

Aula 4

- Armazenamento de informação na memória externa
- Endereçamento indireto por registo com deslocamento
- Instruções de acesso a informação residente na memória externa: LW, SW, LB, LBU, SB
- Codificação das instruções de acesso à memória: formato I
- Restrições de alinhamento nos endereços das variáveis
- Organização de informação em memória: "little-endian" *versus* "big-endian"

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador

Armazenamento de informação – registos internos

- Exemplo de um trecho de código que apenas faz uso de registos internos do CPU para o armazenamento de informação:

```
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
```

- Sendo o equivalente em C:

```
// a, b, c, d e z residem, respetivamente, em:
// $17, $18, $19, $20 e $16
// $8 e $9 representam variáveis temporárias não explicitadas em C
```

```
int a, b, c, d, z;
z = (a + b) - (c + d);
```

Armazenamento de informação – memória externa

- E se se pretendesse somar os elementos de um *array* composto por N elementos?
- Se N for maior do que o número de registos disponíveis no CPU seria necessário recorrer a recursos externos – a memória
- Por outro lado, a arquitetura do MIPS é do tipo **load-store**, pelo que não é possível operar diretamente sobre o conteúdo da memória externa
- Deverão existir, portanto, instruções para transferir informação entre os registos do CPU e a memória externa

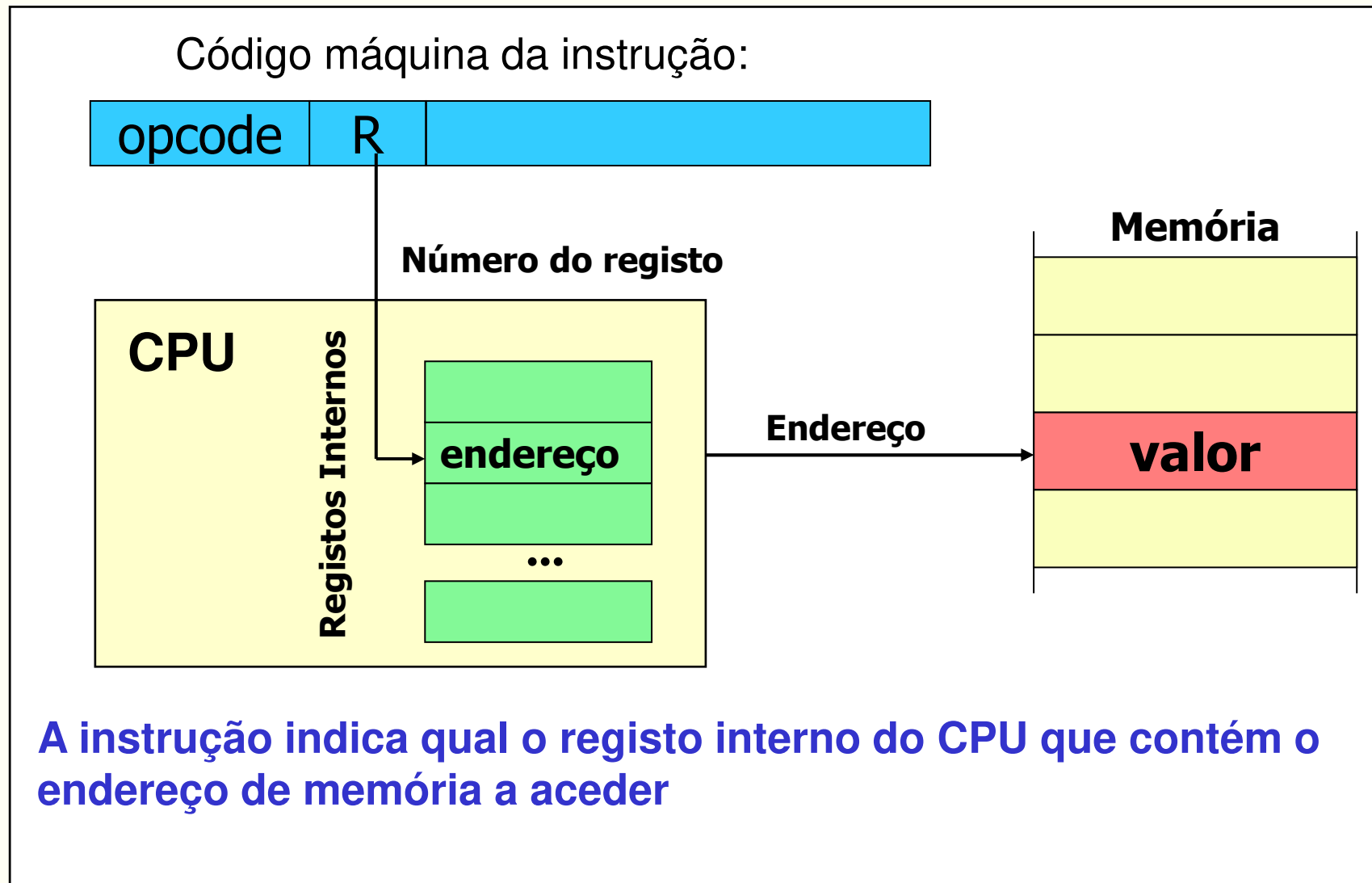
Modos de endereçamento

- 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, os operandos residem em registos internos (um dos operandos também pode ser uma constante)
- 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

- 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?
- Note-se que um endereço de memória no MIPS é representado por 32 bits, pelo que ele sozinho ocuparia a totalidade do código máquina da instrução
- **Solução:** em vez do endereço, a instrução indica um registo que contém o endereço de memória a aceder (recorde-se que um registo interno permite armazenar 32 bits):
 - **endereçamento indireto por registo**

