

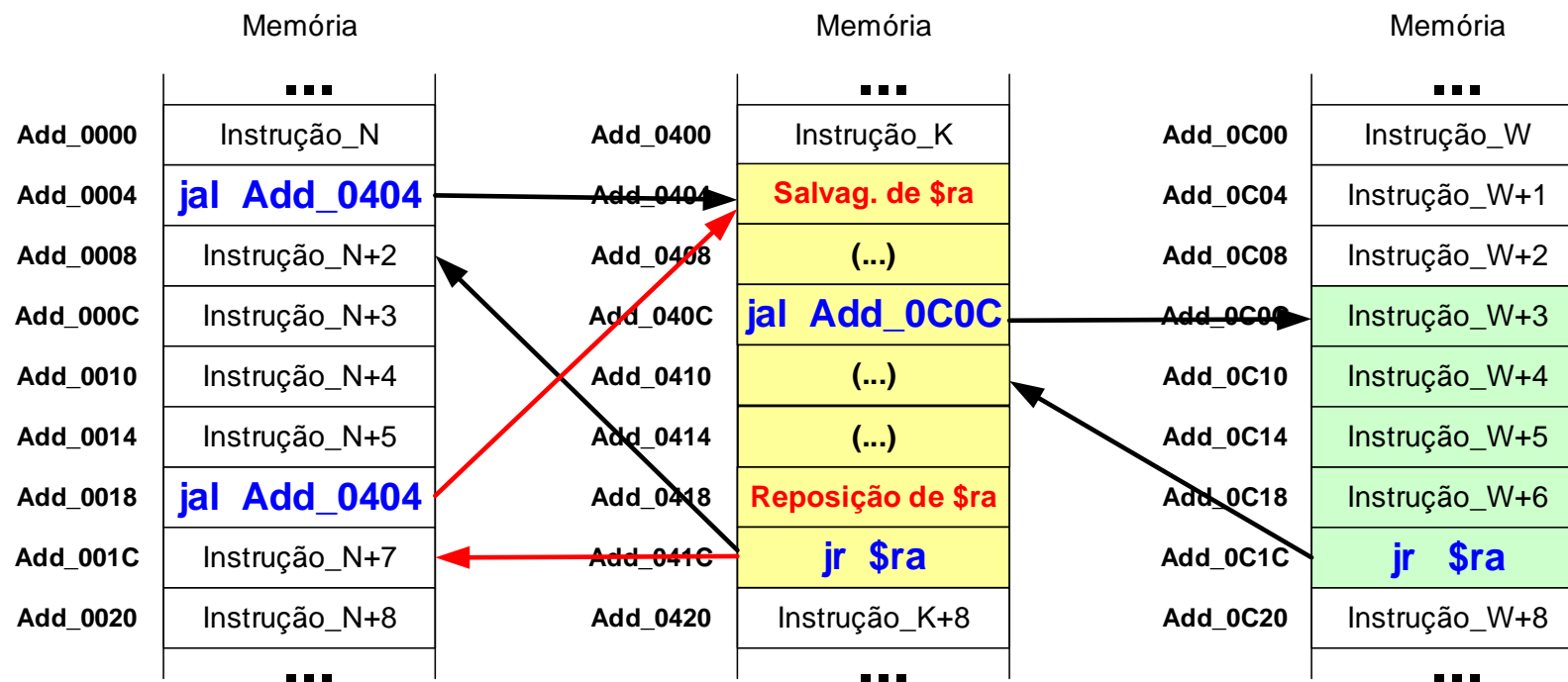
Aula 9

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

