

## Aula 7

- Sub-rotinas: evocação e retorno
- Caracterização das sub-rotinas na perspetiva do "chamador" e do "chamado"
- Convenções adotadas quanto à:
  - passagem de parâmetros para sub-rotinas
  - devolução de valores de sub-rotinas
  - salvaguarda de registos: "caller-saved" *versus* "callee-saved"

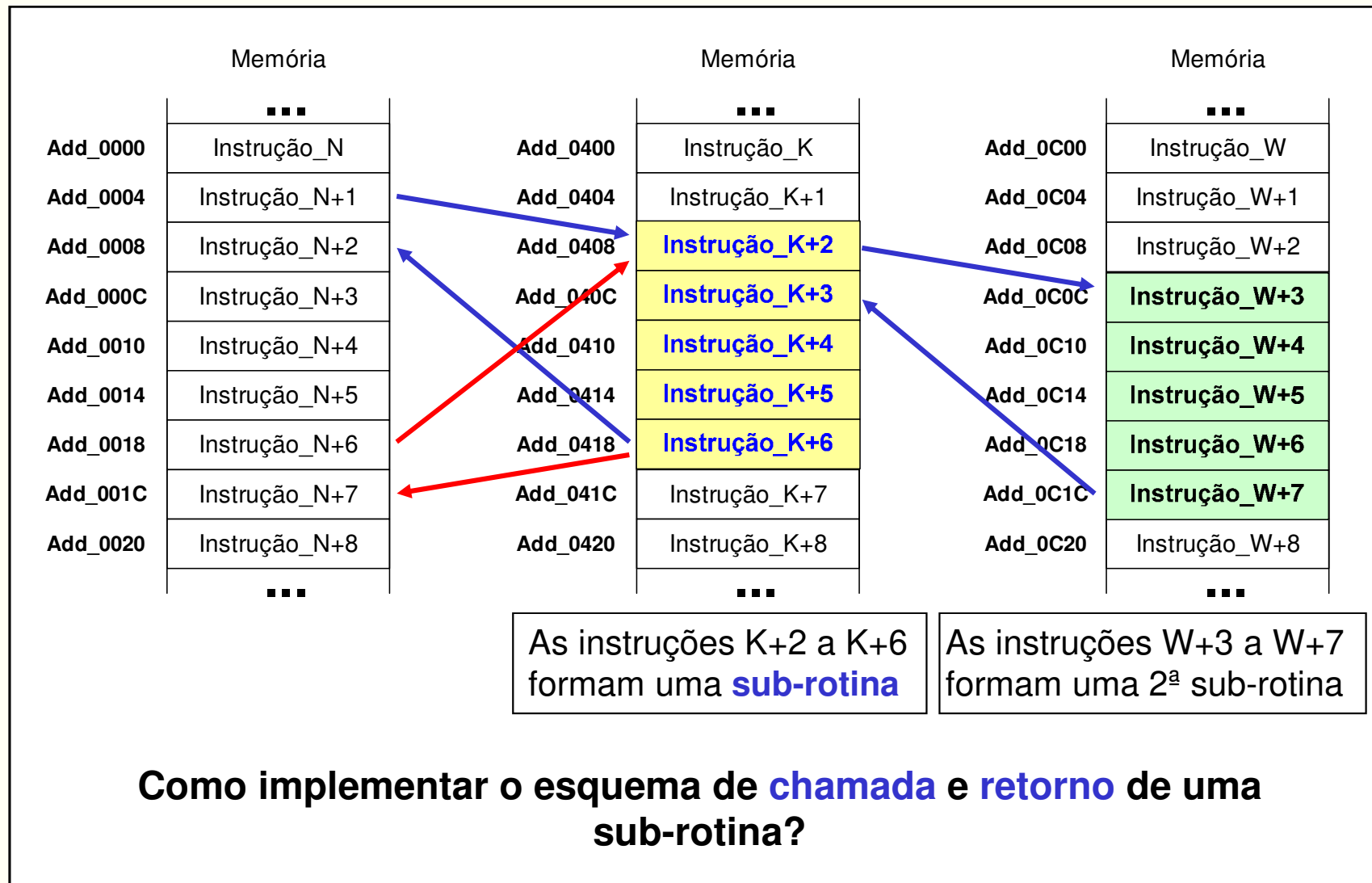
José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador

