

ARQUITETURA DO MIPS E CÓDIGO ASSEMBLY I

Profa. Monica Magalhães Pereira
monicapereira@dimap.ufrn.br

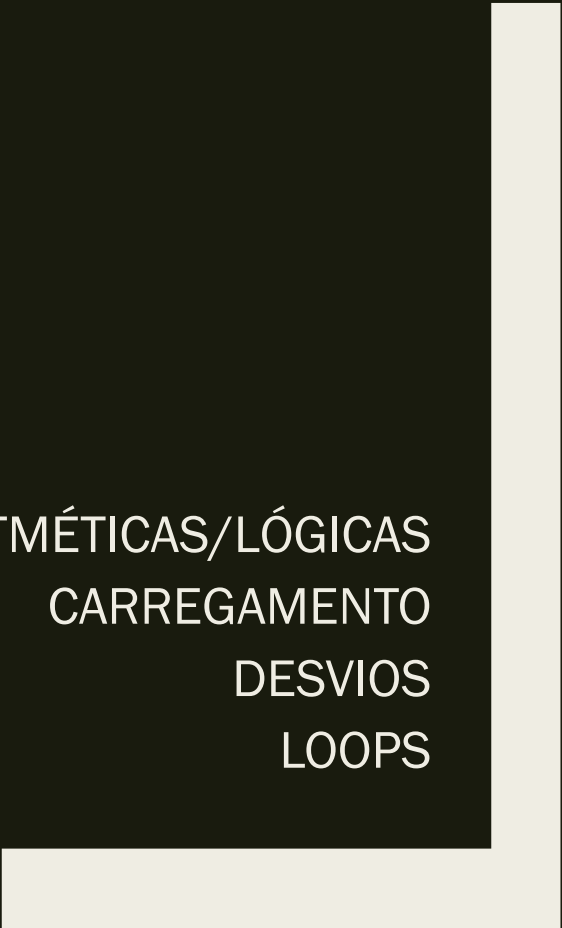
Plano de Aula

- Serão apresentadas as instruções em **assembly MIPS** que podem ser utilizadas para a construção de programas
- Instruções para
 - *Utilizar variáveis*
 - *Realizar operações*
 - *Criar rotinas de laços e condicionais*

Responda o formulário

<https://forms.gle/Fj4ZAXhfViZ5do5LA>

CONJUNTO DE INSTRUÇÕES DO MIPS



ARITMÉTICAS/LÓGICAS
CARREGAMENTO
DESVIOS
LOOPS

Instruções MIPS

Aritmética

■ Soma

- `add $t0, $t1, $t2` $\# t0 = t1 + t2$
- `addi $t2, $t3, 50` $\# t2 = t3 + 50$
- `addi $t2, $t3, -30` $\# t2 = t3 - 30$
- *Instrução imediata evita um carregamento de operando!*

■ Subtração

- `sub $t0, $t1, $t2` $\# t0 = t1 - t2$
- `subi` – não existe
- *Por que não existe subi?*

Instruções MIPS

Aritmética

- Constante Zero
- Registrador 0 (\$zero) é uma constante 0
- \$zero não pode ser reescrito
- Útil para várias operações, por exemplo, mover entre dois registradores

– *add \$t1, \$t2, \$zero* *#t1 = t2*

Instruções MIPS

Operações Lógicas

- Shift Left Logical
 - `sll $t2, $s0, 4` # $\$t2 = \$s0 \ll 4$
- Shift Right Logical
 - `srl $t2, $s0, 4` # $\$t2 = \$s0 \gg 4$
- AND bit a bit
 - `and $t0, $t1, $t2` # $\$t0 = \$t1 \& \$t2$
- OR bit a bit
 - `or $t0, $t1, $t2` # $\$t0 = \$t1 \mid \$t2$
- O que cada operação faz?

Instruções MIPS

Operações Lógicas

- Outros
 - *andi*
 - *ori*
 - *nor*

Instruções MIPS

Carregamento

- *addi \$t1, \$zero, 25* # *t1* = 25
- *lui \$t1, 25* # *t1* = “25”₁₀ * 2¹⁶
- *De acordo com o comentário, o que o lui faz?*

Instruções MIPS

Carregamento

- `addi $t1, $zero, 25` $\# t1 = 2$
- `lui $t1, 2` $\# t1 = "2"_{10} * 2^{16}$
- De acordo com o comentário, o que o `lui` faz?

