
Introdução à Arquitetura de Computadores

Bloco 5

Instruções do MIPS

Pedro M. Lavrador

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro
plavrador@ua.pt

1

Índice

- Linguagem Máquina
 - Introdução
- Codificação de Instruções
 - Tipo-R, tipo-I e tipo-J
 - Exemplos: Aritméticas (**add**) e de *Load/Store* (**lw**)
- Programa em Memória
 - Inicialização e Execução
- Decodificação de Instruções
 - Exemplos: tipo-R (**sub**) e tipo-I (**addi**)

01/04/2025

PML – IAC - 2025

2

2

1 - Linguagem Máquina - Introdução (1)

- Instrução
 - Uma instrução corresponde a uma única operação que o processador pode executar, de entre as múltiplas definidas pelo respectivo *instruction set*.
- Linguagem Máquina (LM)
 - Em LM, as instruções são representadas em binário, por palavras cujo comprimento, nas arquiteturas RISC, é normalmente fixo e igual a 32-bits.

01/04/2025

PML – IAC - 2025

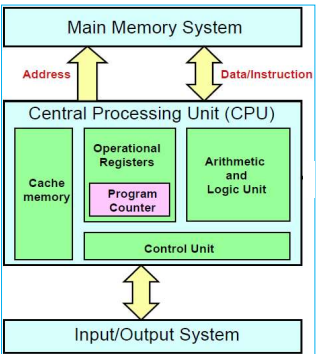
3

3

1 - Linguagem Máquina - Introdução

- Tipo de operações numa Instrução
 - Um computador possui Instruções para quatro tipo de operações:

- Transferência de Dados entre a Memória e os Registos do CPU (ex: *lw* e *sw*).
- Execução de operações Aritméticas ou Lógicas sobre os Dados (na ALU) (ex: *add* e *xor*)



- Controlo do Fluxo de Execução do programa (ex: *bne* e *j*)
- Entrada/Saída de Dados (opcional e ausente no MIPS)

As Instruções são comandos para: transferência de dados ou execução de operações aritméticas/lógicas ou ainda para controlo do fluxo de execução do programa.

01/04/2025

PML – IAC - 2025

4

4

1 - Linguagem Máquina - Introdução (3)

- Instruções na forma de números binários
- Comprimento de palavra de 32-bits (μP de 32-bits):
 - Nos CPUs RISC, assume-se, que tanto os Dados como os Registos do CPU e ainda as Instruções possuem o **mesmo** comprimento.
- Como são codificadas as instruções?
 - Uma vez que a instrução possui um comprimento de 32-bits, esta é dividida em grupos-de-bits (*bitfields*);
 - Cada um destes grupos **diz algo** sobre a instrução:
Exs de bitfields: código de operação, operandos, endereços.

01/04/2025

PML – IAC - 2025

5

5

Índice

- Linguagem Máquina
 - Introdução
- Codificação de Instruções
 - Tipo-R, tipo-I e tipo-J
 - Exemplos: Aritméticas (**add**) e de *Load/Store* (**lw**)
- Programa em Memória
 - Inicialização e Execução
- Descodificação de Instruções
 - Exemplos: tipo-R (sub) e tipo-I (addi)

01/04/2025

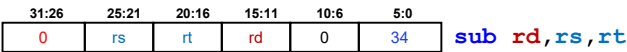
PML – IAC - 2025

6

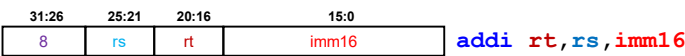
6

1 - Linguagem Máquina - As instruções do µP MIPS

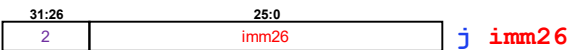
- Codificação binária das instruções
 - Todas as instruções têm 32 bits!
 - Existem 3 formatos de instrução:
 - tipo-R:** dois operandos contidos em registos



- tipo-I:** um dos operandos é uma constante



- tipo-J:** o único operando é um endereço



01/04/2025

PML – IAC - 2025

7

7

1 - Instruções do tipo-R(egister) (1) - BitFields

| | | | | | |
|--------|--------|--------|--------|--------|--------|
| op | rs | rt | rd | shamt | funct |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

- 3 registos
 - rs, rt: 2 registos fonte (rs= primeiro e rt=segundo) ou operandos
 - rd: registo destino
- Outros campos
 - op opcode (é sempre 0 nas instruções tipo-R)
 - funct: Função, a qual juntamente com opcode, especifica a operação a ser executada.
 - shamt: Shift Amount é uma constante usada só nas operações de Shift (deslocamento de bits à direita ou à esquerda). Nos outros casos é zero.

01/04/2025

PML – IAC - 2025

8

8

1 - tipo-R (2) - Assembly vs Máquina (1): Conversão

Código Assembly

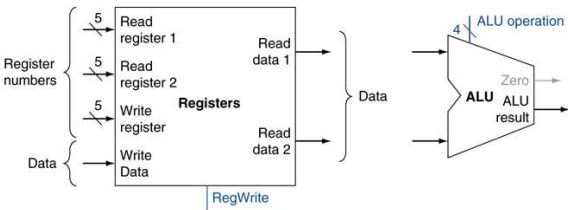
Código Máquina

```
add $s0, $s1, $s2
sub $t0, $t3, $t5
```



| op | rs | rt | rd | shamt | funct |
|--------|--------|--------|--------|--------|--------|
| 0 | 17 | 18 | 16 | 0 | 32 |
| 0 | 11 | 13 | 8 | 0 | 34 |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

- Começamos com o tipo de instruções mais simples (i.e., aquelas cuja execução envolve só a interação entre o Banco de Registos e a Unidade Aritmética e Lógica (ou ALU).



1 - tipo-R (2) - Asm vs Máq (2): opcode & funct

Código Assembly

Código Máquina

```
add $s0, $s1, $s2
sub $t0, $t3, $t5
```



| op | rs | rt | rd | shamt | funct |
|--------|--------|--------|--------|--------|--------|
| 0 | 17 | 18 | 16 | 0 | 32 |
| 0 | 11 | 13 | 8 | 0 | 34 |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

- Nas instruções tipo-R o **opcode** é sempre ZERO. É o campo **funct** determina a operação que a ALU faz.
- Em geral, a conversão das instruções *Assembly* para Código Máquina é feita através da consulta de tabelas.
 - Neste caso, precisamos duma tabela para os registos **rs**, **rt** e **rd** e outra tabela para o código de função **funct**.

1 - tipo-R (2) - Asm vs Máq (3): opcode & funct

Código Assembly

```
add $s0, $s1, $s2
sub $t0, $t3, $t5
```

Código Máquina

| op | rs | rt | rd | shamt | funct |
|----|----|----|----|-------|-------|
| 0 | 17 | 18 | 16 | 0 | 32 |
| 0 | 11 | 13 | 8 | 0 | 34 |

1. O número dos registos é dado pela Tabela 6.1 (para todas as instruções).

2. O código de função é dado pela Tabela B.2. Para as instruções add = 32 e sub = 34.

| Name | Number | Use |
|-----------|--------|---------------------|
| \$t0-\$t7 | 8-15 | temporary variables |
| \$s0-\$s7 | 16-23 | saved variables |
| \$t8-\$t9 | 24-25 | temporary variables |

Tabela 6.1 - (pg. 300)

\$s0 = 16, \$s1 = 17, \$s2 = 18

\$t0 = 8, \$t3 = 11, \$t5 = 13

| Funct | Name |
|-------------|-----------------|
| 100000 (32) | add rd, rs, rt |
| 100001 (33) | addu rd, rs, rt |
| 100010 (34) | sub rd, rs, rt |

Tabela B.2 - (pg. 622)

3. Estas instruções do tipo-R, add e sub, possuem um shamt igual a zero.

01/04/2025

PML – IAC - 2025

11

11

1 - tipo-R (2) - Tabela de Registos

Table 6.1 MIPS register set

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

Tabela 6.1 - Nome, número e respectivo uso, de cada registo.

01/04/2025

PML – IAC - 2025

12

12

PML - 2025

6

1 - tipo-R (2) - Tabela de Código de Função (funct)

Table B.2 R-type instructions, sorted by funct

| Funct | Name | Description |
|-------------|-------------------|---------------------------------|
| 000000 (0) | sll rd, rt, shamt | shift left logical |
| 000010 (2) | srl rd, rt, shamt | shift right logical |
| 000011 (3) | sra rd, rt, shamt | shift right arithmetic |
| 000100 (4) | sllv rd, rt, rs | shift left logical variable |
| 000110 (6) | srlv rd, rt, rs | shift right logical variable |
| 000111 (7) | srav rd, rt, rs | shift right arithmetic variable |
| 001000 (8) | jrr rs | jump register |
| 001001 (9) | jair rs | jump and link register |
| 001100 (12) | syscall | system call |
| 001101 (13) | break | break |
| 010000 (16) | mghi rd | move from hi |
| 010001 (17) | mthi rs | move to hi |
| 010010 (18) | mflo rd | move from lo |
| 010011 (19) | mtlo rs | move to lo |
| 011000 (24) | mult rs, rt | multiply |
| 011001 (25) | multu rs, rt | multiply unsigned |
| 011010 (26) | div rs, rt | divide |
| 011011 (27) | divu rs, rt | divide unsigned |

Table B.2 R-type instructions, sorted by funct field-

| Funct | Name | Description |
|-------------|-----------------|------------------------|
| 100000 (32) | add rd, rs, rt | add |
| 100001 (33) | addu rd, rs, rt | add unsigned |
| 100010 (34) | sub rd, rs, rt | subtract |
| 100011 (35) | subu rd, rs, rt | subtract unsigned |
| 100100 (36) | and rd, rs, rt | and |
| 100101 (37) | or rd, rs, rt | or |
| 100110 (38) | xor rd, rs, rt | xor |
| 100111 (39) | nor rd, rs, rt | nor |
| 101010 (42) | slt rd, rs, rt | set less than |
| 101011 (43) | sltu rd, rs, rt | set less than unsigned |

Tabela B.2
Instruções do tipo-R ordenadas pelo
campo funct.

Type-R Function Code: ADD, SUB

01/04/2025

PML – IAC - 2025

13

13

1 - tipo-R - Asm vs Máq (4) - Em binário e hexadecimal

Código Assembly

```
add $s0, $s1, $s2
sub $t0, $t3, $t5
```

Código Máquina

| op | rs | rt | rd | shamt | funct |
|----|----|----|----|-------|-------|
| 0 | 17 | 18 | 16 | 0 | 32 |
| 0 | 11 | 13 | 8 | 0 | 34 |

6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

Em binário (32-bits):

| 0 | 2 | 3 | 2 | 8 | 0 | 2 | 0 |
|--------|-------|-------|-------|-------|--------|---|---|
| op | rs | rt | rd | shamt | funct | | |
| 000000 | 10001 | 10010 | 10000 | 00000 | 100000 | | |
| 000000 | 01011 | 01101 | 01000 | 00000 | 100010 | | |

6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

0 1 6 D 4 0 2 2

Em hexadecimal (8-hexas):

(0x02328020)

(0x016D4022)

Ordem dos bitfields:

Em Assembly: add rd, rs, rt

Em Máquina: <0> rs, rt, rd,shamt,<funct>

01/04/2025

PML – IAC - 2025

14

14

PML - 2025

7

1 - tipo-R - Exercício Codificação (1) - add (1)

- Qual é o código máquina da seguinte instrução *Assembly*?
 - add** \$t0, \$s4, \$s5
- Sabemos que add é uma instrução tipo-R
- Da Tabela 6.1 (pg. 300), tiramos:
\$t0 = **rd** = 8, \$s4 = **rs** = 20 e \$s5 = **rt** = 21
onde **rd**=registro destino; **rs** = registro op1 e **rt** = registro op2;
- Da Tabela B.2 (pg. 622), o código de função de **add** é 32;
- Por ser do tipo-R o código de operação é 0 e, não sendo de *shift*, o **shamt** também é 0.

