



第2章 信息的表示与处理

100076202: 计算机系统导论

浮点数

Floating Point



任课教师:

计卫星 宿红毅 张艳

Carnegie
Mellon
University

原作者:

Randal E. Bryant and David R. O'Hallaron



议题: 浮点数 Floating Point

- 背景: 二进制小数 Background: Fractional binary numbers
- IEEE浮点标准: 定义 IEEE floating point standard: Definition
- 示例和属性 Example and properties
- 舍入、加法和乘法 Rounding, addition, multiplication
- C语言中的浮点数 Floating point in C
- 小结 Summary



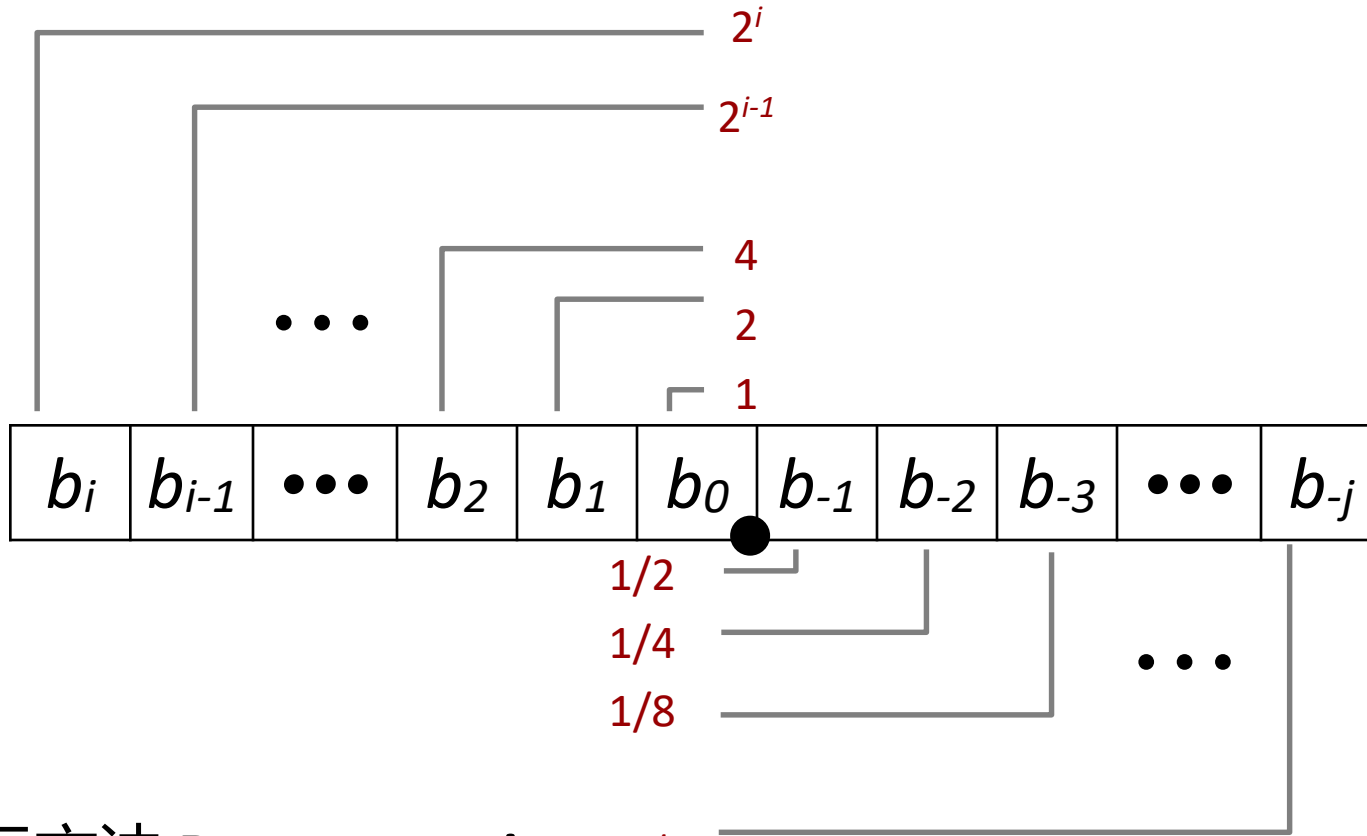
二进制小数

Fractional binary numbers

- What is 1011.101_2 ?



二进制小数 Fractional Binary Numbers



■ 表示方法 Representation 2^{-j}

- “小数点”右边的位代表2的分数幂 Bits to right of “binary point” represent fractional powers of 2

- 代表有理数 Represents rational number:

$$\sum_{k=-j}^i b_k \times 2^k$$

二进制小数示例



Fractional Binary Numbers: Examples

■ 值 Value 表示 Representation

5 $3/4 = 23/4$ 101.11_2

2 $7/8 = 23/8$ 10.111_2

1 $7/16 = 23/16$ 1.0111_2

■ 观察 Observations

- 通过右移来除以2（无符号数） Divide by 2 by shifting right (unsigned)
- 通过左移乘以2 Multiply by 2 by shifting left
- 数字形式 $0.111111..._2$ 是刚好低于1.0的数 Numbers of form $0.111111..._2$ are just below 1.0
 - $1/2 + 1/4 + 1/8 + ... + 1/2^i + ... \rightarrow 1.0$
 - 使用记法为 $1.0 - \epsilon$ Use notation $1.0 - \epsilon$

可表示的数 Representable Numbers



■ 限制#1 Limitation #1

- 仅可以精确地表示 $x/2^k$ 形式的数 Can only exactly represent numbers of the form $x/2^k$
 - 其它有理数有重复的比特位表示 Other rational numbers have repeating bit representations
- 值 Value 表示 Representation
 - $1/3$ $0.0101010101[01]..._2$
 - $1/5$ $0.001100110011[0011]..._2$
 - $1/10$ $0.0001100110011[0011]..._2$

■ 限制#2 Limitation #2 ---定点数

- 在 w 比特位中仅有一个二进制小数点设置 Just one setting of binary point within the w bits
 - 有限的数值范围（非常小的值？非常大的值？） Limited range of numbers (very small values? very large?)



