

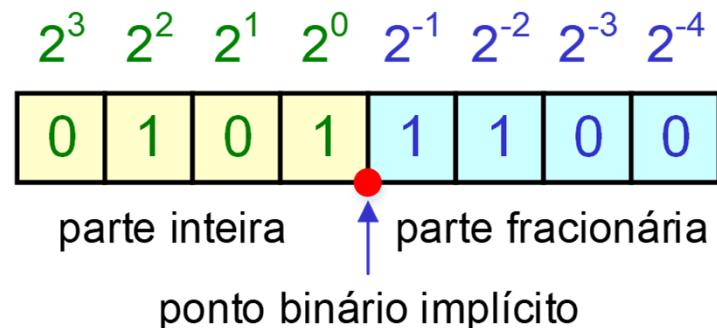
## Aulas 12 e 13

- Representação de números em vírgula flutuante
- A norma IEEE 754
  - Operações aritméticas em vírgula flutuante
  - Precisão simples e precisão dupla
  - Casos particulares
  - Representação desnormalizada
  - Arredondamentos
- Unidade de vírgula flutuante do MIPS
  - Instruções da FPU do MIPS
  - Análise de um exemplo de tradução de C para assembly

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

# Representação de quantidades fracionárias

- A codificação de quantidades numéricas com que trabalhámos 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
- Quantos bits devem ser reservados para a **parte inteira** e quantos para a **parte fracionária**, sabendo nós que o espaço de armazenamento é limitado?
- O número de bits da parte inteira determina a **gama de valores representáveis** ( $2^4$ , no exemplo anterior)
- O número de bits da parte fracionária, determina a **precisão** da representação (passos de  $2^{-4} = 0.0625$ , no exemplo anterior)

# Representação de números em Vírgula Flutuante

- **Exemplo:** **-23.45129** (vírgula fixa). A mesma quantidade pode também ser representada recorrendo à notação científica:

$$-2.345129 \times 10^1$$

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

$$-0.2345129 \times 10^2$$

$$-(0 \times 10^0 + 2 \times 10^{-1} + 3 \times 10^{-2} + 4 \times 10^{-3} + \dots + 9 \times 10^{-7}) \times 10^2$$

- 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 no slide anterior, mas usando agora a base dois:

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

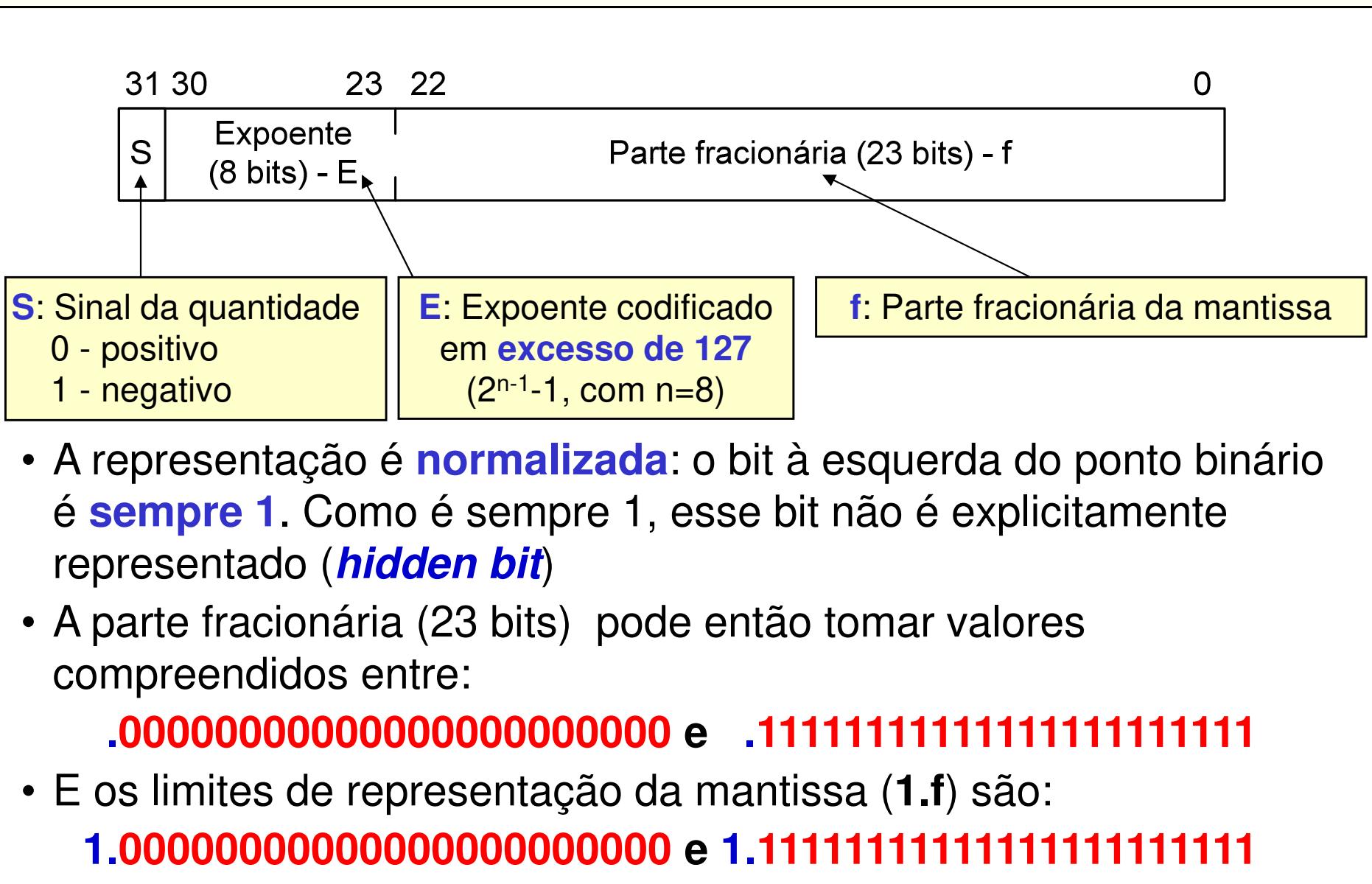
(representação em binário de uma quantidade real, no formato de  
**vírgula flutuante normalizada**)

