

“One language to rule them all.”

# Chamadas de Função

Paulo Ricardo Lisboa de Almeida

# Funções

- O que é uma função?

# Funções

- **O que é uma função?**
  - Um trecho contendo instruções, que recebe alguns parâmetros, e pode ou não retornar uma resposta ao “chamador”

# Funções

- **O que é uma função?**
  - Segundo Patterson e Henessy
    - Uma função é um espião que sai com um plano secreto
    - Adquire recursos, realiza a tarefa, cobre seus rastros e retorna ao ponto de origem com o resultado solicitado
    - Nada mais é perturbado depois da “missão” terminar
    - Um espião sabe apenas o que ele precisa saber
    - Não faz suposições sobre o seu “patrão”
      - Não sabe nada sobre quem o chamou

# Funções

- Como poderíamos imaginar uma função com as instruções do MIPS?

# Funções

- Como poderíamos imaginar uma função com as instruções do MIPS?
  - Podemos imaginar como um grupo de instruções que realiza uma tarefa
  - Saltamos para esse grupo
  - No final, temos que elaborar alguma forma para inserir no contador de programa o endereço da instrução posterior à instrução que saltou para a função
  - Retornar ao “chamador”

# Exemplo

- Vamos criar uma função que faz o seguinte (exemplo de Patterson, Henessy; 2014):

```
int leaf_example(int g, int h, int i, int j){  
    int f;  
    f = (g+h) - (i+j);  
    return f;  
}
```

# Etapas para chamar a função

1. Colocar os parâmetros em algum lugar que a função possa acessar.
  - Onde?



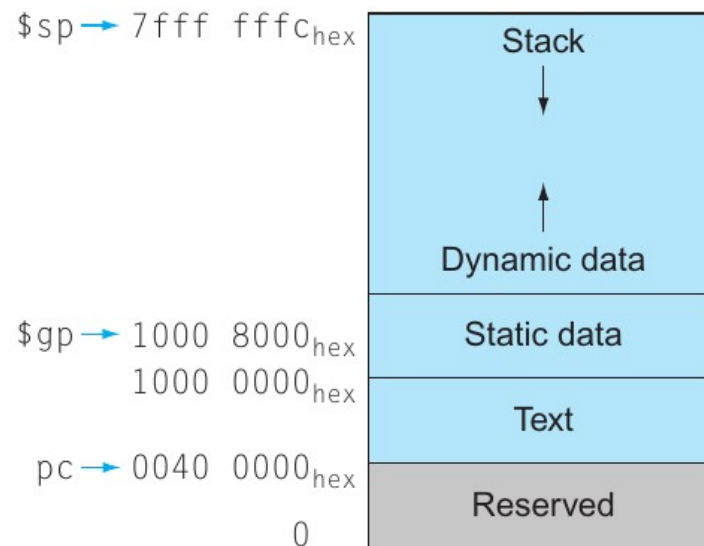
# Etapas para chamar a função

1. Colocar os parâmetros em algum lugar que a função possa acessar.
  - No MIPS, temos os registradores de argumento \$a0, \$a1, \$a2 e \$a3
  - E se precisarmos de mais argumentos?

# Etapas para chamar a função

1. Colocar os parâmetros em algum lugar que a função possa acessar.

- No MIPS, temos os registradores de argumento \$a0, \$a1, \$a2 e \$a3
- E se precisarmos de mais argumentos?
- Salvamos na pilha (memória)



# Etapas para chamar a função

1. Colocar os parâmetros em algum lugar que a função possa acessar.
  - No MIPS, temos os registradores de argumento \$a0, \$a1, \$a2 e \$a3
  - E se precisarmos de mais argumentos?
  - Salvamos na pilha (memória)
- **Arquiteturas diferentes possuem formas diferentes para se passar os parâmetros**
  - Em x86-64
    - No modo 64 bits, os primeiros 6 parâmetros vão em rdi, rsi, rdx, rcx, r8, e r9.
      - Pode diferir ainda se programamos para Windows ou UNIX Like
      - Demais parâmetros na pilha
    - No modo 32 bits, todos parâmetros são passados via pilha
  - PIC 16F6x
    - Passamos via registrador W, pilha ou endereços fixos na memória