议题: 浮点数 Floating Point

- 背景: 二进制小数 Background: Fractional binary numbers
- **IEEE浮点数标准: 定义 IEEE floating point standard: Definition**
- 示例和属性 Example and properties
- 舍入、加法和乘法 Rounding, addition, multiplication
- C语言中的浮点数 Floating point in C
- 小结 Summary

IEEE浮点数 IEEE Floating Point



■ IEEE 754标准 IEEE Standard 754

- 1985年制定作为浮点运算的统一标准 Established in 1985 as uniform standard for floating point arithmetic
 - 在此之前，有很多异质的格式 Before that, many idiosyncratic formats
- 得到所有主流CPU的支持 Supported by all major CPUs
- 由Kahan为Intel处理器设计（获得1989年图灵奖） Designed by W. Kahan for Intel processors (Turing Award 1989)

■ 由数值问题所驱动 Driven by numerical concerns

- 非常好的标准用于舍入、上溢和下溢 Nice standards for rounding, overflow, underflow
- 在硬件上很难快速运算 Hard to make fast in hardware
 - 数值分析师在定义标准时比硬件设计师更占主导地位 Numerical analysts predominated over hardware designers in defining standard

浮点表示 Floating Point Representation



■ 浮点数形式 Numerical Form:

$$(-1)^s M 2^E$$

Example:

$$15213_{10} = (-1)^0 \times 1.1101101101101_2 \times 2^{13}$$

- 符号位 s 确定数值是负还是正 **Sign bit s** determines whether number is negative or positive
- 尾数 M 是范围在 $[1.0, 2.0)$ 之间的普通小数 **Significand M** normally a fractional value in range $[1.0, 2.0)$.
- 阶码 E 是给浮点数指定 2 的 E 次幂权重 **Exponent E** weights value by power of two

■ 编码 Encoding

- 最高位是符号位 s MSB s is sign bit s
- **exp**字段编码 E (但不等于 E) **exp** field encodes E (but is not equal to E)
- **frac**字段编码 M (但不等于 M) **frac** field encodes M (but is not equal to M)





精度选项 Precision options

■ 单精度浮点数 Single precision: 32 bits



■ 双精度浮点数 Double precision: 64 bits



■ 扩展精度 (仅Intel) Extended precision: 80 bits (Intel only)



“规格化”值

“Normalized” Values

$$V = (-1)^s M 2^E$$



- 当阶码非全零和全一时 When: $\text{exp} \neq 000\dots 0$ and $\text{exp} \neq 111\dots 1$
- 阶码编码为一个有偏置值的有符号数: $E = \text{Exp} - \text{Bias}$
Exponent coded as a *biased* value: $E = \text{Exp} - \text{Bias}$
 - Exp : exp 字段的无符号值 Exp : unsigned value of exp field
 - 偏置 $\text{Bias} = 2^{k-1} - 1$ 其中 k 是阶码位的位数 $\text{Bias} = 2^{k-1} - 1$, where k is number of exponent bits
 - 单精度: 127 Single precision: 127 (Exp : 1...254, E : -126...127)
 - 双精度: 1023 Double precision: 1023 (Exp : 1...2046, E : -1022...1023)
- 尾数编码为带隐含的一个前导1 Significand coded with implied leading 1: $M = 1.\text{xxx}\dots\text{x}_2$
 - frac 字段的比特位 $\text{xxx}\dots\text{x}$: bits of frac field
 - 当 frac 全零时值最小 Minimum when $\text{frac} = 000\dots 0$ ($M = 1.0$)
 - 当 frac 全一时值最大 Maximum when $\text{frac} = 111\dots 1$ ($M = 2.0 - \epsilon$)

规格化编码示例



Normalized Encoding Example

■ 值：单精度浮点数 Value: float $F = 15213.0$;

$$\begin{aligned} 15213_{10} &= 11101101101101_2 \\ &= 1.1101101101101_2 \times 2^{13} \end{aligned}$$

$$V = (-1)^S M 2^E$$
$$E = \text{Exp} - \text{Bias}$$

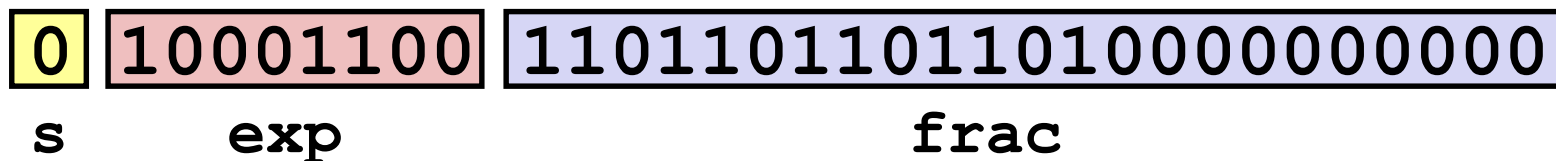
■ 尾数 Significand

$$\begin{aligned} M &= 1.\underline{1101101101101}_2 \\ \text{frac} &= \underline{11011011011010000000000}_2 \end{aligned}$$

■ 阶码 Exponent

$$\begin{aligned} E &= 13 \\ \text{Bias} &= 127 \\ \text{Exp} &= 140 = 10001100_2 \end{aligned}$$

■ Result:





非规格化值 Denormalized Values

- 条件: exp 为全零 Condition: $\text{exp} = 000\dots 0$

$$V = (-1)^S M 2^E$$
$$E = 1 - \text{Bias}$$

- 阶码值: Exponent value: $E = 1 - \text{Bias}$ (instead of $E = 0 - \text{Bias}$)
- 尾数编码为隐含的一个前导零 Significand coded with implied leading 0: $M = 0.\text{xxx}\dots\text{x}_2$
 - frac 字段比特位 $\text{xxx}\dots\text{x}$: bits of frac
- 情况 Cases
 - $\text{exp} = 000\dots 0, \text{frac} = 000\dots 0$
 - 代表0值 Represents zero value
 - 注意区别值: $+0$ 和 -0 (为何?) Note distinct values: $+0$ and -0 (why?)
 - $\text{exp} = 000\dots 0, \text{frac} \neq 000\dots 0$
 - 最接近0.0的数值 Numbers closest to 0.0
 - 平均分布的 Equispaced



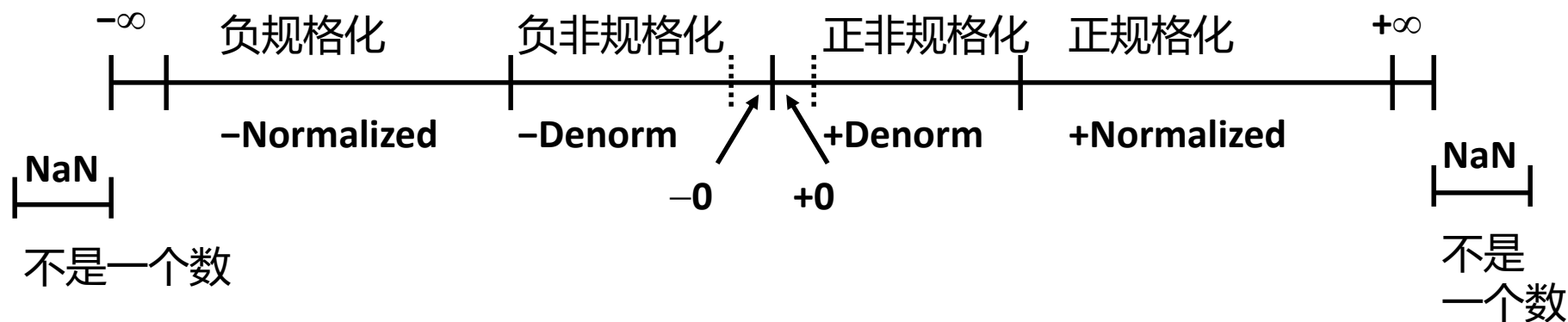
特殊值 Special Values

- 条件: exp 为全一 Condition: $\text{exp} = 111\dots 1$
- 情况 Case: $\text{exp} = 111\dots 1, \text{frac} = 000\dots 0$
 - 代表值无穷大 Represents value ∞ (infinity)
 - 溢出运算 Operation that overflows
 - 正负均如此 Both positive and negative
 - 例如 E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$
- 情况 Case: $\text{exp} = 111\dots 1, \text{frac} \neq 000\dots 0$
 - 不是一个数 Not-a-Number (NaN)
 - 代表无法确定数值的情况 Represents case when no numeric value can be determined
 - 例如 E.g., $\text{sqrt}(-1)$, $\infty - \infty$, $\infty \times 0$



可视化：浮点编码

Visualization: Floating Point Encodings





C语言中的浮点数示例

float: 0xC0A00000

binary: _____



E =

S =

M =

$v = (-1)^S M 2^E =$

$$v = (-1)^S M 2^E$$

$$E = \text{exp} - \text{Bias}$$

$$\text{Bias} = 2^{k-1} - 1 = 127$$

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111



C语言中的浮点数示例#1

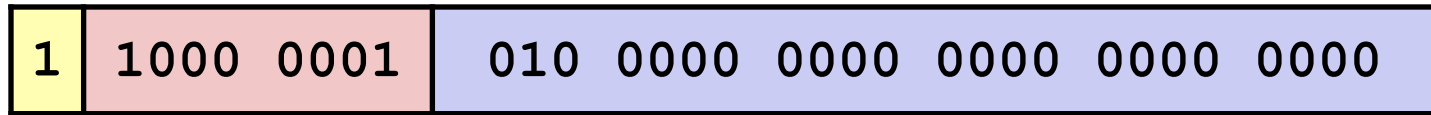
float: 0xC0A00000

$$v = (-1)^s M 2^E$$

$$E = \text{exp} - \text{Bias}$$

$$\text{Bias} = 2^{k-1} - 1 = 127$$

binary: 1100 0000 1010 0000 0000 0000 0000 0000



1

8-bits

23-bits

$$E = \text{exp} - \text{Bias} = 129 - 127 = 2 \text{ (decimal)}$$

$$S = 1 \rightarrow \text{负数}$$

$$M = 1.010 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000$$

$$= 1 + 1/4 = 1.25$$

$$v = (-1)^s M 2^E = (-1)^1 * 1.25 * 2^2 = -5$$

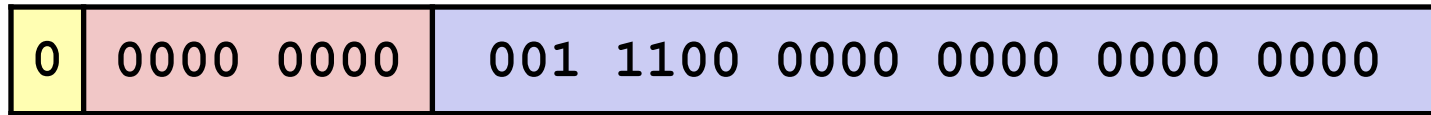
Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111



