Floating Point

15-213/18-213/15-513: Introduction to Computer Systems
4th Lecture, Sept. 7, 2017

Today's Instructor:

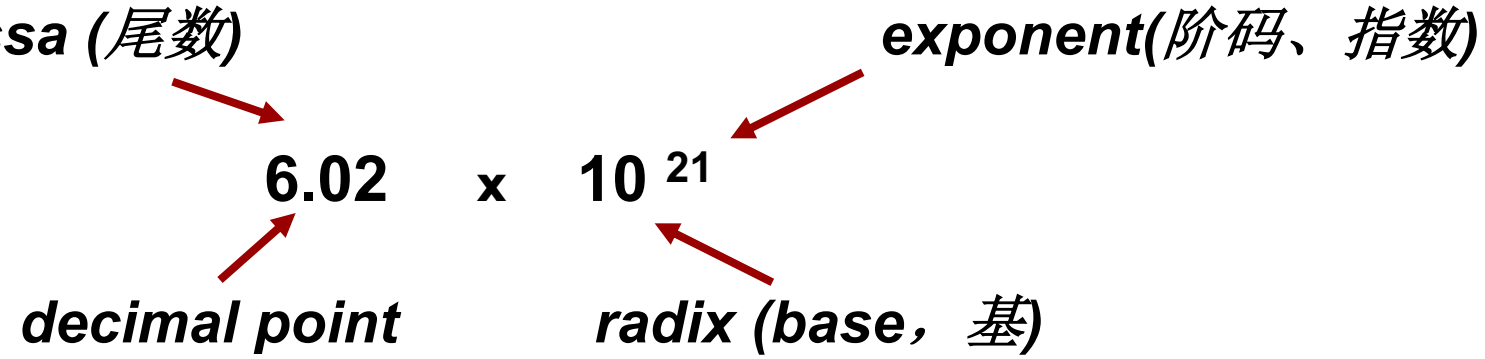
Phil Gibbons

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

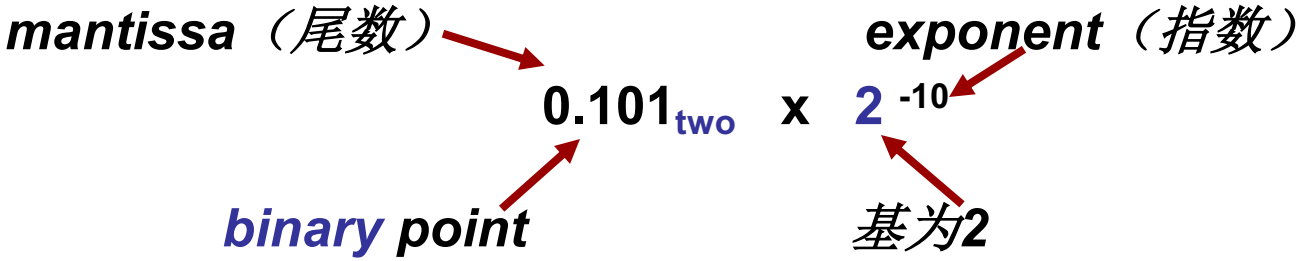
科学计数法(Scientific Notation)与浮点数

Example:



- Normalized form（规格化形式）：小数点前只有一位非0数
- 同一个数有多种表示形式。例：对于数 1/1,000,000,000
  - Normalized (唯一的规格化形式): 1.0×10^{-9}
  - Unnormalized（非规格化形式不唯一）: 0.1×10^{-8}, 10.0×10^{-10}

for Binary Numbers:

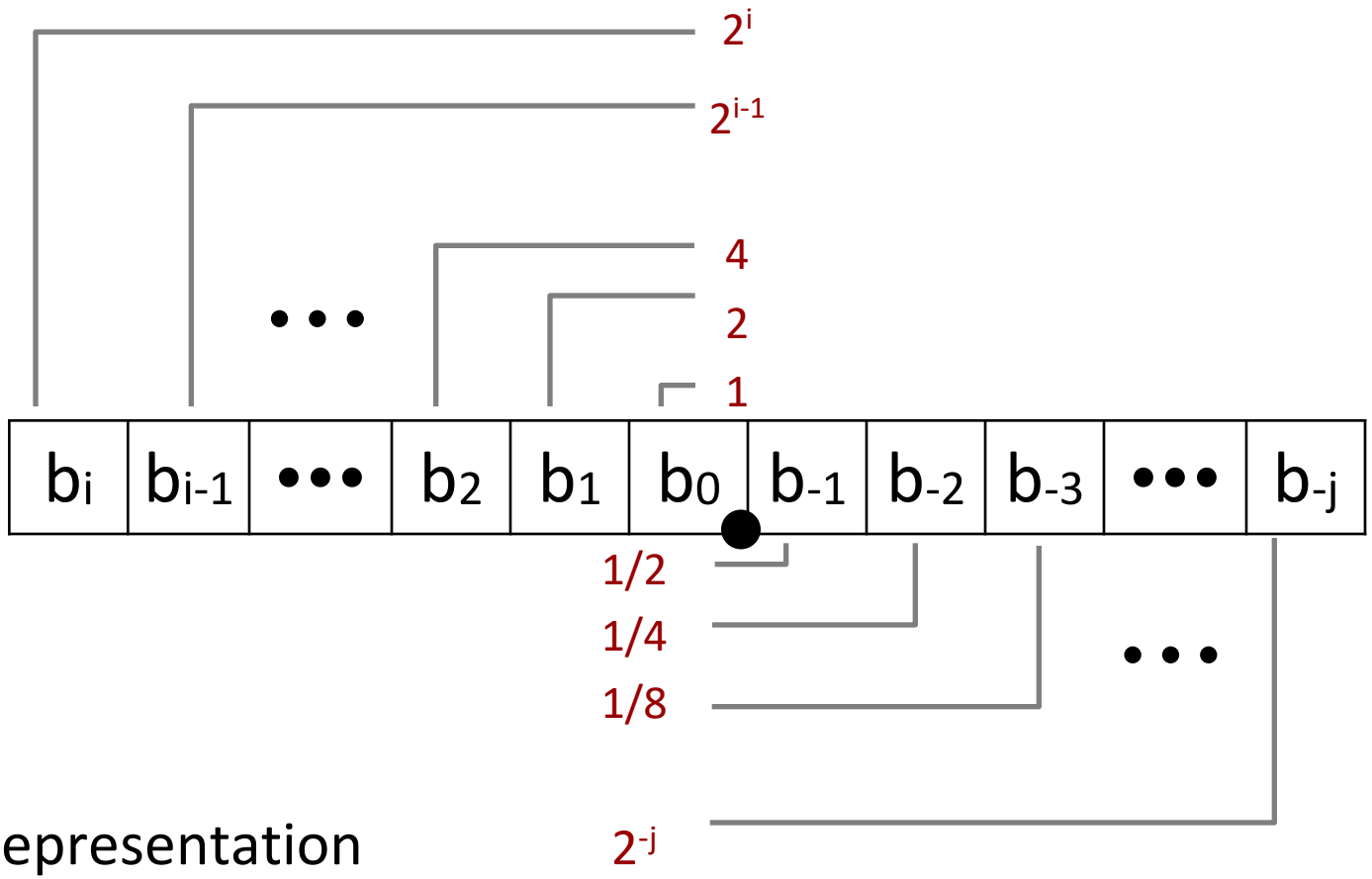


只要对尾数和指数分别编码，就可表示一个浮点数（即：实数）

Fractional binary numbers

- What is 1011.101_2?

Fractional Binary Numbers



- Representation
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number:

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

Fractional Binary Numbers: Examples

- Value Representation

5 3/4	= 23/4	$\mathtt{101.11}_2$	= 4 + 1 + 1/2 + 1/4
2 7/8	= 23/8	$\mathtt{10.111}_2$	= 2 + 1/2 + 1/4 + 1/8
1 7/16	= 23/16	$\mathtt{1.0111}_2$	= 1 + 1/4 + 1/8 + 1/16

- Observations
  - Divide by 2 by shifting right (unsigned)
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

Representable Numbers

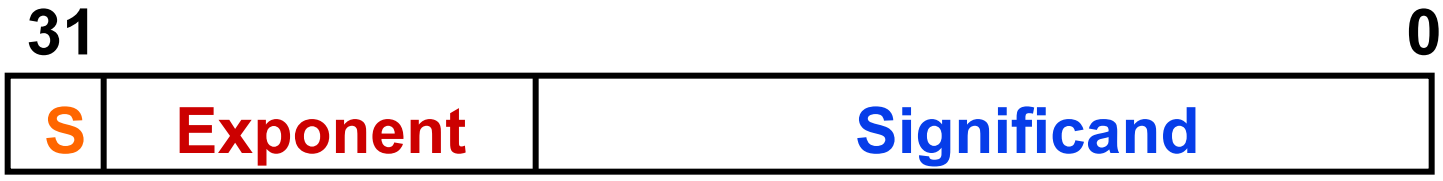
- Limitation #1
  - Can only exactly represent numbers of the form $x/2^k$
    - Other rational numbers have repeating bit representations
  - Value Representation
    - 1/3 $0.0101010101[01]\ldots_2$
    - 1/5 $0.00110011001[0011]\ldots_2$
    - 1/10 $0.000110011001[0011]\ldots_2$
- Limitation #2
  - Just one setting of binary point within the *w* bits
    - Limited range of numbers (very small values? very large?)

