Parte III: Aritmética Computacional

Introdução

Introdução

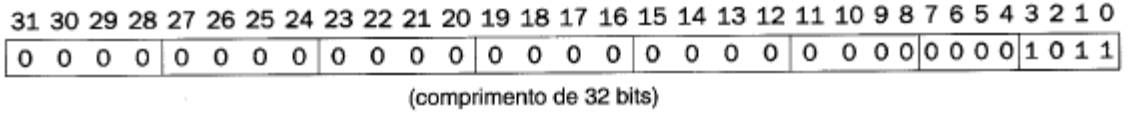
- Como representar números em memória?
  - Em computadores, tudo é binário

- Como representar números negativos e de ponto flutuante?

- Como o computador processa esses dados para realizar cálculos?

Conversão Binário-Decimal

- Humanos são acostumados a representar números em base 10 (decimais)
- Convertendo de binário para decimal:



1011_{dois}

representa

$$(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{dez}$$
$$= (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{dez}$$
$$= 8 + 0 + 2 + 1_{dez}$$
$$= 11_{dez}$$

- Com 32 bits: **0 a $2^{32} - 1$ ($4.294.967.295_{dez}$)**

Bits menos significativos →

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{dois} = 0_{dez}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{dois} = 1_{dez}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{dois} = 2_{dez}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{dois} = 4,294,967,293_{dez}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{dois} = 4,294,967,294_{dez}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{dois} = 4,294,967,295_{dez}$

Representando Caracteres

- Computadores também precisam representar caracteres, pois é comum manipular textos (strings)
  - Código ASCII (Americ. Standard Code for Inform Interchamge)
  - 1 byte é suficiente para representar um caracter em ASCII

Valor em ASCII	Caractere	Valor em ASCII	Caractere	Valor em ASCII	Caractere	Valor em ASCII	Caractere	Valor em ASCII	Caractere	Valor em ASCII	Caractere
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

Figura 3.15 Representação de caracteres em ASCII. Note que os mesmos caracteres, quando expressos em maiúsculas e em minúsculas, diferem exatamente de 32 unidades; esta observação pode cortar caminho na verificação e na troca da caixa de caracteres em geral. Os caracteres de formatação não estão nesta tabela. Por exemplo, 9 representa o caractere tab e 13 o retorno de carro. Outros caracteres úteis são o 8, que representa o backspace, e o 0 para o null, valor este que é usado nos programas em C para marcar o fim de um string.

Representando Números

- Primeiros computadores representavam números através de caracteres expressando valores decimais

- Caracteres não são eficientes para números
  - Ex. Representando o número 1 bilhão
  - Em ASCII: 1000000000
    - 10 dígitos de 8 bits = 80 bits
  - Em binário:
    - Bastam 32 bits, como é a palavra básica do MIPS
  - 2,5 vezes mais bits no ASCII
  - O maior problema: Realizar operações com caracteres!!!
    - Como somar 1 + 2??

Representando Números

- Números possuem sinal e magnitude

- Sinal pode ser representado por 1 bit do número
  - bit de sinal é normalmente o mais significativo
  - Ex. -1 (dec) = 10000000000000000000000000000001
  - Alguns inconvenientes:
    - HW de cálculo, como soma, deve considerar circuito para manipular sinal após a soma
    - Há representação de +0 e -0 (desperdício)

- Representação em Complemento de 2
  - 0s à esquerda são números positivos
  - 1s à esquerda são números negativos
  - Simplifica HW de aritmética

Representando Números

- Em complemento de 2
  - Obs. Primeiro bit ainda mostra o sinal

$$
\begin{aligned}
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{dois} = 0_{dez}\\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{dois} = 1_{dez}\\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{dois} = 2_{dez}\\
&\ldots \qquad\qquad \ldots\\
&0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{dois} = 2{,}147{,}483{,}645_{dez}\\
&0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{dois} = 2{,}147{,}483{,}646_{dez}\\
&0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{dois} = 2{,}147{,}483{,}647_{dez}\\
&1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{dois} = -2{,}147{,}483{,}648_{dez}\\
&1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{dois} = -2{,}147{,}483{,}647_{dez}\\
&1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{dois} = -2{,}147{,}483{,}646_{dez}\\
&\ldots \qquad\qquad \ldots\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{dois} = -3_{dez}\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{dois} = -2_{dez}\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{dois} = -1_{dez}
\end{aligned}
$$

