

---

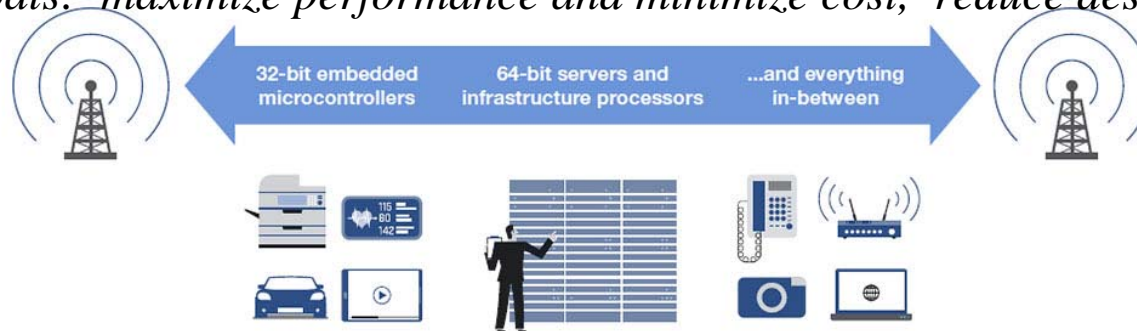
# **MIPS Instructions: Language of the Machine**

# MIPS

---

- Language of the Machine
- More primitive than higher level languages  
e.g., no sophisticated control flow
- Very restrictive  
e.g., MIPS Arithmetic Instructions
- We'll be working with the MIPS instruction set architecture
  - similar to other architectures developed since the 1980's
  - used by NEC, Nintendo, Silicon Graphics, Sony

*Design goals: maximize performance and minimize cost, reduce design time*



# MIPS: Aritmética

---

- All instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code:            **A = B + C**

MIPS code:        **add \$s0, \$s1, \$s2**

(associated with variables by compiler)

# MIPS: Aritmética

---

- Design Principle: simplicity favors regularity. Why?
- Of course this complicates some things...

C code:            **A = B + C + D;**  
                     **E = F - A;**

MIPS code:        **add \$t0, \$s1, \$s2**  
                     **add \$s0, \$t0, \$s3**  
                     **sub \$s4, \$s5, \$s0**

- Operands must be registers, only 32 registers provided
- Design Principle: smaller is faster. Why?

# MIPS: Registradores

---

Name	Register number	Usage
\$zero	0	the constant value 0
\$v0-\$v1	2-3	values for results and expression evaluation
\$a0-\$a3	4-7	arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved
\$t8-\$t9	24-25	more temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

\$1 = \$at: reservado para o *assembler*

\$26-27 = \$k0-\$k1: reservados para o sistema operacional

# MIPS: Organização da Memória

---

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data
...	

# MIPS: Organização da Memória

---

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data
...	

**Registers hold 32 bits of data**

$2^{32}$  bytes with byte addresses from 0 to  $2^{32}-1$

- $2^{30}$  words with byte addresses 0, 4, 8, ...  $2^{32}-4$
- Words are aligned

# MIPS: Instruções

---

- Load and store instructions
- Example:

C code:            `A[8] = h + A[8];`

MIPS code:        `lw    $t0, 32($s3) # $s3 contém o endereço de A`  
                  `add $t0, $s2, $t0`  
                  `sw    $t0, 32($s3)`

- Store word has destination last
- Remember arithmetic operands are registers, not memory!  
(isto é chamado de arquitetura “load-store”)



# MIPS: Instruções

---

- Seja  $g = h + A[i];$   
onde  $A$  é um array de 100 palavras com base apontada por  $\$s3$  e o compilador associa as variáveis  $g$ ,  $h$  e  $i$  com os registradores:
  - $g: \$s1$
  - $h: \$s2$
  - $i: \$s4$

Como seria o código?

