

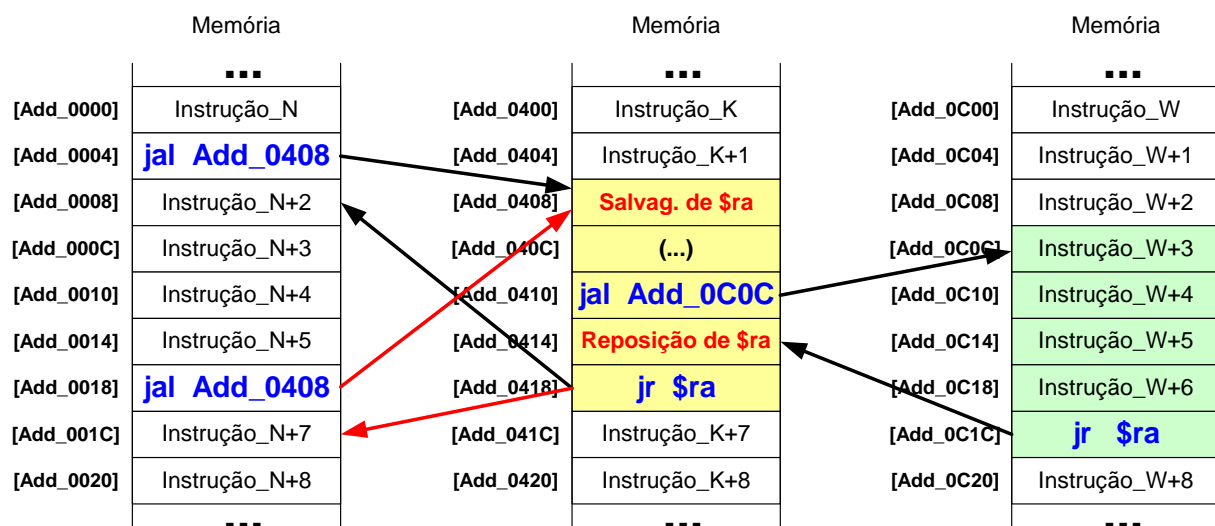
## Aula 9

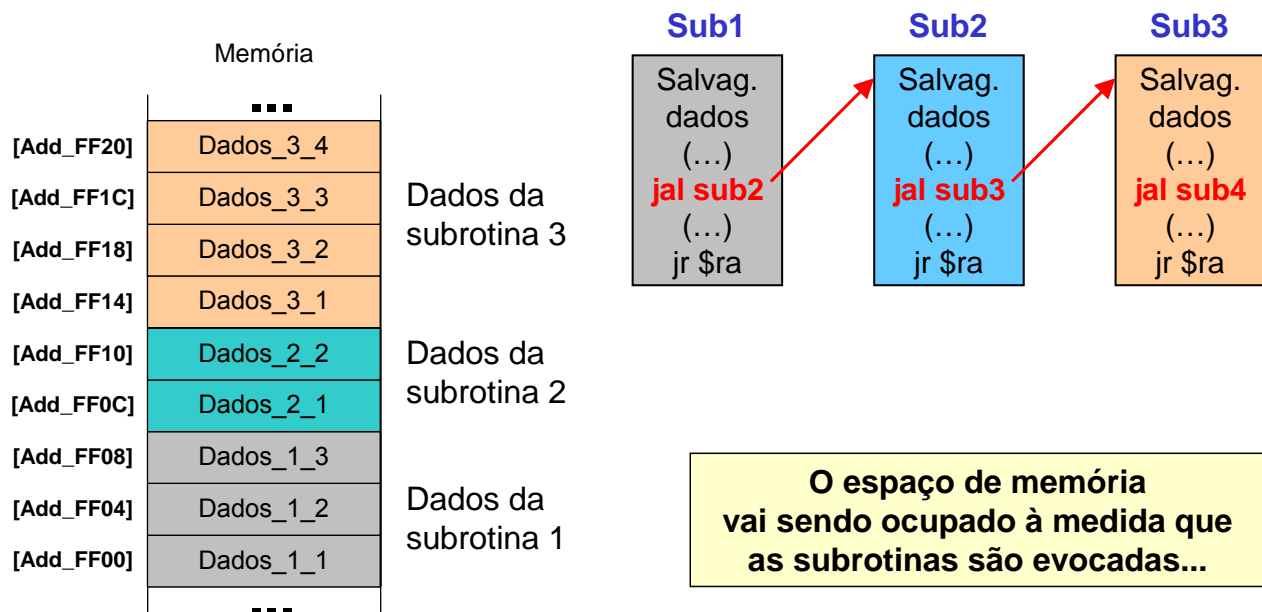
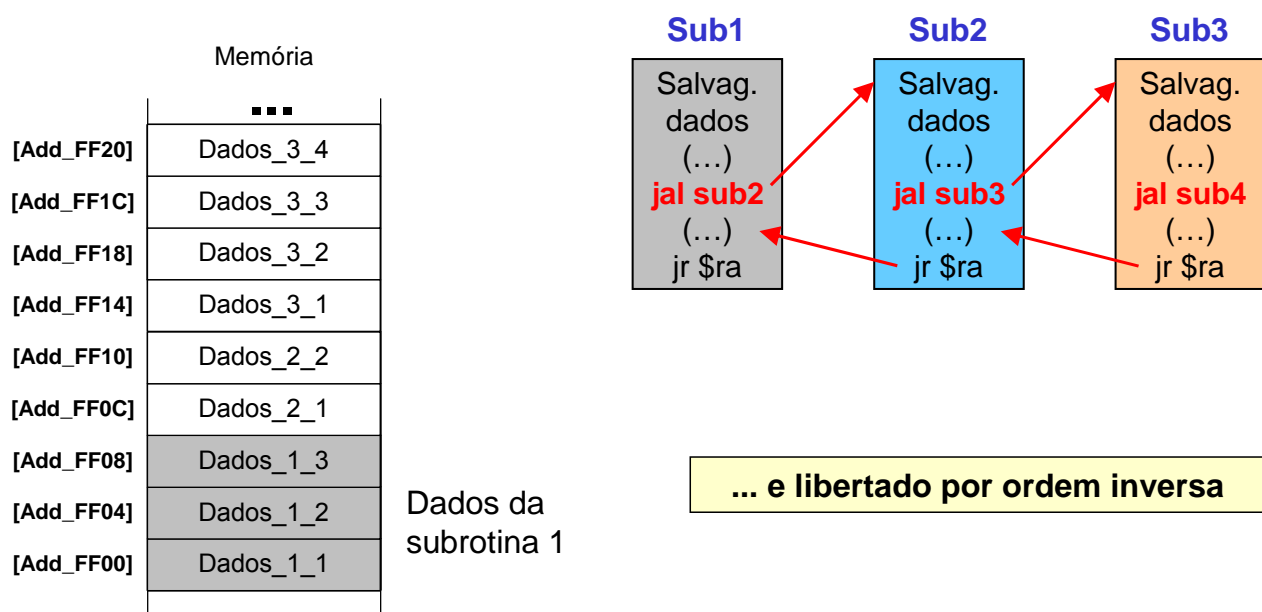
- Utilização de *stacks*
  - Conceito e regras básicas de utilização
  - Utilização da *stack* nas arquitecturas MIPS
- Recursividade
- Análise de um exemplo, incluindo uma subrotina recursiva

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira, Tomás Oliveira e Silva

Relembremos um slide da aula 8...

Como poderemos garantir que os dados, residentes em memória e manipulados por cada subrotina não interferem com os dados das restantes?



**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 normalmente designada por **LIFO** (Last In First Out).
- A estrutura de dados correspondente é conhecida por “pilha” - **STACK**
- As *stacks* são de tal forma importantes que a maioria das arquitecturas suportam directamente instruções específicas para manipulação de *stacks*.
- A operação que permite acrescentar informação à *stack* é normalmente designada por **PUSH**, enquanto que a operação inversa é conhecida por **POP**.
- Estas operações têm normalmente associado um registo designado por **Stack Pointer**. Este registo mantém, de forma permanente, o **endereço do topo da pilha** (*top of stack*).

