

Aulas 9 e 10

- Sub-rotinas: chamada e retorno
- Caracterização das sub-rotinas na perspetiva do "chamador" e do "chamado"
- Convenções adotadas na arquitetura MIPS quanto à:
 - passagem de parâmetros para sub-rotinas
 - devolução de valores de sub-rotinas
 - utilização e salvaguarda de registos
- A *stack* - conceito e operações básicas
- Utilização da *stack* na arquitetura MIPS
- Análise de um exemplo.

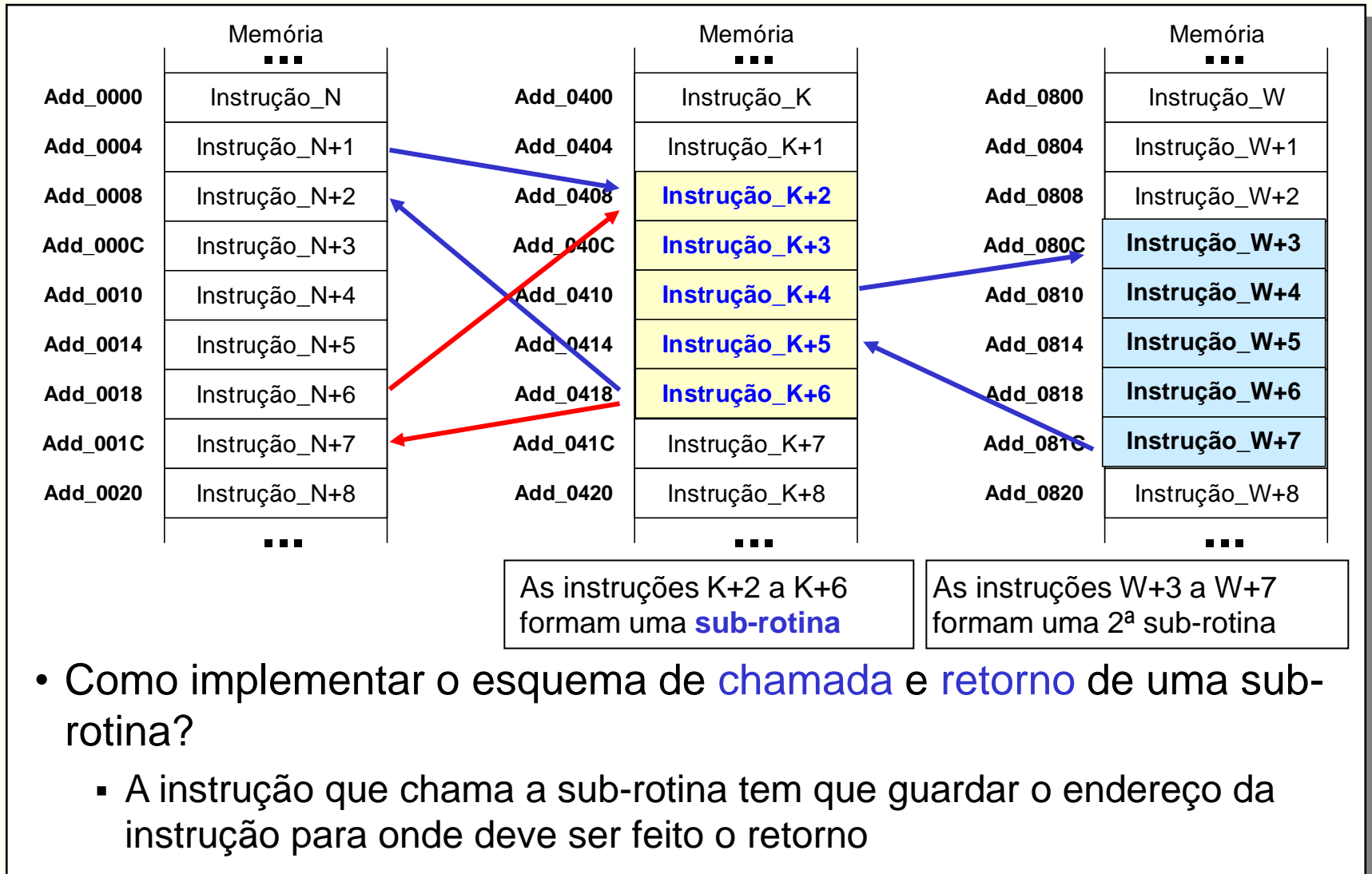
Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Porque se usam funções (sub-rotinas)?

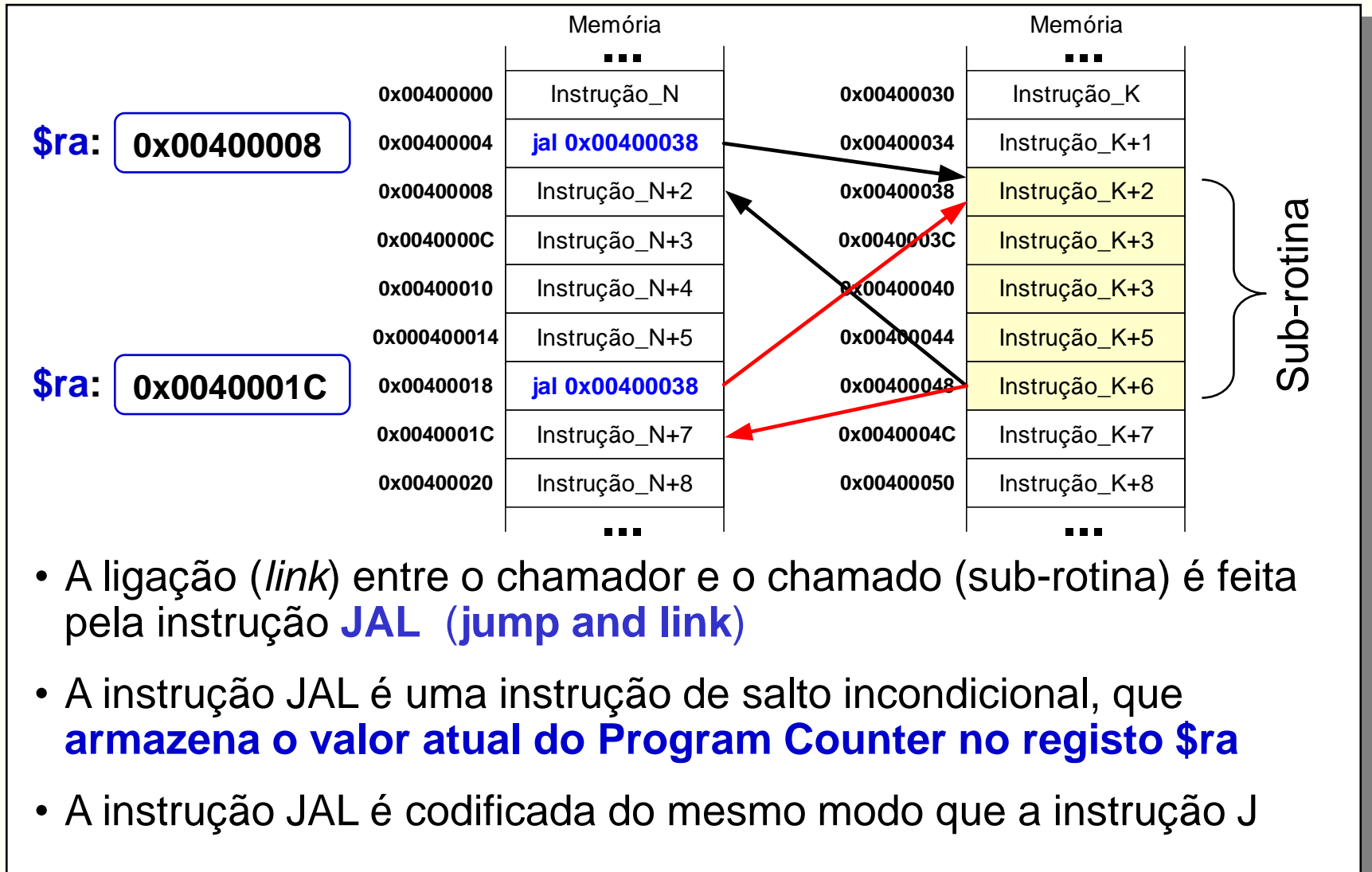
- Há três razões principais que justificam a existência de funções*:
 - A **reutilização no contexto de um determinado programa** - aumento da eficiência na dimensão do código, substituindo a repetição de um mesmo trecho de código por um único trecho evocável de múltiplos pontos do programa
 - A **reutilização 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



