

## Introdução à Arquitetura de Computadores

Aula 18

### Assembly 3: Arrays e Funções

#### Comparação de grandezas (<, >, <= e >=)

- Instrução `slt` (*set on less than*) e `slti, sltiu`

#### Arrays - Acesso a elementos

- Array de inteiros; Instruções `lw` e `sw`
- Código **ASCII**; Caracteres e *bytes*
- Array de *bytes*: Instruções `lb`, `lbu` e `sb`
  - Extensão de byte para 32bits

#### Funções

- Invocação e Retorno: instruções `jal` e `jr`
- Convenção de Uso de Registros:
  - Passagem de argumentos (`$a0-$a3`)
  - Retorno de valor (`$v0`)
  - Efeitos colaterais

### 1 - Comparação: Set on Less Than (slt) - (1)

#### A instrução `slt`

*(...continuação da última aula)*

1. Comparação (slt)

Até aqui usámos só as instruções `beq` e `bne` para testar a **igualdade** ou a **desigualdade** e saltar para um dado *label*.

Existe ainda a instrução `slt` para **comparar grandezas**.

*↳ Set on Less Than*

**Sintaxe:**      `slt $at, $t1, $t2` # \$at = (\$t1 < \$t2)?1:0

**Significado:**    `$at` é igual a **'1'** se `$t1 < $t2` ou igual a **'0'** no caso contrário.

**Uso:**            `slt` é sempre seguida dum **`beq/bne`** para testar o resultado da comparação (`$at`).

```

int sum = 0;
int i;
for(i=1; i<101; i=i*2){
    sum = sum + i;
}

```

for: *slt \$t1, \$s0, \$t0*      *i < 101*      # for(*i=1, i<101, i=x2*)  
*beq \$t1, \$0, done*

add \$s1, \$s1, \$s0      # *sum += i*  
*sll \$s0, \$s0, 1*      # *i x=2*

*j for*

Shift Left Logic

## 1 - Comparação: Set on Less Than (slt) - (2) - bge, etc

Pseudo-instruções: **bge, ble, bgt, blt,...**

Todas as **pseudo-instruções** de 'comparação de grandeza e salto', são convertidas em instruções nativas, pelo *Assembler*, através da instrução **slt**.

**Exemplo:**

```
bge    $t1, $t2, LABEL # jump if $t1 >= $t2
```

**Conversão:**

slt	\$at, \$t1, \$t2	# \$at = (\$t1 < \$t2) ? 1 : 0
beq	\$at, \$0, LABEL	# jump if \$at = 0

## 1 - Comparação: Set on Less Than (slt) - (3) - tipo-R

C Code	MIPS assembly code
<pre>// add the powers of 2 // from 1 to 100 int sum = 0; int i;  for (i=1; i &lt; 101;i = i*2){     sum = sum + i; }  O loop termina quando i &gt;=101</pre>	<pre># \$s0 = i, \$s1 = sum          addi \$s1,\$0,0      # sum = 0         addi \$s0,\$0,1      # i = 1         addi \$t0,\$0,101   # \$t0 = 101         # bge \$s0,\$t0,done # pseudo-instr. loop: slt \$t1,\$s0,\$t0 # \$t1 =(\$s0&lt;\$t0)?1:0         beq \$t1,\$0,done # if(\$t1==0)done         add \$s1,\$s1,\$s0 # sum = sum + i         sll \$s0,\$s0,1     # i = i*2         j    loop done:          \$t1 = 1 if i &lt; 101</pre>

A instrução **slt** seguida do **beq** implementa a pseudo-instrução **bge**. De facto, no MARS, podemos usar diretamente **bge** em vez de **slt + beq**!

