

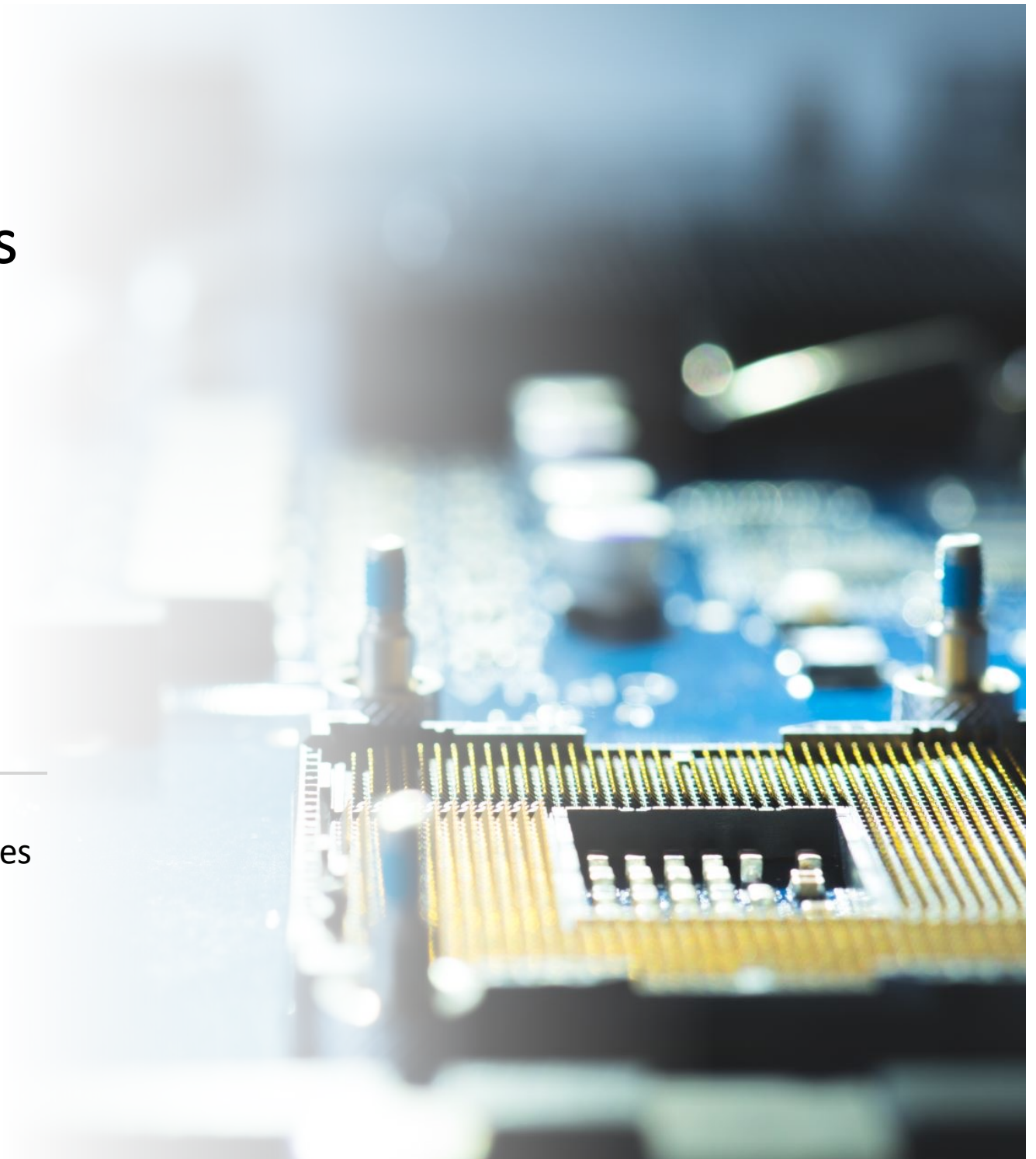


# Arquitectura dos Computadores

## Aula 6 – Operações com bits e Funções Recursivas

---

Professor: Dr. Christophe Soares



## Recordando

Funções são evocadas com *jal*, e retornam com *jr \$ra*

Use a pilha para guardar dados que devem ser preservados. Deve ficar na mesma referência que a encontrou.

