Aula 10

- Arquitetura de um multiplicador de inteiros
- A multiplicação de inteiros no MIPS

- Arquitetura de um divisor de inteiros
- Divisão de inteiros com sinal
- Divisão de inteiros no MIPS

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

Arquitetura de um Multiplicador

- Devido ao aumento de complexidade que daí resulta, nem todas as arquiteturas suportam, ao nível do *hardware*, a capacidade para efetuar operações aritméticas de multiplicação e divisão de inteiros
- 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

Arquitetura de um Multiplicador

- A arquitetura de um multiplicador utiliza, em grande parte, o algoritmo da multiplicação que todos aprendemos a usar na escola primária

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

  - *Para além da solução iterativa estudada nesta aula é também possível construir multiplicadores combinatórios, tal como foi estudado em Sistemas Digitais*

Arquitetura de um Multiplicador

- Esse algoritmo tira partido da propriedade distributiva em relação à adição, permitindo que a multiplicação seja decomposta numa sucessão de somas de produtos parciais
- Considere-se o seguinte produto, em que **M representa o multiplicando** e **m o multiplicador** representados com 4 bits:
  R = M . m

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

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

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

Arquitetura de um Multiplicador

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

- Multiplicar por dois (ou por uma potência de dois) corresponde a deslocar o número multiplicado à esquerda (**shift left**) tantos bits quantos o valor do expoente da 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

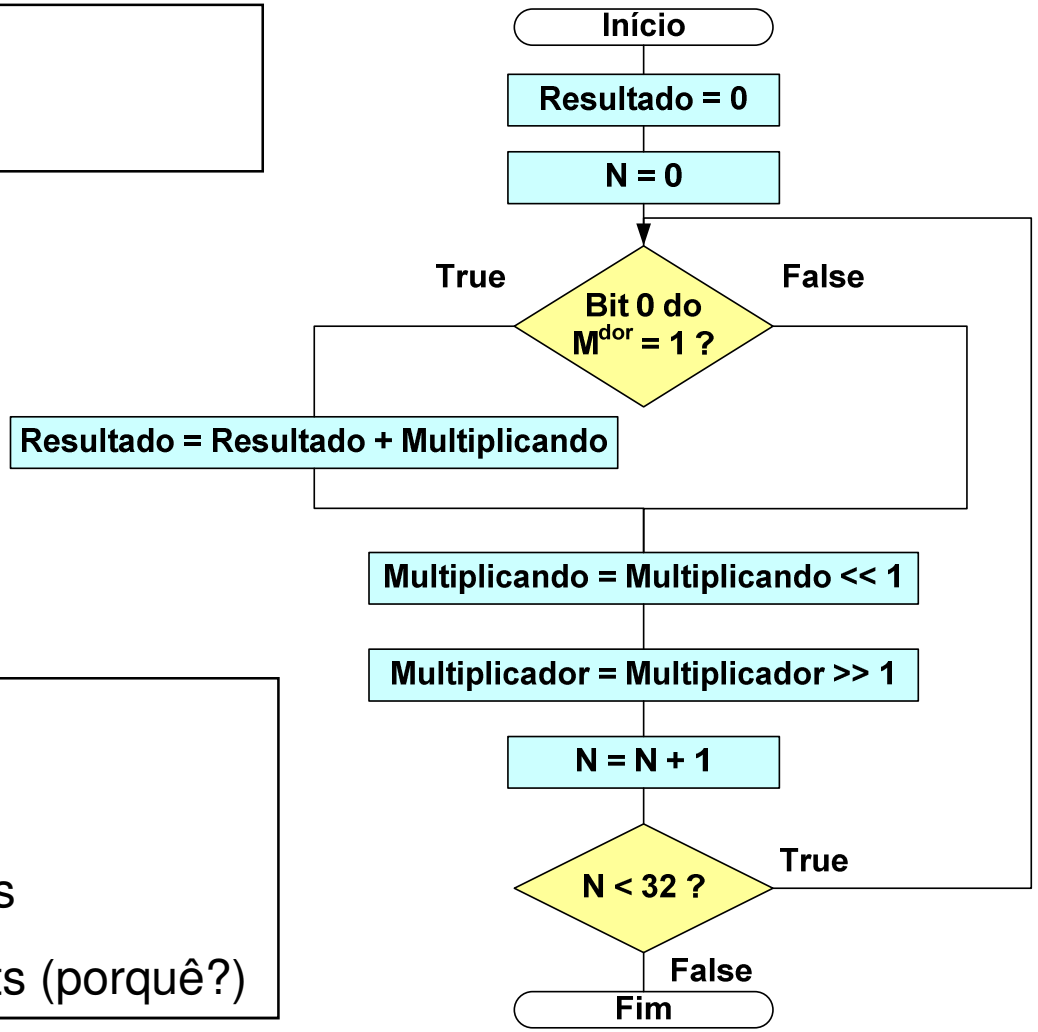
```
   0101
  x0110
  ----
   0000
  01010
 010100
+0000000
  ------
 0011110
```