# Etapas para chamar a função

1. Colocar os parâmetros em algum lugar que a função possa acessar.

- No MIPS, temos os registradores de argumento \$a0, \$a1, \$a2 e \$a3
- No exemplo, vamos considerar que precisamos chamar nossa função passando os seguintes argumentos: g=1,h=2,i=3,j=4

```
int leaf_example(int g, int h, int i, int j){  
    int f;  
    f = (g+h) - (i+j);  
    return f;  
}
```

```
.text  
.globl main  
main:  
    ori $a0, $zero, 1 #argumento g  
    ori $a1, $zero, 2 #argumento h  
    ori $a2, $zero, 3 #argumento i  
    ori $a3, $zero, 4 #argumento j  
  
end:  
    li $v0, 10  
    syscall  
  
leaf_example:  
    #vamos escrever nossa função aqui
```

# Etapas para chamar a função

## 2. Transferir o controle para a função

- Como?

```
.text
.globl main
main:
    ori $a0, $zero, 1    #argumento g
    ori $a1, $zero, 2    #argumento h
    ori $a2, $zero, 3    #argumento i
    ori $a3, $zero, 4    #argumento j

end:
    li $v0, 10
    syscall

leaf_example:
    #vamos escrever nossa função aqui
```

# Etapas para chamar a função

## 2. Transferir o controle para a função

- Poderíamos fazer um jump simples
  - O problema é que não saberemos o endereço para retornar posteriormente!
- O MIPS inclui uma instrução especial chamada jump-and-link
  - **jal EndereçoFunção**
- Salva o endereço da próxima instrução no registrador \$ra (return adress) e só então salta para o endereço especificado

```
.text
.globl main
main:
    ori $a0, $zero, 1 #argumento g
    ori $a1, $zero, 2 #argumento h
    ori $a2, $zero, 3 #argumento i
    ori $a3, $zero, 4 #argumento j
    jal leaf_example
    #próximas instruções ...
```

```
end:
    li $v0, 10
    syscall
```

```
leaf_example:
    #vamos escrever nossa função aqui
```