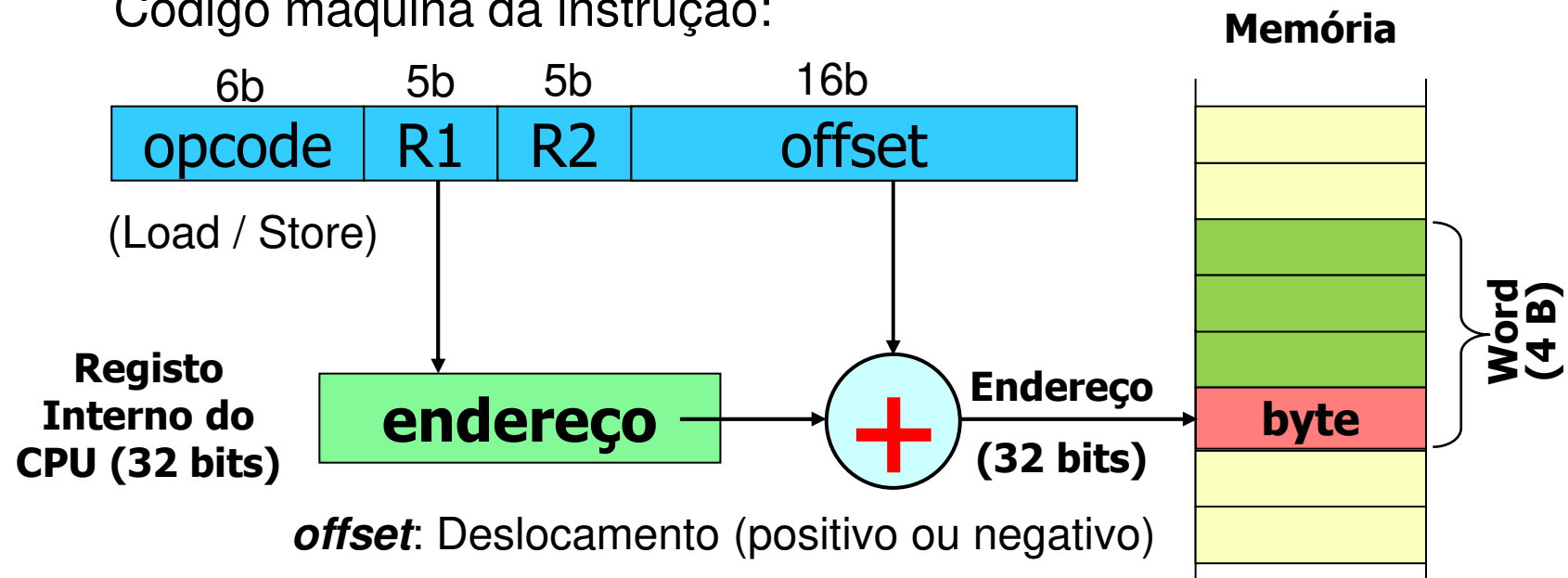
Endereçamento indireto por registo



Endereçamento indireto por registo com deslocamento

- A solução do MIPS

Código máquina da instrução:



R1: Registo de endereçamento

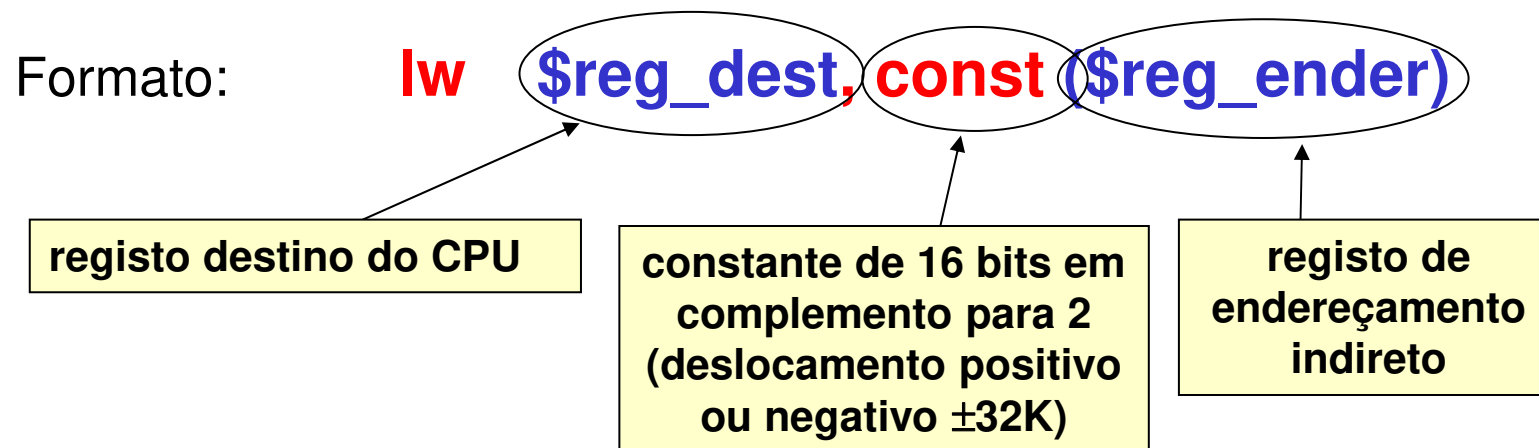
R2: Registo de dados: destino / origem

O endereço de acesso à memória é calculado pela **soma algébrica do conteúdo do registo com o *offset*** (extendido, 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:

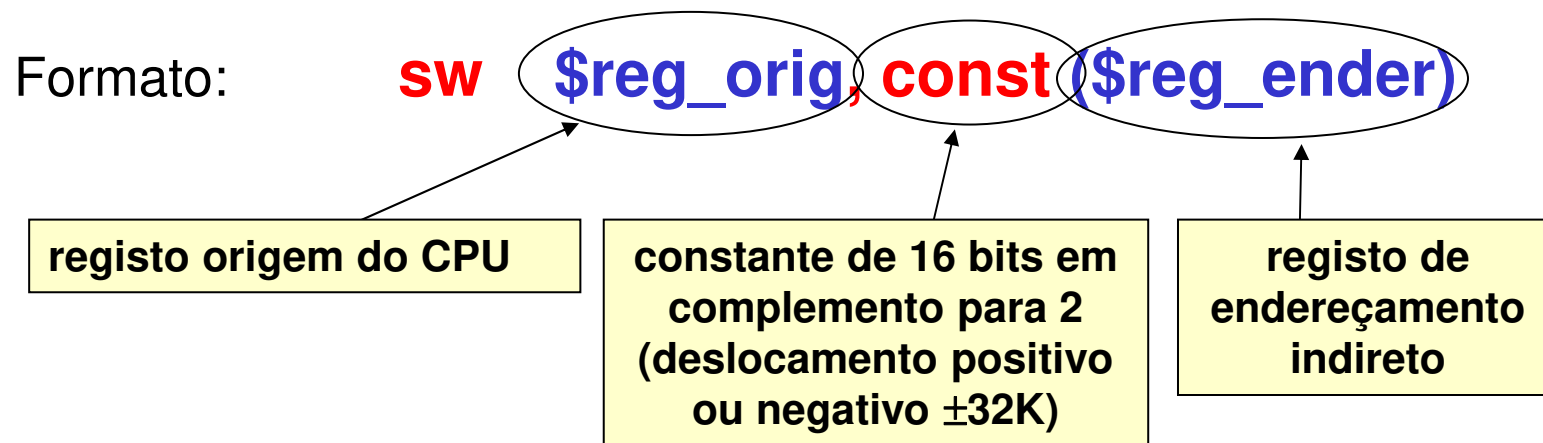


Exemplo:

lw \$5, 4 (\$2) # copia para o registo \$5 a *word* 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 registo interno do CPU para a memória (1 word é armazenada em 4 posições de memória consecutivas)



Exemplo:

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

Acesso à memória: exemplo 1

- Considere-se o seguinte exemplo:

$$g = h + A[5]$$

assumindo que **g**, **h** e o **endereço de início do array A** residem nos registos **\$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 *words*, i.e. 32 bits):

`lw $8, 20($19) # Lê A[5] 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

- Retomemos a primeira instrução:

`lw $8, 20 ($19) # Lê A[5] da memória`

- O endereço da memória é calculado somando o conteúdo do registo indicado entre parêntesis com a constante explicitada na instrução. Se o conteúdo de **\$19** for **0x10010000** o endereço da memória será:

`lw $8, 20 ($19) # Lê A[5] da memória`

$0x14 + 0x10010000 = 0x10010014$

Endereço resultante

- Como cada elemento do *array* ocupa quatro *bytes* (*array* de *words*), o elemento acedido será A[5]

Acesso à memória: exemplo 2

- Se se pretendesse obter:

$$A[5] = h + A[5]$$

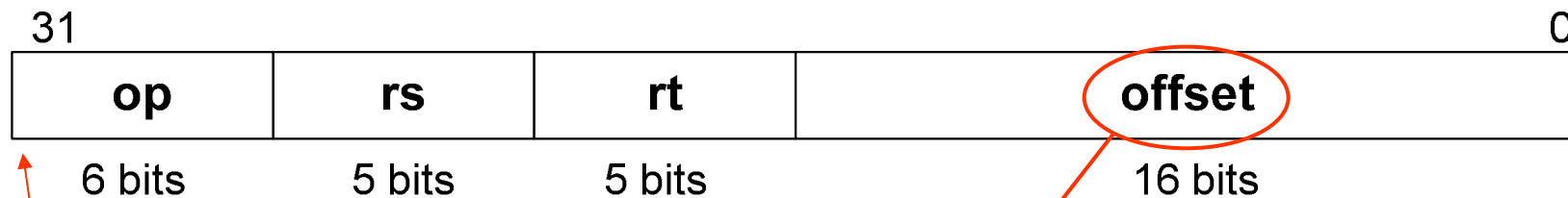
- Assumindo mais uma vez que *h* e o endereço inicial do *array* residem nos registos **\$18** e **\$19**, respetivamente
- Poderíamos fazê-lo com o seguinte código:

lw	\$8, 20 (\$19)	# Lê A[5] da memória
add	\$8, \$18, \$8	# Calcula novo valor
sw	\$8, 20 (\$19)	# Escreve resultado em A[5]

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

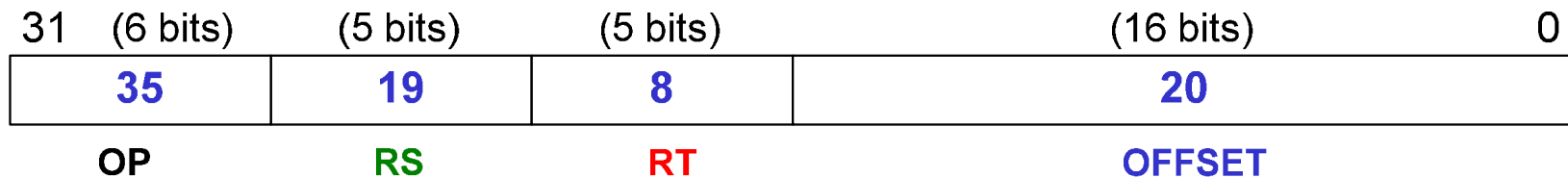
Codificado em complemento para 2 ($\pm 32K$)

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

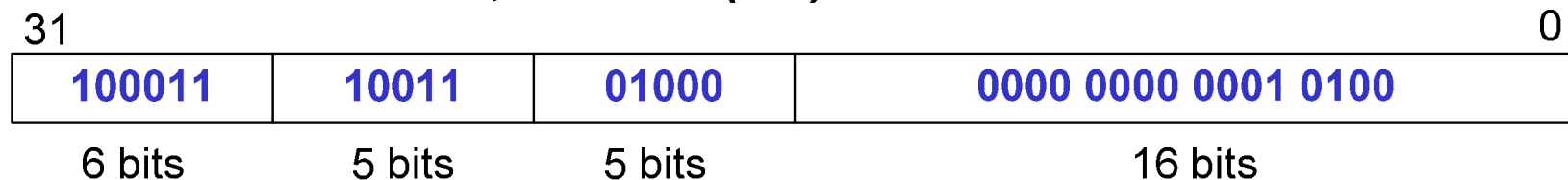
Codificação da instrução LW (Load Word)

lw \$8, 20(\$19) #Lê A[5] da memória

Corresponderia à seguinte instrução máquina:



