Introdução à Arquitetura de Computadores

Aula20

Assembly 5: Mais Assembly

Ponteiro

- Definição
- Declaração e Inicialização:
  Operadores '*' e '&'
- Acesso a Elementos
  Operador '*': Leitura e Escrita
- Incrementar em C
  Tipo-de-dados e Endereço
- Exs com índices vs ponteiros
  Zerar (inteiros); toUpper (carateres)
- Exercício só com ponteiros
  Soma (inteiros)

Instruções *Signed* e *Unsigned*

- Resumo

1 - Ponteiro (1) - Definição

Ponteiro

É uma variável (de determinado tipo) que contém o endereço de memória de outra variável.

O tipo-de-dados apontado* pode ser:

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

Ponteiros?

A sua utilização gera código mais compacto e rápido.

*Ponteiro para uma variável, significa que contém o respectivo endereço de memória.

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

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

1. A declaração do ponteiro p_int

- É feita através do operador '*':

```
int*  p_int;
```

o ponteiro para um inteiro

- p_int aponta para uma variável do tipo int.

2. A inicialização do ponteiro p_int

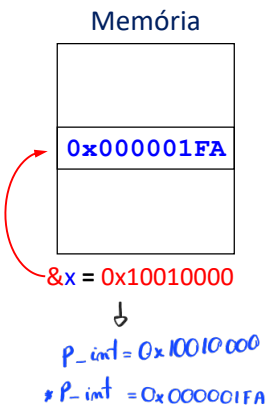
- É feita através do operador '&':

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

endereço do inteiro

o que está guardado na memória

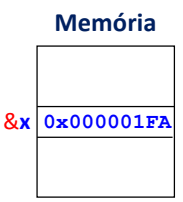
- '&' atribui a p_int o endereço da variável x.



1 - Ponteiro (3) - Operador '*' - C : Acesso à variável x

1. e 2. Declaração e Inicialização

```
int    x;              // decl. de x
int*   p_int  = &x;    // decl. e inicial. de p_int.
```



3. Leitura e Escrita com o ponteiro p_int

- Leitura:

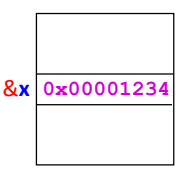
```
x  = *p_int;          // x := 0x1FA
```

x é igual ao conteúdo de *p_int

p_int ≠ *p_int

- Escrita:

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



O operador '*' é usado para declarar o ponteiro e para aceder ao valor da variável apontada (tanto para a leitura como para a escrita).

1 - Ponteiro (4) - C: Incrementar vs Tipo-de-dados (1)

- Ponteiro para char → Cada caracter ocupa 1 byte

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

- Ponteiro para int → Cada inteiro ocupa 4 bytes

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

- A sintaxe é igual em ambos os casos!

1 - Ponteiro (5) - C: Incrementar vs Tipo-de-dados (2)

- Ponteiro para char

```
char  achars[3] = { 'i', 'a', 'c' };
char* p_char = achars;   // p_char = &achars[0]
      p_char += 2;       // p_char avança 2 bytes
                         // *p_char é = 'c'
                         // 1 = sizeof( char )
```

o endereço da posição zero é $(achars[0])

- Ponteiro para int

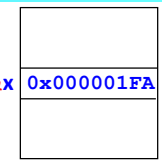
```
int   aints[3]  = { 1234, -432, 12 };
int*  p_int = aints;     // p_int = &aints[0]
      p_int += 2;        // p_int avança 2x4 bytes
                         // *p_int é = 12
                         // 4 = sizeof( int )
```

Avançou 8 endereços (2x4)

- 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: Inicial., Leitura e Escrita

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



Em ASM, suponhamos: p -> $a0 e x -> $s0

1. Inicialização do ponteiro

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

endereço de x

2. Leitura do valor da variável apontada por p

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

3. Escrita de novo valor na variável apontada por p

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


temos de determinar
? &(array[i]) = array + i × tamanho dos elementos

2 - Arrays: Acesso com Índices vs Acesso com Ponteiros (1)

- O índice é o número de ordem do elemento no *array*.