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" no multiplicador

Multiplicação de inteiros de 32 bits - algoritmo

Operandos: 32 bits

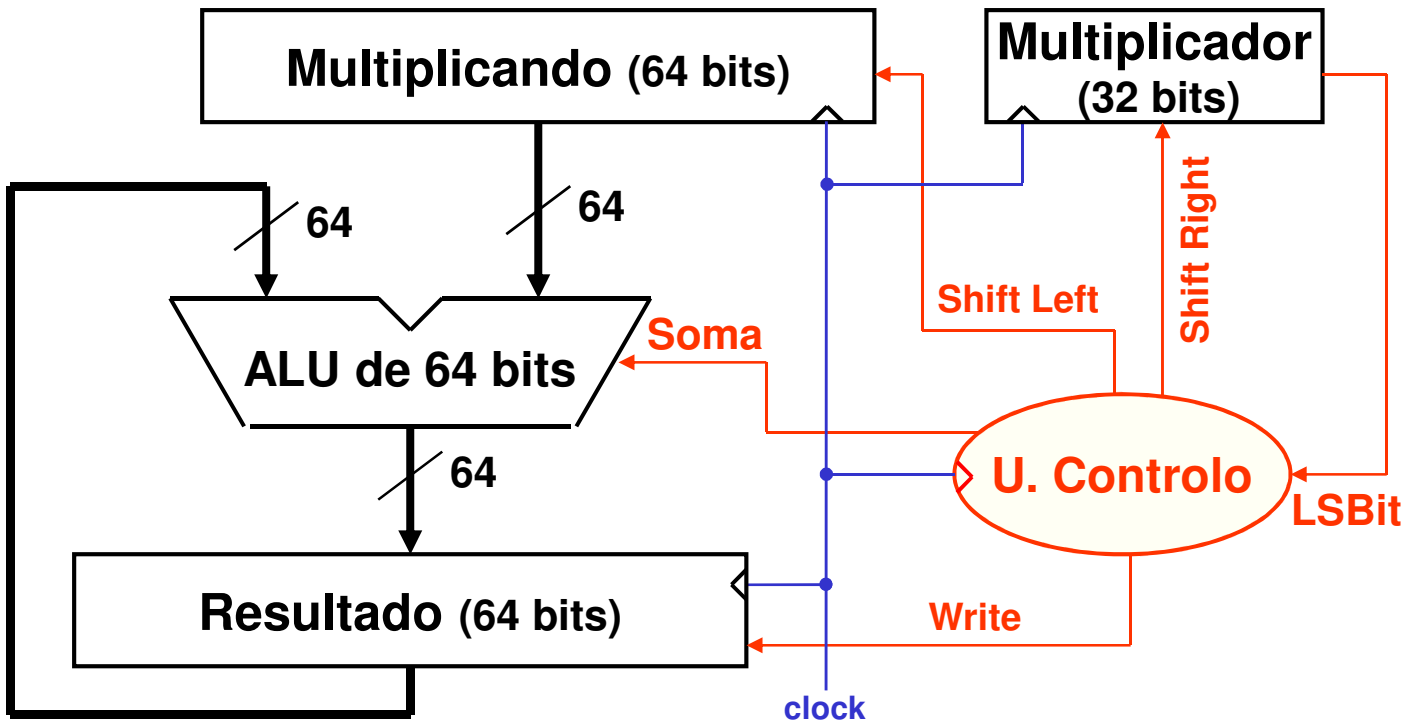
Resultado: 64 bits



Registos necessários:

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

Arquitetura 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
- *Não estão representados os sinais para carregar (load) os valores do multiplicando e do multiplicador e reset do resultado*

Multiplicação de inteiros – exemplo com 4 bits

- Com operandos 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 0000)
  - **multiplicador**: registo de 4 bits

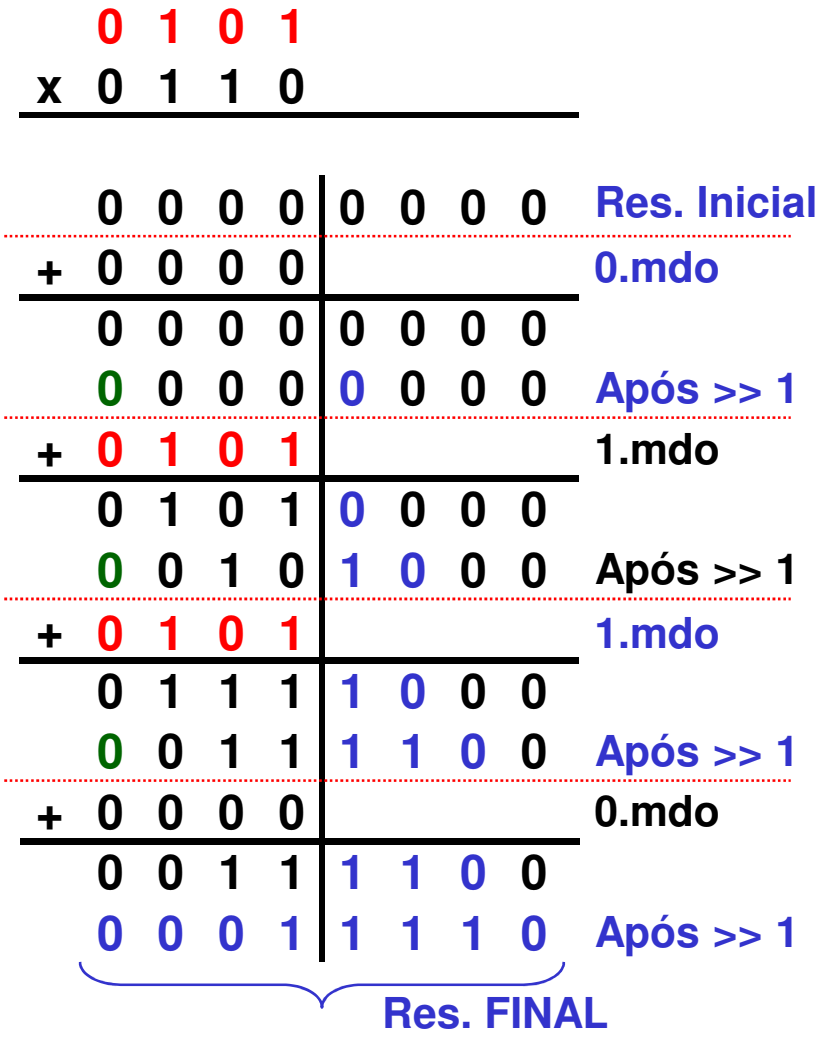
				0	1	0	1		
			x	0	1	1	0		
	0	0	0	0	0	0	0	0	Res. Inicial
+	0	0	0	0	0	0	0	0	0.mdo.2^0
	0	0	0	0	0	0	0	0	
+	0	0	0	0	1	0	1	0	1.mdo.2^1
	0	0	0	0	1	0	1	0	
+	0	0	0	1	0	1	0	0	1.mdo.2^2
	0	0	0	1	1	1	1	0	
+	0	0	0	0	0	0	0	0	0.mdo.2^3
	0	0	0	1	1	1	1	0	
	0	0	0	1	1	1	1	0	Res. FINAL

Arquitetura de um Multiplicador de 32 bits (2ª versão)

- Por cada nova iteração obtém-se o valor final de um novo bit do resultado (as iterações seguintes somam 0 a esse bit)
- Este algoritmo pode ser melhorado:
  - Deslocar, a cada iteração, o resultado para a direita, em vez de o multiplicando para a esquerda (o movimento relativo dos dois é o mesmo, logo o mesmo efeito algorítmico é obtido
  - O multiplicador continua a ser deslocado à direita a cada iteração
- Para cada nova adição é suficiente operar apenas sobre 32 bits dos 64 bits do resultado final
- O registo utilizado para armazenar o multiplicando pode ter apenas 32 bits (em vez de 64 bits) e a ALU é também de 32 bits

