

13. Arrays e Ponteiros

ABF - AC I - MIPS IS_3

43

Array de inteiros na Memória

Address	Data
23B8FF9C	array[999]
23B8FF98	array[998]
⋮	⋮
23B8F004	array[1]
23B8F000	array[0]

Main Memory

Arrays:

- Estrutura de dados em que todos os elementos são do mesmo tipo (*int, char, double, ...*)
- Cada elemento é identificado pelo seu nº de ordem (índice)
- Elementos sucessivos ocupam localizações contíguas na memória
- Dimensão fixa

Figure 6.21 Memory holding `array[1000]` starting at base address `0x23B8F000`

Copyright © 2013 Elsevier Inc. All rights reserved.

C Pointers


- Pointer: uma variável que contém o endereço de outra variável
 - Versão em linguagem de alto nível do endereço em linguagem máquina
- Porquê usar Pointers?
 - Por vezes são a única possibilidade de expressar uma computação
 - Código mais compacto e eficiente

ABF - AC 1 - Arrays e Pointers

45

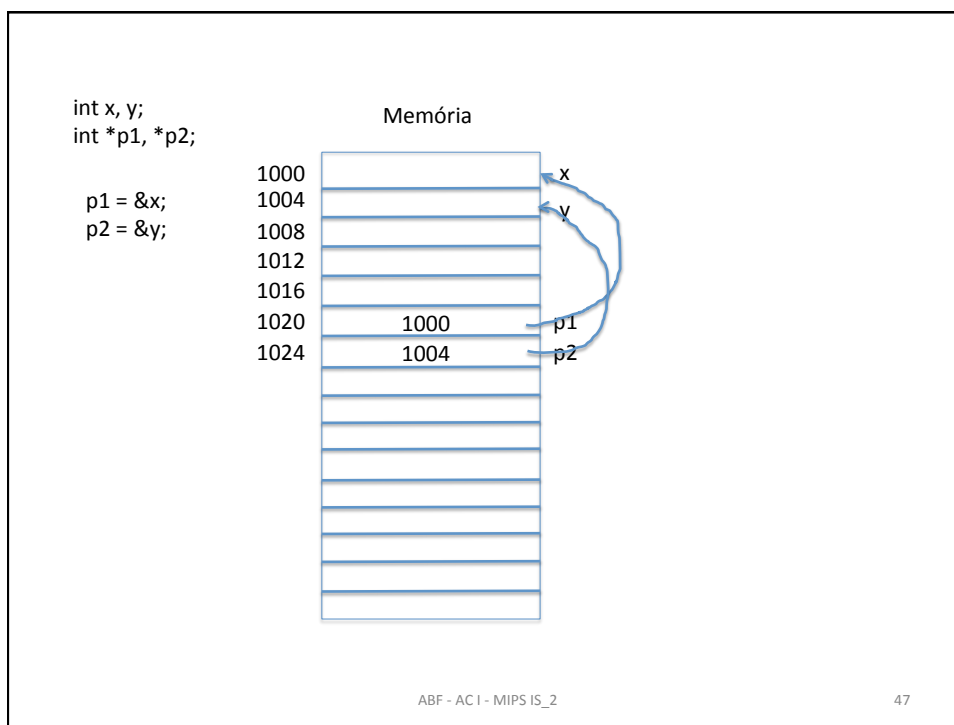
Ponteiros em C

- Variável **c** tem o valor 100, localizado no endereço de memória 0x10000000
- O operador **&** dá o endereço:
p = &c; atribui a **p** o endereço de **c**
 - **p** “points to” **c**: **p == 0x10000000**
- O operador ***** dá o valor para que o ponteiro aponta:
 if **p = &c;** then ***p == c** - “Dereferencing p”

Exemplo: $y = x$  $px = \&x;$
 $y = *px;$

ABF - AC 1 - Arrays e Pointers

46



Ponteiros em C

```

int x, y;
int *px    // px ponteiro para variáveis tipo int

```

Exemplos:

```

char *p    // p ponteiro para caracteres

```

- **Ponteiros como argumentos de funções**

- Em C os argumentos são passados por valor – a função não tem acesso à variável, apenas ao seu valor
- Para ser possível a função ter acesso à variável para alterar o seu valor tem que ser passado como argumento um ponteiro para a variável (i.e. o seu endereço e não o seu valor – “passagem por referência”)