1 - tipo-R - Exercício Codificação (1) - add (2)

- Qual é o código máquina da seguinte instrução *Assembly*?
 - add** \$t0, \$s4, \$s5

Assembly Code

add \$t0, \$s4, \$s5

Field Values

| op | rs | rt | rd | shamt | funct |
|--------|--------|--------|--------|--------|--------|
| 0 | 20 | 21 | 8 | 0 | 32 |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

Tabela B.2 - (pg. 622)

| Funct | Name |
|-------------|-----------------|
| 100000 (32) | add rd, rs, rt |
| 100001 (33) | addu rd, rs, rt |
| 100010 (34) | sub rd, rs, rt |

Tabela 6.1 - (pg. 300)

rd = \$t0 = 8
rs = \$s4 = 20,
rt = \$s5 = 21

Machine Code

| op | rs | rt | rd | shamt | funct |
|--------|-------|-------|-------|-------|--------|
| 000000 | 10100 | 10101 | 01000 | 00000 | 100000 |
| 0 | 2 | 9 | 5 | 4 | 0 |

(0x02954020)

Convém escrever em binário primeiro e só depois em hexadecimal.

2 - Instruções do tipo-I(immediate) (1) - BitFields

| | | | |
|--------|--------|--------|------------------------------|
| op | rs | rt | imm16: constante ou endereço |
| 6 bits | 5 bits | 5 bits | 16 bits |

- Usadas em:
 - instruções Aritméticas/Lógicas Imediatas
 - Load/Store
- 2 registos:
 - rs: registo fonte (addi) ou endereço-base (eg, lw, sw)
 - rt: registo destino para algumas (eg, addi, lw)
- 1 constante (Imediato)
 - imm16: valor imediato de 16-bits (em 2C, exceto nas lógicas)
 - constante: -2^{15} a $+2^{15} - 1$ (mas nas lógicas: 0 a $2^{16} - 1$)
 - endereço: offset adicionado ao endereço-base em rs
- OPCODE
 - op: opcode (ou código de operação) é diferente de zero, e está presente em todas as instruções do tipo-I;

A operação a ser executada é inteiramente determinada pelo opcode (só!).

17

1 - tipo-I (4) - Tabela de Código de Operação (opcode)

| Opcode | Name | Description | Opcode | Name | Description |
|-------------|---------------------------------|--|-------------|------------------|--------------------------------|
| 000000 (0) | R-type | all R-type instructions | 011100 (28) | mul rd, rs, rt | multiply (32-bit result) |
| 000001 (1) | bltz rs, label / bgez rs, label | branch less than zero/branch greater than or equal to zero | 100000 (32) | lb rt, imm(rs) | load byte |
| 000010 (2) | j label | jump | 100001 (33) | lh rt, imm(rs) | load halfword |
| 000011 (3) | jal label | jump and link | 100011 (35) | lw rt, imm(rs) | load word |
| 000100 (4) | beq rs, rt, label | branch if equal | 100100 (36) | lbu rt, imm(rs) | load byte unsigned |
| 000101 (5) | bne rs, rt, label | branch if not equal | 100101 (37) | lhu rt, imm(rs) | load halfword unsigned |
| 000110 (6) | blez rs, label | branch if less than or equal to zero | 101000 (40) | sb rt, imm(rs) | store byte |
| 000111 (7) | bgtz rs, label | branch if greater than zero | 101001 (41) | sh rt, imm(rs) | store halfword |
| 001000 (8) | addi rt, rs, imm | add immediate | 101011 (43) | sw rt, imm(rs) | store word |
| 001001 (9) | addiu rt, rs, imm | add immediate unsigned | 110001 (49) | lwc1 ft, imm(rs) | load word to FP coprocessor 1 |
| 001010 (10) | slti rt, rs, imm | set less than immediate | 111001 (56) | swc1 ft, imm(rs) | store word to FP coprocessor 1 |
| 001011 (11) | sltiu rt, rs, imm | set less than immediate unsigned | | | |
| 001100 (12) | andi rt, rs, imm | and immediate | | | |
| 001101 (13) | ori rt, rs, imm | or immediate | | | |
| 001110 (14) | xori rt, rs, imm | xor immediate | | | |
| 001111 (15) | lui rt, imm | load upper immediate | | | |
| 010000 (16) | mfc0 rt, rd / mtc0 rt, rd | move from/to coprocessor 0 | | | |
| 010001 (17) | F-type | fop = 16/17: F-type instructions | | | |
| 010001 (17) | bcfl label / bclt label | fop = 8: branch if fpcond is FALSE/TRUE | | | |

Table B.1
Instructions sorted by opcode field.

Tipo-I: e.g., ADDI, LW, SW

Tipo-J: J, JAL

2 - tipo-I (2) - Exemplos: Aritmética_imm, lw e sw

Código Assembly

- `addi $s0, $s1, 5`
- `addi $t0, $s3, -12`
- `lw $t2, 32($0)`
- `sw $s1, 4($t1)`

Valor dos Campos

| op | rs | rt | imm |
|----|----|----|-----|
| 8 | 17 | 16 | 5 |
| 8 | 19 | 8 | -12 |
| 35 | 0 | 10 | 32 |
| 43 | 9 | 17 | 4 |

6 bits 5 bits 5 bits 16 bits

Código Máquina

| op | rs | rt | imm | |
|--------|-------|-------|---------------------|--------------|
| 001000 | 10001 | 10000 | 0000 0000 0000 0101 | (0x22300005) |
| 001000 | 10011 | 01000 | 1111 1111 1111 0100 | (0x2268FFF4) |
| 100011 | 00000 | 01010 | 0000 0000 0010 0000 | (0x8C0A0020) |
| 101011 | 01001 | 10001 | 0000 0000 0000 0100 | (0xAD310004) |

6 bits 5 bits 5 bits 16 bits

offset endereço-base

01/04/2025

PML – IAC - 2025

19

19

2 - tipo-I (3) - Exercício Codificação (1) - lw (1) - método

- Qual é o código máquina da seguinte instrução *Assembly*?
`lw $s3, -24($s4)`
- Sabemos que `lw` é uma instrução do tipo-I (2-registos);
 - Da Tabela B.1 (pg. 620), o código de operação para `lw` é 35;
- Da Tabela 6.1 (pg. 300), tiramos `$s3=rt=19` e `$s4=rs=20`, onde `rt`=registro destino; `rs` = registro (com o) endereço-base;
- O valor imediato `-24`, representa o *offset* (16-bits em 2C) a adicionar ao endereço-base (`rs`) para gerar o endereço efetivo.
 - Nota: Esta instrução lê uma palavra do endereço de memória “\$s4-24” e coloca-a no registro \$s3.

01/04/2025

PML – IAC - 2025

20

20

2 - tipo-I (3) - Exercício Codificação (1) - lw (1) - método

- Qual é o código máquina da seguinte instrução *Assembly*?

lw \$s3, -24(\$s4)

Assembly Code

Field Values

| op | rs | rt | imm |
|--------|--------|--------|---------|
| 35 | 20 | 19 | -24 |
| 6 bits | 5 bits | 5 bits | 16 bits |

Tabela B.1 - (pg. 620)

Tabela 6.1 - (pg. 300)

rs = \$s4 = 20

rt = \$s3 = 19

Valor imediato de 16-bits, em 2C :

24₁₀ = 0000 0000 0001 1000₂

-24₁₀ = 1111 1111 1110 1000₂ = FFE8₁₆

Machine Code

| op | rs | rt | imm |
|--------|-------|-------|---------------------|
| 100011 | 10100 | 10011 | 1111 1111 1110 1000 |
| 8 | E | 9 | 3 F F E 8 |

(0x8E93FFE8)

01/04/2025

21

21

3 - Instruções do tipo-J(ump)

| op | addr |
|--------|---------|
| 6 bits | 26 bits |

- Usadas nas instruções **j** e **jal**
- 1 único operando:
 - addr: endereço com 26-bits
- Outros campos:
 - op: o código de operação da instrução jump (op=2)

```
# MIPS assembly - j(ump)
addi $s0, $0, 4    # $s0 = 4
addi $s1, $0, 1    # $s1 = 1
j     target       # jump to target
sra   $s1, $s1, 2   # not executed
addi  $s1, $s1, 1   # not executed
target:
add   $s1, $s1, $s0 # $s1 = 1 + 4 = 5
```

próxima aula!

01/04/2025

PML – IAC - 2025

22

22

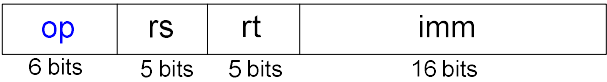
4 - Formato das Instruções do μP MIPS - Resumo

Tipo-R



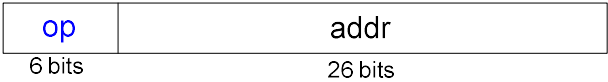
- 3 registos: operandos (rs, rt) e resultado (rd) ; opcode=0

Tipo-I



- 2 registos: operando (rs) e resultado/fonte (rt) ; operando (imm)

Tipo-J



- um único operando (addr)

Índice

- Linguagem Máquina
 - Introdução
- Codificação de Instruções
 - Tipo-R, tipo-I e tipo-J
 - Exemplos: Aritméticas (add) e de Load/Store (lw)
- Programa em Memória
 - Inicialização e Execução
- Decodificação de Instruções
 - Exemplos: tipo-R (sub) e tipo-I (addi)

5 - Programa em Memória

- Programa em memória
 - É um conjunto de instruções e dados para um CPU
 - É a diferença entre duas aplicações
- Vantagens do programa:
 - Não é necessário refazer ligações elétricas
 - Basta armazenar um novo programa na memória para alterar a funcionalidade da máquina.
- Como é feita a Execução do programa?
 - O CPU lê as instruções da memória sequencialmente
 - Cada instrução é decodificada e executada a operação correspondente.

25

5 - Programa em Memória - Início de Execução

Código Assembly

lw

\$t2, 32(\$0)

add

\$s0, \$s1, \$s2

addi

\$t0, \$s3, -12

sub

\$t0, \$t3, \$t5

Código Máquina

0x8C0A0020

0x02328020

0x2268FFF4

0x016D4022

1. O código Assembly é traduzido em código Máquina e carregado em memória.

2. O registo PC é inicializado para apontar para a 1ª instrução.

3. O CPU lê a instrução apontada pelo PC, decodifica-a e executa-a.

4. O valor do PC é incrementado para apontar para a instrução seguinte, i.e., $PC' = PC + 4$.

Programa em Memória

| Endereço | Instruções |
|----------|-----------------|
| ⋮ | ⋮ |
| 0040000C | 0 1 6 D 4 0 2 2 |
| 00400008 | 2 2 6 8 F F F 4 |
| 00400004 | 0 2 3 2 8 0 2 0 |
| 00400000 | 8 C 0 A 0 0 2 0 |
| ⋮ | ⋮ |

Main Memory

PC'

PC

01/04/2025

PML – IAC - 2025

26

26

Índice

- Linguagem Máquina
 - Introdução
- Codificação de Instruções
 - Tipo-R, tipo-I e tipo-J
 - Exemplos: Aritméticas (add) e de *Load/Store* (lw)
- Programa em Memória
 - Inicialização e Execução
- Descodificação de Instruções
 - Exemplos: tipo-R (**sub**) e tipo-I (**addi**)

01/04/2025

PML – IAC - 2025

27

27

Descodificação* de Instruções - método

- Começamos com **opcode** (6-bits mais significativos)
 - Se for igual a ZERO
 - Instrução tipo R -> 3 registos e funct
 - Caso contrário
 - Ver na tabela se o opcode corresponde a uma instrução tipo I ou tipo J
- Exemplo:
 - Converter o código máquina seguinte em instruções Assembly:
 - **0x02F34022**
 - **0x2237FFF1**

01/04/2025

PML – IAC - 2025

28

28

Descodificação - tipo-R - opcode

As instruções Assembly:

0x02F34022

0x2237FFF1

1. Consideremos a primeira instrução;

Verificamos que os 6bits mais significativos (MS) são iguais a zero; Logo é uma instrução do tipo-R.