浮点数的表示

- Normal format（规格化数形式）：

  +/-1.xxxxxxxxxx × $R^{Exponent}$

- 32-bit 规格化数：



S 是符号位（Sign）

Exponent用移码（增码）来表示

Significand 表示 xxxxxxxxxxxxx，尾数部分

(基可以是 2/ 4 / 8 / 16，约定信息，无需显式表示)

- 早期的计算机，各自定义自己的浮点数格式

  问题：浮点数表示不统一会带来什么问题？

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

“Father” of the IEEE 754 standard

直到80年代初，各个机器内部的浮点数表示格式还没有统一
因而相互不兼容，机器之间传送数据时，带来麻烦

1970年代后期，IEEE成立委员会着手制定浮点数标准

1985年完成浮点数标准IEEE 754的制定

This standard was primarily the work of one person, UC Berkeley math professor William Kahan.

www.cs.berkeley.edu/~wkahan/ieee754status/754story.html

Prof. William Kahan

IEEE Floating Point

- IEEE Standard 754
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs
  - Some CPUs don't implement IEEE 754 in full
    e.g., early GPUs, Cell BE processor

- Driven by numerical concerns
  - Nice standards for rounding, overflow, underflow
  - Hard to make fast in hardware
    - Numerical analysts predominated over hardware designers
      in defining standard

Floating Point Representation

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

- Numerical Form:

$$(-1)^s \, M \;\; 2^E$$

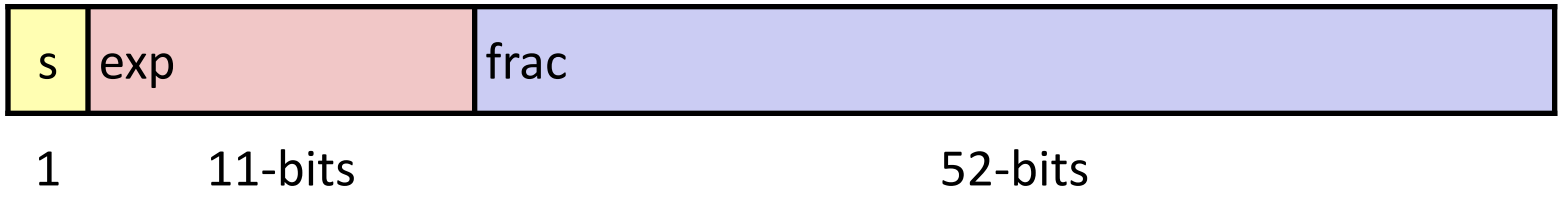
  - Sign bit s determines whether number is negative or positive
  - Significand M normally a fractional value in range [1.0,2.0).
  - Exponent E weights value by power of two

- Encoding
  - MSB s is sign bit s
  - exp field encodes E (but is not equal to E)
  - frac field encodes M (but is not equal to M)

Precision options

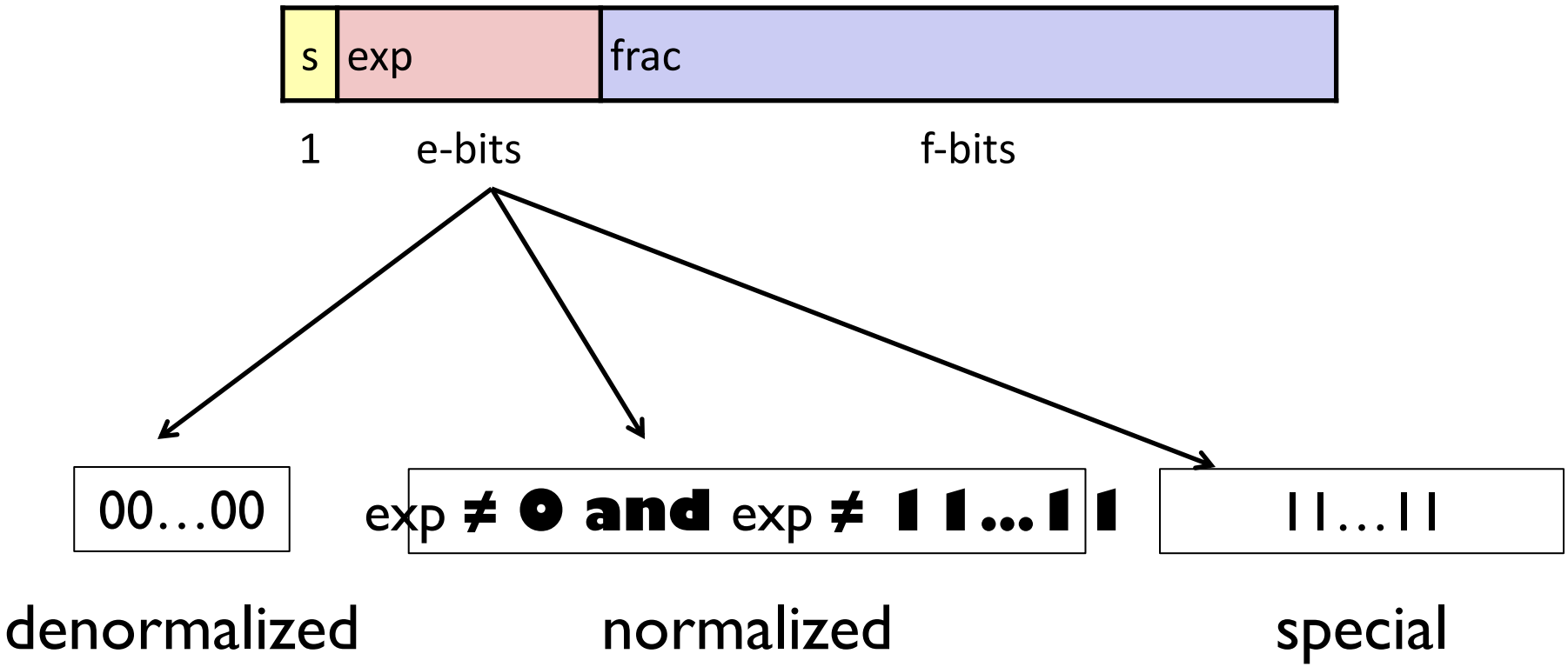
- Single precision: 32 bits
  ≈ 7 decimal digits, $10^{\pm 38}$

