

Aula 11

• Arquitectura de um multiplicador de inteiros

• Algoritmo de Booth para multiplicação de inteiros

• A multiplicação de inteiros no MIPS

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira e Silva

Arquitectura de um Multiplicador

• Devido ao aumento de complexidade que daí resulta, todas as arquitecturas suportam, ao nível do **hardware**, a capacidade para efectuar operações aritméticas de multiplicação

• No caso do MIPS, essas operações são asseguradas por unidade especial de multiplicação e divisão de inteiros

• Note-se que uma multiplicação que envolve **dois operandos de n bits** carece de um espaço de armazenamento, para o resultado, de **$2 \cdot n$ bits**

• Tal implica que o **resultado**, no caso do MIPS, deverá ser armazenado com **64 bits**, o que determina a existência de **registos especiais** para esse mesmo armazenamento

Arquitectura de um Multiplicador

A arquitectura de um multiplicador utiliza, em grande parte, o algoritmo da multiplicação que todos aprendemos na escola primária

Esse algoritmo tira partido da propriedade distributiva em relação à adição, permitindo que a multiplicação seja decomposta numa sucessão de somas de produtos parciais

Consideremos o seguinte produto, em que M representa o multiplicando e m o multiplicador **representados com 4 bits**

$$M \cdot m$$

$$M \cdot m = M \cdot (m_3 \cdot 2^3 + m_2 \cdot 2^2 + m_1 \cdot 2 + m_0)$$

$$\text{Logo: } M \cdot m = (M \cdot 2^3 \cdot m_3) + (M \cdot 2^2 \cdot m_2) + (M \cdot 2^1 \cdot m_1) + (M \cdot 2^0 \cdot m_0)$$

$$\text{Ou: } M \cdot m = ((M \cdot 2^3) \cdot m_3) + ((M \cdot 2^2) \cdot m_2) + ((M \cdot 2) \cdot m_1) + (M \cdot m_0)$$

Arquitectura de um Multiplicador

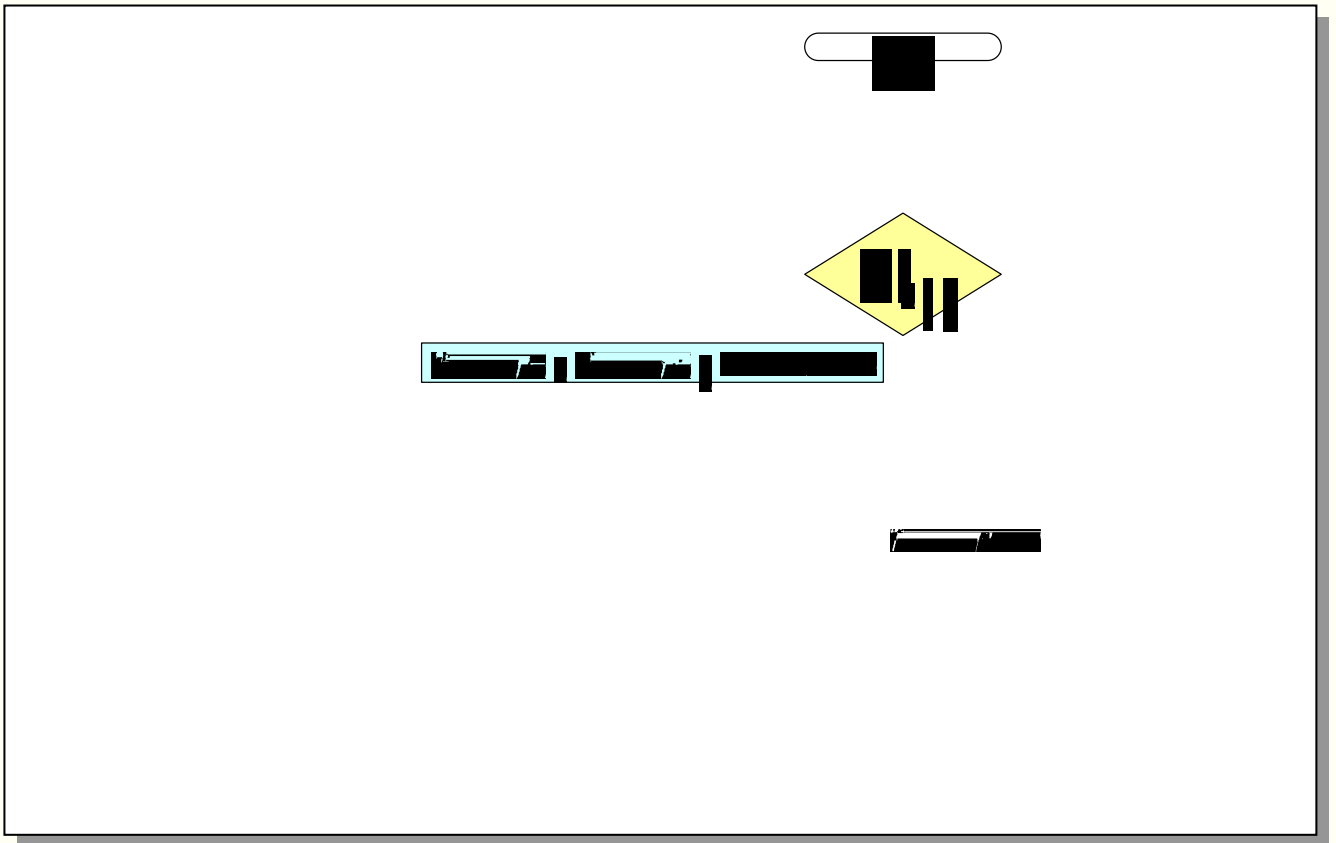
$$M \cdot m = ((M \cdot 2^3) \cdot m_3) + ((M \cdot 2^2) \cdot m_2) + ((M \cdot 2) \cdot m_1) + (M \cdot m_0)$$