1. O opcode (6-bits MS)

Machine Code

(0x02F34022)

op

rs

rt

rd

shamt

funct

000000

10111

10011

01000

00000

100010

0

2

F

3

4

0

2

2

Field Values

op

rs

rt

rd

shamt

funct

0

23

19

8

0

34

2. O funct (6-bits mS)

Sendo do tipo-R, analisamos em seguida o campo funct (6bits mS) para determinar qual a operação a ser executada. A tabela B.2 dá-nos que a operação 34 corresponde a sub.

3. O valores dos registos rs, rt e rd é-nos dado pela Tabela 6.1:

rs = 23 -> \$s7; rt = 19 -> \$s3; rd = 8 -> \$t0

01/04/2025

PML – IAC - 2025

sub \$t0, \$s7, \$s3

29

Descodificação - tipo-I

As instruções Assembly:

0x02F34022

0x2237FFF1

2. Consideremos a segunda instrução.

Os 6bits mais significativos são diferentes de zero. Logo não é uma instrução do tipo-R.

1. O opcode (6-bits MS)

Machine Code

(0x2237FFF1)

op

rs

rt

imm

001000

10001

10111

11111111110001

2

2

3

7

F

F

F

1

Field Values

op

rs

rt

imm

8

17

23

-15

Consultando a Tabela B.1, verificamos que a instrução com o opcode igual a 8 é addi.

2. O valores dos registos rs, rt é : rs = 17 -> \$s1; rt = 23 -> \$s7

3. O valor imediato (16 bits menos signif.) é uma constante em 2C:

0xFFF1₁₆ a que corresponde -0x000F₁₆ ou -15₁₀.

2C = Two's Complement .

addi \$s7, \$s1, -15

30

PML - 2025

15

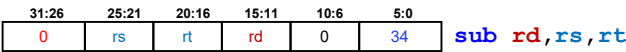
6 - Descodificação (4) - tipo-R e tipo-I : Resumo

As instruções *Assembly*:

0x02F34022
0x2237FFF1

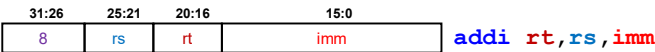
tipo-R

0x02F34022



tipo-I

0x2237FFF1



01/04/2025

PML – IAC - 2025

31

31

Indice - II

- Instruções Lógicas e de Deslocamento (*Shift*)
 - Lógicas AND, OR, XOR e NOR
 - Deslocamento (Shift): Lógico e Aritmético
- Constantes:
 - 16 e 32 bits
- Instruções de “Salto”
 - Condicional (**beq** e **bne**) e incondicional (**j** e **jr**)
- Controlo de fluxo de execução
 - Fluxo condicional: **if** e **if-else**
 - Ciclos iterativos: **for** e **while**

01/04/2025

PML – IAC - 2025

32

32

Índice - II

- Instruções Lógicas e de Deslocamento (*Shift*)
 - Lógicas AND, OR, XOR e NOR
 - Deslocamento (Shift): Lógico e Aritmético
- Constantes:
 - 16 e 32 bits
- Instruções de “Salto”
 - Condicional (**beq** e **bne**) e incondicional (**j** e **jr**)
- Controlo de fluxo de execução
 - Fluxo condicional: **if** e **if-else**
 - Ciclos iterativos: **for** e **while**

01/04/2025

PML – IAC - 2025

33

33

Instruções Lógicas: AND, OR, XOR e NOR

- **and, or, xor, nor** (tipo-R)
 - **and**: **mascara** bits
 - Ex: mascarar todos os bytes exceto o menos significativo duma word :
 $0xF234012F \text{ and } 0x000000FF = 0x0000002F$
 - **or**: **combina** *bitfields*
 - Ex: Combinar 0xF2340000 com 0x000012BC:
 - $0xF2340000 \text{ or } 0x000012BC = 0xF23412BC$
 - **xor e nor**: **invertem** bits:
 - Ex: Inverter todos os bits: $A \text{ nor } \$0 = \text{not } A$
- **andi, ori, xori** (tipo-I)
 - 16-bit imediato é *zero-extended*.
 - a instrução **nori** não existe.

01/04/2025

PML – IAC - 2025

34

34

Instruções Lógicas: Exemplo 1 - tipo-R

Source Registers

\$s1

111111111111000000000000

\$s2

010001101010000111100001011011

Assembly Code

and \$s3,\$s1,\$s2

\$s3

or \$s4,\$s1,\$s2

\$s4

xor \$s5,\$s1,\$s2

\$s5

nor \$s6,\$s1,\$s2

\$s6

Result

and: mascara bits

0xFFFF0000 and 0x46A1F0B7

= 0x46A10000

xor: inverte com '1', não-inverte com '0'

0xFFFF0000 xor 0x46A1F0B7

= 0xB95EF0B7

or: combina bits

0xFFFF0000 or 0x46A1F0B7

= 0xFFFFF0B7

nor: força a '0' com '1', inverte com '0'

0xFFFF0000 or 0x46A1F0B7

= 0x0000F48

01/04/2025

PML – IAC - 2025

35

35

Instruções Lógicas: Exemplo 2 - tipo-I

Source Values

\$s1

0000000000000000000011111111

imm

0000000000001111101000110100

zero-extended

Assembly Code

andi \$s2,\$s1,0xFA34

\$s2

ori \$s3,\$s1,0xFA34

\$s3

xori \$s4,\$s1,0xFA34

\$s4

Result

Nas instruções lógicas: o valor imediato de 16-bits é zero-extended (não sign-extended)

01/04/2025

PML – IAC - 2025

36

36

Instruções de Shift - Valor de 'shift' Constante

- **sll**: *shift left logical*
 - Desloca à esquerda e **preenche com zeros** os bits à direita
 - **sll** i bits = multiplicar por 2^i
 - **Exemplo**: **sll** \$t0, \$t1, 5 **# \$t0 := \$t1 << 5**
- **srl**: *shift right logical*
 - Desloca à direita e **preenche com zeros** os bits à esquerda
 - **srl** i bits = dividir por 2^i (operandos **unsigned**)
 - **Exemplo**: **srl** \$t0, \$t1, 5 **# \$t0 := \$t1 >>> 5**
- **sra**: *shift right arithmetic*
 - **Shift** à direita e **preenche com o bit de sinal** os bits à esquerda
 - **sra** i bits = dividir por 2^i (operandos **signed**)
 - **Exemplo**: **sra** \$t0, \$t1, 5 **# \$t0 := \$t1 >> 5**

01/04/2025

PML – IAC - 2025

37

37

Instruções de Shift (2) - Valor de 'shift' variável

- **sllv**: *shift left logical variable*
 - **Exemplo**: **sllv** \$t0, \$t1, \$t2 **# \$t0 := \$t1 << \$t2**
- **srlv**: *shift right logical variable*
 - **Exemplo**: **srlv** \$t0, \$t1, \$t2 **# \$t0 := \$t1 >>> \$t2**
- **srav**: *shift right arithmetic variable*
 - **Exemplo**: **srav** \$t0, \$t1, \$t2 **# \$t0 := \$t1 >> \$t2**

01/04/2025

PML – IAC - 2025

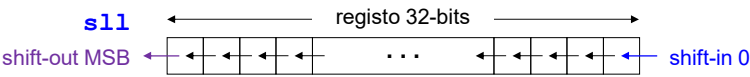
38

38

Shift - Shift Left Logical (SLL) (<<)

Deslocar **k** bits à esquerda equivale a multiplicar por 2^k

```
ori $t1,$0,4    # $t1 = 4
sll $t1,$t1,3    # $t1 = $t1*23 = $t1*8 = 4*8 = 32
```



```
$t1 = 0b0000 0000 ... 0000 0100 = 4
após sll $t1,$t1,3
$t1 = 0b0000 0000 ... 0010 0000 = 32 = (4 * 23)
```

Shift - Shift Right Logical (SRL) (>>>)

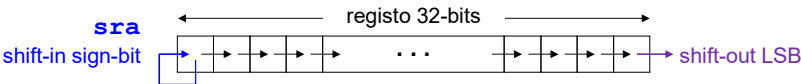
srl comporta-se como **sll** mas desloca para direita em vez de para a esquerda. Corresponde a dividir por 2^k mas só em UB (binário sem sinal).

Exemplo:

```
$t1 = 0b0000 0000 ... 0100 0000 = 0x0040 = 64
após srl $t1,$t1,2
$t1 = 0b0000 0000 ... 0001 0000 = 0x0010 = 16
      = 64*2-2 = 64/4
```

Shift - Shift Right Arithmetic (SRA) (>>) (1)

- **sra** também desloca para direita, mas preserva o bit de sinal.
- Deslocar e preservar o bit-de-sinal corresponde a dividir por 2^k em complemento para 2.



Exemplo: -127 (em 2C):
 $-127_{10} = 0b1\ 000\ 0001$

Dividindo-o por $8 = 2^3$ deveria dar $-127/8 = -15.875 \approx -16$

Possível usando **sra** e deslocando 3 bits para a direita?

*Em C/C++ o operador '>>' executa um *shift*-aritmético se a variável é um inteiro com sinal e um *shift*-lógico em inteiros sem sinal. Em Java é usado um operador distinto: '>>>' (srl).

Instruções de Shift (6) - Exemplos de Codificação

- **Codificação Shift:** mne RD, RT, Shamt

Assembly Code

Field Values

| | op | rs | rt | rd | shamt | funct |
|-------------------|--------|--------|--------|--------|--------|--------|
| sll \$t0, \$s1, 2 | 0 | 0 | 17 | 8 | 2 | 0 |
| srl \$s2, \$s1, 2 | 0 | 0 | 17 | 18 | 2 | 2 |
| sra \$s3, \$s1, 2 | 0 | 0 | 17 | 19 | 2 | 3 |
| | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

Machine Code

| op | rs | rt | rd | shamt | funct | |
|--------|--------|--------|--------|--------|--------|--------------|
| 000000 | 00000 | 10001 | 01000 | 00010 | 000000 | (0x00114080) |
| 000000 | 00000 | 10001 | 10010 | 00010 | 000010 | (0x00119082) |
| 000000 | 00000 | 10001 | 10011 | 00010 | 000011 | (0x00119883) |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits | |

Índice - II

- Instruções Lógicas e de Deslocamento (*Shift*)
 - Lógicas AND, OR, XOR e NOR
 - Deslocamento (Shift): Lógico e Aritmético
- Constantes:
 - 16 e 32 bits
- Instruções de “Salto”
 - Condicional (**beq** e **bne**) e incondicional (**j** e **jr**)
- Controlo de fluxo de execução
 - Fluxo condicional: **if** e **if-else**
 - Ciclos iterativos: **for** e **while**

01/04/2025

PML – IAC - 2025

43

43

Uso de constantes - 16Bits

C Code

```
// int is a 32-bit signed word
int a = 0x4f3c;
```

MIPS assembly code

```
# $s0 = a
addi $s0,$0,0x4f3c
# $s0 = 0x00004f3c
```

A instrução nativa **addi** é útil para carregar constantes com 16-bits num registo, quer sejam positivas quer sejam negativas!