- Conversão

$$(x31\times -2^{31}) + (x30\times 2^{30}) + (x29\times 2^{29}) + \ldots + (x1\times 2^{1}) + (x0\times 2^{0})$$

- Exemplo

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{dois} \longrightarrow$$

$$
\begin{aligned}
&(1\times -2^{31}) + (1\times 2^{30}) + (1\times 2^{29}) + \ldots + (1\times 2^{2}) + (0\times 2^{1}) + (0\times 2^{0})\\
&= -2^{31} + 2^{30} + 2^{29} + \ldots + 2^{2} + 0 + 0\\
&= -2{,}147{,}483{,}648_{dez} + 2{,}147{,}483{,}644_{dez}\\
&= -4_{dez}
\end{aligned}
$$

Representando Números

- Complemento de 2: Regra prática de conversão
  - Se o número é positivo, preceder normalmente
  - Se o número é negativo:
    - Faça a negação (negativo → positivo, positivo → negativo)
    - Converta para decimal normalmente
    - Acrescente o sinal
  - Regra prática de negação:
    - Troque todos os bits em 0 para 1, e os em 1 para 0.
    - Some 1 ao resultado
  - Exemplo
    - Em binário: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{dois}$
    - É negativo, então negue o número

$$\begin{array}{rl} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{dois} \\ + & 1_{dois} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{dois} \\ = & 2_{dez} \end{array}$$

    - Resultado: -2 (dec)

Extensão de Sinal

- Muitas vezes temos que manipular números representados em quantidades de bits diferentes
  - Ex. Colocar um valor de 16 bits em um registrador de 32 bits
  - Procedimento: Estender o último bit (mais significativo para todas as posições à esquerda
  - Exemplo: Convertendo -2 e 2, ambos em 16 bits, para 32 bits
    - 2:

$0000\ 0000\ 0000\ 0010_{dois} = 2_{dez}$ ⟶ $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{dois} = 2_{dez}$

    - -2:

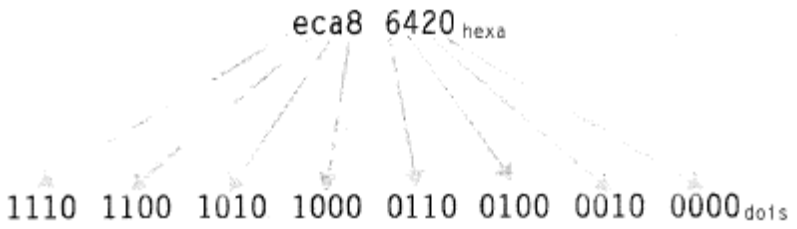
$1111\ 1111\ 1111\ 1110_{dois}$ ⟶ $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{dois} = -2_{dez}$

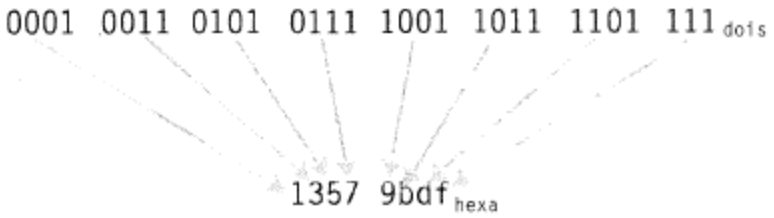
Representação em hexadecimal

- Melhora legibilidade de binários longos
- Organiza bits em grupos de 4

Hexadecimal	Binário	Hexadecimal	Binário	Hexadecimal	Binário	Hexadecimal	Binário
0_{hexa}	0000_{dois}	4_{hexa}	0100_{dois}	8_{hexa}	1000_{dois}	c_{hexa}	1100_{dois}
1_{hexa}	0001_{dois}	5_{hexa}	0101_{dois}	9_{hexa}	1001_{dois}	d_{hexa}	1101_{dois}
2_{hexa}	0010_{dois}	6_{hexa}	0110_{dois}	a_{hexa}	1010_{dois}	e_{hexa}	1110_{dois}
3_{hexa}	0011_{dois}	7_{hexa}	0111_{dois}	b_{hexa}	1011_{dois}	f_{hexa}	1111_{dois}

