

## Aula 10

• 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 básica de 1 bit

• Expansão da ALU de 1 bit para 32 bits

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira e Silva

• Era uma vez, num país longínquo, um conselho de ministros...

• O sr. Primeiro Ministro dirigiu-se aos Srs. Ministros, questionando-os sobre o valor médio no aumento dos impostos que deveria ser feito em 2013, para equilibrar as contas desse país

***Digam-me senhores ministros***

***Nesta situação extraordinária***

***Que % aumentar***

***Aos impostos da maralha***

“Fácil”, disse o Ministro Adjunto e dos Assuntos Particulares: eu acho que devia ser **00000101** %

Não, retorquiu a Ministra da Justiça, para a qual devia ser **01000011 01001001 01001110 01000011 01001111** %

Não concordo, contestou o Ministro da Saúde, de quem devia ser **01010110** %

Nada disso, insurgiu-se o Ministro da Educação. Para resolver o problema necessário um aumento de **01000000101000000000000000000000** %

Por todos os deuses levantou-se o Ministro das Finanças, irritado com um cego não visto que resolve com um aumento de **10000100** %

O Primeiro Ministro desse país longínquo conhecia como um grande especialista em angariar cursos de formação, 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 na resposta dos ministros. A resposta dada por cada um 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 colecções de bits sem sentido

O Ministro Adjunto e dos Assuntos Parlamentares codificou a sua resposta em **binário**:  $10000101_2 = 5_{10}$  %

A Ministra da Justiça usou **ASCII**:

**01000011 01001001 01001110 01000011 01001111** = "CINCO" %

O Ministro da Saúde usou **ASCII** mas para representar numerais romanos: **01010110** = "VI" = 6 %

O Ministro da Educação e Ciência usou representação **flutuante**:

**01000000101000000000000000000000** =  $1.01_2 \times 2^2 = 5$  %

O Ministro de Estado e das Finanças usou **excesso de  $2^{n-1}-1$**  (com  $n=8$ , excesso de 127): **10000100**<sub>2</sub> =  $5_{10}$  %

## Representação de inteiros

No sistema **posicional**, cada algarismo que compõe um número tem um peso que depende da sua posição no número. Os pesos são dados pelos símbolos do alfabeto usado.

Um número com dígitos  $d_{n-1} d_{n-2} \dots d_1 d_0$

representado neste sistema, pode ser decomposto num polinómio da forma

$$d_{n-1} \cdot b^{n-1} + d_{n-2} \cdot b^{n-2} + \dots + d_1 \cdot b^1 + d_0 \cdot b^0$$

em que **b** é a base de representação e corresponde ao **alfabeto** do sistema.

Exemplos:

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

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

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

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

## Representação de inteiros

Uma vez que um computador é um sistema digital, a representação de inteiros faz-se sempre em base 2 (símbolos 0 e 1).

Por outro lado, como o espaço de armazenamento é limitado, a representação é necessariamente limitada. **Tipicamente, um inteiro pode ocupar um número de bits igual à dimensão de um registo do CPU.**

A gama de valores inteiros representáveis é limitada, correspondendo ao número máximo de combinações que podem ser representadas com o número de bits que compõem um registo.

No MIPS, um inteiro ocupa 32 bits, pelo que o número de inteiros representáveis será:

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

## Representação de inteiros

Os circuitos que realizam operações aritméticas são limitados a um número finito de dígitos (bits), sendo igual à dimensão dos registos internos do CPU.

Os circuitos aritméticos operam assim, ou seja em **mod(2<sup>n</sup>)** em que  $n$  é o número de bits de representação.

O maior valor que um resultado aritmético pode ser, portanto, **2<sup>n</sup> - 1**, sendo o valor inteiro imediatamente a seguir o valor zero (**representação circular**).