Ora, multiplicar por dois (ou por uma potência de dois) corresponde a deslocar o número multiplicado à esquerda **shift (left)** tantos bits quantos a potência de dois envolvida

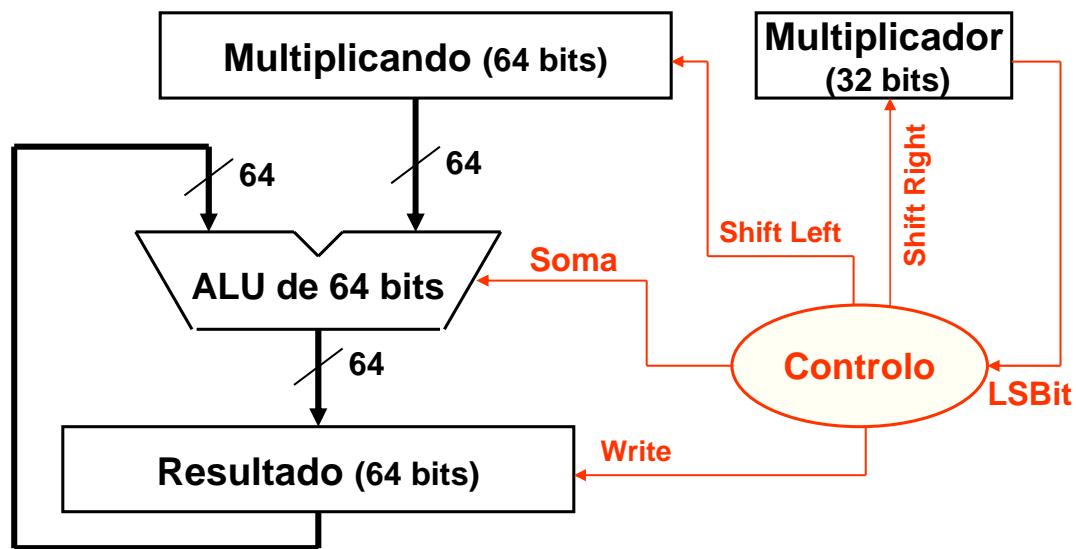
Por outro lado, se for igual a "0", o produto parcial correspondente também será zero, e se for "1", o mesmo produto parcial será igual ao multiplicando deslocado à esquerda de bits

$$\begin{array}{r}
 0101 \\
 \times 0110 \\
 \hline
 0000 \\
 01010 \\
 010100 \\
 + 0000000 \\
 \hline
 0011110
 \end{array}$$

Para cada bit a 1 do multiplicador, o multiplicando é adicionado ao resultado deslocado à esquerda de um número de bits igual ao seu índice do 1 do multiplicador



Arquitetura de um Multiplicador de 32 bits (1ª versão)



Nesta 1ª versão do multiplicador, os registos **Multiplicando** e **Resultado** operam com 64 bits.

