

Aulas 5 e 6

- Armazenamento de informação na memória externa
- Endereçamento indireto por registo com deslocamento
- Instruções de acesso a words de 32 bits armazenadas na memória externa: LW e SW
- Codificação das instruções de acesso à memória: formato I
- Restrições de alinhamento nos endereços das variáveis
- Instruções de acesso a bytes na memória externa: LB, LBU e SB
- Organização de informação em memória: "little-endian" *versus* "big-endian"
- Diretivas do *assembler* MARS

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Armazenamento de informação – registos internos

- Nos exemplos das aulas anteriores apenas se fez uso de registos internos do CPU para o armazenamento de informação (variáveis):

```
int a, b, c , d , z;      // a, b, c, d e z residem, respetivamente, em:  
z = (a + b) – (c + d);    // $17, $18, $19, $20 e $16  
  
add $8, $17, $18          # Soma $17 com $18 e armazena o resultado em $8  
add $9, $19, $20          # Soma $19 com $20 e armazena o resultado em $9  
sub $16, $8, $9           # Subtrai $9 a $8 e armazena o resultado em $16
```

- E se a informação a processar residir na memória externa (por exemplo um *array* de inteiros) ?
- Recorde-se que a arquitetura MIPS é do tipo **load-store**, ou seja, não é possível operar diretamente sobre o conteúdo da memória externa
- Terão que existir instruções para transferir informação entre os registos do CPU e a memória externa

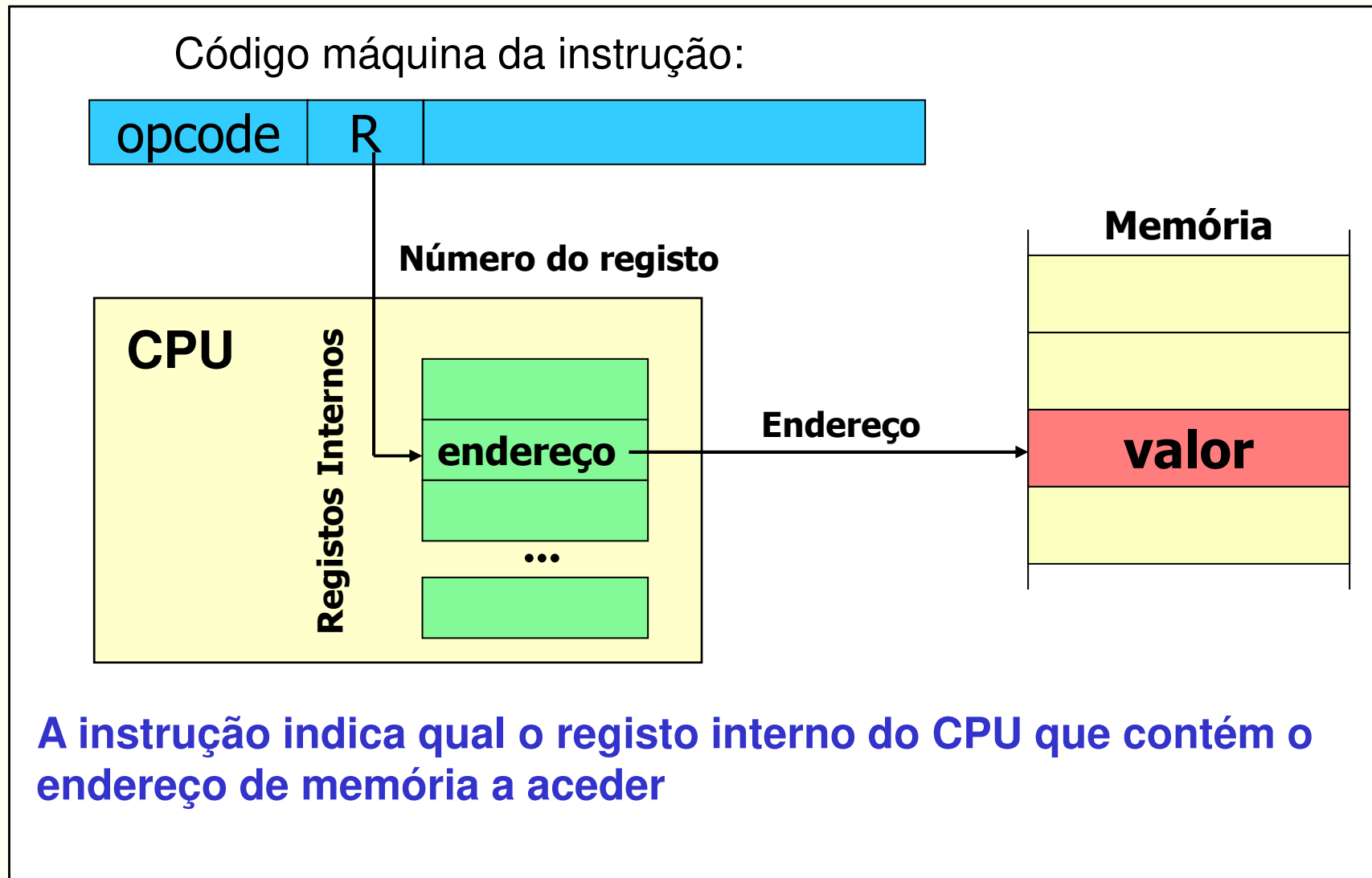
Modos de endereçamento

- O **método** usado pela arquitetura para **aceder ao elemento que contém a informação** que irá ser processada por uma dada instrução é genericamente designado por “**Modo de Endereçamento**”
- Nas instruções aritméticas e lógicas (codificadas com o formato R), os operandos residem ambos em registos internos
 - Os endereços dos registos internos envolvidos na operação são especificados diretamente na própria instrução, em campos de 5 bits: ***rs*** e ***rt***
 - Este modo é designado por **endereçamento tipo registo**

Acesso a informação residente na memória externa

- O acesso à memória externa implica sempre especificar o endereço da posição que se quer ler ou escrever
- O espaço de endereçamento do MIPS é de 32 bits, pelo que um endereço de memória é representado por 32 bits
- Como será então possível codificar as instruções de acesso à memória externa (para escrita e leitura), sabendo que as instruções do MIPS ocupam, todas, exatamente 32 bits?
 - Para codificar apenas o endereço já seriam necessários 32 bits...
- **Solução**: em vez do endereço, a instrução indica um registo que contém o endereço de memória a aceder (no MIPS um registo interno permite armazenar 32 bits):
 - **endereçamento indireto por registo**

