

# Aula 11:

# Memória e Loops – MIPS

*Disciplina: Organização e Arquitetura de Computadores*

**Prof. Luiz Olmes**

*olmes@unifei.edu.br*

# Nas aulas anteriores...

---

## ▶ **O QUE JÁ ESTUDAMOS?**

- ▶ Evolução das máquinas.
- ▶ Processador.
- ▶ Memória.
- ▶ Barramentos.
- ▶ Introdução à Linguagem de Montagem.
- ▶ Condicionais, diretivas, syscalls.

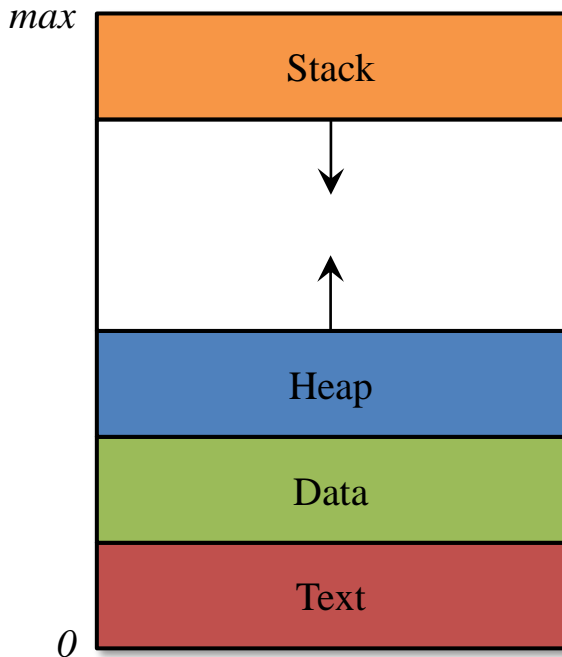
## ▶ **OBJETIVOS:**

- ▶ Memória:
  - ▶ Modelo de memória
  - ▶ Acesso à memória
  - ▶ Modos de endereçamento
- ▶ Loops:
  - ▶ Loop de contagem
  - ▶ Loop com teste de variável
- ▶ Arrays:
  - ▶ Declaração
  - ▶ Percorrendo arrays: `sll`

# Disposição de um processo em memória

---

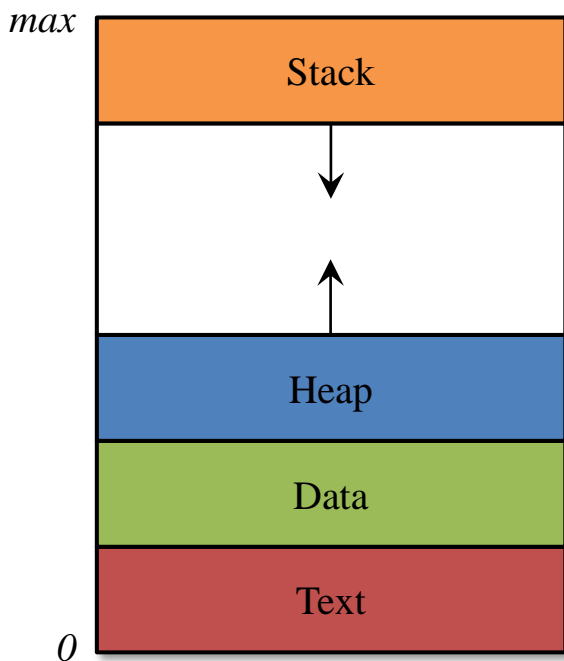
- ▶ Ao ser carregado na memória, um programa é dividido nas seguintes seções:



# Disposição de um processo em memória

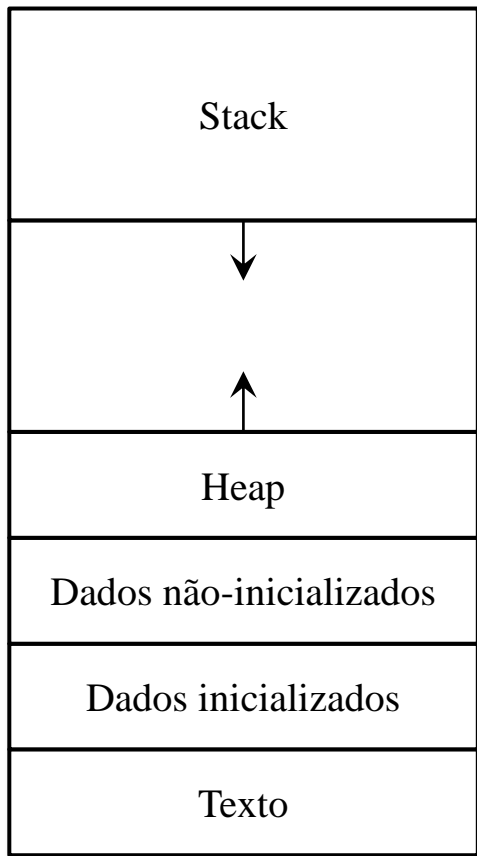
---

- ▶ Ao ser carregado na memória, um programa é dividido nas seguintes seções:



- ▶ **Text**: código executável (instruções de máquina).
- ▶ **Data**: variáveis globais e variáveis inicializadas.
- ▶ **Heap**: memória dinamicamente alocada durante a execução do programa.
- ▶ **Stack**: parâmetros de função, endereços de retorno, variáveis locais, usadas na invocação de funções.

