

Aula 11

- Representação de números inteiros com sinal: complemento para dois. Exemplos de operações aritméticas
- *Overflow* e mecanismos para a sua deteção
- Construção de uma ALU de 32 bits
- Multiplicação de inteiros no MIPS
- Divisão de inteiros no MIPS. Divisão de inteiros com sinal

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Representação de inteiros

- Sendo um computador um sistema digital binário, a representação de inteiros faz-se sempre em base 2 (símbolos 0 e 1).
- **Tipicamente, um inteiro pode ocupar um número de bits igual à dimensão de um registo interno do CPU.**
- A gama de valores inteiros representáveis é, assim, finita, e corresponde ao número máximo de combinações que é possível obter com o número de bits de um registo interno.
- No MIPS, um inteiro ocupa 32 bits, pelo que o número de inteiros representável é:

$$N_{\text{inteiros}} = 2^{32} = 4.294.967.296_{10} = [0, 4.294.967.295_{10}]$$

Representação de inteiros

- Os circuitos que realizam operações aritméticas estão igualmente limitados a um número finito de bits, geralmente igual à dimensão dos registos internos do CPU
- Os circuitos aritméticos operam assim em aritmética modular, ou seja em **mod** (2^n) em que "**n**" é o número de bits da representação
- O maior valor que um resultado aritmético pode tomar será portanto $2^n - 1$, sendo o valor inteiro imediatamente a seguir o valor zero (representação circular)

Representação de inteiros

- Num CPU com uma ALU de 8 bits, por exemplo, o resultado da soma dos números **11001011** e **00110111** seria:

$$11001011 + 00110111 = \text{1} \boxed{00000010}$$

carry resultado com 8 bits

- No caso em que os operandos são do tipo **unsigned**, o bit **carry**, se igual a '1', sinaliza que o resultado não cabe num registo de 8 bits, ou seja sinaliza a ocorrência de **overflow**
- No caso em que os operandos são do tipo **signed** (codificados em complemento para 2) o bit de **carry**, por si só, não tem qualquer significado, e não faz parte do resultado

Representação em complemento para dois

- O método usado em sistemas computacionais para a codificação de quantidades inteiras com sinal (*signed*) é "complemento para dois"
- **Definição:** Se K é um número positivo, então K^* é o seu complemento para 2 (complemento verdadeiro) e é dado por:

$$K^* = 2^n - K$$

em que "n" é o número de bits da representação

- **Exemplo:** determinar a representação de -5, com 4 bits

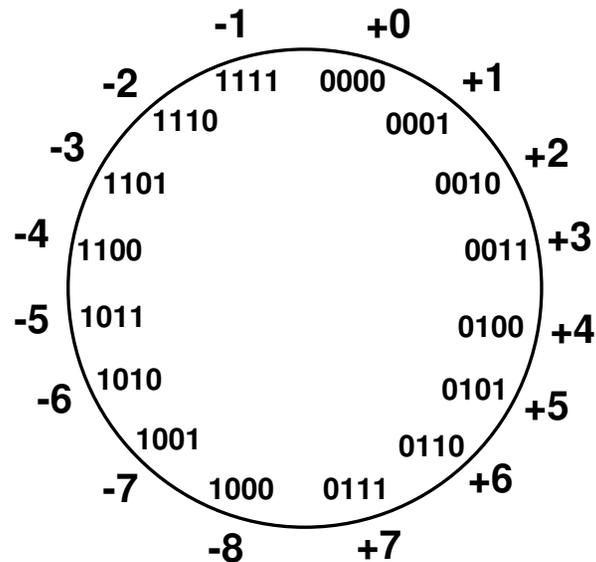
$$N = 5_{10} = 0101_2$$

$$2^n = 2^4 = 10000$$

$$2^n - N = 10000 - 0101 = 1011 = N^*$$

- **Método prático:** inverter todos os bits do valor original e somar 1 (0101 => 1010; 1010 + 1 = 1011)
 - Este método é reversível: $C_1(1011) = 0100$; $0100 + 1 = 0101$

Representação em complemento para dois



↑ +
0 100 = + 4
↓ -
1 100 = - 4
-

O bit mais significativo também **pode ser interpretado como sinal**:
0 = valor positivo,
1 = valor negativo

