UFRR

UNIVERSIDADE FEDERAL DE RORAIMA

Aritmética Computacional

Prof. Herbert Oliveira Rocha

UNIVERSIDADE FEDERAL DE RORAIMA

Aritmética Computacional

Baseado nas aulas do Prof. Dr. Mauricio Figueiredo - UFAM

Prof. Herbert Oliveira Rocha

Introdução

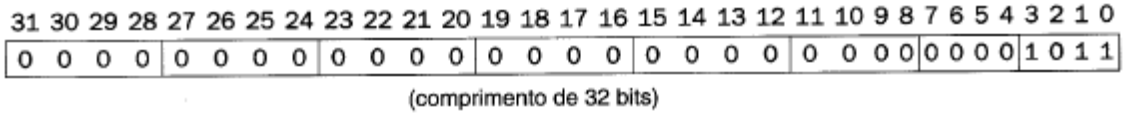
- Como representar números em memória?
  - Em computadores, tudo é binário

- 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:

$\hat{0}$ a $2^{32} - 1$ ($\hat{4}.294.967.295_{dez}$)

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

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

- 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

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

    - 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:

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

    - -2:

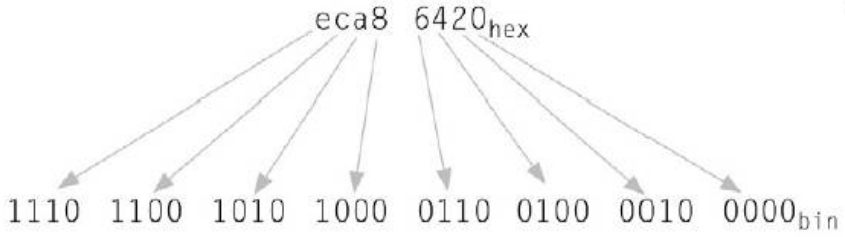
$1111\ 1111\ 1111\ 1110_{dois} \longrightarrow 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{dois} = -2_{dez}$

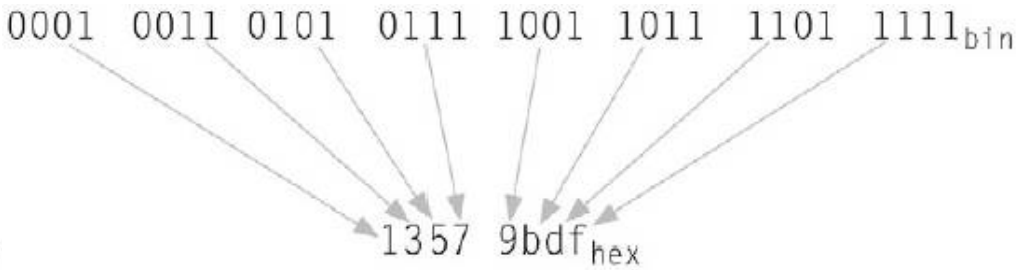
Representação em hexadecimal

- 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}

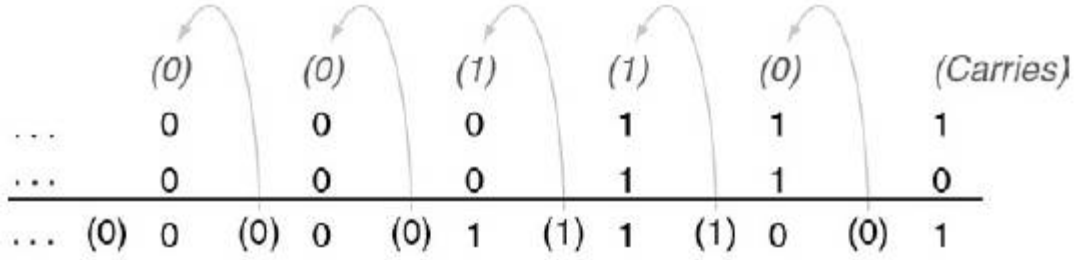
- Exemplos





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

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

- AND (bit-a-bit): Resultado 1 se bits são iguais
  - Ex.

and

=

$0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 0000\ 0000_{dois}$

$0000\ 0000\ 0000\ 0000\ 0011\ 1100\ 0000\ 0000_{dois}$

$0000\ 0000\ 0000\ 0000\ 0000\ 1100\ 0000\ 0000_{dois}$

- OR: (bit-a-bit): Resulta 1 se há 1 bit 1 no operando

Ex.

or

=

$0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 0000\ 0000_{dois}$

$0000\ 0000\ 0000\ 0000\ 0011\ 1100\ 0000\ 0000_{dois}$

$0000\ 0000\ 0000\ 0000\ 0011\ 1101\ 0000\ 0000_{dois}$

Construção de uma ULA

- ULA = Unidade Lógica e Aritmética
    - Um componente fundamental dos processadores
- Blocos Básicos na construção ULAs

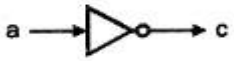
1. Porta AND (c = a · b)

a	b	c = a · b
0	0	0
0	1	0
1	0	0
1	1	1

2. Porta OR (c = a + b)

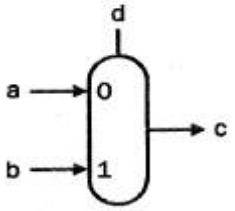
a	b	c = a + b
0	0	0
0	1	1
1	0	1
1	1	1

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)

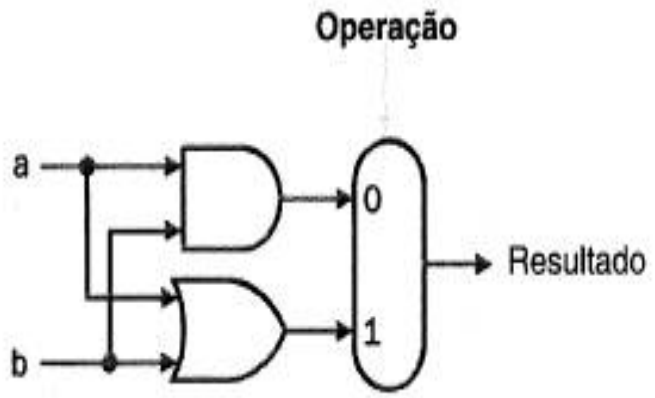
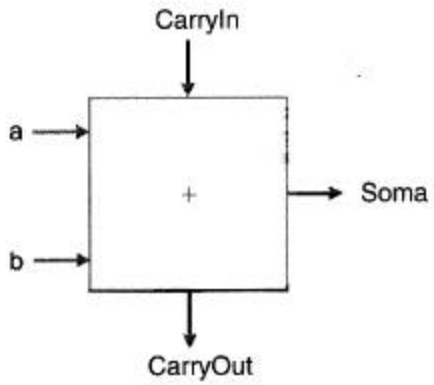


Figura 4.9 Unidade lógica de 1 bit para executar as operações AND e OR.

