> “A principal diferença entre algo que pode dar errado e algo que não pode dar errado de jeito nenhum é que, quando algo que não pode dar errado de jeito nenhum dá errado, geralmente acaba sendo impossível de consertar” (Douglas Adams - O Guia do Mochileiro das Galáxias).

# Ponto Flutuante e IEEE 754

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR

# Números reais e o Hardware

Faça um programa que soma 0.1 10x e mostra o resultado.

```c
#include <stdio.h>

int main() {
   double val = 0.0;
   for (int i = 0; i < 10; i++)
      val += 0.1;

   printf("%.16lf\n", val);
   return 0;
}
```

# Números reais e o Hardware

Faça um programa que soma 0.1 10x e mostra o resultado.

É impossível representar qualquer número real na máquina.

Por quê?

# Para relembrar...

Converter $4{,}1_{10}$ para binário.

# Para relembrar...

Converter $4,1_{10}$ para binário.

1. Converter a parte inteira para binário com divisões sucessivas.

   $4_{10} = 100_2$
2. Converter a parte fracionária usando múltiplas multiplicações.
   a. $0,1_{10} = 00011001100\ldots_2$

$4,1_{10} = 100,00011001100\ldots_2$

Entrada: $r_{10}$, entre 0 e 1
Saída: $r_2$ representado por $((0, d_1), d_2, \ldots, d_j)$

1: $k = 1, F = r_{10}$
2: **Faça:**
3: $\quad F = 2 \times F$
4: $\quad d_k = parteInteira(F)$
5: $\quad F = F - d_k$
6: $\quad k = k + 1$
7: **Enquanto** $(F > 0)$

# Números reais e o Hardware

É impossível representar qualquer número real na máquina.

Temos uma infinidade de números reais, e o hardware é finito.

Armazenamos **aproximações**.

# Ponto Flutuante

Uma forma de armazenar essas aproximações é através de ponto flutuante.

Conceito implementado em grande parte dos processadores comerciais.

# Ponto Flutuante

Uma forma de armazenar essas aproximações é através de ponto flutuante.

Conceito implementado em grande parte dos processadores comerciais.

Similar à notação científica normalizada.

**O número tem um e somente um dígito antes da casa decimal, e não possui zeros antes da casa decimal.**

Exemplos:

$8.0_{10} \times 10^{-9}$ é normalizado.

$0.1_{10} \times 10^{-8}$ não é normalizado.

$10.0_{10} \times 10^{-10}$ não é normalizado.

# Ponto Flutuante

Podemos fazer o mesmo com números binários.

Vamos exibir a base e a potência na base 10 para simplificar a visualização.

Exemplo:

$1.0_2 \times 2^1$ está normalizado.

Vamos chamar o ponto decimal de “ponto binário” para a base 2.

Convenção de Patterson e Hennessy (2020).

Exemplo: normalizar $1111.11_2$.

# Normalizando

Normalizar o valor $1111.11_2$

$1111.11_2 = 1111.11_2 \times 2^0$

$= 111.111_2 \times 2^1$

$= 11.1111_2 \times 2^2$

$= 1.11111_2 \times 2^3 \leftarrow$ Normalizado

O número tem um, e somente um, bit antes do ponto, e não possui zeros antes do ponto.

# Faça você mesmo

Normalize os seguintes valores binários:

a. $11.01_2$
b. $111_2$
c. $0.000001_2$

# Faça você mesmo

Normalize os seguintes valores binários:

a. $11.01_2$ $1.101_2 \times 2^1$
b. $111_2$ $1.11_2 \times 2^2$
c. $0.000001_2$ $1.0_2 \times 2^{-6}$

# Pergunta

Para armazenar um valor binário normalizado arbitrário na memória, como $1.11111_2 \times 2^3$ , quais campos são importantes?

# Ponto Flutuante

Para armazenar um valor binário normalizado arbitrário na memória, como $1.11111_2 \times 2^3$ , quais campos são importantes?

