Aula 9

- Representação de números inteiros com sinal (revisão)
  - Sinal e módulo
  - Complemento para um
  - Complemento para dois
- Exemplos de operações aritméticas
- *Overflow* e mecanismos para a sua deteção
- Construção de uma ALU de 32 bits

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

Era uma vez...

- Era uma vez, num país bué, bué longe, um conselho de ministros...
- O Sr. Primeiro Ministro dirigiu-se aos Srs. Ministros, questionando-os sobre quantos milhares de milhões deveria atribuir ao NewBank no orçamento de 2020, para equilibrar ☺☺☺ os desvarios do mesmo.

Dizei-me senhores ministros
No vosso saber dotado
Qual deve ser o valor
Pr'a dar ao NewBank, coitado

Era uma vez...

- Fácil, disse o Ministro da Cultura: eu acho que devia ser **00000101** G€
- Não, retorquiu o Ministro dos Negócios Estrangeiros, para mim devia ser **01000011 01001001 01001110 01000011 01001111** G€
- Não concordo, contestou o Ministro do Trabalho, eu cá acho que devia ser **01010110** G€
- Nada disso, insurgiu-se o Ministro da Economia. Para sermos justos é necessário um aumento de **01000000101000000000000000000000** G€
- Por todos os deuses – levantou-se o Ministro das Finanças irritado – só um cego não vê que o orçamento só suporta um aumento de **10000100** G€

Era uma vez...

- O Primeiro Ministro desse país longínquo, conhecido como um hábil negociador, nada sabia de códigos de representação e ficou bastante irritado com as respostas dos seus ministros
- No entanto, não havia razão para tal, uma vez que houve unanimidade nas respostas. A resposta dada por cada ministro foi, na realidade, a mesma; apenas usaram uma linguagem (código) diferente
- A extração da informação requer, assim, o conhecimento do código usado, sob pena de as mensagens não passarem de coleções de bits sem sentido

Era uma vez...

O Ministro da Cultura codificou a sua resposta em **binário**:
00000101_2 = 5_{10} G€

O Ministro dos Negócios Estrangeiros usou **ASCII**:
01000011 01001001 01001110 01000011 01001111 = "CINCO"G€

O Ministro do Trabalho usou **ASCII** mas para representar numeração romana:
01010110 = "V" = 5 G€

O Ministro da Economia usou representação em **vírgula flutuante**:
01000000101000000000000000000000 = $1.01_2 x 2^2$ = 5 G€

O Ministro das Finanças usou **excesso de $2^{n-1}-1$** (com n=8, excesso de 127):
10000100_2 = 5_{10} G€

Representação de inteiros

- No sistema árabe, cada algarismo que compõe um dado número tem um peso que é função quer da sua posição no número quer do número de símbolos do alfabeto usado.
- Um número com **n** dígitos $\mathbf{d_{n-1}\ d_{n-2}...d_1\ d_0}$

  representado neste sistema, pode ser decomposto num polinómio da forma

  $$\mathbf{d_{n-1}.b^{n-1} + d_{n-2}.b^{n-2} + ... + d_1.b^1 + d_0.b^0}$$

  em que **b** é a base de representação e corresponde à dimensão do alfabeto
- Exemplos:

  $\mathbf{1230_{10}}$ $= 1 \times 10^3 + 2 \times 10^2 + 3 \times 10 + 0$

  $\mathbf{110101_2}$ $= 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2 + 1 = 53_{10}$

  $\mathbf{721_8}$ $= 7 \times 8^2 + 2 \times 8 + 1 = 465_{10}$

  $\mathbf{5A8_{16}}$ $= 5 \times 16^2 + A \times 16 + 8 = 1448_{10}$

Representação de inteiros

- Sendo um computador um sistema digital binário, a representação de inteiros faz-se sempre em base 2 (símbolos 0 e 1).
- Por outro lado, como o espaço de armazenamento de informação (numérica ou não) é limitado, a representação de inteiros é também necessariamente limitada.
- **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:

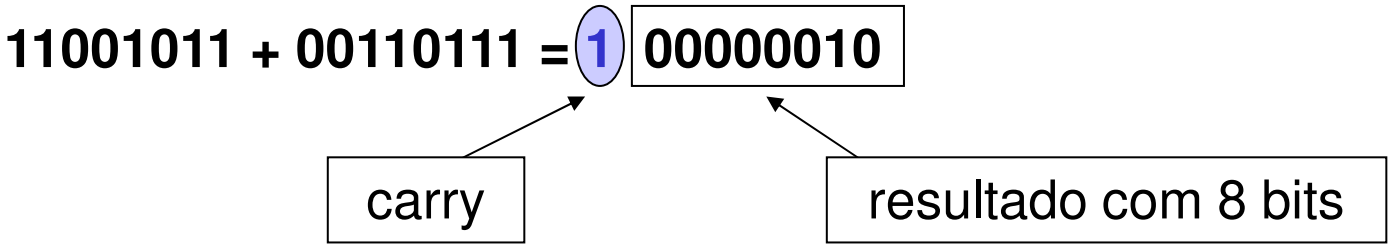
$$N_{inteiros} = 2^{32} = 4.294.967.296_{10} = [0 .. 4.294.967.295_{10}]$$

