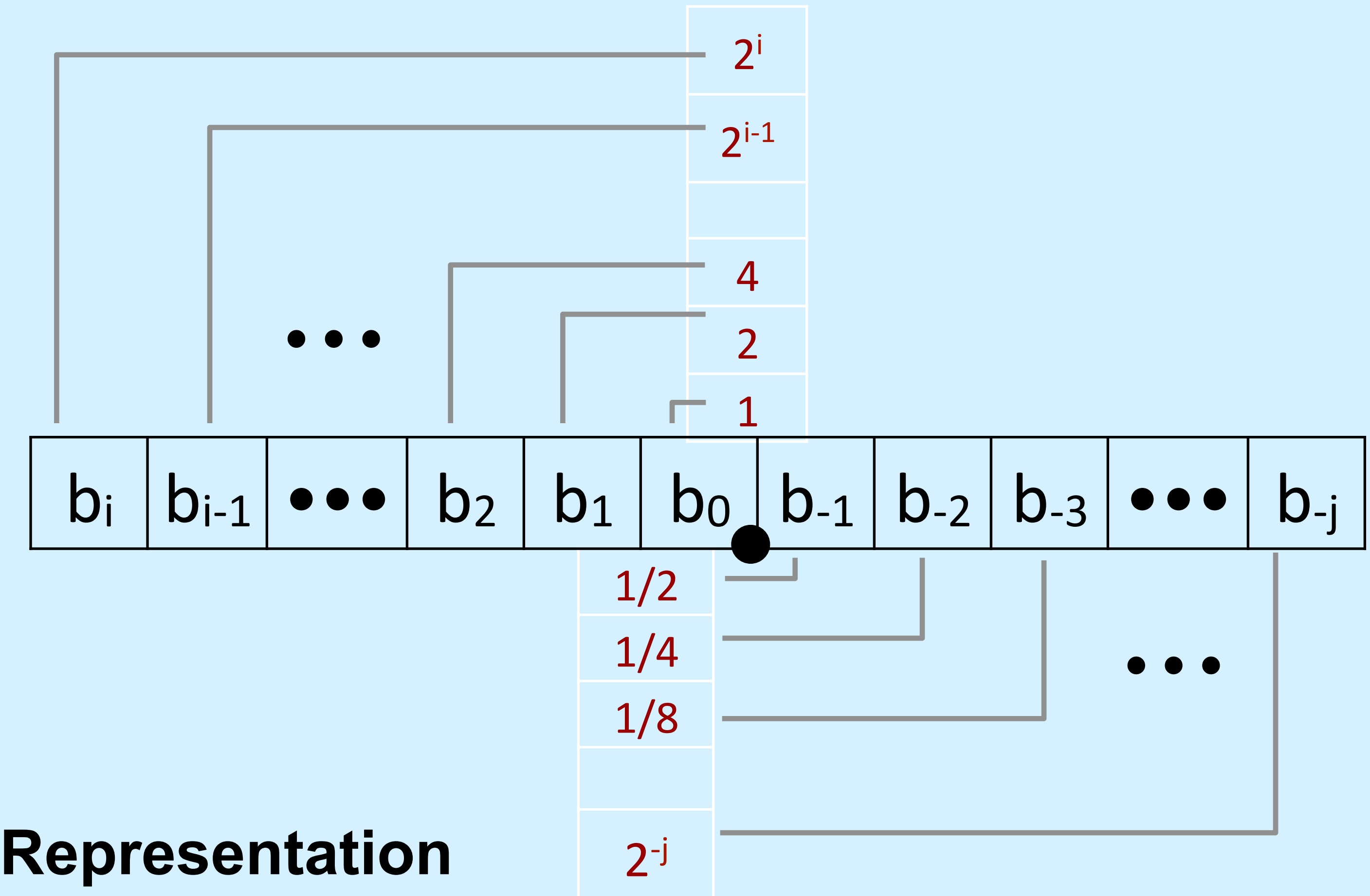
CS 33

Data Representation (Part 2)

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

Representable Numbers

- **Limitation #1**
  - can exactly represent only numbers of the form $n/2^k$
    - » 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 decimal point within the *w* bits
    - » limited range of numbers (very small values? very large?)