# Disposição de um programa C na memória



```
#include <stdio.h>
#include <stdlib.h>
```

```
int x;
int y = 15;
```

```
int main(int argc, char *argv[])
{
```

```
    int *values;
    int i;
```

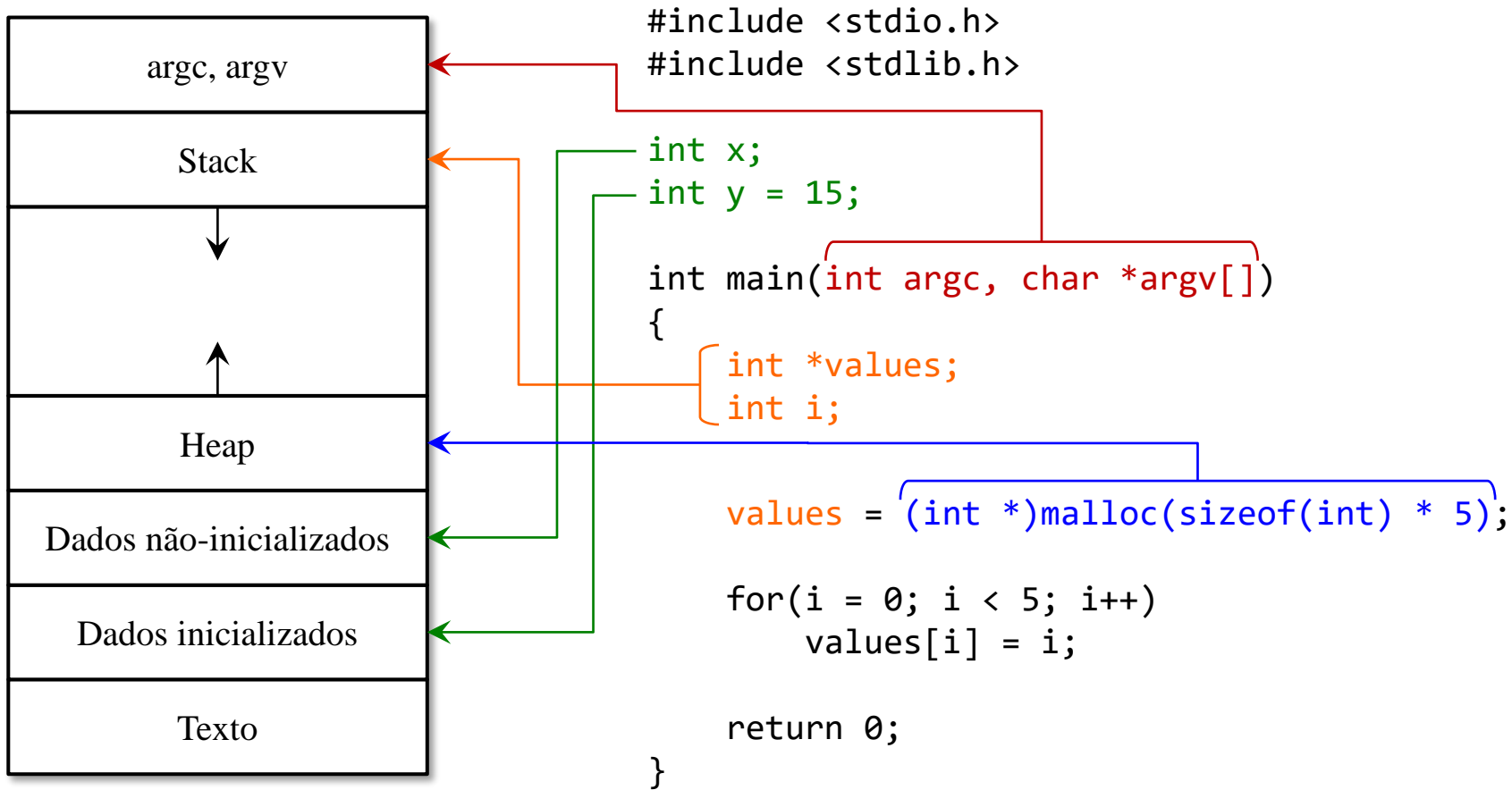
```
    values = (int *)malloc(sizeof(int) * 5);
```

```
    for(i = 0; i < 5; i++)
        values[i] = i;
```

```
    return 0;
```

```
}
```

# Disposição de um programa C na memória



# Modelo de memória

---

- ▶ Para realizar operações com a memória principal, o MIPS utiliza um **modelo de memória linear** (flat memory).
- ▶ O programa enxerga a memória como um espaço de endereçamento **único e contínuo**.
- ▶ Para o programador MIPS, a memória é vista como um conjunto de bytes dispostos um após o outro, como em um array, onde o índice do array é o **endereço** do byte.
- ▶ Uma vez que os bytes da memória possuem endereços próprios, ela é dita ser **endereçável por byte** (byte addressable).