Load upper immediate : carrega o imediato nos bits mais significativos (*upper*) – demais bits são zerados

Instruções MIPS

Carregamento

■ Ler e Escrever na Memória

- *Ler (load) – Carrega um valor que está armazenado em uma posição de memória em um registrador*

lw \$t1, offset(\$t0)

\$t1 recebe valor de memória armazenado na posição [\$t0+offset]

- *lw – word (palavra)*
- *lb – byte*

Instruções MIPS

Carregamento

■ Ler e Escrever na Memória

- *Escrever (store) – Armazena um valor que está em um registrador em uma posição na memória*

`sw $t1, offset($t0)`

a posição de memória de endereço [$\$t0 + \text{offset}$] recebe o valor que está no registrador $\$t1$

- *sw – word (palavra)*
- *sb – byte*

Instruções MIPS

Desvio incondicional

- *jump to Label*

- *j LABEL*

O que é LABEL?

- Ex:

...

operação 0

j PARA_AQUI

operação 1

operação 2

PARA_AQUI: operação 3

operação 1 e *operação 2* não serão executadas, pois serão puladas

Instruções MIPS

Desvio incondicional

- *Jump and link*
 - *jal EndereçoProcedimento*
 - *Pula para a instrução em EndereçoProcedimento e grava o endereço da próxima instrução em \$ra*
- *Jump register*
 - *jr \$ra*
 - *Pula de volta para seguir o fluxo de instruções anterior*

Endereço	Instrução
...	...
10	jal Aqui
11	add \$t1, \$t2, \$t3
...	...

\$ra = 11

Endereço	Instrução
35	Aqui: add \$t1, \$t2, \$t3
36	add \$t1, \$t1, \$t1
37	sub \$t1, \$t1, \$t4
38	jr \$ra

Instruções MIPS

Desvio incondicional

Endereço	Instrução
...	...
10	jal Aqui
11	add \$t1, \$s4, \$t3
...	...
...	...
...	...
...	...
78	jal Aqui
79	
...	...

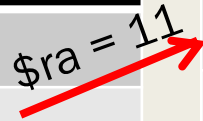
Endereço	Instrução
35	Aqui: add \$t1, \$t2, \$t3
36	add \$t1, \$t1, \$t1
37	sub \$s4, \$t1, \$t4
38	jr \$ra

Instruções MIPS

Desvio incondicional

Endereço	Instrução
...	...
10	jal Aqui
11	add \$t1, \$s4, \$t3
...	...
...	...
...	...
...	...
78	jal Aqui
79	
...	...

\$ra = 11




Endereço	Instrução
35	Aqui: add \$t1, \$t2, \$t3
36	add \$t1, \$t1, \$t1
37	sub \$s4, \$t1, \$t4
38	jr \$ra

Instruções MIPS

Desvio incondicional

Endereço	Instrução
...	...
10	jal Aqui
11	add \$t1, \$s4, \$t3
...	...
...	...
...	...
...	...
78	jal Aqui
79	
...	...

\$ra = 11



Endereço	Instrução
35	Aqui: add \$t1, \$t2, \$t3
36	add \$t1, \$t1, \$t1
37	sub \$s4, \$t1, \$t4
38	jr \$ra

Instruções MIPS

Desvio incondicional

Endereço	Instrução	Endereço	Instrução
...	...	35	Aqui: add \$t1, \$t2, \$t3
10	jal Aqui	36	add \$t1, \$t1, \$t1
11	add \$t1, \$s4, \$t3	37	sub \$s4, \$t1, \$t4
...	...	38	jr \$ra
...	...		
...	...		
...	...		
78	jal Aqui		
79			
...	...		

\$ra = 11

\$ra = 79

Instruções MIPS

Desvio incondicional

Endereço	Instrução	Endereço	Instrução
...	...	35	Aqui: add \$t1, \$t2, \$t3
10	jal Aqui	36	add \$t1, \$t1, \$t1
11	add \$t1, \$s4, \$t3	37	sub \$s4, \$t1, \$t4
...	...	38	jr \$ra
...	...		
...	...		
...	...		
78	jal Aqui		
79			
...	...		

