

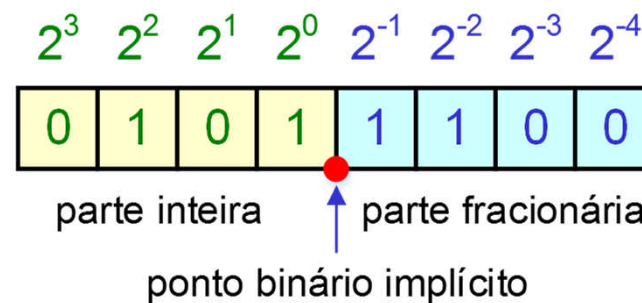
Aulas 25 e 26

- 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

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
- 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:

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

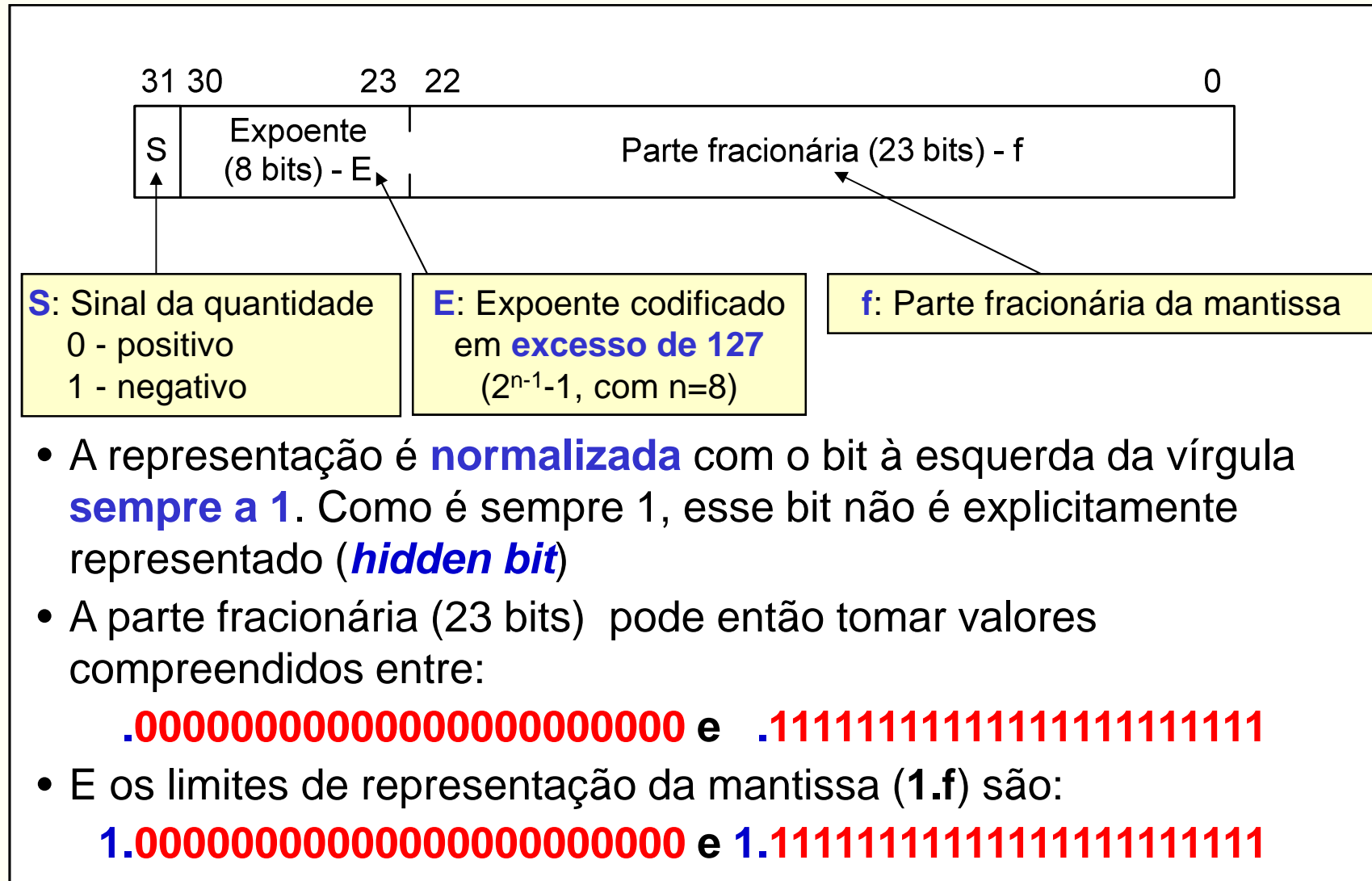
(representação **normalizada** de uma quantidade binária em vírgula flutuante)