Arquitetura de um Multiplicador

Vejamos novamente o exemplo anterior:

Por cada nova iteraç o obt m-se

o valor final de um novo bit do

resultado (as iterações seguintes correspondem a somar 0 a esse bit)

O mesmo efeito algorítmico pode

ser obtido se se deslocar o resultado para a direita, em vez de o multiplicando para a esquerda (o movimento relativo dos dois $\dot{\gamma}$, % o mesmo)

• O multiplicador continua a ser deslocado para a direita a cada iteração.

[illegible]

Arquitetura de um Multiplicador

Uma alternativa ao algoritmo inicial

será portanto deslocar o resultado

direita por cada nova iteração

Vantagens desta nova abordagem:

Para cada nova adição

suficiente operar apenas sobre

4 bits dos 8 bits do resultado

final

O registo utilizado para

armazenar o multiplicando

pode ter apenas 4 bits (em vez de 8 bits)

Neste exemplo não se está a considerar a possibilidade de ocorrência de carry nas sucessivas adições.

0 1 0 1
x 0 1 1 0

0 0 0 0	0 0 0 0	Res. Inicial
0 0 0 0		0.mdo
0 0 0 0	0 0 0 0	
0 0 0 0	0 0 0 0	Após >> 1
+ 0 1 0 1		1.mdo
0 1 0 1	0 0 0 0	
0 0 1 0	1 0 0 0	Após >> 1
+ 0 1 0 1		1.mdo
0 1 1 1	1 0 0 0	
0 0 1 1	1 1 0 0	Após >> 1
+ 0 0 0 0		0.mdo
0 0 1 1	1 1 0 0	
0 0 0 1	1 1 1 0	Após >> 1
Res. FINAL		

Arquitetura de um Multiplicador

No início, os 4 bits menos

significativos do resultado não têm

qualquer informação útil (vistos

preenchidos a cada nova iteração)

O sucessivo deslocamento

do resultado, acompanhado por

um deslocamento idêntico do

multiplicador

Assim, possível utilizar a p

menos significativa do resultado (4

bits no exemplo) para armazenar o

valor inicial do multiplicador

Optimiza-se, deste modo, o espaço

total de armazenamento necessário

nesta arquitetura de multiplicação

0 1 0 1
x 0 1 1 0

0 0 0 0	0 0 0 0	Res. Inicial
0 0 0 0		0.mdo
0 0 0 0	0 0 0 0	
0 0 0 0	0 0 0 0	Após >> 1
+ 0 1 0 1		1.mdo
0 1 0 1	0 0 0 0	
0 0 1 0	1 0 0 0	Após >> 1
+ 0 1 0 1		1.mdo
0 1 1 1	1 0 0 0	
0 0 1 1	1 1 0 0	Após >> 1
+ 0 0 0 0		0.mdo
0 0 1 1	1 1 0 0	
0 0 0 1	1 1 1 0	Após >> 1
Res. FINAL		

Arquitetura de um Multiplicador

Os 4 bits menos significativos do resultado no início do t_{i+1} em qual informação i (vendo sendo preenchidos a cada nova iteração).

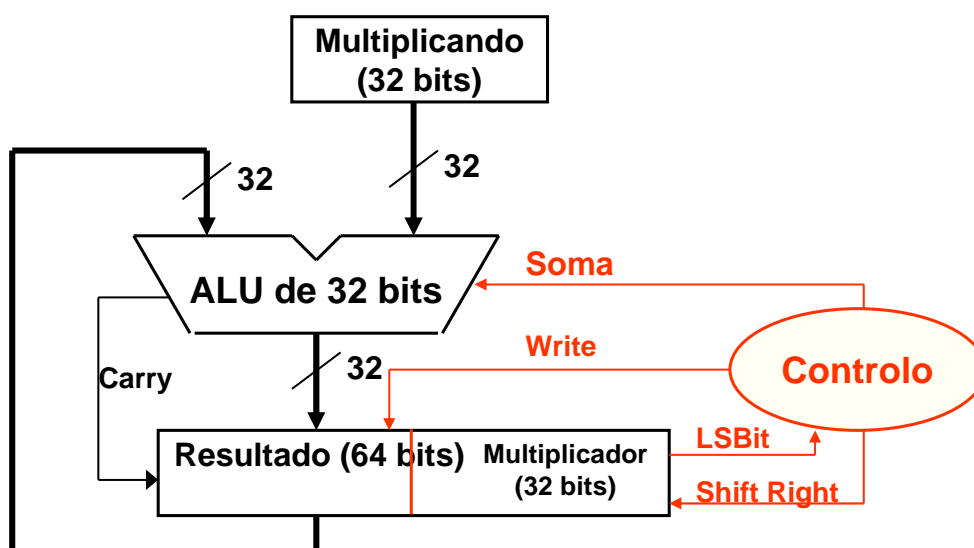
O sucessivo deslocamento à direita do resultado, acompanhado por um deslocamento idêntico do multiplicador