Instruções conhecidas:

Aritméticas: *add, addi, sub, addu, addiu, subu*

Memória: *lw, sw*

Decisão: *beq, bne, slt, slti, sltu, sltiu*

Saltos incondicionais (“Jumps”): *j, jal, jr*

Atenção às regras de utilização dos registos nas chamadas a funções!

# Registos do MIPS

A constante 0

Reservado para o assembler

Valores de retorno

Argumentos

Temporários

Guardados (Saved)

Mais temporários

Usados pelo “Kernel”

Apontador Global

Apontador da pilha

“Frame Pointer”

Endereço de retorno

\$0

\$1

\$2-\$3

\$4-\$7

\$8-\$15

\$16-\$23

\$24-\$25

\$26-27

\$28

\$29

\$30

\$31

\$zero

\$at

\$v0-\$v1

\$a0-\$a3

\$t0-\$t7

\$s0-\$s7

\$t8-\$t9

\$k0-\$k1

\$gp

\$sp

\$fp

\$ra

# Outros Registros

**\$at**: usado pelo assembler em qualquer altura; não deve ser usado

**\$k0-\$k1**: podem ser usados pelo SO em qualquer altura; não devem ser usados

**\$gp, \$fp**: não vamos usar

# Operações com bits

Até agora fizemos aritmética (*add, sub, addi*), acesso à memória (*lw e sw*), e saltos (*j, jr, jal*)

Resultado destas instruções vêm num registo como uma quantidade única (inteiro com ou sem sinal)

**Nova abordagem:** olhar para um registo como uma linha de 32 bits em vez de um número

Podemos pretender aceder a um bit em particular (ou grupo de bits) em vez dos 32 em simultâneo

Instruções lógicas *AND* e *OR*

# Operadores Lógicos (1/2)

**Dois operadores básicos:**

**AND:** saída é 1 se **ambas** as entradas são 1

**OR:** saída é 1 se **pelo menos uma** das entradas é 1

**Tabela de verdade:**

A	B	A <b>AND</b> B	A <b>OR</b> B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1



# Operadores Lógicos (2/2)

## Sintaxe das instruções:

**1** **2**, **3**, **4**

**1)** Nome da operação

**2)** Operando que recebe o resultado (“destino”)

**3)** 1º operando (“origem1”)

**4)** 2º operando (“origem2”) ou constante

## Nomes das instruções:

*and, or:*            4) é um registo

*andi, ori:*        4) é uma constante

# Utilização de Operadores Lógicos (1/3)

Isto pode ser usado para criar uma **máscara**

- Exemplo:

\$t0    1011 0110 1010 0100 0011 1101 1001 1010

0x    0000 0000 0000 0000 0000 1111 1111 1111

- O resultado do *and* : **máscara dos últimos 12 bits**

0000 0000 0000 0000 0000 1101 1001 1010



## Utilização de Operadores Lógicos (2/3)

Usamos a **máscara** para isolar (neste caso) os 12 bits mais à direita da primeira linha de bits

Retirando do resultado os restantes bits (colocando-os a 0)

Operador *and* é usado para colocar partes de uma linha de bits a 0 deixando as outras partes inalteradas

Caso a primeira linha de bits se encontre em \$t0 então teríamos a seguinte instrução:

```
andi $t0, $t0, 0xFFF
```

## Utilização de Operadores Lógicos (3/3)

**Podemos usar o or para forçar alguns bits de uma linha a ficarem 1**

**Se \$t0 contém 0x12345678, Depois desta instrução:**

```
ori $t0, $t0, 0xFFFF
```

**\$t0 contém 0x1234FFFF**

**Os 16 bits mais significativos ficam intocados, enquanto os outros são colocados a 1**



# Funções Recursivas

# Exemplo: Números de Fibonacci 1/7

**Os números de fibonacci estão definidos como :**

$$F(n) = F(n - 1) + F(n - 2), \quad F(0) = F(1) = 1$$

**Em C fica:**

```
int fib (int n) {  
    switch (n){  
        case 0: return 1;  
        case 1: return 1;  
        default: return (fib(n-1)+fib(n-2));  
    }  
}
```

