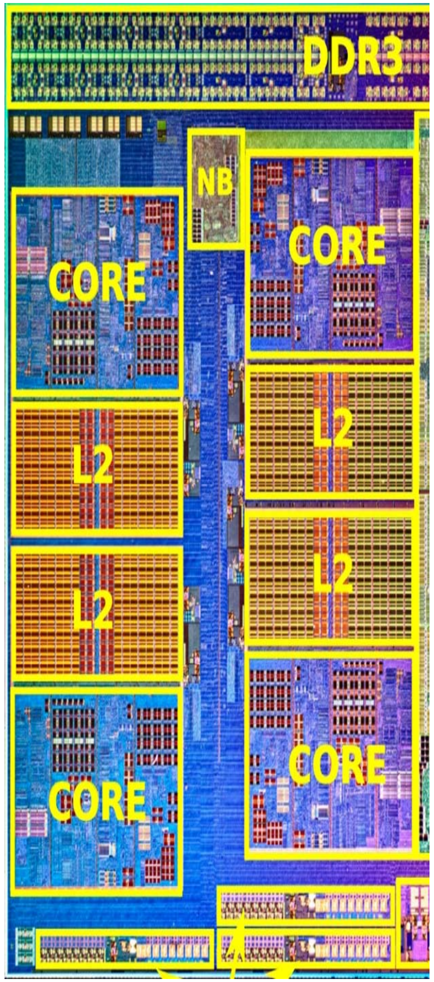
0273359 – Arquitetura e Organização de Computadores 1



Fonte: http://www.techspot.com/article/904-history-of-the-personal-computer-part-5/

ULA / Circuitos Somadores

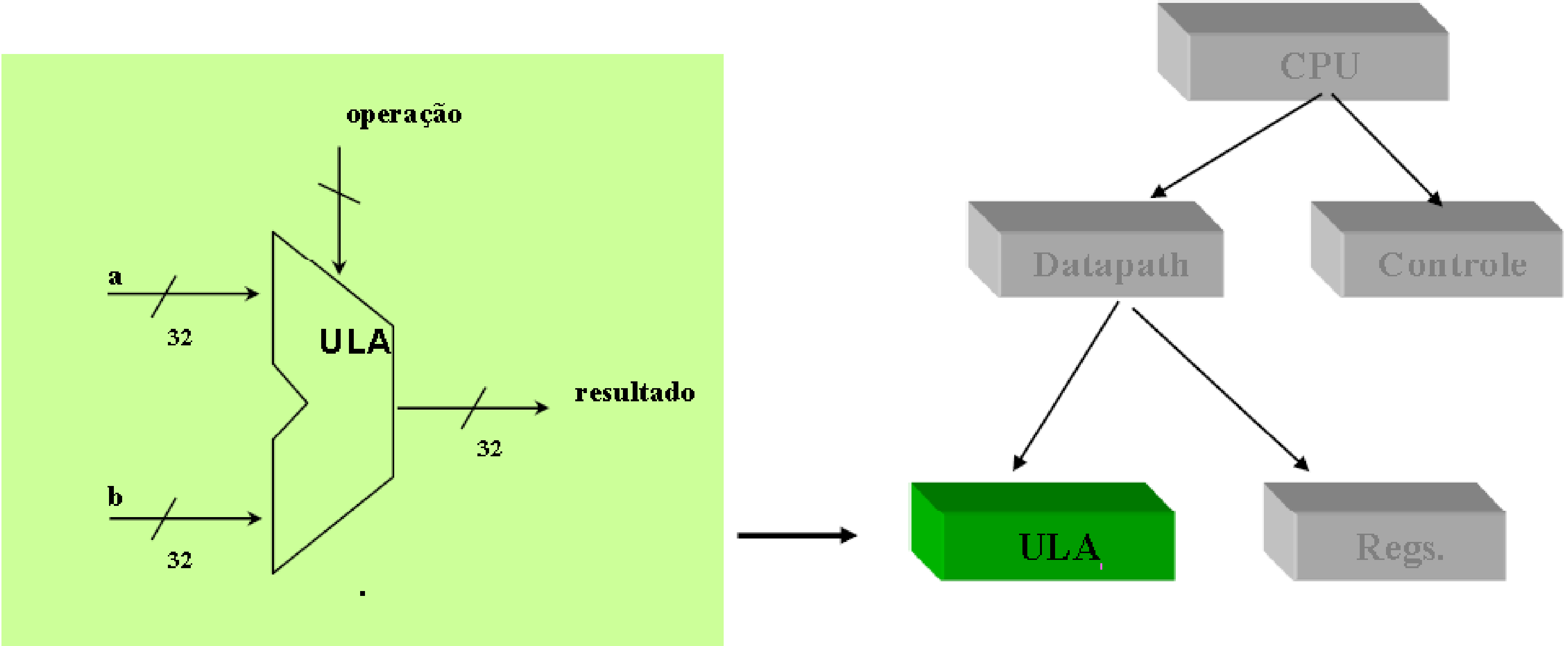
Luciano de Oliveira Neris
luciano@dc.ufscar.br