Representação de inteiros

- Os circuitos que realizam operações aritméticas estão igualmente limitados a um número finito de dígitos (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 de 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 registos de 8 bits, por exemplo, o resultado da soma dos números 11001011 e 00110111 seria:

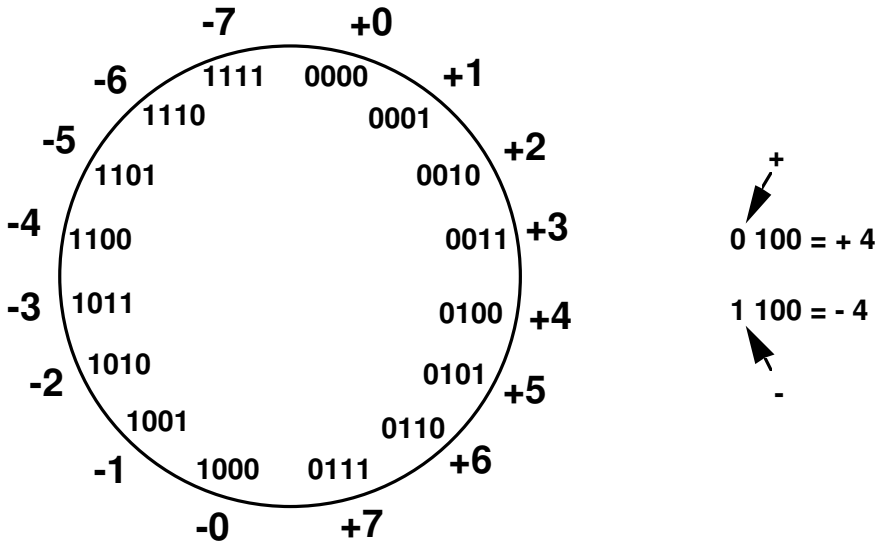


- No caso em que os operandos são do tipo ***unsigned***, o bit ***carry*** sinaliza que o resultado não cabe num registo de 8 bits, ou seja sinaliza a ocorrência de ***overflow***
- No caso em que os operandos são do tipo ***signed*** (codificados em complemento para 2) o bit de ***carry*** não tem qualquer significado e é ignorado.

Representação de inteiros negativos

- A representação de números positivos é a mesma na maioria dos sistemas numéricos
- Os maiores problemas colocam-se quando se procura uma forma de representar quantidades negativas
- Os três esquemas mais usados são:
  - sinal e módulo
  - complemento para um
  - complemento para dois
- Por uma questão de simplicidade vamos admitir, na discussão subsequente, que a dimensão do registo interno do CPU é de 4 bits

Representação em sinal e módulo



- O bit mais significativo é usado para representar o sinal:
  - 0 = positivo (ou zero), 1 = negativo
- A magnitude é representada pelos 3 LSBits: 0 (000) a 7 (111)
- Gama de representação para n bits = $+/-2^{n-1}-1$
- 2 representações para 0

Representação em sinal e módulo

- Este método de representação de inteiros apresenta os seguintes problemas do ponto de vista da implementação numa ALU:
  - Existem duas representações distintas para um mesmo valor (zero)
  - É necessário comparar as magnitudes dos operandos para determinar o sinal do resultado
  - É necessário implementar um somador e um subtrator distintos
  - O bit de sinal tem de ser tratado independentemente dos restantes

Representação em complemento para um

- **Definição**: Se N é um número positivo, então $\overline{N}$ é negativo e o seu complemento para 1 (complemento falso) é dado por:

  $$\overline{N} = (2^n - 1) - N$$

  em que n é o número de bits da representação

- **Exemplo**: determinar o complemento para 1 de 5 (com 4 bits)

