

# Microprocessadores

Hugo Marcondes  
[hugo.marcondes@ifsc.edu.br](mailto:hugo.marcondes@ifsc.edu.br)

Aula 02

Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina

## Programação Assembly: Documentação!

```
# Author:    your name
# Date:      current date
# Description: high-level description of your program
```

```
.data
    (constant and variable definitions)
```

```
.text
```

```
# Section 1: what this part does
# Pseudocode:
#   (algorithm in pseudocode)
# Register mappings:
#   (mapping from pseudocode variables to registers)
```

Comentários na mesma linha do código devem ajudar a esclarecer o pseudocódigo

<sup>2</sup> IFSC - Departamento Acadêmico de Eletrônica

## Expressões lógicas e aritméticas

- Composicionalidade:
  - Técnica útil e poderosa: sintetiza o código, facilita entendimento, reuso.
- Expressões **matemáticas** convencionais são **composicionais!**
  - Ex:  $a * b + [c/d] \dots$  > (Como em linguagem C);
- Instruções em **assembly não!**
  - `mult $t0 [addi $t1, $t1, 3] ????` – Não permitido!
  - Precisa sequenciar as instruções...

<sup>3</sup> IFSC - Departamento Acadêmico de Eletrônica

## Sequenciando Instruções:

- Como sequenciar instruções para resolver expressões lógicas e aritméticas?
- Considerações:
  - Número limitado de instruções
    - MIPS - opera dois registradores ou registrador e imediato
    - Nem toda instrução possui a variante com imediatos
    - Uso do `li` para carregar constantes (tb maiores que 16b)
  - Número limitado de registradores
    - Pode ser necessário ler e armazenar resultados intermediários na memória (comum em programas mais complexos);

<sup>4</sup> IFSC - Departamento Acadêmico de Eletrônica

## Sequenciando Instruções



- Estratégia: Decompor a expressão.
- Separar a expressão em pequenas sub-expressões.
  - Respeite agrupamentos (entre parênteses) e precedência de operadores (assumam a precedência vista em linguagem C);
- Traduza cada sub-expressão e salve os resultados intermediários;
- Combine os resultados!

### Example

```
# Pseudocode:
# d = (a+b) * (c+4)
# Register mappings:
# a: t0, b: t1, c: t2, d: t3
addi $t4, $t0, $t1 # tmp1 = a+b
addi $t5, $t2, 4    # tmp2 = c+4
mul  $t3, $t4, $t5  # d = tmp1 * tmp2
```

```
tmp1 = a+b
tmp2 = c+4
d = tmp1 * tmp2
```

5 IFSC - Departamento Acadêmico de Eletrônica

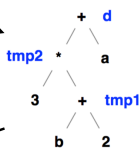
## Sequenciando Instruções:



- Estratégia: Análise e tradução
- Faça a análise sintática da expressão transformando-a em uma árvore de sintaxe abstrata
- Atravesse a árvore em pós-ordem, armazenando o resultado das sub-árvores em registradores temporários
- Essa é a estratégia utilizada pelo compilador!

### Example

```
# Pseudocode:
# c = a + 3*(b+2)
# Register mappings:
# a: t0, b: t1, c: t2
addi $t3, $t1, 2 # tmp1 = b+2
mul  $t4, $t3, 3 # tmp2 = 3*tmp1
add  $t2, $t0, $t4 # c = a + tmp2
```



6 IFSC - Departamento Acadêmico de Eletrônica

## Otimizando o uso de registradores



- Geralmente é possível utilizar poucos registradores fazendo a acumulação do resultado.

```
# Pseudocode:
# c = a + 3*(b+2)
# Register mappings:
# a: $t0, b: $t1, c: $t2
# tmp1: $t3, tmp2: $t4
```

```
# tmp1 = b+2
# tmp2 = 3*tmp1
# c = a + tmp2
addi $t3, $t1, 2
mul  $t4, $t3, 3
add  $t2, $t0, $t4
```



```
# c = b+2
# c = 3*c
# c = a + c
addi $t2, $t1, 2
mul  $t2, $t2, 3
add  $t2, $t0, $t2
```

7 IFSC - Departamento Acadêmico de Eletrônica

## Exercício:



```
# Pseudocode:
# d = a - 3 * (b + c + 8)
# Register mappings:
# a: t0, b: t1, c: t2, d: t3
```

```
addi $t3, $t2, 8 # d = b + c + 8
add  $t3, $t1, $t3
li   $t4, 3      # d = 3 * d
mul  $t3, $t4, $t3
sub  $t3, $t0, $t3 # d = a - d
```

8 IFSC - Departamento Acadêmico de Eletrônica

## Expressões lógicas



- Utilizadas em estruturas de controle:

- branches (if-else);
- loops (while, for);

### Logical expressions

- values: True, False
- boolean operators: not (!), and (&&), or (||)
- relational operators: ==, !=, >, >=, <, <=

- No MIPS, Falso = 0; Verdadeiro = 1;
- Operações lógicas não relacionais são **bit-a-bit**, não booleanas!