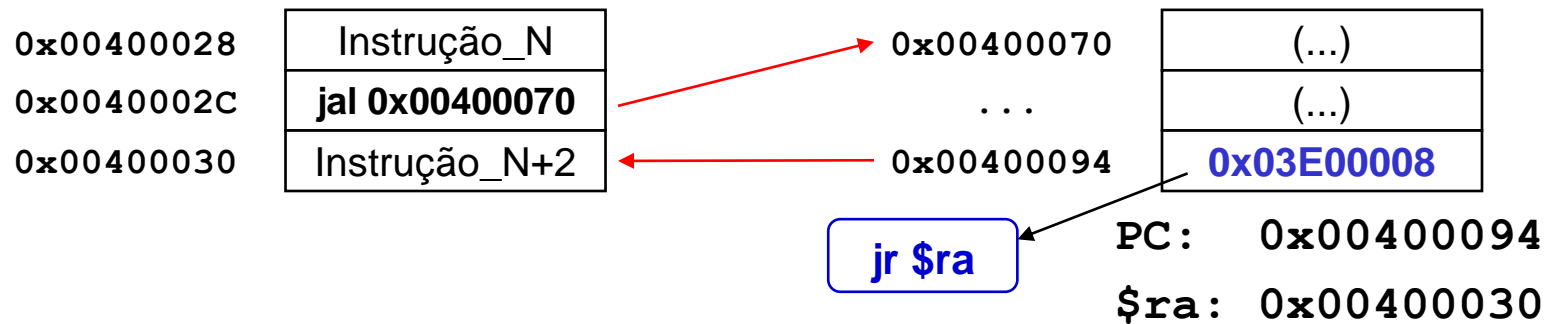
Sub-rotinas no MIPS: a instrução JAL



- A ligação (*link*) entre o chamador e o chamado (sub-rotina) é feita pela instrução **JAL (jump and link)**
- A instrução JAL é uma instrução de salto incondicional, que **armazena o valor atual do Program Counter no registo \$ra**
- A instrução JAL é codificada do mesmo modo que a instrução J

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**")

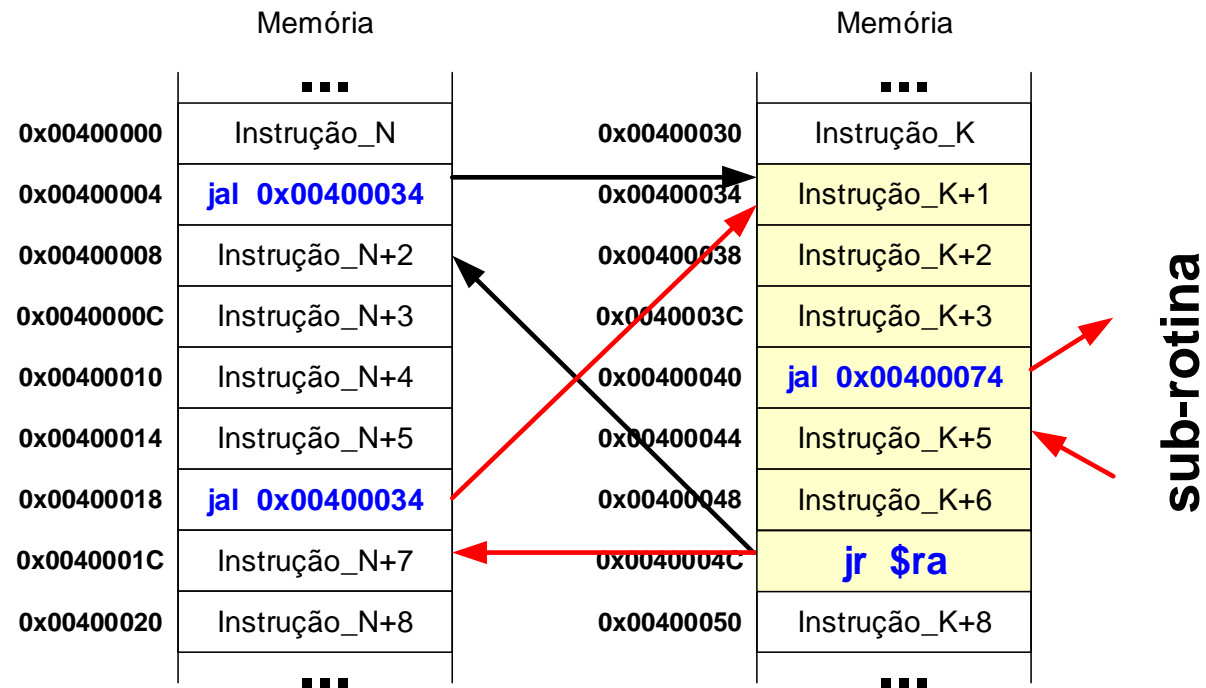


Fase *fetch* { **IR = MEM[PC]** → IR = 0x03E00008
PC = PC + 4 → PC = 0x00400098

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

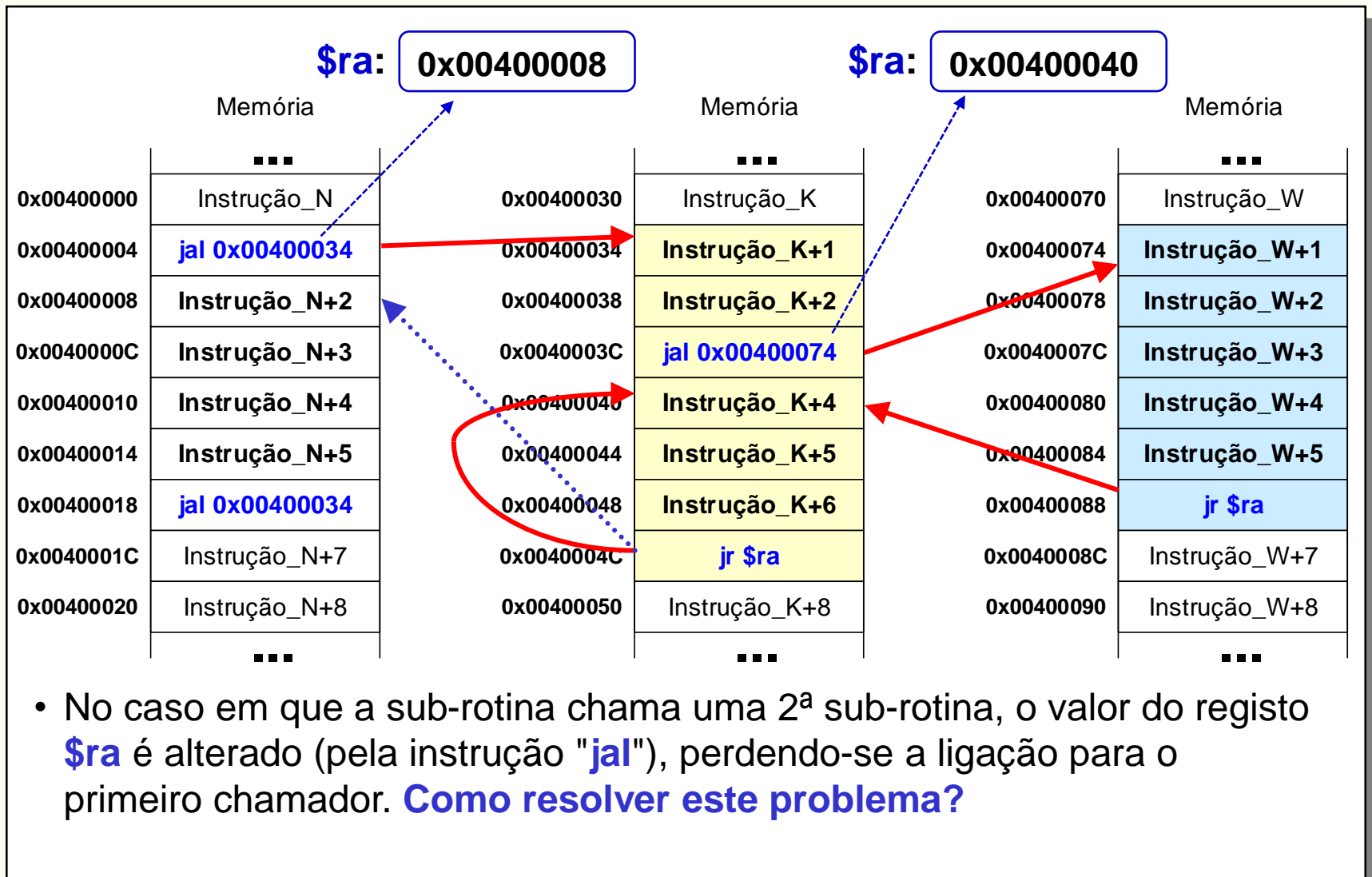
A próxima instrução a ser executada está no endereço **0x00400030**