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

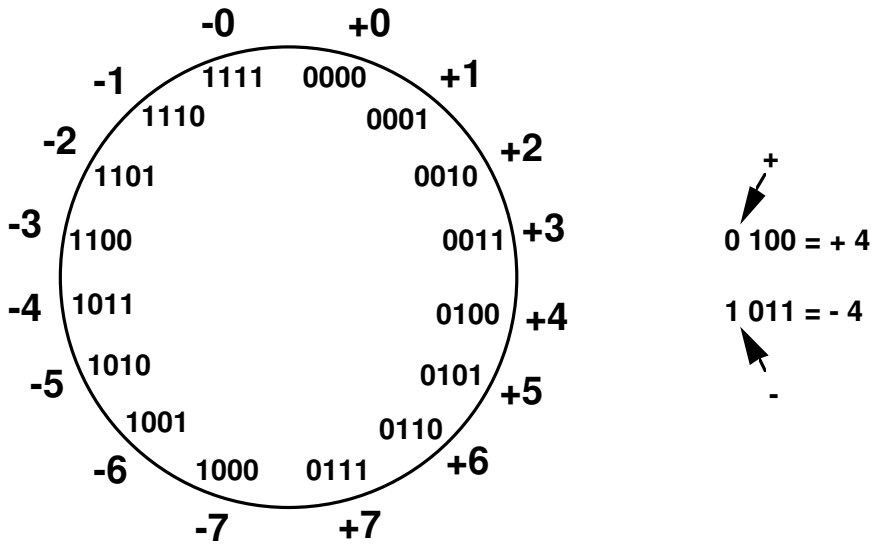
  $2^n = 2^4 = 10000$

  $(2^n - 1) = 10000 - 1 = 1111$

  $(2^n - 1) - N = 1111 - 0101 = \mathbf{1010}$

- **Método prático**: inverter todos os bits do valor original

Representação em complemento para um



- O bit mais significativo também pode ser interpretado como sinal: 0 = valor positivo, 1 = valor negativo
- A subtração faz-se adicionando o complemento para 1
- Há 2 representações para 0 (tem implicações no modo como as operações são realizadas)

Representação em complemento para dois

- **Definição**: Se N é um número positivo, então N* é o seu complemento para 2 (complemento verdadeiro) e é dado por:

  $$N^* = 2^n - N$$

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

- **Exemplo**: determinar o complemento para 2 de 5 (com 4 bits)

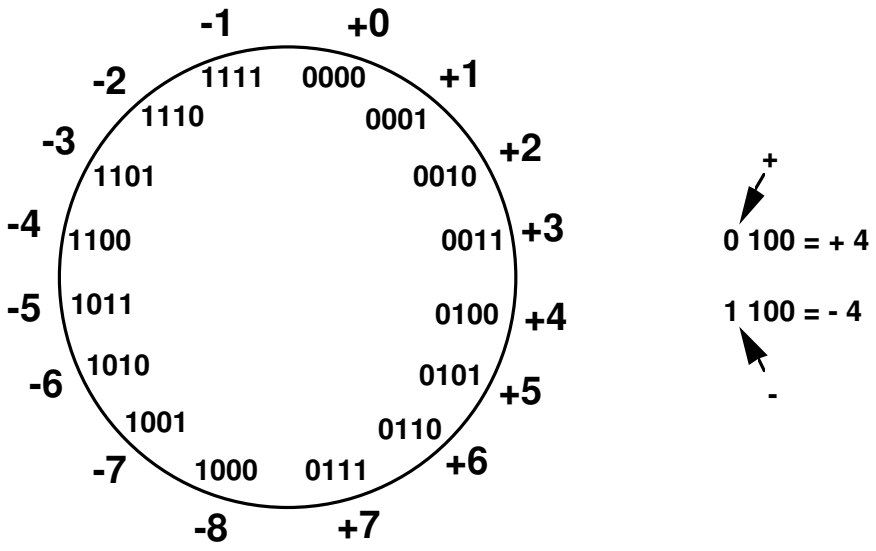
  $N = 5_{10} = 0101_2$

  $2^n = 2^4 = 10000$

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

- **Método prático**: inverter todos os bits do valor original e somar 1

Representação em complemento para dois



- 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 obtida através da soma com o complemento para 2
  (a – b) = (a + (-b))

Representação em complemento para dois

- Uma quantidade de 32 bits codificada em complemento para 2 pode ser representada pelo seguinte polinómio:

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

  Onde o bit de sinal (a_{31}) é multiplicado por -2^{31} e os restantes pela versão positiva do respetivo peso

- **Exemplo**: Qual o valor representado pela quantidade 10100101_2, supondo uma representação com 8 bits e uma codificação em complemento para 2?
  - R1: $10100101_2 = -(1x2^7) + (1x2^5) + (1x2^2) + (1x2^0)$ $= -128 + 32 + 4 + 1 = -91_{10}$
  - R2: Complemento para 2 de 10100101 = 01011010 + 1 $= 01011011_2 = 5B_{16} = 91_{10}$. Ou seja, o valor representado em sinal e módulo, base 10, é -91_{10}

