

## Aula 8

• Subrotinas: evocação e retorno

• Caracterização das subrotinas na perspectiva do chamador e do chamado

• Convenções adoptadas quanto a:

• passagem de parâmetros para subrotinas

• devolução de valores de subrotinas

• salvaguarda de registos `save` e `save` vs `push` e `pop`

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira e Silva

## Funções

• Três razões principais que justificam a existência de funções:

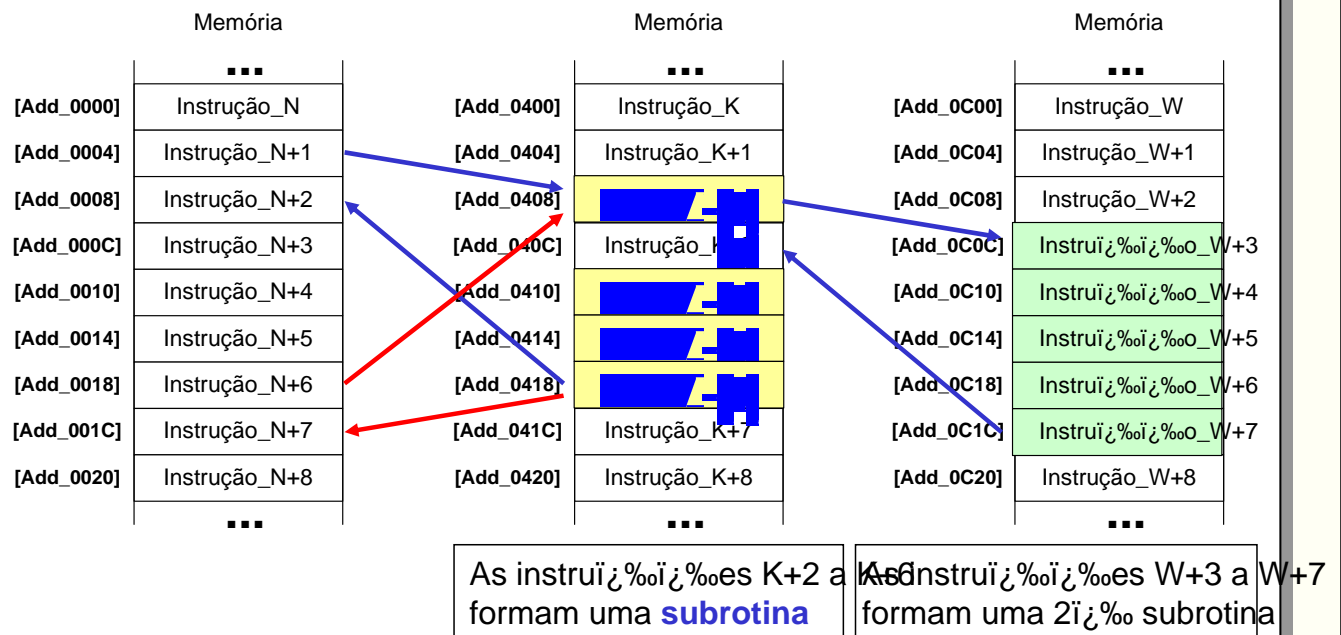
• **Reutilização no contexto de um determinado programa**  
aumentando a eficiência na dimensão do código, evitando a repetição de um mesmo trecho de código por um único evocável de múltiplos pontos do programa.

• **Reutilização no contexto de um conjunto de programas**  
permitindo que o mesmo código possa ser reaproveitado (bibliotecas de funções).

• **Organização e estruturação do código**

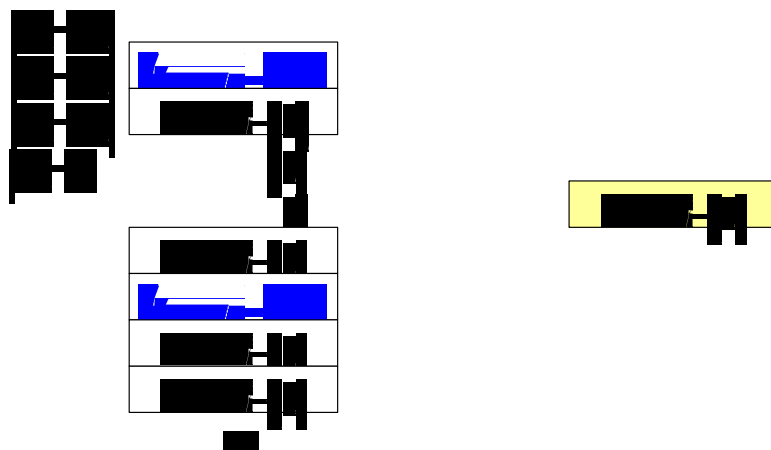
(\*) No contexto da linguagem *Assembly*, as funções e os procedimentos são genericamente conhecidas por **subrotinas**!

