

Introdução à Arquitetura de Computadores

Aula 20

Assembly 5: Ponteiros

Mais Assembly

Ponteiro: Definição e Propriedades

- Uso de Arrays com Ponteiros
- Índices versus Ponteiros
- Exemplos

Instruções Signed/Unsigned - Resumo

A. Nunes da Cruz / DETI - UA

Maio / 2018

1 - Ponteiro (1) - Definição

Ponteiro

É um tipo variável que contém o endereço de memória de outra variável.

1. Ponteiro - Definição

O tipo-de-dados (da variável apontada)

- `char`, `int`, `word`, `float`, `double`,
- `array[]` (de `char`, `int`, `word`, etc)
- ou uma `struct` mais complicada.

Ocupa, em memória, um ou múltiplos *bytes*.

Ponteiros porquê?

Código mais compacto, rápido e eficiente.

Também se diz que o ponteiro 'aponta' para uma variável, como sinónimo do endereço de memória.

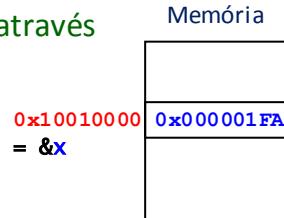
1 - Ponteiro (2) - Declaração e Inicialização - C

Uma variável **x** do tipo inteiro tem o valor **0x1FA** e está localizada no endereço de memória (dados) **0x10010000**.

1. Declaração dum ponteiro, em C

```
// A declaração do ponteiro p_int, é feita através do operador '*'
```

```
int* p_int;
```



2. Inicialização do ponteiro

```
// A inicialização é feita através do operador '&'
```

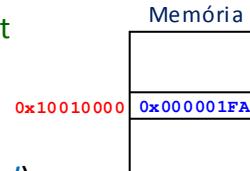
```
int x;
int* p_int = &x; // '&' atribui a p_int o endereço de x
```

1 - Ponteiro (3) - Operador '*' - C

3. Acesso à variável x usando o ponteiro p_int

```
// Declaração e inicialização do ponteiro p_int
```

```
int x;
int* p_int = &x;
```



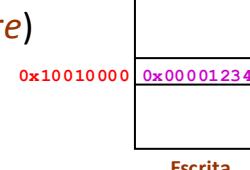
Leitura usando o operador '*' (ASM: *Load*)

```
x = *p_int; // x := 0x1FA
```

Leitura

Escrita usando o operador '*' (ASM: *Store*)

```
*p_int = 0x1234; // x := 0x1234
```



Escrita

1 - Ponteiro (4) - Incrementar/Decrementar - C

Ponteiro para char

```
char* p_char = 0x10010000; // inic. o ponteiro
p_char++;    // aponta para o char seguinte
              // p_char = 0x10010001
```

Ponteiro para int

```
int* p_int = 0x10010000; // inic. o ponteiro
p_int++;    // aponta para o int seguinte
              // p_int = 0x10010004
```

Em C a sintaxe é igual em ambos os casos.

1 - Ponteiro (5) - Arrays - C

Ponteiro para char

```
char a_chars[3] = { 'i', 'a', 'c' };
char* p_char = a_chars; // p_char = &a_chars[0]
p_char += 2;          // p_char avança 2 bytes
                      // *p_char = 'c'
```

Ponteiro para int

```
int a_ints[3] = { 1234, -432, 12 };
int* p_int = a_ints; // p_int = &a_ints[0]
p_int += 2;          // p_int avança 2x4 bytes
                      // *p_int é = 12
                      // 4 = sizeof( int )
```

Em C a sintaxe é igual nos dois casos.

Em ASM são tratados distinta/, i.e., o ponteiro é incrementado em múltiplos do tamanho-em-bytes da variável apontada.

1 - Ponteiro (6) - De C para ASM

A variável `x` do tipo inteiro tem o valor `0x1FA` e está localizada no endereço `0x10010000`.

