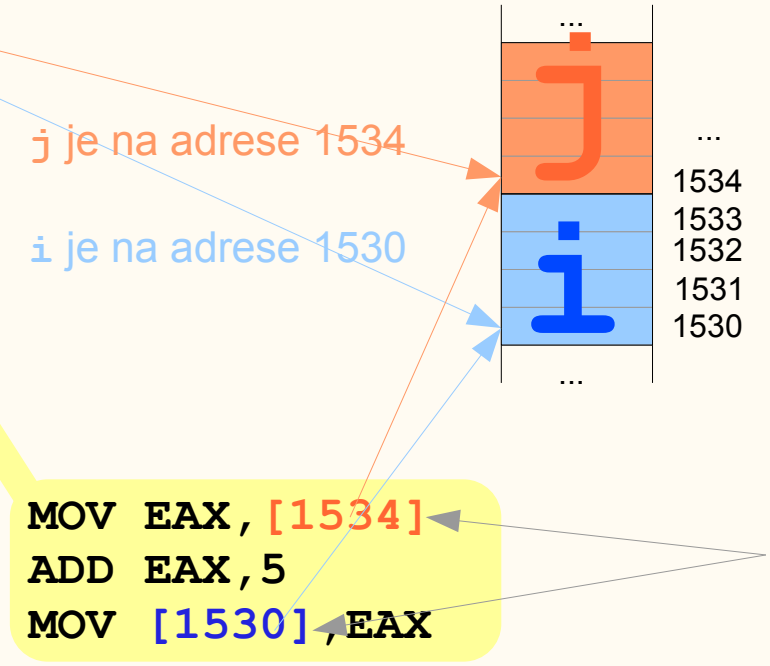
Priama adresa

Jednoduchá globálna premenná

Globálne premenné sú premenné deklarované mimo funkcií. Kompilátor im pridelí pevné miesto v pamäti, teda pevnú adresu.

V našom príklade predpokladajme, že premenná **i** je na adrese 1530 a premenná **j** je hneď za ňou na adrese 1534 (skutočné čísla môžu byť, pravdaže, iné a závisia od toho, aké ďalšie premenné a procedúry sme pred nimi deklarovali ako aj od algoritmu prideľovania adries pre premenné v kompilátore).

```
int32_t i,j;
int main()
{ ...
  i = j+5;
  ...
}
```



Priama adresa je číslo v hranatých zátvorkách

Nepriama adresa

použitie smerníka v C

Smerník (pointer) je premenná, ktorá neobsahuje hodnotu, ale adresu nejakej inej premennej. Keď používame smerník na prístup k hodnote premennej, tak vlastne nevieme, na akej adrese je uložená naša premenná, ale vieme, na akej adrese je uložená jej adresa. To sa nazýva **nepriame adresovanie**.

```
void p(int32_t *x, int32_t n)
{
   ...
   *x = n;
   ...
}
```

```
MOV EAX, x
MOV EDX, n
MOV [EAX], EDX
```

Takéto adresovanie parametrov x a n je tiež prístup do pamäti, vysvetlíme ho ale až neskôr.

Ulož hodnotu EDX do pamäťového miesta, ktorého adresa je v EAX

Nepriama adresa je meno registra v hranatých zátvorkách

Priama a nepriama adresa **teoreticky** stačia na adresovanie ľubovoľných dátových štruktúr a polí.
Pre zjednodušenie programovania a prekladu z vyšších jazykov však majú procesory Intel aj zložitejšie spôsoby adresovania.