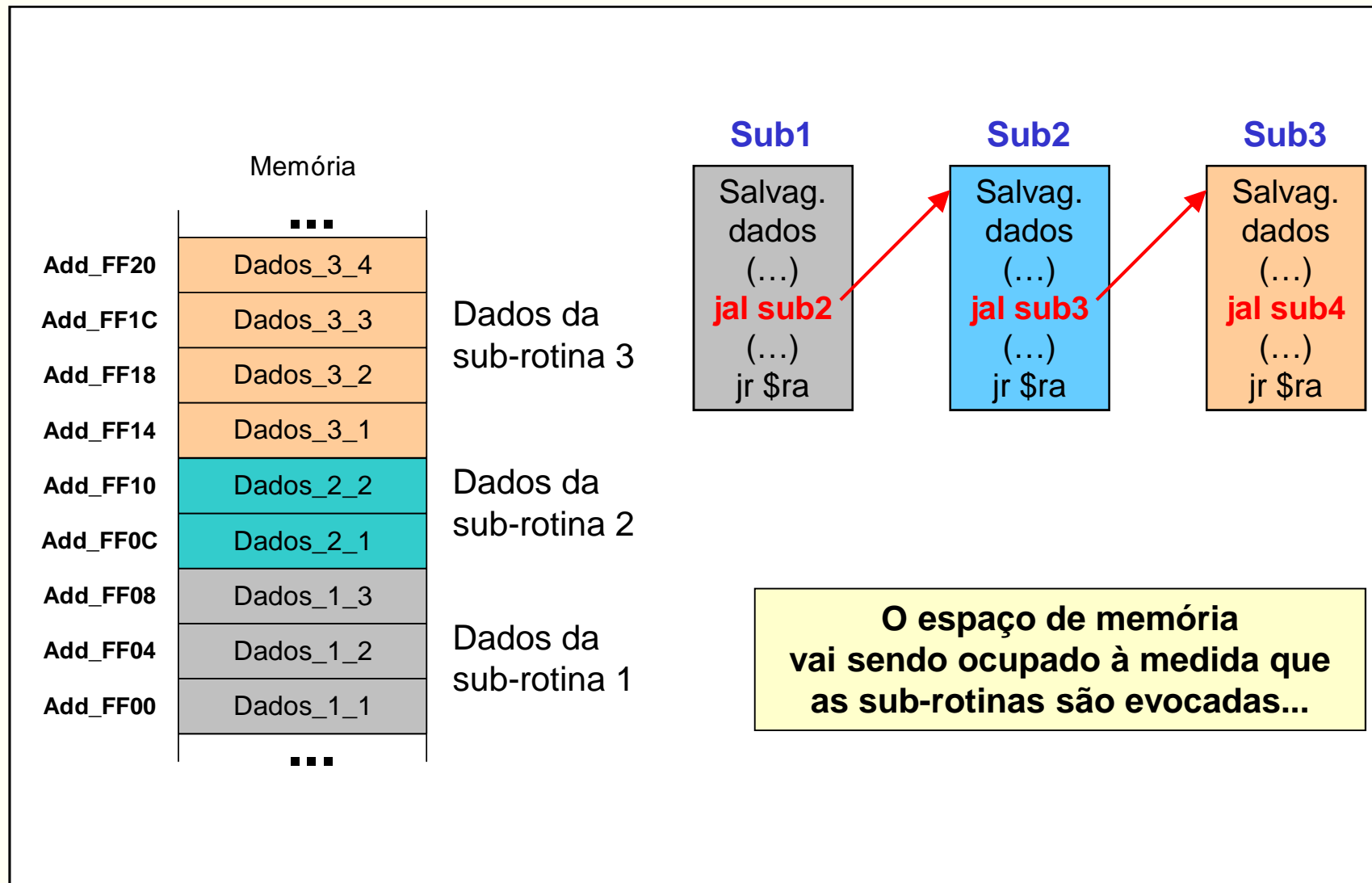
Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Armazenamento temporário de informação