Suponhamos: `p -> $a0` e `x -> $s0`

```

1. p = &x;           // p gets 0x10010000
   la $a0,0x10010000 # p = 0x10010000

2. x = *p;          // x gets 0x01fA
   lw   $s0,0($a0)    # dereferencing p

3. *p = 0x1234;    // x gets 0x1234
   addi $t0,$0,0x1234
   sw   $t0,0($a0)    # dereferencing p
  
```

2 - Arrays: Índices vs Ponteiros (1)

- Usando **índices** implica:
 - Multiplicar o **índice** do elemento pelo respetivo tamanho (em *bytes*) para obter o **offset**
 - Adicionar esse **offset** ao Endereço-Base do Array
- Os **ponteiros** são endereços de memória:
 - A sua utilização, em alternativa ao uso de **índices**, em geral reduz a complexidade do código de acesso ao elementos, bastando atualizar o **ponteiro** em cada iteração.

2 - Arrays: Índices vs Ponteiros (2) - Ex1: 'Zerar' um Array

Índice	Ponteiro
<pre>void clear_i(int array[], int size) { int i=0; do { array[i] = 0; // clear i++; // inc. index } while (i < size); }</pre> <pre>move \$t0,\$0 # i = 0 dw1: sll \$t1,\$t0,2 # \$t1 = i * 4 addu \$t2,\$a0,\$t1 # \$t2 = # &array[i] sw \$0, 0(\$t2) # array[i] = 0 addi \$t0,\$t0,1 # i++ slt \$t3,\$t0,\$a1 # \$t3 = # (i < size) bne \$t3,\$0,dw1 # if(i < size) # goto dw1</pre> <p>loop dw1: 6 instruções $\&\text{array}[i]$: 2 adições + 1 sll</p>	<pre>void clear_p(int *array, int size) { int *p = &array[0]; //or p = array do { *p = 0; // clear p++; // inc. pointer } while (*p < &array[size]); }</pre> <pre>move \$t0,\$a0 # p = &array[0] sll \$t1,\$a1,2 # \$t1 = size*4 addu \$t2,\$a0,\$t1 # \$t2 = # &array[size] dw2: sw \$0,0(\$t0) # *p = 0 addiu \$t0,\$t0,4 # p++ slt \$t3,\$t0,\$t2 # \$t3 = # (p < &array[size]) bne \$t3,\$0,dw2 # if(...) # goto dw2</pre> <p>loop dw2: 4 instruções $p = \&\text{array}[i]$: 1 adição</p>

Nota: $\$a0$ e $\$a1$ são os argumentos das funções `clear_i(...)` e `clear_p(...)`!

2 - Arrays: Idxs vs Ptrs (3) - Ex2: ToUpper - C

- Conversão dum *string* (array de caracteres) em maiúsculas usando índices e ponteiros

```
char str[] = "Arrays:Indexes vs Pointers Example";
```

```
// Indexes
void ToUpperI( char str[]){
int i=0;
    while( str[i]!='\0' ){
        if((str[i]>='a') && (str[i]<='z'))
            str[i]=str[i] - ' ' ;// ' ' = 0x20
        i++;
    }
}
```

```
// Pointers
void ToUpperP( char* str){
char *s=str;
    while( *s != '\0' ){
        if( ( *s >='a') && ( *s <='z' ) )
            *s = *s - ' ' ; // ' ' = 0x20
        s++;
    }
}
```

1. O argumento `str[]` é um *array* de `char`.

2. O operador `&&` é o operador AND-Booleano (diferente do operador `'&` AND-Bitwise).

1. O argumento `str` é um *ponteiro* para `char`.

2. O operador `'*'` desreferencia o ponteiro '`s`', isto é, accede ao valor da variável apontada por '`s`', tanto para leitura como para escrita.

2 - Arrays: Idxs vs Ptrs (4) - ToUpperI - ASM Índices

- Na versão com *índices* cada iteração do *loop* tem de recalcular o endereço de *str[i]*, o que requere **duas** somas.

```
# void ToUpperI( char str[] );    # $a0 = str
toupr_i: li      $t0, 0          # $t0 = i = 0
lp_i:   addu   $t1, $t0, $a0    # $t1 = &str[i]
        lb      $t2, 0($t1)       # $t2 = str[i]
        beq    $t2, $0, done_i    # $t2 = 0? ('\'0')
        blt    $t2, 'a', next_i   # $t2 < 97?
        bgt    $t2, 'z', next_i   # $t2 > 122?
        sub    $t2, $t2, ' '
        sb     $t2, 0($t1)       # convert
        # and store back
next_i:  addi   $t0, $t0, 1      # i++
done_i: j      lp_i