- 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.  $E = Exp + 127$ , em que E é o expoente codificado)

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

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

# Norma IEEE 754 (precisão simples)



**Exemplo:** Qual o valor, em decimal, representado em **0x41580000**?

**0 10000010 10110000000000000000000000000000**

**Sinal** = 0 (quantidade positiva)

**Expoente** =  $130 - \text{offset} = 130 - 127 = 3 \Leftrightarrow (\text{Exp} = E - \text{offset})$

**Mantissa** =  $(1 + \text{parte fracionária}) = 1 + .1011 = 1.1011$

A quantidade representada (R) será então:  $+1.1011 \times 2^3$

$$R = +1.1011 \times 2^3 = (1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4}) \times 2^3$$

$$= +1.6875 \times 8 = +13.5 \quad (+1.1011 \times 2^3 = +1101.1_2 = +13.5)$$

# Norma IEEE 754 (precisão simples)

- **Exemplo:** codificar no formato vírgula flutuante IEEE 754 precisão simples, o valor  $-12593.75_{10} \times 10^{-3}$

$$-12593.75 \times 10^{-3} = -12.59375$$

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:  $+3 + 127 = 130_{10} = 1000010_2$

**1 1000010 100100110000000000000000**

**0xC1498000**

MSb	0.59375
	$\times 2$
	1.18750
	0.18750
	$\times 2$
	0.37500
	0.37500
	$\times 2$
	0.75000
	0.75000
	$\times 2$
	1.50000
	0.50000
	$\times 2$
LSb	1.00000

# Norma IEEE 754 (precisão simples)

- A gama de representação suportada por este formato será portanto:  
 $\pm [1.0000000000000000000000000000000 \times 2^{-126}, 1.11111111111111111111111 \times 2^{+127}]$   
 $\pm [1.175494 \times 10^{-38}, 3.402824 \times 10^{+38}]$
- Qual o número de dígitos à direita da vírgula na representação em decimal (casas decimais)?
- Partindo de uma representação com "**n**" dígitos fracionários na base "**r**", o número máximo de dígitos na base "**s**" que garante que a mudança de base não acrescenta precisão à representação original é:

$$m = \left\lfloor n \frac{\log r}{\log s} \right\rfloor \quad \lfloor . \rfloor \text{ é o operador } \textit{floor}$$

- Assim, 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 =  $\log_2(10) \approx 3.3$ , o número de casas decimais é  $\lfloor 23 / 3.3 \rfloor = 6$  **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 →  $E > 254$ )  
 $N_{resultado} > 1.11111111111111111111111 \times 2^{+127}$
  - **Underflow**: caso em que o expoente é tão pequeno que também não é representável →  $E < 1$ )  
 $0 < N_{resultado} < 1.00000000000000000000000 \times 2^{-126}$

# Norma IEEE 754 – Adição / Subtração

$$\begin{array}{r} 1,110100 \\ + 0,010010 \\ \hline 10,000110 \end{array}$$

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)

$$N = 1.0000\cancel{1}10 \times 2^1 = 1.0001 \times 2^1$$

$$\begin{array}{r} 1.0000\cancel{1}1 \\ + 0.0000\cancel{1}0 \\ \hline 1.000101 \end{array}$$

Exemplo com 4 bits fracionários

# Norma IEEE 754 – Multiplicação

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

**1º Passo:** Somar os expoentes

$$\text{Exp. Resultado} = 0 + (-2) = -2$$

**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)

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

1.0101	111
+ 0.0000	100
1.0110 011	

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

$$\text{Exp. Resultado} = 0 - (-2) = 2$$

**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

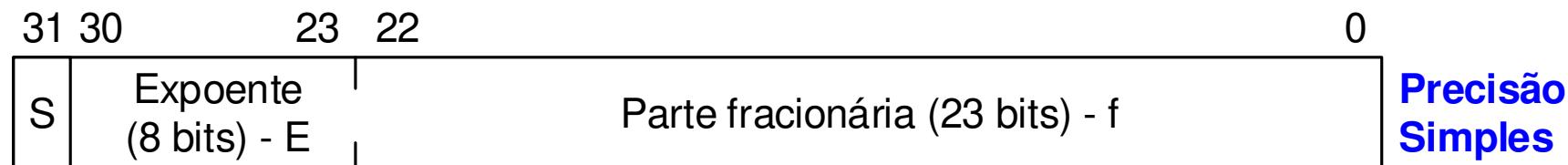
$$N = 1.1 \times 2^1 = 1.1000 \times 2^1$$

1.1000	0
+ 0.0000	1
<hr/>	
1.1000	1

Exemplo com 4 bits fracionários

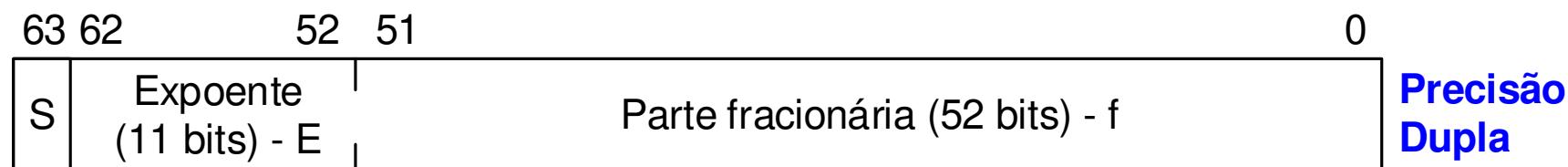
# Norma IEEE 754 (precisão dupla)

