计算机系统基础04

Floating Point

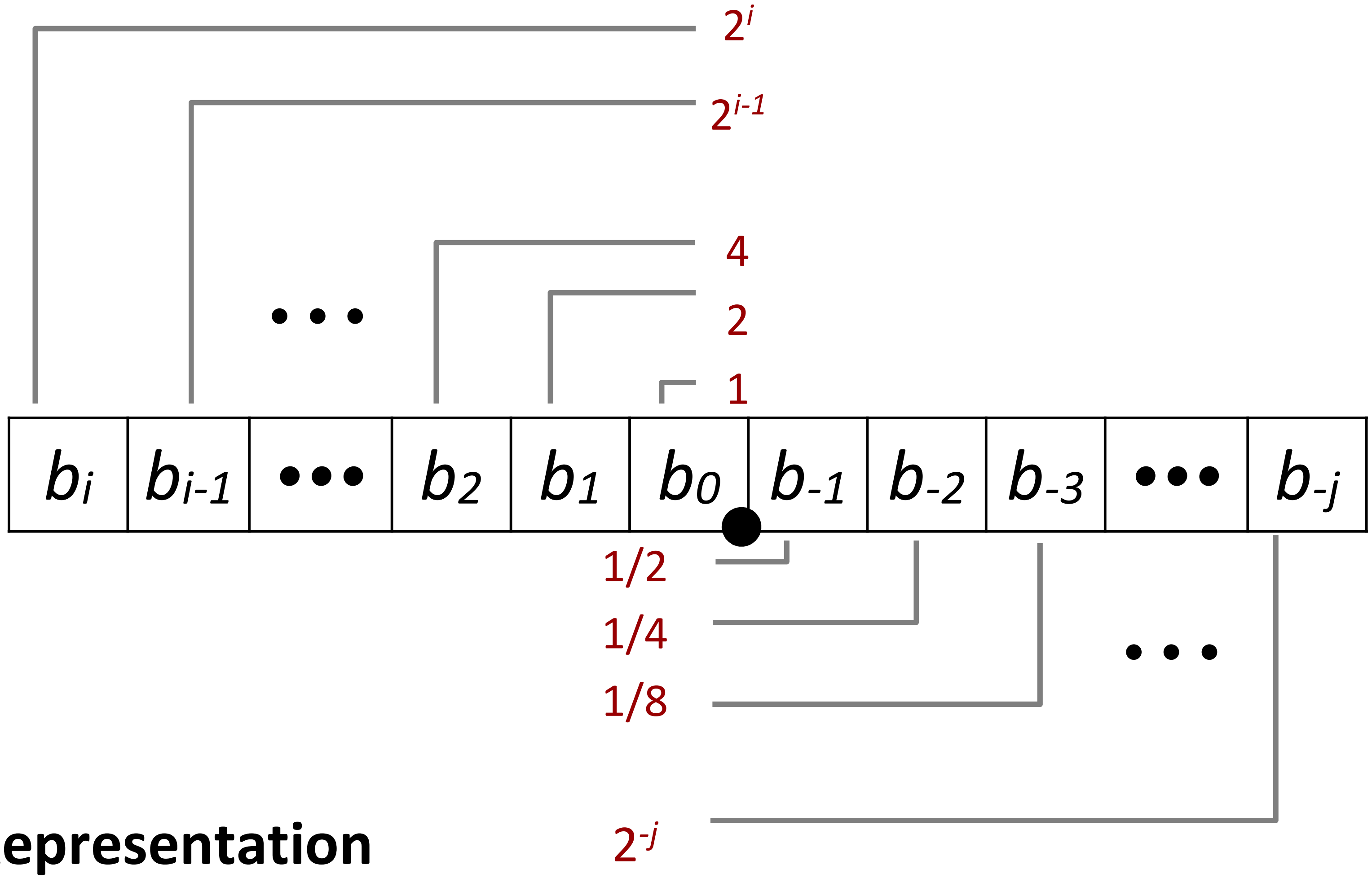
Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Floating point in C
- Summary

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]`…$_2$
    - 1/5 `0.001100110011[0011]`…$_2$
    - 1/10 `0.0001100110011[0011]`…$_2$
- **Limitation #2**
  - Just one setting of binary point within the *w* bits
    - Limited range of numbers (very small values? very large?)