	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">addiu \$9,\$0,0x0000000a</td><td style="padding: 2px;">2: main: li \$t1, 10</td></tr> <tr> <td style="padding: 2px;">addiu \$10,\$0,0x00000009</td><td style="padding: 2px;">3: li \$t2, 9</td></tr> <tr> <td style="padding: 2px;">slt \$t1,\$9,\$10</td><td style="padding: 2px;">4: bge \$t1, \$t2,skip</td></tr> <tr> <td style="padding: 2px;">beq \$t1,\$0,0x00000001</td><td></td></tr> <tr> <td style="padding: 2px;">addiu \$11,\$0,0xfffffff83</td><td style="padding: 2px;">5: li \$t3, -125</td></tr> <tr> <td style="padding: 2px;">nop</td><td style="padding: 2px;">6: skip: nop</td></tr> </table>	addiu \$9,\$0,0x0000000a	2: main: li \$t1, 10	addiu \$10,\$0,0x00000009	3: li \$t2, 9	slt \$t1,\$9,\$10	4: bge \$t1, \$t2,skip	beq \$t1,\$0,0x00000001		addiu \$11,\$0,0xfffffff83	5: li \$t3, -125	nop	6: skip: nop
addiu \$9,\$0,0x0000000a	2: main: li \$t1, 10												
addiu \$10,\$0,0x00000009	3: li \$t2, 9												
slt \$t1,\$9,\$10	4: bge \$t1, \$t2,skip												
beq \$t1,\$0,0x00000001													
addiu \$11,\$0,0xfffffff83	5: li \$t3, -125												
nop	6: skip: nop												

## 1 - Comparação: Set on Less Than Immediate (*slti*) - (4) - tipo-1

```
C Code
// add the powers of 2
// from 1 to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2){
    sum = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum

addi $s1, $0, 0
addi $s0, $0, 1
# addi $t0, $0, 101 # not needed

loop: slti $t1, $s0, 101
      beq $t1, $0, done
      add $s1, $s1, $s0
      sll $s0, $s0, 1
      j loop

done:
```

*Notes:*

- bge \$s0, 101, done

Para além da *slt* e *slti*, existem ainda as variantes *unsigned*, *sltu* e *sltiu*, para comparar grandezas sem sinal (para converter *bgeu*, *bltu*).

### Exemplo:

*slti \$t1, \$0, -1* e *sltiu \$t1, \$0, -1*

Resultados diferentes! Porquê?

*slt \$at, \$t0, \$t1* { Temos de ter em conta o tipo dos dados e o tipo das operações }

## 2 - Array (1) - Uso e Características

### Array

- É uma estrutura de dados usada para armazenar grandes quantidades de elementos do mesmo tipo (e.g., inteiro, caractere, etc).
- Os elementos ocupam posições de memória contíguas.



### Propriedades

- Tamanho (Size: N):** número de elementos
- Índice (0..N-1):** para aceder a cada elemento\*

\* Índice = Número de ordem do elemento no array

## 2 - Array (2) - Acesso a elementos tipo Inteiro - Exemplo

### Array\* com 5 elementos (tipo inteiro) em Memória

`int array[5]; // C Code`

Size = 5; Índice: 0..4

Endereço-Base = 0x10007000  
(Endereço do primeiro elemento)

#### Acesso aos elementos

Primeiro passo:

Colocar o Endereço-Base do array num registo.

\* Em IAC iremos considerar simplesmente arrays unidimensionais (i.e., vectors)!

© A. Nunes da Cruz

IAC - ASM3: Arrays e Funções



Address	Data
0x10007010	array[4]
0x1000700C	array[3]
0x10007008	array[2]
0x10007004	array[1]
0x10007000	array[0]

Cada elemento inteiro ocupa uma word (32bits = 4 bytes)

## 2 - Array (3) - Exemplo: Código C e Acesso em ASM

```
// C Code
int array[5] = {-2, 4, 5, 123, -324};
    array[0] = array[0] * 8;
    array[1] = array[1] * 8;
```

Address	Data
0x10007010	array[4]
0x1000700C	array[3]
0x10007008	array[2]
0x10007004	array[1]
0x10007000	array[0]

#### Procedimento de acesso (leitura/escrita)

1. Colocar o endereço do array num registo \$s0; [ la \$s0, array ]
2. Carregar o valor do elemento array[0] noutro registo \$t1;  
[ lw \$t1, 0(\$s0) ]
3. Neste caso, multiplicar o valor de \$t1 por 8; [ sll \$t1, \$t1, 3 ]
4. Usar a instrução sw para armazenar o novo valor de \$t1 na mesma posição de memória; [ sw \$t1, 0(\$s0) ]
5. Repetir os passos de 2 a 4 para o elemento array[1], ajustando o offset de 0 para 4.