# Porque se usam funções (sub-rotinas)?

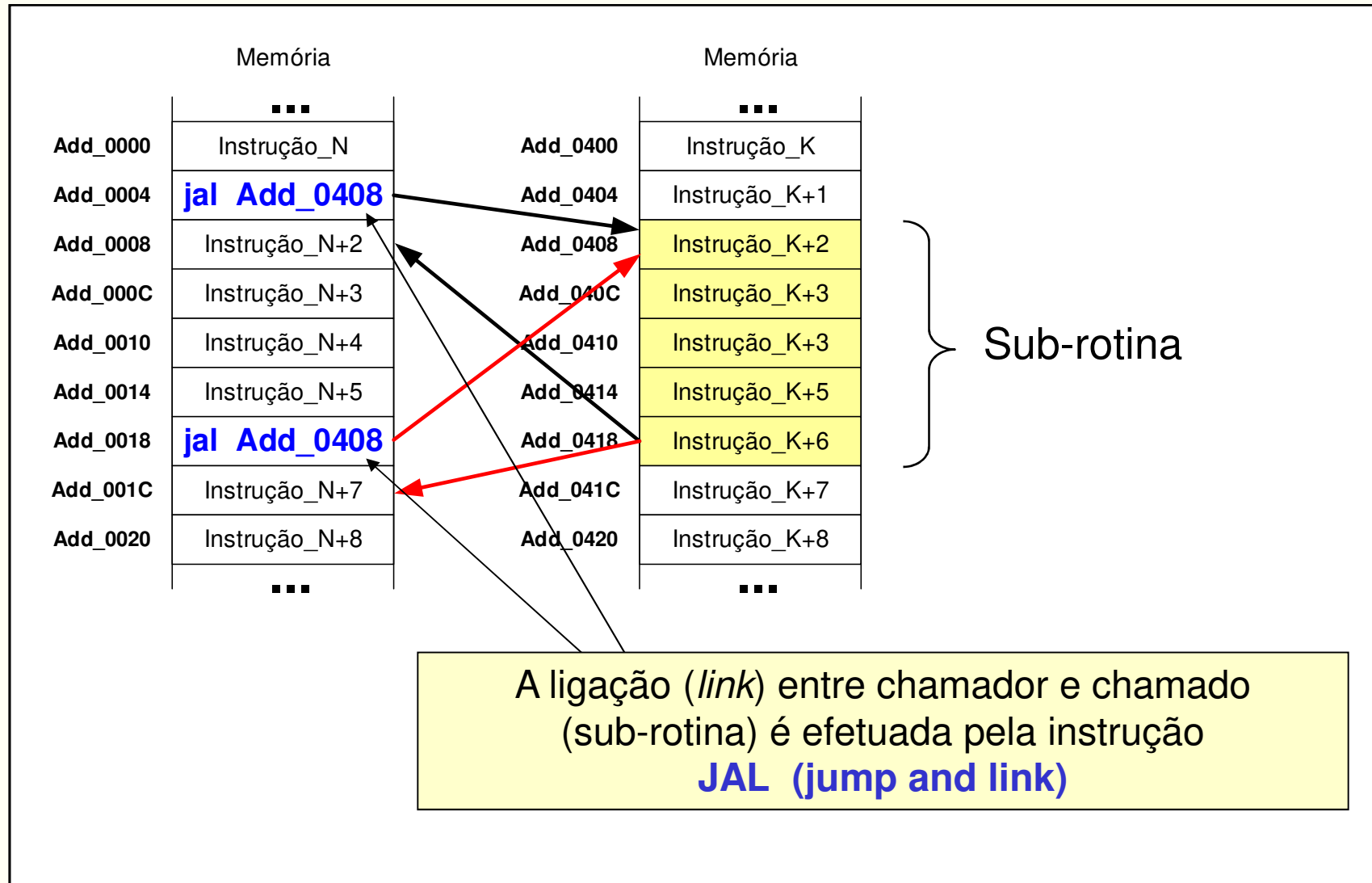
- Há três razões principais que justificam a existência de funções\*:
  - A **reutilização no contexto de um determinado programa** - aumento da eficiência na dimensão do código, substituindo a repetição de um mesmo trecho de código por um único trecho evocável de múltiplos pontos do programa
  - A **reutilização no contexto de um conjunto de programas**, permitindo que o mesmo código possa ser reaproveitado (bibliotecas de funções)
  - A **organização e estruturação do código**