Multiplicação de inteiros – exemplo com 4 bits

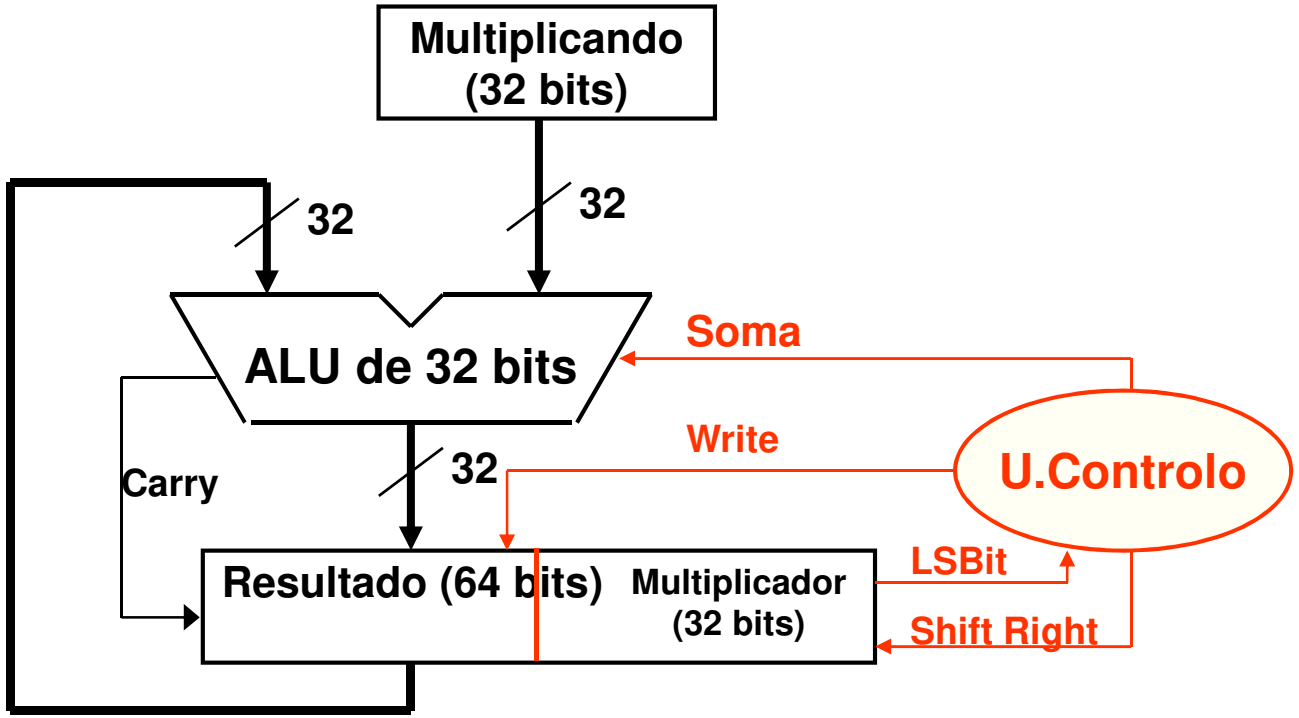
- Uma alternativa ao algoritmo inicial será portanto deslocar o resultado à direita por cada nova iteração
- Vantagens desta nova abordagem:
  - Para cada nova adição é suficiente operar apenas sobre 4 bits dos 8 bits do resultado final
  - O registo utilizado para armazenar o multiplicando pode ter apenas 4 bits (em vez de 8 bits)



Arquitetura de um Multiplicador de 32 bits (versão final)

- Os 32 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 então possível utilizar a parte menos significativa do resultado (32 bits) para armazenar o valor inicial do multiplicador
- Otimiza-se, deste modo, o espaço total de armazenamento e os recursos necessários à arquitetura de multiplicação
  - Registo de 64 bits para o resultado que armazena inicialmente o multiplicador
  - Registo de 32 bits para o armazenamento do multiplicando
  - ALU de 32 bits

Arquitetura de um Multiplicador de 32 bits (versão final)



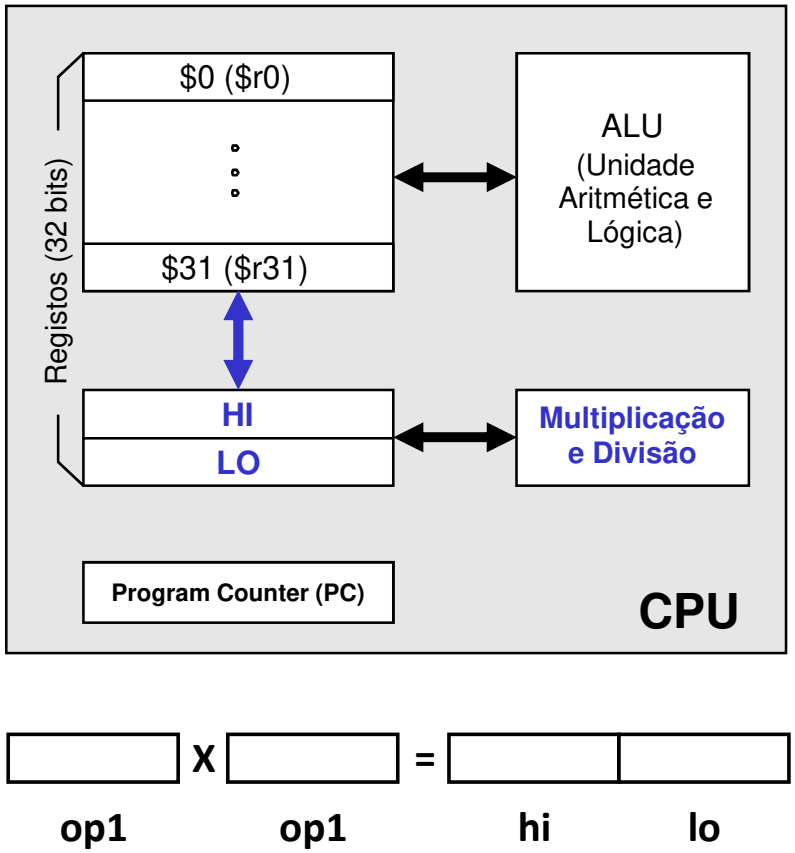
Na versão otimizada do multiplicador, o registo ***Multiplicando*** e a **ALU** passam a ser de 32 bits. O registo ***Multiplicador*** desaparece, sendo substituído pela parte menos significativa do ***Resultado***.