Como o valor imediato da instrução **addi** tem sinal (2C), este é sempre estendido em sinal (*sign-extended*).

C Code

```
// int is a 32-bit signed word
int b = -0x8000; //-32768 (-215)
```

MIPS assembly code

```
# $s0 = b
addi $s0,$0,-32768
# $s0 = 0xffff8000
```

01/04/2025

PML – IAC - 2025

44

44

Uso de constantes (2) - 32Bits

- Constantes de 32-bits requerem duas instruções:

load upper immediate (**lui**) e **ori**:

C Code

```
int a = 0xFEDC8765;
```

MIPS assembly code

```
# $s0 = a
```

```
lui $s0, 0xFEDC # $s0 = 0xFEDC0000
```

```
ori $s0, $s0, 0x8765 # $s0 = 0xFEDC8765
```

- Quando o valor 'não cabe' em 16-bits (i.e., não é representável em 16-bits), é necessário usar duas instruções **lui** e **ori**;
- O MARS faz isso através da **pseudo-instrução li** (*load immediate*).

01/04/2025

PML – IAC - 2025

45

45

Índice - II

- Instruções Lógicas e de Deslocamento (*Shift*)
 - Lógicas AND, OR, XOR e NOR
 - Deslocamento (Shift): Lógico e Aritmético
- Constantes:
 - 16 e 32 bits
- Instruções de “Salto”
 - Condicional (**beq** e **bne**) e incondicional (**j** e **jr**)
- Controlo de fluxo de execução
 - Fluxo condicional: **if** e **if-else**
 - Ciclos iterativos: **for** e **while**

01/04/2025

PML – IAC - 2025

46

46

Instruções de 'Salto' (1) - Tipos

- Permitem a **execução** de código numa forma **não-sequencial**. (i.e., a instrução seguinte a ser executada não reside necessariamente no endereço de memória igual a PC + 4)
- Salto Condicional
 - Branch if equal (**beq**)
 - Branch if not equal (**bne**)
- Salto Incondicional
 - Jump (**j**)
 - Jump register (**jr**)
 - Jump and link (**jal**)

01/04/2025

PML – IAC - 2025

47

47

Instruções de 'Salto' Condicional (1) - **beq**

MIPS assembly - branch if equal

```

addi    $s0, $0, 4      # $s0 = 0 + 4 = 4
addi    $s1, $0, 1      # $s1 = 0 + 1 = 1
sll     $s1, $s1, 2      # $s1 = 1 << 2 = 4
beq    $s0, $s1, target # branch is taken
addi    $s1, $s1, 1      # not executed
sub     $s1, $s1, $s0     # not executed
target:
add     $s1, $s1, $s0     # $s1 = 4 + 4 = 8
  
```

Os **Labels** (etiquetas) indicam o **endereço** de memória da instrução. Não podem ser usadas palavras reservadas (e.g., uma instrução) e devem ter o sufixo ':' (dois pontos).

01/04/2025

PML – IAC - 2025

48

48

Instruções de 'Salto' Condicional (2) - bne

MIPS assembly - branch if not equal

```

addi    $s0, $0, 4      # $s0 = 0 + 4 = 4
addi    $s1, $0, 1      # $s1 = 0 + 1 = 1
sll     $s1, $s1, 2      # $s1 = 1 << 2 = 4
bne     $s0, $s1, target # branch not taken
addi    $s1, $s1, 1      # $s1 = 4 + 1 = 5
sub     $s1, $s1, $s0     # $s1 = 5 - 4 = 1
target:
add     $s1, $s1, $s0     # $s1 = 1 + 4 = 5

```

A codificação de beq e bne será vista mais tarde.

01/04/2025

PML - IAC - 2025

49

49

Instruções de 'Salto' Incondicional (1) - j

MIPS assembly - j (ump)

```

addi    $s0, $0, 4      # $s0 = 4
addi    $s1, $0, 1      # $s1 = 1
j       target          # jump to target
sra     $s1, $s1, 2      # not executed
addi    $s1, $s1, 1      # not executed
sub     $s1, $s1, $s0     # not executed
target:
add     $s1, $s1, $s0     # $s1 = 1 + 4 = 5

```

01/04/2025

PML - IAC - 2025

50

50

Instruções de 'Salto' Incondicional (2) - jr

MIPS assembly - j(ump) r(egister)

```
0x00002000      addi $s0, $0, 0x2010
0x00002004      jr   $s0
0x00002008      addi $s1, $0, 1
0x0000200C      sra  $s1, $s1, 2
0x00002010      lw   $s3, 44($s1)
```

jr é uma instrução do tipo-R.

01/04/2025

PML – IAC - 2025

51

51

Índice - II

- Instruções Lógicas e de Deslocamento (*Shift*)
 - Lógicas AND, OR, XOR e NOR
 - Deslocamento (Shift): Lógico e Aritmético
- Constantes:
 - 16 e 32 bits
- Instruções de “Salto”
 - Condicional (**beq** e **bne**) e incondicional (**j** e **jr**)
- Controlo de fluxo de execução
 - Fluxo condicional: **if** e **if-else**
 - Ciclos iterativos: **for** e **while**

01/04/2025

PML – IAC - 2025

52

52

Execução condicional - If - ASM

C Code

```
if(t0 == t1)
    f = 3;
f = f + 1;
```

C: A expressão condicional (f = 3) é executada se a condição lógica (t0 == t1) for verdadeira.

MIPS assembly code

```
# $s0 = f, # $t0 = t0, $t1 = t1
    bne $t0,$t1,nx # if (t0!=t1)
do: addi $s0,$0, 3 # f = 0+3
nx: addi $s0,$s0, 1 # f = f + 1
```

Asm: A condição lógica testada é a complementar (t0 != t1). Isto conduz a uma codificação mais eficiente (i.e., menos instruções Assembly).

Assembly tests opposite case (i != j) of high-level code (i == j)

Execução condicional - If – ASM Alternativo

C Code

```
if(t0 == t1)
    f = 3;
f = f + 1;
```

C: A expressão condicional (f = 3) é executada se a condição lógica (t0 == t1) for verdadeira.

MIPS assembly code

```
# $s0 = f, # $t0 = t0, $t1 = t1
    bne $t0,$t1,nx # if (t0!=t1)
do: addi $s0,$0, 3 # f = 0+3
nx: addi $s0,$s0, 1 # f = f + 1

# Alternativa menos eficiente
    beq $t0,$t1,do # if (i == j)
    j nx # +1 jump!
do: addi $s0,$0,3 # f = 3
nx: addi $s0,$s0,1 # f = f + 1
```

Usa mais um *j* no final do *if* para saltar o bloco *do*.

Execução condicional - If-else - ASM

| C Code | MIPS assembly code |
|---|---|
| <pre>if (t0 == t1) f = 3; else f = 2;</pre> | <pre># \$s0 = f, bne \$t0, \$t1, else addi \$s0, \$0, 3 #f = 3; j done else: addi \$s0, \$0, 2 #f = 2; done:</pre> |

Requer um *j* no final do *if* para saltar o bloco *else*.

Ciclos Iterativos - while

| C Code | MIPS assembly code |
|---|-----------------------------------|
| <pre>// determines the power of x // such that 2^x = 128 int pow = 1; int x = 0; while (pow != 128) { pow = pow * 2; x = x + 1; }</pre> | <pre># \$s0 = pow, \$s1 = x</pre> |

O código *Assembly* dos ciclos de repetição é semelhante ao código dos *if*'s com um *jump* para trás!

Conversão dum ciclo *while* num *if* com um salto para trás.

| | |
|--|---|
| <pre>while (i < j){ k++ ; i = i * 2 ; }</pre> | <pre>W_LP: if (i < j){ k++ ; i = i * 2 ; goto W_LP ; }</pre> |
|--|---|

Ciclos Iterativos - While

C Code

```
// determines the power of x
// such that 2^x = 128
int pow = 1;
int x = 0;
while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

MIPS assembly code

```
# $s0 = pow, $s1 = x
addi $s0, $0, 1 # pow=1
add $s1, $0, $0 # x=0
addi $t0, $0, 128
wh: beq $s0, $t0, done
sll $s0, $s0, 1 # pow*=2
addi $s1, $s1, 1 # x+=1
j wh
done:
```

Assembly tests for the opposite case (`pow == 128`) of the C code (`pow != 128`).

01/04/2025

PML - IAC - 2025

57

57

Ciclos Iterativos - For

```
for ( inicialização; condição; oper_iterativa ) {
    statement(s);
}
```

- *inicialização*: executada **antes** do *loop* começar
- *condição*: testada **no início** de cada iteração
- *statement(s)*: executado(s) sempre que a condição é satisfeita
- *oper_iterativa*: executada **no final** de cada iteração

O ciclo *for* é semelhante ao ciclo *while* com a vantagem de incluir uma variável de controlo do número de iterações.

01/04/2025

PML - IAC - 2025

58

58

Ciclos Iterativos - For

C Code

```
//add the numbers from 0 to 9
int sum = 0;
int i;

    1º      2º      4º
for (i=0; i!=10; i = i+1){
    sum = sum + i; 3º
}
```

MIPS assembly code

```
# $t0 = i, $t1 = sum
    addi $t1, $0, 0 #sum = 0
    1ºadd $t0, $0, $0 #i = 0
#
    addi $t2, $0, 10 #t2=10
for2ºbeq $t0, $t2, done
    3ºadd $t1, $t1, $t0
    4ºaddi $t0, $t0, 1 # i++
    j    for
done:
```

Indice - III

- 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; Carateres 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 Registos:
 - » Passagem de argumentos (\$a0-\$a3)
 - » Retorno de valor (\$v0)
 - » Efeitos colaterais

Comparação: Set on Less Than (**slt**)

- A instrução **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**.
- **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**).

01/04/2025

PML – IAC - 2025

61

61

Comparação: Set on Less Than (**slt**)

- **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

01/04/2025

PML – IAC - 2025

62

62

Comparação: Set on Less Than (slt)

C Code

```
// add the powers of 2
// from 1 to 10
int sum = 0;
int i;

for (i=1; i < 11; i = i*2){
    sum = sum + i;
}
```

MIPS assembly code

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

addi $s1,$0,0    # sum = 0
addi $s0,$0,1    # i = 1
addi $t0,$0,11   # $t0 = 11
# bge $s0,$t0,done # pseudo-instr.

loop: slt $t1,$s0,$t0 # $t1 = ($s0<$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 < 11

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

| | |
|-----------------------------|-------------------------|
| addiu \$9,\$0,0x0000000a | 2: main: li \$t1, 10 |
| addiu \$10,\$0,0x00000009 | 3: li \$t2, 9 |
| slt \$1,\$9,\$10 | 4: bge \$t1, \$t2, skip |
| beq \$1,\$0,0x00000001 | |
| addiu \$11,\$0,0xffffffff83 | 5: li \$t3, -125 |
| nop | 6: skip: nop |

01/04/2025PML – IAC - 202563

63

Comparação: Set on Less Than Immediate (slti) - tipo-I

C Code

