Aritmética 3/3

Ponto Flutuante

Edson S. Gomi
Revisão: Marco Túlio Andrade

PCS - Departamento de Engenharia de Computação e Sistemas Digitais
Escola Politécnica da Universidade de São Paulo

Outubro, 2020

Agenda

Representação em Ponto Flutuante

- Em áreas como a física e a química aparecem números muito grandes ou muito pequenos:
  - Sabe-se que a maior distância observável do universo mede cerca de 740 000 000 000 000 000 000 000 000 m;
  - A massa de um próton tem, aproximadamente, o valor de 0,000000000000000000000000167 kg.
- E o que dizer sobre o Número de Avogadro (N) e sobre a constante de Planck (h)?
  - $N = 6.02252 \times 10^{23}$;
  - $h = 6.6256 \times 10^{-27}$ erg.sec.
- Para tratar valores como estes, a notação científica é mais aderente. Incorpora a vantagem de poder representar especificamente a quantidade de algarismos significativos adequada para cada ordem de grandeza tratada.

Representação em Ponto Flutuante

- Representação para números reais, abrangendo números muito pequenos e grandes.
- Notação científica:
  - -2.34×10^{56} (normalizado)
  - $+0.002 \times 10^{4}$ (não normalizado)
  - 987.02×10^{9} (não normalizado)

Em binário : $\pm 1.xxxxxxx_2 \times 2^{yyyy}$

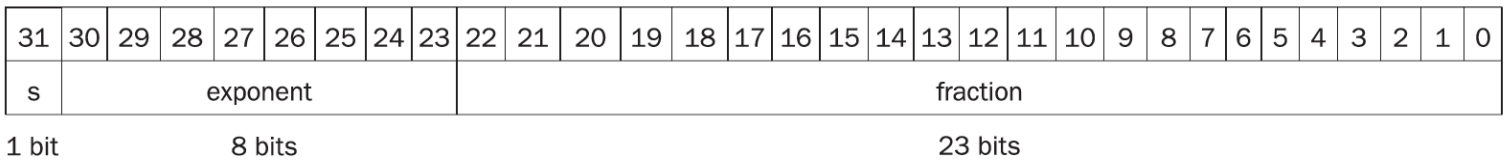
Tipos float e double em C.

Terminologia

- **Scientifc Notation**: representação com números com um dígito único à esquerda do ponto decimal.
- **Normalized**: número em notação científica sem 0s à esquerda (*leading 0s*).
- **Floating Point**: representação de números na qual o ponto binário não é fixo.

Floating Point Single Precision

A figura ilustra o formato da representação em ponto flutuante em precisão simples:



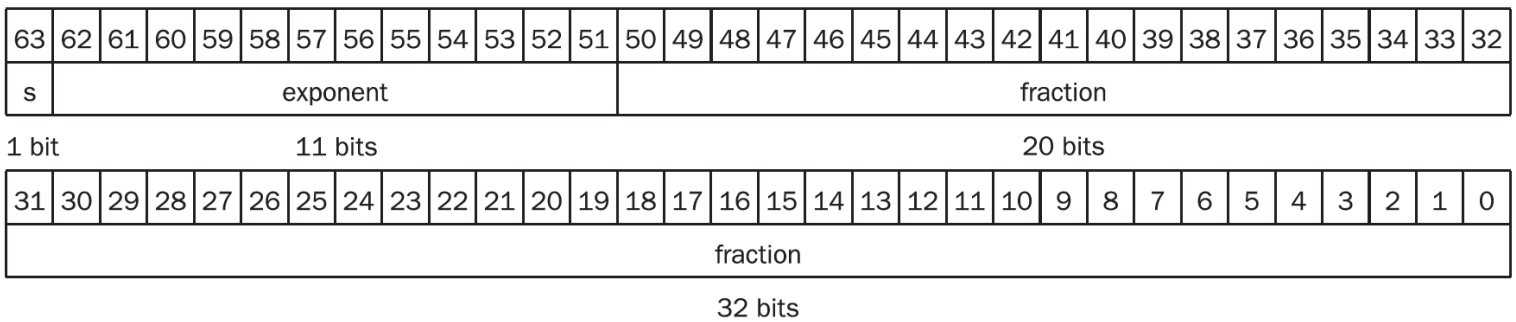
$$\text{Formato} : (-1)^S \times F \times 2^E$$