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

# Sub-rotinas: exemplo

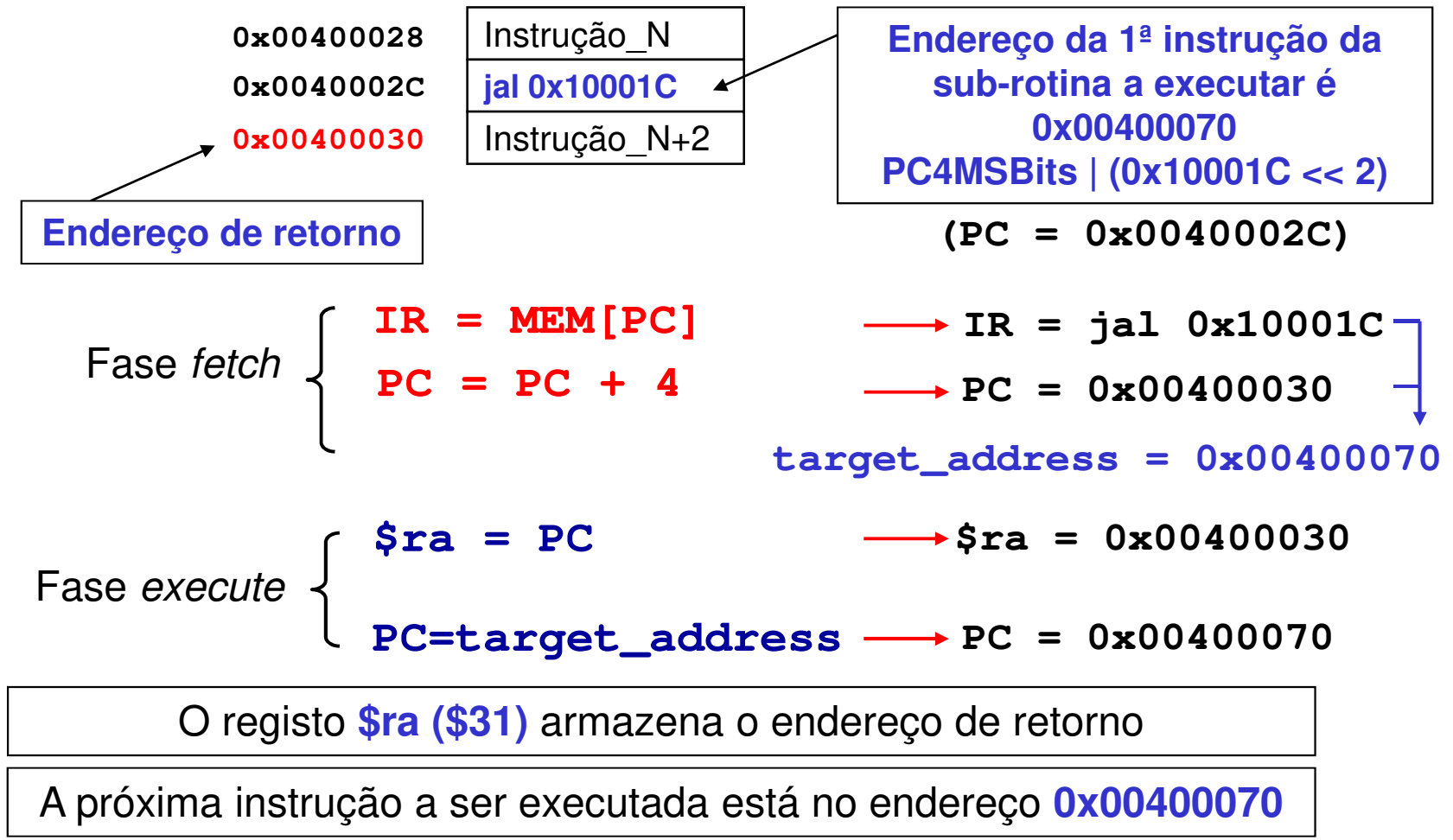


# Sub-rotinas: instrução JAL



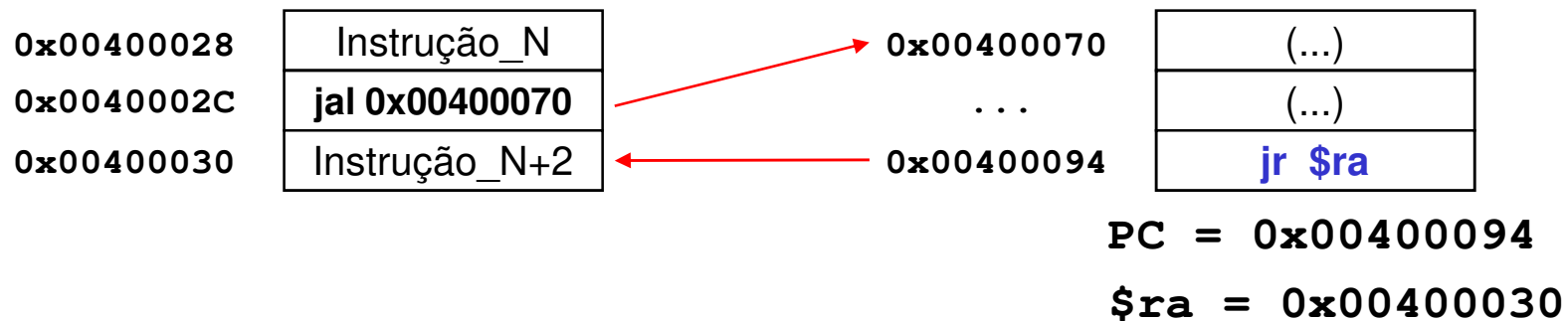
# Ciclo de execução da instrução JAL

- ***jal target\_address***



# Ciclo de execução da instrução JR

- Como **regressar** à instrução que sucede à instrução **"jal"** ?
- Aproveita-se o endereço de retorno armazenado em **\$ra** durante a execução da instrução **"jal"** (instrução **"jr register"**)