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		
a	b	CarryIn	CarryOut	Soma	Comentários
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

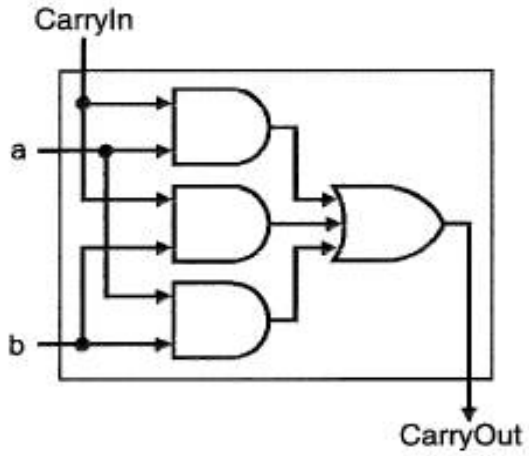
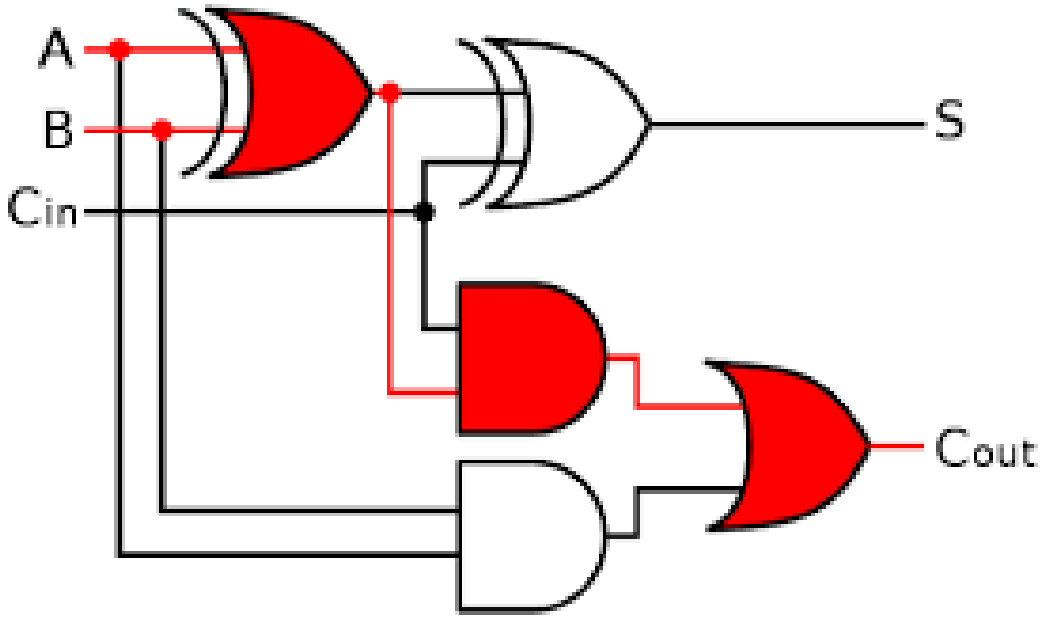


Figura 4.13 Hardware do somador para geração do sinal CarryOut. O restante do hardware do somador é composto pelos circuitos para geração da saída Soma, especificada pelas equações dadas anteriormente.

  - **Circuito Soma ?**

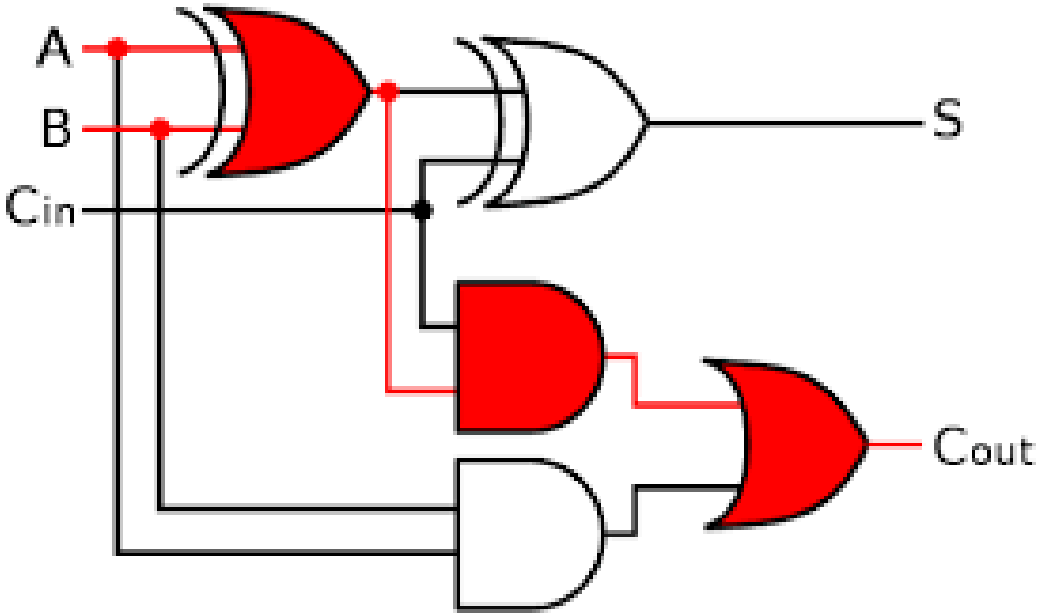
Construção da ULA

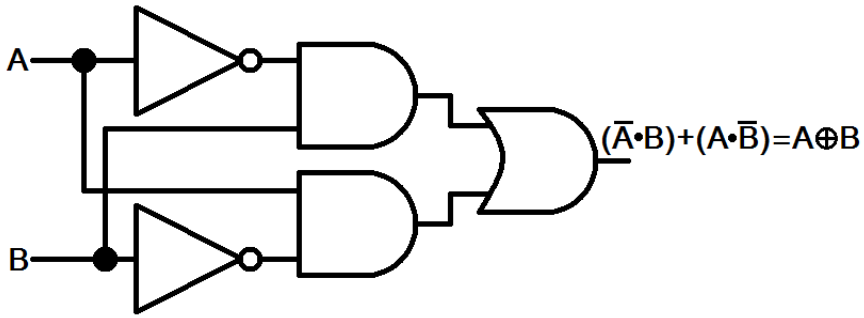
- Circuito Soma ?



Construção da ULA

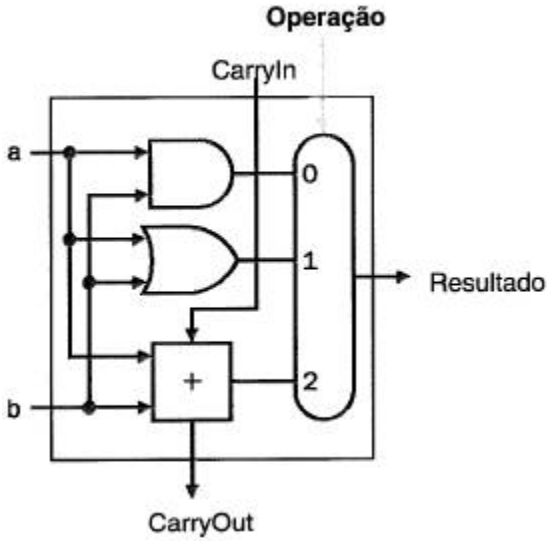
- Circuito Soma ?

