Aula 4

2.4 [5] <§§2.2, 2.3> For the MIPS assembly instructions below, what is the corresponding C statement? Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. Assume that the base address of the arrays A and B are in registers `$s6` and `$s7`, respectively.

```
sll  $t0, $s0, 2     # $t0 = f * 4
add  $t0, $s6, $t0   # $t0 = &A[f]
sll  $t1, $s1, 2     # $t1 = g * 4
add  $t1, $s7, $t1   # $t1 = &B[g]
lw   $s0, 0($t0)     # f = A[f]
addi $t2, $t0, 4
lw   $t0, 0($t2)
add  $t0, $t0, $s0
sw   $t0, 0($t1)
```

Resposta:

B[g] = A[f + 1] + A[f];
f= A[f];

Considere as variáveis f, g, h, i, and j associadas aos registradores $s0, $s1, $s2, $s3, and $s4, respectivamente.

Considere que os vetores A e B estão nos registradores $s6 e $s7, respectivamente.

Converta para MIPS

f = g + A[B[4]-B[3]];

```
lw $t0, 16($s7)     // $t0 = B[4]
lw $t1, 12($s7)     // $t1 = B[3]
sub $t0, $t0, $t1   // $t0 = B[4] – B[3]
sll $t0, $t0, 2      //  $t0 = $t0 * 4
add $t0, $t0, $s6    //  $t0 = &A[B[4] – B[3]]
lw $t1, 0($t0)       // $t1 = A[B[4] – B[3]]
add $s0, $s1, $t1    // f = g + A[B[4] – B[3]
```

MARS

Funções

Chamada de funções

- Funções são usadas para organizar o código, permitindo estruturá-lo e reutilizar determinados trechos
- O programador pode concentrar em uma parte da tarefa de cada vez, organizando apenas os parâmetros das funções

Chamada de funções

1. Os paramêtros são colocados num local onde a funcão possa acessa-los
2. O controle da execução é transferido para a função
3. Os recursos de memória são alocados para a função
4. O bloco de código é executado
5. O valor de resultado é colocado num local onde quem chamou a função possa busca-lo
6. Retorna a execução ao ponto exato onde a função parou sua execução

Chamada de funções

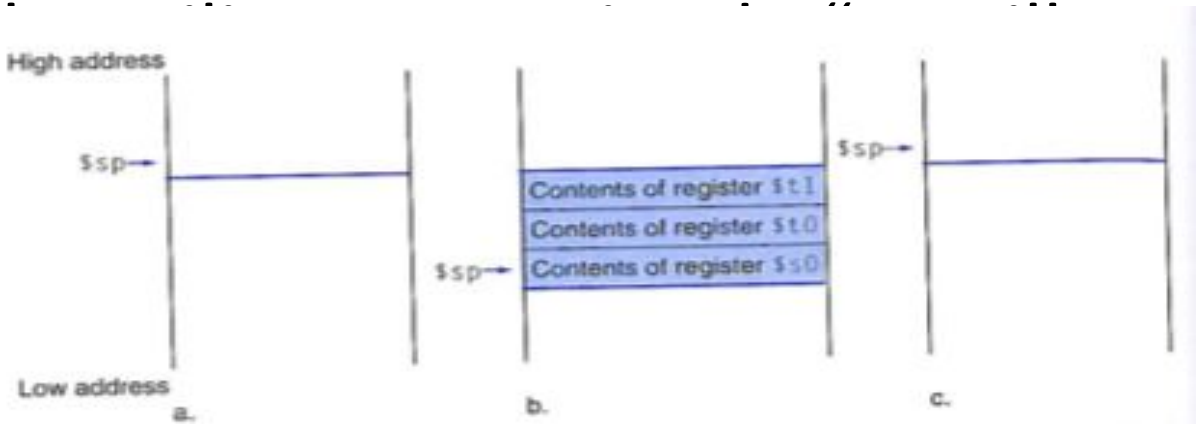
- Convenção do uso de registradores na chamada de funções
  - $a0 - $a3: quatro registradores para argumentos das funções
  - $v0- $v1: registradores para retorno de valores
  - $ra: registrador q. salva o local de retorno da função

Chamada de funções

- Instruções
  - Jal → jump and link: efetua o salto incondicional para o endereço indicado, salvando a origem no registrador $ra
  - Jr → jump register: efetua um salto incondicional para o endereço dentro do registrador. No caso das funções, seria chamado para o registrador $ra, jr $ra

