

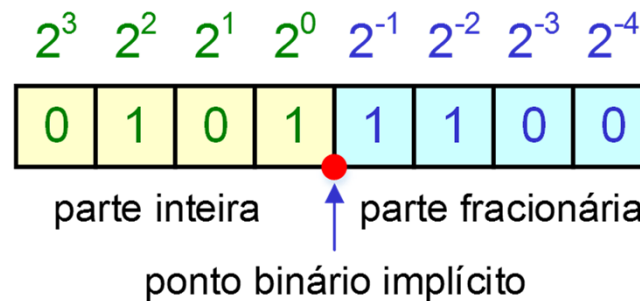
## Aulas 24 e 25

- Representação de números em vírgula flutuante
- A norma IEEE 754
  - Operações aritméticas em vírgula flutuante
  - Precisão simples e precisão dupla
  - Arredondamentos
- Unidade de vírgula flutuante do MIPS
  - Instruções da FPU do MIPS
- Exemplo de codificação utilizando as instruções da FPU do MIPS

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador

# Representação de quantidades fracionárias

- A codificação de quantidades numéricas com que trabalhamos até agora esteve sempre associada à representação de números inteiros
- A representação posicional de inteiros pode também ser usada para representar números racionais considerando-se potências negativas da base
- Por exemplo a representação da quantidade 5.75 em base 2 com 4 bits para a parte inteira e 4 bits para a parte fracionária poderia ser:



- Esta representação designa-se por "**representação em vírgula fixa**"

# Representação de quantidades fracionárias

- A representação de quantidades fracionárias em vírgula fixa coloca de imediato a questão da divisão do espaço de armazenamento para as partes inteira e fracionária
- O número de dígitos da parte inteira determina a **gama de valores representáveis**
- O número de dígitos da parte fracionária, determina a **precisão** da representação (no exemplo anterior, a menor quantidade representável é  $2^{-4} = 0,0625$ )
- No caso geral, quantos dígitos devem então ser reservados para a **parte inteira** e quantos para a **parte fracionária**, sabendo nós que o espaço de armazenamento é limitado?

# Representação de números em Vírgula Flutuante

- **Exemplo: -23.45129876** (vírgula fixa). A mesma quantidade pode também ser representada recorrendo à notação científica:

$$-2.345129876 \times 10^1 \quad -(2 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2} + 5 \times 10^{-3} + \dots + 6 \times 10^{-9}) \times 10^1$$

$$-0.2345129876 \times 10^2 \quad -(0 \times 10^0 + 2 \times 10^{-1} + 3 \times 10^{-2} + 4 \times 10^{-3} + \dots + 6 \times 10^{-10}) \times 10^2$$

- São representações do mesmo valor em que a posição da vírgula tem de ser ponderada, na interpretação numérica da quantidade, pelo valor do expoente de base 10
- Esta técnica, em que a vírgula pode ser deslocada sem alterar o valor representado, designa-se também por **representação em vírgula flutuante (VF)**
- A representação em VF tem a vantagem de não desperdiçar espaço de armazenamento com os zeros à esquerda da quantidade representada
- No primeiro exemplo, o número de dígitos diferentes de zero à esquerda da vírgula é igual a um: diz-se que a **representação está normalizada**

# Representação de números em Vírgula Flutuante

- A representação de quantidades em vírgula flutuante, em sistemas computacionais digitais, faz-se recorrendo à estratégia descrita nos slides anteriores, mas usando agora a base dois:

$$N = (+/-) 1.f \times 2^{\text{Exp}}$$

(representação **normalizada** de uma quantidade binária em vírgula flutuante)