© A. Nunes da Cruz

→ Cada elemento ocupa 4 bytes

IAC - ASM3: Arrays e Funções

7/25

→ S15C seria diferente

## 2 - Array (4) - Exemplo: Acesso em ASM

```
// C Code
int array[5] ={-2, 4, 5, 123, -324};
    array[0] = array[0] * 8;
    array[1] = array[1] * 8;

# MIPS assembly code
# $s0 = array base address ; la $s0,0x10007000
    lui  $s0, 0x1000          # 0x1000 in upper half of $s0
    ori  $s0, $s0, 0x7000      # 0x7000 in lower half of $s0
# array[0]
    lw   $t1, 0($s0)          # $t1 = array[0]
    sll $t1, $t1, 3 → x23  # $t1 = $t1<<3 = $t1 * 8
    sw   $t1, 0($s0)          # array[0] = $t1
# array[1]: byte offset = 4!
    lw   $t1, 4($s0)          # $t1 = array[1]
    sll $t1, $t1, 3 → x23  # $t1 = $t1<<3 = $t1 * 8
    sw   $t1, 4($s0)          # array[1] = $t1
```

*(Handwritten notes: 'Guarda a cópia' with arrows pointing to the first two assignments, and 'lui' written next to the first lui instruction.)*

Address	Data
0x10007010	array[4]
0x1000700C	array[3]
0x10007008	array[2]
0x10007004	array[1]
0x10007000	array[0]

Main Memory

© A. Nunes da Cruz

IAC - ASM3: Arrays e Funções

8/25

## 2 - Array (5) - Ciclo For em C

- A codificação anterior é **ineficiente** para arrays longos.
- Preferencial/ usam-se ciclos iterativos **for**, **while**, etc.

Exemplo: Usando um ciclo **for**

```
// C Code
int array[1000]; // words
int i;
for (i=0; i < 1000; i++) {
    array[i] = array[i] * 8;
```

*(Handwritten note: 'Acesso indexado a um Array (endereço de array índice i)')*

Size =1000; Índice: 0..999	
Address	Data
23B8FF9C	array[999]
23B8FF98	array[998]
:	:
23B8F004	array[1]
23B8F000	array[0]

Main Memory

Endereço-Base = 0x23B8F000

© A. Nunes da Cruz

IAC - ASM3: Arrays e Funções

9/25

*(Handwritten note: 'S(array[i]) = S(array[0]) + i \* tamanho de cada elemento' with arrows pointing to the formula and the table, and 'Lembre que os inteiros são 4 bytes' written below.)*

Calcular o endereço do array índice i SEMPRE que utilizarmos array[i]

## 2 - Array (6) - Ciclo For em ASM

# \$s0 = base address, \$s1 = i  
# for (i=0; i < 1000; i++) {  
# initialization code # la \$s0, 0x23B80000  
lui \$s0, 0x23B8 # \$s0 = 0x23B80000  
ori \$s0, \$s0, 0xF000 # \$s0 = 0x23B8F000  
addi \$s1, \$0, 0 # i = 0  
addi \$t2, \$0, 1000 # \$t2 = 1000  
loop: # for loop  
slt \$t0, \$s1, \$t2 # i < 1000?  
beq \$t0, \$0, done # if not then done

*Calcular o endereço*

*Mudar a cópia*

*Guardar a cópia*

sll \$t0, \$s1, 2 # \$t0 = i \* 4 (byte offset)  
add \$t0, \$t0, \$s0 # address of array[i]; \$t0 = array + 4\*i  
lw \$t1, 0(\$t0) # \$t1 = array[i]  
sll \$t1, \$t1, 3 # \$t1 = array[i] \* 8  
sw \$t1, 0(\$t0) # array[i] = array[i] \* 8