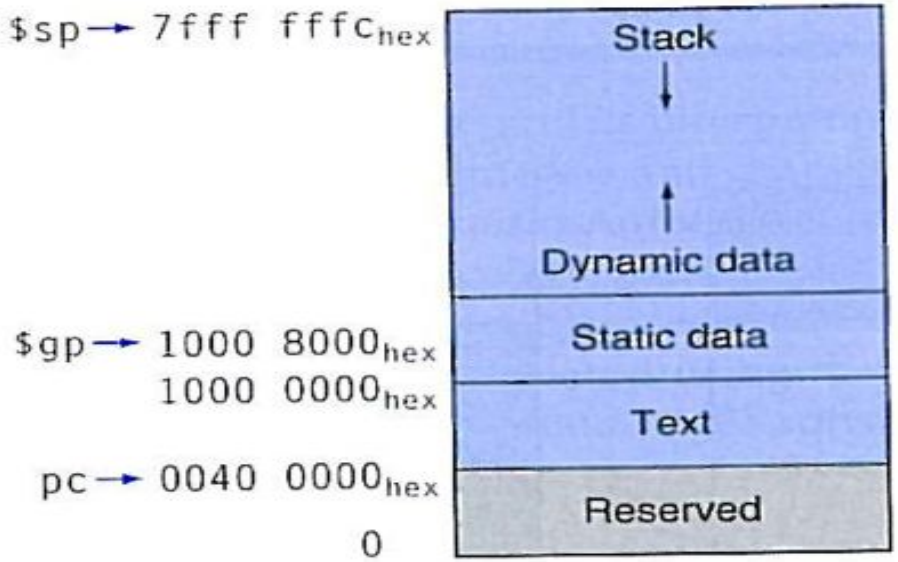
Chamada de funções

- Caso seja necessário chamar funções com mais parametros, o compilador irá alocar uma região da memória para eles, a pilha (stack)
- A p ento” da a o $sp