Assim, possível utilizar a parte menos significativa do resultado (4 bits no exemplo) para armazenar o valor inicial do multiplicador

Optimiza-se, deste modo, o espaço total de armazenamento necessário à arquitectura de multiplicação.

	0	1	0	1		
x	0	1	1	0		
igual	0	0	0	0	0	1
+	0	0	0	0		
o)	0	0	0	0	0	1
eita	0	0	0	0	0	1
+	0	1	0	1		
	0	1	0	1	0	0
+	0	0	0	0		
parte	0	1	0	1	1	0
+	0	1	0	1		
	0	1	1	1	1	0
+	0	0	0	0		
%oO	0	0	1	1	1	1
	0	0	0	1	1	1

Arquitetura de um Multiplicador de 32 bits (versão otimizada)



Na versão otimizada do multiplicador, o registro **Multiplicando** e a **ALU** passam a ser de 32 bits. O registro **Multiplicador** desaparece, sendo substituído pela parte menos significativa do **Resultado**.

Arquitetura de um Multiplicador (Algoritmo de Booth)

A arquitetura de multiplicação vista anteriormente pode operar correctamente quantidades inteiras consideradas sem sinal, ou seja, uma arquitectura til para a realização de operações **unsigned**

A arquitectura que utiliza **algoritmo de Booth** é adequada para a realização de operações **signed** (i.e., em que os operandos estão codificados em complemento para dois)

Qualquer inteiro em base dois pode ser **decomposto** a partir da observação de **cadeias par adjacente de bits**, na forma de uma sequência de somas e subtrações, de acordo com as seguintes regras

$b_i, b_{i-1} = 00$ ou 11 - não contribui para a expressão

$b_i, b_{i-1} = 01$ - soma 2^i

$b_i, b_{i-1} = 10$ - subtrai 2^i

b_i é bit na posição i

b_{i-1} é bit na posição $i-1$

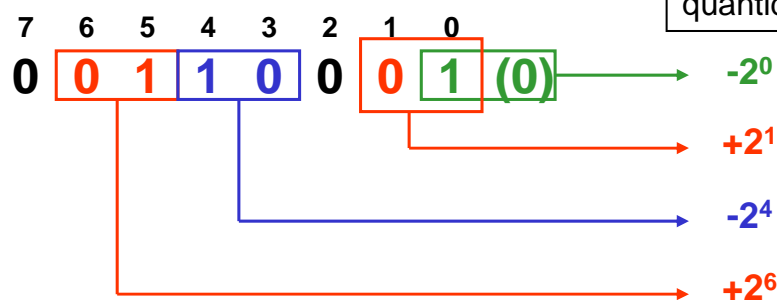
Arquitetura de um Multiplicador (Algoritmo de Booth)

$b_i, b_{i-1} = 00$ ou 11 - não contribui para a expressão

$b_i, b_{i-1} = 01$ - soma 2^i

$b_i, b_{i-1} = 10$ - subtrai 2^i

Exemplo: $49_{10} = 31_{16} = 00110001_2$



Exercício: factorize, de acordo com o algoritmo de Booth, a quantidade 11111110_2

$$N = 2^6 - 2^4 + 2^1 - 2^0 = 64 - 16 + 2 - 1 = 49$$

Arquitetura de um Multiplicador (Algoritmo de Booth)

Vejamos agora um exemplo aplicado ao multiplicar:

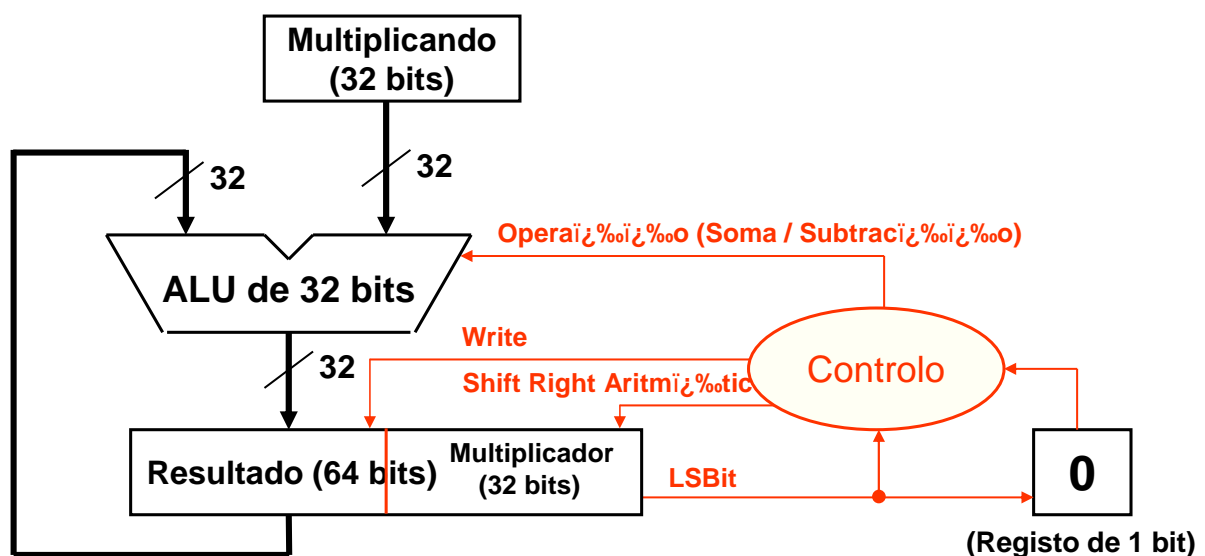
$$(0011_2) \mid (1010_2) = 3_{10} \mid (-6_{10}) = -18_{10} = 11101110_2$$

$$(0011_2) \mid (1010_2) = (0011) \mid (-2^1 + 2^2 - 2^3)$$

$$= -(0011 \mid 2^1) + (0011 \mid 2^2) - (0011 \mid 2^3)$$

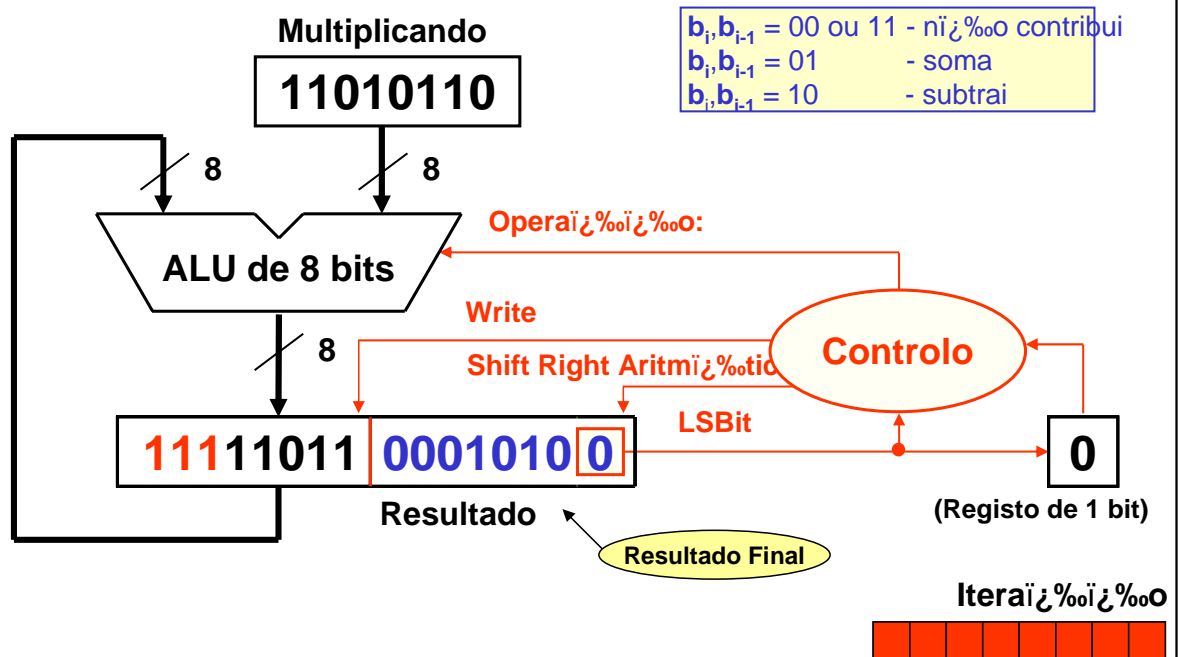
$$\begin{array}{r} 0011 \\ \times 1010 \\ \hline 0000 \\ - 00110 \\ + 001100 \\ - 0011000 \\ \hline 11101110 \end{array}$$