Multiplicação de inteiros com sinal

- A arquitetura de multiplicação que obtivemos no slide anterior anteriormente apenas opera corretamente sobre quantidades binárias consideradas sem sinal
- É portanto uma arquitetura útil para a realização de operações de **multiplicação *unsigned***
- Para a multiplicação de valores em binário com sinal (**multiplicação *signed***), codificados em complemento para dois, utiliza-se o **algoritmo de Booth** (não apresentado nestas aulas)
- A arquitetura que usa o algoritmo de Booth é semelhante à apresentada anteriormente, diferindo apenas na Unidade de Controlo

A Multiplicação de inteiros no MIPS

- No MIPS, a multiplicação é assegurada por uma arquitetura semelhante à anteriormente descrita
- Para o armazenamento do multiplicador e do resultado final, os arquitetos do MIPS incluíram um par de registos especiais designados, respetivamente, por **HI** e **LO**
- Estes registos são de uso específico da unidade de multiplicação e divisão de inteiros



A Multiplicação de inteiros no MIPS

- 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 efetuada automaticamente (por hardware) no início da execução da operação
- A unidade de multiplicação pode operar considerando os operandos sem sinal (multiplicação *unsigned*) ou com sinal (multiplicação *signed*); a distinção é feita através da mnemónica da instrução

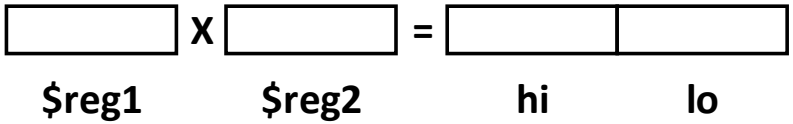
A Multiplicação de inteiros no MIPS

- Em *Assembly*, a multiplicação é efetuada 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**



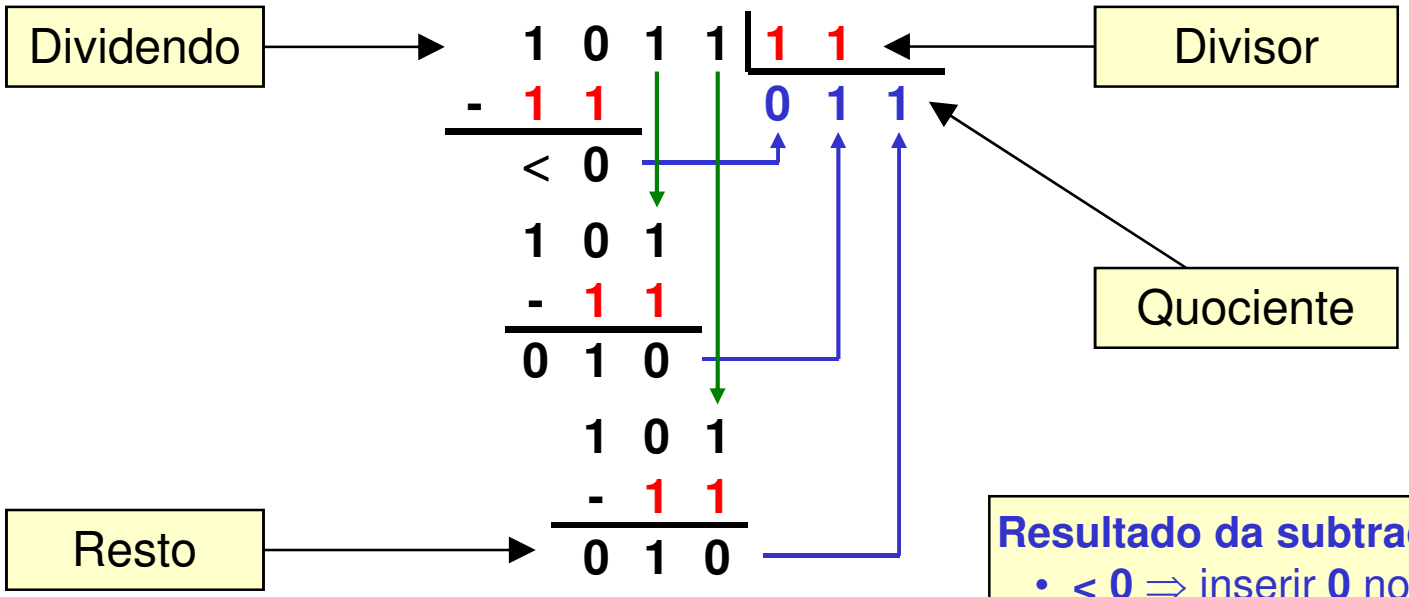
- A transferência de informação entre os registos HI e LO e os restantes registos de uso geral faz-se através das instruções **mfhi** e **mflo**:

```
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
```

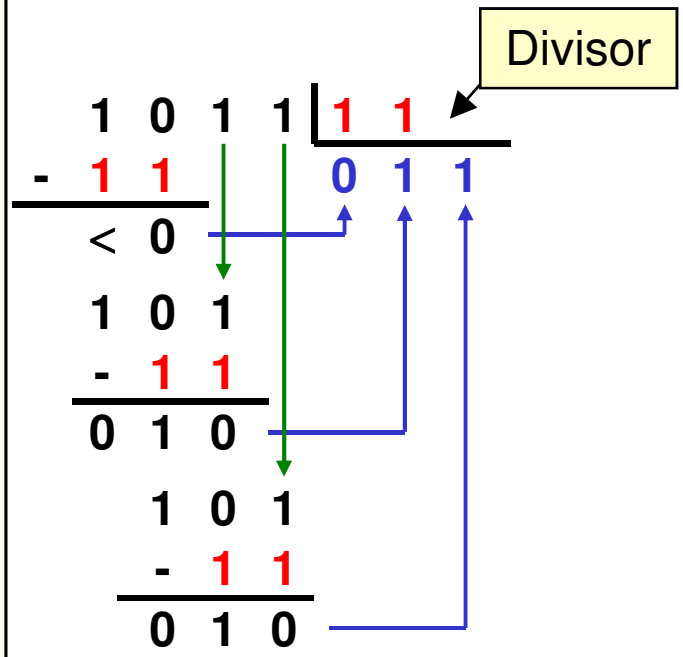
Divisão de inteiros em binário

- Tal como acontecia com a multiplicação, também na divisão se usa uma arquitetura que aproveita o algoritmo que se ensina nos primeiros anos do ensino básico.
- Tomemos como exemplo 1011 ÷ 0011



Resultado da subtração:
- **< 0** ⇒ inserir **0** no quociente
- **≥ 0** ⇒ inserir **1** no quociente