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



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

- e em **precisão dupla (64 bits)**



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

# Norma IEEE 754 (precisão dupla)



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

- Na codificação do expoente, os códigos 0 e 2047 são reservados. O expoente pode então tomar valores entre -1022 e +1023 [códigos 1 a 2046]
- A gama de representação suportada pelo formato de precisão dupla será:  
 $\pm [1.000000000000000...000 \times 2^{-1022}, 1.111111111111111...111 \times 2^{+1023}]$   
 $\pm [2.225073858507201 \times 10^{-308}, 1.797693134862316 \times 10^{+308}]$
- 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 (inf)**. Gama de representação excedida; divisão por 0. Exemplos:  $1.0 / 0.0$ ,  $-1.0 / 0.0$
  - Resultados não numéricos (**NaN – Not a Number**). Exemplo:  $0.0 / 0.0$ ,  $inf / inf$ ,  $nan * 2$
  - 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

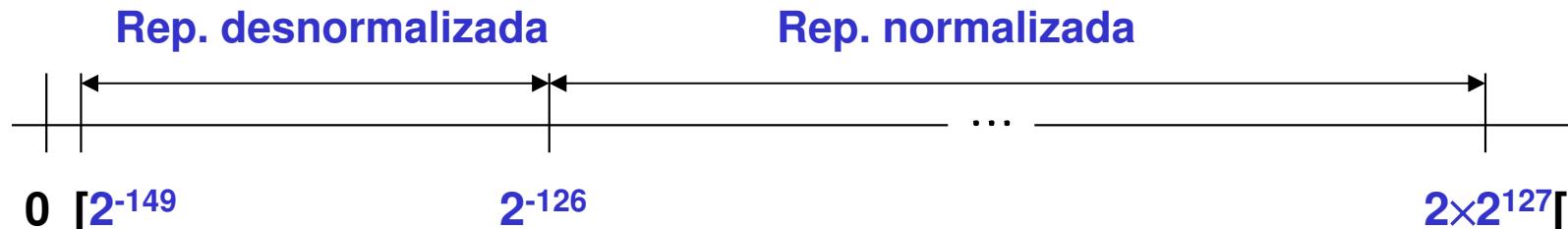
Precisão Simples		Precisão Dupla		Representa
Expoente	Parte Frac.	Expoente	Parte Frac.	
0	0	0	0	0
0	$\neq 0$	0	$\neq 0$	Quantidade desnormalizada
1 a 254	<i>qualquer</i>	1 a 2046	<i>qualquer</i>	<i>Nº em vírgula flutuante normalizado</i>
255	0	2047	0	Infinito
255	$\neq 0$	2047	$\neq 0$	NaN (Not a Number)

# Norma IEEE 754 – representação desnormalizada

- Representação com mantissa desnormalizada: assume-se que o bit à esquerda do ponto binário é 0
- O expoente codificado “E” é 0; **o expoente verdadeiro é -126 (precisão simples) ou -1022 (precisão dupla)**
- Permite a representação de quantidades cada vez mais pequenas (*underflow gradual*)
- Gama de representação com mantissa desnormalizada, em precisão simples:

$$\pm [0.00000000000000000000000000000001 \times 2^{-126}, 0.11111111111111111111111111111111 \times 2^{-126}]$$

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



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

# 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 bits na parte fracionária: d=2)

val	Trunc(val)	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 bits na parte fracionária: d=2)

val	Arred(val)	Erro
x.00	x	0
x.01	x	$x - x.25 = -1/4$
x.10	$x + 1$	$(x+1) - x.5 = +1/2$
x.11	$x + 1$	$(x+1) - x.75 = +1/4$

**Erro médio**

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

- Soma-se 1 ao 1º bit à direita do ponto binário e truncase o resultado (**arred(val)** = **trunc(val + 0.5)**)

$$\begin{array}{r} x.00 \\ + 0.1 \\ \hline x.10 \end{array} \quad \begin{array}{r} x.01 \\ + 0.1 \\ \hline x.11 \end{array} \quad \begin{array}{r} x.10 \\ + 0.1 \\ \hline x+1.00 \end{array} \quad \begin{array}{r} x.11 \\ + 0.1 \\ \hline x+1.01 \end{array}$$

- 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 2 bits na parte fracionária: d=2)

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

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

- Semelhante à técnica de arredondamento simples, mas decidindo, para o caso “**xx.10**”, em função do primeiro bit à esquerda do ponto binário

$$\begin{array}{r} \text{x0.10} \\ \downarrow \\ + 0.0 \\ \hline \text{x0.10} \end{array} \quad \begin{array}{r} \text{x1.10} \\ \downarrow \\ + 0.1 \\ \hline \text{x1 + 1.00} \end{array}$$

- **Erro médio**  $= (0 - 1/4 - 1/2 + 1/4) / 4 + (0 - 1/4 + 1/2 + 1/4) / 4$   
 $= -1/8 + 1/8 = 0$