- Uma única representação para 0
- Codificação assimétrica (mais um negativo do que positivos)
- A subtração é realizada através de uma operação de soma com o complemento para 2 do 2.º operando: $(a-b) = (a+(-b))$
- Uma quantidade de N bits codificada em complemento para 2 pode ser representada pelo seguinte polinómio:

$$-(a_{N-1} \cdot 2^{N-1}) + (a_{N-2} \cdot 2^{N-2}) + \dots + (a_1 \cdot 2^1) + (a_0 \cdot 2^0)$$

Representação em complemento para dois

- Uma quantidade de N bits codificada em complemento para 2 pode então ser representada pelo seguinte polinómio:

$$-(a_{N-1} \cdot 2^{N-1}) + (a_{N-2} \cdot 2^{N-2}) + \dots + (a_1 \cdot 2^1) + (a_0 \cdot 2^0)$$

Onde o bit indicador de sinal (a_{N-1}) é multiplicado por -2^{N-1} e os restantes pela versão positiva do respetivo peso

- **Exemplo:** Qual o valor representado em base 10 pela quantidade 10100101_2 , supondo uma representação em complemento para 2 com 8 bits?

- **R1:** $10100101_2 = -(1 \times 2^7) + (1 \times 2^5) + (1 \times 2^2) + (1 \times 2^0)$
 $= -128 + 32 + 4 + 1 = -91_{10}$

- **R2:** O valor é negativo, calcular o módulo (simétrico de 10100101): $01011010 + 1 = 01011011_2 = 5B_{16} = 91_{10}$
 o módulo da quantidade é 91; logo o valor representado é -91_{10}