Adaptado de slides do prof. Marcio Merino Fernandes

Departamento de Computação
Universidade Federal de São Carlos

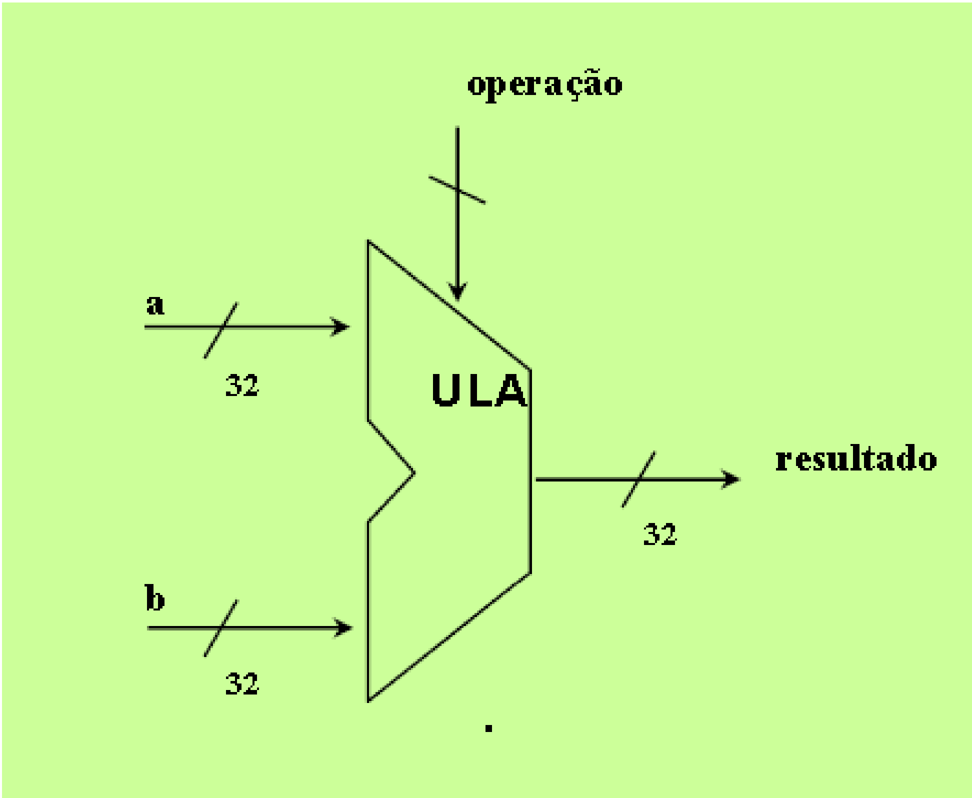
Unidade Lógica e Aritmética (ULA)

- **ULA: "Motor" do computador** -> dispositivo que executa operações aritméticas (add, sub, etc) e lógicas (AND, OR, etc).



Unidade Lógica e Aritmética (ULA)

- Como projetar e implementar uma ULA ?



Números

Números

- Bits são apenas bits (sem nenhum significado inerente)
  - convenções definem a relação entre bits e números
- Números Binários (base 2)
  - 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  - decimal: 0...2n-1
- Naturalmente resultam em algumas sofisticações:
  - números são finitos (overflow)
  - números fracionários e reais
  - números negativos
- Como representamos números negativos?

... ou seja, que padrões de bits representam os números?

Números Negativos

- Representação em Complemento de 2 (C2):
  - Desenvolvida para tornar os circuitos aritméticos mais simples (e consequentemente mais rápidos)
  - Se o número for positivo:
    - Representação binária normal (bit mais significativo= 0)
  - Se o número for negativo:
    - Comece com o número binário positivo
    - Inverta todos os bits
    - Adicione 1 ao resultado
    - * Número Negativo em C2: primeiro bit = 1

Números

- Números com Sinal x Números sem Sinal
  - unsigned 1-byte: 0:255
  - signed 1-byte: -128:+127

- Inteiros são armazenados utilizando-se a notação complemento de 2.
  - Ex: Como armazenar -23 utilizando-se 2-bytes ?
    - 23 em binárito usando 2 bytes é 0000 0000 0001 0111
    - Calculando o complemento: 1111 1111 1110 1000
    - Somar 1: 1111 1111 1110 1001
    - -23 representado como complemento de 2 em 2 bytes= FFE9

- Vantagem do C-2: O mesmo hardware para efetuar adição trabalha da mesma forma, independente de se interpretar o número como sendo com ou sem sinal.

- A ISA normalmente contém instruções distintas para se trabalhar com números com ou sem sinal.

Números????

- *Dado um número X de N bits, representado em complemento de 2:*

$$X + (-X) = 2^N$$

Se ambos estiverem representados em C2, vale a relação acima,
ou seja, (-X) pode ser visto como o complemento de X para se chegar a 2^N

Ex1: 4 bits, X= 5:

```
   1001    (5)
 + 0111    (-5)
-----------
  10000    (2^N)
```

Ex2: 4 bits, X= -6:

```
   1010    (-6)
 + 0110    (+6)
-----------
  10000    (2^N)
```

Números na Arquitetura MIPS

- *Números de 32 bits com sinal em C2:*

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
...
maxint
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$ **minint**
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

Sequencia: 0000...00 até 1111...11:
começa em 0, soma-se 1 até maxint (0111..11), soma-se 1 até minint (1111..11)

Operações em Complemento de 2

- Números Binários Negativos - Complemento de 2

(a) 01010110

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	0	1	0	1	1	0

$64 + 16 + 4 + 2 = \mathbf{+86}$

(b) 10101010

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	0	1	0	1	0

$-128 + 32 + 8 + 2 = \mathbf{-86}$

Operações em Complemento de 2

- Negar um número em complemento de 2: inverter todos os bits e somar 1

Um método mais fácil de obter a negação de um número em complemento de dois é o que se segue:

	Exemplo 1	Exemplo 2
1. A partir da direita, encontre o primeiro '1'	0101001	0101100
2. Inverte todos os bits à esquerda deste '1'	1010111	1010100

Fonte: https://pt.wikipedia.org/wiki/Representação_de_números_com_sinal#Complemento_para_dois

Operações em Complemento de 2

- Convertendo um número de n bits em números com mais de n bits:
  - Um dado imediato de 16 bits do MIPS é convertido para 32 bits
  - Copiar o bit mais significativo (o bit de sinal) para outros bits

    0010 -> 0000 0010

    1010 -> 1111 1010

  - Operação conhecida como "extensão de sinal (sign extension) "
- Exercício: Converter os seguintes números de 4 bits em 8 bits:
  - 5
  - -6

Adição & Subtração

- Como no ensino fundamental: vai-um/ vem/um ()

```
    0111          0111          0110
  + 0110        - 0110        - 0101
```

- Operações em complemento de 2
  - subtração usando soma de números negativos

```
    0111
  + 1010
```

- Overflow (resultado muito grande para ser armazenado em x bits):
  - P.ex., somando dois números de n-bits não resulta em n-bits

```
    0111
  + 0001
   10000
```

Detectando Overflow

- Teste Eficiente p/ detectar o overflow: Se o "vai-um"que chega no bit de sinal for igual ao "vai-um" que sai do bit de sinal, não ocorreu overflow. Se forem diferentes. overflow!

```
vai 1
          1 1   1 1 1
            0 1 1 0 1 0 1 1    (126)
          + 0 0 1 0 1 1 1 0    (  8)
vai 0       ---------------
            1 0 0 1 1 0 0 1    (-122) ???  overflow
```

•Exercício: Utilize as regras acima para detectar (ou não) o overflow nas seguintes operações usando inteiros de 4 bits:

- 2 + 3
- 7 + 5
- -3 - 1
- -4 - 5

Efeitos do Overflow

- Dispara uma exceção (interrupt)
  - A instrução salta para endereço predefinido para a rotina de exceção
  - O endereço da instrução interrompida é salvo para possível retorno

- Nem sempre se requer a detecção do overflow

Números sem sinal são normalmente usados para calcular endereços de acesso à memória, e neste caso o overflow não é considerado um problema, pois os números em questão são limitados pelo tamanho da memória endereçável.