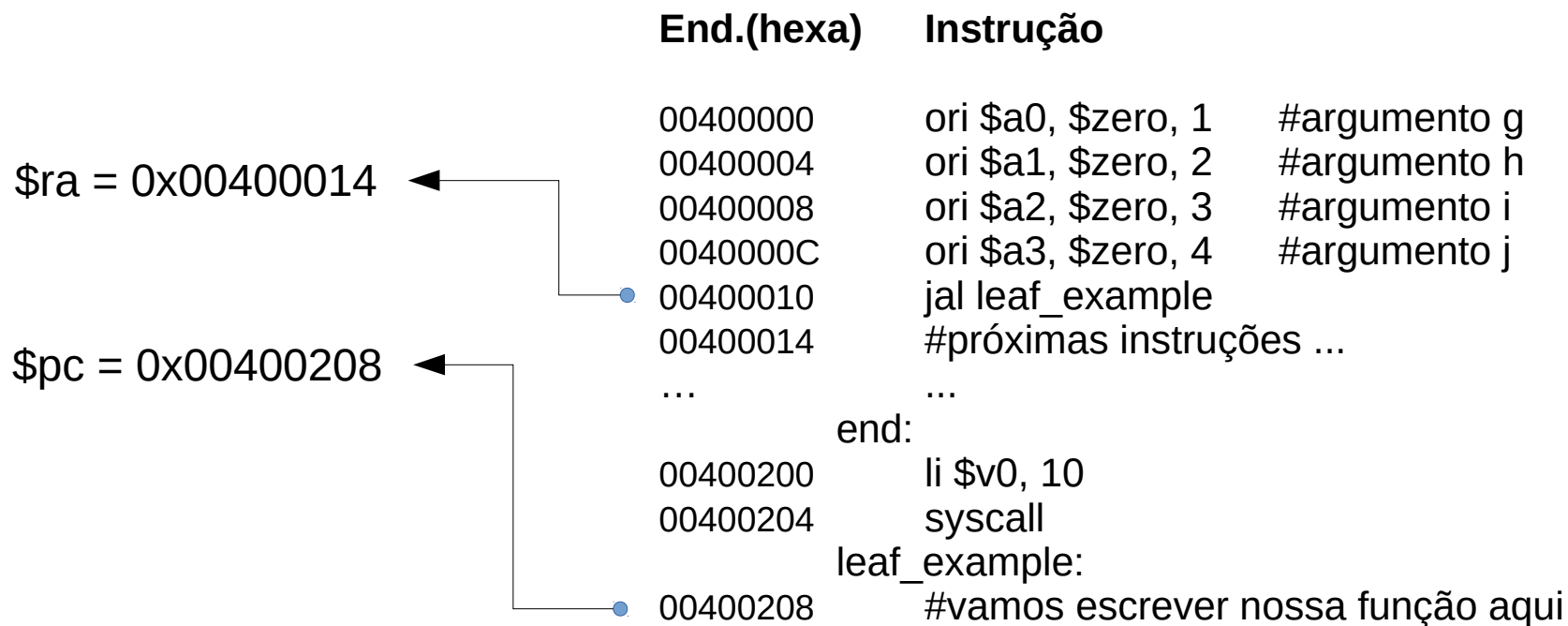
# Exercício

- Qual o endereço em \$ra e em \$pc quando o jal é executado?

End.(hexa)	Instrução
00400000	ori \$a0, \$zero, 1    #argumento g
00400004	ori \$a1, \$zero, 2    #argumento h
00400008	ori \$a2, \$zero, 3    #argumento i
0040000C	ori \$a3, \$zero, 4    #argumento j
<b>00400010</b>	<b>jal leaf_example</b>
00400014	#próximas instruções ...
...	...
	end:
00400200	li \$v0, 10
00400204	syscall
	leaf_example:
00400208	#vamos escrever nossa função aqui

# Exercício

- Qual o endereço em \$ra e em \$pc quando o jal é executado?





# Etapas para chamar a função

## 3. Adquirir os recursos de armazenamento necessários

- Vamos assumir que no nosso exemplo, os registradores \$s0 e \$s1 serão utilizados para realizar a operação da função
- Qual o problema?

```
...  
leaf_example:  
    add $s0,$a0,$a1  
    add $s1,$a2,$a3  
    sub $v0,$s0,$s1
```

```
int leaf_example(int g, int h, int i, int j){  
    int f;  
    f = (g+h) - (i+j);  
    return f;  
}
```

# Etapas para chamar a função

## 3. Adquirir os recursos de armazenamento necessários

- Vamos assumir que no nosso exemplo, os registradores \$s0 e \$s1 serão utilizados para realizar a operação da função
- Qual o problema?
  - Esses registradores podem conter valores que estão sendo utilizados pelo chamador
  - “O espião não pode assumir nada quanto ao seu contratante”
  - “O espião deve limpar seus rastros após a missão”
  - Retornar tudo para a forma que estava anteriormente
  - Como fazer isso?

```
...  
leaf_example:  
    add $s0,$a0,$a1  
    add $s1,$a2,$a3  
    sub $v0,$s0,$s1
```

```
int leaf_example(int g, int h, int i, int j){  
    int f;  
    f = (g+h) - (i+j);  
    return f;  
}
```

# Etapas para chamar a função

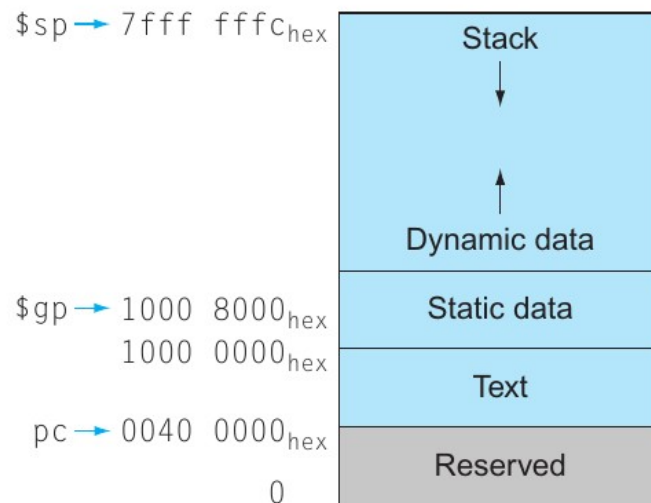
## 3. Adquirir os recursos de armazenamento necessários