\$ra = 11

\$ra = 79

Instruções MIPS

Tomada de Decisão

Qual seria o algoritmo do exemplo?

■ *Branch if equal*

- `beq $s3, $s4, LABEL`
- `bne $s3, $s4, LABEL1`

Vá para *LABEL* se “`$s3 == $s4`”

Vá para *LABEL1* se “`$s3 != $s4`”

■ *Exemplo*

```
beq $s1, $s2, L1
j L2
L1:
    addi $s1, $s1, 1
    j EXIT
L2:
    addi $s1, $s1, 2
EXIT: ...
```

Instruções MIPS

Tomada de Decisão

■ *Branch if equal*

- `beq $s3, $s4, LABEL` # Vá para **LABEL** se “\$s3 == \$s4”
- `bne $s3, $s4, LABEL1` # Vá para **LABEL1** se “\$s3 != \$s4”

■ *Exemplo*

```
beq $s1, $s2, L1      #salte para L1 se ($s1=$s2)
j L2                  #senão, salte L2
L1:
    addi $s1, $s1, 1    #i++
    j EXIT              #salte EXIT
L2:
    addi $s1, $s1, 2    #i+=2
EXIT: ...
```

Instruções MIPS

Tomada de Decisão

Qual seria o algoritmo do exemplo?
É o mesmo do ex. anterior?

■ *Branch if equal*

- `beq $s3, $s4, LABEL` # Vá para *LABEL* se “\$s3 == \$s4”
- `bne $s3, $s4, LABEL1` # Vá para *LABEL1* se “\$s3 != \$s4”

■ *Exemplo*

```
bne $s1, $s2, Else
addi $s1, $s1, 1
j exit
Else:
    addi $s1, $s1, 2
EXIT: ...
```

Instruções MIPS

Tomada de Decisão

■ *Branch if equal*

- `beq $s3, $s4, LABEL` # Vá para *LABEL* se “\$s3 == \$s4”
- `bne $s3, $s4, LABEL1` # Vá para *LABEL1* se “\$s3 != \$s4”

■ *Exemplo*

```
bne $s1, $s2, Else
addi $s1, $s1, 1
j exit
Else:
    addi $s1, $s1, 2
EXIT: ...
```

SIM, e está mais
compacto

Instruções MIPS

Tomada de Decisão

■ DICA:

- *De modo geral, o código será mais eficiente se testarmos a condição oposta ao desvio no lugar do código que realiza a parte “then” subsequente do “if”*

OU SEJA

- *De modo geral, o código será mais eficiente se testarmos primeiro o “else”.*

Instruções MIPS

Tomada de Decisão

■ *Branch if equal*

- `beq $s3, $s4, LABEL` # Vá para **LABEL** se “\$s3 == \$s4”

■ *Exemplo*

```
beq $s1, $s2, L1      #branch if (k == 1)
subi $s2, $s2, 1      #l--
L1: addi $s1, $s1, 1   #k++
```

- **k** está em **\$s1** e **l** está em **\$s2**
- Como poderíamos escrever o algoritmo?

Durante a execução, a sequência do código é seguida mesmo com o label

Instruções MIPS

Tomada de Decisão

■ *Branch if equal*

- `beq $s3, $s4, LABEL` # Vá para **LABEL** se “\$s3 == \$s4”

■ *Exemplo*

```
beq $s1, $s2, L1      #branch if (k == l)
subi $s2, $s2, 1      #l--
L1: addi $s1, $s1, 1   #k++
```

■ k está em \$s1 e l está em \$s2

```
if ( k != l ) {
    l--
}
k++
```

Instruções MIPS

Teste de Igualdade

- *Set on less than*
 - `slt $t0, $s3, $s4`
 - `$t0` será “1” se “`$s3 < $s4`”
 - `$t0` será “0”, cc
- Muito utilizado juntamente com o **beq** na tomada de decisão em desigualdades

■ *Exemplo*

```
slt $s1, $s2, $s3
bne $s1, $zero, Else
addi $s2, $s2, 1
j EXIT
Else:
    addi $s2, $s2, 2
EXIT: ...
```

Instruções MIPS

Teste de Igualdade

- *Set on less than*
 - `slt $t0, $s3, $s4`
 - `$t0` será “1” se “`$s3 < $s4`”
 - `$t0` será “0”, cc
- Muito utilizado juntamente com o **beq** na tomada de decisão em desigualdades

■ Exemplo

```
slt $s1, $s2, $s3
bne $s1, $zero, Else
addi $s2, $s2, 1
j EXIT
Else:
    addi $s2, $s2, 2
EXIT: ...
```

Como seria o algoritmo?

