# **[AULA 04]** Conjunto de instruções 3

Prof. João F. Mari
*joaof.mari@ufv.br*

# Roteiro

- Introdução
- Operações no hardware do computador
- Operandos do hardware do computador
- Representando instruções no computador
- Operações lógicas
- Instruções para tomada de decisões
- Suporte para procedimentos no hardware do computador

# SUPORTE PARA PROCEDIMENTOS NO HARDWARE DO COMPUTADOR

# Suporte para procedimentos

- Procedimentos ou funções:
  - Construções das linguagens de programação;
  - Servem para estruturar programas, tornando-os mais fáceis de entender, depurar e reutilizar;
  - Sete etapas:
    1. Colocar parâmetros em um lugar onde o procedimento possa acessá-lo;
    2. Transferir o controle para o procedimento;
    3. Adquirir os recursos de armazenamento necessários para o procedimento;
    4. Realizar a tarefa desejada;
    5. Colocar o valor de retorno em um local onde o programa que o chamou possa acessá-lo;
    6. Restaurar o contexto de execução do programa;
    7. Retornar o controle para o ponto de origem, pois um procedimento pode ser chamado de vários pontos de um programa.

# Suporte para procedimentos

- O MIPS utiliza a seguinte convenção para a alocação de seus 32 registradores para chamada de procedimentos:
  - `$a0` - `$a3`: quatro registradores de argumento, para passar parâmetros;
  - `$v0` - `$v1`: dois registradores de valor, para valores de retorno;
  - `$ra`: um registrador de endereço de retorno, para retornar ao ponto de origem do programa que efetuou a chamada;

- O *assembly* do MIPS inclui uma instrução apenas para os procedimentos:
  - Ela desvia para um endereço e simultaneamente salva o endereço da instrução seguinte no registrador `$ra` (instrução *jump-and-link*).

# Suporte para procedimentos

- Instrução *jump-and-link* (`jal`)
  - `jal EnderecoProcedimento`
  - O "link" é armazenado no registrador `$ra`, denominado endereço de retorno;
  - Contador de programa ou PC (*program counter*)
    - Um registrador que mantem o endereço da instrução atual que está sendo executada (conceito de programa armazenado):
  - A instrução `jal` salva PC+4 no registrador `$ra` para o link com a instrução seguinte, a fim de preparar o retorno do procedimento.

- Para apoiar tais situações, computadores como o MIPS utilizam uma instrução *jump register* (`jr`):
  - Executa um desvio incondicional para o endereço especificado no registrador:
  - `jr $ra`
  - O programa que chama o procedimento, coloca os valores de parâmetro em `$a0-$a3` e utiliza `jal X` para desviar para o procedimento X.
  - O procedimento X realiza as suas operações, coloca os resultados em `$v0-$v1` e retorna o controle para o *caller* usando `jr $ra`.

# Suporte para procedimentos

- Usando mais registradores:
  - Suponha que um compilador precise de mais registradores para um procedimento do que os quatro disponíveis: utiliza-se a pilha (*stack*)

- Pilha:
  - Estrutura de dados – uma LISTA em que o último que entra é o primeiro que sai:
    - FILO (First In → Last Out)
  - O ponteiro de pilha (s*tack pointer* – `$sp`) é ajustado em uma palavra para cada registrador salvo ou restaurado
    - Push → insere itens
    - Pop → remove itens
  - As pilhas crescem de endereços maiores para menores.

- O MIPS tem o registrador `$sp`, *stack pointer*, usado para salvar os registradores necessários pelo procedimento chamado.

# Suporte para procedimentos

- Compilando um procedimento em C:

```
1. int exemplo_folha (int g, int h, int i, int j){
2.    int f;
3.    f = ( g + h ) - ( i + j );
4.    return f;
5. }
```