Piaty príklad – pole ako parameter

metóda posúvania smerníka – najprv v C

Naprogramujte funkciu:

```
int sucet_prvkov_c(int a[], unsigned int n)
```

Jej výsledkom je súčet prvkov poľa **a**, ktoré má dĺžku **n**.

alebo aj `int *a`

```
int sucet_prvkov_c(int a[], unsigned int n) {
    int i,s;
    int *p;
    p = a;
    s = 0;
    for (i = 0; i < n; i++) {
      s += *p;
      p++;
    }
    return s;
}
```

Podobne ste sa učili pracovať s poľom na cvičeniach s tatrabotom:
http://dai.fmph.uniba.sk/courses/pphw/ex/3.php

Piaty príklad – pole ako parameter

metóda posúvania smerníka – v assembleri

Naprogramujte funkciu:

```
int32_t sucet_prvkov(int32_t a[], uint32_t n)
```

Jej výsledkom je súčet prvkov poľa **a**, ktoré má dĺžku **n**.

```
__asm {
    mov ebx, a          // p = a;
    mov eax, 0          // s = 0;
    mov ecx, n          // for (i = 0; i < n; i++) {
    jecxz a2            // riesime situaciu ked je n == 0
a1: add eax, [ebx]      //   s += *p
    add ebx, 4          //   p++;
    loop a1             //  }
a2:                     // return s;
}
```

Indexovaná adresa

Prvok globálneho poľa

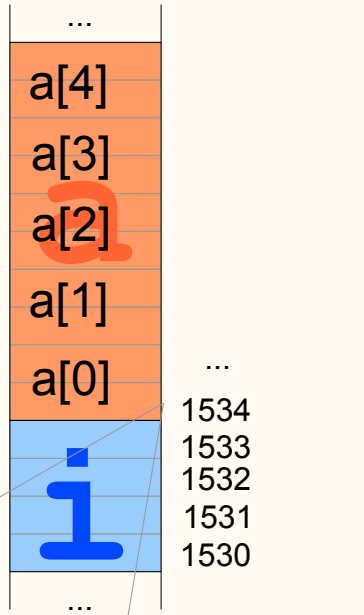
Globálnemu poľu pridelí kompilátor pevné miesto v pamäti,
teda pevnú adresu jeho prvého prvku.
Ak máme deklaráciu poľa: ***typ* a[max];**

a začína v pamäti na adrese **adr(a)**, tak adresa i-teho prvku **a** sa vypočíta:
adr(a[i]) = adr(a)+i*sizeof(*typ*)

```
int32_t i;
int16_t a[5];
int main() {
   ...
   a[i] = i+1;
   ...
}
```

```
MOV AX,[1530]
INC AX
MOV EDX,[1530]
ADD EDX,EDX
ADD EDX,1534
MOV [EDX],AX
```

V EDX máme spočítanú adresu **a[i]**, preto môžeme použiť **nepriamu adresu**




```
MOV AX,[1530]
INC AX
MOV EDX,[1530]
MOV [EDX*2+1534],AX
```

Indexovaná adresa má tvar [reg*c+k] kde reg je 32-bitový register, c je 1, 2, 4 alebo 8 a k je konštanta (môže byť aj záporná).

Bázovaná a indexovaná adresa

Prvky polí prijatých ako parameter

Ak je parameter pole, tak funkcia dostane vždy smerník na pole, teda vlastne **adr(a)**.
Pre výpočet adresy prvku stále platí **`adr(a[i]) = adr(a)+i*sizeof(typ)`**
Len teraz adr(a) nie je konštanta, ale premenná (ktorú musíme uložiť do registra).
Pri adresovaní prvkov potom môže použiť najzložitejší spôsob adresovania - **bázovanú a indexovanú adresu.**

```
void f(int32_t a[], int32_t n)
{
    long i;
    ...
    a[i] = i + 1;
    ...
}
```

```
MOV ECX,i
INC ECX
MOV EDX,i
MOV EAX,a
MOV [EAX+EDX*4],ECX
```

Do **ECX** sme spočítali hodnotu **i+1**

a teraz nie je prvok poľa, ale adresa jeho začiatku

4 je sizeof(int32_t)

Bázovaná a indexovaná adresa má tvar [r1+r2*c+k] r1 a r2 sú 32-bitové registre, c je 1, 2, 4 alebo 8 a k je konštanta (môže byť aj záporná). Jednotlivé členy môžu byť zapísané aj v inom poradí.