C语言中的浮点数示例#2

float: 0x001C0000

binary: 0000 0000 0001 1100 0000 0000 0000 0000



1

8-bits

23-bits

$$E = 1 - \text{Bias} = 1 - 127 = -126 \text{ (decimal)}$$

$$S = 0 \rightarrow \text{正数}$$

$$M = 0.001 \ 1100 \ 0000 \ 0000 \ 0000 \ 0000$$

$$= 1/8 + 1/16 + 1/32 = 7/32 = 7 * 2^{-5}$$

$$v = (-1)^S M 2^E = (-1)^0 * 7 * 2^{-5} * 2^{-126} = 7 * 2^{-131}$$

$$\approx 2.571393892 \times 10^{-39}$$

$$v = (-1)^S M 2^E$$

$$E = 1 - \text{Bias}$$

$$\text{Bias} = 2^{k-1} - 1 = 127$$

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111



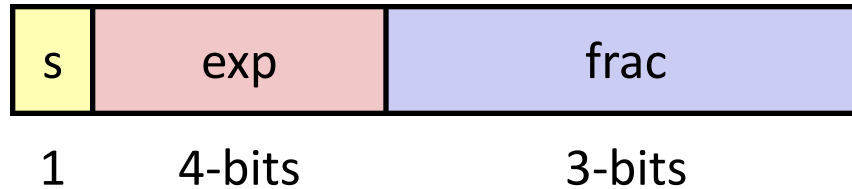
议题: 浮点数 Floating Point

- 背景: 二进制小数 Background: Fractional binary numbers
- IEEE浮点数标准: 定义 IEEE floating point standard: Definition
- 示例和属性 **Example and properties**
- 舍入、加法和乘法 Rounding, addition, multiplication
- C语言中的浮点数 Floating point in C
- 小结 Summary



微小的浮点数示例

Tiny Floating Point Example



■ 8位浮点表示 8-bit Floating Point Representation

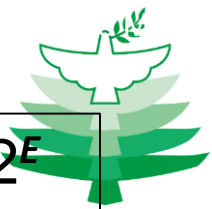
- 符号位是在最高有效位 the sign bit is in the most significant bit
- 然后是四位阶码，偏置为7 the next four bits are the exponent, with a bias of 7
- 最后的三位是尾数 the last three bits are the **frac**

■ 与IEEE格式同样的通用形式 Same general form as IEEE Format

- 规格化、非规格化 normalized, denormalized
- 0、NaN和无穷大的表示 representation of 0, NaN, infinity

动态范围 (仅正数)

Dynamic Range (Positive Only)



$$V = (-1)^s M 2^E$$

$n: E = Exp - Bias$
 $d: E = 1 - Bias$

非规格化数
Denormalized
numbers

s	exp	frac	E	Value
0	0000	000	-6	0
0	0000	001	-6	$1/8 * 1/64 = 1/512$
0	0000	010	-6	$2/8 * 1/64 = 2/512$
...				
0	0000	110	-6	$6/8 * 1/64 = 6/512$
0	0000	111	-6	$7/8 * 1/64 = 7/512$

最接近0 closest to zero

最大非规格化数 largest denorm

规格化数
Normalized
numbers

0	0001	000	-6	$8/8 * 1/64 = 8/512$
0	0001	001	-6	$9/8 * 1/64 = 9/512$
...				
0	0110	110	-1	$14/8 * 1/2 = 14/16$
0	0110	111	-1	$15/8 * 1/2 = 15/16$
0	0111	000	0	$8/8 * 1 = 1$
0	0111	001	0	$9/8 * 1 = 9/8$
0	0111	010	0	$10/8 * 1 = 10/8$
...				
0	1110	110	7	$14/8 * 128 = 224$
0	1110	111	7	$15/8 * 128 = 240$

最小规格化数 smallest norm

最接近1以下 closest to 1 below

最接近1以上 closest to 1 above

最大规格化数 largest norm

无穷大 inf

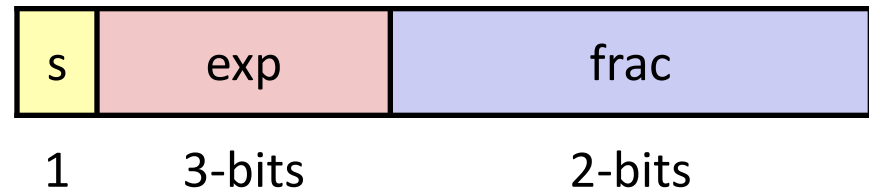
0 1111 000 n/a inf



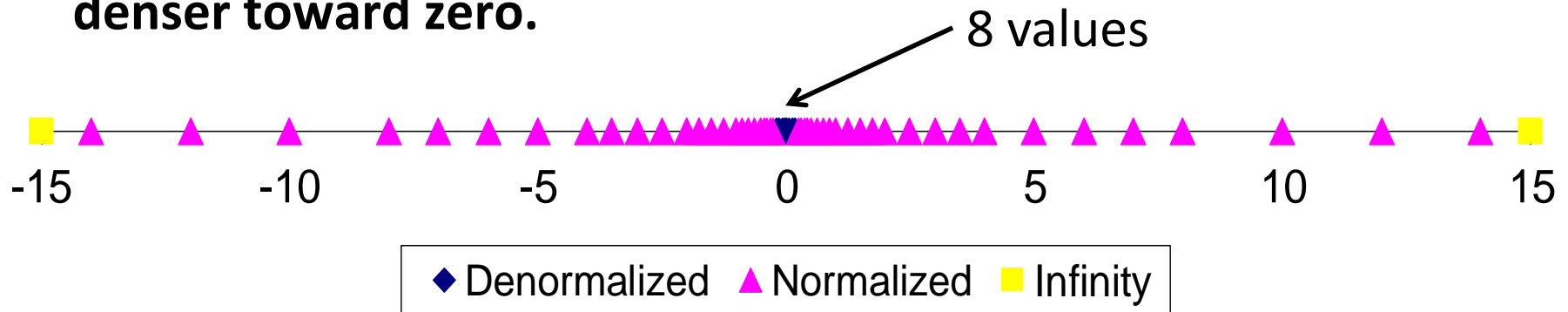
值的分布 Distribution of Values

■ 6位类IEEE格式 6-bit IEEE-like format

- 3位阶码 $e = 3$ exponent bits
- 2位尾数 $f = 2$ fraction bits
- 偏置是3 Bias is $2^{3-1}-1 = 3$



■ 注意到越接近零分布越密集 Notice how the distribution gets denser toward zero.