- Double precision: 64 bits
  ≈ 16 decimal digits, $10^{\pm 308}$



- Other formats: half precision, quad precision

Three "kinds" of floating point numbers



"Normalized" Values

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

- When: **exp** ≠ 000…0 and **exp** ≠ 111…1

- Exponent coded as a biased value: E = **exp** – Bias
  - exp: unsigned value of exp field
  - Bias = $2^{k-1} - 1$, where k is number of exponent bits
    - Single precision: 127 (**exp**: 1…254, E: -126…127)
    - Double precision: 1023 (**exp**: 1…2046, E: -1022…1023)

- Significand coded with implied leading 1: M = $1.xxx…x_2$
  - xxx…x: bits of frac field
  - Minimum when **frac**=000…0 (M = 1.0)
  - Maximum when **frac**=111…1 (M = 2.0 – ε)
  - Get extra leading bit for "free"

Normalized Encoding Example

$v = (-1)^s M 2^E$
$E = \mathbf{exp} - \text{Bias}$

- **Value:** `float F = 15213.0;`
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$

- **Significand**

  M = $1.\underline{1101101101101}_2$

  frac = $\underline{1101101101101}0000000000_2$

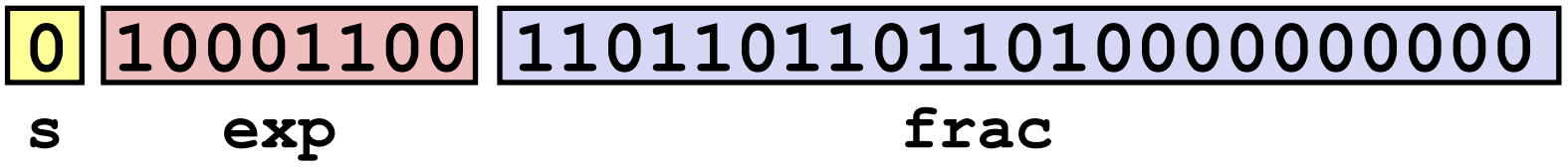
- **Exponent**

  E = 13

  Bias = 127

  exp = 140 = 10001100_2

- **Result:**



Ex: Converting Binary FP to Decimal

BEE00000H is the hex. Rep. Of an IEEE 754 SP FP number

1	011 11101	110 0000 0000 0000 0000 0000

$$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$$

- **Sign**: 1 => negative
- **Exponent**:
  - $0111\ 1101_{two} = 125_{ten}$
  - Bias adjustment: 125 - 127 = -2
- **Significand**:

$$1 + 1\times2^{-1} + 1\times2^{-2} + 0\times2^{-3} + 0\times2^{-4} + 0\times2^{-5} + \ldots$$

$$= 1 + 2^{-1} + 2^{-2} = 1 + 0.5 + 0.25 = 1.75$$

- **Represents**: $-1.75_{ten} \times 2^{-2} = -0.4375$

Ex: Converting Decimal to FP

-12.75

1. Denormalize: -12.75
2. Convert integer part:

   $12 = 8 + 4 = 1100_2$
3. Convert fractional part:

   $.75 = .5 + .25 = .11_2$
4. Put parts together and normalize:

   $1100.11 = 1.10011 \times 2^3$
5. Convert exponent: $127 + 3 = 128 + 2 = 1000\ 0010_2$

1	1000 0010	100 1100 0000 0000 0000 0000

The Hex rep. is C14C0000H

Normalized numbers（规格化数）

前面的定义都是针对规格化数（normalized form）

How about other patterns?