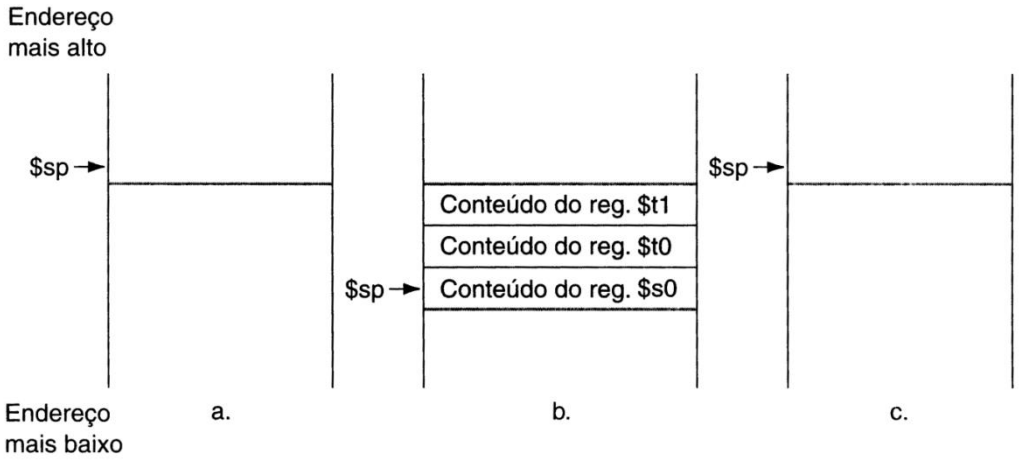
- As variáveis de parâmetro `g`, `h`, `i` e `j` correspondem ao registradores de argumento `$a0-$a3` e `f` corresponde a `$s0`

```
1. exemplo_folha:
2.     addi $sp, $sp,-12 # ajusta a pilha (3 itens)
3.     sw $t1, 8($sp)    # salva registrador
4.     sw $t0, 4($sp)    # salva registrador
5.     sw $s0, 0($sp)    # salva registrador
```

# Suporte para procedimentos

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

```
1.   exemplo_folha:
2.       addi $sp, $sp, -12       # ajusta a pilha (3 itens)
3.       sw $t1, 8($sp)           # salva registrador
4.       sw $t0, 4($sp)           # salva registrador
5.       sw $s0, 0($sp)           # salva registrador
6.       add $t0, $a0, $a1        # $t0 contém g+h
7.       add $t1, $a2, $a3        # $t1 contém i+j
8.       sub $s0, $t0, $t1        # f = (g+h) - (i+j)
9.       add $v0, $s0, $zero      # copia f para reg. de retorno
```



# Suporte para procedimentos

```
1.  exemplo_folha:
2.      addi $sp, $sp, -12   # ajusta a pilha (3 itens)
3.      sw $t1, 8($sp)       # empilha registrador $t1
4.      sw $t0, 4($sp)       # empilha registrador $t0
5.      sw $s0, 0($sp)       # empilha registrador $s0
6.      add $t0, $a0, $a1    # $t0 contém g+h
7.      add $t1, $a2, $a3    # $t1 contém i+j
8.      sub $s0, $t0, $t1    # f = (g+h) - (i+j)
9.      add $v0, $s0, $zero  # copia f para reg. de retorno
10.     lw $s0, 0($sp)       # desempilha registrador $s0
11.     lw $t0, 4($sp)       # desempilha registrador $t0
12.     lw St1, 8($sp)       # desempilha registrador $t1
13.     addi $sp, $sp, 12    # exclui 3 itens da pilha
14.     jr $ra               # Retorna para o endereço em $ra
```

# Suporte para procedimentos

- O software do MIPS separa 18 dos registradores em dois grupos (por convenção):
  - `$t0-$t9`: 10 registradores temporários que não são preservados pelo procedimento chamado;
  - `$s0-$s7`: 8 registradores que precisam ser preservados em uma chamada...
    - Se forem usados, o procedimento chamado os salva e restaura.