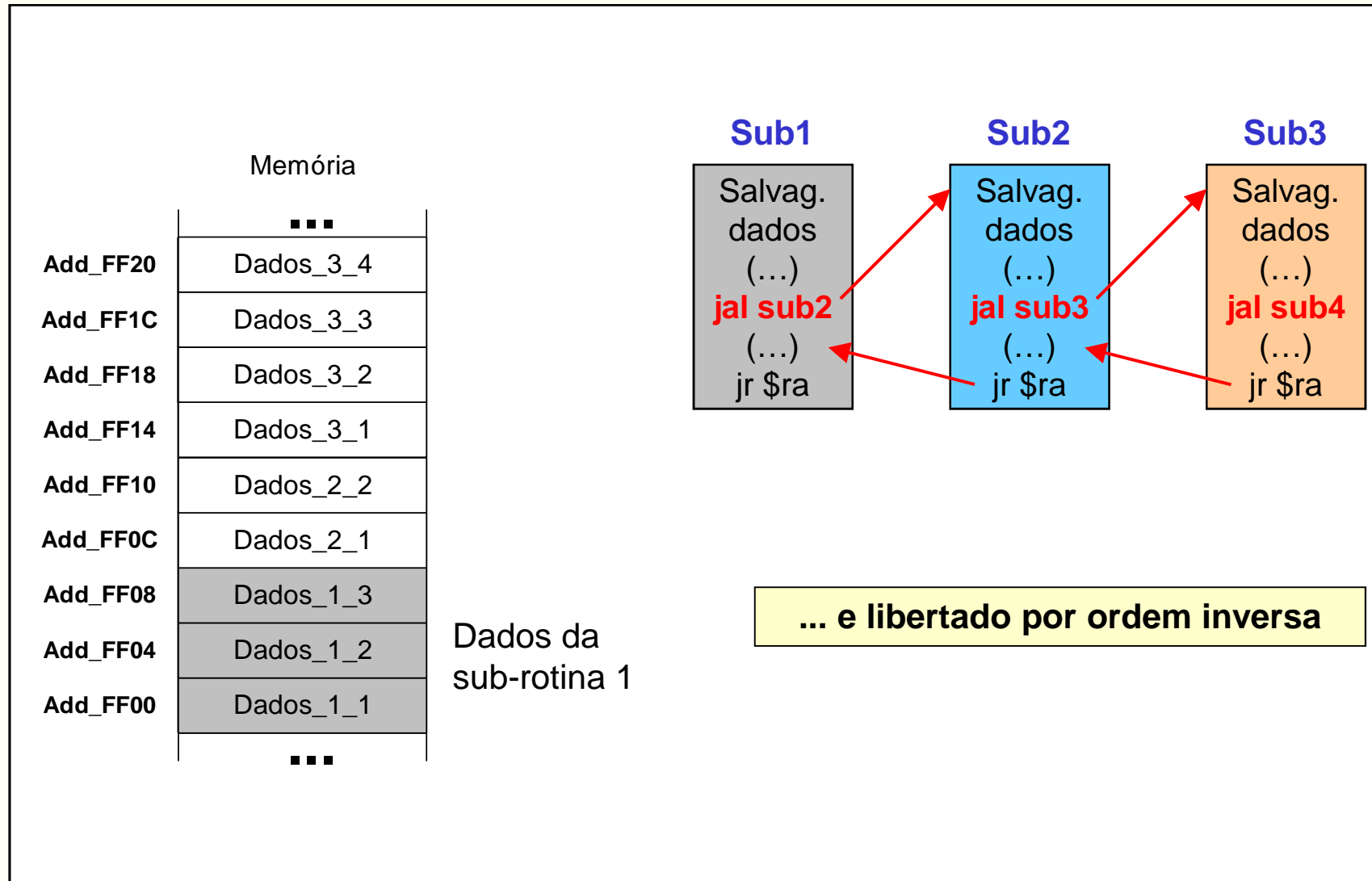
Como poderemos garantir que os dados, residentes em memória e manipulados por cada sub-rotina não interferem com os dados das restantes?