Exponent	Significand	Object
1-254	anything implicit leading 1	Norms
0	0	?
0	nonzero	?
255	0	?
255	nonzero	?

Denormalized Values

$$v = (-1)^s M 2^E$$
$$E = 1 - \text{Bias}$$

- Condition: exp = 000…0

- Exponent value: E = 1 – Bias (instead of **`exp`** – Bias)
- Significand coded with implied leading 0: M = 0.xxx…x_2
  - **`xxx`**…**`x`**: bits of **`frac`**
- Cases
  - **`exp`** = 000…0, **`frac`** = 000…0
    - Represents zero value
    - Note distinct values: +0 and –0 (表示不同含义)
  - **`exp`** = 000…0, **`frac`** ≠ 000…0
    - Numbers closest to 0.0
    - Equispaced （可能的数值分布均匀地接近0）

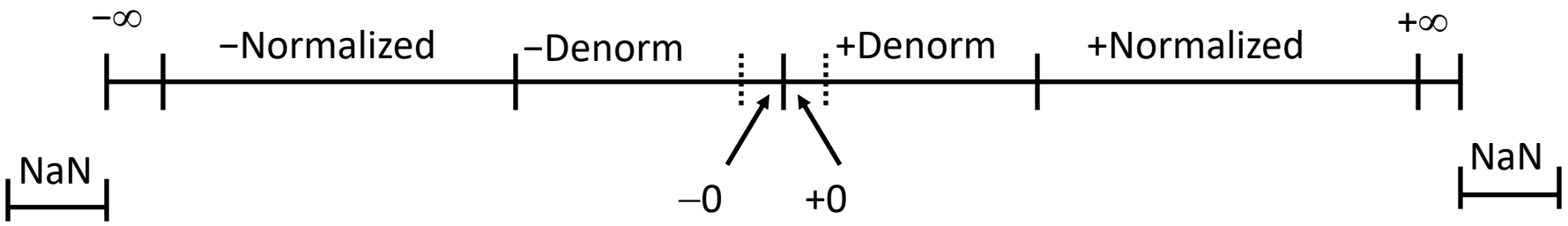
Special Values

- Condition: **`exp`** = **`111…1`**

- Case: **`exp`** = **`111…1`**, **`frac`** = **`000…0`**
  - **Represents value ∞ (infinity)**
  - Operation that overflows
  - Both positive and negative
  - E.g., 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞

- Case: **`exp`** = **`111…1`**, **`frac`** ≠ **`000…0`**
  - **Not-a-Number (NaN)**
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), ∞ − ∞, ∞ × 0

Visualization: Floating Point Encodings



C float Decoding Example

$$v = (-1)^s M 2^E$$
$$E = \mathbf{exp} - \text{Bias}$$

float: **0xC0A00000**

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

binary:

1 8-bits 23-bits

E =

S =

M =

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

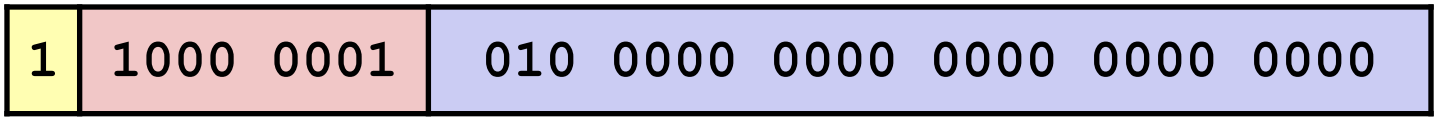
Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

C float Decoding Example

$v = (-1)^s M 2^E$
$E = \mathbf{exp} - \text{Bias}$

float: **0xC0A00000**

binary: **1100 0000 1010 0000 0000 0000 0000 0000**



1 8-bits 23-bits

E =

S =

M = 1 .