Ako **indexový register** je použitý EDX, do ktorého sme si uložili hodnotu premennej **i**.

Ako **bázový register** je použitý EAX, do ktorého sme si uložili adresu začiatku poľa z parametra **a**.

Piaty príklad – pole ako parameter

Bázovaná a indexovaná adresa

Naprogramujte funkciu:

```
int32_t sucet_prvkov_ind(int32_t a[], uint32_t n)
```

Jej výsledkom je súčet prvkov poľa **a**, ktoré má dĺžku **n**.

```
__asm {
    mov ebx, a          // p = a;
    mov eax, 0          // s = 0;
    mov ecx, n          // for (ECX = n; ECX > 0; ECX--) {
    jecxz a2                    // riesime pripad n == 0
a1: add eax, [ebx + ecx*4 - 4] // s += a[ECX-1];
    loop a1             // }
a2:
}
```

adr(a)+(ecx-1)*sizeof(int32_t) = ebx+ecx*4-4

Využijeme ECX ako premennú cyklu. V pôvodnom kóde v C sa i menilo od 0 do n-1. V strojovom kóde sa ale mení ECX od n do 1 (tak sa používa inštrukcia LOOP).
Preto musíme v tele cyklu indexovať nie a[ECX] ale a[ECX-1].

Šiesty príklad – pole ako parameter

metóda posúvania smerníka – najprv v C

Naprogramujte funkciu:

```
int max_prvok_c(int a[], unsigned int n)
```

Jej výsledkom je hodnota najväčšieho prvku v poli a, ktoré má dĺžku n.

```
int max_prvok_c(int a[], unsigned int n) {
    int i,m;
    int *p;
    if (n == 0) return 0;
    p = a;
    m = *p;
    for (i = n - 1; i > 0; i--) {
      p++;
      if (*p > m)
        m = *p;
    }
    return m;
}
```

Šiesty príklad – pole ako parameter

metóda posúvania adresy – v assembleri

Naprogramujte funkciu:

```
int32_t max_prvok(int32_t a[], uint32_t n)
```

Jej výsledkom je hodnota najväčšieho prvku v poli a, ktoré má dĺžku n.

```
__asm {
    mov eax, 0          // if (n == 0)
    mov ecx, n          //    ...
    jecxz a3            //    return 0
    mov ebx, a          // p = a;
    mov eax, [ebx]      // m = *p;
    dec ecx             // for (i = n - 1; i > 0; i--) {
    jecxz a3            // riesime situaciu ked je n == 1
a1: add ebx, 4          //    p++;
    cmp [ebx], eax       //    if (*p > m)
    jle a2
    mov eax, [ebx]      //       m = *p;
a2: loop a1             //  }
a3:                     // return m
}
```

Šiesty príklad – pole ako parameter

Bázovaná a indexovaná adresa

Naprogramujte funkciu:

```
int32_t max_prvok_ind(int32_t a[], uint32_t n)
```

Jej výsledkom je hodnota najväčšieho prvku v poli a, ktoré má dĺžku n.

```
__asm {
    mov eax, 0          // if (n == 0) return 0
    mov ecx, n
    jecxz a3
    mov ebx, a
    mov eax, [ebx]
    dec ecx
    jecxz a3
a1: cmp eax, [ebx + ecx * 4]
    jge a2
    mov eax, [ebx + ecx * 4]
a2: loop a1
a3:
}
```

Teraz sa ECX bude meniť od n-1 do 1, to je presne tak, ako potrebujeme.
Preto budeme v tele cyklu indexovať a[ECX].
adr(a)+ecx*sizeof(int32_t) = ebx+ecx*4

Siedmy príklad – pole ako parameter

Bázovaná a indexovaná adresa

Naprogramujte assemblerovú funkciu:

```
void zamen_ind(int32_t a[], uint32_t n, int32_t x,
  int32_t y)
```

ktorá v zamení v poli **a** všetky výskyty hodnoty **x** hodnotou **y**.