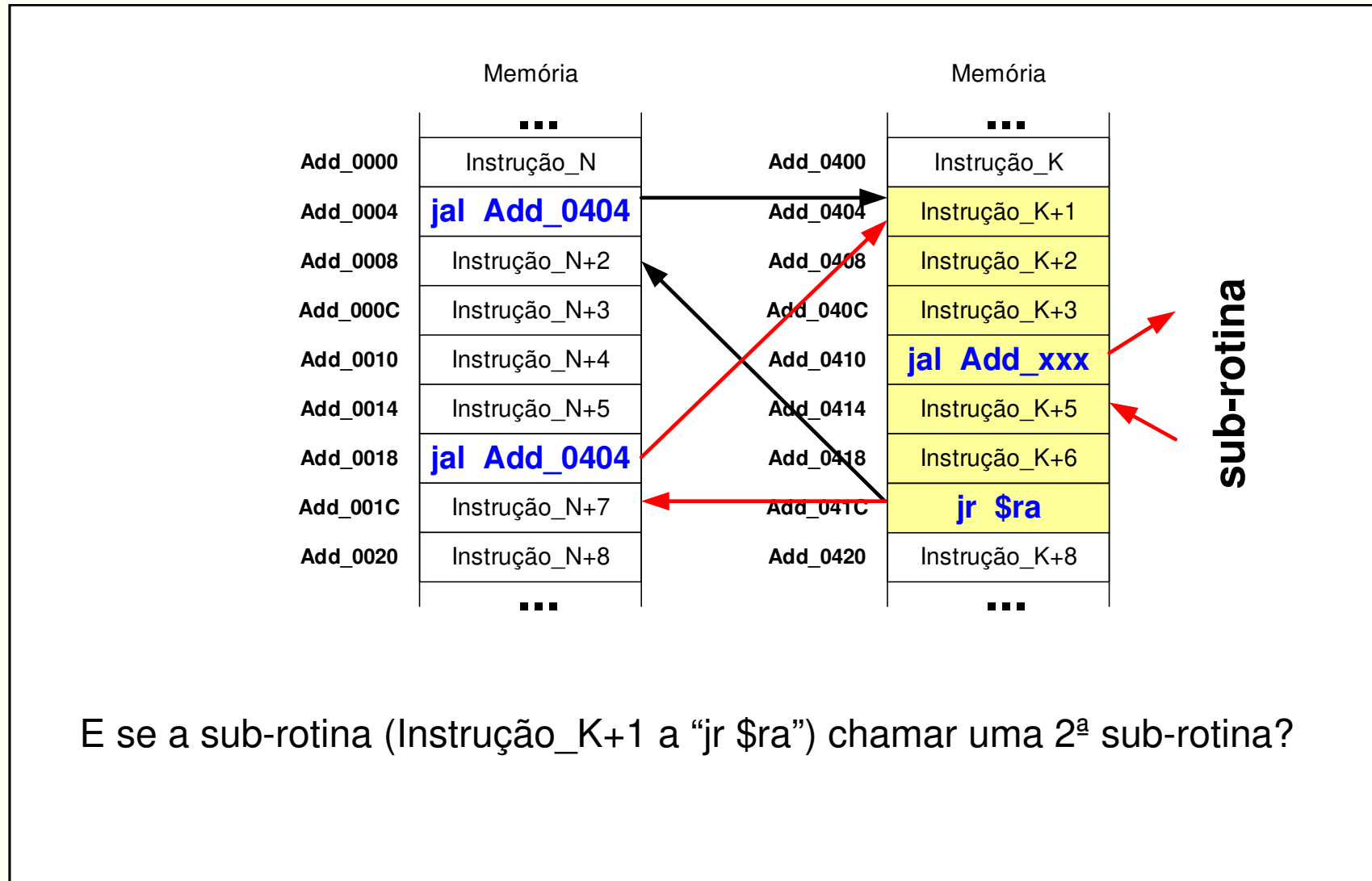


Fase *fetch* { **IR = MEM[PC]** → IR = jr \$ra  
                  **PC = PC + 4** → PC = 0x00400098