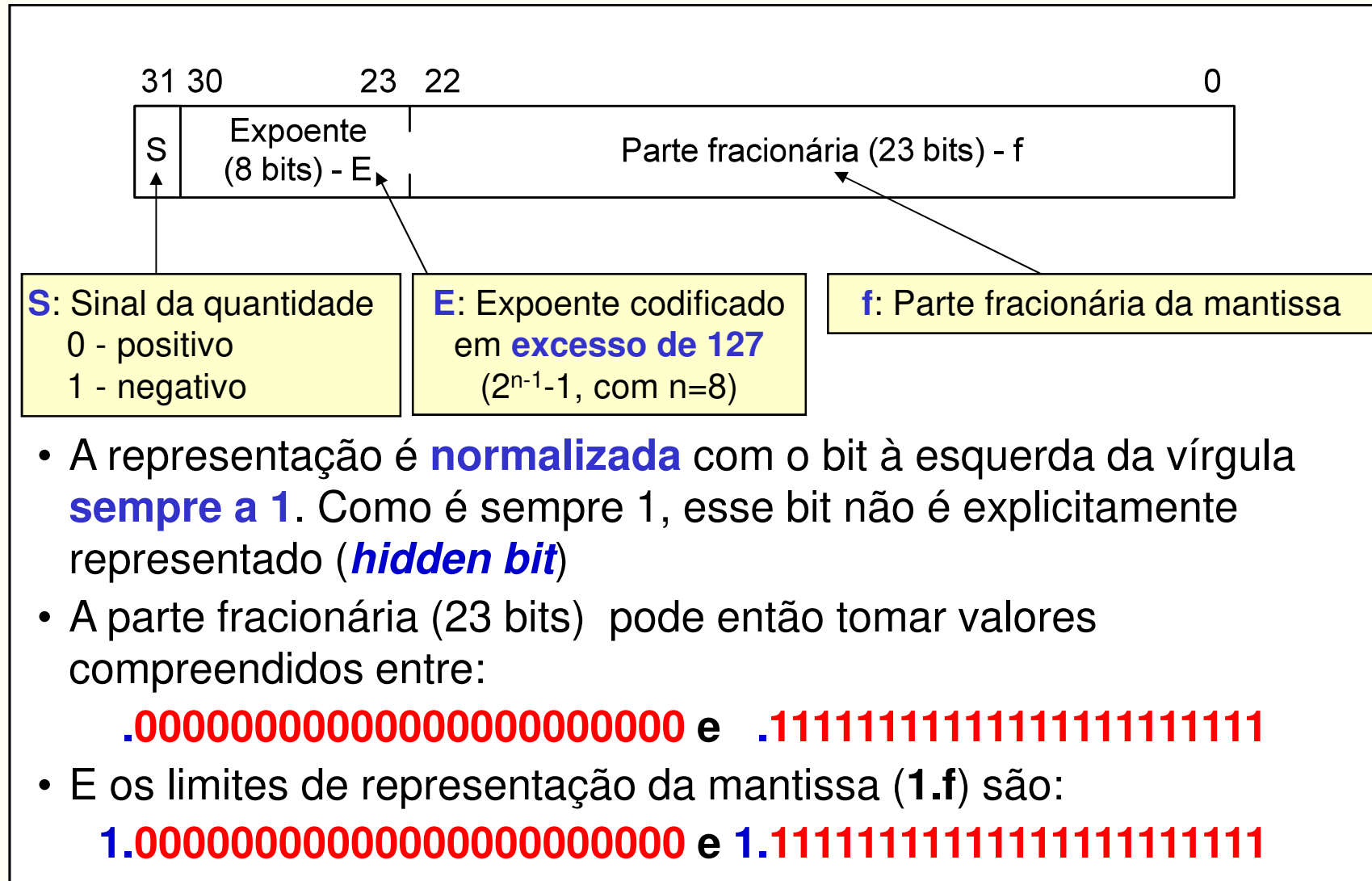
- Em que:

**f** – parte **fracionária** representada por ***n*** bits  
**1.f** – **mantissa** (também designada por significando)  
**Exp** – **expoente** da potência de base 2 representado por ***m*** bits

# Representação de números em Vírgula Flutuante

- O problema da divisão do espaço de armazenamento coloca-se também neste caso, mas agora na determinação do **número de bits** ocupados pela **parte fracionária** e pelo **expoente**
- Essa divisão é um **compromisso** entre **gama de representação** e **precisão**:
  - Aumento do número de bits da parte fracionária  $\Rightarrow$  maior precisão na representação
  - Aumento do número de bits do expoente  $\Rightarrow$  maior gama de representação
- Um bom design implica compromissos adequados!

# Norma IEEE 754 (precisão simples)



## Norma IEEE 754 (precisão simples)



- O expoente é codificado em **excesso de 127** ( $2^{n-1}-1$ ,  $n=8$  bits). Ou seja, é somado ao expoente verdadeiro (Exp) o valor 127 para obter o código de representação (i.e.  **$Exp = E - 127$** , em que E é o expoente codificado)

$$N = (-1)^S 1.f \times 2^{Exp} = (-1)^S 1.f \times 2^{E-127}$$

- O código 127 representa assim o expoente zero, códigos maiores do que 127 representam expoentes positivos e códigos menores que 127 representam expoentes negativos
- O expoente pode, desta forma, tomar valores entre **-126** e **+127** [códigos 1 a 254]. **Os códigos 0 e 255 são reservados**



# Norma IEEE 754 (precisão simples)



**Exemplo:** 0 10000010 101000000000000000000000 (0x41500000)

**Sinal** = 0 (quantidade positiva)

**Expoente** =  $[130] - offset = 130 - 127 = 3 \Leftrightarrow (Exp = E - offset)$

**Mantissa** =  $(1 + \text{parte fracionária}) = 1 + .101 = 1.101$

A quantidade representada, Q, será então:

$$\begin{aligned} Q &= +1.101 \times 2^3 = (1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}) \times 2^3 \\ &= 1.625 \times 8 = 13 \times 10^0 = 13 \end{aligned}$$

# Norma IEEE 754 (precisão simples)



$$N = (-1)^S 1.f \times 2^{\text{Exp}} = (-1)^S 1.f \times 2^{E-127}$$

- A gama de representação suportada por este formato será portanto:  
 $\pm [1.000000000000000000000000 \times 2^{-126}, 1.111111111111111111111111 \times 2^{+127}]$

$$\pm [1.175494 \times 10^{-38}, 3.402824 \times 10^{+38}]$$

- Qual o número de casas decimais? De modo a não exceder a precisão da representação original, a **representação em decimal** deve ter, no máximo, 6 casas decimais:

$$m = \left\lfloor n \frac{\log r}{\log s} \right\rfloor = \left\lfloor 23 \frac{\log 2}{\log 10} \right\rfloor = 6$$

- Ou, sabendo que o nº de bits por casa decimal =  $\log_2(10) \cong 3.3$ , o número de casas decimais é  $\lfloor 23 / 3.3 \rfloor = \mathbf{6 \text{ casas decimais}}$