Today: Floating Point

- Background: Fractional binary numbers
- **IEEE floating point standard: Definition**
- Floating point in C
- Summary

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} = $(-1)^0$ x 1.1101101101101_2 x 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, $10^{\pm 38}$

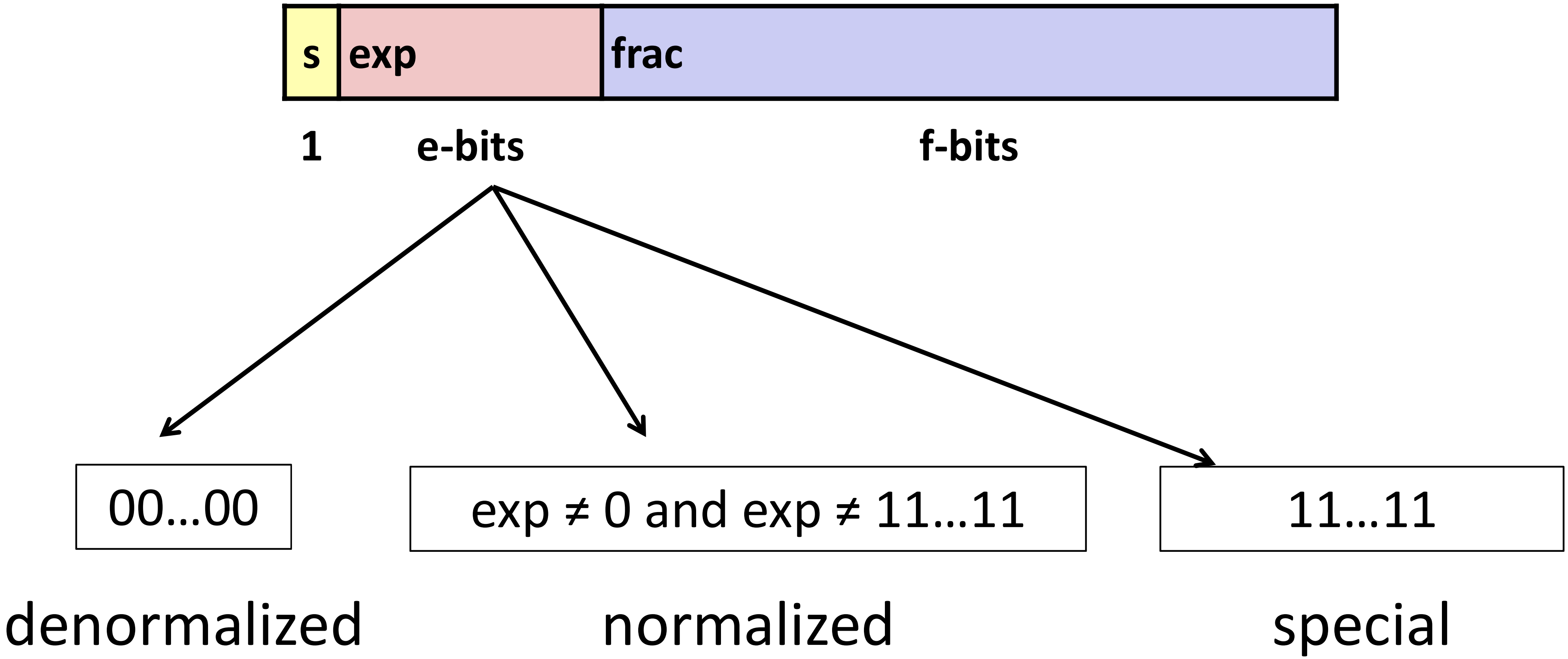
s	exp	frac
1	8-bits	23-bits

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

s	exp	frac
1	11-bits	52-bits

- **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 = $1.xxx\ldots x_2$**
  - 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

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

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

- **Significand**

  $M = 1.\underline{1101101101101}_2$
  $\texttt{frac} = \underline{1101101101101}0000000000_2$