# Técnicas de arredondamento do resultado

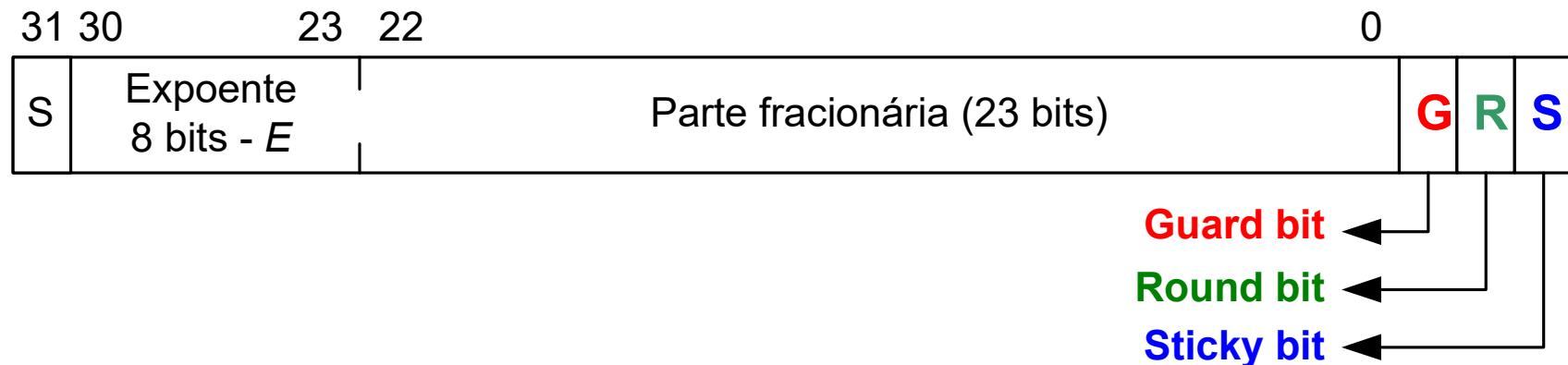
O que fica à direita de $b_{23}$	Exemplo	Resultado
< 0.5	<b>1.b<sub>1</sub>b<sub>2</sub> ... b<sub>22</sub>b<sub>23</sub> 011</b>	<i>Round down</i> : bits à direita de $b_{23}$ são descartados
> 0.5	<b>1.b<sub>1</sub>b<sub>2</sub> ... b<sub>22</sub>b<sub>23</sub> 101</b>	<i>Round up</i> : soma-se 1 a $b_{23}$ (propagando o carry)
= 0.5	<b>1.b<sub>1</sub>b<sub>2</sub> ... b<sub>22</sub>1 100</b>	<i>Round up</i> : soma-se 1 a $b_{23}$ (propagando o carry) (*)
= 0.5	<b>1.b<sub>1</sub>b<sub>2</sub> ... B<sub>22</sub>0 100</b>	<i>Round down</i> : bits à direita de $b_{23}$ são descartados (*)
= 0.5	<b>1.b<sub>1</sub>b<sub>2</sub> ... B<sub>22</sub>1 100</b>	<i>Round down</i> : bits à direita de $b_{23}$ são descartados (**)
= 0.5	<b>1.b<sub>1</sub>b<sub>2</sub> ... b<sub>22</sub>0 100</b>	<i>Round up</i> : soma-se 1 a $b_{23}$ (propagando o carry) (**)

(\*) Arredondamento para o **par mais próximo**.

(\*\*) Arredondamento para o **ímpar mais próximo**.

# Norma IEEE 754 – arredondamentos

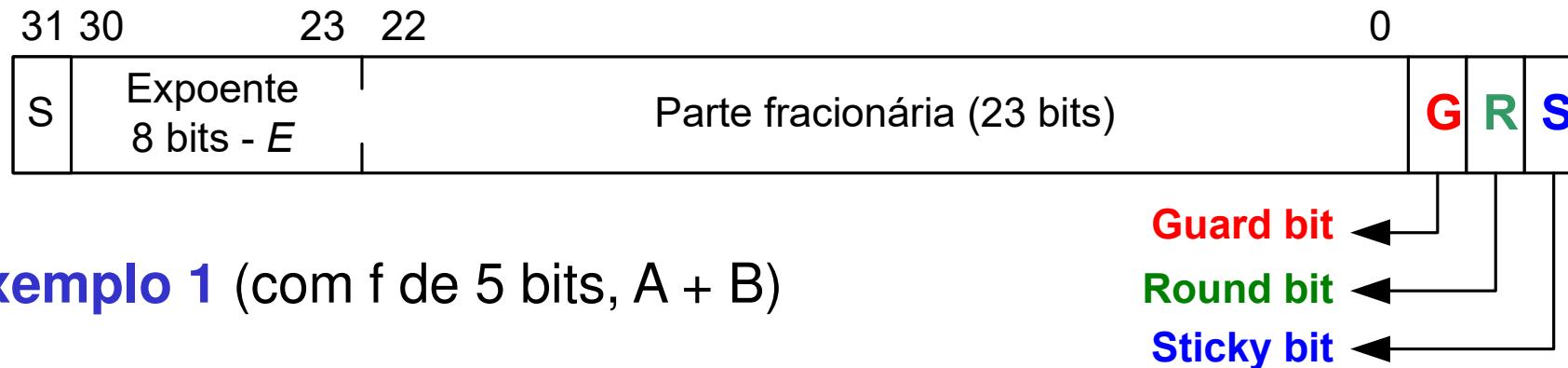
- Os valores resultantes de cada fase intermédia do cálculo de uma operação aritmética são armazenados com três bits adicionais, à direita do bit menos significativo da mantissa (i.e., para o caso de precisão simples, com pesos  $2^{-24}$ ,  $2^{-25}$  e  $2^{-26}$ )



- Objetivos: 1) ter bits suplementares para a pós-normalização e 2) minimizar o erro introduzido pelo processo de arredondamento
  - G – Guard Bit;**
  - R – Round bit**
  - S – Sticky bit** – Resultado 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

$$\begin{array}{r}
 1,11010 \\
 + 0,0100100 \\
 \hline
 10,0001100
 \end{array}$$



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

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

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

$$\begin{aligned}
 \text{Mant}(A+B) &= 1.11010 + 0.0100100 & \text{Expoente}(A+B) &= 0 \\
 &= 10.00011 \text{ } \underset{\text{G}}{0} \underset{\text{R}}{0} \underset{\text{S}}{0} & \text{G} = 0, \text{R} = 0, \text{S} = 0
 \end{aligned}$$