## Subrotinas: exemplo prático

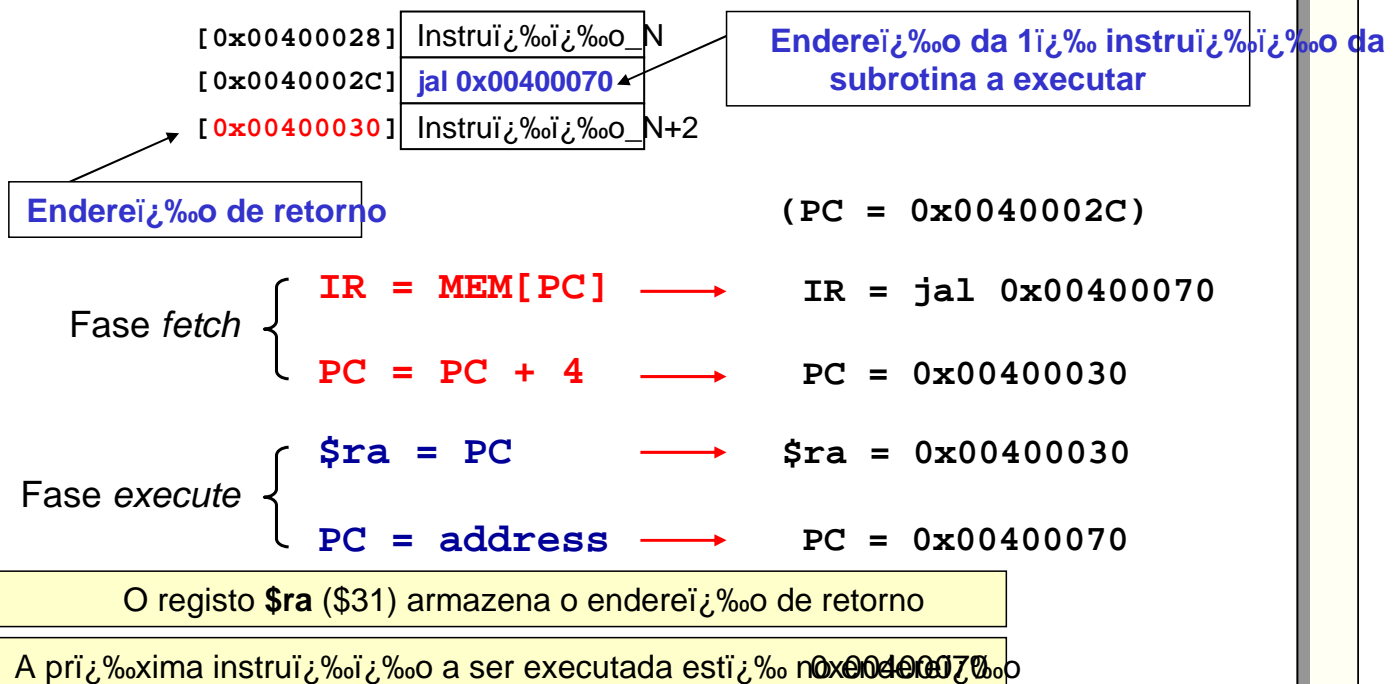


Como implementar o esquema de **chamada** e **retorno** de uma subrotina?