- Vamos salvar \$s0 e \$s1 na pilha
- Registrador \$sp contém o endereço atual do topo da pilha
  - SP → *Stack Pointer* (Ponteiro de Pilha)

leaf\_example:

```
addi $sp,$sp,-8
sw $s0, 0($sp)
sw $s1, 4($sp)
add $s0,$a0,$a1
add $s1,$a2,$a3
sub $v0,$s0,$s1
```

```
#deslocando o topo da pilha 8 bytes
#armazenando $s0 nos últimos 4 bytes da pilha
#armazenando $s1 nos 4 bytes após $s0
```



# Etapas para chamar a função

## 4. Realizar a tarefa

- Já fizemos isso na nossa função

leaf\_example:

```
addi $sp,$sp,-8
```

```
sw $s0, 0($sp)
```

```
sw $s1, 4($sp)
```

```
add $s0,$a0,$a1
```

```
add $s1,$a2,$a3
```

```
sub $v0,$s0,$s1
```

#deslocando o topo da pilha 8 bytes

#armazenando \$s0 nos últimos 4 bytes da pilha

#armazenando \$s1 nos 4 bytes após \$s0

# Etapas para chamar a função

## 5. Colocar o valor de retorno em um lugar visível ao chamador

- No MIPS temos os registradores \$v0 e \$v1 para valores de retorno
- Se precisarmos de mais valores de retorno, podemos mais uma vez utilizar a pilha
- Mais uma vez, pode depender da arquitetura e do S.O.

leaf\_example:

<code>addi \$sp,\$sp,-8</code>	<code>#deslocando o topo da pilha 8 bytes</code>
<code>sw \$s0, 0(\$sp)</code>	<code>#armazenando \$s0 nos últimos 4 bytes da pilha</code>
<code>sw \$s1, 4(\$sp)</code>	<code>#armazenando \$s1 nos 4 bytes após \$s0</code>
<code>add \$s0,\$a0,\$a1</code>	
<code>add \$s1,\$a2,\$a3</code>	
<code>sub \$v0,\$s0,\$s1</code>	<code>#o resultado final está sendo armazenado em \$v0</code>

# Etapas para chamar a função

## 6. Liberar os recursos e limpar os rastros

- Exemplos:
  - Restauramos os valores salvos para os registradores
  - Ajustamos a pilha

leaf\_example:

`addi $sp,$sp,-8`

`#deslocando o topo da pilha 8 bytes`

`sw $s0, 0($sp)`

`#armazenando $s0 nos últimos 4 bytes da pilha`

`sw $s1, 4($sp)`

`#armazenando $s1 nos 4 bytes após $s0`

`add $s0,$a0,$a1`

`add $s1,$a2,$a3`

`sub $v0,$s0,$s1`

`lw $s0, 0($sp)`

`#restaurando o valor de $s0`

`lw $s1, 4($sp)`

`#restaurando o valor de $s1`

`addi $sp,$sp,8`

`#ajustando o topo da pilha para “excluir” os itens`

# ■ Etapas para chamar a função

- Retornamos o controle ao chamador
  - Instrução especial jump register
    - **jr REGISTRADOR**
    - Salta para o endereço armazenado no REGISTRADOR
      - Qual registrador?

# Etapas para chamar a função

- Retornamos o controle ao chamador
  - Instrução especial jump register
    - **jr REGISTRADOR**
    - Salta para o endereço armazenado no REGISTRADOR
      - O endereço de retorno foi salvo em \$ra pela instrução jr

leaf\_example:

addi \$sp,\$sp,-8	#deslocando o topo da pilha 8 bytes
sw \$s0, 0(\$sp)	#armazenando \$s0 nos últimos 4 bytes da pilha
sw \$s1, 4(\$sp)	#armazenando \$s1 nos 4 bytes após \$s0
add \$s0,\$a0,\$a1	
add \$s1,\$a2,\$a3	
sub \$v0,\$s0,\$s1	
lw \$s0, 0(\$sp)	#restaurando o valor de \$s0
lw \$s1, 4(\$sp)	#restaurando o valor de \$s1
addi \$sp,\$sp,8	#ajustando o topo da pilha para “excluir” os itens
jr \$ra	<b>#saltando para o endereço armazenado em \$ra</b>