- 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 10100000000000000000000 (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)

- **Exemplo:** codificar no formato vírgula flutuante IEEE 754 precisão simples, o valor **-12.59375₁₀**

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 (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 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)

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

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)

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

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

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

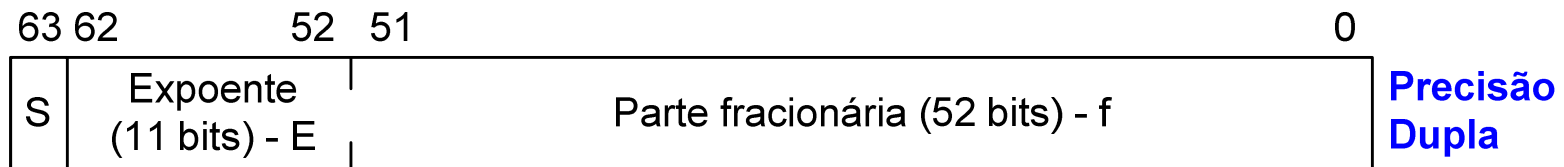
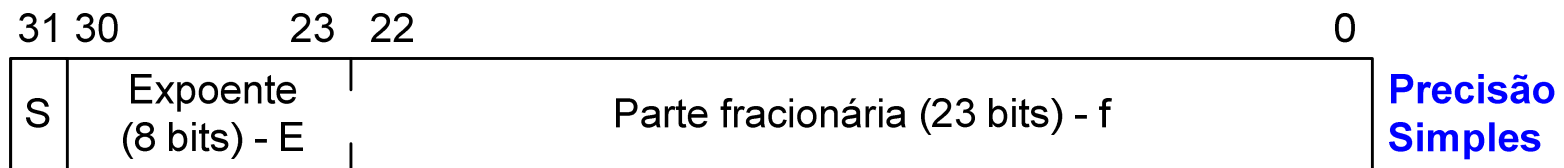
4º Passo: Arredondar o resultado

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

Exemplo com 4 bits fracionários

Norma IEEE 754 (precisão dupla)

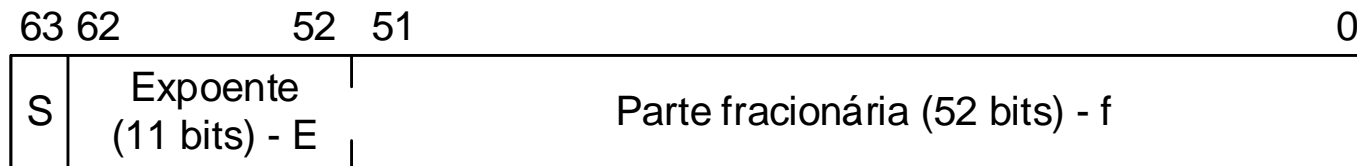
- A norma IEEE 754 suporta a representação de quantidades em **precisão simples (32 bits)** e em **precisão dupla (64 bits)**



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

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

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:

$$\pm [1.000000000000000000000000 \dots 000 \times 2^{-1022}, 1.111111111111111111111111 \dots 111 \times 2^{+1023}]$$

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

- 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
1 a 254	qualquer	1 a 2046	qualquer	Nº em vírgula flutuante normalizado
255	0	2047	0	Infinito
255	$\neq 0$	2047	$\neq 0$	NaN (Not a Number)

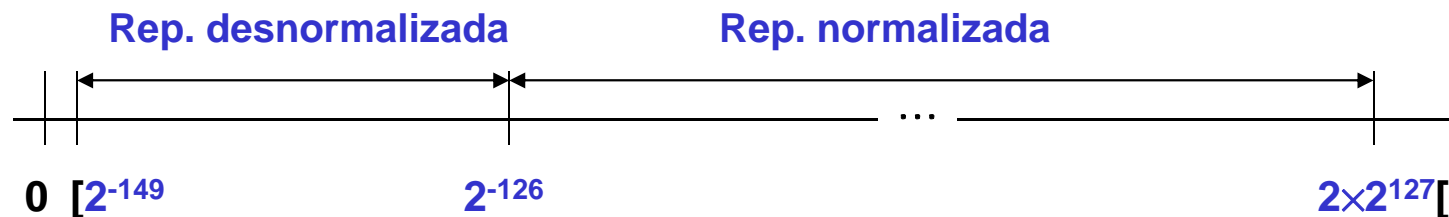
Norma IEEE 754 – representação desnormalizada