$$\text{Faixa} : 2.0_{10} \times 10^{-38} < x < 2.0_{10} \times 10^{+38}$$

- **S** = signal (representação Sinal-Magnitude)
- **F** = fraction
- **E** = exponent

Floating Point Double Precision

A figura ilustra o formato da representação em ponto flutuante em precisão dupla:



$$\text{Formato} : (-1)^S \times F \times 2^E$$

$$\text{Faixa} : 2.0_{10} \times 10^{-308} < x < 2.0_{10} \times 10^{+308}$$

Overflow and Underflow

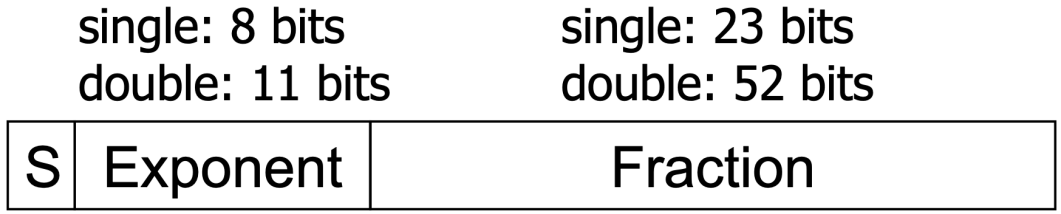
- **Overflow**: o expoente é muito grande para ser representado no campo de expoente E.
- **Underflow**: o expoente negativo é muito grande para ser representado no campo de expoente E.

O que fazer se ocorrer *overflow* ou *underflow*? Uma possibilidade é gerar uma exceção (*exception*, também conhecida como *interrupt*). Uma interrupção provoca a execução de uma *procedure call*: o endereço da instrução que provocou o *overflow*/*underflow* é guardado num registrador e é feito o salto para o início da rotina que fará o tratamento da exceção. O endereço da instrução é guardado para que a execução do programa possa continuar, após a realização de ações corretivas.

IEEE 754 Floating Point Standard

- Desenvolvido em resposta a representações divergentes. Padronização é importante! Por que?
- Portabilidade para código científico.
- Standard quase que universalmente adotado.
- 2 representações:
  - Precisão simples (32 bit)
  - Precisão dupla (64 bit)
- O 1 à esquerda de números normalizados é tornado implícito.
- Fraction representa os 23 ou 52 bits que estão na fração do formato ponto flutuante.
- Significand representa o número com 24 ou 53 bits, ou seja, Significand = 1 mais a Fraction

IEEE 754 Floating Point Standard



$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: $1.0 \leq |\text{significand}| < 2.0$
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the "1." restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

Exponents 00000000 and 11111111 are reserved.

Single Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    ⇒ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    ⇒ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx 2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx 2^{-52}
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Floating Point Encoding

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

- $\pm$infinity : Ao invés de causar uma interrupção após uma divisão por zero, é possível configurar o software para o resultado seja $\pm$infinito.
- NaN (Not a Number) é o resultado de uma operação inválida, como por exemplo 0/0 ou subtração infinity - infinity.

