

## Aula 5

- Diretivas do *assembler* do MIPS
- Introdução à utilização de ponteiros em linguagem C
- Acesso sequencial a elementos de um *array* residente em memória: acesso indexado e acesso com ponteiros

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador

# Diretivas do *Assembler*

- Diretivas são códigos especiais colocados num programa em linguagem *assembly* destinados a instruir o *assembler* para que este execute uma determinada tarefa ou função
- Diretivas **não são instruções** da linguagem *assembly* (não fazem parte do ISA), não gerando qualquer código máquina
- As diretivas podem ser usadas com diversas finalidades:
  - reservar e inicializar espaço em memória para variáveis
  - controlar os endereços reservados para variáveis em memória
  - especificar os endereços de colocação de código e dados na memória
  - definir valores simbólicos
- As diretivas são específicas para um dado *assembler* (em AC1 usaremos as diretivas definidas pelo *assembler* do simulador MARS)

# Diretivas do *Assembler* do MIPS

<b>.ASCIIZ</b>	<b><i>str</i></b>	Reserva espaço e armazena a string <i>str</i> em sucessivas posições de memória; acrescenta o terminador ' <i>\0</i> ' ( <b>NUL</b> )
<b>.SPACE</b>	<b><i>n</i></b>	Reserva <i>n</i> posições consecutivas (endereços) de memória, sem inicialização
<b>.BYTE</b>	<b><i>b</i><sub>1</sub>, <i>b</i><sub>2</sub>, . . . , <i>b</i><sub><i>n</i></sub></b>	Reserva espaço e armazena os <i>bytes</i> <i>b</i> <sub>1</sub> , <i>b</i> <sub>2</sub> , ..., <i>b</i> <sub><i>n</i></sub> em sucessivas posições de memória
<b>.WORD</b>	<b><i>w</i><sub>1</sub>, <i>w</i><sub>2</sub>, . . . , <i>w</i><sub><i>n</i></sub></b>	Reserva espaço e armazena as <i>words</i> <i>w</i> <sub>1</sub> , <i>w</i> <sub>2</sub> , ..., <i>w</i> <sub><i>n</i></sub> em sucessivas posições de memória (cada <i>word</i> em 4 endereços consecutivos)
<b>.ALIGN</b>	<b><i>n</i></b>	Alinha o próximo item num endereço múltiplo de 2 <sup><i>n</i></sup>
<b>.EQV</b>	<b><i>symbol</i>, <i>val</i></b>	Substitui as ocorrências de <b><i>symbol</i></b> no programa por <b><i>val</i></b>

# Diretivas do Assembler - exemplo

<b>.DATA</b>	<b># 0x10010000</b>	0x10010017	??
<b>STR1: .ASCIIZ</b>	<b>"AULA6"</b>	0x10010016	??
<b>ARR1: .WORD</b>	<b>0x1234, MAIN</b>	0x10010015	??
<b>VARB: .BYTE</b>	<b>0x12</b>	<b>VARW 0x10010014</b>	??
<b>.ALIGN</b>	<b>2</b>	0x10010013	?? (unused)
<b>VARW: .SPACE</b>	<b>4 #space for 1 word</b>	0x10010012	?? (unused)
<b>.TEXT</b>	<b># 0x00400000</b>	0x10010011	?? (unused)
<b>.GLOBL</b>	<b>MAIN</b>	<b>VARB 0x10010010</b>	0x12
<b>MAIN:</b>		0x1001000F	0x00
		0x1001000E	0x40
		0x1001000D	0x00
		0x1001000C	0x00
		0x1001000B	0x00
		0x1001000A	0x00
		0x10010009	0x12
		<b>ARR1 0x10010008</b>	0x34
		0x10010007	?? (unused)
		0x10010006	?? (unused)
		0x10010005	'\0' (0x00)
		0x10010004	'6' (0x37)
		0x10010003	'A' (0x41)
		0x10010002	'L' (0x4C)
		0x10010001	'U' (0x55)
		<b>STR1 0x10010000</b>	'A' (0x41)

- Utilizar a diretiva **".align"** sempre que se pretender que o endereço subsequente esteja alinhado
- A diretiva **".word"** alinha automaticamente num endereço múltiplo de 4

## Linguagem C: ponteiros e endereços – o operador &

- Um **ponteiro** é uma **variável que contém o endereço de outra variável** – o acesso à 2ª variável pode fazer-se indiretamente através do ponteiro

- Exemplo:

- **x** é uma variável (por ex. um inteiro) e **px** é um ponteiro. O **endereço da variável x** 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 **erradas**:

```
&5;    &(x+1) ;
```

# Ponteiros e endereços – o operador \*

- O operador "**\***":
  - trata o seu operando como um endereço
  - permite o acesso ao endereço para obter o respetivo conteúdo

- Exemplo:

```
y = *px; // Atribui o conteúdo da variável  
           // apontada por "px" a "y"
```

- A sequência:

```
px = &x; // px é um ponteiro para x
```

```
y = *px; // *px é o valor de x
```

