

# Arquitectura de Sistemas e Computadores I

## Semana 4

Miguel Barão

mjsb@di.uevora.pt

## Resumo

- Instruções de salto incondicional (j, jal, jr)
- Instruções de load e store (lw, sw, lb, sb, lh, sh)
- Acesso a arrays e estruturas de dados em memória
- Directivas para o assembler (.text, .data, .word, .ascii, etc)

j LABEL    # Salta para LABEL

j LABEL	# Salta para LABEL
jr \$t0	# Salta para endereço que está em \$t0

```
j LABEL      # Salta para LABEL
jr $t0       # Salta para endereço que está em $t0
jal LABEL    # Jump and Link: Igual ao j LABEL,
              # mas guarda o PC no registo $ra.
              # O PC guardado é o endereço de 2 instruções à frente.
```

```
j LABEL      # Salta para LABEL
jr $t0       # Salta para endereço que está em $t0
jal LABEL    # Jump and Link: Igual ao j LABEL,
              # mas guarda o PC no registo $ra.
              # O PC guardado é o endereço de 2 instruções à frente.
```

## Delayed branching

A instrução seguinte ao jump é sempre executada!

Qual o efeito do código seguinte?

```
lui $t0, 0x0040  
ori $t0, $t0, 0x0000
```

```
jr $t0  
nop
```

As instruções `jal` e `jr` são muitas vezes usadas em combinação:

Qual o efeito do código seguinte?

```
...  
jal xpto  
nop  
...
```

# e mais a frente temos:

```
xpto:  
...  
jr $ra  
nop
```



## Instruções de salto incondicional: exemplo

As instruções jal e jr são usadas para implementar chamadas e retorno de funções:

```
...
jal xpto          # chama xpto e guarda no registo
nop              # $ra o endereço da instrução add
add $t0, $t1, $zero    <- endereço de retorno
...
jal xpto          # chama xpto e guarda no registo
nop              # $ra o endereço da instrução ori
ori $t1, $zero, 2      <- endereço de retorno
...

# Função xpto:
xpto:
...
jr $ra           # retorna ao ponto de chamada
nop
```

Data
...
Text (code)
...

## Carregar dados em memória: Words

Endereço inicial → 0x10010000

⋮
0xffffffffb3 (-333)
0x00000000 ( 0 )
0x000003e8 (1000)
0xfffffffffc ( -4 )
0x00000002 ( 2 )
0x00000001 ( 1 )
⋮

Directiva para o assembler:

```
.data  
.word 1,2,-4,1000,0,-333
```

	⋮
0x10010014	0xffffffffeb3 (-333)
0x10010010	0x00000000 ( 0 )
0x1001000c	0x000003e8 (1000)
0x10010008	0xfffffffffc (-4 )
0x10010004	0x00000002 ( 2 )
Endereço inicial → 0x10010000	0x00000001 ( 1 )
	⋮

Directiva para o assembler:

```
.data  
.word 1,2,-4,1000,0,-333
```

- Estamos a ver a memória organizada em words (endereço múltiplos de 4).

	⋮
0x10010014	0xffffffffb3 (-333)
0x10010010	0x00000000 ( 0 )
0x1001000c	0x000003e8 (1000)
0x10010008	0xfffffffffc (-4 )
0x10010004	0x00000002 ( 2 )
Endereço inicial → 0x10010000	0x00000001 ( 1 )
	⋮

Directiva para o assembler:

```
.data  
.word 1,2,-4,1000,0,-333
```

- Estamos a ver a memória organizada em words (endereço múltiplos de 4).
- Atenção que não conhecemos a ordenação de bytes usada pelo processador!

Endereço inicial → 0x10010000	⋮
	0x80 (-128)
	0x7f ( 127)
	⋮

Directiva para o assembler:

```
.data  
.byte 127, -128
```

	0x10010001	⋮
Endereço inicial →	0x10010000	0x80 (-128)
		0x7f ( 127)
		⋮

Directiva para o assembler:

```
.data  
.byte 127, -128
```

- Estamos a ver a memória organizada em bytes.
- Não depende da ordenação de bytes (endianness)

- Uma **string** é um **array de caracteres** em memória.
- Cada carácter é codificado por um código (e.g. ASCII).
- A string termina num carácter especial: o **carácter NUL**.