Obs: Ling. C não especifica a detecção de overflow.

Números

- **Representação hexadecimal:** *Forma de representação numérica utilizando 16 dígitos:*
  - **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F**
- *Conveniente para a representação de grupos de bits (bit-patterns).*
  - ***4 bits = 1 dígito hexadecimal***
- *Computadores são normalmente organizados utilizando uma quantidade de bits que é um múltiplo de 4 → Notação hexadecimal é conveniente para representar endereços de memória, instruções, etc.*

Números

Bit pattern	Hexadecimal representation	
0000	0	
0001	1	
0010	2	
0011	3	
0100	4	
0101	5	
0110	6	
0111	7	
1000	8	
1001	9	
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Números

- Exemplo: Calcule a representação hexadecimal da seguinte sequencia de bits:

1010010011001000

Números

- Exemplo: Calcule a representação hexadecimal da seguinte sequencia de bits:

$$1010 \quad 0100 \quad 1100 \quad 1000 \quad = \text{A4C8}$$

A 4 C 8

Revisão sobre Álgebra Booleana e Portas

- Exercício: Considerar uma função lógica com 3 entradas: A, B, e C.

  A saída D é "true" se pelo menos uma entrada é "true"

  A saída E é "true" se exatamente duas entradas são "true"

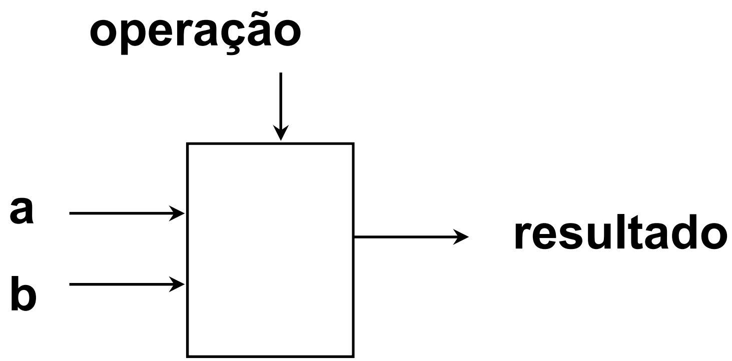
  A saída F é "true" somente se todas as três entradas são "true"

- Mostrar a tabela verdade para essas três funções.
- Mostrar as equações Booleanas para as três funções.
- Mostrar uma implementação consistindo de portas inversoras, AND, e OR.

ULA

Construindo uma ULA de 1 bit

- Considere uma ULA para suportar apenas as instruções **AND** e **OR**

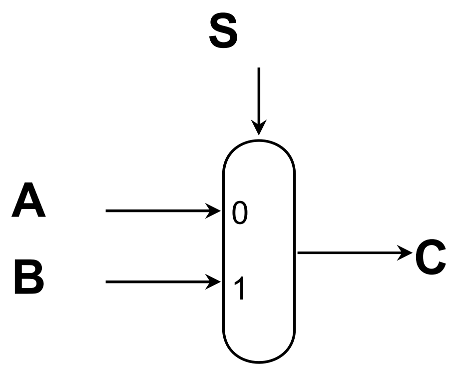


op	a	b	res

- Possível implementação: ?

Construindo uma ULA de 1 bit

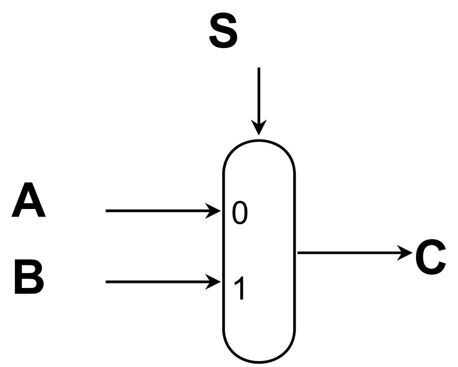
- Multiplexador (MUX): Seleciona uma das entradas para a saída, baseado numa entrada de controle



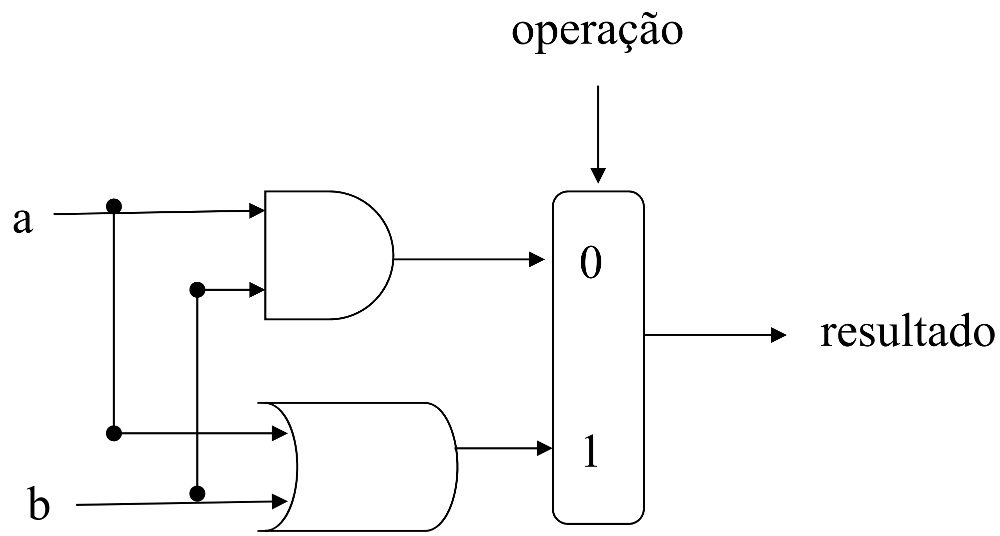
- Nossa ULA pode ser construída usando um MUX