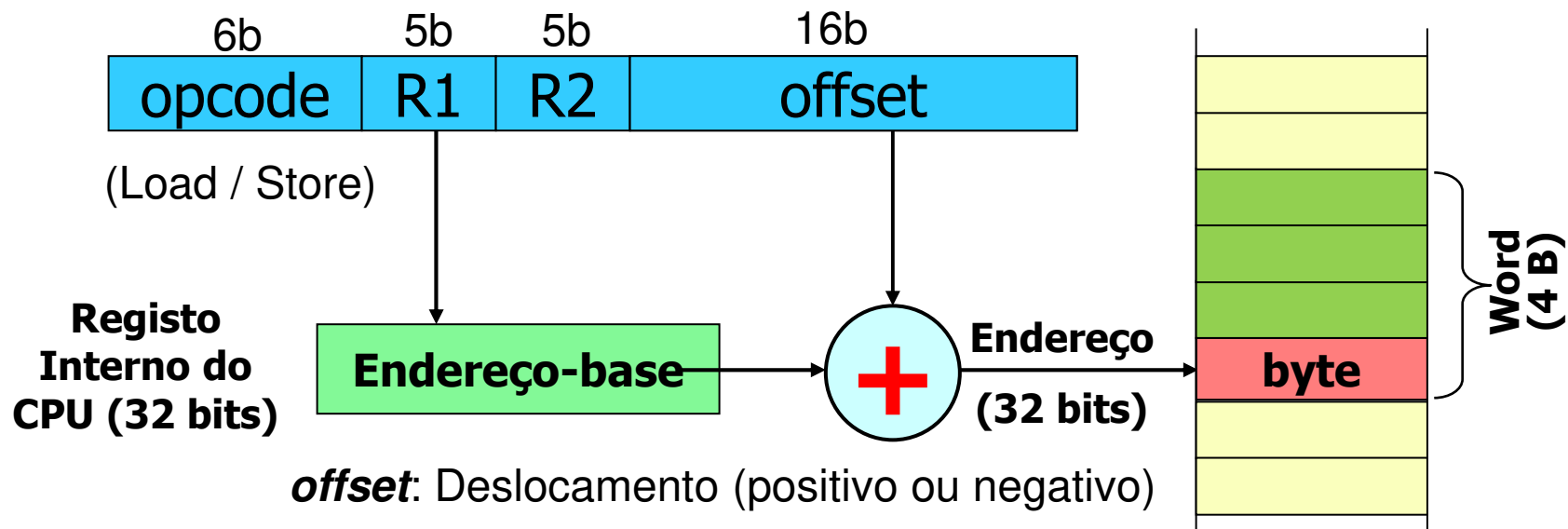
Endereçamento indireto por registo



Endereçamento indireto por registo com deslocamento

- A solução do MIPS

Código máquina da instrução:



offset: Deslocamento (positivo ou negativo)

R1: Registo de endereçamento indireto

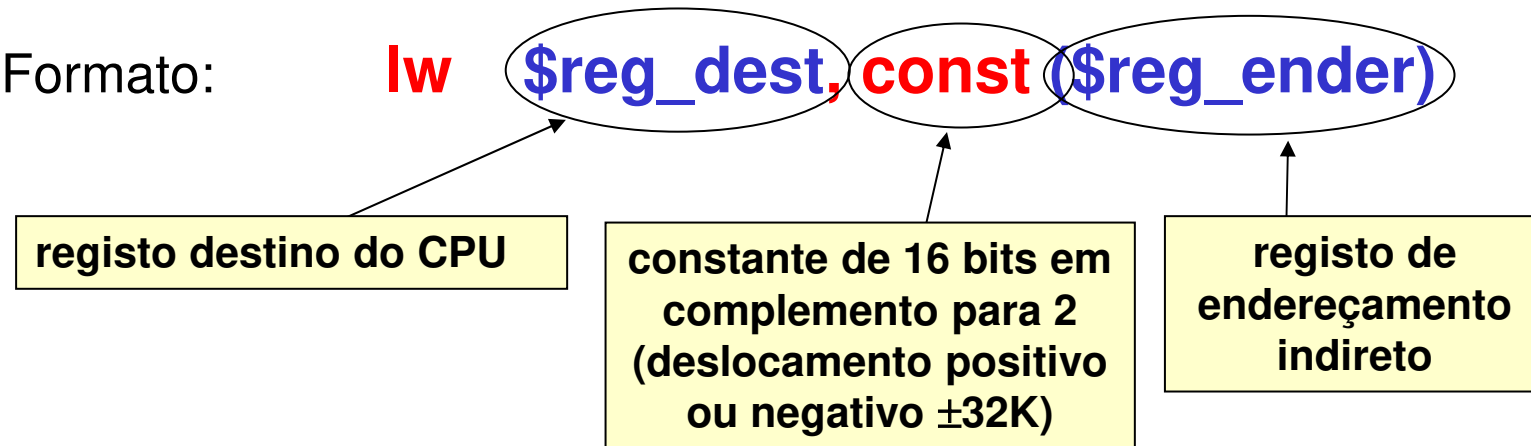
R2: Registo de dados: destino / origem

O endereço de acesso à memória é calculado pela **soma algébrica do conteúdo do registo com o *offset*** (estendido, com sinal, para a dimensão do registo, i.e., para 32 bits)

Leitura da memória – instrução LW

- **LW** - (*load word*) transfere uma palavra de 32 bits da memória para um registo interno do CPU (**1 word é armazenada em 4 posições de memória consecutivas**)

Formato:



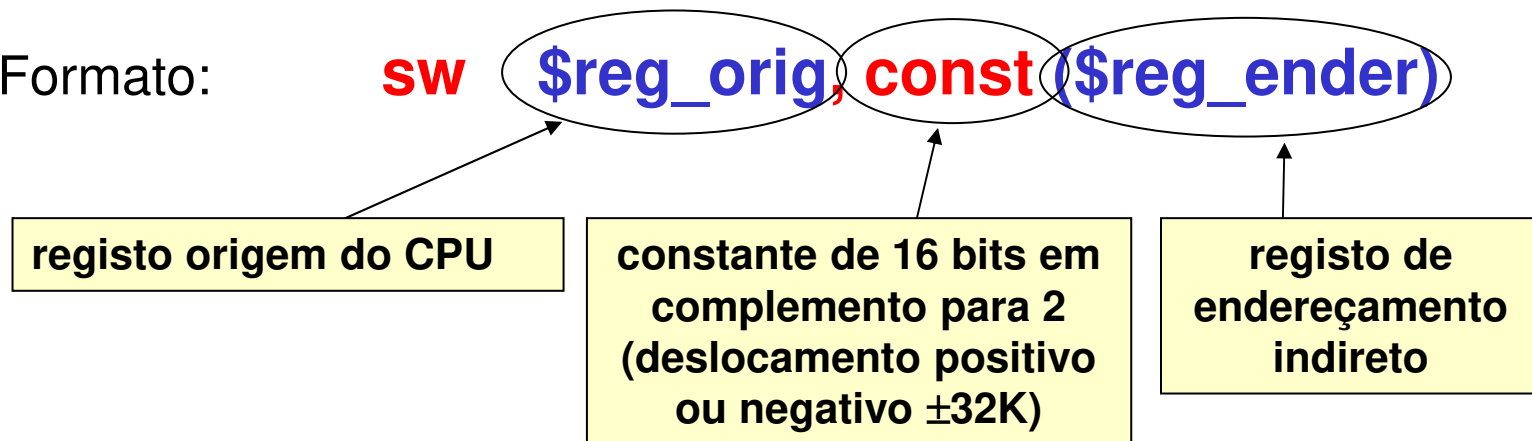
Exemplo:

lw \$5, 4 (\$2) # copia para o registo \$5 a *word* de 32 bits armazenada
a partir do endereço de memória calculado como:
addr = (conteúdo do registo \$2) + 4