# next element  
addi \$s1, \$s1, 1 # i++  
j loop # } repeat  
done:

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

Main Memory

array + ix tam do cada elemento

```
int array[1000]; // words
int i;
for (i=0; i < 1000; i++){
    array[i] = array[i] * 8;
}
```

© A. Nunes da Cruz IAC - ASM3: Arrays e Funções 10/25

## 3 - Carateres e Bytes (1) - Código ASCII

American Standard Code for Information Interchange (ASCII)

Cada **caráter** (de texto) é representado pelo valor (único) de um **byte**

- Exemplos: 'S' = 0x53, 'a' = 0x61, 'A' = 0x41

A diferença entre as minúsculas ('a') e as maiúsculas ('A') é igual a 0x20 (32)

Um array de carateres é um array de múltiplos bytes.

O standard ASCII (1963) veio uniformizar o mapeamento entre carateres (do alfabeto Inglês) e bytes para facilitar a transmissão de texto entre computadores.

3. Caracteres e Bytes

### 3 - Caracteres e Bytes (2) - Tabela ASCII

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	'	70	p
21	!	31	1	41	A	51	0	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	\$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	:	4B	K	5B	L	6B	k	7B	l
2C	,	3C	<	4C	L	5C	\	6C	l	7C	
2D	-	3D	=	4D	M	5D	J	6D	m	7D	j
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

A diferença entre os maiúsculos e minúsculos é o mesmo!

\*Todos os bytes ASCII têm o bit de sinal igual a zero.

### 3 - Instruções Load/Store Byte (1)

- Para carregar (da memória) um registo de 32-bits com um byte, existem duas instruções: **lbu** e **lb**.

**lbu** - **load byte unsigned** → extensão com zeros  
faz a extensão (do byte) para 32-bits com zeros

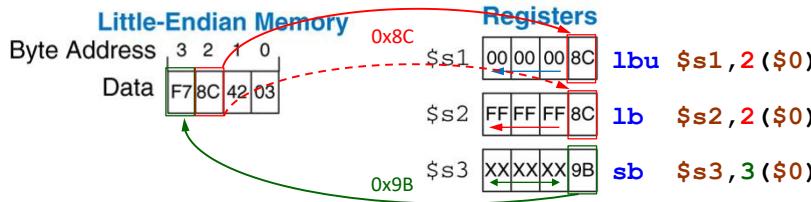
**lb** - **load byte** → extensão com o bit mais significativo  
faz a extensão para 32-bits com o bit de sinal do byte.

- Para armazenar (na memória) um byte dum registo de 32-bits, existe uma só instrução: **sb**.

**sb** - **store byte**  
armazena só o byte menos significativo do registo (LSB), no endereço (de byte) da memória, e ignora os restantes bytes.

Também existem instruções para aceder a *half-words* (16-bits): **lhu**, **lh** e **sth**.

### 3 - Instruções Load/Store Byte (2)



**lbu \$s1, 2 (\$0)** - carrega o byte **0x8C**, do endereço **2**, no LSB do registo **\$s1** e **preenche com zeros** os restantes 3 bytes.

**lb \$s2, 2 (\$0)** - carrega o byte **0x8C**, do endereço **2**, no LSB do registo **\$s2** e **preenche com 1's** (bit sinal) os restantes 3 bytes.

**sb \$s3, 3 (\$0)** - armazena o byte (LSB) **0x9B** do registo **\$s3**, no endereço **3**, da memória, substituindo o byte **0xF7** que lá se encontrava, **ignorando** os restantes bytes (**XX**).

\*Não é a memória que é Little-Endian, mas sim o CPU ☺...