# Exemplo: Números de Fibonacci 2/7

**Necessitamos de espaço para 3  
“words” na Pilha - \$ra, \$a0, e \$s0**

```
int fib (int n) {  
    switch (n){  
        case 0: return 1;  
        case 1: return 1;  
        default: return (fib(n-1)+fib(n-2));  
    }  
}
```

## O Prólogo:

fib:

addi \$sp, \$sp, -12	# store 3 variables, 3*4 bytes
sw \$ra, 8(\$sp)	# return to the caller
sw \$s0, 4(\$sp)	# saved register to store the first returned value
sw \$a0, 0(\$sp)	# preserve n for both operations

(...)

# Exemplo: Números de Fibonacci 3/7

## Agora o Epílogo:

(...)

```
lw $s0, 4($sp)
lw $ra, 8($sp)
addi $sp, $sp, 12
jr $ra
```

```
# recover the original value of s
# restore the address to the caller
# recover the stack pointer to its origin
# go to the caller
```

```
int fib (int n) {
    switch (n){
        case 0: return 1;
        case 1: return 1;
        default: return (fib(n-1)+fib(n-2));
    }
}
```

# Exemplo: Números de Fibonacci 4/7

```
int fib (int n) {  
    switch (n){  
        case 0: return 1;  
        case 1: return 1;  
        default: return (fib(n-1)+fib(n-2));  
    }  
}
```



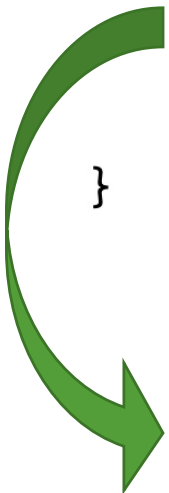
```
int fib(int n) {  
    if(n == 0) { return 1; }  
    if(n == 1) { return 1; }  
    return (fib(n - 1) + fib(n - 2));  
}
```



# Exemplo: Números de Fibonacci 5/7

**Começamos pelos casos específicos (cf. comentários)**

```
int fib(int n) {  
    if(n == 0) { return 1; } /*Traduz-me!*/  
    if(n == 1) { return 1; } /*Traduz-me!*/  
    return (fib(n - 1) + fib(n - 2));  
}
```



# specific cases

beq \$a0, \$zero, fin # if n is 0 go to end

addi \$t0, \$zero, 1

beq \$a0, \$t0, fin # if n is 1 go to end

( ... )

fin:

addi \$v0, \$zero, 1 # set return value to 1

lw \$s0, 4(\$sp) # recover the original value of s

lw \$ra, 8(\$sp) # restore the address to the caller

addi \$sp, \$sp, 12 # recover the stack pointer to its origin

jr \$ra # go to the caller

# Exemplo: Números de Fibonacci 6/7

## Casos genérico / default (cf. comentários)

```
int fib(int n) {  
    if(n == 0) { return 1; }  
    if(n == 1) { return 1; }  
    return (fib(n - 1) + fib(n - 2)); /*Traduz-me!*/  
}
```

( ... )

# default case

addi \$a0, \$a0, -1

jal fib

add \$s0, \$v0, \$zero

lw \$a0, 0(\$sp)

addi \$a0, \$a0, -2

jal fib

add \$v0, \$v0, \$s0

lw \$s0, 4(\$sp)

lw \$ra, 8(\$sp)