# Modelo de memória

---

- ▶ O MIPS trabalha com **palavras** de memória de **32 bits**.
  - ▶ Existem versões que trabalham com 64 bits.
- ▶ Endereços de 32 bits permitem referenciar **4 giga** de palavras de memória:
  - ▶ De 0x00000000 até 0xFFFFFFFF.
- ▶ Isso não significa que toda a memória estará disponível ao programador:
  - ▶ Parte da memória é reservada pelo Sistema Operacional...
  - ▶ Outra parte é usada pelo subsistema de I/O, etc.
- ▶ A memória no MIPS é dividida nos seguintes **segmentos**:



# Modelo de memória

---

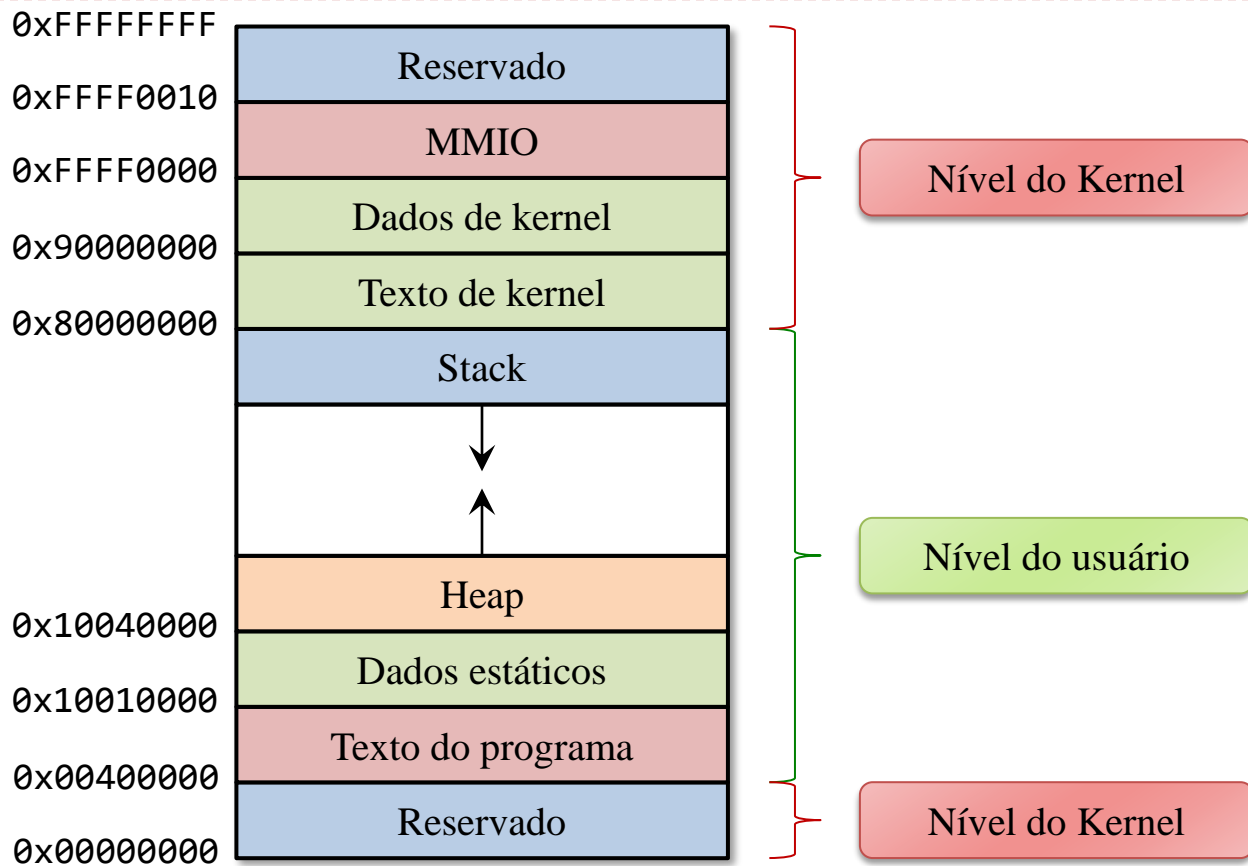
- ▶ **Reservado**: estes endereços de memória não são usados por programas, mas apenas pelo Sistema Operacional.
- ▶ **Texto do programa**: área que armazena código de máquina. Todas as instruções são armazenadas como uma palavra de memória (32 bits ou 4 bytes). Os endereços das instruções são **sempre múltiplos de 4**:
  - ▶ 0x40000000, 0x40000004, etc.
- ▶ **Dados estáticos**: estes dados vêm do segmento de dados do programa (.data). Os tamanhos dos elementos nesta área são definidos quando o programa é criado (assembled e linked), e não se altera durante sua execução.