LW RT, OFFSET(RS)



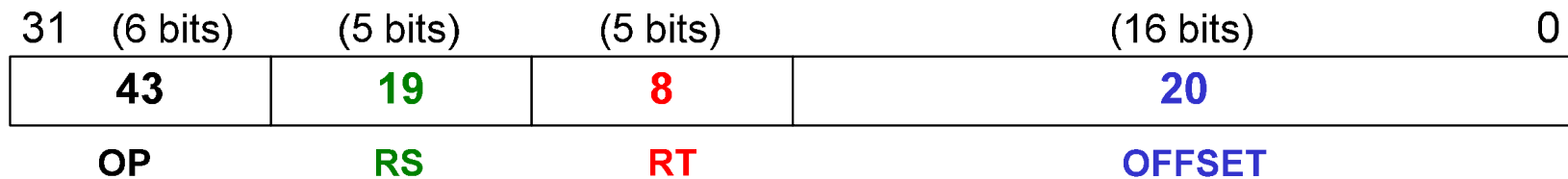
10001110011010000000000000010100₂

1000 1110 0110 1000 0000 0000 0001 0100 = 0x8E680014

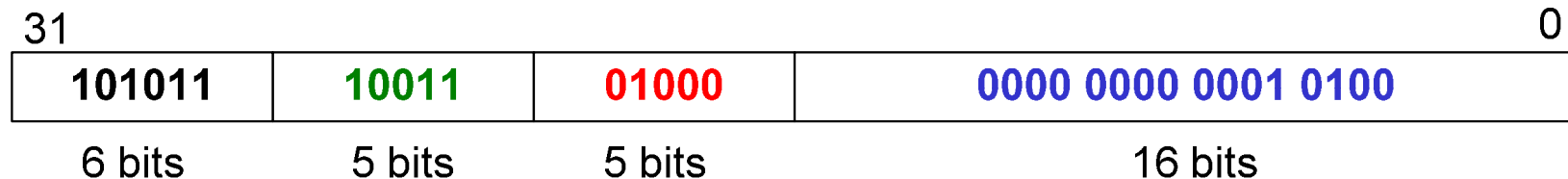
Codificação da instrução SW (Store Word)

sw **\$8**, **20**(**\$19**) #Escreve result. em A[5]

Corresponderia à seguinte instrução máquina:



SW **RT**, **OFFSET**(**RS**)



10101110011010000000000000010100₂

1010 1110 0110 1000 0000 0000 0001 0100 = 0xAE680014

Exemplo de codificação

- O seguinte trecho de código *assembly*:

```
lw    $8, 20($19)      # Lê A[5] da memória
add   $8, $18, $8      # Calcula novo valor
sw    $8, 20($19)      # Escreve resultado em A[5]
```

Corresponde à codificação:

31						0	
0x23	0x13	0x08	0x0014				Formato I
0x00	0x12	0x08	0x08	0x00	0x20		Formato R
0x2B	0x13	0x08	0x0014				Formato I

- Resultando no código máquina:

$1000111001101000000000000010100_2 = 0x8E680014$

$00000010010010000100000000100000_2 = 0x02484020$

$1010111001101000000000000010100_2 = 0xAE680014$

Restrições de alinhamento nos endereços das variáveis

- 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 MIPS tenta executar uma instrução de leitura/escrita de uma **word** da memória, num endereço não 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

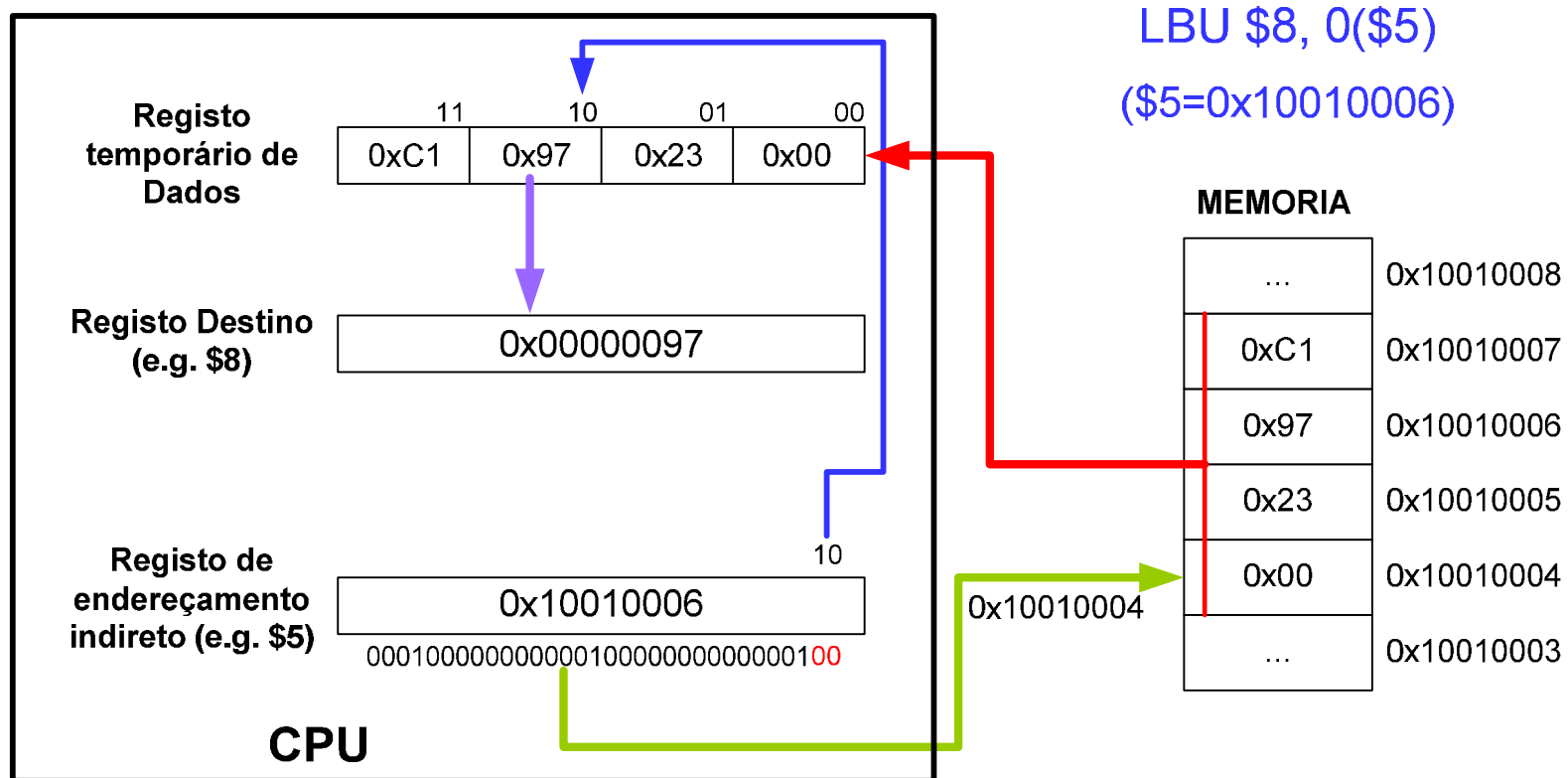
- Se, numa instrução de leitura/escrita de uma *word*, for especificado um endereço **não múltiplo de 4**, quando o MIPS a tenta executar verifica que o endereço é inválido e **gera uma exceção**, terminando aí a execução do programa
- Como se evita o problema ?
 - Garantindo que as variáveis do tipo *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 2ⁿ**)