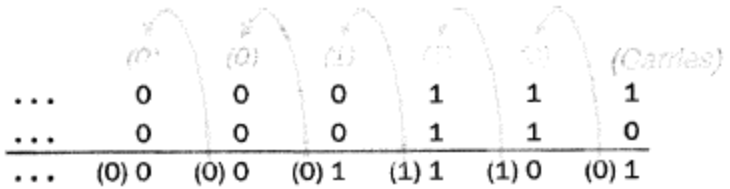
- Exemplos





Soma e Subtração

- Soma bit-a-bit, da direita para a esquerda, passando carries (vai-um) ao bit mais à esquerda
- Subtração: Soma com o subtraendo negado (compl de 2)



- Ex: 7+6

$$\begin{array}{rl} & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0111_{dois} = 7_{dez} \\ + & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0110_{dois} = 6_{dez} \\ \hline = & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,1101_{dois} = 13_{dez} \end{array}$$

- Ex. 7-6

$$\begin{array}{rl} & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0111_{dois} = 7_{dez} \\ + & 1111\,1111\,1111\,1111\,1111\,1111\,1111\,1010_{dois} = -6_{dez} \\ \hline = & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0001_{dois} = 1_{dez} \end{array}$$

Overflow

- Somas e subtrações podem gerar overflow
  - Acontece quando o resultado da operação não cabe no tamanho usado na representação de números pela máquina.
    - Ex. 32 bits no MIPs
  - Ex. 2 000 000 000 + 2 000 000 000 = 4 000 000 000
    - Não é possível representar com sinal em 32 bits
  - Como detectar:
    - Se houver vai-um ou empresta-1 no último bit (bit de sinal)
    - Ex. 2 000 000 000 + 2 000 000 000

```
 01110111001101011001010000000000
+01110111001101011001010000000000
=11101110011010110010100000000000
```

Operações Lógicas

- Shift: Desloca bits à esquerda ou direita
  - Ex. Deslocar oito bits à esquerda

$0000\ 0000\ 0000\ 00000\ 000\ 0000\ 0000\ 0000\ 1101_{dois} \longrightarrow 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 0000\ 0000_{dois}$

- AND (bit-a-bit): Resultado 1 se bits são iguais
  - Ex. $0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 0000\ 0000_{dois}$
    and $0000\ 0000\ 0000\ 0000\ 0011\ 1100\ 0000\ 0000_{dois}$
    = $0000\ 0000\ 0000\ 0000\ 0000\ 1100\ 0000\ 0000_{dois}$

- OR: (bit-a-bit): Resulta 1 se há 1 bit 1 no operando
  Ex. $0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 0000\ 0000_{dois}$
  or $0000\ 0000\ 0000\ 0000\ 0011\ 1100\ 0000\ 0000_{dois}$
  = $0000\ 0000\ 0000\ 0000\ 0011\ 1101\ 0000\ 0000_{dois}$

Construção de uma ULA

- ULA = Unidade Lógica e Aritmética
  - Um componente fundamental dos processadores

- Blocos Básicos na construção ULAs

1. Porta AND ($c = a \cdot b$)

a	b	c = a · b
0	0	0
0	1	0
1	0	0
1	1	1

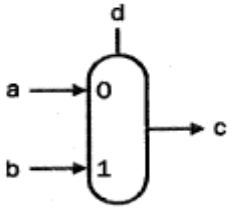
2. Porta OR ($c = a + b$)

a	b	c = a + b
0	0	0
0	1	1
1	0	1
1	1	1

3. Inversor ($c = \bar{a}$)

a	c = ā
0	1
1	0

4. Multiplexador
(se d == 0, c = a;
senão, c = b)



d	c
0	a
1	b

Construção de uma ULA

- Operações lógicas AND e OR são simples e mapeadas diretamente para as portas lógicas respectivas

- Unidade de 1 bit, onde uma entrada no multiplexador define a operação (linha de controle)