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

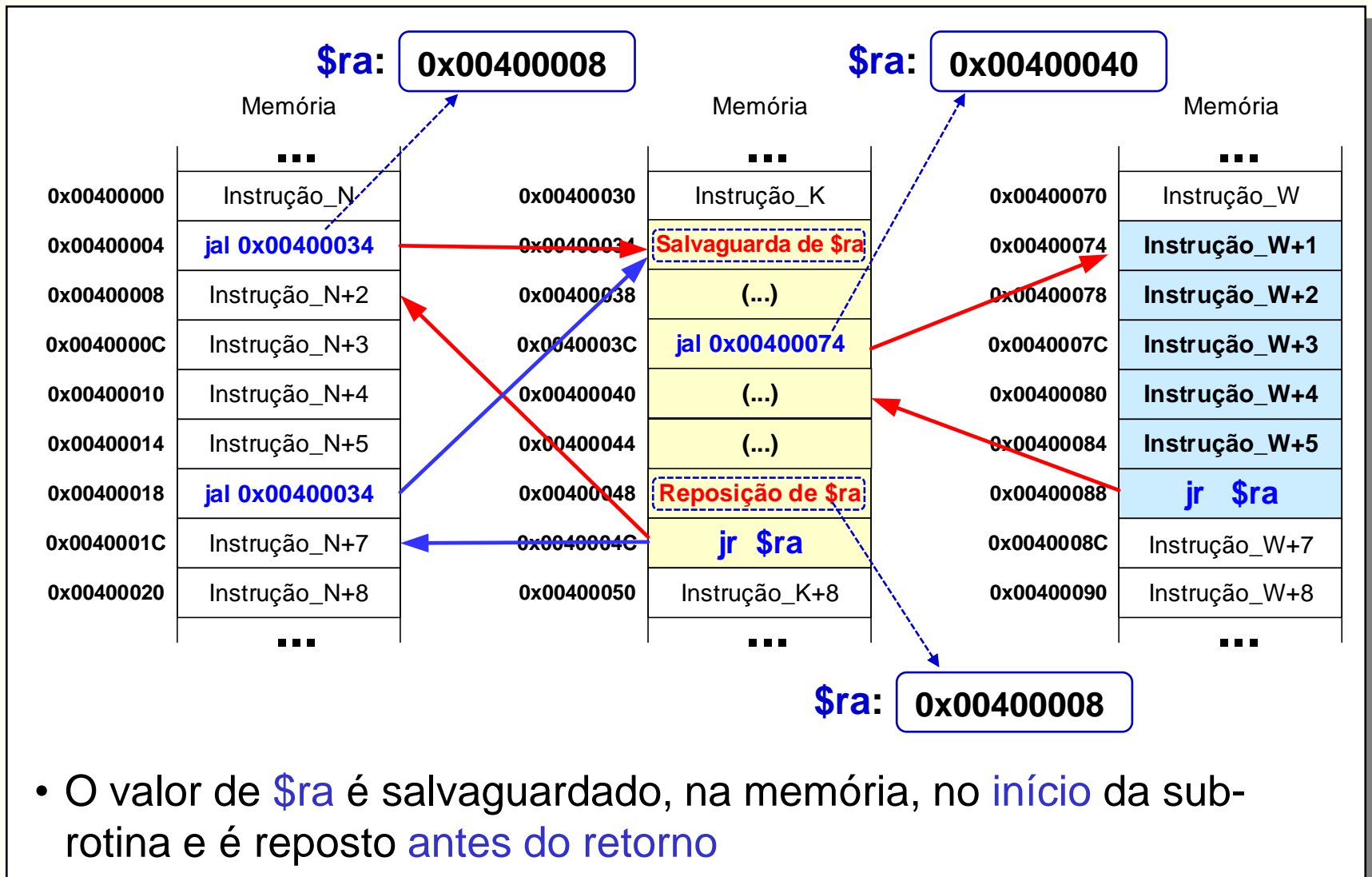


E se a sub-rotina (Instrução_K+1 a “jr \$ra”) chamar uma 2ª sub-rotina?

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



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



Instruções JAL e JALR

- A instrução "**jal**" é codificada do mesmo modo que a instrução "**j**": formato j em que os 26 bits menos significativos são obtidos dos 28 bits menos significativos do endereço-alvo, deslocados à direita dois bits
- Durante a execução, a obtenção do endereço-alvo é feita do mesmo modo que na instrução "**j**"
- A especificação de um endereço-alvo de 32 bits é possível através da utilização da instrução "**jalr**" (**jump and link register**)
- A instrução "**jalr**" funciona de modo idêntico à instrução "**jal**", exceto na obtenção do endereço-alvo:
 - o endereço da sub-rotina é lido do registo especificado na instrução (endereçamento indireto por registo); ex: **jalr \$t2**
- A instrução "**jalr**" é codificada com o formato R