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

- 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

- **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)

s	exp	frac

Precision options

- **Single precision: 32 bits**

s	exp	frac
1	8-bits	23-bits

- **Double precision: 64 bits**

s	exp	frac
1	11-bits	52-bits

- **Extended precision: 80 bits (Intel only)**

s	exp	frac
1	15-bits	64-bits

“Normalized” Values

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

- **Exponent coded as biased value: E = Exp – Bias**
  - **exp: unsigned value exp**
  - **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**
  - **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

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

- **Significand**

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

  `frac` = $\underline{\texttt{1101101101101}}\texttt{0000000000}_2$

- **Exponent**

  E = **13**

  bias = **127**

  exp = **140** = $\texttt{10001100}_2$

- **Result:**

0	10001100	11011011011010000000000
s	exp	frac

Denormalized Values

- Condition: exp = `000…0`
- Exponent value: E = –Bias + 1 (instead of E = 0 – Bias)
- Significand coded with implied leading 0:
  $M = 0.xxx\ldots x_2$
  - `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

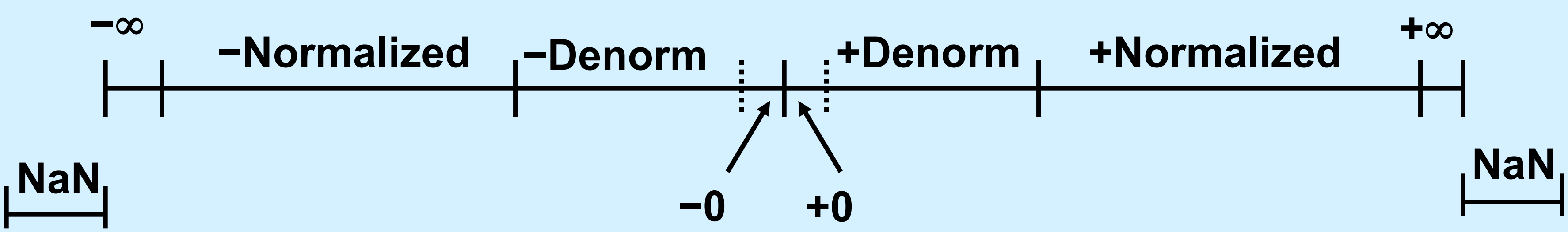
Special Values

- 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 = +∞, 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



Tiny Floating-Point Example

s	exp	frac
1	4-bits	3-bits

- **8-bit Floating Point Representation**
  - the sign bit is in the most significant bit
  - the next four bits are the exponent, with a bias of 7
  - the last three bits are the `frac`

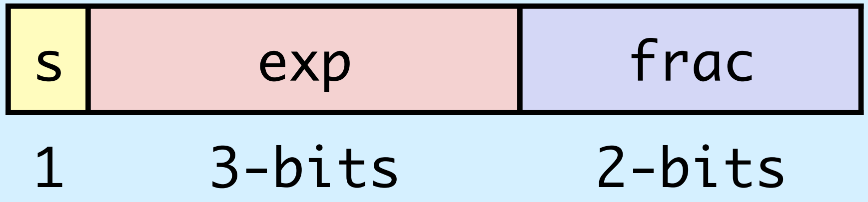
- **Same general form as IEEE Format**
  - normalized, denormalized
  - representation of 0, NaN, infinity

Dynamic Range (Positive Only)

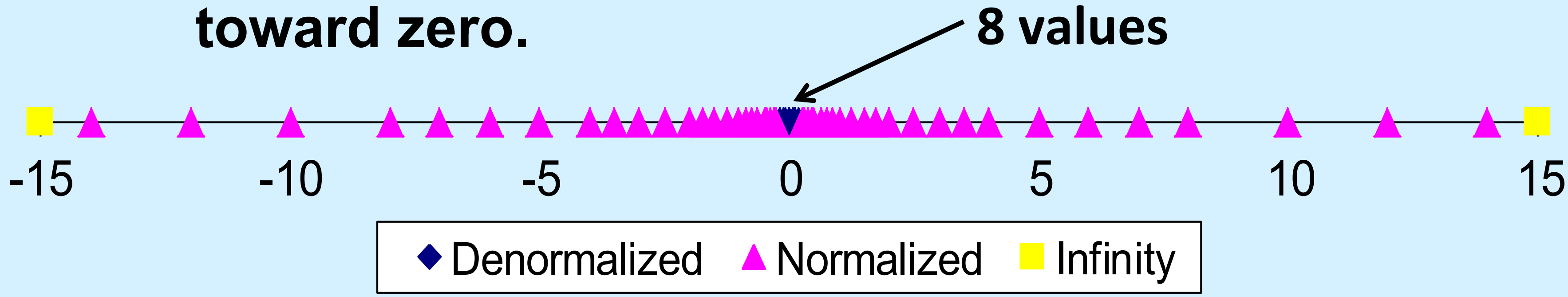
	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	largest denorm
Normalized numbers	0	0001	000	-6	8/8*1/64 = 8/512	smallest norm
	0	0001	001	-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	closest to 1 below
	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	closest to 1 above
	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	largest norm
	0	1111	000	n/a	inf	