# Modelo de memória

---

- ▶ **Heap**: memória dinamicamente alocada durante a execução do programa. Dados da heap são globais.
- ▶ **Stack**: memória dinamicamente alocada por subprogramas e chamadas de funções. Variáveis locais são armazenadas nesta área.
- ▶ **Kernel**: estes endereços de memória não são usados por programas, mas apenas pelo Sistema Operacional.
- ▶ **MMIO**: entrada/saída mapeada em memória. Contém registradores mapeados por dispositivos de I/O.

# Modelo de memória



# Acesso à memória

---

- ▶ A memória do MIPS pode ser acessada de 4 diferentes formas:
  - ▶ **Byte**: 8 bits
  - ▶ **Word**: 32 bits
  - ▶ **Halfword**: 16 bits
  - ▶ **Double word**: 64 bits (ponto flutuante)



- ▶ Dentro de um programa, elas são representadas pelas seguintes diretivas:
  - ▶ **.byte**
  - ▶ **.word**
  - ▶ **.half**
  - ▶ **.double**

# Acesso à memória

---

## ► Exemplo:

```
1. .data
2.
3. a:      .byte "a"      # um caractere ocupa 1 byte
4. b:      .half 0x1234    # parte de um endereço de memória
5. c:      .word 4         # um valor inteiro
6.
7. .text
```

## Acesso à memória

---

- ▶ Todo acesso à memória é realizado através das instruções **Load** (traz um dado da memória para um registrador) e **Store** (grava um valor em registrador na memória).
- ▶ Estas instruções podem manipular bytes (**lb**, **sb**), halfwords (**lh**, **sh**), words (**lw**, **sw**) e doubles (**ld**, **sd**).
  - ▶ Foco em word.
- ▶ Além da instrução **la** (Load Address), que carrega o endereço de um label em um registrador.
  - ▶ **la** não carrega valores vindos da memória.
- ▶ Todas são instruções do **tipo I**.

## Acesso à memória

---

- ▶ A instrução `lw` possui muitas variações em seu formato. Entretanto, dois deles são os mais importantes.
- ▶ O formato real desta instrução é: `lw rt, imm(rs)`
  - ▶ `rt`: registrador no qual o valor da memória será armazenado.
  - ▶ `rs`: registrador que contém um endereço de memória.
  - ▶ `imm`: deslocamento a partir do endereço em `rs`.
  - ▶ Significado:  $rt \leftarrow \text{Memória}[rs + imm]$
- ▶ O segundo formato permite que o endereço de um label seja usado para recuperar um valor: `lw rt, label`
  - ▶ Significado:  $rt \leftarrow \text{Memória}[label]$

## Acesso à memória

---

- ▶ A instrução **sw** possui muitas variações em seu formato. Entretanto, dois deles são os mais importantes.
- ▶ O formato real desta instrução é: **sw** *rt*, *imm(rs)*
  - ▶ **rt**: o valor que será armazenado na memória.
  - ▶ **rs**: registrador que contém um endereço de memória.
  - ▶ **imm**: deslocamento a partir do endereço em **rs**.
  - ▶ **Significado**:  $\text{Memória}[\text{rs} + \text{imm}] \leftarrow \text{rt}$
- ▶ O segundo formato permite que o endereço de um label seja usado para armazenar um valor: **sw** *rt*, *label*
  - ▶ **Significado**:  $\text{Memória}[\text{label}] \leftarrow \text{rt}$



## Modos de endereçamento

---

- ▶ As instruções **lw** e **sw** podem ser utilizadas de diferentes maneiras para realizar o acesso à memória.
- ▶ Dentre os vários modos de endereçamento que podem ser realizados no MIPS, quatro deles são comumente utilizados:
  - ▶ Endereçamento por label.
  - ▶ Endereçamento direto por registrador.
  - ▶ Endereçamento indireto por registrador.
  - ▶ Endereçamento por deslocamento baseado em registrador.

## Endereçamento por label

---

- ▶ Muitas vezes, o endereço de uma posição de memória é conhecido, e um label pode ser definido para este endereço.
- ▶ Os dados endereçados através de um label podem existir apenas no segmento de dados do programa (`.data`).
- ▶ Dessa forma, eles não podem ser movidos de lugar e nem ter seu tamanho alterado.
- ▶ Este tipo de endereçamento é aplicado principalmente para constantes que são definidas no programa.