Instruções MIPS

Teste de Igualdade

■ Exemplo

```
slt $s1, $s2, $s3
bne $s1, $zero, Else
addi $s2, $s2, 1
j EXIT
Else:
    addi $s2, $s2, 2
EXIT: ...
```

Como seria o algoritmo?

```
if ( s2 >= s3 ){
    s2 = s2 + 1
}
else{
    s2 = s2 + 2
}
```

Instruções MIPS

Teste de Igualdade

- Precisa existir “set on **more** than”?
- Outros
 - *slti* (para imediato)

Instruções MIPS

Loop

- Como poderíamos escrever o seguinte algoritmo?

```
inteiro i = 0
inteiro j = 15
enquanto ( i < 10 ) {
    j = j + 5
    i = i + 1
}
```


Instruções MIPS

Loop

- Como poderíamos escrever o seguinte algoritmo?

```
li $t0, 10      # constante 10
li $t1, 0       # contador do laço i
li $t2, 15      # variável j
loop:
    beq $t1, $t0, end    # se t1 == 10, o código acaba
    addi $t2, $t2, 5     # j = j + 5
    addi $t1, $t1, 1     # i = i + 1
    j loop
end:
...
```

Instruções MIPS

Pseudoinstruções

- Elas são reconhecidos pelo assembler, mas traduzidas em pequeno conjunto de instruções de máquina.

move \$t0, \$t1	se torna	add \$t0, \$zero, \$t1	\$t0 = \$t1
blt \$t0, \$t1, L	se torna	slt \$at, \$t0, \$t1 bne \$at, \$zero, \$L	Jump para L se \$t0 < \$t1

Algumas observações

1. O código em linguagem de alto nível não tem a mesma quantidade de instruções do que o código assembly.
2. Muitas vezes, é necessário ter instruções intermediárias em assembly para poder executar a instrução em alto nível

if (s2 < s3)



```
slt $s1, $s2, $s3  
bne $s1, $zero, Else
```

Para treinar

- O que faz trecho de programa abaixo?

L1:

```
add $s0, $s0,  
$t1
```

```
addi $t0, $t0, 1
```

```
bne $t2, $t0, L1
```

```
EXIT: ...
```

Nome	Sintaxe	Significado
<u>Add</u>	<u>add</u> \$1,\$2,\$3	\$1 = \$2 + \$3 (<u>signed</u>)
Sub	sub \$1,\$2,\$3	\$1 = \$2 - \$3 (<u>signed</u>)
<u>Add Immediate</u>	<u>addi</u> \$1,\$2,CONST	\$1 = \$2 + CONST (<u>signed</u>)
Set <u>on less than</u>	<u>slt</u> \$1,\$2,\$3	<u>if</u> (\$2 < \$3) \$1 = 1 <u>else</u> \$1 = 0
<u>Branch on not equal</u>	<u>bne</u> \$1,\$2, <u>Label</u>	<u>if</u> (\$1 != \$2) goto <u>Label</u>
<u>Branch on equal</u>	<u>beq</u> \$1,\$2, <u>Label</u>	<u>if</u> (\$1 == \$2) goto <u>Label</u>
<u>Jump</u>	<u>j</u> <u>Label</u>	goto <u>Label</u>