# Norma IEEE 754 (precisão simples)



- Nas operações com quantidades representadas neste formato podem ocorrer situações de **overflow** e de **underflow**:
  - Overflow**: quando o expoente do resultado não cabe no espaço que lhe está destinado →  **$E > 254$** )
  - Underflow**: caso em que o expoente é tão pequeno que também não é representável →  **$E < 1$** )

$$N_{\text{resultado}} > 1.111111111111111111111111 \times 2^{+127}$$

$$0 < N_{\text{resultado}} < 1.000000000000000000000000 \times 2^{-126}$$

# Norma IEEE 754 (precisão simples)

- **Exemplo:** codificar no formato vírgula flutuante IEEE 754 precisão simples, o valor **-12.59375<sub>10</sub>**

Parte inteira:  $12_{10} = 1100_2$

Parte fracionária:  $0.59375_{10} = 0.10011_2$

$12.59375_{10} = 1100.10011_2 \times 2^0$

Normalização:  $1100.10011_2 \times 2^0 = 1.10010011_2 \times 2^3$

Expoente codificado:  $+3 + 127 = 130_{10} = 10000010_2$


**1** **10000010** **100100110000000000000000**

**0xC1498000**

MSb

0.59375  
× 2  
1.18750  
0.18750  
× 2  
0.37500  
0.37500  
× 2  
0.75000  
0.75000  
× 2  
1.50000  
0.50000  
× 2  
1.00000

LSb



# Norma IEEE 754 – Adição / Subtração

Exemplo:  $N = 1.1101 \times 2^0 + 1.0010 \times 2^{-2}$

**1º Passo:** Igualar os expoentes ao maior dos expoentes

$$a = 1.1101 \times 2^0 \quad b = 0.010010 \times 2^0$$

**2º Passo:** Somar / subtrair as mantissas mantendo os expoentes

$$N = 1.1101 \times 2^0 + 0.010010 \times 2^0 = 10.000110 \times 2^0$$

**3º Passo:** Normalizar o resultado

$$N = 10.000110 \times 2^0 = 1.0000110 \times 2^1$$

**4º Passo:** Arredondar o resultado e renormalizar (se necessário)

$$N = 1.0000110 \times 2^1 = 1.0001 \times 2^1$$

**Exemplo com 4 bits fracionários**

# Norma IEEE 754 – Adição / Subtração

- **Exercício:** A e B representam, em hexadecimal, a codificação no formato IEEE 754 de duas quantidades reais:
  - $A = 0x41600000$
  - $B = 0xC0C00000$
- Realize as seguintes operações, apresentando o resultado codificado no formato IEEE 754, em hexadecimal. Tenha em atenção que as quantidades são codificadas em sinal e módulo.
  - $R1 = A - B$
  - $R2 = B - A$
  - $R3 = A + B$
  - $R4 = B + A$
- Repita o exercício supondo:
  - $A = 0xC0C00000$
  - $B = 0x41600000$

# Norma IEEE 754 – Multiplicação

Exemplo:  $N = (1.1100 \times 2^0) \times (1.1001 \times 2^{-2})$

**1º Passo: Somar** os expoentes

$$\text{Expr} = 0 + (-2) = [0 + 127] + [-2 + 127] = [127 + 125] - 127 = [125] = -2$$

**2º Passo: Multiplicar** as mantissas

$$M_r = 1.1100 \times 1.1001 = 10.101111$$

**3º Passo: Normalizar** o resultado

$$N = 10.101111 \times 2^{-2} = 1.0101111 \times 2^{-1}$$

**4º Passo: Arredondar** o resultado e renormalizar (se necessário)

$$N = 1.0101111 \times 2^{-1} = 1.0110 \times 2^{-1}$$

**Exemplo com 4 bits fracionários**

# Norma IEEE 754 – Divisão

Exemplo:  $N = (1.0010 \times 2^0) / (1.1000 \times 2^{-2})$

**1º Passo: Subtrair** os expoentes

$$\text{Expr} = 0 - (-2) = [0+127] - [-2+127] = [127-125] + 127 = [129] = 2$$

**2º Passo: Dividir** as mantissas

$$M_r = 1.0010 / 1.1000 = 0.11$$

**3º Passo: Normalizar** o resultado

$$N = 0.11 \times 2^2 = 1.1 \times 2^1$$

**4º Passo: Arredondar** o resultado

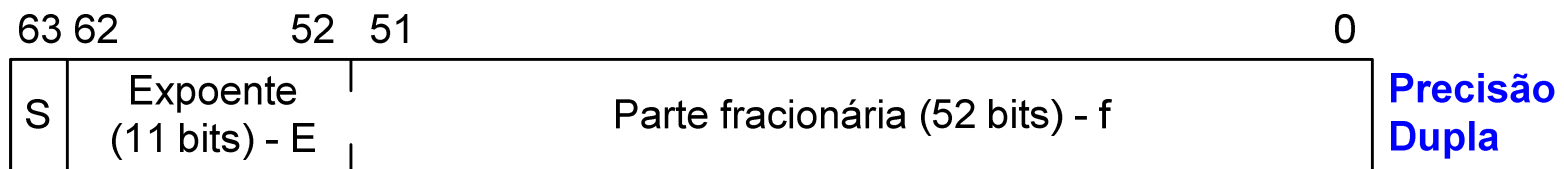
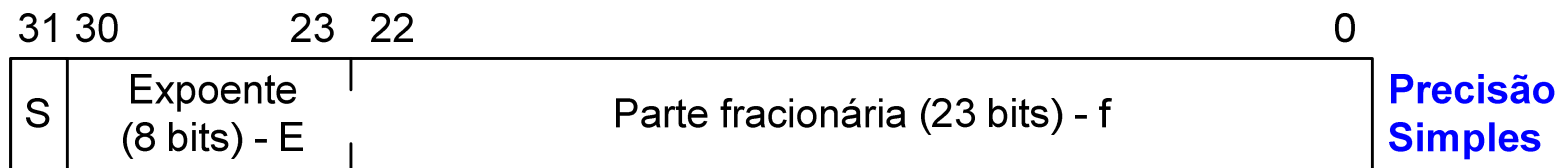
$$N = 1.1 \times 2^1 = 1.1000 \times 2^1$$