Exemplo de Representação

Representação de $-0.75_{10} = -0.11_2 \times 2^0 = -1.1_2 \times 2^{-1}$ Em precisão simples:

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

1 bit 8 bits 23 bits

Em precisão dupla:

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{two}}) \times 2^{(1022-1023)}$$

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	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 11 bits 20 bits

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

32 bits

Conversão de Ponto Flutuante Binário para Decimal

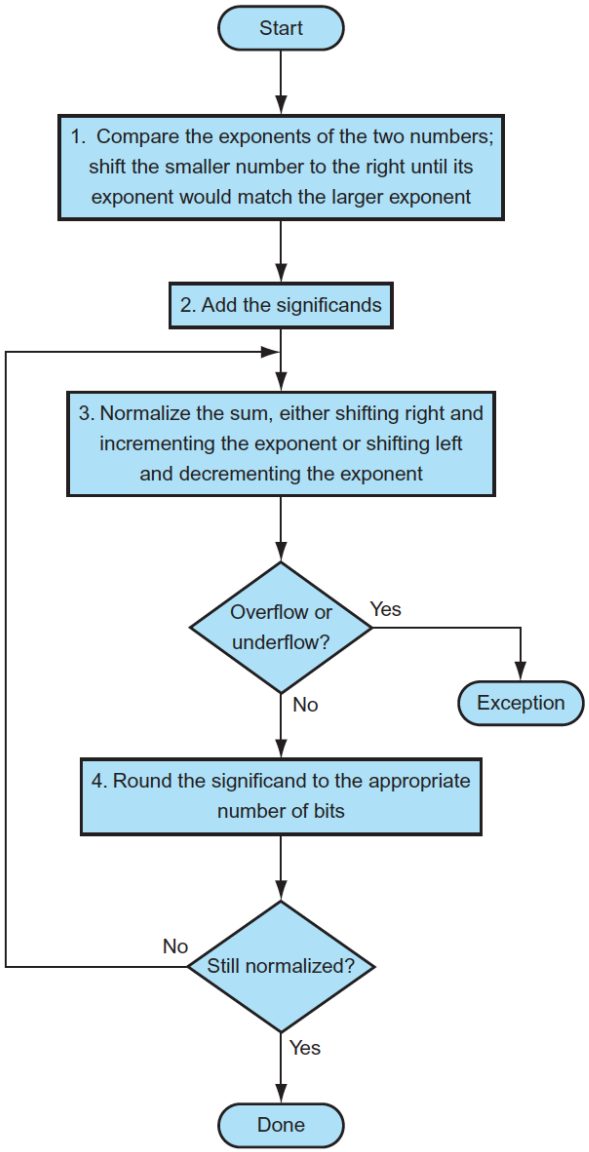
Este ponto flutuante de precisão simples representa qual número decimal ?

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

O bit de sinal é 1, o expoente é 129 e a fração contém $1 \times 2^{-2} = \frac{1}{4} = 0.25$:

$$
\begin{aligned}
(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})} &= (-1)^1 \times (1 + 0.25) \times 2^{(129-127)} \\
&= -1 \times 1.25 \times 2^2 \\
&= -1.25 \times 4 \\
&= -5.0
\end{aligned}
$$

Floating Point Addition



Floating Point Addition - Algorithm

- Consider a 4-digit decimal example
  - $9.999 \times 10^1 + 1.610 \times 10^{-1}$
- 1. Align decimal points
  - Shift number with smaller exponent
  - $9.999 \times 10^1 + 0.016 \times 10^1$
- 2. Add significands
  - $9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$
- 3. Normalize result & check for over/underflow
  - 1.0015×10^2
- 4. Round and renormalize if necessary
  - 1.002×10^2