Divisão de inteiros em binário



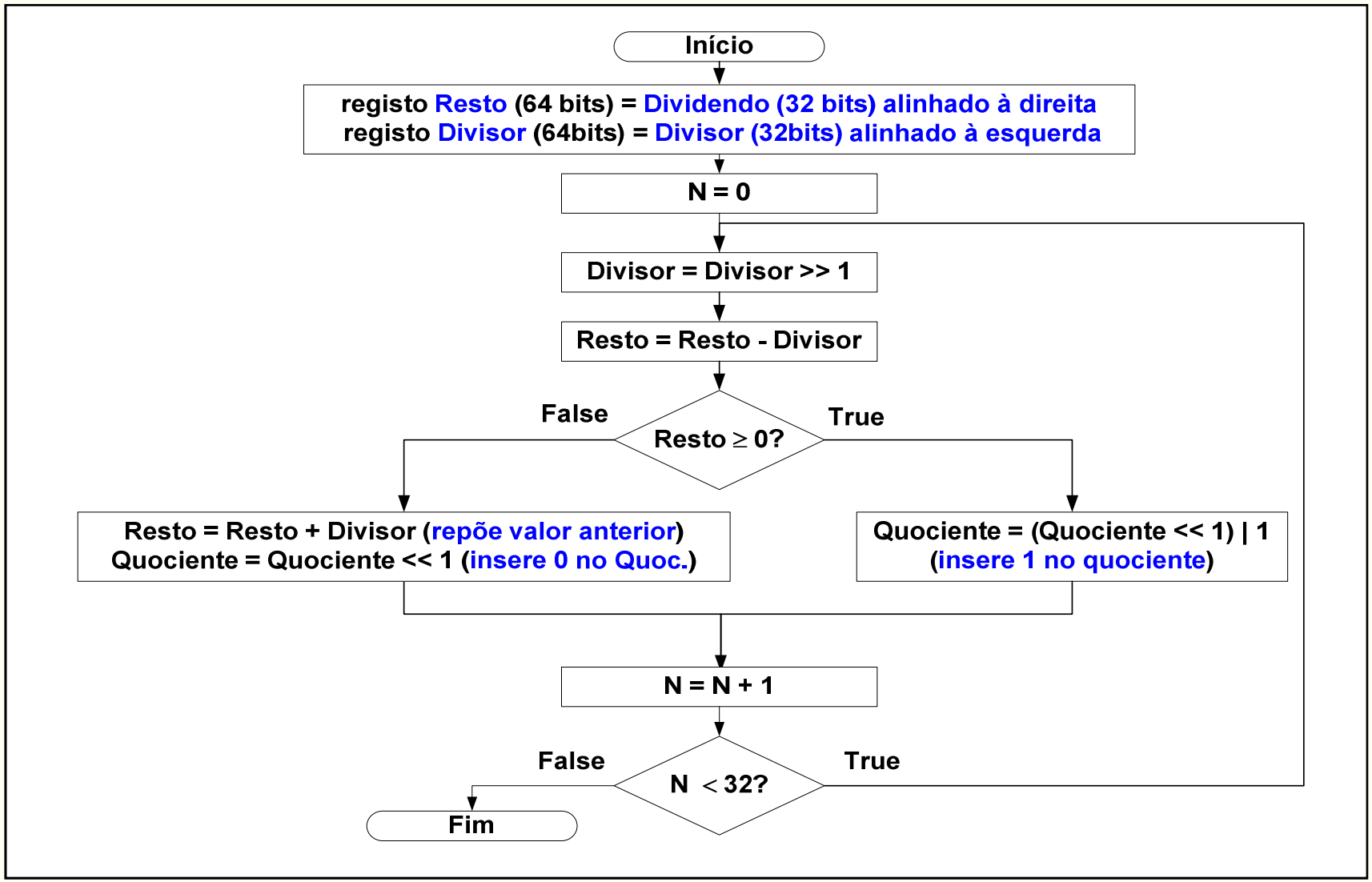
1. Começa-se por alinhar o Divisor à esquerda com o Dividendo
2. **Subtrai-se o Divisor do Dividendo**
   - Se o resultado for **positivo** (i.e. Dividendo >= Divisor) acrescenta-se "**1**" no Quociente
   - Se o resultado for **negativo** (i.e. Dividendo < Divisor) acrescenta-se "**0**" no Quociente e repõe-se o dividendo
3. Se o Divisor ainda não está alinhado à direita com o Dividendo, então desloca-se o Divisor 1 bit para a direita
4. Repete-se desde 2

- Como fazer o alinhamento do divisor à esquerda de forma automática?
- Quantas iterações são necessárias, no caso geral, para fazer a divisão?

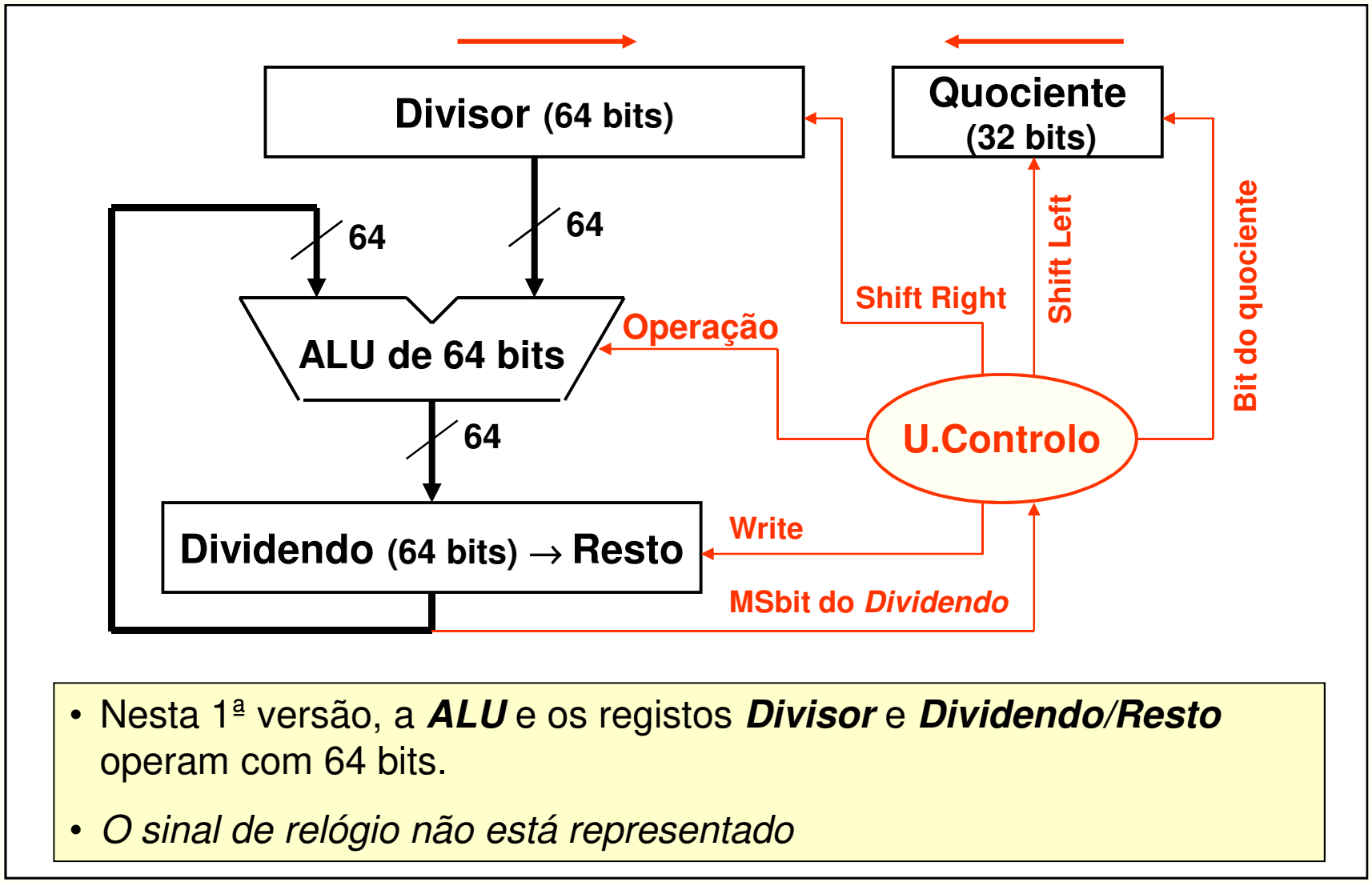
Divisão de inteiros em binário – exemplo com 4 bits