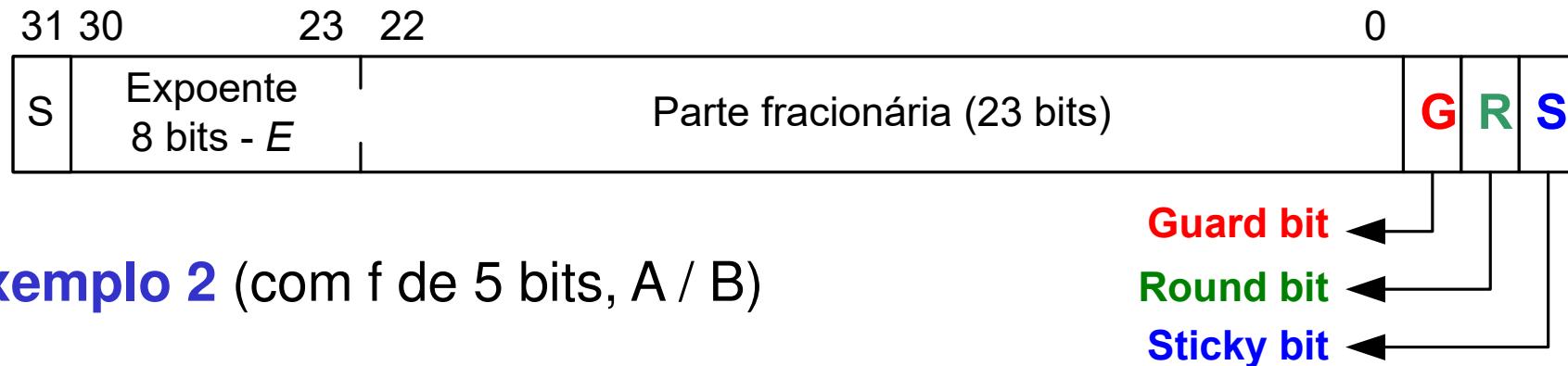
$$\text{Mant}(A+B)_{\text{norm}} = 1.00001 \text{ } \underset{\text{G}}{1} \underset{\text{R}}{1} \underset{\text{S}}{0} \quad \text{G} = 1, \text{R} = 0, \text{S} = 0 \quad \text{Expoente}(A+B) = 1$$

**Arredondamento:**

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

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

# Norma IEEE 754 – arredondamentos



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

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

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

$$= 0.10000 \textcolor{red}{1} \textcolor{green}{1} \textcolor{blue}{0} 0001 \quad \mathbf{G = 1, R = 1, S = OR(00001) = 1}$$

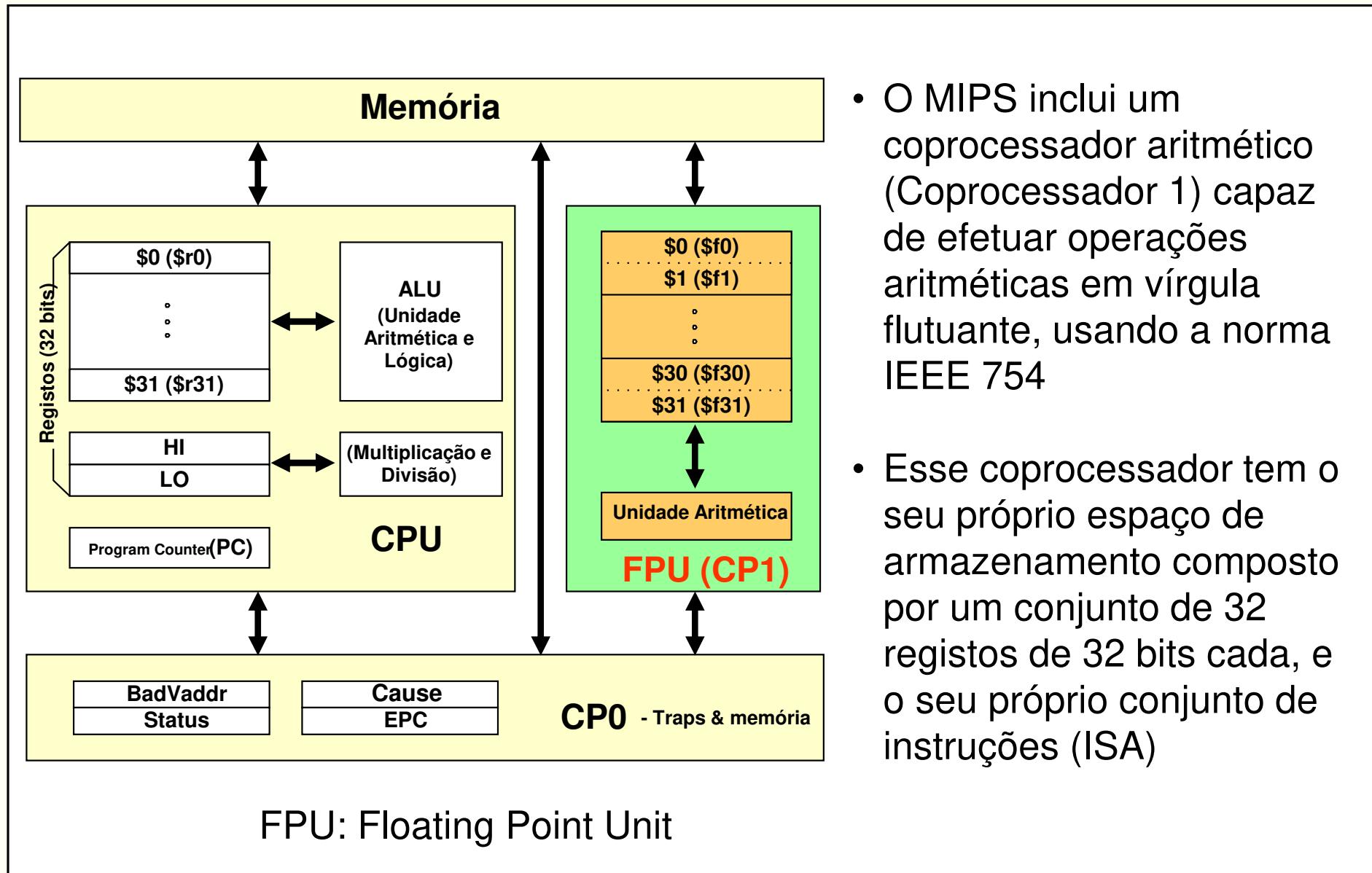
$$= 0.10000 \textcolor{red}{1} \textcolor{green}{1} \textcolor{blue}{1}$$

$$\text{Mant}(A/B)_{\text{norm}} = 1.0000 \boxed{1 \textcolor{red}{1} \textcolor{green}{0}} \quad \boxed{\text{Arred}(1,11_2) = 10_2} \quad \text{Expoente}(A/B) = 2$$