- 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 então:

[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 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 à direita do ponto binário for 0 ou soma-se “1” à parte inteira quando aquele for “1”:

$$\text{arred}(x) = \text{trunc}(x + 0.5)$$

- 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 2 dígitos na parte fracionária: $d=2$)

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
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, mas decidindo, para o caso “**xx.10**”, em função do primeiro dígito à esquerda do ponto binário
- **Erro médio** = $(0 - 1/4 - 1/2 + 1/4) / 4 + (0 - 1/4 + 1/2 + 1/4) / 4$
= $-1/8 + 1/8 = 0$

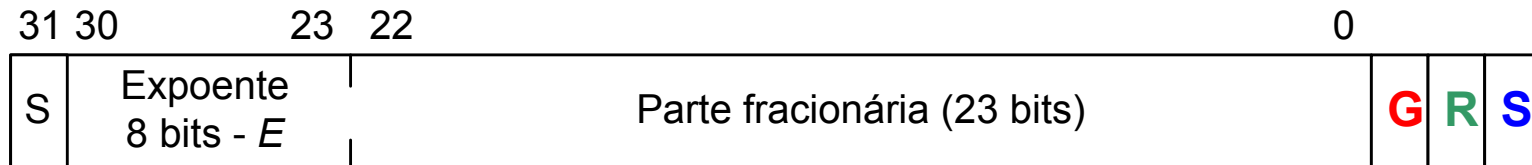
Norma IEEE 754 – arredondamentos

- 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, $A + B$)

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

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

Mant(A+B) = 1.11010 + 0.0100100 Expoente(A+B) = 0

= 10.00011 000 G = 0, R = 0, S = 0

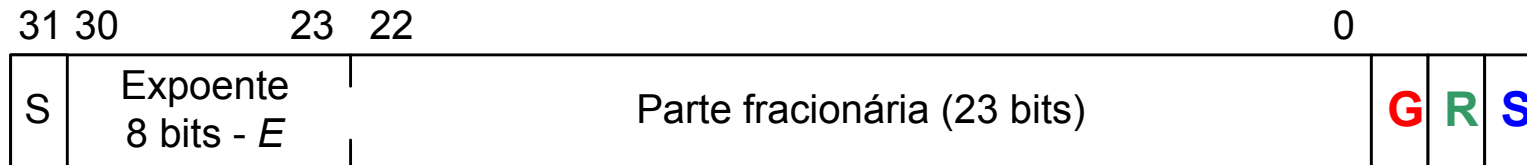
Mant(A+B)_{norm} = 1.00001 100 G = 1, R = 0, S = 0

Arredondamento:

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

Mant(A + B) = 1.00001, 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

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

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

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

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

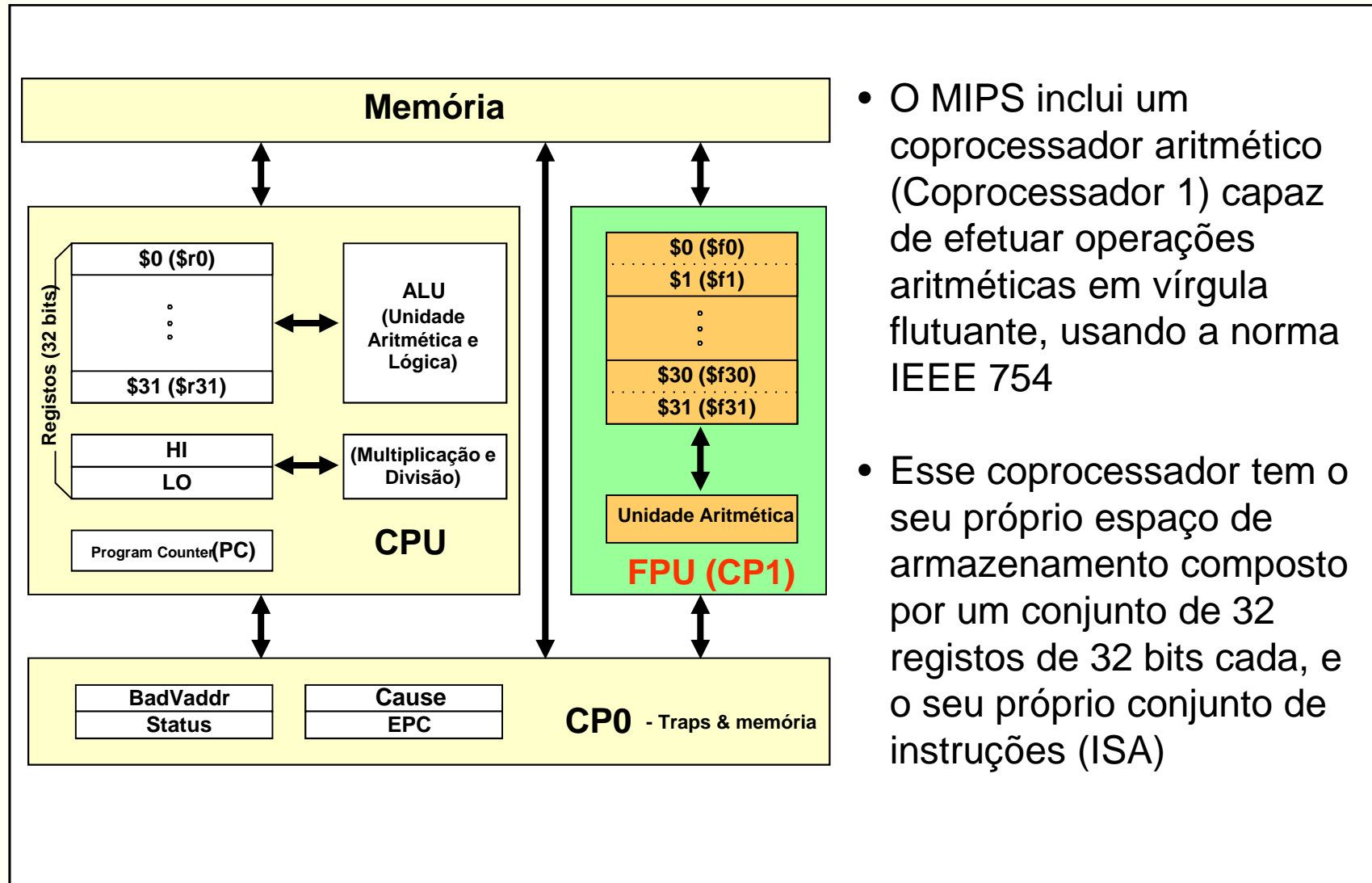
$$\text{Mant}(A/B)_{\text{norm}} = 1.0000 \mathbf{1} \mathbf{1} 11$$

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

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

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

Cálculo em Vírgula Flutuante no MIPS



- O MIPS inclui um coprocessador aritmético (Coprocessador 1) capaz de efetuar operações aritméticas em vírgula flutuante, usando a norma IEEE 754
- Esse coprocessador tem o seu próprio espaço de armazenamento composto por um conjunto de 32 registros de 32 bits cada, e o seu próprio conjunto de instruções (ISA)

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

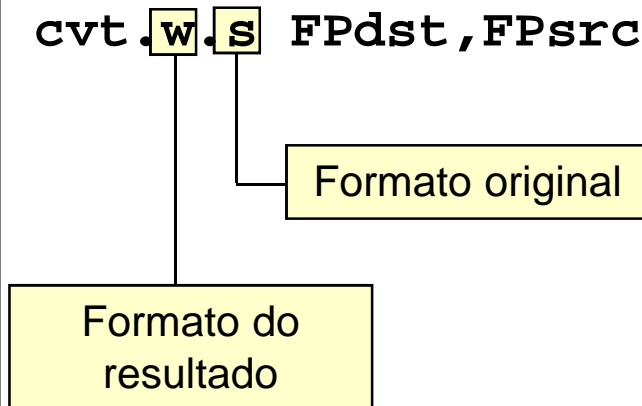
Vírgula Flutuante no MIPS – instruções aritméticas

<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

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

```
cvt.d.s $f6,$f0
```

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

```
$f6=0x00000000 $f7=1 10000000001 1010000...0
```