**Não precisamos armazenar** o 1 antes do ponto binário, nem a base.

Os valores então **sempre** tem o formato (+/-)1.xxxxxxxxx × $2^{yyyy}$.

Onde xxxxxxxxx é a **mantissa** (ou fração) e yyyy é o **expoente**.

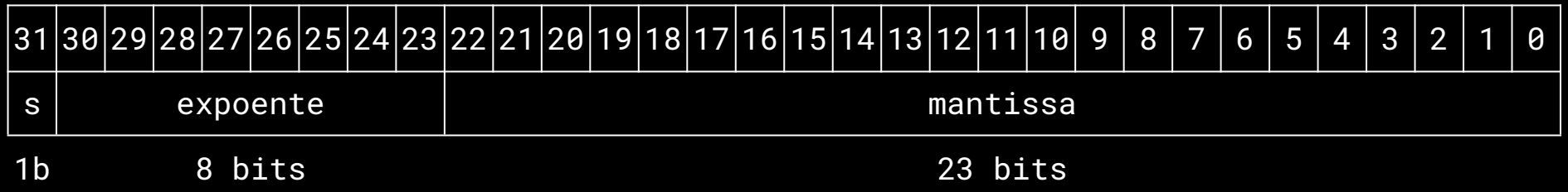
Armazenamos apenas a **mantissa**, o **expoente**, e o **sinal**.

# IEEE 754

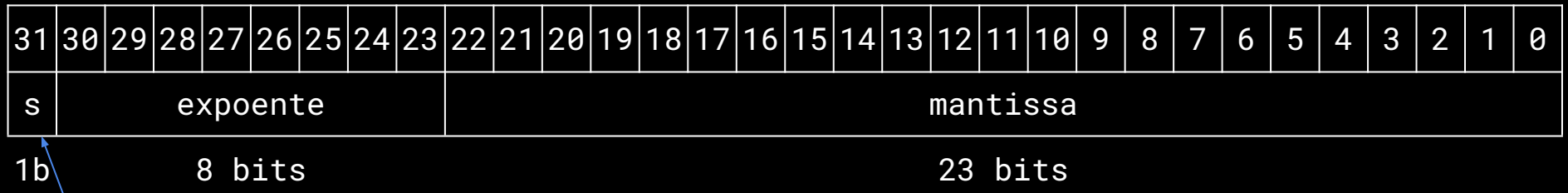
Utilizado em grande parte dos processadores comerciais.

- Inclusive no seu x86-64 e no seu smartphone.
- Quando não disponível no hardware, é simulado via software.
  - Compilador injeta as instruções necessárias.

# IEEE 754 - Precisão Simples



# IEEE 754 - Precisão Simples




Sinal da mantissa.
Representação com **sinal e magnitude**.
0 é positivo, 1 é negativo.

Precisão Simples - Declarado como *float* em C.

# Overflow e Underflow

**Overflow:** o expoente é muito grande para caber na memória.

**Underflow:** o expoente é muito pequeno para caber na memória.

# IEEE 754 – Precisão Dupla

IEEE 754 - Precisão **dupla.**

**52 bits para mantissa, e 11 para expoente.**

Declarado como **double** em C.

Quais as vantagens e desvantagens da precisão dupla em relação à simples?

# IEEE 754 – Precisão Dupla

Quais as vantagens e desvantagens da precisão dupla em relação à simples?

+ Consegue armazenar uma extensão maior de valores;

+ Maior precisão devida a mantissa ser muito maior;

- Ocupa mais memória;

- Pode precisar de mais ciclos do processador para efetuar cálculos.

# Valores Especiais

Como representar o número 0? Qual a dificuldade?

# Valores Especiais

Como representar o número 0? Qual a dificuldade?