- Procedimentos aninhados:
  - A solução é empilhar os valores dos registradores que vão ser utilizados
    - Código que chama (*caller*): empilha (`$a0-$a3`) e (`$t0-$t7`);
    - Procedimento chamado (*callee*): empilha `$ra` e quaisquer registradores salvos usados por ele (`$s0-$s7`).
    - RESUMO: sempre é interessante empilhar registradores usados pelo procedimento

# Suporte para procedimentos

```
1. int fact (int n){
2.     if (n < 1)
3.         return(1);
4.     else
5.         return (n * fact(n-1));
6. }
```

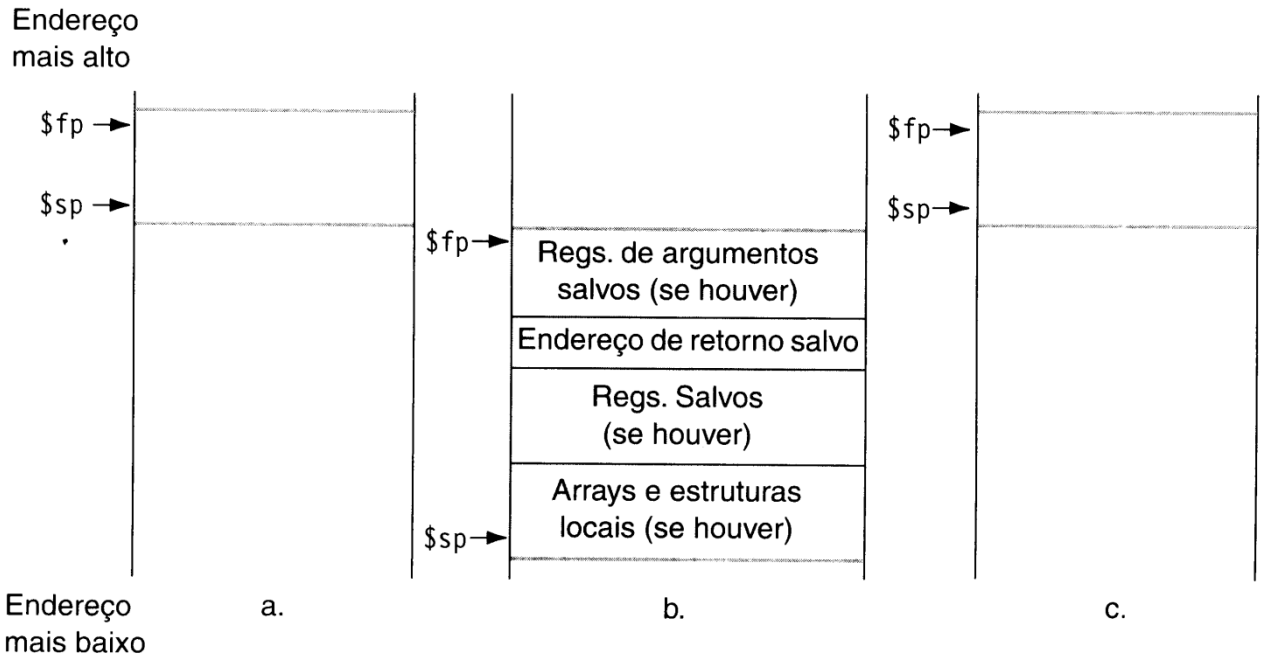
```
1.   fact: addi $sp, $sp, -8      # ajusta pilha para 2 itens
2.       sw $ra, 4($sp)          # salva endereço de retorno
3.       sw $a0, 0($sp)          # salva o argumento n
4.       slti $t0, $a0, 1        # se (n<1) → $t0=1; se n>=1 → $t1=0
5.       beq $t0, $zero, ELSE    # se n >= 1, desvia para ELSE
6.       addi $v0, $zero, 1      # prepara o "retorna 1"
7.       addi $sp, $sp, 8        # retira dois itens da pilha
8.       jr $ra                  # retorna
9.   ELSE: addi $a0,$a0, -1      # argumento recebe n-1
10.      jal fact                # chama fact com n-1
11.      lw $a0, 0($sp)          # retorna de jal: restaura o arg. n
12.      lw $ra, 4($sp)          # restaura o endereço de retorno
13.      addi $sp, $sp, 8        # ajusta pilha para remover 2 itens
14.      mul $v0, $a0, $v0       # calcula n * fact(n-1)
15.      jr $ra                  # retorna para o procedimento que chamou
```

