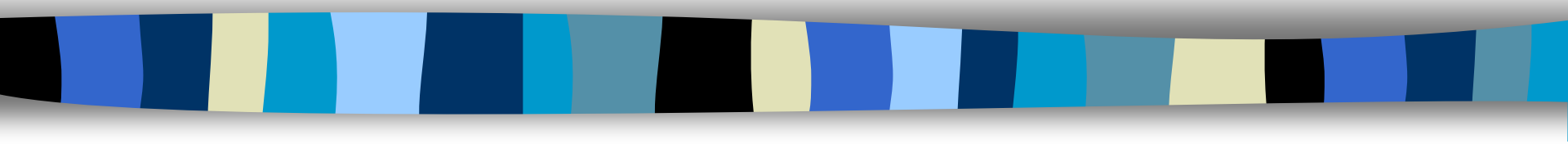
Sistemas da Computação



Linguagem Assembly do MIPS - IV

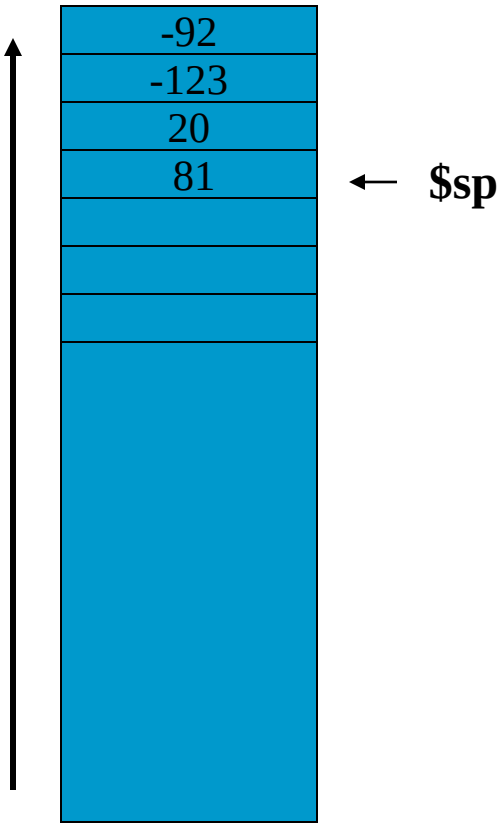
Utilização da Stack e Procedimentos

Stack

- A stack é uma forma de organizar a informação em memória
- O seu modo de funcionamento é designado por LIFO (*last in, first out*)
- No MIPS existe um registo ($sp ou $29) que contém sempre o endereço do topo da stack. É através deste registo que a stack é controlada.