  O endereço de memória desse elemento é calculado:

  1. Multiplicando o índice do elemento pelo respetivo tamanho-em-*bytes* para obter o *offset*;
  2. Adicionando esse *offset* ao Endereço-Base do *array*.

- O ponteiro é, por definição, um endereço de memória.

  A sua utilização, em alternativa ao indíce, reduz a complexidade do código de acesso ao elemento, bastando atualizar o ponteiro em cada iteração.

2 - Arrays: Índices vs Ponteiros (2) - Ex1: Acesso em Asm

```
int aints[3] = { 1234, -432, 12 }; // size=3
```

Acesso com índices:

```
int i = 0;
  while( i != 3 ){
    aints[i] += 18;
    i++;
  }
```

Acesso indexado

aints

...
1234
-432
12
...

Acesso com ponteiros:

```
int* p = &aints[0];
  while( p != &aints[3]){
    *p = *p + 18;
    p++;
  }
```

array indice 3 não existe!

Quando o p chega vai terminar o while = while (i != 3)

Acesso indexado

endereço base

ciclo while

```
i ->$t0; aints = &aints[0]->$a0
  &aints[i]->$t1; aints[i]->$s0
```

```
la      $a0,aints    # $a0=&aints[0]
li      $t0,0        # i=0
#
wh:  beq     $t0,3,ewh    # if(i==3)ewh
     sll     $t1,$t0,2    # $t1 = i*4
     addu    $t1,$t1,$a0  # $t1=&aints[i]
     lw      $s0,0($t1)   # $s0 = val
     addi    $s0,$s0,18   # $s0+= 18
     sw      $s0,0($t1)   # aints[i]=
                          # val+18
     addi    $t0,$t0,1    # i++
     j       wh
ewh: ...
```

t, tem o endereço

Vai buscar/mudar/guarda

Em cada iteração o valor de $a0 é constante!

```
p  = &aints[i]->$a0; *p = aints[i]->$s0
pz = &aints[size]->$a1
```

```
la      $a0,aints    # $a0=&aints[0]
addiu   $a1,$a0,12   # $a1=&aints[3]
                     # 12 = 3*4
wh:  beq     $a0,$a1,ewh # if(i==3)ewh
     lw      $s0,0($a0)  # $s0=*p = val
     addi    $s0,$s0,18  # $s0 += 18
     sw      $s0,0($a0)  # *p = val+18
                         #
     addiu   $a0,$a0,4   # p++
     j       wh
ewh: ...
```

end while

Em cada iteração o valor de $a0 (p), aponta para o elemento i (p = &aints[i]).

Não temos de calcular endereço

vai buscar/mudar/guarda

2 - Arrays: Índices vs Ponteiros (3) - Ex2: 'Zerar' um Array

Índice

```
void clear_i(int array[], int size) {
  int i=0;
  do {
       array[i] = 0;     // clear
       i++;              // inc. index
   } while (i < size);
}
```

```
     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
```

= array[i]

loop dw1: 6 instruções

&array[i]: 2 adições + 1 sll

Ponteiro

```
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] );
}
```

```
     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++
     sltu $t3,$t0,$t2     # $t3 = (p<
                          # &array[size])
     bne  $t3,$0,dw2      # if (...)
                          # goto dw2
```

loop dw2: 4 instruções

p = &array[i]: 1 adição

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

- Ponteiros torna o código mais simples, mas nesta cadeira se nos pedirem para traduzir os códigos em C temos de fazer como é implementado
  - Temos de saber fazer os 2!

2 - Arrays: Idxs vs Ptrs (4) - Ex3: toUpper - C

- Conversão duma *string* (*array* de carateres) em maiúsculas; Acesso aos elementos do array usando índices vs ponteiros.