# Suporte para procedimentos

- Interface hardware/software:
  - Uma variável em C é um local na memória,:
    - Sua interpretação depende do seu tipo quanto da classe de armazenamento;
  - A linguagem C possui duas classes de armazenamento:
    - **estáticas** e **automáticas**;
  - As variáveis **automáticas** são locais a um procedimento:
    - São descartadas quando o procedimento termina;
  - As variáveis **estáticas** permanecem durante entradas e saídas de procedimento;
  - As variáveis C declaradas fora de procedimentos são consideradas **estáticas:**
    - Assim como as variáveis declaradas dentro de procedimento com a palavra reservada `static`;
  - Para simplificar o acesso aos **dados estáticos**, o software do MIPS reserva outro registrador, chamado de ponteiro global, e referenciado como `$gp`;
    - `$gp` é um ponteiro global que referencia a memória para facilitar o acesso através de operações simples de `load` e `store`.

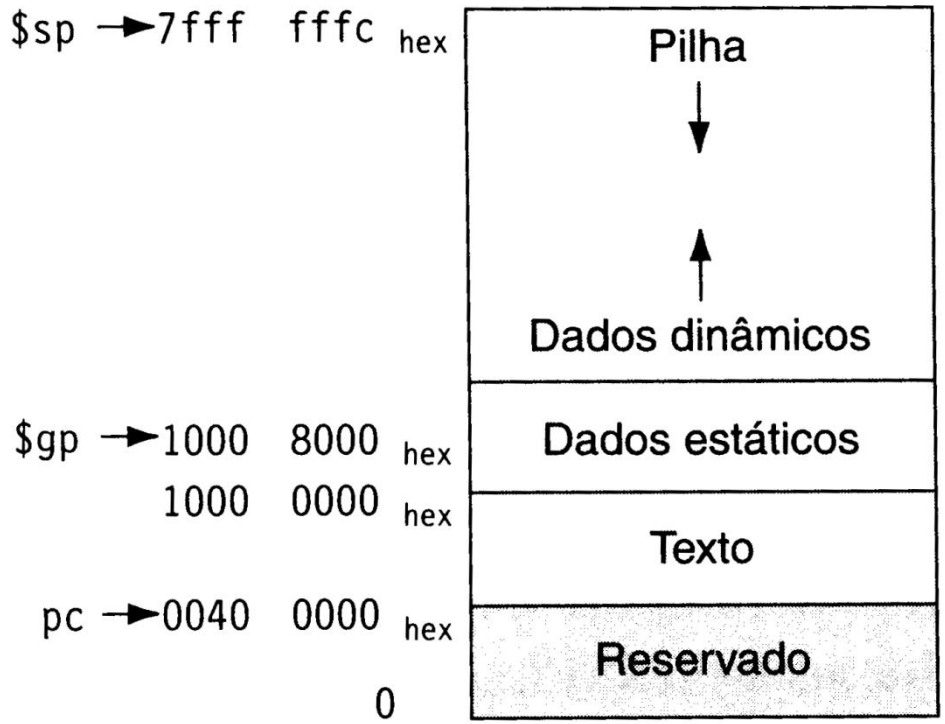
# Suporte para procedimentos

- **Reservando espaço para novos dados na pilha:**
  - A pilha também é utilizada para armazenar variáveis que são locais ao procedimento, que não cabem nos registradores, como arrays ou estruturas locais
  - O segmento da pilha que contém todos os registradores salvos e as variáveis locais de um procedimento é chamado de frame de procedimento ou registro de ativação



# Suporte para procedimentos

- **Reservando espaço para novos dados no *heap:***
  - Além de variáveis que são locais ao procedimentos, programadores precisam de espaço para variáveis estáticas e para estrutura de dados dinâmicas.



# Suporte para procedimentos

- **Reservando espaço para novos dados no *heap:***
  - A pilha e o ***heap*** cresçam um em direção ao outro
    - Permite o uso eficiente da memória enquanto os dois segmentos aumentam e diminuem.
  - A forma pela qual os endereços são usados são convenções do software e não fazem parte da arquitetura MIPS
  - A linguagem C aloca e libera espaço no ***heap*** com funções explícitas:
    - `malloc()`: aloca espaço no *heap*.
    - `free()`: libera espaço no *heap*.

