1º Semestre de 2007/2008

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Aula 12

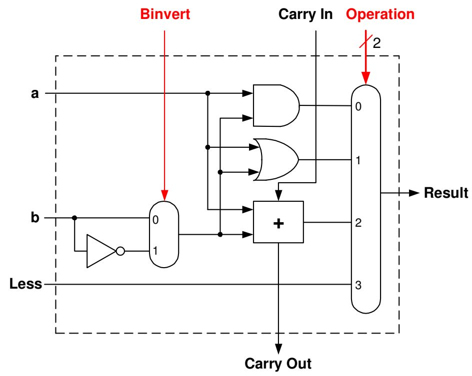
Construção de uma ALU básica de 1 bit

Expansão da ALU para 32 bits

Arquitectura de um multiplicador de inteiros

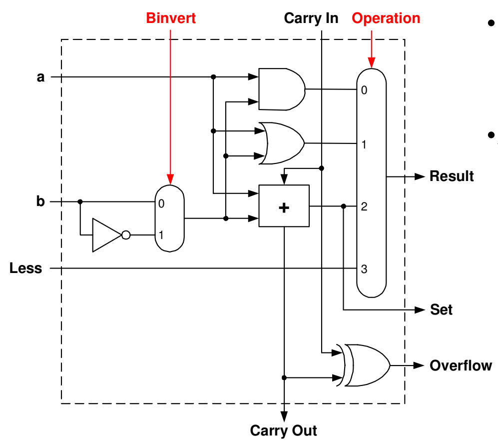
Algoritmo de Booth para multiplicação de inteiros com sinal

Construção de uma ALU básica de 1 bit:



- Esta ALU permite efectuar as operações aritméticas de soma e subtracção, operações lógicas AND, OR e NOT
- A operação é seleccionada pelo sinal *Operation* (2 bits: 00 – AND, 01 – OR, 10 – ADD, 11 – SLT)
- A subtracção obtém-se colocando um "1" em *Binvert* e *Carry In*

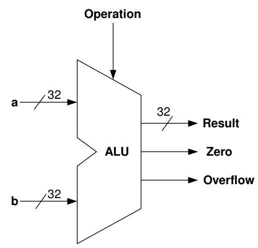
ALU básica de 1 bit, com detecção de *overflow*:



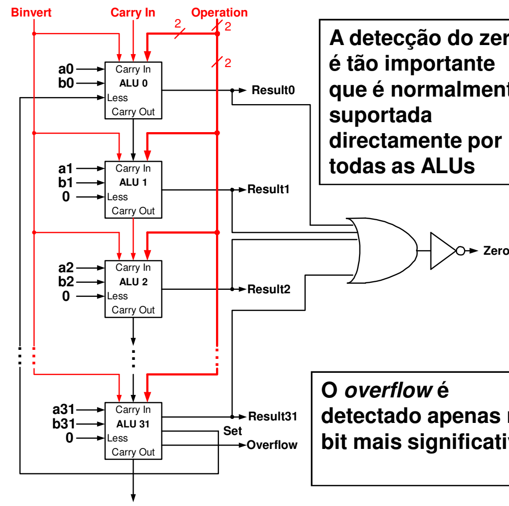
- Esta ALU permite ainda efectuar a operação SLT (*set if less than*)
- A operação SLT é realizada através da operação (a-b):
  - saída "Set" =1 se a < b
  - saída "Set"=0 se a >= b

Arquitectura de Computadores I 2007/08

Expansão para 32 bits em *ripple carry*:



Representação alternativa e simplificada de uma ALU de 32 bits



A detecção do zero é tão importante que é normalmente suportada directamente por todas as ALUs