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

```
// Indexes
void toUpperI( char str[] ){
int i = 0;
    while ( str[i] != '\0' ) {
      if ( (str[i] >= 'a') && (str[i] <= 'z') )
            str[i] = str[i] - 32 ;
      i++;
    }
}
```

1. O argumento str[] é um *array* de char.
2. O operador '&&' é o AND-Booleano (diferente do operador '&' AND-Bitwise).

```
// Pointers
void toUpperP( char* str ){
char *p = str;
    while ( *p != '\0' ) {
      if ( ( *p >= 'a' ) && ( *p <= 'z' ) )
            *p = *p - 32;
      p++;
    }
}
```

1. O argumento str é um *ponteiro* para char.
2. O operador '*' desreferencia o ponteiro 'p', isto é, acede ao valor da variável apontada por 'p', tanto para leitura como para escrita.

2 - Arrays: Idxs vs Ptrs (5) - toUpperI - ASM Indíces

- Com *índices*, cada iteração tem de calcular o endereço de str[i], o que requere duas somas.

```
# void toUpperI( char str[] );     # $a0 = str
toUpperI: li     $t0, 0           # $t0 = i = 0
lpi:      addu   $t1, $t0, $a0 <-- # $t1 = &str[i]
          lb     $t2, 0($t1)      # $t2 = str[i]
          beq    $t2, $0, donei   # $t2 = 0? (='\0')
          blt    $t2, 'a', nexti  # $t2 < 'a'?
          bgt    $t2, 'z', nexti  # $t2 > 'z'?
          addi   $t2, $t2, -32    # convert
          sb     $t2, 0($t1)      # and store back
nexti:    addi   $t0, $t0, 1      # i++
          j      lpi
donei:    jr     $ra
```

Num *array* de *words* o cálculo do endereço de str[i] exigiria ainda uma multiplicação por 4 (slide 9).

```
//Indexes
void toUpperI( char str[] ){
 int i = 0;
     while ( str[i] != 0 ) {
          if ( (str[i] >= 'a') && (str[i] <= 'z' ) )
               str[i] = str[i] - 32 ;
          i++;
     }
}
```

2 - Arrays: Idxs vs Ptrs (6) - toUpperP - ASM Ponteiros

- Com *ponteiros*, usamos um registo com o endereço exato do elemento corrente. Em cada iteração incrementamos esse registo para apontar para o elemento seguinte.

```
# void toUpperP(char* str);      # $a0 = p = str;
toUpperP: lb    $t2, 0($a0)      # $t2 = *p
          beq   $t2, $0, donep   # $t2 == '\0' ?
          blt   $t2, 'a', nextp  # $t2 < 'a'?
          bgt   $t2, 'z', nextp  # $t2 > 'z'?
          addi  $t2, $t2, -32    # convert
          sb    $t2, 0($a0)      # and store back
nextp:    addiu $a0, $a0, 1 <--- # p++; changes $a0!
          j     toUpperP
donep:    jr    $ra
```

```
//Pointers
void toUpperP( char* str ){
char *p = str;
    while (*p != 0) {
        if ( (*p >= 'a' ) && (*p <= 'z' ) )
            *p = *p - 32;
        p++;
    }
}
```

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 9).

2 - Arrays: Idxs vs Ptrs (7) - toUpper - ASM Idx vs Ptr

Índice vs Ponteiro :

Índice

```
# void toUpperI( char str[] );   # $a0 = str
toUpperI: li    $t0, 0           # $t0 = i
lpi:      add   $t1, $t0, $a0    # $t1 = &str[i]
          lb    $t2, 0($t1)      # $t2 = str[i]
          beq   $t2, $0, donei   # $t2 = 0?
          blt   $t2, 'a', nexti  # $t2 < 'a'?
          bgt   $t2, 'z', nexti  # $t2 > 'z'?
          addi  $t2, $t2, -32    # convert
          sb    $t2, 0($t1)      # and store back
nexti:    addi  $t0, $t0, 1      # i++
          j     lpi
donei:    jr    $ra
```