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

```
cvt.w.s $f8,$f0
```

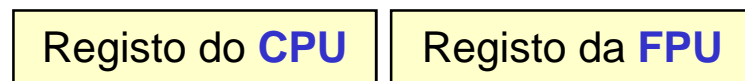
```
Exp = (129-127) = 2
```

```
Val = -1.625 x 22 = -6.5, (int)(-6.5) = -6
```

```
$f8=0xFFFFF000
```


Vírgula Flutuante no MIPS – instruções de transferência

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



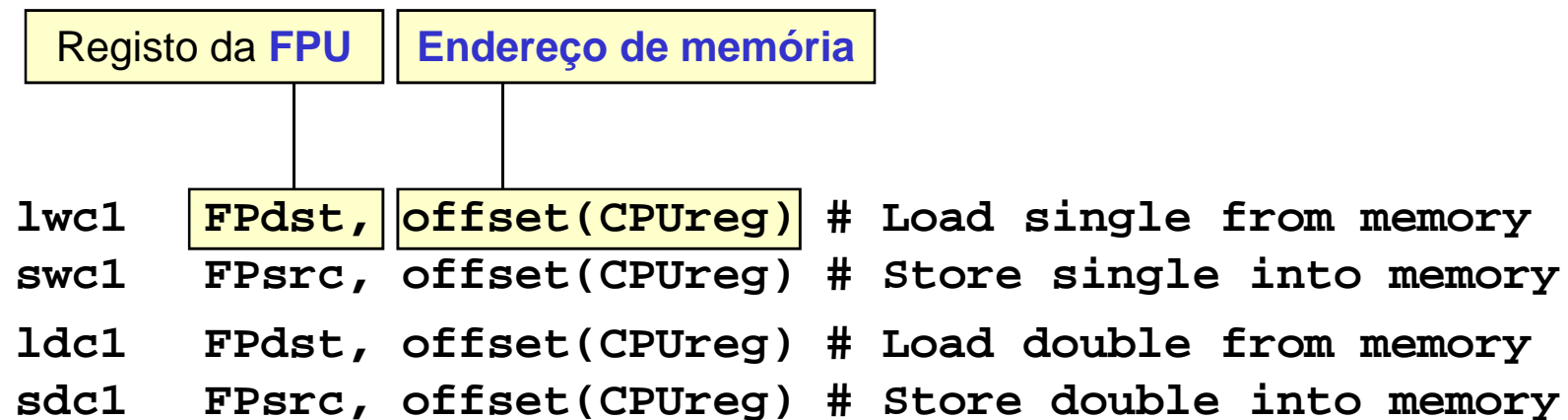
`mtc1 CPUsrc, FPdst` # Move to Coprocessor 1
`mfc1 CPUdst, FPsrc` # Move from Coprocessor 1
`mov.s FPdst, FPsrc` # Move from FPsrc to FPdst (single)
`mov.d FPdst, FPsrc` # Move from FPsrc to FPdst (double)

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



Instruções virtuais (apenas muda a mnemónica):

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

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.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`

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:...
```

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)`

Cálculo em Vírgula Flutuante no MIPS – Exemplo

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

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

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

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

Tradução C / Assembly

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

double average(double *, int)

```
        .data
array:   .space 8 * SIZE
        .text
        .globl main
main:    ...                # void main(void) {
        la      $a0, array  #
        li      $a1, SIZE   #
        jal     average     #
        mov.d   $f12, $f0    #      avg = average(array, SIZE)
        li      $v0, 3       #
        syscall                #      print_double(avg)
        ...
        jr      $ra          # }
```


Tradução C / Assembly

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

```
# $f0 ← av
func: mtc1    $0,    $f0           #
      cvt.d.s $f0, $f0           # av = 0.0
      li     $t0, 0              # i = 0
for:   bge    $t0, $a1, endif      # while(i < N) {
      sll    $t1, $t0, 3          #
      addu   $t1, $t1, $a0        # $t1 = &v[i]
      l.d    $f4, 0($t1)         # $f4 = v[i]
      add.d   $f0, $f0, $f4       # av += v[i]
      addi   $t0, $t0, 1         # i++
      j      for                 # }
endif: mtc1    $a1, $f4           #
      cvt.d.w $f4, $f4           # $f4 = (double)N
      div.d   $f0, $f0, $f4       #
      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 registos 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 |