**Exemplo com 4 bits fracionários**



# Norma IEEE 754 (precisão dupla)

- A norma IEEE 754 suporta a representação de quantidades em **precisão simples (32 bits)** e em **precisão dupla (64 bits)**



$$N = (-1)^S 1.f \times 2^{(E - 127)} \quad (\text{Precisão simples - tipo float})$$

$$N = (-1)^S 1.f \times 2^{(E - 1023)} \quad (\text{Precisão dupla - tipo double})$$

## Norma IEEE 754 (precisão dupla)



$$N = (-1)^S 1.f \times 2^{\text{Exp}} = (-1)^S 1.f \times 2^{E-1023}$$

- A gama de representação suportada pelo formato de precisão dupla será:

$$\pm [1.0000000000000000 \dots 000 \times 2^{-1022}, 1.1111111111111111 \dots 111 \times 2^{+1023}]$$

$$\pm [2.225073858507201 \times 10^{-308}, 1.797693134862316 \times 10^{+308}]$$

- De modo a não exceder a precisão da representação original, a **representação em decimal** deve ter, no máximo,  $\lfloor 52 / \log_2(10) \rfloor = 15$  **casas decimais**

# Norma IEEE 754 – casos particulares

- A norma IEEE 754 suporta ainda a representação de alguns casos particulares:
  - A **quantidade zero**; essa quantidade não seria representável de acordo com o formato descrito até aqui
  - **+/-infinito**. Exemplos: 1.0 / 0.0, -1.0 / 0.0
  - Resultados não numéricos (**NaN – Not a Number**). Exemplo: 0.0 / 0.0
  - Afim de aumentar a resolução (menor quantidade representável) é ainda possível usar um formato de **mantissa desnormalizada** no qual o bit à esquerda do ponto binário é zero

# Norma IEEE 754 – casos particulares

Precisão Simples		Precisão Dupla		Representa
Expoente	Parte Frac.	Expoente	Parte Frac.	
0	0	0	0	0
0	$\neq 0$	0	$\neq 0$	Quantidade desnormalizada
<i>1 a 254</i>	<i>qualquer</i>	<i>1 a 2046</i>	<i>qualquer</i>	<i>Nº em vírgula flutuante normalizado</i>
255	0	2047	0	Infinito
255	$\neq 0$	2047	$\neq 0$	NaN (Not a Number)

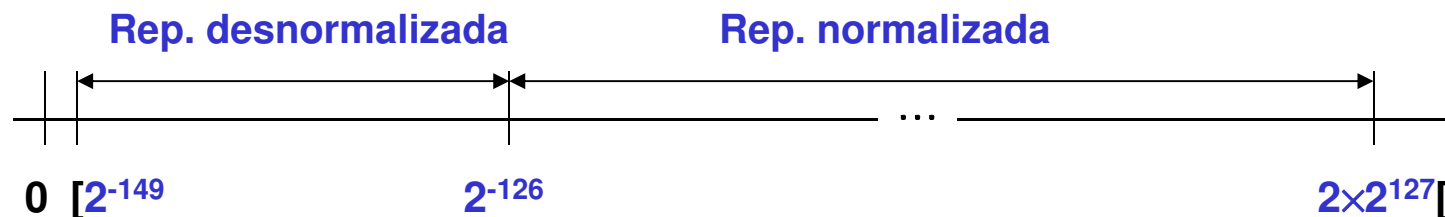
## Norma IEEE 754 – representação desnormalizada

- Permite a representação de quantidades cada vez mais pequenas (com cada vez menos precisão - *underflow* gradual)
- A gama de representação suportada pelo formato de mantissa desnormalizada, em precisão simples, será portanto:

$\pm [0.00000000000000000000000001 \times 2^{-126}, 0.111111111111111111111111 \times 2^{-126}]$

$$\pm [1 \times 2^{-23} \times 2^{-126}, 1.0 \times 2^{-126}]$$

$$\pm [1.401299 \times 10^{-45}, 1.175494 \times 10^{-38}]$$



# Técnicas de arredondamento do resultado

- As operações aritméticas são efetuadas com um número de bits da parte fracionária superior ao disponível no espaço de armazenamento
- Desta forma, na conclusão de qualquer operação aritmética é necessário proceder ao arredondamento do resultado por forma a assegurar a sua adequação ao espaço que lhe está destinado
- As técnicas mais comuns no processo de **arredondamento do resultado** (o qual introduz um erro) são:
  - Truncatura
  - Arredondamento simples
  - Arredondamento para o par (ímpar) mais próximo

# Técnicas de arredondamento do resultado

- **Truncatura** (exemplo com 2 dígitos na parte fracionária:  $d=2$ )

Número	Trunc(x)	Erro
x.00	x	0
x.01	x	-1/4
x.10	x	-1/2
x.11	x	-3/4

$$\begin{aligned}\text{Erro médio} &= (0 - 1/4 - 1/2 - 3/4) / 4 \\ &= -3/8\end{aligned}$$