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

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

# Cálculo em Vírgula Flutuante no MIPS



# Vírgula Flutuante no MIPS – registros

- Os registos do coprocessador 1 são designados por **\$fn**, em que o índice **n** toma valores entre 0 e 31 (\$f0, \$f1, \$f2, ...)
- 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**.
- A referência ao conjunto de 2 registos faz-se sempre indicando como operando o **registro par** (\$f0, \$f2, \$f4,...)
- **Apenas os registos de índice par podem ser usados no contexto das instruções**

# Vírgula Flutuante no MIPS – instruções aritméticas

<b>abs.p</b>	<b>FPdst , FPs<sub>rc</sub></b>	<b>#Absolute Value</b>
<b>neg.p</b>	<b>FPdst , FPs<sub>rc</sub></b>	<b>#Negate</b>
<b>div.p</b>	<b>FPdst , FPs<sub>rc1</sub> , FPs<sub>rc2</sub></b>	<b>#Divide</b>
<b>mul.p</b>	<b>FPdst , FPs<sub>rc1</sub> , FPs<sub>rc2</sub></b>	<b>#Multiply</b>
<b>add.p</b>	<b>FPdst , FPs<sub>rc1</sub> , FPs<sub>rc2</sub></b>	<b>#Addition</b>
<b>sub.p</b>	<b>FPdst , FPs<sub>rc1</sub> , FPs<sub>rc2</sub></b>	<b>#Subtract</b>

- O sufixo **.p** representa a precisão com que é efetuada a operação (simples ou dupla); na instrução é substituído pelas letras **.s** ou **.d** respetivamente
- Exemplos:
  - **add.s \$f0 , \$f4 , \$f6 #** $f0 = f4 + f6$
  - **div.d \$f4 , \$f0 , \$f8 #** $f4(f5) = f0(f1) / f8(f9)$

# Vírgula Flutuante no MIPS – conversão entre tipos

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

- A letra mais à direita especifica o formato original; a letra do meio, especifica o formato do resultado - **s**: float (single), **d**: double, **w**: inteiro
- As **conversões** entre tipos de representação são efetuadas pela FPU: **os registos operando e destino das instruções são obrigatoriamente registos da FPU**

## Conversão entre tipos – exemplos

```
$f0=0xC0D00000 = 11000000110100000000000000000000  
= 11000000110100000000000000000000  
= -1.625 x 22 = -6.5
```

```
cvt.d.s $f6,$f0 #Convert Float to Double  
E = (129-127) + 1023 = 1025 = 10000000012,  
$f6=0x00000000 $f7=1 1000000001 101000...0  
$f6=0x00000000 $f7=0xC01A0000
```

```
cvt.w.s $f8,$f0 #Convert Float to Integer  
Exp = (129-127) = 2  
Val = -1.625 x 22 = -6.5  
Resultado: (int)(-6.5) = trunc(-6.5) = -6  
$f8=0xFFFFFFF8 (-6 em complemento para 2)
```

# Vírgula Flutuante no MIPS – instruções de transferência

- Transferência de informação entre regtos do CPU e da FPU, e entre regtos da FPU

	Registo do <b>CPU</b>	Registo da <b>FPU</b>	
<code>mtc1</code>	<code>CPUSrc</code>	<code>FPdst</code>	#Move <u>to</u> Coprocessor 1 #Ex: <code>mtc1 \$t0,\$f4</code>
<code>mfc1</code>	<code>CPUDst</code>	<code>FPSrc</code>	#Move <u>from</u> Coprocessor 1 #Ex: <code>mfc1 \$a0,\$f6</code>
<code>mov.s</code>	<code>FPdst</code>	<code>FPSrc</code>	#Move from FPSrc to FPdst (single) #Ex: <code>mov.s \$f4,\$f8</code>
<code>mov.d</code>	<code>FPdst</code>	<code>FPSrc</code>	#Move from FPSrc to FPdst (double) #Ex: <code>mov.d \$f2,\$f0</code>

- Estas instruções copiam o conteúdo integral do regsto fonte para o regsto destino
- **Não fazem qualquer tipo de conversão entre tipos de informação**

# Vírgula Flutuante no MIPS – instruções de transferência

- Transferência de informação entre registos da FPU e a memória

Registo da FPU	Endereço de memória	
l.s	FPdst, offset (CPUREG)	#Load Float from memory #Ex: l.s \$f0,4(\$a0)
s.s	FPsrc, offset (CPUREG)	#Store Float into memory #Ex: s.s \$f0,0(\$a0)
l.d	FPdst, offset (CPUREG)	#Load Double from memory #Ex: l.d \$f4,8(\$a1)
s.d	FPsrc, offset (CPUREG)	#Store Double into memory #Ex: s.d \$f4,16(\$t0)

Instruções nativas (só muda a mnemónica):

lwcl	FPdst, offset (CPUREG)	#Load Float from memory
swcl	FPsrc, offset (CPUREG)	#Store Float into memory
ldcl	FPdst, offset (CPUREG)	#Load Double from memory
sdcl	FPsrc, offset (CPUREG)	#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, manualmente, 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 directivas “**.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 (**l.s** ou **l.d**)

# 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

# Cálculo em Vírgula Flutuante no MIPS

- Instruções de comparação:

```
c.xx.s FPUREG1, FPUREG2 # compare float  
c.xx.d FPUREG1, FPUREG2 # compare double
```