> Concordamos que o **1 antes do ponto binário é implícito**, e não é representado pelo hardware
>
> $(+/-)1.xxxxxxxxx \times 2^{yyyy}$.
>
> Mas o número 0 em especial não tem 1 antes do ponto binário!

Essas e outras exceções são tratadas com **valores especiais**.

# Valores Especiais

| Precisão Simples | | Precisão Dupla | | Descrição |
|---|---|---|---|---|
| Expoente | Mantissa | Expoente | Mantissa | |
| 0 | 0 | 0 | 0 | 0 |
| 0 | Diferente de Zero | 0 | Diferente de Zero | +/- número desnormalizados |
| [1..254] | Qualquer | [1..2046] | Qualquer | +/- número em ponto flutuante |
| 255 | 0 | 2047 | 0 | +/- infinito |
| 255 | Diferente de Zero | 2047 | Diferente de Zero | NaN (*Not a Number*) |

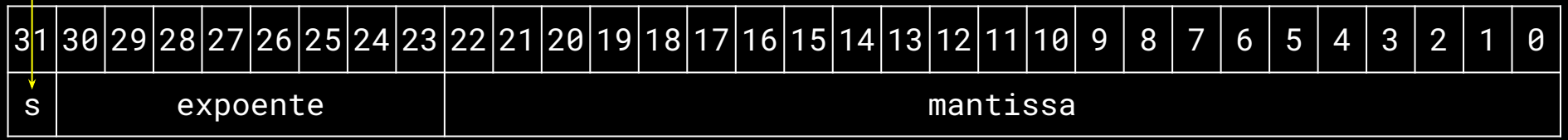
# IEEE 754 - Exemplo

$$-1.11111_2 \times 2^{-2}$$

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| s | expoente | | | | | | | | mantissa | | | | | | | | | | | | | | | | | | | | | | |

