1º Semestre de 2007/2008

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Aula 5

Instruções de transferência de informação

- Instruções de tipo I *versus* instruções de tipo R

Organização de informação em memória:

- *little endian versus big endian*

Na aula anterior observamos o seguinte exemplo:

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

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

Note-se que este trecho de código faz uso apenas de registos internos do CPU

2. Instruções de transferência de informação (cont.)

E se pretendêssemos agora 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.
- Deverão existir, portanto, instruções para transferir informação entre os registos do CPU e os registos da memória externa

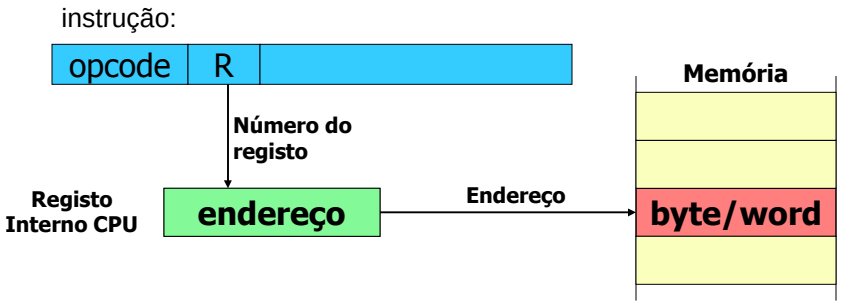
Como será então possível representar as instruções de acesso à memória externa (escrita e leitura), sabendo que as instruções do MIPS ocupam todas exactamente 32 bits?

Note-se que um endereço de memória é representado por 32 bits, pelo que ele sozinho ocuparia a totalidade da instrução

Solução: em vez do endereço, a instrução indica um registo que contém o endereço de memória a aceder (note-se que a dimensão do registo interno é 32 bits). Chama-se a este modo de endereçamento:

endereçamento indirecto a registo

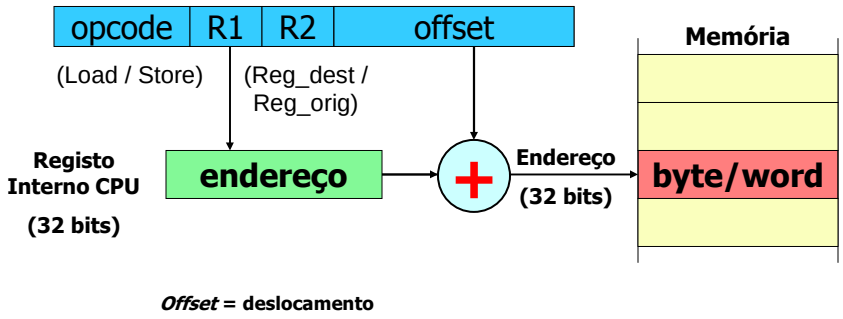
endereçamento indirecto a registo



A solução do MIPS:

endereçamento indirecto a registo com deslocamento

instrução:



Instrução de leitura da memória:

LW - (*load word*) transfere uma palavra de 32 bits da memória para um registo interno

Formato: **lw $reg_dest, const ($reg)**

registo destino

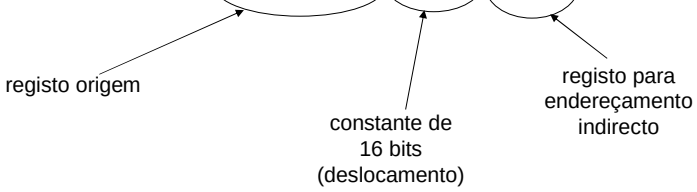
constante de 16 bits (deslocamento)

registo para endereçamento indirecto

Instrução de escrita na memória:

SW - (*store word*) transfere uma palavra de 32 bits de um registo interno para a memória

Formato: **sw $reg_orig, const ($reg)**



A existência de uma constante de 16 bits sugere que não será possível manter o formato das instruções anteriormente apresentadas para o caso da soma e subtracção.