- Mantém-se a parte inteira, desprezando qualquer informação que exista à direita do ponto binário

# Técnicas de arredondamento do resultado

- **Arredondamento simples** (exemplo com 2 dígitos na parte fracionária:  $d=2$ )

Número	Arred(x)	Erro
x.00	x	0
x.01	x	-1/4
x.10	x + 1	+1/2
x.11	x + 1	+1/4

$$\begin{aligned}\text{Erro médio} &= (0 - 1/4 + 1/2 + 1/4) / 4 \\ &= +1/8\end{aligned}$$

- Mantém-se a parte inteira quando o 1º dígito decimal for 0 ou soma-se “1” à parte inteira quando aquele for “1” (**arred(x) = trunc(x + 0.5)**)
- O erro médio é mais próximo de zero do que no caso da truncatura, mas ligeiramente polarizado do lado positivo



# Técnicas de arredondamento do resultado

- **Arredondamento para o par mais próximo** (exemplo com  $d=2$ )

Número	Arred(x)	Erro	Número	Arred(x)	Erro
x0.00	x0	0	x1.00	x1	0
x0.01	x0	-1/4	x1.01	x1	-1/4
<b>x0.10</b>	<b>x0</b>	<b>-1/2</b>	<b>x1.10</b>	<b>x1 + 1</b>	<b>+1/2</b>
x0.11	x1	+1/4	x1.11	x1 + 1	+1/4

- Semelhante à técnica de arredondamento, mas decidindo, para o caso “**xx.10**”, em função do primeiro dígito à esquerda do ponto binário
- **Erro médio** =  $-1/8 + 1/8 = 0$

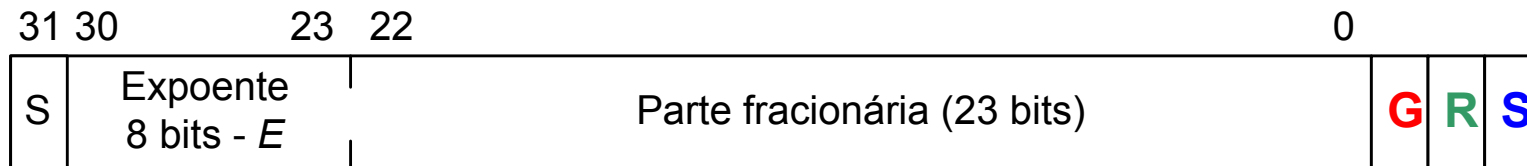
# Norma IEEE 754 – arredondamentos

- Os valores resultantes de cada fase intermédia da execução de uma operação aritmética são armazenados com três bits adicionais, à direita do bit menos significativo da mantissa (i.e., para o caso de precisão simples, com pesos  $2^{-24}$ ,  $2^{-25}$  e  $2^{-26}$ )

? . ?????????????????????????? **G R S** (f c/ 26 bits)

- Objetivos: 1) minimizar o erro introduzido pelo processo de arredondamento e 2) ter bits suplementares para a pós-normalização.
- **G – Guard Bit**
- **R – Round bit**
- **S – Sticky bit** – Bit que resulta da soma lógica de todos os bits à direita do bit R (i.e., se houver à direita de R pelo menos 1 bit a '1', então  $S=1$ )

# Norma IEEE 754 – arredondamentos



Guard bit

Round bit

Sticky bit

**Exemplo 1** (com f de 5 bits,  $A + B$ )

$$A = 1.11010 \times 2^0 \quad B = 1.00100 \times 2^{-2}$$

$$B = 0.0100100 \times 2^0 \text{ (igualar ao maior dos expoentes)}$$

$$\text{Mant}(A+B) = 1.11010 + 0.0100100 \quad \text{Expoente}(A+B) = 0$$

$$= 10.00011 \text{ 000 } G=0, R=0, S=0$$

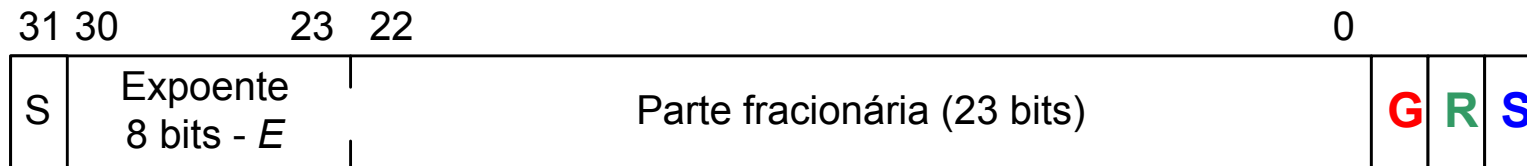
$$\text{Mant}(A+B)_{\text{norm}} = 1.00001 \text{ 100 } G=1, R=0, S=0$$

**Arredondamento:**

$$\text{Mant}(A + B) = 1.00010, \text{ se arred. para o par mais próximo (R=1.00010} \times 2^1)$$

$$\text{Mant}(A + B) = 1.00001, \text{ se arred. para o ímpar mais próximo (R=1.00001} \times 2^1)$$