# Endereçamento por label

---

## ► Exemplo:

```
1. .data
2. a:      .word 1
3. b:      .word 2
4. z:      .word 0
5.
6. .text
7. .globl main
8. main:
9.         # endereçamento por label:
10.        # carregando valores nos
11.        # registradores
12.        lw $s1, a
13.        lw $s2, b
14.
15.        # z = a + b
16.        add $s0, $s1, $s2
17.
18.        # endereçamento por label:
19.        # armazena a resposta em z
20.        sw $s0, z
```

# Endereçamento direto por registrador

---

- ▶ Neste tipo de endereçamento, os valores são armazenados diretamente em um registrador através da instrução do **tipo I: li** (Load Immediate).
- ▶ Não é um tipo de endereçamento propriamente dito, porém auxilia a compreender a diferença para o tipo de endereçamento indireto.

## ▶ Exemplo:

```
1. .text
2. .globl main
3. main:
4.     li $s1, 1 # ender. direto
5.     li $s2, 2
6.
7.     add $s0, $s1, $s2
```

## Endereçamento indireto por registrador

---

- ▶ O endereçamento indireto se difere do endereçamento direto no sentido em que um registrador não contém um valor a ser utilizado em operações, mas sim o **endereço de memória** de um **valor** que será utilizado.
- ▶ Supondo que o segmento **.data** de um programa seja o primeiro segmento de dados encontrado pelo assembler, os endereços são numerados a partir de **0x10010000**.
- ▶ Assim, a primeira palavra, seja **a**, estará nesse endereço. A segunda palavra (**b**), estará no endereço **0x10010004**. A terceira palavra (**c**) estará no endereço **0x10010008**, e assim por diante.

# Endereçamento indireto por registrador

---

## ► Exemplo:

```
1. .data
2.     .word 1 # a
3.     .word 2 # b
4.     .word 0 # z
5.
6. .text
7. .globl main
8. main:
9.     # ender. indireto
10.    lui $t0, 0x1001 # t0 = ender a
11.    lw $s1, 0($t0)  # s1 = a
12.
13.    # proxima palavra: +4 bytes
14.    add $t0, $t0, 4
15.
16.    lw $s2, 0($t0)  # s2 = b
17.
18.    # z = a + b
19.    add $s0, $s1, $s2
20.
21.    # proxima palavra: +4 bytes
22.    add $t0, $t0, 4
23.
24.    # armazena o resultado
25.    sw $s0, 0($t0)
```

## Endereçamento por deslocamento baseado em registrador

---

- ▶ Na instrução `lw`, o campo `imediato` representa o `deslocamento` em relação ao registrador para o valor a ser acessado.
- ▶ No caso do endereçamento indireto, o campo imediato é sempre o valor zero.
- ▶ Entretanto, o imediato pode ser usado para especificar o deslocamento, isto é, o quão distante (em bytes) o valor a ser carregado está do registrador de endereço.
- ▶ Este modo de endereçamento é útil quando se trabalha com `arrays`.

# Endereçamento por deslocamento baseado em registrador

---

## ► Exemplo:

```
1. .data
2.     .word 1 # a
3.     .word 2 # b
4.     .word 0 # z
5.
6. .text
7. .globl main
8. main:
9.     # ender. indireto
10.    lui $t0, 0x1001 # t0 = ender a
11.    lw $s1, 0($t0)  # s1 = a
12.
13.    # proxima palavra: +4 bytes
14.    # enderecando atraves de deslocamento:
15.    lw $s2, 4($t0)  # s2 = b
16.
17.    # z = a + b
18.    add $s0, $s1, $s2
19.
20.    # proxima palavra: +4 bytes
21.    # ender. por deslocamento
22.    sw $s0, 8($t0)
```



# Loops

---

- ▶ Da mesma forma que nas linguagens de programação, **laços** de repetição também podem ser implementados em linguagem de montagem.
- ▶ Entretanto, como no caso das estruturas condicionais, os únicos recursos para controlar os loops são **labels** e **desvios**.
- ▶ As duas principais formas de implementação de loops em assembly são através de uma **variável de contagem** (equivalente ao comando **for** das linguagens de programação) e através do **teste** de sua condição através de uma **variável** (equivalente ao comando **while**).