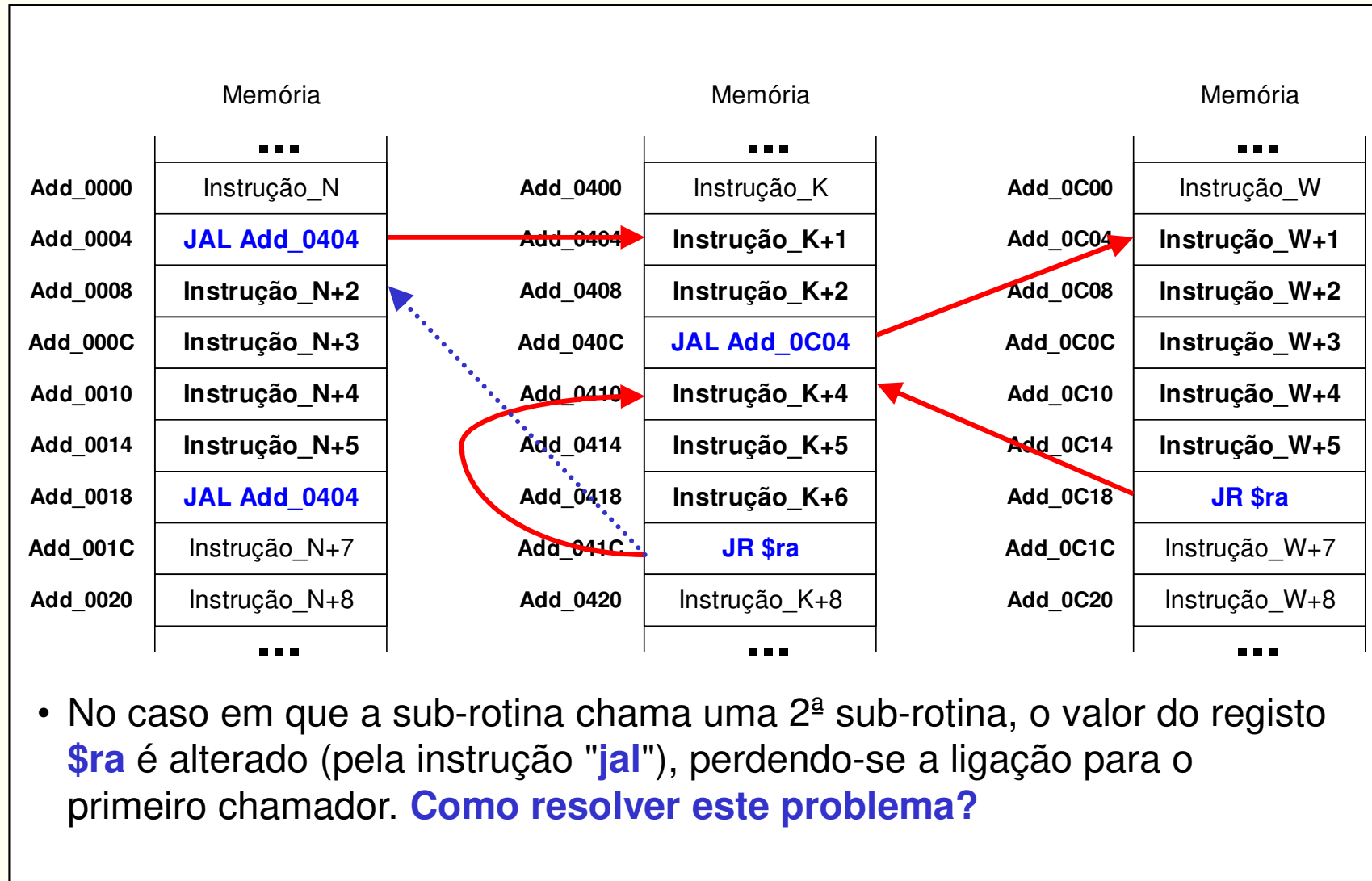
Fase *execute* { **PC = \$ra** → PC = 0x00400030