Representação em complemento para dois

- Exemplos de operações

4	0100
+ 3	0011
7	0111

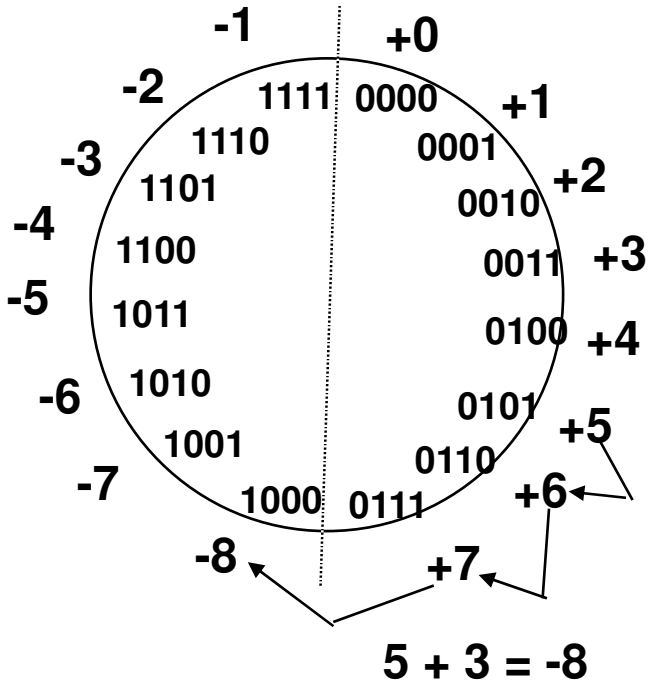
-4	1100
+ (-3)	1101
-7	11001

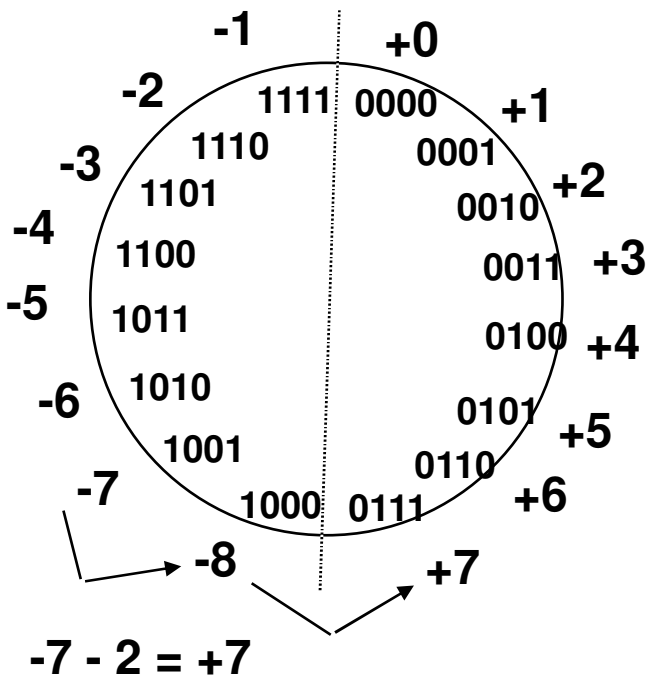
4	0100
- 3	1101
1	10001

-4	1100
+ 3	0011
-1	1111

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

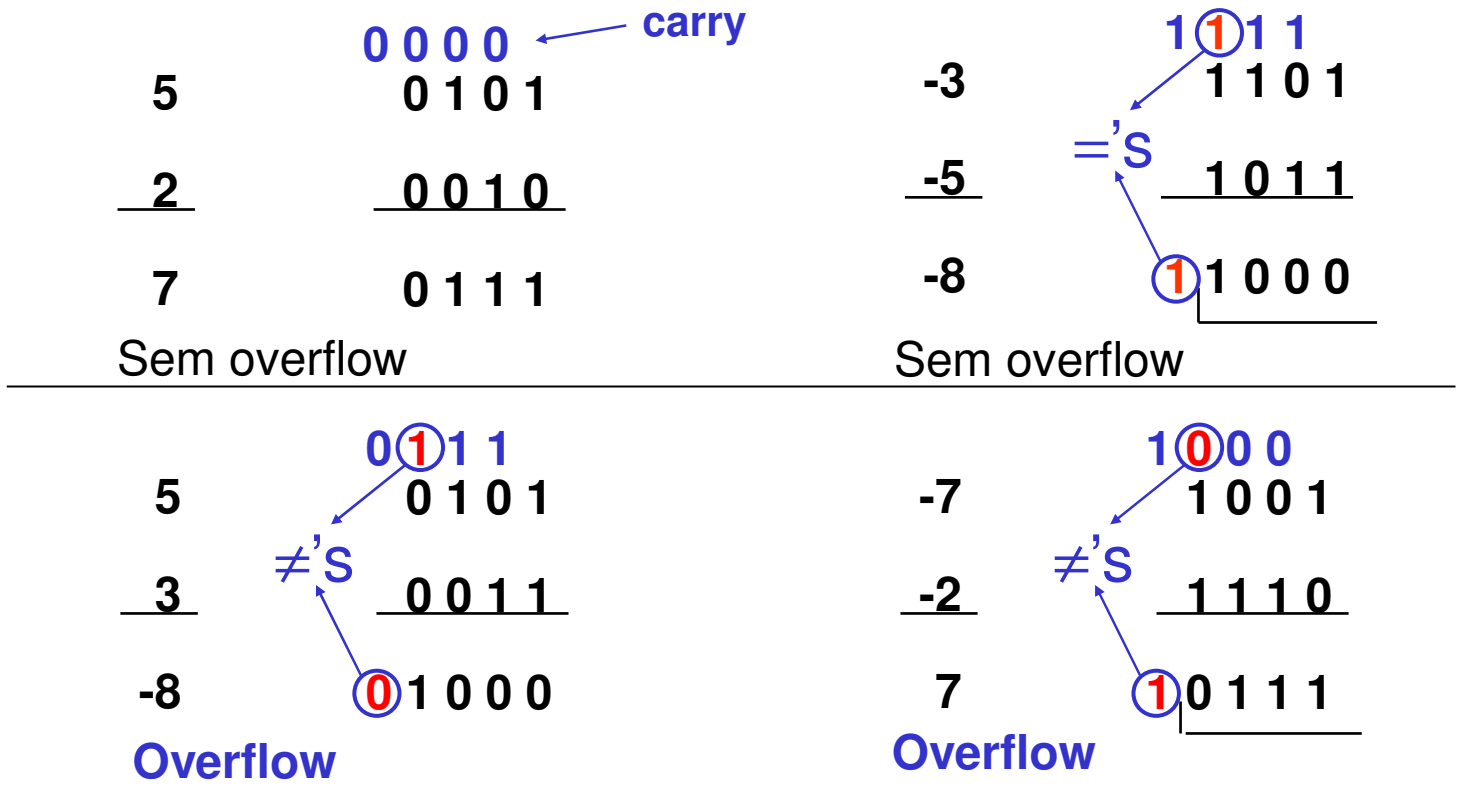
Overflow em complemento para 2





- 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 de sinal não é igual ao *carry-out* , ou seja, quando:

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

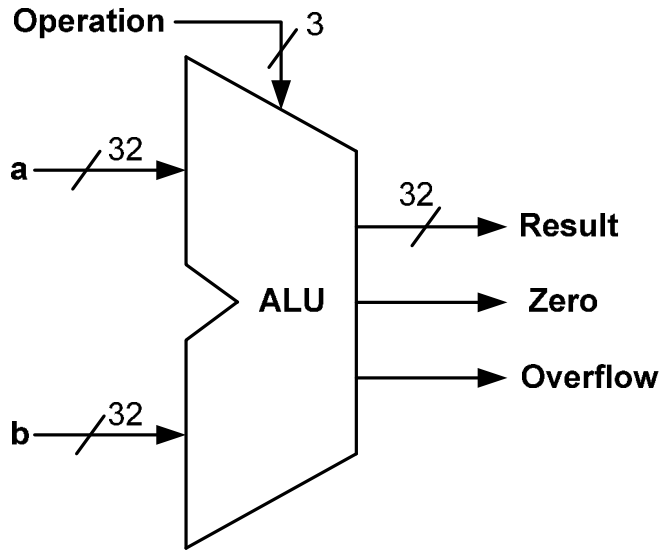
Overflow em operações aritméticas

- **Em operações 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*
- **Em operações 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):
    - **$OVF = R_{n-1} \cdot \overline{A_{n-1}} \cdot \overline{B_{n-1}} + \overline{R_{n-1}} \cdot A_{n-1} \cdot B_{n-1}$**
- Como fazer a deteção de *overflow* em operações com e sem sinal no MIPS?