```
// add the powers of 2
// from 1 to 10
int sum = 0;
int i;

for (i=1; i < 11; i = i*2){
    sum = sum + i;
}
```

MIPS assembly code

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

addi $s1, $0, 0
addi $s0, $0, 1
# addi $t0, $0, 11 # not needed

loop: slti $t1, $s0, 11 # $t1 = 1 if i < 11
      beq $t1, $0, done
      add $s1, $s1, $s0 # sum = sum + i
      sll $s0, $s0, 1   # i = i * 2
      j  loop

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?

01/04/2025PML – IAC - 202564

64

Indice - III

- 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; Carateres 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 Registos:
 - » Passagem de argumentos (\$a0-\$a3)
 - » Retorno de valor (\$v0)
 - » Efeitos colaterais

01/04/2025

PML – IAC - 2025

65

65

Array - Uso e Caraterísticas

- **Array**
 - É uma estrutura de dados usada para armazenar grandes quantidades de elementos do **mesmo tipo** (e.g., inteiro, caratere, etc).
 - Os elementos ocupam posições de **memória contíguas**.
- **Caraterísticas**
 - **Tamanho (Size: N):** número de elementos
 - **Índice (0..N-1):** para aceder a cada elemento (número de ordem do elemento no array)

01/04/2025

PML – IAC - 2025

66

66

Array - Acesso a elementos tipo Inteiro - Exemplo

- **Array*** com 5 elementos (tipo inteiro) em Memória
- `int array[5]; // C Code`
- **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)!

| 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)

Array - Exemplo: Código C e Acesso em ASM

- `// C Code`
`int array[5] = {-2, 4, 5, 123, -324};`
`array[0] = array[0] + 3;`
`array[1] = array[1] + 3;`
- **Procedimento de acesso (leitura/escrita)**
 - 1. Colocar o **endereço** do *array* num registo;
por exemplo `$s0` `la $s0, array`
 - 2. **Carregar** o valor do elemento `array[0]` noutro registo;
Por exemplo `$t1` `lw $t1, 0($s0)`
 - 3. **Neste caso**, somar 3 ao valor de `$t1`;
`addi $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 2 a 4 para o elemento `array[1]`, ajustando o **offset** de 0 para 4.
`lw $t1, 4($s0)`
`addi $t1, $t1, 3`
`sw $t1, 4($s0)`

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

Main Memory

Array - Exemplo: Acesso em ASM

```
// C Code
int array[5] ={-2, 4, 5, 123, -324};
array[0] = array[0] + 3;
array[1] = array[1] + 3;

# 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]
addi $t1, $t1, 3 # $t1 = $t1 + 3
sw $t1, 0($s0) # array[0] = $t1
# array[1]: byte offset = 4!
lw $t1, 4($s0) # $t1 = array[1]
addi $t1, $t1, 3 # $t1 = $t1 + 3
sw $t1, 4($s0) # array[1] = $t1
```

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

Main Memory

69

Array - Ciclo For em C

- A codificação anterior é *ineficiente* para *arrays* longos.
 - 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] + 3;
}
```

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

Main Memory

Endereço-Base = 0x23B8F000

70

Array - Ciclo For em ASM

```
# $s0 = base address, $s1 = i
# for (i=0; i < 1000; i++) {
# initialization code      # la $s0, 0x23B8F000
    lui  $s0, 0x23B8      # $s0 = 0x23B80000
    ori  $s0, $s0, 0xF000 # $s0 = 0x23B8F000
    addi $s1, $0, 0       # i = 0
    addi $t2, $0, 1000    # $t2 = 1000
loop:
    slt  $t0, $s1, $t2    # i < 1000?
    beq  $t0, $0, done    # if not then done
    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 = cópia de array[i]
    addi $t1, $t1, 3      # $t1 = array[i] + 3
    sw   $t1, 0($t0)      # array[i] = array[i] + 3
# next element
    addi $s1, $s1, 1      # i++
    j    loop            # } repeat
done:
01/04/2025
```

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

Main Memory

```
int array[1000]; // words
int i;
for (i=0; i < 1000; i++) {
    array[i] = array[i] + 3;
}
```

71

Carateres e Bytes - Código ASCII

- *American Standard Code for Information Interchange (ASCII)*
- Cada **caratere** (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)
- O standard ASCII (1963) veio uniformizar o **mapeamento** entre carateres (do alfabeto Inglês) e **bytes** para facilitar a transmissão de texto entre computadores.

01/04/2025

PML – IAC - 2025

72

72

Carateres 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 | Q | 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 | [| 6B | k | 7B | { |
| 2C | , | 3C | < | 4C | L | 5C | \ | 6C | l | 7C | |
| 2D | - | 3D | = | 4D | M | 5D |] | 6D | m | 7D | } |
| 2E | . | 3E | > | 4E | N | 5E | ^ | 6E | n | 7E | ~ |
| 2F | / | 3F | ? | 4F | O | 5F | _ | 6F | o | | |

01/04/2025

PML – IAC - 2025

73

73

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*

faz a extensão (do *byte*) para 32-bits com zeros

lb - *load byte*

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*.

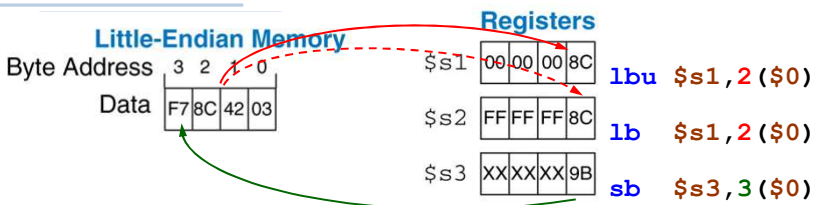
01/04/2025

PML – IAC - 2025

74

74

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).

01/04/2025

PML – IAC - 2025

75

75

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;
```
- Traduzir para Assembly
- 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).
- Vamos assumir que o registo `$s0` já está inicializado com o endereço do array `chararr`.

01/04/2025

PML – IAC - 2025

76

76

Exemplo lb/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?
    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:
```

Indice - III

- 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; Carateres 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 Registos:
 - » Passagem de argumentos (\$a0-\$a3)
 - » Retorno de valor (\$v0)
 - » Efeitos colaterais

Funções - Introdução

• 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.

01/04/2025

PML – IAC - 2025

79

79

Funções - Caller e Callee através de Exemplo

```
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 **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*).

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

01/04/2025

PML – IAC - 2025

80

80

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*.

01/04/2025

PML – IAC - 2025

81

81

Funções (4) - Instruções e 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*)

01/04/2025

PML – IAC - 2025

82

82

Funções (5) - Instruções MIPS: jal e jr

```
int main() {
    simple();
    a = b + c;
    return 0;
}

void simple() {
    return;
}
```

0x00400200 **main:** **jal** **simple**
 0x00400204 **add** \$s0, \$s1, \$s2
 ...
 # no return value!

0x00401020 **simple:** **jr** \$ra

jal **simple** : 'salta' (*jump*) para **simple** e 'liga' (*link*)
 \$ra = PC + 4 = 0x00400204

jr \$ra : 'salta' para o endereço contido em
 \$ra (0x00400204)
 (i.e., regressa ao ponto após a **jal**)
 \$ra - return address

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

01/04/2025

PML - IAC - 2025

83

83

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
}
```

01/04/2025

PML - IAC - 2025

84

84

Funções (7) - Argumentos e Retorno - Código ASM

\$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

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

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

85

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

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

86

Funções (9) - Salvaguarda de Registos - O Problema

```
# $s0 = result
diffosums:
    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
```

- `diffosums` alterou três registos: `$t0` , `$t1` e `$s0` !
- E se a função `main` também usar esses registos?
- `main` e/ou `diffosums` podem salvar temporariamente o conteúdo dos registos na *stack*, permitindo a respectiva reutilização.

Indice - IV

- Funções (continuação)
 - Stack
 - Definição
 - Salvaguarda de Registos
 - Função Terminal e não terminal
 - Recursividade
- Modos de Endereçamento
 - Tipo R: Só Registos
 - Tipo I: Imediato (addi, xori)
Endereço-Base (lw, sw)
PC-Relativo (beq, bne)
 - Tipo J: Pseudo-Direto (j, jal)

Funções – A pilha (Stack)

- Definição:
 - Zona de memória reutilizável, onde são guardadas variáveis temporárias.
 - Semelhante a uma pilha (de pratos, ou papéis): o ultimo a ser colocado é o primeiro a ser retirado (LIFO)
- Funcionamento
 - Expande:** Usa mais espaço de memória quando necessário.
 - Contraí:** Liberta o espaço de memória quando deixa de ser necessário.



Funções (11) - O Stack Pointer (\$sp=29)

- Expande** para baixo (dos endereços maiores para os menores) e contraí para cima.
- Stack Pointer:** **\$sp** aponta para o topo da *stack*.

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

Stack com **uma** só word:
0x12345678

Stack com **mais** duas words:
0xAABBCCDD e 0x11223344

Funções (12) - Salvaguarda de Registos (1)

- As funções que são invocadas não devem gerar efeitos colaterais adversos.
- Mas **diffofsums** altera o conteúdo de 3 registos:

\$t0, \$t1 e \$s0

```
# $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  # return value in $v0
    jr  $ra           # return to caller
```

- Esta alteração pode afectar a **main**!

01/04/2025

PML – IAC - 2025

91

91

Funções - Salvaguarda de Registos (2) - Soluções

- Problema:
 - As funções invocadas não devem prejudicar o bom funcionamento da função *caller*.
 - Mas **diffofsums** altera o conteúdo de 3 registos: **\$t0, \$t1 e \$s0**
- Três soluções possíveis:
 - 1. A *caller* (**main**) guarda na *stack* todos os registos, cujo conteúdo necessita de preservar, antes de invocar a função, i.e., salvaguarda **\$t0, \$t1 e \$s0**;
 - 2. A *callee* (**diffofsums**) guarda na *stack* todos os registos, cujo conteúdo *altera*, i.e., salvaguarda **\$t0, \$t1 e \$s0**;
 - 3. A salvaguarda de registos na *stack* é repartida entre a *caller* e a *callee* (opção usada no MIPS).

01/04/2025

PML – IAC - 2025

92

92

Funções - Salvaguarda de Registos (3) - MIPS

- A salvaguarda de registos na stack é repartida entre a *caller* e a *callee*. Como?
- A *caller* guarda na stack os registos \$*tx*, cujo conteúdo necessita preservar, antes de invocar a *callee*, e restaura-os após o retorno da *jal*.
- e...
- A *callee* guarda na stack os registos \$*sx*, cujo conteúdo altera, e restaura-os antes de retornar.

Funções - Salvaguarda de Registos (4) - MIPS

- A *caller* guarda na stack o conteúdo dos registos \$*tx* (i.e., só se voltar a precisar deles)



| Não-Preservados <i>Caller-Saved</i> | Preservados <i>Callee-Saved</i> |
|--|------------------------------------|
| \$ <i>t</i> 0-\$ <i>t</i> 9 | \$ <i>s</i> 0-\$ <i>s</i> 7 |
| \$ <i>a</i> 0-\$ <i>a</i> 3 | \$ <i>ra</i> |
| \$ <i>v</i> 0-\$ <i>v</i> 1 | \$ <i>sp</i> |



- A *callee* guarda na stack os registos \$*sx* que vai usar.

Funções - Salvag. Registos (5) - main (caller)

```
# A main precisa salvar os registros $t0 e $t1
main:                                # $s0 = y
...
addiu $sp, $sp, -8                  # make space on stack
sw     $t0, 4($sp)                  # save $t0
sw     $t1, 0($sp)                  # save $t1
addi   $a0, $0, 2                   # arg0 = 2
addi   $a1, $0, 3                   # arg1 = 3
addi   $a2, $0, 4                   # arg2 = 4
addi   $a3, $0, 5                   # arg3 = 5
jal    diffosums                    # call Function
lw     $t1, 0($sp)                  # restore $t1
lw     $t0, 4($sp)                  # restore $t0
addiu  $sp, $sp, 8                  # deallocate stack space
add    $s0, $v0, $0                 # y = returned value
...
```

Decrementa o \$sp de 2 words (8 bytes);
Guarda \$t0 em 4(\$sp) e \$t1 em 0(\$sp).