- Com operandos de 4 bits os registos para alojar o **dividendo e o divisor têm 8 bits**
- O **dividendo** é alinhado à **direita** (os 4 MSbits são colocados a 0)
- O **divisor** é alinhado à **esquerda** (os 4 LSbits são colocados a 0)
- Por cada nova iteração o **divisor é deslocado à direita 1 bit**
- O **número total de iterações** é igual ao **número de bits do dividendo original**

Divisão de inteiros em binário – algoritmo

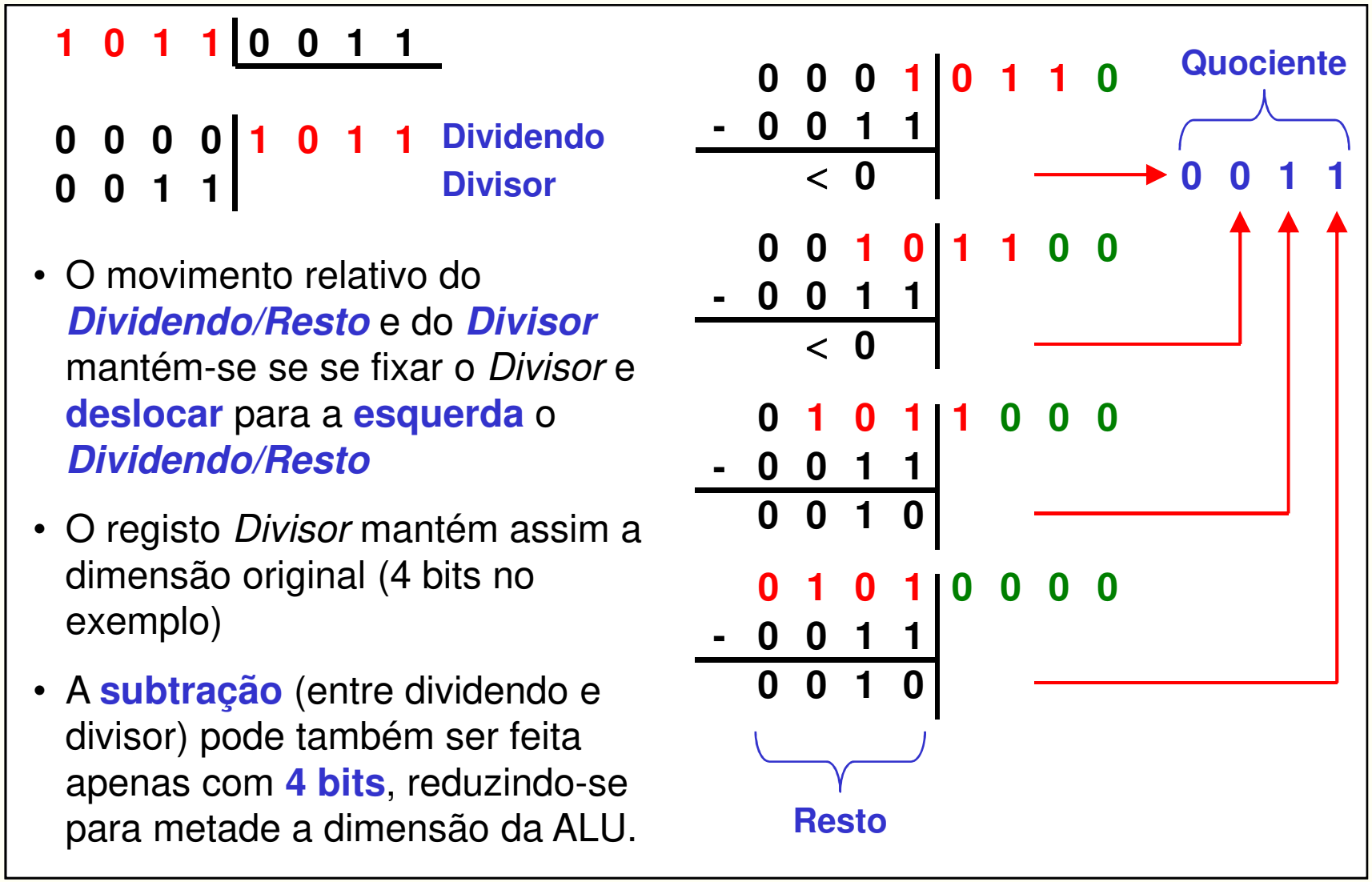


Arquitetura de um Divisor de 32 bits (1ª versão)



