

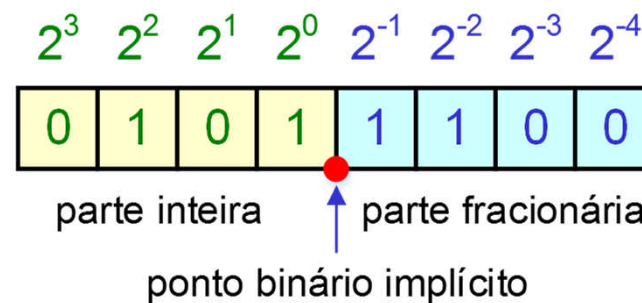
Aulas 12 e 13

- 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 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
- 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:

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

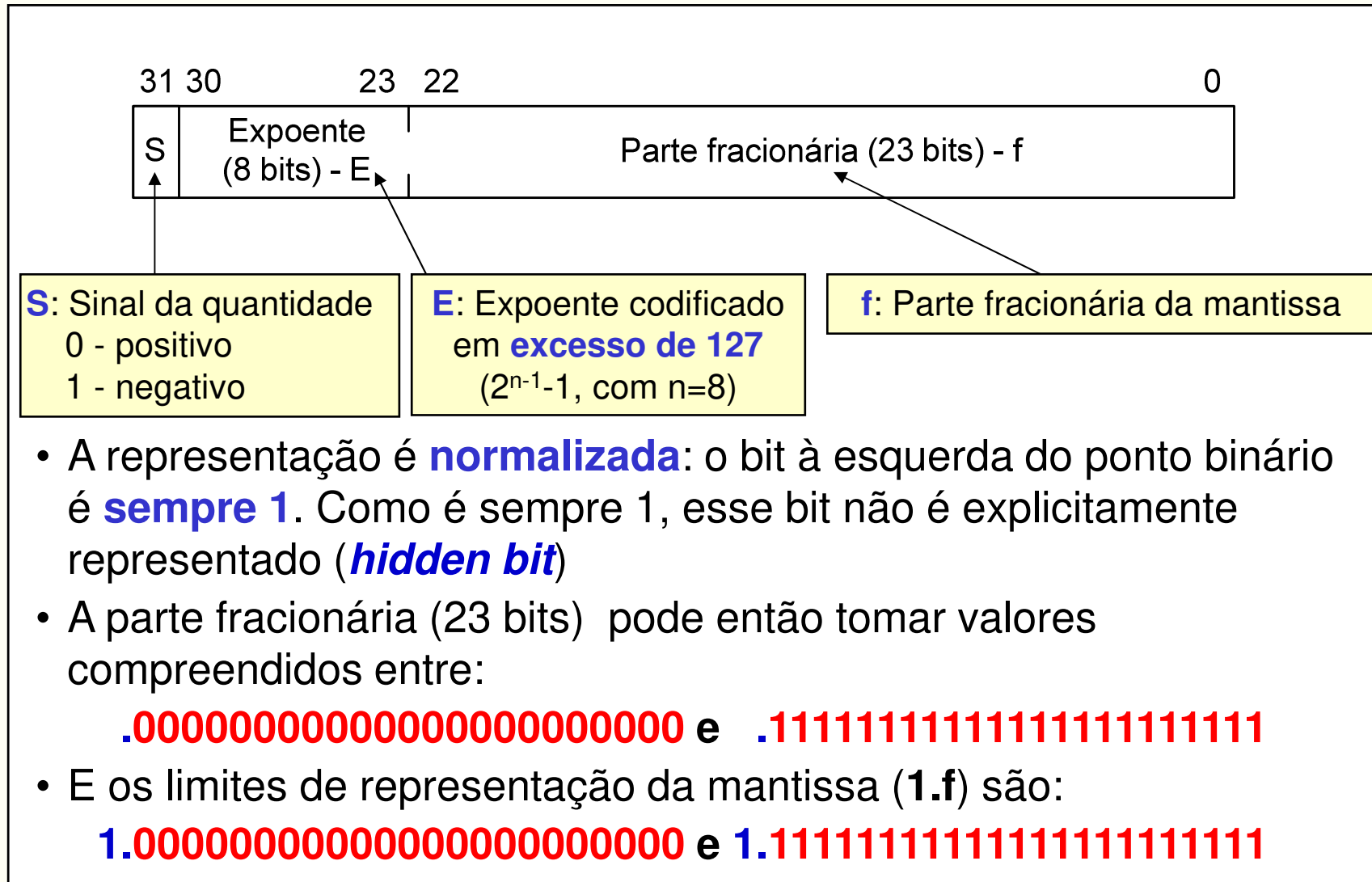
(representação em binário de uma quantidade real, no formato de **vírgula flutuante normalizada**)