Arrays e Ponteiros

```
int a[50]; /* array de 50 inteiros */
int *p;    /* ponteiro para um inteiro */
p = &a[0]; /* atribui a p o endereço de a[0] */
```

- No caso dos arrays a variável que designa o array indica o respetivo endereço-base, i.e.
 $p = a$ é equivalente a $p = \&a[0]$
- Se **p** aponta para um dado elemento do array,
p++ aponta para o elemento seguinte do array

ABF - AC 1 - MIPS IS_2

49

Ponteiros em Assembly

... = ***p**; \Rightarrow **load**

(obter o valor armazenado na posição apontada por p)

***p** = **...**; \Rightarrow **store**

(armazenar o valor na posição apontada por p)

ABF - AC 1 - Arrays e Pointers

50

Ponteiros em assembly

```

int c, tem o valor 100, no endereço
0x10000000,
p em $a0, x em $s0
# p = &c; /* p = 0x10000000 */
lui $a0, 0x1000 # p = 0x10000000
# x = *p; /* x = 100 */
lw $s0, 0($a0) # $s0 = 100
# *p = 200; /* c = 200 */
addi $t0, $0, 200
sw $t0, 0($a0) # c = 200

```

ABF - AC 1 - Arrays e Pointers

51

Indices vs. Ponteiros

```

clear1(int array[], int size) {
    int i;
    for (i = 0; i < size; i += 1)
        array[i] = 0;
}

```

```

        move $t0,$zero    # i = 0
loop1:  sll $t1,$t0,2      # $t1 = i * 4
        add $t2,$a0,$t1   # $t2 =
                                # &array[i]
        sw $zero, 0($t2)   # array[i] = 0
        addi $t0,$t0,1     # i = i + 1
        slt $t3,$t0,$a1    # $t3 =
                                # (i < size)
        bne $t3,$zero,loop1 # if (...)
                                # goto loop1

```

Ciclo: 6 instruções

```

clear2(int *array, int size) {
    int *p;
    for (p = &array[0]; p < &array[size];
        p = p + 1)
        *p = 0;
}

```

```

        move $t0,$a0      # p = & array[0]
        sll $t1,$a1,2      # $t1 = size * 4
        add $t2,$a0,$t1    # $t2 =
                                # &array[size]
loop2:  sw $zero,0($t0)    # Memory[p] = 0
        addi $t0,$t0,4     # p = p + 4
        slt $t3,$t0,$t2    # $t3 =
                                # (p < &array[size])
        bne $t3,$zero,loop2 # if (...)
                                # goto loop2

```

Ciclo: 4 instruções

ABF - AC 1 - Arrays e Pointers

52

11. Representação de outros tipos de dados

ABF - AC I - MIPS IS_2

53

Carateres

- Byte-encoded character sets
 - **ASCII**: 128 carateres – 7 bits + bit de paridade
 - 95 gráficos, 33 carateres de controle
 - **Unicode**: mais de 110 000 carateres
 - Usado em XML, .NET, Java, C++ wide characters, ...
 - Codifica a maioria dos alfabetos existentes, mais símbolos
 - Pode usar diferentes codificações dos carateres:
 - **UTF-8** - carateres ASCII representados num byte
 - **UTF-16** – carateres representados em 2 bytes (*halfword*) - Java

ABF - AC I - MIPS IS_2

54

O Código ASCII

ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(56	8	72	H	88	X	104	h	120	x
41)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

ABF - AC I - MIPS IS_3

55

Operações com *bytes* e *halfwords*

- Podem usar operações bitwise
 - MIPS byte/halfword load/store
 - Usados no processamento de *strings*
- `lb rt, offset(rs)` `lh rt, offset(rs)`
 – Sign extend a 32 bits em rt
- `lbu rt, offset(rs)` `lhu rt, offset(rs)`
 – Zero extend a 32 bits em rt
- `sb rt, offset(rs)` `sh rt, offset(rs)`
 – Store só o byte/halfword mais à direita

ABF - AC I - MIPS IS_2

56

Strings

- 3 alternativas para representar strings:
 1. A primeira posição da string é reservada para armazenar o seu comprimento
 2. Uma variável ligada à string indica o comprimento da string (como numa *structure*) – strings em Java
 3. A última posição da string é indicada por um carater usado para marcar o fim da string – em C uma string é terminada por um byte cujo valor é zero (o carater Null em ASCII)