### 3 - Exemplo lb/sb - Array de Bytes (1)

O seguinte código C converte um *array* com 10 **caracteres de minúsculas para maiúsculas**, subtraindo 32 (0x20) a cada elemento.

```
char chararr[10]; // bytes
int i;
for (i=0; i != 10; i++)
    chararr[i] = chararr[i] - 32;
```

Exemplo (ver tabela ASCII):

'a' (0x61) --> 'A' (0x41)

'A' = 'a' - 0x20 = 'a' - 32

#### Traduzir para Assembly

→ endereço do array índice  $i = \text{array} + i \times 1$   
→ 1 byte por caractere é 1 byte

- Ter em consideração que a **diferença entre endereços** de memória de dois elementos consecutivos do *array* é agora de **um só byte** e **não** de 4 bytes (caso do *array* de inteiros).
- Supor** que o registo **\$s0** já está inicializado com o endereço do *array* **chararr**.

### 3 - Exemplo Ib/sb - Array de Bytes (2)

```

char chararr[10]; // bytes (com bit de sinal = 0)
int i;
for (i=0; i != 10; i++)
    chararr[i] = chararr[i] - 32;

# $s0 = array base address = chararr = &chararr[0]
# $s1 = i
addi $s1, $0, 0          # i = 0
addi $t0, $0, 10          # $t0 = 10
# for loop
for:
    beq $s1, $t0, done    # if (i==10) done
    # $t1 = chararr + i = &chararr[i]
    → # não é necessário multiplicar o valor de i ($s1), porquê? → Porque era multiplicado
        # $t1 = address of chararr[i]
    add $t1, $s1, $s0        # $t1 = address of chararr[i]
    lb $t2, 0($t1)           # $t2 = chararr[i]
    addi $t2, $t2, -32       # conv_to_upcase: $t2 = $t2 - 32
    sb $t2, 0($t1)           # chararr[i] = chararr[i]-32
    addi $s1, $s1, 1          # i++
    j for                   # repeat
done:
    Note-se ainda a utilização das instruções lb e sb em vez de lw e sw.

```

Porque era multiplicado  
por 1  
máscara da  
ordem do  
valor da  
variação

© A. Nunes da Cruz

IAC - ASM3: Arrays e Funções

16/25

### 4 - Funções (1) - Introdução

#### 4. Funções

#### Definição

- As Linguagens de Alto-Nível usam *funções* (ou *subrotinas*) para *estruturar* um programa em módulos *reutilizáveis* e ainda para aumentar a *clareza* do código.

#### Argumentos e Retorno

- As funções possuem entradas, os **argumentos** (ou parâmetros), e uma saída, o valor de **retorno**.

#### Caller e Callee

- Quando uma função (*caller*) invoca outra (*callee*) é necessária uma convenção (conjunto de regras) para a passagem dos argumentos e para a recolha do valor retorno devolvido pela função.

© A. Nunes da Cruz

IAC - ASM3: Arrays e Funções

17/25

## **4 - Funções (2) - Caller e Callee através de Exemplo**

Mano:  
 li \$ao, 42  
 li \$a1, ?  
 j \_main\_

back:  


Sum: add \$v0, \$ao, \$a1  
 j back

não pode volta para back  
 pois, se chovermos outra vez ele volta para traz

```
int sum(int a, int b);  
//  
int main(){  
    int y;  
    y = sum(42, 7);  
    ... // y = 49  
    return 0;  
}  
//  
int sum(int a, int b){  
    return (a + b);  
}
```

• A função sum  
• main  
 a e k  
 video

→ Convenção

- A função `main` invoca a função `sum` para calcular a soma de `a + b`.
  - `main` (*caller*) passa os argumentos `a` e `b` e recebe o resultado devolvido pela função `sum` (*callee*).

A red downward-pointing arrow indicating the direction of the next step in the process.