Escrita na memória – instrução SW

- **SW** - (*store word*) transfere uma palavra de 32 bits de um registro interno do CPU para a memória (1 word é armazenada em 4 posições de memória consecutivas)

Formato:



Exemplo:

sw \$7, -8 (\$4) # copia a *word* armazenada no registro \$7 para a
memória, a partir do endereço calculado como:
addr = (conteúdo do registro \$4) - 8

Acesso à memória

- Exemplo

Address	Data	
0x00000020	0x45	word
0x00000021	0x12	
0x00000022	0x3A	
0x00000023	0xF3	
0x00000024	0xC9	word
0x00000025	0x7D	
0x00000026	0xB3	
0x00000027	0x9D	
0x00000028	0x47	word
0x00000029	0x5F	
0x0000002A	0x6D	
0x0000002B	0x4A	
0x0000002C	0xFD	word
0x0000002D	0xC0	
0x0000002E	0x5A	
0x0000002F	0x7C	
0x00000030	0x1D	
...	...	

`int A[4]; // array de inteiros`

Address	Data	
0x00000020	0x45	A[0]
0x00000021	0x12	
0x00000022	0x3A	
0x00000023	0xF3	
0x00000024	0xC9	A[1]
0x00000025	0x7D	
0x00000026	0xB3	
0x00000027	0x9D	
0x00000028	0x47	A[2]
0x00000029	0x5F	
0x0000002A	0x6D	
0x0000002B	0x4A	
0x0000002C	0xFD	A[3]
0x0000002D	0xC0	
0x0000002E	0x5A	
0x0000002F	0x7C	
0x00000030	0x1D	
...	...	

Endereço inicial do *array* A = endereço de A[0]: **0x20** (0x00000020)

Endereço de A[1]: **0x24**, endereço de A[2]: **0x28**, endereço de A[3]: **0x2C**

Acesso à memória: exemplo 1

- Considere-se o seguinte exemplo:

$g = h + A[3];$

assumindo que ***g***, ***h*** e o **endereço de início do array *A*** residem nos registros ***\$17***, ***\$18*** e ***\$19***, respetivamente

- Usando instruções do *Assembly* do MIPS, a expressão anterior tomaria a seguinte forma (supondo que *A* é um *array* de inteiros, i.e. 32 bits):

`lw $8, 12($19) # Lê A[3] da memória`
`add $17, $18, $8 # Calcula novo valor de g`

Variável temporária (destino)

Não esquecer que a memória está organizada em bytes (*byte-addressable*)

Acesso à memória: exemplo 1

- Na primeira instrução da sequência anterior

`lw $8, 12 ($19) # Lê A[3] da memória`

O endereço da memória é calculado pelo processador somando o conteúdo do registo indicado entre parêntesis com a constante explicitada na instrução

- Se, por exemplo, o conteúdo de **\$19** for **0x00000020** o endereço da memória a que a instrução acede é:

`lw $8, 12 ($19) # Lê A[3] da memória`

0x0000000C + 0x00000020 = 0x0000002C ← **Endereço resultante**

- Se o valor armazenado em \$19 corresponder ao endereço do primeiro elemento do *array* de inteiros e como cada elemento do *array* ocupa quatro *bytes*, o elemento acedido é A[3]

Acesso à memória: exemplo 2

- Se se pretendesse obter:

$$A[3] = h + A[3]$$

Assumindo mais uma vez que *h* e o endereço inicial do *array* residem nos registos **\$18** e **\$19**, respetivamente

- Poderíamos fazê-lo com o seguinte código:

lw	\$8, 12 (\$19)	# Lê A[3] da memória
add	\$8, \$18, \$8	# Calcula novo valor
sw	\$8, 12 (\$19)	# Escreve resultado em A[3]

Arquitetura load/store: as operações aritméticas e lógicas só podem ser efetuadas sobre registos internos do CPU

Codificação das instruções de acesso à memória no MIPS

- A necessidade de codificação de uma constante de 16 bits, obriga à definição de um novo formato de codificação, o **formato I**



Formato I

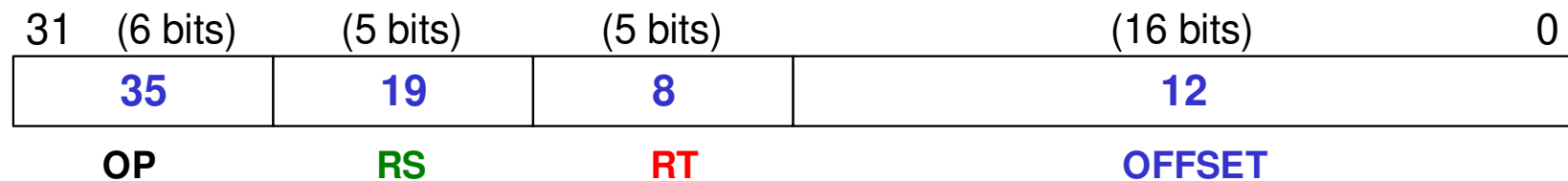
Codificado em complemento para 2 ($\pm 32K$)

- Gama de representação da constante de 16 bits
 - $[-32768, +32767]$

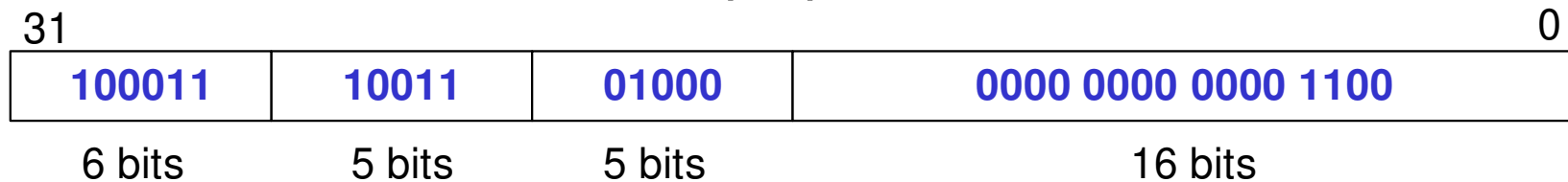
Codificação da instrução LW (Load Word)

lw **\$8,** **12(\$19)** **#Lê A[3] da memória**

Corresponderia à seguinte instrução máquina:



LW **RT, OFFSET(RS)**



10001110011010000000000000001100₂

1000 1110 0110 1000 0000 0000 0000 1100 = 0x8E68000C

Codificação da instrução SW (Store Word)

sw

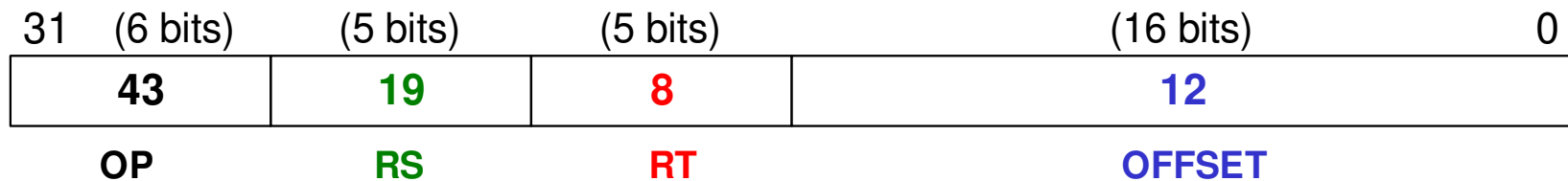
\$8,

12

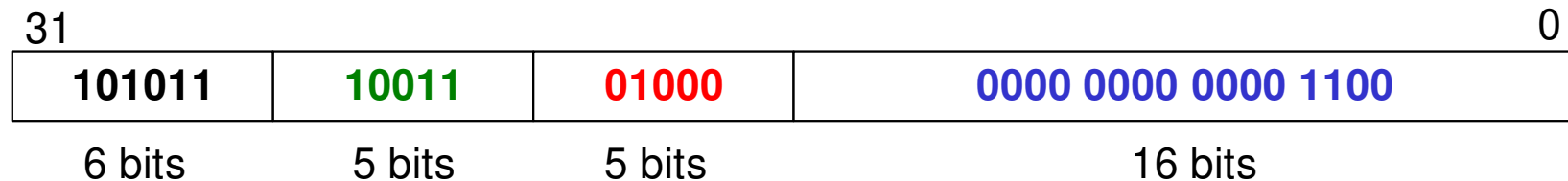
(\$19)

#Escreve result. em A[3]

Corresponderia à seguinte instrução máquina:



SW **RT, OFFSET(RS)**



10101110011010000000000000001100₂

1010 1110 0110 1000 0000 0000 0001 0100 = 0xAE68000C

Exemplo de codificação

- O seguinte trecho de código *assembly*:

```
lw    $8, 12($19)      # Lê A[3] da memória
add   $8, $18, $8       # Calcula novo valor
sw    $8, 12($19)      # Escreve resultado em A[3]
```

Corresponde à codificação:

31						0	
0x23	0x13	0x08	0x000C			Formato I	
0x00	0x12	0x08	0x08	0x00	0x20	Formato R	
0x2B	0x13	0x08	0x000C			Formato I	

- Resultando no código máquina:

$10001110011010000000000000001100_2 = 0x8E68000C$

$00000010010010000100000000100000_2 = 0x02484020$

$10101110011010000000000000001100_2 = 0xAE68000C$

Restrições de alinhamento nas instruções LW e SW

word {

Address	Data
0x00000020	0x45
0x00000021	0x12
0x00000022	0x3A
0x00000023	0xF3
0x00000024	0xC9
0x00000025	0x7D
0x00000026	0xB3
0x00000027	0x9D
0x00000028	0x47
0x00000029	0x5F
0x0000002A	0x6D
0x0000002B	0x4A
0x0000002C	0xFD
0x0000002D	0xC0
0x0000002E	0x5A
0x0000002F	0x7C
0x00000030	0x1D
...	...

} X



Address	Data
0x00000020	0x45123AF3
0x00000024	0xC97DB39D
0x00000028	0x475F6D4A
0x0000002C	0xFDC05A7C
0x00000030	...

O acesso a words só é possível em endereços múltiplos de 4

Restrições de alinhamento nas instruções LW e SW

- Externamente o barramento de endereços do MIPS só tem disponíveis 30 dos 32 bits: $A_{31}...A_2$. Ou seja, qualquer combinação nos bits A_1 e A_0 é ignorada no barramento de endereços exterior.
- Assim, do ponto de vista externo, só são gerados endereços **múltiplos de $2^2 = 4$** (ex: ...**0000**, ...**0100**, , ...**1000**, ...**1100**)

O acesso a words só é possível em endereços múltiplos de 4

- **Questão 1:** O que acontece quando o endereço calculado por uma instrução de leitura/escrita de uma **word** da memória, não é um múltiplo de 4 ?
- **Questão 2:** Como é possível a leitura/escrita de 1 byte de informação uma vez que o ISA do MIPS define que a memória é organizada em bytes (*byte-addressable*) ?

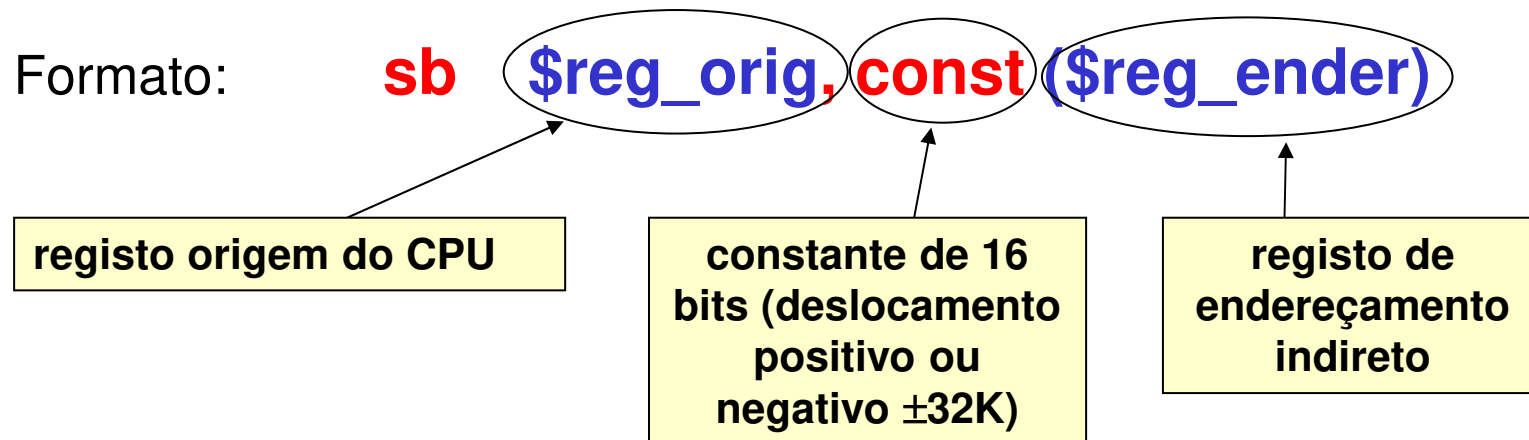
Restrições de alinhamento nos endereços das variáveis