ABF - AC I - MIPS IS_3

59

14. Invocação de funções/procedures (sub-rotinas)

ABF - AC I - MIPS IS_2

60

Funções e Procedimentos em C

- Mecanismo básico de estruturação de soluções aplicando o princípio “dividir para conquistar” (*divide and conquer*)

Tipo_do_resultado Nome_função (argumentos)

Tipo do valor da função
- valor retornado ao invocador
return (expressão)

Lista de parâmetros com
indicação dos respectivos tipos

- Procedimentos** –funções que não retornam valores ao programa que as invoca

ABF - AC I - MIPS IS_2

61

Função – módulo autónomo

- Comunicação entre a função invocada (*callee*) e o programa que a invoca (*caller*) unicamente através dos valores que lhe são passados como argumentos e do valor de retorno

Programa que invoca (*caller*)  Função invocada (*callee*)

- Quando a função é invocada é-lhe atribuído um conjunto de posições na memória (***stack frame***) para armazenar o valor das variáveis declaradas na função (variáveis locais). Quando a execução da função termina esse bloco de memória é libertado.

ABF - AC I - MIPS IS_2

62

Invocação de procedimentos

- Uma função/procedimento tem um contexto próprio de execução o que implica que o estado do processador (i.e. o conteúdo dos registos) seja guardado quando a função é invocada e seja restabelecido quando ela termina.
- A quem atribuir a tarefa de preservar e restaurar o conteúdo dos registos?
 1. A quem invoca a função (**caller**)? ou
 2. À função invocada (**callee**)?

ABF - AC I - MIPS IS_2

63

MIPS: invocação de procedimentos

- Registos usados para a comunicação entre o programa invocador e a função/procedimento invocado:
 - **\$a0 a \$a3** (r4 a r7): usados para passar os **argumentos**
 - **\$v0, \$v1** (r2 e r3): valores dos **resultados** da função
- Quem preserva os valores dos restantes registos?
 - **Caller**: responsável por preservar (no stack) o conteúdo dos registos temporários (**\$t0 a \$t9**) de que necessite após o retorno da função que invoca
 - **Callee**: responsável por preservar (no stack) o conteúdo dos registos **\$s0 a \$s7** que utilize e por o restaurar antes de retornar

ABF - AC I - MIPS IS_2

64

caller Invocação de procedimentos

1. Colocar os parâmetros em registos; guardar no **stack** os registos temporários em utilização
2. Transferir o controlo para o procedimento: **jal**
3. Adquirir espaço de memória para o procedimento (**stack frame**); guardar no **stack** os registos **\$s** que o procedimento vá utilizar
4. Executar as instruções do procedimento
5. Colocar o resultado num registo para o passar ao invocador; restaurar (copiar do **stack**) o conteúdo dos registos **\$s**; libertar o espaço de memória (**stack frame**) do procedimento (restaurar **\$sp**)
6. Regresso ao ponto do programa onde foi feita a chamada do procedimento **jr \$ra**

callee

ABF - AC I - MIPS IS_2

65

Utilização dos registos

- \$a0 – \$a3: argumentos (r4 – r7)
- \$v0, \$v1: valores dos resultados (r2 e r3)
- **\$ra**: return address (r31)
- **\$sp**: stack pointer (r29)
 - Usado para alocar e libertar memória do procedimento (**stack frame**)
- \$t0 – \$t9: temporários
 - O seu conteúdo pode ser destruído pelo procedimento invocado (**callee**)
- \$s0 – \$s7: preservados
 - Têm de ser preservados (saved/restored) pelo **callee**
- \$gp: global pointer para *static data* (r28)
- \$fp: frame pointer (r30)

ABF - AC I - MIPS IS_2

66

Instruções para invocação de procedimentos

- Procedure call: jump and link
jal Procedure_Label
 – Endereço da instrução seguinte colocado em \$ra
 – Salta para o endereço alvo
- Procedure return: jump register
jr \$ra
 – Copia \$ra para o program counter **(PC) = (\$ra)**