Distribution of Values

- **6-bit IEEE-like format**
  - **e = 3 exponent bits**
  - **f = 2 fraction bits**
  - **bias is $2^{3-1}-1 = 3$**

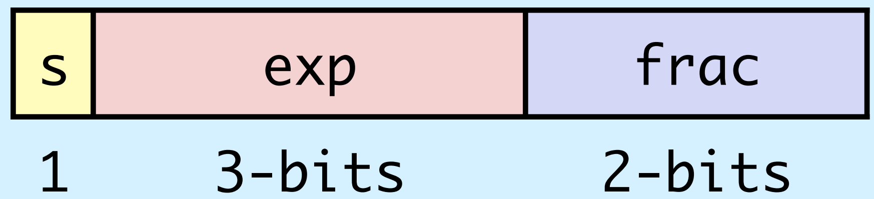


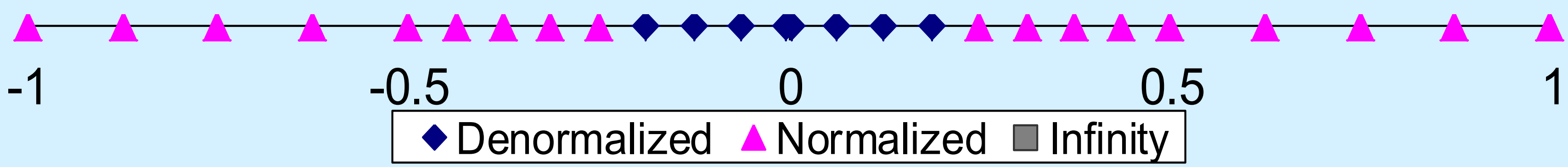
- **Notice how the distribution gets denser toward zero.**



Distribution of Values (close-up view)

- **6-bit IEEE-like format**
  - **e = 3 exponent bits**
  - **f = 2 fraction bits**
  - **bias is 3**





Quiz 1

- **6-bit IEEE-like format**
  - e = 3 exponent bits
  - f = 2 fraction bits
  - bias is 3

s	exp	frac
1	3-bits	2-bits

What number is represented by 0 011 10?

a) 12
b) 1.5
c) .5
d) none of the above

Floating-Point Operations: Basic Idea

- `x +f y = Round(x + y)` ($x +_f y = \mathrm{Round}(x + y)$)

- $x \times_f y = \mathrm{Round}(x \times y)$

- **Basic idea**
  - first **compute exact result**
  - make it fit into desired precision
    - » possibly overflow if exponent too large
    - » possibly **round to fit into** `frac`

Rounding

- **Rounding modes (illustrated with $ rounding)**

	$1.40	$1.60	$1.50	$2.50	–$1.50
towards zero	$1	$1	$1	$2	–$1
round down (−∞)	$1	$1	$1	$2	–$2
round up (+∞)	$2	$2	$2	$3	–$1
nearest even (default)	$1	$2	$2	$2	–$2

Closer Look at Round-To-Nearest-Even

- **Default rounding mode**
  - hard to get any other kind without dropping into assembly
  - all others are statistically biased
    - » sum of set of positive numbers will consistently be over- or under-estimated

- **Applying to other decimal places / bit positions**
  - when exactly halfway between two possible values
    - » round so that least significant digit is even
  - e.g., round to nearest hundredth

1.2349999	1.23	(less than half way)
1.2350001	1.24	(greater than half way)
1.2350000	1.24	(half way—round up)
1.2450000	1.24	(half way—round down)

Rounding Binary Numbers

- **Binary fractional numbers**
  - "even" when least significant bit is `0`
  - "half way" when bits to right of rounding position = $100..._2$

- **Examples**
  - round to nearest 1/4 (2 bits right of binary point)

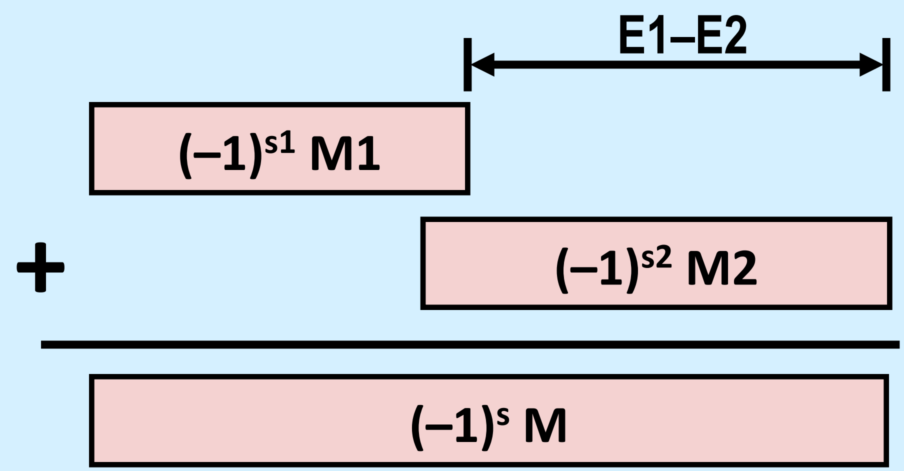
Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2