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

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

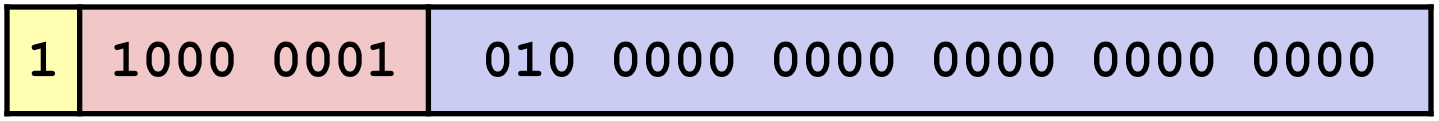
C float Decoding Example

$$v = (-1)^s M 2^E$$
$$E = \texttt{exp} - \text{Bias}$$

float: **0xC0A00000**

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

binary: **1100 0000 1010 0000 0000 0000 0000 0000**



E = **exp** – Bias = 129 – 127 = 2 (decimal)

S = 1 -> negative number

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

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

Floating Point in C

- C Guarantees Two Levels
  - **`float`** single precision
  - **`double`** double precision
- Conversions/Casting
  - Casting between **`int`**, **`float`**, and **`double`** changes bit representation
  - **`double`**/**`float`** → **`int`**
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to TMin
  - **`int`** → **`double`**
    - Exact conversion, as long as **`int`** has ≤ 53 bit word size
  - **`int`** → **`float`**
    - Will round according to rounding mode

Floating Point Puzzles

- For each of the following C expressions, either:
  - Argue that it is true for all argument values
  - Explain why not true

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

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);` ✓

C语言中的浮点数类型-32位

- C语言中有float和double类型，分别对应IEEE 754单精度浮点数格式和双精度浮点数格式
- long double类型的长度和格式随编译器和处理器类型的不同而有所不同，IA-32中是80位扩展精度格式
- 从int转换为float时，不会发生溢出，但可能有数据被舍入
- 从int或 float转换为double时，因为double的有效位数更多，故能保留精确值
- 从double转换为float和int时，可能发生溢出，此外，由于有效位数变少，故可能被舍入
- 从float 或double转换为int时，因为int没有小数部分，所以数据可能会向0方向被截断