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*, 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**"

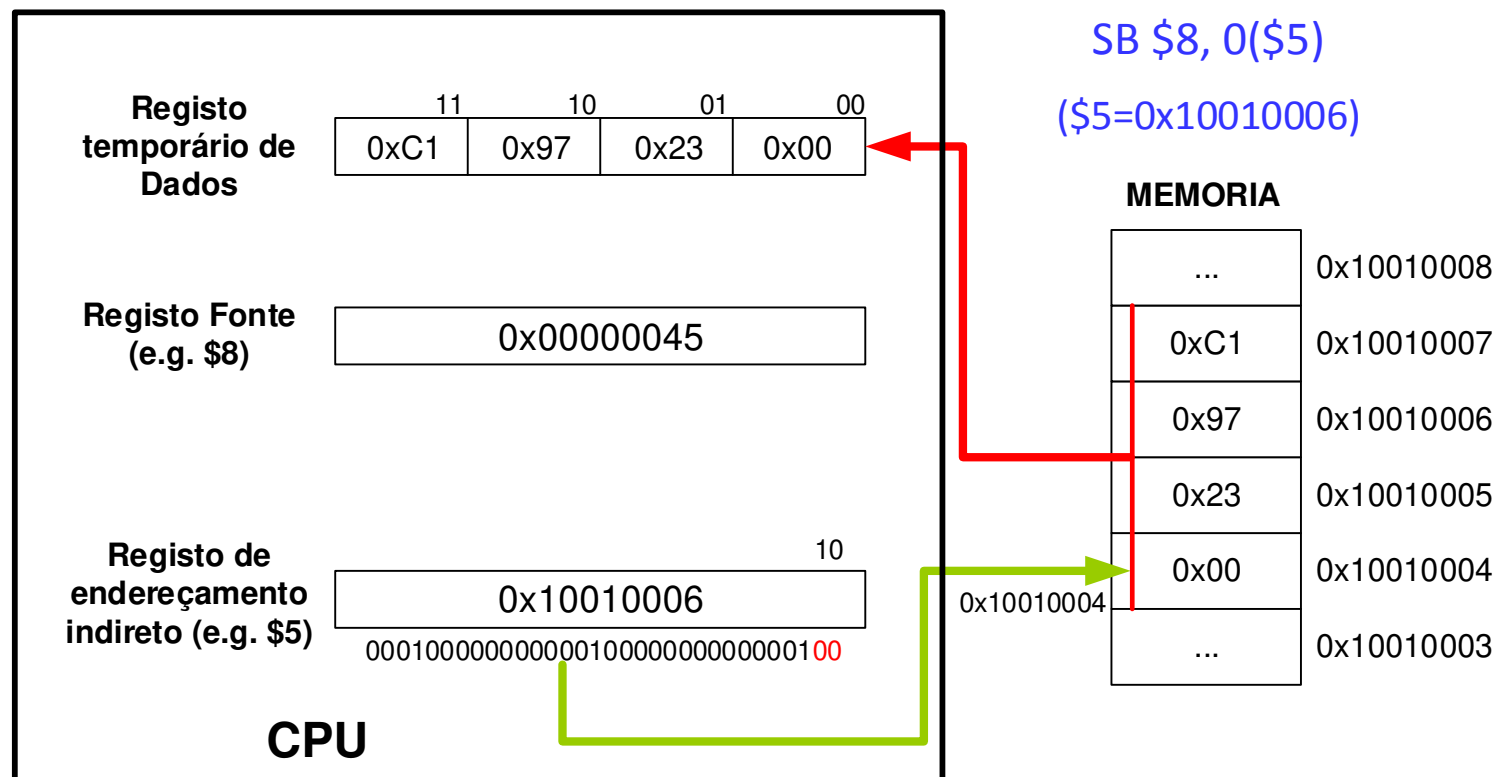
Exemplo: leitura de 1 *byte* da memória

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



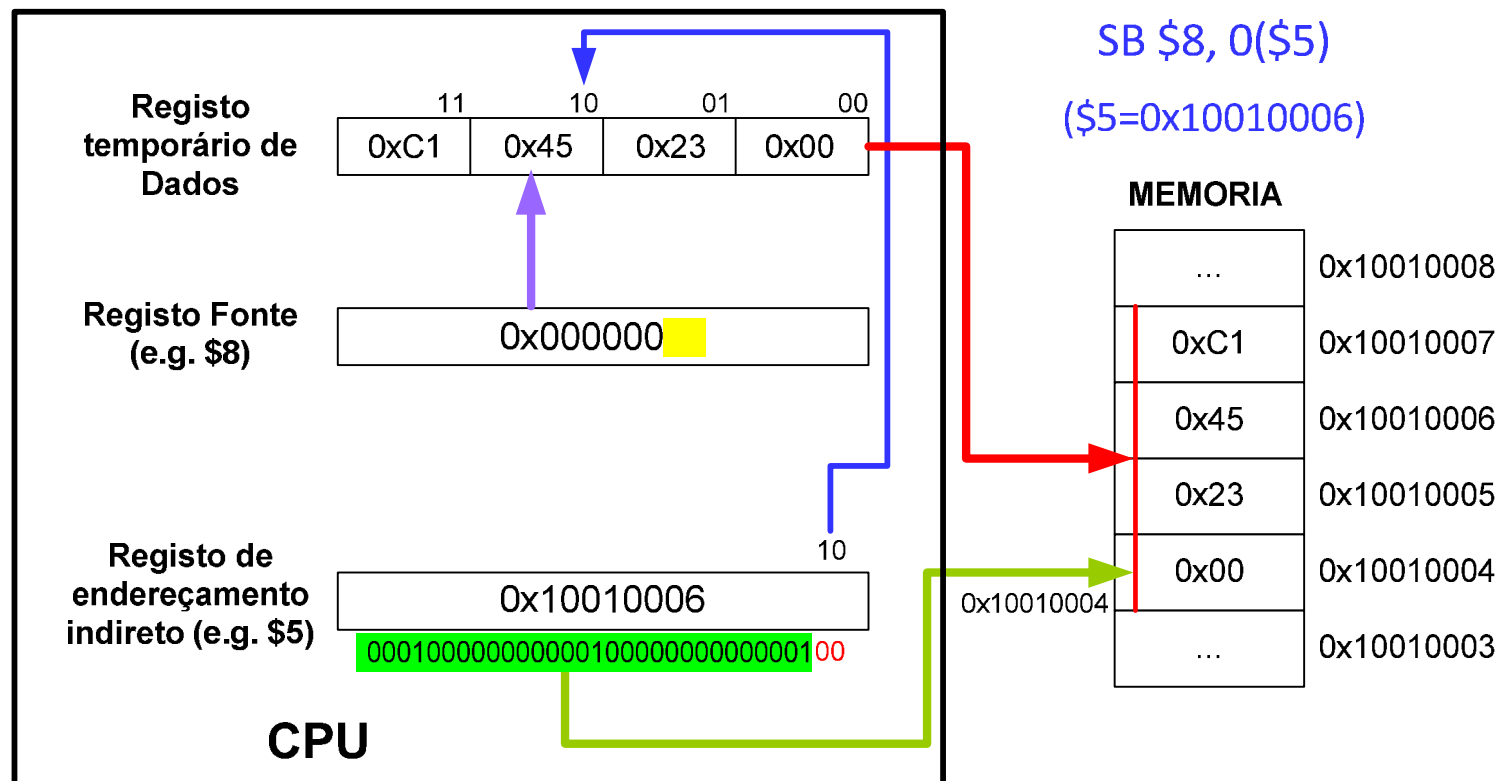
Exemplo: escrita de 1 *byte* da memória (fase 1)