Consideremos o seguinte exemplo:

g = h + A[5] (A é um *array* de *words – 32 bits)*

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

Usando instruções do *Assembly* do MIPS, a expressão anterior tomaria a seguinte forma:

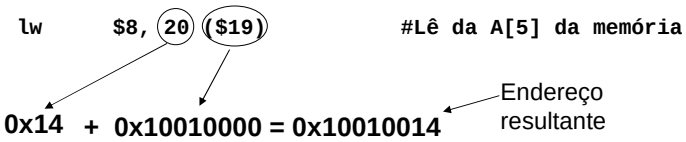
```
lw    $8, 20($19)     # Lê A[5] da memória
add   $17, $18, $8    # Calcula novo valor de g
```

Variável temporária (destino)

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:



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

Se pretendêssemos agora obter:

A[5] = h + A[5]

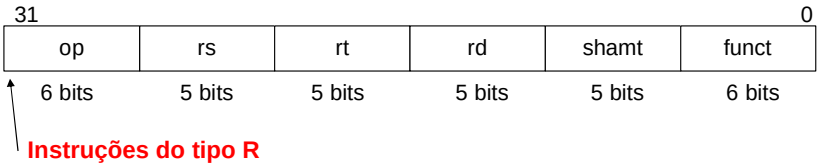
assumindo mais uma vez que ***h*** e o endereço inicial do *array* residem nos registos **$18** e **$19**, respectivamente

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]
```

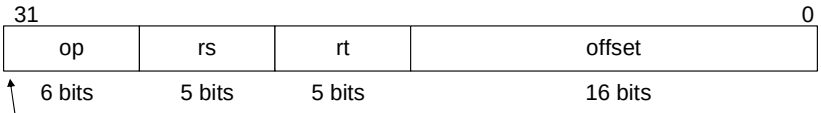
Arquitectura load/store: as operações só podem ser efectuadas sobre registos

Representação de instruções de soma e subtracção no MIPS:



Instruções do tipo R

Representação de instruções de transferência memória/registo no MIPS:

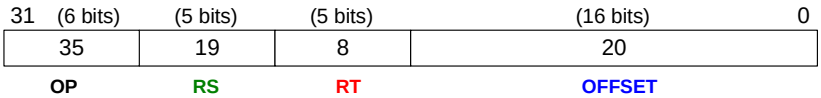


Instruções do tipo I

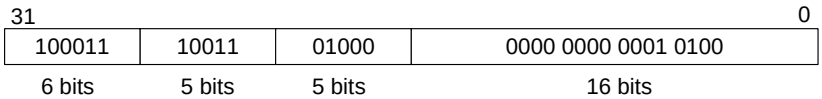
Relembrando uma instrução já nossa conhecida:

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

Corresponderia à seguinte instrução máquina:




```
LW    RT, OFFSET(RS)
```

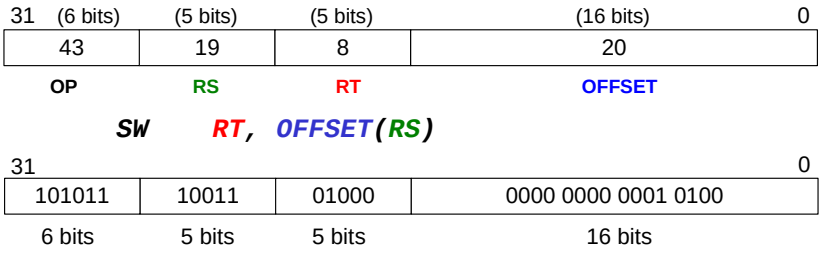


1000 1110 0110 1000 0000 0000 0001 0100_2 = 0x8E680014

Relembrando outra instrução já nossa conhecida:

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

Corresponderia à seguinte instrução máquina:



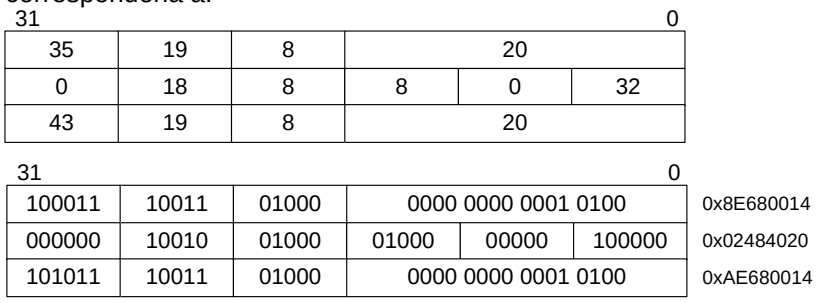
SW RT, OFFSET(RS)

1010 1110 0110 1000 0000 0000 0001 0100_2 = 0xAE680014

Enquanto que o trecho de 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]
```

corresponderia a:



Restrições de alinhamento nos endereços das variáveis

- Externamente o barramento de endereços do MIPS só tem disponíveis 30 bits ($A_{31}...A_2$), ou seja A_1 e A_0 não existem
- Assim, do ponto de vista externo, só são gerados endereços **múltiplos de $2^2 = 4$**
- **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 a memória é *byte-addressable*) ?

Restrições de alinhamento nos endereços das variáveis (cont.)

Resposta 1: 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 excepção, terminando aí a execução do programa

- Como se evita o problema ?
  - Resposta: garantindo que as variáveis do tipo ***word*** estão armazenadas num endereço múltiplo de 4
  - Directiva `.align n` do *Assembler* (alinhamento num endereço múltiplo de 2^n)

Restrições de alinhamento nos endereços das variáveis (cont.)

Questão 2: Como é possível a leitura/escrita de 1 byte de informação (uma vez que a memória é *byte-addressable*) ?

Resposta 2: 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?

Restrições de alinhamento nos endereços das variáveis (cont.)

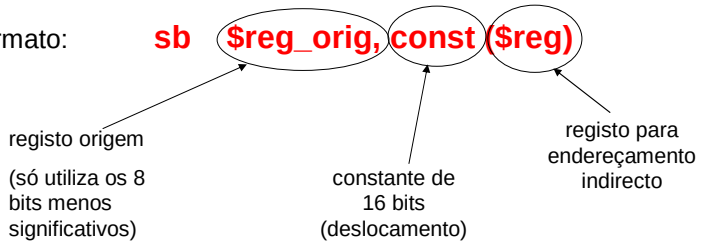
Resposta: o MIPS gera o endereço múltiplo de 4 (EM4) que inclui o endereço pretendido

- No caso de Leitura:
  - Executa uma instrução de leitura de **1 word** do endereço EM4, e retira os 8 bits do endereço pretendido
- No caso de Escrita: **(*Read Modify Write*)**
  - Executa uma instrução de leitura de **1 word** do endereço EM4
  - Substitui os 8 bits no endereço pretendido
  - Escreve a **word** modificada em EM4

Instrução de escrita de 1 byte na memória:

SB - (*store byte*) transfere um byte de um registo interno (os bits menos significativos) para a memória

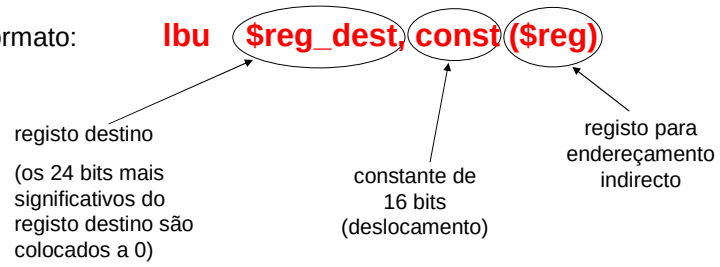
Formato: **sb $reg_orig, const ($reg)**



Instrução de leitura de 1 byte da memória:

LBU - (*load byte unsigned*) transfere um byte da memória para um registo interno

Formato: **lbu $reg_dest, const ($reg)**



Instrução de leitura de 1 byte da memória:

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

Formato: **lb $reg_dest, const ($reg)**

registo destino

(o bit 7 do valor lido é replicado nos 24 bits mais significativos do registo destino)

constante de 16 bits (deslocamento)

registo para endereçamento indirecto

Organização da informação em memória

Considere-se o valor: 0x012387A5

32 bits, logo, 4 bytes

numa memória do tipo *byte addressable*, qual a ordem de armazenamento dos *bytes*?

Como será ele armazenado na memória?

Exemplo – Valor a armazenar: 0x01 23 87 A5