

## Aula 7

• Directivas do *Assembler* do MIPS

• Introduz o uso de ponteiros em linguagem C

• Acesso sequencial a elementos de *array* residente em memória:

• Indexado

• Com ponteiros

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira e Silva

### Directivas do *Assembler*:

<b>.ASCIIZ</b> <i>str</i>	Reserva espaço e armazena a string em sucessivas posições de memória; acrescenta o terminador '\0' (NULL)
<b>.SPACE</b> <i>n</i>	Reserva <i>n</i> posições de memória (sem inicialização)
<b>.BYTE</b> <i>b<sub>1</sub>, b<sub>2</sub>, ..., b<sub>n</sub></i>	Reserva espaço e armazena bytes <i>b<sub>1</sub>, b<sub>2</sub>, ..., b<sub>n</sub></i> em sucessivas posições de memória
<b>.WORD</b> <i>w<sub>1</sub>, w<sub>2</sub>, ..., w<sub>n</sub></i>	Reserva espaço e armazena words <i>w<sub>1</sub>, w<sub>2</sub>, ..., w<sub>n</sub></i> em sucessivas posições de memória (cada word em 4 posições)
<b>.ALIGN</b> <i>n</i>	Alinha o próximo item num endereço múltiplo de 2

**Directivas do Assembler - Exemplo:**

```
.DATA # 0x10010000
```

```
STR1: .ASCIIZ "AULA7"
```

```
.ALIGN 2
```

```
ARR1: .WORD 0x1234, MAIN
```

```
VAR1: .BYTE 0x12
```

```
ARR2: .SPACE 2
```

```
STR2: .ASCIIZ " : ) "
```

```
.TEXT # 0x00400000
```

```
.GLOBL MAIN
```

```
MAIN: Se não for possível utilizar a directiva align sempre que se pretender que o endereço subsequente esteja alinhado
```

STR2

ARR2

VAR1

ARR1

STR1

0x10010015	'\0' (0x00)
0x10010014	' ) ' (0x29)
0x10010013	' : ' (0x3A)
0x10010012	????????
0x10010011	????????
0x10010010	0x12
0x1001000F	0x00
0x1001000E	0x40
0x1001000D	0x00
0x1001000C	0x00
0x1001000B	0x00
0x1001000A	0x00
0x10010009	0x12
0x10010008	0x34
0x10010007	????????
0x10010006	????????
0x10010005	'\0' (0x00)
0x10010004	'7' (0x37)
0x10010003	'A' (0x41)
0x10010002	'L' (0x4C)
0x10010001	'U' (0x55)
0x10010000	'A' (0x41)

**Linguagem C****Ponteiros e endereços**

Um **ponteiro** é uma **variável** que contém o endereço de outra **variável**. O acesso à variável pode fazer-se indirectamente através do ponteiro.

**Exemplo**

Se **px** é uma variável (por ex. um inteiro) é um ponteiro. O endereço da variável pode ser obtido através do operador **&**, do seguinte modo:

```
px = &x; // Atribui o endereço de x a px
```

Diz-se que **px** é um ponteiro que aponta para **x**. O operador **&** apenas pode ser utilizado com variáveis e elementos de arrays.

Exemplos de utilizações incorrectas:

```
&5; &(x+1); (sendo x uma variável)
```

## Ponteiros e endereços **operador \***

O operador **\***

Trata o seu operando como um endereço.

Permite o acesso ao endereço para obter o respectivo conteúdo.

Exemplo

```
y = *px; // Atribui o conteúdo do endereço
           // apontado por px a y
```

A sequência:

```
px = &x;
```

```
y = *px;
```

Atribui a **y** o mesmo valor que: **y = x;**

## Ponteiros e endereços **declaração de variáveis**

As variáveis envolvidas têm que ser declaradas.

Para o exemplo anterior, supondo que se tratava de variáveis inteiras:

```
int x, y; // x, y - variáveis do tipo inteiro
```

```
int *px; // ou int* px;
```

A declaração do ponteiro (**int \*px;**) deve ser entendida como uma mnemónica e significa que **px** é um ponteiro e que o conjunto **\*px** é do tipo inteiro.

Exemplos de ponteiros:

```
char *p; // p é um ponteiro para caracter
```

```
double *v; // v é um ponteiro para double
```

## Ponteiros e manipulação em expressões

Exemplo: Supondo que `px` aponta para `x` (`px = &x;`), a expressão `*px = *px + 1;` atribui a `y` o valor de `x` acrescido de 1

Os ponteiros podem igualmente ser utilizados na parte esquerda de uma expressão. Por exemplo, (supondo que `px = &x;`)

```
*px = 0;           // equivalente a x=0
```

ou

```
*px = *px + 1; // equiv. a x = x + 1
```

```
*px += 1;
```

```
(*px)++;
```

## Ponteiros como argumentos de funções

Em C os argumentos das funções são passados por valor (isto é, uma cópia do conteúdo da variável original)

Isso significa que a função chamada só pode alterar o valor da cópia da variável original, isto é, uma função não pode alterar directamente o valor de uma variável da função chamadora

Se tal for necessário, a solução reside no uso de ponteiros

Suponhamos que se pretende implementar uma função para a troca do conteúdo de duas variáveis: `troca(a, b);`.