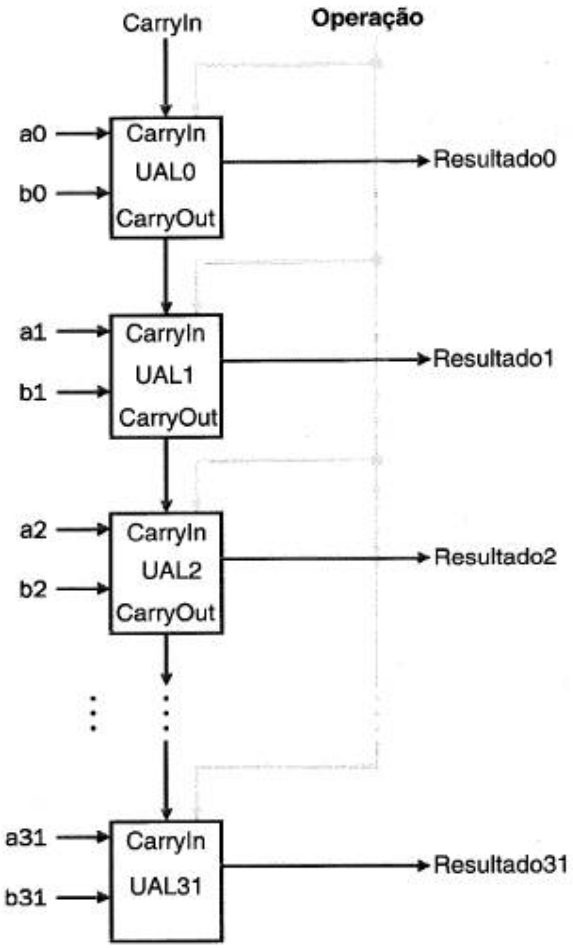




Construção da ULA

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

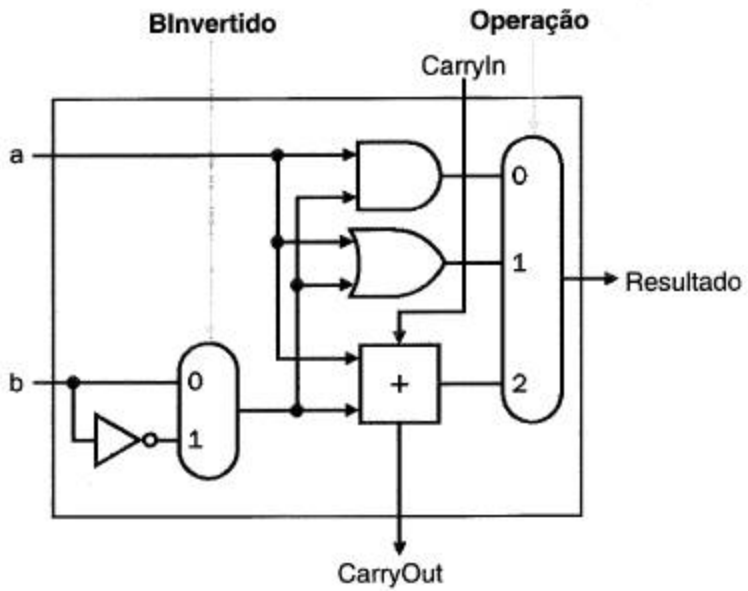




Combinação dos circuitos
(linha Operação com 2 bits)

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

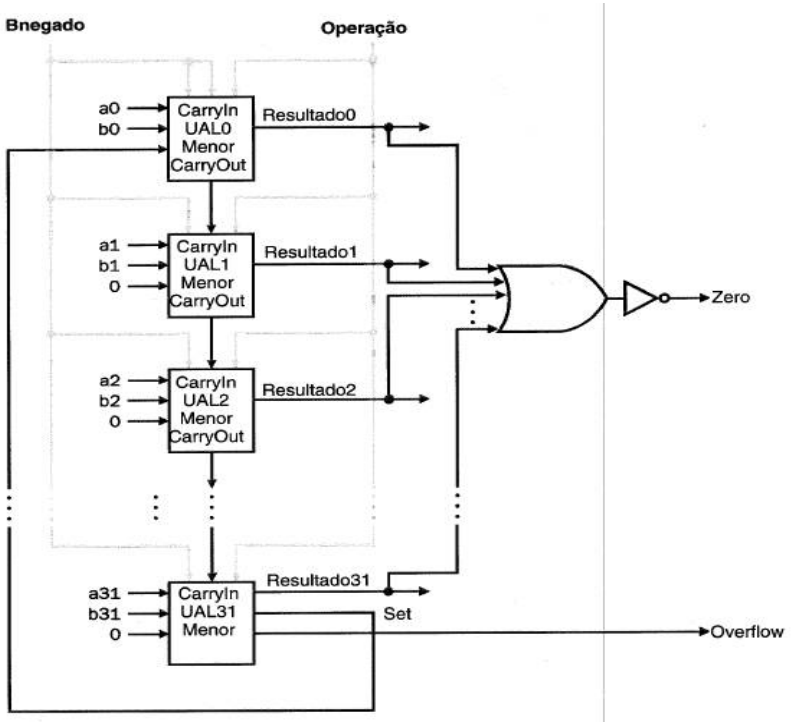
Bits 0~30

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.

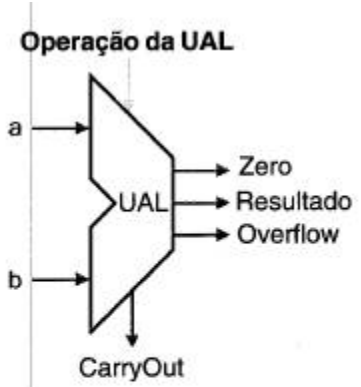


Figura 4.21 Símbolo comumente usado em diagramas lógicos para representar a UAL da Figura 4.19. Este símbolo também pode ser usado para representar um somador, de maneira que ele deve ser identificado explicitamente como sendo uma UAL ou um Somador.

Multiplicação

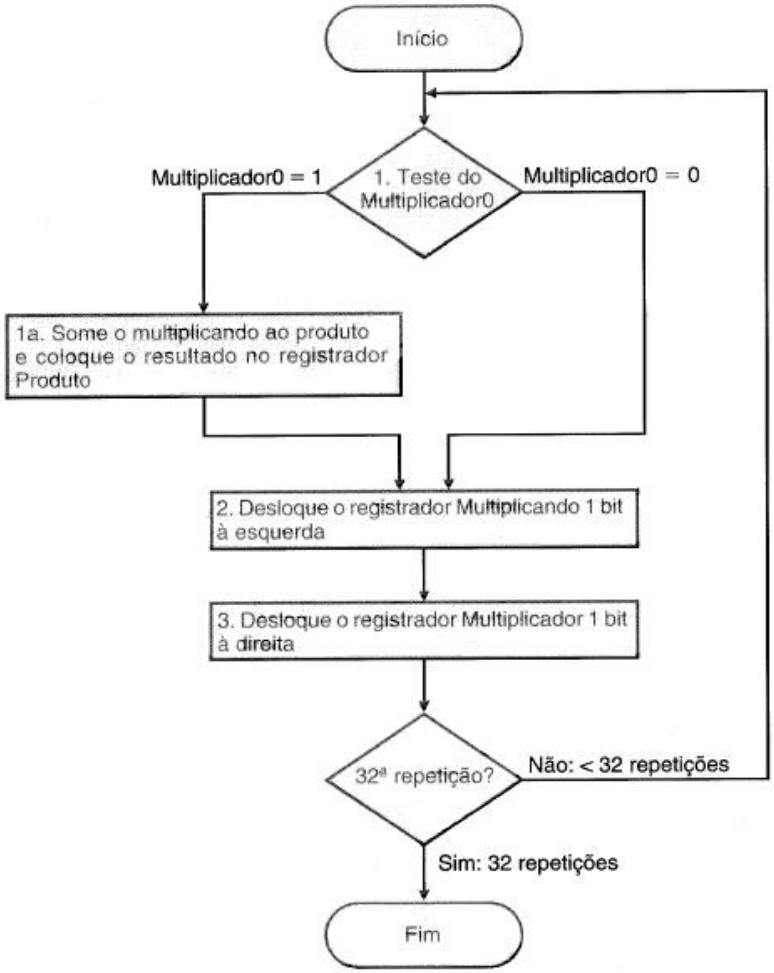
- Importante às ULAs, mas complexo.
- Exemplo como conhecemos:
  - m x n bits = produto com m+n bits
  - No MIPs: 32 x 32 bits = 64 bits, usando regs **hi** e **lo**.

