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 detecção
- Construção de uma ALU básica de 1 bit
- Expansão da ALU de 1 bit para 32 bits

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira, Tomás 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 Parlamentares: eu acho que devia ser **00000101** %
- Não, retorquiu a Ministra da Justiça, para mim devia ser **01000011 01001001 01001110 01000011 01001111** %
- Não concordo, contestou o Ministro da Saúde, eu cá acho que devia ser **01010110** %
- Nada disso, insurgiu-se o Ministro da Educação e Ciência. Para resolver o problema é necessário um aumento de **01000000101000000000000000000000** %
- Por todos os deuses – levantou-se o Ministro de Estado e das Finanças, irritado – só um cego não vê que o *déficit* só se resolve com um aumento de **10000100** %

- O Primeiro Ministro desse país longínquo, conhecido 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 extracção 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**: **00000101_2 = 5_{10}** %

A Ministra da Justiça usou **ASCII**:

01000011 01001001 01001110 01000011 01001111 = "CINCO“ %

O Ministro da Saúde usou **ASCII** mas para representar numeração romana: **01010110 = "V“ = 5** %

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

01000000101000000000000000000000 = $1.01_2 x 2^2$ = 5 %

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

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 $d_{n-1}\, d_{n-2} \ldots d_1\, d_0$

representado neste sistema, pode ser decomposto num polinómio da forma

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

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

- Uma vez que um computador é 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 que compõem um registo.
- No MIPS, um inteiro ocupa 32 bits, pelo que o número de inteiros representável será:

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

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

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

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

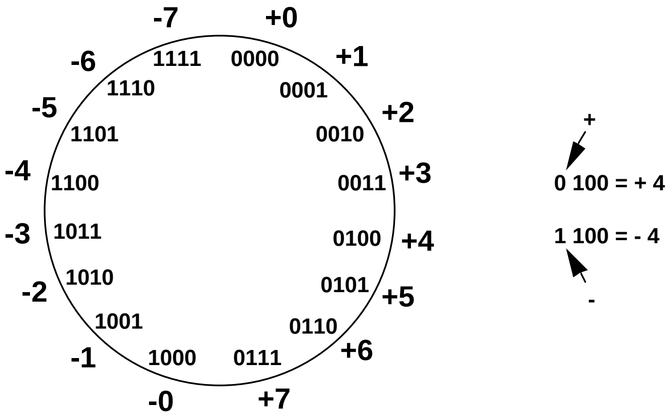
Resultado com 8 bits

Representação de inteiros negativos

- A representação de números positivos é 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 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 subtractor 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)

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

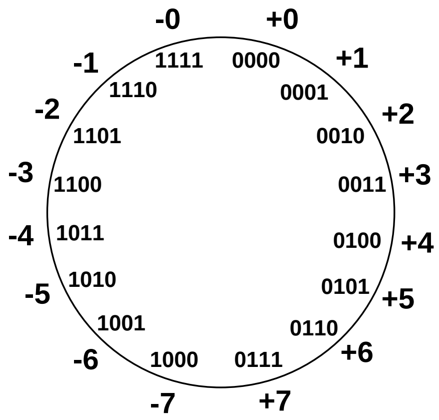
$2^n = 2^4 = 10000$

$(2^n – 1) = 10000 – 1 = 1111$

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

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

Representação em complemento para um



\+

0 100 = + 4

1 011 = - 4

\-

- O bit mais significativo também pode ser interpretado como sinal:
  0 = positivo, 1 = negativo
- Há 2 representações para 0
- A subtracção faz-se adicionando o complemento para 1

Exemplos de adições e subtracções em complemento para 1

4	0100
+ 3	0011
7	0111

-4	1011
+ 3	0011
-1	1110

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

4	0100
- 3	1100
1	10000
carry final	1
	0001

-4	1011
+ (-3)	1100
-7	10111
carry final	1
	1000

Adições e subtracções em complemento para 1

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

É equivalente a subtrair 2^n e somar 1 $\Leftrightarrow$ **$(-2^n + 1)$**

1) M – N com M > N (resultado é positivo: M – N)

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

O resultado anterior só é correcto se for adicionado $(-2^n + 1)$

2) -M + (-N) (resultado é negativo: $2^n - 1 - (M + N)$) c/ $(M + N) < 2^{n-1}$

$$-M + (-N) = \overline{M} + \overline{N} = (2^n - 1 - M) + (2^n - 1 - N) = \underbrace{2^n - 1 - (M + N)}_{\text{Representação correcta do resultado}} + 2^n - 1$$

O resultado anterior só é correcto se for adicionado $(-2^n + 1)$

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:

$N^* = 2^n - N$ em que "n" é o número de bits da representação

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

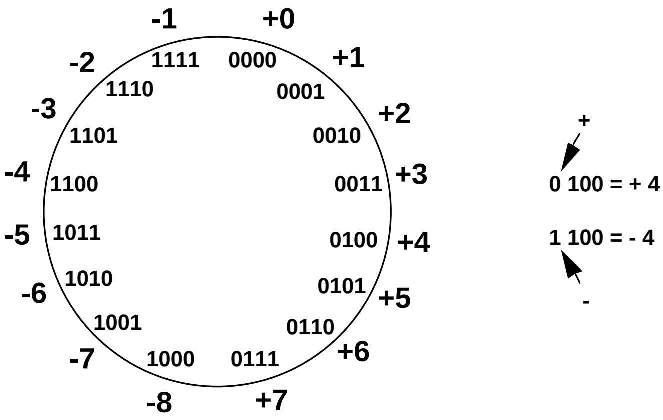
$N = 5_{10} = \mathbf{0101}_2$

$2^n = 2^4 = 10000$

$2^n - N = 10000 - 0101 = \mathbf{1011 = N^*}$

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 positivos)

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 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: $10100101_2 = -(1x2^7) + (1x2^5) + (1x2^2) + (1x2^0) = -128+32+4+1 = -91_{10}$

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

Exemplos de adições e subtracções em complemento para 2

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 arquitectura de computadores

Adições e subtracções em complemento para 2

Porque pode o último *carry-out* ser ignorado?

É equivalente a subtrair 2^n

1) **M – N com M > N (resultado é positivo: M – N)**

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

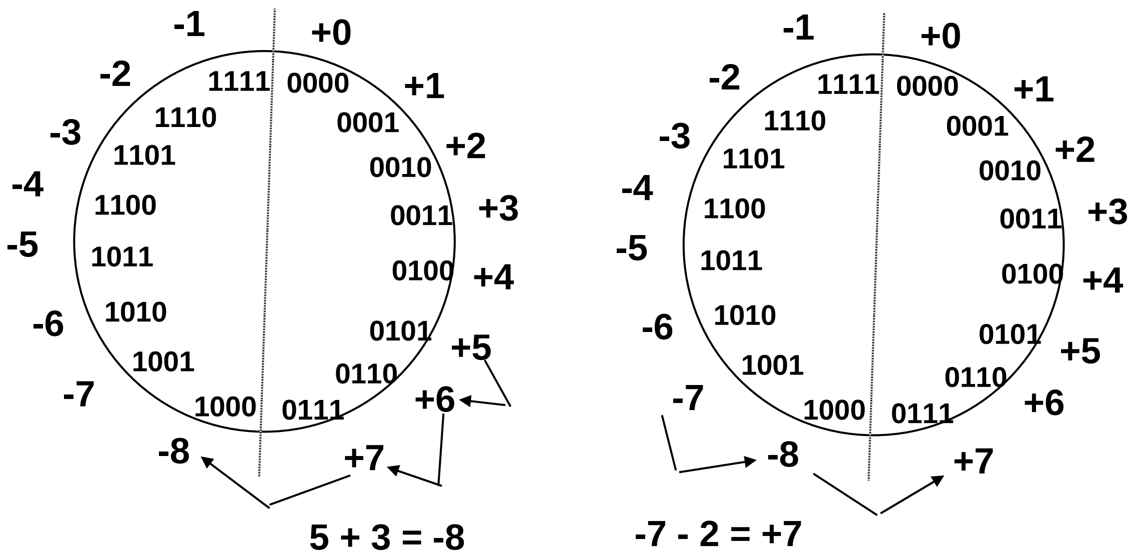
O resultado anterior só é correcto se for adicionado (-2^n)

2) **-M + (-N) (resultado é negativo: $2^n - (M + N)$) c/ $(M + N) \leq 2^{n-1}$**

$$-M + (-N) = M^* + N^* = (2^n - M) + (2^n - N) = \underbrace{2^n - (M + N)}_{\text{Representação correcta do resultado}} + 2^n$$

O resultado anterior só é correcto se for adicionado (-2^n)

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

	0 0 0 0 ← carry		1 1 1 1
5	0 1 0 1	-3	1 1 0 1
2	0 0 1 0	-5	1 0 1 1
7	0 1 1 1	-8	1 1 0 0 0
Sem overflow		Sem overflow (=’S)	

	0 1 1 1		1 0 0 0
5	0 1 0 1	-7	1 0 0 1
3	0 0 1 1	-2	1 1 1 0
-8	0 1 0 0 0	7	1 0 1 1 1
Overflow (≠’S)		Overflow (≠’S)	