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

**EQ** – **equal** =  
**LT** – **less than** <  
**LE** – **less or equal**  $\leq$

Exemplos:

```
c.eq.s $f0,$f2 / c.le.d $f4,$f8
```

- Instruções de salto:

```
bc1t label # branch if true  
bc1f label # branch if false
```

# Vírgula Flutuante no MIPS – instruções de decisão

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

```
# a: $f0  
# b: $f2  
...  
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:...
```

# Convenções de utilização dos registos

- Registos para **passar parâmetros** para sub-rotinas (do tipo float ou double):
  - **\$f12 (\$f13), \$f14 (\$f15)**, por esta ordem
- Registos para **devolução de resultados** das sub-rotinas:
  - **\$f0 (\$f1)**
- Registos que **podem** ser livremente usados e alterados pelas sub-rotinas ("caller-saved"):
  - **\$f0 (\$f1) a \$f18 (\$f19)**
- Registos que **não podem** ser alterados pelas sub-rotinas ("callee-saved"):
  - **\$f20 (\$f21) a \$f30 (\$f31)**

# Tradução C / Assembly – Exemplo 1

```
#define SIZE 25
double average(double *, int);

void main(void)
{
    double array[SIZE];
    double avg;
    ...
    avg = average( array, SIZE );
    print_double( avg );      // syscall 3
}
```

```
double average(double *v, int N)
{
    double sum = 0.0;
    int i;

    for(i = 0; i < N; i++)
        sum += v[i];
    return sum / (double)N;
}
```

Conversão entre tipos  
(inteiro para double)

# Tradução C / Assembly – Exemplo 1

```
void main(void)
{
    static double array[SIZE];
    double avg;
    ...
    avg = average( array, SIZE );
    print_double( avg );      // syscall 3
}
```

```
double average(double *, int)
```

```
.data
array: .space 200          # 8*SIZE (alinhado múltiplo 8)
       .eqv SIZE,25
.text
.globl main                # avg: $f12
main: ...                  # Salvaguarda $ra
       la      $a0, array   #
       li      $a1, SIZE    #
       jal     average    #
       mov.d  $f12, $f0    # avg = average(array, SIZE)
       li      $v0, 3       #
       syscall             # print_double(avg)
       ...                 # Repõe $ra
       jr      $ra          #
```

# Tradução C / Assembly – Exemplo 1

```
double average(double *v, int N)
{
    double sum = 0.0;
    int i;
    for(i = 0; i < N; i++)
        sum += v[i];
    return sum / (double)N;
}
```

```
# sum: $f0    /  tmp1: $f4    /  i: $t0    /  tmp2: $t1
average: mtc1    $0,  $f0          #
                  cvt.d.w  $f0,  $f0          #  sum = 0.0
                  li     $t0,  0           #  i = 0
for:   bge    $t0,  $a1,  endf      #  while(i < N) {
                  sll    $t1,  $t0,  3           #  tmp = i * 8
                  addu   $t1,  $t1,  $a0          #  $t1 = &v[i]
                  l.d    $f4,  0($t1)          #  $f4 = v[i]
                  add.d   $f0,  $f0,  $f4          #  sum += v[i]
                  addi    $t0,  $t0,  1           #  i++
                  j      for                   #  }
endf:  mtc1    $a1,  $f4          #
                  cvt.d.w  $f4,  $f4          #  $f4 = (double)N
                  div.d   $f0,  $f0,  $f4          #  return sum / (double)N
                  jr     $ra                   #
```

# Tradução C / Assembly – Exemplo 2

```
float fun(float, int);  
  
void main(void)  
{  
    float res;  
  
    res = fun( 12.5E-2, 2 );  
    print_float( res ); // syscall 2  
}
```

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

# Tradução C / Assembly – Exemplo 2

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

    res = fun( 12.5E-2, 2 );
    print_float( res );           // syscall 2
}
```

float fun(float a, int k)

```
.data
k1: .float 12.5E-2          # 12.5 x 10^-2
k2: .float -5.6
k3: .float 32.0
k4: .float 0.0
.text
.globl main                  # res: $f12
main: ...
    la    $t0, k1
    l.s   $f12, 0($t0)      # $f12 = 12.5E-2
    li    $a0, 2             # $a0 = 2
    jal   fun                #
    mov.s $f12, $f0          # res = fun(12.5E-2, 2)
    li    $v0, 2             #
    syscall                  # print_float(res)
    ...
    jr    $ra                # repõe $ra
```

*↳ \$t0 guarda o valor do endereço*

# Tradução C / Assembly – Exemplo 2