## Subrotinas: exemplo prático (MIPS)

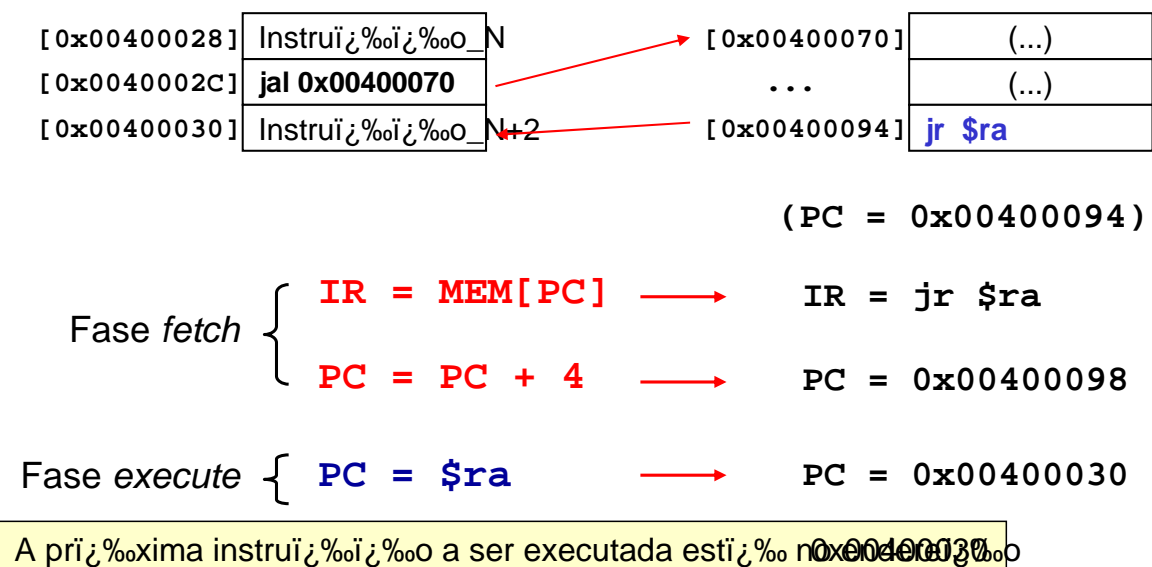


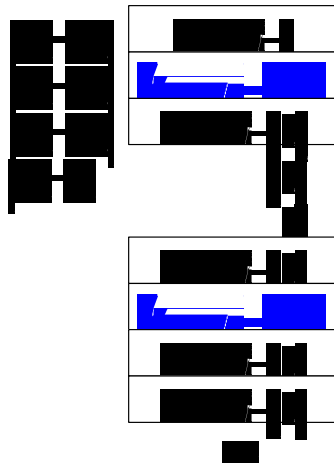
## Ciclo de execução da instrução "jump and link" **jal**



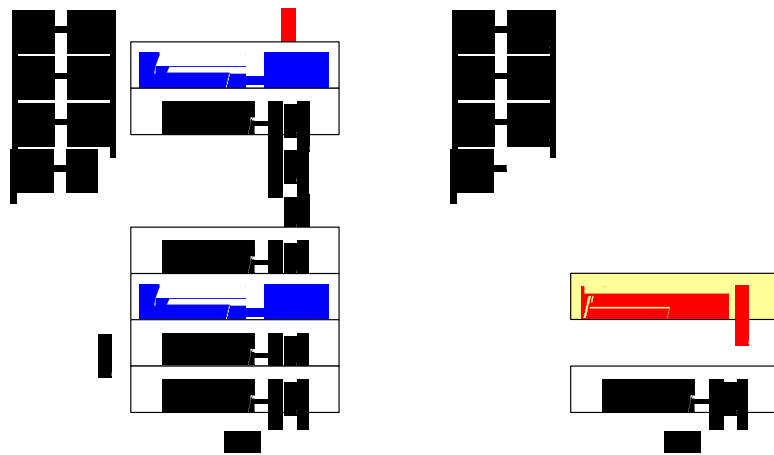
## E como regressar à instrução que sucede à instrução "jal"?

Resposta: aproveitando o endereço de retorno armazenado em **\$ra** durante a execução da instrução **jal**





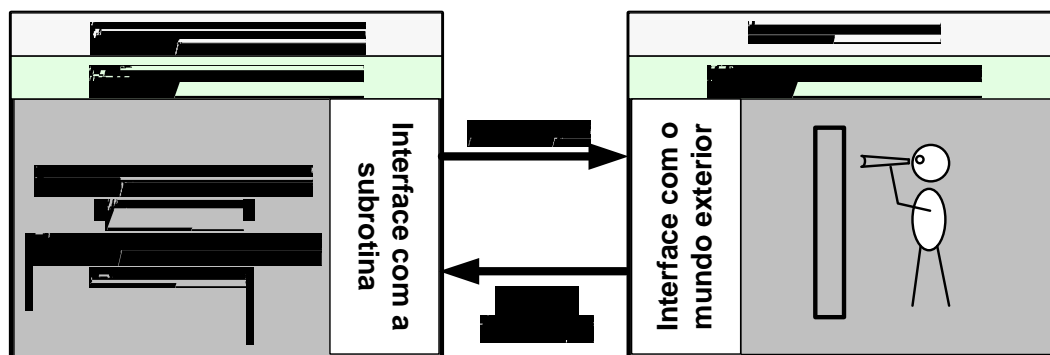
E se a subrotina (Instruções  $K+1$  a  $J$ ) não é uma subrotina?