O *overflow* é detectado apenas no bit mais significativo

Arquitectura de Computadores I 2007/08

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
- Note-se que uma multiplicação que envolva dois operandos de *n* bits carece de um espaço de armazenamento de 2*n bits.
- No caso do MIPS, essas operações são asseguradas directamente pela ALU. Tal, implica que o resultado 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 replica 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

$$\mathbf{M \cdot m}$$

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

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

Arquitectura de um Multiplicador

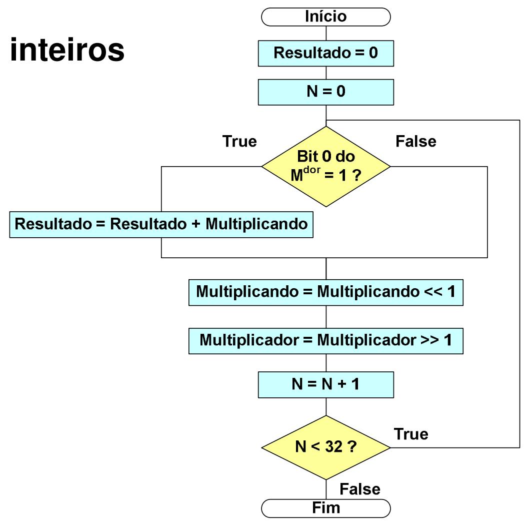
$\mathbf{M \cdot m = ((M \cdot 2^3) \cdot m_3) + ((M \cdot 2^2) \cdot m_2) + ((M \cdot 2) \cdot m_1) + (M \cdot m_0)}$

- Ora, multiplicar por dois (ou por uma potência de dois) corresponde a deslocar o número multiplicado à esquerda (*shift*) tantos bits quantos a potência de dois envolvida.
- Por outro lado, se $\mathbf{m_n}$ 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.

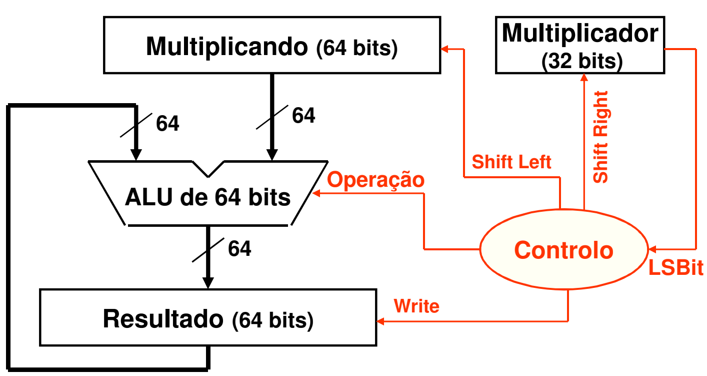
```
    0101
   x0110
   -----
    0000
   01010
  010100
+0000000
--------
 0011110
```

Para cada bit "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



Arquitectura de um Multiplicador (1ª versão)



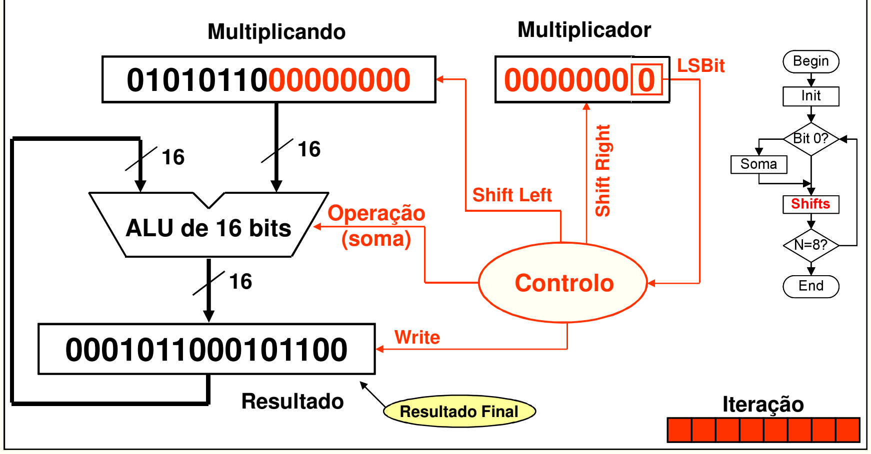
Nesta 1ª versão do multiplicador, a ***ALU*** e os registos ***Multiplicando*** e ***Resultado*** operam com 64 bits.