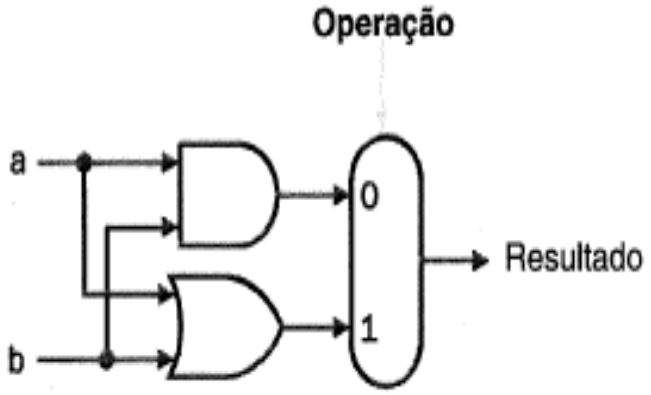
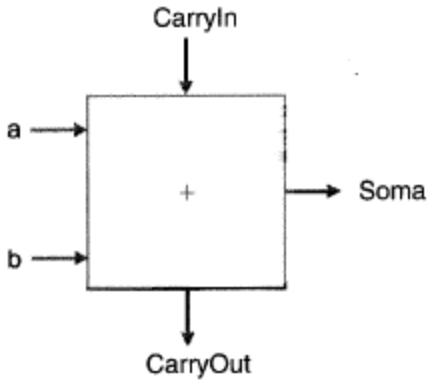


Figura 4.9 Unidade lógica de 1 bit para executar as operações AND e OR.

Construção da ULA

- Adição
  - A soma em 1 bit deve considerar Carry-out (vai-um) e Carry-in (vem um)
  - Especificação de entradas e saídas do somador de 1 bit (Tab. Verdade)



Entradas			Saídas		Comentários
a	b	CarryIn	CarryOut	Soma	
0	0	0	0	0	$0 + 0 + 0 = 00_{dois}$
0	0	1	0	1	$0 + 0 + 1 = 01_{dois}$
0	1	0	0	1	$0 + 1 + 0 = 01_{dois}$
0	1	1	1	0	$0 + 1 + 1 = 10_{dois}$
1	0	0	0	1	$1 + 0 + 0 = 01_{dois}$
1	0	1	1	0	$1 + 0 + 1 = 10_{dois}$
1	1	0	1	0	$1 + 1 + 0 = 10_{dois}$
1	1	1	1	1	$1 + 1 + 1 = 11_{dois}$

Figura 4.11 Especificação das entradas e saídas de um somador de 1 bit.

  - Lógica para Carry-out

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b) + (a \cdot b \cdot \text{CarryIn}) \longrightarrow \text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

  - Lógica para Soma

$$\text{Soma} = (a \cdot \bar{b} \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

Construção da ULA

- Adição
  - Circuito Carry-out

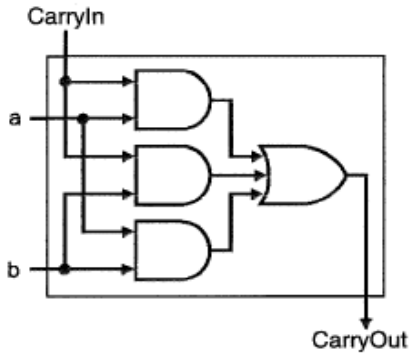
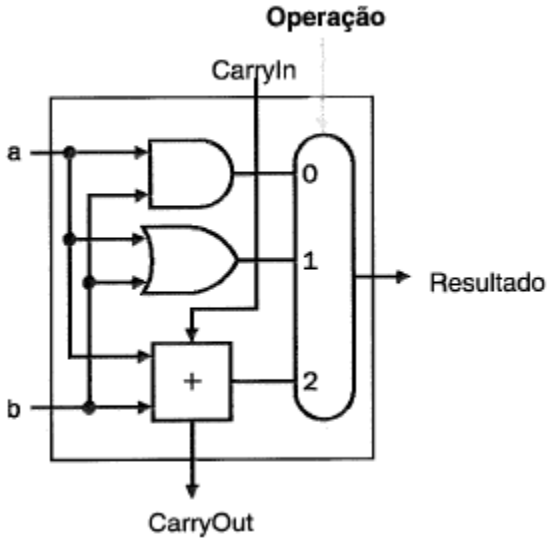


Figura 4.13 Hardware do somador para geração do sinal CarryOut. O restante do hardware do somador é composto pelos circuitos para geração da saída Soma, especificada pelas equações dadas anteriormente.