Construindo uma ULA de 1 bit

- Considere uma ULA para suportar apenas as instruções **AND** e **OR**

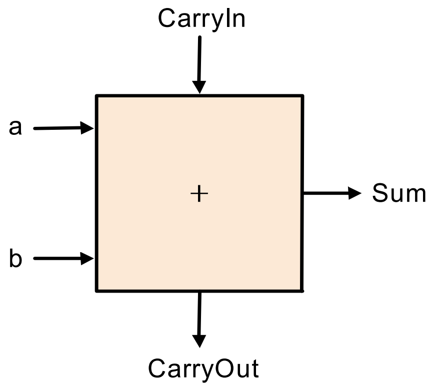


- Possível implementação:



Construindo uma ULA de 1 bit

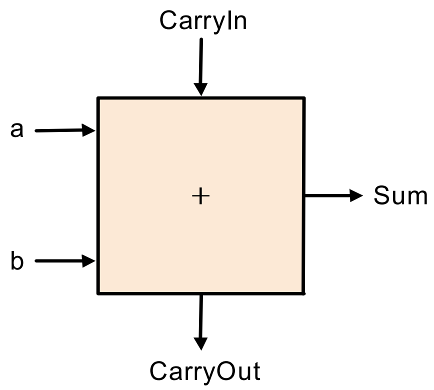
- Incluir recursos para executar a operação de Adição na nossa ULA:
- Considere um circuito somador de 1-bit:



- Exercício: Construir a tabela verdade e o circuito correspondente para o somador de 1 bit definido acima.
  - Obs: 3 entradas (a, b, CarryIn)
  - Obs: 2 saídas (CarrayOut, Sum)

Construindo uma ULA de 1 bit

- Incluir recursos para executar a operação de Adição na nossa ULA:
- Seja a ULA de 1-bit para soma:



Expressões p/ CarryOut e Sum:

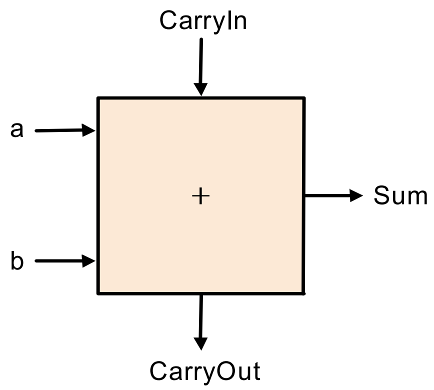
`CarryOut = a b + a xor b CarryIn`

`Sum = a xor b xor CarryIn`

- Exercício: Verificar que a expressão acima corresponde à tabela verdade para o somador de 1 bit. Construir o circuito correspondente.

Construindo uma ULA de 1 bit

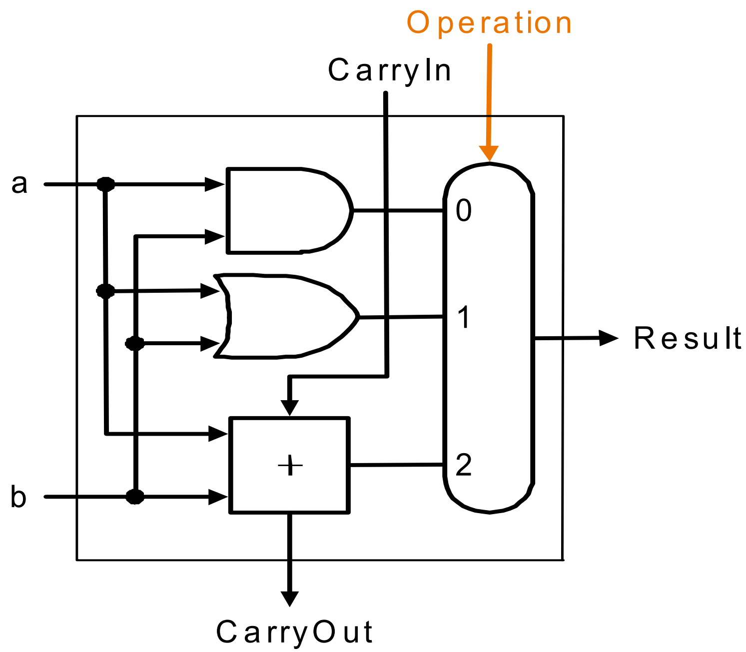
- Incluir recursos para executar a operação de Adição na nossa ULA:
- Temos o somador de 1 bit:



- Agora, podemos acoplar o somador de 1 bit à ULA p/ AND e OR anteriormente projetada:

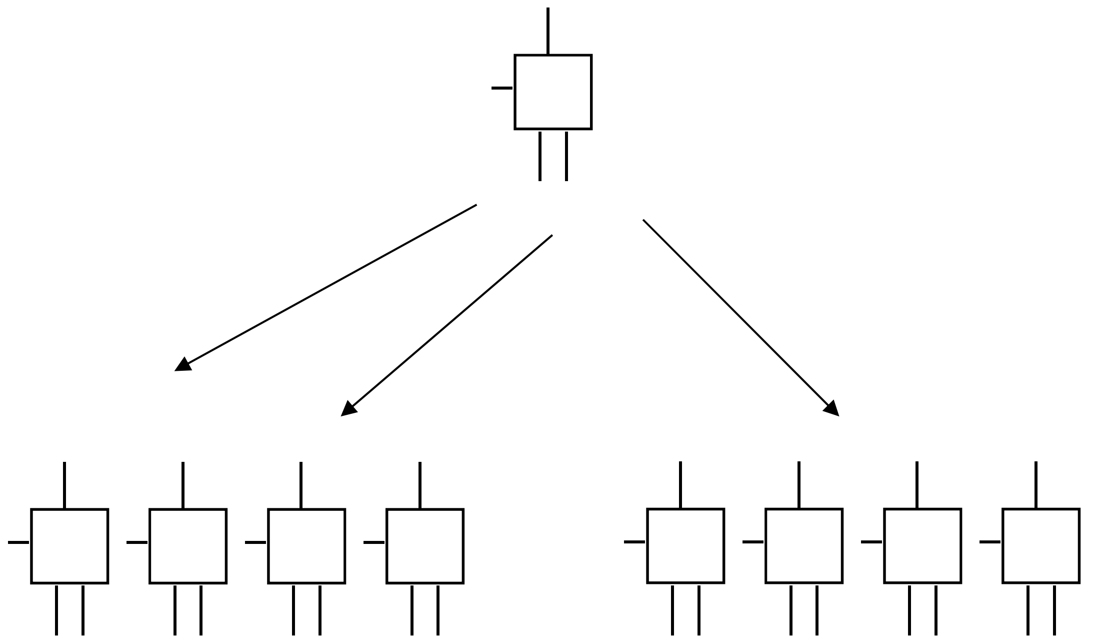
Construindo uma ULA de 1 bit

ULA para efetuar ADD, AND e OR com duas entradas de 1 bit:



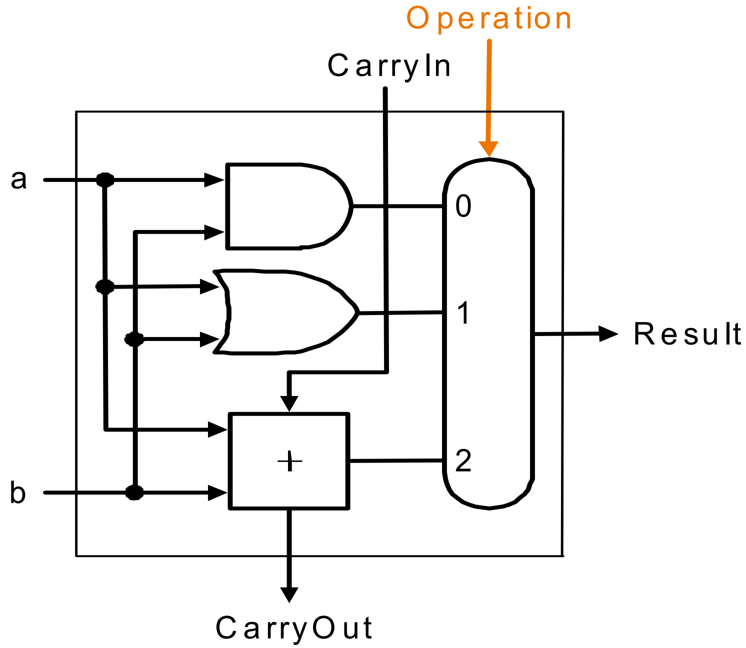
Construindo uma ULA de 32 bits

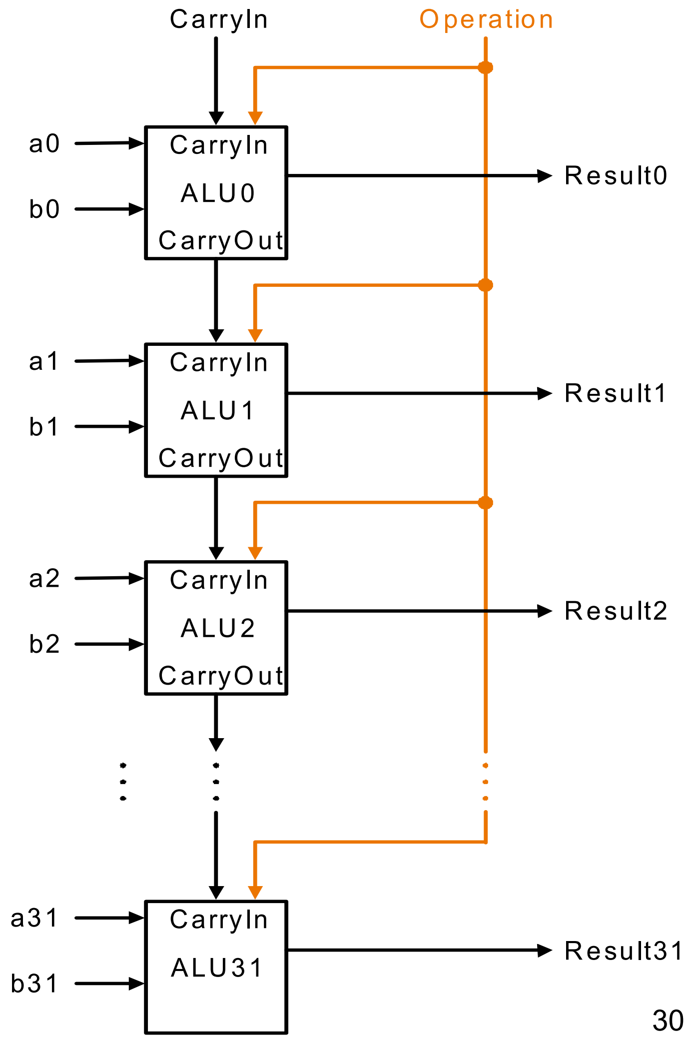
Projetar uma ULA de 1 bit, e replicá-la 32 vezes



31 30 .... 2 1 0

Construindo uma ULA de 32 bits





Obs: este esquema é conhecido como somador *ripple carry* (vai-um-propagado)