addi \$sp, \$sp, 12

jr \$ra

# set argument to n-1

# call fib - recursive function evocation

# move the first result to the s register

# recover the original n

# set argument to n-2

# call fib - recursive function evocation

# move final result to v0

# recover the original value of s

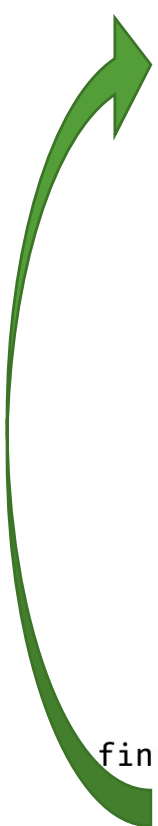
# restore the address to the caller

# recover the stack pointer to its origin

# go to the caller

# Exemplo: Números de Fibonacci 7/7

fib:



```
addi $sp, $sp, -12      # store 3 variables, 3*4 bytes
sw $ra, 8($sp)          # return to the caller
sw $s0, 4($sp)          # saved register to store the first returned value
sw $a0, 0($sp)          # preserve n for both operations
# specific cases
beq $a0, $zero, fin     # if n is 0 go to end
addi $t0, $zero, 1
beq $a0, $t0, fin       # if n is 1 go to end
# default case
addi $a0, $a0, -1       # set argument to n-1
jal fib                 # call fib - recursive function evocation
add $s0, $v0, $zero     # move the first result to the s register
lw $a0, 0($sp)          # recover the original n
addi $a0, $a0, -2       # set argument to n-2
jal fib                 # call fib - recursive function evocation
add $v0, $v0, $s0       # move final result to v0
lw $s0, 4($sp)          # recover the original value of s
lw $ra, 8($sp)          # restore the address to the caller
addi $sp, $sp, 12       # recover the stack pointer to its origin
jr $ra                  # go to the caller
```

fin:

```
addi $v0, $zero, 1      # set return value to 1
lw $s0, 4($sp)          # recover the original value of s
lw $ra, 8($sp)          # restore the address to the caller
addi $sp, $sp, 12       # recover the stack pointer to its origin
jr $ra                  # go to the caller
```

# Exemplo: Números de Fibonacci 7/7

fib:

```
    addi $sp, $sp, -12    # store 3 variables, 3*4 bytes
    sw $ra, 8($sp)        # return to the caller
    sw $s0, 4($sp)        # saved register to store the first returned value
    sw $a0, 0($sp)        # preserve n for both operations
    # specific cases
    addi $v0, $zero, 1    # set return value to 1
    beq $a0, $zero, fin   # if n is 0 go to end
    addi $t0, $zero, 1
    beq $a0, $t0, fin     # if n is 1 go to end
    # default case
    addi $a0, $a0, -1     # set argument to n-1
    jal fib               # call fib - recursive function evocation
    add $s0, $v0, $zero    # move the first result to the s register
    lw $a0, 0($sp)        # recover the original n
    addi $a0, $a0, -2     # set argument to n-2
    jal fib               # call fib - recursive function evocation
    add $v0, $v0, $s0     # move final result to v0
```

fin:

```
    lw $s0, 4($sp)        # recover the original value of s
    lw $ra, 8($sp)        # restore the address to the caller
    addi $sp, $sp, 12     # recover the stack pointer to its origin
    jr $ra                # go to the caller
```



# Exercício (1/2)

**Reescreva a seguinte função em MIPS**

```
int power (int base, int exp){  
    if(exp == 0) return 1;  
    else return base * power(base, exp-1);  
}
```

## Exercício (2/2)

```
power:  beq $a1, $0, end
        addi $sp, $sp, -8
        sw $ra, 4($sp)
        sw $a0, 0($sp)
        addi $a1, $a1, -1
        jal power
        lw $a0, 0($sp)
        mul $v0, $v0, $a0
        lw $ra, 4($sp)
        addi $sp, $sp, 8
        jr $ra
```

```
end:    addi $v0, $0, 1
        jr $ra
```

```
int power (int base, int exp){
    if(exp == 0) return 1;
    else return base * power(base, exp-1);
}
```

# Conclusão...

- Instruções conhecidas
  - Aritméticas:  
add, addi, sub, addu, addiu, subu
  - Memória: lw, sw
  - Decisão: beq, bne, slt, slti, sltu, sltiu
  - Saltos incondicionais (“Jumps”):  
j, jal, jr
- Registos:
  - Todos!
- Funções e Funções Recursivas!