```

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

# val: $f2  / a: $f12  /  m: $a0

```

```

.data
k1: .float 12.5E-2
k2: .float -5.6
k3: .float 32.0
k4: .float 0.0

```

```

fun:   la      $t0, k2
         l.s    $f0, 0($t0)      # $f0 = -5.6
         c.lt.s $f12,$f0        # if( a >= -5.6 )
         bc1t  else             # {
         la      $t0, k3
         l.s    $f2, 0($t0)      #     val = 32.0
         sub.s  $f2, $f12, $f2    #     val = a - 32.0
         mtc1   $a0, $f0          #     $f0 = m
         cvt.s.w $f0, $f0          #     $f0 = (float)m
         mul.s  $f2, $f0, $f2      #     val = (float)m * val
         j      endif            # } else
else:  la      $t0, k4
         l.s    $f2, 0($t0)      #     val = 0.0
endif: mov.s  $f0, $f2        # return val;
         jr      $ra              #

```

# Exercícios

- ① • Na conversão de uma quantidade codificada em formato IEEE754 precisão simples para decimal, qual o número máximo de casas decimais com que o resultado deve ser apresentado? E se o valor original estiver representado em formato IEEE754 precisão dupla?
- ② • Determine a representação em formato IEEE754 precisão simples da quantidade real **19,1875**. Determine a representação da mesma quantidade em precisão dupla
- ③ • Determine o valor em decimal da quantidade representada em formato IEEE754, precisão simples, como **0xC19AB000**
- ④ • Determine o valor em decimal da quantidade representada em formato IEEE754, precisão simples, como **0x80580000**

# Exercícios

9

- Considere que o conteúdo dos dois seguintes regtos da FPU representam a codificação de duas quantidades reais no formato IEEE754 precisão simples:
  - **\$f0 = 0x416A0000**
  - **\$f2 = 0xC0C00000**

Calcule o resultado das instruções seguintes, apresentando o resultado em hexadecimal:

- |                  |                          |                                    |
|------------------|--------------------------|------------------------------------|
| ▪ <b>abs.s</b>   | <b>\$f4, \$f2</b>        | # <b>\$f4 = abs(\$f2)</b>          |
| ▪ <b>neg.s</b>   | <b>\$f6, \$f0</b>        | # <b>\$f6 = neg(\$f0)</b>          |
| ▪ <b>sub.s</b>   | <b>\$f8, \$f0, \$f2</b>  | # <b>\$f8 = \$f0 - \$f2</b>        |
| ▪ <b>sub.s</b>   | <b>\$f10, \$f2, \$f0</b> | # <b>\$f10 = \$f2 - \$f0</b>       |
| ▪ <b>add.s</b>   | <b>\$f12, \$f0, \$f2</b> | # <b>\$f12 = \$f0 + \$f2</b>       |
| ▪ <b>mul.s</b>   | <b>\$f14, \$f0, \$f2</b> | # <b>\$f14 = \$f0 * \$f2</b>       |
| ▪ <b>div.s</b>   | <b>\$f16, \$f0, \$f2</b> | # <b>\$f16 = \$f0 / \$f2</b>       |
| ▪ <b>div.s</b>   | <b>\$f18, \$f2, \$f0</b> | # <b>\$f18 = \$f2 / \$f0</b>       |
| ▪ <b>cvt.d.s</b> | <b>\$f20, \$f2</b>       | # <b>Convert single to double</b>  |
| ▪ <b>cvt.w.s</b> | <b>\$f22, \$f0</b>       | # <b>Convert single to integer</b> |

1

## Preciso single

Um número em IEEE 754 de preciso single tem 24 bits de preciso na mantissa.

Esses 24 bits permitem representar aproximadamente 7 a 9 casas decimais.

Por isso, um valor convertido para decimal terá um número máximo de 9 casas decimais.

## Preciso Double

Um número em IEEE 754 de preciso double possui 53 bits de preciso na mantissa.

Esses 53 bits de preciso permitem representar aproximadamente 15 a 17 casas decimais.

Logo, ao converter o valor para decimal, o número máximo de casas decimais será 17.

2

## 19,1875 IEEE 754

### Preciso single:

$$19,1875 \quad 10011,0011 \rightarrow 1, \boxed{0011\ 0011} \times 2^4$$

$$\begin{array}{ll}
 19 / 2 = 9 & R: 1 \uparrow \\
 9 / 2 = 4 & R: 1 \\
 4 / 2 = 2 & R: 0 \\
 2 / 2 = 1 & R: 0 \\
 1 / 2 = 0 & R: 1
 \end{array}$$

$$\begin{aligned}
 0,1875 \times 2 &= 0,375 \\
 0,375 \times 2 &= 0,75 \\
 0,75 \times 2 &= 1,5 \\
 0,5 \times 2 &= 1,0
 \end{aligned}$$

Sinal: 0

$$127 + 4 = \boxed{131} \rightarrow \boxed{1000\ 0011_2}$$

No formato IEEE754 0 1000 0011 0011 0011 0000 0000 0000 0000

## Preciso duplo

$$1,0011\ 0011 \times 2^4 \quad 4 + 1023 = 1027 \rightarrow 1000\ 0000\ 011$$

Sinal: 0

IEEE 754 : 0 1000 0000 011 0011 0011 0000 ... 0000  
53 bits

(3)

0xC19AB000

↓

1100 0001 1001 1010 1011 0000 0000 0000

Sinal: 1

$$E: 1000\ 0011 \rightarrow 131 - 127 = 4$$

M: 0011 0101 0110 0000 0000 000

1,0011 0101 0110

10011, 0101 0110

-19,3359375

(4)

0x80580000

1000 0000 0101 1000 0000 0000 0000 0000

Sinal: 1

E: 0000 0000

M: 1010 0000 0000 0000 0000 000

Quantidade desnormalizada

5

$0x416A\ 0000$

0100 0001 0110 1010 0000 0000 0000 0000

S.ind.: 0

$$E = 1000\ 0010 = 130 - 127 = 3$$

$$M = 1101\ 0100\ 0000\ 0000\ 0000\ 000$$

$$1,1101\ 0100 \times 2^3$$

$$1110,10100$$

14,5

$0xC0C0\ 0000$

1100 0000 1100 0000 0000 0000 0000 0000

S.ind.: 1

$$E: 1000\ 0001 = 124 - 127 = 2$$

$$M: 1000\ 0000\ 0000\ 0000\ 0000\ 000$$

$$1,1000 \times 2^2$$

$$110,00$$

-6

• abs.  $\rightarrow \$f4 = 0x40C0\ 0000$  6

• neg.  $\rightarrow \$f0 = 0x C16A\ 0000$  -14,625

• sub.  $\rightarrow 0x\ 4144\ 0000$  20

• mult.  $\rightarrow 0x\ C1A4\ 0000$  -20

• add.  $\rightarrow 0x\ 4108\ 0000$  8,5

• mult.  $\rightarrow 0x\ C2AF\ 0000$  -87

• div.  $\rightarrow 0x\ C01A\ 6666$  -2,41666...

• div.  $\rightarrow 0x\ BE D29E8C$  -0,4137931 ...

• cvt. d.  $\rightarrow 0x\ C018\ 0000\ 0000\ 0000$  -6

• cvt. w.  $\rightarrow 0x\ 0000\ 000E$  14