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

$$11001011 + 00110111 = 1\ 00000010$$

**Carry** é o resultado que não cabe num registo de 8 bits (se os operandos são do tipo **unsigned** o carry a 1 sinaliza a ocorrência de **overflow**).

Resultado com 8 bits

## Representação de inteiros negativos

A representação de números positivos é **amplamente** 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:

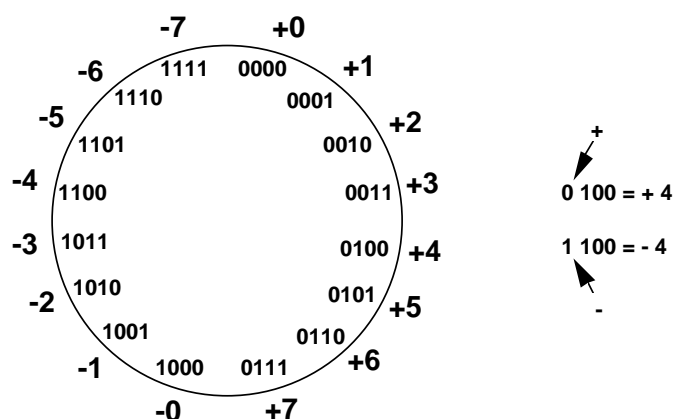
sinal e módulo

complemento para um

complemento para dois

Por uma questão de simplicidade vamos admitir, na discussão subsequente, que a dimensão do registo interno do PC é de 4 bits

## Representação em sinal e módulo



O bit mais significativo é o sinal: 0 = positivo, 1 = negativo

A magnitude é representada pelos 3 LSBs: 000 a 111

Gama de representação para n bits =  $\pm 2^{n-1}$

2 representações para 0

## Representação em sinal e módulo

Este método de representação de inteiros apresenta problemas do ponto de vista da implementação na ULA

Existem duas representações distintas para o valor zero

É necessário comparar as magnitudes dos operandos para determinar o sinal do resultado

É necessário implementar um somador e um subtrator

O bit de sinal tem de ser tratado independentemente dos restantes

## Representação em complemento para um

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

$$\bar{N} = (2^n - 1) - N \quad \text{em que } n \text{ é o número de bits da representação}$$

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

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

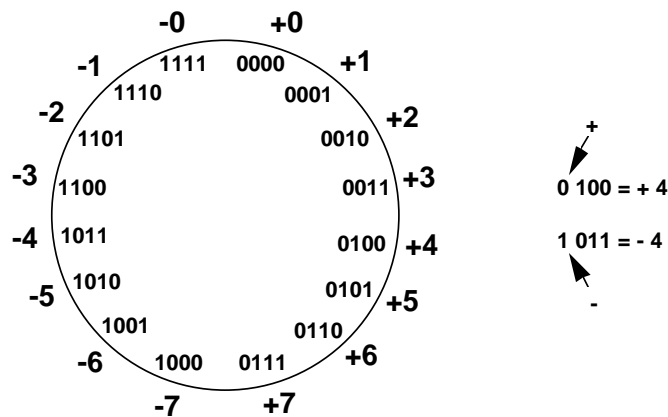
$$2^n = 2^4 = 10000$$

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

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

O complemento para 1 pode ser calculado negando um a um os bits que compõem o número.

## Representação em complemento para um



O bit mais significativo também pode ser interpretado como sinal:

0 = positivo, 1 = negativo

Há 2 representações para 0

A subtração faz-se adicionando o complemento para

## Exemplos de adições e subtrações em complemento para um

Quando ocorre *carry* este tem de ser somado ao resultado intermédio!

$$\begin{array}{r}
 4 \quad 0100 \\
 + 3 \quad 0011 \\
 \hline
 7 \quad 0111 \\
 \\
 4 \quad 0100 \\
 - 3 \quad 1100 \\
 \hline
 1 \quad 10000 \\
 \text{carry final} \rightarrow 1 \\
 \hline
 0001
 \end{array}$$

$$\begin{array}{r}
 -4 \quad 1011 \\
 + 3 \quad 0011 \\
 \hline
 -1 \quad 1110 \\
 \\
 -4 \quad 1011 \\
 + (-3) \quad 1100 \\
 \hline
 -7 \quad 10111 \\
 \text{carry final} \rightarrow 1 \\
 \hline
 1000
 \end{array}$$

## Adições e subtrações em complemento para 1

Porque deve ser adicionado o *carry* final (*end-around carry*)?