- Exemplo para o caso da escrita (instrução **sb** a escrever o conteúdo da posição de memória **0x10010006**) - READ



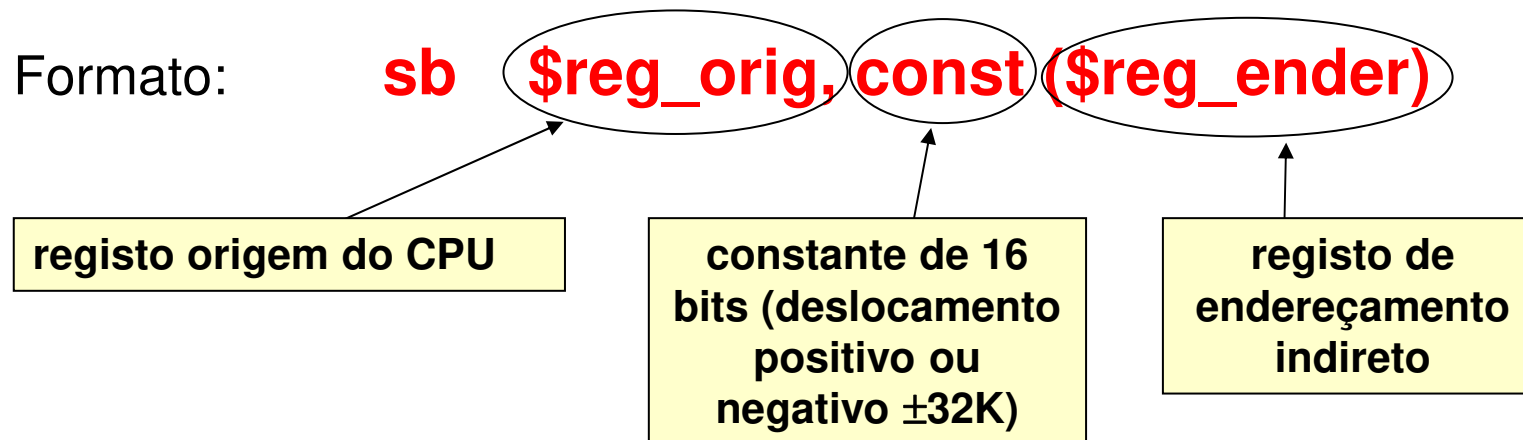
Exemplo: escrita de 1 *byte* da memória (fase 2)

- Exemplo para o caso da escrita (instrução **sb** a escrever o conteúdo da posição de memória **0x10010006**) – MODIFY / WRITE