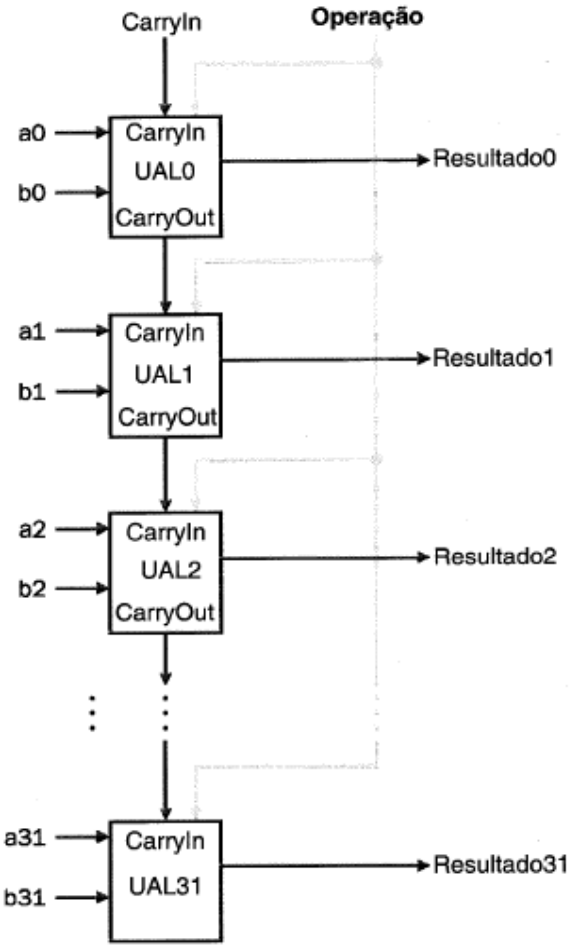
  - Circuito Soma ?

Construção da ULA

- Adição em 32 bits
  - Basta interligar 32 unidades de 1 bit

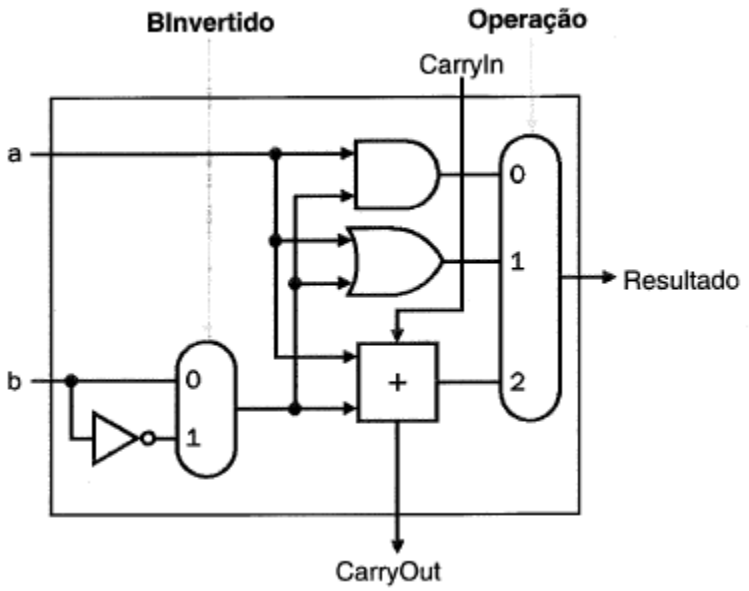


Combinação dos circuitos
(linha Operação com 2 bits)



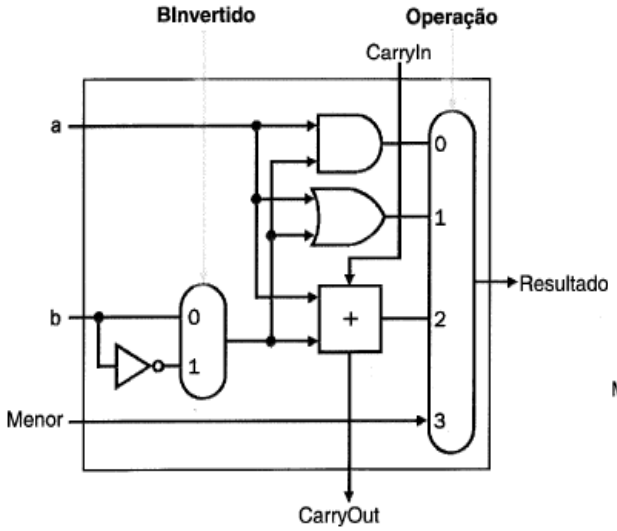
Construção da ULA

- Subtração de A - B
  - Basta usar o complemento de 2 de B
    - Determinado por Binvertido
    - inverte operando bit-a-bit
    - Soma 1
  - Para somar 1, basta colocar carry-in = 1 no bit menos significativo do somador
    - Pode ser a mesma ligação do Binvertido