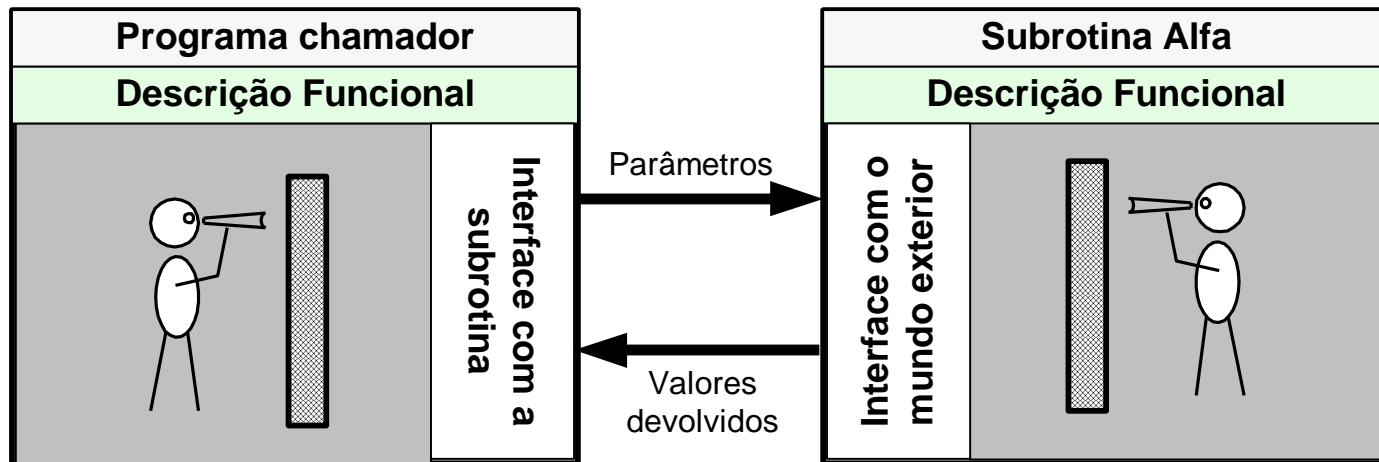
Sub-rotinas

- A **reutilização de sub-rotinas** é essencial em programação, 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 frequentemente agrupadas em **bibliotecas**, a partir das quais podem ser evocadas por qualquer programa
- A utilização de sub-rotinas escritas por outros para serviço dos nossos programas, **não deverá implicar 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

- Na perspetiva 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 uma interface que ele próprio pode determinar em função das necessidades
- O facto de uma sub-rotina ser escrita para poder ser reutilizada implica que o programador não conhece antecipadamente as características do programa que a irá chamar
- Torna-se óbvia a necessidade de definir um conjunto de **regras que regulem a relação entre o programa “chamador” e a sub-rotina “chamada”**:
 - definição da interface entre ambos, i.e., quais os parâmetros de entrada e como os passar para a sub-rotina e como devolver resultados ao programa chamador
 - princípios que assegurem uma “sã convivência” entre os dois, de modo a que um não destrua os dados do outro

Sub-rotinas



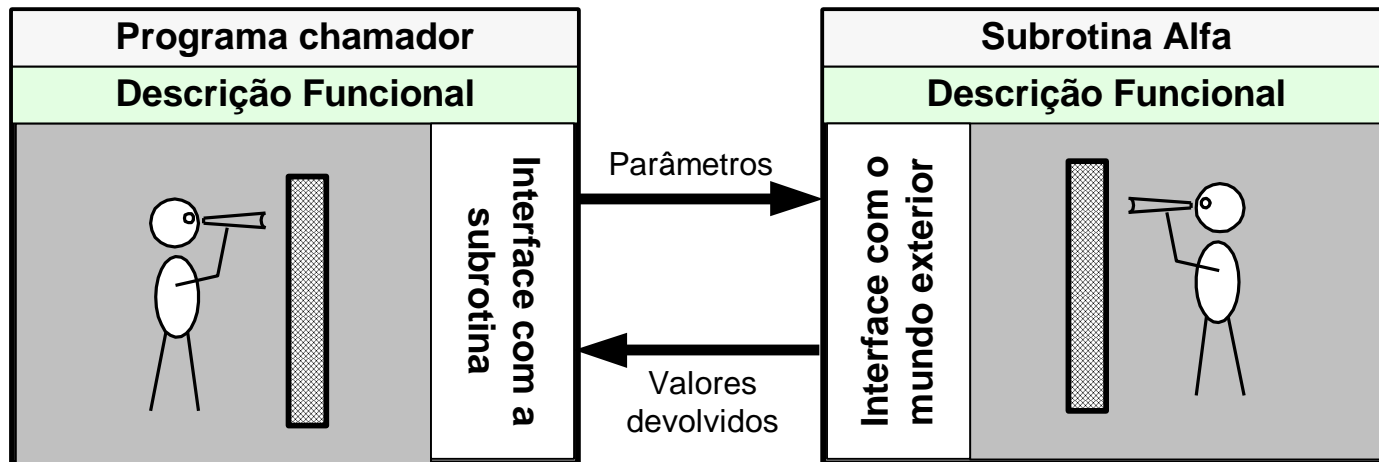
Exemplo (Programa chamador)

```
...  
li      $t0,0  
lab1:   bge     $t0,5,endif1  
...  
jal     sub1  
...  
addi    $t0,$t0,1  
j       lab1  
endif1: ...
```



Quantas vezes vai ser executado o ciclo do programa chamador?

Sub-rotinas



Exemplo (Programa chamador)

```

...
li      $t0,0
lab1:   bge    $t0,5,endif1
...
jal     sub1
...
addi    $t0,$t0,1
j       lab1
endif1: ...
    
```

(Sub-rotina)

```

sub1:   li      $t0,3
lab2:   ble     $t0,0,endif2
...
        addi    $t0,$t0,-1
        j       lab2
endif2: jr      $ra
    
```

Quantas vezes vai ser executado o ciclo do programa chamador?

Regras a definir entre chamador e a sub-rotina chamada

- Ao nível da interface:
 - Como **passar parâmetros** do “chamador” para a sub-rotina, identificar quantos e onde são passados
 - Como **devolver**, para o “chamador”, resultados produzidos pela sub-rotina
- Ao nível das regras de “sã convivência”:
 - Que registos do CPU podem “chamador” e sub-rotina usar, sem que haja alteração indevida de informação por parte da sub-rotina (por exemplo a sub-rotina alterar o conteúdo de um registo que é usado pelo chamador e que tem informação que não pode ser perdida)
 - Como partilhar a memória usada para armazenar dados, sem risco de sobreposição (e consequente perda de informação armazenada)

Convenções do MIPS (passagem e devolução de valores)

- 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 seja superior a quatro, os 4 primeiros são passados nos registos \$ai e os restantes (pela ordem em que são declarados) deverão ser passados na stack*
- 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)**

Exemplo (chamador)

```
int max(int, int);
void main(void)
{
    int val1=19;
    int val2=35;
    int maxVal;
    ...
    maxVal=max(val1, val2);
    print_int10(maxVal);
}
```

parâmetros passados em \$a0 e \$a1

chamada da sub-rotina

valor devolvido em \$v0

```
val1:    $t2
val2:    $t3
maxVal:  $a0
```

```
.text
main:   (...)    #Salvaguada $ra
        li       $t2, 19
        li       $t3, 35
        ...
        move     $a0,$t2
        move     $a1,$t3
        jal      max
        move     $a0,$v0
        li       $v0,1
        syscall
        (...)    #Repõe $ra
        jr       $ra
```

- Para escrever o programa “chamador”, não é necessário conhecer os detalhes de implementação da sub-rotina

Exemplo (sub-rotina)

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

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

vmax: \$t0

Em Assembly:

```
max:  move    $t0, $a0
      ble     $a1, $t0, endif
      move    $t0, $a1
endif: move    $v0, $t0
      jr      $ra
```

parâmetros

Valor a devolver

regresso ao chamador

- Para escrever o código da sub-rotina, não é necessário conhecer os detalhes de implementação do “chamador”
- Será necessário salvaguardar 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 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
 - Essa responsabilidade pode ser atribuída ao chamador ou à sub-rotina (ou aos dois)

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 chamar a sub-rotina
 - Cabe-lhe também a tarefa de repor posteriormente o seu valor
 - No limite, é admissível que o “chamador” salvasse apenas os registos de cujo conteúdo venha a precisar mais tarde
- Estratégia “**callee-saved**”
 - Entrega-se à sub-rotina a responsabilidade pela prévia salvaguarda dos registos que vai usar
 - Assegura, igualmente, a tarefa de repor o seu valor imediatamente antes de regressar ao programa “chamador”