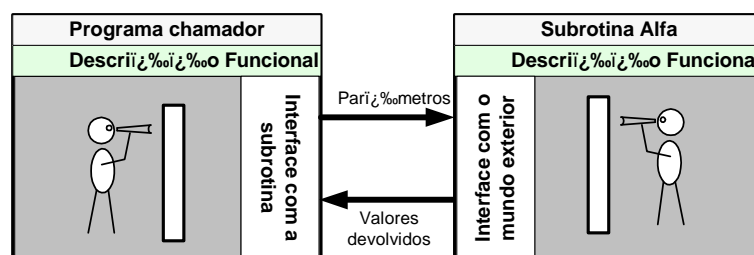
## Caracterização de uma subrotina na perspectiva do programador

Na perspectiva do programador, a subrotina que tem a responsabilidade de escrever um **trecho de código isolado**, com uma funcionalidade bem definida, e com um interface que ele próprio pode determinar em função das necessidades

Em contrapartida, o facto de a subrotina ter descrita para ser reutilizada implica necessariamente que o programador não pode antecipadamente as características do programa que irá evocar o seu código



Torna-se assim aparente a necessidade de definir um conjunto de **regras que regulem a relação entre o programador e a subrotina**, independentemente de qual seja o que respeita à definição do interface entre ambos que respeita aos princípios que assegurem uma **convivência pacífica**



```

...
li      $t0,0
11:     bge    $t0,5,endif1
...
jal     sub1
...
addi    $t0,$t0,1
j       11
endif1: ...

```

```

sub1:   li      $t0,3
12:     ble    $t0,0,endif2
...
addi    $t0,$t0,-1
j       12
endif2: jr      $ra

```

Quantas vezes vai ser executado o ciclo do programa chamador?

## Regras a definir entre o programa chamador e a subrotina chamada

### Ao nível do interface

Como **passar parâmetros** do chamador para o chamado, quantos e onde;

Como **receber**, do lado do chamador, **valores devolvidos** pelo chamado;

### Ao nível das regras de convivência

Que registos do CPU podem o chamador e o chamado usar, sem que isso represente um conflito de interesses (pelo facto alterar o conteúdo de um registo que está simultaneamente usado pelo outro)

Como partilhar a memória usada para armazenar, sem risco de sobreposição (e consequente perda de informação)

## Convenções adoptadas quanto à passagem de parâmetros

Caso a subrotina defina parâmetros no seu interface, a passagem desses parâmetros entre o chamador e o chamado deve, no caso do MIPS, respeitar as seguintes regras:

Os parâmetros que possam ser armazenados na diferença de um registo (32 bits) devem ser passados à subrotina nos registos **\$a0 a \$a3 (\$4 a \$7)** por esta ordem (**o primeiro parâmetro sempre em \$a0, o segundo em \$a1 e assim sucessivamente**).

Caso o número de parâmetros a passar nos registos seja superior a quatro, os restantes (pela ordem em que são declarados) deverão ser passados na stack.

No caso de um ou mais parâmetros serem do tipo **float** ou **double**, os registos utilizados para os passar serão os registos **\$f12 e \$f14** do coprocessador de **variável flutuante**

## Convenções adoptadas quanto à devolução de valores

Caso a subrotina pretenda devolver um valor ao programa chamador, essa devolução deve, no caso do MIPS, respeitar a seguinte:

A subrotina pode devolver um valor de 32 bits ou de 64 bits:

Se o valor a devolver é **32 bits**, utilizado o registo **\$v0**

Se o valor a devolver é **64 bits**, são utilizados os registos **\$v1** (32 bits mais significativos) e **\$v0** (32 bits menos significativos)

No caso de o valor a devolver ser do tipo **float** ou **double**, o registo a utilizar será o registo **\$f0** do coprocessador de **vírgula flutuante**.

### Exemplo:

```
int max(int, int);

void main(void)
{
    static int maxVal;
    maxVal = max(19, 35);
}
```

Em Assembly:

```
.data
maxVal: .space 4
.text
main: (...) # Salvag. $ra
    li    $a0, 19
    li    $a1, 35
    jal   max
    la    $t0, maxVal
    sw    $v0, 0($t0)
    (...) # Repõe $ra
    jr    $ra
```

Note-se que, para escrever o programa chamador, não é necessário conhecer os detalhes de implementação do chamado.

parâmetros

evocação da subrotina

valor devolvido



**Exemplo (continuação):**

```
int max(int a, int b)
{
    int vmax = a;

    if(b > vmax)
        vmax = b;
    return vmax;
}
```

Note-se que, para escrever o programa *chamado*, não é necessário conhecer os detalhes de implementação do *chamador*.