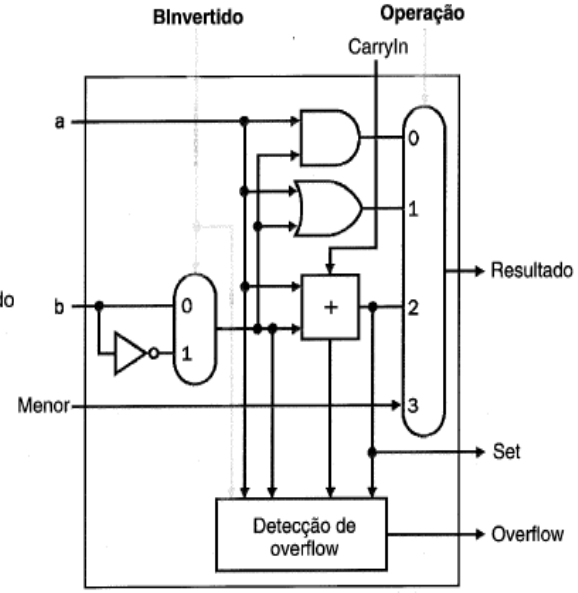


Construção da ULA

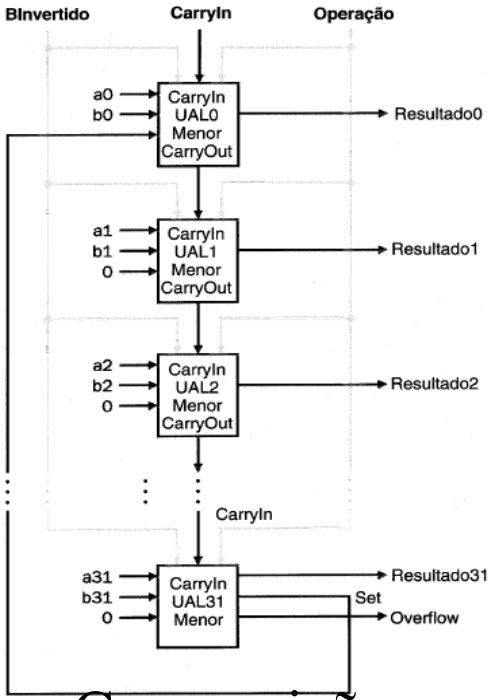
- Instrução stl (set on less than) do MIPS
  - Resultado 1 se A < B
  - Quando executada coloca 0 em todos os bits, exceto no menos significativo, que depende da comparação
  - Para a comparação basta verificar se A-B < 0, que o caso que causa bit menos significativo como 1. Caso A-B >= 0, o resultado é positivo
  - Então basta olhar o bit de sinal gerado da subtração (bit mais signif.)
  - Bit mais significativo também serve para verificar overflow



Bits 0~30



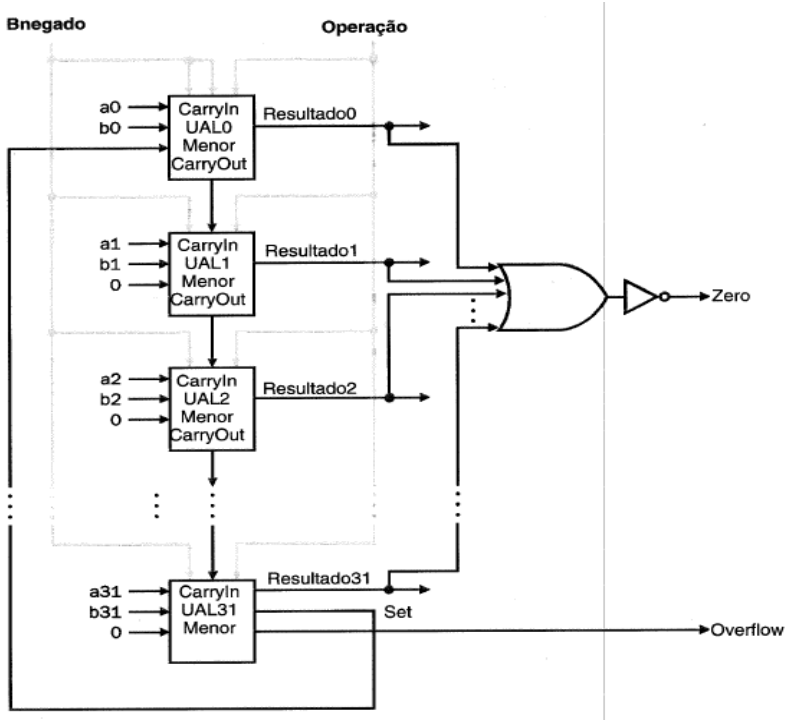
Bit 31 (mais signif.)