ABF - AC I - MIPS IS_2

67

Procedimentos: invocação e retorno

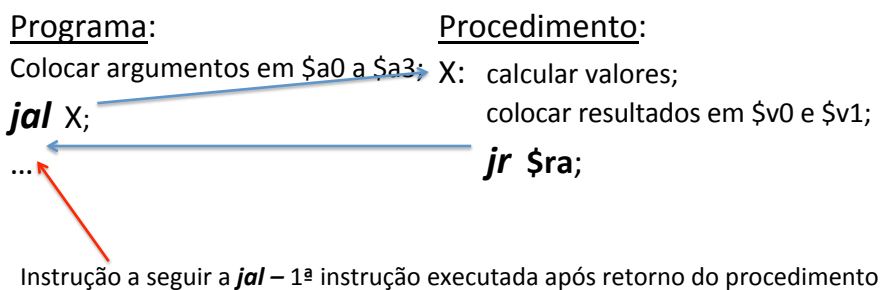
Programa:

Colocar argumentos em \$a0 a \$a3;

jal X;

...

Procedimento:

X: calcular valores;
colocar resultados em \$v0 e \$v1;***jr*** \$ra;


Instrução a seguir a ***jal*** – 1ª instrução executada após retorno do procedimento

ABF - AC I - MIPS IS_2

68

O stack

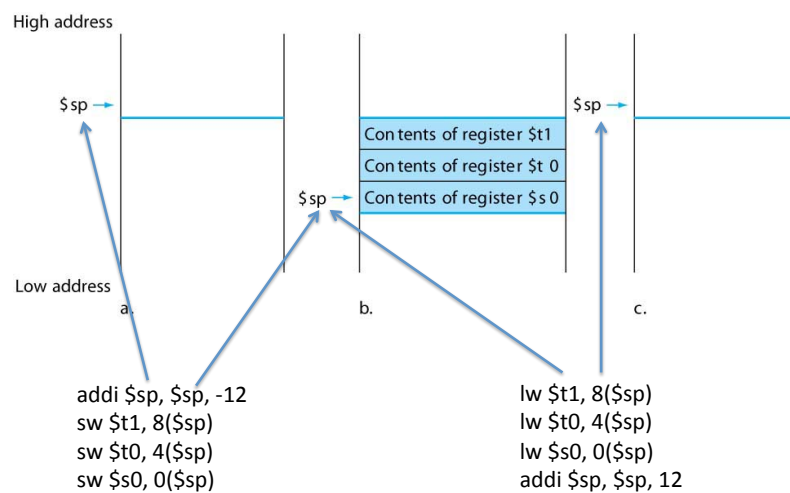
- Cresce dos endereços maiores para os mais pequenos
- **Stack Pointer (\$sp)** – aponta para o topo do stack

Address	Data		Address	Data	
7FFFFFFC	12345678	←\$sp	7FFFFFFC	12345678	
7FFFFFF8			7FFFFFF8	AABBCCDD	
7FFFFFF4			7FFFFFF4	11223344	←\$sp
7FFFFFF0			7FFFFFF0		
⋮	⋮		⋮	⋮	
⋮	⋮		⋮	⋮	

ABF - AC 1 - MIPS IS_2

69

O stack



ABF - AC 1 - MIPS IS_2

70

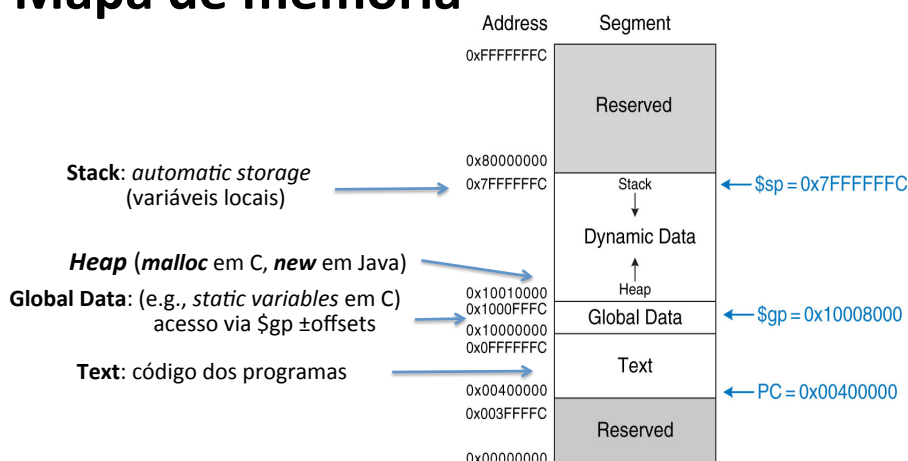
Tipos de funções / procedimentos e respectivas *stack frames*