Stack: espaço de armazenamento temporário



Stack: espaço de armazenamento temporário

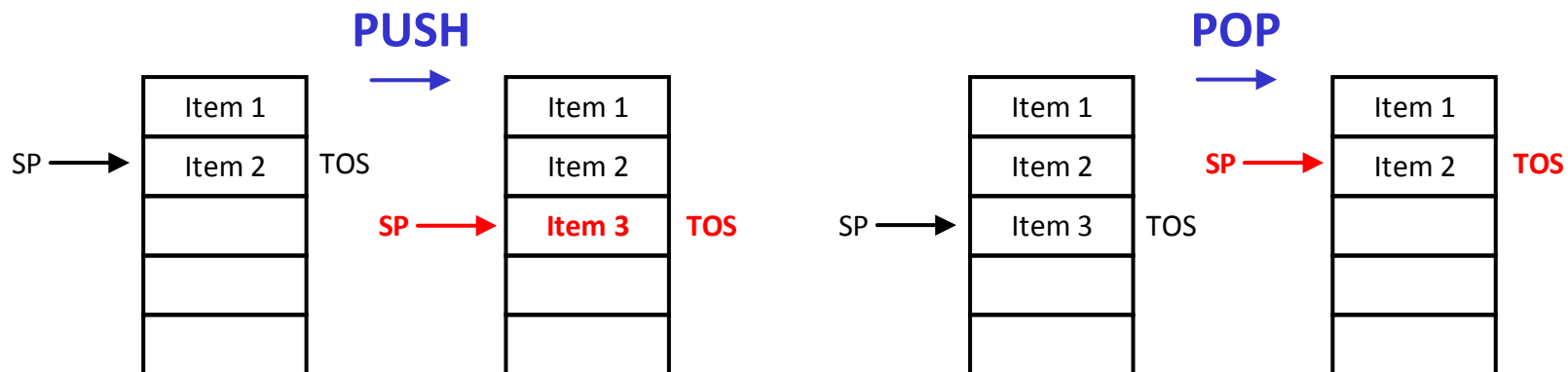


Stack: espaço de armazenamento temporário

- A estratégia de gestão dinâmica do espaço de memória - em que a última informação acrescentada é a primeira a ser retirada – é 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 arquiteturas suportam diretamente instruções específicas para manipulação de *stacks* (por exemplo a 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*