- 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)

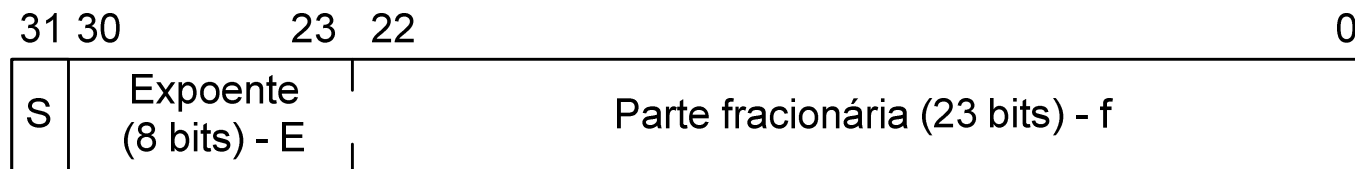


- O expoente é codificado em **excesso de 127** ($2^{n-1}-1$, $n=8$ bits). Ou seja, é somado ao expoente verdadeiro (**Exp**) o valor 127 para obter o código de representação (i.e. **$E = \text{Exp} + 127$** , em que E é o expoente codificado)

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

- O código 127 representa, assim, o expoente zero; códigos maiores do que 127 representam expoentes positivos e códigos menores que 127 representam expoentes negativos
- **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 101100000000000000000000

Sinal = 0 (quantidade positiva)

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

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 $-12593.75_{10} \times 10^{-3}$

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

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

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

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

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

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

1 **10000010** **100100110000000000000000**

0xC1498000

	0.59375
	× 2
MSb	1 .18750
	0.18750
	× 2
	0 .37500
	0.37500
	× 2
	0 .75000
	0.75000
	× 2
	1 .50000
	0.50000
	× 2
LSb	1 .00000

Norma IEEE 754 (precisão simples)

- 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 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:

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

- 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 n° 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 → **$E > 254$**)

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

- Underflow**: caso em que o expoente é tão pequeno que também não é representável → **$E < 1$**)

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

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)

$$N = 1.0000\underline{110} \times 2^1 = 1.0001 \times 2^1$$

1.0000	11
+ 0.0000	10
<hr/>	
1.0001	01

Exemplo com 4 bits fracionários

1.1 101
0.0 10010
10.0 0011 0

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
<hr/>	
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

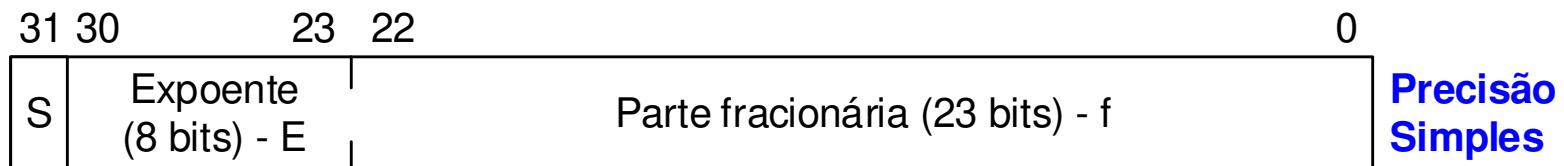
$$N = 1.1 \times 2^1 = 1.1000 \times 2^1$$