# MIPS: Instruções

---

- Seja  $g = h + A[i]$ ;  
onde A é um array de 100 palavras com base apontada por  $\$s3$  e o compilador associa as variáveis g, h e i com os registradores:
  - g:  $\$s1$
  - h:  $\$s2$
  - i:  $\$s4$
- Antes de  $\$t1 \leftarrow A[i]$  é necessário calcular o endereço do elemento ( $A+4*i$ ):

```
add $t1, $s4, $s4    # $t1 <- 2*$s4 (2*i)
add $t1, $t1, $t1    # $t1 <- 2*$t1 (4*i)
add $t1, $s3, $t1
```
- Agora é possível ler o endereço apontado por \$t1 e executar a soma

```
lw  $t0, 0($t1)      #temp $t0 <- A[i]
add $s1, $s2, $t0,    #g <- h + A[i]
```

# MIPS: Instruções de Controle

---

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed

- MIPS conditional branch instructions:

```
bne $t0, $t1, Label  
beq $t0, $t1, Label
```

- MIPS unconditional branch instructions:

```
j label
```

- Meaning:

```
bne $t4,$t5,Label  Next instruction is at Label if $t4 ≠ $t5  
beq $t4,$t5,Label  Next instruction is at Label if $t4 = $t5  
j Label           Next instruction is at Label
```

# MIPS: Instruções de Controle

---

- (assumir `f g h i j`  $\Rightarrow$  `$s0 -> $s4`)

```
        if (i == j) goto L1;
        f = g + h;
L1:     f = f - i;
```

```
        beq $s3, $s4, Label    # goto label if i = j
        add $s0, $s1, $s2      # faz a soma
Label:  sub $s0, $s0, s$3
```

- e se não há label explícito no código C?

```
if (i != j) f = g + h;
f = f - i;
```

assembler cria label

# MIPS: Instruções de Controle

---

- MIPS unconditional branch instructions:  
    **j label**
- Example:
- **(assumir  $h \Rightarrow \$s3$   $i \Rightarrow \$s4$   $j \Rightarrow \$s5$ )**

```
if (i!=j)
    h=i+j;
else
    h=i-j;
```

# MIPS: Instruções de Controle

---

- MIPS unconditional branch instructions:

`j label`

- Example:

- (assumir  $h \Rightarrow \$s3$   $i \Rightarrow \$s4$   $j \Rightarrow \$s5$ )

```
if (i!=j)
    h=i+j;
else
    h=i-j;
```

```
beq $s4, $s5, Lab1
add $s3, $s4, $s5
j Lab2
Lab1: sub $s3, $s4, $s5
Lab2: ...
```

# MIPS: Instruções de Controle

---

```
if (i == j)          f = g + h ; else f = g - h;  
    $s3  $s4        $s0 $s1 $s2
```

# MIPS: Instruções de Controle

---

```
if (i == j)          f = g + h ; else f = g - h;
    $s3  $s4          $s0 $s1 $s2
```

```
                bne $s3,$s4,Else    # goto Else if i ≠ j
                add $s0,$s1,$s2     # f = g + h
                j Exit
Else:           sub $s0,$s1,$s2     # f = g - h
Exit:           ....
```



# MIPS: Instruções

---

- | <u>Instruction</u>              | <u>Meaning</u>                                       |
|---------------------------------|--|
| <code>add \$s1,\$s2,\$s3</code> | <code>\$s1 = \$s2 + \$s3</code>                      |
| <code>sub \$s1,\$s2,\$s3</code> | <code>\$s1 = \$s2 - \$s3</code>                      |
| <code>lw \$s1,100(\$s2)</code>  | <code>\$s1 = Memory[\$s2+100]</code>                 |
| <code>sw \$s1,100(\$s2)</code>  | <code>Memory[\$s2+100] = \$s1</code>                 |
| <code>bne \$s4,\$s5,L</code>    | Next instr. is at Label if <code>\$s4 != \$s5</code> |
| <code>beq \$s4,\$s5,L</code>    | Next instr. is at Label if <code>\$s4 = \$s5</code>  |
| <code>j Label</code>            | Next instr. is at Label                              |

# MIPS: Instruções de Controle

---

- **slt (set-on-less-than):**

```
slt $t0, $s1, $s2
```

```
if $s1 < $s2 then  
    $t0 = 1  
else  
    $t0 = 0
```