Construção de uma ALU de 32 bits

- A ALU deverá implementar as operações:
  - AND, OR
  - ADD, SUB
  - SLT (set if less than)
- Deverá ainda:
  - Detetar e sinalizar *overflow*
  - Sinalizar resultado igual a zero

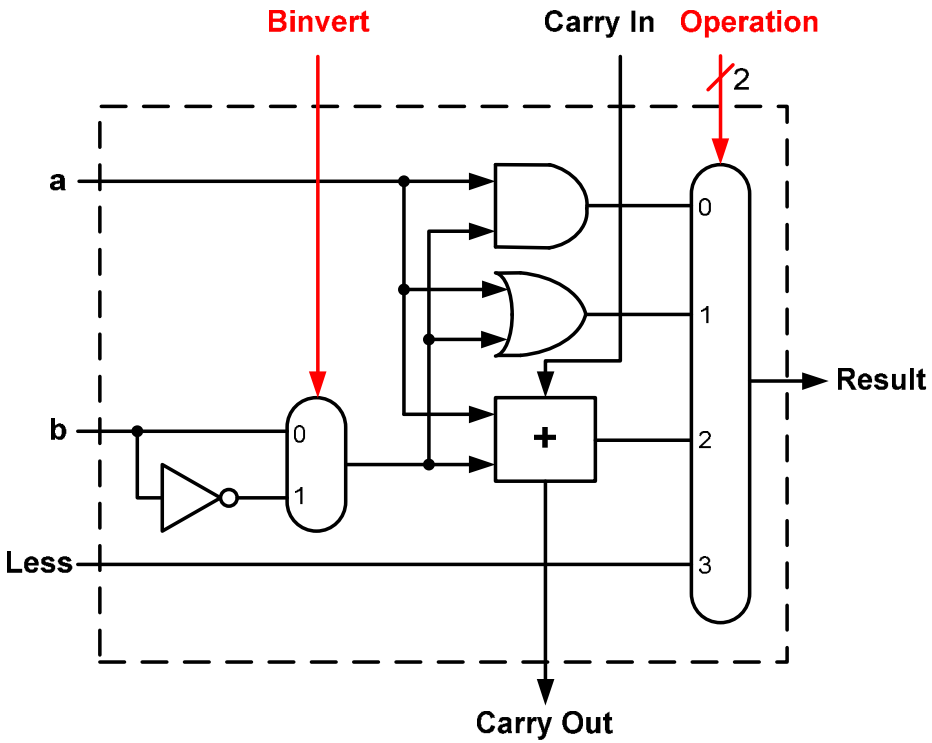
Operation	ALU Action
0 0 0	And
0 0 1	Or
0 1 0	Add
1 1 0	Subtract
1 1 1	Set if less than



Bloco funcional correspondente a uma ALU de 32 bits

Construção de uma ALU de 32 bits

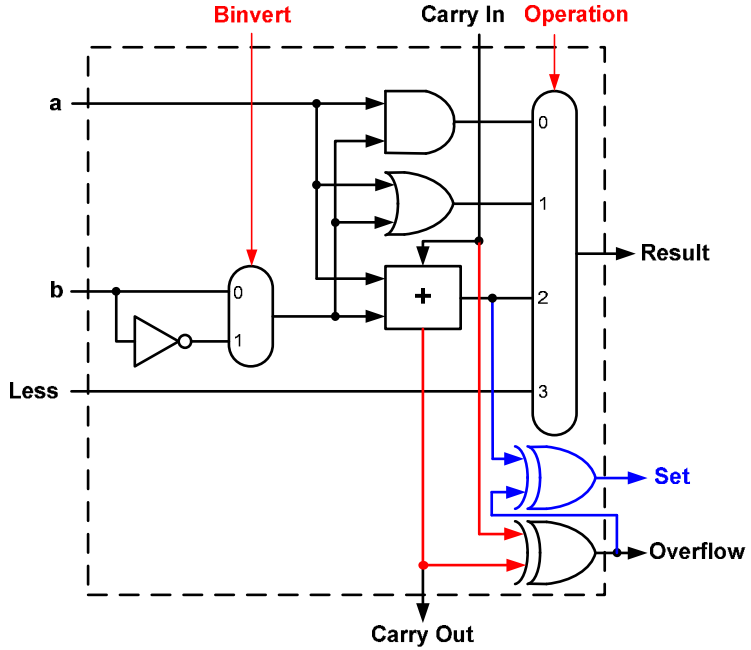
Construção de uma ALU básica de 1 bit:



- Esta ALU permite efetuar as operações aritméticas de adição e subtração, e as operações lógicas AND e OR
- A operação é selecionada pelo sinal *Operation* (2 bits: 00 – AND, 01 – OR, 10 – ADD, 11 – SLT)
- A subtração obtém-se colocando um "1" em *Binvert* e *Carry In*