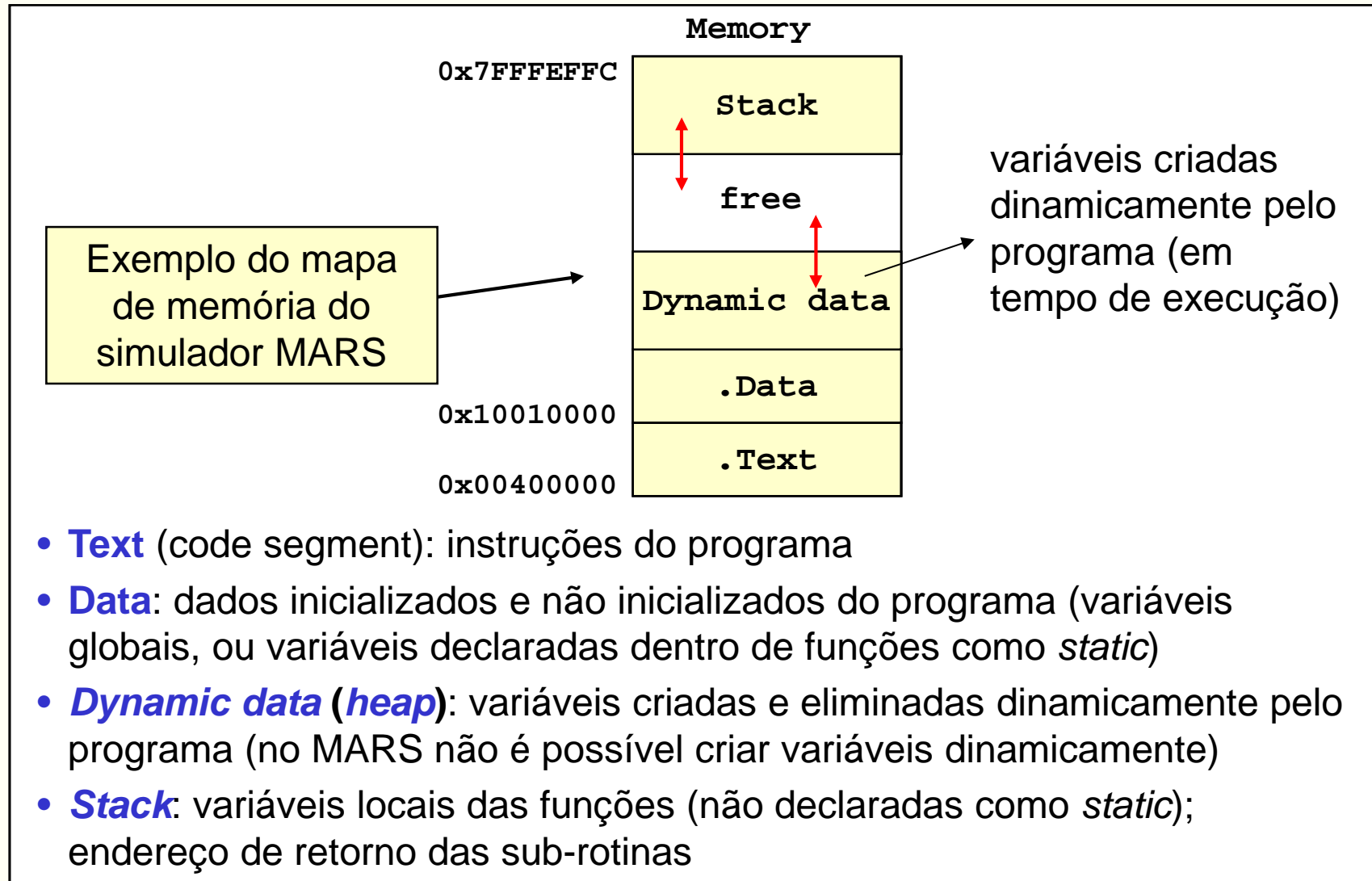
- Estas operações têm associado um registo designado por **Stack Pointer (SP)**
- O registo **Stack Pointer** mantém, de forma permanente, o **endereço do topo da stack (TOS - top of stack)** e aponta sempre **para o último endereço ocupado**
 - Numa operação de **PUSH** é necessário pré-atualizar o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



Atualização do *stack pointer*

- A atualização do *stack pointer*, durante a fase de 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*