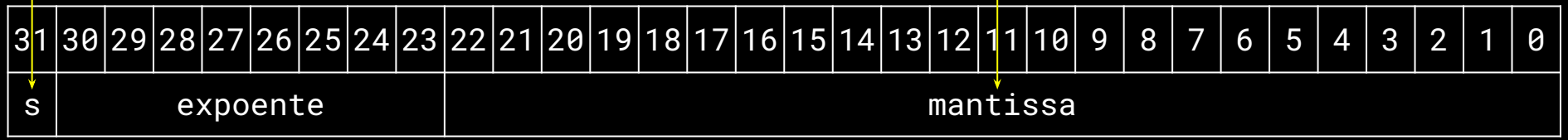
# IEEE 754 - Exemplo

$-1.11111_2 \times 2^{-2}$

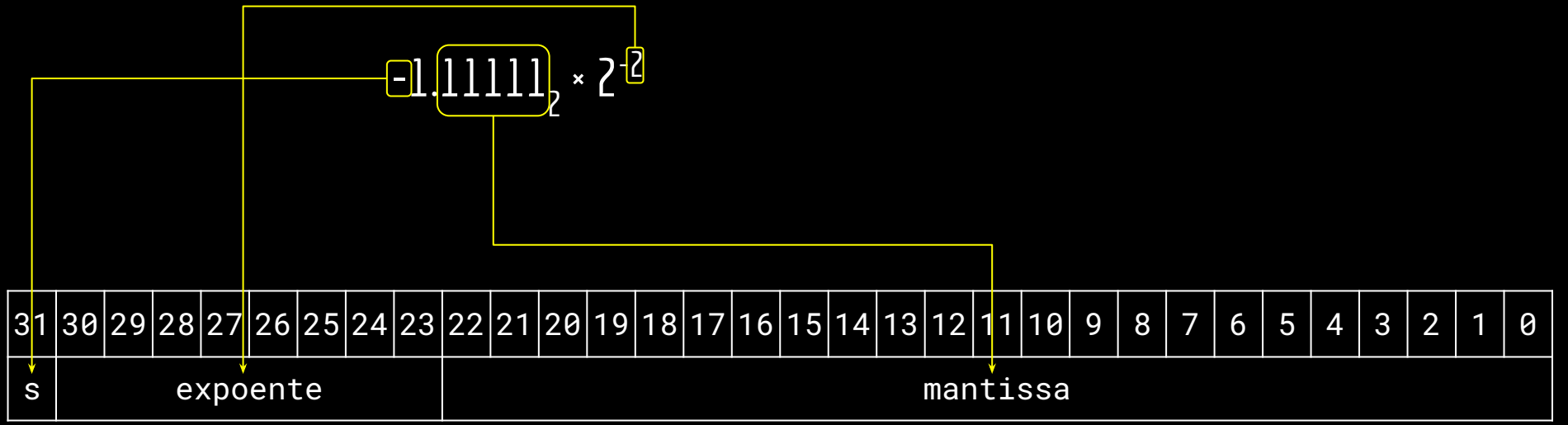


# IEEE 754 - Exemplo

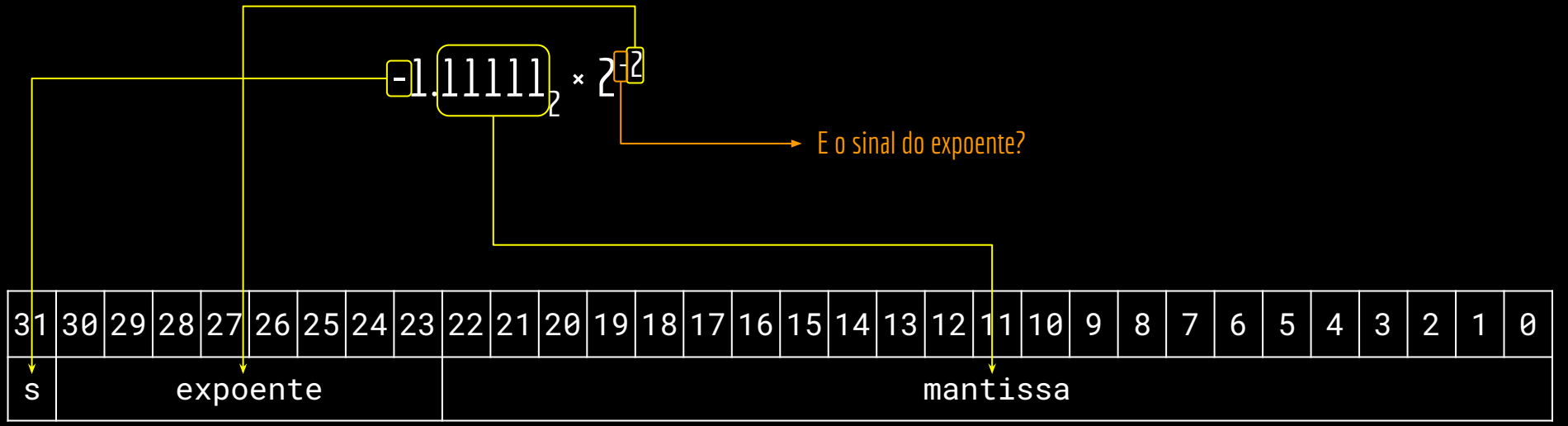
$$-1.11111_2 \times 2^{-2}$$



# IEEE 754 - Exemplo



# IEEE 754 - Exemplo



# Sinal do Expoente

E o sinal do expoente?

Poderíamos utilizar complemento de 2.

Ou então sinal e magnitude, como na mantissa.

# Sinal do Expoente

E o sinal do expoente?

Poderíamos utilizar complemento de 2.

Ou então sinal e magnitude, como na mantissa.

Para que facilitar, se podemos complicar?

# Notação com bias

O IEEE 754 especifica que o expoente utiliza uma **notação com bias.**

***Biased Exponent.***

O **bias** é o **valor intermediário** entre todos os possíveis de serem representados no expoente.