Representação em complemento para dois

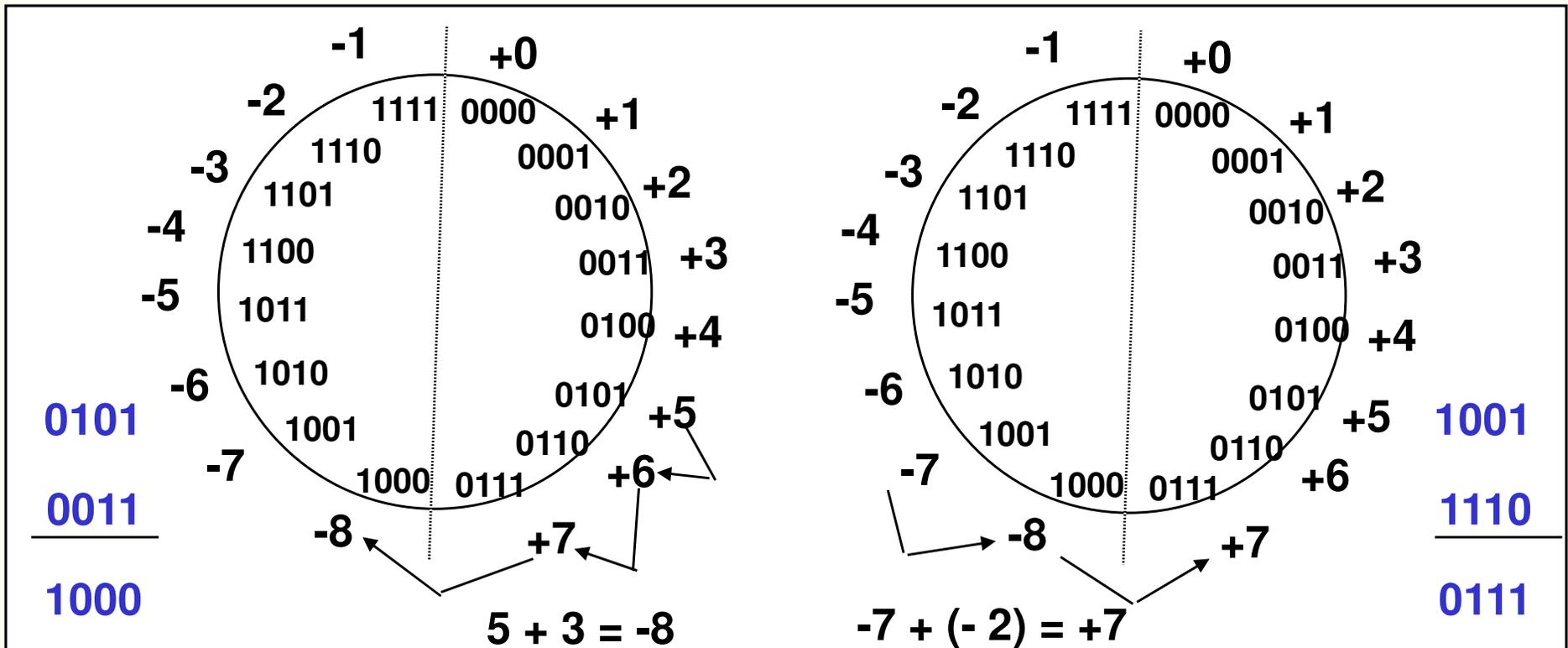
- Exemplos de operações, com 4 bits

(4 + 3)	4	0100	(-4 - 3)	-4	1100
	<u>+ 3</u>	<u>0011</u>		<u>+ (-3)</u>	<u>1101</u>
	7	0111		-7	11001

(4 - 3)	4	0100	(-4 + 3)	-4	1100
	<u>+ (-3)</u>	<u>1101</u>		<u>+ 3</u>	<u>0011</u>
	1	10001		-1	1111

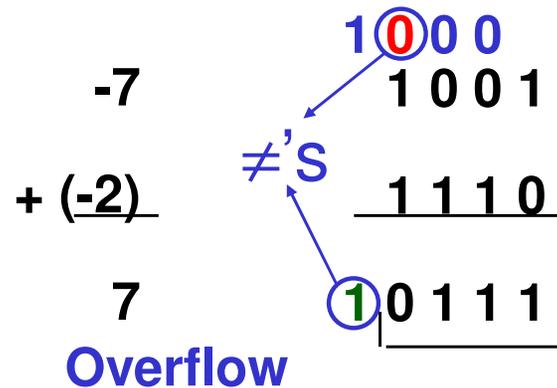
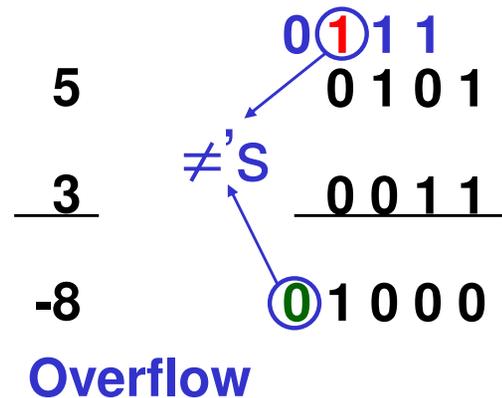
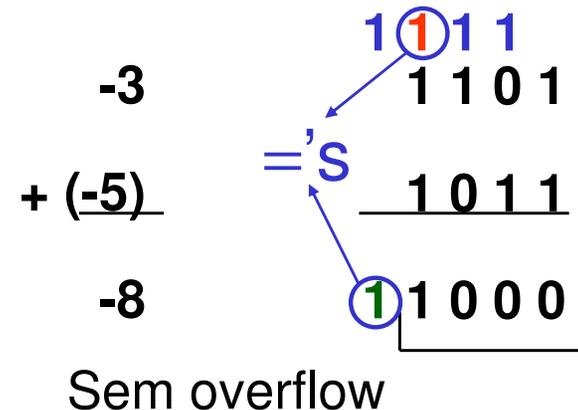
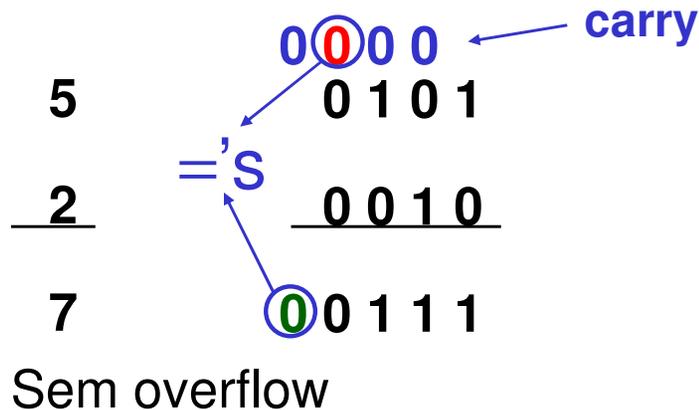
- Este esquema simples de adição com sinal torna o complemento para 2 o preferido para representação de inteiros em arquitetura de computadores