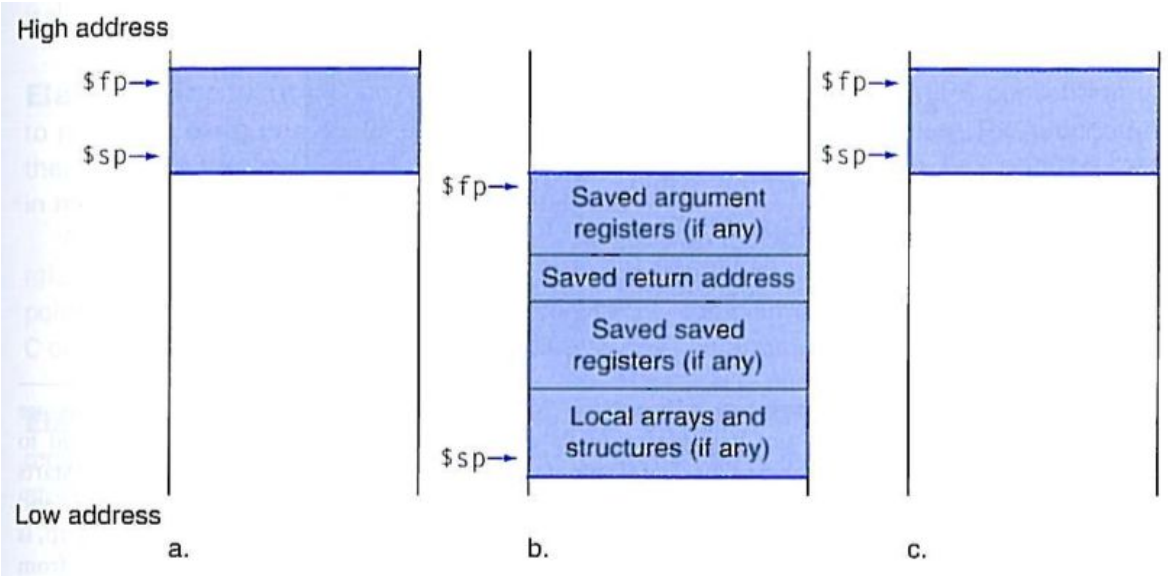


Chamada de funções

- Organização da memória



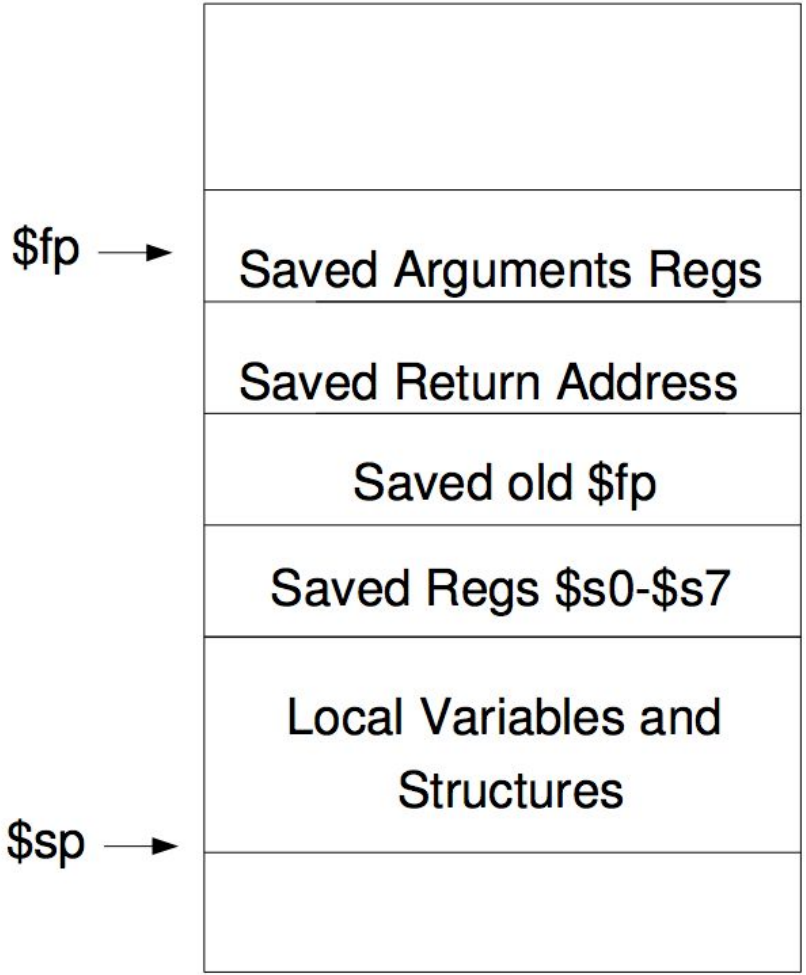
Chamada de funções



Chamada de funções

- Colocar um item na pilha:
  - Aloca espaço para o topo. Ex, três itens serão colocados: addi $sp, $sp,12
  - Cada um dos itens se encontra dentro de um registrador e serão colocados um a um:

```
sw $s0,8($sp)
sw $s1,4($sp)
sw $s2,0($sp)
```


$fp
Saved Arguments Regs
Saved Return Address
Saved old $fp
Saved Regs $s0-$s7
Local Variables and Structures
$sp

```
void func_a(void) {
        int a = 10;
        int b = 20;
        return;
}
```

```
.globl func_a
func_a:
sub $sp, $sp, 12
sw $ra, 8($sp)
li $t8, 10
sw $t8, 4($sp)
li $t8, 20
sw $t8, 0($sp)
lw $ra, 8($sp)
addi $sp, $sp, 12
jr $ra
```

```
void func_a(void) {
        int a = 10;
        int b = 20;
        return;
}
```

```
.globl func_a
func_a:
sub $sp, $sp, 12
sw $ra, 8($sp)
li $t8, 10
sw $t8, 4($sp)
li $t8, 20
sw $t8, 0($sp)
lw $ra, 8($sp)
addi $sp, $sp, 12
jr $ra
```

```
void func_a(void) {
    int a = 10;
    int b = 20;
    return;
}
```

```
.globl func_a
func_a:
sub $sp, $sp, 12
sw $ra, 8($sp)
li $t8, 10
sw $t8, 4($sp)
li $t8, 20
sw $t8, 0($sp)
lw $ra, 8($sp)
addi $sp, $sp, 12
jr $ra
```

```
void func_a(void) {
        int a = 10;
        int b = 20;
        return;
}
```

```
.globl func_a
func_a:
sub $sp, $sp, 12
sw $ra, 8($sp)
li $t8, 10
sw $t8, 4($sp)
li $t8, 20
sw $t8, 0($sp)
lw $ra, 8($sp)
addi $sp, $sp, 12
jr $ra
```

```
int leaf_example (int g, int h, int i, int j)
{
      int f;

      f = (g + h) - (i + j);
      return f;
}
```