- Os compiladores classificam as rotinas numa das seguintes categorias:
 - Funções que invocam outras funções ou a si próprias (recursivas) – ***non-leaf routines***
 - Funções que não invocam outras – ***leaf routines***
 - Requerem espaço no *stack* para variáveis locais
 - Não requerem espaço no *stack* para variáveis locais

ABF - AC I - MIPS IS_3

71

Mapa de memória



Reserved: reservado para o Sistema Operativo

ABF - AC I - MIPS IS_2

72

O que é e não é preservado na invocação de uma sub-rotina

Preservação da responsabilidade do invocado (*callee*)

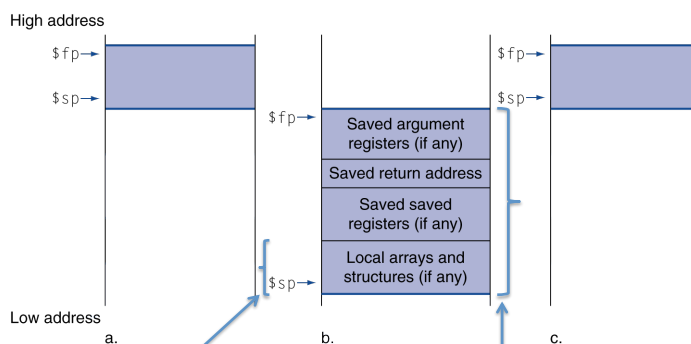
Preservação da responsabilidade do invocador (*caller*)

Registos preservados	Registos não preservados
\$s0 - \$s7	\$t0 - \$t9
Stack pointer \$sp	Registos dos argumentos \$a0 - \$a3
Registo com o endereço de retorno \$ra	Registos que retornam valores \$v0 - \$v1
Conteúdo do stack acima de \$sp	Conteúdo do stack abaixo de \$sp

ABF - AC I - MIPS IS_3

73

Dados locais no stack



Local data allocated by callee
(variáveis locais para que não haja
espaço nos registos)

Procedure frame (activation record)
\$fp - usado por alguns compiladores

ABF - AC I - MIPS IS_2

74

Exemplo: Função que não invoca outra

C

```
int main ()
{
    int y;
    ...
    y = diffsoma (1, 2, 3, 4);
    ...
}

int diffsoma (int g, h, i, j)
{
    int resultado;
    resultado = (g + h) - (i + j);
    return (resultado);
}
```

MIPS Assembly

```
# y mapeado em $s0
main:
    ...
    addi $a0, $0, 1
    addi $a1, $0, 2
    addi $a2, $0, 3
    addi $a3, $0, 4
    jal diffsoma
    add $s0, $v0, $0
    ...
diffsoma:
    # ver slide seguinte
```

ABF - AC I - MIPS IS_2

75

di ffsoma: código MIPS

utiliza registos \$s0, \$t0 e \$t1

```
addi $sp, $sp, -4
sw   $s0, 0($sp)
```

Guardar \$s0 no stack

```
add $t0, $a0, $a1
add $t1, $a2, $a3
sub $s0, $t0, $t1
```

Corpo do procedimento

```
add $v0, $s0, $zero
```

Resultado em \$v0

```
lw   $s0, 0($sp)
addi $sp, $sp, 4
```

Restaurar \$s0,
limpar stack

```
jr   $ra
```

Retorno ao *caller*

Stack



ABF - AC I - MIPS IS_2

76

Procedimentos que invocam outros procedimentos

- Para invocações em cadeia o **caller** precisa de guardar no stack:
 - O seu endereço de retorno
 - Valores de argumentos e variáveis temporárias de que necessite depois da invocação
- Restaurar o stack quando o procedimento que invocou retorna

ABF - AC1 - MIPS IS_2

77

Exemplo: Selection Sort (ordenação por ordem crescente)

- Ordenação dos elementos de um array escolhendo o menor dos elementos ainda não ordenado e colocando-o a seguir ao ultimo elemento ordenado. Para um array de inteiros com n elementos:
for (cada posição do índice i no array) {
 Determinar o índice j do menor valor entre i e n-1
 Trocar os elementos i e j do array
}

ABF - AC1 - MIPS IS_2

78

Selection Sort

```
void SelectionSort (int a[], int n)
{
    int i, j;

    for (i = 0; i < n; i++) {
        j = EncontrarMenor (a[], i, n-1);
        Troca (a[], i, j);
    }
}
```

- Procedimento que invoca outros – necessita salvar os argumentos com que foi invocada e o endereço de retorno. No entanto, devido ao seu 1º argumento ser também o 1º argumento das funções que invoca, não é necessário guardá-lo.