Overflow em complemento para 2



- Ocorre **overflow** quando é ultrapassada a gama de representação. Isso acontece quando:
 - se somam dois positivos e o resultado obtido é negativo
 - se somam dois negativos e o resultado obtido é positivo

Overflow em complemento para 2



A situação de **overflow ocorre** quando o *carry-in* do bit mais significativo não é igual ao *carry-out*, ou seja, quando:

$$C_{n-1} \oplus C_n = 1$$

Overflow em operações aritméticas

- Operandos interpretados em complemento para 2 (i.e. com sinal):

- Quando $A + B > 2^{n-1}-1$ ou $A + B < -2^{n-1}$

- $OVF = (C_{n-1} \cdot \overline{C_n}) + (\overline{C_{n-1}} \cdot C_n) = C_{n-1} \oplus C_n$

- Alternativamente, não tendo acesso aos bits intermédios de *carry*, ($R = A + B$):

- $OVF = R_{n-1} \cdot \overline{A_{n-1}} \cdot \overline{B_{n-1}} + \overline{R_{n-1}} \cdot A_{n-1} \cdot B_{n-1}$

- Operandos interpretados sem sinal:

- Quando $A+B > 2^n-1$ ou $A-B$ c/ $B > A$

- O bit de *carry* $C_n = 1$ sinaliza a ocorrência de *overflow*

- O MIPS apenas deteta *overflow* nas operações de adição com sinal (ADD, SUB, ADDI) e, quando isso acontece, gera uma exceção. ADDU, SUBU e ADDIU não detetam *overflow*

Construção de uma ALU de 32 bits

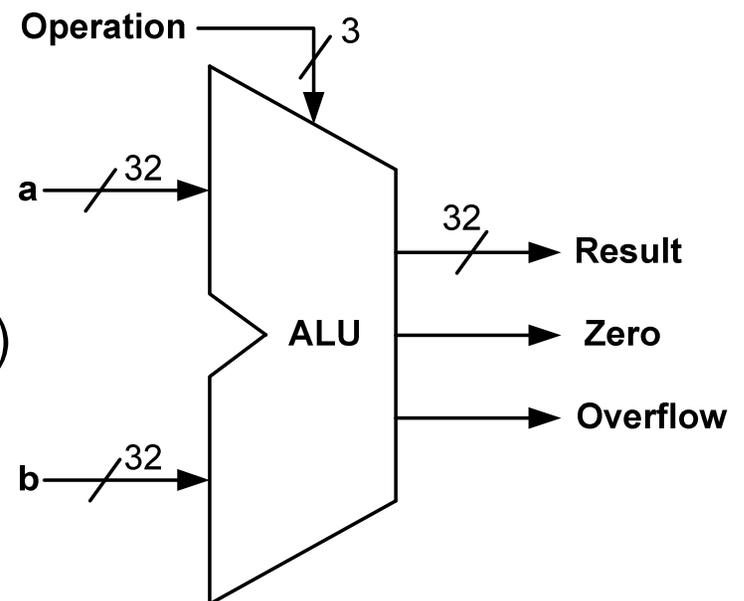
- A ALU deverá realizar as operações:

- ADD, SUB
- AND, OR
- SLT (set if less than)

- Deverá ainda:

- Detetar e sinalizar *overflow* (operandos em complemento para 2)
- Sinalizar resultado igual a zero

Operation	ALU Action
0 0 0	And
0 0 1	Or
0 1 0	Add
1 1 0	Subtract
1 1 1	Set if less than



Bloco funcional correspondente a uma ALU de 32 bits

Construção de uma ALU de 32 bits – VHDL

```
entity alu32 is
  port ( a      : in  std_logic_vector(31 downto 0);
        b      : in  std_logic_vector(31 downto 0);
        oper   : in  std_logic_vector(2  downto 0);
        res    : out std_logic_vector(31 downto 0);
        zero   : out std_logic;
        ovf    : out std_logic);
end alu32;
```