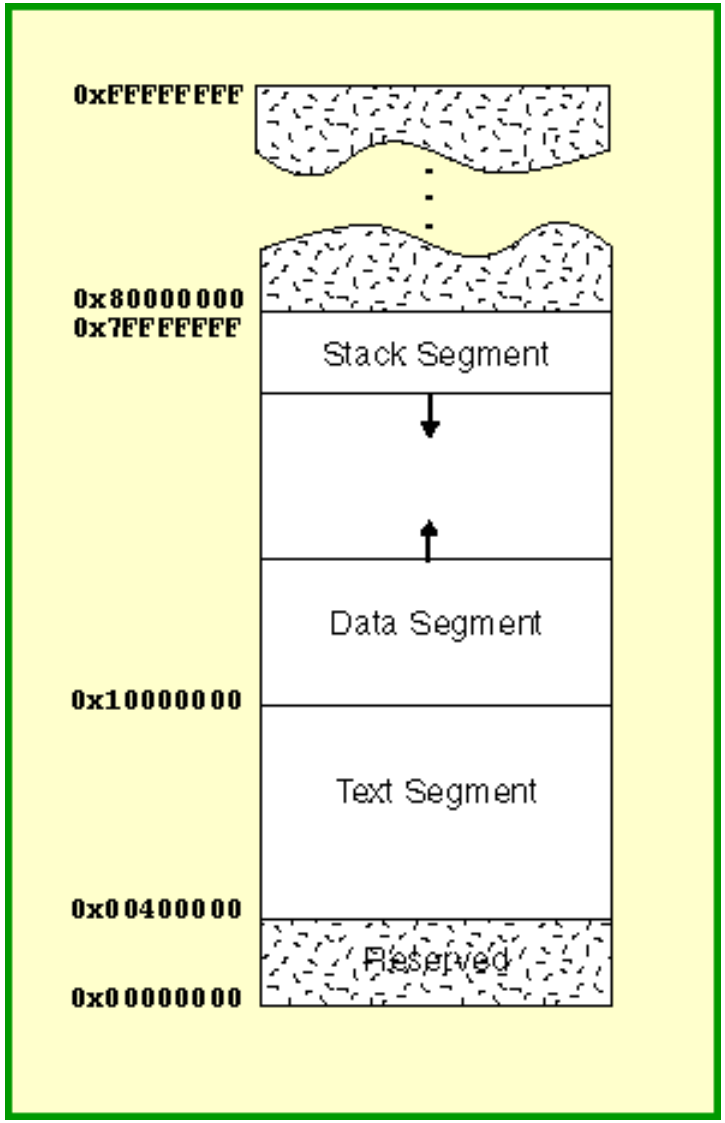
Stack

- A stack cresce no sentido contrário (o proximo elemento é inserido no endereço mais baixo)
- Para colocar um elemento na stack (push):
  - subu $sp,$sp,4 (subtração s/ sinal)
  - sw $t0, ($sp)
- Para retirar um elemento da stack (pop):
  - lw $t0, ($sp)
  - addiu $sp, $sp, 4



Stack

- No Mips existe a capacidade de endereçar 4 Gbytes de memória (endereços de 32 bits), sendo aproximadamente 1.8 Gbytes destinados a dados)
- Habitualmente o data segment e o stack segment concorrem pelo mesmo espaço crescendo em sentidos opostos (maior flexibilidade)



Stack

- Exercícios
  - Implemente em Linguagem Assembly do MIPS um programa que permita inverter a ordem dos caracteres de uma string, utilizando para esse efeito a *stack*.
    - Numa primeira fase considere a que a string está declarada em memória.
    - Numa segunda fase altere o programa para que passe a receber a string como input e mostre a string invertida no écran.

Programa em Assembly

```
        .data
str:    .asciiz "hello world"
        .text
main:                           # execution starts here
                la $t1,str      # a0 points to the string
nextCh:         lb $t0,($t1)    # get a byte from string
                beqz $t0,strEnd # zero means end of string
                sub $sp,$sp,4   # adjust stack pointer
                sw $t0,($sp)    # PUSH the t0 register
                addi $t1,$t1,1  # move pointer one character
                j nextCh        # go round the loop again
strEnd:         la $t1,str      # a0 points to the string
store:          lb $t0,($t1)    # get a byte from string
                beqz $t0,done   # zero means end of string
                lw $t0,($sp)    # POP a value from the stack
                addi $sp,$sp,4  # and adjust the pointer
                sb $t0,($t1)    # store in string
                addi $t1,$t1,1  # move pointer one character
                j store
done:           li $v0,10
                syscall         # au revoir...
```

Procedimentos no MIPS

O que é que constitui um procedimento/função?

• É um conjunto de instruções que podem ser usadas repetidamente ao longo de um programa