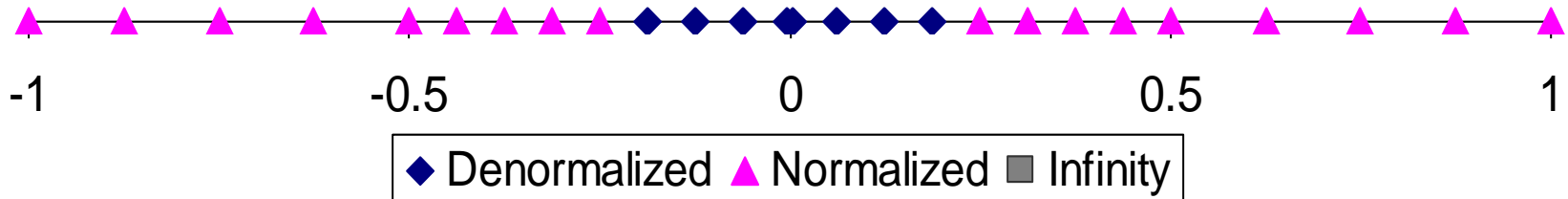
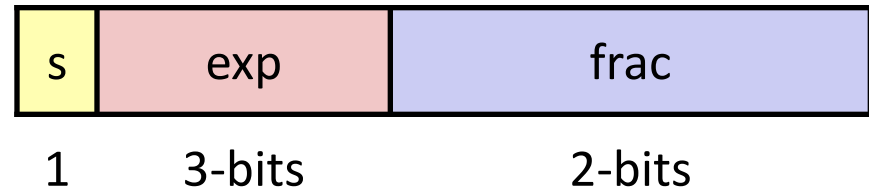


值的分布（近处观察）

Distribution of Values (close-up view)

■ 6位类IEEE格式 6-bit IEEE-like format

- 3位阶码 $e = 3$ exponent bits
- 2位尾数 $f = 2$ fraction bits
- 偏置量是3 Bias is 3



有趣的数值 Interesting Numbers

{single, double}



<i>Description</i>	<i>exp</i>	<i>frac</i>	<i>Numeric Value</i>
■ Zero	00...00	00...00	0.0
■ Smallest Pos. Denorm.	00...00	00...01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
<ul style="list-style-type: none"> ■ Single $\approx 1.4 \times 10^{-45}$ ■ Double $\approx 4.9 \times 10^{-324}$ 			
■ Largest Denormalized	00...00	11...11	$(1.0 - \epsilon) \times 2^{-\{126,1022\}}$
<ul style="list-style-type: none"> ■ Single $\approx 1.18 \times 10^{-38}$ ■ Double $\approx 2.2 \times 10^{-308}$ 			
■ Smallest Pos. Normalized	00...01	00...00	$1.0 \times 2^{-\{126,1022\}}$
<ul style="list-style-type: none"> ■ Just larger than largest denormalized 			
■ One	01...11	00...00	1.0
■ Largest Normalized	11...10	11...11	$(2.0 - \epsilon) \times 2^{\{127,1023\}}$
<ul style="list-style-type: none"> ■ Single $\approx 3.4 \times 10^{38}$ ■ Double $\approx 1.8 \times 10^{308}$ 			



IEEE编码的特殊属性

Special Properties of the IEEE Encoding

- 浮点数的零和整数的零相同 **FP Zero Same as Integer Zero**
 - 所有比特位为0 All bits = 0
- 几乎可以使用无符号整数比较 **Can (Almost) Use Unsigned Integer Comparison**
 - 必须首先比较符号位 Must first compare sign bits
 - 必须考虑-0=0 Must consider $-0 = 0$
 - 不是一个数NaN的问题 NaNs problematic
 - 比任何其它值都大 Will be greater than any other values
 - 还应该比较什么? What should comparison yield?
 - 否则都没有问题 Otherwise OK
 - 非规格化和规格化 Denorm vs. normalized
 - 规格化和无穷大 Normalized vs. infinity



议题: 浮点数 Floating Point

- 背景: 二进制小数 Background: Fractional binary numbers
- IEEE浮点数标准: 定义 IEEE floating point standard: Definition
- 示例和属性 Example and properties
- 舍入、加法和乘法 Rounding, addition, multiplication
- C语言中的浮点数 Floating point in C
- 小结 Summary



舍入 Rounding

■ 舍入模式（用美元来说明） Rounding Modes (illustrate with \$ rounding)

■		\$1.40	\$1.60	\$1.50	\$2.50	-\$1.50
■	向零 Towards zero	\$1	\$1	\$1	\$2	-\$1
■	向下 Round down ($-\infty$)	\$1	\$1	\$1	\$2	-\$2
■	向上 Round up ($+\infty$)	\$2	\$2	\$2	\$3	-\$1
■	最近的偶数（默认）	\$1	\$2	\$2	\$2	-\$2
■	Nearest Even (default)					