- **Caller:** função *Invocadora* (`main`)
  - **Callee:** função *Invocada* (`sum`)

main:  
 li \$a0, 42  
 li \$a1, 7  
 jal sum  
 A.N.  
 back:  
 jal colora mo  
 e andarle  
 jal sum foi  
 de onthe  
 comando

nes da Cruz

IAC - ASM3: Arrays e Funções

18/25

## **4 - Funções (3) - Procedimento de Invocação e Retorno**

## ***Caller* (Invocadora)**

- Passa os **argumentos** à *Callee*
  - 'Salta' para o código da *Callee*
  - Usa (ou não) o resultado devolvido

### **Callee (Invocada)**

- Usa os argumentos para **executar** o código da função
  - **Devolve** o resultado à *Caller*
  - **Regressa** ao código donde foi chamada
  - **Não deve alterar** registos ou memória necessários à *Caller*.

## 4 - Funções (4) - Instruções e Convenção MIPS

4.1 Funções - Convenção MIPS

### Instruções

- **Invocar** uma função: jump and link (**jal**)  
(*Caller*: executa **jal <Callee>**)
- **Retornar** duma função: jump register (**jr**)  
(*Callee*: executa **jr \$ra**)

### Convenção

- **Argumentos:**      **\$a0 - \$a3**  
(*Caller*: passa **\$a0..\$a3** à *Callee*)
- **Valor de Retorno:** **\$v0**  
(*Callee*: devolve **\$v0** à *Caller*)

## 4 - Funções (5) - Instruções MIPS: jal e jr

<pre> int main() {     simple();     a = b + c;     return 0; }  void simple() {     return; } </pre>	<p>→ 0x00400200 <b>main:</b> jal simple  → 0x00400204                add \$s0, \$s1, \$s2  ...  # no return value!</p> <p>→ 0x00401020 <b>simple:</b> jr \$ra</p> <p><b>jal simple :</b> 'salta' (jump) para <b>simple</b> e 'liga' (link)  <b>\$ra</b> = <u>PC + 4</u> = 0x00400204  <small>To Program counter</small></p> <p><b>jr \$ra :</b> 'salta' para o endereço contido em  <b>\$ra</b> (0x00400204)  (i.e., regressa ao ponto após a <b>jal</b>)  <b>\$ra</b> - <u>r</u>eturn <u>a</u>ddress</p>
---	---

**void** - significa que a função 'simple' não devolve qualquer valor.

## 4 - Funções (6) - Argumentos e Retorno - Código C

A função `diffofsums` é invocada com **quatro** argumentos e devolve o resultado em `$v0`.

A função `caller` coloca os argumentos nos registos `$a0-$a3`.

A função `callee` devolve o resultado no registo `$v0`.

```
int main(){
    int y;
    ...
    y = diffofsums(2, 3, 4, 5); // 4 arguments
    ...
}

int diffofsums(int f, int g, int h, int i){
    int result;
    result = (f + g) - (h + i);
    return result;           // return value
}
```

## 4 - Funções (7) - Argumentos e Retorno - Código ASM

<code># \$s0 = y</code>	main coloca os argumentos nos registos <code>\$a0-\$a3</code> ;
<code>main:</code>	<code>diffofsums</code> devolve o resultado no registo <code>\$v0</code> .

```
...
addi $a0, $0, 2      # arg0 (f) = 2
addi $a1, $0, 3      # arg1 (g) = 3
addi $a2, $0, 4      # arg2 (h) = 4
addi $a3, $0, 5      # arg3 (i) = 5
→ jal diffofsums    # call Function
→ add $s0, $v0, $0   # y = returned value
...

# $s0 = result → Podem estar a ser utilizados na main. Temos um problema!
diffofsums:
    add $t0, $a0, $a1    # $t0 = f + g
    add $t1, $a2, $a3    # $t1 = h + i
    sub $s0, $t0, $t1    # result = (f + g) - (h + i)
    add $v0, $s0, $0      # put return value in $v0
→ jr $ra              # return to caller
```