Floating-Point Multiplication

- $(-1)^{s1}\ M1\ \ 2^{E1}\quad x\quad (-1)^{s2}\ M2\ \ 2^{E2}$
- Exact result: $(-1)^{s}\ M\ \ 2^{E}$
  - sign s: s1 ^ s2
  - significand M: M1 x M2
  - exponent E: E1 + E2
- Fixing
  - if M ≥ 2, shift M right, increment E
  - if E out of range, overflow (or underflow)
  - round M to fit `frac` precision
- Implementation
  - biggest chore is multiplying significands

Floating-Point Addition

- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ +\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
  - –assume E1 > E2



- **Exact result:** $(-1)^{s}\ M\ \ 2^{E}$
  - –sign s, significand M:
    - » result of signed align & add
  - –exponent E: E1

- **Fixing**
  - –if M ≥ 2, shift M right, increment E
  - –if M < 1, shift M left k positions, decrement E by k
  - –overflow if E out of range
  - –round M to fit `frac` precision

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

Creating Floating-Point Numbers

s	exp	frac
1	4-bits	3-bits

- **Steps**
  - normalize to have leading 1
  - round to fit within fraction
  - postnormalize to deal with effects of rounding

- **Case study**
  - convert 8-bit unsigned numbers to tiny floating point format

example numbers

```
128     10000000
 13     00001101
 33     00010001
 35     00010011
138     10001010
 63     00111111
```

Normalize

s	exp	frac
1	4-bits	3-bits

- **Requirement**
  - set binary point so that numbers of form 1.xxxxx
  - adjust all to have leading one
    - » decrement exponent as shift left

Value	Binary	Fraction	Exponent
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

Rounding

1.BBGRXXX

Guard bit: LSB of result

Round bit: 1st bit removed

Sticky bit: OR of remaining bits

- **Round-up conditions**
  - round = 1, sticky = 1 ⇒ > 0.5
  - guard = 1, round = 1, sticky = 0 ⇒ round to even

Value	Fraction	GRS	Incr?	Rounded
128	1.0000000	000	N	1.000
13	1.1010000	100	N	1.101
17	1.0001000	010	N	1.000
19	1.0011000	110	Y	1.010
138	1.0001010	011	Y	1.001
63	1.1111100	111	Y	10.000

Postnormalize

- **Issue**
  - rounding may have caused overflow
  - handle by shifting right once & incrementing exponent

Value	Rounded	Exp	Adjusted	Result
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	$1.000*2^6$	64

Quiz 2

Suppose *f*, declared to be a `float`, is assigned the largest possible floating-point positive value (other than $+\infty$). What is the value of $g = f+1.0$?

a) f

b) $+\infty$

c) NAN

d) 0

Float is not Rational …

- **Floating addition**
  - **commutative: $a +^f b = b +^f a$**
    - » **yes!**
  - **associative: $a +^f (b +^f c) = (a +^f b) +^f c$**
    - » **no!**
      - **$2 +^f (1e10 +^f -1e10) = 2$**
      - **$(2 +^f 1e10) +^f -1e10 = 0$**

Float is not Rational …

- **Multiplication**
  - commutative: $a *^f b = b *^f a$
    - » yes!
  - associative: $a *^f (b *^f c) = (a *^f b) *^f c$
    - » no!
      - $1e20 *^f (1e20 *^f 1e\text{-}20) = 1e20$
      - $(1e20 *^f 1e20) *^f 1e\text{-}20 = +\infty$

Float is not Rational …

- More …
  - multiplication distributes over addition:
    $a *^f (b +^f c) = (a *^f b) +^f (a *^f c)$
    - » no!
    - » $1e20 *^f (1e20 +^f -1e20) = 0$
    - » $(1e20 *^f 1e20) +^f (1e20 *^f -1e20) = NaN$
  - loss of significance:
    x=y+1
    z=2/(x-y)
    z==2?
    - » not necessarily!
      - consider y = 1e20