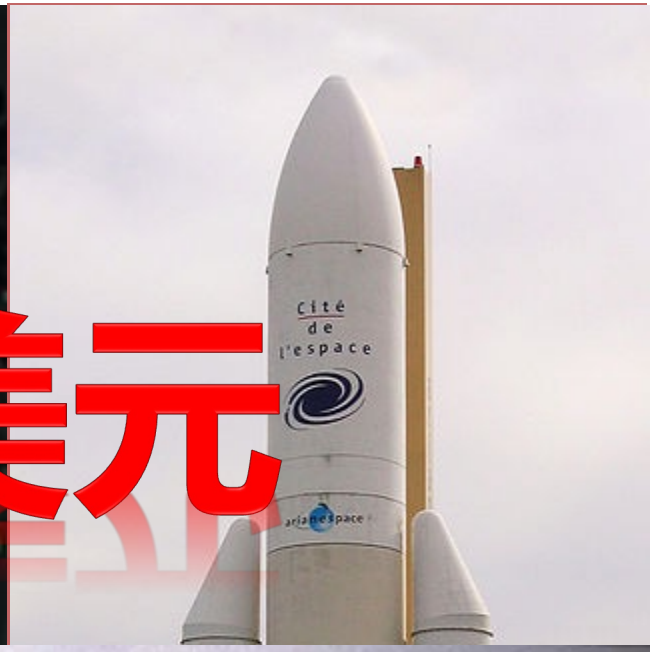
浮点运算举例-1

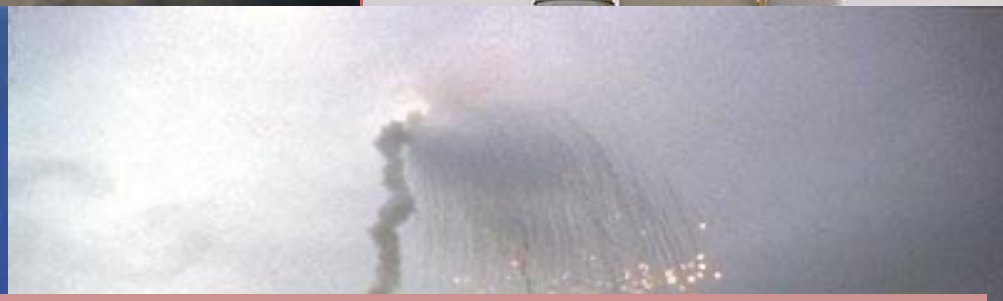
欧洲阿丽亚娜5火箭

37秒

10年

70亿美元





故障原因：64位→16位舍入错误

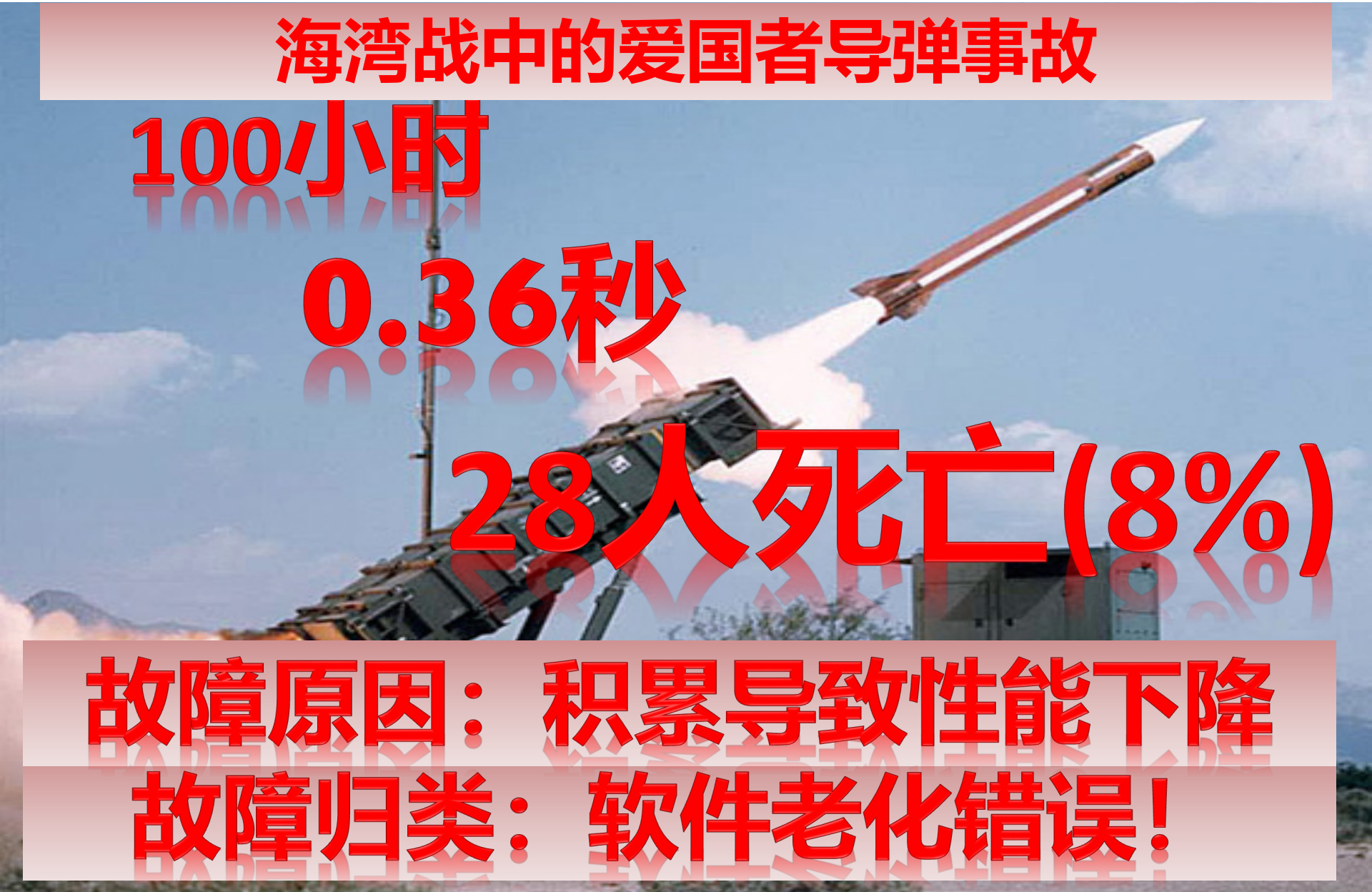
故障归类：软件设计/重用错误！

浮点运算举例

- 1996年6月4日，Ariana 5火箭初次航行，在发射仅仅37秒钟后，偏离了飞行路线，然后解体爆炸，火箭上载有价值5亿美元的通信卫星。
- 原因是在将一个64位浮点数转换为16位带符号整数时，产生了溢出异常。溢出的值是火箭的水平速率，这比原来的Ariana 4火箭所能达到的速率高出了5倍。在设计Ariana 4火箭软件时，设计者确认水平速率决不会超出一个16位的整数，但在设计Ariana 5时，他们没有重新检查这部分，而是直接使用了原来的设计。
- 在不同数据类型之间转换时，往往隐藏着一些不容易被察觉的错误，这种错误有时会带来重大损失，因此，编程时要非常小心。

浮点运算举例-2

海湾战中的爱国者导弹事故



100小时

0.36秒

28人死亡(8%)