- Se, numa instrução de leitura/escrita **de uma word**, for calculado um endereço **não múltiplo de 4**, quando o MIPS a tenta aceder à memória verifica que o endereço é inválido e **gera uma exceção**, terminando aí a execução do programa
- Como se evita o problema ?
 - Garantindo que as variáveis do tipo *word* estão armazenadas num endereço múltiplo de 4
- Diretiva **.align n** do *assembler* (força o alinhamento do endereço de uma variável num valor **múltiplo de 2^n**)
- Como se pode verificar facilmente que um endereço de 32 bits é múltiplo de 4? E múltiplo de 8?

Instrução de escrita de 1 *byte* na memória - SB

- **SB** - (store byte) transfere um *byte* de um registo interno para a memória – **só são usados os 8 bits menos significativos**

Formato:



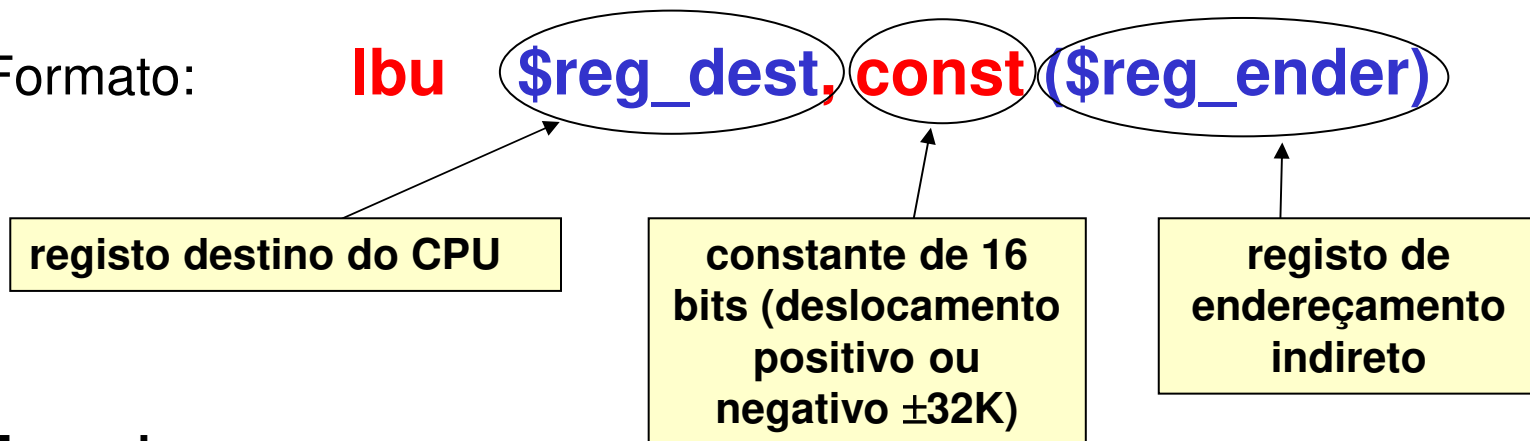
Exemplo:

sb \$7, 5 (\$4) # transfere o *byte* armazenado no registo \$7 (8
bits menos significativos) para o endereço de
memória calculado como:
addr = (conteúdo do registo \$4) + 5

Instrução de leitura de 1 *byte* na memória - LBU

- **LBU** - (load byte unsigned) transfere um *byte* da memória para um registro interno - **os 24 bits mais significativos do registro destino são colocados a 0**

Formato:



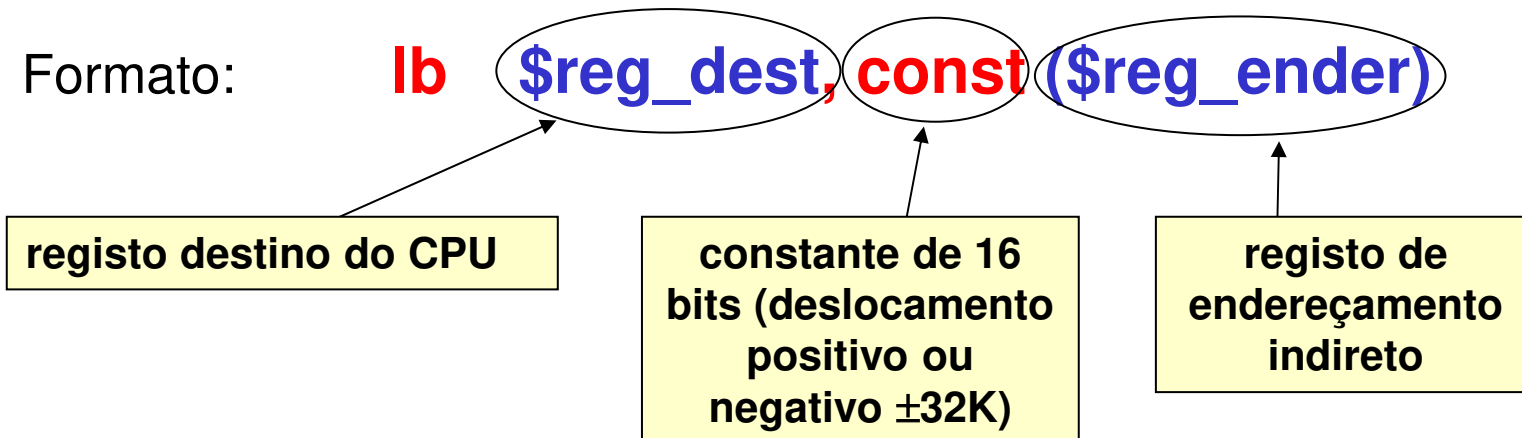
Exemplo:

Ibu \$5, -3 (\$2) # transfere para o registro \$5 o *byte* armazenado
no endereço de memória calculado como:
addr = (conteúdo do registro \$2) - 3
os 24 bits mais significativos de \$5 são
colocados a zero

Instrução de leitura de 1 *byte* na memória - LB

- **LB** - (load byte) transfere um *byte* da memória para um registo interno, **fazendo extensão de sinal do valor lido de 8 para 32 bits**