$127_{10}$ ($0111\ 1111_2$) para precisão simples.
$1023_{10}$ ($011\ 1111\ 1111_2$) para precisão dupla.

No caso geral, o bias é $2^{x-1}-1$, onde $x$ é o número de bits no expoente.

**O expoente é somado ao bias.**

# Notação com bias

Exemplos:

O expoente $-1_{10}$ na notação com bias se torna

$$-1_{10} + 127_{10} = 126_{10} = 01111110_2$$

O expoente $1_{10}$ na notação com bias se torna

$$1_{10} + 127_{10} = 128_{10} = 10000000_2$$

# IEEE 754

Um ponto flutuante é então representado por:

$$(-1)^{\text{sinal}} \times (1.0 + \text{mantissa}) \times 2^{(\text{expoente-bias})}$$

Apenas os itens em azul são armazenados na memória.

# Exemplo

Representar $-0.75_{10}$ em precisão simples

# Exemplo

Representar $-0.75_{10}$ em precisão simples

Convertendo para binário pelo método da multiplicação: $0.11_2$

Normalizando: $0.11_2 \times 2^0 = 1.1_2 \times 2^{-1}$

Mantissa: 1 (somente a parte fracionária)
Expoente: $-1 + 127 = 126_{10} = 01111110_2$
Sinal: 1 (negativo)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

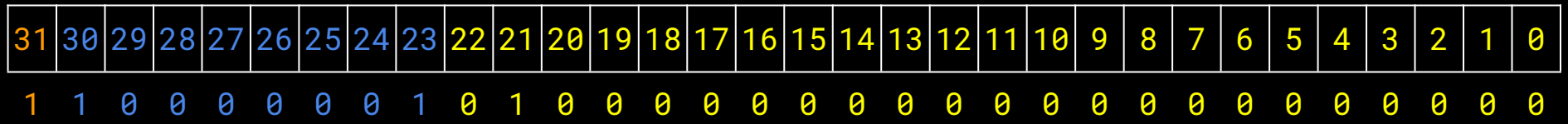
# Exemplo

Converta para decimal.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Exemplo

Converta para decimal.




O expoente é $10000001_2 = 129_{10}$

Subtraindo o bias temos $129 - 127 = 2_{10}$

A mantissa é $1.01_2 = 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} = 1.25_{10}$

O bit de sinal é 1 (negativo)

Logo temos $-1.25 \times 2^2 = -5^{10}$

# Relembrando...

Realizar uma adição utilizando notação científica.

Vamos realizar na base 10, mas o procedimento é o mesmo na base 2.

Considere a seguinte adição: $9.999_{10} \times 10^{1} + 1.610_{10} \times 10^{-1}$.

Considerando que podemos representar 4 dígitos na memória.

# Relembrando...

Considere a seguinte adição: $9.999_{10} \times 10^{1} + 1.610_{10} \times 10^{-1}$.

Considerando que podemos representar 4 dígitos na memória.

Primeiro precisamos igualar o expoente do número com o menor expoente, com o número de maior expoente.

$1.610_{10} \times 10^{-1} = 0.01610_{10} \times 10^{1}$.

# Relembrando...

Considere a seguinte adição: $9.999_{10} \times 10^{1} + 1.610_{10} \times 10^{-1}$.

Considerando que podemos representar 4 dígitos na memória.

Primeiro precisamos igualar o expoente do número com o menor expoente, com o número de maior expoente.

$1.610_{10} \times 10^{-1} = 0.01610_{10} \times 10^{1}$.

Problema: só podemos armazenar 4 dígitos.

$1.610_{10} \times 10^{-1} = 0.016_{10} \times 10^{1}$.