Incluindo a subtração (a-b) na ULA

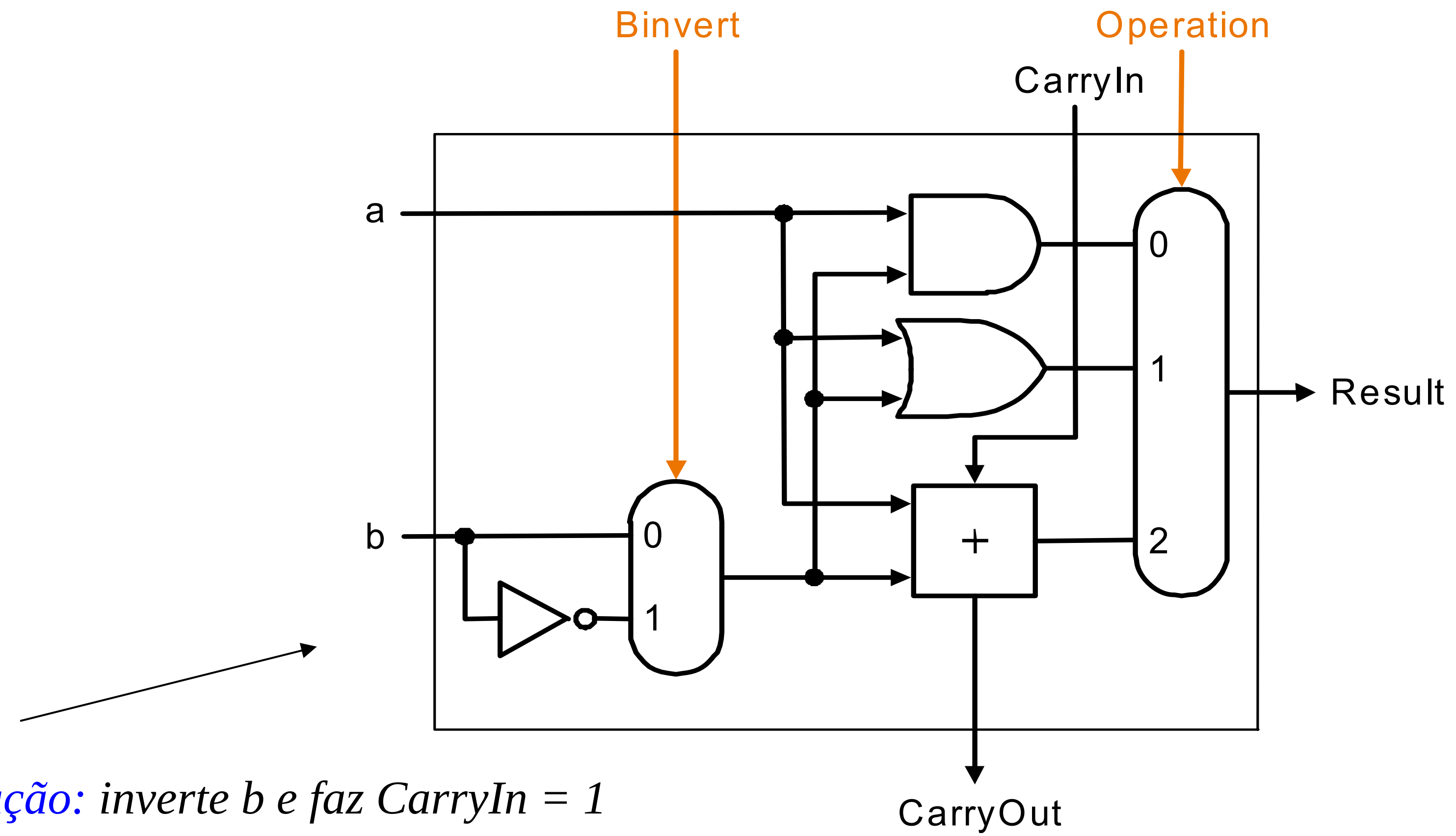
- Para subtrair (a-b) na ULA construída anteriormente, usar a técnica do complemento de 2: apenas negar b e somar.

- Como negar um dos operandos de entrada, de maneira eficiente ?

E sobre a subtração (a – b) ?

- Como negar um dos operandos de entrada, de maneira eficiente ?

  Solução:



subtração: inverte b e faz CarryIn = 1

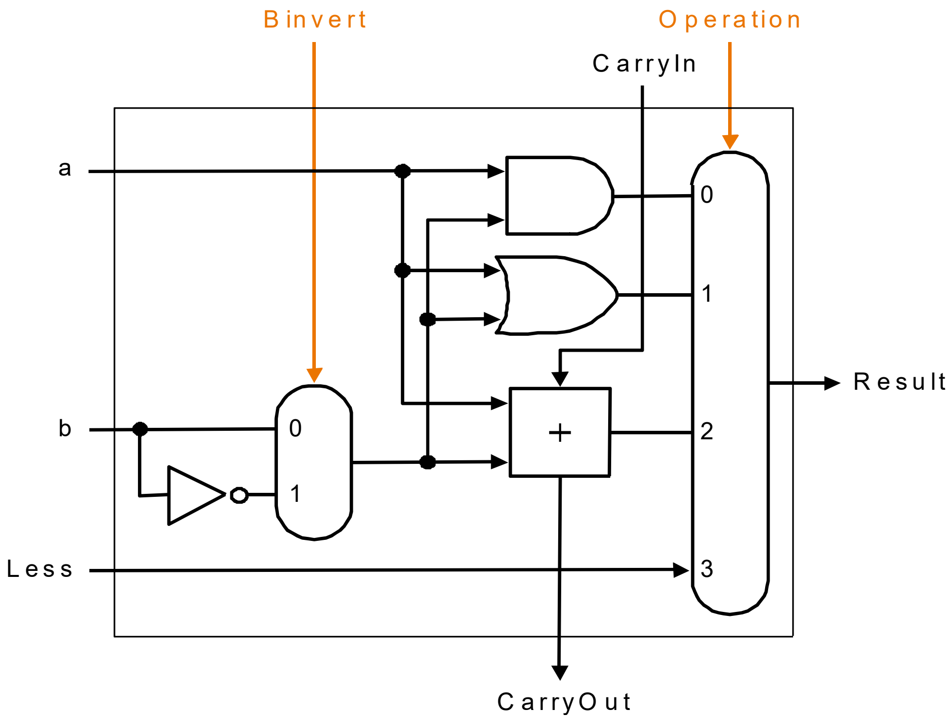
Instrução SLT

- MIPS deve suportar a instrução set-on-less-than (SLT)
  - lembrar: SLT é uma instrução aritmética, do ponto de vista do hardware
  - produz um 1 se rs < rt, e 0 caso contrário
  - implementar usando subtração:
    - Se (a - b) < 0, significa que a < b
  - Como ?