Em Assembly:

```
max:  move    $v0, $a0
      ble     $a1, $v0, endif
      move    $v0, $a1
endif: jr     $ra
```

regresso ao chamador

Valor a devolver

parâmetros

Será necessário salvar o valor de \$ra?

**Convenientes adoptadas quanto à salvaguarda de registo**

Que registos pode usar uma subrotina, sem que se corra o risco de que os mesmos registos estejam a ser usados pelo programa *chamador*, potenciando assim a destruição de informação vital para a execução do programa como um todo?

Uma hipótese seria dividir de forma estática os registos existentes entre *chamador* e *chamado*? Mas o que fazer então caso o *chamado* fosse simultaneamente "chamador" (subrotina que chama outra subrotina)?

Outra hipótese, mais prática, consiste em atribuir a todos os parceiros a responsabilidade de copiar previamente para memória externa o conteúdo de qualquer registo que pretendam utilizar (**salvar o registo**) e posteriormente, repor o valor original lá armazenado.

## Convenções adoptadas quanto à salvaguarda de registo

Existem duas estratégias que são geralmente adoptadas (alternativa) pela maior parte das arquitecturas:

1. Deixa-se ao cuidado do programador a responsabilidade de salvaguardar o conteúdo da totalidade dos registos antes de evocar a subrotina, cabendo-lhe também a tarefa de repor posteriormente o seu valor (embora, no limite, seja admissível que o programador salve apenas o conteúdo dos registos de que efectivamente tenha a precisar mais tarde). Estamos nesse caso perante uma estratégia **caller-saved**.

2. Entrega-se à subrotina a responsabilidade pela salvaguarda dos registos de que possa necessitar, assegurando a mesma subrotina a tarefa de repor o seu valor imediatamente antes de regressar ao programa chamador. Estamos nesse caso perante uma estratégia **callee-saved**.

## Convenções adoptadas quanto à salvaguarda de registos

No caso do MIPS, a estratégia adoptada é uma variação das anteriores, e baseia-se nas duas regras seguintes:

1. Os registos **\$t0..\$t9**, **\$v0..\$v1** e **\$a0..\$a3** podem ser livremente utilizados e alterados pelas subrotinas

2. Os valores dos registos **\$s0..\$s7** não podem, na perspectiva do chamador, ser alterados pelas subrotinas

## Convenções adoptadas quanto à salvaguarda de registos

Os registos **\$t0..\$t9**, **\$v0..\$v1** e **\$a0..\$a3** podem ser livremente utilizados e alterados pelas subrotinas

Esta regra implica, na prática, que um programador que esteja a usar um ou mais destes registos, deverá **salvaguardar o seu conteúdo antes de evocar uma subrotina**, sob pena de que esta os venha a alterar.

Os valores dos registos **\$s0..\$s7** não podem, na perspectiva do chamador, ser alterados pelas subrotinas

A segunda regra implica que, se uma dada subrotina precisar de usar um registo do tipo **\$sn**, compete a essa subrotina copiar **previamente** o seu conteúdo para um lugar seguro (memória externa), onde-o imediatamente antes de terminar. Dessa forma, do ponto de vista do programa chamador (que não vê o código da subrotina) o registo não tivesse sido usado ou alterado.

## Considerações sobre a utilização da convenção

**Subrotinas terminais** (que não chamam qualquer subrotina)

Se devem utilizar (preferencialmente) registos que não necessitam de ser salvaguardados (**\$t0..\$t9**, **\$v0..\$v1** e **\$a0..\$a3**)

**Subrotinas que chamam outras subrotinas**

Devem utilizar os registos **\$s0..\$s7** para o armazenamento de valores que se pretenda preservar. A utilização dos registos implica a sua prévia salvaguarda na memória externa no início da subrotina e a respectiva reposição no fim

Devem utilizar os registos **\$t0..\$t9**, **\$v0..\$v1** e **\$a0..\$a3** para os restantes valores

O problema detectado na codificação do programa da subrotina do slide 13 pode facilmente ser resolvido se a convenção de salvaguarda de registos for aplicada

A variável índice do ciclo do programa chamador reside num registo (por exemplo no \$s0) e, **garantidamente**, a subrotina não vai alterar

O código da subrotina desconhecido do programador do programa chamador e vice-versa

```
(...) # Salv. $s0
...
li      $s0,0
11:     bge    $s0,5,endif1
...
jal     sub1
...
addi    $s0,$s0,1
j       11
endif1: ...
(...) # Repoe $s0
```

```
sub1:   li      $t0,3
12:     ble    $t0,0,endif2
...
addi    $t0,$t0,-1
j       12
endif2: jr      $ra
```

Quantas vezes vai ser executado o ciclo do programa chamador?