Aula 11

- Arquitectura de um multiplicador de inteiros
- Algoritmo de Booth para multiplicação de inteiros com sinal
- A multiplicação de inteiros no MIPS

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira, Tomás Oliveira e Silva

Arquitectura de um Multiplicador

- Devido ao aumento de complexidade que daí resulta, nem todas as arquitecturas suportam, ao nível do *hardware*, a capacidade para efectuar operações aritméticas de multiplicação e divisão
- No caso do MIPS, essas operações são asseguradas por uma unidade especial de multiplicação e divisão de inteiros
- Note-se que uma multiplicação que envolva **dois operandos de *n* bits** carece de um espaço de armazenamento, para o resultado, de **2*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 a usar 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**

$$M \cdot m$$

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

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

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 m_n 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 ***n*** bits

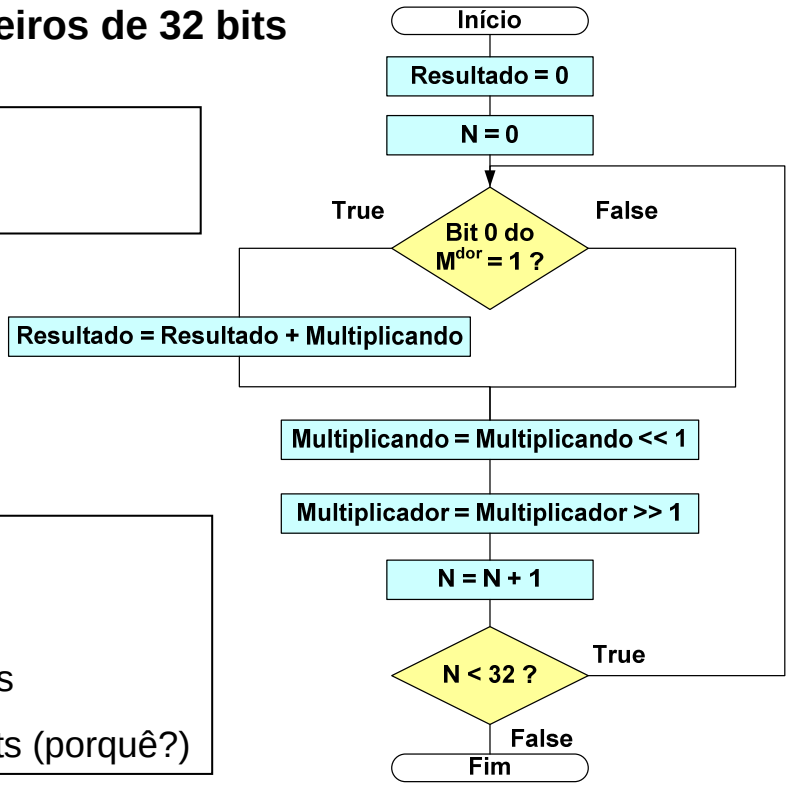
```
  0101
x0110
 0000
 01010
 010100
+0000000
 0011110
```

Para cada bit a "1" do multiplicador, o multiplicando é adicionado ao resultado deslocado à esquerda de um nº de bits igual à posição do "1" do multiplicador

Multiplicação de inteiros de 32 bits

Operandos: 32 bits

Resultado: 64 bits



Registos necessários:

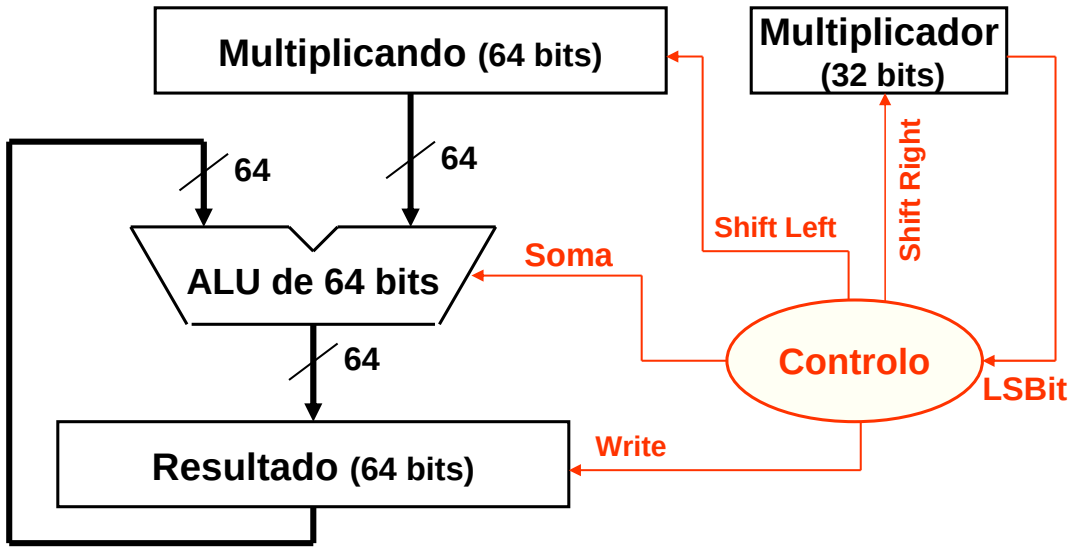
- **Resultado**: 64 bits
- **Multiplicador**: 32 bits
- **Multiplicando**: 64 bits (porquê?)