Exemplo do que se pretende:

Se inicialmente: `a=2` e `b=5`

Após a chamada à função: `a=5` e `b=2`

```
void troca(int, int);
```

```
void main(void)
{
    int a, b;
    (...)

    if(a < b)
        troca(a, b);
    (...)
}
```

```
void troca(int *,int *);
```

```
void main(void)
{
    int a, b;
    (...)

    if(a < b)
        troca(&a, &b);
    (...)
}
```

```
void troca(int x, int y)
{
    int aux;
    aux = x;
    x = y;
    y = aux;
}
```

```
void troca(int *x, int *y)
{
    int aux;
    aux = *x;    // aux = a
    *x = *y;     // a = b
    *y = aux;    // b = aux
}
```

## Ponteiros e arrays

Sejam as declarações

```
int a[10];    // array de inteiros
              // 10 elementos

int *pa;      // ponteiro para um inteiro

int v;        // variável do tipo inteiro
```

A expressão `pa = &a[0];` atribui a `pa` o endereço do 1º elemento do array. Então, a expressão `*pa;` atribui a `v` o valor de `a[0]`

Se `pa` aponta para um dado elemento do array, `pa+1` aponta para o seguinte. Então, em geral, `pa+i` aponta para o elemento `i` do array e `*(pa+i)` refere-se ao seu conteúdo

A expressão `pa = &a[0];` pode também ser escrita como `pa = a;` isto é, o nome do array é o endereço do seu primeiro elemento

## Aritmética de Ponteiros

Se  $pa$  é um ponteiro, então a expressão  $pa++$  incrementa  $pa$  de modo a apontar para o elemento seguinte (seja qual for o tipo de variável para o qual aponta)

Do mesmo modo  $pa = pa + i$ ; incrementa  $pa$  para apontar para  $i$  elementos à frente do elemento actual

**A tradução das expressões anteriores para Assembly tem que ter em conta o tipo de variável para o qual o ponteiro aponta**

Por exemplo, se um inteiro for definido com 4 bytes (32 bits), então a expressão  $pa++$ ; implica adicionar 4 ao valor actual do endereço correspondente

## Acesso sequencial a elementos de um array

O acesso **sequencial** a elementos de um *array* apoia-se em uma de duas estratégias:

Acesso indexado, isto é, endereço a partir do *array* e de um índice que identifica o elemento a que se pretende aceder:

**$f = a[i];$**

Utilização de um ponteiro (endereço armazenado num **registo**) que identifica em cada instante o endereço do elemento a que se pretende aceder:

**$f = *pt;$**  // com  $pt = \text{endereço de } a[i]$  ( i.e.  $pt = \&a[i]$  )

## Acesso indexado

**f = a[i];** // Com  $i \in \mathbb{N}$

Para aceder ao elemento  $i$  do array, o programa começa por calcular o respectivo endereço, a partir do endereço inicial do array:

**endereço do elemento a aceder = endereço inicial do array +**  
**(índice  $\times$  dimensão em bytes de cada posição do array)**

## Acesso por ponteiro

**f = \*pt;**

O endereço do elemento a aceder está armazenado no registo

**endereço do elemento seguinte = endereço actual**  
**+ dimensão em bytes de cada posição do array**

## Dois exemplos de acesso sequencial a arrays:

```
//Exemplo1
int i;
static int array[size];

for (i = 0; i < size; i++)
{
    array[i] = 0;
}
```

Acesso indexado

```
//Exemplo2
int *p;
static int array[size];

for (p=&array[0]; p < &array[size]; p++)
{
    *p = 0;
}
```

Acesso por ponteiro

Também pode ser escrito com `for(p=array; p < array+size; p++)`

//Exemplo1

```
int i;
static int array[size];
for (i = 0; i < size; i++)
{
    array[i] = 0;
}
```

\$t0 |↙| i  
 \$t1 |↙| temp  
 \$t2 |↙| &(array[0])  
 \$a0 |↙| size

```
.DATA
array: .SPACE size * 4
.TEXT
(...)
    la    $t2, array    # $t2 = &(array[0]);
    li    $t0, 0        # i = 0;
loop: bge    $t0, $a0, endf # while (i < size) {
    sll    $t1, $t0, 2    # temp = i * 4;
    addu   $t1, $t2, $t1  # temp = &(array[i])
    sw     $0, 0($t1)     # array[i] = 0;
    addi   $t0, $t0, 1    # i = i + 1;
    j      loop          # }
endf: ...
```

//Exemplo2

```
int *p;
static int array[size];
for (p=&array[0]; p < &array[size]; p++)
{
    *p = 0;
}
```

\$t0 |↙| p  
 \$t1 |↙| &(array[size])  
 \$a0 |↙| size

```
.DATA
array: .SPACE size * 4
.TEXT
(...)
    la    $t0, array    # $t0 = &(array[0]);
    sll    $t1, $a0, 2    # $t1 = size * 4;
    addu   $t1, $t1, $t0  # $t1 = &(array[size]);
loop: bgeu   $t0, $t1, endf # while (p < &array[size]) {
    sw     $0, 0($t0)     # *p = 0;
    addiu  $t0, $t0, 4    # p = p + 1;
    j      loop          # }
endf: ...
```