- Aloca espaço na pilha para salvar os registradores que serão alterados

```
addi $sp, $sp, -12     # adjust stack to make room for 3 items
sw   $t1, 8($sp)       # save register $t1 for use afterwards
sw   $t0, 4($sp)       # save register $t0 for use afterwards
sw   $s0, 0($sp)       # save register $s0 for use afterwards
```

- Efetua a soma

```
add $t0,$a0,$a1 # register $t0 contains g + h
add $t1,$a2,$a3 # register $t1 contains i + j
sub $s0,$t0,$t1 # f = $t0 - $t1, which is (g + h)-(i + j)
```

```
add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)
```

- Salva o resultado no registrador de retorno

- Restaura os registradores com valores anteriores

```
lw  $s0, 0($sp)  # restore register $s0 for caller
lw  $t0, 4($sp)  # restore register $t0 for caller
lw  $t1, 8($sp)  # restore register $t1 for caller
addi $sp,$sp,12  # adjust stack to delete 3 items
```

```
jr  $ra    # jump back to calling routine
```

- Retorna

Chamada de funções

- Exemplo:
- int soma(int a, int b)

{ return a+b; }

Em MIPS:

Soma:

addi $sp,$sp,8#armazena espaço para dois itens

add $v0,$a0,$a1 # registradores $a0 e $a1 contém os parametros a,b

Jr $ra

Chamada de funções

```
a=0;
b=1;
c=soma(a,b);
--------------------------------
add $a0,$zero,$zero
addi $a1,$zero,1
jal soma
lw $v0,c
```

```
void swap(int v[], int k)
 {
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
 }
```

Procedure body

```
swap:  sll    $t1, $a1, 2          # reg $t1 = k * 4
       add    $t1, $a0, $t1        # reg $t1 = v + (k * 4)
                                   # reg $t1 has the address of v[k]
       lw     $t0, 0($t1)          # reg $t0 (temp) = v[k]
       lw     $t2, 4($t1)          # reg $t2 = v[k + 1]
                                   # refers to next element of v
       sw     $t2, 0($t1)          # v[k] = reg $t2
       sw     $t0, 4($t1)          # v[k+1] = reg $t0 (temp)
```

Procedure return

```
       jr     $ra                  # return to calling routine
```

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j =1) {
            swap(v,j);
        }
    }
}
```

Saving registers

```
sort:   addi    $sp,$sp, -20        # make room on stack for 5 registers
        sw      $ra, 16($sp)# save $ra on stack
        sw      $s3,12($sp)         # save $s3 on stack
        sw      $s2, 8($sp)# save $s2 on stack
        sw      $s1, 4($sp)# save $s1 on stack
        sw      $s0, 0($sp)# save $s0 on stack
```

Procedure body			
Move parameters		move	$s2, $a0 # copy parameter $a0 into $s2 (save $a0)
		move	$s3, $a1 # copy parameter $a1 into $s3 (save $a1)
Outer loop		move	$s0, $zero# i = 0
	for1tst:	slt	$t0, $s0,$s3 #reg$t0=0if$s0Š$s3(iŠn)
		beq	$t0, $zero, exit1# go to exit1 if $s0 Š $s3 (i Š n)
Inner loop		addi	$s1, $s0, -1# j = i - 1
	for2tst:	slti	$t0, $s1,0 #reg$t0=1if$s1<0(j<0)
		bne	$t0, $zero, exit2# go to exit2 if $s1 < 0 (j < 0)
		sll	$t1, $s1, 2# reg $t1 = j * 4
		add	$t2, $s2, $t1# reg $t2 = v + (j * 4)
		lw	$t3, 0($t2)# reg $t3 = v[j]
		lw	$t4, 4($t2)# reg $t4 = v[j + 1]
		slt	$t0, $t4, $t3 # reg $t0 = 0 if $t4 Š $t3
		beq	$t0, $zero, exit2# go to exit2 if $t4 Š $t3
Pass parameters and call		move	$a0, $s2 # 1st parameter of swap is v (old $a0)
		move	$a1, $s1 # 2nd parameter of swap is j
		jal	swap # swap code shown in Figure 2.25
Inner loop		addi	$s1, $s1, -1# j -= 1
		j	for2tst # jump to test of inner loop
Outer loop	exit2:	addi	$s0, $s0, 1 # i += 1
		j	for1tst # jump to test of outer loop

Restoring registers