Para treinar

- Implementar em assembly MIPS

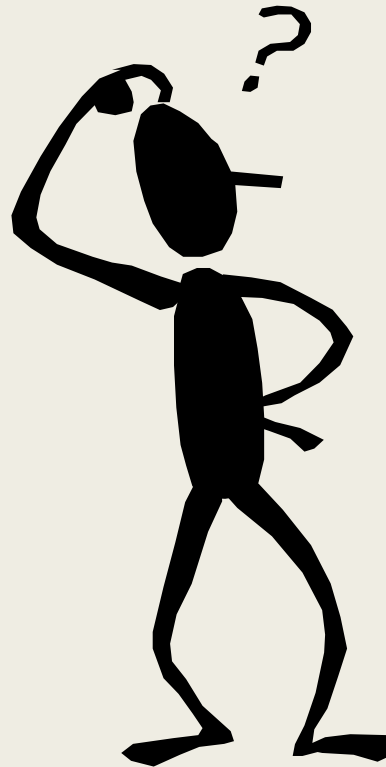
- $\$t0 = x$

- $\$t1 = y$

```
se (y==0) faça
    x = x+y-1;
senão
    x = x-y+1;
fim se
```

Nome	Sintaxe	Significado
<u>Add</u>	<u>add</u> \$1,\$2,\$3	\$1 = \$2 + \$3 (<u>signed</u>)
<u>Sub</u>	<u>sub</u> \$1,\$2,\$3	\$1 = \$2 - \$3 (<u>signed</u>)
<u>Add Immediate</u>	<u>addi</u> \$1,\$2,CONST	\$1 = \$2 + CONST (<u>signed</u>)
<u>Set on less than</u>	<u>slt</u> \$1,\$2,\$3	<u>if</u> (\$2 < \$3) \$1 = 1 <u>else</u> \$1 = 0
<u>Branch on not equal</u>	<u>bne</u> \$1,\$2, <u>Label</u>	<u>if</u> (\$1 != \$2) goto <u>Label</u>
<u>Branch on equal</u>	<u>beq</u> \$1,\$2, <u>Label</u>	<u>if</u> (\$1 == \$2) goto <u>Label</u>
<u>Jump</u>	<u>j</u> <u>Label</u>	goto <u>Label</u>

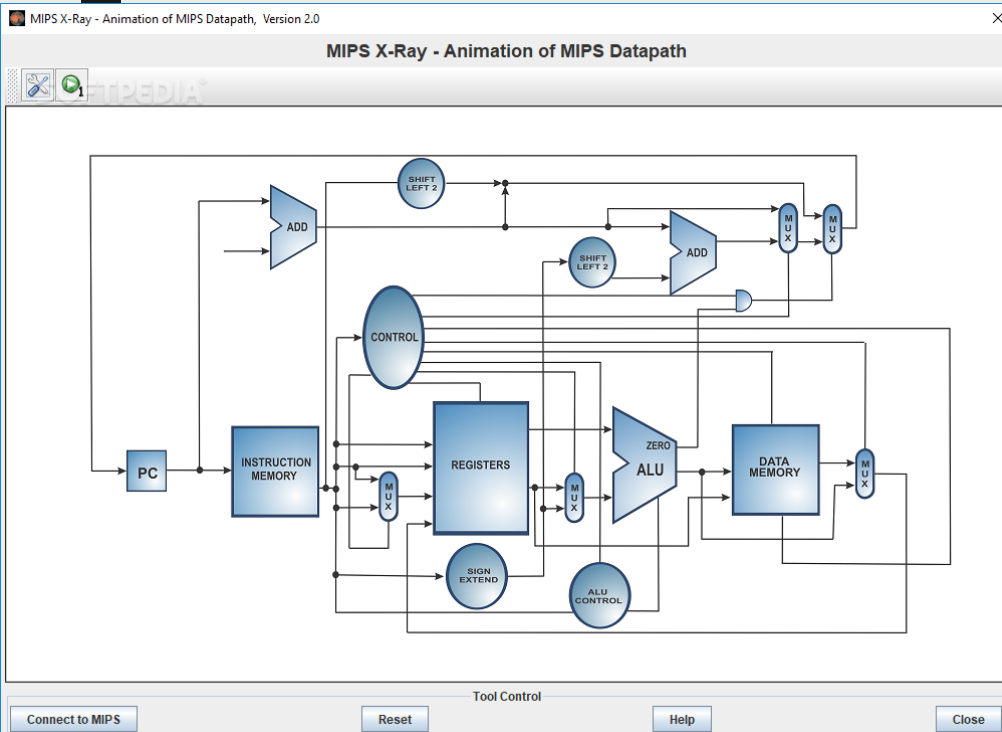




Para saber mais ...

- *STALLINGS, William. Arquitetura e organização de computadores. 8. ed. São Paulo: Pearson, 2010. Capítulo 2*
- *PATTERSON, D.A. & HENNESSY, J. L. Organização e Projeto de Computadores - A Interface Hardware/Software. 3ª ed. Campus, 2005. Capítulo 1*

Próxima aula



Simulador MARS