é equivalente a subtrair  $2$  e somar  $1$   $\rightarrow (-2^n + 1)$

1)  $M \dot{-} N$  com  $M > N$  (resultado é positivo:  $M \dot{-} N$ )

$$M \dot{-} N = \bar{M} + N = M + (2^n - N) = (M \dot{-} N) + 2^n - 1$$

O resultado anterior só é correcto se for adicionado  $1$

2)  $-M + (-N)$  (resultado é negativo:  $2^n - 1 \dot{-} (M + N)$ )

$$-M + (-N) = \bar{M} + \bar{N} = (2^n \dot{-} 1 \dot{-} M) + (2^n \dot{-} 1 \dot{-} N) = (2^n \dot{-} 1 \dot{-} (M + N)) + 1$$

Representação correcta do resultado

O resultado anterior só é correcto se for adicionado  $1$

## 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:

$$N^* = 2^n - N \quad \text{em que } n \text{ é o número de bits da representação}$$

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

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

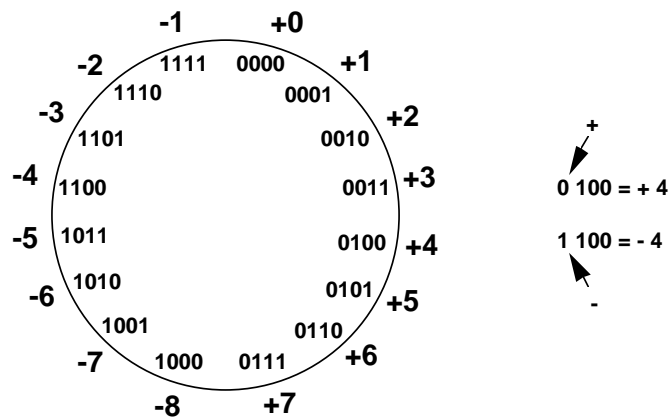
$$2^n = 2^4 = 10000$$

$$2^n \dot{-} N = 10000 \dot{-} 0101 = N^*$$

O complemento para 2 pode ser calculado obtendo o complemento para 1 e somando 1 ao resultado



## Representação em complemento para dois



O bit mais significativo também pode ser interpretado como sinal:

0 = positivo, 1 = negativo

Uma única representação para 0

Codificação assimétrica (mais um negativo do que positivo)

## Representação em complemento para dois

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

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

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

**Exemplo:** Qual o valor representado pela quantidade  $10100101_2$ , supondo uma representação com 8 bits e uma codificação em complemento para 2?

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

**R2:** Complemento para 2 de  $10100101 = 01011010 + 1$   
 $= 01011011_2 = 5B_{16} = 91_{10}$ . Ou seja, o valor representado é  $-91_{10}$

## Exemplos de adições e subtrações em complemento para 2

$$\begin{array}{r} 4 \quad 0100 \\ + 3 \quad 0011 \\ \hline 7 \quad 0111 \end{array}$$

$$\begin{array}{r} 4 \quad 0100 \\ - 3 \quad 1101 \\ \hline 1 \quad 10001 \end{array}$$

$$\begin{array}{r} -4 \quad 1100 \\ + (-3) \quad 1101 \\ \hline -7 \quad 11001 \end{array}$$

$$\begin{array}{r} -4 \quad 1100 \\ + 3 \quad 0011 \\ \hline -1 \quad 1111 \end{array}$$

Este esquema simples de adição com sinal torna-se complemento para 2 o preferido para representação de inteiros na arquitectura de computadores

## Adições e subtrações em complemento para 2

Porque pode o último carry-out ser ignorado?