**Stack: espaço de armazenamento temporário**

Uma operação de **PUSH** pode seguir uma de duas estratégias:

- Se o *stack pointer* aponta para o **último endereço ocupado**, então é necessário pré-actualizar esse registo antes de escrever na *stack*
- Alternativamente, o *stack pointer* pode apontar para o **primeiro endereço livre** acima do topo da pilha. Nesse caso, a informação é adicionada à *stack*, seguindo-se uma pós-actualização do *stack-pointer*

Uma operação de **POP** terá, necessariamente, que acompanhar a estratégia escolhida para o **PUSH**, funcionando de forma simétrica:

- Leitura da *stack* seguida de actualização do *stack pointer* para a primeira estratégia
- Pré-actualização do *stack pointer* seguida de leitura no caso da segunda

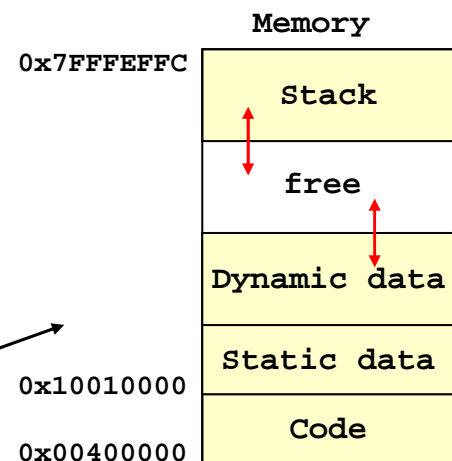
## Stack: espaço de armazenamento temporário

A actualização do *stack pointer*, durante a fase de escrita de informação, pode também 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 segunda estratégia, de **crescimento da stack por ordem decrescente de endereços**, é geralmente a adoptada, por permitir uma **gestão simplificada** da fronteira entre os segmentos de dados e de *stack*