Construção de uma ALU de 32 bits

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

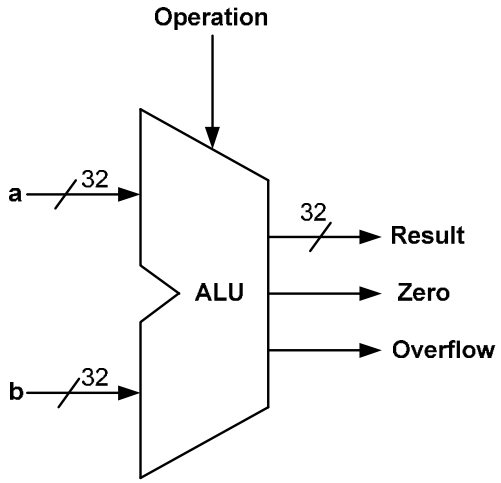


- Detecção de overflow: overflow=carry_in ⊕ carry_out, no bit mais significativo da ALU
- Esta ALU permite ainda efectuar a operação SLT (*set if less than*)
- A operação SLT é realizada através da operação (a-b):
  - saída *Set=1* se a < b → (a-b) < 0
  - saída *Set=0* se a ≥ b → (a-b) ≥ 0

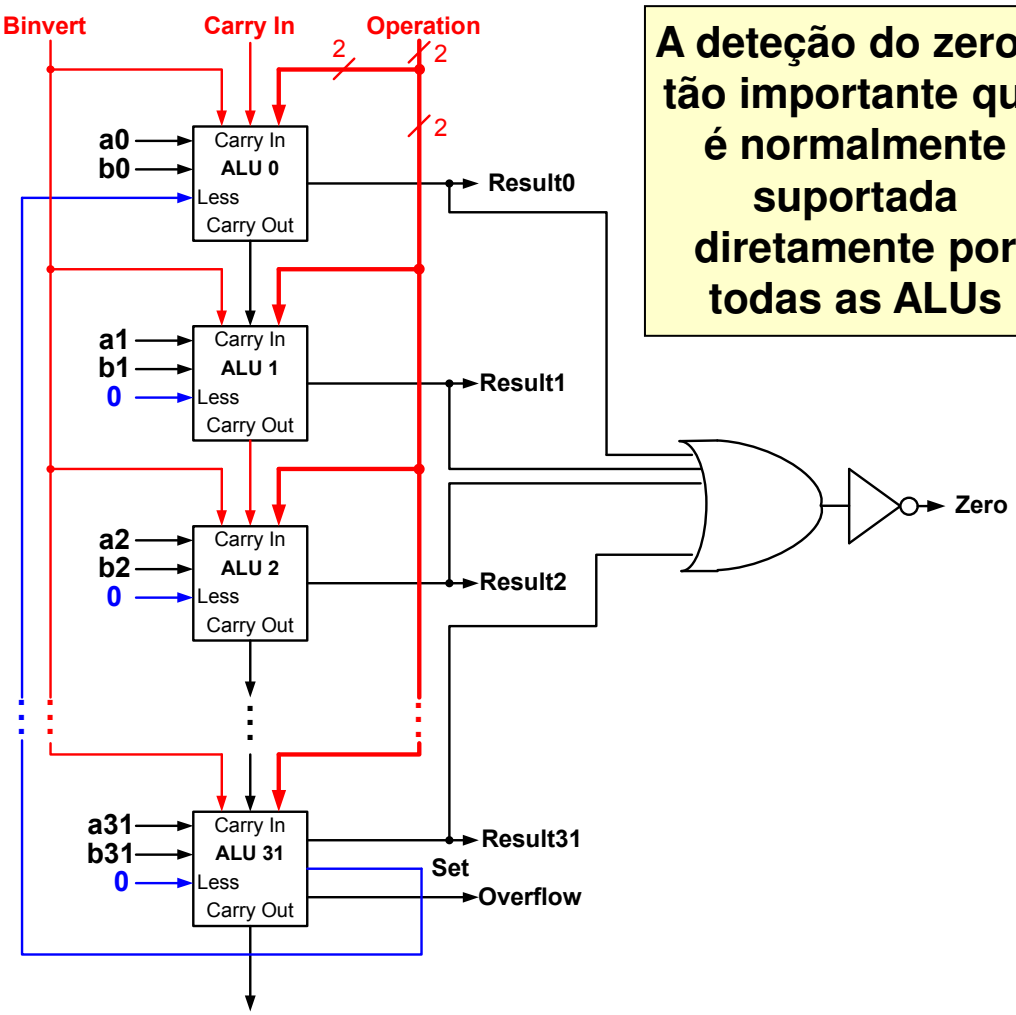
- Na operação de subtracção (a+(-b)) o bit mais significativo do resultado é "1" se a < b e "0" se a ≥ b. Esse bit pode, assim, ser usado para a implementação da instrução SLT
- No entanto, quando ocorre overflow, o bit mais significativo do resultado vem trocado, pelo que, nessa situação, é necessário negá-lo para que a saída *set* seja correcta