Formato:



Exemplo:

Ib \$5, 0 (\$2)

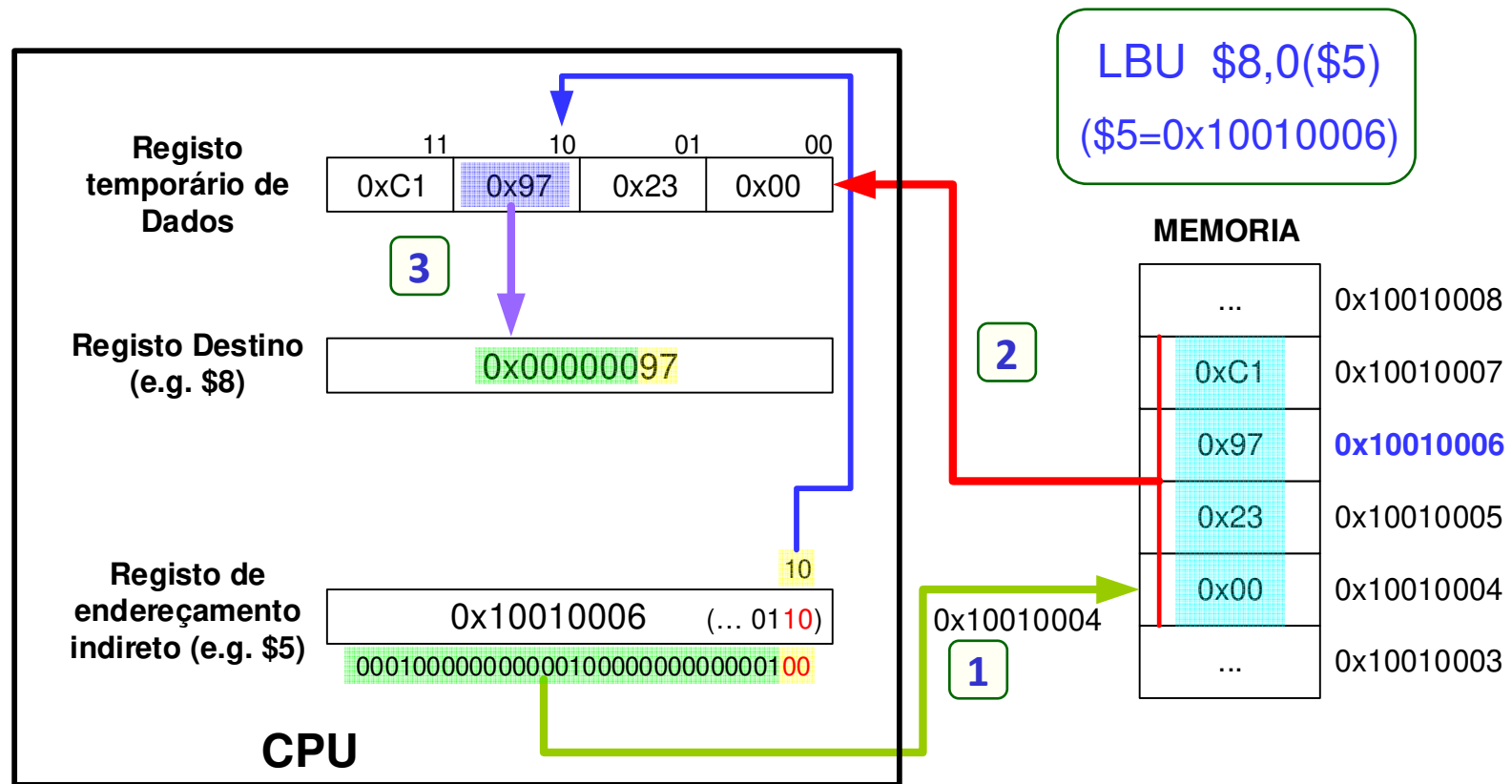
transfere para o registo \$5 o *byte* armazenado
no endereço de memória calculado como:
addr = (conteúdo do registo \$2) + 0
o bit mais significativo do *byte* transferido é
replicado nos 24 bits mais significativos de \$5

Escrita / leitura de 1 *byte* na memória

- Na leitura/escrita de 1 *byte* de informação o problema do alinhamento, do ponto de vista do programador, não se coloca
- Como é que o MIPS resolve o acesso?
 - O MIPS gera o endereço múltiplo de 4 (EM4) que, no acesso a uma *word* de 32 bits inclui o endereço pretendido
 - No caso de **Leitura** (instruções **lb**, **lbu**):
 - Executa uma instrução de leitura de 1 *word* do endereço EM4 e, dos 32 bits lidos, retira os 8 bits correspondentes ao endereço pretendido
 - No caso de **Escrita** (instrução **sb**):
 - Executa uma instrução de leitura de 1 *word* do endereço EM4
 - De entre os 32 bits lidos substitui os 8 bits que correspondem ao endereço pretendido
 - Escreve a *word* modificada em EM4
 - Sequência conhecida como "**read-modify-write**"