*将数字向上或者向下舍入，使得结果的最低有效位是偶数

近处观察向偶数舍入



Closer Look at Round-To-Even

■ 默认的舍入模式 Default Rounding Mode

- 没有深入汇编级很难理解任何其它类型的舍入 Hard to get any other kind without dropping into assembly
- 所有其它的舍入都有统计偏差 All others are statistically biased
 - 一组正数的和将始终高估或低估 Sum of set of positive numbers will consistently be over- or under- estimated

■ 适用于其它小数位/位位置 Applying to Other Decimal Places / Bit Positions

- 当正好位于两个可能值中间时 When exactly halfway between two possible values
 - 舍入以便最低位是偶数 Round so that least significant digit is even
- 例如舍入到最近的百分位 E.g., round to nearest hundredth

7.8949999	7.89	(低于中间值 Less than half way)
7.8950001	7.90	(高于中间值 Greater than half way)
7.8950000	7.90	(中间值-向上舍入 Half way—round up)
7.8850000	7.88	(中间值-向下舍入 Half way—round down)



舍入二进制数 Rounding Binary Numbers

■ 二进制小数 Binary Fractional Numbers

- 当最低位是0时为偶数 “Even” when least significant bit is 0
- 当舍入位置右边=100...₂时为“中间值” “Half way” when bits to right of rounding position = 100...₂

■ 举例 Examples

- 舍入到最接近1/4 (小数点右边2位) Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00 011 ₂	10.00 ₂	(<1/2—down向下)	2
2 3/16	10.00 110 ₂	10.01 ₂	(>1/2—up向上)	2 1/4
2 7/8	10.11 100 ₂	11.00 ₂	(1/2—up向上)	3
2 5/8	10.10 100 ₂	10.10 ₂	(1/2—down向下)	2 1/2

舍入 Rounding



1 . BBG**RXXX**

监督位：结果的最低位

Guard bit: LSB of result

舍入位：删除的第一位

Round bit: 1st bit removed

固着位：剩余位的或

Sticky bit: OR of remaining bits

■ 向上舍入条件 Round up conditions

- Round = 1, Sticky = 1 \rightarrow > 0.5
- Guard = 1, Round = 1, Sticky = 0 \rightarrow Round to even 向偶数舍入

<i>Value</i>	<i>Fraction</i>	<i>GRS</i>	<i>Incr?</i>	<i>Rounded</i>
128	1.000 0000	000	N	1.000
15	1.101 0000	100	N	1.101
17	1.000 1000	010	N	1.000
19	1.001 1000	110	Y	1.010
138	1.000 1010	011	Y	1.001
63	1.111 1100	111	Y	10.000



浮点运算：基本思想

Floating Point Operations: Basic Idea

■ $x +_f y = \text{Round}(x + y)$

■ $x \times_f y = \text{Round}(x \times y)$

■ 基本思想 Basic idea

- 首先计算精确的结果 First **compute exact result**
- 使它适合需要的精度 Make it fit into desired precision
 - 如果阶码太大可能会溢出 Possibly overflow if exponent too large
 - 可能需要舍入才能适合尾数位数 Possibly **round to fit into frac**

浮点加法 Floating Point Addition



■ $(-1)^{s1} M1 2^{E1} + (-1)^{s2} M2 2^{E2}$

▪ 假设 Assume $E1 > E2$

小数点对齐 Get binary points lined up

■ 精确的结果 **Exact Result:** $(-1)^s M 2^E$

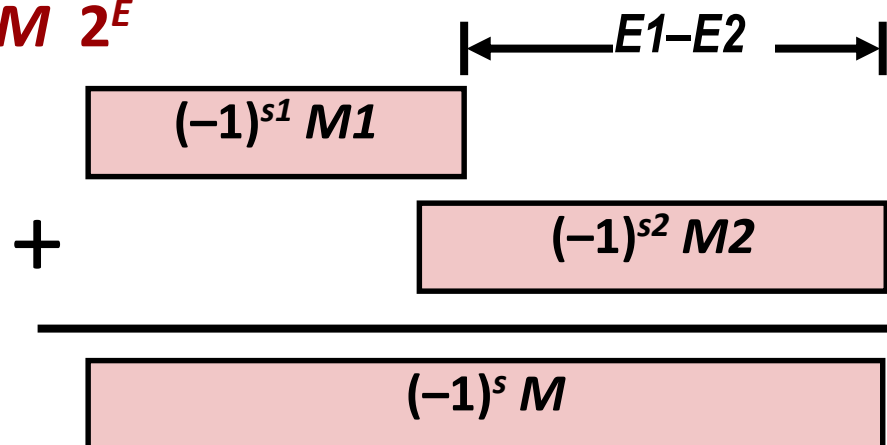
▪ 符号位 s , 尾数 M :

▪ Sign s , significand M :

▪ 有符号数对齐并相加的结果

▪ Result of signed align & add

▪ 阶码 E 是 $E1$ Exponent E : $E1$



■ 修正 Fixing

▪ 如果 M 大于等于 2, M 右移, E 加一 If $M \geq 2$, shift M right, increment E

▪ 如果 M 小于 1, M 左移 k 位, E 减 k if $M < 1$, shift M left k positions, decrement E by k

▪ 如果 E 超过范围则溢出 Overflow if E out of range

▪ 舍入 M 到适合 frac 的精度 Round M to fit **frac** precision



浮点加法 Floating Point Addition

$$\begin{aligned} 1.010 * 2^2 + 1.110 * 2^3 &= (0.1010 + 1.1100) * 2^3 \\ &= 1\textcolor{red}{0}.0110 * 2^3 = 1.001\textcolor{red}{10} * 2^4 = 1.010 * 2^4 \end{aligned}$$