## 4 - Funções (8) - Argumentos e Retorno - Código ASM\_2

```
# $s0 = y
main:
    ...
    addi $a0, $0, 2      # arg0 (f) = 2
    addi $a1, $0, 3      # arg1 (g) = 3
    addi $a2, $0, 4      # arg2 (h) = 4
    addi $a3, $0, 5      # arg3 (i) = 5
    jal diffofsums       # call Function
    add $s0, $v0, $0      # y = returned value
    ...

# $s0 = result; isto não é necessário!
diffofsums:
    add $t0, $a0, $a1    # $t0 = f + g
    add $t1, $a2, $a3    # $t1 = h + i
    sub $v0, $t0, $t1    # $v0 = (f + g) - (h + i)
    #add $v0, $s0, $0    # put return value in $v0
    jr $ra               # return to caller
```

main coloca os argumentos nos registos \$a0-\$a3;  
diffofsums devolve o resultado no registo \$v0 .

O código de diffofsums podia ser simplificado, mas esse não o ponto, por agora ☺.

## 4 - Funções (9) - Salvaguarda de Registos - O Problema

```
# $s0 = result
diffofsums:
    add $t0, $a0, $a1    # $t0 = f + g
    add $t1, $a2, $a3    # $t1 = h + i
    sub $s0, $t0, $t1    # result = (f + g) - (h + i)
    add $v0, $s0, $0      # put return value in $v0
    jr $ra               # return to caller
```

diffofsums alterou três registos: \$t0 , \$t1 e \$s0 !

4.2 Funções - Efeitos colaterais

P: E se a função main também usar esses registos?

R: main e/ou diffofsums podem salvaguardar temporaria/ o conteúdo dos registos na **stack**, permitindo a respectiva **reutilização**.

próxima aula: **stack**

## X - Registros MIPS

Name	Register Number	Usage
\$0	0	the constant value 0
\$at	1	assembler temporary
\$v0-\$v1	2-3	Function return values
\$a0-\$a3	4-7	Function arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved variables
\$t8-\$t9	24-25	more temporaries
\$k0-\$k1	26-27	OS temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	Function return address

© A. Nunes da Cruz

IAC - ASM3: Arrays e Funções

27/26

## Y - MIPS - Assembly Instruction Set - Summary

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3	Three register operands
	subtract	sub \$s1,\$s2,\$s3	\$s1 = \$s2 - \$s3	Three register operands
	add immediate	addi \$s1,\$s2,20	\$s1 = \$s2 + 20	Used to add constants
Data transfer	load word	lw \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Word from memory to register
	store word	sw \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Word from register to memory
	load half	lh \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Halfword memory to register
	load half unsigned	lhu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Halfword memory to register
	store half	sh \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Halfword register to memory
	load byte	lb \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Byte from memory to register
	store byte	sb \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Byte from register to memory
	load linked word	ll \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Load word as 1st half of atomic swap
	store condition.word	sc \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1; \$s1 = 0 or 1	Store word as 2nd half of atomic swap
Logical	load upper immed.	lui \$s1,20	\$s1 = 20 * 2 <sup>16</sup>	Loads constant in upper 16 bits
	and	and \$s1,\$s2,\$s3	\$s1 = \$s2 & \$s3	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	\$s1 = \$s2   \$s3	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	\$s1 = ~(\$s2   \$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,20	\$s1 = \$s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,20	\$s1 = \$s2   20	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	\$s1 = \$s2 << 10	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	\$s1 = \$s2 >> 10	Shift right by constant
	branch on equal	beq \$s1,\$s2,25	If (\$s1 == \$s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	If (\$s1 != \$s2) go to PC + 4 + 100	Not equal test; PC-relative
Conditional branch	set on less than	slt \$s1,\$s2,\$s3	If (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than; for beq, bne
	set on less than	sltu \$s1,\$s2,\$s3	If (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than unsigned
	set less than immediate	slti \$s1,\$s2,20	If (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu \$s1,\$s2,20	If (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant unsigned
	jump	jr \$s1,2500	go to 10000	Jump to target address
Unconditional	jump register	jr \$s1	go to \$ra	For switch, procedure return
	jump and link	jal 2500	\$ra = PC + 4; go to 10000	For procedure call

© A. Nunes da Cruz

28/26