Multiplicação de inteiros – exemplo com 4 bits

- No exemplo anterior (**0101**x**0110** - os operandos são de 4 bits) o resultado terá uma dimensão máxima de 8 bits
- Para a implementação de um multiplicador de 4 bits, que aplique o algoritmo do slide anterior, os registos necessários são:
  - **resultado**: registo de 8 bits
  - **multiplicando**: registo de 8 bits (inicialmente os bits mais significativos são colocados a 0)
  - **multiplicador**: registo de 4 bits

				0	1	0	1	
				x 0	1	1	0	
	0	0	0	0	0	0	0	Res. Inicial
+	0	0	0	0	0	0	0	0.mdo.2^0
	0	0	0	0	0	0	0	
+	0	0	0	0	1	0	1	0 1.mdo.2^1
	0	0	0	0	1	0	1	0
+	0	0	0	1	0	1	0	0 1.mdo.2^2
	0	0	0	1	1	1	1	0
+	0	0	0	0	0	0	0	0 0.mdo.2^3
	0	0	0	1	1	1	1	0
	0	0	0	1	1	1	1	0 Res. FINAL

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



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

Arquitectura de um Multiplicador

Vejamos novamente o exemplo anterior:

- Por cada nova iteração obtém-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 é o mesmo)
- O multiplicador continua a ser deslocado à direita a cada iteração

					0	1	0	1
					x 0	1	1	0
	0	0	0	0	0	0	0	0	Res. Inicial
+	0	0	0	0	0	0	0	0	0.mdo.2^0
	0	0	0	0	0	0	0	0
+	0	0	0	0	1	0	1	0	1.mdo.2^1
	0	0	0	0	1	0	1	0
+	0	0	0	1	0	1	0	0	1.mdo.2^2
	0	0	0	1	1	1	1	0
+	0	0	0	0	0	0	0	0	0.mdo.2^3
	0	0	0	1	1	1	1	0
	0	0	0	1	1	1	1	0	Res. FINAL