## Loop de contagem

---

- ▶ Um **loop de contagem** é aquele que executa um determinado bloco de código por um **número** fixo, **predefinido** de vezes.
- ▶ Normalmente, sua definição envolve especificar o **valor inicial** do contador, o valor **final** e o **passo** de contagem (incremento / decremento).
- ▶ Por exemplo, em C:

```
1. total = 0;  
2. for(i = 0; i < n; i++)  
3. {  
4.     total = total + i;  
5. }
```

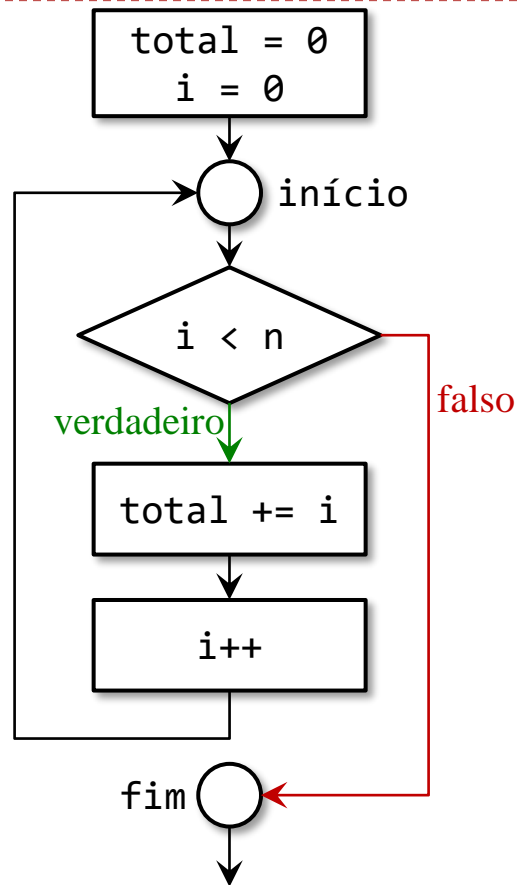
# Loop de contagem

---

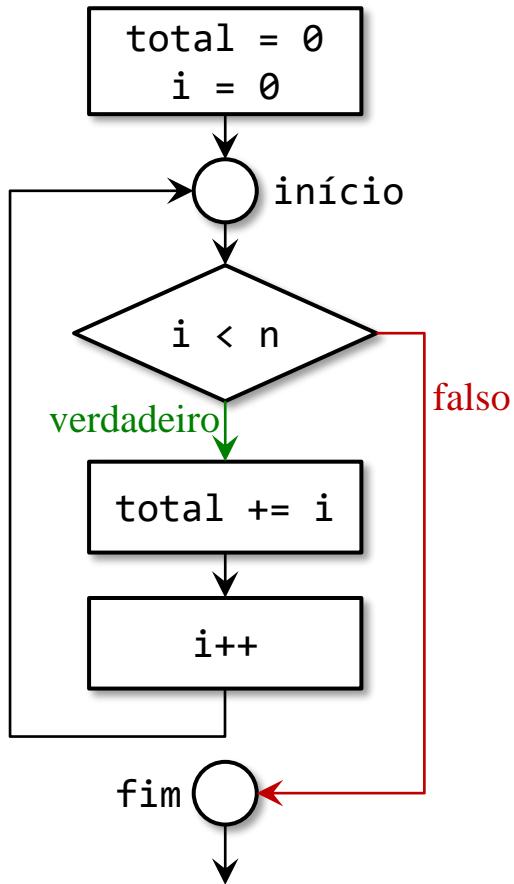
- ▶ Os passos para se implementar um **loop de contagem** em MIPS assembly são:
  1. Definir registradores com os valores de início e fim do loop.
  2. Criar labels de início e fim do loop.
  3. Implementar a verificação de entrada no loop, ou interrompê-lo quando o número máximo de execuções foi atingido.
  4. Implementar o passo e definir o desvio para o início do loop.
  5. Implementar o bloco do loop.

# Loop de contagem

```
1. total = 0;  
2. for(i = 0; i < n; i++)  
3. {  
4.     total += i;  
5. } // fim
```



# Loop de contagem



```
1.      li $s0, 0 # total
2.
3.      # inicializacao
4.      li $t0, 0 # i = 0
5.      li $t1, 5 # n = 5
6.
7. ini:
8.      sub $t2, $t1, $t0 # t2 = t1 - t0
9.      beq $t2, $zero, fim # t2 = 0? Sim -> fim
10.
11.     add $s0, $s0, $t0 # bloco do loop
12.
13.     addi $t0, $t0, 1 # passo
14.     j ini           # volta ao inicio
15.
16. fim:
```

## Loop com teste de variável

---

- ▶ Este tipo de loop executa um mesmo bloco de código por um número **desconhecido** de vezes.
- ▶ Sua condição de parada normalmente envolve o **valor** de uma ou mais **variáveis** que são **calculadas** e **alteradas** no bloco do loop.
- ▶ Por exemplo, em C:

```
1. total = 0;
2. scanf("%d", &i);
3. while(i > 0)
4. {
5.     total = total + i;
6.     scanf("%d", &i);
7. }
```