# Norma IEEE 754 – arredondamentos



**Exemplo 2** (com f de 5 bits,  $A / B$ )

Guard bit  
Round bit  
Sticky bit

$$A = 1.00001 \times 2^2 \quad B = 1.11111 \times 2^{-1}$$

$$\text{Mant}(A/B) = 1.00001 / 1.11111 \quad \text{Expoente}(A/B) = 2 - (-1) = 3$$

$$= 0.10000 \mathbf{1} 100001 \quad G = 1, R = 1, S = \text{OR}(00001) = 1$$

$$= 0.10000 \mathbf{1} 11$$

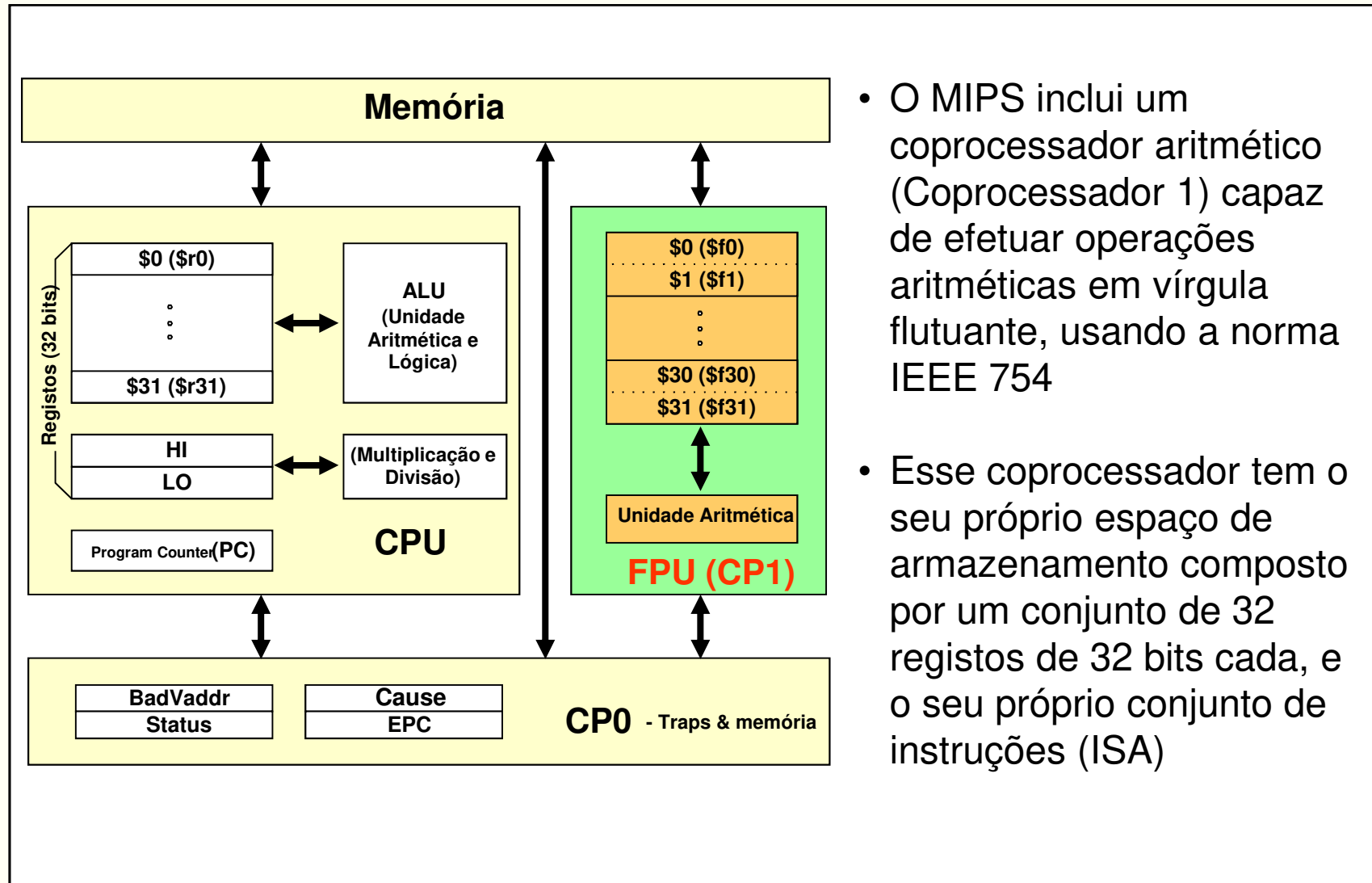
$$\text{Mant}(A/B)_{\text{norm}} = 1.0000 \mathbf{1} 11$$

$$\text{Arred}(1, 11_2) = 10_2$$

**Arredondamento**  $\Rightarrow \text{Mant}(A/B) = 1.00010$

$$A/B = 1.00010 \times 2^2$$

# Cálculo em Vírgula Flutuante no MIPS



## Vírgula Flutuante no MIPS – registos

- Os registos do coprocessador aritmético são designados, no *Assembly* do MIPS, pelas letras **\$fn**, em que o índice **n** toma valores entre 0 e 31
- Cada par de registos consecutivos [**\$fn,\$fn+1**] (**com n par**) pode funcionar como um registo de 64 bits para armazenar valores em **precisão dupla**.
- Em *Assembly* a referência ao par de registos faz-se indicando como operando o **registo par**
- **Apenas os registos de índice par** podem ser usados no contexto das instruções