A próxima instrução a ser executada está no endereço **0x00400030**

## Chamada a uma sub-rotina a partir de outra sub-rotina

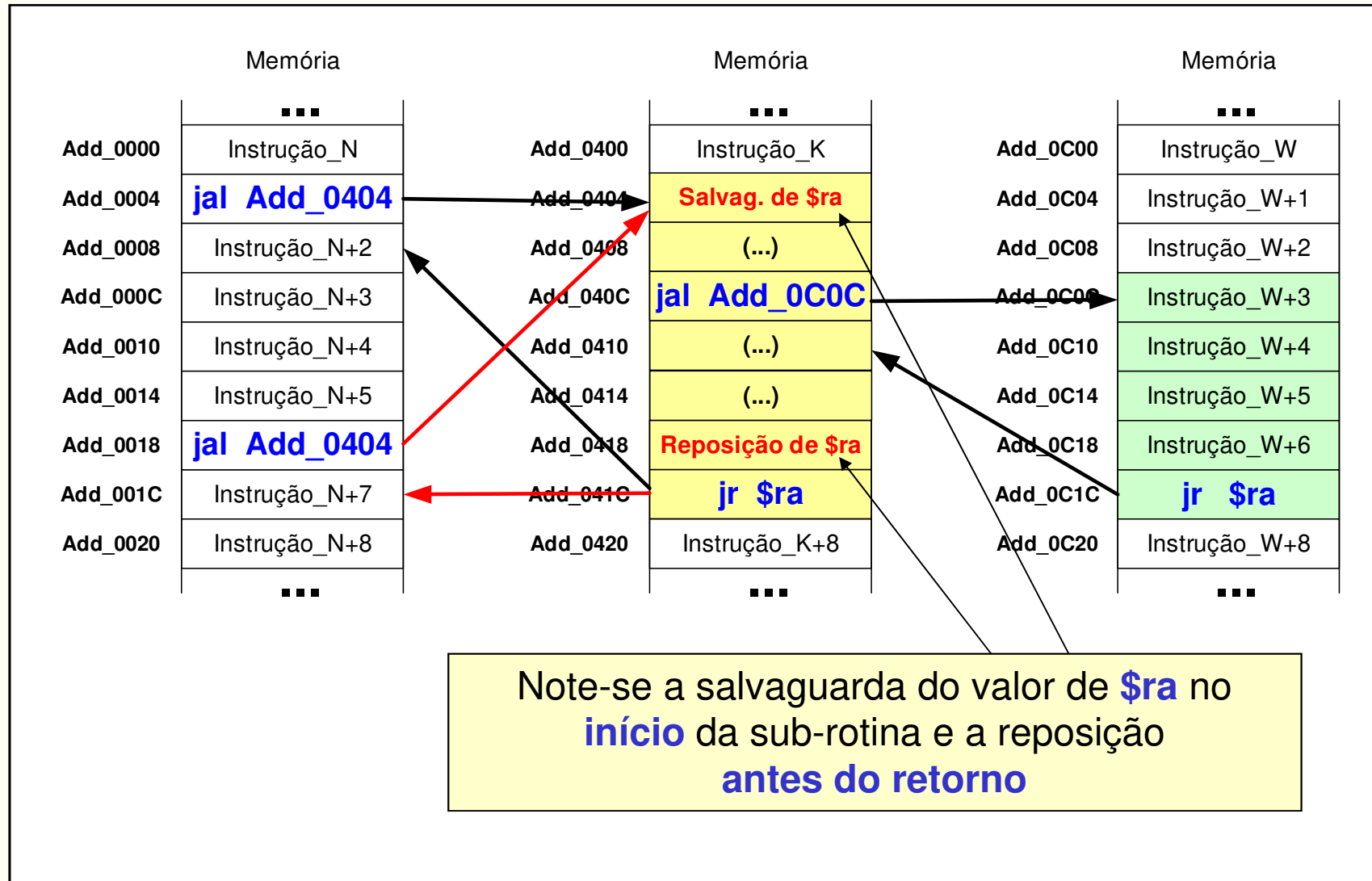


## Chamada a uma sub-rotina a partir de outra sub-rotina





## Chamada a uma sub-rotina a partir de outra sub-rotina

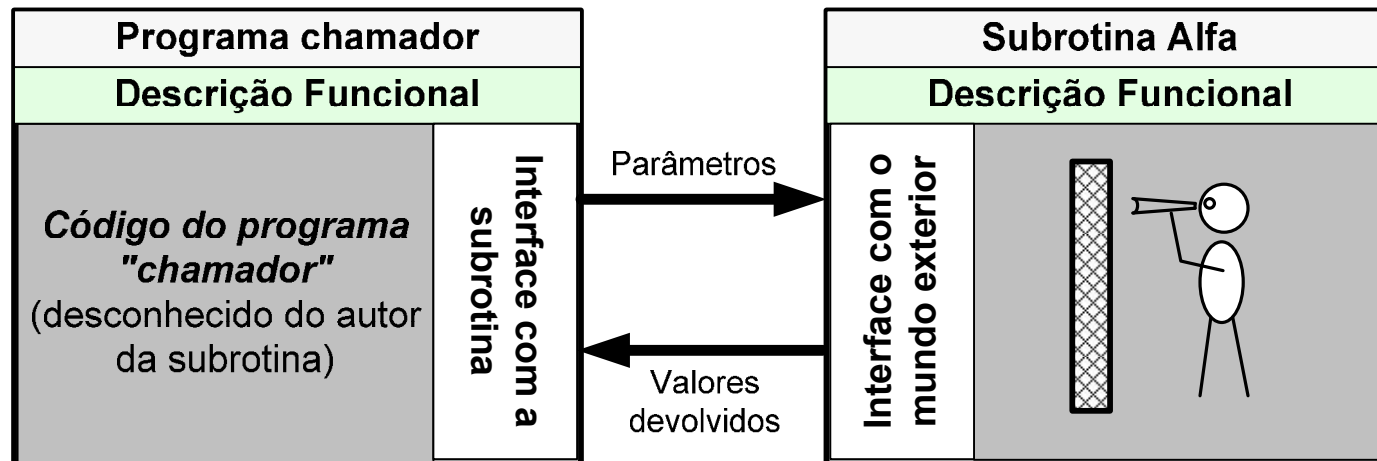


## Sub-rotinas – perspectiva do utilizador (programador)

- A **reutilização das sub-rotinas** torna-as particularmente atrativas, em especial quando suportam funcionalidades básicas, quer do ponto de vista computacional como do ponto de vista do interface entre o computador, os periféricos e o utilizador humano
- As sub-rotinas surgem, assim, frequentemente agrupadas em **bibliotecas**, a partir das quais podem ser evocadas por qualquer programa externo
- Este facto determina que o recurso a sub-rotinas escritas por outros para serviço dos nossos programas, **não deverá implicar necessariamente o conhecimento dos detalhes da sua implementação**
- Geralmente, o acesso ao código fonte da sub-rotina (conjunto de instruções originalmente escritas pelo programador) não é sequer possível, a menos que o mesmo seja tornado público pelo seu autor