Atualização do *stack pointer*



Regras de utilização da *stack* na arquitetura MIPS

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

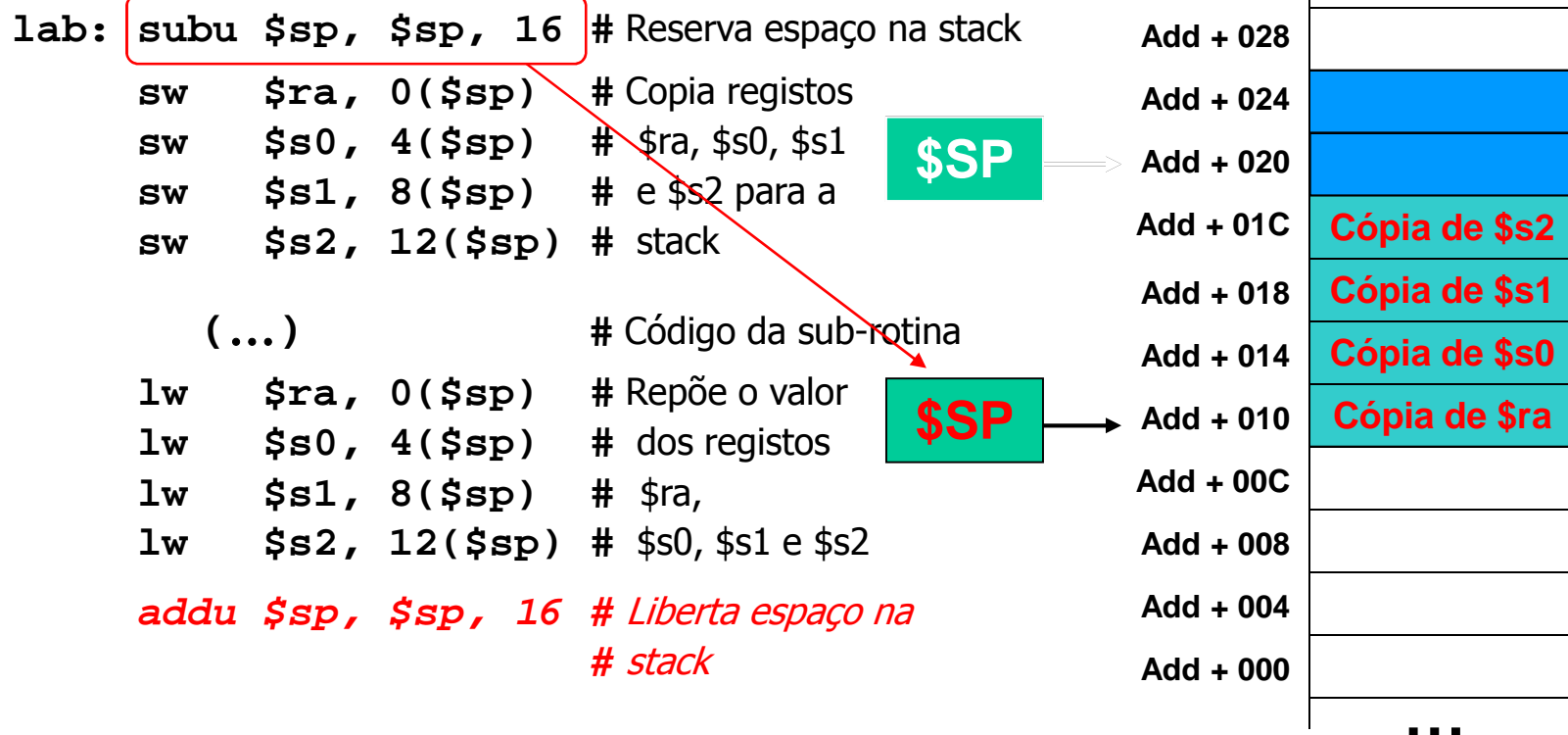
2. A *stack* **cresce** no **sentido decrescente** dos endereços da memória

\$sp = \$29



Regras de utilização da *stack* na arquitetura MIPS

• Exemplo



Regras de utilização da *stack* na arquitetura MIPS

- Exemplo

```
lab: subu $sp, $sp, 16 # Reserva espaço na stack
      sw   $ra, 0($sp) # Copia registros
      sw   $s0, 4($sp) # $ra, $s0, $s1
      sw   $s1, 8($sp) # e $s2 para a
      sw   $s2, 12($sp) # stack

      (...)           # Código da sub-rotina
      lw   $ra, 0($sp) # Repõe o valor
      lw   $s0, 4($sp) # dos registros
      lw   $s1, 8($sp) # $ra,
      lw   $s2, 12($sp) # $s0, $s1 e $s2
      addu $sp, $sp, 16 # Liberta espaço na
                        # stack
```

\$SP

Memória	
...	
Add + 028	
Add + 024	
Add + 020	
Add + 01C	Cópia de \$s2
Add + 018	Cópia de \$s1
Add + 014	Cópia de \$s0
Add + 010	Cópia de \$ra
Add + 00C	
Add + 008	
Add + 004	
Add + 000	
...	