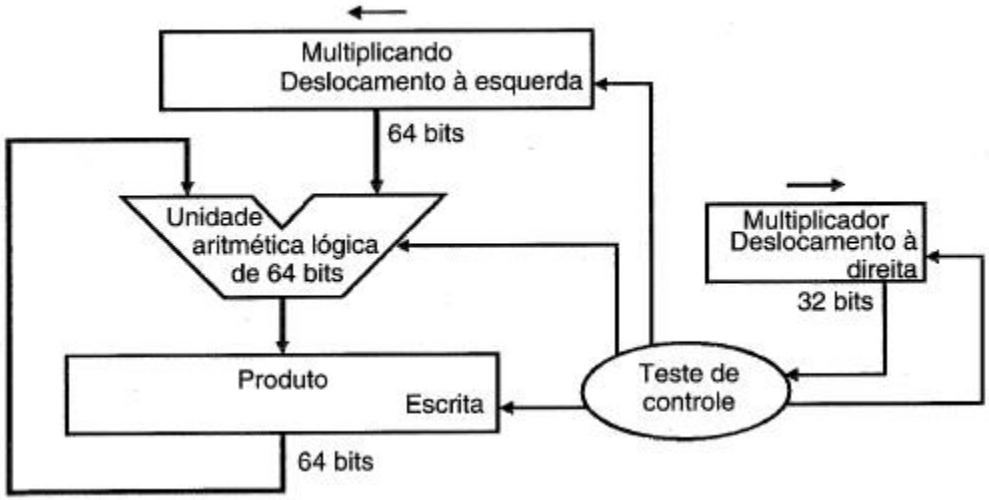
```
Multiplicando        1000   8
Multiplicador       x1001   9


    1000
   0000
  0000
+1000
01001000
```

Multiplicação

- Algoritmo:
  - Produto inicializado em zero
  - Deslocamento para a esquerda do multiplicando.
  - Soma, ou não soma, dependendo do bit do multiplicador.
  - Repete para cada bit do multiplicador deslocando-o à direita





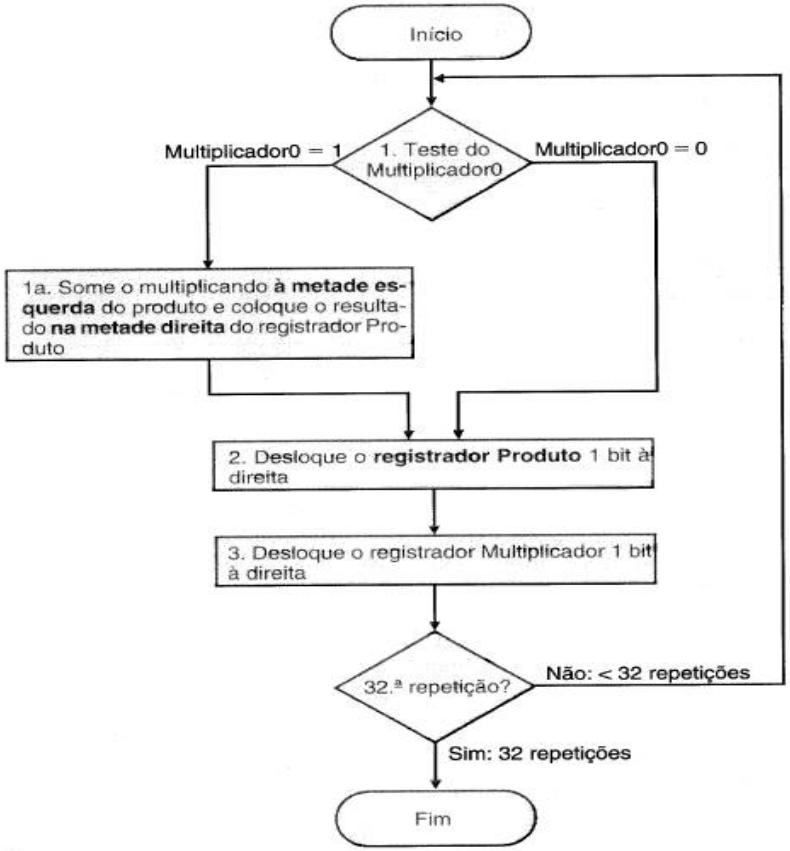
Multiplicação

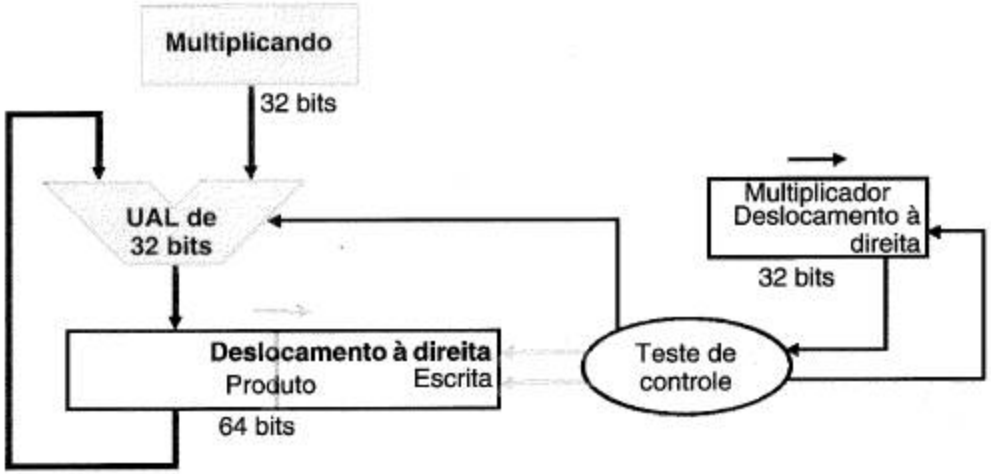
- Exemplo: 2 x 3 (4 bits) = 0010 x 0011

Iteração	Passo	Multiplicador	Multiplicando	Produto
0	Valores iniciais	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	**0000 0010**
	2: Deslocamento à esquerda do Multiplicando	0011	**0000 0100**	0000 0010
	3: Deslocamento à direita do Multiplicador	**0001**	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	**0000 0110**
	2: Deslocamento à esquerda do multiplicando	0001	**0000 1000**	0000 0110
	3: Deslocamento à direita do Multiplicador	**0000**	0000 1000	0000 0110
3	1: 0 ⇒ nenhuma operação	0000	0000 1000	0000 0110
	2: Deslocamento à esquerda do Multiplicando	0000	**0001 0000**	0000 0110
	3: Deslocamento à direita do Multiplicador	**0000**	0001 0000	0000 0110
4	1: 0 ⇒ nenhuma operação	0000	0001 0000	0000 0110
	2: Deslocamento à esquerda do Multiplicando	0000	**0010 0000**	0000 0110
	3: Deslocamento à direita do Multiplicador	**0000**	0010 0000	0000 0110

Multiplicação

- Problema:
  - Muitos ciclos de clock para realizar multiplicação, o que pode impactar muito no desempenho.
  - Ainda, ULA tinha que ter 64 bits de vido ao desloc do multiplocando
- Segunda versão:
  - Desloca prod. à direita ao invés de deslocar multiplicando