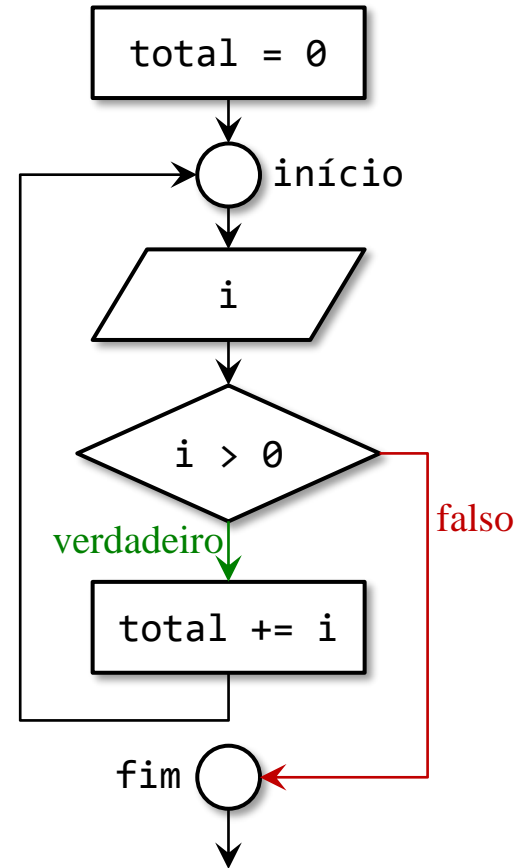
## Loop com teste de variável

---

- ▶ Os passos para se implementar um **loop com teste de variável** em MIPS assembly são:
  1. Definir os valores de teste antes do início do loop.
  2. Criar labels de início e fim do loop.
  3. Implementar a verificação de entrada no loop, ou interrompê-lo quando o teste de variável for falso.
  4. Alterar a variável de comparação e definir o desvio para o início do loop.
  5. Implementar o bloco do loop.

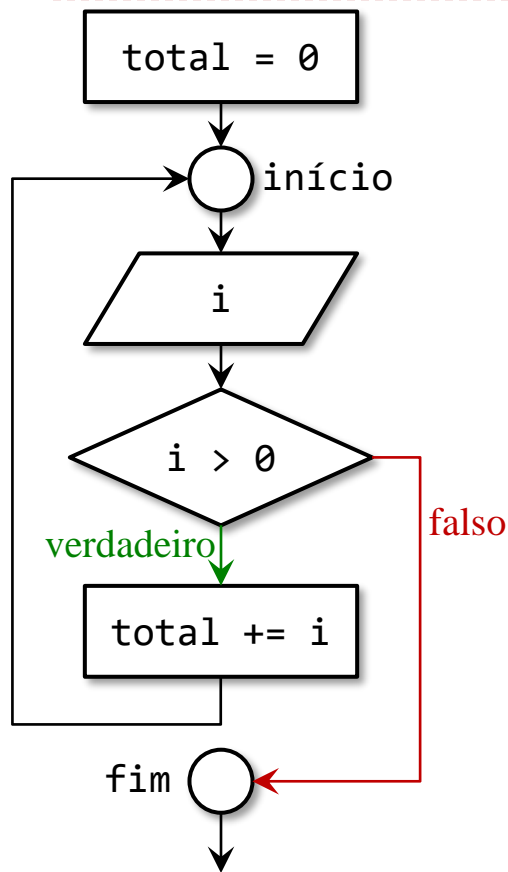
# Loop com teste de variável

```
1. total = 0;
2. scanf("%d", &i);
3. while(i > 0)
4. {
5.     total = total + i;
6.     scanf("%d", &i);
7. }
```





# Loop com teste de variável



```
1.      li $s0, 0 # total
2.
3.  ini:
4.      # leitura
5.      li $v0, 5 # i
6.      syscall
7.
8.      # teste da variavel
9.      sub $t0, $zero, $v0 # t0 = 0 - v0
10.
11.     slt $t1, $t0, $zero # if(t0 < 0) t1 = 1; else t1 = 0
12.     beq $t1, $zero, fim # if(t1 == 0) fim
13.
14.     add $s0, $s0, $v0   # bloco do loop
15.
16.     j ini               # volta ao inicio
17.
18. fim:
```

# Arrays

---

- ▶ No MIPS, um **array** é uma sequência multivalorada de dados armazenados em uma região de **memória contínua**, contendo elementos de mesmo tamanho.
- ▶ As informações mínimas para se definir um array são:
  - ▶ Um endereço inicial: **base**
  - ▶ O tamanho de cada elemento: **size**
  - ▶ O espaço para armazenar os elementos.
- ▶ Para acessar cada elemento do array, deve-se, primeiro, **calcular** o seu **endereço**. A partir de então, o elemento pode ser lido ou escrito na posição calculada.

# Arrays

---

- ▶ O calculo do endereço de um elemento do array é dado pela expressão:

$$\text{enderElem} = \text{baseEnder} + \text{idx} * \text{tam}$$

- ▶ Onde:
  - ▶ **enderElem**: é o endereço do elemento desejado.
  - ▶ **baseEnder**: é o endereço onde se inicia o array.
  - ▶ **idx**: é o índice do elemento.
  - ▶ **tam**: o tamanho de cada elemento.
- ▶ Na expressão apresentada, o primeiro elemento do array encontra-se na posição **zero**.