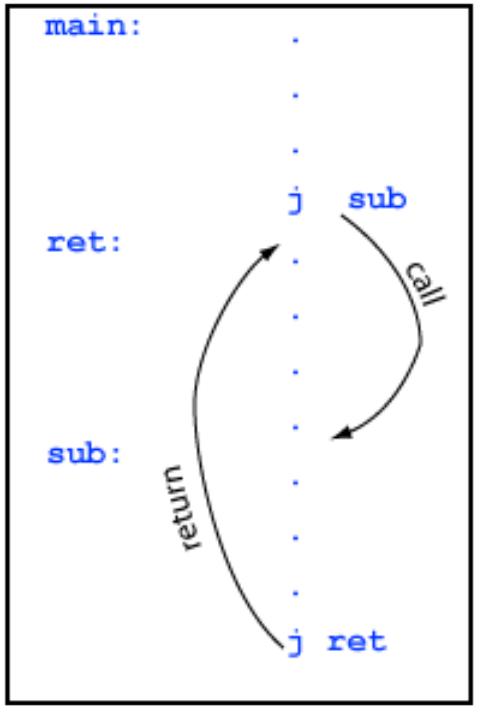
• É constituído por parâmetros, instruções e valor de retorno (ou resultado)

Que vantagens se retiram do uso de procedimentos?

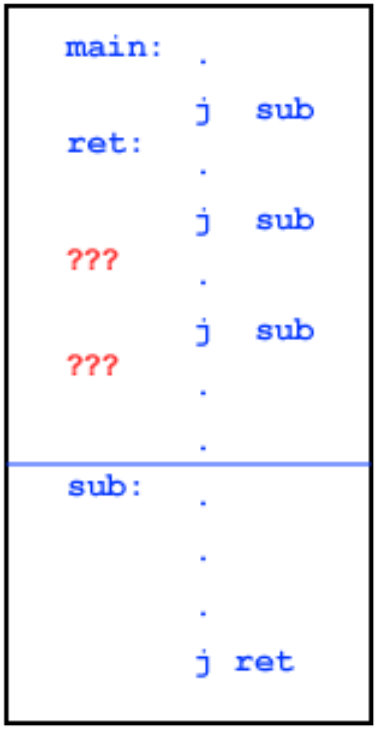
• Maior clareza do programa

• Reutilização de código

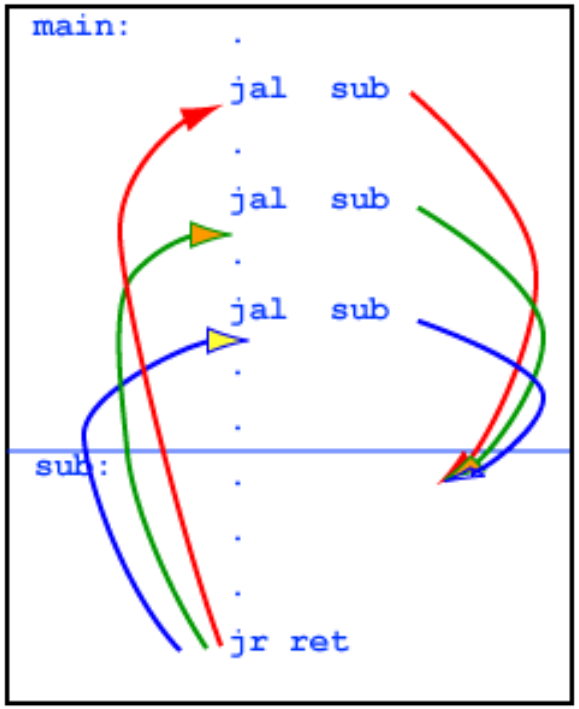
Procedimentos no MIPS



Subroutine Called Once



Subroutine Called Three Times



Correct Subroutine Linkage

Procedimentos no MIPS

Passos a seguir para a utilização de procedimentos num programa:

1. Colocar parâmetros num local onde o procedimento lhes possa aceder
2. Transferir o controlo para o procedimento
3. Preservar os valores dos registos não temporários usados no procedimento
4. Executar a tarefa
5. Colocar o resultado num local onde o programa invocador lhe possa aceder
6. Recuperar os valores dos registos guardados em stack
7. Retornar o controlo ao ponto de origem

Procedimentos no MIPS

1. **Colocar parâmetros num local onde o procedimento lhes possa aceder**

   => usar registos $a0 .. $a3

Procedimentos no MIPS

2. Transferir o controlo para o procedimento

=> instrução `jal` (jump and link)