1.1000		0
+ 0.0000		1
<hr/>		
1.1000		1

Exemplo com 4 bits fracionários

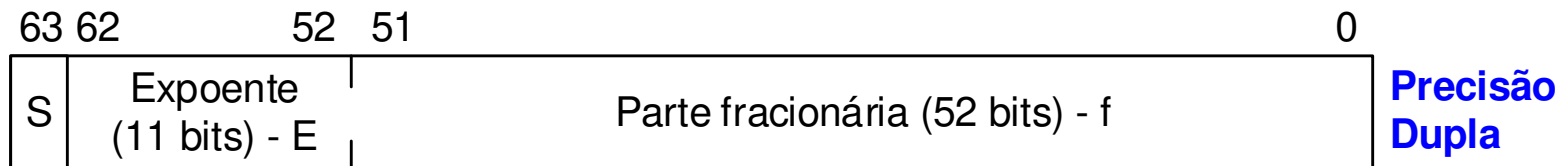
Norma IEEE 754 (precisão dupla)

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



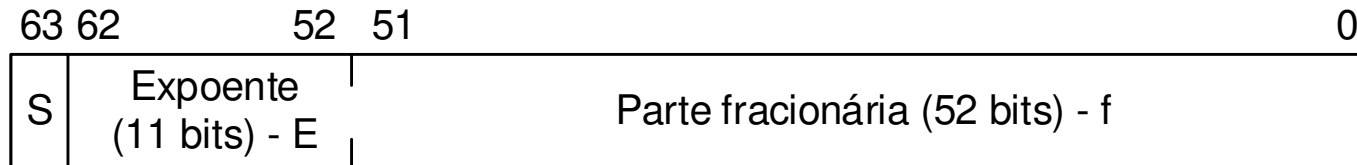
$$N = (-1)^S 1.f \times 2^{(E - 127)} \quad (\text{Precisão simples - tipo float})$$

- e em **precisão dupla (64 bits)**



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

Norma IEEE 754 (precisão dupla)



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

- 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:

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

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

- 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 , $\text{nan} * 2$
 - 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

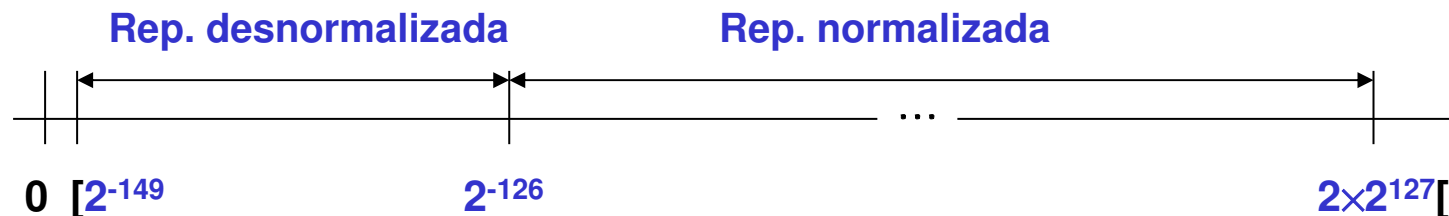
Precisão Simples		Precisão Dupla		Representa
Expoente	Parte Frac.	Expoente	Parte Frac.	
0	0	0	0	0
0	$\neq 0$	0	$\neq 0$	Quantidade desnormalizada
<i>1 a 254</i>	<i>qualquer</i>	<i>1 a 2046</i>	<i>qualquer</i>	<i>Nº em vírgula flutuante normalizado</i>
255	0	2047	0	Infinito
255	$\neq 0$	2047	$\neq 0$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

Norma IEEE 754 – representação desnormalizada

- Representação com mantissa desnormalizada: assume-se que o bit à esquerda do ponto binário é 0
- O expoente codificado 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:

[illegible]