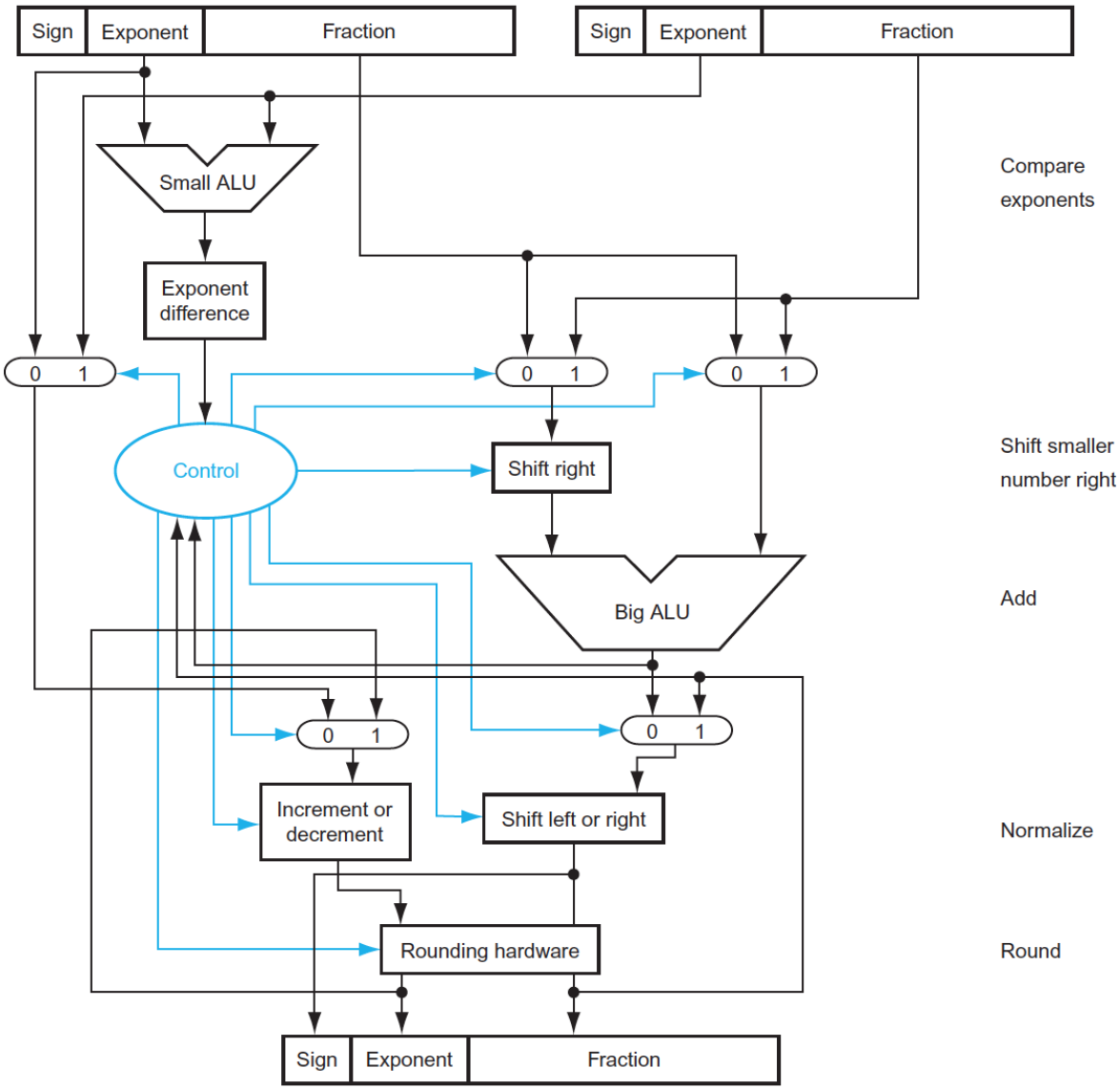
Floating Point Addition

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. Align binary points
  - Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-}1 = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