Operation	ALU Action
0 0 0	And
0 0 1	Or
0 1 0	Add
1 1 0	Subtract
1 1 1	Set if less than

```
architecture Behavioral of alu32 is
  signal s_res : std_logic_vector(31 downto 0);
  signal s_b   : unsigned(31 downto 0);
begin
  s_b  <= not(unsigned(b)) + 1 when oper = "110" else
         unsigned(b); -- simétrico de b (se subtração)
  res  <= s_res;
  zero <= '1' when s_res = X"00000000" else '0';
  ovf  <= (not a(31) and not s_b(31) and s_res(31)) or
         (a(31) and s_b(31) and not s_res(31));
  --(continua)
```

```

process (oper, a, b, s_b)
begin
  case oper is
    when "000" =>    -- AND
      s_res <= a and b;
    when "001" =>    -- OR
      s_res <= a or b;
    when "010" =>    -- ADD
      s_res <= std_logic_vector(unsigned(a) + s_b);
    when "110" =>    -- SUB
      s_res <= std_logic_vector(unsigned(a) + s_b);
    when "111" =>    -- SLT
      if(signed(a) < signed(b)) then
        s_res <= X"00000001";
      else
        s_res <= X"00000000";
      end if;
    when others =>
      s_res <= (others => '-');
  end case;
end process;
end Behavioral;

```

Construção de uma ALU de 32 bits (continuação)

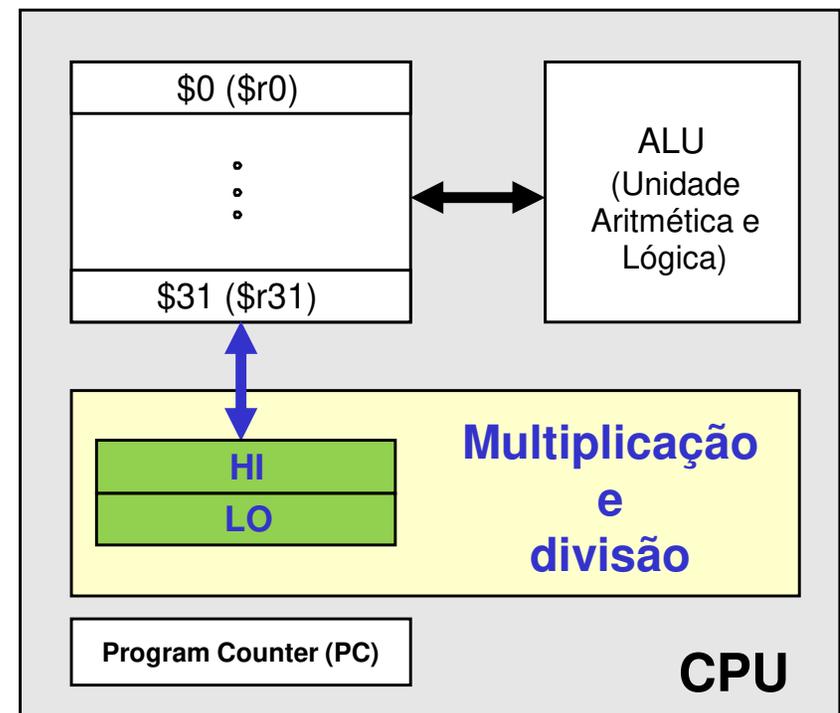
Operation	ALU Action
0 0 0	And
0 0 1	Or
0 1 0	Add
1 1 0	Subtract
1 1 1	Set if less than

Multiplicação de inteiros

- 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
- Multiplicação de quantidades **sem sinal**: algoritmo clássico que é usado na multiplicação em decimal
- Multiplicação de quantidades **com sinal** (representadas em complemento para dois): algoritmo de Booth
- Uma multiplicação que envolva **dois operandos de N bits** carece de um espaço de armazenamento, para o resultado, de **$2 \cdot N$ bits**