Convenção para salvaguarda de registos no MIPS

- Os registos **\$t0** a **\$t9**, **\$v0** e **\$v1**, e **\$a0** a **\$a3** podem ser livremente utilizados e alterados pelas sub-rotinas
- Os registos **\$s0** a **\$s7** não podem, **na perspetiva do chamador**, ser alterados pelas sub-rotinas
 - Se uma dada sub-rotina precisar de usar qualquer um dos registos **\$s0** a **\$s7** compete a essa sub-rotina **salvaguardar previamente o seu conteúdo**, repondo-o imediatamente antes de terminar
 - Ou seja, é seguro para o programa chamador usar um registo **\$sn** para armazenar um valor que vai necessitar após a chamada à sub-rotina, uma vez que tem a garantia que esta não o modifica

Considerações práticas sobre a utilização da convenção

- **sub-rotinas terminais** (sub-rotinas folha, i.e., que não chamam qualquer sub-rotina)
 - Só devem utilizar registos que não têm a responsabilidade de salvar (salvar) (\$t0..\$t9, \$v0..\$v1 e \$a0..\$a3)
- **sub-rotinas intermédias** (sub-rotinas que chamam outras sub-rotinas)
 - Devem utilizar os registos \$s0..\$s7 para o armazenamento de valores que pretendam preservar durante a chamada à sub-rotina seguinte
 - A utilização de qualquer um dos registos \$s0 a \$s7 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 13 e 14 pode facilmente ser resolvido se a convenção de salvaguarda de registos for aplicada

O código da sub-rotina é desconhecido do programador do “chamador” e vice-versa

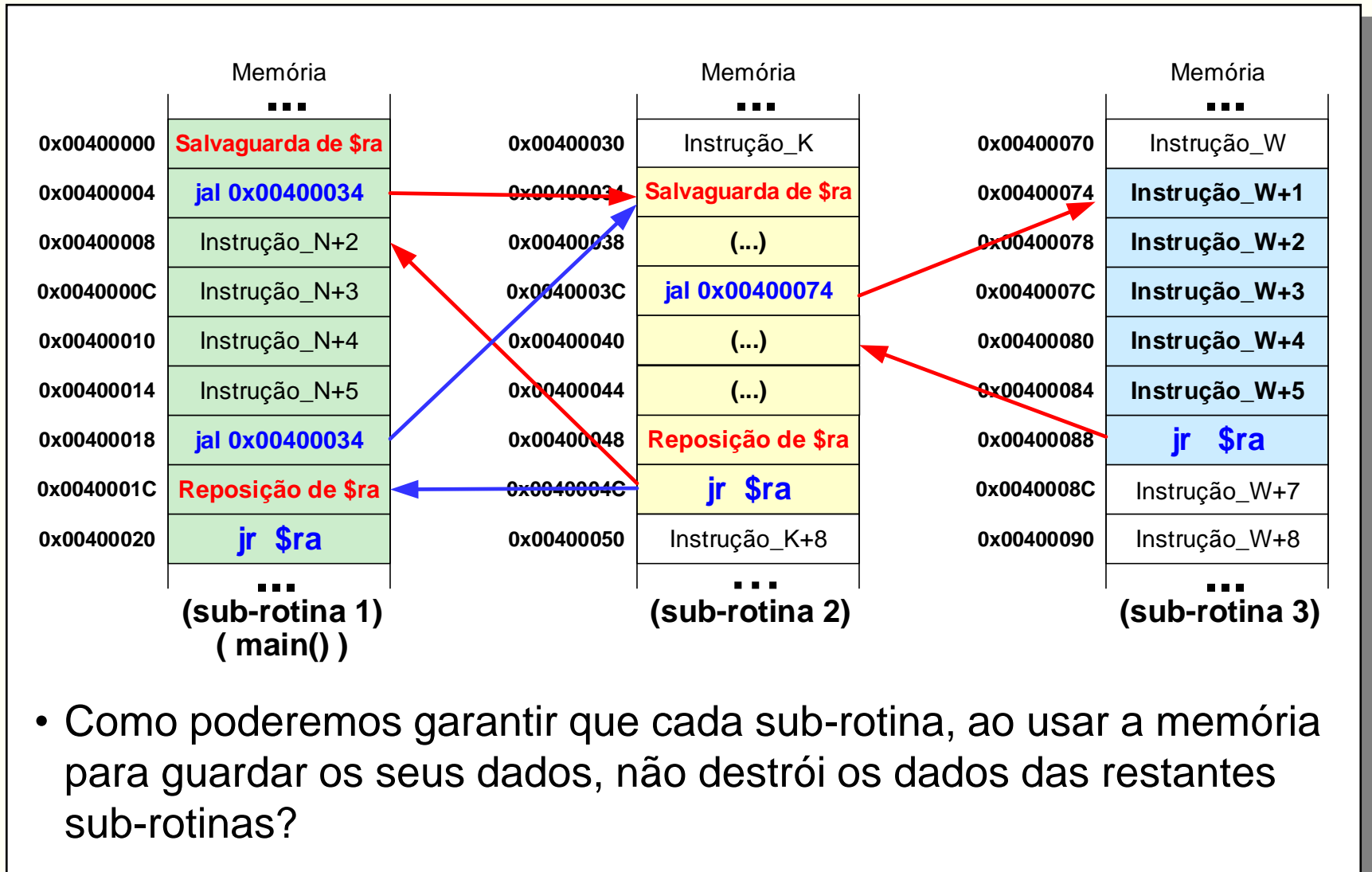
```
(...) # Salv. $s0
...
li      $s0,0
lab1:   bge    $s0,5,endif1
...
jal     sub1
...
addi    $s0,$s0,1
j       lab1
endif1: ...
(...) # Repoe $s0
```

```
sub1:   li      $t0,3
lab2:   ble     $t0,0,endif2
...
addi    $t0,$t0,-1
j       lab2
endif2: jr      $ra
```

Quantas vezes vai ser executado o ciclo do programa chamador?

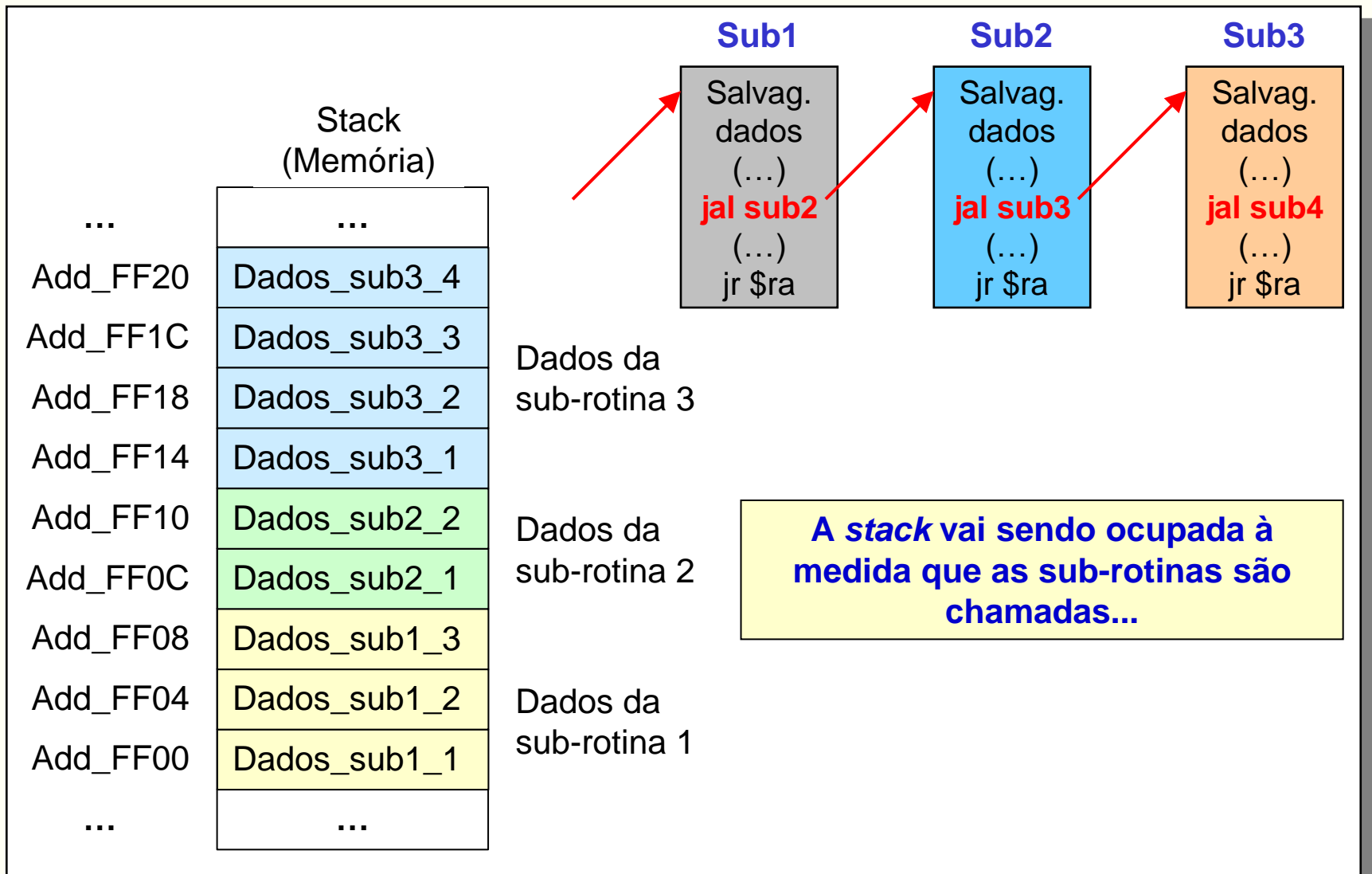
- A variável de controlo do ciclo do chamador deverá residir num registo **\$sn** (por exemplo no \$s0) – registo que, **garantidamente**, a sub-rotina não vai alterar