Assume-se que a main vai necessitar
de usar os registros \$t0 e \$t1, após a jal!

01/04/2025

PML - IAC - 2025

95

95

Funções - Salvag. Registos (6) - diffosums (callee)

```
# $s0 = result
# A convenção MIPS obriga a callee a preservar $s0
diffosums:
addiu  $sp, $sp, -4                # make space on stack
sw     $s0, 0($sp)                 # save $s0
...
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
lw     $s0, 0($sp)                 # restore $s0
addiu  $sp, $sp, 4                 # deallocate stack space
jr     $ra                         # return to caller
```

Decrementa o \$sp de 1 word (4 bytes);
Guarda \$s0 em 0(\$sp).

*Não é necessário guardar na stack \$t0 e \$t1 porque essa tarefa é da responsabilidade da caller!

01/04/2025

PML - IAC - 2025

96

96

Funções - Stack: durante a 'jal diffofsums'

| Address | Data |
|---------|------|
| FC | ? |
| F8 | |
| F4 | |
| F0 | |
| ⋮ | ⋮ |

a) **main**: antes da **jal**

| Address | Data |
|---------|------|
| FC | ? |
| F8 | \$t0 |
| F4 | \$t1 |
| F0 | |
| ⋮ | ⋮ |

b) **main**: no momento da **jal**

| Address | Data |
|---------|------|
| FC | ? |
| F8 | \$t0 |
| F4 | \$t1 |
| F0 | \$s0 |
| ⋮ | ⋮ |

c) **diffofsums**: durante a execução

b) **main**: decrementa o **\$sp** de 8 bytes (2 words); guarda **\$t0** em 4(**\$sp**) e **\$t1** em 0(**\$sp**).

```
addiu $sp, $sp, -8
sw    $t0, 4($sp)
sw    $t1, 0($sp)
```

c) **diffofsums**: decrementa o **\$sp** de 4 bytes; guarda **\$s0** em 0(**\$sp**).

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

97

Funções - Stack: após o restauro dos registos

| Address | Data |
|---------|------|
| FC | ? |
| F8 | \$t0 |
| F4 | \$t1 |
| F0 | \$s0 |
| ⋮ | ⋮ |

a) **diffofsums**: antes de **jr \$ra**

| Address | Data |
|---------|------|
| FC | ? |
| F8 | \$t0 |
| F4 | \$t1 |
| F0 | |
| ⋮ | ⋮ |

b) **main**: após a **jal** e o restauro dos registos

| Address | Data |
|---------|------|
| FC | ? |
| F8 | |
| F4 | |
| F0 | |
| ⋮ | ⋮ |

c) **main**: regresso ao estado anterior à **jal**

a) **diffofsums**: restaura **\$s0** de 0(**\$sp**) da stack; incrementa o **\$sp** de 4 bytes.

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

b) **main**: restaura **\$t0** de 4(**\$sp**) e **\$t1** de 0(**\$sp**); incrementa o **\$sp** de 8 bytes (2 words).

```
lw    $t1, 0($sp)
lw    $t0, 4($sp)
addiu $sp, $sp, 8
```

98

PML - 2025

49

Funções - Salvaguarda de Registos (7) - Solução

- **diffofsums** altera o valor dos registos **\$t0**, **\$t1** e **\$s0**, mas isso não interfere com o bom funcionamento da **main**, se usarmos a convenção anterior. Porquê?
 - **1. main**: garante que **\$t0** e **\$t1** preservam o valor após **diffofsums** ter sido invocada, guardando **\$t0** e **\$t1** na *stack* antes da *call* (**jal <>**) e restaurando-os após.
 - **2. diffofsums**: garante que o valor de **\$s0** é preservado, guardando-o na *stack* à entrada e restaurando-o à saída.

01/04/2025

PML – IAC - 2025

99

99

Funções - Invocação em Cadeia: proc1->proc2

- Quando uma função (**proc1**) invoca outra (**proc2**), tem de guardar na *stack* o registo **\$ra** para que **proc1** possa regressar ao código que a invocou (visto que a instrução **jal** usa implicitamente o registo **\$ra**).

proc1:

```
addiu $sp, $sp, -4 # make space on stack
sw    $ra, 0($sp) # save $ra
jal   proc2       # recall: jal changes $ra!
...
lw    $ra, 0($sp) # restore $ra
addiu $sp, $sp, 4 # deallocate stack space
jr    $ra         # return to caller
```

01/04/2025

PML – IAC - 2025

100

100

Funções - Função Terminal vs Não-Terminal

- **Função terminal vs Função não-terminal (*leaf* e *nonleaf*)**
 - Uma função que não invoca outras é designada por *terminal*, **diffosums** é um exemplo.
 - Uma função que invoca outras é designada por *não-terminal*, **main** é um exemplo.
- **Regras de Salvaguarda de Registos na *Stack* (de novo)**
 - 1. *caller* salvaguarda os registos que não são preservados pela convenção (\$t0-\$t9, \$a0-\$a3 e \$v0-\$t1), caso sejam necessários após a *call*.
 - 2. *callee* salvaguarda os registos que são preservados pela convenção (\$s0-\$s7, \$ra e \$sp), caso modifique o respetivo valor.

01/04/2025

PML – IAC - 2025

101

101

Funções - Recursivas (1) - Factorial C

- As implementações recursivas são em geral **mais compactas** (elegantes), embora sejam frequentemente **mais lentas** do que as implementações iterativas! Vão ser abordadas para ilustrar o funcionamento da *stack*.

```
int factorial(int n) {
    if (n <= 1) return 1;
    //else
    return n * factorial(n-1);
}
```

Recursiva

```
int factorial( int n ){
    int i, f = 1;
    for ( i = n ; i>1; i-- )
        f = i*f;
    return f;
}
```

Iterativa
(não-recursiva)

01/04/2025

PML – IAC - 2025

102

102

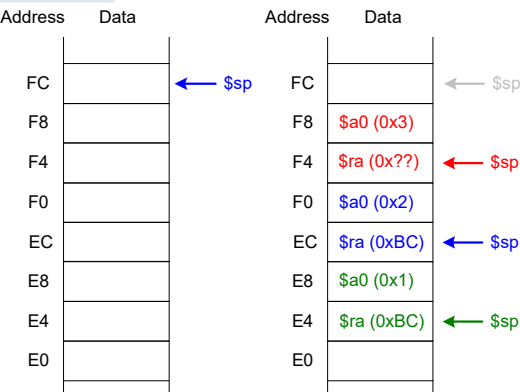
Funções - Recursivas (2) - Factorial ASM

```
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n-1);
}
```

A função factorial modifica os valores de \$ra e de \$a0, por isso salvaguarda os respectivos valores na stack.

```
0x90 factorial: addiu $sp, $sp, -8 # make room
0x94            sw  $a0, 4($sp) # store $a0 (n)
0x98            sw  $ra, 0($sp) # store $ra
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0 # n <= 1 ?
0xA4            beq  $t0, $0, else # no: go to else
0xA8            addi $v0, $0, 1 # yes: return 1
0xAC            addi $sp, $sp, 8 # restore $sp; $ra no chg!
0xB0            jr   $ra # return
#
0xB4 else:      addi $a0, $a0, -1 # n = n - 1
0xB8            jal  factorial # factorial(n-1)
0xBC            lw   $a0, 4($sp) # restore $a0 (n)
0xC0            mulu $v0, $a0, $v0 # n * factorial(n-1)
0xC4            lw   $ra, 0($sp) # restore $ra
0xC8            addiu $sp, $sp, 8 # restore $sp
0xCC            jr   $ra # return
```

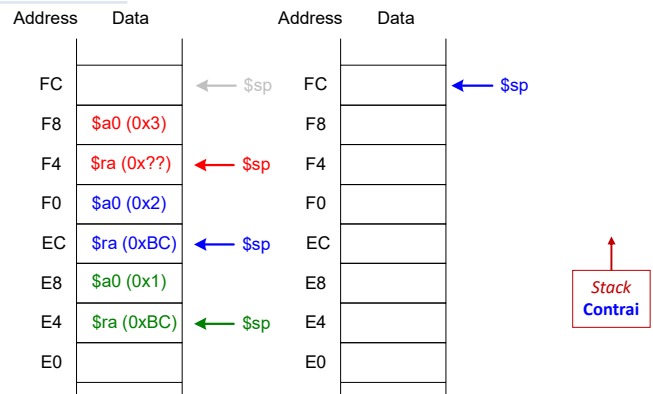
Funções - Recursivas (3): Stack 'jal factorial' - Inv.



Stack durante a **Invocação** da 'jal factorial' (c/ \$a0=3):

- A **1ª vez** cria uma *stack-frame* a **vermelho**, \$a0 = 3 e \$ra = 0x?? (main)
- A **2ª vez** cria uma *stack-frame* a **azul**, \$a0 = 2 e \$ra = 0xBC
- A **3ª vez** cria uma *stack-frame* a **verde**, \$a0 = 1 e \$ra = 0xBC

Funções (26) - Recursivas (4): Stack 'jal factorial' - Ret.

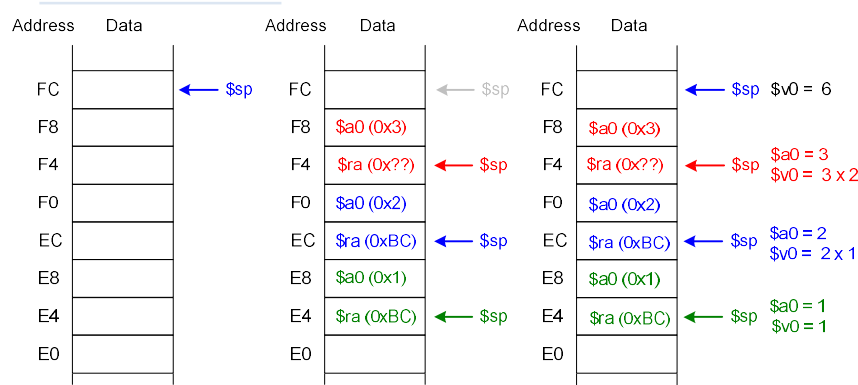


Stack antes do retorno através de 'jr \$ra':

- A 1ª vez liberta a stack-frame a verde, e retorna com \$ra = 0xBC
- A 2ª vez liberta a stack-frame a azul, e retorna com \$ra = 0xBC
- A 3ª vez liberta a stack-frame a vermelho, e retorna com \$ra = 0x??

01/04/2025 PML – IAC - 2025 105

Funções (27) - Recursivas (5): Stack 'jal factorial' - I e R



Stack durante a invocação e o retorno da 'jal factorial' (\$a0 = 3)

- A 1ª vez retorna com \$v0 = 1 para \$a0 = 1;
- A 2ª vez retorna com \$v0 = 2x1 = 2 para \$a0 = 2;
- A 3ª vez retorna com \$v0 = 3x2x1 = 6 para \$a0 = 3.

01/04/2025 PML – IAC - 2025 106

Funções - Recursivas (6) - factorial otimizada - I

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

A função **fact** modifica os valores de **\$ra** e de **\$a0**, por isso salvaguarda os respectivos valores na *stack*.

```
0x90 fact: addiu $sp, $sp, -8 # make room
0x94      sw  $a0, 4($sp) # store $a0
0x98      sw  $ra, 0($sp) # store $ra
0x9C      addi $v0, $0, 1 # f = 1
0xA0      addi $t0, $0, 2 #
0xA4      slt  $t0, $a0, $t0 #
0xA8      bne $t0, $0, fex # if(n < 2) return 1
#
0xAC      addi $a0, $a0, -1 # n = n - 1
0xB0      jal  fact # fact(n-1)
0xB4      lw   $a0, 4($sp) # restore $a0
0xB8      mulu $v0, $a0, $v0 # n * factorial(n-1)
0xBC      lw   $ra, 0($sp) # restore $ra
0xC0 fex: addiu $sp, $sp, 8 # restore $sp
0xC4      jr   $ra # return
```

Alternativa1: Mudando o beq em bne, obtemos uma redução de 16 para 14 instruções!