```

Num array de *words* o cálculo do endereço de *str[i]* exigiria ainda uma multiplicação por 4 (slide 8).

```
//Indexes
void ToUpperI( char str[] ){
int i = 0;
while ( str[i] != 0 ) {
if ( (str[i] >= 'a') && (str[i] <= 'z') )
str[i] = str[i] - ' ';
i++;
}
```

© A. Nunes da Cruz

IAC - ASMS: Ponteiros

10/20

2 - Arrays: Idxs vs Ptrs (5) - ToUpperP - ASM Ponteiros

- Na versão com *ponteiros*, usamos um registo com o endereço exato do elemento corrente. Em cada iteração do *loop* incrementamos o valor desse registo para apontar para o elemento seguinte.

```
# void ToUpperP(char* str);      # $a0 = str; $a0 chgs!
toupr_p: lb      $t2, 0($a0)    # $t2 = *s
        beq    $t2, $0, done_p    # $t2 = 0? ('\'0')
        blt    $t2, 'a', next_p   # $t2 < 'a'?
        bgt    $t2, 'z', next_p   # $t2 > 'z'?
        sub    $t2, $t2, ' '
        sb     $t2, 0($a0)       # convert
        # and store back
next_p: addi   $a0, $a0, 1      # s++
done_p: jr      $ra

```

Num array de *words* teríamos de **incrementar o ponteiro** por 4. De qq modo, precisaríamos de uma só adição em vez de duas adições e uma multiplicação por 4 (*sll*) (ver slide 8).

```
//Pointers
void ToUpperP( char* str){
char *s = str;
while (*s != 0) {
if ( (*s >= 'a') && (*s <= 'z') )
*s = *s + ' ';
s++;
}
}
```

© A. Nunes da Cruz

IAC - ASMS: Ponteiros

11/20

2 - Arrays: Idxs vs Ptrs (6) - ToUpper - ASM Idx vs Ptr

```

str:    .ascii "Arrays: Indexes vs Pointers Example\n"
        # void ToUpperI( char str[] );      # $a0 = str
        toupr_i: li    $t0, 0             # $t0 = i
        lp_i:    add   $t1, $t0, $a0     # $t1 = &str[i]
        lb     $t2, 0($t1)            # $t2 = str[i]
Índice   beq   $t2, $0, done_i # $t2 = 0?
        blt   $t2, 'a', next_i # $t2 < 'a'?
        bgt   $t2, 'z', next_i # $t2 > 'z'?
        sub   $t2, $t2, ' '
        sb     $t2, 0($t1)            # convert
        done_i: addi  $t0, $t0, 1       # and store back
        next_i: addi  $t0, $t0, 1       # i++
        j     lp_i
done_i:  jr    $ra

```

```

# void ToUpperP(char* str);      # $a0 = str; changes $a0!
        toupr_p: lb    $t2, 0($a0)        # $t2 = *s
        beq   $t2, $0, done_p # $t2 = 0?
        blt   $t2, 'a', next_p # $t2 < 'a'?
        bgt   $t2, 'z', next_p # $t2 > 'z'?
        sub   $t2, $t2, ' '
        sb     $t2, 0($a0)            # and store back
next_p:  addi  $a0, $a0, 1       # s++
        j     toupr_p
done_p:  jr    $ra

```

© A. Nunes da Cruz

IAC - ASMs: Ponteiros

12/20

2 - Arrays: Idxs vs Ptrs (7) - Ex3: Soma Ptrs - C

■ TPC?

```

#define SIZE 4
void main(void){
    // Declara um array estático de 4 inteiros e inicializa-o
    static int a_ints[SIZE] = { 7692, 23, 5, 234 };

    int *p = &a_ints[0];           // Declara um ponteiro para inteiro
                                    // 'p' é inicializado com &a_ints[0]
    int *pultimo = &a_ints[SIZE-1]; // "pultimo" é inicializado com o
                                    // &a_ints[3]

    int soma = 0;
    while( p <= pultimo ) {
        soma += *p;              // acumula o valor em soma
        p++;                     // Incrementa o ponteiro
    }
    print_int10(soma);
}