Armazenamento temporário de informação

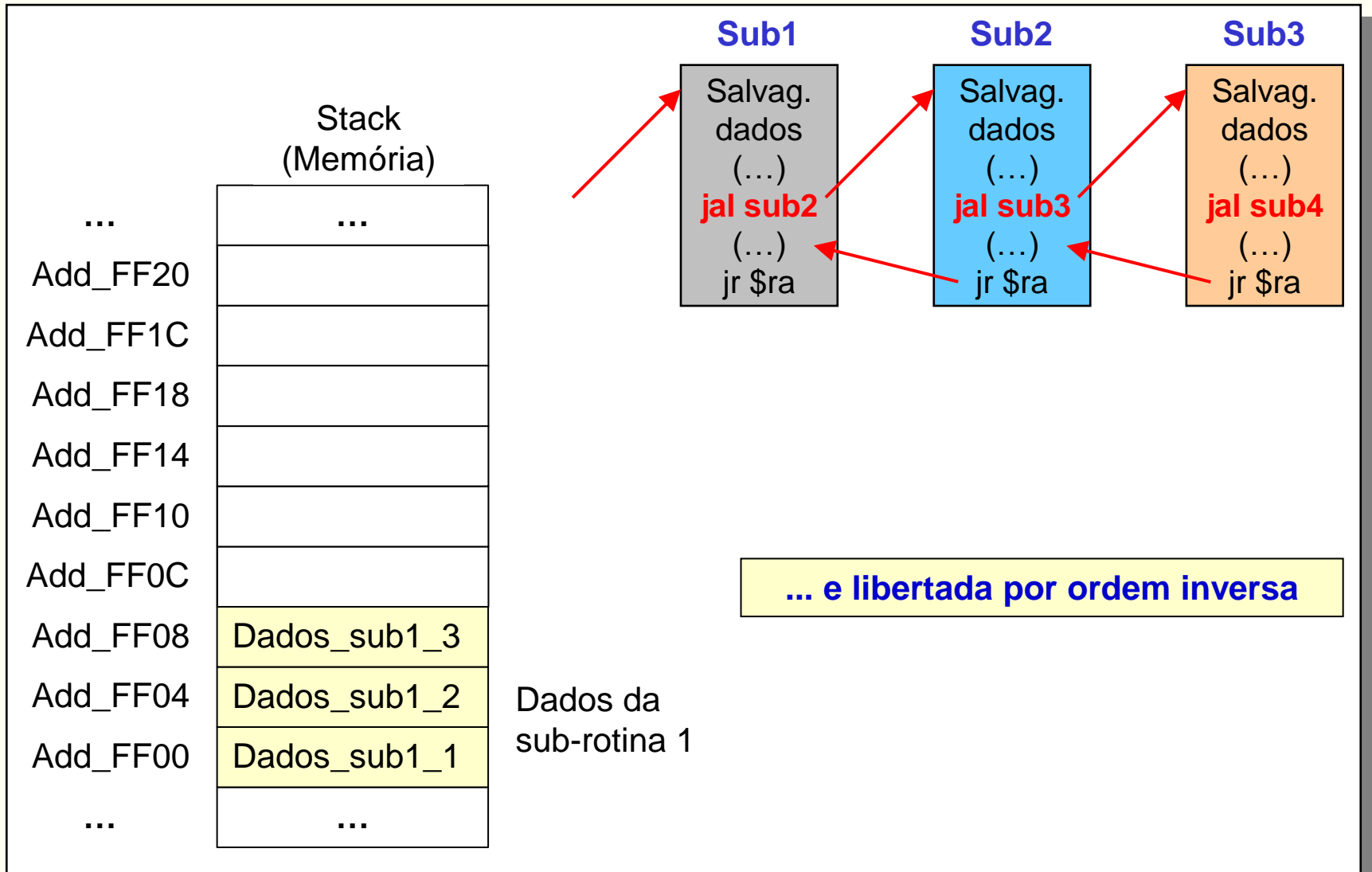


- Como poderemos garantir que cada sub-rotina, ao usar a memória para guardar os seus dados, não destrói os dados das restantes sub-rotinas?

Stack: espaço de armazenamento temporário



Stack: espaço de armazenamento temporário

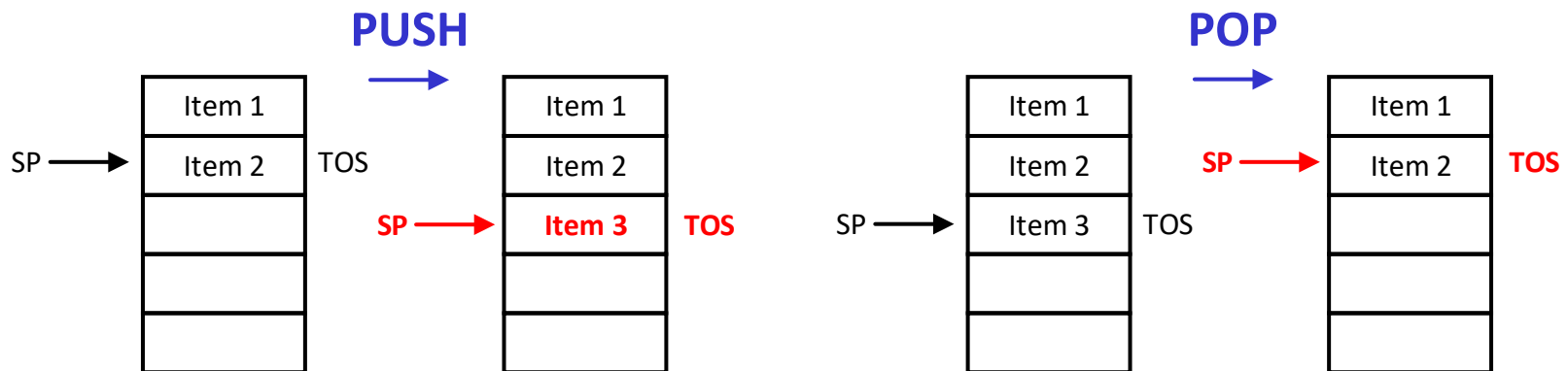


Stack: espaço de armazenamento temporário

- A estratégia de gestão dinâmica do espaço de memória - em que a última informação acrescentada é a primeira a ser retirada – é designada por **LIFO** (*Last In First Out*)
- A estrutura de dados correspondente é conhecida por **stack** ("pilha")
- As *stacks* são de tal forma importantes que muitas arquiteturas suportam diretamente instruções específicas para a sua manipulação (por exemplo a arquitetura Intel x86)
- A operação que permite acrescentar informação à *stack* é normalmente designada por **PUSH**, enquanto que a operação inversa é conhecida por **POP**