Exemplo do mapa de memória do MARS

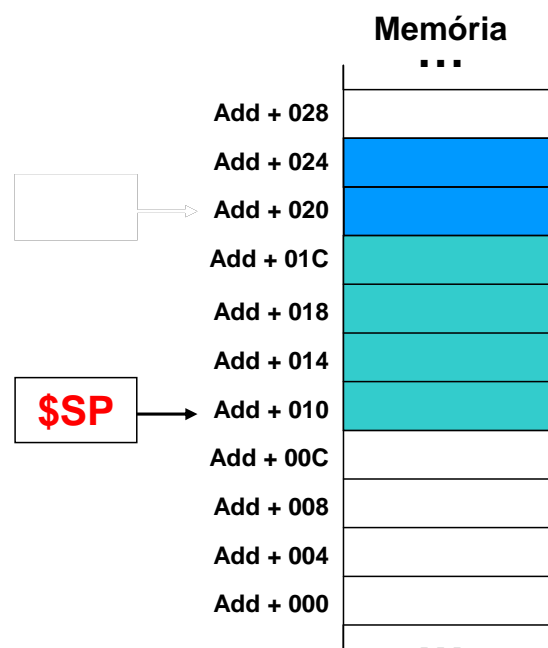


## Regras de utilização da stack na arquitectura MIPS

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

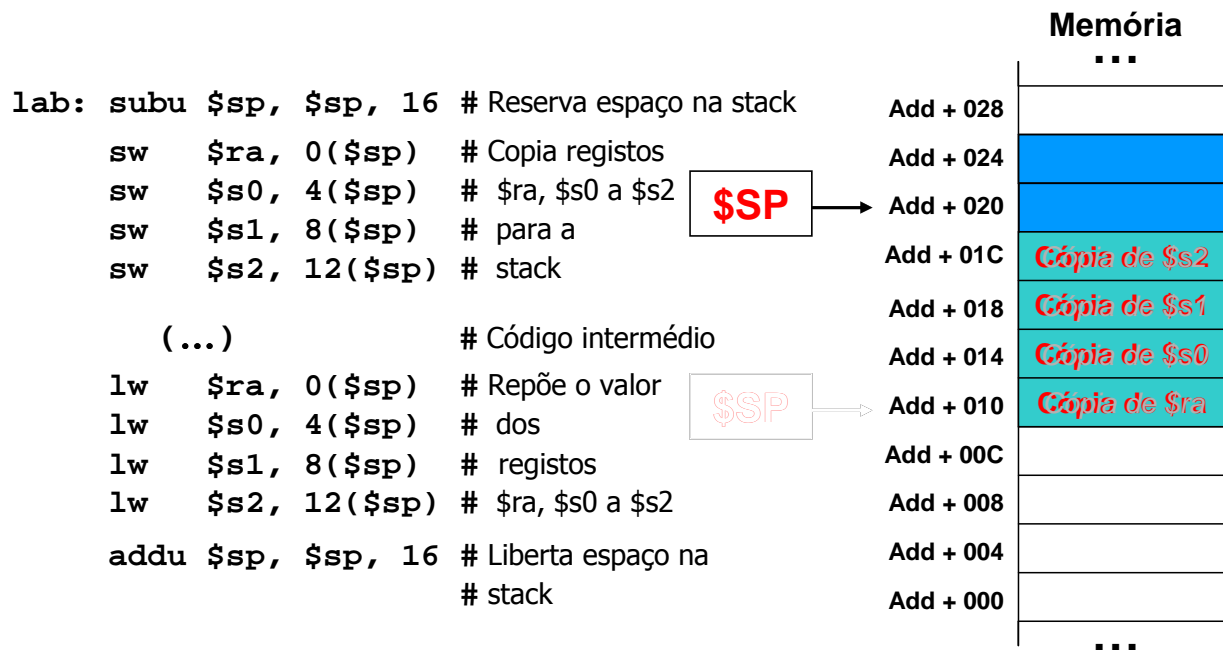
2. A *stack* **cresce** no **sentido decrescente** dos endereços da memória

**\$sp = \$29**



## Regras de utilização da *stack* na arquitectura MIPS

### Exemplo:



## Análise de um exemplo completo

Consideremos o código seguinte:

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

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

Evocação de uma subrotina e  
atribuição do valor devolvido à  
variável inteira

## Código correspondente em *Assembly* do MIPS:

```
# $t0 -> variável "result"
#
```

```
        .data
array:  .space 400           # Reserva de espaço p/ o array
                                # (100 words => 400 bytes)

        .text
main:    subu    $sp, $sp, 4   # Reserva espaço na stack
        sw      $ra, 0($sp)   # Salvaguarda o registo $ra
        ...
        la      $a0, array    # inicialização dos registos
        li      $a1, 100      # que vão passar os parâmetros
        jal     soma          # soma(array, 100)
        move    $t0, $v0      # result = soma(array, 100)
        move    $a0, $t0      #
        li      $v0, 1        #
        syscall                     # print_int(result)
        lw      $ra, 0($sp)    # Recupera o valor do reg. $ra
        addu    $sp, $sp, 4    # Liberta espaço na stack
        jr      $ra           # Retorno
```

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

## Olhemos agora para a função:

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

## Ou, alternativamente, com ponteiros:

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

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

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

**Código correspondente em *Assembly* do MIPS:**

(versão com ponteiros)

```
# $t1 armazena p
# $v0 armazena res
#
```

```
soma:  li    $v0, 0           # res = 0;
       move  $t1, $a0        # p = array;
       sll   $a1, $a1, 2      # nelem *= 4;
       addu  $a0, $a0, $a1    # $a0 = array + nelem;
for:    bgeu  $t1, $a0, endf   # while(p < &(array[nelem])){
       lw    $t2, 0($t1)      #
       add   $v0, $v0, $t2     # res = res + (*p);
       addiu $t1, $t1, 4      # p++;
       j     for              # }
endf:   jr    $ra             # return res;
```

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

A subrotina não evoca nenhuma outra e não são usados registos \$Sn, pelo que não é necessário salvar qualquer registo

**Olhemos agora para outra função:**

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

chama função soma

Valor de *nelem* é necessário depois de chamada a função "soma"!