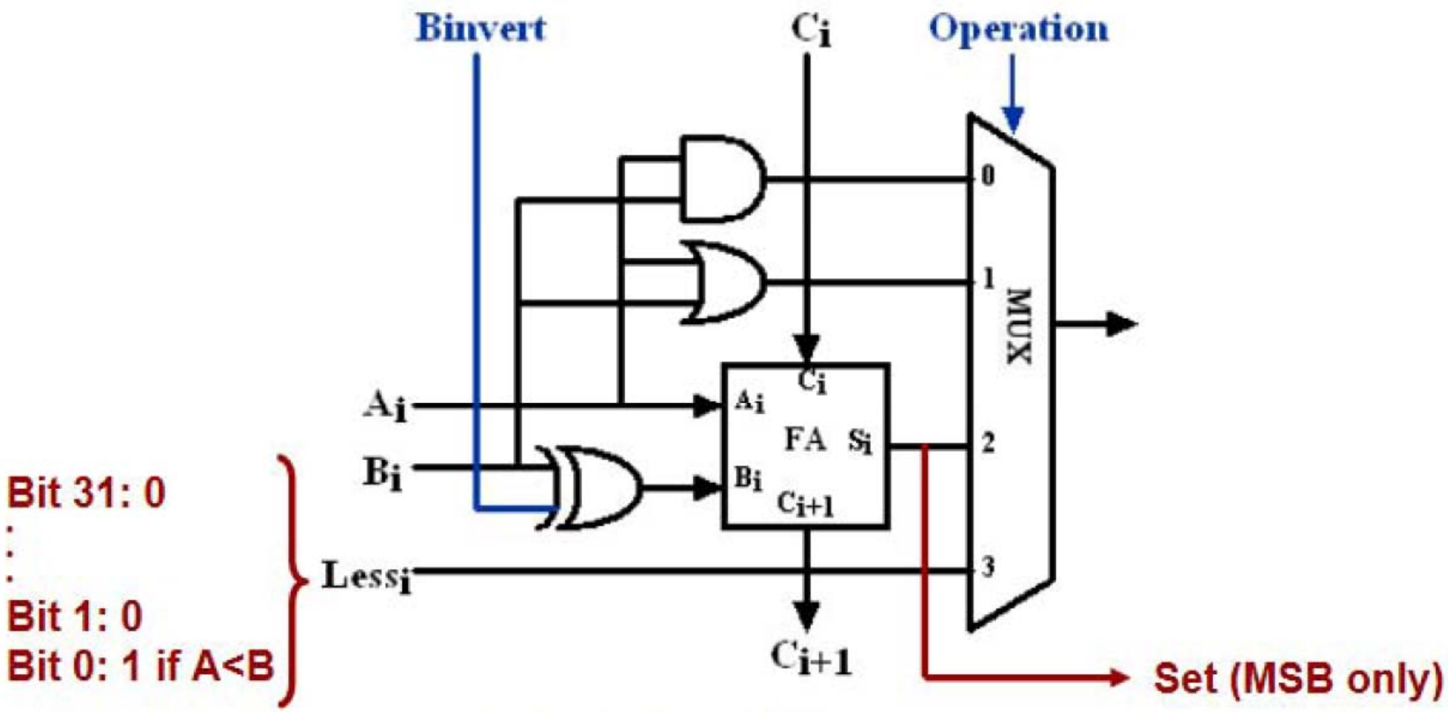
Instrução SLT

- MIPS deve suportar a instrução set-on-less-than (SLT)
  - lembrar: SLT é uma instrução aritmética, do ponto de vista do hardware
  - produz um 1 se rs < rt, e 0 caso contrário
  - implementar usando subtração:
    - Se (a - b) < 0, significa que a < b
  - Como ?
  - rs < rt significa que o resultado é negativo, logo o bit de sinal = 1
  - set inicializa o registrador que armazena o resultado c/ ...0000001
  - **Logo, basta conectar o bit de sinal do resultado da subtração ao último bit do registrador de destino.**
    - Complicação: casos em que ocorre overflow.