# Relembrando...

Feito isso, basta adicionar as mantissas:

$$9.999_{10} \times 10^{1} + 0.016_{10} \times 10^{1} = 10.015_{10} \times 10^{1}$$

# Relembrando...

Feito isso, basta adicionar as mantissas:

$9.999_{10} \times 10^{1} + 0.016_{10} \times 10^{1} = 10.015_{10} \times 10^{1}$

O resultado não está normalizado. Normalizar:

$1.0015_{10} \times 10^{2}$

# Relembrando...

Feito isso, basta adicionar as mantissas:

$9.999_{10} \times 10^{1} + 0.016_{10} \times 10^{1} = 10.015_{10} \times 10^{1}$

O resultado não está normalizado. Normalizar:

$1.0015_{10} \times 10^{2}$

Mais uma vez não cabe em 4 casas. Arredondando:

$1.002_{10} \times 10^{2}$

# Relembrando...

Feito isso, basta adicionar as mantissas:

$9.999_{10} \times 10^{1} + 0.016_{10} \times 10^{1} = 10.015_{10} \times 10^{1}$

O resultado não está normalizado. Normalizar:

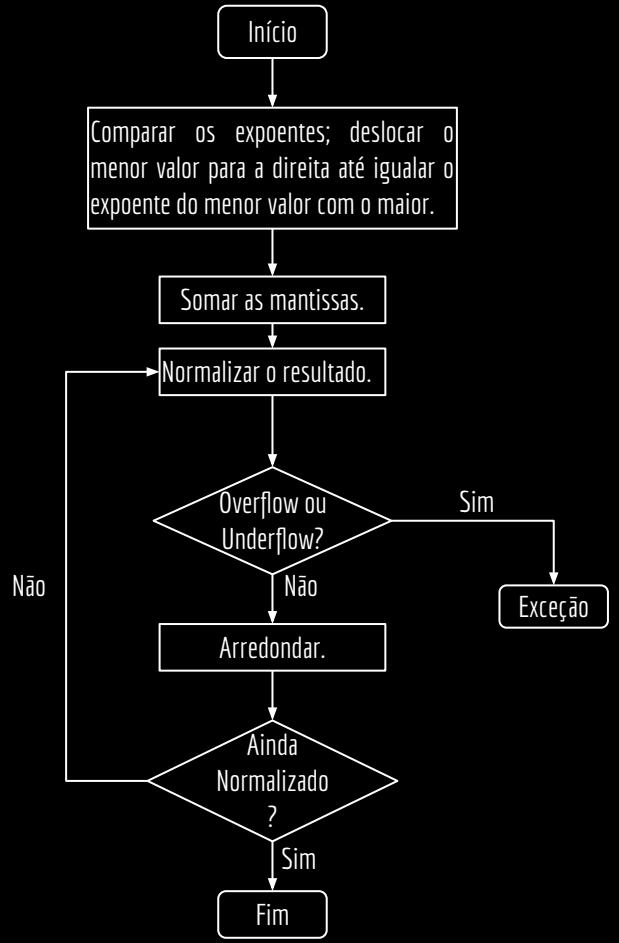
$1.0015_{10} \times 10^{2}$

Mais uma vez não cabe em 4 casas. Arredondando:

$1.002_{10} \times 10^{2}$

Se não tivéssemos que truncar/arrendondar os valores para caber em 4 casas, o resultado correto seria $1,00151 \times 10^{2}$.

# Soma


Início
Comparar os expoentes; deslocar o menor valor para a direita até igualar o expoente do menor valor com o maior.
Somar as mantissas.
Normalizar o resultado.
Overflow ou Underflow?
Sim
Exceção
Não
Não
Arredondar.
Ainda Normalizado ?
Sim
Fim

# Multiplicação

Soma os expoentes (tomando cuidado com o bias).

Multiplicar as mantissas.

# Arredondamento

O IEEE 754 define 4 formas de arredondamento, que podem ser setadas no processador (quando implementado no hardware):

- Arredondar para cima (teto).
- Arredondar para baixo (piso).
- Truncar (ignorar as demais casas).
- Nearest even (par mais próximo) ‹- Mais utilizado.