Funções - Recursivas (7) - factorial otimizada - II

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

A função **fact** modifica os valores de **\$ra** e de **\$a0**, por isso salvaguarda os respectivos valores na *stack*.

```
0x90 fact: addi $v0, $0, 1 # f = 1
0x94      slti $at, $a0, 2 #
0x98      bne $at, $0, fex # if(n < 2) return 1
#
0x9C      addiu $sp, $sp, -8 # make room
0xA0      sw  $ra, 0($sp) # save $ra
0xA4      sw  $a0, 4($sp) # save $a0
0xA8      addi $a0, $a0, -1 # n = n - 1
0xAC      jal  fact # fact(n-1)
0xB0      lw   $a0, 4($sp) # restore $a0
0xB4      mulu $v0, $a0, $v0 # n* fact(n-1)
0xB8      lw   $ra, 0($sp) # restore $ra
0xBC      addiu $sp, $sp, 8 # restore $sp
0xC0 fex: jr   $ra
```

Alternativa2: 1. A utilização de slti em vez de slt poupa mais uma instrução;
2. Não há necessidade de criar uma *stack frame* para n <= 1.

Funções - Convenção de Uso de Registos (Resumo)

- **Caller**
 - Guarda na *stack*, decrementando o **\$sp**, os registos a preservar (**\$t0-\$t9**, **\$a0-\$a3** e **\$v0-\$v1**)
 - Coloca os argumentos em **\$a0-\$a3**
 - 'Salta' para o código da função *invocada* (**jal <callee>**)
 - Utiliza o resultado devolvido em **\$v0**
 - Restaura o conteúdo dos registos e o valor do **\$sp**
- **Callee**
 - Guarda na *stack* os registos cujo conteúdo modifica (**\$ra**, **\$s0-\$s7**)
 - Usa os argumentos em **\$a0-\$a3**, executa a função e coloca o resultado em **\$v0**
 - Restaura o conteúdo dos registos e o valor do **\$sp**
 - Regressa ao código da *caller* (**jr \$ra**)

Indice - IV

- Funções (continuação)
 - Stack
 - Definição
 - Salvaguarda de Registos
 - Função Terminal e não terminal
 - Recursividade
- Modos de Endereçamento
 - Tipo R: Só Registos
 - Tipo I: Imediato (addi, xori)
Endereço-Base (lw, sw)
PC-Relativo (beq, bne)
 - Tipo J: Pseudo-Direto (j, jal)

Modos de Endereçamento* (1)

- Onde estão os operandos da instrução?
 - Todos em Registos
 - Registos e Imediato₁₆ (constante)
 - Endereço-Base (Registo) e Imediato₁₆
 - Relativo-ao-PC: $(PC4 + 4 * Imediato_{16})$
 - Pseudo-Direto: $(PC4_{31..28} : 4 * Imediato_{26})$
- *'Addressing Modes':* vão ser usados aquando da implementação do *Datapath* do CPU.

01/04/2025

PML – IAC - 2025

111

111

Endereçamento (2) - Register Only & Immediate

- 1. Só Registos (tipo-R)
 - Todos os operandos contidos em registos:


```
add $s0, $t2, $t3
sub $t8, $s1, $0
```
- 2. Imediato (tipo-I)
 - Valor imediato de 16-bits usado como operando:


```
addi $s4, $t5, -73
ori  $t3, $t7, 0xFF
```

– A extensão dos 16-bits para 32-bits, é *sign-extended* para **addi** mas *zero-extended* para **ori**!

01/04/2025

PML – IAC - 2025

112

112

Endereçamento (3) - Base Addressing

- 3. Endereço-Base (tipo-I)
 - O Endereço-efetivo do operando é dado por:
 - Endereço-Base + *Imediato16* (*sign-extended*)
- `lw $s4, 72($0)`
– Endereço-efetivo = $\$0 + 72$
- `sw $t2, -25($t1)`
– Endereço-efetivo = $\$t1 - 25$

Endereçamento (4) - PC-Relativo (Branches) - BTA

4. Relativo-ao-PC (tipo-I)

0x10

0x14 (PC+4)

0x18

0x1C

0x20 **else:**

0x24

`beq $t0, $0, else`

`addi $v0, $0, 1`

`addi $sp, $sp, i`

`jr $ra`

`addi $a0, $a0, -1`

`jal factorial`

Assembly Code

`beq $t0,$0,else`
`(beq $t0,$0,3)`

Field Values

| op | rs | rt | imm |
|--------|--------|--------|---------|
| 4 | 8 | 0 | 3 |
| 6 bits | 5 bits | 5 bits | 16 bits |

$BTA = (PC + 4) + imm \ll 2;$

Ex: $0x14 + 3 * 4 = 0x20$

BTA - *Branch Target Address*; O valor *imm* (3) representa o número de instruções.

Endereçamento (5) - PC-Relativo (Branches) - Imm=-1

4. Relativo-ao-PC (tipo-I) - Caso particular

```
0x10 stuck: beq $0, $0, stuck
0x14 (PC+4) addi $v0, $0, 1 # never reached!
```

Assembly Code

Field Values

| | | | |
|--------|--------|--------|---------|
| op | rs | rt | imm |
| 4 | 0 | 0 | -1 |
| 6 bits | 5 bits | 5 bits | 16 bits |

$BTA = (PC + 4) + imm \ll 2$; ex: $0x14 - 1 * 4 = 0x10$

BTA - Branch Target Address; O valor imm (-1) representa o número de instruções.

Endereçamento (6) - Pseudo-Direto (Jumps)

5. Pseudo-Direto (j e jal) - (tipo-J)

```
0x0040005C      jal sum
...
0x004000A0 sum:  add $v0, $a0, $a1
```

- Endereço de sum (quase) completo, está codificado na instrução:

| | | | |
|-------------|--------|------------------------------------|--------------|
| | op | addr | |
| Tipo-J | 000011 | 00 0001 0000 0000 0000 0010 1000 | (0x0C100028) |
| | 6 bits | 26 bits | |
| 26-bit addr | 0000 | 0000 0100 0000 0000 0000 1010 0000 | (0x0100028) |
| | | 0 1 0 0 0 0 2 8 | |
| JTA | 0000 | 0000 0100 0000 0000 0000 1010 0000 | (0x004000A0) |
| | | 0 0 4 0 0 0 A 0 | |

$JTA: (PC+4)_{31..28} : (Imm26 \ll 2)$ ('-' -> concatenação)

Ex: $0x0 : (0x010\ 0028) * 4 = 0x0040\ 00A0$

JTA - Jump Target Address. O endereço codificado na instrução está dividido por 4!

Endereçamento (6) - Resumo

1. Immediate addressing

| | | | |
|----|----|----|-----------|
| op | rs | rt | Immediate |
|----|----|----|-----------|

`addi $s4, $t5, -73`

2. Register addressing

| | | | | | |
|----|----|----|----|-----|-------|
| op | rs | rt | rd | ... | funct |
|----|----|----|----|-----|-------|

Registers

Register

`add $s0, $t2, $t3`

3. Base addressing

| | | | |
|----|----|----|---------|
| op | rs | rt | Address |
|----|----|----|---------|

Register

+

Memory

Byte Halfword Word

`sw $t2, -25($t1)`

4. PC-relative addressing

| | | | |
|----|----|----|---------|
| op | rs | rt | Address |
|----|----|----|---------|

PC

+

Memory

Word

`beq $t0, $t1, label`

5. Pseudodirect addressing

| | |
|----|---------|
| op | Address |
|----|---------|

PC

:

Memory

Word

`jal sum`

01/04/2025

PML – IAC - 2025

117

117

Índice V

– Ponteiro

- Definição
- Declaração e Inicialização:
 - Operadores '*' e '&'
- Acesso a Elementos
 - Operador '*': Leitura e Escrita
- Incrementar em C
 - Tipo-de-dados e Endereço
- Exs com índices vs ponteiros
 - Zerar (inteiros); toUpper (carateres)
- Exercício só com ponteiros
 - Soma (inteiros)

01/04/2025

PML – IAC - 2025

118

118

Ponteiro (1) - Definição

- **Ponteiro**
 - É uma variável (de determinado tipo) que contém o endereço de memória de outra variável.
- O **tipo-de-dados** apontado* pode ser:
 - `char`, `int`, `word`, `float`, `double`,
 - `array[]` (de `char`, `int`, `word`, etc)
 - ou uma `struct` mais complicada.
- **Ponteiros?**
 - A sua utilização gera código mais compacto e rápido.

01/04/2025

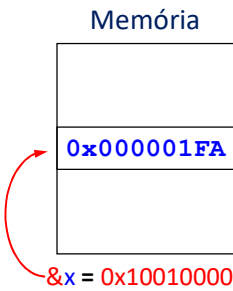
PML – IAC - 2025

119

119

Ponteiro (2) - Declaração e Inicialização - Em C

- Ex: Uma variável `x`, do tipo inteiro, tem o valor `0x1FA` e está localizada no endereço de memória (de dados) `0x10010000`.
- 1. A declaração do ponteiro `p_int`
 - É feita através do operador `'*'`:
 - `int*` `p_int`;
 - `p_int` aponta para uma variável do tipo `int`.
- 2. A inicialização do ponteiro `p_int`
 - É feita através do operador `'&'`:
 - `int` `x`;
 - `int*` `p_int` = `&x`;
 - `'&'` atribui a `p_int` o endereço da variável `x`.



01/04/2025

PML – IAC - 2025

120

120

Ponteiro (3) - Operador '*' - C : Acesso à variável x

1. e 2. Declaração e Inicialização

```
int x; // decl. de x
int* p_int = &x; // decl. e inicial. de p_int.
```

3. Leitura e Escrita com o ponteiro p_int

Leitura:

```
x = *p_int; // x := 0x1FA
```

Escrita:

```
*p_int = 0x1234; // x := 0x1234
```

- O operador '*' é usado para declarar o ponteiro e para aceder ao valor da variável apontada (tanto para a leitura como para a escrita).

01/04/2025

PML - IAC - 2025

121

121

Ponteiro (4) - C: Incrementar vs Tipo-de-dados (1)

- Ponteiro para char

```
char* p_char = 0x10010000; // inicializa o ponteiro;
p_char++; // ++ aponta para o char
// seguinte:
// p_char = 0x10010001
```

- Ponteiro para int

```
int* p_int = 0x10010000; // inicializa o ponteiro;
p_int++; // ++ aponta para o int
// seguinte:
// p_int = 0x10010004
```

A sintaxe é igual em ambos os casos!

01/04/2025

PML - IAC - 2025

122

122

Ponteiro (5) - C: Incrementar vs Tipo-de-dados (2)

- Ponteiro para `char`

```
char achars[3] = { 'i', 'a', 'c' };
char* p_char = achars;           // p_char = &achars[0]
p_char += 2;                     // p_char avança 2 bytes
                                // *p_char é 'c'
                                // 1 = sizeof( char )
```

- Ponteiro para `int`

```
int aints[3] = { 1234, -432, 47 };
int* p_int = aints;              // p_int = &aints[0]
p_int += 2;                      // p_int avança 2x4 bytes
                                // *p_int é 47
                                // 4 = sizeof( int )
```

- Em `C` a sintaxe é igual nos dois casos.
- Em `ASM` são tratados *distintamente*, i.e., o *ponteiro* é incrementado em múltiplos do *tamanho-em-bytes* da variável apontada.