Arquitectura de um Multiplicador

- Uma alternativa ao algoritmo inicial será portanto deslocar o resultado à direita por cada nova iteração
- Vantagens desta nova abordagem:
  - Para cada nova adição é 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
```


Arquitectura de um Multiplicador

- No início, os 4 bits menos significativos do resultado não têm qualquer informação útil (vão sendo preenchidos a cada nova iteração)
- O sucessivo deslocamento à direita do resultado, é acompanhado por um deslocamento idêntico do multiplicador
- É, 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

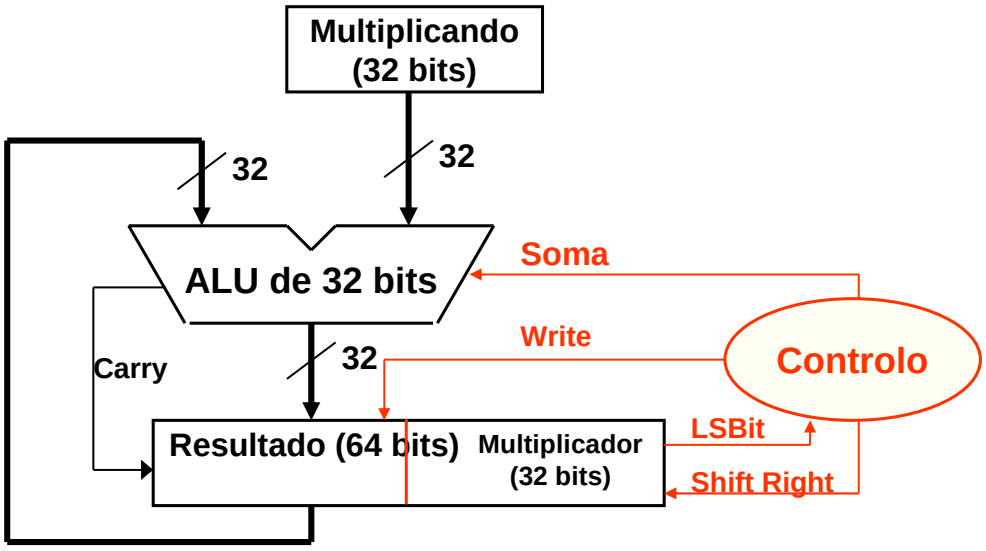
```
   0 1 0 1
 x 0 1 1 0
-----------
   0 0 0 0 | 0 0 0 0   Res. Inicial
 + 0 0 0 0 |           0.mdo
-----------
   0 0 0 0 | 0 0 0 0
   0 0 0 0 | 0 0 0 0   Após >> 1
 + 0 1 0 1 |           1.mdo
-----------
   0 1 0 1 | 0 0 0 0
   0 0 1 0 | 1 0 0 0   Após >> 1
 + 0 1 0 1 |           1.mdo
-----------
   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
```

Arquitectura de um Multiplicador

- Os 4 bits menos significativos do resultado no início não têm qualquer informação útil (vão sendo preenchidos a cada nova iteração)
- O sucessivo deslocamento à direita do resultado, é acompanhado por um deslocamento idêntico do multiplicador
- É, 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
-------------------------------------
  0 0 0 0 | 0 1 1 0   Res. Inicial
+ 0 0 0 0 |           0.mdo
-------------------------------------
  0 0 0 0 | 0 1 1 0
  0 0 0 0 | 0 0 1 1   Após >> 1
+ 0 1 0 1 |           1.mdo
-------------------------------------
  0 1 0 1 | 0 0 1 1
  0 0 1 0 | 1 0 0 1   Após >> 1
+ 0 1 0 1 |           1.mdo
-------------------------------------
  0 1 1 1 | 1 0 0 1
  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
```


Arquitectura de um Multiplicador de 32 bits (versão optimizada)



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

Arquitectura de um Multiplicador (Algoritmo de Booth)

- A arquitectura de multiplicação vista anteriormente apenas pode operar correctamente quantidades inteiras consideradas sem sinal, ou seja, é uma arquitectura útil para a realização de operações de **multiplicação *unsigned***
- A arquitectura que utiliza o **algoritmo de Booth** é adequada para a realização de operações de **multiplicação *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 **cada par adjacente de bits**, na forma de uma sequência de somas e subtracções, 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

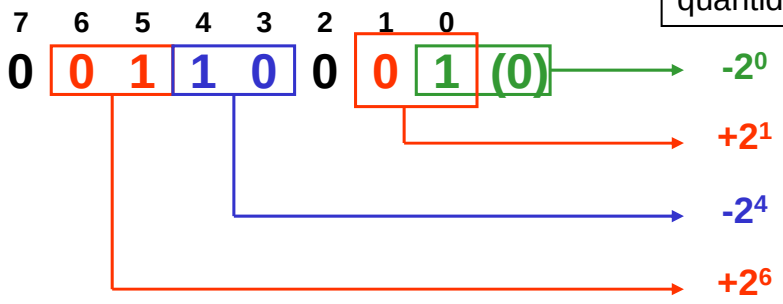
b_i – bit na posição **i**
b_{i-1} – bit na posição **i-1**

Arquitectura 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: $\mathbf{49_{10}} = 31_{16} = \mathbf{00110001_2}$

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



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

Arquitectura de um Multiplicador (Algoritmo de Booth)

Vejamos agora um exemplo aplicado à multiplicação:

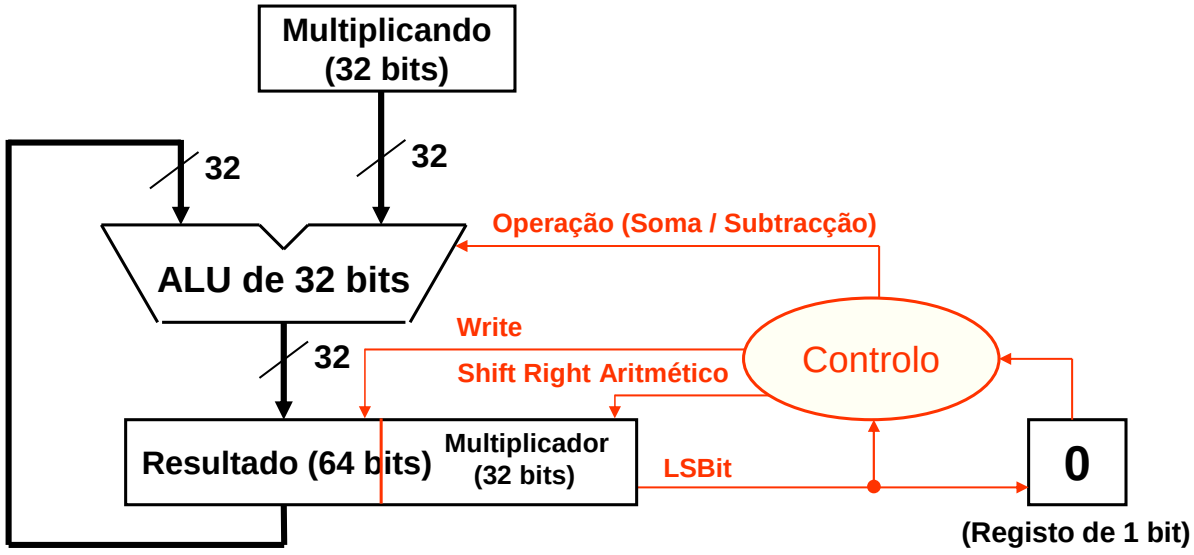
$(0011_2) \times (1010_2) = 3_{10} \times -6_{10} = -18_{10} = 11101110_2$

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

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

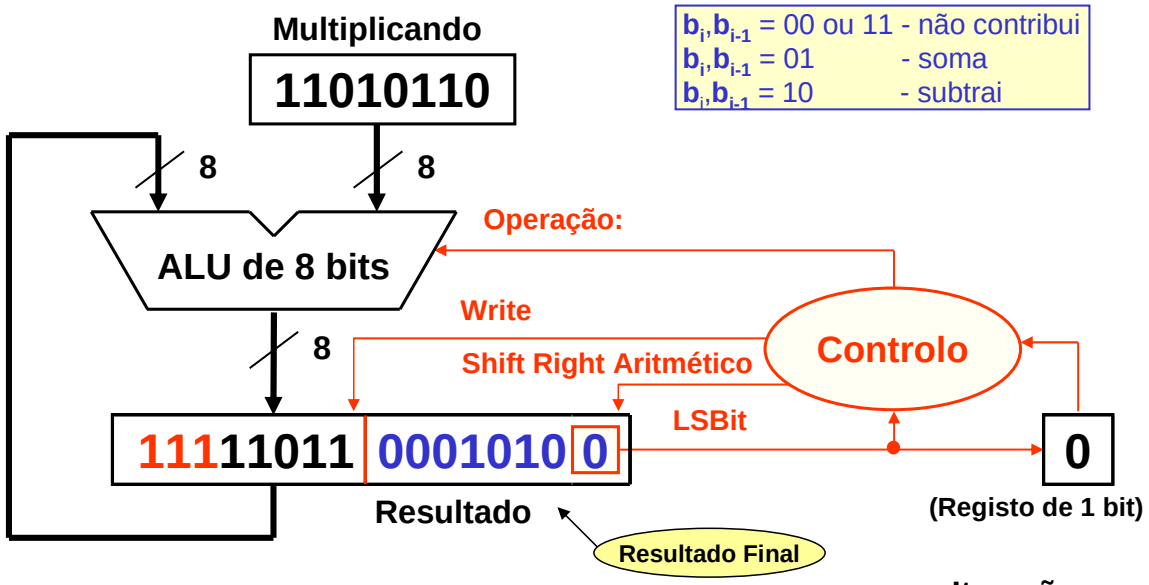
```
   0011
  x1010
  ----
   0000
 -  00110
 +  001100
 -  0011000
  --------
  11101110
```


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



Arquitectura de um Multiplicador (Algoritmo de Booth)

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



Arquitectura 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:

$$(m_{i-1} - m_i)$$

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 ou 11 - não contribui
b_i,b_{i-1} = 01 - soma
b_i,b_{i-1} = 10 - subtrai

O produto pode assim ser expresso na seguinte forma:

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

Arquitectura de um Multiplicador (Algoritmo de Booth)

O produto pode assim ser expresso na seguinte forma:

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

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

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:

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

Representação em complemento para dois de "m"

A Multiplicação de inteiros no MIPS

- No MIPS, a multiplicação é assegurada por uma arquitectura semelhante à anteriormente descrita
- Para o armazenamento do multiplicador e do resultado final, os arquitectos do MIPS incluiram 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 efectuada automaticamente (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
```