Tabela ASCII

	hex		hex		hex		hex		hex		hex		hex
NUL	00	DLE	10		20	0	30	@	40	P	50	'	60
SOH	01	DC1	11	!	21	1	31	A	41	Q	51	a	61
STX	02	DC2	12	"	22	2	32	B	42	R	52	b	62
ETX	03	DC3	13	#	23	3	33	C	43	S	53	c	63
EOT	04	DC4	14	\$	24	4	34	D	44	T	54	d	64
ENQ	05	NAK	15	%	25	5	35	E	45	U	55	e	65
ACK	06	SYN	16	&	26	6	36	F	46	V	56	f	66
BEL	07	ETB	17	'	27	7	37	G	47	W	57	g	67
BS	08	CAN	18	(	28	8	38	H	48	X	58	h	68
TAB	09	EM	19	)	29	9	39	I	49	Y	59	i	69
LF	0a	SUB	1a	*	2a	:	3a	J	4a	Z	5a	j	6a
VT	0b	ESC	1b	+	2b	;	3b	K	4b	[	5b	k	6b
FF	0c	FS	1c	,	2c	<	3c	L	4c	\	5c	l	6c
CR	0d	GS	1d	-	2d	=	3d	M	4d	]	5d	m	6d
SO	0e	RS	1e	.	2e	>	3e	N	4e	^	5e	n	6e
SI	0f	US	1f	/	2f	?	3f	O	4f	_	5f	o	6f
													DEL
													7f



```
.data  
.asciiz "Hello World!"
```

		⋮	
0x1001000c	0x00 ('\\0')		← Carácter nulo
0x1001000b	0x21 ('!')		
0x1001000a	0x64 ('d')		
0x10010009	0x6c ('l')		
0x10010008	0x72 ('r')		
0x10010007	0x6f ('o')		
0x10010006	0x57 ('W')		
0x10010005	0x20 (' ')		
0x10010004	0x6f ('o')		
0x10010003	0x6c ('l')		
0x10010002	0x6c ('l')		
0x10010001	0x65 ('e')		
Endereço inicial → 0x10010000	0x48 ('H')		
		⋮	

Permitem transferir dados de e para a memória.

```
lw $t0, 4($t1)    # Load Word  
                  # t0=mem[t1+4] isto é, acede a memória  
                  # ao endereço calculado com t1+4 e  
                  # lê uma word (4 bytes) para o registo t0.
```

Permitem transferir dados de e para a memória.

```
lw $t0, 4($t1)    # Load Word  
                  # t0=mem[t1+4] isto é, acede a memória  
                  # ao endereço calculado com t1+4 e  
                  # lê uma word (4 bytes) para o registo t0.
```

```
sw $t0, -8($t1)    # Store Word  
                  # mem[t1-8]=t0 isto é, acede a memória  
                  # ao endereço calculado com t1-8 e  
                  # guarda a word (4 bytes) do registo t0.
```

Permitem transferir dados de e para a memória.

```
lw $t0, 4($t1)    # Load Word
                   # t0=mem[t1+4] isto é, acede a memória
                   # ao endereço calculado com t1+4 e
                   # le uma word (4 bytes) para o registo t0.
```

```
sw $t0, -8($t1)    # Store Word
                   # mem[t1-8]=t0 isto é, acede a memória
                   # ao endereço calculado com t1-8 e
                   # guarda a word (4 bytes) do registo t0.
```

Restrições:

- Respeita o *endianness* (ordenação dos bytes em memória)

Permitem transferir dados de e para a memória.

```
lw $t0, 4($t1)    # Load Word
                   # t0=mem[t1+4] isto é, acede a memória
                   # ao endereço calculado com t1+4 e
                   # le uma word (4 bytes) para o registo t0.
```

```
sw $t0, -8($t1)    # Store Word
                   # mem[t1-8]=t0 isto é, acede a memória
                   # ao endereço calculado com t1-8 e
                   # guarda a word (4 bytes) do registo t0.
```

Restrições:

- Respeita o *endianness* (ordenação dos bytes em memória)
- Os endereços têm de ser múltiplos de 4 (*word aligned*)

Permitem transferir dados de e para a memória.

```
lw $t0, 4($t1)    # Load Word
                  # t0=mem[t1+4] isto é, acede a memória
                  # ao endereço calculado com t1+4 e
                  # le uma word (4 bytes) para o registo t0.
```

```
sw $t0, -8($t1)    # Store Word
                  # mem[t1-8]=t0 isto é, acede a memória
                  # ao endereço calculado com t1-8 e
                  # guarda a word (4 bytes) do registo t0.
```

Restrições:

- Respeita o *endianness* (ordenação dos bytes em memória)
- Os endereços têm de ser múltiplos de 4 (*word aligned*)
- O *offset* é um número de 16 bits com sinal.

desenho...

ver quadro de giz →

```
lb $t0, 1($t1)    # Load Byte  
                  # t0=mem[t1+1] isto é, acede a memória  
                  # ao endereço calculado com t1+1 e  
                  # le um byte para o registo t0 (sign extended).
```



lb \$t0, 1(\$t1)    # Load Byte  
# t0=mem[t1+1] isto é, acede a memória  
# ao endereço calculado com t1+1 e  
# le um byte para o registo t0 (sign extended).

sb \$t0, -3(\$t1)    # Store Byte  
# mem[t1-3]=t0 isto é, acede a memória  
# ao endereço calculado com t1-3 e  
# guarda o LSB do registo t0.  
# (LSB = Least Significant Byte)

lb \$t0, 1(\$t1)    # Load Byte  
# t0=mem[t1+1] isto é, acede a memória  
# ao endereço calculado com t1+1 e  
# le um byte para o registo t0 (sign extended).