- **Exponent**

  $E = 13$
  $Bias = 127$
  $\texttt{exp} = 140 = 10001100_2$

- **Result:**

0	10001100	11011011011010000000000
s	exp	frac

Denormalized Values

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

$E = 1 - Bias$

- **Condition:** exp = 000…0

- **Exponent value:** E **= 1 – Bias (instead of** `exp` **–** ***Bias*****)** (why?)
- **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 (why?)
  - `exp` = 000…0, `frac` ≠ 000…0
    - Numbers closest to 0.0
    - Equispaced

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., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$

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

C float Decoding Example

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

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

float: `0xC0A00000`

binary: ____ ____ ____ ____ ____ ____ ____ ____

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

E =

S =

M =

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

C float Decoding Example

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

float: `0xC0A00000`

binary: 1100 0000 1010 0000 0000 0000 0000 0000

1	1000 0001	010 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

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

float: 0xC0A00000

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

binary: 1100 0000 1010 0000 0000 0000 0000 0000

1	1000 0001	010 0000 0000 0000 0000 0000
1	8-bits	23-bits

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

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

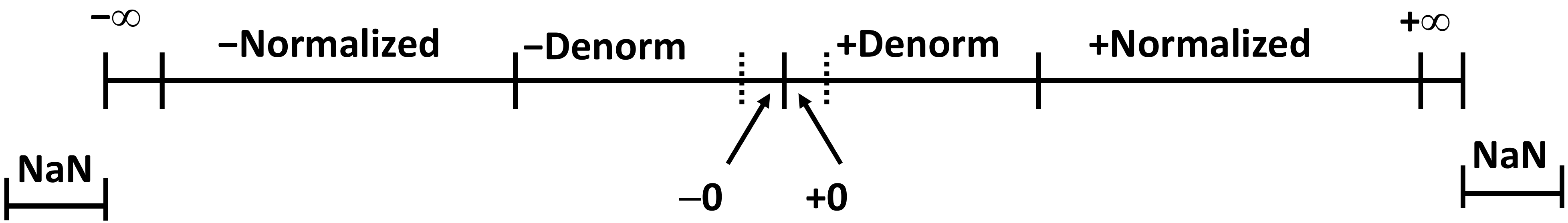
S = 1 -> negative number

M = 1.010 0000 0000 0000 0000 0000

= 1 + 1/4 = 1.25

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

Visualization: Floating Point Encodings



Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- **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

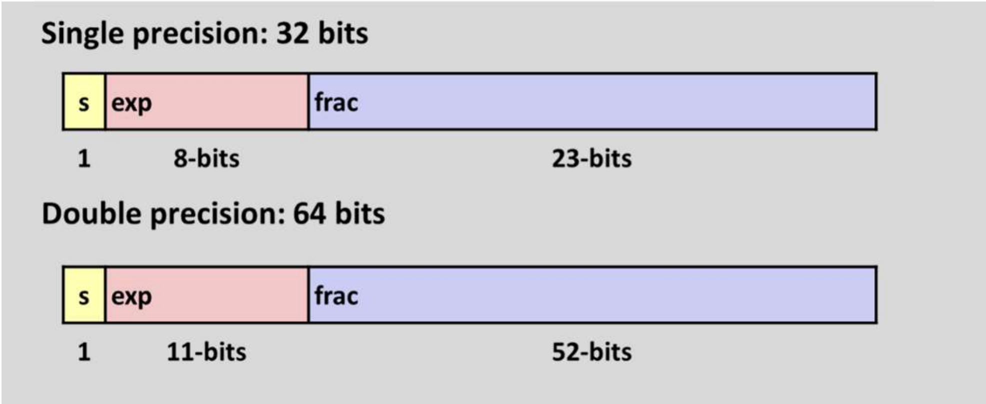
```
int x = …;
float f = …;
double d = …;
```

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);` ✔
- `2/3 == 2/3.0` ✘
- `d < 0.0` ⇒ `((d*2) < 0.0)` ✔
- `d > f` ⇒ `-f > -d` ✔
- `d * d >= 0.0` ✔
- `(d+f)-d == f` ✘

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



教材阅读

- 第2章 2.4-2.4.2、 2.4.6