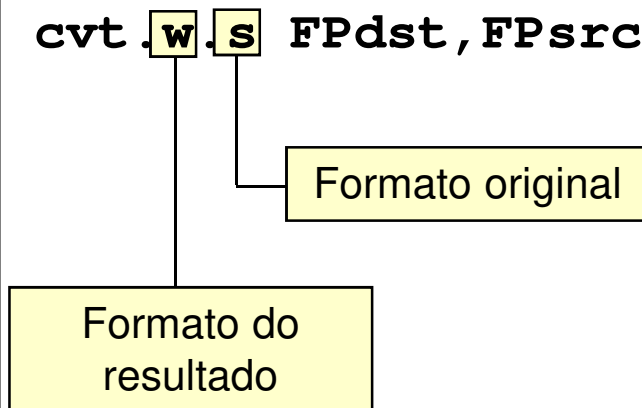
## Vírgula Flutuante no MIPS – instruções aritméticas

<code>abs.p</code>	<code>FPdst,FPsrc</code>	<code># Absolute Value</code>
<code>neg.p</code>	<code>FPdst,FPsrc</code>	<code># Negate</code>
<code>div.p</code>	<code>FPdst,FPsrc1,FPsrc2</code>	<code># Divide</code>
<code>mul.p</code>	<code>FPdst,FPsrc1,FPsrc2</code>	<code># Multiply</code>
<code>add.p</code>	<code>FPdst,FPsrc1,FPsrc2</code>	<code># Addition</code>
<code>sub.p</code>	<code>FPdst,FPsrc1,FPsrc2</code>	<code># Subtract</code>

O sufixo **.p** representa a **precisão** com que é efetuada a operação (simples ou dupla). Deverá, na instrução, ser substituído pelas letras **.s** ou **.d** respetivamente.

## Vírgula Flutuante no MIPS – conversão entre tipos

<code>cvt.d.s FPdst,FPsrc</code>	<code># Convert Single to Double</code>
<code>cvt.d.w FPdst,FPsrc</code>	<code># Convert Integer to Double</code>
<code>cvt.s.d FPdst,FPsrc</code>	<code># Convert Double to Single</code>
<code>cvt.s.w FPdst,FPsrc</code>	<code># Convert Integer to Single</code>
<code>cvt.w.d FPdst,FPsrc</code>	<code># Convert Double to Integer</code>
<code>cvt.w.s FPdst,FPsrc</code>	<code># Convert Single to Integer</code>



As **conversões** entre tipos de representação **são efetuadas pela FPU** pelo que apenas podem ter como operandos/destinos registros da FPU



## Conversão entre tipos – exemplos

[illegible]

**cvt.d.s \$f6,\$f0**

**Exp = (129-127) + 1023 = 1025 = 10000000001**

\$f6=0x00000000    \$f7=**1** **100000000001** **1010000...**0

**\$f6=0x00000000    \$f7=0xC01A0000**

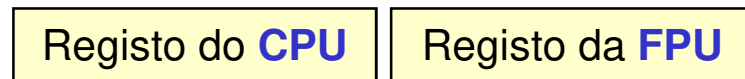
**cvt.w.s    \$f8, \$f0**

$$\text{Exp} = (129 - 127) = 2$$
$$\text{Val} = -1.625 \times 2^2 = -6.5, \quad (\text{int}) (-6.5) = -6$$

**\$f8=0xFFFFFFFFFA**

## Vírgula Flutuante no MIPS – instruções de transferência

- **Transferência de informação** entre registros do CPU e da FPU, e entre registros da FPU



<code>mtc1</code>	<code>CPUSrc, FPdst</code>	<code># Move to Coprocessor 1</code>
<code>mfc1</code>	<code>CPUDst, FPsrc</code>	<code># Move from Coprocessor 1</code>
<code>mov.s</code>	<code>FPdst, FPsrc</code>	<code># Move from FPsrc to FPdst (single)</code>
<code>mov.d</code>	<code>FPdst, FPsrc</code>	<code># Move from FPsrc to FPdst (double)</code>

Estas instruções copiam o conteúdo integral do registro fonte para o registro destino.

**Não efetuam qualquer tipo de conversão entre tipos de informação.**

## Vírgula Flutuante no MIPS – instruções de transferência

- **Transferência de informação** entre registros da FPU e a memória

	Registro da <b>FPU</b>	<b>Endereço de memória</b>	
<b>lwc1</b>	<b>FPdst,</b>	<b>offset (CPUreg)</b>	# Load single from memory
<b>swc1</b>	<b>FPsrc,</b>	<b>offset (CPUreg)</b>	# Store single into memory
<b>ldc1</b>	<b>FPdst,</b>	<b>offset (CPUreg)</b>	# Load double from memory
<b>sdc1</b>	<b>FPsrc,</b>	<b>offset (CPUreg)</b>	# Store double into memory

**Instruções virtuais** (apenas muda a mnemónica):

<b>l.s</b>	<b>FPdst,</b>	<b>offset (CPUreg)</b>	# Load single from memory
<b>s.s</b>	<b>FPsrc,</b>	<b>offset (CPUreg)</b>	# Store single into memory
<b>l.d</b>	<b>FPdst,</b>	<b>offset (CPUreg)</b>	# Load double from memory
<b>s.d</b>	<b>FPsrc,</b>	<b>offset (CPUreg)</b>	# Store double into memory