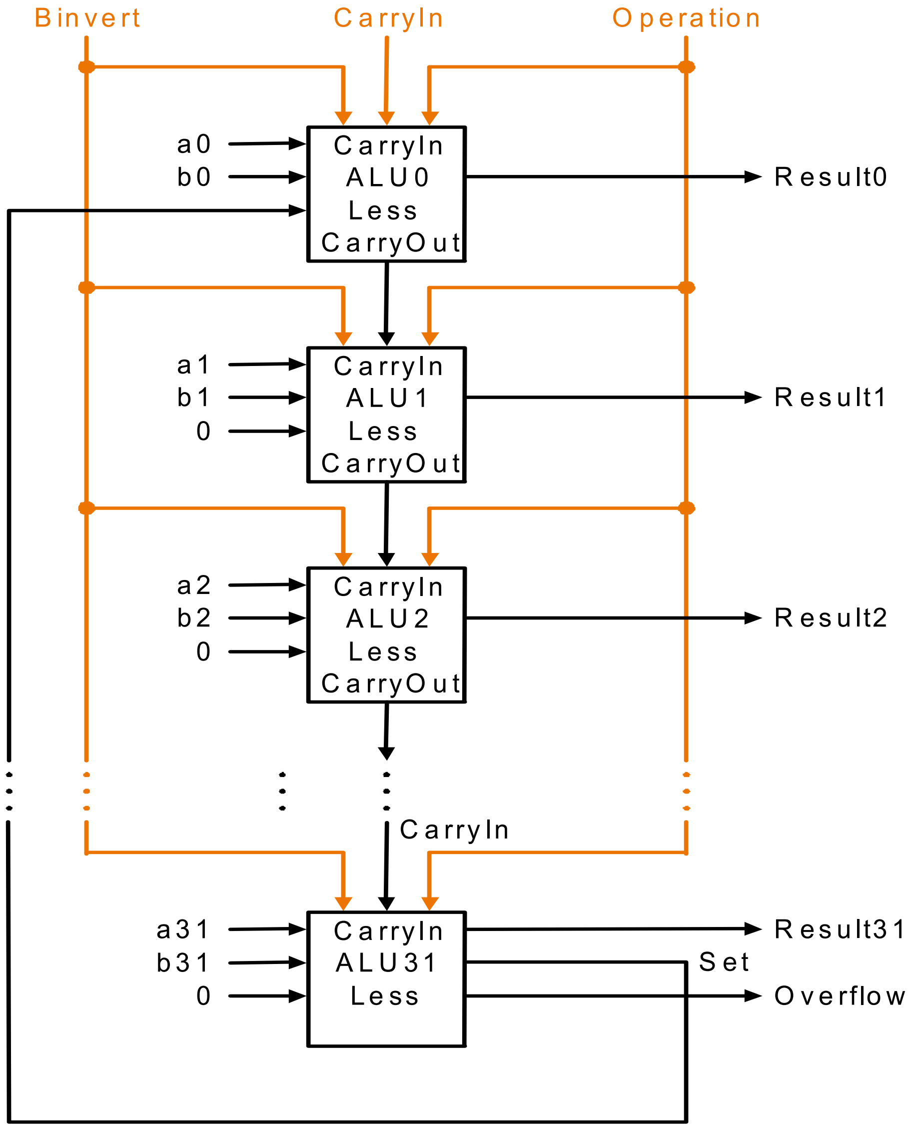
Instrução SLT



Instrução SLT



Instrução SLT p/ 32 bits



Instrução BEQ

- MIPS deve suportar o teste para implementar a instrução de desvio "beq"
  - Ex: beq $t5, $t6, $t7 - Se $t6 = $t7, desviar para endereço $t5

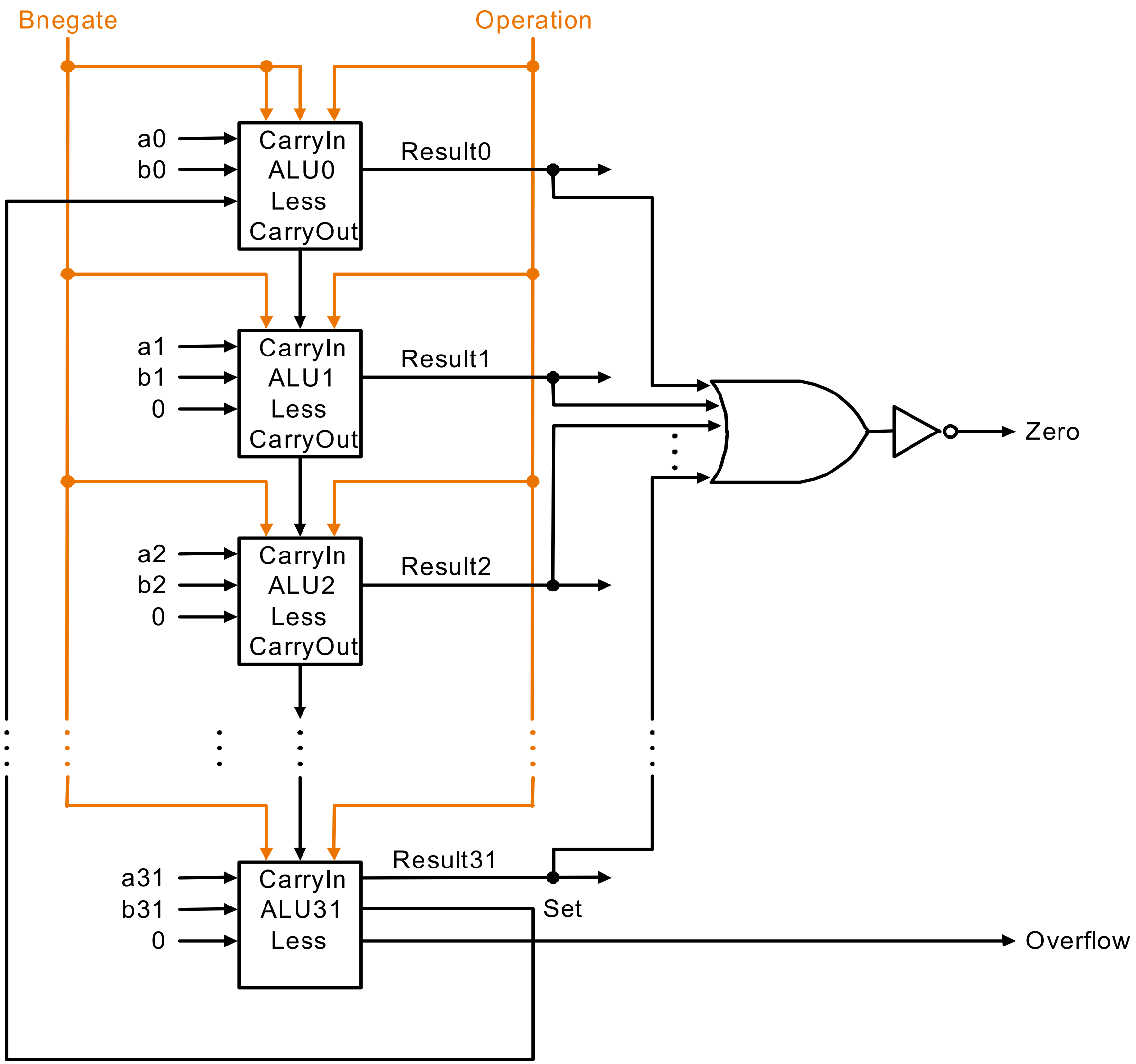
- implementar usando subtração: (a - b) = 0 implica a = b
- Como ?

Instrução BEQ

- MIPS deve suportar o teste para igualdade (beq \$t5, \$t6, \$t7)
- Implementar usando subtração: (a - b) = 0 implica a = b
- Como ?

- Adicionar hardware para testar se o resultado da subtração é igual a zero.
- Efetuar uma operação OR entre todos os bits da saída.
- Se resultado for igual a zero, significa que a=b → Enviar o sinal 1 (beq=true), simplesmente invertendo a saída do OR.

Instrução BEQ para 32 bits



Conclusão

- Podemos construir uma ULA para suportar o conjunto de instruções MIPS:
  - Usando multiplexador para selecionar a saída desejada
  - Realizando uma subtração usando o complemento de 2
  - Replicando uma ULA de 1-bit para produzir uma ULA de 32-bits

- Pontos importantes sobre o hardware
  - Todas as portas estão sempre trabalhando
  - A velocidade de uma porta é afetada pelo número de entradas da porta
  - A velocidade de um circuito é afetado pelo número de portas em série (no caminho crítico do nível mais profundo da lógica)
  - Mudanças inteligentes na organização podem melhorar o desempenho (similar a usar algoritmos melhores em software)