# Suporte para procedimentos

| Categoria | Instrução | Exemplo | Significado |
|---|---|---|---|
| Aritmética | Add | `add $s1, $s2, $s3` | $s1 = $s2 + $s3 |
| | Subtract | `sub $s1, $s2, $s3` | $s1 = $s2 - $s3 |
| Transf. de Dados | Load word | `lw $s1, 100($s3)` | $s1 = Memória[$s2 + 100] |
| | Store word | `sw $s1, 100($s3)` | Memória[$s2 + 100] = $s1 |
| Lógica | And | `and $s1, $s2, $s3` | $s1 = $s2 & $s3 |
| | Or | `or $s1, $s2, $s3` | $s1 = $s2 \| $s3 |
| | Nor | `nor $s1, $s2, $s3` | $s1 = ~($s2 \| $s3) |
| | And immediate | `andi $s1, $s2, 100` | $s1 = $s2 & 100 |
| | Or immediate | `ori $s1, $s2, 100` | $s1 = $s2 \| 100 |
| | Shift left logical | `sll $s1, $s2, 10` | $s1 = $s2 << 10 |
| | Shift righ logical | `srl $s1, $s2, 10` | $s1 = $s2 >> 10 |
| Desvio condicional | Branch on equal | `beq $s1, $s2, L` | If ($s1 == $s2) go to L |
| | Branch on not equal | `bnq $s1, $s2, L` | If ($s1 != $s2) go to L |
| | Set on less than | `slt $s1, $s2, $s3` | If ($s2 < $s3) $s1 = 1<br>else $s1 = 0 |
| | Set on less than immediate | `slti $s1, $s2, 100` | If ($s2 < 100) $s1 = 1<br>else $s1 = 0 |
| Desvio incondicional | Jump | `j L` | Go to L |
| | Jump register | `jr $ra` | Go to $ra |
| | Jump and link | `jal L` | $ra = PC + 4 e go to L |

# COMUNICANDO-SE COM AS PESSOAS

# Suporte para procedimentos

- Representação de caracteres:
  - Código ASCII de 8 bits;
  - Computadores necessitam de instruções que realizem a movimentação de grupos de bits de palavras:

```
1. lb $t0, 0($sp) # Load byte
2. sb $t0, 0($sp) # Store byte
```

  - *Strings* são representados por conjuntos de caracteres, com geralmente três opções para representação:
    - A primeira posição da *string:*
      - Reservada para indicar o tamanho de uma *string*;
    - Uma variável acompanhante possui o tamanho da *string:*
      - Como uma estrutura;
    - A última posição da *string:*
      - Ocupada por um caractere que serve para marcar o final da *string* ( '\0' na linguagem C).

# Suporte para procedimentos

- Compilando um procedimento de cópia de string para demonstrar o uso de strings em C

```
1. void strcpy(char x[], char y[]){
2.    int i;
3.    i = 0;
4.    while ( (x[i] = y[i]) != '\0')
5.        i += 1;
6. }
```

- Considerando que os endereços base para os arrays `x` e `y` são encontrados em `$a0` e `$a1`

```
1. strcpy: addi $sp, $sp, -4
2.         sw $s0, 0($sp)
3.         add $s0, $zero, $zero
4. L1:
```

# Suporte para procedimentos

- Linguagem de alto nível (C):

```
1.  void strcpy(char x[], char y[]){
2.      int i;
3.      i = 0;
4.      while ( (x[i] = y[i]) != '\0')
5.          i += 1;
6.  }
```

- Assembly do MIPS:

```
1.  strcpy: addi $sp, $sp, -4
2.          sw   $s0, 0($sp)
3.          add  $s0, $zero, $zero
4.  L1:     add  $t1, $s0, $a1  # endereço de y[i] em $t1
5.          lb   $t2, 0($t1)    # $t2 = y[i], como é um byte, não i*4
6.          add  $t3, $s0, $a0  # endereço de x[i] em $t3
7.          sb   $t2, 0($t3)    # x[i] = y[i]
8.          beq  $t2, $zero, L2 # se y[i]== 0 , vai para L2
9.          addi $s0, $s0, 1    # i = i + 1
10.         j    L1
11. L2:     lw   $s0, 0 ($sp)
12.         addi $sp, $sp, 4
13.         jr   $ra
```