# MIPS: Instruções de Controle

---

```
if( i < j )           # (  a      b      c      i      j)
    a = b + c;        # ($s0    $s1    $s2    $s3    $s4)
else
    a = b - c;
```

# MIPS: Instruções de Controle

---

<b>if</b> ( i < j )	# ( a b c i j)
a = b + c;	# (\$s0 \$s1 \$s2 \$s3 \$s4)
<b>else</b>	
a = b - c;	

slt \$t0, \$s3, \$s4	
bne \$t0, \$zero, ELSE	
add \$s0, \$s1, \$s2	#a = b + c; (se \$t0 <> 0)
j Exit	#desvia para exit
ELSE: sub \$s0, \$s3, \$s4	#a = b - c; (se \$t0 = 0)
Exit:	

## MIPS: Exemplo com loop e array

---

```
Loop:      g = g + A[i];      # (g      h      i      j      A)
           i = i + j;        # ($s1    $s2    $s3    $s4    $s5)
           if (i != h) goto Loop;
```

## MIPS: Exemplo com loop e array

---

**Loop:**      **g = g + A[i];**                      **# (g        h        i        j        A)**  
                 **i = i + j;**                      **# (\$s1    \$s2    \$s3    \$s4    \$s5)**  
                 **if (i != h) goto Loop;**

**Loop:**      **add \$t1, \$s3, \$s3**            **# \$t1  $\leftarrow$  i \* 2**  
                 **add \$t1, \$t1, \$t1**            **# \$t1  $\leftarrow$  i \* 4**  
                 **add \$t1, \$t1, \$s5**            **# \$t1  $\leftarrow$  i \* 4 + A (posição do elemento)**  
                 **lw \$t0, 0(\$t1)**            **# \$t0  $\leftarrow$  A[i]**  
                 **add \$s1, \$s1, \$t0**            **# g  $\leftarrow$  g + A[i]**  
                 **add \$s3, \$s3, \$s4**            **# i = i + j**  
                 **bne \$s3, \$s2, Loop**        **# goto Loop if i  $\neq$  h**

## MIPS: Exemplo while loop

---

Loop:        while (save[i] == k)    # (i        j        k        save)  
              i = i + j;                # (\$s3    \$s4    \$s5        \$s6)

## MIPS: Exemplo while loop

---

**Loop:**        **while (save[i] == k)    # (i        j        k        save)**  
                 **i = i + j;                    # (\$s3    \$s4    \$s5        \$s6)**

**Loop:**        **add \$t1, \$s3, \$s3        # \$t1  $\leftarrow$  i \* 2**  
                 **add \$t1, \$t1, \$t1        # \$t1  $\leftarrow$  i \* 4**  
                 **add \$t1, \$t1, \$s6        # \$t1  $\leftarrow$  i \* 4 + save**  
                 **lw \$t0, 0(\$t1)            # \$t0  $\leftarrow$  save[i]**  
                 **bne \$t0, \$s5, Exit        # goto Exit if save[i]  $\neq$  k**  
                 **add \$s3, \$s3, \$s4        # i = i + j**  
                 **j Loop**

**Exit:**        **....**



# MIPS: Constantes

---

- Small constants are used quite frequently (50% of operands)  
e.g.,     **A = A + 5;**  
           **B = B + 1;**  
           **C = C - 18;**
- Solutions? Why not?
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like \$zero) for constants like one.

- MIPS Instructions:

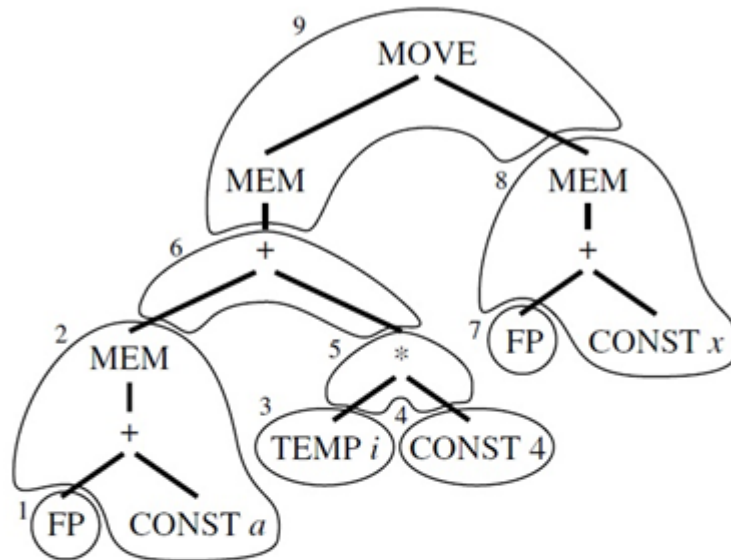
```
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4
```

---

# **MIPS**

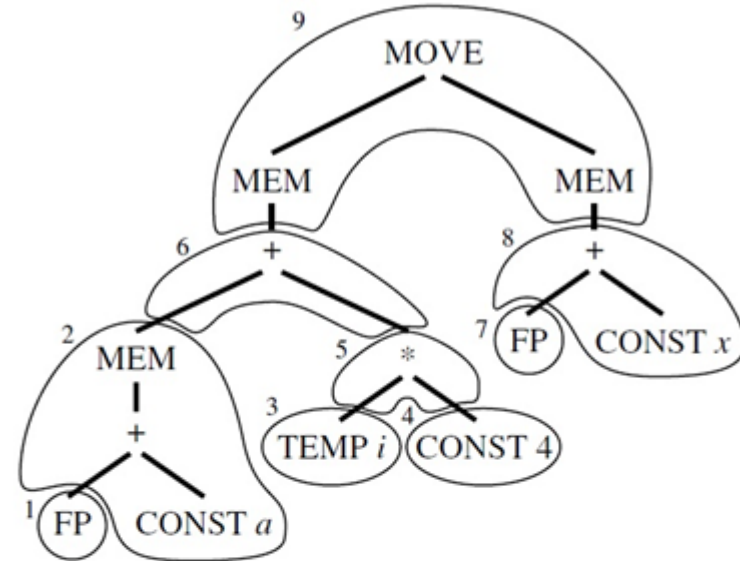
## **Seleção de Instruções** **Expressões**

# Seleção de Instruções



2	LOAD	$r_1 \leftarrow M[\mathbf{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_i \times r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	LOAD	$r_2 \leftarrow M[\mathbf{fp} + x]$
9	STORE	$M[r_1 + 0] \leftarrow r_2$

Programação Dinâmica



2	LOAD	$r_1 \leftarrow M[\mathbf{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_i \times r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	ADDI	$r_2 \leftarrow \mathbf{fp} + x$
9	MOVEM	$M[r_1] \leftarrow M[r_2]$

Maximal Munch

# Padrão Jouette

Name	Effect	Trees
—	$r_i$	TEMP
ADD	$r_i \leftarrow r_j + r_k$	$\begin{array}{c} + \\ \swarrow \quad \searrow \end{array}$
MUL	$r_i \leftarrow r_j \times r_k$	$\begin{array}{c} * \\ \swarrow \quad \searrow \end{array}$
SUB	$r_i \leftarrow r_j - r_k$	$\begin{array}{c} - \\ \swarrow \quad \searrow \end{array}$
DIV	$r_i \leftarrow r_j / r_k$	$\begin{array}{c} / \\ \swarrow \quad \searrow \end{array}$
ADDI	$r_i \leftarrow r_j + c$	$\begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$
SUBI	$r_i \leftarrow r_j - c$	$\begin{array}{c} - \\ \swarrow \quad \searrow \\ \text{CONST} \end{array}$
LOAD	$r_i \leftarrow M[r_j + c]$	$\begin{array}{c} \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \\   \quad   \quad   \quad   \\ + \quad + \quad \text{CONST} \quad   \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$
STORE	$M[r_j + c] \leftarrow r_i$	$\begin{array}{c} \text{MOVE} \quad \text{MOVE} \quad \text{MOVE} \quad \text{MOVE} \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \\   \quad   \quad   \quad   \\ + \quad + \quad \text{CONST} \quad   \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$
MOVEM	$M[r_j] \leftarrow M[r_i]$	$\begin{array}{c} \text{MOVE} \\ \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \\   \quad   \end{array}$

---

# **MIPS**

## **Seleção de Instruções Estruturas de Controle**

# MIPS: Tradução - if-then-else

---

```
if (condição)
{
    //Corpo do then
}
else
{
    //Corpo do else
}
```

Como seria a tradução para assembly?