Análise de um exemplo completo

Considere-se o seguinte código C:

```
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 função 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)  
        .eqv    print_int, 1 #  
        .text  
        .globl  main  
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, print_int  
        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);  
}
```

Código da função soma()

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

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

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

A mesma função usando ponteiros:

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

Código correspondente em *Assembly* do MIPS

- Versão com ponteiros

```
# $t1 > p
# $v0 > 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, endif  # while(p < &(array[nelem])){
       lw     $t2, 0($t1)      #
       add    $v0, $v0, $t2    #     res = res + (*p);
       addiu  $t1, $t1, 4      #     p++;
       j      for             # }
endif: 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 sub-rotina não evoca nenhuma outra e não são usados registos \$Sn, pelo que não é necessário salvar qualquer registo

Exemplo – função para cálculo da média

```
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”!

```
# res > $t0, array > $a0, nelem > $a1
media: subu    $sp,$sp,8      # Reserva espaço na stack
       sw     $ra,0($sp)     # salvaguarda $ra e $s0
       sw     $s0,4($sp)     # guarda valor $s0 antes de o usar
       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
```


Recursividade – função soma()

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

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

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

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

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

(...)

$$\underbrace{\hspace{10em}}_{\text{array}[\text{nelem} - 1]}$$

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

Recursividade – função soma()

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

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?

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

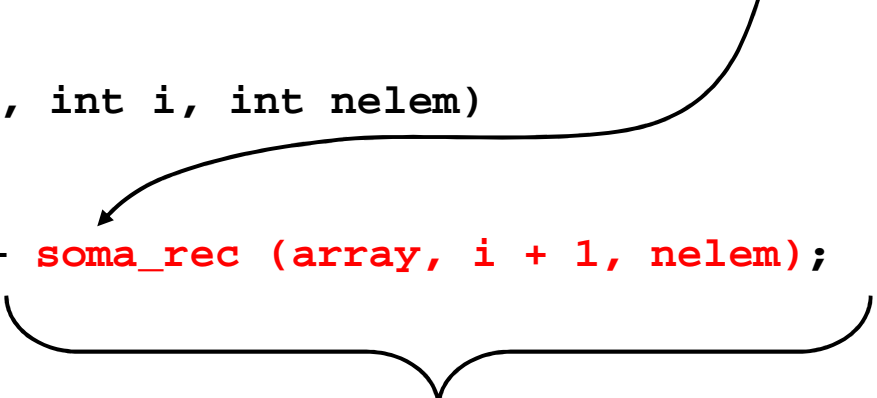
Recursividade – função soma()

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

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 função 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*

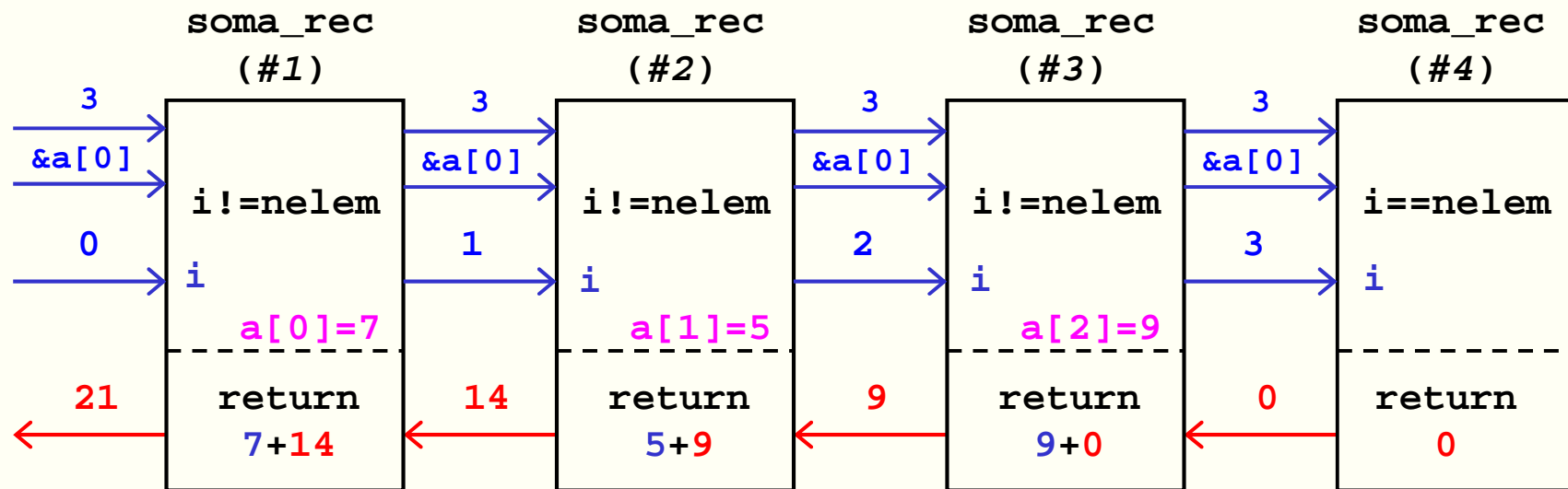
Recursividade – função soma()

```
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: $a[0]=7$, $a[1]=5$, $a[2]=9$



Recursividade – função soma()