Consideremos o caso da multiplicação de 86 por 66, cujo produto é 5676:

```
         01010110
       x 01000010
       ----------
         00000000
      + 01010110
      -----------
        010101100
  + 01010110
  ----------------
  0001011000101100
```


Arquitectura de um Multiplicador (1ª versão)

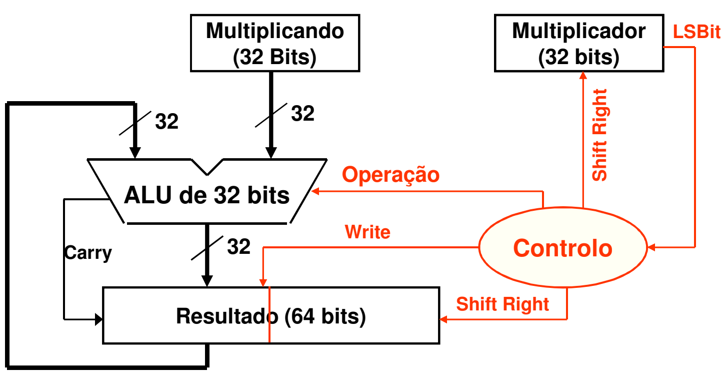
(exemplo c/ operandos de 8 bits: 01000010 x 01010110 = 00010110 00101100)



Arquitectura de um Multiplicador

- O movimento relativo (*shift*) entre *Multiplicando* e *Resultado* indicia que o mesmo resultado poderia ser obtido se, em vez de deslocar para a esquerda o conteúdo do *Multiplicando,* em cada nova iteração do algoritmo deslocássemos para a direita o conteúdo do registo *Resultado*.
- Por outro lado, verifica-se que em cada nova iteração se obtém o valor final de um novo bit do resultado, começando pelo menos significativo. Isto leva-nos a concluir que para cada nova adição é suficiente operar sobre apenas 32 bits dos 64 bits do resultado final.

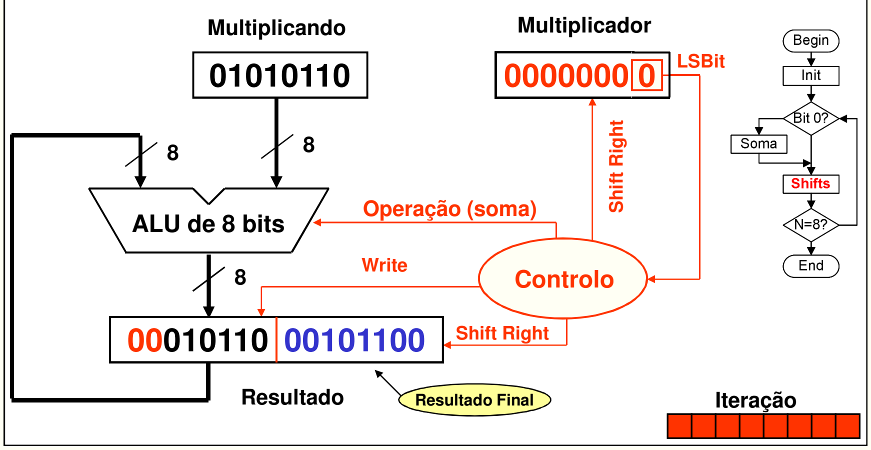
Arquitectura de um Multiplicador (2ª versão)



Nesta 2ª versão do multiplicador, a ***ALU*** e o registo ***Multiplicando*** operam apenas com 32 bits.

Arquitectura de um Multiplicador (2ª versão)

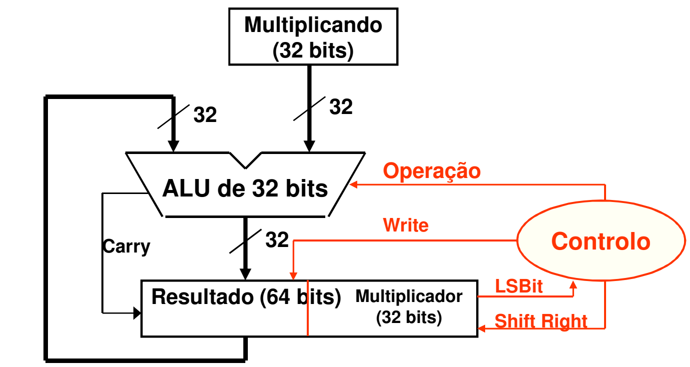
(exemplo c/ operandos de 8 bits: 01000010 x 01010110 = 00010110 00101100)



Arquitectura de um Multiplicador

- Finalmente, pode verificar-se que o deslocamento à direita do conteúdo do registo *Resultado*, é acompanhado por um deslocamento idêntico do registo *Multiplicador*.
- Uma vez que por cada novo bit acrescentado a *Resultado*, se perde um à direita do *Multiplicador*, é possível utilizar a parte menos significativa de *Resultado* para armazenar o valor inicial do *Multiplicador*, optimizando assim o espaço total de armazenamento necessário à arquitectura de multiplicação.