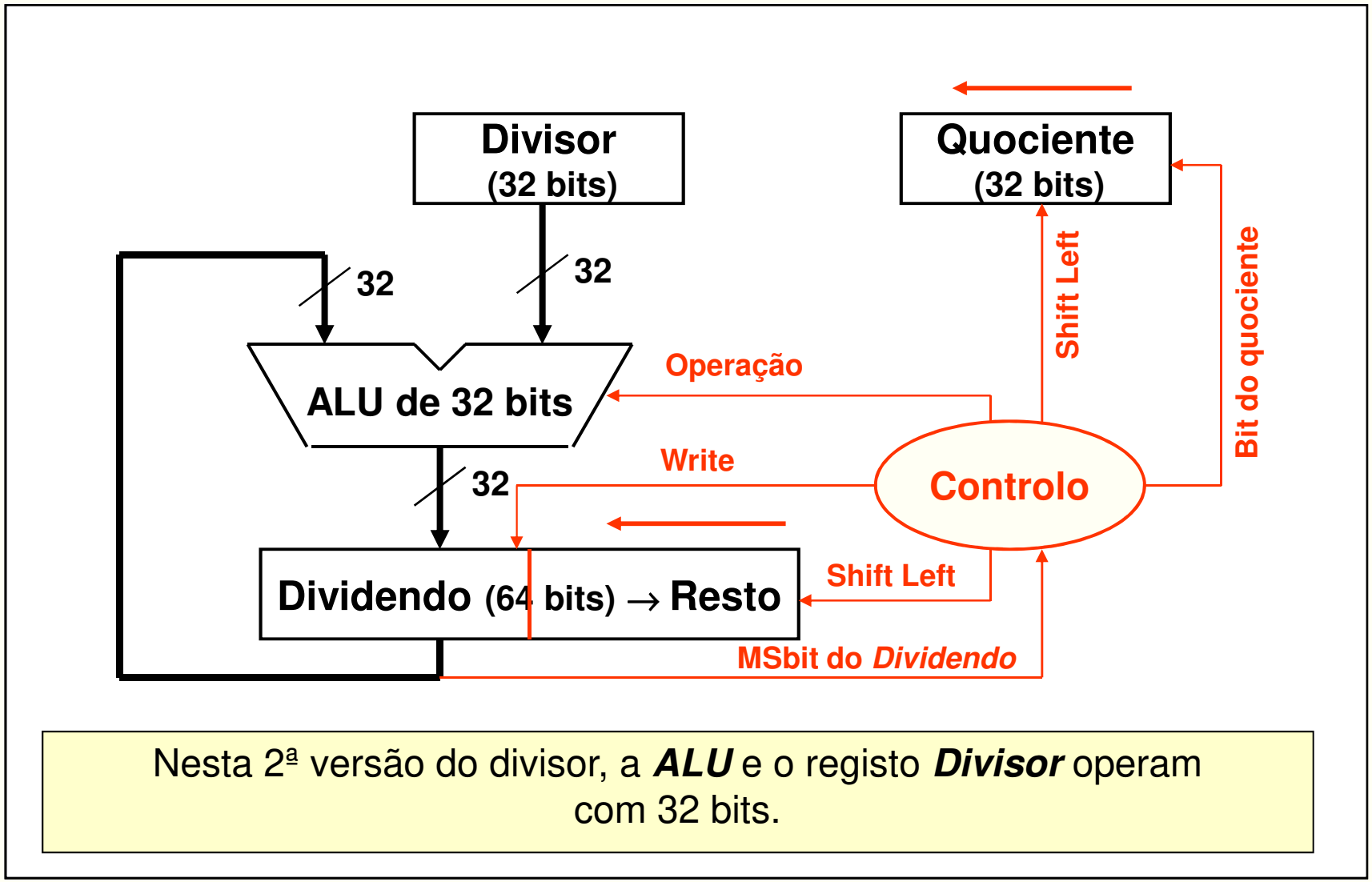
- Nesta 1ª versão, a ***ALU*** e os registos ***Divisor*** e ***Dividendo/Resto*** operam com 64 bits.
- *O sinal de relógio não está representado*

Divisão de inteiros em binário – exemplo com 4 bits



- O movimento relativo do ***Dividendo/Resto*** e do ***Divisor*** mantém-se se se fixar o *Divisor* e **deslocar** para a **esquerda** o ***Dividendo/Resto***
- O registo *Divisor* mantém assim a dimensão original (4 bits no exemplo)
- A **subtração** (entre dividendo e divisor) pode também ser feita apenas com **4 bits**, reduzindo-se para metade a dimensão da ALU.

Arquitetura de um Divisor de 32 bits (2ª versão)



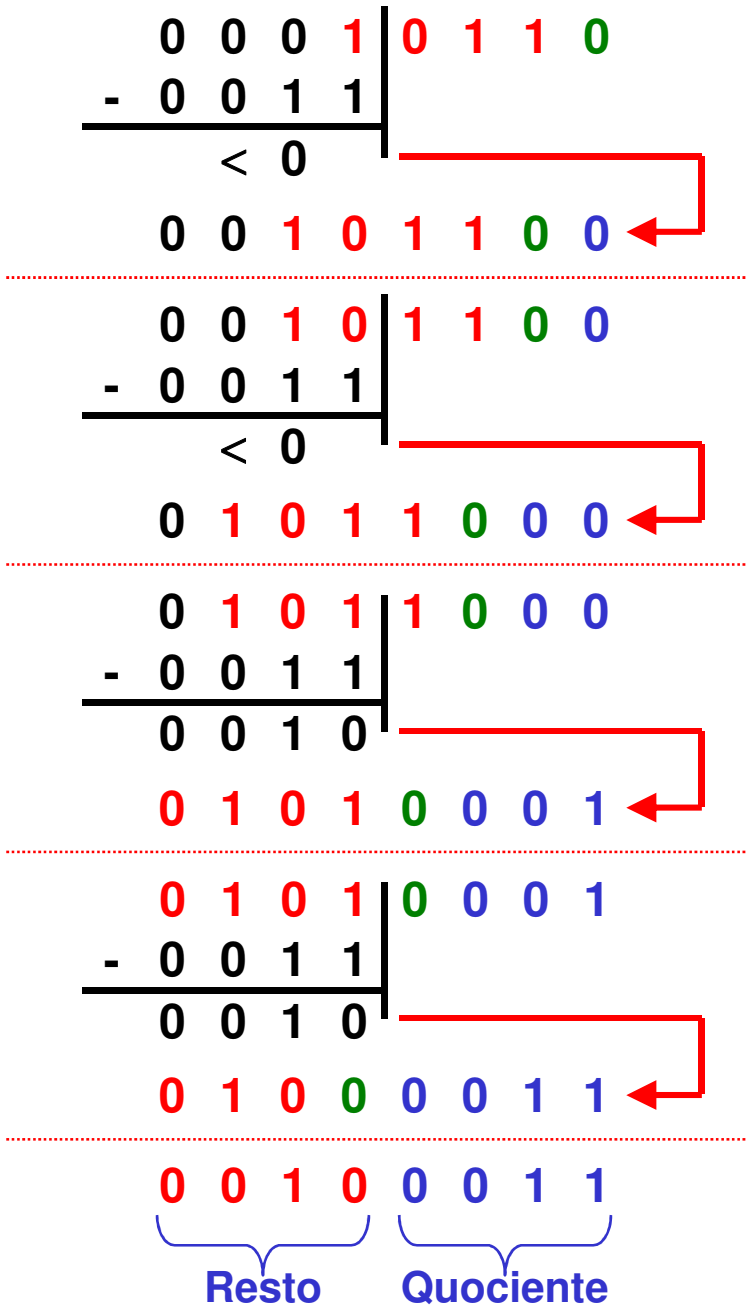
Nesta 2ª versão do divisor, a ***ALU*** e o registo ***Divisor*** operam com 32 bits.