sb \$t0, -3(\$t1)    # Store Byte  
# mem[t1-3]=t0 isto é, acede a memória  
# ao endereço calculado com t1-3 e  
# guarda o LSB do registo t0.  
# (LSB = Least Significant Byte)

- Não tem restrições de alinhamento dos endereços.

lb \$t0, 1(\$t1)    # Load Byte  
# t0=mem[t1+1] isto é, acede a memória  
# ao endereço calculado com t1+1 e  
# le um byte para o registo t0 (sign extended).

sb \$t0, -3(\$t1)    # Store Byte  
# mem[t1-3]=t0 isto é, acede a memória  
# ao endereço calculado com t1-3 e  
# guarda o LSB do registo t0.  
# (LSB = Least Significant Byte)

- Não tem restrições de alinhamento dos endereços.
- Também existem instruções **lh** e **sh** para transferir halfwords (16 bits).

## Exemplo: Percorrer um array de words

```
.data
# vamos colocar uns numeros em memoria:
.word 1,2,-4,1000,0,-333
```

```
.text
# agora vem o codigo a executar:
```

main:

```
lui $t0, 0x????
ori $t0, $t0, 0x????
addi $t1, $zero, 6 # comprimento do array
```

LER\_ARRAY:

```
lw $t2, 0($t0)      # le word cujo endereco e' t0
addi $t0, $t0, 4     # avancar no array
addi $t1, $t1, -1    # decrementar contador
bne $t1, $zero, LER_ARRAY
nop
```

## Exemplo: Percorrer um array de words

```
.data
# vamos colocar uns numeros em memoria:
.word 1,2,-4,1000,0,-333

.text
# agora vem o codigo a executar:

main:
    lui $t0, 0x????
    ori $t0, $t0, 0x????
    addi $t1, $zero, 6 # comprimento do array

LER_ARRAY:
    lw $t2, 0($t0)      # le word cujo endereco e' t0
    addi $t1, $t1, -1   # decrementar contador
    bne $t1, $zero, LER_ARRAY
    addi $t0, $t0, 4     # avancar no array
```

## Exemplo: Percorrer uma string (array de chars)

Cada carácter (**char**) é um byte. Uma **string** é um array de chars terminada com o carácter nulo.

```
.data  
.asciiz "Hello World!"
```

```
.text
```

**main:**

```
lui $t0, 0x????  
ori $t0, $t0, 0x????
```

**LER\_STRING:**

```
lb $t2, 0($t0)      # le char do endereço t0  
bne $t2, $zero, LER_STRING  
addi $t0, $t0, 1     # endereço seguinte
```

## Alterar uma string (array chars)

Trocar pares de caracteres: "Hello World!" → "eHll ooWlr!d"

```
.data
```

```
.asciiz "Hello World!"
```

```
.text
```

```
main: lui $t0, 0x????
```

```
ori $t0, $t0, 0x????
```

```
TROCA2: lb $t1, 0($t0)      # le primeiro char
```

```
lb $t2, 1($t0)      # le segundo char
```

```
sb $t1, 1($t0)      # guarda na outra posicao
```

```
sb $t2, 0($t0)      # idem
```

```
j TROCA2
```

```
addi $t0, $t0, 2
```

## Alterar uma string (array chars)

Trocar pares de caracteres: "Hello World!" → "eHll ooWlr!d"

```
.data
```

```
.asciiz "Hello World!"
```

```
.text
```

```
main: lui $t0, 0x????
```

```
ori $t0, $t0, 0x????
```

```
TROCA2: lb $t1, 0($t0)      # le primeiro char
```

```
lb $t2, 1($t0)      # le segundo char
```

```
sb $t1, 1($t0)      # guarda na outra posicao
```

```
sb $t2, 0($t0)      # idem
```

```
j TROCA2
```

```
addi $t0, $t0, 2
```

Este programa "rebenta". Porquê?



## Alterar uma string (array chars)

Trocar pares de caracteres: "Hello World!" → "eHll ooWlr!d"

```
.data
.asciiz "Hello World!"

.text

main:   lui $t0, 0x????
        ori $t0, $t0, 0x????

TROCA2: lb $t1, 0($t0)      # le primeiro char
        lb $t2, 1($t0)      # le segundo char

        sb $t1, 1($t0)      # guarda na outra posicao
        sb $t2, 0($t0)      # idem

        j TROCA2
        addi $t0, $t0, 2
```

Este programa "rebenta". Porquê?

Corrija de modo a parar no final da string.

## Problema

- 1 *Escreva um programa que calcule o comprimento de uma string. Assuma que a string está no endereço em \$t0.*
- 2 *Escreva um programa que determine se a ordenação de bytes é little endian ou big endian. (Sugestão: use um sw seguido de um lb)*
- 3 *Escreva um programa que inverta a ordem de um array de números de 32 bits (array de words). Assuma que o array está no endereço \$t0 e tem comprimento \$t1.*