---

---

---

---

---

---

---

---

## Instruções lógicas



- Operações de deslocamento (shift)
- Deslocam todos os bits de uma palavra para esquerda ou direita, preenchendo os bits vazios com zero (não cíclico);
- São instruções do tipo R (registrador)
- A quantidade de bits a serem deslocados é especificada pelo campo shamt (shift amount)

---

---

---

---

---

---

---

---

## Instruções lógicas



### • shift left logical (sll)

```
sll $t2, $s0, 4 # $t2 ← $s0 << 4
```

#### Conteúdo:

\$s0	0110	1000	1111	0000	0111	0110	1111	1111
	↙	↙	↙	↙	↙	↙	↙	
\$t2	1000	1111	0000	0111	0110	1111	1111	0000

---

---

---

---

---

---

---

---

## Instruções lógicas



### • shift right logical (srl)

```
srl $t2, $s0, 8 # $t2 ← $s0 >> 8
```

#### Conteúdo:

\$s0	0110	1000	1111	0000	0111	0110	1111	1111
	↘							
\$t2	0000	0000	0110	1000	1111	0000	0111	0110

---

---

---

---

---

---

---

---

• **shift right arithmetic (sra)**

- Desloca bits à direita, **preservando** o sinal (compl. a 2)
- Deslocamento de n bits corresponde à divisão por  $2^n$
- Dica: para divisões com valores de tipo integer

```
sra $t2, $s0, 4 # $t2 ← $s0 >> 4
```

Conteúdo:

```
$s0 1110 1000 1111 0000 0111 0110 1111 1111
    |
    |
    v
$t2 1111 1110 1000 1111 0000 0111 0110 1111
```

```
and $t1, $t2, $t3 # $t1 = $t2 & $t3 (bitwise and)
or  $t1, $t2, $t3 # $t1 = $t2 | $t3 (bitwise or)
xor $t1, $t2, $t3 # $t1 = $t2 ^ $t3 (bitwise xor)
```

Example: 0110 'op' 0011

<pre>  1010 and 0011 ----   0010 1 iff both are 1</pre>	<pre>  1010 or  0011 ----   1011 1 iff either is 1</pre>	<pre>  1010 xor 0011 ----   1001 1 iff exactly one 1</pre>
---	--	--

Immediate variants

```
andi $t1, $t2, 0x0F # $t1 = $t2 & 0x0F (bitwise and)
ori  $t1, $t2, 0xF0 # $t1 = $t2 | 0xF0 (bitwise or)
xori $t1, $t2, 0xFF # $t1 = $t2 ^ 0xFF (bitwise xor)
```

- Para **and**, **or**, **xor**:
  - Equivalente** quando **Falso=0** e **Verdadeiro=1**;
  - Não equivalente** quando **Falso=0** e **Verdadeiro≠0** (como é em C);
- Not:
  - Macro fornecida pelo MARS: não equivalente ao “**not** lógico”;
  - Inverte cada um dos bits:
    - Se Verdadeiro=1, not[Verdadeiro]=0xFFFFFFE;

How can we implement logical not?

```
xori $t1, $t2, 1 # $t1 = not $t2 (logical not)
```

Logical expressions

- values: True, False
- boolean operators: not (!), and (&&), or (||)
- relational operators: ==, !=, >, >=, <, <=

```
seq $t1, $t2, $t3 # $t1 = $t2 == $t3 ? 1 : 0
sne $t1, $t2, $t3 # $t1 = $t2 != $t3 ? 1 : 0
sge $t1, $t2, $t3 # $t1 = $t2 >= $t3 ? 1 : 0
sgt $t1, $t2, $t3 # $t1 = $t2 > $t3 ? 1 : 0
sle $t1, $t2, $t3 # $t1 = $t2 <= $t3 ? 1 : 0
slt $t1, $t2, $t3 # $t1 = $t2 < $t3 ? 1 : 0

slti $t1, $t2, 42 # $t1 = $t2 < 42 ? 1 : 0
```

## Exercício:



```
# Pseudocode:  
# c = (a < b) || ((a+b) == 10)  
# Register mappings:  
# a: t0, b: t1, c: t2
```

```
add $t3, $t0, $t1 # tmp = a+b  
li $t4, 10 # tmp = tmp == 10  
seq $t3, $t3, $t4  
slt $t2, $t0, $t1 # c = a < b  
or $t2, $t2, $t3 # c = c | tmp
```

---

---

---

---

---

---

---

---

## Exercício:



```
# Pseudocode:  
# c = (a < b) && ((a+b) % 3) == 2  
# Register mappings:  
# a: t0, b: t1, c: t2  
  
# tmp1: t3, tmp2: t4
```



---

---

---

---

---

---

---

---