01/04/2025

PML - IAC - 2025

123

123

Ponteiro (6) - De C para ASM: Inicial., Leitura e Escrita

- A variável `x` do tipo inteiro tem o valor `0x1FA` e está localizada no endereço `0x10010000`.

Em ASM, suponhamos: `p` -> `$a0` e `x` -> `$s0`

- 1. Inicialização do ponteiro

```
p = &x;           // p gets 0x10010000
la $a0, 0x10010000 # p = 0x10010000
```

- 2. Leitura do valor da variável apontada por `p`

```
x = *p;           // x gets 0x01fA
lw $s0, 0($a0)     # dereferencing p
```

- 3. Escrita de novo valor na variável apontada por `p`

```
*p = 0x1234;       // x gets 0x1234
addi $t0, $0, 0x1234
sw $t0, 0($a0)     # dereferencing p
```

01/04/2025

PML - IAC - 2025

124

124

Arrays: Índices vs Ponteiros (3) - Ex2: 'Zerar' um Array

| Índice | Ponteiro |
|--|--|
| <pre>void clear_i(int array[], int size) { int i=0; do { array[i] = 0; // clear i++; // inc. index } while (i < size); }</pre> | <pre>void clear_p(int *array, int size) { int *p = &array[0]; //or p = array do { *p = 0; // clear p++; // inc. pointer } while (p < &array[size]); }</pre> |
| <pre>move \$t0,\$0 # i = 0 dw1: sll \$t1,\$t0,2 # \$t1 = i * 4 addu \$t2,\$a0,\$t1 # \$t2 = # &array[i] sw \$0, 0(\$t2) # array[i] = 0 addi \$t0,\$t0,1 # i++ slt \$t3,\$t0,\$a1 # \$t3 = # (i < size) bne \$t3,\$0,dw1 # if(i < size) # goto dw1</pre> | <pre>move \$t0,\$a0 # p = &array[0] sll \$t1,\$a1,2 # \$t1 = size*4 addu \$t2,\$a0,\$t1 # \$t2 = # &array[size] dw2: sw \$0,0(\$t0) # *p = 0 addiu \$t0,\$t0,4 # p++ sltu \$t3,\$t0,\$t2 # \$t3 = (p< # &array[size]) bne \$t3,\$0,dw2 # if (...) # goto dw2</pre> |
| <pre>loop dw1: 6 instruções &array[i]: 2 adições + 1 sll</pre> | <pre>loop dw2: 4 instruções p = &array[i]: 1 adição</pre> |

Nota: \$a0 e \$a1 são os argumentos das funções clear_i(...) e clear_p(...) !

01/04/2025

PML – IAC - 2025

127

127

Arrays: Idxs vs Ptrs (5) - Ex3: toUpper - C

- Conversão duma *string* (array de caracteres) em maiúsculas; Acesso aos elementos do array usando índices vs ponteiros.

```
char str[] = "Arrays: Indexes vs Pointers";
```

```
// Indexes
void toUpperI( char str[] ){
    int i = 0;
    while ( str[i] != '\0' ) {
        if ( (str[i] >= 'a') && (str[i] <= 'z') )
            str[i] = str[i] - 32;
        i++;
    }
}
```

```
// Pointers
void toUpperP( char* str ){
    char *p = str;
    while ( *p != '\0' ) {
        if ( ( *p >= 'a' ) && ( *p <= 'z' ) )
            *p = *p - 32;
        p++;
    }
}
```

1. O argumento `str[]` é um *array* de `char`.

2. O operador `'&&'` é o operador AND-Booleano (diferente do operador `'&'` AND-Bitwise).

1. O argumento `str` é um *ponteiro* para `char`.

2. O operador `'*'` desreferencia o ponteiro `'p'`, isto é, acede ao valor da variável apontada por `'p'`, tanto para leitura como para escrita.

01/04/2025

PML – IAC - 2025

128

128

PML - 2025

64

Arrays: Idxs vs Ptrs (6) - toUpperI - ASM Índices

- Com *índices*, cada iteração tem de calcular o endereço de `str[i]`, o que requiere duas somas.

```
# void toUpperI( char str[] );
toUpperI: li    $t0, 0
lpi:      addu  $t1, $t0, $a0
          lb    $t2, 0($t1)
          beq   $t2, $0, donei
          blt   $t2, 'a', nexti
          bgt   $t2, 'z', nexti
          addi  $t2, $t2, -32
          sb    $t2, 0($t1)
nexti:    addi  $t0, $t0, 1
          j     lpi
donei:    jr    $ra

# $a0 = str
# $t0 = i = 0
# $t1 = &str[i]
# $t2 = str[i]
# $t2 = 0? (''\0')
# $t2 < 'a'?
# $t2 > 'z'?
# convert
# and store back
# i++
```

```
//Indexes
void toUpperI( char str[] ){
    int i = 0;
    while ( str[i] != 0 ) {
        if ( (str[i] >= 'a') && (str[i] <= 'z') )
            str[i] = str[i] - 32;
        i++;
    }
}
```

Num array de words o cálculo do endereço de `str[i]` exigiria ainda uma multiplicação por 4 (slide 9).

01/04/2025

PML - IAC - 2025

129

129

Arrays: Idxs vs Ptrs (7) - toUpperP - ASM Ponteiros

- Com *ponteiros*, usamos um registo com o endereço exato do elemento corrente. Em cada iteração incrementamos esse registo para apontar para o elemento seguinte.

```
# void toUpperP(char* str);
toUpperP: lb    $t2, 0($a0)
          beq   $t2, $0, donep
          blt   $t2, 'a', nextp
          bgt   $t2, 'z', nextp
          addi  $t2, $t2, -32
          sb    $t2, 0($a0)
nextp:    addiu  $a0, $a0, 1
          j     toUpperP
donep:    jr    $ra

# $a0 = p = str;
# $t2 = *p
# $t2 == '\0' ?
# $t2 < 'a'?
# $t2 > 'z'?
# convert
# and store back
# p++; changes $a0!
```

```
//Pointers
void toUpperP( char* str ){
    char *p = str;
    while (*p != 0) {
        if ( (*p >= 'a') && (*p <= 'z') )
            *p = *p - 32;
        p++;
    }
}
```

Num array de words teríamos de incrementar o ponteiro por 4. De qq modo, precisaríamos de uma só adição em vez de duas adições e uma multiplicação por 4 (*sll*) (ver slide 9).

01/04/2025

PML - IAC - 2025

130

130

Arrays: Idxs vs Ptrs (8) - toUpper - ASM Idx vs Ptr

Índice vs Ponteiro :

Índice

```
# void toUpperI( char str[] ); # $a0 = str
toUpperI: li $t0, 0 # $t0 = i
lpi: add $t1, $t0, $a0 # $t1 = &str[i]
lb $t2, 0($t1) # $t2 = str[i]
beq $t2, $0, donei # $t2 = 0?
blt $t2, 'a', nexti # $t2 < 'a'?
bgt $t2, 'z', nexti # $t2 > 'z'?
addi $t2, $t2, -32 # convert
sb $t2, 0($t1) # and store back
nexti: addi $t0, $t0, 1 # i++
j lpi
donei: jr $ra
```

Ponteiro

```
# void toUpperP(char* str); # $a0 = str;
toUpperP: lb $t2, 0($a0) # $t2 = *s
beq $t2, $0, donep # $t2 = 0?
blt $t2, 'a', nextp # $t2 < 'a'?
bgt $t2, 'z', nextp # $t2 > 'z'?
addi $t2, $t2, -32 # convert
sb $t2, 0($a0) # and store back
nextp: addiu $a0, $a0, 1 # p++;changes
$a0!
j toUpperP
donep: jr $ra
```

01/04/2025

PML – IAC - 2025

131

131

Arrays: Idxs vs Ptrs (9) - Ex4: Soma Ptrs - C

- Soma dos elementos dum array (com ponteiros)

```
#define SIZE 4
void main (void) {
    // Declara um array estático de 4 inteiros e inicializa-o
    static int aints[ SIZE ] = { 7692, 23, 5, 234 };
    int *p = &aints[0]; //declara um ponteiro para inteiro
                        // 'p' é inicializado com &aints[0]
    int * pultimo = &aints[ SIZE-1 ]; // "pultimo" é
    inicializado com &aints[3]

    int soma = 0; // soma=0
    while( p <= pultimo ) {
        soma += *p ; // acumula o valor em soma
        p++; // incrementa o ponteiro
    }
    print_int10 ( soma ); // imprime a soma
}
```

01/04/2025

PML – IAC - 2025

132

132

Arrays: Idxs vs Ptrs (10) - Ex4: Soma Ptrs - ASM

```
#define SIZE 4
void main (void) {
    // Declara um array ...
    static
    int aints[SIZE] = {7692,23,5,234};
    // Ponteiros
    int *p = &aints[0];
    int *pultimo = &aints[SIZE-1];
    // soma=0
    int soma = 0;
    while( p <= pultimo ) {
        soma += *p ;
        p++;
    }
    print_int10(soma);
}
```

```
.eqv print_int10,1
.eqv exit,10
.eqv SIZE3,12 # 3*4
.data
aints: .word 7692,23,5,234 # int aints[]={...}
.text
.globl main
# -----
# $t0 = p ; $t1 = pultimo;
# $t2 = *p ; $t3 = soma
# -----
main: la $t0, aints # $t0 = p = aints
# pultimo = aints + (NSIZE-1)*sizeof(int)
addiu $t1, $t0, SIZE3 # $t1 = aints + 3*4
li $t3, 0 # soma = 0
# if( p > pultimo ) ewh
wh: bgtu $t0, $t1, ewh #
lw $t2, 0($t0) # $t2 = *p
add $t3, $t3, $t2 # soma += *p
addiu $t0, $t0, 4 # p++
j wh
ewh: move $a0, $t3 # print sum
li $v0, print_int10
syscall
li $v0, exit # exit
syscall
# soma: 7954
```

01/04/2025

PML – IAC - 2025

133

133

Instruções Signed/Unsigned (1) - add, addi e sub

- Adição e Subtração
- Multiplicação e Divisão
- Comparação: *Set Less Than*
- **Signed (com sinal):** **add, addi, sub**
 - Mesma operação que as versões *unsigned*
 - O CPU gera exceção de *overflow*
- **Unsigned (sem sinal):** **addu, addiu, subu**
 - Não gera exceção de *overflow*

NOTA: **addiu** - sign-extends the immediate

134

Instruções Signed/Unsigned (2) - mul, div e slt

- Multiplicação e Divisão
 - Signed: **mult, div**
 - Unsigned: **multu, divu**
- Comparação: Set Less Than
 - Signed: **slt, slti**
 - Unsigned: **sltu, sltiu**
- **sltiu** - *also sign-extends the immediate before comparing it to the register.*

01/04/2025

PML – IAC - 2025

135

135

Instruções Signed/Unsigned (3) - lb e lh

- Signed
 - **Sign-extends** to create a 32-bit value (to load into register)
 - Load byte: **lb**
 - Load halfword: **lh**
- Unsigned (sem sinal)
 - **Zero-extends** to create a 32-bit value (to load into register)
 - Load byte unsigned: **lbu**
 - Load halfword unsigned: **lhu**

01/04/2025

PML – IAC - 2025

136

136

Instruções Signed/Unsigned (3) - lb e lh

- **Signed**

- **Sign-extends** to create a 32-bit value (to load into register)
- Load byte: **lb**
- Load halfword: **lh**

- **Unsigned (sem sinal)**

- **Zero-extends** to create a 32-bit value (to load into register)
- Load byte unsigned: **lbu**
- Load halfword unsigned: **lhu**

01/04/2025

PML – IAC - 2025

137

137