ABF - AC I - MIPS IS_2

79

Selection Sort - Assembly

\$a0 – endereço base do array; \$a1 – numero de elementos do array
i em \$s0

```
SelSort:    add    $s0, $0, $0        # i = 0
For1:      addi   $sp, $sp, -8
           sw     $a1, 4($sp)        # salva n
           sw     $ra, 0($sp)        # salva endereço de retorno
           addi   $a2, $a1, -1       # n-1 em $a2
           add    $a1, $s0, $0       # i em $a1
           jal    EncontrarMenor
           add    $a2, $v0, $0       # j em $a2
           jal    Troca
           lw     $a1, 4($sp)
           lw     $ra, 0($sp)
           addi   $sp, $sp, 8
           addi   $s0, $a1, 1        # i = i + 1
           blt    $s0, $a1, For1
           jr     $ra
```

ABF - AC I - MIPS IS_2

80

Encontrar menor elemento de (sub)array

- Determinar o índice j do menor valor entre i e n-1

```

EncontrarMenor (int array[], int low, int high)
{
    int i, posmenor;

    posmenor = low;
    for (i = low; i ≤ high; i++) {
        if (array[i] < array[posmenor]) posmenor = i;
    }
    return (posmenor);
}

```

ABF - AC I - MIPS IS_2

81

Encontrar Posição do menor: assembly

posmenor em \$t0; i em \$t1

```

EncontrarMenor:    add $t0, $a1, $0    # i = low
                   add $t1, $a1, $0    # posmenor = low
Forstart :         sll  $t8, $t0, 2
                   add $t8, $a0, $t8    # $t8 = endereço de a[i]
                   sll  $t9, $t1, 2
                   add $t9, $a0, $t9    # $t9 = endereço de a[posmenor]
                   lw   $t2, 0($t8)     # $t2 = a[i]
                   lw   $t3, 0($t9)     # $t3 = a[posmenor]
                   slt  $t4, $t3, $t2
                   beq  $t4, $0, Cont
                   add $t0, $t1, $0    # se (a[posmenor] < a[i]) posmenor = i
Cont:              addi $t0, $t1, 1
                   ble  $t0, $a2, Forstart
                   jr   $ra

```

ABF - AC I - MIPS IS_2

82

Troca

- Trocar os elementos i e j do array

```
void troca (int a[], int i, int j)
{
    int temp;
    temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

ABF - AC 1 - MIPS IS_2

83

Troca em assembly

```
Troca:  sll    $t1, $a1, 2
        add    $t1, $a0, $t1 # $t1 = endereço de a[i]
        sll    $t2, $a1, 2
        add    $t2, $a0, $t2 # $t2 = endereço de a[j]
        lw     $t0, 0($t1)    # $t0 (temp) = a[i]
        lw     $t3, 0($t2)    # $t3 = a[j]
        sw     $t3, 0($t1)    # a[i] = a[j]
        sw     $t0, 0($t2)    # a[j] = temp
        jr     $ra
```

ABF - AC 1 - Arrays e Pointers

84

Exemplo 3 – fatorial recursivo

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

- Argumento **n** em \$a0
- Resultado em \$v0

ABF - AC I - MIPS IS_2

85

fatorial recursivo – código MIPS

```
fact: addi $sp, $sp, -8    # push 2 items para o stack
      sw  $ra, 0($sp)     # save return address
      sw  $a0, 4($sp)     # save argument
      slti $t0, $a0, 1    # teste se n < 1
      beq $t0, $zero, L1  # if n < 1, resultado = 1
      addi $v0, $zero, 1  # limpar stack
      addi $sp, $sp, 8
      jr  $ra             # return
L1:   addi $a0, $a0, -1    # else decrementar n
      jal fact            # call recursiva
      lw  $a0, 4($sp)     # restaurar valor do argumento
      lw  $ra, 0($sp)     # restaurar return address
      addi $sp, $sp, 8    # pop 2 items do stack
      mul $v0, $a0, $v0   # resultado em $v0
      jr  $ra             # return
```

ABF - AC I - MIPS IS_2

86