浮点加法的数学性质

Mathematical Properties of FP Add

■ 相比于其它阿贝尔群 Compare to those of Abelian Group

- 加法封闭吗? Closed under addition? **Yes**
 - 但可能产生无穷大或NaN But may generate infinity or NaN
- 可交换吗? Commutative? **Yes**
- 可结合吗? Associative? **No**
 - 溢出和舍入的不精确性 Overflow and inexactness of rounding
 - $(3.14+1e10)-1e10 = 0$, $3.14+(1e10-1e10) = 3.14$
- 0是加性恒等 (单位元) 的吗? 0 is additive identity? **Yes**
- 每个元素都有加法逆元吗? Every element has additive inverse?
 - 对, 除了无穷大和NaNs Yes, except for infinities & NaNs **Almost**

■ 单调性 Monotonicity

- $a \geq b \Rightarrow a+c \geq b+c$? **Almost**
 - 除了无穷大和NaNs Except for infinities & NaNs



浮点乘法 FP Multiplication

■ $(-1)^{s1} M1 2^{E1} \times (-1)^{s2} M2 2^{E2}$

■ 精确的结果 **Exact Result:** $(-1)^s M 2^E$

- 符号位s: 异或 Sign s: $s1 \wedge s2$
- 尾数M: 相乘 Significand M: $M1 \times M2$
- 阶码E: 相加 Exponent E: $E1 + E2$

■ **修正 Fixing**

- 如果M大于等于2, M右移, 阶码E加一 If $M \geq 2$, shift M right, increment E
- 如果E超过范围, 溢出 If E out of range, overflow
- 舍入M到适合frac的精度 Round M to fit **frac** precision

■ **实现 Implementation**

- 最繁琐的工作是尾数相乘 Biggest chore is multiplying significands

4 位尾数: $1.010 \times 2^2 \times 1.110 \times 2^3 = 10.0011 \times 2^5$
 $= 1.00011 \times 2^6 = 1.001 \times 2^6$

浮点乘法的数学性质



Mathematical Properties of FP Mult

■ 相比于交换环 Compare to Commutative Ring

- 乘法封闭吗? Closed under multiplication? **Yes**
 - 但可能产生无穷大或NaN But may generate infinity or NaN
- 乘法可交换吗? Multiplication Commutative? **Yes**
- 乘法具有结合性吗? Multiplication is Associative? **No**
 - 可能溢出, 不精确的舍入 Possibility of overflow, inexactness of rounding
 - Ex: $(1e20 * 1e20) * 1e-20 = \text{inf}$, $1e20 * (1e20 * 1e-20) = 1e20$
- 1是乘法恒等 (单位元) 的吗? 1 is multiplicative identity? **Yes**
- 乘法对加法是可分配的吗? Multiplication distributes over addition? **No**
 - 可能溢出, 不精确舍入 Possibility of overflow, inexactness of rounding
 - $1e20 * (1e20 - 1e20) = 0.0$, $1e20 * 1e20 - 1e20 * 1e20 = \text{NaN}$

■ 单调性 Monotonicity

Almost

- $a \geq b \ \& \ c \geq 0 \Rightarrow a * c \geq b * c$
 - 除了无穷大和NaNs Except for infinities & NaNs



议题: 浮点数 Floating Point

- 背景: 二进制小数 Background: Fractional binary numbers
- IEEE浮点数标准: 定义 IEEE floating point standard: Definition
- 示例和属性 Example and properties
- 舍入、加法和乘法 Rounding, addition, multiplication
- C语言中的浮点数 Floating point in C
- 小结 Summary

C语言中的浮点数 Floating Point in C



■ C语言确保两个级别的浮点数 C Guarantees Two Levels

- **float** single precision 单精度
- **double** double precision 双精度

■ 转换/强制转换 Conversions/Casting

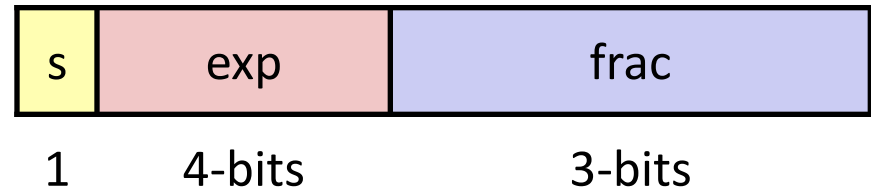
- 在int, float和double之间强制转换改变比特位表示 Casting between **int**, **float**, and **double** changes bit representation
- **double/float** → **int**
 - 截断尾数部分 Truncates fractional part
 - 就像向零舍入 Like rounding toward zero
 - 当超过范围或NaN时没有定义：一般设置为TMin Not defined when out of range or NaN: Generally sets to TMin
- **int** → **double**
 - 精确转换，只要int字长小于等于53位 Exact conversion, as long as **int** has ≤ 53 bit word size
- **int** → **float**
 - 将按照舍入模式进行舍入 Will round according to rounding mode

创建浮点数 Creating Floating Point Number



■ 步骤 Steps

- 用前导1规格化尾数 Normalize to have leading 1
- 舍入以适合尾数位 Round to fit within fraction
- 后规格化以处理舍入的影响 Postnormalize to deal with effects of rounding