# Sub-rotinas – perspectiva do programador

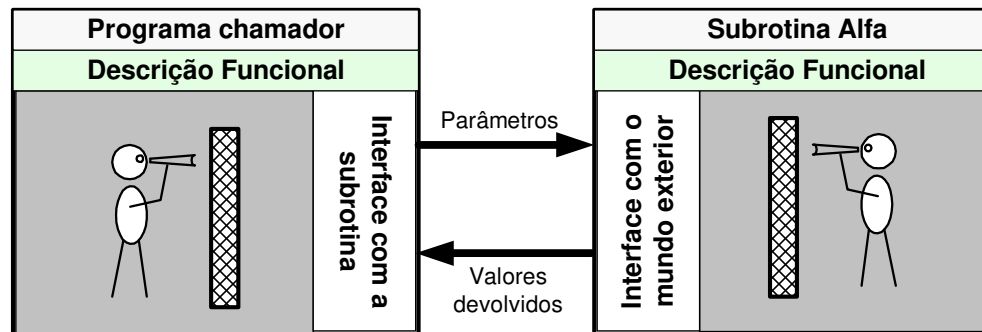
- Na perspectiva do programador, a sub-rotina que este tem a responsabilidade de escrever é um **trecho de código isolado**, com uma funcionalidade bem definida, e com um interface que ele próprio pode determinar em função das necessidades
- O facto de a sub-rotina ter de ser escrita para ser reutilizada implica que o programador não conhece antecipadamente as características do programa que irá evocar o seu código



# Regras a definir entre chamador e a sub-brotina chamada

- Torna-se óbvia a necessidade de definir um conjunto de **regras que regulem a relação entre o programa “chamador” e a sub-rotina “chamada”**: a) definição do interface entre ambos e b) princípios que assegurem uma “sã convivência” entre os dois!

Exemplo



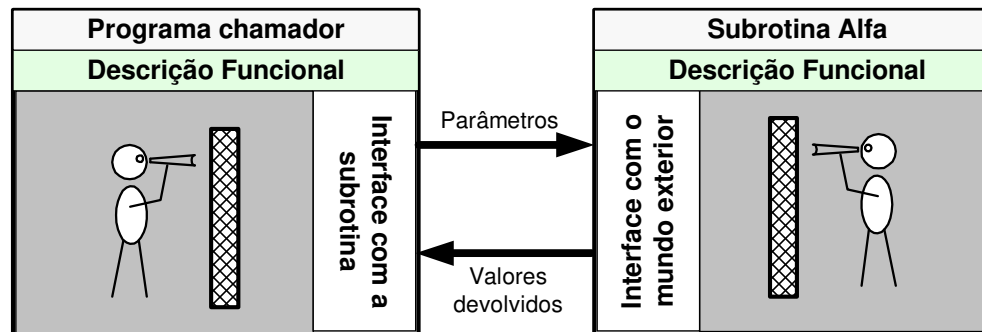
```
...  
li      $t0, 0  
11:     bge    $t0, 5, endf1  
...  
jal     sub1  
...  
addi    $t0, $t0, 1  
j       11  
endf1:  ...
```

Quantas vezes vai ser executado o ciclo do programa chamador?

# Regras a definir entre chamador e a sub-brotina chamada

- Torna-se óbvia a necessidade de definir um conjunto de **regras que regulem a relação entre o programa “chamador” e a sub-rotina “chamada”**: a) definição do interface entre ambos e b) princípios que assegurem uma “sã convivência” entre os dois!

Exemplo



```
...  
li      $t0, 0  
11:     bge    $t0, 5, endf1  
...  
jal     sub1  
...  
addi    $t0, $t0, 1  
j       11  
endf1:  ...
```

```
sub1:   li      $t0, 3  
12:     ble    $t0, 0, endf2  
...  
        addi    $t0, $t0, -1  
        j       12  
endf2:  jr      $ra
```

Quantas vezes vai ser executado o ciclo do programa chamador?