**Código correspondente em *Assembly* do MIPS:**

```
# res->$t0, array->$a0, nelem->$a1
media: subu   $sp,$sp,8      # Reserva espaço na stack
       sw     $ra,0($sp)     # salva $ra e $s0
       sw     $s0,4($sp)     # guarda valor $s0 antes de usar $s0
       move   $s0,$a1        # nelem é necessário depois
                               # da chamada à função soma
       jal    soma           # soma(array,nelem);
       move   $t0,$v0        # res = retorno de soma
       div    $v0,$t0,$s0    # res/nelem
       lw     $ra,0($sp)     # recupera valor de $ra
       lw     $s0,4($sp)     # e $s0
       addu   $sp,$sp,8      # Liberta espaço na stack
       jr     $ra            # retorna
```

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

O resultado do somatório pode também ser obtido da seguinte forma:

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

$$\text{array}[1] + \sum_{n=2}^{\text{nelem}-1} (\text{array}[n])$$

$$\text{array}[2] + \sum_{n=3}^{\text{nelem}-1} (\text{array}[n])$$

(...)

$$\text{array}[\text{nelem} - 1]$$

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

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

**res = soma (array, 0, nelem);**

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

**array[0] + soma (array, 1, nelem);**

$$\text{array}[1] + \sum_{n=2}^{\text{nelem}-1} (\text{array}[n])$$

**array[1] + soma (array, 2, nelem);**

**int soma(int \*array, int i, int nelem);**

```
int soma(int *array, int i, int nelem)
{
    return array[i] + soma(array, i+1, nelem);
}
```

**O que falta nesta função?**

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



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

A função **soma()** pode, assim, ser escrita de forma **recursiva**:

O valor devolvido é posteriormente adicionado com o valor armazenado na posição **i** do array

```
int soma_rec (int *array, int i, int nelem)
{
    if (i != nelem) {
        return array[i] + soma_rec (array, i + 1, nelem);
    } else
        return 0;
}
```

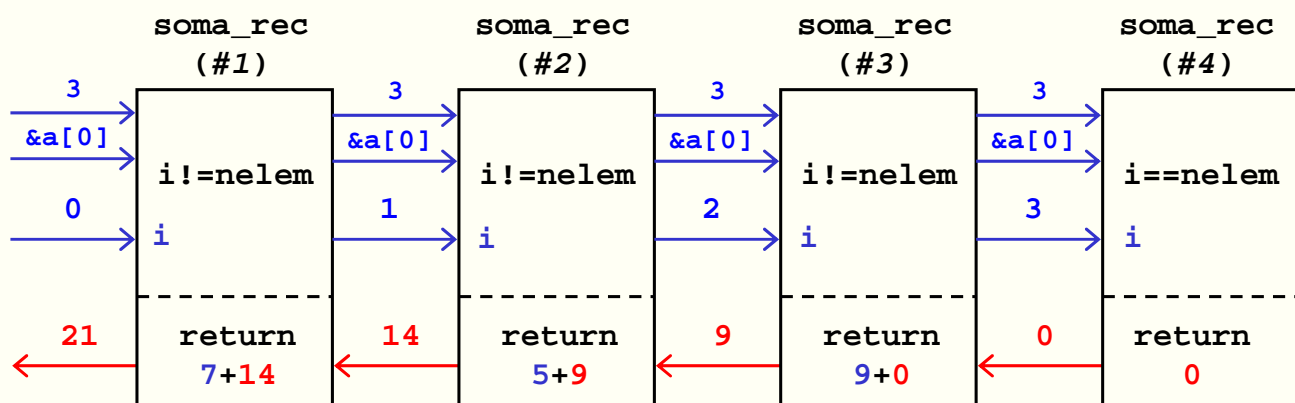
A **subrotina evoca-se a si mesma**, passando como primeiro parâmetro o endereço do início do *array*, como segundo parâmetro o elemento a partir do qual se pretende obter a soma e como terceiro parâmetro o número de elementos do *array*

```
int soma_rec (int *array, int i, int nelem)
{
    if (i != nelem) {
        return array[i] + soma_rec (array, i + 1, nelem);
    } else
        return 0;
}
```