Construção de uma ALU de 32 bits

Expansão para 32 bits em *ripple carry*:



Bloco funcional correspondente a uma ALU de 32 bits



A deteção do zero é tão importante que é normalmente suportada diretamente por todas as ALUs

Construção de uma ALU de 32 bits

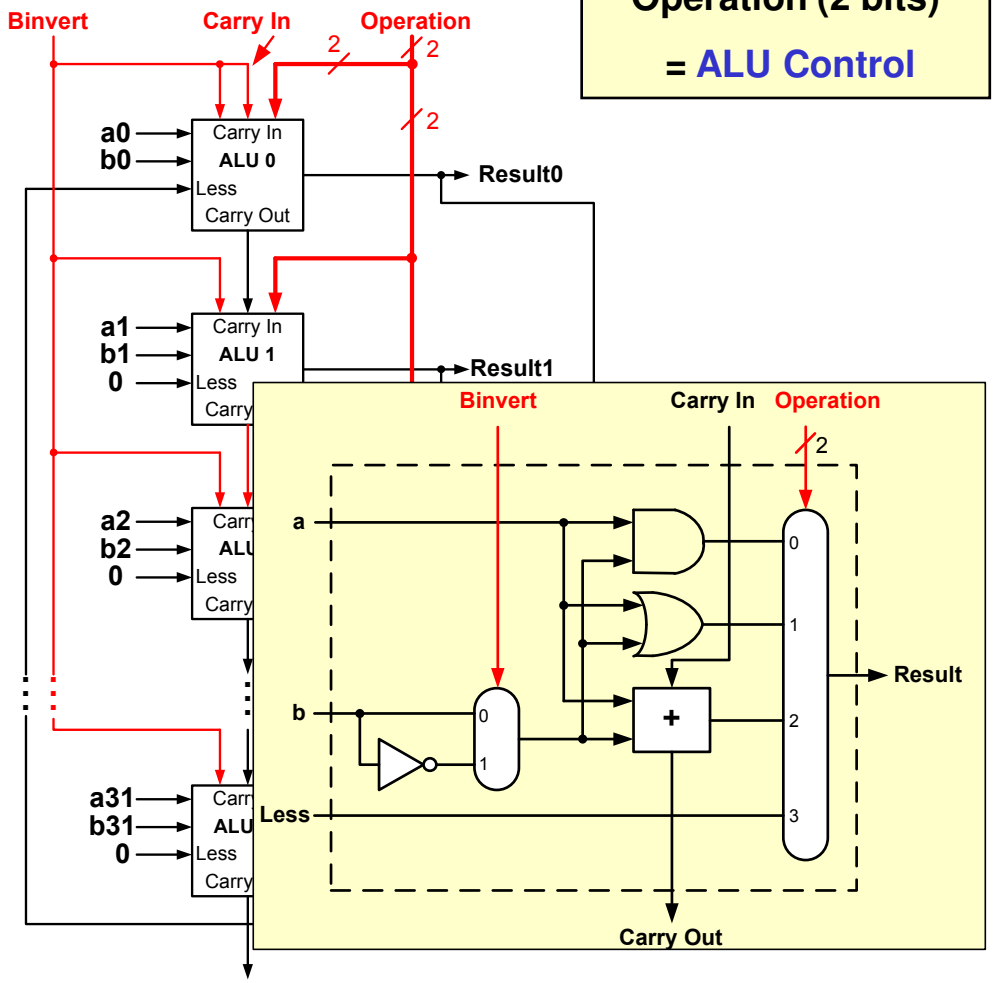
Sinais de controlo da ALU

Os sinais de controlo directo da ALU podem ser reduzidos a três, uma vez que os sinais *Binvert* e *Carry In* podem ser combinados num só.

ALU Control	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

Bit "Binvert"

Binvert (1 bit) + Operation (2 bits) = ALU Control



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;

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); -- complemento para 2 (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)
```

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

Construção de uma ALU de 32 bits (continuação)

```
  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 <= (others => '0');
          end if;
       when others =>
          s_res <= (others => '-');
    end case;
  end process;
end Behavioral;
```

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

ALU – resultado da simulação