```

© A. Nunes da Cruz

IAC - ASMs: Ponteiros

13/20

2 - Arrays: Idxs vs Ptrs (8) - Ex3: Soma Ptrs - ASM

```

#      int soma =0
#      int* p = a_ints;
#      int* pulultimo = a_ints + 3;
#      while( p <= pulultimo ) {
#          soma += *p ;
#          p++;
#      }
#      print_int10( soma );
# -----
.eqv PRINT_STR,4
.eqv PRINT_INT10,1
.eqv EXIT,10
#
.eqv SIZE3,12
.data
# int a_ints[] = {7692, 23, 5, 234};
a_ints: .word 7692, 23, 5, 234
#
.sum_hdr: .asciz
    "\nSoma do array de inteiros: "

```

```

.text
.globl main
# $t0 = p; $t1 = pulultimo;
# $t2 = *p; $t3 = soma
#
main: li    $t3,0 # Soma =0
      la    $t0,a_ints # $t0 = p = a_ints
# pulultimo = a_ints + (NSIZE-1)*sizeof(int)
      addiu $t1,$t0,SIZE3 # $t1 = a_ints + 3*4
      # if( p > pulultimo ) wh_end
# wh: bgtu $t0,$t1,wh_end # bgtu is 'pseudo'
wh:   sltu $at,$t1,$t0 # $at =($t1<$t0)?1:0
      bne  $at,$0,wh_end # if( $at ) wh_end
      #
      lw    $t2, 0($t0) # $t2 = *p
      add  $t3, $t3, $t2 # soma += *p
      #
      addiu $t0,$t0,4     # p++
      j    wh
      # write output header
wh_end:la   $a0,s_sum_hdr
        li   $v0,PRINT_STR
        syscall
        # print sum
        move $a0,$t3
        li   $v0,PRINT_INT10
        syscall
        # exit
        li   $v0,EXIT
        syscall
> Soma do array de inteiros : 7954

```

Mas, se andarem muito ocupados ☺...

© A. Nunes da Cruz

IAC - ASM5: Ponteiros

14/20

3 - Instruções Signed/Unsigned (1) - Adição subtração

3. Signed/Unsigned

- Adição e Subtração
- Multiplicação e Divisão
- Comparação: *Set Less Than*

• Signed: add, addi, sub

- Mesma operação que as versões *unsigned*
- O processador gera exceção de *overflow*

• Unsigned: addu, addiu, subu

- Não geram exceção de *overflow*

addiu - sign-extends the immediate

© A. Nunes da Cruz

IAC - ASM5: Ponteiros

15/20

3 - Instruções Signed/Unsigned (2) - MUL, DIV e SLT

- Multiplicação e Divisão
 - Signed: mult, div
 - Unsigned: multu, divu

- Comparação: Set Less Than
 - Signed: slt, slti
 - Unsigned: sltu, sltiu

sltiu - sign-extends the immediate before comparing it to the register

3 - Instruções Signed/Unsigned (3) - Loads

- Signed (Com Sinal)
 - Sign-extends to create a 32-bit value to load into register
 - Load byte: lb
 - Load halfword: lh

- Unsigned (Sem Sinal)
 - Zero-extends to create a 32-bit value
 - Load byte unsigned: lbu
 - Load halfword unsigned: lhu

3 - Instruções Signed/Unsigned (4) - ADDIU vs ADDI

Porquê addiu em vez de addi ?

addiu, addu e subu - Nao geram exceções de *overflow*
(s o usadas pelos ponteiros!)

addi, add e sub - podem gerar exceções de *overflow*

```
.text
.globl main
main: li      $t0,0x7FFFFFFF  # max. positive value
#      addi    $t1, $t0, 4      # arithmetic overflow exception
#      causing program to abort
#      execution (MARS).
addiu   $t1, $t0, 4      # does NOT generate overflow
#
li      $v0,10
syscall
```

Runtime exception at 0x00400008: arithmetic overflow

3 - Instruções Signed/Unsigned (5) - SLT vs SLTU

\$s0 = 1111 1111 1111 1111 1111 1111 1111 1111 1111

\$s1 = 0000 0000 0000 0000 0000 0000 0000 0000 0001

```
slt    $t0, $s0, $s1  # signed
# -1 < +1           => $t0 = 1
```

```
sltu   $t0, $s0, $s1  # unsigned
# +4,294,967,295 > +1 => $t0 = 0
```

Aparentemente as instru es fazem a mesma coisa, mas o resultado   exactamente o oposto. Porqu ?

Porque na realidade elas s o diferentes!

XX - NEXT: Assembling & Loading