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**

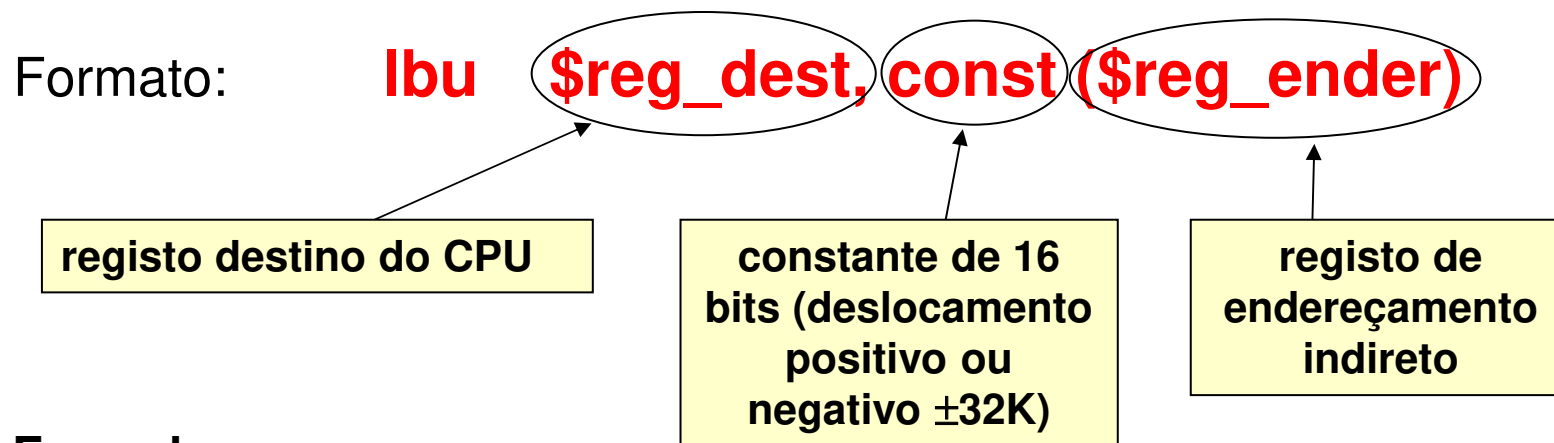


Exemplo:

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

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

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



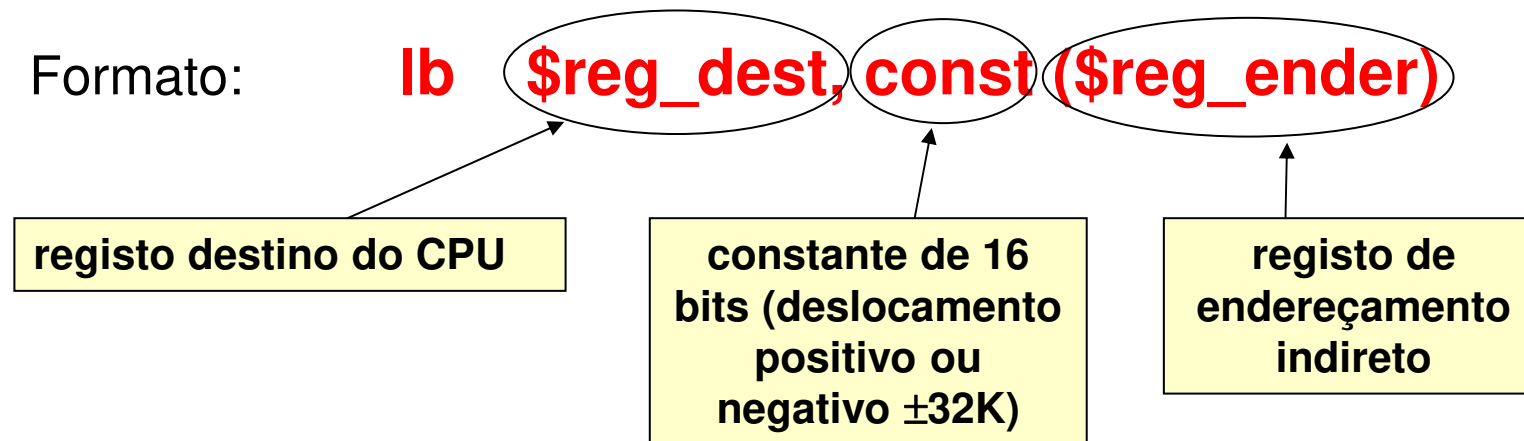
Exemplo:

lbu \$5, -4 (\$2) # transfere para o registo \$5 o *byte* armazenado
no endereço de memória calculado como:
addr = (conteúdo do registo \$2) - 4
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:



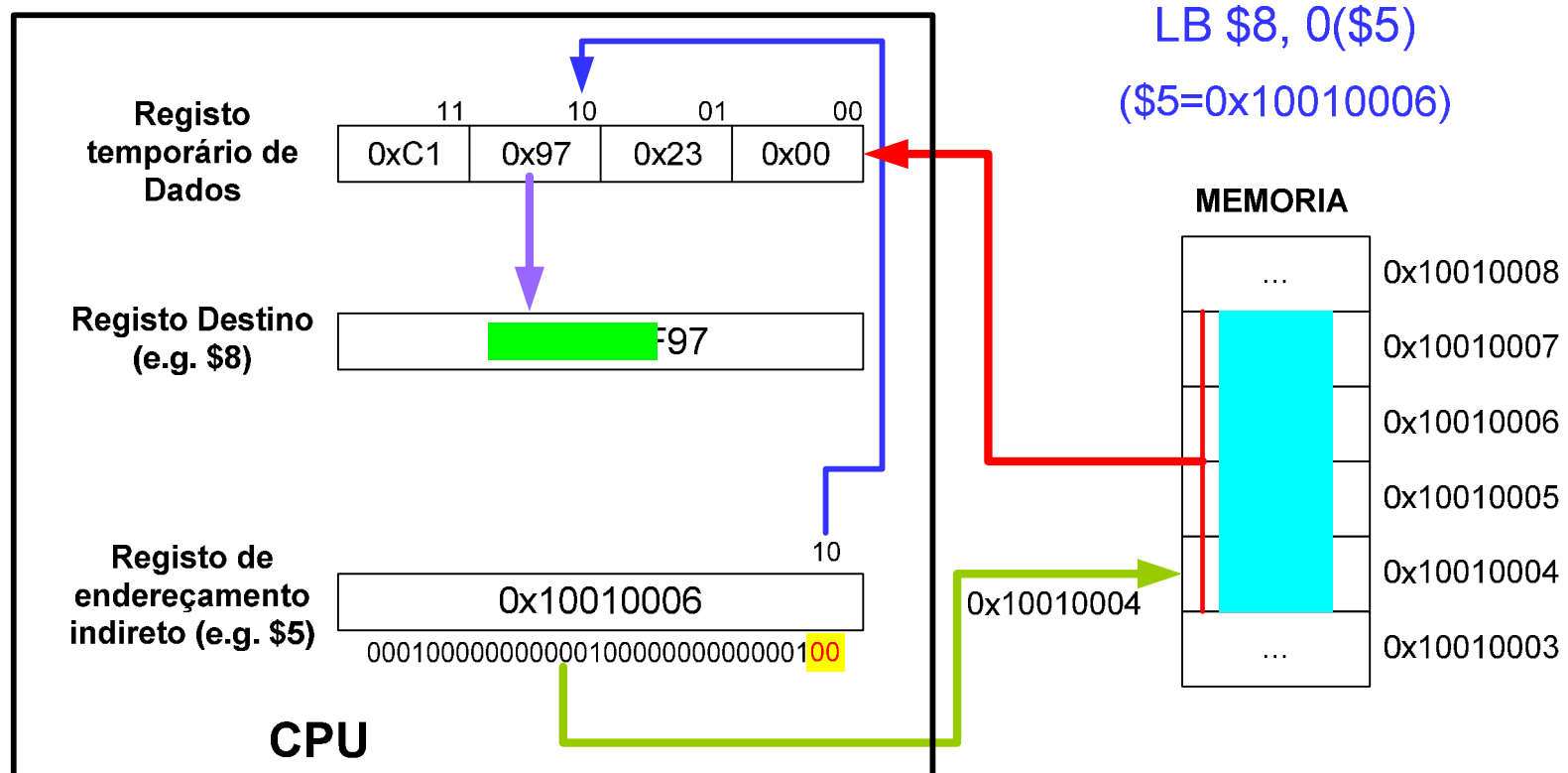
Exemplo:

lb \$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

Exemplo: leitura de 1 *byte* da memória

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

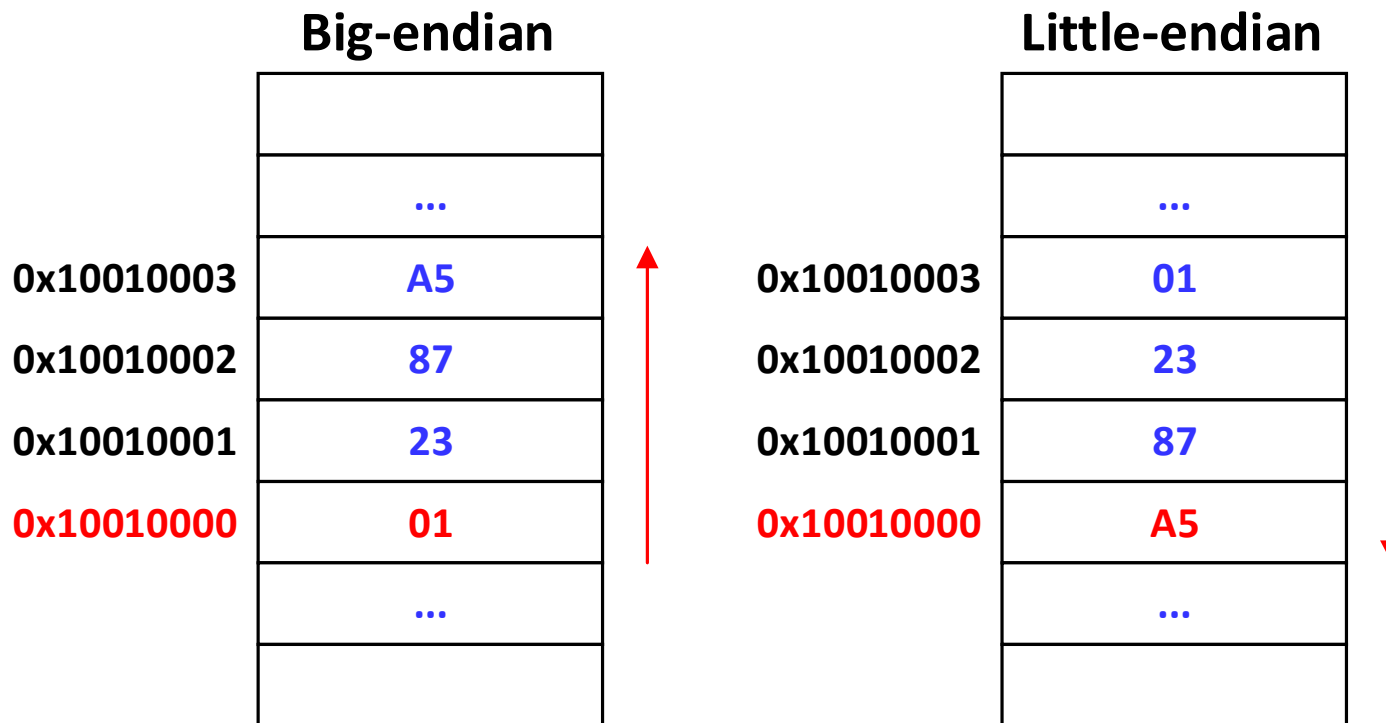


Organização da informação na memória

- 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 (formato **big-endian**)
 - *byte* menos significativo armazenado no endereço mais baixo da memória (formato **little-endian**)

Organização da informação na memória

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



- O simulador MARS (usado nas aulas práticas) implementa o formato "little-endian"