# Suporte para procedimentos

- Caracteres e strings em Java:
  - **Unicode** é uma codificação universal dos alfabetos da maior parte das linguagens humanas;
  - 16 bits para representar um caractere:

```
1. lh $t0, 0($sp) # Load half-word
2. sh $t0, 0($sp) # Store half-word
```

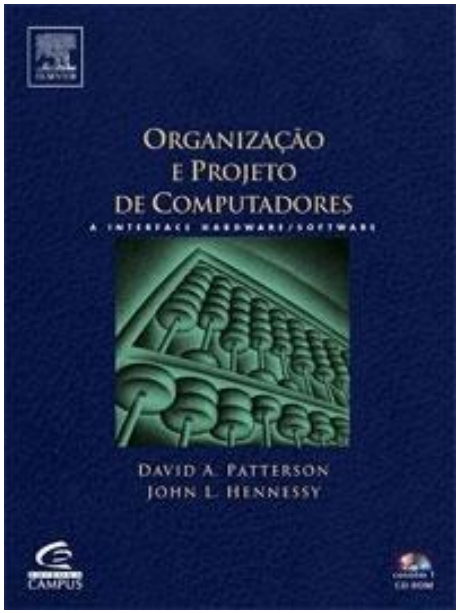
  - Ao contrário da linguagem C, Java reserva uma palavra para indicar o tamanho da string.

## Suporte para procedimentos

| Categoria | Instrução | Exemplo | Significado |
|---|---|---|---|
| Aritmética | Add | `add $s1, $s2, $s3` | $s1 = $s2 + $s3 |
| | Subtract | `sub $s1, $s2, $s3` | $s1 = $s2 - $s3 |
| Transf. de Dados | Load word | `lw $s1, 100($s2)` | $s1 = Memória[$s2 + 100] |
| | Store word | `sw $s1, 100($s2)` | Memória[$s2 + 100] = $s1 |
| | Load half | `lh $s1, 100($s2)` | $s1 = Memória[$s2 + 100] |
| | Store half | `sh $s1, 100($s2)` | Memória[$s2 + 100] = $s1 |
| | Load byte | `lb $s1, 100($s2)` | $s1 = Memória[$s2 + 100] |
| | Store byte | `sb $s1, 100($s2)` | Memória[$s2 + 100] = $s1 |
| Lógica | And | `and $s1, $s2, $s3` | $s1 = $s2 & $s3 |
| | Or | `or $s1, $s2, $s3` | $s1 = $s2 \| $s3 |
| | Nor | `nor $s1, $s2, $s3` | $s1 = ~($s2 \| $s3) |
| | And immediate | `andi $s1, $s2, 100` | $s1 = $s2 & 100 |
| | Or immediate | `ori $s1, $s2, 100` | $s1 = $s2 \| 100 |
| | Shift left logical | `sll $s1, $s2, 10` | $s1 = $s2 << 10 |
| | Shift righ logical | `srl $s1, $s2, 10` | $s1 = $s2 >> 10 |
| Desvio condicional | Branch on equal | `beq $s1, $s2, L` | If ($s1 == $s2) go to L |
| | Branch on not equal | `bnq $s1, $s2, L` | If ($s1 != $s2) go to L |
| | Set on less than | `slt $s1, $s2, $s3` | If ($s2 < $s3) $s1 = 1; else $s1 = 0 |
| | Set on less than immediate | `slti $s1, $s2, 100` | If ($s2 < 100) $s1 = 1; else $s1 = 0 |
| Desvio incondicional | Jump | `j L` | Go to L |
| | Jump register | `jr $ra` | Go to $ra |
| | Jump and link | `jal L` | $ra = PC + 4 e go to L |

# BIBLIOGRAFIA


- PATTERSON, D.A; HENNESSY, J.L. **Organização e Projeto de Computadores: A Interface Hardware/Software.** 3a. Ed. Elsevier, 2005.
  - Capítulo 2.




- Notas de aula do prof. Luciano J. Senger:
  - http://www.ljsenger.net/classroom.html

# [FIM]

- FIM:
  - **[AULA 04]** Conjunto de instruções 3
- Próxima aula:
  - **[AULA 05]** Avaliando o desempenho 1