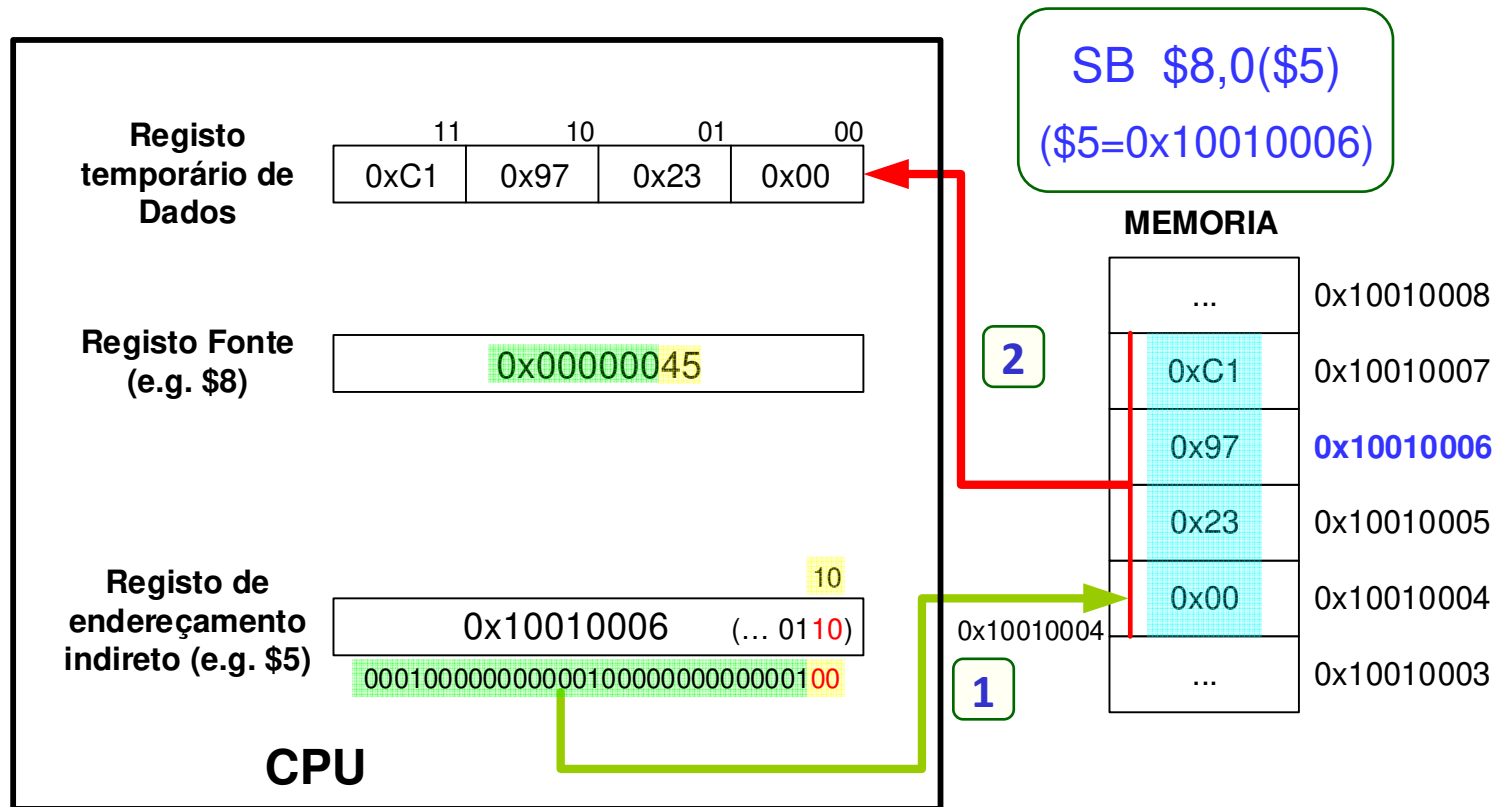
Exemplo: leitura de 1 *byte* da memória

- Exemplo para o caso da leitura (instrução **lbu** a ler o conteúdo da posição de memória **0x10010006**)



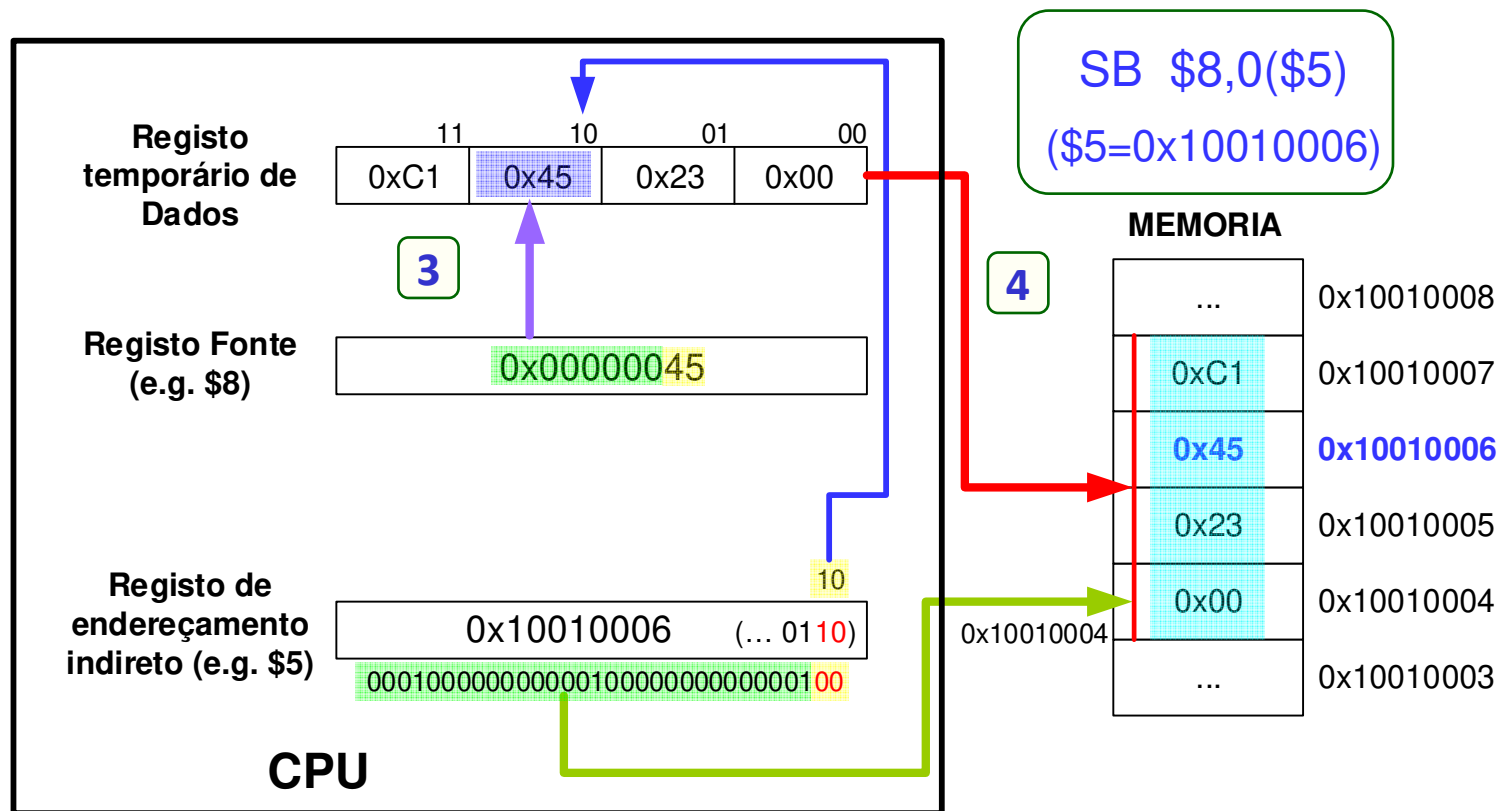
Exemplo: escrita de 1 *byte* da memória (fase 1)

- Exemplo para o caso da escrita (instrução **sb** a copiar um valor para a posição de memória **0x10010006**) - **READ**



Exemplo: escrita de 1 *byte* da memória (fase 2)

- Exemplo para o caso da escrita (instrução **sb** a copiar um valor para a posição de memória **0x10010006**) - **MODIFY / WRITE**