1 0 1 1 | 0 0 1 1

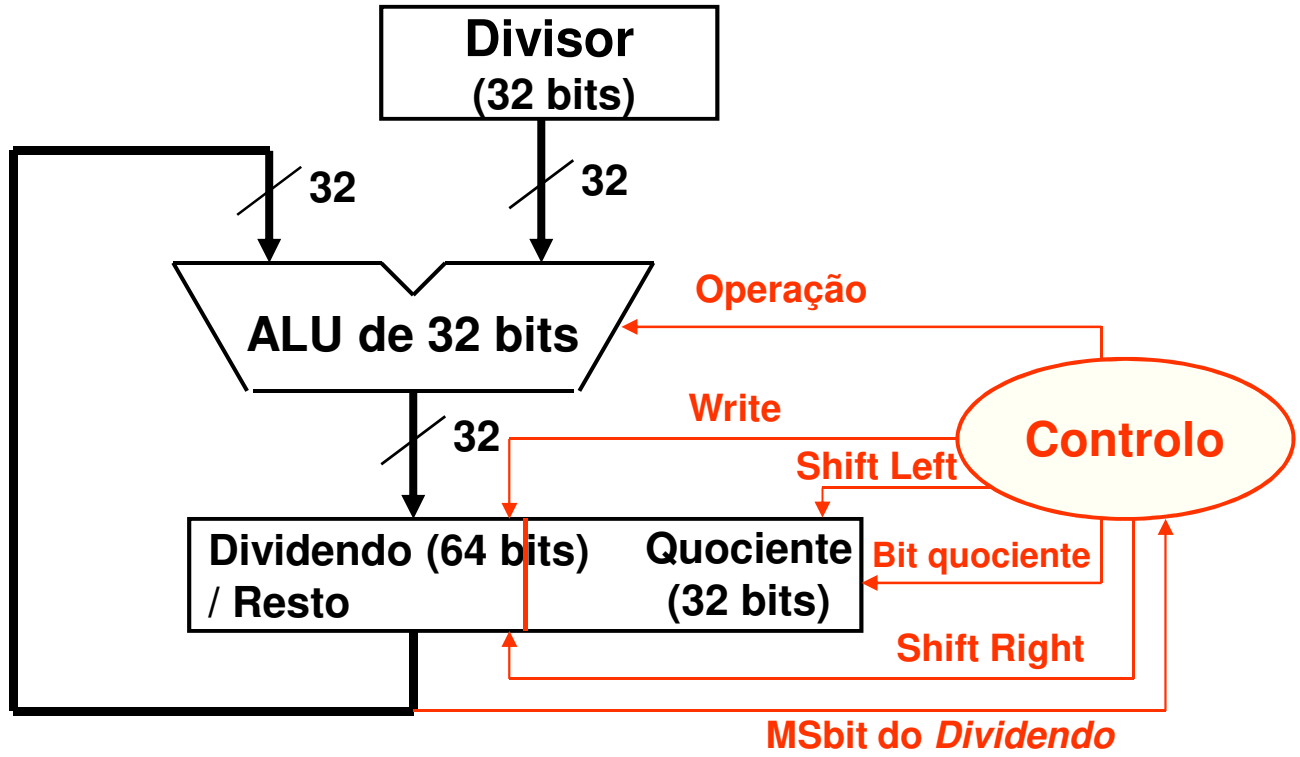
0 0 0 0 1 0 1 1 **Dividendo**

0 0 0 1 | 0 1 1 0 **Dividendo após << 1**
0 0 1 1 **Divisor**

- Pode verificar-se que o deslocamento à esquerda do conteúdo do *Dividendo*, é acompanhado por um deslocamento idêntico do *Quociente*
- Em cada deslocamento à esquerda do *Dividendo* é introduzido um zero (não útil) no bit menos significativo
- **Esse espaço pode ser aproveitado para guardar o próximo bit do quociente**
- Desta forma poupa-se ainda o espaço que seria necessário para armazenar esse quociente



Arquitetura de um Divisor (versão final)



Na versão final do divisor, o registo ***Quociente*** desaparece, sendo substituído pela parte menos significativa do registo ***Dividendo/Resto***.

Divisão de inteiros com sinal

- **A divisão de inteiros com sinal faz-se, do ponto de vista algorítmico, em sinal e módulo**
- Nas divisões com sinal aplicam-se as seguintes regras:
  - Dividem-se dividendo por divisor em módulo
  - O quociente tem sinal negativo se os sinais de dividendo e divisor forem diferentes
  - O resto tem o mesmo sinal do dividendo
- Exemplos:

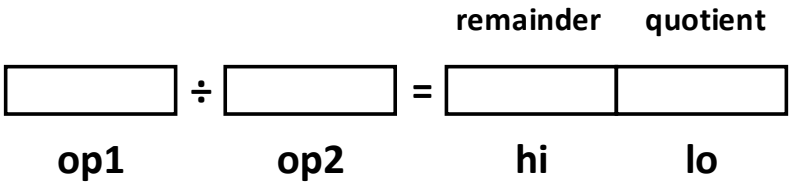
  -7 / 3 = -2 resto = -1

  7 / -3 = -2 resto = 1

- Notar que **Dividendo = Divisor * Quociente + Resto**

A Divisão de inteiros no MIPS

- No MIPS, a divisão é assegurada por uma arquitetura hardware semelhante à anteriormente descrita para a multiplicação (a unidade de controlo é que estabelece a diferença)
- Tal como acontecia na multiplicação, continua a existir a necessidade de um registo de 64 bits para armazenar o valor inicial do dividendo, e bem assim o resultado final na forma de um quociente e de um resto.
- Os mesmos registos, **HI** e **LO**, que tinham já sido apresentados para o caso da multiplicação, são igualmente utilizados para a divisão:
  - o registo **HI armazena o resto da divisão** inteira
  - o registo **LO armazena o quociente da divisão** inteira

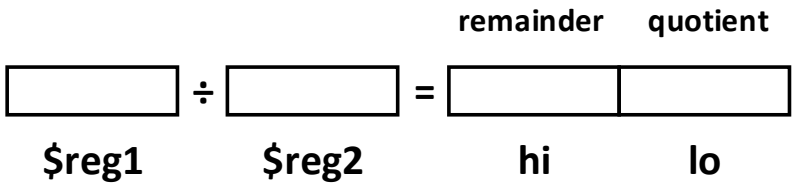


A Divisão de inteiros no MIPS

- Em *Assembly*, a divisão é efetuada pela instrução

```
div     $reg1, $reg2    # Divide (signed)
divu    $reg1, $reg2    # Divide unsigned
```

- em que **$reg1** é o dividendo e **$reg2** o divisor. O **resultado** fica armazenado nos registos **HI (resto)** e **LO (quociente)**.



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

```
mfhi    $reg    # move from hi - Copia HI para $reg
mflo    $reg    # move from lo - Copia LO para $reg
```