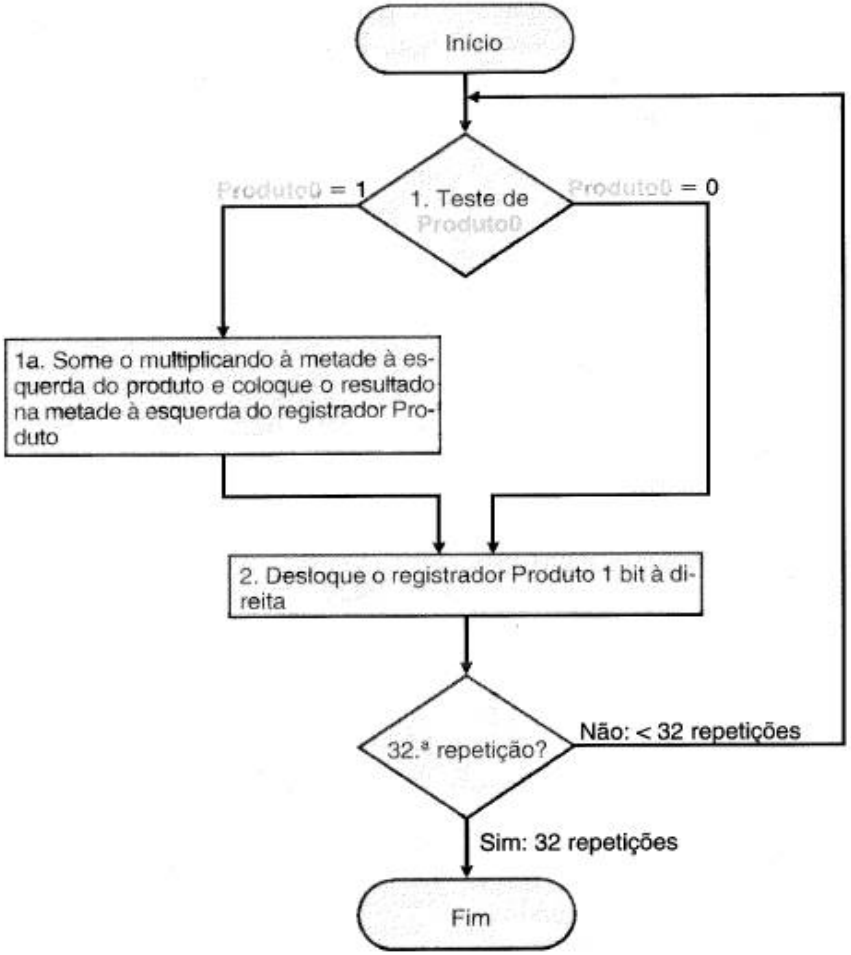
Multiplicação

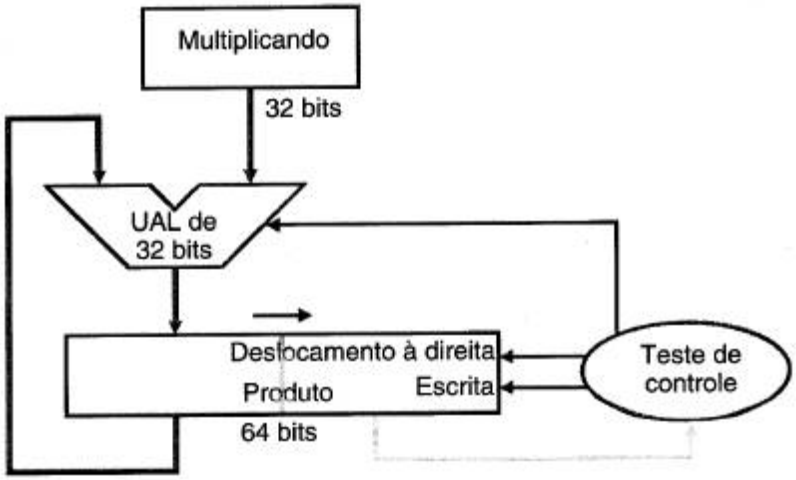
- Exemplo: 2 x 3 (4 bits) = 0010 x 0011

Iteração	Passo	Multiplicador	Multiplicando	Produto
0	Valores iniciais	0011	0010	0000 0000
1	1a: 1 => Prod = Prod + Mcand	0011	0010	**0010** 0000
	2: Deslocamento à direita do Produto	0011	0010	**0001 0000**
	3: Deslocamento à direita do Multiplicador	**0001**	0010	0001 0000
2	1a: 1 => Prod = Prod + Mcand	0001	0010	**0011** 0000
	2: Deslocamento à direita do Produto	0001	0010	**0001 1000**
	3: Deslocamento à direita do Multiplicador	**0000**	0010	0001 1000
3	1: 0 => Nenhuma operação	0000	0010	0001 1000
	2: Deslocamento à direita do Produto	0000	0010	**0000 1100**
	3: Deslocamento à direita do Multiplicador	**0000**	0010	0000 1100
4	1: 0 => nenhuma operação	0000	0010	0000 1100
	2: Deslocamento à direita do Produto	0000	0010	**0000 0110**
	3: Deslocamento à direita do Multiplicador	**0000**	0010	0000 0110

Multiplicacão

- Terceira versão (e final)
  - Combina a parte mais à direita do Produto com o multiplicador, economizando hardware





Multiplicação

- Exemplo: 2 x 3 (4 bits) = 0010 x 0011

Iteração	Passo	Multiplicando	Produto
0	Valores iniciais	0010	0000 0011
1	1a: 1 => Prod = Prod + Mcand	0010	**0010** 0011
	2: Deslocamento à direita do Produto	0010	**0001 0001**
2	1a: 1 => Prod = Prod + Mcand	0010	**0011** 0001
	2: Deslocamento à direita do Produto	0010	**0001 1000**
3	1: 0 => nenhuma operação	0010	0001 1000
	2: Deslocamento à direita do Produto	0010	**0000 1100**
4	1: 0 => nenhuma operação	0010	0000 1100
	2: Deslocamento à direita do Produto	0010	**0000 0110**

Multiplicação

- Multiplicação com Sinal
  - Método simples:
  1. Converter o multiplicador e o multiplicando para números positivos, guardando os sinais originais.

  2. Fazer as 31 iterações, ignorando os sinais.

  3. Trocar o sinal do produto se os sinais do multiplicador e do multiplicando forem diferentes.

Multiplicação

- Algoritmo de Booth
  - Observação antiga, mas ainda verdadeira, de que deslocar bits é mais rápido do que somar
  - 00: Meio de um string de 0s, portanto não há nenhuma operação aritmética a ser realizada.
    01: Fim de um string de 1s, portanto some o multiplicando à metade esquerda do produto.
    10: Início de um string de 1s, portanto subtraia o multiplicando da metade esquerda do produto.
    11: Meio de um string de 1s, portanto não há operação aritmética a ser realizada.

l e

Multiplicação

- Algoritmo de Booth: Exemplo

Iteração	Multiplicando	Algoritmo original		Algoritmo de Booth	
		Passo	Produto	Passo	Produto
0	0010	Valores iniciais	0000 0110	Valores iniciais	0000 0110 0
1	0010	1: 0 => nenhuma operação	0000 0110	1a: 00 ⇒ nenhuma operação	0000 0110 0
	0010	2: Deslocamento à direita do Produto	**0000 0011**	2: Deslocamento à direita do Produto	**0000 0011 0**
2	0010	1a: 1 => Prod = Prod + Mcand	**0010** 0011	1c: 10 ⇒ Prod = Prod – Mcand	**1110** 0011 0
	0010	2: Deslocamento à direita do Produto	**0001 0001**	2: Deslocamento à direita do Produto	**1111 0001 1**
3	0010	1a: 1 => Prod = Prod + Mcand	**0011** 0001	1d: 11 ⇒ nenhuma operação	1111 0001 1
	0010	2: Deslocamento à direita do Produto	**0001 1000**	2: Deslocamento à direita do Produto	**1111 1000 1**
4	0010	1: 0 => nenhuma operação	0001 1000	1b: 01 ⇒ Prod = Prod + Mcand	**0001** 1000 1
	0010	2: Deslocamento à direita do Produto	**0000 1100**	2: Deslocamento à direita do Produto	**0000 1100 0**

Multiplicação

- Algoritmo de Booth funciona com sinal
- Exemplo: 2 x –3 = 0010 x 1101 = 11111010