Composição

Construção da ULA

- Suporte a branches (saltos)
  - Sempre se dá comparando a igualdade de dois valores
  - Se executarmos A-B, com A=B, o resultado será 0.
  - Mais simples: o OR de todos os bit tem que dar zero, bastando negar esse resultado para determinar o salto
  - Então basta acrescentar circuito para essa verificação



Construção da ULA

- Controle para as operações:
  - Formado por Bnegado (interligado ao Carry-in) e as 2 linhas de Operação

Linhas de controle da unidade aritmética lógica	Função
000	and
001	or
010	soma
110	subtração
111	set on less than

Figura 4.20 Valores das três linhas de controle da UAL e as operações correspondentes na UAL.

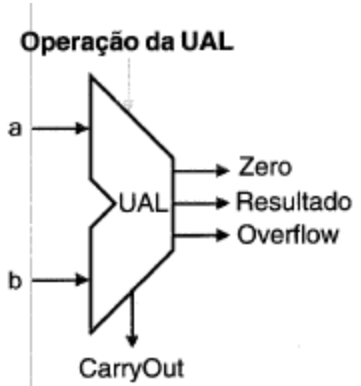


Figura 4.21 Símbolo comumente usado em diagramas lógicos para representar a UAL da Figura 4.19. Este símbolo também pode ser usado para representar um somador, de maneira que ele deve ser identificado explicitamente como sendo uma UAL ou um Somador.

Aritmética no MIPS

Categoria	Instrução	Exemplo	Significado	Comentário
Aritmética	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Três operandos; detecção de overflow
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Três operandos; detecção de overflow
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + $s3`	+ constante: detecção de overflow
	add unsigned	`addu $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Três operandos: sem detecção de overflow
	subtract unsigned	`subu $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Três operandos: sem detecção de overflow
	add immediate unsigned	`addiu $s1, $s2, 100`	`$s1 = $s2 + $s3`	+ constante: sem detecção de overflow
	move from coprocessor register	`mfc0 $s1, $epc`	`$s1 = $epc`	Usado para fazer cópia do EPC (Exception PC) e de outros registradores especiais
	multiply	`mult $s2, $s3`	**Hi, Lo = `$s2` × `$s3`**	**Produto de 64 bits com sinal em Hi e Lo**
	multiply unsigned	`multu $s2, $s3`	**Hi, Lo = `$s2` × `$s3`**	**Produto de 64 bits sem sinal em Hi e Lo**
	divide	`div $s2, $s3`	**Lo = `$s2 / $s3`, Hi = `$s2` mod `$s3`**	**Lo = quociente; Hi = Resto**
	divide unsigned	`divu $s2, $s3`	**Lo = `$s2 / $s3`, Hi = `$s2` mod `$s3`**	**Quociente e Resto sem sinal**
	move from Hi	`mfhi $s1`	**`$s1` = Hi**	**Usado para obter cópia de Hi**
	move from Lo	`mflo $s1`	**`$s1` = Lo**	**Usado para obter cópia de Lo**
Lógica	and	`and $s1, $s2, $s3`	`$s1 = $s2 & $s3`	Operandos em três registradores: AND lógico
	or	`or $s1, $s2, $s3`	`$s1 = $s2 | $s3`	Operandos em três registradores: OR lógico
	and immediate	`andi $s1, $s2, 100`	`$s1 = $s2 & 100`	AND lógico do conteúdo de um registrador com uma constante
	or immediate	`ori $s1, $s2, 100`	`$s1 = $s2 | 100`	OR lógico do conteúdo de um registrador com uma constante
	shift left logical	`sll $s1, $s2, 10`	`$s1 = $s2 << 10`	Deslocamento à esquerda (número de bits deslocados armazenado em uma constante)
	shift right logical	`srl $s1, $s2, 10`	`$s1 = $s2 >> 10`	Deslocamento à direita (número de bits deslocados armazenado em uma constante)