A Multiplicação de inteiros no MIPS

- No MIPS, a multiplicação e a divisão são asseguradas por um módulo independente da ALU
- Os operandos são registos de 32 bits. Na multiplicação, tal implica que o **resultado** tem de ser armazenado com **64 bits**
- Os resultados são armazenados num par de registos especiais designados por **HI** e **LO**, cada um com 32 bits
- Estes registos são de uso específico da unidade de multiplicação e divisão de inteiros



$$\boxed{} \times \boxed{} = \boxed{} \boxed{}$$

Rsrc1 **Rsrc2** **hi** **lo**

A Multiplicação de inteiros no MIPS

- O registo **HI** armazena os **32 bits mais significativos do resultado**
- O registo **LO** armazena os **32 bits menos significativos do resultado**
- 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 **Rdst** # **move from hi**: copia HI para Rdst

mflo **Rdst** # **move from lo**: copia LO para Rdst

- A unidade de multiplicação pode operar considerando os operandos com sinal (multiplicação *signed*) ou sem sinal (multiplicação *unsigned*); a distinção é feita através da mnemónica da instrução:
 - **mult** – multiplicação "signed"
 - **multu** – multiplicação "unsigned"

A Multiplicação de inteiros no MIPS

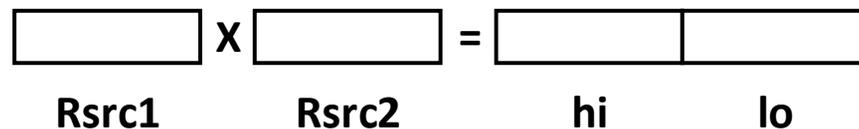
- Em *Assembly*, a multiplicação é então efetuada pelas instruções

mult **Rsrc1**, **Rsrc2** # Multiply (signed)

multu **Rsrc1**, **Rsrc2** # Multiply unsigned

em que **Rsrc1** e **Rsrc2** são os dois registos a multiplicar

- O **resultado** fica armazenado nos **registos HI e LO**



- **Exemplo:** Multiplicar os registos \$t0 e \$t1 e colocar o resultado nos registos \$a1 (32 bits mais significativos) e \$a0 (32 bits menos significativos); os operandos devem ser interpretados com sinal

mult **\$t0, \$t1** # resultado em hi e lo

mfhi **\$a1** # copia hi para registo \$a1

mflo **\$a0** # copia lo para registo \$a0

Instruções virtuais de multiplicação

Multiplicação *signed*

mul **Rdst, Rsrc1, Rsrc2**

mult Rsrc1, Rsrc2

mflo Rdst

Multiplicação *unsigned*

mulu **Rdst, Rsrc1, Rsrc2**

multu Rsrc1, Rsrc2

mflo Rdst

Multiplicação *unsigned* com detecção de overflow

mulou **Rdst, Rsrc1, Rsrc2**

multu Rsrc1, Rsrc2

mfhi \$1

beq \$1, \$0, cont

break

cont: mflo Rdst

Multiplicação *signed* com detecção de overflow

mulo **Rdst, Rsrc1, Rsrc2**

mult Rsrc1, Rsrc2

mfhi \$1

mflo Rdst

sra Rdst, Rdst, 31

beq \$1, Rdst, cont

break

cont: mflo Rdst

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:
 - Divide-se dividendo por divisor, em módulo
 - O quociente tem sinal negativo se os sinais do dividendo e do divisor forem diferentes
 - O resto tem o mesmo sinal do dividendo

- Exemplo 1 (**dividendo = -7, divisor = 3**):

$$-7 / 3 = -2 \quad \text{resto} = -1$$

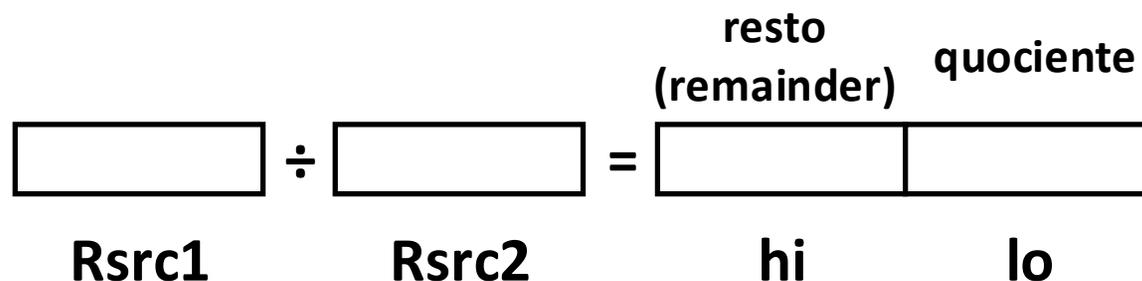
- Exemplo 2 (dividendo = 7, divisor = -3):

$$7 / -3 = -2 \quad \text{resto} = 1$$

Note que: **Dividendo = Divisor * Quociente + Resto**

A Divisão de inteiros no MIPS

- Tal como na multiplicação, continua a existir a necessidade de um registo de 64 bits para armazenar o resultado final na forma de um quociente e de um resto
- Os mesmos registos, **HI** e **LO**, que tinham já sido usados para a 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