Iteração	Passo	Multiplicando	Produto
0	Valores iniciais	0010	0000 1101 0
1	1c: 10 ⇒ Prod = Prod – Mcand	0010	**1110** 1101 0
	2: Deslocamento à direita do Produto	0010	**1111 0110** 1
2	1b: 01 ⇒ Prod = Prod + Mcand	0010	**0001** 0110 1
	2: Deslocamento à direita do Produto	0010	**0000 1011** 0
3	1c: 10 ⇒ Prod = Prod – Mcand	0010	**1110** 1011 0
	2: Deslocamento à direita do Produto	0010	**1111 0101** 1
4	1d: 11 ⇒ nenhuma operação	0010	1111 0101 1
	2: Deslocamento à direita do produto	0010	**1111 1010 1**

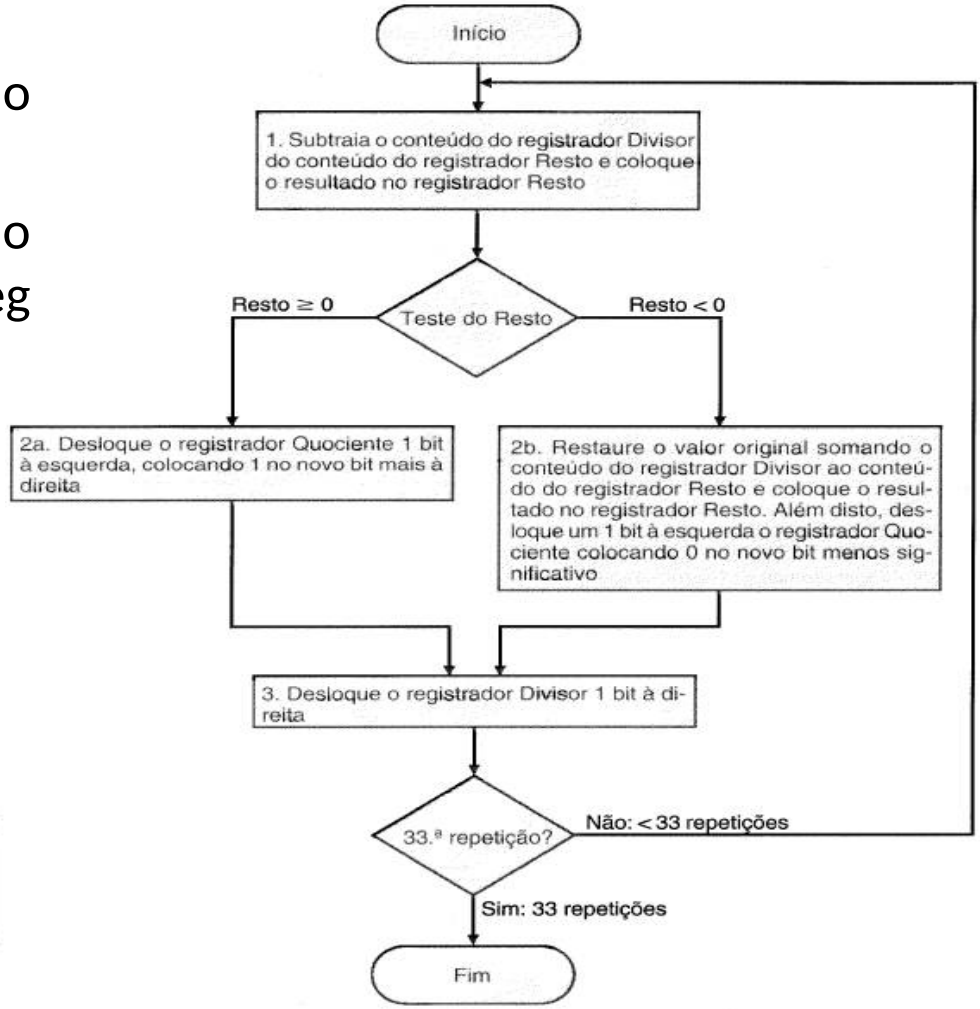
Divisão

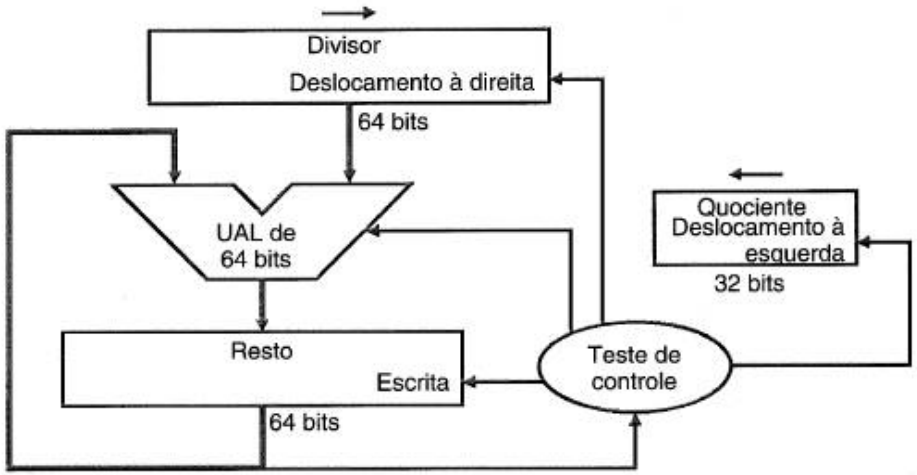
- Menos comum que multiplicação, mas mais complexa
    - Ainda temos o resto
- No MIPS, o resto fica do reg HI e quociente no LO
- Exemplo

$$
\begin{array}{lrl}
 & 1001_{dez} & \text{Quociente} \\
\text{Divisor } 1000_{dez} & \overline{)\,1001010_{dez}} & \text{Dividendo} \\
 & \underline{-1000} & \\
 & 10 & \\
 & 101 & \\
 & 1010 & \\
 & \underline{-1000} & \\
 & 10_{dez} & \text{Resto}
\end{array}
$$

Divisão

- Primeira versão
  - Verifica quantas vezes cabe o divisor no dividendo
  - Divisor inicialmente à esqueda do reg. Divisor, dividendo no reg Resto





Divisão

- Exemplo 7/2 = 0111/0010 (4 bits)

Iteração	Passo	Quociente	Divisor	Resto
0	Valores iniciais	0000	0010 0000	0000 0111
1	1: Resto = Rem – Div	0000	0010 0000	**(1)110 0111**
	2b: Resto < 0 ⇒ +Div, sll Q, Q0 = 0	**0000**	0010 0000	**0000 0111**
	3: Deslocamento do Divisor 1 bit à direita	0000	**0001 0000**	0000 0111
2	1: Resto = Resto – Div	0000	0001 0000	**(1)111 0111**
	2b: Resto < 0 ⇒ +Div, sll Q, Q0 = 0	**0000**	0001 0000	**0000 0111**
	3: Deslocamento do Divisor 1 bit à direita	0000	**0000 1000**	0000 0111
3	1: Resto = Resto – Div	0000	0000 1000	**(1)111 1111**
	2b: Resto < 0 ⇒ +Div, sll Q, Q0 = 0	**0000**	0000 1000	**0000 0111**
	3: Deslocamento do Divisor 1 bit à direita	0000	**0000 0100**	0000 0111
4	1: Resto = Resto – Div	0000	0000 0100	**(0)000 0011**
	2a: Resto ≥ 0 ⇒ sll Q, Q0 = 1	**0001**	0000 0100	0000 0011
	3: Deslocamento do Divisor 1 bit à direita	0001	**0000 0010**	0000 0011
5	1: Resto = Resto – Div	0001	0000 0010	**(0)000 0001**
	2a: Resto ≥ 0 ⇒ sll Q, Q0 = 1	**0011**	0000 0010	0000 0001
	3: Deslocamento do Divisor 1 bit à direita	0011	**0000 0001**	0000 0001