## Regras a definir entre chamador e a sub-brotina 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 “sã convivência”:
  - Que registos do CPU podem “chamador” e “chamado” usar, sem que haja alteração indevida de informação (por exemplo um alterar o conteúdo de um registo que está simultaneamente a ser usado pelo outro)
  - Como partilhar a memória usada para armazenar dados, sem risco de sobreposição (e consequente perda de informação armazenada)

## Convenção para a passagem de parâmetros no MIPS

- Os parâmetros que possam ser armazenados na dimensão de um registo (32 bits, i.e., **char**, **int**, **ponteiros**) devem ser passados à sub-rotina 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 \$ai 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 co-processador de vírgula flutuante (ponteiros são passados nos registos \$ai)

# Convenção para a devolução de valores no MIPS

- A sub-rotina pode devolver um valor de 32 bits ou um de 64 bits:
  - Se o valor a devolver é de **32 bits** é utilizado o registo **\$v0**
  - Se o valor a devolver é de **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 co-processador de vírgula flutuante



# Exemplo (chamador)

```
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 da sub-rotina

parâmetros

evocação da sub-rotina

valor devolvido

## Exemplo (sub-rotina)

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

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

Note-se que , para escrever o código da sub-rotina, 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
```

parâmetros

Valor a devolver

regresso ao chamador

Será necessário salvar o valor de \$ra?

# Estratégias para a salvaguarda de registos

- Que registos pode usar uma sub-rotina, 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”! Nesse caso, o que fazer quando o “chamado” é simultaneamente “chamador” (sub-rotina que chama outra sub-rotina)?
- Outra hipótese, mais praticável, consiste em atribuir a um dos “parceiros” a responsabilidade de copiar previamente para a memória externa o conteúdo de qualquer registo que pretenda utilizar (**salvaguardar o registo**) e repor, posteriormente, o valor original lá armazenado

# Estratégias para a salvaguarda de registos

- Estratégia “**caller-saved**”
  - Deixa-se ao cuidado do programa “chamador” a responsabilidade de salvaguardar o conteúdo da totalidade dos registos antes de evocar a sub-rotina
  - Cabe-lhe também a tarefa de repor posteriormente o seu valor
  - No limite, é admissível que o “chamador” salvaguarde apenas o conteúdo dos registos de que venha a precisar mais tarde
- Estratégia “**callee-saved**”
  - Entrega-se à sub-rotina a responsabilidade pela prévia salvaguarda dos registos de que possa necessitar
  - Assegura, igualmente, a tarefa de repor o seu valor imediatamente antes de regressar ao programa “chamador”

## Convenção para salvaguarda de registos no MIPS

- No caso do MIPS, a estratégia adotada é uma versão mista das anteriores, e baseia-se nas duas regras seguintes:
  - Os registos **\$t0..\$t9**, **\$v0..\$v1** e **\$a0..\$a3** podem ser livremente utilizados e alterados pelas sub-rotinas
  - Os valores dos registos **\$s0..\$s7** não podem, **na perspetiva do chamador**, ser alterados pelas sub-rotinas
- Então, se os registos **\$t0..\$t9**, **\$v0..\$v1** e **\$a0..\$a3** podem ser livremente utilizados e alterados pelas sub-rotinas
  - Um programa “chamador” que esteja a usar um ou mais destes registos, deverá salvaguardar o seu conteúdo antes de evocar uma sub-rotina, sob pena de que esta os venha a alterar

## Convenção para salvaguarda de registos no MIPS

- Os valores dos registos **\$s0..\$s7** não podem, na perspetiva do chamador, ser alterados pelas sub-rotinas
  - Se uma dada sub-rotina precisar de usar um registo do tipo **\$sn**, compete a essa sub-rotina **copiar previamente o seu conteúdo** para um lugar seguro (memória externa), repondo-o imediatamente antes de terminar
  - Dessa forma, do ponto de vista do programa “chamador” (que não “vê” o código da sub-rotina) é como se esse registo não tivesse sido usado ou alterado

## Considerações práticas sobre a utilização da convenção

- **sub-rotinas terminais** (sub-rotinas folha, i.e., que não chamam qualquer sub-rotina)
  - Só devem utilizar (preferencialmente) registos que não necessitam de ser salvaguardados (**\$t0..\$t9**, **\$v0..\$v1** e **\$a0..\$a3**)
- **sub-rotinas que chamam outras sub-rotinas**
  - Devem utilizar os registos **\$s0..\$s7** para o armazenamento de valores que se pretenda preservar. A utilização destes registos implica a sua prévia salvaguarda na memória externa logo no início da sub-rotina e a respetiva reposição no final
  - Devem utilizar os registos \$t0..\$t9, \$v0..\$v1 e \$a0..\$a3 para os restantes valores

# Utilização da convenção - exemplo

- O problema detetado na codificação do programa chamador e da sub-rotina dos slides 12 e 13 pode facilmente ser resolvido se a convenção de salvaguarda de registos for aplicada
- A variável índice do ciclo do programa chamador passará a residir num registo **\$sn** (por exemplo no \$s0) – registo que, **garantidamente**, a sub-rotina não vai alterar

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

```
(...) # Salv. $s0
...
li      $s0, 0
11:     bge  $s0, 5, endf1
...
jal     sub1
...
addi    $s0, $s0, 1
j       11
endf1:  ...
(...) # Repoe $s0
```

```
sub1:   li      $t0, 3
12:     ble    $t0, 0, endf2
...
addi    $t0, $t0, -1
j       12
endf2:  jr      $ra
```

Quantas vezes vai ser executado o ciclo do programa chamador?