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; (ponteiro para inteiro)
          // Esta declaração apenas reserva o
          // espaço para o ponteiro
```

- A declaração do ponteiro (**int \*px; ou 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 **declarações de ponteiros**:

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

# Manipulação de ponteiros em expressões

- Exemplo: supondo que **px** aponta para **x** (**px = &x;**), a expressão **y = \*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;** **// equivalente a x = x + 1**

**\*px += 1;**                    **// o mesmo que \*px = \*px + 1**

**(\*px) ++;**                    **// o mesmo que \*px = \*px + 1**



# Ponteiros como argumentos de funções

- Em C os argumentos das funções são passados por valor (cópia do conteúdo da variável original)
- Isso significa que a função chamada só pode alterar diretamente o valor da cópia da variável original, isto é, uma função chamada não pode alterar diretamente o valor de uma variável da função chamadora
- Se tal for necessário, a solução reside na utilização de ponteiros
- **Exemplo:** implementar uma função para a troca do conteúdo de duas variáveis (**troca(a, b);**):
  - Se, antes da chamada à função, a=2 e b=5, então
  - Após a chamada à função: a=5 e b= 2

# Ponteiros como argumentos de funções

```
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 "a" com
           // 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 `v = *pa;` atribui a `v` o valor de `a[0]`
- Se `pa` aponta para um dado elemento do *array*, `pa+1` aponta para o seguinte
- Se `pa` aponta para o primeiro elemento do *array*, então `(pa+i)` aponta para o elemento `i` 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 **pa** aponta)
- Do mesmo modo **pa = pa + i;** incrementa **pa** para apontar para **i** elementos à frente do elemento atual
- **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 atual do endereço correspondente

## Exemplo 1

- Analise o código C deste e dos slides seguintes e determine o resultado produzido

```
void main(void)
{
    char s[]="Hello"; // "s" é um array de
                       // caracteres (string)
                       // terminado com o
                       // caractere '\0' (0x00)

    int i = 0;

    while(s[i] != '\0')
    {
        printf("%c", s[i]); // imprime caractere
        i++;
    }
}
```

H	e	l	l	o	\0
---	---	---	---	---	----

## Exemplo 2

```
void main(void)
{
    char s[] = "Hello";
    char *p;    // Declara um ponteiro para
                // carater (reserva espaço)
    p = s;      // Inicializa o ponteiro com
                // endereço inicial do array
    while (*p != '\0')
    {
        printf("%c", *p); // imprime carater
        p++;              // incrementa o ponteiro
    }
}
```

- O ponteiro "p" é usado pelo "printf()" para aceder ao carater a imprimir (\*p)
- O ponteiro é depois incrementado, i.e., fica a apontar para o carater seguinte do *array*

## Exemplo 3

```
void main(void)
{
    char s[] = "Hello";
    char *p1 = s; // p1 = &s[0]
    char *p2 = s; // p2 = &s[0]

    while(*p2 != '\0') p2++;
    while(p1 < p2)
    {
        printf("%c", *p1);
        p1++;
    }
}
```

- Após o primeiro **while** o ponteiro **p2** aponta para o fim da *string* (i.e., para o carater '\0')
- O ponteiro **p1** é usado pelo **printf()** para aceder ao carater a imprimir (**\*p1**); o ponteiro **p1** é incrementado na linha seguinte

## Exemplo 4

```
void main(void)
{
    char s[] = "Hello";
    char *p = s;

    while(*p != '\0')
    {
        printf("%c", *p++);
    }
}
```

- O ponteiro "p" é usado pelo "printf()" para aceder ao carater a imprimir (\*p)
- O ponteiro "p" é incrementado após o acesso ao conteúdo (pós-incremento)
- Esta construção é equivalente à do exemplo 2



## Exemplo 5

```
void main(void)
{
    char s[] = "Hello";
    char *p = s;
    int i;

    for(i = 0; i < 5; i++)
    {
        printf("%c", (*p)++); —————> IIIII
    }
}
```

- O ponteiro "p" é usado pelo "printf()" para aceder ao carater a imprimir (\*p)
- A operação de incremento está a ser aplicada à variável apontada pelo ponteiro
- Neste exemplo o ponteiro "p" nunca é incrementado
- Qual a sequência de caracteres impressa?

## Acesso sequencial a elementos de um *array*

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

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

**f = a[i];**

2. 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 = endereço de a[i] ( i.e. **pt = &a[i]** )

- Estas 2 formas de acesso traduzem-se em **implementações distintas** em *assembly*

# Acesso sequencial a elementos de um *array*

- **Acesso indexado**

- **$f = a[i];$**  // Com  $i \geq 0$
- Para aceder ao elemento "**i**" do *array* "**a**", o programa começa por calcular o respetivo endereço, a partir do endereço inicial do *array*:

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

- **Acesso por ponteiro**

- **$f = *pt;$**
- O endereço do elemento a aceder está armazenado num registo

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

# Exemplos de acesso sequencial a *arrays*

```
// Exemplo 1
int i;
static int array[size];

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

Acesso indexado

```
// Exemplo 2
int *p;
static int array[size];

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

Acesso por ponteiro

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

# Acesso sequencial a arrays – exemplo 1

```
#define SIZE 10
```

```
void main(void) {  
    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 40          # SIZE*4 não é suportado pelo MARS  
      .EQV    SIZE, 10  
      .TEXT  
      .GLOBL  main  
main:  li     $a0, SIZE  
      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:  ...
```

## Acesso sequencial a arrays – exemplo 2

```

#define SIZE 10
void main(void){
    int *p;
    static int array[size];
    for (p=&array[0]; p < &array[size]; p++){
        *p = 0;
    }

```

  

```

        .DATA
array:  .SPACE 40          #SIZE * 4      $t0 > p
        .EQV    SIZE, 10    $t1 > &(array[size])
        .TEXT
        .GLOBL  main
main:    li      $a0, SIZE    #
        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:    ...

```