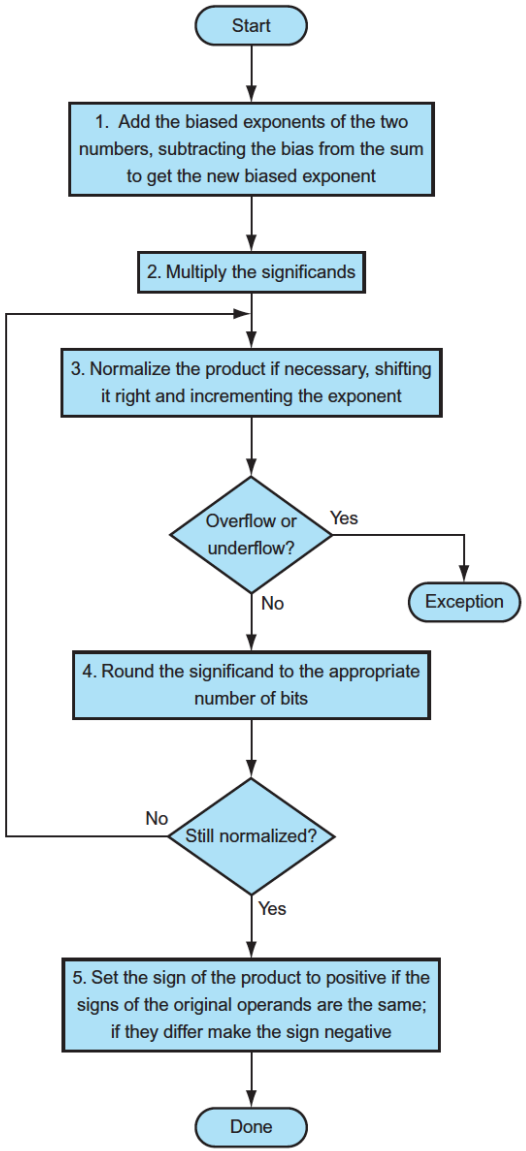
Floating Point Addition Hardware

- É muito mais complexo que o somador inteiro.
- Executar em 1 único ciclo de clock custaria muito tempo:
  - Toma mais tempo que operações inteiras.
  - O clock mais lento penalizaria todas as instruções.
- O somador em ponto flutuante utiliza vários ciclos de clock, podendo ter um *pipeline*.

Floating Point Addition Hardware



Floating Point Multiplication - Algorithm



Floating Point Multiplication

- Consider a 4-digit decimal example
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
  - For biased exponents, subtract bias from sum
  - New exponent = 10 + –5 = 5
- 2. Multiply significands
  - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$
- 3. Normalize result & check for over/underflow
  - 1.0212×10^{6}
- 4. Round and renormalize if necessary
  - 1.021×10^{6}
- 5. Determine sign of result from signs of operands
  - $+1.021 \times 10^{6}$

Floating Point Multiplication

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)
- 1. Add exponents
  - Unbiased: $-1 + -2 = -3$
  - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: +ve × –ve ⇒ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875$

Floating Point Multiplication Hardware

- O multiplicador tem complexidade similar ao somador em ponto flutuante.
- Executar em 1 único ciclo de clock custaria muito tempo:
  - Toma mais tempo que operações inteiras.
  - O clock mais lento penalizaria todas as instruções.
- O multiplicador em ponto flutuante utiliza vários ciclos de clock, podendo ter um *pipeline*.

Leitura Recomendada

- A leitura recomendada sobre os conceitos ministrados nesta aula é a Seção 3.5 do livro *Computer Organization and Design - ARM Edition*, do Patterson e Hennessy.
- Leitura complementar:
  - Knuth, Donald E. The Art of Computer Programming - Volume 2 - Seminumerical Algorithms - Third Edition; 4.2 Section, Addison-Wesley, 1998.
  - Stallings, William. Arquitetura e Organização de Computadores; 5a Edição, Seção 8.4, Prentice Hall, 2.002.

Referências

D. Patterson and J. Hennessy.
Computer Organization and Design ARM Edition: The Hardware Software Interface.
The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2016.

[1]

Obrigado!


Universidade de São Paulo
Departamento de Engenharia de Computação e Sistemas Digitais
PCS