Ponteiro

```
# void toUpperP(char* str);      # $a0 = str;
toUpperP: lb    $t2, 0($a0)      # $t2 = *s
          beq   $t2, $0, donep   # $t2 = 0?
          blt   $t2, 'a', nextp  # $t2 < 'a'?
          bgt   $t2, 'z', nextp  # $t2 > 'z'?
          addi  $t2, $t2, -32    # convert
          sb    $t2, 0($a0)      # and store back
nextp:    addiu $a0, $a0, 1      # p++;changes $a0!
          j     toUpperP
donep:    jr    $ra
```

2 - Arrays: Idxs vs Ptrs (8) - Ex4: Soma Ptrs - C

Soma dos elementos dum *array* (com ponteiros)

```c
#define SIZE 4

void main (void) {

// Declara um array estático de 4 inteiros e inicializa-o
static int aints[ SIZE ] = { 7692,  23,  5,  234 };
int *p = &aints[0];                 // declara um ponteiro para inteiro
                                    // 'p' é inicializado com &aints[0]
int * pultimo = &aints[ SIZE-1 ];   // "pultimo" é inicializado com &aints[3]

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


2 - Arrays: Idxs vs Ptrs (9) - Ex4: Soma Ptrs - ASM

```c
#define SIZE 4

void main (void) {

// Declara um array ...
static
int aints[SIZE] = {7692,23,5,234};
// Ponteiros
int *p = &aints[0];
int *pultimo = &aints[SIZE-1];
// soma=0
int soma = 0;
      while( p <= pultimo ) {
            soma += *p ;
            p++;
      }
      print_int10(soma);
}
```

```
        .eqv   print_int10,1
        .eqv   exit,10
        .eqv   SIZE3,12  # 3*4
        .data
aints:  .word 7692,23,5,234 # int aints[]={...}
        .text
        .globl main
# --------------------------
# $t0 = p  ; $t1 = pultimo;
# $t2 = *p ; $t3 = soma
# --------------------------
main:   la     $t0,aints       # $t0 = p = aints
# pultimo = aints + (NSIZE-1)*sizeof(int)
        addiu  $t1,$t0,SIZE3   # $t1 = aints + 3*4
        li     $t3,0           # soma =0
        # if( p > pultimo ) ewh
wh:     bgtu   $t0, $t1, ewh   #
        lw     $t2, 0($t0)     # $t2 = *p
        add    $t3, $t3, $t2   # soma += *p
        addiu $t0,$t0,4        # p++
        j      wh
ewh:    move   $a0,$t3         # print sum
        li     $v0,print_int10
        syscall
        li     $v0,exit        # exit
        syscall
# soma: 7954
```

3 - Instruções Signed/Unsigned (1) - add, addi e sub

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

- **Signed** (com sinal): **add, addi, sub**
  - Mesma operação que as versões *unsigned*
  - O CPU gera excepção de *overflow*
- **Unsigned** (sem sinal): **addu, addiu, subu**
  - Não gera excepção de *overflow*

int x,y; ⊄ unsigned x,y;
x = y + 1
addi ≠ addiu

addiu - sign-extends the immediate

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**

Comparações com sinal e sem sinal ⊄

sltiu - *also sign-extends the immediate before comparing it to the register.*

3 - Instruções Signed/Unsigned (3) - lb e lh

- **Signed**
  - 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
    (to load into register)
    - Load byte unsigned: **lbu**
    - Load halfword unsigned: **lhu**

3 - Instruções Signed/Unsigned (4) - addi vs addiu

Porquê **addiu** em vez de **addi** ?

addiu, addu e subu - Não geram excepções de *overflow*
(são usadas pelos ponteiros!)

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

```
        .text
        .globl main
main:   li      $t0,0x7FFFFFFF  # max. positive value
#       addi    $t1, $t0, 1     # arithmetic overflow exception
                                # causing program to abort
                                # execution (MARS).
        addiu   $t1, $t0, 1     # 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
$s1 = 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 instruções fazem a mesma coisa, mas o resultado é exactamente o oposto. Porquê?

Porque na realidade elas são diferentes!