.text  
.globl main

main:

```
ori $s0, $zero 1  
ori $s1, $zero 2
```

Carregando valores para \$0 e \$s1 para  
testar se eles serão restaurados

```
ori $a0, $zero, 1  #argumento g  
ori $a1, $zero, 2  #argumento h  
ori $a2, $zero, 3  #argumento i  
ori $a3, $zero, 4  #argumento j  
jal leaf_example
```

Preparando argumentos e fazendo a  
chamada para a função

```
or $a0,$v0,$zero  
ori $v0,$zero,1  
syscall
```

O resultado é retornado em \$v0.  
Transferindo para \$a0 e chamando o S.O.  
para imprimir o valor na tela

end:

```
li $v0, 10  
syscall
```

leaf\_example:

```
addi $sp,$sp,-8      #deslocando o topo da pilha 8 bytes  
sw $s0, 0($sp)       #$s0 nos últimos 4 bytes da pilha  
sw $s1, 4($sp)       #$s1 nos 4 bytes após $s0  
add $s0,$a0,$a1  
add $s1,$a2,$a3  
sub $v0,$s0,$s1  
lw $s0, 0($sp)       #restaurando o valor de $s0  
lw $s1, 4($sp)       #restaurando o valor de $s1  
addi $sp,$sp,8       #ajustando o topo da pilha para excluir itens  
jr $ra
```

```
int leaf_example(int g, int h, int i, int j){  
    int f;  
    f = (g+h) - (i+j);  
    return f;  
}
```

# Salvar ou não salvar, eis a questão

- No exemplo salvamos os registradores  $\$s$  para recuperá-los posteriormente
  - Isso não é necessário (por convenção) para todos os registradores
  - Veja na tabela o que sua função deve ou não preservar
  - Note que o processador não salva sozinho. Você é quem deve garantir que os conteúdos são salvos

Isso significa que, por exemplo:

- Quem chamar sua função espera que  $\$s0-\$s7$  sejam devolvidos intactos ao término da função
- O conteúdo de  $\$t0-\$t9$  não é garantido de se manter intacto após a chamada

Preservado	Não preservado
$\$s0-\$s7$	$\$t0-\$t9$
$\$sp$	$\$a0-\$a3$
$\$ra$	$\$v0-\$v1$
Pilha acima de $\$sp$	Pilha abaixo de $\$sp$

# Exercícios

1. Execute o programa de exemplo no MARS passo a passo, verificando os conteúdos dos registradores sendo modificados e os conteúdos da memória.
2. Crie uma função que retorna o  $n$ -ésimo número da sequência de Fibonacci. Considere que  $n$  é passado como parâmetro.
  - O seu programa principal deve ser um loop que pede o valor de  $n$  para o usuário repetidas vezes, e chama a função de Fibonacci passando esse valor de  $n$ .
  - O programa deve exibir o número retornado pela função na tela.
  - O programa termina quando o usuário digitar um valor negativo
3. Crie uma função que recebe um valor inteiro  $N$ , e retorne quantos dígitos  $N$  possui
  - Exemplo: 12345 possui 5 dígitos
  - Dica: utilize sucessivas divisões por 10 para obter o valor
4. Considere os polinômios de terceiro grau, que são da seguinte forma:
  - $ax^3 + bx^2 + cx + d$
  - Crie uma função que recebe como parâmetro os coeficientes  $a, b, c$  e  $d$ , e também um ponto  $x$ , e devolve o valor de  $x$  no ponto especificado
  - Considere que todos os valores são inteiros

**Submeta os exercícios 2,3 e 4 no Moodle.**

# Referências

- D. Patterson; J. Henessy. **Organização e Projeto de Computadores: A Interface Hardware / Software**. 4a Edição. Elsevier Brasil, 2014.
- STALLINGS, William. **Arquitetura e organização de computadores**. 8. ed. São Paulo: Pearson Education do Brasil, 2010.
- Bob Plantz. **Introduction to Computer Organization: A Guide to X86-64 Assembly Language and GNU/Linux**. 2019.