Divisão

- Divisão com sinal
  - Guarda os sinais do divisor e dividendo
  - Realiza operação normalmente
  - Troca sinal do quociente se divisor e dividendo não forem iguais
  - Resto mantém sinal do dividendo

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)

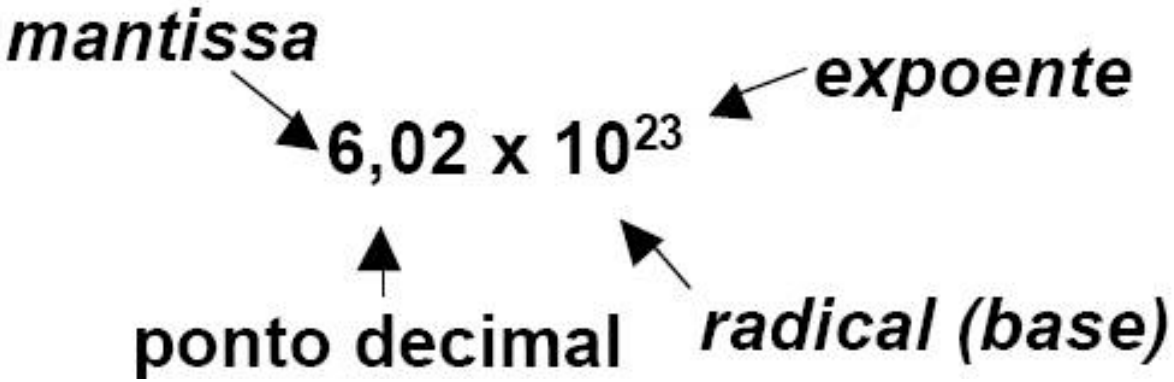
Operações em Ponto Flutuante

- Além dos números inteiros com e sem sinal, como representar os números reais?
  - Números muito grandes? (segundos/século)
    - 3.155.760.00010 (3,1557610 x 109)
  - Números muito pequenos? (diâmetro atômico)
    - 0,0000000110 (1,010 x 10-8) cm
  - Racionais (padrão repetitivo)
    - 2/3 (0,666666666. . .)
  - Irracionais
    - 21/2 (1,414213562373. . .)
    - e (2,718...), Pi(3,141...)

- Todos são representados em notação científica.

Operações em Ponto Flutuante

- Notação Científica



- Forma Normalizada: sem zeros à frente.

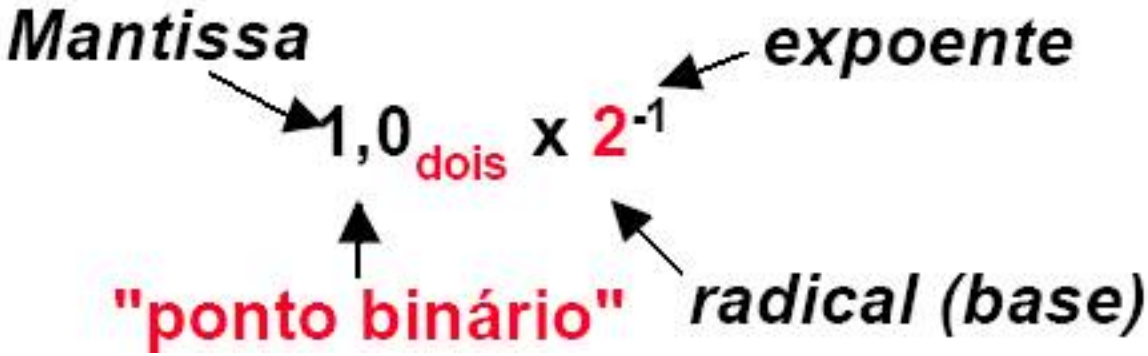
  - Exatamente um dígito à esquerda do ponto decimal

- Alternativas para se representar 1/1.000.000.000 :

  - Normalizada: $1,0 \times 10^{-9}$
  - Não normalizada: $0,1 \times 10^{-8}$ ou $10,0 \times 10^{-10}$

Operações em Ponto Flutuante

- Notação Binária



- A aritmética do computador que suporta isto é chamada de ponto flutuante, porque ela representa números onde o ponto binário não é fixo, como é o caso dos inteiros.

- Declaramos tais variáveis como `float` em C.

Operações em Ponto Flutuante

- Formato Normal:

  $+1,xxxxxxxxxx_{dois} * 2^{yyyy}$

- No MIPS, Múltiplo do Tamanho da Palavra (32 bits):

  - S representa o Sinal do número
  - Expoente representa y's
  - Significando representa x's, **parte fracionária!**
  - O Primeito bit, à esquerda do ponto, **fica implícito quando normalizado!**

- Representa números tão pequenos quanto 2,0 x 10^{-38} até tão grande quanto 2,0 x 10^{38}.

Operações em Ponto Flutuante

Representação:

Sinal, expoente e mantissa:

$$(-1)^{\text{Sinal}} \times \text{mantissa} \times 2^{\text{expoente}}$$

Quanto mais bits a mantissa tiver, maior a precisão.

Quanto mais bits o expoente tiver, maior é a faixa representável.

Operações em Ponto Flutuante

- E se o resultado for muito grande? (> $2{,}0 \times 10^{38}$)
  - Overflow!
  - Overflow => Expoente maior do que pode ser representado com os 8 bits do campo Expoente.

- E se for muito pequeno? (< $2{,}0 \times 10^{-38}$)
  - Underflow!
  - Underflow => Expoente negativo maior do que pode ser representado com os 8 bits do campo Expoente.

- Como reduzir as chances de overflow ou underflow?
  - Representar números maiores?
  - Notação em Precisão Dupla.

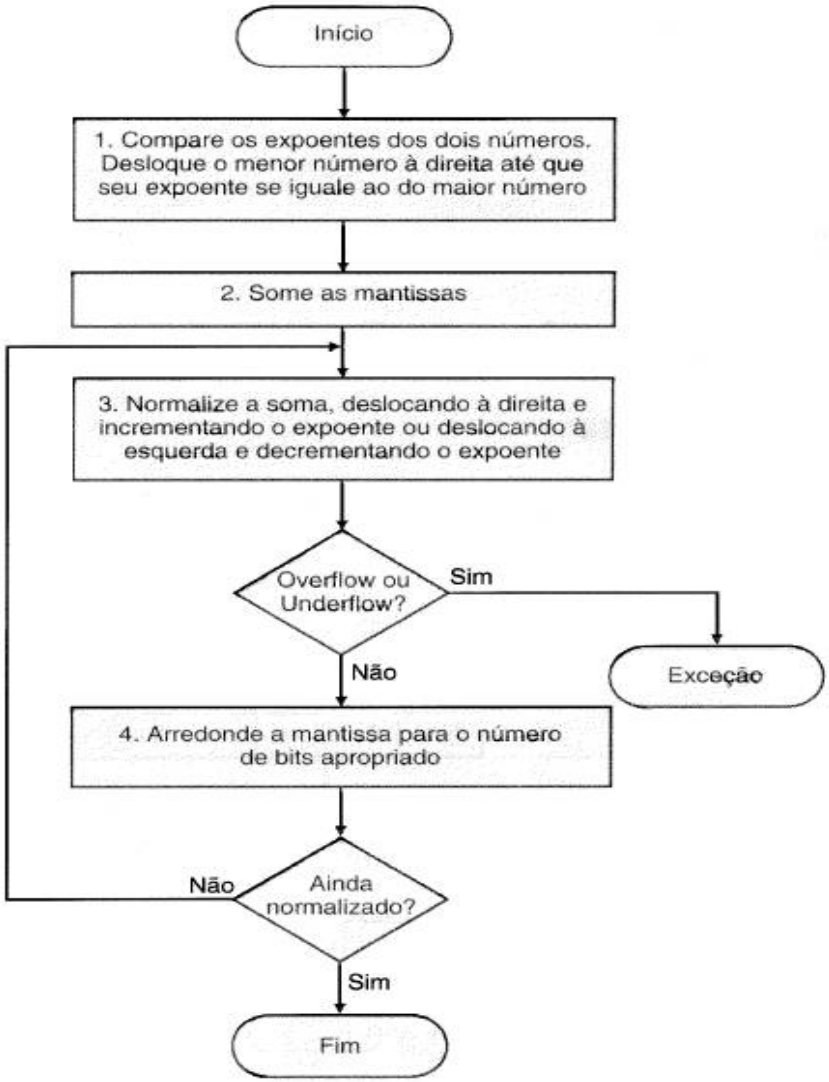
Operações em Ponto Flutuante

- No MIPS, usa 2 palavras (64 bits)
  - Segue IEEE 754 (padrão para interoperabilidade)
  - Aumenta o expoente e o significando