Stack: operações *push* e *pop*

- Estas operações têm associado um registo designado por **Stack Pointer (SP)**
- O registo **Stack Pointer** mantém, de forma permanente, o **endereço do topo da stack (TOS - top of stack)** e aponta sempre **para o último endereço ocupado**
 - Numa operação de **PUSH** é necessário pré-atualizar o *stack pointer* antes de copiar informação para a *stack*
 - Numa operação de **POP** é feita uma leitura da *stack*, do endereço atual do *stack pointer*, seguida de atualização do valor do *stack pointer*



Atualização do *stack pointer*

- A atualização do ***stack pointer***, numa operação de *push* (escrita de informação), pode seguir uma de duas estratégias:
 - Ser incrementado, fazendo crescer a *stack* no sentido crescente dos endereços
 - Ser decrementado, fazendo crescer a *stack* no sentido decrescente dos endereços
- A estratégia de crescimento da *stack* no sentido dos endereços mais baixos é, geralmente, a adotada
- A estratégia de crescimento da *stack* no sentido dos endereços mais baixos permite uma gestão simplificada da fronteira entre os segmentos de dados e de *stack*

Regras de utilização da *stack* na arquitetura MIPS

1. O registo **\$sp** (*stack pointer*) contém o endereço da **última posição ocupada** da *stack*

\$sp = \$29

\$SP: **0xFFFF0010**

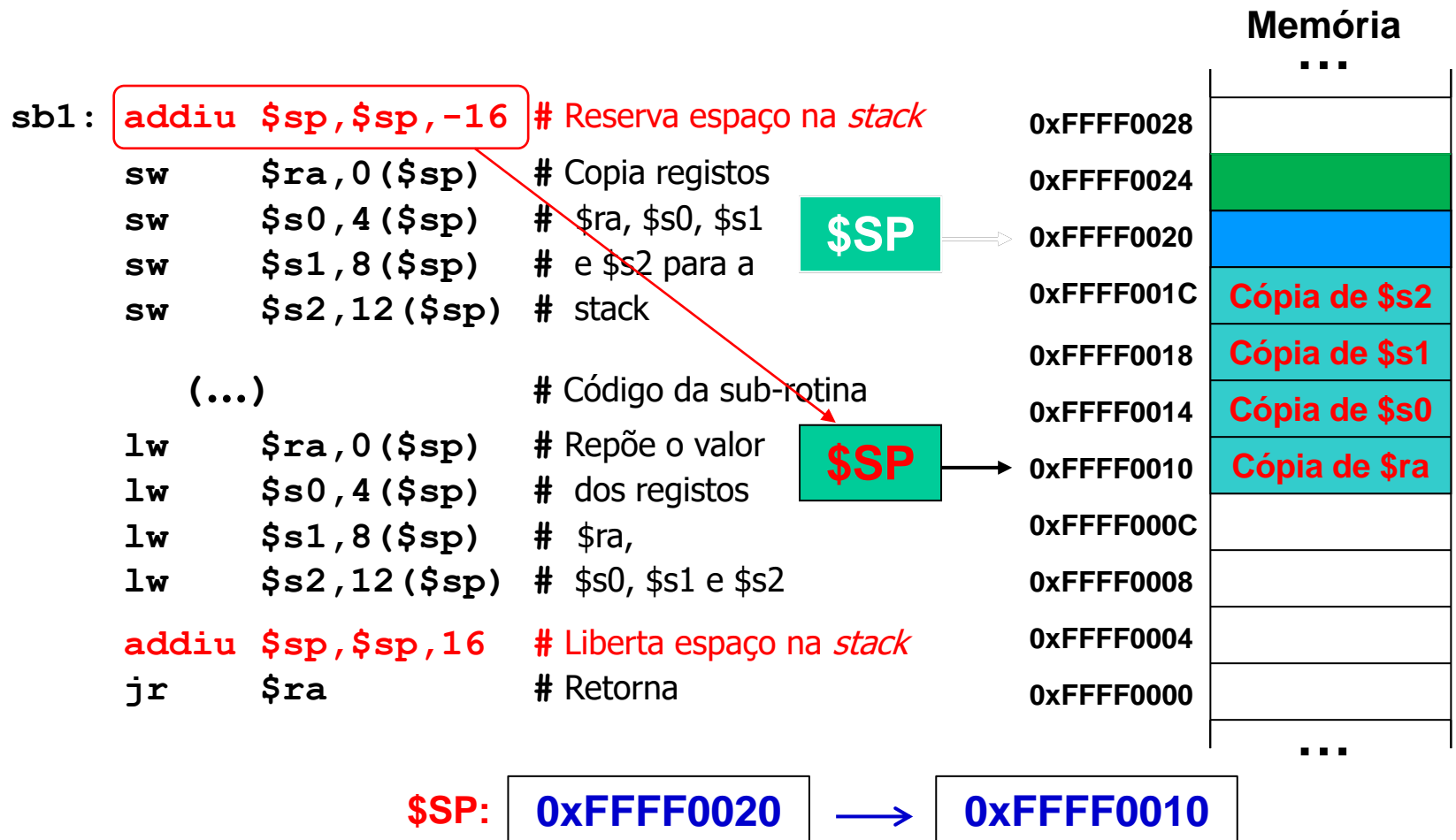
2. A *stack* **cresce** no **sentido decrescente** dos endereços da memória