ebx + (ecx-1)*4 = ebx + ecx*4 - 4

```
__asm {
    mov ebx, a
    mov edx, x
    mov eax, y
    mov ecx, n                    // for i in range(n,0,-1):
    jecxz a3                      //   # ak je nahodou n == 0
a1: cmp edx, [ebx+ecx*4-4]        //   if P == a[i-1]:
    jne a2
    mov [ebx+ecx*4-4], eax        //     a[i-1] = y
a2: loop a1                       // # koniec cyklu for
a3:
}
```

Ôsmy príklad – reťazec ako parameter

metóda posúvania adresy

Typ **char** je totožný s typom **int8_t**. Pre zvýraznenie, že máme na mysli znaky a nie čísla budeme používať typ **char**.

Naprogramujte assemblerovú funkciu:

```
void zamen(char * a, char x, char y)
```

ktorá v zamení v reťazci **a** všetky výskyty hodnoty **x** hodnotou **y**.

Reťazec je vlastne pole prvkov typu **char** (teda **int8_t**).
Za posledným znakom reťazca je znak s kódom 0. Preto nepotrebujeme vedieť dĺžku dopredu, ale musíme ju "zistiť" v cykle typu **while**.

```
__asm {
    mov ebx, a        // p = a;
    mov dl, x         //
    mov dh, y         //
a1: mov al, [ebx]     // while (*p != 0) {
    cmp al, 0         //
    je a3             //
    cmp dl, [ebx]     //   if (x == *p)
    jne a2            //
    mov [ebx], dh     //     *p = y;
a2: inc ebx           //   p++;
    jmp a1            // }
a3:
}
```

Deviaty príklad

Kedy naozaj potrebujeme bázovanú a indexovanú adresu

Naprogramujte assemblerovú funkciu:

```
void vynuluj(int32_t a[], uit32_t n, uint32_t indexy[],
  uint32_t m)
```

ktorá v v poli **a** (dĺžky **n**)vynuluje všetky prvky na indexoch, ktoré dostane v poli **indexy** (dĺžky **m**).

```
__asm {
    mov ebx, a
    mov edx, indexy
    mov ecx, m
    jecxz a3
a1: mov eax, [edx]
    mov [ebx + eax*4], 0
    add edx, 4
a2: loop a1
a3:
}
```

do poľa **indexy** pristupujeme pomocou posúvania smerníka

```
// p = indexy;
// for (i = m; i > 0; i--) {
//    # ak je nahodou m == 0
//    eax = *p;
//    a[eax] = 0;
//    p++;
// }
```

do poľa **a** pristupujeme bázovanou a indexovanou adresou

Adresovanie pamäte

Zhrnutie a ďalšie informácie

- Najviac jeden operand smie byť v pamäti

  ~~MOV [EAX+2],[EBX+4]~~ ~~CMP [EBX],n~~

- V adrese nesmú byť guľaté zátvorky ani premenné:

  ~~MOV EAX, [EBX+(ECX-1)*4]~~ ← Treba roznásobiť: MOV EAX, [EBX+ECX*4-4]

  ~~MOV EAX, [EBX+n*4]~~ ← Treba najprv uložiť n do nejakého registra

- Určenie dĺžky operandu (keď assembler nevie):

  INC [EBX] ←

  MOV [EDX+8],0 ← nie je jasná dĺžka operandu. Skutočný Intel assembler by hlásil chybu. **Ale visual Studio vtedy predpokladá, že operand má jeden bajt!**

  MOV **BYTE PTR** [EDX+8],0 ← operand má jeden bajt

  MOV **WORD PTR** [EDX+8],0 ← operand má dva bajty

  MOV **DWORD PTR** [EDX+8],0 ← operand má štyri bajty

V assembleri NASM sa nepíše PTR, teda len BYTE, WORD alebo DWORD.