# MIPS: Tradução - if-then-else

---

```
if (condição)
{
    //Corpo do then
}
else
{
    //Corpo do else
}
```

- tradução da árvore de expressão da **condição** para assembly;
- se o valor da raiz da árvore tiver valor igual a zero, então saltar para o label **LabelElse**;

```
//Corpo do then
j SaidaIf;
```

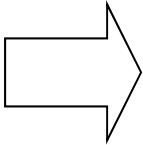
```
LabelElse:
    //Corpo do else
```

```
SaidaIf:
```

# MIPS: Tradução - if-then-else

---

<pre>if (condição) {     //Corpo do then } else {     //Corpo do else }</pre>	<ul style="list-style-type: none"><li>- tradução da árvore de expressão da <b>condição</b> para assembly;</li><li>- se o valor da raiz da árvore tiver valor igual a zero, então saltar para o label <b>LabelElse</b>;</li></ul> <pre>    //Corpo do then     j SaidaIf;  LabelElse:     //Corpo do else  SaidaIf:</pre>
---	--

<pre>if(a&lt;b) {     //Corpo do then }</pre>		<pre>slt r1, a, b beq r1, \$zero, labelSaidaIf //corpo do then labelSaidaIf:</pre>
---	--	--



# MIPS: Tradução - while

---

```
while (condição)
{
    //Corpo do while
}
```

Como seria a tradução para assembly?

# MIPS: Tradução - while

---

```
while (condição)
{
    //Corpo do while
}
```

LabelWhileTest:

- tradução da árvore de expressão da **condição** para assembly;
- se o valor da raiz da árvore tiver valor IGUAL a ZERO, então saltar para o label **LabelSaidaWhile**;

    //Corpo do while

    j LabelWhileTest;

LabelSaidaWhile:

# MIPS: Tradução - for

---

```
for (inicialização; condição-parada; ajustes-valores)
{
    //Corpo do For
}
```

Como seria a tradução para assembly?

# MIPS: Tradução - for

---

```
for (inicialização; condição-parada; ajustes-valores)
{
    //Corpo do For
}
```

- tradução da `inicialização` para assembly;

`LabelForTest:`

- tradução da `condição-parada` para assembly;
- se o valor da raiz da árvore `condição-parada` for igual a ZERO (ou falso), então saltar para o label `LabelForExit`;

`//Corpo do For`

- tradução da árvore `ajustes-valores`;
- `j LabelForTest;`

`LabelForExit:`

# MIPS: Tradução - do-while

---

```
do
{
    //Corpo do do-while
}
while (condição)
```

Como seria a tradução para assembly?