é equivalente a subtrair 2

1)  $M + N$  com  $M > N$  (resultado positivo:  $M + N$ )

$$M + N = M + N^* = M + (2^n - N) = (M + N) + 2$$

O resultado anterior só é correcto se for adicionado 2

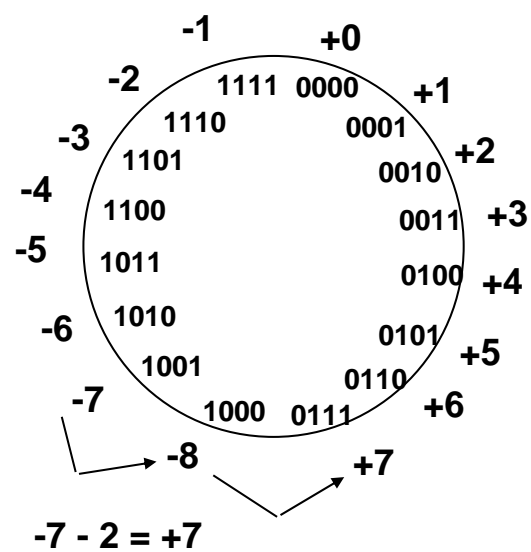
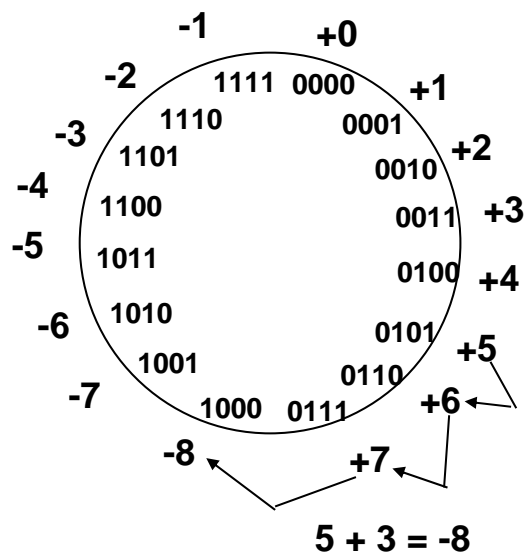
2)  $-M + (-N)$  (resultado negativo:  $-(M + N)$ )

$$-M + (-N) = M^* + N^* = (2^n - M) + (2^n - N) = 2^{n+1} - (M + N) = -(M + N) + 2$$

Representação correcta do resultado

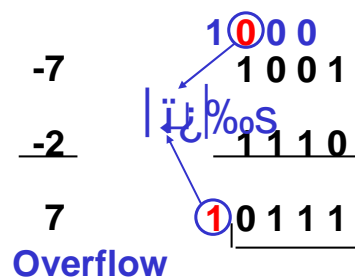
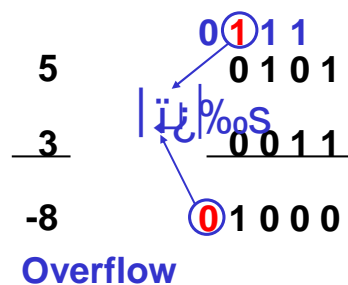
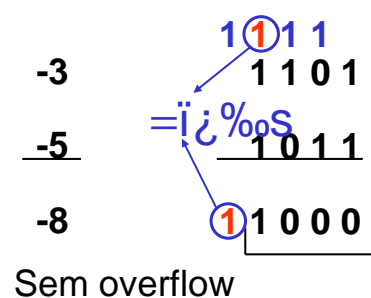
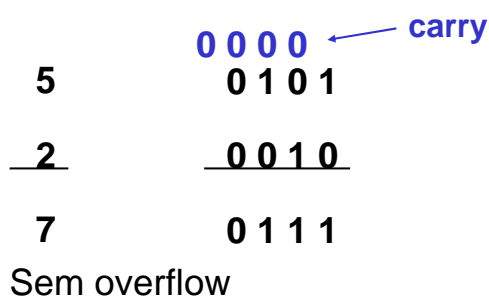
O resultado anterior só é correcto se for adicionado 2