故障原因：积累导致性能下降

故障归类：软件老化错误！

浮点运算举例

- 1991年2月25日，海湾战争中，美国在沙特阿拉伯达摩地区设置的爱国者导弹拦截伊拉克的飞毛腿导弹失败，致使飞毛腿导弹击中了沙特阿拉伯载赫蓝的一个美军军营，杀死了美国陆军第十四军需分队的28名士兵。其原因是由于爱国者导弹系统时钟内的一个软件错误造成的，引起这个软件错误的原因是浮点数的精度问题。
- 爱国者导弹系统中有一个内置时钟，用计数器实现，每隔0.1秒计数一次。程序用0.1的一个24位定点二进制小数x来乘以计数值作为以秒为单位的时间。
- 0.1的二进制表示是一个无限循环序列：0.00011[0011]...，x=0.000 1100 1100 1100 1100 1100B。显然，x是0.1的近似表示，0.1-x

  = 0.000 1100 1100 1100 1100 1100 [1100]... -

  0.000 1100 1100 1100 1100 1100B，即为：

  =0.000 0000 0000 0000 0000 0000 1100 [1100]...B

  $=2^{-20}\times 0.1 \approx 9.54\times 10^{-8}$

浮点运算举例

已知在爱国者导弹准备拦截飞毛腿导弹之前，已经连续工作了100小时，飞毛腿的速度大约为2000米/秒，则由于时钟计算误差而导致的距离误差是多少?

100小时相当于计数了$100\times60\times60\times10=36\times10^5$次，因而导弹的时钟已经偏差了$9.54\times10^{-8}\times36\times10^5\approx0.343$秒

因此，距离误差是2000×0.343秒 $\approx$ 687米

小故事：实际上，以色列方面已经发现了这个问题并于1991年2月11日知会了美国陆军及爱国者计划办公室（软件制造商）。以色列方面建议重新启动爱国者系统的电脑作为暂时解决方案，可是美国陆军方面却不知道每次需要间隔多少时间重新启动系统一次。1991年2月16日，制造商向美国陆军提供了更新软件，但这个软件最终却在飞毛腿导弹击中军营后的一天才运抵部队。

浮点运算举例

- 若x用float型表示，则x的机器数是什么？0.1与x的偏差是多少？系统运行100小时后的时钟偏差是多少？在飞毛腿速度为2000米/秒的情况下，预测的距离偏差为多少？
  - 0.1= 0.0 0011[0011]B=+1.1 0011 0011 0011 0011 0011 00B×2^{-4}，故x的机器数为0 011 1101 1 100 1100 1100 1100 1100 1100
  - Float型仅24位有效位数，后面的有效位全被截断，故x与0.1之间的误差为：|x–0.1|=0.000 0000 0000 0000 0000 0000 0000 1100 [1100]...B。这个值等于$2^{-24}\times0.1 \approx 5.96\times10^{-9}$。100小时后时钟偏差$5.96\times10^{-9}\times36\times10^{5} \approx 0.0215$秒。距离偏差$0.0215\times2000\approx43$米。比爱国者导弹系统精确约16倍。
- 若用32位二进制定点小数x=0.000 1100 1100 1100 1100 1100 1100 1101 B表示0.1，则误差比用float表示误差更大还是更小？
  - 当x=0.000 1100 1100 1100 1100 1100 1100 1101 B时，与0.1之间的误差约为：|x–0.1|=0.000 0000 0000 0000 0000 0000 0000 0000 00 1100 [1100]...B。这个值等于$2^{-30}\times0.1 \approx 9.31\times10^{-11}$。100小时后时钟偏差$9.31\times10^{-11}\times36\times10^{5} \approx 0.000335$秒。预测的距离偏差仅为$0.000335\times2000 \approx 0.67$米。

爱国者导弹采用全向爆炸破片弹头！

浮点运算举例

- 从上述结果可以看出：
  - 用32位定点小数表示0.1，比采用float精度高64倍
  - 用float表示在计算速度上更慢，必须先把计数值转换为IEEE 754格式浮点数，然后再对两个IEEE 754格式的数相乘，故采用float比直接将两个二进制数相乘要慢得多
- Ariana 5火箭和爱国者导弹的例子带来的启示

✓ 程序员应对底层机器级数据的表示和运算有深刻理解

✓ 计算机世界里，经常是“差之毫厘，失之千里”，需要细心再细心，精确再精确

✓ 不能遇到小数就用浮点数表示，有些情况下（如需要将一个整数变量乘以一个确定的小数常量），可先用一个确定的定点整数与整数变量相乘，然后再通过移位运算来确定小数点

Summary

- IEEE Floating Point has clear mathematical properties
- Represents numbers of form M x 2^E
- One can reason about operations independent of implementation
  - As if computed with perfect precision and then rounded
- Not the same as real arithmetic
  - Violates associativity/distributivity
  - Makes life difficult for compilers & serious numerical applications programmers