# MIPS: Tradução - do-while

---

```
do
{
    //Corpo do do-while
}
while (condição)
```

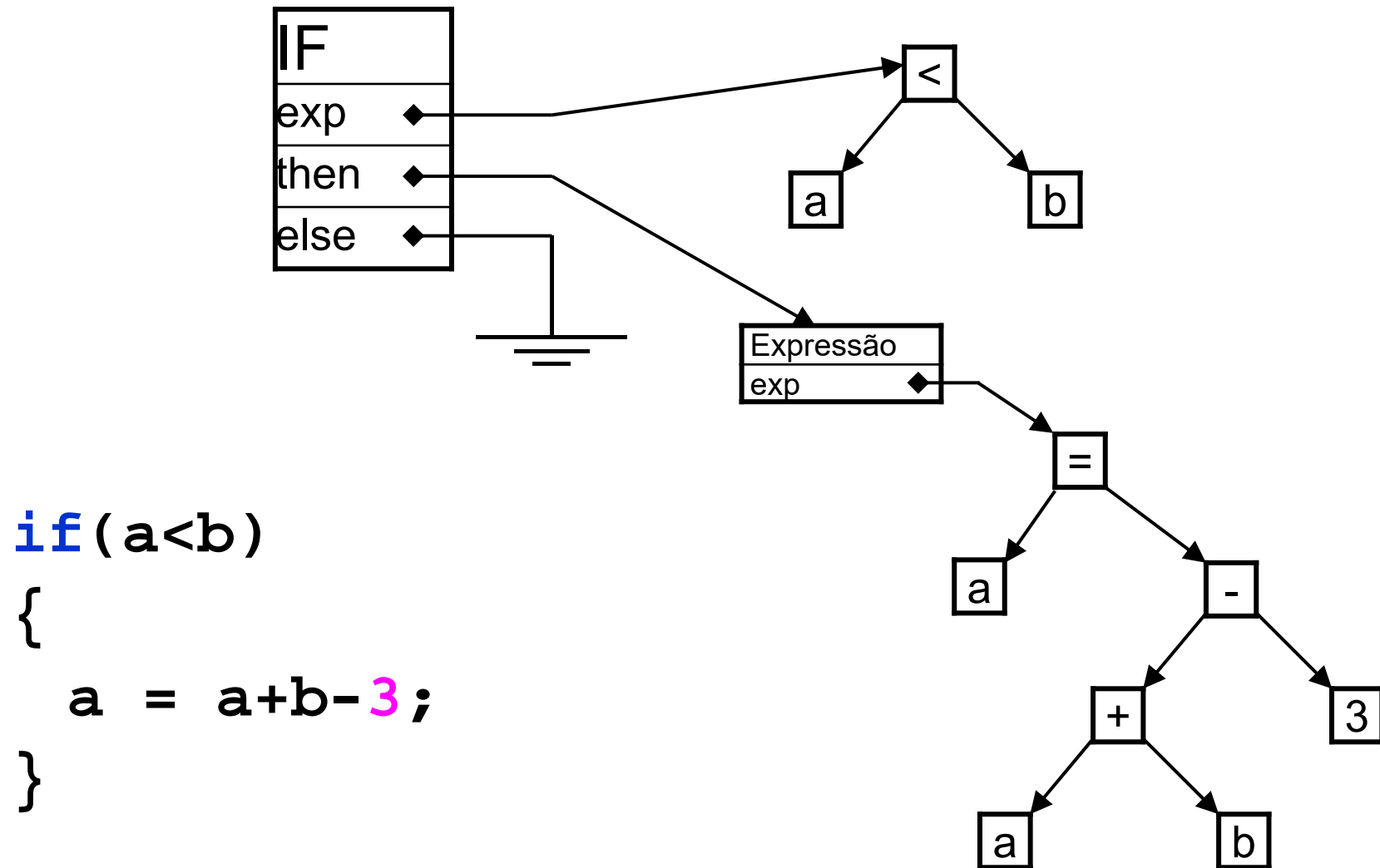
LabelDoWhile:

```
//Corpo do do-while
```