## Overflow em complemento para 2



Ocorre **overflow** quando somamos dois positivos e obtemos um negativo ou somamos dois negativos e obtemos um 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 ( $C_{n-1} \neq C_n$ )

## Overflow em operações aritméticas de adição:

Em operações **sem sinal**:

Quando  $A+B > 2^n - 1$  ou  $A-B < 0$  /  $B > A$

A detecção ocorre quando o bit de carry  $C_n = 1$

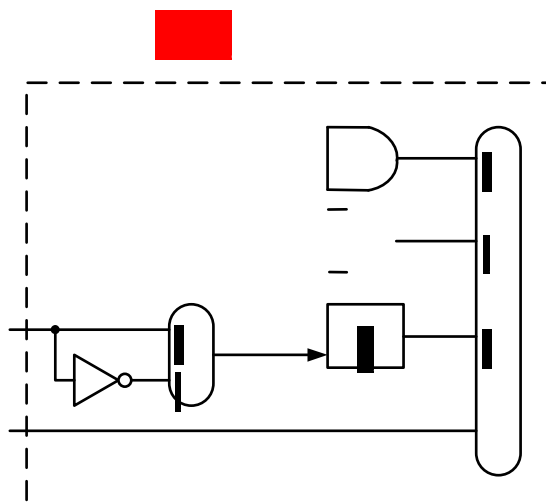
Como fazer a detecção de overflow em operações sem sinal no MIPS?

Em operações **com sinal**:

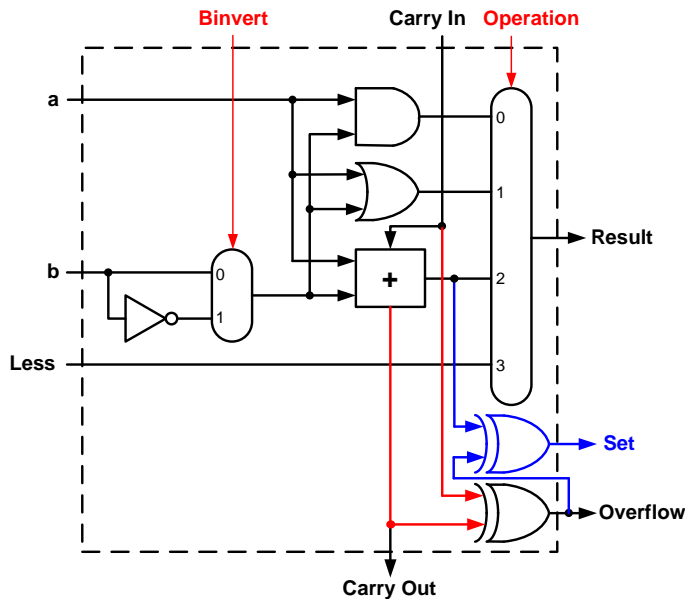
Quando  $A + B > 2^{n-1} - 1$  ou  $A + B < -2^{n-1}$

A detecção ocorre quando  $(C_{n-1} \neq 0 \text{ e } C_n = 0)$  ou  $(C_{n-1} = 0 \text{ e } C_n = 1)$

Ou seja, há overflow quando  $C_{n-1} \neq C_n$



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



Detecção de overflow:

$\text{overflow} = \text{carry\_in} \oplus \text{carry\_out}$ , no bit mais significativo da ALU

Esta ALU permite ainda efectuar a operação **SLT** (set if less than)