## Vírgula Flutuante no MIPS – manipulação de constantes

- Nas instruções da FPU do MIPS os operandos têm que residir em registos internos, o que significa que **não há suporte para a manipulação direta de constantes**. Como lidar então com operandos que são constantes?
- **Método 1:**
  - Determinar, em tempo de compilação, o valor que codifica a constante (32 bits para precisão simples ou 64 bits para precisão dupla)
  - Carregar essa constante em 1 ou 2 registos do CPU e copiar o(s) seu(s) valor(es) para o(s) registo(s) da FPU
- **Método 2:**
  - Usar as diretivas “**.float**” ou “**.double**” para definir em memória o valor da constante: 32 bits (**.float**) ou 64 bits (**.double**)
  - Ler o valor da constante da memória para um registo da FPU usando as instruções de acesso à memória vistas anteriormente ( **l.s** ou **l.d** )

## Vírgula Flutuante no MIPS – manipulação de constantes

- O MARS disponibiliza duas instruções virtuais que permitem usar o método 2 (i.e., definição da constante em memória) de forma simplificada. Essas instruções têm o seguinte formato:

**l.s**    \$FPdst, label

**l.d**    \$FPdst, label

em que "**label**" representa o endereço onde a constante está armazenada em memória.

- A decomposição em instruções nativas destas instruções é (admitindo, por exemplo, que a constante K1 está armazenada no endereço **0x1001000C**):

**l.s**    \$f0, k1

    lui    \$1, 0x1001

    lwc1 \$f0, 0x000C(\$1)

**l.d**    \$f0, k1

    lui    \$1, 0x1001

    ldc1 \$f0, 0x000C(\$1)

## Vírgula Flutuante no MIPS – instruções de decisão

- A tomada de decisões envolvendo quantidades em vírgula flutuante realiza-se de forma distinta da utilizada para o mesmo tipo de operação envolvendo quantidades inteiras
- Para quantidades em vírgula flutuante são necessárias duas instruções em sequência: uma **comparação das duas quantidades, seguida da decisão** (que usa a informação produzida pela comparação):
  - A instrução de comparação coloca a **True** ou **False** uma *flag* (1 bit), dependendo de a condição em comparação ser verdadeira ou falsa, respetivamente
  - Em **função do estado dessa flag** a instrução de decisão (instrução de salto) pode alterar a sequência de execução

## Vírgula Flutuante no MIPS – instruções de decisão

```
float a, b;  
...  
  
if( a > b)  
    a = a + b;  
else  
    a = a - b;
```

```
# $f0 ← a  
# $f2 ← b  
...  
if:    c.le.s $f0, $f2           # if(a > b)  
      bc1t  else                # {  
      add.s $f0, $f0, $f2       #     a = a + b;  
      j     endif              # }  
                                     # else  
else:  sub.s $f0, $f0, $f2       #     a = a - b;  
endif:...
```

# Cálculo em Vírgula Flutuante no MIPS

- **Instruções de comparação:**

`c.x.s FPUreg1, FPUreg2 # compare single`

`c.x.d FPUreg1, FPUreg2 # compare double`

Em que **x** pode ser uma das seguintes condições:

**EQ** - equal

**LT** - less than

**LE** - less or equal

- **Instruções de salto:**

`bc1t # branch if true`

`bc1f # branch if false`



# Convenções quanto à utilização de registros

- Registos para **passar parâmetros** para funções:
  - `$f12 ($f13), $f14 ($f15)`
- Registos para **devolução de resultados** das funções:
  - `$f0 ($f1), $f2 ($f3)`
- Registos que **não podem ser alterados pelas funções**:
  - `$f20 ($f21) ... $f30 ($f31)`
- Registos que **podem ser alterados pelas funções**:
  - `$f4 ($f5) ... $f10 ($f11)`
  - `$f16 ($f17), $f18 ($f19)`

# Cálculo em Vírgula Flutuante no MIPS – Exemplo

```
float func(float, int);

void main(void)
{
    float res;

    res = func( 12.5E-2, 2 );
    printFloat( res ); // syscall 2
}

float func(float a, int k)
{
    float val;
    if( a >= -5.6)
        val = (float)k * (a - 32.0);
    else
        val = 0.0;
    return val;
}
```

# Tradução C / Assembly

```
void main(void)
{
    float res;

    res = func( 12.5E-2, 2 );
    printFloat( res ); // syscall 2
}
```

```
float func(float a, int k)
```

```
        .data
k1:     .float 12.5E-2
k2:     .float -5.6
k3:     .float 32.0
k4:     .float 0.0
        .text
        .globl main
main:   ...                # void main(void) {
        l.s      $f12, k1   #
        li       $a0, 2     #
        jal      func       #
        mov.s    $f12, $f0   #      res = func(12.5E-2, 2)
        li       $v0, 2     #
        syscall    #      print_float(res)
        ...
        jr       $ra        # }
```

# Tradução C / Assembly

```
float func(float a, int k)
{
    float val;
    if( a >= -5.6)
        val = (float)k * (a - 32.0);
    else
        val = 0.0;
    return val;
}
```

<pre># \$f4 ← val func:  l.s      \$f4, k2       c.lt.s   \$f12, \$f4       bclt    else       mtc1    \$a0, \$f0       cvt.s.w \$f0, \$f0       l.s     \$f4, k3       sub.s   \$f4, \$f12, \$f4       mul.s   \$f4, \$f0, \$f4       j       endif else:  l.s     \$f4, k4 endif: mov.s   \$f0, \$f4       jr      \$ra</pre>	<pre># float func(float, int){ #   \$f4 = -5.6 #   if( a &gt;= -5.6 ) #   { #       \$f0 = k #       \$f0 = (float)k #       val = 32.0 #       val = a - 32.0 #       val = (float)k * val #   } else #       val = 0.0 #   return val; # }</pre>
---	--