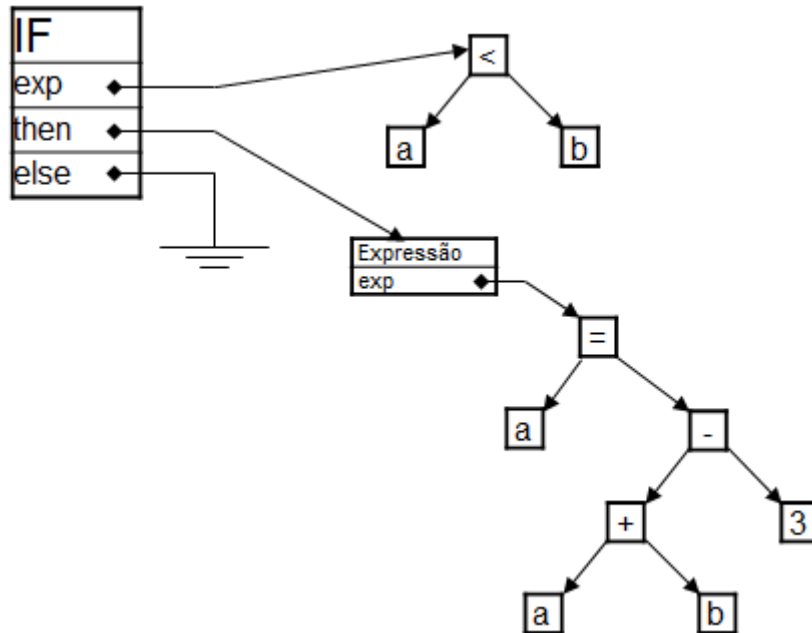
- tradução da árvore de expressão da condição para assembly;
- se o valor da raiz da árvore tiver valor diferente de ZERO, então saltar para o label LabelDoWhile;

LabelSaidaDoWhile:

# MIPS: Tradução para assembly



# MIPS: Tradução para assembly



```
if (condição)
{
    //Corpo do then
}
else
{
    //Corpo do else
}
```

- tradução da árvore de expressão da **condição** para assembly;
- se o valor da raiz da árvore tiver valor igual a zero, então saltar para o label **LabelElse**;

```
//Corpo do then
J SaidaIf;
```

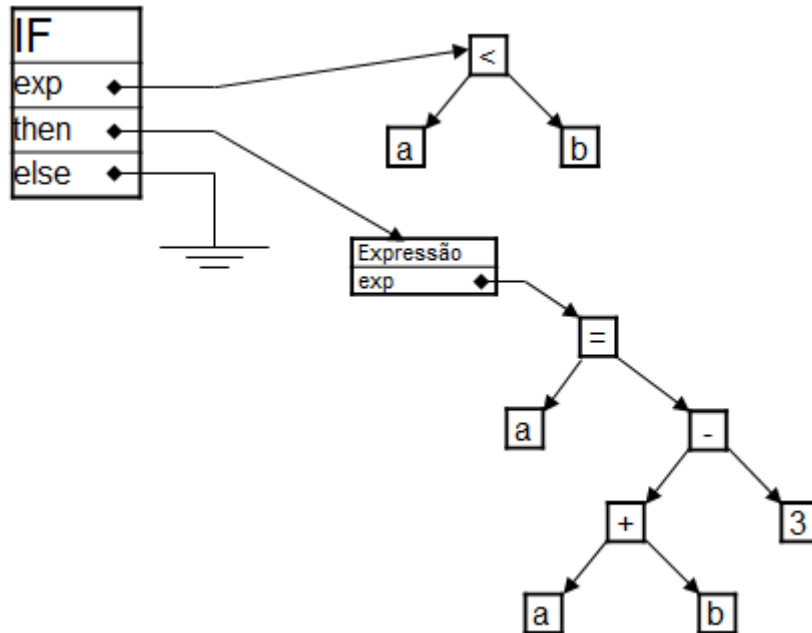
```
LabelElse:
    //Corpo do else
```

```
SaidaIf:
```

Como seria a tradução para assembly?



# MIPS: Tradução para assembly



```
if (condição)
{
    //Corpo do then
}
else
{
    //Corpo do else
}
```

- tradução da árvore de expressão da **condição** para assembly;  
- se o valor da raiz da árvore tiver valor igual a zero, então saltar para o label **LabelElse**;

```
//Corpo do then
J SaidaIf;
```

```
LabelElse:
    //Corpo do else
```

```
SaidaIf:
```

```
slt  r1, a, b
beq  r1, $zero, labelSaidaIf
add  r2, a, b
addi r3, $zero, 3
sub  a, r2, r3
labelSaidaIf:
```

# Conteúdo Programático e Cronograma

---

## 2º Semestre

~~Geração de código~~

~~Análise de fluxo de dados~~

~~Otimização de código~~

~~Alocação de registradores~~

~~Assembly de MIPS~~