```
exit1:  lw      $s0, 0($sp)         # restore $s0 from stack
        lw      $s1, 4($sp)# restore $s1 from stack
        lw      $s2, 8($sp)# restore $s2 from stack
        lw      $s3,12($sp)         # restore $s3 from stack
        lw      $ra,16($sp)         # restore $ra from stack
        addi    $sp,$sp, 20         # restore stack pointer
```

Procedure return

```
        jr      $ra                 # return to calling routine
```

Funções Recursivas

```
int fact (int n)
{
    if (n < 1) return (1);
          else return (n * fact(n - 1));
}
```

Funções Recursivas

- Na recursividade, os parâmetros devem ser salvos na pilha, para permitir restaurar o

```
fact:
    addi   $sp, $sp, -8  # adjust stack for 2 items
    sw     $ra, 4($sp)   # save the return address
    sw     $a0, 0($sp)   # save the argument n
```

Funções Recursivas

- Verifica a condição de para da recursividade

```
slti   $t0,$a0,1       # test for n < 1
beq    $t0,$zero,L1    # if n >= 1, go to L1
```

Funções Recursivas

- Sai da função recursiva

```
addi    $v0,$zero,1 # return 1
addi    $sp,$sp,8   # pop 2 items off stack
jr      $ra         # return to caller
```

Funções Recursivas

- Modifica o parâmetro para n-1 e chama a função novamente

```
L1: addi $a0,$a0,-1    # n >= 1: argument gets (n - 1)
    jal  fact          # call fact with (n - 1)
```

-

```
    lw   $a0, 0($sp)   # return from jal: restore argument n
    lw   $ra, 4($sp)   # restore the return address
    addi $sp, $sp, 8   # adjust stack pointer to pop 2 items
```

```
mul  $v0,$a0,$v0   # return n * fact (n - 1)

    jr   $ra            # return to the caller
```

Funções

- Nem toda função recursiva precisa ser implementada recursivamente em MIPS

```
int sum (int n, int acc) {
  if (n > 0)
      return sum(n - 1, acc + n);
  else
      return acc;
}
```

Funções

```
sum: slti $a0,1                  # test if n <= 0
     beq $a0, $zero, sum_exit    # go to sum_exit if n <= 0
     add $a1, $a1, $a0           # add n to acc
     addi $a0, $a0, -1           # subtract 1 from n
     j sum                       # go to sum
sum_exit:
     add $v0, $a1, $zero         # return value acc
     jr $ra                      # return to caller
```

Faça em MIPS

a.

```
int compare(int a, int b) {
    if (sub(a, b) >= 0)
        return 1;
    else
        return 0;
}
int sub (int a, int b) {
    return a-b;
}
```

b.

```
int fib_iter(int a, int b, int n){
    if(n == 0)
        return b;
    else
        return fib_iter(a+b, a, n-1);
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <stdlib.h>

int Vetor[10];
int indice, soma = 0, med =0;
int main() {
    soma(Vetor);
    media();
    return 0;
}

void soma(int Vetor[]){
    for(indice=0; indice<10; indice++) {
        soma = Vetor[indice] + soma;
    }
    printf("\nSoma: %d", soma);
}

void media(){
    med = soma / 10;
    printf("\nMedia: %d", med);
}
```

```
main:
  # configurações do programa principal
    subu $sp, $sp, 32   # cria um frame de pilha com 32 bytes
    sw $ra, 20($sp)     # salva o registrador $ra
    sw $fp, 16($sp)     # salva o registrador $fp
    addiu $sp, $sp, 28  # alinhamento de memória
```

```
  la $a0, vetor # carrega o vetor
  jal soma      # chama o procedimento soma

# Move o conteúdo do registrador de retorno ($v0) para o registrador
# ($a1), liberando-o para ser usado novamente
  move $a1, $v0

# chama o procedimento media
  jal media

  move $a1, $v0 # libera $v0
```

```
.text

 # imprimindo a string
  li $v0, 4
  la $a0, $LS
  syscall

 # imprimindo o inteiro
  li $v0, 1
  move $a0, $s1
  syscall

 # imprimindo a string
  li $v0, 4
  la $a0, $LM
  syscall

 # imprimindo o inteiro
  li $v0, 1
  move $a0, $s2
  syscall
```

```
# configurações do programa principal
lw $ra, 20($sp)       # restaura valor de $ra
lw $fp, 16($sp)       # restaura valor de $fp
addiu $sp, $sp, 32    # remove o frame de pilha
j fim                 # encerra o programa
```

Exercicio

- Faca a funcao Soma e funcao Media do exemplo anterior