Arquitetura de um Multiplicador de 32 bits (Algoritmo de Booth)



Arquitetura de um Multiplicador (Algoritmo de Booth)

(exemplo c/ operandos de 8 bits: 00011110 x 11010110 = 11111011 00010100)



Arquitetura de um Multiplicador (Algoritmo de Booth)

Porque funciona o algoritmo de Booth para quantidades *signed*?

Se exprimirmos a observação de cada par de bits do multiplicador na forma de uma diferença entre eles:

$$(m_{i-1} - m_i) \cdot 2^i$$

o resultado da expressão pode ser usado da seguinte forma:

- 0 - não fazer nada
- +1 - somar o multiplicando
- 1 - subtrair o multiplicando

$$b_i, b_{i-1} = 00 \text{ ou } 11 - \text{ não contribui}$$

$$b_i, b_{i-1} = 01 - \text{ soma}$$

$$b_i, b_{i-1} = 10 - \text{ subtrai}$$

O produto pode assim ser expresso na seguinte forma:

$$(m_{-1} - m_0) \cdot M \cdot 2^0 + (m_0 - m_1) \cdot M \cdot 2^1 + (m_1 - m_2) \cdot M \cdot 2^2 + \dots + (m_{29} - m_{30}) \cdot M \cdot 2^{30} + (m_{30} - m_{31}) \cdot M \cdot 2^{31}$$

Arquitectura de um Multiplicador (Algoritmo de Booth)

O produto pode assim ser expresso na seguinte forma:

$$M.[(m_{-1}-m_0).2^0 + (m_0-m_1).2^1 + (m_1-m_2).2^2 + \dots + (m_{29}-m_{30}).2^{30} + (m_{30}-m_{31}).2^{31}]$$

$-m_{30}.2^{30} + m_{30}.2^{31}$

Note-se, contudo que:

$$-m_i.2^i + m_i.2^{i+1} = -m_i.2^i + 2.m_i.2^i = (-m_i + 2.m_i).2^i = m_i.2^i$$

A expressão anterior pode assim ser reduzida a:

$$M.[-(m_{31}.2^{31}) + (m_{30}.2^{30}) + \dots + (m_1.2^1) + (m_0.2^0)]$$

Representa-se em complemento para dois de m_i

A Multiplicação de inteiros no MIPS

No MIPS, a multiplicação é assegurada por uma estrutura semelhante à anteriormente descrita

Para o armazenamento do multiplicador e do resultado final, os arquitectos do MIPS incluíram um par de registos especiais designados, respectivamente, por **HI** e **LO** (de uso específico da unidade de multiplicação de inteiros):

o registo **HI** armazena os **32 bits mais significativos do resultado**

o registo **LO** armazena, inicialmente, o multiplicador e, após a execução da operação, os **32 bits menos significativos do resultado**

A transferência do multiplicador para o registo **LO** é automática (por *hardware*) no início da execução da operação

A Multiplicação de inteiros no MIPS

Em *Assembly*, a multiplicação é efectuada pela instrução

mult \$reg1, \$reg2 # **Multiply (signed)**
multu \$reg1, \$reg2 # **Multiply unsigned**

em que \$reg1 é o multiplicando e \$reg2 o multiplicador. O **resultado** fica armazenado nos **registos HI e LO**

A **transferência** de informação entre os registos **HI e LO** e os restantes **registos de uso geral** faz-se através das instruções:

mfhi \$reg # **move from hi** - Copia HI para \$reg
mflo \$reg # **move from lo** - Copia LO para \$reg
mthi \$reg # **move to hi** - Copia \$reg para HI
mtlo \$reg # **move to lo** - Copia \$reg para LO