■ 案例研究 Case Study

- 转换8位无符号数成微小浮点数格式 Convert 8-bit unsigned numbers to tiny floating point format

示例数值 Example Numbers

128 10000000

13 00001101

17 00010001

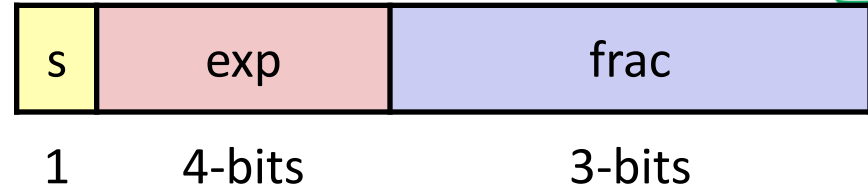
19 00010011

138 10001010

63 00111111



规格化 Normalize



■ 需求 Requirement

- 设置小数点以便数值格式为1.xxxxx Set binary point so that numbers of form 1.xxxxx
- 调整所有位得到前导1 Adjust all to have leading one
 - 随着尾数左移，阶码减一 Decrement exponent as shift left
 - **尾数右移，阶码+1**

<i>Value</i>	<i>Binary</i>	<i>Fraction</i>	<i>Exponent</i>
128	10000000	1.0000000	7
13	00001101	1.1010000	3
17	00010001	1.0001000	4
19	00010011	1.0011000	4
138	10001010	1.0001010	7
63	00111111	1.1111100	5



后规格化 Postnormalize

■ 问题 Issue

- 舍入可能导致溢出 Rounding may have caused overflow
- 通过右移一次进行处理同时阶码加一 Handle by shifting right once & incrementing exponent

<i>Value</i>	<i>Rounded</i>	<i>Exp</i>	<i>Adjusted</i>	<i>Result</i>
128	1.000	7		128
13	1.101	3		13
17	1.000	4		16
19	1.010	4		20
138	1.001	7		134
63	10.000	5	1.000/6	64



浮点数难题 Floating Point Puzzles

- 对于下面的每个C表达式 For each of the following C expressions, either: 完成其中一个工作
 - 对于所有的参数值解释其值为真 Argue that it is true for all argument values
 - 解释为何不为真 Explain why not true

```
int x = ...;  
float f = ...;  
double d = ...;
```

假设d和f都不是NaN
Assume neither
d nor f is NaN

- `x == (int)(float) x` ❌
- `x == (int)(double) x` ✅
- `f == (float)(double) f` ✅
- `d == (double)(float) d` ❌
- `f == -(-f);` ✅
- `2/3 == 2/3.0` ❌
- `d < 0.0` \Rightarrow `((d*2) < 0.0)` ✅
- `d > f` \Rightarrow `-f > -d` ✅
- `d * d >= 0.0` ✅
- `(d+f) - d == f` ❌



小结 Summary

- IEEE浮点数有明确的数学性质 IEEE Floating Point has clear mathematical properties
- 表示浮点数的形式为 Represents numbers of form $M \times 2^E$
- 运算和实现是相互独立的 One can reason about operations independent of implementation
 - 好像采用完美的精度进行计算，然后进行舍入 As if computed with perfect precision and then rounded
- 与实数运算并不相同 Not the same as real arithmetic
 - 违背结合率和分配律 Violates associativity/distributivity
 - 对编译器和严谨数值应用程序员是一个挑战 Makes life difficult for compilers & serious numerical applications programmers

关于浮点数的灾难性影响 (I)



Disastrous effects on floating Point (I)

- 浮点运算的不精确性 The imprecision of floating-point arithmetic
- 1991年2月25日，在第一次海湾战争期间，位于沙特阿拉伯 Dharan 的美国爱国者导弹连未能拦截来袭的伊拉克飞毛腿导弹。飞毛腿袭击了美国陆军营房并杀死了28名士兵。 On February 25, 1991, during the first Gulf War, an American **Patriot Missile** battery in Dharan, Saudi Arabia, failed to intercept an incoming Iraqi **Scud missile**. The Scud struck an American Army barracks and killed 28 soldiers.
- 爱国者系统包含一个内部时钟，以计数器的形式实现，每0.1 秒递增一次。为了确定以秒为单位的时间，程序会将此计数器的值乘以24比特位的小数，该小数是1/10的二进制近似值。 The Patriot system contains an internal clock, implemented as a counter that is incremented every 0.1 seconds. To determine the time in seconds, the program would multiply the value of this counter by a 24-bit quantity that was a fractional binary approximation to 1/10 .



关于浮点数的灾难性影响 (I)

Disastrous effects on floating Point (I)

- 程序采用0.1的近似值，其值为 $x=0.0001100110011001100$
The program approximated 0.1, as a value
 $x=0.0001100110011001100$
- $0.1-x = 0.000000000000000000000000\textcolor{red}{000}[1100][1100]..._2$
- $0.1-x = 2^{-20} \times 0.1 = 9.54 \times 10^{-8}$
- 开始100小时后，有0.343秒的差异 After starting 100 hrs, there are 0.343s difference
- 飞毛腿以2000米/秒左右的速度行驶，686米差距 Scud travels at around 2000 m/s, 686 meters far off
- 软件升级未完成，有时使用准确的定时，有时读取不准确的定时 Software upgrade not completed, sometimes using accurate timing, reading inaccurate timing otherwise

关于浮点数的灾难性影响 (II)



Disastrous effects on floating Point (II)

- 1996年6月4日，阿丽亚娜Ariane 5号火箭的处女航行 The maiden voyage of the Ariane 5 rocket, on June 4, 1996.
- 发射后仅37秒，火箭就偏离了飞行路径，解体并爆炸 Just 37 seconds after liftoff, the rocket veered off its flight path, broke up, and exploded.
- 在将64位浮点数转换为16位有符号整数的过程中发生了溢出 An overflow had occurred during the conversion of a 64-bit floating-point number to a 16-bit signed integer
- 溢出的值测量了火箭的水平速度 The value that overflowed measured the horizontal velocity of the rocket.
- 在Ariane 4中，水平速度永远不会溢出16位数字。Ariane 5只是以5倍的速度重用相同的软件 In the Ariane 4, the horizontal velocity would never overflow a 16-bit number. Ariane 5 just reuses the same software with 5 times higher velocity.