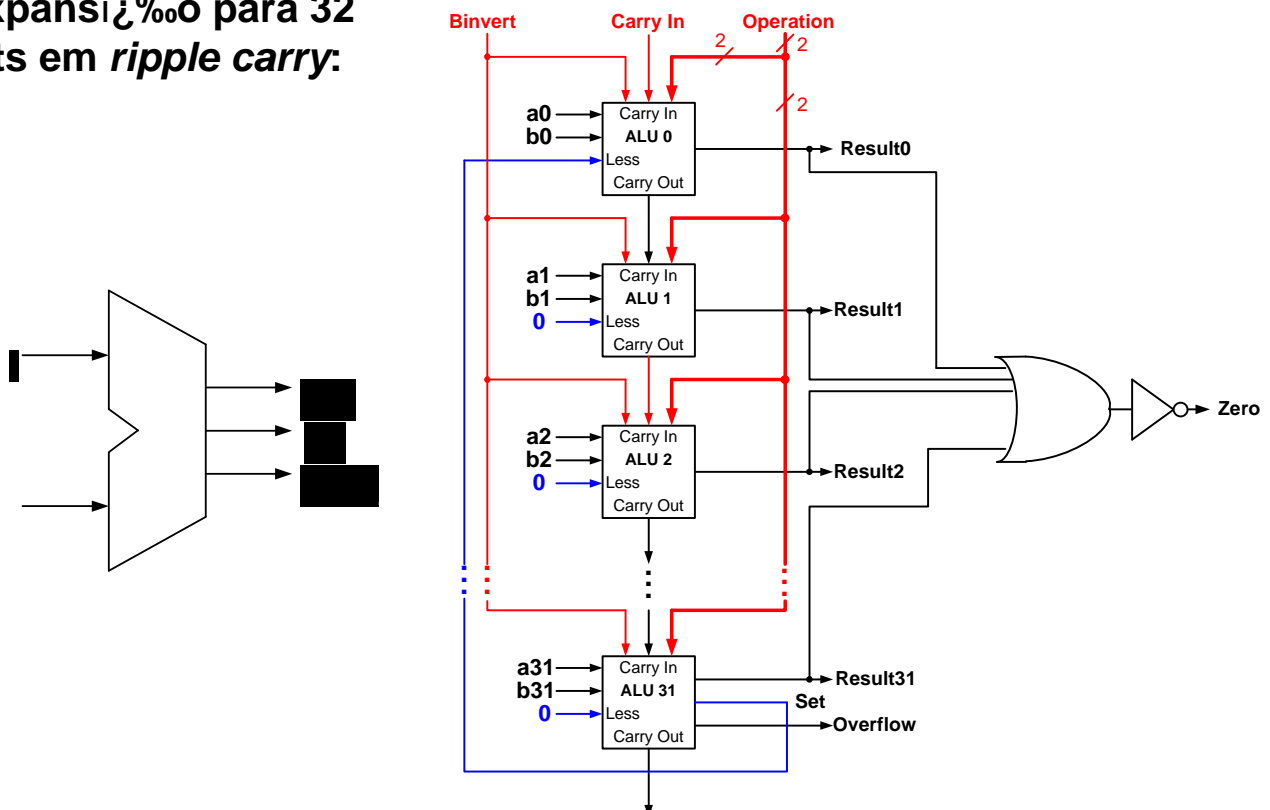
A operação **SLT** é realizada através da operação  $(a-b)$ :

Sai 1 se  $a < b$  |  $(a-b) < 0$

Sai 0 se  $a \geq b$  |  $(a-b) \geq 0$

Na operação de subtracção  $(a+(-b))$  o bit mais significativo do resultado é **1** se  $a < b$  e **0** se  $a \geq b$ . 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 para que a saída seja correcta

## Expansão para 32 bits em ripple carry:



## 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 sinal.

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"