Instruções virtuais de divisão

Divisão *signed*

div	Rdst, Rsrc1, Rsrc2
div	Rsrc1, Rsrc2
mflo	Rdst

Divisão *unsigned*

divu	Rdst, Rsrc1, Rsrc2
divu	Rsrc1, Rsrc2
mflo	Rdst

Resto da divisão *signed*

rem	Rdst, Rsrc1, Rsrc2
div	Rsrc1, Rsrc2
mfhi	Rdst

Resto da divisão *unsigned*

remu	Rdst, Rsrc1, Rsrc2
divu	Rsrc1, Rsrc2
mfhi	Rdst

Exercícios

- Para uma codificação em complemento para 2, apresente a gama de representação que é possível obter com 3, 4, 5, 8 e 16 bits (indique os valores-limite da representação em binário, hexadecimal e em decimal com sinal e módulo).
- Determine a representação em complemento para 2 com 16 bits das seguintes quantidades:
 - 5, -3, -128, -32768, 31, -8, 256, -32
- Determine o valor em decimal representado por cada uma das quantidades seguintes, supondo que estão codificadas em complemento para 2 com 8 bits:
 - 00101011_2 , $0xA5$, 10101101_2 , $0x6B$, $0xFA$, $0x80$
- Determine a representação das quantidades do exercício anterior em hexadecimal com 16 bits (também codificadas em complemento para 2).

Exercícios

- Como é realizada a detecção de *overflow* em operações de adição com quantidades sem sinal? E com quantidades com sinal (codificadas em complemento para 2)?
- Para a multiplicação de dois operandos de "**m**" e "**n**" bits, respectivamente, qual o número de bits necessário para o armazenamento do resultado?
- Apresente a decomposição em instruções nativas da instrução virtual **mul \$5, \$6, \$7**
- Determine o resultado da instrução anterior, quando **\$6=0xFFFFFFFFE** e **\$7=0x00000005**.
- Apresente a decomposição em instruções nativas das instruções virtuais **div \$5, \$6, \$7** e **rem \$5, \$6, \$7**
- Determine o resultado das instruções anteriores, quando **\$6=0xFFFFFFFF0** e **\$7=0x00000003**

Exercícios

- As duas sub-rotinas do slide seguinte permitem detetar *overflow* nas operações de adição com e sem sinal, no MIPS. Analise o código apresentado e determine o resultado produzido, pelas duas sub-rotinas, nas seguintes situações:
 - `$a0=0x7FFFFFF1, $a1=0x0000000E;`
 - `$a0=0x7FFFFFF1, $a1=0x0000000F;`
 - `$a0=0xFFFFFFFF1, $a1=0xFFFFFFFF;`
 - `$a0=0x80000000, $a1=0x80000000;`
- Ainda no código das sub-rotinas, qual a razão para não haver salvaguarda de qualquer registo na *stack*?

Exercícios

```
# Overflow detection, signed
# int isovf_signed(int a, int b);
isovf_signed:  ori  $v0, $0, 0
               xor  $1, $a0, $a1
               slt  $1, $1, $0
               bne  $1, $0, notovf_s
               addu $1, $a0, $a1
               xor  $1, $1, $a0
               slt  $1, $1, $0
               beq  $1, $0, notovf_s
               ori  $v0, $0, 1
notovf_s:      jr   $ra

# Overflow detection, unsigned
# int isovf_unsigned(unsigned int a, unsigned int b);
isovf_unsigned: ori  $v0, $0, 0
                nor  $1, $a1, $0
                sltu $1, $1, $a0
                beq  $1, $0, notovf_u
                ori  $v0, $0, 1
notovf_u:      jr   $ra
```