A situação de ***overflow* ocorre** quando o *carry-in* do bit de sinal não é igual ao *carry-out* ($C_{n-1} \oplus C_n = 1$)

Overflow em operações aritméticas de adição:

- **Em operações sem sinal:**

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

  A detecção dá-se quando o bit de *carry* $C_n = 1$

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

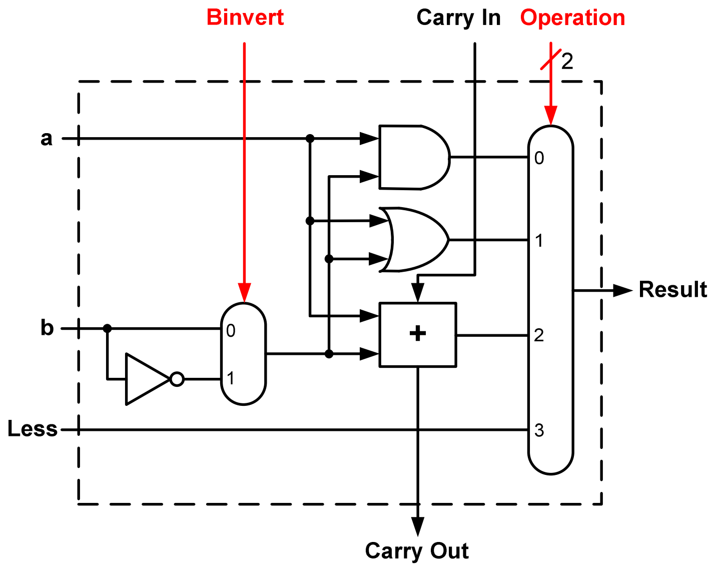
- **Em operações com sinal:**

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

  A detecção dá-se quando $\underbrace{(C_{n-1} = 1 \text{ e } C_n = 0) \text{ ou } (C_{n-1} = 0 \text{ e } C_n = 1)}$

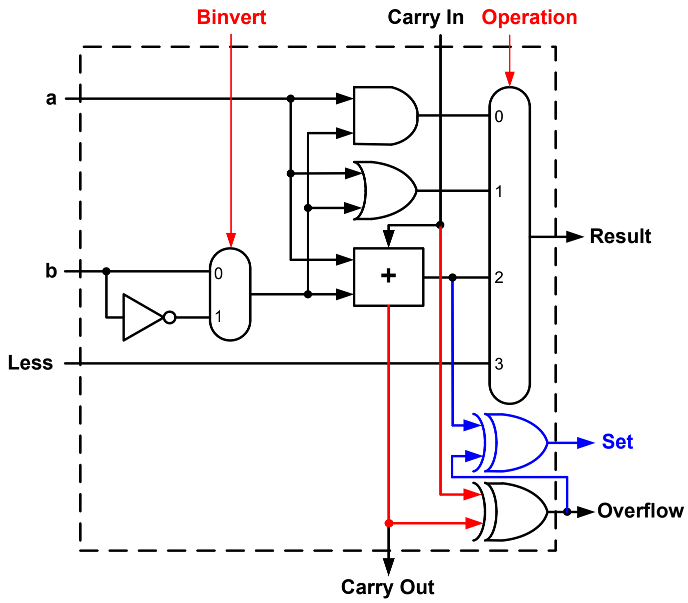
Ou seja, há *overflow* quando $C_{n-1} \oplus C_n = 1$

Construção de uma ALU básica de 1 bit:



- Esta ALU permite efectuar as operações aritméticas de **adição** e **subtracção**, e as operações lógicas **AND** e **OR**
- 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***

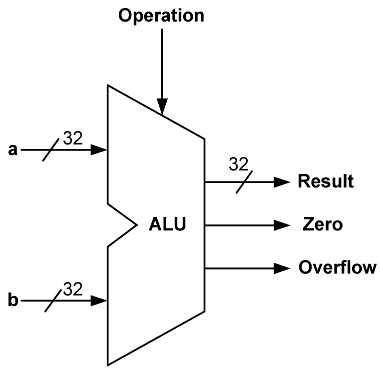
ALU básica de 1 bit, com detecção de *overflow*:



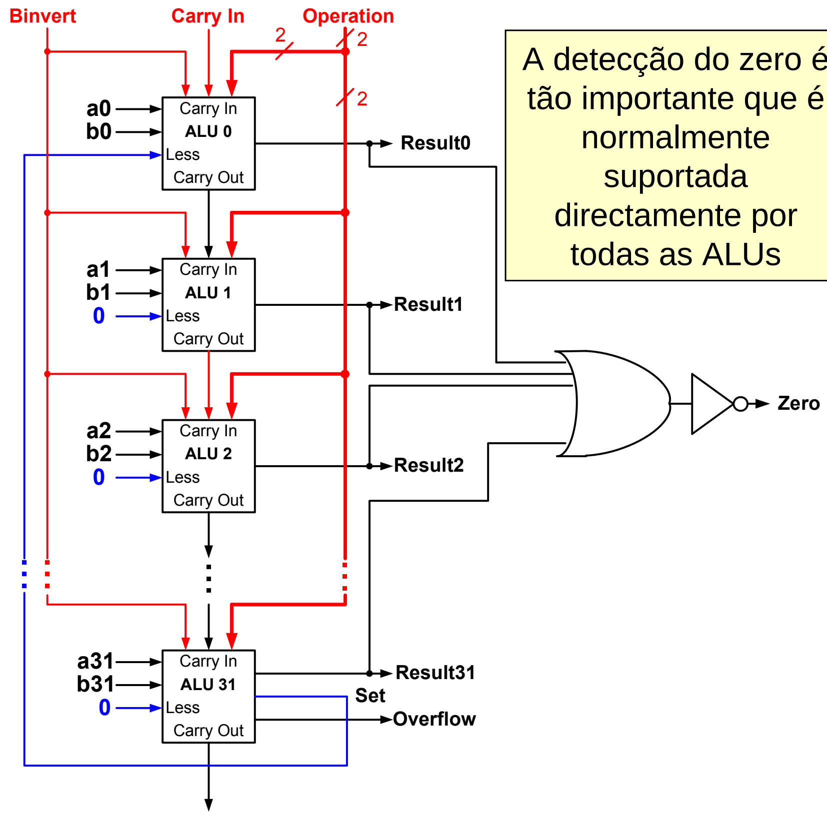
- Detecção de overflow: **overflow=carry_in ⊕ carry_out**, no bit mais significativo da ALU
- Esta ALU permite ainda efectuar a operação **SLT** (*set if less than*)
- A operação SLT é realizada através da operação (a-b):
  - saída ***Set=1*** se **a < b** → (a-b) < 0
  - saída ***Set=0*** se **a ≥ b**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

Expansão para 32 bits em *ripple carry*:



Bloco funcional correspondente a uma ALU de 32 bits



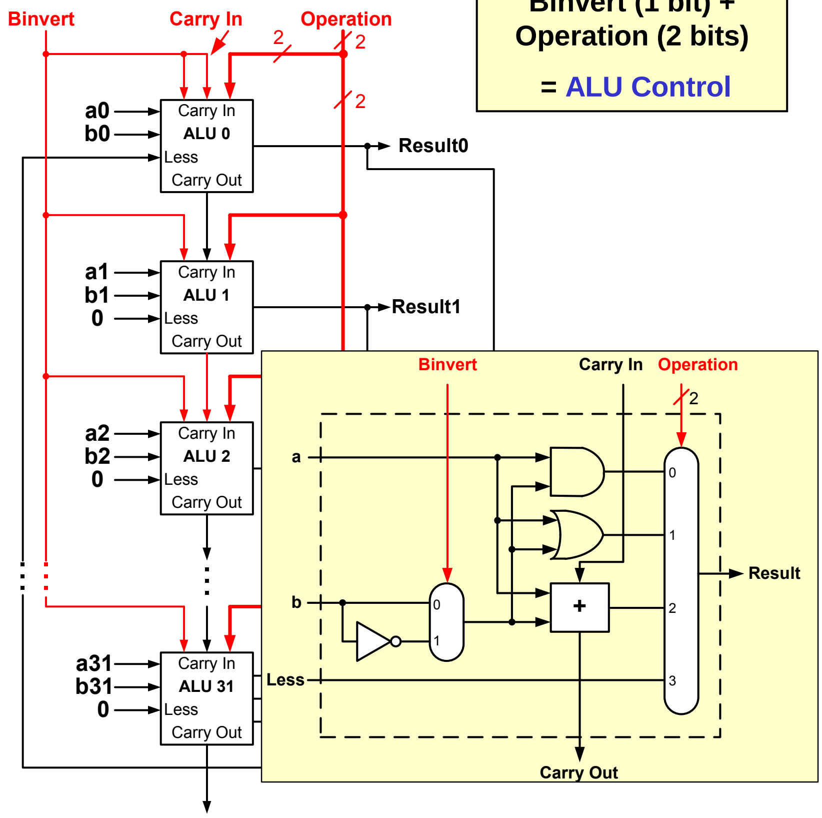
A detecção do zero é tão importante que é normalmente suportada directamente por todas as ALUs

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.

Binvert (1 bit) + Operation (2 bits)

= ALU Control



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"