3. A *stack* **está organizada em words de 32 bits**



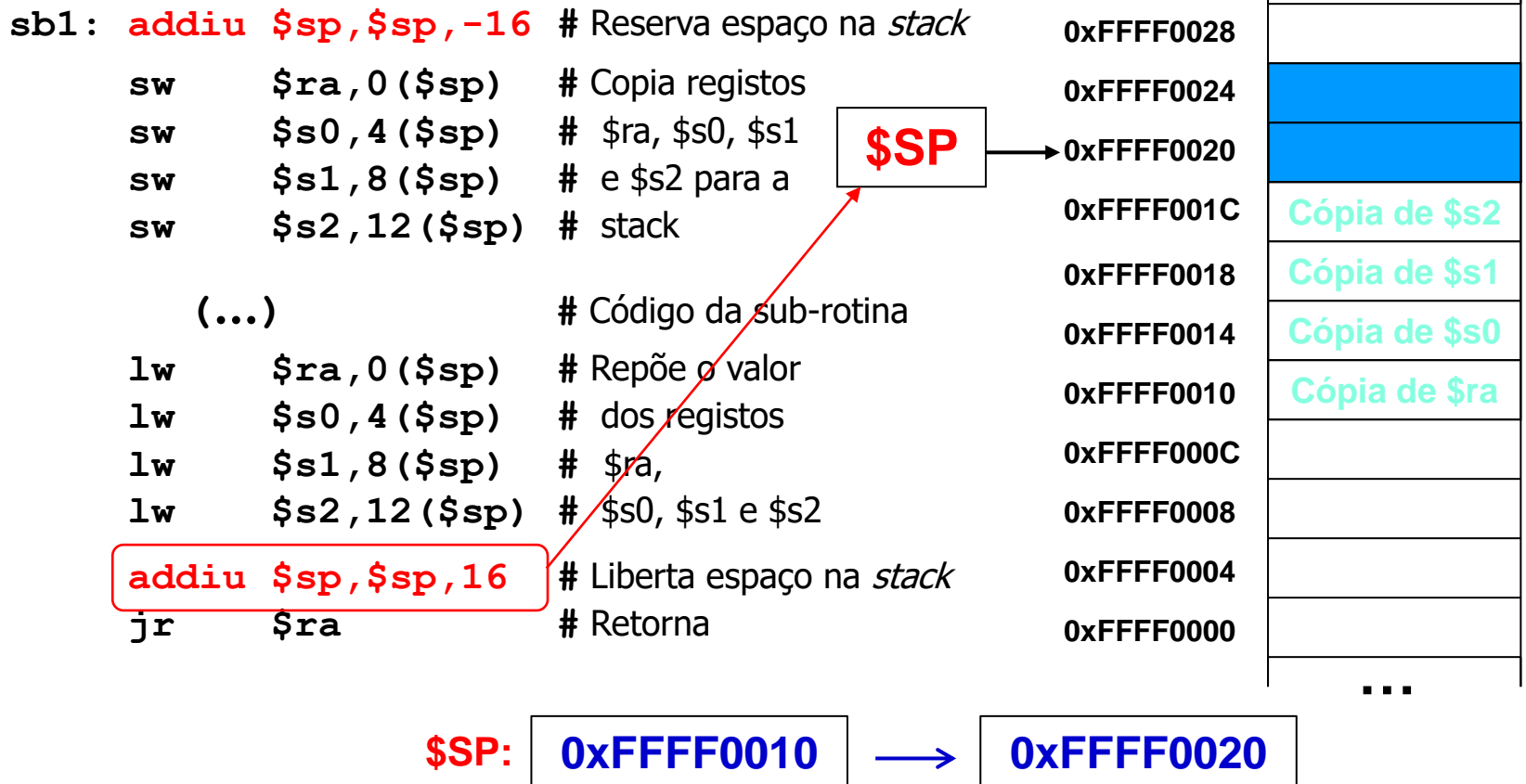
Regras de utilização da *stack* na arquitetura MIPS

- Exemplo (salvaguarda de 4 registos)



Regras de utilização da *stack* na arquitetura MIPS

- Exemplo (salvaguarda de 4 registos)



Regras de utilização da *stack* na arquitetura MIPS

- Exemplo

Prólogo

```
sb1: addiu $sp,$sp,-16 # Reserva espaço na stack
      sw    $ra,0($sp) # Copia registos
      sw    $s0,4($sp) # $ra, $s0, $s1
      sw    $s1,8($sp) # e $s2 para a
      sw    $s2,12($sp) # stack
```

\$SP

(...)

Código da sub-rotina

```
lw    $ra,0($sp) # Repõe o valor
lw    $s0,4($sp) # dos registos
lw    $s1,8($sp) # $ra,
lw    $s2,12($sp) # $s0, $s1 e $s2
addiu $sp,$sp,16 # Liberta espaço na stack
jr    $ra        # Retorna
```

Epílogo

Memória

0xFFFF0028

0xFFFF0024

0xFFFF0020

0xFFFF001C

0xFFFF0018

0xFFFF0014

0xFFFF0010

0xFFFF000C

0xFFFF0008

0xFFFF0004

0xFFFF0000

Cópia de \$s2

Cópia de \$s1

Cópia de \$s0

Cópia de \$ra

Análise de um exemplo completo

Considere-se o seguinte código C:

```
int soma(int *, int);
```

```
void main(void)
{
```

```
    static int temp[100]; // reside em memória
```

```
    int result;
```

```
    ...                // código de inicialização do array
```

```
    result = soma(temp, 100);
```

```
    print_int10(result); // syscall
```

```
}
```

Declaração de um *array static*
(reside no “data segment”)

Declaração de uma variável
inteira (pode residir num registo
interno)

Afixação do resultado
no ecrã

Chamada de uma função e
atribuição do valor devolvido à
variável inteira

Análise de um exemplo completo

```
int soma(int *, int);

void main(void)
{
    static int temp[100]; //reside em memória
    int result;
    ...                // código de inicialização do array
    result = soma(temp, 100);
    print_int10(result); // syscall
}
```

- A função `main()` é uma função intermédia (chama a função `soma()`); registo `$ra` tem que ser salvaguardado
- A variável "`result`" não tem atribuído qualquer valor que seja necessário depois da chamada à função `soma()`; deve residir num registo `$tn`, `$vn` ou `$an` (pode ser usado `$t0`)

Código correspondente em *Assembly* do MIPS

```
# result: $t0
#
```

```
        .data
temp:    .space 400

        .eqv    print_int10, 1
        .text
        .globl  main
main:    addiu   $sp, $sp, -4
        sw      $ra, 0($sp)
        la      $a0, temp
        li      $a1, 100
        jal     soma
        move    $t0, $v0
        move    $a0, $t0
        li      $v0, print_int10
        syscall
        lw      $ra, 0($sp)
        addiu   $sp, $sp, 4
        jr      $ra
```

```
void main(void) {
    static int temp[100];
    int result;
    result = soma(temp, 100);
    print_int(result);
}
```

```
# Reserva de espaço p/ o array
# (100 words => 400 bytes)
#
# Reserva espaço na stack
# Salva o registo $ra
# inicialização dos registos
# que vão passar os parâmetros
# soma(temp, 100)
# result = soma(temp, 100)
#
# print_int(result)
# Recupera o valor do reg. $ra
# Liberta espaço na stack
# Retorno
```

Código da função soma()

```
int  soma (int *array, int n)
{
    int i, res;
    for (i = 0, res = 0; i < n; i++)
    {
        res = res + array[i];
    }
    return res;
}
```

Esta função recebe dois parâmetros (um ponteiro para inteiro e um inteiro) e calcula o seguinte resultado:

$$\text{res} = \sum_{i=0}^{n-1} (\text{array}[i])$$

A mesma função usando ponteiros:

```
int  soma (int *array, int n)
{
    int res = 0;
    int *p = array;
    for (; p < &(array[n]); p++) // ou: ; p < (array + n);
    {
        res += (*p);
    }
    return res;
}
```

Código de uma função para cálculo da média

```
int media (int *array, int n)
{
    int res;

    res = soma(array, n);
    return res / n;
}
```

- A função `media()` é uma **função intermédia**:
 - é necessário **salvar \$ra**
- O valor da variável "`res`" só é definido após a chamada à função; deve residir num registo de utilização livre, por exemplo **\$t0**
- O número de elementos do `array` "`n`" (**\$a1**), é necessário após a chamada à função `soma()`; **o registo \$a1 tem que ser copiado para um registo \$sn** (por exemplo **\$s1**)

Exemplo – função para cálculo da média

```
int media (int *array, int n)
{
    int res;
    res = soma(array, n);
    return res / n;
}
```

chama função soma()

Valor de *n* é necessário depois de chamada a função “soma”!

```
# res: $t0, array: $a0, n: $a1 -> $s1
```

```
media:  addiu  $sp,$sp,-8      # Reserva espaço na stack
        sw     $ra,0($sp)     # salvaguarda $ra e $s1
        sw     $s1,4($sp)     # guarda valor $s1 antes de o usar
        move   $s1,$a1        # "n" é necessário depois
                                # da chamada à função soma
        jal     soma          # soma(array,n);
        move   $t0,$v0        # res = soma(array,n);
        div    $v0,$t0,$s1    # res/n
        lw     $ra,0($sp)     # recupera valor de $ra
        lw     $s1,4($sp)     # e $s1
        addiu  $sp,$sp,8      # Liberta espaço na stack
        jr     $ra            # retorna
```