Organização das *words* de 32 bits na memória

- A memória no MIPS está organizada em *bytes* (*byte-addressable memory*)
- Se a quantidade a armazenar tiver uma dimensão superior a 8 bits vão ser necessárias várias posições de memória consecutivas (por exemplo, para uma *word* de 32 bits são necessárias 4 posições de memória)
- Exemplo: **0x012387A5** (4 bytes: **01 23 87 A5**)
- Qual a ordem de armazenamento dos *bytes* na memória?
Duas alternativas:
 - *byte* mais significativo armazenado no endereço mais baixo da memória (***big-endian***)
 - *byte* menos significativo armazenado no endereço mais baixo da memória (***little-endian***)

Organização das words de 32 bits na memória

- Exemplo: **0x012387A5** (**0x01 23 87 A5**)

Address	Data
0x10010008	?
0x10010009	?
0x1001000A	?
0x1001000B	?
0x1001000C	0x01
0x1001000D	0x23
0x1001000E	0x87
0x1001000F	0xA5
0x10010010	?
0x10010011	?
0x10010012	?
0x10010013	?
0x10010014	?
...	...



Big-Endian

Address	Data
0x10010008	?
0x10010009	?
0x1001000A	?
0x1001000B	?
0x1001000C	0xA5
0x1001000D	0x87
0x1001000E	0x23
0x1001000F	0x01
0x10010010	?
0x10010011	?
0x10010012	?
0x10010013	?
0x10010014	?
...	...



Little-Endian

- O simulador MARS implementa "little-endian"

Diretivas do *Assembler*

- Diretivas são comandos especiais colocados num programa em linguagem *assembly* destinados a instruir o *assembler* a executar uma determinada tarefa ou função
- Diretivas **não são instruções** da linguagem *assembly* (não fazem parte do ISA), não gerando qualquer código máquina
- As diretivas podem ser usadas com diversas finalidades:
 - reservar e inicializar espaço em memória para variáveis
 - controlar os endereços reservados para variáveis em memória
 - especificar os endereços de colocação de código e dados na memória
 - definir valores simbólicos
- As diretivas são específicas para um dado *assembler* (em AC1 usaremos as diretivas definidas pelo *assembler* do simulador MARS)

Diretivas do *Assembler* do MARS

.ASCIIZ *str*

Reserva espaço e armazena a string *str* em sucessivas posições de memória; acrescenta o terminador '`\0`' (**NUL**)

Ex:

msg1: .ASCIIZ "Arquitetura de Computadores"

.SPACE *n*

Reserva *n* posições consecutivas (endereços) de memória, sem inicialização

Ex:

array: .SPACE 20

.BYTE b_1, b_2, \dots, b_n Reserva espaço e armazena os bytes b_1, b_2, \dots, b_n em sucessivas posições de memória

Ex:

array: .BYTE 0x41, 0x43, 0x31, 0x00

Diretivas do *Assembler* do MARS

.WORD w_1, w_2, \dots, w_n Reserva espaço e armazena as *words* w_1, w_2, \dots, w_n em sucessivas posições de memória (cada *word* em 4 endereços consecutivos)

Ex:

array: .WORD 0x012387A5, 0xF34, 0x678AC

.ALIGN n Alinha o próximo item num endereço múltiplo de 2^n

Ex:

.ALIGN 2 # próximo item está alinhado num endereço
múltiplo de 4

.EQV *symbol, val* Atribui a um símbolo um valor. No programa o assembler substitui as ocorrências de ***symbol*** por ***val***

Ex:

.EQV TRUE, 1
.EQV FALSE, 0

Diretivas do Assembler - exemplo

```

        .DATA      # 0x10010000
STR1:   .ASCIIZ    "AULA5"
ARR1:   .WORD      0x1234, MAIN
VARB:   .BYTE      0x12
        .ALIGN     2
VARW:   .SPACE     4 #space for 1 word
        .TEXT      # 0x00400000
        .GLOBL     MAIN
MAIN:

```

- Utilizar a diretiva **".align"** sempre que se pretender que o endereço subsequente esteja alinhado
- A diretiva **".word"** alinha automaticamente num endereço múltiplo de 4

0x10010017	??
0x10010016	??
0x10010015	??
VARW 0x10010014	??
0x10010013	?? (unused)
0x10010012	?? (unused)
0x10010011	?? (unused)
VARB 0x10010010	0x12
0x1001000F	0x00
0x1001000E	0x40
0x1001000D	0x00
0x1001000C	0x00
0x1001000B	0x00
0x1001000A	0x00
0x10010009	0x12
ARR1 0x10010008	0x34
0x10010007	?? (unused)
0x10010006	?? (unused)
0x10010005	'\0' (0x00)
0x10010004	'5' (0x35)
0x10010003	'A' (0x41)
0x10010002	'L' (0x4C)
0x10010001	'U' (0x55)
STR1 0x10010000	'A' (0x41)

Questões / exercícios

- Qual o modo de endereçamento usado pelo MIPS para acesso a quantidades residentes na memória externa?
- Na instrução "**lw** \$3, 0x24(\$5)" qual a função dos registos \$3 e \$5 e da constante 0x24?
- Qual o formato de codificação das instruções de acesso à memória no MIPS e qual o significado de cada um dos seus campos?
- Qual a diferença entre as instruções "**sw**" e "**sb**"? O que distingue as instruções "**lb**" e "**lbu**"?
- O que acontece quando uma instrução **lw/sw** acede a um endereço que não é múltiplo de 4?
- Sabendo que o *opcode* da instrução "**lw**" é 0x23, determine o código máquina, expresso em hexadecimal, da instrução "**lw** \$3, 0x24(\$5)".

Questões / exercícios

- Suponha que a memória externa foi inicializada, a partir do endereço **0x10010000**, com os valores **0x01, 0x02, 0x03, 0x04, 0x05, ...**. Suponha ainda que **\$3=0x1001** e **\$5=0x10010000**. Qual o valor armazenado no registo destino após a execução da instrução **"lw \$3, 0x24(\$5)"**?
- Nas condições anteriores qual o valor armazenado no registo destino pelas instruções: **"lbu \$3, 0xA3(\$5)"** e **"lb \$4, 0xA3(\$5)"**?
- Quantos bytes são reservados em memória por cada uma das diretivas:
L1: .asciiz "Aulas5&6T"
L2: .word 5, 8, 23
L3: .byte 5, 8, 23
L4: .space 8
- Acrescente a diretiva **".align 2"** a seguir a L3. Desenhe esquematicamente a memória e preencha-a com o resultado das diretivas anteriores.
- Supondo que **"L1"** corresponde ao endereço inicial do segmento de dados, e que esse endereço é **0x10010000**, determine os endereços a que correspondem os *labels* **"L2"**, **"L3"** e **"L4"**, nas condições da questão anterior.