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



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

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

$$= (0 - 1/4 + 1/2 + 1/4) / 4$$

$$= +1/8$$

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

$$\begin{array}{r} x.00 \\ + 0.1 \\ \hline x.10 \end{array}$$

$$\begin{array}{r} x.01 \\ + 0.1 \\ \hline x.11 \end{array}$$

$$\begin{array}{r} x.10 \\ + 0.1 \\ \hline x+1.00 \end{array}$$

$$\begin{array}{r} x.11 \\ + 0.1 \\ \hline x+1.01 \end{array}$$

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

$$\begin{array}{r}
 x0.10 \\
 + 0.0 \\
 \hline
 x0.10
 \end{array}$$

$$\begin{array}{r}
 x1.10 \\
 + 0.1 \\
 \hline
 x1 + 1.00
 \end{array}$$

- **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

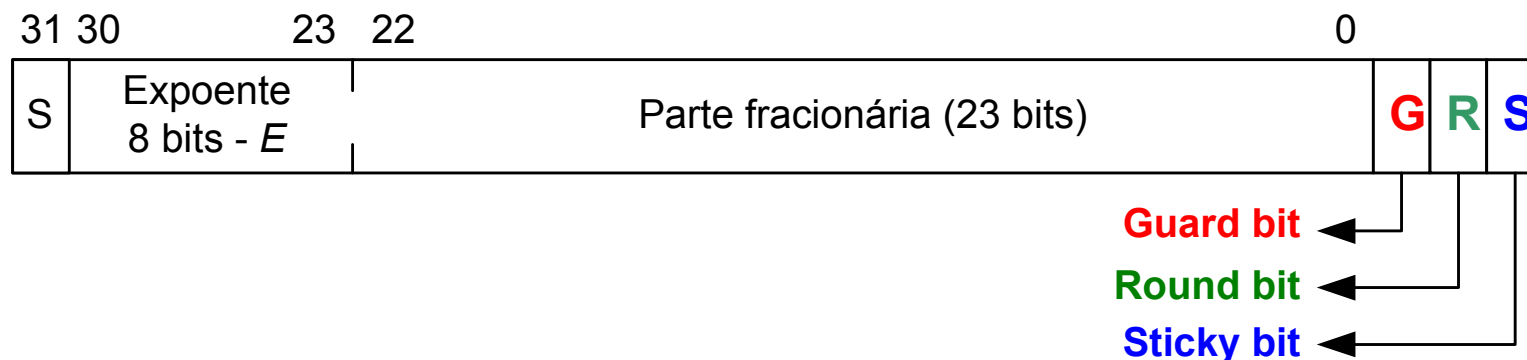
O que fica à direita de b_{23}	Exemplo	Resultado
< 0.5	$1.b_1b_2 \dots b_{22}b_{23} \text{ 011}$	<i>Round down</i> : bits à direita de b_{23} são descartados
> 0.5	$1.b_1b_2 \dots b_{22}b_{23} \text{ 101}$	<i>Round up</i> : soma-se 1 a b_{23} (propagando o <i>carry</i>)
$= 0.5$	$1.b_1b_2 \dots b_{22} \text{ 1 100}$	<i>Round up</i> : soma-se 1 a b_{23} (propagando o <i>carry</i>) (*)
$= 0.5$	$1.b_1b_2 \dots b_{22} \text{ 0 100}$	<i>Round down</i> : bits à direita de b_{23} são descartados (*)
$= 0.5$	$1.b_1b_2 \dots b_{22} \text{ 1 100}$	<i>Round down</i> : bits à direita de b_{23} são descartados (**)
$= 0.5$	$1.b_1b_2 \dots b_{22} \text{ 0 100}$	<i>Round up</i> : soma-se 1 a b_{23} (propagando o <i>carry</i>) (**)