Ex: `jal Proc`

Procedimentos no MIPS

3. Preservar os valores dos registos não temporários usados no procedimento

=> alocar espaço em stack e lá preservar os registos não temporários

Ex: procedimento usa $s0 e $s1

- **alocar duas posições em stack (actualizar o stack pointer)**
- **guardar valores de $s0 e $s1 na stack**

Procedimentos no MIPS

4. Executar a tarefa

=> definir o conjunto de instruções que constituem o procedimento

Procedimentos no MIPS

5. Colocar o resultado num local onde o programa invocador lhe possa aceder

=> usar registos $v0 e $v1

Procedimentos no MIPS

6. Recuperar os valores dos registos guardados em stack

=> carregar os valores em stack para os respectivos registos

=> actualizar o stack pointer

Procedimentos no MIPS

7. Retornar o controlo ao ponto de origem

=> `jr $ra`

Procedimentos no MIPS

Exemplo de procedimento (programa invocador)

```
li $s0, 2
move $a0, $s0      # passa argumento
jal dobro          # invoca o proc. dobro
move $a0, $v0
li $v0, 1          # imprime o resultado
syscall
li $v0, 10
syscall
```

Procedimentos no MIPS

Exemplo de procedimento (procedimento)

```
dobro:
   sub $sp, $sp, 4 # aloca espaço stack
   sw $s0, 0($sp)  # guarda registo
   add $s0, $a0, $a0
   move $v0, $s0   # guarda resultado
   lw $s0, 0($sp)  # recupera registo
   addi $sp, $sp, 4 # actualiza $sp
   jr $ra          # retorna
```

Procedimentos no MIPS

- Exercícios
  - Implemente em assembly do MIPS um procedimento (addnsub) que execute a operação f=(g+h)-(i+j), sendo as variáveis locais aos procedimentos e inicializadas com valores arbitrários.

```
int addnsub(void)
{
   int temp0, temp1, f,g,h,i,j;

   g=3; h=7; i=1; j=4;
   temp0=g+h;
   temp1=i+j;
   f=temp0-temp1;
   printf("\nNo procedimento:");
   printf("\nf=%d\ng=%d\nh=%d",f,g,h);
   printf("\ni=%d\nj=%d",i,j);
   return f;
}

void main(void)
{
   int f,g,h,i,j,t0;

   f=10; g=20; h=30; i=40; j=50;
   printf("\nFora do procedimento:");
   printf("\nf=%d\ng=%d\nh=%d",f,g,h);
   printf("\ni=%d\nj=%d",i,j);
   t0=addnsub();
   printf("\nValor devolvido:%d",t0);
   printf("\nFora do procedimento:");
   printf("\nf=%d\ng=%d\nh=%d",f,g,h);
   printf("\ni=%d\nj=%d",i,j);
}
```

Procedimentos no MIPS

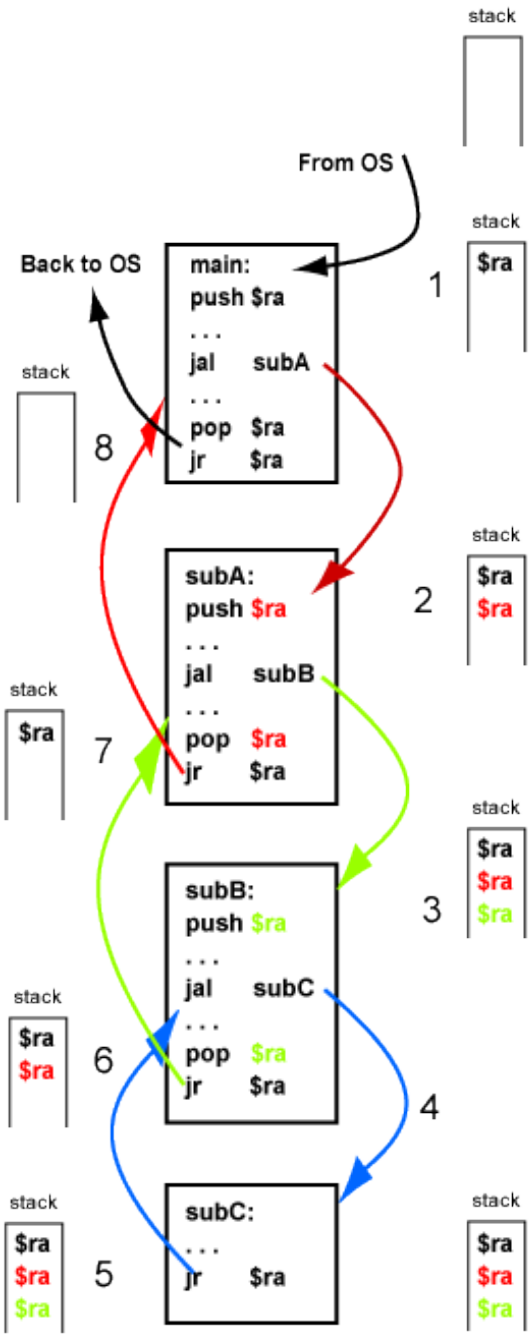
- Exercícios
  - Implemente em *assembly* do MIPS um procedimento que permita calcular o valor da função de *Fibonacci* para um dado valor de n