- Precisão Dupla (vs. Precisão Simples)
  - Variável C declarada como double.
  - Representa números quase tão pequenos quanto $2,0 \times 10^{-308}$ e quase tão grandes como $2,0 \times 10^{308}$.
  - Mas a vantagem principal é a maior precisão devido a um significando maior.

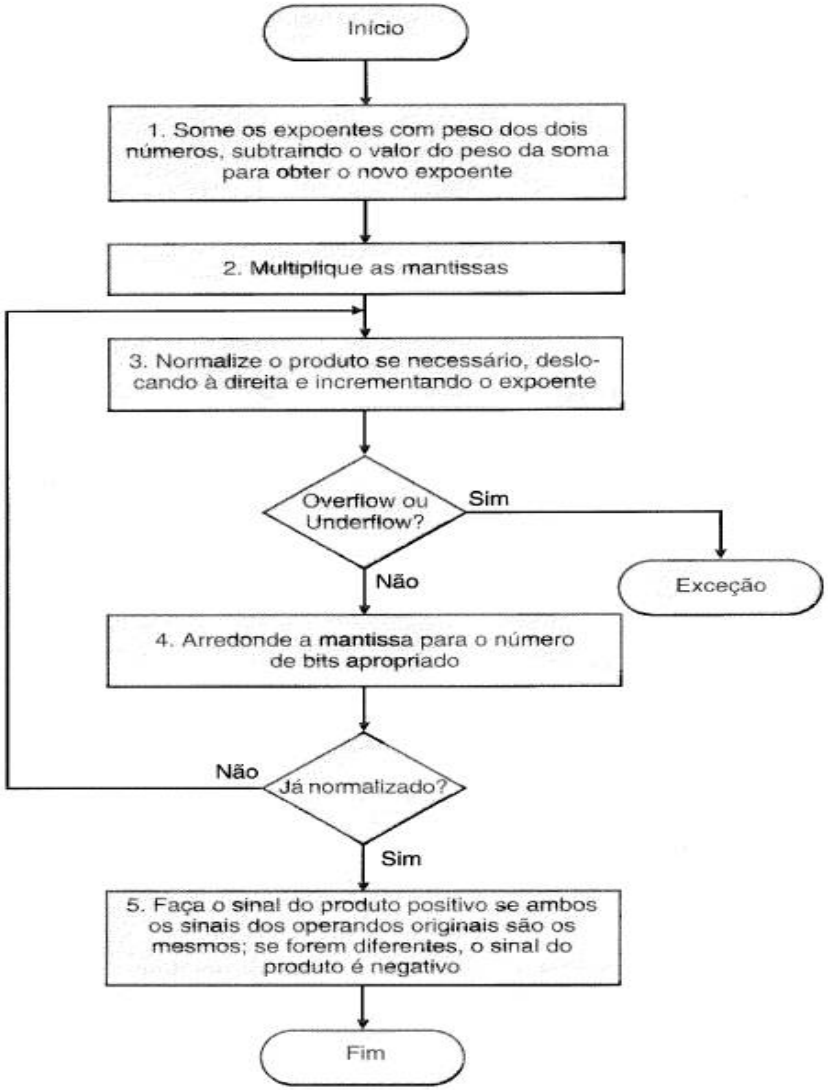
Operações em Ponto Flutuante

- Soma em PF
  - Necessidade de ajustar mantissa no menor número de forma que seu expoente fique igual ao do maior número



Operações em Ponto Flutuante

- Multiplicação em PF
  - Baseado na soma dos expoentes (a base é a mesma)



PF no MIPS

- Instruções separadas para ponto flutuante:
  - Precisão Simples: add.s, sub.s, mul.s, div.s
  - Precisão Dupla: add.d, sub.d, mul.d, div.d
  - Load e store específicos: lwcl e swcl
  - Regs específicos: $f0, $f1, ...
  - Outras

Mais sobre FPs

- Instruções FP são muito mas complicadas do que suas correspondentes inteiras, de modo que devem demorar muito mais para serem executadas.
  - Em 1980, arquiteturas normalmente usavam abordagem de co-processadores, separando o processador principal do ALU de FP (dois chips), devido à complexidade de uma ALU integrada.
  - 1990, com evolução dos CIs, passaram a ficar no mesmo chip.

- Operações em Pcs sempre geram imprecisões, principalmente em FP.
  - Arrendondamentos, truncamentos
  - É impossível representar números reais com HW finito
  - Para aumentar precisão, só usando mais bits
  - Durante uma operação, vários truncamentos intermediários podem ser realizados, causando impacto no resultado
  - IEEE 754 especifica 2 bits de guarda e arredondamento para maior precisão nos cálculos intermediários.

PF no MIPS

Categoria	Instrução	Exemplo	Significado	Comentários
Aritmética	FP add single	add.s $f2, $f4, $f6	$f2 = $f4 + $f6	Soma em PF (precisão simples)
	FP subtract single	sub.s $f2, $f4, $f6	$f2 = $f4 - $f6	Subtração em PF (precisão simples)
	FP multiply single	mul.s $f2, $f4, $f6	$f2 = $f4 × $f6	Multiplicação em PF (precisão simples)
	FP divide single	div.s $f2, $f4, $f6	$f2 = $f4 / $f6	Divisão em PF (precisão simples)
	FP add double	add.d $f2, $f4, $f6	$f2 = $f4 + $f6	Soma em PF (precisão dupla)
	FP subtract double	sub.d $f2, $f4, $f6	$f2 = $f4 - $f6	Subtração em PF (precisão dupla)
	FP multiply double	mul.d $f2, $f4, $f6	$f2 = $f4 × $f6	Multiplicação em PF (precisão dupla)
	FP divide double	div.d $f2, $f4, $f6	$f2 = $f4 / $f6	Divisão em PF (precisão dupla)
Transferência de dados	load word copr. 1	lwc1 $f1.100($s2)	$f1 = Memória($s2 + 100)	Dados de 32 bits para os registradores de PF
	store word copr. 1	swc1 $s1, 100($s2)	Memória[$s2 + 100] = $f1	Dados de 32 bits para a memória
Desvio condicional	branch on FP true	bc1t 25	se(cond == 1) vá para PC + 4 + 100	Desvio relativo ao PC Se Condição (em PF)
	branch on FP false	bc1f 25	se(cond == 0) vá para PC + 4 + 100	Desvio relativo ao PC Se não Condição (em PF)
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.2 $f2, $f4	se($f2 < $f4) cond = 1; senão cond = 0	Comparação de menor que de valores em PF, precisão simples
	FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d $f2, $f4	se($f2 < $f4) cond = 1; senão cond = 0	Comparação de menor que de valores em PF, precisão dupla