**Exemplo:**

Nº elementos do array "a": 3

Array inicializado com: **a[0]=7, a[1]=5, a[2]=9**



```
int soma_rec (int *array, int i, int nelem)
{
    if (i != nelem) {
        return array[i] + soma_rec (array, i + 1, nelem);
    } else
        return 0;
}
```

A função `soma_rec()` pode ser simplificada, utilizando um **ponteiro para a posição do array a partir da qual se pretende obter a soma** (em vez do índice) e o **número de elementos do array que falta processar** (em vez do número total de elementos).

```
int soma_rec (int *array, int nelem)
{
    if (nelem != 0) {
        return *array + soma_rec (array + 1, nelem - 1);
    } else
        return 0;
}
```

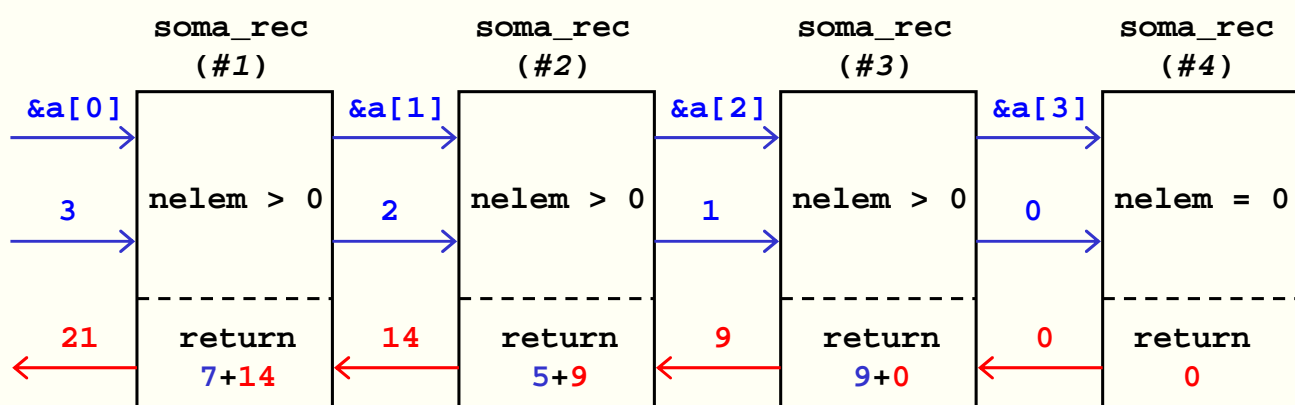
O segundo parâmetro representa o **número de elementos do array ainda não processados**

```
int soma_rec (int *array, int nelem)
{
    if (nelem != 0) {
        return *array + soma_rec (array + 1, nelem - 1);
    } else
        return 0;
}
```

Exemplo:

Nº elementos do array "a": 3

Array inicializado com: `a[0]=7, a[1]=5, a[2]=9`



## Código correspondente em Assembly do MIPS:

```
int soma_rec (int *array, int nelem)
{
    if (nelem != 0) {
        return *array+soma_rec(array+1,nelem-1);
    } else
        return 0;
}
```

soma\_rec:

```
    beq    $a1, $0, else # if (nelem != 0) {
    subu    $sp, $sp, 8   # stack allocation
    sw      $ra, 0($sp)   # save $ra
    sw      $s0, 4($sp)   # save $s0
    move    $s0, $a0      # $s0 = array
    addiu   $a0, $a0, 4    # array + 1;
    sub     $a1, $a1, 1    # nelem=nelem-1;
    jal     soma_rec      # soma_rec(array+1, nelem-1);
    lw      $t0, 0($s0)   # aux = *array;
    add     $v0, $v0, $t0 # val = val + aux;
    lw      $ra, 0($sp)   # restore $ra
    lw      $s0, 4($sp)   # restore $s0
    addiu   $sp, $sp, 8   # free stack
    jr      $ra           # return val;
    # }
else:
    li      $v0, 0        #
    jr      $ra           # return 0;
    # }
```

← - Salvag. **\$ra** (a subrotina não é terminal)

- **array** é necessário depois da chamada à subrotina (copia-se para **\$s0**)

← O **stack pointer** tem obrigatoriamente que ser **atualizado antes de terminar a subrotina**