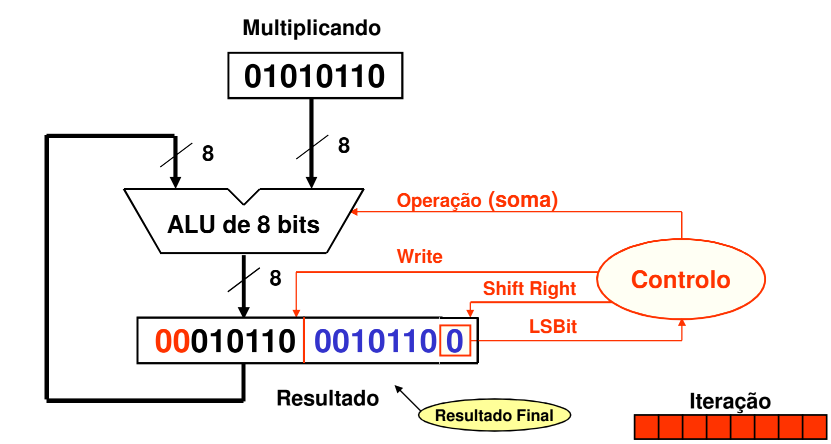
Arquitectura de um Multiplicador (versão final)



Na versão final do multiplicador, o registo ***Multiplicador*** desaparece, sendo substituído pela parte menos significativa do ***Resultado***.

Arquitectura de um Multiplicador (versão final)

(exemplo c/ operandos de 8 bits: 01000010 x 01010110 = 00010110 00101100)



Arquitectura de um Multiplicador (Algoritmo de Booth)

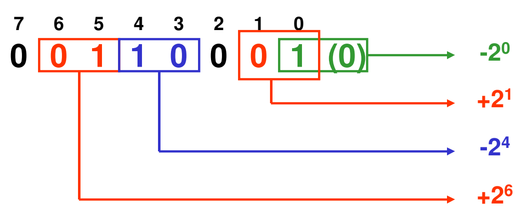
Verifica-se que qualquer inteiro em base dois pode ser factorizado a partir da observação de cada par de bits, na forma de uma sequência de somas e subtracções, de acordo com as seguintes regras:

$\mathbf{d_i,d_{i-1}}$ = 00 ou 11	- não contribui para a expressão
$\mathbf{d_i,d_{i-1}}$ = 01	- soma 2^i
$\mathbf{d_i,d_{i-1}}$ = 10	- subtrai 2^i

Arquitectura de um Multiplicador (Algoritmo de Booth)

d_i,d_{i-1} = 00 ou 11	- não contribui para a expressão
d_i,d_{i-1} = 01	- soma 2^i
d_i,d_{i-1} = 10	- subtrai 2^i

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



$$\mathbf{N = 2^6 - 2^4 + 2^1 - 2^0 = 64 - 16 + 2 - 1 = 49}$$

Arquitectura de Computadores I 2007/08

Arquitectura de um Multiplicador (Algoritmo de Booth)

Vejamos agora um exemplo aplicado à multiplicação:

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

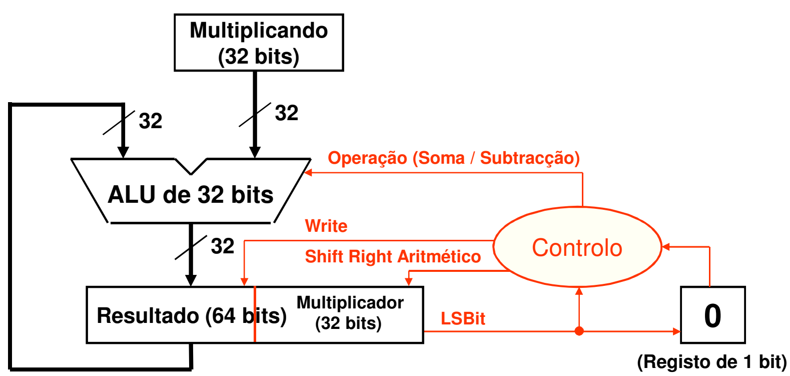
$(0011_2) . (1010_2) = (0011) . (-2^1 + 2^2 - 2^3)$

$= \underbrace{-(2^1 . 0011)} + \underbrace{(2^2 . 0011)} - \underbrace{(2^3 . 0011)}$

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

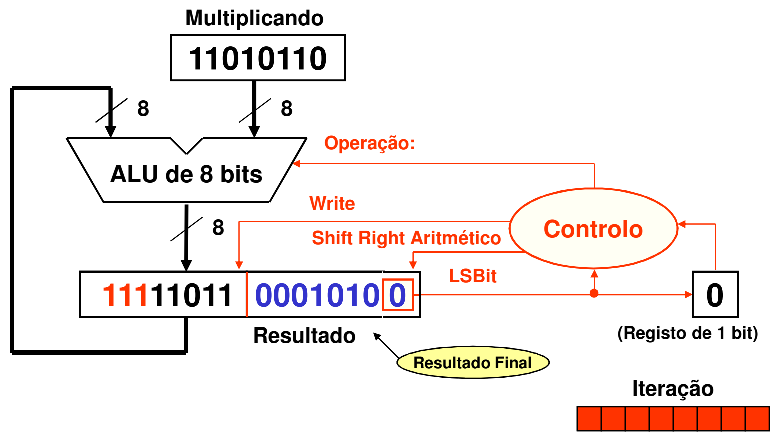

Arquitectura de Computadores I 2007/08

Arquitectura de um Multiplicador (Booth)



Arquitectura de um Multiplicador (Booth)

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



Arquitectura de um Multiplicador (Algoritmo de Booth)

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

$$(a_{i-1} - a_i)$$

o resultado da expressão pode ser usado da seguinte forma:

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

O produto pode assim ser expresso na seguinte forma:

$(a_{-1}-a_0).M.2^0+ (a_0-a_1).M.2^1+ (a_1-a_2).M.2^2+ ...+(a_{29}-a_{30}).M.2^{30}+ (a_{30}-a_{31})-M.2^{31}$

Arquitectura de um Multiplicador (Algoritmo de Booth)

O produto pode assim ser expresso na seguinte forma:

$$M.[(a_{-1}-a_0).2^0 + (a_0-a_1).2^1 + (a_1-a_2).2^2 + ... + \underbrace{(a_{29}-a_{30}).2^{30} + (a_{30}-a_{31}).2^{31}}_{-a_{30}x2^{30} + a_{30}x2^{31}}]$$

Note-se, contudo que:

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

A expressão anterior pode assim ser reduzida a:

$$M\ .\ \boxed{((a_{31}.-2^{31}) + (a_{30}.2^{30}) + ... + (a_1.2^1) + (a_0.2^0))}$$

Representação em complemento para dois de “a”

A Multiplicação de inteiros no MIPS

- No MIPS, a multiplicação é assegurada por uma arquitectura semelhante à anteriormente descrita. A única particularidade, para além da unidade de controlo, resulta da necessidade de existir um registo de 64 bits para armazenar o multiplicador e o resultado final.
- A solução encontrada pelos arquitectos do MIPS consistiu na inclusão de um par de registos especiais, designados respectivamente por **HI** e **LO**, e que, em conjunto, formam o aludido registo de 64 bits:
  - o registo **HI** armazena os **32 bits mais significativos do resultado**
  - o registo **LO** armazena os **32 bits menos significativos do resultado**

A Multiplicação de inteiros no MIPS

Em *Assembly*, a multiplicação é efectuada pela instrução

```
mult  $reg1, $reg2     # Multiply
multu $reg1, $reg2     # Multiply unsigned
```

em que $reg1 é o multiplicando e $reg2 o multiplicador. O resultado, como se disse, 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
```