Detalhes sobre o arredondamento e suas implementações, são encontrados em Patterson e Hennessy (2020).

- Requerem que o processador mantenha bits extras para gerência.

# IEEE 754

Comumente o padrão é embutido no hardware.

Hardwares simples podem não implementar o padrão.

Ex.: microcontroladores.

Nesse caso, o padrão é implementado via software.

De qualquer forma, **independe de linguagem.**

Um ponto flutuante de precisão simples (float) é o mesmo em C, Java, C#, Python, ...

# Outras Precisões

Existem ainda os padrões IEEE 754 para.

- Precisão estendida (comum em processadores x86-64);
- Half-Precision;
- Quad-Precision.

# Exercícios

1. Represente -0.75 em precisão dupla. Compare a resposta com a obtida durante a aula para precisão simples.
2. Qual o maior e o menor valor que podem ser representados em ponto flutuante de precisão dupla e simples (desconsiderando +/ - ∞)? Quais são seus equivalentes em decimal?
3. Exiba os seguintes valores em ponto flutuante. Quando necessário trunque (ignore os bits que não couberem) os valores. Faça o desenho da memória como nos exemplos e coloque os endereços dos bits (para deixar claro a ordem dos bits).
    a. $-16.015625_{10}$ para precisão simples
    b. $-0.1_{10}$ para precisão simples
    c. $0.125_{10}$ em "meia precisão" (half-precision): 10 bits pra mantissa, 5 para expoente e 1 para sinal.
4. Sabendo-se que um número em quad precision possui 15 bits no expoente, responda (deduza você mesmo, sem pesquisar).
    a. Qual o bias para esses números.
    b. Qual o maior e o menor valor que podem ser representados (desconsiderando o +/- infinito)?

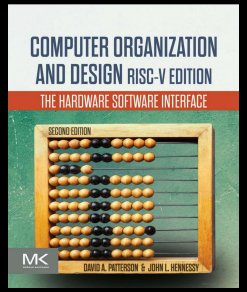
# Exercícios

Considerando seus conhecimentos sobre representação de números em IEEE 754, o que pode dar errado com o programa em C a seguir. Como corrigir?

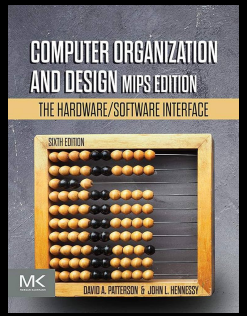
```
//...

double valor;
//faz alguns cálculos

if (valor == 5){
    printf("O resultado é ...");
}
//...
```

# Referências

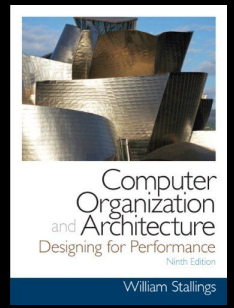

Patterson, Hennessy. Computer Organization and Design RISC-V Edition: The Hardware Software Interface. 2020.



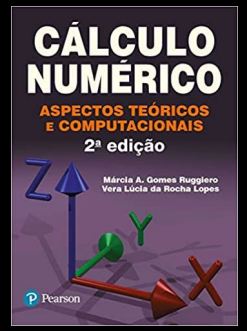

Patterson, Hennessy. Computer Organization and Design MIPS Edition: The Hardware/Software Interface. 2020.



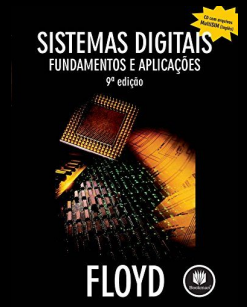

Stallings, W. Organização de Arquitetura de Computadores. 10a Ed. 2016.




Ruggiero, Lopes. Cálculo numérico: aspectos teóricos e computacionais. 1996.




Floyd. Sistemas Digitais: Fundamentos e Aplicações. 2009.



# Licença