```
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 visitar** (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 visitados**

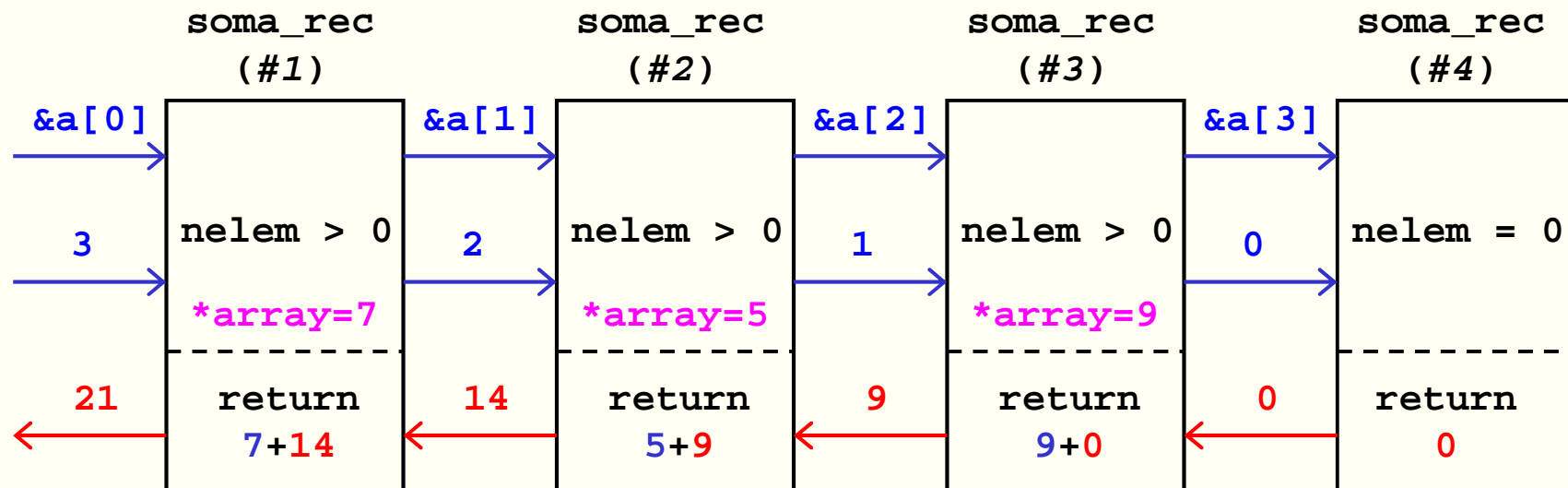
Recursividade – função soma()

```
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
        addu     $sp, $sp, 8      # free stack
        jr       $ra            # return val;
                                # }
else:
        li       $v0, 0          #
        jr       $ra            # return 0; }
```

Salvag. **\$ra** (a sub-rotina não é terminal)
array é necessário depois da chamada à sub-rotina (copia para **\$s0**)

O **stack pointer** tem obrigatoriamente que ser **atualizado antes de terminar a sub-rotina**