(Não implementar versão recursiva)

$$F_n = \begin{cases} 0 & se\ n = 0 \\ 1 & se\ n = 1 \\ F_{n-1} + F_{n-2} & se\ n \geq 2 \end{cases}$$

Recursividade

- Quer no caso de funções recursivas, ou no caso de funções que simplesmente invocam outras funções, é necessário guardar o endereço de retorno.
- Para isso recorre-se à *stack*. O push na stack do endereço de retorno faz-se logo no início antes de salvaguardar os conteúdos dos registos $s0..$s7.


stack
From OS
stack
$ra
1
Back to OS
main:
push $ra
. . .
jal subA
. . .
pop $ra
jr $ra
stack
8
stack
$ra
$ra
2
subA:
push $ra
. . .
jal subB
. . .
pop $ra
jr $ra
stack
$ra
7
stack
$ra
$ra
$ra
3
subB:
push $ra
. . .
jal subC
. . .
pop $ra
jr $ra
stack
$ra
$ra
6
4
stack
$ra
$ra
$ra
5
subC:
. . .
jr $ra
stack
$ra
$ra
$ra

Programa do factorial (recursivo)

Exercícios: Implemente em Assembly do MIPS um programa que recorre a um procedimento recursivo para calcular o factorial de um número.

Programa em Assembly

```
        .data
prompt1: .asciiz "Introduza o número:"
prompt2: .asciiz "O factorial é:"
        .text
        .globl  __start
__start:
         li     $v0,4              #   print string service
         la     $a0,prompt1        #   address of prompt1
         syscall
         li     $v0,5              #   read integer service
         syscall                   #   $v0 gets the integer
         move   $s0, $v0
         move   $a0, $s0
         jal    fact               #   Jump and link to subroutine
         move $s0, $v0
         li     $v0,4              #   print string service
         la     $a0,prompt2        #   address of prompt
         syscall
         move    $a0,$s0           #   load result into $a0
         li      $v0,1             #   print integer service
         syscall
        li $v0, 10
        syscall
```

Programa em Assembly (cont.)

```
fact:
        sub     $sp,$sp,4       # Push return address
        sw      $ra,($sp)
        sub     $sp,$sp,4       # Push register $s1
        sw      $s1,($sp)
        move    $s1,$a0         # save argument in $s1
        li      $t1,1           # get a 1
        bgt     $s1,$t1,calc    # if ( n<=1)
        li      $v0,1           #   return 1
        b       fim
calc:   sub     $a0,$s1,1       # else
        jal     fact
        mul     $v0,$v0,$s1     # n*fact(n-1)
fim:    lw      $s1,($sp)       # Pop register $s1
        add     $sp,$sp,4
        lw      $ra,($sp)       # Pop $ra
        add     $sp,$sp,4

        jr      $ra             #return to caller
```