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

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

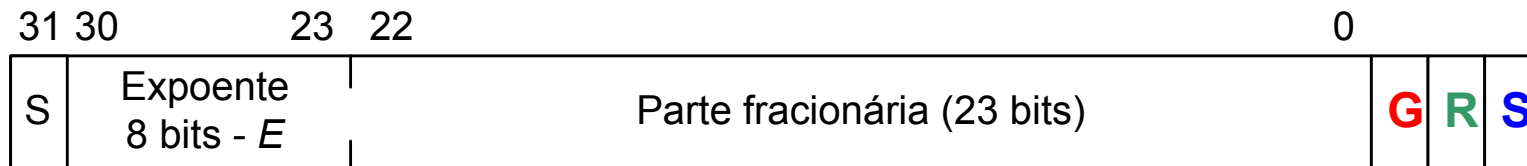
Norma IEEE 754 – arredondamentos

- 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



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

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

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

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

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

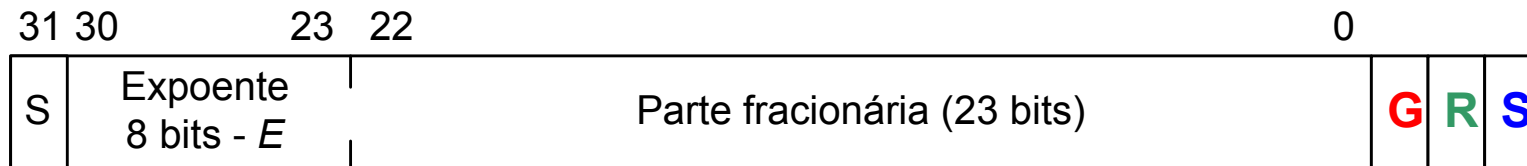
$$\text{Mant}(A+B)_{\text{norm}} = 1.00001 \text{ 100 } G=1, R=0, S=0 \quad \text{Expoente}(A+B) = 1$$

Arredondamento:

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

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

Norma IEEE 754 – arredondamentos



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

Guard bit

Round bit

Sticky bit

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

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

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

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

$$\text{Mant}(A/B)_{\text{norm}} = 1.0000 \mathbf{1} \mathbf{1} 10$$

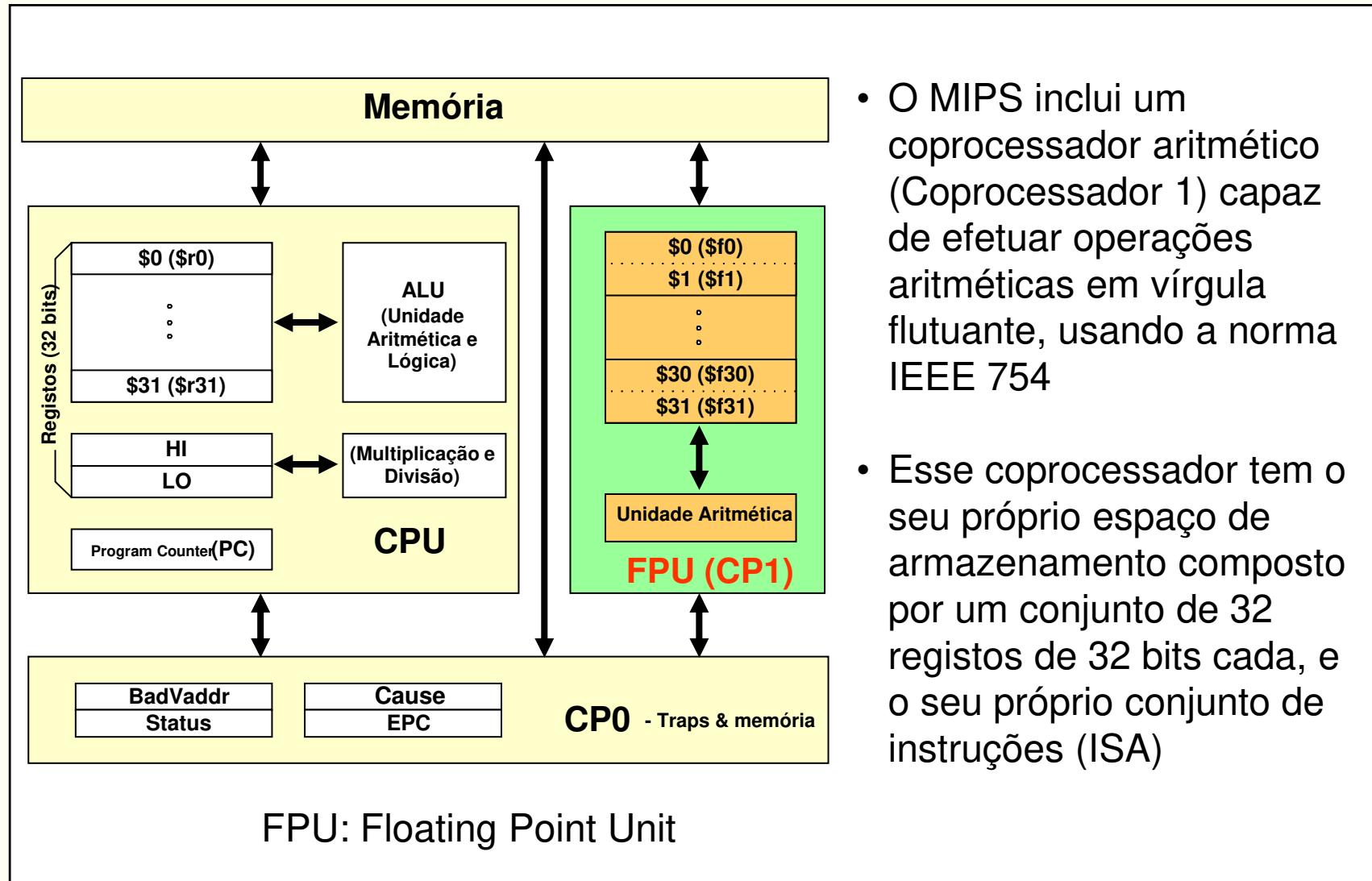
$$\text{Arred}(1, 11_2) = 10_2$$

$$\text{Expoente}(A/B) = 2$$

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 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 **registo par** (\$f0, \$f2, \$f4,...)
- **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); 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

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

- 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 = 11000000011010000000000000000000  
                  = 11000000011010000000000000000000  
                  = -1.625 x 22 = -6.5
```

```
cvt.d.s $f6,$f0 #Convert Float to Double
```

$$\mathbf{E} = (129-127) + 1023 = 1025 = 10000000001_2$$

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

```
$f6=0x00000000 $f7=0xC01A0000
```

```
cvt.w.s $f8,$f0 #Convert Float to Integer
```

$$\mathbf{Exp} = (129-127) = 2$$
$$\mathbf{Val} = -1.625 \times 2^2 = -6.5$$

Resultado: `(int) (-6.5) = trunc(-6.5) = -6`

\$f8=0xFFFFFFFFA (-6 em complemento para 2)

Vírgula Flutuante no MIPS – instruções de transferência

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

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

- Estas instruções copiam o conteúdo integral do registo fonte para o registo 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):

lwc1	FPdst , offset (CPUreg)	#Load Float from memory
swc1	FPsrc , offset (CPUreg)	#Store Float into memory
ldc1	FPdst , offset (CPUreg)	#Load Double from memory
sdc1	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 – Manipulação de constantes

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

l.s **FPdst**, **label** **#Ex:** **l.s** \$f0, K1

l.d **FPdst**, **label** **#Ex:** **l.d** \$f4, K1

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 K1 corresponde ao endereço **0x10010008**):

```
l.s    $f0, k1
      lui $1, 0x1001
      l.s $f0, 0x0008 ($1)
```

```
l.d    $f4, k1
      lui $1, 0x1001
      l.d $f4, 0x0008 ($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

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

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

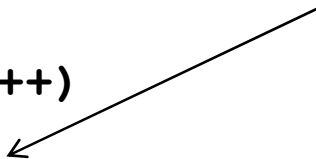
```
#define SIZE 25
double average(double *, int);

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:  ...                  # salvaguarda $ra
      l.s    $f12, k1        # $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)
      ...                  # repõe $ra
      jr     $ra             #
```

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;
}
```

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

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

```
fun:  l.s      $f0, k2           # $f0 = -5.6
      c.lt.s   $f12, $f0        # if( a >= -5.6 )
      bclt    else             # {
      l.s     $f2, k3           #     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:  l.s     $f2, k4           #     val = 0.0
endif: mov.s  $f0, $f2         # return val;
      jr      $ra              #
```

Exercícios

- 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?
- Determine a representação em formato IEEE754 precisão simples da quantidade real **19,1875**. Determine a representação da mesma quantidade em precisão dupla
- Determine o valor em decimal da quantidade representada em formato IEEE754, precisão simples, como **0xC19AB000**
- Determine o valor em decimal da quantidade representada em formato IEEE754, precisão simples, como **0x80580000**

Exercícios

- Considere que o conteúdo dos dois seguintes registros 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:

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