# Arrays

---

- ▶ A forma mais básica de inicializar um array é declarar os seus elementos diretamente, separados por vírgula, na seção `.data`.

- ▶ **Exemplo:**

```
1. .data
2.
3. arrA: .byte 'h', 'e', 'l', 'l', 'o' # um array de bytes
4.
5. arrB: .word 1, 2, 3, 4, 5, 6, 7, 8 # um array de inteiros
6.
```

# Arrays

---

- ▶ A forma mais básica de inicializar um array é declarar os seus elementos diretamente, separados por vírgula, na seção `.data`.
- ▶ Uma outra maneira é utilizar o formato `M:N`, que declara um array de N elementos, onde todas as posições estão inicializadas com o valor M.

- ▶ **Exemplo:**

```
1. .data
2.
3. arrC: .word 2 : 10 # array de 10 elementos inicializados com o valor 2
4.
5. arrD: .byte 'x' : 5 # array de 5 elementos inicializados com o 'x'
```

# Arrays

---

- ▶ Para um array de palavras (`.word`), cada elemento do array ocupa 4 bytes.
- ▶ Nesse caso, supondo que o elemento de índice 0 está no endereço `0x10010000`, o elemento de índice 1 está no endereço `0x10010004`, o elemento de índice 2 está no endereço `0x10010008`, e assim por diante.

[0]	[1]	[2]	[3]
2	51	7	...
10010000	10010004	10010008	1001000C

- ▶ O índice é incrementado de 1 em 1. O endereço é incrementado de 4 em 4.

# Arrays

---

- ▶ Para percorrer um array, pode-se usar a instrução `sll` (Shift Left Logical) para computar o endereço a partir do índice.
- ▶ A instrução realiza o **deslocamento de bits à esquerda**. Cada bit deslocado multiplica o valor do operando por 2:
  - ▶ 1 bit =  $\times 2$
  - ▶ 2 bits =  $\times 4$
  - ▶ etc...
- ▶ Esta instrução do **tipo R** tem o seguinte formato: `sll rd, rt, shamt`
  - ▶ **rd**: registrador onde o resultado será armazenado.
  - ▶ **rt**: registrador que contém o operando.
  - ▶ **shamt**: quantidade de bits a serem deslocados.

# Arrays

---

[0]	[1]	[2]	[3]
2	51	7	...
10010000	10010004	10010008	1001000C


- ▶ Índice 0:  $0x10010000$  (base do array)
- ▶ Índice 1:  $\text{base} + 4 \times 1 = 0x10010000 + 4 = 0x10010004$
- ▶ Índice 2:  $\text{base} + 4 \times 2 = 0x10010000 + 8 = 0x10010008$
- ▶ Índice i:  $\text{base} + 4 \times i$



# Arrays

---

[0]	[1]	[2]	[3]
2	51	7	...
10010000	10010004	10010008	1001000C

- ▶ Índice 0:  $0x10010000$  (base do array)
- ▶ Índice 1:  $\text{base} + 4 \times 1 = 0x10010000 + 4 = 0x10010004$
- ▶ Índice 2:  $\text{base} + 4 \times 2 = 0x10010000 + 8 = 0x10010008$
- ▶ Índice i:  $\text{base} + 4 \times i$  

# Arrays

---

- ▶ **Exemplo:** dado um array com 5 elementos, mostrar a soma de seus elementos. Percorrer o array com o auxílio da instrução `sl1`.
- ▶ Para você treinar:
- ▶ **Exercício 1:** ler um array com 10 elementos. Percorrer o vetor e mostrar os valores maiores que 5.
- ▶ **Exercício 2:** ler um array com 10 elementos. Percorrer o vetor e mostrar quantos elementos maiores que 5 estão presentes no vetor.
- ▶ **Exercício 3:** ler um array com 10 elementos. Percorrer o vetor e mostrar o maior elemento digitado e o índice em que se encontra.

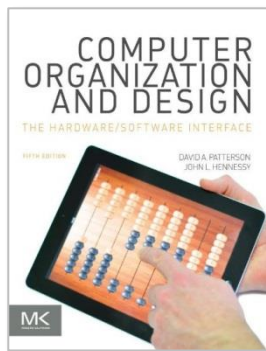
# Dúvidas?

---



# Sugestão de Estudo

- ▶ **Capítulo 2 e Apêndice A:**  
PATTERSON, D. A.; HENESSY, J. L.  
*Computer Organization and Design*. 2013.



- ▶ **Capítulo 7:** TANENBAUM, A. S.; TODD, A. *Organização Estruturada de Computadores*, 2007



# Aula 11:

# Memória e Loops – MIPS

*Disciplina: Organização e Arquitetura de Computadores*

**Prof. Luiz Olmes**

*olmes@unifei.edu.br*